****
59492 Tue Apr 23 05:24:00 2019
new/usr/src/common/ctf/ctf_create.c
10812 ctf tools shouldn't add blank labels
10813 ctf symbol mapping needs work
Reviewed by: Jerry Jelinek <jerry.jelinek@joyent.com>
****

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

/*
 * Copyright 2006 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */
/*
 * Copyright (c) 2019, Joyent, Inc.
 * Copyright (c) 2015, Joyent, Inc.
 */

#include <sys/sysmacros.h>
#include <sys/param.h>
#include <sys/mman.h>
#include <ctf_impl.h>
#include <sys/debug.h>

/*
 * This static string is used as the template for initially populating a
 * dynamic container's string table.  We always store \0 in the first byte,
 * and we use the generic string "PARENT" to mark this container's parent
 * if one is associated with the container using ctf_import().
 */
static const char _CTF_STRTAB_TEMPLATE[] = "\0PARENT";

/*
 * To create an empty CTF container, we just declare a zeroed header and call
 * ctf_bufopen() on it.  If ctf_bufopen succeeds, we mark the new container r/w
 * and initialize the dynamic members.  We set dtstrlen to 1 to reserve the
 * first byte of the string table for a \0 byte, and we start assigning type
 * IDs at 1 because type ID 0 is used as a sentinel.
 */
ctf_file_t *
ctf_create(int *errp)
{
        static const ctf_header_t hdr = { { CTF_MAGIC, CTF_VERSION, 0 } };

        const ulong_t hashlen = 128;
        ctf_dtdef_t **hash = ctf_alloc(hashlen * sizeof (ctf_dtdef_t *));
```



```
        ctf_sect_t cts;
        ctf_file_t *fp;

        if (hash == NULL)
                return (ctf_set_open_errno(errp, EAGAIN));

        cts.cts_name = _CTF_SECTION;
        cts.cts_type = SHT_PROGBITS;
        cts.cts_flags = 0;
        cts.cts_data = &hdr;
        cts.cts_size = sizeof (hdr);
        cts.cts_entsize = 1;
        cts.cts_offset = 0;

        if ((fp = ctf_bufopen(&cts, NULL, NULL, errp)) == NULL) {
                ctf_free(hash, hashlen * sizeof (ctf_dtdef_t *));
                return (NULL);
        }

        fp->ctf_flags |= LCTF_RDWR;
        fp->ctf_dthashlen = hashlen;
        bzero(hash, hashlen * sizeof (ctf_dtdef_t *));
        fp->ctf_dthash = hash;
        fp->ctf_dtstrlen = sizeof (_CTF_STRTAB_TEMPLATE);
        fp->ctf_dtnextid = 1;
        fp->ctf_dtoldid = 0;

        return (fp);
}
```

_____unchanged_portion_omitted_

```
int
ctf_add_label(ctf_file_t *fp, const char *name, ctf_id_t type, uint_t position)
{
        ctf_file_t *fpd;
        ctf_dldef_t *dld;

        if (name == NULL)
                return (ctf_set_errno(fp, EINVAL));

        if (!(fp->ctf_flags & LCTF_RDWR))
                return (ctf_set_errno(fp, ECTF_RDONLY));

        fpd = fp;
        if (type != 0 && ctf_lookup_by_id(&fpd, type) == NULL)
                return (CTF_ERR); /* errno is set for us */

        if (type != 0 && (fp->ctf_flags & LCTF_CHILD) &&
            CTF_TYPE_ISPARENT(type))
                return (ctf_set_errno(fp, ECTF_NOPARENT));

        if (ctf_dld_lookup(fp, name) != NULL)
                return (ctf_set_errno(fp, ECTF_LABELEXISTS));

        if ((dld = ctf_alloc(sizeof (ctf_dldef_t))) == NULL)
                return (ctf_set_errno(fp, EAGAIN));

        if ((dld->dld_name = ctf_strdup(name)) == NULL) {
                ctf_free(dld, sizeof (ctf_dldef_t));
                return (ctf_set_errno(fp, EAGAIN));
        }

        ctf_dprintf("adding label %s, %ld\n", name, type);
        dld->dld_type = type;
        fp->ctf_dtstrlen += strlen(name) + 1;
        ctf_dld_insert(fp, dld, position);
```

```
2133         fp->ctf_flags |= LCTF_DIRTY;

2135         return (0);
2136 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    4307 Tue Apr 23 05:24:01 2019
new/usr/src/lib/libctf/common/ctf_convert.c
10812 ctf tools shouldn't add blank labels
10813 ctf symbol mapping needs work
Reviewed by: Jerry Jelinek <jerry.jelinek@joyent.com>
**********************************************************
```

```
/*
 * This file and its contents are supplied under the terms of the
 * Common Development and Distribution License ("CDDL"), version 1.0.
 * You may only use this file in accordance with the terms of version
 * 1.0 of the CDDL.
 *
 * A full copy of the text of the CDDL should have accompanied this
 * source.  A copy of the CDDL is also available via the Internet at
 * http://www.illumos.org/license/CDDL.
 */

/*
 * Copyright 2019 Joyent, Inc.
 * Copyright 2015 Joyent, Inc.
 */

/*
 * Main conversion entry points. This has been designed such that there can be
 * any number of different conversion backends. Currently we only have one that
 * understands DWARFv2 (and bits of DWARFv4). Each backend should be placed in
 * the ctf_converters list and each will be tried in turn.
 */

#include <libctf_impl.h>
#include <gelf.h>

ctf_convert_f ctf_converters[] = {
        ctf_dwarf_convert
};
```

*unchanged_portion_omitted_*

```
static ctf_file_t *
ctf_elfconvert(int fd, Elf *elf, const char *label, uint_t nthrs, uint_t flags,
    int *errp, char *errbuf, size_t errlen)
{
        int err, i;
        ctf_file_t *fp = NULL;
        boolean_t notsup = B_TRUE;
        ctf_convert_source_t type;

        if (errp == NULL)
                errp = &err;

        if (elf == NULL) {
                *errp = EINVAL;
                return (NULL);
        }

        if (flags & ~CTF_CONVERT_F_IGNNONC) {
                *errp = EINVAL;
                return (NULL);
        }

        if (elf_kind(elf) != ELF_K_ELF) {
                *errp = ECTF_FMT;
                return (NULL);
        }

        ctf_convert_ftypes(elf, &type);
```



```
        ctf_dprintf("got types: %d\n", type);
        if (flags & CTF_CONVERT_F_IGNNONC) {
                if (type == CTFCONV_SOURCE_NONE ||
                    (type & CTFCONV_SOURCE_UNKNOWN)) {
                        *errp = ECTF_CONVNOCSRC;
                        return (NULL);
                }
        }

        for (i = 0; i < NCONVERTS; i++) {
                ctf_conv_status_t cs;

                fp = NULL;
                cs = ctf_converters[i](fd, elf, nthrs, errp, &fp, errbuf,
                    errlen);
                if (cs == CTF_CONV_SUCCESS) {
                        notsup = B_FALSE;
                        break;
                }
                if (cs == CTF_CONV_ERROR) {
                        fp = NULL;
                        notsup = B_FALSE;
                        break;
                }
        }

        if (notsup == B_TRUE) {
                if ((flags & CTF_CONVERT_F_IGNNONC) != 0 &&
                    (type & CTFCONV_SOURCE_C) == 0) {
                        *errp = ECTF_CONVNOCSRC;
                        return (NULL);
                }
                *errp = ECTF_NOCONVBKEND;
                return (NULL);
        }

        /*
         * Succsesful conversion.
         */
        if (fp != NULL && label != NULL) {
                if (label == NULL)
                        label = "";
                if (ctf_add_label(fp, label, fp->ctf_typemax, 0) == CTF_ERR) {
                        *errp = ctf_errno(fp);
                        ctf_close(fp);
                        return (NULL);
                }
                if (ctf_update(fp) == CTF_ERR) {
                        *errp = ctf_errno(fp);
                        ctf_close(fp);
                        return (NULL);
                }
        }

        return (fp);
}
```

*unchanged_portion_omitted_*

```
**********************************************************
   82711 Tue Apr 23 05:24:01 2019
new/usr/src/lib/libctf/common/ctf_dwarf.c
10812 ctf tools shouldn't add blank labels
10813 ctf symbol mapping needs work
Reviewed by: Jerry Jelinek <jerry.jelinek@joyent.com>
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2007 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */
/*
 * Copyright 2012 Jason King.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * Copyright 2019 Joyent, Inc.
 * Copyright 2018 Joyent, Inc.
 */

/*
 * CTF DWARF conversion theory.
 *
 * DWARF data contains a series of compilation units. Each compilation unit
 * generally refers to an object file or what once was, in the case of linked
 * binaries and shared objects. Each compilation unit has a series of what DWARF
 * calls a DIE (Debugging Information Entry). The set of entries that we care
 * about have type information stored in a series of attributes. Each DIE also
 * has a tag that identifies the kind of attributes that it has.
 *
 * A given DIE may itself have children. For example, a DIE that represents a
 * structure has children which represent members. Whenever we encounter a DIE
 * that has children or other values or types associated with it, we recursively
 * process those children first so that way we can then refer to the generated
 * CTF type id while processing its parent. This reduces the amount of unknowns
 * and fixups that we need. It also ensures that we don't accidentally add types
 * that an overzealous compiler might add to the DWARF data but aren't used by
 * anything in the system.
 *
 * Once we do a conversion, we store a mapping in an AVL tree that goes from the
 * DWARF's die offset, which is relative to the given compilation unit, to a
 * ctf_id_t.
 *
 * Unfortunately, some compilers actually will emit duplicate entries for a
 * given type that look similar, but aren't quite. To that end, we go through
```



```
 * and do a variant on a merge once we're done processing a single compilation
 * unit which deduplicates all of the types that are in the unit.
 *
 * Finally, if we encounter an object that has multiple compilation units, then
 * we'll convert all of the compilation units separately and then do a merge, so
 * that way we can result in one single ctf_file_t that represents everything
 * for the object.
 *
 * Conversion Steps
 * ----------------
 *
 * Because a given object we've been given to convert may have multiple
 * compilation units, we break the work into two halves. The first half
 * processes each compilation unit (potentially in parallel) and then the second
 * half optionally merges all of the dies in the first half. First, we'll cover
 * what's involved in converting a single ctf_cu_t's dwarf to CTF. This covers
 * the work done in ctf_dwarf_convert_one().
 *
 * An individual ctf_cu_t, which represents a compilation unit, is converted to
 * CTF in a series of multiple passes.
 *
 * Pass 1: During the first pass we walk all of the top-level dies and if we
 * find a function, variable, struct, union, enum or typedef, we recursively
 * transform all of its types. We don't recurse or process everything, because
 * we don't want to add some of the types that compilers may add which are
 * effectively unused.
 *
 * During pass 1, if we encounter any structures or unions we mark them for
 * fixing up later. This is necessary because we may not be able to determine
 * the full size of a structure at the beginning of time. This will happen if
 * the DWARF attribute DW_AT_byte_size is not present for a member. Because of
 * this possibility we defer adding members to structures or even converting
 * them during pass 1 and save that for pass 2. Adding all of the base
 * structures without any of their members helps deal with any circular
 * dependencies that we might encounter.
 *
 * Pass 2: This pass is used to do the first half of fixing up structures and
 * unions. Rather than walk the entire type space again, we actually walk the
 * list of structures and unions that we marked for later fixing up. Here, we
 * iterate over every structure and add members to the underlying ctf_file_t,
 * but not to the structs themselves. One might wonder why we don't, and the
 * main reason is that libctf requires a ctf_update() be done before adding the
 * members to structures or unions.
 *
 * Pass 3: This pass is used to do the second half of fixing up structures and
 * unions. During this part we always go through and add members to structures
 * and unions that we added to the container in the previous pass. In addition,
 * we set the structure and union's actual size, which may have additional
 * padding added by the compiler, it isn't simply the last offset. DWARF always
 * guarantees an attribute exists for this. Importantly no ctf_id_t's change
 * during pass 2.
 *
 * Pass 4: The next phase is to add CTF entries for all of the symbols and
 * variables that are present in this die. During pass 1 we added entries to a
 * map for each variable and function. During this pass, we iterate over the
 * symbol table and when we encounter a symbol that we have in our lists of
 * translated information which matches, we then add it to the ctf_file_t.
 *
 * Pass 5: Here we go and look for any weak symbols and functions and see if
 * they match anything that we recognize. If so, then we add type information
 * for them at this point based on the matching type.
 *
 * Pass 6: This pass is actually a variant on a merge. The traditional merge
 * process expects there to be no duplicate types. As such, at the end of
 * conversion, we do a dedup on all of the types in the system. The
 * deduplication process is described in lib/libctf/common/ctf_merge.c.
```

```
 125  *
 126  * Once pass 6 is done, we've finished processing the individual compilation
 127  * unit.
 128  *
 129  * The following steps reflect the general process of doing a conversion.
 130  *
 131  * 1) Walk the dwarf section and determine the number of compilation units
 132  * 2) Create a ctf_cu_t for each compilation unit
 133  * 3) Add all ctf_cu_t's to a workq
 134  * 4) Have the workq process each die with ctf_dwarf_convert_one. This itself
 135  *    is comprised of several steps, which were already enumerated.
 136  * 5) If we have multiple cu's, we do a ctf merge of all the dies. The mechanics
 137  *    of the merge are discussed in lib/libctf/common/ctf_merge.c.
 138  * 6) Free everything up and return a ctf_file_t to the user. If we only had a
 139  *    single compilation unit, then we give that to the user. Otherwise, we
 140  *    return the merged ctf_file_t.
 141  *
 142  * Threading
 143  * ---------
 144  *
 145  * The process has been designed to be amenable to threading. Each compilation
 146  * unit has its own type stream, therefore the logical place to divide and
 147  * conquer is at the compilation unit. Each ctf_cu_t has been built to be able
 148  * to be processed independently of the others. It has its own libdwarf handle,
 149  * as a given libdwarf handle may only be used by a single thread at a time.
 150  * This allows the various ctf_cu_t's to be processed in parallel by different
 151  * threads.
 152  *
 153  * All of the ctf_cu_t's are loaded into a workq which allows for a number of
 154  * threads to be specified and used as a thread pool to process all of the
 155  * queued work. We set the number of threads to use in the workq equal to the
 156  * number of threads that the user has specified.
 157  *
 158  * After all of the compilation units have been drained, we use the same number
 159  * of threads when performing a merge of multiple compilation units, if they
 160  * exist.
 161  *
 162  * While all of these different parts do support and allow for multiple threads,
 163  * it's important that when only a single thread is specified, that it be the
 164  * calling thread. This allows the conversion routines to be used in a context
 165  * that doesn't allow additional threads, such as rtld.
 166  *
 167  * Common DWARF Mechanics and Notes
 168  * --------------------------------
 169  *
 170  * At this time, we really only support DWARFv2, though support for DWARFv4 is
 171  * mostly there. There is no intent to support DWARFv3.
 172  *
 173  * Generally types for something are stored in the DW_AT_type attribute. For
 174  * example, a function's return type will be stored in the local DW_AT_type
 175  * attribute while the arguments will be in child DIEs. There are also various
 176  * times when we don't have any DW_AT_type. In that case, the lack of a type
 177  * implies, at least for C, that its C type is void. Because DWARF doesn't emit
 178  * one, we have a synthetic void type that we create and manipulate instead and
 179  * pass it off to consumers on an as-needed basis. If nothing has a void type,
 180  * it will not be emitted.
 181  *
 182  * Architecture Specific Parts
 183  * ---------------------------
 184  *
 185  * The CTF tooling encodes various information about the various architectures
 186  * in the system. Importantly, the tool assumes that every architecture has a
 187  * data model where long and pointer are the same size. This is currently the
 188  * case, as the two data models illumos supports are ILP32 and LP64.
 189  *
 190  * In addition, we encode the mapping of various floating point sizes to various
```



```
 191  * types for each architecture. If a new architecture is being added, it should
 192  * be added to the list. The general design of the ctf conversion tools is to be
 193  * architecture independent. eg. any of the tools here should be able to convert
 194  * any architecture's DWARF into ctf; however, this has not been rigorously
 195  * tested and more importantly, the ctf routines don't currently write out the
 196  * data in an endian-aware form, they only use that of the currently running
 197  * library.
 198  */

 200 #include <libctf_impl.h>
 201 #include <sys/avl.h>
 202 #include <sys/debug.h>
 203 #include <gelf.h>
 204 #include <libdwarf.h>
 205 #include <dwarf.h>
 206 #include <libgen.h>
 207 #include <workq.h>
 208 #include <errno.h>

 210 #define DWARF_VERSION_TWO       2
 211 #define DWARF_VARARGS_NAME      "..."

 213 /*
 214  * Dwarf may refer recursively to other types that we've already processed. To
 215  * see if we've already converted them, we look them up in an AVL tree that's
 216  * sorted by the DWARF id.
 217  */
 218 typedef struct ctf_dwmap {
 219         avl_node_t      cdm_avl;
 220         Dwarf_Off       cdm_off;
 221         Dwarf_Die       cdm_die;
 222         ctf_id_t        cdm_id;
 223         boolean_t       cdm_fix;
 224 } ctf_dwmap_t;
_____unchanged_portion_omitted_

 276 static int ctf_dwarf_offset(ctf_cu_t *, Dwarf_Die, Dwarf_Off *);
 277 static int ctf_dwarf_convert_die(ctf_cu_t *, Dwarf_Die);
 278 static int ctf_dwarf_convert_type(ctf_cu_t *, Dwarf_Die, ctf_id_t *, int);

 280 static int ctf_dwarf_function_count(ctf_cu_t *, Dwarf_Die, ctf_funcinfo_t *,
 281     boolean_t);
 282 static int ctf_dwarf_convert_fargs(ctf_cu_t *, Dwarf_Die, ctf_funcinfo_t *,
 283     ctf_id_t *);

 285 typedef int (ctf_dwarf_symtab_f)(ctf_cu_t *, const GElf_Sym *, ulong_t,
 286     const char *, const char *, void *);

 285 /*
 286  * This is a generic way to set a CTF Conversion backend error depending on what
 287  * we were doing. Unless it was one of a specific set of errors that don't
 288  * indicate a programming / translation bug, eg. ENOMEM, then we transform it
 289  * into a CTF backend error and fill in the error buffer.
 290  */
 291 static int
 292 ctf_dwarf_error(ctf_cu_t *cup, ctf_file_t *cfp, int err, const char *fmt, ...)
 293 {
 294         va_list ap;
 295         int ret;
 296         size_t off = 0;
 297         ssize_t rem = cup->cu_errlen;
 298         if (cfp != NULL)
 299                 err = ctf_errno(cfp);

 301         if (err == ENOMEM)
 302                 return (err);
```

```
 304         ret = snprintf(cup->cu_errbuf, rem, "die %s: ", cup->cu_name);
 305         if (ret < 0)
 306                 goto err;
 307         off += ret;
 308         rem = MAX(rem - ret, 0);

 310         va_start(ap, fmt);
 311         ret = vsnprintf(cup->cu_errbuf + off, rem, fmt, ap);
 312         va_end(ap);
 313         if (ret < 0)
 314                 goto err;

 316         off += ret;
 317         rem = MAX(rem - ret, 0);
 318         if (fmt[strlen(fmt) - 1] != '\n') {
 319                 (void) snprintf(cup->cu_errbuf + off, rem,
 320                     ": %s\n", ctf_errmsg(err));
 321         }
 322         va_end(ap);
 323         return (ECTF_CONVBKERR);

 325 err:
 326         cup->cu_errbuf[0] = '\0';
 327         return (ECTF_CONVBKERR);
 328 }
```

*unchanged_portion_omitted_*

```
2166 /*
2167  * The DWARF information about a symbol and the information in the symbol table
2168  * may not be the same due to symbol reduction that is performed by ld due to a
2169  * mapfile or other such directive. We process weak symbols at a later time.
2170  *
2171  * The following are the rules that we employ:
2172  *
2173  * 1. A DWARF function that is considered exported matches STB_GLOBAL entries
2174  * with the same name.
2175  *
2176  * 2. A DWARF function that is considered exported matches STB_LOCAL entries
2177  * with the same name and the same file. This case may happen due to mapfile
2178  * reduction.
2179  *
2180  * 3. A DWARF function that is not considered exported matches STB_LOCAL entries
2181  * with the same name and the same file.
2182  *
2183  * 4. A DWARF function that has the same name as the symbol table entry, but the
2184  * files do not match. This is considered a 'fuzzy' match. This may also happen
2185  * due to a mapfile reduction. Fuzzy matching is only used when we know that the
2186  * file in question refers to the primary object. This is because when a symbol
2187  * is reduced in a mapfile, it's always going to be tagged as a local value in
2188  * the generated output and it is considered as to belong to the primary file
2189  * which is the first STT_FILE symbol we see.
2190  */
2191 static boolean_t
2192 ctf_dwarf_symbol_match(const char *symtab_file, const char *symtab_name,
2193     uint_t symtab_bind, const char *dwarf_file, const char *dwarf_name,
2194     boolean_t dwarf_global, boolean_t *is_fuzzy)
2195 {
2196         *is_fuzzy = B_FALSE;

2198         if (symtab_bind != STB_LOCAL && symtab_bind != STB_GLOBAL) {
2199                 return (B_FALSE);
2200         }

2202         if (strcmp(symtab_name, dwarf_name) != 0) {
2203                 return (B_FALSE);
```



```
2204         }

2206         if (symtab_bind == STB_GLOBAL) {
2207                 return (dwarf_global);
2208         }

2210         if (strcmp(symtab_file, dwarf_file) == 0) {
2211                 return (B_TRUE);
2212         }

2214         if (dwarf_global) {
2215                 *is_fuzzy = B_TRUE;
2216                 return (B_TRUE);
2217         }

2219         return (B_FALSE);
2220 }

2222 static ctf_dwfunc_t *
2223 ctf_dwarf_match_func(ctf_cu_t *cup, const char *file, const char *name,
2224     uint_t bind, boolean_t primary)
2171     int bind)
2225 {
2226         ctf_dwfunc_t *cdf, *fuzzy = NULL;
2173         ctf_dwfunc_t *cdf;

2228         if (bind == STB_WEAK)
2229                 return (NULL);

2178         /* Nothing we can do if we can't find a name to compare it to. */
2231         if (bind == STB_LOCAL && (file == NULL || cup->cu_name == NULL))
2232                 return (NULL);

2234         for (cdf = ctf_list_next(&cup->cu_funcs); cdf != NULL;
2235             cdf = ctf_list_next(cdf)) {
2236                 boolean_t is_fuzzy = B_FALSE;

2238                 if (ctf_dwarf_symbol_match(file, name, bind, cup->cu_name,
2239                     cdf->cdf_name, cdf->cdf_global, &is_fuzzy)) {
2240                         if (is_fuzzy) {
2241                                 if (primary) {
2242                                         fuzzy = cdf;
2243                                 }
2184                 if (bind == STB_GLOBAL && cdf->cdf_global == B_FALSE)
2244                                         continue;
2245                         } else {
2186                 if (bind == STB_LOCAL && cdf->cdf_global == B_TRUE)
2187                         continue;
2188                 if (strcmp(name, cdf->cdf_name) != 0)
2189                         continue;
2190                 if (bind == STB_LOCAL && strcmp(file, cup->cu_name) != 0)
2191                         continue;
2246                                 return (cdf);
2247                         }
2248                 }
2249         }

2251         return (fuzzy);
2195         return (NULL);
2252 }

2254 static ctf_dwvar_t *
2255 ctf_dwarf_match_var(ctf_cu_t *cup, const char *file, const char *name,
2256     uint_t bind, boolean_t primary)
2199     int bind)
2257 {
```

```
2258         ctf_dwvar_t *cdv, *fuzzy = NULL;
2201         ctf_dwvar_t *cdv;

2260         if (bind == STB_WEAK)
2261                 return (NULL);

2203         /* Nothing we can do if we can't find a name to compare it to. */
2263         if (bind == STB_LOCAL && (file == NULL || cup->cu_name == NULL))
2264                 return (NULL);
2206         ctf_dprintf("Still considering %s\n", name);

2266         for (cdv = ctf_list_next(&cup->cu_vars); cdv != NULL;
2267             cdv = ctf_list_next(cdv)) {
2268                 boolean_t is_fuzzy = B_FALSE;
2210                 if (bind == STB_GLOBAL && cdv->cdv_global == B_FALSE)
2211                         continue;
2212                 if (bind == STB_LOCAL && cdv->cdv_global == B_TRUE)
2213                         continue;
2214                 if (strcmp(name, cdv->cdv_name) != 0)
2215                         continue;
2216                 if (bind == STB_LOCAL && strcmp(file, cup->cu_name) != 0)
2217                         continue;
2218                 return (cdv);
2219         }

2270                 if (ctf_dwarf_symbol_match(file, name, bind, cup->cu_name,
2271                     cdv->cdv_name, cdv->cdv_global, &is_fuzzy)) {
2272                         if (is_fuzzy) {
2273                                 if (primary) {
2274                                         fuzzy = cdv;
2221         return (NULL);
2222 }

2224 static int
2225 ctf_dwarf_symtab_iter(ctf_cu_t *cup, ctf_dwarf_symtab_f *func, void *arg)
2226 {
2227         int ret;
2228         ulong_t i;
2229         ctf_file_t *fp = cup->cu_ctfp;
2230         const char *file = NULL;
2231         uintptr_t symbase = (uintptr_t)fp->ctf_symtab.cts_data;
2232         uintptr_t strbase = (uintptr_t)fp->ctf_strtab.cts_data;

2234         for (i = 0; i < fp->ctf_nsyms; i++) {
2235                 const char *name;
2236                 int type;
2237                 GElf_Sym gsym;
2238                 const GElf_Sym *gsymp;

2240                 if (fp->ctf_symtab.cts_entsize == sizeof (Elf32_Sym)) {
2241                         const Elf32_Sym *symp = (Elf32_Sym *)symbase + i;
2242                         type = ELF32_ST_TYPE(symp->st_info);
2243                         if (type == STT_FILE) {
2244                                 file = (char *)(strbase + symp->st_name);
2245                                 continue;
2275                                 }
2247                         if (type != STT_OBJECT && type != STT_FUNC)
2248                                 continue;
2249                         if (ctf_sym_valid(strbase, type, symp->st_shndx,
2250                             symp->st_value, symp->st_name) == B_FALSE)
2251                                 continue;
2252                         name = (char *)(strbase + symp->st_name);
2253                         gsym.st_name = symp->st_name;
2254                         gsym.st_value = symp->st_value;
2255                         gsym.st_size = symp->st_size;
2256                         gsym.st_info = symp->st_info;
```



```
2257                         gsym.st_other = symp->st_other;
2258                         gsym.st_shndx = symp->st_shndx;
2259                         gsymp = &gsym;
2276                         } else {
2277                                 return (cdv);
2261                         const Elf64_Sym *symp = (Elf64_Sym *)symbase + i;
2262                         type = ELF64_ST_TYPE(symp->st_info);
2263                         if (type == STT_FILE) {
2264                                 file = (char *)(strbase + symp->st_name);
2265                                 continue;
2278                         }
2267                         if (type != STT_OBJECT && type != STT_FUNC)
2268                                 continue;
2269                         if (ctf_sym_valid(strbase, type, symp->st_shndx,
2270                             symp->st_value, symp->st_name) == B_FALSE)
2271                                 continue;
2272                         name = (char *)(strbase + symp->st_name);
2273                         gsymp = symp;
2279                 }

2276                 ret = func(cup, gsymp, i, file, name, arg);
2277                 if (ret != 0)
2278                         return (ret);
2280         }

2282         return (fuzzy);
2281         return (0);
2283 }

2285 static int
2286 ctf_dwarf_conv_funcvars_cb(const Elf64_Sym *symp, ulong_t idx,
2287     const char *file, const char *name, boolean_t primary, void *arg)
2285 ctf_dwarf_conv_funcvars_cb(ctf_cu_t *cup, const GElf_Sym *symp, ulong_t idx,
2286     const char *file, const char *name, void *arg)
2288 {
2289         int ret;
2290         uint_t bind, type;
2291         ctf_cu_t *cup = arg;
2288         int ret, bind, type;

2293         bind = GELF_ST_BIND(symp->st_info);
2294         type = GELF_ST_TYPE(symp->st_info);

2296         /*
2297          * Come back to weak symbols in another pass
2298          */
2299         if (bind == STB_WEAK)
2300                 return (0);

2302         if (type == STT_OBJECT) {
2303                 ctf_dwvar_t *cdv = ctf_dwarf_match_var(cup, file, name,
2304                     bind, primary);
2301                     bind);
2302                 ctf_dprintf("match for %s (%d): %p\n", name, idx, cdv);
2305                 if (cdv == NULL)
2306                         return (0);
2307                 ret = ctf_add_object(cup->cu_ctfp, idx, cdv->cdv_type);
2308                 ctf_dprintf("added object %s->%ld\n", name, cdv->cdv_type);
2306                 ctf_dprintf("added object %s\n", name);
2309         } else {
2310                 ctf_dwfunc_t *cdf = ctf_dwarf_match_func(cup, file, name,
2311                     bind, primary);
2309                     bind);
2312                 if (cdf == NULL)
2313                         return (0);
2314                 ret = ctf_add_function(cup->cu_ctfp, idx, &cdf->cdf_fip,
```

```
2315                    cdf->cdf_argv);
2316                ctf_dprintf("added function %s\n", name);
2317        }

2319        if (ret == CTF_ERR) {
2320                return (ctf_errno(cup->cu_ctfp));
2321        }

2323        return (0);
2324 }

2326 static int
2327 ctf_dwarf_conv_funcvars(ctf_cu_t *cup)
2328 {
2329        return (ctf_symtab_iter(cup->cu_ctfp, ctf_dwarf_conv_funcvars_cb, cup));
2326        return (ctf_dwarf_symtab_iter(cup, ctf_dwarf_conv_funcvars_cb, NULL));
2330 }

2332 /*
2333  * If we have a weak symbol, attempt to find the strong symbol it will resolve
2334  * to.  Note: the code where this actually happens is in sym_process() in
2335  * cmd/sgs/libld/common/syms.c
2336  *
2337  * Finding the matching symbol is unfortunately not trivial.  For a symbol to be
2338  * a candidate, it must:
2339  *
2340  * - have the same type (function, object)
2341  * - have the same value (address)
2342  * - have the same size
2343  * - not be another weak symbol
2344  * - belong to the same section (checked via section index)
2345  *
2346  * To perform this check, we first iterate over the symbol table. For each weak
2347  * symbol that we encounter, we then do a second walk over the symbol table,
2348  * calling ctf_dwarf_conv_check_weak(). If a symbol matches the above, then it's
2349  * either a local or global symbol. If we find a global symbol then we go with
2350  * it and stop searching for additional matches.
2351  *
2352  * If instead, we find a local symbol, things are more complicated. The first
2353  * thing we do is to try and see if we have file information about both symbols
2354  * (STT_FILE). If they both have file information and it matches, then we treat
2355  * that as a good match and stop searching for additional matches.
2356  *
2357  * Otherwise, this means we have a non-matching file and a local symbol. We
2358  * treat this as a candidate and if we find a better match (one of the two cases
2359  * above), use that instead. There are two different ways this can happen.
2360  * Either this is a completely different symbol, or it's a once-global symbol
2361  * that was scoped to local via a mapfile.  In the former case, curfile is
2362  * likely inaccurate since the linker does not preserve the needed curfile in
2363  * the order of the symbol table (see the comments about locally scoped symbols
2364  * in libld's update_osym()).  As we can't tell this case from the former one,
2365  * we use this symbol iff no other matching symbol is found.
2366  *
2367  * What we really need here is a SUNW section containing weak<->strong mappings
2368  * that we can consume.
2369  */
2370 typedef struct ctf_dwarf_weak_arg {
2371        const Elf64_Sym *cweak_symp;
2368        const GElf_Sym *cweak_symp;
2372        const char *cweak_file;
2373        boolean_t cweak_candidate;
2374        ulong_t cweak_idx;
2375 } ctf_dwarf_weak_arg_t;

2377 static int
2378 ctf_dwarf_conv_check_weak(const Elf64_Sym *symp, ulong_t idx, const char *file,
```



```
2379     const char *name, boolean_t primary, void *arg)
2375 ctf_dwarf_conv_check_weak(ctf_cu_t *cup, const GElf_Sym *symp,
2376     ulong_t idx, const char *file, const char *name, void *arg)
2380 {
2381        ctf_dwarf_weak_arg_t *cweak = arg;
2379        const GElf_Sym *wsymp = cweak->cweak_symp;

2383        const Elf64_Sym *wsymp = cweak->cweak_symp;

2385        ctf_dprintf("comparing weak to %s\n", name);

2387        if (GELF_ST_BIND(symp->st_info) == STB_WEAK) {
2388                return (0);
2389        }

2391        if (GELF_ST_TYPE(wsymp->st_info) != GELF_ST_TYPE(symp->st_info)) {
2392                return (0);
2393        }

2395        if (wsymp->st_value != symp->st_value) {
2396                return (0);
2397        }

2399        if (wsymp->st_size != symp->st_size) {
2400                return (0);
2401        }

2403        if (wsymp->st_shndx != symp->st_shndx) {
2404                return (0);
2405        }

2407        /*
2408         * Check if it's a weak candidate.
2409         */
2410        if (GELF_ST_BIND(symp->st_info) == STB_LOCAL &&
2411            (file == NULL || cweak->cweak_file == NULL ||
2412            strcmp(file, cweak->cweak_file) != 0)) {
2413                cweak->cweak_candidate = B_TRUE;
2414                cweak->cweak_idx = idx;
2415                return (0);
2416        }

2418        /*
2419         * Found a match, break.
2420         */
2421        cweak->cweak_idx = idx;
2422        return (1);
2423 }
______unchanged_portion_omitted_

2482 static int
2483 ctf_dwarf_conv_weaks_cb(const Elf64_Sym *symp, ulong_t idx, const char *file,
2484     const char *name, boolean_t primary, void *arg)
2479 ctf_dwarf_conv_weaks_cb(ctf_cu_t *cup, const GElf_Sym *symp,
2480     ulong_t idx, const char *file, const char *name, void *arg)
2485 {
2486        int ret, type;
2487        ctf_dwarf_weak_arg_t cweak;
2488        ctf_cu_t *cup = arg;

2490        /*
2491         * We only care about weak symbols.
2492         */
2493        if (GELF_ST_BIND(symp->st_info) != STB_WEAK)
2494                return (0);
```

```
2496         type = GELF_ST_TYPE(symp->st_info);
2497         ASSERT(type == STT_OBJECT || type == STT_FUNC);

2499         /*
2500          * For each weak symbol we encounter, we need to do a second iteration
2501          * to try and find a match. We should probably think about other
2502          * techniques to try and save us time in the future.
2503          */
2504         cweak.cweak_symp = symp;
2505         cweak.cweak_file = file;
2506         cweak.cweak_candidate = B_FALSE;
2507         cweak.cweak_idx = 0;

2509         ctf_dprintf("Trying to find weak equiv for %s\n", name);

2511         ret = ctf_symtab_iter(cup->cu_ctfp, ctf_dwarf_conv_check_weak, &cweak);
2506         ret = ctf_dwarf_symtab_iter(cup, ctf_dwarf_conv_check_weak, &cweak);
2512         VERIFY(ret == 0 || ret == 1);

2514         /*
2515          * Nothing was ever found, we're not going to add anything for this
2516          * entry.
2517          */
2518         if (ret == 0 && cweak.cweak_candidate == B_FALSE) {
2519                 ctf_dprintf("found no weak match for %s\n", name);
2520                 return (0);
2521         }

2523         /*
2524          * Now, finally go and add the type based on the match.
2525          */
2526         ctf_dprintf("matched weak symbol %lu to %lu\n", idx, cweak.cweak_idx);
2527         if (type == STT_OBJECT) {
2528                 ret = ctf_dwarf_duplicate_sym(cup, idx, cweak.cweak_idx);
2529         } else {
2530                 ret = ctf_dwarf_duplicate_func(cup, idx, cweak.cweak_idx);
2531         }

2533         return (ret);
2534 }

2536 static int
2537 ctf_dwarf_conv_weaks(ctf_cu_t *cup)
2538 {
2539         return (ctf_symtab_iter(cup->cu_ctfp, ctf_dwarf_conv_weaks_cb, cup));
2533         return (ctf_dwarf_symtab_iter(cup, ctf_dwarf_conv_weaks_cb, NULL));
2540 }

2542 /* ARGSUSED */
2543 static int
2544 ctf_dwarf_convert_one(void *arg, void *unused)
2545 {
2546         int ret;
2547         ctf_file_t *dedup;
2548         ctf_cu_t *cup = arg;

2550         ctf_dprintf("converting die: %s\n", cup->cu_name);
2551         ctf_dprintf("max offset: %x\n", cup->cu_maxoff);
2552         VERIFY(cup != NULL);

2554         ret = ctf_dwarf_convert_die(cup, cup->cu_cu);
2555         ctf_dprintf("ctf_dwarf_convert_die (%s) returned %d\n", cup->cu_name,
2556             ret);
2557         if (ret != 0) {
2558                 return (ret);
2559         }
```



```
2560         if (ctf_update(cup->cu_ctfp) != 0) {
2561                 return (ctf_dwarf_error(cup, cup->cu_ctfp, 0,
2562                     "failed to update output ctf container"));
2563         }

2565         ret = ctf_dwarf_fixup_die(cup, B_FALSE);
2566         ctf_dprintf("ctf_dwarf_fixup_die (%s) returned %d\n", cup->cu_name,
2567             ret);
2568         if (ret != 0) {
2569                 return (ret);
2570         }
2571         if (ctf_update(cup->cu_ctfp) != 0) {
2572                 return (ctf_dwarf_error(cup, cup->cu_ctfp, 0,
2573                     "failed to update output ctf container"));
2574         }

2576         ret = ctf_dwarf_fixup_die(cup, B_TRUE);
2577         ctf_dprintf("ctf_dwarf_fixup_die (%s) returned %d\n", cup->cu_name,
2578             ret);
2579         if (ret != 0) {
2580                 return (ret);
2581         }
2582         if (ctf_update(cup->cu_ctfp) != 0) {
2583                 return (ctf_dwarf_error(cup, cup->cu_ctfp, 0,
2584                     "failed to update output ctf container"));
2585         }


2588         if ((ret = ctf_dwarf_conv_funcvars(cup)) != 0) {
2589                 return (ctf_dwarf_error(cup, NULL, ret,
2590                     "failed to convert strong functions and variables"));
2591         }

2593         if (ctf_update(cup->cu_ctfp) != 0) {
2594                 return (ctf_dwarf_error(cup, cup->cu_ctfp, 0,
2595                     "failed to update output ctf container"));
2596         }

2598         if (cup->cu_doweaks == B_TRUE) {
2599                 if ((ret = ctf_dwarf_conv_weaks(cup)) != 0) {
2600                         return (ctf_dwarf_error(cup, NULL, ret,
2601                             "failed to convert weak functions and variables"));
2602                 }

2604                 if (ctf_update(cup->cu_ctfp) != 0) {
2605                         return (ctf_dwarf_error(cup, cup->cu_ctfp, 0,
2606                             "failed to update output ctf container"));
2607                 }
2608         }

2610         ctf_phase_dump(cup->cu_ctfp, "pre-dwarf-dedup", cup->cu_name);
2604         ctf_phase_dump(cup->cu_ctfp, "pre-dedup");
2611         ctf_dprintf("adding inputs for dedup\n");
2612         if ((ret = ctf_merge_add(cup->cu_cmh, cup->cu_ctfp)) != 0) {
2613                 return (ctf_dwarf_error(cup, NULL, ret,
2614                     "failed to add inputs for merge"));
2615         }

2617         ctf_dprintf("starting dedup of %s\n", cup->cu_name);
2611         ctf_dprintf("starting merge\n");
2618         if ((ret = ctf_merge_dedup(cup->cu_cmh, &dedup)) != 0) {
2619                 return (ctf_dwarf_error(cup, NULL, ret,
2620                     "failed to deduplicate die"));
2621         }
2622         ctf_close(cup->cu_ctfp);
2623         cup->cu_ctfp = dedup;
```

```
2624          ctf_phase_dump(cup->cu_ctfp, "post-dwarf-dedup", cup->cu_name);

2626          return (0);
2627  }
______unchanged_portion_omitted_


2839  ctf_conv_status_t
2840  ctf_dwarf_convert(int fd, Elf *elf, uint_t nthrs, int *errp, ctf_file_t **fpp,
2841      char *errmsg, size_t errlen)
2842  {
2843          int err, ret, ndies, i;
2844          Dwarf_Debug dw;
2845          Dwarf_Error derr;
2846          ctf_cu_t *cdies = NULL, *cup;
2847          workq_t *wqp = NULL;

2849          if (errp == NULL)
2850                  errp = &err;
2851          *errp = 0;
2852          *fpp = NULL;

2854          ret = dwarf_elf_init(elf, DW_DLC_READ, NULL, NULL, &dw, &derr);
2855          if (ret != DW_DLV_OK) {
2856                  /*
2857                   * We may want to expect DWARF data here and fail conversion if
2858                   * it's missing. In this case, if we actually have some amount
2859                   * of DWARF, but no section, for now, just go ahead and create
2860                   * an empty CTF file.
2861                   */
2862                  if (ret == DW_DLV_NO_ENTRY ||
2863                      dwarf_errno(derr) == DW_DLE_DEBUG_INFO_NULL) {
2864                          *fpp = ctf_create(errp);
2865                          return (*fpp != NULL ? CTF_CONV_SUCCESS :
2866                              CTF_CONV_ERROR);
2867                  }
2868                  (void) snprintf(errmsg, errlen,
2869                      "failed to initialize DWARF: %s\n",
2870                      dwarf_errmsg(derr));
2871                  *errp = ECTF_CONVBKERR;
2872                  return (CTF_CONV_ERROR);
2873          }

2875          /*
2876           * Iterate over all of the compilation units and create a ctf_cu_t for
2877           * each of them.  This is used to determine if we have zero, one, or
2878           * multiple dies to convert. If we have zero, that's an error. If
2879           * there's only one die, that's the simple case.  No merge needed and
2880           * only a single Dwarf_Debug as well.
2881           */
2882          ndies = 0;
2883          ret = ctf_dwarf_count_dies(dw, &derr, &ndies, errmsg, errlen);
2884          if (ret != 0) {
2885                  *errp = ret;
2886                  goto out;
2887          }

2889          (void) dwarf_finish(dw, &derr);
2890          cdies = ctf_alloc(sizeof (ctf_cu_t) * ndies);
2891          if (cdies == NULL) {
2892                  *errp = ENOMEM;
2893                  return (CTF_CONV_ERROR);
2894          }

2896          for (i = 0; i < ndies; i++) {
2897                  cup = &cdies[i];
```



```
2898                  ret = dwarf_elf_init(elf, DW_DLC_READ, NULL, NULL,
2899                      &cup->cu_dwarf, &derr);
2900                  if (ret != 0) {
2901                          ctf_free(cdies, sizeof (ctf_cu_t) * ndies);
2902                          (void) snprintf(errmsg, errlen,
2903                              "failed to initialize DWARF: %s\n",
2904                              dwarf_errmsg(derr));
2905                          *errp = ECTF_CONVBKERR;
2906                          return (CTF_CONV_ERROR);
2907                  }

2909                  ret = ctf_dwarf_init_die(fd, elf, &cdies[i], i, errmsg, errlen);
2910                  if (ret != 0) {
2911                          *errp = ret;
2912                          goto out;
2913                  }

2915                  cup->cu_doweaks = ndies > 1 ? B_FALSE : B_TRUE;
2916          }

2918          ctf_dprintf("found %d DWARF CUs\n", ndies);
2910          ctf_dprintf("found %d DWARF die(s)\n", ndies);

2920          /*
2921           * If we only have one compilation unit, there's no reason to use
2922           * multiple threads, even if the user requested them. After all, they
2923           * just gave us an upper bound.
2924           */
2925          if (ndies == 1)
2926                  nthrs = 1;

2928          if (workq_init(&wqp, nthrs) == -1) {
2929                  *errp = errno;
2930                  goto out;
2931          }

2933          for (i = 0; i < ndies; i++) {
2934                  cup = &cdies[i];
2935                  ctf_dprintf("adding cu %s: %p, %x %x\n", cup->cu_name,
2927                  ctf_dprintf("adding die %s: %p, %x %x\n", cup->cu_name,
2936                      cup->cu_cu, cup->cu_cuoff, cup->cu_maxoff);
2937                  if (workq_add(wqp, cup) == -1) {
2938                          *errp = errno;
2939                          goto out;
2940                  }
2941          }

2943          ret = workq_work(wqp, ctf_dwarf_convert_one, NULL, errp);
2944          if (ret == WORKQ_ERROR) {
2945                  *errp = errno;
2946                  goto out;
2947          } else if (ret == WORKQ_UERROR) {
2948                  ctf_dprintf("internal convert failed: %s\n",
2949                      ctf_errmsg(*errp));
2950                  goto out;
2951          }

2953          ctf_dprintf("Determining next phase: have %d CUs\n", ndies);
2945          ctf_dprintf("Determining next phase: have %d dies\n", ndies);
2954          if (ndies != 1) {
2955                  ctf_merge_t *cmp;

2957                  cmp = ctf_merge_init(fd, &ret);
2958                  if (cmp == NULL) {
2959                          *errp = ret;
2960                          goto out;
```

```
2961                }

2963                ctf_dprintf("setting threads\n");
2964                if ((ret = ctf_merge_set_nthreads(cmp, nthrs)) != 0) {
2965                        ctf_merge_fini(cmp);
2966                        *errp = ret;
2967                        goto out;
2968                }

2962                ctf_dprintf("adding dies\n");
2970                for (i = 0; i < ndies; i++) {
2971                        cup = &cdies[i];
2972                        ctf_dprintf("adding cu %s (%p)\n", cup->cu_name,
2973                            cup->cu_ctfp);
2974                        if ((ret = ctf_merge_add(cmp, cup->cu_ctfp)) != 0) {
2975                                ctf_merge_fini(cmp);
2976                                *errp = ret;
2977                                goto out;
2978                        }
2979                }

2981                ctf_dprintf("performing merge\n");
2982                ret = ctf_merge_merge(cmp, fpp);
2983                if (ret != 0) {
2984                        ctf_dprintf("failed merge!\n");
2985                        *fpp = NULL;
2986                        ctf_merge_fini(cmp);
2987                        *errp = ret;
2988                        goto out;
2989                }
2990                ctf_merge_fini(cmp);
2991                *errp = 0;
2992                ctf_dprintf("successfully converted!\n");
2993        } else {
2994                *errp = 0;
2995                *fpp = cdies->cu_ctfp;
2996                cdies->cu_ctfp = NULL;
2997                ctf_dprintf("successfully converted!\n");
2998        }

3000 out:
3001        workq_fini(wqp);
3002        ctf_dwarf_free_dies(cdies, ndies);
3003        return (*fpp != NULL ? CTF_CONV_SUCCESS : CTF_CONV_ERROR);
3004 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   21605 Tue Apr 23 05:24:01 2019
new/usr/src/lib/libctf/common/ctf_lib.c
10812 ctf tools shouldn't add blank labels
10813 ctf symbol mapping needs work
Reviewed by: Jerry Jelinek <jerry.jelinek@joyent.com>
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License, Version 1.0 only
 * (the "License").  You may not use this file except in compliance
 * with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2003 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */
/*
 * Copyright (c) 2019, Joyent, Inc.
 * Copyright (c) 2015, Joyent, Inc.
 */

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <libctf_impl.h>
#include <ctf_impl.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <dlfcn.h>
#include <gelf.h>
#include <zlib.h>
#include <sys/debug.h>

#ifdef _LP64
static const char *_libctf_zlib = "/usr/lib/64/libz.so.1";
#else
static const char *_libctf_zlib = "/usr/lib/libz.so.1";
#endif

static struct {
        int (*z_uncompress)(uchar_t *, ulong_t *, const uchar_t *, ulong_t);
        int (*z_initcomp)(z_stream *, int, const char *, int);
        int (*z_compress)(z_stream *, int);
        int (*z_finicomp)(z_stream *);
        const char *(*z_error)(int);
        void *z_dlp;
} zlib;
```

_____*unchanged_portion_omitted_*


```
/*
 * A utility function for folks debugging CTF conversion and merging.
 */
void
ctf_phase_dump(ctf_file_t *fp, const char *phase, const char *name)
ctf_phase_dump(ctf_file_t *fp, const char *phase)
{
        int fd;
        static char *base;
        char path[MAXPATHLEN];

        if (base == NULL && (base = getenv("LIBCTF_WRITE_PHASES")) == NULL)
                return;

        if (name == NULL)
                name = "libctf";

        (void) snprintf(path, sizeof (path), "%s/%s.%s.%d.ctf", base, name,
        (void) snprintf(path, sizeof (path), "%s/libctf.%s.%d.ctf", base,
            phase != NULL ? phase : "",
            ctf_phase);
        if ((fd = open(path, O_CREAT | O_TRUNC | O_RDWR, 0777)) < 0)
                return;
        (void) ctf_write(fp, fd);
        (void) close(fd);
}

void
ctf_phase_bump(void)
{
        ctf_phase++;
}

int
ctf_symtab_iter(ctf_file_t *fp, ctf_symtab_f func, void *arg)
{
        ulong_t i;
        uintptr_t symbase;
        uintptr_t strbase;
        const char *file = NULL;
        boolean_t primary = B_TRUE;

        if (fp->ctf_symtab.cts_data == NULL ||
            fp->ctf_strtab.cts_data == NULL) {
                return (ECTF_NOSYMTAB);
        }

        symbase = (uintptr_t)fp->ctf_symtab.cts_data;
        strbase = (uintptr_t)fp->ctf_strtab.cts_data;

        for (i = 0; i < fp->ctf_nsyms; i++) {
                const char *name;
                int ret;
                uint_t type;
                Elf64_Sym sym;

                /*
                 * The CTF library has historically tried to handle large file
                 * offsets itself so that way clients can be unaware of such
                 * isseus. Therefore, we translate everything to a 64-bit ELF
                 * symbol, this is done to make it so that the rest of the
                 * library doesn't have to know about these differences. For
                 * more information see, lib/libctf/common/ctf_lib.c.
                 */
                if (fp->ctf_symtab.cts_entsize == sizeof (Elf32_Sym)) {
                        const Elf32_Sym *symp = (Elf32_Sym *)symbase + i;
```

```
                        uint_t bind, itype;

                        sym.st_name = symp->st_name;
                        sym.st_value = symp->st_value;
                        sym.st_size = symp->st_size;
                        bind = ELF32_ST_BIND(symp->st_info);
                        itype = ELF32_ST_TYPE(symp->st_info);
                        sym.st_info = ELF64_ST_INFO(bind, itype);
                        sym.st_other = symp->st_other;
                        sym.st_shndx = symp->st_shndx;
                } else {
                        const Elf64_Sym *symp = (Elf64_Sym *)symbase + i;

                        sym = *symp;
                }

                type = ELF64_ST_TYPE(sym.st_info);
                name = (const char *)(strbase + sym.st_name);

                /*
                 * Check first if we have an STT_FILE entry. This is used to
                 * distinguish between various local symbols when merging.
                 */
                if (type == STT_FILE) {
                        if (file != NULL) {
                                primary = B_FALSE;
                        }
                        file = name;
                        continue;
                }

                /*
                 * Check if this is a symbol that we care about.
                 */
                if (!ctf_sym_valid(strbase, type, sym.st_shndx, sym.st_value,
                    sym.st_name)) {
                        continue;
                }

                if ((ret = func(&sym, i, file, name, primary, arg)) != 0) {
                        return (ret);
                }
        }

        return (0);
}
```

*unchanged_portion_omitted_*

```
**********************************************************
   43216 Tue Apr 23 05:24:02 2019
new/usr/src/lib/libctf/common/ctf_merge.c
10812 ctf tools shouldn't add blank labels
10813 ctf symbol mapping needs work
Reviewed by: Jerry Jelinek <jerry.jelinek@joyent.com>
**********************************************************
```

```
/*
 * This file and its contents are supplied under the terms of the
 * Common Development and Distribution License ("CDDL"), version 1.0.
 * You may only use this file in accordance with the terms of version
 * 1.0 of the CDDL.
 *
 * A full copy of the text of the CDDL should have accompanied this
 * source.  A copy of the CDDL is also available via the Internet at
 * http://www.illumos.org/license/CDDL.
 */

/*
 * Copyright (c) 2019 Joyent, Inc.
 * Copyright (c) 2015 Joyent, Inc.
 */

/*
 * To perform a merge of two CTF containers, we first diff the two containers
 * types. For every type that's in the src container, but not in the dst
 * container, we note it and add it to dst container. If there are any objects
 * or functions associated with src, we go through and update the types that
 * they refer to such that they all refer to types in the dst container.
 *
 * The bulk of the logic for the merge, after we've run the diff, occurs in
 * ctf_merge_common().
 *
 * In terms of exported APIs, we don't really export a simple merge two
 * containers, as the general way this is used, in something like ctfmerge(1),
 * is to add all the containers and then let us figure out the best way to merge
 * it.
 */

#include <libctf_impl.h>
#include <sys/debug.h>
#include <sys/list.h>
#include <stddef.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <mergeq.h>
#include <errno.h>

typedef struct ctf_merge_tinfo {
        uint16_t cmt_map;       /* Map to the type in out */
        boolean_t cmt_fixup;
        boolean_t cmt_forward;
        boolean_t cmt_missing;
} ctf_merge_tinfo_t;
______unchanged_portion_omitted_

typedef struct ctf_merge_objmap {
        list_node_t cmo_node;
        const char *cmo_name;           /* Symbol name */
        const char *cmo_file;           /* Symbol file */
        ulong_t cmo_idx;                /* Symbol ID */
        Elf64_Sym cmo_sym;              /* Symbol Entry */
        ctf_id_t cmo_tid;               /* Type ID */
} ctf_merge_objmap_t;
```



```
typedef struct ctf_merge_funcmap {
        list_node_t cmf_node;
        const char *cmf_name;           /* Symbol name */
        const char *cmf_file;           /* Symbol file */
        ulong_t cmf_idx;                /* Symbol ID */
        Elf64_Sym cmf_sym;              /* Symbol Entry */
        ctf_id_t cmf_rtid;              /* Type ID */
        uint_t cmf_flags;               /* ctf_funcinfo_t ctc_flags */
        uint_t cmf_argc;                /* Number of arguments */
        ctf_id_t cmf_args[];            /* Types of arguments */
} ctf_merge_funcmap_t;
______unchanged_portion_omitted_

typedef struct ctf_merge_symbol_arg {
        list_t *cmsa_objmap;
        list_t *cmsa_funcmap;
        ctf_file_t *cmsa_out;
        boolean_t cmsa_dedup;
} ctf_merge_symbol_arg_t;

static int ctf_merge_add_type(ctf_merge_types_t *, ctf_id_t);

static ctf_id_t
ctf_merge_gettype(ctf_merge_types_t *cmp, ctf_id_t id)
{
        if (cmp->cm_dedup == B_FALSE) {
                VERIFY(cmp->cm_tmap[id].cmt_map != 0);
                return (cmp->cm_tmap[id].cmt_map);
        }

        while (cmp->cm_tmap[id].cmt_missing == B_FALSE) {
                VERIFY(cmp->cm_tmap[id].cmt_map != 0);
                id = cmp->cm_tmap[id].cmt_map;
        }
        VERIFY(cmp->cm_tmap[id].cmt_map != 0);
        return (cmp->cm_tmap[id].cmt_map);
}
______unchanged_portion_omitted_


/*
 * We're going to do three passes over the containers.
 *
 * Pass 1 checks for forward references in the output container that we know
 * exist in the source container.
 *
 * Pass 2 adds all the missing types from the source container. As part of this
 * we may be adding a type as a forward reference that doesn't exist yet.
 * Any types that we encounter in this form, we need to add to a third pass.
 *
 * Pass 3 is the fixup pass. Here we go through and find all the types that were
 * missing in the first.
 *
 * Importantly, we *must* call ctf_update between the second and third pass,
 * otherwise several of the libctf functions will not properly find the data in
 * the container. If we're doing a dedup we also fix up the type mapping.
 */
static int
ctf_merge_common(ctf_merge_types_t *cmp)
{
        int ret, i;

        ctf_phase_dump(cmp->cm_src, "merge-common-src", NULL);
        ctf_phase_dump(cmp->cm_out, "merge-common-dest", NULL);
        ctf_phase_dump(cmp->cm_src, "merge-common-src");
        ctf_phase_dump(cmp->cm_out, "merge-common-dest");
```

```
678         /* Pass 1 */
679         for (i = 1; i <= cmp->cm_src->ctf_typemax; i++) {
680                 if (cmp->cm_tmap[i].cmt_forward == B_TRUE) {
681                         ret = ctf_merge_add_sou(cmp, i, B_TRUE);
682                         if (ret != 0) {
683                                 return (ret);
684                         }
685                 }
686         }
687
688         /* Pass 2 */
689         for (i = 1; i <= cmp->cm_src->ctf_typemax; i++) {
690                 if (cmp->cm_tmap[i].cmt_missing == B_TRUE) {
691                         ret = ctf_merge_add_type(cmp, i);
692                         if (ret != 0) {
693                                 ctf_dprintf("Failed to merge type %d\n", i);
694                                 return (ret);
695                         }
696                 }
697         }
698
699         ret = ctf_update(cmp->cm_out);
700         if (ret != 0)
701                 return (ret);
702
703         if (cmp->cm_dedup == B_TRUE) {
704                 ctf_merge_fixup_dedup_map(cmp);
705         }
706
707         ctf_dprintf("Beginning merge pass 3\n");
708         /* Pass 3 */
709         for (i = 1; i <= cmp->cm_src->ctf_typemax; i++) {
710                 if (cmp->cm_tmap[i].cmt_fixup == B_TRUE) {
711                         ret = ctf_merge_fixup_type(cmp, i);
712                         if (ret != 0)
713                                 return (ret);
714                 }
715         }
716
717         if (cmp->cm_dedup == B_TRUE) {
718                 ctf_merge_fixup_dedup_map(cmp);
719         }
720
721         return (0);
722 }
```

*unchanged_portion_omitted_*

```
777 /*
778  * After performing a pass, we need to go through the object and function type
779  * maps and potentially fix them up based on the new maps that we have.
780  */
781 static void
782 ctf_merge_fixup_symmaps(ctf_merge_types_t *cmp, ctf_merge_input_t *cmi)
783 {
784         ctf_merge_objmap_t *cmo;
785         ctf_merge_funcmap_t *cmf;
786
787         for (cmo = list_head(&cmi->cmi_omap); cmo != NULL;
788             cmo = list_next(&cmi->cmi_omap, cmo)) {
789                 VERIFY3S(cmo->cmo_tid, !=, 0);
790                 VERIFY(cmp->cm_tmap[cmo->cmo_tid].cmt_map != 0);
791                 cmo->cmo_tid = cmp->cm_tmap[cmo->cmo_tid].cmt_map;
792         }
793
794         for (cmf = list_head(&cmi->cmi_fmap); cmf != NULL;
```



```
795             cmf = list_next(&cmi->cmi_fmap, cmf)) {
796                 int i;
797
798                 VERIFY(cmp->cm_tmap[cmf->cmf_rtid].cmt_map != 0);
799                 cmf->cmf_rtid = cmp->cm_tmap[cmf->cmf_rtid].cmt_map;
800                 for (i = 0; i < cmf->cmf_argc; i++) {
801                         VERIFY(cmp->cm_tmap[cmf->cmf_args[i]].cmt_map != 0);
802                         cmf->cmf_args[i] =
803                             cmp->cm_tmap[cmf->cmf_args[i]].cmt_map;
804                 }
805         }
806 }
807
808 /*
809  * Merge the types contained inside of two input files. The second input file is
810  * always going to be the destination. We're guaranteed that it's always
811  * writeable.
812  */
813 static int
814 ctf_merge_types(void *arg, void *arg2, void **outp, void *unsued)
815 {
816         int ret;
817         ctf_merge_types_t cm;
818         ctf_diff_t *cdp;
777         ctf_merge_objmap_t *cmo;
778         ctf_merge_funcmap_t *cmf;
819         ctf_merge_input_t *scmi = arg;
820         ctf_merge_input_t *dcmi = arg2;
821         ctf_file_t *out = dcmi->cmi_input;
822         ctf_file_t *source = scmi->cmi_input;
823
824         ctf_dprintf("merging %p->%p\n", source, out);
825
826         if (!(out->ctf_flags & LCTF_RDWR))
827                 return (ctf_set_errno(out, ECTF_RDONLY));
828
829         if (ctf_getmodel(out) != ctf_getmodel(source))
830                 return (ctf_set_errno(out, ECTF_DMODEL));
831
832         if ((ret = ctf_diff_init(out, source, &cdp)) != 0)
833                 return (ret);
834
835         cm.cm_out = out;
836         cm.cm_src = source;
837         cm.cm_dedup = B_FALSE;
838         cm.cm_unique = B_FALSE;
839         ret = ctf_merge_types_init(&cm);
840         if (ret != 0) {
841                 ctf_diff_fini(cdp);
842                 return (ctf_set_errno(out, ret));
843         }
844
845         ret = ctf_diff_types(cdp, ctf_merge_diffcb, &cm);
846         if (ret != 0)
847                 goto cleanup;
848         ret = ctf_merge_common(&cm);
849         ctf_dprintf("merge common returned with %d\n", ret);
850         if (ret == 0) {
851                 ret = ctf_update(out);
852                 ctf_dprintf("update returned with %d\n", ret);
853         } else {
854                 goto cleanup;
855         }
856
857         /*
858          * Now we need to fix up the object and function maps.
```

```
859          */
860         ctf_merge_fixup_symmaps(&cm, scmi);
820         for (cmo = list_head(&scmi->cmi_omap); cmo != NULL;
821             cmo = list_next(&scmi->cmi_omap, cmo)) {
822                 if (cmo->cmo_tid == 0)
823                         continue;
824                 VERIFY(cm.cm_tmap[cmo->cmo_tid].cmt_map != 0);
825                 cmo->cmo_tid = cm.cm_tmap[cmo->cmo_tid].cmt_map;
826         }

828         for (cmf = list_head(&scmi->cmi_fmap); cmf != NULL;
829             cmf = list_next(&scmi->cmi_fmap, cmf)) {
830                 int i;

832                 VERIFY(cm.cm_tmap[cmf->cmf_rtid].cmt_map != 0);
833                 cmf->cmf_rtid = cm.cm_tmap[cmf->cmf_rtid].cmt_map;
834                 for (i = 0; i < cmf->cmf_argc; i++) {
835                         VERIFY(cm.cm_tmap[cmf->cmf_args[i]].cmt_map != 0);
836                         cmf->cmf_args[i] = cm.cm_tmap[cmf->cmf_args[i]].cmt_map;
837                 }
838         }

862         /*
863          * Now that we've fixed things up, we need to give our function and
864          * object maps to the destination, such that it can continue to update
865          * them going forward.
866          */
867         list_move_tail(&dcmi->cmi_fmap, &scmi->cmi_fmap);
868         list_move_tail(&dcmi->cmi_omap, &scmi->cmi_omap);

870 cleanup:
871         if (ret == 0)
872                 *outp = dcmi;
873         ctf_merge_types_fini(&cm);
874         ctf_diff_fini(cdp);
875         if (ret != 0)
876                 return (ctf_errno(out));
877         ctf_phase_bump();
878         return (0);
879 }

858 /*
859  * After performing a pass, we need to go through the object and function type
860  * maps and potentially fix them up based on the new maps that we haev.
861  */
862 static void
863 ctf_merge_fixup_nontypes(ctf_merge_types_t *cmp, ctf_merge_input_t *cmi)
864 {
865         ctf_merge_objmap_t *cmo;
866         ctf_merge_funcmap_t *cmf;

868         for (cmo = list_head(&cmi->cmi_omap); cmo != NULL;
869             cmo = list_next(&cmi->cmi_omap, cmo)) {
870                 if (cmo->cmo_tid == 0)
871                         continue;
872                 VERIFY(cmp->cm_tmap[cmo->cmo_tid].cmt_map != 0);
873                 cmo->cmo_tid = cmp->cm_tmap[cmo->cmo_tid].cmt_map;
874         }

876         for (cmf = list_head(&cmi->cmi_fmap); cmf != NULL;
877             cmf = list_next(&cmi->cmi_fmap, cmf)) {
878                 int i;

880                 VERIFY(cmp->cm_tmap[cmf->cmf_rtid].cmt_map != 0);
881                 cmf->cmf_rtid = cmp->cm_tmap[cmf->cmf_rtid].cmt_map;
882                 for (i = 0; i < cmf->cmf_argc; i++) {
```



```
883                         VERIFY(cmp->cm_tmap[cmf->cmf_args[i]].cmt_map !=
884                             0);
885                         cmf->cmf_args[i] =
886                             cmp->cm_tmap[cmf->cmf_args[i]].cmt_map;
887                 }
888         }
889 }

881 static int
882 ctf_uniquify_types(ctf_merge_t *cmh, ctf_file_t *src, ctf_file_t **outp)
883 {
884         int err, ret;
885         ctf_file_t *out;
886         ctf_merge_types_t cm;
887         ctf_diff_t *cdp;
888         ctf_merge_input_t *cmi;
889         ctf_file_t *parent = cmh->cmh_unique;

891         *outp = NULL;
892         out = ctf_fdcreate(cmh->cmh_ofd, &err);
893         if (out == NULL)
894                 return (ctf_set_errno(src, err));

896         out->ctf_parname = cmh->cmh_pname;
897         if (ctf_setmodel(out, ctf_getmodel(parent)) != 0) {
898                 (void) ctf_set_errno(src, ctf_errno(out));
899                 ctf_close(out);
900                 return (CTF_ERR);
901         }

903         if (ctf_import(out, parent) != 0) {
904                 (void) ctf_set_errno(src, ctf_errno(out));
905                 ctf_close(out);
906                 return (CTF_ERR);
907         }

909         if ((ret = ctf_diff_init(parent, src, &cdp)) != 0) {
910                 ctf_close(out);
911                 return (ctf_set_errno(src, ctf_errno(parent)));
912         }

914         cm.cm_out = parent;
915         cm.cm_src = src;
916         cm.cm_dedup = B_FALSE;
917         cm.cm_unique = B_TRUE;
918         ret = ctf_merge_types_init(&cm);
919         if (ret != 0) {
920                 ctf_close(out);
921                 ctf_diff_fini(cdp);
922                 return (ctf_set_errno(src, ret));
923         }

925         ret = ctf_diff_types(cdp, ctf_merge_diffcb, &cm);
926         if (ret == 0) {
927                 cm.cm_out = out;
928                 ret = ctf_merge_uniquify_types(&cm);
929                 if (ret == 0)
930                         ret = ctf_update(out);
931         }

933         if (ret != 0) {
934                 ctf_merge_types_fini(&cm);
935                 ctf_diff_fini(cdp);
936                 return (ctf_set_errno(src, ctf_errno(cm.cm_out)));
937         }
```

```
 939         for (cmi = list_head(&cmh->cmh_inputs); cmi != NULL;
 940             cmi = list_next(&cmh->cmh_inputs, cmi)) {
 941                 ctf_merge_fixup_symmaps(&cm, cmi);
 951                 ctf_merge_fixup_nontypes(&cm, cmi);
 942         }

 944         ctf_merge_types_fini(&cm);
 945         ctf_diff_fini(cdp);
 946         *outp = out;
 947         return (0);
 948 }
_____unchanged_portion_omitted_

1052 static int
1053 ctf_merge_add_function(ctf_merge_input_t *cmi, ctf_funcinfo_t *fip, ulong_t idx,
1054     const char *file, const char *name, const Elf64_Sym *symp)
1063 ctf_merge_add_funcs_cb(const char *name, ulong_t idx, ctf_funcinfo_t *fip,
1064     void *arg)
1055 {
1066         ctf_merge_input_t *cmi = arg;
1056         ctf_merge_funcmap_t *fmap;

1058         fmap = ctf_alloc(sizeof (ctf_merge_funcmap_t) +
1059             sizeof (ctf_id_t) * fip->ctc_argc);
1060         if (fmap == NULL)
1061                 return (ENOMEM);

1063         fmap->cmf_idx = idx;
1064         fmap->cmf_sym = *symp;
1065         fmap->cmf_rtid = fip->ctc_return;
1066         fmap->cmf_flags = fip->ctc_flags;
1067         fmap->cmf_argc = fip->ctc_argc;
1068         fmap->cmf_name = name;
1069         if (ELF64_ST_BIND(symp->st_info) == STB_LOCAL) {
1070                 fmap->cmf_file = file;
1071         } else {
1072                 fmap->cmf_file = NULL;
1073         }

1075         if (ctf_func_args(cmi->cmi_input, idx, fmap->cmf_argc,
1076             fmap->cmf_args) != 0) {
1077                 ctf_free(fmap, sizeof (ctf_merge_funcmap_t) +
1078                     sizeof (ctf_id_t) * fip->ctc_argc);
1079                 return (ctf_errno(cmi->cmi_input));
1080         }

1082         ctf_dprintf("added initial function %s, %lu, %s %u\n", name, idx,
1083             fmap->cmf_file != NULL ? fmap->cmf_file : "global",
1084             ELF64_ST_BIND(symp->st_info));
1085         list_insert_tail(&cmi->cmi_fmap, fmap);
1086         return (0);
1087 }

1089 static int
1090 ctf_merge_add_object(ctf_merge_input_t *cmi, ctf_id_t id, ulong_t idx,
1091     const char *file, const char *name, const Elf64_Sym *symp)
1092 ctf_merge_add_objs_cb(const char *name, ctf_id_t id, ulong_t idx, void *arg)
1092 {
1094         ctf_merge_input_t *cmi = arg;
1093         ctf_merge_objmap_t *cmo;

1095         cmo = ctf_alloc(sizeof (ctf_merge_objmap_t));
1096         if (cmo == NULL)
1097                 return (ENOMEM);

1099         cmo->cmo_name = name;
```



```
1100         if (ELF64_ST_BIND(symp->st_info) == STB_LOCAL) {
1101                 cmo->cmo_file = file;
1102         } else {
1103                 cmo->cmo_file = NULL;
1104         }
1105         cmo->cmo_idx = idx;
1106         cmo->cmo_tid = id;
1107         cmo->cmo_sym = *symp;
1108         list_insert_tail(&cmi->cmi_omap, cmo);

1110         ctf_dprintf("added initial object %s, %lu, %ld, %s\n", name, idx, id,
1111             cmo->cmo_file != NULL ? cmo->cmo_file : "global");

1113         return (0);
1114 }

1116 static int
1117 ctf_merge_add_symbol(const Elf64_Sym *symp, ulong_t idx, const char *file,
1118     const char *name, boolean_t primary, void *arg)
1119 {
1120         ctf_merge_input_t *cmi = arg;
1121         ctf_file_t *fp = cmi->cmi_input;
1122         ushort_t *data, funcbase;
1123         uint_t type;
1124         ctf_funcinfo_t fi;

1126         /*
1127          * See if there is type information for this. If there is no
1128          * type information for this entry or no translation, then we
1129          * will find the value zero. This indicates no type ID for
1130          * objects and encodes unknown information for functions.
1131          */
1132         if (fp->ctf_sxlate[idx] == -1u)
1133                 return (0);
1134         data = (ushort_t *)((uintptr_t)fp->ctf_buf + fp->ctf_sxlate[idx]);
1135         if (*data == 0)
1136                 return (0);

1138         type = ELF64_ST_TYPE(symp->st_info);

1140         switch (type) {
1141         case STT_FUNC:
1142                 funcbase = *data;
1143                 if (LCTF_INFO_KIND(fp, funcbase) != CTF_K_FUNCTION)
1144                         return (0);
1145                 data++;
1146                 fi.ctc_return = *data;
1147                 data++;
1148                 fi.ctc_argc = LCTF_INFO_VLEN(fp, funcbase);
1149                 fi.ctc_flags = 0;

1151                 if (fi.ctc_argc != 0 && data[fi.ctc_argc - 1] == 0) {
1152                         fi.ctc_flags |= CTF_FUNC_VARARG;
1153                         fi.ctc_argc--;
1154                 }
1155                 return (ctf_merge_add_function(cmi, &fi, idx, file, name,
1156                     symp));
1157         case STT_OBJECT:
1158                 return (ctf_merge_add_object(cmi, *data, idx, file, name,
1159                     symp));
1160         default:
1161                 return (0);
1162         }
1163 }

1165 /*
```

```
 * Whenever we create an entry to merge, we then go and add a second empty
 * ctf_file_t which we use for the purposes of our merging. It's not the best,
 * but it's the best that we've got at the moment.
 */
int
ctf_merge_add(ctf_merge_t *cmh, ctf_file_t *input)
{
        int ret;
        ctf_merge_input_t *cmi;
        ctf_file_t *empty;

        ctf_dprintf("adding input %p\n", input);

        if (input->ctf_flags & LCTF_CHILD)
                return (ECTF_MCHILD);

        cmi = ctf_alloc(sizeof (ctf_merge_input_t));
        if (cmi == NULL)
                return (ENOMEM);

        cmi->cmi_created = B_FALSE;
        cmi->cmi_input = input;
        list_create(&cmi->cmi_fmap, sizeof (ctf_merge_funcmap_t),
            offsetof(ctf_merge_funcmap_t, cmf_node));
        list_create(&cmi->cmi_omap, sizeof (ctf_merge_funcmap_t),
            offsetof(ctf_merge_objmap_t, cmo_node));

        if (cmh->cmh_msyms == B_TRUE) {
                if ((ret = ctf_symtab_iter(input, ctf_merge_add_symbol,
                if ((ret = ctf_function_iter(input, ctf_merge_add_funcs_cb,
                    cmi)) != 0) {
                        ctf_merge_fini_input(cmi);
                        return (ret);
                }

                if ((ret = ctf_object_iter(input, ctf_merge_add_objs_cb,
                    cmi)) != 0) {
                        ctf_merge_fini_input(cmi);
                        return (ret);
        }
        }

        list_insert_tail(&cmh->cmh_inputs, cmi);
        cmh->cmh_ninputs++;

        /* And now the empty one to merge into this */
        cmi = ctf_alloc(sizeof (ctf_merge_input_t));
        if (cmi == NULL)
                return (ENOMEM);
        list_create(&cmi->cmi_fmap, sizeof (ctf_merge_funcmap_t),
            offsetof(ctf_merge_funcmap_t, cmf_node));
        list_create(&cmi->cmi_omap, sizeof (ctf_merge_funcmap_t),
            offsetof(ctf_merge_objmap_t, cmo_node));

        empty = ctf_fdcreate(cmh->cmh_ofd, &ret);
        if (empty == NULL)
                return (ret);
        cmi->cmi_input = empty;
        cmi->cmi_created = B_TRUE;

        if (ctf_setmodel(empty, ctf_getmodel(input)) == CTF_ERR) {
                return (ctf_errno(empty));
        }

        list_insert_tail(&cmh->cmh_inputs, cmi);
        cmh->cmh_ninputs++;
```



```
        ctf_dprintf("added containers %p and %p\n", input, empty);
        return (0);
}
```

*unchanged_portion_omitted_*

```
/*
 * Symbol matching rules: the purpose of this is to verify that the type
 * information that we have for a given symbol actually matches the output
 * symbol. This is unfortunately complicated by several different factors:
 *
 * 1. When merging multiple .o's into a single item, the symbol table index will
 * not match.
 *
 * 2. Visibility of a symbol may not be identical to the object file or the
 * DWARF information due to symbol reduction via a mapfile.
 *
 * As such, we have to employ the following rules:
 *
 * 1. A global symbol table entry always matches a global CTF symbol with the
 * same name.
 *
 * 2. A local symbol table entry always matches a local CTF symbol if they have
 * the same name and they belong to the same file.
 *
 * 3. A weak symbol matches a non-weak symbol. This happens if we find that the
 * types match, the values match, the sizes match, and the section indexes
 * match. This happens when we do a conversion in one pass, it almost never
 * happens when we're merging multiple object files. If we match a CTF global
 * symbol, that's a fixed match, otherwise it's a fuzzy match.
 *
 * 4. A local symbol table entry matches a global CTF entry if the
 * other pieces fail, but they have the same name. This is considered a fuzzy
 * match and is not used unless we have no other options.
 *
 * 5. A weak symbol table entry matches a weak CTF entry if the other pieces
 * fail, but they have the same name. This is considered a fuzzy match and is
 * not used unless we have no other options. When merging independent .o files,
 * this is often the only recourse we have to matching weak symbols.
 *
 * In the end, this would all be much simpler if we were able to do this as part
 * of libld which would be able to do all the symbol transformations.
 */
static boolean_t
ctf_merge_symbol_match(const char *ctf_file, const char *ctf_name,
    const Elf64_Sym *ctf_symp, const char *symtab_file, const char *symtab_name,
    const Elf64_Sym *symtab_symp, boolean_t *is_fuzzy)
static int
ctf_merge_symbols(ctf_merge_t *cmh, ctf_file_t *fp)
{
        *is_fuzzy = B_FALSE;
        uint_t symtab_bind, ctf_bind;
        int err;
        ulong_t i;

        symtab_bind = ELF64_ST_BIND(symtab_symp->st_info);
        ctf_bind = ELF64_ST_BIND(ctf_symp->st_info);
        uintptr_t symbase = (uintptr_t)fp->ctf_symtab.cts_data;
        uintptr_t strbase = (uintptr_t)fp->ctf_strtab.cts_data;

        ctf_dprintf("comparing merge match for %s/%s/%u->%s/%s/%u\n",
            symtab_file, symtab_name, symtab_bind,
            ctf_file, ctf_name, ctf_bind);
        if (strcmp(ctf_name, symtab_name) != 0) {
                return (B_FALSE);
        }
        for (i = 0; i < fp->ctf_nsyms; i++) {
```

```
1207                const char *name;
1208                ctf_merge_input_t *cmi;
1209                ctf_merge_objmap_t *cmo;

1305        if (symtab_bind == STB_GLOBAL && ctf_bind == STB_GLOBAL) {
1306                return (B_TRUE);
1307        } else if (symtab_bind == STB_GLOBAL) {
1308                return (B_FALSE);
1211                if (fp->ctf_symtab.cts_entsize == sizeof (Elf32_Sym)) {
1212                        const Elf32_Sym *symp = (Elf32_Sym *)symbase + i;
1213                        int type = ELF32_ST_TYPE(symp->st_info);
1214                        if (type != STT_OBJECT)
1215                                continue;
1216                        if (ctf_sym_valid(strbase, type, symp->st_shndx,
1217                            symp->st_value, symp->st_name) == B_FALSE)
1218                                continue;
1219                        name = (char *)(strbase + symp->st_name);
1220                } else {
1221                        const Elf64_Sym *symp = (Elf64_Sym *)symbase + i;
1222                        int type = ELF64_ST_TYPE(symp->st_info);
1223                        if (type != STT_OBJECT)
1224                                continue;
1225                        if (ctf_sym_valid(strbase, type, symp->st_shndx,
1226                            symp->st_value, symp->st_name) == B_FALSE)
1227                                continue;
1228                        name = (char *)(strbase + symp->st_name);
1309        }

1311        if (ctf_bind == STB_LOCAL && ctf_bind == symtab_bind &&
1312            ctf_file != NULL && symtab_file != NULL &&
1313            strcmp(ctf_file, symtab_file) == 0) {
1314                return (B_TRUE);
1231                cmo = NULL;
1232                for (cmi = list_head(&cmh->cmh_inputs); cmi != NULL;
1233                    cmi = list_next(&cmh->cmh_inputs, cmi)) {
1234                        for (cmo = list_head(&cmi->cmi_omap); cmo != NULL;
1235                            cmo = list_next(&cmi->cmi_omap, cmo)) {
1236                                if (strcmp(cmo->cmo_name, name) == 0)
1237                                        goto found;
1315        }

1317        if (symtab_bind == STB_WEAK && ctf_bind != STB_WEAK &&
1318            ELF64_ST_TYPE(symtab_symp->st_info) ==
1319            ELF64_ST_TYPE(ctf_symp->st_info) &&
1320            symtab_symp->st_value == ctf_symp->st_value &&
1321            symtab_symp->st_size == ctf_symp->st_size &&
1322            symtab_symp->st_shndx == ctf_symp->st_shndx) {
1323                if (ctf_bind == STB_GLOBAL) {
1324                        return (B_TRUE);
1325                }

1327                if (ctf_bind == STB_LOCAL && ctf_file != NULL &&
1328                    symtab_file != NULL && strcmp(ctf_file, symtab_file) == 0) {
1329                        *is_fuzzy = B_TRUE;
1330                        return (B_TRUE);
1240 found:
1241                if (cmo != NULL) {
1242                        if (cmo->cmo_tid == 0)
1243                                continue;
1244                        if ((err = ctf_add_object(fp, i, cmo->cmo_tid)) != 0) {
1245                                ctf_dprintf("Failed to add symbol %s->%d: %s\n",
1246                                    name, cmo->cmo_tid,
1247                                    ctf_errmsg(ctf_errno(fp)));
1248                                return (err);
1331                }
1332        }
```



```
1334        if (ctf_bind == STB_GLOBAL ||
1335            (ctf_bind == STB_WEAK && symtab_bind == STB_WEAK)) {
1336                *is_fuzzy = B_TRUE;
1337                return (B_TRUE);
1338        }

1340        return (B_FALSE);
1253        return (0);
1341 }

1343 /*
1344  * For each symbol, try and find a match. We will attempt to find an exact
1345  * match; however, we will settle for a fuzzy match in general. There is one
1346  * case where we will not opt to use a fuzzy match, which is when performing the
1347  * deduplication of a container. In such a case we are trying to reduce common
1348  * types and a fuzzy match would be inappropriate as if we're in the context of
1349  * a single container, the conversion process should have identified any exact
1350  * or fuzzy matches that were required.
1351  */
1352 static int
1353 ctf_merge_symbols(const Elf64_Sym *symp, ulong_t idx, const char *file,
1354     const char *name, boolean_t primary, void *arg)
1257 ctf_merge_functions(ctf_merge_t *cmh, ctf_file_t *fp)
1355 {
1356        int err;
1357        uint_t type, bind;
1358        ctf_merge_symbol_arg_t *csa = arg;
1359        ctf_file_t *fp = csa->cmsa_out;
1260        ulong_t i;
1261        ctf_funcinfo_t fi;

1361        type = ELF64_ST_TYPE(symp->st_info);
1362        bind = ELF64_ST_BIND(symp->st_info);
1263        uintptr_t symbase = (uintptr_t)fp->ctf_symtab.cts_data;
1264        uintptr_t strbase = (uintptr_t)fp->ctf_strtab.cts_data;

1364        ctf_dprintf("Trying to find match for %s/%s/%u\n", file, name,
1365            ELF64_ST_BIND(symp->st_info));
1266        for (i = 0; i < fp->ctf_nsyms; i++) {
1267                const char *name;
1268                ctf_merge_input_t *cmi;
1269                ctf_merge_funcmap_t *cmf;

1367        if (type == STT_OBJECT) {
1368                ctf_merge_objmap_t *cmo, *match = NULL;

1370                for (cmo = list_head(csa->cmsa_objmap); cmo != NULL;
1371                    cmo = list_next(csa->cmsa_objmap, cmo)) {
1372                        boolean_t is_fuzzy = B_FALSE;
1373                        if (ctf_merge_symbol_match(cmo->cmo_file, cmo->cmo_name,
1374                            &cmo->cmo_sym, file, name, symp, &is_fuzzy)) {
1375                                if (is_fuzzy && csa->cmsa_dedup &&
1376                                    bind != STB_WEAK) {
1271                if (fp->ctf_symtab.cts_entsize == sizeof (Elf32_Sym)) {
1272                        const Elf32_Sym *symp = (Elf32_Sym *)symbase + i;
1273                        int type = ELF32_ST_TYPE(symp->st_info);
1274                        if (ELF32_ST_TYPE(symp->st_info) != STT_FUNC)
1377                                        continue;
1378                                }
1379                                match = cmo;
1380                                if (is_fuzzy) {
1276                        if (ctf_sym_valid(strbase, type, symp->st_shndx,
1277                            symp->st_value, symp->st_name) == B_FALSE)
1381                                        continue;
1382                                }
```

```
                                break;
                        }
                }

                if (match == NULL) {
                        return (0);
                }

                if ((err = ctf_add_object(fp, idx, match->cmo_tid)) != 0) {
                        ctf_dprintf("Failed to add symbol %s->%d: %s\n", name,
                            match->cmo_tid, ctf_errmsg(ctf_errno(fp)));
                        return (ctf_errno(fp));
                }
                ctf_dprintf("mapped object into output %s/%s->%ld\n", file,
                    name, match->cmo_tid);
                        name = (char *)(strbase + symp->st_name);
        } else {
                ctf_merge_funcmap_t *cmf, *match = NULL;
                ctf_funcinfo_t fi;

                for (cmf = list_head(csa->cmsa_funcmap); cmf != NULL;
                    cmf = list_next(csa->cmsa_funcmap, cmf)) {
                        boolean_t is_fuzzy = B_FALSE;
                        if (ctf_merge_symbol_match(cmf->cmf_file, cmf->cmf_name,
                            &cmf->cmf_sym, file, name, symp, &is_fuzzy)) {
                                if (is_fuzzy && csa->cmsa_dedup &&
                                    bind != STB_WEAK) {
                        const Elf64_Sym *symp = (Elf64_Sym *)symbase + i;
                        int type = ELF64_ST_TYPE(symp->st_info);
                        if (ELF64_ST_TYPE(symp->st_info) != STT_FUNC)
                                        continue;
                                }
                                match = cmf;
                                if (is_fuzzy) {
                        if (ctf_sym_valid(strbase, type, symp->st_shndx,
                            symp->st_value, symp->st_name) == B_FALSE)
                                        continue;
                        name = (char *)(strbase + symp->st_name);
                                }
                                break;
                        }
                }

                if (match == NULL) {
                        return (0);
                cmf = NULL;
                for (cmi = list_head(&cmh->cmh_inputs); cmi != NULL;
                    cmi = list_next(&cmh->cmh_inputs, cmi)) {
                        for (cmf = list_head(&cmi->cmi_fmap); cmf != NULL;
                            cmf = list_next(&cmi->cmi_fmap, cmf)) {
                                if (strcmp(cmf->cmf_name, name) == 0)
                                        goto found;
                }

                fi.ctc_return = match->cmf_rtid;
                fi.ctc_argc = match->cmf_argc;
                fi.ctc_flags = match->cmf_flags;
                if ((err = ctf_add_function(fp, idx, &fi, match->cmf_args)) !=
                    0) {
                        ctf_dprintf("Failed to add function %s: %s\n", name,
                            ctf_errmsg(ctf_errno(fp)));
                        return (ctf_errno(fp));
                }
                ctf_dprintf("mapped function into output %s/%s\n", file,
                    name);
found:
```



```
                if (cmf != NULL) {
                        fi.ctc_return = cmf->cmf_rtid;
                        fi.ctc_argc = cmf->cmf_argc;
                        fi.ctc_flags = cmf->cmf_flags;
                        if ((err = ctf_add_function(fp, i, &fi,
                            cmf->cmf_args)) != 0)
                                return (err);
        }
        }

        return (0);

}

int
ctf_merge_merge(ctf_merge_t *cmh, ctf_file_t **outp)
{
        int err, merr;
        ctf_merge_input_t *cmi;
        ctf_id_t ltype;
        mergeq_t *mqp;
        ctf_merge_input_t *final;
        ctf_file_t *out;

        ctf_dprintf("Beginning ctf_merge_merge()\n");
        if (cmh->cmh_label != NULL && cmh->cmh_unique != NULL) {
                const char *label = ctf_label_topmost(cmh->cmh_unique);
                if (label == NULL)
                        return (ECTF_NOLABEL);
                if (strcmp(label, cmh->cmh_label) != 0)
                        return (ECTF_LCONFLICT);
        }

        if (mergeq_init(&mqp, cmh->cmh_nthreads) == -1) {
                return (errno);
        }

        VERIFY(cmh->cmh_ninputs % 2 == 0);
        for (cmi = list_head(&cmh->cmh_inputs); cmi != NULL;
            cmi = list_next(&cmh->cmh_inputs, cmi)) {
                if (mergeq_add(mqp, cmi) == -1) {
                        err = errno;
                        mergeq_fini(mqp);
                }
        }

        err = mergeq_merge(mqp, ctf_merge_types, NULL, (void **)&final, &merr);
        mergeq_fini(mqp);

        if (err == MERGEQ_ERROR) {
                return (errno);
        } else if (err == MERGEQ_UERROR) {
                return (merr);
        }

        /*
         * Disassociate the generated ctf_file_t from the original input. That
         * way when the input gets cleaned up, we don't accidentally kill the
         * final reference to the ctf_file_t. If it gets uniquified then we'll
         * kill it.
         */
        VERIFY(final->cmi_input != NULL);
        out = final->cmi_input;
        final->cmi_input = NULL;

        ctf_dprintf("preparing to uniquify against: %p\n", cmh->cmh_unique);
```

```
1491         if (cmh->cmh_unique != NULL) {
1492                 ctf_file_t *u;
1493                 err = ctf_uniquify_types(cmh, out, &u);
1494                 if (err != 0) {
1495                         err = ctf_errno(out);
1496                         ctf_close(out);
1497                         return (err);
1498                 }
1499                 ctf_close(out);
1500                 out = u;
1501         }

1503         ltype = out->ctf_typemax;
1504         if ((out->ctf_flags & LCTF_CHILD) && ltype != 0)
1505                 ltype += CTF_CHILD_START;
1506         ctf_dprintf("trying to add the label\n");
1507         if (cmh->cmh_label != NULL &&
1508             ctf_add_label(out, cmh->cmh_label, ltype, 0) != 0) {
1509                 ctf_close(out);
1510                 return (ctf_errno(out));
1511         }

1513         ctf_dprintf("merging symbols and the like\n");
1514         if (cmh->cmh_msyms == B_TRUE) {
1515                 ctf_merge_symbol_arg_t arg;
1516                 arg.cmsa_objmap = &final->cmi_omap;
1517                 arg.cmsa_funcmap = &final->cmi_fmap;
1518                 arg.cmsa_out = out;
1519                 arg.cmsa_dedup = B_FALSE;
1520                 err = ctf_symtab_iter(out, ctf_merge_symbols, &arg);
1390                 err = ctf_merge_symbols(cmh, out);
1521                 if (err != 0) {
1522                         ctf_close(out);
1523                         return (err);
1393                         return (ctf_errno(out));
1524                 }

1396                 err = ctf_merge_functions(cmh, out);
1397                 if (err != 0) {
1398                         ctf_close(out);
1399                         return (ctf_errno(out));
1525         }
1401         }

1527         err = ctf_update(out);
1528         if (err != 0) {
1529                 err = ctf_errno(out);
1530                 ctf_close(out);
1531                 return (err);
1406                 return (ctf_errno(out));
1532         }

1534         *outp = out;
1535         return (0);
1536 }
```

*unchanged_portion_omitted_*

```
1569 /*
1570  * Dedup a CTF container.
1571  *
1572  * DWARF and other encoding formats that we use to create CTF data may create
1573  * multiple copies of a given type. However, after doing a conversion, and
1574  * before doing a merge, we'd prefer, if possible, to have every input container
1575  * to be unique.
1576  *
1577  * Doing a deduplication is like a normal merge. However, when we diff the types
```



```
1578  * in the container, rather than doing a normal diff, we instead want to diff
1579  * against any already processed types. eg, for a given type i in a container,
1580  * we want to diff it from 0 to i - 1.
1581  */
1582 int
1583 ctf_merge_dedup(ctf_merge_t *cmp, ctf_file_t **outp)
1584 {
1585         int ret;
1586         ctf_diff_t *cdp = NULL;
1587         ctf_merge_input_t *cmi, *cmc;
1588         ctf_file_t *ifp, *ofp;
1589         ctf_merge_types_t cm;

1591         if (cmp == NULL || outp == NULL)
1592                 return (EINVAL);

1594         ctf_dprintf("encountered %d inputs\n", cmp->cmh_ninputs);
1595         if (cmp->cmh_ninputs != 2)
1596                 return (EINVAL);

1598         ctf_dprintf("passed argument sanity check\n");

1600         cmi = list_head(&cmp->cmh_inputs);
1601         VERIFY(cmi != NULL);
1602         cmc = list_next(&cmp->cmh_inputs, cmi);
1603         VERIFY(cmc != NULL);
1604         ifp = cmi->cmi_input;
1605         ofp = cmc->cmi_input;
1606         VERIFY(ifp != NULL);
1607         VERIFY(ofp != NULL);
1608         cm.cm_src = ifp;
1609         cm.cm_out = ofp;
1610         cm.cm_dedup = B_TRUE;
1611         cm.cm_unique = B_FALSE;

1613         if ((ret = ctf_merge_types_init(&cm)) != 0) {
1614                 return (ret);
1615         }

1617         if ((ret = ctf_diff_init(ifp, ifp, &cdp)) != 0)
1618                 goto err;

1620         ctf_dprintf("Successfully initialized dedup\n");
1621         if ((ret = ctf_diff_self(cdp, ctf_dedup_cb, &cm)) != 0)
1622                 goto err;

1624         ctf_dprintf("Successfully diffed types\n");
1625         ret = ctf_merge_common(&cm);
1626         ctf_dprintf("deduping types result: %d\n", ret);
1627         if (ret == 0)
1628                 ret = ctf_update(cm.cm_out);
1629         if (ret != 0)
1630                 goto err;

1632         ctf_dprintf("Successfully deduped types\n");
1633         ctf_phase_dump(cm.cm_out, "dedup-pre-syms", NULL);
1508         ctf_phase_dump(cm.cm_out, "dedup-pre-syms");

1635         /*
1636          * Now we need to fix up the object and function maps.
1637          */
1638         ctf_merge_fixup_symmaps(&cm, cmi);
1513         ctf_merge_fixup_nontypes(&cm, cmi);

1640         if (cmp->cmh_msyms == B_TRUE) {
1641                 ctf_merge_symbol_arg_t arg;
```

```
1642                 arg.cmsa_objmap = &cmi->cmi_omap;
1643                 arg.cmsa_funcmap = &cmi->cmi_fmap;
1644                 arg.cmsa_out = cm.cm_out;
1645                 arg.cmsa_dedup = B_TRUE;
1646                 ret = ctf_symtab_iter(cm.cm_out, ctf_merge_symbols, &arg);
1516                 ret = ctf_merge_symbols(cmp, cm.cm_out);
1647                 if (ret != 0) {
1518                         ret = ctf_errno(cm.cm_out);
1648                         ctf_dprintf("failed to dedup symbols: %s\n",
1649                             ctf_errmsg(ret));
1650                         goto err;
1651                 }

1524                 ret = ctf_merge_functions(cmp, cm.cm_out);
1525                 if (ret != 0) {
1526                         ret = ctf_errno(cm.cm_out);
1527                         ctf_dprintf("failed to dedup functions: %s\n",
1528                             ctf_errmsg(ret));
1529                         goto err;
1652         }
1531         }

1654         ret = ctf_update(cm.cm_out);
1655         if (ret == 0) {
1656                 cmc->cmi_input = NULL;
1657                 *outp = cm.cm_out;
1658         }
1659         ctf_phase_dump(cm.cm_out, "dedup-post-syms", NULL);
1660 err:
1661         ctf_merge_types_fini(&cm);
1662         ctf_diff_fini(cdp);
1663         return (ret);
1664 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    1648 Tue Apr 23 05:24:02 2019
new/usr/src/lib/libctf/common/libctf_impl.h
10812 ctf tools shouldn't add blank labels
10813 ctf symbol mapping needs work
Reviewed by: Jerry Jelinek <jerry.jelinek@joyent.com>
**********************************************************
```

```
/*
 * This file and its contents are supplied under the terms of the
 * Common Development and Distribution License ("CDDL"), version 1.0.
 * You may only use this file in accordance with the terms of version
 * 1.0 of the CDDL.
 *
 * A full copy of the text of the CDDL should have accompanied this
 * source.  A copy of the CDDL is also available via the Internet at
 * http://www.illumos.org/license/CDDL.
 */

/*
 * Copyright 2019 Joyent, Inc.
 * Copyright 2015 Joyent, Inc.
 */

#ifndef _LIBCTF_IMPL_H
#define _LIBCTF_IMPL_H

/*
 * Portions of libctf implementations that are only suitable for CTF's userland
 * library, eg. converting and merging related routines.
 */

#include <libelf.h>
#include <libctf.h>
#include <ctf_impl.h>

#ifdef __cplusplus
extern "C" {
#endif

typedef enum ctf_conv_status {
        CTF_CONV_SUCCESS        = 0,
        CTF_CONV_ERROR          = 1,
        CTF_CONV_NOTSUP         = 2
} ctf_conv_status_t;

typedef ctf_conv_status_t (*ctf_convert_f)(int, Elf *, uint_t, int *,
    ctf_file_t **, char *, size_t);
extern ctf_conv_status_t ctf_dwarf_convert(int, Elf *, uint_t, int *,
    ctf_file_t **, char *, size_t);

/*
 * Symbol walking
 */
typedef int (*ctf_symtab_f)(const Elf64_Sym *, ulong_t, const char *,
    const char *, boolean_t, void *);
extern int ctf_symtab_iter(ctf_file_t *, ctf_symtab_f, void *);

/*
 * zlib compression routines
 */
extern int ctf_compress(ctf_file_t *fp, void **, size_t *, size_t *);

extern int ctf_diff_self(ctf_diff_t *, ctf_diff_type_f, void *);

/*
 * Internal debugging aids
```



```
 */
extern void ctf_phase_dump(ctf_file_t *, const char *, const char *);
extern void ctf_phase_bump(void);
extern void ctf_phase_dump(ctf_file_t *, const char *);

#ifdef __cplusplus
}
_____unchanged_portion_omitted_
```