```
**********************************************************
   22310 Thu Oct 30 22:02:12 2014
new/usr/src/cmd/sgs/libld/common/unwind.c
5270 ld(1) cannot handle CIE version 3 in .eh_frame
**********************************************************
```

```
/*
 * 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) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright 2014 Nexenta Systems, Inc.
 */

#include        <string.h>
#include        <stdio.h>
#include        <sys/types.h>
#include        <sgs.h>
#include        <debug.h>
#include        <_libld.h>
#include        <dwarf.h>
#include        <stdlib.h>

/*
 * A EH_FRAME_HDR consists of the following:
 *
 *      Encoding        Field
 *      --------------------------------
 *      unsigned byte   version
 *      unsigned byte   eh_frame_ptr_enc
 *      unsigned byte   fde_count_enc
 *      unsigned byte   table_enc
 *      encoded         eh_frame_ptr
 *      encoded         fde_count
 *      [ binary search table ]
 *
 * The binary search table entries each consists of:
 *
 *      encoded         initial_func_loc
 *      encoded         FDE_address
 *
 * The entries in the binary search table are sorted
 * in a increasing order by the initial location.
 *
 *
 * version
 *
 *   Version of the .eh_frame_hdr format. This value shall be 1.
 *
```



```
 * eh_frame_ptr_enc
 *
 *    The encoding format of the eh_frame_ptr field.  For shared
 *    libraries the encoding must be
 *    DW_EH_PE_sdata4|DW_EH_PE_pcrel or
 *    DW_EH_PE_sdata4|DW_EH_PE_datarel.
 *
 *
 * fde_count_enc
 *
 *    The encoding format of the fde_count field. A value of
 *    DW_EH_PE_omit indicates the binary search table is not
 *    present.
 *
 * table_enc
 *
 *    The encoding format of the entries in the binary search
 *    table. A value of DW_EH_PE_omit indicates the binary search
 *    table is not present. For shared libraries the encoding
 *    must be DW_EH_PE_sdata4|DW_EH_PE_pcrel or
 *    DW_EH_PE_sdata4|DW_EH_PE_datarel.
 *
 *
 * eh_frame_ptr
 *
 *    The encoded value of the pointer to the start of the
 *    .eh_frame section.
 *
 * fde_count
 *
 *    The encoded value of the count of entries in the binary
 *    search table.
 *
 * binary search table
 *
 *    A binary search table containing fde_count entries. Each
 *    entry of the table consist of two encoded values, the
 *    initial location of the function to which an FDE applies,
 *    and the address of the FDE. The entries are sorted in an
 *    increasing order by the initial location value.
 *
 */


/*
 * EH_FRAME sections
 * =================
 *
 * The call frame information needed for unwinding the stack is output in
 * an ELF section(s) of type SHT_AMD64_UNWIND (amd64) or SHT_PROGBITS (other).
 * In the simplest case there will be one such section per object file and it
 * will be named ".eh_frame".  An .eh_frame section consists of one or more
 * subsections. Each subsection contains a CIE (Common Information Entry)
 * followed by varying number of FDEs (Frame Descriptor Entry). A FDE
 * corresponds to an explicit or compiler generated function in a
 * compilation unit, all FDEs can access the CIE that begins their
 * subsection for data.
 *
 * If an object file contains C++ template instantiations, there shall be
 * a separate CIE immediately preceding each FDE corresponding to an
 * instantiation.
 *
 * Using the preferred encoding specified below, the .eh_frame section can
 * be entirely resolved at link time and thus can become part of the
 * text segment.
 *
```

```
 128  * .eh_frame Section Layout
 129  * ------------------------
 130  *
 131  * EH_PE encoding below refers to the pointer encoding as specified in the
 132  * enhanced LSB Chapter 7 for Eh_Frame_Hdr.
 133  *
 134  * Common Information Entry (CIE)
 135  * ------------------------------
 136  * CIE has the following format:
 137  *
 138  *                             Length
 139  *                               in
 140  *     Field                    Byte      Description
 141  *     -----                   ------     -----------
 142  *  1. Length                    4        Length of CIE (not including
 143  *                                        this 4-byte field).
 144  *
 145  *  2. CIE id                    4        Value Zero (0) for .eh_frame
 146  *                                        (used to distinguish CIEs and
 147  *                                        FDEs when scanning the section)
 148  *
 149  *  3. Version                   1        Value One (1)
 150  *
 151  *  4. CIE Augmentation        string     Null-terminated string with legal
 152  *                                        values being "" or 'z' optionally
 153  *                                        followed by single occurrences of
 154  *                                        'P', 'L', or 'R' in any order.
 155  *     String                             The presence of character(s) in the
 156  *                                        string dictates the content of
 157  *                                        field 8, the Augmentation Section.
 158  *                                        Each character has one or two
 159  *                                        associated operands in the AS.
 160  *                                        Operand order depends on
 161  *                                        position in the string ('z' must
 162  *                                        be first).
 163  *
 164  *  5. Code Align Factor      uleb128     To be multiplied with the
 165  *                                        "Advance Location" instructions in
 166  *                                        the Call Frame Instructions
 167  *
 168  *  6. Data Align Factor      sleb128     To be multiplied with all offset
 169  *                                        in the Call Frame Instructions
 170  *
 171  *  7. Ret Address Reg           1        A "virtual" register representation
 172  *                                        of the return address. In Dwarf V2,
 173  *                                        this is a byte, otherwise it is
 174  *                                        uleb128. It is a byte in gcc 3.3.x
 175  *
 176  *  8. Optional CIE           varying     Present if Augmentation String in
 177  *     Augmentation Section               field 4 is not 0.
 178  *
 179  *     z:
 180  *      size               uleb128       Length of the remainder of the
 181  *                                        Augmentation Section
 182  *
 183  *     P:
 184  *      personality_enc     1            Encoding specifier - preferred
 185  *                                       value is a pc-relative, signed
 186  *                                       4-byte
 187  *
 188  *
 189  *        personality routine (encoded)  Encoded pointer to personality
 190  *                                       routine (actually to the PLT
 191  *                                       entry for the personality
 192  *                                       routine)
 193  *     R:
```



```
 194  *      code_enc           1          Non-default encoding for the
 195  *                                    code-pointers (FDE members
 196  *                                    "initial_location" and "address_range"
 197  *                                    and the operand for DW_CFA_set_loc)
 198  *                                    - preferred value is pc-relative,
 199  *                                    signed 4-byte.
 200  *     L:
 201  *      lsda_enc           1          FDE augmentation bodies may contain
 202  *                                    LSDA pointers. If so they are
 203  *                                    encoded as specified here -
 204  *                                    preferred value is pc-relative,
 205  *                                    signed 4-byte possibly indirect
 206  *                                    thru a GOT entry.
 207  *
 208  *
 209  *  9. Optional Call Frame varying
 210  *     Instructions
 211  *
 212  * The size of the optional call frame instruction area must be computed
 213  * based on the overall size and the offset reached while scanning the
 214  * preceding fields of the CIE.
 215  *
 216  *
 217  * Frame Descriptor Entry (FDE)
 218  * ----------------------------
 219  * FDE has the following format:
 220  *
 221  *                               Length
 222  *                                 in
 223  *     Field                      Byte      Description
 224  *     -----                     ------     -----------
 225  *  1. Length                      4        Length of remainder of this FDE
 226  *
 227  *  2. CIE Pointer                 4        Distance from this field to the
 228  *                                          nearest preceding CIE
 229  *                                          (uthe value is subtracted from the
 230  *                                          current address). This value
 231  *                                          can never be zero and thus can
 232  *                                          be used to distinguish CIE's and
 233  *                                          FDE's when scanning the
 234  *                                          .eh_frame section
 235  *
 236  *  3. Initial Location         varying     Reference to the function code
 237  *                                          corresponding to this FDE.
 238  *                                          If 'R' is missing from the CIE
 239  *                                          Augmentation String, the field is an
 240  *                                          8-byte absolute pointer. Otherwise,
 241  *                                          the corresponding EH_PE encoding in the
 242  *                                          CIE Augmentation Section is used to
 243  *                                          interpret the reference.
 244  *
 245  *  4. Address Range            varying     Size of the function code corresponding
 246  *                                          to this FDE.
 247  *                                          If 'R' is missing from the CIE
 248  *                                          Augmentation String, the field is an
 249  *                                          8-byte unsigned number. Otherwise,
 250  *                                          the size is determined by the
 251  *                                          corresponding EH_PE encoding in the
 252  *                                          CIE Augmentation Section (the
 253  *                                          value is always absolute).
 254  *
 255  *  5. Optional FDE             varying     present if CIE augmentation
 256  *     Augmentation Section                 string is non-empty.
 257  *
 258  *
 259  *     'z':
```

```
 *      length             uleb128       length of the remainder of the
 *                                       FDE augmentation section
 *
 *     'L' (and length > 0):
 *         LSDA               varying     LSDA pointer, encoded in the
 *                                        format specified by the
 *                                        corresponding operand in the CIE's
 *                                        augmentation body.
 *
 *  6. Optional Call          varying
 *     Frame Instructions
 *
 * The size of the optional call frame instruction area must be computed
 * based on the overall size and the offset reached while scanning the
 * preceding fields of the FDE.
 *
 * The overall size of a .eh_frame section is given in the ELF section
 * header.  The only way to determine the number of entries is to scan
 * the section till the end and count.
 *
 */




static uint_t
extract_uint(const uchar_t *data, uint64_t *ndx, int do_swap)
{
        uint_t  r;
        uchar_t *p = (uchar_t *)&r;

        data += *ndx;
        if (do_swap)
                UL_ASSIGN_BSWAP_WORD(p, data);
        else
                UL_ASSIGN_WORD(p, data);

        (*ndx) += 4;
        return (r);
}

/*
 * Create an unwind header (.eh_frame_hdr) output section.
 * The section is created and space reserved, but the data
 * is not copied into place. That is done by a later call
 * to ld_unwind_populate(), after active relocations have been
 * processed.
 *
 * When GNU linkonce processing is in effect, we can end up in a situation
 * where the FDEs related to discarded sections remain in the eh_frame
 * section. Ideally, we would remove these dead entries from eh_frame.
 * However, that optimization has not yet been implemented. In the current
 * implementation, the number of dead FDEs cannot be determined until
 * active relocations are processed, and that processing follows the
 * call to this function. This means that we are unable to detect dead FDEs
 * here, and the section created by this routine is sized for maximum case
 * where all FDEs are valid.
 */
uintptr_t
ld_unwind_make_hdr(Ofl_desc *ofl)
{
        int             bswap = (ofl->ofl_flags1 & FLG_OF1_ENCDIFF) != 0;
        Shdr            *shdr;
        Elf_Data        *elfdata;
        Is_desc         *isp;
```



```
        size_t          size;
        Xword           fde_cnt;
        Aliste          idx1;
        Os_desc         *osp;

        /*
         * we only build a unwind header if we have
         * some unwind information in the file.
         */
        if (ofl->ofl_unwind == NULL)
                return (1);

        /*
         * Allocate and initialize the Elf_Data structure.
         */
        if ((elfdata = libld_calloc(sizeof (Elf_Data), 1)) == NULL)
                return (S_ERROR);
        elfdata->d_type = ELF_T_BYTE;
        elfdata->d_align = ld_targ.t_m.m_word_align;
        elfdata->d_version = ofl->ofl_dehdr->e_version;

        /*
         * Allocate and initialize the Shdr structure.
         */
        if ((shdr = libld_calloc(sizeof (Shdr), 1)) == NULL)
                return (S_ERROR);
        shdr->sh_type = ld_targ.t_m.m_sht_unwind;
        shdr->sh_flags = SHF_ALLOC;
        shdr->sh_addralign = ld_targ.t_m.m_word_align;
        shdr->sh_entsize = 0;

        /*
         * Allocate and initialize the Is_desc structure.
         */
        if ((isp = libld_calloc(1, sizeof (Is_desc))) == NULL)
                return (S_ERROR);
        isp->is_name = MSG_ORIG(MSG_SCN_UNWINDHDR);
        isp->is_shdr = shdr;
        isp->is_indata = elfdata;

        if ((ofl->ofl_unwindhdr = ld_place_section(ofl, isp, NULL,
            ld_targ.t_id.id_unwindhdr, NULL)) == (Os_desc *)S_ERROR)
                return (S_ERROR);

        /*
         * Scan through all of the input Frame information, counting each FDE
         * that requires an index.  Each fde_entry gets a corresponding entry
         * in the binary search table.
         */
        fde_cnt = 0;
        for (APLIST_TRAVERSE(ofl->ofl_unwind, idx1, osp)) {
                Aliste  idx2;
                int     os_isdescs_idx;

                OS_ISDESCS_TRAVERSE(os_isdescs_idx, osp, idx2, isp) {
                        uchar_t         *data;
                        uint64_t        off = 0;

                        data = isp->is_indata->d_buf;
                        size = isp->is_indata->d_size;

                        while (off < size) {
                                uint_t          length, id;
                                uint64_t        ndx = 0;

                                /*
```

```
392                                 * Extract length in lsb format.  A zero length
393                                 * indicates that this CIE is a terminator and
394                                 * that processing for unwind information is
395                                 * complete.
396                                 */
397                                length = extract_uint(data + off, &ndx, bswap);
398                                if (length == 0)
399                                        break;

401                                /*
402                                 * Extract CIE id in lsb format.
403                                 */
404                                id = extract_uint(data + off, &ndx, bswap);

406                                /*
407                                 * A CIE record has a id of '0', otherwise
408                                 * this is a FDE entry and the 'id' is the
409                                 * CIE pointer.
410                                 */
411                                if (id == 0) {
412                                        uint_t  cieversion;
412                                        /*
413                                         * The only CIE version supported
414                                         * is '1' - quick sanity check
415                                         * here.
416                                         */
413                                        cieversion = data[off + ndx];
414                                        ndx += 1;
415                                        /* BEGIN CSTYLED */
416                                        if (cieversion != 1 && cieversion != 3)
420                                        if (cieversion != 1) {
417                                            ld_eprintf(ofl, ERR_FATAL,
418                                                MSG_INTL(MSG_UNW_BADCIEVERS),
419                                                isp->is_file->ifl_name,
420                                                isp->is_name, off);
421                                            return (S_ERROR);
422                                        }
423                                        /* END CSTYLED */
424                                } else {
425                                        fde_cnt++;
426                                }
427                                off += length + 4;
428                        }
429                }
430        }

432        /*
433         * section size:
434         *      byte        version             +1
435         *      byte        eh_frame_ptr_enc    +1
436         *      byte        fde_count_enc       +1
437         *      byte        table_enc           +1
438         *      4 bytes     eh_frame_ptr        +4
439         *      4 bytes     fde_count           +4
440         *      [4 bytes] [4bytes] * fde_count  ...
441         */
442        size = 12 + (8 * fde_cnt);

444        if ((elfdata->d_buf = libld_calloc(size, 1)) == NULL)
445                return (S_ERROR);
446        elfdata->d_size = size;
447        shdr->sh_size = (Xword)size;

449        return (1);
450 }
```

*unchanged_portion_omitted_*


```
482 uintptr_t
483 ld_unwind_populate_hdr(Ofl_desc *ofl)
484 {
485        uchar_t         *hdrdata;
486        uint_t          *binarytable;
487        uint_t          hdroff;
488        Aliste          idx;
489        Addr            hdraddr;
490        Os_desc         *hdrosp;
491        Os_desc         *osp;
492        Os_desc         *first_unwind;
493        uint_t          fde_count;
494        uint_t          *uint_ptr;
495        int             bswap = (ofl->ofl_flags1 & FLG_OF1_ENCDIFF) != 0;

497        /*
498         * Are we building the unwind hdr?
499         */
500        if ((hdrosp = ofl->ofl_unwindhdr) == 0)
501                return (1);

503        hdrdata = hdrosp->os_outdata->d_buf;
504        hdraddr = hdrosp->os_shdr->sh_addr;
505        hdroff = 0;

507        /*
508         * version == 1
509         */
510        hdrdata[hdroff++] = 1;
511        /*
512         * The encodings are:
513         *
514         *  eh_frameptr_enc     sdata4 | pcrel
515         *  fde_count_enc       udata4
516         *  table_enc           sdata4 | datarel
517         */
518        hdrdata[hdroff++] = DW_EH_PE_sdata4 | DW_EH_PE_pcrel;
519        hdrdata[hdroff++] = DW_EH_PE_udata4;
520        hdrdata[hdroff++] = DW_EH_PE_sdata4 | DW_EH_PE_datarel;

522        /*
523         *      Header Offsets
524         *      ----------------------------------
525         *      byte        version             +1
526         *      byte        eh_frame_ptr_enc    +1
527         *      byte        fde_count_enc       +1
528         *      byte        table_enc           +1
529         *      4 bytes     eh_frame_ptr        +4
530         *      4 bytes     fde_count           +4
531         */
532        /* LINTED */
533        binarytable =  (uint_t *)(hdrdata + 12);
534        first_unwind = 0;
535        fde_count = 0;

537        for (APLIST_TRAVERSE(ofl->ofl_unwind, idx, osp)) {
538                uchar_t         *data;
539                size_t          size;
540                uint64_t        off = 0;
541                uint_t          cieRflag = 0, ciePflag = 0;
542                Shdr            *shdr;

544                /*
545                 * remember first UNWIND section to
546                 * point to in the frame_ptr entry.
```

```
547                  */
548                 if (first_unwind == 0)
549                         first_unwind = osp;

551                 data = osp->os_outdata->d_buf;
552                 shdr = osp->os_shdr;
553                 size = shdr->sh_size;

555                 while (off < size) {
556                         uint_t      length, id;
557                         uint64_t    ndx = 0;

559                         /*
560                          * Extract length in lsb format.  A zero length
561                          * indicates that this CIE is a terminator and that
562                          * processing of unwind information is complete.
563                          */
564                         length = extract_uint(data + off, &ndx, bswap);
565                         if (length == 0)
566                                 goto done;

568                         /*
569                          * Extract CIE id in lsb format.
570                          */
571                         id = extract_uint(data + off, &ndx, bswap);

573                         /*
574                          * A CIE record has a id of '0'; otherwise
575                          * this is a FDE entry and the 'id' is the
576                          * CIE pointer.
577                          */
578                         if (id == 0) {
579                                 char    *cieaugstr;
580                                 uint_t  cieaugndx;
581                                 uint_t  cieversion;

583                                 ciePflag = 0;
584                                 cieRflag = 0;
585                                 /*
586                                  * We need to drill through the CIE
587                                  * to find the Rflag.  It's the Rflag
588                                  * which describes how the FDE code-pointers
589                                  * are encoded.
590                                  */

592                                 cieversion = data[off + ndx];
593                                 ndx += 1;
595                                 /*
596                                  * burn through version
597                                  */
598                                 ndx++;

595                                 /*
596                                  * augstr
597                                  */
598                                 cieaugstr = (char *)(&data[off + ndx]);
599                                 ndx += strlen(cieaugstr) + 1;

601                                 /*
602                                  * calign & dalign
603                                  */
604                                 (void) uleb_extract(&data[off], &ndx);
605                                 (void) sleb_extract(&data[off], &ndx);

607                                 /*
608                                  * retreg
```



```
609                                  */
610                                 if (cieversion == 1)
611                                         ndx++;
612                                 else
613                                         (void) uleb_extract(&data[off], &ndx);

614                                 /*
615                                  * we walk through the augmentation
616                                  * section now looking for the Rflag
617                                  */
618                                 for (cieaugndx = 0; cieaugstr[cieaugndx];
619                                     cieaugndx++) {
620                                         /* BEGIN CSTYLED */
621                                         switch (cieaugstr[cieaugndx]) {
622                                         case 'z':
623                                             /* size */
624                                             (void) uleb_extract(&data[off],
625                                                 &ndx);
626                                             break;
627                                         case 'P':
628                                             /* personality */
629                                             ciePflag = data[off + ndx];
630                                             ndx++;
631                                                 /*
632                                                  * Just need to extract the
633                                                  * value to move on to the next
634                                                  * field.
635                                                  */
636                                             (void) dwarf_ehe_extract(
637                                                 &data[off + ndx],
638                                                 &ndx, ciePflag,
639                                                 ofl->ofl_dehdr->e_ident, B_FALSE
640                                                 shdr->sh_addr, off + ndx, 0);
641                                             break;
642                                         case 'R':
643                                             /* code encoding */
644                                             cieRflag = data[off + ndx];
645                                             ndx++;
646                                             break;
647                                         case 'L':
648                                             /* lsda encoding */
649                                             ndx++;
650                                             break;
651                                         }
652                                         /* END CSTYLED */
653                                 }
654                         } else {
655                                 uint_t      bintabndx;
656                                 uint64_t    initloc;
657                                 uint64_t    fdeaddr;
658                                 uint64_t    gotaddr = 0;

660                                 if (ofl->ofl_osgot != NULL)
661                                         gotaddr =
662                                             ofl->ofl_osgot->os_shdr->sh_addr;

664                                 initloc = dwarf_ehe_extract(&data[off],
665                                     &ndx, cieRflag, ofl->ofl_dehdr->e_ident,
666                                     B_FALSE,
667                                     shdr->sh_addr, off + ndx,
668                                     gotaddr);

670                                 /*
671                                  * Ignore FDEs with initloc set to 0.
672                                  * initloc will not be 0 unless this FDE was
673                                  * abandoned due to GNU linkonce processing.
```

```
674                                 * The 0 value occurs because we don't resolve
675                                 * sloppy relocations for unwind header target
676                                 * sections.
677                                 */
678                                if (initloc != 0) {
679                                        bintabndx = fde_count * 2;
680                                        fde_count++;

682                                        /*
683                                         * FDEaddr is adjusted
684                                         * to account for the length & id which
685                                         * have already been consumed.
686                                         */
687                                        fdeaddr = shdr->sh_addr + off;

689                                        binarytable[bintabndx] =
690                                            (uint_t)(initloc - hdraddr);
691                                        binarytable[bintabndx + 1] =
692                                            (uint_t)(fdeaddr - hdraddr);
693                                }
694                        }

696                        /*
697                         * the length does not include the length
698                         * itself - so account for that too.
699                         */
700                        off += length + 4;
701                }
702        }

704 done:
705        /*
706         * Do a quicksort on the binary table. If this is a cross
707         * link from a system with the opposite byte order, xlate
708         * the resulting values into LSB order.
709         */
710        framehdr_addr = hdraddr;
711        qsort((void *)binarytable, (size_t)fde_count,
712            (size_t)(sizeof (uint_t) * 2), bintabcompare);
713        if (bswap) {
714                uint_t  *btable = binarytable;
715                uint_t  cnt;

717                for (cnt = fde_count * 2; cnt-- > 0; btable++)
718                        *btable = ld_bswap_Word(*btable);
719        }

721        /*
722         * Fill in:
723         *      first_frame_ptr
724         *      fde_count
725         */
726        hdroff = 4;
727        /* LINTED */
728        uint_ptr = (uint_t *)(&hdrdata[hdroff]);
729        *uint_ptr = first_unwind->os_shdr->sh_addr -
730            (hdrosp->os_shdr->sh_addr + hdroff);
731        if (bswap)
732                *uint_ptr = ld_bswap_Word(*uint_ptr);

734        hdroff += 4;
735        /* LINTED */
736        uint_ptr = (uint_t *)&hdrdata[hdroff];
737        *uint_ptr = fde_count;
738        if (bswap)
739                *uint_ptr = ld_bswap_Word(*uint_ptr);
```



```
741        /*
742         * If relaxed relocations are active, then there is a chance
743         * that we didn't use all the space reserved for this section.
744         * For details, see the note at head of ld_unwind_make_hdr() above.
745         *
746         * Find the PT_SUNW_UNWIND program header, and change the size values
747         * to the size of the subset of the section that was actually used.
748         */
749        if (ofl->ofl_flags1 & FLG_OF1_RLXREL) {
750                Word    phnum = ofl->ofl_nehdr->e_phnum;
751                Phdr    *phdr = ofl->ofl_phdr;

753                for (; phnum-- > 0; phdr++) {
754                        if (phdr->p_type == PT_SUNW_UNWIND) {
755                                phdr->p_memsz = 12 + (8 * fde_count);
756                                phdr->p_filesz = phdr->p_memsz;
757                                break;
758                        }
759                }
760        }

762        return (1);
763 }
```

*unchanged_portion_omitted_*