SEE TEXT ONLY

THE LLD LINKER IN FREEBSD

A LINKER IS A PROGRAM THAT TAKES ONE OR MORE OBJECT FILES GENERATED BY A COMPILER OR ASSEMBLER AND COMBINES THEM INTO ONE EXECUTABLE PROGRAM OR LIBRARY. IT IS PART OF AN OPERATING SYSTEM'S TOOLCHAIN, THE SET OF PROGRAMS USED TO BUILD, DEBUG, AND TEST SOFTWARE.

by Ed Maste

From the beginning, FreeBSD has used the GNU binutils linker. It is also known as the "BFD" linker, after the Binary File Descriptor library upon which it is built. FreeBSD developers kept the copy of the linker in the FreeBSD tree up-to-date, importing new versions as they were released by the GNU project.

Then, in the mid-2000s, the Free Software Foundation (FSF) changed the license to version 3 of the GNU General Public License (GPLv3). It added new restrictions that some FreeBSD developers and users found unpalatable. Since that time, developers in the FreeBSD Project have continued to apply minor updates and bug fixes, but the linker has not been updated beyond version 2.17.50.

The need for a new linker for FreeBSD has been apparent for some time, and a viable candidate emerged recently: LLD, the linker in the LLVM family of projects. LLD is intended to be a high-speed linker with support for multiple object file formats. It supports the ELF format

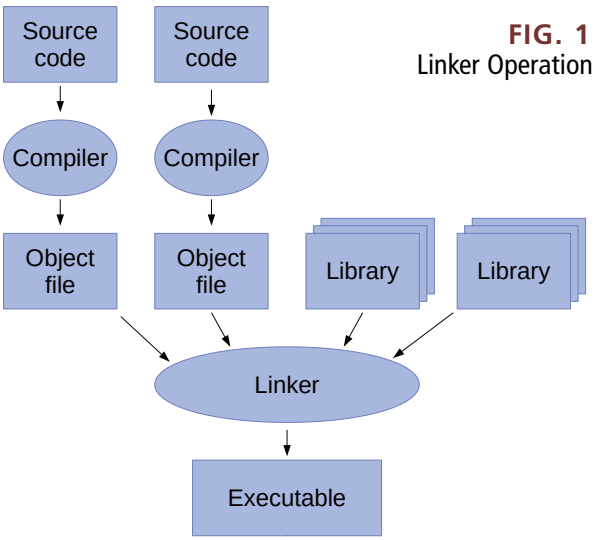


FIG. 1
Linker Operation

used by Unix-like operating systems, Windows' COFF format, and Darwin/OS X's Mach-O.

Where possible, LLD maintains command-line and functional compatibility with existing linkers such as GNU ld, but LLD's authors are not constrained by strict compatibility where it hampers performance or desired functionality. As with other LLVM projects, LLD is released under a permissive Free Software license.

LLD can deliver for FreeBSD a modern, maintainable, and high-performance linker for the base system. It will allow us to support the new CPU architectures in FreeBSD using an in-tree toolchain, and will enable new performance optimizations like Link-Time Optimization (LTO).

HISTORY

LLD was added to the upstream LLVM source repository at the end of 2011, with a design based largely on linking requirements of the Mach-O format used by Apple and the atom model it implies. Atoms are the smallest indivisible chunks of code or data, and are a rather generic representation of an object file. Along with Mach-O support, LLD initially included an atom-based ELF and COFF linking implementation.

Linking the Mach-O format needs the flexibility afforded by the atom model, but it's an unnecessary complication for the ELF and COFF formats. For these formats, the section is the smallest usable unit.

In May 2015, Rui Ueyama of Google started working on a new section-based COFF linker, and two months later in July, Michael J. Spencer committed a new ELF linker implementation based on the section-based COFF support. The new COFF implementation was enabled by default in August, and likewise for ELF in November.

By the end of 2015, LLD could form part of a self-hosting FreeBSD amd64 toolchain: that is, it was possible to build Clang/LLVM and LLD using Clang/LLVM and LLD. Many developers improved LLD throughout 2016, with notable FreeBSD-related contributions from Rui Ueyama, Rafael Espindola, George Rimar, and Davide Italiano.

Over the past year, I have been experimenting with building the FreeBSD base system with LLD. Although LLD could self-host at the end of 2015, it lacked support for many features required by FreeBSD. I created a bug report in LLVM's bug tracker as a "meta bug" to keep track of all of the issues preventing the use of LLD as a FreeBSD system linker. Over time it grew to 63 individual issues, of which seven remain open.

Issues included lack of relocatable output, fine-grained control over library search paths, arithmetic expressions in linker scripts, comprehensive versioned symbol support, and miscellaneous command-line options. I had to disable the build of FreeBSD's boot loaders, 32-bit compatibility libraries, tests, rescue binaries, and the GDB debugger in order to link a subset of userland binaries, although many did not run. Linking the kernel was not possible, as it relies extensively on the use of a linker script.

By March 2016, with a few temporary workarounds in FreeBSD and LLD, I was able to build a usable subset of the FreeBSD userland. Symbol versioning and linker script expressions were still unsupported, and the kernel could still not be built as a result. However, the rapid progress of LLD's development convinced me it was on track to become a viable system linker.

Symbol versioning and linker script expression evaluation arrived a few months later. By August, relocatable output and a somewhat esoteric command line option used in building boot loader components were the significant features outstanding. Now LLD is usable for nearly all aspects of the amd64 FreeBSD base-system userland and kernel build; the boot loaders are the only FreeBSD components that still do not build with LLD.

DESIGN

LLD's design goals include speed, simplicity, and extensibility. LLD attempts to be fast by doing less work, and when it is necessary to do something, doing it only once. Traditional UNIX linkers visit objects sequentially, including objects specified directly on the link command line and those included in an archive. The linker builds a list of undefined symbols, and links those objects that satisfy the requirements for those symbols. The new objects may in turn introduce new undefined symbols, and objects may need to be visited multiple times to resolve all undefined symbols.

LLD, instead, performs only one pass over the specified object files and archives, keeping track of both defined and undefined symbols found in each object it visits. Symbols are resolved without

having to revisit objects or libraries again. This results in slightly different linking semantics when compared to traditional linkers, but well-written software should see no difference. It is possible to craft a scenario that works with a conventional linker and fails with LLD, but that is expected to be unlikely in practice. No broken cases have yet been found while testing a variety of third-party software with LLD as the linker.

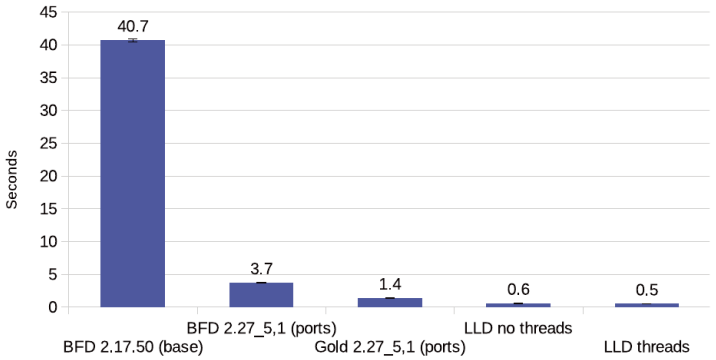
The ELF and COFF linkers in LLD share the same general design, but do not share code. They provide the same command line user interface as the native linkers for each file format: GNU ld for ELF, and Microsoft's linker for COFF. This avoids the complexity and runtime cost of an abstraction layer. As a result of this approach, the ELF linker is only about 13,000 lines of code. This number is much smaller than other linkers (GNU ld and GNU gold), even though it isn't directly comparable, as all three rely in different amounts on support libraries.

As with other components in the LLVM family, the linkers in LLD are implemented as libraries with a lightweight command line driver. This allows it to be easily embedded in other projects.

PERFORMANCE

One of LLD's primary design goals is to be a high-performance linker. Small programs should link quickly, and linking should not become exponentially larger as program size increases.

FIG. 2 Real Link Time, Clang Release Build

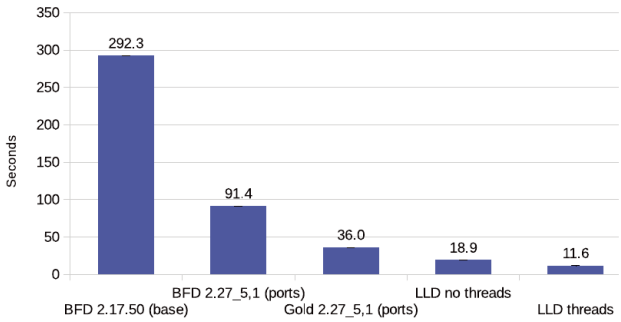


The graph in Figure 2 compares the real (wall-clock) time taken to link a release build of the Clang compiler using several different linkers. The linker inputs consist of 158 object (.o), static library (.a), and shared library (.so) files, totaling 91 megabytes in size.

The experiment was performed on my development desktop, which has a quad-core (8 thread) Intel i7-3770 CPU and 32GB of memory. I compared BFD ld 2.17.50 (the current FreeBSD base system linker), BFD and Gold 2.27_5,1 from the FreeBSD ports tree, and LLD built from the LLVM source repository. By default, the Gold linker in the FreeBSD ports collection is built without threading support, and Gold's threaded mode was not investigated further.

Figure 3 compares the same linkers for a debug build of Clang. The linker processes the same number of files in this case, but the addition of

FIG. 3 Real Link Time, Clang Debug Build



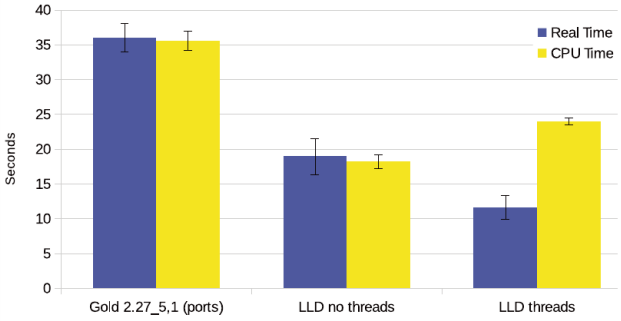
debug data makes them substantially larger and requires the linker to do much more work.

LLD achieves this high performance in several ways that can be briefly enumerated as: avoid doing work where possible, perform expensive but necessary operations only once, and use threads to perform operations in parallel.

Several tasks in LLD can be performed in parallel, including uncompressing input sections, splitting mergeable sections, merging common strings, and identical code folding. Figure 4 demonstrates the effect of LLD's use of threading. The link completed 63% faster, but consumed about one-third more CPU time.

This trade-off is worthwhile in a typical devel-

FIG. 4 Real & CPU Link Time, Clang Debug Build



oper's edit-compile-test cycle. The final linking step requires all of the individual compiler invocations to be complete, and there is likely no other work the computer could do. When linking software on a shared resource (for example, building the FreeBSD package sets), disabling threads may be a better choice.

ARCHITECTURAL SUPPORT

LLD includes at least some support for almost all the CPU architectures relevant to FreeBSD. As described earlier, amd64 (or x86-64) is well supported, and LLD built from the development repository is capable of linking a working FreeBSD/amd64 kernel and userland, except for the boot loaders.

32-bit x86 (i386) and AArch64 (arm64) support is also quite mature, but not yet well tested in FreeBSD. LLD can self-host on arm64 and link a working FreeBSD/arm64 kernel, with small workarounds for outstanding issues.

32-bit ARM, 32-bit and 64-bit MIPS, and 32-bit and 64-bit PowerPC are supported by LLD, and are capable of at least linking trivial applications, but are not yet viable as a FreeBSD system linker.

RISC-V support is being planned, but has not yet started. It seems that sparc64 is the only FreeBSD CPU architecture that is unlikely to be supported by LLD.

LINKER OPTIMIZATIONS

One large benefit LLD will bring to FreeBSD is base system support for whole-program Link-Time Optimization (LTO). LTO refers to optimization that is performed across modules at link time.

A common optimization is to eliminate unused code paths. With conventional linking, this can be applied only within a single source file. With LTO, the compiler and linker work together and can eliminate code paths that can only be determined as unused when examining the whole program.

For LTO the compiler emits an LLVM bitcode file instead of native machine instructions in an ELF object file. The linker processes these bitcode files similarly to regular object files and allows both types to be used together.

Another optimization can be performed by the linker. Large C++ applications often contain functions that compile to machine instructions identical to another function. Identical Code Folding (ICF) is an optimization that identifies read-only sections that happen to have the same content. For a sample of large binaries, ICF reduces the size by 5% to 8%.

NEXT STEPS

Clang/LLVM 3.9 was recently imported into FreeBSD's development branch, and that work included LLD 3.9. LLD is now installed as /usr/bin/ld.lld on amd64 and arm64 for experimentation and testing. On arm64, it is also now installed as ld, as the BFD linker version 2.17.50 does not include arm64 support.

Linking the boot loaders with LLD must be addressed next. LLD developers are actively working on issues in LLD that prevent this, and some changes may also be required in FreeBSD. Once this is complete, a newer snapshot of LLD will be imported into FreeBSD and made available with a build-time configuration setting (for example, WITH_LLD_AS_LD=yes). The ports tree will then be extensively tested using the in-tree LLD as the system linker.

The investigation and iterative bug fixing approach will need to be undertaken on all of FreeBSD's supported CPU architecture. Experimentation with linker optimizations will then proceed for both the FreeBSD base system and ports collection. ●

ED MASTE manages project development for the FreeBSD Foundation and works in an engineering support role with the University of Cambridge Computer Laboratory. He is also a member of the elected FreeBSD Core Team. Aside from FreeBSD and LLVM projects, he is a contributor to several other open-source projects. He lives in Kitchener, Canada, with his wife, Anna, and sons, Pieter and Daniel.