How to add a new target to LLD

Peter Smith, Linaro

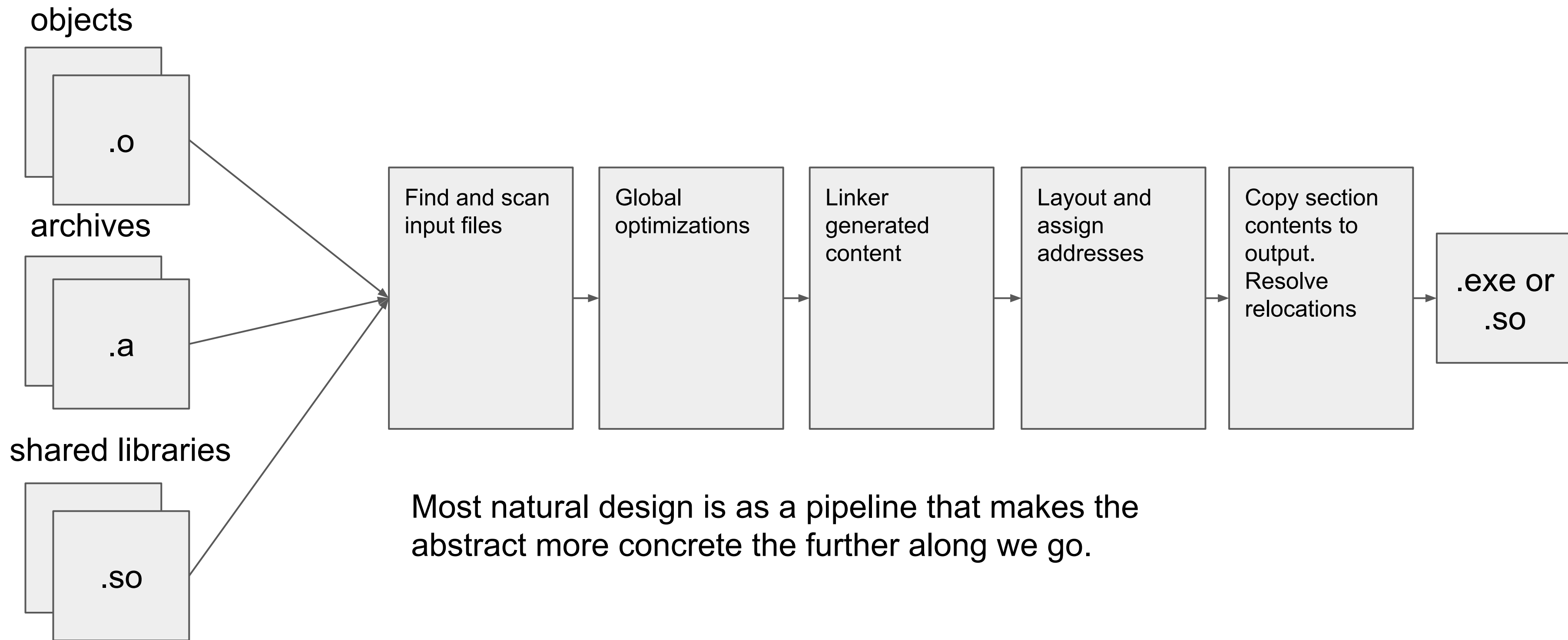
Introduction and assumptions

- What we are covering Today
  - Introduction to the LLD ELF linker and its structure
  - Common porting work for all architectures
  - Some thoughts on adding support for new features not in LLD
- Assumptions of familiarity
  - Object file concepts such as Sections, Symbols and Relocations
  - Static and dynamic libraries
  - SysV style dynamic linking concepts including the PLT and GOT
- About me
  - Currently adding support for ARM to the LLD ELF linker
  - Background in ARM toolchains

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



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

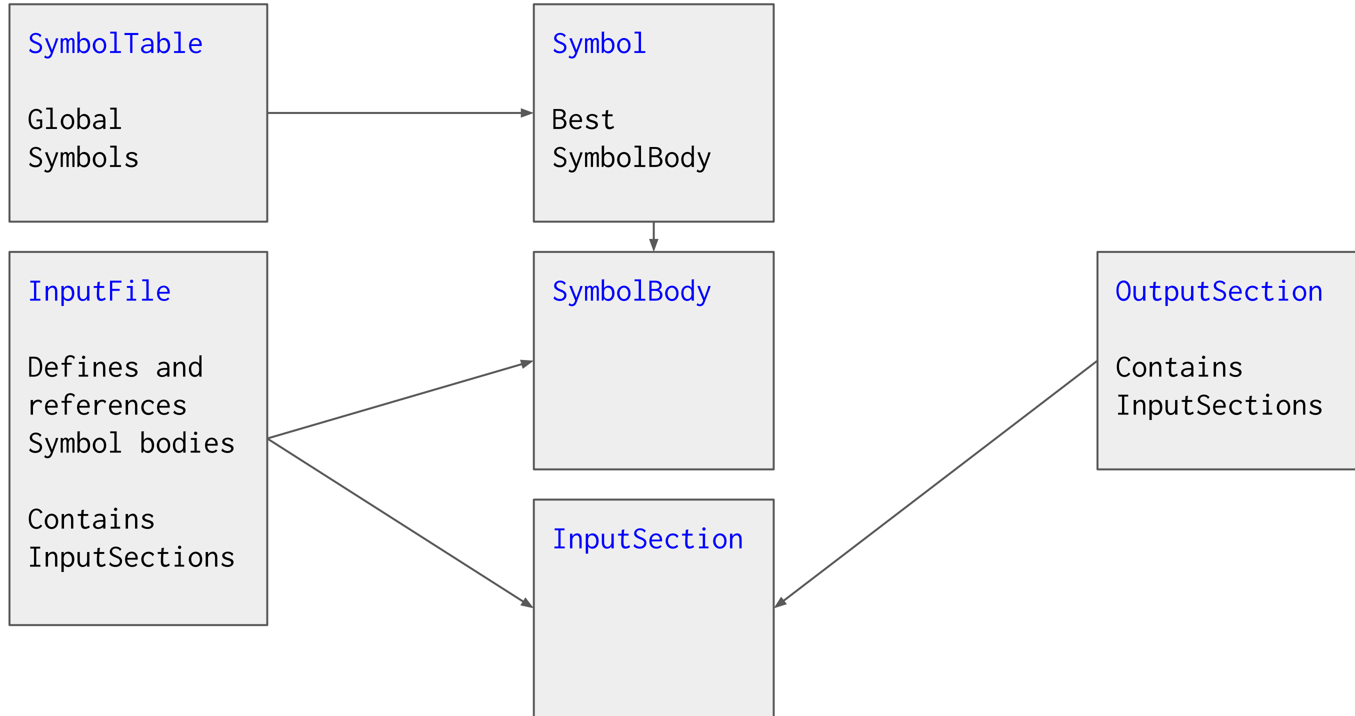
LLD Introduction

- 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

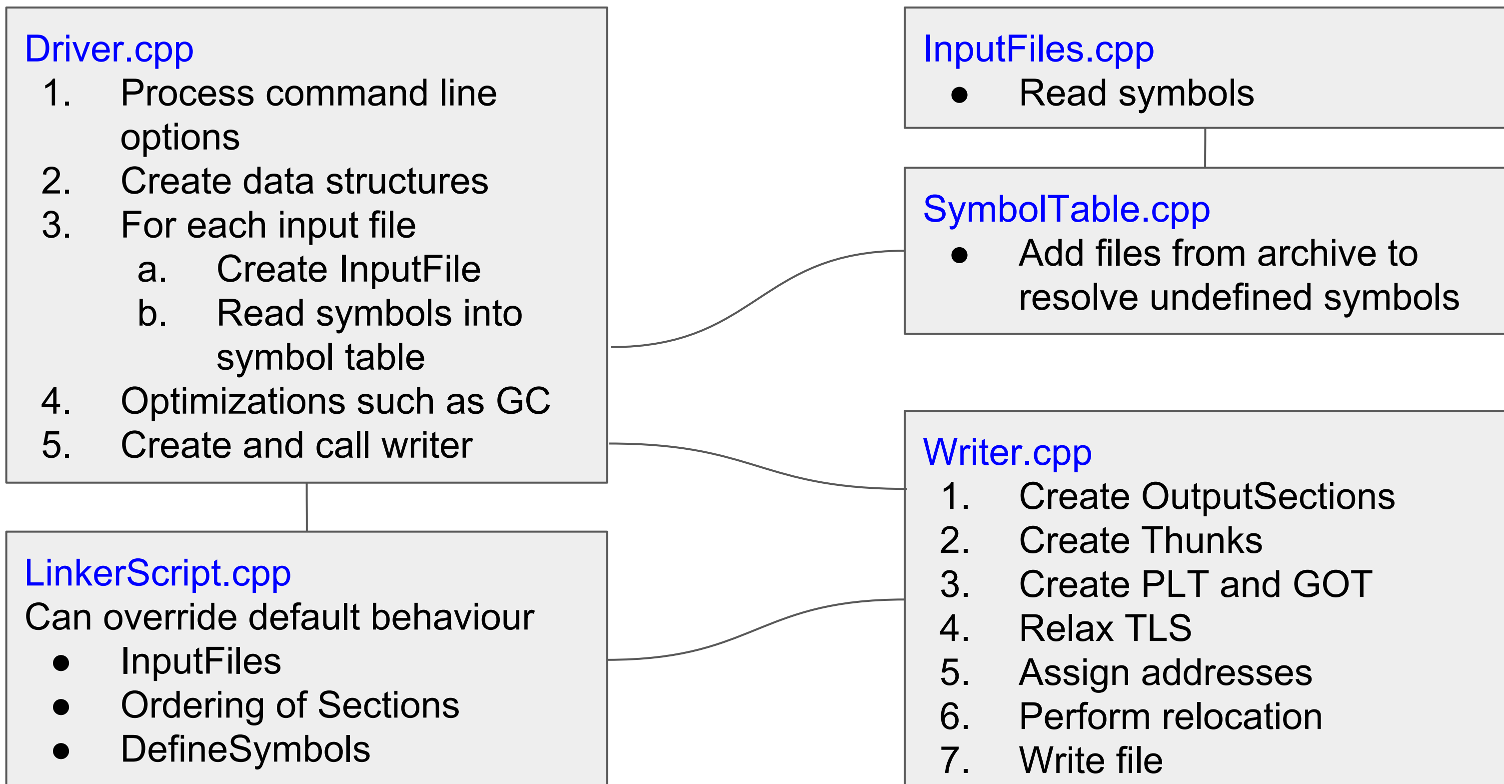
LLD Key Data Structures

- 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



LLD ELF Simplified Control Flow



Adding a new architecture to LLD

- Consult your ABI
  - Parts of the generic ELF specification that are not implemented in LLD
    - LLD only implements what its Targets need
  - All the features in the target specific ELF supplement are candidates
  - Relocation directives
  - Target specific PLT sequences and TLS relaxations
  - Target specific thunks
- Not all ABI features are created equal
  - The pareto principle applies, choose features to implement wisely
    - Most programs can be linked with only a small number of implemented features
    - A long tail of programs that (ab)use a specific feature
  - Getting hello world to run is a good first step

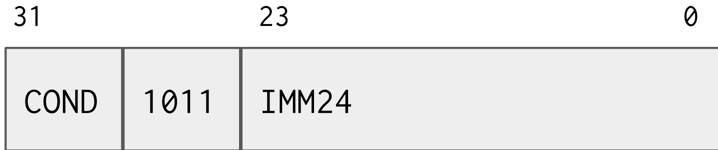
Porting common to all architectures

- Add a subclass to TargetInfo for your machine type
    - Creating an instance of this class in response to handle the machine type
- Add enough relocations to link your initial application
    - Hello world usually only needs a small number
- Identify your common dynamic SysV relocations identified by TargetInfo
    - R_386_COPY, R_ARM_COPY, R_MIPS_COPY ...
- Add PLT sequences early
    - Dynamically linking against the C-library uses fewer linker features than the static C-library
- Other TargetInfo subclasses are useful guides

Implementing Relocations

- Relocations in ELF are described by:
  - Type : Identification of relocation
  - Place **P** : where the relocation is applied
  - Symbol **S** : the destination of the relocation
  - Addend **A** : constant encoded in the place for REL or in the relocation for RELA
- Type tells the linker what to do with P, S and A
- Relocations in TargetInfo are handled by up to 3 member functions
  - getRelExpr() : Map Type to a RelExpr
    - LLD uses RelExpr to abstract relocation processing across architectures
    - Example: R_PLT_PC = PLT(**S**) + **A** - **P**
  - getImplicitAddend() : for REL how to extract **A** from **P**. Not needed for RELA.
  - relocateOne() : how to encode result of relocation to **P**

Relocation example ARM BL



- Rel relocation with type R_ARM_CALL
- Can be indirected via a PLT
- PC-Relative, calculation is **S** + **A** - **P**
- Addend **A** is bottom 24-bits of instruction, with result shifted left by 2 to form signed 26-bit offset
  - For ARM a relocated call always has **A** as -8 to account for PC-bias

1. getImplicitAddend(), extracts A from IMM24
   a. `SignExtend64<26>(read32le(Buf) << 2);`
2. getRelExpr() returns R_PLT_PC for R_ARM_CALL
   a. LLD converts to R_PC if no PLT entry needed
3. relocateOne() checks overflow and writes back to IMM24
   a. `checkInt<26>(Val, Type);`
   b. `write32le(Loc, (read32le(Loc) & ~0x00ffffff) | ((Val >> 2) & 0x00ffffff));`

PLT Sequences

- Two member functions must be implemented
  - writePltHeader() : PLT[0] for the lazy binding call to the dynamic loader
  - writePLT() : PLT[N] for standard entries
- Consult your ABI and dynamic loader for the calling conventions required. For example in ARM:
  - PLT[N] must set the **IP** register to the contents of .got.plt(N)
  - PLT[0] can’t use normally corruptible **IP** register for address of dynamic loader entry point
  - Convention that PLT[0] stacks and uses **LR** for address of dynamic loader entry point
    - Dynamic loader restores **LR** from stack

Thread local storage

- LLD has support for the standard and descriptor based TLS dialect
- Common code to identify and create dynamic relocations
- Identify dynamic relocations in TargetInfo
  - TlsModuleIndexRel (Global Dynamic, and Local Dynamic)
  - TlsOffsetRel (Global Dynamic and Local Dynamic)
  - TlsGotRel (Initial Exec)
  - TlsDescRel (Descriptor dialect)
- TcbSize selects between variant 1 and variant 2 (TcbSize == 0)
- Implement static TLS relocations
- Implement or disable TLS relaxations

The non-standard parts

- Many architectures have custom requirements. For example in ARM:
    - There are two states ARM and Thumb that the linker is responsible for interworking
        - Choice of BL or BLX made at link time depending on target state
        - Interworking thunks required for B instructions
        - Interworking thunk to PLT entries needed
    - ARM uses Itanium style exception tables with ordering dependency requirements
    - ARM TLS relocations can’t be relaxed
    - Linker responsible for range extension thunks
    - Mapping symbols needed for correct disassembly

Non standard parts continued

- Beware of phase order problems
  - Need to wait for information to become available but your phase alters information used by some previous phrase
- Do you really need the full extension right now?
  - Can you implement a simpler subset in a way that is less disruptive to the implementation
- If the new phase could affect performance, but only for your target, make it target specific.
- Don’t expect reviewers to be familiar with non-standard extensions
  - Provide links to documentation
  - Reference implementations in other linkers
  - Test cases to show how features are used in practice

Summary

- The COFF and ELF LLD implementations are intended to be a drop in replacement for link.exe and ld respectively
    - Some architectures closer to achieving this than others
- Porting a new architecture that closely resembles an existing one is straightforward and doesn’t take much code
- Expect to take much longer for architectures with many non standard features

References

- LLD homepage
- Generic ELF Specification
- ELF for the ARM Architecture
- ELF handling for Thread Local Storage

The End

Backup

ELF Recap

```c
#include <stdio.h>

static int x = 10;
int y;

int function2(void)
{
    return x + y;
}

static void function1(void)
{
    rw += 1;
    printf("%d\n",
           function2());
}
```

Sections

```
.text
Type: SHT_PROGBITS
Flags: SHF_ALLOC, SHF_EXECINSTR
```

```
rel.text
Type: SHT_REL
```

```
.rodata.str1.1
Type: SHT_PROGBITS
Flags: SHF_ALLOC, SHF_MERGE,
SHF_STRINGS
```

```
.data
Type: SHT_PROGBITS
Flags: SHF_ALLOC, SHF_WRITE
```

```
.bss
Type: SHT_NOBITS
Flags: SHF_ALLOC, SHF_WRITE
```

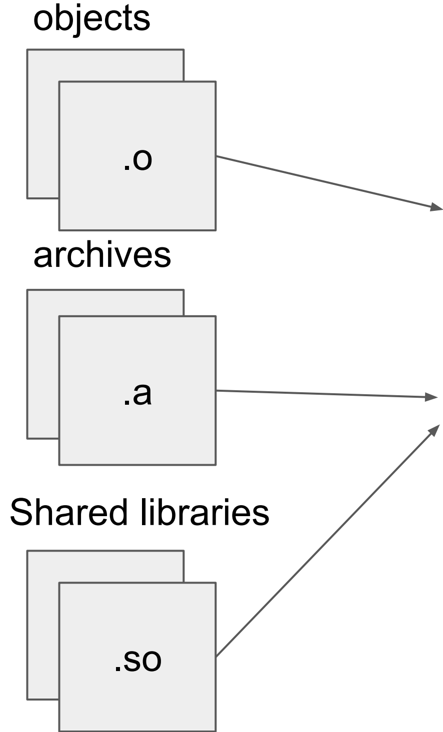
Symbols

```
x, STT_OBJ, STB_LOCAL, .data
y, STT_OBJ, STB_GLOBAL, .bss
function2, STT_FUNC, STB_GLOBAL, .text
function1, STT_FUNC, STB_LOCAL, .text
printf, STT_FUNC, 0 (undefined reference)
```

Relocations

```
R_ARM_MOVW_ABS_NC rw
R_ARM_MOVT_ABS    rw
R_ARM_MOVW_ABS_NC zi
R_ARM_MOVT_ABS    zi
R_ARM_MOVW_ABS_NC .L.str
R_ARM_MOVT_ABS    .L.str
R_ARM_CALL        function2
R_ARM_CALL        printf
```

Introduction to Linking: loading content



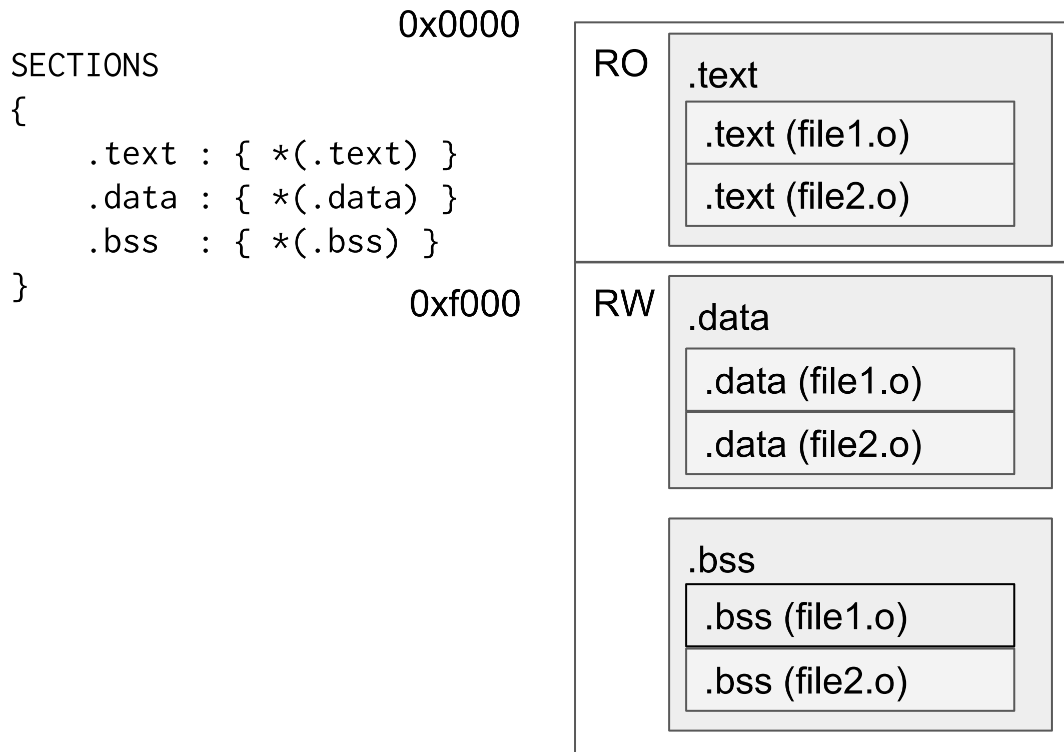
Load Content

- Load objects on command line
    - Match symbol references with definitions
    - Maintain list of unresolved references
- Iterate until fixed point
    - Load symbol definitions to resolve references
    - Add unresolved references

Result

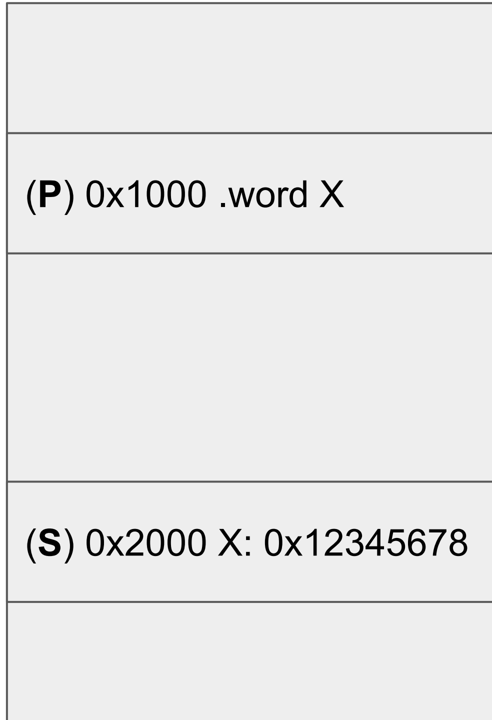
- Global symbols defined
- Input objects recorded
    - Sections
        - Relocations
    - Local Symbols
- Shared library dependencies

Introduction to linking: Layout and address



- Sections from objects InputSections are assigned to OutputSections
- Can be controlled by script or by defaults
- OutputSections assigned an address
- InputSections assigned offsets within OutputSections
- Similar OutputSections are described by segments

Introduction to linking: Relocation

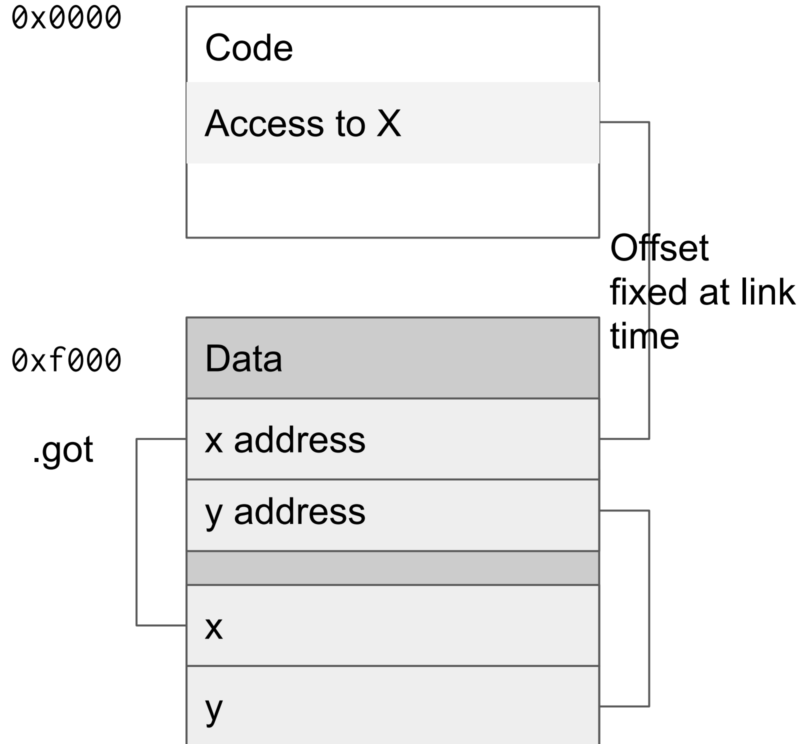


R_ARM_ABS32 (**S**+**A**)

Once final addresses of all sections are known then relocations are fixed up. In general for a relocation at address **P**

- Extract addend **A** from relocation record (RELA) or from location (REL)
- Find destination symbol address **S**
- Perform calculation
    - **S** + **A** for absolute
    - **S** + **A** - **P** for relative
- Write result to **P**

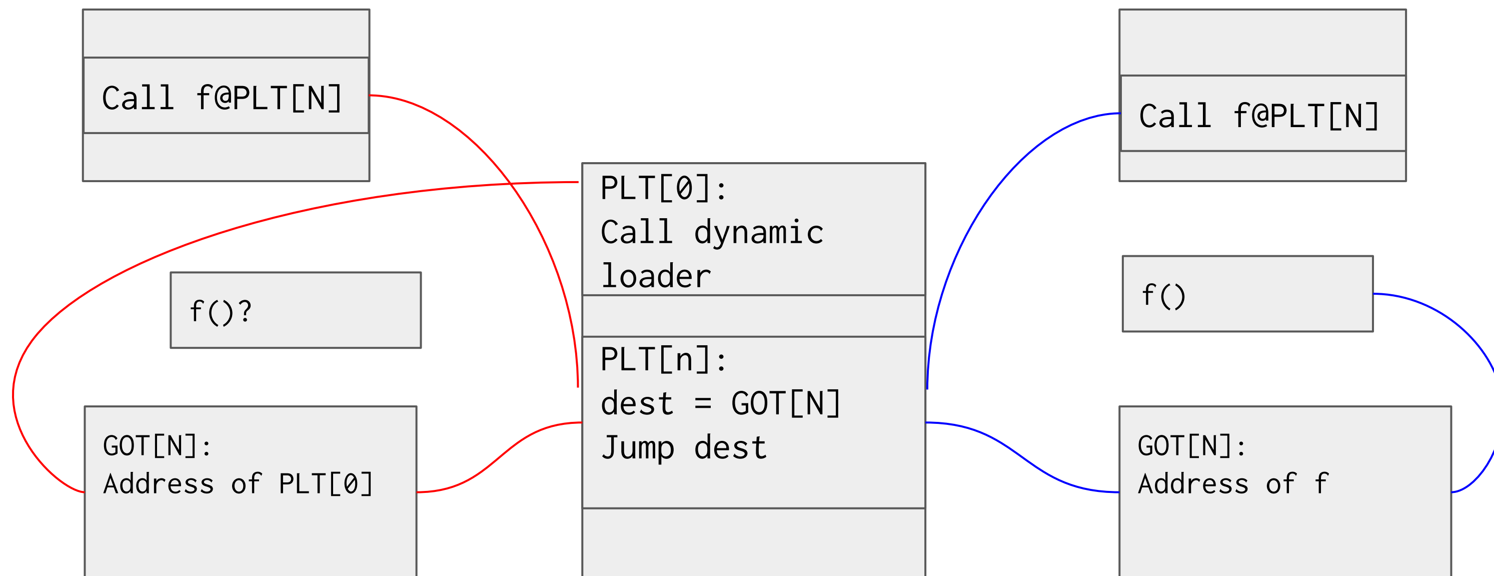
Position independent code via GOT



Global **O**ffset **T**able (GOT) is constructed by the linker in response to specific relocations

- Offset from code to data is known
- Code loads address of variable from GOT
- GOT filled in/relocated by dynamic linker

Calling a function via PLT



Lazy binding, 1st call

Subsequent calls