```
**********************************************************
   29173 Mon Feb  8 20:19:28 2016
new/usr/src/cmd/mdb/common/modules/genunix/vfs.c
6638 ::pfiles walks out of bounds on array of vnode types
**********************************************************
______unchanged_portion_omitted_

 530 const struct fs_type {
 531         vtype_t type;
 531         int type;
 532         const char *name;
 533 } fs_types[] = {
______unchanged_portion_omitted_

 922 #define NUM_SOCK_PRINTS                                         \
 923         (sizeof (sock_prints) / sizeof (struct sock_print))

 925 static int
 926 pfile_callback(uintptr_t addr, const struct file *f, struct pfiles_cbdata *cb)
 927 {
 928         vnode_t v, layer_vn;
 929         int myfd = cb->fd;
 930         const char *type;
 931         char path[MAXPATHLEN];
 932         uintptr_t top_vnodep, realvpp;
 933         char fsname[_ST_FSTYPSZ];
 934         int err, i;

 936         cb->fd++;

 938         if (addr == NULL) {
 939                 return (WALK_NEXT);
 940         }

 942         top_vnodep = realvpp = (uintptr_t)f->f_vnode;

 944         if (mdb_vread(&v, sizeof (v), realvpp) == -1) {
 945                 mdb_warn("failed to read vnode");
 946                 return (DCMD_ERR);
 947         }

 949         type = "?";
 950         for (i = 0; i < NUM_FS_TYPES; i++) {
 951                 if (fs_types[i].type == v.v_type) {
 950         for (i = 0; i <= NUM_FS_TYPES; i++) {
 951                 if (fs_types[i].type == v.v_type)
 952                         type = fs_types[i].name;
 953                         break;
 954                 }
 955 #endif /* ! codereview */
 956         }

 958         do {
 959                 uintptr_t next_realvpp;

 961                 err = next_realvp(realvpp, &layer_vn, &next_realvpp);
 962                 if (next_realvpp != NULL)
 963                         realvpp = next_realvpp;

 965         } while (err == REALVP_CONTINUE);

 967         if (err == REALVP_ERR) {
 968                 mdb_warn("failed to do realvp() for %p", realvpp);
 969                 return (DCMD_ERR);
 970         }
```



```
 972         if (read_fsname((uintptr_t)layer_vn.v_vfsp, fsname) == -1)
 973                 return (DCMD_ERR);

 975         mdb_printf("%4d %4s %?0p ", myfd, type, top_vnodep);

 977         if (cb->opt_p) {
 978                 if (pfiles_dig_pathname(top_vnodep, path) == -1)
 979                         return (DCMD_ERR);

 981                 mdb_printf("%s\n", path);
 982                 return (DCMD_OK);
 983         }

 985         /*
 986          * Sockets generally don't have interesting pathnames; we only
 987          * show those in the '-p' view.
 988          */
 989         path[0] = '\0';
 990         if (v.v_type != VSOCK) {
 991                 if (pfiles_dig_pathname(top_vnodep, path) == -1)
 992                         return (DCMD_ERR);
 993         }
 994         mdb_printf("%s%s", path, path[0] == '\0' ? "" : " ");

 996         switch (v.v_type) {
 997         case VDOOR:
 998         {
 999                 door_node_t doornode;
1000                 proc_t pr;

1002                 if (mdb_vread(&doornode, sizeof (doornode),
1003                     (uintptr_t)layer_vn.v_data) == -1) {
1004                         mdb_warn("failed to read door_node");
1005                         return (DCMD_ERR);
1006                 }

1008                 if (mdb_vread(&pr, sizeof (pr),
1009                     (uintptr_t)doornode.door_target) == -1) {
1010                         mdb_warn("failed to read door server process %p",
1011                             doornode.door_target);
1012                         return (DCMD_ERR);
1013                 }
1014                 mdb_printf("[door to '%s' (proc=%p)]", pr.p_user.u_comm,
1015                     doornode.door_target);
1016                 break;
1017         }

1019         case VSOCK:
1020         {
1021                 vnode_t v_sock;
1022                 struct sonode so;

1024                 if (mdb_vread(&v_sock, sizeof (v_sock), realvpp) == -1) {
1025                         mdb_warn("failed to read socket vnode");
1026                         return (DCMD_ERR);
1027                 }

1029                 /*
1030                  * Sockets can be non-stream or stream, they have to be dealed
1031                  * with differently.
1032                  */
1033                 if (v_sock.v_stream == NULL) {
1034                         if (pfiles_get_sonode(&v_sock, &so) == -1)
1035                                 return (DCMD_ERR);

1037                         /* Pick the proper methods. */
```

```
1038                        for (i = 0; i <= NUM_SOCK_PRINTS; i++) {
1039                                if ((sock_prints[i].family == so.so_family &&
1040                                    sock_prints[i].type == so.so_type &&
1041                                    sock_prints[i].pro == so.so_protocol) ||
1042                                    (sock_prints[i].family == so.so_family &&
1043                                    sock_prints[i].type == so.so_type &&
1044                                    so.so_type == SOCK_RAW)) {
1045                                        if ((*sock_prints[i].print)(&so) == -1)
1046                                                return (DCMD_ERR);
1047                                }
1048                        }
1049                } else {
1050                        sotpi_sonode_t sotpi_sonode;

1052                        if (pfiles_get_sonode(&v_sock, &so) == -1)
1053                                return (DCMD_ERR);

1055                        /*
1056                         * If the socket is a fallback socket, read its related
1057                         * information separately; otherwise, read it as a whole
1058                         * tpi socket.
1059                         */
1060                        if (so.so_state & SS_FALLBACK_COMP) {
1061                                sotpi_sonode.st_sonode = so;

1063                                if (mdb_vread(&(sotpi_sonode.st_info),
1064                                    sizeof (sotpi_info_t),
1065                                    (uintptr_t)so.so_priv) == -1)
1066                                        return (DCMD_ERR);
1067                        } else {
1068                                if (pfiles_get_tpi_sonode(&v_sock,
1069                                    &sotpi_sonode) == -1)
1070                                        return (DCMD_ERR);
1071                        }

1073                        if (tpi_sock_print(&sotpi_sonode) == -1)
1074                                return (DCMD_ERR);
1075                }

1077                break;
1078        }

1080        case VPORT:
1081                mdb_printf("[event port (port=%p)]", v.v_data);
1082                break;

1084        case VPROC:
1085        {
1086                prnode_t prnode;
1087                prcommon_t prcommon;

1089                if (mdb_vread(&prnode, sizeof (prnode),
1090                    (uintptr_t)layer_vn.v_data) == -1) {
1091                        mdb_warn("failed to read prnode");
1092                        return (DCMD_ERR);
1093                }

1095                if (mdb_vread(&prcommon, sizeof (prcommon),
1096                    (uintptr_t)prnode.pr_common) == -1) {
1097                        mdb_warn("failed to read prcommon %p",
1098                            prnode.pr_common);
1099                        return (DCMD_ERR);
1100                }

1102                mdb_printf("(proc=%p)", prcommon.prc_proc);
1103                break;
```



```
1104        }

1106        default:
1107                break;
1108        }

1110        mdb_printf("\n");

1112        return (WALK_NEXT);
1113 }

1115 static int
1116 file_t_callback(uintptr_t addr, const struct file *f, struct pfiles_cbdata *cb)
1117 {
1118        int myfd = cb->fd;

1120        cb->fd++;

1122        if (addr == NULL) {
1123                return (WALK_NEXT);
1124        }

1126        /*
1127         * We really need 20 digits to print a 64-bit offset_t, but this
1128         * is exceedingly rare, so we cheat and assume a column width of 10
1129         * digits, in order to fit everything cleanly into 80 columns.
1130         */
1131        mdb_printf("%?0p %4d %8x %?0p %10lld %?0p %4d\n",
1132            addr, myfd, f->f_flag, f->f_vnode, f->f_offset, f->f_cred,
1133            f->f_count);

1135        return (WALK_NEXT);
1136 }

1138 int
1139 pfiles(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
1140 {
1141        int opt_f = 0;

1143        struct pfiles_cbdata cb;

1145        bzero(&cb, sizeof (cb));

1147        if (!(flags & DCMD_ADDRSPEC))
1148                return (DCMD_USAGE);

1150        if (mdb_getopts(argc, argv,
1151            'p', MDB_OPT_SETBITS, TRUE, &cb.opt_p,
1152            'f', MDB_OPT_SETBITS, TRUE, &opt_f, NULL) != argc)
1153                return (DCMD_USAGE);

1155        if (opt_f) {
1156                mdb_printf("%<u>%?s %4s %8s %?s %10s %?s %4s%</u>\n", "FILE",
1157                    "FD", "FLAG", "VNODE", "OFFSET", "CRED", "CNT");
1158                if (mdb_pwalk("allfile", (mdb_walk_cb_t)file_t_callback, &cb,
1159                    addr) == -1) {
1160                        mdb_warn("failed to walk 'allfile'");
1161                        return (DCMD_ERR);
1162                }
1163        } else {
1164                mdb_printf("%<u>%-4s %4s %?s ", "FD", "TYPE", "VNODE");
1165                if (cb.opt_p)
1166                        mdb_printf("PATH");
1167                else
1168                        mdb_printf("INFO");
1169                mdb_printf("%</u>\n");
```

```
1171                 if (mdb_pwalk("allfile", (mdb_walk_cb_t)pfile_callback, &cb,
1172                     addr) == -1) {
1173                         mdb_warn("failed to walk 'allfile'");
1174                         return (DCMD_ERR);
1175                 }
1176         }


1179         return (DCMD_OK);
1180 }

1182 void
1183 pfiles_help(void)
1184 {
1185         mdb_printf(
1186             "Given the address of a process, print information about files\n"
1187             "which the process has open.  By default, this includes decoded\n"
1188             "information about the file depending on file and filesystem type\n"
1189             "\n"
1190             "\t-p\tPathnames; omit decoded information.  Only display "
1191             "pathnames\n"
1192             "\t-f\tfile_t view; show the file_t structure corresponding to "
1193             "the fd\n");
1194 }
```

****
16933 Mon Feb 8 20:19:28 2016
new/usr/src/cmd/mdb/common/modules/uhci/uhci.c
6639 uhci_qh walker contains whacky boolean logic
****

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

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

#include <gelf.h>

#include <sys/mdb_modapi.h>
#include <mdb/mdb_ks.h>

#include <sys/usb/usba.h>
#include <sys/usb/usba/usba_types.h>

#include <sys/usb/hcd/uhci/uhci.h>
#include <sys/usb/hcd/uhci/uhcid.h>
#include <sys/usb/hcd/uhci/uhciutil.h>


#define UHCI_TD 0
#define UHCI_QH 1


/* Prototypes */

int     uhci_td(uintptr_t, uint_t, int, const mdb_arg_t *);
int     uhci_qh(uintptr_t, uint_t, int, const mdb_arg_t *);
int     uhci_td_walk_init(mdb_walk_state_t *);
int     uhci_td_walk_step(mdb_walk_state_t *);
int     uhci_qh_walk_init(mdb_walk_state_t *);
int     uhci_qh_walk_step(mdb_walk_state_t *);


/*
 * Callback for find_uhci_statep (called back from walk "softstate" in
 * find_uhci_statep).
 *
 * - uhci_instancep is the value of the current pointer in the array of soft
 * state instance pointers (see i_ddi_soft_state in ddi_impldefs.h)
```



```
 * - local_ss is a pointer to the copy of the i_ddi_soft_state in local space
 * - cb_arg is a pointer to the cb arg (an instance of state_find_data).
 *
 * For the current uchi_state_t*, see if the td address is in its pool.
 *
 * Returns WALK_NEXT on success (match not found yet), WALK_ERR on errors.
 *
 * WALK_DONE is returned, cb_data.found is set to TRUE, and
 * *cb_data.fic_uhci_statep is filled in with the contents of the state
 * struct in core. This forces the walk to terminate.
 */
typedef struct find_instance_struct {
        void            *fic_td_qh;     /* td/qh we want uhci instance for */
        boolean_t       fic_td_or_qh;   /* which one td_qh points to */
        boolean_t       fic_found;
        uhci_state_t    *fic_uhci_statep; /* buffer uhci_state's written into */
} find_instance_cb_t;
_____unchanged_portion_omitted_

/*
 * At each step, read a QH into our private storage, and then invoke
 * the callback function.  We terminate when we reach a QH, or
 * link_ptr is NULL.
 */
int
uhci_qh_walk_step(mdb_walk_state_t *wsp)
{
        int status;
        uhci_state_t    *uhcip = (uhci_state_t *)wsp->walk_arg;


        if (wsp->walk_addr == NULL)     /* Should never occur */
                return (WALK_DONE);

        if (mdb_vread(wsp->walk_data, sizeof (queue_head_t), wsp->walk_addr)
            == -1) {
                mdb_warn("failure reading qh at %p", wsp->walk_addr);
                return (WALK_DONE);
        }

        status = wsp->walk_callback(wsp->walk_addr, wsp->walk_data,
            wsp->walk_cbdata);

        /* Next QH. */
        wsp->walk_addr = ((queue_head_t *)wsp->walk_data)->link_ptr;


        /* Check if we're at the last element */
        if (wsp->walk_addr == NULL || wsp->walk_addr & HC_END_OF_LIST)  {
                return (WALK_DONE);
        }

        /* Make sure next element is a QH.  If a TD, stop.  */
        if (((((queue_head_t *)wsp->walk_data)->link_ptr) & HC_QUEUE_HEAD)
            != HC_QUEUE_HEAD)  {
        if (!  ((((queue_head_t *)wsp->walk_data)->link_ptr) & HC_QUEUE_HEAD)
            == HC_QUEUE_HEAD)  {
                return (WALK_DONE);
        }

        /* Strip terminate etc. bits.  */
        wsp->walk_addr &= QH_LINK_PTR_MASK;

        if (wsp->walk_addr == NULL)
                return (WALK_DONE);
```

```
 596         /*
 597          * Convert link_ptr paddr to vaddr
 598          * Note: uhcip needed by QH_VADDR macro
 599          */
 600         wsp->walk_addr = (uintptr_t)QH_VADDR(wsp->walk_addr);

 602         return (status);
 603 }
```

_____***unchanged_portion_omitted_***