mold: Modern Linker

Rui Ueyama
Blue Whale Systems PTE LTD

Overview of this talk

1. Status of the project
2. What is the linker?
3. Why does the linker's speed matter?
4. Performance comparison of mold, LLVM lld and GNU gold
5. Why is mold so fast?
6. Hints on writing faster programs

Who am I?

Rui Ueyama

Full-time open-source developer

The original creator of the LLVM lld and mold linkers

Consider supporting my project via GitHub Sponsors or making a commercial support contract!

https://github.com/rui314

Status of the project

Status of the project

mold for Unix (Linux) is production-ready and used by various projects and in many organizations. It's *extremely* fast (I'll talk about it later.)

mold for macOS/iOS/watchOS/etc is in pre-alpha. It looks like it's also *extremely* fast but not for general use yet. So stay tuned!

mold for Windows is planned after mold/macOS.

What is the linker?

What is the linker?

Modern linkers seem to be pretty complicated because they have lots of miscellaneous features, but the fundamental concept is very simple:

1. A compiler compiles a piece of source code into an object file (a file containing machine code), and
2. a linker combines object files into a single executable or a dynamic library file.

Unless you are writing a program in a scripting language, chances are you are using a linker!

The origin of the linker

Linkers and Loaders, ISBN 978-1558604964, John R. Levine (1999) Chapter 1.2

> *Programmers were using libraries of subprograms even before they used assemblers. By 1947, John Mauchly, who led the ENIAC project, wrote about loading programs along with subprograms selected from a catalog of programs stored on tapes, and of the need to relocate the subprograms' code to reflect the addresses at which they were loaded. Perhaps surprisingly, these two basic linker functions, relocation and library search, appear to predate even assemblers, as Mauchly expected both the program and subprograms to be written in machine language. The relocating loader allowed the authors and users of the subprograms to write each subprogram as though it would start at location zero, and to defer the actual address binding until the subprograms were linked with a particular main program.*

A linker would be a program you want to write after inventing a digital computer, even before writing an assembler for it!

For example, if you write a sort() function in machine code in hex, you'd want to use it in many programs. That naturally led to a desire to write a linker.

Linker in action (1)

```
$ make
c++ -c -o main.o main.cc
c++ -c -o object_file.o object_file.cc
c++ -c -o input_sections.o input_sections.cc
……
c++ -c -o glob.o glob.cc
c++ main.o object_file.o input_sections.o output_chunks.o mapfile.o perf.o linker_script.o archive_file.o
output_file.o subprocess.o gc_sections.o icf.o symbols.o commandline.o filepath.o glob.o -o mold
-L/home/ruiu/mold/oneTBB/build/linux_intel64_gcc_cc9.3.0_libc2.31_kernel5.10.0_release/
-Wl,-rpath=/home/ruiu/mold/oneTBB/build/linux_intel64_gcc_cc9.3.0_libc2.31_kernel5.10.0_release/
-L/home/ruiu/mold/mimalloc/out/release -Wl,-rpath=/home/ruiu/mold/mimalloc/out/release -L/home/ruiu/mold/xxHash
-lcrypto -pthread -ltbb -lmimalloc -lz xxHash/libxxhash.a
```

- The bold part is the command line to invoke a linker. That usually occurs at the very end of a build.
- gcc/clang/cc (or g++/clang++/c++) are a *compiler driver* and not a compiler itself (in a narrower sense.) They invoke appropriate backend commands based on given files' file extensions.
- The linker command, "ld", is also expected to be indirectly invoked by a compiler driver.

Linker in action (2)

Appending "-###" makes a compiler driver to show the actual ld command line:

```
$ clang -o hello hello.o main.o -###
/usr/bin/ld -z relro --hash-style=gnu --build-id --eh-frame-hdr -m elf_x86_64
-dynamic-linker /lib64/ld-linux-x86-64.so.2 -o hello
/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../x86_64-linux-gnu/crt1.o
/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../x86_64-linux-gnu/crti.o
/usr/bin/../lib/gcc/x86_64-linux-gnu/10/crtbegin.o -L/usr/bin/../lib/gcc/x86_64-linux-gnu/10
-L/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../x86_64-linux-gnu
-L/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../../lib64 -L/lib/x86_64-linux-gnu
-L/lib/../lib64 -L/usr/lib/x86_64-linux-gnu -L/usr/lib/../lib64
-L/usr/lib/x86_64-linux-gnu/../../lib64 -L/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../..
-L/usr/lib/llvm-10/bin/../lib -L/lib -L/usr/lib hello.o main.o -lgcc --as-needed -lgcc_s
--no-as-needed -lc -lgcc --as-needed -lgcc_s --no-as-needed
/usr/bin/../lib/gcc/x86_64-linux-gnu/10/crtend.o
/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../x86_64-linux-gnu/crtn.o
```

Why do we need a linker?

We don't actually need a linker! But such system is impractical.

- It is theoretically doable to make a compiler to directly emit an executable
- To make it possible, we have to pass *all* source code to a compiler
- It is inefficient to compile all source code every time we rebuild a file
- Ex. functions in libc: https://git.musl-libc.org/cgit/musl/tree/src/stdio/vfprintf.c
  - It's just a waste of time to compile functions like this every time we build something

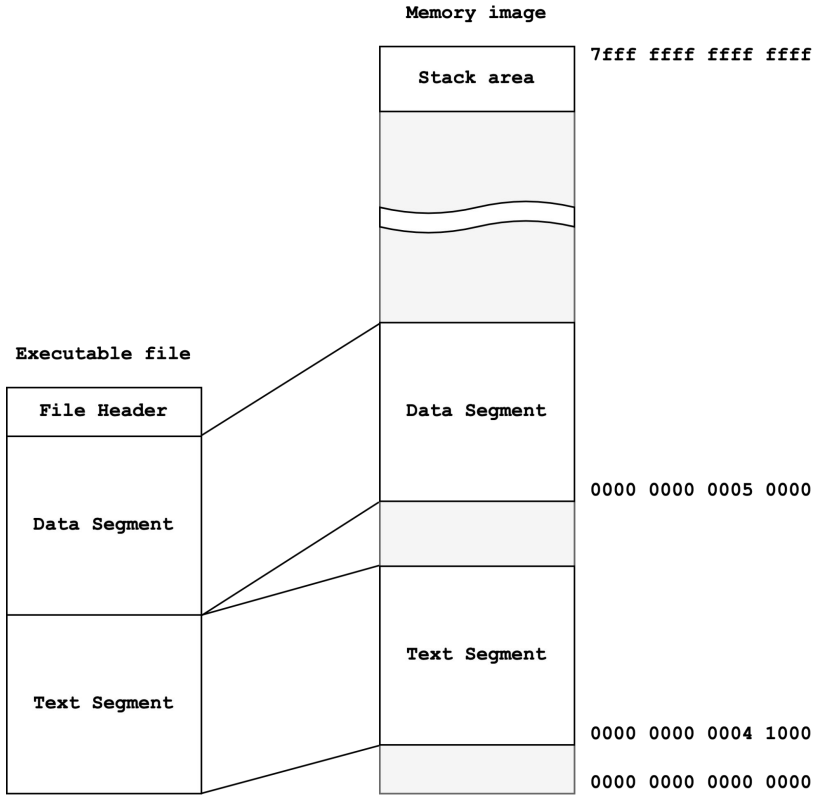
Separate compilation

- A compiler takes a fragment of a program and emit compiled machine code for it
- A linker combines machine code fragments to complete a program

Executable file and its memory image

Executable contains code and data

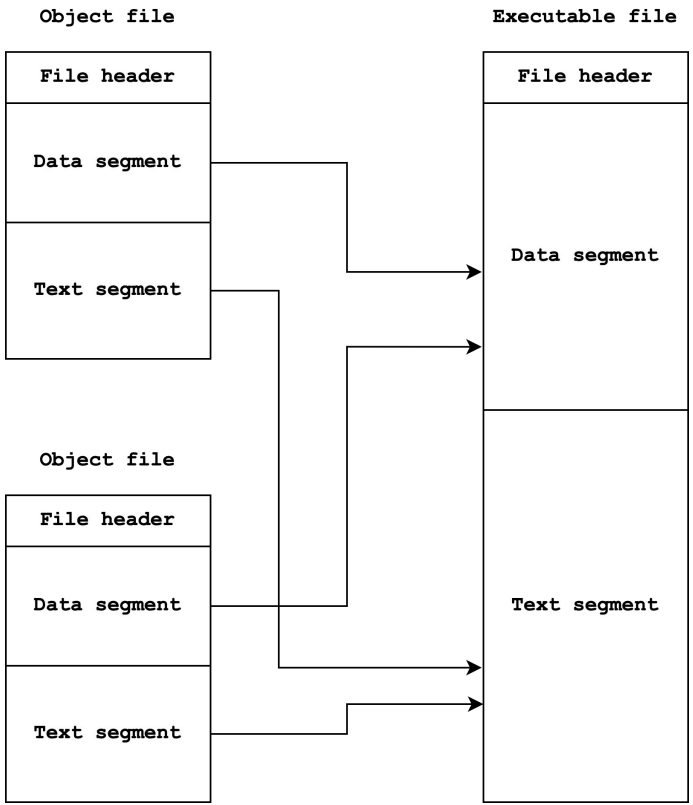
Code and data are mapped to memory



Object file structure

Each object file contains code and data

Linker combines them into larger segments



Relocation (1)

```
$ cat foo.c
int printf();
int main() {
  printf("Hello world\n");
}

$ cc -fno-PIC -S foo.c -c
-o-
```

```
.text
.section        .rodata
.LC0:
.string "Hello world\n"
.text
.globl  main
.type   main, @function
main:
pushq   %rbp
movq    %rsp, %rbp
movl    $.LC0, %edi
movl    $0, %eax
call    printf
movl    $0, %eax
popq    %rbp
ret
```

Relocation (2)

```
$ objdump -dr /tmp/foo.o
Disassembly of section .text:
0000000000000000 <main>:
   0:   f3 0f 1e fa             endbr64
   4:   55                      push   %rbp
   5:   48 89 e5                mov    %rsp,%rbp
   8:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # f <main+0xf>
                        b: R_X86_64_PC32        .rodata-0x4
   f:   b8 00 00 00 00          mov    $0x0,%eax
  14:   e8 00 00 00 00          callq  19 <main+0x19>
                        15: R_X86_64_PLT32      printf-0x4
  19:   b8 00 00 00 00          mov    $0x0,%eax
  1e:   5d                      pop    %rbp
  1f:   c3                      retq
```

Relocation (3)

```
14:   e8 00 00 00 00          callq  19 <main+0x19>
                      15: R_X86_64_PLT32      printf-0x4
```

- 0xe8 is the opcode of callq which is followed by four bytes offset.
- The offset to the printf's machine code is not known at compile time, as the compiler has no idea as to where the printf is at runtime.
- Compiler instead emits a *relocation record* saying that "fill the offset X of section Y with a symbol address Z".
- Linker applies relocations to complete a program.

Fundamental features of the linker

1. Combining object files into a single file, and
2. applying relocations

Everything else is secondary!

Static library (1)

In libc, each function is stored to its own source file and compiled to an object file individually.

https://github.com/ifduyue/musl/tree/master/src/stdio

You can see the contents of a static libc library with the following command:

```
$ ar t /usr/lib/x86_64-linux-gnu/libc.a
init-first.o
libc-start.o
sysdep.o
version.o
check_fds.o
libc-tls.o
elf-init.o
dso_handle.o
...
```

Static library (2)

Reasons to split libc into multiple object files

- We don't want to copy unnecessary functions to an output file
- For example, if we use only printf, we want to copy only printf.o and other object files that printf.o depends to an output file.

You generally don't know which object files in libc provides functions that you use, but you don't have to!

If an archive file is given, the linker pulls out object files that are necessary to complete a link. Unnecessary files are not linked.

Linking static libraries

A linker does the following to link static libraries:

1. Reads the contents of static libraries to find defined symbols, and
2. pulls out object files defining necessary symbols from static libraries and append it to an output file.

Static library (2)

Advantages of static libraries

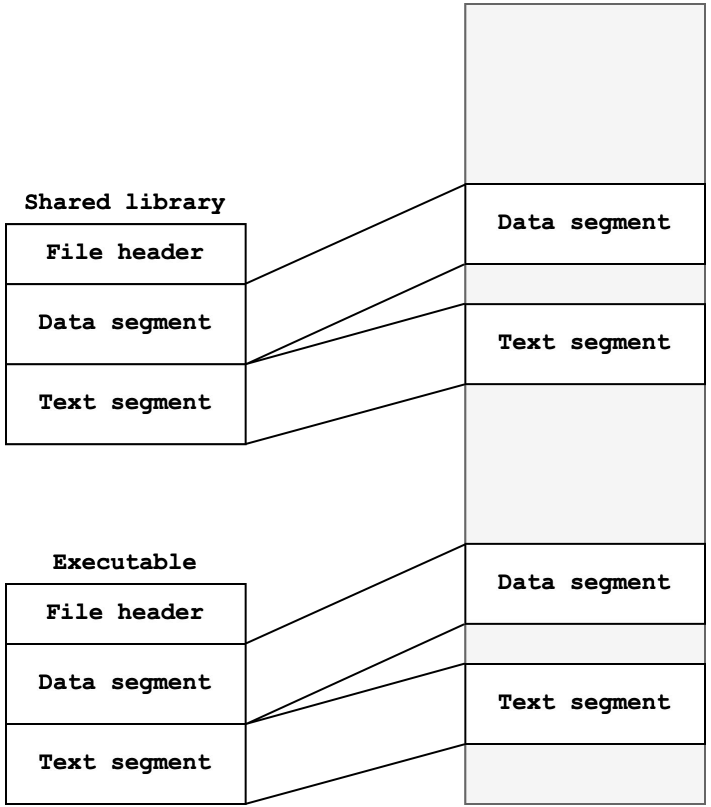
- Simple
- Library's code and data are directly copied to an output file, so the output is self-contained (doesn't depend on another file at runtime.)

Disadvantages of static libraries

- Duplicated code and data are copied to multiple binaries
- If you want to update library's code, you need to re-link binaries.

Dynamic library (1)

Dynamic libraries are loaded to memory along with the main executable.



Dynamic library (2)

Advantages of dynamic libraries

- Unlike static libraries, library's actual code and data are not copied to executables
- If you want to update a library, you can just replace the library file

Disadvantages of dynamic libraries

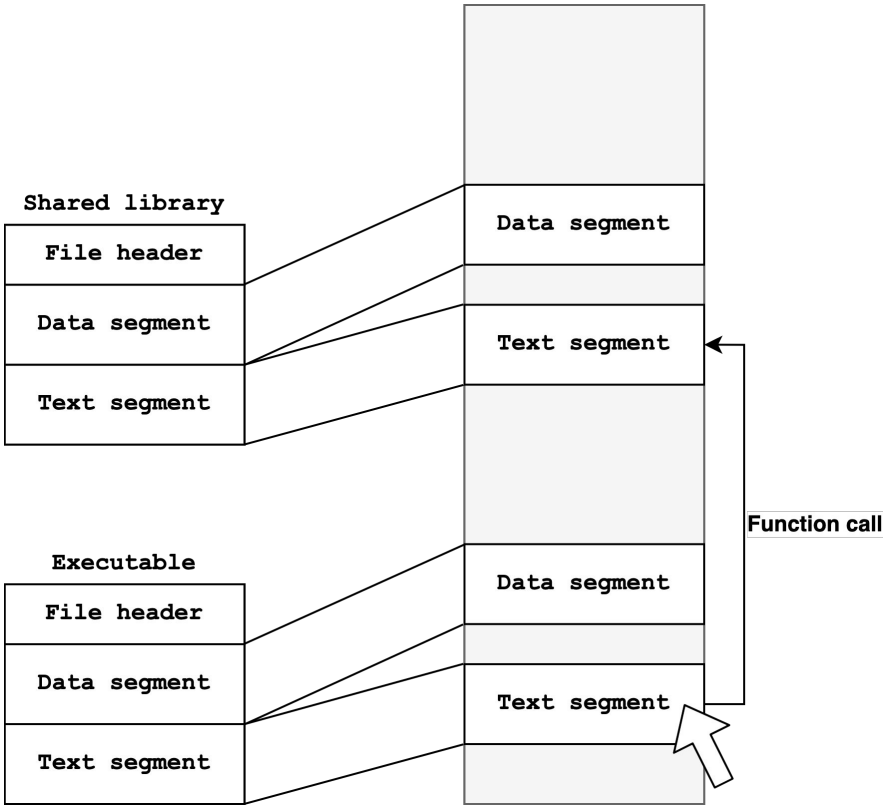
- Complex
- There is an overhead in process startup because the loader has to do more than just loading files into memory
- Copying the executable file by itself does not complete the installation
- If the behavior of the updated library is different than it was, the program will not run properly

Dynamic relocations

After loading the executable and the dynamic library into memory, the addresses of function calls between them need to be modified in memory.

It's called "dynamic relocation."

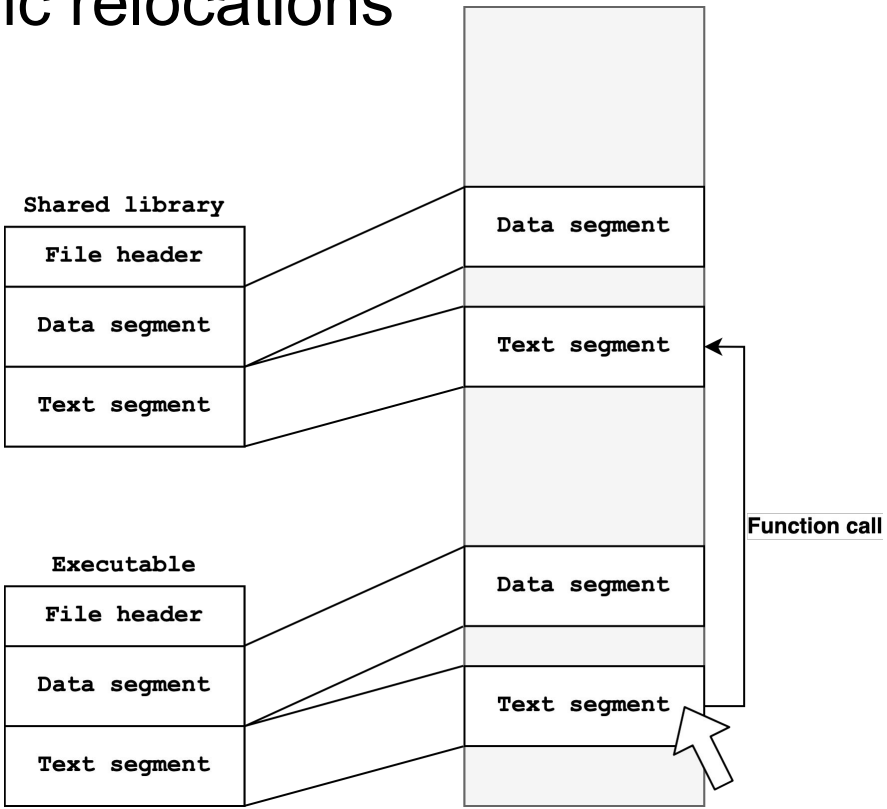
The linker can resolve relocations for those who refer locations in the same file. For external relocations, the linker creates dynamic relocations to let the program loader to apply them at load-time.



Problems with simple dynamic relocations

There are usually so many places where dynamic relocation should be applied. libc functions, for example, are called from everywhere.

Relocating everything would slow down the program loading.



PLT and GOT

PLT (Procedure Linkage Table)
GOT (Global Offset Table)

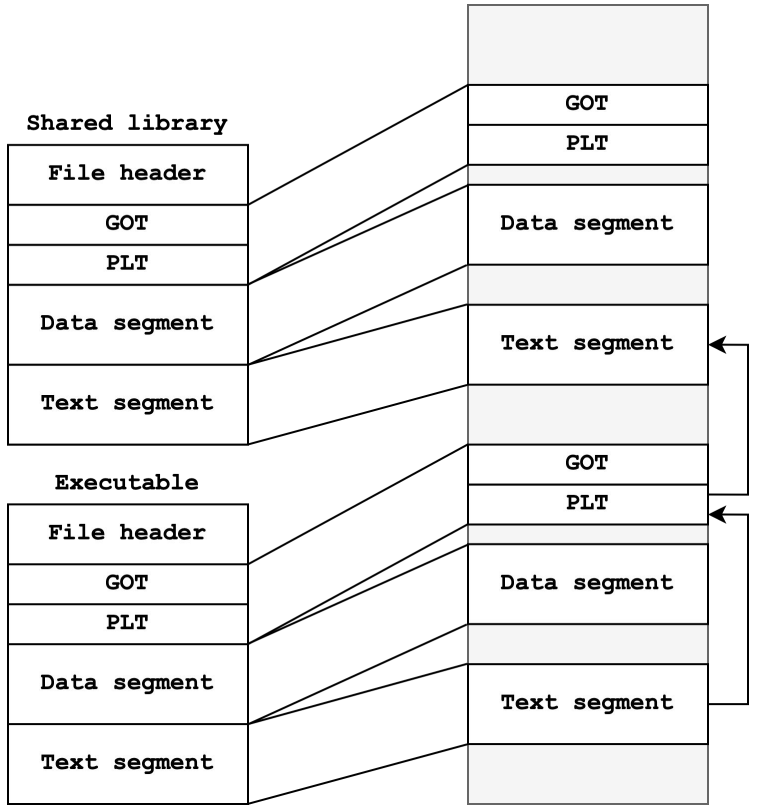
A single location that references other modules

Relative position between code and GOT/PLT in the same file is always fixed, even in memory

When you want to call a function in another module, jump to PLT first, then jump to another module from there

In this way, you can make text segment read-only and make the program loader to fix up only GOT and PLT entries.

PLT/GOT are created by the linker.



Dynamic libraries and linker

Here is what the linker has to do to link against dynamic libraries:

- Read the contents of dynamic libraries to find the symbols that are defined by the libraries
- Create a PLT or GOT for symbols in the dynamic libraires (If the call is within the same output file, there is no need to create a PLT or GOT)

Summary of linker's tasks

1. Reads object files, static libraries, and dynamic libraries given as arguments
2. Resolve symbols
3. Scan relocation tables to determine contents of PLTs and GOTs
4. Determine the layout of the output
5. Copy data from an input file to an output file
6. Apply relocations

Why does the linker's speed matter?

Because faster is better!

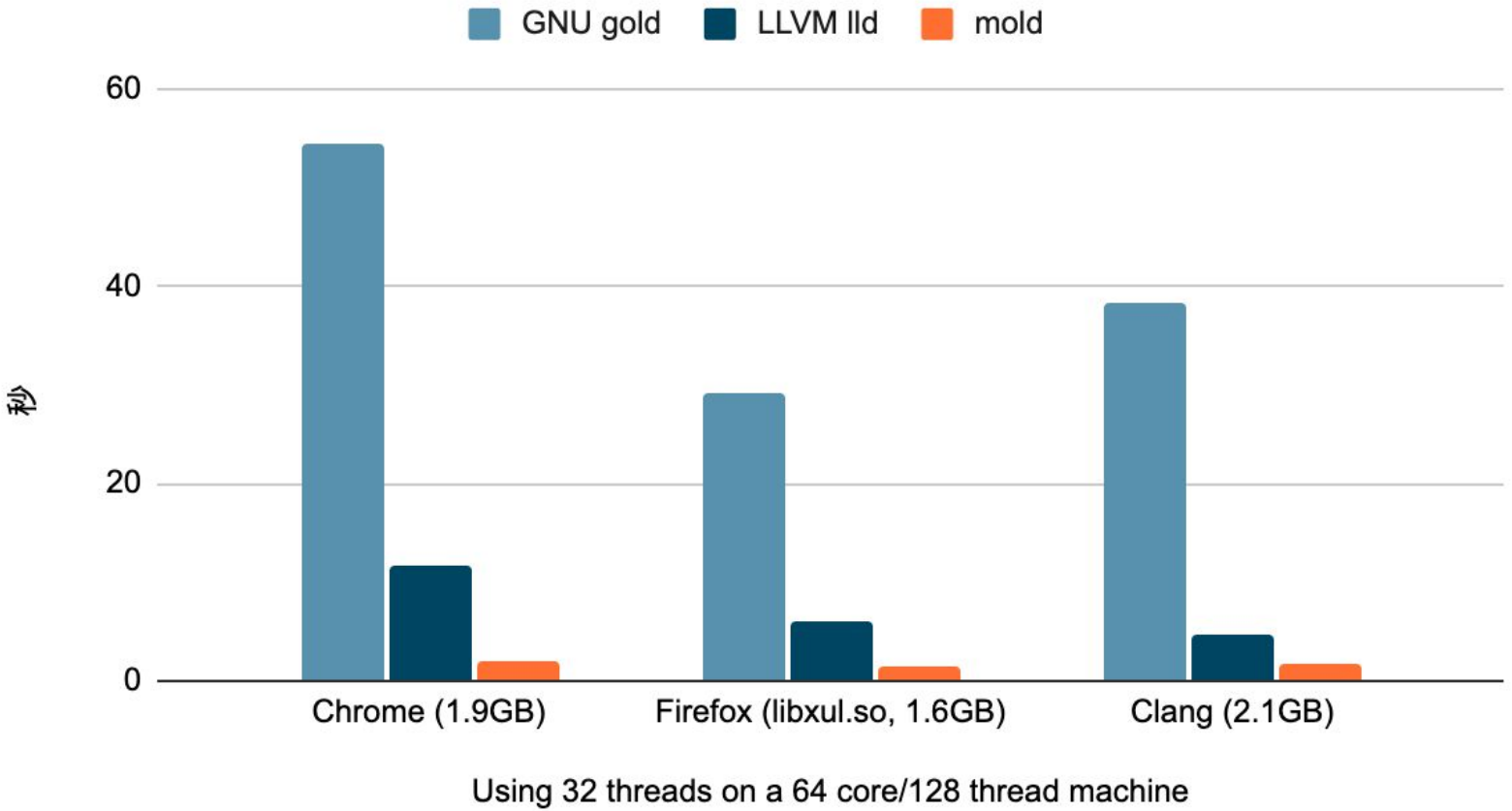
A build system such as make compiles only source files modified since the last build. If you change one file and rebuild, a compiler and a linker run.

Linking tends to be a bottleneck because the linker always takes all object files as arguments.

If your build takes 30 seconds, you would switch the window to start web browsing. But if it's only 3 seconds, you can wait. So it's not just saving 27 seconds for each build. It helps developers to maintain focus.

Performance comparison of mold, LLVM lld and GNU gold

Comparison of time required for linking



Speed of mold

mold's speed is about 1 second per 1 GB output file on a high-core-count modern x86 machine.

It's about twice as slow as copying the output file to another file with cp, which is extremely fast because the linker does substantially more work than cp does.

It's probably almost impossible to create a linker significantly faster than mold, as mold is already almost I/O-bounded.

Why is mold so fast?

Why is mold so fast?

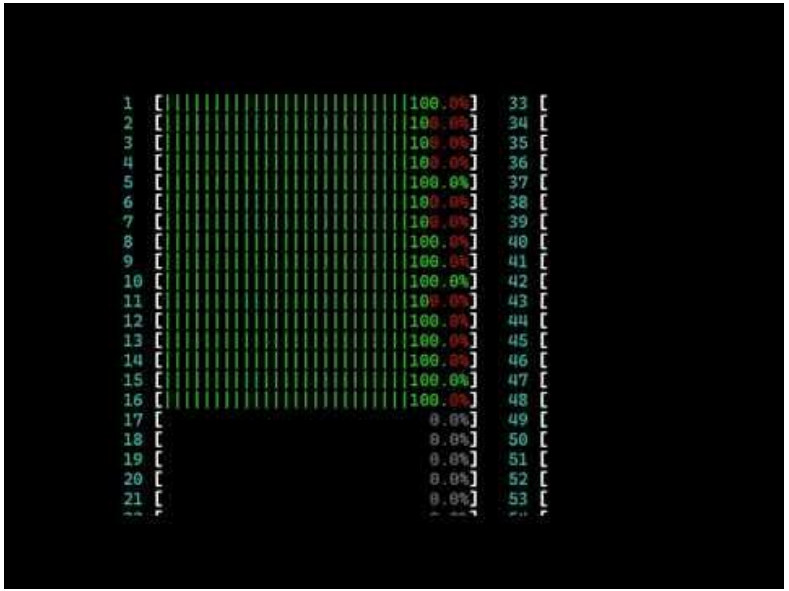
Directly use data on mmapped input files without creating intermediate representations

Parallelize all internal passes whenever possible

- Due to Amdahl's law, the performance of a parallelized program tends to be determined by the non-parallelized parts of the program
- For example, if a single-threaded part of a program consists 50% of the total execution time, it cannot be faster than 2x no matter how many cores you add

Use clever techniques to make it faster

Comparison with htop（lld and mold）



Scale of data

Here is a list of elements in input files and their numbers when linking Chrome (about 1.9 GB with debugging information)

There is a large amount of data of the same type, sometimes in the order of millions.

Data type	# of items
Relocations	62024719
Relocations against SHF_ALLOC sections	39496375
Sections	27543225
Symbols	23953607
Public defined symbols	10512135
Regular sections	10345314
Comdat groups	9914510
Mergeable strings	1579996
Public undefined symbols	1428149
Object files	30723

Data parallelism

Data parallelism is a parallelism to process large amounts of the same type of data in parallel

Scales well because there are usually no or very few dependencies between threads

Easier to understand than parallelization with complex synchronization mechanisms because it is just a for loop in which each iteration may run in parallel

Concurrent container

Concurrent Map

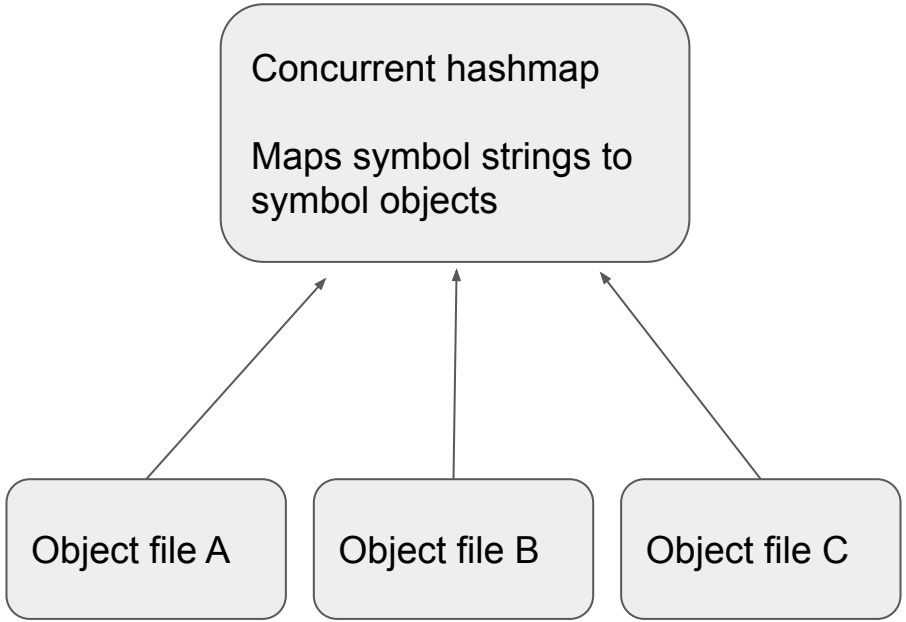
A map that allows elements to be inserted from multiple threads simultaneously

- It's usually implemented using sophisticated data structures and scales well with the number of threads.
- mold uses Intel TBB's tbb::concurrent_hash_map

https://github.com/oneapi-src/oneTBB

Symbol resolution in mold

Run a parallel for loop on the input file and add symbols to the parallel hash map at the same time



Map-reduce pattern

If you pass the --build-id option to the linker, the linker computes a cryptographic hash for an entire output file and embed it to the output file

Computing a cryptographic hash for a file is time consuming, so we split into two stages:

1. Consider the output file as being made up of 10 MiB records and compute SHA256 for each record -- This step can be done in parallel
2. Compute a SHA256 of SHA256 records

Miscellaneous speed-up techniques

glibc's default malloc doesn't scale well for a large number of cores

- We are using Microsoft's mimalloc
- Out of jemalloc, tbbmalloc, tcmalloc and mimalloc, mimalloc performed the best for us

It is faster to overwrite a file already in the buffer cache than to create a new file and write data to it

- If there is an existing executable file, overwrite it.

If a large number of files are mmap'ed, it takes several hundred milliseconds to terminate the process

- As a workaround, fork a child process and do the actual processing in that process, and exit from the parent process as soon as the child process outputs a file
- Exit from the child process takes several hundred milliseconds, but it is not a problem because it is not an interactive process

Hints on writing faster programs

Hints on writing faster programs

Don't guess, measure.

- Speculations are usually wrong, so don't waste time to optimize code that doesn't matter.

Don't try to write faster code. But rather, design your data structures in such a way that the program naturally becomes faster.

- Data is usually more important than code

Implement multiple algorithms and choose the fastest one

- Thinking is not enough to know which is better, so just do it.

Write the same program several times

- Learn from the first implementation, then use that knowledge to reimplement
- There are a few types of optimizations that you cannot implement without re-doing everything from scratch
- Developing it a second or third time is faster, so your time on the first iteration won't be completely wasted