New LLD linker for ELF

A high performance linker from the LLVM project

Rui Ueyama
Google

History

We started rewriting COFF (Windows) linker in May 2015.

Ported the proven design to ELF (Unix) in July 2015.

Capabilities

Has a GNU-compatible command line interface.

Can build the full x86-64 FreeBSD userland (buildworld) with a few small workarounds.

PPC64, AArch64 and MIPS support is in progress. LLD can link small programs for these platforms.

Linker script support is in progress.

Design goals

Speed

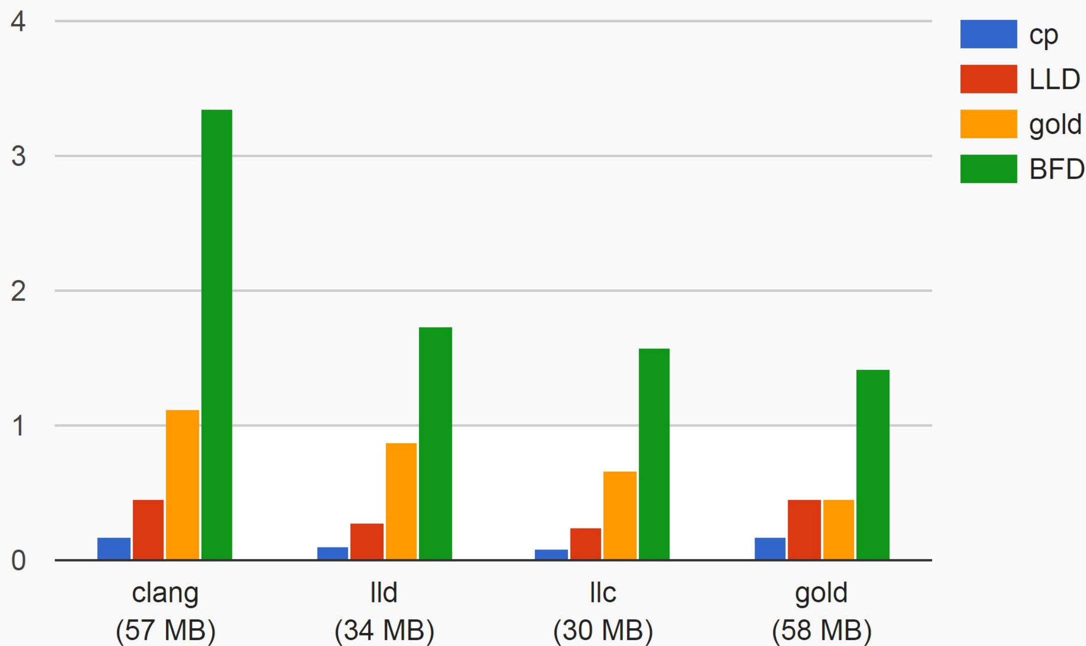
Simplicity

Extensibility

Linking medium-sized programs

Built Clang, LLD, llc and GNU gold with LLD.

Multi-threading support was enabled if the linker supports it. Measured on Xeon E5-2680 2.8 GHz processor. All files are on SSD.



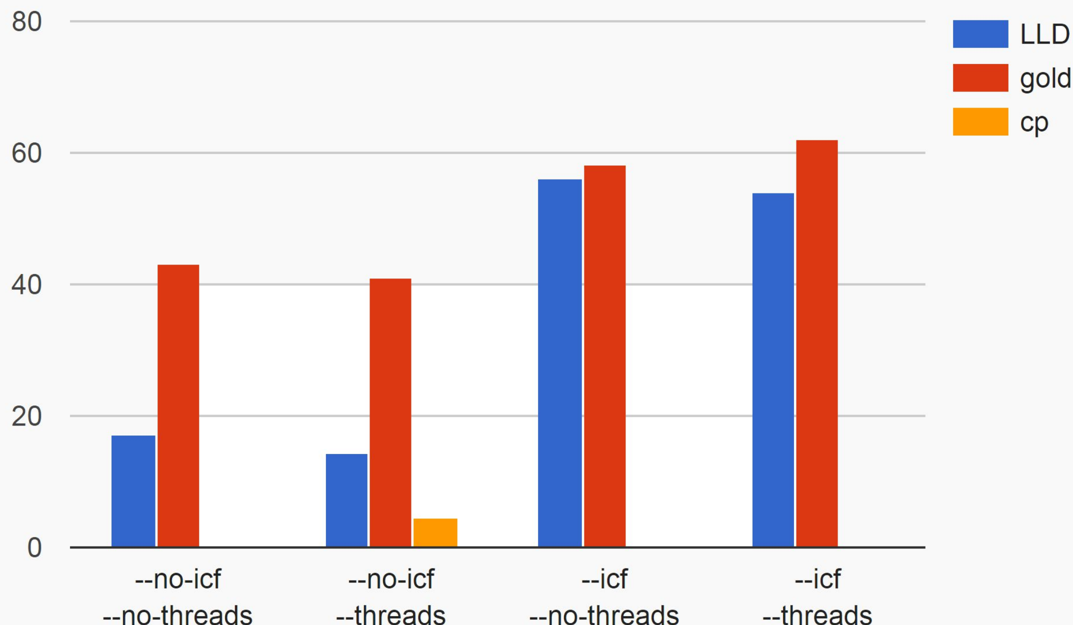
Linking a large program

Built Chrome with debug info. The resulting executable size is about 2 GB.

ICF is an optimization to detect identical sections to eliminate them. It is computationally intensive.

"cp" sets the lower bound. It takes 4.4 seconds to copy the 2 GB file on my SSD.

LLD creates it in 14.4 seconds at its best. gold at least takes 41.1 seconds.



Speed by design

Three key design choices for speed:

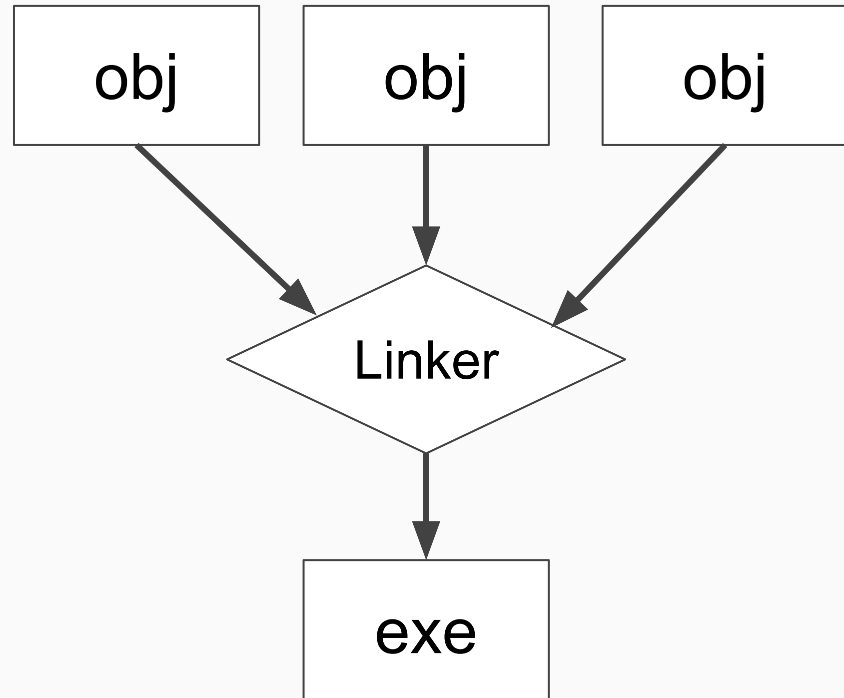
- Efficient archive file handling
- Do less rather than do it efficiently
- If a costly operation is inevitable, do it only once

What is linking?

To describe the difference of the archive file handling between the traditional Unix linker and LLD, I'll describe the traditional linker's semantics first.

A primitive form of the linker is basically a "cat" program with additional code to fix up addresses referring other compilation units.

Object files have global symbols and relocations for fix up.

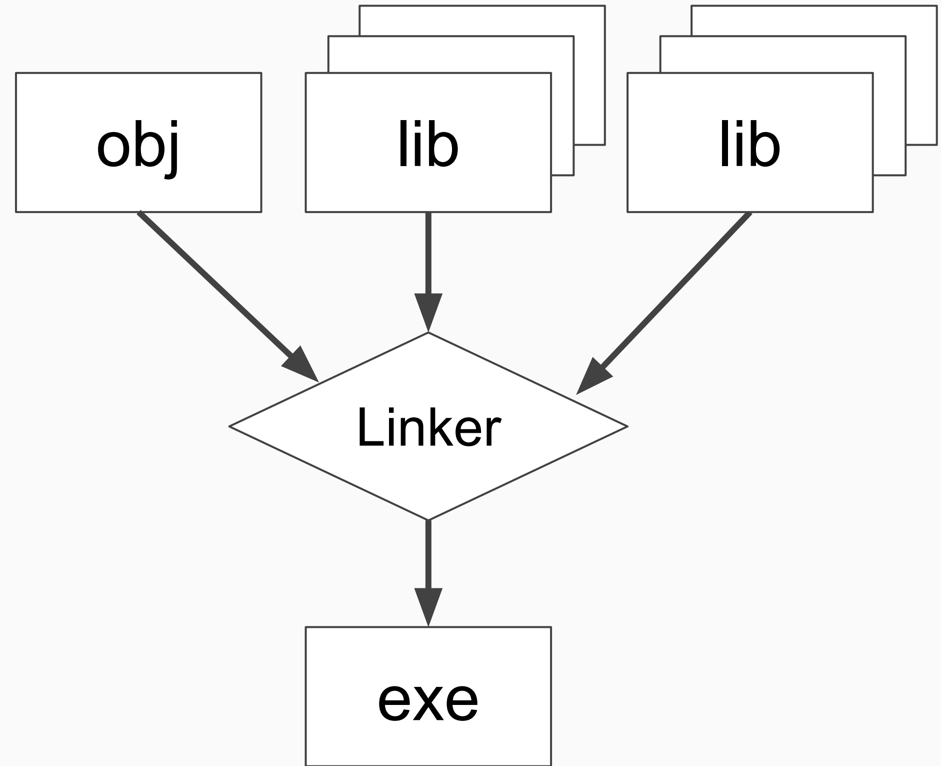


What is linking?

Managing the list of object files is painful, so the static archive file was invented.

Archive files contain object files. The linker extracts object files that provide definitions for undefined symbols.

Archive files have their own symbol tables in their headers for fast symbol lookup.



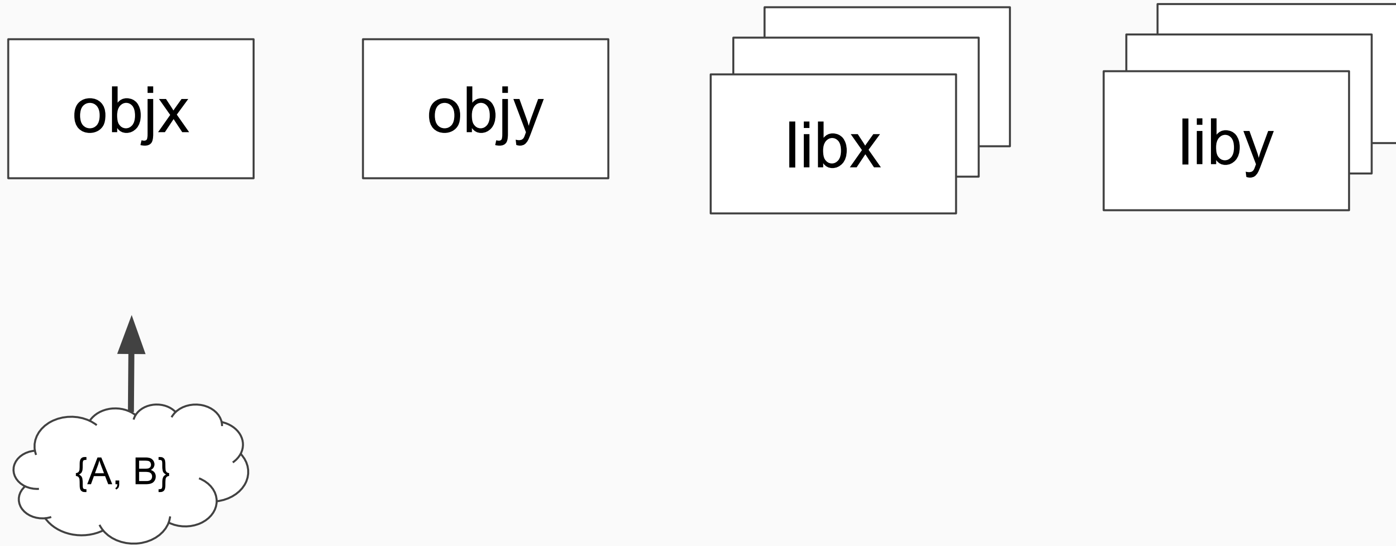
A weakness of Unix linker's semantics

The linker visits files sequentially. What it does depends on file type.

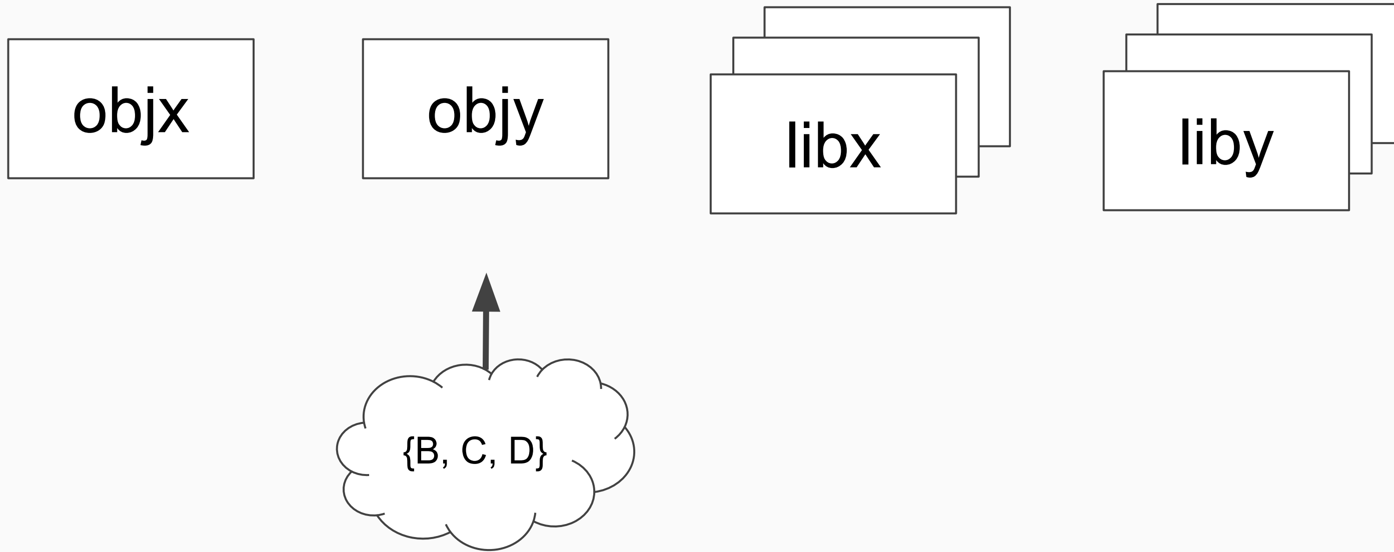
- If it is an object file, link it.
- If it is an archive file, extract object files that have definitions for undefined symbols and link them.

That means the linker has a set of undefined symbols and will read files until the set becomes empty.

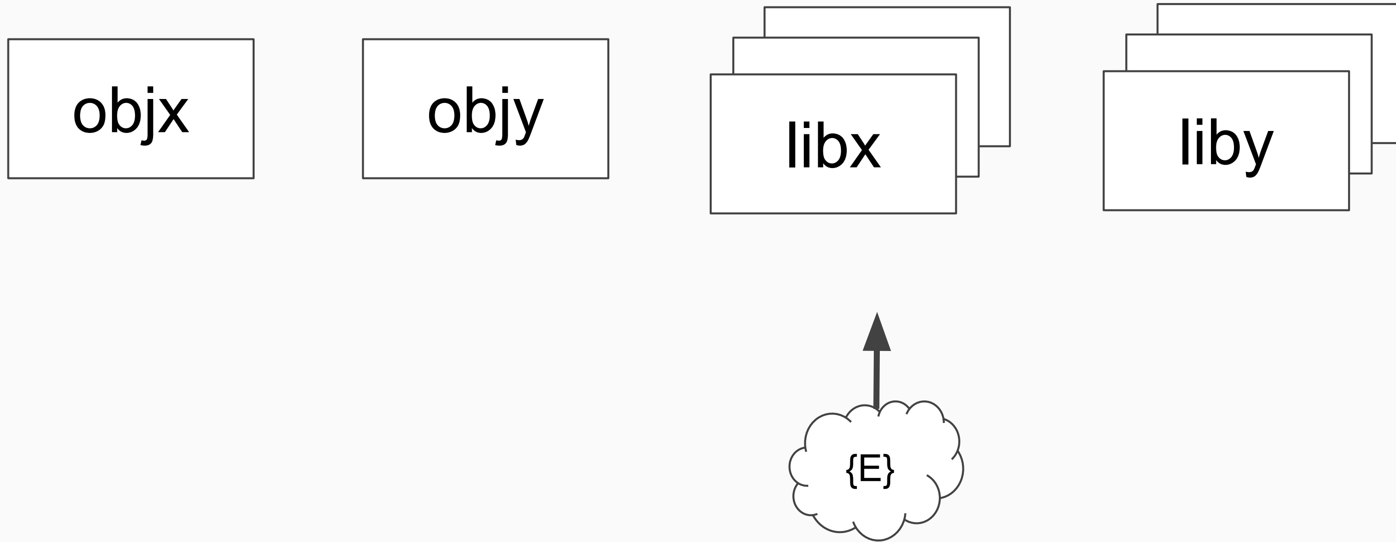
Sequential file handling



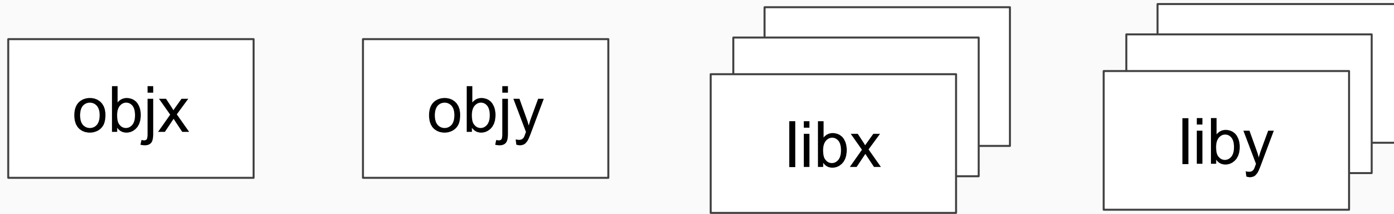
Sequential file handling



Sequential file handling

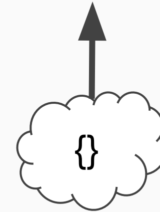


Sequential file handling



When the set becomes empty, the link is done.

The linker then concatenates all object files it has visited, fixes up relocations, and writes it down to a file.



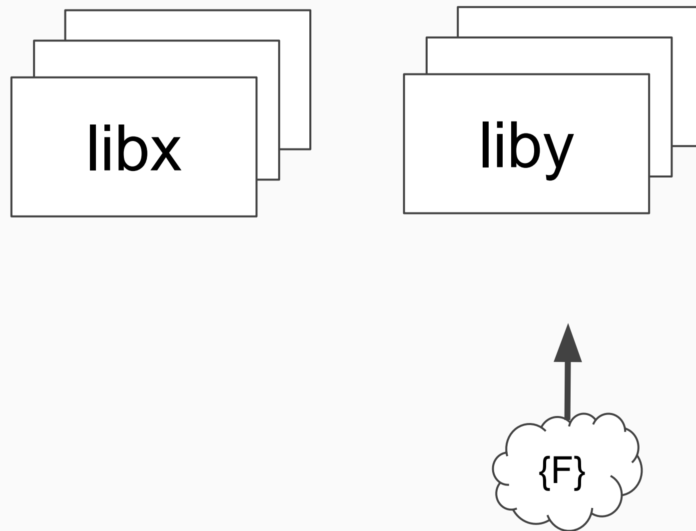
Linking mutually-dependent archive files

Linking mutually-dependent archive files is tricky because the linker visits files only sequentially.

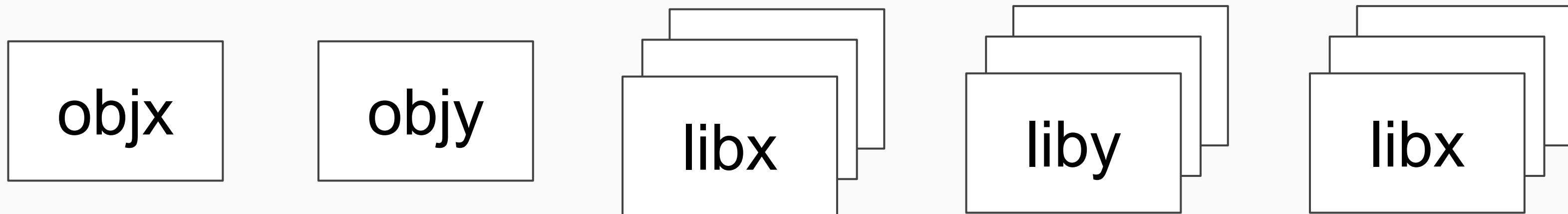
In the case to the right, the link will fail even if libx provides a definition of F.

There are solutions.

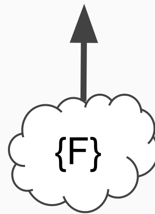
You may either specify the same file more than once or use the special options --start-group and --end-group to resolve mutual dependencies.



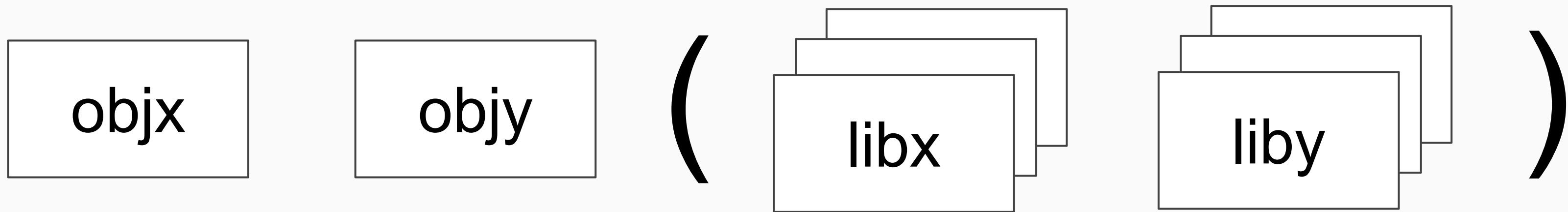
Specifying the same file more than once



Obviously, it is awkward. Unless you know how many repetitions are needed, it may fail.

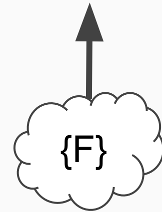


Use --start-group and --end-group



The linker repeats the files between --start-group and --end-group until no new symbols are added to the set.

It works, but it is inefficient as the linker visits the same file many times (at least twice).



LLD's semantics on archive files

In both cases it is awkward to handle mutually-dependent archive files.

A natural question would be "*why does the linker have to visit sequentially?*"

The answer is "*it doesn't have to.*" That's the way how LLD handles archive files.

objx

objy

libx

liby

A: Defined
B: Undefined

objx

objy

libx

liby

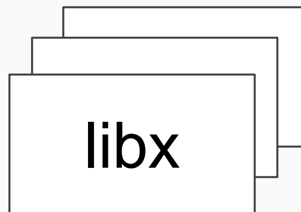
A: Defined
B: Undefined

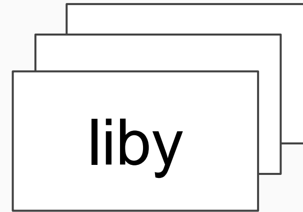
A: Defined
B: Defined
C: Undefined
D: Undefined

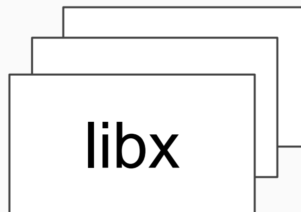
objx

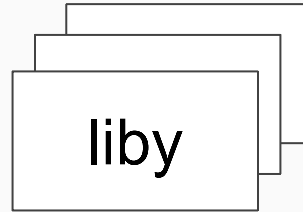
A: Defined
B: Undefined

objy

A: Defined
B: Defined
C: Undefined
D: Undefined

libx

A: Defined
B: Defined
C: Defined
D: Undefined
E: Undefined
F: Lazy (obja in libx)
G: Lazy (objb in libx)
H: Lazy (objb in libx)

liby





A: Defined
B: Undefined

A: Defined
B: Defined
C: Undefined
D: Undefined

A: Defined
B: Defined
C: Defined
D: Undefined
E: Undefined
F: Lazy (obja in libx)
G: Lazy (objb in libx)
H: Lazy (objb in libx)

A: Defined
B: Defined
C: Defined
D: Defined
E: Defined
F: Lazy (obja in libx)
G: Defined
H: Lazy (objb in libx)

LLD's semantics on archive files

Is this completely compatible with the traditional semantics?

- No. You can observe the difference if you carefully craft archive files to exploit it intentionally.

Is it going to cause problems?

- Unlikely. We don't know any program that fails to link because of this.

It is a very good tradeoff.

Numbers you want to know

Today's programs are large. Chrome with debug info is almost 2 GB. In order to produce the executable, the linker reads and processes

- 17,000 files,
- 1,800,000 sections,
- 6,300,000 symbols, and
- 13,000,000 relocations.

LLD does this in 14.4 seconds. Copying 2 GB on my SSD takes 4.4 seconds, so we only have 10 seconds left.

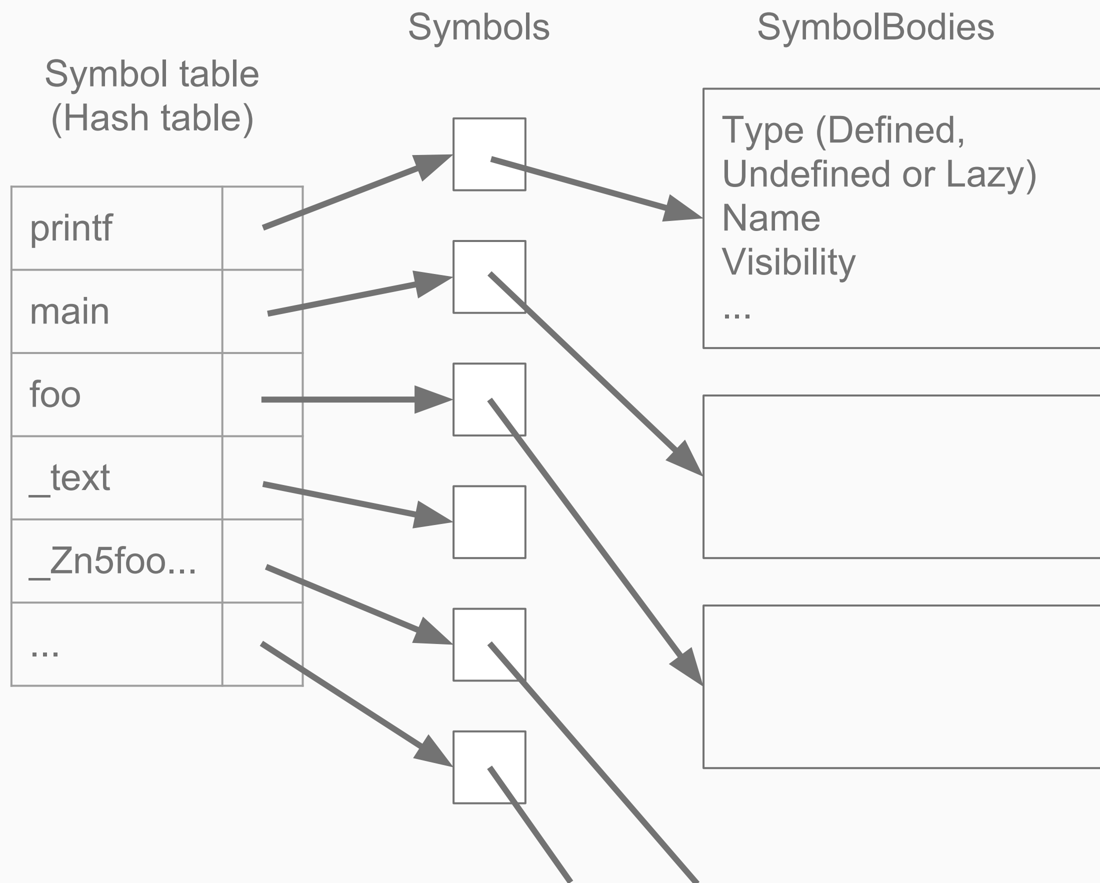
Efficient symbol handling

Inserting 6.6 million strings to a hash table takes 1.5 seconds. We don't want to do that more than once.

We separate symbol into two: Symbol and SymbolBody. Symbols are handles for SymbolBodies. SymbolBodies are container of actual data.

For each new string, we look in the hash table only once.

Symbol is just a pointer. Each SymbolBody has a pointer to a Symbol.

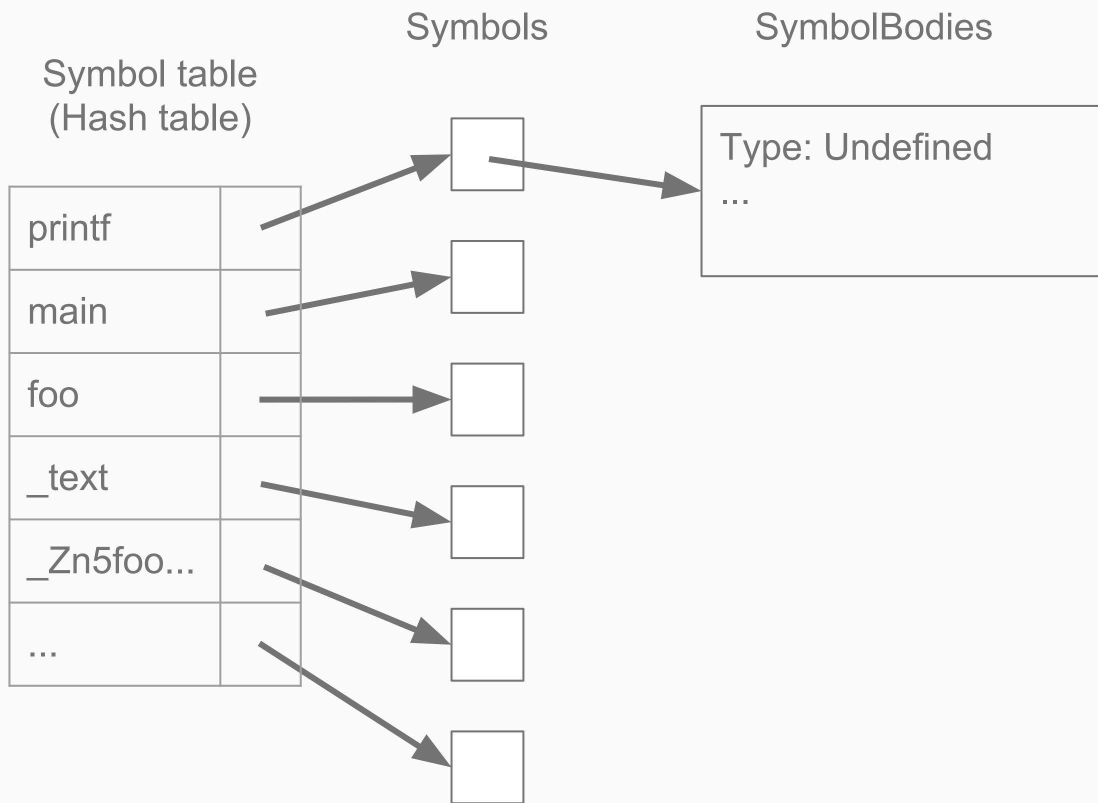


Pointer mutation is name resolution

Symbol resolver's job is to mutate pointers so that they point to the "best" symbols of all known ones.

Preference is binary relations:
Undefined < Lazy < Defined

The linker creates SymbolBodies for all symbols for each object file and adds them to the symbol table. Symbol pointers are then updated according to the symbol preferences.

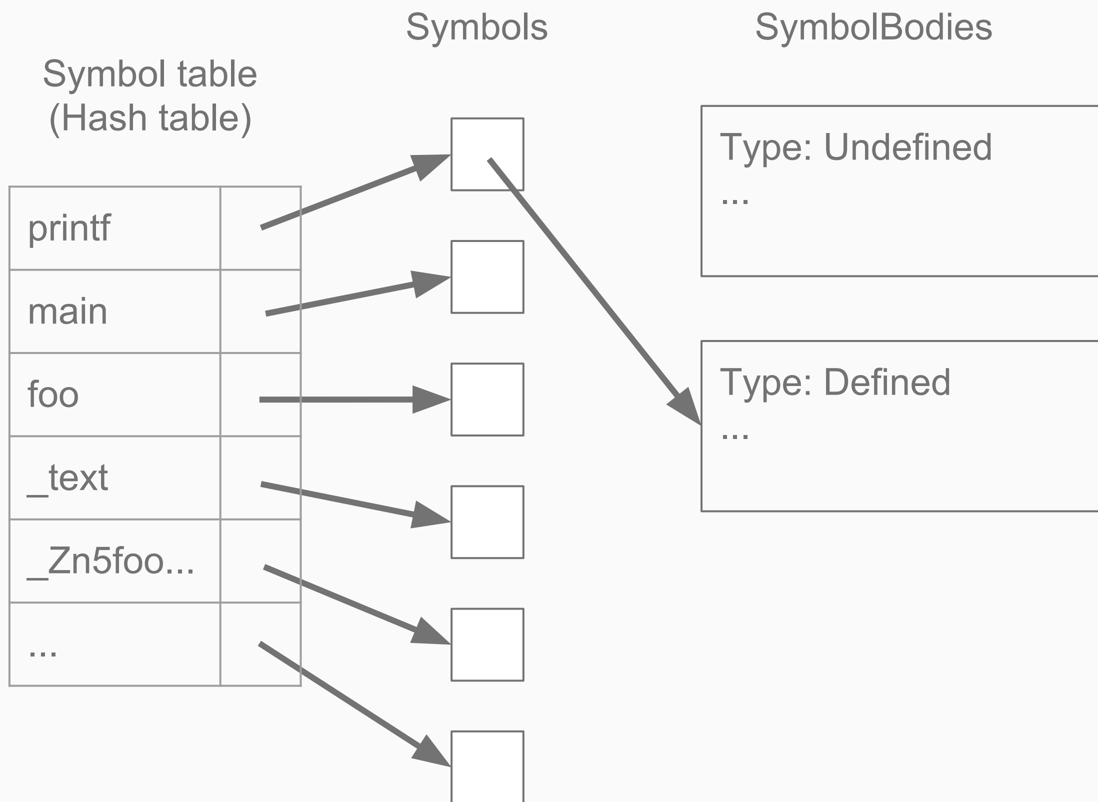


Pointer mutation is name resolution

Symbol resolver's job is to mutate pointers so that they point to the "best" symbols of all known ones.

Preference is binary relations:
Undefined < Lazy < Defined

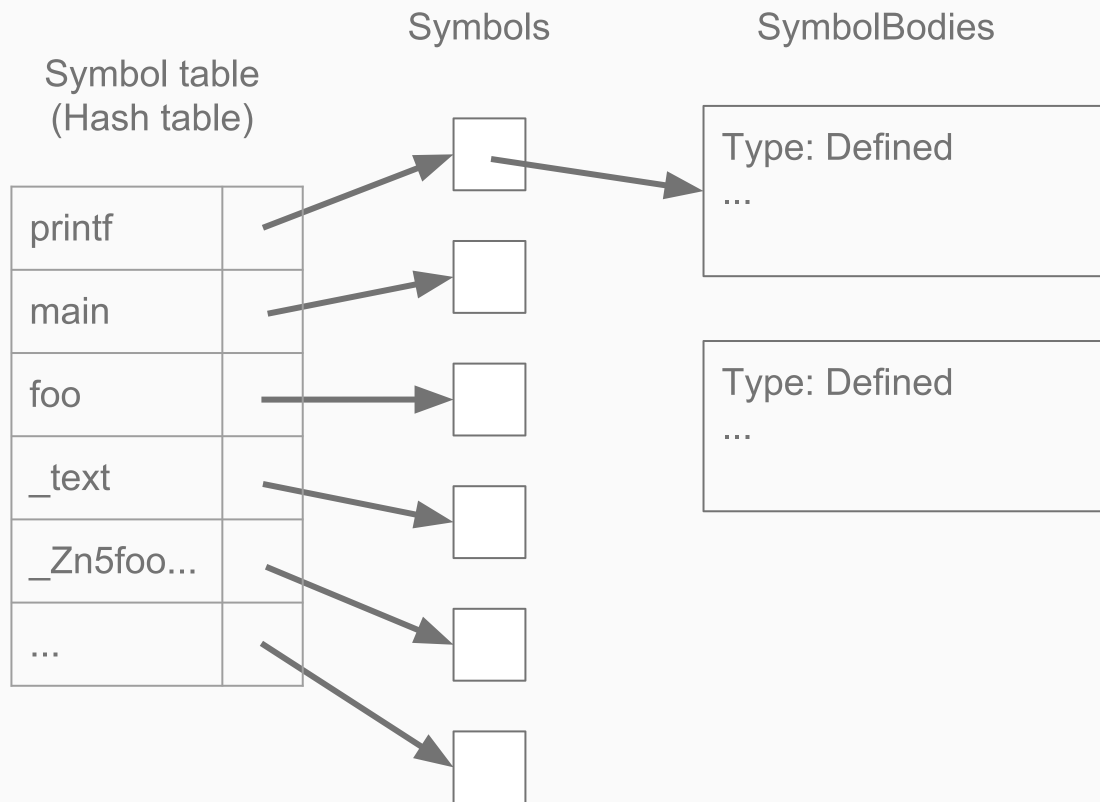
The linker creates SymbolBodies for all symbols for each object file and adds them to the symbol table. Symbol pointers are then updated according to the symbol preferences.



Handling duplicate definition errors

If two Defined symbols A and B are added, and if neither A < B nor B < A, they are "conflicting" symbols.

In that case, the linker reports an error.

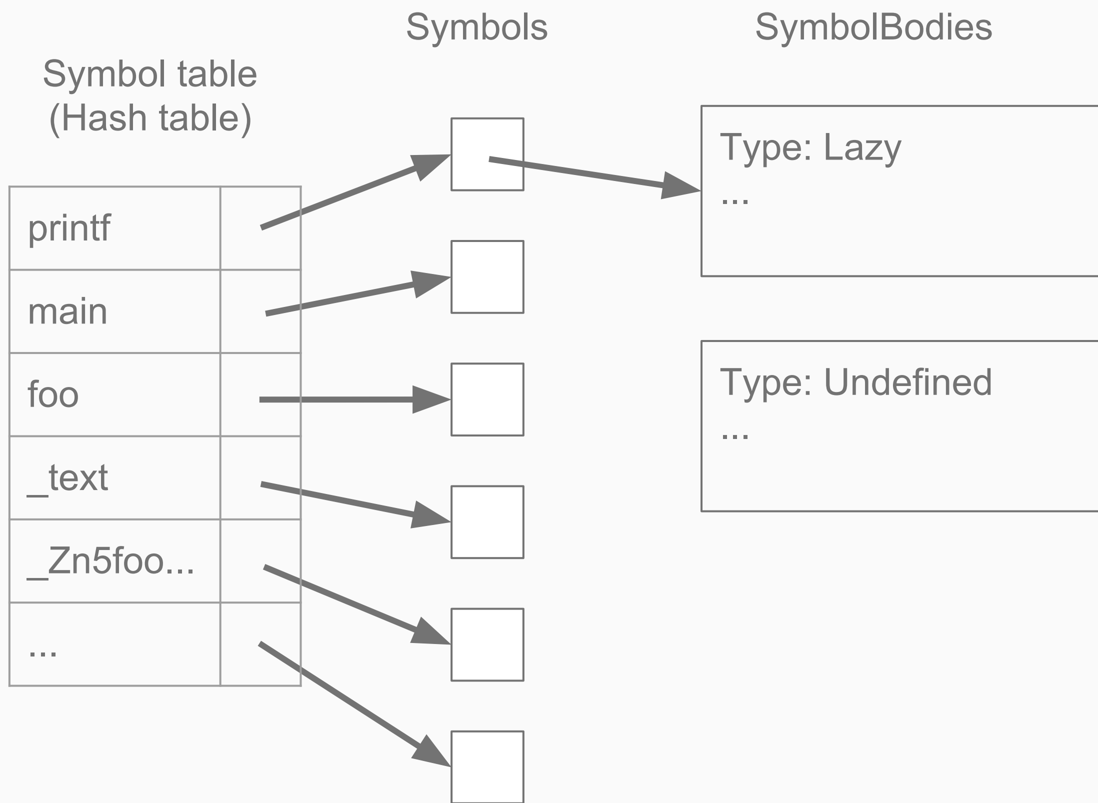


Extracting files from archive files

This is where object files in archive files are extracted.

If one symbol is Lazy and the other is Undefined, then we need to extract the object file for the Lazy symbol to get a real Defined symbol of the same name.

The resolver triggers Lazy to let it load the object file.

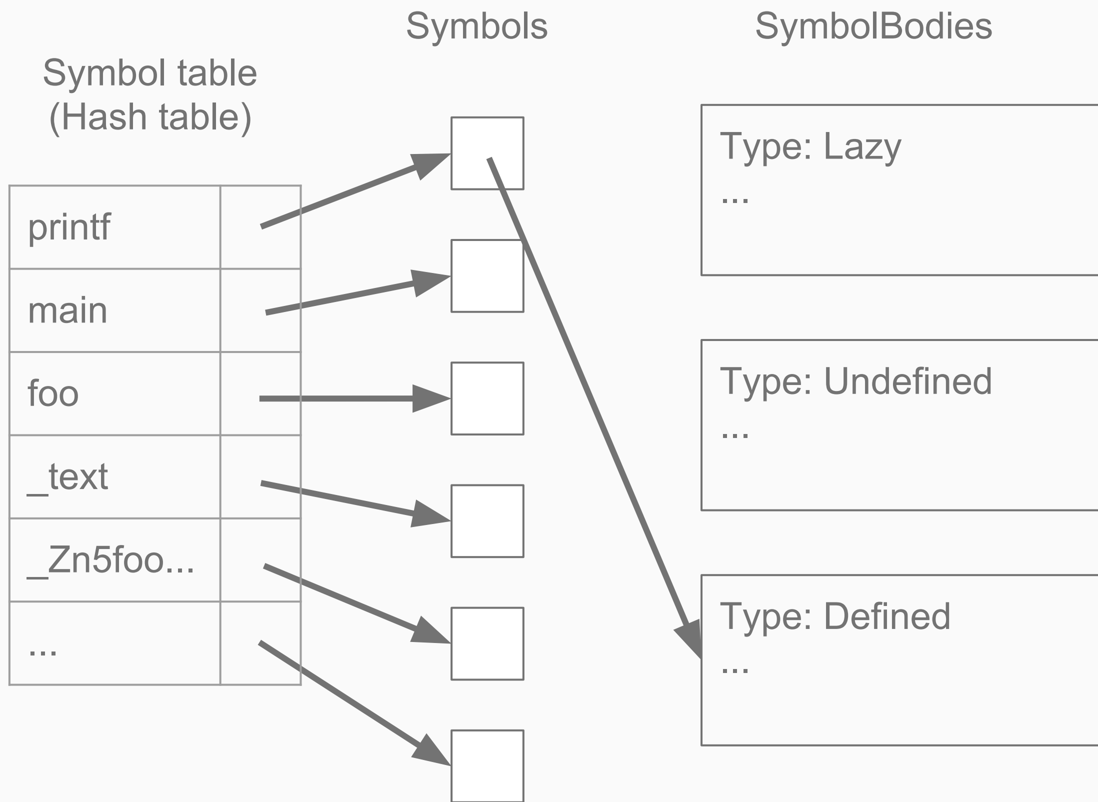


Extracting files from archive files

This is where object files in archive files are extracted.

If one symbol is Lazy and the other is Undefined, then we need to extract the object file for the Lazy symbol to get a real Defined symbol of the same name.

The resolver triggers Lazy to let it load the object file.

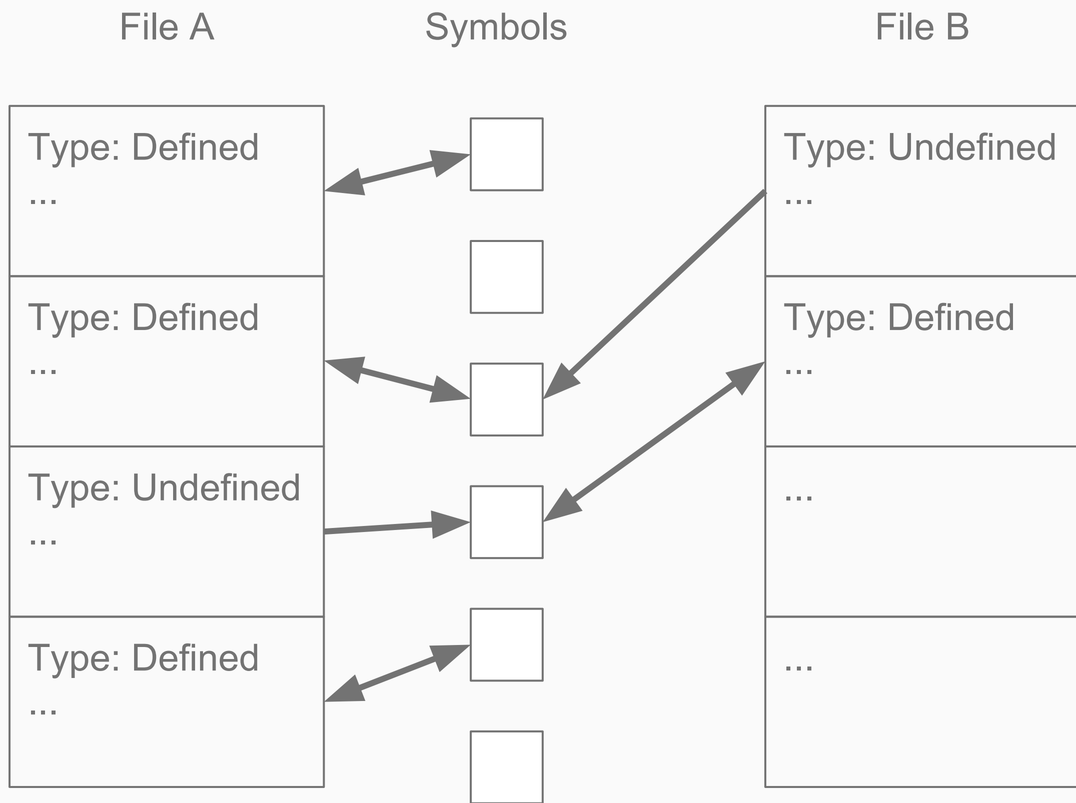


Obtaining symbol resolution results

SymbolBodies are created for each object file.

To apply relocations, we need to know how symbols were resolved. Because the number of relocations is large, it must be doable very efficiently.

It can be done with two pointer dereferences because only the most preferred SymbolBodies are pointed to by Symbols. It's very cheap!



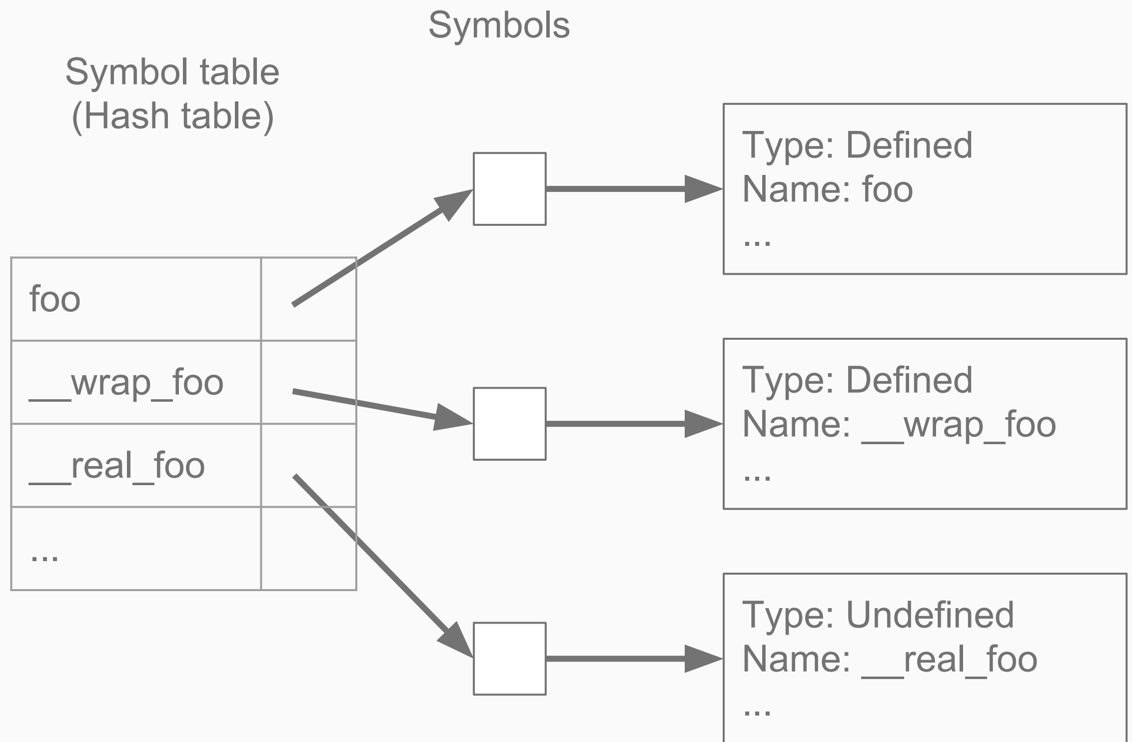
Even tricky features are easy to implement

This data structure allows us to implement symbol renaming very efficiently and easily.

If --wrap=*foo* is given, any undefined reference to *foo* will be resolved to "__wrap_*foo*". Any undefined reference to "__real_*foo*" will be resolved to *foo*.

Complicated? It is actually not.

We first resolve all symbols normally and then swap pointers as shown to the right.



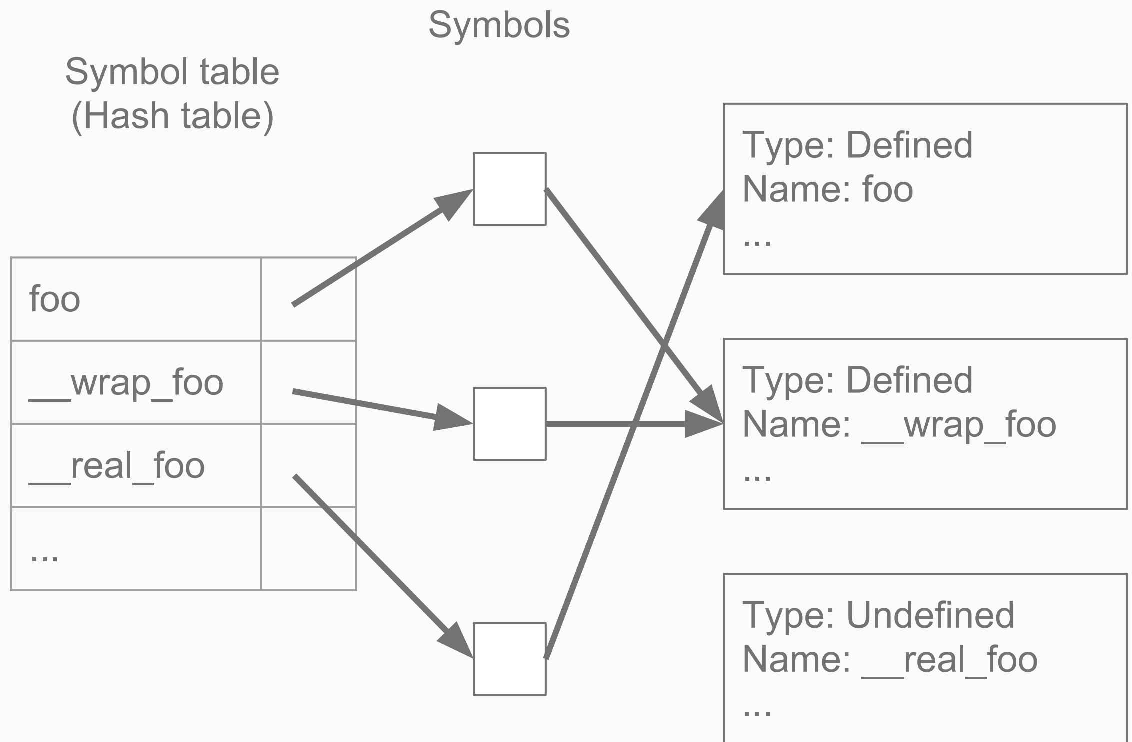
Even tricky features are easy to implement

This data structure allows us to implement symbol renaming very efficiently and easily.

If --wrap=*foo* is given, any undefined reference to *foo* will be resolved to "__wrap_*foo*". Any undefined reference to "__real_*foo*" will be resolved to *foo*.

Complicated? It is actually not.

We first resolve all symbols normally and then swap pointers as shown to the right.



Link-Time Optimization (LTO)

LTO support is in progress.

For LTO, LLVM bitcode files are given to the linker instead of object files in the native (ELF) format. The linker resolves all symbols normally. Once all symbols are resolved, it passes all bitcode files to LLVM to get one gigantic object file.

It then replace all bitcode symbols with object file symbols by pointer mutation to continue linking as if it were given native object files from beginning.

Future work

- Make it usable as a system's default linker
  - FreeBSD is likely to be the first operating system to adopt LLD
  - Comprehensive linker script support is needed to link the kernel
- Optimize performance
  - Although it's already pretty fast, the current performance numbers are naturally achieved by design, so we haven't done any serious optimization yet
  - Parallelize using threads
- Support more platforms