BSD Linker

Konstantin Belousov <kib@FreeBSD.org>

Draft rev. 2, November 14, 2011

Contents

1 Introduction

The document describes the goals of the project together with enumeration of the planned features, and gives some considerations for the architectural solutions.

The current version of the document is the work in progress. If you note that some very important ELF feature is missing from the text, but absolutely must be included as a requirement for the useful general-purpose ELF OS linker, please notify me instead of making bold claims. Thanks.

2 Overview of the existing linkers

Linking is (usually) the last step in producing the on-disk binary in the classic compiler processing flow. Linker combines separately compiled object files into single resulting object, resolving symbolic references and transforming the chunks of data for the binary into its on-disk form. In the modern systems, like ELF-based UNIXes, the binary should be further processed at the image activation time, e.g. to resolve external references to the dynamic linking libraries and to execute dynamic relocations. Due to this, `ld(1)` linker sometimes called static linker, to distinguish it from the run-time, or dynamic, linker.

Modern linkers are quite flexible (and buggy) tools. GNU ld has $\approx$ 100 KLoC of machine- and format-independed code, with bfd library measured at $\approx$ 600 KLoC to support formats and processors. The Gold, Google' reimplementation of the GNU ld in C++ by Ian Lance Taylor, weights 130 KLoC. Gold does not support ia64 and mips. A copyright owner for both GNU ld and Gold is Free Software Foundation, code is licensed under GPLv3 and relies on the GPLv3 BFD library.

Another popular ELF linker is provided by Sun as /usr/ccs/bin/ld in Solaris. Sources for Sun ld were never released, as far as I know.

3 Basic Functionality

The linker will target only ELF platforms. This means that support for PE / PE+ / COFF and Mach-O is explicitely excluded.

Successfully finished project shall produce a linker that alone can be used as THE linker to build fully functional FreeBSD systems and to execute make universe. Nonetheless, please note that modern linkers offer more features then utilized by the FreeBSD base system. The scope described in the document is explicitely larger then neccessary only for the FreeBSD src/ tree. Still, I expect that more work will be needed to cover all existing usage of the linker, e.g. by the Ports Collection and other third-party software.

3.1 Architectures

The linker must link ELF objects for all architectures currently supported by FreeBSD. First versions may be limited by x86 only, but eventually other architectures shall be available. The linker must be organized to allow new architecture addition without restructuring of the core code.

The list of the architectures is: i386, amd64, sparc64, ia64, mips, powerpc, arm. For several architectures in the list, several ABIs must be implemented:

- o32, n32 and n64 for mips, both big- and little-endian.

- For powerpc, we need 32- and 64-bit support. `-dotsyms`.
- **XXX** For arm, EABI.
- **XXX** Do we need big-endian ia64 ?

By default, the build of the linker must support all architectures and ABIs. The target object ABI shall be determined automatically from the source objects, if possible. Of course, the manual override of target ABI shall be provided.

The interesting proposition is to build the linker executable as the wrapper around the linker library. The library will export the API which is mostly equivalent to the linker command line. A potential user of the library, besides the linker itself, would be compiler frontends and IDEs.

3.1.1 Portability

The linker shall be reasonably portable. In particular, all needed ELF definitions shall be carried in the linker source, not depending on the platform varying ELF headers. The platform headers are often not enough for the linker implementation compilation anyway, and definitely omit the definitions for cross-architectures.

Use of the common features of modern Unixes should be not a problem.

The restrictions on the use of the host system resources can be useful, but probably a secondary target in the course of development.

3.2 Operation

Depending on the linker invocation, the link target may be of several types. The linker will need to operate in several modes, to be able to produce the following results:

- binary, default.
- PIE binary, `-pie`.
- dynamic shared object, `-shared`.
- relocatable ELF object file, for incremental linking, `-r`.

`-shared`, `-Bshareable`	Produce shared library.
`-pie`, `-pic-executable`	Produce PIE executable.
`-r`, `--relocatable`	Perform partial linking.
`-Ur`	Specify that partial linking shall be performed, but do resolve the references to constructor.
`-o out`, `--output=out`	Specify the name of the link result.
`--noinhibit-exec`	Create the linking results even if errors are encountered.

Table 1: Linking-mode

A feature of the GNU ld is its ability to produce the resulting binaries even if errors occur during the linking, controlled by the `--noinhibit-exec`. The binary is not marked as executable in case of errors, that does not help for the shared libraries, though. Traditional Unix linker exits on the first error encountered, that is sometimes unuseful.

Diagnostic from the linkers is often confusing for the users, due to the concepts used during the link stage are often unknown, and most programmers ignore the low-level plumbing in the build system, at least until it works.

See Table 1 for the description of the options affecting the mode of the linker operations.

3.3 Linker control

The new linker has no choice but use the same syntax for the command line as the GNU ld.

Linker must support the control of the layout of the final object by using the linker script language. There, the GNU syntax shall be supported too. Since only vague description of the GNU linker script syntax is available in the texinfo documentation for GNU ld, reference to the bison grammar for ld and gold will be required.

XXX should the project proposal include the definition of the linker script language ? Even if yes, the actual implemented language will be definitely different from what is written there, due to bugs in the specification.

Options allowing to control the linker scripts usage are described in the Table 2.

`-T script`, `--script=script`	Makes the `script` the linker script for the current linking.
`-dT script`, `--default-script=script`	Makes the `script` the linker script for the current linking. The processing of the script is postponed until the whole command line is read.

Table 2: Linker scripts

3.4 Sections

The linking operation combines the sections from the input files into the sections and possibly segments[1] of the output file. Sections with the same name are combined into a single section, then sections are merged into segments based on the attributes and the instructions in the linker script. GNU linker script allows to specify section names as globs, to concatenate differently named sections.

To support modern features of C++, for instance, C++ templates instantion and vtables generation, compiler sometimes have to emit the same section data from different compilation units. GNU linker allows to merge the duplicate content using the `.gnu.linkonce` naming convention for sections.

XXX elaborate

The link-time operations on the sections are described in the Table 3.

3.5 Symbols

See the Table 5 for the linker options related to the handling of symbols.

[1] For non-partial linking.

`-Tbss=orig`, `-Tdata=orig`, `-Ttext=orig`	Specify the expression `orig` as the starting virtual address for `.bss`, `.data` and `.text` sections respectively.
`-Ttext-segment=orig`	An alias for the `-Ttext` option.
`--section-start=sectionname=orig`	The start of the section `sectionname` is at `orig`.
`--sort-section=name`, `--sort-section=alignment`	The sections combined due to a wildcard section pattern in the linker script, will be sorted by name or alignment, respectively.
`--gc-sections`	Garbage collect unused sections. The used sections are the one containing the entry point, symbols which were marked undefined on the command line, and sections referenced by dynamic objects, and sections referenced from relocations of the sections above. When building the shared library, all visible exported symbols are referenced.
`--no-gc-sections`	Disables garbage collection of the unused sections.
`--warn-section-align`	Warn if alignment changes the section start address, for the sections which address is specified explicitely.
`--check-sections`	Check section addresses for overlap.
`--no-check-sections`	Do not check section addresses for overlap.
`--unique[=section]`	Create a separate output section for each input section which is matched by glob `section`. By default, linker concatenates the input sections of the same name. If `section` is ommitted, the same is done for any section not listed in the linker script(s).

Table 3: Sections

3.5.1 Symbol hash tables

ELF requires that static linker prepares the symbol hash table, that hashes the symbol names for faster lookup. Recently, GNU linkers developed the improved version of the hash table, using djb hash function. Also, they provide a Bloom filter over the set of hashed symbol names, to facilitate fast negative lookup.

FreeBSD dynamic linker uses only the old ELF hash table. Implementing GNU extension is not needed initially.

3.5.2 Weak symbols

Normal rules for ELF forbid the appearance of more then one definition for the symbol. Weak symbols are exempt from this restriction, and are only taken into account when reference is made to the symbol for which the normal (strong) definition is absent. Weak references do not cause fatal linking error, and are resolved to zero if no symbol definition is found.

3.5.3 Syntesized symbols

XXX

3.5.4 Symbol versioning

GNU style symbol-versioning extends the versioning introduced by Sun. Versioning allows to declare that given object supports specific ABI interfaces, and also allows to migrate the ABI of the single symbol among different ABI interfaces.

Symbol versioning is implemented as a set of tables in the ELF object, generated by the static linker based on the supplied version script and additional specifications from the object files.

The specification for the GNU symbol versioning tables is available at [6]. See Table 4 for the list of linker options affecting symbol versioning.

`--default-symver`	Create default symbol version, equal to the soname of the linking object, for unversioned exported symbols.
`--default-imported-symver`	Create default symbol version for unversioned imported symbols.
`--no-undefined-version`	Generate a fatal error if a symbol with undefined version is encountered.
`--version-script=filename`	Specify the file name for the version script.

Table 4: Symbol versioning

3.5.5 Commons

Common symbols are implicitely initialized to zero and are kept in the .bss. Different declarations of the common symbols are merged by the linker, with the final allocation taking space of the largest declared common. To support somewhat sloppy but popular practice of redeclaring common with the initialized symbol, linker supports overriding common symbol by defined with the same name.

Commons-related options are listed in the Table 6.

3.6 Libraries

The libraries support is the critical and the most visible part of the linker interaction with the user. The well-known linker facitilities to direct the library search and to specify the used libraries on the command line are the `-L` and `-l` switches. The `-L` adds the path to be used during the library search, the `-l`*name* specifies which library to use.

On the ELF platform, for the `-l`*name* specification, linker will search the shared library `lib`*name*`.so` first, and the static library `lib`*name*`.a` second. This behaviour may be adjusted with the use of `-Bstatic`, `-Bdynamic` switches, which require to use only static archives, and turns on the default behaviour, respectively.

See the Table 7 for the options controlling the generic linker behaviour related to the libraries.

3.6.1 Static libraries

The linker scans the static libraries specified on the command line sequentially, using archive members to resolve all found undefined symbols, before moving to the next archive. Only archive members that define some symbol, which was undefined in the previously processed object file, is fetched. This behaviour require right order of the libraries on the command line, which is sometimes confusing for novices.

The static library may contain an index, maintained by `ranlib(1)` command, which allowed the linker to satisfy the undefined symbols from the library using the library members. Without the index, traditional Unix linker was unable to satisfy undefined symbols from the library members by other members from the same library. There are several small variations of the archive format and the index, BSD[2], SysVR3 and SysVR4 / GNU. Support for the variants is easy, but only SysVR4 has practical meaning in the ELF world.

GNU `ld(1)` extends the idea of indexes by allowing to specify a group of static libraries on the command line. The libraries from the group are used to resolve each other undefined symbols until no progress can be made.

3.6.2 Shared libraries

See the Table 8 for description of the options for shared libraries build.

The use of the shared library for the linking stage is limited to the following actions:

- Enumeration of the undefined symbols that are satisfied by the library. Found symbols are not marked undefined anymore and are not searched in the following objects. The GOT and possibly PLT entries are generated to avoid relocations against text. See 3.7.3 for the further discussion.
- Special handling of the `R_ARCH_COPY` relocations. The data objects which are referenced from the main binary but defined in the shared objects, are handled specially. The definition of the symbol is added to the commons of the linked binary, and `R_ARCH_COPY` relocation is placed over the common object. The runtime linker copies the initial value of the object from the shared library into the executable object upon image activation. This is done to emulate the semantic of the static archives while using shared libraries.
- The found shared library is recorded in the linked object by storing the content of the `DT_SONAME` dynamic tag from the library in the `DT_NEEDED` dynamic tag of the linkage result.

By default, the mere fact that the library was specified on the command line causes the `DT_NEEDED` to be recorded. The `--as-needed` flag directs linker to only record the library if it indeed participated in the resolution of the undefined symbols. The option allows avoiding loading not needed libraries without requiring the developer to carefully inspect the use of each shared object.

3.6.3 Undefined symbols in the shared libraries

On the ELF platform, there is very rare to have undefined symbols in the shared libraries. Usually, if such symbol appears, this means that programmer forgot to explicitely list the library dependencies. Besides being a sloppy practice, the missed

[2] Also Seven edition.

recorded dependencies can cause very confusing errors in the programs that use the wrongly built library at the build or run time. There are several facilities to help the programmer to avoid the mistake.

The `--no-allow-shlib-undefined` causes the linker to emit the error if linked shared library contains undefined symbol. Note that it is not always an indication of error, but typically is.

Traditionally, ELF linkers implicitely added shared libraries, specified by `DT_NEEDED` of the explicitely listed shared libraries, to the link process. This way, if `liba.so` depends of `libb.so`, and `-la` was specified on the linker command line, then the whole namespace from `libb.so` is brought into the link. This implicit action may be turned off with the `--no-copy-dt-needed-entries`. Recent `gcc(1)` started specifying the `--no-copy-dt-needed-entries` option for the link stage. Also, there is a possibility that the option behaviour will become the default for recent versions of the GNU `ld(1)`. It is already the default for `gold(1)`.

3.6.4 `-rpath`

The `-rpath` `ld(1)` option is used to inform the dynamic linker about the search path for the shared libraries. The arguments of all `-rpath` options are combined and stored in the string table, pointed to by `DT_RPATH` or `DT_RUNPATH` dynamic tags. I am not sure about the differences between two tags semantic.

Historical behaviour of the SunOS static linker was to automatically construct rpath from the all `-L` options specified for the link. This behaviour appeared to be more troublesome then helpful, so now rpath is only recorded when explicitely provided.

3.7 Relocations

3.7.1 Relocation processing

The relocation is an instruction to a linker to modify some location in the object, according to the relocation-type specific rules. Relocation types are closely tied to the instruction set of the CPU, they are often made to modify selected bit fields in the encoding of the specific instruction class for the CPU. This can be mostly seen on the RISC architectures with the fixed width of the instructions, where immediate operands are placed in the designated subfield of the instruction word. X86 relocations affect the whole byte or word.

Relocation processing is the main purpose of the static linker. The dynamic loading facilities of the ELF makes it impossible to fully resolve object file relocations during the static linkage phase. Some relocations are postponed for the dynamic linking phase, some must be transformed into the form appropriate for the dynamic linking. See below the description of the GOT and PLT tables.

See the Table 9 for the options related to the relocation processing.

Relocations allow the late binding of the references to their definitions. Also, the functioning of the symbol interposing and thread local storage heavily depends on the relocations.

3.7.2 GOT

Global Offset Table (GOT) is the main mechanism that allows ELF platform to simulate the static libraries behaviour with the dynamic shared objects. By default, all interposable symbol references are tunneled through the GOT, which is the single place to fixup

external symbol in the object during the image activation. Static linker generates GOT and places the entries into the table when it sees some relocations in the linked object files. Kind of relocations is platform-specific. GOT is put into the writeable segment, to prevent the patching of the read-only text segment. Properly generated code should reference the GOT entries.

3.7.3 Position-independed code, PLT

ELF handling of loading of the shared libraries requires the libraries to use position-independed code. Since some architectures do not support position-independence of the machine code naturally, relocations must be applied that fixup the code for the load address. ELF still allows to have the non-PIC code in the text segment for shared libraries, which prevents sharing of the pages of the text, by using the `DT_TEXTREL` dynamic tag. It is a linker responsibility to properly mark objects needing application of relocations to the read-only segments with the tag.

Procedure Linkage Table (PLT) is the complementary structure to the GOT. It allows lazy resolution of the external code references, while still keeping the text purity. PLT works in conjunction with GOT.

Both GOT and PLT structures are architecture-depended. Linker must generate them from the thin air when doing final linking for either executable that reference shared libraries, or when linking shared library.

The Sun `ld(1)` has an option `-b` which prevents the linker from generating the GOT and PLT. Instead `ld(1)` keeps the relocations against the text segment that are resolved at the image activation time. Also, the `R_ARCH_COPY` relocations are not created, leaving data objects in the shared libraries. Gnu `ld(1)` has the option `-z nocopyreloc` that inhibits creation of the copy relocation.

3.7.4 Direct bindings

XXX

3.7.5 `PT_GNU_RELRO`

The GNU ELF extension `PT_GNU_RELRO` allows to have a set of the read-only pages in the image that still can be the target of the relocations. The dynamic linker must mark the pages that contain the region of [`p_vaddr`, `p_vaddr` + `p_memsz`] as readonly, after the dynamic relocations are applied.

3.7.6 Architecture-specific code generation

Several architectures use function descriptors to represent function pointers. Among them are IA64 and PowerPC64, at least. Linker must support them. For PowerPC 64, see ELF ABI supplement version 1.9 [8].

For PowerPC 64, recent version of binutils changed the naming of the symbol pointing to the actual function `fun` address from `.fun` to `.L.fun`.

3.8 Unwinding

Unwinding on the ELF platforms is based on the table-driven exception tables, as specified by the IA64 C++ ABI standard [7]. Linker participates in the preparation of the tables, merging the per-object file tables into the target exception table.

The unwinding table proper is supplied in the .eh_frame section. There, the Common Information Entry (CIE) followed by Frame Description Entries (FDE) are placed. Each CIE introduces a sequence of FDEs. E.g., one CIE may be provided for each object file. The FDE describes the way uwinding shall be performed for the range of instructions recorded in FDE.

Modern method of providing the unwind run-time with the references to the `.eh_frame` is to create addition section `.eh_frame_hdr`, specified by the `--eh-frame-hdr` option of GNU `ld(1)`. The section consists of some header and sorted array of two-word records, each specifying starting instruction and corresponding FDE location.

The unwind run-time finds the .eh_frame_hdr through the `PT_GNU_EH_FRAME` program header. Details of the tables are specified in the Linux Standard Base [9], which references the DWARF standard [5].

Unwind-related options are listed in the Table 10.

3.9 `-g`

The debugging information on the ELF platforms usually presented as the Dwarf version 2 / 3 / 4 records [5]. Probably, among the modern compilers, only not very newest versions of the Sun compiler emited stubs instead of Dwarf when compiler is given the `-g` switch. Handling of the Dwarf symbolic information from the linker perspective should not require any additional work except the normal section and relocation processing.

Gnu `ld(1)` ignores the `-g` switch altogether, for this reason. See Table 11 for the full list of the options related to handling of debugging information.

4 Miscellaneous features

Options not fit into other lists are summarized in the Table 12.

4.1 `PT_GNU_STACK`

The section named `.note.GNU-stack` do not contribute to the image bytes in memory. Instead, the presence of the section in the object file indicates that the code in the file do not require executable permission of the thread stack.

If even single object file used during the linking contained the `.note.GNU-stack` section, linker will generate the `PT_GNU_STACK` program header. The only significant information in the header if the access permission bits. They must always include `PF_R` and `PF_W`. If all object files used during linkage included `.note.GNU-stack` section, the `PT_GNU_STACK` header flags field has `PF_X` flag cleared, indicating that the resulting object do not require executable stack.

See Table 13 for the related options.

4.2 Map files and inspection of the linking

Linkers traditionally provide the map file which lists the details of the linking process in the human-readable form. At least, the map file contain the memory map and the assigned symbols values. GNU `ld(1)` also tracks the libraries references.

It is not clear whether it is feasible to produce map file which is char-to-char compatible with the Gnu `ld(1)`.

The options controlling map file generation and overall introspection of the linking process are listed in the Table 14.

4.3 Unwritten ELF quirks

The program headers must be located within the first page of the ELF object. The ELF branding note must be located within the first page of the object too.

4.4 Filters

Linker must support both normal and auxillary filter objects on the shared libraries. The support is almost trivial, since linker shall only create `DT_FILTER` and `DT_AUXILIARY` dynamic tags basing on the command line switches.

Filtering options are present in the Table 15.

4.5 C++ mangling and demangling

To provide better diagnostic messages, linker might employ symbol demangling algorithms. To ease the construction of the symbol versioning scripts, as well as for use with the command line switches and linker scripts, the mangling of the C++ identifiers may be used.

Mangling is generally compiler-specific. Linker must support at least the GNU C++ mangling and demangling algorithms.

See Table 16 for the related options.

5 Future

The facilities described in the section are not planned for the first series of the linker releases. They are listed as highly desired features, which shall be considered when designing the linker architecture, to not make it impossible to graft them later.

5.1 Plugins

The main method for extending linker[3] is the use of the linker plugins. There is no well-established plugin API for the linkers, in part because there is no much variety in the available linkers, in part because any plugin API will be closely tied to the linker internals. Defining the API will be the part of the further project, after the basic linker functionality is done.

Next subsections describe some prospective applications of the plugins in the context of the project, mostly in the hand-waving form.

5.2 Audit

There are two audits point when linking process is involved. One is the static linking process, and another one is the later dynamic linking. Static linking shall allow the audit plugins. Also, it arranges for the run-time auditing.

Static linking audit is performed by loading the auditing plugin. XXX elaborate.

Audit-related options are listed in the Table 17.

[3] As opposed to *embedding* it.

5.3 LTO

LTO stands for the Link Time Optimization. The technique performs the final optimization pass at the linking stage, since there the full text of the resulting binary is available. The LTO can be reasonably executed only over the intermediate code representation, thus compiler output for LTO usually contains both native code sections and intermediate language (IR) sections. The non-LTO link uses native code, while LTO picks the IR.

The IR is ultimately compiler-specific. Confining the general-purpose linker to the single compiler implementation, possibly even to the (lagging) versions of the compiler seems unuseful. Instead, the LTO shall be delegated to the plugins, supplied by compiler authors or third-party. Linker duty is to provide a plugin API that makes the LTO possible.

A About me

I am kib@freebsd.org. During the last years I implemented the following features in the FreeBSD `ld-elf.so(8)`:

- Support for dynamic token strings in rpath and soname, like `$ORIGIN`.
- Support for the filters on dynamic shared objects.
- Optimization to provide a system information to the startup through ELF aux vectors, which eliminated around ten sysctl(2) calls from each exec.
- GNU-style non-executable stacks using `PT_GNU_STACK` program header, generated by the GNU- and compatible toolchains.
- Cleanup of the `atexit(3)` handlers installed by shared object, on the object unload.

I ported the libunwind stack unwinding library [10] to FreeBSD, supporting i386 and amd64 architectures.

References

[1] Steve Chamberlain, Ian Lance Taylor. The GNU linker. From GNU binutils 2.21.1.

[2] John R. Levine. Linkers and loaders.

[3] Oracle. Linker and Libraries Guide.

[4] Tool Interface Standard (TIS). Executable and Linking Format (ELF) Specification Version 1.2.

[5] DWARF Debugging Information Format Workgroup. DWARF Debugging Information Format Version 3.

[6] Ulrich Drepper. ELF Symbol Versioning.

[7] Itanium C++ ABI `http://sourcery.mentor.com/public/cxx-abi/abi.html`.

[8] Ian Lance Taylor, Zembu Labs. 64-bit PowerPC ELF. Application Binary Interface Supplement 1.9.

[9] Linux Standard Base.

[10] libuwind `http://www.nongnu.org/libunwind/`.

`-E`, `--export-dynamic`	Specifies that all global symbols from the linking object must appear in the dynamic symbol table.
`--no-export-dynamic`	Reverts the `--export-dynamic`, restoring the default behaviour of only adding symbol to the dynamic symbol table if referenced by some shared library specified for linking.
`--defsym=symbol=value`	Enters the symbol `symbol` into the linker symbol table with the value `value`.
`-u symbol`, `--undefined=symbol`	Enters the symbol `symbol` into the linker symbol table as undefined. As a result, the symbol will be searched for in libraries.
`--retain-symbols-file=file`	Only retain dynamic symbols listed in the file `file`.
`-x`, `--discard-all`	Delete all local symbols, i.e. symbols not referenced externally.
`-X`, `--discard-locals`	Delete all local symbols name of which starts with `.L`.
`-R filename`, `--just-symbols=filename`	The content of the file `filename` is not included in the linking output. Only symbol names and their values are used.
`--dynamic-list=file`	The symbols which name are listed in the `file`, are added to the GOT.
`--dynamic-list-data`	Add all global data symbols to the GOT.
`--dynamic-list-cpp-new`	Add the symbols implementing C++ new and delete operators, to the GOT.
`--dynamic-list-cpp-typeinfo`	Add the symbols implementing C++ typeinfo data, to the GOT.
`--unresolved-symbols=method`	Handle the unresolved symbols according to the `method`. The `method` can be one of • `ignore-all` • `report-all` • `ignore-in-object-files` • `ignore-in-shared-libs`
`-z defs`, `--no-undefined`	Report undefined symbols from the normal object files, even if linking non-symbolic shared library.
`-z muldefs`, `--allow-multiple-definition`	Allow multiple definitions of the same symbol. The first definition encountered is used.
`--warn-once`	Warn about each undefined symbol only once, instead of making a message on processing each reference.
`--warn-unresolved-symbols`	Generate a warning, and not the error, when linker encounters undefined symbol.
`--error-unresolved-symbols`	Turns off `--warn-unresolved-symbols`.

Table 5: Symbols

`-d`, `-dc`, `-dp`	Assign space for the common symbols even for relocatable linking.
`--no-define-common`	Do not allocate commons for non-relocatable linking.
`--sort-common{=method}`	Sort common symbols by alignment. The `method` is `ascending` or `descending`, default descending.
`--warn-common`	Warn on the common symbol mixing with the definition.
`--no-define-common`	Inhibit the allocation of space for the referenced common symbols. This allows to specify that shared library references the common symbols from the main binary or other shared library, instead of making a copy in the library itself.

Table 6: Commons

`-L path`	Specifies the libraries search path.
`-lname`	Requests the use of the static or dynamic library named libname.
`-Bstatic`, `-static`, `-dn`	Prefer static libraries.
`-Bdynamic`, `-call_shared`, `-dy`	Prefer dynamic libraries.
`--whole-archive`	Requests inclusion of the whole content of the static archives into the linking results. The object files from an archive are linked even if not resolving any undefined symbol from the previously processed objects.
`--no-whole-archive`	Turns off `--whole-archive`.
`-(`, `--start-group`	Start the group of the static archives that are searched as a group for any undefined symbols, until all possible symbols are resolved from the group.
`-)`, `--end-group`	Ends the group started by `--start-group`.
`-nostdlib`	Library search paths from the linker scripts are ignored. Only the paths specified with the `-L` option are searched.
`-Y path`	Add `path` to the default library search path.

Table 7: Libraries

`--copy-dt-needed-entries`	Copy the `DT_NEEDED` entries from the shared libraries specified on the command line, into the linking result.
`--no-copy-dt-needed-entries`	Do not copy the `DT_NEEDED` entries from the shared libraries specified on the command line, into the linking result.
`-h name, -soname=name`	Specifies the name recorded into the `DT_SONAME` dynamic tag for the shared library.
`--exclude-libs lib1,lib2,...`	List of static libraries symbols from which are made hidden.
`--as-needed`	Record `DT_NEEDED` only for the libraries which are referenced by a symbol relocations.
`--no-as-needed`	Turns off `--as-needed`.
`--allow-shlib-undefined`	Allow the linking to succeed even if the resulting shared library contains unresolved symbols (default).
`--no-allow-shlib-undefined`	Fail the link if the resulting shared library contains undefined symbols.
`--rpath=dir`	Add the directory `dir` to the path used by the dynamic linker to search dependencies.
`--rpath-link=dir`	Add the directory `dir` to the path used by the static linker to search dependencies of the shared libraries specified on the command line.
`-Bsymbolic`	The symbolic references from the resulting object are bound to the object itself. The object is the first interposer for the resolution of the symbols from itself.
`-Bsymbolic-functions`	The function references from the resulting object are bound to the object itself.
`-Bgroup`	Mark the resulting object and its dependencies for the dynamic linker as allowing resolution of the undefined symbols only among itself. Implies `--unresolved-symbols=report-all`.
`-z initfirst`	Mark the resulting object as requiring the initialization first, before other object loaded due to the same dependency chain. The object is finalized last.
`-z interpose`	The symbols of the resulting object interpose all other objects symbols, except the main executable.
`-z nodelete`	Mark the resulting object as not unloadable after the last `dlclose(3)`.
`-z nodlopen`	Mark the resulting object as not loadable with `dlopen(3)`.
`-z nodump`	Mark the resulting object as not dumpable by `dldump(3)`.
`-z now`	Mark the resulting object as requiring the immediate relocation processing, even for the lazy PLT relocations.
`-z lazy`	Mark the resulting object as accepting the lazy relocation processing for PLT entries.
`-z origin`	Marks the resulting object as requiring dynamic tokens expansion for rpath and sonames.
`-z nodefaultlib`	Mark the resulting object as ignoring the default paths for searching the dependencies.

Table 8: Shared libraries

`-q`, `--emit-relocs`	Keep relocations in the final executable.
`-z combreloc`	Combine multiple relocation sections into one.
`-z nocombreloc`	Do not combine relocation sections.
`-z nocopyreloc`	Do not create copy relocations.
`-z norelro`	Do not generate `PT_GNU_RELRO` segment.
`-z relro`	Enable the generation of the `PT_GNU_RELRO` segment.
`--warn-shared-textrel`	Warn if relocations are recorded against the text segment.

Table 9: Relocations

`--eh-frame-hdr`	Generate `.eh_frame_hdr`.

Table 10: Unwinding

`-g`	Ignored.
`-s`	Remove all unneeded symbol and debugging information from the resulting object.
`-S`	Remove debugging information from the resulting object.

Table 11: Debugging information

`-@ file`	Batch files. Options are read from the file.
`-e entry`, `--entry entry`	Specify entry point.
`-init=symbol`	Call the function `symbol` at the object initialization time. By default, the `_init` function is called, if defined.
`-fini=symbol`	Call the function `symbol` at the object unload time. By default, the `_fini` function is called, if defined.
`-I filename`, `--dynamic-linker=filename`	Use the named dynamic interpreter for the image activation.
`--wrap=symbol`	The undefined references to the `symbol` are resolved to the `-__wrap_symbol`. The references to the `__real_symbol` are resolved to `symbol`.
`--build-id=uuid|sha1|md5|hexstring|none`	Create the `.note.gnu.build-id` section in the output file.
`--build-id` `-z max-page-size=val`	The same as `--build-id`=sha1. Set the maximum page size to the value `val`.
`-z common-page-size=value`	Set the common page size to the value `val`.

Table 12: Miscellaneous options

`-z execstack`	Mark the object as requiring executable stack.
`-z noexecstack`	Mark the object as not requiring executable stack.

Table 13: Executable stack

`-M`, `--print-map`	Print the linking map file to the standard output.
`-Map=mapfile`	Write the linking map to mapfile.
`-t`, `--trace`	Print the names of the input files on the first open.
`--verbose=N`	Set verbosity level.
`--print-gc-sections`	Print the names of the section which are garbage-collected.
`--no-print-gc-sections`	Turns off `--print-gc-sections`.
`-y symbol`, `--trace-symbol=symbol`	Print the name of each input file or archive member which references symbol `symbol`.
`-v`, `--version`, `-V`	Print the version of the linker.
`--cref`	Output the cross-reference information for the symbols.
`--fatal-warnings`	Any warning causes the fatal error.
`--no-fatal-warnings`	Turns off `--fatal-warnings`.
`--no-warn-mismatch`	Do not warn if an object file not compatible with the currently linking object was encountered. The file is ignored.
`--no-warn-search-mismatch`	Do not warn if an library incompatible with the currently linking object is found on the search path.

Table 14: Linking process inspection

`-F name`, `--filter=name`	Use the object with the name `name` as a filter.
`-f name`, `--auxiliary=name`	Use the object with the name `name` as an auxiliary filter.
`-z loadfltr`	Mark the object for the dynamic linker, noting that the filters for the object shall be loaded immediately.

Table 15: Filtering

`--demangle[=style]`	The symbol names used in the output are demangled according to the rules of the mangler for C++ compiler, specified by `style`. Also affects `--dynamic-list-cpp-new` and `--dynamic-list-cpp-typeinfo` options.
`--no-demangle`	Do not demangle the symbol names on output.

Table 16: Mangling

`--audit lib`	Record the library `lib` as `DT_AUDIT`.
`--depaudit lib`, `-P lib`	Record the library `lib` as `DT_DEPAUDIT`.

Table 17: Audit