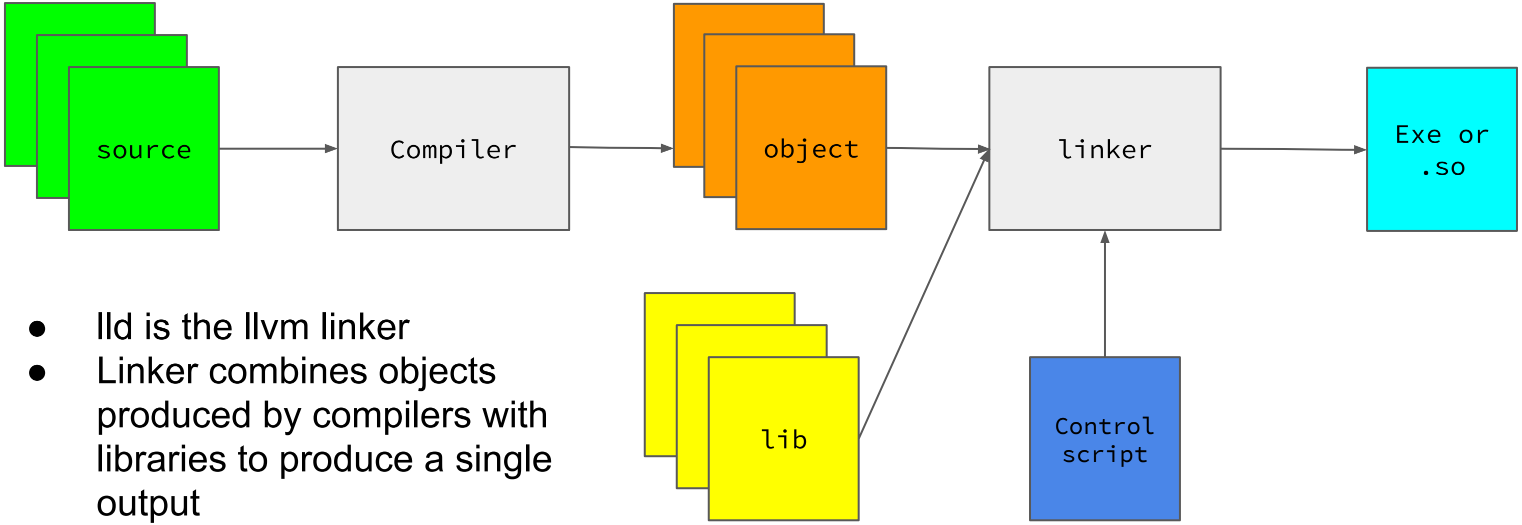
LLD from a user’s perspective

Peter Smith, Linaro

Introduction and assumptions

- What we are covering Today
  - What is lld and why might I want to use it?
  - How to build and use lld as a substitute for GNU ld or Gold.
  - What can I expect if I use lld?
  - How can I contribute to lld?
- What we won’t be covering
  - Coff and Mach-O versions of lld
- About me
  - Currently adding support for ARM to the LLD ELF linker
  - Background in ARM toolchains

What is lld?



- lld is the llvm linker
- Linker combines objects produced by compilers with libraries to produce a single output
- Other linkers you may have heard of are ld and gold

What is lld?

- Since May 2015, 3 separate linkers in one project
    - ELF, COFF and the Atom based linker (Mach-O)
    - ELF and COFF have a similar design but don’t share code
    - Primarily designed to be system linkers
        - ELF Linker a drop in replacement for GNU ld or gold
        - COFF linker a drop in replacement for link.exe
    - Atom based linker is a more abstract set of linker construction tools
        - Only supports Mach-O output
    - Uses llvm object reading libraries and core data structures
- Key design choices
    - Do not abstract file formats (c.f. BFD)
    - Emphasis on performance at the high-level, do minimal amount as late as possible.
    - Have a similar interface to existing system linkers but simplify where possible

Why use lld?

- Performance
  - LLD can be significantly faster than GNU ld and gold.
  - Xeon E5-1660 3.2 Ghz, 8 cores on an ssd, rough performance.
  - Your mileage may vary, the figures below are from a quick experiment on my machine!
  - Smaller programs or those that make heavier use of shared libraries yield much less of a difference. The linker output files below range in size from roughly 1 to 1.5 Gb.

program/linker	GNU ld	GNU gold	lld
Clang static dbg	**1**m**17**s, **7**s non-dbg	**23**s, **2.5**s non-dbg	**6**s, **0.9**s non-dbg
libxul.so	**27**s	**10**s	**2.7**s
chromium	**1**m54s	**15**s	**3.74**s

Why not to use lld?

- Can be used in place of GNU ld or gold but does not guarantee identical results given the same inputs
- Not all ld or gold features implemented
- Limited amount of users and hence less well tested compared to ld and gold
- Limited number of targets supported
- Not aware of anyone using lld for embedded systems yet
- You are happy with the existence performance of your linker

Building lld

- No binary packages available for lld, only option is to build from source
- Same tool and library dependencies as llvm
  - gcc 4.8, clang 3.1, MSVC 2015, cmake 3.4.3, make or ninja

```
$svn co http://llvm.org/svn/llvm-project/llvm/trunk llvm
$cd llvm/tools
$svn co http://llvm.org/svn/llvm-project/lld/trunk lld
$cd ../../..
$mkdir build
$cd build
$cmake -G “Ninja” -DCMAKE_BUILD_TYPE=”Release”
-DCMAKE_INSTALL_PREFIX=”your_preferred_location” ../llvm
$ninja lld
$ninja install
```

Using lld (1)

- Output of build will include generic lld executable and a symlink to lld called **ld.lld**
- Generic lld executable requires a **-flavor=”gnu”** option to behave like GNU ld
- The “gnu” flavor is inferred if the executable name is **ld.lld** or **ld**
- Intention is that lld will be invoked by a compiler driver that provides a GNU ld compatible command line
- We need to make the compiler driver invoke lld rather than GNU ld

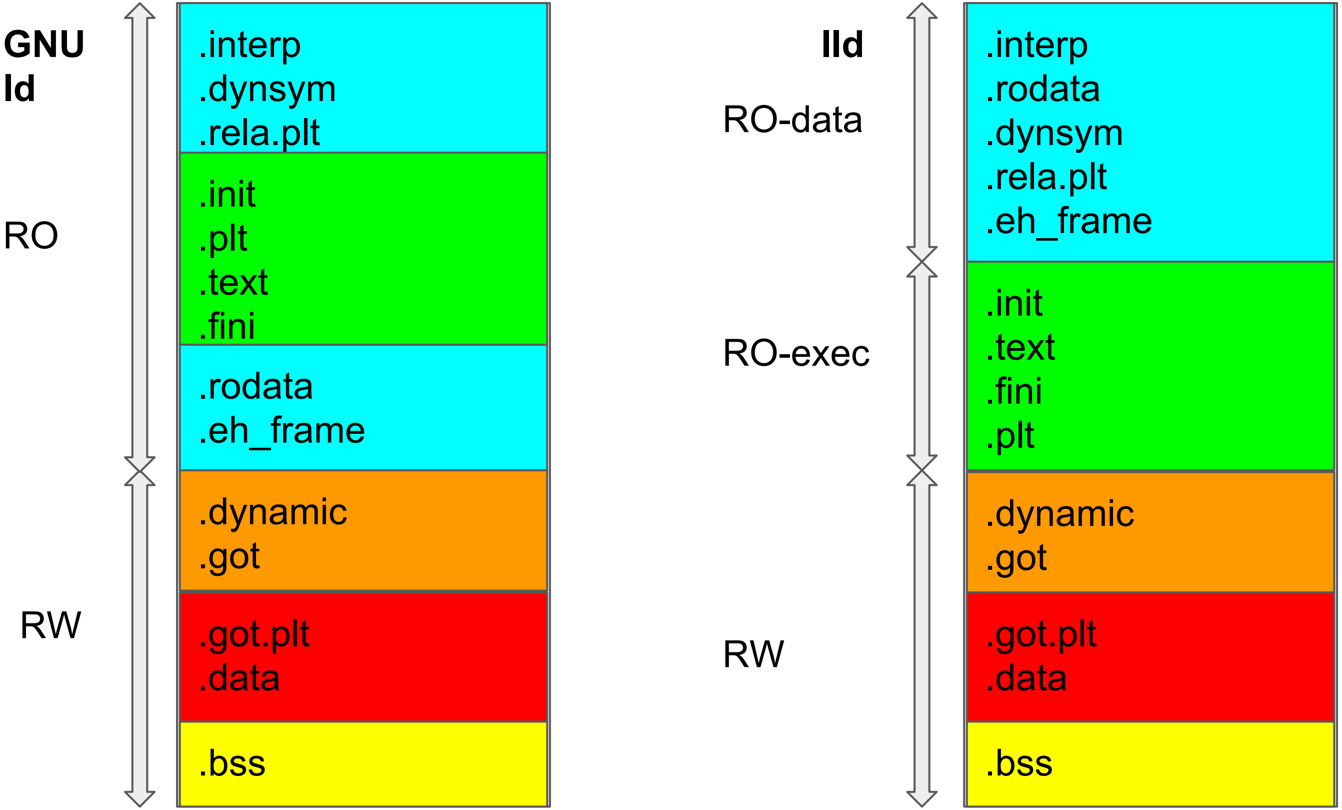
Using lld (2)

- Most reliable way is to symlink the system **ld** to **ld.lld**
  - On many systems **ld** is already a symlink to **ld.bfd** or **ld.gold**
  - Works with both gcc and clang drivers
  - Beware of build systems that include their own
- When clang is the compiler driver **-fuse-ld=lld** can be used to get the driver to invoke the linker using **ld.lld**
  - No need to symlink ld
  - **-fuse-ld=lld** not accepted by gcc

GNU ld and lld differences

- Archive search order
    - GNU ld and lld handle archive searching in a different way that in rare cases can result in a different library member being selected.
        - Only relevant if you have more than one library member defining the same global symbol with each library in a different archive.
        - Details of library search in http://lld.llvm.org/NewLLD.html
- No default linker script
    - Separate code path when no linker script given
- Default output section creation and ordering
    - lld does not attempt to match the GNU ld default script or inference rules
    - Read only data before executable data, 3 loadable segments rather than 2.
- Some command line options accepted but ignored
    - See Options.td in llvm/tools/lld directory

Example image layout differences



GNU ld

RO

.interp
.dynsym
.rela.plt

.init
.plt
.text
.fini

.rodata
.eh_frame

RW

.dynamic
.got

.got.plt
.data

.bss

lld

RO-data

.interp
.rodata
.dynsym
.rela.plt
.eh_frame

RO-exec

.init
.text
.fini
.plt

RW

.dynamic
.got

.got.plt
.data

.bss

LLD status

- Amd64
  - Most mature and most heavily tested
  - Several build bots for linking and running clang, llvm, tools and test suite with lld as the linker
  - As of January 2017 FreeBSD base system kernel + userland can link with lld
  - Effort ongoing to go through Poudriere ports 20k of 26k linking successfully
    - Progress tracked in https://llvm.org/bugs/show_bug.cgi?id=23214
- AArch64
  - Little endian support only
  - Build bot running that links and runs clang, llvm tools and test suite with lld as the linker
- ARM
  - Little endian support only
  - Missing range extension thunks which limits size of output to branch relocation range

LLD status

- Mips
    - Actively maintained, last public status message that I can find was that all Single and Multi source tests in llvm test suite are passing
- X86 32-bit
    - Complete but as far as I know, not actively tested
- Power and AMDgpu
    - Unknown

Contributing to lld

- Try it out and find out what is missing, and what isn't working
    - Report bugs at https://llvm.org/bugs/
    - Developers can be found on the llvm-dev mailing list http://lists.llvm.org/mailman/listinfo
- Patches are of course welcome
    - lld is covered by the http://llvm.org/docs/DeveloperPolicy.html
- Existing material on lld development
    - How to add a new target to LLD http://llvm.org/devmtg/2016-09/
    - New LLD linker for ELF http://llvm.org/devmtg/2016-03/
- Links to content from before 2015 will relate to the atom based lld that ELF and COFF linkers no longer use

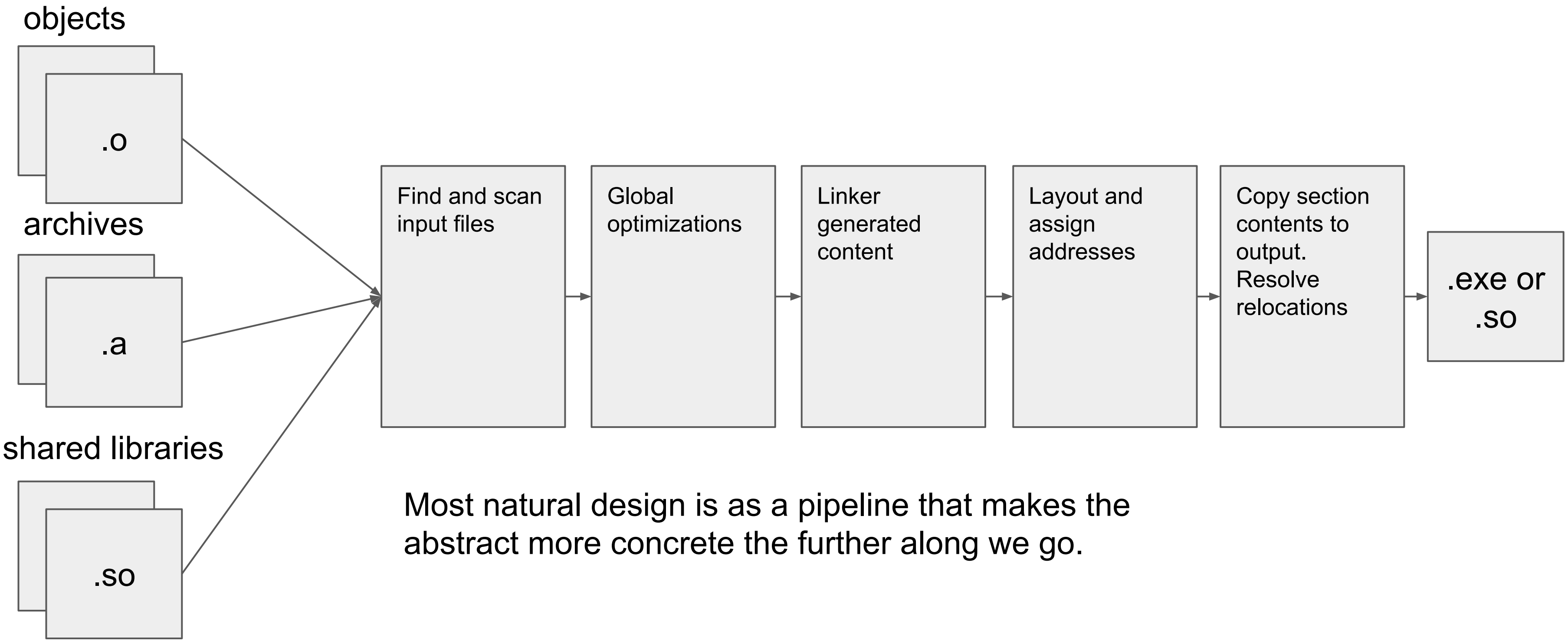
Thanks for listening

Backup

Linker Design Constraints

- All linkers must:
  - Gather the input objects of a program from the command line and libraries
  - Record any shared library dependencies
  - Layout the sections from the input in a well defined order
  - Create data structures such as the PLT and GOT needed by the program
  - Copy the section contents from the input objects to the output
  - Resolve the relocations between the sections
  - Write the output file
- Optionally:
  - Garbage collect unused sections
  - Merge common data and code
  - Call link-time optimizer

Linker design


objects
.o
archives
.a
shared libraries
.so
Find and scan input files
Global optimizations
Linker generated content
Layout and assign addresses
Copy section contents to output. Resolve relocations
.exe or .so

Most natural design is as a pipeline that makes the abstract more concrete the further along we go.

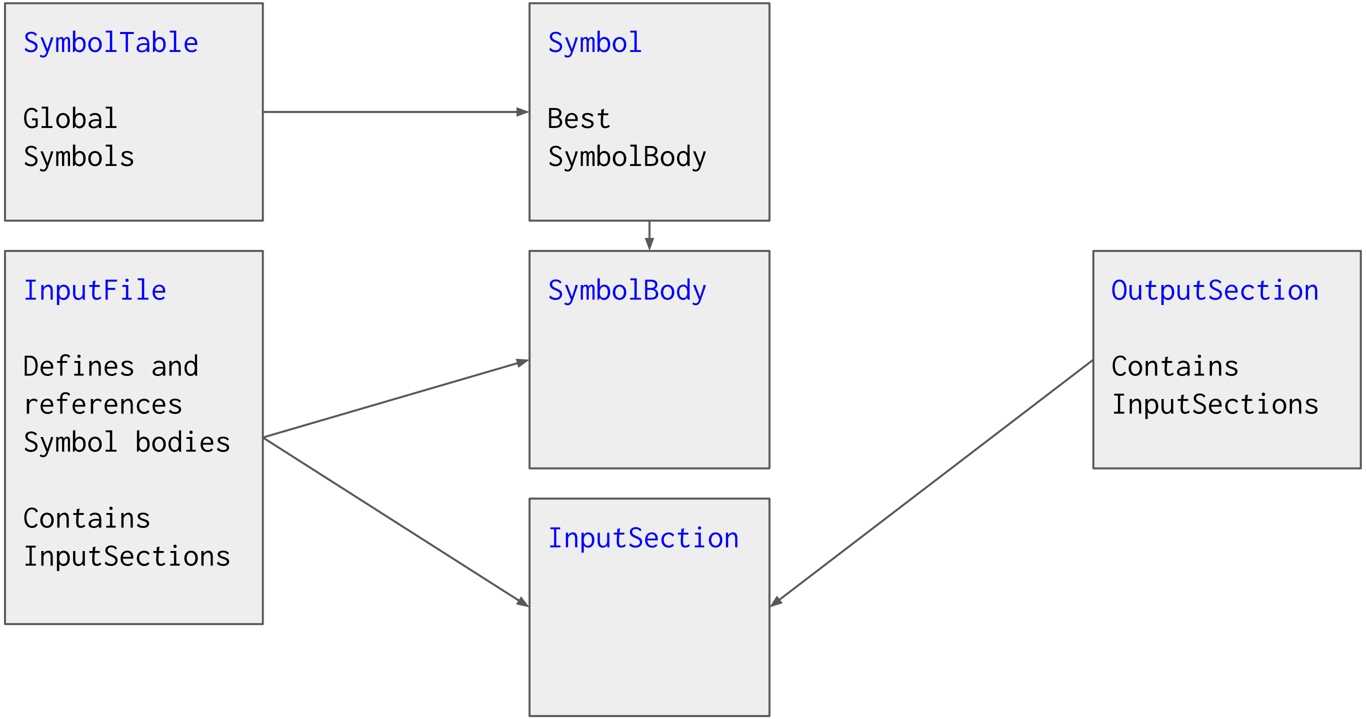
LLD Introduction

- Since May 2015, 3 separate linkers in one project
  - ELF, COFF and the Atom based linker (Mach-O)
  - ELF and COFF have a similar design but don’t share code
  - Primarily designed to be system linkers
    - ELF Linker a drop in replacement for GNU ld
    - COFF linker a drop in replacement for link.exe
  - Atom based linker is a more abstract set of linker tools
    - Only supports Mach-O output
  - Uses llvm object reading libraries and core data structures
- Key design choices
  - Do not abstract file formats (c.f. BFD)
  - Emphasis on performance at the high-level, do minimal amount as late as possible.
  - Have a similar interface to existing system linkers but simplify where possible

LLD Key Data Structures

- InputFile : abstraction for input files
  - Subclasses for specific types such as object, archive
  - Own InputSections and SymbolBodies from InputFile
- InputSection : an ELF section to be aggregated
  - Typically read from objects
- OutputSection : an ELF section in the output file
  - Typically composed from one or more InputSections
- Symbol and SymbolBody
  - One Symbol per unique global symbol name. A container for SymbolBody
  - SymbolBody records details of the symbol
- TargetInfo
  - Customization point for all architectures

LLD Key Data Structure Relationship



SymbolTable

Global Symbols

Symbol

Best SymbolBody

InputFile

Defines and references Symbol bodies

Contains InputSections

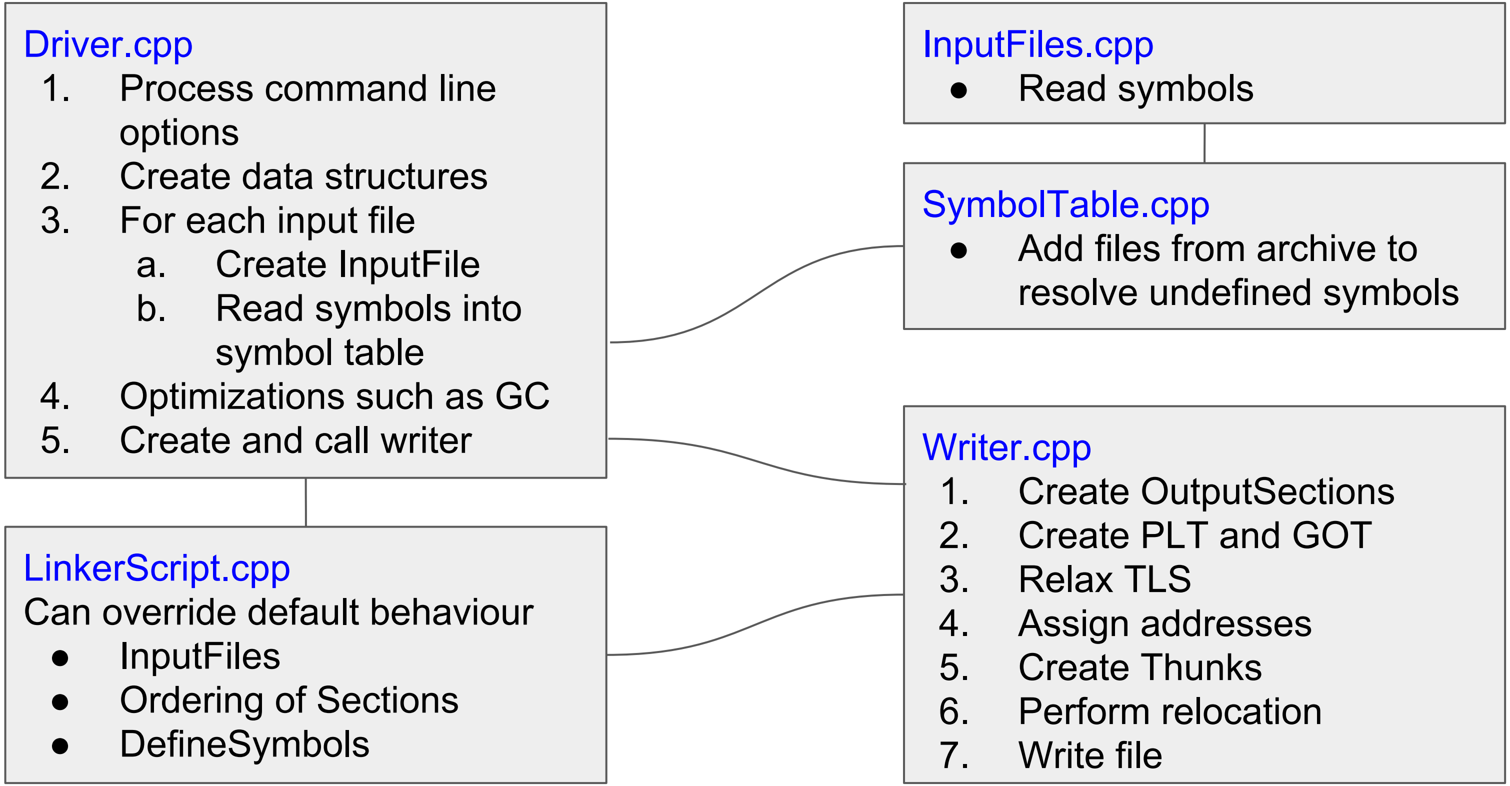
SymbolBody

InputSection

OutputSection

Contains InputSections

LLD ELF Simplified Control Flow



Driver.cpp

1. Process command line options
2. Create data structures
3. For each input file
   a. Create InputFile
   b. Read symbols into symbol table
4. Optimizations such as GC
5. Create and call writer

InputFiles.cpp

- Read symbols

SymbolTable.cpp

- Add files from archive to resolve undefined symbols

Writer.cpp

1. Create OutputSections
2. Create PLT and GOT
3. Relax TLS
4. Assign addresses
5. Create Thunks
6. Perform relocation
7. Write file

LinkerScript.cpp

Can override default behaviour

- InputFiles
- Ordering of Sections
- DefineSymbols