```
**********************************************************
    4180 Tue Jan 14 16:49:30 2014
new/usr/src/cmd/dtrace/test/tst/common/Makefile
4477 DTrace should speak JSON
Reviewed by: Bryan Cantrill <bmc@joyent.com>
**********************************************************
```

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

#
# Copyright 2008 Sun Microsystems, Inc.  All rights reserved.
# Use is subject to license terms.
#

#
# Copyright (c) 2012 by Delphix. All rights reserved.
# Copyright (c) 2012, Joyent, Inc. All rights reserved.
#endif /* ! codereview */
#

include $(SRC)/Makefile.master
include ../Makefile.com

SNOOPDIR = $(SRC)/cmd/cmd-inet/usr.sbin/snoop
SNOOPOBJS = nfs4_xdr.o
SNOOPSRCS = ${SNOOPOBJS:%.o=%.c}
CLOBBERFILES += nfs/$(SNOOPOBJS)

RPCSVCDIR = $(SRC)/head/rpcsvc
RPCSVCOBJS = nfs_prot.o
RPCSVCSRCS = ${RPCSVCOBJS:%o=%c}
CLOBBERFILES += nfs/$(RPCSVCOBJS) $(RPCSVCDIR)/$(RPCSVCSRCS)
CLOBBERFILES += usdt/forker.h usdt/lazyprobe.h

fasttrap/tst.fasttrap.exe := LDLIBS += -ldtrace
fasttrap/tst.stack.exe := LDLIBS += -ldtrace

sysevent/tst.post.exe := LDLIBS += -lsysevent
sysevent/tst.post_chan.exe := LDLIBS += -lsysevent

ustack/tst.bigstack.exe := COPTFLAG += -xO1

GCC = $(ONBLD_TOOLS)/bin/$(MACH)/cw -_gcc

nfs/%.o: $(SNOOPDIR)/%.c
        $(COMPILE.c) -o $@ $< -I$(SNOOPDIR)
        $(POST_PROCESS_O)
nfs/tst.call.exe: nfs/tst.call.o nfs/$(SNOOPOBJS)
```



```
        $(LINK.c) -o $@ nfs/tst.call.o nfs/$(SNOOPOBJS) $(LDLIBS) -lnsl
        $(POST_PROCESS) ; $(STRIP_STABS)
$(RPCSVCDIR)/%.c: $(RPCSVCDIR)/%.x
        $(RPCGEN) -Cc $< > $@
nfs/$(RPCSVCOBJS): $(RPCSVCDIR)/$(RPCSVCSRCS)
        $(COMPILE.c) -o $@ $(RPCSVCDIR)/$(RPCSVCSRCS)
        $(POST_PROCESS_O)
nfs/tst.call3.exe: nfs/tst.call3.o nfs/$(RPCSVCOBJS)
        $(LINK.c) -o $@ nfs/tst.call3.o nfs/$(RPCSVCOBJS) \
        $(LDLIBS) -lnsl -lrpcsvc
        $(POST_PROCESS) ; $(STRIP_STABS)

pid/tst.gcc.exe: pid/tst.gcc.c
        $(GCC) -o pid/tst.gcc.exe pid/tst.gcc.c $(LDFLAGS)
        $(POST_PROCESS) ; $(STRIP_STABS)

json/tst.usdt.o: json/usdt.h

json/usdt.h: json/usdt.d
        $(DTRACE) -h -s json/usdt.d -o json/usdt.h

json/usdt.o: json/usdt.d json/tst.usdt.o
        $(COMPILE.d) -o json/usdt.o -s json/usdt.d json/tst.usdt.o

json/tst.usdt.exe: json/tst.usdt.o json/usdt.o
        $(LINK.c) -o json/tst.usdt.exe json/tst.usdt.o json/usdt.o $(LDLIBS)
        $(POST_PROCESS) ; $(STRIP_STABS)

#endif /* ! codereview */
usdt/tst.args.exe: usdt/tst.args.o usdt/args.o
        $(LINK.c) -o usdt/tst.args.exe usdt/tst.args.o usdt/args.o $(LDLIBS)
        $(POST_PROCESS) ; $(STRIP_STABS)

usdt/args.o: usdt/args.d usdt/tst.args.o
        $(COMPILE.d) -o usdt/args.o -s usdt/args.d usdt/tst.args.o

usdt/tst.argmap.exe: usdt/tst.argmap.o usdt/argmap.o
        $(LINK.c) -o usdt/tst.argmap.exe \
            usdt/tst.argmap.o usdt/argmap.o $(LDLIBS)
        $(POST_PROCESS) ; $(STRIP_STABS)

usdt/argmap.o: usdt/argmap.d usdt/tst.argmap.o
        $(COMPILE.d) -o usdt/argmap.o -s usdt/argmap.d usdt/tst.argmap.o

usdt/tst.forker.exe: usdt/tst.forker.o usdt/forker.o
        $(LINK.c) -o usdt/tst.forker.exe \
            usdt/tst.forker.o usdt/forker.o $(LDLIBS)
        $(POST_PROCESS) ; $(STRIP_STABS)

usdt/forker.o: usdt/forker.d usdt/tst.forker.o
        $(COMPILE.d) -o usdt/forker.o -s usdt/forker.d usdt/tst.forker.o

usdt/tst.forker.o: usdt/forker.h

usdt/forker.h: usdt/forker.d
        $(DTRACE) -h -s usdt/forker.d -o usdt/forker.h

usdt/tst.lazyprobe.exe: usdt/tst.lazyprobe.o usdt/lazyprobe.o
        $(LINK.c) -o usdt/tst.lazyprobe.exe \
            usdt/tst.lazyprobe.o usdt/lazyprobe.o $(LDLIBS)
        $(POST_PROCESS) ; $(STRIP_STABS)

usdt/lazyprobe.o: usdt/lazyprobe.d usdt/tst.lazyprobe.o
        $(COMPILE.d) -xlazyload -o usdt/lazyprobe.o \
            -s usdt/lazyprobe.d usdt/tst.lazyprobe.o
```

```
127 usdt/tst.lazyprobe.o: usdt/lazyprobe.h

129 usdt/lazyprobe.h: usdt/lazyprobe.d
130         $(DTRACE) -h -s usdt/lazyprobe.d -o usdt/lazyprobe.h

132 SUBDIRS = java_api
133 include ../../Makefile.subdirs
```

```
**********************************************************
    3096 Tue Jan 14 16:49:31 2014
new/usr/src/cmd/dtrace/test/tst/common/aggs/tst.subr.d
4477 DTrace should speak JSON
Reviewed by: Bryan Cantrill <bmc@joyent.com>
**********************************************************
```

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

/*
 * Copyright 2007 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 * Copyright (c) 2012, Joyent, Inc.  All rights reserved.
#endif /* ! codereview */
 */

#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))
INTFUNC(strtoll("0x12EE5D5", 16))
STRFUNC(json("{\"systemtap\": false}", "systemtap"))
#endif /* ! codereview */

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

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

```
**********************************************************
    3846 Tue Jan 14 16:49:31 2014
new/usr/src/cmd/dtrace/test/tst/common/json/tst.general.d
4477 DTrace should speak JSON
Reviewed by: Bryan Cantrill <bmc@joyent.com>
**********************************************************
```

```
/*
 * This file and its contents are supplied under the terms of the
 * Common Development and Distribution License ("CDDL"), version 1.0.
 * You may only use this file in accordance with the terms of version
 * 1.0 of the CDDL.
 *
 * A full copy of the text of the CDDL should have accompanied this
 * source.  A copy of the CDDL is also available via the Internet at
 * http://www.illumos.org/license/CDDL.
 */

/*
 * Copyright 2012, Joyent, Inc.  All rights reserved.
 */

/*
 * General functional tests of JSON parser for json().
 */

#pragma D option quiet
#pragma D option strsize=1k

#define TST(name)                               \
        printf("\ntst |%s|\n", name)
#define IN2(vala, valb)                         \
        in = strjoin(vala, valb);               \
        printf("in  |%s|\n", in)
#define IN(val)                                 \
        in = val;                               \
        printf("in  |%s|\n", in)
#define SEL(ss)                                 \
        out = json(in, ss);                     \
        printf("sel |%s|\nout |%s|\n", ss,      \
            out != NULL ? out : "<NULL>")

BEGIN
{
        TST("empty array");
        IN("[]");
        SEL("0");

        TST("one-element array: integer");
        IN("[1]");
        SEL("0");
        SEL("1");
        SEL("100");
        SEL("-1");

        TST("one-element array: hex integer (not in spec, not supported)");
        IN("[0x1000]");
        SEL("0");

        TST("one-element array: float");
        IN("[1.5001]");
        SEL("0");

        TST("one-element array: float + exponent");
        IN("[16.3e10]");
        SEL("0");
```



```
        TST("one-element array: integer + whitespace");
        IN("[ \t   5\t]");
        SEL("0");

        TST("one-element array: integer + exponent + whitespace");
        IN("[ \t    \t 16E10  \t ]");
        SEL("0");

        TST("one-element array: string");
        IN("[\"alpha\"]");
        SEL("0");

        TST("alternative first-element indexing");
        IN("[1,5,10,15,20]");
        SEL("[0]");
        SEL("[3]");
        SEL("[4]");
        SEL("[5]");

        TST("one-element array: object");
        IN("[ { \"first\": true, \"second\": false }]");
        SEL("0.first");
        SEL("0.second");
        SEL("0.third");

        TST("many-element array: integers");
        IN("[0,1,1,2,3,5,8,13,21,34,55,89,144,233,377]");
        SEL("10"); /* F(10) = 55 */
        SEL("14"); /* F(14) = 377 */
        SEL("19");

        TST("many-element array: multiple types");
        IN2("[\"string\",32,true,{\"a\":9,\"b\":false},100.3e10,false,200.5,",
            "{\"key\":\"val\"},null]");
        SEL("0");
        SEL("0.notobject");
        SEL("1");
        SEL("2");
        SEL("3");
        SEL("3.a");
        SEL("3.b");
        SEL("3.c");
        SEL("4");
        SEL("5");
        SEL("6");
        SEL("7");
        SEL("7.key");
        SEL("7.key.notobject");
        SEL("7.nonexist");
        SEL("8");
        SEL("9");

        TST("many-element array: multiple types + whitespace");
        IN2("\n[\t\"string\" ,\t32 , true\t,\t {\"a\":  9,\t\"b\": false},\t\t",
            "100.3e10, false, 200.5,{\"key\" \t:\n \"val\"},\t\t null ]\t\t");
        SEL("0");
        SEL("0.notobject");
        SEL("1");
        SEL("2");
        SEL("3");
        SEL("3.a");
        SEL("3.b");
        SEL("3.c");
        SEL("4");
        SEL("5");
        SEL("6");
```

```
127         SEL("7");
128         SEL("7.key");
129         SEL("7.key.notobject");
130         SEL("7.nonexist");
131         SEL("8");
132         SEL("9");

134         TST("two-element array: various string escape codes");
135         IN2("[\"abcd \\\" \\\\ \\/ \\b \\f \\n \\r \\t \\u0000 \\uf00F \", ",
136             "\"final\"]");
137         SEL("0");
138         SEL("1");

140         TST("three-element array: broken escape code");
141         IN("[\"fine here\", \"dodgey \\u00AZ\", \"wont get here\"]");
142         SEL("0");
143         SEL("1");
144         SEL("2");

146         TST("nested objects");
147         IN2("{ \"top\": { \"mid\"  : { \"legs\": \"feet\" }, \"number\": 9, ",
148             "\"array\":[0,1,{\"a\":true,\"bb\":[1,2,false,{\"x\":\"yz\"}]}]}}");
149         SEL("top");
150         SEL("fargo");
151         SEL("top.mid");
152         SEL("top.centre");
153         SEL("top.mid.legs");
154         SEL("top.mid.number");
155         SEL("top.mid.array");
156         SEL("top.number");
157         SEL("top.array");
158         SEL("top.array[0]");
159         SEL("top.array[1]");
160         SEL("top.array[2]");
161         SEL("top.array[2].a");
162         SEL("top.array[2].b");
163         SEL("top.array[2].bb");
164         SEL("top.array[2].bb[0]");
165         SEL("top.array[2].bb[1]");
166         SEL("top.array[2].bb[2]");
167         SEL("top.array[2].bb[3]");
168         SEL("top.array[2].bb[3].x");
169         SEL("top.array[2].bb[3].x.nofurther");
170         SEL("top.array[2].bb[4]");
171         SEL("top.array[3]");

173         exit(0);
174 }

176 ERROR
177 {
178         exit(1);
179 }
180 #endif /* ! codereview */
```

```
**********************************************************
    3653 Tue Jan 14 16:49:31 2014
new/usr/src/cmd/dtrace/test/tst/common/json/tst.general.d.out
4477 DTrace should speak JSON
Reviewed by: Bryan Cantrill <bmc@joyent.com>
**********************************************************

tst |empty array|
in  |[]|
sel |0|
out |<NULL>|

tst |one-element array: integer|
in  |[1]|
sel |0|
out |1|
sel |1|
out |<NULL>|
sel |100|
out |<NULL>|
sel |-1|
out |<NULL>|

tst |one-element array: hex integer (not in spec, not supported)|
in  |[0x1000]|
sel |0|
out |<NULL>|

tst |one-element array: float|
in  |[1.5001]|
sel |0|
out |1.5001|

tst |one-element array: float + exponent|
in  |[16.3e10]|
sel |0|
out |16.3e10|

tst |one-element array: integer + whitespace|
in  |[     5    ]|
sel |0|
out |5|

tst |one-element array: integer + exponent + whitespace|
in  |[          16E10          ]|
sel |0|
out |16E10|

tst |one-element array: string|
in  |["alpha"]|
sel |0|
out |alpha|

tst |alternative first-element indexing|
in  |[1,5,10,15,20]|
sel |[0]|
out |1|
sel |[3]|
out |15|
sel |[4]|
out |20|
sel |[5]|
out |<NULL>|

tst |one-element array: object|
in  |[ { "first": true, "second": false }]|
```



```
sel |0.first|
out |true|
sel |0.second|
out |false|
sel |0.third|
out |<NULL>|

tst |many-element array: integers|
in  |[0,1,1,2,3,5,8,13,21,34,55,89,144,233,377]|
sel |10|
out |55|
sel |14|
out |377|
sel |19|
out |<NULL>|

tst |many-element array: multiple types|
in  |["string",32,true,{"a":9,"b":false},100.3e10,false,200.5,{"key":"val"},null
sel |0|
out |string|
sel |0.notobject|
out |<NULL>|
sel |1|
out |32|
sel |2|
out |true|
sel |3|
out |{"a":9,"b":false}|
sel |3.a|
out |9|
sel |3.b|
out |false|
sel |3.c|
out |<NULL>|
sel |4|
out |100.3e10|
sel |5|
out |false|
sel |6|
out |200.5|
sel |7|
out |{"key":"val"}|
sel |7.key|
out |val|
sel |7.key.notobject|
out |<NULL>|
sel |7.nonexist|
out |<NULL>|
sel |8|
out |null|
sel |9|
out |<NULL>|

tst |many-element array: multiple types + whitespace|
in  |
[       "string" ,      32 , true       ,        {"a":  9,      "b": false},
 "val"},                 null ]         |
sel |0|
out |string|
sel |0.notobject|
out |<NULL>|
sel |1|
out |32|
sel |2|
out |true|
sel |3|
```

```
out |{"a":  9,  "b": false}|
sel |3.a|
out |9|
sel |3.b|
out |false|
sel |3.c|
out |<NULL>|
sel |4|
out |100.3e10|
sel |5|
out |false|
sel |6|
out |200.5|
sel |7|
out |{"key"     :
 "val"}|
sel |7.key|
out |val|
sel |7.key.notobject|
out |<NULL>|
sel |7.nonexist|
out |<NULL>|
sel |8|
out |null|
sel |9|
out |<NULL>|

tst |two-element array: various string escape codes|
in  |["abcd \" \\ \/ \b \f \n \r \t \u0000 \uf00F ", "final"]|
sel |0|
out |abcd \" \\ \/ \b \f \n \r \t \u0000 \uf00F |
sel |1|
out |final|

tst |three-element array: broken escape code|
in  |["fine here", "dodgey \u00AZ", "wont get here"]|
sel |0|
out |fine here|
sel |1|
out |<NULL>|
sel |2|
out |<NULL>|

tst |nested objects|
in  |{ "top": { "mid"  : { "legs": "feet" }, "number": 9, "array":[0,1,{"a":true
sel |top|
out |{ "mid"  : { "legs": "feet" }, "number": 9, "array":[0,1,{"a":true,"bb":[1,
sel |fargo|
out |<NULL>|
sel |top.mid|
out |{ "legs": "feet" }|
sel |top.centre|
out |<NULL>|
sel |top.mid.legs|
out |feet|
sel |top.mid.number|
out |<NULL>|
sel |top.mid.array|
out |<NULL>|
sel |top.number|
out |9|
sel |top.array|
out |[0,1,{"a":true,"bb":[1,2,false,{"x":"yz"}]}]|
sel |top.array[0]|
out |0|
sel |top.array[1]|
```



```
out |1|
sel |top.array[2]|
out |{"a":true,"bb":[1,2,false,{"x":"yz"}]}|
sel |top.array[2].a|
out |true|
sel |top.array[2].b|
out |<NULL>|
sel |top.array[2].bb|
out |[1,2,false,{"x":"yz"}]|
sel |top.array[2].bb[0]|
out |1|
sel |top.array[2].bb[1]|
out |2|
sel |top.array[2].bb[2]|
out |false|
sel |top.array[2].bb[3]|
out |{"x":"yz"}|
sel |top.array[2].bb[3].x|
out |yz|
sel |top.array[2].bb[3].x.nofurther|
out |<NULL>|
sel |top.array[2].bb[4]|
out |<NULL>|
sel |top.array[3]|
out |<NULL>|

#endif /* ! codereview */
```

```
**********************************************************
    1329 Tue Jan 14 16:49:31 2014
new/usr/src/cmd/dtrace/test/tst/common/json/tst.strsize.d
4477 DTrace should speak JSON
Reviewed by: Bryan Cantrill <bmc@joyent.com>
**********************************************************
```

```
/*
 * This file and its contents are supplied under the terms of the
 * Common Development and Distribution License ("CDDL"), version 1.0.
 * You may only use this file in accordance with the terms of version
 * 1.0 of the CDDL.
 *
 * A full copy of the text of the CDDL should have accompanied this
 * source.  A copy of the CDDL is also available via the Internet at
 * http://www.illumos.org/license/CDDL.
 */

/*
 * Copyright 2012, Joyent, Inc.  All rights reserved.
 */

/*
 * ASSERTION:
 *   json() run time must be bounded above by strsize.  This test makes strsize
 *   small and deliberately overflows it to prove we bail and return NULL in
 *   the event that we run off the end of the string.
 *
 */

#pragma D option quiet
#pragma D option strsize=18

BEGIN
{
        in = "{\"a\":         1024}"; /* length == 19 */
        out = json(in, "a");
        printf("|%s|\n%s\n\n", in, out != NULL ? out : "<NULL>");

        in = "{\"a\": 1024}"; /* length == 11 */
        out = json(in, "a");
        printf("|%s|\n%s\n\n", in, out != NULL ? out : "<NULL>");

        in = "{\"a\":false,\"b\":true}"; /* length == 20 */
        out = json(in, "b");
        printf("|%s|\n%s\n\n", in, out != NULL ? out : "<NULL>");

        in = "{\"a\":false,\"b\":20}"; /* length == 18 */
        out = json(in, "b");
        printf("|%s|\n%s\n\n", in, out != NULL ? out : "<NULL>");

        exit(0);
}

ERROR
{
        exit(1);
}
#endif /* ! codereview */
```

```
**********************************************************
     104 Tue Jan 14 16:49:32 2014
new/usr/src/cmd/dtrace/test/tst/common/json/tst.strsize.d.out
4477 DTrace should speak JSON
Reviewed by: Bryan Cantrill <bmc@joyent.com>
**********************************************************
|{"a":         1024|
<NULL>

|{"a": 1024}|
1024

|{"a":false,"b":tru|
<NULL>

|{"a":false,"b":20}|
20


#endif /* ! codereview */
```

```
**********************************************************
    1279 Tue Jan 14 16:49:32 2014
new/usr/src/cmd/dtrace/test/tst/common/json/tst.usdt.c
4477 DTrace should speak JSON
Reviewed by: Bryan Cantrill <bmc@joyent.com>
**********************************************************
```

```
/*
 * This file and its contents are supplied under the terms of the
 * Common Development and Distribution License ("CDDL"), version 1.0.
 * You may only use this file in accordance with the terms of version
 * 1.0 of the CDDL.
 *
 * A full copy of the text of the CDDL should have accompanied this
 * source.  A copy of the CDDL is also available via the Internet at
 * http://www.illumos.org/license/CDDL.
 */

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

#include <sys/sdt.h>
#include "usdt.h"

#define FMT     "{" \
                "  \"sizes\": [ \"first\", 2, %f ]," \
                "  \"index\": %d," \
                "  \"facts\": {" \
                "    \"odd\": \"%s\"," \
                "    \"even\": \"%s\"" \
                "  }," \
                "  \"action\": \"%s\"" \
                "}\n"

int
waiting(volatile int *a)
{
        return (*a);
}

int
main(int argc, char **argv)
{
        volatile int a = 0;
        int idx;
        double size = 250.5;

        while (waiting(&a) == 0)
                continue;

        for (idx = 0; idx < 10; idx++) {
                char *odd, *even, *json, *action;

                size *= 1.78;
                odd = idx % 2 == 1 ? "true" : "false";
                even = idx % 2 == 0 ? "true" : "false";
                action = idx == 7 ? "ignore" : "print";

                asprintf(&json, FMT, size, idx, odd, even, action);
                BUNYAN_FAKE_LOG_DEBUG(json);
                free(json);
        }

        BUNYAN_FAKE_LOG_DEBUG("{\"finished\": true}");

        return (0);
```



```
}
#endif /* ! codereview */
```

```
**********************************************************
    1529 Tue Jan 14 16:49:32 2014
new/usr/src/cmd/dtrace/test/tst/common/json/tst.usdt.d
4477 DTrace should speak JSON
Reviewed by: Bryan Cantrill <bmc@joyent.com>
**********************************************************
```

```
/*
 * This file and its contents are supplied under the terms of the
 * Common Development and Distribution License ("CDDL"), version 1.0.
 * You may only use this file in accordance with the terms of version
 * 1.0 of the CDDL.
 *
 * A full copy of the text of the CDDL should have accompanied this
 * source.  A copy of the CDDL is also available via the Internet at
 * http://www.illumos.org/license/CDDL.
 */

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

#pragma D option strsize=4k
#pragma D option quiet
#pragma D option destructive

/*
 * This test reads a JSON string from a USDT probe, roughly simulating the
 * primary motivating use case for the json() subroutine: filtering
 * JSON-formatted log messages from a logging subsystem like node-bunyan.
 */

pid$1:a.out:waiting:entry
{
        this->value = (int *)alloca(sizeof (int));
        *this->value = 1;
        copyout(this->value, arg0, sizeof (int));
}

bunyan*$1:::log-*
{
        this->j = copyinstr(arg0);
}

bunyan*$1:::log-*
/json(this->j, "finished") == NULL && json(this->j, "action") != "ignore"/
{
        this->index = strtoll(json(this->j, "index"));
        this->size = json(this->j, "sizes[2]");
        this->odd = json(this->j, "facts.odd");
        this->even = json(this->j, "facts.even");
        printf("[%d] sz %s odd %s even %s\n", this->index, this->size,
            this->odd, this->even);
}

bunyan*$1:::log-*
/json(this->j, "finished") != NULL/
{
        printf("FINISHED!\n");
        exit(0);
}

tick-10s
{
        printf("ERROR: Timed out before finish message!\n");
        exit(1);
}
```



```
ERROR
{
        exit(1);
}
#endif /* ! codereview */
```

```
**********************************************************
     363 Tue Jan 14 16:49:32 2014
new/usr/src/cmd/dtrace/test/tst/common/json/tst.usdt.d.out
4477 DTrace should speak JSON
Reviewed by: Bryan Cantrill <bmc@joyent.com>
**********************************************************
[0] sz 445.890000 odd false even true
[1] sz 793.684200 odd true even false
[2] sz 1412.757876 odd false even true
[3] sz 2514.709019 odd true even false
[4] sz 4476.182054 odd false even true
[5] sz 7967.604057 odd true even false
[6] sz 14182.335221 odd false even true
[8] sz 44935.310914 odd false even true
[9] sz 79984.853427 odd true even false
FINISHED!

#endif /* ! codereview */
```

```
**********************************************************
     753 Tue Jan 14 16:49:32 2014
new/usr/src/cmd/dtrace/test/tst/common/json/usdt.d
4477 DTrace should speak JSON
Reviewed by: Bryan Cantrill <bmc@joyent.com>
**********************************************************
```

```
/*
 * This file and its contents are supplied under the terms of the
 * Common Development and Distribution License ("CDDL"), version 1.0.
 * You may only use this file in accordance with the terms of version
 * 1.0 of the CDDL.
 *
 * A full copy of the text of the CDDL should have accompanied this
 * source.  A copy of the CDDL is also available via the Internet at
 * http://www.illumos.org/license/CDDL.
 */

/*
 * Copyright 2012, Joyent, Inc.  All rights reserved.
 */

/*
 * Sets up a fake node-bunyan-like USDT provider for use from C.
 */

provider bunyan_fake {
        probe log__trace(char *msg);
        probe log__debug(char *msg);
        probe log__info(char *msg);
        probe log__warn(char *msg);
        probe log__error(char *msg);
        probe log__fatal(char *msg);
};
#endif /* ! codereview */
```

```
**********************************************************
    2609 Tue Jan 14 16:49:33 2014
new/usr/src/cmd/dtrace/test/tst/common/privs/tst.func_access.ksh
4477 DTrace should speak JSON
Reviewed by: Bryan Cantrill <bmc@joyent.com>
**********************************************************
```

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

#
# Copyright 2006 Sun Microsystems, Inc.  All rights reserved.
# Use is subject to license terms.
# Copyright (c) 2012, Joyent, Inc. All rights reserved.
#endif /* ! codereview */
#
#ident  "%Z%%M% %I%     %E% SMI"

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

/usr/sbin/dtrace -q -s /dev/stdin <<"EOF"

BEGIN {
        errorcount = 0;
        expected_errorcount = 27;
        expected_errorcount = 23;
}

BEGIN { trace(mutex_owned(&‘pidlock)); }
BEGIN { trace(mutex_owner(&‘pidlock)); }
BEGIN { trace(mutex_type_adaptive(&‘pidlock)); }
BEGIN { trace(mutex_type_spin(&‘pidlock)); }

BEGIN { trace(rw_read_held(&‘ksyms_lock)); }
BEGIN { trace(rw_write_held(&‘ksyms_lock)); }
BEGIN { trace(rw_iswriter(&‘ksyms_lock)); }

BEGIN { x = alloca(10); bcopy(‘initname, x, 10); trace(stringof(x)); }
/* We have no reliable way to test msgsize */

BEGIN { trace(strlen(‘initname)); }
BEGIN { trace(strchr(‘initname, 0x69)); }
BEGIN { trace(strrchr(‘initname, 0x69)); }
BEGIN { trace(strstr("/sbin/init/foo", ‘initname)); }
BEGIN { trace(strstr(‘initname, "in")); }
BEGIN { trace(strtok(‘initname, "/")); }
BEGIN { trace(strtok(NULL, "/")); }
BEGIN { trace(strtok("foo/bar", ‘initname)); }
BEGIN { trace(strtok(NULL, ‘initname)); }
```



```
BEGIN { trace(strtoll(‘initname)); }
BEGIN { trace(strtoll(‘initname, 10)); }
#endif /* ! codereview */
BEGIN { trace(substr(‘initname, 2, 3)); }

BEGIN { trace(ddi_pathname(‘top_devinfo, 1)); }
BEGIN { trace(strjoin(‘initname, "foo")); }
BEGIN { trace(strjoin("foo", ‘initname)); }
BEGIN { trace(dirname(‘initname)); }
BEGIN { trace(cleanpath(‘initname)); }

BEGIN { j = "{\"/sbin/init\":\"uh oh\"}"; trace(json(j, ‘initname)); }
BEGIN { trace(json(‘initname, "x")); }

#endif /* ! codereview */
ERROR {
        errorcount++;
}

BEGIN /errorcount == expected_errorcount/ {
        trace("test passed");
        exit(0);
}

BEGIN /errorcount != expected_errorcount/ {
        printf("fail: expected %d.  saw %d.", expected_errorcount, errorcount);
        exit(1);
}
EOF


exit $?
```

****
749 Tue Jan 14 16:49:33 2014
new/usr/src/cmd/dtrace/test/tst/common/strtoll/err.BaseTooLarge.d
4477 DTrace should speak JSON
Reviewed by: Bryan Cantrill <bmc@joyent.com>
****

```
/*
 * This file and its contents are supplied under the terms of the
 * Common Development and Distribution License ("CDDL"), version 1.0.
 * You may only use this file in accordance with the terms of version
 * 1.0 of the CDDL.
 *
 * A full copy of the text of the CDDL should have accompanied this
 * source.  A copy of the CDDL is also available via the Internet at
 * http://www.illumos.org/license/CDDL.
 */

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

/*
 * ASSERTION:
 *   The largest base we will accept is Base 36 -- i.e. using all of 0-9 and
 *   A-Z as numerals.
 *
 * SECTION: Actions and Subroutines/strtoll()
 */

#pragma D option quiet

BEGIN
{
        printf("%d\n", strtoll("0", 37));
        exit(0);
}

ERROR
{
        exit(1);
}
#endif /* ! codereview */
```

****
698 Tue Jan 14 16:49:34 2014
new/usr/src/cmd/dtrace/test/tst/common/strtoll/err.BaseTooSmall.d
4477 DTrace should speak JSON
Reviewed by: Bryan Cantrill <bmc@joyent.com>
****

```
/*
 * This file and its contents are supplied under the terms of the
 * Common Development and Distribution License ("CDDL"), version 1.0.
 * You may only use this file in accordance with the terms of version
 * 1.0 of the CDDL.
 *
 * A full copy of the text of the CDDL should have accompanied this
 * source.  A copy of the CDDL is also available via the Internet at
 * http://www.illumos.org/license/CDDL.
 */

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

/*
 * ASSERTION:
 *   The smallest base we will accept is Base 2.
 *
 * SECTION: Actions and Subroutines/strtoll()
 */

#pragma D option quiet

BEGIN
{
        printf("%d\n", strtoll("0", 1));
        exit(0);
}

ERROR
{
        exit(1);
}
#endif /* ! codereview */
```

```
**********************************************************
    1699 Tue Jan 14 16:49:34 2014
new/usr/src/cmd/dtrace/test/tst/common/strtoll/tst.strtoll.d
4477 DTrace should speak JSON
Reviewed by: Bryan Cantrill <bmc@joyent.com>
**********************************************************
```

```
/*
 * This file and its contents are supplied under the terms of the
 * Common Development and Distribution License ("CDDL"), version 1.0.
 * You may only use this file in accordance with the terms of version
 * 1.0 of the CDDL.
 *
 * A full copy of the text of the CDDL should have accompanied this
 * source.  A copy of the CDDL is also available via the Internet at
 * http://www.illumos.org/license/CDDL.
 */

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

/*
 * ASSERTION:
 *   Test the strtoll() subroutine.
 *
 * SECTION: Actions and Subroutines/strtoll()
 */

#pragma D option quiet

BEGIN
{

        /* minimum base (2) and maximum base (36): */
        printf("%d\n", strtoll("0", 2));
        printf("%d\n", strtoll("1", 36));

        /* simple tests: */
        printf("%d\n", strtoll("0x20", 16));
        printf("%d\n", strtoll("-32", 10));
        printf("%d\n", strtoll("010", 8));
        printf("%d\n", strtoll("101010", 2));

        /* INT64_MIN and INT64_MAX: */
        printf("%d\n", strtoll("9223372036854775807"));
        printf("%d\n", strtoll("-9223372036854775808"));
        printf("%d\n", strtoll("0777777777777777777777", 8));
        printf("%d\n", strtoll("-01000000000000000000000", 8));

        /* wrapping: */
        printf("%d\n", strtoll("100000000000000000000", 8));
        printf("%d\n", strtoll("-100000000000000000001", 8));

        /* hex without prefix: */
        printf("%d\n", strtoll("baddcafe", 16));

        /* stopping at first out-of-base character: */
        printf("%d\n", strtoll("12j", 10));
        printf("%d\n", strtoll("102", 2));

        /* base 36: */
        printf("%d\n", strtoll("-0DTrace4EverZ", 36));

        /* base 10 is assumed: */
        printf("%d\n", strtoll("1985"));
        printf("%d\n", strtoll("-2012"));
```



```
        /* empty string: */
        printf("%d\n", strtoll(""));

        exit(0);
}
#endif /* ! codereview */
```

```
**********************************************************
     190 Tue Jan 14 16:49:34 2014
new/usr/src/cmd/dtrace/test/tst/common/strtoll/tst.strtoll.d.out
4477 DTrace should speak JSON
Reviewed by: Bryan Cantrill <bmc@joyent.com>
**********************************************************
```

```
   1 0
   2 1
   3 32
   4 -32
   5 8
   6 42
   7 9223372036854775807
   8 -9223372036854775808
   9 9223372036854775807
  10 -9223372036854775808
  11 -9223372036854775808
  12 9223372036854775807
  13 3135097598
  14 12
  15 2
  16 -1819882045752187535
  17 1985
  18 -2012
  19 0

  21 #endif /* ! codereview */
```

```
**********************************************************
    2529 Tue Jan 14 16:49:34 2014
new/usr/src/common/util/strtolctype.h
4477 DTrace should speak JSON
Reviewed by: Bryan Cantrill <bmc@joyent.com>
**********************************************************
```

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

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

/*      Copyright (c) 1988 AT&T */
/*        All Rights Reserved   */

#ifndef _COMMON_UTIL_CTYPE_H
#define _COMMON_UTIL_CTYPE_H

#ifdef  __cplusplus
extern "C" {
#endif

/*
 * This header file contains a collection of macros that the strtou?ll?
 * functions in common/util use to test characters.  What we need is a kernel
 * version of ctype.h.
 *
 * NOTE: These macros are used within several DTrace probe context functions.
 * They must not be altered to make function calls or perform actions not
 * safe in probe context.
#endif /* ! codereview */
 */

#if     defined(_KERNEL) && !defined(_BOOT)

#define isalnum(ch)     (isalpha(ch) || isdigit(ch))
#define isalpha(ch)     (isupper(ch) || islower(ch))
#define isdigit(ch)     ((ch) >= '0' && (ch) <= '9')
#define islower(ch)     ((ch) >= 'a' && (ch) <= 'z')
#define isspace(ch)     (((ch) == ' ') || ((ch) == '\r') || ((ch) == '\n') || \
                        ((ch) == '\t') || ((ch) == '\f'))
#define isupper(ch)     ((ch) >= 'A' && (ch) <= 'Z')
#define isxdigit(ch)    (isdigit(ch) || ((ch) >= 'a' && (ch) <= 'f') || \
                        ((ch) >= 'A' && (ch) <= 'F'))

#endif  /* _KERNEL && !_BOOT */
```



```
#define DIGIT(x)        \
        (isdigit(x) ? (x) - '0' : islower(x) ? (x) + 10 - 'a' : (x) + 10 - 'A')

#define MBASE   ('z' - 'a' + 1 + 10)

/*
 * The following macro is a version of isalnum() that limits alphabetic
 * characters to the ranges a-z and A-Z; locale dependent characters will not
 * return 1.  The members of a-z and A-Z are assumed to be in ascending order
 * and contiguous.
 */
#define lisalnum(x)     \
        (isdigit(x) || ((x) >= 'a' && (x) <= 'z') || ((x) >= 'A' && (x) <= 'Z'))

#ifdef  __cplusplus
}
#endif

#endif  /* _COMMON_UTIL_CTYPE_H */
```

```
**********************************************************
   54172 Tue Jan 14 16:49:35 2014
new/usr/src/lib/libdtrace/common/dt_open.c
4477 DTrace should speak JSON
Reviewed by: Bryan Cantrill <bmc@joyent.com>
**********************************************************
_____unchanged_portion_omitted_

  82 /*
  83  * The version number should be increased for every customer visible release
  84  * of DTrace. The major number should be incremented when a fundamental
  85  * change has been made that would affect all consumers, and would reflect
  86  * sweeping changes to DTrace or the D language. The minor number should be
  87  * incremented when a change is introduced that could break scripts that had
  88  * previously worked; for example, adding a new built-in variable could break
  89  * a script which was already using that identifier. The micro number should
  90  * be changed when introducing functionality changes or major bug fixes that
  91  * do not affect backward compatibility -- this is merely to make capabilities
  92  * easily determined from the version number. Minor bugs do not require any
  93  * modification to the version number.
  94  */
  95 #define DT_VERS_1_0     DT_VERSION_NUMBER(1, 0, 0)
  96 #define DT_VERS_1_1     DT_VERSION_NUMBER(1, 1, 0)
  97 #define DT_VERS_1_2     DT_VERSION_NUMBER(1, 2, 0)
  98 #define DT_VERS_1_2_1   DT_VERSION_NUMBER(1, 2, 1)
  99 #define DT_VERS_1_2_2   DT_VERSION_NUMBER(1, 2, 2)
 100 #define DT_VERS_1_3     DT_VERSION_NUMBER(1, 3, 0)
 101 #define DT_VERS_1_4     DT_VERSION_NUMBER(1, 4, 0)
 102 #define DT_VERS_1_4_1   DT_VERSION_NUMBER(1, 4, 1)
 103 #define DT_VERS_1_5     DT_VERSION_NUMBER(1, 5, 0)
 104 #define DT_VERS_1_6     DT_VERSION_NUMBER(1, 6, 0)
 105 #define DT_VERS_1_6_1   DT_VERSION_NUMBER(1, 6, 1)
 106 #define DT_VERS_1_6_2   DT_VERSION_NUMBER(1, 6, 2)
 107 #define DT_VERS_1_6_3   DT_VERSION_NUMBER(1, 6, 3)
 108 #define DT_VERS_1_7     DT_VERSION_NUMBER(1, 7, 0)
 109 #define DT_VERS_1_7_1   DT_VERSION_NUMBER(1, 7, 1)
 110 #define DT_VERS_1_8     DT_VERSION_NUMBER(1, 8, 0)
 111 #define DT_VERS_1_8_1   DT_VERSION_NUMBER(1, 8, 1)
 112 #define DT_VERS_1_9     DT_VERSION_NUMBER(1, 9, 0)
 113 #define DT_VERS_1_9_1   DT_VERSION_NUMBER(1, 9, 1)
 114 #define DT_VERS_1_10    DT_VERSION_NUMBER(1, 10, 0)
 115 #define DT_VERS_1_11    DT_VERSION_NUMBER(1, 11, 0)
 116 #define DT_VERS_LATEST  DT_VERS_1_11
 117 #define DT_VERS_STRING  "Sun D 1.11"
 115 #define DT_VERS_LATEST  DT_VERS_1_10
 116 #define DT_VERS_STRING  "Sun D 1.10"

 119 const dt_version_t _dtrace_versions[] = {
 120         DT_VERS_1_0,    /* D API 1.0.0 (PSARC 2001/466) Solaris 10 FCS */
 121         DT_VERS_1_1,    /* D API 1.1.0 Solaris Express 6/05 */
 122         DT_VERS_1_2,    /* D API 1.2.0 Solaris 10 Update 1 */
 123         DT_VERS_1_2_1,  /* D API 1.2.1 Solaris Express 4/06 */
 124         DT_VERS_1_2_2,  /* D API 1.2.2 Solaris Express 6/06 */
 125         DT_VERS_1_3,    /* D API 1.3 Solaris Express 10/06 */
 126         DT_VERS_1_4,    /* D API 1.4 Solaris Express 2/07 */
 127         DT_VERS_1_4_1,  /* D API 1.4.1 Solaris Express 4/07 */
 128         DT_VERS_1_5,    /* D API 1.5 Solaris Express 7/07 */
 129         DT_VERS_1_6,    /* D API 1.6 */
 130         DT_VERS_1_6_1,  /* D API 1.6.1 */
 131         DT_VERS_1_6_2,  /* D API 1.6.2 */
 132         DT_VERS_1_6_3,  /* D API 1.6.3 */
 133         DT_VERS_1_7,    /* D API 1.7 */
 134         DT_VERS_1_7_1,  /* D API 1.7.1 */
 135         DT_VERS_1_8,    /* D API 1.8 */
 136         DT_VERS_1_8_1,  /* D API 1.8.1 */
 137         DT_VERS_1_9,    /* D API 1.9 */
```



```
 138         DT_VERS_1_9_1,  /* D API 1.9.1 */
 139         DT_VERS_1_10,   /* D API 1.10 */
 140         DT_VERS_1_11,   /* D API 1.11 */
 141 #endif /* ! codereview */
 142         0
 143 };

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

```
204 { "copyinstr", DT_IDENT_FUNC, 0, DIF_SUBR_COPYINSTR,
205         DT_ATTR_STABCMN, DT_VERS_1_0,
206         &dt_idops_func, "string(uintptr_t, [size_t])" },
207 { "copyinto", DT_IDENT_FUNC, 0, DIF_SUBR_COPYINTO, DT_ATTR_STABCMN,
208         DT_VERS_1_0, &dt_idops_func, "void(uintptr_t, size_t, void *)" },
209 { "copyout", DT_IDENT_FUNC, 0, DIF_SUBR_COPYOUT, DT_ATTR_STABCMN, DT_VERS_1_0,
210         &dt_idops_func, "void(void *, uintptr_t, size_t)" },
211 { "copyoutstr", DT_IDENT_FUNC, 0, DIF_SUBR_COPYOUTSTR,
212         DT_ATTR_STABCMN, DT_VERS_1_0,
213         &dt_idops_func, "void(char *, uintptr_t, size_t)" },
214 { "count", DT_IDENT_AGGFUNC, 0, DTRACEAGG_COUNT, DT_ATTR_STABCMN, DT_VERS_1_0,
215         &dt_idops_func, "void()" },
216 { "curthread", DT_IDENT_SCALAR, 0, DIF_VAR_CURTHREAD,
217         { DTRACE_STABILITY_STABLE, DTRACE_STABILITY_PRIVATE,
218         DTRACE_CLASS_COMMON }, DT_VERS_1_0,
219         &dt_idops_type, "genunix`kthread_t *" },
220 { "ddi_pathname", DT_IDENT_FUNC, 0, DIF_SUBR_DDI_PATHNAME,
221         DT_ATTR_EVOLCMN, DT_VERS_1_0,
222         &dt_idops_func, "string(void *, int64_t)" },
223 { "denormalize", DT_IDENT_ACTFUNC, 0, DT_ACT_DENORMALIZE, DT_ATTR_STABCMN,
224         DT_VERS_1_0, &dt_idops_func, "void(...)" },
225 { "dirname", DT_IDENT_FUNC, 0, DIF_SUBR_DIRNAME, DT_ATTR_STABCMN, DT_VERS_1_0,
226         &dt_idops_func, "string(const char *)" },
227 { "discard", DT_IDENT_ACTFUNC, 0, DT_ACT_DISCARD, DT_ATTR_STABCMN, DT_VERS_1_0,
228         &dt_idops_func, "void(int)" },
229 { "epid", DT_IDENT_SCALAR, 0, DIF_VAR_EPID, DT_ATTR_STABCMN, DT_VERS_1_0,
230         &dt_idops_type, "uint_t" },
231 { "errno", DT_IDENT_SCALAR, 0, DIF_VAR_ERRNO, DT_ATTR_STABCMN, DT_VERS_1_0,
232         &dt_idops_type, "int" },
233 { "execname", DT_IDENT_SCALAR, 0, DIF_VAR_EXECNAME,
234         DT_ATTR_STABCMN, DT_VERS_1_0, &dt_idops_type, "string" },
235 { "exit", DT_IDENT_ACTFUNC, 0, DT_ACT_EXIT, DT_ATTR_STABCMN, DT_VERS_1_0,
236         &dt_idops_func, "void(int)" },
237 { "freopen", DT_IDENT_ACTFUNC, 0, DT_ACT_FREOPEN, DT_ATTR_STABCMN,
238         DT_VERS_1_1, &dt_idops_func, "void(@, ...)" },
239 { "ftruncate", DT_IDENT_ACTFUNC, 0, DT_ACT_FTRUNCATE, DT_ATTR_STABCMN,
240         DT_VERS_1_0, &dt_idops_func, "void()" },
241 { "func", DT_IDENT_ACTFUNC, 0, DT_ACT_SYM, DT_ATTR_STABCMN,
242         DT_VERS_1_2, &dt_idops_func, "_symaddr(uintptr_t)" },
243 { "getmajor", DT_IDENT_FUNC, 0, DIF_SUBR_GETMAJOR,
244         DT_ATTR_EVOLCMN, DT_VERS_1_0,
245         &dt_idops_func, "genunix`major_t(genunix`dev_t)" },
246 { "getminor", DT_IDENT_FUNC, 0, DIF_SUBR_GETMINOR,
247         DT_ATTR_EVOLCMN, DT_VERS_1_0,
248         &dt_idops_func, "genunix`minor_t(genunix`dev_t)" },
249 { "htonl", DT_IDENT_FUNC, 0, DIF_SUBR_HTONL, DT_ATTR_EVOLCMN, DT_VERS_1_3,
250         &dt_idops_func, "uint32_t(uint32_t)" },
251 { "htonll", DT_IDENT_FUNC, 0, DIF_SUBR_HTONLL, DT_ATTR_EVOLCMN, DT_VERS_1_3,
252         &dt_idops_func, "uint64_t(uint64_t)" },
253 { "htons", DT_IDENT_FUNC, 0, DIF_SUBR_HTONS, DT_ATTR_EVOLCMN, DT_VERS_1_3,
254         &dt_idops_func, "uint16_t(uint16_t)" },
255 { "getf", DT_IDENT_FUNC, 0, DIF_SUBR_GETF, DT_ATTR_STABCMN, DT_VERS_1_10,
256         &dt_idops_func, "file_t *(int)" },
257 { "gid", DT_IDENT_SCALAR, 0, DIF_VAR_GID, DT_ATTR_STABCMN, DT_VERS_1_0,
258         &dt_idops_type, "gid_t" },
259 { "id", DT_IDENT_SCALAR, 0, DIF_VAR_ID, DT_ATTR_STABCMN, DT_VERS_1_0,
260         &dt_idops_type, "uint_t" },
261 { "index", DT_IDENT_FUNC, 0, DIF_SUBR_INDEX, DT_ATTR_STABCMN, DT_VERS_1_1,
262         &dt_idops_func, "int(const char *, const char *, [int])" },
263 { "inet_ntoa", DT_IDENT_FUNC, 0, DIF_SUBR_INET_NTOA, DT_ATTR_STABCMN,
264         DT_VERS_1_5, &dt_idops_func, "string(ipaddr_t *)" },
265 { "inet_ntoa6", DT_IDENT_FUNC, 0, DIF_SUBR_INET_NTOA6, DT_ATTR_STABCMN,
266         DT_VERS_1_5, &dt_idops_func, "string(in6_addr_t *)" },
267 { "inet_ntop", DT_IDENT_FUNC, 0, DIF_SUBR_INET_NTOP, DT_ATTR_STABCMN,
268         DT_VERS_1_5, &dt_idops_func, "string(int, void *)" },
269 { "ipl", DT_IDENT_SCALAR, 0, DIF_VAR_IPL, DT_ATTR_STABCMN, DT_VERS_1_0,
```



```
270         &dt_idops_type, "uint_t" },
271 { "json", DT_IDENT_FUNC, 0, DIF_SUBR_JSON, DT_ATTR_STABCMN, DT_VERS_1_11,
272         &dt_idops_func, "string(const char *, const char *)" },
273 #endif /* ! codereview */
274 { "jstack", DT_IDENT_ACTFUNC, 0, DT_ACT_JSTACK, DT_ATTR_STABCMN, DT_VERS_1_0,
275         &dt_idops_func, "stack(...)" },
276 { "lltostr", DT_IDENT_FUNC, 0, DIF_SUBR_LLTOSTR, DT_ATTR_STABCMN, DT_VERS_1_0,
277         &dt_idops_func, "string(int64_t, [int])" },
278 { "llquantize", DT_IDENT_AGGFUNC, 0, DTRACEAGG_LLQUANTIZE, DT_ATTR_STABCMN,
279         DT_VERS_1_7, &dt_idops_func,
280         "void(@, int32_t, int32_t, int32_t, int32_t, ...)" },
281 { "lquantize", DT_IDENT_AGGFUNC, 0, DTRACEAGG_LQUANTIZE,
282         DT_ATTR_STABCMN, DT_VERS_1_0,
283         &dt_idops_func, "void(@, int32_t, int32_t, ...)" },
284 { "max", DT_IDENT_AGGFUNC, 0, DTRACEAGG_MAX, DT_ATTR_STABCMN, DT_VERS_1_0,
285         &dt_idops_func, "void(@)" },
286 { "min", DT_IDENT_AGGFUNC, 0, DTRACEAGG_MIN, DT_ATTR_STABCMN, DT_VERS_1_0,
287         &dt_idops_func, "void(@)" },
288 { "mod", DT_IDENT_ACTFUNC, 0, DT_ACT_MOD, DT_ATTR_STABCMN,
289         DT_VERS_1_2, &dt_idops_func, "_symaddr(uintptr_t)" },
290 { "msgdsize", DT_IDENT_FUNC, 0, DIF_SUBR_MSGDSIZE,
291         DT_ATTR_STABCMN, DT_VERS_1_0,
292         &dt_idops_func, "size_t(mblk_t *)" },
293 { "msgsize", DT_IDENT_FUNC, 0, DIF_SUBR_MSGSIZE,
294         DT_ATTR_STABCMN, DT_VERS_1_0,
295         &dt_idops_func, "size_t(mblk_t *)" },
296 { "mutex_owned", DT_IDENT_FUNC, 0, DIF_SUBR_MUTEX_OWNED,
297         DT_ATTR_EVOLCMN, DT_VERS_1_0,
298         &dt_idops_func, "int(genunix`kmutex_t *)" },
299 { "mutex_owner", DT_IDENT_FUNC, 0, DIF_SUBR_MUTEX_OWNER,
300         DT_ATTR_EVOLCMN, DT_VERS_1_0,
301         &dt_idops_func, "genunix`kthread_t *(genunix`kmutex_t *)" },
302 { "mutex_type_adaptive", DT_IDENT_FUNC, 0, DIF_SUBR_MUTEX_TYPE_ADAPTIVE,
303         DT_ATTR_EVOLCMN, DT_VERS_1_0,
304         &dt_idops_func, "int(genunix`kmutex_t *)" },
305 { "mutex_type_spin", DT_IDENT_FUNC, 0, DIF_SUBR_MUTEX_TYPE_SPIN,
306         DT_ATTR_EVOLCMN, DT_VERS_1_0,
307         &dt_idops_func, "int(genunix`kmutex_t *)" },
308 { "ntohl", DT_IDENT_FUNC, 0, DIF_SUBR_NTOHL, DT_ATTR_EVOLCMN, DT_VERS_1_3,
309         &dt_idops_func, "uint32_t(uint32_t)" },
310 { "ntohll", DT_IDENT_FUNC, 0, DIF_SUBR_NTOHLL, DT_ATTR_EVOLCMN, DT_VERS_1_3,
311         &dt_idops_func, "uint64_t(uint64_t)" },
312 { "ntohs", DT_IDENT_FUNC, 0, DIF_SUBR_NTOHS, DT_ATTR_EVOLCMN, DT_VERS_1_3,
313         &dt_idops_func, "uint16_t(uint16_t)" },
314 { "normalize", DT_IDENT_ACTFUNC, 0, DT_ACT_NORMALIZE, DT_ATTR_STABCMN,
315         DT_VERS_1_0, &dt_idops_func, "void(...)" },
316 { "panic", DT_IDENT_ACTFUNC, 0, DT_ACT_PANIC, DT_ATTR_STABCMN, DT_VERS_1_0,
317         &dt_idops_func, "void()" },
318 { "pid", DT_IDENT_SCALAR, 0, DIF_VAR_PID, DT_ATTR_STABCMN, DT_VERS_1_0,
319         &dt_idops_type, "pid_t" },
320 { "ppid", DT_IDENT_SCALAR, 0, DIF_VAR_PPID, DT_ATTR_STABCMN, DT_VERS_1_0,
321         &dt_idops_type, "pid_t" },
322 { "print", DT_IDENT_ACTFUNC, 0, DT_ACT_PRINT, DT_ATTR_STABCMN, DT_VERS_1_9,
323         &dt_idops_func, "void(@)" },
324 { "printa", DT_IDENT_ACTFUNC, 0, DT_ACT_PRINTA, DT_ATTR_STABCMN, DT_VERS_1_0,
325         &dt_idops_func, "void(@, ...)" },
326 { "printf", DT_IDENT_ACTFUNC, 0, DT_ACT_PRINTF, DT_ATTR_STABCMN, DT_VERS_1_0,
327         &dt_idops_func, "void(@, ...)" },
328 { "probefunc", DT_IDENT_SCALAR, 0, DIF_VAR_PROBEFUNC,
329         DT_ATTR_STABCMN, DT_VERS_1_0, &dt_idops_type, "string" },
330 { "probemod", DT_IDENT_SCALAR, 0, DIF_VAR_PROBEMOD,
331         DT_ATTR_STABCMN, DT_VERS_1_0, &dt_idops_type, "string" },
332 { "probename", DT_IDENT_SCALAR, 0, DIF_VAR_PROBENAME,
333         DT_ATTR_STABCMN, DT_VERS_1_0, &dt_idops_type, "string" },
334 { "probeprov", DT_IDENT_SCALAR, 0, DIF_VAR_PROBEPROV,
335         DT_ATTR_STABCMN, DT_VERS_1_0, &dt_idops_type, "string" },
```

```
{ "progenyof", DT_IDENT_FUNC, 0, DIF_SUBR_PROGENYOF,
        DT_ATTR_STABCMN, DT_VERS_1_0,
        &dt_idops_func, "int(pid_t)" },
{ "quantize", DT_IDENT_AGGFUNC, 0, DTRACEAGG_QUANTIZE,
        DT_ATTR_STABCMN, DT_VERS_1_0,
        &dt_idops_func, "void(@, ...)" },
{ "raise", DT_IDENT_ACTFUNC, 0, DT_ACT_RAISE, DT_ATTR_STABCMN, DT_VERS_1_0,
        &dt_idops_func, "void(int)" },
{ "rand", DT_IDENT_FUNC, 0, DIF_SUBR_RAND, DT_ATTR_STABCMN, DT_VERS_1_0,
        &dt_idops_func, "int()" },
{ "rindex", DT_IDENT_FUNC, 0, DIF_SUBR_RINDEX, DT_ATTR_STABCMN, DT_VERS_1_1,
        &dt_idops_func, "int(const char *, const char *, [int])" },
{ "rw_iswriter", DT_IDENT_FUNC, 0, DIF_SUBR_RW_ISWRITER,
        DT_ATTR_EVOLCMN, DT_VERS_1_0,
        &dt_idops_func, "int(genunix`krwlock_t *)" },
{ "rw_read_held", DT_IDENT_FUNC, 0, DIF_SUBR_RW_READ_HELD,
        DT_ATTR_EVOLCMN, DT_VERS_1_0,
        &dt_idops_func, "int(genunix`krwlock_t *)" },
{ "rw_write_held", DT_IDENT_FUNC, 0, DIF_SUBR_RW_WRITE_HELD,
        DT_ATTR_EVOLCMN, DT_VERS_1_0,
        &dt_idops_func, "int(genunix`krwlock_t *)" },
{ "self", DT_IDENT_PTR, 0, 0, DT_ATTR_STABCMN, DT_VERS_1_0,
        &dt_idops_type, "void" },
{ "setopt", DT_IDENT_ACTFUNC, 0, DT_ACT_SETOPT, DT_ATTR_STABCMN,
        DT_VERS_1_2, &dt_idops_func, "void(const char *, [const char *])" },
{ "speculate", DT_IDENT_ACTFUNC, 0, DT_ACT_SPECULATE,
        DT_ATTR_STABCMN, DT_VERS_1_0,
        &dt_idops_func, "void(int)" },
{ "speculation", DT_IDENT_FUNC, 0, DIF_SUBR_SPECULATION,
        DT_ATTR_STABCMN, DT_VERS_1_0,
        &dt_idops_func, "int()" },
{ "stack", DT_IDENT_ACTFUNC, 0, DT_ACT_STACK, DT_ATTR_STABCMN, DT_VERS_1_0,
        &dt_idops_func, "stack(...)" },
{ "stackdepth", DT_IDENT_SCALAR, 0, DIF_VAR_STACKDEPTH,
        DT_ATTR_STABCMN, DT_VERS_1_0,
        &dt_idops_type, "uint32_t" },
{ "stddev", DT_IDENT_AGGFUNC, 0, DTRACEAGG_STDDEV, DT_ATTR_STABCMN,
        DT_VERS_1_6, &dt_idops_func, "void(@)" },
{ "stop", DT_IDENT_ACTFUNC, 0, DT_ACT_STOP, DT_ATTR_STABCMN, DT_VERS_1_0,
        &dt_idops_func, "void()" },
{ "strchr", DT_IDENT_FUNC, 0, DIF_SUBR_STRCHR, DT_ATTR_STABCMN, DT_VERS_1_1,
        &dt_idops_func, "string(const char *, char)" },
{ "strlen", DT_IDENT_FUNC, 0, DIF_SUBR_STRLEN, DT_ATTR_STABCMN, DT_VERS_1_0,
        &dt_idops_func, "size_t(const char *)" },
{ "strjoin", DT_IDENT_FUNC, 0, DIF_SUBR_STRJOIN, DT_ATTR_STABCMN, DT_VERS_1_0,
        &dt_idops_func, "string(const char *, const char *)" },
{ "strrchr", DT_IDENT_FUNC, 0, DIF_SUBR_STRRCHR, DT_ATTR_STABCMN, DT_VERS_1_1,
        &dt_idops_func, "string(const char *, char)" },
{ "strstr", DT_IDENT_FUNC, 0, DIF_SUBR_STRSTR, DT_ATTR_STABCMN, DT_VERS_1_1,
        &dt_idops_func, "string(const char *, const char *)" },
{ "strtok", DT_IDENT_FUNC, 0, DIF_SUBR_STRTOK, DT_ATTR_STABCMN, DT_VERS_1_1,
        &dt_idops_func, "string(const char *, const char *)" },
{ "strtoll", DT_IDENT_FUNC, 0, DIF_SUBR_STRTOLL, DT_ATTR_STABCMN, DT_VERS_1_11,
        &dt_idops_func, "int64_t(const char *, [int])" },
#endif /* ! codereview */
{ "substr", DT_IDENT_FUNC, 0, DIF_SUBR_SUBSTR, DT_ATTR_STABCMN, DT_VERS_1_1,
        &dt_idops_func, "string(const char *, int, [int])" },
{ "sum", DT_IDENT_AGGFUNC, 0, DTRACEAGG_SUM, DT_ATTR_STABCMN, DT_VERS_1_0,
        &dt_idops_func, "void(@)" },
{ "sym", DT_IDENT_ACTFUNC, 0, DT_ACT_SYM, DT_ATTR_STABCMN,
        DT_VERS_1_2, &dt_idops_func, "_symaddr(uintptr_t)" },
{ "system", DT_IDENT_ACTFUNC, 0, DT_ACT_SYSTEM, DT_ATTR_STABCMN, DT_VERS_1_0,
        &dt_idops_func, "void(@, ...)" },
{ "this", DT_IDENT_PTR, 0, 0, DT_ATTR_STABCMN, DT_VERS_1_0,
        &dt_idops_type, "void" },
{ "tid", DT_IDENT_SCALAR, 0, DIF_VAR_TID, DT_ATTR_STABCMN, DT_VERS_1_0,
```



```
        &dt_idops_type, "id_t" },
{ "timestamp", DT_IDENT_SCALAR, 0, DIF_VAR_TIMESTAMP,
        DT_ATTR_STABCMN, DT_VERS_1_0,
        &dt_idops_type, "uint64_t" },
{ "tolower", DT_IDENT_FUNC, 0, DIF_SUBR_TOLOWER, DT_ATTR_STABCMN, DT_VERS_1_8,
        &dt_idops_func, "string(const char *)" },
{ "toupper", DT_IDENT_FUNC, 0, DIF_SUBR_TOUPPER, DT_ATTR_STABCMN, DT_VERS_1_8,
        &dt_idops_func, "string(const char *)" },
{ "trace", DT_IDENT_ACTFUNC, 0, DT_ACT_TRACE, DT_ATTR_STABCMN, DT_VERS_1_0,
        &dt_idops_func, "void(@)" },
{ "tracemem", DT_IDENT_ACTFUNC, 0, DT_ACT_TRACEMEM,
        DT_ATTR_STABCMN, DT_VERS_1_0,
        &dt_idops_func, "void(@, size_t, ...)" },
{ "trunc", DT_IDENT_ACTFUNC, 0, DT_ACT_TRUNC, DT_ATTR_STABCMN,
        DT_VERS_1_0, &dt_idops_func, "void(...)" },
{ "uaddr", DT_IDENT_ACTFUNC, 0, DT_ACT_UADDR, DT_ATTR_STABCMN,
        DT_VERS_1_2, &dt_idops_func, "_usymaddr(uintptr_t)" },
{ "ucaller", DT_IDENT_SCALAR, 0, DIF_VAR_UCALLER, DT_ATTR_STABCMN,
        DT_VERS_1_2, &dt_idops_type, "uint64_t" },
{ "ufunc", DT_IDENT_ACTFUNC, 0, DT_ACT_USYM, DT_ATTR_STABCMN,
        DT_VERS_1_2, &dt_idops_func, "_usymaddr(uintptr_t)" },
{ "uid", DT_IDENT_SCALAR, 0, DIF_VAR_UID, DT_ATTR_STABCMN, DT_VERS_1_0,
        &dt_idops_type, "uid_t" },
{ "umod", DT_IDENT_ACTFUNC, 0, DT_ACT_UMOD, DT_ATTR_STABCMN,
        DT_VERS_1_2, &dt_idops_func, "_usymaddr(uintptr_t)" },
{ "uregs", DT_IDENT_ARRAY, 0, DIF_VAR_UREGS, DT_ATTR_STABCMN, DT_VERS_1_0,
        &dt_idops_regs, NULL },
{ "ustack", DT_IDENT_ACTFUNC, 0, DT_ACT_USTACK, DT_ATTR_STABCMN, DT_VERS_1_0,
        &dt_idops_func, "stack(...)" },
{ "ustackdepth", DT_IDENT_SCALAR, 0, DIF_VAR_USTACKDEPTH,
        DT_ATTR_STABCMN, DT_VERS_1_2,
        &dt_idops_type, "uint32_t" },
{ "usym", DT_IDENT_ACTFUNC, 0, DT_ACT_USYM, DT_ATTR_STABCMN,
        DT_VERS_1_2, &dt_idops_func, "_usymaddr(uintptr_t)" },
{ "vmregs", DT_IDENT_ARRAY, 0, DIF_VAR_VMREGS, DT_ATTR_STABCMN, DT_VERS_1_7,
        &dt_idops_regs, NULL },
{ "vtimestamp", DT_IDENT_SCALAR, 0, DIF_VAR_VTIMESTAMP,
        DT_ATTR_STABCMN, DT_VERS_1_0,
        &dt_idops_type, "uint64_t" },
{ "walltimestamp", DT_IDENT_SCALAR, 0, DIF_VAR_WALLTIMESTAMP,
        DT_ATTR_STABCMN, DT_VERS_1_0,
        &dt_idops_type, "int64_t" },
{ "zonename", DT_IDENT_SCALAR, 0, DIF_VAR_ZONENAME,
        DT_ATTR_STABCMN, DT_VERS_1_0, &dt_idops_type, "string" },
{ NULL, 0, 0, 0, { 0, 0, 0 }, 0, NULL, NULL }
};

/*
 * Tables of ILP32 intrinsic integer and floating-point type templates to use
 * to populate the dynamic "C" CTF type container.
 */
static const dt_intrinsic_t _dtrace_intrinsics_32[] = {
{ "void", { CTF_INT_SIGNED, 0, 0 }, CTF_K_INTEGER },
{ "signed", { CTF_INT_SIGNED, 0, 32 }, CTF_K_INTEGER },
{ "unsigned", { 0, 0, 32 }, CTF_K_INTEGER },
{ "char", { CTF_INT_SIGNED | CTF_INT_CHAR, 0, 8 }, CTF_K_INTEGER },
{ "short", { CTF_INT_SIGNED, 0, 16 }, CTF_K_INTEGER },
{ "int", { CTF_INT_SIGNED, 0, 32 }, CTF_K_INTEGER },
{ "long", { CTF_INT_SIGNED, 0, 32 }, CTF_K_INTEGER },
{ "long long", { CTF_INT_SIGNED, 0, 64 }, CTF_K_INTEGER },
{ "signed char", { CTF_INT_SIGNED | CTF_INT_CHAR, 0, 8 }, CTF_K_INTEGER },
{ "signed short", { CTF_INT_SIGNED, 0, 16 }, CTF_K_INTEGER },
{ "signed int", { CTF_INT_SIGNED, 0, 32 }, CTF_K_INTEGER },
{ "signed long", { CTF_INT_SIGNED, 0, 32 }, CTF_K_INTEGER },
{ "signed long long", { CTF_INT_SIGNED, 0, 64 }, CTF_K_INTEGER },
{ "unsigned char", { CTF_INT_CHAR, 0, 8 }, CTF_K_INTEGER },
```

```
{ "unsigned short", { 0, 0, 16 }, CTF_K_INTEGER },
{ "unsigned int", { 0, 0, 32 }, CTF_K_INTEGER },
{ "unsigned long", { 0, 0, 32 }, CTF_K_INTEGER },
{ "unsigned long long", { 0, 0, 64 }, CTF_K_INTEGER },
{ "_Bool", { CTF_INT_BOOL, 0, 8 }, CTF_K_INTEGER },
{ "float", { CTF_FP_SINGLE, 0, 32 }, CTF_K_FLOAT },
{ "double", { CTF_FP_DOUBLE, 0, 64 }, CTF_K_FLOAT },
{ "long double", { CTF_FP_LDOUBLE, 0, 128 }, CTF_K_FLOAT },
{ "float imaginary", { CTF_FP_IMAGRY, 0, 32 }, CTF_K_FLOAT },
{ "double imaginary", { CTF_FP_DIMAGRY, 0, 64 }, CTF_K_FLOAT },
{ "long double imaginary", { CTF_FP_LDIMAGRY, 0, 128 }, CTF_K_FLOAT },
{ "float complex", { CTF_FP_CPLX, 0, 64 }, CTF_K_FLOAT },
{ "double complex", { CTF_FP_DCPLX, 0, 128 }, CTF_K_FLOAT },
{ "long double complex", { CTF_FP_LDCPLX, 0, 256 }, CTF_K_FLOAT },
{ NULL, { 0, 0, 0 }, 0 }
};

/*
 * Tables of LP64 intrinsic integer and floating-point type templates to use
 * to populate the dynamic "C" CTF type container.
 */
static const dt_intrinsic_t _dtrace_intrinsics_64[] = {
{ "void", { CTF_INT_SIGNED, 0, 0 }, CTF_K_INTEGER },
{ "signed", { CTF_INT_SIGNED, 0, 32 }, CTF_K_INTEGER },
{ "unsigned", { 0, 0, 32 }, CTF_K_INTEGER },
{ "char", { CTF_INT_SIGNED | CTF_INT_CHAR, 0, 8 }, CTF_K_INTEGER },
{ "short", { CTF_INT_SIGNED, 0, 16 }, CTF_K_INTEGER },
{ "int", { CTF_INT_SIGNED, 0, 32 }, CTF_K_INTEGER },
{ "long", { CTF_INT_SIGNED, 0, 64 }, CTF_K_INTEGER },
{ "long long", { CTF_INT_SIGNED, 0, 64 }, CTF_K_INTEGER },
{ "signed char", { CTF_INT_SIGNED | CTF_INT_CHAR, 0, 8 }, CTF_K_INTEGER },
{ "signed short", { CTF_INT_SIGNED, 0, 16 }, CTF_K_INTEGER },
{ "signed int", { CTF_INT_SIGNED, 0, 32 }, CTF_K_INTEGER },
{ "signed long", { CTF_INT_SIGNED, 0, 64 }, CTF_K_INTEGER },
{ "signed long long", { CTF_INT_SIGNED, 0, 64 }, CTF_K_INTEGER },
{ "unsigned char", { CTF_INT_CHAR, 0, 8 }, CTF_K_INTEGER },
{ "unsigned short", { 0, 0, 16 }, CTF_K_INTEGER },
{ "unsigned int", { 0, 0, 32 }, CTF_K_INTEGER },
{ "unsigned long", { 0, 0, 64 }, CTF_K_INTEGER },
{ "unsigned long long", { 0, 0, 64 }, CTF_K_INTEGER },
{ "_Bool", { CTF_INT_BOOL, 0, 8 }, CTF_K_INTEGER },
{ "float", { CTF_FP_SINGLE, 0, 32 }, CTF_K_FLOAT },
{ "double", { CTF_FP_DOUBLE, 0, 64 }, CTF_K_FLOAT },
{ "long double", { CTF_FP_LDOUBLE, 0, 128 }, CTF_K_FLOAT },
{ "float imaginary", { CTF_FP_IMAGRY, 0, 32 }, CTF_K_FLOAT },
{ "double imaginary", { CTF_FP_DIMAGRY, 0, 64 }, CTF_K_FLOAT },
{ "long double imaginary", { CTF_FP_LDIMAGRY, 0, 128 }, CTF_K_FLOAT },
{ "float complex", { CTF_FP_CPLX, 0, 64 }, CTF_K_FLOAT },
{ "double complex", { CTF_FP_DCPLX, 0, 128 }, CTF_K_FLOAT },
{ "long double complex", { CTF_FP_LDCPLX, 0, 256 }, CTF_K_FLOAT },
{ NULL, { 0, 0, 0 }, 0 }
};

/*
 * Tables of ILP32 typedefs to use to populate the dynamic "D" CTF container.
 * These aliases ensure that D definitions can use typical <sys/types.h> names.
 */
static const dt_typedef_t _dtrace_typedefs_32[] = {
{ "char", "int8_t" },
{ "short", "int16_t" },
{ "int", "int32_t" },
{ "long long", "int64_t" },
{ "int", "intptr_t" },
{ "int", "ssize_t" },
{ "unsigned char", "uint8_t" },
{ "unsigned short", "uint16_t" },
```



```
{ "unsigned", "uint32_t" },
{ "unsigned long long", "uint64_t" },
{ "unsigned char", "uchar_t" },
{ "unsigned short", "ushort_t" },
{ "unsigned", "uint_t" },
{ "unsigned long", "ulong_t" },
{ "unsigned long long", "u_longlong_t" },
{ "int", "ptrdiff_t" },
{ "unsigned", "uintptr_t" },
{ "unsigned", "size_t" },
{ "long", "id_t" },
{ "long", "pid_t" },
{ NULL, NULL }
};

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

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

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

```
{ "unsigned long long", NULL, CTF_ERR, 0xffffffffffffffffULL }
};

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

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

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

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

const dtrace_attribute_t _dtrace_defattr = {
        DTRACE_STABILITY_STABLE,
        DTRACE_STABILITY_STABLE,
        DTRACE_CLASS_COMMON
};

const dtrace_attribute_t _dtrace_symattr = {
        DTRACE_STABILITY_PRIVATE,
        DTRACE_STABILITY_PRIVATE,
        DTRACE_CLASS_UNKNOWN
};

const dtrace_attribute_t _dtrace_typattr = {
```



```
        DTRACE_STABILITY_PRIVATE,
        DTRACE_STABILITY_PRIVATE,
        DTRACE_CLASS_UNKNOWN
};

const dtrace_attribute_t _dtrace_prvattr = {
        DTRACE_STABILITY_PRIVATE,
        DTRACE_STABILITY_PRIVATE,
        DTRACE_CLASS_UNKNOWN
};

const dtrace_pattr_t _dtrace_prvdesc = {
{ DTRACE_STABILITY_UNSTABLE, DTRACE_STABILITY_UNSTABLE, DTRACE_CLASS_COMMON },
{ DTRACE_STABILITY_UNSTABLE, DTRACE_STABILITY_UNSTABLE, DTRACE_CLASS_COMMON },
{ DTRACE_STABILITY_UNSTABLE, DTRACE_STABILITY_UNSTABLE, DTRACE_CLASS_COMMON },
{ DTRACE_STABILITY_UNSTABLE, DTRACE_STABILITY_UNSTABLE, DTRACE_CLASS_COMMON },
{ DTRACE_STABILITY_UNSTABLE, DTRACE_STABILITY_UNSTABLE, DTRACE_CLASS_COMMON },
};

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

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

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

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

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

#pragma init(_dtrace_init)
void
_dtrace_init(void)
{
        _dtrace_debug = getenv("DTRACE_DEBUG") != NULL;

        for (; _dtrace_rdvers > 0; _dtrace_rdvers--) {
                if (rd_init(_dtrace_rdvers) == RD_OK)
                        break;
        }
}

static dtrace_hdl_t *
set_open_errno(dtrace_hdl_t *dtp, int *errp, int err)
{
        if (dtp != NULL)
                dtrace_close(dtp);
        if (errp != NULL)
                *errp = err;
        return (NULL);
}
```

```
732 static void
733 dt_provmod_open(dt_provmod_t **provmod, dt_fdlist_t *dfp)
734 {
735         dt_provmod_t *prov;
736         char path[PATH_MAX];
737         struct dirent *dp, *ep;
738         DIR *dirp;
739         int fd;

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

744         ep = alloca(sizeof (struct dirent) + PATH_MAX + 1);
745         bzero(ep, sizeof (struct dirent) + PATH_MAX + 1);

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

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

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

758                         dfp->df_fds = fds;
759                         dfp->df_size = size;
760                 }

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

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

768                 if (((prov = malloc(sizeof (dt_provmod_t))) == NULL) ||
769                     (prov->dp_name = malloc(strlen(dp->d_name) + 1)) == NULL) {
770                         free(prov);
771                         (void) close(fd);
772                         break;
773                 }

775                 (void) strcpy(prov->dp_name, dp->d_name);
776                 prov->dp_next = *provmod;
777                 *provmod = prov;

779                 dt_dprintf("opened provider %s\n", dp->d_name);
780                 dfp->df_fds[dfp->df_ents++] = fd;
781         }

783         (void) closedir(dirp);
784 }

786 static void
787 dt_provmod_destroy(dt_provmod_t **provmod)
788 {
789         dt_provmod_t *next, *current;

791         for (current = *provmod; current != NULL; current = next) {
792                 next = current->dp_next;
793                 free(current->dp_name);
794                 free(current);
795         }

797         *provmod = NULL;
```



```
798 }

800 static const char *
801 dt_get_sysinfo(int cmd, char *buf, size_t len)
802 {
803         ssize_t rv = sysinfo(cmd, buf, len);
804         char *p = buf;

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

809         while ((p = strchr(p, '.')) != NULL)
810                 *p++ = '_';

812         return (buf);
813 }

815 static dtrace_hdl_t *
816 dt_vopen(int version, int flags, int *errp,
817     const dtrace_vector_t *vector, void *arg)
818 {
819         dtrace_hdl_t *dtp = NULL;
820         int dtfd = -1, ftfd = -1, fterr = 0;
821         dtrace_prog_t *pgp;
822         dt_module_t *dmp;
823         dt_provmod_t *provmod = NULL;
824         int i, err;
825         struct rlimit rl;

827         const dt_intrinsic_t *dinp;
828         const dt_typedef_t *dtyp;
829         const dt_ident_t *idp;

831         dtrace_typeinfo_t dtt;
832         ctf_funcinfo_t ctc;
833         ctf_arinfo_t ctr;

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

837         char isadef[32], utsdef[32];
838         char s1[64], s2[64];

840         if (version <= 0)
841                 return (set_open_errno(dtp, errp, EINVAL));

843         if (version > DTRACE_VERSION)
844                 return (set_open_errno(dtp, errp, EDT_VERSION));

846         if (version < DTRACE_VERSION) {
847                 /*
848                  * Currently, increasing the library version number is used to
849                  * denote a binary incompatible change.  That is, a consumer
850                  * of the library cannot run on a version of the library with
851                  * a higher DTRACE_VERSION number than the consumer compiled
852                  * against.  Once the library API has been committed to,
853                  * backwards binary compatibility will be required; at that
854                  * time, this check should change to return EDT_OVERSION only
855                  * if the specified version number is less than the version
856                  * number at the time of interface commitment.
857                  */
858                 return (set_open_errno(dtp, errp, EDT_OVERSION));
859         }

861         if (flags & ~DTRACE_O_MASK)
862                 return (set_open_errno(dtp, errp, EINVAL));
```

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

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

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

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

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

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

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

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

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

        free(df.df_fds);

        /*
         * If we failed to open the dtrace device, fail dtrace_open().
         * We convert some kernel errnos to custom libdtrace errnos to
         * improve the resulting message from the usual strerror().
         */
        if (dtfd == -1) {
                dt_provmod_destroy(&provmod);
                switch (err) {
                case ENOENT:
                        err = EDT_NOENT;
                        break;
                case EBUSY:
                        err = EDT_BUSY;
                        break;
                case EACCES:
                        err = EDT_ACCESS;
                        break;
                }
                return (set_open_errno(dtp, errp, err));
        }
```



```
        (void) fcntl(dtfd, F_SETFD, FD_CLOEXEC);
        (void) fcntl(ftfd, F_SETFD, FD_CLOEXEC);

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

        bzero(dtp, sizeof (dtrace_hdl_t));
        dtp->dt_oflags = flags;
        dtp->dt_prcmode = DT_PROC_STOP_PREINIT;
        dtp->dt_linkmode = DT_LINK_KERNEL;
        dtp->dt_linktype = DT_LTYP_ELF;
        dtp->dt_xlatemode = DT_XL_STATIC;
        dtp->dt_stdcmode = DT_STDC_XA;
        dtp->dt_version = version;
        dtp->dt_fd = dtfd;
        dtp->dt_ftfd = ftfd;
        dtp->dt_fterr = fterr;
        dtp->dt_cdefs_fd = -1;
        dtp->dt_ddefs_fd = -1;
        dtp->dt_stdout_fd = -1;
        dtp->dt_modbuckets = _dtrace_strbuckets;
        dtp->dt_mods = calloc(dtp->dt_modbuckets, sizeof (dt_module_t *));
        dtp->dt_provbuckets = _dtrace_strbuckets;
        dtp->dt_provs = calloc(dtp->dt_provbuckets, sizeof (dt_provider_t *));
        dt_proc_init(dtp);
        dtp->dt_vmax = DT_VERS_LATEST;
        dtp->dt_cpp_path = strdup(_dtrace_defcpp);
        dtp->dt_cpp_argv = malloc(sizeof (char *));
        dtp->dt_cpp_argc = 1;
        dtp->dt_cpp_args = 1;
        dtp->dt_ld_path = strdup(_dtrace_defld);
        dtp->dt_provmod = provmod;
        dtp->dt_vector = vector;
        dtp->dt_varg = arg;
        dt_dof_init(dtp);
        (void) uname(&dtp->dt_uts);

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

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

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

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

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

        if (dt_cpp_add_arg(dtp, "-D__sun") == NULL ||
            dt_cpp_add_arg(dtp, "-D__unix") == NULL ||
            dt_cpp_add_arg(dtp, "-D__SVR4") == NULL ||
            dt_cpp_add_arg(dtp, "-D__SUNW_D=1") == NULL ||
            dt_cpp_add_arg(dtp, isadef) == NULL ||
            dt_cpp_add_arg(dtp, utsdef) == NULL)
                return (set_open_errno(dtp, errp, EDT_NOMEM));

        if (flags & DTRACE_O_NODEV)
                bcopy(&_dtrace_conf, &dtp->dt_conf, sizeof (_dtrace_conf));
```

```
 996         else if (dt_ioctl(dtp, DTRACEIOC_CONF, &dtp->dt_conf) != 0)
 997                 return (set_open_errno(dtp, errp, errno));

 999         if (flags & DTRACE_O_LP64)
1000                 dtp->dt_conf.dtc_ctfmodel = CTF_MODEL_LP64;
1001         else if (flags & DTRACE_O_ILP32)
1002                 dtp->dt_conf.dtc_ctfmodel = CTF_MODEL_ILP32;

1004 #ifdef __sparc
1005         /*
1006          * On SPARC systems, __sparc is always defined for <sys/isa_defs.h>
1007          * and __sparcv9 is defined if we are doing a 64-bit compile.
1008          */
1009         if (dt_cpp_add_arg(dtp, "-D__sparc") == NULL)
1010                 return (set_open_errno(dtp, errp, EDT_NOMEM));

1012         if (dtp->dt_conf.dtc_ctfmodel == CTF_MODEL_LP64 &&
1013             dt_cpp_add_arg(dtp, "-D__sparcv9") == NULL)
1014                 return (set_open_errno(dtp, errp, EDT_NOMEM));
1015 #endif

1017 #ifdef __x86
1018         /*
1019          * On x86 systems, __i386 is defined for <sys/isa_defs.h> for 32-bit
1020          * compiles and __amd64 is defined for 64-bit compiles.  Unlike SPARC,
1021          * they are defined exclusive of one another (see PSARC 2004/619).
1022          */
1023         if (dtp->dt_conf.dtc_ctfmodel == CTF_MODEL_LP64) {
1024                 if (dt_cpp_add_arg(dtp, "-D__amd64") == NULL)
1025                         return (set_open_errno(dtp, errp, EDT_NOMEM));
1026         } else {
1027                 if (dt_cpp_add_arg(dtp, "-D__i386") == NULL)
1028                         return (set_open_errno(dtp, errp, EDT_NOMEM));
1029         }
1030 #endif

1032         if (dtp->dt_conf.dtc_difversion < DIF_VERSION)
1033                 return (set_open_errno(dtp, errp, EDT_DIFVERS));

1035         if (dtp->dt_conf.dtc_ctfmodel == CTF_MODEL_ILP32)
1036                 bcopy(_dtrace_ints_32, dtp->dt_ints, sizeof (_dtrace_ints_32));
1037         else
1038                 bcopy(_dtrace_ints_64, dtp->dt_ints, sizeof (_dtrace_ints_64));

1040         dtp->dt_macros = dt_idhash_create("macro", NULL, 0, UINT_MAX);
1041         dtp->dt_aggs = dt_idhash_create("aggregation", NULL,
1042             DTRACE_AGGVARIDNONE + 1, UINT_MAX);

1044         dtp->dt_globals = dt_idhash_create("global", _dtrace_globals,
1045             DIF_VAR_OTHER_UBASE, DIF_VAR_OTHER_MAX);

1047         dtp->dt_tls = dt_idhash_create("thread local", NULL,
1048             DIF_VAR_OTHER_UBASE, DIF_VAR_OTHER_MAX);

1050         if (dtp->dt_macros == NULL || dtp->dt_aggs == NULL ||
1051             dtp->dt_globals == NULL || dtp->dt_tls == NULL)
1052                 return (set_open_errno(dtp, errp, EDT_NOMEM));

1054         /*
1055          * Populate the dt_macros identifier hash table by hand: we can't use
1056          * the dt_idhash_populate() mechanism because we're not yet compiling
1057          * and dtrace_update() needs to immediately reference these idents.
1058          */
1059         for (idp = _dtrace_macros; idp->di_name != NULL; idp++) {
1060                 if (dt_idhash_insert(dtp->dt_macros, idp->di_name,
1061                     idp->di_kind, idp->di_flags, idp->di_id, idp->di_attr,
```



```
1062                     idp->di_vers, idp->di_ops ? idp->di_ops : &dt_idops_thaw,
1063                     idp->di_iarg, 0) == NULL)
1064                         return (set_open_errno(dtp, errp, EDT_NOMEM));
1065         }

1067         /*
1068          * Update the module list using /system/object and load the values for
1069          * the macro variable definitions according to the current process.
1070          */
1071         dtrace_update(dtp);

1073         /*
1074          * Select the intrinsics and typedefs we want based on the data model.
1075          * The intrinsics are under "C".  The typedefs are added under "D".
1076          */
1077         if (dtp->dt_conf.dtc_ctfmodel == CTF_MODEL_ILP32) {
1078                 dinp = _dtrace_intrinsics_32;
1079                 dtyp = _dtrace_typedefs_32;
1080         } else {
1081                 dinp = _dtrace_intrinsics_64;
1082                 dtyp = _dtrace_typedefs_64;
1083         }

1085         /*
1086          * Create a dynamic CTF container under the "C" scope for intrinsic
1087          * types and types defined in ANSI-C header files that are included.
1088          */
1089         if ((dmp = dtp->dt_cdefs = dt_module_create(dtp, "C")) == NULL)
1090                 return (set_open_errno(dtp, errp, EDT_NOMEM));

1092         if ((dmp->dm_ctfp = ctf_create(&dtp->dt_ctferr)) == NULL)
1093                 return (set_open_errno(dtp, errp, EDT_CTF));

1095         dt_dprintf("created CTF container for %s (%p)\n",
1096             dmp->dm_name, (void *)dmp->dm_ctfp);

1098         (void) ctf_setmodel(dmp->dm_ctfp, dtp->dt_conf.dtc_ctfmodel);
1099         ctf_setspecific(dmp->dm_ctfp, dmp);

1101         dmp->dm_flags = DT_DM_LOADED; /* fake up loaded bit */
1102         dmp->dm_modid = -1; /* no module ID */

1104         /*
1105          * Fill the dynamic "C" CTF container with all of the intrinsic
1106          * integer and floating-point types appropriate for this data model.
1107          */
1108         for (; dinp->din_name != NULL; dinp++) {
1109                 if (dinp->din_kind == CTF_K_INTEGER) {
1110                         err = ctf_add_integer(dmp->dm_ctfp, CTF_ADD_ROOT,
1111                             dinp->din_name, &dinp->din_data);
1112                 } else {
1113                         err = ctf_add_float(dmp->dm_ctfp, CTF_ADD_ROOT,
1114                             dinp->din_name, &dinp->din_data);
1115                 }

1117                 if (err == CTF_ERR) {
1118                         dt_dprintf("failed to add %s to C container: %s\n",
1119                             dinp->din_name, ctf_errmsg(
1120                             ctf_errno(dmp->dm_ctfp)));
1121                         return (set_open_errno(dtp, errp, EDT_CTF));
1122                 }
1123         }

1125         if (ctf_update(dmp->dm_ctfp) != 0) {
1126                 dt_dprintf("failed to update C container: %s\n",
1127                     ctf_errmsg(ctf_errno(dmp->dm_ctfp)));
```

```
1128                return (set_open_errno(dtp, errp, EDT_CTF));
1129        }
1130
1131        /*
1132         * Add intrinsic pointer types that are needed to initialize printf
1133         * format dictionary types (see table in dt_printf.c).
1134         */
1135        (void) ctf_add_pointer(dmp->dm_ctfp, CTF_ADD_ROOT,
1136            ctf_lookup_by_name(dmp->dm_ctfp, "void"));
1137
1138        (void) ctf_add_pointer(dmp->dm_ctfp, CTF_ADD_ROOT,
1139            ctf_lookup_by_name(dmp->dm_ctfp, "char"));
1140
1141        (void) ctf_add_pointer(dmp->dm_ctfp, CTF_ADD_ROOT,
1142            ctf_lookup_by_name(dmp->dm_ctfp, "int"));
1143
1144        if (ctf_update(dmp->dm_ctfp) != 0) {
1145                dt_dprintf("failed to update C container: %s\n",
1146                    ctf_errmsg(ctf_errno(dmp->dm_ctfp)));
1147                return (set_open_errno(dtp, errp, EDT_CTF));
1148        }
1149
1150        /*
1151         * Create a dynamic CTF container under the "D" scope for types that
1152         * are defined by the D program itself or on-the-fly by the D compiler.
1153         * The "D" CTF container is a child of the "C" CTF container.
1154         */
1155        if ((dmp = dtp->dt_ddefs = dt_module_create(dtp, "D")) == NULL)
1156                return (set_open_errno(dtp, errp, EDT_NOMEM));
1157
1158        if ((dmp->dm_ctfp = ctf_create(&dtp->dt_ctferr)) == NULL)
1159                return (set_open_errno(dtp, errp, EDT_CTF));
1160
1161        dt_dprintf("created CTF container for %s (%p)\n",
1162            dmp->dm_name, (void *)dmp->dm_ctfp);
1163
1164        (void) ctf_setmodel(dmp->dm_ctfp, dtp->dt_conf.dtc_ctfmodel);
1165        ctf_setspecific(dmp->dm_ctfp, dmp);
1166
1167        dmp->dm_flags = DT_DM_LOADED; /* fake up loaded bit */
1168        dmp->dm_modid = -1; /* no module ID */
1169
1170        if (ctf_import(dmp->dm_ctfp, dtp->dt_cdefs->dm_ctfp) == CTF_ERR) {
1171                dt_dprintf("failed to import D parent container: %s\n",
1172                    ctf_errmsg(ctf_errno(dmp->dm_ctfp)));
1173                return (set_open_errno(dtp, errp, EDT_CTF));
1174        }
1175
1176        /*
1177         * Fill the dynamic "D" CTF container with all of the built-in typedefs
1178         * that we need to use for our D variable and function definitions.
1179         * This ensures that basic inttypes.h names are always available to us.
1180         */
1181        for (; dtyp->dty_src != NULL; dtyp++) {
1182                if (ctf_add_typedef(dmp->dm_ctfp, CTF_ADD_ROOT,
1183                    dtyp->dty_dst, ctf_lookup_by_name(dmp->dm_ctfp,
1184                    dtyp->dty_src)) == CTF_ERR) {
1185                        dt_dprintf("failed to add typedef %s %s to D "
1186                            "container: %s", dtyp->dty_src, dtyp->dty_dst,
1187                            ctf_errmsg(ctf_errno(dmp->dm_ctfp)));
1188                        return (set_open_errno(dtp, errp, EDT_CTF));
1189                }
1190        }
1191
1192        /*
1193         * Insert a CTF ID corresponding to a pointer to a type of kind
```



```
1194         * CTF_K_FUNCTION we can use in the compiler for function pointers.
1195         * CTF treats all function pointers as "int (*)()" so we only need one.
1196         */
1197        ctc.ctc_return = ctf_lookup_by_name(dmp->dm_ctfp, "int");
1198        ctc.ctc_argc = 0;
1199        ctc.ctc_flags = 0;
1200
1201        dtp->dt_type_func = ctf_add_function(dmp->dm_ctfp,
1202            CTF_ADD_ROOT, &ctc, NULL);
1203
1204        dtp->dt_type_fptr = ctf_add_pointer(dmp->dm_ctfp,
1205            CTF_ADD_ROOT, dtp->dt_type_func);
1206
1207        /*
1208         * We also insert CTF definitions for the special D intrinsic types
1209         * string and <DYN> into the D container.  The string type is added
1210         * as a typedef of char[n].  The <DYN> type is an alias for void.
1211         * We compare types to these special CTF ids throughout the compiler.
1212         */
1213        ctr.ctr_contents = ctf_lookup_by_name(dmp->dm_ctfp, "char");
1214        ctr.ctr_index = ctf_lookup_by_name(dmp->dm_ctfp, "long");
1215        ctr.ctr_nelems = _dtrace_strsize;
1216
1217        dtp->dt_type_str = ctf_add_typedef(dmp->dm_ctfp, CTF_ADD_ROOT,
1218            "string", ctf_add_array(dmp->dm_ctfp, CTF_ADD_ROOT, &ctr));
1219
1220        dtp->dt_type_dyn = ctf_add_typedef(dmp->dm_ctfp, CTF_ADD_ROOT,
1221            "<DYN>", ctf_lookup_by_name(dmp->dm_ctfp, "void"));
1222
1223        dtp->dt_type_stack = ctf_add_typedef(dmp->dm_ctfp, CTF_ADD_ROOT,
1224            "stack", ctf_lookup_by_name(dmp->dm_ctfp, "void"));
1225
1226        dtp->dt_type_symaddr = ctf_add_typedef(dmp->dm_ctfp, CTF_ADD_ROOT,
1227            "_symaddr", ctf_lookup_by_name(dmp->dm_ctfp, "void"));
1228
1229        dtp->dt_type_usymaddr = ctf_add_typedef(dmp->dm_ctfp, CTF_ADD_ROOT,
1230            "_usymaddr", ctf_lookup_by_name(dmp->dm_ctfp, "void"));
1231
1232        if (dtp->dt_type_func == CTF_ERR || dtp->dt_type_fptr == CTF_ERR ||
1233            dtp->dt_type_str == CTF_ERR || dtp->dt_type_dyn == CTF_ERR ||
1234            dtp->dt_type_stack == CTF_ERR || dtp->dt_type_symaddr == CTF_ERR ||
1235            dtp->dt_type_usymaddr == CTF_ERR) {
1236                dt_dprintf("failed to add intrinsic to D container: %s\n",
1237                    ctf_errmsg(ctf_errno(dmp->dm_ctfp)));
1238                return (set_open_errno(dtp, errp, EDT_CTF));
1239        }
1240
1241        if (ctf_update(dmp->dm_ctfp) != 0) {
1242                dt_dprintf("failed update D container: %s\n",
1243                    ctf_errmsg(ctf_errno(dmp->dm_ctfp)));
1244                return (set_open_errno(dtp, errp, EDT_CTF));
1245        }
1246
1247        /*
1248         * Initialize the integer description table used to convert integer
1249         * constants to the appropriate types.  Refer to the comments above
1250         * dt_node_int() for a complete description of how this table is used.
1251         */
1252        for (i = 0; i < sizeof (dtp->dt_ints) / sizeof (dtp->dt_ints[0]); i++) {
1253                if (dtrace_lookup_by_type(dtp, DTRACE_OBJ_EVERY,
1254                    dtp->dt_ints[i].did_name, &dtt) != 0) {
1255                        dt_dprintf("failed to lookup integer type %s: %s\n",
1256                            dtp->dt_ints[i].did_name,
1257                            dtrace_errmsg(dtp, dtrace_errno(dtp)));
1258                        return (set_open_errno(dtp, errp, dtp->dt_errno));
1259                }
```

```
1260                 dtp->dt_ints[i].did_ctfp = dtt.dtt_ctfp;
1261                 dtp->dt_ints[i].did_type = dtt.dtt_type;
1262         }

1264         /*
1265          * Now that we've created the "C" and "D" containers, move them to the
1266          * start of the module list so that these types and symbols are found
1267          * first (for stability) when iterating through the module list.
1268          */
1269         dt_list_delete(&dtp->dt_modlist, dtp->dt_ddefs);
1270         dt_list_prepend(&dtp->dt_modlist, dtp->dt_ddefs);

1272         dt_list_delete(&dtp->dt_modlist, dtp->dt_cdefs);
1273         dt_list_prepend(&dtp->dt_modlist, dtp->dt_cdefs);

1275         if (dt_pfdict_create(dtp) == -1)
1276                 return (set_open_errno(dtp, errp, dtp->dt_errno));

1278         /*
1279          * If we are opening libdtrace DTRACE_O_NODEV enable C_ZDEFS by default
1280          * because without /dev/dtrace open, we will not be able to load the
1281          * names and attributes of any providers or probes from the kernel.
1282          */
1283         if (flags & DTRACE_O_NODEV)
1284                 dtp->dt_cflags |= DTRACE_C_ZDEFS;

1286         /*
1287          * Load hard-wired inlines into the definition cache by calling the
1288          * compiler on the raw definition string defined above.
1289          */
1290         if ((pgp = dtrace_program_strcompile(dtp, _dtrace_hardwire,
1291             DTRACE_PROBESPEC_NONE, DTRACE_C_EMPTY, 0, NULL)) == NULL) {
1292                 dt_dprintf("failed to load hard-wired definitions: %s\n",
1293                     dtrace_errmsg(dtp, dtrace_errno(dtp)));
1294                 return (set_open_errno(dtp, errp, EDT_HARDWIRE));
1295         }

1297         dt_program_destroy(dtp, pgp);

1299         /*
1300          * Set up the default DTrace library path.  Once set, the next call to
1301          * dt_compile() will compile all the libraries.  We intentionally defer
1302          * library processing to improve overhead for clients that don't ever
1303          * compile, and to provide better error reporting (because the full
1304          * reporting of compiler errors requires dtrace_open() to succeed).
1305          */
1306         if (dtrace_setopt(dtp, "libdir", _dtrace_libdir) != 0)
1307                 return (set_open_errno(dtp, errp, dtp->dt_errno));

1309         return (dtp);
1310 }

1312 dtrace_hdl_t *
1313 dtrace_open(int version, int flags, int *errp)
1314 {
1315         return (dt_vopen(version, flags, errp, NULL, NULL));
1316 }

1318 dtrace_hdl_t *
1319 dtrace_vopen(int version, int flags, int *errp,
1320     const dtrace_vector_t *vector, void *arg)
1321 {
1322         return (dt_vopen(version, flags, errp, vector, arg));
1323 }

1325 void
```



```
1326 dtrace_close(dtrace_hdl_t *dtp)
1327 {
1328         dt_ident_t *idp, *ndp;
1329         dt_module_t *dmp;
1330         dt_provider_t *pvp;
1331         dtrace_prog_t *pgp;
1332         dt_xlator_t *dxp;
1333         dt_dirpath_t *dirp;
1334         int i;

1336         if (dtp->dt_procs != NULL)
1337                 dt_proc_fini(dtp);

1339         while ((pgp = dt_list_next(&dtp->dt_programs)) != NULL)
1340                 dt_program_destroy(dtp, pgp);

1342         while ((dxp = dt_list_next(&dtp->dt_xlators)) != NULL)
1343                 dt_xlator_destroy(dtp, dxp);

1345         dt_free(dtp, dtp->dt_xlatormap);

1347         for (idp = dtp->dt_externs; idp != NULL; idp = ndp) {
1348                 ndp = idp->di_next;
1349                 dt_ident_destroy(idp);
1350         }

1352         if (dtp->dt_macros != NULL)
1353                 dt_idhash_destroy(dtp->dt_macros);
1354         if (dtp->dt_aggs != NULL)
1355                 dt_idhash_destroy(dtp->dt_aggs);
1356         if (dtp->dt_globals != NULL)
1357                 dt_idhash_destroy(dtp->dt_globals);
1358         if (dtp->dt_tls != NULL)
1359                 dt_idhash_destroy(dtp->dt_tls);

1361         while ((dmp = dt_list_next(&dtp->dt_modlist)) != NULL)
1362                 dt_module_destroy(dtp, dmp);

1364         while ((pvp = dt_list_next(&dtp->dt_provlist)) != NULL)
1365                 dt_provider_destroy(dtp, pvp);

1367         if (dtp->dt_fd != -1)
1368                 (void) close(dtp->dt_fd);
1369         if (dtp->dt_ftfd != -1)
1370                 (void) close(dtp->dt_ftfd);
1371         if (dtp->dt_cdefs_fd != -1)
1372                 (void) close(dtp->dt_cdefs_fd);
1373         if (dtp->dt_ddefs_fd != -1)
1374                 (void) close(dtp->dt_ddefs_fd);
1375         if (dtp->dt_stdout_fd != -1)
1376                 (void) close(dtp->dt_stdout_fd);

1378         dt_epid_destroy(dtp);
1379         dt_aggid_destroy(dtp);
1380         dt_format_destroy(dtp);
1381         dt_strdata_destroy(dtp);
1382         dt_buffered_destroy(dtp);
1383         dt_aggregate_destroy(dtp);
1384         dt_pfdict_destroy(dtp);
1385         dt_provmod_destroy(&dtp->dt_provmod);
1386         dt_dof_fini(dtp);

1388         for (i = 1; i < dtp->dt_cpp_argc; i++)
1389                 free(dtp->dt_cpp_argv[i]);

1391         while ((dirp = dt_list_next(&dtp->dt_lib_path)) != NULL) {
```

```
1392                 dt_list_delete(&dtp->dt_lib_path, dirp);
1393                 free(dirp->dir_path);
1394                 free(dirp);
1395         }

1397         free(dtp->dt_cpp_argv);
1398         free(dtp->dt_cpp_path);
1399         free(dtp->dt_ld_path);

1401         free(dtp->dt_mods);
1402         free(dtp->dt_provs);
1403         free(dtp);
1404 }

1406 int
1407 dtrace_provider_modules(dtrace_hdl_t *dtp, const char **mods, int nmods)
1408 {
1409         dt_provmod_t *prov;
1410         int i = 0;

1412         for (prov = dtp->dt_provmod; prov != NULL; prov = prov->dp_next, i++) {
1413                 if (i < nmods)
1414                         mods[i] = prov->dp_name;
1415         }

1417         return (i);
1418 }

1420 int
1421 dtrace_ctlfd(dtrace_hdl_t *dtp)
1422 {
1423         return (dtp->dt_fd);
1424 }
```

```
**********************************************************
  120669 Tue Jan 14 16:49:35 2014
new/usr/src/pkg/manifests/system-dtrace-tests.mf
4477 DTrace should speak JSON
Reviewed by: Bryan Cantrill <bmc@joyent.com>
**********************************************************
```

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

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

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



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

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



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

```
file path=opt/SUNWdtrt/tst/common/aggs/tst.avg_neg.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.clear.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.clear.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.clearavg.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.clearavg.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.clearavg2.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.clearavg2.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.cleardenormalize.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.cleardenormalize.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.clearlquantize.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.clearlquantize.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.clearnormalize.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.clearnormalize.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.clearstddev.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.clearstddev.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.count.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.count.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.count2.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.count2.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.count3.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.denormalize.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.denormalize.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.denormalizeonly.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.denormalizeonly.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.fmtnormalize.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.fmtnormalize.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.forms.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.forms.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.goodkey.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.keysort.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.keysort.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.lquantize.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.lquantize.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.lquantnormal.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.lquantnormal.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.lquantrange.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.lquantrange.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.lquantround.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.lquantround.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.lquantzero.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.lquantzero.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.max.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.max.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.max_neg.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.max_neg.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.min.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.min.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.min_neg.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.min_neg.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.multiaggs1.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.multiaggs2.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.multiaggs2.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.multiaggs3.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.multiaggs3.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.multinormalize.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.multinormalize.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.neglquant.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.neglquant.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.negorder.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.negorder.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.negquant.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.negquant.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.negtrunc.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.negtrunc.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.negtruncquant.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.negtruncquant.d.out mode=0444
```



```
file path=opt/SUNWdtrt/tst/common/aggs/tst.normalize.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.normalize.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.order.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.order.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.quantize.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.quantize.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.quantmany.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.quantmany.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.quantround.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.quantround.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.quantzero.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.quantzero.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.signature.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.signedkeys.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.signedkeys.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.signedkeyspos.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.signedkeyspos.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.sizedkeys.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.sizedkeys.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.stddev.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.stddev.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.subr.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.sum.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.sum.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.trunc.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.trunc.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.trunc0.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.trunc0.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.truncquant.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.truncquant.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.valsortkeypos.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.valsortkeypos.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/arithmetic/err.D_DIV_ZERO.divby0.d mode=0444
file path=opt/SUNWdtrt/tst/common/arithmetic/err.D_DIV_ZERO.divby0_1.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/arithmetic/err.D_DIV_ZERO.divby0_2.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/arithmetic/err.D_DIV_ZERO.modby0.d mode=0444
file path=opt/SUNWdtrt/tst/common/arithmetic/err.D_SYNTAX.addmin.d mode=0444
file path=opt/SUNWdtrt/tst/common/arithmetic/err.D_SYNTAX.divmin.d mode=0444
file path=opt/SUNWdtrt/tst/common/arithmetic/err.D_SYNTAX.muladd.d mode=0444
file path=opt/SUNWdtrt/tst/common/arithmetic/err.D_SYNTAX.muldiv.d mode=0444
file path=opt/SUNWdtrt/tst/common/arithmetic/tst.basics.d mode=0444
file path=opt/SUNWdtrt/tst/common/arithmetic/tst.basics.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/arithmetic/tst.compcast.d mode=0444
file path=opt/SUNWdtrt/tst/common/arithmetic/tst.compcast.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/arithmetic/tst.compnarrowassign.d mode=0444
file path=opt/SUNWdtrt/tst/common/arithmetic/tst.compnarrowassign.d.out \
    mode=0444
file path=opt/SUNWdtrt/tst/common/arithmetic/tst.execcast.d mode=0444
file path=opt/SUNWdtrt/tst/common/arithmetic/tst.execcast.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/arrays/err.D_ARR_BADREF.bad.d mode=0444
file path=opt/SUNWdtrt/tst/common/arrays/err.D_DECL_ARRBIG.toobig.d mode=0444
file path=opt/SUNWdtrt/tst/common/arrays/err.D_DECL_ARRNULL.bad.d mode=0444
file path=opt/SUNWdtrt/tst/common/arrays/err.D_DECL_ARRSUB.bad.d mode=0444
file path=opt/SUNWdtrt/tst/common/arrays/err.D_DECL_PROTO_TYPE.badtuple.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/arrays/err.D_IDENT_UNDEF.badureg.d mode=0444
file path=opt/SUNWdtrt/tst/common/arrays/tst.basic1.d mode=0444
file path=opt/SUNWdtrt/tst/common/arrays/tst.basic2.d mode=0444
file path=opt/SUNWdtrt/tst/common/arrays/tst.basic3.d mode=0444
file path=opt/SUNWdtrt/tst/common/arrays/tst.basic4.d mode=0444
file path=opt/SUNWdtrt/tst/common/arrays/tst.basic5.d mode=0444
file path=opt/SUNWdtrt/tst/common/arrays/tst.basic6.d mode=0444
file path=opt/SUNWdtrt/tst/common/arrays/tst.uregsarray.d mode=0444
file path=opt/SUNWdtrt/tst/common/assocs/err.D_OP_INCOMPAT.dupgtype.d \
```

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



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

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



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

```
655 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceModule2.d.ksh \
656     mode=0444
657 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceModule3.d.ksh \
658     mode=0444
659 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceModule4.d.ksh \
660     mode=0444
661 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceModule5.d.ksh \
662     mode=0444
663 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceModule6.d.ksh \
664     mode=0444
665 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceModule7.d.ksh \
666     mode=0444
667 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceModule8.d.ksh \
668     mode=0444
669 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceName1.d.ksh \
670     mode=0444
671 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceName2.d.ksh \
672     mode=0444
673 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceName3.d.ksh \
674     mode=0444
675 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceName4.d.ksh \
676     mode=0444
677 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceName5.d.ksh \
678     mode=0444
679 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceName6.d.ksh \
680     mode=0444
681 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceName7.d.ksh \
682     mode=0444
683 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceName8.d.ksh \
684     mode=0444
685 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceName9.d.ksh \
686     mode=0444
687 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceProvider1.d.ksh \
688     mode=0444
689 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceProvider2.d.ksh \
690     mode=0444
691 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceProvider3.d.ksh \
692     mode=0444
693 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceProvider4.d.ksh \
694     mode=0444
695 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceProvider5.d.ksh \
696     mode=0444
697 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.MultipleInvalidProbeId.d.ksh \
698     mode=0444
699 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.PreprocessorStatement.d.ksh \
700     mode=0444
701 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.QuietMode.d.ksh mode=0444
702 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.QuietMode.d.ksh.out mode=0444
703 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.TestCompile.d.ksh mode=0444
704 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.TestCompile.d.ksh.out \
705     mode=0444
706 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.UnDefineNameWithCPP.d.ksh \
707     mode=0444
708 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.ZeroFunctionProbes.d.ksh \
709     mode=0444
710 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.ZeroFunctionProbes.d.ksh.out \
711     mode=0444
712 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.ZeroModuleProbes.d.ksh \
713     mode=0444
714 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.ZeroModuleProbes.d.ksh.out \
715     mode=0444
716 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.ZeroNameProbes.d.ksh \
717     mode=0444
718 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.ZeroNameProbes.d.ksh.out \
719     mode=0444
720 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.ZeroProbeIdentfier.d.ksh \
```



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

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



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

```
file path=opt/SUNWdtrt/tst/common/ip/tst.ipv4remotetcp.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/ip/tst.ipv4remotetcp.ksh.out mode=0444
file path=opt/SUNWdtrt/tst/common/ip/tst.ipv4remoteudp.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/ip/tst.ipv4remoteudp.ksh.out mode=0444
file path=opt/SUNWdtrt/tst/common/ip/tst.ipv6localicmp.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/ip/tst.ipv6localicmp.ksh.out mode=0444
file path=opt/SUNWdtrt/tst/common/ip/tst.ipv6remoteicmp.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/ip/tst.ipv6remoteicmp.ksh.out mode=0444
file path=opt/SUNWdtrt/tst/common/ip/tst.localtcpstate.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/ip/tst.localtcpstate.ksh.out mode=0444
file path=opt/SUNWdtrt/tst/common/ip/tst.remotetcpstate.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/ip/tst.remotetcpstate.ksh.out mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/test.jar
file path=opt/SUNWdtrt/tst/common/java_api/tst.Abort.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.Abort.ksh.out mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.Bean.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.Bean.ksh.out mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.Close.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.Close.ksh.out mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.Drop.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.Drop.ksh.out mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.Enable.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.Enable.ksh.out mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.FunctionLookup.exe mode=0555
file path=opt/SUNWdtrt/tst/common/java_api/tst.FunctionLookup.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.FunctionLookup.ksh.out \
    mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.GetAggregate.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.MaxConsumers.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.MaxConsumers.ksh.out mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.MultiAggPrinta.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.MultiAggPrinta.ksh.out \
    mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.ProbeData.exe mode=0555
file path=opt/SUNWdtrt/tst/common/java_api/tst.ProbeData.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.ProbeData.ksh.out mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.ProbeDescription.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.ProbeDescription.ksh.out \
    mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.StateMachine.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.StateMachine.ksh.out mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.StopLock.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.StopLock.ksh.out mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.printa.d mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.printa.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/json/tst.general.d mode=0444
file path=opt/SUNWdtrt/tst/common/json/tst.general.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/json/tst.strsize.d mode=0444
file path=opt/SUNWdtrt/tst/common/json/tst.strsize.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/json/tst.usdt.d mode=0444
file path=opt/SUNWdtrt/tst/common/json/tst.usdt.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/json/tst.usdt.exe mode=0555
#endif /* ! codereview */
file path=opt/SUNWdtrt/tst/common/lexer/err.D_CHR_NL.char.d mode=0444
file path=opt/SUNWdtrt/tst/common/lexer/err.D_CHR_NULL.char.d mode=0444
file path=opt/SUNWdtrt/tst/common/lexer/err.D_INT_DIGIT.InvalidDigit.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/lexer/err.D_INT_OFLOW.BigInt.d mode=0444
file path=opt/SUNWdtrt/tst/common/lexer/err.D_STR_NL.string.d mode=0444
file path=opt/SUNWdtrt/tst/common/lexer/err.D_SYNTAX.brace1.d mode=0444
file path=opt/SUNWdtrt/tst/common/lexer/err.D_SYNTAX.brace2.d mode=0444
file path=opt/SUNWdtrt/tst/common/lexer/err.D_SYNTAX.brack1.d mode=0444
file path=opt/SUNWdtrt/tst/common/lexer/err.D_SYNTAX.brack2.d mode=0444
file path=opt/SUNWdtrt/tst/common/lexer/err.D_SYNTAX.brack3.d mode=0444
file path=opt/SUNWdtrt/tst/common/lexer/err.D_SYNTAX.paren1.d mode=0444
file path=opt/SUNWdtrt/tst/common/lexer/err.D_SYNTAX.paren2.d mode=0444
```



```
file path=opt/SUNWdtrt/tst/common/lexer/err.D_SYNTAX.paren3.d mode=0444
file path=opt/SUNWdtrt/tst/common/lexer/tst.D_MACRO_OFLOW.ParIntOvflow.d.ksh \
    mode=0444
file \
    path=opt/SUNWdtrt/tst/common/llquantize/err.D_LLQUANT_FACTOREVEN.nodivide.d
    mode=0444
file \
    path=opt/SUNWdtrt/tst/common/llquantize/err.D_LLQUANT_FACTOREVEN.notfactor.d
    mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/err.D_LLQUANT_FACTORMATCH.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/err.D_LLQUANT_FACTORNSTEPS.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/err.D_LLQUANT_FACTORSMALL.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/err.D_LLQUANT_FACTORTYPE.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/err.D_LLQUANT_FACTORVAL.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/err.D_LLQUANT_HIGHMATCH.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/err.D_LLQUANT_HIGHTYPE.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/err.D_LLQUANT_HIGHVAL.d mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/err.D_LLQUANT_LOWMATCH.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/err.D_LLQUANT_LOWTYPE.d mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/err.D_LLQUANT_LOWVAL.d mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/err.D_LLQUANT_MAGRANGE.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/err.D_LLQUANT_MAGTOOBIG.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/err.D_LLQUANT_NSTEPMATCH.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/err.D_LLQUANT_NSTEPTYPE.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/err.D_LLQUANT_NSTEPVAL.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/tst.bases.d mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/tst.bases.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/tst.basic.d mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/tst.basic.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/tst.negorder.d mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/tst.negorder.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/tst.negvalue.d mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/tst.negvalue.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/tst.normal.d mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/tst.normal.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/tst.range.d mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/tst.range.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/tst.steps.d mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/tst.steps.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/tst.trunc.d mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/tst.trunc.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/mdb/tst.dtracedcmd.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/mib/tst.icmp.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/mib/tst.tcp.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/mib/tst.udp.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/misc/err.D_PRAGMA_OPTSET.d mode=0444
file path=opt/SUNWdtrt/tst/common/misc/tst.badopt.d mode=0444
file path=opt/SUNWdtrt/tst/common/misc/tst.boolopt.d mode=0444
file path=opt/SUNWdtrt/tst/common/misc/tst.boolopt.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/misc/tst.dynopt.d mode=0444
file path=opt/SUNWdtrt/tst/common/misc/tst.dynopt.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/misc/tst.enablerace.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/misc/tst.haslam.d mode=0444
```

```
file path=opt/SUNWdtrt/tst/common/misc/tst.include.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/misc/tst.macroglob.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/misc/tst.macroglob.ksh.out mode=0444
file path=opt/SUNWdtrt/tst/common/misc/tst.roch.d mode=0444
file path=opt/SUNWdtrt/tst/common/misc/tst.schrock.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/multiaggs/err.D_PRINTA_AGGKEY.d mode=0444
file path=opt/SUNWdtrt/tst/common/multiaggs/err.D_PRINTA_AGGPROTO.d mode=0444
file path=opt/SUNWdtrt/tst/common/multiaggs/tst.many.d mode=0444
file path=opt/SUNWdtrt/tst/common/multiaggs/tst.many.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/multiaggs/tst.same.d mode=0444
file path=opt/SUNWdtrt/tst/common/multiaggs/tst.same.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/multiaggs/tst.sort.d mode=0444
file path=opt/SUNWdtrt/tst/common/multiaggs/tst.sort.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/multiaggs/tst.sortpos.d mode=0444
file path=opt/SUNWdtrt/tst/common/multiaggs/tst.sortpos.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/multiaggs/tst.tuplecompat.d mode=0444
file path=opt/SUNWdtrt/tst/common/multiaggs/tst.tuplecompat.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/multiaggs/tst.zero.d mode=0444
file path=opt/SUNWdtrt/tst/common/multiaggs/tst.zero.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/multiaggs/tst.zero2.d mode=0444
file path=opt/SUNWdtrt/tst/common/multiaggs/tst.zero2.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/multiaggs/tst.zero3.d mode=0444
file path=opt/SUNWdtrt/tst/common/multiaggs/tst.zero3.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/nfs/tst.call.d mode=0444
file path=opt/SUNWdtrt/tst/common/nfs/tst.call.exe mode=0555
file path=opt/SUNWdtrt/tst/common/nfs/tst.call3.d mode=0444
file path=opt/SUNWdtrt/tst/common/nfs/tst.call3.exe mode=0555
file path=opt/SUNWdtrt/tst/common/offsetof/err.D_OFFSETOF_BITFIELD.bitfield.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/offsetof/err.D_OFFSETOF_TYPE.badtype.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/offsetof/err.D_OFFSETOF_TYPE.notsou.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/offsetof/err.D_UNKNOWN.OffsetofNULL.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/offsetof/err.D_UNKNOWN.badmemb.d mode=0444
file path=opt/SUNWdtrt/tst/common/offsetof/tst.OffsetofAlias.d mode=0444
file path=opt/SUNWdtrt/tst/common/offsetof/tst.OffsetofArith.d mode=0444
file path=opt/SUNWdtrt/tst/common/offsetof/tst.OffsetofUnion.d mode=0444
file path=opt/SUNWdtrt/tst/common/offsetof/tst.struct.d mode=0444
file path=opt/SUNWdtrt/tst/common/offsetof/tst.struct.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/offsetof/tst.union.d mode=0444
file path=opt/SUNWdtrt/tst/common/offsetof/tst.union.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/operators/tst.ternary.d mode=0444
file path=opt/SUNWdtrt/tst/common/operators/tst.ternary.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/pid/err.D_PDESC_ZERO.badlib.d mode=0444
file path=opt/SUNWdtrt/tst/common/pid/err.D_PDESC_ZERO.badlib.exe mode=0555
file path=opt/SUNWdtrt/tst/common/pid/err.D_PDESC_ZERO.badproc1.d mode=0444
file path=opt/SUNWdtrt/tst/common/pid/err.D_PROC_BADPID.badproc2.d mode=0444
file path=opt/SUNWdtrt/tst/common/pid/err.D_PROC_CREATEFAIL.many.d mode=0444
file path=opt/SUNWdtrt/tst/common/pid/err.D_PROC_CREATEFAIL.many.exe mode=0555
file path=opt/SUNWdtrt/tst/common/pid/err.D_PROC_FUNC.badfunc.d mode=0444
file path=opt/SUNWdtrt/tst/common/pid/err.D_PROC_FUNC.badfunc.exe mode=0555
file path=opt/SUNWdtrt/tst/common/pid/err.D_PROC_LIB.libdash.d mode=0444
file path=opt/SUNWdtrt/tst/common/pid/err.D_PROC_LIB.libdash.exe mode=0555
file path=opt/SUNWdtrt/tst/common/pid/err.D_PROC_NAME.alldash.d mode=0444
file path=opt/SUNWdtrt/tst/common/pid/err.D_PROC_NAME.alldash.exe mode=0555
file path=opt/SUNWdtrt/tst/common/pid/err.D_PROC_NAME.badname.d mode=0444
file path=opt/SUNWdtrt/tst/common/pid/err.D_PROC_NAME.badname.exe mode=0555
file path=opt/SUNWdtrt/tst/common/pid/err.D_PROC_NAME.globdash.d mode=0444
file path=opt/SUNWdtrt/tst/common/pid/err.D_PROC_NAME.globdash.exe mode=0555
file path=opt/SUNWdtrt/tst/common/pid/err.D_PROC_OFF.toobig.d mode=0444
file path=opt/SUNWdtrt/tst/common/pid/err.D_PROC_OFF.toobig.exe mode=0555
file path=opt/SUNWdtrt/tst/common/pid/tst.addprobes.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/pid/tst.args1.d mode=0444
file path=opt/SUNWdtrt/tst/common/pid/tst.args1.exe mode=0555
```



```
file path=opt/SUNWdtrt/tst/common/pid/tst.coverage.d mode=0444
file path=opt/SUNWdtrt/tst/common/pid/tst.coverage.exe mode=0555
file path=opt/SUNWdtrt/tst/common/pid/tst.emptystack.d mode=0444
file path=opt/SUNWdtrt/tst/common/pid/tst.emptystack.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/pid/tst.emptystack.exe mode=0555
file path=opt/SUNWdtrt/tst/common/pid/tst.float.d mode=0444
file path=opt/SUNWdtrt/tst/common/pid/tst.float.exe mode=0555
file path=opt/SUNWdtrt/tst/common/pid/tst.fork.d mode=0444
file path=opt/SUNWdtrt/tst/common/pid/tst.fork.exe mode=0555
file path=opt/SUNWdtrt/tst/common/pid/tst.gcc.d mode=0444
file path=opt/SUNWdtrt/tst/common/pid/tst.gcc.exe mode=0555
file path=opt/SUNWdtrt/tst/common/pid/tst.killonerror.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/pid/tst.main.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/pid/tst.manypids.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/pid/tst.newprobes.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/pid/tst.newprobes.ksh.out mode=0444
file path=opt/SUNWdtrt/tst/common/pid/tst.probemod.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/pid/tst.provregex1.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/pid/tst.provregex2.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/pid/tst.provregex2.ksh.out mode=0444
file path=opt/SUNWdtrt/tst/common/pid/tst.provregex3.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/pid/tst.provregex3.ksh.out mode=0444
file path=opt/SUNWdtrt/tst/common/pid/tst.provregex4.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/pid/tst.provregex4.ksh.out mode=0444
file path=opt/SUNWdtrt/tst/common/pid/tst.ret1.d mode=0444
file path=opt/SUNWdtrt/tst/common/pid/tst.ret1.exe mode=0555
file path=opt/SUNWdtrt/tst/common/pid/tst.ret2.d mode=0444
file path=opt/SUNWdtrt/tst/common/pid/tst.ret2.exe mode=0555
file path=opt/SUNWdtrt/tst/common/pid/tst.utf8probefunc.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/pid/tst.utf8probefunc.ksh.out mode=0444
file path=opt/SUNWdtrt/tst/common/pid/tst.utf8probemod.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/pid/tst.utf8probemod.ksh.out mode=0444
file path=opt/SUNWdtrt/tst/common/pid/tst.vfork.d mode=0444
file path=opt/SUNWdtrt/tst/common/pid/tst.vfork.exe mode=0555
file path=opt/SUNWdtrt/tst/common/pid/tst.weak1.d mode=0444
file path=opt/SUNWdtrt/tst/common/pid/tst.weak1.exe mode=0555
file path=opt/SUNWdtrt/tst/common/pid/tst.weak2.d mode=0444
file path=opt/SUNWdtrt/tst/common/pid/tst.weak2.exe mode=0555
file path=opt/SUNWdtrt/tst/common/plockstat/tst.available.d mode=0444
file path=opt/SUNWdtrt/tst/common/plockstat/tst.available.exe mode=0555
file path=opt/SUNWdtrt/tst/common/plockstat/tst.libmap.d mode=0444
file path=opt/SUNWdtrt/tst/common/plockstat/tst.libmap.exe mode=0555
file path=opt/SUNWdtrt/tst/common/pointers/err.BadAlign.d mode=0444
file path=opt/SUNWdtrt/tst/common/pointers/err.D_ADDROF_VAR.ArrayVar.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/pointers/err.D_ADDROF_VAR.DynamicVar.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/pointers/err.D_ADDROF_VAR.agg.d mode=0444
file path=opt/SUNWdtrt/tst/common/pointers/err.D_DEREF_NONPTR.noptr.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/pointers/err.D_DEREF_VOID.VoidPointerDeref.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/pointers/err.D_OP_ARRFUN.ArrayAssignment.d \
    mode=0444
file \
    path=opt/SUNWdtrt/tst/common/pointers/err.D_OP_INCOMPAT.VoidPointerArith.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/pointers/err.D_OP_LVAL.AddressChange.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/pointers/err.D_OP_PTR.NonPointerAccess.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/pointers/err.D_OP_PTR.badpointer.d mode=0444
file path=opt/SUNWdtrt/tst/common/pointers/err.D_OP_SOU.BadPointerAccess.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/pointers/err.D_OP_SOU.badpointer.d mode=0444
file path=opt/SUNWdtrt/tst/common/pointers/err.InvalidAddress1.d mode=0444
```

```
file path=opt/SUNWdtrt/tst/common/pointers/err.InvalidAddress2.d mode=0444
file path=opt/SUNWdtrt/tst/common/pointers/err.InvalidAddress3.d mode=0444
file path=opt/SUNWdtrt/tst/common/pointers/err.InvalidAddress4.d mode=0444
file path=opt/SUNWdtrt/tst/common/pointers/err.InvalidAddress5.d mode=0444
file path=opt/SUNWdtrt/tst/common/pointers/tst.ArrayPointer1.d mode=0444
file path=opt/SUNWdtrt/tst/common/pointers/tst.ArrayPointer2.d mode=0444
file path=opt/SUNWdtrt/tst/common/pointers/tst.ArrayPointer3.d mode=0444
file path=opt/SUNWdtrt/tst/common/pointers/tst.GlobalVar.d mode=0444
file path=opt/SUNWdtrt/tst/common/pointers/tst.IntegerArithmetic1.d mode=0444
file path=opt/SUNWdtrt/tst/common/pointers/tst.PointerArithmetic1.d mode=0444
file path=opt/SUNWdtrt/tst/common/pointers/tst.PointerArithmetic2.d mode=0444
file path=opt/SUNWdtrt/tst/common/pointers/tst.PointerArithmetic3.d mode=0444
file path=opt/SUNWdtrt/tst/common/pointers/tst.PointerAssignment.d mode=0444
file path=opt/SUNWdtrt/tst/common/pointers/tst.ValidPointer1.d mode=0444
file path=opt/SUNWdtrt/tst/common/pointers/tst.ValidPointer2.d mode=0444
file path=opt/SUNWdtrt/tst/common/pointers/tst.VoidCast.d mode=0444
file path=opt/SUNWdtrt/tst/common/pointers/tst.assigncast1.d mode=0444
file path=opt/SUNWdtrt/tst/common/pointers/tst.assigncast2.d mode=0444
file path=opt/SUNWdtrt/tst/common/pointers/tst.basic1.d mode=0444
file path=opt/SUNWdtrt/tst/common/pointers/tst.basic2.d mode=0444
file path=opt/SUNWdtrt/tst/common/pragma/err.D_PRAGERR.d mode=0444
file path=opt/SUNWdtrt/tst/common/pragma/err.D_PRAGMA_DEPEND.main.d mode=0444
file path=opt/SUNWdtrt/tst/common/pragma/err.D_PRAGMA_INVAL.d mode=0444
file path=opt/SUNWdtrt/tst/common/pragma/err.D_PRAGMA_MALFORM.d mode=0444
file path=opt/SUNWdtrt/tst/common/pragma/err.D_PRAGMA_UNUSED.UnusedPragma.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/pragma/err.circlibdep.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/pragma/err.invalidlibdep.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/pragma/tst.libchain.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/pragma/tst.libdep.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/pragma/tst.libdepfullyconnected.ksh \
    mode=0444
file path=opt/SUNWdtrt/tst/common/pragma/tst.libdepsepdir.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/pragma/tst.temporal.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/pragma/tst.temporal2.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/pragma/tst.temporal3.d mode=0444
file path=opt/SUNWdtrt/tst/common/predicates/err.D_PRED_SCALAR.NonScalarPred.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/predicates/err.D_SYNTAX.invalid.d mode=0444
file path=opt/SUNWdtrt/tst/common/predicates/err.D_SYNTAX.operr.d mode=0444
file path=opt/SUNWdtrt/tst/common/predicates/tst.argsnotcached.d mode=0444
file path=opt/SUNWdtrt/tst/common/predicates/tst.basics.d mode=0444
file path=opt/SUNWdtrt/tst/common/predicates/tst.basics.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/predicates/tst.complex.d mode=0444
file path=opt/SUNWdtrt/tst/common/predicates/tst.complex.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/preprocessor/err.D_IDENT_UNDEF.afterprobe.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/preprocessor/err.D_PRAGCTL_INVAL.tabdefine.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/preprocessor/err.D_SYNTAX.withoutpound.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/preprocessor/err.defincomp.d mode=0444
file path=opt/SUNWdtrt/tst/common/preprocessor/err.ifdefelsenotendif.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/preprocessor/err.ifdefincomp.d mode=0444
file path=opt/SUNWdtrt/tst/common/preprocessor/err.ifdefnotendif.d mode=0444
file path=opt/SUNWdtrt/tst/common/preprocessor/err.incompelse.d mode=0444
file path=opt/SUNWdtrt/tst/common/preprocessor/err.mulelse.d mode=0444
file path=opt/SUNWdtrt/tst/common/preprocessor/tst.ifdef.d mode=0444
file path=opt/SUNWdtrt/tst/common/preprocessor/tst.ifdef.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/preprocessor/tst.ifndef.d mode=0444
file path=opt/SUNWdtrt/tst/common/preprocessor/tst.ifndef.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/preprocessor/tst.ifnotdef.d mode=0444
file path=opt/SUNWdtrt/tst/common/preprocessor/tst.ifnotdef.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/preprocessor/tst.logicaland.d mode=0444
file path=opt/SUNWdtrt/tst/common/preprocessor/tst.logicaland.d.out mode=0444
```



```
file path=opt/SUNWdtrt/tst/common/preprocessor/tst.logicalandor.d mode=0444
file path=opt/SUNWdtrt/tst/common/preprocessor/tst.logicalandor.d.out \
    mode=0444
file path=opt/SUNWdtrt/tst/common/preprocessor/tst.logicalor.d mode=0444
file path=opt/SUNWdtrt/tst/common/preprocessor/tst.logicalor.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/preprocessor/tst.muland.d mode=0444
file path=opt/SUNWdtrt/tst/common/preprocessor/tst.muland.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/preprocessor/tst.mulor.d mode=0444
file path=opt/SUNWdtrt/tst/common/preprocessor/tst.mulor.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/preprocessor/tst.precondi.d mode=0444
file path=opt/SUNWdtrt/tst/common/preprocessor/tst.precondi.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/preprocessor/tst.predicatedeclare.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/preprocessor/tst.preexp.d mode=0444
file path=opt/SUNWdtrt/tst/common/preprocessor/tst.preexp.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/preprocessor/tst.preexpelse.d mode=0444
file path=opt/SUNWdtrt/tst/common/preprocessor/tst.preexpelse.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/preprocessor/tst.preexpif.d mode=0444
file path=opt/SUNWdtrt/tst/common/preprocessor/tst.preexpif.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/preprocessor/tst.preexpifelse.d mode=0444
file path=opt/SUNWdtrt/tst/common/preprocessor/tst.preexpifelse.d.out \
    mode=0444
file path=opt/SUNWdtrt/tst/common/preprocessor/tst.withinprobe.d mode=0444
file path=opt/SUNWdtrt/tst/common/print/err.D_PRINT_AGG.bad.d mode=0444
file path=opt/SUNWdtrt/tst/common/print/err.D_PRINT_VOID.bad.d mode=0444
file path=opt/SUNWdtrt/tst/common/print/err.D_PROTO_LEN.bad.d mode=0444
file path=opt/SUNWdtrt/tst/common/print/tst.array.d mode=0444
file path=opt/SUNWdtrt/tst/common/print/tst.array.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/print/tst.bitfield.d mode=0444
file path=opt/SUNWdtrt/tst/common/print/tst.bitfield.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/print/tst.dyn.d mode=0444
file path=opt/SUNWdtrt/tst/common/print/tst.enum.d mode=0444
file path=opt/SUNWdtrt/tst/common/print/tst.enum.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/print/tst.primitive.d mode=0444
file path=opt/SUNWdtrt/tst/common/print/tst.primitive.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/print/tst.struct.d mode=0444
file path=opt/SUNWdtrt/tst/common/print/tst.struct.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/print/tst.xlate.d mode=0444
file path=opt/SUNWdtrt/tst/common/print/tst.xlate.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/printa/err.D_PRINTA_AGGARG.badagg.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/printa/err.D_PRINTA_AGGARG.badfmt.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/printa/err.D_PRINTA_AGGARG.badval.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/printa/err.D_PRINTA_PROTO.bad.d mode=0444
file path=opt/SUNWdtrt/tst/common/printa/err.D_PRINTF_ARG_TYPE.jstack.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/printa/err.D_PRINTF_ARG_TYPE.stack.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/printa/err.D_PRINTF_ARG_TYPE.ustack.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/printa/tst.basics.d mode=0444
file path=opt/SUNWdtrt/tst/common/printa/tst.basics.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/printa/tst.def.d mode=0444
file path=opt/SUNWdtrt/tst/common/printa/tst.def.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/printa/tst.dynwidth.d mode=0444
file path=opt/SUNWdtrt/tst/common/printa/tst.dynwidth.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/printa/tst.fmt.d mode=0444
file path=opt/SUNWdtrt/tst/common/printa/tst.fmt.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/printa/tst.largeusersym.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/printa/tst.many.d mode=0444
file path=opt/SUNWdtrt/tst/common/printa/tst.manyval.d mode=0444
file path=opt/SUNWdtrt/tst/common/printa/tst.manyval.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/printa/tst.stack.d mode=0444
file path=opt/SUNWdtrt/tst/common/printa/tst.tuple.d mode=0444
```

```
file path=opt/SUNWdtrt/tst/common/printa/tst.tuple.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/printa/tst.walltimestamp.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/printa/tst.walltimestamp.ksh.out mode=0444
file path=opt/SUNWdtrt/tst/common/printf/err.D_PRINTF_AGG_CONV.aggfmt.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/printf/err.D_PRINTF_ARG_EXTRA.toomany.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/printf/err.D_PRINTF_ARG_EXTRA.widths.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/printf/err.D_PRINTF_ARG_FMT.badfmt.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/printf/err.D_PRINTF_ARG_PROTO.novalue.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/printf/err.D_PRINTF_ARG_TYPE.aggarg.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/printf/err.D_PRINTF_ARG_TYPE.recursive.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/printf/err.D_PRINTF_DYN_PROTO.noprec.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/printf/err.D_PRINTF_DYN_PROTO.nowidth.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/printf/err.D_PRINTF_DYN_TYPE.badprec.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/printf/err.D_PRINTF_DYN_TYPE.badwidth.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/printf/err.D_PROTO_LEN.toofew.d mode=0444
file path=opt/SUNWdtrt/tst/common/printf/err.D_SYNTAX.badconv1.d mode=0444
file path=opt/SUNWdtrt/tst/common/printf/err.D_SYNTAX.badconv2.d mode=0444
file path=opt/SUNWdtrt/tst/common/printf/err.D_SYNTAX.badconv3.d mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.basics.d mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.basics.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.flags.d mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.flags.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.hello.d mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.hello.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.ints.d mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.ints.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.precs.d mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.precs.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.print-f.d mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.print-f.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.printT.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.printT.ksh.out mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.printY.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.printY.ksh.out mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.printcont.d mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.printcont.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.printeE.d mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.printeE.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.printgG.d mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.printgG.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.rawfmt.d mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.rawfmt.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.signs.d mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.signs.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.str.d mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.str.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.sym.d mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.sym.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.uints.d mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.uints.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.widths.d mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.widths.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.widths1.d mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.wp.d mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.wp.d.out mode=0444
```



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

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



```
file path=opt/SUNWdtrt/tst/common/safety/tst.copyin2.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.ddi_pathname.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.dirname.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.errno.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.execname.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.gid.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.hton.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.index.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.msgdsize.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.msgsize.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.null.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.pid.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.ppid.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.progenyof.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.random.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.rw.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.shortstr.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.stack.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.stackdepth.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.stddev.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.strchr.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.strjoin.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.strstr.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.strtok.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.substr.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.ucaller.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.uid.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.unalign.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.uregs.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.ustack.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.ustackdepth.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.vahole.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.violentdeath.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.zonename.d mode=0444
file path=opt/SUNWdtrt/tst/common/scalars/err.D_ARR_LOCAL.thisarray.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/scalars/err.D_DECL_CLASS.selfthis.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/scalars/err.D_DECL_CLASS.thisself.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/scalars/err.D_DECL_IDRED.errval.d mode=0444
file path=opt/SUNWdtrt/tst/common/scalars/err.D_OP_INCOMPAT.dec.err.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/scalars/err.D_OP_INCOMPAT.dupgtype.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/scalars/err.D_OP_INCOMPAT.dupltype.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/scalars/err.D_OP_INCOMPAT.dupttype.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/scalars/err.D_SYNTAX.declare.d mode=0444
file path=opt/SUNWdtrt/tst/common/scalars/tst.basicvar.d mode=0444
file path=opt/SUNWdtrt/tst/common/scalars/tst.basicvar.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/scalars/tst.localvar.d mode=0444
file path=opt/SUNWdtrt/tst/common/scalars/tst.misc.d mode=0444
file path=opt/SUNWdtrt/tst/common/scalars/tst.self.d mode=0444
file path=opt/SUNWdtrt/tst/common/scalars/tst.selfarray.d mode=0444
file path=opt/SUNWdtrt/tst/common/scalars/tst.selfarray2.d mode=0444
file path=opt/SUNWdtrt/tst/common/scalars/tst.selfthis.d mode=0444
file path=opt/SUNWdtrt/tst/common/scalars/tst.this.d mode=0444
file path=opt/SUNWdtrt/tst/common/scalars/tst.thisself.d mode=0444
file path=opt/SUNWdtrt/tst/common/sched/tst.enqueue.d mode=0444
file path=opt/SUNWdtrt/tst/common/sched/tst.oncpu.d mode=0444
file path=opt/SUNWdtrt/tst/common/sched/tst.stackdepth.d mode=0444
file path=opt/SUNWdtrt/tst/common/scripting/err.D_MACRO_UNDEF.invalidargs.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/scripting/err.D_OP_LVAL.rdonly.d mode=0444
```

```
1579 file path=opt/SUNWdtrt/tst/common/scripting/err.D_OP_WRITE.usepidmacro.d \
1580     mode=0444
1581 file path=opt/SUNWdtrt/tst/common/scripting/err.D_SYNTAX.concat.d mode=0444
1582 file path=opt/SUNWdtrt/tst/common/scripting/err.D_SYNTAX.desc.d mode=0444
1583 file path=opt/SUNWdtrt/tst/common/scripting/err.D_SYNTAX.inval.d mode=0444
1584 file path=opt/SUNWdtrt/tst/common/scripting/err.D_SYNTAX.pid.d mode=0444
1585 file path=opt/SUNWdtrt/tst/common/scripting/tst.D_MACRO_UNUSED.overflow.ksh \
1586     mode=0444
1587 file path=opt/SUNWdtrt/tst/common/scripting/tst.arg0.d mode=0444
1588 file path=opt/SUNWdtrt/tst/common/scripting/tst.arguments.ksh mode=0444
1589 file path=opt/SUNWdtrt/tst/common/scripting/tst.assign.d mode=0444
1590 file path=opt/SUNWdtrt/tst/common/scripting/tst.basic.d mode=0444
1591 file path=opt/SUNWdtrt/tst/common/scripting/tst.egid.d mode=0444
1592 file path=opt/SUNWdtrt/tst/common/scripting/tst.egid.ksh mode=0444
1593 file path=opt/SUNWdtrt/tst/common/scripting/tst.euid.d mode=0444
1594 file path=opt/SUNWdtrt/tst/common/scripting/tst.euid.ksh mode=0444
1595 file path=opt/SUNWdtrt/tst/common/scripting/tst.gid.d mode=0444
1596 file path=opt/SUNWdtrt/tst/common/scripting/tst.gid.ksh mode=0444
1597 file path=opt/SUNWdtrt/tst/common/scripting/tst.pgid.d mode=0444
1598 file path=opt/SUNWdtrt/tst/common/scripting/tst.pid.d mode=0444
1599 file path=opt/SUNWdtrt/tst/common/scripting/tst.ppid.d mode=0444
1600 file path=opt/SUNWdtrt/tst/common/scripting/tst.ppid.ksh mode=0444
1601 file path=opt/SUNWdtrt/tst/common/scripting/tst.projid.d mode=0444
1602 file path=opt/SUNWdtrt/tst/common/scripting/tst.projid.ksh mode=0444
1603 file path=opt/SUNWdtrt/tst/common/scripting/tst.quite.d mode=0444
1604 file path=opt/SUNWdtrt/tst/common/scripting/tst.sid.d mode=0444
1605 file path=opt/SUNWdtrt/tst/common/scripting/tst.sid.ksh mode=0444
1606 file path=opt/SUNWdtrt/tst/common/scripting/tst.stringmacro.ksh mode=0444
1607 file path=opt/SUNWdtrt/tst/common/scripting/tst.taskid.d mode=0444
1608 file path=opt/SUNWdtrt/tst/common/scripting/tst.taskid.ksh mode=0444
1609 file path=opt/SUNWdtrt/tst/common/scripting/tst.trace.d mode=0444
1610 file path=opt/SUNWdtrt/tst/common/scripting/tst.uid.d mode=0444
1611 file path=opt/SUNWdtrt/tst/common/scripting/tst.uid.ksh mode=0444
1612 file path=opt/SUNWdtrt/tst/common/sdt/tst.sdtargs.d mode=0444
1613 file path=opt/SUNWdtrt/tst/common/sdt/tst.sdtargs.exe mode=0555
1614 file path=opt/SUNWdtrt/tst/common/sizeof/err.D_IDENT_BADREF.SizeofAssoc.d \
1615     mode=0444
1616 file path=opt/SUNWdtrt/tst/common/sizeof/err.D_IDENT_UNDEF.UnknownSymbol.d \
1617     mode=0444
1618 file path=opt/SUNWdtrt/tst/common/sizeof/err.D_SIZEOF_TYPE.badstruct.d \
1619     mode=0444
1620 file path=opt/SUNWdtrt/tst/common/sizeof/err.D_SIZEOF_TYPE.d mode=0444
1621 file path=opt/SUNWdtrt/tst/common/sizeof/err.D_SYNTAX.SizeofBadType.d \
1622     mode=0444
1623 file path=opt/SUNWdtrt/tst/common/sizeof/tst.SizeofArray.d mode=0444
1624 file path=opt/SUNWdtrt/tst/common/sizeof/tst.SizeofDataTypes.d mode=0444
1625 file path=opt/SUNWdtrt/tst/common/sizeof/tst.SizeofExpression.d mode=0444
1626 file path=opt/SUNWdtrt/tst/common/sizeof/tst.SizeofNULL.d mode=0444
1627 file path=opt/SUNWdtrt/tst/common/sizeof/tst.SizeofStrConst.d mode=0444
1628 file path=opt/SUNWdtrt/tst/common/sizeof/tst.SizeofStrConst.d.out mode=0444
1629 file path=opt/SUNWdtrt/tst/common/sizeof/tst.SizeofString1.d mode=0444
1630 file path=opt/SUNWdtrt/tst/common/sizeof/tst.SizeofString1.d.out mode=0444
1631 file path=opt/SUNWdtrt/tst/common/sizeof/tst.SizeofString2.d mode=0444
1632 file path=opt/SUNWdtrt/tst/common/sizeof/tst.SizeofString2.d.out mode=0444
1633 file path=opt/SUNWdtrt/tst/common/speculation/err.BufSizeVariations1.d \
1634     mode=0444
1635 file path=opt/SUNWdtrt/tst/common/speculation/err.BufSizeVariations2.d \
1636     mode=0444
1637 file \
1638     path=opt/SUNWdtrt/tst/common/speculation/err.D_ACT_SPEC.SpeculateWithBreakPo
1639     mode=0444
1640 file \
1641     path=opt/SUNWdtrt/tst/common/speculation/err.D_ACT_SPEC.SpeculateWithChill.d
1642     mode=0444
1643 file \
1644     path=opt/SUNWdtrt/tst/common/speculation/err.D_ACT_SPEC.SpeculateWithCopyOut
```



```
1645     mode=0444
1646 file \
1647     path=opt/SUNWdtrt/tst/common/speculation/err.D_ACT_SPEC.SpeculateWithCopyOut
1648     mode=0444
1649 file \
1650     path=opt/SUNWdtrt/tst/common/speculation/err.D_ACT_SPEC.SpeculateWithPanic.d
1651     mode=0444
1652 file \
1653     path=opt/SUNWdtrt/tst/common/speculation/err.D_ACT_SPEC.SpeculateWithRaise.d
1654     mode=0444
1655 file \
1656     path=opt/SUNWdtrt/tst/common/speculation/err.D_ACT_SPEC.SpeculateWithStop.d
1657     mode=0444
1658 file path=opt/SUNWdtrt/tst/common/speculation/err.D_AGG_COMM.AggAftCommit.d \
1659     mode=0444
1660 file \
1661     path=opt/SUNWdtrt/tst/common/speculation/err.D_AGG_SPEC.SpeculateWithAvg.d \
1662     mode=0444
1663 file \
1664     path=opt/SUNWdtrt/tst/common/speculation/err.D_AGG_SPEC.SpeculateWithCount.d
1665     mode=0444
1666 file \
1667     path=opt/SUNWdtrt/tst/common/speculation/err.D_AGG_SPEC.SpeculateWithLquant.
1668     mode=0444
1669 file \
1670     path=opt/SUNWdtrt/tst/common/speculation/err.D_AGG_SPEC.SpeculateWithMax.d \
1671     mode=0444
1672 file \
1673     path=opt/SUNWdtrt/tst/common/speculation/err.D_AGG_SPEC.SpeculateWithMin.d \
1674     mode=0444
1675 file \
1676     path=opt/SUNWdtrt/tst/common/speculation/err.D_AGG_SPEC.SpeculateWithQuant.d
1677     mode=0444
1678 file \
1679     path=opt/SUNWdtrt/tst/common/speculation/err.D_AGG_SPEC.SpeculateWithStddev.
1680     mode=0444
1681 file \
1682     path=opt/SUNWdtrt/tst/common/speculation/err.D_AGG_SPEC.SpeculateWithSum.d \
1683     mode=0444
1684 file \
1685     path=opt/SUNWdtrt/tst/common/speculation/err.D_COMM_COMM.CommitAftCommit.d \
1686     mode=0444
1687 file path=opt/SUNWdtrt/tst/common/speculation/err.D_COMM_COMM.DisjointCommit.d \
1688     mode=0444
1689 file \
1690     path=opt/SUNWdtrt/tst/common/speculation/err.D_COMM_DREC.CommitAftDataRec.d
1691     mode=0444
1692 file \
1693     path=opt/SUNWdtrt/tst/common/speculation/err.D_DREC_COMM.DataRecAftCommit.d
1694     mode=0444
1695 file \
1696     path=opt/SUNWdtrt/tst/common/speculation/err.D_DREC_COMM.ExitAfterCommit.d \
1697     mode=0444
1698 file path=opt/SUNWdtrt/tst/common/speculation/err.D_EXIT_SPEC.ExitAftSpec.d \
1699     mode=0444
1700 file path=opt/SUNWdtrt/tst/common/speculation/err.D_PRAGMA_MALFORM.NspecExpr.d \
1701     mode=0444
1702 file \
1703     path=opt/SUNWdtrt/tst/common/speculation/err.D_PRAGMA_OPTSET.HugeNspecValue.
1704     mode=0444
1705 file \
1706     path=opt/SUNWdtrt/tst/common/speculation/err.D_PRAGMA_OPTSET.InvalidSpecSize
1707     mode=0444
1708 file \
1709     path=opt/SUNWdtrt/tst/common/speculation/err.D_PRAGMA_OPTSET.NegSpecSize.d \
1710     mode=0444
```

```
file path=opt/SUNWdtrt/tst/common/speculation/err.D_PROTO_LEN.SpecNoId.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/err.D_SPEC_COMM.SpecAftCommit.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/err.D_SPEC_DREC.SpecAftDataRec.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/err.D_SPEC_SPEC.SpecAftSpec.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/err.NegativeBufSize.d mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/err.NegativeNspec.d mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/err.NegativeSpecSize.d mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/err.SpecSizeVariations1.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/err.SpecSizeVariations2.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/tst.CommitAfterDiscard.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/tst.CommitWithZero.d mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/tst.DataRecAftDiscard.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/tst.DiscardAftCommit.d mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/tst.DiscardAftDataRec.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/tst.DiscardAftDiscard.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/tst.DiscardWithZero.d mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/tst.ExitAftDiscard.d mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/tst.NoSpecBuffer.d mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/tst.SpecSizeVariations1.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/tst.SpecSizeVariations2.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/tst.SpecSizeVariations3.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/tst.SpeculateWithRandom.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/tst.SpeculationCommit.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/tst.SpeculationDiscard.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/tst.SpeculationID.d mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/tst.SpeculationWithZero.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/tst.TwoSpecBuffers.d mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/tst.negcommit.d mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/tst.negspec.d mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/tst.zerosize.d mode=0444
file path=opt/SUNWdtrt/tst/common/stability/err.D_ATTR_MIN.MinAttributes.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/stack/err.D_STACK_PROTO.bad.d mode=0444
file path=opt/SUNWdtrt/tst/common/stack/err.D_STACK_SIZE.d mode=0444
file path=opt/SUNWdtrt/tst/common/stack/err.D_USTACK_FRAMES.bad.d mode=0444
file path=opt/SUNWdtrt/tst/common/stack/err.D_USTACK_PROTO.bad.d mode=0444
file path=opt/SUNWdtrt/tst/common/stack/err.D_USTACK_STRSIZE.bad.d mode=0444
file path=opt/SUNWdtrt/tst/common/stack/tst.default.d mode=0444
file path=opt/SUNWdtrt/tst/common/stackdepth/tst.default.d mode=0444
file path=opt/SUNWdtrt/tst/common/stop/tst.stop1.d mode=0444
file path=opt/SUNWdtrt/tst/common/stop/tst.stop1.exe mode=0555
file path=opt/SUNWdtrt/tst/common/stop/tst.stop2.d mode=0444
file path=opt/SUNWdtrt/tst/common/stop/tst.stop2.exe mode=0555
file path=opt/SUNWdtrt/tst/common/strlen/tst.strlen1.d mode=0444
file path=opt/SUNWdtrt/tst/common/strtoll/err.BaseTooLarge.d mode=0444
file path=opt/SUNWdtrt/tst/common/strtoll/err.BaseTooSmall.d mode=0444
file path=opt/SUNWdtrt/tst/common/strtoll/tst.strtoll.d mode=0444
file path=opt/SUNWdtrt/tst/common/strtoll/tst.strtoll.d.out mode=0444
#endif /* ! codereview */
```



```
file path=opt/SUNWdtrt/tst/common/struct/err.D_ADDROF_VAR.StructPointer.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/struct/err.D_DECL_COMBO.StructWithoutColon.d \
    mode=0444
file \
    path=opt/SUNWdtrt/tst/common/struct/err.D_DECL_COMBO.StructWithoutColon1.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/struct/err.D_DECL_INCOMPLETE.circular.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/struct/err.D_DECL_INCOMPLETE.order.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/struct/err.D_DECL_INCOMPLETE.order2.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/struct/err.D_DECL_INCOMPLETE.recursive.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/struct/err.D_DECL_INCOMPLETE.simple.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/struct/err.D_DECL_VOIDOBJ.baddec.d mode=0444
file path=opt/SUNWdtrt/tst/common/struct/err.D_PROTO_ARG.DupStructAssoc.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/struct/tst.StructAssoc.d mode=0444
file path=opt/SUNWdtrt/tst/common/struct/tst.StructDataTypes.d mode=0444
file path=opt/SUNWdtrt/tst/common/struct/tst.StructInside.d mode=0444
file path=opt/SUNWdtrt/tst/common/struct/tst.clauselocal.d mode=0444
file path=opt/SUNWdtrt/tst/common/struct/tst.clauselocal.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/syscall/tst.args.d mode=0444
file path=opt/SUNWdtrt/tst/common/syscall/tst.args.exe mode=0555
file path=opt/SUNWdtrt/tst/common/syscall/tst.openret.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/sysevent/tst.post.d mode=0444
file path=opt/SUNWdtrt/tst/common/sysevent/tst.post.exe mode=0555
file path=opt/SUNWdtrt/tst/common/sysevent/tst.post_chan.d mode=0444
file path=opt/SUNWdtrt/tst/common/sysevent/tst.post_chan.exe mode=0555
file path=opt/SUNWdtrt/tst/common/tick-n/err.D_PDESC_ZERO.tick.d mode=0444
file path=opt/SUNWdtrt/tst/common/tick-n/err.D_PDESC_ZEROonens.d mode=0444
file path=opt/SUNWdtrt/tst/common/tick-n/err.D_PDESC_ZEROonensec.d mode=0444
file path=opt/SUNWdtrt/tst/common/tick-n/err.D_PDESC_ZEROoneus.d mode=0444
file path=opt/SUNWdtrt/tst/common/tick-n/err.D_PDESC_ZEROoneusec.d mode=0444
file path=opt/SUNWdtrt/tst/common/tick-n/tst.tickarg0.d mode=0444
file path=opt/SUNWdtrt/tst/common/tick-n/tst.tickms.d mode=0444
file path=opt/SUNWdtrt/tst/common/tick-n/tst.tickms.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/tick-n/tst.tickmsec.d mode=0444
file path=opt/SUNWdtrt/tst/common/tick-n/tst.tickmsec.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/tick-n/tst.tickns.d mode=0444
file path=opt/SUNWdtrt/tst/common/tick-n/tst.tickns.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/tick-n/tst.ticknsec.d mode=0444
file path=opt/SUNWdtrt/tst/common/tick-n/tst.ticknsec.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/tick-n/tst.ticks.d mode=0444
file path=opt/SUNWdtrt/tst/common/tick-n/tst.ticks.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/tick-n/tst.ticksec.d mode=0444
file path=opt/SUNWdtrt/tst/common/tick-n/tst.ticksec.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/tick-n/tst.tickus.d mode=0444
file path=opt/SUNWdtrt/tst/common/tick-n/tst.tickus.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/tick-n/tst.tickusec.d mode=0444
file path=opt/SUNWdtrt/tst/common/tick-n/tst.tickusec.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/trace/err.D_PROTO_LEN.bad.d mode=0444
file path=opt/SUNWdtrt/tst/common/trace/err.D_TRACE_AGG.bad.d mode=0444
file path=opt/SUNWdtrt/tst/common/trace/err.D_TRACE_VOID.bad.d mode=0444
file path=opt/SUNWdtrt/tst/common/trace/tst.dyn.d mode=0444
file path=opt/SUNWdtrt/tst/common/trace/tst.misc.d mode=0444
file path=opt/SUNWdtrt/tst/common/trace/tst.qstring.d mode=0444
file path=opt/SUNWdtrt/tst/common/trace/tst.qstring.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/trace/tst.string.d mode=0444
file path=opt/SUNWdtrt/tst/common/tracemem/err.D_PROTO_ARG.badsize.d mode=0444
file path=opt/SUNWdtrt/tst/common/tracemem/err.D_PROTO_LEN.toofew.d mode=0444
file path=opt/SUNWdtrt/tst/common/tracemem/err.D_TRACEMEM_ADDR.badaddr.d \
    mode=0444
```

```
1843 file path=opt/SUNWdtrt/tst/common/tracemem/err.D_TRACEMEM_ARGS.d mode=0444
1844 file path=opt/SUNWdtrt/tst/common/tracemem/err.D_TRACEMEM_DYNSIZE.d mode=0444
1845 file path=opt/SUNWdtrt/tst/common/tracemem/err.D_TRACEMEM_SIZE.negsize.d \
1846     mode=0444
1847 file path=opt/SUNWdtrt/tst/common/tracemem/err.D_TRACEMEM_SIZE.zerosize.d \
1848     mode=0444
1849 file path=opt/SUNWdtrt/tst/common/tracemem/tst.dynsize.d mode=0444
1850 file path=opt/SUNWdtrt/tst/common/tracemem/tst.dynsize.d.out mode=0444
1851 file path=opt/SUNWdtrt/tst/common/tracemem/tst.rootvp.d mode=0444
1852 file path=opt/SUNWdtrt/tst/common/tracemem/tst.smallsize.d mode=0444
1853 file path=opt/SUNWdtrt/tst/common/tracemem/tst.smallsize.d.out mode=0444
1854 file \
1855     path=opt/SUNWdtrt/tst/common/translators/err.D_DECL_TYPERED.BadTransDecl.d \
1856     mode=0444
1857 file \
1858     path=opt/SUNWdtrt/tst/common/translators/err.D_OP_INCOMPLETE.NonExistentInpu
1859     mode=0444
1860 file path=opt/SUNWdtrt/tst/common/translators/err.D_SYNTAX.BadTransDecl1.d \
1861     mode=0444
1862 file path=opt/SUNWdtrt/tst/common/translators/err.D_SYNTAX.BadTransDecl3.d \
1863     mode=0444
1864 file path=opt/SUNWdtrt/tst/common/translators/err.D_SYNTAX.BadTransDecl4.d \
1865     mode=0444
1866 file \
1867     path=opt/SUNWdtrt/tst/common/translators/err.D_TYPE_MEMBER.NonExistentInput2
1868     mode=0444
1869 file \
1870     path=opt/SUNWdtrt/tst/common/translators/err.D_XLATE_INCOMPAT.BadInputType1.
1871     mode=0444
1872 file \
1873     path=opt/SUNWdtrt/tst/common/translators/err.D_XLATE_MEMB.NonExistentOutput2
1874     mode=0444
1875 file path=opt/SUNWdtrt/tst/common/translators/err.D_XLATE_NONE.BadTransDecl6.d \
1876     mode=0444
1877 file \
1878     path=opt/SUNWdtrt/tst/common/translators/err.D_XLATE_REDECL.RepeatTransDecl.
1879     mode=0444
1880 file path=opt/SUNWdtrt/tst/common/translators/err.D_XLATE_SOU.BadTransDecl8.d \
1881     mode=0444
1882 file path=opt/SUNWdtrt/tst/common/translators/err.D_XLATE_SOU.BadTransInt.d \
1883     mode=0444
1884 file \
1885     path=opt/SUNWdtrt/tst/common/translators/err.D_XLATE_SOU.NonExistentOutput1.
1886     mode=0444
1887 file path=opt/SUNWdtrt/tst/common/translators/tst.CircularTransDecl.d \
1888     mode=0444
1889 file path=opt/SUNWdtrt/tst/common/translators/tst.EmptyTransDecl.d mode=0444
1890 file path=opt/SUNWdtrt/tst/common/translators/tst.ForwardTag.d mode=0444
1891 file path=opt/SUNWdtrt/tst/common/translators/tst.InputAliasTrans.d mode=0444
1892 file path=opt/SUNWdtrt/tst/common/translators/tst.InputIntTrans.d mode=0444
1893 file path=opt/SUNWdtrt/tst/common/translators/tst.OutputAliasTrans.d mode=0444
1894 file path=opt/SUNWdtrt/tst/common/translators/tst.PartialDereferencing.d \
1895     mode=0444
1896 file path=opt/SUNWdtrt/tst/common/translators/tst.PartialOutputTransDefn.d \
1897     mode=0444
1898 file path=opt/SUNWdtrt/tst/common/translators/tst.ProcModelTrans.d mode=0444
1899 file path=opt/SUNWdtrt/tst/common/translators/tst.RepeatDeclaration.d \
1900     mode=0444
1901 file path=opt/SUNWdtrt/tst/common/translators/tst.SimultaneousTranslators.d \
1902     mode=0444
1903 file path=opt/SUNWdtrt/tst/common/translators/tst.StructureAssignment.d \
1904     mode=0444
1905 file path=opt/SUNWdtrt/tst/common/translators/tst.TestTransStability1.ksh \
1906     mode=0444
1907 file path=opt/SUNWdtrt/tst/common/translators/tst.TestTransStability1.ksh.out \
1908     mode=0444
```



```
1909 file path=opt/SUNWdtrt/tst/common/translators/tst.TestTransStability2.ksh \
1910     mode=0444
1911 file path=opt/SUNWdtrt/tst/common/translators/tst.TestTransStability2.ksh.out \
1912     mode=0444
1913 file path=opt/SUNWdtrt/tst/common/translators/tst.TransNonPointer.d mode=0444
1914 file path=opt/SUNWdtrt/tst/common/translators/tst.TransOutputPointer.d \
1915     mode=0444
1916 file path=opt/SUNWdtrt/tst/common/translators/tst.TransPointer.d mode=0444
1917 file path=opt/SUNWdtrt/tst/common/translators/tst.TranslateSelf.d mode=0444
1918 file path=opt/SUNWdtrt/tst/common/translators/tst.UnionInputTrans.d mode=0444
1919 file path=opt/SUNWdtrt/tst/common/translators/tst.UnionOutputTrans.d mode=0444
1920 file path=opt/SUNWdtrt/tst/common/typedef/err.D_DECL_IDRED.DupTypeDef.d \
1921     mode=0444
1922 file path=opt/SUNWdtrt/tst/common/typedef/err.D_SYNTAX.BadExistingTypedef.d \
1923     mode=0444
1924 file path=opt/SUNWdtrt/tst/common/typedef/err.D_SYNTAX.TypedefInClause.d \
1925     mode=0444
1926 file path=opt/SUNWdtrt/tst/common/typedef/tst.ChainTypedef.d mode=0444
1927 file path=opt/SUNWdtrt/tst/common/typedef/tst.TypedefDataAssign.d mode=0444
1928 file path=opt/SUNWdtrt/tst/common/types/err.D_CAST_INVAL.badcast.d mode=0444
1929 file path=opt/SUNWdtrt/tst/common/types/err.D_CG_DYN.ResultDynType.d mode=0444
1930 file path=opt/SUNWdtrt/tst/common/types/err.D_CHR_OFLOW.charconst.d mode=0444
1931 file path=opt/SUNWdtrt/tst/common/types/err.D_DECL_BADCLASS.bad.d mode=0444
1932 file path=opt/SUNWdtrt/tst/common/types/err.D_DECL_CHARATTR.badtype3.d \
1933     mode=0444
1934 file path=opt/SUNWdtrt/tst/common/types/err.D_DECL_COMBO.badtype4.d mode=0444
1935 file path=opt/SUNWdtrt/tst/common/types/err.D_DECL_COMBO.badtype5.d mode=0444
1936 file path=opt/SUNWdtrt/tst/common/types/err.D_DECL_ENCONST.badeval.d mode=0444
1937 file path=opt/SUNWdtrt/tst/common/types/err.D_DECL_ENOFLOW.enoflow.d mode=0444
1938 file path=opt/SUNWdtrt/tst/common/types/err.D_DECL_ENOFLOW.enuflow.d mode=0444
1939 file path=opt/SUNWdtrt/tst/common/types/err.D_DECL_SCOPE.scopeop.d mode=0444
1940 file path=opt/SUNWdtrt/tst/common/types/err.D_DECL_USELESS.baddec.d mode=0444
1941 file path=opt/SUNWdtrt/tst/common/types/err.D_OP_ACT.badcond.d mode=0444
1942 file path=opt/SUNWdtrt/tst/common/types/err.D_OP_ARITH.badoperand.d mode=0444
1943 file path=opt/SUNWdtrt/tst/common/types/err.D_OP_INCOMPAT.badassign.d \
1944     mode=0444
1945 file path=opt/SUNWdtrt/tst/common/types/err.D_OP_INT.badbitop.d mode=0444
1946 file path=opt/SUNWdtrt/tst/common/types/err.D_OP_INT.badshift.d mode=0444
1947 file path=opt/SUNWdtrt/tst/common/types/err.D_OP_SCALAR.badcond.d mode=0444
1948 file path=opt/SUNWdtrt/tst/common/types/err.D_OP_SCALAR.badincop.d mode=0444
1949 file path=opt/SUNWdtrt/tst/common/types/err.D_OP_SCALAR.badlogop.d mode=0444
1950 file path=opt/SUNWdtrt/tst/common/types/err.D_PROTO_LEN.badcond1.d mode=0444
1951 file path=opt/SUNWdtrt/tst/common/types/err.D_SYNTAX.badenum.d mode=0444
1952 file path=opt/SUNWdtrt/tst/common/types/err.D_SYNTAX.badid.d mode=0444
1953 file path=opt/SUNWdtrt/tst/common/types/err.D_SYNTAX.badstruct.d mode=0444
1954 file path=opt/SUNWdtrt/tst/common/types/err.D_UNKNOWN.badtype1.d mode=0444
1955 file path=opt/SUNWdtrt/tst/common/types/err.D_UNKNOWN.badtype2.d mode=0444
1956 file path=opt/SUNWdtrt/tst/common/types/err.D_UNKNOWN.dupenum.d mode=0444
1957 file path=opt/SUNWdtrt/tst/common/types/err.D_UNKNOWN.dupstruct.d mode=0444
1958 file path=opt/SUNWdtrt/tst/common/types/err.D_XLATE_REDECL.ResultDynType.d \
1959     mode=0444
1960 file path=opt/SUNWdtrt/tst/common/types/tst.assignops.d mode=0444
1961 file path=opt/SUNWdtrt/tst/common/types/tst.badshiftops.d mode=0444
1962 file path=opt/SUNWdtrt/tst/common/types/tst.basics.d mode=0444
1963 file path=opt/SUNWdtrt/tst/common/types/tst.basics.d.out mode=0444
1964 file path=opt/SUNWdtrt/tst/common/types/tst.bitops.d mode=0444
1965 file path=opt/SUNWdtrt/tst/common/types/tst.charconstants.d mode=0444
1966 file path=opt/SUNWdtrt/tst/common/types/tst.complex.d mode=0444
1967 file path=opt/SUNWdtrt/tst/common/types/tst.condexpr.d mode=0444
1968 file path=opt/SUNWdtrt/tst/common/types/tst.const.d mode=0444
1969 file path=opt/SUNWdtrt/tst/common/types/tst.constants.d mode=0444
1970 file path=opt/SUNWdtrt/tst/common/types/tst.conv.d mode=0444
1971 file path=opt/SUNWdtrt/tst/common/types/tst.enum.d mode=0444
1972 file path=opt/SUNWdtrt/tst/common/types/tst.intincop.d mode=0444
1973 file path=opt/SUNWdtrt/tst/common/types/tst.intops.d mode=0444
1974 file path=opt/SUNWdtrt/tst/common/types/tst.inttypes.d mode=0444
```

```
1975 file path=opt/SUNWdtrt/tst/common/types/tst.ptrincop.d mode=0444
1976 file path=opt/SUNWdtrt/tst/common/types/tst.ptrops.d mode=0444
1977 file path=opt/SUNWdtrt/tst/common/types/tst.relenum.d mode=0444
1978 file path=opt/SUNWdtrt/tst/common/types/tst.relstring.d mode=0444
1979 file path=opt/SUNWdtrt/tst/common/types/tst.shiftops.d mode=0444
1980 file path=opt/SUNWdtrt/tst/common/types/tst.stringconstants.d mode=0444
1981 file path=opt/SUNWdtrt/tst/common/types/tst.struct.d mode=0444
1982 file path=opt/SUNWdtrt/tst/common/types/tst.typedef.d mode=0444
1983 file path=opt/SUNWdtrt/tst/common/types/tst.unaryop.d mode=0444
1984 file path=opt/SUNWdtrt/tst/common/union/err.D_ADDROF_VAR.UnionPointer.d \
1985     mode=0444
1986 file path=opt/SUNWdtrt/tst/common/union/err.D_DECL_COMBO.UnionWithoutColon.d \
1987     mode=0444
1988 file path=opt/SUNWdtrt/tst/common/union/err.D_DECL_COMBO.UnionWithoutColon1.d \
1989     mode=0444
1990 file path=opt/SUNWdtrt/tst/common/union/err.D_DECL_INCOMPLETE.circular.d \
1991     mode=0444
1992 file path=opt/SUNWdtrt/tst/common/union/err.D_DECL_INCOMPLETE.order.d \
1993     mode=0444
1994 file path=opt/SUNWdtrt/tst/common/union/err.D_DECL_INCOMPLETE.recursive.d \
1995     mode=0444
1996 file path=opt/SUNWdtrt/tst/common/union/err.D_DECL_INCOMPLETE.simple.d \
1997     mode=0444
1998 file path=opt/SUNWdtrt/tst/common/union/err.D_PROTO_ARG.DupUnionAssoc.d \
1999     mode=0444
2000 file path=opt/SUNWdtrt/tst/common/union/tst.UnionAssoc.d mode=0444
2001 file path=opt/SUNWdtrt/tst/common/union/tst.UnionDataTypes.d mode=0444
2002 file path=opt/SUNWdtrt/tst/common/union/tst.UnionInside.d mode=0444
2003 file path=opt/SUNWdtrt/tst/common/usdt/tst.andpid.ksh mode=0444
2004 file path=opt/SUNWdtrt/tst/common/usdt/tst.argmap.d mode=0444
2005 file path=opt/SUNWdtrt/tst/common/usdt/tst.argmap.exe mode=0555
2006 file path=opt/SUNWdtrt/tst/common/usdt/tst.args.d mode=0444
2007 file path=opt/SUNWdtrt/tst/common/usdt/tst.args.exe mode=0555
2008 file path=opt/SUNWdtrt/tst/common/usdt/tst.badguess.ksh mode=0444
2009 file path=opt/SUNWdtrt/tst/common/usdt/tst.corruptenv.ksh mode=0444
2010 file path=opt/SUNWdtrt/tst/common/usdt/tst.dlclose1.ksh mode=0444
2011 file path=opt/SUNWdtrt/tst/common/usdt/tst.dlclose1.ksh.out mode=0444
2012 file path=opt/SUNWdtrt/tst/common/usdt/tst.dlclose2.ksh mode=0444
2013 file path=opt/SUNWdtrt/tst/common/usdt/tst.dlclose2.ksh.out mode=0444
2014 file path=opt/SUNWdtrt/tst/common/usdt/tst.dlclose3.ksh mode=0444
2015 file path=opt/SUNWdtrt/tst/common/usdt/tst.eliminate.ksh mode=0444
2016 file path=opt/SUNWdtrt/tst/common/usdt/tst.enabled.ksh mode=0444
2017 file path=opt/SUNWdtrt/tst/common/usdt/tst.enabled.ksh.out mode=0444
2018 file path=opt/SUNWdtrt/tst/common/usdt/tst.enabled2.ksh mode=0444
2019 file path=opt/SUNWdtrt/tst/common/usdt/tst.enabled2.ksh.out mode=0444
2020 file path=opt/SUNWdtrt/tst/common/usdt/tst.entryreturn.ksh mode=0444
2021 file path=opt/SUNWdtrt/tst/common/usdt/tst.entryreturn.ksh.out mode=0444
2022 file path=opt/SUNWdtrt/tst/common/usdt/tst.fork.ksh mode=0444
2023 file path=opt/SUNWdtrt/tst/common/usdt/tst.fork.ksh.out mode=0444
2024 file path=opt/SUNWdtrt/tst/common/usdt/tst.forker.exe mode=0555
2025 file path=opt/SUNWdtrt/tst/common/usdt/tst.forker.ksh mode=0444
2026 file path=opt/SUNWdtrt/tst/common/usdt/tst.guess32.ksh mode=0444
2027 file path=opt/SUNWdtrt/tst/common/usdt/tst.guess64.ksh mode=0444
2028 file path=opt/SUNWdtrt/tst/common/usdt/tst.header.ksh mode=0444
2029 file path=opt/SUNWdtrt/tst/common/usdt/tst.include.ksh mode=0444
2030 file path=opt/SUNWdtrt/tst/common/usdt/tst.lazyprobe.exe mode=0555
2031 file path=opt/SUNWdtrt/tst/common/usdt/tst.lazyprobe1.ksh mode=0444
2032 file path=opt/SUNWdtrt/tst/common/usdt/tst.lazyprobe2.ksh mode=0444
2033 file path=opt/SUNWdtrt/tst/common/usdt/tst.linkpriv.ksh mode=0444
2034 file path=opt/SUNWdtrt/tst/common/usdt/tst.linkunpriv.ksh mode=0444
2035 file path=opt/SUNWdtrt/tst/common/usdt/tst.multiple.ksh mode=0444
2036 file path=opt/SUNWdtrt/tst/common/usdt/tst.multiple.ksh.out mode=0444
2037 file path=opt/SUNWdtrt/tst/common/usdt/tst.multiprov.ksh mode=0444
2038 file path=opt/SUNWdtrt/tst/common/usdt/tst.multiprov.ksh.out mode=0444
2039 file path=opt/SUNWdtrt/tst/common/usdt/tst.nodtrace.ksh mode=0444
2040 file path=opt/SUNWdtrt/tst/common/usdt/tst.noprobes.ksh mode=0444
```



```
2041 file path=opt/SUNWdtrt/tst/common/usdt/tst.noreap.ksh mode=0444
2042 file path=opt/SUNWdtrt/tst/common/usdt/tst.noreapring.ksh mode=0444
2043 file path=opt/SUNWdtrt/tst/common/usdt/tst.onlyenabled.ksh mode=0444
2044 file path=opt/SUNWdtrt/tst/common/usdt/tst.reap.ksh mode=0444
2045 file path=opt/SUNWdtrt/tst/common/usdt/tst.reeval.ksh mode=0444
2046 file path=opt/SUNWdtrt/tst/common/usdt/tst.static.ksh mode=0444
2047 file path=opt/SUNWdtrt/tst/common/usdt/tst.static.ksh.out mode=0444
2048 file path=opt/SUNWdtrt/tst/common/usdt/tst.static2.ksh mode=0444
2049 file path=opt/SUNWdtrt/tst/common/usdt/tst.static2.ksh.out mode=0444
2050 file path=opt/SUNWdtrt/tst/common/usdt/tst.user.ksh mode=0444
2051 file path=opt/SUNWdtrt/tst/common/usdt/tst.user.ksh.out mode=0444
2052 file path=opt/SUNWdtrt/tst/common/ustack/tst.bigstack.d mode=0444
2053 file path=opt/SUNWdtrt/tst/common/ustack/tst.bigstack.exe mode=0555
2054 file path=opt/SUNWdtrt/tst/common/ustack/tst.depth.ksh mode=0444
2055 file path=opt/SUNWdtrt/tst/common/ustack/tst.spin.exe mode=0555
2056 file path=opt/SUNWdtrt/tst/common/ustack/tst.spin.ksh mode=0444
2057 file path=opt/SUNWdtrt/tst/common/vars/tst.gid.d mode=0444
2058 file path=opt/SUNWdtrt/tst/common/vars/tst.nullassign.d mode=0444
2059 file path=opt/SUNWdtrt/tst/common/vars/tst.ppid.d mode=0444
2060 file path=opt/SUNWdtrt/tst/common/vars/tst.ucaller.ksh mode=0444
2061 file path=opt/SUNWdtrt/tst/common/vars/tst.ucaller.ksh.out mode=0444
2062 file path=opt/SUNWdtrt/tst/common/vars/tst.uid.d mode=0444
2063 file path=opt/SUNWdtrt/tst/common/vars/tst.walltimestamp.d mode=0444
2064 file path=opt/SUNWdtrt/tst/common/version/tst.1.0.d mode=0444
2065 $(i386_ONLY)file path=opt/SUNWdtrt/tst/i86xpv/xdt/tst.basic.ksh mode=0444
2066 $(i386_ONLY)file path=opt/SUNWdtrt/tst/i86xpv/xdt/tst.hvmenable.ksh mode=0444
2067 $(i386_ONLY)file path=opt/SUNWdtrt/tst/i86xpv/xdt/tst.memenable.ksh mode=0444
2068 $(i386_ONLY)file path=opt/SUNWdtrt/tst/i86xpv/xdt/tst.schedargs.ksh mode=0444
2069 $(i386_ONLY)file path=opt/SUNWdtrt/tst/i86xpv/xdt/tst.schedenable.ksh \
2070     mode=0444
2071 legacy pkg=SUNWdtrt category=internal \
2072     desc="DTrace Test Suite Internal Distribution" \
2073     hotline="Contact the DTrace discussion forum" name="DTrace Test Suite"
2074 license cr_Sun license=cr_Sun
2075 license lic_CDDL license=lic_CDDL
2076 depend fmri=runtime/java type=require
2077 depend fmri=runtime/java/runtime64 type=require
```

```
**********************************************************
  435357 Tue Jan 14 16:49:36 2014
new/usr/src/uts/common/dtrace/dtrace.c
4477 DTrace should speak JSON
Reviewed by: Bryan Cantrill <bmc@joyent.com>
**********************************************************
```

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

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

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



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

/*
 * DTrace Tunable Variables
 *
 * The following variables may be tuned by adding a line to /etc/system that
 * includes both the name of the DTrace module ("dtrace") and the name of the
 * variable.  For example:
 *
 *   set dtrace:dtrace_destructive_disallow = 1
 *
 * In general, the only variables that one should be tuning this way are those
 * that affect system-wide DTrace behavior, and for which the default behavior
 * is undesirable.  Most of these variables are tunable on a per-consumer
 * basis using DTrace options, and need not be tuned on a system-wide basis.
 * When tuning these variables, avoid pathological values; while some attempt
 * is made to verify the integrity of these variables, they are not considered
 * part of the supported interface to DTrace, and they are therefore not
 * checked comprehensively.  Further, these variables should not be tuned
 * dynamically via "mdb -kw" or other means; they should only be tuned via
 * /etc/system.
 */
int             dtrace_destructive_disallow = 0;
dtrace_optval_t dtrace_nonroot_maxsize = (16 * 1024 * 1024);
size_t          dtrace_difo_maxsize = (256 * 1024);
dtrace_optval_t dtrace_dof_maxsize = (256 * 1024);
size_t          dtrace_global_maxsize = (16 * 1024);
size_t          dtrace_actions_max = (16 * 1024);
size_t          dtrace_retain_max = 1024;
dtrace_optval_t dtrace_helper_actions_max = 1024;
dtrace_optval_t dtrace_helper_providers_max = 32;
```

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

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

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



```
static dtrace_dynvar_t  dtrace_dynhash_sink;    /* end of dynamic hash chains */
static int              dtrace_dynvar_failclean; /* dynvars failed to clean */

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

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

static void
dtrace_nullop(void)
{}

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

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

```
259         (void (*)(void *, dtrace_id_t, void *))dtrace_nullop,
260         (void (*)(void *, dtrace_id_t, void *))dtrace_nullop,
261         NULL,
262         NULL,
263         NULL,
264         (void (*)(void *, dtrace_id_t, void *))dtrace_nullop
265 };

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

271 /*
272  * DTrace Helper Tracing Variables
273  */
274 uint32_t dtrace_helptrace_next = 0;
275 uint32_t dtrace_helptrace_nlocals;
276 char    *dtrace_helptrace_buffer;
277 int     dtrace_helptrace_bufsize = 512 * 1024;

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

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

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

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

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

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

322 #define DTRACE_AGGHASHSIZE_SLEW         17

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



```
326 /*
327  * The key for a thread-local variable consists of the lower 61 bits of the
328  * t_did, plus the 3 bits of the highest active interrupt above LOCK_LEVEL.
329  * We add DIF_VARIABLE_MAX to t_did to assure that the thread key is never
330  * equal to a variable identifier.  This is necessary (but not sufficient) to
331  * assure that global associative arrays never collide with thread-local
332  * variables.  To guarantee that they cannot collide, we must also define the
333  * order for keying dynamic variables.  That order is:
334  *
335  *   [ key0 ] ... [ keyn ] [ variable-key ] [ tls-key ]
336  *
337  * Because the variable-key and the tls-key are in orthogonal spaces, there is
338  * no way for a global variable key signature to match a thread-local key
339  * signature.
340  */
341 #define DTRACE_TLS_THRKEY(where) { \
342         uint_t intr = 0; \
343         uint_t actv = CPU->cpu_intr_actv >> (LOCK_LEVEL + 1); \
344         for (; actv; actv >>= 1) \
345                 intr++; \
346         ASSERT(intr < (1 << 3)); \
347         (where) = ((curthread->t_did + DIF_VARIABLE_MAX) & \
348             (((uint64_t)1 << 61) - 1)) | ((uint64_t)intr << 61); \
349 }

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

356 #define DT_MASK_LO 0x00000000FFFFFFFFULL

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

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

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

383 /*
384  * Test whether alloc_sz bytes will fit in the scratch region.  We isolate
385  * alloc_sz on the righthand side of the comparison in order to avoid overflow
386  * or underflow in the comparison with it.  This is simpler than the INRANGE
387  * check above, because we know that the dtms_scratch_ptr is valid in the
388  * range.  Allocations of size zero are allowed.
389  */
390 #define DTRACE_INSCRATCH(mstate, alloc_sz) \
```

```
391         ((mstate)->dtms_scratch_base + (mstate)->dtms_scratch_size - \
392         (mstate)->dtms_scratch_ptr >= (alloc_sz))

394 #define DTRACE_LOADFUNC(bits)                                           \
395 /*CSTYLED*/                                                             \
396 uint##bits##_t                                                          \
397 dtrace_load##bits(uintptr_t addr)                                       \
398 {                                                                       \
399         size_t size = bits / NBBY;                                      \
400         /*CSTYLED*/                                                     \
401         uint##bits##_t rval;                                            \
402         int i;                                                          \
403         volatile uint16_t *flags = (volatile uint16_t *)                \
404             &cpu_core[CPU->cpu_id].cpuc_dtrace_flags;                   \
405                                                                         \
406         DTRACE_ALIGNCHECK(addr, size, flags);                           \
407                                                                         \
408         for (i = 0; i < dtrace_toxranges; i++) {                        \
409                 if (addr >= dtrace_toxrange[i].dtt_limit)               \
410                         continue;                                       \
411                                                                         \
412                 if (addr + size <= dtrace_toxrange[i].dtt_base)         \
413                         continue;                                       \
414                                                                         \
415                 /*                                                      \
416                  * This address falls within a toxic region; return 0.  \
417                  */                                                     \
418                 *flags |= CPU_DTRACE_BADADDR;                           \
419                 cpu_core[CPU->cpu_id].cpuc_dtrace_illval = addr;        \
420                 return (0);                                             \
421         }                                                               \
422                                                                         \
423         *flags |= CPU_DTRACE_NOFAULT;                                   \
424         /*CSTYLED*/                                                     \
425         rval = *((volatile uint##bits##_t *)addr);                      \
426         *flags &= ~CPU_DTRACE_NOFAULT;                                  \
427                                                                         \
428         return (!(*flags & CPU_DTRACE_FAULT) ? rval : 0);               \
429 }

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

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

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

447 #define DTRACE_FLAGS2FLT(flags)                                         \
448         (((flags) & CPU_DTRACE_BADADDR) ? DTRACEFLT_BADADDR :           \
449         ((flags) & CPU_DTRACE_ILLOP) ? DTRACEFLT_ILLOP :                \
450         ((flags) & CPU_DTRACE_DIVZERO) ? DTRACEFLT_DIVZERO :            \
451         ((flags) & CPU_DTRACE_KPRIV) ? DTRACEFLT_KPRIV :                \
452         ((flags) & CPU_DTRACE_UPRIV) ? DTRACEFLT_UPRIV :                \
453         ((flags) & CPU_DTRACE_TUPOFLOW) ?  DTRACEFLT_TUPOFLOW :         \
454         ((flags) & CPU_DTRACE_BADALIGN) ?  DTRACEFLT_BADALIGN :         \
455         ((flags) & CPU_DTRACE_NOSCRATCH) ?  DTRACEFLT_NOSCRATCH :       \
456         ((flags) & CPU_DTRACE_BADSTACK) ?  DTRACEFLT_BADSTACK :         \
```



```
457         DTRACEFLT_UNKNOWN)

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

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

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

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

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

516         /*
517          * We just need something here that even the most clever compiler
518          * cannot optimize away.
519          */
520         return (a[(uintptr_t)f]);
521 }
```

```
523 /*
524  * Atomically increment a specified error counter from probe context.
525  */
526 static void
527 dtrace_error(uint32_t *counter)
528 {
529         /*
530          * Most counters stored to in probe context are per-CPU counters.
531          * However, there are some error conditions that are sufficiently
532          * arcane that they don't merit per-CPU storage.  If these counters
533          * are incremented concurrently on different CPUs, scalability will be
534          * adversely affected -- but we don't expect them to be white-hot in a
535          * correctly constructed enabling...
536          */
537         uint32_t oval, nval;
538
539         do {
540                 oval = *counter;
541
542                 if ((nval = oval + 1) == 0) {
543                         /*
544                          * If the counter would wrap, set it to 1 -- assuring
545                          * that the counter is never zero when we have seen
546                          * errors.  (The counter must be 32-bits because we
547                          * aren't guaranteed a 64-bit compare&swap operation.)
548                          * To save this code both the infamy of being fingered
549                          * by a priggish news story and the indignity of being
550                          * the target of a neo-puritan witch trial, we're
551                          * carefully avoiding any colorful description of the
552                          * likelihood of this condition -- but suffice it to
553                          * say that it is only slightly more likely than the
554                          * overflow of predicate cache IDs, as discussed in
555                          * dtrace_predicate_create().
556                          */
557                         nval = 1;
558                 }
559         } while (dtrace_cas32(counter, oval, nval) != oval);
560 }
561
562 /*
563  * Use the DTRACE_LOADFUNC macro to define functions for each of loading a
564  * uint8_t, a uint16_t, a uint32_t and a uint64_t.
565  */
566 DTRACE_LOADFUNC(8)
567 DTRACE_LOADFUNC(16)
568 DTRACE_LOADFUNC(32)
569 DTRACE_LOADFUNC(64)
570
571 static int
572 dtrace_inscratch(uintptr_t dest, size_t size, dtrace_mstate_t *mstate)
573 {
574         if (dest < mstate->dtms_scratch_base)
575                 return (0);
576
577         if (dest + size < dest)
578                 return (0);
579
580         if (dest + size > mstate->dtms_scratch_ptr)
581                 return (0);
582
583         return (1);
584 }
585
586 static int
587 dtrace_canstore_statvar(uint64_t addr, size_t sz,
588     dtrace_statvar_t **svars, int nsvars)
```



```
589 {
590         int i;
591
592         for (i = 0; i < nsvars; i++) {
593                 dtrace_statvar_t *svar = svars[i];
594
595                 if (svar == NULL || svar->dtsv_size == 0)
596                         continue;
597
598                 if (DTRACE_INRANGE(addr, sz, svar->dtsv_data, svar->dtsv_size))
599                         return (1);
600         }
601
602         return (0);
603 }
604
605 /*
606  * Check to see if the address is within a memory region to which a store may
607  * be issued.  This includes the DTrace scratch areas, and any DTrace variable
608  * region.  The caller of dtrace_canstore() is responsible for performing any
609  * alignment checks that are needed before stores are actually executed.
610  */
611 static int
612 dtrace_canstore(uint64_t addr, size_t sz, dtrace_mstate_t *mstate,
613     dtrace_vstate_t *vstate)
614 {
615         /*
616          * First, check to see if the address is in scratch space...
617          */
618         if (DTRACE_INRANGE(addr, sz, mstate->dtms_scratch_base,
619             mstate->dtms_scratch_size))
620                 return (1);
621
622         /*
623          * Now check to see if it's a dynamic variable.  This check will pick
624          * up both thread-local variables and any global dynamically-allocated
625          * variables.
626          */
627         if (DTRACE_INRANGE(addr, sz, vstate->dtvs_dynvars.dtds_base,
628             vstate->dtvs_dynvars.dtds_size)) {
629                 dtrace_dstate_t *dstate = &vstate->dtvs_dynvars;
630                 uintptr_t base = (uintptr_t)dstate->dtds_base +
631                     (dstate->dtds_hashsize * sizeof (dtrace_dynhash_t));
632                 uintptr_t chunkoffs;
633
634                 /*
635                  * Before we assume that we can store here, we need to make
636                  * sure that it isn't in our metadata -- storing to our
637                  * dynamic variable metadata would corrupt our state.  For
638                  * the range to not include any dynamic variable metadata,
639                  * it must:
640                  *
641                  *      (1) Start above the hash table that is at the base of
642                  *      the dynamic variable space
643                  *
644                  *      (2) Have a starting chunk offset that is beyond the
645                  *      dtrace_dynvar_t that is at the base of every chunk
646                  *
647                  *      (3) Not span a chunk boundary
648                  *
649                  */
650                 if (addr < base)
651                         return (0);
652
653                 chunkoffs = (addr - base) % dstate->dtds_chunksize;
```

```
655                 if (chunkoffs < sizeof (dtrace_dynvar_t))
656                         return (0);
657
658                 if (chunkoffs + sz > dstate->dtds_chunksize)
659                         return (0);
660
661                 return (1);
662         }
663
664         /*
665          * Finally, check the static local and global variables.  These checks
666          * take the longest, so we perform them last.
667          */
668         if (dtrace_canstore_statvar(addr, sz,
669             vstate->dtvs_locals, vstate->dtvs_nlocals))
670                 return (1);
671
672         if (dtrace_canstore_statvar(addr, sz,
673             vstate->dtvs_globals, vstate->dtvs_nglobals))
674                 return (1);
675
676         return (0);
677 }
678
679
680 /*
681  * Convenience routine to check to see if the address is within a memory
682  * region in which a load may be issued given the user's privilege level;
683  * if not, it sets the appropriate error flags and loads 'addr' into the
684  * illegal value slot.
685  *
686  * DTrace subroutines (DIF_SUBR_*) should use this helper to implement
687  * appropriate memory access protection.
688  */
689 static int
690 dtrace_canload(uint64_t addr, size_t sz, dtrace_mstate_t *mstate,
691     dtrace_vstate_t *vstate)
692 {
693         volatile uintptr_t *illval = &cpu_core[CPU->cpu_id].cpuc_dtrace_illval;
694         file_t *fp;
695
696         /*
697          * If we hold the privilege to read from kernel memory, then
698          * everything is readable.
699          */
700         if ((mstate->dtms_access & DTRACE_ACCESS_KERNEL) != 0)
701                 return (1);
702
703         /*
704          * You can obviously read that which you can store.
705          */
706         if (dtrace_canstore(addr, sz, mstate, vstate))
707                 return (1);
708
709         /*
710          * We're allowed to read from our own string table.
711          */
712         if (DTRACE_INRANGE(addr, sz, mstate->dtms_difo->dtdo_strtab,
713             mstate->dtms_difo->dtdo_strlen))
714                 return (1);
715
716         if (vstate->dtvs_state != NULL &&
717             dtrace_priv_proc(vstate->dtvs_state, mstate)) {
718                 proc_t *p;
719
720                 /*
```



```
721                  * When we have privileges to the current process, there are
722                  * several context-related kernel structures that are safe to
723                  * read, even absent the privilege to read from kernel memory.
724                  * These reads are safe because these structures contain only
725                  * state that (1) we're permitted to read, (2) is harmless or
726                  * (3) contains pointers to additional kernel state that we're
727                  * not permitted to read (and as such, do not present an
728                  * opportunity for privilege escalation).  Finally (and
729                  * critically), because of the nature of their relation with
730                  * the current thread context, the memory associated with these
731                  * structures cannot change over the duration of probe context,
732                  * and it is therefore impossible for this memory to be
733                  * deallocated and reallocated as something else while it's
734                  * being operated upon.
735                  */
736                 if (DTRACE_INRANGE(addr, sz, curthread, sizeof (kthread_t)))
737                         return (1);
738
739                 if ((p = curthread->t_procp) != NULL && DTRACE_INRANGE(addr,
740                     sz, curthread->t_procp, sizeof (proc_t))) {
741                         return (1);
742                 }
743
744                 if (curthread->t_cred != NULL && DTRACE_INRANGE(addr, sz,
745                     curthread->t_cred, sizeof (cred_t))) {
746                         return (1);
747                 }
748
749                 if (p != NULL && p->p_pidp != NULL && DTRACE_INRANGE(addr, sz,
750                     &(p->p_pidp->pid_id), sizeof (pid_t))) {
751                         return (1);
752                 }
753
754                 if (curthread->t_cpu != NULL && DTRACE_INRANGE(addr, sz,
755                     curthread->t_cpu, offsetof(cpu_t, cpu_pause_thread))) {
756                         return (1);
757                 }
758         }
759
760         if ((fp = mstate->dtms_getf) != NULL) {
761                 uintptr_t psz = sizeof (void *);
762                 vnode_t *vp;
763                 vnodeops_t *op;
764
765                 /*
766                  * When getf() returns a file_t, the enabling is implicitly
767                  * granted the (transient) right to read the returned file_t
768                  * as well as the v_path and v_op->vnop_name of the underlying
769                  * vnode.  These accesses are allowed after a successful
770                  * getf() because the members that they refer to cannot change
771                  * once set -- and the barrier logic in the kernel's closef()
772                  * path assures that the file_t and its referenced vode_t
773                  * cannot themselves be stale (that is, it impossible for
774                  * either dtms_getf itself or its f_vnode member to reference
775                  * freed memory).
776                  */
777                 if (DTRACE_INRANGE(addr, sz, fp, sizeof (file_t)))
778                         return (1);
779
780                 if ((vp = fp->f_vnode) != NULL) {
781                         if (DTRACE_INRANGE(addr, sz, &vp->v_path, psz))
782                                 return (1);
783
784                         if (vp->v_path != NULL && DTRACE_INRANGE(addr, sz,
785                             vp->v_path, strlen(vp->v_path) + 1)) {
786                                 return (1);
```

```
787                         }
788
789                         if (DTRACE_INRANGE(addr, sz, &vp->v_op, psz))
790                                 return (1);
791
792                         if ((op = vp->v_op) != NULL &&
793                             DTRACE_INRANGE(addr, sz, &op->vnop_name, psz)) {
794                                 return (1);
795                         }
796
797                         if (op != NULL && op->vnop_name != NULL &&
798                             DTRACE_INRANGE(addr, sz, op->vnop_name,
799                             strlen(op->vnop_name) + 1)) {
800                                 return (1);
801                         }
802                 }
803         }
804
805         DTRACE_CPUFLAG_SET(CPU_DTRACE_KPRIV);
806         *illval = addr;
807         return (0);
808 }
809
810 /*
811  * Convenience routine to check to see if a given string is within a memory
812  * region in which a load may be issued given the user's privilege level;
813  * this exists so that we don't need to issue unnecessary dtrace_strlen()
814  * calls in the event that the user has all privileges.
815  */
816 static int
817 dtrace_strcanload(uint64_t addr, size_t sz, dtrace_mstate_t *mstate,
818     dtrace_vstate_t *vstate)
819 {
820         size_t strsz;
821
822         /*
823          * If we hold the privilege to read from kernel memory, then
824          * everything is readable.
825          */
826         if ((mstate->dtms_access & DTRACE_ACCESS_KERNEL) != 0)
827                 return (1);
828
829         strsz = 1 + dtrace_strlen((char *)(uintptr_t)addr, sz);
830         if (dtrace_canload(addr, strsz, mstate, vstate))
831                 return (1);
832
833         return (0);
834 }
835
836 /*
837  * Convenience routine to check to see if a given variable is within a memory
838  * region in which a load may be issued given the user's privilege level.
839  */
840 static int
841 dtrace_vcanload(void *src, dtrace_diftype_t *type, dtrace_mstate_t *mstate,
842     dtrace_vstate_t *vstate)
843 {
844         size_t sz;
845         ASSERT(type->dtdt_flags & DIF_TF_BYREF);
846
847         /*
848          * If we hold the privilege to read from kernel memory, then
849          * everything is readable.
850          */
851         if ((mstate->dtms_access & DTRACE_ACCESS_KERNEL) != 0)
852                 return (1);
```



```
853
854         if (type->dtdt_kind == DIF_TYPE_STRING)
855                 sz = dtrace_strlen(src,
856                     vstate->dtvs_state->dts_options[DTRACEOPT_STRSIZE]) + 1;
857         else
858                 sz = type->dtdt_size;
859
860         return (dtrace_canload((uintptr_t)src, sz, mstate, vstate));
861 }
862
863 /*
864  * Convert a string to a signed integer using safe loads.
865  *
866  * NOTE: This function uses various macros from strtolctype.h to manipulate
867  * digit values, etc -- these have all been checked to ensure they make
868  * no additional function calls.
869  */
870 static int64_t
871 dtrace_strtoll(char *input, int base, size_t limit)
872 {
873         uintptr_t pos = (uintptr_t)input;
874         int64_t val = 0;
875         int x;
876         boolean_t neg = B_FALSE;
877         char c, cc, ccc;
878         uintptr_t end = pos + limit;
879
880         /*
881          * Consume any whitespace preceding digits.
882          */
883         while ((c = dtrace_load8(pos)) == ' ' || c == '\t')
884                 pos++;
885
886         /*
887          * Handle an explicit sign if one is present.
888          */
889         if (c == '-' || c == '+') {
890                 if (c == '-')
891                         neg = B_TRUE;
892                 c = dtrace_load8(++pos);
893         }
894
895         /*
896          * Check for an explicit hexadecimal prefix ("0x" or "0X") and skip it
897          * if present.
898          */
899         if (base == 16 && c == '0' && ((cc = dtrace_load8(pos + 1)) == 'x' ||
900             cc == 'X') && isxdigit(ccc = dtrace_load8(pos + 2))) {
901                 pos += 2;
902                 c = ccc;
903         }
904
905         /*
906          * Read in contiguous digits until the first non-digit character.
907          */
908         for (; pos < end && c != '\0' && lisalnum(c) && (x = DIGIT(c)) < base;
909             c = dtrace_load8(++pos))
910                 val = val * base + x;
911
912         return (neg ? -val : val);
913 }
914
915 /*
916 #endif /* ! codereview */
917  * Compare two strings using safe loads.
918  */
```

```
static int
dtrace_strncmp(char *s1, char *s2, size_t limit)
{
        uint8_t c1, c2;
        volatile uint16_t *flags;

        if (s1 == s2 || limit == 0)
                return (0);

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

        do {
                if (s1 == NULL) {
                        c1 = '\0';
                } else {
                        c1 = dtrace_load8((uintptr_t)s1++);
                }

                if (s2 == NULL) {
                        c2 = '\0';
                } else {
                        c2 = dtrace_load8((uintptr_t)s2++);
                }

                if (c1 != c2)
                        return (c1 - c2);
        } while (--limit && c1 != '\0' && !(*flags & CPU_DTRACE_FAULT));

        return (0);
}

/*
 * Compute strlen(s) for a string using safe memory accesses.  The additional
 * len parameter is used to specify a maximum length to ensure completion.
 */
static size_t
dtrace_strlen(const char *s, size_t lim)
{
        uint_t len;

        for (len = 0; len != lim; len++) {
                if (dtrace_load8((uintptr_t)s++) == '\0')
                        break;
        }

        return (len);
}

/*
 * Check if an address falls within a toxic region.
 */
static int
dtrace_istoxic(uintptr_t kaddr, size_t size)
{
        uintptr_t taddr, tsize;
        int i;

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

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



```
                if (taddr - kaddr < size) {
                        DTRACE_CPUFLAG_SET(CPU_DTRACE_BADADDR);
                        cpu_core[CPU->cpu_id].cpuc_dtrace_illval = taddr;
                        return (1);
                }
        }

        return (0);
}

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

                if (s1 <= s2) {
                        do {
                                *s1++ = dtrace_load8((uintptr_t)s2++);
                        } while (--len != 0);
                } else {
                        s2 += len;
                        s1 += len;

                        do {
                                *--s1 = dtrace_load8((uintptr_t)--s2);
                        } while (--len != 0);
                }
        }
}

/*
 * Copy src to dst using safe memory accesses, up to either the specified
 * length, or the point that a nul byte is encountered.  The src is assumed to
 * be unsafe memory specified by the DIF program.  The dst is assumed to be
 * safe memory that we can store to directly because it is managed by DTrace.
 * Unlike dtrace_bcopy(), overlapping regions are not handled.
 */
static void
dtrace_strcpy(const void *src, void *dst, size_t len)
{
        if (len != 0) {
                uint8_t *s1 = dst, c;
                const uint8_t *s2 = src;

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

/*
 * Copy src to dst, deriving the size and type from the specified (BYREF)
 * variable type.  The src is assumed to be unsafe memory specified by the DIF
 * program.  The dst is assumed to be DTrace variable memory that is of the
 * specified type; we assume that we can store to directly.
 */
static void
```

```
1051 dtrace_vcopy(void *src, void *dst, dtrace_diftype_t *type)
1052 {
1053         ASSERT(type->dtdt_flags & DIF_TF_BYREF);

1055         if (type->dtdt_kind == DIF_TYPE_STRING) {
1056                 dtrace_strcpy(src, dst, type->dtdt_size);
1057         } else {
1058                 dtrace_bcopy(src, dst, type->dtdt_size);
1059         }
1060 }

1062 /*
1063  * Compare s1 to s2 using safe memory accesses.  The s1 data is assumed to be
1064  * unsafe memory specified by the DIF program.  The s2 data is assumed to be
1065  * safe memory that we can access directly because it is managed by DTrace.
1066  */
1067 static int
1068 dtrace_bcmp(const void *s1, const void *s2, size_t len)
1069 {
1070         volatile uint16_t *flags;

1072         flags = (volatile uint16_t *)&cpu_core[CPU->cpu_id].cpuc_dtrace_flags;

1074         if (s1 == s2)
1075                 return (0);

1077         if (s1 == NULL || s2 == NULL)
1078                 return (1);

1080         if (s1 != s2 && len != 0) {
1081                 const uint8_t *ps1 = s1;
1082                 const uint8_t *ps2 = s2;

1084                 do {
1085                         if (dtrace_load8((uintptr_t)ps1++) != *ps2++)
1086                                 return (1);
1087                 } while (--len != 0 && !(*flags & CPU_DTRACE_FAULT));
1088         }
1089         return (0);
1090 }

1092 /*
1093  * Zero the specified region using a simple byte-by-byte loop.  Note that this
1094  * is for safe DTrace-managed memory only.
1095  */
1096 static void
1097 dtrace_bzero(void *dst, size_t len)
1098 {
1099         uchar_t *cp;

1101         for (cp = dst; len != 0; len--)
1102                 *cp++ = 0;
1103 }

1105 static void
1106 dtrace_add_128(uint64_t *addend1, uint64_t *addend2, uint64_t *sum)
1107 {
1108         uint64_t result[2];

1110         result[0] = addend1[0] + addend2[0];
1111         result[1] = addend1[1] + addend2[1] +
1112             (result[0] < addend1[0] || result[0] < addend2[0] ? 1 : 0);

1114         sum[0] = result[0];
1115         sum[1] = result[1];
1116 }
```



```
1118 /*
1119  * Shift the 128-bit value in a by b. If b is positive, shift left.
1120  * If b is negative, shift right.
1121  */
1122 static void
1123 dtrace_shift_128(uint64_t *a, int b)
1124 {
1125         uint64_t mask;

1127         if (b == 0)
1128                 return;

1130         if (b < 0) {
1131                 b = -b;
1132                 if (b >= 64) {
1133                         a[0] = a[1] >> (b - 64);
1134                         a[1] = 0;
1135                 } else {
1136                         a[0] >>= b;
1137                         mask = 1LL << (64 - b);
1138                         mask -= 1;
1139                         a[0] |= ((a[1] & mask) << (64 - b));
1140                         a[1] >>= b;
1141                 }
1142         } else {
1143                 if (b >= 64) {
1144                         a[1] = a[0] << (b - 64);
1145                         a[0] = 0;
1146                 } else {
1147                         a[1] <<= b;
1148                         mask = a[0] >> (64 - b);
1149                         a[1] |= mask;
1150                         a[0] <<= b;
1151                 }
1152         }
1153 }

1155 /*
1156  * The basic idea is to break the 2 64-bit values into 4 32-bit values,
1157  * use native multiplication on those, and then re-combine into the
1158  * resulting 128-bit value.
1159  *
1160  * (hi1 << 32 + lo1) * (hi2 << 32 + lo2) =
1161  *     hi1 * hi2 << 64 +
1162  *     hi1 * lo2 << 32 +
1163  *     hi2 * lo1 << 32 +
1164  *     lo1 * lo2
1165  */
1166 static void
1167 dtrace_multiply_128(uint64_t factor1, uint64_t factor2, uint64_t *product)
1168 {
1169         uint64_t hi1, hi2, lo1, lo2;
1170         uint64_t tmp[2];

1172         hi1 = factor1 >> 32;
1173         hi2 = factor2 >> 32;

1175         lo1 = factor1 & DT_MASK_LO;
1176         lo2 = factor2 & DT_MASK_LO;

1178         product[0] = lo1 * lo2;
1179         product[1] = hi1 * hi2;

1181         tmp[0] = hi1 * lo2;
1182         tmp[1] = 0;
```

```
1183         dtrace_shift_128(tmp, 32);
1184         dtrace_add_128(product, tmp, product);

1186         tmp[0] = hi2 * lo1;
1187         tmp[1] = 0;
1188         dtrace_shift_128(tmp, 32);
1189         dtrace_add_128(product, tmp, product);
1190 }

1192 /*
1193  * This privilege check should be used by actions and subroutines to
1194  * verify that the user credentials of the process that enabled the
1195  * invoking ECB match the target credentials
1196  */
1197 static int
1198 dtrace_priv_proc_common_user(dtrace_state_t *state)
1199 {
1200         cred_t *cr, *s_cr = state->dts_cred.dcr_cred;

1202         /*
1203          * We should always have a non-NULL state cred here, since if cred
1204          * is null (anonymous tracing), we fast-path bypass this routine.
1205          */
1206         ASSERT(s_cr != NULL);

1208         if ((cr = CRED()) != NULL &&
1209             s_cr->cr_uid == cr->cr_uid &&
1210             s_cr->cr_uid == cr->cr_ruid &&
1211             s_cr->cr_uid == cr->cr_suid &&
1212             s_cr->cr_gid == cr->cr_gid &&
1213             s_cr->cr_gid == cr->cr_rgid &&
1214             s_cr->cr_gid == cr->cr_sgid)
1215                 return (1);

1217         return (0);
1218 }

1220 /*
1221  * This privilege check should be used by actions and subroutines to
1222  * verify that the zone of the process that enabled the invoking ECB
1223  * matches the target credentials
1224  */
1225 static int
1226 dtrace_priv_proc_common_zone(dtrace_state_t *state)
1227 {
1228         cred_t *cr, *s_cr = state->dts_cred.dcr_cred;

1230         /*
1231          * We should always have a non-NULL state cred here, since if cred
1232          * is null (anonymous tracing), we fast-path bypass this routine.
1233          */
1234         ASSERT(s_cr != NULL);

1236         if ((cr = CRED()) != NULL && s_cr->cr_zone == cr->cr_zone)
1237                 return (1);

1239         return (0);
1240 }

1242 /*
1243  * This privilege check should be used by actions and subroutines to
1244  * verify that the process has not setuid or changed credentials.
1245  */
1246 static int
1247 dtrace_priv_proc_common_nocd()
1248 {
```



```
1249         proc_t *proc;

1251         if ((proc = ttoproc(curthread)) != NULL &&
1252             !(proc->p_flag & SNOCD))
1253                 return (1);

1255         return (0);
1256 }

1258 static int
1259 dtrace_priv_proc_destructive(dtrace_state_t *state, dtrace_mstate_t *mstate)
1260 {
1261         int action = state->dts_cred.dcr_action;

1263         if (!(mstate->dtms_access & DTRACE_ACCESS_PROC))
1264                 goto bad;

1266         if (((action & DTRACE_CRA_PROC_DESTRUCTIVE_ALLZONE) == 0) &&
1267             dtrace_priv_proc_common_zone(state) == 0)
1268                 goto bad;

1270         if (((action & DTRACE_CRA_PROC_DESTRUCTIVE_ALLUSER) == 0) &&
1271             dtrace_priv_proc_common_user(state) == 0)
1272                 goto bad;

1274         if (((action & DTRACE_CRA_PROC_DESTRUCTIVE_CREDCHG) == 0) &&
1275             dtrace_priv_proc_common_nocd() == 0)
1276                 goto bad;

1278         return (1);

1280 bad:
1281         cpu_core[CPU->cpu_id].cpuc_dtrace_flags |= CPU_DTRACE_UPRIV;

1283         return (0);
1284 }

1286 static int
1287 dtrace_priv_proc_control(dtrace_state_t *state, dtrace_mstate_t *mstate)
1288 {
1289         if (mstate->dtms_access & DTRACE_ACCESS_PROC) {
1290                 if (state->dts_cred.dcr_action & DTRACE_CRA_PROC_CONTROL)
1291                         return (1);

1293                 if (dtrace_priv_proc_common_zone(state) &&
1294                     dtrace_priv_proc_common_user(state) &&
1295                     dtrace_priv_proc_common_nocd())
1296                         return (1);
1297         }

1299         cpu_core[CPU->cpu_id].cpuc_dtrace_flags |= CPU_DTRACE_UPRIV;

1301         return (0);
1302 }

1304 static int
1305 dtrace_priv_proc(dtrace_state_t *state, dtrace_mstate_t *mstate)
1306 {
1307         if ((mstate->dtms_access & DTRACE_ACCESS_PROC) &&
1308             (state->dts_cred.dcr_action & DTRACE_CRA_PROC))
1309                 return (1);

1311         cpu_core[CPU->cpu_id].cpuc_dtrace_flags |= CPU_DTRACE_UPRIV;

1313         return (0);
1314 }
```

```
1316 static int
1317 dtrace_priv_kernel(dtrace_state_t *state)
1318 {
1319         if (state->dts_cred.dcr_action & DTRACE_CRA_KERNEL)
1320                 return (1);
1322         cpu_core[CPU->cpu_id].cpuc_dtrace_flags |= CPU_DTRACE_KPRIV;
1324         return (0);
1325 }
1327 static int
1328 dtrace_priv_kernel_destructive(dtrace_state_t *state)
1329 {
1330         if (state->dts_cred.dcr_action & DTRACE_CRA_KERNEL_DESTRUCTIVE)
1331                 return (1);
1333         cpu_core[CPU->cpu_id].cpuc_dtrace_flags |= CPU_DTRACE_KPRIV;
1335         return (0);
1336 }
1338 /*
1339  * Determine if the dte_cond of the specified ECB allows for processing of
1340  * the current probe to continue.  Note that this routine may allow continued
1341  * processing, but with access(es) stripped from the mstate's dtms_access
1342  * field.
1343  */
1344 static int
1345 dtrace_priv_probe(dtrace_state_t *state, dtrace_mstate_t *mstate,
1346     dtrace_ecb_t *ecb)
1347 {
1348         dtrace_probe_t *probe = ecb->dte_probe;
1349         dtrace_provider_t *prov = probe->dtpr_provider;
1350         dtrace_pops_t *pops = &prov->dtpv_pops;
1351         int mode = DTRACE_MODE_NOPRIV_DROP;
1353         ASSERT(ecb->dte_cond);
1355         if (pops->dtps_mode != NULL) {
1356                 mode = pops->dtps_mode(prov->dtpv_arg,
1357                     probe->dtpr_id, probe->dtpr_arg);
1359                 ASSERT(mode & (DTRACE_MODE_USER | DTRACE_MODE_KERNEL));
1360                 ASSERT(mode & (DTRACE_MODE_NOPRIV_RESTRICT |
1361                     DTRACE_MODE_NOPRIV_DROP));
1362         }
1364         /*
1365          * If the dte_cond bits indicate that this consumer is only allowed to
1366          * see user-mode firings of this probe, check that the probe was fired
1367          * while in a user context.  If that's not the case, use the policy
1368          * specified by the provider to determine if we drop the probe or
1369          * merely restrict operation.
1370          */
1371         if (ecb->dte_cond & DTRACE_COND_USERMODE) {
1372                 ASSERT(mode != DTRACE_MODE_NOPRIV_DROP);
1374                 if (!(mode & DTRACE_MODE_USER)) {
1375                         if (mode & DTRACE_MODE_NOPRIV_DROP)
1376                                 return (0);
1378                         mstate->dtms_access &= ~DTRACE_ACCESS_ARGS;
1379                 }
1380         }
```



```
1382         /*
1383          * This is more subtle than it looks. We have to be absolutely certain
1384          * that CRED() isn't going to change out from under us so it's only
1385          * legit to examine that structure if we're in constrained situations.
1386          * Currently, the only times we'll this check is if a non-super-user
1387          * has enabled the profile or syscall providers -- providers that
1388          * allow visibility of all processes. For the profile case, the check
1389          * above will ensure that we're examining a user context.
1390          */
1391         if (ecb->dte_cond & DTRACE_COND_OWNER) {
1392                 cred_t *cr;
1393                 cred_t *s_cr = state->dts_cred.dcr_cred;
1394                 proc_t *proc;
1396                 ASSERT(s_cr != NULL);
1398                 if ((cr = CRED()) == NULL ||
1399                     s_cr->cr_uid != cr->cr_uid ||
1400                     s_cr->cr_uid != cr->cr_ruid ||
1401                     s_cr->cr_uid != cr->cr_suid ||
1402                     s_cr->cr_gid != cr->cr_gid ||
1403                     s_cr->cr_gid != cr->cr_rgid ||
1404                     s_cr->cr_gid != cr->cr_sgid ||
1405                     (proc = ttoproc(curthread)) == NULL ||
1406                     (proc->p_flag & SNOCD)) {
1407                         if (mode & DTRACE_MODE_NOPRIV_DROP)
1408                                 return (0);
1410                         mstate->dtms_access &= ~DTRACE_ACCESS_PROC;
1411                 }
1412         }
1414         /*
1415          * If our dte_cond is set to DTRACE_COND_ZONEOWNER and we are not
1416          * in our zone, check to see if our mode policy is to restrict rather
1417          * than to drop; if to restrict, strip away both DTRACE_ACCESS_PROC
1418          * and DTRACE_ACCESS_ARGS
1419          */
1420         if (ecb->dte_cond & DTRACE_COND_ZONEOWNER) {
1421                 cred_t *cr;
1422                 cred_t *s_cr = state->dts_cred.dcr_cred;
1424                 ASSERT(s_cr != NULL);
1426                 if ((cr = CRED()) == NULL ||
1427                     s_cr->cr_zone->zone_id != cr->cr_zone->zone_id) {
1428                         if (mode & DTRACE_MODE_NOPRIV_DROP)
1429                                 return (0);
1431                         mstate->dtms_access &=
1432                             ~(DTRACE_ACCESS_PROC | DTRACE_ACCESS_ARGS);
1433                 }
1434         }
1436         /*
1437          * By merits of being in this code path at all, we have limited
1438          * privileges.  If the provider has indicated that limited privileges
1439          * are to denote restricted operation, strip off the ability to access
1440          * arguments.
1441          */
1442         if (mode & DTRACE_MODE_LIMITEDPRIV_RESTRICT)
1443                 mstate->dtms_access &= ~DTRACE_ACCESS_ARGS;
1445         return (1);
1446 }
```

```
1448 /*
1449  * Note:  not called from probe context.  This function is called
1450  * asynchronously (and at a regular interval) from outside of probe context to
1451  * clean the dirty dynamic variable lists on all CPUs.  Dynamic variable
1452  * cleaning is explained in detail in <sys/dtrace_impl.h>.
1453  */
1454 void
1455 dtrace_dynvar_clean(dtrace_dstate_t *dstate)
1456 {
1457         dtrace_dynvar_t *dirty;
1458         dtrace_dstate_percpu_t *dcpu;
1459         dtrace_dynvar_t **rinsep;
1460         int i, j, work = 0;
1462         for (i = 0; i < NCPU; i++) {
1463                 dcpu = &dstate->dtds_percpu[i];
1464                 rinsep = &dcpu->dtdsc_rinsing;
1466                 /*
1467                  * If the dirty list is NULL, there is no dirty work to do.
1468                  */
1469                 if (dcpu->dtdsc_dirty == NULL)
1470                         continue;
1472                 if (dcpu->dtdsc_rinsing != NULL) {
1473                         /*
1474                          * If the rinsing list is non-NULL, then it is because
1475                          * this CPU was selected to accept another CPU's
1476                          * dirty list -- and since that time, dirty buffers
1477                          * have accumulated.  This is a highly unlikely
1478                          * condition, but we choose to ignore the dirty
1479                          * buffers -- they'll be picked up a future cleanse.
1480                          */
1481                         continue;
1482                 }
1484                 if (dcpu->dtdsc_clean != NULL) {
1485                         /*
1486                          * If the clean list is non-NULL, then we're in a
1487                          * situation where a CPU has done deallocations (we
1488                          * have a non-NULL dirty list) but no allocations (we
1489                          * also have a non-NULL clean list).  We can't simply
1490                          * move the dirty list into the clean list on this
1491                          * CPU, yet we also don't want to allow this condition
1492                          * to persist, lest a short clean list prevent a
1493                          * massive dirty list from being cleaned (which in
1494                          * turn could lead to otherwise avoidable dynamic
1495                          * drops).  To deal with this, we look for some CPU
1496                          * with a NULL clean list, NULL dirty list, and NULL
1497                          * rinsing list -- and then we borrow this CPU to
1498                          * rinse our dirty list.
1499                          */
1500                         for (j = 0; j < NCPU; j++) {
1501                                 dtrace_dstate_percpu_t *rinser;
1503                                 rinser = &dstate->dtds_percpu[j];
1505                                 if (rinser->dtdsc_rinsing != NULL)
1506                                         continue;
1508                                 if (rinser->dtdsc_dirty != NULL)
1509                                         continue;
1511                                 if (rinser->dtdsc_clean != NULL)
1512                                         continue;
```



```
1514                                 rinsep = &rinser->dtdsc_rinsing;
1515                                 break;
1516                         }
1518                         if (j == NCPU) {
1519                                 /*
1520                                  * We were unable to find another CPU that
1521                                  * could accept this dirty list -- we are
1522                                  * therefore unable to clean it now.
1523                                  */
1524                                 dtrace_dynvar_failclean++;
1525                                 continue;
1526                         }
1527                 }
1529                 work = 1;
1531                 /*
1532                  * Atomically move the dirty list aside.
1533                  */
1534                 do {
1535                         dirty = dcpu->dtdsc_dirty;
1537                         /*
1538                          * Before we zap the dirty list, set the rinsing list.
1539                          * (This allows for a potential assertion in
1540                          * dtrace_dynvar():  if a free dynamic variable appears
1541                          * on a hash chain, either the dirty list or the
1542                          * rinsing list for some CPU must be non-NULL.)
1543                          */
1544                         *rinsep = dirty;
1545                         dtrace_membar_producer();
1546                 } while (dtrace_casptr(&dcpu->dtdsc_dirty,
1547                     dirty, NULL) != dirty);
1548         }
1550         if (!work) {
1551                 /*
1552                  * We have no work to do; we can simply return.
1553                  */
1554                 return;
1555         }
1557         dtrace_sync();
1559         for (i = 0; i < NCPU; i++) {
1560                 dcpu = &dstate->dtds_percpu[i];
1562                 if (dcpu->dtdsc_rinsing == NULL)
1563                         continue;
1565                 /*
1566                  * We are now guaranteed that no hash chain contains a pointer
1567                  * into this dirty list; we can make it clean.
1568                  */
1569                 ASSERT(dcpu->dtdsc_clean == NULL);
1570                 dcpu->dtdsc_clean = dcpu->dtdsc_rinsing;
1571                 dcpu->dtdsc_rinsing = NULL;
1572         }
1574         /*
1575          * Before we actually set the state to be DTRACE_DSTATE_CLEAN, make
1576          * sure that all CPUs have seen all of the dtdsc_clean pointers.
1577          * This prevents a race whereby a CPU incorrectly decides that
1578          * the state should be something other than DTRACE_DSTATE_CLEAN
```

```
1579          * after dtrace_dynvar_clean() has completed.
1580          */
1581         dtrace_sync();

1583         dstate->dtds_state = DTRACE_DSTATE_CLEAN;
1584 }

1586 /*
1587  * Depending on the value of the op parameter, this function looks-up,
1588  * allocates or deallocates an arbitrarily-keyed dynamic variable.  If an
1589  * allocation is requested, this function will return a pointer to a
1590  * dtrace_dynvar_t corresponding to the allocated variable -- or NULL if no
1591  * variable can be allocated.  If NULL is returned, the appropriate counter
1592  * will be incremented.
1593  */
1594 dtrace_dynvar_t *
1595 dtrace_dynvar(dtrace_dstate_t *dstate, uint_t nkeys,
1596     dtrace_key_t *key, size_t dsize, dtrace_dynvar_op_t op,
1597     dtrace_mstate_t *mstate, dtrace_vstate_t *vstate)
1598 {
1599         uint64_t hashval = DTRACE_DYNHASH_VALID;
1600         dtrace_dynhash_t *hash = dstate->dtds_hash;
1601         dtrace_dynvar_t *free, *new_free, *next, *dvar, *start, *prev = NULL;
1602         processorid_t me = CPU->cpu_id, cpu = me;
1603         dtrace_dstate_percpu_t *dcpu = &dstate->dtds_percpu[me];
1604         size_t bucket, ksize;
1605         size_t chunksize = dstate->dtds_chunksize;
1606         uintptr_t kdata, lock, nstate;
1607         uint_t i;

1609         ASSERT(nkeys != 0);

1611         /*
1612          * Hash the key.  As with aggregations, we use Jenkins' "One-at-a-time"
1613          * algorithm.  For the by-value portions, we perform the algorithm in
1614          * 16-bit chunks (as opposed to 8-bit chunks).  This speeds things up a
1615          * bit, and seems to have only a minute effect on distribution.  For
1616          * the by-reference data, we perform "One-at-a-time" iterating (safely)
1617          * over each referenced byte.  It's painful to do this, but it's much
1618          * better than pathological hash distribution.  The efficacy of the
1619          * hashing algorithm (and a comparison with other algorithms) may be
1620          * found by running the ::dtrace_dynstat MDB dcmd.
1621          */
1622         for (i = 0; i < nkeys; i++) {
1623                 if (key[i].dttk_size == 0) {
1624                         uint64_t val = key[i].dttk_value;

1626                         hashval += (val >> 48) & 0xffff;
1627                         hashval += (hashval << 10);
1628                         hashval ^= (hashval >> 6);

1630                         hashval += (val >> 32) & 0xffff;
1631                         hashval += (hashval << 10);
1632                         hashval ^= (hashval >> 6);

1634                         hashval += (val >> 16) & 0xffff;
1635                         hashval += (hashval << 10);
1636                         hashval ^= (hashval >> 6);

1638                         hashval += val & 0xffff;
1639                         hashval += (hashval << 10);
1640                         hashval ^= (hashval >> 6);
1641                 } else {
1642                         /*
1643                          * This is incredibly painful, but it beats the hell
1644                          * out of the alternative.
```



```
1645                          */
1646                         uint64_t j, size = key[i].dttk_size;
1647                         uintptr_t base = (uintptr_t)key[i].dttk_value;

1649                         if (!dtrace_canload(base, size, mstate, vstate))
1650                                 break;

1652                         for (j = 0; j < size; j++) {
1653                                 hashval += dtrace_load8(base + j);
1654                                 hashval += (hashval << 10);
1655                                 hashval ^= (hashval >> 6);
1656                         }
1657                 }
1658         }

1660         if (DTRACE_CPUFLAG_ISSET(CPU_DTRACE_FAULT))
1661                 return (NULL);

1663         hashval += (hashval << 3);
1664         hashval ^= (hashval >> 11);
1665         hashval += (hashval << 15);

1667         /*
1668          * There is a remote chance (ideally, 1 in 2^31) that our hashval
1669          * comes out to be one of our two sentinel hash values.  If this
1670          * actually happens, we set the hashval to be a value known to be a
1671          * non-sentinel value.
1672          */
1673         if (hashval == DTRACE_DYNHASH_FREE || hashval == DTRACE_DYNHASH_SINK)
1674                 hashval = DTRACE_DYNHASH_VALID;

1676         /*
1677          * Yes, it's painful to do a divide here.  If the cycle count becomes
1678          * important here, tricks can be pulled to reduce it.  (However, it's
1679          * critical that hash collisions be kept to an absolute minimum;
1680          * they're much more painful than a divide.)  It's better to have a
1681          * solution that generates few collisions and still keeps things
1682          * relatively simple.
1683          */
1684         bucket = hashval % dstate->dtds_hashsize;

1686         if (op == DTRACE_DYNVAR_DEALLOC) {
1687                 volatile uintptr_t *lockp = &hash[bucket].dtdh_lock;

1689                 for (;;) {
1690                         while ((lock = *lockp) & 1)
1691                                 continue;

1693                         if (dtrace_casptr((void *)lockp,
1694                             (void *)lock, (void *)(lock + 1)) == (void *)lock)
1695                                 break;
1696                 }

1698                 dtrace_membar_producer();
1699         }

1701 top:
1702         prev = NULL;
1703         lock = hash[bucket].dtdh_lock;

1705         dtrace_membar_consumer();

1707         start = hash[bucket].dtdh_chain;
1708         ASSERT(start != NULL && (start->dtdv_hashval == DTRACE_DYNHASH_SINK ||
1709             start->dtdv_hashval != DTRACE_DYNHASH_FREE ||
1710             op != DTRACE_DYNVAR_DEALLOC));
```

```
1712         for (dvar = start; dvar != NULL; dvar = dvar->dtdv_next) {
1713                 dtrace_tuple_t *dtuple = &dvar->dtdv_tuple;
1714                 dtrace_key_t *dkey = &dtuple->dtt_key[0];
1715
1716                 if (dvar->dtdv_hashval != hashval) {
1717                         if (dvar->dtdv_hashval == DTRACE_DYNHASH_SINK) {
1718                                 /*
1719                                  * We've reached the sink, and therefore the
1720                                  * end of the hash chain; we can kick out of
1721                                  * the loop knowing that we have seen a valid
1722                                  * snapshot of state.
1723                                  */
1724                                 ASSERT(dvar->dtdv_next == NULL);
1725                                 ASSERT(dvar == &dtrace_dynhash_sink);
1726                                 break;
1727                         }
1728
1729                         if (dvar->dtdv_hashval == DTRACE_DYNHASH_FREE) {
1730                                 /*
1731                                  * We've gone off the rails:  somewhere along
1732                                  * the line, one of the members of this hash
1733                                  * chain was deleted.  Note that we could also
1734                                  * detect this by simply letting this loop run
1735                                  * to completion, as we would eventually hit
1736                                  * the end of the dirty list.  However, we
1737                                  * want to avoid running the length of the
1738                                  * dirty list unnecessarily (it might be quite
1739                                  * long), so we catch this as early as
1740                                  * possible by detecting the hash marker.  In
1741                                  * this case, we simply set dvar to NULL and
1742                                  * break; the conditional after the loop will
1743                                  * send us back to top.
1744                                  */
1745                                 dvar = NULL;
1746                                 break;
1747                         }
1748
1749                         goto next;
1750                 }
1751
1752                 if (dtuple->dtt_nkeys != nkeys)
1753                         goto next;
1754
1755                 for (i = 0; i < nkeys; i++, dkey++) {
1756                         if (dkey->dttk_size != key[i].dttk_size)
1757                                 goto next; /* size or type mismatch */
1758
1759                         if (dkey->dttk_size != 0) {
1760                                 if (dtrace_bcmp(
1761                                     (void *)(uintptr_t)key[i].dttk_value,
1762                                     (void *)(uintptr_t)dkey->dttk_value,
1763                                     dkey->dttk_size))
1764                                         goto next;
1765                         } else {
1766                                 if (dkey->dttk_value != key[i].dttk_value)
1767                                         goto next;
1768                         }
1769                 }
1770
1771                 if (op != DTRACE_DYNVAR_DEALLOC)
1772                         return (dvar);
1773
1774                 ASSERT(dvar->dtdv_next == NULL ||
1775                     dvar->dtdv_next->dtdv_hashval != DTRACE_DYNHASH_FREE);
```



```
1777                 if (prev != NULL) {
1778                         ASSERT(hash[bucket].dtdh_chain != dvar);
1779                         ASSERT(start != dvar);
1780                         ASSERT(prev->dtdv_next == dvar);
1781                         prev->dtdv_next = dvar->dtdv_next;
1782                 } else {
1783                         if (dtrace_casptr(&hash[bucket].dtdh_chain,
1784                             start, dvar->dtdv_next) != start) {
1785                                 /*
1786                                  * We have failed to atomically swing the
1787                                  * hash table head pointer, presumably because
1788                                  * of a conflicting allocation on another CPU.
1789                                  * We need to reread the hash chain and try
1790                                  * again.
1791                                  */
1792                                 goto top;
1793                         }
1794                 }
1795
1796                 dtrace_membar_producer();
1797
1798                 /*
1799                  * Now set the hash value to indicate that it's free.
1800                  */
1801                 ASSERT(hash[bucket].dtdh_chain != dvar);
1802                 dvar->dtdv_hashval = DTRACE_DYNHASH_FREE;
1803
1804                 dtrace_membar_producer();
1805
1806                 /*
1807                  * Set the next pointer to point at the dirty list, and
1808                  * atomically swing the dirty pointer to the newly freed dvar.
1809                  */
1810                 do {
1811                         next = dcpu->dtdsc_dirty;
1812                         dvar->dtdv_next = next;
1813                 } while (dtrace_casptr(&dcpu->dtdsc_dirty, next, dvar) != next);
1814
1815                 /*
1816                  * Finally, unlock this hash bucket.
1817                  */
1818                 ASSERT(hash[bucket].dtdh_lock == lock);
1819                 ASSERT(lock & 1);
1820                 hash[bucket].dtdh_lock++;
1821
1822                 return (NULL);
1823 next:
1824                 prev = dvar;
1825                 continue;
1826         }
1827
1828         if (dvar == NULL) {
1829                 /*
1830                  * If dvar is NULL, it is because we went off the rails:
1831                  * one of the elements that we traversed in the hash chain
1832                  * was deleted while we were traversing it.  In this case,
1833                  * we assert that we aren't doing a dealloc (deallocs lock
1834                  * the hash bucket to prevent themselves from racing with
1835                  * one another), and retry the hash chain traversal.
1836                  */
1837                 ASSERT(op != DTRACE_DYNVAR_DEALLOC);
1838                 goto top;
1839         }
1840
1841         if (op != DTRACE_DYNVAR_ALLOC) {
1842                 /*
```

```
1843                  * If we are not to allocate a new variable, we want to
1844                  * return NULL now.  Before we return, check that the value
1845                  * of the lock word hasn't changed.  If it has, we may have
1846                  * seen an inconsistent snapshot.
1847                  */
1848                 if (op == DTRACE_DYNVAR_NOALLOC) {
1849                         if (hash[bucket].dtdh_lock != lock)
1850                                 goto top;
1851                 } else {
1852                         ASSERT(op == DTRACE_DYNVAR_DEALLOC);
1853                         ASSERT(hash[bucket].dtdh_lock == lock);
1854                         ASSERT(lock & 1);
1855                         hash[bucket].dtdh_lock++;
1856                 }
1857
1858                 return (NULL);
1859         }
1860
1861         /*
1862          * We need to allocate a new dynamic variable.  The size we need is the
1863          * size of dtrace_dynvar plus the size of nkeys dtrace_key_t's plus the
1864          * size of any auxiliary key data (rounded up to 8-byte alignment) plus
1865          * the size of any referred-to data (dsize).  We then round the final
1866          * size up to the chunksize for allocation.
1867          */
1868         for (ksize = 0, i = 0; i < nkeys; i++)
1869                 ksize += P2ROUNDUP(key[i].dttk_size, sizeof (uint64_t));
1870
1871         /*
1872          * This should be pretty much impossible, but could happen if, say,
1873          * strange DIF specified the tuple.  Ideally, this should be an
1874          * assertion and not an error condition -- but that requires that the
1875          * chunksize calculation in dtrace_difo_chunksize() be absolutely
1876          * bullet-proof.  (That is, it must not be able to be fooled by
1877          * malicious DIF.)  Given the lack of backwards branches in DIF,
1878          * solving this would presumably not amount to solving the Halting
1879          * Problem -- but it still seems awfully hard.
1880          */
1881         if (sizeof (dtrace_dynvar_t) + sizeof (dtrace_key_t) * (nkeys - 1) +
1882             ksize + dsize > chunksize) {
1883                 dcpu->dtdsc_drops++;
1884                 return (NULL);
1885         }
1886
1887         nstate = DTRACE_DSTATE_EMPTY;
1888
1889         do {
1890 retry:
1891                 free = dcpu->dtdsc_free;
1892
1893                 if (free == NULL) {
1894                         dtrace_dynvar_t *clean = dcpu->dtdsc_clean;
1895                         void *rval;
1896
1897                         if (clean == NULL) {
1898                                 /*
1899                                  * We're out of dynamic variable space on
1900                                  * this CPU.  Unless we have tried all CPUs,
1901                                  * we'll try to allocate from a different
1902                                  * CPU.
1903                                  */
1904                                 switch (dstate->dtds_state) {
1905                                 case DTRACE_DSTATE_CLEAN: {
1906                                         void *sp = &dstate->dtds_state;
1907
1908                                         if (++cpu >= NCPU)
```



```
1909                                                 cpu = 0;
1910
1911                                         if (dcpu->dtdsc_dirty != NULL &&
1912                                             nstate == DTRACE_DSTATE_EMPTY)
1913                                                 nstate = DTRACE_DSTATE_DIRTY;
1914
1915                                         if (dcpu->dtdsc_rinsing != NULL)
1916                                                 nstate = DTRACE_DSTATE_RINSING;
1917
1918                                         dcpu = &dstate->dtds_percpu[cpu];
1919
1920                                         if (cpu != me)
1921                                                 goto retry;
1922
1923                                         (void) dtrace_cas32(sp,
1924                                             DTRACE_DSTATE_CLEAN, nstate);
1925
1926                                         /*
1927                                          * To increment the correct bean
1928                                          * counter, take another lap.
1929                                          */
1930                                         goto retry;
1931                                 }
1932
1933                                 case DTRACE_DSTATE_DIRTY:
1934                                         dcpu->dtdsc_dirty_drops++;
1935                                         break;
1936
1937                                 case DTRACE_DSTATE_RINSING:
1938                                         dcpu->dtdsc_rinsing_drops++;
1939                                         break;
1940
1941                                 case DTRACE_DSTATE_EMPTY:
1942                                         dcpu->dtdsc_drops++;
1943                                         break;
1944                                 }
1945
1946                                 DTRACE_CPUFLAG_SET(CPU_DTRACE_DROP);
1947                                 return (NULL);
1948                         }
1949
1950                         /*
1951                          * The clean list appears to be non-empty.  We want to
1952                          * move the clean list to the free list; we start by
1953                          * moving the clean pointer aside.
1954                          */
1955                         if (dtrace_casptr(&dcpu->dtdsc_clean,
1956                             clean, NULL) != clean) {
1957                                 /*
1958                                  * We are in one of two situations:
1959                                  *
1960                                  *  (a) The clean list was switched to the
1961                                  *      free list by another CPU.
1962                                  *
1963                                  *  (b) The clean list was added to by the
1964                                  *      cleansing cyclic.
1965                                  *
1966                                  * In either of these situations, we can
1967                                  * just reattempt the free list allocation.
1968                                  */
1969                                 goto retry;
1970                         }
1971
1972                         ASSERT(clean->dtdv_hashval == DTRACE_DYNHASH_FREE);
1973
1974                         /*
```

```
1975                         * Now we'll move the clean list to our free list.
1976                         * It's impossible for this to fail:  the only way
1977                         * the free list can be updated is through this
1978                         * code path, and only one CPU can own the clean list.
1979                         * Thus, it would only be possible for this to fail if
1980                         * this code were racing with dtrace_dynvar_clean().
1981                         * (That is, if dtrace_dynvar_clean() updated the clean
1982                         * list, and we ended up racing to update the free
1983                         * list.)  This race is prevented by the dtrace_sync()
1984                         * in dtrace_dynvar_clean() -- which flushes the
1985                         * owners of the clean lists out before resetting
1986                         * the clean lists.
1987                         */
1988                        dcpu = &dstate->dtds_percpu[me];
1989                        rval = dtrace_casptr(&dcpu->dtdsc_free, NULL, clean);
1990                        ASSERT(rval == NULL);
1991                        goto retry;
1992                }
1994                dvar = free;
1995                new_free = dvar->dtdv_next;
1996        } while (dtrace_casptr(&dcpu->dtdsc_free, free, new_free) != free);
1998        /*
1999         * We have now allocated a new chunk.  We copy the tuple keys into the
2000         * tuple array and copy any referenced key data into the data space
2001         * following the tuple array.  As we do this, we relocate dttk_value
2002         * in the final tuple to point to the key data address in the chunk.
2003         */
2004        kdata = (uintptr_t)&dvar->dtdv_tuple.dtt_key[nkeys];
2005        dvar->dtdv_data = (void *)(kdata + ksize);
2006        dvar->dtdv_tuple.dtt_nkeys = nkeys;
2008        for (i = 0; i < nkeys; i++) {
2009                dtrace_key_t *dkey = &dvar->dtdv_tuple.dtt_key[i];
2010                size_t kesize = key[i].dttk_size;
2012                if (kesize != 0) {
2013                        dtrace_bcopy(
2014                            (const void *)(uintptr_t)key[i].dttk_value,
2015                            (void *)kdata, kesize);
2016                        dkey->dttk_value = kdata;
2017                        kdata += P2ROUNDUP(kesize, sizeof (uint64_t));
2018                } else {
2019                        dkey->dttk_value = key[i].dttk_value;
2020                }
2022                dkey->dttk_size = kesize;
2023        }
2025        ASSERT(dvar->dtdv_hashval == DTRACE_DYNHASH_FREE);
2026        dvar->dtdv_hashval = hashval;
2027        dvar->dtdv_next = start;
2029        if (dtrace_casptr(&hash[bucket].dtdh_chain, start, dvar) == start)
2030                return (dvar);
2032        /*
2033         * The cas has failed.  Either another CPU is adding an element to
2034         * this hash chain, or another CPU is deleting an element from this
2035         * hash chain.  The simplest way to deal with both of these cases
2036         * (though not necessarily the most efficient) is to free our
2037         * allocated block and tail-call ourselves.  Note that the free is
2038         * to the dirty list and _not_ to the free list.  This is to prevent
2039         * races with allocators, above.
2040         */
```



```
2041        dvar->dtdv_hashval = DTRACE_DYNHASH_FREE;
2043        dtrace_membar_producer();
2045        do {
2046                free = dcpu->dtdsc_dirty;
2047                dvar->dtdv_next = free;
2048        } while (dtrace_casptr(&dcpu->dtdsc_dirty, free, dvar) != free);
2050        return (dtrace_dynvar(dstate, nkeys, key, dsize, op, mstate, vstate));
2051 }
2053 /*ARGSUSED*/
2054 static void
2055 dtrace_aggregate_min(uint64_t *oval, uint64_t nval, uint64_t arg)
2056 {
2057        if ((int64_t)nval < (int64_t)*oval)
2058                *oval = nval;
2059 }
2061 /*ARGSUSED*/
2062 static void
2063 dtrace_aggregate_max(uint64_t *oval, uint64_t nval, uint64_t arg)
2064 {
2065        if ((int64_t)nval > (int64_t)*oval)
2066                *oval = nval;
2067 }
2069 static void
2070 dtrace_aggregate_quantize(uint64_t *quanta, uint64_t nval, uint64_t incr)
2071 {
2072        int i, zero = DTRACE_QUANTIZE_ZEROBUCKET;
2073        int64_t val = (int64_t)nval;
2075        if (val < 0) {
2076                for (i = 0; i < zero; i++) {
2077                        if (val <= DTRACE_QUANTIZE_BUCKETVAL(i)) {
2078                                quanta[i] += incr;
2079                                return;
2080                        }
2081                }
2082        } else {
2083                for (i = zero + 1; i < DTRACE_QUANTIZE_NBUCKETS; i++) {
2084                        if (val < DTRACE_QUANTIZE_BUCKETVAL(i)) {
2085                                quanta[i - 1] += incr;
2086                                return;
2087                        }
2088                }
2090                quanta[DTRACE_QUANTIZE_NBUCKETS - 1] += incr;
2091                return;
2092        }
2094        ASSERT(0);
2095 }
2097 static void
2098 dtrace_aggregate_lquantize(uint64_t *lquanta, uint64_t nval, uint64_t incr)
2099 {
2100        uint64_t arg = *lquanta++;
2101        int32_t base = DTRACE_LQUANTIZE_BASE(arg);
2102        uint16_t step = DTRACE_LQUANTIZE_STEP(arg);
2103        uint16_t levels = DTRACE_LQUANTIZE_LEVELS(arg);
2104        int32_t val = (int32_t)nval, level;
2106        ASSERT(step != 0);
```

```
2107         ASSERT(levels != 0);
2108
2109         if (val < base) {
2110                 /*
2111                  * This is an underflow.
2112                  */
2113                 lquanta[0] += incr;
2114                 return;
2115         }
2116
2117         level = (val - base) / step;
2118
2119         if (level < levels) {
2120                 lquanta[level + 1] += incr;
2121                 return;
2122         }
2123
2124         /*
2125          * This is an overflow.
2126          */
2127         lquanta[levels + 1] += incr;
2128 }
2129
2130 static int
2131 dtrace_aggregate_llquantize_bucket(uint16_t factor, uint16_t low,
2132     uint16_t high, uint16_t nsteps, int64_t value)
2133 {
2134         int64_t this = 1, last, next;
2135         int base = 1, order;
2136
2137         ASSERT(factor <= nsteps);
2138         ASSERT(nsteps % factor == 0);
2139
2140         for (order = 0; order < low; order++)
2141                 this *= factor;
2142
2143         /*
2144          * If our value is less than our factor taken to the power of the
2145          * low order of magnitude, it goes into the zeroth bucket.
2146          */
2147         if (value < (last = this))
2148                 return (0);
2149
2150         for (this *= factor; order <= high; order++) {
2151                 int nbuckets = this > nsteps ? nsteps : this;
2152
2153                 if ((next = this * factor) < this) {
2154                         /*
2155                          * We should not generally get log/linear quantizations
2156                          * with a high magnitude that allows 64-bits to
2157                          * overflow, but we nonetheless protect against this
2158                          * by explicitly checking for overflow, and clamping
2159                          * our value accordingly.
2160                          */
2161                         value = this - 1;
2162                 }
2163
2164                 if (value < this) {
2165                         /*
2166                          * If our value lies within this order of magnitude,
2167                          * determine its position by taking the offset within
2168                          * the order of magnitude, dividing by the bucket
2169                          * width, and adding to our (accumulated) base.
2170                          */
2171                         return (base + (value - last) / (this / nbuckets));
2172                 }
```



```
2173
2174                 base += nbuckets - (nbuckets / factor);
2175                 last = this;
2176                 this = next;
2177         }
2178
2179         /*
2180          * Our value is greater than or equal to our factor taken to the
2181          * power of one plus the high magnitude -- return the top bucket.
2182          */
2183         return (base);
2184 }
2185
2186 static void
2187 dtrace_aggregate_llquantize(uint64_t *llquanta, uint64_t nval, uint64_t incr)
2188 {
2189         uint64_t arg = *llquanta++;
2190         uint16_t factor = DTRACE_LLQUANTIZE_FACTOR(arg);
2191         uint16_t low = DTRACE_LLQUANTIZE_LOW(arg);
2192         uint16_t high = DTRACE_LLQUANTIZE_HIGH(arg);
2193         uint16_t nsteps = DTRACE_LLQUANTIZE_NSTEP(arg);
2194
2195         llquanta[dtrace_aggregate_llquantize_bucket(factor,
2196             low, high, nsteps, nval)] += incr;
2197 }
2198
2199 /*ARGSUSED*/
2200 static void
2201 dtrace_aggregate_avg(uint64_t *data, uint64_t nval, uint64_t arg)
2202 {
2203         data[0]++;
2204         data[1] += nval;
2205 }
2206
2207 /*ARGSUSED*/
2208 static void
2209 dtrace_aggregate_stddev(uint64_t *data, uint64_t nval, uint64_t arg)
2210 {
2211         int64_t snval = (int64_t)nval;
2212         uint64_t tmp[2];
2213
2214         data[0]++;
2215         data[1] += nval;
2216
2217         /*
2218          * What we want to say here is:
2219          *
2220          * data[2] += nval * nval;
2221          *
2222          * But given that nval is 64-bit, we could easily overflow, so
2223          * we do this as 128-bit arithmetic.
2224          */
2225         if (snval < 0)
2226                 snval = -snval;
2227
2228         dtrace_multiply_128((uint64_t)snval, (uint64_t)snval, tmp);
2229         dtrace_add_128(data + 2, tmp, data + 2);
2230 }
2231
2232 /*ARGSUSED*/
2233 static void
2234 dtrace_aggregate_count(uint64_t *oval, uint64_t nval, uint64_t arg)
2235 {
2236         *oval = *oval + 1;
2237 }
```

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

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

        if (buf == NULL)
                return;

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

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

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

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

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

        if (buf->dtb_offset == 0) {
                /*
                 * We just kludge up approximately 1/8th of the size to be
                 * buckets.  If this guess ends up being routinely
```



```
                 * off-the-mark, we may need to dynamically readjust this
                 * based on past performance.
                 */
                uintptr_t hashsize = (buf->dtb_size >> 3) / sizeof (uintptr_t);

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

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

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

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

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

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

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

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

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

        /*
         * Yes, the divide here is expensive -- but it's generally the least
         * of the performance issues given the amount of data that we iterate
```

```
2371          * over to compute hash values, compare data, etc.
2372          */
2373         ndx = hashval % agb->dtagb_hashsize;

2375         for (key = agb->dtagb_hash[ndx]; key != NULL; key = key->dtak_next) {
2376                 ASSERT((caddr_t)key >= tomax);
2377                 ASSERT((caddr_t)key < tomax + buf->dtb_size);

2379                 if (hashval != key->dtak_hashval || key->dtak_size != size)
2380                         continue;

2382                 kdata = key->dtak_data;
2383                 ASSERT(kdata >= tomax && kdata < tomax + buf->dtb_size);

2385                 for (act = agg->dtag_first; act->dta_intuple;
2386                     act = act->dta_next) {
2387                         i = act->dta_rec.dtrd_offset - agg->dtag_base;
2388                         limit = i + act->dta_rec.dtrd_size;
2389                         ASSERT(limit <= size);
2390                         isstr = DTRACEACT_ISSTRING(act);

2392                         for (; i < limit; i++) {
2393                                 if (kdata[i] != data[i])
2394                                         goto next;

2396                                 if (isstr && data[i] == '\0')
2397                                         break;
2398                         }
2399                 }

2401                 if (action != key->dtak_action) {
2402                         /*
2403                          * We are aggregating on the same value in the same
2404                          * aggregation with two different aggregating actions.
2405                          * (This should have been picked up in the compiler,
2406                          * so we may be dealing with errant or devious DIF.)
2407                          * This is an error condition; we indicate as much,
2408                          * and return.
2409                          */
2410                         DTRACE_CPUFLAG_SET(CPU_DTRACE_ILLOP);
2411                         return;
2412                 }

2414                 /*
2415                  * This is a hit:  we need to apply the aggregator to
2416                  * the value at this key.
2417                  */
2418                 agg->dtag_aggregate((uint64_t *)(kdata + size), expr, arg);
2419                 return;
2420 next:
2421                 continue;
2422         }

2424         /*
2425          * We didn't find it.  We need to allocate some zero-filled space,
2426          * link it into the hash table appropriately, and apply the aggregator
2427          * to the (zero-filled) value.
2428          */
2429         offs = buf->dtb_offset;
2430         while (offs & (align - 1))
2431                 offs += sizeof (uint32_t);

2433         /*
2434          * If we don't have enough room to both allocate a new key _and_
2435          * its associated data, increment the drop count and return.
2436          */
```



```
2437         if ((uintptr_t)tomax + offs + fsize >
2438             agb->dtagb_free - sizeof (dtrace_aggkey_t)) {
2439                 dtrace_buffer_drop(buf);
2440                 return;
2441         }

2443         /*CONSTCOND*/
2444         ASSERT(!(sizeof (dtrace_aggkey_t) & (sizeof (uintptr_t) - 1)));
2445         key = (dtrace_aggkey_t *)(agb->dtagb_free - sizeof (dtrace_aggkey_t));
2446         agb->dtagb_free -= sizeof (dtrace_aggkey_t);

2448         key->dtak_data = kdata = tomax + offs;
2449         buf->dtb_offset = offs + fsize;

2451         /*
2452          * Now copy the data across.
2453          */
2454         *((dtrace_aggid_t *)kdata) = agg->dtag_id;

2456         for (i = sizeof (dtrace_aggid_t); i < size; i++)
2457                 kdata[i] = data[i];

2459         /*
2460          * Because strings are not zeroed out by default, we need to iterate
2461          * looking for actions that store strings, and we need to explicitly
2462          * pad these strings out with zeroes.
2463          */
2464         for (act = agg->dtag_first; act->dta_intuple; act = act->dta_next) {
2465                 int nul;

2467                 if (!DTRACEACT_ISSTRING(act))
2468                         continue;

2470                 i = act->dta_rec.dtrd_offset - agg->dtag_base;
2471                 limit = i + act->dta_rec.dtrd_size;
2472                 ASSERT(limit <= size);

2474                 for (nul = 0; i < limit; i++) {
2475                         if (nul) {
2476                                 kdata[i] = '\0';
2477                                 continue;
2478                         }

2480                         if (data[i] != '\0')
2481                                 continue;

2483                         nul = 1;
2484                 }
2485         }

2487         for (i = size; i < fsize; i++)
2488                 kdata[i] = 0;

2490         key->dtak_hashval = hashval;
2491         key->dtak_size = size;
2492         key->dtak_action = action;
2493         key->dtak_next = agb->dtagb_hash[ndx];
2494         agb->dtagb_hash[ndx] = key;

2496         /*
2497          * Finally, apply the aggregator.
2498          */
2499         *((uint64_t *)(key->dtak_data + size)) = agg->dtag_initial;
2500         agg->dtag_aggregate((uint64_t *)(key->dtak_data + size), expr, arg);
2501 }
```

```
2503 /*
2504  * Given consumer state, this routine finds a speculation in the INACTIVE
2505  * state and transitions it into the ACTIVE state.  If there is no speculation
2506  * in the INACTIVE state, 0 is returned.  In this case, no error counter is
2507  * incremented -- it is up to the caller to take appropriate action.
2508  */
2509 static int
2510 dtrace_speculation(dtrace_state_t *state)
2511 {
2512         int i = 0;
2513         dtrace_speculation_state_t current;
2514         uint32_t *stat = &state->dts_speculations_unavail, count;

2516         while (i < state->dts_nspeculations) {
2517                 dtrace_speculation_t *spec = &state->dts_speculations[i];

2519                 current = spec->dtsp_state;

2521                 if (current != DTRACESPEC_INACTIVE) {
2522                         if (current == DTRACESPEC_COMMITTINGMANY ||
2523                             current == DTRACESPEC_COMMITTING ||
2524                             current == DTRACESPEC_DISCARDING)
2525                                 stat = &state->dts_speculations_busy;
2526                         i++;
2527                         continue;
2528                 }

2530                 if (dtrace_cas32((uint32_t *)&spec->dtsp_state,
2531                     current, DTRACESPEC_ACTIVE) == current)
2532                         return (i + 1);
2533         }

2535         /*
2536          * We couldn't find a speculation.  If we found as much as a single
2537          * busy speculation buffer, we'll attribute this failure as "busy"
2538          * instead of "unavail".
2539          */
2540         do {
2541                 count = *stat;
2542         } while (dtrace_cas32(stat, count, count + 1) != count);

2544         return (0);
2545 }

2547 /*
2548  * This routine commits an active speculation.  If the specified speculation
2549  * is not in a valid state to perform a commit(), this routine will silently do
2550  * nothing.  The state of the specified speculation is transitioned according
2551  * to the state transition diagram outlined in <sys/dtrace_impl.h>
2552  */
2553 static void
2554 dtrace_speculation_commit(dtrace_state_t *state, processorid_t cpu,
2555     dtrace_specid_t which)
2556 {
2557         dtrace_speculation_t *spec;
2558         dtrace_buffer_t *src, *dest;
2559         uintptr_t daddr, saddr, dlimit, slimit;
2560         dtrace_speculation_state_t current, new;
2561         intptr_t offs;
2562         uint64_t timestamp;

2564         if (which == 0)
2565                 return;

2567         if (which > state->dts_nspeculations) {
2568                 cpu_core[cpu].cpuc_dtrace_flags |= CPU_DTRACE_ILLOP;
```



```
2569                 return;
2570         }

2572         spec = &state->dts_speculations[which - 1];
2573         src = &spec->dtsp_buffer[cpu];
2574         dest = &state->dts_buffer[cpu];

2576         do {
2577                 current = spec->dtsp_state;

2579                 if (current == DTRACESPEC_COMMITTINGMANY)
2580                         break;

2582                 switch (current) {
2583                 case DTRACESPEC_INACTIVE:
2584                 case DTRACESPEC_DISCARDING:
2585                         return;

2587                 case DTRACESPEC_COMMITTING:
2588                         /*
2589                          * This is only possible if we are (a) commit()'ing
2590                          * without having done a prior speculate() on this CPU
2591                          * and (b) racing with another commit() on a different
2592                          * CPU.  There's nothing to do -- we just assert that
2593                          * our offset is 0.
2594                          */
2595                         ASSERT(src->dtb_offset == 0);
2596                         return;

2598                 case DTRACESPEC_ACTIVE:
2599                         new = DTRACESPEC_COMMITTING;
2600                         break;

2602                 case DTRACESPEC_ACTIVEONE:
2603                         /*
2604                          * This speculation is active on one CPU.  If our
2605                          * buffer offset is non-zero, we know that the one CPU
2606                          * must be us.  Otherwise, we are committing on a
2607                          * different CPU from the speculate(), and we must
2608                          * rely on being asynchronously cleaned.
2609                          */
2610                         if (src->dtb_offset != 0) {
2611                                 new = DTRACESPEC_COMMITTING;
2612                                 break;
2613                         }
2614                         /*FALLTHROUGH*/

2616                 case DTRACESPEC_ACTIVEMANY:
2617                         new = DTRACESPEC_COMMITTINGMANY;
2618                         break;

2620                 default:
2621                         ASSERT(0);
2622                 }
2623         } while (dtrace_cas32((uint32_t *)&spec->dtsp_state,
2624             current, new) != current);

2626         /*
2627          * We have set the state to indicate that we are committing this
2628          * speculation.  Now reserve the necessary space in the destination
2629          * buffer.
2630          */
2631         if ((offs = dtrace_buffer_reserve(dest, src->dtb_offset,
2632             sizeof (uint64_t), state, NULL)) < 0) {
2633                 dtrace_buffer_drop(dest);
2634                 goto out;
```

```
2635         }
2636
2637         /*
2638          * We have sufficient space to copy the speculative buffer into the
2639          * primary buffer.  First, modify the speculative buffer, filling
2640          * in the timestamp of all entries with the current time.  The data
2641          * must have the commit() time rather than the time it was traced,
2642          * so that all entries in the primary buffer are in timestamp order.
2643          */
2644         timestamp = dtrace_gethrtime();
2645         saddr = (uintptr_t)src->dtb_tomax;
2646         slimit = saddr + src->dtb_offset;
2647         while (saddr < slimit) {
2648                 size_t size;
2649                 dtrace_rechdr_t *dtrh = (dtrace_rechdr_t *)saddr;
2650
2651                 if (dtrh->dtrh_epid == DTRACE_EPIDNONE) {
2652                         saddr += sizeof (dtrace_epid_t);
2653                         continue;
2654                 }
2655                 ASSERT3U(dtrh->dtrh_epid, <=, state->dts_necbs);
2656                 size = state->dts_ecbs[dtrh->dtrh_epid - 1]->dte_size;
2657
2658                 ASSERT3U(saddr + size, <=, slimit);
2659                 ASSERT3U(size, >=, sizeof (dtrace_rechdr_t));
2660                 ASSERT3U(DTRACE_RECORD_LOAD_TIMESTAMP(dtrh), ==, UINT64_MAX);
2661
2662                 DTRACE_RECORD_STORE_TIMESTAMP(dtrh, timestamp);
2663
2664                 saddr += size;
2665         }
2666
2667         /*
2668          * Copy the buffer across.  (Note that this is a
2669          * highly subobtimal bcopy(); in the unlikely event that this becomes
2670          * a serious performance issue, a high-performance DTrace-specific
2671          * bcopy() should obviously be invented.)
2672          */
2673         daddr = (uintptr_t)dest->dtb_tomax + offs;
2674         dlimit = daddr + src->dtb_offset;
2675         saddr = (uintptr_t)src->dtb_tomax;
2676
2677         /*
2678          * First, the aligned portion.
2679          */
2680         while (dlimit - daddr >= sizeof (uint64_t)) {
2681                 *((uint64_t *)daddr) = *((uint64_t *)saddr);
2682
2683                 daddr += sizeof (uint64_t);
2684                 saddr += sizeof (uint64_t);
2685         }
2686
2687         /*
2688          * Now any left-over bit...
2689          */
2690         while (dlimit - daddr)
2691                 *((uint8_t *)daddr++) = *((uint8_t *)saddr++);
2692
2693         /*
2694          * Finally, commit the reserved space in the destination buffer.
2695          */
2696         dest->dtb_offset = offs + src->dtb_offset;
2697
2698 out:
2699         /*
2700          * If we're lucky enough to be the only active CPU on this speculation
```



```
2701          * buffer, we can just set the state back to DTRACESPEC_INACTIVE.
2702          */
2703         if (current == DTRACESPEC_ACTIVE ||
2704             (current == DTRACESPEC_ACTIVEONE && new == DTRACESPEC_COMMITTING)) {
2705                 uint32_t rval = dtrace_cas32((uint32_t *)&spec->dtsp_state,
2706                     DTRACESPEC_COMMITTING, DTRACESPEC_INACTIVE);
2707
2708                 ASSERT(rval == DTRACESPEC_COMMITTING);
2709         }
2710
2711         src->dtb_offset = 0;
2712         src->dtb_xamot_drops += src->dtb_drops;
2713         src->dtb_drops = 0;
2714 }
2715
2716 /*
2717  * This routine discards an active speculation.  If the specified speculation
2718  * is not in a valid state to perform a discard(), this routine will silently
2719  * do nothing.  The state of the specified speculation is transitioned
2720  * according to the state transition diagram outlined in <sys/dtrace_impl.h>
2721  */
2722 static void
2723 dtrace_speculation_discard(dtrace_state_t *state, processorid_t cpu,
2724     dtrace_specid_t which)
2725 {
2726         dtrace_speculation_t *spec;
2727         dtrace_speculation_state_t current, new;
2728         dtrace_buffer_t *buf;
2729
2730         if (which == 0)
2731                 return;
2732
2733         if (which > state->dts_nspeculations) {
2734                 cpu_core[cpu].cpuc_dtrace_flags |= CPU_DTRACE_ILLOP;
2735                 return;
2736         }
2737
2738         spec = &state->dts_speculations[which - 1];
2739         buf = &spec->dtsp_buffer[cpu];
2740
2741         do {
2742                 current = spec->dtsp_state;
2743
2744                 switch (current) {
2745                 case DTRACESPEC_INACTIVE:
2746                 case DTRACESPEC_COMMITTINGMANY:
2747                 case DTRACESPEC_COMMITTING:
2748                 case DTRACESPEC_DISCARDING:
2749                         return;
2750
2751                 case DTRACESPEC_ACTIVE:
2752                 case DTRACESPEC_ACTIVEMANY:
2753                         new = DTRACESPEC_DISCARDING;
2754                         break;
2755
2756                 case DTRACESPEC_ACTIVEONE:
2757                         if (buf->dtb_offset != 0) {
2758                                 new = DTRACESPEC_INACTIVE;
2759                         } else {
2760                                 new = DTRACESPEC_DISCARDING;
2761                         }
2762                         break;
2763
2764                 default:
2765                         ASSERT(0);
2766                 }
```

```
2767         } while (dtrace_cas32((uint32_t *)&spec->dtsp_state,
2768             current, new) != current);

2770         buf->dtb_offset = 0;
2771         buf->dtb_drops = 0;
2772 }

2774 /*
2775  * Note:  not called from probe context.  This function is called
2776  * asynchronously from cross call context to clean any speculations that are
2777  * in the COMMITTINGMANY or DISCARDING states.  These speculations may not be
2778  * transitioned back to the INACTIVE state until all CPUs have cleaned the
2779  * speculation.
2780  */
2781 static void
2782 dtrace_speculation_clean_here(dtrace_state_t *state)
2783 {
2784         dtrace_icookie_t cookie;
2785         processorid_t cpu = CPU->cpu_id;
2786         dtrace_buffer_t *dest = &state->dts_buffer[cpu];
2787         dtrace_specid_t i;

2789         cookie = dtrace_interrupt_disable();

2791         if (dest->dtb_tomax == NULL) {
2792                 dtrace_interrupt_enable(cookie);
2793                 return;
2794         }

2796         for (i = 0; i < state->dts_nspeculations; i++) {
2797                 dtrace_speculation_t *spec = &state->dts_speculations[i];
2798                 dtrace_buffer_t *src = &spec->dtsp_buffer[cpu];

2800                 if (src->dtb_tomax == NULL)
2801                         continue;

2803                 if (spec->dtsp_state == DTRACESPEC_DISCARDING) {
2804                         src->dtb_offset = 0;
2805                         continue;
2806                 }

2808                 if (spec->dtsp_state != DTRACESPEC_COMMITTINGMANY)
2809                         continue;

2811                 if (src->dtb_offset == 0)
2812                         continue;

2814                 dtrace_speculation_commit(state, cpu, i + 1);
2815         }

2817         dtrace_interrupt_enable(cookie);
2818 }

2820 /*
2821  * Note:  not called from probe context.  This function is called
2822  * asynchronously (and at a regular interval) to clean any speculations that
2823  * are in the COMMITTINGMANY or DISCARDING states.  If it discovers that there
2824  * is work to be done, it cross calls all CPUs to perform that work;
2825  * COMMITMANY and DISCARDING speculations may not be transitioned back to the
2826  * INACTIVE state until they have been cleaned by all CPUs.
2827  */
2828 static void
2829 dtrace_speculation_clean(dtrace_state_t *state)
2830 {
2831         int work = 0, rv;
2832         dtrace_specid_t i;
```



```
2834         for (i = 0; i < state->dts_nspeculations; i++) {
2835                 dtrace_speculation_t *spec = &state->dts_speculations[i];

2837                 ASSERT(!spec->dtsp_cleaning);

2839                 if (spec->dtsp_state != DTRACESPEC_DISCARDING &&
2840                     spec->dtsp_state != DTRACESPEC_COMMITTINGMANY)
2841                         continue;

2843                 work++;
2844                 spec->dtsp_cleaning = 1;
2845         }

2847         if (!work)
2848                 return;

2850         dtrace_xcall(DTRACE_CPUALL,
2851             (dtrace_xcall_t)dtrace_speculation_clean_here, state);

2853         /*
2854          * We now know that all CPUs have committed or discarded their
2855          * speculation buffers, as appropriate.  We can now set the state
2856          * to inactive.
2857          */
2858         for (i = 0; i < state->dts_nspeculations; i++) {
2859                 dtrace_speculation_t *spec = &state->dts_speculations[i];
2860                 dtrace_speculation_state_t current, new;

2862                 if (!spec->dtsp_cleaning)
2863                         continue;

2865                 current = spec->dtsp_state;
2866                 ASSERT(current == DTRACESPEC_DISCARDING ||
2867                     current == DTRACESPEC_COMMITTINGMANY);

2869                 new = DTRACESPEC_INACTIVE;

2871                 rv = dtrace_cas32((uint32_t *)&spec->dtsp_state, current, new);
2872                 ASSERT(rv == current);
2873                 spec->dtsp_cleaning = 0;
2874         }
2875 }

2877 /*
2878  * Called as part of a speculate() to get the speculative buffer associated
2879  * with a given speculation.  Returns NULL if the specified speculation is not
2880  * in an ACTIVE state.  If the speculation is in the ACTIVEONE state -- and
2881  * the active CPU is not the specified CPU -- the speculation will be
2882  * atomically transitioned into the ACTIVEMANY state.
2883  */
2884 static dtrace_buffer_t *
2885 dtrace_speculation_buffer(dtrace_state_t *state, processorid_t cpuid,
2886     dtrace_specid_t which)
2887 {
2888         dtrace_speculation_t *spec;
2889         dtrace_speculation_state_t current, new;
2890         dtrace_buffer_t *buf;

2892         if (which == 0)
2893                 return (NULL);

2895         if (which > state->dts_nspeculations) {
2896                 cpu_core[cpuid].cpuc_dtrace_flags |= CPU_DTRACE_ILLOP;
2897                 return (NULL);
2898         }
```

```
2900         spec = &state->dts_speculations[which - 1];
2901         buf = &spec->dtsp_buffer[cpuid];
2902
2903         do {
2904                 current = spec->dtsp_state;
2905
2906                 switch (current) {
2907                 case DTRACESPEC_INACTIVE:
2908                 case DTRACESPEC_COMMITTINGMANY:
2909                 case DTRACESPEC_DISCARDING:
2910                         return (NULL);
2911
2912                 case DTRACESPEC_COMMITTING:
2913                         ASSERT(buf->dtb_offset == 0);
2914                         return (NULL);
2915
2916                 case DTRACESPEC_ACTIVEONE:
2917                         /*
2918                          * This speculation is currently active on one CPU.
2919                          * Check the offset in the buffer; if it's non-zero,
2920                          * that CPU must be us (and we leave the state alone).
2921                          * If it's zero, assume that we're starting on a new
2922                          * CPU -- and change the state to indicate that the
2923                          * speculation is active on more than one CPU.
2924                          */
2925                         if (buf->dtb_offset != 0)
2926                                 return (buf);
2927
2928                         new = DTRACESPEC_ACTIVEMANY;
2929                         break;
2930
2931                 case DTRACESPEC_ACTIVEMANY:
2932                         return (buf);
2933
2934                 case DTRACESPEC_ACTIVE:
2935                         new = DTRACESPEC_ACTIVEONE;
2936                         break;
2937
2938                 default:
2939                         ASSERT(0);
2940                 }
2941         } while (dtrace_cas32((uint32_t *)&spec->dtsp_state,
2942             current, new) != current);
2943
2944         ASSERT(new == DTRACESPEC_ACTIVEONE || new == DTRACESPEC_ACTIVEMANY);
2945         return (buf);
2946 }
2947
2948 /*
2949  * Return a string.  In the event that the user lacks the privilege to access
2950  * arbitrary kernel memory, we copy the string out to scratch memory so that we
2951  * don't fail access checking.
2952  *
2953  * dtrace_dif_variable() uses this routine as a helper for various
2954  * builtin values such as 'execname' and 'probefunc.'
2955  */
2956 uintptr_t
2957 dtrace_dif_varstr(uintptr_t addr, dtrace_state_t *state,
2958     dtrace_mstate_t *mstate)
2959 {
2960         uint64_t size = state->dts_options[DTRACEOPT_STRSIZE];
2961         uintptr_t ret;
2962         size_t strsz;
2963
2964         /*
```



```
2965          * The easy case: this probe is allowed to read all of memory, so
2966          * we can just return this as a vanilla pointer.
2967          */
2968         if ((mstate->dtms_access & DTRACE_ACCESS_KERNEL) != 0)
2969                 return (addr);
2970
2971         /*
2972          * This is the tougher case: we copy the string in question from
2973          * kernel memory into scratch memory and return it that way: this
2974          * ensures that we won't trip up when access checking tests the
2975          * BYREF return value.
2976          */
2977         strsz = dtrace_strlen((char *)addr, size) + 1;
2978
2979         if (mstate->dtms_scratch_ptr + strsz >
2980             mstate->dtms_scratch_base + mstate->dtms_scratch_size) {
2981                 DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
2982                 return (NULL);
2983         }
2984
2985         dtrace_strcpy((const void *)addr, (void *)mstate->dtms_scratch_ptr,
2986             strsz);
2987         ret = mstate->dtms_scratch_ptr;
2988         mstate->dtms_scratch_ptr += strsz;
2989         return (ret);
2990 }
2991
2992 /*
2993  * This function implements the DIF emulator's variable lookups.  The emulator
2994  * passes a reserved variable identifier and optional built-in array index.
2995  */
2996 static uint64_t
2997 dtrace_dif_variable(dtrace_mstate_t *mstate, dtrace_state_t *state, uint64_t v,
2998     uint64_t ndx)
2999 {
3000         /*
3001          * If we're accessing one of the uncached arguments, we'll turn this
3002          * into a reference in the args array.
3003          */
3004         if (v >= DIF_VAR_ARG0 && v <= DIF_VAR_ARG9) {
3005                 ndx = v - DIF_VAR_ARG0;
3006                 v = DIF_VAR_ARGS;
3007         }
3008
3009         switch (v) {
3010         case DIF_VAR_ARGS:
3011                 if (!(mstate->dtms_access & DTRACE_ACCESS_ARGS)) {
3012                         cpu_core[CPU->cpu_id].cpuc_dtrace_flags |=
3013                             CPU_DTRACE_KPRIV;
3014                         return (0);
3015                 }
3016
3017                 ASSERT(mstate->dtms_present & DTRACE_MSTATE_ARGS);
3018                 if (ndx >= sizeof (mstate->dtms_arg) /
3019                     sizeof (mstate->dtms_arg[0])) {
3020                         int aframes = mstate->dtms_probe->dtpr_aframes + 2;
3021                         dtrace_provider_t *pv;
3022                         uint64_t val;
3023
3024                         pv = mstate->dtms_probe->dtpr_provider;
3025                         if (pv->dtpv_pops.dtps_getargval != NULL)
3026                                 val = pv->dtpv_pops.dtps_getargval(pv->dtpv_arg,
3027                                     mstate->dtms_probe->dtpr_id,
3028                                     mstate->dtms_probe->dtpr_arg, ndx, aframes);
3029                         else
3030                                 val = dtrace_getarg(ndx, aframes);
```

```
3032                        /*
3033                         * This is regrettably required to keep the compiler
3034                         * from tail-optimizing the call to dtrace_getarg().
3035                         * The condition always evaluates to true, but the
3036                         * compiler has no way of figuring that out a priori.
3037                         * (None of this would be necessary if the compiler
3038                         * could be relied upon to _always_ tail-optimize
3039                         * the call to dtrace_getarg() -- but it can't.)
3040                         */
3041                        if (mstate->dtms_probe != NULL)
3042                                return (val);

3044                        ASSERT(0);
3045                }

3047                return (mstate->dtms_arg[ndx]);

3049        case DIF_VAR_UREGS: {
3050                klwp_t *lwp;

3052                if (!dtrace_priv_proc(state, mstate))
3053                        return (0);

3055                if ((lwp = curthread->t_lwp) == NULL) {
3056                        DTRACE_CPUFLAG_SET(CPU_DTRACE_BADADDR);
3057                        cpu_core[CPU->cpu_id].cpuc_dtrace_illval = NULL;
3058                        return (0);
3059                }

3061                return (dtrace_getreg(lwp->lwp_regs, ndx));
3062        }

3064        case DIF_VAR_VMREGS: {
3065                uint64_t rval;

3067                if (!dtrace_priv_kernel(state))
3068                        return (0);

3070                DTRACE_CPUFLAG_SET(CPU_DTRACE_NOFAULT);

3072                rval = dtrace_getvmreg(ndx,
3073                    &cpu_core[CPU->cpu_id].cpuc_dtrace_flags);

3075                DTRACE_CPUFLAG_CLEAR(CPU_DTRACE_NOFAULT);

3077                return (rval);
3078        }

3080        case DIF_VAR_CURTHREAD:
3081                if (!dtrace_priv_proc(state, mstate))
3082                        return (0);
3083                return ((uint64_t)(uintptr_t)curthread);

3085        case DIF_VAR_TIMESTAMP:
3086                if (!(mstate->dtms_present & DTRACE_MSTATE_TIMESTAMP)) {
3087                        mstate->dtms_timestamp = dtrace_gethrtime();
3088                        mstate->dtms_present |= DTRACE_MSTATE_TIMESTAMP;
3089                }
3090                return (mstate->dtms_timestamp);

3092        case DIF_VAR_VTIMESTAMP:
3093                ASSERT(dtrace_vtime_references != 0);
3094                return (curthread->t_dtrace_vtime);

3096        case DIF_VAR_WALLTIMESTAMP:
```



```
3097                if (!(mstate->dtms_present & DTRACE_MSTATE_WALLTIMESTAMP)) {
3098                        mstate->dtms_walltimestamp = dtrace_gethrestime();
3099                        mstate->dtms_present |= DTRACE_MSTATE_WALLTIMESTAMP;
3100                }
3101                return (mstate->dtms_walltimestamp);

3103        case DIF_VAR_IPL:
3104                if (!dtrace_priv_kernel(state))
3105                        return (0);
3106                if (!(mstate->dtms_present & DTRACE_MSTATE_IPL)) {
3107                        mstate->dtms_ipl = dtrace_getipl();
3108                        mstate->dtms_present |= DTRACE_MSTATE_IPL;
3109                }
3110                return (mstate->dtms_ipl);

3112        case DIF_VAR_EPID:
3113                ASSERT(mstate->dtms_present & DTRACE_MSTATE_EPID);
3114                return (mstate->dtms_epid);

3116        case DIF_VAR_ID:
3117                ASSERT(mstate->dtms_present & DTRACE_MSTATE_PROBE);
3118                return (mstate->dtms_probe->dtpr_id);

3120        case DIF_VAR_STACKDEPTH:
3121                if (!dtrace_priv_kernel(state))
3122                        return (0);
3123                if (!(mstate->dtms_present & DTRACE_MSTATE_STACKDEPTH)) {
3124                        int aframes = mstate->dtms_probe->dtpr_aframes + 2;

3126                        mstate->dtms_stackdepth = dtrace_getstackdepth(aframes);
3127                        mstate->dtms_present |= DTRACE_MSTATE_STACKDEPTH;
3128                }
3129                return (mstate->dtms_stackdepth);

3131        case DIF_VAR_USTACKDEPTH:
3132                if (!dtrace_priv_proc(state, mstate))
3133                        return (0);
3134                if (!(mstate->dtms_present & DTRACE_MSTATE_USTACKDEPTH)) {
3135                        /*
3136                         * See comment in DIF_VAR_PID.
3137                         */
3138                        if (DTRACE_ANCHORED(mstate->dtms_probe) &&
3139                            CPU_ON_INTR(CPU)) {
3140                                mstate->dtms_ustackdepth = 0;
3141                        } else {
3142                                DTRACE_CPUFLAG_SET(CPU_DTRACE_NOFAULT);
3143                                mstate->dtms_ustackdepth =
3144                                    dtrace_getustackdepth();
3145                                DTRACE_CPUFLAG_CLEAR(CPU_DTRACE_NOFAULT);
3146                        }
3147                        mstate->dtms_present |= DTRACE_MSTATE_USTACKDEPTH;
3148                }
3149                return (mstate->dtms_ustackdepth);

3151        case DIF_VAR_CALLER:
3152                if (!dtrace_priv_kernel(state))
3153                        return (0);
3154                if (!(mstate->dtms_present & DTRACE_MSTATE_CALLER)) {
3155                        int aframes = mstate->dtms_probe->dtpr_aframes + 2;

3157                        if (!DTRACE_ANCHORED(mstate->dtms_probe)) {
3158                                /*
3159                                 * If this is an unanchored probe, we are
3160                                 * required to go through the slow path:
3161                                 * dtrace_caller() only guarantees correct
3162                                 * results for anchored probes.
```

```
3163                                 */
3164                                pc_t caller[2];
3165
3166                                dtrace_getpcstack(caller, 2, aframes,
3167                                    (uint32_t *)(uintptr_t)mstate->dtms_arg[0]);
3168                                mstate->dtms_caller = caller[1];
3169                        } else if ((mstate->dtms_caller =
3170                            dtrace_caller(aframes)) == -1) {
3171                                /*
3172                                 * We have failed to do this the quick way;
3173                                 * we must resort to the slower approach of
3174                                 * calling dtrace_getpcstack().
3175                                 */
3176                                pc_t caller;
3177
3178                                dtrace_getpcstack(&caller, 1, aframes, NULL);
3179                                mstate->dtms_caller = caller;
3180                        }
3181
3182                        mstate->dtms_present |= DTRACE_MSTATE_CALLER;
3183                }
3184                return (mstate->dtms_caller);
3185
3186        case DIF_VAR_UCALLER:
3187                if (!dtrace_priv_proc(state, mstate))
3188                        return (0);
3189
3190                if (!(mstate->dtms_present & DTRACE_MSTATE_UCALLER)) {
3191                        uint64_t ustack[3];
3192
3193                        /*
3194                         * dtrace_getupcstack() fills in the first uint64_t
3195                         * with the current PID.  The second uint64_t will
3196                         * be the program counter at user-level.  The third
3197                         * uint64_t will contain the caller, which is what
3198                         * we're after.
3199                         */
3200                        ustack[2] = NULL;
3201                        DTRACE_CPUFLAG_SET(CPU_DTRACE_NOFAULT);
3202                        dtrace_getupcstack(ustack, 3);
3203                        DTRACE_CPUFLAG_CLEAR(CPU_DTRACE_NOFAULT);
3204                        mstate->dtms_ucaller = ustack[2];
3205                        mstate->dtms_present |= DTRACE_MSTATE_UCALLER;
3206                }
3207
3208                return (mstate->dtms_ucaller);
3209
3210        case DIF_VAR_PROBEPROV:
3211                ASSERT(mstate->dtms_present & DTRACE_MSTATE_PROBE);
3212                return (dtrace_dif_varstr(
3213                    (uintptr_t)mstate->dtms_probe->dtpr_provider->dtpv_name,
3214                    state, mstate));
3215
3216        case DIF_VAR_PROBEMOD:
3217                ASSERT(mstate->dtms_present & DTRACE_MSTATE_PROBE);
3218                return (dtrace_dif_varstr(
3219                    (uintptr_t)mstate->dtms_probe->dtpr_mod,
3220                    state, mstate));
3221
3222        case DIF_VAR_PROBEFUNC:
3223                ASSERT(mstate->dtms_present & DTRACE_MSTATE_PROBE);
3224                return (dtrace_dif_varstr(
3225                    (uintptr_t)mstate->dtms_probe->dtpr_func,
3226                    state, mstate));
3227
3228        case DIF_VAR_PROBENAME:
```



```
3229                ASSERT(mstate->dtms_present & DTRACE_MSTATE_PROBE);
3230                return (dtrace_dif_varstr(
3231                    (uintptr_t)mstate->dtms_probe->dtpr_name,
3232                    state, mstate));
3233
3234        case DIF_VAR_PID:
3235                if (!dtrace_priv_proc(state, mstate))
3236                        return (0);
3237
3238                /*
3239                 * Note that we are assuming that an unanchored probe is
3240                 * always due to a high-level interrupt.  (And we're assuming
3241                 * that there is only a single high level interrupt.)
3242                 */
3243                if (DTRACE_ANCHORED(mstate->dtms_probe) && CPU_ON_INTR(CPU))
3244                        return (pid0.pid_id);
3245
3246                /*
3247                 * It is always safe to dereference one's own t_procp pointer:
3248                 * it always points to a valid, allocated proc structure.
3249                 * Further, it is always safe to dereference the p_pidp member
3250                 * of one's own proc structure.  (These are truisms becuase
3251                 * threads and processes don't clean up their own state --
3252                 * they leave that task to whomever reaps them.)
3253                 */
3254                return ((uint64_t)curthread->t_procp->p_pidp->pid_id);
3255
3256        case DIF_VAR_PPID:
3257                if (!dtrace_priv_proc(state, mstate))
3258                        return (0);
3259
3260                /*
3261                 * See comment in DIF_VAR_PID.
3262                 */
3263                if (DTRACE_ANCHORED(mstate->dtms_probe) && CPU_ON_INTR(CPU))
3264                        return (pid0.pid_id);
3265
3266                /*
3267                 * It is always safe to dereference one's own t_procp pointer:
3268                 * it always points to a valid, allocated proc structure.
3269                 * (This is true because threads don't clean up their own
3270                 * state -- they leave that task to whomever reaps them.)
3271                 */
3272                return ((uint64_t)curthread->t_procp->p_ppid);
3273
3274        case DIF_VAR_TID:
3275                /*
3276                 * See comment in DIF_VAR_PID.
3277                 */
3278                if (DTRACE_ANCHORED(mstate->dtms_probe) && CPU_ON_INTR(CPU))
3279                        return (0);
3280
3281                return ((uint64_t)curthread->t_tid);
3282
3283        case DIF_VAR_EXECNAME:
3284                if (!dtrace_priv_proc(state, mstate))
3285                        return (0);
3286
3287                /*
3288                 * See comment in DIF_VAR_PID.
3289                 */
3290                if (DTRACE_ANCHORED(mstate->dtms_probe) && CPU_ON_INTR(CPU))
3291                        return ((uint64_t)(uintptr_t)p0.p_user.u_comm);
3292
3293                /*
3294                 * It is always safe to dereference one's own t_procp pointer:
```

```
3295                 * it always points to a valid, allocated proc structure.
3296                 * (This is true because threads don't clean up their own
3297                 * state -- they leave that task to whomever reaps them.)
3298                 */
3299                return (dtrace_dif_varstr(
3300                    (uintptr_t)curthread->t_procp->p_user.u_comm,
3301                    state, mstate));

3303        case DIF_VAR_ZONENAME:
3304                if (!dtrace_priv_proc(state, mstate))
3305                        return (0);

3307                /*
3308                 * See comment in DIF_VAR_PID.
3309                 */
3310                if (DTRACE_ANCHORED(mstate->dtms_probe) && CPU_ON_INTR(CPU))
3311                        return ((uint64_t)(uintptr_t)p0.p_zone->zone_name);

3313                /*
3314                 * It is always safe to dereference one's own t_procp pointer:
3315                 * it always points to a valid, allocated proc structure.
3316                 * (This is true because threads don't clean up their own
3317                 * state -- they leave that task to whomever reaps them.)
3318                 */
3319                return (dtrace_dif_varstr(
3320                    (uintptr_t)curthread->t_procp->p_zone->zone_name,
3321                    state, mstate));

3323        case DIF_VAR_UID:
3324                if (!dtrace_priv_proc(state, mstate))
3325                        return (0);

3327                /*
3328                 * See comment in DIF_VAR_PID.
3329                 */
3330                if (DTRACE_ANCHORED(mstate->dtms_probe) && CPU_ON_INTR(CPU))
3331                        return ((uint64_t)p0.p_cred->cr_uid);

3333                /*
3334                 * It is always safe to dereference one's own t_procp pointer:
3335                 * it always points to a valid, allocated proc structure.
3336                 * (This is true because threads don't clean up their own
3337                 * state -- they leave that task to whomever reaps them.)
3338                 *
3339                 * Additionally, it is safe to dereference one's own process
3340                 * credential, since this is never NULL after process birth.
3341                 */
3342                return ((uint64_t)curthread->t_procp->p_cred->cr_uid);

3344        case DIF_VAR_GID:
3345                if (!dtrace_priv_proc(state, mstate))
3346                        return (0);

3348                /*
3349                 * See comment in DIF_VAR_PID.
3350                 */
3351                if (DTRACE_ANCHORED(mstate->dtms_probe) && CPU_ON_INTR(CPU))
3352                        return ((uint64_t)p0.p_cred->cr_gid);

3354                /*
3355                 * It is always safe to dereference one's own t_procp pointer:
3356                 * it always points to a valid, allocated proc structure.
3357                 * (This is true because threads don't clean up their own
3358                 * state -- they leave that task to whomever reaps them.)
3359                 *
3360                 * Additionally, it is safe to dereference one's own process
```



```
3361                 * credential, since this is never NULL after process birth.
3362                 */
3363                return ((uint64_t)curthread->t_procp->p_cred->cr_gid);

3365        case DIF_VAR_ERRNO: {
3366                klwp_t *lwp;
3367                if (!dtrace_priv_proc(state, mstate))
3368                        return (0);

3370                /*
3371                 * See comment in DIF_VAR_PID.
3372                 */
3373                if (DTRACE_ANCHORED(mstate->dtms_probe) && CPU_ON_INTR(CPU))
3374                        return (0);

3376                /*
3377                 * It is always safe to dereference one's own t_lwp pointer in
3378                 * the event that this pointer is non-NULL.  (This is true
3379                 * because threads and lwps don't clean up their own state --
3380                 * they leave that task to whomever reaps them.)
3381                 */
3382                if ((lwp = curthread->t_lwp) == NULL)
3383                        return (0);

3385                return ((uint64_t)lwp->lwp_errno);
3386        }
3387        default:
3388                DTRACE_CPUFLAG_SET(CPU_DTRACE_ILLOP);
3389                return (0);
3390        }
3391 }


3394 typedef enum dtrace_json_state {
3395        DTRACE_JSON_REST = 1,
3396        DTRACE_JSON_OBJECT,
3397        DTRACE_JSON_STRING,
3398        DTRACE_JSON_STRING_ESCAPE,
3399        DTRACE_JSON_STRING_ESCAPE_UNICODE,
3400        DTRACE_JSON_COLON,
3401        DTRACE_JSON_COMMA,
3402        DTRACE_JSON_VALUE,
3403        DTRACE_JSON_IDENTIFIER,
3404        DTRACE_JSON_NUMBER,
3405        DTRACE_JSON_NUMBER_FRAC,
3406        DTRACE_JSON_NUMBER_EXP,
3407        DTRACE_JSON_COLLECT_OBJECT
3408 } dtrace_json_state_t;

3410 /*
3411  * This function possesses just enough knowledge about JSON to extract a single
3412  * value from a JSON string and store it in the scratch buffer.  It is able
3413  * to extract nested object values, and members of arrays by index.
3414  *
3415  * elemlist is a list of JSON keys, stored as packed NUL-terminated strings, to
3416  * be looked up as we descend into the object tree.  e.g.
3417  *
3418  *    foo[0].bar.baz[32] --> "foo" NUL "0" NUL "bar" NUL "baz" NUL "32" NUL
3419  *       with nelems = 5.
3420  *
3421  * The run time of this function must be bounded above by strsize to limit the
3422  * amount of work done in probe context.  As such, it is implemented as a
3423  * simple state machine, reading one character at a time using safe loads
3424  * until we find the requested element, hit a parsing error or run off the
3425  * end of the object or string.
3426  *
```

```
3427  * As there is no way for a subroutine to return an error without interrupting
3428  * clause execution, we simply return NULL in the event of a missing key or any
3429  * other error condition.  Each NULL return in this function is commented with
3430  * the error condition it represents -- parsing or otherwise.
3431  *
3432  * The set of states for the state machine closely matches the JSON
3433  * specification (http://json.org/).  Briefly:
3434  *
3435  *   DTRACE_JSON_REST:
3436  *     Skip whitespace until we find either a top-level Object, moving
3437  *     to DTRACE_JSON_OBJECT; or an Array, moving to DTRACE_JSON_VALUE.
3438  *
3439  *   DTRACE_JSON_OBJECT:
3440  *     Locate the next key String in an Object.  Sets a flag to denote
3441  *     the next String as a key string and moves to DTRACE_JSON_STRING.
3442  *
3443  *   DTRACE_JSON_COLON:
3444  *     Skip whitespace until we find the colon that separates key Strings
3445  *     from their values.  Once found, move to DTRACE_JSON_VALUE.
3446  *
3447  *   DTRACE_JSON_VALUE:
3448  *     Detects the type of the next value (String, Number, Identifier, Object
3449  *     or Array) and routes to the states that process that type.  Here we also
3450  *     deal with the element selector list if we are requested to traverse down
3451  *     into the object tree.
3452  *
3453  *   DTRACE_JSON_COMMA:
3454  *     Skip whitespace until we find the comma that separates key-value pairs
3455  *     in Objects (returning to DTRACE_JSON_OBJECT) or values in Arrays
3456  *     (similarly DTRACE_JSON_VALUE).  All following literal value processing
3457  *     states return to this state at the end of their value, unless otherwise
3458  *     noted.
3459  *
3460  *   DTRACE_JSON_NUMBER, DTRACE_JSON_NUMBER_FRAC, DTRACE_JSON_NUMBER_EXP:
3461  *     Processes a Number literal from the JSON, including any exponent
3462  *     component that may be present.  Numbers are returned as strings, which
3463  *     may be passed to strtoll() if an integer is required.
3464  *
3465  *   DTRACE_JSON_IDENTIFIER:
3466  *     Processes a "true", "false" or "null" literal in the JSON.
3467  *
3468  *   DTRACE_JSON_STRING, DTRACE_JSON_STRING_ESCAPE,
3469  *   DTRACE_JSON_STRING_ESCAPE_UNICODE:
3470  *     Processes a String literal from the JSON, whether the String denotes
3471  *     a key, a value or part of a larger Object.  Handles all escape sequences
3472  *     present in the specification, including four-digit unicode characters,
3473  *     but merely includes the escape sequence without converting it to the
3474  *     actual escaped character.  If the String is flagged as a key, we
3475  *     move to DTRACE_JSON_COLON rather than DTRACE_JSON_COMMA.
3476  *
3477  *   DTRACE_JSON_COLLECT_OBJECT:
3478  *     This state collects an entire Object (or Array), correctly handling
3479  *     embedded strings.  If the full element selector list matches this nested
3480  *     object, we return the Object in full as a string.  If not, we use this
3481  *     state to skip to the next value at this level and continue processing.
3482  *
3483  * NOTE: This function uses various macros from strtolctype.h to manipulate
3484  * digit values, etc -- these have all been checked to ensure they make
3485  * no additional function calls.
3486  */
3487 static char *
3488 dtrace_json(uint64_t size, uintptr_t json, char *elemlist, int nelems,
3489     char *dest)
3490 {
3491         dtrace_json_state_t state = DTRACE_JSON_REST;
3492         int64_t array_elem = INT64_MIN;
```



```
3493         int64_t array_pos = 0;
3494         uint8_t escape_unicount = 0;
3495         boolean_t string_is_key = B_FALSE;
3496         boolean_t collect_object = B_FALSE;
3497         boolean_t found_key = B_FALSE;
3498         boolean_t in_array = B_FALSE;
3499         uint32_t braces = 0, brackets = 0;
3500         char *elem = elemlist;
3501         char *dd = dest;
3502         uintptr_t cur;
3503
3504         for (cur = json; cur < json + size; cur++) {
3505                 char cc = dtrace_load8(cur);
3506                 if (cc == '\0')
3507                         return (NULL);
3508
3509                 switch (state) {
3510                 case DTRACE_JSON_REST:
3511                         if (isspace(cc))
3512                                 break;
3513
3514                         if (cc == '{') {
3515                                 state = DTRACE_JSON_OBJECT;
3516                                 break;
3517                         }
3518
3519                         if (cc == '[') {
3520                                 in_array = B_TRUE;
3521                                 array_pos = 0;
3522                                 array_elem = dtrace_strtoll(elem, 10, size);
3523                                 found_key = array_elem == 0 ? B_TRUE : B_FALSE;
3524                                 state = DTRACE_JSON_VALUE;
3525                                 break;
3526                         }
3527
3528                         /*
3529                          * ERROR: expected to find a top-level object or array.
3530                          */
3531                         return (NULL);
3532                 case DTRACE_JSON_OBJECT:
3533                         if (isspace(cc))
3534                                 break;
3535
3536                         if (cc == '"') {
3537                                 state = DTRACE_JSON_STRING;
3538                                 string_is_key = B_TRUE;
3539                                 break;
3540                         }
3541
3542                         /*
3543                          * ERROR: either the object did not start with a key
3544                          * string, or we've run off the end of the object
3545                          * without finding the requested key.
3546                          */
3547                         return (NULL);
3548                 case DTRACE_JSON_STRING:
3549                         if (cc == '\\') {
3550                                 *dd++ = '\\';
3551                                 state = DTRACE_JSON_STRING_ESCAPE;
3552                                 break;
3553                         }
3554
3555                         if (cc == '"') {
3556                                 if (collect_object) {
3557                                         /*
3558                                          * We don't reset the dest here, as
```

```
3559                                     * the string is part of a larger
3560                                     * object being collected.
3561                                     */
3562                                    *dd++ = cc;
3563                                    collect_object = B_FALSE;
3564                                    state = DTRACE_JSON_COLLECT_OBJECT;
3565                                    break;
3566                            }
3567                            *dd = '\0';
3568                            dd = dest; /* reset string buffer */
3569                            if (string_is_key) {
3570                                    if (dtrace_strncmp(dest, elem,
3571                                        size) == 0)
3572                                            found_key = B_TRUE;
3573                            } else if (found_key) {
3574                                    if (nelems > 1) {
3575                                            /*
3576                                             * We expected an object, not
3577                                             * this string.
3578                                             */
3579                                            return (NULL);
3580                                    }
3581                                    return (dest);
3582                            }
3583                            state = string_is_key ? DTRACE_JSON_COLON :
3584                                DTRACE_JSON_COMMA;
3585                            string_is_key = B_FALSE;
3586                            break;
3587                    }

3589                    *dd++ = cc;
3590                    break;
3591            case DTRACE_JSON_STRING_ESCAPE:
3592                    *dd++ = cc;
3593                    if (cc == 'u') {
3594                            escape_unicount = 0;
3595                            state = DTRACE_JSON_STRING_ESCAPE_UNICODE;
3596                    } else {
3597                            state = DTRACE_JSON_STRING;
3598                    }
3599                    break;
3600            case DTRACE_JSON_STRING_ESCAPE_UNICODE:
3601                    if (!isxdigit(cc)) {
3602                            /*
3603                             * ERROR: invalid unicode escape, expected
3604                             * four valid hexidecimal digits.
3605                             */
3606                            return (NULL);
3607                    }

3609                    *dd++ = cc;
3610                    if (++escape_unicount == 4)
3611                            state = DTRACE_JSON_STRING;
3612                    break;
3613            case DTRACE_JSON_COLON:
3614                    if (isspace(cc))
3615                            break;

3617                    if (cc == ':') {
3618                            state = DTRACE_JSON_VALUE;
3619                            break;
3620                    }

3622                    /*
3623                     * ERROR: expected a colon.
3624                     */
```



```
3625                            return (NULL);
3626                    case DTRACE_JSON_COMMA:
3627                            if (isspace(cc))
3628                                    break;

3630                            if (cc == ',') {
3631                                    if (in_array) {
3632                                            state = DTRACE_JSON_VALUE;
3633                                            if (++array_pos == array_elem)
3634                                                    found_key = B_TRUE;
3635                                    } else {
3636                                            state = DTRACE_JSON_OBJECT;
3637                                    }
3638                                    break;
3639                            }

3641                            /*
3642                             * ERROR: either we hit an unexpected character, or
3643                             * we reached the end of the object or array without
3644                             * finding the requested key.
3645                             */
3646                            return (NULL);
3647                    case DTRACE_JSON_IDENTIFIER:
3648                            if (islower(cc)) {
3649                                    *dd++ = cc;
3650                                    break;
3651                            }

3653                            *dd = '\0';
3654                            dd = dest; /* reset string buffer */

3656                            if (dtrace_strncmp(dest, "true", 5) == 0 ||
3657                                dtrace_strncmp(dest, "false", 6) == 0 ||
3658                                dtrace_strncmp(dest, "null", 5) == 0) {
3659                                    if (found_key) {
3660                                            if (nelems > 1) {
3661                                                    /*
3662                                                     * ERROR: We expected an object,
3663                                                     * not this identifier.
3664                                                     */
3665                                                    return (NULL);
3666                                            }
3667                                            return (dest);
3668                                    } else {
3669                                            cur--;
3670                                            state = DTRACE_JSON_COMMA;
3671                                            break;
3672                                    }
3673                            }

3675                            /*
3676                             * ERROR: we did not recognise the identifier as one
3677                             * of those in the JSON specification.
3678                             */
3679                            return (NULL);
3680                    case DTRACE_JSON_NUMBER:
3681                            if (cc == '.') {
3682                                    *dd++ = cc;
3683                                    state = DTRACE_JSON_NUMBER_FRAC;
3684                                    break;
3685                            }

3687                            if (cc == 'x' || cc == 'X') {
3688                                    /*
3689                                     * ERROR: specification explicitly excludes
3690                                     * hexidecimal or octal numbers.
```

```
3691                                 */
3692                                return (NULL);
3693                        }

3695                        /* FALLTHRU */
3696                case DTRACE_JSON_NUMBER_FRAC:
3697                        if (cc == 'e' || cc == 'E') {
3698                                *dd++ = cc;
3699                                state = DTRACE_JSON_NUMBER_EXP;
3700                                break;
3701                        }

3703                        if (cc == '+' || cc == '-') {
3704                                /*
3705                                 * ERROR: expect sign as part of exponent only.
3706                                 */
3707                                return (NULL);
3708                        }
3709                        /* FALLTHRU */
3710                case DTRACE_JSON_NUMBER_EXP:
3711                        if (isdigit(cc) || cc == '+' || cc == '-') {
3712                                *dd++ = cc;
3713                                break;
3714                        }

3716                        *dd = '\0';
3717                        dd = dest; /* reset string buffer */
3718                        if (found_key) {
3719                                if (nelems > 1) {
3720                                        /*
3721                                         * ERROR: We expected an object, not
3722                                         * this number.
3723                                         */
3724                                        return (NULL);
3725                                }
3726                                return (dest);
3727                        }

3729                        cur--;
3730                        state = DTRACE_JSON_COMMA;
3731                        break;
3732                case DTRACE_JSON_VALUE:
3733                        if (isspace(cc))
3734                                break;

3736                        if (cc == '{' || cc == '[') {
3737                                if (nelems > 1 && found_key) {
3738                                        in_array = cc == '[' ? B_TRUE : B_FALSE;
3739                                        /*
3740                                         * If our element selector directs us
3741                                         * to descend into this nested object,
3742                                         * then move to the next selector
3743                                         * element in the list and restart the
3744                                         * state machine.
3745                                         */
3746                                        while (*elem != '\0')
3747                                                elem++;
3748                                        elem++; /* skip the inter-element NUL */
3749                                        nelems--;
3750                                        dd = dest;
3751                                        if (in_array) {
3752                                                state = DTRACE_JSON_VALUE;
3753                                                array_pos = 0;
3754                                                array_elem = dtrace_strtoll(
3755                                                    elem, 10, size);
3756                                                found_key = array_elem == 0 ?
```



```
3757                                                    B_TRUE : B_FALSE;
3758                                        } else {
3759                                                found_key = B_FALSE;
3760                                                state = DTRACE_JSON_OBJECT;
3761                                        }
3762                                        break;
3763                                }

3765                                /*
3766                                 * Otherwise, we wish to either skip this
3767                                 * nested object or return it in full.
3768                                 */
3769                                if (cc == '[')
3770                                        brackets = 1;
3771                                else
3772                                        braces = 1;
3773                                *dd++ = cc;
3774                                state = DTRACE_JSON_COLLECT_OBJECT;
3775                                break;
3776                        }

3778                        if (cc == '"') {
3779                                state = DTRACE_JSON_STRING;
3780                                break;
3781                        }

3783                        if (islower(cc)) {
3784                                /*
3785                                 * Here we deal with true, false and null.
3786                                 */
3787                                *dd++ = cc;
3788                                state = DTRACE_JSON_IDENTIFIER;
3789                                break;
3790                        }

3792                        if (cc == '-' || isdigit(cc)) {
3793                                *dd++ = cc;
3794                                state = DTRACE_JSON_NUMBER;
3795                                break;
3796                        }

3798                        /*
3799                         * ERROR: unexpected character at start of value.
3800                         */
3801                        return (NULL);
3802                case DTRACE_JSON_COLLECT_OBJECT:
3803                        if (cc == '\0')
3804                                /*
3805                                 * ERROR: unexpected end of input.
3806                                 */
3807                                return (NULL);

3809                        *dd++ = cc;
3810                        if (cc == '"') {
3811                                collect_object = B_TRUE;
3812                                state = DTRACE_JSON_STRING;
3813                                break;
3814                        }

3816                        if (cc == ']') {
3817                                if (brackets-- == 0) {
3818                                        /*
3819                                         * ERROR: unbalanced brackets.
3820                                         */
3821                                        return (NULL);
3822                                }
```

```
3823                        } else if (cc == '}') {
3824                                if (braces-- == 0) {
3825                                        /*
3826                                         * ERROR: unbalanced braces.
3827                                         */
3828                                        return (NULL);
3829                                }
3830                        } else if (cc == '{') {
3831                                braces++;
3832                        } else if (cc == '[') {
3833                                brackets++;
3834                        }

3836                        if (brackets == 0 && braces == 0) {
3837                                if (found_key) {
3838                                        *dd = '\0';
3839                                        return (dest);
3840                                }
3841                                dd = dest; /* reset string buffer */
3842                                state = DTRACE_JSON_COMMA;
3843                        }
3844                        break;
3845                }
3846        }
3847        return (NULL);
3848 }

3850 #endif /* ! codereview */
3851 /*
3852  * Emulate the execution of DTrace ID subroutines invoked by the call opcode.
3853  * Notice that we don't bother validating the proper number of arguments or
3854  * their types in the tuple stack.  This isn't needed because all argument
3855  * interpretation is safe because of our load safety -- the worst that can
3856  * happen is that a bogus program can obtain bogus results.
3857  */
3858 static void
3859 dtrace_dif_subr(uint_t subr, uint_t rd, uint64_t *regs,
3860     dtrace_key_t *tupregs, int nargs,
3861     dtrace_mstate_t *mstate, dtrace_state_t *state)
3862 {
3863        volatile uint16_t *flags = &cpu_core[CPU->cpu_id].cpuc_dtrace_flags;
3864        volatile uintptr_t *illval = &cpu_core[CPU->cpu_id].cpuc_dtrace_illval;
3865        dtrace_vstate_t *vstate = &state->dts_vstate;

3867        union {
3868                mutex_impl_t mi;
3869                uint64_t mx;
3870        } m;

3872        union {
3873                krwlock_t ri;
3874                uintptr_t rw;
3875        } r;

3877        switch (subr) {
3878        case DIF_SUBR_RAND:
3879                regs[rd] = (dtrace_gethrtime() * 2416 + 374441) % 1771875;
3880                break;

3882        case DIF_SUBR_MUTEX_OWNED:
3883                if (!dtrace_canload(tupregs[0].dttk_value, sizeof (kmutex_t),
3884                    mstate, vstate)) {
3885                        regs[rd] = NULL;
3886                        break;
3887                }
```



```
3889                m.mx = dtrace_load64(tupregs[0].dttk_value);
3890                if (MUTEX_TYPE_ADAPTIVE(&m.mi))
3891                        regs[rd] = MUTEX_OWNER(&m.mi) != MUTEX_NO_OWNER;
3892                else
3893                        regs[rd] = LOCK_HELD(&m.mi.m_spin.m_spinlock);
3894                break;

3896        case DIF_SUBR_MUTEX_OWNER:
3897                if (!dtrace_canload(tupregs[0].dttk_value, sizeof (kmutex_t),
3898                    mstate, vstate)) {
3899                        regs[rd] = NULL;
3900                        break;
3901                }

3903                m.mx = dtrace_load64(tupregs[0].dttk_value);
3904                if (MUTEX_TYPE_ADAPTIVE(&m.mi) &&
3905                    MUTEX_OWNER(&m.mi) != MUTEX_NO_OWNER)
3906                        regs[rd] = (uintptr_t)MUTEX_OWNER(&m.mi);
3907                else
3908                        regs[rd] = 0;
3909                break;

3911        case DIF_SUBR_MUTEX_TYPE_ADAPTIVE:
3912                if (!dtrace_canload(tupregs[0].dttk_value, sizeof (kmutex_t),
3913                    mstate, vstate)) {
3914                        regs[rd] = NULL;
3915                        break;
3916                }

3918                m.mx = dtrace_load64(tupregs[0].dttk_value);
3919                regs[rd] = MUTEX_TYPE_ADAPTIVE(&m.mi);
3920                break;

3922        case DIF_SUBR_MUTEX_TYPE_SPIN:
3923                if (!dtrace_canload(tupregs[0].dttk_value, sizeof (kmutex_t),
3924                    mstate, vstate)) {
3925                        regs[rd] = NULL;
3926                        break;
3927                }

3929                m.mx = dtrace_load64(tupregs[0].dttk_value);
3930                regs[rd] = MUTEX_TYPE_SPIN(&m.mi);
3931                break;

3933        case DIF_SUBR_RW_READ_HELD: {
3934                uintptr_t tmp;

3936                if (!dtrace_canload(tupregs[0].dttk_value, sizeof (uintptr_t),
3937                    mstate, vstate)) {
3938                        regs[rd] = NULL;
3939                        break;
3940                }

3942                r.rw = dtrace_loadptr(tupregs[0].dttk_value);
3943                regs[rd] = _RW_READ_HELD(&r.ri, tmp);
3944                break;
3945        }

3947        case DIF_SUBR_RW_WRITE_HELD:
3948                if (!dtrace_canload(tupregs[0].dttk_value, sizeof (krwlock_t),
3949                    mstate, vstate)) {
3950                        regs[rd] = NULL;
3951                        break;
3952                }

3954                r.rw = dtrace_loadptr(tupregs[0].dttk_value);
```

```
3955                regs[rd] = _RW_WRITE_HELD(&r.ri);
3956                break;
3957
3958        case DIF_SUBR_RW_ISWRITER:
3959                if (!dtrace_canload(tupregs[0].dttk_value, sizeof (krwlock_t),
3960                    mstate, vstate)) {
3961                        regs[rd] = NULL;
3962                        break;
3963                }
3964
3965                r.rw = dtrace_loadptr(tupregs[0].dttk_value);
3966                regs[rd] = _RW_ISWRITER(&r.ri);
3967                break;
3968
3969        case DIF_SUBR_BCOPY: {
3970                /*
3971                 * We need to be sure that the destination is in the scratch
3972                 * region -- no other region is allowed.
3973                 */
3974                uintptr_t src = tupregs[0].dttk_value;
3975                uintptr_t dest = tupregs[1].dttk_value;
3976                size_t size = tupregs[2].dttk_value;
3977
3978                if (!dtrace_inscratch(dest, size, mstate)) {
3979                        *flags |= CPU_DTRACE_BADADDR;
3980                        *illval = regs[rd];
3981                        break;
3982                }
3983
3984                if (!dtrace_canload(src, size, mstate, vstate)) {
3985                        regs[rd] = NULL;
3986                        break;
3987                }
3988
3989                dtrace_bcopy((void *)src, (void *)dest, size);
3990                break;
3991        }
3992
3993        case DIF_SUBR_ALLOCA:
3994        case DIF_SUBR_COPYIN: {
3995                uintptr_t dest = P2ROUNDUP(mstate->dtms_scratch_ptr, 8);
3996                uint64_t size =
3997                    tupregs[subr == DIF_SUBR_ALLOCA ? 0 : 1].dttk_value;
3998                size_t scratch_size = (dest - mstate->dtms_scratch_ptr) + size;
3999
4000                /*
4001                 * This action doesn't require any credential checks since
4002                 * probes will not activate in user contexts to which the
4003                 * enabling user does not have permissions.
4004                 */
4005
4006                /*
4007                 * Rounding up the user allocation size could have overflowed
4008                 * a large, bogus allocation (like -1ULL) to 0.
4009                 */
4010                if (scratch_size < size ||
4011                    !DTRACE_INSCRATCH(mstate, scratch_size)) {
4012                        DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
4013                        regs[rd] = NULL;
4014                        break;
4015                }
4016
4017                if (subr == DIF_SUBR_COPYIN) {
4018                        DTRACE_CPUFLAG_SET(CPU_DTRACE_NOFAULT);
4019                        dtrace_copyin(tupregs[0].dttk_value, dest, size, flags);
4020                        DTRACE_CPUFLAG_CLEAR(CPU_DTRACE_NOFAULT);
```



```
4021                }
4022
4023                mstate->dtms_scratch_ptr += scratch_size;
4024                regs[rd] = dest;
4025                break;
4026        }
4027
4028        case DIF_SUBR_COPYINTO: {
4029                uint64_t size = tupregs[1].dttk_value;
4030                uintptr_t dest = tupregs[2].dttk_value;
4031
4032                /*
4033                 * This action doesn't require any credential checks since
4034                 * probes will not activate in user contexts to which the
4035                 * enabling user does not have permissions.
4036                 */
4037                if (!dtrace_inscratch(dest, size, mstate)) {
4038                        *flags |= CPU_DTRACE_BADADDR;
4039                        *illval = regs[rd];
4040                        break;
4041                }
4042
4043                DTRACE_CPUFLAG_SET(CPU_DTRACE_NOFAULT);
4044                dtrace_copyin(tupregs[0].dttk_value, dest, size, flags);
4045                DTRACE_CPUFLAG_CLEAR(CPU_DTRACE_NOFAULT);
4046                break;
4047        }
4048
4049        case DIF_SUBR_COPYINSTR: {
4050                uintptr_t dest = mstate->dtms_scratch_ptr;
4051                uint64_t size = state->dts_options[DTRACEOPT_STRSIZE];
4052
4053                if (nargs > 1 && tupregs[1].dttk_value < size)
4054                        size = tupregs[1].dttk_value + 1;
4055
4056                /*
4057                 * This action doesn't require any credential checks since
4058                 * probes will not activate in user contexts to which the
4059                 * enabling user does not have permissions.
4060                 */
4061                if (!DTRACE_INSCRATCH(mstate, size)) {
4062                        DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
4063                        regs[rd] = NULL;
4064                        break;
4065                }
4066
4067                DTRACE_CPUFLAG_SET(CPU_DTRACE_NOFAULT);
4068                dtrace_copyinstr(tupregs[0].dttk_value, dest, size, flags);
4069                DTRACE_CPUFLAG_CLEAR(CPU_DTRACE_NOFAULT);
4070
4071                ((char *)dest)[size - 1] = '\0';
4072                mstate->dtms_scratch_ptr += size;
4073                regs[rd] = dest;
4074                break;
4075        }
4076
4077        case DIF_SUBR_MSGSIZE:
4078        case DIF_SUBR_MSGDSIZE: {
4079                uintptr_t baddr = tupregs[0].dttk_value, daddr;
4080                uintptr_t wptr, rptr;
4081                size_t count = 0;
4082                int cont = 0;
4083
4084                while (baddr != NULL && !(*flags & CPU_DTRACE_FAULT)) {
4085
4086                        if (!dtrace_canload(baddr, sizeof (mblk_t), mstate,
```

```
4087                            vstate)) {
4088                                regs[rd] = NULL;
4089                                break;
4090                        }
4091
4092                        wptr = dtrace_loadptr(baddr +
4093                            offsetof(mblk_t, b_wptr));
4094
4095                        rptr = dtrace_loadptr(baddr +
4096                            offsetof(mblk_t, b_rptr));
4097
4098                        if (wptr < rptr) {
4099                                *flags |= CPU_DTRACE_BADADDR;
4100                                *illval = tupregs[0].dttk_value;
4101                                break;
4102                        }
4103
4104                        daddr = dtrace_loadptr(baddr +
4105                            offsetof(mblk_t, b_datap));
4106
4107                        baddr = dtrace_loadptr(baddr +
4108                            offsetof(mblk_t, b_cont));
4109
4110                        /*
4111                         * We want to prevent against denial-of-service here,
4112                         * so we're only going to search the list for
4113                         * dtrace_msgdsize_max mblks.
4114                         */
4115                        if (cont++ > dtrace_msgdsize_max) {
4116                                *flags |= CPU_DTRACE_ILLOP;
4117                                break;
4118                        }
4119
4120                        if (subr == DIF_SUBR_MSGDSIZE) {
4121                                if (dtrace_load8(daddr +
4122                                    offsetof(dblk_t, db_type)) != M_DATA)
4123                                        continue;
4124                        }
4125
4126                        count += wptr - rptr;
4127                }
4128
4129                if (!(*flags & CPU_DTRACE_FAULT))
4130                        regs[rd] = count;
4131
4132                break;
4133        }
4134
4135        case DIF_SUBR_PROGENYOF: {
4136                pid_t pid = tupregs[0].dttk_value;
4137                proc_t *p;
4138                int rval = 0;
4139
4140                DTRACE_CPUFLAG_SET(CPU_DTRACE_NOFAULT);
4141
4142                for (p = curthread->t_procp; p != NULL; p = p->p_parent) {
4143                        if (p->p_pidp->pid_id == pid) {
4144                                rval = 1;
4145                                break;
4146                        }
4147                }
4148
4149                DTRACE_CPUFLAG_CLEAR(CPU_DTRACE_NOFAULT);
4150
4151                regs[rd] = rval;
4152                break;
```



```
4153        }
4154
4155        case DIF_SUBR_SPECULATION:
4156                regs[rd] = dtrace_speculation(state);
4157                break;
4158
4159        case DIF_SUBR_COPYOUT: {
4160                uintptr_t kaddr = tupregs[0].dttk_value;
4161                uintptr_t uaddr = tupregs[1].dttk_value;
4162                uint64_t size = tupregs[2].dttk_value;
4163
4164                if (!dtrace_destructive_disallow &&
4165                    dtrace_priv_proc_control(state, mstate) &&
4166                    !dtrace_istoxic(kaddr, size)) {
4167                        DTRACE_CPUFLAG_SET(CPU_DTRACE_NOFAULT);
4168                        dtrace_copyout(kaddr, uaddr, size, flags);
4169                        DTRACE_CPUFLAG_CLEAR(CPU_DTRACE_NOFAULT);
4170                }
4171                break;
4172        }
4173
4174        case DIF_SUBR_COPYOUTSTR: {
4175                uintptr_t kaddr = tupregs[0].dttk_value;
4176                uintptr_t uaddr = tupregs[1].dttk_value;
4177                uint64_t size = tupregs[2].dttk_value;
4178
4179                if (!dtrace_destructive_disallow &&
4180                    dtrace_priv_proc_control(state, mstate) &&
4181                    !dtrace_istoxic(kaddr, size)) {
4182                        DTRACE_CPUFLAG_SET(CPU_DTRACE_NOFAULT);
4183                        dtrace_copyoutstr(kaddr, uaddr, size, flags);
4184                        DTRACE_CPUFLAG_CLEAR(CPU_DTRACE_NOFAULT);
4185                }
4186                break;
4187        }
4188
4189        case DIF_SUBR_STRLEN: {
4190                size_t sz;
4191                uintptr_t addr = (uintptr_t)tupregs[0].dttk_value;
4192                sz = dtrace_strlen((char *)addr,
4193                    state->dts_options[DTRACEOPT_STRSIZE]);
4194
4195                if (!dtrace_canload(addr, sz + 1, mstate, vstate)) {
4196                        regs[rd] = NULL;
4197                        break;
4198                }
4199
4200                regs[rd] = sz;
4201
4202                break;
4203        }
4204
4205        case DIF_SUBR_STRCHR:
4206        case DIF_SUBR_STRRCHR: {
4207                /*
4208                 * We're going to iterate over the string looking for the
4209                 * specified character.  We will iterate until we have reached
4210                 * the string length or we have found the character.  If this
4211                 * is DIF_SUBR_STRRCHR, we will look for the last occurrence
4212                 * of the specified character instead of the first.
4213                 */
4214                uintptr_t saddr = tupregs[0].dttk_value;
4215                uintptr_t addr = tupregs[0].dttk_value;
4216                uintptr_t limit = addr + state->dts_options[DTRACEOPT_STRSIZE];
4217                char c, target = (char)tupregs[1].dttk_value;
```

```
4219                for (regs[rd] = NULL; addr < limit; addr++) {
4220                        if ((c = dtrace_load8(addr)) == target) {
4221                                regs[rd] = addr;

4223                                if (subr == DIF_SUBR_STRCHR)
4224                                        break;
4225                        }

4227                        if (c == '\0')
4228                                break;
4229                }

4231                if (!dtrace_canload(saddr, addr - saddr, mstate, vstate)) {
4232                        regs[rd] = NULL;
4233                        break;
4234                }

4236                break;
4237        }

4239        case DIF_SUBR_STRSTR:
4240        case DIF_SUBR_INDEX:
4241        case DIF_SUBR_RINDEX: {
4242                /*
4243                 * We're going to iterate over the string looking for the
4244                 * specified string.  We will iterate until we have reached
4245                 * the string length or we have found the string.  (Yes, this
4246                 * is done in the most naive way possible -- but considering
4247                 * that the string we're searching for is likely to be
4248                 * relatively short, the complexity of Rabin-Karp or similar
4249                 * hardly seems merited.)
4250                 */
4251                char *addr = (char *)(uintptr_t)tupregs[0].dttk_value;
4252                char *substr = (char *)(uintptr_t)tupregs[1].dttk_value;
4253                uint64_t size = state->dts_options[DTRACEOPT_STRSIZE];
4254                size_t len = dtrace_strlen(addr, size);
4255                size_t sublen = dtrace_strlen(substr, size);
4256                char *limit = addr + len, *orig = addr;
4257                int notfound = subr == DIF_SUBR_STRSTR ? 0 : -1;
4258                int inc = 1;

4260                regs[rd] = notfound;

4262                if (!dtrace_canload((uintptr_t)addr, len + 1, mstate, vstate)) {
4263                        regs[rd] = NULL;
4264                        break;
4265                }

4267                if (!dtrace_canload((uintptr_t)substr, sublen + 1, mstate,
4268                    vstate)) {
4269                        regs[rd] = NULL;
4270                        break;
4271                }

4273                /*
4274                 * strstr() and index()/rindex() have similar semantics if
4275                 * both strings are the empty string: strstr() returns a
4276                 * pointer to the (empty) string, and index() and rindex()
4277                 * both return index 0 (regardless of any position argument).
4278                 */
4279                if (sublen == 0 && len == 0) {
4280                        if (subr == DIF_SUBR_STRSTR)
4281                                regs[rd] = (uintptr_t)addr;
4282                        else
4283                                regs[rd] = 0;
4284                        break;
```



```
4285                }

4287                if (subr != DIF_SUBR_STRSTR) {
4288                        if (subr == DIF_SUBR_RINDEX) {
4289                                limit = orig - 1;
4290                                addr += len;
4291                                inc = -1;
4292                        }

4294                        /*
4295                         * Both index() and rindex() take an optional position
4296                         * argument that denotes the starting position.
4297                         */
4298                        if (nargs == 3) {
4299                                int64_t pos = (int64_t)tupregs[2].dttk_value;

4301                                /*
4302                                 * If the position argument to index() is
4303                                 * negative, Perl implicitly clamps it at
4304                                 * zero.  This semantic is a little surprising
4305                                 * given the special meaning of negative
4306                                 * positions to similar Perl functions like
4307                                 * substr(), but it appears to reflect a
4308                                 * notion that index() can start from a
4309                                 * negative index and increment its way up to
4310                                 * the string.  Given this notion, Perl's
4311                                 * rindex() is at least self-consistent in
4312                                 * that it implicitly clamps positions greater
4313                                 * than the string length to be the string
4314                                 * length.  Where Perl completely loses
4315                                 * coherence, however, is when the specified
4316                                 * substring is the empty string ("").  In
4317                                 * this case, even if the position is
4318                                 * negative, rindex() returns 0 -- and even if
4319                                 * the position is greater than the length,
4320                                 * index() returns the string length.  These
4321                                 * semantics violate the notion that index()
4322                                 * should never return a value less than the
4323                                 * specified position and that rindex() should
4324                                 * never return a value greater than the
4325                                 * specified position.  (One assumes that
4326                                 * these semantics are artifacts of Perl's
4327                                 * implementation and not the results of
4328                                 * deliberate design -- it beggars belief that
4329                                 * even Larry Wall could desire such oddness.)
4330                                 * While in the abstract one would wish for
4331                                 * consistent position semantics across
4332                                 * substr(), index() and rindex() -- or at the
4333                                 * very least self-consistent position
4334                                 * semantics for index() and rindex() -- we
4335                                 * instead opt to keep with the extant Perl
4336                                 * semantics, in all their broken glory.  (Do
4337                                 * we have more desire to maintain Perl's
4338                                 * semantics than Perl does?  Probably.)
4339                                 */
4340                                if (subr == DIF_SUBR_RINDEX) {
4341                                        if (pos < 0) {
4342                                                if (sublen == 0)
4343                                                        regs[rd] = 0;
4344                                                break;
4345                                        }

4347                                        if (pos > len)
4348                                                pos = len;
4349                                } else {
4350                                        if (pos < 0)
```

```
4351                                            pos = 0;
4352
4353                                    if (pos >= len) {
4354                                            if (sublen == 0)
4355                                                    regs[rd] = len;
4356                                            break;
4357                                    }
4358                            }
4359
4360                            addr = orig + pos;
4361                    }
4362            }
4363
4364            for (regs[rd] = notfound; addr != limit; addr += inc) {
4365                    if (dtrace_strncmp(addr, substr, sublen) == 0) {
4366                            if (subr != DIF_SUBR_STRSTR) {
4367                                    /*
4368                                     * As D index() and rindex() are
4369                                     * modeled on Perl (and not on awk),
4370                                     * we return a zero-based (and not a
4371                                     * one-based) index.  (For you Perl
4372                                     * weenies: no, we're not going to add
4373                                     * $[ -- and shouldn't you be at a con
4374                                     * or something?)
4375                                     */
4376                                    regs[rd] = (uintptr_t)(addr - orig);
4377                                    break;
4378                            }
4379
4380                            ASSERT(subr == DIF_SUBR_STRSTR);
4381                            regs[rd] = (uintptr_t)addr;
4382                            break;
4383                    }
4384            }
4385
4386            break;
4387    }
4388
4389    case DIF_SUBR_STRTOK: {
4390            uintptr_t addr = tupregs[0].dttk_value;
4391            uintptr_t tokaddr = tupregs[1].dttk_value;
4392            uint64_t size = state->dts_options[DTRACEOPT_STRSIZE];
4393            uintptr_t limit, toklimit = tokaddr + size;
4394            uint8_t c, tokmap[32];   /* 256 / 8 */
4395            char *dest = (char *)mstate->dtms_scratch_ptr;
4396            int i;
4397
4398            /*
4399             * Check both the token buffer and (later) the input buffer,
4400             * since both could be non-scratch addresses.
4401             */
4402            if (!dtrace_strcanload(tokaddr, size, mstate, vstate)) {
4403                    regs[rd] = NULL;
4404                    break;
4405            }
4406
4407            if (!DTRACE_INSCRATCH(mstate, size)) {
4408                    DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
4409                    regs[rd] = NULL;
4410                    break;
4411            }
4412
4413            if (addr == NULL) {
4414                    /*
4415                     * If the address specified is NULL, we use our saved
4416                     * strtok pointer from the mstate.  Note that this
```



```
4417                     * means that the saved strtok pointer is _only_
4418                     * valid within multiple enablings of the same probe --
4419                     * it behaves like an implicit clause-local variable.
4420                     */
4421                    addr = mstate->dtms_strtok;
4422            } else {
4423                    /*
4424                     * If the user-specified address is non-NULL we must
4425                     * access check it.  This is the only time we have
4426                     * a chance to do so, since this address may reside
4427                     * in the string table of this clause-- future calls
4428                     * (when we fetch addr from mstate->dtms_strtok)
4429                     * would fail this access check.
4430                     */
4431                    if (!dtrace_strcanload(addr, size, mstate, vstate)) {
4432                            regs[rd] = NULL;
4433                            break;
4434                    }
4435            }
4436
4437            /*
4438             * First, zero the token map, and then process the token
4439             * string -- setting a bit in the map for every character
4440             * found in the token string.
4441             */
4442            for (i = 0; i < sizeof (tokmap); i++)
4443                    tokmap[i] = 0;
4444
4445            for (; tokaddr < toklimit; tokaddr++) {
4446                    if ((c = dtrace_load8(tokaddr)) == '\0')
4447                            break;
4448
4449                    ASSERT((c >> 3) < sizeof (tokmap));
4450                    tokmap[c >> 3] |= (1 << (c & 0x7));
4451            }
4452
4453            for (limit = addr + size; addr < limit; addr++) {
4454                    /*
4455                     * We're looking for a character that is _not_ contained
4456                     * in the token string.
4457                     */
4458                    if ((c = dtrace_load8(addr)) == '\0')
4459                            break;
4460
4461                    if (!(tokmap[c >> 3] & (1 << (c & 0x7))))
4462                            break;
4463            }
4464
4465            if (c == '\0') {
4466                    /*
4467                     * We reached the end of the string without finding
4468                     * any character that was not in the token string.
4469                     * We return NULL in this case, and we set the saved
4470                     * address to NULL as well.
4471                     */
4472                    regs[rd] = NULL;
4473                    mstate->dtms_strtok = NULL;
4474                    break;
4475            }
4476
4477            /*
4478             * From here on, we're copying into the destination string.
4479             */
4480            for (i = 0; addr < limit && i < size - 1; addr++) {
4481                    if ((c = dtrace_load8(addr)) == '\0')
4482                            break;
```

```
4484                        if (tokmap[c >> 3] & (1 << (c & 0x7)))
4485                                break;
4486
4487                        ASSERT(i < size);
4488                        dest[i++] = c;
4489                }
4490
4491                ASSERT(i < size);
4492                dest[i] = '\0';
4493                regs[rd] = (uintptr_t)dest;
4494                mstate->dtms_scratch_ptr += size;
4495                mstate->dtms_strtok = addr;
4496                break;
4497        }
4498
4499        case DIF_SUBR_SUBSTR: {
4500                uintptr_t s = tupregs[0].dttk_value;
4501                uint64_t size = state->dts_options[DTRACEOPT_STRSIZE];
4502                char *d = (char *)mstate->dtms_scratch_ptr;
4503                int64_t index = (int64_t)tupregs[1].dttk_value;
4504                int64_t remaining = (int64_t)tupregs[2].dttk_value;
4505                size_t len = dtrace_strlen((char *)s, size);
4506                int64_t i;
4507
4508                if (!dtrace_canload(s, len + 1, mstate, vstate)) {
4509                        regs[rd] = NULL;
4510                        break;
4511                }
4512
4513                if (!DTRACE_INSCRATCH(mstate, size)) {
4514                        DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
4515                        regs[rd] = NULL;
4516                        break;
4517                }
4518
4519                if (nargs <= 2)
4520                        remaining = (int64_t)size;
4521
4522                if (index < 0) {
4523                        index += len;
4524
4525                        if (index < 0 && index + remaining > 0) {
4526                                remaining += index;
4527                                index = 0;
4528                        }
4529                }
4530
4531                if (index >= len || index < 0) {
4532                        remaining = 0;
4533                } else if (remaining < 0) {
4534                        remaining += len - index;
4535                } else if (index + remaining > size) {
4536                        remaining = size - index;
4537                }
4538
4539                for (i = 0; i < remaining; i++) {
4540                        if ((d[i] = dtrace_load8(s + index + i)) == '\0')
4541                                break;
4542                }
4543
4544                d[i] = '\0';
4545
4546                mstate->dtms_scratch_ptr += size;
4547                regs[rd] = (uintptr_t)d;
4548                break;
```



```
4549        }
4550
4551        case DIF_SUBR_JSON: {
4552                uint64_t size = state->dts_options[DTRACEOPT_STRSIZE];
4553                uintptr_t json = tupregs[0].dttk_value;
4554                size_t jsonlen = dtrace_strlen((char *)json, size);
4555                uintptr_t elem = tupregs[1].dttk_value;
4556                size_t elemlen = dtrace_strlen((char *)elem, size);
4557
4558                char *dest = (char *)mstate->dtms_scratch_ptr;
4559                char *elemlist = (char *)mstate->dtms_scratch_ptr + jsonlen + 1;
4560                char *ee = elemlist;
4561                int nelems = 1;
4562                uintptr_t cur;
4563
4564                if (!dtrace_canload(json, jsonlen + 1, mstate, vstate) ||
4565                    !dtrace_canload(elem, elemlen + 1, mstate, vstate)) {
4566                        regs[rd] = NULL;
4567                        break;
4568                }
4569
4570                if (!DTRACE_INSCRATCH(mstate, jsonlen + 1 + elemlen + 1)) {
4571                        DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
4572                        regs[rd] = NULL;
4573                        break;
4574                }
4575
4576                /*
4577                 * Read the element selector and split it up into a packed list
4578                 * of strings.
4579                 */
4580                for (cur = elem; cur < elem + elemlen; cur++) {
4581                        char cc = dtrace_load8(cur);
4582
4583                        if (cur == elem && cc == '[') {
4584                                /*
4585                                 * If the first element selector key is
4586                                 * actually an array index then ignore the
4587                                 * bracket.
4588                                 */
4589                                continue;
4590                        }
4591
4592                        if (cc == ']')
4593                                continue;
4594
4595                        if (cc == '.' || cc == '[') {
4596                                nelems++;
4597                                cc = '\0';
4598                        }
4599
4600                        *ee++ = cc;
4601                }
4602                *ee++ = '\0';
4603
4604                if ((regs[rd] = (uintptr_t)dtrace_json(size, json, elemlist,
4605                    nelems, dest)) != NULL)
4606                        mstate->dtms_scratch_ptr += jsonlen + 1;
4607                break;
4608        }
4609
4610 #endif /* ! codereview */
4611        case DIF_SUBR_TOUPPER:
4612        case DIF_SUBR_TOLOWER: {
4613                uintptr_t s = tupregs[0].dttk_value;
4614                uint64_t size = state->dts_options[DTRACEOPT_STRSIZE];
```

```
4615                 char *dest = (char *)mstate->dtms_scratch_ptr, c;
4616                 size_t len = dtrace_strlen((char *)s, size);
4617                 char lower, upper, convert;
4618                 int64_t i;
4619
4620                 if (subr == DIF_SUBR_TOUPPER) {
4621                         lower = 'a';
4622                         upper = 'z';
4623                         convert = 'A';
4624                 } else {
4625                         lower = 'A';
4626                         upper = 'Z';
4627                         convert = 'a';
4628                 }
4629
4630                 if (!dtrace_canload(s, len + 1, mstate, vstate)) {
4631                         regs[rd] = NULL;
4632                         break;
4633                 }
4634
4635                 if (!DTRACE_INSCRATCH(mstate, size)) {
4636                         DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
4637                         regs[rd] = NULL;
4638                         break;
4639                 }
4640
4641                 for (i = 0; i < size - 1; i++) {
4642                         if ((c = dtrace_load8(s + i)) == '\0')
4643                                 break;
4644
4645                         if (c >= lower && c <= upper)
4646                                 c = convert + (c - lower);
4647
4648                         dest[i] = c;
4649                 }
4650
4651                 ASSERT(i < size);
4652                 dest[i] = '\0';
4653                 regs[rd] = (uintptr_t)dest;
4654                 mstate->dtms_scratch_ptr += size;
4655                 break;
4656         }
4657
4658 case DIF_SUBR_GETMAJOR:
4659 #ifdef _LP64
4660                 regs[rd] = (tupregs[0].dttk_value >> NBITSMINOR64) & MAXMAJ64;
4661 #else
4662                 regs[rd] = (tupregs[0].dttk_value >> NBITSMINOR) & MAXMAJ;
4663 #endif
4664                 break;
4665
4666         case DIF_SUBR_GETMINOR:
4667 #ifdef _LP64
4668                 regs[rd] = tupregs[0].dttk_value & MAXMIN64;
4669 #else
4670                 regs[rd] = tupregs[0].dttk_value & MAXMIN;
4671 #endif
4672                 break;
4673
4674         case DIF_SUBR_DDI_PATHNAME: {
4675                 /*
4676                  * This one is a galactic mess.  We are going to roughly
4677                  * emulate ddi_pathname(), but it's made more complicated
4678                  * by the fact that we (a) want to include the minor name and
4679                  * (b) must proceed iteratively instead of recursively.
4680                  */
```



```
4681                 uintptr_t dest = mstate->dtms_scratch_ptr;
4682                 uint64_t size = state->dts_options[DTRACEOPT_STRSIZE];
4683                 char *start = (char *)dest, *end = start + size - 1;
4684                 uintptr_t daddr = tupregs[0].dttk_value;
4685                 int64_t minor = (int64_t)tupregs[1].dttk_value;
4686                 char *s;
4687                 int i, len, depth = 0;
4688
4689                 /*
4690                  * Due to all the pointer jumping we do and context we must
4691                  * rely upon, we just mandate that the user must have kernel
4692                  * read privileges to use this routine.
4693                  */
4694                 if ((mstate->dtms_access & DTRACE_ACCESS_KERNEL) == 0) {
4695                         *flags |= CPU_DTRACE_KPRIV;
4696                         *illval = daddr;
4697                         regs[rd] = NULL;
4698                 }
4699
4700                 if (!DTRACE_INSCRATCH(mstate, size)) {
4701                         DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
4702                         regs[rd] = NULL;
4703                         break;
4704                 }
4705
4706                 *end = '\0';
4707
4708                 /*
4709                  * We want to have a name for the minor.  In order to do this,
4710                  * we need to walk the minor list from the devinfo.  We want
4711                  * to be sure that we don't infinitely walk a circular list,
4712                  * so we check for circularity by sending a scout pointer
4713                  * ahead two elements for every element that we iterate over;
4714                  * if the list is circular, these will ultimately point to the
4715                  * same element.  You may recognize this little trick as the
4716                  * answer to a stupid interview question -- one that always
4717                  * seems to be asked by those who had to have it laboriously
4718                  * explained to them, and who can't even concisely describe
4719                  * the conditions under which one would be forced to resort to
4720                  * this technique.  Needless to say, those conditions are
4721                  * found here -- and probably only here.  Is this the only use
4722                  * of this infamous trick in shipping, production code?  If it
4723                  * isn't, it probably should be...
4724                  */
4725                 if (minor != -1) {
4726                         uintptr_t maddr = dtrace_loadptr(daddr +
4727                             offsetof(struct dev_info, devi_minor));
4728
4729                         uintptr_t next = offsetof(struct ddi_minor_data, next);
4730                         uintptr_t name = offsetof(struct ddi_minor_data,
4731                             d_minor) + offsetof(struct ddi_minor, name);
4732                         uintptr_t dev = offsetof(struct ddi_minor_data,
4733                             d_minor) + offsetof(struct ddi_minor, dev);
4734                         uintptr_t scout;
4735
4736                         if (maddr != NULL)
4737                                 scout = dtrace_loadptr(maddr + next);
4738
4739                         while (maddr != NULL && !(*flags & CPU_DTRACE_FAULT)) {
4740                                 uint64_t m;
4741 #ifdef _LP64
4742                                 m = dtrace_load64(maddr + dev) & MAXMIN64;
4743 #else
4744                                 m = dtrace_load32(maddr + dev) & MAXMIN;
4745 #endif
4746                                 if (m != minor) {
```

```
4747                                    maddr = dtrace_loadptr(maddr + next);
4749                                    if (scout == NULL)
4750                                            continue;
4752                                    scout = dtrace_loadptr(scout + next);
4754                                    if (scout == NULL)
4755                                            continue;
4757                                    scout = dtrace_loadptr(scout + next);
4759                                    if (scout == NULL)
4760                                            continue;
4762                                    if (scout == maddr) {
4763                                            *flags |= CPU_DTRACE_ILLOP;
4764                                            break;
4765                                    }
4767                                    continue;
4768                            }
4770                            /*
4771                             * We have the minor data.  Now we need to
4772                             * copy the minor's name into the end of the
4773                             * pathname.
4774                             */
4775                            s = (char *)dtrace_loadptr(maddr + name);
4776                            len = dtrace_strlen(s, size);
4778                            if (*flags & CPU_DTRACE_FAULT)
4779                                    break;
4781                            if (len != 0) {
4782                                    if ((end -= (len + 1)) < start)
4783                                            break;
4785                                    *end = ':';
4786                            }
4788                            for (i = 1; i <= len; i++)
4789                                    end[i] = dtrace_load8((uintptr_t)s++);
4790                            break;
4791                    }
4792            }
4794            while (daddr != NULL && !(*flags & CPU_DTRACE_FAULT)) {
4795                    ddi_node_state_t devi_state;
4797                    devi_state = dtrace_load32(daddr +
4798                        offsetof(struct dev_info, devi_node_state));
4800                    if (*flags & CPU_DTRACE_FAULT)
4801                            break;
4803                    if (devi_state >= DS_INITIALIZED) {
4804                            s = (char *)dtrace_loadptr(daddr +
4805                                offsetof(struct dev_info, devi_addr));
4806                            len = dtrace_strlen(s, size);
4808                            if (*flags & CPU_DTRACE_FAULT)
4809                                    break;
4811                            if (len != 0) {
4812                                    if ((end -= (len + 1)) < start)
```



```
4813                                            break;
4815                                    *end = '@';
4816                            }
4818                            for (i = 1; i <= len; i++)
4819                                    end[i] = dtrace_load8((uintptr_t)s++);
4820                    }
4822                    /*
4823                     * Now for the node name...
4824                     */
4825                    s = (char *)dtrace_loadptr(daddr +
4826                        offsetof(struct dev_info, devi_node_name));
4828                    daddr = dtrace_loadptr(daddr +
4829                        offsetof(struct dev_info, devi_parent));
4831                    /*
4832                     * If our parent is NULL (that is, if we're the root
4833                     * node), we're going to use the special path
4834                     * "devices".
4835                     */
4836                    if (daddr == NULL)
4837                            s = "devices";
4839                    len = dtrace_strlen(s, size);
4840                    if (*flags & CPU_DTRACE_FAULT)
4841                            break;
4843                    if ((end -= (len + 1)) < start)
4844                            break;
4846                    for (i = 1; i <= len; i++)
4847                            end[i] = dtrace_load8((uintptr_t)s++);
4848                    *end = '/';
4850                    if (depth++ > dtrace_devdepth_max) {
4851                            *flags |= CPU_DTRACE_ILLOP;
4852                            break;
4853                    }
4854            }
4856            if (end < start)
4857                    DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
4859            if (daddr == NULL) {
4860                    regs[rd] = (uintptr_t)end;
4861                    mstate->dtms_scratch_ptr += size;
4862            }
4864            break;
4865    }
4867    case DIF_SUBR_STRJOIN: {
4868            char *d = (char *)mstate->dtms_scratch_ptr;
4869            uint64_t size = state->dts_options[DTRACEOPT_STRSIZE];
4870            uintptr_t s1 = tupregs[0].dttk_value;
4871            uintptr_t s2 = tupregs[1].dttk_value;
4872            int i = 0;
4874            if (!dtrace_strcanload(s1, size, mstate, vstate) ||
4875                !dtrace_strcanload(s2, size, mstate, vstate)) {
4876                    regs[rd] = NULL;
4877                    break;
4878            }
```

```
4880                 if (!DTRACE_INSCRATCH(mstate, size)) {
4881                         DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
4882                         regs[rd] = NULL;
4883                         break;
4884                 }

4886                 for (;;) {
4887                         if (i >= size) {
4888                                 DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
4889                                 regs[rd] = NULL;
4890                                 break;
4891                         }

4893                         if ((d[i++] = dtrace_load8(s1++)) == '\0') {
4894                                 i--;
4895                                 break;
4896                         }
4897                 }

4899                 for (;;) {
4900                         if (i >= size) {
4901                                 DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
4902                                 regs[rd] = NULL;
4903                                 break;
4904                         }

4906                         if ((d[i++] = dtrace_load8(s2++)) == '\0')
4907                                 break;
4908                 }

4910                 if (i < size) {
4911                         mstate->dtms_scratch_ptr += i;
4912                         regs[rd] = (uintptr_t)d;
4913                 }

4915                 break;
4916         }

4918         case DIF_SUBR_STRTOLL: {
4919                 uintptr_t s = tupregs[0].dttk_value;
4920                 uint64_t size = state->dts_options[DTRACEOPT_STRSIZE];
4921                 int base = 10;

4923                 if (nargs > 1) {
4924                         if ((base = tupregs[1].dttk_value) <= 1 ||
4925                             base > ('z' - 'a' + 1) + ('9' - '0' + 1)) {
4926                                 *flags |= CPU_DTRACE_ILLOP;
4927                                 break;
4928                         }
4929                 }

4931                 if (!dtrace_strcanload(s, size, mstate, vstate)) {
4932                         regs[rd] = INT64_MIN;
4933                         break;
4934                 }

4936                 regs[rd] = dtrace_strtoll((char *)s, base, size);
4937                 break;
4938         }

4940 #endif /* ! codereview */
4941         case DIF_SUBR_LLTOSTR: {
4942                 int64_t i = (int64_t)tupregs[0].dttk_value;
4943                 uint64_t val, digit;
4944                 uint64_t size = 65;     /* enough room for 2^64 in binary */
```



```
4945                 char *end = (char *)mstate->dtms_scratch_ptr + size - 1;
4946                 int base = 10;

4948                 if (nargs > 1) {
4949                         if ((base = tupregs[1].dttk_value) <= 1 ||
4950                             base > ('z' - 'a' + 1) + ('9' - '0' + 1)) {
4951                                 *flags |= CPU_DTRACE_ILLOP;
4952                                 break;
4953                         }
4954                 }

4956                 val = (base == 10 && i < 0) ? i * -1 : i;

4958                 if (!DTRACE_INSCRATCH(mstate, size)) {
4959                         DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
4960                         regs[rd] = NULL;
4961                         break;
4962                 }

4964                 for (*end-- = '\0'; val; val /= base) {
4965                         if ((digit = val % base) <= '9' - '0') {
4966                                 *end-- = '0' + digit;
4967                         } else {
4968                                 *end-- = 'a' + (digit - ('9' - '0') - 1);
4969                         }
4970                 }

4972                 if (i == 0 && base == 16)
4973                         *end-- = '0';

4975                 if (base == 16)
4976                         *end-- = 'x';

4978                 if (i == 0 || base == 8 || base == 16)
4979                         *end-- = '0';

4981                 if (i < 0 && base == 10)
4982                         *end-- = '-';

4984                 regs[rd] = (uintptr_t)end + 1;
4985                 mstate->dtms_scratch_ptr += size;
4986                 break;
4987         }

4989         case DIF_SUBR_HTONS:
4990         case DIF_SUBR_NTOHS:
4991 #ifdef _BIG_ENDIAN
4992                 regs[rd] = (uint16_t)tupregs[0].dttk_value;
4993 #else
4994                 regs[rd] = DT_BSWAP_16((uint16_t)tupregs[0].dttk_value);
4995 #endif
4996                 break;


4999         case DIF_SUBR_HTONL:
5000         case DIF_SUBR_NTOHL:
5001 #ifdef _BIG_ENDIAN
5002                 regs[rd] = (uint32_t)tupregs[0].dttk_value;
5003 #else
5004                 regs[rd] = DT_BSWAP_32((uint32_t)tupregs[0].dttk_value);
5005 #endif
5006                 break;


5009         case DIF_SUBR_HTONLL:
5010         case DIF_SUBR_NTOHLL:
```

```
#ifdef _BIG_ENDIAN
                regs[rd] = (uint64_t)tupregs[0].dttk_value;
#else
                regs[rd] = DT_BSWAP_64((uint64_t)tupregs[0].dttk_value);
#endif
                break;


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

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

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

                /*
                 * The basename and dirname for a zero-length string is
                 * defined to be "."
                 */
                if (len == 0) {
                        len = 1;
                        src = (uintptr_t)".";
                }

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

                if (i >= 0)
                        lastbase = i;

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

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

                /*
                 * Now keep going until we find a non-slash character.  That
```



```
                 * character is the last character in the dirname.
                 */
                for (; i >= 0; i--) {
                        if (dtrace_load8(src + i) != '/')
                                break;
                }

                if (i >= 0)
                        lastdir = i;

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

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

                if (firstbase == -1) {
                        /*
                         * The entire string consists only of a basename
                         * component.  If we're looking for dirname, we need
                         * to change our string to be just "."; if we're
                         * looking for a basename, we'll just set the first
                         * character of the basename to be 0.
                         */
                        if (subr == DIF_SUBR_DIRNAME) {
                                ASSERT(lastdir == -1);
                                src = (uintptr_t)".";
                                lastdir = 0;
                        } else {
                                firstbase = 0;
                        }
                }

                if (subr == DIF_SUBR_DIRNAME) {
                        if (lastdir == -1) {
                                /*
                                 * We know that we have a slash in the name --
                                 * or lastdir would be set to 0, above.  And
                                 * because lastdir is -1, we know that this
                                 * slash must be the first character.  (That
                                 * is, the full string must be of the form
                                 * "/basename".)  In this case, the last
                                 * character of the directory name is 0.
                                 */
                                lastdir = 0;
                        }

                        start = 0;
                        end = lastdir;
                } else {
                        ASSERT(subr == DIF_SUBR_BASENAME);
                        ASSERT(firstbase != -1 && lastbase != -1);
                        start = firstbase;
                        end = lastbase;
                }

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

```
5144                dest[j] = '\0';
5145                regs[rd] = (uintptr_t)dest;
5146                mstate->dtms_scratch_ptr += size;
5147                break;
5148        }

5150        case DIF_SUBR_GETF: {
5151                uintptr_t fd = tupregs[0].dttk_value;
5152                uf_info_t *finfo = &curthread->t_procp->p_user.u_finfo;
5153                file_t *fp;

5155                if (!dtrace_priv_proc(state, mstate)) {
5156                        regs[rd] = NULL;
5157                        break;
5158                }

5160                /*
5161                 * This is safe because fi_nfiles only increases, and the
5162                 * fi_list array is not freed when the array size doubles.
5163                 * (See the comment in flist_grow() for details on the
5164                 * management of the u_finfo structure.)
5165                 */
5166                fp = fd < finfo->fi_nfiles ? finfo->fi_list[fd].uf_file : NULL;

5168                mstate->dtms_getf = fp;
5169                regs[rd] = (uintptr_t)fp;
5170                break;
5171        }

5173        case DIF_SUBR_CLEANPATH: {
5174                char *dest = (char *)mstate->dtms_scratch_ptr, c;
5175                uint64_t size = state->dts_options[DTRACEOPT_STRSIZE];
5176                uintptr_t src = tupregs[0].dttk_value;
5177                int i = 0, j = 0;
5178                zone_t *z;

5180                if (!dtrace_strcanload(src, size, mstate, vstate)) {
5181                        regs[rd] = NULL;
5182                        break;
5183                }

5185                if (!DTRACE_INSCRATCH(mstate, size)) {
5186                        DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
5187                        regs[rd] = NULL;
5188                        break;
5189                }

5191                /*
5192                 * Move forward, loading each character.
5193                 */
5194                do {
5195                        c = dtrace_load8(src + i++);
5196 next:
5197                        if (j + 5 >= size)      /* 5 = strlen("/..c\0") */
5198                                break;

5200                        if (c != '/') {
5201                                dest[j++] = c;
5202                                continue;
5203                        }

5205                        c = dtrace_load8(src + i++);

5207                        if (c == '/') {
5208                                /*
```



```
5209                                 * We have two slashes -- we can just advance
5210                                 * to the next character.
5211                                 */
5212                                goto next;
5213                        }

5215                        if (c != '.') {
5216                                /*
5217                                 * This is not "." and it's not ".." -- we can
5218                                 * just store the "/" and this character and
5219                                 * drive on.
5220                                 */
5221                                dest[j++] = '/';
5222                                dest[j++] = c;
5223                                continue;
5224                        }

5226                        c = dtrace_load8(src + i++);

5228                        if (c == '/') {
5229                                /*
5230                                 * This is a "/./" component.  We're not going
5231                                 * to store anything in the destination buffer;
5232                                 * we're just going to go to the next component.
5233                                 */
5234                                goto next;
5235                        }

5237                        if (c != '.') {
5238                                /*
5239                                 * This is not ".." -- we can just store the
5240                                 * "/." and this character and continue
5241                                 * processing.
5242                                 */
5243                                dest[j++] = '/';
5244                                dest[j++] = '.';
5245                                dest[j++] = c;
5246                                continue;
5247                        }

5249                        c = dtrace_load8(src + i++);

5251                        if (c != '/' && c != '\0') {
5252                                /*
5253                                 * This is not ".." -- it's "..[mumble]".
5254                                 * We'll store the "/.." and this character
5255                                 * and continue processing.
5256                                 */
5257                                dest[j++] = '/';
5258                                dest[j++] = '.';
5259                                dest[j++] = '.';
5260                                dest[j++] = c;
5261                                continue;
5262                        }

5264                        /*
5265                         * This is "/../" or "/..\0".  We need to back up
5266                         * our destination pointer until we find a "/".
5267                         */
5268                        i--;
5269                        while (j != 0 && dest[--j] != '/')
5270                                continue;

5272                        if (c == '\0')
5273                                dest[++j] = '/';
5274                } while (c != '\0');
```

```
5276                dest[j] = '\0';

5278                if (mstate->dtms_getf != NULL &&
5279                    !(mstate->dtms_access & DTRACE_ACCESS_KERNEL) &&
5280                    (z = state->dts_cred.dcr_cred->cr_zone) != kcred->cr_zone) {
5281                        /*
5282                         * If we've done a getf() as a part of this ECB and we
5283                         * don't have kernel access (and we're not in the global
5284                         * zone), check if the path we cleaned up begins with
5285                         * the zone's root path, and trim it off if so.  Note
5286                         * that this is an output cleanliness issue, not a
5287                         * security issue: knowing one's zone root path does
5288                         * not enable privilege escalation.
5289                         */
5290                        if (strstr(dest, z->zone_rootpath) == dest)
5291                                dest += strlen(z->zone_rootpath) - 1;
5292                }

5294                regs[rd] = (uintptr_t)dest;
5295                mstate->dtms_scratch_ptr += size;
5296                break;
5297        }

5299        case DIF_SUBR_INET_NTOA:
5300        case DIF_SUBR_INET_NTOA6:
5301        case DIF_SUBR_INET_NTOP: {
5302                size_t size;
5303                int af, argi, i;
5304                char *base, *end;

5306                if (subr == DIF_SUBR_INET_NTOP) {
5307                        af = (int)tupregs[0].dttk_value;
5308                        argi = 1;
5309                } else {
5310                        af = subr == DIF_SUBR_INET_NTOA ? AF_INET: AF_INET6;
5311                        argi = 0;
5312                }

5314                if (af == AF_INET) {
5315                        ipaddr_t ip4;
5316                        uint8_t *ptr8, val;

5318                        /*
5319                         * Safely load the IPv4 address.
5320                         */
5321                        ip4 = dtrace_load32(tupregs[argi].dttk_value);

5323                        /*
5324                         * Check an IPv4 string will fit in scratch.
5325                         */
5326                        size = INET_ADDRSTRLEN;
5327                        if (!DTRACE_INSCRATCH(mstate, size)) {
5328                                DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
5329                                regs[rd] = NULL;
5330                                break;
5331                        }
5332                        base = (char *)mstate->dtms_scratch_ptr;
5333                        end = (char *)mstate->dtms_scratch_ptr + size - 1;

5335                        /*
5336                         * Stringify as a dotted decimal quad.
5337                         */
5338                        *end-- = '\0';
5339                        ptr8 = (uint8_t *)&ip4;
5340                        for (i = 3; i >= 0; i--) {
```



```
5341                                val = ptr8[i];

5343                                if (val == 0) {
5344                                        *end-- = '0';
5345                                } else {
5346                                        for (; val; val /= 10) {
5347                                                *end-- = '0' + (val % 10);
5348                                        }
5349                                }

5351                                if (i > 0)
5352                                        *end-- = '.';
5353                        }
5354                        ASSERT(end + 1 >= base);

5356                } else if (af == AF_INET6) {
5357                        struct in6_addr ip6;
5358                        int firstzero, tryzero, numzero, v6end;
5359                        uint16_t val;
5360                        const char digits[] = "0123456789abcdef";

5362                        /*
5363                         * Stringify using RFC 1884 convention 2 - 16 bit
5364                         * hexadecimal values with a zero-run compression.
5365                         * Lower case hexadecimal digits are used.
5366                         *      eg, fe80::214:4fff:fe0b:76c8.
5367                         * The IPv4 embedded form is returned for inet_ntop,
5368                         * just the IPv4 string is returned for inet_ntoa6.
5369                         */

5371                        /*
5372                         * Safely load the IPv6 address.
5373                         */
5374                        dtrace_bcopy(
5375                            (void *)(uintptr_t)tupregs[argi].dttk_value,
5376                            (void *)(uintptr_t)&ip6, sizeof (struct in6_addr));

5378                        /*
5379                         * Check an IPv6 string will fit in scratch.
5380                         */
5381                        size = INET6_ADDRSTRLEN;
5382                        if (!DTRACE_INSCRATCH(mstate, size)) {
5383                                DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
5384                                regs[rd] = NULL;
5385                                break;
5386                        }
5387                        base = (char *)mstate->dtms_scratch_ptr;
5388                        end = (char *)mstate->dtms_scratch_ptr + size - 1;
5389                        *end-- = '\0';

5391                        /*
5392                         * Find the longest run of 16 bit zero values
5393                         * for the single allowed zero compression - "::".
5394                         */
5395                        firstzero = -1;
5396                        tryzero = -1;
5397                        numzero = 1;
5398                        for (i = 0; i < sizeof (struct in6_addr); i++) {
5399                                if (ip6._S6_un._S6_u8[i] == 0 &&
5400                                    tryzero == -1 && i % 2 == 0) {
5401                                        tryzero = i;
5402                                        continue;
5403                                }

5405                                if (tryzero != -1 &&
5406                                    (ip6._S6_un._S6_u8[i] != 0 ||
```

```
5407                                    i == sizeof (struct in6_addr) - 1)) {
5409                                        if (i - tryzero <= numzero) {
5410                                                tryzero = -1;
5411                                                continue;
5412                                        }
5414                                        firstzero = tryzero;
5415                                        numzero = i - i % 2 - tryzero;
5416                                        tryzero = -1;
5418                                        if (ip6._S6_un._S6_u8[i] == 0 &&
5419                                            i == sizeof (struct in6_addr) - 1)
5420                                                numzero += 2;
5421                                }
5422                        }
5423                        ASSERT(firstzero + numzero <= sizeof (struct in6_addr));
5425                        /*
5426                         * Check for an IPv4 embedded address.
5427                         */
5428                        v6end = sizeof (struct in6_addr) - 2;
5429                        if (IN6_IS_ADDR_V4MAPPED(&ip6) ||
5430                            IN6_IS_ADDR_V4COMPAT(&ip6)) {
5431                                for (i = sizeof (struct in6_addr) - 1;
5432                                    i >= DTRACE_V4MAPPED_OFFSET; i--) {
5433                                        ASSERT(end >= base);
5435                                        val = ip6._S6_un._S6_u8[i];
5437                                        if (val == 0) {
5438                                                *end-- = '0';
5439                                        } else {
5440                                                for (; val; val /= 10) {
5441                                                        *end-- = '0' + val % 10;
5442                                                }
5443                                        }
5445                                        if (i > DTRACE_V4MAPPED_OFFSET)
5446                                                *end-- = '.';
5447                                }
5449                                if (subr == DIF_SUBR_INET_NTOA6)
5450                                        goto inetout;
5452                                /*
5453                                 * Set v6end to skip the IPv4 address that
5454                                 * we have already stringified.
5455                                 */
5456                                v6end = 10;
5457                        }
5459                        /*
5460                         * Build the IPv6 string by working through the
5461                         * address in reverse.
5462                         */
5463                        for (i = v6end; i >= 0; i -= 2) {
5464                                ASSERT(end >= base);
5466                                if (i == firstzero + numzero - 2) {
5467                                        *end-- = ':';
5468                                        *end-- = ':';
5469                                        i -= numzero - 2;
5470                                        continue;
5471                                }
```



```
5473                                if (i < 14 && i != firstzero - 2)
5474                                        *end-- = ':';
5476                                val = (ip6._S6_un._S6_u8[i] << 8) +
5477                                    ip6._S6_un._S6_u8[i + 1];
5479                                if (val == 0) {
5480                                        *end-- = '0';
5481                                } else {
5482                                        for (; val; val /= 16) {
5483                                                *end-- = digits[val % 16];
5484                                        }
5485                                }
5486                        }
5487                        ASSERT(end + 1 >= base);
5489                } else {
5490                        /*
5491                         * The user didn't use AH_INET or AH_INET6.
5492                         */
5493                        DTRACE_CPUFLAG_SET(CPU_DTRACE_ILLOP);
5494                        regs[rd] = NULL;
5495                        break;
5496                }
5498 inetout:       regs[rd] = (uintptr_t)end + 1;
5499                mstate->dtms_scratch_ptr += size;
5500                break;
5501        }
5503        }
5504 }
5506 /*
5507  * Emulate the execution of DTrace IR instructions specified by the given
5508  * DIF object.  This function is deliberately void of assertions as all of
5509  * the necessary checks are handled by a call to dtrace_difo_validate().
5510  */
5511 static uint64_t
5512 dtrace_dif_emulate(dtrace_difo_t *difo, dtrace_mstate_t *mstate,
5513     dtrace_vstate_t *vstate, dtrace_state_t *state)
5514 {
5515        const dif_instr_t *text = difo->dtdo_buf;
5516        const uint_t textlen = difo->dtdo_len;
5517        const char *strtab = difo->dtdo_strtab;
5518        const uint64_t *inttab = difo->dtdo_inttab;
5520        uint64_t rval = 0;
5521        dtrace_statvar_t *svar;
5522        dtrace_dstate_t *dstate = &vstate->dtvs_dynvars;
5523        dtrace_difv_t *v;
5524        volatile uint16_t *flags = &cpu_core[CPU->cpu_id].cpuc_dtrace_flags;
5525        volatile uintptr_t *illval = &cpu_core[CPU->cpu_id].cpuc_dtrace_illval;
5527        dtrace_key_t tupregs[DIF_DTR_NREGS + 2]; /* +2 for thread and id */
5528        uint64_t regs[DIF_DIR_NREGS];
5529        uint64_t *tmp;
5531        uint8_t cc_n = 0, cc_z = 0, cc_v = 0, cc_c = 0;
5532        int64_t cc_r;
5533        uint_t pc = 0, id, opc;
5534        uint8_t ttop = 0;
5535        dif_instr_t instr;
5536        uint_t r1, r2, rd;
5538        /*
```

```
5539          * We stash the current DIF object into the machine state: we need it
5540          * for subsequent access checking.
5541          */
5542         mstate->dtms_difo = difo;

5544         regs[DIF_REG_R0] = 0;           /* %r0 is fixed at zero */

5546         while (pc < textlen && !(*flags & CPU_DTRACE_FAULT)) {
5547                 opc = pc;

5549                 instr = text[pc++];
5550                 r1 = DIF_INSTR_R1(instr);
5551                 r2 = DIF_INSTR_R2(instr);
5552                 rd = DIF_INSTR_RD(instr);

5554                 switch (DIF_INSTR_OP(instr)) {
5555                 case DIF_OP_OR:
5556                         regs[rd] = regs[r1] | regs[r2];
5557                         break;
5558                 case DIF_OP_XOR:
5559                         regs[rd] = regs[r1] ^ regs[r2];
5560                         break;
5561                 case DIF_OP_AND:
5562                         regs[rd] = regs[r1] & regs[r2];
5563                         break;
5564                 case DIF_OP_SLL:
5565                         regs[rd] = regs[r1] << regs[r2];
5566                         break;
5567                 case DIF_OP_SRL:
5568                         regs[rd] = regs[r1] >> regs[r2];
5569                         break;
5570                 case DIF_OP_SUB:
5571                         regs[rd] = regs[r1] - regs[r2];
5572                         break;
5573                 case DIF_OP_ADD:
5574                         regs[rd] = regs[r1] + regs[r2];
5575                         break;
5576                 case DIF_OP_MUL:
5577                         regs[rd] = regs[r1] * regs[r2];
5578                         break;
5579                 case DIF_OP_SDIV:
5580                         if (regs[r2] == 0) {
5581                                 regs[rd] = 0;
5582                                 *flags |= CPU_DTRACE_DIVZERO;
5583                         } else {
5584                                 regs[rd] = (int64_t)regs[r1] /
5585                                     (int64_t)regs[r2];
5586                         }
5587                         break;

5589                 case DIF_OP_UDIV:
5590                         if (regs[r2] == 0) {
5591                                 regs[rd] = 0;
5592                                 *flags |= CPU_DTRACE_DIVZERO;
5593                         } else {
5594                                 regs[rd] = regs[r1] / regs[r2];
5595                         }
5596                         break;

5598                 case DIF_OP_SREM:
5599                         if (regs[r2] == 0) {
5600                                 regs[rd] = 0;
5601                                 *flags |= CPU_DTRACE_DIVZERO;
5602                         } else {
5603                                 regs[rd] = (int64_t)regs[r1] %
5604                                     (int64_t)regs[r2];
```



```
5605                         }
5606                         break;

5608                 case DIF_OP_UREM:
5609                         if (regs[r2] == 0) {
5610                                 regs[rd] = 0;
5611                                 *flags |= CPU_DTRACE_DIVZERO;
5612                         } else {
5613                                 regs[rd] = regs[r1] % regs[r2];
5614                         }
5615                         break;

5617                 case DIF_OP_NOT:
5618                         regs[rd] = ~regs[r1];
5619                         break;
5620                 case DIF_OP_MOV:
5621                         regs[rd] = regs[r1];
5622                         break;
5623                 case DIF_OP_CMP:
5624                         cc_r = regs[r1] - regs[r2];
5625                         cc_n = cc_r < 0;
5626                         cc_z = cc_r == 0;
5627                         cc_v = 0;
5628                         cc_c = regs[r1] < regs[r2];
5629                         break;
5630                 case DIF_OP_TST:
5631                         cc_n = cc_v = cc_c = 0;
5632                         cc_z = regs[r1] == 0;
5633                         break;
5634                 case DIF_OP_BA:
5635                         pc = DIF_INSTR_LABEL(instr);
5636                         break;
5637                 case DIF_OP_BE:
5638                         if (cc_z)
5639                                 pc = DIF_INSTR_LABEL(instr);
5640                         break;
5641                 case DIF_OP_BNE:
5642                         if (cc_z == 0)
5643                                 pc = DIF_INSTR_LABEL(instr);
5644                         break;
5645                 case DIF_OP_BG:
5646                         if ((cc_z | (cc_n ^ cc_v)) == 0)
5647                                 pc = DIF_INSTR_LABEL(instr);
5648                         break;
5649                 case DIF_OP_BGU:
5650                         if ((cc_c | cc_z) == 0)
5651                                 pc = DIF_INSTR_LABEL(instr);
5652                         break;
5653                 case DIF_OP_BGE:
5654                         if ((cc_n ^ cc_v) == 0)
5655                                 pc = DIF_INSTR_LABEL(instr);
5656                         break;
5657                 case DIF_OP_BGEU:
5658                         if (cc_c == 0)
5659                                 pc = DIF_INSTR_LABEL(instr);
5660                         break;
5661                 case DIF_OP_BL:
5662                         if (cc_n ^ cc_v)
5663                                 pc = DIF_INSTR_LABEL(instr);
5664                         break;
5665                 case DIF_OP_BLU:
5666                         if (cc_c)
5667                                 pc = DIF_INSTR_LABEL(instr);
5668                         break;
5669                 case DIF_OP_BLE:
5670                         if (cc_z | (cc_n ^ cc_v))
```

```
5671                                    pc = DIF_INSTR_LABEL(instr);
5672                            break;
5673                    case DIF_OP_BLEU:
5674                            if (cc_c | cc_z)
5675                                    pc = DIF_INSTR_LABEL(instr);
5676                            break;
5677                    case DIF_OP_RLDSB:
5678                            if (!dtrace_canload(regs[r1], 1, mstate, vstate))
5679                                    break;
5680                            /*FALLTHROUGH*/
5681                    case DIF_OP_LDSB:
5682                            regs[rd] = (int8_t)dtrace_load8(regs[r1]);
5683                            break;
5684                    case DIF_OP_RLDSH:
5685                            if (!dtrace_canload(regs[r1], 2, mstate, vstate))
5686                                    break;
5687                            /*FALLTHROUGH*/
5688                    case DIF_OP_LDSH:
5689                            regs[rd] = (int16_t)dtrace_load16(regs[r1]);
5690                            break;
5691                    case DIF_OP_RLDSW:
5692                            if (!dtrace_canload(regs[r1], 4, mstate, vstate))
5693                                    break;
5694                            /*FALLTHROUGH*/
5695                    case DIF_OP_LDSW:
5696                            regs[rd] = (int32_t)dtrace_load32(regs[r1]);
5697                            break;
5698                    case DIF_OP_RLDUB:
5699                            if (!dtrace_canload(regs[r1], 1, mstate, vstate))
5700                                    break;
5701                            /*FALLTHROUGH*/
5702                    case DIF_OP_LDUB:
5703                            regs[rd] = dtrace_load8(regs[r1]);
5704                            break;
5705                    case DIF_OP_RLDUH:
5706                            if (!dtrace_canload(regs[r1], 2, mstate, vstate))
5707                                    break;
5708                            /*FALLTHROUGH*/
5709                    case DIF_OP_LDUH:
5710                            regs[rd] = dtrace_load16(regs[r1]);
5711                            break;
5712                    case DIF_OP_RLDUW:
5713                            if (!dtrace_canload(regs[r1], 4, mstate, vstate))
5714                                    break;
5715                            /*FALLTHROUGH*/
5716                    case DIF_OP_LDUW:
5717                            regs[rd] = dtrace_load32(regs[r1]);
5718                            break;
5719                    case DIF_OP_RLDX:
5720                            if (!dtrace_canload(regs[r1], 8, mstate, vstate))
5721                                    break;
5722                            /*FALLTHROUGH*/
5723                    case DIF_OP_LDX:
5724                            regs[rd] = dtrace_load64(regs[r1]);
5725                            break;
5726                    case DIF_OP_ULDSB:
5727                            regs[rd] = (int8_t)
5728                                dtrace_fuword8((void *)(uintptr_t)regs[r1]);
5729                            break;
5730                    case DIF_OP_ULDSH:
5731                            regs[rd] = (int16_t)
5732                                dtrace_fuword16((void *)(uintptr_t)regs[r1]);
5733                            break;
5734                    case DIF_OP_ULDSW:
5735                            regs[rd] = (int32_t)
5736                                dtrace_fuword32((void *)(uintptr_t)regs[r1]);
```



```
5737                            break;
5738                    case DIF_OP_ULDUB:
5739                            regs[rd] =
5740                                dtrace_fuword8((void *)(uintptr_t)regs[r1]);
5741                            break;
5742                    case DIF_OP_ULDUH:
5743                            regs[rd] =
5744                                dtrace_fuword16((void *)(uintptr_t)regs[r1]);
5745                            break;
5746                    case DIF_OP_ULDUW:
5747                            regs[rd] =
5748                                dtrace_fuword32((void *)(uintptr_t)regs[r1]);
5749                            break;
5750                    case DIF_OP_ULDX:
5751                            regs[rd] =
5752                                dtrace_fuword64((void *)(uintptr_t)regs[r1]);
5753                            break;
5754                    case DIF_OP_RET:
5755                            rval = regs[rd];
5756                            pc = textlen;
5757                            break;
5758                    case DIF_OP_NOP:
5759                            break;
5760                    case DIF_OP_SETX:
5761                            regs[rd] = inttab[DIF_INSTR_INTEGER(instr)];
5762                            break;
5763                    case DIF_OP_SETS:
5764                            regs[rd] = (uint64_t)(uintptr_t)
5765                                (strtab + DIF_INSTR_STRING(instr));
5766                            break;
5767                    case DIF_OP_SCMP: {
5768                            size_t sz = state->dts_options[DTRACEOPT_STRSIZE];
5769                            uintptr_t s1 = regs[r1];
5770                            uintptr_t s2 = regs[r2];

5772                            if (s1 != NULL &&
5773                                !dtrace_strcanload(s1, sz, mstate, vstate))
5774                                    break;
5775                            if (s2 != NULL &&
5776                                !dtrace_strcanload(s2, sz, mstate, vstate))
5777                                    break;

5779                            cc_r = dtrace_strncmp((char *)s1, (char *)s2, sz);

5781                            cc_n = cc_r < 0;
5782                            cc_z = cc_r == 0;
5783                            cc_v = cc_c = 0;
5784                            break;
5785                    }
5786                    case DIF_OP_LDGA:
5787                            regs[rd] = dtrace_dif_variable(mstate, state,
5788                                r1, regs[r2]);
5789                            break;
5790                    case DIF_OP_LDGS:
5791                            id = DIF_INSTR_VAR(instr);

5793                            if (id >= DIF_VAR_OTHER_UBASE) {
5794                                    uintptr_t a;

5796                                    id -= DIF_VAR_OTHER_UBASE;
5797                                    svar = vstate->dtvs_globals[id];
5798                                    ASSERT(svar != NULL);
5799                                    v = &svar->dtsv_var;

5801                                    if (!(v->dtdv_type.dtdt_flags & DIF_TF_BYREF)) {
5802                                            regs[rd] = svar->dtsv_data;
```

```
5803                                    break;
5804                            }
5805
5806                            a = (uintptr_t)svar->dtsv_data;
5807
5808                            if (*(uint8_t *)a == UINT8_MAX) {
5809                                    /*
5810                                     * If the 0th byte is set to UINT8_MAX
5811                                     * then this is to be treated as a
5812                                     * reference to a NULL variable.
5813                                     */
5814                                    regs[rd] = NULL;
5815                            } else {
5816                                    regs[rd] = a + sizeof (uint64_t);
5817                            }
5818
5819                            break;
5820                    }
5821
5822                    regs[rd] = dtrace_dif_variable(mstate, state, id, 0);
5823                    break;
5824
5825            case DIF_OP_STGS:
5826                    id = DIF_INSTR_VAR(instr);
5827
5828                    ASSERT(id >= DIF_VAR_OTHER_UBASE);
5829                    id -= DIF_VAR_OTHER_UBASE;
5830
5831                    svar = vstate->dtvs_globals[id];
5832                    ASSERT(svar != NULL);
5833                    v = &svar->dtsv_var;
5834
5835                    if (v->dtdv_type.dtdt_flags & DIF_TF_BYREF) {
5836                            uintptr_t a = (uintptr_t)svar->dtsv_data;
5837
5838                            ASSERT(a != NULL);
5839                            ASSERT(svar->dtsv_size != 0);
5840
5841                            if (regs[rd] == NULL) {
5842                                    *(uint8_t *)a = UINT8_MAX;
5843                                    break;
5844                            } else {
5845                                    *(uint8_t *)a = 0;
5846                                    a += sizeof (uint64_t);
5847                            }
5848                            if (!dtrace_vcanload(
5849                                (void *)(uintptr_t)regs[rd], &v->dtdv_type,
5850                                mstate, vstate))
5851                                    break;
5852
5853                            dtrace_vcopy((void *)(uintptr_t)regs[rd],
5854                                (void *)a, &v->dtdv_type);
5855                            break;
5856                    }
5857
5858                    svar->dtsv_data = regs[rd];
5859                    break;
5860
5861            case DIF_OP_LDTA:
5862                    /*
5863                     * There are no DTrace built-in thread-local arrays at
5864                     * present.  This opcode is saved for future work.
5865                     */
5866                    *flags |= CPU_DTRACE_ILLOP;
5867                    regs[rd] = 0;
5868                    break;
```



```
5870            case DIF_OP_LDLS:
5871                    id = DIF_INSTR_VAR(instr);
5872
5873                    if (id < DIF_VAR_OTHER_UBASE) {
5874                            /*
5875                             * For now, this has no meaning.
5876                             */
5877                            regs[rd] = 0;
5878                            break;
5879                    }
5880
5881                    id -= DIF_VAR_OTHER_UBASE;
5882
5883                    ASSERT(id < vstate->dtvs_nlocals);
5884                    ASSERT(vstate->dtvs_locals != NULL);
5885
5886                    svar = vstate->dtvs_locals[id];
5887                    ASSERT(svar != NULL);
5888                    v = &svar->dtsv_var;
5889
5890                    if (v->dtdv_type.dtdt_flags & DIF_TF_BYREF) {
5891                            uintptr_t a = (uintptr_t)svar->dtsv_data;
5892                            size_t sz = v->dtdv_type.dtdt_size;
5893
5894                            sz += sizeof (uint64_t);
5895                            ASSERT(svar->dtsv_size == NCPU * sz);
5896                            a += CPU->cpu_id * sz;
5897
5898                            if (*(uint8_t *)a == UINT8_MAX) {
5899                                    /*
5900                                     * If the 0th byte is set to UINT8_MAX
5901                                     * then this is to be treated as a
5902                                     * reference to a NULL variable.
5903                                     */
5904                                    regs[rd] = NULL;
5905                            } else {
5906                                    regs[rd] = a + sizeof (uint64_t);
5907                            }
5908
5909                            break;
5910                    }
5911
5912                    ASSERT(svar->dtsv_size == NCPU * sizeof (uint64_t));
5913                    tmp = (uint64_t *)(uintptr_t)svar->dtsv_data;
5914                    regs[rd] = tmp[CPU->cpu_id];
5915                    break;
5916
5917            case DIF_OP_STLS:
5918                    id = DIF_INSTR_VAR(instr);
5919
5920                    ASSERT(id >= DIF_VAR_OTHER_UBASE);
5921                    id -= DIF_VAR_OTHER_UBASE;
5922                    ASSERT(id < vstate->dtvs_nlocals);
5923
5924                    ASSERT(vstate->dtvs_locals != NULL);
5925                    svar = vstate->dtvs_locals[id];
5926                    ASSERT(svar != NULL);
5927                    v = &svar->dtsv_var;
5928
5929                    if (v->dtdv_type.dtdt_flags & DIF_TF_BYREF) {
5930                            uintptr_t a = (uintptr_t)svar->dtsv_data;
5931                            size_t sz = v->dtdv_type.dtdt_size;
5932
5933                            sz += sizeof (uint64_t);
5934                            ASSERT(svar->dtsv_size == NCPU * sz);
```

```
5935                                a += CPU->cpu_id * sz;
5936
5937                                if (regs[rd] == NULL) {
5938                                        *(uint8_t *)a = UINT8_MAX;
5939                                        break;
5940                                } else {
5941                                        *(uint8_t *)a = 0;
5942                                        a += sizeof (uint64_t);
5943                                }
5944
5945                                if (!dtrace_vcanload(
5946                                    (void *)(uintptr_t)regs[rd], &v->dtdv_type,
5947                                    mstate, vstate))
5948                                        break;
5949
5950                                dtrace_vcopy((void *)(uintptr_t)regs[rd],
5951                                    (void *)a, &v->dtdv_type);
5952                                break;
5953                        }
5954
5955                        ASSERT(svar->dtsv_size == NCPU * sizeof (uint64_t));
5956                        tmp = (uint64_t *)(uintptr_t)svar->dtsv_data;
5957                        tmp[CPU->cpu_id] = regs[rd];
5958                        break;
5959
5960                case DIF_OP_LDTS: {
5961                        dtrace_dynvar_t *dvar;
5962                        dtrace_key_t *key;
5963
5964                        id = DIF_INSTR_VAR(instr);
5965                        ASSERT(id >= DIF_VAR_OTHER_UBASE);
5966                        id -= DIF_VAR_OTHER_UBASE;
5967                        v = &vstate->dtvs_tlocals[id];
5968
5969                        key = &tupregs[DIF_DTR_NREGS];
5970                        key[0].dttk_value = (uint64_t)id;
5971                        key[0].dttk_size = 0;
5972                        DTRACE_TLS_THRKEY(key[1].dttk_value);
5973                        key[1].dttk_size = 0;
5974
5975                        dvar = dtrace_dynvar(dstate, 2, key,
5976                            sizeof (uint64_t), DTRACE_DYNVAR_NOALLOC,
5977                            mstate, vstate);
5978
5979                        if (dvar == NULL) {
5980                                regs[rd] = 0;
5981                                break;
5982                        }
5983
5984                        if (v->dtdv_type.dtdt_flags & DIF_TF_BYREF) {
5985                                regs[rd] = (uint64_t)(uintptr_t)dvar->dtdv_data;
5986                        } else {
5987                                regs[rd] = *((uint64_t *)dvar->dtdv_data);
5988                        }
5989
5990                        break;
5991                }
5992
5993                case DIF_OP_STTS: {
5994                        dtrace_dynvar_t *dvar;
5995                        dtrace_key_t *key;
5996
5997                        id = DIF_INSTR_VAR(instr);
5998                        ASSERT(id >= DIF_VAR_OTHER_UBASE);
5999                        id -= DIF_VAR_OTHER_UBASE;
```



```
6001                        key = &tupregs[DIF_DTR_NREGS];
6002                        key[0].dttk_value = (uint64_t)id;
6003                        key[0].dttk_size = 0;
6004                        DTRACE_TLS_THRKEY(key[1].dttk_value);
6005                        key[1].dttk_size = 0;
6006                        v = &vstate->dtvs_tlocals[id];
6007
6008                        dvar = dtrace_dynvar(dstate, 2, key,
6009                            v->dtdv_type.dtdt_size > sizeof (uint64_t) ?
6010                            v->dtdv_type.dtdt_size : sizeof (uint64_t),
6011                            regs[rd] ? DTRACE_DYNVAR_ALLOC :
6012                            DTRACE_DYNVAR_DEALLOC, mstate, vstate);
6013
6014                        /*
6015                         * Given that we're storing to thread-local data,
6016                         * we need to flush our predicate cache.
6017                         */
6018                        curthread->t_predcache = NULL;
6019
6020                        if (dvar == NULL)
6021                                break;
6022
6023                        if (v->dtdv_type.dtdt_flags & DIF_TF_BYREF) {
6024                                if (!dtrace_vcanload(
6025                                    (void *)(uintptr_t)regs[rd],
6026                                    &v->dtdv_type, mstate, vstate))
6027                                        break;
6028
6029                                dtrace_vcopy((void *)(uintptr_t)regs[rd],
6030                                    dvar->dtdv_data, &v->dtdv_type);
6031                        } else {
6032                                *((uint64_t *)dvar->dtdv_data) = regs[rd];
6033                        }
6034
6035                        break;
6036                }
6037
6038                case DIF_OP_SRA:
6039                        regs[rd] = (int64_t)regs[r1] >> regs[r2];
6040                        break;
6041
6042                case DIF_OP_CALL:
6043                        dtrace_dif_subr(DIF_INSTR_SUBR(instr), rd,
6044                            regs, tupregs, ttop, mstate, state);
6045                        break;
6046
6047                case DIF_OP_PUSHTR:
6048                        if (ttop == DIF_DTR_NREGS) {
6049                                *flags |= CPU_DTRACE_TUPOFLOW;
6050                                break;
6051                        }
6052
6053                        if (r1 == DIF_TYPE_STRING) {
6054                                /*
6055                                 * If this is a string type and the size is 0,
6056                                 * we'll use the system-wide default string
6057                                 * size.  Note that we are _not_ looking at
6058                                 * the value of the DTRACEOPT_STRSIZE option;
6059                                 * had this been set, we would expect to have
6060                                 * a non-zero size value in the "pushtr".
6061                                 */
6062                                tupregs[ttop].dttk_size =
6063                                    dtrace_strlen((char *)(uintptr_t)regs[rd],
6064                                    regs[r2] ? regs[r2] :
6065                                    dtrace_strsize_default) + 1;
6066                        } else {
```

```
6067                                    tupregs[ttop].dttk_size = regs[r2];
6068                            }
6069
6070                            tupregs[ttop++].dttk_value = regs[rd];
6071                            break;
6072
6073                    case DIF_OP_PUSHTV:
6074                            if (ttop == DIF_DTR_NREGS) {
6075                                    *flags |= CPU_DTRACE_TUPOFLOW;
6076                                    break;
6077                            }
6078
6079                            tupregs[ttop].dttk_value = regs[rd];
6080                            tupregs[ttop++].dttk_size = 0;
6081                            break;
6082
6083                    case DIF_OP_POPTS:
6084                            if (ttop != 0)
6085                                    ttop--;
6086                            break;
6087
6088                    case DIF_OP_FLUSHTS:
6089                            ttop = 0;
6090                            break;
6091
6092                    case DIF_OP_LDGAA:
6093                    case DIF_OP_LDTAA: {
6094                            dtrace_dynvar_t *dvar;
6095                            dtrace_key_t *key = tupregs;
6096                            uint_t nkeys = ttop;
6097
6098                            id = DIF_INSTR_VAR(instr);
6099                            ASSERT(id >= DIF_VAR_OTHER_UBASE);
6100                            id -= DIF_VAR_OTHER_UBASE;
6101
6102                            key[nkeys].dttk_value = (uint64_t)id;
6103                            key[nkeys++].dttk_size = 0;
6104
6105                            if (DIF_INSTR_OP(instr) == DIF_OP_LDTAA) {
6106                                    DTRACE_TLS_THRKEY(key[nkeys].dttk_value);
6107                                    key[nkeys++].dttk_size = 0;
6108                                    v = &vstate->dtvs_tlocals[id];
6109                            } else {
6110                                    v = &vstate->dtvs_globals[id]->dtsv_var;
6111                            }
6112
6113                            dvar = dtrace_dynvar(dstate, nkeys, key,
6114                                v->dtdv_type.dtdt_size > sizeof (uint64_t) ?
6115                                v->dtdv_type.dtdt_size : sizeof (uint64_t),
6116                                DTRACE_DYNVAR_NOALLOC, mstate, vstate);
6117
6118                            if (dvar == NULL) {
6119                                    regs[rd] = 0;
6120                                    break;
6121                            }
6122
6123                            if (v->dtdv_type.dtdt_flags & DIF_TF_BYREF) {
6124                                    regs[rd] = (uint64_t)(uintptr_t)dvar->dtdv_data;
6125                            } else {
6126                                    regs[rd] = *((uint64_t *)dvar->dtdv_data);
6127                            }
6128
6129                            break;
6130                    }
6131
6132                    case DIF_OP_STGAA:
```



```
6133                    case DIF_OP_STTAA: {
6134                            dtrace_dynvar_t *dvar;
6135                            dtrace_key_t *key = tupregs;
6136                            uint_t nkeys = ttop;
6137
6138                            id = DIF_INSTR_VAR(instr);
6139                            ASSERT(id >= DIF_VAR_OTHER_UBASE);
6140                            id -= DIF_VAR_OTHER_UBASE;
6141
6142                            key[nkeys].dttk_value = (uint64_t)id;
6143                            key[nkeys++].dttk_size = 0;
6144
6145                            if (DIF_INSTR_OP(instr) == DIF_OP_STTAA) {
6146                                    DTRACE_TLS_THRKEY(key[nkeys].dttk_value);
6147                                    key[nkeys++].dttk_size = 0;
6148                                    v = &vstate->dtvs_tlocals[id];
6149                            } else {
6150                                    v = &vstate->dtvs_globals[id]->dtsv_var;
6151                            }
6152
6153                            dvar = dtrace_dynvar(dstate, nkeys, key,
6154                                v->dtdv_type.dtdt_size > sizeof (uint64_t) ?
6155                                v->dtdv_type.dtdt_size : sizeof (uint64_t),
6156                                regs[rd] ? DTRACE_DYNVAR_ALLOC :
6157                                DTRACE_DYNVAR_DEALLOC, mstate, vstate);
6158
6159                            if (dvar == NULL)
6160                                    break;
6161
6162                            if (v->dtdv_type.dtdt_flags & DIF_TF_BYREF) {
6163                                    if (!dtrace_vcanload(
6164                                        (void *)(uintptr_t)regs[rd], &v->dtdv_type,
6165                                        mstate, vstate))
6166                                            break;
6167
6168                                    dtrace_vcopy((void *)(uintptr_t)regs[rd],
6169                                        dvar->dtdv_data, &v->dtdv_type);
6170                            } else {
6171                                    *((uint64_t *)dvar->dtdv_data) = regs[rd];
6172                            }
6173
6174                            break;
6175                    }
6176
6177                    case DIF_OP_ALLOCS: {
6178                            uintptr_t ptr = P2ROUNDUP(mstate->dtms_scratch_ptr, 8);
6179                            size_t size = ptr - mstate->dtms_scratch_ptr + regs[r1];
6180
6181                            /*
6182                             * Rounding up the user allocation size could have
6183                             * overflowed large, bogus allocations (like -1ULL) to
6184                             * 0.
6185                             */
6186                            if (size < regs[r1] ||
6187                                !DTRACE_INSCRATCH(mstate, size)) {
6188                                    DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
6189                                    regs[rd] = NULL;
6190                                    break;
6191                            }
6192
6193                            dtrace_bzero((void *) mstate->dtms_scratch_ptr, size);
6194                            mstate->dtms_scratch_ptr += size;
6195                            regs[rd] = ptr;
6196                            break;
6197                    }
```

```
6199                case DIF_OP_COPYS:
6200                        if (!dtrace_canstore(regs[rd], regs[r2],
6201                            mstate, vstate)) {
6202                                *flags |= CPU_DTRACE_BADADDR;
6203                                *illval = regs[rd];
6204                                break;
6205                        }
6206
6207                        if (!dtrace_canload(regs[r1], regs[r2], mstate, vstate))
6208                                break;
6209
6210                        dtrace_bcopy((void *)(uintptr_t)regs[r1],
6211                            (void *)(uintptr_t)regs[rd], (size_t)regs[r2]);
6212                        break;
6213
6214                case DIF_OP_STB:
6215                        if (!dtrace_canstore(regs[rd], 1, mstate, vstate)) {
6216                                *flags |= CPU_DTRACE_BADADDR;
6217                                *illval = regs[rd];
6218                                break;
6219                        }
6220                        *((uint8_t *)(uintptr_t)regs[rd]) = (uint8_t)regs[r1];
6221                        break;
6222
6223                case DIF_OP_STH:
6224                        if (!dtrace_canstore(regs[rd], 2, mstate, vstate)) {
6225                                *flags |= CPU_DTRACE_BADADDR;
6226                                *illval = regs[rd];
6227                                break;
6228                        }
6229                        if (regs[rd] & 1) {
6230                                *flags |= CPU_DTRACE_BADALIGN;
6231                                *illval = regs[rd];
6232                                break;
6233                        }
6234                        *((uint16_t *)(uintptr_t)regs[rd]) = (uint16_t)regs[r1];
6235                        break;
6236
6237                case DIF_OP_STW:
6238                        if (!dtrace_canstore(regs[rd], 4, mstate, vstate)) {
6239                                *flags |= CPU_DTRACE_BADADDR;
6240                                *illval = regs[rd];
6241                                break;
6242                        }
6243                        if (regs[rd] & 3) {
6244                                *flags |= CPU_DTRACE_BADALIGN;
6245                                *illval = regs[rd];
6246                                break;
6247                        }
6248                        *((uint32_t *)(uintptr_t)regs[rd]) = (uint32_t)regs[r1];
6249                        break;
6250
6251                case DIF_OP_STX:
6252                        if (!dtrace_canstore(regs[rd], 8, mstate, vstate)) {
6253                                *flags |= CPU_DTRACE_BADADDR;
6254                                *illval = regs[rd];
6255                                break;
6256                        }
6257                        if (regs[rd] & 7) {
6258                                *flags |= CPU_DTRACE_BADALIGN;
6259                                *illval = regs[rd];
6260                                break;
6261                        }
6262                        *((uint64_t *)(uintptr_t)regs[rd]) = regs[r1];
6263                        break;
6264                }
```



```
6265        }
6266
6267        if (!(*flags & CPU_DTRACE_FAULT))
6268                return (rval);
6269
6270        mstate->dtms_fltoffs = opc * sizeof (dif_instr_t);
6271        mstate->dtms_present |= DTRACE_MSTATE_FLTOFFS;
6272
6273        return (0);
6274 }
6275
6276 static void
6277 dtrace_action_breakpoint(dtrace_ecb_t *ecb)
6278 {
6279        dtrace_probe_t *probe = ecb->dte_probe;
6280        dtrace_provider_t *prov = probe->dtpr_provider;
6281        char c[DTRACE_FULLNAMELEN + 80], *str;
6282        char *msg = "dtrace: breakpoint action at probe ";
6283        char *ecbmsg = " (ecb ";
6284        uintptr_t mask = (0xf << (sizeof (uintptr_t) * NBBY / 4));
6285        uintptr_t val = (uintptr_t)ecb;
6286        int shift = (sizeof (uintptr_t) * NBBY) - 4, i = 0;
6287
6288        if (dtrace_destructive_disallow)
6289                return;
6290
6291        /*
6292         * It's impossible to be taking action on the NULL probe.
6293         */
6294        ASSERT(probe != NULL);
6295
6296        /*
6297         * This is a poor man's (destitute man's?) sprintf():  we want to
6298         * print the provider name, module name, function name and name of
6299         * the probe, along with the hex address of the ECB with the breakpoint
6300         * action -- all of which we must place in the character buffer by
6301         * hand.
6302         */
6303        while (*msg != '\0')
6304                c[i++] = *msg++;
6305
6306        for (str = prov->dtpv_name; *str != '\0'; str++)
6307                c[i++] = *str;
6308        c[i++] = ':';
6309
6310        for (str = probe->dtpr_mod; *str != '\0'; str++)
6311                c[i++] = *str;
6312        c[i++] = ':';
6313
6314        for (str = probe->dtpr_func; *str != '\0'; str++)
6315                c[i++] = *str;
6316        c[i++] = ':';
6317
6318        for (str = probe->dtpr_name; *str != '\0'; str++)
6319                c[i++] = *str;
6320
6321        while (*ecbmsg != '\0')
6322                c[i++] = *ecbmsg++;
6323
6324        while (shift >= 0) {
6325                mask = (uintptr_t)0xf << shift;
6326
6327                if (val >= ((uintptr_t)1 << shift))
6328                        c[i++] = "0123456789abcdef"[(val & mask) >> shift];
6329                shift -= 4;
6330        }
```

```
6332         c[i++] = ')';
6333         c[i] = '\0';

6335         debug_enter(c);
6336 }

6338 static void
6339 dtrace_action_panic(dtrace_ecb_t *ecb)
6340 {
6341         dtrace_probe_t *probe = ecb->dte_probe;

6343         /*
6344          * It's impossible to be taking action on the NULL probe.
6345          */
6346         ASSERT(probe != NULL);

6348         if (dtrace_destructive_disallow)
6349                 return;

6351         if (dtrace_panicked != NULL)
6352                 return;

6354         if (dtrace_casptr(&dtrace_panicked, NULL, curthread) != NULL)
6355                 return;

6357         /*
6358          * We won the right to panic.  (We want to be sure that only one
6359          * thread calls panic() from dtrace_probe(), and that panic() is
6360          * called exactly once.)
6361          */
6362         dtrace_panic("dtrace: panic action at probe %s:%s:%s:%s (ecb %p)",
6363             probe->dtpr_provider->dtpv_name, probe->dtpr_mod,
6364             probe->dtpr_func, probe->dtpr_name, (void *)ecb);
6365 }

6367 static void
6368 dtrace_action_raise(uint64_t sig)
6369 {
6370         if (dtrace_destructive_disallow)
6371                 return;

6373         if (sig >= NSIG) {
6374                 DTRACE_CPUFLAG_SET(CPU_DTRACE_ILLOP);
6375                 return;
6376         }

6378         /*
6379          * raise() has a queue depth of 1 -- we ignore all subsequent
6380          * invocations of the raise() action.
6381          */
6382         if (curthread->t_dtrace_sig == 0)
6383                 curthread->t_dtrace_sig = (uint8_t)sig;

6385         curthread->t_sig_check = 1;
6386         aston(curthread);
6387 }

6389 static void
6390 dtrace_action_stop(void)
6391 {
6392         if (dtrace_destructive_disallow)
6393                 return;

6395         if (!curthread->t_dtrace_stop) {
6396                 curthread->t_dtrace_stop = 1;
```



```
6397                 curthread->t_sig_check = 1;
6398                 aston(curthread);
6399         }
6400 }

6402 static void
6403 dtrace_action_chill(dtrace_mstate_t *mstate, hrtime_t val)
6404 {
6405         hrtime_t now;
6406         volatile uint16_t *flags;
6407         cpu_t *cpu = CPU;

6409         if (dtrace_destructive_disallow)
6410                 return;

6412         flags = (volatile uint16_t *)&cpu_core[cpu->cpu_id].cpuc_dtrace_flags;

6414         now = dtrace_gethrtime();

6416         if (now - cpu->cpu_dtrace_chillmark > dtrace_chill_interval) {
6417                 /*
6418                  * We need to advance the mark to the current time.
6419                  */
6420                 cpu->cpu_dtrace_chillmark = now;
6421                 cpu->cpu_dtrace_chilled = 0;
6422         }

6424         /*
6425          * Now check to see if the requested chill time would take us over
6426          * the maximum amount of time allowed in the chill interval.  (Or
6427          * worse, if the calculation itself induces overflow.)
6428          */
6429         if (cpu->cpu_dtrace_chilled + val > dtrace_chill_max ||
6430             cpu->cpu_dtrace_chilled + val < cpu->cpu_dtrace_chilled) {
6431                 *flags |= CPU_DTRACE_ILLOP;
6432                 return;
6433         }

6435         while (dtrace_gethrtime() - now < val)
6436                 continue;

6438         /*
6439          * Normally, we assure that the value of the variable "timestamp" does
6440          * not change within an ECB.  The presence of chill() represents an
6441          * exception to this rule, however.
6442          */
6443         mstate->dtms_present &= ~DTRACE_MSTATE_TIMESTAMP;
6444         cpu->cpu_dtrace_chilled += val;
6445 }

6447 static void
6448 dtrace_action_ustack(dtrace_mstate_t *mstate, dtrace_state_t *state,
6449     uint64_t *buf, uint64_t arg)
6450 {
6451         int nframes = DTRACE_USTACK_NFRAMES(arg);
6452         int strsize = DTRACE_USTACK_STRSIZE(arg);
6453         uint64_t *pcs = &buf[1], *fps;
6454         char *str = (char *)&pcs[nframes];
6455         int size, offs = 0, i, j;
6456         uintptr_t old = mstate->dtms_scratch_ptr, saved;
6457         uint16_t *flags = &cpu_core[CPU->cpu_id].cpuc_dtrace_flags;
6458         char *sym;

6460         /*
6461          * Should be taking a faster path if string space has not been
6462          * allocated.
```

```
6463          */
6464         ASSERT(strsize != 0);

6466         /*
6467          * We will first allocate some temporary space for the frame pointers.
6468          */
6469         fps = (uint64_t *)P2ROUNDUP(mstate->dtms_scratch_ptr, 8);
6470         size = (uintptr_t)fps - mstate->dtms_scratch_ptr +
6471             (nframes * sizeof (uint64_t));

6473         if (!DTRACE_INSCRATCH(mstate, size)) {
6474                 /*
6475                  * Not enough room for our frame pointers -- need to indicate
6476                  * that we ran out of scratch space.
6477                  */
6478                 DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
6479                 return;
6480         }

6482         mstate->dtms_scratch_ptr += size;
6483         saved = mstate->dtms_scratch_ptr;

6485         /*
6486          * Now get a stack with both program counters and frame pointers.
6487          */
6488         DTRACE_CPUFLAG_SET(CPU_DTRACE_NOFAULT);
6489         dtrace_getufpstack(buf, fps, nframes + 1);
6490         DTRACE_CPUFLAG_CLEAR(CPU_DTRACE_NOFAULT);

6492         /*
6493          * If that faulted, we're cooked.
6494          */
6495         if (*flags & CPU_DTRACE_FAULT)
6496                 goto out;

6498         /*
6499          * Now we want to walk up the stack, calling the USTACK helper.  For
6500          * each iteration, we restore the scratch pointer.
6501          */
6502         for (i = 0; i < nframes; i++) {
6503                 mstate->dtms_scratch_ptr = saved;

6505                 if (offs >= strsize)
6506                         break;

6508                 sym = (char *)(uintptr_t)dtrace_helper(
6509                     DTRACE_HELPER_ACTION_USTACK,
6510                     mstate, state, pcs[i], fps[i]);

6512                 /*
6513                  * If we faulted while running the helper, we're going to
6514                  * clear the fault and null out the corresponding string.
6515                  */
6516                 if (*flags & CPU_DTRACE_FAULT) {
6517                         *flags &= ~CPU_DTRACE_FAULT;
6518                         str[offs++] = '\0';
6519                         continue;
6520                 }

6522                 if (sym == NULL) {
6523                         str[offs++] = '\0';
6524                         continue;
6525                 }

6527                 DTRACE_CPUFLAG_SET(CPU_DTRACE_NOFAULT);
```



```
6529                 /*
6530                  * Now copy in the string that the helper returned to us.
6531                  */
6532                 for (j = 0; offs + j < strsize; j++) {
6533                         if ((str[offs + j] = sym[j]) == '\0')
6534                                 break;
6535                 }

6537                 DTRACE_CPUFLAG_CLEAR(CPU_DTRACE_NOFAULT);

6539                 offs += j + 1;
6540         }

6542         if (offs >= strsize) {
6543                 /*
6544                  * If we didn't have room for all of the strings, we don't
6545                  * abort processing -- this needn't be a fatal error -- but we
6546                  * still want to increment a counter (dts_stkstroverflows) to
6547                  * allow this condition to be warned about.  (If this is from
6548                  * a jstack() action, it is easily tuned via jstackstrsize.)
6549                  */
6550                 dtrace_error(&state->dts_stkstroverflows);
6551         }

6553         while (offs < strsize)
6554                 str[offs++] = '\0';

6556 out:
6557         mstate->dtms_scratch_ptr = old;
6558 }

6560 /*
6561  * If you're looking for the epicenter of DTrace, you just found it.  This
6562  * is the function called by the provider to fire a probe -- from which all
6563  * subsequent probe-context DTrace activity emanates.
6564  */
6565 void
6566 dtrace_probe(dtrace_id_t id, uintptr_t arg0, uintptr_t arg1,
6567     uintptr_t arg2, uintptr_t arg3, uintptr_t arg4)
6568 {
6569         processorid_t cpuid;
6570         dtrace_icookie_t cookie;
6571         dtrace_probe_t *probe;
6572         dtrace_mstate_t mstate;
6573         dtrace_ecb_t *ecb;
6574         dtrace_action_t *act;
6575         intptr_t offs;
6576         size_t size;
6577         int vtime, onintr;
6578         volatile uint16_t *flags;
6579         hrtime_t now, end;

6581         /*
6582          * Kick out immediately if this CPU is still being born (in which case
6583          * curthread will be set to -1) or the current thread can't allow
6584          * probes in its current context.
6585          */
6586         if (((uintptr_t)curthread & 1) || (curthread->t_flag & T_DONTDTRACE))
6587                 return;

6589         cookie = dtrace_interrupt_disable();
6590         probe = dtrace_probes[id - 1];
6591         cpuid = CPU->cpu_id;
6592         onintr = CPU_ON_INTR(CPU);

6594         CPU->cpu_dtrace_probes++;
```

```
6596         if (!onintr && probe->dtpr_predcache != DTRACE_CACHEIDNONE &&
6597             probe->dtpr_predcache == curthread->t_predcache) {
6598                 /*
6599                  * We have hit in the predicate cache; we know that
6600                  * this predicate would evaluate to be false.
6601                  */
6602                 dtrace_interrupt_enable(cookie);
6603                 return;
6604         }
6605
6606         if (panic_quiesce) {
6607                 /*
6608                  * We don't trace anything if we're panicking.
6609                  */
6610                 dtrace_interrupt_enable(cookie);
6611                 return;
6612         }
6613
6614         now = dtrace_gethrtime();
6615         vtime = dtrace_vtime_references != 0;
6616
6617         if (vtime && curthread->t_dtrace_start)
6618                 curthread->t_dtrace_vtime += now - curthread->t_dtrace_start;
6619
6620         mstate.dtms_difo = NULL;
6621         mstate.dtms_probe = probe;
6622         mstate.dtms_strtok = NULL;
6623         mstate.dtms_arg[0] = arg0;
6624         mstate.dtms_arg[1] = arg1;
6625         mstate.dtms_arg[2] = arg2;
6626         mstate.dtms_arg[3] = arg3;
6627         mstate.dtms_arg[4] = arg4;
6628
6629         flags = (volatile uint16_t *)&cpu_core[cpuid].cpuc_dtrace_flags;
6630
6631         for (ecb = probe->dtpr_ecb; ecb != NULL; ecb = ecb->dte_next) {
6632                 dtrace_predicate_t *pred = ecb->dte_predicate;
6633                 dtrace_state_t *state = ecb->dte_state;
6634                 dtrace_buffer_t *buf = &state->dts_buffer[cpuid];
6635                 dtrace_buffer_t *aggbuf = &state->dts_aggbuffer[cpuid];
6636                 dtrace_vstate_t *vstate = &state->dts_vstate;
6637                 dtrace_provider_t *prov = probe->dtpr_provider;
6638                 uint64_t tracememsize = 0;
6639                 int committed = 0;
6640                 caddr_t tomax;
6641
6642                 /*
6643                  * A little subtlety with the following (seemingly innocuous)
6644                  * declaration of the automatic 'val':  by looking at the
6645                  * code, you might think that it could be declared in the
6646                  * action processing loop, below.  (That is, it's only used in
6647                  * the action processing loop.)  However, it must be declared
6648                  * out of that scope because in the case of DIF expression
6649                  * arguments to aggregating actions, one iteration of the
6650                  * action loop will use the last iteration's value.
6651                  */
6652 #ifdef lint
6653                 uint64_t val = 0;
6654 #else
6655                 uint64_t val;
6656 #endif
6657
6658                 mstate.dtms_present = DTRACE_MSTATE_ARGS | DTRACE_MSTATE_PROBE;
6659                 mstate.dtms_access = DTRACE_ACCESS_ARGS | DTRACE_ACCESS_PROC;
6660                 mstate.dtms_getf = NULL;
```



```
6662                 *flags &= ~CPU_DTRACE_ERROR;
6663
6664                 if (prov == dtrace_provider) {
6665                         /*
6666                          * If dtrace itself is the provider of this probe,
6667                          * we're only going to continue processing the ECB if
6668                          * arg0 (the dtrace_state_t) is equal to the ECB's
6669                          * creating state.  (This prevents disjoint consumers
6670                          * from seeing one another's metaprobes.)
6671                          */
6672                         if (arg0 != (uint64_t)(uintptr_t)state)
6673                                 continue;
6674                 }
6675
6676                 if (state->dts_activity != DTRACE_ACTIVITY_ACTIVE) {
6677                         /*
6678                          * We're not currently active.  If our provider isn't
6679                          * the dtrace pseudo provider, we're not interested.
6680                          */
6681                         if (prov != dtrace_provider)
6682                                 continue;
6683
6684                         /*
6685                          * Now we must further check if we are in the BEGIN
6686                          * probe.  If we are, we will only continue processing
6687                          * if we're still in WARMUP -- if one BEGIN enabling
6688                          * has invoked the exit() action, we don't want to
6689                          * evaluate subsequent BEGIN enablings.
6690                          */
6691                         if (probe->dtpr_id == dtrace_probeid_begin &&
6692                             state->dts_activity != DTRACE_ACTIVITY_WARMUP) {
6693                                 ASSERT(state->dts_activity ==
6694                                     DTRACE_ACTIVITY_DRAINING);
6695                                 continue;
6696                         }
6697                 }
6698
6699                 if (ecb->dte_cond && !dtrace_priv_probe(state, &mstate, ecb))
6700                         continue;
6701
6702                 if (now - state->dts_alive > dtrace_deadman_timeout) {
6703                         /*
6704                          * We seem to be dead.  Unless we (a) have kernel
6705                          * destructive permissions (b) have explicitly enabled
6706                          * destructive actions and (c) destructive actions have
6707                          * not been disabled, we're going to transition into
6708                          * the KILLED state, from which no further processing
6709                          * on this state will be performed.
6710                          */
6711                         if (!dtrace_priv_kernel_destructive(state) ||
6712                             !state->dts_cred.dcr_destructive ||
6713                             dtrace_destructive_disallow) {
6714                                 void *activity = &state->dts_activity;
6715                                 dtrace_activity_t current;
6716
6717                                 do {
6718                                         current = state->dts_activity;
6719                                 } while (dtrace_cas32(activity, current,
6720                                     DTRACE_ACTIVITY_KILLED) != current);
6721
6722                                 continue;
6723                         }
6724                 }
6725
6726                 if ((offs = dtrace_buffer_reserve(buf, ecb->dte_needed,
```

```
6727                    ecb->dte_alignment, state, &mstate)) < 0)
6728                        continue;
6729
6730                tomax = buf->dtb_tomax;
6731                ASSERT(tomax != NULL);
6732
6733                if (ecb->dte_size != 0) {
6734                        dtrace_rechdr_t dtrh;
6735                        if (!(mstate.dtms_present & DTRACE_MSTATE_TIMESTAMP)) {
6736                                mstate.dtms_timestamp = dtrace_gethrtime();
6737                                mstate.dtms_present |= DTRACE_MSTATE_TIMESTAMP;
6738                        }
6739                        ASSERT3U(ecb->dte_size, >=, sizeof (dtrace_rechdr_t));
6740                        dtrh.dtrh_epid = ecb->dte_epid;
6741                        DTRACE_RECORD_STORE_TIMESTAMP(&dtrh,
6742                            mstate.dtms_timestamp);
6743                        *((dtrace_rechdr_t *)(tomax + offs)) = dtrh;
6744                }
6745
6746                mstate.dtms_epid = ecb->dte_epid;
6747                mstate.dtms_present |= DTRACE_MSTATE_EPID;
6748
6749                if (state->dts_cred.dcr_visible & DTRACE_CRV_KERNEL)
6750                        mstate.dtms_access |= DTRACE_ACCESS_KERNEL;
6751
6752                if (pred != NULL) {
6753                        dtrace_difo_t *dp = pred->dtp_difo;
6754                        int rval;
6755
6756                        rval = dtrace_dif_emulate(dp, &mstate, vstate, state);
6757
6758                        if (!(*flags & CPU_DTRACE_ERROR) && !rval) {
6759                                dtrace_cacheid_t cid = probe->dtpr_predcache;
6760
6761                                if (cid != DTRACE_CACHEIDNONE && !onintr) {
6762                                        /*
6763                                         * Update the predicate cache...
6764                                         */
6765                                        ASSERT(cid == pred->dtp_cacheid);
6766                                        curthread->t_predcache = cid;
6767                                }
6768
6769                                continue;
6770                        }
6771                }
6772
6773                for (act = ecb->dte_action; !(*flags & CPU_DTRACE_ERROR) &&
6774                    act != NULL; act = act->dta_next) {
6775                        size_t valoffs;
6776                        dtrace_difo_t *dp;
6777                        dtrace_recdesc_t *rec = &act->dta_rec;
6778
6779                        size = rec->dtrd_size;
6780                        valoffs = offs + rec->dtrd_offset;
6781
6782                        if (DTRACEACT_ISAGG(act->dta_kind)) {
6783                                uint64_t v = 0xbad;
6784                                dtrace_aggregation_t *agg;
6785
6786                                agg = (dtrace_aggregation_t *)act;
6787
6788                                if ((dp = act->dta_difo) != NULL)
6789                                        v = dtrace_dif_emulate(dp,
6790                                            &mstate, vstate, state);
6791
6792                                if (*flags & CPU_DTRACE_ERROR)
```



```
6793                                        continue;
6794
6795                                /*
6796                                 * Note that we always pass the expression
6797                                 * value from the previous iteration of the
6798                                 * action loop.  This value will only be used
6799                                 * if there is an expression argument to the
6800                                 * aggregating action, denoted by the
6801                                 * dtag_hasarg field.
6802                                 */
6803                                dtrace_aggregate(agg, buf,
6804                                    offs, aggbuf, v, val);
6805                                continue;
6806                        }
6807
6808                        switch (act->dta_kind) {
6809                        case DTRACEACT_STOP:
6810                                if (dtrace_priv_proc_destructive(state,
6811                                    &mstate))
6812                                        dtrace_action_stop();
6813                                continue;
6814
6815                        case DTRACEACT_BREAKPOINT:
6816                                if (dtrace_priv_kernel_destructive(state))
6817                                        dtrace_action_breakpoint(ecb);
6818                                continue;
6819
6820                        case DTRACEACT_PANIC:
6821                                if (dtrace_priv_kernel_destructive(state))
6822                                        dtrace_action_panic(ecb);
6823                                continue;
6824
6825                        case DTRACEACT_STACK:
6826                                if (!dtrace_priv_kernel(state))
6827                                        continue;
6828
6829                                dtrace_getpcstack((pc_t *)(tomax + valoffs),
6830                                    size / sizeof (pc_t), probe->dtpr_aframes,
6831                                    DTRACE_ANCHORED(probe) ? NULL :
6832                                    (uint32_t *)arg0);
6833
6834                                continue;
6835
6836                        case DTRACEACT_JSTACK:
6837                        case DTRACEACT_USTACK:
6838                                if (!dtrace_priv_proc(state, &mstate))
6839                                        continue;
6840
6841                                /*
6842                                 * See comment in DIF_VAR_PID.
6843                                 */
6844                                if (DTRACE_ANCHORED(mstate.dtms_probe) &&
6845                                    CPU_ON_INTR(CPU)) {
6846                                        int depth = DTRACE_USTACK_NFRAMES(
6847                                            rec->dtrd_arg) + 1;
6848
6849                                        dtrace_bzero((void *)(tomax + valoffs),
6850                                            DTRACE_USTACK_STRSIZE(rec->dtrd_arg)
6851                                            + depth * sizeof (uint64_t));
6852
6853                                        continue;
6854                                }
6855
6856                                if (DTRACE_USTACK_STRSIZE(rec->dtrd_arg) != 0 &&
6857                                    curproc->p_dtrace_helpers != NULL) {
6858                                        /*
```

```
6859                                     * This is the slow path -- we have
6860                                     * allocated string space, and we're
6861                                     * getting the stack of a process that
6862                                     * has helpers.  Call into a separate
6863                                     * routine to perform this processing.
6864                                     */
6865                                    dtrace_action_ustack(&mstate, state,
6866                                        (uint64_t *)(tomax + valoffs),
6867                                        rec->dtrd_arg);
6868                                    continue;
6869                            }

6871                            /*
6872                             * Clear the string space, since there's no
6873                             * helper to do it for us.
6874                             */
6875                            if (DTRACE_USTACK_STRSIZE(rec->dtrd_arg) != 0) {
6876                                    int depth = DTRACE_USTACK_NFRAMES(
6877                                        rec->dtrd_arg);
6878                                    size_t strsize = DTRACE_USTACK_STRSIZE(
6879                                        rec->dtrd_arg);
6880                                    uint64_t *buf = (uint64_t *)(tomax +
6881                                        valoffs);
6882                                    void *strspace = &buf[depth + 1];

6884                                    dtrace_bzero(strspace,
6885                                        MIN(depth, strsize));
6886                            }

6888                            DTRACE_CPUFLAG_SET(CPU_DTRACE_NOFAULT);
6889                            dtrace_getupcstack((uint64_t *)
6890                                (tomax + valoffs),
6891                                DTRACE_USTACK_NFRAMES(rec->dtrd_arg) + 1);
6892                            DTRACE_CPUFLAG_CLEAR(CPU_DTRACE_NOFAULT);
6893                            continue;

6895                    default:
6896                            break;
6897                    }

6899                    dp = act->dta_difo;
6900                    ASSERT(dp != NULL);

6902                    val = dtrace_dif_emulate(dp, &mstate, vstate, state);

6904                    if (*flags & CPU_DTRACE_ERROR)
6905                            continue;

6907                    switch (act->dta_kind) {
6908                    case DTRACEACT_SPECULATE: {
6909                            dtrace_rechdr_t *dtrh;

6911                            ASSERT(buf == &state->dts_buffer[cpuid]);
6912                            buf = dtrace_speculation_buffer(state,
6913                                cpuid, val);

6915                            if (buf == NULL) {
6916                                    *flags |= CPU_DTRACE_DROP;
6917                                    continue;
6918                            }

6920                            offs = dtrace_buffer_reserve(buf,
6921                                ecb->dte_needed, ecb->dte_alignment,
6922                                state, NULL);

6924                            if (offs < 0) {
```



```
6925                                    *flags |= CPU_DTRACE_DROP;
6926                                    continue;
6927                            }

6929                            tomax = buf->dtb_tomax;
6930                            ASSERT(tomax != NULL);

6932                            if (ecb->dte_size == 0)
6933                                    continue;

6935                            ASSERT3U(ecb->dte_size, >=,
6936                                sizeof (dtrace_rechdr_t));
6937                            dtrh = ((void *)(tomax + offs));
6938                            dtrh->dtrh_epid = ecb->dte_epid;
6939                            /*
6940                             * When the speculation is committed, all of
6941                             * the records in the speculative buffer will
6942                             * have their timestamps set to the commit
6943                             * time.  Until then, it is set to a sentinel
6944                             * value, for debugability.
6945                             */
6946                            DTRACE_RECORD_STORE_TIMESTAMP(dtrh, UINT64_MAX);
6947                            continue;
6948                    }

6950                    case DTRACEACT_CHILL:
6951                            if (dtrace_priv_kernel_destructive(state))
6952                                    dtrace_action_chill(&mstate, val);
6953                            continue;

6955                    case DTRACEACT_RAISE:
6956                            if (dtrace_priv_proc_destructive(state,
6957                                &mstate))
6958                                    dtrace_action_raise(val);
6959                            continue;

6961                    case DTRACEACT_COMMIT:
6962                            ASSERT(!committed);

6964                            /*
6965                             * We need to commit our buffer state.
6966                             */
6967                            if (ecb->dte_size)
6968                                    buf->dtb_offset = offs + ecb->dte_size;
6969                            buf = &state->dts_buffer[cpuid];
6970                            dtrace_speculation_commit(state, cpuid, val);
6971                            committed = 1;
6972                            continue;

6974                    case DTRACEACT_DISCARD:
6975                            dtrace_speculation_discard(state, cpuid, val);
6976                            continue;

6978                    case DTRACEACT_DIFEXPR:
6979                    case DTRACEACT_LIBACT:
6980                    case DTRACEACT_PRINTF:
6981                    case DTRACEACT_PRINTA:
6982                    case DTRACEACT_SYSTEM:
6983                    case DTRACEACT_FREOPEN:
6984                    case DTRACEACT_TRACEMEM:
6985                            break;

6987                    case DTRACEACT_TRACEMEM_DYNSIZE:
6988                            tracememsize = val;
6989                            break;
```

```
6991                        case DTRACEACT_SYM:
6992                        case DTRACEACT_MOD:
6993                                if (!dtrace_priv_kernel(state))
6994                                        continue;
6995                                break;
6996
6997                        case DTRACEACT_USYM:
6998                        case DTRACEACT_UMOD:
6999                        case DTRACEACT_UADDR: {
7000                                struct pid *pid = curthread->t_procp->p_pidp;
7001
7002                                if (!dtrace_priv_proc(state, &mstate))
7003                                        continue;
7004
7005                                DTRACE_STORE(uint64_t, tomax,
7006                                    valoffs, (uint64_t)pid->pid_id);
7007                                DTRACE_STORE(uint64_t, tomax,
7008                                    valoffs + sizeof (uint64_t), val);
7009
7010                                continue;
7011                        }
7012
7013                        case DTRACEACT_EXIT: {
7014                                /*
7015                                 * For the exit action, we are going to attempt
7016                                 * to atomically set our activity to be
7017                                 * draining.  If this fails (either because
7018                                 * another CPU has beat us to the exit action,
7019                                 * or because our current activity is something
7020                                 * other than ACTIVE or WARMUP), we will
7021                                 * continue.  This assures that the exit action
7022                                 * can be successfully recorded at most once
7023                                 * when we're in the ACTIVE state.  If we're
7024                                 * encountering the exit() action while in
7025                                 * COOLDOWN, however, we want to honor the new
7026                                 * status code.  (We know that we're the only
7027                                 * thread in COOLDOWN, so there is no race.)
7028                                 */
7029                                void *activity = &state->dts_activity;
7030                                dtrace_activity_t current = state->dts_activity;
7031
7032                                if (current == DTRACE_ACTIVITY_COOLDOWN)
7033                                        break;
7034
7035                                if (current != DTRACE_ACTIVITY_WARMUP)
7036                                        current = DTRACE_ACTIVITY_ACTIVE;
7037
7038                                if (dtrace_cas32(activity, current,
7039                                    DTRACE_ACTIVITY_DRAINING) != current) {
7040                                        *flags |= CPU_DTRACE_DROP;
7041                                        continue;
7042                                }
7043
7044                                break;
7045                        }
7046
7047                        default:
7048                                ASSERT(0);
7049                        }
7050
7051                        if (dp->dtdo_rtype.dtdt_flags & DIF_TF_BYREF) {
7052                                uintptr_t end = valoffs + size;
7053
7054                                if (tracememsize != 0 &&
7055                                    valoffs + tracememsize < end) {
7056                                        end = valoffs + tracememsize;
```



```
7057                                        tracememsize = 0;
7058                                }
7059
7060                                if (!dtrace_vcanload((void *)(uintptr_t)val,
7061                                    &dp->dtdo_rtype, &mstate, vstate))
7062                                        continue;
7063
7064                                /*
7065                                 * If this is a string, we're going to only
7066                                 * load until we find the zero byte -- after
7067                                 * which we'll store zero bytes.
7068                                 */
7069                                if (dp->dtdo_rtype.dtdt_kind ==
7070                                    DIF_TYPE_STRING) {
7071                                        char c = '\0' + 1;
7072                                        int intuple = act->dta_intuple;
7073                                        size_t s;
7074
7075                                        for (s = 0; s < size; s++) {
7076                                                if (c != '\0')
7077                                                        c = dtrace_load8(val++);
7078
7079                                                DTRACE_STORE(uint8_t, tomax,
7080                                                    valoffs++, c);
7081
7082                                                if (c == '\0' && intuple)
7083                                                        break;
7084                                        }
7085
7086                                        continue;
7087                                }
7088
7089                                while (valoffs < end) {
7090                                        DTRACE_STORE(uint8_t, tomax, valoffs++,
7091                                            dtrace_load8(val++));
7092                                }
7093
7094                                continue;
7095                        }
7096
7097                        switch (size) {
7098                        case 0:
7099                                break;
7100
7101                        case sizeof (uint8_t):
7102                                DTRACE_STORE(uint8_t, tomax, valoffs, val);
7103                                break;
7104                        case sizeof (uint16_t):
7105                                DTRACE_STORE(uint16_t, tomax, valoffs, val);
7106                                break;
7107                        case sizeof (uint32_t):
7108                                DTRACE_STORE(uint32_t, tomax, valoffs, val);
7109                                break;
7110                        case sizeof (uint64_t):
7111                                DTRACE_STORE(uint64_t, tomax, valoffs, val);
7112                                break;
7113                        default:
7114                                /*
7115                                 * Any other size should have been returned by
7116                                 * reference, not by value.
7117                                 */
7118                                ASSERT(0);
7119                                break;
7120                        }
7121                }
```

```
7123                if (*flags & CPU_DTRACE_DROP)
7124                        continue;

7126                if (*flags & CPU_DTRACE_FAULT) {
7127                        int ndx;
7128                        dtrace_action_t *err;

7130                        buf->dtb_errors++;

7132                        if (probe->dtpr_id == dtrace_probeid_error) {
7133                                /*
7134                                 * There's nothing we can do -- we had an
7135                                 * error on the error probe.  We bump an
7136                                 * error counter to at least indicate that
7137                                 * this condition happened.
7138                                 */
7139                                dtrace_error(&state->dts_dblerrors);
7140                                continue;
7141                        }

7143                        if (vtime) {
7144                                /*
7145                                 * Before recursing on dtrace_probe(), we
7146                                 * need to explicitly clear out our start
7147                                 * time to prevent it from being accumulated
7148                                 * into t_dtrace_vtime.
7149                                 */
7150                                curthread->t_dtrace_start = 0;
7151                        }

7153                        /*
7154                         * Iterate over the actions to figure out which action
7155                         * we were processing when we experienced the error.
7156                         * Note that act points _past_ the faulting action; if
7157                         * act is ecb->dte_action, the fault was in the
7158                         * predicate, if it's ecb->dte_action->dta_next it's
7159                         * in action #1, and so on.
7160                         */
7161                        for (err = ecb->dte_action, ndx = 0;
7162                            err != act; err = err->dta_next, ndx++)
7163                                continue;

7165                        dtrace_probe_error(state, ecb->dte_epid, ndx,
7166                            (mstate.dtms_present & DTRACE_MSTATE_FLTOFFS) ?
7167                            mstate.dtms_fltoffs : -1, DTRACE_FLAGS2FLT(*flags),
7168                            cpu_core[cpuid].cpuc_dtrace_illval);

7170                        continue;
7171                }

7173                if (!committed)
7174                        buf->dtb_offset = offs + ecb->dte_size;
7175        }

7177        end = dtrace_gethrtime();
7178        if (vtime)
7179                curthread->t_dtrace_start = end;

7181        CPU->cpu_dtrace_nsec += end - now;

7183        dtrace_interrupt_enable(cookie);
7184 }

7186 /*
7187  * DTrace Probe Hashing Functions
7188  *
```



```
7189  * The functions in this section (and indeed, the functions in remaining
7190  * sections) are not _called_ from probe context.  (Any exceptions to this are
7191  * marked with a "Note:".)  Rather, they are called from elsewhere in the
7192  * DTrace framework to look-up probes in, add probes to and remove probes from
7193  * the DTrace probe hashes.  (Each probe is hashed by each element of the
7194  * probe tuple -- allowing for fast lookups, regardless of what was
7195  * specified.)
7196  */
7197 static uint_t
7198 dtrace_hash_str(char *p)
7199 {
7200         unsigned int g;
7201         uint_t hval = 0;

7203         while (*p) {
7204                 hval = (hval << 4) + *p++;
7205                 if ((g = (hval & 0xf0000000)) != 0)
7206                         hval ^= g >> 24;
7207                 hval &= ~g;
7208         }
7209         return (hval);
7210 }

7212 static dtrace_hash_t *
7213 dtrace_hash_create(uintptr_t stroffs, uintptr_t nextoffs, uintptr_t prevoffs)
7214 {
7215         dtrace_hash_t *hash = kmem_zalloc(sizeof (dtrace_hash_t), KM_SLEEP);

7217         hash->dth_stroffs = stroffs;
7218         hash->dth_nextoffs = nextoffs;
7219         hash->dth_prevoffs = prevoffs;

7221         hash->dth_size = 1;
7222         hash->dth_mask = hash->dth_size - 1;

7224         hash->dth_tab = kmem_zalloc(hash->dth_size *
7225             sizeof (dtrace_hashbucket_t *), KM_SLEEP);

7227         return (hash);
7228 }

7230 static void
7231 dtrace_hash_destroy(dtrace_hash_t *hash)
7232 {
7233 #ifdef DEBUG
7234         int i;

7236         for (i = 0; i < hash->dth_size; i++)
7237                 ASSERT(hash->dth_tab[i] == NULL);
7238 #endif

7240         kmem_free(hash->dth_tab,
7241             hash->dth_size * sizeof (dtrace_hashbucket_t *));
7242         kmem_free(hash, sizeof (dtrace_hash_t));
7243 }

7245 static void
7246 dtrace_hash_resize(dtrace_hash_t *hash)
7247 {
7248         int size = hash->dth_size, i, ndx;
7249         int new_size = hash->dth_size << 1;
7250         int new_mask = new_size - 1;
7251         dtrace_hashbucket_t **new_tab, *bucket, *next;

7253         ASSERT((new_size & new_mask) == 0);
```

```
7255         new_tab = kmem_zalloc(new_size * sizeof (void *), KM_SLEEP);

7257         for (i = 0; i < size; i++) {
7258                 for (bucket = hash->dth_tab[i]; bucket != NULL; bucket = next) {
7259                         dtrace_probe_t *probe = bucket->dthb_chain;

7261                         ASSERT(probe != NULL);
7262                         ndx = DTRACE_HASHSTR(hash, probe) & new_mask;

7264                         next = bucket->dthb_next;
7265                         bucket->dthb_next = new_tab[ndx];
7266                         new_tab[ndx] = bucket;
7267                 }
7268         }

7270         kmem_free(hash->dth_tab, hash->dth_size * sizeof (void *));
7271         hash->dth_tab = new_tab;
7272         hash->dth_size = new_size;
7273         hash->dth_mask = new_mask;
7274 }

7276 static void
7277 dtrace_hash_add(dtrace_hash_t *hash, dtrace_probe_t *new)
7278 {
7279         int hashval = DTRACE_HASHSTR(hash, new);
7280         int ndx = hashval & hash->dth_mask;
7281         dtrace_hashbucket_t *bucket = hash->dth_tab[ndx];
7282         dtrace_probe_t **nextp, **prevp;

7284         for (; bucket != NULL; bucket = bucket->dthb_next) {
7285                 if (DTRACE_HASHEQ(hash, bucket->dthb_chain, new))
7286                         goto add;
7287         }

7289         if ((hash->dth_nbuckets >> 1) > hash->dth_size) {
7290                 dtrace_hash_resize(hash);
7291                 dtrace_hash_add(hash, new);
7292                 return;
7293         }

7295         bucket = kmem_zalloc(sizeof (dtrace_hashbucket_t), KM_SLEEP);
7296         bucket->dthb_next = hash->dth_tab[ndx];
7297         hash->dth_tab[ndx] = bucket;
7298         hash->dth_nbuckets++;

7300 add:
7301         nextp = DTRACE_HASHNEXT(hash, new);
7302         ASSERT(*nextp == NULL && *(DTRACE_HASHPREV(hash, new)) == NULL);
7303         *nextp = bucket->dthb_chain;

7305         if (bucket->dthb_chain != NULL) {
7306                 prevp = DTRACE_HASHPREV(hash, bucket->dthb_chain);
7307                 ASSERT(*prevp == NULL);
7308                 *prevp = new;
7309         }

7311         bucket->dthb_chain = new;
7312         bucket->dthb_len++;
7313 }

7315 static dtrace_probe_t *
7316 dtrace_hash_lookup(dtrace_hash_t *hash, dtrace_probe_t *template)
7317 {
7318         int hashval = DTRACE_HASHSTR(hash, template);
7319         int ndx = hashval & hash->dth_mask;
7320         dtrace_hashbucket_t *bucket = hash->dth_tab[ndx];
```



```
7322         for (; bucket != NULL; bucket = bucket->dthb_next) {
7323                 if (DTRACE_HASHEQ(hash, bucket->dthb_chain, template))
7324                         return (bucket->dthb_chain);
7325         }

7327         return (NULL);
7328 }

7330 static int
7331 dtrace_hash_collisions(dtrace_hash_t *hash, dtrace_probe_t *template)
7332 {
7333         int hashval = DTRACE_HASHSTR(hash, template);
7334         int ndx = hashval & hash->dth_mask;
7335         dtrace_hashbucket_t *bucket = hash->dth_tab[ndx];

7337         for (; bucket != NULL; bucket = bucket->dthb_next) {
7338                 if (DTRACE_HASHEQ(hash, bucket->dthb_chain, template))
7339                         return (bucket->dthb_len);
7340         }

7342         return (NULL);
7343 }

7345 static void
7346 dtrace_hash_remove(dtrace_hash_t *hash, dtrace_probe_t *probe)
7347 {
7348         int ndx = DTRACE_HASHSTR(hash, probe) & hash->dth_mask;
7349         dtrace_hashbucket_t *bucket = hash->dth_tab[ndx];

7351         dtrace_probe_t **prevp = DTRACE_HASHPREV(hash, probe);
7352         dtrace_probe_t **nextp = DTRACE_HASHNEXT(hash, probe);

7354         /*
7355          * Find the bucket that we're removing this probe from.
7356          */
7357         for (; bucket != NULL; bucket = bucket->dthb_next) {
7358                 if (DTRACE_HASHEQ(hash, bucket->dthb_chain, probe))
7359                         break;
7360         }

7362         ASSERT(bucket != NULL);

7364         if (*prevp == NULL) {
7365                 if (*nextp == NULL) {
7366                         /*
7367                          * The removed probe was the only probe on this
7368                          * bucket; we need to remove the bucket.
7369                          */
7370                         dtrace_hashbucket_t *b = hash->dth_tab[ndx];

7372                         ASSERT(bucket->dthb_chain == probe);
7373                         ASSERT(b != NULL);

7375                         if (b == bucket) {
7376                                 hash->dth_tab[ndx] = bucket->dthb_next;
7377                         } else {
7378                                 while (b->dthb_next != bucket)
7379                                         b = b->dthb_next;
7380                                 b->dthb_next = bucket->dthb_next;
7381                         }

7383                         ASSERT(hash->dth_nbuckets > 0);
7384                         hash->dth_nbuckets--;
7385                         kmem_free(bucket, sizeof (dtrace_hashbucket_t));
7386                         return;
```

```
7387                }

7389                bucket->dthb_chain = *nextp;
7390        } else {
7391                *(DTRACE_HASHNEXT(hash, *prevp)) = *nextp;
7392        }

7394        if (*nextp != NULL)
7395                *(DTRACE_HASHPREV(hash, *nextp)) = *prevp;
7396 }

7398 /*
7399  * DTrace Utility Functions
7400  *
7401  * These are random utility functions that are _not_ called from probe context.
7402  */
7403 static int
7404 dtrace_badattr(const dtrace_attribute_t *a)
7405 {
7406        return (a->dtat_name > DTRACE_STABILITY_MAX ||
7407            a->dtat_data > DTRACE_STABILITY_MAX ||
7408            a->dtat_class > DTRACE_CLASS_MAX);
7409 }

7411 /*
7412  * Return a duplicate copy of a string.  If the specified string is NULL,
7413  * this function returns a zero-length string.
7414  */
7415 static char *
7416 dtrace_strdup(const char *str)
7417 {
7418        char *new = kmem_zalloc((str != NULL ? strlen(str) : 0) + 1, KM_SLEEP);

7420        if (str != NULL)
7421                (void) strcpy(new, str);

7423        return (new);
7424 }

7426 #define DTRACE_ISALPHA(c)       \
7427        (((c) >= 'a' && (c) <= 'z') || ((c) >= 'A' && (c) <= 'Z'))

7429 static int
7430 dtrace_badname(const char *s)
7431 {
7432        char c;

7434        if (s == NULL || (c = *s++) == '\0')
7435                return (0);

7437        if (!DTRACE_ISALPHA(c) && c != '-' && c != '_' && c != '.')
7438                return (1);

7440        while ((c = *s++) != '\0') {
7441                if (!DTRACE_ISALPHA(c) && (c < '0' || c > '9') &&
7442                    c != '-' && c != '_' && c != '.' && c != '`')
7443                        return (1);
7444        }

7446        return (0);
7447 }

7449 static void
7450 dtrace_cred2priv(cred_t *cr, uint32_t *privp, uid_t *uidp, zoneid_t *zoneidp)
7451 {
7452        uint32_t priv;
```



```
7454        if (cr == NULL || PRIV_POLICY_ONLY(cr, PRIV_ALL, B_FALSE)) {
7455                /*
7456                 * For DTRACE_PRIV_ALL, the uid and zoneid don't matter.
7457                 */
7458                priv = DTRACE_PRIV_ALL;
7459        } else {
7460                *uidp = crgetuid(cr);
7461                *zoneidp = crgetzoneid(cr);

7463                priv = 0;
7464                if (PRIV_POLICY_ONLY(cr, PRIV_DTRACE_KERNEL, B_FALSE))
7465                        priv |= DTRACE_PRIV_KERNEL | DTRACE_PRIV_USER;
7466                else if (PRIV_POLICY_ONLY(cr, PRIV_DTRACE_USER, B_FALSE))
7467                        priv |= DTRACE_PRIV_USER;
7468                if (PRIV_POLICY_ONLY(cr, PRIV_DTRACE_PROC, B_FALSE))
7469                        priv |= DTRACE_PRIV_PROC;
7470                if (PRIV_POLICY_ONLY(cr, PRIV_PROC_OWNER, B_FALSE))
7471                        priv |= DTRACE_PRIV_OWNER;
7472                if (PRIV_POLICY_ONLY(cr, PRIV_PROC_ZONE, B_FALSE))
7473                        priv |= DTRACE_PRIV_ZONEOWNER;
7474        }

7476        *privp = priv;
7477 }

7479 #ifdef DTRACE_ERRDEBUG
7480 static void
7481 dtrace_errdebug(const char *str)
7482 {
7483        int hval = dtrace_hash_str((char *)str) % DTRACE_ERRHASHSZ;
7484        int occupied = 0;

7486        mutex_enter(&dtrace_errlock);
7487        dtrace_errlast = str;
7488        dtrace_errthread = curthread;

7490        while (occupied++ < DTRACE_ERRHASHSZ) {
7491                if (dtrace_errhash[hval].dter_msg == str) {
7492                        dtrace_errhash[hval].dter_count++;
7493                        goto out;
7494                }

7496                if (dtrace_errhash[hval].dter_msg != NULL) {
7497                        hval = (hval + 1) % DTRACE_ERRHASHSZ;
7498                        continue;
7499                }

7501                dtrace_errhash[hval].dter_msg = str;
7502                dtrace_errhash[hval].dter_count = 1;
7503                goto out;
7504        }

7506        panic("dtrace: undersized error hash");
7507 out:
7508        mutex_exit(&dtrace_errlock);
7509 }
7510 #endif

7512 /*
7513  * DTrace Matching Functions
7514  *
7515  * These functions are used to match groups of probes, given some elements of
7516  * a probe tuple, or some globbed expressions for elements of a probe tuple.
7517  */
7518 static int
```

```
7519 dtrace_match_priv(const dtrace_probe_t *prp, uint32_t priv, uid_t uid,
7520     zoneid_t zoneid)
7521 {
7522         if (priv != DTRACE_PRIV_ALL) {
7523                 uint32_t ppriv = prp->dtpr_provider->dtpv_priv.dtpp_flags;
7524                 uint32_t match = priv & ppriv;
7525
7526                 /*
7527                  * No PRIV_DTRACE_* privileges...
7528                  */
7529                 if ((priv & (DTRACE_PRIV_PROC | DTRACE_PRIV_USER |
7530                     DTRACE_PRIV_KERNEL)) == 0)
7531                         return (0);
7532
7533                 /*
7534                  * No matching bits, but there were bits to match...
7535                  */
7536                 if (match == 0 && ppriv != 0)
7537                         return (0);
7538
7539                 /*
7540                  * Need to have permissions to the process, but don't...
7541                  */
7542                 if (((ppriv & ~match) & DTRACE_PRIV_OWNER) != 0 &&
7543                     uid != prp->dtpr_provider->dtpv_priv.dtpp_uid) {
7544                         return (0);
7545                 }
7546
7547                 /*
7548                  * Need to be in the same zone unless we possess the
7549                  * privilege to examine all zones.
7550                  */
7551                 if (((ppriv & ~match) & DTRACE_PRIV_ZONEOWNER) != 0 &&
7552                     zoneid != prp->dtpr_provider->dtpv_priv.dtpp_zoneid) {
7553                         return (0);
7554                 }
7555         }
7556
7557         return (1);
7558 }
7559
7560 /*
7561  * dtrace_match_probe compares a dtrace_probe_t to a pre-compiled key, which
7562  * consists of input pattern strings and an ops-vector to evaluate them.
7563  * This function returns >0 for match, 0 for no match, and <0 for error.
7564  */
7565 static int
7566 dtrace_match_probe(const dtrace_probe_t *prp, const dtrace_probekey_t *pkp,
7567     uint32_t priv, uid_t uid, zoneid_t zoneid)
7568 {
7569         dtrace_provider_t *pvp = prp->dtpr_provider;
7570         int rv;
7571
7572         if (pvp->dtpv_defunct)
7573                 return (0);
7574
7575         if ((rv = pkp->dtpk_pmatch(pvp->dtpv_name, pkp->dtpk_prov, 0)) <= 0)
7576                 return (rv);
7577
7578         if ((rv = pkp->dtpk_mmatch(prp->dtpr_mod, pkp->dtpk_mod, 0)) <= 0)
7579                 return (rv);
7580
7581         if ((rv = pkp->dtpk_fmatch(prp->dtpr_func, pkp->dtpk_func, 0)) <= 0)
7582                 return (rv);
7583
7584         if ((rv = pkp->dtpk_nmatch(prp->dtpr_name, pkp->dtpk_name, 0)) <= 0)
```



```
7585                 return (rv);
7586
7587         if (dtrace_match_priv(prp, priv, uid, zoneid) == 0)
7588                 return (0);
7589
7590         return (rv);
7591 }
7592
7593 /*
7594  * dtrace_match_glob() is a safe kernel implementation of the gmatch(3GEN)
7595  * interface for matching a glob pattern 'p' to an input string 's'.  Unlike
7596  * libc's version, the kernel version only applies to 8-bit ASCII strings.
7597  * In addition, all of the recursion cases except for '*' matching have been
7598  * unwound.  For '*', we still implement recursive evaluation, but a depth
7599  * counter is maintained and matching is aborted if we recurse too deep.
7600  * The function returns 0 if no match, >0 if match, and <0 if recursion error.
7601  */
7602 static int
7603 dtrace_match_glob(const char *s, const char *p, int depth)
7604 {
7605         const char *olds;
7606         char s1, c;
7607         int gs;
7608
7609         if (depth > DTRACE_PROBEKEY_MAXDEPTH)
7610                 return (-1);
7611
7612         if (s == NULL)
7613                 s = ""; /* treat NULL as empty string */
7614
7615 top:
7616         olds = s;
7617         s1 = *s++;
7618
7619         if (p == NULL)
7620                 return (0);
7621
7622         if ((c = *p++) == '\0')
7623                 return (s1 == '\0');
7624
7625         switch (c) {
7626         case '[': {
7627                 int ok = 0, notflag = 0;
7628                 char lc = '\0';
7629
7630                 if (s1 == '\0')
7631                         return (0);
7632
7633                 if (*p == '!') {
7634                         notflag = 1;
7635                         p++;
7636                 }
7637
7638                 if ((c = *p++) == '\0')
7639                         return (0);
7640
7641                 do {
7642                         if (c == '-' && lc != '\0' && *p != ']') {
7643                                 if ((c = *p++) == '\0')
7644                                         return (0);
7645                                 if (c == '\\' && (c = *p++) == '\0')
7646                                         return (0);
7647
7648                                 if (notflag) {
7649                                         if (s1 < lc || s1 > c)
7650                                                 ok++;
```

```
7651                                    else
7652                                            return (0);
7653                            } else if (lc <= s1 && s1 <= c)
7654                                    ok++;
7655
7656                    } else if (c == '\\' && (c = *p++) == '\0')
7657                            return (0);
7658
7659                    lc = c; /* save left-hand 'c' for next iteration */
7660
7661                    if (notflag) {
7662                            if (s1 != c)
7663                                    ok++;
7664                            else
7665                                    return (0);
7666                    } else if (s1 == c)
7667                            ok++;
7668
7669                    if ((c = *p++) == '\0')
7670                            return (0);
7671
7672            } while (c != ']');
7673
7674            if (ok)
7675                    goto top;
7676
7677            return (0);
7678    }
7679
7680    case '\\':
7681            if ((c = *p++) == '\0')
7682                    return (0);
7683            /*FALLTHRU*/
7684
7685    default:
7686            if (c != s1)
7687                    return (0);
7688            /*FALLTHRU*/
7689
7690    case '?':
7691            if (s1 != '\0')
7692                    goto top;
7693            return (0);
7694
7695    case '*':
7696            while (*p == '*')
7697                    p++; /* consecutive *'s are identical to a single one */
7698
7699            if (*p == '\0')
7700                    return (1);
7701
7702            for (s = olds; *s != '\0'; s++) {
7703                    if ((gs = dtrace_match_glob(s, p, depth + 1)) != 0)
7704                            return (gs);
7705            }
7706
7707            return (0);
7708    }
7709 }
7710
7711 /*ARGSUSED*/
7712 static int
7713 dtrace_match_string(const char *s, const char *p, int depth)
7714 {
7715         return (s != NULL && strcmp(s, p) == 0);
7716 }
```



```
7717
7718 /*ARGSUSED*/
7719 static int
7720 dtrace_match_nul(const char *s, const char *p, int depth)
7721 {
7722         return (1); /* always match the empty pattern */
7723 }
7724
7725 /*ARGSUSED*/
7726 static int
7727 dtrace_match_nonzero(const char *s, const char *p, int depth)
7728 {
7729         return (s != NULL && s[0] != '\0');
7730 }
7731
7732 static int
7733 dtrace_match(const dtrace_probekey_t *pkp, uint32_t priv, uid_t uid,
7734     zoneid_t zoneid, int (*matched)(dtrace_probe_t *, void *), void *arg)
7735 {
7736         dtrace_probe_t template, *probe;
7737         dtrace_hash_t *hash = NULL;
7738         int len, rc, best = INT_MAX, nmatched = 0;
7739         dtrace_id_t i;
7740
7741         ASSERT(MUTEX_HELD(&dtrace_lock));
7742
7743         /*
7744          * If the probe ID is specified in the key, just lookup by ID and
7745          * invoke the match callback once if a matching probe is found.
7746          */
7747         if (pkp->dtpk_id != DTRACE_IDNONE) {
7748                 if ((probe = dtrace_probe_lookup_id(pkp->dtpk_id)) != NULL &&
7749                     dtrace_match_probe(probe, pkp, priv, uid, zoneid) > 0) {
7750                         if ((*matched)(probe, arg) == DTRACE_MATCH_FAIL)
7751                                 return (DTRACE_MATCH_FAIL);
7752                         nmatched++;
7753                 }
7754                 return (nmatched);
7755         }
7756
7757         template.dtpr_mod = (char *)pkp->dtpk_mod;
7758         template.dtpr_func = (char *)pkp->dtpk_func;
7759         template.dtpr_name = (char *)pkp->dtpk_name;
7760
7761         /*
7762          * We want to find the most distinct of the module name, function
7763          * name, and name.  So for each one that is not a glob pattern or
7764          * empty string, we perform a lookup in the corresponding hash and
7765          * use the hash table with the fewest collisions to do our search.
7766          */
7767         if (pkp->dtpk_mmatch == &dtrace_match_string &&
7768             (len = dtrace_hash_collisions(dtrace_bymod, &template)) < best) {
7769                 best = len;
7770                 hash = dtrace_bymod;
7771         }
7772
7773         if (pkp->dtpk_fmatch == &dtrace_match_string &&
7774             (len = dtrace_hash_collisions(dtrace_byfunc, &template)) < best) {
7775                 best = len;
7776                 hash = dtrace_byfunc;
7777         }
7778
7779         if (pkp->dtpk_nmatch == &dtrace_match_string &&
7780             (len = dtrace_hash_collisions(dtrace_byname, &template)) < best) {
7781                 best = len;
7782                 hash = dtrace_byname;
```

```
7783         }

7785         /*
7786          * If we did not select a hash table, iterate over every probe and
7787          * invoke our callback for each one that matches our input probe key.
7788          */
7789         if (hash == NULL) {
7790                 for (i = 0; i < dtrace_nprobes; i++) {
7791                         if ((probe = dtrace_probes[i]) == NULL ||
7792                             dtrace_match_probe(probe, pkp, priv, uid,
7793                             zoneid) <= 0)
7794                                 continue;

7796                         nmatched++;

7798                         if ((rc = (*matched)(probe, arg)) !=
7799                             DTRACE_MATCH_NEXT) {
7800                                 if (rc == DTRACE_MATCH_FAIL)
7801                                         return (DTRACE_MATCH_FAIL);
7802                                 break;
7803                         }
7804                 }

7806                 return (nmatched);
7807         }

7809         /*
7810          * If we selected a hash table, iterate over each probe of the same key
7811          * name and invoke the callback for every probe that matches the other
7812          * attributes of our input probe key.
7813          */
7814         for (probe = dtrace_hash_lookup(hash, &template); probe != NULL;
7815             probe = *(DTRACE_HASHNEXT(hash, probe))) {

7817                 if (dtrace_match_probe(probe, pkp, priv, uid, zoneid) <= 0)
7818                         continue;

7820                 nmatched++;

7822                 if ((rc = (*matched)(probe, arg)) != DTRACE_MATCH_NEXT) {
7823                         if (rc == DTRACE_MATCH_FAIL)
7824                                 return (DTRACE_MATCH_FAIL);
7825                         break;
7826                 }
7827         }

7829         return (nmatched);
7830 }

7832 /*
7833  * Return the function pointer dtrace_probecmp() should use to compare the
7834  * specified pattern with a string.  For NULL or empty patterns, we select
7835  * dtrace_match_nul().  For glob pattern strings, we use dtrace_match_glob().
7836  * For non-empty non-glob strings, we use dtrace_match_string().
7837  */
7838 static dtrace_probekey_f *
7839 dtrace_probekey_func(const char *p)
7840 {
7841         char c;

7843         if (p == NULL || *p == '\0')
7844                 return (&dtrace_match_nul);

7846         while ((c = *p++) != '\0') {
7847                 if (c == '[' || c == '?' || c == '*' || c == '\\')
7848                         return (&dtrace_match_glob);
```



```
7849         }

7851         return (&dtrace_match_string);
7852 }

7854 /*
7855  * Build a probe comparison key for use with dtrace_match_probe() from the
7856  * given probe description.  By convention, a null key only matches anchored
7857  * probes: if each field is the empty string, reset dtpk_fmatch to
7858  * dtrace_match_nonzero().
7859  */
7860 static void
7861 dtrace_probekey(const dtrace_probedesc_t *pdp, dtrace_probekey_t *pkp)
7862 {
7863         pkp->dtpk_prov = pdp->dtpd_provider;
7864         pkp->dtpk_pmatch = dtrace_probekey_func(pdp->dtpd_provider);

7866         pkp->dtpk_mod = pdp->dtpd_mod;
7867         pkp->dtpk_mmatch = dtrace_probekey_func(pdp->dtpd_mod);

7869         pkp->dtpk_func = pdp->dtpd_func;
7870         pkp->dtpk_fmatch = dtrace_probekey_func(pdp->dtpd_func);

7872         pkp->dtpk_name = pdp->dtpd_name;
7873         pkp->dtpk_nmatch = dtrace_probekey_func(pdp->dtpd_name);

7875         pkp->dtpk_id = pdp->dtpd_id;

7877         if (pkp->dtpk_id == DTRACE_IDNONE &&
7878             pkp->dtpk_pmatch == &dtrace_match_nul &&
7879             pkp->dtpk_mmatch == &dtrace_match_nul &&
7880             pkp->dtpk_fmatch == &dtrace_match_nul &&
7881             pkp->dtpk_nmatch == &dtrace_match_nul)
7882                 pkp->dtpk_fmatch = &dtrace_match_nonzero;
7883 }

7885 /*
7886  * DTrace Provider-to-Framework API Functions
7887  *
7888  * These functions implement much of the Provider-to-Framework API, as
7889  * described in <sys/dtrace.h>.  The parts of the API not in this section are
7890  * the functions in the API for probe management (found below), and
7891  * dtrace_probe() itself (found above).
7892  */

7894 /*
7895  * Register the calling provider with the DTrace framework.  This should
7896  * generally be called by DTrace providers in their attach(9E) entry point.
7897  */
7898 int
7899 dtrace_register(const char *name, const dtrace_pattr_t *pap, uint32_t priv,
7900     cred_t *cr, const dtrace_pops_t *pops, void *arg, dtrace_provider_id_t *idp)
7901 {
7902         dtrace_provider_t *provider;

7904         if (name == NULL || pap == NULL || pops == NULL || idp == NULL) {
7905                 cmn_err(CE_WARN, "failed to register provider '%s': invalid "
7906                     "arguments", name ? name : "<NULL>");
7907                 return (EINVAL);
7908         }

7910         if (name[0] == '\0' || dtrace_badname(name)) {
7911                 cmn_err(CE_WARN, "failed to register provider '%s': invalid "
7912                     "provider name", name);
7913                 return (EINVAL);
7914         }
```

```
7916         if ((pops->dtps_provide == NULL && pops->dtps_provide_module == NULL) ||
7917             pops->dtps_enable == NULL || pops->dtps_disable == NULL ||
7918             pops->dtps_destroy == NULL ||
7919             ((pops->dtps_resume == NULL) != (pops->dtps_suspend == NULL))) {
7920                 cmn_err(CE_WARN, "failed to register provider '%s': invalid "
7921                     "provider ops", name);
7922                 return (EINVAL);
7923         }

7925         if (dtrace_badattr(&pap->dtpa_provider) ||
7926             dtrace_badattr(&pap->dtpa_mod) ||
7927             dtrace_badattr(&pap->dtpa_func) ||
7928             dtrace_badattr(&pap->dtpa_name) ||
7929             dtrace_badattr(&pap->dtpa_args)) {
7930                 cmn_err(CE_WARN, "failed to register provider '%s': invalid "
7931                     "provider attributes", name);
7932                 return (EINVAL);
7933         }

7935         if (priv & ~DTRACE_PRIV_ALL) {
7936                 cmn_err(CE_WARN, "failed to register provider '%s': invalid "
7937                     "privilege attributes", name);
7938                 return (EINVAL);
7939         }

7941         if ((priv & DTRACE_PRIV_KERNEL) &&
7942             (priv & (DTRACE_PRIV_USER | DTRACE_PRIV_OWNER)) &&
7943             pops->dtps_mode == NULL) {
7944                 cmn_err(CE_WARN, "failed to register provider '%s': need "
7945                     "dtps_mode() op for given privilege attributes", name);
7946                 return (EINVAL);
7947         }

7949         provider = kmem_zalloc(sizeof (dtrace_provider_t), KM_SLEEP);
7950         provider->dtpv_name = kmem_alloc(strlen(name) + 1, KM_SLEEP);
7951         (void) strcpy(provider->dtpv_name, name);

7953         provider->dtpv_attr = *pap;
7954         provider->dtpv_priv.dtpp_flags = priv;
7955         if (cr != NULL) {
7956                 provider->dtpv_priv.dtpp_uid = crgetuid(cr);
7957                 provider->dtpv_priv.dtpp_zoneid = crgetzoneid(cr);
7958         }
7959         provider->dtpv_pops = *pops;

7961         if (pops->dtps_provide == NULL) {
7962                 ASSERT(pops->dtps_provide_module != NULL);
7963                 provider->dtpv_pops.dtps_provide =
7964                     (void (*)(void *, const dtrace_probedesc_t *))dtrace_nullop;
7965         }

7967         if (pops->dtps_provide_module == NULL) {
7968                 ASSERT(pops->dtps_provide != NULL);
7969                 provider->dtpv_pops.dtps_provide_module =
7970                     (void (*)(void *, struct modctl *))dtrace_nullop;
7971         }

7973         if (pops->dtps_suspend == NULL) {
7974                 ASSERT(pops->dtps_resume == NULL);
7975                 provider->dtpv_pops.dtps_suspend =
7976                     (void (*)(void *, dtrace_id_t, void *))dtrace_nullop;
7977                 provider->dtpv_pops.dtps_resume =
7978                     (void (*)(void *, dtrace_id_t, void *))dtrace_nullop;
7979         }
```



```
7981         provider->dtpv_arg = arg;
7982         *idp = (dtrace_provider_id_t)provider;

7984         if (pops == &dtrace_provider_ops) {
7985                 ASSERT(MUTEX_HELD(&dtrace_provider_lock));
7986                 ASSERT(MUTEX_HELD(&dtrace_lock));
7987                 ASSERT(dtrace_anon.dta_enabling == NULL);

7989                 /*
7990                  * We make sure that the DTrace provider is at the head of
7991                  * the provider chain.
7992                  */
7993                 provider->dtpv_next = dtrace_provider;
7994                 dtrace_provider = provider;
7995                 return (0);
7996         }

7998         mutex_enter(&dtrace_provider_lock);
7999         mutex_enter(&dtrace_lock);

8001         /*
8002          * If there is at least one provider registered, we'll add this
8003          * provider after the first provider.
8004          */
8005         if (dtrace_provider != NULL) {
8006                 provider->dtpv_next = dtrace_provider->dtpv_next;
8007                 dtrace_provider->dtpv_next = provider;
8008         } else {
8009                 dtrace_provider = provider;
8010         }

8012         if (dtrace_retained != NULL) {
8013                 dtrace_enabling_provide(provider);

8015                 /*
8016                  * Now we need to call dtrace_enabling_matchall() -- which
8017                  * will acquire cpu_lock and dtrace_lock.  We therefore need
8018                  * to drop all of our locks before calling into it...
8019                  */
8020                 mutex_exit(&dtrace_lock);
8021                 mutex_exit(&dtrace_provider_lock);
8022                 dtrace_enabling_matchall();

8024                 return (0);
8025         }

8027         mutex_exit(&dtrace_lock);
8028         mutex_exit(&dtrace_provider_lock);

8030         return (0);
8031 }

8033 /*
8034  * Unregister the specified provider from the DTrace framework.  This should
8035  * generally be called by DTrace providers in their detach(9E) entry point.
8036  */
8037 int
8038 dtrace_unregister(dtrace_provider_id_t id)
8039 {
8040         dtrace_provider_t *old = (dtrace_provider_t *)id;
8041         dtrace_provider_t *prev = NULL;
8042         int i, self = 0, noreap = 0;
8043         dtrace_probe_t *probe, *first = NULL;

8045         if (old->dtpv_pops.dtps_enable ==
8046             (int (*)(void *, dtrace_id_t, void *))dtrace_enable_nullop) {
```

```
8047                 /*
8048                  * If DTrace itself is the provider, we're called with locks
8049                  * already held.
8050                  */
8051                 ASSERT(old == dtrace_provider);
8052                 ASSERT(dtrace_devi != NULL);
8053                 ASSERT(MUTEX_HELD(&dtrace_provider_lock));
8054                 ASSERT(MUTEX_HELD(&dtrace_lock));
8055                 self = 1;

8057                 if (dtrace_provider->dtpv_next != NULL) {
8058                         /*
8059                          * There's another provider here; return failure.
8060                          */
8061                         return (EBUSY);
8062                 }
8063         } else {
8064                 mutex_enter(&dtrace_provider_lock);
8065                 mutex_enter(&mod_lock);
8066                 mutex_enter(&dtrace_lock);
8067         }

8069         /*
8070          * If anyone has /dev/dtrace open, or if there are anonymous enabled
8071          * probes, we refuse to let providers slither away, unless this
8072          * provider has already been explicitly invalidated.
8073          */
8074         if (!old->dtpv_defunct &&
8075             (dtrace_opens || (dtrace_anon.dta_state != NULL &&
8076             dtrace_anon.dta_state->dts_necbs > 0))) {
8077                 if (!self) {
8078                         mutex_exit(&dtrace_lock);
8079                         mutex_exit(&mod_lock);
8080                         mutex_exit(&dtrace_provider_lock);
8081                 }
8082                 return (EBUSY);
8083         }

8085         /*
8086          * Attempt to destroy the probes associated with this provider.
8087          */
8088         for (i = 0; i < dtrace_nprobes; i++) {
8089                 if ((probe = dtrace_probes[i]) == NULL)
8090                         continue;

8092                 if (probe->dtpr_provider != old)
8093                         continue;

8095                 if (probe->dtpr_ecb == NULL)
8096                         continue;

8098                 /*
8099                  * If we are trying to unregister a defunct provider, and the
8100                  * provider was made defunct within the interval dictated by
8101                  * dtrace_unregister_defunct_reap, we'll (asynchronously)
8102                  * attempt to reap our enablings.  To denote that the provider
8103                  * should reattempt to unregister itself at some point in the
8104                  * future, we will return a differentiable error code (EAGAIN
8105                  * instead of EBUSY) in this case.
8106                  */
8107                 if (dtrace_gethrtime() - old->dtpv_defunct >
8108                     dtrace_unregister_defunct_reap)
8109                         noreap = 1;

8111                 if (!self) {
8112                         mutex_exit(&dtrace_lock);
```



```
8113                         mutex_exit(&mod_lock);
8114                         mutex_exit(&dtrace_provider_lock);
8115                 }

8117                 if (noreap)
8118                         return (EBUSY);

8120                 (void) taskq_dispatch(dtrace_taskq,
8121                     (task_func_t *)dtrace_enabling_reap, NULL, TQ_SLEEP);

8123                 return (EAGAIN);
8124         }

8126         /*
8127          * All of the probes for this provider are disabled; we can safely
8128          * remove all of them from their hash chains and from the probe array.
8129          */
8130         for (i = 0; i < dtrace_nprobes; i++) {
8131                 if ((probe = dtrace_probes[i]) == NULL)
8132                         continue;

8134                 if (probe->dtpr_provider != old)
8135                         continue;

8137                 dtrace_probes[i] = NULL;

8139                 dtrace_hash_remove(dtrace_bymod, probe);
8140                 dtrace_hash_remove(dtrace_byfunc, probe);
8141                 dtrace_hash_remove(dtrace_byname, probe);

8143                 if (first == NULL) {
8144                         first = probe;
8145                         probe->dtpr_nextmod = NULL;
8146                 } else {
8147                         probe->dtpr_nextmod = first;
8148                         first = probe;
8149                 }
8150         }

8152         /*
8153          * The provider's probes have been removed from the hash chains and
8154          * from the probe array.  Now issue a dtrace_sync() to be sure that
8155          * everyone has cleared out from any probe array processing.
8156          */
8157         dtrace_sync();

8159         for (probe = first; probe != NULL; probe = first) {
8160                 first = probe->dtpr_nextmod;

8162                 old->dtpv_pops.dtps_destroy(old->dtpv_arg, probe->dtpr_id,
8163                     probe->dtpr_arg);
8164                 kmem_free(probe->dtpr_mod, strlen(probe->dtpr_mod) + 1);
8165                 kmem_free(probe->dtpr_func, strlen(probe->dtpr_func) + 1);
8166                 kmem_free(probe->dtpr_name, strlen(probe->dtpr_name) + 1);
8167                 vmem_free(dtrace_arena, (void *)(uintptr_t)(probe->dtpr_id), 1);
8168                 kmem_free(probe, sizeof (dtrace_probe_t));
8169         }

8171         if ((prev = dtrace_provider) == old) {
8172                 ASSERT(self || dtrace_devi == NULL);
8173                 ASSERT(old->dtpv_next == NULL || dtrace_devi == NULL);
8174                 dtrace_provider = old->dtpv_next;
8175         } else {
8176                 while (prev != NULL && prev->dtpv_next != old)
8177                         prev = prev->dtpv_next;
```

```
8179                 if (prev == NULL) {
8180                         panic("attempt to unregister non-existent "
8181                             "dtrace provider %p\n", (void *)id);
8182                 }
8183 
8184                 prev->dtpv_next = old->dtpv_next;
8185         }
8186 
8187         if (!self) {
8188                 mutex_exit(&dtrace_lock);
8189                 mutex_exit(&mod_lock);
8190                 mutex_exit(&dtrace_provider_lock);
8191         }
8192 
8193         kmem_free(old->dtpv_name, strlen(old->dtpv_name) + 1);
8194         kmem_free(old, sizeof (dtrace_provider_t));
8195 
8196         return (0);
8197 }
8198 
8199 /*
8200  * Invalidate the specified provider.  All subsequent probe lookups for the
8201  * specified provider will fail, but its probes will not be removed.
8202  */
8203 void
8204 dtrace_invalidate(dtrace_provider_id_t id)
8205 {
8206         dtrace_provider_t *pvp = (dtrace_provider_t *)id;
8207 
8208         ASSERT(pvp->dtpv_pops.dtps_enable !=
8209             (int (*)(void *, dtrace_id_t, void *))dtrace_enable_nullop);
8210 
8211         mutex_enter(&dtrace_provider_lock);
8212         mutex_enter(&dtrace_lock);
8213 
8214         pvp->dtpv_defunct = dtrace_gethrtime();
8215 
8216         mutex_exit(&dtrace_lock);
8217         mutex_exit(&dtrace_provider_lock);
8218 }
8219 
8220 /*
8221  * Indicate whether or not DTrace has attached.
8222  */
8223 int
8224 dtrace_attached(void)
8225 {
8226         /*
8227          * dtrace_provider will be non-NULL iff the DTrace driver has
8228          * attached.  (It's non-NULL because DTrace is always itself a
8229          * provider.)
8230          */
8231         return (dtrace_provider != NULL);
8232 }
8233 
8234 /*
8235  * Remove all the unenabled probes for the given provider.  This function is
8236  * not unlike dtrace_unregister(), except that it doesn't remove the provider
8237  * -- just as many of its associated probes as it can.
8238  */
8239 int
8240 dtrace_condense(dtrace_provider_id_t id)
8241 {
8242         dtrace_provider_t *prov = (dtrace_provider_t *)id;
8243         int i;
8244         dtrace_probe_t *probe;
```



```
8245 
8246         /*
8247          * Make sure this isn't the dtrace provider itself.
8248          */
8249         ASSERT(prov->dtpv_pops.dtps_enable !=
8250             (int (*)(void *, dtrace_id_t, void *))dtrace_enable_nullop);
8251 
8252         mutex_enter(&dtrace_provider_lock);
8253         mutex_enter(&dtrace_lock);
8254 
8255         /*
8256          * Attempt to destroy the probes associated with this provider.
8257          */
8258         for (i = 0; i < dtrace_nprobes; i++) {
8259                 if ((probe = dtrace_probes[i]) == NULL)
8260                         continue;
8261 
8262                 if (probe->dtpr_provider != prov)
8263                         continue;
8264 
8265                 if (probe->dtpr_ecb != NULL)
8266                         continue;
8267 
8268                 dtrace_probes[i] = NULL;
8269 
8270                 dtrace_hash_remove(dtrace_bymod, probe);
8271                 dtrace_hash_remove(dtrace_byfunc, probe);
8272                 dtrace_hash_remove(dtrace_byname, probe);
8273 
8274                 prov->dtpv_pops.dtps_destroy(prov->dtpv_arg, i + 1,
8275                     probe->dtpr_arg);
8276                 kmem_free(probe->dtpr_mod, strlen(probe->dtpr_mod) + 1);
8277                 kmem_free(probe->dtpr_func, strlen(probe->dtpr_func) + 1);
8278                 kmem_free(probe->dtpr_name, strlen(probe->dtpr_name) + 1);
8279                 kmem_free(probe, sizeof (dtrace_probe_t));
8280                 vmem_free(dtrace_arena, (void *)((uintptr_t)i + 1), 1);
8281         }
8282 
8283         mutex_exit(&dtrace_lock);
8284         mutex_exit(&dtrace_provider_lock);
8285 
8286         return (0);
8287 }
8288 
8289 /*
8290  * DTrace Probe Management Functions
8291  *
8292  * The functions in this section perform the DTrace probe management,
8293  * including functions to create probes, look-up probes, and call into the
8294  * providers to request that probes be provided.  Some of these functions are
8295  * in the Provider-to-Framework API; these functions can be identified by the
8296  * fact that they are not declared "static".
8297  */
8298 
8299 /*
8300  * Create a probe with the specified module name, function name, and name.
8301  */
8302 dtrace_id_t
8303 dtrace_probe_create(dtrace_provider_id_t prov, const char *mod,
8304     const char *func, const char *name, int aframes, void *arg)
8305 {
8306         dtrace_probe_t *probe, **probes;
8307         dtrace_provider_t *provider = (dtrace_provider_t *)prov;
8308         dtrace_id_t id;
8309 
8310         if (provider == dtrace_provider) {
```

```
8311                ASSERT(MUTEX_HELD(&dtrace_lock));
8312        } else {
8313                mutex_enter(&dtrace_lock);
8314        }

8316        id = (dtrace_id_t)(uintptr_t)vmem_alloc(dtrace_arena, 1,
8317            VM_BESTFIT | VM_SLEEP);
8318        probe = kmem_zalloc(sizeof (dtrace_probe_t), KM_SLEEP);

8320        probe->dtpr_id = id;
8321        probe->dtpr_gen = dtrace_probegen++;
8322        probe->dtpr_mod = dtrace_strdup(mod);
8323        probe->dtpr_func = dtrace_strdup(func);
8324        probe->dtpr_name = dtrace_strdup(name);
8325        probe->dtpr_arg = arg;
8326        probe->dtpr_aframes = aframes;
8327        probe->dtpr_provider = provider;

8329        dtrace_hash_add(dtrace_bymod, probe);
8330        dtrace_hash_add(dtrace_byfunc, probe);
8331        dtrace_hash_add(dtrace_byname, probe);

8333        if (id - 1 >= dtrace_nprobes) {
8334                size_t osize = dtrace_nprobes * sizeof (dtrace_probe_t *);
8335                size_t nsize = osize << 1;

8337                if (nsize == 0) {
8338                        ASSERT(osize == 0);
8339                        ASSERT(dtrace_probes == NULL);
8340                        nsize = sizeof (dtrace_probe_t *);
8341                }

8343                probes = kmem_zalloc(nsize, KM_SLEEP);

8345                if (dtrace_probes == NULL) {
8346                        ASSERT(osize == 0);
8347                        dtrace_probes = probes;
8348                        dtrace_nprobes = 1;
8349                } else {
8350                        dtrace_probe_t **oprobes = dtrace_probes;

8352                        bcopy(oprobes, probes, osize);
8353                        dtrace_membar_producer();
8354                        dtrace_probes = probes;

8356                        dtrace_sync();

8358                        /*
8359                         * All CPUs are now seeing the new probes array; we can
8360                         * safely free the old array.
8361                         */
8362                        kmem_free(oprobes, osize);
8363                        dtrace_nprobes <<= 1;
8364                }

8366                ASSERT(id - 1 < dtrace_nprobes);
8367        }

8369        ASSERT(dtrace_probes[id - 1] == NULL);
8370        dtrace_probes[id - 1] = probe;

8372        if (provider != dtrace_provider)
8373                mutex_exit(&dtrace_lock);

8375        return (id);
8376 }
```



```
8378 static dtrace_probe_t *
8379 dtrace_probe_lookup_id(dtrace_id_t id)
8380 {
8381         ASSERT(MUTEX_HELD(&dtrace_lock));

8383         if (id == 0 || id > dtrace_nprobes)
8384                 return (NULL);

8386         return (dtrace_probes[id - 1]);
8387 }

8389 static int
8390 dtrace_probe_lookup_match(dtrace_probe_t *probe, void *arg)
8391 {
8392         *((dtrace_id_t *)arg) = probe->dtpr_id;

8394         return (DTRACE_MATCH_DONE);
8395 }

8397 /*
8398  * Look up a probe based on provider and one or more of module name, function
8399  * name and probe name.
8400  */
8401 dtrace_id_t
8402 dtrace_probe_lookup(dtrace_provider_id_t prid, const char *mod,
8403     const char *func, const char *name)
8404 {
8405         dtrace_probekey_t pkey;
8406         dtrace_id_t id;
8407         int match;

8409         pkey.dtpk_prov = ((dtrace_provider_t *)prid)->dtpv_name;
8410         pkey.dtpk_pmatch = &dtrace_match_string;
8411         pkey.dtpk_mod = mod;
8412         pkey.dtpk_mmatch = mod ? &dtrace_match_string : &dtrace_match_nul;
8413         pkey.dtpk_func = func;
8414         pkey.dtpk_fmatch = func ? &dtrace_match_string : &dtrace_match_nul;
8415         pkey.dtpk_name = name;
8416         pkey.dtpk_nmatch = name ? &dtrace_match_string : &dtrace_match_nul;
8417         pkey.dtpk_id = DTRACE_IDNONE;

8419         mutex_enter(&dtrace_lock);
8420         match = dtrace_match(&pkey, DTRACE_PRIV_ALL, 0, 0,
8421             dtrace_probe_lookup_match, &id);
8422         mutex_exit(&dtrace_lock);

8424         ASSERT(match == 1 || match == 0);
8425         return (match ? id : 0);
8426 }

8428 /*
8429  * Returns the probe argument associated with the specified probe.
8430  */
8431 void *
8432 dtrace_probe_arg(dtrace_provider_id_t id, dtrace_id_t pid)
8433 {
8434         dtrace_probe_t *probe;
8435         void *rval = NULL;

8437         mutex_enter(&dtrace_lock);

8439         if ((probe = dtrace_probe_lookup_id(pid)) != NULL &&
8440             probe->dtpr_provider == (dtrace_provider_t *)id)
8441                 rval = probe->dtpr_arg;
```

```
8443         mutex_exit(&dtrace_lock);

8445         return (rval);
8446 }

8448 /*
8449  * Copy a probe into a probe description.
8450  */
8451 static void
8452 dtrace_probe_description(const dtrace_probe_t *prp, dtrace_probedesc_t *pdp)
8453 {
8454         bzero(pdp, sizeof (dtrace_probedesc_t));
8455         pdp->dtpd_id = prp->dtpr_id;

8457         (void) strncpy(pdp->dtpd_provider,
8458             prp->dtpr_provider->dtpv_name, DTRACE_PROVNAMELEN - 1);

8460         (void) strncpy(pdp->dtpd_mod, prp->dtpr_mod, DTRACE_MODNAMELEN - 1);
8461         (void) strncpy(pdp->dtpd_func, prp->dtpr_func, DTRACE_FUNCNAMELEN - 1);
8462         (void) strncpy(pdp->dtpd_name, prp->dtpr_name, DTRACE_NAMELEN - 1);
8463 }

8465 /*
8466  * Called to indicate that a probe -- or probes -- should be provided by a
8467  * specfied provider.  If the specified description is NULL, the provider will
8468  * be told to provide all of its probes.  (This is done whenever a new
8469  * consumer comes along, or whenever a retained enabling is to be matched.) If
8470  * the specified description is non-NULL, the provider is given the
8471  * opportunity to dynamically provide the specified probe, allowing providers
8472  * to support the creation of probes on-the-fly.  (So-called _autocreated_
8473  * probes.)  If the provider is NULL, the operations will be applied to all
8474  * providers; if the provider is non-NULL the operations will only be applied
8475  * to the specified provider.  The dtrace_provider_lock must be held, and the
8476  * dtrace_lock must _not_ be held -- the provider's dtps_provide() operation
8477  * will need to grab the dtrace_lock when it reenters the framework through
8478  * dtrace_probe_lookup(), dtrace_probe_create(), etc.
8479  */
8480 static void
8481 dtrace_probe_provide(dtrace_probedesc_t *desc, dtrace_provider_t *prv)
8482 {
8483         struct modctl *ctl;
8484         int all = 0;

8486         ASSERT(MUTEX_HELD(&dtrace_provider_lock));

8488         if (prv == NULL) {
8489                 all = 1;
8490                 prv = dtrace_provider;
8491         }

8493         do {
8494                 /*
8495                  * First, call the blanket provide operation.
8496                  */
8497                 prv->dtpv_pops.dtps_provide(prv->dtpv_arg, desc);

8499                 /*
8500                  * Now call the per-module provide operation.  We will grab
8501                  * mod_lock to prevent the list from being modified.  Note
8502                  * that this also prevents the mod_busy bits from changing.
8503                  * (mod_busy can only be changed with mod_lock held.)
8504                  */
8505                 mutex_enter(&mod_lock);

8507                 ctl = &modules;
8508                 do {
```



```
8509                         if (ctl->mod_busy || ctl->mod_mp == NULL)
8510                                 continue;

8512                         prv->dtpv_pops.dtps_provide_module(prv->dtpv_arg, ctl);

8514                 } while ((ctl = ctl->mod_next) != &modules);

8516                 mutex_exit(&mod_lock);
8517         } while (all && (prv = prv->dtpv_next) != NULL);
8518 }

8520 /*
8521  * Iterate over each probe, and call the Framework-to-Provider API function
8522  * denoted by offs.
8523  */
8524 static void
8525 dtrace_probe_foreach(uintptr_t offs)
8526 {
8527         dtrace_provider_t *prov;
8528         void (*func)(void *, dtrace_id_t, void *);
8529         dtrace_probe_t *probe;
8530         dtrace_icookie_t cookie;
8531         int i;

8533         /*
8534          * We disable interrupts to walk through the probe array.  This is
8535          * safe -- the dtrace_sync() in dtrace_unregister() assures that we
8536          * won't see stale data.
8537          */
8538         cookie = dtrace_interrupt_disable();

8540         for (i = 0; i < dtrace_nprobes; i++) {
8541                 if ((probe = dtrace_probes[i]) == NULL)
8542                         continue;

8544                 if (probe->dtpr_ecb == NULL) {
8545                         /*
8546                          * This probe isn't enabled -- don't call the function.
8547                          */
8548                         continue;
8549                 }

8551                 prov = probe->dtpr_provider;
8552                 func = *((void(**)(void *, dtrace_id_t, void *))
8553                     ((uintptr_t)&prov->dtpv_pops + offs));

8555                 func(prov->dtpv_arg, i + 1, probe->dtpr_arg);
8556         }

8558         dtrace_interrupt_enable(cookie);
8559 }

8561 static int
8562 dtrace_probe_enable(const dtrace_probedesc_t *desc, dtrace_enabling_t *enab)
8563 {
8564         dtrace_probekey_t pkey;
8565         uint32_t priv;
8566         uid_t uid;
8567         zoneid_t zoneid;

8569         ASSERT(MUTEX_HELD(&dtrace_lock));
8570         dtrace_ecb_create_cache = NULL;

8572         if (desc == NULL) {
8573                 /*
8574                  * If we're passed a NULL description, we're being asked to
```

```
8575                 * create an ECB with a NULL probe.
8576                 */
8577                (void) dtrace_ecb_create_enable(NULL, enab);
8578                return (0);
8579        }

8581        dtrace_probekey(desc, &pkey);
8582        dtrace_cred2priv(enab->dten_vstate->dtvs_state->dts_cred.dcr_cred,
8583            &priv, &uid, &zoneid);

8585        return (dtrace_match(&pkey, priv, uid, zoneid, dtrace_ecb_create_enable,
8586            enab));
8587 }

8589 /*
8590  * DTrace Helper Provider Functions
8591  */
8592 static void
8593 dtrace_dofattr2attr(dtrace_attribute_t *attr, const dof_attr_t dofattr)
8594 {
8595        attr->dtat_name = DOF_ATTR_NAME(dofattr);
8596        attr->dtat_data = DOF_ATTR_DATA(dofattr);
8597        attr->dtat_class = DOF_ATTR_CLASS(dofattr);
8598 }

8600 static void
8601 dtrace_dofprov2hprov(dtrace_helper_provdesc_t *hprov,
8602     const dof_provider_t *dofprov, char *strtab)
8603 {
8604        hprov->dthpv_provname = strtab + dofprov->dofpv_name;
8605        dtrace_dofattr2attr(&hprov->dthpv_pattr.dtpa_provider,
8606            dofprov->dofpv_provattr);
8607        dtrace_dofattr2attr(&hprov->dthpv_pattr.dtpa_mod,
8608            dofprov->dofpv_modattr);
8609        dtrace_dofattr2attr(&hprov->dthpv_pattr.dtpa_func,
8610            dofprov->dofpv_funcattr);
8611        dtrace_dofattr2attr(&hprov->dthpv_pattr.dtpa_name,
8612            dofprov->dofpv_nameattr);
8613        dtrace_dofattr2attr(&hprov->dthpv_pattr.dtpa_args,
8614            dofprov->dofpv_argsattr);
8615 }

8617 static void
8618 dtrace_helper_provide_one(dof_helper_t *dhp, dof_sec_t *sec, pid_t pid)
8619 {
8620        uintptr_t daddr = (uintptr_t)dhp->dofhp_dof;
8621        dof_hdr_t *dof = (dof_hdr_t *)daddr;
8622        dof_sec_t *str_sec, *prb_sec, *arg_sec, *off_sec, *enoff_sec;
8623        dof_provider_t *provider;
8624        dof_probe_t *probe;
8625        uint32_t *off, *enoff;
8626        uint8_t *arg;
8627        char *strtab;
8628        uint_t i, nprobes;
8629        dtrace_helper_provdesc_t dhpv;
8630        dtrace_helper_probedesc_t dhpb;
8631        dtrace_meta_t *meta = dtrace_meta_pid;
8632        dtrace_mops_t *mops = &meta->dtm_mops;
8633        void *parg;

8635        provider = (dof_provider_t *)(uintptr_t)(daddr + sec->dofs_offset);
8636        str_sec = (dof_sec_t *)(uintptr_t)(daddr + dof->dofh_secoff +
8637            provider->dofpv_strtab * dof->dofh_secsize);
8638        prb_sec = (dof_sec_t *)(uintptr_t)(daddr + dof->dofh_secoff +
8639            provider->dofpv_probes * dof->dofh_secsize);
8640        arg_sec = (dof_sec_t *)(uintptr_t)(daddr + dof->dofh_secoff +
```



```
8641            provider->dofpv_prargs * dof->dofh_secsize);
8642        off_sec = (dof_sec_t *)(uintptr_t)(daddr + dof->dofh_secoff +
8643            provider->dofpv_proffs * dof->dofh_secsize);

8645        strtab = (char *)(uintptr_t)(daddr + str_sec->dofs_offset);
8646        off = (uint32_t *)(uintptr_t)(daddr + off_sec->dofs_offset);
8647        arg = (uint8_t *)(uintptr_t)(daddr + arg_sec->dofs_offset);
8648        enoff = NULL;

8650        /*
8651         * See dtrace_helper_provider_validate().
8652         */
8653        if (dof->dofh_ident[DOF_ID_VERSION] != DOF_VERSION_1 &&
8654            provider->dofpv_prenoffs != DOF_SECT_NONE) {
8655                enoff_sec = (dof_sec_t *)(uintptr_t)(daddr + dof->dofh_secoff +
8656                    provider->dofpv_prenoffs * dof->dofh_secsize);
8657                enoff = (uint32_t *)(uintptr_t)(daddr + enoff_sec->dofs_offset);
8658        }

8660        nprobes = prb_sec->dofs_size / prb_sec->dofs_entsize;

8662        /*
8663         * Create the provider.
8664         */
8665        dtrace_dofprov2hprov(&dhpv, provider, strtab);

8667        if ((parg = mops->dtms_provide_pid(meta->dtm_arg, &dhpv, pid)) == NULL)
8668                return;

8670        meta->dtm_count++;

8672        /*
8673         * Create the probes.
8674         */
8675        for (i = 0; i < nprobes; i++) {
8676                probe = (dof_probe_t *)(uintptr_t)(daddr +
8677                    prb_sec->dofs_offset + i * prb_sec->dofs_entsize);

8679                dhpb.dthpb_mod = dhp->dofhp_mod;
8680                dhpb.dthpb_func = strtab + probe->dofpr_func;
8681                dhpb.dthpb_name = strtab + probe->dofpr_name;
8682                dhpb.dthpb_base = probe->dofpr_addr;
8683                dhpb.dthpb_offs = off + probe->dofpr_offidx;
8684                dhpb.dthpb_noffs = probe->dofpr_noffs;
8685                if (enoff != NULL) {
8686                        dhpb.dthpb_enoffs = enoff + probe->dofpr_enoffidx;
8687                        dhpb.dthpb_nenoffs = probe->dofpr_nenoffs;
8688                } else {
8689                        dhpb.dthpb_enoffs = NULL;
8690                        dhpb.dthpb_nenoffs = 0;
8691                }
8692                dhpb.dthpb_args = arg + probe->dofpr_argidx;
8693                dhpb.dthpb_nargc = probe->dofpr_nargc;
8694                dhpb.dthpb_xargc = probe->dofpr_xargc;
8695                dhpb.dthpb_ntypes = strtab + probe->dofpr_nargv;
8696                dhpb.dthpb_xtypes = strtab + probe->dofpr_xargv;

8698                mops->dtms_create_probe(meta->dtm_arg, parg, &dhpb);
8699        }
8700 }

8702 static void
8703 dtrace_helper_provide(dof_helper_t *dhp, pid_t pid)
8704 {
8705        uintptr_t daddr = (uintptr_t)dhp->dofhp_dof;
8706        dof_hdr_t *dof = (dof_hdr_t *)daddr;
```

```
8707         int i;

8709         ASSERT(MUTEX_HELD(&dtrace_meta_lock));

8711         for (i = 0; i < dof->dofh_secnum; i++) {
8712                 dof_sec_t *sec = (dof_sec_t *)(uintptr_t)(daddr +
8713                     dof->dofh_secoff + i * dof->dofh_secsize);

8715                 if (sec->dofs_type != DOF_SECT_PROVIDER)
8716                         continue;

8718                 dtrace_helper_provide_one(dhp, sec, pid);
8719         }

8721         /*
8722          * We may have just created probes, so we must now rematch against
8723          * any retained enablings.  Note that this call will acquire both
8724          * cpu_lock and dtrace_lock; the fact that we are holding
8725          * dtrace_meta_lock now is what defines the ordering with respect to
8726          * these three locks.
8727          */
8728         dtrace_enabling_matchall();
8729 }

8731 static void
8732 dtrace_helper_provider_remove_one(dof_helper_t *dhp, dof_sec_t *sec, pid_t pid)
8733 {
8734         uintptr_t daddr = (uintptr_t)dhp->dofhp_dof;
8735         dof_hdr_t *dof = (dof_hdr_t *)daddr;
8736         dof_sec_t *str_sec;
8737         dof_provider_t *provider;
8738         char *strtab;
8739         dtrace_helper_provdesc_t dhpv;
8740         dtrace_meta_t *meta = dtrace_meta_pid;
8741         dtrace_mops_t *mops = &meta->dtm_mops;

8743         provider = (dof_provider_t *)(uintptr_t)(daddr + sec->dofs_offset);
8744         str_sec = (dof_sec_t *)(uintptr_t)(daddr + dof->dofh_secoff +
8745             provider->dofpv_strtab * dof->dofh_secsize);

8747         strtab = (char *)(uintptr_t)(daddr + str_sec->dofs_offset);

8749         /*
8750          * Create the provider.
8751          */
8752         dtrace_dofprov2hprov(&dhpv, provider, strtab);

8754         mops->dtms_remove_pid(meta->dtm_arg, &dhpv, pid);

8756         meta->dtm_count--;
8757 }

8759 static void
8760 dtrace_helper_provider_remove(dof_helper_t *dhp, pid_t pid)
8761 {
8762         uintptr_t daddr = (uintptr_t)dhp->dofhp_dof;
8763         dof_hdr_t *dof = (dof_hdr_t *)daddr;
8764         int i;

8766         ASSERT(MUTEX_HELD(&dtrace_meta_lock));

8768         for (i = 0; i < dof->dofh_secnum; i++) {
8769                 dof_sec_t *sec = (dof_sec_t *)(uintptr_t)(daddr +
8770                     dof->dofh_secoff + i * dof->dofh_secsize);

8772                 if (sec->dofs_type != DOF_SECT_PROVIDER)
```



```
8773                         continue;

8775                 dtrace_helper_provider_remove_one(dhp, sec, pid);
8776         }
8777 }

8779 /*
8780  * DTrace Meta Provider-to-Framework API Functions
8781  *
8782  * These functions implement the Meta Provider-to-Framework API, as described
8783  * in <sys/dtrace.h>.
8784  */
8785 int
8786 dtrace_meta_register(const char *name, const dtrace_mops_t *mops, void *arg,
8787     dtrace_meta_provider_id_t *idp)
8788 {
8789         dtrace_meta_t *meta;
8790         dtrace_helpers_t *help, *next;
8791         int i;

8793         *idp = DTRACE_METAPROVNONE;

8795         /*
8796          * We strictly don't need the name, but we hold onto it for
8797          * debuggability. All hail error queues!
8798          */
8799         if (name == NULL) {
8800                 cmn_err(CE_WARN, "failed to register meta-provider: "
8801                     "invalid name");
8802                 return (EINVAL);
8803         }

8805         if (mops == NULL ||
8806             mops->dtms_create_probe == NULL ||
8807             mops->dtms_provide_pid == NULL ||
8808             mops->dtms_remove_pid == NULL) {
8809                 cmn_err(CE_WARN, "failed to register meta-register %s: "
8810                     "invalid ops", name);
8811                 return (EINVAL);
8812         }

8814         meta = kmem_zalloc(sizeof (dtrace_meta_t), KM_SLEEP);
8815         meta->dtm_mops = *mops;
8816         meta->dtm_name = kmem_alloc(strlen(name) + 1, KM_SLEEP);
8817         (void) strcpy(meta->dtm_name, name);
8818         meta->dtm_arg = arg;

8820         mutex_enter(&dtrace_meta_lock);
8821         mutex_enter(&dtrace_lock);

8823         if (dtrace_meta_pid != NULL) {
8824                 mutex_exit(&dtrace_lock);
8825                 mutex_exit(&dtrace_meta_lock);
8826                 cmn_err(CE_WARN, "failed to register meta-register %s: "
8827                     "user-land meta-provider exists", name);
8828                 kmem_free(meta->dtm_name, strlen(meta->dtm_name) + 1);
8829                 kmem_free(meta, sizeof (dtrace_meta_t));
8830                 return (EINVAL);
8831         }

8833         dtrace_meta_pid = meta;
8834         *idp = (dtrace_meta_provider_id_t)meta;

8836         /*
8837          * If there are providers and probes ready to go, pass them
8838          * off to the new meta provider now.
```

```
8839          */
8841         help = dtrace_deferred_pid;
8842         dtrace_deferred_pid = NULL;
8844         mutex_exit(&dtrace_lock);
8846         while (help != NULL) {
8847                 for (i = 0; i < help->dthps_nprovs; i++) {
8848                         dtrace_helper_provide(&help->dthps_provs[i]->dthp_prov,
8849                             help->dthps_pid);
8850                 }
8852                 next = help->dthps_next;
8853                 help->dthps_next = NULL;
8854                 help->dthps_prev = NULL;
8855                 help->dthps_deferred = 0;
8856                 help = next;
8857         }
8859         mutex_exit(&dtrace_meta_lock);
8861         return (0);
8862 }
8864 int
8865 dtrace_meta_unregister(dtrace_meta_provider_id_t id)
8866 {
8867         dtrace_meta_t **pp, *old = (dtrace_meta_t *)id;
8869         mutex_enter(&dtrace_meta_lock);
8870         mutex_enter(&dtrace_lock);
8872         if (old == dtrace_meta_pid) {
8873                 pp = &dtrace_meta_pid;
8874         } else {
8875                 panic("attempt to unregister non-existent "
8876                     "dtrace meta-provider %p\n", (void *)old);
8877         }
8879         if (old->dtm_count != 0) {
8880                 mutex_exit(&dtrace_lock);
8881                 mutex_exit(&dtrace_meta_lock);
8882                 return (EBUSY);
8883         }
8885         *pp = NULL;
8887         mutex_exit(&dtrace_lock);
8888         mutex_exit(&dtrace_meta_lock);
8890         kmem_free(old->dtm_name, strlen(old->dtm_name) + 1);
8891         kmem_free(old, sizeof (dtrace_meta_t));
8893         return (0);
8894 }
8897 /*
8898  * DTrace DIF Object Functions
8899  */
8900 static int
8901 dtrace_difo_err(uint_t pc, const char *format, ...)
8902 {
8903         if (dtrace_err_verbose) {
8904                 va_list alist;
```



```
8906                 (void) uprintf("dtrace DIF object error: [%u]: ", pc);
8907                 va_start(alist, format);
8908                 (void) vuprintf(format, alist);
8909                 va_end(alist);
8910         }
8912 #ifdef DTRACE_ERRDEBUG
8913         dtrace_errdebug(format);
8914 #endif
8915         return (1);
8916 }
8918 /*
8919  * Validate a DTrace DIF object by checking the IR instructions.  The following
8920  * rules are currently enforced by dtrace_difo_validate():
8921  *
8922  * 1. Each instruction must have a valid opcode
8923  * 2. Each register, string, variable, or subroutine reference must be valid
8924  * 3. No instruction can modify register %r0 (must be zero)
8925  * 4. All instruction reserved bits must be set to zero
8926  * 5. The last instruction must be a "ret" instruction
8927  * 6. All branch targets must reference a valid instruction _after_ the branch
8928  */
8929 static int
8930 dtrace_difo_validate(dtrace_difo_t *dp, dtrace_vstate_t *vstate, uint_t nregs,
8931     cred_t *cr)
8932 {
8933         int err = 0, i;
8934         int (*efunc)(uint_t pc, const char *, ...) = dtrace_difo_err;
8935         int kcheckload;
8936         uint_t pc;
8938         kcheckload = cr == NULL ||
8939             (vstate->dtvs_state->dts_cred.dcr_visible & DTRACE_CRV_KERNEL) == 0;
8941         dp->dtdo_destructive = 0;
8943         for (pc = 0; pc < dp->dtdo_len && err == 0; pc++) {
8944                 dif_instr_t instr = dp->dtdo_buf[pc];
8946                 uint_t r1 = DIF_INSTR_R1(instr);
8947                 uint_t r2 = DIF_INSTR_R2(instr);
8948                 uint_t rd = DIF_INSTR_RD(instr);
8949                 uint_t rs = DIF_INSTR_RS(instr);
8950                 uint_t label = DIF_INSTR_LABEL(instr);
8951                 uint_t v = DIF_INSTR_VAR(instr);
8952                 uint_t subr = DIF_INSTR_SUBR(instr);
8953                 uint_t type = DIF_INSTR_TYPE(instr);
8954                 uint_t op = DIF_INSTR_OP(instr);
8956                 switch (op) {
8957                 case DIF_OP_OR:
8958                 case DIF_OP_XOR:
8959                 case DIF_OP_AND:
8960                 case DIF_OP_SLL:
8961                 case DIF_OP_SRL:
8962                 case DIF_OP_SRA:
8963                 case DIF_OP_SUB:
8964                 case DIF_OP_ADD:
8965                 case DIF_OP_MUL:
8966                 case DIF_OP_SDIV:
8967                 case DIF_OP_UDIV:
8968                 case DIF_OP_SREM:
8969                 case DIF_OP_UREM:
8970                 case DIF_OP_COPYS:
```

```
8971                    if (r1 >= nregs)
8972                            err += efunc(pc, "invalid register %u\n", r1);
8973                    if (r2 >= nregs)
8974                            err += efunc(pc, "invalid register %u\n", r2);
8975                    if (rd >= nregs)
8976                            err += efunc(pc, "invalid register %u\n", rd);
8977                    if (rd == 0)
8978                            err += efunc(pc, "cannot write to %r0\n");
8979                    break;
8980            case DIF_OP_NOT:
8981            case DIF_OP_MOV:
8982            case DIF_OP_ALLOCS:
8983                    if (r1 >= nregs)
8984                            err += efunc(pc, "invalid register %u\n", r1);
8985                    if (r2 != 0)
8986                            err += efunc(pc, "non-zero reserved bits\n");
8987                    if (rd >= nregs)
8988                            err += efunc(pc, "invalid register %u\n", rd);
8989                    if (rd == 0)
8990                            err += efunc(pc, "cannot write to %r0\n");
8991                    break;
8992            case DIF_OP_LDSB:
8993            case DIF_OP_LDSH:
8994            case DIF_OP_LDSW:
8995            case DIF_OP_LDUB:
8996            case DIF_OP_LDUH:
8997            case DIF_OP_LDUW:
8998            case DIF_OP_LDX:
8999                    if (r1 >= nregs)
9000                            err += efunc(pc, "invalid register %u\n", r1);
9001                    if (r2 != 0)
9002                            err += efunc(pc, "non-zero reserved bits\n");
9003                    if (rd >= nregs)
9004                            err += efunc(pc, "invalid register %u\n", rd);
9005                    if (rd == 0)
9006                            err += efunc(pc, "cannot write to %r0\n");
9007                    if (kcheckload)
9008                            dp->dtdo_buf[pc] = DIF_INSTR_LOAD(op +
9009                                DIF_OP_RLDSB - DIF_OP_LDSB, r1, rd);
9010                    break;
9011            case DIF_OP_RLDSB:
9012            case DIF_OP_RLDSH:
9013            case DIF_OP_RLDSW:
9014            case DIF_OP_RLDUB:
9015            case DIF_OP_RLDUH:
9016            case DIF_OP_RLDUW:
9017            case DIF_OP_RLDX:
9018                    if (r1 >= nregs)
9019                            err += efunc(pc, "invalid register %u\n", r1);
9020                    if (r2 != 0)
9021                            err += efunc(pc, "non-zero reserved bits\n");
9022                    if (rd >= nregs)
9023                            err += efunc(pc, "invalid register %u\n", rd);
9024                    if (rd == 0)
9025                            err += efunc(pc, "cannot write to %r0\n");
9026                    break;
9027            case DIF_OP_ULDSB:
9028            case DIF_OP_ULDSH:
9029            case DIF_OP_ULDSW:
9030            case DIF_OP_ULDUB:
9031            case DIF_OP_ULDUH:
9032            case DIF_OP_ULDUW:
9033            case DIF_OP_ULDX:
9034                    if (r1 >= nregs)
9035                            err += efunc(pc, "invalid register %u\n", r1);
9036                    if (r2 != 0)
```



```
9037                            err += efunc(pc, "non-zero reserved bits\n");
9038                    if (rd >= nregs)
9039                            err += efunc(pc, "invalid register %u\n", rd);
9040                    if (rd == 0)
9041                            err += efunc(pc, "cannot write to %r0\n");
9042                    break;
9043            case DIF_OP_STB:
9044            case DIF_OP_STH:
9045            case DIF_OP_STW:
9046            case DIF_OP_STX:
9047                    if (r1 >= nregs)
9048                            err += efunc(pc, "invalid register %u\n", r1);
9049                    if (r2 != 0)
9050                            err += efunc(pc, "non-zero reserved bits\n");
9051                    if (rd >= nregs)
9052                            err += efunc(pc, "invalid register %u\n", rd);
9053                    if (rd == 0)
9054                            err += efunc(pc, "cannot write to 0 address\n");
9055                    break;
9056            case DIF_OP_CMP:
9057            case DIF_OP_SCMP:
9058                    if (r1 >= nregs)
9059                            err += efunc(pc, "invalid register %u\n", r1);
9060                    if (r2 >= nregs)
9061                            err += efunc(pc, "invalid register %u\n", r2);
9062                    if (rd != 0)
9063                            err += efunc(pc, "non-zero reserved bits\n");
9064                    break;
9065            case DIF_OP_TST:
9066                    if (r1 >= nregs)
9067                            err += efunc(pc, "invalid register %u\n", r1);
9068                    if (r2 != 0 || rd != 0)
9069                            err += efunc(pc, "non-zero reserved bits\n");
9070                    break;
9071            case DIF_OP_BA:
9072            case DIF_OP_BE:
9073            case DIF_OP_BNE:
9074            case DIF_OP_BG:
9075            case DIF_OP_BGU:
9076            case DIF_OP_BGE:
9077            case DIF_OP_BGEU:
9078            case DIF_OP_BL:
9079            case DIF_OP_BLU:
9080            case DIF_OP_BLE:
9081            case DIF_OP_BLEU:
9082                    if (label >= dp->dtdo_len) {
9083                            err += efunc(pc, "invalid branch target %u\n",
9084                                label);
9085                    }
9086                    if (label <= pc) {
9087                            err += efunc(pc, "backward branch to %u\n",
9088                                label);
9089                    }
9090                    break;
9091            case DIF_OP_RET:
9092                    if (r1 != 0 || r2 != 0)
9093                            err += efunc(pc, "non-zero reserved bits\n");
9094                    if (rd >= nregs)
9095                            err += efunc(pc, "invalid register %u\n", rd);
9096                    break;
9097            case DIF_OP_NOP:
9098            case DIF_OP_POPTS:
9099            case DIF_OP_FLUSHTS:
9100                    if (r1 != 0 || r2 != 0 || rd != 0)
9101                            err += efunc(pc, "non-zero reserved bits\n");
9102                    break;
```

```
9103                case DIF_OP_SETX:
9104                        if (DIF_INSTR_INTEGER(instr) >= dp->dtdo_intlen) {
9105                                err += efunc(pc, "invalid integer ref %u\n",
9106                                    DIF_INSTR_INTEGER(instr));
9107                        }
9108                        if (rd >= nregs)
9109                                err += efunc(pc, "invalid register %u\n", rd);
9110                        if (rd == 0)
9111                                err += efunc(pc, "cannot write to %r0\n");
9112                        break;
9113                case DIF_OP_SETS:
9114                        if (DIF_INSTR_STRING(instr) >= dp->dtdo_strlen) {
9115                                err += efunc(pc, "invalid string ref %u\n",
9116                                    DIF_INSTR_STRING(instr));
9117                        }
9118                        if (rd >= nregs)
9119                                err += efunc(pc, "invalid register %u\n", rd);
9120                        if (rd == 0)
9121                                err += efunc(pc, "cannot write to %r0\n");
9122                        break;
9123                case DIF_OP_LDGA:
9124                case DIF_OP_LDTA:
9125                        if (r1 > DIF_VAR_ARRAY_MAX)
9126                                err += efunc(pc, "invalid array %u\n", r1);
9127                        if (r2 >= nregs)
9128                                err += efunc(pc, "invalid register %u\n", r2);
9129                        if (rd >= nregs)
9130                                err += efunc(pc, "invalid register %u\n", rd);
9131                        if (rd == 0)
9132                                err += efunc(pc, "cannot write to %r0\n");
9133                        break;
9134                case DIF_OP_LDGS:
9135                case DIF_OP_LDTS:
9136                case DIF_OP_LDLS:
9137                case DIF_OP_LDGAA:
9138                case DIF_OP_LDTAA:
9139                        if (v < DIF_VAR_OTHER_MIN || v > DIF_VAR_OTHER_MAX)
9140                                err += efunc(pc, "invalid variable %u\n", v);
9141                        if (rd >= nregs)
9142                                err += efunc(pc, "invalid register %u\n", rd);
9143                        if (rd == 0)
9144                                err += efunc(pc, "cannot write to %r0\n");
9145                        break;
9146                case DIF_OP_STGS:
9147                case DIF_OP_STTS:
9148                case DIF_OP_STLS:
9149                case DIF_OP_STGAA:
9150                case DIF_OP_STTAA:
9151                        if (v < DIF_VAR_OTHER_UBASE || v > DIF_VAR_OTHER_MAX)
9152                                err += efunc(pc, "invalid variable %u\n", v);
9153                        if (rs >= nregs)
9154                                err += efunc(pc, "invalid register %u\n", rd);
9155                        break;
9156                case DIF_OP_CALL:
9157                        if (subr > DIF_SUBR_MAX)
9158                                err += efunc(pc, "invalid subr %u\n", subr);
9159                        if (rd >= nregs)
9160                                err += efunc(pc, "invalid register %u\n", rd);
9161                        if (rd == 0)
9162                                err += efunc(pc, "cannot write to %r0\n");

9164                        if (subr == DIF_SUBR_COPYOUT ||
9165                            subr == DIF_SUBR_COPYOUTSTR) {
9166                                dp->dtdo_destructive = 1;
9167                        }
```



```
9169                        if (subr == DIF_SUBR_GETF) {
9170                                /*
9171                                 * If we have a getf() we need to record that
9172                                 * in our state.  Note that our state can be
9173                                 * NULL if this is a helper -- but in that
9174                                 * case, the call to getf() is itself illegal,
9175                                 * and will be caught (slightly later) when
9176                                 * the helper is validated.
9177                                 */
9178                                if (vstate->dtvs_state != NULL)
9179                                        vstate->dtvs_state->dts_getf++;
9180                        }

9182                        break;
9183                case DIF_OP_PUSHTR:
9184                        if (type != DIF_TYPE_STRING && type != DIF_TYPE_CTF)
9185                                err += efunc(pc, "invalid ref type %u\n", type);
9186                        if (r2 >= nregs)
9187                                err += efunc(pc, "invalid register %u\n", r2);
9188                        if (rs >= nregs)
9189                                err += efunc(pc, "invalid register %u\n", rs);
9190                        break;
9191                case DIF_OP_PUSHTV:
9192                        if (type != DIF_TYPE_CTF)
9193                                err += efunc(pc, "invalid val type %u\n", type);
9194                        if (r2 >= nregs)
9195                                err += efunc(pc, "invalid register %u\n", r2);
9196                        if (rs >= nregs)
9197                                err += efunc(pc, "invalid register %u\n", rs);
9198                        break;
9199                default:
9200                        err += efunc(pc, "invalid opcode %u\n",
9201                            DIF_INSTR_OP(instr));
9202                }
9203        }

9205        if (dp->dtdo_len != 0 &&
9206            DIF_INSTR_OP(dp->dtdo_buf[dp->dtdo_len - 1]) != DIF_OP_RET) {
9207                err += efunc(dp->dtdo_len - 1,
9208                    "expected 'ret' as last DIF instruction\n");
9209        }

9211        if (!(dp->dtdo_rtype.dtdt_flags & DIF_TF_BYREF)) {
9212                /*
9213                 * If we're not returning by reference, the size must be either
9214                 * 0 or the size of one of the base types.
9215                 */
9216                switch (dp->dtdo_rtype.dtdt_size) {
9217                case 0:
9218                case sizeof (uint8_t):
9219                case sizeof (uint16_t):
9220                case sizeof (uint32_t):
9221                case sizeof (uint64_t):
9222                        break;

9224                default:
9225                        err += efunc(dp->dtdo_len - 1, "bad return size\n");
9226                }
9227        }

9229        for (i = 0; i < dp->dtdo_varlen && err == 0; i++) {
9230                dtrace_difv_t *v = &dp->dtdo_vartab[i], *existing = NULL;
9231                dtrace_diftype_t *vt, *et;
9232                uint_t id, ndx;

9234                if (v->dtdv_scope != DIFV_SCOPE_GLOBAL &&
```

```
9235                    v->dtdv_scope != DIFV_SCOPE_THREAD &&
9236                    v->dtdv_scope != DIFV_SCOPE_LOCAL) {
9237                        err += efunc(i, "unrecognized variable scope %d\n",
9238                            v->dtdv_scope);
9239                        break;
9240                }

9242                if (v->dtdv_kind != DIFV_KIND_ARRAY &&
9243                    v->dtdv_kind != DIFV_KIND_SCALAR) {
9244                        err += efunc(i, "unrecognized variable type %d\n",
9245                            v->dtdv_kind);
9246                        break;
9247                }

9249                if ((id = v->dtdv_id) > DIF_VARIABLE_MAX) {
9250                        err += efunc(i, "%d exceeds variable id limit\n", id);
9251                        break;
9252                }

9254                if (id < DIF_VAR_OTHER_UBASE)
9255                        continue;

9257                /*
9258                 * For user-defined variables, we need to check that this
9259                 * definition is identical to any previous definition that we
9260                 * encountered.
9261                 */
9262                ndx = id - DIF_VAR_OTHER_UBASE;

9264                switch (v->dtdv_scope) {
9265                case DIFV_SCOPE_GLOBAL:
9266                        if (ndx < vstate->dtvs_nglobals) {
9267                                dtrace_statvar_t *svar;

9269                                if ((svar = vstate->dtvs_globals[ndx]) != NULL)
9270                                        existing = &svar->dtsv_var;
9271                        }

9273                        break;

9275                case DIFV_SCOPE_THREAD:
9276                        if (ndx < vstate->dtvs_ntlocals)
9277                                existing = &vstate->dtvs_tlocals[ndx];
9278                        break;

9280                case DIFV_SCOPE_LOCAL:
9281                        if (ndx < vstate->dtvs_nlocals) {
9282                                dtrace_statvar_t *svar;

9284                                if ((svar = vstate->dtvs_locals[ndx]) != NULL)
9285                                        existing = &svar->dtsv_var;
9286                        }

9288                        break;
9289                }

9291                vt = &v->dtdv_type;

9293                if (vt->dtdt_flags & DIF_TF_BYREF) {
9294                        if (vt->dtdt_size == 0) {
9295                                err += efunc(i, "zero-sized variable\n");
9296                                break;
9297                        }

9299                        if (v->dtdv_scope == DIFV_SCOPE_GLOBAL &&
9300                            vt->dtdt_size > dtrace_global_maxsize) {
```



```
9301                                err += efunc(i, "oversized by-ref global\n");
9302                                break;
9303                        }
9304                }

9306                if (existing == NULL || existing->dtdv_id == 0)
9307                        continue;

9309                ASSERT(existing->dtdv_id == v->dtdv_id);
9310                ASSERT(existing->dtdv_scope == v->dtdv_scope);

9312                if (existing->dtdv_kind != v->dtdv_kind)
9313                        err += efunc(i, "%d changed variable kind\n", id);

9315                et = &existing->dtdv_type;

9317                if (vt->dtdt_flags != et->dtdt_flags) {
9318                        err += efunc(i, "%d changed variable type flags\n", id);
9319                        break;
9320                }

9322                if (vt->dtdt_size != 0 && vt->dtdt_size != et->dtdt_size) {
9323                        err += efunc(i, "%d changed variable type size\n", id);
9324                        break;
9325                }
9326        }

9328        return (err);
9329 }

9331 /*
9332  * Validate a DTrace DIF object that it is to be used as a helper.  Helpers
9333  * are much more constrained than normal DIFOs.  Specifically, they may
9334  * not:
9335  *
9336  * 1. Make calls to subroutines other than copyin(), copyinstr() or
9337  *    miscellaneous string routines
9338  * 2. Access DTrace variables other than the args[] array, and the
9339  *    curthread, pid, ppid, tid, execname, zonename, uid and gid variables.
9340  * 3. Have thread-local variables.
9341  * 4. Have dynamic variables.
9342  */
9343 static int
9344 dtrace_difo_validate_helper(dtrace_difo_t *dp)
9345 {
9346        int (*efunc)(uint_t pc, const char *, ...) = dtrace_difo_err;
9347        int err = 0;
9348        uint_t pc;

9350        for (pc = 0; pc < dp->dtdo_len; pc++) {
9351                dif_instr_t instr = dp->dtdo_buf[pc];

9353                uint_t v = DIF_INSTR_VAR(instr);
9354                uint_t subr = DIF_INSTR_SUBR(instr);
9355                uint_t op = DIF_INSTR_OP(instr);

9357                switch (op) {
9358                case DIF_OP_OR:
9359                case DIF_OP_XOR:
9360                case DIF_OP_AND:
9361                case DIF_OP_SLL:
9362                case DIF_OP_SRL:
9363                case DIF_OP_SRA:
9364                case DIF_OP_SUB:
9365                case DIF_OP_ADD:
9366                case DIF_OP_MUL:
```

```
9367                 case DIF_OP_SDIV:
9368                 case DIF_OP_UDIV:
9369                 case DIF_OP_SREM:
9370                 case DIF_OP_UREM:
9371                 case DIF_OP_COPYS:
9372                 case DIF_OP_NOT:
9373                 case DIF_OP_MOV:
9374                 case DIF_OP_RLDSB:
9375                 case DIF_OP_RLDSH:
9376                 case DIF_OP_RLDSW:
9377                 case DIF_OP_RLDUB:
9378                 case DIF_OP_RLDUH:
9379                 case DIF_OP_RLDUW:
9380                 case DIF_OP_RLDX:
9381                 case DIF_OP_ULDSB:
9382                 case DIF_OP_ULDSH:
9383                 case DIF_OP_ULDSW:
9384                 case DIF_OP_ULDUB:
9385                 case DIF_OP_ULDUH:
9386                 case DIF_OP_ULDUW:
9387                 case DIF_OP_ULDX:
9388                 case DIF_OP_STB:
9389                 case DIF_OP_STH:
9390                 case DIF_OP_STW:
9391                 case DIF_OP_STX:
9392                 case DIF_OP_ALLOCS:
9393                 case DIF_OP_CMP:
9394                 case DIF_OP_SCMP:
9395                 case DIF_OP_TST:
9396                 case DIF_OP_BA:
9397                 case DIF_OP_BE:
9398                 case DIF_OP_BNE:
9399                 case DIF_OP_BG:
9400                 case DIF_OP_BGU:
9401                 case DIF_OP_BGE:
9402                 case DIF_OP_BGEU:
9403                 case DIF_OP_BL:
9404                 case DIF_OP_BLU:
9405                 case DIF_OP_BLE:
9406                 case DIF_OP_BLEU:
9407                 case DIF_OP_RET:
9408                 case DIF_OP_NOP:
9409                 case DIF_OP_POPTS:
9410                 case DIF_OP_FLUSHTS:
9411                 case DIF_OP_SETX:
9412                 case DIF_OP_SETS:
9413                 case DIF_OP_LDGA:
9414                 case DIF_OP_LDLS:
9415                 case DIF_OP_STGS:
9416                 case DIF_OP_STLS:
9417                 case DIF_OP_PUSHTR:
9418                 case DIF_OP_PUSHTV:
9419                         break;

9421                 case DIF_OP_LDGS:
9422                         if (v >= DIF_VAR_OTHER_UBASE)
9423                                 break;

9425                         if (v >= DIF_VAR_ARG0 && v <= DIF_VAR_ARG9)
9426                                 break;

9428                         if (v == DIF_VAR_CURTHREAD || v == DIF_VAR_PID ||
9429                             v == DIF_VAR_PPID || v == DIF_VAR_TID ||
9430                             v == DIF_VAR_EXECNAME || v == DIF_VAR_ZONENAME ||
9431                             v == DIF_VAR_UID || v == DIF_VAR_GID)
9432                                 break;
```



```
9434                         err += efunc(pc, "illegal variable %u\n", v);
9435                         break;

9437                 case DIF_OP_LDTA:
9438                 case DIF_OP_LDTS:
9439                 case DIF_OP_LDGAA:
9440                 case DIF_OP_LDTAA:
9441                         err += efunc(pc, "illegal dynamic variable load\n");
9442                         break;

9444                 case DIF_OP_STTS:
9445                 case DIF_OP_STGAA:
9446                 case DIF_OP_STTAA:
9447                         err += efunc(pc, "illegal dynamic variable store\n");
9448                         break;

9450                 case DIF_OP_CALL:
9451                         if (subr == DIF_SUBR_ALLOCA ||
9452                             subr == DIF_SUBR_BCOPY ||
9453                             subr == DIF_SUBR_COPYIN ||
9454                             subr == DIF_SUBR_COPYINTO ||
9455                             subr == DIF_SUBR_COPYINSTR ||
9456                             subr == DIF_SUBR_INDEX ||
9457                             subr == DIF_SUBR_INET_NTOA ||
9458                             subr == DIF_SUBR_INET_NTOA6 ||
9459                             subr == DIF_SUBR_INET_NTOP ||
9460                             subr == DIF_SUBR_JSON ||
9461 #endif /* ! codereview */
9462                             subr == DIF_SUBR_LLTOSTR ||
9463                             subr == DIF_SUBR_STRTOLL ||
9464 #endif /* ! codereview */
9465                             subr == DIF_SUBR_RINDEX ||
9466                             subr == DIF_SUBR_STRCHR ||
9467                             subr == DIF_SUBR_STRJOIN ||
9468                             subr == DIF_SUBR_STRRCHR ||
9469                             subr == DIF_SUBR_STRSTR ||
9470                             subr == DIF_SUBR_HTONS ||
9471                             subr == DIF_SUBR_HTONL ||
9472                             subr == DIF_SUBR_HTONLL ||
9473                             subr == DIF_SUBR_NTOHS ||
9474                             subr == DIF_SUBR_NTOHL ||
9475                             subr == DIF_SUBR_NTOHLL)
9476                                 break;

9478                         err += efunc(pc, "invalid subr %u\n", subr);
9479                         break;

9481                 default:
9482                         err += efunc(pc, "invalid opcode %u\n",
9483                             DIF_INSTR_OP(instr));
9484                 }
9485         }

9487         return (err);
9488 }

9490 /*
9491  * Returns 1 if the expression in the DIF object can be cached on a per-thread
9492  * basis; 0 if not.
9493  */
9494 static int
9495 dtrace_difo_cacheable(dtrace_difo_t *dp)
9496 {
9497         int i;
```

```
9499         if (dp == NULL)
9500                 return (0);

9502         for (i = 0; i < dp->dtdo_varlen; i++) {
9503                 dtrace_difv_t *v = &dp->dtdo_vartab[i];

9505                 if (v->dtdv_scope != DIFV_SCOPE_GLOBAL)
9506                         continue;

9508                 switch (v->dtdv_id) {
9509                 case DIF_VAR_CURTHREAD:
9510                 case DIF_VAR_PID:
9511                 case DIF_VAR_TID:
9512                 case DIF_VAR_EXECNAME:
9513                 case DIF_VAR_ZONENAME:
9514                         break;

9516                 default:
9517                         return (0);
9518                 }
9519         }

9521         /*
9522          * This DIF object may be cacheable.  Now we need to look for any
9523          * array loading instructions, any memory loading instructions, or
9524          * any stores to thread-local variables.
9525          */
9526         for (i = 0; i < dp->dtdo_len; i++) {
9527                 uint_t op = DIF_INSTR_OP(dp->dtdo_buf[i]);

9529                 if ((op >= DIF_OP_LDSB && op <= DIF_OP_LDX) ||
9530                     (op >= DIF_OP_ULDSB && op <= DIF_OP_ULDX) ||
9531                     (op >= DIF_OP_RLDSB && op <= DIF_OP_RLDX) ||
9532                     op == DIF_OP_LDGA || op == DIF_OP_STTS)
9533                         return (0);
9534         }

9536         return (1);
9537 }

9539 static void
9540 dtrace_difo_hold(dtrace_difo_t *dp)
9541 {
9542         int i;

9544         ASSERT(MUTEX_HELD(&dtrace_lock));

9546         dp->dtdo_refcnt++;
9547         ASSERT(dp->dtdo_refcnt != 0);

9549         /*
9550          * We need to check this DIF object for references to the variable
9551          * DIF_VAR_VTIMESTAMP.
9552          */
9553         for (i = 0; i < dp->dtdo_varlen; i++) {
9554                 dtrace_difv_t *v = &dp->dtdo_vartab[i];

9556                 if (v->dtdv_id != DIF_VAR_VTIMESTAMP)
9557                         continue;

9559                 if (dtrace_vtime_references++ == 0)
9560                         dtrace_vtime_enable();
9561         }
9562 }

9564 /*
```



```
9565  * This routine calculates the dynamic variable chunksize for a given DIF
9566  * object.  The calculation is not fool-proof, and can probably be tricked by
9567  * malicious DIF -- but it works for all compiler-generated DIF.  Because this
9568  * calculation is likely imperfect, dtrace_dynvar() is able to gracefully fail
9569  * if a dynamic variable size exceeds the chunksize.
9570  */
9571 static void
9572 dtrace_difo_chunksize(dtrace_difo_t *dp, dtrace_vstate_t *vstate)
9573 {
9574         uint64_t sval;
9575         dtrace_key_t tupregs[DIF_DTR_NREGS + 2]; /* +2 for thread and id */
9576         const dif_instr_t *text = dp->dtdo_buf;
9577         uint_t pc, srd = 0;
9578         uint_t ttop = 0;
9579         size_t size, ksize;
9580         uint_t id, i;

9582         for (pc = 0; pc < dp->dtdo_len; pc++) {
9583                 dif_instr_t instr = text[pc];
9584                 uint_t op = DIF_INSTR_OP(instr);
9585                 uint_t rd = DIF_INSTR_RD(instr);
9586                 uint_t r1 = DIF_INSTR_R1(instr);
9587                 uint_t nkeys = 0;
9588                 uchar_t scope;

9590                 dtrace_key_t *key = tupregs;

9592                 switch (op) {
9593                 case DIF_OP_SETX:
9594                         sval = dp->dtdo_inttab[DIF_INSTR_INTEGER(instr)];
9595                         srd = rd;
9596                         continue;

9598                 case DIF_OP_STTS:
9599                         key = &tupregs[DIF_DTR_NREGS];
9600                         key[0].dttk_size = 0;
9601                         key[1].dttk_size = 0;
9602                         nkeys = 2;
9603                         scope = DIFV_SCOPE_THREAD;
9604                         break;

9606                 case DIF_OP_STGAA:
9607                 case DIF_OP_STTAA:
9608                         nkeys = ttop;

9610                         if (DIF_INSTR_OP(instr) == DIF_OP_STTAA)
9611                                 key[nkeys++].dttk_size = 0;

9613                         key[nkeys++].dttk_size = 0;

9615                         if (op == DIF_OP_STTAA) {
9616                                 scope = DIFV_SCOPE_THREAD;
9617                         } else {
9618                                 scope = DIFV_SCOPE_GLOBAL;
9619                         }

9621                         break;

9623                 case DIF_OP_PUSHTR:
9624                         if (ttop == DIF_DTR_NREGS)
9625                                 return;

9627                         if ((srd == 0 || sval == 0) && r1 == DIF_TYPE_STRING) {
9628                                 /*
9629                                  * If the register for the size of the "pushtr"
9630                                  * is %r0 (or the value is 0) and the type is
```

```
9631                                 * a string, we'll use the system-wide default
9632                                 * string size.
9633                                 */
9634                                tupregs[ttop++].dttk_size =
9635                                    dtrace_strsize_default;
9636                        } else {
9637                                if (srd == 0)
9638                                        return;

9640                                tupregs[ttop++].dttk_size = sval;
9641                        }

9643                        break;

9645                case DIF_OP_PUSHTV:
9646                        if (ttop == DIF_DTR_NREGS)
9647                                return;

9649                        tupregs[ttop++].dttk_size = 0;
9650                        break;

9652                case DIF_OP_FLUSHTS:
9653                        ttop = 0;
9654                        break;

9656                case DIF_OP_POPTS:
9657                        if (ttop != 0)
9658                                ttop--;
9659                        break;
9660                }

9662                sval = 0;
9663                srd = 0;

9665                if (nkeys == 0)
9666                        continue;

9668                /*
9669                 * We have a dynamic variable allocation; calculate its size.
9670                 */
9671                for (ksize = 0, i = 0; i < nkeys; i++)
9672                        ksize += P2ROUNDUP(key[i].dttk_size, sizeof (uint64_t));

9674                size = sizeof (dtrace_dynvar_t);
9675                size += sizeof (dtrace_key_t) * (nkeys - 1);
9676                size += ksize;

9678                /*
9679                 * Now we need to determine the size of the stored data.
9680                 */
9681                id = DIF_INSTR_VAR(instr);

9683                for (i = 0; i < dp->dtdo_varlen; i++) {
9684                        dtrace_difv_t *v = &dp->dtdo_vartab[i];

9686                        if (v->dtdv_id == id && v->dtdv_scope == scope) {
9687                                size += v->dtdv_type.dtdt_size;
9688                                break;
9689                        }
9690                }

9692                if (i == dp->dtdo_varlen)
9693                        return;

9695                /*
9696                 * We have the size.  If this is larger than the chunk size
```



```
9697                 * for our dynamic variable state, reset the chunk size.
9698                 */
9699                size = P2ROUNDUP(size, sizeof (uint64_t));

9701                if (size > vstate->dtvs_dynvars.dtds_chunksize)
9702                        vstate->dtvs_dynvars.dtds_chunksize = size;
9703        }
9704 }

9706 static void
9707 dtrace_difo_init(dtrace_difo_t *dp, dtrace_vstate_t *vstate)
9708 {
9709        int i, oldsvars, osz, nsz, otlocals, ntlocals;
9710        uint_t id;

9712        ASSERT(MUTEX_HELD(&dtrace_lock));
9713        ASSERT(dp->dtdo_buf != NULL && dp->dtdo_len != 0);

9715        for (i = 0; i < dp->dtdo_varlen; i++) {
9716                dtrace_difv_t *v = &dp->dtdo_vartab[i];
9717                dtrace_statvar_t *svar, ***svarp;
9718                size_t dsize = 0;
9719                uint8_t scope = v->dtdv_scope;
9720                int *np;

9722                if ((id = v->dtdv_id) < DIF_VAR_OTHER_UBASE)
9723                        continue;

9725                id -= DIF_VAR_OTHER_UBASE;

9727                switch (scope) {
9728                case DIFV_SCOPE_THREAD:
9729                        while (id >= (otlocals = vstate->dtvs_ntlocals)) {
9730                                dtrace_difv_t *tlocals;

9732                                if ((ntlocals = (otlocals << 1)) == 0)
9733                                        ntlocals = 1;

9735                                osz = otlocals * sizeof (dtrace_difv_t);
9736                                nsz = ntlocals * sizeof (dtrace_difv_t);

9738                                tlocals = kmem_zalloc(nsz, KM_SLEEP);

9740                                if (osz != 0) {
9741                                        bcopy(vstate->dtvs_tlocals,
9742                                            tlocals, osz);
9743                                        kmem_free(vstate->dtvs_tlocals, osz);
9744                                }

9746                                vstate->dtvs_tlocals = tlocals;
9747                                vstate->dtvs_ntlocals = ntlocals;
9748                        }

9750                        vstate->dtvs_tlocals[id] = *v;
9751                        continue;

9753                case DIFV_SCOPE_LOCAL:
9754                        np = &vstate->dtvs_nlocals;
9755                        svarp = &vstate->dtvs_locals;

9757                        if (v->dtdv_type.dtdt_flags & DIF_TF_BYREF)
9758                                dsize = NCPU * (v->dtdv_type.dtdt_size +
9759                                    sizeof (uint64_t));
9760                        else
9761                                dsize = NCPU * sizeof (uint64_t);
```

```
9763                            break;

9765                    case DIFV_SCOPE_GLOBAL:
9766                            np = &vstate->dtvs_nglobals;
9767                            svarp = &vstate->dtvs_globals;

9769                            if (v->dtdv_type.dtdt_flags & DIF_TF_BYREF)
9770                                    dsize = v->dtdv_type.dtdt_size +
9771                                        sizeof (uint64_t);

9773                            break;

9775                    default:
9776                            ASSERT(0);
9777                    }

9779                    while (id >= (oldsvars = *np)) {
9780                            dtrace_statvar_t **statics;
9781                            int newsvars, oldsize, newsize;

9783                            if ((newsvars = (oldsvars << 1)) == 0)
9784                                    newsvars = 1;

9786                            oldsize = oldsvars * sizeof (dtrace_statvar_t *);
9787                            newsize = newsvars * sizeof (dtrace_statvar_t *);

9789                            statics = kmem_zalloc(newsize, KM_SLEEP);

9791                            if (oldsize != 0) {
9792                                    bcopy(*svarp, statics, oldsize);
9793                                    kmem_free(*svarp, oldsize);
9794                            }

9796                            *svarp = statics;
9797                            *np = newsvars;
9798                    }

9800                    if ((svar = (*svarp)[id]) == NULL) {
9801                            svar = kmem_zalloc(sizeof (dtrace_statvar_t), KM_SLEEP);
9802                            svar->dtsv_var = *v;

9804                            if ((svar->dtsv_size = dsize) != 0) {
9805                                    svar->dtsv_data = (uint64_t)(uintptr_t)
9806                                        kmem_zalloc(dsize, KM_SLEEP);
9807                            }

9809                            (*svarp)[id] = svar;
9810                    }

9812                    svar->dtsv_refcnt++;
9813            }

9815            dtrace_difo_chunksize(dp, vstate);
9816            dtrace_difo_hold(dp);
9817    }

9819    static dtrace_difo_t *
9820    dtrace_difo_duplicate(dtrace_difo_t *dp, dtrace_vstate_t *vstate)
9821    {
9822            dtrace_difo_t *new;
9823            size_t sz;

9825            ASSERT(dp->dtdo_buf != NULL);
9826            ASSERT(dp->dtdo_refcnt != 0);

9828            new = kmem_zalloc(sizeof (dtrace_difo_t), KM_SLEEP);
```



```
9830            ASSERT(dp->dtdo_buf != NULL);
9831            sz = dp->dtdo_len * sizeof (dif_instr_t);
9832            new->dtdo_buf = kmem_alloc(sz, KM_SLEEP);
9833            bcopy(dp->dtdo_buf, new->dtdo_buf, sz);
9834            new->dtdo_len = dp->dtdo_len;

9836            if (dp->dtdo_strtab != NULL) {
9837                    ASSERT(dp->dtdo_strlen != 0);
9838                    new->dtdo_strtab = kmem_alloc(dp->dtdo_strlen, KM_SLEEP);
9839                    bcopy(dp->dtdo_strtab, new->dtdo_strtab, dp->dtdo_strlen);
9840                    new->dtdo_strlen = dp->dtdo_strlen;
9841            }

9843            if (dp->dtdo_inttab != NULL) {
9844                    ASSERT(dp->dtdo_intlen != 0);
9845                    sz = dp->dtdo_intlen * sizeof (uint64_t);
9846                    new->dtdo_inttab = kmem_alloc(sz, KM_SLEEP);
9847                    bcopy(dp->dtdo_inttab, new->dtdo_inttab, sz);
9848                    new->dtdo_intlen = dp->dtdo_intlen;
9849            }

9851            if (dp->dtdo_vartab != NULL) {
9852                    ASSERT(dp->dtdo_varlen != 0);
9853                    sz = dp->dtdo_varlen * sizeof (dtrace_difv_t);
9854                    new->dtdo_vartab = kmem_alloc(sz, KM_SLEEP);
9855                    bcopy(dp->dtdo_vartab, new->dtdo_vartab, sz);
9856                    new->dtdo_varlen = dp->dtdo_varlen;
9857            }

9859            dtrace_difo_init(new, vstate);
9860            return (new);
9861    }

9863    static void
9864    dtrace_difo_destroy(dtrace_difo_t *dp, dtrace_vstate_t *vstate)
9865    {
9866            int i;

9868            ASSERT(dp->dtdo_refcnt == 0);

9870            for (i = 0; i < dp->dtdo_varlen; i++) {
9871                    dtrace_difv_t *v = &dp->dtdo_vartab[i];
9872                    dtrace_statvar_t *svar, **svarp;
9873                    uint_t id;
9874                    uint8_t scope = v->dtdv_scope;
9875                    int *np;

9877                    switch (scope) {
9878                    case DIFV_SCOPE_THREAD:
9879                            continue;

9881                    case DIFV_SCOPE_LOCAL:
9882                            np = &vstate->dtvs_nlocals;
9883                            svarp = vstate->dtvs_locals;
9884                            break;

9886                    case DIFV_SCOPE_GLOBAL:
9887                            np = &vstate->dtvs_nglobals;
9888                            svarp = vstate->dtvs_globals;
9889                            break;

9891                    default:
9892                            ASSERT(0);
9893                    }
```

```
9895                if ((id = v->dtdv_id) < DIF_VAR_OTHER_UBASE)
9896                        continue;

9898                id -= DIF_VAR_OTHER_UBASE;
9899                ASSERT(id < *np);

9901                svar = svarp[id];
9902                ASSERT(svar != NULL);
9903                ASSERT(svar->dtsv_refcnt > 0);

9905                if (--svar->dtsv_refcnt > 0)
9906                        continue;

9908                if (svar->dtsv_size != 0) {
9909                        ASSERT(svar->dtsv_data != NULL);
9910                        kmem_free((void *)(uintptr_t)svar->dtsv_data,
9911                            svar->dtsv_size);
9912                }

9914                kmem_free(svar, sizeof (dtrace_statvar_t));
9915                svarp[id] = NULL;
9916        }

9918        kmem_free(dp->dtdo_buf, dp->dtdo_len * sizeof (dif_instr_t));
9919        kmem_free(dp->dtdo_inttab, dp->dtdo_intlen * sizeof (uint64_t));
9920        kmem_free(dp->dtdo_strtab, dp->dtdo_strlen);
9921        kmem_free(dp->dtdo_vartab, dp->dtdo_varlen * sizeof (dtrace_difv_t));

9923        kmem_free(dp, sizeof (dtrace_difo_t));
9924 }

9926 static void
9927 dtrace_difo_release(dtrace_difo_t *dp, dtrace_vstate_t *vstate)
9928 {
9929        int i;

9931        ASSERT(MUTEX_HELD(&dtrace_lock));
9932        ASSERT(dp->dtdo_refcnt != 0);

9934        for (i = 0; i < dp->dtdo_varlen; i++) {
9935                dtrace_difv_t *v = &dp->dtdo_vartab[i];

9937                if (v->dtdv_id != DIF_VAR_VTIMESTAMP)
9938                        continue;

9940                ASSERT(dtrace_vtime_references > 0);
9941                if (--dtrace_vtime_references == 0)
9942                        dtrace_vtime_disable();
9943        }

9945        if (--dp->dtdo_refcnt == 0)
9946                dtrace_difo_destroy(dp, vstate);
9947 }

9949 /*
9950  * DTrace Format Functions
9951  */
9952 static uint16_t
9953 dtrace_format_add(dtrace_state_t *state, char *str)
9954 {
9955        char *fmt, **new;
9956        uint16_t ndx, len = strlen(str) + 1;

9958        fmt = kmem_zalloc(len, KM_SLEEP);
9959        bcopy(str, fmt, len);
```



```
9961        for (ndx = 0; ndx < state->dts_nformats; ndx++) {
9962                if (state->dts_formats[ndx] == NULL) {
9963                        state->dts_formats[ndx] = fmt;
9964                        return (ndx + 1);
9965                }
9966        }

9968        if (state->dts_nformats == USHRT_MAX) {
9969                /*
9970                 * This is only likely if a denial-of-service attack is being
9971                 * attempted.  As such, it's okay to fail silently here.
9972                 */
9973                kmem_free(fmt, len);
9974                return (0);
9975        }

9977        /*
9978         * For simplicity, we always resize the formats array to be exactly the
9979         * number of formats.
9980         */
9981        ndx = state->dts_nformats++;
9982        new = kmem_alloc((ndx + 1) * sizeof (char *), KM_SLEEP);

9984        if (state->dts_formats != NULL) {
9985                ASSERT(ndx != 0);
9986                bcopy(state->dts_formats, new, ndx * sizeof (char *));
9987                kmem_free(state->dts_formats, ndx * sizeof (char *));
9988        }

9990        state->dts_formats = new;
9991        state->dts_formats[ndx] = fmt;

9993        return (ndx + 1);
9994 }

9996 static void
9997 dtrace_format_remove(dtrace_state_t *state, uint16_t format)
9998 {
9999        char *fmt;

10001        ASSERT(state->dts_formats != NULL);
10002        ASSERT(format <= state->dts_nformats);
10003        ASSERT(state->dts_formats[format - 1] != NULL);

10005        fmt = state->dts_formats[format - 1];
10006        kmem_free(fmt, strlen(fmt) + 1);
10007        state->dts_formats[format - 1] = NULL;
10008 }

10010 static void
10011 dtrace_format_destroy(dtrace_state_t *state)
10012 {
10013        int i;

10015        if (state->dts_nformats == 0) {
10016                ASSERT(state->dts_formats == NULL);
10017                return;
10018        }

10020        ASSERT(state->dts_formats != NULL);

10022        for (i = 0; i < state->dts_nformats; i++) {
10023                char *fmt = state->dts_formats[i];

10025                if (fmt == NULL)
10026                        continue;
```

```
10028                   kmem_free(fmt, strlen(fmt) + 1);
10029           }

10031           kmem_free(state->dts_formats, state->dts_nformats * sizeof (char *));
10032           state->dts_nformats = 0;
10033           state->dts_formats = NULL;
10034   }

10036   /*
10037    * DTrace Predicate Functions
10038    */
10039   static dtrace_predicate_t *
10040   dtrace_predicate_create(dtrace_difo_t *dp)
10041   {
10042           dtrace_predicate_t *pred;

10044           ASSERT(MUTEX_HELD(&dtrace_lock));
10045           ASSERT(dp->dtdo_refcnt != 0);

10047           pred = kmem_zalloc(sizeof (dtrace_predicate_t), KM_SLEEP);
10048           pred->dtp_difo = dp;
10049           pred->dtp_refcnt = 1;

10051           if (!dtrace_difo_cacheable(dp))
10052                   return (pred);

10054           if (dtrace_predcache_id == DTRACE_CACHEIDNONE) {
10055                   /*
10056                    * This is only theoretically possible -- we have had 2^32
10057                    * cacheable predicates on this machine.  We cannot allow any
10058                    * more predicates to become cacheable:  as unlikely as it is,
10059                    * there may be a thread caching a (now stale) predicate cache
10060                    * ID. (N.B.: the temptation is being successfully resisted to
10061                    * have this cmn_err() "Holy shit -- we executed this code!")
10062                    */
10063                   return (pred);
10064           }

10066           pred->dtp_cacheid = dtrace_predcache_id++;

10068           return (pred);
10069   }

10071   static void
10072   dtrace_predicate_hold(dtrace_predicate_t *pred)
10073   {
10074           ASSERT(MUTEX_HELD(&dtrace_lock));
10075           ASSERT(pred->dtp_difo != NULL && pred->dtp_difo->dtdo_refcnt != 0);
10076           ASSERT(pred->dtp_refcnt > 0);

10078           pred->dtp_refcnt++;
10079   }

10081   static void
10082   dtrace_predicate_release(dtrace_predicate_t *pred, dtrace_vstate_t *vstate)
10083   {
10084           dtrace_difo_t *dp = pred->dtp_difo;

10086           ASSERT(MUTEX_HELD(&dtrace_lock));
10087           ASSERT(dp != NULL && dp->dtdo_refcnt != 0);
10088           ASSERT(pred->dtp_refcnt > 0);

10090           if (--pred->dtp_refcnt == 0) {
10091                   dtrace_difo_release(pred->dtp_difo, vstate);
10092                   kmem_free(pred, sizeof (dtrace_predicate_t));
```



```
10093           }
10094   }

10096   /*
10097    * DTrace Action Description Functions
10098    */
10099   static dtrace_actdesc_t *
10100   dtrace_actdesc_create(dtrace_actkind_t kind, uint32_t ntuple,
10101       uint64_t uarg, uint64_t arg)
10102   {
10103           dtrace_actdesc_t *act;

10105           ASSERT(!DTRACEACT_ISPRINTFLIKE(kind) || (arg != NULL &&
10106               arg >= KERNELBASE) || (arg == NULL && kind == DTRACEACT_PRINTA));

10108           act = kmem_zalloc(sizeof (dtrace_actdesc_t), KM_SLEEP);
10109           act->dtad_kind = kind;
10110           act->dtad_ntuple = ntuple;
10111           act->dtad_uarg = uarg;
10112           act->dtad_arg = arg;
10113           act->dtad_refcnt = 1;

10115           return (act);
10116   }

10118   static void
10119   dtrace_actdesc_hold(dtrace_actdesc_t *act)
10120   {
10121           ASSERT(act->dtad_refcnt >= 1);
10122           act->dtad_refcnt++;
10123   }

10125   static void
10126   dtrace_actdesc_release(dtrace_actdesc_t *act, dtrace_vstate_t *vstate)
10127   {
10128           dtrace_actkind_t kind = act->dtad_kind;
10129           dtrace_difo_t *dp;

10131           ASSERT(act->dtad_refcnt >= 1);

10133           if (--act->dtad_refcnt != 0)
10134                   return;

10136           if ((dp = act->dtad_difo) != NULL)
10137                   dtrace_difo_release(dp, vstate);

10139           if (DTRACEACT_ISPRINTFLIKE(kind)) {
10140                   char *str = (char *)(uintptr_t)act->dtad_arg;

10142                   ASSERT((str != NULL && (uintptr_t)str >= KERNELBASE) ||
10143                       (str == NULL && act->dtad_kind == DTRACEACT_PRINTA));

10145                   if (str != NULL)
10146                           kmem_free(str, strlen(str) + 1);
10147           }

10149           kmem_free(act, sizeof (dtrace_actdesc_t));
10150   }

10152   /*
10153    * DTrace ECB Functions
10154    */
10155   static dtrace_ecb_t *
10156   dtrace_ecb_add(dtrace_state_t *state, dtrace_probe_t *probe)
10157   {
10158           dtrace_ecb_t *ecb;
```

```
10159         dtrace_epid_t epid;

10161         ASSERT(MUTEX_HELD(&dtrace_lock));

10163         ecb = kmem_zalloc(sizeof (dtrace_ecb_t), KM_SLEEP);
10164         ecb->dte_predicate = NULL;
10165         ecb->dte_probe = probe;

10167         /*
10168          * The default size is the size of the default action: recording
10169          * the header.
10170          */
10171         ecb->dte_size = ecb->dte_needed = sizeof (dtrace_rechdr_t);
10172         ecb->dte_alignment = sizeof (dtrace_epid_t);

10174         epid = state->dts_epid++;

10176         if (epid - 1 >= state->dts_necbs) {
10177                 dtrace_ecb_t **oecbs = state->dts_ecbs, **ecbs;
10178                 int necbs = state->dts_necbs << 1;

10180                 ASSERT(epid == state->dts_necbs + 1);

10182                 if (necbs == 0) {
10183                         ASSERT(oecbs == NULL);
10184                         necbs = 1;
10185                 }

10187                 ecbs = kmem_zalloc(necbs * sizeof (*ecbs), KM_SLEEP);

10189                 if (oecbs != NULL)
10190                         bcopy(oecbs, ecbs, state->dts_necbs * sizeof (*ecbs));

10192                 dtrace_membar_producer();
10193                 state->dts_ecbs = ecbs;

10195                 if (oecbs != NULL) {
10196                         /*
10197                          * If this state is active, we must dtrace_sync()
10198                          * before we can free the old dts_ecbs array:  we're
10199                          * coming in hot, and there may be active ring
10200                          * buffer processing (which indexes into the dts_ecbs
10201                          * array) on another CPU.
10202                          */
10203                         if (state->dts_activity != DTRACE_ACTIVITY_INACTIVE)
10204                                 dtrace_sync();

10206                         kmem_free(oecbs, state->dts_necbs * sizeof (*ecbs));
10207                 }

10209                 dtrace_membar_producer();
10210                 state->dts_necbs = necbs;
10211         }

10213         ecb->dte_state = state;

10215         ASSERT(state->dts_ecbs[epid - 1] == NULL);
10216         dtrace_membar_producer();
10217         state->dts_ecbs[(ecb->dte_epid = epid) - 1] = ecb;

10219         return (ecb);
10220 }

10222 static int
10223 dtrace_ecb_enable(dtrace_ecb_t *ecb)
10224 {
```



```
10225         dtrace_probe_t *probe = ecb->dte_probe;

10227         ASSERT(MUTEX_HELD(&cpu_lock));
10228         ASSERT(MUTEX_HELD(&dtrace_lock));
10229         ASSERT(ecb->dte_next == NULL);

10231         if (probe == NULL) {
10232                 /*
10233                  * This is the NULL probe -- there's nothing to do.
10234                  */
10235                 return (0);
10236         }

10238         if (probe->dtpr_ecb == NULL) {
10239                 dtrace_provider_t *prov = probe->dtpr_provider;

10241                 /*
10242                  * We're the first ECB on this probe.
10243                  */
10244                 probe->dtpr_ecb = probe->dtpr_ecb_last = ecb;

10246                 if (ecb->dte_predicate != NULL)
10247                         probe->dtpr_predcache = ecb->dte_predicate->dtp_cacheid;

10249                 return (prov->dtpv_pops.dtps_enable(prov->dtpv_arg,
10250                     probe->dtpr_id, probe->dtpr_arg));
10251         } else {
10252                 /*
10253                  * This probe is already active.  Swing the last pointer to
10254                  * point to the new ECB, and issue a dtrace_sync() to assure
10255                  * that all CPUs have seen the change.
10256                  */
10257                 ASSERT(probe->dtpr_ecb_last != NULL);
10258                 probe->dtpr_ecb_last->dte_next = ecb;
10259                 probe->dtpr_ecb_last = ecb;
10260                 probe->dtpr_predcache = 0;

10262                 dtrace_sync();
10263                 return (0);
10264         }
10265 }

10267 static void
10268 dtrace_ecb_resize(dtrace_ecb_t *ecb)
10269 {
10270         dtrace_action_t *act;
10271         uint32_t curneeded = UINT32_MAX;
10272         uint32_t aggbase = UINT32_MAX;

10274         /*
10275          * If we record anything, we always record the dtrace_rechdr_t.  (And
10276          * we always record it first.)
10277          */
10278         ecb->dte_size = sizeof (dtrace_rechdr_t);
10279         ecb->dte_alignment = sizeof (dtrace_epid_t);

10281         for (act = ecb->dte_action; act != NULL; act = act->dta_next) {
10282                 dtrace_recdesc_t *rec = &act->dta_rec;
10283                 ASSERT(rec->dtrd_size > 0 || rec->dtrd_alignment == 1);

10285                 ecb->dte_alignment = MAX(ecb->dte_alignment,
10286                     rec->dtrd_alignment);

10288                 if (DTRACEACT_ISAGG(act->dta_kind)) {
10289                         dtrace_aggregation_t *agg = (dtrace_aggregation_t *)act;
```

```
10291                         ASSERT(rec->dtrd_size != 0);
10292                         ASSERT(agg->dtag_first != NULL);
10293                         ASSERT(act->dta_prev->dta_intuple);
10294                         ASSERT(aggbase != UINT32_MAX);
10295                         ASSERT(curneeded != UINT32_MAX);

10297                         agg->dtag_base = aggbase;

10299                         curneeded = P2ROUNDUP(curneeded, rec->dtrd_alignment);
10300                         rec->dtrd_offset = curneeded;
10301                         curneeded += rec->dtrd_size;
10302                         ecb->dte_needed = MAX(ecb->dte_needed, curneeded);

10304                         aggbase = UINT32_MAX;
10305                         curneeded = UINT32_MAX;
10306                 } else if (act->dta_intuple) {
10307                         if (curneeded == UINT32_MAX) {
10308                                 /*
10309                                  * This is the first record in a tuple.  Align
10310                                  * curneeded to be at offset 4 in an 8-byte
10311                                  * aligned block.
10312                                  */
10313                                 ASSERT(act->dta_prev == NULL ||
10314                                     !act->dta_prev->dta_intuple);
10315                                 ASSERT3U(aggbase, ==, UINT32_MAX);
10316                                 curneeded = P2PHASEUP(ecb->dte_size,
10317                                     sizeof (uint64_t), sizeof (dtrace_aggid_t));

10319                                 aggbase = curneeded - sizeof (dtrace_aggid_t);
10320                                 ASSERT(IS_P2ALIGNED(aggbase,
10321                                     sizeof (uint64_t)));
10322                         }
10323                         curneeded = P2ROUNDUP(curneeded, rec->dtrd_alignment);
10324                         rec->dtrd_offset = curneeded;
10325                         curneeded += rec->dtrd_size;
10326                 } else {
10327                         /* tuples must be followed by an aggregation */
10328                         ASSERT(act->dta_prev == NULL ||
10329                             !act->dta_prev->dta_intuple);

10331                         ecb->dte_size = P2ROUNDUP(ecb->dte_size,
10332                             rec->dtrd_alignment);
10333                         rec->dtrd_offset = ecb->dte_size;
10334                         ecb->dte_size += rec->dtrd_size;
10335                         ecb->dte_needed = MAX(ecb->dte_needed, ecb->dte_size);
10336                 }
10337         }

10339         if ((act = ecb->dte_action) != NULL &&
10340             !(act->dta_kind == DTRACEACT_SPECULATE && act->dta_next == NULL) &&
10341             ecb->dte_size == sizeof (dtrace_rechdr_t)) {
10342                 /*
10343                  * If the size is still sizeof (dtrace_rechdr_t), then all
10344                  * actions store no data; set the size to 0.
10345                  */
10346                 ecb->dte_size = 0;
10347         }

10349         ecb->dte_size = P2ROUNDUP(ecb->dte_size, sizeof (dtrace_epid_t));
10350         ecb->dte_needed = P2ROUNDUP(ecb->dte_needed, (sizeof (dtrace_epid_t)));
10351         ecb->dte_state->dts_needed = MAX(ecb->dte_state->dts_needed,
10352             ecb->dte_needed);
10353 }

10355 static dtrace_action_t *
10356 dtrace_ecb_aggregation_create(dtrace_ecb_t *ecb, dtrace_actdesc_t *desc)
```



```
10357 {
10358         dtrace_aggregation_t *agg;
10359         size_t size = sizeof (uint64_t);
10360         int ntuple = desc->dtad_ntuple;
10361         dtrace_action_t *act;
10362         dtrace_recdesc_t *frec;
10363         dtrace_aggid_t aggid;
10364         dtrace_state_t *state = ecb->dte_state;

10366         agg = kmem_zalloc(sizeof (dtrace_aggregation_t), KM_SLEEP);
10367         agg->dtag_ecb = ecb;

10369         ASSERT(DTRACEACT_ISAGG(desc->dtad_kind));

10371         switch (desc->dtad_kind) {
10372         case DTRACEAGG_MIN:
10373                 agg->dtag_initial = INT64_MAX;
10374                 agg->dtag_aggregate = dtrace_aggregate_min;
10375                 break;

10377         case DTRACEAGG_MAX:
10378                 agg->dtag_initial = INT64_MIN;
10379                 agg->dtag_aggregate = dtrace_aggregate_max;
10380                 break;

10382         case DTRACEAGG_COUNT:
10383                 agg->dtag_aggregate = dtrace_aggregate_count;
10384                 break;

10386         case DTRACEAGG_QUANTIZE:
10387                 agg->dtag_aggregate = dtrace_aggregate_quantize;
10388                 size = (((sizeof (uint64_t) * NBBY) - 1) * 2 + 1) *
10389                     sizeof (uint64_t);
10390                 break;

10392         case DTRACEAGG_LQUANTIZE: {
10393                 uint16_t step = DTRACE_LQUANTIZE_STEP(desc->dtad_arg);
10394                 uint16_t levels = DTRACE_LQUANTIZE_LEVELS(desc->dtad_arg);

10396                 agg->dtag_initial = desc->dtad_arg;
10397                 agg->dtag_aggregate = dtrace_aggregate_lquantize;

10399                 if (step == 0 || levels == 0)
10400                         goto err;

10402                 size = levels * sizeof (uint64_t) + 3 * sizeof (uint64_t);
10403                 break;
10404         }

10406         case DTRACEAGG_LLQUANTIZE: {
10407                 uint16_t factor = DTRACE_LLQUANTIZE_FACTOR(desc->dtad_arg);
10408                 uint16_t low = DTRACE_LLQUANTIZE_LOW(desc->dtad_arg);
10409                 uint16_t high = DTRACE_LLQUANTIZE_HIGH(desc->dtad_arg);
10410                 uint16_t nsteps = DTRACE_LLQUANTIZE_NSTEP(desc->dtad_arg);
10411                 int64_t v;

10413                 agg->dtag_initial = desc->dtad_arg;
10414                 agg->dtag_aggregate = dtrace_aggregate_llquantize;

10416                 if (factor < 2 || low >= high || nsteps < factor)
10417                         goto err;

10419                 /*
10420                  * Now check that the number of steps evenly divides a power
10421                  * of the factor.  (This assures both integer bucket size and
10422                  * linearity within each magnitude.)
```

```
10423                  */
10424                 for (v = factor; v < nsteps; v *= factor)
10425                         continue;
10426
10427                 if ((v % nsteps) || (nsteps % factor))
10428                         goto err;
10429
10430                 size = (dtrace_aggregate_llquantize_bucket(factor,
10431                     low, high, nsteps, INT64_MAX) + 2) * sizeof (uint64_t);
10432                 break;
10433         }
10434
10435         case DTRACEAGG_AVG:
10436                 agg->dtag_aggregate = dtrace_aggregate_avg;
10437                 size = sizeof (uint64_t) * 2;
10438                 break;
10439
10440         case DTRACEAGG_STDDEV:
10441                 agg->dtag_aggregate = dtrace_aggregate_stddev;
10442                 size = sizeof (uint64_t) * 4;
10443                 break;
10444
10445         case DTRACEAGG_SUM:
10446                 agg->dtag_aggregate = dtrace_aggregate_sum;
10447                 break;
10448
10449         default:
10450                 goto err;
10451         }
10452
10453         agg->dtag_action.dta_rec.dtrd_size = size;
10454
10455         if (ntuple == 0)
10456                 goto err;
10457
10458         /*
10459          * We must make sure that we have enough actions for the n-tuple.
10460          */
10461         for (act = ecb->dte_action_last; act != NULL; act = act->dta_prev) {
10462                 if (DTRACEACT_ISAGG(act->dta_kind))
10463                         break;
10464
10465                 if (--ntuple == 0) {
10466                         /*
10467                          * This is the action with which our n-tuple begins.
10468                          */
10469                         agg->dtag_first = act;
10470                         goto success;
10471                 }
10472         }
10473
10474         /*
10475          * This n-tuple is short by ntuple elements.  Return failure.
10476          */
10477         ASSERT(ntuple != 0);
10478 err:
10479         kmem_free(agg, sizeof (dtrace_aggregation_t));
10480         return (NULL);
10481
10482 success:
10483         /*
10484          * If the last action in the tuple has a size of zero, it's actually
10485          * an expression argument for the aggregating action.
10486          */
10487         ASSERT(ecb->dte_action_last != NULL);
10488         act = ecb->dte_action_last;
```



```
10490         if (act->dta_kind == DTRACEACT_DIFEXPR) {
10491                 ASSERT(act->dta_difo != NULL);
10492
10493                 if (act->dta_difo->dtdo_rtype.dtdt_size == 0)
10494                         agg->dtag_hasarg = 1;
10495         }
10496
10497         /*
10498          * We need to allocate an id for this aggregation.
10499          */
10500         aggid = (dtrace_aggid_t)(uintptr_t)vmem_alloc(state->dts_aggid_arena, 1,
10501             VM_BESTFIT | VM_SLEEP);
10502
10503         if (aggid - 1 >= state->dts_naggregations) {
10504                 dtrace_aggregation_t **oaggs = state->dts_aggregations;
10505                 dtrace_aggregation_t **aggs;
10506                 int naggs = state->dts_naggregations << 1;
10507                 int onaggs = state->dts_naggregations;
10508
10509                 ASSERT(aggid == state->dts_naggregations + 1);
10510
10511                 if (naggs == 0) {
10512                         ASSERT(oaggs == NULL);
10513                         naggs = 1;
10514                 }
10515
10516                 aggs = kmem_zalloc(naggs * sizeof (*aggs), KM_SLEEP);
10517
10518                 if (oaggs != NULL) {
10519                         bcopy(oaggs, aggs, onaggs * sizeof (*aggs));
10520                         kmem_free(oaggs, onaggs * sizeof (*aggs));
10521                 }
10522
10523                 state->dts_aggregations = aggs;
10524                 state->dts_naggregations = naggs;
10525         }
10526
10527         ASSERT(state->dts_aggregations[aggid - 1] == NULL);
10528         state->dts_aggregations[(agg->dtag_id = aggid) - 1] = agg;
10529
10530         frec = &agg->dtag_first->dta_rec;
10531         if (frec->dtrd_alignment < sizeof (dtrace_aggid_t))
10532                 frec->dtrd_alignment = sizeof (dtrace_aggid_t);
10533
10534         for (act = agg->dtag_first; act != NULL; act = act->dta_next) {
10535                 ASSERT(!act->dta_intuple);
10536                 act->dta_intuple = 1;
10537         }
10538
10539         return (&agg->dtag_action);
10540 }
10541
10542 static void
10543 dtrace_ecb_aggregation_destroy(dtrace_ecb_t *ecb, dtrace_action_t *act)
10544 {
10545         dtrace_aggregation_t *agg = (dtrace_aggregation_t *)act;
10546         dtrace_state_t *state = ecb->dte_state;
10547         dtrace_aggid_t aggid = agg->dtag_id;
10548
10549         ASSERT(DTRACEACT_ISAGG(act->dta_kind));
10550         vmem_free(state->dts_aggid_arena, (void *)(uintptr_t)aggid, 1);
10551
10552         ASSERT(state->dts_aggregations[aggid - 1] == agg);
10553         state->dts_aggregations[aggid - 1] = NULL;
```

```
10555         kmem_free(agg, sizeof (dtrace_aggregation_t));
10556 }

10558 static int
10559 dtrace_ecb_action_add(dtrace_ecb_t *ecb, dtrace_actdesc_t *desc)
10560 {
10561         dtrace_action_t *action, *last;
10562         dtrace_difo_t *dp = desc->dtad_difo;
10563         uint32_t size = 0, align = sizeof (uint8_t), mask;
10564         uint16_t format = 0;
10565         dtrace_recdesc_t *rec;
10566         dtrace_state_t *state = ecb->dte_state;
10567         dtrace_optval_t *opt = state->dts_options, nframes, strsize;
10568         uint64_t arg = desc->dtad_arg;

10570         ASSERT(MUTEX_HELD(&dtrace_lock));
10571         ASSERT(ecb->dte_action == NULL || ecb->dte_action->dta_refcnt == 1);

10573         if (DTRACEACT_ISAGG(desc->dtad_kind)) {
10574                 /*
10575                  * If this is an aggregating action, there must be neither
10576                  * a speculate nor a commit on the action chain.
10577                  */
10578                 dtrace_action_t *act;

10580                 for (act = ecb->dte_action; act != NULL; act = act->dta_next) {
10581                         if (act->dta_kind == DTRACEACT_COMMIT)
10582                                 return (EINVAL);

10584                         if (act->dta_kind == DTRACEACT_SPECULATE)
10585                                 return (EINVAL);
10586                 }

10588                 action = dtrace_ecb_aggregation_create(ecb, desc);

10590                 if (action == NULL)
10591                         return (EINVAL);
10592         } else {
10593                 if (DTRACEACT_ISDESTRUCTIVE(desc->dtad_kind) ||
10594                     (desc->dtad_kind == DTRACEACT_DIFEXPR &&
10595                     dp != NULL && dp->dtdo_destructive)) {
10596                         state->dts_destructive = 1;
10597                 }

10599                 switch (desc->dtad_kind) {
10600                 case DTRACEACT_PRINTF:
10601                 case DTRACEACT_PRINTA:
10602                 case DTRACEACT_SYSTEM:
10603                 case DTRACEACT_FREOPEN:
10604                 case DTRACEACT_DIFEXPR:
10605                         /*
10606                          * We know that our arg is a string -- turn it into a
10607                          * format.
10608                          */
10609                         if (arg == NULL) {
10610                                 ASSERT(desc->dtad_kind == DTRACEACT_PRINTA ||
10611                                     desc->dtad_kind == DTRACEACT_DIFEXPR);
10612                                 format = 0;
10613                         } else {
10614                                 ASSERT(arg != NULL);
10615                                 ASSERT(arg > KERNELBASE);
10616                                 format = dtrace_format_add(state,
10617                                     (char *)(uintptr_t)arg);
10618                         }

10620                         /*FALLTHROUGH*/
```



```
10621                 case DTRACEACT_LIBACT:
10622                 case DTRACEACT_TRACEMEM:
10623                 case DTRACEACT_TRACEMEM_DYNSIZE:
10624                         if (dp == NULL)
10625                                 return (EINVAL);

10627                         if ((size = dp->dtdo_rtype.dtdt_size) != 0)
10628                                 break;

10630                         if (dp->dtdo_rtype.dtdt_kind == DIF_TYPE_STRING) {
10631                                 if (!(dp->dtdo_rtype.dtdt_flags & DIF_TF_BYREF))
10632                                         return (EINVAL);

10634                                 size = opt[DTRACEOPT_STRSIZE];
10635                         }

10637                         break;

10639                 case DTRACEACT_STACK:
10640                         if ((nframes = arg) == 0) {
10641                                 nframes = opt[DTRACEOPT_STACKFRAMES];
10642                                 ASSERT(nframes > 0);
10643                                 arg = nframes;
10644                         }

10646                         size = nframes * sizeof (pc_t);
10647                         break;

10649                 case DTRACEACT_JSTACK:
10650                         if ((strsize = DTRACE_USTACK_STRSIZE(arg)) == 0)
10651                                 strsize = opt[DTRACEOPT_JSTACKSTRSIZE];

10653                         if ((nframes = DTRACE_USTACK_NFRAMES(arg)) == 0)
10654                                 nframes = opt[DTRACEOPT_JSTACKFRAMES];

10656                         arg = DTRACE_USTACK_ARG(nframes, strsize);

10658                         /*FALLTHROUGH*/
10659                 case DTRACEACT_USTACK:
10660                         if (desc->dtad_kind != DTRACEACT_JSTACK &&
10661                             (nframes = DTRACE_USTACK_NFRAMES(arg)) == 0) {
10662                                 strsize = DTRACE_USTACK_STRSIZE(arg);
10663                                 nframes = opt[DTRACEOPT_USTACKFRAMES];
10664                                 ASSERT(nframes > 0);
10665                                 arg = DTRACE_USTACK_ARG(nframes, strsize);
10666                         }

10668                         /*
10669                          * Save a slot for the pid.
10670                          */
10671                         size = (nframes + 1) * sizeof (uint64_t);
10672                         size += DTRACE_USTACK_STRSIZE(arg);
10673                         size = P2ROUNDUP(size, (uint32_t)(sizeof (uintptr_t)));

10675                         break;

10677                 case DTRACEACT_SYM:
10678                 case DTRACEACT_MOD:
10679                         if (dp == NULL || ((size = dp->dtdo_rtype.dtdt_size) !=
10680                             sizeof (uint64_t)) ||
10681                             (dp->dtdo_rtype.dtdt_flags & DIF_TF_BYREF))
10682                                 return (EINVAL);
10683                         break;

10685                 case DTRACEACT_USYM:
10686                 case DTRACEACT_UMOD:
```

```
10687                 case DTRACEACT_UADDR:
10688                         if (dp == NULL ||
10689                             (dp->dtdo_rtype.dtdt_size != sizeof (uint64_t)) ||
10690                             (dp->dtdo_rtype.dtdt_flags & DIF_TF_BYREF))
10691                                 return (EINVAL);
10692 
10693                         /*
10694                          * We have a slot for the pid, plus a slot for the
10695                          * argument.  To keep things simple (aligned with
10696                          * bitness-neutral sizing), we store each as a 64-bit
10697                          * quantity.
10698                          */
10699                         size = 2 * sizeof (uint64_t);
10700                         break;
10701 
10702                 case DTRACEACT_STOP:
10703                 case DTRACEACT_BREAKPOINT:
10704                 case DTRACEACT_PANIC:
10705                         break;
10706 
10707                 case DTRACEACT_CHILL:
10708                 case DTRACEACT_DISCARD:
10709                 case DTRACEACT_RAISE:
10710                         if (dp == NULL)
10711                                 return (EINVAL);
10712                         break;
10713 
10714                 case DTRACEACT_EXIT:
10715                         if (dp == NULL ||
10716                             (size = dp->dtdo_rtype.dtdt_size) != sizeof (int) ||
10717                             (dp->dtdo_rtype.dtdt_flags & DIF_TF_BYREF))
10718                                 return (EINVAL);
10719                         break;
10720 
10721                 case DTRACEACT_SPECULATE:
10722                         if (ecb->dte_size > sizeof (dtrace_rechdr_t))
10723                                 return (EINVAL);
10724 
10725                         if (dp == NULL)
10726                                 return (EINVAL);
10727 
10728                         state->dts_speculates = 1;
10729                         break;
10730 
10731                 case DTRACEACT_COMMIT: {
10732                         dtrace_action_t *act = ecb->dte_action;
10733 
10734                         for (; act != NULL; act = act->dta_next) {
10735                                 if (act->dta_kind == DTRACEACT_COMMIT)
10736                                         return (EINVAL);
10737                         }
10738 
10739                         if (dp == NULL)
10740                                 return (EINVAL);
10741                         break;
10742                 }
10743 
10744                 default:
10745                         return (EINVAL);
10746                 }
10747 
10748                 if (size != 0 || desc->dtad_kind == DTRACEACT_SPECULATE) {
10749                         /*
10750                          * If this is a data-storing action or a speculate,
10751                          * we must be sure that there isn't a commit on the
10752                          * action chain.
```



```
10753                          */
10754                         dtrace_action_t *act = ecb->dte_action;
10755 
10756                         for (; act != NULL; act = act->dta_next) {
10757                                 if (act->dta_kind == DTRACEACT_COMMIT)
10758                                         return (EINVAL);
10759                         }
10760                 }
10761 
10762                 action = kmem_zalloc(sizeof (dtrace_action_t), KM_SLEEP);
10763                 action->dta_rec.dtrd_size = size;
10764         }
10765 
10766         action->dta_refcnt = 1;
10767         rec = &action->dta_rec;
10768         size = rec->dtrd_size;
10769 
10770         for (mask = sizeof (uint64_t) - 1; size != 0 && mask > 0; mask >>= 1) {
10771                 if (!(size & mask)) {
10772                         align = mask + 1;
10773                         break;
10774                 }
10775         }
10776 
10777         action->dta_kind = desc->dtad_kind;
10778 
10779         if ((action->dta_difo = dp) != NULL)
10780                 dtrace_difo_hold(dp);
10781 
10782         rec->dtrd_action = action->dta_kind;
10783         rec->dtrd_arg = arg;
10784         rec->dtrd_uarg = desc->dtad_uarg;
10785         rec->dtrd_alignment = (uint16_t)align;
10786         rec->dtrd_format = format;
10787 
10788         if ((last = ecb->dte_action_last) != NULL) {
10789                 ASSERT(ecb->dte_action != NULL);
10790                 action->dta_prev = last;
10791                 last->dta_next = action;
10792         } else {
10793                 ASSERT(ecb->dte_action == NULL);
10794                 ecb->dte_action = action;
10795         }
10796 
10797         ecb->dte_action_last = action;
10798 
10799         return (0);
10800 }
10801 
10802 static void
10803 dtrace_ecb_action_remove(dtrace_ecb_t *ecb)
10804 {
10805         dtrace_action_t *act = ecb->dte_action, *next;
10806         dtrace_vstate_t *vstate = &ecb->dte_state->dts_vstate;
10807         dtrace_difo_t *dp;
10808         uint16_t format;
10809 
10810         if (act != NULL && act->dta_refcnt > 1) {
10811                 ASSERT(act->dta_next == NULL || act->dta_next->dta_refcnt == 1);
10812                 act->dta_refcnt--;
10813         } else {
10814                 for (; act != NULL; act = next) {
10815                         next = act->dta_next;
10816                         ASSERT(next != NULL || act == ecb->dte_action_last);
10817                         ASSERT(act->dta_refcnt == 1);
```

```
10819                           if ((format = act->dta_rec.dtrd_format) != 0)
10820                                   dtrace_format_remove(ecb->dte_state, format);
10821
10822                           if ((dp = act->dta_difo) != NULL)
10823                                   dtrace_difo_release(dp, vstate);
10824
10825                           if (DTRACEACT_ISAGG(act->dta_kind)) {
10826                                   dtrace_ecb_aggregation_destroy(ecb, act);
10827                           } else {
10828                                   kmem_free(act, sizeof (dtrace_action_t));
10829                           }
10830                   }
10831           }
10832
10833           ecb->dte_action = NULL;
10834           ecb->dte_action_last = NULL;
10835           ecb->dte_size = 0;
10836   }
10837
10838   static void
10839   dtrace_ecb_disable(dtrace_ecb_t *ecb)
10840   {
10841           /*
10842            * We disable the ECB by removing it from its probe.
10843            */
10844           dtrace_ecb_t *pecb, *prev = NULL;
10845           dtrace_probe_t *probe = ecb->dte_probe;
10846
10847           ASSERT(MUTEX_HELD(&dtrace_lock));
10848
10849           if (probe == NULL) {
10850                   /*
10851                    * This is the NULL probe; there is nothing to disable.
10852                    */
10853                   return;
10854           }
10855
10856           for (pecb = probe->dtpr_ecb; pecb != NULL; pecb = pecb->dte_next) {
10857                   if (pecb == ecb)
10858                           break;
10859                   prev = pecb;
10860           }
10861
10862           ASSERT(pecb != NULL);
10863
10864           if (prev == NULL) {
10865                   probe->dtpr_ecb = ecb->dte_next;
10866           } else {
10867                   prev->dte_next = ecb->dte_next;
10868           }
10869
10870           if (ecb == probe->dtpr_ecb_last) {
10871                   ASSERT(ecb->dte_next == NULL);
10872                   probe->dtpr_ecb_last = prev;
10873           }
10874
10875           /*
10876            * The ECB has been disconnected from the probe; now sync to assure
10877            * that all CPUs have seen the change before returning.
10878            */
10879           dtrace_sync();
10880
10881           if (probe->dtpr_ecb == NULL) {
10882                   /*
10883                    * That was the last ECB on the probe; clear the predicate
10884                    * cache ID for the probe, disable it and sync one more time
```



```
10885                    * to assure that we'll never hit it again.
10886                    */
10887                   dtrace_provider_t *prov = probe->dtpr_provider;
10888
10889                   ASSERT(ecb->dte_next == NULL);
10890                   ASSERT(probe->dtpr_ecb_last == NULL);
10891                   probe->dtpr_predcache = DTRACE_CACHEIDNONE;
10892                   prov->dtpv_pops.dtps_disable(prov->dtpv_arg,
10893                       probe->dtpr_id, probe->dtpr_arg);
10894                   dtrace_sync();
10895           } else {
10896                   /*
10897                    * There is at least one ECB remaining on the probe.  If there
10898                    * is _exactly_ one, set the probe's predicate cache ID to be
10899                    * the predicate cache ID of the remaining ECB.
10900                    */
10901                   ASSERT(probe->dtpr_ecb_last != NULL);
10902                   ASSERT(probe->dtpr_predcache == DTRACE_CACHEIDNONE);
10903
10904                   if (probe->dtpr_ecb == probe->dtpr_ecb_last) {
10905                           dtrace_predicate_t *p = probe->dtpr_ecb->dte_predicate;
10906
10907                           ASSERT(probe->dtpr_ecb->dte_next == NULL);
10908
10909                           if (p != NULL)
10910                                   probe->dtpr_predcache = p->dtp_cacheid;
10911                   }
10912
10913                   ecb->dte_next = NULL;
10914           }
10915   }
10916
10917   static void
10918   dtrace_ecb_destroy(dtrace_ecb_t *ecb)
10919   {
10920           dtrace_state_t *state = ecb->dte_state;
10921           dtrace_vstate_t *vstate = &state->dts_vstate;
10922           dtrace_predicate_t *pred;
10923           dtrace_epid_t epid = ecb->dte_epid;
10924
10925           ASSERT(MUTEX_HELD(&dtrace_lock));
10926           ASSERT(ecb->dte_next == NULL);
10927           ASSERT(ecb->dte_probe == NULL || ecb->dte_probe->dtpr_ecb != ecb);
10928
10929           if ((pred = ecb->dte_predicate) != NULL)
10930                   dtrace_predicate_release(pred, vstate);
10931
10932           dtrace_ecb_action_remove(ecb);
10933
10934           ASSERT(state->dts_ecbs[epid - 1] == ecb);
10935           state->dts_ecbs[epid - 1] = NULL;
10936
10937           kmem_free(ecb, sizeof (dtrace_ecb_t));
10938   }
10939
10940   static dtrace_ecb_t *
10941   dtrace_ecb_create(dtrace_state_t *state, dtrace_probe_t *probe,
10942       dtrace_enabling_t *enab)
10943   {
10944           dtrace_ecb_t *ecb;
10945           dtrace_predicate_t *pred;
10946           dtrace_actdesc_t *act;
10947           dtrace_provider_t *prov;
10948           dtrace_ecbdesc_t *desc = enab->dten_current;
10949
10950           ASSERT(MUTEX_HELD(&dtrace_lock));
```

```
10951         ASSERT(state != NULL);

10953         ecb = dtrace_ecb_add(state, probe);
10954         ecb->dte_uarg = desc->dted_uarg;

10956         if ((pred = desc->dted_pred.dtpdd_predicate) != NULL) {
10957                 dtrace_predicate_hold(pred);
10958                 ecb->dte_predicate = pred;
10959         }

10961         if (probe != NULL) {
10962                 /*
10963                  * If the provider shows more leg than the consumer is old
10964                  * enough to see, we need to enable the appropriate implicit
10965                  * predicate bits to prevent the ecb from activating at
10966                  * revealing times.
10967                  *
10968                  * Providers specifying DTRACE_PRIV_USER at register time
10969                  * are stating that they need the /proc-style privilege
10970                  * model to be enforced, and this is what DTRACE_COND_OWNER
10971                  * and DTRACE_COND_ZONEOWNER will then do at probe time.
10972                  */
10973                 prov = probe->dtpr_provider;
10974                 if (!(state->dts_cred.dcr_visible & DTRACE_CRV_ALLPROC) &&
10975                     (prov->dtpv_priv.dtpp_flags & DTRACE_PRIV_USER))
10976                         ecb->dte_cond |= DTRACE_COND_OWNER;

10978                 if (!(state->dts_cred.dcr_visible & DTRACE_CRV_ALLZONE) &&
10979                     (prov->dtpv_priv.dtpp_flags & DTRACE_PRIV_USER))
10980                         ecb->dte_cond |= DTRACE_COND_ZONEOWNER;

10982                 /*
10983                  * If the provider shows us kernel innards and the user
10984                  * is lacking sufficient privilege, enable the
10985                  * DTRACE_COND_USERMODE implicit predicate.
10986                  */
10987                 if (!(state->dts_cred.dcr_visible & DTRACE_CRV_KERNEL) &&
10988                     (prov->dtpv_priv.dtpp_flags & DTRACE_PRIV_KERNEL))
10989                         ecb->dte_cond |= DTRACE_COND_USERMODE;
10990         }

10992         if (dtrace_ecb_create_cache != NULL) {
10993                 /*
10994                  * If we have a cached ecb, we'll use its action list instead
10995                  * of creating our own (saving both time and space).
10996                  */
10997                 dtrace_ecb_t *cached = dtrace_ecb_create_cache;
10998                 dtrace_action_t *act = cached->dte_action;

11000                 if (act != NULL) {
11001                         ASSERT(act->dta_refcnt > 0);
11002                         act->dta_refcnt++;
11003                         ecb->dte_action = act;
11004                         ecb->dte_action_last = cached->dte_action_last;
11005                         ecb->dte_needed = cached->dte_needed;
11006                         ecb->dte_size = cached->dte_size;
11007                         ecb->dte_alignment = cached->dte_alignment;
11008                 }

11010                 return (ecb);
11011         }

11013         for (act = desc->dted_action; act != NULL; act = act->dtad_next) {
11014                 if ((enab->dten_error = dtrace_ecb_action_add(ecb, act)) != 0) {
11015                         dtrace_ecb_destroy(ecb);
11016                         return (NULL);
```



```
11017                 }
11018         }

11020         dtrace_ecb_resize(ecb);

11022         return (dtrace_ecb_create_cache = ecb);
11023 }

11025 static int
11026 dtrace_ecb_create_enable(dtrace_probe_t *probe, void *arg)
11027 {
11028         dtrace_ecb_t *ecb;
11029         dtrace_enabling_t *enab = arg;
11030         dtrace_state_t *state = enab->dten_vstate->dtvs_state;

11032         ASSERT(state != NULL);

11034         if (probe != NULL && probe->dtpr_gen < enab->dten_probegen) {
11035                 /*
11036                  * This probe was created in a generation for which this
11037                  * enabling has previously created ECBs; we don't want to
11038                  * enable it again, so just kick out.
11039                  */
11040                 return (DTRACE_MATCH_NEXT);
11041         }

11043         if ((ecb = dtrace_ecb_create(state, probe, enab)) == NULL)
11044                 return (DTRACE_MATCH_DONE);

11046         if (dtrace_ecb_enable(ecb) < 0)
11047                 return (DTRACE_MATCH_FAIL);

11049         return (DTRACE_MATCH_NEXT);
11050 }

11052 static dtrace_ecb_t *
11053 dtrace_epid2ecb(dtrace_state_t *state, dtrace_epid_t id)
11054 {
11055         dtrace_ecb_t *ecb;

11057         ASSERT(MUTEX_HELD(&dtrace_lock));

11059         if (id == 0 || id > state->dts_necbs)
11060                 return (NULL);

11062         ASSERT(state->dts_necbs > 0 && state->dts_ecbs != NULL);
11063         ASSERT((ecb = state->dts_ecbs[id - 1]) == NULL || ecb->dte_epid == id);

11065         return (state->dts_ecbs[id - 1]);
11066 }

11068 static dtrace_aggregation_t *
11069 dtrace_aggid2agg(dtrace_state_t *state, dtrace_aggid_t id)
11070 {
11071         dtrace_aggregation_t *agg;

11073         ASSERT(MUTEX_HELD(&dtrace_lock));

11075         if (id == 0 || id > state->dts_naggregations)
11076                 return (NULL);

11078         ASSERT(state->dts_naggregations > 0 && state->dts_aggregations != NULL);
11079         ASSERT((agg = state->dts_aggregations[id - 1]) == NULL ||
11080             agg->dtag_id == id);

11082         return (state->dts_aggregations[id - 1]);
```

```
11083 }

11085 /*
11086  * DTrace Buffer Functions
11087  *
11088  * The following functions manipulate DTrace buffers.  Most of these functions
11089  * are called in the context of establishing or processing consumer state;
11090  * exceptions are explicitly noted.
11091  */

11093 /*
11094  * Note:  called from cross call context.  This function switches the two
11095  * buffers on a given CPU.  The atomicity of this operation is assured by
11096  * disabling interrupts while the actual switch takes place; the disabling of
11097  * interrupts serializes the execution with any execution of dtrace_probe() on
11098  * the same CPU.
11099  */
11100 static void
11101 dtrace_buffer_switch(dtrace_buffer_t *buf)
11102 {
11103         caddr_t tomax = buf->dtb_tomax;
11104         caddr_t xamot = buf->dtb_xamot;
11105         dtrace_icookie_t cookie;
11106         hrtime_t now;

11108         ASSERT(!(buf->dtb_flags & DTRACEBUF_NOSWITCH));
11109         ASSERT(!(buf->dtb_flags & DTRACEBUF_RING));

11111         cookie = dtrace_interrupt_disable();
11112         now = dtrace_gethrtime();
11113         buf->dtb_tomax = xamot;
11114         buf->dtb_xamot = tomax;
11115         buf->dtb_xamot_drops = buf->dtb_drops;
11116         buf->dtb_xamot_offset = buf->dtb_offset;
11117         buf->dtb_xamot_errors = buf->dtb_errors;
11118         buf->dtb_xamot_flags = buf->dtb_flags;
11119         buf->dtb_offset = 0;
11120         buf->dtb_drops = 0;
11121         buf->dtb_errors = 0;
11122         buf->dtb_flags &= ~(DTRACEBUF_ERROR | DTRACEBUF_DROPPED);
11123         buf->dtb_interval = now - buf->dtb_switched;
11124         buf->dtb_switched = now;
11125         dtrace_interrupt_enable(cookie);
11126 }

11128 /*
11129  * Note:  called from cross call context.  This function activates a buffer
11130  * on a CPU.  As with dtrace_buffer_switch(), the atomicity of the operation
11131  * is guaranteed by the disabling of interrupts.
11132  */
11133 static void
11134 dtrace_buffer_activate(dtrace_state_t *state)
11135 {
11136         dtrace_buffer_t *buf;
11137         dtrace_icookie_t cookie = dtrace_interrupt_disable();

11139         buf = &state->dts_buffer[CPU->cpu_id];

11141         if (buf->dtb_tomax != NULL) {
11142                 /*
11143                  * We might like to assert that the buffer is marked inactive,
11144                  * but this isn't necessarily true:  the buffer for the CPU
11145                  * that processes the BEGIN probe has its buffer activated
11146                  * manually.  In this case, we take the (harmless) action
11147                  * re-clearing the bit INACTIVE bit.
11148                  */
```



```
11149                 buf->dtb_flags &= ~DTRACEBUF_INACTIVE;
11150         }

11152         dtrace_interrupt_enable(cookie);
11153 }

11155 static int
11156 dtrace_buffer_alloc(dtrace_buffer_t *bufs, size_t size, int flags,
11157     processorid_t cpu, int *factor)
11158 {
11159         cpu_t *cp;
11160         dtrace_buffer_t *buf;
11161         int allocated = 0, desired = 0;

11163         ASSERT(MUTEX_HELD(&cpu_lock));
11164         ASSERT(MUTEX_HELD(&dtrace_lock));

11166         *factor = 1;

11168         if (size > dtrace_nonroot_maxsize &&
11169             !PRIV_POLICY_CHOICE(CRED(), PRIV_ALL, B_FALSE))
11170                 return (EFBIG);

11172         cp = cpu_list;

11174         do {
11175                 if (cpu != DTRACE_CPUALL && cpu != cp->cpu_id)
11176                         continue;

11178                 buf = &bufs[cp->cpu_id];

11180                 /*
11181                  * If there is already a buffer allocated for this CPU, it
11182                  * is only possible that this is a DR event.  In this case,
11183                  * the buffer size must match our specified size.
11184                  */
11185                 if (buf->dtb_tomax != NULL) {
11186                         ASSERT(buf->dtb_size == size);
11187                         continue;
11188                 }

11190                 ASSERT(buf->dtb_xamot == NULL);

11192                 if ((buf->dtb_tomax = kmem_zalloc(size,
11193                     KM_NOSLEEP | KM_NORMALPRI)) == NULL)
11194                         goto err;

11196                 buf->dtb_size = size;
11197                 buf->dtb_flags = flags;
11198                 buf->dtb_offset = 0;
11199                 buf->dtb_drops = 0;

11201                 if (flags & DTRACEBUF_NOSWITCH)
11202                         continue;

11204                 if ((buf->dtb_xamot = kmem_zalloc(size,
11205                     KM_NOSLEEP | KM_NORMALPRI)) == NULL)
11206                         goto err;
11207         } while ((cp = cp->cpu_next) != cpu_list);

11209         return (0);

11211 err:
11212         cp = cpu_list;

11214         do {
```

```
11215                 if (cpu != DTRACE_CPUALL && cpu != cp->cpu_id)
11216                         continue;

11218                 buf = &bufs[cp->cpu_id];
11219                 desired += 2;

11221                 if (buf->dtb_xamot != NULL) {
11222                         ASSERT(buf->dtb_tomax != NULL);
11223                         ASSERT(buf->dtb_size == size);
11224                         kmem_free(buf->dtb_xamot, size);
11225                         allocated++;
11226                 }

11228                 if (buf->dtb_tomax != NULL) {
11229                         ASSERT(buf->dtb_size == size);
11230                         kmem_free(buf->dtb_tomax, size);
11231                         allocated++;
11232                 }

11234                 buf->dtb_tomax = NULL;
11235                 buf->dtb_xamot = NULL;
11236                 buf->dtb_size = 0;
11237         } while ((cp = cp->cpu_next) != cpu_list);

11239         *factor = desired / (allocated > 0 ? allocated : 1);

11241         return (ENOMEM);
11242 }

11244 /*
11245  * Note:  called from probe context.  This function just increments the drop
11246  * count on a buffer.  It has been made a function to allow for the
11247  * possibility of understanding the source of mysterious drop counts.  (A
11248  * problem for which one may be particularly disappointed that DTrace cannot
11249  * be used to understand DTrace.)
11250  */
11251 static void
11252 dtrace_buffer_drop(dtrace_buffer_t *buf)
11253 {
11254         buf->dtb_drops++;
11255 }

11257 /*
11258  * Note:  called from probe context.  This function is called to reserve space
11259  * in a buffer.  If mstate is non-NULL, sets the scratch base and size in the
11260  * mstate.  Returns the new offset in the buffer, or a negative value if an
11261  * error has occurred.
11262  */
11263 static intptr_t
11264 dtrace_buffer_reserve(dtrace_buffer_t *buf, size_t needed, size_t align,
11265     dtrace_state_t *state, dtrace_mstate_t *mstate)
11266 {
11267         intptr_t offs = buf->dtb_offset, soffs;
11268         intptr_t woffs;
11269         caddr_t tomax;
11270         size_t total;

11272         if (buf->dtb_flags & DTRACEBUF_INACTIVE)
11273                 return (-1);

11275         if ((tomax = buf->dtb_tomax) == NULL) {
11276                 dtrace_buffer_drop(buf);
11277                 return (-1);
11278         }

11280         if (!(buf->dtb_flags & (DTRACEBUF_RING | DTRACEBUF_FILL))) {
```



```
11281                 while (offs & (align - 1)) {
11282                         /*
11283                          * Assert that our alignment is off by a number which
11284                          * is itself sizeof (uint32_t) aligned.
11285                          */
11286                         ASSERT(!((align - (offs & (align - 1))) &
11287                             (sizeof (uint32_t) - 1)));
11288                         DTRACE_STORE(uint32_t, tomax, offs, DTRACE_EPIDNONE);
11289                         offs += sizeof (uint32_t);
11290                 }

11292                 if ((soffs = offs + needed) > buf->dtb_size) {
11293                         dtrace_buffer_drop(buf);
11294                         return (-1);
11295                 }

11297                 if (mstate == NULL)
11298                         return (offs);

11300                 mstate->dtms_scratch_base = (uintptr_t)tomax + soffs;
11301                 mstate->dtms_scratch_size = buf->dtb_size - soffs;
11302                 mstate->dtms_scratch_ptr = mstate->dtms_scratch_base;

11304                 return (offs);
11305         }

11307         if (buf->dtb_flags & DTRACEBUF_FILL) {
11308                 if (state->dts_activity != DTRACE_ACTIVITY_COOLDOWN &&
11309                     (buf->dtb_flags & DTRACEBUF_FULL))
11310                         return (-1);
11311                 goto out;
11312         }

11314         total = needed + (offs & (align - 1));

11316         /*
11317          * For a ring buffer, life is quite a bit more complicated.  Before
11318          * we can store any padding, we need to adjust our wrapping offset.
11319          * (If we've never before wrapped or we're not about to, no adjustment
11320          * is required.)
11321          */
11322         if ((buf->dtb_flags & DTRACEBUF_WRAPPED) ||
11323             offs + total > buf->dtb_size) {
11324                 woffs = buf->dtb_xamot_offset;

11326                 if (offs + total > buf->dtb_size) {
11327                         /*
11328                          * We can't fit in the end of the buffer.  First, a
11329                          * sanity check that we can fit in the buffer at all.
11330                          */
11331                         if (total > buf->dtb_size) {
11332                                 dtrace_buffer_drop(buf);
11333                                 return (-1);
11334                         }

11336                         /*
11337                          * We're going to be storing at the top of the buffer,
11338                          * so now we need to deal with the wrapped offset.  We
11339                          * only reset our wrapped offset to 0 if it is
11340                          * currently greater than the current offset.  If it
11341                          * is less than the current offset, it is because a
11342                          * previous allocation induced a wrap -- but the
11343                          * allocation didn't subsequently take the space due
11344                          * to an error or false predicate evaluation.  In this
11345                          * case, we'll just leave the wrapped offset alone: if
11346                          * the wrapped offset hasn't been advanced far enough
```

```
11347                          * for this allocation, it will be adjusted in the
11348                          * lower loop.
11349                          */
11350                         if (buf->dtb_flags & DTRACEBUF_WRAPPED) {
11351                                 if (woffs >= offs)
11352                                         woffs = 0;
11353                         } else {
11354                                 woffs = 0;
11355                         }
11356
11357                         /*
11358                          * Now we know that we're going to be storing to the
11359                          * top of the buffer and that there is room for us
11360                          * there.  We need to clear the buffer from the current
11361                          * offset to the end (there may be old gunk there).
11362                          */
11363                         while (offs < buf->dtb_size)
11364                                 tomax[offs++] = 0;
11365
11366                         /*
11367                          * We need to set our offset to zero.  And because we
11368                          * are wrapping, we need to set the bit indicating as
11369                          * much.  We can also adjust our needed space back
11370                          * down to the space required by the ECB -- we know
11371                          * that the top of the buffer is aligned.
11372                          */
11373                         offs = 0;
11374                         total = needed;
11375                         buf->dtb_flags |= DTRACEBUF_WRAPPED;
11376                 } else {
11377                         /*
11378                          * There is room for us in the buffer, so we simply
11379                          * need to check the wrapped offset.
11380                          */
11381                         if (woffs < offs) {
11382                                 /*
11383                                  * The wrapped offset is less than the offset.
11384                                  * This can happen if we allocated buffer space
11385                                  * that induced a wrap, but then we didn't
11386                                  * subsequently take the space due to an error
11387                                  * or false predicate evaluation.  This is
11388                                  * okay; we know that _this_ allocation isn't
11389                                  * going to induce a wrap.  We still can't
11390                                  * reset the wrapped offset to be zero,
11391                                  * however: the space may have been trashed in
11392                                  * the previous failed probe attempt.  But at
11393                                  * least the wrapped offset doesn't need to
11394                                  * be adjusted at all...
11395                                  */
11396                                 goto out;
11397                         }
11398                 }
11399
11400                 while (offs + total > woffs) {
11401                         dtrace_epid_t epid = *(uint32_t *)(tomax + woffs);
11402                         size_t size;
11403
11404                         if (epid == DTRACE_EPIDNONE) {
11405                                 size = sizeof (uint32_t);
11406                         } else {
11407                                 ASSERT3U(epid, <=, state->dts_necbs);
11408                                 ASSERT(state->dts_ecbs[epid - 1] != NULL);
11409
11410                                 size = state->dts_ecbs[epid - 1]->dte_size;
11411                         }
```



```
11413                         ASSERT(woffs + size <= buf->dtb_size);
11414                         ASSERT(size != 0);
11415
11416                         if (woffs + size == buf->dtb_size) {
11417                                 /*
11418                                  * We've reached the end of the buffer; we want
11419                                  * to set the wrapped offset to 0 and break
11420                                  * out.  However, if the offs is 0, then we're
11421                                  * in a strange edge-condition:  the amount of
11422                                  * space that we want to reserve plus the size
11423                                  * of the record that we're overwriting is
11424                                  * greater than the size of the buffer.  This
11425                                  * is problematic because if we reserve the
11426                                  * space but subsequently don't consume it (due
11427                                  * to a failed predicate or error) the wrapped
11428                                  * offset will be 0 -- yet the EPID at offset 0
11429                                  * will not be committed.  This situation is
11430                                  * relatively easy to deal with:  if we're in
11431                                  * this case, the buffer is indistinguishable
11432                                  * from one that hasn't wrapped; we need only
11433                                  * finish the job by clearing the wrapped bit,
11434                                  * explicitly setting the offset to be 0, and
11435                                  * zero'ing out the old data in the buffer.
11436                                  */
11437                                 if (offs == 0) {
11438                                         buf->dtb_flags &= ~DTRACEBUF_WRAPPED;
11439                                         buf->dtb_offset = 0;
11440                                         woffs = total;
11441
11442                                         while (woffs < buf->dtb_size)
11443                                                 tomax[woffs++] = 0;
11444                                 }
11445
11446                                 woffs = 0;
11447                                 break;
11448                         }
11449
11450                         woffs += size;
11451                 }
11452
11453                 /*
11454                  * We have a wrapped offset.  It may be that the wrapped offset
11455                  * has become zero -- that's okay.
11456                  */
11457                 buf->dtb_xamot_offset = woffs;
11458         }
11459
11460 out:
11461         /*
11462          * Now we can plow the buffer with any necessary padding.
11463          */
11464         while (offs & (align - 1)) {
11465                 /*
11466                  * Assert that our alignment is off by a number which
11467                  * is itself sizeof (uint32_t) aligned.
11468                  */
11469                 ASSERT(!((align - (offs & (align - 1))) &
11470                     (sizeof (uint32_t) - 1)));
11471                 DTRACE_STORE(uint32_t, tomax, offs, DTRACE_EPIDNONE);
11472                 offs += sizeof (uint32_t);
11473         }
11474
11475         if (buf->dtb_flags & DTRACEBUF_FILL) {
11476                 if (offs + needed > buf->dtb_size - state->dts_reserve) {
11477                         buf->dtb_flags |= DTRACEBUF_FULL;
11478                         return (-1);
```

```
11479                 }
11480         }

11482         if (mstate == NULL)
11483                 return (offs);

11485         /*
11486          * For ring buffers and fill buffers, the scratch space is always
11487          * the inactive buffer.
11488          */
11489         mstate->dtms_scratch_base = (uintptr_t)buf->dtb_xamot;
11490         mstate->dtms_scratch_size = buf->dtb_size;
11491         mstate->dtms_scratch_ptr = mstate->dtms_scratch_base;

11493         return (offs);
11494 }

11496 static void
11497 dtrace_buffer_polish(dtrace_buffer_t *buf)
11498 {
11499         ASSERT(buf->dtb_flags & DTRACEBUF_RING);
11500         ASSERT(MUTEX_HELD(&dtrace_lock));

11502         if (!(buf->dtb_flags & DTRACEBUF_WRAPPED))
11503                 return;

11505         /*
11506          * We need to polish the ring buffer.  There are three cases:
11507          *
11508          * - The first (and presumably most common) is that there is no gap
11509          *   between the buffer offset and the wrapped offset.  In this case,
11510          *   there is nothing in the buffer that isn't valid data; we can
11511          *   mark the buffer as polished and return.
11512          *
11513          * - The second (less common than the first but still more common
11514          *   than the third) is that there is a gap between the buffer offset
11515          *   and the wrapped offset, and the wrapped offset is larger than the
11516          *   buffer offset.  This can happen because of an alignment issue, or
11517          *   can happen because of a call to dtrace_buffer_reserve() that
11518          *   didn't subsequently consume the buffer space.  In this case,
11519          *   we need to zero the data from the buffer offset to the wrapped
11520          *   offset.
11521          *
11522          * - The third (and least common) is that there is a gap between the
11523          *   buffer offset and the wrapped offset, but the wrapped offset is
11524          *   _less_ than the buffer offset.  This can only happen because a
11525          *   call to dtrace_buffer_reserve() induced a wrap, but the space
11526          *   was not subsequently consumed.  In this case, we need to zero the
11527          *   space from the offset to the end of the buffer _and_ from the
11528          *   top of the buffer to the wrapped offset.
11529          */
11530         if (buf->dtb_offset < buf->dtb_xamot_offset) {
11531                 bzero(buf->dtb_tomax + buf->dtb_offset,
11532                     buf->dtb_xamot_offset - buf->dtb_offset);
11533         }

11535         if (buf->dtb_offset > buf->dtb_xamot_offset) {
11536                 bzero(buf->dtb_tomax + buf->dtb_offset,
11537                     buf->dtb_size - buf->dtb_offset);
11538                 bzero(buf->dtb_tomax, buf->dtb_xamot_offset);
11539         }
11540 }

11542 /*
11543  * This routine determines if data generated at the specified time has likely
11544  * been entirely consumed at user-level.  This routine is called to determine
```



```
11545  * if an ECB on a defunct probe (but for an active enabling) can be safely
11546  * disabled and destroyed.
11547  */
11548 static int
11549 dtrace_buffer_consumed(dtrace_buffer_t *bufs, hrtime_t when)
11550 {
11551         int i;

11553         for (i = 0; i < NCPU; i++) {
11554                 dtrace_buffer_t *buf = &bufs[i];

11556                 if (buf->dtb_size == 0)
11557                         continue;

11559                 if (buf->dtb_flags & DTRACEBUF_RING)
11560                         return (0);

11562                 if (!buf->dtb_switched && buf->dtb_offset != 0)
11563                         return (0);

11565                 if (buf->dtb_switched - buf->dtb_interval < when)
11566                         return (0);
11567         }

11569         return (1);
11570 }

11572 static void
11573 dtrace_buffer_free(dtrace_buffer_t *bufs)
11574 {
11575         int i;

11577         for (i = 0; i < NCPU; i++) {
11578                 dtrace_buffer_t *buf = &bufs[i];

11580                 if (buf->dtb_tomax == NULL) {
11581                         ASSERT(buf->dtb_xamot == NULL);
11582                         ASSERT(buf->dtb_size == 0);
11583                         continue;
11584                 }

11586                 if (buf->dtb_xamot != NULL) {
11587                         ASSERT(!(buf->dtb_flags & DTRACEBUF_NOSWITCH));
11588                         kmem_free(buf->dtb_xamot, buf->dtb_size);
11589                 }

11591                 kmem_free(buf->dtb_tomax, buf->dtb_size);
11592                 buf->dtb_size = 0;
11593                 buf->dtb_tomax = NULL;
11594                 buf->dtb_xamot = NULL;
11595         }
11596 }

11598 /*
11599  * DTrace Enabling Functions
11600  */
11601 static dtrace_enabling_t *
11602 dtrace_enabling_create(dtrace_vstate_t *vstate)
11603 {
11604         dtrace_enabling_t *enab;

11606         enab = kmem_zalloc(sizeof (dtrace_enabling_t), KM_SLEEP);
11607         enab->dten_vstate = vstate;

11609         return (enab);
11610 }
```

```
11612 static void
11613 dtrace_enabling_add(dtrace_enabling_t *enab, dtrace_ecbdesc_t *ecb)
11614 {
11615         dtrace_ecbdesc_t **ndesc;
11616         size_t osize, nsize;

11618         /*
11619          * We can't add to enablings after we've enabled them, or after we've
11620          * retained them.
11621          */
11622         ASSERT(enab->dten_probegen == 0);
11623         ASSERT(enab->dten_next == NULL && enab->dten_prev == NULL);

11625         if (enab->dten_ndesc < enab->dten_maxdesc) {
11626                 enab->dten_desc[enab->dten_ndesc++] = ecb;
11627                 return;
11628         }

11630         osize = enab->dten_maxdesc * sizeof (dtrace_enabling_t *);

11632         if (enab->dten_maxdesc == 0) {
11633                 enab->dten_maxdesc = 1;
11634         } else {
11635                 enab->dten_maxdesc <<= 1;
11636         }

11638         ASSERT(enab->dten_ndesc < enab->dten_maxdesc);

11640         nsize = enab->dten_maxdesc * sizeof (dtrace_enabling_t *);
11641         ndesc = kmem_zalloc(nsize, KM_SLEEP);
11642         bcopy(enab->dten_desc, ndesc, osize);
11643         kmem_free(enab->dten_desc, osize);

11645         enab->dten_desc = ndesc;
11646         enab->dten_desc[enab->dten_ndesc++] = ecb;
11647 }

11649 static void
11650 dtrace_enabling_addlike(dtrace_enabling_t *enab, dtrace_ecbdesc_t *ecb,
11651     dtrace_probedesc_t *pd)
11652 {
11653         dtrace_ecbdesc_t *new;
11654         dtrace_predicate_t *pred;
11655         dtrace_actdesc_t *act;

11657         /*
11658          * We're going to create a new ECB description that matches the
11659          * specified ECB in every way, but has the specified probe description.
11660          */
11661         new = kmem_zalloc(sizeof (dtrace_ecbdesc_t), KM_SLEEP);

11663         if ((pred = ecb->dted_pred.dtpdd_predicate) != NULL)
11664                 dtrace_predicate_hold(pred);

11666         for (act = ecb->dted_action; act != NULL; act = act->dtad_next)
11667                 dtrace_actdesc_hold(act);

11669         new->dted_action = ecb->dted_action;
11670         new->dted_pred = ecb->dted_pred;
11671         new->dted_probe = *pd;
11672         new->dted_uarg = ecb->dted_uarg;

11674         dtrace_enabling_add(enab, new);
11675 }
```



```
11677 static void
11678 dtrace_enabling_dump(dtrace_enabling_t *enab)
11679 {
11680         int i;

11682         for (i = 0; i < enab->dten_ndesc; i++) {
11683                 dtrace_probedesc_t *desc = &enab->dten_desc[i]->dted_probe;

11685                 cmn_err(CE_NOTE, "enabling probe %d (%s:%s:%s:%s)", i,
11686                     desc->dtpd_provider, desc->dtpd_mod,
11687                     desc->dtpd_func, desc->dtpd_name);
11688         }
11689 }

11691 static void
11692 dtrace_enabling_destroy(dtrace_enabling_t *enab)
11693 {
11694         int i;
11695         dtrace_ecbdesc_t *ep;
11696         dtrace_vstate_t *vstate = enab->dten_vstate;

11698         ASSERT(MUTEX_HELD(&dtrace_lock));

11700         for (i = 0; i < enab->dten_ndesc; i++) {
11701                 dtrace_actdesc_t *act, *next;
11702                 dtrace_predicate_t *pred;

11704                 ep = enab->dten_desc[i];

11706                 if ((pred = ep->dted_pred.dtpdd_predicate) != NULL)
11707                         dtrace_predicate_release(pred, vstate);

11709                 for (act = ep->dted_action; act != NULL; act = next) {
11710                         next = act->dtad_next;
11711                         dtrace_actdesc_release(act, vstate);
11712                 }

11714                 kmem_free(ep, sizeof (dtrace_ecbdesc_t));
11715         }

11717         kmem_free(enab->dten_desc,
11718             enab->dten_maxdesc * sizeof (dtrace_enabling_t *));

11720         /*
11721          * If this was a retained enabling, decrement the dts_nretained count
11722          * and take it off of the dtrace_retained list.
11723          */
11724         if (enab->dten_prev != NULL || enab->dten_next != NULL ||
11725             dtrace_retained == enab) {
11726                 ASSERT(enab->dten_vstate->dtvs_state != NULL);
11727                 ASSERT(enab->dten_vstate->dtvs_state->dts_nretained > 0);
11728                 enab->dten_vstate->dtvs_state->dts_nretained--;
11729                 dtrace_retained_gen++;
11730         }

11732         if (enab->dten_prev == NULL) {
11733                 if (dtrace_retained == enab) {
11734                         dtrace_retained = enab->dten_next;

11736                         if (dtrace_retained != NULL)
11737                                 dtrace_retained->dten_prev = NULL;
11738                 }
11739         } else {
11740                 ASSERT(enab != dtrace_retained);
11741                 ASSERT(dtrace_retained != NULL);
11742                 enab->dten_prev->dten_next = enab->dten_next;
```

```
11743         }

11745         if (enab->dten_next != NULL) {
11746                 ASSERT(dtrace_retained != NULL);
11747                 enab->dten_next->dten_prev = enab->dten_prev;
11748         }

11750         kmem_free(enab, sizeof (dtrace_enabling_t));
11751 }

11753 static int
11754 dtrace_enabling_retain(dtrace_enabling_t *enab)
11755 {
11756         dtrace_state_t *state;

11758         ASSERT(MUTEX_HELD(&dtrace_lock));
11759         ASSERT(enab->dten_next == NULL && enab->dten_prev == NULL);
11760         ASSERT(enab->dten_vstate != NULL);

11762         state = enab->dten_vstate->dtvs_state;
11763         ASSERT(state != NULL);

11765         /*
11766          * We only allow each state to retain dtrace_retain_max enablings.
11767          */
11768         if (state->dts_nretained >= dtrace_retain_max)
11769                 return (ENOSPC);

11771         state->dts_nretained++;
11772         dtrace_retained_gen++;

11774         if (dtrace_retained == NULL) {
11775                 dtrace_retained = enab;
11776                 return (0);
11777         }

11779         enab->dten_next = dtrace_retained;
11780         dtrace_retained->dten_prev = enab;
11781         dtrace_retained = enab;

11783         return (0);
11784 }

11786 static int
11787 dtrace_enabling_replicate(dtrace_state_t *state, dtrace_probedesc_t *match,
11788     dtrace_probedesc_t *create)
11789 {
11790         dtrace_enabling_t *new, *enab;
11791         int found = 0, err = ENOENT;

11793         ASSERT(MUTEX_HELD(&dtrace_lock));
11794         ASSERT(strlen(match->dtpd_provider) < DTRACE_PROVNAMELEN);
11795         ASSERT(strlen(match->dtpd_mod) < DTRACE_MODNAMELEN);
11796         ASSERT(strlen(match->dtpd_func) < DTRACE_FUNCNAMELEN);
11797         ASSERT(strlen(match->dtpd_name) < DTRACE_NAMELEN);

11799         new = dtrace_enabling_create(&state->dts_vstate);

11801         /*
11802          * Iterate over all retained enablings, looking for enablings that
11803          * match the specified state.
11804          */
11805         for (enab = dtrace_retained; enab != NULL; enab = enab->dten_next) {
11806                 int i;

11808                 /*
```



```
11809                  * dtvs_state can only be NULL for helper enablings -- and
11810                  * helper enablings can't be retained.
11811                  */
11812                 ASSERT(enab->dten_vstate->dtvs_state != NULL);

11814                 if (enab->dten_vstate->dtvs_state != state)
11815                         continue;

11817                 /*
11818                  * Now iterate over each probe description; we're looking for
11819                  * an exact match to the specified probe description.
11820                  */
11821                 for (i = 0; i < enab->dten_ndesc; i++) {
11822                         dtrace_ecbdesc_t *ep = enab->dten_desc[i];
11823                         dtrace_probedesc_t *pd = &ep->dted_probe;

11825                         if (strcmp(pd->dtpd_provider, match->dtpd_provider))
11826                                 continue;

11828                         if (strcmp(pd->dtpd_mod, match->dtpd_mod))
11829                                 continue;

11831                         if (strcmp(pd->dtpd_func, match->dtpd_func))
11832                                 continue;

11834                         if (strcmp(pd->dtpd_name, match->dtpd_name))
11835                                 continue;

11837                         /*
11838                          * We have a winning probe!  Add it to our growing
11839                          * enabling.
11840                          */
11841                         found = 1;
11842                         dtrace_enabling_addlike(new, ep, create);
11843                 }
11844         }

11846         if (!found || (err = dtrace_enabling_retain(new)) != 0) {
11847                 dtrace_enabling_destroy(new);
11848                 return (err);
11849         }

11851         return (0);
11852 }

11854 static void
11855 dtrace_enabling_retract(dtrace_state_t *state)
11856 {
11857         dtrace_enabling_t *enab, *next;

11859         ASSERT(MUTEX_HELD(&dtrace_lock));

11861         /*
11862          * Iterate over all retained enablings, destroy the enablings retained
11863          * for the specified state.
11864          */
11865         for (enab = dtrace_retained; enab != NULL; enab = next) {
11866                 next = enab->dten_next;

11868                 /*
11869                  * dtvs_state can only be NULL for helper enablings -- and
11870                  * helper enablings can't be retained.
11871                  */
11872                 ASSERT(enab->dten_vstate->dtvs_state != NULL);

11874                 if (enab->dten_vstate->dtvs_state == state) {
```

```
11875                         ASSERT(state->dts_nretained > 0);
11876                         dtrace_enabling_destroy(enab);
11877                 }
11878         }

11880         ASSERT(state->dts_nretained == 0);
11881 }

11883 static int
11884 dtrace_enabling_match(dtrace_enabling_t *enab, int *nmatched)
11885 {
11886         int i = 0;
11887         int total_matched = 0, matched = 0;

11889         ASSERT(MUTEX_HELD(&cpu_lock));
11890         ASSERT(MUTEX_HELD(&dtrace_lock));

11892         for (i = 0; i < enab->dten_ndesc; i++) {
11893                 dtrace_ecbdesc_t *ep = enab->dten_desc[i];

11895                 enab->dten_current = ep;
11896                 enab->dten_error = 0;

11898                 /*
11899                  * If a provider failed to enable a probe then get out and
11900                  * let the consumer know we failed.
11901                  */
11902                 if ((matched = dtrace_probe_enable(&ep->dted_probe, enab)) < 0)
11903                         return (EBUSY);

11905                 total_matched += matched;

11907                 if (enab->dten_error != 0) {
11908                         /*
11909                          * If we get an error half-way through enabling the
11910                          * probes, we kick out -- perhaps with some number of
11911                          * them enabled.  Leaving enabled probes enabled may
11912                          * be slightly confusing for user-level, but we expect
11913                          * that no one will attempt to actually drive on in
11914                          * the face of such errors.  If this is an anonymous
11915                          * enabling (indicated with a NULL nmatched pointer),
11916                          * we cmn_err() a message.  We aren't expecting to
11917                          * get such an error -- such as it can exist at all,
11918                          * it would be a result of corrupted DOF in the driver
11919                          * properties.
11920                          */
11921                         if (nmatched == NULL) {
11922                                 cmn_err(CE_WARN, "dtrace_enabling_match() "
11923                                     "error on %p: %d", (void *)ep,
11924                                     enab->dten_error);
11925                         }

11927                         return (enab->dten_error);
11928                 }
11929         }

11931         enab->dten_probegen = dtrace_probegen;
11932         if (nmatched != NULL)
11933                 *nmatched = total_matched;

11935         return (0);
11936 }

11938 static void
11939 dtrace_enabling_matchall(void)
11940 {
```



```
11941         dtrace_enabling_t *enab;

11943         mutex_enter(&cpu_lock);
11944         mutex_enter(&dtrace_lock);

11946         /*
11947          * Iterate over all retained enablings to see if any probes match
11948          * against them.  We only perform this operation on enablings for which
11949          * we have sufficient permissions by virtue of being in the global zone
11950          * or in the same zone as the DTrace client.  Because we can be called
11951          * after dtrace_detach() has been called, we cannot assert that there
11952          * are retained enablings.  We can safely load from dtrace_retained,
11953          * however:  the taskq_destroy() at the end of dtrace_detach() will
11954          * block pending our completion.
11955          */
11956         for (enab = dtrace_retained; enab != NULL; enab = enab->dten_next) {
11957                 dtrace_cred_t *dcr = &enab->dten_vstate->dtvs_state->dts_cred;
11958                 cred_t *cr = dcr->dcr_cred;
11959                 zoneid_t zone = cr != NULL ? crgetzoneid(cr) : 0;

11961                 if ((dcr->dcr_visible & DTRACE_CRV_ALLZONE) || (cr != NULL &&
11962                     (zone == GLOBAL_ZONEID || getzoneid() == zone)))
11963                         (void) dtrace_enabling_match(enab, NULL);
11964         }

11966         mutex_exit(&dtrace_lock);
11967         mutex_exit(&cpu_lock);
11968 }

11970 /*
11971  * If an enabling is to be enabled without having matched probes (that is, if
11972  * dtrace_state_go() is to be called on the underlying dtrace_state_t), the
11973  * enabling must be _primed_ by creating an ECB for every ECB description.
11974  * This must be done to assure that we know the number of speculations, the
11975  * number of aggregations, the minimum buffer size needed, etc. before we
11976  * transition out of DTRACE_ACTIVITY_INACTIVE.  To do this without actually
11977  * enabling any probes, we create ECBs for every ECB decription, but with a
11978  * NULL probe -- which is exactly what this function does.
11979  */
11980 static void
11981 dtrace_enabling_prime(dtrace_state_t *state)
11982 {
11983         dtrace_enabling_t *enab;
11984         int i;

11986         for (enab = dtrace_retained; enab != NULL; enab = enab->dten_next) {
11987                 ASSERT(enab->dten_vstate->dtvs_state != NULL);

11989                 if (enab->dten_vstate->dtvs_state != state)
11990                         continue;

11992                 /*
11993                  * We don't want to prime an enabling more than once, lest
11994                  * we allow a malicious user to induce resource exhaustion.
11995                  * (The ECBs that result from priming an enabling aren't
11996                  * leaked -- but they also aren't deallocated until the
11997                  * consumer state is destroyed.)
11998                  */
11999                 if (enab->dten_primed)
12000                         continue;

12002                 for (i = 0; i < enab->dten_ndesc; i++) {
12003                         enab->dten_current = enab->dten_desc[i];
12004                         (void) dtrace_probe_enable(NULL, enab);
12005                 }
```

```
12007                   enab->dten_primed = 1;
12008           }
12009   }

12011   /*
12012    * Called to indicate that probes should be provided due to retained
12013    * enablings.  This is implemented in terms of dtrace_probe_provide(), but it
12014    * must take an initial lap through the enabling calling the dtps_provide()
12015    * entry point explicitly to allow for autocreated probes.
12016    */
12017   static void
12018   dtrace_enabling_provide(dtrace_provider_t *prv)
12019   {
12020           int i, all = 0;
12021           dtrace_probedesc_t desc;
12022           dtrace_genid_t gen;

12024           ASSERT(MUTEX_HELD(&dtrace_lock));
12025           ASSERT(MUTEX_HELD(&dtrace_provider_lock));

12027           if (prv == NULL) {
12028                   all = 1;
12029                   prv = dtrace_provider;
12030           }

12032           do {
12033                   dtrace_enabling_t *enab;
12034                   void *parg = prv->dtpv_arg;

12036   retry:
12037                   gen = dtrace_retained_gen;
12038                   for (enab = dtrace_retained; enab != NULL;
12039                       enab = enab->dten_next) {
12040                           for (i = 0; i < enab->dten_ndesc; i++) {
12041                                   desc = enab->dten_desc[i]->dted_probe;
12042                                   mutex_exit(&dtrace_lock);
12043                                   prv->dtpv_pops.dtps_provide(parg, &desc);
12044                                   mutex_enter(&dtrace_lock);
12045                                   /*
12046                                    * Process the retained enablings again if
12047                                    * they have changed while we weren't holding
12048                                    * dtrace_lock.
12049                                    */
12050                                   if (gen != dtrace_retained_gen)
12051                                           goto retry;
12052                           }
12053                   }
12054           } while (all && (prv = prv->dtpv_next) != NULL);

12056           mutex_exit(&dtrace_lock);
12057           dtrace_probe_provide(NULL, all ? NULL : prv);
12058           mutex_enter(&dtrace_lock);
12059   }

12061   /*
12062    * Called to reap ECBs that are attached to probes from defunct providers.
12063    */
12064   static void
12065   dtrace_enabling_reap(void)
12066   {
12067           dtrace_provider_t *prov;
12068           dtrace_probe_t *probe;
12069           dtrace_ecb_t *ecb;
12070           hrtime_t when;
12071           int i;
```



```
12073           mutex_enter(&cpu_lock);
12074           mutex_enter(&dtrace_lock);

12076           for (i = 0; i < dtrace_nprobes; i++) {
12077                   if ((probe = dtrace_probes[i]) == NULL)
12078                           continue;

12080                   if (probe->dtpr_ecb == NULL)
12081                           continue;

12083                   prov = probe->dtpr_provider;

12085                   if ((when = prov->dtpv_defunct) == 0)
12086                           continue;

12088                   /*
12089                    * We have ECBs on a defunct provider:  we want to reap these
12090                    * ECBs to allow the provider to unregister.  The destruction
12091                    * of these ECBs must be done carefully:  if we destroy the ECB
12092                    * and the consumer later wishes to consume an EPID that
12093                    * corresponds to the destroyed ECB (and if the EPID metadata
12094                    * has not been previously consumed), the consumer will abort
12095                    * processing on the unknown EPID.  To reduce (but not, sadly,
12096                    * eliminate) the possibility of this, we will only destroy an
12097                    * ECB for a defunct provider if, for the state that
12098                    * corresponds to the ECB:
12099                    *
12100                    *  (a) There is no speculative tracing (which can effectively
12101                    *      cache an EPID for an arbitrary amount of time).
12102                    *
12103                    *  (b) The principal buffers have been switched twice since the
12104                    *      provider became defunct.
12105                    *
12106                    *  (c) The aggregation buffers are of zero size or have been
12107                    *      switched twice since the provider became defunct.
12108                    *
12109                    * We use dts_speculates to determine (a) and call a function
12110                    * (dtrace_buffer_consumed()) to determine (b) and (c).  Note
12111                    * that as soon as we've been unable to destroy one of the ECBs
12112                    * associated with the probe, we quit trying -- reaping is only
12113                    * fruitful in as much as we can destroy all ECBs associated
12114                    * with the defunct provider's probes.
12115                    */
12116                   while ((ecb = probe->dtpr_ecb) != NULL) {
12117                           dtrace_state_t *state = ecb->dte_state;
12118                           dtrace_buffer_t *buf = state->dts_buffer;
12119                           dtrace_buffer_t *aggbuf = state->dts_aggbuffer;

12121                           if (state->dts_speculates)
12122                                   break;

12124                           if (!dtrace_buffer_consumed(buf, when))
12125                                   break;

12127                           if (!dtrace_buffer_consumed(aggbuf, when))
12128                                   break;

12130                           dtrace_ecb_disable(ecb);
12131                           ASSERT(probe->dtpr_ecb != ecb);
12132                           dtrace_ecb_destroy(ecb);
12133                   }
12134           }

12136           mutex_exit(&dtrace_lock);
12137           mutex_exit(&cpu_lock);
12138   }
```

```
12140 /*
12141  * DTrace DOF Functions
12142  */
12143 /*ARGSUSED*/
12144 static void
12145 dtrace_dof_error(dof_hdr_t *dof, const char *str)
12146 {
12147         if (dtrace_err_verbose)
12148                 cmn_err(CE_WARN, "failed to process DOF: %s", str);

12150 #ifdef DTRACE_ERRDEBUG
12151         dtrace_errdebug(str);
12152 #endif
12153 }

12155 /*
12156  * Create DOF out of a currently enabled state.  Right now, we only create
12157  * DOF containing the run-time options -- but this could be expanded to create
12158  * complete DOF representing the enabled state.
12159  */
12160 static dof_hdr_t *
12161 dtrace_dof_create(dtrace_state_t *state)
12162 {
12163         dof_hdr_t *dof;
12164         dof_sec_t *sec;
12165         dof_optdesc_t *opt;
12166         int i, len = sizeof (dof_hdr_t) +
12167             roundup(sizeof (dof_sec_t), sizeof (uint64_t)) +
12168             sizeof (dof_optdesc_t) * DTRACEOPT_MAX;

12170         ASSERT(MUTEX_HELD(&dtrace_lock));

12172         dof = kmem_zalloc(len, KM_SLEEP);
12173         dof->dofh_ident[DOF_ID_MAG0] = DOF_MAG_MAG0;
12174         dof->dofh_ident[DOF_ID_MAG1] = DOF_MAG_MAG1;
12175         dof->dofh_ident[DOF_ID_MAG2] = DOF_MAG_MAG2;
12176         dof->dofh_ident[DOF_ID_MAG3] = DOF_MAG_MAG3;

12178         dof->dofh_ident[DOF_ID_MODEL] = DOF_MODEL_NATIVE;
12179         dof->dofh_ident[DOF_ID_ENCODING] = DOF_ENCODE_NATIVE;
12180         dof->dofh_ident[DOF_ID_VERSION] = DOF_VERSION;
12181         dof->dofh_ident[DOF_ID_DIFVERS] = DIF_VERSION;
12182         dof->dofh_ident[DOF_ID_DIFIREG] = DIF_DIR_NREGS;
12183         dof->dofh_ident[DOF_ID_DIFTREG] = DIF_DTR_NREGS;

12185         dof->dofh_flags = 0;
12186         dof->dofh_hdrsize = sizeof (dof_hdr_t);
12187         dof->dofh_secsize = sizeof (dof_sec_t);
12188         dof->dofh_secnum = 1;   /* only DOF_SECT_OPTDESC */
12189         dof->dofh_secoff = sizeof (dof_hdr_t);
12190         dof->dofh_loadsz = len;
12191         dof->dofh_filesz = len;
12192         dof->dofh_pad = 0;

12194         /*
12195          * Fill in the option section header...
12196          */
12197         sec = (dof_sec_t *)((uintptr_t)dof + sizeof (dof_hdr_t));
12198         sec->dofs_type = DOF_SECT_OPTDESC;
12199         sec->dofs_align = sizeof (uint64_t);
12200         sec->dofs_flags = DOF_SECF_LOAD;
12201         sec->dofs_entsize = sizeof (dof_optdesc_t);

12203         opt = (dof_optdesc_t *)((uintptr_t)sec +
12204             roundup(sizeof (dof_sec_t), sizeof (uint64_t)));
```



```
12206         sec->dofs_offset = (uintptr_t)opt - (uintptr_t)dof;
12207         sec->dofs_size = sizeof (dof_optdesc_t) * DTRACEOPT_MAX;

12209         for (i = 0; i < DTRACEOPT_MAX; i++) {
12210                 opt[i].dofo_option = i;
12211                 opt[i].dofo_strtab = DOF_SECIDX_NONE;
12212                 opt[i].dofo_value = state->dts_options[i];
12213         }

12215         return (dof);
12216 }

12218 static dof_hdr_t *
12219 dtrace_dof_copyin(uintptr_t uarg, int *errp)
12220 {
12221         dof_hdr_t hdr, *dof;

12223         ASSERT(!MUTEX_HELD(&dtrace_lock));

12225         /*
12226          * First, we're going to copyin() the sizeof (dof_hdr_t).
12227          */
12228         if (copyin((void *)uarg, &hdr, sizeof (hdr)) != 0) {
12229                 dtrace_dof_error(NULL, "failed to copyin DOF header");
12230                 *errp = EFAULT;
12231                 return (NULL);
12232         }

12234         /*
12235          * Now we'll allocate the entire DOF and copy it in -- provided
12236          * that the length isn't outrageous.
12237          */
12238         if (hdr.dofh_loadsz >= dtrace_dof_maxsize) {
12239                 dtrace_dof_error(&hdr, "load size exceeds maximum");
12240                 *errp = E2BIG;
12241                 return (NULL);
12242         }

12244         if (hdr.dofh_loadsz < sizeof (hdr)) {
12245                 dtrace_dof_error(&hdr, "invalid load size");
12246                 *errp = EINVAL;
12247                 return (NULL);
12248         }

12250         dof = kmem_alloc(hdr.dofh_loadsz, KM_SLEEP);

12252         if (copyin((void *)uarg, dof, hdr.dofh_loadsz) != 0 ||
12253             dof->dofh_loadsz != hdr.dofh_loadsz) {
12254                 kmem_free(dof, hdr.dofh_loadsz);
12255                 *errp = EFAULT;
12256                 return (NULL);
12257         }

12259         return (dof);
12260 }

12262 static dof_hdr_t *
12263 dtrace_dof_property(const char *name)
12264 {
12265         uchar_t *buf;
12266         uint64_t loadsz;
12267         unsigned int len, i;
12268         dof_hdr_t *dof;

12270         /*
```

```
12271          * Unfortunately, array of values in .conf files are always (and
12272          * only) interpreted to be integer arrays.  We must read our DOF
12273          * as an integer array, and then squeeze it into a byte array.
12274          */
12275         if (ddi_prop_lookup_int_array(DDI_DEV_T_ANY, dtrace_devi, 0,
12276             (char *)name, (int **)&buf, &len) != DDI_PROP_SUCCESS)
12277                 return (NULL);
12278
12279         for (i = 0; i < len; i++)
12280                 buf[i] = (uchar_t)(((int *)buf)[i]);
12281
12282         if (len < sizeof (dof_hdr_t)) {
12283                 ddi_prop_free(buf);
12284                 dtrace_dof_error(NULL, "truncated header");
12285                 return (NULL);
12286         }
12287
12288         if (len < (loadsz = ((dof_hdr_t *)buf)->dofh_loadsz)) {
12289                 ddi_prop_free(buf);
12290                 dtrace_dof_error(NULL, "truncated DOF");
12291                 return (NULL);
12292         }
12293
12294         if (loadsz >= dtrace_dof_maxsize) {
12295                 ddi_prop_free(buf);
12296                 dtrace_dof_error(NULL, "oversized DOF");
12297                 return (NULL);
12298         }
12299
12300         dof = kmem_alloc(loadsz, KM_SLEEP);
12301         bcopy(buf, dof, loadsz);
12302         ddi_prop_free(buf);
12303
12304         return (dof);
12305 }
12306
12307 static void
12308 dtrace_dof_destroy(dof_hdr_t *dof)
12309 {
12310         kmem_free(dof, dof->dofh_loadsz);
12311 }
12312
12313 /*
12314  * Return the dof_sec_t pointer corresponding to a given section index.  If the
12315  * index is not valid, dtrace_dof_error() is called and NULL is returned.  If
12316  * a type other than DOF_SECT_NONE is specified, the header is checked against
12317  * this type and NULL is returned if the types do not match.
12318  */
12319 static dof_sec_t *
12320 dtrace_dof_sect(dof_hdr_t *dof, uint32_t type, dof_secidx_t i)
12321 {
12322         dof_sec_t *sec = (dof_sec_t *)(uintptr_t)
12323             ((uintptr_t)dof + dof->dofh_secoff + i * dof->dofh_secsize);
12324
12325         if (i >= dof->dofh_secnum) {
12326                 dtrace_dof_error(dof, "referenced section index is invalid");
12327                 return (NULL);
12328         }
12329
12330         if (!(sec->dofs_flags & DOF_SECF_LOAD)) {
12331                 dtrace_dof_error(dof, "referenced section is not loadable");
12332                 return (NULL);
12333         }
12334
12335         if (type != DOF_SECT_NONE && type != sec->dofs_type) {
12336                 dtrace_dof_error(dof, "referenced section is the wrong type");
```



```
12337                 return (NULL);
12338         }
12339
12340         return (sec);
12341 }
12342
12343 static dtrace_probedesc_t *
12344 dtrace_dof_probedesc(dof_hdr_t *dof, dof_sec_t *sec, dtrace_probedesc_t *desc)
12345 {
12346         dof_probedesc_t *probe;
12347         dof_sec_t *strtab;
12348         uintptr_t daddr = (uintptr_t)dof;
12349         uintptr_t str;
12350         size_t size;
12351
12352         if (sec->dofs_type != DOF_SECT_PROBEDESC) {
12353                 dtrace_dof_error(dof, "invalid probe section");
12354                 return (NULL);
12355         }
12356
12357         if (sec->dofs_align != sizeof (dof_secidx_t)) {
12358                 dtrace_dof_error(dof, "bad alignment in probe description");
12359                 return (NULL);
12360         }
12361
12362         if (sec->dofs_offset + sizeof (dof_probedesc_t) > dof->dofh_loadsz) {
12363                 dtrace_dof_error(dof, "truncated probe description");
12364                 return (NULL);
12365         }
12366
12367         probe = (dof_probedesc_t *)(uintptr_t)(daddr + sec->dofs_offset);
12368         strtab = dtrace_dof_sect(dof, DOF_SECT_STRTAB, probe->dofp_strtab);
12369
12370         if (strtab == NULL)
12371                 return (NULL);
12372
12373         str = daddr + strtab->dofs_offset;
12374         size = strtab->dofs_size;
12375
12376         if (probe->dofp_provider >= strtab->dofs_size) {
12377                 dtrace_dof_error(dof, "corrupt probe provider");
12378                 return (NULL);
12379         }
12380
12381         (void) strncpy(desc->dtpd_provider,
12382             (char *)(str + probe->dofp_provider),
12383             MIN(DTRACE_PROVNAMELEN - 1, size - probe->dofp_provider));
12384
12385         if (probe->dofp_mod >= strtab->dofs_size) {
12386                 dtrace_dof_error(dof, "corrupt probe module");
12387                 return (NULL);
12388         }
12389
12390         (void) strncpy(desc->dtpd_mod, (char *)(str + probe->dofp_mod),
12391             MIN(DTRACE_MODNAMELEN - 1, size - probe->dofp_mod));
12392
12393         if (probe->dofp_func >= strtab->dofs_size) {
12394                 dtrace_dof_error(dof, "corrupt probe function");
12395                 return (NULL);
12396         }
12397
12398         (void) strncpy(desc->dtpd_func, (char *)(str + probe->dofp_func),
12399             MIN(DTRACE_FUNCNAMELEN - 1, size - probe->dofp_func));
12400
12401         if (probe->dofp_name >= strtab->dofs_size) {
12402                 dtrace_dof_error(dof, "corrupt probe name");
```

```
12403                   return (NULL);
12404           }

12406           (void) strncpy(desc->dtpd_name, (char *)(str + probe->dofp_name),
12407               MIN(DTRACE_NAMELEN - 1, size - probe->dofp_name));

12409           return (desc);
12410   }

12412   static dtrace_difo_t *
12413   dtrace_dof_difo(dof_hdr_t *dof, dof_sec_t *sec, dtrace_vstate_t *vstate,
12414       cred_t *cr)
12415   {
12416           dtrace_difo_t *dp;
12417           size_t ttl = 0;
12418           dof_difohdr_t *dofd;
12419           uintptr_t daddr = (uintptr_t)dof;
12420           size_t max = dtrace_difo_maxsize;
12421           int i, l, n;

12423           static const struct {
12424                   int section;
12425                   int bufoffs;
12426                   int lenoffs;
12427                   int entsize;
12428                   int align;
12429                   const char *msg;
12430           } difo[] = {
12431                   { DOF_SECT_DIF, offsetof(dtrace_difo_t, dtdo_buf),
12432                   offsetof(dtrace_difo_t, dtdo_len), sizeof (dif_instr_t),
12433                   sizeof (dif_instr_t), "multiple DIF sections" },

12435                   { DOF_SECT_INTTAB, offsetof(dtrace_difo_t, dtdo_inttab),
12436                   offsetof(dtrace_difo_t, dtdo_intlen), sizeof (uint64_t),
12437                   sizeof (uint64_t), "multiple integer tables" },

12439                   { DOF_SECT_STRTAB, offsetof(dtrace_difo_t, dtdo_strtab),
12440                   offsetof(dtrace_difo_t, dtdo_strlen), 0,
12441                   sizeof (char), "multiple string tables" },

12443                   { DOF_SECT_VARTAB, offsetof(dtrace_difo_t, dtdo_vartab),
12444                   offsetof(dtrace_difo_t, dtdo_varlen), sizeof (dtrace_difv_t),
12445                   sizeof (uint_t), "multiple variable tables" },

12447                   { DOF_SECT_NONE, 0, 0, 0, NULL }
12448           };

12450           if (sec->dofs_type != DOF_SECT_DIFOHDR) {
12451                   dtrace_dof_error(dof, "invalid DIFO header section");
12452                   return (NULL);
12453           }

12455           if (sec->dofs_align != sizeof (dof_secidx_t)) {
12456                   dtrace_dof_error(dof, "bad alignment in DIFO header");
12457                   return (NULL);
12458           }

12460           if (sec->dofs_size < sizeof (dof_difohdr_t) ||
12461               sec->dofs_size % sizeof (dof_secidx_t)) {
12462                   dtrace_dof_error(dof, "bad size in DIFO header");
12463                   return (NULL);
12464           }

12466           dofd = (dof_difohdr_t *)(uintptr_t)(daddr + sec->dofs_offset);
12467           n = (sec->dofs_size - sizeof (*dofd)) / sizeof (dof_secidx_t) + 1;
```



```
12469           dp = kmem_zalloc(sizeof (dtrace_difo_t), KM_SLEEP);
12470           dp->dtdo_rtype = dofd->dofd_rtype;

12472           for (l = 0; l < n; l++) {
12473                   dof_sec_t *subsec;
12474                   void **bufp;
12475                   uint32_t *lenp;

12477                   if ((subsec = dtrace_dof_sect(dof, DOF_SECT_NONE,
12478                       dofd->dofd_links[l])) == NULL)
12479                           goto err; /* invalid section link */

12481                   if (ttl + subsec->dofs_size > max) {
12482                           dtrace_dof_error(dof, "exceeds maximum size");
12483                           goto err;
12484                   }

12486                   ttl += subsec->dofs_size;

12488                   for (i = 0; difo[i].section != DOF_SECT_NONE; i++) {
12489                           if (subsec->dofs_type != difo[i].section)
12490                                   continue;

12492                           if (!(subsec->dofs_flags & DOF_SECF_LOAD)) {
12493                                   dtrace_dof_error(dof, "section not loaded");
12494                                   goto err;
12495                           }

12497                           if (subsec->dofs_align != difo[i].align) {
12498                                   dtrace_dof_error(dof, "bad alignment");
12499                                   goto err;
12500                           }

12502                           bufp = (void **)((uintptr_t)dp + difo[i].bufoffs);
12503                           lenp = (uint32_t *)((uintptr_t)dp + difo[i].lenoffs);

12505                           if (*bufp != NULL) {
12506                                   dtrace_dof_error(dof, difo[i].msg);
12507                                   goto err;
12508                           }

12510                           if (difo[i].entsize != subsec->dofs_entsize) {
12511                                   dtrace_dof_error(dof, "entry size mismatch");
12512                                   goto err;
12513                           }

12515                           if (subsec->dofs_entsize != 0 &&
12516                               (subsec->dofs_size % subsec->dofs_entsize) != 0) {
12517                                   dtrace_dof_error(dof, "corrupt entry size");
12518                                   goto err;
12519                           }

12521                           *lenp = subsec->dofs_size;
12522                           *bufp = kmem_alloc(subsec->dofs_size, KM_SLEEP);
12523                           bcopy((char *)(uintptr_t)(daddr + subsec->dofs_offset),
12524                               *bufp, subsec->dofs_size);

12526                           if (subsec->dofs_entsize != 0)
12527                                   *lenp /= subsec->dofs_entsize;

12529                           break;
12530                   }

12532                   /*
12533                    * If we encounter a loadable DIFO sub-section that is not
12534                    * known to us, assume this is a broken program and fail.
```

```
12535                  */
12536                 if (difo[i].section == DOF_SECT_NONE &&
12537                     (subsec->dofs_flags & DOF_SECF_LOAD)) {
12538                         dtrace_dof_error(dof, "unrecognized DIFO subsection");
12539                         goto err;
12540                 }
12541         }

12543         if (dp->dtdo_buf == NULL) {
12544                 /*
12545                  * We can't have a DIF object without DIF text.
12546                  */
12547                 dtrace_dof_error(dof, "missing DIF text");
12548                 goto err;
12549         }

12551         /*
12552          * Before we validate the DIF object, run through the variable table
12553          * looking for the strings -- if any of their size are under, we'll set
12554          * their size to be the system-wide default string size.  Note that
12555          * this should _not_ happen if the "strsize" option has been set --
12556          * in this case, the compiler should have set the size to reflect the
12557          * setting of the option.
12558          */
12559         for (i = 0; i < dp->dtdo_varlen; i++) {
12560                 dtrace_difv_t *v = &dp->dtdo_vartab[i];
12561                 dtrace_diftype_t *t = &v->dtdv_type;

12563                 if (v->dtdv_id < DIF_VAR_OTHER_UBASE)
12564                         continue;

12566                 if (t->dtdt_kind == DIF_TYPE_STRING && t->dtdt_size == 0)
12567                         t->dtdt_size = dtrace_strsize_default;
12568         }

12570         if (dtrace_difo_validate(dp, vstate, DIF_DIR_NREGS, cr) != 0)
12571                 goto err;

12573         dtrace_difo_init(dp, vstate);
12574         return (dp);

12576 err:
12577         kmem_free(dp->dtdo_buf, dp->dtdo_len * sizeof (dif_instr_t));
12578         kmem_free(dp->dtdo_inttab, dp->dtdo_intlen * sizeof (uint64_t));
12579         kmem_free(dp->dtdo_strtab, dp->dtdo_strlen);
12580         kmem_free(dp->dtdo_vartab, dp->dtdo_varlen * sizeof (dtrace_difv_t));

12582         kmem_free(dp, sizeof (dtrace_difo_t));
12583         return (NULL);
12584 }

12586 static dtrace_predicate_t *
12587 dtrace_dof_predicate(dof_hdr_t *dof, dof_sec_t *sec, dtrace_vstate_t *vstate,
12588     cred_t *cr)
12589 {
12590         dtrace_difo_t *dp;

12592         if ((dp = dtrace_dof_difo(dof, sec, vstate, cr)) == NULL)
12593                 return (NULL);

12595         return (dtrace_predicate_create(dp));
12596 }

12598 static dtrace_actdesc_t *
12599 dtrace_dof_actdesc(dof_hdr_t *dof, dof_sec_t *sec, dtrace_vstate_t *vstate,
12600     cred_t *cr)
```



```
12601 {
12602         dtrace_actdesc_t *act, *first = NULL, *last = NULL, *next;
12603         dof_actdesc_t *desc;
12604         dof_sec_t *difosec;
12605         size_t offs;
12606         uintptr_t daddr = (uintptr_t)dof;
12607         uint64_t arg;
12608         dtrace_actkind_t kind;

12610         if (sec->dofs_type != DOF_SECT_ACTDESC) {
12611                 dtrace_dof_error(dof, "invalid action section");
12612                 return (NULL);
12613         }

12615         if (sec->dofs_offset + sizeof (dof_actdesc_t) > dof->dofh_loadsz) {
12616                 dtrace_dof_error(dof, "truncated action description");
12617                 return (NULL);
12618         }

12620         if (sec->dofs_align != sizeof (uint64_t)) {
12621                 dtrace_dof_error(dof, "bad alignment in action description");
12622                 return (NULL);
12623         }

12625         if (sec->dofs_size < sec->dofs_entsize) {
12626                 dtrace_dof_error(dof, "section entry size exceeds total size");
12627                 return (NULL);
12628         }

12630         if (sec->dofs_entsize != sizeof (dof_actdesc_t)) {
12631                 dtrace_dof_error(dof, "bad entry size in action description");
12632                 return (NULL);
12633         }

12635         if (sec->dofs_size / sec->dofs_entsize > dtrace_actions_max) {
12636                 dtrace_dof_error(dof, "actions exceed dtrace_actions_max");
12637                 return (NULL);
12638         }

12640         for (offs = 0; offs < sec->dofs_size; offs += sec->dofs_entsize) {
12641                 desc = (dof_actdesc_t *)(daddr +
12642                     (uintptr_t)sec->dofs_offset + offs);
12643                 kind = (dtrace_actkind_t)desc->dofa_kind;

12645                 if ((DTRACEACT_ISPRINTFLIKE(kind) &&
12646                     (kind != DTRACEACT_PRINTA ||
12647                     desc->dofa_strtab != DOF_SECIDX_NONE)) ||
12648                     (kind == DTRACEACT_DIFEXPR &&
12649                     desc->dofa_strtab != DOF_SECIDX_NONE)) {
12650                         dof_sec_t *strtab;
12651                         char *str, *fmt;
12652                         uint64_t i;

12654                         /*
12655                          * The argument to these actions is an index into the
12656                          * DOF string table.  For printf()-like actions, this
12657                          * is the format string.  For print(), this is the
12658                          * CTF type of the expression result.
12659                          */
12660                         if ((strtab = dtrace_dof_sect(dof,
12661                             DOF_SECT_STRTAB, desc->dofa_strtab)) == NULL)
12662                                 goto err;

12664                         str = (char *)((uintptr_t)dof +
12665                             (uintptr_t)strtab->dofs_offset);
```

```
12667                         for (i = desc->dofa_arg; i < strtab->dofs_size; i++) {
12668                                 if (str[i] == '\0')
12669                                         break;
12670                         }

12672                         if (i >= strtab->dofs_size) {
12673                                 dtrace_dof_error(dof, "bogus format string");
12674                                 goto err;
12675                         }

12677                         if (i == desc->dofa_arg) {
12678                                 dtrace_dof_error(dof, "empty format string");
12679                                 goto err;
12680                         }

12682                         i -= desc->dofa_arg;
12683                         fmt = kmem_alloc(i + 1, KM_SLEEP);
12684                         bcopy(&str[desc->dofa_arg], fmt, i + 1);
12685                         arg = (uint64_t)(uintptr_t)fmt;
12686                 } else {
12687                         if (kind == DTRACEACT_PRINTA) {
12688                                 ASSERT(desc->dofa_strtab == DOF_SECIDX_NONE);
12689                                 arg = 0;
12690                         } else {
12691                                 arg = desc->dofa_arg;
12692                         }
12693                 }

12695                 act = dtrace_actdesc_create(kind, desc->dofa_ntuple,
12696                     desc->dofa_uarg, arg);

12698                 if (last != NULL) {
12699                         last->dtad_next = act;
12700                 } else {
12701                         first = act;
12702                 }

12704                 last = act;

12706                 if (desc->dofa_difo == DOF_SECIDX_NONE)
12707                         continue;

12709                 if ((difosec = dtrace_dof_sect(dof,
12710                     DOF_SECT_DIFOHDR, desc->dofa_difo)) == NULL)
12711                         goto err;

12713                 act->dtad_difo = dtrace_dof_difo(dof, difosec, vstate, cr);

12715                 if (act->dtad_difo == NULL)
12716                         goto err;
12717         }

12719         ASSERT(first != NULL);
12720         return (first);

12722 err:
12723         for (act = first; act != NULL; act = next) {
12724                 next = act->dtad_next;
12725                 dtrace_actdesc_release(act, vstate);
12726         }

12728         return (NULL);
12729 }

12731 static dtrace_ecbdesc_t *
12732 dtrace_dof_ecbdesc(dof_hdr_t *dof, dof_sec_t *sec, dtrace_vstate_t *vstate,
```



```
12733     cred_t *cr)
12734 {
12735         dtrace_ecbdesc_t *ep;
12736         dof_ecbdesc_t *ecb;
12737         dtrace_probedesc_t *desc;
12738         dtrace_predicate_t *pred = NULL;

12740         if (sec->dofs_size < sizeof (dof_ecbdesc_t)) {
12741                 dtrace_dof_error(dof, "truncated ECB description");
12742                 return (NULL);
12743         }

12745         if (sec->dofs_align != sizeof (uint64_t)) {
12746                 dtrace_dof_error(dof, "bad alignment in ECB description");
12747                 return (NULL);
12748         }

12750         ecb = (dof_ecbdesc_t *)((uintptr_t)dof + (uintptr_t)sec->dofs_offset);
12751         sec = dtrace_dof_sect(dof, DOF_SECT_PROBEDESC, ecb->dofe_probes);

12753         if (sec == NULL)
12754                 return (NULL);

12756         ep = kmem_zalloc(sizeof (dtrace_ecbdesc_t), KM_SLEEP);
12757         ep->dted_uarg = ecb->dofe_uarg;
12758         desc = &ep->dted_probe;

12760         if (dtrace_dof_probedesc(dof, sec, desc) == NULL)
12761                 goto err;

12763         if (ecb->dofe_pred != DOF_SECIDX_NONE) {
12764                 if ((sec = dtrace_dof_sect(dof,
12765                     DOF_SECT_DIFOHDR, ecb->dofe_pred)) == NULL)
12766                         goto err;

12768                 if ((pred = dtrace_dof_predicate(dof, sec, vstate, cr)) == NULL)
12769                         goto err;

12771                 ep->dted_pred.dtpdd_predicate = pred;
12772         }

12774         if (ecb->dofe_actions != DOF_SECIDX_NONE) {
12775                 if ((sec = dtrace_dof_sect(dof,
12776                     DOF_SECT_ACTDESC, ecb->dofe_actions)) == NULL)
12777                         goto err;

12779                 ep->dted_action = dtrace_dof_actdesc(dof, sec, vstate, cr);

12781                 if (ep->dted_action == NULL)
12782                         goto err;
12783         }

12785         return (ep);

12787 err:
12788         if (pred != NULL)
12789                 dtrace_predicate_release(pred, vstate);
12790         kmem_free(ep, sizeof (dtrace_ecbdesc_t));
12791         return (NULL);
12792 }

12794 /*
12795  * Apply the relocations from the specified 'sec' (a DOF_SECT_URELHDR) to the
12796  * specified DOF.  At present, this amounts to simply adding 'ubase' to the
12797  * site of any user SETX relocations to account for load object base address.
12798  * In the future, if we need other relocations, this function can be extended.
```

```
 */
static int
dtrace_dof_relocate(dof_hdr_t *dof, dof_sec_t *sec, uint64_t ubase)
{
        uintptr_t daddr = (uintptr_t)dof;
        dof_relohdr_t *dofr =
            (dof_relohdr_t *)(uintptr_t)(daddr + sec->dofs_offset);
        dof_sec_t *ss, *rs, *ts;
        dof_relodesc_t *r;
        uint_t i, n;

        if (sec->dofs_size < sizeof (dof_relohdr_t) ||
            sec->dofs_align != sizeof (dof_secidx_t)) {
                dtrace_dof_error(dof, "invalid relocation header");
                return (-1);
        }

        ss = dtrace_dof_sect(dof, DOF_SECT_STRTAB, dofr->dofr_strtab);
        rs = dtrace_dof_sect(dof, DOF_SECT_RELTAB, dofr->dofr_relsec);
        ts = dtrace_dof_sect(dof, DOF_SECT_NONE, dofr->dofr_tgtsec);

        if (ss == NULL || rs == NULL || ts == NULL)
                return (-1); /* dtrace_dof_error() has been called already */

        if (rs->dofs_entsize < sizeof (dof_relodesc_t) ||
            rs->dofs_align != sizeof (uint64_t)) {
                dtrace_dof_error(dof, "invalid relocation section");
                return (-1);
        }

        r = (dof_relodesc_t *)(uintptr_t)(daddr + rs->dofs_offset);
        n = rs->dofs_size / rs->dofs_entsize;

        for (i = 0; i < n; i++) {
                uintptr_t taddr = daddr + ts->dofs_offset + r->dofr_offset;

                switch (r->dofr_type) {
                case DOF_RELO_NONE:
                        break;
                case DOF_RELO_SETX:
                        if (r->dofr_offset >= ts->dofs_size || r->dofr_offset +
                            sizeof (uint64_t) > ts->dofs_size) {
                                dtrace_dof_error(dof, "bad relocation offset");
                                return (-1);
                        }

                        if (!IS_P2ALIGNED(taddr, sizeof (uint64_t))) {
                                dtrace_dof_error(dof, "misaligned setx relo");
                                return (-1);
                        }

                        *(uint64_t *)taddr += ubase;
                        break;
                default:
                        dtrace_dof_error(dof, "invalid relocation type");
                        return (-1);
                }

                r = (dof_relodesc_t *)((uintptr_t)r + rs->dofs_entsize);
        }

        return (0);
}

/*
 * The dof_hdr_t passed to dtrace_dof_slurp() should be a partially validated
```



```
 * header:  it should be at the front of a memory region that is at least
 * sizeof (dof_hdr_t) in size -- and then at least dof_hdr.dofh_loadsz in
 * size.  It need not be validated in any other way.
 */
static int
dtrace_dof_slurp(dof_hdr_t *dof, dtrace_vstate_t *vstate, cred_t *cr,
    dtrace_enabling_t **enabp, uint64_t ubase, int noprobes)
{
        uint64_t len = dof->dofh_loadsz, seclen;
        uintptr_t daddr = (uintptr_t)dof;
        dtrace_ecbdesc_t *ep;
        dtrace_enabling_t *enab;
        uint_t i;

        ASSERT(MUTEX_HELD(&dtrace_lock));
        ASSERT(dof->dofh_loadsz >= sizeof (dof_hdr_t));

        /*
         * Check the DOF header identification bytes.  In addition to checking
         * valid settings, we also verify that unused bits/bytes are zeroed so
         * we can use them later without fear of regressing existing binaries.
         */
        if (bcmp(&dof->dofh_ident[DOF_ID_MAG0],
            DOF_MAG_STRING, DOF_MAG_STRLEN) != 0) {
                dtrace_dof_error(dof, "DOF magic string mismatch");
                return (-1);
        }

        if (dof->dofh_ident[DOF_ID_MODEL] != DOF_MODEL_ILP32 &&
            dof->dofh_ident[DOF_ID_MODEL] != DOF_MODEL_LP64) {
                dtrace_dof_error(dof, "DOF has invalid data model");
                return (-1);
        }

        if (dof->dofh_ident[DOF_ID_ENCODING] != DOF_ENCODE_NATIVE) {
                dtrace_dof_error(dof, "DOF encoding mismatch");
                return (-1);
        }

        if (dof->dofh_ident[DOF_ID_VERSION] != DOF_VERSION_1 &&
            dof->dofh_ident[DOF_ID_VERSION] != DOF_VERSION_2) {
                dtrace_dof_error(dof, "DOF version mismatch");
                return (-1);
        }

        if (dof->dofh_ident[DOF_ID_DIFVERS] != DIF_VERSION_2) {
                dtrace_dof_error(dof, "DOF uses unsupported instruction set");
                return (-1);
        }

        if (dof->dofh_ident[DOF_ID_DIFIREG] > DIF_DIR_NREGS) {
                dtrace_dof_error(dof, "DOF uses too many integer registers");
                return (-1);
        }

        if (dof->dofh_ident[DOF_ID_DIFTREG] > DIF_DTR_NREGS) {
                dtrace_dof_error(dof, "DOF uses too many tuple registers");
                return (-1);
        }

        for (i = DOF_ID_PAD; i < DOF_ID_SIZE; i++) {
                if (dof->dofh_ident[i] != 0) {
                        dtrace_dof_error(dof, "DOF has invalid ident byte set");
                        return (-1);
                }
        }
```

```
12932         if (dof->dofh_flags & ~DOF_FL_VALID) {
12933                 dtrace_dof_error(dof, "DOF has invalid flag bits set");
12934                 return (-1);
12935         }

12937         if (dof->dofh_secsize == 0) {
12938                 dtrace_dof_error(dof, "zero section header size");
12939                 return (-1);
12940         }

12942         /*
12943          * Check that the section headers don't exceed the amount of DOF
12944          * data.  Note that we cast the section size and number of sections
12945          * to uint64_t's to prevent possible overflow in the multiplication.
12946          */
12947         seclen = (uint64_t)dof->dofh_secnum * (uint64_t)dof->dofh_secsize;

12949         if (dof->dofh_secoff > len || seclen > len ||
12950             dof->dofh_secoff + seclen > len) {
12951                 dtrace_dof_error(dof, "truncated section headers");
12952                 return (-1);
12953         }

12955         if (!IS_P2ALIGNED(dof->dofh_secoff, sizeof (uint64_t))) {
12956                 dtrace_dof_error(dof, "misaligned section headers");
12957                 return (-1);
12958         }

12960         if (!IS_P2ALIGNED(dof->dofh_secsize, sizeof (uint64_t))) {
12961                 dtrace_dof_error(dof, "misaligned section size");
12962                 return (-1);
12963         }

12965         /*
12966          * Take an initial pass through the section headers to be sure that
12967          * the headers don't have stray offsets.  If the 'noprobes' flag is
12968          * set, do not permit sections relating to providers, probes, or args.
12969          */
12970         for (i = 0; i < dof->dofh_secnum; i++) {
12971                 dof_sec_t *sec = (dof_sec_t *)(daddr +
12972                     (uintptr_t)dof->dofh_secoff + i * dof->dofh_secsize);

12974                 if (noprobes) {
12975                         switch (sec->dofs_type) {
12976                         case DOF_SECT_PROVIDER:
12977                         case DOF_SECT_PROBES:
12978                         case DOF_SECT_PRARGS:
12979                         case DOF_SECT_PROFFS:
12980                                 dtrace_dof_error(dof, "illegal sections "
12981                                     "for enabling");
12982                                 return (-1);
12983                         }
12984                 }

12986                 if (DOF_SEC_ISLOADABLE(sec->dofs_type) &&
12987                     !(sec->dofs_flags & DOF_SECF_LOAD)) {
12988                         dtrace_dof_error(dof, "loadable section with load "
12989                             "flag unset");
12990                         return (-1);
12991                 }

12993                 if (!(sec->dofs_flags & DOF_SECF_LOAD))
12994                         continue; /* just ignore non-loadable sections */

12996                 if (sec->dofs_align & (sec->dofs_align - 1)) {
```



```
12997                         dtrace_dof_error(dof, "bad section alignment");
12998                         return (-1);
12999                 }

13001                 if (sec->dofs_offset & (sec->dofs_align - 1)) {
13002                         dtrace_dof_error(dof, "misaligned section");
13003                         return (-1);
13004                 }

13006                 if (sec->dofs_offset > len || sec->dofs_size > len ||
13007                     sec->dofs_offset + sec->dofs_size > len) {
13008                         dtrace_dof_error(dof, "corrupt section header");
13009                         return (-1);
13010                 }

13012                 if (sec->dofs_type == DOF_SECT_STRTAB && *((char *)daddr +
13013                     sec->dofs_offset + sec->dofs_size - 1) != '\0') {
13014                         dtrace_dof_error(dof, "non-terminating string table");
13015                         return (-1);
13016                 }
13017         }

13019         /*
13020          * Take a second pass through the sections and locate and perform any
13021          * relocations that are present.  We do this after the first pass to
13022          * be sure that all sections have had their headers validated.
13023          */
13024         for (i = 0; i < dof->dofh_secnum; i++) {
13025                 dof_sec_t *sec = (dof_sec_t *)(daddr +
13026                     (uintptr_t)dof->dofh_secoff + i * dof->dofh_secsize);

13028                 if (!(sec->dofs_flags & DOF_SECF_LOAD))
13029                         continue; /* skip sections that are not loadable */

13031                 switch (sec->dofs_type) {
13032                 case DOF_SECT_URELHDR:
13033                         if (dtrace_dof_relocate(dof, sec, ubase) != 0)
13034                                 return (-1);
13035                         break;
13036                 }
13037         }

13039         if ((enab = *enabp) == NULL)
13040                 enab = *enabp = dtrace_enabling_create(vstate);

13042         for (i = 0; i < dof->dofh_secnum; i++) {
13043                 dof_sec_t *sec = (dof_sec_t *)(daddr +
13044                     (uintptr_t)dof->dofh_secoff + i * dof->dofh_secsize);

13046                 if (sec->dofs_type != DOF_SECT_ECBDESC)
13047                         continue;

13049                 if ((ep = dtrace_dof_ecbdesc(dof, sec, vstate, cr)) == NULL) {
13050                         dtrace_enabling_destroy(enab);
13051                         *enabp = NULL;
13052                         return (-1);
13053                 }

13055                 dtrace_enabling_add(enab, ep);
13056         }

13058         return (0);
13059 }

13061 /*
13062  * Process DOF for any options.  This routine assumes that the DOF has been
```

```
 * at least processed by dtrace_dof_slurp().
 */
static int
dtrace_dof_options(dof_hdr_t *dof, dtrace_state_t *state)
{
        int i, rval;
        uint32_t entsize;
        size_t offs;
        dof_optdesc_t *desc;

        for (i = 0; i < dof->dofh_secnum; i++) {
                dof_sec_t *sec = (dof_sec_t *)((uintptr_t)dof +
                    (uintptr_t)dof->dofh_secoff + i * dof->dofh_secsize);

                if (sec->dofs_type != DOF_SECT_OPTDESC)
                        continue;

                if (sec->dofs_align != sizeof (uint64_t)) {
                        dtrace_dof_error(dof, "bad alignment in "
                            "option description");
                        return (EINVAL);
                }

                if ((entsize = sec->dofs_entsize) == 0) {
                        dtrace_dof_error(dof, "zeroed option entry size");
                        return (EINVAL);
                }

                if (entsize < sizeof (dof_optdesc_t)) {
                        dtrace_dof_error(dof, "bad option entry size");
                        return (EINVAL);
                }

                for (offs = 0; offs < sec->dofs_size; offs += entsize) {
                        desc = (dof_optdesc_t *)((uintptr_t)dof +
                            (uintptr_t)sec->dofs_offset + offs);

                        if (desc->dofo_strtab != DOF_SECIDX_NONE) {
                                dtrace_dof_error(dof, "non-zero option string");
                                return (EINVAL);
                        }

                        if (desc->dofo_value == DTRACEOPT_UNSET) {
                                dtrace_dof_error(dof, "unset option");
                                return (EINVAL);
                        }

                        if ((rval = dtrace_state_option(state,
                            desc->dofo_option, desc->dofo_value)) != 0) {
                                dtrace_dof_error(dof, "rejected option");
                                return (rval);
                        }
                }
        }

        return (0);
}

/*
 * DTrace Consumer State Functions
 */
int
dtrace_dstate_init(dtrace_dstate_t *dstate, size_t size)
{
        size_t hashsize, maxper, min, chunksize = dstate->dtds_chunksize;
        void *base;
```



```
        uintptr_t limit;
        dtrace_dynvar_t *dvar, *next, *start;
        int i;

        ASSERT(MUTEX_HELD(&dtrace_lock));
        ASSERT(dstate->dtds_base == NULL && dstate->dtds_percpu == NULL);

        bzero(dstate, sizeof (dtrace_dstate_t));

        if ((dstate->dtds_chunksize = chunksize) == 0)
                dstate->dtds_chunksize = DTRACE_DYNVAR_CHUNKSIZE;

        if (size < (min = dstate->dtds_chunksize + sizeof (dtrace_dynhash_t)))
                size = min;

        if ((base = kmem_zalloc(size, KM_NOSLEEP | KM_NORMALPRI)) == NULL)
                return (ENOMEM);

        dstate->dtds_size = size;
        dstate->dtds_base = base;
        dstate->dtds_percpu = kmem_cache_alloc(dtrace_state_cache, KM_SLEEP);
        bzero(dstate->dtds_percpu, NCPU * sizeof (dtrace_dstate_percpu_t));

        hashsize = size / (dstate->dtds_chunksize + sizeof (dtrace_dynhash_t));

        if (hashsize != 1 && (hashsize & 1))
                hashsize--;

        dstate->dtds_hashsize = hashsize;
        dstate->dtds_hash = dstate->dtds_base;

        /*
         * Set all of our hash buckets to point to the single sink, and (if
         * it hasn't already been set), set the sink's hash value to be the
         * sink sentinel value.  The sink is needed for dynamic variable
         * lookups to know that they have iterated over an entire, valid hash
         * chain.
         */
        for (i = 0; i < hashsize; i++)
                dstate->dtds_hash[i].dtdh_chain = &dtrace_dynhash_sink;

        if (dtrace_dynhash_sink.dtdv_hashval != DTRACE_DYNHASH_SINK)
                dtrace_dynhash_sink.dtdv_hashval = DTRACE_DYNHASH_SINK;

        /*
         * Determine number of active CPUs.  Divide free list evenly among
         * active CPUs.
         */
        start = (dtrace_dynvar_t *)
            ((uintptr_t)base + hashsize * sizeof (dtrace_dynhash_t));
        limit = (uintptr_t)base + size;

        maxper = (limit - (uintptr_t)start) / NCPU;
        maxper = (maxper / dstate->dtds_chunksize) * dstate->dtds_chunksize;

        for (i = 0; i < NCPU; i++) {
                dstate->dtds_percpu[i].dtdsc_free = dvar = start;

                /*
                 * If we don't even have enough chunks to make it once through
                 * NCPUs, we're just going to allocate everything to the first
                 * CPU.  And if we're on the last CPU, we're going to allocate
                 * whatever is left over.  In either case, we set the limit to
                 * be the limit of the dynamic variable space.
                 */
                if (maxper == 0 || i == NCPU - 1) {
```

```
13195                           limit = (uintptr_t)base + size;
13196                           start = NULL;
13197                   } else {
13198                           limit = (uintptr_t)start + maxper;
13199                           start = (dtrace_dynvar_t *)limit;
13200                   }

13202                   ASSERT(limit <= (uintptr_t)base + size);

13204                   for (;;) {
13205                           next = (dtrace_dynvar_t *)((uintptr_t)dvar +
13206                               dstate->dtds_chunksize);

13208                           if ((uintptr_t)next + dstate->dtds_chunksize >= limit)
13209                                   break;

13211                           dvar->dtdv_next = next;
13212                           dvar = next;
13213                   }

13215                   if (maxper == 0)
13216                           break;
13217           }

13219           return (0);
13220   }

13222   void
13223   dtrace_dstate_fini(dtrace_dstate_t *dstate)
13224   {
13225           ASSERT(MUTEX_HELD(&cpu_lock));

13227           if (dstate->dtds_base == NULL)
13228                   return;

13230           kmem_free(dstate->dtds_base, dstate->dtds_size);
13231           kmem_cache_free(dtrace_state_cache, dstate->dtds_percpu);
13232   }

13234   static void
13235   dtrace_vstate_fini(dtrace_vstate_t *vstate)
13236   {
13237           /*
13238            * Logical XOR, where are you?
13239            */
13240           ASSERT((vstate->dtvs_nglobals == 0) ^ (vstate->dtvs_globals != NULL));

13242           if (vstate->dtvs_nglobals > 0) {
13243                   kmem_free(vstate->dtvs_globals, vstate->dtvs_nglobals *
13244                       sizeof (dtrace_statvar_t *));
13245           }

13247           if (vstate->dtvs_ntlocals > 0) {
13248                   kmem_free(vstate->dtvs_tlocals, vstate->dtvs_ntlocals *
13249                       sizeof (dtrace_difv_t));
13250           }

13252           ASSERT((vstate->dtvs_nlocals == 0) ^ (vstate->dtvs_locals != NULL));

13254           if (vstate->dtvs_nlocals > 0) {
13255                   kmem_free(vstate->dtvs_locals, vstate->dtvs_nlocals *
13256                       sizeof (dtrace_statvar_t *));
13257           }
13258   }

13260   static void
```



```
13261   dtrace_state_clean(dtrace_state_t *state)
13262   {
13263           if (state->dts_activity == DTRACE_ACTIVITY_INACTIVE)
13264                   return;

13266           dtrace_dynvar_clean(&state->dts_vstate.dtvs_dynvars);
13267           dtrace_speculation_clean(state);
13268   }

13270   static void
13271   dtrace_state_deadman(dtrace_state_t *state)
13272   {
13273           hrtime_t now;

13275           dtrace_sync();

13277           now = dtrace_gethrtime();

13279           if (state != dtrace_anon.dta_state &&
13280               now - state->dts_laststatus >= dtrace_deadman_user)
13281                   return;

13283           /*
13284            * We must be sure that dts_alive never appears to be less than the
13285            * value upon entry to dtrace_state_deadman(), and because we lack a
13286            * dtrace_cas64(), we cannot store to it atomically.  We thus instead
13287            * store INT64_MAX to it, followed by a memory barrier, followed by
13288            * the new value.  This assures that dts_alive never appears to be
13289            * less than its true value, regardless of the order in which the
13290            * stores to the underlying storage are issued.
13291            */
13292           state->dts_alive = INT64_MAX;
13293           dtrace_membar_producer();
13294           state->dts_alive = now;
13295   }

13297   dtrace_state_t *
13298   dtrace_state_create(dev_t *devp, cred_t *cr)
13299   {
13300           minor_t minor;
13301           major_t major;
13302           char c[30];
13303           dtrace_state_t *state;
13304           dtrace_optval_t *opt;
13305           int bufsize = NCPU * sizeof (dtrace_buffer_t), i;

13307           ASSERT(MUTEX_HELD(&dtrace_lock));
13308           ASSERT(MUTEX_HELD(&cpu_lock));

13310           minor = (minor_t)(uintptr_t)vmem_alloc(dtrace_minor, 1,
13311               VM_BESTFIT | VM_SLEEP);

13313           if (ddi_soft_state_zalloc(dtrace_softstate, minor) != DDI_SUCCESS) {
13314                   vmem_free(dtrace_minor, (void *)(uintptr_t)minor, 1);
13315                   return (NULL);
13316           }

13318           state = ddi_get_soft_state(dtrace_softstate, minor);
13319           state->dts_epid = DTRACE_EPIDNONE + 1;

13321           (void) snprintf(c, sizeof (c), "dtrace_aggid_%d", minor);
13322           state->dts_aggid_arena = vmem_create(c, (void *)1, UINT32_MAX, 1,
13323               NULL, NULL, NULL, 0, VM_SLEEP | VMC_IDENTIFIER);

13325           if (devp != NULL) {
13326                   major = getemajor(*devp);
```

```
13327         } else {
13328                 major = ddi_driver_major(dtrace_devi);
13329         }

13331         state->dts_dev = makedevice(major, minor);

13333         if (devp != NULL)
13334                 *devp = state->dts_dev;

13336         /*
13337          * We allocate NCPU buffers.  On the one hand, this can be quite
13338          * a bit of memory per instance (nearly 36K on a Starcat).  On the
13339          * other hand, it saves an additional memory reference in the probe
13340          * path.
13341          */
13342         state->dts_buffer = kmem_zalloc(bufsize, KM_SLEEP);
13343         state->dts_aggbuffer = kmem_zalloc(bufsize, KM_SLEEP);
13344         state->dts_cleaner = CYCLIC_NONE;
13345         state->dts_deadman = CYCLIC_NONE;
13346         state->dts_vstate.dtvs_state = state;

13348         for (i = 0; i < DTRACEOPT_MAX; i++)
13349                 state->dts_options[i] = DTRACEOPT_UNSET;

13351         /*
13352          * Set the default options.
13353          */
13354         opt = state->dts_options;
13355         opt[DTRACEOPT_BUFPOLICY] = DTRACEOPT_BUFPOLICY_SWITCH;
13356         opt[DTRACEOPT_BUFRESIZE] = DTRACEOPT_BUFRESIZE_AUTO;
13357         opt[DTRACEOPT_NSPEC] = dtrace_nspec_default;
13358         opt[DTRACEOPT_SPECSIZE] = dtrace_specsize_default;
13359         opt[DTRACEOPT_CPU] = (dtrace_optval_t)DTRACE_CPUALL;
13360         opt[DTRACEOPT_STRSIZE] = dtrace_strsize_default;
13361         opt[DTRACEOPT_STACKFRAMES] = dtrace_stackframes_default;
13362         opt[DTRACEOPT_USTACKFRAMES] = dtrace_ustackframes_default;
13363         opt[DTRACEOPT_CLEANRATE] = dtrace_cleanrate_default;
13364         opt[DTRACEOPT_AGGRATE] = dtrace_aggrate_default;
13365         opt[DTRACEOPT_SWITCHRATE] = dtrace_switchrate_default;
13366         opt[DTRACEOPT_STATUSRATE] = dtrace_statusrate_default;
13367         opt[DTRACEOPT_JSTACKFRAMES] = dtrace_jstackframes_default;
13368         opt[DTRACEOPT_JSTACKSTRSIZE] = dtrace_jstackstrsize_default;

13370         state->dts_activity = DTRACE_ACTIVITY_INACTIVE;

13372         /*
13373          * Depending on the user credentials, we set flag bits which alter probe
13374          * visibility or the amount of destructiveness allowed.  In the case of
13375          * actual anonymous tracing, or the possession of all privileges, all of
13376          * the normal checks are bypassed.
13377          */
13378         if (cr == NULL || PRIV_POLICY_ONLY(cr, PRIV_ALL, B_FALSE)) {
13379                 state->dts_cred.dcr_visible = DTRACE_CRV_ALL;
13380                 state->dts_cred.dcr_action = DTRACE_CRA_ALL;
13381         } else {
13382                 /*
13383                  * Set up the credentials for this instantiation.  We take a
13384                  * hold on the credential to prevent it from disappearing on
13385                  * us; this in turn prevents the zone_t referenced by this
13386                  * credential from disappearing.  This means that we can
13387                  * examine the credential and the zone from probe context.
13388                  */
13389                 crhold(cr);
13390                 state->dts_cred.dcr_cred = cr;

13392                 /*
```



```
13393                  * CRA_PROC means "we have *some* privilege for dtrace" and
13394                  * unlocks the use of variables like pid, zonename, etc.
13395                  */
13396                 if (PRIV_POLICY_ONLY(cr, PRIV_DTRACE_USER, B_FALSE) ||
13397                     PRIV_POLICY_ONLY(cr, PRIV_DTRACE_PROC, B_FALSE)) {
13398                         state->dts_cred.dcr_action |= DTRACE_CRA_PROC;
13399                 }

13401                 /*
13402                  * dtrace_user allows use of syscall and profile providers.
13403                  * If the user also has proc_owner and/or proc_zone, we
13404                  * extend the scope to include additional visibility and
13405                  * destructive power.
13406                  */
13407                 if (PRIV_POLICY_ONLY(cr, PRIV_DTRACE_USER, B_FALSE)) {
13408                         if (PRIV_POLICY_ONLY(cr, PRIV_PROC_OWNER, B_FALSE)) {
13409                                 state->dts_cred.dcr_visible |=
13410                                     DTRACE_CRV_ALLPROC;

13412                                 state->dts_cred.dcr_action |=
13413                                     DTRACE_CRA_PROC_DESTRUCTIVE_ALLUSER;
13414                         }

13416                         if (PRIV_POLICY_ONLY(cr, PRIV_PROC_ZONE, B_FALSE)) {
13417                                 state->dts_cred.dcr_visible |=
13418                                     DTRACE_CRV_ALLZONE;

13420                                 state->dts_cred.dcr_action |=
13421                                     DTRACE_CRA_PROC_DESTRUCTIVE_ALLZONE;
13422                         }

13424                         /*
13425                          * If we have all privs in whatever zone this is,
13426                          * we can do destructive things to processes which
13427                          * have altered credentials.
13428                          */
13429                         if (priv_isequalset(priv_getset(cr, PRIV_EFFECTIVE),
13430                             cr->cr_zone->zone_privset)) {
13431                                 state->dts_cred.dcr_action |=
13432                                     DTRACE_CRA_PROC_DESTRUCTIVE_CREDCHG;
13433                         }
13434                 }

13436                 /*
13437                  * Holding the dtrace_kernel privilege also implies that
13438                  * the user has the dtrace_user privilege from a visibility
13439                  * perspective.  But without further privileges, some
13440                  * destructive actions are not available.
13441                  */
13442                 if (PRIV_POLICY_ONLY(cr, PRIV_DTRACE_KERNEL, B_FALSE)) {
13443                         /*
13444                          * Make all probes in all zones visible.  However,
13445                          * this doesn't mean that all actions become available
13446                          * to all zones.
13447                          */
13448                         state->dts_cred.dcr_visible |= DTRACE_CRV_KERNEL |
13449                             DTRACE_CRV_ALLPROC | DTRACE_CRV_ALLZONE;

13451                         state->dts_cred.dcr_action |= DTRACE_CRA_KERNEL |
13452                             DTRACE_CRA_PROC;
13453                         /*
13454                          * Holding proc_owner means that destructive actions
13455                          * for *this* zone are allowed.
13456                          */
13457                         if (PRIV_POLICY_ONLY(cr, PRIV_PROC_OWNER, B_FALSE))
13458                                 state->dts_cred.dcr_action |=
```

```
13459                                   DTRACE_CRA_PROC_DESTRUCTIVE_ALLUSER;

13461                           /*
13462                            * Holding proc_zone means that destructive actions
13463                            * for this user/group ID in all zones is allowed.
13464                            */
13465                           if (PRIV_POLICY_ONLY(cr, PRIV_PROC_ZONE, B_FALSE))
13466                                   state->dts_cred.dcr_action |=
13467                                       DTRACE_CRA_PROC_DESTRUCTIVE_ALLZONE;

13469                           /*
13470                            * If we have all privs in whatever zone this is,
13471                            * we can do destructive things to processes which
13472                            * have altered credentials.
13473                            */
13474                           if (priv_isequalset(priv_getset(cr, PRIV_EFFECTIVE),
13475                               cr->cr_zone->zone_privset)) {
13476                                   state->dts_cred.dcr_action |=
13477                                       DTRACE_CRA_PROC_DESTRUCTIVE_CREDCHG;
13478                           }
13479                   }

13481                   /*
13482                    * Holding the dtrace_proc privilege gives control over fasttrap
13483                    * and pid providers.  We need to grant wider destructive
13484                    * privileges in the event that the user has proc_owner and/or
13485                    * proc_zone.
13486                    */
13487                   if (PRIV_POLICY_ONLY(cr, PRIV_DTRACE_PROC, B_FALSE)) {
13488                           if (PRIV_POLICY_ONLY(cr, PRIV_PROC_OWNER, B_FALSE))
13489                                   state->dts_cred.dcr_action |=
13490                                       DTRACE_CRA_PROC_DESTRUCTIVE_ALLUSER;

13492                           if (PRIV_POLICY_ONLY(cr, PRIV_PROC_ZONE, B_FALSE))
13493                                   state->dts_cred.dcr_action |=
13494                                       DTRACE_CRA_PROC_DESTRUCTIVE_ALLZONE;
13495                   }
13496           }

13498           return (state);
13499   }

13501   static int
13502   dtrace_state_buffer(dtrace_state_t *state, dtrace_buffer_t *buf, int which)
13503   {
13504           dtrace_optval_t *opt = state->dts_options, size;
13505           processorid_t cpu;
13506           int flags = 0, rval, factor, divisor = 1;

13508           ASSERT(MUTEX_HELD(&dtrace_lock));
13509           ASSERT(MUTEX_HELD(&cpu_lock));
13510           ASSERT(which < DTRACEOPT_MAX);
13511           ASSERT(state->dts_activity == DTRACE_ACTIVITY_INACTIVE ||
13512               (state == dtrace_anon.dta_state &&
13513               state->dts_activity == DTRACE_ACTIVITY_ACTIVE));

13515           if (opt[which] == DTRACEOPT_UNSET || opt[which] == 0)
13516                   return (0);

13518           if (opt[DTRACEOPT_CPU] != DTRACEOPT_UNSET)
13519                   cpu = opt[DTRACEOPT_CPU];

13521           if (which == DTRACEOPT_SPECSIZE)
13522                   flags |= DTRACEBUF_NOSWITCH;

13524           if (which == DTRACEOPT_BUFSIZE) {
```



```
13525                   if (opt[DTRACEOPT_BUFPOLICY] == DTRACEOPT_BUFPOLICY_RING)
13526                           flags |= DTRACEBUF_RING;

13528                   if (opt[DTRACEOPT_BUFPOLICY] == DTRACEOPT_BUFPOLICY_FILL)
13529                           flags |= DTRACEBUF_FILL;

13531                   if (state != dtrace_anon.dta_state ||
13532                       state->dts_activity != DTRACE_ACTIVITY_ACTIVE)
13533                           flags |= DTRACEBUF_INACTIVE;
13534           }

13536           for (size = opt[which]; size >= sizeof (uint64_t); size /= divisor) {
13537                   /*
13538                    * The size must be 8-byte aligned.  If the size is not 8-byte
13539                    * aligned, drop it down by the difference.
13540                    */
13541                   if (size & (sizeof (uint64_t) - 1))
13542                           size -= size & (sizeof (uint64_t) - 1);

13544                   if (size < state->dts_reserve) {
13545                           /*
13546                            * Buffers always must be large enough to accommodate
13547                            * their prereserved space.  We return E2BIG instead
13548                            * of ENOMEM in this case to allow for user-level
13549                            * software to differentiate the cases.
13550                            */
13551                           return (E2BIG);
13552                   }

13554                   rval = dtrace_buffer_alloc(buf, size, flags, cpu, &factor);

13556                   if (rval != ENOMEM) {
13557                           opt[which] = size;
13558                           return (rval);
13559                   }

13561                   if (opt[DTRACEOPT_BUFRESIZE] == DTRACEOPT_BUFRESIZE_MANUAL)
13562                           return (rval);

13564                   for (divisor = 2; divisor < factor; divisor <<= 1)
13565                           continue;
13566           }

13568           return (ENOMEM);
13569   }

13571   static int
13572   dtrace_state_buffers(dtrace_state_t *state)
13573   {
13574           dtrace_speculation_t *spec = state->dts_speculations;
13575           int rval, i;

13577           if ((rval = dtrace_state_buffer(state, state->dts_buffer,
13578               DTRACEOPT_BUFSIZE)) != 0)
13579                   return (rval);

13581           if ((rval = dtrace_state_buffer(state, state->dts_aggbuffer,
13582               DTRACEOPT_AGGSIZE)) != 0)
13583                   return (rval);

13585           for (i = 0; i < state->dts_nspeculations; i++) {
13586                   if ((rval = dtrace_state_buffer(state,
13587                       spec[i].dtsp_buffer, DTRACEOPT_SPECSIZE)) != 0)
13588                           return (rval);
13589           }
```

```
13591         return (0);
13592 }

13594 static void
13595 dtrace_state_prereserve(dtrace_state_t *state)
13596 {
13597         dtrace_ecb_t *ecb;
13598         dtrace_probe_t *probe;

13600         state->dts_reserve = 0;

13602         if (state->dts_options[DTRACEOPT_BUFPOLICY] != DTRACEOPT_BUFPOLICY_FILL)
13603                 return;

13605         /*
13606          * If our buffer policy is a "fill" buffer policy, we need to set the
13607          * prereserved space to be the space required by the END probes.
13608          */
13609         probe = dtrace_probes[dtrace_probeid_end - 1];
13610         ASSERT(probe != NULL);

13612         for (ecb = probe->dtpr_ecb; ecb != NULL; ecb = ecb->dte_next) {
13613                 if (ecb->dte_state != state)
13614                         continue;

13616                 state->dts_reserve += ecb->dte_needed + ecb->dte_alignment;
13617         }
13618 }

13620 static int
13621 dtrace_state_go(dtrace_state_t *state, processorid_t *cpu)
13622 {
13623         dtrace_optval_t *opt = state->dts_options, sz, nspec;
13624         dtrace_speculation_t *spec;
13625         dtrace_buffer_t *buf;
13626         cyc_handler_t hdlr;
13627         cyc_time_t when;
13628         int rval = 0, i, bufsize = NCPU * sizeof (dtrace_buffer_t);
13629         dtrace_icookie_t cookie;

13631         mutex_enter(&cpu_lock);
13632         mutex_enter(&dtrace_lock);

13634         if (state->dts_activity != DTRACE_ACTIVITY_INACTIVE) {
13635                 rval = EBUSY;
13636                 goto out;
13637         }

13639         /*
13640          * Before we can perform any checks, we must prime all of the
13641          * retained enablings that correspond to this state.
13642          */
13643         dtrace_enabling_prime(state);

13645         if (state->dts_destructive && !state->dts_cred.dcr_destructive) {
13646                 rval = EACCES;
13647                 goto out;
13648         }

13650         dtrace_state_prereserve(state);

13652         /*
13653          * Now we want to do is try to allocate our speculations.
13654          * We do not automatically resize the number of speculations; if
13655          * this fails, we will fail the operation.
13656          */
```



```
13657         nspec = opt[DTRACEOPT_NSPEC];
13658         ASSERT(nspec != DTRACEOPT_UNSET);

13660         if (nspec > INT_MAX) {
13661                 rval = ENOMEM;
13662                 goto out;
13663         }

13665         spec = kmem_zalloc(nspec * sizeof (dtrace_speculation_t),
13666             KM_NOSLEEP | KM_NORMALPRI);

13668         if (spec == NULL) {
13669                 rval = ENOMEM;
13670                 goto out;
13671         }

13673         state->dts_speculations = spec;
13674         state->dts_nspeculations = (int)nspec;

13676         for (i = 0; i < nspec; i++) {
13677                 if ((buf = kmem_zalloc(bufsize,
13678                     KM_NOSLEEP | KM_NORMALPRI)) == NULL) {
13679                         rval = ENOMEM;
13680                         goto err;
13681                 }

13683                 spec[i].dtsp_buffer = buf;
13684         }

13686         if (opt[DTRACEOPT_GRABANON] != DTRACEOPT_UNSET) {
13687                 if (dtrace_anon.dta_state == NULL) {
13688                         rval = ENOENT;
13689                         goto out;
13690                 }

13692                 if (state->dts_necbs != 0) {
13693                         rval = EALREADY;
13694                         goto out;
13695                 }

13697                 state->dts_anon = dtrace_anon_grab();
13698                 ASSERT(state->dts_anon != NULL);
13699                 state = state->dts_anon;

13701                 /*
13702                  * We want "grabanon" to be set in the grabbed state, so we'll
13703                  * copy that option value from the grabbing state into the
13704                  * grabbed state.
13705                  */
13706                 state->dts_options[DTRACEOPT_GRABANON] =
13707                     opt[DTRACEOPT_GRABANON];

13709                 *cpu = dtrace_anon.dta_beganon;

13711                 /*
13712                  * If the anonymous state is active (as it almost certainly
13713                  * is if the anonymous enabling ultimately matched anything),
13714                  * we don't allow any further option processing -- but we
13715                  * don't return failure.
13716                  */
13717                 if (state->dts_activity != DTRACE_ACTIVITY_INACTIVE)
13718                         goto out;
13719         }

13721         if (opt[DTRACEOPT_AGGSIZE] != DTRACEOPT_UNSET &&
13722             opt[DTRACEOPT_AGGSIZE] != 0) {
```

```
13723                 if (state->dts_aggregations == NULL) {
13724                         /*
13725                          * We're not going to create an aggregation buffer
13726                          * because we don't have any ECBs that contain
13727                          * aggregations -- set this option to 0.
13728                          */
13729                         opt[DTRACEOPT_AGGSIZE] = 0;
13730                 } else {
13731                         /*
13732                          * If we have an aggregation buffer, we must also have
13733                          * a buffer to use as scratch.
13734                          */
13735                         if (opt[DTRACEOPT_BUFSIZE] == DTRACEOPT_UNSET ||
13736                             opt[DTRACEOPT_BUFSIZE] < state->dts_needed) {
13737                                 opt[DTRACEOPT_BUFSIZE] = state->dts_needed;
13738                         }
13739                 }
13740         }

13742         if (opt[DTRACEOPT_SPECSIZE] != DTRACEOPT_UNSET &&
13743             opt[DTRACEOPT_SPECSIZE] != 0) {
13744                 if (!state->dts_speculates) {
13745                         /*
13746                          * We're not going to create speculation buffers
13747                          * because we don't have any ECBs that actually
13748                          * speculate -- set the speculation size to 0.
13749                          */
13750                         opt[DTRACEOPT_SPECSIZE] = 0;
13751                 }
13752         }

13754         /*
13755          * The bare minimum size for any buffer that we're actually going to
13756          * do anything to is sizeof (uint64_t).
13757          */
13758         sz = sizeof (uint64_t);

13760         if ((state->dts_needed != 0 && opt[DTRACEOPT_BUFSIZE] < sz) ||
13761             (state->dts_speculates && opt[DTRACEOPT_SPECSIZE] < sz) ||
13762             (state->dts_aggregations != NULL && opt[DTRACEOPT_AGGSIZE] < sz)) {
13763                 /*
13764                  * A buffer size has been explicitly set to 0 (or to a size
13765                  * that will be adjusted to 0) and we need the space -- we
13766                  * need to return failure.  We return ENOSPC to differentiate
13767                  * it from failing to allocate a buffer due to failure to meet
13768                  * the reserve (for which we return E2BIG).
13769                  */
13770                 rval = ENOSPC;
13771                 goto out;
13772         }

13774         if ((rval = dtrace_state_buffers(state)) != 0)
13775                 goto err;

13777         if ((sz = opt[DTRACEOPT_DYNVARSIZE]) == DTRACEOPT_UNSET)
13778                 sz = dtrace_dstate_defsize;

13780         do {
13781                 rval = dtrace_dstate_init(&state->dts_vstate.dtvs_dynvars, sz);

13783                 if (rval == 0)
13784                         break;

13786                 if (opt[DTRACEOPT_BUFRESIZE] == DTRACEOPT_BUFRESIZE_MANUAL)
13787                         goto err;
13788         } while (sz >>= 1);
```



```
13790         opt[DTRACEOPT_DYNVARSIZE] = sz;

13792         if (rval != 0)
13793                 goto err;

13795         if (opt[DTRACEOPT_STATUSRATE] > dtrace_statusrate_max)
13796                 opt[DTRACEOPT_STATUSRATE] = dtrace_statusrate_max;

13798         if (opt[DTRACEOPT_CLEANRATE] == 0)
13799                 opt[DTRACEOPT_CLEANRATE] = dtrace_cleanrate_max;

13801         if (opt[DTRACEOPT_CLEANRATE] < dtrace_cleanrate_min)
13802                 opt[DTRACEOPT_CLEANRATE] = dtrace_cleanrate_min;

13804         if (opt[DTRACEOPT_CLEANRATE] > dtrace_cleanrate_max)
13805                 opt[DTRACEOPT_CLEANRATE] = dtrace_cleanrate_max;

13807         hdlr.cyh_func = (cyc_func_t)dtrace_state_clean;
13808         hdlr.cyh_arg = state;
13809         hdlr.cyh_level = CY_LOW_LEVEL;

13811         when.cyt_when = 0;
13812         when.cyt_interval = opt[DTRACEOPT_CLEANRATE];

13814         state->dts_cleaner = cyclic_add(&hdlr, &when);

13816         hdlr.cyh_func = (cyc_func_t)dtrace_state_deadman;
13817         hdlr.cyh_arg = state;
13818         hdlr.cyh_level = CY_LOW_LEVEL;

13820         when.cyt_when = 0;
13821         when.cyt_interval = dtrace_deadman_interval;

13823         state->dts_alive = state->dts_laststatus = dtrace_gethrtime();
13824         state->dts_deadman = cyclic_add(&hdlr, &when);

13826         state->dts_activity = DTRACE_ACTIVITY_WARMUP;

13828         if (state->dts_getf != 0 &&
13829             !(state->dts_cred.dcr_visible & DTRACE_CRV_KERNEL)) {
13830                 /*
13831                  * We don't have kernel privs but we have at least one call
13832                  * to getf(); we need to bump our zone's count, and (if
13833                  * this is the first enabling to have an unprivileged call
13834                  * to getf()) we need to hook into closef().
13835                  */
13836                 state->dts_cred.dcr_cred->cr_zone->zone_dtrace_getf++;

13838                 if (dtrace_getf++ == 0) {
13839                         ASSERT(dtrace_closef == NULL);
13840                         dtrace_closef = dtrace_getf_barrier;
13841                 }
13842         }

13844         /*
13845          * Now it's time to actually fire the BEGIN probe.  We need to disable
13846          * interrupts here both to record the CPU on which we fired the BEGIN
13847          * probe (the data from this CPU will be processed first at user
13848          * level) and to manually activate the buffer for this CPU.
13849          */
13850         cookie = dtrace_interrupt_disable();
13851         *cpu = CPU->cpu_id;
13852         ASSERT(state->dts_buffer[*cpu].dtb_flags & DTRACEBUF_INACTIVE);
13853         state->dts_buffer[*cpu].dtb_flags &= ~DTRACEBUF_INACTIVE;
```

```
13855         dtrace_probe(dtrace_probeid_begin,
13856             (uint64_t)(uintptr_t)state, 0, 0, 0, 0);
13857         dtrace_interrupt_enable(cookie);
13858         /*
13859          * We may have had an exit action from a BEGIN probe; only change our
13860          * state to ACTIVE if we're still in WARMUP.
13861          */
13862         ASSERT(state->dts_activity == DTRACE_ACTIVITY_WARMUP ||
13863             state->dts_activity == DTRACE_ACTIVITY_DRAINING);

13865         if (state->dts_activity == DTRACE_ACTIVITY_WARMUP)
13866                 state->dts_activity = DTRACE_ACTIVITY_ACTIVE;

13868         /*
13869          * Regardless of whether or not now we're in ACTIVE or DRAINING, we
13870          * want each CPU to transition its principal buffer out of the
13871          * INACTIVE state.  Doing this assures that no CPU will suddenly begin
13872          * processing an ECB halfway down a probe's ECB chain; all CPUs will
13873          * atomically transition from processing none of a state's ECBs to
13874          * processing all of them.
13875          */
13876         dtrace_xcall(DTRACE_CPUALL,
13877             (dtrace_xcall_t)dtrace_buffer_activate, state);
13878         goto out;

13880 err:
13881         dtrace_buffer_free(state->dts_buffer);
13882         dtrace_buffer_free(state->dts_aggbuffer);

13884         if ((nspec = state->dts_nspeculations) == 0) {
13885                 ASSERT(state->dts_speculations == NULL);
13886                 goto out;
13887         }

13889         spec = state->dts_speculations;
13890         ASSERT(spec != NULL);

13892         for (i = 0; i < state->dts_nspeculations; i++) {
13893                 if ((buf = spec[i].dtsp_buffer) == NULL)
13894                         break;

13896                 dtrace_buffer_free(buf);
13897                 kmem_free(buf, bufsize);
13898         }

13900         kmem_free(spec, nspec * sizeof (dtrace_speculation_t));
13901         state->dts_nspeculations = 0;
13902         state->dts_speculations = NULL;

13904 out:
13905         mutex_exit(&dtrace_lock);
13906         mutex_exit(&cpu_lock);

13908         return (rval);
13909 }

13911 static int
13912 dtrace_state_stop(dtrace_state_t *state, processorid_t *cpu)
13913 {
13914         dtrace_icookie_t cookie;

13916         ASSERT(MUTEX_HELD(&dtrace_lock));

13918         if (state->dts_activity != DTRACE_ACTIVITY_ACTIVE &&
13919             state->dts_activity != DTRACE_ACTIVITY_DRAINING)
13920                 return (EINVAL);
```



```
13922         /*
13923          * We'll set the activity to DTRACE_ACTIVITY_DRAINING, and issue a sync
13924          * to be sure that every CPU has seen it.  See below for the details
13925          * on why this is done.
13926          */
13927         state->dts_activity = DTRACE_ACTIVITY_DRAINING;
13928         dtrace_sync();

13930         /*
13931          * By this point, it is impossible for any CPU to be still processing
13932          * with DTRACE_ACTIVITY_ACTIVE.  We can thus set our activity to
13933          * DTRACE_ACTIVITY_COOLDOWN and know that we're not racing with any
13934          * other CPU in dtrace_buffer_reserve().  This allows dtrace_probe()
13935          * and callees to know that the activity is DTRACE_ACTIVITY_COOLDOWN
13936          * iff we're in the END probe.
13937          */
13938         state->dts_activity = DTRACE_ACTIVITY_COOLDOWN;
13939         dtrace_sync();
13940         ASSERT(state->dts_activity == DTRACE_ACTIVITY_COOLDOWN);

13942         /*
13943          * Finally, we can release the reserve and call the END probe.  We
13944          * disable interrupts across calling the END probe to allow us to
13945          * return the CPU on which we actually called the END probe.  This
13946          * allows user-land to be sure that this CPU's principal buffer is
13947          * processed last.
13948          */
13949         state->dts_reserve = 0;

13951         cookie = dtrace_interrupt_disable();
13952         *cpu = CPU->cpu_id;
13953         dtrace_probe(dtrace_probeid_end,
13954             (uint64_t)(uintptr_t)state, 0, 0, 0, 0);
13955         dtrace_interrupt_enable(cookie);

13957         state->dts_activity = DTRACE_ACTIVITY_STOPPED;
13958         dtrace_sync();

13960         if (state->dts_getf != 0 &&
13961             !(state->dts_cred.dcr_visible & DTRACE_CRV_KERNEL)) {
13962                 /*
13963                  * We don't have kernel privs but we have at least one call
13964                  * to getf(); we need to lower our zone's count, and (if
13965                  * this is the last enabling to have an unprivileged call
13966                  * to getf()) we need to clear the closef() hook.
13967                  */
13968                 ASSERT(state->dts_cred.dcr_cred->cr_zone->zone_dtrace_getf > 0);
13969                 ASSERT(dtrace_closef == dtrace_getf_barrier);
13970                 ASSERT(dtrace_getf > 0);

13972                 state->dts_cred.dcr_cred->cr_zone->zone_dtrace_getf--;

13974                 if (--dtrace_getf == 0)
13975                         dtrace_closef = NULL;
13976         }

13978         return (0);
13979 }

13981 static int
13982 dtrace_state_option(dtrace_state_t *state, dtrace_optid_t option,
13983     dtrace_optval_t val)
13984 {
13985         ASSERT(MUTEX_HELD(&dtrace_lock));
```

```
13987         if (state->dts_activity != DTRACE_ACTIVITY_INACTIVE)
13988                 return (EBUSY);
13989
13990         if (option >= DTRACEOPT_MAX)
13991                 return (EINVAL);
13992
13993         if (option != DTRACEOPT_CPU && val < 0)
13994                 return (EINVAL);
13995
13996         switch (option) {
13997         case DTRACEOPT_DESTRUCTIVE:
13998                 if (dtrace_destructive_disallow)
13999                         return (EACCES);
14000
14001                 state->dts_cred.dcr_destructive = 1;
14002                 break;
14003
14004         case DTRACEOPT_BUFSIZE:
14005         case DTRACEOPT_DYNVARSIZE:
14006         case DTRACEOPT_AGGSIZE:
14007         case DTRACEOPT_SPECSIZE:
14008         case DTRACEOPT_STRSIZE:
14009                 if (val < 0)
14010                         return (EINVAL);
14011
14012                 if (val >= LONG_MAX) {
14013                         /*
14014                          * If this is an otherwise negative value, set it to
14015                          * the highest multiple of 128m less than LONG_MAX.
14016                          * Technically, we're adjusting the size without
14017                          * regard to the buffer resizing policy, but in fact,
14018                          * this has no effect -- if we set the buffer size to
14019                          * ~LONG_MAX and the buffer policy is ultimately set to
14020                          * be "manual", the buffer allocation is guaranteed to
14021                          * fail, if only because the allocation requires two
14022                          * buffers.  (We set the the size to the highest
14023                          * multiple of 128m because it ensures that the size
14024                          * will remain a multiple of a megabyte when
14025                          * repeatedly halved -- all the way down to 15m.)
14026                          */
14027                         val = LONG_MAX - (1 << 27) + 1;
14028                 }
14029         }
14030
14031         state->dts_options[option] = val;
14032
14033         return (0);
14034 }
14035
14036 static void
14037 dtrace_state_destroy(dtrace_state_t *state)
14038 {
14039         dtrace_ecb_t *ecb;
14040         dtrace_vstate_t *vstate = &state->dts_vstate;
14041         minor_t minor = getminor(state->dts_dev);
14042         int i, bufsize = NCPU * sizeof (dtrace_buffer_t);
14043         dtrace_speculation_t *spec = state->dts_speculations;
14044         int nspec = state->dts_nspeculations;
14045         uint32_t match;
14046
14047         ASSERT(MUTEX_HELD(&dtrace_lock));
14048         ASSERT(MUTEX_HELD(&cpu_lock));
14049
14050         /*
14051          * First, retract any retained enablings for this state.
14052          */
```



```
14053         dtrace_enabling_retract(state);
14054         ASSERT(state->dts_nretained == 0);
14055
14056         if (state->dts_activity == DTRACE_ACTIVITY_ACTIVE ||
14057             state->dts_activity == DTRACE_ACTIVITY_DRAINING) {
14058                 /*
14059                  * We have managed to come into dtrace_state_destroy() on a
14060                  * hot enabling -- almost certainly because of a disorderly
14061                  * shutdown of a consumer.  (That is, a consumer that is
14062                  * exiting without having called dtrace_stop().) In this case,
14063                  * we're going to set our activity to be KILLED, and then
14064                  * issue a sync to be sure that everyone is out of probe
14065                  * context before we start blowing away ECBs.
14066                  */
14067                 state->dts_activity = DTRACE_ACTIVITY_KILLED;
14068                 dtrace_sync();
14069         }
14070
14071         /*
14072          * Release the credential hold we took in dtrace_state_create().
14073          */
14074         if (state->dts_cred.dcr_cred != NULL)
14075                 crfree(state->dts_cred.dcr_cred);
14076
14077         /*
14078          * Now we can safely disable and destroy any enabled probes.  Because
14079          * any DTRACE_PRIV_KERNEL probes may actually be slowing our progress
14080          * (especially if they're all enabled), we take two passes through the
14081          * ECBs:  in the first, we disable just DTRACE_PRIV_KERNEL probes, and
14082          * in the second we disable whatever is left over.
14083          */
14084         for (match = DTRACE_PRIV_KERNEL; ; match = 0) {
14085                 for (i = 0; i < state->dts_necbs; i++) {
14086                         if ((ecb = state->dts_ecbs[i]) == NULL)
14087                                 continue;
14088
14089                         if (match && ecb->dte_probe != NULL) {
14090                                 dtrace_probe_t *probe = ecb->dte_probe;
14091                                 dtrace_provider_t *prov = probe->dtpr_provider;
14092
14093                                 if (!(prov->dtpv_priv.dtpp_flags & match))
14094                                         continue;
14095                         }
14096
14097                         dtrace_ecb_disable(ecb);
14098                         dtrace_ecb_destroy(ecb);
14099                 }
14100
14101                 if (!match)
14102                         break;
14103         }
14104
14105         /*
14106          * Before we free the buffers, perform one more sync to assure that
14107          * every CPU is out of probe context.
14108          */
14109         dtrace_sync();
14110
14111         dtrace_buffer_free(state->dts_buffer);
14112         dtrace_buffer_free(state->dts_aggbuffer);
14113
14114         for (i = 0; i < nspec; i++)
14115                 dtrace_buffer_free(spec[i].dtsp_buffer);
14116
14117         if (state->dts_cleaner != CYCLIC_NONE)
14118                 cyclic_remove(state->dts_cleaner);
```

```
14120         if (state->dts_deadman != CYCLIC_NONE)
14121                 cyclic_remove(state->dts_deadman);
14122
14123         dtrace_dstate_fini(&vstate->dtvs_dynvars);
14124         dtrace_vstate_fini(vstate);
14125         kmem_free(state->dts_ecbs, state->dts_necbs * sizeof (dtrace_ecb_t *));
14126
14127         if (state->dts_aggregations != NULL) {
14128 #ifdef DEBUG
14129                 for (i = 0; i < state->dts_naggregations; i++)
14130                         ASSERT(state->dts_aggregations[i] == NULL);
14131 #endif
14132                 ASSERT(state->dts_naggregations > 0);
14133                 kmem_free(state->dts_aggregations,
14134                     state->dts_naggregations * sizeof (dtrace_aggregation_t *));
14135         }
14136
14137         kmem_free(state->dts_buffer, bufsize);
14138         kmem_free(state->dts_aggbuffer, bufsize);
14139
14140         for (i = 0; i < nspec; i++)
14141                 kmem_free(spec[i].dtsp_buffer, bufsize);
14142
14143         kmem_free(spec, nspec * sizeof (dtrace_speculation_t));
14144
14145         dtrace_format_destroy(state);
14146
14147         vmem_destroy(state->dts_aggid_arena);
14148         ddi_soft_state_free(dtrace_softstate, minor);
14149         vmem_free(dtrace_minor, (void *)(uintptr_t)minor, 1);
14150 }
14151
14152 /*
14153  * DTrace Anonymous Enabling Functions
14154  */
14155 static dtrace_state_t *
14156 dtrace_anon_grab(void)
14157 {
14158         dtrace_state_t *state;
14159
14160         ASSERT(MUTEX_HELD(&dtrace_lock));
14161
14162         if ((state = dtrace_anon.dta_state) == NULL) {
14163                 ASSERT(dtrace_anon.dta_enabling == NULL);
14164                 return (NULL);
14165         }
14166
14167         ASSERT(dtrace_anon.dta_enabling != NULL);
14168         ASSERT(dtrace_retained != NULL);
14169
14170         dtrace_enabling_destroy(dtrace_anon.dta_enabling);
14171         dtrace_anon.dta_enabling = NULL;
14172         dtrace_anon.dta_state = NULL;
14173
14174         return (state);
14175 }
14176
14177 static void
14178 dtrace_anon_property(void)
14179 {
14180         int i, rv;
14181         dtrace_state_t *state;
14182         dof_hdr_t *dof;
14183         char c[32];             /* enough for "dof-data-" + digits */
```



```
14185         ASSERT(MUTEX_HELD(&dtrace_lock));
14186         ASSERT(MUTEX_HELD(&cpu_lock));
14187
14188         for (i = 0; ; i++) {
14189                 (void) snprintf(c, sizeof (c), "dof-data-%d", i);
14190
14191                 dtrace_err_verbose = 1;
14192
14193                 if ((dof = dtrace_dof_property(c)) == NULL) {
14194                         dtrace_err_verbose = 0;
14195                         break;
14196                 }
14197
14198                 /*
14199                  * We want to create anonymous state, so we need to transition
14200                  * the kernel debugger to indicate that DTrace is active.  If
14201                  * this fails (e.g. because the debugger has modified text in
14202                  * some way), we won't continue with the processing.
14203                  */
14204                 if (kdi_dtrace_set(KDI_DTSET_DTRACE_ACTIVATE) != 0) {
14205                         cmn_err(CE_NOTE, "kernel debugger active; anonymous "
14206                             "enabling ignored.");
14207                         dtrace_dof_destroy(dof);
14208                         break;
14209                 }
14210
14211                 /*
14212                  * If we haven't allocated an anonymous state, we'll do so now.
14213                  */
14214                 if ((state = dtrace_anon.dta_state) == NULL) {
14215                         state = dtrace_state_create(NULL, NULL);
14216                         dtrace_anon.dta_state = state;
14217
14218                         if (state == NULL) {
14219                                 /*
14220                                  * This basically shouldn't happen:  the only
14221                                  * failure mode from dtrace_state_create() is a
14222                                  * failure of ddi_soft_state_zalloc() that
14223                                  * itself should never happen.  Still, the
14224                                  * interface allows for a failure mode, and
14225                                  * we want to fail as gracefully as possible:
14226                                  * we'll emit an error message and cease
14227                                  * processing anonymous state in this case.
14228                                  */
14229                                 cmn_err(CE_WARN, "failed to create "
14230                                     "anonymous state");
14231                                 dtrace_dof_destroy(dof);
14232                                 break;
14233                         }
14234                 }
14235
14236                 rv = dtrace_dof_slurp(dof, &state->dts_vstate, CRED(),
14237                     &dtrace_anon.dta_enabling, 0, B_TRUE);
14238
14239                 if (rv == 0)
14240                         rv = dtrace_dof_options(dof, state);
14241
14242                 dtrace_err_verbose = 0;
14243                 dtrace_dof_destroy(dof);
14244
14245                 if (rv != 0) {
14246                         /*
14247                          * This is malformed DOF; chuck any anonymous state
14248                          * that we created.
14249                          */
14250                         ASSERT(dtrace_anon.dta_enabling == NULL);
```

```
14251                         dtrace_state_destroy(state);
14252                         dtrace_anon.dta_state = NULL;
14253                         break;
14254                 }
14255
14256                 ASSERT(dtrace_anon.dta_enabling != NULL);
14257         }
14258
14259         if (dtrace_anon.dta_enabling != NULL) {
14260                 int rval;
14261
14262                 /*
14263                  * dtrace_enabling_retain() can only fail because we are
14264                  * trying to retain more enablings than are allowed -- but
14265                  * we only have one anonymous enabling, and we are guaranteed
14266                  * to be allowed at least one retained enabling; we assert
14267                  * that dtrace_enabling_retain() returns success.
14268                  */
14269                 rval = dtrace_enabling_retain(dtrace_anon.dta_enabling);
14270                 ASSERT(rval == 0);
14271
14272                 dtrace_enabling_dump(dtrace_anon.dta_enabling);
14273         }
14274 }
14275
14276 /*
14277  * DTrace Helper Functions
14278  */
14279 static void
14280 dtrace_helper_trace(dtrace_helper_action_t *helper,
14281     dtrace_mstate_t *mstate, dtrace_vstate_t *vstate, int where)
14282 {
14283         uint32_t size, next, nnext, i;
14284         dtrace_helptrace_t *ent;
14285         uint16_t flags = cpu_core[CPU->cpu_id].cpuc_dtrace_flags;
14286
14287         if (!dtrace_helptrace_enabled)
14288                 return;
14289
14290         ASSERT(vstate->dtvs_nlocals <= dtrace_helptrace_nlocals);
14291
14292         /*
14293          * What would a tracing framework be without its own tracing
14294          * framework?  (Well, a hell of a lot simpler, for starters...)
14295          */
14296         size = sizeof (dtrace_helptrace_t) + dtrace_helptrace_nlocals *
14297             sizeof (uint64_t) - sizeof (uint64_t);
14298
14299         /*
14300          * Iterate until we can allocate a slot in the trace buffer.
14301          */
14302         do {
14303                 next = dtrace_helptrace_next;
14304
14305                 if (next + size < dtrace_helptrace_bufsize) {
14306                         nnext = next + size;
14307                 } else {
14308                         nnext = size;
14309                 }
14310         } while (dtrace_cas32(&dtrace_helptrace_next, next, nnext) != next);
14311
14312         /*
14313          * We have our slot; fill it in.
14314          */
14315         if (nnext == size)
14316                 next = 0;
```



```
14317
14318         ent = (dtrace_helptrace_t *)&dtrace_helptrace_buffer[next];
14319         ent->dtht_helper = helper;
14320         ent->dtht_where = where;
14321         ent->dtht_nlocals = vstate->dtvs_nlocals;
14322
14323         ent->dtht_fltoffs = (mstate->dtms_present & DTRACE_MSTATE_FLTOFFS) ?
14324             mstate->dtms_fltoffs : -1;
14325         ent->dtht_fault = DTRACE_FLAGS2FLT(flags);
14326         ent->dtht_illval = cpu_core[CPU->cpu_id].cpuc_dtrace_illval;
14327
14328         for (i = 0; i < vstate->dtvs_nlocals; i++) {
14329                 dtrace_statvar_t *svar;
14330
14331                 if ((svar = vstate->dtvs_locals[i]) == NULL)
14332                         continue;
14333
14334                 ASSERT(svar->dtsv_size >= NCPU * sizeof (uint64_t));
14335                 ent->dtht_locals[i] =
14336                     ((uint64_t *)(uintptr_t)svar->dtsv_data)[CPU->cpu_id];
14337         }
14338 }
14339
14340 static uint64_t
14341 dtrace_helper(int which, dtrace_mstate_t *mstate,
14342     dtrace_state_t *state, uint64_t arg0, uint64_t arg1)
14343 {
14344         uint16_t *flags = &cpu_core[CPU->cpu_id].cpuc_dtrace_flags;
14345         uint64_t sarg0 = mstate->dtms_arg[0];
14346         uint64_t sarg1 = mstate->dtms_arg[1];
14347         uint64_t rval;
14348         dtrace_helpers_t *helpers = curproc->p_dtrace_helpers;
14349         dtrace_helper_action_t *helper;
14350         dtrace_vstate_t *vstate;
14351         dtrace_difo_t *pred;
14352         int i, trace = dtrace_helptrace_enabled;
14353
14354         ASSERT(which >= 0 && which < DTRACE_NHELPER_ACTIONS);
14355
14356         if (helpers == NULL)
14357                 return (0);
14358
14359         if ((helper = helpers->dthps_actions[which]) == NULL)
14360                 return (0);
14361
14362         vstate = &helpers->dthps_vstate;
14363         mstate->dtms_arg[0] = arg0;
14364         mstate->dtms_arg[1] = arg1;
14365
14366         /*
14367          * Now iterate over each helper.  If its predicate evaluates to 'true',
14368          * we'll call the corresponding actions.  Note that the below calls
14369          * to dtrace_dif_emulate() may set faults in machine state.  This is
14370          * okay:  our caller (the outer dtrace_dif_emulate()) will simply plow
14371          * the stored DIF offset with its own (which is the desired behavior).
14372          * Also, note the calls to dtrace_dif_emulate() may allocate scratch
14373          * from machine state; this is okay, too.
14374          */
14375         for (; helper != NULL; helper = helper->dtha_next) {
14376                 if ((pred = helper->dtha_predicate) != NULL) {
14377                         if (trace)
14378                                 dtrace_helper_trace(helper, mstate, vstate, 0);
14379
14380                         if (!dtrace_dif_emulate(pred, mstate, vstate, state))
14381                                 goto next;
```

```
14383                         if (*flags & CPU_DTRACE_FAULT)
14384                                 goto err;
14385                 }

14387                 for (i = 0; i < helper->dtha_nactions; i++) {
14388                         if (trace)
14389                                 dtrace_helper_trace(helper,
14390                                     mstate, vstate, i + 1);

14392                         rval = dtrace_dif_emulate(helper->dtha_actions[i],
14393                             mstate, vstate, state);

14395                         if (*flags & CPU_DTRACE_FAULT)
14396                                 goto err;
14397                 }

14399 next:
14400                 if (trace)
14401                         dtrace_helper_trace(helper, mstate, vstate,
14402                             DTRACE_HELPTRACE_NEXT);
14403         }

14405         if (trace)
14406                 dtrace_helper_trace(helper, mstate, vstate,
14407                     DTRACE_HELPTRACE_DONE);

14409         /*
14410          * Restore the arg0 that we saved upon entry.
14411          */
14412         mstate->dtms_arg[0] = sarg0;
14413         mstate->dtms_arg[1] = sarg1;

14415         return (rval);

14417 err:
14418         if (trace)
14419                 dtrace_helper_trace(helper, mstate, vstate,
14420                     DTRACE_HELPTRACE_ERR);

14422         /*
14423          * Restore the arg0 that we saved upon entry.
14424          */
14425         mstate->dtms_arg[0] = sarg0;
14426         mstate->dtms_arg[1] = sarg1;

14428         return (NULL);
14429 }

14431 static void
14432 dtrace_helper_action_destroy(dtrace_helper_action_t *helper,
14433     dtrace_vstate_t *vstate)
14434 {
14435         int i;

14437         if (helper->dtha_predicate != NULL)
14438                 dtrace_difo_release(helper->dtha_predicate, vstate);

14440         for (i = 0; i < helper->dtha_nactions; i++) {
14441                 ASSERT(helper->dtha_actions[i] != NULL);
14442                 dtrace_difo_release(helper->dtha_actions[i], vstate);
14443         }

14445         kmem_free(helper->dtha_actions,
14446             helper->dtha_nactions * sizeof (dtrace_difo_t *));
14447         kmem_free(helper, sizeof (dtrace_helper_action_t));
14448 }
```



```
14450 static int
14451 dtrace_helper_destroygen(int gen)
14452 {
14453         proc_t *p = curproc;
14454         dtrace_helpers_t *help = p->p_dtrace_helpers;
14455         dtrace_vstate_t *vstate;
14456         int i;

14458         ASSERT(MUTEX_HELD(&dtrace_lock));

14460         if (help == NULL || gen > help->dthps_generation)
14461                 return (EINVAL);

14463         vstate = &help->dthps_vstate;

14465         for (i = 0; i < DTRACE_NHELPER_ACTIONS; i++) {
14466                 dtrace_helper_action_t *last = NULL, *h, *next;

14468                 for (h = help->dthps_actions[i]; h != NULL; h = next) {
14469                         next = h->dtha_next;

14471                         if (h->dtha_generation == gen) {
14472                                 if (last != NULL) {
14473                                         last->dtha_next = next;
14474                                 } else {
14475                                         help->dthps_actions[i] = next;
14476                                 }

14478                                 dtrace_helper_action_destroy(h, vstate);
14479                         } else {
14480                                 last = h;
14481                         }
14482                 }
14483         }

14485         /*
14486          * Interate until we've cleared out all helper providers with the
14487          * given generation number.
14488          */
14489         for (;;) {
14490                 dtrace_helper_provider_t *prov;

14492                 /*
14493                  * Look for a helper provider with the right generation. We
14494                  * have to start back at the beginning of the list each time
14495                  * because we drop dtrace_lock. It's unlikely that we'll make
14496                  * more than two passes.
14497                  */
14498                 for (i = 0; i < help->dthps_nprovs; i++) {
14499                         prov = help->dthps_provs[i];

14501                         if (prov->dthp_generation == gen)
14502                                 break;
14503                 }

14505                 /*
14506                  * If there were no matches, we're done.
14507                  */
14508                 if (i == help->dthps_nprovs)
14509                         break;

14511                 /*
14512                  * Move the last helper provider into this slot.
14513                  */
14514                 help->dthps_nprovs--;
```

```
14515                 help->dthps_provs[i] = help->dthps_provs[help->dthps_nprovs];
14516                 help->dthps_provs[help->dthps_nprovs] = NULL;

14518                 mutex_exit(&dtrace_lock);

14520                 /*
14521                  * If we have a meta provider, remove this helper provider.
14522                  */
14523                 mutex_enter(&dtrace_meta_lock);
14524                 if (dtrace_meta_pid != NULL) {
14525                         ASSERT(dtrace_deferred_pid == NULL);
14526                         dtrace_helper_provider_remove(&prov->dthp_prov,
14527                             p->p_pid);
14528                 }
14529                 mutex_exit(&dtrace_meta_lock);

14531                 dtrace_helper_provider_destroy(prov);

14533                 mutex_enter(&dtrace_lock);
14534         }

14536         return (0);
14537 }

14539 static int
14540 dtrace_helper_validate(dtrace_helper_action_t *helper)
14541 {
14542         int err = 0, i;
14543         dtrace_difo_t *dp;

14545         if ((dp = helper->dtha_predicate) != NULL)
14546                 err += dtrace_difo_validate_helper(dp);

14548         for (i = 0; i < helper->dtha_nactions; i++)
14549                 err += dtrace_difo_validate_helper(helper->dtha_actions[i]);

14551         return (err == 0);
14552 }

14554 static int
14555 dtrace_helper_action_add(int which, dtrace_ecbdesc_t *ep)
14556 {
14557         dtrace_helpers_t *help;
14558         dtrace_helper_action_t *helper, *last;
14559         dtrace_actdesc_t *act;
14560         dtrace_vstate_t *vstate;
14561         dtrace_predicate_t *pred;
14562         int count = 0, nactions = 0, i;

14564         if (which < 0 || which >= DTRACE_NHELPER_ACTIONS)
14565                 return (EINVAL);

14567         help = curproc->p_dtrace_helpers;
14568         last = help->dthps_actions[which];
14569         vstate = &help->dthps_vstate;

14571         for (count = 0; last != NULL; last = last->dtha_next) {
14572                 count++;
14573                 if (last->dtha_next == NULL)
14574                         break;
14575         }

14577         /*
14578          * If we already have dtrace_helper_actions_max helper actions for this
14579          * helper action type, we'll refuse to add a new one.
14580          */
```



```
14581         if (count >= dtrace_helper_actions_max)
14582                 return (ENOSPC);

14584         helper = kmem_zalloc(sizeof (dtrace_helper_action_t), KM_SLEEP);
14585         helper->dtha_generation = help->dthps_generation;

14587         if ((pred = ep->dted_pred.dtpdd_predicate) != NULL) {
14588                 ASSERT(pred->dtp_difo != NULL);
14589                 dtrace_difo_hold(pred->dtp_difo);
14590                 helper->dtha_predicate = pred->dtp_difo;
14591         }

14593         for (act = ep->dted_action; act != NULL; act = act->dtad_next) {
14594                 if (act->dtad_kind != DTRACEACT_DIFEXPR)
14595                         goto err;

14597                 if (act->dtad_difo == NULL)
14598                         goto err;

14600                 nactions++;
14601         }

14603         helper->dtha_actions = kmem_zalloc(sizeof (dtrace_difo_t *) *
14604             (helper->dtha_nactions = nactions), KM_SLEEP);

14606         for (act = ep->dted_action, i = 0; act != NULL; act = act->dtad_next) {
14607                 dtrace_difo_hold(act->dtad_difo);
14608                 helper->dtha_actions[i++] = act->dtad_difo;
14609         }

14611         if (!dtrace_helper_validate(helper))
14612                 goto err;

14614         if (last == NULL) {
14615                 help->dthps_actions[which] = helper;
14616         } else {
14617                 last->dtha_next = helper;
14618         }

14620         if (vstate->dtvs_nlocals > dtrace_helptrace_nlocals) {
14621                 dtrace_helptrace_nlocals = vstate->dtvs_nlocals;
14622                 dtrace_helptrace_next = 0;
14623         }

14625         return (0);
14626 err:
14627         dtrace_helper_action_destroy(helper, vstate);
14628         return (EINVAL);
14629 }

14631 static void
14632 dtrace_helper_provider_register(proc_t *p, dtrace_helpers_t *help,
14633     dof_helper_t *dofhp)
14634 {
14635         ASSERT(MUTEX_NOT_HELD(&dtrace_lock));

14637         mutex_enter(&dtrace_meta_lock);
14638         mutex_enter(&dtrace_lock);

14640         if (!dtrace_attached() || dtrace_meta_pid == NULL) {
14641                 /*
14642                  * If the dtrace module is loaded but not attached, or if
14643                  * there aren't isn't a meta provider registered to deal with
14644                  * these provider descriptions, we need to postpone creating
14645                  * the actual providers until later.
14646                  */
```

```
14648                 if (help->dthps_next == NULL && help->dthps_prev == NULL &&
14649                     dtrace_deferred_pid != help) {
14650                         help->dthps_deferred = 1;
14651                         help->dthps_pid = p->p_pid;
14652                         help->dthps_next = dtrace_deferred_pid;
14653                         help->dthps_prev = NULL;
14654                         if (dtrace_deferred_pid != NULL)
14655                                 dtrace_deferred_pid->dthps_prev = help;
14656                         dtrace_deferred_pid = help;
14657                 }

14659                 mutex_exit(&dtrace_lock);

14661         } else if (dofhp != NULL) {
14662                 /*
14663                  * If the dtrace module is loaded and we have a particular
14664                  * helper provider description, pass that off to the
14665                  * meta provider.
14666                  */

14668                 mutex_exit(&dtrace_lock);

14670                 dtrace_helper_provide(dofhp, p->p_pid);

14672         } else {
14673                 /*
14674                  * Otherwise, just pass all the helper provider descriptions
14675                  * off to the meta provider.
14676                  */

14678                 int i;
14679                 mutex_exit(&dtrace_lock);

14681                 for (i = 0; i < help->dthps_nprovs; i++) {
14682                         dtrace_helper_provide(&help->dthps_provs[i]->dthp_prov,
14683                             p->p_pid);
14684                 }
14685         }

14687         mutex_exit(&dtrace_meta_lock);
14688 }

14690 static int
14691 dtrace_helper_provider_add(dof_helper_t *dofhp, int gen)
14692 {
14693         dtrace_helpers_t *help;
14694         dtrace_helper_provider_t *hprov, **tmp_provs;
14695         uint_t tmp_maxprovs, i;

14697         ASSERT(MUTEX_HELD(&dtrace_lock));

14699         help = curproc->p_dtrace_helpers;
14700         ASSERT(help != NULL);

14702         /*
14703          * If we already have dtrace_helper_providers_max helper providers,
14704          * we're refuse to add a new one.
14705          */
14706         if (help->dthps_nprovs >= dtrace_helper_providers_max)
14707                 return (ENOSPC);

14709         /*
14710          * Check to make sure this isn't a duplicate.
14711          */
14712         for (i = 0; i < help->dthps_nprovs; i++) {
```



```
14713                 if (dofhp->dofhp_dof ==
14714                     help->dthps_provs[i]->dthp_prov.dofhp_dof)
14715                         return (EALREADY);
14716         }

14718         hprov = kmem_zalloc(sizeof (dtrace_helper_provider_t), KM_SLEEP);
14719         hprov->dthp_prov = *dofhp;
14720         hprov->dthp_ref = 1;
14721         hprov->dthp_generation = gen;

14723         /*
14724          * Allocate a bigger table for helper providers if it's already full.
14725          */
14726         if (help->dthps_maxprovs == help->dthps_nprovs) {
14727                 tmp_maxprovs = help->dthps_maxprovs;
14728                 tmp_provs = help->dthps_provs;

14730                 if (help->dthps_maxprovs == 0)
14731                         help->dthps_maxprovs = 2;
14732                 else
14733                         help->dthps_maxprovs *= 2;
14734                 if (help->dthps_maxprovs > dtrace_helper_providers_max)
14735                         help->dthps_maxprovs = dtrace_helper_providers_max;

14737                 ASSERT(tmp_maxprovs < help->dthps_maxprovs);

14739                 help->dthps_provs = kmem_zalloc(help->dthps_maxprovs *
14740                     sizeof (dtrace_helper_provider_t *), KM_SLEEP);

14742                 if (tmp_provs != NULL) {
14743                         bcopy(tmp_provs, help->dthps_provs, tmp_maxprovs *
14744                             sizeof (dtrace_helper_provider_t *));
14745                         kmem_free(tmp_provs, tmp_maxprovs *
14746                             sizeof (dtrace_helper_provider_t *));
14747                 }
14748         }

14750         help->dthps_provs[help->dthps_nprovs] = hprov;
14751         help->dthps_nprovs++;

14753         return (0);
14754 }

14756 static void
14757 dtrace_helper_provider_destroy(dtrace_helper_provider_t *hprov)
14758 {
14759         mutex_enter(&dtrace_lock);

14761         if (--hprov->dthp_ref == 0) {
14762                 dof_hdr_t *dof;
14763                 mutex_exit(&dtrace_lock);
14764                 dof = (dof_hdr_t *)(uintptr_t)hprov->dthp_prov.dofhp_dof;
14765                 dtrace_dof_destroy(dof);
14766                 kmem_free(hprov, sizeof (dtrace_helper_provider_t));
14767         } else {
14768                 mutex_exit(&dtrace_lock);
14769         }
14770 }

14772 static int
14773 dtrace_helper_provider_validate(dof_hdr_t *dof, dof_sec_t *sec)
14774 {
14775         uintptr_t daddr = (uintptr_t)dof;
14776         dof_sec_t *str_sec, *prb_sec, *arg_sec, *off_sec, *enoff_sec;
14777         dof_provider_t *provider;
14778         dof_probe_t *probe;
```

```
14779         uint8_t *arg;
14780         char *strtab, *typestr;
14781         dof_stridx_t typeidx;
14782         size_t typesz;
14783         uint_t nprobes, j, k;

14785         ASSERT(sec->dofs_type == DOF_SECT_PROVIDER);

14787         if (sec->dofs_offset & (sizeof (uint_t) - 1)) {
14788                 dtrace_dof_error(dof, "misaligned section offset");
14789                 return (-1);
14790         }

14792         /*
14793          * The section needs to be large enough to contain the DOF provider
14794          * structure appropriate for the given version.
14795          */
14796         if (sec->dofs_size <
14797             ((dof->dofh_ident[DOF_ID_VERSION] == DOF_VERSION_1) ?
14798             offsetof(dof_provider_t, dofpv_prenoffs) :
14799             sizeof (dof_provider_t))) {
14800                 dtrace_dof_error(dof, "provider section too small");
14801                 return (-1);
14802         }

14804         provider = (dof_provider_t *)(uintptr_t)(daddr + sec->dofs_offset);
14805         str_sec = dtrace_dof_sect(dof, DOF_SECT_STRTAB, provider->dofpv_strtab);
14806         prb_sec = dtrace_dof_sect(dof, DOF_SECT_PROBES, provider->dofpv_probes);
14807         arg_sec = dtrace_dof_sect(dof, DOF_SECT_PRARGS, provider->dofpv_prargs);
14808         off_sec = dtrace_dof_sect(dof, DOF_SECT_PROFFS, provider->dofpv_proffs);

14810         if (str_sec == NULL || prb_sec == NULL ||
14811             arg_sec == NULL || off_sec == NULL)
14812                 return (-1);

14814         enoff_sec = NULL;

14816         if (dof->dofh_ident[DOF_ID_VERSION] != DOF_VERSION_1 &&
14817             provider->dofpv_prenoffs != DOF_SECT_NONE &&
14818             (enoff_sec = dtrace_dof_sect(dof, DOF_SECT_PRENOFFS,
14819             provider->dofpv_prenoffs)) == NULL)
14820                 return (-1);

14822         strtab = (char *)(uintptr_t)(daddr + str_sec->dofs_offset);

14824         if (provider->dofpv_name >= str_sec->dofs_size ||
14825             strlen(strtab + provider->dofpv_name) >= DTRACE_PROVNAMELEN) {
14826                 dtrace_dof_error(dof, "invalid provider name");
14827                 return (-1);
14828         }

14830         if (prb_sec->dofs_entsize == 0 ||
14831             prb_sec->dofs_entsize > prb_sec->dofs_size) {
14832                 dtrace_dof_error(dof, "invalid entry size");
14833                 return (-1);
14834         }

14836         if (prb_sec->dofs_entsize & (sizeof (uintptr_t) - 1)) {
14837                 dtrace_dof_error(dof, "misaligned entry size");
14838                 return (-1);
14839         }

14841         if (off_sec->dofs_entsize != sizeof (uint32_t)) {
14842                 dtrace_dof_error(dof, "invalid entry size");
14843                 return (-1);
14844         }
```



```
14846         if (off_sec->dofs_offset & (sizeof (uint32_t) - 1)) {
14847                 dtrace_dof_error(dof, "misaligned section offset");
14848                 return (-1);
14849         }

14851         if (arg_sec->dofs_entsize != sizeof (uint8_t)) {
14852                 dtrace_dof_error(dof, "invalid entry size");
14853                 return (-1);
14854         }

14856         arg = (uint8_t *)(uintptr_t)(daddr + arg_sec->dofs_offset);

14858         nprobes = prb_sec->dofs_size / prb_sec->dofs_entsize;

14860         /*
14861          * Take a pass through the probes to check for errors.
14862          */
14863         for (j = 0; j < nprobes; j++) {
14864                 probe = (dof_probe_t *)(uintptr_t)(daddr +
14865                     prb_sec->dofs_offset + j * prb_sec->dofs_entsize);

14867                 if (probe->dofpr_func >= str_sec->dofs_size) {
14868                         dtrace_dof_error(dof, "invalid function name");
14869                         return (-1);
14870                 }

14872                 if (strlen(strtab + probe->dofpr_func) >= DTRACE_FUNCNAMELEN) {
14873                         dtrace_dof_error(dof, "function name too long");
14874                         return (-1);
14875                 }

14877                 if (probe->dofpr_name >= str_sec->dofs_size ||
14878                     strlen(strtab + probe->dofpr_name) >= DTRACE_NAMELEN) {
14879                         dtrace_dof_error(dof, "invalid probe name");
14880                         return (-1);
14881                 }

14883                 /*
14884                  * The offset count must not wrap the index, and the offsets
14885                  * must also not overflow the section's data.
14886                  */
14887                 if (probe->dofpr_offidx + probe->dofpr_noffs <
14888                     probe->dofpr_offidx ||
14889                     (probe->dofpr_offidx + probe->dofpr_noffs) *
14890                     off_sec->dofs_entsize > off_sec->dofs_size) {
14891                         dtrace_dof_error(dof, "invalid probe offset");
14892                         return (-1);
14893                 }

14895                 if (dof->dofh_ident[DOF_ID_VERSION] != DOF_VERSION_1) {
14896                         /*
14897                          * If there's no is-enabled offset section, make sure
14898                          * there aren't any is-enabled offsets. Otherwise
14899                          * perform the same checks as for probe offsets
14900                          * (immediately above).
14901                          */
14902                         if (enoff_sec == NULL) {
14903                                 if (probe->dofpr_enoffidx != 0 ||
14904                                     probe->dofpr_nenoffs != 0) {
14905                                         dtrace_dof_error(dof, "is-enabled "
14906                                             "offsets with null section");
14907                                         return (-1);
14908                                 }
14909                         } else if (probe->dofpr_enoffidx +
14910                             probe->dofpr_nenoffs < probe->dofpr_enoffidx ||
```

```
14911                           (probe->dofpr_enoffidx + probe->dofpr_nenoffs) *
14912                           enoff_sec->dofs_entsize > enoff_sec->dofs_size) {
14913                               dtrace_dof_error(dof, "invalid is-enabled "
14914                                   "offset");
14915                               return (-1);
14916                       }

14918                       if (probe->dofpr_noffs + probe->dofpr_nenoffs == 0) {
14919                               dtrace_dof_error(dof, "zero probe and "
14920                                   "is-enabled offsets");
14921                               return (-1);
14922                       }
14923               } else if (probe->dofpr_noffs == 0) {
14924                       dtrace_dof_error(dof, "zero probe offsets");
14925                       return (-1);
14926               }

14928               if (probe->dofpr_argidx + probe->dofpr_xargc <
14929                   probe->dofpr_argidx ||
14930                   (probe->dofpr_argidx + probe->dofpr_xargc) *
14931                   arg_sec->dofs_entsize > arg_sec->dofs_size) {
14932                       dtrace_dof_error(dof, "invalid args");
14933                       return (-1);
14934               }

14936               typeidx = probe->dofpr_nargv;
14937               typestr = strtab + probe->dofpr_nargv;
14938               for (k = 0; k < probe->dofpr_nargc; k++) {
14939                       if (typeidx >= str_sec->dofs_size) {
14940                               dtrace_dof_error(dof, "bad "
14941                                   "native argument type");
14942                               return (-1);
14943                       }

14945                       typesz = strlen(typestr) + 1;
14946                       if (typesz > DTRACE_ARGTYPELEN) {
14947                               dtrace_dof_error(dof, "native "
14948                                   "argument type too long");
14949                               return (-1);
14950                       }
14951                       typeidx += typesz;
14952                       typestr += typesz;
14953               }

14955               typeidx = probe->dofpr_xargv;
14956               typestr = strtab + probe->dofpr_xargv;
14957               for (k = 0; k < probe->dofpr_xargc; k++) {
14958                       if (arg[probe->dofpr_argidx + k] > probe->dofpr_nargc) {
14959                               dtrace_dof_error(dof, "bad "
14960                                   "native argument index");
14961                               return (-1);
14962                       }

14964                       if (typeidx >= str_sec->dofs_size) {
14965                               dtrace_dof_error(dof, "bad "
14966                                   "translated argument type");
14967                               return (-1);
14968                       }

14970                       typesz = strlen(typestr) + 1;
14971                       if (typesz > DTRACE_ARGTYPELEN) {
14972                               dtrace_dof_error(dof, "translated argument "
14973                                   "type too long");
14974                               return (-1);
14975                       }
```



```
14977                           typeidx += typesz;
14978                           typestr += typesz;
14979                   }
14980           }

14982           return (0);
14983   }

14985   static int
14986   dtrace_helper_slurp(dof_hdr_t *dof, dof_helper_t *dhp)
14987   {
14988           dtrace_helpers_t *help;
14989           dtrace_vstate_t *vstate;
14990           dtrace_enabling_t *enab = NULL;
14991           int i, gen, rv, nhelpers = 0, nprovs = 0, destroy = 1;
14992           uintptr_t daddr = (uintptr_t)dof;

14994           ASSERT(MUTEX_HELD(&dtrace_lock));

14996           if ((help = curproc->p_dtrace_helpers) == NULL)
14997                   help = dtrace_helpers_create(curproc);

14999           vstate = &help->dthps_vstate;

15001           if ((rv = dtrace_dof_slurp(dof, vstate, NULL, &enab,
15002               dhp != NULL ? dhp->dofhp_addr : 0, B_FALSE)) != 0) {
15003                   dtrace_dof_destroy(dof);
15004                   return (rv);
15005           }

15007           /*
15008            * Look for helper providers and validate their descriptions.
15009            */
15010           if (dhp != NULL) {
15011                   for (i = 0; i < dof->dofh_secnum; i++) {
15012                           dof_sec_t *sec = (dof_sec_t *)(uintptr_t)(daddr +
15013                               dof->dofh_secoff + i * dof->dofh_secsize);

15015                           if (sec->dofs_type != DOF_SECT_PROVIDER)
15016                                   continue;

15018                           if (dtrace_helper_provider_validate(dof, sec) != 0) {
15019                                   dtrace_enabling_destroy(enab);
15020                                   dtrace_dof_destroy(dof);
15021                                   return (-1);
15022                           }

15024                           nprovs++;
15025                   }
15026           }

15028           /*
15029            * Now we need to walk through the ECB descriptions in the enabling.
15030            */
15031           for (i = 0; i < enab->dten_ndesc; i++) {
15032                   dtrace_ecbdesc_t *ep = enab->dten_desc[i];
15033                   dtrace_probedesc_t *desc = &ep->dted_probe;

15035                   if (strcmp(desc->dtpd_provider, "dtrace") != 0)
15036                           continue;

15038                   if (strcmp(desc->dtpd_mod, "helper") != 0)
15039                           continue;

15041                   if (strcmp(desc->dtpd_func, "ustack") != 0)
15042                           continue;
```

```
15044                 if ((rv = dtrace_helper_action_add(DTRACE_HELPER_ACTION_USTACK,
15045                     ep)) != 0) {
15046                         /*
15047                          * Adding this helper action failed -- we are now going
15048                          * to rip out the entire generation and return failure.
15049                          */
15050                         (void) dtrace_helper_destroygen(help->dthps_generation);
15051                         dtrace_enabling_destroy(enab);
15052                         dtrace_dof_destroy(dof);
15053                         return (-1);
15054                 }

15056                 nhelpers++;
15057         }

15059         if (nhelpers < enab->dten_ndesc)
15060                 dtrace_dof_error(dof, "unmatched helpers");

15062         gen = help->dthps_generation++;
15063         dtrace_enabling_destroy(enab);

15065         if (dhp != NULL && nprovs > 0) {
15066                 dhp->dofhp_dof = (uint64_t)(uintptr_t)dof;
15067                 if (dtrace_helper_provider_add(dhp, gen) == 0) {
15068                         mutex_exit(&dtrace_lock);
15069                         dtrace_helper_provider_register(curproc, help, dhp);
15070                         mutex_enter(&dtrace_lock);

15072                         destroy = 0;
15073                 }
15074         }

15076         if (destroy)
15077                 dtrace_dof_destroy(dof);

15079         return (gen);
15080 }

15082 static dtrace_helpers_t *
15083 dtrace_helpers_create(proc_t *p)
15084 {
15085         dtrace_helpers_t *help;

15087         ASSERT(MUTEX_HELD(&dtrace_lock));
15088         ASSERT(p->p_dtrace_helpers == NULL);

15090         help = kmem_zalloc(sizeof (dtrace_helpers_t), KM_SLEEP);
15091         help->dthps_actions = kmem_zalloc(sizeof (dtrace_helper_action_t *) *
15092             DTRACE_NHELPER_ACTIONS, KM_SLEEP);

15094         p->p_dtrace_helpers = help;
15095         dtrace_helpers++;

15097         return (help);
15098 }

15100 static void
15101 dtrace_helpers_destroy(void)
15102 {
15103         dtrace_helpers_t *help;
15104         dtrace_vstate_t *vstate;
15105         proc_t *p = curproc;
15106         int i;

15108         mutex_enter(&dtrace_lock);
```



```
15110         ASSERT(p->p_dtrace_helpers != NULL);
15111         ASSERT(dtrace_helpers > 0);

15113         help = p->p_dtrace_helpers;
15114         vstate = &help->dthps_vstate;

15116         /*
15117          * We're now going to lose the help from this process.
15118          */
15119         p->p_dtrace_helpers = NULL;
15120         dtrace_sync();

15122         /*
15123          * Destory the helper actions.
15124          */
15125         for (i = 0; i < DTRACE_NHELPER_ACTIONS; i++) {
15126                 dtrace_helper_action_t *h, *next;

15128                 for (h = help->dthps_actions[i]; h != NULL; h = next) {
15129                         next = h->dtha_next;
15130                         dtrace_helper_action_destroy(h, vstate);
15131                         h = next;
15132                 }
15133         }

15135         mutex_exit(&dtrace_lock);

15137         /*
15138          * Destroy the helper providers.
15139          */
15140         if (help->dthps_maxprovs > 0) {
15141                 mutex_enter(&dtrace_meta_lock);
15142                 if (dtrace_meta_pid != NULL) {
15143                         ASSERT(dtrace_deferred_pid == NULL);

15145                         for (i = 0; i < help->dthps_nprovs; i++) {
15146                                 dtrace_helper_provider_remove(
15147                                     &help->dthps_provs[i]->dthp_prov, p->p_pid);
15148                         }
15149                 } else {
15150                         mutex_enter(&dtrace_lock);
15151                         ASSERT(help->dthps_deferred == 0 ||
15152                             help->dthps_next != NULL ||
15153                             help->dthps_prev != NULL ||
15154                             help == dtrace_deferred_pid);

15156                         /*
15157                          * Remove the helper from the deferred list.
15158                          */
15159                         if (help->dthps_next != NULL)
15160                                 help->dthps_next->dthps_prev = help->dthps_prev;
15161                         if (help->dthps_prev != NULL)
15162                                 help->dthps_prev->dthps_next = help->dthps_next;
15163                         if (dtrace_deferred_pid == help) {
15164                                 dtrace_deferred_pid = help->dthps_next;
15165                                 ASSERT(help->dthps_prev == NULL);
15166                         }

15168                         mutex_exit(&dtrace_lock);
15169                 }

15171                 mutex_exit(&dtrace_meta_lock);

15173                 for (i = 0; i < help->dthps_nprovs; i++) {
15174                         dtrace_helper_provider_destroy(help->dthps_provs[i]);
```

```
15175                 }
15176
15177                 kmem_free(help->dthps_provs, help->dthps_maxprovs *
15178                     sizeof (dtrace_helper_provider_t *));
15179         }
15180
15181         mutex_enter(&dtrace_lock);
15182
15183         dtrace_vstate_fini(&help->dthps_vstate);
15184         kmem_free(help->dthps_actions,
15185             sizeof (dtrace_helper_action_t *) * DTRACE_NHELPER_ACTIONS);
15186         kmem_free(help, sizeof (dtrace_helpers_t));
15187
15188         --dtrace_helpers;
15189         mutex_exit(&dtrace_lock);
15190 }
15191
15192 static void
15193 dtrace_helpers_duplicate(proc_t *from, proc_t *to)
15194 {
15195         dtrace_helpers_t *help, *newhelp;
15196         dtrace_helper_action_t *helper, *new, *last;
15197         dtrace_difo_t *dp;
15198         dtrace_vstate_t *vstate;
15199         int i, j, sz, hasprovs = 0;
15200
15201         mutex_enter(&dtrace_lock);
15202         ASSERT(from->p_dtrace_helpers != NULL);
15203         ASSERT(dtrace_helpers > 0);
15204
15205         help = from->p_dtrace_helpers;
15206         newhelp = dtrace_helpers_create(to);
15207         ASSERT(to->p_dtrace_helpers != NULL);
15208
15209         newhelp->dthps_generation = help->dthps_generation;
15210         vstate = &newhelp->dthps_vstate;
15211
15212         /*
15213          * Duplicate the helper actions.
15214          */
15215         for (i = 0; i < DTRACE_NHELPER_ACTIONS; i++) {
15216                 if ((helper = help->dthps_actions[i]) == NULL)
15217                         continue;
15218
15219                 for (last = NULL; helper != NULL; helper = helper->dtha_next) {
15220                         new = kmem_zalloc(sizeof (dtrace_helper_action_t),
15221                             KM_SLEEP);
15222                         new->dtha_generation = helper->dtha_generation;
15223
15224                         if ((dp = helper->dtha_predicate) != NULL) {
15225                                 dp = dtrace_difo_duplicate(dp, vstate);
15226                                 new->dtha_predicate = dp;
15227                         }
15228
15229                         new->dtha_nactions = helper->dtha_nactions;
15230                         sz = sizeof (dtrace_difo_t *) * new->dtha_nactions;
15231                         new->dtha_actions = kmem_alloc(sz, KM_SLEEP);
15232
15233                         for (j = 0; j < new->dtha_nactions; j++) {
15234                                 dtrace_difo_t *dp = helper->dtha_actions[j];
15235
15236                                 ASSERT(dp != NULL);
15237                                 dp = dtrace_difo_duplicate(dp, vstate);
15238                                 new->dtha_actions[j] = dp;
15239                         }
```



```
15241                         if (last != NULL) {
15242                                 last->dtha_next = new;
15243                         } else {
15244                                 newhelp->dthps_actions[i] = new;
15245                         }
15246
15247                         last = new;
15248                 }
15249         }
15250
15251         /*
15252          * Duplicate the helper providers and register them with the
15253          * DTrace framework.
15254          */
15255         if (help->dthps_nprovs > 0) {
15256                 newhelp->dthps_nprovs = help->dthps_nprovs;
15257                 newhelp->dthps_maxprovs = help->dthps_nprovs;
15258                 newhelp->dthps_provs = kmem_alloc(newhelp->dthps_nprovs *
15259                     sizeof (dtrace_helper_provider_t *), KM_SLEEP);
15260                 for (i = 0; i < newhelp->dthps_nprovs; i++) {
15261                         newhelp->dthps_provs[i] = help->dthps_provs[i];
15262                         newhelp->dthps_provs[i]->dthp_ref++;
15263                 }
15264
15265                 hasprovs = 1;
15266         }
15267
15268         mutex_exit(&dtrace_lock);
15269
15270         if (hasprovs)
15271                 dtrace_helper_provider_register(to, newhelp, NULL);
15272 }
15273
15274 /*
15275  * DTrace Hook Functions
15276  */
15277 static void
15278 dtrace_module_loaded(struct modctl *ctl)
15279 {
15280         dtrace_provider_t *prv;
15281
15282         mutex_enter(&dtrace_provider_lock);
15283         mutex_enter(&mod_lock);
15284
15285         ASSERT(ctl->mod_busy);
15286
15287         /*
15288          * We're going to call each providers per-module provide operation
15289          * specifying only this module.
15290          */
15291         for (prv = dtrace_provider; prv != NULL; prv = prv->dtpv_next)
15292                 prv->dtpv_pops.dtps_provide_module(prv->dtpv_arg, ctl);
15293
15294         mutex_exit(&mod_lock);
15295         mutex_exit(&dtrace_provider_lock);
15296
15297         /*
15298          * If we have any retained enablings, we need to match against them.
15299          * Enabling probes requires that cpu_lock be held, and we cannot hold
15300          * cpu_lock here -- it is legal for cpu_lock to be held when loading a
15301          * module.  (In particular, this happens when loading scheduling
15302          * classes.)  So if we have any retained enablings, we need to dispatch
15303          * our task queue to do the match for us.
15304          */
15305         mutex_enter(&dtrace_lock);
```

```
15307         if (dtrace_retained == NULL) {
15308                 mutex_exit(&dtrace_lock);
15309                 return;
15310         }

15312         (void) taskq_dispatch(dtrace_taskq,
15313             (task_func_t *)dtrace_enabling_matchall, NULL, TQ_SLEEP);

15315         mutex_exit(&dtrace_lock);

15317         /*
15318          * And now, for a little heuristic sleaze:  in general, we want to
15319          * match modules as soon as they load.  However, we cannot guarantee
15320          * this, because it would lead us to the lock ordering violation
15321          * outlined above.  The common case, of course, is that cpu_lock is
15322          * _not_ held -- so we delay here for a clock tick, hoping that that's
15323          * long enough for the task queue to do its work.  If it's not, it's
15324          * not a serious problem -- it just means that the module that we
15325          * just loaded may not be immediately instrumentable.
15326          */
15327         delay(1);
15328 }

15330 static void
15331 dtrace_module_unloaded(struct modctl *ctl)
15332 {
15333         dtrace_probe_t template, *probe, *first, *next;
15334         dtrace_provider_t *prov;

15336         template.dtpr_mod = ctl->mod_modname;

15338         mutex_enter(&dtrace_provider_lock);
15339         mutex_enter(&mod_lock);
15340         mutex_enter(&dtrace_lock);

15342         if (dtrace_bymod == NULL) {
15343                 /*
15344                  * The DTrace module is loaded (obviously) but not attached;
15345                  * we don't have any work to do.
15346                  */
15347                 mutex_exit(&dtrace_provider_lock);
15348                 mutex_exit(&mod_lock);
15349                 mutex_exit(&dtrace_lock);
15350                 return;
15351         }

15353         for (probe = first = dtrace_hash_lookup(dtrace_bymod, &template);
15354             probe != NULL; probe = probe->dtpr_nextmod) {
15355                 if (probe->dtpr_ecb != NULL) {
15356                         mutex_exit(&dtrace_provider_lock);
15357                         mutex_exit(&mod_lock);
15358                         mutex_exit(&dtrace_lock);

15360                         /*
15361                          * This shouldn't _actually_ be possible -- we're
15362                          * unloading a module that has an enabled probe in it.
15363                          * (It's normally up to the provider to make sure that
15364                          * this can't happen.)  However, because dtps_enable()
15365                          * doesn't have a failure mode, there can be an
15366                          * enable/unload race.  Upshot:  we don't want to
15367                          * assert, but we're not going to disable the
15368                          * probe, either.
15369                          */
15370                         if (dtrace_err_verbose) {
15371                                 cmn_err(CE_WARN, "unloaded module '%s' had "
15372                                     "enabled probes", ctl->mod_modname);
```



```
15373                         }

15375                         return;
15376                 }
15377         }

15379         probe = first;

15381         for (first = NULL; probe != NULL; probe = next) {
15382                 ASSERT(dtrace_probes[probe->dtpr_id - 1] == probe);

15384                 dtrace_probes[probe->dtpr_id - 1] = NULL;

15386                 next = probe->dtpr_nextmod;
15387                 dtrace_hash_remove(dtrace_bymod, probe);
15388                 dtrace_hash_remove(dtrace_byfunc, probe);
15389                 dtrace_hash_remove(dtrace_byname, probe);

15391                 if (first == NULL) {
15392                         first = probe;
15393                         probe->dtpr_nextmod = NULL;
15394                 } else {
15395                         probe->dtpr_nextmod = first;
15396                         first = probe;
15397                 }
15398         }

15400         /*
15401          * We've removed all of the module's probes from the hash chains and
15402          * from the probe array.  Now issue a dtrace_sync() to be sure that
15403          * everyone has cleared out from any probe array processing.
15404          */
15405         dtrace_sync();

15407         for (probe = first; probe != NULL; probe = first) {
15408                 first = probe->dtpr_nextmod;
15409                 prov = probe->dtpr_provider;
15410                 prov->dtpv_pops.dtps_destroy(prov->dtpv_arg, probe->dtpr_id,
15411                     probe->dtpr_arg);
15412                 kmem_free(probe->dtpr_mod, strlen(probe->dtpr_mod) + 1);
15413                 kmem_free(probe->dtpr_func, strlen(probe->dtpr_func) + 1);
15414                 kmem_free(probe->dtpr_name, strlen(probe->dtpr_name) + 1);
15415                 vmem_free(dtrace_arena, (void *)(uintptr_t)probe->dtpr_id, 1);
15416                 kmem_free(probe, sizeof (dtrace_probe_t));
15417         }

15419         mutex_exit(&dtrace_lock);
15420         mutex_exit(&mod_lock);
15421         mutex_exit(&dtrace_provider_lock);
15422 }

15424 void
15425 dtrace_suspend(void)
15426 {
15427         dtrace_probe_foreach(offsetof(dtrace_pops_t, dtps_suspend));
15428 }

15430 void
15431 dtrace_resume(void)
15432 {
15433         dtrace_probe_foreach(offsetof(dtrace_pops_t, dtps_resume));
15434 }

15436 static int
15437 dtrace_cpu_setup(cpu_setup_t what, processorid_t cpu)
15438 {
```

```
15439         ASSERT(MUTEX_HELD(&cpu_lock));
15440         mutex_enter(&dtrace_lock);

15442         switch (what) {
15443         case CPU_CONFIG: {
15444                 dtrace_state_t *state;
15445                 dtrace_optval_t *opt, rs, c;

15447                 /*
15448                  * For now, we only allocate a new buffer for anonymous state.
15449                  */
15450                 if ((state = dtrace_anon.dta_state) == NULL)
15451                         break;

15453                 if (state->dts_activity != DTRACE_ACTIVITY_ACTIVE)
15454                         break;

15456                 opt = state->dts_options;
15457                 c = opt[DTRACEOPT_CPU];

15459                 if (c != DTRACE_CPUALL && c != DTRACEOPT_UNSET && c != cpu)
15460                         break;

15462                 /*
15463                  * Regardless of what the actual policy is, we're going to
15464                  * temporarily set our resize policy to be manual.  We're
15465                  * also going to temporarily set our CPU option to denote
15466                  * the newly configured CPU.
15467                  */
15468                 rs = opt[DTRACEOPT_BUFRESIZE];
15469                 opt[DTRACEOPT_BUFRESIZE] = DTRACEOPT_BUFRESIZE_MANUAL;
15470                 opt[DTRACEOPT_CPU] = (dtrace_optval_t)cpu;

15472                 (void) dtrace_state_buffers(state);

15474                 opt[DTRACEOPT_BUFRESIZE] = rs;
15475                 opt[DTRACEOPT_CPU] = c;

15477                 break;
15478         }

15480         case CPU_UNCONFIG:
15481                 /*
15482                  * We don't free the buffer in the CPU_UNCONFIG case.  (The
15483                  * buffer will be freed when the consumer exits.)
15484                  */
15485                 break;

15487         default:
15488                 break;
15489         }

15491         mutex_exit(&dtrace_lock);
15492         return (0);
15493 }

15495 static void
15496 dtrace_cpu_setup_initial(processorid_t cpu)
15497 {
15498         (void) dtrace_cpu_setup(CPU_CONFIG, cpu);
15499 }

15501 static void
15502 dtrace_toxrange_add(uintptr_t base, uintptr_t limit)
15503 {
15504         if (dtrace_toxranges >= dtrace_toxranges_max) {
```



```
15505                 int osize, nsize;
15506                 dtrace_toxrange_t *range;

15508                 osize = dtrace_toxranges_max * sizeof (dtrace_toxrange_t);

15510                 if (osize == 0) {
15511                         ASSERT(dtrace_toxrange == NULL);
15512                         ASSERT(dtrace_toxranges_max == 0);
15513                         dtrace_toxranges_max = 1;
15514                 } else {
15515                         dtrace_toxranges_max <<= 1;
15516                 }

15518                 nsize = dtrace_toxranges_max * sizeof (dtrace_toxrange_t);
15519                 range = kmem_zalloc(nsize, KM_SLEEP);

15521                 if (dtrace_toxrange != NULL) {
15522                         ASSERT(osize != 0);
15523                         bcopy(dtrace_toxrange, range, osize);
15524                         kmem_free(dtrace_toxrange, osize);
15525                 }

15527                 dtrace_toxrange = range;
15528         }

15530         ASSERT(dtrace_toxrange[dtrace_toxranges].dtt_base == NULL);
15531         ASSERT(dtrace_toxrange[dtrace_toxranges].dtt_limit == NULL);

15533         dtrace_toxrange[dtrace_toxranges].dtt_base = base;
15534         dtrace_toxrange[dtrace_toxranges].dtt_limit = limit;
15535         dtrace_toxranges++;
15536 }

15538 static void
15539 dtrace_getf_barrier()
15540 {
15541         /*
15542          * When we have unprivileged (that is, non-DTRACE_CRV_KERNEL) enablings
15543          * that contain calls to getf(), this routine will be called on every
15544          * closef() before either the underlying vnode is released or the
15545          * file_t itself is freed.  By the time we are here, it is essential
15546          * that the file_t can no longer be accessed from a call to getf()
15547          * in probe context -- that assures that a dtrace_sync() can be used
15548          * to clear out any enablings referring to the old structures.
15549          */
15550         if (curthread->t_procp->p_zone->zone_dtrace_getf != 0 ||
15551             kcred->cr_zone->zone_dtrace_getf != 0)
15552                 dtrace_sync();
15553 }

15555 /*
15556  * DTrace Driver Cookbook Functions
15557  */
15558 /*ARGSUSED*/
15559 static int
15560 dtrace_attach(dev_info_t *devi, ddi_attach_cmd_t cmd)
15561 {
15562         dtrace_provider_id_t id;
15563         dtrace_state_t *state = NULL;
15564         dtrace_enabling_t *enab;

15566         mutex_enter(&cpu_lock);
15567         mutex_enter(&dtrace_provider_lock);
15568         mutex_enter(&dtrace_lock);

15570         if (ddi_soft_state_init(&dtrace_softstate,
```

```
15571             sizeof (dtrace_state_t), 0) != 0) {
15572                 cmn_err(CE_NOTE, "/dev/dtrace failed to initialize soft state");
15573                 mutex_exit(&cpu_lock);
15574                 mutex_exit(&dtrace_provider_lock);
15575                 mutex_exit(&dtrace_lock);
15576                 return (DDI_FAILURE);
15577         }

15579         if (ddi_create_minor_node(devi, DTRACEMNR_DTRACE, S_IFCHR,
15580             DTRACEMNRN_DTRACE, DDI_PSEUDO, NULL) == DDI_FAILURE ||
15581             ddi_create_minor_node(devi, DTRACEMNR_HELPER, S_IFCHR,
15582             DTRACEMNRN_HELPER, DDI_PSEUDO, NULL) == DDI_FAILURE) {
15583                 cmn_err(CE_NOTE, "/dev/dtrace couldn't create minor nodes");
15584                 ddi_remove_minor_node(devi, NULL);
15585                 ddi_soft_state_fini(&dtrace_softstate);
15586                 mutex_exit(&cpu_lock);
15587                 mutex_exit(&dtrace_provider_lock);
15588                 mutex_exit(&dtrace_lock);
15589                 return (DDI_FAILURE);
15590         }

15592         ddi_report_dev(devi);
15593         dtrace_devi = devi;

15595         dtrace_modload = dtrace_module_loaded;
15596         dtrace_modunload = dtrace_module_unloaded;
15597         dtrace_cpu_init = dtrace_cpu_setup_initial;
15598         dtrace_helpers_cleanup = dtrace_helpers_destroy;
15599         dtrace_helpers_fork = dtrace_helpers_duplicate;
15600         dtrace_cpustart_init = dtrace_suspend;
15601         dtrace_cpustart_fini = dtrace_resume;
15602         dtrace_debugger_init = dtrace_suspend;
15603         dtrace_debugger_fini = dtrace_resume;

15605         register_cpu_setup_func((cpu_setup_func_t *)dtrace_cpu_setup, NULL);

15607         ASSERT(MUTEX_HELD(&cpu_lock));

15609         dtrace_arena = vmem_create("dtrace", (void *)1, UINT32_MAX, 1,
15610             NULL, NULL, NULL, 0, VM_SLEEP | VMC_IDENTIFIER);
15611         dtrace_minor = vmem_create("dtrace_minor", (void *)DTRACEMNRN_CLONE,
15612             UINT32_MAX - DTRACEMNRN_CLONE, 1, NULL, NULL, NULL, 0,
15613             VM_SLEEP | VMC_IDENTIFIER);
15614         dtrace_taskq = taskq_create("dtrace_taskq", 1, maxclsyspri,
15615             1, INT_MAX, 0);

15617         dtrace_state_cache = kmem_cache_create("dtrace_state_cache",
15618             sizeof (dtrace_dstate_percpu_t) * NCPU, DTRACE_STATE_ALIGN,
15619             NULL, NULL, NULL, NULL, NULL, 0);

15621         ASSERT(MUTEX_HELD(&cpu_lock));
15622         dtrace_bymod = dtrace_hash_create(offsetof(dtrace_probe_t, dtpr_mod),
15623             offsetof(dtrace_probe_t, dtpr_nextmod),
15624             offsetof(dtrace_probe_t, dtpr_prevmod));

15626         dtrace_byfunc = dtrace_hash_create(offsetof(dtrace_probe_t, dtpr_func),
15627             offsetof(dtrace_probe_t, dtpr_nextfunc),
15628             offsetof(dtrace_probe_t, dtpr_prevfunc));

15630         dtrace_byname = dtrace_hash_create(offsetof(dtrace_probe_t, dtpr_name),
15631             offsetof(dtrace_probe_t, dtpr_nextname),
15632             offsetof(dtrace_probe_t, dtpr_prevname));

15634         if (dtrace_retain_max < 1) {
15635                 cmn_err(CE_WARN, "illegal value (%lu) for dtrace_retain_max; "
15636                     "setting to 1", dtrace_retain_max);
```



```
15637                 dtrace_retain_max = 1;
15638         }

15640         /*
15641          * Now discover our toxic ranges.
15642          */
15643         dtrace_toxic_ranges(dtrace_toxrange_add);

15645         /*
15646          * Before we register ourselves as a provider to our own framework,
15647          * we would like to assert that dtrace_provider is NULL -- but that's
15648          * not true if we were loaded as a dependency of a DTrace provider.
15649          * Once we've registered, we can assert that dtrace_provider is our
15650          * pseudo provider.
15651          */
15652         (void) dtrace_register("dtrace", &dtrace_provider_attr,
15653             DTRACE_PRIV_NONE, 0, &dtrace_provider_ops, NULL, &id);

15655         ASSERT(dtrace_provider != NULL);
15656         ASSERT((dtrace_provider_id_t)dtrace_provider == id);

15658         dtrace_probeid_begin = dtrace_probe_create((dtrace_provider_id_t)
15659             dtrace_provider, NULL, NULL, "BEGIN", 0, NULL);
15660         dtrace_probeid_end = dtrace_probe_create((dtrace_provider_id_t)
15661             dtrace_provider, NULL, NULL, "END", 0, NULL);
15662         dtrace_probeid_error = dtrace_probe_create((dtrace_provider_id_t)
15663             dtrace_provider, NULL, NULL, "ERROR", 1, NULL);

15665         dtrace_anon_property();
15666         mutex_exit(&cpu_lock);

15668         /*
15669          * If DTrace helper tracing is enabled, we need to allocate the
15670          * trace buffer and initialize the values.
15671          */
15672         if (dtrace_helptrace_enabled) {
15673                 ASSERT(dtrace_helptrace_buffer == NULL);
15674                 dtrace_helptrace_buffer =
15675                     kmem_zalloc(dtrace_helptrace_bufsize, KM_SLEEP);
15676                 dtrace_helptrace_next = 0;
15677         }

15679         /*
15680          * If there are already providers, we must ask them to provide their
15681          * probes, and then match any anonymous enabling against them.  Note
15682          * that there should be no other retained enablings at this time:
15683          * the only retained enablings at this time should be the anonymous
15684          * enabling.
15685          */
15686         if (dtrace_anon.dta_enabling != NULL) {
15687                 ASSERT(dtrace_retained == dtrace_anon.dta_enabling);

15689                 dtrace_enabling_provide(NULL);
15690                 state = dtrace_anon.dta_state;

15692                 /*
15693                  * We couldn't hold cpu_lock across the above call to
15694                  * dtrace_enabling_provide(), but we must hold it to actually
15695                  * enable the probes.  We have to drop all of our locks, pick
15696                  * up cpu_lock, and regain our locks before matching the
15697                  * retained anonymous enabling.
15698                  */
15699                 mutex_exit(&dtrace_lock);
15700                 mutex_exit(&dtrace_provider_lock);

15702                 mutex_enter(&cpu_lock);
```

```
15703                 mutex_enter(&dtrace_provider_lock);
15704                 mutex_enter(&dtrace_lock);

15706                 if ((enab = dtrace_anon.dta_enabling) != NULL)
15707                         (void) dtrace_enabling_match(enab, NULL);

15709                 mutex_exit(&cpu_lock);
15710         }

15712         mutex_exit(&dtrace_lock);
15713         mutex_exit(&dtrace_provider_lock);

15715         if (state != NULL) {
15716                 /*
15717                  * If we created any anonymous state, set it going now.
15718                  */
15719                 (void) dtrace_state_go(state, &dtrace_anon.dta_beganon);
15720         }

15722         return (DDI_SUCCESS);
15723 }

15725 /*ARGSUSED*/
15726 static int
15727 dtrace_open(dev_t *devp, int flag, int otyp, cred_t *cred_p)
15728 {
15729         dtrace_state_t *state;
15730         uint32_t priv;
15731         uid_t uid;
15732         zoneid_t zoneid;

15734         if (getminor(*devp) == DTRACEMNRN_HELPER)
15735                 return (0);

15737         /*
15738          * If this wasn't an open with the "helper" minor, then it must be
15739          * the "dtrace" minor.
15740          */
15741         if (getminor(*devp) != DTRACEMNRN_DTRACE)
15742                 return (ENXIO);

15744         /*
15745          * If no DTRACE_PRIV_* bits are set in the credential, then the
15746          * caller lacks sufficient permission to do anything with DTrace.
15747          */
15748         dtrace_cred2priv(cred_p, &priv, &uid, &zoneid);
15749         if (priv == DTRACE_PRIV_NONE)
15750                 return (EACCES);

15752         /*
15753          * Ask all providers to provide all their probes.
15754          */
15755         mutex_enter(&dtrace_provider_lock);
15756         dtrace_probe_provide(NULL, NULL);
15757         mutex_exit(&dtrace_provider_lock);

15759         mutex_enter(&cpu_lock);
15760         mutex_enter(&dtrace_lock);
15761         dtrace_opens++;
15762         dtrace_membar_producer();

15764         /*
15765          * If the kernel debugger is active (that is, if the kernel debugger
15766          * modified text in some way), we won't allow the open.
15767          */
15768         if (kdi_dtrace_set(KDI_DTSET_DTRACE_ACTIVATE) != 0) {
```



```
15769                 dtrace_opens--;
15770                 mutex_exit(&cpu_lock);
15771                 mutex_exit(&dtrace_lock);
15772                 return (EBUSY);
15773         }

15775         state = dtrace_state_create(devp, cred_p);
15776         mutex_exit(&cpu_lock);

15778         if (state == NULL) {
15779                 if (--dtrace_opens == 0 && dtrace_anon.dta_enabling == NULL)
15780                         (void) kdi_dtrace_set(KDI_DTSET_DTRACE_DEACTIVATE);
15781                 mutex_exit(&dtrace_lock);
15782                 return (EAGAIN);
15783         }

15785         mutex_exit(&dtrace_lock);

15787         return (0);
15788 }

15790 /*ARGSUSED*/
15791 static int
15792 dtrace_close(dev_t dev, int flag, int otyp, cred_t *cred_p)
15793 {
15794         minor_t minor = getminor(dev);
15795         dtrace_state_t *state;

15797         if (minor == DTRACEMNRN_HELPER)
15798                 return (0);

15800         state = ddi_get_soft_state(dtrace_softstate, minor);

15802         mutex_enter(&cpu_lock);
15803         mutex_enter(&dtrace_lock);

15805         if (state->dts_anon) {
15806                 /*
15807                  * There is anonymous state. Destroy that first.
15808                  */
15809                 ASSERT(dtrace_anon.dta_state == NULL);
15810                 dtrace_state_destroy(state->dts_anon);
15811         }

15813         dtrace_state_destroy(state);
15814         ASSERT(dtrace_opens > 0);

15816         /*
15817          * Only relinquish control of the kernel debugger interface when there
15818          * are no consumers and no anonymous enablings.
15819          */
15820         if (--dtrace_opens == 0 && dtrace_anon.dta_enabling == NULL)
15821                 (void) kdi_dtrace_set(KDI_DTSET_DTRACE_DEACTIVATE);

15823         mutex_exit(&dtrace_lock);
15824         mutex_exit(&cpu_lock);

15826         return (0);
15827 }

15829 /*ARGSUSED*/
15830 static int
15831 dtrace_ioctl_helper(int cmd, intptr_t arg, int *rv)
15832 {
15833         int rval;
15834         dof_helper_t help, *dhp = NULL;
```

```
15836         switch (cmd) {
15837         case DTRACEHIOC_ADDDOF:
15838                 if (copyin((void *)arg, &help, sizeof (help)) != 0) {
15839                         dtrace_dof_error(NULL, "failed to copyin DOF helper");
15840                         return (EFAULT);
15841                 }
15843                 dhp = &help;
15844                 arg = (intptr_t)help.dofhp_dof;
15845                 /*FALLTHROUGH*/
15847         case DTRACEHIOC_ADD: {
15848                 dof_hdr_t *dof = dtrace_dof_copyin(arg, &rval);
15850                 if (dof == NULL)
15851                         return (rval);
15853                 mutex_enter(&dtrace_lock);
15855                 /*
15856                  * dtrace_helper_slurp() takes responsibility for the dof --
15857                  * it may free it now or it may save it and free it later.
15858                  */
15859                 if ((rval = dtrace_helper_slurp(dof, dhp)) != -1) {
15860                         *rv = rval;
15861                         rval = 0;
15862                 } else {
15863                         rval = EINVAL;
15864                 }
15866                 mutex_exit(&dtrace_lock);
15867                 return (rval);
15868         }
15870         case DTRACEHIOC_REMOVE: {
15871                 mutex_enter(&dtrace_lock);
15872                 rval = dtrace_helper_destroygen(arg);
15873                 mutex_exit(&dtrace_lock);
15875                 return (rval);
15876         }
15878         default:
15879                 break;
15880         }
15882         return (ENOTTY);
15883 }
15885 /*ARGSUSED*/
15886 static int
15887 dtrace_ioctl(dev_t dev, int cmd, intptr_t arg, int md, cred_t *cr, int *rv)
15888 {
15889         minor_t minor = getminor(dev);
15890         dtrace_state_t *state;
15891         int rval;
15893         if (minor == DTRACEMNRN_HELPER)
15894                 return (dtrace_ioctl_helper(cmd, arg, rv));
15896         state = ddi_get_soft_state(dtrace_softstate, minor);
15898         if (state->dts_anon) {
15899                 ASSERT(dtrace_anon.dta_state == NULL);
15900                 state = state->dts_anon;
```



```
15901         }
15903         switch (cmd) {
15904         case DTRACEIOC_PROVIDER: {
15905                 dtrace_providerdesc_t pvd;
15906                 dtrace_provider_t *pvp;
15908                 if (copyin((void *)arg, &pvd, sizeof (pvd)) != 0)
15909                         return (EFAULT);
15911                 pvd.dtvd_name[DTRACE_PROVNAMELEN - 1] = '\0';
15912                 mutex_enter(&dtrace_provider_lock);
15914                 for (pvp = dtrace_provider; pvp != NULL; pvp = pvp->dtpv_next) {
15915                         if (strcmp(pvp->dtpv_name, pvd.dtvd_name) == 0)
15916                                 break;
15917                 }
15919                 mutex_exit(&dtrace_provider_lock);
15921                 if (pvp == NULL)
15922                         return (ESRCH);
15924                 bcopy(&pvp->dtpv_priv, &pvd.dtvd_priv, sizeof (dtrace_ppriv_t));
15925                 bcopy(&pvp->dtpv_attr, &pvd.dtvd_attr, sizeof (dtrace_pattr_t));
15926                 if (copyout(&pvd, (void *)arg, sizeof (pvd)) != 0)
15927                         return (EFAULT);
15929                 return (0);
15930         }
15932         case DTRACEIOC_EPROBE: {
15933                 dtrace_eprobedesc_t epdesc;
15934                 dtrace_ecb_t *ecb;
15935                 dtrace_action_t *act;
15936                 void *buf;
15937                 size_t size;
15938                 uintptr_t dest;
15939                 int nrecs;
15941                 if (copyin((void *)arg, &epdesc, sizeof (epdesc)) != 0)
15942                         return (EFAULT);
15944                 mutex_enter(&dtrace_lock);
15946                 if ((ecb = dtrace_epid2ecb(state, epdesc.dtepd_epid)) == NULL) {
15947                         mutex_exit(&dtrace_lock);
15948                         return (EINVAL);
15949                 }
15951                 if (ecb->dte_probe == NULL) {
15952                         mutex_exit(&dtrace_lock);
15953                         return (EINVAL);
15954                 }
15956                 epdesc.dtepd_probeid = ecb->dte_probe->dtpr_id;
15957                 epdesc.dtepd_uarg = ecb->dte_uarg;
15958                 epdesc.dtepd_size = ecb->dte_size;
15960                 nrecs = epdesc.dtepd_nrecs;
15961                 epdesc.dtepd_nrecs = 0;
15962                 for (act = ecb->dte_action; act != NULL; act = act->dta_next) {
15963                         if (DTRACEACT_ISAGG(act->dta_kind) || act->dta_intuple)
15964                                 continue;
15966                         epdesc.dtepd_nrecs++;
```

```
15967                 }

15969                 /*
15970                  * Now that we have the size, we need to allocate a temporary
15971                  * buffer in which to store the complete description.  We need
15972                  * the temporary buffer to be able to drop dtrace_lock()
15973                  * across the copyout(), below.
15974                  */
15975                 size = sizeof (dtrace_eprobedesc_t) +
15976                     (epdesc.dtepd_nrecs * sizeof (dtrace_recdesc_t));

15978                 buf = kmem_alloc(size, KM_SLEEP);
15979                 dest = (uintptr_t)buf;

15981                 bcopy(&epdesc, (void *)dest, sizeof (epdesc));
15982                 dest += offsetof(dtrace_eprobedesc_t, dtepd_rec[0]);

15984                 for (act = ecb->dte_action; act != NULL; act = act->dta_next) {
15985                         if (DTRACEACT_ISAGG(act->dta_kind) || act->dta_intuple)
15986                                 continue;

15988                         if (nrecs-- == 0)
15989                                 break;

15991                         bcopy(&act->dta_rec, (void *)dest,
15992                             sizeof (dtrace_recdesc_t));
15993                         dest += sizeof (dtrace_recdesc_t);
15994                 }

15996                 mutex_exit(&dtrace_lock);

15998                 if (copyout(buf, (void *)arg, dest - (uintptr_t)buf) != 0) {
15999                         kmem_free(buf, size);
16000                         return (EFAULT);
16001                 }

16003                 kmem_free(buf, size);
16004                 return (0);
16005         }

16007         case DTRACEIOC_AGGDESC: {
16008                 dtrace_aggdesc_t aggdesc;
16009                 dtrace_action_t *act;
16010                 dtrace_aggregation_t *agg;
16011                 int nrecs;
16012                 uint32_t offs;
16013                 dtrace_recdesc_t *lrec;
16014                 void *buf;
16015                 size_t size;
16016                 uintptr_t dest;

16018                 if (copyin((void *)arg, &aggdesc, sizeof (aggdesc)) != 0)
16019                         return (EFAULT);

16021                 mutex_enter(&dtrace_lock);

16023                 if ((agg = dtrace_aggid2agg(state, aggdesc.dtagd_id)) == NULL) {
16024                         mutex_exit(&dtrace_lock);
16025                         return (EINVAL);
16026                 }

16028                 aggdesc.dtagd_epid = agg->dtag_ecb->dte_epid;

16030                 nrecs = aggdesc.dtagd_nrecs;
16031                 aggdesc.dtagd_nrecs = 0;
```



```
16033                 offs = agg->dtag_base;
16034                 lrec = &agg->dtag_action.dta_rec;
16035                 aggdesc.dtagd_size = lrec->dtrd_offset + lrec->dtrd_size - offs;

16037                 for (act = agg->dtag_first; ; act = act->dta_next) {
16038                         ASSERT(act->dta_intuple ||
16039                             DTRACEACT_ISAGG(act->dta_kind));

16041                         /*
16042                          * If this action has a record size of zero, it
16043                          * denotes an argument to the aggregating action.
16044                          * Because the presence of this record doesn't (or
16045                          * shouldn't) affect the way the data is interpreted,
16046                          * we don't copy it out to save user-level the
16047                          * confusion of dealing with a zero-length record.
16048                          */
16049                         if (act->dta_rec.dtrd_size == 0) {
16050                                 ASSERT(agg->dtag_hasarg);
16051                                 continue;
16052                         }

16054                         aggdesc.dtagd_nrecs++;

16056                         if (act == &agg->dtag_action)
16057                                 break;
16058                 }

16060                 /*
16061                  * Now that we have the size, we need to allocate a temporary
16062                  * buffer in which to store the complete description.  We need
16063                  * the temporary buffer to be able to drop dtrace_lock()
16064                  * across the copyout(), below.
16065                  */
16066                 size = sizeof (dtrace_aggdesc_t) +
16067                     (aggdesc.dtagd_nrecs * sizeof (dtrace_recdesc_t));

16069                 buf = kmem_alloc(size, KM_SLEEP);
16070                 dest = (uintptr_t)buf;

16072                 bcopy(&aggdesc, (void *)dest, sizeof (aggdesc));
16073                 dest += offsetof(dtrace_aggdesc_t, dtagd_rec[0]);

16075                 for (act = agg->dtag_first; ; act = act->dta_next) {
16076                         dtrace_recdesc_t rec = act->dta_rec;

16078                         /*
16079                          * See the comment in the above loop for why we pass
16080                          * over zero-length records.
16081                          */
16082                         if (rec.dtrd_size == 0) {
16083                                 ASSERT(agg->dtag_hasarg);
16084                                 continue;
16085                         }

16087                         if (nrecs-- == 0)
16088                                 break;

16090                         rec.dtrd_offset -= offs;
16091                         bcopy(&rec, (void *)dest, sizeof (rec));
16092                         dest += sizeof (dtrace_recdesc_t);

16094                         if (act == &agg->dtag_action)
16095                                 break;
16096                 }

16098                 mutex_exit(&dtrace_lock);
```

```
16100                 if (copyout(buf, (void *)arg, dest - (uintptr_t)buf) != 0) {
16101                         kmem_free(buf, size);
16102                         return (EFAULT);
16103                 }
16104
16105                 kmem_free(buf, size);
16106                 return (0);
16107         }
16108
16109         case DTRACEIOC_ENABLE: {
16110                 dof_hdr_t *dof;
16111                 dtrace_enabling_t *enab = NULL;
16112                 dtrace_vstate_t *vstate;
16113                 int err = 0;
16114
16115                 *rv = 0;
16116
16117                 /*
16118                  * If a NULL argument has been passed, we take this as our
16119                  * cue to reevaluate our enablings.
16120                  */
16121                 if (arg == NULL) {
16122                         dtrace_enabling_matchall();
16123
16124                         return (0);
16125                 }
16126
16127                 if ((dof = dtrace_dof_copyin(arg, &rval)) == NULL)
16128                         return (rval);
16129
16130                 mutex_enter(&cpu_lock);
16131                 mutex_enter(&dtrace_lock);
16132                 vstate = &state->dts_vstate;
16133
16134                 if (state->dts_activity != DTRACE_ACTIVITY_INACTIVE) {
16135                         mutex_exit(&dtrace_lock);
16136                         mutex_exit(&cpu_lock);
16137                         dtrace_dof_destroy(dof);
16138                         return (EBUSY);
16139                 }
16140
16141                 if (dtrace_dof_slurp(dof, vstate, cr, &enab, 0, B_TRUE) != 0) {
16142                         mutex_exit(&dtrace_lock);
16143                         mutex_exit(&cpu_lock);
16144                         dtrace_dof_destroy(dof);
16145                         return (EINVAL);
16146                 }
16147
16148                 if ((rval = dtrace_dof_options(dof, state)) != 0) {
16149                         dtrace_enabling_destroy(enab);
16150                         mutex_exit(&dtrace_lock);
16151                         mutex_exit(&cpu_lock);
16152                         dtrace_dof_destroy(dof);
16153                         return (rval);
16154                 }
16155
16156                 if ((err = dtrace_enabling_match(enab, rv)) == 0) {
16157                         err = dtrace_enabling_retain(enab);
16158                 } else {
16159                         dtrace_enabling_destroy(enab);
16160                 }
16161
16162                 mutex_exit(&cpu_lock);
16163                 mutex_exit(&dtrace_lock);
16164                 dtrace_dof_destroy(dof);
```



```
16166                 return (err);
16167         }
16168
16169         case DTRACEIOC_REPLICATE: {
16170                 dtrace_repldesc_t desc;
16171                 dtrace_probedesc_t *match = &desc.dtrpd_match;
16172                 dtrace_probedesc_t *create = &desc.dtrpd_create;
16173                 int err;
16174
16175                 if (copyin((void *)arg, &desc, sizeof (desc)) != 0)
16176                         return (EFAULT);
16177
16178                 match->dtpd_provider[DTRACE_PROVNAMELEN - 1] = '\0';
16179                 match->dtpd_mod[DTRACE_MODNAMELEN - 1] = '\0';
16180                 match->dtpd_func[DTRACE_FUNCNAMELEN - 1] = '\0';
16181                 match->dtpd_name[DTRACE_NAMELEN - 1] = '\0';
16182
16183                 create->dtpd_provider[DTRACE_PROVNAMELEN - 1] = '\0';
16184                 create->dtpd_mod[DTRACE_MODNAMELEN - 1] = '\0';
16185                 create->dtpd_func[DTRACE_FUNCNAMELEN - 1] = '\0';
16186                 create->dtpd_name[DTRACE_NAMELEN - 1] = '\0';
16187
16188                 mutex_enter(&dtrace_lock);
16189                 err = dtrace_enabling_replicate(state, match, create);
16190                 mutex_exit(&dtrace_lock);
16191
16192                 return (err);
16193         }
16194
16195         case DTRACEIOC_PROBEMATCH:
16196         case DTRACEIOC_PROBES: {
16197                 dtrace_probe_t *probe = NULL;
16198                 dtrace_probedesc_t desc;
16199                 dtrace_probekey_t pkey;
16200                 dtrace_id_t i;
16201                 int m = 0;
16202                 uint32_t priv;
16203                 uid_t uid;
16204                 zoneid_t zoneid;
16205
16206                 if (copyin((void *)arg, &desc, sizeof (desc)) != 0)
16207                         return (EFAULT);
16208
16209                 desc.dtpd_provider[DTRACE_PROVNAMELEN - 1] = '\0';
16210                 desc.dtpd_mod[DTRACE_MODNAMELEN - 1] = '\0';
16211                 desc.dtpd_func[DTRACE_FUNCNAMELEN - 1] = '\0';
16212                 desc.dtpd_name[DTRACE_NAMELEN - 1] = '\0';
16213
16214                 /*
16215                  * Before we attempt to match this probe, we want to give
16216                  * all providers the opportunity to provide it.
16217                  */
16218                 if (desc.dtpd_id == DTRACE_IDNONE) {
16219                         mutex_enter(&dtrace_provider_lock);
16220                         dtrace_probe_provide(&desc, NULL);
16221                         mutex_exit(&dtrace_provider_lock);
16222                         desc.dtpd_id++;
16223                 }
16224
16225                 if (cmd == DTRACEIOC_PROBEMATCH)  {
16226                         dtrace_probekey(&desc, &pkey);
16227                         pkey.dtpk_id = DTRACE_IDNONE;
16228                 }
16229
16230                 dtrace_cred2priv(cr, &priv, &uid, &zoneid);
```

```
16232                 mutex_enter(&dtrace_lock);

16234                 if (cmd == DTRACEIOC_PROBEMATCH) {
16235                         for (i = desc.dtpd_id; i <= dtrace_nprobes; i++) {
16236                                 if ((probe = dtrace_probes[i - 1]) != NULL &&
16237                                     (m = dtrace_match_probe(probe, &pkey,
16238                                     priv, uid, zoneid)) != 0)
16239                                         break;
16240                         }

16242                         if (m < 0) {
16243                                 mutex_exit(&dtrace_lock);
16244                                 return (EINVAL);
16245                         }

16247                 } else {
16248                         for (i = desc.dtpd_id; i <= dtrace_nprobes; i++) {
16249                                 if ((probe = dtrace_probes[i - 1]) != NULL &&
16250                                     dtrace_match_priv(probe, priv, uid, zoneid))
16251                                         break;
16252                         }
16253                 }

16255                 if (probe == NULL) {
16256                         mutex_exit(&dtrace_lock);
16257                         return (ESRCH);
16258                 }

16260                 dtrace_probe_description(probe, &desc);
16261                 mutex_exit(&dtrace_lock);

16263                 if (copyout(&desc, (void *)arg, sizeof (desc)) != 0)
16264                         return (EFAULT);

16266                 return (0);
16267         }

16269         case DTRACEIOC_PROBEARG: {
16270                 dtrace_argdesc_t desc;
16271                 dtrace_probe_t *probe;
16272                 dtrace_provider_t *prov;

16274                 if (copyin((void *)arg, &desc, sizeof (desc)) != 0)
16275                         return (EFAULT);

16277                 if (desc.dtargd_id == DTRACE_IDNONE)
16278                         return (EINVAL);

16280                 if (desc.dtargd_ndx == DTRACE_ARGNONE)
16281                         return (EINVAL);

16283                 mutex_enter(&dtrace_provider_lock);
16284                 mutex_enter(&mod_lock);
16285                 mutex_enter(&dtrace_lock);

16287                 if (desc.dtargd_id > dtrace_nprobes) {
16288                         mutex_exit(&dtrace_lock);
16289                         mutex_exit(&mod_lock);
16290                         mutex_exit(&dtrace_provider_lock);
16291                         return (EINVAL);
16292                 }

16294                 if ((probe = dtrace_probes[desc.dtargd_id - 1]) == NULL) {
16295                         mutex_exit(&dtrace_lock);
16296                         mutex_exit(&mod_lock);
```



```
16297                         mutex_exit(&dtrace_provider_lock);
16298                         return (EINVAL);
16299                 }

16301                 mutex_exit(&dtrace_lock);

16303                 prov = probe->dtpr_provider;

16305                 if (prov->dtpv_pops.dtps_getargdesc == NULL) {
16306                         /*
16307                          * There isn't any typed information for this probe.
16308                          * Set the argument number to DTRACE_ARGNONE.
16309                          */
16310                         desc.dtargd_ndx = DTRACE_ARGNONE;
16311                 } else {
16312                         desc.dtargd_native[0] = '\0';
16313                         desc.dtargd_xlate[0] = '\0';
16314                         desc.dtargd_mapping = desc.dtargd_ndx;

16316                         prov->dtpv_pops.dtps_getargdesc(prov->dtpv_arg,
16317                             probe->dtpr_id, probe->dtpr_arg, &desc);
16318                 }

16320                 mutex_exit(&mod_lock);
16321                 mutex_exit(&dtrace_provider_lock);

16323                 if (copyout(&desc, (void *)arg, sizeof (desc)) != 0)
16324                         return (EFAULT);

16326                 return (0);
16327         }

16329         case DTRACEIOC_GO: {
16330                 processorid_t cpuid;
16331                 rval = dtrace_state_go(state, &cpuid);

16333                 if (rval != 0)
16334                         return (rval);

16336                 if (copyout(&cpuid, (void *)arg, sizeof (cpuid)) != 0)
16337                         return (EFAULT);

16339                 return (0);
16340         }

16342         case DTRACEIOC_STOP: {
16343                 processorid_t cpuid;

16345                 mutex_enter(&dtrace_lock);
16346                 rval = dtrace_state_stop(state, &cpuid);
16347                 mutex_exit(&dtrace_lock);

16349                 if (rval != 0)
16350                         return (rval);

16352                 if (copyout(&cpuid, (void *)arg, sizeof (cpuid)) != 0)
16353                         return (EFAULT);

16355                 return (0);
16356         }

16358         case DTRACEIOC_DOFGET: {
16359                 dof_hdr_t hdr, *dof;
16360                 uint64_t len;

16362                 if (copyin((void *)arg, &hdr, sizeof (hdr)) != 0)
```

```
16363                         return (EFAULT);

16365                 mutex_enter(&dtrace_lock);
16366                 dof = dtrace_dof_create(state);
16367                 mutex_exit(&dtrace_lock);

16369                 len = MIN(hdr.dofh_loadsz, dof->dofh_loadsz);
16370                 rval = copyout(dof, (void *)arg, len);
16371                 dtrace_dof_destroy(dof);

16373                 return (rval == 0 ? 0 : EFAULT);
16374         }

16376         case DTRACEIOC_AGGSNAP:
16377         case DTRACEIOC_BUFSNAP: {
16378                 dtrace_bufdesc_t desc;
16379                 caddr_t cached;
16380                 dtrace_buffer_t *buf;

16382                 if (copyin((void *)arg, &desc, sizeof (desc)) != 0)
16383                         return (EFAULT);

16385                 if (desc.dtbd_cpu < 0 || desc.dtbd_cpu >= NCPU)
16386                         return (EINVAL);

16388                 mutex_enter(&dtrace_lock);

16390                 if (cmd == DTRACEIOC_BUFSNAP) {
16391                         buf = &state->dts_buffer[desc.dtbd_cpu];
16392                 } else {
16393                         buf = &state->dts_aggbuffer[desc.dtbd_cpu];
16394                 }

16396                 if (buf->dtb_flags & (DTRACEBUF_RING | DTRACEBUF_FILL)) {
16397                         size_t sz = buf->dtb_offset;

16399                         if (state->dts_activity != DTRACE_ACTIVITY_STOPPED) {
16400                                 mutex_exit(&dtrace_lock);
16401                                 return (EBUSY);
16402                         }

16404                         /*
16405                          * If this buffer has already been consumed, we're
16406                          * going to indicate that there's nothing left here
16407                          * to consume.
16408                          */
16409                         if (buf->dtb_flags & DTRACEBUF_CONSUMED) {
16410                                 mutex_exit(&dtrace_lock);

16412                                 desc.dtbd_size = 0;
16413                                 desc.dtbd_drops = 0;
16414                                 desc.dtbd_errors = 0;
16415                                 desc.dtbd_oldest = 0;
16416                                 sz = sizeof (desc);

16418                                 if (copyout(&desc, (void *)arg, sz) != 0)
16419                                         return (EFAULT);

16421                                 return (0);
16422                         }

16424                         /*
16425                          * If this is a ring buffer that has wrapped, we want
16426                          * to copy the whole thing out.
16427                          */
16428                         if (buf->dtb_flags & DTRACEBUF_WRAPPED) {
```



```
16429                                 dtrace_buffer_polish(buf);
16430                                 sz = buf->dtb_size;
16431                         }

16433                         if (copyout(buf->dtb_tomax, desc.dtbd_data, sz) != 0) {
16434                                 mutex_exit(&dtrace_lock);
16435                                 return (EFAULT);
16436                         }

16438                         desc.dtbd_size = sz;
16439                         desc.dtbd_drops = buf->dtb_drops;
16440                         desc.dtbd_errors = buf->dtb_errors;
16441                         desc.dtbd_oldest = buf->dtb_xamot_offset;
16442                         desc.dtbd_timestamp = dtrace_gethrtime();

16444                         mutex_exit(&dtrace_lock);

16446                         if (copyout(&desc, (void *)arg, sizeof (desc)) != 0)
16447                                 return (EFAULT);

16449                         buf->dtb_flags |= DTRACEBUF_CONSUMED;

16451                         return (0);
16452                 }

16454                 if (buf->dtb_tomax == NULL) {
16455                         ASSERT(buf->dtb_xamot == NULL);
16456                         mutex_exit(&dtrace_lock);
16457                         return (ENOENT);
16458                 }

16460                 cached = buf->dtb_tomax;
16461                 ASSERT(!(buf->dtb_flags & DTRACEBUF_NOSWITCH));

16463                 dtrace_xcall(desc.dtbd_cpu,
16464                     (dtrace_xcall_t)dtrace_buffer_switch, buf);

16466                 state->dts_errors += buf->dtb_xamot_errors;

16468                 /*
16469                  * If the buffers did not actually switch, then the cross call
16470                  * did not take place -- presumably because the given CPU is
16471                  * not in the ready set.  If this is the case, we'll return
16472                  * ENOENT.
16473                  */
16474                 if (buf->dtb_tomax == cached) {
16475                         ASSERT(buf->dtb_xamot != cached);
16476                         mutex_exit(&dtrace_lock);
16477                         return (ENOENT);
16478                 }

16480                 ASSERT(cached == buf->dtb_xamot);

16482                 /*
16483                  * We have our snapshot; now copy it out.
16484                  */
16485                 if (copyout(buf->dtb_xamot, desc.dtbd_data,
16486                     buf->dtb_xamot_offset) != 0) {
16487                         mutex_exit(&dtrace_lock);
16488                         return (EFAULT);
16489                 }

16491                 desc.dtbd_size = buf->dtb_xamot_offset;
16492                 desc.dtbd_drops = buf->dtb_xamot_drops;
16493                 desc.dtbd_errors = buf->dtb_xamot_errors;
16494                 desc.dtbd_oldest = 0;
```

```
16495                 desc.dtbd_timestamp = buf->dtb_switched;

16497                 mutex_exit(&dtrace_lock);

16499                 /*
16500                  * Finally, copy out the buffer description.
16501                  */
16502                 if (copyout(&desc, (void *)arg, sizeof (desc)) != 0)
16503                         return (EFAULT);

16505                 return (0);
16506         }

16508         case DTRACEIOC_CONF: {
16509                 dtrace_conf_t conf;

16511                 bzero(&conf, sizeof (conf));
16512                 conf.dtc_difversion = DIF_VERSION;
16513                 conf.dtc_difintregs = DIF_DIR_NREGS;
16514                 conf.dtc_diftupregs = DIF_DTR_NREGS;
16515                 conf.dtc_ctfmodel = CTF_MODEL_NATIVE;

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

16520                 return (0);
16521         }

16523         case DTRACEIOC_STATUS: {
16524                 dtrace_status_t stat;
16525                 dtrace_dstate_t *dstate;
16526                 int i, j;
16527                 uint64_t nerrs;

16529                 /*
16530                  * See the comment in dtrace_state_deadman() for the reason
16531                  * for setting dts_laststatus to INT64_MAX before setting
16532                  * it to the correct value.
16533                  */
16534                 state->dts_laststatus = INT64_MAX;
16535                 dtrace_membar_producer();
16536                 state->dts_laststatus = dtrace_gethrtime();

16538                 bzero(&stat, sizeof (stat));

16540                 mutex_enter(&dtrace_lock);

16542                 if (state->dts_activity == DTRACE_ACTIVITY_INACTIVE) {
16543                         mutex_exit(&dtrace_lock);
16544                         return (ENOENT);
16545                 }

16547                 if (state->dts_activity == DTRACE_ACTIVITY_DRAINING)
16548                         stat.dtst_exiting = 1;

16550                 nerrs = state->dts_errors;
16551                 dstate = &state->dts_vstate.dtvs_dynvars;

16553                 for (i = 0; i < NCPU; i++) {
16554                         dtrace_dstate_percpu_t *dcpu = &dstate->dtds_percpu[i];

16556                         stat.dtst_dyndrops += dcpu->dtdsc_drops;
16557                         stat.dtst_dyndrops_dirty += dcpu->dtdsc_dirty_drops;
16558                         stat.dtst_dyndrops_rinsing += dcpu->dtdsc_rinsing_drops;

16560                         if (state->dts_buffer[i].dtb_flags & DTRACEBUF_FULL)
```



```
16561                                 stat.dtst_filled++;

16563                         nerrs += state->dts_buffer[i].dtb_errors;

16565                         for (j = 0; j < state->dts_nspeculations; j++) {
16566                                 dtrace_speculation_t *spec;
16567                                 dtrace_buffer_t *buf;

16569                                 spec = &state->dts_speculations[j];
16570                                 buf = &spec->dtsp_buffer[i];
16571                                 stat.dtst_specdrops += buf->dtb_xamot_drops;
16572                         }
16573                 }

16575                 stat.dtst_specdrops_busy = state->dts_speculations_busy;
16576                 stat.dtst_specdrops_unavail = state->dts_speculations_unavail;
16577                 stat.dtst_stkstroverflows = state->dts_stkstroverflows;
16578                 stat.dtst_dblerrors = state->dts_dblerrors;
16579                 stat.dtst_killed =
16580                     (state->dts_activity == DTRACE_ACTIVITY_KILLED);
16581                 stat.dtst_errors = nerrs;

16583                 mutex_exit(&dtrace_lock);

16585                 if (copyout(&stat, (void *)arg, sizeof (stat)) != 0)
16586                         return (EFAULT);

16588                 return (0);
16589         }

16591         case DTRACEIOC_FORMAT: {
16592                 dtrace_fmtdesc_t fmt;
16593                 char *str;
16594                 int len;

16596                 if (copyin((void *)arg, &fmt, sizeof (fmt)) != 0)
16597                         return (EFAULT);

16599                 mutex_enter(&dtrace_lock);

16601                 if (fmt.dtfd_format == 0 ||
16602                     fmt.dtfd_format > state->dts_nformats) {
16603                         mutex_exit(&dtrace_lock);
16604                         return (EINVAL);
16605                 }

16607                 /*
16608                  * Format strings are allocated contiguously and they are
16609                  * never freed; if a format index is less than the number
16610                  * of formats, we can assert that the format map is non-NULL
16611                  * and that the format for the specified index is non-NULL.
16612                  */
16613                 ASSERT(state->dts_formats != NULL);
16614                 str = state->dts_formats[fmt.dtfd_format - 1];
16615                 ASSERT(str != NULL);

16617                 len = strlen(str) + 1;

16619                 if (len > fmt.dtfd_length) {
16620                         fmt.dtfd_length = len;

16622                         if (copyout(&fmt, (void *)arg, sizeof (fmt)) != 0) {
16623                                 mutex_exit(&dtrace_lock);
16624                                 return (EINVAL);
16625                         }
16626                 } else {
```

```
16627                         if (copyout(str, fmt.dtfd_string, len) != 0) {
16628                                 mutex_exit(&dtrace_lock);
16629                                 return (EINVAL);
16630                         }
16631                 }

16633                 mutex_exit(&dtrace_lock);
16634                 return (0);
16635         }

16637         default:
16638                 break;
16639         }

16641         return (ENOTTY);
16642 }

16644 /*ARGSUSED*/
16645 static int
16646 dtrace_detach(dev_info_t *dip, ddi_detach_cmd_t cmd)
16647 {
16648         dtrace_state_t *state;

16650         switch (cmd) {
16651         case DDI_DETACH:
16652                 break;

16654         case DDI_SUSPEND:
16655                 return (DDI_SUCCESS);

16657         default:
16658                 return (DDI_FAILURE);
16659         }

16661         mutex_enter(&cpu_lock);
16662         mutex_enter(&dtrace_provider_lock);
16663         mutex_enter(&dtrace_lock);

16665         ASSERT(dtrace_opens == 0);

16667         if (dtrace_helpers > 0) {
16668                 mutex_exit(&dtrace_provider_lock);
16669                 mutex_exit(&dtrace_lock);
16670                 mutex_exit(&cpu_lock);
16671                 return (DDI_FAILURE);
16672         }

16674         if (dtrace_unregister((dtrace_provider_id_t)dtrace_provider) != 0) {
16675                 mutex_exit(&dtrace_provider_lock);
16676                 mutex_exit(&dtrace_lock);
16677                 mutex_exit(&cpu_lock);
16678                 return (DDI_FAILURE);
16679         }

16681         dtrace_provider = NULL;

16683         if ((state = dtrace_anon_grab()) != NULL) {
16684                 /*
16685                  * If there were ECBs on this state, the provider should
16686                  * have not been allowed to detach; assert that there is
16687                  * none.
16688                  */
16689                 ASSERT(state->dts_necbs == 0);
16690                 dtrace_state_destroy(state);

16692                 /*
```



```
16693                  * If we're being detached with anonymous state, we need to
16694                  * indicate to the kernel debugger that DTrace is now inactive.
16695                  */
16696                 (void) kdi_dtrace_set(KDI_DTSET_DTRACE_DEACTIVATE);
16697         }

16699         bzero(&dtrace_anon, sizeof (dtrace_anon_t));
16700         unregister_cpu_setup_func((cpu_setup_func_t *)dtrace_cpu_setup, NULL);
16701         dtrace_cpu_init = NULL;
16702         dtrace_helpers_cleanup = NULL;
16703         dtrace_helpers_fork = NULL;
16704         dtrace_cpustart_init = NULL;
16705         dtrace_cpustart_fini = NULL;
16706         dtrace_debugger_init = NULL;
16707         dtrace_debugger_fini = NULL;
16708         dtrace_modload = NULL;
16709         dtrace_modunload = NULL;

16711         ASSERT(dtrace_getf == 0);
16712         ASSERT(dtrace_closef == NULL);

16714         mutex_exit(&cpu_lock);

16716         if (dtrace_helptrace_enabled) {
16717                 kmem_free(dtrace_helptrace_buffer, dtrace_helptrace_bufsize);
16718                 dtrace_helptrace_buffer = NULL;
16719         }

16721         kmem_free(dtrace_probes, dtrace_nprobes * sizeof (dtrace_probe_t *));
16722         dtrace_probes = NULL;
16723         dtrace_nprobes = 0;

16725         dtrace_hash_destroy(dtrace_bymod);
16726         dtrace_hash_destroy(dtrace_byfunc);
16727         dtrace_hash_destroy(dtrace_byname);
16728         dtrace_bymod = NULL;
16729         dtrace_byfunc = NULL;
16730         dtrace_byname = NULL;

16732         kmem_cache_destroy(dtrace_state_cache);
16733         vmem_destroy(dtrace_minor);
16734         vmem_destroy(dtrace_arena);

16736         if (dtrace_toxrange != NULL) {
16737                 kmem_free(dtrace_toxrange,
16738                     dtrace_toxranges_max * sizeof (dtrace_toxrange_t));
16739                 dtrace_toxrange = NULL;
16740                 dtrace_toxranges = 0;
16741                 dtrace_toxranges_max = 0;
16742         }

16744         ddi_remove_minor_node(dtrace_devi, NULL);
16745         dtrace_devi = NULL;

16747         ddi_soft_state_fini(&dtrace_softstate);

16749         ASSERT(dtrace_vtime_references == 0);
16750         ASSERT(dtrace_opens == 0);
16751         ASSERT(dtrace_retained == NULL);

16753         mutex_exit(&dtrace_lock);
16754         mutex_exit(&dtrace_provider_lock);

16756         /*
16757          * We don't destroy the task queue until after we have dropped our
16758          * locks (taskq_destroy() may block on running tasks).  To prevent
```

```
16759          * attempting to do work after we have effectively detached but before
16760          * the task queue has been destroyed, all tasks dispatched via the
16761          * task queue must check that DTrace is still attached before
16762          * performing any operation.
16763          */
16764         taskq_destroy(dtrace_taskq);
16765         dtrace_taskq = NULL;

16767         return (DDI_SUCCESS);
16768 }

16770 /*ARGSUSED*/
16771 static int
16772 dtrace_info(dev_info_t *dip, ddi_info_cmd_t infocmd, void *arg, void **result)
16773 {
16774         int error;

16776         switch (infocmd) {
16777         case DDI_INFO_DEVT2DEVINFO:
16778                 *result = (void *)dtrace_devi;
16779                 error = DDI_SUCCESS;
16780                 break;
16781         case DDI_INFO_DEVT2INSTANCE:
16782                 *result = (void *)0;
16783                 error = DDI_SUCCESS;
16784                 break;
16785         default:
16786                 error = DDI_FAILURE;
16787         }
16788         return (error);
16789 }

16791 static struct cb_ops dtrace_cb_ops = {
16792         dtrace_open,            /* open */
16793         dtrace_close,           /* close */
16794         nulldev,                /* strategy */
16795         nulldev,                /* print */
16796         nodev,                  /* dump */
16797         nodev,                  /* read */
16798         nodev,                  /* write */
16799         dtrace_ioctl,           /* ioctl */
16800         nodev,                  /* devmap */
16801         nodev,                  /* mmap */
16802         nodev,                  /* segmap */
16803         nochpoll,               /* poll */
16804         ddi_prop_op,            /* cb_prop_op */
16805         0,                      /* streamtab  */
16806         D_NEW | D_MP            /* Driver compatibility flag */
16807 };

16809 static struct dev_ops dtrace_ops = {
16810         DEVO_REV,               /* devo_rev */
16811         0,                      /* refcnt */
16812         dtrace_info,            /* get_dev_info */
16813         nulldev,                /* identify */
16814         nulldev,                /* probe */
16815         dtrace_attach,          /* attach */
16816         dtrace_detach,          /* detach */
16817         nodev,                  /* reset */
16818         &dtrace_cb_ops,         /* driver operations */
16819         NULL,                   /* bus operations */
16820         nodev,                  /* dev power */
16821         ddi_quiesce_not_needed,         /* quiesce */
16822 };

16824 static struct modldrv modldrv = {
```



```
16825         &mod_driverops,         /* module type (this is a pseudo driver) */
16826         "Dynamic Tracing",      /* name of module */
16827         &dtrace_ops,            /* driver ops */
16828 };

16830 static struct modlinkage modlinkage = {
16831         MODREV_1,
16832         (void *)&modldrv,
16833         NULL
16834 };

16836 int
16837 _init(void)
16838 {
16839         return (mod_install(&modlinkage));
16840 }

16842 int
16843 _info(struct modinfo *modinfop)
16844 {
16845         return (mod_info(&modlinkage, modinfop));
16846 }

16848 int
16849 _fini(void)
16850 {
16851         return (mod_remove(&modlinkage));
16852 }
```

```
**********************************************************
  101984 Tue Jan 14 16:49:37 2014
new/usr/src/uts/common/sys/dtrace.h
4477 DTrace should speak JSON
Reviewed by: Bryan Cantrill <bmc@joyent.com>
**********************************************************
______unchanged_portion_omitted_

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

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



```
 155 #define DIF_OP_LDTA     43              /* ldta var, ri, rd */
 156 #define DIF_OP_LDTS     44              /* ldts var, rd */
 157 #define DIF_OP_STTS     45              /* stts var, rs */
 158 #define DIF_OP_SRA      46              /* sra  r1, r2, rd */
 159 #define DIF_OP_CALL     47              /* call subr, rd */
 160 #define DIF_OP_PUSHTR   48              /* pushtr type, rs, rr */
 161 #define DIF_OP_PUSHTV   49              /* pushtv type, rs, rv */
 162 #define DIF_OP_POPTS    50              /* popts */
 163 #define DIF_OP_FLUSHTS  51              /* flushts */
 164 #define DIF_OP_LDGAA    52              /* ldgaa var, rd */
 165 #define DIF_OP_LDTAA    53              /* ldtaa var, rd */
 166 #define DIF_OP_STGAA    54              /* stgaa var, rs */
 167 #define DIF_OP_STTAA    55              /* sttaa var, rs */
 168 #define DIF_OP_LDLS     56              /* ldls var, rd */
 169 #define DIF_OP_STLS     57              /* stls var, rs */
 170 #define DIF_OP_ALLOCS   58              /* allocs r1, rd */
 171 #define DIF_OP_COPYS    59              /* copys  r1, r2, rd */
 172 #define DIF_OP_STB      60              /* stb  r1, [rd] */
 173 #define DIF_OP_STH      61              /* sth  r1, [rd] */
 174 #define DIF_OP_STW      62              /* stw  r1, [rd] */
 175 #define DIF_OP_STX      63              /* stx  r1, [rd] */
 176 #define DIF_OP_ULDSB    64              /* uldsb [r1], rd */
 177 #define DIF_OP_ULDSH    65              /* uldsh [r1], rd */
 178 #define DIF_OP_ULDSW    66              /* uldsw [r1], rd */
 179 #define DIF_OP_ULDUB    67              /* uldub [r1], rd */
 180 #define DIF_OP_ULDUH    68              /* ulduh [r1], rd */
 181 #define DIF_OP_ULDUW    69              /* ulduw [r1], rd */
 182 #define DIF_OP_ULDX     70              /* uldx  [r1], rd */
 183 #define DIF_OP_RLDSB    71              /* rldsb [r1], rd */
 184 #define DIF_OP_RLDSH    72              /* rldsh [r1], rd */
 185 #define DIF_OP_RLDSW    73              /* rldsw [r1], rd */
 186 #define DIF_OP_RLDUB    74              /* rldub [r1], rd */
 187 #define DIF_OP_RLDUH    75              /* rlduh [r1], rd */
 188 #define DIF_OP_RLDUW    76              /* rlduw [r1], rd */
 189 #define DIF_OP_RLDX     77              /* rldx  [r1], rd */
 190 #define DIF_OP_XLATE    78              /* xlate xlrindex, rd */
 191 #define DIF_OP_XLARG    79              /* xlarg xlrindex, rd */

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

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

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

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

```
221 #define DIF_VAR_ARG4            0x010a  /* fifth argument */
222 #define DIF_VAR_ARG5            0x010b  /* sixth argument */
223 #define DIF_VAR_ARG6            0x010c  /* seventh argument */
224 #define DIF_VAR_ARG7            0x010d  /* eighth argument */
225 #define DIF_VAR_ARG8            0x010e  /* ninth argument */
226 #define DIF_VAR_ARG9            0x010f  /* tenth argument */
227 #define DIF_VAR_STACKDEPTH      0x0110  /* stack depth */
228 #define DIF_VAR_CALLER          0x0111  /* caller */
229 #define DIF_VAR_PROBEPROV       0x0112  /* probe provider */
230 #define DIF_VAR_PROBEMOD        0x0113  /* probe module */
231 #define DIF_VAR_PROBEFUNC       0x0114  /* probe function */
232 #define DIF_VAR_PROBENAME       0x0115  /* probe name */
233 #define DIF_VAR_PID             0x0116  /* process ID */
234 #define DIF_VAR_TID             0x0117  /* (per-process) thread ID */
235 #define DIF_VAR_EXECNAME        0x0118  /* name of executable */
236 #define DIF_VAR_ZONENAME        0x0119  /* zone name associated with process */
237 #define DIF_VAR_WALLTIMESTAMP   0x011a  /* wall-clock timestamp */
238 #define DIF_VAR_USTACKDEPTH     0x011b  /* user-land stack depth */
239 #define DIF_VAR_UCALLER         0x011c  /* user-level caller */
240 #define DIF_VAR_PPID            0x011d  /* parent process ID */
241 #define DIF_VAR_UID             0x011e  /* process user ID */
242 #define DIF_VAR_GID             0x011f  /* process group ID */
243 #define DIF_VAR_ERRNO           0x0120  /* thread errno */

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



```
287 #define DIF_SUBR_INET_NTOA              42
288 #define DIF_SUBR_INET_NTOA6             43
289 #define DIF_SUBR_TOUPPER                44
290 #define DIF_SUBR_TOLOWER                45
291 #define DIF_SUBR_GETF                   46
292 #define DIF_SUBR_JSON                   47
293 #define DIF_SUBR_STRTOLL                48
294 #endif /* ! codereview */

296 #define DIF_SUBR_MAX                    48      /* max subroutine value */
292 #define DIF_SUBR_MAX                    46      /* max subroutine value */

298 typedef uint32_t dif_instr_t;

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

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

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

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

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

```
 352 } dtrace_diftype_t;
______unchanged_portion_omitted_
```

```
**********************************************************
    2206 Tue Jan 14 16:49:38 2014
new/usr/src/uts/intel/dtrace/Makefile
4477 DTrace should speak JSON
Reviewed by: Bryan Cantrill <bmc@joyent.com>
**********************************************************
```

```
#
# CDDL HEADER START
#
# The contents of this file are subject to the terms of the
# Common Development and Distribution License (the "License").
# You may not use this file except in compliance with the License.
#
# You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
# or http://www.opensolaris.org/os/licensing.
# See the License for the specific language governing permissions
# and limitations under the License.
#
# When distributing Covered Code, include this CDDL HEADER in each
# file and include the License file at usr/src/OPENSOLARIS.LICENSE.
# If applicable, add the following below this CDDL HEADER, with the
# fields enclosed by brackets "[]" replaced with your own identifying
# information: Portions Copyright [yyyy] [name of copyright owner]
#
# CDDL HEADER END
#
#
# Copyright 2008 Sun Microsystems, Inc.  All rights reserved.
# Use is subject to license terms.
#

UTSBASE = ../..

MODULE          = dtrace
OBJECTS         = $(DTRACE_OBJS:%=$(OBJS_DIR)/%)
LINTS           = $(DTRACE_OBJS:%.o=$(LINTS_DIR)/%.ln)
ROOTMODULE      = $(ROOT_DRV_DIR)/$(MODULE)
CONF_SRCDIR     = $(UTSBASE)/common/dtrace

include $(UTSBASE)/intel/Makefile.intel

#
# For now, disable these lint checks; maintainers should endeavor
# to investigate and remove these for maximum lint coverage.
# Please do not carry these forward to new Makefiles.
#
LINTTAGS        += -erroff=E_SUSPICIOUS_COMPARISON
LINTTAGS        += -erroff=E_BAD_PTR_CAST_ALIGN
LINTTAGS        += -erroff=E_SUPPRESSION_DIRECTIVE_UNUSED
LINTTAGS        += -erroff=E_STATIC_UNUSED
LINTTAGS        += -erroff=E_PTRDIFF_OVERFLOW
LINTTAGS        += -erroff=E_ASSIGN_NARROW_CONV

CERRWARN        += -_gcc=-Wno-parentheses
CERRWARN        += -_gcc=-Wno-type-limits
CERRWARN        += -_gcc=-Wno-uninitialized

CPPFLAGS        += -I$(SRC)/common/util

#endif /* ! codereview */
ALL_TARGET      = $(BINARY) $(SRC_CONFILE)
LINT_TARGET     = $(MODULE).lint
INSTALL_TARGET  = $(BINARY) $(ROOTMODULE) $(ROOT_CONFFILE)
AS_INC_PATH     += -I$(DSF_DIR)/$(OBJS_DIR)

ASSYM_H         = $(DSF_DIR)/$(OBJS_DIR)/assym.h
```



```
.KEEP_STATE:

def:            $(DEF_DEPS)

all:            $(ALL_DEPS)

clean:          $(CLEAN_DEPS)

clobber:        $(CLOBBER_DEPS)

lint:           $(LINT_DEPS)

modlintlib:     $(MODLINTLIB_DEPS)

clean.lint:     $(CLEAN_LINT_DEPS)

install:        $(INSTALL_DEPS)

$(BINARY):      $(ASSYM_H)

include $(UTSBASE)/intel/Makefile.targ
```

```
**********************************************************
    2550 Tue Jan 14 16:49:39 2014
new/usr/src/uts/sparc/dtrace/Makefile
4477 DTrace should speak JSON
Reviewed by: Bryan Cantrill <bmc@joyent.com>
**********************************************************
```

```
#
# CDDL HEADER START
#
# The contents of this file are subject to the terms of the
# Common Development and Distribution License (the "License").
# You may not use this file except in compliance with the License.
#
# You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
# or http://www.opensolaris.org/os/licensing.
# See the License for the specific language governing permissions
# and limitations under the License.
#
# When distributing Covered Code, include this CDDL HEADER in each
# file and include the License file at usr/src/OPENSOLARIS.LICENSE.
# If applicable, add the following below this CDDL HEADER, with the
# fields enclosed by brackets "[]" replaced with your own identifying
# information: Portions Copyright [yyyy] [name of copyright owner]
#
# CDDL HEADER END
#
#
# Copyright 2008 Sun Microsystems, Inc.  All rights reserved.
# Use is subject to license terms.
#

UTSBASE = ../..

PLATFORM        = sun4u
MODULE          = dtrace
OBJECTS         = $(DTRACE_OBJS:%=$(OBJS_DIR)/%)
LINTS           = $(DTRACE_OBJS:%.o=$(LINTS_DIR)/%.ln)
ROOTMODULE      = $(ROOT_DRV_DIR)/$(MODULE)
CONF_SRCDIR     = $(UTSBASE)/common/dtrace

include $(UTSBASE)/sparc/Makefile.sparc

#
#       Redefine        DSF_DIR
#
DSF_DIR         = $(UTSBASE)/$(PLATFORM)/genassym

CERRWARN        += -_gcc=-Wno-parentheses
CERRWARN        += -_gcc=-Wno-type-limits
CERRWARN        += -_gcc=-Wno-uninitialized

ALL_TARGET      = $(BINARY) $(SRC_CONFILE)
LINT_TARGET     = $(MODULE).lint
INSTALL_TARGET  = $(BINARY) $(ROOTMODULE) $(ROOT_CONFFILE)

DTRACE_INC_32   = -I$(UTSBASE)/sparc/v7
DTRACE_INC_64   = -I$(UTSBASE)/sparc/v9

CFLAGS += $(CCVERBOSE)
CPPFLAGS += $(DTRACE_INC_$(CLASS))
CPPFLAGS += -I$(SRC)/common/util
#endif /* ! codereview */

DTRACE_XAS_32   = -xarch=v8plus
DTRACE_XAS_64   = -xarch=v9
```



```
AS_CPPFLAGS     += $(DTRACE_INC_64)
ASFLAGS         += $(DTRACE_XAS_$(CLASS))
AS_INC_PATH     += -I$(DSF_DIR)/$(OBJS_DIR)

ASSYM_H         = $(DSF_DIR)/$(OBJS_DIR)/assym.h

#
# For now, disable these lint checks; maintainers should endeavor
# to investigate and remove these for maximum lint coverage.
# Please do not carry these forward to new Makefiles.
#
LINTTAGS        += -erroff=E_SUSPICIOUS_COMPARISON
LINTTAGS        += -erroff=E_BAD_PTR_CAST_ALIGN
LINTTAGS        += -erroff=E_SUPPRESSION_DIRECTIVE_UNUSED
LINTTAGS        += -erroff=E_STATIC_UNUSED
LINTTAGS        += -erroff=E_PTRDIFF_OVERFLOW
LINTTAGS        += -erroff=E_ASSIGN_NARROW_CONV

.KEEP_STATE:

def:            $(DEF_DEPS)

all:            $(ALL_DEPS)

clean:          $(CLEAN_DEPS)

clobber:        $(CLOBBER_DEPS)

lint:           $(LINT_DEPS)

modlintlib:     $(MODLINTLIB_DEPS)

clean.lint:     $(CLEAN_LINT_DEPS)

install:        $(INSTALL_DEPS)

$(BINARY):      $(ASSYM_H)

include $(UTSBASE)/sparc/Makefile.targ
```