```
**********************************************************
   25398 Mon Mar  4 02:11:14 2019
new/usr/src/cmd/sgs/rtld/amd64/amd64_elf.c
smatch clean rtld
**********************************************************
______unchanged_portion_omitted_

 197 /*
 198  * Function binding routine - invoked on the first call to a function through
 199  * the procedure linkage table;
 200  * passes first through an assembly language interface.
 201  *
 202  * Takes the offset into the relocation table of the associated
 203  * relocation entry and the address of the link map (rt_private_map struct)
 204  * for the entry.
 205  *
 206  * Returns the address of the function referenced after re-writing the PLT
 207  * entry to invoke the function directly.
 208  *
 209  * On error, causes process to terminate with a signal.
 210  */
 211 ulong_t
 212 elf_bndr(Rt_map *lmp, ulong_t pltndx, caddr_t from)
 213 {
 214         Rt_map          *nlmp, *llmp;
 215         ulong_t         addr, reloff, symval, rsymndx;
 216         char            *name;
 217         Rela            *rptr;
 218         Sym             *rsym, *nsym;
 219         uint_t          binfo, sb_flags = 0, dbg_class;
 220         Slookup         sl;
 221         Sresult         sr;
 222         int             entry, lmflags;
 223         Lm_list         *lml;

 225         /*
 226          * For compatibility with libthread (TI_VERSION 1) we track the entry
 227          * value.  A zero value indicates we have recursed into ld.so.1 to
 228          * further process a locking request.  Under this recursion we disable
 229          * tsort and cleanup activities.
 230          */
 231         entry = enter(0);

 233         lml = LIST(lmp);
 234         if ((lmflags = lml->lm_flags) & LML_FLG_RTLDLM) {
 235                 dbg_class = dbg_desc->d_class;
 236                 dbg_desc->d_class = 0;
 237         }

 239         /*
 240          * Perform some basic sanity checks.  If the relocation offset is
 241          * invalid then its possible someone has walked over the .got entries.
 240          * Perform some basic sanity checks.  If we didn't get a load map or
 241          * the relocation offset is invalid then its possible someone has walked
 242          * over the .got entries or jumped to plt0 out of the blue.
 242          */
 243         if (pltndx > (ulong_t)PLTRELSZ(lmp) / (ulong_t)RELENT(lmp)) {
 244         if ((!lmp) && (pltndx <=
 245             (ulong_t)PLTRELSZ(lmp) / (ulong_t)RELENT(lmp))) {
 244                 Conv_inv_buf_t inv_buf;

 246                 eprintf(lml, ERR_FATAL, MSG_INTL(MSG_REL_PLTREF),
 247                     conv_reloc_amd64_type(R_AMD64_JUMP_SLOT, 0, &inv_buf),
 248                     EC_NATPTR(lmp), EC_XWORD(pltndx), EC_NATPTR(from));
 249                 rtldexit(lml, 1);
 250         }
```



```
 251         reloff = pltndx * (ulong_t)RELENT(lmp);

 253         /*
 254          * Use relocation entry to get symbol table entry and symbol name.
 255          */
 256         addr = (ulong_t)JMPREL(lmp);
 257         rptr = (Rela *)(addr + reloff);
 258         rsymndx = ELF_R_SYM(rptr->r_info);
 259         rsym = (Sym *)((ulong_t)SYMTAB(lmp) + (rsymndx * SYMENT(lmp)));
 260         name = (char *)(STRTAB(lmp) + rsym->st_name);

 262         /*
 263          * Determine the last link-map of this list, this'll be the starting
 264          * point for any tsort() processing.
 265          */
 266         llmp = lml->lm_tail;

 268         /*
 269          * Find definition for symbol.  Initialize the symbol lookup, and
 270          * symbol result, data structures.
 271          */
 272         SLOOKUP_INIT(sl, name, lmp, lml->lm_head, ld_entry_cnt, 0,
 273             rsymndx, rsym, 0, LKUP_DEFT);
 274         SRESULT_INIT(sr, name);

 276         if (lookup_sym(&sl, &sr, &binfo, NULL) == 0) {
 277                 eprintf(lml, ERR_FATAL, MSG_INTL(MSG_REL_NOSYM), NAME(lmp),
 278                     demangle(name));
 279                 rtldexit(lml, 1);
 280         }

 282         name = (char *)sr.sr_name;
 283         nlmp = sr.sr_dmap;
 284         nsym = sr.sr_sym;

 286         symval = nsym->st_value;

 288         if (!(FLAGS(nlmp) & FLG_RT_FIXED) &&
 289             (nsym->st_shndx != SHN_ABS))
 290                 symval += ADDR(nlmp);
 291         if ((lmp != nlmp) && ((FLAGS1(nlmp) & FL1_RT_NOINIFIN) == 0)) {
 292                 /*
 293                  * Record that this new link map is now bound to the caller.
 294                  */
 295                 if (bind_one(lmp, nlmp, BND_REFER) == 0)
 296                         rtldexit(lml, 1);
 297         }

 299         if ((lml->lm_tflags | AFLAGS(lmp) | AFLAGS(nlmp)) &
 300             LML_TFLG_AUD_SYMBIND) {
 301                 uint_t  symndx = (((uintptr_t)nsym -
 302                     (uintptr_t)SYMTAB(nlmp)) / SYMENT(nlmp));
 303                 symval = audit_symbind(lmp, nlmp, nsym, symndx, symval,
 304                     &sb_flags);
 305         }

 307         if (!(rtld_flags & RT_FL_NOBIND)) {
 308                 addr = rptr->r_offset;
 309                 if (!(FLAGS(lmp) & FLG_RT_FIXED))
 310                         addr += ADDR(lmp);
 311                 if (((lml->lm_tflags | AFLAGS(lmp)) &
 312                     (LML_TFLG_AUD_PLTENTER | LML_TFLG_AUD_PLTEXIT)) &&
 313                     AUDINFO(lmp)->ai_dynplts) {
 314                         int     fail = 0;
 315                         uint_t  pltndx = reloff / sizeof (Rela);
 316                         uint_t  symndx = (((uintptr_t)nsym -
```

```
317                             (uintptr_t)SYMTAB(nlmp)) / SYMENT(nlmp));

319                         symval = (ulong_t)elf_plt_trace_write(addr, lmp, nlmp,
320                             nsym, symndx, pltndx, (caddr_t)symval, sb_flags,
321                             &fail);
322                         if (fail)
323                                 rtldexit(lml, 1);
324                 } else {
325                         /*
326                          * Write standard PLT entry to jump directly
327                          * to newly bound function.
328                          */
329                         *(ulong_t *)addr = symval;
330                 }
331         }

333         /*
334          * Print binding information and rebuild PLT entry.
335          */
336         DBG_CALL(Dbg_bind_global(lmp, (Addr)from, (Off)(from - ADDR(lmp)),
337             (Xword)(reloff / sizeof (Rela)), PLT_T_FULL, nlmp, (Addr)symval,
338             nsym->st_value, name, binfo));

340         /*
341          * Complete any processing for newly loaded objects.  Note we don't
342          * know exactly where any new objects are loaded (we know the object
343          * that supplied the symbol, but others may have been loaded lazily as
344          * we searched for the symbol), so sorting starts from the last
345          * link-map know on entry to this routine.
346          */
347         if (entry)
348                 load_completion(llmp);

350         /*
351          * Some operations like dldump() or dlopen()'ing a relocatable object
352          * result in objects being loaded on rtld's link-map, make sure these
353          * objects are initialized also.
354          */
355         if ((LIST(nlmp)->lm_flags & LML_FLG_RTLDLM) && LIST(nlmp)->lm_init)
356                 load_completion(nlmp);

358         /*
359          * Make sure the object to which we've bound has had it's .init fired.
360          * Cleanup before return to user code.
361          */
362         if (entry) {
363                 is_dep_init(nlmp, lmp);
364                 leave(lml, 0);
365         }

367         if (lmflags & LML_FLG_RTLDLM)
368                 dbg_desc->d_class = dbg_class;

370         return (symval);
371 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   99688 Mon Mar  4 02:11:14 2019
new/usr/src/cmd/sgs/rtld/common/analyze.c
smatch clean rtld
**********************************************************
______unchanged_portion_omitted_
```

```
**********************************************************
   43886 Mon Mar  4 02:11:15 2019
new/usr/src/cmd/sgs/rtld/common/audit.c
smatch clean rtld
**********************************************************
______unchanged_portion_omitted_

 698 /*
 699  * la_objclose() caller.  Traverse through all audit libraries and call any
 700  * la_objclose() entry points found.
 701  */
 702 void
 703 _audit_objclose(APlist *list, Rt_map *lmp)
 704 {
 705         Audit_list      *alp;
 706         Aliste          idx;
 707         Lm_list         *lml = LIST(lmp);

 709         for (APLIST_TRAVERSE(list, idx, alp)) {
 710                 Audit_client    *acp;
 711                 Rt_map          *almp = alp->al_lmp;
 712                 Lm_list         *alml = LIST(almp);

 714                 if (alp->al_objclose == NULL)
 715                         continue;
 716                 if ((acp = _audit_client(AUDINFO(lmp), almp)) == NULL)
 717                         continue;

 719                 DBG_CALL(Dbg_audit_objclose(lml, alp->al_libname, NAME(lmp)));

 721                 leave(alml, thr_flg_reenter);
 722                 (void) (*alp->al_objclose)(&(acp->ac_cookie));
 722                 (*alp->al_objclose)(&(acp->ac_cookie));
 723                 (void) enter(thr_flg_reenter);
 724         }
 725 }
______unchanged_portion_omitted_
```

```
**********************************************************
   38030 Mon Mar  4 02:11:15 2019
new/usr/src/cmd/sgs/rtld/common/cap.c
smatch clean rtld
**********************************************************
______unchanged_portion_omitted_
 985 #undef  ELFCAP_STYLE

 987 /*
 988  * Create an AVL tree of objects that are to be validated against an alternative
 989  * system capabilities value.
 990  */
 991 static int
 992 cap_files(const char *str)
 993 {
 994         char    *caps, *name, *next;

 996         if ((caps = strdup(str)) == NULL)
 997                 return (0);

 999         for (name = strtok_r(caps, MSG_ORIG(MSG_CAP_DELIMIT), &next);
1000             name != NULL;
1001             name = strtok_r(NULL, MSG_ORIG(MSG_CAP_DELIMIT), &next)) {
1002                 avl_index_t     where;
1003                 PathNode        *pnp;
1004                 uint_t          hash = sgs_str_hash(name);

1006                 /*
1007                  * Determine whether this pathname has already been recorded.
1008                  */
1009                 if (pnavl_recorded(&capavl, name, hash, &where))
1010                         continue;

1012                 if ((pnp = calloc(1, sizeof (PathNode))) != NULL) {
1012                 if ((pnp = calloc(sizeof (PathNode), 1)) != NULL) {
1013                         pnp->pn_name = name;
1014                         pnp->pn_hash = hash;
1015                         avl_insert(capavl, pnp, where);
1016                 }
1017         }

1019         return (1);
1020 }
______unchanged_portion_omitted_
```

```
**********************************************************
   62075 Mon Mar  4 02:11:16 2019
new/usr/src/cmd/sgs/rtld/common/dlfcns.c
smatch clean rtld
**********************************************************
______unchanged_portion_omitted_

 851 /*
 852  * Internal dlopen() activity.  Called from user level or directly for internal
 853  * opens that require a handle.
 854  */
 855 Grp_hdl *
 856 dlmopen_intn(Lm_list *lml, const char *path, int mode, Rt_map *clmp,
 857     uint_t flags, uint_t orig)
 858 {
 859         Lm_list *olml = lml;
 860         Rt_map  *dlmp = NULL;
 861         Grp_hdl *ghp;
 862         int     in_nfavl = 0;

 864         /*
 865          * Check for magic link-map list values:
 866          *
 867          *  LM_ID_BASE:         Operate on the PRIMARY (executables) link map
 868          *  LM_ID_LDSO:         Operation on ld.so.1's link map
 869          *  LM_ID_NEWLM:        Create a new link-map.
 870          */
 871         if (lml == (Lm_list *)LM_ID_NEWLM) {
 872                 if ((lml = calloc(1, sizeof (Lm_list))) == NULL)
 872                 if ((lml = calloc(sizeof (Lm_list), 1)) == NULL)
 873                         return (NULL);

 875                 /*
 876                  * Establish the new link-map flags from the callers and those
 877                  * explicitly provided.
 878                  */
 879                 lml->lm_tflags = LIST(clmp)->lm_tflags;
 880                 if (flags & FLG_RT_AUDIT) {
 881                         /*
 882                          * Unset any auditing flags - an auditor shouldn't be
 883                          * audited.  Insure all audit dependencies are loaded.
 884                          */
 885                         lml->lm_tflags &= ~LML_TFLG_AUD_MASK;
 886                         lml->lm_tflags |= (LML_TFLG_NOLAZYLD |
 887                             LML_TFLG_LOADFLTR | LML_TFLG_NOAUDIT);
 888                 }

 890                 if (aplist_append(&dynlm_list, lml, AL_CNT_DYNLIST) == NULL) {
 891                         free(lml);
 892                         return (NULL);
 893                 }
 894                 if (newlmid(lml) == 0) {
 895                         (void) aplist_delete_value(dynlm_list, lml);
 896                         free(lml);
 897                         return (NULL);
 898                 }
 899         } else if ((uintptr_t)lml < LM_ID_NUM) {
 900                 if ((uintptr_t)lml == LM_ID_BASE)
 901                         lml = &lml_main;
 902                 else if ((uintptr_t)lml == LM_ID_LDSO)
 903                         lml = &lml_rtld;
 904         }

 906         /*
 907          * Open the required object on the associated link-map list.
 908          */
```



```
 909         ghp = dlmopen_core(lml, olml, path, mode, clmp, flags, orig, &in_nfavl);

 911         /*
 912          * If the object could not be found it is possible that the "not-found"
 913          * AVL tree had indicated that the file does not exist.  In case the
 914          * file system has changed since this "not-found" recording was made,
 915          * retry the dlopen() with a clean "not-found" AVL tree.
 916          */
 917         if ((ghp == NULL) && in_nfavl) {
 918                 avl_tree_t      *oavlt = nfavl;

 920                 nfavl = NULL;
 921                 ghp = dlmopen_core(lml, olml, path, mode, clmp, flags, orig,
 922                     NULL);

 924                 /*
 925                  * If the file is found, then its full path name will have been
 926                  * registered in the FullPath AVL tree.  Remove any new
 927                  * "not-found" AVL information, and restore the former AVL tree.
 928                  */
 929                 nfavl_remove(nfavl);
 930                 nfavl = oavlt;
 931         }

 933         /*
 934          * Establish the new link-map from which .init processing will begin.
 935          * Ignore .init firing when constructing a configuration file (crle(1)).
 936          */
 937         if (ghp && ((mode & RTLD_CONFGEN) == 0))
 938                 dlmp = ghp->gh_ownlmp;

 940         /*
 941          * If loading an auditor was requested, and the auditor already existed,
 942          * then the link-map returned will be to the original auditor.  Remove
 943          * the link-map control list that was created for this request.
 944          */
 945         if (dlmp && (flags & FLG_RT_AUDIT) && (LIST(dlmp) != lml)) {
 946                 remove_lml(lml);
 947                 lml = LIST(dlmp);
 948         }

 950         /*
 951          * If this load failed, remove any alternative link-map list.
 952          */
 953         if ((ghp == NULL) &&
 954             ((lml->lm_flags & (LML_FLG_BASELM | LML_FLG_RTLDLM)) == 0)) {
 955                 remove_lml(lml);
 956                 lml = NULL;
 957         }

 959         /*
 960          * Finish this load request.  If objects were loaded, .init processing
 961          * is computed.  Finally, the debuggers are informed of the link-map
 962          * lists being stable.
 963          */
 964         load_completion(dlmp);

 966         return (ghp);
 967 }
______unchanged_portion_omitted_
```

```
**********************************************************
   86340 Mon Mar  4 02:11:16 2019
new/usr/src/cmd/sgs/rtld/common/elf.c
smatch clean rtld
**********************************************************
______unchanged_portion_omitted_

1644 /*
1645  * Create a new Rt_map structure for an ELF object and initialize
1646  * all values.
1647  */
1648 Rt_map *
1649 elf_new_lmp(Lm_list *lml, Aliste lmco, Fdesc *fdp, Addr addr, size_t msize,
1650     void *odyn, Rt_map *clmp, int *in_nfavl)
1651 {
1652         const char      *name = fdp->fd_nname;
1653         Rt_map          *lmp;
1654         Ehdr            *ehdr = (Ehdr *)addr;
1655         Phdr            *phdr, *tphdr = NULL, *dphdr = NULL, *uphdr = NULL;
1656         Dyn             *dyn = (Dyn *)odyn;
1657         Cap             *cap = NULL;
1658         int             ndx;
1659         Addr            base, fltr = 0, audit = 0, cfile = 0, crle = 0;
1660         Xword           rpath = 0;
1661         size_t          lmsz, rtsz, epsz, dynsz = 0;
1662         uint_t          dyncnt = 0;

1664         DBG_CALL(Dbg_file_elf(lml, name, addr, msize, lml->lm_lmidstr, lmco));

1666         /*
1667          * If this is a shared object, the base address of the shared object is
1668          * added to all address values defined within the object.  Otherwise, if
1669          * this is an executable, all object addresses are used as is.
1670          */
1671         if (ehdr->e_type == ET_EXEC)
1672                 base = 0;
1673         else
1674                 base = addr;

1676         /*
1677          * Traverse the program header table, picking off required items.  This
1678          * traversal also provides for the sizing of the PT_DYNAMIC section.
1679          */
1680         phdr = (Phdr *)((uintptr_t)ehdr + ehdr->e_phoff);
1681         for (ndx = 0; ndx < (int)ehdr->e_phnum; ndx++,
1682             phdr = (Phdr *)((uintptr_t)phdr + ehdr->e_phentsize)) {
1683                 switch (phdr->p_type) {
1684                 case PT_DYNAMIC:
1685                         dphdr = phdr;
1686                         dyn = (Dyn *)((uintptr_t)phdr->p_vaddr + base);
1687                         break;
1688                 case PT_TLS:
1689                         tphdr = phdr;
1690                         break;
1691                 case PT_SUNWCAP:
1692                         cap = (Cap *)((uintptr_t)phdr->p_vaddr + base);
1693                         break;
1694                 case PT_SUNW_UNWIND:
1695                 case PT_SUNW_EH_FRAME:
1696                         uphdr = phdr;
1697                         break;
1698                 default:
1699                         break;
1700                 }
1701         }
```



```
1703         /*
1704          * Determine the number of PT_DYNAMIC entries for the DYNINFO()
1705          * allocation.  Sadly, this is a little larger than we really need,
1706          * as there are typically padding DT_NULL entries.  However, adding
1707          * this data to the initial link-map allocation is a win.
1708          */
1709         if (dyn) {
1710                 dyncnt = dphdr->p_filesz / sizeof (Dyn);
1711                 dynsz = dyncnt * sizeof (Dyninfo);
1712         }

1714         /*
1715          * Allocate space for the link-map, private elf information, and
1716          * DYNINFO() data.  Once these are allocated and initialized,
1717          * remove_so(0, lmp) can be used to tear down the link-map allocation
1718          * should any failures occur.
1719          */
1720         rtsz = S_DROUND(sizeof (Rt_map));
1721         epsz = S_DROUND(sizeof (Rt_elfp));
1722         lmsz = rtsz + epsz + dynsz;
1723         if ((lmp = calloc(1, lmsz)) == NULL)
1723         if ((lmp = calloc(lmsz, 1)) == NULL)
1724                 return (NULL);
1725         ELFPRV(lmp) = (void *)((uintptr_t)lmp + rtsz);
1726         DYNINFO(lmp) = (Dyninfo *)((uintptr_t)lmp + rtsz + epsz);
1727         LMSIZE(lmp) = lmsz;

1729         /*
1730          * All fields not filled in were set to 0 by calloc.
1731          */
1732         NAME(lmp) = (char *)name;
1733         ADDR(lmp) = addr;
1734         MSIZE(lmp) = msize;
1735         SYMINTP(lmp) = elf_find_sym;
1736         FCT(lmp) = &elf_fct;
1737         LIST(lmp) = lml;
1738         OBJFLTRNDX(lmp) = FLTR_DISABLED;
1739         SORTVAL(lmp) = -1;
1740         DYN(lmp) = dyn;
1741         DYNINFOCNT(lmp) = dyncnt;
1742         PTUNWIND(lmp) = uphdr;

1744         if (ehdr->e_type == ET_EXEC)
1745                 FLAGS(lmp) |= FLG_RT_FIXED;

1747         /*
1748          * Fill in rest of the link map entries with information from the file's
1749          * dynamic structure.
1750          */
1751         if (dyn) {
1752                 Dyninfo         *dip;
1753                 uint_t          dynndx;
1754                 Xword           pltpadsz = 0;
1755                 Rti_desc        *rti;
1756                 Dyn             *pdyn;
1757                 Word            lmtflags = lml->lm_tflags;
1758                 int             ignore = 0;

1760                 /*
1761                  * Note, we use DT_NULL to terminate processing, and the
1762                  * dynamic entry count as a fall back.  Normally, a DT_NULL
1763                  * entry marks the end of the dynamic section.  Any non-NULL
1764                  * items following the first DT_NULL are silently ignored.
1765                  * This situation should only occur through use of elfedit(1)
1766                  * or a similar tool.
1767                  */
```

```
1768                for (dynndx = 0, pdyn = NULL, dip = DYNINFO(lmp);
1769                    dynndx < dyncnt; dynndx++, pdyn = dyn++, dip++) {

1771                        if (ignore) {
1772                                dip->di_flags |= FLG_DI_IGNORE;
1773                                continue;
1774                        }

1776                        switch ((Xword)dyn->d_tag) {
1777                        case DT_NULL:
1778                                dip->di_flags |= ignore = FLG_DI_IGNORE;
1779                                break;
1780                        case DT_POSFLAG_1:
1781                                dip->di_flags |= FLG_DI_POSFLAG1;
1782                                break;
1783                        case DT_NEEDED:
1784                        case DT_USED:
1785                                dip->di_flags |= FLG_DI_NEEDED;

1787                                /* BEGIN CSTYLED */
1788                                if (pdyn && (pdyn->d_tag == DT_POSFLAG_1)) {
1789                                    /*
1790                                     * Identify any non-deferred lazy load for
1791                                     * future processing, unless LD_NOLAZYLOAD
1792                                     * has been set.
1793                                     */
1794                                    if ((pdyn->d_un.d_val & DF_P1_LAZYLOAD) &&
1795                                        ((lmtflags & LML_TFLG_NOLAZYLD) == 0))
1796                                            dip->di_flags |= FLG_DI_LAZY;

1798                                    /*
1799                                     * Identify any group permission
1800                                     * requirements.
1801                                     */
1802                                    if (pdyn->d_un.d_val & DF_P1_GROUPPERM)
1803                                            dip->di_flags |= FLG_DI_GROUP;

1805                                    /*
1806                                     * Identify any deferred dependencies.
1807                                     */
1808                                    if (pdyn->d_un.d_val & DF_P1_DEFERRED)
1809                                            dip->di_flags |= FLG_DI_DEFERRED;
1810                                }
1811                                /* END CSTYLED */
1812                                break;
1813                        case DT_SYMTAB:
1814                                SYMTAB(lmp) = (void *)(dyn->d_un.d_ptr + base);
1815                                break;
1816                        case DT_SUNW_SYMTAB:
1817                                SUNWSYMTAB(lmp) =
1818                                    (void *)(dyn->d_un.d_ptr + base);
1819                                break;
1820                        case DT_SUNW_SYMSZ:
1821                                SUNWSYMSZ(lmp) = dyn->d_un.d_val;
1822                                break;
1823                        case DT_STRTAB:
1824                                STRTAB(lmp) = (void *)(dyn->d_un.d_ptr + base);
1825                                break;
1826                        case DT_SYMENT:
1827                                SYMENT(lmp) = dyn->d_un.d_val;
1828                                break;
1829                        case DT_FEATURE_1:
1830                                if (dyn->d_un.d_val & DTF_1_CONFEXP)
1831                                        crle = 1;
1832                                break;
1833                        case DT_MOVESZ:
```



```
1834                                MOVESZ(lmp) = dyn->d_un.d_val;
1835                                FLAGS(lmp) |= FLG_RT_MOVE;
1836                                break;
1837                        case DT_MOVEENT:
1838                                MOVEENT(lmp) = dyn->d_un.d_val;
1839                                break;
1840                        case DT_MOVETAB:
1841                                MOVETAB(lmp) = (void *)(dyn->d_un.d_ptr + base);
1842                                break;
1843                        case DT_REL:
1844                        case DT_RELA:
1845                                /*
1846                                 * At this time, ld.so. can only handle one
1847                                 * type of relocation per object.
1848                                 */
1849                                REL(lmp) = (void *)(dyn->d_un.d_ptr + base);
1850                                break;
1851                        case DT_RELSZ:
1852                        case DT_RELASZ:
1853                                RELSZ(lmp) = dyn->d_un.d_val;
1854                                break;
1855                        case DT_RELENT:
1856                        case DT_RELAENT:
1857                                RELENT(lmp) = dyn->d_un.d_val;
1858                                break;
1859                        case DT_RELCOUNT:
1860                        case DT_RELACOUNT:
1861                                RELACOUNT(lmp) = (uint_t)dyn->d_un.d_val;
1862                                break;
1863                        case DT_HASH:
1864                                HASH(lmp) = (uint_t *)(dyn->d_un.d_ptr + base);
1865                                break;
1866                        case DT_PLTGOT:
1867                                PLTGOT(lmp) =
1868                                    (uint_t *)(dyn->d_un.d_ptr + base);
1869                                break;
1870                        case DT_PLTRELSZ:
1871                                PLTRELSZ(lmp) = dyn->d_un.d_val;
1872                                break;
1873                        case DT_JMPREL:
1874                                JMPREL(lmp) = (void *)(dyn->d_un.d_ptr + base);
1875                                break;
1876                        case DT_INIT:
1877                                if (dyn->d_un.d_ptr != NULL)
1878                                        INIT(lmp) =
1879                                            (void (*)())(dyn->d_un.d_ptr +
1880                                            base);
1881                                break;
1882                        case DT_FINI:
1883                                if (dyn->d_un.d_ptr != NULL)
1884                                        FINI(lmp) =
1885                                            (void (*)())(dyn->d_un.d_ptr +
1886                                            base);
1887                                break;
1888                        case DT_INIT_ARRAY:
1889                                INITARRAY(lmp) = (Addr *)(dyn->d_un.d_ptr +
1890                                    base);
1891                                break;
1892                        case DT_INIT_ARRAYSZ:
1893                                INITARRAYSZ(lmp) = (uint_t)dyn->d_un.d_val;
1894                                break;
1895                        case DT_FINI_ARRAY:
1896                                FINIARRAY(lmp) = (Addr *)(dyn->d_un.d_ptr +
1897                                    base);
1898                                break;
1899                        case DT_FINI_ARRAYSZ:
```

```
1900                            FINIARRAYSZ(lmp) = (uint_t)dyn->d_un.d_val;
1901                            break;
1902                    case DT_PREINIT_ARRAY:
1903                            PREINITARRAY(lmp) = (Addr *)(dyn->d_un.d_ptr +
1904                                base);
1905                            break;
1906                    case DT_PREINIT_ARRAYSZ:
1907                            PREINITARRAYSZ(lmp) = (uint_t)dyn->d_un.d_val;
1908                            break;
1909                    case DT_RPATH:
1910                    case DT_RUNPATH:
1911                            rpath = dyn->d_un.d_val;
1912                            break;
1913                    case DT_FILTER:
1914                            dip->di_flags |= FLG_DI_STDFLTR;
1915                            fltr = dyn->d_un.d_val;
1916                            OBJFLTRNDX(lmp) = dynndx;
1917                            FLAGS1(lmp) |= FL1_RT_OBJSFLTR;
1918                            break;
1919                    case DT_AUXILIARY:
1920                            dip->di_flags |= FLG_DI_AUXFLTR;
1921                            if (!(rtld_flags & RT_FL_NOAUXFLTR)) {
1922                                    fltr = dyn->d_un.d_val;
1923                                    OBJFLTRNDX(lmp) = dynndx;
1924                            }
1925                            FLAGS1(lmp) |= FL1_RT_OBJAFLTR;
1926                            break;
1927                    case DT_SUNW_FILTER:
1928                            dip->di_flags |=
1929                                (FLG_DI_STDFLTR | FLG_DI_SYMFLTR);
1930                            SYMSFLTRCNT(lmp)++;
1931                            FLAGS1(lmp) |= FL1_RT_SYMSFLTR;
1932                            break;
1933                    case DT_SUNW_AUXILIARY:
1934                            dip->di_flags |=
1935                                (FLG_DI_AUXFLTR | FLG_DI_SYMFLTR);
1936                            if (!(rtld_flags & RT_FL_NOAUXFLTR)) {
1937                                    SYMAFLTRCNT(lmp)++;
1938                            }
1939                            FLAGS1(lmp) |= FL1_RT_SYMAFLTR;
1940                            break;
1941                    case DT_DEPAUDIT:
1942                            if (!(rtld_flags & RT_FL_NOAUDIT)) {
1943                                    audit = dyn->d_un.d_val;
1944                                    FLAGS1(lmp) |= FL1_RT_DEPAUD;
1945                            }
1946                            break;
1947                    case DT_CONFIG:
1948                            cfile = dyn->d_un.d_val;
1949                            break;
1950                    case DT_DEBUG:
1951                            /*
1952                             * DT_DEBUG entries are only created in
1953                             * dynamic objects that require an interpretor
1954                             * (ie. all dynamic executables and some shared
1955                             * objects), and provide for a hand-shake with
1956                             * old debuggers.  This entry is initialized to
1957                             * zero by the link-editor.  If a debugger is
1958                             * monitoring us, and has updated this entry,
1959                             * set the debugger monitor flag, and finish
1960                             * initializing the debugging structure.  See
1961                             * setup().  Also, switch off any configuration
1962                             * object use as most debuggers can't handle
1963                             * fixed dynamic executables as dependencies.
1964                             */
1965                            if (dyn->d_un.d_ptr)
```



```
1966                                    rtld_flags |=
1967                                        (RT_FL_DEBUGGER | RT_FL_NOOBJALT);
1968                            dyn->d_un.d_ptr = (Addr)&r_debug;
1969                            break;
1970                    case DT_VERNEED:
1971                            VERNEED(lmp) = (Verneed *)(dyn->d_un.d_ptr +
1972                                base);
1973                            break;
1974                    case DT_VERNEEDNUM:
1975                            /* LINTED */
1976                            VERNEEDNUM(lmp) = (int)dyn->d_un.d_val;
1977                            break;
1978                    case DT_VERDEF:
1979                            VERDEF(lmp) = (Verdef *)(dyn->d_un.d_ptr +
1980                                base);
1981                            break;
1982                    case DT_VERDEFNUM:
1983                            /* LINTED */
1984                            VERDEFNUM(lmp) = (int)dyn->d_un.d_val;
1985                            break;
1986                    case DT_VERSYM:
1987                            /*
1988                             * The Solaris ld does not produce DT_VERSYM,
1989                             * but the GNU ld does, in order to support
1990                             * their style of versioning, which differs
1991                             * from ours in some ways, while using the
1992                             * same data structures. The presence of
1993                             * DT_VERSYM therefore means that GNU
1994                             * versioning rules apply to the given file.
1995                             * If DT_VERSYM is not present, then Solaris
1996                             * versioning rules apply.
1997                             */
1998                            VERSYM(lmp) = (Versym *)(dyn->d_un.d_ptr +
1999                                base);
2000                            break;
2001                    case DT_BIND_NOW:
2002                            if ((dyn->d_un.d_val & DF_BIND_NOW) &&
2003                                ((rtld_flags2 & RT_FL2_BINDLAZY) == 0)) {
2004                                    MODE(lmp) |= RTLD_NOW;
2005                                    MODE(lmp) &= ~RTLD_LAZY;
2006                            }
2007                            break;
2008                    case DT_FLAGS:
2009                            FLAGS1(lmp) |= FL1_RT_DTFLAGS;
2010                            if (dyn->d_un.d_val & DF_SYMBOLIC)
2011                                    FLAGS1(lmp) |= FL1_RT_SYMBOLIC;
2012                            if ((dyn->d_un.d_val & DF_BIND_NOW) &&
2013                                ((rtld_flags2 & RT_FL2_BINDLAZY) == 0)) {
2014                                    MODE(lmp) |= RTLD_NOW;
2015                                    MODE(lmp) &= ~RTLD_LAZY;
2016                            }
2017                            /*
2018                             * Capture any static TLS use, and enforce that
2019                             * this object be non-deletable.
2020                             */
2021                            if (dyn->d_un.d_val & DF_STATIC_TLS) {
2022                                    FLAGS1(lmp) |= FL1_RT_TLSSTAT;
2023                                    MODE(lmp) |= RTLD_NODELETE;
2024                            }
2025                            break;
2026                    case DT_FLAGS_1:
2027                            if (dyn->d_un.d_val & DF_1_DISPRELPND)
2028                                    FLAGS1(lmp) |= FL1_RT_DISPREL;
2029                            if (dyn->d_un.d_val & DF_1_GROUP)
2030                                    FLAGS(lmp) |=
2031                                        (FLG_RT_SETGROUP | FLG_RT_PUBHDL);
```

```
2032                                if ((dyn->d_un.d_val & DF_1_NOW) &&
2033                                    ((rtld_flags2 & RT_FL2_BINDLAZY) == 0)) {
2034                                        MODE(lmp) |= RTLD_NOW;
2035                                        MODE(lmp) &= ~RTLD_LAZY;
2036                                }
2037                                if (dyn->d_un.d_val & DF_1_NODELETE)
2038                                        MODE(lmp) |= RTLD_NODELETE;
2039                                if (dyn->d_un.d_val & DF_1_INITFIRST)
2040                                        FLAGS(lmp) |= FLG_RT_INITFRST;
2041                                if (dyn->d_un.d_val & DF_1_NOOPEN)
2042                                        FLAGS(lmp) |= FLG_RT_NOOPEN;
2043                                if (dyn->d_un.d_val & DF_1_LOADFLTR)
2044                                        FLAGS(lmp) |= FLG_RT_LOADFLTR;
2045                                if (dyn->d_un.d_val & DF_1_NODUMP)
2046                                        FLAGS(lmp) |= FLG_RT_NODUMP;
2047                                if (dyn->d_un.d_val & DF_1_CONFALT)
2048                                        crle = 1;
2049                                if (dyn->d_un.d_val & DF_1_DIRECT)
2050                                        FLAGS1(lmp) |= FL1_RT_DIRECT;
2051                                if (dyn->d_un.d_val & DF_1_NODEFLIB)
2052                                        FLAGS1(lmp) |= FL1_RT_NODEFLIB;
2053                                if (dyn->d_un.d_val & DF_1_ENDFILTEE)
2054                                        FLAGS1(lmp) |= FL1_RT_ENDFILTE;
2055                                if (dyn->d_un.d_val & DF_1_TRANS)
2056                                        FLAGS(lmp) |= FLG_RT_TRANS;

2058                                /*
2059                                 * Global auditing is only meaningful when
2060                                 * specified by the initiating object of the
2061                                 * process - typically the dynamic executable.
2062                                 * If this is the initiating object, its link-
2063                                 * map will not yet have been added to the
2064                                 * link-map list, and consequently the link-map
2065                                 * list is empty.  (see setup()).
2066                                 */
2067                                if (dyn->d_un.d_val & DF_1_GLOBAUDIT) {
2068                                        if (lml_main.lm_head == NULL)
2069                                                FLAGS1(lmp) |= FL1_RT_GLOBAUD;
2070                                        else
2071                                                DBG_CALL(Dbg_audit_ignore(lmp));
2072                                }

2074                                /*
2075                                 * If this object identifies itself as an
2076                                 * interposer, but relocation processing has
2077                                 * already started, then demote it.  It's too
2078                                 * late to guarantee complete interposition.
2079                                 */
2080                                /* BEGIN CSTYLED */
2081                                if (dyn->d_un.d_val &
2082                                    (DF_1_INTERPOSE | DF_1_SYMINTPOSE)) {
2083                                    if (lml->lm_flags & LML_FLG_STARTREL) {
2084                                        DBG_CALL(Dbg_util_intoolate(lmp));
2085                                        if (lml->lm_flags & LML_FLG_TRC_ENABLE)
2086                                            (void) printf(
2087                                                MSG_INTL(MSG_LDD_REL_ERR2),
2088                                                NAME(lmp));
2089                                    } else if (dyn->d_un.d_val & DF_1_INTERPOSE)
2090                                        FLAGS(lmp) |= FLG_RT_OBJINTPO;
2091                                    else
2092                                        FLAGS(lmp) |= FLG_RT_SYMINTPO;
2093                                }
2094                                /* END CSTYLED */
2095                                break;
2096                        case DT_SYMINFO:
2097                                SYMINFO(lmp) = (Syminfo *)(dyn->d_un.d_ptr +
```



```
2098                                        base);
2099                                break;
2100                        case DT_SYMINENT:
2101                                SYMINENT(lmp) = dyn->d_un.d_val;
2102                                break;
2103                        case DT_PLTPAD:
2104                                PLTPAD(lmp) = (void *)(dyn->d_un.d_ptr + base);
2105                                break;
2106                        case DT_PLTPADSZ:
2107                                pltpadsz = dyn->d_un.d_val;
2108                                break;
2109                        case DT_SUNW_RTLDINF:
2110                                /*
2111                                 * Maintain a list of RTLDINFO structures.
2112                                 * Typically, libc is the only supplier, and
2113                                 * only one structure is provided.  However,
2114                                 * multiple suppliers and multiple structures
2115                                 * are supported.  For example, one structure
2116                                 * may provide thread_init, and another
2117                                 * structure may provide atexit reservations.
2118                                 */
2119                                if ((rti = alist_append(&lml->lm_rti, NULL,
2120                                    sizeof (Rti_desc),
2121                                    AL_CNT_RTLDINFO)) == NULL) {
2122                                        remove_so(0, lmp, clmp);
2123                                        return (NULL);
2124                                }
2125                                rti->rti_lmp = lmp;
2126                                rti->rti_info = (void *)(dyn->d_un.d_ptr +
2127                                    base);
2128                                break;
2129                        case DT_SUNW_SORTENT:
2130                                SUNWSORTENT(lmp) = dyn->d_un.d_val;
2131                                break;
2132                        case DT_SUNW_SYMSORT:
2133                                SUNWSYMSORT(lmp) =
2134                                    (void *)(dyn->d_un.d_ptr + base);
2135                                break;
2136                        case DT_SUNW_SYMSORTSZ:
2137                                SUNWSYMSORTSZ(lmp) = dyn->d_un.d_val;
2138                                break;
2139                        case DT_DEPRECATED_SPARC_REGISTER:
2140                        case M_DT_REGISTER:
2141                                dip->di_flags |= FLG_DI_REGISTER;
2142                                FLAGS(lmp) |= FLG_RT_REGSYMS;
2143                                break;
2144                        case DT_SUNW_CAP:
2145                                CAP(lmp) = (void *)(dyn->d_un.d_ptr + base);
2146                                break;
2147                        case DT_SUNW_CAPINFO:
2148                                CAPINFO(lmp) = (void *)(dyn->d_un.d_ptr + base);
2149                                break;
2150                        case DT_SUNW_CAPCHAIN:
2151                                CAPCHAIN(lmp) = (void *)(dyn->d_un.d_ptr +
2152                                    base);
2153                                break;
2154                        case DT_SUNW_CAPCHAINENT:
2155                                CAPCHAINENT(lmp) = dyn->d_un.d_val;
2156                                break;
2157                        case DT_SUNW_CAPCHAINSZ:
2158                                CAPCHAINSZ(lmp) = dyn->d_un.d_val;
2159                                break;
2160                        }
2161                }

2163                /*
```

```
2164                 * Update any Dyninfo string pointers now that STRTAB() is
2165                 * known.
2166                 */
2167                for (dynndx = 0, dyn = DYN(lmp), dip = DYNINFO(lmp);
2168                    !(dip->di_flags & FLG_DI_IGNORE); dyn++, dip++) {

2170                        switch ((Xword)dyn->d_tag) {
2171                        case DT_NEEDED:
2172                        case DT_USED:
2173                        case DT_FILTER:
2174                        case DT_AUXILIARY:
2175                        case DT_SUNW_FILTER:
2176                        case DT_SUNW_AUXILIARY:
2177                                dip->di_name = STRTAB(lmp) + dyn->d_un.d_val;
2178                                break;
2179                        }
2180                }

2182                /*
2183                 * Assign any padding.
2184                 */
2185                if (PLTPAD(lmp)) {
2186                        if (pltpadsz == (Xword)0)
2187                                PLTPAD(lmp) = NULL;
2188                        else
2189                                PLTPADEND(lmp) = (void *)((Addr)PLTPAD(lmp) +
2190                                    pltpadsz);
2191                }
2192        }

2194        /*
2195         * A dynsym contains only global functions. We want to have
2196         * a version of it that also includes local functions, so that
2197         * dladdr() will be able to report names for local functions
2198         * when used to generate a stack trace for a stripped file.
2199         * This version of the dynsym is provided via DT_SUNW_SYMTAB.
2200         *
2201         * In producing DT_SUNW_SYMTAB, ld uses a non-obvious trick
2202         * in order to avoid having to have two copies of the global
2203         * symbols held in DT_SYMTAB: The local symbols are placed in
2204         * a separate section than the globals in the dynsym, but the
2205         * linker conspires to put the data for these two sections adjacent
2206         * to each other. DT_SUNW_SYMTAB points at the top of the local
2207         * symbols, and DT_SUNW_SYMSZ is the combined length of both tables.
2208         *
2209         * If the two sections are not adjacent, then something went wrong
2210         * at link time. We use ASSERT to kill the process if this is
2211         * a debug build. In a production build, we will silently ignore
2212         * the presence of the .ldynsym and proceed. We can detect this
2213         * situation by checking to see that DT_SYMTAB lies in
2214         * the range given by DT_SUNW_SYMTAB/DT_SUNW_SYMSZ.
2215         */
2216        if ((SUNWSYMTAB(lmp) != NULL) &&
2217            (((char *)SYMTAB(lmp) <= (char *)SUNWSYMTAB(lmp)) ||
2218            (((char *)SYMTAB(lmp) >=
2219            (SUNWSYMSZ(lmp) + (char *)SUNWSYMTAB(lmp)))))) {
2220                ASSERT(0);
2221                SUNWSYMTAB(lmp) = NULL;
2222                SUNWSYMSZ(lmp) = 0;
2223        }

2225        /*
2226         * If configuration file use hasn't been disabled, and a configuration
2227         * file hasn't already been set via an environment variable, see if any
2228         * application specific configuration file is specified.  An LD_CONFIG
2229         * setting is used first, but if this image was generated via crle(1)
```



```
2230         * then a default configuration file is a fall-back.
2231         */
2232        if ((!(rtld_flags & RT_FL_NOCFG)) && (config->c_name == NULL)) {
2233                if (cfile)
2234                        config->c_name = (const char *)(cfile +
2235                            (char *)STRTAB(lmp));
2236                else if (crle)
2237                        rtld_flags |= RT_FL_CONFAPP;
2238        }

2240        if (rpath)
2241                RPATH(lmp) = (char *)(rpath + (char *)STRTAB(lmp));
2242        if (fltr)
2243                REFNAME(lmp) = (char *)(fltr + (char *)STRTAB(lmp));

2245        /*
2246         * For Intel ABI compatibility.  It's possible that a JMPREL can be
2247         * specified without any other relocations (e.g. a dynamic executable
2248         * normally only contains .plt relocations).  If this is the case then
2249         * no REL, RELSZ or RELENT will have been created.  For us to be able
2250         * to traverse the .plt relocations under LD_BIND_NOW we need to know
2251         * the RELENT for these relocations.  Refer to elf_reloc() for more
2252         * details.
2253         */
2254        if (!RELENT(lmp) && JMPREL(lmp))
2255                RELENT(lmp) = sizeof (M_RELOC);

2257        /*
2258         * Establish any per-object auditing.  If we're establishing main's
2259         * link-map its too early to go searching for audit objects so just
2260         * hold the object name for later (see setup()).
2261         */
2262        if (audit) {
2263                char    *cp = audit + (char *)STRTAB(lmp);

2265                if (*cp) {
2266                        if (((AUDITORS(lmp) =
2267                            calloc(1, sizeof (Audit_desc))) == NULL) ||
2268                            ((AUDITORS(lmp)->ad_name = strdup(cp)) == NULL)) {
2269                                remove_so(0, lmp, clmp);
2270                                return (NULL);
2271                        }
2272                        if (lml_main.lm_head) {
2273                                if (audit_setup(lmp, AUDITORS(lmp), 0,
2274                                    in_nfavl) == 0) {
2275                                        remove_so(0, lmp, clmp);
2276                                        return (NULL);
2277                                }
2278                                AFLAGS(lmp) |= AUDITORS(lmp)->ad_flags;
2279                                lml->lm_flags |= LML_FLG_LOCAUDIT;
2280                        }
2281                }
2282        }

2284        if (tphdr && (tls_assign(lml, lmp, tphdr) == 0)) {
2285                remove_so(0, lmp, clmp);
2286                return (NULL);
2287        }

2289        /*
2290         * A capabilities section should be identified by a DT_SUNW_CAP entry,
2291         * and if non-empty object capabilities are included, a PT_SUNWCAP
2292         * header should reference the section.  Make sure CAP() is set
2293         * regardless.
2294         */
2295        if ((CAP(lmp) == NULL) && cap)
```

```
2296                    CAP(lmp) = cap;

2298            /*
2299             * Make sure any capabilities information or chain can be handled.
2300             */
2301            if (CAPINFO(lmp) && (CAPINFO(lmp)[0] > CAPINFO_CURRENT))
2302                    CAPINFO(lmp) = NULL;
2303            if (CAPCHAIN(lmp) && (CAPCHAIN(lmp)[0] > CAPCHAIN_CURRENT))
2304                    CAPCHAIN(lmp) = NULL;

2306            /*
2307             * As part of processing dependencies, a file descriptor is populated
2308             * with capabilities information following validation.
2309             */
2310            if (fdp->fd_flags & FLG_FD_ALTCHECK) {
2311                    FLAGS1(lmp) |= FL1_RT_ALTCHECK;
2312                    CAPSET(lmp) = fdp->fd_scapset;

2314                    if (fdp->fd_flags & FLG_FD_ALTCAP)
2315                            FLAGS1(lmp) |= FL1_RT_ALTCAP;

2317            } else if ((cap = CAP(lmp)) != NULL) {
2318                    /*
2319                     * Processing of the a.out and ld.so.1 does not involve a file
2320                     * descriptor as exec() did all the work, so capture the
2321                     * capabilities for these cases.
2322                     */
2323                    while (cap->c_tag != CA_SUNW_NULL) {
2324                            switch (cap->c_tag) {
2325                            case CA_SUNW_HW_1:
2326                                    CAPSET(lmp).sc_hw_1 = cap->c_un.c_val;
2327                                    break;
2328                            case CA_SUNW_SF_1:
2329                                    CAPSET(lmp).sc_sf_1 = cap->c_un.c_val;
2330                                    break;
2331                            case CA_SUNW_HW_2:
2332                                    CAPSET(lmp).sc_hw_2 = cap->c_un.c_val;
2333                                    break;
2334                            case CA_SUNW_PLAT:
2335                                    CAPSET(lmp).sc_plat = STRTAB(lmp) +
2336                                        cap->c_un.c_ptr;
2337                                    break;
2338                            case CA_SUNW_MACH:
2339                                    CAPSET(lmp).sc_mach = STRTAB(lmp) +
2340                                        cap->c_un.c_ptr;
2341                                    break;
2342                            }
2343                            cap++;
2344                    }
2345            }

2347            /*
2348             * If a capabilities chain table exists, duplicate it.  The chain table
2349             * is inspected for each initial call to a capabilities family lead
2350             * symbol.  From this chain, each family member is inspected to
2351             * determine the 'best' family member.  The chain table is then updated
2352             * so that the best member is immediately selected for any further
2353             * family searches.
2354             */
2355            if (CAPCHAIN(lmp)) {
2356                    Capchain        *capchain;

2358                    if ((capchain = calloc(1, CAPCHAINSZ(lmp))) == NULL)
2358                    if ((capchain = calloc(CAPCHAINSZ(lmp), 1)) == NULL)
2359                            return (NULL);
2360                    (void) memcpy(capchain, CAPCHAIN(lmp), CAPCHAINSZ(lmp));
```



```
2361                    CAPCHAIN(lmp) = capchain;
2362            }

2364            /*
2365             * Add the mapped object to the end of the link map list.
2366             */
2367            lm_append(lml, lmco, lmp);

2369            /*
2370             * Start the system loading in the ELF information we'll be processing.
2371             */
2372            if (REL(lmp)) {
2373                    (void) madvise((void *)ADDR(lmp), (uintptr_t)REL(lmp) +
2374                        (uintptr_t)RELSZ(lmp) - (uintptr_t)ADDR(lmp),
2375                        MADV_WILLNEED);
2376            }
2377            return (lmp);
2378 }
```

*unchanged_portion_omitted_*

```
**********************************************************
    8216 Mon Mar  4 02:11:17 2019
new/usr/src/cmd/sgs/rtld/common/locale.c
smatch clean rtld
**********************************************************
______unchanged_portion_omitted_


 206 /*
 207  * Two interfaces are established to support our internationalization.
 208  * gettext(3i) calls originate from all link-editor libraries, and thus the
 209  * SUNW_OST_SGS domain is assumed.  dgettext() calls originate from
 210  * dependencies such as libelf and libc.
 211  *
 212  * Presently we support two domains (libc's strerror() uses SUNW_OST_OSLIB).
 213  * If ld.so.1's dependencies evolve to require more then the 'domain' array
 214  * maintained below can be enlarged or made more dynamic in nature.
 215  */
 216 char *
 217 dgettext(const char *domain, const char *msgid)
 218 {
 219         static int      domaincnt = 0;
 220         static Domain   *domains;
 221         Domain          *_domain;
 222         int             cnt;

 224         if (glcs[CI_LCMESSAGES].lc_un.lc_val == 0)
 225                 return ((char *)msgid);

 227         /*
 228          * Determine if we've initialized any domains yet.
 229          */
 230         if (domaincnt == 0) {
 231                 if ((domains = calloc(2, sizeof (Domain))) == NULL)
 231                 if ((domains = calloc(sizeof (Domain), 2)) == NULL)
 232                         return ((char *)msgid);
 233                 domains[0].dom_name = MSG_ORIG(MSG_SUNW_OST_SGS);
 234                 domains[1].dom_name = MSG_ORIG(MSG_SUNW_OST_OSLIB);
 235                 domaincnt = 2;
 236         }

 238         /*
 239          * If this is a new locale make sure we clean up any old ones.
 240          */
 241         if (rtld_flags & RT_FL_NEWLOCALE) {
 242                 cnt = 0;

 244                 for (_domain = domains; cnt < domaincnt; _domain++, cnt++) {
 245                         if (_domain->dom_msghdr == 0)
 246                                 continue;

 248                         if (_domain->dom_msghdr != (Msghdr *)-1)
 249                                 (void) munmap((caddr_t)_domain->dom_msghdr,
 250                                     _domain->dom_msgsz);

 252                         _domain->dom_msghdr = 0;
 253                 }
 254                 rtld_flags &= ~RT_FL_NEWLOCALE;
 255         }

 257         /*
 258          * Determine which domain we need.
 259          */
 260         for (cnt = 0, _domain = domains; cnt < domaincnt; _domain++, cnt++) {
 261                 if (_domain->dom_name == domain)
 262                         break;
```



```
 263                 if (strcmp(_domain->dom_name, domain) == 0)
 264                         break;
 265         }
 266         if (cnt == domaincnt)
 267                 return ((char *)msgid);

 269         /*
 270          * Determine if the domain has been initialized yet.
 271          */
 272         if (_domain->dom_msghdr == 0)
 273                 open_mofile(_domain);
 274         if (_domain->dom_msghdr == (Msghdr *)-1)
 275                 return ((char *)msgid);

 277         return ((char *)msgid_to_msgstr(_domain->dom_msghdr, msgid));
 278 }
______unchanged_portion_omitted_
```

```
**********************************************************
   11922 Mon Mar  4 02:11:17 2019
new/usr/src/cmd/sgs/rtld/common/object.c
smatch clean rtld
**********************************************************
```

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

/*
 * Copyright (c) 1992, 2010, Oracle and/or its affiliates. All rights reserved.
 */

/*
 * Object file dependent suport for ELF objects.
 */

#include        <sys/mman.h>
#include        <stdio.h>
#include        <unistd.h>
#include        <libelf.h>
#include        <string.h>
#include        <dlfcn.h>
#include        <debug.h>
#include        <libld.h>
#include        "_rtld.h"
#include        "_audit.h"
#include        "_elf.h"

static Rt_map   *olmp = NULL;
static Alist    *mpalp = NULL;

static Ehdr     dehdr = { { ELFMAG0, ELFMAG1, ELFMAG2, ELFMAG3,
                            M_CLASS, M_DATA }, 0, M_MACH, EV_CURRENT };

/*
 * Process a relocatable object.  The static object link map pointer is used as
 * a flag to determine whether a concatenation is already in progress (ie. an
 * LD_PRELOAD may specify a list of objects).  The link map returned simply
 * specifies an 'object' flag which the caller can interpret and thus call
 * elf_obj_fini() to complete the concatenation.
 */
static Rt_map *
elf_obj_init(Lm_list *lml, Aliste lmco, const char *oname)
{
        Ofl_desc        *ofl;
        const char      *name;
        size_t          lmsz;
```



```
        /*
         * Allocate the name of this object, as the original name may be
         * associated with a data buffer that can be reused to load the
         * dependencies needed to processes this object.
         */
        if ((name = stravl_insert(oname, 0, 0, 0)) == NULL)
                return (NULL);

        /*
         * Initialize an output file descriptor and the entrance criteria.
         */
        if ((ofl = calloc(1, sizeof (Ofl_desc))) == NULL)
        if ((ofl = calloc(sizeof (Ofl_desc), 1)) == NULL)
                return (NULL);

        ofl->ofl_dehdr = &dehdr;

        ofl->ofl_flags = (FLG_OF_DYNAMIC | FLG_OF_SHAROBJ | FLG_OF_STRIP);
        ofl->ofl_flags1 = (FLG_OF1_RELDYN | FLG_OF1_TEXTOFF | FLG_OF1_MEMORY);
        ofl->ofl_lml = lml;

        /*
         * As ent_setup() will effectively lazy load the necessary support
         * libraries, make sure ld.so.1 is initialized for plt relocations.
         * Then configure libld.so to process objects of the desired target
         * type (this is the first call to libld.so, which will effectively
         * lazyload it).
         */
        if ((elf_rtld_load() == 0) || (ld_init_target(lml, M_MACH) != 0)) {
                free(ofl);
                return (NULL);
        }

        /*
         * Obtain a generic set of entrance criteria, and generate a link map
         * place holder and use the ELFPRV() element to maintain the output
         * file descriptor.
         */
        lmsz = S_DROUND(sizeof (Rt_map)) + sizeof (Rt_elfp);
        if ((ld_ent_setup(ofl, syspagsz) == S_ERROR) ||
            ((olmp = calloc(1, sizeof (Rt_map))) == NULL)) {
            ((olmp = calloc(lmsz, 1)) == NULL)) {
                free(ofl);
                return (NULL);
        }

        DBG_CALL(Dbg_file_elf(lml, name, 0, 0, lml->lm_lmidstr, lmco));
        FLAGS(olmp) |= FLG_RT_OBJECT;
        ELFPRV(olmp) = (void *)ofl;

        /*
         * Initialize string tables.
         */
        if (ld_init_strings(ofl) == S_ERROR) {
                free(ofl);
                free(olmp);
                olmp = NULL;
                return (NULL);
        }

        /*
         * Assign the output file name to be the initial object that got us
         * here.  This name is being used for diagnostic purposes only as we
         * don't actually generate an output file unless debugging is enabled.
         */
        ofl->ofl_name = name;
```

```
124         NAME(olmp) = (char *)name;
125         LIST(olmp) = lml;

127         lm_append(lml, lmco, olmp);
128         return (olmp);
129 }
```

unchanged_portion_omitted_

```
225 /*
226  * Finish relocatable object processing.  Having already initially processed one
227  * or more objects, complete the generation of a shared object image by calling
228  * the appropriate link-edit functionality (refer to sgs/ld/common/main.c).
229  */
230 Rt_map *
231 elf_obj_fini(Lm_list *lml, Rt_map *lmp, Rt_map *clmp, int *in_nfavl)
232 {
233         Ofl_desc                *ofl = (Ofl_desc *)ELFPRV(lmp);
234         Rt_map                  *nlmp, *tlmp;
235         Ehdr                    *ehdr;
236         Phdr                    *phdr;
237         mmapobj_result_t        *mpp, *hmpp;
238         uint_t                  phnum;
239         int                     mnum;
240         Lm_cntl                 *lmc;
241         Aliste                  idx1;
242         Mmap_desc               *mdp;
243         Fdesc                   fd = { 0 };
244         Grp_hdl                 *ghp;
245         Rej_desc                rej = { 0 };
246         Syscapset               *scapset;
247         elfcap_mask_t           omsk;
248         Alist                   *oalp;

250         DBG_CALL(Dbg_util_nl(lml, DBG_NL_STD));

252         if (ld_reloc_init(ofl) == S_ERROR)
253                 return (NULL);
254         if (ld_sym_validate(ofl) == S_ERROR)
255                 return (NULL);

257         /*
258          * At this point, all input section processing is complete.  If any
259          * capabilities have been established, ensure that they are appropriate
260          * for this system.
261          */
262         if (pnavl_recorded(&capavl, ofl->ofl_name, NULL, NULL))
263                 scapset = alt_scapset;
264         else
265                 scapset = org_scapset;

267         if ((((omsk = ofl->ofl_ocapset.oc_hw_1.cm_val) != 0) &&
268             (hwcap1_check(scapset, omsk, &rej) == 0)) ||
269             (((omsk = ofl->ofl_ocapset.oc_sf_1.cm_val) != 0) &&
270             (sfcap1_check(scapset, omsk, &rej) == 0)) ||
271             (((omsk = ofl->ofl_ocapset.oc_hw_2.cm_val) != 0) &&
272             (hwcap2_check(scapset, omsk, &rej) == 0)) ||
273             (((oalp = ofl->ofl_ocapset.oc_plat.cl_val) != NULL) &&
274             (check_plat_names(scapset, oalp, &rej) == 0)) ||
275             (((oalp = ofl->ofl_ocapset.oc_mach.cl_val) != NULL) &&
276             (check_mach_names(scapset, oalp, &rej) == 0))) {
277                 if ((lml_main.lm_flags & LML_FLG_TRC_LDDSTUB) &&
279                 if ((lml_main.lm_flags & LML_FLG_TRC_LDDSTUB) && lmp &&
278                     (FLAGS1(lmp) & FL1_RT_LDDSTUB) && (NEXT(lmp) == NULL)) {
279                         /* LINTED */
280                         (void) printf(MSG_INTL(ldd_reject[rej.rej_type]),
281                             ofl->ofl_name, rej.rej_str);
```



```
282                 }
283                 return (NULL);
284         }

286         /*
287          * Finish creating the output file.
288          */
289         if (ld_make_sections(ofl) == S_ERROR)
290                 return (NULL);
291         if (ld_create_outfile(ofl) == S_ERROR)
292                 return (NULL);
293         if (ld_update_outfile(ofl) == S_ERROR)
294                 return (NULL);
295         if (ld_reloc_process(ofl) == S_ERROR)
296                 return (NULL);

298         /*
299          * At this point we have a memory image of the shared object.  The link
300          * editor would normally simply write this to the required output file.
301          * If we're debugging generate a standard temporary output file.
302          */
303         DBG_CALL(Dbg_file_output(ofl));

305         /*
306          * Allocate a mapping array to retain mapped segment information.
307          */
308         ehdr = ofl->ofl_nehdr;
309         phdr = ofl->ofl_phdr;

311         if ((mpp = hmpp = calloc(ehdr->e_phnum,
312             sizeof (mmapobj_result_t))) == NULL)
313                 return (NULL);
314         for (mnum = 0, phnum = 0; phnum < ehdr->e_phnum; phnum++) {
315                 if (phdr[phnum].p_type != PT_LOAD)
316                         continue;

318                 mpp[mnum].mr_addr = (caddr_t)((uintptr_t)phdr[phnum].p_vaddr +
319                     (uintptr_t)ehdr);
320                 mpp[mnum].mr_msize = phdr[phnum].p_memsz;
321                 mpp[mnum].mr_fsize = phdr[phnum].p_filesz;
322                 mpp[mnum].mr_prot = (PROT_READ | PROT_WRITE | PROT_EXEC);
323                 mnum++;
324         }

326         /*
327          * Generate a new link map representing the memory image created.
328          */
329         fd.fd_nname = ofl->ofl_name;
330         if ((nlmp = elf_new_lmp(lml, CNTL(olmp), &fd, (Addr)hmpp->mr_addr,
331             ofl->ofl_size, NULL, clmp, in_nfavl)) == NULL)
332                 return (NULL);

334         MMAPS(nlmp) = hmpp;
335         MMAPCNT(nlmp) = mnum;
336         PADSTART(nlmp) = (ulong_t)hmpp->mr_addr;
337         PADIMLEN(nlmp) = mpp->mr_addr + mpp->mr_msize - hmpp->mr_addr;

339         /*
340          * Replace the original (temporary) link map with the new link map.
341          */
342         /* LINTED */
343         lmc = (Lm_cntl *)alist_item_by_offset(lml->lm_lists, CNTL(nlmp));
344         lml->lm_obj--;

346         if ((tlmp = PREV_RT_MAP(nlmp)) == olmp)
347                 tlmp = nlmp;
```

```
349         if (PREV(olmp)) {
350                 NEXT(PREV_RT_MAP(olmp)) = (Link_map *)nlmp;
351                 PREV(nlmp) = PREV(olmp);
352         } else {
353                 PREV(nlmp) = NULL;
354                 lmc->lc_head = nlmp;
355                 if (CNTL(nlmp) == ALIST_OFF_DATA)
356                         lml->lm_head = nlmp;
357         }

359         if (NEXT(olmp) != (Link_map *)nlmp) {
360                 NEXT(nlmp) = NEXT(olmp);
361                 PREV(NEXT_RT_MAP(olmp)) = (Link_map *)nlmp;
362         }

364         NEXT(tlmp) = NULL;

366         lmc->lc_tail = tlmp;
367         if (CNTL(nlmp) == ALIST_OFF_DATA)
368                 lml->lm_tail = tlmp;

370         HANDLES(nlmp) = HANDLES(olmp);
371         GROUPS(nlmp) = GROUPS(olmp);
372         STDEV(nlmp) = STDEV(olmp);
373         STINO(nlmp) = STINO(olmp);

375         FLAGS(nlmp) |= ((FLAGS(olmp) & ~FLG_RT_OBJECT) | FLG_RT_IMGALLOC);
376         FLAGS1(nlmp) |= FLAGS1(olmp);
377         MODE(nlmp) |= MODE(olmp);

379         NAME(nlmp) = NAME(olmp);

381         /*
382          * Reassign any original handles to the new link-map.
383          */
384         for (APLIST_TRAVERSE(HANDLES(nlmp), idx1, ghp)) {
385                 Grp_desc        *gdp;
386                 Aliste          idx2;

388                 ghp->gh_ownlmp = nlmp;

390                 for (ALIST_TRAVERSE(ghp->gh_depends, idx2, gdp)) {
391                         if (gdp->gd_depend == olmp) {
392                                 gdp->gd_depend = nlmp;
393                                 break;
394                         }
395                 }
396         }

398         ld_ofl_cleanup(ofl);
399         free(ELFPRV(olmp));
400         free(olmp);
401         olmp = 0;

403         /*
404          * Unmap the original relocatable object.
405          */
406         for (ALIST_TRAVERSE(mpalp, idx1, mdp)) {
407                 unmap_obj(mdp->md_mpp, mdp->md_mnum);
408                 free(mdp->md_mpp);
409         }
410         free(mpalp);
411         mpalp = NULL;

413         /*
```



```
414          * Now that we've allocated our permanent link map structure, expand the
415          * PATHNAME() and insert this path name into the FullPathNode AVL tree.
416          */
417         (void) fullpath(nlmp, 0);
418         if (fpavl_insert(lml, nlmp, PATHNAME(nlmp), 0) == 0)
419                 return (NULL);

421         /*
422          * If we're being audited tell the audit library of the file we've just
423          * opened.
424          */
425         if ((lml->lm_tflags | AFLAGS(nlmp)) & LML_TFLG_AUD_MASK) {
426                 if (audit_objopen(nlmp, nlmp) == 0)
427                         return (NULL);
428         }
429         return (nlmp);
430 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    8939 Mon Mar  4 02:11:18 2019
new/usr/src/cmd/sgs/rtld/common/tls.c
smatch clean rtld
**********************************************************
______unchanged_portion_omitted_

 255 int
 256 tls_statmod(Lm_list *lml, Rt_map *lmp)
 257 {
 258         uint_t          tlsmodndx, tlsmodcnt = lml->lm_tls;
 259         TLS_modinfo     **tlsmodlist, *tlsbuflist;
 260         Phdr            *tlsphdr;
 261         int             (*fptr)(TLS_modinfo **, ulong_t);

 263         fptr = lml->lm_lcs[CI_TLS_STATMOD].lc_un.lc_func;

 265         /*
 266          * Allocate a buffer to report the TLS modules, the buffer consists of:
 267          *
 268          *      TLS_modinfo *   ptrs[tlsmodcnt + 1]
 269          *      TLS_modinfo     bufs[tlsmodcnt]
 270          *
 271          * The ptrs are initialized to the bufs - except the last one which
 272          * null terminates the array.
 273          *
 274          * Note, even if no TLS has yet been observed, we still supply a
 275          * TLS buffer with a single null entry.  This allows us to initialize
 276          * the backup TLS reservation.
 277          */
 278         if ((tlsmodlist = calloc(1, (sizeof (TLS_modinfo *) * (tlsmodcnt + 1)) +
 279             (sizeof (TLS_modinfo) * tlsmodcnt))) == NULL)
 278         if ((tlsmodlist = calloc((sizeof (TLS_modinfo *) * (tlsmodcnt + 1)) +
 279             (sizeof (TLS_modinfo) * tlsmodcnt), 1)) == NULL)
 280                 return (0);

 282         lml->lm_tls = 0;

 284         /*
 285          * If we don't have any TLS modules - report that and return.
 286          */
 287         if (tlsmodcnt == 0) {
 288                 if (fptr != NULL)
 289                         (void) (*fptr)(tlsmodlist, tls_static_resv);
 290                 DBG_CALL(Dbg_tls_static_block(&lml_main, 0, 0,
 291                     tls_static_resv));
 292                 return (1);
 293         }

 295         /*
 296          * Initialize the TLS buffer.
 297          */
 298         tlsbuflist = (TLS_modinfo *)((uintptr_t)tlsmodlist +
 299             ((tlsmodcnt + 1) * sizeof (TLS_modinfo *)));

 301         for (tlsmodndx = 0; tlsmodndx < tlsmodcnt; tlsmodndx++)
 302                 tlsmodlist[tlsmodndx] = &tlsbuflist[tlsmodndx];

 304         /*
 305          * Account for the initial dtv ptr in the TLSSIZE calculation.
 306          */
 307         tlsmodndx = 0;
 308         for (lmp = lml->lm_head; lmp; lmp = NEXT_RT_MAP(lmp)) {
 309                 if (THIS_IS_NOT_ELF(lmp) ||
 310                     (PTTLS(lmp) == 0) || (PTTLS(lmp)->p_memsz == 0))
 311                         continue;
```



```
 313                 tlsphdr = PTTLS(lmp);

 315                 tlsmodlist[tlsmodndx]->tm_modname = PATHNAME(lmp);
 316                 tlsmodlist[tlsmodndx]->tm_modid = TLSMODID(lmp);
 317                 tlsmodlist[tlsmodndx]->tm_tlsblock = (void *)(tlsphdr->p_vaddr);

 319                 if (!(FLAGS(lmp) & FLG_RT_FIXED)) {
 320                         tlsmodlist[tlsmodndx]->tm_tlsblock = (void *)
 321                             ((uintptr_t)tlsmodlist[tlsmodndx]->tm_tlsblock +
 322                             ADDR(lmp));
 323                 }
 324                 tlsmodlist[tlsmodndx]->tm_filesz = tlsphdr->p_filesz;
 325                 tlsmodlist[tlsmodndx]->tm_memsz = tlsphdr->p_memsz;
 326                 tlsmodlist[tlsmodndx]->tm_flags = TM_FLG_STATICTLS;
 327                 tlsmodlist[tlsmodndx]->tm_stattlsoffset = TLSSTATOFF(lmp);
 328                 tlsmodndx++;
 329         }

 331         DBG_CALL(Dbg_tls_static_block(&lml_main, (void *)tlsmodlist,
 332             tls_static_size, tls_static_resv));
 333         (void) (*fptr)(tlsmodlist, (tls_static_size + tls_static_resv));

 335         /*
 336          * We're done with the list - clean it up.
 337          */
 338         free(tlsmodlist);
 339         return (1);
 340 }
______unchanged_portion_omitted_
```

```
**********************************************************
  101245 Mon Mar  4 02:11:18 2019
new/usr/src/cmd/sgs/rtld/common/util.c
smatch clean rtld
**********************************************************
______unchanged_portion_omitted_

 371 /*
 372  * Insert a name into the FullPathNode AVL tree for the link-map list.  The
 373  * objects NAME() is the path that would have originally been searched for, and
 374  * is therefore the name to associate with any "where" value.  If the object has
 375  * a different PATHNAME(), perhaps because it has resolved to a different file
 376  * (see fullpath()), then this name will be recorded as a separate FullPathNode
 377  * (see load_file()).
 378  */
 379 int
 380 fpavl_insert(Lm_list *lml, Rt_map *lmp, const char *name, avl_index_t where)
 381 {
 382         FullPathNode    *fpnp;
 383         uint_t          hash = sgs_str_hash(name);

 385         if (where == 0) {
 386                 /* LINTED */
 387                 Rt_map  *_lmp = fpavl_recorded(lml, name, hash, &where);

 389                 /*
 390                  * We better not get a hit now, we do not want duplicates in
 391                  * the tree.
 392                  */
 393                 ASSERT(_lmp == NULL);
 394         }

 396         /*
 397          * Insert new node in tree.
 398          */
 399         if ((fpnp = calloc(1, sizeof (FullPathNode))) == NULL)
 399         if ((fpnp = calloc(sizeof (FullPathNode), 1)) == NULL)
 400                 return (0);

 402         fpnp->fpn_node.pn_name = name;
 403         fpnp->fpn_node.pn_hash = hash;
 404         fpnp->fpn_lmp = lmp;

 406         if (aplist_append(&FPNODE(lmp), fpnp, AL_CNT_FPNODE) == NULL) {
 407                 free(fpnp);
 408                 return (0);
 409         }

 411         ASSERT(lml->lm_fpavl != NULL);
 412         avl_insert(lml->lm_fpavl, fpnp, where);
 413         return (1);
 414 }
______unchanged_portion_omitted_

 433 /*
 434  * Insert a path name into the not-found AVL tree.
 435  *
 436  * This tree maintains a node for each path name that ld.so.1 has explicitly
 437  * inspected, but has failed to load during a single ld.so.1 operation.  If the
 438  * path name does not exist in this AVL tree, then the next insertion point is
 439  * deposited in "where".  This value can be used by nfavl_insert() to expedite
 440  * the insertion.
 441  */
 442 void
 443 nfavl_insert(const char *name, avl_index_t where)
 444 {
```



```
 445         PathNode        *pnp;
 446         uint_t          hash = sgs_str_hash(name);

 448         if (where == 0) {
 449                 /* LINTED */
 450                 int     in_nfavl = pnavl_recorded(&nfavl, name, hash, &where);

 452                 /*
 453                  * We better not get a hit now, we do not want duplicates in
 454                  * the tree.
 455                  */
 456                 ASSERT(in_nfavl == 0);
 457         }

 459         /*
 460          * Insert new node in tree.
 461          */
 462         if ((pnp = calloc(1, sizeof (PathNode))) != NULL) {
 462         if ((pnp = calloc(sizeof (PathNode), 1)) != NULL) {
 463                 pnp->pn_name = name;
 464                 pnp->pn_hash = hash;
 465                 avl_insert(nfavl, pnp, where);
 466         }
 467 }

 469 /*
 470  * Insert the directory name, of a full path name, into the secure path AVL
 471  * tree.
 472  *
 473  * This tree is used to maintain a list of directories in which the dependencies
 474  * of a secure process have been found.  This list provides a fall-back in the
 475  * case that a $ORIGIN expansion is deemed insecure, when the expansion results
 476  * in a path name that has already provided dependencies.
 477  */
 478 void
 479 spavl_insert(const char *name)
 480 {
 481         char            buffer[PATH_MAX], *str;
 482         size_t          size;
 483         avl_index_t     where;
 484         PathNode        *pnp;
 485         uint_t          hash;

 487         /*
 488          * Separate the directory name from the path name.
 489          */
 490         if ((str = strrchr(name, '/')) == name)
 491                 size = 1;
 492         else
 493                 size = str - name;

 495         (void) strncpy(buffer, name, size);
 496         buffer[size] = '\0';
 497         hash = sgs_str_hash(buffer);

 499         /*
 500          * Determine whether this directory name is already recorded, or if
 501          * not, 'where" will provide the insertion point for the new string.
 502          */
 503         if (pnavl_recorded(&spavl, buffer, hash, &where))
 504                 return;

 506         /*
 507          * Insert new node in tree.
 508          */
 509         if ((pnp = calloc(1, sizeof (PathNode))) != NULL) {
```

```
 509         if ((pnp = calloc(sizeof (PathNode), 1)) != NULL) {
 510                 pnp->pn_name = strdup(buffer);
 511                 pnp->pn_hash = hash;
 512                 avl_insert(spavl, pnp, where);
 513         }
 514 }
_____unchanged_portion_omitted_

2872 static char     errbuf[ERRSIZE], *nextptr = errbuf, *prevptr = NULL;

2874 /*
2875  * All error messages go through eprintf().  During process initialization,
2876  * these messages are directed to the standard error, however once control has
2877  * been passed to the applications code these messages are stored in an internal
2878  * buffer for use with dlerror().  Note, fatal error conditions that may occur
2879  * while running the application will still cause a standard error message, see
2880  * rtldexit() in this file for details.
2881  * The RT_FL_APPLIC flag serves to indicate the transition between process
2882  * initialization and when the applications code is running.
2883  */
2884 void
2885 veprintf(Lm_list *lml, Error error, const char *format, va_list args)
2886 {
2887         int             overflow = 0;
2888         static int      lock = 0;
2889         Prfbuf          prf;

2891         if (lock || (nextptr > (errbuf + (ERRSIZE - 1))))
2891         if (lock || (nextptr == (errbuf + ERRSIZE)))
2892                 return;

2894         /*
2895          * Note: this lock is here to prevent the same thread from recursively
2896          * entering itself during a eprintf.  ie: during eprintf malloc() fails
2897          * and we try and call eprintf ... and then malloc() fails ....
2898          */
2899         lock = 1;

2901         /*
2902          * If we have completed startup initialization, all error messages
2903          * must be saved.  These are reported through dlerror().  If we're
2904          * still in the initialization stage, output the error directly and
2905          * add a newline.
2906          */
2907         prf.pr_buf = prf.pr_cur = nextptr;
2908         prf.pr_len = ERRSIZE - (nextptr - errbuf);

2910         if ((rtld_flags & RT_FL_APPLIC) == 0)
2911                 prf.pr_fd = 2;
2912         else
2913                 prf.pr_fd = -1;

2915         if (error > ERR_NONE) {
2916                 if ((error == ERR_FATAL) && (rtld_flags2 & RT_FL2_FTL2WARN))
2917                         error = ERR_WARNING;
2918                 switch (error) {
2919                 case ERR_WARNING_NF:
2920                         if (err_strs[ERR_WARNING_NF] == NULL)
2921                                 err_strs[ERR_WARNING_NF] =
2922                                     MSG_INTL(MSG_ERR_WARNING);
2923                         break;
2924                 case ERR_WARNING:
2925                         if (err_strs[ERR_WARNING] == NULL)
2926                                 err_strs[ERR_WARNING] =
2927                                     MSG_INTL(MSG_ERR_WARNING);
2928                         break;
```



```
2929                 case ERR_GUIDANCE:
2930                         if (err_strs[ERR_GUIDANCE] == NULL)
2931                                 err_strs[ERR_GUIDANCE] =
2932                                     MSG_INTL(MSG_ERR_GUIDANCE);
2933                         break;
2934                 case ERR_FATAL:
2935                         if (err_strs[ERR_FATAL] == NULL)
2936                                 err_strs[ERR_FATAL] = MSG_INTL(MSG_ERR_FATAL);
2937                         break;
2938                 case ERR_ELF:
2939                         if (err_strs[ERR_ELF] == NULL)
2940                                 err_strs[ERR_ELF] = MSG_INTL(MSG_ERR_ELF);
2941                         break;
2942                 }
2943                 if (procname) {
2944                         if (bufprint(&prf, MSG_ORIG(MSG_STR_EMSGFOR1),
2945                             rtldname, procname, err_strs[error]) == 0)
2946                                 overflow = 1;
2947                 } else {
2948                         if (bufprint(&prf, MSG_ORIG(MSG_STR_EMSGFOR2),
2949                             rtldname, err_strs[error]) == 0)
2950                                 overflow = 1;
2951                 }
2952                 if (overflow == 0) {
2953                         /*
2954                          * Remove the terminating '\0'.
2955                          */
2956                         prf.pr_cur--;
2957                 }
2958         }

2960         if ((overflow == 0) && doprf(format, args, &prf) == 0)
2961                 overflow = 1;

2963         /*
2964          * If this is an ELF error, it will have been generated by a support
2965          * object that has a dependency on libelf.  ld.so.1 doesn't generate any
2966          * ELF error messages as it doesn't interact with libelf.  Determine the
2967          * ELF error string.
2968          */
2969         if ((overflow == 0) && (error == ERR_ELF)) {
2970                 static int              (*elfeno)() = 0;
2971                 static const char       *(*elfemg)();
2972                 const char              *emsg;
2973                 Rt_map                  *dlmp, *lmp = lml_rtld.lm_head;

2975                 if (NEXT(lmp) && (elfeno == 0)) {
2976                         if (((elfemg = (const char *(*)())dlsym_intn(RTLD_NEXT,
2977                             MSG_ORIG(MSG_SYM_ELFERRMSG),
2978                             lmp, &dlmp)) == NULL) ||
2979                             ((elfeno = (int (*)())dlsym_intn(RTLD_NEXT,
2980                             MSG_ORIG(MSG_SYM_ELFERRNO), lmp, &dlmp)) == NULL))
2981                                 elfeno = 0;
2982                 }

2984                 /*
2985                  * Lookup the message; equivalent to elf_errmsg(elf_errno()).
2986                  */
2987                 if (elfeno && ((emsg = (* elfemg)((* elfeno)())) != NULL)) {
2988                         prf.pr_cur--;
2989                         if (bufprint(&prf, MSG_ORIG(MSG_STR_EMSGFOR2),
2990                             emsg) == 0)
2991                                 overflow = 1;
2992                 }
2993         }
```

```
2995         /*
2996          * Push out any message that's been built.  Note, in the case of an
2997          * overflow condition, this message may be incomplete, in which case
2998          * make sure any partial string is null terminated.
2999          */
3000         if ((rtld_flags & (RT_FL_APPLIC | RT_FL_SILENCERR)) == 0) {
3001                 *(prf.pr_cur - 1) = '\n';
3002                 (void) dowrite(&prf);
3003         }
3004         if (overflow)
3005                 *(prf.pr_cur - 1) = '\0';

3007         DBG_CALL(Dbg_util_str(lml, nextptr));

3009         /*
3010          * Determine if there was insufficient space left in the buffer to
3011          * complete the message.  If so, we'll have printed out as much as had
3012          * been processed if we're not yet executing the application.
3013          * Otherwise, there will be some debugging diagnostic indicating
3014          * as much of the error message as possible.  Write out a final buffer
3015          * overflow diagnostic - unlocalized, so we don't chance more errors.
3016          */
3017         if (overflow) {
3018                 char    *str = (char *)MSG_INTL(MSG_EMG_BUFOVRFLW);

3020                 if ((rtld_flags & RT_FL_SILENCERR) == 0) {
3021                         lasterr = str;

3023                         if ((rtld_flags & RT_FL_APPLIC) == 0) {
3024                                 (void) write(2, str, strlen(str));
3025                                 (void) write(2, MSG_ORIG(MSG_STR_NL),
3026                                     MSG_STR_NL_SIZE);
3027                         }
3028                 }
3029                 DBG_CALL(Dbg_util_str(lml, str));

3031                 lock = 0;
3032                 nextptr = errbuf + ERRSIZE;
3033                 return;
3034         }

3036         /*
3037          * If the application has started, then error messages are being saved
3038          * for retrieval by dlerror(), or possible flushing from rtldexit() in
3039          * the case of a fatal error.  In this case, establish the next error
3040          * pointer.  If we haven't started the application, the whole message
3041          * buffer can be reused.
3042          */
3043         if ((rtld_flags & RT_FL_SILENCERR) == 0) {
3044                 lasterr = nextptr;

3046                 /*
3047                  * Note, should we encounter an error such as ENOMEM, there may
3048                  * be a number of the same error messages (ie. an operation
3049                  * fails with ENOMEM, and then the attempts to construct the
3050                  * error message itself, which incurs additional ENOMEM errors).
3051                  * Compare any previous error message with the one we've just
3052                  * created to prevent any duplication clutter.
3053                  */
3054                 if ((rtld_flags & RT_FL_APPLIC) &&
3055                     ((prevptr == NULL) || (strcmp(prevptr, nextptr) != 0))) {
3056                         prevptr = nextptr;
3057                         nextptr = prf.pr_cur;
3058                         *nextptr = '\0';
3059                 }
3060         }
```



```
3061         lock = 0;
3062 }
______unchanged_portion_omitted_

3097 /*
3098  * Exit.  If we arrive here with a non zero status it's because of a fatal
3099  * error condition (most commonly a relocation error).  If the application has
3100  * already had control, then the actual fatal error message will have been
3101  * recorded in the dlerror() message buffer.  Print the message before really
3102  * exiting.
3103  */
3104 void
3105 rtldexit(Lm_list * lml, int status)
3106 {
3107         if (status) {
3108                 if (rtld_flags & RT_FL_APPLIC) {
3109                         /*
3110                          * If the error buffer has been used, write out all
3111                          * pending messages - lasterr is simply a pointer to
3112                          * the last message in this buffer.  However, if the
3113                          * buffer couldn't be created at all, lasterr points
3114                          * to a constant error message string.
3115                          */
3116                         if (*errbuf) {
3117                                 char    *errptr = errbuf;
3118                                 char    *errend = errbuf + ERRSIZE;

3120                                 while ((errptr < errend) && *errptr) {
3121                                         size_t  size = strlen(errptr);
3122                                         (void) write(2, errptr, size);
3123                                         (void) write(2, MSG_ORIG(MSG_STR_NL),
3124                                             MSG_STR_NL_SIZE);
3125                                         errptr += (size + 1);
3126                                 }
3127                         }
3128                         if (lasterr && ((lasterr < errbuf) ||
3129                             (lasterr > (errbuf + (ERRSIZE - 1))))) {
3129                             (lasterr > (errbuf + ERRSIZE))) {
3130                                 (void) write(2, lasterr, strlen(lasterr));
3131                                 (void) write(2, MSG_ORIG(MSG_STR_NL),
3132                                     MSG_STR_NL_SIZE);
3133                         }
3134                 }
3135                 leave(lml, 0);
3136                 (void) _lwp_kill(_lwp_self(), killsig);
3137         }
3138         _exit(status);
3139 }
______unchanged_portion_omitted_
```

```
**********************************************************
   27141 Mon Mar  4 02:11:19 2019
new/usr/src/cmd/sgs/rtld/i386/i386_elf.c
smatch clean rtld
**********************************************************
______unchanged_portion_omitted_

 178 /*
 179  * Function binding routine - invoked on the first call to a function through
 180  * the procedure linkage table;
 181  * passes first through an assembly language interface.
 182  *
 183  * Takes the offset into the relocation table of the associated
 184  * relocation entry and the address of the link map (rt_private_map struct)
 185  * for the entry.
 186  *
 187  * Returns the address of the function referenced after re-writing the PLT
 188  * entry to invoke the function directly.
 189  *
 190  * On error, causes process to terminate with a signal.
 191  */
 192 ulong_t
 193 elf_bndr(Rt_map *lmp, ulong_t reloff, caddr_t from)
 194 {
 195         Rt_map          *nlmp, *llmp;
 196         ulong_t         addr, symval, rsymndx;
 197         char            *name;
 198         Rel             *rptr;
 199         Sym             *rsym, *nsym;
 200         uint_t          binfo, sb_flags = 0, dbg_class;
 201         Slookup         sl;
 202         Sresult         sr;
 203         int             entry, lmflags;
 204         Lm_list         *lml;

 206         /*
 207          * For compatibility with libthread (TI_VERSION 1) we track the entry
 208          * value.  A zero value indicates we have recursed into ld.so.1 to
 209          * further process a locking request.  Under this recursion we disable
 210          * tsort and cleanup activities.
 211          */
 212         entry = enter(0);

 214         lml = LIST(lmp);
 215         if ((lmflags = lml->lm_flags) & LML_FLG_RTLDLM) {
 216                 dbg_class = dbg_desc->d_class;
 217                 dbg_desc->d_class = 0;
 218         }

 220         /*
 221          * Perform some basic sanity checks.  If we didn't get a load map or
 222          * the relocation offset is invalid then its possible someone has walked
 223          * over the .got entries or jumped to plt0 out of the blue.
 224          */
 225         if ((reloff % sizeof (Rel)) != 0) {
 225         if (!lmp || ((reloff % sizeof (Rel)) != 0)) {
 226                 Conv_inv_buf_t inv_buf;

 228                 eprintf(lml, ERR_FATAL, MSG_INTL(MSG_REL_PLTREF),
 229                     conv_reloc_386_type(R_386_JMP_SLOT, 0, &inv_buf),
 230                     EC_NATPTR(lmp), EC_XWORD(reloff), EC_NATPTR(from));
 231                 rtldexit(lml, 1);
 232         }

 234         /*
 235          * Use relocation entry to get symbol table entry and symbol name.
```



```
 236          */
 237         addr = (ulong_t)JMPREL(lmp);
 238         rptr = (Rel *)(addr + reloff);
 239         rsymndx = ELF_R_SYM(rptr->r_info);
 240         rsym = (Sym *)((ulong_t)SYMTAB(lmp) + (rsymndx * SYMENT(lmp)));
 241         name = (char *)(STRTAB(lmp) + rsym->st_name);

 243         /*
 244          * Determine the last link-map of this list, this'll be the starting
 245          * point for any tsort() processing.
 246          */
 247         llmp = lml->lm_tail;

 249         /*
 250          * Find definition for symbol.  Initialize the symbol lookup, and
 251          * symbol result, data structures.
 252          */
 253         SLOOKUP_INIT(sl, name, lmp, lml->lm_head, ld_entry_cnt, 0,
 254             rsymndx, rsym, 0, LKUP_DEFT);
 255         SRESULT_INIT(sr, name);

 257         if (lookup_sym(&sl, &sr, &binfo, NULL) == 0) {
 258                 eprintf(lml, ERR_FATAL, MSG_INTL(MSG_REL_NOSYM), NAME(lmp),
 259                     demangle(name));
 260                 rtldexit(lml, 1);
 261         }

 263         name = (char *)sr.sr_name;
 264         nlmp = sr.sr_dmap;
 265         nsym = sr.sr_sym;

 267         symval = nsym->st_value;

 269         if (!(FLAGS(nlmp) & FLG_RT_FIXED) &&
 270             (nsym->st_shndx != SHN_ABS))
 271                 symval += ADDR(nlmp);
 272         if ((lmp != nlmp) && ((FLAGS1(nlmp) & FL1_RT_NOINIFIN) == 0)) {
 273                 /*
 274                  * Record that this new link map is now bound to the caller.
 275                  */
 276                 if (bind_one(lmp, nlmp, BND_REFER) == 0)
 277                         rtldexit(lml, 1);
 278         }

 280         if ((lml->lm_tflags | AFLAGS(lmp) | AFLAGS(nlmp)) &
 281             LML_TFLG_AUD_SYMBIND) {
 282                 uint_t  symndx = (((uintptr_t)nsym -
 283                     (uintptr_t)SYMTAB(nlmp)) / SYMENT(nlmp));
 284                 symval = audit_symbind(lmp, nlmp, nsym, symndx, symval,
 285                     &sb_flags);
 286         }

 288         if (!(rtld_flags & RT_FL_NOBIND)) {
 289                 addr = rptr->r_offset;
 290                 if (!(FLAGS(lmp) & FLG_RT_FIXED))
 291                         addr += ADDR(lmp);
 292                 if (((lml->lm_tflags | AFLAGS(lmp)) &
 293                     (LML_TFLG_AUD_PLTENTER | LML_TFLG_AUD_PLTEXIT)) &&
 294                     AUDINFO(lmp)->ai_dynplts) {
 295                         int     fail = 0;
 296                         uint_t  pltndx = reloff / sizeof (Rel);
 297                         uint_t  symndx = (((uintptr_t)nsym -
 298                             (uintptr_t)SYMTAB(nlmp)) / SYMENT(nlmp));

 300                         symval = (ulong_t)elf_plt_trace_write(addr, lmp, nlmp,
 301                             nsym, symndx, pltndx, (caddr_t)symval, sb_flags,
```

```
302                             &fail);
303                         if (fail)
304                                 rtldexit(lml, 1);
305                 } else {
306                         /*
307                          * Write standard PLT entry to jump directly
308                          * to newly bound function.
309                          */
310                         *(ulong_t *)addr = symval;
311                 }
312         }

314         /*
315          * Print binding information and rebuild PLT entry.
316          */
317         DBG_CALL(Dbg_bind_global(lmp, (Addr)from, (Off)(from - ADDR(lmp)),
318             (Xword)(reloff / sizeof (Rel)), PLT_T_FULL, nlmp, (Addr)symval,
319             nsym->st_value, name, binfo));

321         /*
322          * Complete any processing for newly loaded objects.  Note we don't
323          * know exactly where any new objects are loaded (we know the object
324          * that supplied the symbol, but others may have been loaded lazily as
325          * we searched for the symbol), so sorting starts from the last
326          * link-map know on entry to this routine.
327          */
328         if (entry)
329                 load_completion(llmp);

331         /*
332          * Some operations like dldump() or dlopen()'ing a relocatable object
333          * result in objects being loaded on rtld's link-map, make sure these
334          * objects are initialized also.
335          */
336         if ((LIST(nlmp)->lm_flags & LML_FLG_RTLDLM) && LIST(nlmp)->lm_init)
337                 load_completion(nlmp);

339         /*
340          * Make sure the object to which we've bound has had it's .init fired.
341          * Cleanup before return to user code.
342          */
343         if (entry) {
344                 is_dep_init(nlmp, lmp);
345                 leave(lml, 0);
346         }

348         if (lmflags & LML_FLG_RTLDLM)
349                 dbg_desc->d_class = dbg_class;

351         return (symval);
352 }
```

_____***unchanged_portion_omitted_***