Operating System Capstone

Lecture 3: Compiler, Assembler, Linker, Loader

Tsung Tai Yeh

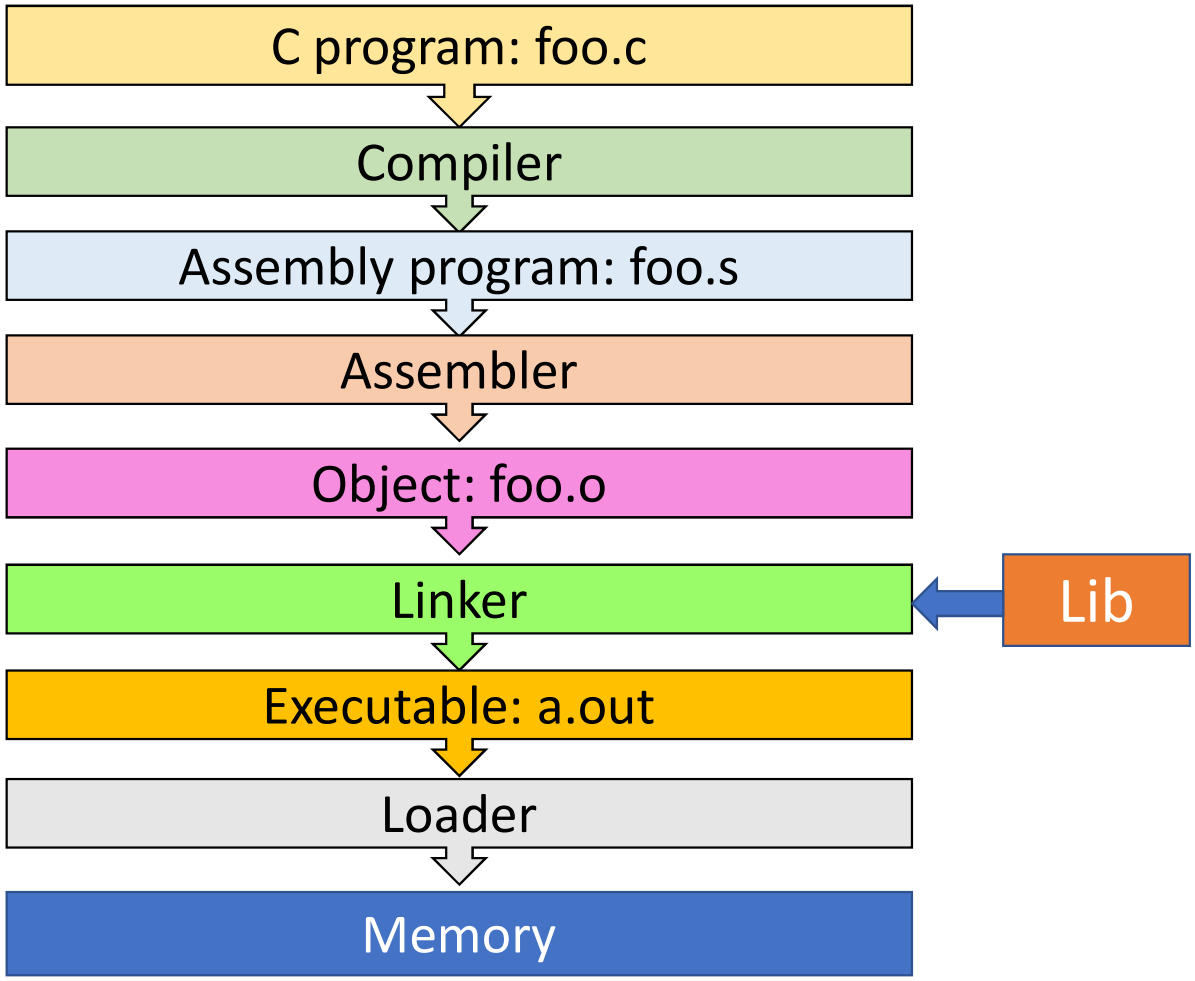
Tuesday: 3:30 – 5:20 pm
Classroom: ED-302

Acknowledgements and Disclaimer

- Slides was developed in the reference with
  MIT 6.828 Operating system engineering class, 2018
  MIT 6.004 Operating system, 2018
  UC Berkeley, CS 61 C, Great ideas in computer architecture (Machine structures), 2020
  Remzi H. Arpaci-Dusseau etl. , Operating systems: Three easy pieces. WISC
  Christopher Hallinan, Embedded Linux Primer, A Practical Real-World Approach, Prentice Hall, 2010

Outline

- Translation
- Compiler
- Assembler
- Linker
- Loader
- Example



Translation vs. interpretation

- How do we run a program written in a source language ?
  - Interpreter: directly executes a program in the source language
  - Translator: converts a program from the source language to an equivalent program in another language
- Directly interpret a high level language when efficiency is not critical
- Translate to a lower level language when increased performance is desired

Translation vs. interpretation

- Generally easier to write an interpreter
- Interpreter closer to high-level, so can give better error messages (e.g. Python)
- Interpreter is slower (~10x), but code is smaller (~2x)
- Interpreter provides instruction set independence: can run on any machines

Translation vs. interpretation

- Translated/compiled code almost always more efficient and therefore higher performance
  - Important for many applications, particularly operating systems
- Translation/compilation helps “hide” the program “source” from the users
  - One model for creating value in the marketplace (e.g. Microsoft keeps all their source code secret)
  - Alternative model, “open source”, creates value by publishing the source code and fostering a community of developers

C Translation

- **Recall:**
  - A key feature of C is that it allows you to compile files separately, later combining them into a single executable
- What can be accessed across files ?
  - Functions
  - Global variables

Compiler

- **Input:**
  - High-level language (HLL) code (e.g. C, Java in files such as foo.c)
- **Output:**
  - Assembly language code (e.g. foo.s for RISC-V)
- Note that the output may contain pseudo-instructions
- In reality, there's a preprocessor step before this to handle #directives

Assembler

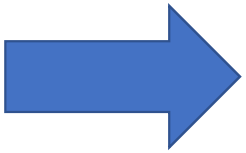
- **Input**
  - Assembly language code (e.g. foo.s for RISC-V)
- **Output**
  - Object code (True assembly), information tables (e.g. foo.o for RISC-V)
  - **Object file**
- Reads and uses directives
- Replaces pseudo-instructions
- Produces machine language

Assembler directives

- Give directions to assembler, but do not produce machine instructions
  - .text: Subsequent items put in user **text segment**
    - It is where the instruction codes are declared within the executable program
  - .data: Subsequent items put in user **data segment**
    - Declares data elements that are declared with an initial value
  - .globl sym: declares sym global and can be referenced from other files
  - .asciiz str: create an ASCII string in memory determinated by the null character (‘\0’)
  - .word $w_1 \ldots w_n$: store the n 32-bit quantities in successive memory words

Pseudo-instruction replacement

Pseudo	
mv	t0, t1
neg	t0, t1
li	t0, imm
not	t0, t1
beqz	t0, loop
la	t0, str



Real	
addi	t0, t1, 0
sub	t0, zero, t1
addi	t0, zero, imm
xori	t0, t1, -1
beq	t0, zero, loop
lui	t0, str[31:12]
addi	t0, t0, str[11:0]

Producing machine language

- Simple cases
  - Arithmetic and logical instructions, shifts, etc.
  - All necessary info contained in the instruction
- What about branches and jumps ?
  - Branches and jumps require a relative address
  - Once pseudo-instructions are replaced by real ones, we know by how many instructions to branch, so no problem

Producing machine language

- "Forward Reference" problem
  - Branch instructions can refer to labels that are "forward" in the program

```
        or    s0,  x0,  x0
L1:     slt   t0,  x0,  a1
        beq   t0,  x0,  L2
        addi  a1,  a1,  -1
        j     L1
L2:     add   t1,  a0,  a1
```

  - Solution: Make two passes over the program

Two passes overview

- **Pass 1**
  - Expands pseudo instructions encountered
  - Remember position of labels
  - Take out comments, empty lines, etc.
  - Error checking
- **Pass 2**
  - Use label positions to generate relative addresses (for branches and jumps)
  - Outputs the object file, a collection of instructions in binary code

Producing machine language

- What about jumps to external labels ?
  - Requiring knowing a final address
  - Forward or not, can't generate machine instruction without knowing the position of instructions in memory
- What about reference to data ?
  - la gets broken up into lui and addi
  - These will require the full 32-bit address of the data
  - These can't be determined yet, so we create two tables

Symbol table

- List of “items” that may be used by other files
  - Each file has its own symbol table
- What are they ?
  - Labels: function calling
  - Data: anything in the .data section
  - Variables may be accessed across files
- Keeping track of the labels fixes the forward reference problem

Symbol table example

Symbol	Type
bar	U
dec	U
main	T
“Here”	D
num	D
printf	U
“%\n”	D

U: undefined indicates the external file reference
T: .Text section
D: .Data section

```
1  #include <stdio.h>
2  extern int bar();
3  extern int dec;
4  int main() {
5      char *output = “Here”;
6      static int num = 7;
7      int i = 5;
8      while (i > 0) {
9          i --;
10         int temp = bar(num);
11         printf(“%d\n”, temp);
12     }
13 }
```

Relocation table

- List of “items” this file will need the address of later (currently underdetermined)
  - E.g. functions not defined in this file’s text segment
- What are they ?
  - Any external label jumped to: jal or jalr
    - Internal
    - External (including library files)
  - Any piece of data
    - Such as anything referenced in the data section

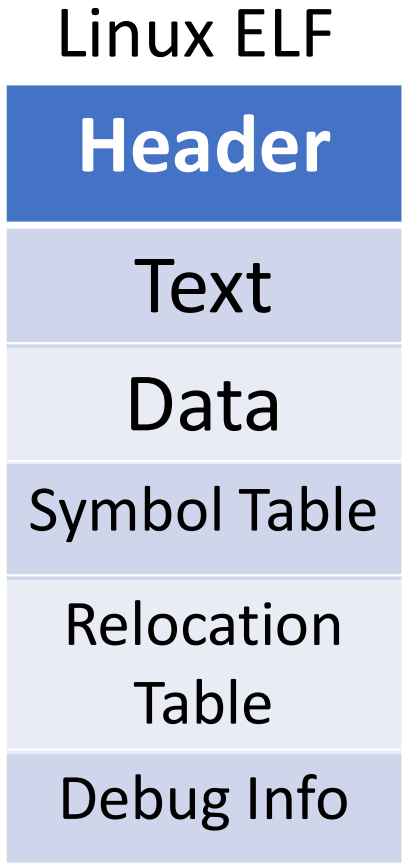
Relocation table example

Line	Instr.	Dependency
5	MV	“Here”
10	LD	Num
10	BR	bar
11	MV	“%d”
11	BR	printf

```
1  #include <stdio.h>
2  extern int bar();
3  extern int dec;
4  int main() {
5      char *output = “Here”;
6      static int num = 7;
7      int i = 5;
8      while (i > 0) {
9          i --;
10         int temp = bar(num);
11         printf(“%d\n”, temp);
12     }
13 }
```

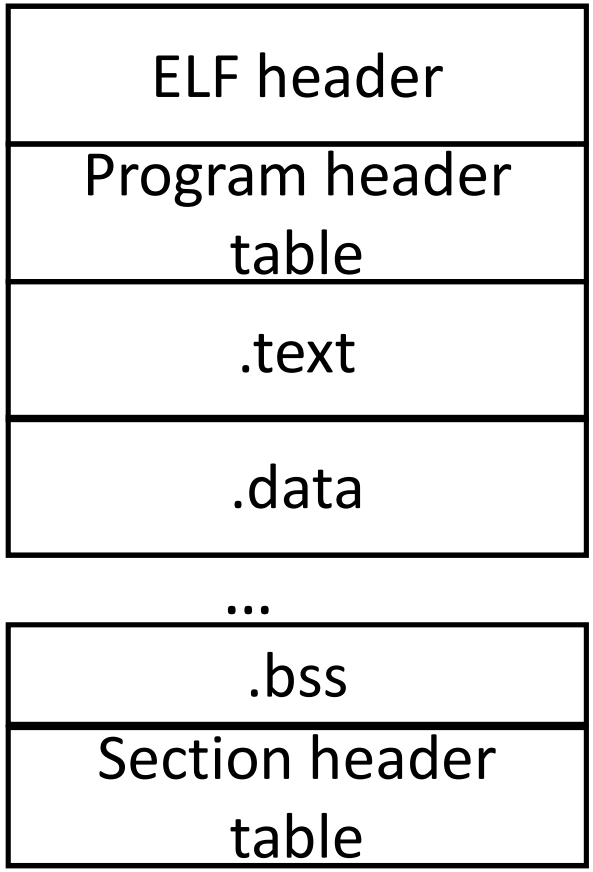
Object file format (ELF)

- Object file header:
  - size and position of the other pieces of the object file
- Text segment
  - The machine code
- Data segment
  - Data in the source file (binary)
- Relocation table
  - Identifies lines of code that need to be “handled”
- Symbol table
  - List of this file’s labels and data that can be referenced
- Debugging information



Executable and Linkable Format (ELF)

- The ELF object file contains the following information
  - ELF header
  - Program header
    - Linker reads it on program loading to search for necessary sections
  - Section header
    - For program analysis through “readelf”
  - The section contains segments
    - **.text (code) segment**: store CPU instructions
    - **.data segment:**
      - Initialized variables: global/static variables
      - Read-only area (constant variable), read-write area
    - **.bss:** store uninitialized data



Case study: memory segment

- Demonstrate memory segment

```
root@amcc:~# ./hello
Hello, World! Main is executing at 0x100000418
This address (0x7ff8ebb0) is in our stack frame
This address (0x10010a1c) is in our bss section
This address (0x10010a18) is in our data section
```

- Assume our computer has 256 MB RAM
  - The process called hello is executing somewhere in high RAM just above the 256 MB boundary
  - The stack address is roughly halfway into a 32-bit address space
  - The virtual address were assigned by the kernel and are backed by physical RAM somewhere within 256 MB range of available memory

ELF header

1. All ELF object files start with ELF header
2. e_shoff: the location of the section header
3. e_shnum is the number of section header entries

ELF object file data structure

```
# define EI_NIDENT 16

typedef struct {
unsigned char
e_ident[EI_NIDENT];
Elf64_Half e_type;
Elf64_Half e_machine;
Elf64_Word e_version;
Elf64_Addr e_entry;
Elf64_Off e_phoff;
Elf64_Off e_shoff;
Elf64_Word e_flags;
Elf64_Half e_ehsize;
Elf64_Half e_phentsize;
Elf64_Half e_phnum;
Elf64_Half e_shentsize;
Elf64_Half e_shnum;
Elf64_Half e_shstrndx;
} Elf64_Ehdr;
```

readelf –h study

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x4004b0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          16112 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         9
  Size of section headers:           64 (bytes)
  Number of section headers:         30
  Section header string table index: 29
```

ELF sections

Using “objdump –d study” to analyze each section of the ELF file

- ELF sections contain:
  - **Binary executable code**
    - All binary executable code goes into the **.text** section
  - **Code for initialized variables**
    - Are grouped in the **.data** section
  - **Code for uninitialized variables**
    - Are grouped in the **.bss** section
  - **Constant strings**
    - Are grouped in **.shstrtab** section
  - Information about **the variable and function names** used in the program
    - Go in **.symtab** (symbol table)
  - **Debug** information
    - In **.debug** section

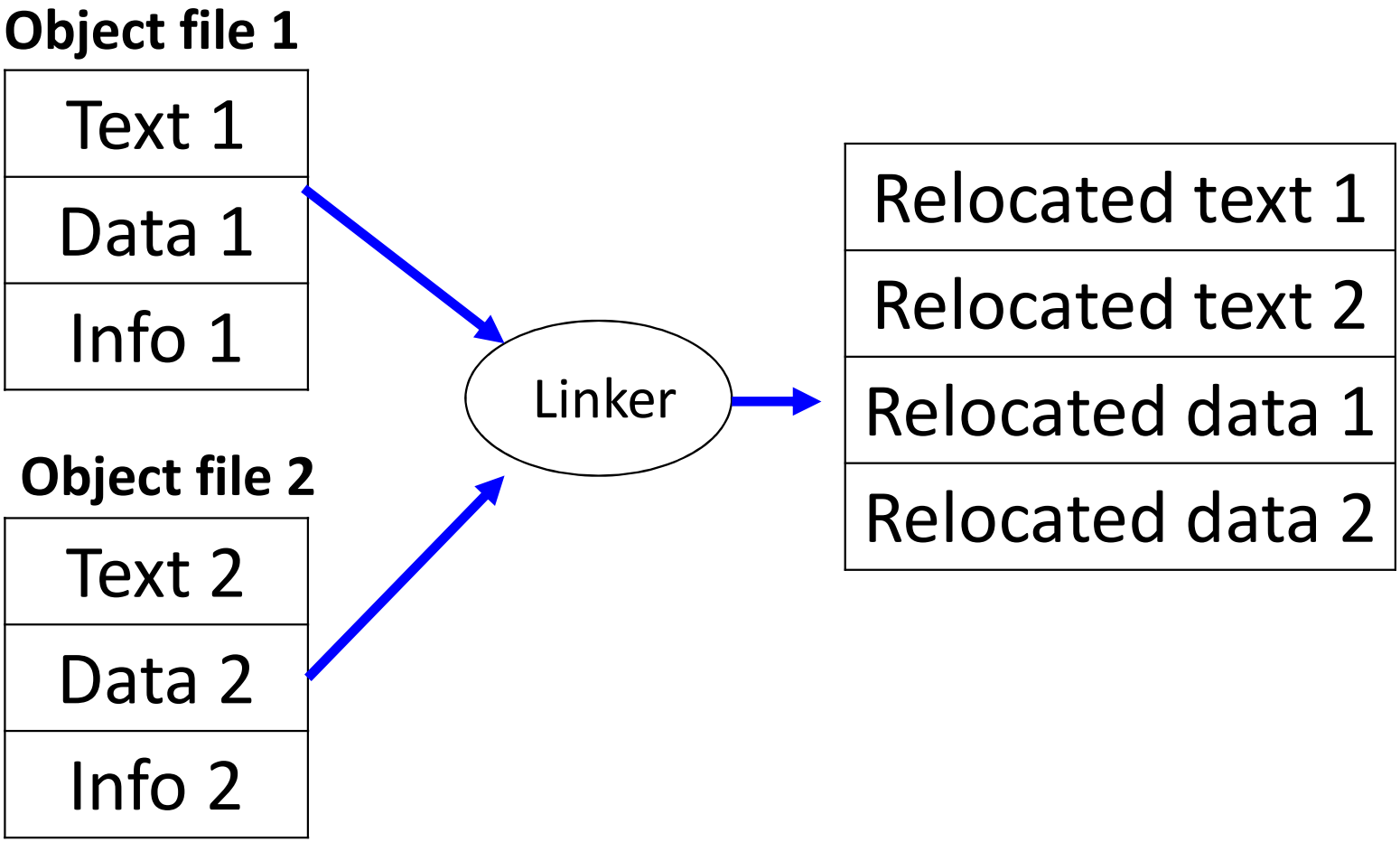
ELF program header

- The ELF program header indicates which ELF section goes to a particular memory location
  - e_phoff provides the offset at which the ELF program header is presented in the ELF file
  - e_phnum is the number of program header entries
  - The size of each entry is e_phentsize
  - “readelf –l study” to see the program header information of study.c

Linker

- **Input**
  - Object code files, information tables (e.g. foo.o, lib.o for RISC-V)
- **Output**
  - Executable code (e.g. a.out for RISC-V)
- Combines several object (.o) files into a single executable ("linking")
- Enables separate compilation of files
  - Changes to one file do not require recompilation of whole program

Linker


Object file 1
Text 1
Data 1
Info 1
Object file 2
Text 2
Data 2
Info 2
Linker
Relocated text 1
Relocated text 2
Relocated data 1
Relocated data 2

Linker

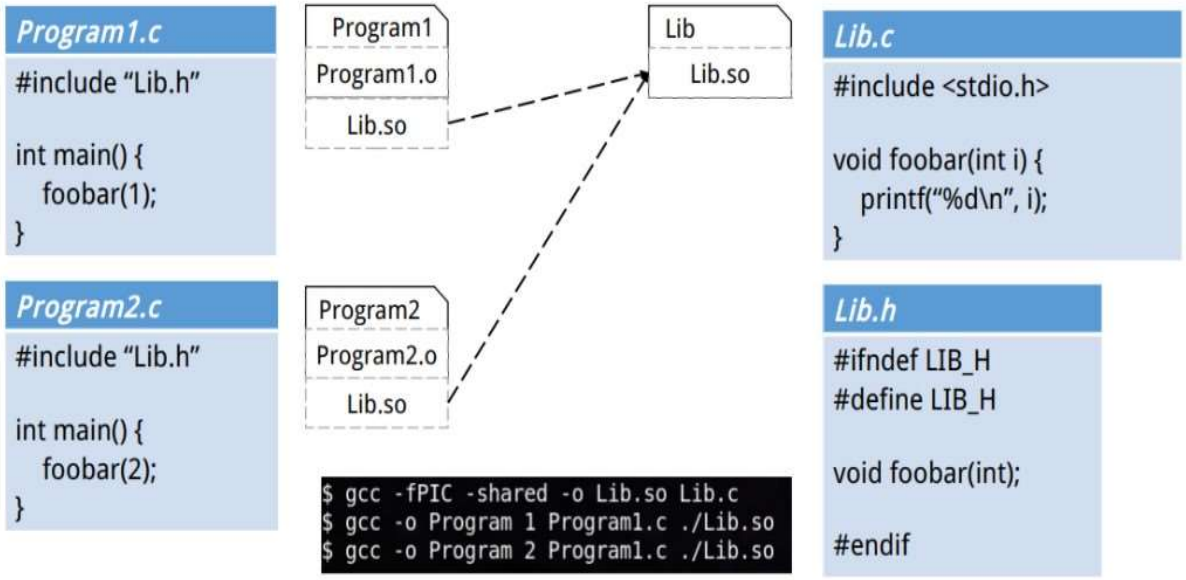
- Take text segment from each .o file and put them together
- Take data segment from each .o file, put them together, and concatenate this onto end of text segments
- Resolve references
  - Go through relocation table; handle each entry
  - i.e. fill in all absolute addresses

Static linking

- **Static linking**
  - Copy all the libraries required for the program into the final executable file
  - The linker combines the relevant libraries with the program code to resolve external references **(.a file)**
  - Large executable file because it is connected with other files
- **Archive a static library**
  - gcc hello.c –c; ar crsv liboperator.a hello.o
  - gcc –static hello.c -loperator
- **Example**
  - gcc main /usr/local/foo/lib/liboperator.a –o main
  - gcc main.c –L /usr/local/foo/lib –loperator –o main

Dynamic linking

- **Dynamic linking**
  - The linking occurs at run time when the executable file and libraries are loaded to the memory
  - Dynamic shared objects (.so file)
  - There is only one copy of a shared library is in the memory
  - Small executable file against the one through the static linking



https://hackmd.io/@ofAlpaca/r1L5Ecc_7?type=view

Three Types of Addresses

```
int global;
int func(void) { return global; }
```

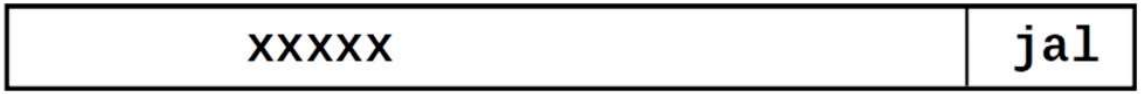
1. Assume absolute address of global: 0x20000004
2. The PC of the first instruction in func is 0x10000008

```
func:
    auipc a0, 0x10000
    lw    a0, 0x004(a0)
```

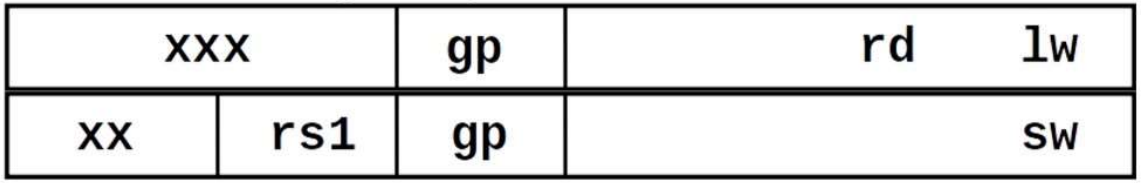
- **PC-Relative addressing** (beq, bne, jal)
  - Never relocate
  - External function reference (usually jal)
    - Always relocate
  - Static data reference (often auipc and addi)
    - Always relocate
    - RISC-V often uses auipc rather than lui so that a big block of stuff can be further relocated as long as it is fixed relative to the pc
    - auipc (Add Upper Immediate to Program Counter) instruction
    - auipc is setting a0 to 0x20000004: a0 = PC + (imm20 << 12), PC = 0x10000004 and imm20 is 0x10000

Absolute addresses in RISC-V

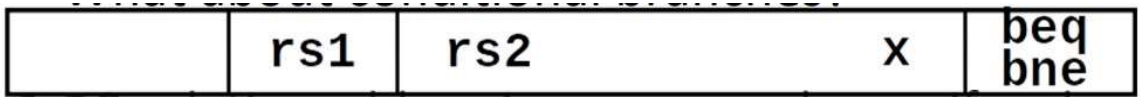
- Which instructions need relocation editing ?
  - J-format: jump/jump and link



  - Loads and stores to variables in static area, relative to global pointer



  - PC-relative addressing preserved even if code moves



Resolving references

- Linker assumes the first word of the first text segment is at 0x10000 for RV32
- Linker knows
  - Length of each text and data segment
  - Ordering of text and data segments
- Linker calculates
  - Absolute address of each label to be jumped to (internal or external) and each piece of data being referenced

Resolving references

- To resolve references
  - Search for reference (data or label) in all “user” symbol tables
  - If not found, search library files (e.g. printf)
  - Once absolute address is determined, fill in the machine appropriately
- Output of linker
  - Executable file containing text and data (plus header)

Loader

- **Input**
  - Executable code (e.g. a.out for RISC-V)
- **Output**
  - <program is run>
- Executable files are stored on disk
- When one is run, loader's job is load it into memory and start it running
- In reality, loader is the operating system (OS)
  - Loading is one of the OS tasks

Loader

- Reads executable file's header
  - determine size of text and data segments
- Creates new address space for program large enough
  - hold text and data segments, along with a stack segment
- Copies instructions and data from executable file into the new address space

Loader

- Copies arguments passed to the program onto the stack
- Initializes machine registers
  - Most registers cleared, but stack point assigned address of 1st free stack location
- Jumps to start-up routine that copies program's arguments from stack to registers and set the PC
  - If main routine returns, start-up routine terminates program with the exit system call

C.A.L.L. Example

```
#include <stdio.h>
int main()
{
   printf("Hello, %s\n", "world");
   return 0;
}
```

Compiled Hello.c:Hello.s

```
.text
  .align 2
  .globl main
main:
  addi sp,sp,-16
  sw   ra,12(sp)
  lui  a0,%hi(string1)
  addi a0,a0,%lo(string1)
  lui  a1,%hi(string2)
  addi a1,a1,%lo(string2)
  call printf
  lw   ra,12(sp)
  addi sp,sp,16
  li   a0,0
  ret
  .section .rodata
  .balign 4
string1:
  .string "Hello, %s!\n"
string2:
  .string "world"
```

```
# Directive: enter text section
# Directive: align code to 2^2 bytes
# Directive: declare global symbol main
# label for start of main
# allocate stack frame
# save return address
# compute address of
#   string1
# compute address of
#   string2
# call function printf
# restore return address
# deallocate stack frame
# load return value 0
# return
# Directive: enter read-only data
section
# Directive: align data section to 4
bytes
# label for first string
# Directive: null-terminated string
# label for second string
# Directive: null-terminated string
```

Assembled Hellos: Linkable Hello.o

```
00000000 <main>:
0:  ff010113 addi sp,sp,-16
4:  00112623 sw ra,12(sp)
8:  00000537 lui a0,0x0 # addr placeholder
c:  00050513 addi a0,a0,0   # addr placeholder
10: 000005b7 lui a1,0x0 # addr placeholder
14: 00058593 addi a1,a1,0 # addr placeholder
18: 00000097 auipc ra,0x0   # addr placeholder
1c: 000080e7 jalr ra    # addr placeholder
20: 00c12083 lw ra,12(sp)
24: 01010113 addi sp,sp,16
28: 00000513 addi a0,a0,0
2c: 00008067 jalr ra
```

Linked Hello.o: a.out

```
000101b0 <main>:
   101b0: ff010113 addi sp,sp,-16
   101b4: 00112623 sw   ra,12(sp)
   101b8: 00021537 lui  a0,0x21
   101bc: a1050513 addi a0,a0,-1520 # 20a10
<string1>
   101c0: 000215b7 lui  a1,0x21
   101c4: a1c58593 addi a1,a1,-1508 # 20a1c
<string2>
   101c8: 288000ef jal  ra,10450    # <printf>
   101cc: 00c12083 lw   ra,12(sp)
   101d0: 01010113 addi sp,sp,16
   101d4: 00000513 addi a0,0,0
   101d8: 00008067 jalr  ra
```

Summary

- **Compiler**
  - converts a single HLL file into a single assembly file (.c -> .s)
- **Assembler**
  - Remove pseudo-instructions, converts assembly codes to machine language, and create a checklist for linker (relocation table) (.s -> .o)
- **Linker**
  - Combines several object files and resolves absolute addresses (.o -> .out)
  - Enable separate compilation and use of libraries
- **Loader**
  - Loads executable into memory and begins execution