Course Info

- Lab 4 will be released after class (10 a.m.), get yourself prepared before going to lab sessions!
- Project 1.1 available, and will be marked in lab sessions. Deadline March 13th.
- HW3 this week, keep an eye on piazza.
- Discussion next week on CALL

信息科学与技术学院
School of Information Science and Technology

CS 110
Computer Architecture
CALL

Instructors:

Siting Liu & Chundong Wang

Course website: https://toast-lab.sist.shanghaitech.edu.cn/courses/CS110@ShanghaiTech/Spring-2023/index.html

School of Information Science and Technology (SIST)

ShanghaiTech University

2023/2/6

CALL

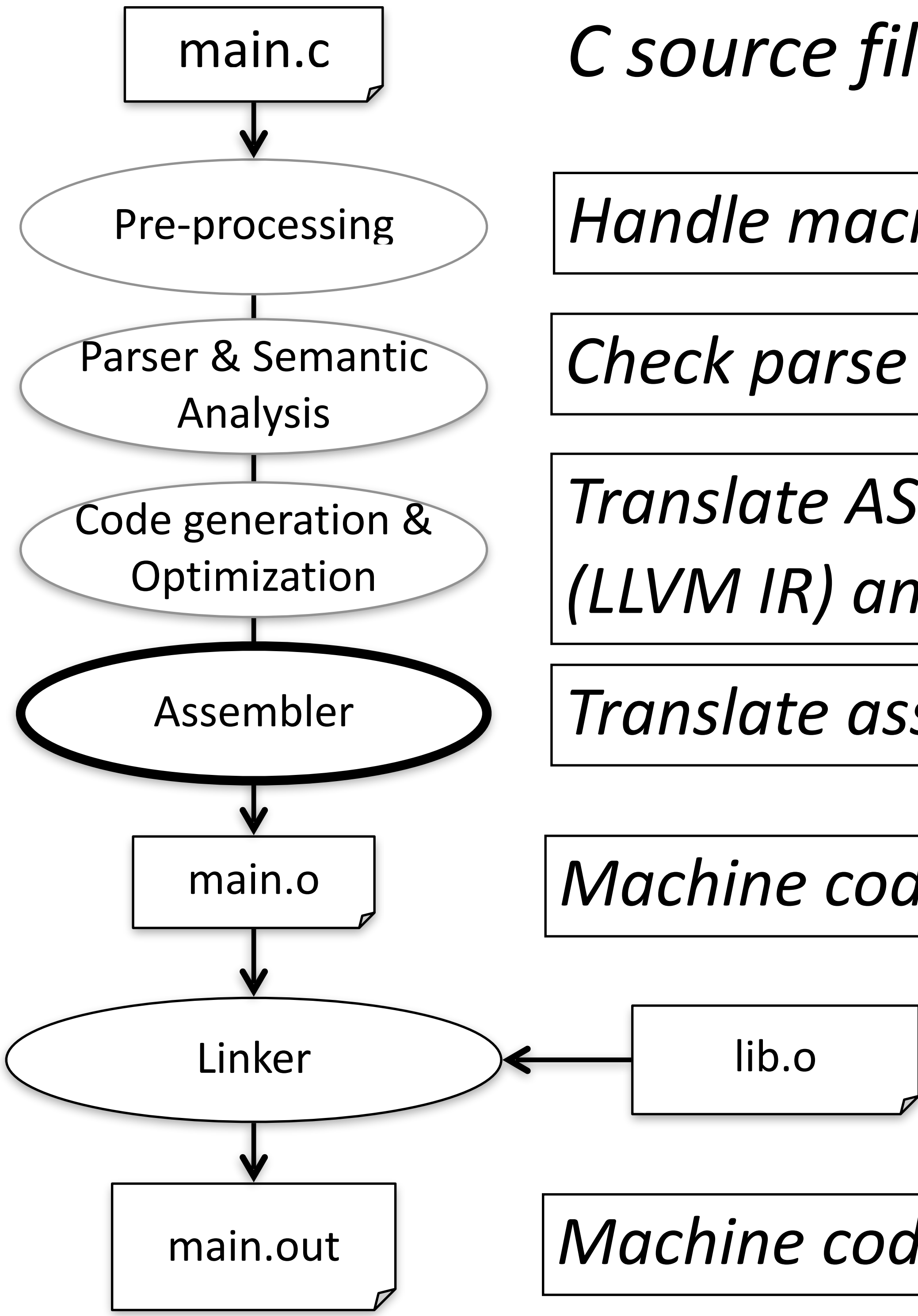
Compiler

Assembler

Linker

Loader

C Compilation Simplified Overview



C source files (text)

Handle macro expansion, #include, etc.

Check parse errors, generate “Abstract Syntax Tree” (AST)

Translate AST to lower level intermediate representation (LLVM IR) and then generate assembly (.s)

Translate assembly to machine code (binary, .o)

Machine code object files (.o)

Pre-built object file libraries

Machine code executable file (.out)

CALL

Compiler

Assembler

Linker

Loader

Assembler 汇编器

- Input: assembly language code (generated by compiler, usually contains pseudo-instructions)
- Output: object code, information tables
- Reads and uses `directives` 指令
- Replace pseudo-instructions
- Produce machine language
- Creates object file

Directives 指令

- Give directions to assembler, but do not produce machine instructions
  - `.text`: Subsequent items put in user text segment (instructions)
  - `.data`: Subsequent items put in user data segment (binary rep of data in source file)
  - `.globl sym`: declares sym global and can be referenced from other files
  - `.asciiz str`: Store the string str in memory and null-terminate it
  - `.word w1…wn`: Store the n 32-bit quantities in successive memory words

Pseudo-instruction Examples

Assembler

Pseudo	Real
nop	addi x0, x0, 0
not rd, rs	xori rd, rs, -1
beqz rs, offset	beq rs, x0, offset
bgt rs1, rs2, offset	blt rs2, rs1, offset
j offset	jal x0, offset
ret	jalr x0, x1, offset
call offset (too big to jal)	auipc x6, offset[31:12] jalr x1, x6, offset[11:0]
tail offset (too far to j)	auipc x6, offset[31:12] jalr x0, x6, offset[11:0]
li/la rd imm/label	lui rd <hi20bits> (too large) addi rd, x0, <low12bits>
mv rs1, rs2	addi rs1, rs2, 0

Tail

- Nested procedures

a1 a0 caller

x11 x10

```
int fact (int n, int prod) {
    if (n>1) return fact(n-1, prod*n);
    else return (prod);
}
```

```
fact: addi   t0,x0,1
      ble    x11,t0,Exit
      mul    x10,x10,x11
      addi   x11,x11,-1
      jalr   x0,fact
Exit: addi   x10,x0,x10
      jalr   x0,0(x1)
```

Producing Machine Language (1/3)

- Simple Case
  - Arithmetic, Logical, Shifts, and so on
  - All necessary info is within the instruction already
- What about Branches?
  - PC-Relative (e.g., `beq`/`bne` and `jal`), position-independent code (PIC), within one file
  - So once pseudo-instructions are replaced by real ones, we know by how many instructions to branch

Producing Machine Language (2/3)

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

3 instructions forward

3 instructions **back**

```
      addi t2,  zero,  9      # t2 = 9
L1:   slt  t1,  zero,  t2     # 0 < t2? Set t1
      beq  t1,  zero,  L2     # NO! t2 <= 0; Go to L2
      addi t2,  t2, -1# YES! t2 > 0; t2--
      j  L1                   # Go to L1
L2:
```

  - Solved by taking two passes over the program
    - First pass remembers position of labels (symbol table)
    - Second pass uses label position to generate code

Producing Machine Language (3/3)

- What about jumps (`j`, `jal`)?
  - Jumps within a file are PC relative (and we can easily compute):
    - Just count the number of instructions between target and jump to determine the offset: position-independent code (PIC)
  - Jumps to other files we can't
- What about references to static data/external functions/multiple files?
  - `la` gets broken up into `lui` and `addi`
  - These 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" in this file that may be used by other files
- What are they?
  - Labels: function calling; .global directive
  - Data: anything in the `.data` section; variables which may be accessed across files

Relocation Table

- List of "items" whose (absolute) address this file needs later. What are they?
  - Any external label jumped to: `jal`, `jalr`
    - External (including lib files)
  - Any piece of data in static section
    - Such as the `la` instruction
      E.g., for `lw`/`sw` base register

Summary: Object File Format

- <u>object file header</u>: size and position of the other pieces of the object file
- <u>text segment</u>: the machine code
- <u>data segment</u>: binary representation of the static data in the source file
- <u>relocation information</u>: identifies lines of code that need to be fixed up later (by linker)
- <u>symbol table</u>: list of this file's labels and static data that can be referenced
- <u>debugging information</u>
- A standard format is ELF (except MS) http://www.skyfree.org/linux/references/ELF_Format.pdf

CALL

Compiler

Assembler

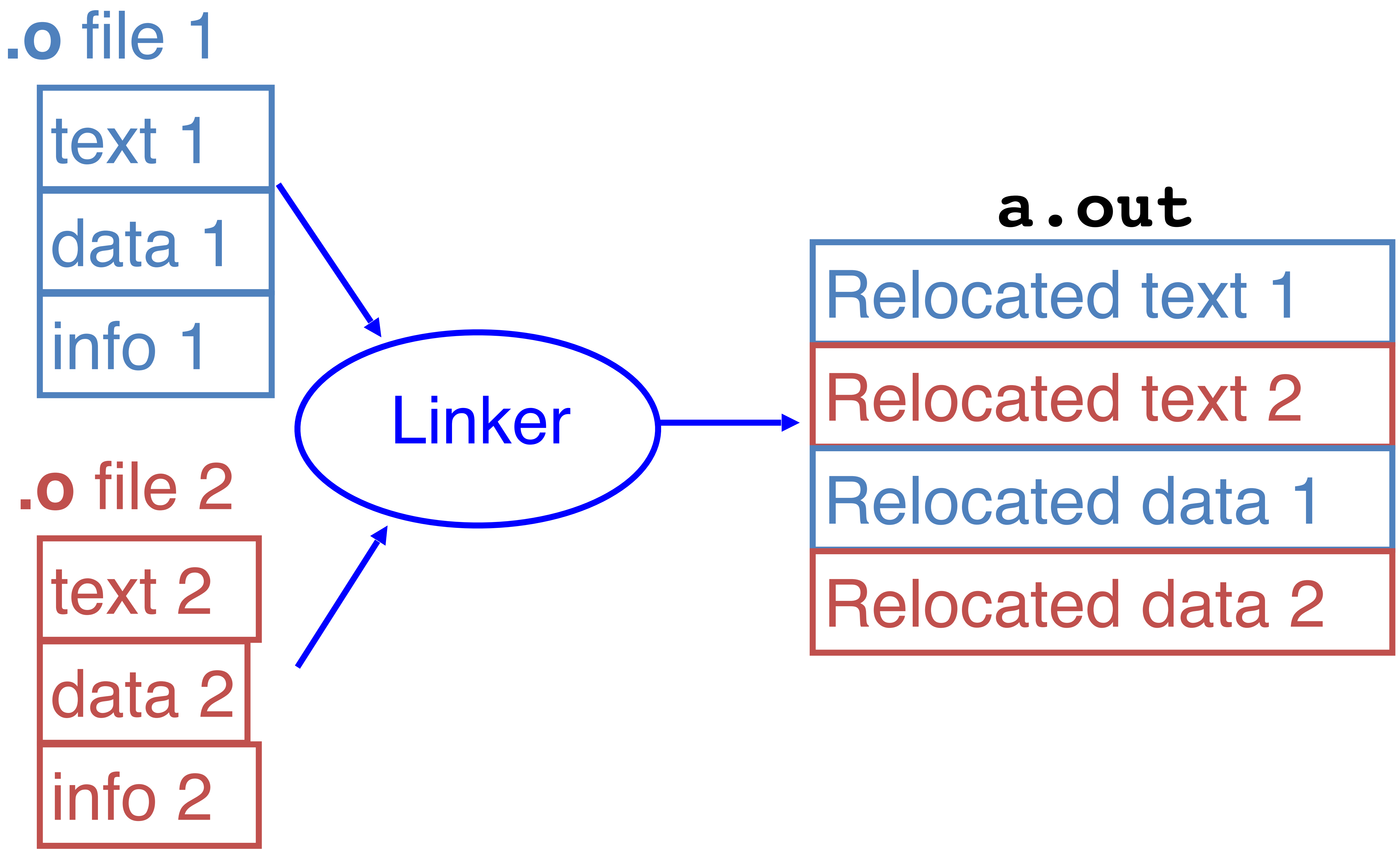
Linker

Loader

Linker (1/3)

- Input: Object code files, information tables (e.g., `<your C code>.o`, `libc.o` for RISC-V)
- Output: Executable code (e.g., a.out for RISC-V)
- Combines several object (.o) files into a single executable ("linking")
- Enable separate compilation of files
  - Changes to one file do not require recompilation of the whole program
    - Linux source > 20 M lines of code!
  - Old name "Link Editor" from editing the "links" in jump and link instructions

Linker (2/3)


.o file 1
text 1
data 1
info 1
Linker
.o file 2
text 2
data 2
info 2
a.out
Relocated text 1
Relocated text 2
Relocated data 1
Relocated data 2

Linker (3/3)

- Step 1: 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
- Step 2: Determine the addresses of data and instruction labels
- Step 3: Resolve references
  - Go through Relocation Table; handle each entry
  - That is, fill in all absolute addresses

Three Types of Addresses

- PC-Relative Addressing (`beq`, `bne`, `jal`)
  - Never need to relocate (PIC: position independent code)
- External Function Reference (usually `jal`)
  - Always relocate
- Static Data Reference (often `auipc`/`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`

Absolute Addresses in RISC-V

- Which instructions need relocation editing?
  - J-format: jump and link: ONLY for external jumps

`xxxxx`	`rd`	`jal`

  - I-,S- Format: Loads and stores to variables in static area, relative to global pointer

`xxx`		`gp`		`rd`	`lw`
`xx`	`rs1`	`gp`		`x`	`sw`

  - What about conditional branches?

`xx`	`rs1`	`rs2`		`x`	`beq` `bne`

  - PC-relative addressing preserved even if code moves

Resolving References (1/2)

- 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 and each piece of data being referenced

Resolving References (2/2)

- To resolve references:
  - search for reference (data or label) in all "user" symbol tables
  - if not found, search library files
    (for example, `printf, malloc`)
  - once absolute address is determined, fill in the machine code appropriately
- Output of linker: executable file containing text and data (plus header)

Static vs. Dynamic Linking

- What we've described is the traditional way: statically-linked approach
  - The library is now part of the executable, so if the library updates, we don't get the fix (have to recompile if we have source)
  - It includes the entire library even if not all of it will be used
  - Executable is self-contained
- An alternative is dynamically linked libraries (DLL), common on Windows (.dll) & UNIX (.so) & MacOS (.dylib) platforms

en.wikipedia.org/wiki/Dynamic_linking

Dynamically linked libraries

- Space/time issues
  - + Storing a program requires less disk space
  - + Sending a program requires less time
  - + Executing two programs requires less memory (if they share a library)
  - – At runtime, there's time overhead to do link
- Upgrades
  - + Replacing one file (libXYZ.so) upgrades every program that uses library "XYZ"
  - – Having the executable isn't enough anymore
  - Thus "containers": We hate dependencies, so we are just going to ship around all the libraries and everything else as part of the 'application'

Overall, dynamic linking adds quite a bit of complexity to the compiler, linker, and operating system. However, it provides many benefits that often outweigh these

CALL

Compiler
Assembler
Linker
Loader

Loader Basics

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

Loader … what does it do?

- Reads executable file's header to determine size of text and data segments
- Creates new address space for program large enough to hold text and data segments, along with a stack segment
- Copies instructions and data from executable file into the new address space
- Copies arguments passed to the program onto the stack
- Initializes machine registers
  - Most registers cleared, but stack pointer assigned address of 1st free stack location
- Jumps to start-up routine that copies program's arguments from stack to registers & sets the PC
  - If main routine returns, start-up routine terminates program with the exit system call

Question

At what point in process are all the machine code bits generated for the following assembly instructions:

1) `add x6, x7, x8`

2) `jal x1, fprintf` → Linker时才找到

A: 1) & 2) After compilation

B: 1) After compilation, 2) After assembly

C: 1) After assembly, 2) After linking

D: 1) After assembly, 2) After loading

E: 1) After compilation, 2) After linking

Answer

At what point in process are all the machine code bits determined for the following assembly instructions:

1) `add x6, x7, x8`
2) `jal x1, fprintf`

C: (1) After assembly, (2) After linking

Example

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

C -> assembly -> obj. -> exe.

Example

```
    .attribute stack_align, 16
    .text
    .section .rodata
    .align3
.LC0:
    .string  "world"
    .align3
.LC1:
    .string  "Hello, %s\n"
    .text
    .align1
    .globlmain
    .type main, @function
```

Store double word

```
main:
    addi    sp,sp,-16
    sd  ra,8(sp)
    sd  s0,0(sp)
    addi    s0,sp,16
    lui a5,%hi(.LC0)
    addi    a1,a5,%lo(.LC0)
    lui a5,%hi(.LC1)
    addi    a0,a5,%lo(.LC1)
    call    printf
    li  a5,0
    mv  a0,a5
    ld  ra,8(sp)
    ld  s0,0(sp)
    addi    sp,sp,16
    jr  ra
    .size main, .-main
    .ident"GCC: (g2ee5e430018-dirty)
```

C -> **assembly** -> obj. -> exe.

Example

```
0000000000000000 <main>:
   0: 1141                  add sp,sp,-16
   2: e406                  sd  ra,8(sp)
   4: e022                  sd  s0,0(sp)
   6: 0800                  add s0,sp,16
   8: 000007b7            lui   a5,0x0
   c: 00078593            mv    a1,a5
  10: 000007b7            lui   a5,0x0
  14: 00078513            mv    a0,a5
  18: 00000097            auipc ra,0x0
  1c: 000080e7            jalr  ra # 18 <main+0x18>
  20: 4781                  li  a5,0
  22: 853e                  mv  a0,a5
  24: 60a2                  ld  ra,8(sp)
  26: 6402                  ld  s0,0(sp)
  28: 0141                  add sp,sp,16
  2a: 8082                  ret
```

RVC included

Address placeholder

gcc -c

C -> assembly -> obj. -> exe.

Example

objdump -r hello.o

RELOCATION RECORDS FOR [.text]:

OFFSET	**TYPE**		VALUE
0000000000000008	**R_RISC**	V_HI20	.LC0
0000000000000008	**R_RISC**	V_RELAX	*ABS*
000000000000000c	**R_RISC**	V_LO12_I	.LC0
000000000000000c	**R_RISC**	V_RELAX	*ABS*
0000000000000010	**R_RISC**	V_HI20	.LC1
0000000000000010	**R_RISC**	V_RELAX	*ABS*
0000000000000014	**R_RISC**	V_LO12_I	.LC1
0000000000000014	**R_RISC**	V_RELAX	*ABS*
0000000000000018	**R_RISC**	V_CALL_PLT	printf
0000000000000018	**R_RISC**	V_RELAX	*ABS*

C -> assembly -> **obj.** -> exe.

Example

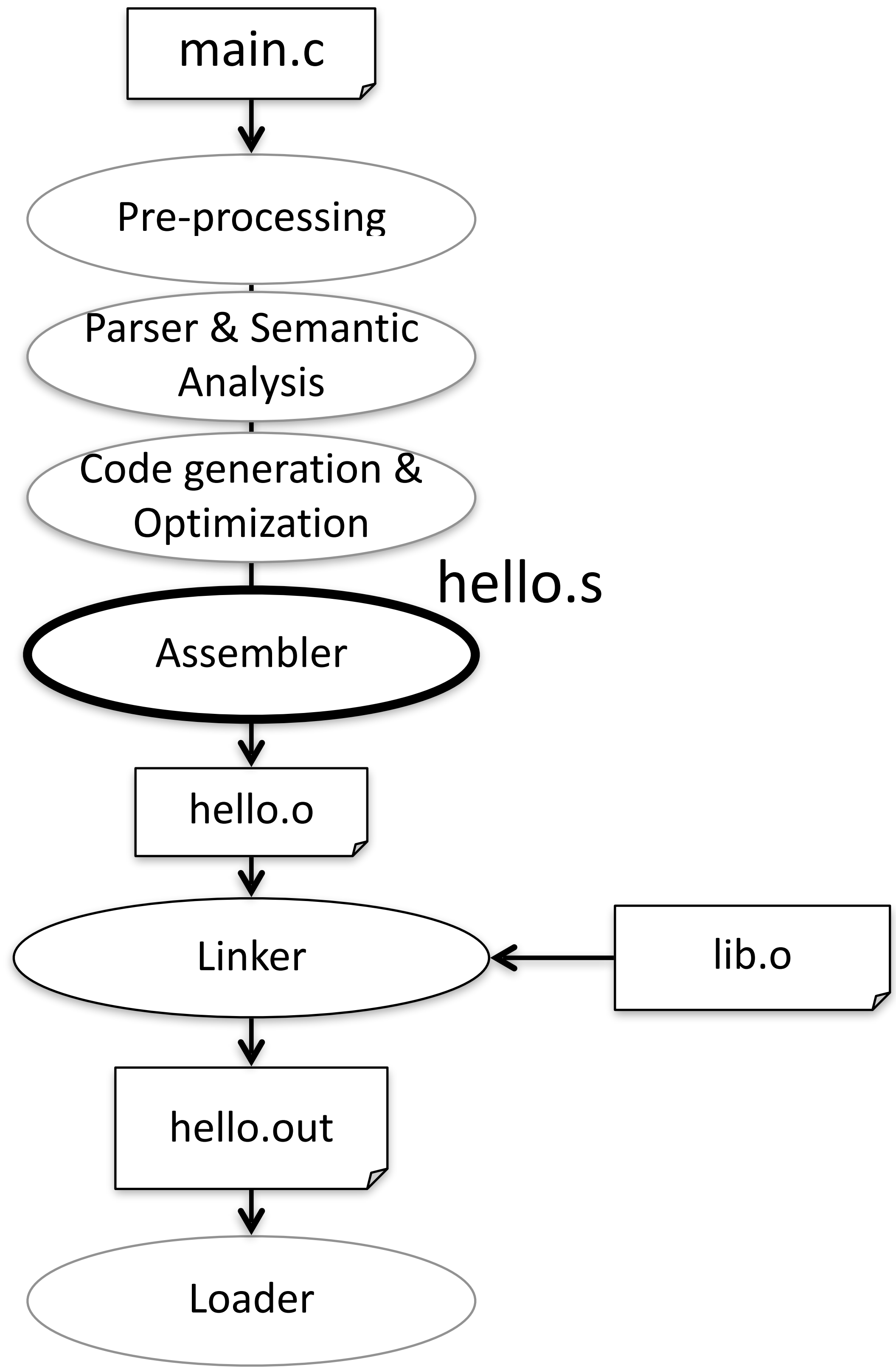
```
0000000000101ac <main>:
   101ac: 1141                       add sp,sp,-16
   101ae: e406                       sd  ra,8(sp)
   101b0: e022                       sd  s0,0(sp)
   101b2: 0800                       add s0,sp,16
   101b4: 67f5                       lui a5,0x1d
   101b6: a4078593                 add   a1,a5,-1472 # 1ca40 <__clzdi2+0x46>
   101ba: 67f5                       lui a5,0x1d
   101bc: a4878513                 add   a0,a5,-1464 # 1ca48 <__clzdi2+0x4e>
   101c0: 146000ef                 jal   10306 <printf>
   101c4: 4781                       li  a5,0
   101c6: 853e                       mv  a0,a5
   101c8: 60a2                       ld  ra,8(sp)
   101ca: 6402                       ld  s0,0(sp)
   101cc: 0141                       add sp,sp,16
   101ce: 8082                       ret
```

and the other libs

C -> assembly -> obj. -> **exe.**

In Conclusion…

- Compiler converts a single HLL file into a single assembly language file.
- Assembler removes pseudo-instructions, converts what it can to machine language, and creates a checklist for the linker (relocation table). A `.s` file becomes a `.o` file.
  - Does 2 passes to resolve addresses, handling internal forward references
- Linker combines several `.o` files and resolves absolute addresses.
  - Enables separate compilation, libraries that need not be compiled, and resolves remaining addresses
- Loader loads executable into memory and begins execution.



In Conclusion…

- Compiler converts a single HLL file into a single assembly language file.
- Assembler removes pseudo-instructions, converts what it can to machine language, and creates a checklist for the linker (relocation table). A `.s` file becomes a `.o` file.
  - Does 2 passes to resolve addresses, handling internal forward references
- Linker combines several `.o` files and resolves absolute addresses.
  - Enables separate compilation, libraries that need not be compiled, and resolves remaining addresses
- Loader loads executable into memory and begins execution.