```
**********************************************************
   74002 Sat Jan 10 12:31:08 2015
new/usr/src/lib/libproc/common/Pcore.c
5383 5234 breaks build on sparc
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */
/*
 * Copyright 2012 DEY Storage Systems, Inc.  All rights reserved.
 * Copyright (c) 2014, Joyent, Inc. All rights reserved.
 * Copyright (c) 2013 by Delphix. All rights reserved.
 * Copyright 2015 Gary Mills
 */

#include <sys/types.h>
#include <sys/utsname.h>
#include <sys/sysmacros.h>
#include <sys/proc.h>

#include <alloca.h>
#include <rtld_db.h>
#include <libgen.h>
#include <limits.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <gelf.h>
#include <stddef.h>
#include <signal.h>

#include "libproc.h"
#include "Pcontrol.h"
#include "P32ton.h"
#include "Putil.h"
#ifdef __x86
#include "Pcore_linux.h"
#endif

/*
 * Pcore.c - Code to initialize a ps_prochandle from a core dump.  We
 * allocate an additional structure to hold information from the core
 * file, and attach this to the standard ps_prochandle in place of the
 * ability to examine /proc/<pid>/ files.
```



```
 */

/*
 * Basic i/o function for reading and writing from the process address space
 * stored in the core file and associated shared libraries.  We compute the
 * appropriate fd and offsets, and let the provided prw function do the rest.
 */
static ssize_t
core_rw(struct ps_prochandle *P, void *buf, size_t n, uintptr_t addr,
    ssize_t (*prw)(int, void *, size_t, off64_t))
{
        ssize_t resid = n;

        while (resid != 0) {
                map_info_t *mp = Paddr2mptr(P, addr);

                uintptr_t mapoff;
                ssize_t len;
                off64_t off;
                int fd;

                if (mp == NULL)
                        break;  /* No mapping for this address */

                if (mp->map_pmap.pr_mflags & MA_RESERVED1) {
                        if (mp->map_file == NULL || mp->map_file->file_fd < 0)
                                break;  /* No file or file not open */

                        fd = mp->map_file->file_fd;
                } else
                        fd = P->asfd;

                mapoff = addr - mp->map_pmap.pr_vaddr;
                len = MIN(resid, mp->map_pmap.pr_size - mapoff);
                off = mp->map_offset + mapoff;

                if ((len = prw(fd, buf, len, off)) <= 0)
                        break;

                resid -= len;
                addr += len;
                buf = (char *)buf + len;
        }

        /*
         * Important: Be consistent with the behavior of i/o on the as file:
         * writing to an invalid address yields EIO; reading from an invalid
         * address falls through to returning success and zero bytes.
         */
        if (resid == n && n != 0 && prw != pread64) {
                errno = EIO;
                return (-1);
        }

        return (n - resid);
}
______unchanged_portion_omitted_

/*ARGSUSED*/
static void
Pfini_core(struct ps_prochandle *P, void *data)
{
        core_info_t *core = data;

        if (core != NULL) {
                extern void __priv_free_info(void *);
```

```
195                 lwp_info_t *nlwp, *lwp = list_next(&core->core_lwp_head);
196                 int i;

198                 for (i = 0; i < core->core_nlwp; i++, lwp = nlwp) {
199                         nlwp = list_next(lwp);
200 #ifdef __sparc
201                         if (lwp->lwp_gwins != NULL)
202                                 free(lwp->lwp_gwins);
203                         if (lwp->lwp_xregs != NULL)
204                                 free(lwp->lwp_xregs);
205                         if (lwp->lwp_asrs != NULL)
206                                 free(lwp->lwp_asrs);
207 #endif
208                         free(lwp);
209                 }

211                 if (core->core_platform != NULL)
212                         free(core->core_platform);
213                 if (core->core_uts != NULL)
214                         free(core->core_uts);
215                 if (core->core_cred != NULL)
216                         free(core->core_cred);
217                 if (core->core_priv != NULL)
218                         free(core->core_priv);
219                 if (core->core_privinfo != NULL)
220                         __priv_free_info(core->core_privinfo);
221                 if (core->core_ppii != NULL)
222                         free(core->core_ppii);
223                 if (core->core_zonename != NULL)
224                         free(core->core_zonename);
225 #ifdef __x86
222 #if defined(__i386) || defined(__amd64)
226                 if (core->core_ldt != NULL)
227                         free(core->core_ldt);
228 #endif

230                 free(core);
231         }
232 }
```

_____*unchanged_portion_omitted_*

```
277 #ifdef __x86
274 #if defined(__i386) || defined(__amd64)
278 /*ARGSUSED*/
279 static int
280 Pldt_core(struct ps_prochandle *P, struct ssd *pldt, int nldt, void *data)
281 {
282         core_info_t *core = data;

284         if (pldt == NULL || nldt == 0)
285                 return (core->core_nldt);

287         if (core->core_ldt != NULL) {
288                 nldt = MIN(nldt, core->core_nldt);

290                 (void) memcpy(pldt, core->core_ldt,
291                     nldt * sizeof (struct ssd));

293                 return (nldt);
294         }

296         errno = ENODATA;
297         return (-1);
298 }
299 #endif
```



```
301 static const ps_ops_t P_core_ops = {
302         .pop_pread      = Pread_core,
303         .pop_pwrite     = Pwrite_core,
304         .pop_cred       = Pcred_core,
305         .pop_priv       = Ppriv_core,
306         .pop_psinfo     = Ppsinfo_core,
307         .pop_fini       = Pfini_core,
308         .pop_platform   = Pplatform_core,
309         .pop_uname      = Puname_core,
310         .pop_zonename   = Pzonename_core,
311 #ifdef __x86
308 #if defined(__i386) || defined(__amd64)
312         .pop_ldt        = Pldt_core
313 #endif
314 };
```

_____*unchanged_portion_omitted_*

```
437 #ifdef __x86

439 static void
440 lx_prpsinfo32_to_psinfo(lx_prpsinfo32_t *p32, psinfo_t *psinfo)
441 {
442         psinfo->pr_flag = p32->pr_flag;
443         psinfo->pr_pid = p32->pr_pid;
444         psinfo->pr_ppid = p32->pr_ppid;
445         psinfo->pr_uid = p32->pr_uid;
446         psinfo->pr_gid = p32->pr_gid;
447         psinfo->pr_sid = p32->pr_sid;
448         psinfo->pr_pgid = p32->pr_pgrp;

450         (void) memcpy(psinfo->pr_fname, p32->pr_fname,
451             sizeof (psinfo->pr_fname));
452         (void) memcpy(psinfo->pr_psargs, p32->pr_psargs,
453             sizeof (psinfo->pr_psargs));
454 }
```

_____*unchanged_portion_omitted_*

```
637 #endif /* __x86 */

639 static int
640 note_psinfo(struct ps_prochandle *P, size_t nbytes)
641 {
642 #ifdef _LP64
643         core_info_t *core = P->data;

645         if (core->core_dmodel == PR_MODEL_ILP32) {
646                 psinfo32_t ps32;

648                 if (nbytes < sizeof (psinfo32_t) ||
649                     read(P->asfd, &ps32, sizeof (ps32)) != sizeof (ps32))
650                         goto err;

652                 psinfo_32_to_n(&ps32, &P->psinfo);
653         } else
654 #endif
655         if (nbytes < sizeof (psinfo_t) ||
656             read(P->asfd, &P->psinfo, sizeof (psinfo_t)) != sizeof (psinfo_t))
657                 goto err;

659         dprintf("pr_fname = <%s>\n", P->psinfo.pr_fname);
660         dprintf("pr_psargs = <%s>\n", P->psinfo.pr_psargs);
661         dprintf("pr_wstat = 0x%x\n", P->psinfo.pr_wstat);

663         return (0);

665 err:
```

```
 666         dprintf("Pgrab_core: failed to read NT_PSINFO\n");
 667         return (-1);
 668 }
______unchanged_portion_omitted_

 837 #ifdef __x86
 830 #if defined(__i386) || defined(__amd64)
 838 static int
 839 note_ldt(struct ps_prochandle *P, size_t nbytes)
 840 {
 841         core_info_t *core = P->data;
 842         struct ssd *pldt;
 843         uint_t nldt;

 845         if (core->core_ldt != NULL || nbytes < sizeof (struct ssd))
 846                 return (0);     /* Already seen or bad size */

 848         nldt = nbytes / sizeof (struct ssd);
 849         nbytes = nldt * sizeof (struct ssd);

 851         if ((pldt = malloc(nbytes)) == NULL)
 852                 return (-1);

 854         if (read(P->asfd, pldt, nbytes) != nbytes) {
 855                 dprintf("Pgrab_core: failed to read NT_LDT\n");
 856                 free(pldt);
 857                 return (-1);
 858         }

 860         core->core_ldt = pldt;
 861         core->core_nldt = nldt;
 862         return (0);
 863 }
______unchanged_portion_omitted_

1129 /*
1130  * Populate a table of function pointers indexed by Note type with our
1131  * functions to process each type of core file note:
1132  */
1133 static int (*nhdlrs[])(struct ps_prochandle *, size_t) = {
1134         note_notsup,            /*  0   unassigned              */
1135 #ifdef __x86
1136         note_linux_prstatus,            /*  1   NT_PRSTATUS (old)       */
1137 #else
1138         note_notsup,            /*  1   NT_PRSTATUS (old)       */
1139 #endif
1140         note_notsup,            /*  2   NT_PRFPREG (old)        */
1141 #ifdef __x86
1142         note_linux_psinfo,              /*  3   NT_PRPSINFO (old)       */
1143 #else
1144         note_notsup,            /*  3   NT_PRPSINFO (old)       */
1145 #endif
1146 #ifdef __sparc
1147         note_xreg,              /*  4   NT_PRXREG               */
1148 #else
1149         note_notsup,            /*  4   NT_PRXREG               */
1150 #endif
1151         note_platform,          /*  5   NT_PLATFORM             */
1152         note_auxv,              /*  6   NT_AUXV                 */
1153 #ifdef __sparc
1154         note_gwindows,          /*  7   NT_GWINDOWS             */
1155 #ifdef __sparcv9
1156         note_asrs,              /*  8   NT_ASRS                 */
1157 #else
1158         note_notsup,            /*  8   NT_ASRS                 */
1159 #endif
```



```
1160 #else
1161         note_notsup,            /*  7   NT_GWINDOWS             */
1162         note_notsup,            /*  8   NT_ASRS                 */
1163 #endif
1164 #ifdef __x86
1149 #if defined(__i386) || defined(__amd64)
1165         note_ldt,               /*  9   NT_LDT                  */
1166 #else
1167         note_notsup,            /*  9   NT_LDT                  */
1168 #endif
1169         note_pstatus,           /* 10   NT_PSTATUS              */
1170         note_notsup,            /* 11   unassigned              */
1171         note_notsup,            /* 12   unassigned              */
1172         note_psinfo,            /* 13   NT_PSINFO               */
1173         note_cred,              /* 14   NT_PRCRED               */
1174         note_utsname,           /* 15   NT_UTSNAME              */
1175         note_lwpstatus,         /* 16   NT_LWPSTATUS            */
1176         note_lwpsinfo,          /* 17   NT_LWPSINFO             */
1177         note_priv,              /* 18   NT_PRPRIV               */
1178         note_priv_info,         /* 19   NT_PRPRIVINFO           */
1179         note_content,           /* 20   NT_CONTENT              */
1180         note_zonename,          /* 21   NT_ZONENAME             */
1181         note_fdinfo,            /* 22   NT_FDINFO               */
1182         note_spymaster,         /* 23   NT_SPYMASTER            */
1183 };
______unchanged_portion_omitted_

2200 /*
2201  * Main engine for core file initialization: given an fd for the core file
2202  * and an optional pathname, construct the ps_prochandle.  The aout_path can
2203  * either be a suggested executable pathname, or a suggested directory to
2204  * use as a possible current working directory.
2205  */
2206 struct ps_prochandle *
2207 Pfgrab_core(int core_fd, const char *aout_path, int *perr)
2208 {
2209         struct ps_prochandle *P;
2210         core_info_t *core_info;
2211         map_info_t *stk_mp, *brk_mp;
2212         const char *execname;
2213         char *interp;
2214         int i, notes, pagesize;
2215         uintptr_t addr, base_addr;
2216         struct stat64 stbuf;
2217         void *phbuf, *php;
2218         size_t nbytes;
2219 #ifdef __x86
2220         boolean_t from_linux = B_FALSE;
2221 #endif

2223         elf_file_t aout;
2224         elf_file_t core;

2226         Elf_Scn *scn, *intp_scn = NULL;
2227         Elf_Data *dp;

2229         GElf_Phdr phdr, note_phdr;
2230         GElf_Shdr shdr;
2231         GElf_Xword nleft;

2233         if (elf_version(EV_CURRENT) == EV_NONE) {
2234                 dprintf("libproc ELF version is more recent than libelf\n");
2235                 *perr = G_ELF;
2236                 return (NULL);
2237         }
```

```
2239         aout.e_elf = NULL;
2240         aout.e_fd = -1;

2242         core.e_elf = NULL;
2243         core.e_fd = core_fd;

2245         /*
2246          * Allocate and initialize a ps_prochandle structure for the core.
2247          * There are several key pieces of initialization here:
2248          *
2249          * 1. The PS_DEAD state flag marks this prochandle as a core file.
2250          *    PS_DEAD also thus prevents all operations which require state
2251          *    to be PS_STOP from operating on this handle.
2252          *
2253          * 2. We keep the core file fd in P->asfd since the core file contains
2254          *    the remnants of the process address space.
2255          *
2256          * 3. We set the P->info_valid bit because all information about the
2257          *    core is determined by the end of this function; there is no need
2258          *    for proc_update_maps() to reload mappings at any later point.
2259          *
2260          * 4. The read/write ops vector uses our core_rw() function defined
2261          *    above to handle i/o requests.
2262          */
2263         if ((P = malloc(sizeof (struct ps_prochandle))) == NULL) {
2264                 *perr = G_STRANGE;
2265                 return (NULL);
2266         }

2268         (void) memset(P, 0, sizeof (struct ps_prochandle));
2269         (void) mutex_init(&P->proc_lock, USYNC_THREAD, NULL);
2270         P->state = PS_DEAD;
2271         P->pid = (pid_t)-1;
2272         P->asfd = core.e_fd;
2273         P->ctlfd = -1;
2274         P->statfd = -1;
2275         P->agentctlfd = -1;
2276         P->agentstatfd = -1;
2277         P->zoneroot = NULL;
2278         P->info_valid = 1;
2279         Pinit_ops(&P->ops, &P_core_ops);

2281         Pinitsym(P);

2283         /*
2284          * Fstat and open the core file and make sure it is a valid ELF core.
2285          */
2286         if (fstat64(P->asfd, &stbuf) == -1) {
2287                 *perr = G_STRANGE;
2288                 goto err;
2289         }

2291         if (core_elf_fdopen(&core, ET_CORE, perr) == -1)
2292                 goto err;

2294         /*
2295          * Allocate and initialize a core_info_t to hang off the ps_prochandle
2296          * structure.  We keep all core-specific information in this structure.
2297          */
2298         if ((core_info = calloc(1, sizeof (core_info_t))) == NULL) {
2299                 *perr = G_STRANGE;
2300                 goto err;
2301         }

2303         P->data = core_info;
2304         list_link(&core_info->core_lwp_head, NULL);
```



```
2305         core_info->core_size = stbuf.st_size;
2306         /*
2307          * In the days before adjustable core file content, this was the
2308          * default core file content. For new core files, this value will
2309          * be overwritten by the NT_CONTENT note section.
2310          */
2311         core_info->core_content = CC_CONTENT_STACK | CC_CONTENT_HEAP |
2312             CC_CONTENT_DATA | CC_CONTENT_RODATA | CC_CONTENT_ANON |
2313             CC_CONTENT_SHANON;

2315         switch (core.e_hdr.e_ident[EI_CLASS]) {
2316         case ELFCLASS32:
2317                 core_info->core_dmodel = PR_MODEL_ILP32;
2318                 break;
2319         case ELFCLASS64:
2320                 core_info->core_dmodel = PR_MODEL_LP64;
2321                 break;
2322         default:
2323                 *perr = G_FORMAT;
2324                 goto err;
2325         }
2326         core_info->core_osabi = core.e_hdr.e_ident[EI_OSABI];

2328         /*
2329          * Because the core file may be a large file, we can't use libelf to
2330          * read the Phdrs.  We use e_phnum and e_phentsize to simplify things.
2331          */
2332         nbytes = core.e_hdr.e_phnum * core.e_hdr.e_phentsize;

2334         if ((phbuf = malloc(nbytes)) == NULL) {
2335                 *perr = G_STRANGE;
2336                 goto err;
2337         }

2339         if (pread64(core_fd, phbuf, nbytes, core.e_hdr.e_phoff) != nbytes) {
2340                 *perr = G_STRANGE;
2341                 free(phbuf);
2342                 goto err;
2343         }

2345         /*
2346          * Iterate through the program headers in the core file.
2347          * We're interested in two types of Phdrs: PT_NOTE (which
2348          * contains a set of saved /proc structures), and PT_LOAD (which
2349          * represents a memory mapping from the process's address space).
2350          * In the case of PT_NOTE, we're interested in the last PT_NOTE
2351          * in the core file; currently the first PT_NOTE (if present)
2352          * contains /proc structs in the pre-2.6 unstructured /proc format.
2353          */
2354         for (php = phbuf, notes = 0, i = 0; i < core.e_hdr.e_phnum; i++) {
2355                 if (core.e_hdr.e_ident[EI_CLASS] == ELFCLASS64)
2356                         (void) memcpy(&phdr, php, sizeof (GElf_Phdr));
2357                 else
2358                         core_phdr_to_gelf(php, &phdr);

2360                 switch (phdr.p_type) {
2361                 case PT_NOTE:
2362                         note_phdr = phdr;
2363                         notes++;
2364                         break;

2366                 case PT_LOAD:
2367                         if (core_add_mapping(P, &phdr) == -1) {
2368                                 *perr = G_STRANGE;
2369                                 free(phbuf);
2370                                 goto err;
```

```
2371                        }
2372                        break;
2373                default:
2374                        dprintf("Pgrab_core: unknown phdr %d\n", phdr.p_type);
2375                        break;
2376                }

2378                php = (char *)php + core.e_hdr.e_phentsize;
2379        }

2381        free(phbuf);

2383        Psort_mappings(P);

2385        /*
2386         * If we couldn't find anything of type PT_NOTE, or only one PT_NOTE
2387         * was present, abort.  The core file is either corrupt or too old.
2388         */
2389        if (notes == 0 || (notes == 1 && core_info->core_osabi ==
2390            ELFOSABI_SOLARIS)) {
2391                *perr = G_NOTE;
2392                goto err;
2393        }

2395        /*
2396         * Advance the seek pointer to the start of the PT_NOTE data
2397         */
2398        if (lseek64(P->asfd, note_phdr.p_offset, SEEK_SET) == (off64_t)-1) {
2399                dprintf("Pgrab_core: failed to lseek to PT_NOTE data\n");
2400                *perr = G_STRANGE;
2401                goto err;
2402        }

2404        /*
2405         * Now process the PT_NOTE structures.  Each one is preceded by
2406         * an Elf{32/64}_Nhdr structure describing its type and size.
2407         *
2408         *  +--------+
2409         *  | header |
2410         *  +--------+
2411         *  | name   |
2412         *  | ...    |
2413         *  +--------+
2414         *  | desc   |
2415         *  | ...    |
2416         *  +--------+
2417         */
2418        for (nleft = note_phdr.p_filesz; nleft > 0; ) {
2419                Elf64_Nhdr nhdr;
2420                off64_t off, namesz, descsz;

2422                /*
2423                 * Although <sys/elf.h> defines both Elf32_Nhdr and Elf64_Nhdr
2424                 * as different types, they are both of the same content and
2425                 * size, so we don't need to worry about 32/64 conversion here.
2426                 */
2427                if (read(P->asfd, &nhdr, sizeof (nhdr)) != sizeof (nhdr)) {
2428                        dprintf("Pgrab_core: failed to read ELF note header\n");
2429                        *perr = G_NOTE;
2430                        goto err;
2431                }

2433                /*
2434                 * According to the System V ABI, the amount of padding
2435                 * following the name field should align the description
2436                 * field on a 4 byte boundary for 32-bit binaries or on an 8
```



```
2437                 * byte boundary for 64-bit binaries. However, this change
2438                 * was not made correctly during the 64-bit port so all
2439                 * descriptions can assume only 4-byte alignment. We ignore
2440                 * the name field and the padding to 4-byte alignment.
2441                 */
2442                namesz = P2ROUNDUP((off64_t)nhdr.n_namesz, (off64_t)4);

2444                if (lseek64(P->asfd, namesz, SEEK_CUR) == (off64_t)-1) {
2445                        dprintf("failed to seek past name and padding\n");
2446                        *perr = G_STRANGE;
2447                        goto err;
2448                }

2450                dprintf("Note hdr n_type=%u n_namesz=%u n_descsz=%u\n",
2451                    nhdr.n_type, nhdr.n_namesz, nhdr.n_descsz);

2453                off = lseek64(P->asfd, (off64_t)0L, SEEK_CUR);

2455                /*
2456                 * Invoke the note handler function from our table
2457                 */
2458                if (nhdr.n_type < sizeof (nhdlrs) / sizeof (nhdlrs[0])) {
2459                        if (nhdlrs[nhdr.n_type](P, nhdr.n_descsz) < 0) {
2460                                dprintf("handler for type %d returned < 0",
2461                                    nhdr.n_type);
2462                                *perr = G_NOTE;
2463                                goto err;
2464                        }
2465                        /*
2466                         * The presence of either of these notes indicates that
2467                         * the dump was generated on Linux.
2468                         */
2469 #ifdef __x86
2470                        if (nhdr.n_type == NT_PRSTATUS ||
2471                            nhdr.n_type == NT_PRPSINFO)
2472                                from_linux = B_TRUE;
2473 #endif
2474                } else {
2475                        (void) note_notsup(P, nhdr.n_descsz);
2476                }

2478                /*
2479                 * Seek past the current note data to the next Elf_Nhdr
2480                 */
2481                descsz = P2ROUNDUP((off64_t)nhdr.n_descsz, (off64_t)4);
2482                if (lseek64(P->asfd, off + descsz, SEEK_SET) == (off64_t)-1) {
2483                        dprintf("Pgrab_core: failed to seek to next nhdr\n");
2484                        *perr = G_STRANGE;
2485                        goto err;
2486                }

2488                /*
2489                 * Subtract the size of the header and its data from what
2490                 * we have left to process.
2491                 */
2492                nleft -= sizeof (nhdr) + namesz + descsz;
2493        }

2495 #ifdef __x86
2496        if (from_linux) {
2497                size_t tcount, pid;
2498                lwp_info_t *lwp;

2500                P->status.pr_dmodel = core_info->core_dmodel;

2502                lwp = list_next(&core_info->core_lwp_head);
```

```
2504                pid = P->status.pr_pid;

2506                for (tcount = 0; tcount < core_info->core_nlwp;
2507                    tcount++, lwp = list_next(lwp)) {
2508                        dprintf("Linux thread with id %d\n", lwp->lwp_id);

2510                        /*
2511                         * In the case we don't have a valid psinfo (i.e. pid is
2512                         * 0, probably because of gdb creating the core) assume
2513                         * lowest pid count is the first thread (what if the
2514                         * next thread wraps the pid around?)
2515                         */
2516                        if (P->status.pr_pid == 0 &&
2517                            ((pid == 0 && lwp->lwp_id > 0) ||
2518                            (lwp->lwp_id < pid))) {
2519                                pid = lwp->lwp_id;
2520                        }
2521                }

2523                if (P->status.pr_pid != pid) {
2524                        dprintf("No valid pid, setting to %ld\n", (ulong_t)pid);
2525                        P->status.pr_pid = pid;
2526                        P->psinfo.pr_pid = pid;
2527                }

2529                /*
2530                 * Consumers like mdb expect the first thread to actually have
2531                 * an id of 1, on linux that is actually the pid. Find the the
2532                 * thread with our process id, and set the id to 1
2533                 */
2534                if ((lwp = lwpid2info(P, pid)) == NULL) {
2535                        dprintf("Couldn't find first thread\n");
2536                        *perr = G_STRANGE;
2537                        goto err;
2538                }

2540                dprintf("setting representative thread: %d\n", lwp->lwp_id);

2542                lwp->lwp_id = 1;
2543                lwp->lwp_status.pr_lwpid = 1;

2545                /* set representative thread */
2546                (void) memcpy(&P->status.pr_lwp, &lwp->lwp_status,
2547                    sizeof (P->status.pr_lwp));
2548        }
2549 #endif /* __x86 */

2551        if (nleft != 0) {
2552                dprintf("Pgrab_core: note section malformed\n");
2553                *perr = G_STRANGE;
2554                goto err;
2555        }

2557        if ((pagesize = Pgetauxval(P, AT_PAGESZ)) == -1) {
2558                pagesize = getpagesize();
2559                dprintf("AT_PAGESZ missing; defaulting to %d\n", pagesize);
2560        }

2562        /*
2563         * Locate and label the mappings corresponding to the end of the
2564         * heap (MA_BREAK) and the base of the stack (MA_STACK).
2565         */
2566        if ((P->status.pr_brkbase != 0 || P->status.pr_brksize != 0) &&
2567            (brk_mp = Paddr2mptr(P, P->status.pr_brkbase +
2568            P->status.pr_brksize - 1)) != NULL)
```



```
2569                brk_mp->map_pmap.pr_mflags |= MA_BREAK;
2570        else
2571                brk_mp = NULL;

2573        if ((stk_mp = Paddr2mptr(P, P->status.pr_stkbase)) != NULL)
2574                stk_mp->map_pmap.pr_mflags |= MA_STACK;

2576        /*
2577         * At this point, we have enough information to look for the
2578         * executable and open it: we have access to the auxv, a psinfo_t,
2579         * and the ability to read from mappings provided by the core file.
2580         */
2581        (void) Pfindexec(P, aout_path, core_exec_open, &aout);
2582        dprintf("P->execname = \"%s\"\n", P->execname ? P->execname : "NULL");
2583        execname = P->execname ? P->execname : "a.out";

2585        /*
2586         * Iterate through the sections, looking for the .dynamic and .interp
2587         * sections.  If we encounter them, remember their section pointers.
2588         */
2589        for (scn = NULL; (scn = elf_nextscn(aout.e_elf, scn)) != NULL; ) {
2590                char *sname;

2592                if ((gelf_getshdr(scn, &shdr) == NULL) ||
2593                    (sname = elf_strptr(aout.e_elf, aout.e_hdr.e_shstrndx,
2594                    (size_t)shdr.sh_name)) == NULL)
2595                        continue;

2597                if (strcmp(sname, ".interp") == 0)
2598                        intp_scn = scn;
2599        }

2601        /*
2602         * Get the AT_BASE auxv element.  If this is missing (-1), then
2603         * we assume this is a statically-linked executable.
2604         */
2605        base_addr = Pgetauxval(P, AT_BASE);

2607        /*
2608         * In order to get librtld_db initialized, we'll need to identify
2609         * and name the mapping corresponding to the run-time linker.  The
2610         * AT_BASE auxv element tells us the address where it was mapped,
2611         * and the .interp section of the executable tells us its path.
2612         * If for some reason that doesn't pan out, just use ld.so.1.
2613         */
2614        if (intp_scn != NULL && (dp = elf_getdata(intp_scn, NULL)) != NULL &&
2615            dp->d_size != 0) {
2616                dprintf(".interp = <%s>\n", (char *)dp->d_buf);
2617                interp = dp->d_buf;

2619        } else if (base_addr != (uintptr_t)-1L) {
2620                if (core_info->core_dmodel == PR_MODEL_LP64)
2621                        interp = "/usr/lib/64/ld.so.1";
2622                else
2623                        interp = "/usr/lib/ld.so.1";

2625                dprintf(".interp section is missing or could not be read; "
2626                    "defaulting to %s\n", interp);
2627        } else
2628                dprintf("detected statically linked executable\n");

2630        /*
2631         * If we have an AT_BASE element, name the mapping at that address
2632         * using the interpreter pathname.  Name the corresponding data
2633         * mapping after the interpreter as well.
2634         */
```

```
2635        if (base_addr != (uintptr_t)-1L) {
2636                elf_file_t intf;

2638                P->map_ldso = core_name_mapping(P, base_addr, interp);

2640                if (core_elf_open(&intf, interp, ET_DYN, NULL) == 0) {
2641                        rd_loadobj_t rl;
2642                        map_info_t *dmp;

2644                        rl.rl_base = base_addr;
2645                        dmp = core_find_data(P, intf.e_elf, &rl);

2647                        if (dmp != NULL) {
2648                                dprintf("renamed data at %p to %s\n",
2649                                    (void *)rl.rl_data_base, interp);
2650                                (void) strncpy(dmp->map_pmap.pr_mapname,
2651                                    interp, PRMAPSZ);
2652                                dmp->map_pmap.pr_mapname[PRMAPSZ - 1] = '\0';
2653                        }
2654                }

2656                core_elf_close(&intf);
2657        }

2659        /*
2660         * If we have an AT_ENTRY element, name the mapping at that address
2661         * using the special name "a.out" just like /proc does.
2662         */
2663        if ((addr = Pgetauxval(P, AT_ENTRY)) != (uintptr_t)-1L)
2664                P->map_exec = core_name_mapping(P, addr, "a.out");

2666        /*
2667         * If we're a statically linked executable, then just locate the
2668         * executable's text and data and name them after the executable.
2669         */
2670        if (base_addr == (uintptr_t)-1L ||
2671            core_info->core_osabi == ELFOSABI_NONE) {
2672                dprintf("looking for text and data: %s\n", execname);
2673                map_info_t *tmp, *dmp;
2674                file_info_t *fp;
2675                rd_loadobj_t rl;

2677                if ((tmp = core_find_text(P, aout.e_elf, &rl)) != NULL &&
2678                    (dmp = core_find_data(P, aout.e_elf, &rl)) != NULL) {
2679                        (void) strncpy(tmp->map_pmap.pr_mapname,
2680                            execname, PRMAPSZ);
2681                        tmp->map_pmap.pr_mapname[PRMAPSZ - 1] = '\0';
2682                        (void) strncpy(dmp->map_pmap.pr_mapname,
2683                            execname, PRMAPSZ);
2684                        dmp->map_pmap.pr_mapname[PRMAPSZ - 1] = '\0';
2685                }

2687                if ((P->map_exec = tmp) != NULL &&
2688                    (fp = malloc(sizeof (file_info_t))) != NULL) {

2690                        (void) memset(fp, 0, sizeof (file_info_t));

2692                        list_link(fp, &P->file_head);
2693                        tmp->map_file = fp;
2694                        P->num_files++;

2696                        fp->file_ref = 1;
2697                        fp->file_fd = -1;

2699                        fp->file_lo = malloc(sizeof (rd_loadobj_t));
2700                        fp->file_lname = strdup(execname);
```



```
2702                        if (fp->file_lo)
2703                                *fp->file_lo = rl;
2704                        if (fp->file_lname)
2705                                fp->file_lbase = basename(fp->file_lname);
2706                        if (fp->file_rname)
2707                                fp->file_rbase = basename(fp->file_rname);

2709                        (void) strcpy(fp->file_pname,
2710                            P->mappings[0].map_pmap.pr_mapname);
2711                        fp->file_map = tmp;

2713                        Pbuild_file_symtab(P, fp);

2715                        if (dmp != NULL) {
2716                                dmp->map_file = fp;
2717                                fp->file_ref++;
2718                        }
2719                }
2720        }

2722        core_elf_close(&aout);

2724        /*
2725         * We now have enough information to initialize librtld_db.
2726         * After it warms up, we can iterate through the load object chain
2727         * in the core, which will allow us to construct the file info
2728         * we need to provide symbol information for the other shared
2729         * libraries, and also to fill in the missing mapping names.
2730         */
2731        rd_log(_libproc_debug);

2733        if ((P->rap = rd_new(P)) != NULL) {
2734                (void) rd_loadobj_iter(P->rap, (rl_iter_f *)
2735                    core_iter_mapping, P);

2737                if (core_info->core_errno != 0) {
2738                        errno = core_info->core_errno;
2739                        *perr = G_STRANGE;
2740                        goto err;
2741                }
2742        } else
2743                dprintf("failed to initialize rtld_db agent\n");

2745        /*
2746         * If there are sections, load them and process the data from any
2747         * sections that we can use to annotate the file_info_t's.
2748         */
2749        core_load_shdrs(P, &core);

2751        /*
2752         * If we previously located a stack or break mapping, and they are
2753         * still anonymous, we now assume that they were MAP_ANON mappings.
2754         * If brk_mp turns out to now have a name, then the heap is still
2755         * sitting at the end of the executable's data+bss mapping: remove
2756         * the previous MA_BREAK setting to be consistent with /proc.
2757         */
2758        if (stk_mp != NULL && stk_mp->map_pmap.pr_mapname[0] == '\0')
2759                stk_mp->map_pmap.pr_mflags |= MA_ANON;
2760        if (brk_mp != NULL && brk_mp->map_pmap.pr_mapname[0] == '\0')
2761                brk_mp->map_pmap.pr_mflags |= MA_ANON;
2762        else if (brk_mp != NULL)
2763                brk_mp->map_pmap.pr_mflags &= ~MA_BREAK;

2765        *perr = 0;
2766        return (P);
```

```
2768 err:
2769         Pfree(P);
2770         core_elf_close(&aout);
2771         return (NULL);
2772 }
```

_____***unchanged_portion_omitted_***