```
**********************************************************
   20688 Mon Jan 19 19:54:47 2015
new/usr/src/lib/libproc/common/Psymtab_machelf32.c
5547 libproc's fake_elf should give up if there's no .hash
5546 libproc's fake_elf may free stack junk when reading corrupt dumps
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

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

#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <string.h>
#include <memory.h>
#include <sys/sysmacros.h>
#include <sys/machelf.h>

#include "Pcontrol.h"
#include "Psymtab_machelf.h"


/*
 * This file contains code for use by Psymtab.c that is compiled once
 * for each supported ELFCLASS.
 *
 * When processing ELF files, it is common to encounter a situation where
 * a program with one ELFCLASS (32 or 64-bit) is required to examine a
 * file with a different ELFCLASS. For example, the 32-bit linker (ld) may
 * be used to link a 64-bit program. The simplest solution to this problem
 * is to duplicate each such piece of code, modifying only the data types,
 * and to use if statements to select the code to run. The problem with
 * doing it that way is that the resulting code is difficult to maintain.
 * It is inevitable that the copies will not always get modified identically,
 * and will drift apart. The only robust solution is to generate the
 * multiple instances of code automatically from a single piece of code.
 *
 * The solution used within the Solaris linker is to write the code once,
 * using the data types defined in sys/machelf.h, and then to compile that
 * code twice, once with _ELF64 defined (to generate ELFCLASS64 code) and
 * once without (to generate ELFCLASS32). We use the same approach here.
 *
 * Note that the _ELF64 definition does not refer to the ELFCLASS of
```



```
 * the resulting code, but rather, to the ELFCLASS of the data it
 * examines. By repeating the above double-compilation for both 32-bit
 * and 64-bit builds, we end up with 4 instances, which collectively
 * can handle any combination of program and ELF data class:
 *
 *                  \  Compilation class
 *                   \    32    64
 *                    \------------------
 *                     |
 *                  32 |   X     X
 *   ELF Data Class    |
 *                  64 |   X     X
 */



/*
 * Read data from the specified process and construct an in memory
 * image of an ELF file that will let us use libelf for most of the
 * work we need to later (e.g. symbol table lookups). This is used
 * in cases where no usable on-disk image for the process is available.
 * We need sections for the dynsym, dynstr, and plt, and we need
 * the program headers from the text section. The former is used in
 * Pbuild_file_symtab(); the latter is used in several functions in
 * Pcore.c to reconstruct the origin of each mapping from the load
 * object that spawned it.
 *
 * Here are some useful pieces of elf trivia that will help
 * to elucidate this code.
 *
 * All the information we need about the dynstr can be found in these
 * two entries in the dynamic section:
 *
 *      DT_STRTAB       base of dynstr
 *      DT_STRSZ        size of dynstr
 *
 * So deciphering the dynstr is pretty straightforward.
 *
 * The dynsym is a little trickier.
 *
 *      DT_SYMTAB       base of dynsym
 *      DT_SYMENT       size of a dynstr entry (Elf{32,64}_Sym)
 *      DT_HASH         base of hash table for dynamic lookups
 *
 * The DT_SYMTAB entry gives us any easy way of getting to the base
 * of the dynsym, but getting the size involves rooting around in the
 * dynamic lookup hash table. Here's the layout of the hash table:
 *
 *              +-------------------+
 *              |       nbucket     |   All values are 32-bit
 *              +-------------------+   (Elf32_Word or Elf64_Word)
 *              |       nchain      |
 *              +-------------------+
 *              |       bucket[0]   |
 *              |         . . .     |
 *              |  bucket[nbucket-1] |
 *              +-------------------+
 *              |       chain[0]    |
 *              |         . . .     |
 *              |   chain[nchain-1]  |
 *              +-------------------+
 *      (figure 5-12 from the SYS V Generic ABI)
 *
 * Symbols names are hashed into a particular bucket which contains
 * an index into the symbol table. Each entry in the symbol table
 * has a corresponding entry in the chain table which tells the
```

```
 127  * consumer where the next entry in the hash chain is. We can use
 128  * the nchain field to find out the size of the dynsym.
 129  *
 130  * If there is a dynsym present, there may also be an optional
 131  * section called the SUNW_ldynsym that augments the dynsym by
 132  * providing local function symbols. When the Solaris linker lays
 133  * out a file that has both of these sections, it makes sure that
 134  * the data for the two sections is adjacent with the SUNW_ldynsym
 135  * in front. This allows the runtime linker to treat these two
 136  * symbol tables as being a single larger table. There are two
 137  * items in the dynamic section for this:
 138  *
 139  *      DT_SUNW_SYMTAB  base of the SUNW_ldynsym
 140  *      DT_SUNW_SYMSZ   total size of SUNW_ldynsym and dynsym
 141  *                      added together. We can figure out the
 142  *                      size of the SUNW_ldynsym section by
 143  *                      subtracting the size of the dynsym
 144  *                      (described above) from this value.
 145  *
 146  * We can figure out the size of the .plt section, but it takes some
 147  * doing. We need to use the following information:
 148  *
 149  *      DT_PLTGOT       GOT PLT entry offset (on x86) or PLT offset (on sparc)
 150  *      DT_JMPREL       base of the PLT's relocation section
 151  *      DT_PLTRELSZ     size of the PLT's relocation section
 152  *      DT_PLTREL       type of the PLT's relocation section
 153  *
 154  * We can use the number of relocation entries to calculate the size of
 155  * the PLT.  We get the address of the PLT by looking up the
 156  * _PROCEDURE_LINKAGE_TABLE_ symbol.
 157  *
 158  * For more information, check out the System V Generic ABI.
 159  */


 162 /*
 163  * The fake_elfXX() function generated by this file uses the following
 164  * string as the string table for the section names. Since it is critical
 165  * to count correctly, and to improve readability, the SHSTR_NDX_ macros
 166  * supply the proper offset for each name within the string.
 167  */
 168 static char shstr[] =
 169         ".shstrtab\0.dynsym\0.dynstr\0.dynamic\0.plt\0.SUNW_ldynsym";

 171 /* Offsets within shstr for each name */
 172 #define SHSTR_NDX_shstrtab      0
 173 #define SHSTR_NDX_dynsym        10
 174 #define SHSTR_NDX_dynstr        18
 175 #define SHSTR_NDX_dynamic       26
 176 #define SHSTR_NDX_plt           35
 177 #define SHSTR_NDX_SUNW_ldynsym  40


 180 /*
 181  * Section header alignment for 32 and 64-bit ELF files differs
 182  */
 183 #ifdef _ELF64
 184 #define SH_ADDRALIGN    8
 185 #else
 186 #define SH_ADDRALIGN    4
 187 #endif

 189 /*
 190  * This is the smallest number of PLT relocation entries allowed in a proper
 191  * .plt section.
 192  */
```



```
 193 #ifdef  __sparc
 194 #define PLTREL_MIN_ENTRIES      4       /* SPARC psABI 3.0 and SCD 2.4 */
 195 #else
 196 #ifdef  __lint
 197 /*
 198  * On x86, lint would complain about unsigned comparison with
 199  * PLTREL_MIN_ENTRIES. This define fakes up the value of PLTREL_MIN_ENTRIES
 200  * and silences lint. On SPARC, there is no such issue.
 201  */
 202 #define PLTREL_MIN_ENTRIES      1
 203 #else
 204 #define PLTREL_MIN_ENTRIES      0
 205 #endif
 206 #endif

 208 #ifdef _ELF64
 209 Elf *
 210 fake_elf64(struct ps_prochandle *P, file_info_t *fptr, uintptr_t addr,
 211     Ehdr *ehdr, uint_t phnum, Phdr *phdr)
 212 #else
 213 Elf *
 214 fake_elf32(struct ps_prochandle *P, file_info_t *fptr, uintptr_t addr,
 215     Ehdr *ehdr, uint_t phnum, Phdr *phdr)
 216 #endif
 217 {
 218         enum {
 219                 DI_PLTGOT,
 220                 DI_JMPREL,
 221                 DI_PLTRELSZ,
 222                 DI_PLTREL,
 223                 DI_SYMTAB,
 224                 DI_HASH,
 225                 DI_SYMENT,
 226                 DI_STRTAB,
 227                 DI_STRSZ,
 228                 DI_SUNW_SYMTAB,
 229                 DI_SUNW_SYMSZ,
 230                 DI_NENT
 231         };
 232         /*
 233          * Mask of dynamic options that must be present in a well
 234          * formed dynamic section. We need all of these in order to
 235          * put together a complete set of elf sections. They are
 236          * mandatory in both executables and shared objects so if one
 237          * of them is missing, we're in some trouble and should abort.
 238          * The PLT items are expected, but we will let them slide if
 239          * need be. The DI_SUNW_SYM* items are completely optional, so
 240          * we use them if they are present and ignore them otherwise.
 241          */
 242         const int di_req_mask = (1 << DI_SYMTAB) | (1 << DI_HASH) |
 243                 (1 << DI_SYMENT) | (1 << DI_STRTAB) | (1 << DI_STRSZ);
 244         int di_mask = 0;
 245         size_t size = 0;
 246         caddr_t elfdata = NULL;
 247         Elf *elf;
 248         size_t dynsym_size = 0, ldynsym_size;
 249         int dynstr_shndx;
 250         Ehdr *ep;
 251         Shdr *sp;
 252         Dyn *dp = NULL;
 252         Dyn *dp;
 253         Dyn *d[DI_NENT] = { 0 };
 254         uint_t i;
 255         Off off;
 256         size_t pltsz = 0, pltentries = 0;
 257         uintptr_t hptr = NULL;
```

```
258         Word hnchains, hnbuckets;

260         if (ehdr->e_type == ET_DYN)
261                 phdr->p_vaddr += addr;

263         if (P->rap != NULL) {
264                 if (rd_get_dyns(P->rap, addr, (void **)&dp, NULL) != RD_OK)
265                         goto bad;
266         } else {
267                 if ((dp = malloc(phdr->p_filesz)) == NULL)
268                         goto bad;
269                 if (Pread(P, dp, phdr->p_filesz, phdr->p_vaddr) !=
270                     phdr->p_filesz)
271                         goto bad;
272         }

274         /*
275          * Iterate over the items in the dynamic section, grabbing
276          * the address of items we want and saving them in dp[].
277          */
278         for (i = 0; i < phdr->p_filesz / sizeof (Dyn); i++) {
279                 switch (dp[i].d_tag) {
280                 /* For the .plt section */
281                 case DT_PLTGOT:
282                         d[DI_PLTGOT] = &dp[i];
283                         break;
284                 case DT_JMPREL:
285                         d[DI_JMPREL] = &dp[i];
286                         break;
287                 case DT_PLTRELSZ:
288                         d[DI_PLTRELSZ] = &dp[i];
289                         break;
290                 case DT_PLTREL:
291                         d[DI_PLTREL] = &dp[i];
292                         break;

294                 /* For the .dynsym section */
295                 case DT_SYMTAB:
296                         d[DI_SYMTAB] = &dp[i];
297                         di_mask |= (1 << DI_SYMTAB);
298                         break;
299                 case DT_HASH:
300                         d[DI_HASH] = &dp[i];
301                         di_mask |= (1 << DI_HASH);
302                         break;
303                 case DT_SYMENT:
304                         d[DI_SYMENT] = &dp[i];
305                         di_mask |= (1 << DI_SYMENT);
306                         break;
307                 case DT_SUNW_SYMTAB:
308                         d[DI_SUNW_SYMTAB] = &dp[i];
309                         break;
310                 case DT_SUNW_SYMSZ:
311                         d[DI_SUNW_SYMSZ] = &dp[i];
312                         break;

314                 /* For the .dynstr section */
315                 case DT_STRTAB:
316                         d[DI_STRTAB] = &dp[i];
317                         di_mask |= (1 << DI_STRTAB);
318                         break;
319                 case DT_STRSZ:
320                         d[DI_STRSZ] = &dp[i];
321                         di_mask |= (1 << DI_STRSZ);
322                         break;
323                 }
```



```
324         }

326         /* Ensure all required entries were collected */
327         if ((di_mask & di_req_mask) != di_req_mask) {
328                 dprintf("text section missing required dynamic entries\n");
329                 goto bad;
330         }

332         /* SUNW_ldynsym must be adjacent to dynsym. Ignore if not */
333         if ((d[DI_SUNW_SYMTAB] != NULL) && (d[DI_SUNW_SYMSZ] != NULL) &&
334             ((d[DI_SYMTAB]->d_un.d_ptr <= d[DI_SUNW_SYMTAB]->d_un.d_ptr) ||
335             (d[DI_SYMTAB]->d_un.d_ptr >= (d[DI_SUNW_SYMTAB]->d_un.d_ptr +
336             d[DI_SUNW_SYMSZ]->d_un.d_val)))) {
337                 d[DI_SUNW_SYMTAB] = NULL;
338                 d[DI_SUNW_SYMSZ] = NULL;
339         }

341         /* elf header */
342         size = sizeof (Ehdr);

344         /* program headers from in-core elf fragment */
345         size += phnum * ehdr->e_phentsize;

347         /* unused shdr, and .shstrtab section */
348         size += sizeof (Shdr);
349         size += sizeof (Shdr);
350         size += roundup(sizeof (shstr), SH_ADDRALIGN);

352         if (d[DI_HASH] != NULL) {
353                 Word hash[2];

355                 hptr = d[DI_HASH]->d_un.d_ptr;
356                 if (ehdr->e_type == ET_DYN)
357                         hptr += addr;

359                 if (Pread(P, hash, sizeof (hash), hptr) != sizeof (hash)) {
360                         dprintf("Pread of .hash at %lx failed\n",
361                             (long)(hptr));
362                         goto bad;
363                 }

365                 hnbuckets = hash[0];
366                 hnchains = hash[1];
367         }

369         if ((d[DI_HASH] == NULL) || (hnbuckets == 0) || (hnchains == 0)) {
370                 dprintf("empty or missing .hash\n");
371                 goto bad;
372         }

374 #endif /* ! codereview */
375         /*
376          * .dynsym and .SUNW_ldynsym sections.
377          *
378          * The string table section used for the symbol table and
379          * dynamic sections lies immediately after the dynsym, so the
380          * presence of SUNW_ldynsym changes the dynstr section index.
381          */
382         if (d[DI_SUNW_SYMTAB] != NULL) {
383                 size += sizeof (Shdr);  /* SUNW_ldynsym shdr */
384                 ldynsym_size = (size_t)d[DI_SUNW_SYMSZ]->d_un.d_val;
385                 dynsym_size = ldynsym_size - (d[DI_SYMTAB]->d_un.d_ptr
386                     - d[DI_SUNW_SYMTAB]->d_un.d_ptr);
387                 ldynsym_size -= dynsym_size;
388                 dynstr_shndx = 4;
389         } else {
```

```
390                 dynsym_size = sizeof (Sym) * hnchains;
391                 ldynsym_size = 0;
392                 dynstr_shndx = 3;
393         }
394         size += sizeof (Shdr) + ldynsym_size + dynsym_size;

396         /* .dynstr section */
397         size += sizeof (Shdr);
398         size += roundup(d[DI_STRSZ]->d_un.d_val, SH_ADDRALIGN);

400         /* .dynamic section */
401         size += sizeof (Shdr);
402         size += roundup(phdr->p_filesz, SH_ADDRALIGN);

404         /* .plt section */
405         if (d[DI_PLTGOT] != NULL && d[DI_JMPREL] != NULL &&
406             d[DI_PLTRELSZ] != NULL && d[DI_PLTREL] != NULL) {
407                 size_t pltrelsz = d[DI_PLTRELSZ]->d_un.d_val;

409                 if (d[DI_PLTREL]->d_un.d_val == DT_RELA) {
410                         pltentries = pltrelsz / sizeof (Rela);
411                 } else if (d[DI_PLTREL]->d_un.d_val == DT_REL) {
412                         pltentries = pltrelsz / sizeof (Rel);
413                 } else {
414                         /* fall back to the platform default */
415 #if ((defined(__i386) || defined(__amd64)) && !defined(_ELF64))
416                         pltentries = pltrelsz / sizeof (Rel);
417                         dprintf("DI_PLTREL not found, defaulting to Rel");
418 #else /* (!(__i386 || __amd64)) || _ELF64 */
419                         pltentries = pltrelsz / sizeof (Rela);
420                         dprintf("DI_PLTREL not found, defaulting to Rela");
421 #endif /* (!(__i386 || __amd64) || _ELF64 */
422                 }

424                 if (pltentries < PLTREL_MIN_ENTRIES) {
425                         dprintf("too few PLT relocation entries "
426                             "(found %lu, expected at least %d)\n",
427                             (long)pltentries, PLTREL_MIN_ENTRIES);
428                         goto bad;
429                 }
430                 if (pltentries < PLTREL_MIN_ENTRIES + 2)
431                         goto done_with_plt;

433                 /*
434                  * Now that we know the number of plt relocation entries
435                  * we can calculate the size of the plt.
436                  */
437                 pltsz = (pltentries + M_PLT_XNumber) * M_PLT_ENTSIZE;
438 #if defined(__sparc)
439                 /* The sparc PLT always has a (delay slot) nop at the end */
440                 pltsz += 4;
441 #endif /* __sparc */

443                 size += sizeof (Shdr);
444                 size += roundup(pltsz, SH_ADDRALIGN);
445         }
446 done_with_plt:

448         if ((elfdata = calloc(1, size)) == NULL)
449                 goto bad;

451         /* LINTED - alignment */
452         ep = (Ehdr *)elfdata;
453         (void) memcpy(ep, ehdr, offsetof(Ehdr, e_phoff));

455         ep->e_ehsize = sizeof (Ehdr);
```



```
456         ep->e_phoff = sizeof (Ehdr);
457         ep->e_phentsize = ehdr->e_phentsize;
458         ep->e_phnum = phnum;
459         ep->e_shoff = ep->e_phoff + phnum * ep->e_phentsize;
460         ep->e_shentsize = sizeof (Shdr);
461         /*
462          * Plt and SUNW_ldynsym sections are optional. C logical
463          * binary operators return a 0 or 1 value, so the following
464          * adds 1 for each optional section present.
465          */
466         ep->e_shnum = 5 + (pltsz != 0) + (d[DI_SUNW_SYMTAB] != NULL);
467         ep->e_shstrndx = 1;

469         /* LINTED - alignment */
470         sp = (Shdr *)(elfdata + ep->e_shoff);
471         off = ep->e_shoff + ep->e_shentsize * ep->e_shnum;

473         /*
474          * Copying the program headers directly from the process's
475          * address space is a little suspect, but since we only
476          * use them for their address and size values, this is fine.
477          */
478         if (Pread(P, &elfdata[ep->e_phoff], phnum * ep->e_phentsize,
479             addr + ehdr->e_phoff) != phnum * ep->e_phentsize) {
480                 dprintf("failed to read program headers\n");
481                 goto bad;
482         }

484         /*
485          * The first elf section is always skipped.
486          */
487         sp++;

489         /*
490          * Section Header: .shstrtab
491          */
492         sp->sh_name = SHSTR_NDX_shstrtab;
493         sp->sh_type = SHT_STRTAB;
494         sp->sh_flags = SHF_STRINGS;
495         sp->sh_addr = 0;
496         sp->sh_offset = off;
497         sp->sh_size = sizeof (shstr);
498         sp->sh_link = 0;
499         sp->sh_info = 0;
500         sp->sh_addralign = 1;
501         sp->sh_entsize = 0;

503         (void) memcpy(&elfdata[off], shstr, sizeof (shstr));
504         off += roundup(sp->sh_size, SH_ADDRALIGN);
505         sp++;

507         /*
508          * Section Header: .SUNW_ldynsym
509          */
510         if (d[DI_SUNW_SYMTAB] != NULL) {
511                 sp->sh_name = SHSTR_NDX_SUNW_ldynsym;
512                 sp->sh_type = SHT_SUNW_LDYNSYM;
513                 sp->sh_flags = SHF_ALLOC;
514                 sp->sh_addr = d[DI_SUNW_SYMTAB]->d_un.d_ptr;
515                 if (ehdr->e_type == ET_DYN)
516                         sp->sh_addr += addr;
517                 sp->sh_offset = off;
518                 sp->sh_size = ldynsym_size;
519                 sp->sh_link = dynstr_shndx;
520                 /* Index of 1st global in table that has none == # items */
521                 sp->sh_info = sp->sh_size / sizeof (Sym);
```

```
522                 sp->sh_addralign = SH_ADDRALIGN;
523                 sp->sh_entsize = sizeof (Sym);
524
525                 if (Pread(P, &elfdata[off], sp->sh_size,
526                     sp->sh_addr) != sp->sh_size) {
527                         dprintf("failed to read .SUNW_ldynsym at %lx\n",
528                             (long)sp->sh_addr);
529                         goto bad;
530                 }
531                 off += sp->sh_size;
532                 /* No need to round up ldynsym data. Dynsym data is same type */
533                 sp++;
534         }
535
536         /*
537          * Section Header: .dynsym
538          */
539         sp->sh_name = SHSTR_NDX_dynsym;
540         sp->sh_type = SHT_DYNSYM;
541         sp->sh_flags = SHF_ALLOC;
542         sp->sh_addr = d[DI_SYMTAB]->d_un.d_ptr;
543         if (ehdr->e_type == ET_DYN)
544                 sp->sh_addr += addr;
545         sp->sh_offset = off;
546         sp->sh_size = dynsym_size;
547         sp->sh_link = dynstr_shndx;
548         sp->sh_info = 1;        /* Index of 1st global in table */
549         sp->sh_addralign = SH_ADDRALIGN;
550         sp->sh_entsize = sizeof (Sym);
551
552         if (Pread(P, &elfdata[off], sp->sh_size,
553             sp->sh_addr) != sp->sh_size) {
554                 dprintf("failed to read .dynsym at %lx\n",
555                     (long)sp->sh_addr);
556                 goto bad;
557         }
558
559         off += roundup(sp->sh_size, SH_ADDRALIGN);
560         sp++;
561
562         /*
563          * Section Header: .dynstr
564          */
565         sp->sh_name = SHSTR_NDX_dynstr;
566         sp->sh_type = SHT_STRTAB;
567         sp->sh_flags = SHF_ALLOC | SHF_STRINGS;
568         sp->sh_addr = d[DI_STRTAB]->d_un.d_ptr;
569         if (ehdr->e_type == ET_DYN)
570                 sp->sh_addr += addr;
571         sp->sh_offset = off;
572         sp->sh_size = d[DI_STRSZ]->d_un.d_val;
573         sp->sh_link = 0;
574         sp->sh_info = 0;
575         sp->sh_addralign = 1;
576         sp->sh_entsize = 0;
577
578         if (Pread(P, &elfdata[off], sp->sh_size,
579             sp->sh_addr) != sp->sh_size) {
580                 dprintf("failed to read .dynstr\n");
581                 goto bad;
582         }
583         off += roundup(sp->sh_size, SH_ADDRALIGN);
584         sp++;
585
586         /*
587          * Section Header: .dynamic
```



```
588          */
589         sp->sh_name = SHSTR_NDX_dynamic;
590         sp->sh_type = SHT_DYNAMIC;
591         sp->sh_flags = SHF_WRITE | SHF_ALLOC;
592         sp->sh_addr = phdr->p_vaddr;
593         if (ehdr->e_type == ET_DYN)
594                 sp->sh_addr -= addr;
595         sp->sh_offset = off;
596         sp->sh_size = phdr->p_filesz;
597         sp->sh_link = dynstr_shndx;
598         sp->sh_info = 0;
599         sp->sh_addralign = SH_ADDRALIGN;
600         sp->sh_entsize = sizeof (Dyn);
601
602         (void) memcpy(&elfdata[off], dp, sp->sh_size);
603         off += roundup(sp->sh_size, SH_ADDRALIGN);
604         sp++;
605
606         /*
607          * Section Header: .plt
608          */
609         if (pltsz != 0) {
610                 ulong_t         plt_symhash;
611                 uint_t          htmp, ndx;
612                 uintptr_t       strtabptr, strtabname;
613                 Sym             sym, *symtabptr;
614                 uint_t          *hash;
615                 char            strbuf[sizeof ("_PROCEDURE_LINKAGE_TABLE_")];
616
617                 /*
618                  * Now we need to find the address of the plt by looking
619                  * up the "_PROCEDURE_LINKAGE_TABLE_" symbol.
620                  */
621
622                 /* get the address of the symtab and strtab sections */
623                 strtabptr = d[DI_STRTAB]->d_un.d_ptr;
624                 symtabptr = (Sym *)(uintptr_t)d[DI_SYMTAB]->d_un.d_ptr;
625                 if (ehdr->e_type == ET_DYN) {
626                         strtabptr += addr;
627                         symtabptr = (Sym*)((uintptr_t)symtabptr + addr);
628                 }
629
630                 /* find the .hash bucket address for this symbol */
631                 plt_symhash = elf_hash("_PROCEDURE_LINKAGE_TABLE_");
632                 htmp = plt_symhash % hnbuckets;
633                 hash = &((uint_t *)hptr)[2 + htmp];
634
635                 /* read the elf hash bucket index */
636                 if (Pread(P, &ndx, sizeof (ndx), (uintptr_t)hash) !=
637                     sizeof (ndx)) {
638                         dprintf("Pread of .hash at %lx failed\n", (long)hash);
639                         goto bad;
640                 }
641
642                 while (ndx) {
643                         if (Pread(P, &sym, sizeof (sym),
644                             (uintptr_t)&symtabptr[ndx]) != sizeof (sym)) {
645                                 dprintf("Pread of .symtab at %lx failed\n",
646                                     (long)&symtabptr[ndx]);
647                                 goto bad;
648                         }
649
650                         strtabname = strtabptr + sym.st_name;
651                         if (Pread_string(P, strbuf, sizeof (strbuf),
652                             strtabname) < 0) {
653                                 dprintf("Pread of .strtab at %lx failed\n",
```

```
654                                     (long)strtabname);
655                                 goto bad;
656                         }

658                         if (strcmp("_PROCEDURE_LINKAGE_TABLE_", strbuf) == 0)
659                                 break;

661                         hash = &((uint_t *)hptr)[2 + hnbuckets + ndx];
662                         if (Pread(P, &ndx, sizeof (ndx), (uintptr_t)hash) !=
663                             sizeof (ndx)) {
664                                 dprintf("Pread of .hash at %lx failed\n",
665                                     (long)hash);
666                                 goto bad;
667                         }
668                 }

670 #if defined(__sparc)
671                 if (sym.st_value != d[DI_PLTGOT]->d_un.d_ptr) {
672                         dprintf("warning: DI_PLTGOT (%lx) doesn't match "
673                             ".plt symbol pointer (%lx)",
674                             (long)d[DI_PLTGOT]->d_un.d_ptr,
675                             (long)sym.st_value);
676                 }
677 #endif /* __sparc */

679                 if (ndx == 0) {
680                         dprintf(
681                             "Failed to find \"_PROCEDURE_LINKAGE_TABLE_\"\n");
682                         goto bad;
683                 }

685                 sp->sh_name = SHSTR_NDX_plt;
686                 sp->sh_type = SHT_PROGBITS;
687                 sp->sh_flags = SHF_WRITE | SHF_ALLOC | SHF_EXECINSTR;
688                 sp->sh_addr = sym.st_value;
689                 if (ehdr->e_type == ET_DYN)
690                         sp->sh_addr += addr;
691                 sp->sh_offset = off;
692                 sp->sh_size = pltsz;
693                 sp->sh_link = 0;
694                 sp->sh_info = 0;
695                 sp->sh_addralign = SH_ADDRALIGN;
696                 sp->sh_entsize = M_PLT_ENTSIZE;

698                 if (Pread(P, &elfdata[off], sp->sh_size, sp->sh_addr) !=
699                     sp->sh_size) {
700                         dprintf("failed to read .plt at %lx\n",
701                             (long)sp->sh_addr);
702                         goto bad;
703                 }
704                 off += roundup(sp->sh_size, SH_ADDRALIGN);
705                 sp++;
706         }

708         /* make sure we didn't write past the end of allocated memory */
709         sp++;
710         assert(((uintptr_t)(sp) - 1) < ((uintptr_t)elfdata + size));

712         free(dp);
713         if ((elf = elf_memory(elfdata, size)) == NULL) {
714                 free(elfdata);
715                 return (NULL);
716         }

718         fptr->file_elfmem = elfdata;
```



```
720         return (elf);

722 bad:
723         if (dp != NULL)
724                 free(dp);
725         if (elfdata != NULL)
726                 free(elfdata);
727         return (NULL);
728 }
```