```
**********************************************************
  146669 Wed Jan 23 13:19:01 2013
new/usr/src/cmd/mdb/common/mdb/mdb_proc.c
XXX AVX procfs
**********************************************************
```

```
______unchanged_portion_omitted_

4723 #ifdef  __sparc

4723 /*ARGSUSED*/
4724 static int
4725 pt_lwp_getxregs(mdb_tgt_t *t, void *tap, mdb_tgt_tid_t tid, prxregset_t *xregs)
4726 {
4727         if (t->t_pshandle != NULL) {
4728                 return (ptl_err(Plwp_getxregs(t->t_pshandle,
4729                     (lwpid_t)tid, xregs)));
4730         }
4731         return (set_errno(EMDB_NOPROC));
4732 }
______unchanged_portion_omitted_

4748 #endif  /* __sparc */

4746 /*ARGSUSED*/
4747 static int
4748 pt_lwp_getfpregs(mdb_tgt_t *t, void *tap, mdb_tgt_tid_t tid,
4749     prfpregset_t *fpregs)
4750 {
4751         if (t->t_pshandle != NULL) {
4752                 return (ptl_err(Plwp_getfpregs(t->t_pshandle,
4753                     (lwpid_t)tid, fpregs)));
4754         }
4755         return (set_errno(EMDB_NOPROC));
4756 }
______unchanged_portion_omitted_

4770 static const pt_ptl_ops_t proc_lwp_ops = {
4771         (int (*)()) mdb_tgt_nop,
4772         (void (*)()) mdb_tgt_nop,
4773         pt_lwp_tid,
4774         pt_lwp_iter,
4775         pt_lwp_getregs,
4776         pt_lwp_setregs,
4781 #ifdef __sparc
4777         pt_lwp_getxregs,
4778         pt_lwp_setxregs,
4784 #endif
4779         pt_lwp_getfpregs,
4780         pt_lwp_setfpregs
4781 };
______unchanged_portion_omitted_
```

```
**********************************************************
    8639 Wed Jan 23 13:19:01 2013
new/usr/src/cmd/mdb/common/mdb/mdb_proc.h
XXX AVX procfs
**********************************************************
```

```
/*
 * 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.
 */

#ifndef _MDB_PROC_H
#define _MDB_PROC_H

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

#include <mdb/mdb_target_impl.h>
#include <mdb/mdb_io_impl.h>
#include <mdb/mdb_addrvec.h>
#include <mdb/mdb_modapi.h>
#include <mdb/mdb_gelf.h>
#include <mdb/mdb_tdb.h>

#include <sys/param.h>
#include <libproc.h>

#ifdef  __cplusplus
extern "C" {
#endif

#ifdef _MDB

/*
 * The proc target must provide support for examining multi-threaded processes
 * that use the raw LWP interface, as well as those that use either of the
 * existing libthread.so implementations.  We must also support multiple active
 * instances of the proc target, as well as the notion that a clean process
 * can dlopen() libthread after startup, at which point we need to switch to
 * using libthread_db interfaces to properly debug it.  To satisfy these
 * constraints, we declare an ops vector of functions for obtaining the
 * register sets of each thread.  The proc target will define two versions
 * of this vector, one for the LWP mode and one for the libthread_db mode,
 * and then switch the ops vector pointer as appropriate during debugging.
 * The macros defined below expand to calls to the appropriate entry point.
 */
typedef struct pt_ptl_ops {
```



```
        int (*ptl_ctor)(mdb_tgt_t *);
        void (*ptl_dtor)(mdb_tgt_t *, void *);
        mdb_tgt_tid_t (*ptl_tid)(mdb_tgt_t *, void *);
        int (*ptl_iter)(mdb_tgt_t *, void *, mdb_addrvec_t *);
        int (*ptl_getregs)(mdb_tgt_t *, void *, mdb_tgt_tid_t, prgregset_t);
        int (*ptl_setregs)(mdb_tgt_t *, void *, mdb_tgt_tid_t, prgregset_t);
#ifdef  __sparc
        int (*ptl_getxregs)(mdb_tgt_t *, void *, mdb_tgt_tid_t,
            prxregset_t *);
        int (*ptl_setxregs)(mdb_tgt_t *, void *, mdb_tgt_tid_t,
            const prxregset_t *);
#endif
        int (*ptl_getfpregs)(mdb_tgt_t *, void *, mdb_tgt_tid_t,
            prfpregset_t *);
        int (*ptl_setfpregs)(mdb_tgt_t *, void *, mdb_tgt_tid_t,
            const prfpregset_t *);
} pt_ptl_ops_t;

#define PTL_CTOR(t) \
        (((pt_data_t *)(t)->t_data)->p_ptl_ops->ptl_ctor(t))

#define PTL_DTOR(t) \
        (((pt_data_t *)(t)->t_data)->p_ptl_ops->ptl_dtor((t), \
        ((pt_data_t *)((t)->t_data))->p_ptl_hdl))

#define PTL_TID(t) \
        (((pt_data_t *)((t)->t_data))->p_ptl_ops->ptl_tid((t), \
        ((pt_data_t *)(t)->t_data)->p_ptl_hdl))

#define PTL_ITER(t, ap) \
        (((pt_data_t *)(t)->t_data)->p_ptl_ops->ptl_iter((t), \
        ((pt_data_t *)((t)->t_data))->p_ptl_hdl, (ap)))

#define PTL_GETREGS(t, tid, gregs) \
        (((pt_data_t *)((t)->t_data))->p_ptl_ops->ptl_getregs((t), \
        ((pt_data_t *)((t)->t_data))->p_ptl_hdl, (tid), (gregs)))

#define PTL_SETREGS(t, tid, gregs) \
        (((pt_data_t *)((t)->t_data))->p_ptl_ops->ptl_setregs((t), \
        ((pt_data_t *)((t)->t_data))->p_ptl_hdl, (tid), (gregs)))

#ifdef  __sparc

#define PTL_GETXREGS(t, tid, xregs) \
        (((pt_data_t *)((t)->t_data))->p_ptl_ops->ptl_getxregs((t), \
        ((pt_data_t *)((t)->t_data))->p_ptl_hdl, (tid), (xregs)))

#define PTL_SETXREGS(t, tid, xregs) \
        (((pt_data_t *)((t)->t_data))->p_ptl_ops->ptl_setxregs((t), \
        ((pt_data_t *)((t)->t_data))->p_ptl_hdl, (tid), (xregs)))

#endif  /* __sparc */

#define PTL_GETFPREGS(t, tid, fpregs) \
        (((pt_data_t *)((t)->t_data))->p_ptl_ops->ptl_getfpregs((t), \
        ((pt_data_t *)((t)->t_data))->p_ptl_hdl, (tid), (fpregs)))

#define PTL_SETFPREGS(t, tid, fpregs) \
        (((pt_data_t *)((t)->t_data))->p_ptl_ops->ptl_setfpregs((t), \
        ((pt_data_t *)((t)->t_data))->p_ptl_hdl, (tid), (fpregs)))

/*
 * When we are following children and a vfork(2) occurs, we append the libproc
 * handle for the parent to a list of vfork parents.  We need to keep track of
 * this handle so that when the child subsequently execs or dies, we clear out
 * our breakpoints before releasing the parent.
```

```
 120  */
 121 typedef struct pt_vforkp {
 122         mdb_list_t p_list;                      /* List forward/back pointers */
 123         struct ps_prochandle *p_pshandle;       /* libproc handle */
 124 } pt_vforkp_t;
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   15428 Wed Jan 23 13:19:02 2013
new/usr/src/cmd/mdb/intel/mdb/proc_amd64dep.c
XXX AVX procfs
**********************************************************
```

```
/*
 * 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"

/*
 * User Process Target Intel 32-bit component
 *
 * This file provides the ISA-dependent portion of the user process target.
 * For more details on the implementation refer to mdb_proc.c.
 */

#include <mdb/mdb_proc.h>
#include <mdb/mdb_kreg.h>
#include <mdb/mdb_err.h>
#include <mdb/mdb_amd64util.h>
#include <mdb/mdb.h>

#include <sys/frame.h>
#include <libproc.h>
#include <sys/fp.h>
#include <ieeefp.h>

const mdb_tgt_regdesc_t pt_regdesc[] = {
        { "r15",        REG_R15,        MDB_TGT_R_EXPORT },
        { "r14",        REG_R14,        MDB_TGT_R_EXPORT },
        { "r13",        REG_R13,        MDB_TGT_R_EXPORT },
        { "r12",        REG_R12,        MDB_TGT_R_EXPORT },
        { "r11",        REG_R11,        MDB_TGT_R_EXPORT },
        { "r10",        REG_R10,        MDB_TGT_R_EXPORT },
        { "r9",         REG_R9,         MDB_TGT_R_EXPORT },
        { "r8",         REG_R8,         MDB_TGT_R_EXPORT },
        { "rdi",        REG_RDI,        MDB_TGT_R_EXPORT },
        { "rsi",        REG_RSI,        MDB_TGT_R_EXPORT },
        { "rbp",        REG_RBP,        MDB_TGT_R_EXPORT },
        { "rbx",        REG_RBX,        MDB_TGT_R_EXPORT },
        { "rdx",        REG_RDX,        MDB_TGT_R_EXPORT },
        { "rcx",        REG_RCX,        MDB_TGT_R_EXPORT },
```



```
        { "rax",        REG_RAX,        MDB_TGT_R_EXPORT },
        { "trapno",     REG_TRAPNO,     MDB_TGT_R_EXPORT },
        { "err",        REG_ERR,        MDB_TGT_R_EXPORT },
        { "rip",        REG_RIP,        MDB_TGT_R_EXPORT },
        { "cs",         REG_CS,         MDB_TGT_R_EXPORT },
        { "rflags",     REG_RFL,        MDB_TGT_R_EXPORT },
        { "rsp",        REG_RSP,        MDB_TGT_R_EXPORT },
        { "ss",         REG_SS,         MDB_TGT_R_EXPORT },
        { "fs",         REG_FS,         MDB_TGT_R_EXPORT },
        { "gs",         REG_GS,         MDB_TGT_R_EXPORT },
        { "es",         REG_ES,         MDB_TGT_R_EXPORT },
        { "ds",         REG_DS,         MDB_TGT_R_EXPORT },
        { "fsbase",     REG_FSBASE,     MDB_TGT_R_EXPORT },
        { "gsbase",     REG_GSBASE,     MDB_TGT_R_EXPORT },
        { NULL, 0, 0 }
};
```

_____*unchanged_portion_omitted_*

```
/*ARGSUSED*/
int
pt_regs(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
{
        mdb_tgt_t *t = mdb.m_target;
        mdb_tgt_tid_t tid;
        prgregset_t grs;
        prgreg_t rflags;

        if (argc != 0)
                return (DCMD_USAGE);

        if (t->t_pshandle == NULL || Pstate(t->t_pshandle) == PS_UNDEAD) {
                mdb_warn("no process active\n");
                return (DCMD_ERR);
        }

        if (Pstate(t->t_pshandle) == PS_LOST) {
                mdb_warn("debugger has lost control of process\n");
                return (DCMD_ERR);
        }

        if (flags & DCMD_ADDRSPEC)
                tid = (mdb_tgt_tid_t)addr;
        else
                tid = PTL_TID(t);

        if (PTL_GETREGS(t, tid, grs) != 0) {
                mdb_warn("failed to get current register set");
                return (DCMD_ERR);
        }

        rflags = grs[REG_RFL];

        mdb_printf("%%rax = 0x%0?p\t%%r8  = 0x%0?p\n",
            grs[REG_RAX], grs[REG_R8]);
        mdb_printf("%%rbx = 0x%0?p\t%%r9  = 0x%0?p\n",
            grs[REG_RBX], grs[REG_R9]);
        mdb_printf("%%rcx = 0x%0?p\t%%r10 = 0x%0?p\n",
            grs[REG_RCX], grs[REG_R10]);
        mdb_printf("%%rdx = 0x%0?p\t%%r11 = 0x%0?p\n",
            grs[REG_RDX], grs[REG_R11]);
        mdb_printf("%%rsi = 0x%0?p\t%%r12 = 0x%0?p\n",
            grs[REG_RSI], grs[REG_R12]);
        mdb_printf("%%rdi = 0x%0?p\t%%r13 = 0x%0?p\n",
            grs[REG_RDI], grs[REG_R13]);
        mdb_printf("         %?s\t%%r14 = 0x%0?p\n",
            "", grs[REG_R14]);
```

```
 140         mdb_printf("         %?s\t%%r15 = 0x%0?p\n",
 141             "", grs[REG_R15]);

 143         mdb_printf("\n");

 145         mdb_printf("%%cs = 0x%04x\t%%fs = 0x%04x\t%%gs = 0x%04x\n",
 146             grs[REG_CS], grs[REG_FS], grs[REG_GS]);
 147         mdb_printf("%%ds = 0x%04x\t%%es = 0x%04x\t%%ss = 0x%04x\n",
 148             grs[REG_DS], grs[REG_ES], grs[REG_SS]);

 150         mdb_printf("\n");

 152         mdb_printf("%%rip = 0x%0?p %A\n", grs[REG_RIP], grs[REG_RIP]);
 153         mdb_printf("%%rbp = 0x%0?p\n", grs[REG_RBP], grs[REG_RBP]);
 154         mdb_printf("%%rsp = 0x%0?p\n", grs[REG_RSP], grs[REG_RSP]);

 156         mdb_printf("\n");

 158         mdb_printf("%%rflags = 0x%08x\n", rflags);

 160         mdb_printf("  id=%u vip=%u vif=%u ac=%u vm=%u rf=%u nt=%u iopl=0x%x\n",
 161             (rflags & KREG_EFLAGS_ID_MASK) >> KREG_EFLAGS_ID_SHIFT,
 162             (rflags & KREG_EFLAGS_VIP_MASK) >> KREG_EFLAGS_VIP_SHIFT,
 163             (rflags & KREG_EFLAGS_VIF_MASK) >> KREG_EFLAGS_VIF_SHIFT,
 164             (rflags & KREG_EFLAGS_AC_MASK) >> KREG_EFLAGS_AC_SHIFT,
 165             (rflags & KREG_EFLAGS_VM_MASK) >> KREG_EFLAGS_VM_SHIFT,
 166             (rflags & KREG_EFLAGS_RF_MASK) >> KREG_EFLAGS_RF_SHIFT,
 167             (rflags & KREG_EFLAGS_NT_MASK) >> KREG_EFLAGS_NT_SHIFT,
 168             (rflags & KREG_EFLAGS_IOPL_MASK) >> KREG_EFLAGS_IOPL_SHIFT);

 170         mdb_printf("  status=<%s,%s,%s,%s,%s,%s,%s,%s,%s>\n",
 171             (rflags & KREG_EFLAGS_OF_MASK) ? "OF" : "of",
 172             (rflags & KREG_EFLAGS_DF_MASK) ? "DF" : "df",
 173             (rflags & KREG_EFLAGS_IF_MASK) ? "IF" : "if",
 174             (rflags & KREG_EFLAGS_TF_MASK) ? "TF" : "tf",
 175             (rflags & KREG_EFLAGS_SF_MASK) ? "SF" : "sf",
 176             (rflags & KREG_EFLAGS_ZF_MASK) ? "ZF" : "zf",
 177             (rflags & KREG_EFLAGS_AF_MASK) ? "AF" : "af",
 178             (rflags & KREG_EFLAGS_PF_MASK) ? "PF" : "pf",
 179             (rflags & KREG_EFLAGS_CF_MASK) ? "CF" : "cf");

 181         mdb_printf("\n");

 183         mdb_printf("%%gsbase = 0x%0?p\n", grs[REG_GSBASE]);
 184         mdb_printf("%%fsbase = 0x%0?p\n", grs[REG_FSBASE]);
 185         mdb_printf("%%trapno = 0x%x\n", grs[REG_TRAPNO]);
 186         mdb_printf("   %%err = 0x%x\n", grs[REG_ERR]);

 188         return (DCMD_OK);
 190         return (set_errno(ENOTSUP));
 189 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   80862 Wed Jan 23 13:19:02 2013
new/usr/src/lib/libc_db/common/thread_db.c
XXX AVX procfs
**********************************************************
______unchanged_portion_omitted_

1957 /*
1958  * Get the size of the extra state register set for this architecture.
1959  * Currently unused by dbx.
1960  */
1961 #pragma weak td_thr_getxregsize = __td_thr_getxregsize
1962 /* ARGSUSED */
1963 td_err_e
1964 __td_thr_getxregsize(td_thrhandle_t *th_p, int *xregsize)
1965 {
1966 #if defined(__sparc)
1966         struct ps_prochandle *ph_p;
1967         td_err_e return_val;

1969         if ((ph_p = ph_lock_th(th_p, &return_val)) == NULL)
1970                 return (return_val);
1971         if (ps_pstop(ph_p) != PS_OK) {
1972                 ph_unlock(th_p->th_ta_p);
1973                 return (TD_DBERR);
1974         }

1976         if (ps_lgetxregsize(ph_p, thr_to_lwpid(th_p), xregsize) != PS_OK)
1977                 return_val = TD_DBERR;

1979         (void) ps_pcontinue(ph_p);
1980         ph_unlock(th_p->th_ta_p);
1981         return (return_val);
1983 #else   /* __sparc */
1984         return (TD_NOXREGS);
1985 #endif  /* __sparc */
1982 }

1984 /*
1985  * Get a thread's extra state register set.
1986  */
1987 #pragma weak td_thr_getxregs = __td_thr_getxregs
1988 /* ARGSUSED */
1989 td_err_e
1990 __td_thr_getxregs(td_thrhandle_t *th_p, void *xregset)
1991 {
1996 #if defined(__sparc)
1992         struct ps_prochandle *ph_p;
1993         td_err_e return_val;

1995         if ((ph_p = ph_lock_th(th_p, &return_val)) == NULL)
1996                 return (return_val);
1997         if (ps_pstop(ph_p) != PS_OK) {
1998                 ph_unlock(th_p->th_ta_p);
1999                 return (TD_DBERR);
2000         }

2002         if (ps_lgetxregs(ph_p, thr_to_lwpid(th_p), (caddr_t)xregset) != PS_OK)
2003                 return_val = TD_DBERR;

2005         (void) ps_pcontinue(ph_p);
2006         ph_unlock(th_p->th_ta_p);
2007         return (return_val);
2013 #else   /* __sparc */
2014         return (TD_NOXREGS);
2015 #endif  /* __sparc */
```



```
2008 }

2010 /*
2011  * Set a thread's extra state register set.
2012  */
2013 #pragma weak td_thr_setxregs = __td_thr_setxregs
2014 /* ARGSUSED */
2015 td_err_e
2016 __td_thr_setxregs(td_thrhandle_t *th_p, const void *xregset)
2017 {
2026 #if defined(__sparc)
2018         struct ps_prochandle *ph_p;
2019         td_err_e return_val;

2021         if ((ph_p = ph_lock_th(th_p, &return_val)) == NULL)
2022                 return (return_val);
2023         if (ps_pstop(ph_p) != PS_OK) {
2024                 ph_unlock(th_p->th_ta_p);
2025                 return (TD_DBERR);
2026         }

2028         if (ps_lsetxregs(ph_p, thr_to_lwpid(th_p), (caddr_t)xregset) != PS_OK)
2029                 return_val = TD_DBERR;

2031         (void) ps_pcontinue(ph_p);
2032         ph_unlock(th_p->th_ta_p);
2033         return (return_val);
2043 #else   /* __sparc */
2044         return (TD_NOXREGS);
2045 #endif  /* __sparc */
2034 }
______unchanged_portion_omitted_
```

****
12690 Wed Jan 23 13:19:03 2013
new/usr/src/lib/libproc/common/Pcontrol.h
XXX AVX procfs
****
_____***unchanged_portion_omitted_***

```
 130 typedef struct lwp_info {       /* per-lwp information from core file */
 131         plist_t lwp_list;       /* linked list */
 132         lwpid_t lwp_id;         /* lwp identifier */
 133         lwpsinfo_t lwp_psinfo;  /* /proc/<pid>/lwp/<lwpid>/lwpsinfo data */
 134         lwpstatus_t lwp_status; /* /proc/<pid>/lwp/<lwpid>/lwpstatus data */
 135         prxregset_t *lwp_xregs; /* /proc/<pid>/lwp/<lwpid>/xregs data */
 136 #endif /* ! codereview */
 137 #if defined(sparc) || defined(__sparc)
 138         gwindows_t *lwp_gwins;  /* /proc/<pid>/lwp/<lwpid>/gwindows data */
 135         prxregset_t *lwp_xregs; /* /proc/<pid>/lwp/<lwpid>/xregs data */
 139         int64_t *lwp_asrs;      /* /proc/<pid>/lwp/<lwpid>/asrs data */
 140 #endif
 141 } lwp_info_t;
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   58499 Wed Jan 23 13:19:03 2013
new/usr/src/lib/libproc/common/Pcore.c
XXX AVX procfs
**********************************************************
______unchanged_portion_omitted_

590 #ifdef __sparc
590 static int
591 note_xreg(struct ps_prochandle *P, size_t nbytes)
592 {
593         lwp_info_t *lwp = P->core->core_lwp;
594         size_t xbytes = sizeof (prxregset_t);
595         prxregset_t *xregs;

597         if (lwp == NULL || lwp->lwp_xregs != NULL || nbytes < xbytes)
598                 return (0);     /* No lwp yet, already seen, or bad size */

600         if ((xregs = malloc(xbytes)) == NULL)
601                 return (-1);
602 #ifdef __sparc

603         if (read(P->asfd, xregs, xbytes) != xbytes) {
604 #else
605         panic("port me");
606 #endif
607 #endif /* ! codereview */
608                 dprintf("Pgrab_core: failed to read NT_PRXREG\n");
609                 free(xregs);
610                 return (-1);
611         }

612         lwp->lwp_xregs = xregs;
613         return (0);
614 }

616 #ifdef __sparc
617 #endif /* ! codereview */
618 static int
619 note_gwindows(struct ps_prochandle *P, size_t nbytes)
620 {
621         lwp_info_t *lwp = P->core->core_lwp;

623         if (lwp == NULL || lwp->lwp_gwins != NULL || nbytes == 0)
624                 return (0);     /* No lwp yet or already seen or no data */

626         if ((lwp->lwp_gwins = malloc(sizeof (gwindows_t))) == NULL)
627                 return (-1);

629         /*
630          * Since the amount of gwindows data varies with how many windows were
631          * actually saved, we just read up to the minimum of the note size
632          * and the size of the gwindows_t type.  It doesn't matter if the read
633          * fails since we have to zero out gwindows first anyway.
634          */
635 #ifdef _LP64
636         if (P->core->core_dmodel == PR_MODEL_ILP32) {
637                 gwindows32_t g32;

639                 (void) memset(&g32, 0, sizeof (g32));
640                 (void) read(P->asfd, &g32, MIN(nbytes, sizeof (g32)));
641                 gwindows_32_to_n(&g32, lwp->lwp_gwins);

643         } else {
644 #endif
645                 (void) memset(lwp->lwp_gwins, 0, sizeof (gwindows_t));
```



```
646                 (void) read(P->asfd, lwp->lwp_gwins,
647                     MIN(nbytes, sizeof (gwindows_t)));
648 #ifdef _LP64
649         }
650 #endif
651         return (0);
652 }

654 #ifdef __sparcv9
655 static int
656 note_asrs(struct ps_prochandle *P, size_t nbytes)
657 {
658         lwp_info_t *lwp = P->core->core_lwp;
659         int64_t *asrs;

661         if (lwp == NULL || lwp->lwp_asrs != NULL || nbytes < sizeof (asrset_t))
662                 return (0);     /* No lwp yet, already seen, or bad size */

664         if ((asrs = malloc(sizeof (asrset_t))) == NULL)
665                 return (-1);

667         if (read(P->asfd, asrs, sizeof (asrset_t)) != sizeof (asrset_t)) {
668                 dprintf("Pgrab_core: failed to read NT_ASRS\n");
669                 free(asrs);
670                 return (-1);
671         }

673         lwp->lwp_asrs = asrs;
674         return (0);
675 }
676 #endif  /* __sparcv9 */
677 #endif  /* __sparc */

679 /*ARGSUSED*/
680 static int
681 note_notsup(struct ps_prochandle *P, size_t nbytes)
682 {
683         dprintf("skipping unsupported note type\n");
684         return (0);
685 }

687 /*
688  * Populate a table of function pointers indexed by Note type with our
689  * functions to process each type of core file note:
690  */
691 static int (*nhdlrs[])(struct ps_prochandle *, size_t) = {
692         note_notsup,            /*  0   unassigned              */
693         note_notsup,            /*  1   NT_PRSTATUS (old)       */
694         note_notsup,            /*  2   NT_PRFPREG (old)        */
695         note_notsup,            /*  3   NT_PRPSINFO (old)       */
610 #ifdef __sparc
696         note_xreg,              /*  4   NT_PRXREG               */
612 #else
613         note_notsup,            /*  4   NT_PRXREG               */
614 #endif
697         note_platform,          /*  5   NT_PLATFORM             */
698         note_auxv,              /*  6   NT_AUXV                 */
699 #ifdef __sparc
700         note_gwindows,          /*  7   NT_GWINDOWS             */
701 #ifdef __sparcv9
702         note_asrs,              /*  8   NT_ASRS                 */
703 #else
704         note_notsup,            /*  8   NT_ASRS                 */
705 #endif
706 #else
707         note_notsup,            /*  7   NT_GWINDOWS             */
```

```
 708         note_notsup,            /*  8   NT_ASRS                 */
 709 #endif
 710 #if defined(__i386) || defined(__amd64)
 711         note_ldt,               /*  9   NT_LDT                  */
 712 #else
 713         note_notsup,            /*  9   NT_LDT                  */
 714 #endif
 715         note_pstatus,           /* 10   NT_PSTATUS              */
 716         note_notsup,            /* 11   unassigned              */
 717         note_notsup,            /* 12   unassigned              */
 718         note_psinfo,            /* 13   NT_PSINFO               */
 719         note_cred,              /* 14   NT_PRCRED               */
 720         note_utsname,           /* 15   NT_UTSNAME              */
 721         note_lwpstatus,         /* 16   NT_LWPSTATUS            */
 722         note_lwpsinfo,          /* 17   NT_LWPSINFO             */
 723         note_priv,              /* 18   NT_PRPRIV               */
 724         note_priv_info,         /* 19   NT_PRPRIVINFO           */
 725         note_content,           /* 20   NT_CONTENT              */
 726         note_zonename,          /* 21   NT_ZONENAME             */
 727         note_fdinfo,            /* 22   NT_FDINFO               */
 728 };
______unchanged_portion_omitted_
```

```
**********************************************************
   10935 Wed Jan 23 13:19:04 2013
new/usr/src/lib/libproc/common/Plwpregs.c
XXX AVX procfs
**********************************************************
```

```
  1 /*
  2  * CDDL HEADER START
  3  *
  4  * The contents of this file are subject to the terms of the
  5  * Common Development and Distribution License (the "License").
  6  * You may not use this file except in compliance with the License.
  7  *
  8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
  9  * or http://www.opensolaris.org/os/licensing.
 10  * See the License for the specific language governing permissions
 11  * and limitations under the License.
 12  *
 13  * When distributing Covered Code, include this CDDL HEADER in each
 14  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 15  * If applicable, add the following below this CDDL HEADER, with the
 16  * fields enclosed by brackets "[]" replaced with your own identifying
 17  * information: Portions Copyright [yyyy] [name of copyright owner]
 18  *
 19  * CDDL HEADER END
 20  */
 21 /*
 22  * Copyright 2006 Sun Microsystems, Inc.  All rights reserved.
 23  * Use is subject to license terms.
 24  */

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

 26 #include <sys/types.h>
 27 #include <sys/uio.h>
 28 #include <string.h>
 29 #include <errno.h>
 30 #include <limits.h>

 32 #include "Pcontrol.h"
 33 #include "P32ton.h"

 35 /*
 36  * This file implements the routines to read and write per-lwp register
 37  * information from either a live process or core file opened with libproc.
 38  * We build up a few common routines for reading and writing register
 39  * information, and then the public functions are all trivial calls to these.
 40  */

 42 /*
 43  * Utility function to return a pointer to the structure of cached information
 44  * about an lwp in the core file, given its lwpid.
 45  */
 46 static lwp_info_t *
 47 getlwpcore(struct ps_prochandle *P, lwpid_t lwpid)
 48 {
 49         lwp_info_t *lwp = list_next(&P->core->core_lwp_head);
 50         uint_t i;

 52         for (i = 0; i < P->core->core_nlwp; i++, lwp = list_next(lwp)) {
 53                 if (lwp->lwp_id == lwpid)
 54                         return (lwp);
 55         }

 57         errno = EINVAL;
 58         return (NULL);
 59 }
```

_____*unchanged_portion_omitted_*


```
222 #if defined(sparc) || defined(__sparc)
220 int
221 Plwp_getxregs(struct ps_prochandle *P, lwpid_t lwpid, prxregset_t *xregs)
222 {
223         lwp_info_t *lwp;

225         if (P->state == PS_IDLE) {
226                 errno = ENODATA;
227                 return (-1);
228         }

230         if (P->state != PS_DEAD) {
231                 if (P->state != PS_STOP) {
232                         errno = EBUSY;
233                         return (-1);
234                 }

236                 return (getlwpfile(P, lwpid, "xregs",
237                     xregs, sizeof (prxregset_t)));
238         }

240         if ((lwp = getlwpcore(P, lwpid)) != NULL && lwp->lwp_xregs != NULL) {
241                 (void) memcpy(xregs, lwp->lwp_xregs, sizeof (prxregset_t));
242                 return (0);
243         }

245         if (lwp != NULL)
246                 errno = ENODATA;
247         return (-1);
248 }
```

_____*unchanged_portion_omitted_*

```
256 #if defined(sparc) || defined(__sparc)
257 #endif /* ! codereview */
258 int
259 Plwp_getgwindows(struct ps_prochandle *P, lwpid_t lwpid, gwindows_t *gwins)
260 {
261         lwp_info_t *lwp;

263         if (P->state == PS_IDLE) {
264                 errno = ENODATA;
265                 return (-1);
266         }

268         if (P->state != PS_DEAD) {
269                 if (P->state != PS_STOP) {
270                         errno = EBUSY;
271                         return (-1);
272                 }

274                 return (getlwpfile(P, lwpid, "gwindows",
275                     gwins, sizeof (gwindows_t)));
276         }

278         if ((lwp = getlwpcore(P, lwpid)) != NULL && lwp->lwp_gwins != NULL) {
279                 *gwins = *lwp->lwp_gwins;
280                 return (0);
281         }

283         if (lwp != NULL)
284                 errno = ENODATA;
285         return (-1);
286 }

288 #if defined(__sparcv9)
```

```
289 int
290 Plwp_getasrs(struct ps_prochandle *P, lwpid_t lwpid, asrset_t asrs)
291 {
292         lwp_info_t *lwp;

294         if (P->state == PS_IDLE) {
295                 errno = ENODATA;
296                 return (-1);
297         }

299         if (P->state != PS_DEAD) {
300                 if (P->state != PS_STOP) {
301                         errno = EBUSY;
302                         return (-1);
303                 }

305                 return (getlwpfile(P, lwpid, "asrs", asrs, sizeof (asrset_t)));
306         }

308         if ((lwp = getlwpcore(P, lwpid)) != NULL && lwp->lwp_asrs != NULL) {
309                 (void) memcpy(asrs, lwp->lwp_asrs, sizeof (asrset_t));
310                 return (0);
311         }

313         if (lwp != NULL)
314                 errno = ENODATA;
315         return (-1);

317 }

319 int
320 Plwp_setasrs(struct ps_prochandle *P, lwpid_t lwpid, const asrset_t asrs)
321 {
322         return (setlwpregs(P, lwpid, PCSASRS, asrs, sizeof (asrset_t)));
323 }
324 #endif  /* __sparcv9 */
325 #endif  /* __sparc */

327 int
328 Plwp_getpsinfo(struct ps_prochandle *P, lwpid_t lwpid, lwpsinfo_t *lps)
329 {
330         lwp_info_t *lwp;

332         if (P->state == PS_IDLE) {
333                 errno = ENODATA;
334                 return (-1);
335         }

337         if (P->state != PS_DEAD) {
338                 return (getlwpfile(P, lwpid, "lwpsinfo",
339                     lps, sizeof (lwpsinfo_t)));
340         }

342         if ((lwp = getlwpcore(P, lwpid)) != NULL) {
343                 (void) memcpy(lps, &lwp->lwp_psinfo, sizeof (lwpsinfo_t));
344                 return (0);
345         }

347         return (-1);
348 }

350 int
351 Plwp_stack(struct ps_prochandle *P, lwpid_t lwpid, stack_t *stkp)
352 {
353         uintptr_t addr;
```



```
355         if (P->state == PS_IDLE) {
356                 errno = ENODATA;
357                 return (-1);
358         }

360         if (P->state != PS_DEAD) {
361                 lwpstatus_t ls;
362                 if (getlwpfile(P, lwpid, "lwpstatus", &ls, sizeof (ls)) != 0)
363                         return (-1);
364                 addr = ls.pr_ustack;
365         } else {
366                 lwp_info_t *lwp;
367                 if ((lwp = getlwpcore(P, lwpid)) == NULL)
368                         return (-1);
369                 addr = lwp->lwp_status.pr_ustack;
370         }


373         if (P->status.pr_dmodel == PR_MODEL_NATIVE) {
374                 if (Pread(P, stkp, sizeof (*stkp), addr) != sizeof (*stkp))
375                         return (-1);
376 #ifdef _LP64
377         } else {
378                 stack32_t stk32;

380                 if (Pread(P, &stk32, sizeof (stk32), addr) != sizeof (stk32))
381                         return (-1);

383                 stack_32_to_n(&stk32, stkp);
384 #endif
385         }

387         return (0);
388 }

390 int
391 Plwp_main_stack(struct ps_prochandle *P, lwpid_t lwpid, stack_t *stkp)
392 {
393         uintptr_t addr;
394         lwpstatus_t ls;

396         if (P->state == PS_IDLE) {
397                 errno = ENODATA;
398                 return (-1);
399         }

401         if (P->state != PS_DEAD) {
402                 if (getlwpfile(P, lwpid, "lwpstatus", &ls, sizeof (ls)) != 0)
403                         return (-1);
404         } else {
405                 lwp_info_t *lwp;
406                 if ((lwp = getlwpcore(P, lwpid)) == NULL)
407                         return (-1);
408                 ls = lwp->lwp_status;
409         }

411         addr = ls.pr_ustack;

413         /*
414          * Read out the current stack; if the SS_ONSTACK flag is set then
415          * this LWP is operating on the alternate signal stack. We can
416          * recover the original stack from pr_oldcontext.
417          */
418         if (P->status.pr_dmodel == PR_MODEL_NATIVE) {
419                 if (Pread(P, stkp, sizeof (*stkp), addr) != sizeof (*stkp))
420                         return (-1);
```

```
422                 if (stkp->ss_flags & SS_ONSTACK)
423                         goto on_altstack;
424 #ifdef _LP64
425         } else {
426                 stack32_t stk32;
427
428                 if (Pread(P, &stk32, sizeof (stk32), addr) != sizeof (stk32))
429                         return (-1);
430
431                 if (stk32.ss_flags & SS_ONSTACK)
432                         goto on_altstack;
433
434                 stack_32_to_n(&stk32, stkp);
435 #endif
436         }
437
438         return (0);
439
440 on_altstack:
441
442         if (P->status.pr_dmodel == PR_MODEL_NATIVE) {
443                 ucontext_t *ctxp = (void *)ls.pr_oldcontext;
444
445                 if (Pread(P, stkp, sizeof (*stkp),
446                     (uintptr_t)&ctxp->uc_stack) != sizeof (*stkp))
447                         return (-1);
448 #ifdef _LP64
449         } else {
450                 ucontext32_t *ctxp = (void *)ls.pr_oldcontext;
451                 stack32_t stk32;
452
453                 if (Pread(P, &stk32, sizeof (stk32),
454                     (uintptr_t)&ctxp->uc_stack) != sizeof (stk32))
455                         return (-1);
456
457                 stack_32_to_n(&stk32, stkp);
458 #endif
459         }
460
461         return (0);
462 }
463
464 int
465 Plwp_alt_stack(struct ps_prochandle *P, lwpid_t lwpid, stack_t *stkp)
466 {
467         if (P->state == PS_IDLE) {
468                 errno = ENODATA;
469                 return (-1);
470         }
471
472         if (P->state != PS_DEAD) {
473                 lwpstatus_t ls;
474
475                 if (getlwpfile(P, lwpid, "lwpstatus", &ls, sizeof (ls)) != 0)
476                         return (-1);
477
478                 if (ls.pr_altstack.ss_flags & SS_DISABLE) {
479                         errno = ENODATA;
480                         return (-1);
481                 }
482
483                 *stkp = ls.pr_altstack;
484         } else {
485                 lwp_info_t *lwp;
```



```
486
487                 if ((lwp = getlwpcore(P, lwpid)) == NULL)
488                         return (-1);
489
490                 if (lwp->lwp_status.pr_altstack.ss_flags & SS_DISABLE) {
491                         errno = ENODATA;
492                         return (-1);
493                 }
494
495                 *stkp = lwp->lwp_status.pr_altstack;
496         }
497
498         return (0);
499 }
```

```
**********************************************************
    8963 Wed Jan 23 13:19:04 2013
new/usr/src/lib/libproc/common/Pservice.c
XXX AVX procfs
**********************************************************
```

```
/*
 * 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.
 */

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

#include <stdarg.h>
#include <string.h>
#include "Pcontrol.h"

/*
 * This file implements the process services declared in <proc_service.h>.
 * This enables libproc to be used in conjunction with libc_db and
 * librtld_db.  As most of these facilities are already provided by
 * (more elegant) interfaces in <libproc.h>, we can just call those.
 *
 * NOTE: We explicitly do *not* implement the functions ps_kill() and
 * ps_lrolltoaddr() in this library.  The very existence of these functions
 * causes libc_db to create an "agent thread" in the target process.
 * The only way to turn off this behavior is to omit these functions.
 */

#pragma weak ps_pdread = ps_pread
#pragma weak ps_ptread = ps_pread
#pragma weak ps_pdwrite = ps_pwrite
#pragma weak ps_ptwrite = ps_pwrite

ps_err_e
ps_pdmodel(struct ps_prochandle *P, int *modelp)
{
        *modelp = P->status.pr_dmodel;
        return (PS_OK);
}
```

_____unchanged_portion_omitted_

```
#if defined(sparc) || defined(__sparc)

ps_err_e
ps_lgetxregsize(struct ps_prochandle *P, lwpid_t lwpid, int *xrsize)
{
```



```
        char fname[PATH_MAX];
        struct stat statb;

        if (P->state == PS_DEAD) {
                lwp_info_t *lwp = list_next(&P->core->core_lwp_head);
                uint_t i;

                for (i = 0; i < P->core->core_nlwp; i++, lwp = list_next(lwp)) {
                        if (lwp->lwp_id == lwpid) {
                                if (lwp->lwp_xregs != NULL)
                                        *xrsize = sizeof (prxregset_t);
                                else
                                        *xrsize = 0;
                                return (PS_OK);
                        }
                }

                return (PS_BADLID);
        }

        (void) snprintf(fname, sizeof (fname), "%s/%d/lwp/%d/xregs",
            procfs_path, (int)P->status.pr_pid, (int)lwpid);

        if (stat(fname, &statb) != 0)
                return (PS_BADLID);

        *xrsize = (int)statb.st_size;
        return (PS_OK);
}
```

_____unchanged_portion_omitted_

```
#endif  /* sparc */

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

ps_err_e
ps_lgetLDT(struct ps_prochandle *P, lwpid_t lwpid, struct ssd *ldt)
{
#if defined(__amd64) && defined(_LP64)
        if (P->status.pr_dmodel != PR_MODEL_NATIVE) {
#endif
        prgregset_t regs;
        struct ssd *ldtarray;
        ps_err_e error;
        uint_t gs;
        int nldt;
        int i;

        if (P->state != PS_STOP && P->state != PS_DEAD)
                return (PS_ERR);

        /*
         * We need to get the ldt entry that matches the
         * value in the lwp's GS register.
         */
        if ((error = ps_lgetregs(P, lwpid, regs)) != PS_OK)
                return (error);

        gs = regs[GS];

        if ((nldt = Pldt(P, NULL, 0)) <= 0 ||
            (ldtarray = malloc(nldt * sizeof (struct ssd))) == NULL)
                return (PS_ERR);
        if ((nldt = Pldt(P, ldtarray, nldt)) <= 0) {
                free(ldtarray);
                return (PS_ERR);
```

```
 259         }

 261         for (i = 0; i < nldt; i++) {
 262                 if (gs == ldtarray[i].sel) {
 263                         *ldt = ldtarray[i];
 264                         break;
 265                 }
 266         }
 267         free(ldtarray);

 269         if (i < nldt)
 270                 return (PS_OK);
 271 #if defined(__amd64) && defined(_LP64)
 272         }
 273 #endif

 275         return (PS_ERR);
 276 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   29219 Wed Jan 23 13:19:05 2013
new/usr/src/lib/libproc/common/libproc.h
XXX AVX procfs
**********************************************************
______unchanged_portion_omitted_

 174 /* values for type */
 175 #define AT_BYVAL        1
 176 #define AT_BYREF        2

 178 /* values for inout */
 179 #define AI_INPUT        1
 180 #define AI_OUTPUT       2
 181 #define AI_INOUT        3

 183 /* maximum number of syscall arguments */
 184 #define MAXARGS         8

 186 /* maximum size in bytes of a BYREF argument */
 187 #define MAXARGL         (4*1024)

 189 /*
 190  * Function prototypes for routines in the process control package.
 191  */
 192 extern struct ps_prochandle *Pcreate(const char *, char *const *,
 193     int *, char *, size_t);
 194 extern struct ps_prochandle *Pxcreate(const char *, char *const *,
 195     char *const *, int *, char *, size_t);

 197 extern const char *Pcreate_error(int);

 199 extern struct ps_prochandle *Pgrab(pid_t, int, int *);
 200 extern struct ps_prochandle *Pgrab_core(const char *, const char *, int, int *);
 201 extern struct ps_prochandle *Pfgrab_core(int, const char *, int *);
 202 extern struct ps_prochandle *Pgrab_file(const char *, int *);
 203 extern const char *Pgrab_error(int);

 205 extern  int     Preopen(struct ps_prochandle *);
 206 extern  void    Prelease(struct ps_prochandle *, int);
 207 extern  void    Pfree(struct ps_prochandle *);

 209 extern  int     Pasfd(struct ps_prochandle *);
 210 extern  char   *Pbrandname(struct ps_prochandle *, char *, size_t);
 211 extern  int     Pctlfd(struct ps_prochandle *);
 212 extern  int     Pcreate_agent(struct ps_prochandle *);
 213 extern  void    Pdestroy_agent(struct ps_prochandle *);
 214 extern  int     Pstopstatus(struct ps_prochandle *, long, uint_t);
 215 extern  int     Pwait(struct ps_prochandle *, uint_t);
 216 extern  int     Pstop(struct ps_prochandle *, uint_t);
 217 extern  int     Pdstop(struct ps_prochandle *);
 218 extern  int     Pstate(struct ps_prochandle *);
 219 extern  const psinfo_t *Ppsinfo(struct ps_prochandle *);
 220 extern  const pstatus_t *Pstatus(struct ps_prochandle *);
 221 extern  int     Pcred(struct ps_prochandle *, prcred_t *, int);
 222 extern  int     Psetcred(struct ps_prochandle *, const prcred_t *);
 223 extern  ssize_t Ppriv(struct ps_prochandle *, prpriv_t *, size_t);
 224 extern  int     Psetpriv(struct ps_prochandle *, prpriv_t *);
 225 extern  void   *Pprivinfo(struct ps_prochandle *);
 226 extern  int     Psetzoneid(struct ps_prochandle *, zoneid_t);
 227 extern  int     Pgetareg(struct ps_prochandle *, int, prgreg_t *);
 228 extern  int     Pputareg(struct ps_prochandle *, int, prgreg_t);
 229 extern  int     Psetrun(struct ps_prochandle *, int, int);
 230 extern  ssize_t Pread(struct ps_prochandle *, void *, size_t, uintptr_t);
 231 extern  ssize_t Pread_string(struct ps_prochandle *, char *, size_t, uintptr_t);
 232 extern  ssize_t Pwrite(struct ps_prochandle *, const void *, size_t, uintptr_t);
```



```
 233 extern  int     Pclearsig(struct ps_prochandle *);
 234 extern  int     Pclearfault(struct ps_prochandle *);
 235 extern  int     Psetbkpt(struct ps_prochandle *, uintptr_t, ulong_t *);
 236 extern  int     Pdelbkpt(struct ps_prochandle *, uintptr_t, ulong_t);
 237 extern  int     Pxecbkpt(struct ps_prochandle *, ulong_t);
 238 extern  int     Psetwapt(struct ps_prochandle *, const prwatch_t *);
 239 extern  int     Pdelwapt(struct ps_prochandle *, const prwatch_t *);
 240 extern  int     Pxecwapt(struct ps_prochandle *, const prwatch_t *);
 241 extern  int     Psetflags(struct ps_prochandle *, long);
 242 extern  int     Punsetflags(struct ps_prochandle *, long);
 243 extern  int     Psignal(struct ps_prochandle *, int, int);
 244 extern  int     Pfault(struct ps_prochandle *, int, int);
 245 extern  int     Psysentry(struct ps_prochandle *, int, int);
 246 extern  int     Psysexit(struct ps_prochandle *, int, int);
 247 extern  void    Psetsignal(struct ps_prochandle *, const sigset_t *);
 248 extern  void    Psetfault(struct ps_prochandle *, const fltset_t *);
 249 extern  void    Psetsysentry(struct ps_prochandle *, const sysset_t *);
 250 extern  void    Psetsysexit(struct ps_prochandle *, const sysset_t *);

 252 extern  void    Psync(struct ps_prochandle *);
 253 extern  int     Psyscall(struct ps_prochandle *, sysret_t *,
 254                         int, uint_t, argdes_t *);
 255 extern  int     Pisprocdir(struct ps_prochandle *, const char *);

 257 /*
 258  * Function prototypes for lwp-specific operations.
 259  */
 260 extern  struct ps_lwphandle *Lgrab(struct ps_prochandle *, lwpid_t, int *);
 261 extern  const char *Lgrab_error(int);

 263 extern  struct ps_prochandle *Lprochandle(struct ps_lwphandle *);
 264 extern  void    Lfree(struct ps_lwphandle *);

 266 extern  int     Lctlfd(struct ps_lwphandle *);
 267 extern  int     Lwait(struct ps_lwphandle *, uint_t);
 268 extern  int     Lstop(struct ps_lwphandle *, uint_t);
 269 extern  int     Ldstop(struct ps_lwphandle *);
 270 extern  int     Lstate(struct ps_lwphandle *);
 271 extern  const lwpsinfo_t *Lpsinfo(struct ps_lwphandle *);
 272 extern  const lwpstatus_t *Lstatus(struct ps_lwphandle *);
 273 extern  int     Lgetareg(struct ps_lwphandle *, int, prgreg_t *);
 274 extern  int     Lputareg(struct ps_lwphandle *, int, prgreg_t);
 275 extern  int     Lsetrun(struct ps_lwphandle *, int, int);
 276 extern  int     Lclearsig(struct ps_lwphandle *);
 277 extern  int     Lclearfault(struct ps_lwphandle *);
 278 extern  int     Lxecbkpt(struct ps_lwphandle *, ulong_t);
 279 extern  int     Lxecwapt(struct ps_lwphandle *, const prwatch_t *);
 280 extern  void    Lsync(struct ps_lwphandle *);

 282 extern  int     Lstack(struct ps_lwphandle *, stack_t *);
 283 extern  int     Lmain_stack(struct ps_lwphandle *, stack_t *);
 284 extern  int     Lalt_stack(struct ps_lwphandle *, stack_t *);

 286 /*
 287  * Function prototypes for system calls forced on the victim process.
 288  */
 289 extern  int     pr_open(struct ps_prochandle *, const char *, int, mode_t);
 290 extern  int     pr_creat(struct ps_prochandle *, const char *, mode_t);
 291 extern  int     pr_close(struct ps_prochandle *, int);
 292 extern  int     pr_access(struct ps_prochandle *, const char *, int);
 293 extern  int     pr_door_info(struct ps_prochandle *, int, struct door_info *);
 294 extern  void    *pr_mmap(struct ps_prochandle *,
 295                         void *, size_t, int, int, int, off_t);
 296 extern  void    *pr_zmap(struct ps_prochandle *,
 297                         void *, size_t, int, int);
 298 extern  int     pr_munmap(struct ps_prochandle *, void *, size_t);
```

```
299 extern  int     pr_memcntl(struct ps_prochandle *,
300                         caddr_t, size_t, int, caddr_t, int, int);
301 extern  int     pr_meminfo(struct ps_prochandle *, const uint64_t *addrs,
302                         int addr_count, const uint_t *info, int info_count,
303                         uint64_t *outdata, uint_t *validity);
304 extern  int     pr_sigaction(struct ps_prochandle *,
305                         int, const struct sigaction *, struct sigaction *);
306 extern  int     pr_getitimer(struct ps_prochandle *,
307                         int, struct itimerval *);
308 extern  int     pr_setitimer(struct ps_prochandle *,
309                         int, const struct itimerval *, struct itimerval *);
310 extern  int     pr_ioctl(struct ps_prochandle *, int, int, void *, size_t);
311 extern  int     pr_fcntl(struct ps_prochandle *, int, int, void *);
312 extern  int     pr_stat(struct ps_prochandle *, const char *, struct stat *);
313 extern  int     pr_lstat(struct ps_prochandle *, const char *, struct stat *);
314 extern  int     pr_fstat(struct ps_prochandle *, int, struct stat *);
315 extern  int     pr_stat64(struct ps_prochandle *, const char *,
316                         struct stat64 *);
317 extern  int     pr_lstat64(struct ps_prochandle *, const char *,
318                         struct stat64 *);
319 extern  int     pr_fstat64(struct ps_prochandle *, int, struct stat64 *);
320 extern  int     pr_statvfs(struct ps_prochandle *, const char *, statvfs_t *);
321 extern  int     pr_fstatvfs(struct ps_prochandle *, int, statvfs_t *);
322 extern  projid_t pr_getprojid(struct ps_prochandle *Pr);
323 extern  taskid_t pr_gettaskid(struct ps_prochandle *Pr);
324 extern  taskid_t pr_settaskid(struct ps_prochandle *Pr, projid_t project,
325                         int flags);
326 extern  zoneid_t pr_getzoneid(struct ps_prochandle *Pr);
327 extern  int     pr_getrctl(struct ps_prochandle *,
328                         const char *, rctlblk_t *, rctlblk_t *, int);
329 extern  int     pr_setrctl(struct ps_prochandle *,
330                         const char *, rctlblk_t *, rctlblk_t *, int);
331 extern  int     pr_getrlimit(struct ps_prochandle *,
332                         int, struct rlimit *);
333 extern  int     pr_setrlimit(struct ps_prochandle *,
334                         int, const struct rlimit *);
335 extern  int     pr_setprojrctl(struct ps_prochandle *, const char *,
336                         rctlblk_t *, size_t, int);
337 #if defined(_LARGEFILE64_SOURCE)
338 extern  int     pr_getrlimit64(struct ps_prochandle *,
339                         int, struct rlimit64 *);
340 extern  int     pr_setrlimit64(struct ps_prochandle *,
341                         int, const struct rlimit64 *);
342 #endif  /* _LARGEFILE64_SOURCE */
343 extern  int     pr_lwp_exit(struct ps_prochandle *);
344 extern  int     pr_exit(struct ps_prochandle *, int);
345 extern  int     pr_waitid(struct ps_prochandle *,
346                         idtype_t, id_t, siginfo_t *, int);
347 extern  off_t   pr_lseek(struct ps_prochandle *, int, off_t, int);
348 extern  offset_t pr_llseek(struct ps_prochandle *, int, offset_t, int);
349 extern  int     pr_rename(struct ps_prochandle *, const char *, const char *);
350 extern  int     pr_link(struct ps_prochandle *, const char *, const char *);
351 extern  int     pr_unlink(struct ps_prochandle *, const char *);
352 extern  int     pr_getpeerucred(struct ps_prochandle *, int, ucred_t **);
353 extern  int     pr_getpeername(struct ps_prochandle *,
354                         int, struct sockaddr *, socklen_t *);
355 extern  int     pr_getsockname(struct ps_prochandle *,
356                         int, struct sockaddr *, socklen_t *);
357 extern  int     pr_getsockopt(struct ps_prochandle *,
358                         int, int, int, void *, int *);
359 extern  int     pr_processor_bind(struct ps_prochandle *,
360                         idtype_t, id_t, int, int *);

362 /*
363  * Function prototypes for accessing per-LWP register information.
364  */
```



```
365 extern int Plwp_getregs(struct ps_prochandle *, lwpid_t, prgregset_t);
366 extern int Plwp_setregs(struct ps_prochandle *, lwpid_t, const prgregset_t);

368 extern int Plwp_getfpregs(struct ps_prochandle *, lwpid_t, prfpregset_t *);
369 extern int Plwp_setfpregs(struct ps_prochandle *, lwpid_t,
370     const prfpregset_t *);

372 #if defined(__sparc)

372 extern int Plwp_getxregs(struct ps_prochandle *, lwpid_t, prxregset_t *);
373 extern int Plwp_setxregs(struct ps_prochandle *, lwpid_t, const prxregset_t *);

375 #if defined(__sparc)

377 #endif /* ! codereview */
378 extern int Plwp_getgwindows(struct ps_prochandle *, lwpid_t, gwindows_t *);

380 #if defined(__sparcv9)
381 extern int Plwp_getasrs(struct ps_prochandle *, lwpid_t, asrset_t);
382 extern int Plwp_setasrs(struct ps_prochandle *, lwpid_t, const asrset_t);
383 #endif  /* __sparcv9 */

385 #endif  /* __sparc */

387 #if defined(__i386) || defined(__amd64)
388 extern  int     Pldt(struct ps_prochandle *, struct ssd *, int);
389 extern  int     proc_get_ldt(pid_t, struct ssd *, int);
390 #endif  /* __i386 || __amd64 */

392 extern int Plwp_getpsinfo(struct ps_prochandle *, lwpid_t, lwpsinfo_t *);

394 extern int Plwp_stack(struct ps_prochandle *, lwpid_t, stack_t *);
395 extern int Plwp_main_stack(struct ps_prochandle *, lwpid_t, stack_t *);
396 extern int Plwp_alt_stack(struct ps_prochandle *, lwpid_t, stack_t *);

398 /*
399  * LWP iteration interface; iterate over all active LWPs.
400  */
401 typedef int proc_lwp_f(void *, const lwpstatus_t *);
402 extern int Plwp_iter(struct ps_prochandle *, proc_lwp_f *, void *);

404 /*
405  * LWP iteration interface; iterate over all LWPs, active and zombie.
406  */
407 typedef int proc_lwp_all_f(void *, const lwpstatus_t *, const lwpsinfo_t *);
408 extern int Plwp_iter_all(struct ps_prochandle *, proc_lwp_all_f *, void *);

410 /*
411  * Process iteration interface; iterate over all non-system processes.
412  */
413 typedef int proc_walk_f(psinfo_t *, lwpsinfo_t *, void *);
414 extern int proc_walk(proc_walk_f *, void *, int);

416 #define PR_WALK_PROC    0               /* walk processes only */
417 #define PR_WALK_LWP     1               /* walk all lwps */

419 /*
420  * Determine if an lwp is in a set as returned from proc_arg_xgrab().
421  */
422 extern int proc_lwp_in_set(const char *, lwpid_t);
423 extern int proc_lwp_range_valid(const char *);

425 /*
426  * Symbol table interfaces.
427  */
```

```
/*
 * Pseudo-names passed to Plookup_by_name() for well-known load objects.
 * NOTE: It is required that PR_OBJ_EXEC and PR_OBJ_LDSO exactly match
 * the definitions of PS_OBJ_EXEC and PS_OBJ_LDSO from <proc_service.h>.
 */
#define PR_OBJ_EXEC     ((const char *)0)       /* search the executable file */
#define PR_OBJ_LDSO     ((const char *)1)       /* search ld.so.1 */
#define PR_OBJ_EVERY    ((const char *)-1)      /* search every load object */

/*
 * Special Lmid_t passed to Plookup_by_lmid() to search all link maps.  The
 * special values LM_ID_BASE and LM_ID_LDSO from <link.h> may also be used.
 * If PR_OBJ_EXEC is used as the object name, the lmid must be PR_LMID_EVERY
 * or LM_ID_BASE in order to return a match.  If PR_OBJ_LDSO is used as the
 * object name, the lmid must be PR_LMID_EVERY or LM_ID_LDSO to return a match.
 */
#define PR_LMID_EVERY   ((Lmid_t)-1UL)          /* search every link map */

/*
 * 'object_name' is the name of a load object obtained from an
 * iteration over the process's address space mappings (Pmapping_iter),
 * or an iteration over the process's mapped objects (Pobject_iter),
 * or else it is one of the special PR_OBJ_* values above.
 */
extern int Plookup_by_name(struct ps_prochandle *,
    const char *, const char *, GElf_Sym *);

extern int Plookup_by_addr(struct ps_prochandle *,
    uintptr_t, char *, size_t, GElf_Sym *);

typedef struct prsyminfo {
        const char      *prs_object;            /* object name */
        const char      *prs_name;              /* symbol name */
        Lmid_t          prs_lmid;               /* link map id */
        uint_t          prs_id;                 /* symbol id */
        uint_t          prs_table;              /* symbol table id */
} prsyminfo_t;

extern int Pxlookup_by_name(struct ps_prochandle *,
    Lmid_t, const char *, const char *, GElf_Sym *, prsyminfo_t *);

extern int Pxlookup_by_addr(struct ps_prochandle *,
    uintptr_t, char *, size_t, GElf_Sym *, prsyminfo_t *);
extern int Pxlookup_by_addr_resolved(struct ps_prochandle *,
    uintptr_t, char *, size_t, GElf_Sym *, prsyminfo_t *);

typedef int proc_map_f(void *, const prmap_t *, const char *);

extern int Pmapping_iter(struct ps_prochandle *, proc_map_f *, void *);
extern int Pmapping_iter_resolved(struct ps_prochandle *, proc_map_f *, void *);
extern int Pobject_iter(struct ps_prochandle *, proc_map_f *, void *);
extern int Pobject_iter_resolved(struct ps_prochandle *, proc_map_f *, void *);

extern const prmap_t *Paddr_to_map(struct ps_prochandle *, uintptr_t);
extern const prmap_t *Paddr_to_text_map(struct ps_prochandle *, uintptr_t);
extern const prmap_t *Pname_to_map(struct ps_prochandle *, const char *);
extern const prmap_t *Plmid_to_map(struct ps_prochandle *,
    Lmid_t, const char *);

extern const rd_loadobj_t *Paddr_to_loadobj(struct ps_prochandle *, uintptr_t);
extern const rd_loadobj_t *Pname_to_loadobj(struct ps_prochandle *,
    const char *);
extern const rd_loadobj_t *Plmid_to_loadobj(struct ps_prochandle *,
    Lmid_t, const char *);

extern ctf_file_t *Paddr_to_ctf(struct ps_prochandle *, uintptr_t);
```



```
extern ctf_file_t *Pname_to_ctf(struct ps_prochandle *, const char *);

extern char *Pplatform(struct ps_prochandle *, char *, size_t);
extern int Puname(struct ps_prochandle *, struct utsname *);
extern char *Pzonename(struct ps_prochandle *, char *, size_t);
extern char *Pfindobj(struct ps_prochandle *, const char *, char *, size_t);

extern char *Pexecname(struct ps_prochandle *, char *, size_t);
extern char *Pobjname(struct ps_prochandle *, uintptr_t, char *, size_t);
extern char *Pobjname_resolved(struct ps_prochandle *, uintptr_t, char *,
    size_t);
extern int Plmid(struct ps_prochandle *, uintptr_t, Lmid_t *);

typedef int proc_env_f(void *, struct ps_prochandle *, uintptr_t, const char *);
extern  int Penv_iter(struct ps_prochandle *, proc_env_f *, void *);
extern char *Pgetenv(struct ps_prochandle *, const char *, char *, size_t);
extern long Pgetauxval(struct ps_prochandle *, int);
extern const auxv_t *Pgetauxvec(struct ps_prochandle *);

extern void Pset_procfs_path(const char *);

/*
 * Symbol table iteration interface.  The special lmid constants LM_ID_BASE,
 * LM_ID_LDSO, and PR_LMID_EVERY may be used with Psymbol_iter_by_lmid.
 */
typedef int proc_sym_f(void *, const GElf_Sym *, const char *);
typedef int proc_xsym_f(void *, const GElf_Sym *, const char *,
    const prsyminfo_t *);

extern int Psymbol_iter(struct ps_prochandle *,
    const char *, int, int, proc_sym_f *, void *);
extern int Psymbol_iter_by_addr(struct ps_prochandle *,
    const char *, int, int, proc_sym_f *, void *);
extern int Psymbol_iter_by_name(struct ps_prochandle *,
    const char *, int, int, proc_sym_f *, void *);

extern int Psymbol_iter_by_lmid(struct ps_prochandle *,
    Lmid_t, const char *, int, int, proc_sym_f *, void *);

extern int Pxsymbol_iter(struct ps_prochandle *,
    Lmid_t, const char *, int, int, proc_xsym_f *, void *);

/*
 * 'which' selects which symbol table and can be one of the following.
 */
#define PR_SYMTAB       1
#define PR_DYNSYM       2
/*
 * 'type' selects the symbols of interest by binding and type.  It is a bit-
 * mask of one or more of the following flags, whose order MUST match the
 * order of STB and STT constants in <sys/elf.h>.
 */
#define BIND_LOCAL      0x0001
#define BIND_GLOBAL     0x0002
#define BIND_WEAK       0x0004
#define BIND_ANY (BIND_LOCAL|BIND_GLOBAL|BIND_WEAK)
#define TYPE_NOTYPE     0x0100
#define TYPE_OBJECT     0x0200
#define TYPE_FUNC       0x0400
#define TYPE_SECTION    0x0800
#define TYPE_FILE       0x1000
#define TYPE_ANY (TYPE_NOTYPE|TYPE_OBJECT|TYPE_FUNC|TYPE_SECTION|TYPE_FILE)

/*
 * This returns the rtld_db agent handle for the process.
 * The handle will become invalid at the next successful exec() and
```

```
 * must not be used beyond that point (see Preset_maps(), below).
 */
extern rd_agent_t *Prd_agent(struct ps_prochandle *);

/*
 * This should be called when an RD_DLACTIVITY event with the
 * RD_CONSISTENT state occurs via librtld_db's event mechanism.
 * This makes libproc's address space mappings and symbol tables current.
 * The variant Pupdate_syms() can be used to preload all symbol tables as well.
 */
extern void Pupdate_maps(struct ps_prochandle *);
extern void Pupdate_syms(struct ps_prochandle *);

/*
 * This must be called after the victim process performs a successful
 * exec() if any of the symbol table interface functions have been called
 * prior to that point.  This is essential because an exec() invalidates
 * all previous symbol table and address space mapping information.
 * It is always safe to call, but if it is called other than after an
 * exec() by the victim process it just causes unnecessary overhead.
 *
 * The rtld_db agent handle obtained from a previous call to Prd_agent() is
 * made invalid by Preset_maps() and Prd_agent() must be called again to get
 * the new handle.
 */
extern void Preset_maps(struct ps_prochandle *);

/*
 * Given an address, Ppltdest() determines if this is part of a PLT, and if
 * so returns a pointer to the symbol name that will be used for resolution.
 * If the specified address is not part of a PLT, the function returns NULL.
 */
extern const char *Ppltdest(struct ps_prochandle *, uintptr_t);

/*
 * See comments for Pissyscall(), in Pisadep.h
 */
extern int Pissyscall_prev(struct ps_prochandle *, uintptr_t, uintptr_t *);

/*
 * Stack frame iteration interface.
 */
typedef int proc_stack_f(void *, prgregset_t, uint_t, const long *);

extern int Pstack_iter(struct ps_prochandle *,
    const prgregset_t, proc_stack_f *, void *);

/*
 * The following functions define a set of passive interfaces: libproc provides
 * default, empty definitions that are called internally.  If a client wishes
 * to override these definitions, it can simply provide its own version with
 * the same signature that interposes on the libproc definition.
 *
 * If the client program wishes to report additional error information, it
 * can provide its own version of Perror_printf.
 *
 * If the client program wishes to receive a callback after Pcreate forks
 * but before it execs, it can provide its own version of Pcreate_callback.
 */
extern void Perror_printf(struct ps_prochandle *P, const char *format, ...);
extern void Pcreate_callback(struct ps_prochandle *);

/*
 * Remove unprintable characters from psinfo.pr_psargs and replace with
 * whitespace characters so it is safe for printing.
 */
```



```
extern void proc_unctrl_psinfo(psinfo_t *);

/*
 * Utility functions for processing arguments which should be /proc files,
 * pids, and/or core files.  The returned error code can be passed to
 * Pgrab_error() in order to convert it to an error string.
 */
#define PR_ARG_PIDS     0x1     /* Allow pid and /proc file arguments */
#define PR_ARG_CORES    0x2     /* Allow core file arguments */

#define PR_ARG_ANY      (PR_ARG_PIDS | PR_ARG_CORES)

extern struct ps_prochandle *proc_arg_grab(const char *, int, int, int *);
extern struct ps_prochandle *proc_arg_xgrab(const char *, const char *, int,
    int, int *, const char **);
extern pid_t proc_arg_psinfo(const char *, int, psinfo_t *, int *);
extern pid_t proc_arg_xpsinfo(const char *, int, psinfo_t *, int *,
    const char **);

/*
 * Utility functions for obtaining information via /proc without actually
 * performing a Pcreate() or Pgrab():
 */
extern int proc_get_auxv(pid_t, auxv_t *, int);
extern int proc_get_cred(pid_t, prcred_t *, int);
extern prpriv_t *proc_get_priv(pid_t);
extern int proc_get_psinfo(pid_t, psinfo_t *);
extern int proc_get_status(pid_t, pstatus_t *);

/*
 * Utility functions for debugging tools to convert numeric fault,
 * signal, and system call numbers to symbolic names:
 */
#define FLT2STR_MAX 32  /* max. string length of faults (like SIG2STR_MAX) */
#define SYS2STR_MAX 32  /* max. string length of syscalls (like SIG2STR_MAX) */

extern char *proc_fltname(int, char *, size_t);
extern char *proc_signame(int, char *, size_t);
extern char *proc_sysname(int, char *, size_t);

/*
 * Utility functions for debugging tools to convert fault, signal, and system
 * call names back to the numeric constants:
 */
extern int proc_str2flt(const char *, int *);
extern int proc_str2sig(const char *, int *);
extern int proc_str2sys(const char *, int *);

/*
 * Utility functions for debugging tools to convert a fault, signal or system
 * call set to a string representation (e.g. "BUS,SEGV" or "open,close,read").
 */
#define PRSIGBUFSZ      1024    /* buffer size for proc_sigset2str() */

extern char *proc_fltset2str(const fltset_t *, const char *, int,
    char *, size_t);
extern char *proc_sigset2str(const sigset_t *, const char *, int,
    char *, size_t);
extern char *proc_sysset2str(const sysset_t *, const char *, int,
    char *, size_t);

extern int Pgcore(struct ps_prochandle *, const char *, core_content_t);
extern int Pfgcore(struct ps_prochandle *, int, core_content_t);
extern core_content_t Pcontent(struct ps_prochandle *);

/*
```

```
 * Utility functions for debugging tools to convert a string representation of
 * a fault, signal or system call set back to the numeric value of the
 * corresponding set type.
 */
extern char *proc_str2fltset(const char *, const char *, int, fltset_t *);
extern char *proc_str2sigset(const char *, const char *, int, sigset_t *);
extern char *proc_str2sysset(const char *, const char *, int, sysset_t *);

/*
 * Utility functions for converting between strings and core_content_t.
 */
#define PRCONTENTBUFSZ  80      /* buffer size for proc_content2str() */

extern int proc_str2content(const char *, core_content_t *);
extern int proc_content2str(core_content_t, char *, size_t);

/*
 * Utility functions for buffering output to stdout, stderr while
 * process is grabbed.  Prevents deadlocks due to pfiles 'pgrep xterm'
 * and other varients.
 */
extern int proc_initstdio(void);
extern int proc_flushstdio(void);
extern int proc_finistdio(void);

/*
 * Iterate over all open files.
 */
typedef int proc_fdinfo_f(void *, prfdinfo_t *);
extern int Pfdinfo_iter(struct ps_prochandle *, proc_fdinfo_f *, void *);

#ifdef  __cplusplus
}
#endif

#endif  /* _LIBPROC_H */
```

```
**********************************************************
   15962 Wed Jan 23 13:19:05 2013
new/usr/src/lib/libproc/common/llib-lproc
XXX AVX procfs
**********************************************************
/*
 * 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
 */
/* LINTLIBRARY */
/* PROTOLIB1 */

/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */
#include "libproc.h"

/*
 * usr/src/lib/libproc
 */

/* Pcontrol.c */
int     _libproc_debug;
struct ps_prochandle *Pcreate(const char *file, char *const *argv,
                int *perr, char *path, size_t len);
const char *Pcreate_error(int error);
void    Pcreate_callback(struct ps_prochandle *Pr);
struct ps_prochandle *Pgrab(pid_t pid, int gflag, int *perr);
const char *Pgrab_error(int error);
void    Pfree(struct ps_prochandle *Pr);
int     Pstate(struct ps_prochandle *Pr);
int     Pasfd(struct ps_prochandle *Pr);
int     Pctlfd(struct ps_prochandle *Pr);
const psinfo_t *Ppsinfo(struct ps_prochandle *Pr);
const pstatus_t *Pstatus(struct ps_prochandle *Pr);
int     Pcred(struct ps_prochandle *Pr, prcred_t *pcrp, int ngroups);
ssize_t Ppriv(struct ps_prochandle *Pr, prpriv_t *pprivp, size_t);
void    Psync(struct ps_prochandle *Pr);
int     Pcreate_agent(struct ps_prochandle *Pr);
void    Pdestroy_agent(struct ps_prochandle *Pr);
int     Preopen(struct ps_prochandle *Pr);
void    Prelease(struct ps_prochandle *Pr, int flags);
int     Pstopstatus(struct ps_prochandle *Pr, long cmd, uint_t msec);
int     Pwait(struct ps_prochandle *Pr, uint_t msec);
int     Pstop(struct ps_prochandle *Pr, uint_t msec);
int     Pdstop(struct ps_prochandle *Pr);
int     Pgetareg(struct ps_prochandle *Pr, int regno, prgreg_t *preg);
int     Pputareg(struct ps_prochandle *Pr, int regno, prgreg_t reg);
int     Psetrun(struct ps_prochandle *Pr, int sig, int flags);
```



```
ssize_t Pread(struct ps_prochandle *Pr,
                void *buf, size_t nbyte, uintptr_t address);
ssize_t Pread_string(struct ps_prochandle *Pr,
                char *buf, size_t nbyte, uintptr_t address);
ssize_t Pwrite(struct ps_prochandle *Pr,
                const void *buf, size_t nbyte, uintptr_t address);
int     Pclearsig(struct ps_prochandle *Pr);
int     Pclearfault(struct ps_prochandle *Pr);
int     Psetbkpt(struct ps_prochandle *Pr, uintptr_t address, ulong_t *saved);
int     Pdelbkpt(struct ps_prochandle *Pr, uintptr_t address, ulong_t saved);
int     Pxecbkpt(struct ps_prochandle *Pr, ulong_t saved);
int     Psetwapt(struct ps_prochandle *Pr, const prwatch_t *wp);
int     Pdelwapt(struct ps_prochandle *Pr, const prwatch_t *wp);
int     Pxecwapt(struct ps_prochandle *Pr, const prwatch_t *wp);
int     Psetflags(struct ps_prochandle *Pr, long flags);
int     Punsetflags(struct ps_prochandle *Pr, long flags);
int     Psignal(struct ps_prochandle *Pr, int which, int stop);
void    Psetsignal(struct ps_prochandle *Pr, const sigset_t *set);
int     Pfault(struct ps_prochandle *Pr, int which, int stop);
void    Psetfault(struct ps_prochandle *Pr, const fltset_t *set);
int     Psysentry(struct ps_prochandle *Pr, int which, int stop);
void    Psetsysentry(struct ps_prochandle *Pr, const sysset_t *set);
int     Psysexit(struct ps_prochandle *Pr, int which, int stop);
void    Psetsysexit(struct ps_prochandle *Pr, const sysset_t *set);
int     Plwp_iter(struct ps_prochandle *Pr, proc_lwp_f *func, void *cd);
int     Psyscall(struct ps_prochandle *Pr, sysret_t *,
                int sysindex, uint_t nargs, argdes_t *argp);

struct ps_lwphandle *Lgrab(struct ps_prochandle *P, lwpid_t lwpid, int *perr);
const char *Lgrab_error(int error);
struct ps_prochandle *Lprochandle(struct ps_lwphandle *Lwp);
void    Lfree(struct ps_lwphandle *Lwp);
int     Lctlfd(struct ps_lwphandle *Lwp);
int     Lwait(struct ps_lwphandle *Lwp, uint_t msec);
int     Lstop(struct ps_lwphandle *Lwp, uint_t msec);
int     Ldstop(struct ps_lwphandle *Lwp);
int     Lstate(struct ps_lwphandle *Lwp);
const lwpsinfo_t *Lpsinfo(struct ps_lwphandle *Lwp);
const lwpstatus_t *Lstatus(struct ps_lwphandle *Lwp);
int     Lgetareg(struct ps_lwphandle *Lwp, int regno, prgreg_t *preg);
int     Lputareg(struct ps_lwphandle *Lwp, int regno, prgreg_t reg);
int     Lsetrun(struct ps_lwphandle *Lwp, int sig, int flags);
int     Lclearsig(struct ps_lwphandle *Lwp);
int     Lclearfault(struct ps_lwphandle *Lwp);
int     Lxecbkpt(struct ps_lwphandle *Lwp, ulong_t saved);
int     Lxecwapt(struct ps_lwphandle *Lwp, const prwatch_t *wp);
void    Lsync(struct ps_lwphandle *Lwp);

/* Plwpregs.c */
int Plwp_getregs(struct ps_prochandle *Pr, lwpid_t i, prgregset_t gr);
int Plwp_setregs(struct ps_prochandle *Pr, lwpid_t i, const prgregset_t gr);
int Plwp_getfpregs(struct ps_prochandle *Pr, lwpid_t i, prfpregset_t *fp);
int Plwp_setfpregs(struct ps_prochandle *Pr, lwpid_t i, const prfpregset_t *fp);
#if defined(sparc) || defined(__sparc)
int Plwp_getxregs(struct ps_prochandle *Pr, lwpid_t i, prxregset_t *xr);
int Plwp_setxregs(struct ps_prochandle *Pr, lwpid_t i, const prxregset_t *xr);
#if defined(__sparcv9)
int Plwp_getasrs(struct ps_prochandle *Pr, lwpid_t i, asrset_t asrs);
int Plwp_setasrs(struct ps_prochandle *Pr, lwpid_t i, const asrset_t asrs);
#endif  /* __sparcv9 */
#endif  /* __sparc */
int Plwp_getpsinfo(struct ps_prochandle *Pr, lwpid_t i, lwpsinfo_t *lps);

/* Pcore.c */
struct ps_prochandle *Pfgrab_core(int fd, const char *aout, int *perr);
struct ps_prochandle *Pgrab_core(const char *core, const char *aout,
```

```
126         int gflag, int *perr);

128 /* Pisprocdir.c */
129 int     Pisprocdir(struct ps_prochandle *Pr, const char *dir);

131 /* Pservice.c */
132 ps_err_e ps_pdmodel(struct ps_prochandle *Pr, int *modelp);
133 ps_err_e ps_pread(struct ps_prochandle *Pr,
134                 psaddr_t addr, void *buf, size_t size);
135 ps_err_e ps_pwrite(struct ps_prochandle *Pr,
136                 psaddr_t addr, const void *buf, size_t size);
137 ps_err_e ps_pdread(struct ps_prochandle *Pr,
138                 psaddr_t addr, void *buf, size_t size);
139 ps_err_e ps_pdwrite(struct ps_prochandle *Pr,
140                 psaddr_t addr, const void *buf, size_t size);
141 ps_err_e ps_ptread(struct ps_prochandle *Pr,
142                 psaddr_t addr, void *buf, size_t size);
143 ps_err_e ps_ptwrite(struct ps_prochandle *Pr,
144                 psaddr_t addr, const void *buf, size_t size);
145 ps_err_e ps_pstop(struct ps_prochandle *Pr);
146 ps_err_e ps_pcontinue(struct ps_prochandle *Pr);
147 ps_err_e ps_lstop(struct ps_prochandle *Pr, lwpid_t lwpid);
148 ps_err_e ps_lcontinue(struct ps_prochandle *Pr, lwpid_t lwpid);
149 ps_err_e ps_lgetregs(struct ps_prochandle *Pr,
150                 lwpid_t lwpid, prgregset_t regs);
151 ps_err_e ps_lsetregs(struct ps_prochandle *Pr,
152                 lwpid_t lwpid, const prgregset_t regs);
153 ps_err_e ps_lgetfpregs(struct ps_prochandle *Pr,
154                 lwpid_t lwpid, prfpregset_t *regs);
155 ps_err_e ps_lsetfpregs(struct ps_prochandle *Pr,
156                 lwpid_t lwpid, const prfpregset_t *regs);
157 #if defined(sparc) || defined(__sparc)
158 ps_err_e ps_lgetxregsize(struct ps_prochandle *Pr,
159                 lwpid_t lwpid, int *xrsize);
160 ps_err_e ps_lgetxregs(struct ps_prochandle *Pr,
161                 lwpid_t lwpid, caddr_t xregs);
162 ps_err_e ps_lsetxregs(struct ps_prochandle *Pr,
163                 lwpid_t lwpid, caddr_t xregs);
164 #endif  /* sparc */
165 #if defined(__i386) || defined(__amd64)
166 ps_err_e ps_lgetLDT(struct ps_prochandle *Pr,
167                 lwpid_t lwpid, struct ssd *ldt);
168 #endif  /* __i386 || __amd6464 */
169 void    ps_plog(const char *fmt, ...);

171 /* Psymtab.c */
172 void    Pupdate_maps(struct ps_prochandle *Pr);
173 void    Pupdate_syms(struct ps_prochandle *Pr);
174 rd_agent_t *Prd_agent(struct ps_prochandle *Pr);
175 const prmap_t *Paddr_to_map(struct ps_prochandle *Pr, uintptr_t addr);
176 const prmap_t *Paddr_to_text_map(struct ps_prochandle *Pr, uintptr_t addr);
177 const prmap_t *Pname_to_map(struct ps_prochandle *Pr, const char *name);
178 const prmap_t *Plmid_to_map(struct ps_prochandle *Pr, Lmid_t lmid,
179                 const char *name);
180 int     Plookup_by_addr(struct ps_prochandle *Pr, uintptr_t addr,
181                 char *sym_name_buffer, size_t bufsize, GElf_Sym *symbolp);
182 int     Plookup_by_name(struct ps_prochandle *Pr,
183                 const char *object_name, const char *symbol_name,
184                 GElf_Sym *sym);
185 int     Plookup_by_lmid(struct ps_prochandle *Pr,
186                 Lmid_t lmid, const char *object_name, const char *symbol_name,
187                 GElf_Sym *sym);
188 const rd_loadobj_t *Paddr_to_loadobj(struct ps_prochandle *, uintptr_t);
189 const rd_loadobj_t *Pname_to_loadobj(struct ps_prochandle *, const char *);
190 const rd_loadobj_t *Plmid_to_loadobj(struct ps_prochandle *, Lmid_t,
191                 const char *);
```



```
192 int     Pmapping_iter(struct ps_prochandle *Pr, proc_map_f *func, void *cd);
193 int     Pmapping_iter_resolved(struct ps_prochandle *Pr, proc_map_f *func,
194                 void *cd);
195 int     Pobject_iter(struct ps_prochandle *Pr, proc_map_f *func, void *cd);
196 int     Pobject_iter_resolved(struct ps_prochandle *Pr, proc_map_f *func,
197                 void *cd);
198 char    *Pobjname(struct ps_prochandle *Pr, uintptr_t addr,
199                 char *buffer, size_t bufsize);
200 char    *Pobjname_resolved(struct ps_prochandle *Pr, uintptr_t addr,
201                 char *buffer, size_t bufsize);
202 int     Plmid(struct ps_prochandle *Pr, uintptr_t addr, Lmid_t *lmidp);
203 int     Psymbol_iter(struct ps_prochandle *Pr, const char *object_name,
204                 int which, int type, proc_sym_f *func, void *cd);
205 int     Psymbol_iter_by_lmid(struct ps_prochandle *Pr, Lmid_t lmid,
206                 const char *object_name, int which, int type,
207                 proc_sym_f *func, void *cd);
208 char    *Pgetenv(struct ps_prochandle *Pr, const char *name,
209                 char *buffer, size_t bufsize);
210 char    *Pplatform(struct ps_prochandle *Pr, char *s, size_t n);
211 int     Puname(struct ps_prochandle *Pr, struct utsname *u);
212 char    *Pzonename(struct ps_prochandle *Pr, char *s, size_t n);
213 char    *Pfindobj(struct ps_prochandle *Pr, const char *path,
214                 char *s, size_t n);
215 char    *Pexecname(struct ps_prochandle *Pr, char *buffer, size_t bufsize);
216 void    Preset_maps(struct ps_prochandle *Pr);

218 ps_err_e ps_pglobal_lookup(struct ps_prochandle *Pr,
219                 const char *object_name, const char *sym_name,
220                 psaddr_t *sym_addr);

222 ps_err_e ps_pglobal_sym(struct ps_prochandle *Pr,
223                 const char *object_name, const char *sym_name,
224                 ps_sym_t *symp);

226 long    Pgetauxval(struct ps_prochandle *Pr, int type);
227 const auxv_t *Pgetauxvec(struct ps_prochandle *Pr);
228 ps_err_e ps_pauxv(struct ps_prochandle *Pr, const auxv_t **aux);

230 /* Putil.c */
231 void    Perror_printf(struct ps_prochandle *Pr, const char *format, ...);

233 /* pr_door.c */
234 int     pr_door_info(struct ps_prochandle *Pr, int did, door_info_t *di);

236 /* pr_exit.c */
237 int     pr_exit(struct ps_prochandle *Pr, int status);
238 int     pr_lwp_exit(struct ps_prochandle *Pr);

240 /* pr_fcntl.c */
241 int     pr_fcntl(struct ps_prochandle *Pr, int fd, int cmd, void *argp);

243 /* pr_getitimer.c */
244 int     pr_getitimer(struct ps_prochandle *Pr,
245                 int which, struct itimerval *itv);
246 int     pr_setitimer(struct ps_prochandle *Pr,
247                 int which, const struct itimerval *itv, struct itimerval *oitv);

249 /* pr_getrctl.c */
250 int     pr_getrctl(struct ps_prochandle *Pr, const char *rname,
251                 rctlblk_t *old_blk, rctlblk_t *new_blk, int rflag);
252 int   pr_setrctl(struct ps_prochandle *Pr, const char *rname,
253                 rctlblk_t *old_blk, rctlblk_t *new_blk, int rflag);
254 int   pr_setprojrctl(struct ps_prochandle *Pr, const char *rname,
255                 rctlblk_t *new_blk, size_t size, int rflag);

257 /* pr_getrlimit.c */
```

```
258 int     pr_getrlimit(struct ps_prochandle *Pr,
259                 int resource, struct rlimit *rlp);
260 int     pr_setrlimit(struct ps_prochandle *Pr,
261                 int resource, const struct rlimit *rlp);
262 int     pr_getrlimit64(struct ps_prochandle *Pr,
263                 int resource, struct rlimit64 *rlp);
264 int     pr_setrlimit64(struct ps_prochandle *Pr,
265                 int resource, const struct rlimit64 *rlp);

267 /* pr_getsockname.c */
268 int     pr_getsockname(struct ps_prochandle *Pr,
269                 int sock, struct sockaddr *name, socklen_t *namelen);
270 int     pr_getpeername(struct ps_prochandle *Pr,
271                 int sock, struct sockaddr *name, socklen_t *namelen);

273 /* pr_ioctl.c */
274 int     pr_ioctl(struct ps_prochandle *Pr,
275                 int fd, int code, void *buf, size_t size);

277 /* pr_lseek.c */
278 off_t   pr_lseek(struct ps_prochandle *Pr,
279                 int filedes, off_t offset, int whence);
280 offset_t pr_llseek(struct ps_prochandle *Pr,
281                 int filedes, offset_t offset, int whence);

283 /* pr_memcntl.c */
284 int     pr_memcntl(struct ps_prochandle *Pr,
285         caddr_t addr, size_t len, int cmd, caddr_t arg, int attr, int mask);

287 /* pr_mmap.c */
288 void    *pr_mmap(struct ps_prochandle *Pr,
289                 void *addr, size_t len, int prot, int flags, int fd, off_t off);
290 int     pr_munmap(struct ps_prochandle *Pr,
291                 void *addr, size_t len);
292 void    *pr_zmap(struct ps_prochandle *Pr,
293                 void *addr, size_t len, int prot, int flags);

295 /* pr_open.c */
296 int     pr_open(struct ps_prochandle *Pr,
297                 const char *filename, int flags, mode_t mode);
298 int     pr_creat(struct ps_prochandle *Pr,
299                 const char *filename, mode_t mode);
300 int     pr_close(struct ps_prochandle *Pr, int fd);
301 int     pr_access(struct ps_prochandle *Pr, const char *path, int amode);

303 /* pr_pbind.c */
304 int     pr_processor_bind(struct ps_prochandle *Pr, idtype_t, id_t, int, int *);

306 /* pr_rename.c */
307 int     pr_rename(struct ps_prochandle *Pr, const char *old, const char *new);
308 int     pr_link(struct ps_prochandle *Pr, const char *exist, const char *new);
309 int     pr_unlink(struct ps_prochandle *Pr, const char *);

311 /* pr_sigaction.c */
312 int     pr_sigaction(struct ps_prochandle *Pr,
313                 int sig, const struct sigaction *act, struct sigaction *oact);

315 /* pr_stat.c */
316 int     pr_stat(struct ps_prochandle *Pr, const char *path, struct stat *buf);
317 int     pr_lstat(struct ps_prochandle *Pr, const char *path, struct stat *buf);
318 int     pr_fstat(struct ps_prochandle *Pr, int fd, struct stat *buf);
319 int     pr_stat64(struct ps_prochandle *Pr, const char *path,
320                 struct stat64 *buf);
321 int     pr_lstat64(struct ps_prochandle *Pr, const char *path,
322                 struct stat64 *buf);
323 int     pr_fstat64(struct ps_prochandle *Pr, int fd, struct stat64 *buf);
```



```
325 /* pr_statvfs.c */
326 int     pr_statvfs(struct ps_prochandle *Pr, const char *path, statvfs_t *buf);
327 int     pr_fstatvfs(struct ps_prochandle *Pr, int fd, statvfs_t *buf);

329 /* pr_tasksys.c */
330 projid_t pr_getprojid(struct ps_prochandle *Pr);
331 taskid_t pr_gettaskid(struct ps_prochandle *Pr);
332 taskid_t pr_settaskid(struct ps_prochandle *Pr, projid_t project, int flags);

334 /* pr_waitid.c */
335 int     pr_waitid(struct ps_prochandle *Pr,
336                 idtype_t idtype, id_t id, siginfo_t *infop, int options);

338 /* proc_get_info.c */
339 int     proc_get_cred(pid_t pid, prcred_t *credp, int ngroups);
340 prpriv_t *proc_get_priv(pid_t pid);
341 int     proc_get_psinfo(pid_t pid, psinfo_t *psp);
342 int     proc_get_status(pid_t pid, pstatus_t *psp);
343 int     proc_get_auxv(pid_t pid, auxv_t *pauxv, int naux);

345 /* proc_names.c */
346 char    *proc_fltname(int flt, char *buf, size_t bufsz);
347 char    *proc_signame(int sig, char *buf, size_t bufsz);
348 char    *proc_sysname(int sys, char *buf, size_t bufsz);

350 int     proc_str2flt(const char *str, int *fltnum);
351 int     proc_str2sig(const char *str, int *signum);
352 int     proc_str2sys(const char *str, int *sysnum);

354 char    *proc_fltset2str(const fltset_t *set, const char *delim, int members,
355                 char *buf, size_t nbytes);
356 char    *proc_sigset2str(const sigset_t *set, const char *delim, int members,
357                 char *buf, size_t nbytes);
358 char    *proc_sysset2str(const sysset_t *set, const char *delim, int members,
359                 char *buf, size_t nbytes);

361 char    *proc_str2fltset(const char *str, const char *delim, int members,
362                 fltset_t *set);
363 char    *proc_str2sigset(const char *str, const char *delim, int members,
364                 sigset_t *set);
365 char    *proc_str2sysset(const char *str, const char *delim, int members,
366                 sysset_t *set);

368 int     proc_walk(proc_walk_f *func, void *arg, int flags);

370 /* proc_arg.c */
371 struct ps_prochandle *proc_arg_grab(const char *arg,
372         int oflag, int gflag, int *perr);

374 pid_t   proc_arg_psinfo(const char *arg, int oflag, psinfo_t *psp, int *perr);
375 void    proc_unctrl_psinfo(psinfo_t *psp);

377 /* proc_set.c */
378 int     Psetcred(struct ps_prochandle *Pr, const prcred_t *pcred);

380 /* Pstack.c */
381 int     Pstack_iter(struct ps_prochandle *Pr,
382                 const prgregset_t regs, proc_stack_f *func, void *arg);

384 /* Pisadep.c */
385 const char *Ppltdest(struct ps_prochandle *Pr, uintptr_t addr);
```

```
**********************************************************
   57418 Wed Jan 23 13:19:05 2013
new/usr/src/uts/common/fs/proc/prcontrol.c
XXX AVX procfs
**********************************************************
```

```
/*
 * 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 2010 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#include <sys/types.h>
#include <sys/uio.h>
#include <sys/param.h>
#include <sys/cmn_err.h>
#include <sys/cred.h>
#include <sys/policy.h>
#include <sys/debug.h>
#include <sys/errno.h>
#include <sys/file.h>
#include <sys/inline.h>
#include <sys/kmem.h>
#include <sys/proc.h>
#include <sys/brand.h>
#include <sys/regset.h>
#include <sys/sysmacros.h>
#include <sys/systm.h>
#include <sys/vfs.h>
#include <sys/vnode.h>
#include <sys/signal.h>
#include <sys/auxv.h>
#include <sys/user.h>
#include <sys/class.h>
#include <sys/fault.h>
#include <sys/syscall.h>
#include <sys/procfs.h>
#include <sys/zone.h>
#include <sys/copyops.h>
#include <sys/schedctl.h>
#include <vm/as.h>
#include <vm/seg.h>
#include <fs/proc/prdata.h>
#include <sys/contract/process_impl.h>

static  void    pr_settrace(proc_t *, sigset_t *);
static  int     pr_setfpregs(prnode_t *, prfpregset_t *);
```



```
static  int     pr_setxregs(prnode_t *, prxregset_t *);
#endif /* ! codereview */
#if defined(__sparc)
static  int     pr_setxregs(prnode_t *, prxregset_t *);
static  int     pr_setasrs(prnode_t *, asrset_t);
#endif
static  int     pr_setvaddr(prnode_t *, caddr_t);
static  int     pr_clearsig(prnode_t *);
static  int     pr_clearflt(prnode_t *);
static  int     pr_watch(prnode_t *, prwatch_t *, int *);
static  int     pr_agent(prnode_t *, prgregset_t, int *);
static  int     pr_rdwr(proc_t *, enum uio_rw, priovec_t *);
static  int     pr_scred(proc_t *, prcred_t *, cred_t *, boolean_t);
static  int     pr_spriv(proc_t *, prpriv_t *, cred_t *);
static  int     pr_szoneid(proc_t *, zoneid_t, cred_t *);
static  void    pauselwps(proc_t *);
static  void    unpauselwps(proc_t *);

typedef union {
        long            sig;            /* PCKILL, PCUNKILL */
        long            nice;           /* PCNICE */
        long            timeo;          /* PCTWSTOP */
        ulong_t         flags;          /* PCRUN, PCSET, PCUNSET */
        caddr_t         vaddr;          /* PCSVADDR */
        siginfo_t       siginfo;        /* PCSSIG */
        sigset_t        sigset;         /* PCSTRACE, PCSHOLD */
        fltset_t        fltset;         /* PCSFAULT */
        sysset_t        sysset;         /* PCSENTRY, PCSEXIT */
        prgregset_t     prgregset;      /* PCSREG, PCAGENT */
        prfpregset_t    prfpregset;     /* PCSFPREG */
        prxregset_t     prxregset;      /* PCSXREG */
#endif /* ! codereview */
#if defined(__sparc)
        prxregset_t     prxregset;      /* PCSXREG */
        asrset_t        asrset;         /* PCSASRS */
#endif
        prwatch_t       prwatch;        /* PCWATCH */
        priovec_t       priovec;        /* PCREAD, PCWRITE */
        prcred_t        prcred;         /* PCSCRED */
        prpriv_t        prpriv;         /* PCSPRIV */
        long            przoneid;       /* PCSZONE */
} arg_t;

static  int     pr_control(long, arg_t *, prnode_t *, cred_t *);

static size_t
ctlsize(long cmd, size_t resid, arg_t *argp)
{
        size_t size = sizeof (long);
        size_t rnd;
        int ngrp;

        switch (cmd) {
        case PCNULL:
        case PCSTOP:
        case PCDSTOP:
        case PCWSTOP:
        case PCCSIG:
        case PCCFAULT:
                break;
        case PCSSIG:
                size += sizeof (siginfo_t);
                break;
        case PCTWSTOP:
                size += sizeof (long);
                break;
```

```
126         case PCKILL:
127         case PCUNKILL:
128         case PCNICE:
129                 size += sizeof (long);
130                 break;
131         case PCRUN:
132         case PCSET:
133         case PCUNSET:
134                 size += sizeof (ulong_t);
135                 break;
136         case PCSVADDR:
137                 size += sizeof (caddr_t);
138                 break;
139         case PCSTRACE:
140         case PCSHOLD:
141                 size += sizeof (sigset_t);
142                 break;
143         case PCSFAULT:
144                 size += sizeof (fltset_t);
145                 break;
146         case PCSENTRY:
147         case PCSEXIT:
148                 size += sizeof (sysset_t);
149                 break;
150         case PCSREG:
151         case PCAGENT:
152                 size += sizeof (prgregset_t);
153                 break;
154         case PCSFPREG:
155                 size += sizeof (prfpregset_t);
156                 break;
153 #if defined(__sparc)
157         case PCSXREG:
158                 size += sizeof (prxregset_t);
159                 break;
160 #if defined(__sparc)
161 #endif /* ! codereview */
162         case PCSASRS:
163                 size += sizeof (asrset_t);
164                 break;
165 #endif
166         case PCWATCH:
167                 size += sizeof (prwatch_t);
168                 break;
169         case PCREAD:
170         case PCWRITE:
171                 size += sizeof (priovec_t);
172                 break;
173         case PCSCRED:
174                 size += sizeof (prcred_t);
175                 break;
176         case PCSCREDX:
177                 /*
178                  * We cannot derefence the pr_ngroups fields if it
179                  * we don't have enough data.
180                  */
181                 if (resid < size + sizeof (prcred_t) - sizeof (gid_t))
182                         return (0);
183                 ngrp = argp->prcred.pr_ngroups;
184                 if (ngrp < 0 || ngrp > ngroups_max)
185                         return (0);

187                 /* The result can be smaller than sizeof (prcred_t) */
188                 size += sizeof (prcred_t) - sizeof (gid_t);
189                 size += ngrp * sizeof (gid_t);
190                 break;
```



```
191         case PCSPRIV:
192                 if (resid >= size + sizeof (prpriv_t))
193                         size += priv_prgetprivsize(&argp->prpriv);
194                 else
195                         return (0);
196                 break;
197         case PCSZONE:
198                 size += sizeof (long);
199                 break;
200         default:
201                 return (0);
202         }

204         /* Round up to a multiple of long, unless exact amount written */
205         if (size < resid) {
206                 rnd = size & (sizeof (long) - 1);

208                 if (rnd != 0)
209                         size += sizeof (long) - rnd;
210         }

212         if (size > resid)
213                 return (0);
214         return (size);
215 }

217 /*
218  * Control operations (lots).
219  */
220 int
221 prwritectl(vnode_t *vp, uio_t *uiop, cred_t *cr)
222 {
223 #define MY_BUFFER_SIZE \
224                 100 > 1 + sizeof (arg_t) / sizeof (long) ? \
225                 100 : 1 + sizeof (arg_t) / sizeof (long)
226         long buf[MY_BUFFER_SIZE];
227         long *bufp;
228         size_t resid = 0;
229         size_t size;
230         prnode_t *pnp = VTOP(vp);
231         int error;
232         int locked = 0;

234         while (uiop->uio_resid) {
235                 /*
236                  * Read several commands in one gulp.
237                  */
238                 bufp = buf;
239                 if (resid) {    /* move incomplete command to front of buffer */
240                         long *tail;

242                         if (resid >= sizeof (buf))
243                                 break;
244                         tail = (long *)((char *)buf + sizeof (buf) - resid);
245                         do {
246                                 *bufp++ = *tail++;
247                         } while ((resid -= sizeof (long)) != 0);
248                 }
249                 resid = sizeof (buf) - ((char *)bufp - (char *)buf);
250                 if (resid > uiop->uio_resid)
251                         resid = uiop->uio_resid;
252                 if (error = uiomove((caddr_t)bufp, resid, UIO_WRITE, uiop))
253                         return (error);
254                 resid += (char *)bufp - (char *)buf;
255                 bufp = buf;
```

```
257                 do {            /* loop over commands in buffer */
258                         long cmd = bufp[0];
259                         arg_t *argp = (arg_t *)&bufp[1];

261                         size = ctlsize(cmd, resid, argp);
262                         if (size == 0)  /* incomplete or invalid command */
263                                 break;
264                         /*
265                          * Perform the specified control operation.
266                          */
267                         if (!locked) {
268                                 if ((error = prlock(pnp, ZNO)) != 0)
269                                         return (error);
270                                 locked = 1;
271                         }
272                         if (error = pr_control(cmd, argp, pnp, cr)) {
273                                 if (error == -1)        /* -1 is timeout */
274                                         locked = 0;
275                                 else
276                                         return (error);
277                         }
278                         bufp = (long *)((char *)bufp + size);
279                 } while ((resid -= size) != 0);

281                 if (locked) {
282                         prunlock(pnp);
283                         locked = 0;
284                 }
285         }
286         return (resid? EINVAL : 0);
287 }

289 static int
290 pr_control(long cmd, arg_t *argp, prnode_t *pnp, cred_t *cr)
291 {
292         prcommon_t *pcp;
293         proc_t *p;
294         int unlocked;
295         int error = 0;

297         if (cmd == PCNULL)
298                 return (0);

300         pcp = pnp->pr_common;
301         p = pcp->prc_proc;
302         ASSERT(p != NULL);

304         /* System processes defy control. */
305         if (p->p_flag & SSYS) {
306                 prunlock(pnp);
307                 return (EBUSY);
308         }

310         switch (cmd) {

312         default:
313                 error = EINVAL;
314                 break;

316         case PCSTOP:    /* direct process or lwp to stop and wait for stop */
317         case PCDSTOP:   /* direct process or lwp to stop, don't wait */
318         case PCWSTOP:   /* wait for process or lwp to stop */
319         case PCTWSTOP:  /* wait for process or lwp to stop, with timeout */
320                 {
321                         time_t timeo;
```



```
323                         /*
324                          * Can't apply to a system process.
325                          */
326                         if (p->p_as == &kas) {
327                                 error = EBUSY;
328                                 break;
329                         }

331                         if (cmd == PCSTOP || cmd == PCDSTOP)
332                                 pr_stop(pnp);

334                         if (cmd == PCDSTOP)
335                                 break;

337                         /*
338                          * If an lwp is waiting for itself or its process,
339                          * don't wait. The stopped lwp would never see the
340                          * fact that it is stopped.
341                          */
342                         if ((pcp->prc_flags & PRC_LWP)?
343                             (pcp->prc_thread == curthread) : (p == curproc)) {
344                                 if (cmd == PCWSTOP || cmd == PCTWSTOP)
345                                         error = EBUSY;
346                                 break;
347                         }

349                         timeo = (cmd == PCTWSTOP)? (time_t)argp->timeo : 0;
350                         if ((error = pr_wait_stop(pnp, timeo)) != 0)
351                                 return (error);

353                         break;
354                 }

356         case PCRUN:     /* make lwp or process runnable */
357                 error = pr_setrun(pnp, argp->flags);
358                 break;

360         case PCSTRACE:  /* set signal trace mask */
361                 pr_settrace(p,  &argp->sigset);
362                 break;

364         case PCSSIG:    /* set current signal */
365                 error = pr_setsig(pnp, &argp->siginfo);
366                 if (argp->siginfo.si_signo == SIGKILL && error == 0) {
367                         prunlock(pnp);
368                         pr_wait_die(pnp);
369                         return (-1);
370                 }
371                 break;

373         case PCKILL:    /* send signal */
374                 error = pr_kill(pnp, (int)argp->sig, cr);
375                 if (error == 0 && argp->sig == SIGKILL) {
376                         prunlock(pnp);
377                         pr_wait_die(pnp);
378                         return (-1);
379                 }
380                 break;

382         case PCUNKILL:  /* delete a pending signal */
383                 error = pr_unkill(pnp, (int)argp->sig);
384                 break;

386         case PCNICE:    /* set nice priority */
387                 error = pr_nice(p, (int)argp->nice, cr);
388                 break;
```

```
390         case PCSENTRY:  /* set syscall entry bit mask */
391         case PCSEXIT:   /* set syscall exit bit mask */
392                 pr_setentryexit(p, &argp->sysset, cmd == PCSENTRY);
393                 break;

395         case PCSET:     /* set process flags */
396                 error = pr_set(p, argp->flags);
397                 break;

399         case PCUNSET:   /* unset process flags */
400                 error = pr_unset(p, argp->flags);
401                 break;

403         case PCSREG:    /* set general registers */
404                 {
405                         kthread_t *t = pr_thread(pnp);

407                         if (!ISTOPPED(t) && !VSTOPPED(t) && !DSTOPPED(t)) {
408                                 thread_unlock(t);
409                                 error = EBUSY;
410                         } else {
411                                 thread_unlock(t);
412                                 mutex_exit(&p->p_lock);
413                                 prsetprregs(ttolwp(t), argp->prgregset, 0);
414                                 mutex_enter(&p->p_lock);
415                         }
416                         break;
417                 }

419         case PCSFPREG:  /* set floating-point registers */
420                 error = pr_setfpregs(pnp, &argp->prfpregset);
421                 break;

423         case PCSXREG:   /* set extra registers */
157 #if defined(__sparc)
424                 error = pr_setxregs(pnp, &argp->prxregset);
159 #else
160                 error = EINVAL;
161 #endif
425                 break;

427 #if defined(__sparc)
428         case PCSASRS:   /* set ancillary state registers */
429                 error = pr_setasrs(pnp, argp->asrset);
430                 break;
431 #endif

433         case PCSVADDR:  /* set virtual address at which to resume */
434                 error = pr_setvaddr(pnp, argp->vaddr);
435                 break;

437         case PCSHOLD:   /* set signal-hold mask */
438                 pr_sethold(pnp, &argp->sigset);
439                 break;

441         case PCSFAULT:  /* set mask of traced faults */
442                 pr_setfault(p, &argp->fltset);
443                 break;

445         case PCCSIG:    /* clear current signal */
446                 error = pr_clearsig(pnp);
447                 break;

449         case PCCFAULT:  /* clear current fault */
450                 error = pr_clearflt(pnp);
```



```
451                 break;

453         case PCWATCH:   /* set or clear watched areas */
454                 error = pr_watch(pnp, &argp->prwatch, &unlocked);
455                 if (error && unlocked)
456                         return (error);
457                 break;

459         case PCAGENT:   /* create the /proc agent lwp in the target process */
460                 error = pr_agent(pnp, argp->prgregset, &unlocked);
461                 if (error && unlocked)
462                         return (error);
463                 break;

465         case PCREAD:    /* read from the address space */
466                 error = pr_rdwr(p, UIO_READ, &argp->priovec);
467                 break;

469         case PCWRITE:   /* write to the address space */
470                 error = pr_rdwr(p, UIO_WRITE, &argp->priovec);
471                 break;

473         case PCSCRED:   /* set the process credentials */
474         case PCSCREDX:
475                 error = pr_scred(p, &argp->prcred, cr, cmd == PCSCREDX);
476                 break;

478         case PCSPRIV:   /* set the process privileges */
479                 error = pr_spriv(p, &argp->prpriv, cr);
480                 break;
481         case PCSZONE:   /* set the process's zoneid credentials */
482                 error = pr_szoneid(p, (zoneid_t)argp->przoneid, cr);
483                 break;
484         }

486         if (error)
487                 prunlock(pnp);
488         return (error);
489 }

491 #ifdef _SYSCALL32_IMPL

493 typedef union {
494         int32_t         sig;            /* PCKILL, PCUNKILL */
495         int32_t         nice;           /* PCNICE */
496         int32_t         timeo;          /* PCTWSTOP */
497         uint32_t        flags;          /* PCRUN, PCSET, PCUNSET */
498         caddr32_t       vaddr;          /* PCSVADDR */
499         siginfo32_t     siginfo;        /* PCSSIG */
500         sigset_t        sigset;         /* PCSTRACE, PCSHOLD */
501         fltset_t        fltset;         /* PCSFAULT */
502         sysset_t        sysset;         /* PCSENTRY, PCSEXIT */
503         prgregset32_t   prgregset;      /* PCSREG, PCAGENT */
504         prfpregset32_t  prfpregset;     /* PCSFPREG */
242 #if defined(__sparc)
505         prxregset_t     prxregset;      /* PCSXREG */
244 #endif
506         prwatch32_t     prwatch;        /* PCWATCH */
507         priovec32_t     priovec;        /* PCREAD, PCWRITE */
508         prcred32_t      prcred;         /* PCSCRED */
509         prpriv_t        prpriv;         /* PCSPRIV */
510         int32_t         przoneid;       /* PCSZONE */
511 } arg32_t;

513 static  int     pr_control32(int32_t, arg32_t *, prnode_t *, cred_t *);
514 static  int     pr_setfpregs32(prnode_t *, prfpregset32_t *);
```

```
516 /*
517  * Note that while ctlsize32() can use argp, it must do so only in a way
518  * that assumes 32-bit rather than 64-bit alignment as argp is a pointer
519  * to an array of 32-bit values and only 32-bit alignment is ensured.
520  */
521 static size_t
522 ctlsize32(int32_t cmd, size_t resid, arg32_t *argp)
523 {
524         size_t size = sizeof (int32_t);
525         size_t rnd;
526         int ngrp;

528         switch (cmd) {
529         case PCNULL:
530         case PCSTOP:
531         case PCDSTOP:
532         case PCWSTOP:
533         case PCCSIG:
534         case PCCFAULT:
535                 break;
536         case PCSSIG:
537                 size += sizeof (siginfo32_t);
538                 break;
539         case PCTWSTOP:
540                 size += sizeof (int32_t);
541                 break;
542         case PCKILL:
543         case PCUNKILL:
544         case PCNICE:
545                 size += sizeof (int32_t);
546                 break;
547         case PCRUN:
548         case PCSET:
549         case PCUNSET:
550                 size += sizeof (uint32_t);
551                 break;
552         case PCSVADDR:
553                 size += sizeof (caddr32_t);
554                 break;
555         case PCSTRACE:
556         case PCSHOLD:
557                 size += sizeof (sigset_t);
558                 break;
559         case PCSFAULT:
560                 size += sizeof (fltset_t);
561                 break;
562         case PCSENTRY:
563         case PCSEXIT:
564                 size += sizeof (sysset_t);
565                 break;
566         case PCSREG:
567         case PCAGENT:
568                 size += sizeof (prgregset32_t);
569                 break;
570         case PCSFPREG:
571                 size += sizeof (prfpregset32_t);
572                 break;
312 #if defined(__sparc)
573         case PCSXREG:
574                 size += sizeof (prxregset_t);
575                 break;
316 #endif
576         case PCWATCH:
577                 size += sizeof (prwatch32_t);
578                 break;
```



```
579         case PCREAD:
580         case PCWRITE:
581                 size += sizeof (priovec32_t);
582                 break;
583         case PCSCRED:
584                 size += sizeof (prcred32_t);
585                 break;
586         case PCSCREDX:
587                 /*
588                  * We cannot derefence the pr_ngroups fields if it
589                  * we don't have enough data.
590                  */
591                 if (resid < size + sizeof (prcred32_t) - sizeof (gid32_t))
592                         return (0);
593                 ngrp = argp->prcred.pr_ngroups;
594                 if (ngrp < 0 || ngrp > ngroups_max)
595                         return (0);

597                 /* The result can be smaller than sizeof (prcred32_t) */
598                 size += sizeof (prcred32_t) - sizeof (gid32_t);
599                 size += ngrp * sizeof (gid32_t);
600                 break;
601         case PCSPRIV:
602                 if (resid >= size + sizeof (prpriv_t))
603                         size += priv_prgetprivsize(&argp->prpriv);
604                 else
605                         return (0);
606                 break;
607         case PCSZONE:
608                 size += sizeof (int32_t);
609                 break;
610         default:
611                 return (0);
612         }

614         /* Round up to a multiple of int32_t */
615         rnd = size & (sizeof (int32_t) - 1);

617         if (rnd != 0)
618                 size += sizeof (int32_t) - rnd;

620         if (size > resid)
621                 return (0);
622         return (size);
623 }
_____unchanged_portion_omitted_

711 static int
712 pr_control32(int32_t cmd, arg32_t *argp, prnode_t *pnp, cred_t *cr)
713 {
714         prcommon_t *pcp;
715         proc_t *p;
716         int unlocked;
717         int error = 0;

719         if (cmd == PCNULL)
720                 return (0);

722         pcp = pnp->pr_common;
723         p = pcp->prc_proc;
724         ASSERT(p != NULL);

726         if (p->p_flag & SSYS) {
727                 prunlock(pnp);
728                 return (EBUSY);
729         }
```

```
731         switch (cmd) {
733         default:
734                 error = EINVAL;
735                 break;
737         case PCSTOP:    /* direct process or lwp to stop and wait for stop */
738         case PCDSTOP:   /* direct process or lwp to stop, don't wait */
739         case PCWSTOP:   /* wait for process or lwp to stop */
740         case PCTWSTOP:  /* wait for process or lwp to stop, with timeout */
741                 {
742                         time_t timeo;
744                         /*
745                          * Can't apply to a system process.
746                          */
747                         if (p->p_as == &kas) {
748                                 error = EBUSY;
749                                 break;
750                         }
752                         if (cmd == PCSTOP || cmd == PCDSTOP)
753                                 pr_stop(pnp);
755                         if (cmd == PCDSTOP)
756                                 break;
758                         /*
759                          * If an lwp is waiting for itself or its process,
760                          * don't wait. The lwp will never see the fact that
761                          * itself is stopped.
762                          */
763                         if ((pcp->prc_flags & PRC_LWP)?
764                             (pcp->prc_thread == curthread) : (p == curproc)) {
765                                 if (cmd == PCWSTOP || cmd == PCTWSTOP)
766                                         error = EBUSY;
767                                 break;
768                         }
770                         timeo = (cmd == PCTWSTOP)? (time_t)argp->timeo : 0;
771                         if ((error = pr_wait_stop(pnp, timeo)) != 0)
772                                 return (error);
774                         break;
775                 }
777         case PCRUN:     /* make lwp or process runnable */
778                 error = pr_setrun(pnp, (ulong_t)argp->flags);
779                 break;
781         case PCSTRACE:  /* set signal trace mask */
782                 pr_settrace(p,  &argp->sigset);
783                 break;
785         case PCSSIG:    /* set current signal */
786                 if (PROCESS_NOT_32BIT(p))
787                         error = EOVERFLOW;
788                 else {
789                         int sig = (int)argp->siginfo.si_signo;
790                         siginfo_t siginfo;
792                         bzero(&siginfo, sizeof (siginfo));
793                         siginfo_32tok(&argp->siginfo, (k_siginfo_t *)&siginfo);
794                         error = pr_setsig(pnp, &siginfo);
795                         if (sig == SIGKILL && error == 0) {
```



```
796                                 prunlock(pnp);
797                                 pr_wait_die(pnp);
798                                 return (-1);
799                         }
800                 }
801                 break;
803         case PCKILL:    /* send signal */
804                 error = pr_kill(pnp, (int)argp->sig, cr);
805                 if (error == 0 && argp->sig == SIGKILL) {
806                         prunlock(pnp);
807                         pr_wait_die(pnp);
808                         return (-1);
809                 }
810                 break;
812         case PCUNKILL:  /* delete a pending signal */
813                 error = pr_unkill(pnp, (int)argp->sig);
814                 break;
816         case PCNICE:    /* set nice priority */
817                 error = pr_nice(p, (int)argp->nice, cr);
818                 break;
820         case PCSENTRY:  /* set syscall entry bit mask */
821         case PCSEXIT:   /* set syscall exit bit mask */
822                 pr_setentryexit(p, &argp->sysset, cmd == PCSENTRY);
823                 break;
825         case PCSET:     /* set process flags */
826                 error = pr_set(p, (long)argp->flags);
827                 break;
829         case PCUNSET:   /* unset process flags */
830                 error = pr_unset(p, (long)argp->flags);
831                 break;
833         case PCSREG:    /* set general registers */
834                 if (PROCESS_NOT_32BIT(p))
835                         error = EOVERFLOW;
836                 else {
837                         kthread_t *t = pr_thread(pnp);
839                         if (!ISTOPPED(t) && !VSTOPPED(t) && !DSTOPPED(t)) {
840                                 thread_unlock(t);
841                                 error = EBUSY;
842                         } else {
843                                 prgregset_t prgregset;
844                                 klwp_t *lwp = ttolwp(t);
846                                 thread_unlock(t);
847                                 mutex_exit(&p->p_lock);
848                                 prgregset_32ton(lwp, argp->prgregset,
849                                     prgregset);
850                                 prsetprregs(lwp, prgregset, 0);
851                                 mutex_enter(&p->p_lock);
852                         }
853                 }
854                 break;
856         case PCSFPREG:  /* set floating-point registers */
857                 if (PROCESS_NOT_32BIT(p))
858                         error = EOVERFLOW;
859                 else
860                         error = pr_setfpregs32(pnp, &argp->prfpregset);
861                 break;
```

```
863         case PCSXREG:   /* set extra registers */
605 #if defined(__sparc)
864                 if (PROCESS_NOT_32BIT(p))
865                         error = EOVERFLOW;
866                 else
867                         error = pr_setxregs(pnp, &argp->prxregset);
610 #else
611                 error = EINVAL;
612 #endif
868                 break;

870         case PCSVADDR:  /* set virtual address at which to resume */
871                 if (PROCESS_NOT_32BIT(p))
872                         error = EOVERFLOW;
873                 else
874                         error = pr_setvaddr(pnp,
875                             (caddr_t)(uintptr_t)argp->vaddr);
876                 break;

878         case PCSHOLD:   /* set signal-hold mask */
879                 pr_sethold(pnp, &argp->sigset);
880                 break;

882         case PCSFAULT:  /* set mask of traced faults */
883                 pr_setfault(p, &argp->fltset);
884                 break;

886         case PCCSIG:    /* clear current signal */
887                 error = pr_clearsig(pnp);
888                 break;

890         case PCCFAULT:  /* clear current fault */
891                 error = pr_clearflt(pnp);
892                 break;

894         case PCWATCH:   /* set or clear watched areas */
895                 if (PROCESS_NOT_32BIT(p))
896                         error = EOVERFLOW;
897                 else {
898                         prwatch_t prwatch;

900                         prwatch.pr_vaddr = argp->prwatch.pr_vaddr;
901                         prwatch.pr_size = argp->prwatch.pr_size;
902                         prwatch.pr_wflags = argp->prwatch.pr_wflags;
903                         prwatch.pr_pad = argp->prwatch.pr_pad;
904                         error = pr_watch(pnp, &prwatch, &unlocked);
905                         if (error && unlocked)
906                                 return (error);
907                 }
908                 break;

910         case PCAGENT:   /* create the /proc agent lwp in the target process */
911                 if (PROCESS_NOT_32BIT(p))
912                         error = EOVERFLOW;
913                 else {
914                         prgregset_t prgregset;
915                         kthread_t *t = pr_thread(pnp);
916                         klwp_t *lwp = ttolwp(t);
917                         thread_unlock(t);
918                         mutex_exit(&p->p_lock);
919                         prgregset_32ton(lwp, argp->prgregset, prgregset);
920                         mutex_enter(&p->p_lock);
921                         error = pr_agent(pnp, prgregset, &unlocked);
922                         if (error && unlocked)
923                                 return (error);
```



```
924                 }
925                 break;

927         case PCREAD:    /* read from the address space */
928         case PCWRITE:   /* write to the address space */
929                 if (PROCESS_NOT_32BIT(p))
930                         error = EOVERFLOW;
931                 else {
932                         enum uio_rw rw = (cmd == PCREAD)? UIO_READ : UIO_WRITE;
933                         priovec_t priovec;

935                         priovec.pio_base =
936                             (void *)(uintptr_t)argp->priovec.pio_base;
937                         priovec.pio_len = (size_t)argp->priovec.pio_len;
938                         priovec.pio_offset = (off_t)
939                             (uint32_t)argp->priovec.pio_offset;
940                         error = pr_rdwr(p, rw, &priovec);
941                 }
942                 break;

944         case PCSCRED:   /* set the process credentials */
945         case PCSCREDX:
946                 {
947                         /*
948                          * All the fields in these structures are exactly the
949                          * same and so the structures are compatible.  In case
950                          * this ever changes, we catch this with the ASSERT
951                          * below.
952                          */
953                         prcred_t *prcred = (prcred_t *)&argp->prcred;

955 #ifndef __lint
956                         ASSERT(sizeof (prcred_t) == sizeof (prcred32_t));
957 #endif

959                         error = pr_scred(p, prcred, cr, cmd == PCSCREDX);
960                         break;
961                 }

963         case PCSPRIV:   /* set the process privileges */
964                 error = pr_spriv(p, &argp->prpriv, cr);
965                 break;

967         case PCSZONE:   /* set the process's zoneid */
968                 error = pr_szoneid(p, (zoneid_t)argp->przoneid, cr);
969                 break;
970         }

972         if (error)
973                 prunlock(pnp);
974         return (error);
975 }
_____unchanged_portion_omitted_
1698 #endif  /* _SYSCALL32_IMPL */

1445 #if defined(__sparc)
1700 /* ARGSUSED */
1701 static int
1702 pr_setxregs(prnode_t *pnp, prxregset_t *prxregset)
1703 {
1704         proc_t *p = pnp->pr_common->prc_proc;
1705         kthread_t *t = pr_thread(pnp);  /* returns locked thread */

1707         if (!ISTOPPED(t) && !VSTOPPED(t) && !DSTOPPED(t)) {
1708                 thread_unlock(t);
1709                 return (EBUSY);
```

```
1710         }
1711         thread_unlock(t);

1713         if (!prhasx(p))
1714                 return (EINVAL);        /* No extra register support */

1716         /* drop p_lock while touching the lwp's stack */
1717         mutex_exit(&p->p_lock);
1718         prsetprxregs(ttolwp(t), (caddr_t)prxregset);
1719         mutex_enter(&p->p_lock);

1721         return (0);
1722 }

1724 #if defined(__sparc)
1725 #endif /* ! codereview */
1726 static int
1727 pr_setasrs(prnode_t *pnp, asrset_t asrset)
1728 {
1729         proc_t *p = pnp->pr_common->prc_proc;
1730         kthread_t *t = pr_thread(pnp);  /* returns locked thread */

1732         if (!ISTOPPED(t) && !VSTOPPED(t) && !DSTOPPED(t)) {
1733                 thread_unlock(t);
1734                 return (EBUSY);
1735         }
1736         thread_unlock(t);

1738         /* drop p_lock while touching the lwp's stack */
1739         mutex_exit(&p->p_lock);
1740         prsetasregs(ttolwp(t), asrset);
1741         mutex_enter(&p->p_lock);

1743         return (0);
1744 }
1745 #endif

1747 static int
1748 pr_setvaddr(prnode_t *pnp, caddr_t vaddr)
1749 {
1750         proc_t *p = pnp->pr_common->prc_proc;
1751         kthread_t *t = pr_thread(pnp);  /* returns locked thread */

1753         if (!ISTOPPED(t) && !VSTOPPED(t) && !DSTOPPED(t)) {
1754                 thread_unlock(t);
1755                 return (EBUSY);
1756         }

1758         /* drop p_lock while touching the lwp's stack */
1759         thread_unlock(t);
1760         mutex_exit(&p->p_lock);
1761         prsvaddr(ttolwp(t), vaddr);
1762         mutex_enter(&p->p_lock);

1764         return (0);
1765 }

1767 void
1768 pr_sethold(prnode_t *pnp, sigset_t *sp)
1769 {
1770         proc_t *p = pnp->pr_common->prc_proc;
1771         kthread_t *t = pr_thread(pnp);  /* returns locked thread */

1773         schedctl_finish_sigblock(t);
1774         sigutok(sp, &t->t_hold);
1775         if (ISWAKEABLE(t) &&
```



```
1776             (fsig(&p->p_sig, t) || fsig(&t->t_sig, t)))
1777                 setrun_locked(t);
1778         t->t_sig_check = 1;     /* so thread will see new holdmask */
1779         thread_unlock(t);
1780 }

1782 void
1783 pr_setfault(proc_t *p, fltset_t *fltp)
1784 {
1785         prassignset(&p->p_fltmask, fltp);
1786         if (!prisempty(&p->p_fltmask))
1787                 p->p_proc_flag |= P_PR_TRACE;
1788         else if (sigisempty(&p->p_sigmask)) {
1789                 user_t *up = PTOU(p);
1790                 if (up->u_systrap == 0)
1791                         p->p_proc_flag &= ~P_PR_TRACE;
1792         }
1793 }

1795 static int
1796 pr_clearsig(prnode_t *pnp)
1797 {
1798         kthread_t *t = pr_thread(pnp);  /* returns locked thread */
1799         klwp_t *lwp = ttolwp(t);

1801         thread_unlock(t);
1802         if (lwp->lwp_cursig == SIGKILL)
1803                 return (EBUSY);

1805         /*
1806          * Discard current siginfo_t, if any.
1807          */
1808         lwp->lwp_cursig = 0;
1809         lwp->lwp_extsig = 0;
1810         if (lwp->lwp_curinfo) {
1811                 siginfofree(lwp->lwp_curinfo);
1812                 lwp->lwp_curinfo = NULL;
1813         }

1815         return (0);
1816 }

1818 static int
1819 pr_clearflt(prnode_t *pnp)
1820 {
1821         kthread_t *t = pr_thread(pnp);  /* returns locked thread */

1823         thread_unlock(t);
1824         ttolwp(t)->lwp_curflt = 0;

1826         return (0);
1827 }

1829 static int
1830 pr_watch(prnode_t *pnp, prwatch_t *pwp, int *unlocked)
1831 {
1832         proc_t *p = pnp->pr_common->prc_proc;
1833         struct as *as = p->p_as;
1834         uintptr_t vaddr = pwp->pr_vaddr;
1835         size_t size = pwp->pr_size;
1836         int wflags = pwp->pr_wflags;
1837         ulong_t newpage = 0;
1838         struct watched_area *pwa;
1839         int error;

1841         *unlocked = 0;
```

```
1843         /*
1844          * Can't apply to a system process.
1845          */
1846         if ((p->p_flag & SSYS) || p->p_as == &kas)
1847                 return (EBUSY);
1848
1849         /*
1850          * Verify that the address range does not wrap
1851          * and that only the proper flags were specified.
1852          */
1853         if ((wflags & ~WA_TRAPAFTER) == 0)
1854                 size = 0;
1855         if (vaddr + size < vaddr ||
1856             (wflags & ~(WA_READ|WA_WRITE|WA_EXEC|WA_TRAPAFTER)) != 0 ||
1857             ((wflags & ~WA_TRAPAFTER) != 0 && size == 0))
1858                 return (EINVAL);
1859
1860         /*
1861          * Don't let the address range go above as->a_userlimit.
1862          * There is no error here, just a limitation.
1863          */
1864         if (vaddr >= (uintptr_t)as->a_userlimit)
1865                 return (0);
1866         if (vaddr + size > (uintptr_t)as->a_userlimit)
1867                 size = (uintptr_t)as->a_userlimit - vaddr;
1868
1869         /*
1870          * Compute maximum number of pages this will add.
1871          */
1872         if ((wflags & ~WA_TRAPAFTER) != 0) {
1873                 ulong_t pagespan = (vaddr + size) - (vaddr & PAGEMASK);
1874                 newpage = btopr(pagespan);
1875                 if (newpage > 2 * prnwatch)
1876                         return (E2BIG);
1877         }
1878
1879         /*
1880          * Force the process to be fully stopped.
1881          */
1882         if (p == curproc) {
1883                 prunlock(pnp);
1884                 while (holdwatch() != 0)
1885                         continue;
1886                 if ((error = prlock(pnp, ZNO)) != 0) {
1887                         continuelwps(p);
1888                         *unlocked = 1;
1889                         return (error);
1890                 }
1891         } else {
1892                 pauselwps(p);
1893                 while (pr_allstopped(p, 0) > 0) {
1894                         /*
1895                          * This cv/mutex pair is persistent even
1896                          * if the process disappears after we
1897                          * unmark it and drop p->p_lock.
1898                          */
1899                         kcondvar_t *cv = &pr_pid_cv[p->p_slot];
1900                         kmutex_t *mp = &p->p_lock;
1901
1902                         prunmark(p);
1903                         (void) cv_wait(cv, mp);
1904                         mutex_exit(mp);
1905                         if ((error = prlock(pnp, ZNO)) != 0) {
1906                                 /*
1907                                  * Unpause the process if it exists.
```



```
1908                                  */
1909                                 p = pr_p_lock(pnp);
1910                                 mutex_exit(&pr_pidlock);
1911                                 if (p != NULL) {
1912                                         unpauselwps(p);
1913                                         prunlock(pnp);
1914                                 }
1915                                 *unlocked = 1;
1916                                 return (error);
1917                         }
1918                 }
1919         }
1920
1921         /*
1922          * Drop p->p_lock in order to perform the rest of this.
1923          * The process is still locked with the P_PR_LOCK flag.
1924          */
1925         mutex_exit(&p->p_lock);
1926
1927         pwa = kmem_alloc(sizeof (struct watched_area), KM_SLEEP);
1928         pwa->wa_vaddr = (caddr_t)vaddr;
1929         pwa->wa_eaddr = (caddr_t)vaddr + size;
1930         pwa->wa_flags = (ulong_t)wflags;
1931
1932         error = ((pwa->wa_flags & ~WA_TRAPAFTER) == 0)?
1933             clear_watched_area(p, pwa) : set_watched_area(p, pwa);
1934
1935         if (p == curproc) {
1936                 setallwatch();
1937                 mutex_enter(&p->p_lock);
1938                 continuelwps(p);
1939         } else {
1940                 mutex_enter(&p->p_lock);
1941                 unpauselwps(p);
1942         }
1943
1944         return (error);
1945 }
1946
1947 /* jobcontrol stopped, but with a /proc directed stop in effect */
1948 #define JDSTOPPED(t)    \
1949         ((t)->t_state == TS_STOPPED && \
1950         (t)->t_whystop == PR_JOBCONTROL && \
1951         ((t)->t_proc_flag & TP_PRSTOP))
1952
1953 /*
1954  * pr_agent() creates the agent lwp. If the process is exiting while
1955  * we are creating an agent lwp, then exitlwps() waits until the
1956  * agent has been created using prbarrier().
1957  */
1958 static int
1959 pr_agent(prnode_t *pnp, prgregset_t prgregset, int *unlocked)
1960 {
1961         proc_t *p = pnp->pr_common->prc_proc;
1962         prcommon_t *pcp;
1963         kthread_t *t;
1964         kthread_t *ct;
1965         klwp_t *clwp;
1966         k_sigset_t smask;
1967         int cid;
1968         void *bufp = NULL;
1969         int error;
1970
1971         *unlocked = 0;
1972
1973         /*
```

```
1974          * Cannot create the /proc agent lwp if :-
1975          * - the process is not fully stopped or directed to stop.
1976          * - there is an agent lwp already.
1977          * - the process has been killed.
1978          * - the process is exiting.
1979          * - it's a vfork(2) parent.
1980          */
1981         t = prchoose(p);        /* returns locked thread */
1982         ASSERT(t != NULL);

1984         if ((!ISTOPPED(t) && !VSTOPPED(t) && !SUSPENDED(t) && !JDSTOPPED(t)) ||
1985             p->p_agenttp != NULL ||
1986             (p->p_flag & (SKILLED | SEXITING | SVFWAIT))) {
1987                 thread_unlock(t);
1988                 return (EBUSY);
1989         }

1991         thread_unlock(t);
1992         mutex_exit(&p->p_lock);

1994         sigfillset(&smask);
1995         sigdiffset(&smask, &cantmask);
1996         clwp = lwp_create(lwp_rtt, NULL, 0, p, TS_STOPPED,
1997             t->t_pri, &smask, NOCLASS, 0);
1998         if (clwp == NULL) {
1999                 mutex_enter(&p->p_lock);
2000                 return (ENOMEM);
2001         }
2002         prsetprregs(clwp, prgregset, 1);
2003 retry:
2004         cid = t->t_cid;
2005         (void) CL_ALLOC(&bufp, cid, KM_SLEEP);
2006         mutex_enter(&p->p_lock);
2007         if (cid != t->t_cid) {
2008                 /*
2009                  * Someone just changed this thread's scheduling class,
2010                  * so try pre-allocating the buffer again.  Hopefully we
2011                  * don't hit this often.
2012                  */
2013                 mutex_exit(&p->p_lock);
2014                 CL_FREE(cid, bufp);
2015                 goto retry;
2016         }

2018         clwp->lwp_ap = clwp->lwp_arg;
2019         clwp->lwp_eosys = NORMALRETURN;
2020         ct = lwptot(clwp);
2021         ct->t_clfuncs = t->t_clfuncs;
2022         CL_FORK(t, ct, bufp);
2023         ct->t_cid = t->t_cid;
2024         ct->t_proc_flag |= TP_PRSTOP;
2025         /*
2026          * Setting t_sysnum to zero causes post_syscall()
2027          * to bypass all syscall checks and go directly to
2028          *      if (issig()) psig();
2029          * so that the agent lwp will stop in issig_forreal()
2030          * showing PR_REQUESTED.
2031          */
2032         ct->t_sysnum = 0;
2033         ct->t_post_sys = 1;
2034         ct->t_sig_check = 1;
2035         p->p_agenttp = ct;
2036         ct->t_proc_flag &= ~TP_HOLDLWP;

2038         pcp = pnp->pr_pcommon;
2039         mutex_enter(&pcp->prc_mutex);
```



```
2041         lwp_create_done(ct);

2043         /*
2044          * Don't return until the agent is stopped on PR_REQUESTED.
2045          */

2047         for (;;) {
2048                 prunlock(pnp);
2049                 *unlocked = 1;

2051                 /*
2052                  * Wait for the agent to stop and notify us.
2053                  * If we've been interrupted, return that information.
2054                  */
2055                 error = pr_wait(pcp, NULL, 0);
2056                 if (error == EINTR) {
2057                         error = 0;
2058                         break;
2059                 }

2061                 /*
2062                  * Confirm that the agent LWP has stopped.
2063                  */

2065                 if ((error = prlock(pnp, ZNO)) != 0)
2066                         break;
2067                 *unlocked = 0;

2069                 /*
2070                  * Since we dropped the lock on the process, the agent
2071                  * may have disappeared or changed. Grab the current
2072                  * agent and check fail if it has disappeared.
2073                  */
2074                 if ((ct = p->p_agenttp) == NULL) {
2075                         error = ENOENT;
2076                         break;
2077                 }

2079                 mutex_enter(&pcp->prc_mutex);
2080                 thread_lock(ct);

2082                 if (ISTOPPED(ct)) {
2083                         thread_unlock(ct);
2084                         mutex_exit(&pcp->prc_mutex);
2085                         break;
2086                 }

2088                 thread_unlock(ct);
2089         }

2091         return (error ? error : -1);
2092 }

2094 static int
2095 pr_rdwr(proc_t *p, enum uio_rw rw, priovec_t *pio)
2096 {
2097         caddr_t base = (caddr_t)pio->pio_base;
2098         size_t cnt = pio->pio_len;
2099         uintptr_t offset = (uintptr_t)pio->pio_offset;
2100         struct uio auio;
2101         struct iovec aiov;
2102         int error = 0;

2104         if ((p->p_flag & SSYS) || p->p_as == &kas)
2105                 error = EIO;
```

```
2106         else if ((base + cnt) < base || (offset + cnt) < offset)
2107                 error = EINVAL;
2108         else if (cnt != 0) {
2109                 aiov.iov_base = base;
2110                 aiov.iov_len = cnt;

2112                 auio.uio_loffset = offset;
2113                 auio.uio_iov = &aiov;
2114                 auio.uio_iovcnt = 1;
2115                 auio.uio_resid = cnt;
2116                 auio.uio_segflg = UIO_USERSPACE;
2117                 auio.uio_llimit = (longlong_t)MAXOFFSET_T;
2118                 auio.uio_fmode = FREAD|FWRITE;
2119                 auio.uio_extflg = UIO_COPY_DEFAULT;

2121                 mutex_exit(&p->p_lock);
2122                 error = prusrio(p, rw, &auio, 0);
2123                 mutex_enter(&p->p_lock);

2125                 /*
2126                  * We have no way to return the i/o count,
2127                  * like read() or write() would do, so we
2128                  * return an error if the i/o was truncated.
2129                  */
2130                 if (auio.uio_resid != 0 && error == 0)
2131                         error = EIO;
2132         }

2134         return (error);
2135 }

2137 static int
2138 pr_scred(proc_t *p, prcred_t *prcred, cred_t *cr, boolean_t dogrps)
2139 {
2140         kthread_t *t;
2141         cred_t *oldcred;
2142         cred_t *newcred;
2143         uid_t oldruid;
2144         int error;
2145         zone_t *zone = crgetzone(cr);

2147         if (!VALID_UID(prcred->pr_euid, zone) ||
2148             !VALID_UID(prcred->pr_ruid, zone) ||
2149             !VALID_UID(prcred->pr_suid, zone) ||
2150             !VALID_GID(prcred->pr_egid, zone) ||
2151             !VALID_GID(prcred->pr_rgid, zone) ||
2152             !VALID_GID(prcred->pr_sgid, zone))
2153                 return (EINVAL);

2155         if (dogrps) {
2156                 int ngrp = prcred->pr_ngroups;
2157                 int i;

2159                 if (ngrp < 0 || ngrp > ngroups_max)
2160                         return (EINVAL);

2162                 for (i = 0; i < ngrp; i++) {
2163                         if (!VALID_GID(prcred->pr_groups[i], zone))
2164                                 return (EINVAL);
2165                 }
2166         }

2168         error = secpolicy_allow_setid(cr, prcred->pr_euid, B_FALSE);

2170         if (error == 0 && prcred->pr_ruid != prcred->pr_euid)
2171                 error = secpolicy_allow_setid(cr, prcred->pr_ruid, B_FALSE);
```



```
2173         if (error == 0 && prcred->pr_suid != prcred->pr_euid &&
2174             prcred->pr_suid != prcred->pr_ruid)
2175                 error = secpolicy_allow_setid(cr, prcred->pr_suid, B_FALSE);

2177         if (error)
2178                 return (error);

2180         mutex_exit(&p->p_lock);

2182         /* hold old cred so it doesn't disappear while we dup it */
2183         mutex_enter(&p->p_crlock);
2184         crhold(oldcred = p->p_cred);
2185         mutex_exit(&p->p_crlock);
2186         newcred = crdup(oldcred);
2187         oldruid = crgetruid(oldcred);
2188         crfree(oldcred);

2190         /* Error checking done above */
2191         (void) crsetresuid(newcred, prcred->pr_ruid, prcred->pr_euid,
2192             prcred->pr_suid);
2193         (void) crsetresgid(newcred, prcred->pr_rgid, prcred->pr_egid,
2194             prcred->pr_sgid);

2196         if (dogrps) {
2197                 (void) crsetgroups(newcred, prcred->pr_ngroups,
2198                     prcred->pr_groups);

2200         }

2202         mutex_enter(&p->p_crlock);
2203         oldcred = p->p_cred;
2204         p->p_cred = newcred;
2205         mutex_exit(&p->p_crlock);
2206         crfree(oldcred);

2208         /*
2209          * Keep count of processes per uid consistent.
2210          */
2211         if (oldruid != prcred->pr_ruid) {
2212                 zoneid_t zoneid = crgetzoneid(newcred);

2214                 mutex_enter(&pidlock);
2215                 upcount_dec(oldruid, zoneid);
2216                 upcount_inc(prcred->pr_ruid, zoneid);
2217                 mutex_exit(&pidlock);
2218         }

2220         /*
2221          * Broadcast the cred change to the threads.
2222          */
2223         mutex_enter(&p->p_lock);
2224         t = p->p_tlist;
2225         do {
2226                 t->t_pre_sys = 1; /* so syscall will get new cred */
2227         } while ((t = t->t_forw) != p->p_tlist);

2229         return (0);
2230 }

2232 /*
2233  * Change process credentials to specified zone.  Used to temporarily
2234  * set a process to run in the global zone; only transitions between
2235  * the process's actual zone and the global zone are allowed.
2236  */
2237 static int
```

```
2238 pr_szoneid(proc_t *p, zoneid_t zoneid, cred_t *cr)
2239 {
2240         kthread_t *t;
2241         cred_t *oldcred;
2242         cred_t *newcred;
2243         zone_t *zptr;
2244         zoneid_t oldzoneid;

2246         if (secpolicy_zone_config(cr) != 0)
2247                 return (EPERM);
2248         if (zoneid != GLOBAL_ZONEID && zoneid != p->p_zone->zone_id)
2249                 return (EINVAL);
2250         if ((zptr = zone_find_by_id(zoneid)) == NULL)
2251                 return (EINVAL);
2252         mutex_exit(&p->p_lock);
2253         mutex_enter(&p->p_crlock);
2254         oldcred = p->p_cred;
2255         crhold(oldcred);
2256         mutex_exit(&p->p_crlock);
2257         newcred = crdup(oldcred);
2258         oldzoneid = crgetzoneid(oldcred);
2259         crfree(oldcred);

2261         crsetzone(newcred, zptr);
2262         zone_rele(zptr);

2264         mutex_enter(&p->p_crlock);
2265         oldcred = p->p_cred;
2266         p->p_cred = newcred;
2267         mutex_exit(&p->p_crlock);
2268         crfree(oldcred);

2270         /*
2271          * The target process is changing zones (according to its cred), so
2272          * update the per-zone upcounts, which are based on process creds.
2273          */
2274         if (oldzoneid != zoneid) {
2275                 uid_t ruid = crgetruid(newcred);

2277                 mutex_enter(&pidlock);
2278                 upcount_dec(ruid, oldzoneid);
2279                 upcount_inc(ruid, zoneid);
2280                 mutex_exit(&pidlock);
2281         }
2282         /*
2283          * Broadcast the cred change to the threads.
2284          */
2285         mutex_enter(&p->p_lock);
2286         t = p->p_tlist;
2287         do {
2288                 t->t_pre_sys = 1;       /* so syscall will get new cred */
2289         } while ((t = t->t_forw) != p->p_tlist);

2291         return (0);
2292 }

2294 static int
2295 pr_spriv(proc_t *p, prpriv_t *prpriv, cred_t *cr)
2296 {
2297         kthread_t *t;
2298         int err;

2300         ASSERT(MUTEX_HELD(&p->p_lock));

2302         if ((err = priv_pr_spriv(p, prpriv, cr)) == 0) {
2303                 /*
```



```
2304                  * Broadcast the cred change to the threads.
2305                  */
2306                 t = p->p_tlist;
2307                 do {
2308                         t->t_pre_sys = 1; /* so syscall will get new cred */
2309                 } while ((t = t->t_forw) != p->p_tlist);
2310         }

2312         return (err);
2313 }

2315 /*
2316  * Return -1 if the process is the parent of a vfork(1) whose child has yet to
2317  * terminate or perform an exec(2).
2318  *
2319  * Returns 0 if the process is fully stopped except for the current thread (if
2320  * we are operating on our own process), 1 otherwise.
2321  *
2322  * If the watchstop flag is set, then we ignore threads with TP_WATCHSTOP set.
2323  * See holdwatch() for details.
2324  */
2325 int
2326 pr_allstopped(proc_t *p, int watchstop)
2327 {
2328         kthread_t *t;
2329         int rv = 0;

2331         ASSERT(MUTEX_HELD(&p->p_lock));

2333         if (p->p_flag & SVFWAIT)        /* waiting for vfork'd child to exec */
2334                 return (-1);

2336         if ((t = p->p_tlist) != NULL) {
2337                 do {
2338                         if (t == curthread || VSTOPPED(t) ||
2339                             (watchstop && (t->t_proc_flag & TP_WATCHSTOP)))
2340                                 continue;
2341                         thread_lock(t);
2342                         switch (t->t_state) {
2343                         case TS_ZOMB:
2344                         case TS_STOPPED:
2345                                 break;
2346                         case TS_SLEEP:
2347                                 if (!(t->t_flag & T_WAKEABLE) ||
2348                                     t->t_wchan0 == NULL)
2349                                         rv = 1;
2350                                 break;
2351                         default:
2352                                 rv = 1;
2353                                 break;
2354                         }
2355                         thread_unlock(t);
2356                 } while (rv == 0 && (t = t->t_forw) != p->p_tlist);
2357         }

2359         return (rv);
2360 }

2362 /*
2363  * Cause all lwps in the process to pause (for watchpoint operations).
2364  */
2365 static void
2366 pauselwps(proc_t *p)
2367 {
2368         kthread_t *t;
```

```
2370         ASSERT(MUTEX_HELD(&p->p_lock));
2371         ASSERT(p != curproc);

2373         if ((t = p->p_tlist) != NULL) {
2374                 do {
2375                         thread_lock(t);
2376                         t->t_proc_flag |= TP_PAUSE;
2377                         aston(t);
2378                         if ((ISWAKEABLE(t) && (t->t_wchan0 == NULL)) ||
2379                             ISWAITING(t)) {
2380                                 setrun_locked(t);
2381                         }
2382                         prpokethread(t);
2383                         thread_unlock(t);
2384                 } while ((t = t->t_forw) != p->p_tlist);
2385         }
2386 }

2388 /*
2389  * undo the effects of pauselwps()
2390  */
2391 static void
2392 unpauselwps(proc_t *p)
2393 {
2394         kthread_t *t;

2396         ASSERT(MUTEX_HELD(&p->p_lock));
2397         ASSERT(p != curproc);

2399         if ((t = p->p_tlist) != NULL) {
2400                 do {
2401                         thread_lock(t);
2402                         t->t_proc_flag &= ~TP_PAUSE;
2403                         if (t->t_state == TS_STOPPED) {
2404                                 t->t_schedflag |= TS_UNPAUSE;
2405                                 t->t_dtrace_stop = 0;
2406                                 setrun_locked(t);
2407                         }
2408                         thread_unlock(t);
2409                 } while ((t = t->t_forw) != p->p_tlist);
2410         }
2411 }

2413 /*
2414  * Cancel all watched areas.  Called from prclose().
2415  */
2416 proc_t *
2417 pr_cancel_watch(prnode_t *pnp)
2418 {
2419         proc_t *p = pnp->pr_pcommon->prc_proc;
2420         struct as *as;
2421         kthread_t *t;

2423         ASSERT(MUTEX_HELD(&p->p_lock) && (p->p_proc_flag & P_PR_LOCK));

2425         if (!pr_watch_active(p))
2426                 return (p);

2428         /*
2429          * Pause the process before dealing with the watchpoints.
2430          */
2431         if (p == curproc) {
2432                 prunlock(pnp);
2433                 while (holdwatch() != 0)
2434                         continue;
2435                 p = pr_p_lock(pnp);
```



```
2436                 mutex_exit(&pr_pidlock);
2437                 ASSERT(p == curproc);
2438         } else {
2439                 pauselwps(p);
2440                 while (p != NULL && pr_allstopped(p, 0) > 0) {
2441                         /*
2442                          * This cv/mutex pair is persistent even
2443                          * if the process disappears after we
2444                          * unmark it and drop p->p_lock.
2445                          */
2446                         kcondvar_t *cv = &pr_pid_cv[p->p_slot];
2447                         kmutex_t *mp = &p->p_lock;

2449                         prunmark(p);
2450                         (void) cv_wait(cv, mp);
2451                         mutex_exit(mp);
2452                         p = pr_p_lock(pnp);  /* NULL if process disappeared */
2453                         mutex_exit(&pr_pidlock);
2454                 }
2455         }

2457         if (p == NULL)          /* the process disappeared */
2458                 return (NULL);

2460         ASSERT(p == pnp->pr_pcommon->prc_proc);
2461         ASSERT(MUTEX_HELD(&p->p_lock) && (p->p_proc_flag & P_PR_LOCK));

2463         if (pr_watch_active(p)) {
2464                 pr_free_watchpoints(p);
2465                 if ((t = p->p_tlist) != NULL) {
2466                         do {
2467                                 watch_disable(t);

2469                         } while ((t = t->t_forw) != p->p_tlist);
2470                 }
2471         }

2473         if ((as = p->p_as) != NULL) {
2474                 avl_tree_t *tree;
2475                 struct watched_page *pwp;

2477                 /*
2478                  * If this is the parent of a vfork, the watched page
2479                  * list has been moved temporarily to p->p_wpage.
2480                  */
2481                 if (avl_numnodes(&p->p_wpage) != 0)
2482                         tree = &p->p_wpage;
2483                 else
2484                         tree = &as->a_wpage;

2486                 mutex_exit(&p->p_lock);
2487                 AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);

2489                 for (pwp = avl_first(tree); pwp != NULL;
2490                     pwp = AVL_NEXT(tree, pwp)) {
2491                         pwp->wp_read = 0;
2492                         pwp->wp_write = 0;
2493                         pwp->wp_exec = 0;
2494                         if ((pwp->wp_flags & WP_SETPROT) == 0) {
2495                                 pwp->wp_flags |= WP_SETPROT;
2496                                 pwp->wp_prot = pwp->wp_oprot;
2497                                 pwp->wp_list = p->p_wprot;
2498                                 p->p_wprot = pwp;
2499                         }
2500                 }
```

```
2502                    AS_LOCK_EXIT(as, &as->a_lock);
2503                    mutex_enter(&p->p_lock);
2504            }

2506            /*
2507             * Unpause the process now.
2508             */
2509            if (p == curproc)
2510                    continuelwps(p);
2511            else
2512                    unpauselwps(p);

2514            return (p);
2515    }
```

```
**********************************************************
   93855 Wed Jan 23 13:19:06 2013
new/usr/src/uts/common/fs/proc/prioctl.c
XXX AVX procfs
**********************************************************
______unchanged_portion_omitted_


 132 /*
 133  * Control operations (lots).
 134  */
 135 /*ARGSUSED*/
 136 #ifdef _SYSCALL32_IMPL
 137 static int
 138 prioctl64(
 139         struct vnode *vp,
 140         int cmd,
 141         intptr_t arg,
 142         int flag,
 143         cred_t *cr,
 144         int *rvalp,
 145         caller_context_t *ct)
 146 #else
 147 int
 148 prioctl(
 149         struct vnode *vp,
 150         int cmd,
 151         intptr_t arg,
 152         int flag,
 153         cred_t *cr,
 154         int *rvalp,
 155         caller_context_t *ct)
 156 #endif  /* _SYSCALL32_IMPL */
 157 {
 158         int nsig = PROC_IS_BRANDED(curproc)? BROP(curproc)->b_nsig : NSIG;
 159         caddr_t cmaddr = (caddr_t)arg;
 160         proc_t *p;
 161         user_t *up;
 162         kthread_t *t;
 163         klwp_t *lwp;
 164         prnode_t *pnp = VTOP(vp);
 165         prcommon_t *pcp;
 166         prnode_t *xpnp = NULL;
 167         int error;
 168         int zdisp;
 169         void *thing = NULL;
 170         size_t thingsize = 0;

 172         /*
 173          * For copyin()/copyout().
 174          */
 175         union {
 176                 caddr_t         va;
 177                 int             signo;
 178                 int             nice;
 179                 uint_t          lwpid;
 180                 long            flags;
 181                 prstatus_t      prstat;
 182                 prrun_t         prrun;
 183                 sigset_t        smask;
 184                 siginfo_t       info;
 185                 sysset_t        prmask;
 186                 prgregset_t     regs;
 187                 prfpregset_t    fpregs;
 188                 prpsinfo_t      prps;
 189                 sigset_t        holdmask;
```



```
 190                 fltset_t        fltmask;
 191                 prcred_t        prcred;
 192                 prhusage_t      prhusage;
 193                 prmap_t         prmap;
 194                 auxv_t          auxv[__KERN_NAUXV_IMPL];
 195         } un;

 197         if (pnp->pr_type == PR_TMPL)
 198                 return (prctioctl(pnp, cmd, arg, flag, cr));

 200         /*
 201          * Support for old /proc interface.
 202          */
 203         if (pnp->pr_pidfile != NULL) {
 204                 ASSERT(pnp->pr_type == PR_PIDDIR);
 205                 vp = pnp->pr_pidfile;
 206                 pnp = VTOP(vp);
 207                 ASSERT(pnp->pr_type == PR_PIDFILE);
 208         }

 210         if (pnp->pr_type != PR_PIDFILE && pnp->pr_type != PR_LWPIDFILE)
 211                 return (ENOTTY);

 213         /*
 214          * Fail ioctls which are logically "write" requests unless
 215          * the user has write permission.
 216          */
 217         if ((flag & FWRITE) == 0 && isprwrioctl(cmd))
 218                 return (EBADF);

 220         /*
 221          * Perform any necessary copyin() operations before
 222          * locking the process.  Helps avoid deadlocks and
 223          * improves performance.
 224          *
 225          * Also, detect invalid ioctl codes here to avoid
 226          * locking a process unnnecessarily.
 227          *
 228          * Also, prepare to allocate space that will be needed below,
 229          * case by case.
 230          */
 231         error = 0;
 232         switch (cmd) {
 233         case PIOCGETPR:
 234                 thingsize = sizeof (proc_t);
 235                 break;
 236         case PIOCGETU:
 237                 thingsize = sizeof (user_t);
 238                 break;
 239         case PIOCSTOP:
 240         case PIOCWSTOP:
 241         case PIOCLWPIDS:
 242         case PIOCGTRACE:
 243         case PIOCGENTRY:
 244         case PIOCGEXIT:
 245         case PIOCSRLC:
 246         case PIOCRRLC:
 247         case PIOCSFORK:
 248         case PIOCRFORK:
 249         case PIOCGREG:
 250         case PIOCGFPREG:
 251         case PIOCSTATUS:
 252         case PIOCLSTATUS:
 253         case PIOCPSINFO:
 254         case PIOCMAXSIG:
 255         case PIOCGXREGSIZE:
```

```
256                 break;
257         case PIOCSXREG:         /* set extra registers */
258         case PIOCGXREG:         /* get extra registers */
259 #if defined(__sparc)
259                 thingsize = sizeof (prxregset_t);
261 #else
262                 thingsize = 0;
263 #endif
260                 break;
261         case PIOCACTION:
262                 thingsize = (nsig-1) * sizeof (struct sigaction);
263                 break;
264         case PIOCGHOLD:
265         case PIOCNMAP:
266         case PIOCMAP:
267         case PIOCGFAULT:
268         case PIOCCFAULT:
269         case PIOCCRED:
270         case PIOCGROUPS:
271         case PIOCUSAGE:
272         case PIOCLUSAGE:
273                 break;
274         case PIOCOPENPD:
275                 /*
276                  * We will need this below.
277                  * Allocate it now, before locking the process.
278                  */
279                 xpnp = prgetnode(vp, PR_OPAGEDATA);
280                 break;
281         case PIOCNAUXV:
282         case PIOCAUXV:
283                 break;

285 #if defined(__i386) || defined(__amd64)
286         case PIOCNLDT:
287         case PIOCLDT:
288                 break;
289 #endif  /* __i386 || __amd64 */

291 #if defined(__sparc)
292         case PIOCGWIN:
293                 thingsize = sizeof (gwindows_t);
294                 break;
295 #endif  /* __sparc */

297         case PIOCOPENM:         /* open mapped object for reading */
298                 if (cmaddr == NULL)
299                         un.va = NULL;
300                 else if (copyin(cmaddr, &un.va, sizeof (un.va)))
301                         error = EFAULT;
302                 break;

304         case PIOCRUN:           /* make lwp or process runnable */
305                 if (cmaddr == NULL)
306                         un.prrun.pr_flags = 0;
307                 else if (copyin(cmaddr, &un.prrun, sizeof (un.prrun)))
308                         error = EFAULT;
309                 break;

311         case PIOCOPENLWP:       /* return /proc lwp file descriptor */
312                 if (copyin(cmaddr, &un.lwpid, sizeof (un.lwpid)))
313                         error = EFAULT;
314                 break;

316         case PIOCSTRACE:        /* set signal trace mask */
317                 if (copyin(cmaddr, &un.smask, sizeof (un.smask)))
```



```
318                         error = EFAULT;
319                 break;

321         case PIOCSSIG:          /* set current signal */
322                 if (cmaddr == NULL)
323                         un.info.si_signo = 0;
324                 else if (copyin(cmaddr, &un.info, sizeof (un.info)))
325                         error = EFAULT;
326                 break;

328         case PIOCKILL:          /* send signal */
329         case PIOCUNKILL:        /* delete a signal */
330                 if (copyin(cmaddr, &un.signo, sizeof (un.signo)))
331                         error = EFAULT;
332                 break;

334         case PIOCNICE:          /* set nice priority */
335                 if (copyin(cmaddr, &un.nice, sizeof (un.nice)))
336                         error = EFAULT;
337                 break;

339         case PIOCSENTRY:        /* set syscall entry bit mask */
340         case PIOCSEXIT:         /* set syscall exit bit mask */
341                 if (copyin(cmaddr, &un.prmask, sizeof (un.prmask)))
342                         error = EFAULT;
343                 break;

345         case PIOCSET:           /* set process flags */
346         case PIOCRESET:         /* reset process flags */
347                 if (copyin(cmaddr, &un.flags, sizeof (un.flags)))
348                         error = EFAULT;
349                 break;

351         case PIOCSREG:          /* set general registers */
352                 if (copyin(cmaddr, un.regs, sizeof (un.regs)))
353                         error = EFAULT;
354                 break;

356         case PIOCSFPREG:        /* set floating-point registers */
357                 if (copyin(cmaddr, &un.fpregs, sizeof (un.fpregs)))
358                         error = EFAULT;
359                 break;

361         case PIOCSHOLD:         /* set signal-hold mask */
362                 if (copyin(cmaddr, &un.holdmask, sizeof (un.holdmask)))
363                         error = EFAULT;
364                 break;

366         case PIOCSFAULT:        /* set mask of traced faults */
367                 if (copyin(cmaddr, &un.fltmask, sizeof (un.fltmask)))
368                         error = EFAULT;
369                 break;

371         default:
372                 error = EINVAL;
373                 break;
374         }

376         if (error)
377                 return (error);

379 startover:
380         /*
381          * If we need kmem_alloc()d space then we allocate it now, before
382          * grabbing the process lock.  Using kmem_alloc(KM_SLEEP) while
383          * holding the process lock leads to deadlock with the clock thread.
```

```
384          * (The clock thread wakes up the pageout daemon to free up space.
385          * If the clock thread blocks behind us and we are sleeping waiting
386          * for space, then space may never become available.)
387          */
388         if (thingsize) {
389                 ASSERT(thing == NULL);
390                 thing = kmem_alloc(thingsize, KM_SLEEP);
391         }

393         switch (cmd) {
394         case PIOCPSINFO:
395         case PIOCGETPR:
396         case PIOCUSAGE:
397         case PIOCLUSAGE:
398                 zdisp = ZYES;
399                 break;
400         case PIOCSXREG:         /* set extra registers */
401                 /*
402                  * perform copyin before grabbing the process lock
403                  */
404                 if (thing) {
405                         if (copyin(cmaddr, thing, thingsize)) {
406                                 kmem_free(thing, thingsize);
407                                 return (EFAULT);
408                         }
409                 }
410                 /* fall through... */
411         default:
412                 zdisp = ZNO;
413                 break;
414         }

416         if ((error = prlock(pnp, zdisp)) != 0) {
417                 if (thing != NULL)
418                         kmem_free(thing, thingsize);
419                 if (xpnp)
420                         prfreenode(xpnp);
421                 return (error);
422         }

424         pcp = pnp->pr_common;
425         p = pcp->prc_proc;
426         ASSERT(p != NULL);

428         /*
429          * Choose a thread/lwp for the operation.
430          */
431         if (zdisp == ZNO && cmd != PIOCSTOP && cmd != PIOCWSTOP) {
432                 if (pnp->pr_type == PR_LWPIDFILE && cmd != PIOCLSTATUS) {
433                         t = pcp->prc_thread;
434                         ASSERT(t != NULL);
435                 } else {
436                         t = prchoose(p);        /* returns locked thread */
437                         ASSERT(t != NULL);
438                         thread_unlock(t);
439                 }
440                 lwp = ttolwp(t);
441         }

443         error = 0;
444         switch (cmd) {

446         case PIOCGETPR:         /* read struct proc */
447         {
448                 proc_t *prp = thing;
```



```
450                 *prp = *p;
451                 prunlock(pnp);
452                 if (copyout(prp, cmaddr, sizeof (proc_t)))
453                         error = EFAULT;
454                 kmem_free(prp, sizeof (proc_t));
455                 thing = NULL;
456                 break;
457         }

459         case PIOCGETU:          /* read u-area */
460         {
461                 user_t *userp = thing;

463                 up = PTOU(p);
464                 *userp = *up;
465                 prunlock(pnp);
466                 if (copyout(userp, cmaddr, sizeof (user_t)))
467                         error = EFAULT;
468                 kmem_free(userp, sizeof (user_t));
469                 thing = NULL;
470                 break;
471         }

473         case PIOCOPENM:         /* open mapped object for reading */
474                 error = propenm(pnp, cmaddr, un.va, rvalp, cr);
475                 /* propenm() called prunlock(pnp) */
476                 break;

478         case PIOCSTOP:          /* stop process or lwp from running */
479         case PIOCWSTOP:         /* wait for process or lwp to stop */
480                 /*
481                  * Can't apply to a system process.
482                  */
483                 if ((p->p_flag & SSYS) || p->p_as == &kas) {
484                         prunlock(pnp);
485                         error = EBUSY;
486                         break;
487                 }

489                 if (cmd == PIOCSTOP)
490                         pr_stop(pnp);

492                 /*
493                  * If an lwp is waiting for itself or its process, don't wait.
494                  * The stopped lwp would never see the fact that it is stopped.
495                  */
496                 if ((pnp->pr_type == PR_LWPIDFILE)?
497                     (pcp->prc_thread == curthread) : (p == curproc)) {
498                         if (cmd == PIOCWSTOP)
499                                 error = EBUSY;
500                         prunlock(pnp);
501                         break;
502                 }

504                 if ((error = pr_wait_stop(pnp, (time_t)0)) != 0)
505                         break;  /* pr_wait_stop() unlocked the process */

507                 if (cmaddr == NULL)
508                         prunlock(pnp);
509                 else {
510                         /*
511                          * Return process/lwp status information.
512                          */
513                         t = pr_thread(pnp);     /* returns locked thread */
514                         thread_unlock(t);
515                         oprgetstatus(t, &un.prstat, VTOZONE(vp));
```

```
516                         prunlock(pnp);
517                         if (copyout(&un.prstat, cmaddr, sizeof (un.prstat)))
518                                 error = EFAULT;
519                 }
520                 break;

522         case PIOCRUN:           /* make lwp or process runnable */
523         {
524                 long flags = un.prrun.pr_flags;

526                 /*
527                  * Cannot set an lwp running is it is not stopped.
528                  * Also, no lwp other than the /proc agent lwp can
529                  * be set running so long as the /proc agent lwp exists.
530                  */
531                 if ((!ISTOPPED(t) && !VSTOPPED(t) &&
532                     !(t->t_proc_flag & TP_PRSTOP)) ||
533                     (p->p_agenttp != NULL &&
534                     (t != p->p_agenttp || pnp->pr_type != PR_LWPIDFILE))) {
535                         prunlock(pnp);
536                         error = EBUSY;
537                         break;
538                 }

540                 if (flags & (PRSHOLD|PRSTRACE|PRSFAULT|PRSVADDR))
541                         prsetrun(t, &un.prrun);

543                 error = pr_setrun(pnp, prmaprunflags(flags));

545                 prunlock(pnp);
546                 break;
547         }

549         case PIOCLWPIDS:        /* get array of lwp identifiers */
550         {
551                 int nlwp;
552                 int Nlwp;
553                 id_t *idp;
554                 id_t *Bidp;

556                 Nlwp = nlwp = p->p_lwpcnt;

558                 if (thing && thingsize != (Nlwp+1) * sizeof (id_t)) {
559                         kmem_free(thing, thingsize);
560                         thing = NULL;
561                 }
562                 if (thing == NULL) {
563                         thingsize = (Nlwp+1) * sizeof (id_t);
564                         thing = kmem_alloc(thingsize, KM_NOSLEEP);
565                 }
566                 if (thing == NULL) {
567                         prunlock(pnp);
568                         goto startover;
569                 }

571                 idp = thing;
572                 thing = NULL;
573                 Bidp = idp;
574                 if ((t = p->p_tlist) != NULL) {
575                         do {
576                                 ASSERT(!(t->t_proc_flag & TP_LWPEXIT));
577                                 ASSERT(nlwp > 0);
578                                 --nlwp;
579                                 *idp++ = t->t_tid;
580                         } while ((t = t->t_forw) != p->p_tlist);
581                 }
```



```
582                 *idp = 0;
583                 ASSERT(nlwp == 0);
584                 prunlock(pnp);
585                 if (copyout(Bidp, cmaddr, (Nlwp+1) * sizeof (id_t)))
586                         error = EFAULT;
587                 kmem_free(Bidp, (Nlwp+1) * sizeof (id_t));
588                 break;
589         }

591         case PIOCOPENLWP:       /* return /proc lwp file descriptor */
592         {
593                 vnode_t *xvp;
594                 int n;

596                 prunlock(pnp);
597                 if ((xvp = prlwpnode(pnp, un.lwpid)) == NULL)
598                         error = ENOENT;
599                 else if (error = fassign(&xvp, flag & (FREAD|FWRITE), &n)) {
600                         VN_RELE(xvp);
601                 } else
602                         *rvalp = n;
603                 break;
604         }

606         case PIOCOPENPD:        /* return /proc page data file descriptor */
607         {
608                 vnode_t *xvp = PTOV(xpnp);
609                 vnode_t *dp = pnp->pr_parent;
610                 int n;

612                 if (pnp->pr_type == PR_LWPIDFILE) {
613                         dp = VTOP(dp)->pr_parent;
614                         dp = VTOP(dp)->pr_parent;
615                 }
616                 ASSERT(VTOP(dp)->pr_type == PR_PIDDIR);

618                 VN_HOLD(dp);
619                 pcp = pnp->pr_pcommon;
620                 xpnp->pr_ino = ptoi(pcp->prc_pid);
621                 xpnp->pr_common = pcp;
622                 xpnp->pr_pcommon = pcp;
623                 xpnp->pr_parent = dp;

625                 xpnp->pr_next = p->p_plist;
626                 p->p_plist = xvp;

628                 prunlock(pnp);
629                 if (error = fassign(&xvp, FREAD, &n)) {
630                         VN_RELE(xvp);
631                 } else
632                         *rvalp = n;

634                 xpnp = NULL;
635                 break;
636         }

638         case PIOCGTRACE:        /* get signal trace mask */
639                 prassignset(&un.smask, &p->p_sigmask);
640                 prunlock(pnp);
641                 if (copyout(&un.smask, cmaddr, sizeof (un.smask)))
642                         error = EFAULT;
643                 break;

645         case PIOCSTRACE:        /* set signal trace mask */
646                 prdelset(&un.smask, SIGKILL);
647                 prassignset(&p->p_sigmask, &un.smask);
```

```
648                 if (!sigisempty(&p->p_sigmask))
649                         p->p_proc_flag |= P_PR_TRACE;
650                 else if (prisempty(&p->p_fltmask)) {
651                         up = PTOU(p);
652                         if (up->u_systrap == 0)
653                                 p->p_proc_flag &= ~P_PR_TRACE;
654                 }
655                 prunlock(pnp);
656                 break;

658         case PIOCSSIG:          /* set current signal */
659                 error = pr_setsig(pnp, &un.info);
660                 prunlock(pnp);
661                 if (un.info.si_signo == SIGKILL && error == 0)
662                         pr_wait_die(pnp);
663                 break;

665         case PIOCKILL:          /* send signal */
666         {
667                 int sig = (int)un.signo;

669                 error = pr_kill(pnp, sig, cr);
670                 prunlock(pnp);
671                 if (sig == SIGKILL && error == 0)
672                         pr_wait_die(pnp);
673                 break;
674         }

676         case PIOCUNKILL:        /* delete a signal */
677                 error = pr_unkill(pnp, (int)un.signo);
678                 prunlock(pnp);
679                 break;

681         case PIOCNICE:          /* set nice priority */
682                 error = pr_nice(p, (int)un.nice, cr);
683                 prunlock(pnp);
684                 break;

686         case PIOCGENTRY:        /* get syscall entry bit mask */
687         case PIOCGEXIT:         /* get syscall exit bit mask */
688                 up = PTOU(p);
689                 if (cmd == PIOCGENTRY) {
690                         prassignset(&un.prmask, &up->u_entrymask);
691                 } else {
692                         prassignset(&un.prmask, &up->u_exitmask);
693                 }
694                 prunlock(pnp);
695                 if (copyout(&un.prmask, cmaddr, sizeof (un.prmask)))
696                         error = EFAULT;
697                 break;

699         case PIOCSENTRY:        /* set syscall entry bit mask */
700         case PIOCSEXIT:         /* set syscall exit bit mask */
701                 pr_setentryexit(p, &un.prmask, cmd == PIOCSENTRY);
702                 prunlock(pnp);
703                 break;

705         case PIOCSRLC:          /* obsolete: set running on last /proc close */
706                 error = pr_set(p, prmapsetflags(PR_RLC));
707                 prunlock(pnp);
708                 break;

710         case PIOCRRLC:          /* obsolete: reset run-on-last-close flag */
711                 error = pr_unset(p, prmapsetflags(PR_RLC));
712                 prunlock(pnp);
713                 break;
```



```
715         case PIOCSFORK:         /* obsolete: set inherit-on-fork flag */
716                 error = pr_set(p, prmapsetflags(PR_FORK));
717                 prunlock(pnp);
718                 break;

720         case PIOCRFORK:         /* obsolete: reset inherit-on-fork flag */
721                 error = pr_unset(p, prmapsetflags(PR_FORK));
722                 prunlock(pnp);
723                 break;

725         case PIOCSET:           /* set process flags */
726                 error = pr_set(p, prmapsetflags(un.flags));
727                 prunlock(pnp);
728                 break;

730         case PIOCRESET:         /* reset process flags */
731                 error = pr_unset(p, prmapsetflags(un.flags));
732                 prunlock(pnp);
733                 break;

735         case PIOCGREG:          /* get general registers */
736                 if (t->t_state != TS_STOPPED && !VSTOPPED(t))
737                         bzero(un.regs, sizeof (un.regs));
738                 else {
739                         /* drop p_lock while touching the lwp's stack */
740                         mutex_exit(&p->p_lock);
741                         prgetprregs(lwp, un.regs);
742                         mutex_enter(&p->p_lock);
743                 }
744                 prunlock(pnp);
745                 if (copyout(un.regs, cmaddr, sizeof (un.regs)))
746                         error = EFAULT;
747                 break;

749         case PIOCSREG:          /* set general registers */
750                 if (!ISTOPPED(t) && !VSTOPPED(t) && !DSTOPPED(t))
751                         error = EBUSY;
752                 else {
753                         /* drop p_lock while touching the lwp's stack */
754                         mutex_exit(&p->p_lock);
755                         prsetprregs(lwp, un.regs, 0);
756                         mutex_enter(&p->p_lock);
757                 }
758                 prunlock(pnp);
759                 break;

761         case PIOCGFPREG:        /* get floating-point registers */
762                 if (!prhasfp()) {
763                         prunlock(pnp);
764                         error = EINVAL; /* No FP support */
765                         break;
766                 }

768                 if (t->t_state != TS_STOPPED && !VSTOPPED(t))
769                         bzero(&un.fpregs, sizeof (un.fpregs));
770                 else {
771                         /* drop p_lock while touching the lwp's stack */
772                         mutex_exit(&p->p_lock);
773                         prgetprfpregs(lwp, &un.fpregs);
774                         mutex_enter(&p->p_lock);
775                 }
776                 prunlock(pnp);
777                 if (copyout(&un.fpregs, cmaddr, sizeof (un.fpregs)))
778                         error = EFAULT;
779                 break;
```

```
781         case PIOCSFPREG:        /* set floating-point registers */
782                 if (!prhasfp())
783                         error = EINVAL; /* No FP support */
784                 else if (!ISTOPPED(t) && !VSTOPPED(t) && !DSTOPPED(t))
785                         error = EBUSY;
786                 else {
787                         /* drop p_lock while touching the lwp's stack */
788                         mutex_exit(&p->p_lock);
789                         prsetprfpregs(lwp, &un.fpregs);
790                         mutex_enter(&p->p_lock);
791                 }
792                 prunlock(pnp);
793                 break;

795         case PIOCGXREGSIZE:     /* get the size of the extra registers */
796         {
797                 int xregsize;

799                 if (prhasx(p)) {
800                         xregsize = prgetprxregsize(p);
801                         prunlock(pnp);
802                         if (copyout(&xregsize, cmaddr, sizeof (xregsize)))
803                                 error = EFAULT;
804                 } else {
805                         prunlock(pnp);
806                         error = EINVAL; /* No extra register support */
807                 }
808                 break;
809         }

811         case PIOCGXREG:         /* get extra registers */
812                 if (prhasx(p)) {
813                         bzero(thing, thingsize);
814                         if (t->t_state == TS_STOPPED || VSTOPPED(t)) {
815                                 /* drop p_lock to touch the stack */
816                                 mutex_exit(&p->p_lock);
817                                 prgetprxregs(lwp, thing);
818                                 mutex_enter(&p->p_lock);
819                         }
820                         prunlock(pnp);
821                         if (copyout(thing, cmaddr, thingsize))
822                                 error = EFAULT;
823                 } else {
824                         prunlock(pnp);
825                         error = EINVAL; /* No extra register support */
826                 }
827                 if (thing) {
828                         kmem_free(thing, thingsize);
829                         thing = NULL;
830                 }
831                 break;

833         case PIOCSXREG:         /* set extra registers */
834                 if (!ISTOPPED(t) && !VSTOPPED(t) && !DSTOPPED(t))
835                         error = EBUSY;
836                 else if (!prhasx(p))
837                         error = EINVAL; /* No extra register support */
838                 else if (thing) {
839                         /* drop p_lock while touching the lwp's stack */
840                         mutex_exit(&p->p_lock);
841                         prsetprxregs(lwp, thing);
842                         mutex_enter(&p->p_lock);
843                 }
844                 prunlock(pnp);
845                 if (thing) {
```



```
846                         kmem_free(thing, thingsize);
847                         thing = NULL;
848                 }
849                 break;

851         case PIOCSTATUS:        /* get process/lwp status */
852                 oprgetstatus(t, &un.prstat, VTOZONE(vp));
853                 prunlock(pnp);
854                 if (copyout(&un.prstat, cmaddr, sizeof (un.prstat)))
855                         error = EFAULT;
856                 break;

858         case PIOCLSTATUS:       /* get status for process & all lwps */
859         {
860                 int Nlwp;
861                 int nlwp;
862                 prstatus_t *Bprsp;
863                 prstatus_t *prsp;

865                 nlwp = Nlwp = p->p_lwpcnt;

867                 if (thing && thingsize != (Nlwp+1) * sizeof (prstatus_t)) {
868                         kmem_free(thing, thingsize);
869                         thing = NULL;
870                 }
871                 if (thing == NULL) {
872                         thingsize = (Nlwp+1) * sizeof (prstatus_t);
873                         thing = kmem_alloc(thingsize, KM_NOSLEEP);
874                 }
875                 if (thing == NULL) {
876                         prunlock(pnp);
877                         goto startover;
878                 }

880                 Bprsp = thing;
881                 thing = NULL;
882                 prsp = Bprsp;
883                 oprgetstatus(t, prsp, VTOZONE(vp));
884                 t = p->p_tlist;
885                 do {
886                         ASSERT(!(t->t_proc_flag & TP_LWPEXIT));
887                         ASSERT(nlwp > 0);
888                         --nlwp;
889                         oprgetstatus(t, ++prsp, VTOZONE(vp));
890                 } while ((t = t->t_forw) != p->p_tlist);
891                 ASSERT(nlwp == 0);
892                 prunlock(pnp);
893                 if (copyout(Bprsp, cmaddr, (Nlwp+1) * sizeof (prstatus_t)))
894                         error = EFAULT;

896                 kmem_free(Bprsp, (Nlwp+1) * sizeof (prstatus_t));
897                 break;
898         }

900         case PIOCPSINFO:        /* get ps(1) information */
901         {
902                 prpsinfo_t *psp = &un.prps;

904                 oprgetpsinfo(p, psp,
905                     (pnp->pr_type == PR_LWPIDFILE)? pcp->prc_thread : NULL);

907                 prunlock(pnp);
908                 if (copyout(&un.prps, cmaddr, sizeof (un.prps)))
909                         error = EFAULT;
910                 break;
911         }
```

```
913         case PIOCMAXSIG:        /* get maximum signal number */
914         {
915                 int n = nsig-1;

917                 prunlock(pnp);
918                 if (copyout(&n, cmaddr, sizeof (n)))
919                         error = EFAULT;
920                 break;
921         }

923         case PIOCACTION:        /* get signal action structures */
924         {
925                 uint_t sig;
926                 struct sigaction *sap = thing;

928                 up = PTOU(p);
929                 for (sig = 1; sig < nsig; sig++)
930                         prgetaction(p, up, sig, &sap[sig-1]);
931                 prunlock(pnp);
932                 if (copyout(sap, cmaddr, (nsig-1) * sizeof (struct sigaction)))
933                         error = EFAULT;
934                 kmem_free(sap, (nsig-1) * sizeof (struct sigaction));
935                 thing = NULL;
936                 break;
937         }

939         case PIOCGHOLD:         /* get signal-hold mask */
940                 schedctl_finish_sigblock(t);
941                 sigktou(&t->t_hold, &un.holdmask);
942                 prunlock(pnp);
943                 if (copyout(&un.holdmask, cmaddr, sizeof (un.holdmask)))
944                         error = EFAULT;
945                 break;

947         case PIOCSHOLD:         /* set signal-hold mask */
948                 pr_sethold(pnp, &un.holdmask);
949                 prunlock(pnp);
950                 break;

952         case PIOCNMAP:          /* get number of memory mappings */
953         {
954                 int n;
955                 struct as *as = p->p_as;

957                 if ((p->p_flag & SSYS) || as == &kas)
958                         n = 0;
959                 else {
960                         mutex_exit(&p->p_lock);
961                         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
962                         n = prnsegs(as, 0);
963                         AS_LOCK_EXIT(as, &as->a_lock);
964                         mutex_enter(&p->p_lock);
965                 }
966                 prunlock(pnp);
967                 if (copyout(&n, cmaddr, sizeof (int)))
968                         error = EFAULT;
969                 break;
970         }

972         case PIOCMAP:           /* get memory map information */
973         {
974                 list_t iolhead;
975                 struct as *as = p->p_as;

977                 if ((p->p_flag & SSYS) || as == &kas) {
```



```
978                         error = 0;
979                         prunlock(pnp);
980                 } else {
981                         mutex_exit(&p->p_lock);
982                         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
983                         error = oprgetmap(p, &iolhead);
984                         AS_LOCK_EXIT(as, &as->a_lock);
985                         mutex_enter(&p->p_lock);
986                         prunlock(pnp);

988                         error = pr_iol_copyout_and_free(&iolhead,
989                             &cmaddr, error);
990                 }
991                 /*
992                  * The procfs PIOCMAP ioctl returns an all-zero buffer
993                  * to indicate the end of the prmap[] array.
994                  * Append it to whatever has already been copied out.
995                  */
996                 bzero(&un.prmap, sizeof (un.prmap));
997                 if (!error && copyout(&un.prmap, cmaddr, sizeof (un.prmap)))
998                         error = EFAULT;

1000                break;
1001        }

1003        case PIOCGFAULT:        /* get mask of traced faults */
1004                prassignset(&un.fltmask, &p->p_fltmask);
1005                prunlock(pnp);
1006                if (copyout(&un.fltmask, cmaddr, sizeof (un.fltmask)))
1007                        error = EFAULT;
1008                break;

1010        case PIOCSFAULT:        /* set mask of traced faults */
1011                pr_setfault(p, &un.fltmask);
1012                prunlock(pnp);
1013                break;

1015        case PIOCCFAULT:        /* clear current fault */
1016                lwp->lwp_curflt = 0;
1017                prunlock(pnp);
1018                break;

1020        case PIOCCRED:          /* get process credentials */
1021        {
1022                cred_t *cp;

1024                mutex_enter(&p->p_crlock);
1025                cp = p->p_cred;
1026                un.prcred.pr_euid = crgetuid(cp);
1027                un.prcred.pr_ruid = crgetruid(cp);
1028                un.prcred.pr_suid = crgetsuid(cp);
1029                un.prcred.pr_egid = crgetgid(cp);
1030                un.prcred.pr_rgid = crgetrgid(cp);
1031                un.prcred.pr_sgid = crgetsgid(cp);
1032                un.prcred.pr_ngroups = crgetngroups(cp);
1033                mutex_exit(&p->p_crlock);

1035                prunlock(pnp);
1036                if (copyout(&un.prcred, cmaddr, sizeof (un.prcred)))
1037                        error = EFAULT;
1038                break;
1039        }

1041        case PIOCGROUPS:        /* get supplementary groups */
1042        {
1043                cred_t *cp;
```

```
1045                mutex_enter(&p->p_crlock);
1046                cp = p->p_cred;
1047                crhold(cp);
1048                mutex_exit(&p->p_crlock);

1050                prunlock(pnp);
1051                if (copyout(crgetgroups(cp), cmaddr,
1052                    MAX(crgetngroups(cp), 1) * sizeof (gid_t)))
1053                        error = EFAULT;
1054                crfree(cp);
1055                break;
1056        }

1058        case PIOCUSAGE:         /* get usage info */
1059        {
1060                /*
1061                 * For an lwp file descriptor, return just the lwp usage.
1062                 * For a process file descriptor, return total usage,
1063                 * all current lwps plus all defunct lwps.
1064                 */
1065                prhusage_t *pup = &un.prhusage;
1066                prusage_t *upup;

1068                bzero(pup, sizeof (*pup));
1069                pup->pr_tstamp = gethrtime();

1071                if (pnp->pr_type == PR_LWPIDFILE) {
1072                        t = pcp->prc_thread;
1073                        if (t != NULL)
1074                                prgetusage(t, pup);
1075                        else
1076                                error = ENOENT;
1077                } else {
1078                        pup->pr_count  = p->p_defunct;
1079                        pup->pr_create = p->p_mstart;
1080                        pup->pr_term   = p->p_mterm;

1082                        pup->pr_rtime    = p->p_mlreal;
1083                        pup->pr_utime    = p->p_acct[LMS_USER];
1084                        pup->pr_stime    = p->p_acct[LMS_SYSTEM];
1085                        pup->pr_ttime    = p->p_acct[LMS_TRAP];
1086                        pup->pr_tftime   = p->p_acct[LMS_TFAULT];
1087                        pup->pr_dftime   = p->p_acct[LMS_DFAULT];
1088                        pup->pr_kftime   = p->p_acct[LMS_KFAULT];
1089                        pup->pr_ltime    = p->p_acct[LMS_USER_LOCK];
1090                        pup->pr_slptime  = p->p_acct[LMS_SLEEP];
1091                        pup->pr_wtime    = p->p_acct[LMS_WAIT_CPU];
1092                        pup->pr_stoptime = p->p_acct[LMS_STOPPED];

1094                        pup->pr_minf  = p->p_ru.minflt;
1095                        pup->pr_majf  = p->p_ru.majflt;
1096                        pup->pr_nswap = p->p_ru.nswap;
1097                        pup->pr_inblk = p->p_ru.inblock;
1098                        pup->pr_oublk = p->p_ru.oublock;
1099                        pup->pr_msnd  = p->p_ru.msgsnd;
1100                        pup->pr_mrcv  = p->p_ru.msgrcv;
1101                        pup->pr_sigs  = p->p_ru.nsignals;
1102                        pup->pr_vctx  = p->p_ru.nvcsw;
1103                        pup->pr_ictx  = p->p_ru.nivcsw;
1104                        pup->pr_sysc  = p->p_ru.sysc;
1105                        pup->pr_ioch  = p->p_ru.ioch;

1107                        /*
1108                         * Add the usage information for each active lwp.
1109                         */
```



```
1110                        if ((t = p->p_tlist) != NULL &&
1111                            !(pcp->prc_flags & PRC_DESTROY)) {
1112                                do {
1113                                        ASSERT(!(t->t_proc_flag & TP_LWPEXIT));
1114                                        pup->pr_count++;
1115                                        praddusage(t, pup);
1116                                } while ((t = t->t_forw) != p->p_tlist);
1117                        }
1118                }

1120                prunlock(pnp);

1122                upup = kmem_zalloc(sizeof (*upup), KM_SLEEP);
1123                prcvtusage(&un.prhusage, upup);
1124                if (copyout(upup, cmaddr, sizeof (*upup)))
1125                        error = EFAULT;
1126                kmem_free(upup, sizeof (*upup));

1128                break;
1129        }

1131        case PIOCLUSAGE:        /* get detailed usage info */
1132        {
1133                int Nlwp;
1134                int nlwp;
1135                prusage_t *upup;
1136                prusage_t *Bupup;
1137                prhusage_t *pup;
1138                hrtime_t curtime;

1140                nlwp = Nlwp = (pcp->prc_flags & PRC_DESTROY)? 0 : p->p_lwpcnt;

1142                if (thing && thingsize !=
1143                    sizeof (prhusage_t) + (Nlwp+1) * sizeof (prusage_t)) {
1144                        kmem_free(thing, thingsize);
1145                        thing = NULL;
1146                }
1147                if (thing == NULL) {
1148                        thingsize = sizeof (prhusage_t) +
1149                            (Nlwp+1) * sizeof (prusage_t);
1150                        thing = kmem_alloc(thingsize, KM_NOSLEEP);
1151                }
1152                if (thing == NULL) {
1153                        prunlock(pnp);
1154                        goto startover;
1155                }

1157                pup = thing;
1158                upup = Bupup = (prusage_t *)(pup + 1);

1160                ASSERT(p == pcp->prc_proc);

1162                curtime = gethrtime();

1164                /*
1165                 * First the summation over defunct lwps.
1166                 */
1167                bzero(pup, sizeof (*pup));
1168                pup->pr_count  = p->p_defunct;
1169                pup->pr_tstamp = curtime;
1170                pup->pr_create = p->p_mstart;
1171                pup->pr_term   = p->p_mterm;

1173                pup->pr_rtime    = p->p_mlreal;
1174                pup->pr_utime    = p->p_acct[LMS_USER];
1175                pup->pr_stime    = p->p_acct[LMS_SYSTEM];
```

```
1176                pup->pr_ttime    = p->p_acct[LMS_TRAP];
1177                pup->pr_tftime   = p->p_acct[LMS_TFAULT];
1178                pup->pr_dftime   = p->p_acct[LMS_DFAULT];
1179                pup->pr_kftime   = p->p_acct[LMS_KFAULT];
1180                pup->pr_ltime    = p->p_acct[LMS_USER_LOCK];
1181                pup->pr_slptime  = p->p_acct[LMS_SLEEP];
1182                pup->pr_wtime    = p->p_acct[LMS_WAIT_CPU];
1183                pup->pr_stoptime = p->p_acct[LMS_STOPPED];

1185                pup->pr_minf  = p->p_ru.minflt;
1186                pup->pr_majf  = p->p_ru.majflt;
1187                pup->pr_nswap = p->p_ru.nswap;
1188                pup->pr_inblk = p->p_ru.inblock;
1189                pup->pr_oublk = p->p_ru.oublock;
1190                pup->pr_msnd  = p->p_ru.msgsnd;
1191                pup->pr_mrcv  = p->p_ru.msgrcv;
1192                pup->pr_sigs  = p->p_ru.nsignals;
1193                pup->pr_vctx  = p->p_ru.nvcsw;
1194                pup->pr_ictx  = p->p_ru.nivcsw;
1195                pup->pr_sysc  = p->p_ru.sysc;
1196                pup->pr_ioch  = p->p_ru.ioch;

1198                prcvtusage(pup, upup);

1200                /*
1201                 * Fill one prusage struct for each active lwp.
1202                 */
1203                if ((t = p->p_tlist) != NULL &&
1204                    !(pcp->prc_flags & PRC_DESTROY)) {
1205                        do {
1206                                ASSERT(!(t->t_proc_flag & TP_LWPEXIT));
1207                                ASSERT(nlwp > 0);
1208                                --nlwp;
1209                                upup++;
1210                                prgetusage(t, pup);
1211                                prcvtusage(pup, upup);
1212                        } while ((t = t->t_forw) != p->p_tlist);
1213                }
1214                ASSERT(nlwp == 0);

1216                prunlock(pnp);
1217                if (copyout(Bupup, cmaddr, (Nlwp+1) * sizeof (prusage_t)))
1218                        error = EFAULT;
1219                kmem_free(thing, thingsize);
1220                thing = NULL;
1221                break;
1222        }

1224        case PIOCNAUXV:         /* get number of aux vector entries */
1225        {
1226                int n = __KERN_NAUXV_IMPL;

1228                prunlock(pnp);
1229                if (copyout(&n, cmaddr, sizeof (int)))
1230                        error = EFAULT;
1231                break;
1232        }

1234        case PIOCAUXV:          /* get aux vector (see sys/auxv.h) */
1235        {
1236                up = PTOU(p);
1237                bcopy(up->u_auxv, un.auxv,
1238                    __KERN_NAUXV_IMPL * sizeof (auxv_t));
1239                prunlock(pnp);
1240                if (copyout(un.auxv, cmaddr,
1241                    __KERN_NAUXV_IMPL * sizeof (auxv_t)))
```



```
1242                        error = EFAULT;
1243                break;
1244        }

1246 #if defined(__i386) || defined(__amd64)
1247        case PIOCNLDT:          /* get number of LDT entries */
1248        {
1249                int n;

1251                mutex_exit(&p->p_lock);
1252                mutex_enter(&p->p_ldtlock);
1253                n = prnldt(p);
1254                mutex_exit(&p->p_ldtlock);
1255                mutex_enter(&p->p_lock);
1256                prunlock(pnp);
1257                if (copyout(&n, cmaddr, sizeof (n)))
1258                        error = EFAULT;
1259                break;
1260        }

1262        case PIOCLDT:           /* get LDT entries */
1263        {
1264                struct ssd *ssd;
1265                int n;

1267                mutex_exit(&p->p_lock);
1268                mutex_enter(&p->p_ldtlock);
1269                n = prnldt(p);

1271                if (thing && thingsize != (n+1) * sizeof (*ssd)) {
1272                        kmem_free(thing, thingsize);
1273                        thing = NULL;
1274                }
1275                if (thing == NULL) {
1276                        thingsize = (n+1) * sizeof (*ssd);
1277                        thing = kmem_alloc(thingsize, KM_NOSLEEP);
1278                }
1279                if (thing == NULL) {
1280                        mutex_exit(&p->p_ldtlock);
1281                        mutex_enter(&p->p_lock);
1282                        prunlock(pnp);
1283                        goto startover;
1284                }

1286                ssd = thing;
1287                thing = NULL;
1288                if (n != 0)
1289                        prgetldt(p, ssd);
1290                mutex_exit(&p->p_ldtlock);
1291                mutex_enter(&p->p_lock);
1292                prunlock(pnp);

1294                /* mark the end of the list with a null entry */
1295                bzero(&ssd[n], sizeof (*ssd));
1296                if (copyout(ssd, cmaddr, (n+1) * sizeof (*ssd)))
1297                        error = EFAULT;
1298                kmem_free(ssd, (n+1) * sizeof (*ssd));
1299                break;
1300        }
1301 #endif  /* __i386 || __amd64 */

1303 #if defined(__sparc)
1304        case PIOCGWIN:          /* get gwindows_t (see sys/reg.h) */
1305        {
1306                gwindows_t *gwp = thing;
```

```
1308                 /* drop p->p_lock while touching the stack */
1309                 mutex_exit(&p->p_lock);
1310                 bzero(gwp, sizeof (*gwp));
1311                 prgetwindows(lwp, gwp);
1312                 mutex_enter(&p->p_lock);
1313                 prunlock(pnp);
1314                 if (copyout(gwp, cmaddr, sizeof (*gwp)))
1315                         error = EFAULT;
1316                 kmem_free(gwp, sizeof (gwindows_t));
1317                 thing = NULL;
1318                 break;
1319         }
1320 #endif  /* __sparc */

1322         default:
1323                 prunlock(pnp);
1324                 error = EINVAL;
1325                 break;

1327         }

1329         ASSERT(thing == NULL);
1330         ASSERT(xpnp == NULL);
1331         return (error);
1332 }
```

_____***unchanged_portion_omitted_***

****
141047 Wed Jan 23 13:19:06 2013
new/usr/src/uts/common/fs/proc/prvnops.c
XXX AVX procfs
****
_____***unchanged_portion_omitted_***

```
1523 /* ARGSUSED */
1524 static int
1525 pr_read_xregs(prnode_t *pnp, uio_t *uiop)
1526 {
1527 #if defined(__sparc)
1527         proc_t *p;
1528         kthread_t *t;
1529         int error;
1530         char *xreg;
1531         size_t size;

1533         ASSERT(pnp->pr_type == PR_XREGS);

1535         xreg = kmem_zalloc(sizeof (prxregset_t), KM_SLEEP);

1537         if ((error = prlock(pnp, ZNO)) != 0)
1538                 goto out;

1540         p = pnp->pr_common->prc_proc;
1541         t = pnp->pr_common->prc_thread;

1543         size = prhasx(p)? prgetprxregsize(p) : 0;
1544         if (uiop->uio_offset >= size) {
1545                 prunlock(pnp);
1546                 goto out;
1547         }

1549         /* drop p->p_lock while (possibly) touching the stack */
1550         mutex_exit(&p->p_lock);
1551         prgetprxregs(ttolwp(t), xreg);
1552         mutex_enter(&p->p_lock);
1553         prunlock(pnp);

1555         error = pr_uioread(xreg, size, uiop);
1556 out:
1557         kmem_free(xreg, sizeof (prxregset_t));
1558         return (error);
1560 #else
1561         return (0);
1562 #endif
1559 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    6490 Wed Jan 23 13:19:07 2013
new/usr/src/uts/intel/Makefile.files
XXX AVX procfs
**********************************************************
```

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

  22 #
  23 # Copyright (c) 1999, 2010, Oracle and/or its affiliates. All rights reserved.
  24 # Copyright (c) 2012, Joyent, Inc. All rights reserved.
  25 #

  27 #
  28 #       This Makefile defines all file modules and build rules for the
  29 # directory uts/intel and its children. These are the source files which
  30 # are specific to x86 processor architectures.
  31 #

  33 #
  34 #       Core (unix) objects
  35 #
  36 CORE_OBJS +=            \
  37         arch_kdi.o      \
  38         copy.o          \
  39         copy_subr.o     \
  40         cpc_subr.o      \
  41         ddi_arch.o      \
  42         ddi_i86.o       \
  43         ddi_i86_asm.o   \
  44         desctbls.o      \
  45         desctbls_asm.o  \
  46         exception.o     \
  47         float.o         \
  48         fmsmb.o         \
  49         fpu.o           \
  50         i86_subr.o      \
  51         lock_prim.o     \
  52         ovbcopy.o       \
  53         polled_io.o     \
  54         sseblk.o        \
  55         sundep.o        \
  56         swtch.o         \
  57         sysi86.o

  59 #
  60 # 64-bit multiply/divide compiler helper routines
  61 # used only for ia32
```



```
  62 #

  64 SPECIAL_OBJS_32 +=      \
  65         muldiv.o

  67 #
  68 #       Generic-unix Module
  69 #
  70 GENUNIX_OBJS +=         \
  71         archdep.o       \
  72         getcontext.o    \
  73         install_utrap.o \
  74         lwp_private.o   \
  75         prom_enter.o    \
  76         prom_exit.o     \
  77         prom_panic.o    \
  78         sendsig.o       \
  79         syscall.o       \
  80         xregs.o
  79         syscall.o


  83 #
  84 #       PROM Routines
  85 #
  86 GENUNIX_OBJS +=         \
  87         prom_env.o      \
  88         prom_emul.o     \
  89         prom_getchar.o  \
  90         prom_init.o     \
  91         prom_node.o     \
  92         prom_printf.o   \
  93         prom_prop.o     \
  94         prom_putchar.o  \
  95         prom_reboot.o   \
  96         prom_version.o

  98 #
  99 #       file system modules
 100 #
 101 CORE_OBJS +=            \
 102         prmachdep.o

 104 #
 105 #       ZFS file system module
 106 #
 107 ZFS_OBJS +=             \
 108         spa_boot.o

 110 #
 111 #       Decompression code
 112 #
 113 CORE_OBJS += decompress.o

 115 #
 116 #       Microcode utilities
 117 #
 118 CORE_OBJS += ucode_utils.o

 120 #
 121 #       Driver modules
 122 #
 123 AGPGART_OBJS += agpgart.o agp_kstat.o
 124 AGPTARGET_OBJS += agptarget.o
 125 AMD64GART_OBJS += amd64_gart.o
 126 ARCMSR_OBJS += arcmsr.o
```

```
ATA_OBJS += $(GHD_OBJS) ata_blacklist.o ata_common.o ata_disk.o \
        ata_dma.o atapi.o atapi_fsm.o ata_debug.o \
        sil3xxx.o
BSCBUS_OBJS += bscbus.o
BSCV_OBJS += bscv.o
CMDK_OBJS += cmdk.o
CMLB_OBJS += cmlb.o
CPUNEX_OBJS += cpunex.o
DADK_OBJS += dadk.o
DCOPY_OBJS += dcopy.o
DNET_OBJS += dnet.o dnet_mii.o
FD_OBJS += fd.o
GDA_OBJS += gda.o
GHD_OBJS += ghd.o ghd_debug.o ghd_dma.o ghd_queue.o ghd_scsa.o \
        ghd_scsi.o ghd_timer.o ghd_waitq.o ghd_gcmd.o
I915_OBJS += i915_dma.o i915_drv.o i915_irq.o i915_mem.o \
        i915_gem.o i915_gem_debug.o i915_gem_tiling.o
NSKERN_OBJS += nsc_asm.o
PCICFG_OBJS += pcicfg.o
PCI_PCINEXUS_OBJS += pci_pci.o
PCIEB_OBJS += pcieb_x86.o
PIT_BEEP_OBJS += pit_beep.o
POWER_OBJS += power.o
PCI_AUTOCONFIG_OBJS += pci_autoconfig.o pci_boot.o pcie_nvidia.o \
                       pci_memlist.o pci_resource.o
RADEON_OBJS += r300_cmdbuf.o radeon_cp.o radeon_drv.o \
                       radeon_state.o radeon_irq.o radeon_mem.o
SD_OBJS += sd.o sd_xbuf.o

HECI_OBJS +=            \
        heci_init.o     \
        heci_intr.o     \
        heci_interface.o \
        io_heci.o       \
        heci_main.o

STRATEGY_OBJS += strategy.o
UCODE_OBJS += ucode_drv.o
VGATEXT_OBJS += vgatext.o vgasubr.o

#
#       Kernel linker
#
KRTLD_OBJS +=           \
        bootrd.o        \
        ufsops.o        \
        hsfs.o          \
        doreloc.o       \
        kobj_boot.o     \
        kobj_convrelstr.o \
        kobj_crt.o      \
        kobj_isa.o      \
        kobj_reloc.o

#
#       misc. modules
#
ACPICA_OBJS     += dbcmds.o dbdisply.o \
                   dbexec.o dbfileio.o dbhistry.o dbinput.o dbstats.o \
                   dbutils.o dbxface.o evevent.o evgpe.o evgpeblk.o \
                   evmisc.o evregion.o evrgnini.o evsci.o evxface.o \
                   evxfevnt.o evxfregn.o hwacpi.o hwgpe.o hwregs.o \
                   hwsleep.o hwtimer.o dsfield.o dsinit.o dsmethod.o \
                   dsmthdat.o dsobject.o dsopcode.o dsutils.o dswexec.o \
                   dswload.o dswscope.o dswstate.o exconfig.o exconvrt.o \
                   excreate.o exdump.o exfield.o exfldio.o exmisc.o \
```



```
                   exmutex.o exnames.o exoparg1.o exoparg2.o exoparg3.o \
                   exoparg6.o exprep.o exregion.o exresnte.o exresolv.o \
                   exresop.o exstore.o exstoren.o exstorob.o exsystem.o \
                   exutils.o psargs.o psopcode.o psparse.o psscope.o \
                   pstree.o psutils.o pswalk.o psxface.o nsaccess.o \
                   nsalloc.o nsdump.o nsdumpdv.o nseval.o nsinit.o \
                   nsload.o nsnames.o nsobject.o nsparse.o nssearch.o \
                   nsutils.o nswalk.o nsxfeval.o nsxfname.o nsxfobj.o \
                   rsaddr.o rscalc.o rscreate.o rsdump.o \
                   rsinfo.o rsio.o rsirq.o rslist.o rsmemory.o rsmisc.o \
                   rsutils.o rsxface.o tbfadt.o tbfind.o tbinstal.o  \
                   tbutils.o tbxface.o tbxfroot.o \
                   utalloc.o utclib.o utcopy.o utdebug.o utdelete.o \
                   uteval.o utglobal.o utinit.o utmath.o utmisc.o \
                   utobject.o utresrc.o utxface.o acpica.o acpi_enum.o \
                   master_ops.o osl.o osl_ml.o acpica_ec.o utcache.o \
                   utmutex.o utstate.o dmbuffer.o dmnames.o dmobject.o \
                   dmopcode.o dmresrc.o dmresrcl.o dmresrcs.o dmutils.o \
                   dmwalk.o psloop.o nspredef.o hwxface.o hwvalid.o \
                   utlock.o utids.o nsrepair.o nsrepair2.o \
                   dbmethod.o dbnames.o dsargs.o dscontrol.o dswload2.o \
                   evglock.o evgpeinit.o evgpeutil.o evxfgpe.o exdebug.o \
                   hwpci.o utdecode.o utosi.o utxferror.o


AGP_OBJS += agpmaster.o
FBT_OBJS += fbt.o
SDT_OBJS += sdt.o

#
#       AMD8111 NIC driver module
#
AMD8111S_OBJS += amd8111s_main.o amd8111s_hw.o

#
#       Pentium Performance Counter BackEnd module
#
P123_PCBE_OBJS = p123_pcbe.o

#
#       Pentium 4 Performance Counter BackEnd module
#
P4_PCBE_OBJS = p4_pcbe.o

#
#       AMD Opteron/Athlon64 Performance Counter BackEnd module
#
OPTERON_PCBE_OBJS = opteron_pcbe.o

#
#       Intel Core Architecture Performance Counter BackEnd module
#
CORE_PCBE_OBJS = core_pcbe.o

#
#       AMR module
#
AMR_OBJS = amr.o

#
#       IPMI module
#
IPMI_OBJS +=    ipmi_main.o ipmi.o ipmi_kcs.o

#
#       IOMMULIB module
#
```

```
259 IOMMULIB_OBJS = iommulib.o

261 #
262 #       Brand modules
263 #
264 SN1_BRAND_OBJS  =       sn1_brand.o sn1_brand_asm.o
265 S10_BRAND_OBJS  =       s10_brand.o s10_brand_asm.o

267 #
268 #       special files
269 #
270 MODSTUB_OBJ +=          \
271         modstubs.o

273 BOOTDEV_OBJS +=         \
274         bootdev.o

276 INC_PATH        += -I$(UTSBASE)/intel


279 CPR_INTEL_OBJS +=       cpr_intel.o

281 #
282 # AMD family 0xf memory controller module
283 #
284 include $(SRC)/common/mc/mc-amd/Makefile.mcamd
285 MCAMD_OBJS      += \
286         $(MCAMD_CMN_OBJS) \
287         mcamd_drv.o \
288         mcamd_dimmcfg.o \
289         mcamd_subr.o \
290         mcamd_pcicfg.o
```

```
**********************************************************
    8493 Wed Jan 23 13:19:07 2013
new/usr/src/uts/intel/amd64/sys/privregs.h
XXX AVX procfs
**********************************************************
______unchanged_portion_omitted_

 104 #define r_r0    r_rax   /* r0 for portability */
 105 #define r_r1    r_rdx   /* r1 for portability */
 106 #define r_fp    r_rbp   /* kernel frame pointer */
 107 #define r_sp    r_rsp   /* user stack pointer */
 108 #define r_pc    r_rip   /* user's instruction pointer */
 109 #define r_ps    r_rfl   /* user's RFLAGS */

 111 #ifdef _KERNEL
 112 #define lwptoregs(lwp)  ((struct regs *)((lwp)->lwp_regs))
 113 #define lwptofpu(lwp)   ((kfpu_t *)((lwp)->lwp_fpu))
 114 #endif /* ! codereview */
 115 #endif /* _KERNEL */

 117 #else   /* !_ASM */

 119 #if defined(_MACHDEP)

 121 #include <sys/machprivregs.h>
 122 #include <sys/pcb.h>

 124 /*
 125  * We can not safely sample {fs,gs}base on the hypervisor. The rdmsr
 126  * instruction triggers a #gp fault which is emulated in the hypervisor
 127  * on behalf of the guest. This is normally ok but if the guest is in
 128  * the special failsafe handler it must not fault again or the hypervisor
 129  * will kill the domain. We could use something different than INTR_PUSH
 130  * in xen_failsafe_callback but for now we will not sample them.
 131  */
 132 #if defined(DEBUG) && !defined(__xpv)
 133 #define __SAVE_BASES                            \
 134         movl    $MSR_AMD_FSBASE, %ecx;          \
 135         rdmsr;                                  \
 136         movl    %eax, REGOFF_FSBASE(%rsp);      \
 137         movl    %edx, REGOFF_FSBASE+4(%rsp);    \
 138         movl    $MSR_AMD_GSBASE, %ecx;          \
 139         rdmsr;                                  \
 140         movl    %eax, REGOFF_GSBASE(%rsp);      \
 141         movl    %edx, REGOFF_GSBASE+4(%rsp)
 142 #else
 143 #define __SAVE_BASES
 144 #endif

 146 /*
 147  * Create a struct regs on the stack suitable for an
 148  * interrupt trap.
 149  *
 150  * Assumes that the trap handler has already pushed an
 151  * appropriate r_err and r_trapno
 152  */
 153 #define __SAVE_REGS                             \
 154         movq    %r15, REGOFF_R15(%rsp);         \
 155         movq    %r14, REGOFF_R14(%rsp);         \
 156         movq    %r13, REGOFF_R13(%rsp);         \
 157         movq    %r12, REGOFF_R12(%rsp);         \
 158         movq    %r11, REGOFF_R11(%rsp);         \
 159         movq    %r10, REGOFF_R10(%rsp);         \
 160         movq    %rbp, REGOFF_RBP(%rsp);         \
 161         movq    %rbx, REGOFF_RBX(%rsp);         \
 162         movq    %rax, REGOFF_RAX(%rsp);         \
```



```
 163         movq    %r9, REGOFF_R9(%rsp);           \
 164         movq    %r8, REGOFF_R8(%rsp);           \
 165         movq    %rcx, REGOFF_RCX(%rsp);         \
 166         movq    %rdx, REGOFF_RDX(%rsp);         \
 167         movq    %rsi, REGOFF_RSI(%rsp);         \
 168         movq    %rdi, REGOFF_RDI(%rsp);         \
 169         movq    %rbp, REGOFF_SAVFP(%rsp);       \
 170         movq    REGOFF_RIP(%rsp), %rcx;         \
 171         movq    %rcx, REGOFF_SAVPC(%rsp);       \
 172         xorl    %ecx, %ecx;                     \
 173         movw    %gs, %cx;                       \
 174         movq    %rcx, REGOFF_GS(%rsp);          \
 175         movw    %fs, %cx;                       \
 176         movq    %rcx, REGOFF_FS(%rsp);          \
 177         movw    %es, %cx;                       \
 178         movq    %rcx, REGOFF_ES(%rsp);          \
 179         movw    %ds, %cx;                       \
 180         movq    %rcx, REGOFF_DS(%rsp);          \
 181         __SAVE_BASES

 183 #define __RESTORE_REGS                          \
 184         movq    REGOFF_RDI(%rsp),       %rdi;   \
 185         movq    REGOFF_RSI(%rsp),       %rsi;   \
 186         movq    REGOFF_RDX(%rsp),       %rdx;   \
 187         movq    REGOFF_RCX(%rsp),       %rcx;   \
 188         movq    REGOFF_R8(%rsp),        %r8;    \
 189         movq    REGOFF_R9(%rsp),        %r9;    \
 190         movq    REGOFF_RAX(%rsp),       %rax;   \
 191         movq    REGOFF_RBX(%rsp),       %rbx;   \
 192         movq    REGOFF_RBP(%rsp),       %rbp;   \
 193         movq    REGOFF_R10(%rsp),       %r10;   \
 194         movq    REGOFF_R11(%rsp),       %r11;   \
 195         movq    REGOFF_R12(%rsp),       %r12;   \
 196         movq    REGOFF_R13(%rsp),       %r13;   \
 197         movq    REGOFF_R14(%rsp),       %r14;   \
 198         movq    REGOFF_R15(%rsp),       %r15

 200 /*
 201  * Push register state onto the stack. If we've
 202  * interrupted userland, do a swapgs as well.
 203  */
 204 #define INTR_PUSH                               \
 205         subq    $REGOFF_TRAPNO, %rsp;           \
 206         __SAVE_REGS;                            \
 207         cmpw    $KCS_SEL, REGOFF_CS(%rsp);      \
 208         je      6f;                             \
 209         movq    $0, REGOFF_SAVFP(%rsp);         \
 210         SWAPGS;                                 \
 211 6:      CLEAN_CS

 213 #define INTR_POP                                \
 214         leaq    sys_lcall32(%rip), %r11;\
 215         cmpq    %r11, REGOFF_RIP(%rsp); \
 216         __RESTORE_REGS;                 \
 217         je      5f;                     \
 218         cmpw    $KCS_SEL, REGOFF_CS(%rsp);\
 219         je      8f;                     \
 220 5:      SWAPGS;                         \
 221 8:      addq    $REGOFF_RIP, %rsp

 223 #define USER_POP                        \
 224         __RESTORE_REGS;                 \
 225         SWAPGS;                         \
 226         addq    $REGOFF_RIP, %rsp       /* Adjust %rsp to prepare for iretq */

 228 #define USER32_POP                      \
```

```
229         movl    REGOFF_RDI(%rsp), %edi; \
230         movl    REGOFF_RSI(%rsp), %esi; \
231         movl    REGOFF_RDX(%rsp), %edx; \
232         movl    REGOFF_RCX(%rsp), %ecx; \
233         movl    REGOFF_RAX(%rsp), %eax; \
234         movl    REGOFF_RBX(%rsp), %ebx; \
235         movl    REGOFF_RBP(%rsp), %ebp; \
236         SWAPGS;                         \
237         addq    $REGOFF_RIP, %rsp       /* Adjust %rsp to prepare for iretq */

239 #define DFTRAP_PUSH                             \
240         subq    $REGOFF_TRAPNO, %rsp;           \
241         __SAVE_REGS

243 #endif  /* _MACHDEP */

245 /*
246  * Used to set rflags to known values at the head of an
247  * interrupt gate handler, i.e. interrupts are -already- disabled.
248  */
249 #define INTGATE_INIT_KERNEL_FLAGS       \
250         pushq   $F_OFF;                 \
251         popfq

253 #endif  /* !_ASM */

255 #include <sys/controlregs.h>

257 #if defined(_KERNEL) && !defined(_ASM)
258 #if !defined(__lint) && defined(__GNUC__)

260 extern __GNU_INLINE ulong_t
261 getcr8(void)
262 {
263         uint64_t value;

265         __asm__ __volatile__(
266             "movq %%cr8, %0"
267             : "=r" (value));
268         return (value);
269 }

271 extern __GNU_INLINE void
272 setcr8(ulong_t value)
273 {
274         __asm__ __volatile__(
275             "movq %0, %%cr8"
276             : /* no output */
277             : "r" (value));
278 }

280 #else

282 extern ulong_t getcr8(void);
283 extern void setcr8(ulong_t);

285 #endif  /* !defined(__lint) && defined(__GNUC__) */
286 #endif  /* _KERNEL && !_ASM */

288 /* Control register layout for panic dump */

290 #define CREGSZ          0x68
291 #define CREG_GDT        0
292 #define CREG_IDT        0x10
293 #define CREG_LDT        0x20
294 #define CREG_TASKR      0x28
```



```
295 #define CREG_CR0        0x30
296 #define CREG_CR2        0x38
297 #define CREG_CR3        0x40
298 #define CREG_CR4        0x48
299 #define CREG_CR8        0x50
300 #define CREG_KGSBASE    0x58
301 #define CREG_EFER       0x60

303 #if !defined(_ASM) && defined(_INT64_TYPE)

305 typedef uint64_t        creg64_t;
306 typedef upad128_t       creg128_t;

308 struct cregs {
309         creg128_t       cr_gdt;
310         creg128_t       cr_idt;
311         creg64_t        cr_ldt;
312         creg64_t        cr_task;
313         creg64_t        cr_cr0;
314         creg64_t        cr_cr2;
315         creg64_t        cr_cr3;
316         creg64_t        cr_cr4;
317         creg64_t        cr_cr8;
318         creg64_t        cr_kgsbase;
319         creg64_t        cr_efer;
320 };

322 #if defined(_KERNEL)
323 extern void getcregs(struct cregs *);
324 #endif  /* _KERNEL */

326 #endif  /* !_ASM && _INT64_TYPE */

328 #ifdef __cplusplus
329 }
330 #endif

332 #endif  /* !_AMD64_SYS_PRIVREGS_H */
```

```
**********************************************************
   12721 Wed Jan 23 13:19:08 2013
new/usr/src/uts/intel/fs/proc/prmachdep.c
XXX AVX procfs
**********************************************************
______unchanged_portion_omitted_
 236 #endif  /* _SYSCALL32_IMPL */

 238 /*
 239  * Does the system support extra register state?
 240  */
 241 /* ARGSUSED */
 241 int
 242 prhasx(proc_t *p)
 243 {
 244         /* XXX */
 245         return (1);
 245         return (0);
 246 }

 248 /*
 249  * Get the size of the extra registers.
 250  */
 251 /* ARGSUSED */
 251 int
 252 prgetprxregsize(proc_t *p)
 253 {
 254         return (xregs_getsize(p));
 255         return (0);
 255 }

 257 /*
 258  * Get extra registers.
 259  */
 261 /*ARGSUSED*/
 260 void
 261 prgetprxregs(klwp_t *lwp, caddr_t prx)
 262 {
 263         extern void xregs_get(struct _klwp *, caddr_t);

 265         xregs_get(lwp, prx);
 265         /* no extra registers */
 266 }

 268 /*
 269  * Set extra registers.
 270  */
 271 /*ARGSUSED*/
 271 void
 272 prsetprxregs(klwp_t *lwp, caddr_t prx)
 273 {
 274         extern void xregs_set(struct _klwp *, caddr_t);

 276         xregs_set(lwp, prx);
 275         /* no extra registers */
 277 }
______unchanged_portion_omitted_
```

```
**********************************************************
   22655 Wed Jan 23 13:19:08 2013
new/usr/src/uts/intel/ia32/os/sendsig.c
XXX AVX procfs
**********************************************************
______unchanged_portion_omitted_

 126 int
 127 sendsig(int sig, k_siginfo_t *sip, void (*hdlr)())
 128 {
 129         volatile int minstacksz;
 130         int newstack;
 131         label_t ljb;
 132         volatile caddr_t sp;
 133         caddr_t fp;
 134         volatile struct regs *rp;
 135         volatile greg_t upc;
 136         proc_t *volatile p = ttoproc(curthread);
 136         volatile proc_t *p = ttoproc(curthread);
 137         struct as *as = p->p_as;
 138         klwp_t *lwp = ttolwp(curthread);
 139         ucontext_t *volatile tuc = NULL;
 140         ucontext_t *uc;
 141         siginfo_t *sip_addr;
 142         volatile int watched;
 143         char *volatile xregs = NULL;
 144         volatile size_t xregs_size = 0;
 145 #endif /* ! codereview */

 147         /*
 148          * This routine is utterly dependent upon STACK_ALIGN being
 149          * 16 and STACK_ENTRY_ALIGN being 8. Let's just acknowledge
 150          * that and require it.
 151          */

 153 #if STACK_ALIGN != 16 || STACK_ENTRY_ALIGN != 8
 154 #error "sendsig() amd64 did not find the expected stack alignments"
 155 #endif

 157         rp = lwptoregs(lwp);
 158         upc = rp->r_pc;

 160         /*
 161          * Since we're setting up to run the signal handler we have to
 162          * arrange that the stack at entry to the handler is (only)
 163          * STACK_ENTRY_ALIGN (i.e. 8) byte aligned so that when the handler
 164          * executes its push of %rbp, the stack realigns to STACK_ALIGN
 165          * (i.e. 16) correctly.
 166          *
 167          * The new sp will point to the sigframe and the ucontext_t. The
 168          * above means that sp (and thus sigframe) will be 8-byte aligned,
 169          * but not 16-byte aligned. ucontext_t, however, contains %xmm regs
 170          * which must be 16-byte aligned. Because of this, for correct
 171          * alignment, sigframe must be a multiple of 8-bytes in length, but
 172          * not 16-bytes. This will place ucontext_t at a nice 16-byte boundary.
 173          */

 175         /* LINTED: logical expression always true: op "||" */
 176         ASSERT((sizeof (struct sigframe) % 16) == 8);

 178         minstacksz = sizeof (struct sigframe) + SA(sizeof (*uc));
 179         if (sip != NULL)
 180                 minstacksz += SA(sizeof (siginfo_t));

 182         /*
 183          * Extra registers, if supported by this platform, may be of arbitrary
```



```
 184          * length. Size them now so we know how big the signal frame has to be.
 185          */
 186         xregs_size = xregs_getsize(p);
 187         minstacksz += SA(xregs_size);

 189 #endif /* ! codereview */
 190         ASSERT((minstacksz & (STACK_ENTRY_ALIGN - 1ul)) == 0);

 192         /*
 193          * Figure out whether we will be handling this signal on
 194          * an alternate stack specified by the user.  Then allocate
 195          * and validate the stack requirements for the signal handler
 196          * context.  on_fault will catch any faults.
 197          */
 198         newstack = sigismember(&PTOU(curproc)->u_sigonstack, sig) &&
 199             !(lwp->lwp_sigaltstack.ss_flags & (SS_ONSTACK|SS_DISABLE));

 201         if (newstack) {
 202                 fp = (caddr_t)(SA((uintptr_t)lwp->lwp_sigaltstack.ss_sp) +
 203                     SA(lwp->lwp_sigaltstack.ss_size) - STACK_ALIGN);
 204         } else {
 205                 /*
 206                  * Drop below the 128-byte reserved region of the stack frame
 207                  * we're interrupting.
 208                  */
 209                 fp = (caddr_t)rp->r_sp - STACK_RESERVE;
 210         }

 212         /*
 213          * Force proper stack pointer alignment, even in the face of a
 214          * misaligned stack pointer from user-level before the signal.
 215          */
 216         fp = (caddr_t)((uintptr_t)fp & ~(STACK_ENTRY_ALIGN - 1ul));

 218         /*
 219          * Most of the time during normal execution, the stack pointer
 220          * is aligned on a STACK_ALIGN (i.e. 16 byte) boundary.  However,
 221          * (for example) just after a call instruction (which pushes
 222          * the return address), the callers stack misaligns until the
 223          * 'push %rbp' happens in the callee prolog.  So while we should
 224          * expect the stack pointer to be always at least STACK_ENTRY_ALIGN
 225          * aligned, we should -not- expect it to always be STACK_ALIGN aligned.
 226          * We now adjust to ensure that the new sp is aligned to
 227          * STACK_ENTRY_ALIGN but not to STACK_ALIGN.
 228          */
 229         sp = fp - minstacksz;
 230         if (((uintptr_t)sp & (STACK_ALIGN - 1ul)) == 0) {
 231                 sp -= STACK_ENTRY_ALIGN;
 232                 minstacksz = fp - sp;
 233         }

 235         /*
 236          * Now, make sure the resulting signal frame address is sane
 237          */
 238         if (sp >= as->a_userlimit || fp >= as->a_userlimit) {
 239 #ifdef DEBUG
 240                 printf("sendsig: bad signal stack cmd=%s, pid=%d, sig=%d\n",
 241                     PTOU(p)->u_comm, p->p_pid, sig);
 242                 printf("sigsp = 0x%p, action = 0x%p, upc = 0x%lx\n",
 243                     (void *)sp, (void *)hdlr, (uintptr_t)upc);
 244                 printf("sp above USERLIMIT\n");
 245 #endif
 246                 return (0);
 247         }

 249         watched = watch_disable_addr((caddr_t)sp, minstacksz, S_WRITE);
```

```
251         if (on_fault(&ljb))
252                 goto badstack;
253
254         if (sip != NULL) {
255                 zoneid_t zoneid;
256
257                 fp -= SA(sizeof (siginfo_t));
258                 uzero(fp, sizeof (siginfo_t));
259                 if (SI_FROMUSER(sip) &&
260                     (zoneid = p->p_zone->zone_id) != GLOBAL_ZONEID &&
261                     zoneid != sip->si_zoneid) {
262                         k_siginfo_t sani_sip = *sip;
263
264                         sani_sip.si_pid = p->p_zone->zone_zsched->p_pid;
265                         sani_sip.si_uid = 0;
266                         sani_sip.si_ctid = -1;
267                         sani_sip.si_zoneid = zoneid;
268                         copyout_noerr(&sani_sip, fp, sizeof (sani_sip));
269                 } else
270                         copyout_noerr(sip, fp, sizeof (*sip));
271                 sip_addr = (siginfo_t *)fp;
272
273                 if (sig == SIGPROF &&
274                     curthread->t_rprof != NULL &&
275                     curthread->t_rprof->rp_anystate) {
276                         /*
277                          * We stand on our head to deal with
278                          * the real time profiling signal.
279                          * Fill in the stuff that doesn't fit
280                          * in a normal k_siginfo structure.
281                          */
282                         int i = sip->si_nsysarg;
283
284                         while (--i >= 0)
285                                 sulword_noerr(
286                                     (ulong_t *)&(sip_addr->si_sysarg[i]),
287                                     (ulong_t)lwp->lwp_arg[i]);
288                         copyout_noerr(curthread->t_rprof->rp_state,
289                             sip_addr->si_mstate,
290                             sizeof (curthread->t_rprof->rp_state));
291                 }
292         } else
293                 sip_addr = NULL;
294
295         /*
296          * save the current context on the user stack directly after the
297          * sigframe. Since sigframe is 8-byte-but-not-16-byte aligned,
298          * and since sizeof (struct sigframe) is 24, this guarantees
299          * 16-byte alignment for ucontext_t and its %xmm registers.
300          */
301         uc = (ucontext_t *)(sp + sizeof (struct sigframe));
302         tuc = kmem_alloc(sizeof (*tuc), KM_SLEEP);
303         savecontext(tuc, &lwp->lwp_sigoldmask);
304
305         /*
306          * Save extra register state if it exists.
307          */
308         if (xregs_size != 0) {
309                 xregs_setptr(lwp, tuc, sp);
310                 xregs = kmem_alloc(xregs_size, KM_SLEEP);
311                 xregs_get(lwp, xregs);
312                 copyout_noerr(xregs, sp, xregs_size);
313                 kmem_free(xregs, xregs_size);
314                 xregs = NULL;
315                 sp += SA(xregs_size);
```



```
316         }
317
318 #endif /* ! codereview */
319         copyout_noerr(tuc, uc, sizeof (*tuc));
320         kmem_free(tuc, sizeof (*tuc));
321         tuc = NULL;
322
323         lwp->lwp_oldcontext = (uintptr_t)uc;
324
325         if (newstack) {
326                 lwp->lwp_sigaltstack.ss_flags |= SS_ONSTACK;
327                 if (lwp->lwp_ustack)
328                         copyout_noerr(&lwp->lwp_sigaltstack,
329                             (stack_t *)lwp->lwp_ustack, sizeof (stack_t));
330         }
331
332         /*
333          * Set up signal handler return and stack linkage
334          */
335         {
336                 struct sigframe frame;
337
338                 /*
339                  * ensure we never return "normally"
340                  */
341                 frame.retaddr = (caddr_t)(uintptr_t)-1L;
342                 frame.signo = sig;
343                 frame.sip = sip_addr;
344                 copyout_noerr(&frame, sp, sizeof (frame));
345         }
346
347         no_fault();
348         if (watched)
349                 watch_enable_addr((caddr_t)sp, minstacksz, S_WRITE);
350
351         /*
352          * Set up user registers for execution of signal handler.
353          */
354         rp->r_sp = (greg_t)sp;
355         rp->r_pc = (greg_t)hdlr;
356         rp->r_ps = PSL_USER | (rp->r_ps & PS_IOPL);
357
358         rp->r_rdi = sig;
359         rp->r_rsi = (uintptr_t)sip_addr;
360         rp->r_rdx = (uintptr_t)uc;
361
362         if ((rp->r_cs & 0xffff) != UCS_SEL ||
363             (rp->r_ss & 0xffff) != UDS_SEL) {
364                 /*
365                  * Try our best to deliver the signal.
366                  */
367                 rp->r_cs = UCS_SEL;
368                 rp->r_ss = UDS_SEL;
369         }
370
371         /*
372          * Don't set lwp_eosys here.  sendsig() is called via psig() after
373          * lwp_eosys is handled, so setting it here would affect the next
374          * system call.
375          */
376         return (1);
377
378 badstack:
379         no_fault();
380         if (watched)
381                 watch_enable_addr((caddr_t)sp, minstacksz, S_WRITE);
```

```
382         if (tuc)
383                 kmem_free(tuc, sizeof (*tuc));
384         if (xregs)
385                 kmem_free(xregs, xregs_size);
386 #endif /* ! codereview */
387 #ifdef DEBUG
388         printf("sendsig: bad signal stack cmd=%s, pid=%d, sig=%d\n",
389             PTOU(p)->u_comm, p->p_pid, sig);
390         printf("on fault, sigsp = 0x%p, action = 0x%p, upc = 0x%lx\n",
391             (void *)sp, (void *)hdlr, (uintptr_t)upc);
392 #endif
393         return (0);
394 }

396 #ifdef _SYSCALL32_IMPL

398 /*
399  * An i386 SVR4/ABI signal frame looks like this on the stack:
400  *
401  * old %esp:
402  *              <a siginfo32_t [optional]>
403  *              <a ucontext32_t>
404  *              <pointer to that ucontext32_t>
405  *              <pointer to that siginfo32_t>
406  *              <signo>
407  * new %esp:    <return address (deliberately invalid)>
408  */
409 struct sigframe32 {
410         caddr32_t       retaddr;
411         uint32_t        signo;
412         caddr32_t       sip;
413         caddr32_t       ucp;
414 };

416 int
417 sendsig32(int sig, k_siginfo_t *sip, void (*hdlr)())
418 {
419         volatile int minstacksz;
420         int newstack;
421         label_t ljb;
422         volatile caddr_t sp;
423         caddr_t fp;
424         volatile struct regs *rp;
425         volatile greg_t upc;
426         proc_t *volatile p = ttoproc(curthread);
143         volatile proc_t *p = ttoproc(curthread);
427         klwp_t *lwp = ttolwp(curthread);
428         ucontext32_t *volatile tuc = NULL;
429         ucontext32_t *uc;
430         siginfo32_t *sip_addr;
431         volatile int watched;
432         char *volatile xregs = NULL;
433         volatile size_t xregs_size = 0;
434 #endif /* ! codereview */

436         rp = lwptoregs(lwp);
437         upc = rp->r_pc;

439         minstacksz = SA32(sizeof (struct sigframe32)) + SA32(sizeof (*uc));
440         if (sip != NULL)
441                 minstacksz += SA32(sizeof (siginfo32_t));

443         /*
444          * Extra registers, if supported by this platform, may be of arbitrary
445          * length. Size them now so we know how big the signal frame has to be.
446          */
```



```
447         xregs_size = xregs_getsize(p);
448         minstacksz += SA32(xregs_size);

450 #endif /* ! codereview */
451         ASSERT((minstacksz & (STACK_ALIGN32 - 1)) == 0);

453         /*
454          * Figure out whether we will be handling this signal on
455          * an alternate stack specified by the user.  Then allocate
456          * and validate the stack requirements for the signal handler
457          * context.  on_fault will catch any faults.
458          */
459         newstack = sigismember(&PTOU(curproc)->u_sigonstack, sig) &&
460             !(lwp->lwp_sigaltstack.ss_flags & (SS_ONSTACK|SS_DISABLE));

462         if (newstack) {
463                 fp = (caddr_t)(SA32((uintptr_t)lwp->lwp_sigaltstack.ss_sp) +
464                     SA32(lwp->lwp_sigaltstack.ss_size) - STACK_ALIGN32);
465         } else if ((rp->r_ss & 0xffff) != UDS_SEL) {
466                 user_desc_t *ldt;
467                 /*
468                  * If the stack segment selector is -not- pointing at
469                  * the UDS_SEL descriptor and we have an LDT entry for
470                  * it instead, add the base address to find the effective va.
471                  */
472                 if ((ldt = p->p_ldt) != NULL)
473                         fp = (caddr_t)rp->r_sp +
474                             USEGD_GETBASE(&ldt[SELTOIDX(rp->r_ss)]);
475                 else
476                         fp = (caddr_t)rp->r_sp;
477         } else
478                 fp = (caddr_t)rp->r_sp;

480         /*
481          * Force proper stack pointer alignment, even in the face of a
482          * misaligned stack pointer from user-level before the signal.
483          * Don't use the SA32() macro because that rounds up, not down.
484          */
485         fp = (caddr_t)((uintptr_t)fp & ~(STACK_ALIGN32 - 1));
486         sp = fp - minstacksz;

488         /*
489          * Make sure lwp hasn't trashed its stack
490          */
491         if (sp >= (caddr_t)(uintptr_t)USERLIMIT32 ||
492             fp >= (caddr_t)(uintptr_t)USERLIMIT32) {
493 #ifdef DEBUG
494                 printf("sendsig32: bad signal stack cmd=%s, pid=%d, sig=%d\n",
495                     PTOU(p)->u_comm, p->p_pid, sig);
496                 printf("sigsp = 0x%p, action = 0x%p, upc = 0x%lx\n",
497                     (void *)sp, (void *)hdlr, (uintptr_t)upc);
498                 printf("sp above USERLIMIT\n");
499 #endif
500                 return (0);
501         }

503         watched = watch_disable_addr((caddr_t)sp, minstacksz, S_WRITE);

505         if (on_fault(&ljb))
506                 goto badstack;

508         if (sip != NULL) {
509                 siginfo32_t si32;
510                 zoneid_t zoneid;

512                 siginfo_kto32(sip, &si32);
```

```
513                 if (SI_FROMUSER(sip) &&
514                     (zoneid = p->p_zone->zone_id) != GLOBAL_ZONEID &&
515                     zoneid != sip->si_zoneid) {
516                         si32.si_pid = p->p_zone->zone_zsched->p_pid;
517                         si32.si_uid = 0;
518                         si32.si_ctid = -1;
519                         si32.si_zoneid = zoneid;
520                 }
521                 fp -= SA32(sizeof (si32));
522                 uzero(fp, sizeof (si32));
523                 copyout_noerr(&si32, fp, sizeof (si32));
524                 sip_addr = (siginfo32_t *)fp;

526                 if (sig == SIGPROF &&
527                     curthread->t_rprof != NULL &&
528                     curthread->t_rprof->rp_anystate) {
529                         /*
530                          * We stand on our head to deal with
531                          * the real-time profiling signal.
532                          * Fill in the stuff that doesn't fit
533                          * in a normal k_siginfo structure.
534                          */
535                         int i = sip->si_nsysarg;

537                         while (--i >= 0)
538                                 suword32_noerr(&(sip_addr->si_sysarg[i]),
539                                     (uint32_t)lwp->lwp_arg[i]);
540                         copyout_noerr(curthread->t_rprof->rp_state,
541                             sip_addr->si_mstate,
542                             sizeof (curthread->t_rprof->rp_state));
543                 }
544         } else
545                 sip_addr = NULL;

547         /* save the current context on the user stack */
548         fp -= SA32(sizeof (*tuc));
549         uc = (ucontext32_t *)fp;
550         tuc = kmem_alloc(sizeof (*tuc), KM_SLEEP);
551         savecontext32(tuc, &lwp->lwp_sigoldmask);

553         /*
554          * Save extra register state if it exists.
555          */
556         if (xregs_size != 0) {
557                 xregs_setptr32(lwp, tuc, (caddr32_t)(uintptr_t)sp);
558                 xregs = kmem_alloc(xregs_size, KM_SLEEP);
559                 xregs_get(lwp, xregs);
560                 copyout_noerr(xregs, sp, xregs_size);
561                 kmem_free(xregs, xregs_size);
562                 xregs = NULL;
563                 sp += SA32(xregs_size);
564         }

566 #endif /* ! codereview */
567         copyout_noerr(tuc, uc, sizeof (*tuc));
568         kmem_free(tuc, sizeof (*tuc));
569         tuc = NULL;

571         lwp->lwp_oldcontext = (uintptr_t)uc;

573         if (newstack) {
574                 lwp->lwp_sigaltstack.ss_flags |= SS_ONSTACK;
575                 if (lwp->lwp_ustack) {
576                         stack32_t stk32;

578                         stk32.ss_sp = (caddr32_t)(uintptr_t)
```



```
579                             lwp->lwp_sigaltstack.ss_sp;
580                         stk32.ss_size = (size32_t)
581                             lwp->lwp_sigaltstack.ss_size;
582                         stk32.ss_flags = (int32_t)
583                             lwp->lwp_sigaltstack.ss_flags;
584                         copyout_noerr(&stk32,
585                             (stack32_t *)lwp->lwp_ustack, sizeof (stk32));
586                 }
587         }

589         /*
590          * Set up signal handler arguments
591          */
592         {
593                 struct sigframe32 frame32;

595                 frame32.sip = (caddr32_t)(uintptr_t)sip_addr;
596                 frame32.ucp = (caddr32_t)(uintptr_t)uc;
597                 frame32.signo = sig;
598                 frame32.retaddr = 0xffffffff;   /* never return! */
599                 copyout_noerr(&frame32, sp, sizeof (frame32));
600         }

602         no_fault();
603         if (watched)
604                 watch_enable_addr((caddr_t)sp, minstacksz, S_WRITE);

606         rp->r_sp = (greg_t)(uintptr_t)sp;
607         rp->r_pc = (greg_t)(uintptr_t)hdlr;
608         rp->r_ps = PSL_USER | (rp->r_ps & PS_IOPL);

610         if ((rp->r_cs & 0xffff) != U32CS_SEL ||
611             (rp->r_ss & 0xffff) != UDS_SEL) {
612                 /*
613                  * Try our best to deliver the signal.
614                  */
615                 rp->r_cs = U32CS_SEL;
616                 rp->r_ss = UDS_SEL;
617         }

619         /*
620          * Don't set lwp_eosys here.  sendsig() is called via psig() after
621          * lwp_eosys is handled, so setting it here would affect the next
622          * system call.
623          */
624         return (1);

626 badstack:
627         no_fault();
628         if (watched)
629                 watch_enable_addr((caddr_t)sp, minstacksz, S_WRITE);
630         if (tuc)
631                 kmem_free(tuc, sizeof (*tuc));
632         if (xregs_size)
633                 kmem_free(xregs, xregs_size);
634 #endif /* ! codereview */
635 #ifdef DEBUG
636         printf("sendsig32: bad signal stack cmd=%s pid=%d, sig=%d\n",
637             PTOU(p)->u_comm, p->p_pid, sig);
638         printf("on fault, sigsp = 0x%p, action = 0x%p, upc = 0x%lx\n",
639             (void *)sp, (void *)hdlr, (uintptr_t)upc);
640 #endif
641         return (0);
642 }

644 #endif  /* _SYSCALL32_IMPL */
```

```
646 #elif defined(__i386)

648 /*
649  * An i386 SVR4/ABI signal frame looks like this on the stack:
650  *
651  * old %esp:
652  *              <a siginfo32_t [optional]>
653  *              <a ucontext32_t>
654  *              <pointer to that ucontext32_t>
655  *              <pointer to that siginfo32_t>
656  *              <signo>
657  * new %esp:    <return address (deliberately invalid)>
658  */
659 struct sigframe {
660         void            (*retaddr)();
661         uint_t          signo;
662         siginfo_t       *sip;
663         ucontext_t      *ucp;
664 };

666 int
667 sendsig(int sig, k_siginfo_t *sip, void (*hdlr)())
668 {
669         volatile int minstacksz;
670         int newstack;
671         label_t ljb;
672         volatile caddr_t sp;
673         caddr_t fp;
674         struct regs *rp;
675         volatile greg_t upc;
676         volatile proc_t *p = ttoproc(curthread);
677         klwp_t *lwp = ttolwp(curthread);
678         ucontext_t *volatile tuc = NULL;
679         ucontext_t *uc;
680         siginfo_t *sip_addr;
681         volatile int watched;

683         rp = lwptoregs(lwp);
684         upc = rp->r_pc;

686         minstacksz = SA(sizeof (struct sigframe)) + SA(sizeof (*uc));
687         if (sip != NULL)
688                 minstacksz += SA(sizeof (siginfo_t));
689         ASSERT((minstacksz & (STACK_ALIGN - 1ul)) == 0);

691         /*
692          * Figure out whether we will be handling this signal on
693          * an alternate stack specified by the user. Then allocate
694          * and validate the stack requirements for the signal handler
695          * context. on_fault will catch any faults.
696          */
697         newstack = sigismember(&PTOU(curproc)->u_sigonstack, sig) &&
698             !(lwp->lwp_sigaltstack.ss_flags & (SS_ONSTACK|SS_DISABLE));

700         if (newstack) {
701                 fp = (caddr_t)(SA((uintptr_t)lwp->lwp_sigaltstack.ss_sp) +
702                     SA(lwp->lwp_sigaltstack.ss_size) - STACK_ALIGN);
703         } else if ((rp->r_ss & 0xffff) != UDS_SEL) {
704                 user_desc_t *ldt;
705                 /*
706                  * If the stack segment selector is -not- pointing at
707                  * the UDS_SEL descriptor and we have an LDT entry for
708                  * it instead, add the base address to find the effective va.
709                  */
710                 if ((ldt = p->p_ldt) != NULL)
```



```
711                         fp = (caddr_t)rp->r_sp +
712                             USEGD_GETBASE(&ldt[SELTOIDX(rp->r_ss)]);
713                 else
714                         fp = (caddr_t)rp->r_sp;
715         } else
716                 fp = (caddr_t)rp->r_sp;

718         /*
719          * Force proper stack pointer alignment, even in the face of a
720          * misaligned stack pointer from user-level before the signal.
721          * Don't use the SA() macro because that rounds up, not down.
722          */
723         fp = (caddr_t)((uintptr_t)fp & ~(STACK_ALIGN - 1ul));
724         sp = fp - minstacksz;

726         /*
727          * Make sure lwp hasn't trashed its stack.
728          */
729         if (sp >= (caddr_t)USERLIMIT || fp >= (caddr_t)USERLIMIT) {
730 #ifdef DEBUG
731                 printf("sendsig: bad signal stack cmd=%s, pid=%d, sig=%d\n",
732                     PTOU(p)->u_comm, p->p_pid, sig);
733                 printf("sigsp = 0x%p, action = 0x%p, upc = 0x%lx\n",
734                     (void *)sp, (void *)hdlr, (uintptr_t)upc);
735                 printf("sp above USERLIMIT\n");
736 #endif
737                 return (0);
738         }

740         watched = watch_disable_addr((caddr_t)sp, minstacksz, S_WRITE);

742         if (on_fault(&ljb))
743                 goto badstack;

745         if (sip != NULL) {
746                 zoneid_t zoneid;

748                 fp -= SA(sizeof (siginfo_t));
749                 uzero(fp, sizeof (siginfo_t));
750                 if (SI_FROMUSER(sip) &&
751                     (zoneid = p->p_zone->zone_id) != GLOBAL_ZONEID &&
752                     zoneid != sip->si_zoneid) {
753                         k_siginfo_t sani_sip = *sip;

755                         sani_sip.si_pid = p->p_zone->zone_zsched->p_pid;
756                         sani_sip.si_uid = 0;
757                         sani_sip.si_ctid = -1;
758                         sani_sip.si_zoneid = zoneid;
759                         copyout_noerr(&sani_sip, fp, sizeof (sani_sip));
760                 } else
761                         copyout_noerr(sip, fp, sizeof (*sip));
762                 sip_addr = (siginfo_t *)fp;

764                 if (sig == SIGPROF &&
765                     curthread->t_rprof != NULL &&
766                     curthread->t_rprof->rp_anystate) {
767                         /*
768                          * We stand on our head to deal with
769                          * the real time profiling signal.
770                          * Fill in the stuff that doesn't fit
771                          * in a normal k_siginfo structure.
772                          */
773                         int i = sip->si_nsysarg;

775                         while (--i >= 0)
776                                 suword32_noerr(&(sip_addr->si_sysarg[i]),
```

```
777                                     (uint32_t)lwp->lwp_arg[i]);
778                             copyout_noerr(curthread->t_rprof->rp_state,
779                                 sip_addr->si_mstate,
780                                 sizeof (curthread->t_rprof->rp_state));
781                     }
782             } else
783                     sip_addr = NULL;
784
785             /* save the current context on the user stack */
786             fp -= SA(sizeof (*tuc));
787             uc = (ucontext_t *)fp;
788             tuc = kmem_alloc(sizeof (*tuc), KM_SLEEP);
789             savecontext(tuc, &lwp->lwp_sigoldmask);
790             copyout_noerr(tuc, uc, sizeof (*tuc));
791             kmem_free(tuc, sizeof (*tuc));
792             tuc = NULL;
793
794             lwp->lwp_oldcontext = (uintptr_t)uc;
795
796             if (newstack) {
797                     lwp->lwp_sigaltstack.ss_flags |= SS_ONSTACK;
798                     if (lwp->lwp_ustack)
799                             copyout_noerr(&lwp->lwp_sigaltstack,
800                                 (stack_t *)lwp->lwp_ustack, sizeof (stack_t));
801             }
802
803             /*
804              * Set up signal handler arguments
805              */
806             {
807                     struct sigframe frame;
808
809                     frame.sip = sip_addr;
810                     frame.ucp = uc;
811                     frame.signo = sig;
812                     frame.retaddr = (void (*)())0xffffffff; /* never return! */
813                     copyout_noerr(&frame, sp, sizeof (frame));
814             }
815
816             no_fault();
817             if (watched)
818                     watch_enable_addr((caddr_t)sp, minstacksz, S_WRITE);
819
820             rp->r_sp = (greg_t)sp;
821             rp->r_pc = (greg_t)hdlr;
822             rp->r_ps = PSL_USER | (rp->r_ps & PS_IOPL);
823
824             if ((rp->r_cs & 0xffff) != UCS_SEL ||
825                 (rp->r_ss & 0xffff) != UDS_SEL) {
826                     rp->r_cs = UCS_SEL;
827                     rp->r_ss = UDS_SEL;
828             }
829
830             /*
831              * Don't set lwp_eosys here.  sendsig() is called via psig() after
832              * lwp_eosys is handled, so setting it here would affect the next
833              * system call.
834              */
835             return (1);
836
837     badstack:
838             no_fault();
839             if (watched)
840                     watch_enable_addr((caddr_t)sp, minstacksz, S_WRITE);
841             if (tuc)
842                     kmem_free(tuc, sizeof (*tuc));
```



```
843 #ifdef DEBUG
844             printf("sendsig: bad signal stack cmd=%s, pid=%d, sig=%d\n",
845                 PTOU(p)->u_comm, p->p_pid, sig);
846             printf("on fault, sigsp = 0x%p, action = 0x%p, upc = 0x%lx\n",
847                 (void *)sp, (void *)hdlr, (uintptr_t)upc);
848 #endif
849             return (0);
850 }
851
852 #endif  /* __i386 */
```

```
**********************************************************
    4681 Wed Jan 23 13:19:09 2013
new/usr/src/uts/intel/ia32/os/xregs.c
XXX AVX procfs
**********************************************************
```

```
/*
 * 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 1994-1998,2003 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * Copyright 2013 David Hoeppner.  All rights reserved.
 */

#include <sys/types.h>
#include <sys/t_lock.h>
#include <sys/klwp.h>
#include <sys/proc.h>
#include <sys/ucontext.h>
#include <sys/procfs.h>
#include <sys/privregs.h>
#include <sys/fp.h>
#include <sys/cpuvar.h>
#include <sys/cmn_err.h>
#include <sys/disp.h>
#include <sys/systm.h>
#include <sys/archsystm.h>
#include <sys/note.h>

/*
 * Clear the struct ucontext extra register state pointer.
 */
void
xregs_clrptr(klwp_id_t lwp, ucontext_t *uc)
{
        uc->uc_xrs.xrs_id = 0;
        uc->uc_xrs.xrs_ptr = NULL;
}

/*
 * Indicate whether or not an extra register state
 * pointer is associated with a struct ucontext.
 */
int
xregs_hasptr(klwp_id_t lwp, ucontext_t *uc)
```



```
{
        _NOTE(ARGUNUSED(lwp));

        return (uc->uc_xrs.xrs_id == XRS_ID);
}

/*
 * Get the struct ucontext extra register state pointer field.
 */
caddr_t
xregs_getptr(klwp_id_t lwp, ucontext_t *uc)
{
        _NOTE(ARGUNUSED(lwp));

        if (uc->uc_xrs.xrs_id == XRS_ID)
                return (uc->uc_xrs.xrs_ptr);

        return (NULL);
}

/*
 * Set the struct ucontext extra register state pointer field.
 */
void
xregs_setptr(klwp_id_t lwp, ucontext_t *uc, caddr_t xrp)
{
        _NOTE(ARGUNUSED(lwp));

        uc->uc_xrs.xrs_id = XRS_ID;
        uc->uc_xrs.xrs_ptr = xrp;
}

#if defined(_SYSCALL32_IMPL)

void
xregs_clrptr32(klwp_id_t lwp, ucontext32_t *uc)
{
        _NOTE(ARGUNUSED(lwp));

        uc->uc_xrs.xrs_id = 0;
        uc->uc_xrs.xrs_ptr = NULL;
}

int
xregs_hasptr32(klwp_id_t lwp, ucontext32_t *uc)
{
        _NOTE(ARGUNUSED(lwp));

        return (uc->uc_xrs.xrs_id == XRS_ID);
}

caddr32_t
xregs_getptr32(klwp_id_t lwp, ucontext32_t *uc)
{
        _NOTE(ARGUNUSED(lwp));

        if (uc->uc_xrs.xrs_id == XRS_ID)
                return (uc->uc_xrs.xrs_ptr);

        return (0);
}

void
xregs_setptr32(klwp_id_t lwp, ucontext32_t *uc, caddr32_t xrp)
{
        _NOTE(ARGUNUSED(lwp));
```

```
129         uc->uc_xrs.xrs_id = XRS_ID;
130         uc->uc_xrs.xrs_ptr = xrp;
131 }

133 #endif  /* _SYSCALL32_IMPL */

135 /*
136  * Fill in the extra register state area specified with the
137  * specified lwp's floating point extra register state information.
138  */
139 void
140 xregs_getfpregs(klwp_id_t lwp, caddr_t xrp)
141 {
142         prxregset_t *xregs = (prxregset_t *)xrp;
143         kfpu_t *fp = lwptofpu(lwp);

145         if (xregs == NULL)
146                 return;

148         kpreempt_disable();

150         xregs->pr_type = XR_TYPE_XSAVE;

152         kpreempt_enable();
153 }

155 /*
156  * Fill in the extra register state area specified with
157  * the specified kwp's extra register state information.
158  */
159 void
160 xregs_get(klwp_id_t lwp, caddr_t xrp)
161 {

163         if (xrp != NULL) {
164                 bzero(xrp, sizeof (prxregset_t));
165                 xregs_getfpregs(lwp, xrp);
166         }
167 }

169 /*
170  * Set the specified lwp's floating-point extra
171  * register state based on the specified input.
172  */
173 void
174 xregs_setfpregs(klwp_id_t lwp, caddr_t xrp)
175 {
176         prxregset_t *xregs = (prxregset_t *)xrp;
177         kfpu_t *fp = lwptofpu(lwp);
178         fpu_ctx_t *fpu = &lwp->lwp_pcb.pcb_fpu;

180         if (xregs == NULL)
181                 return;

183 #if defined(DEBUG)
184         if (xregs->pr_type != XR_TYPE_XSAVE) {
185                 cmn_err(CE_WARN,
186                     "xregs_setfpregs: pr_type is %d and should be %d",
187                     xregs->pr_type, XR_TYPE_XSAVE);
188         }
189 #endif  /* DEBUG */

191         if (fpu->fpu_flags & FPU_EN) {
192                 kpreempt_disable();
193                 (void) kcopy(&xregs->pr_un.pr_xsave.pr_ymm,
```



```
194                     &fp->kfpu_u.kfpu_xs.xs_ymm,
195                     sizeof (&xregs->pr_un.pr_xsave.pr_ymm));

197                 /*
198                  * If not the current lwp then resume() will handle it.
199                  */
200                 if (lwp != ttolwp(curthread)) {
201                         /* Force resume to reload fp regs */
202                         kpreempt_enable();
203                         return;
204                 }

206                 if (fpu_exists) {
207                 }

209                 kpreempt_enable();
210         }
211 }

213 /*
214  * Set the specified lwp's extra register
215  * state based on the specified input.
216  */
217 void
218 xregs_set(klwp_id_t lwp, caddr_t xrp)
219 {
220         if (xrp != NULL) {
221                 xregs_setfpregs(lwp, xrp);
222         }
223 }

225 /*
226  * Return the size of the extra register state.
227  */
228 int
229 xregs_getsize(proc_t *p)
230 {
231         return (sizeof (prxregset_t));
232 }
233 #endif /* ! codereview */
```

```
**********************************************************
    5772 Wed Jan 23 13:19:09 2013
new/usr/src/uts/intel/ia32/sys/privregs.h
XXX AVX procfs
**********************************************************
```

```
/*
 * 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.
 */

#ifndef _IA32_SYS_PRIVREGS_H
#define _IA32_SYS_PRIVREGS_H

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

#ifdef __cplusplus
extern "C" {
#endif

/*
 * This file describes the cpu's privileged register set, and
 * how the machine state is saved on the stack when a trap occurs.
 */

#if !defined(__i386)
#error  "non-i386 code depends on i386 privileged header!"
#endif

#ifndef _ASM

/*
 * This is NOT the structure to use for general purpose debugging;
 * see /proc for that.  This is NOT the structure to use to decode
 * the ucontext or grovel about in a core file; see <sys/regset.h>.
 */

struct regs {
        /*
         * Extra frame for mdb to follow through high level interrupts and
         * system traps.  Set them to 0 to terminate stacktrace.
         */
        greg_t  r_savfp;        /* a copy of %ebp */
        greg_t  r_savpc;        /* a copy of %eip */

        greg_t  r_gs;
```



```
        greg_t  r_fs;
        greg_t  r_es;
        greg_t  r_ds;
        greg_t  r_edi;
        greg_t  r_esi;
        greg_t  r_ebp;
        greg_t  r_esp;
        greg_t  r_ebx;
        greg_t  r_edx;
        greg_t  r_ecx;
        greg_t  r_eax;
        greg_t  r_trapno;
        greg_t  r_err;
        greg_t  r_eip;
        greg_t  r_cs;
        greg_t  r_efl;
        greg_t  r_uesp;
        greg_t  r_ss;
};

#define r_r0    r_eax           /* r0 for portability */
#define r_r1    r_edx           /* r1 for portability */
#define r_fp    r_ebp           /* system frame pointer */
#define r_sp    r_uesp          /* user stack pointer */
#define r_pc    r_eip           /* user's instruction pointer */
#define r_ps    r_efl           /* user's EFLAGS */

#define GREG_NUM        8       /* Number of regs between %edi and %eax */

#ifdef _KERNEL
#define lwptoregs(lwp)  ((struct regs *)((lwp)->lwp_regs))
#define lwptofpu(lwp)   ((kfpu_t *)((lwp)->lwp_fpu))
#endif /* ! codereview */
#endif /* _KERNEL */

#else   /* !_ASM */

#if defined(_MACHDEP)

#include <sys/machprivregs.h>

/*
 * Save current frame on the stack.  Uses %eax.
 */
#define __FRAME_PUSH                            \
        subl    $8, %esp;                       \
        movl    REGOFF_EIP(%esp), %eax;         \
        movl    %eax, REGOFF_SAVPC(%esp);       \
        movl    %ebp, REGOFF_SAVFP(%esp);

/*
 * Save segment registers on the stack.
 */
#define __SEGREGS_PUSH          \
        subl    $16, %esp;      \
        movw    %ds, 12(%esp);  \
        movw    %es, 8(%esp);   \
        movw    %fs, 4(%esp);   \
        movw    %gs, 0(%esp);

/*
 * Load segment register with kernel selectors.
 * %gs must be the last one to be set to make the
 * check in cmnint valid.
 */
#define __SEGREGS_LOAD_KERNEL   \
```

```
126         movw    $KDS_SEL, %cx;  \
127         movw    %cx, %ds;       \
128         movw    %cx, %es;       \
129         movw    $KFS_SEL, %cx;  \
130         movw    $KGS_SEL, %dx;  \
131         movw    %cx, %fs;       \
132         movw    %dx, %gs;

134 /*
135  * Restore segment registers off the stack.
136  *
137  * NOTE THE ORDER IS VITAL!
138  *
139  * Also note the subtle interdependency with kern_gpfault()
140  * that needs to disassemble these instructions to diagnose
141  * what happened when things (like bad segment register
142  * values) go horribly wrong.
143  */
144 #define __SEGREGS_POP           \
145         movw    0(%esp), %gs;   \
146         movw    4(%esp), %fs;   \
147         movw    8(%esp), %es;   \
148         movw    12(%esp), %ds;  \
149         addl    $16, %esp;

151 /*
152  * Macros for saving all registers necessary on interrupt entry,
153  * and restoring them on exit.
154  */
155 #define INTR_PUSH                       \
156         cld;                            \
157         pusha;                          \
158         __SEGREGS_PUSH                  \
159         __FRAME_PUSH                    \
160         cmpw    $KGS_SEL, REGOFF_GS(%esp); \
161         je      8f;                     \
162         movl    $0, REGOFF_SAVFP(%esp); \
163         __SEGREGS_LOAD_KERNEL           \
164 8:      CLEAN_CS

166 #define __INTR_POP                      \
167         popa;                           \
168         addl    $8, %esp;       /* get TRAPNO and ERR off the stack */

170 #define INTR_POP_USER                   \
171         addl    $8, %esp;       /* get extra frame off the stack */ \
172         __SEGREGS_POP                   \
173         __INTR_POP

175 #define INTR_POP_KERNEL                                 \
176         addl    $24, %esp;      /* skip extra frame and segment registers */ \
177         __INTR_POP
178 /*
179  * Macros for saving all registers necessary on system call entry,
180  * and restoring them on exit.
181  */
182 #define SYSCALL_PUSH                    \
183         cld;                            \
184         pusha;                          \
185         __SEGREGS_PUSH                  \
186         subl    $8, %esp;               \
187         pushfl;                         \
188         popl    %ecx;                   \
189         orl     $PS_IE, %ecx;           \
190         movl    %ecx, REGOFF_EFL(%esp); \
191         movl    $0, REGOFF_SAVPC(%esp); \
```



```
192         movl    $0, REGOFF_SAVFP(%esp); \
193         __SEGREGS_LOAD_KERNEL;          \

195 #define SYSENTER_PUSH                   \
196         cld;                            \
197         pusha;                          \
198         __SEGREGS_PUSH                  \
199         subl    $8, %esp;               \
200         movl    $0, REGOFF_SAVPC(%esp); \
201         movl    $0, REGOFF_SAVFP(%esp); \
202         __SEGREGS_LOAD_KERNEL

204 #define SYSCALL_POP                     \
205         INTR_POP_USER

207 #endif  /* _MACHDEP */

209 /*
210  * This is used to set eflags to known values at the head of an
211  * interrupt gate handler, i.e. interrupts are -already- disabled.
212  */
213 #define INTGATE_INIT_KERNEL_FLAGS       \
214         pushl   $F_OFF;                 \
215         popfl

217 #endif  /* !_ASM */

219 #include <sys/controlregs.h>

221 /* Control register layout for panic dump */

223 #define CREGSZ          36
224 #define CREG_GDT        0
225 #define CREG_IDT        8
226 #define CREG_LDT        16
227 #define CREG_TASKR      18
228 #define CREG_CR0        20
229 #define CREG_CR2        24
230 #define CREG_CR3        28
231 #define CREG_CR4        32

233 #if !defined(_ASM) && defined(_INT64_TYPE)

235 typedef uint64_t        creg64_t;

237 struct cregs {
238         creg64_t        cr_gdt;
239         creg64_t        cr_idt;
240         uint16_t        cr_ldt;
241         uint16_t        cr_task;
242         uint32_t        cr_cr0;
243         uint32_t        cr_cr2;
244         uint32_t        cr_cr3;
245         uint32_t        cr_cr4;
246 };

248 #if defined(_KERNEL)
249 extern void getcregs(struct cregs *);
250 #endif  /* _KERNEL */

252 #endif  /* !_ASM && _INT64_TYPE */

254 #ifdef __cplusplus
255 }
256 #endif
```

```
258 #endif  /* !_IA32_SYS_PRIVREGS_H */
```

```
**********************************************************
   10393 Wed Jan 23 13:19:09 2013
new/usr/src/uts/intel/ia32/syscall/getcontext.c
XXX AVX procfs
**********************************************************
______unchanged_portion_omitted_


int
getsetcontext(int flag, void *arg)
{
        ucontext_t uc;
        ucontext_t *ucp;
        klwp_t *lwp = ttolwp(curthread);
        stack_t dummy_stk;
        caddr_t xregs = NULL;
        int xregs_size = 0;
#endif /* ! codereview */

        /*
         * In future releases, when the ucontext structure grows,
         * getcontext should be modified to only return the fields
         * specified in the uc_flags.  That way, the structure can grow
         * and still be binary compatible will all .o's which will only
         * have old fields defined in uc_flags
         */

        switch (flag) {
        default:
                return (set_errno(EINVAL));

        case GETCONTEXT:
                schedctl_finish_sigblock(curthread);
                savecontext(&uc, &curthread->t_hold);
                if (uc.uc_flags & UC_SIGMASK)
                        SIGSET_NATIVE_TO_BRAND(&uc.uc_sigmask);
                if (copyout(&uc, arg, sizeof (uc)))
                        return (set_errno(EFAULT));
                return (0);

        case SETCONTEXT:
                ucp = arg;
                if (ucp == NULL)
                        exit(CLD_EXITED, 0);
                /*
                 * Don't copyin filler or floating state unless we need it.
                 * The ucontext_t struct and fields are specified in the ABI.
                 */
                if (copyin(ucp, &uc, sizeof (ucontext_t) -
                    sizeof (uc.uc_filler) -
                    sizeof (uc.uc_mcontext.fpregs))) {
                        return (set_errno(EFAULT));
                }
                if (uc.uc_flags & UC_SIGMASK)
                        SIGSET_BRAND_TO_NATIVE(&uc.uc_sigmask);

                if ((uc.uc_flags & UC_FPU) &&
                    copyin(&ucp->uc_mcontext.fpregs, &uc.uc_mcontext.fpregs,
                    sizeof (uc.uc_mcontext.fpregs))) {
                        return (set_errno(EFAULT));
                }

                /*
                 * Get extra register state.
                 */
                xregs_clrptr(lwp, &uc);
```



```
#endif /* ! codereview */
                restorecontext(&uc);

                if ((uc.uc_flags & UC_STACK) && (lwp->lwp_ustack != 0))
                        (void) copyout(&uc.uc_stack, (stack_t *)lwp->lwp_ustack,
                            sizeof (uc.uc_stack));

                /*
                 * Free extra register state.
                 */
                if (xregs_size)
                        kmem_free(xregs, xregs_size);

#endif /* ! codereview */
                return (0);

        case GETUSTACK:
                if (copyout(&lwp->lwp_ustack, arg, sizeof (caddr_t)))
                        return (set_errno(EFAULT));
                return (0);

        case SETUSTACK:
                if (copyin(arg, &dummy_stk, sizeof (dummy_stk)))
                        return (set_errno(EFAULT));
                lwp->lwp_ustack = (uintptr_t)arg;
                return (0);
        }
}

#ifdef _SYSCALL32_IMPL

/*
 * Save user context for 32-bit processes.
 */
void
savecontext32(ucontext32_t *ucp, const k_sigset_t *mask)
{
        proc_t *p = ttoproc(curthread);
        klwp_t *lwp = ttolwp(curthread);
        struct regs *rp = lwptoregs(lwp);

        bzero(&ucp->uc_mcontext.fpregs, sizeof (ucontext32_t) -
            offsetof(ucontext32_t, uc_mcontext.fpregs));

        ucp->uc_flags = UC_ALL;
        ucp->uc_link = (caddr32_t)lwp->lwp_oldcontext;

        if (lwp->lwp_ustack == NULL ||
            copyin((void *)lwp->lwp_ustack, &ucp->uc_stack,
            sizeof (ucp->uc_stack)) != 0 ||
            ucp->uc_stack.ss_size == 0) {

                if (lwp->lwp_sigaltstack.ss_flags == SS_ONSTACK) {
                        ucp->uc_stack.ss_sp =
                            (caddr32_t)(uintptr_t)lwp->lwp_sigaltstack.ss_sp;
                        ucp->uc_stack.ss_size =
                            (size32_t)lwp->lwp_sigaltstack.ss_size;
                        ucp->uc_stack.ss_flags = SS_ONSTACK;
                } else {
                        ucp->uc_stack.ss_sp = (caddr32_t)(uintptr_t)
                            (p->p_usrstack - p->p_stksize);
                        ucp->uc_stack.ss_size = (size32_t)p->p_stksize;
                        ucp->uc_stack.ss_flags = 0;
                }
        }
```

```
305         /*
306          * If either the trace flag or REQUEST_STEP is set, arrange
307          * for single-stepping and turn off the trace flag.
308          */
309         if ((rp->r_ps & PS_T) || (lwp->lwp_pcb.pcb_flags & REQUEST_STEP)) {
310                 /*
311                  * Clear PS_T so that saved user context won't have trace
312                  * flag set.
313                  */
314                 rp->r_ps &= ~PS_T;

316                 if (!(lwp->lwp_pcb.pcb_flags & REQUEST_NOSTEP)) {
317                         lwp->lwp_pcb.pcb_flags |= DEBUG_PENDING;
318                         /*
319                          * See comments in savecontext().
320                          */
321                         aston(curthread);
322                 }
323         }

325         getgregs32(lwp, ucp->uc_mcontext.gregs);
326         if (lwp->lwp_pcb.pcb_fpu.fpu_flags & FPU_EN)
327                 getfpregs32(lwp, &ucp->uc_mcontext.fpregs);
328         else
329                 ucp->uc_flags &= ~UC_FPU;

331         sigktou(mask, &ucp->uc_sigmask);
332 }

334 int
335 getsetcontext32(int flag, void *arg)
336 {
337         ucontext32_t uc;
338         ucontext_t ucnat;
339         ucontext32_t *ucp;
340         klwp_t *lwp = ttolwp(curthread);
341         caddr32_t ustack32;
342         stack32_t dummy_stk32;

344         switch (flag) {
345         default:
346                 return (set_errno(EINVAL));

348         case GETCONTEXT:
349                 schedctl_finish_sigblock(curthread);
350                 savecontext32(&uc, &curthread->t_hold);
351                 if (uc.uc_flags & UC_SIGMASK)
352                         SIGSET_NATIVE_TO_BRAND(&uc.uc_sigmask);
353                 if (copyout(&uc, arg, sizeof (uc)))
354                         return (set_errno(EFAULT));
355                 return (0);

357         case SETCONTEXT:
358                 ucp = arg;
359                 if (ucp == NULL)
360                         exit(CLD_EXITED, 0);
361                 if (copyin(ucp, &uc, sizeof (uc) -
362                     sizeof (uc.uc_filler) -
363                     sizeof (uc.uc_mcontext.fpregs))) {
364                         return (set_errno(EFAULT));
365                 }
366                 if (uc.uc_flags & UC_SIGMASK)
367                         SIGSET_BRAND_TO_NATIVE(&uc.uc_sigmask);
368                 if ((uc.uc_flags & UC_FPU) &&
369                     copyin(&ucp->uc_mcontext.fpregs, &uc.uc_mcontext.fpregs,
```



```
370                     sizeof (uc.uc_mcontext.fpregs))) {
371                         return (set_errno(EFAULT));
372                 }

374                 ucontext_32ton(&uc, &ucnat);
375                 restorecontext(&ucnat);

377                 if ((uc.uc_flags & UC_STACK) && (lwp->lwp_ustack != 0))
378                         (void) copyout(&uc.uc_stack,
379                             (stack32_t *)lwp->lwp_ustack, sizeof (uc.uc_stack));
380                 return (0);

382         case GETUSTACK:
383                 ustack32 = (caddr32_t)lwp->lwp_ustack;
384                 if (copyout(&ustack32, arg, sizeof (ustack32)))
385                         return (set_errno(EFAULT));
386                 return (0);

388         case SETUSTACK:
389                 if (copyin(arg, &dummy_stk32, sizeof (dummy_stk32)))
390                         return (set_errno(EFAULT));
391                 lwp->lwp_ustack = (uintptr_t)arg;
392                 return (0);
393         }
394 }

396 #endif  /* _SYSCALL32_IMPL */
```

```
**********************************************************
    6429 Wed Jan 23 13:19:10 2013
new/usr/src/uts/intel/sys/archsystm.h
XXX AVX procfs
**********************************************************
```

```
/*
 * 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) 1993, 2010, Oracle and/or its affiliates. All rights reserved.
 */

#ifndef _SYS_ARCHSYSTM_H
#define _SYS_ARCHSYSTM_H

/*
 * A selection of ISA-dependent interfaces
 */

#include <vm/seg_enum.h>
#include <vm/page.h>

#ifdef __cplusplus
extern "C" {
#endif

#ifdef _KERNEL

extern greg_t getfp(void);
extern int getpil(void);

extern ulong_t getcr0(void);
extern void setcr0(ulong_t);
extern ulong_t getcr8(void);
extern void setcr8(ulong_t);
extern ulong_t getcr2(void);
extern void clflush_insn(caddr_t addr);
extern void mfence_insn(void);

#if defined(__i386)
extern uint16_t getgs(void);
extern void setgs(uint16_t);

extern void patch_sse(void);
extern void patch_sse2(void);
#endif

extern void patch_xsave(void);
```



```
extern void cli(void);
extern void sti(void);

extern void tenmicrosec(void);

extern void restore_int_flag(ulong_t);
extern void intr_restore(ulong_t);
extern ulong_t clear_int_flag(void);
extern ulong_t intr_clear(void);
extern ulong_t getflags(void);
extern int interrupts_enabled(void);

extern void int3(void);
extern void int18(void);
extern void int20(void);
extern void int_cmci(void);

#if defined(__amd64)
extern void sys_syscall();
extern void sys_syscall32();
extern void sys_lcall32();
extern void sys_syscall_int();
extern void brand_sys_syscall();
extern void brand_sys_syscall32();
extern void brand_sys_syscall_int();
extern int update_sregs();
extern void reset_sregs();
#elif defined(__i386)
extern void sys_call();
extern void brand_sys_call();
#endif
extern void sys_sysenter();
extern void _sys_sysenter_post_swapgs();
extern void brand_sys_sysenter();
extern void _brand_sys_sysenter_post_swapgs();

extern void dosyscall(void);

extern void bind_hwcap(void);

extern uint16_t inw(int port);
extern uint32_t inl(int port);
extern void outw(int port, uint16_t value);
extern void outl(int port, uint32_t value);

extern void pc_reset(void) __NORETURN;
extern void efi_reset(void) __NORETURN;
extern void reset(void) __NORETURN;
extern int goany(void);

extern void setgregs(klwp_t *, gregset_t);
extern void getgregs(klwp_t *, gregset_t);
extern void setfpregs(klwp_t *, fpregset_t *);
extern void getfpregs(klwp_t *, fpregset_t *);

#if defined(_SYSCALL32_IMPL)
extern void getgregs32(klwp_t *, gregset32_t);
extern void setfpregs32(klwp_t *, fpregset32_t *);
extern void getfpregs32(klwp_t *, fpregset32_t *);
#endif

struct ucontext;
extern void xregs_clrptr(struct _klwp *, struct ucontext *);
extern int xregs_hasptr(struct _klwp *, struct ucontext *);
extern caddr_t xregs_getptr(struct _klwp *, struct ucontext *);
```

```
128 extern void xregs_setptr(struct _klwp *, struct ucontext *, caddr_t);

130 #if defined(_SYSCALL32_IMPL)
131 struct ucontext32;
132 extern void xregs_clrptr32(struct _klwp *, struct ucontext32 *);
133 extern int xregs_hasptr32(struct _klwp *, struct ucontext32 *);
134 extern caddr32_t xregs_getptr32(struct _klwp *, struct ucontext32 *);
135 extern void xregs_setptr32(struct _klwp *, struct ucontext32 *, caddr32_t);
136 #endif /* _SYSCALL32_IMPL */

138 extern void xregs_get(struct _klwp *, caddr_t);
139 extern void xregs_set(struct _klwp *, caddr_t);
140 extern int xregs_getsize(struct proc *);

142 #endif /* ! codereview */
143 struct fpu_ctx;

145 extern void fp_free(struct fpu_ctx *, int);
146 extern void fp_save(struct fpu_ctx *);
147 extern void fp_restore(struct fpu_ctx *);

149 extern int fpu_pentium_fdivbug;

151 extern void sep_save(void *);
152 extern void sep_restore(void *);

154 extern void brand_interpositioning_enable(void);
155 extern void brand_interpositioning_disable(void);

157 struct regs;

159 extern int instr_size(struct regs *, caddr_t *, enum seg_rw);

161 extern int enable_cbcp; /* patchable in /etc/system */

163 extern uint_t cpu_hwcap_flags;
164 extern uint_t cpu_freq;
165 extern uint64_t cpu_freq_hz;

167 extern int use_sse_pagecopy;
168 extern int use_sse_pagezero;
169 extern int use_sse_copy;

171 extern caddr_t i86devmap(pfn_t, pgcnt_t, uint_t);
172 extern page_t *page_numtopp_alloc(pfn_t pfnum);

174 extern void hwblkclr(void *, size_t);
175 extern void hwblkpagecopy(const void *, void *);
176 extern void page_copy_no_xmm(void *dst, void *src);
177 extern void block_zero_no_xmm(void *dst, int len);
178 #define BLOCKZEROALIGN (4 * sizeof (void *))

180 extern void (*kcpc_hw_enable_cpc_intr)(void);

182 extern void init_desctbls(void);

184 extern user_desc_t *cpu_get_gdt(void);

186 extern void switch_sp_and_call(void *, void (*)(uint_t, uint_t), uint_t,
187         uint_t);
188 extern hrtime_t (*gethrtimef)(void);
189 extern hrtime_t (*gethrtimeunscaledf)(void);
190 extern void (*scalehrtimef)(hrtime_t *);
191 extern uint64_t (*unscalehrtimef)(hrtime_t);
192 extern void (*gethrestimef)(timestruc_t *);
```



```
194 extern void av_dispatch_softvect(uint_t);
195 extern void av_dispatch_autovect(uint_t);
196 extern uint_t atomic_btr32(uint32_t *, uint_t);
197 extern uint_t bsrw_insn(uint16_t);
198 extern int sys_rtt_common(struct regs *);
199 extern void fakesoftint(void);

201 extern void *plat_traceback(void *);

203 #if defined(__xpv)
204 extern void xen_init_callbacks(void);
205 extern void xen_set_callback(void (*)(void), uint_t, uint_t);
206 extern void xen_printf(const char *, ...);
207 #define cpr_dprintf xen_printf
208 extern int xpv_panicking;
209 #define IN_XPV_PANIC() (xpv_panicking > 0)
210 #else
211 extern void setup_mca(void);
212 extern void pat_sync(void);
213 extern void patch_tsc_read(int);
214 #if defined(__amd64) && !defined(__xpv)
215 extern void patch_memops(uint_t);
216 #endif  /* defined(__amd64) && !defined(__xpv) */
217 extern void setup_xfem(void);
218 #define cpr_dprintf prom_printf
219 #define IN_XPV_PANIC() (__lintzero)
220 #endif

222 #endif /* _KERNEL */

224 #if defined(_KERNEL) || defined(_BOOT)
225 extern uint8_t inb(int port);
226 extern void outb(int port, uint8_t value);
227 #endif

229 #ifdef __cplusplus
230 }
231 #endif

233 #endif  /* _SYS_ARCHSYSTM_H */
```

```
**********************************************************
    3979 Wed Jan 23 13:19:10 2013
new/usr/src/uts/intel/sys/procfs_isa.h
XXX AVX procfs
**********************************************************
```

```
/*
 * 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 2005 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#ifndef _SYS_PROCFS_ISA_H
#define _SYS_PROCFS_ISA_H

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

/*
 * Instruction Set Architecture specific component of <sys/procfs.h>
 * i386 version
 */

#include <sys/regset.h>

#ifdef  __cplusplus
extern "C" {
#endif

/*
 * Possible values of pr_dmodel.
 * This isn't isa-specific, but it needs to be defined here for other reasons.
 */
#define PR_MODEL_UNKNOWN 0
#define PR_MODEL_ILP32  1       /* process data model is ILP32 */
#define PR_MODEL_LP64   2       /* process data model is LP64 */

/*
 * To determine whether application is running native.
 */
#if defined(_LP64)
#define PR_MODEL_NATIVE PR_MODEL_LP64
#elif defined(_ILP32)
#define PR_MODEL_NATIVE PR_MODEL_ILP32
#else
#error "No DATAMODEL_NATIVE specified"
#endif  /* _LP64 || _ILP32 */
```



```
#if defined(__i386) || defined(__amd64)
/*
 * Holds one i386 or amd64 instruction
 */
typedef uchar_t instr_t;
#endif

#define NPRGREG         _NGREG
#define prgreg_t        greg_t
#define prgregset_t     gregset_t
#define prfpregset      fpu
#define prfpregset_t    fpregset_t

#if defined(_SYSCALL32)
/*
 * kernel view of the ia32 register set
 */
typedef uchar_t         instr32_t;
#if defined(__amd64)
#define NPRGREG32       _NGREG32
#define prgreg32_t      greg32_t
#define prgregset32_t   gregset32_t
#define prfpregset32    fpu32
#define prfpregset32_t  fpregset32_t
#else
#define NPRGREG32       _NGREG
#define prgreg32_t      greg_t
#define prgregset32_t   gregset_t
#define prfpregset32    fpu
#define prfpregset32_t  fpregset_t
#endif
#endif  /* _SYSCALL32 */

#if defined(__amd64)
/*
 * The following defines are for portability (see <sys/regset.h>).
 */
#define R_PC    REG_RIP
#define R_PS    REG_RFL
#define R_SP    REG_RSP
#define R_FP    REG_RBP
#define R_R0    REG_RAX
#define R_R1    REG_RDX
#elif defined(__i386)
/*
 * The following defines are for portability (see <sys/regset.h>).
 */
#define R_PC    EIP
#define R_PS    EFL
#define R_SP    UESP
#define R_FP    EBP
#define R_R0    EAX
#define R_R1    EDX
#endif

#define XR_TYPE_XSAVE   0x101

typedef struct prxregset {
        uint32_t        pr_type;
        uint32_t        pr_align;
        uint32_t        pr_xsize;
        uint32_t        pr_pad;
        union {
                struct pr_xsave {
                        uint16_t        pr_fcw;
                        uint16_t        pr_fsw;
```

```
126                         uint16_t        pr_fctw;
127                         uint16_t        pr_fop;
128 #if defined(__amd64)
129                         uint64_t        pr_rip;
130                         uint64_t        pr_rdp;
131 #else
132                         uint32_t        pr_eip;
133                         uint16_t        pr_cs;
134                         uint16_t        __pr_ign0;
135                         uint32_t        pr_dp;
136                         uint16_t        pr_ds;
137                         uint16_t        __pr_ign1;
138 #endif
139                         uint32_t        pr_mxcsr;
140                         uint32_t        pr_mxcsr_mask;
141                         union {
142                                 uint16_t        pr_fpr_16[5];
143                                 u_longlong_t    pr_fpr_mmx;
144                                 uint32_t        __pr_fpr_pad[4];
145                         } pr_st[8];
146 #if defined(__amd64)
147                         upad128_t       pr_xmm[16];
148                         upad128_t       __pr_ign2[3];
149 #else
150                         upad128_t       pr_xmm[8];
151                         upad128_t       __pr_ign2[11];
152 #endif
153                         union {
154                                 struct {
155                                         uint64_t        pr_xcr0;
156                                         uint64_t        pr_mbz[2];
157                                 } pr_xsave_info;
158                                 upad128_t       __pr_pad[3];
159                         } pr_sw_avail;
160                         uint64_t        pr_xstate_bv;
161                         uint64_t        pr_rsv_mbz[2];
162                         uint64_t        pr_reserved[5];
163 #if defined(__amd64)
164                         upad128_t       pr_ymm[16];
165 #else
166                         upad128_t       pr_ymm[8];
167                         upad128_t       __pr_ign3[8];
168 #endif
169                 } pr_xsave;
170         } pr_un;
171 } prxregset_t;

173 #endif /* ! codereview */
174 #ifdef  __cplusplus
175 }
176 #endif

178 #endif  /* _SYS_PROCFS_ISA_H */
```

```
**********************************************************
    3907 Wed Jan 23 13:19:11 2013
new/usr/src/uts/intel/sys/ucontext.h
XXX AVX procfs
**********************************************************
______unchanged_portion_omitted_
```

```
 66 #endif /* _STACK_T */
 67 #endif /* defined(_XPG4_2) && !defined(__EXTENSIONS__) */

 69 #if !defined(_XPG4_2) || defined(__EXTENSIONS__)
 70 typedef struct ucontext ucontext_t;
 71 #else
 72 typedef struct __ucontext ucontext_t;
 73 #endif /* !defined(_XPG4_2) || defined(__EXTENSIONS__) */

 75 #define XRS_ID  0x00737278      /* the string "xrs" */

 77 typedef struct {
 78         unsigned long   xrs_id;
 79         caddr_t         xrs_ptr;
 80 } xrs_t;

 82 #endif /* ! codereview */
 83 #if !defined(_XPG4_2) || defined(__EXTENSIONS__)
 84 struct  ucontext {
 85 #else
 86 struct  __ucontext {
 87 #endif
 88         unsigned long   uc_flags;
 89         ucontext_t      *uc_link;
 90         sigset_t        uc_sigmask;
 91         stack_t         uc_stack;
 92         mcontext_t      uc_mcontext;
 93         xrs_t           uc_xrs;
 94         long            uc_filler[3];   /* see ABI spec for Intel386 */
 75         long            uc_filler[5];   /* see ABI spec for Intel386 */
 95 };

 97 #if defined(_SYSCALL32)

 99 typedef struct {
100         uint32_t        xrs_id;
101         caddr32_t       xrs_ptr;
102 } xrs32_t;

104 #endif /* ! codereview */
105 /* Kernel view of user ILP32 ucontext structure */

107 typedef struct ucontext32 {
108         uint32_t        uc_flags;
109         caddr32_t       uc_link;
110         sigset_t        uc_sigmask;
111         stack32_t       uc_stack;
112         mcontext32_t    uc_mcontext;
113         xrs32_t         uc_xrs;
114         int32_t         uc_filler[3];
 80         int32_t         uc_filler[5];
115 } ucontext32_t;

117 #if defined(_KERNEL)
118 extern void ucontext_nto32(const ucontext_t *src, ucontext32_t *dest);
119 extern void ucontext_32ton(const ucontext32_t *src, ucontext_t *dest);
120 #endif

122 #endif  /* _SYSCALL32 */
```



```
124 #if !defined(_XPG4_2) || defined(__EXTENSIONS__)
125 #define GETCONTEXT      0
126 #define SETCONTEXT      1
127 #define GETUSTACK       2
128 #define SETUSTACK       3

130 /*
131  * values for uc_flags
132  * these are implementation dependent flags, that should be hidden
133  * from the user interface, defining which elements of ucontext
134  * are valid, and should be restored on call to setcontext
135  */

137 #define UC_SIGMASK      0x01
138 #define UC_STACK        0x02
139 #define UC_CPU          0x04
140 #define UC_MAU          0x08
141 #define UC_XREGS        0x10
142 #endif /* ! codereview */
143 #define UC_FPU          UC_MAU

145 #define UC_MCONTEXT     (UC_CPU|UC_FPU)

147 /*
148  * UC_ALL specifies the default context
149  */

151 #define UC_ALL          (UC_SIGMASK|UC_STACK|UC_MCONTEXT)
152 #endif /* !defined(_XPG4_2) || defined(__EXTENSIONS__) */

154 #ifdef _KERNEL
155 void savecontext(ucontext_t *, const k_sigset_t *);
156 void restorecontext(ucontext_t *);

158 #ifdef _SYSCALL32
159 extern void savecontext32(ucontext32_t *, const k_sigset_t *);
160 #endif
161 #endif

163 #ifdef  __cplusplus
164 }
165 #endif

167 #endif /* _SYS_UCONTEXT_H */
```