

ECCS 665
COMPILER
CONSTRUCTION

Postcompilation

Previously...

Review – The Heap

Heap Memory

- Using the heap
- OS interface

Garbage collection

- Reference Counting
- Mark and Sweep

You Should Know

- How to program with heap memory
- The basic concepts of the two garbage collection schemes



Machine Codegen



Today's Lecture

Postcompilation

Compiler Toolchains

- Overview

Component Definitions

- Detour - History
- What GCC Does

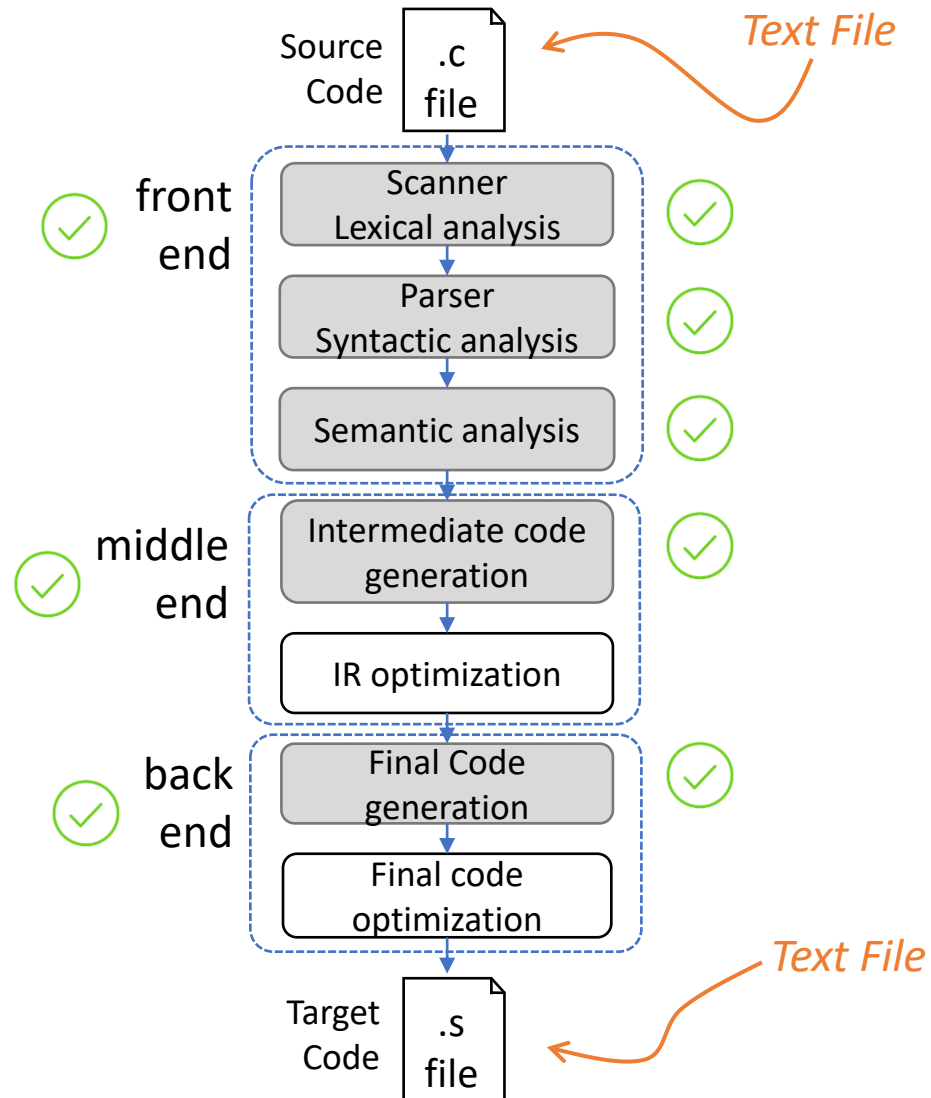
Component Walkthrough

- Assembler
- Linker
- Loader



Compiler Construction: Status

Progress Pics



Implemented all modules of a non-optimizing compiler

- Front end, middle end, and back end are all complete

One detail remains

- What use is an assembly text file?

OS Perspective: Still Just Text

Compiler Toolchains - Overview

From text file to text file

- Source code and assembly code equally useless to processor

Pretty unlike that runtime understands a text format

- Nearly always a better binary representation
- Some work left after the compiler has finished



***From text file to text file:
OS sees nothing accomplished***

OS Perspective: Still Just Text

Compiler Toolchains - Overview

From text file to text file

- Source code and assembly code equally useless to processor

Pretty unlike that runtime understands a text format

- Nearly always a better binary representation
- Some work left after the compiler has finished

“Shouldn't the compiler target that?”



Compiler doesn't (need to) output object code / executables

“But when I run gcc I get a binary”

- *You, maybe*

Then perhaps gcc is more than a compiler!



Did I just blow your mind?

Probably not, because it's just a matter of definitions

Compiler vs Compiler Toolchain

(A pedantic distinction)

Program lifecycle components often collectively referred to as “compiler” (vs. compiler toolchain)*

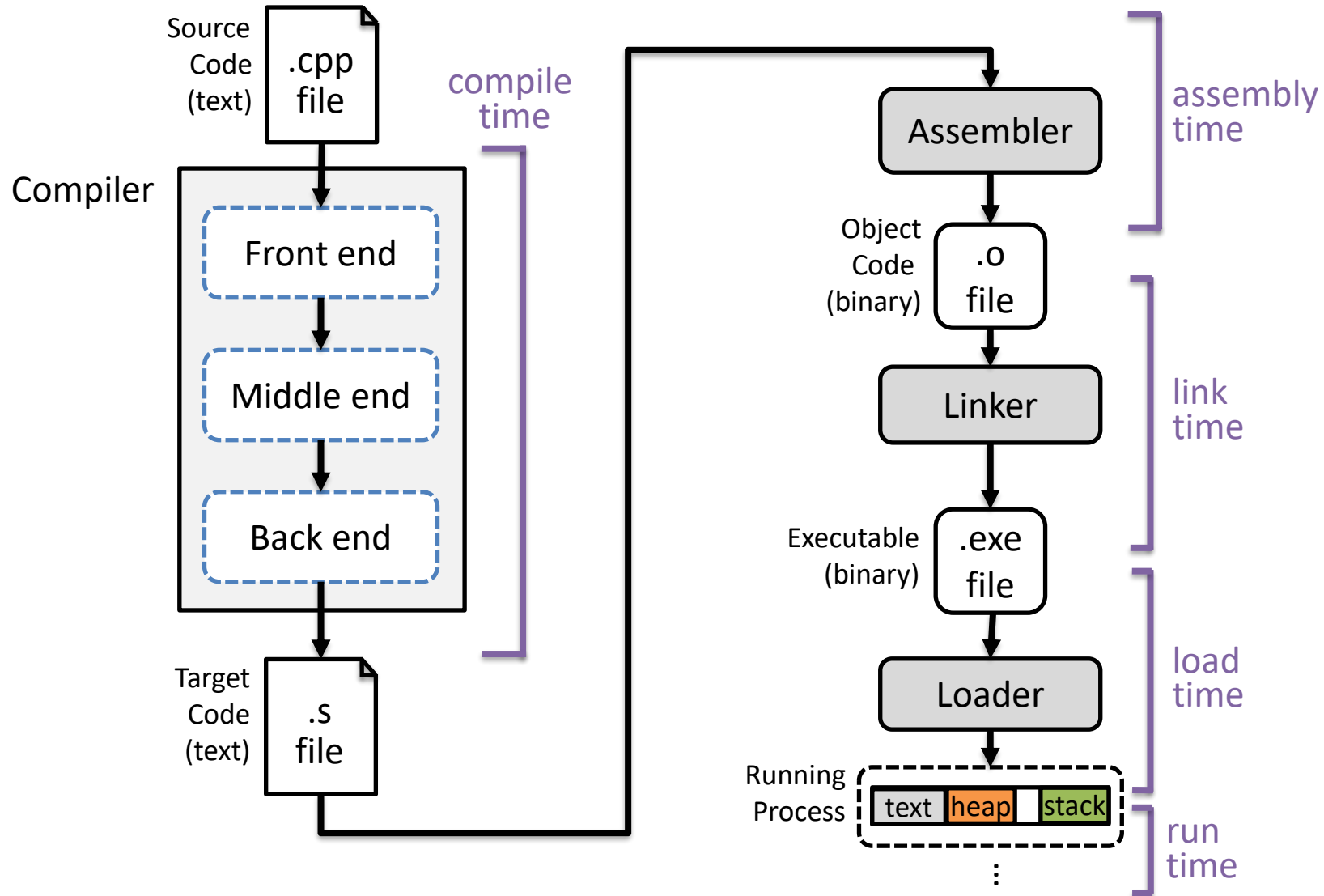
- These components *can* be invoked separately



[*]: Unfortunately (in my opinion) this convention kinda erases the term for the high-level text code to low-level text code component

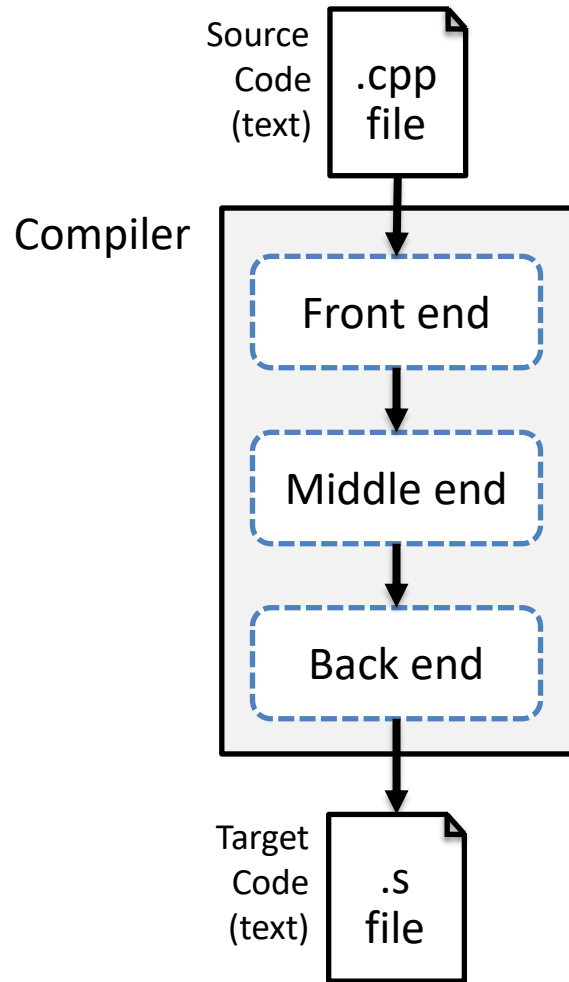
The Compiler Toolchain

Postcompilation - Component Walkthrough



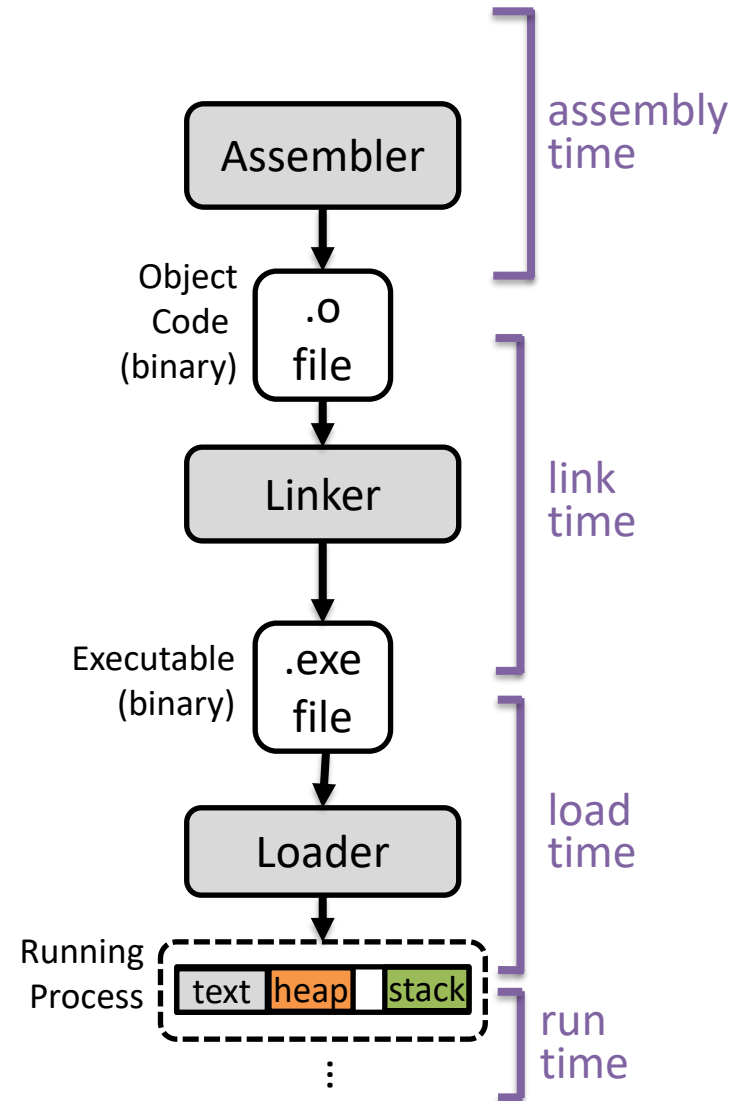
The Compiler Toolchain

Postcompilation - Component Walkthrough



The Compiler Toolchain

Postcompilation - Component Walkthrough



Today's Lecture

Postcompilation

Compiler Toolchains

- Overview

Component Definitions

- Detour - History
- What GCC Does

Component Walkthrough

- Assembler
- Linker
- Loader



Defining Toolchain Components

Postcompilation - Component Walkthrough

DEFINITIONS

(They're a little bit fuzzy)

Historical Factors

Detour: Component Terminology



- Compiler toolchain largely evolved from need to handle more complex programming tasks
 - Definitions have likewise evolved, with somewhat uneven consensus

Genesis of “Computer”

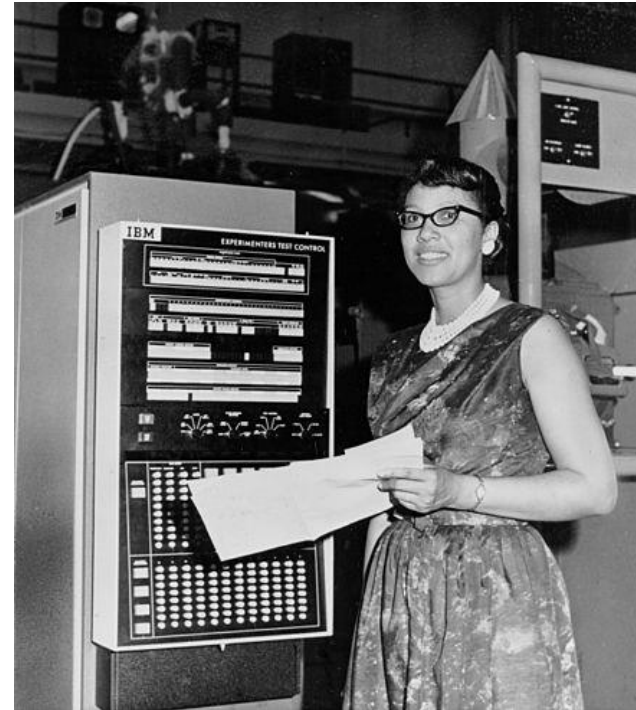
Detour: Component Terminology



From 1613-1945:

- “Computer” exclusively meant “a person who carries out computations”

Original programming consists of inputting numeric machine codes



Melba Roy Mouton: Early “human computer”

Genesis of “Assembler”

Detour: Component Terminology



1949: EDSAC features first
code akin to modern
notion of assembler

1953: Term “assembler”
popularized for a program
that concretized symbolic
programs



Maurice Wilkes: developed
assembler for EDSAC



Nathanial Rochester:
Published assembler
design

Genesis of “Linker”

Detour: Component Terminology



1952: A-0 system features first code akin to the modern notion of a linker (confusingly called a compiler at the time)

Allowed multiple programs to be combined together before being run



Grace Hopper: developed A-0

Genesis of “Compiler”

Detour: Component Terminology



1951: Publication of a translator from a high-level language to low-level code, considered the first compiler

(Also specifies the compiler in new high-level language)



Corrado Böhm: published first compiler in his PhD thesis

Genesis of “Loader”

Detour: Component Terminology



1947: ENIAC introduces the notion of relocating code to accommodate pre-defined subroutines, a feature somewhat akin to loaders (though pre-OS)



John Mauchley: Wrote about ENIAC procedures to relocate code in memory so that debugged subprograms could be loaded

Enough History for Now

Detour: Component Terminology



Practical upshot: terms have changed somewhat

Program Lifecycle for Our Purpose

Postcompilation - Component Walkthrough

- Compile high-level source code text to low-level assembly code text
- Assemble assembly code into object code binary modules
- Link object code into a single executable
- Load program into memory and run

Compiler

cc

Assembler

as

Linker

ld

Loader

(launch program)

These modules
can be invoked
separately!

Today's Lecture

Postcompilation

Compiler Toolchains

- Overview

Component Definitions

- Detour - History
- What GCC Does

Component Walkthrough

- Assembler
- Linker
- Loader



The gcc “subprograms”

Fun home experiment:

- Create empty.c

```
int main() {}
```

- Run gcc in verbose mode on empty.c

```
cycle1: gcc -v empty.c
```

a807d786@cycle1:~/libc_link\$ gcc -v empty.c

Target: x86_64-linux-gnu

[...]

Thread model: posix

gcc version 7.5.0 (Ubuntu 7.5.0-3ubuntu1~18.04)

COLLECT_GCC_OPTIONS='-v' '-mtune=generic' '-march=x86-64'

/usr/lib/gcc/x86_64-linux-gnu/7/cc1 -quiet -v -imultiarch x86_64-linux-gnu empty.c -quiet -dumpbase empty.c -mtune=generic -march=x86-64 -auxbase empty -version -fstack-protector-strong -Wformat -Wformat-security -o /tmp/ccs6BRmG.s

[...]

GNU C11 (Ubuntu 7.5.0-3ubuntu1~18.04) version 7.5.0 (x86_64-linux-gnu)

compiled by GNU C version 7.5.0, GMP version 6.1.2, MPFR version 4.0.1, MPC version 1.1.0, isl version isl-0.19-GMP

The assembler

GGC heuristics: --param ggc-min-expand=100 --param ggc-min-heapsize=131072

Compiler executable checksum: b62ed4a2880cd4159476ea8293b72fa8

COLLECT_GCC_OPTIONS='-v' '-mtune=generic' '-march=x86-64'

as -v --64 -o /tmp/cc4FfCb1.o /tmp/ccs6BRmG.s

[...]

Object code

COLLECT_GCC_OPTIONS='-v' '-mtune=generic' '-march=x86-64'

/usr/lib/gcc/x86_64-linux-gnu/7/collect2 -plugin /usr/lib/gcc/x86_64-linux-gnu/7/liblto_plugin.so -plugin-opt=/usr/lib/gcc/x86_64-linux-gnu/7/lto-wrapper -plugin-opt=-fresolution=/tmp/ccAxjj1l.res -plugin-opt=-pass-through=-lgcc -plugin-opt=-pass-through=-lgcc_s -plugin-opt=-pass-through=-lc -plugin-opt=-pass-through=-lgcc -plugin-opt=-pass-through=-lgcc_s --build-id --eh-frame-hdr -m elf_x86_64 --hash-style=gnu --as-needed -dynamic-linker /lib64/ld-linux-x86-64.so.2 -pie -z now -z relro /usr/lib/gcc/x86_64-linux-gnu/7/../../../../x86_64-linux-gnu/Scrt1.o /usr/lib/gcc/x86_64-linux-gnu/7/../../../../x86_64-linux-gnu/crti.o /usr/lib/gcc/x86_64-linux-gnu/7/crtbeginS.o -L/usr/lib/gcc/x86_64-linux-gnu/7 -L/usr/lib/gcc/x86_64-linux-gnu/7/../../../../x86_64-linux-gnu -L/usr/lib/gcc/x86_64-linux-gnu/7/../../../../lib -L/lib/x86_64-linux-gnu -L/lib/../../lib -L/usr/lib/x86_64-linux-gnu -L/usr/lib/../../lib -L/usr/lib/x86_64-linux-gnu -L/usr/lib/gcc/x86_64-linux-gnu/7/../../../../tmp/cc4FfCb1.o -lgcc --push-state --as-needed -lgcc_s --pop-state -lc -lgcc --push-state --as-needed -lgcc_s --pop-state /usr/lib/gcc/x86_64-linux-gnu/7/crtendS.o /usr/lib/gcc/x86_64-linux-gnu/7/../../../../x86_64-linux-gnu/crtn.o

The linker

Source code (input)

Assembly code

Runtime components

Today's Lecture

Postcompilation

Compiler Toolchains

- Overview

Component Definitions

- Detour - History
- What GCC Does

Component Walkthrough

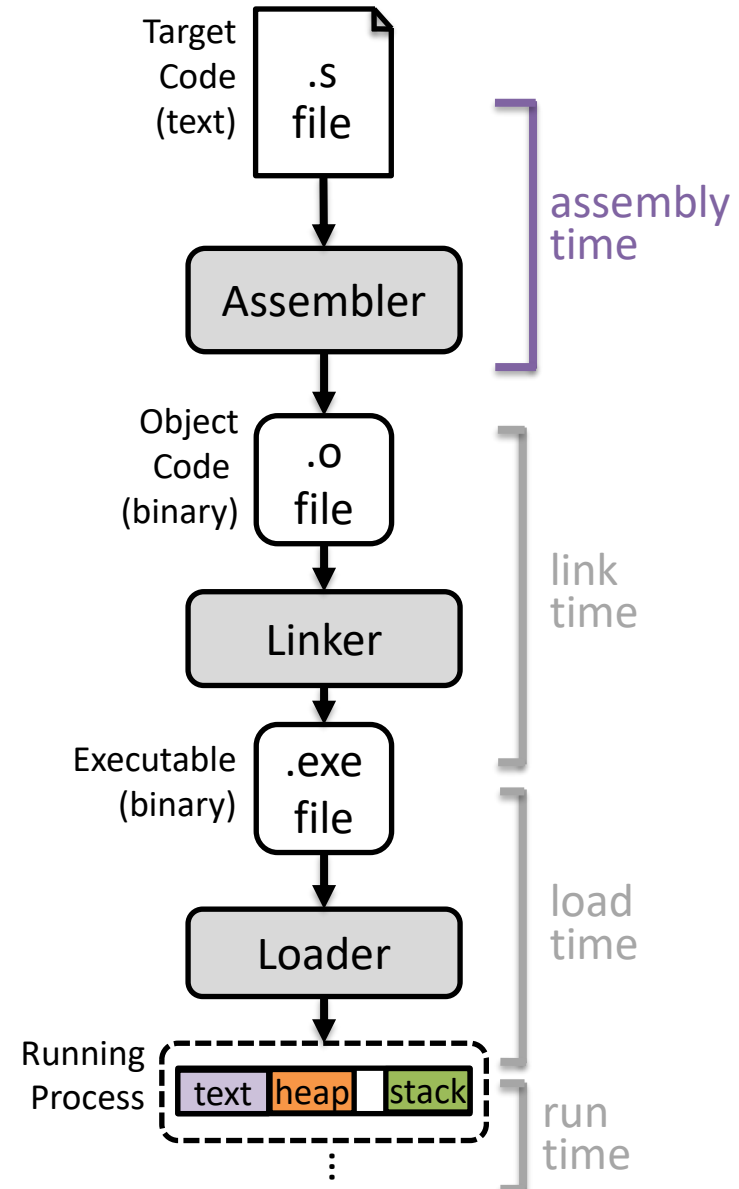
- Assembler
- Linker
- Loader



The Assembler

Module Role and Operation

- Modern use:
 - Translates labels and instruction mnemonics to binary equivalent
- Output may still contain placeholders for some code and data



The Assembler: Responsibilities

Module Role and Operation

- Collect symbolic definitions (e.g. labels) into “low-level” symbol table
- Put code and data into segments
- Replace uses of labels with placeholder addresses (where available)

```
.data
_label1: .word

.text
jal _label7

.data
_str: .asciiz ...

.text
la $t0 _label1
```



symbols

```
.data
(binary)

.text
(binary)
```

The Assembler

Module Role and Operation

src code

```
1 .data
2 gbl: .quad 12
3 .globl _start
4 .text
5 _start: jmp _start
6
```

as -o out.o file.s
objdump -Dwrt out.o

object code

```
1
2 code.o:      file format elf64-x86-64
3
4 SYMBOL TABLE:
5 0000000000000000 l      d  .text 0000000000000000 .text
6 0000000000000000 l      d  .data 0000000000000000 .data
7 0000000000000000 l      d  .bss  0000000000000000 .bss
8 0000000000000000 l      .data 0000000000000000 gbl
9 0000000000000000 g      .text 0000000000000000 _start
10
11
12 Disassembly of section .text:
13
14 0000000000000000 <_start>:
15     0:  eb fe                jmp     0 <_start>
16
17 Disassembly of section .data:
18
19 0000000000000000 <gbl>:
20     0:  0c 00                or      $0x0,%al
21     2:  00 00                add     %al,(%rax)
22     4:  00 00                add     %al,(%rax)
23
24     ...
```

The Assembler: Invocation

Module Role and Operation

```
(to get assembly): cc -S ret3.c -o /tmp/a.s
```

```
as /tmp/a.s -o /tmp/a.o
```

Today's Lecture

Postcompilation

Compiler Toolchains

- Overview

Component Definitions

- Detour - History
- What GCC Does

Component Walkthrough

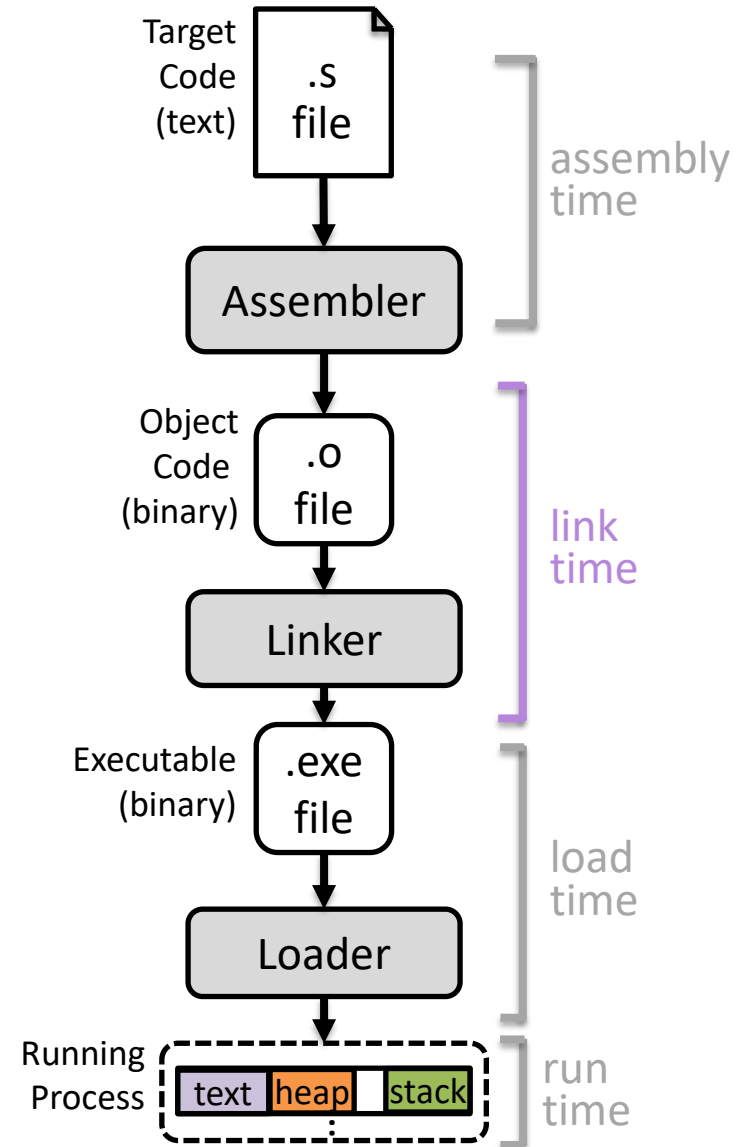
- Assembler
- **Linker**
- Loader



The Linker

Module Role and Operation

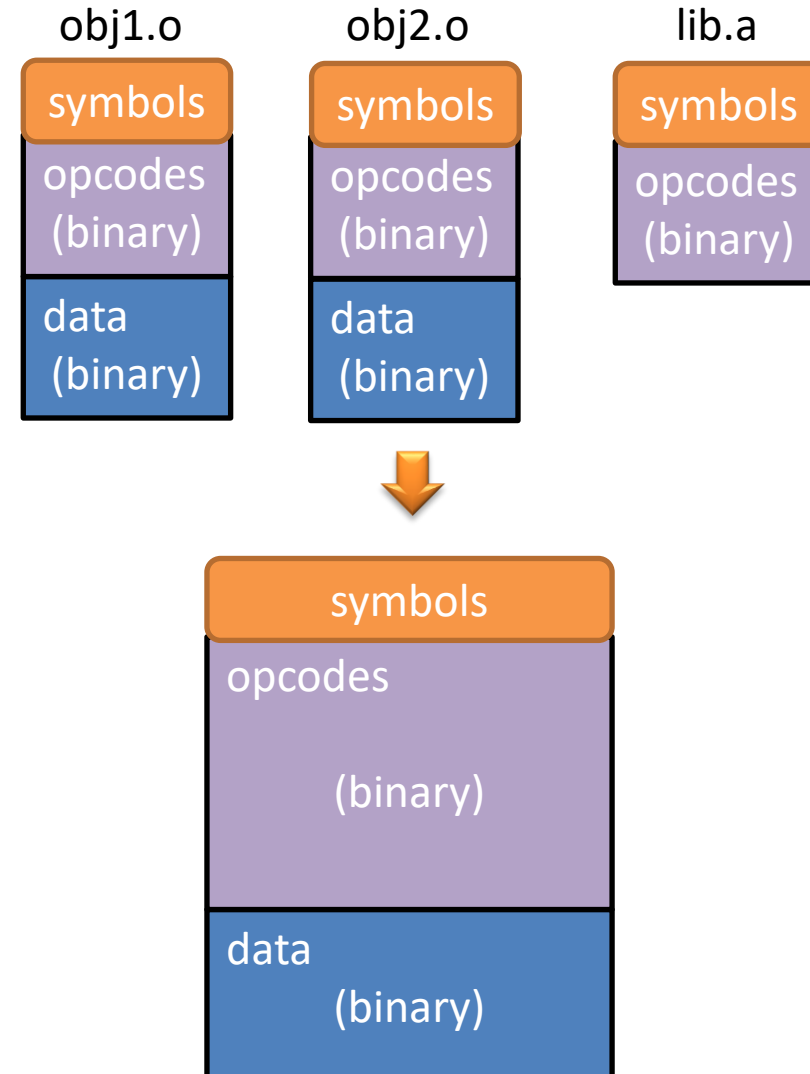
- Resolves placeholders betwixt multiple object files
- Stitches objects files (and static libraries) into a single binary



The Linker: Responsibilities

Module Role and Operation

- Calculate segment size
- Relocate segments
- Resolve symbol imports with exports where available
 - *e.g. call in obj1.o, callee in obj2.o*
- Add initialization routines
- Optimization!



The Linker

Module Role and Operation

```
ld -o code.prog code.o  
objdump -Dwrt prog
```

Object code

```
1  
2 code.o:      file format elf64-x86-64  
3  
4 SYMBOL TABLE:  
5 0000000000000000 l    d  .text 0000000000000000 .text  
6 0000000000000000 l    d  .data 0000000000000000 .data  
7 0000000000000000 l    d  .bss  0000000000000000 .bss  
8 0000000000000000 l    d  .data 0000000000000000 gbl  
9 0000000000000000 g      .text 0000000000000000 _start  
10  
11  
12  
13 Disassembly of section .text:  
14  
15 0000000000000000 <_start>:  
16  0:  eb fe                jmp     0 <_start>  
17  
18 Disassembly of section .data:  
19  
20 0000000000000000 <gbl>:  
21  0:  0c 00                or      $0x0,%al  
22  2:  00 00                add     %al,(%rax)  
23  4:  00 00                add     %al,(%rax)  
24  ...
```

Executable code

```
1  
2 code.prog:    file format elf64-x86-64  
3  
4 SYMBOL TABLE:  
5 00000000004000b0 l    d  .text 0000000000000000 .text  
6 00000000006000b2 l    d  .data 0000000000000000 .data  
7 0000000000000000 l    df *ABS* 0000000000000000 code.o  
8 00000000006000b2 l    d  .data 0000000000000000 gbl  
9 00000000004000b0 g      .text 0000000000000000 _start  
10 00000000006000ba g      .data 0000000000000000 __bss_start  
11 00000000006000ba g      .data 0000000000000000 _edata  
12 00000000006000c0 g      .data 0000000000000000 _end  
13  
14  
15  
16 Disassembly of section .text:  
17  
18 00000000004000b0 <_start>:  
19  4000b0:  eb fe                jmp     4000b0 <_start>  
20  
21 Disassembly of section .data:  
22  
23 00000000006000b2 <gbl>:  
24  6000b2:  0c 00                or      $0x0,%al  
25  6000b4:  00 00                add     %al,(%rax)  
26  6000b6:  00 00                add     %al,(%rax)  
27  ...
```

The Linker: Simple Invocation

Module Role and Operation



**Project
concept**

The simplest use, “linking” a single object file:

```
ld prog.o -o prog.exe
```

More complex use, linking multiple object files

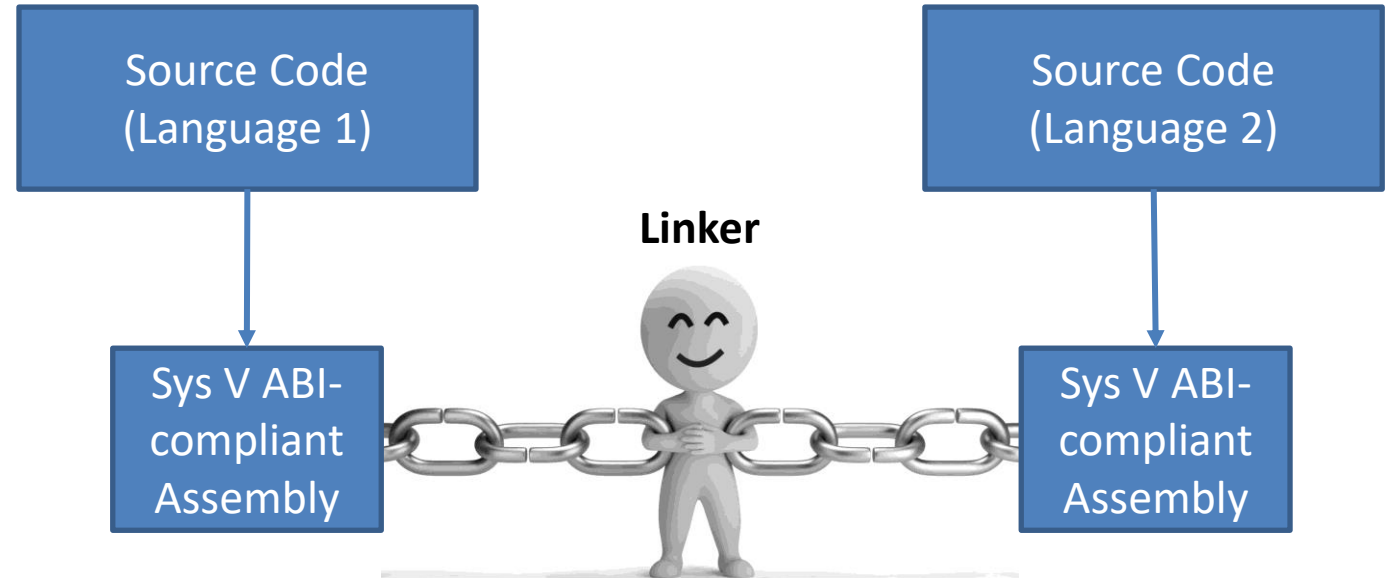
```
ld prog.o a.o b.o -o a.exe
```

Cross-language linking

Module Role and Operation

Object code is raw x64 code (containing placeholders)

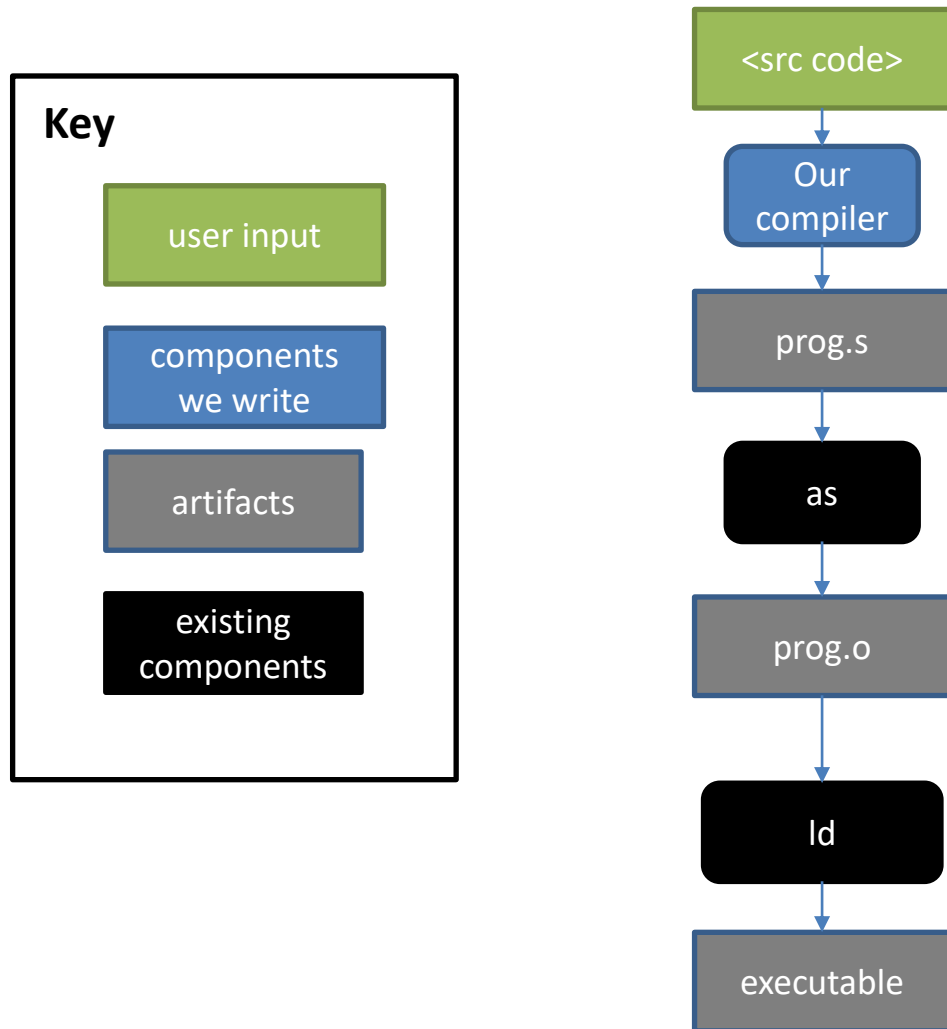
- Should obey the System V ABI
- We can link with object code from other languages!



Cross-language linking

Module Role and Operation

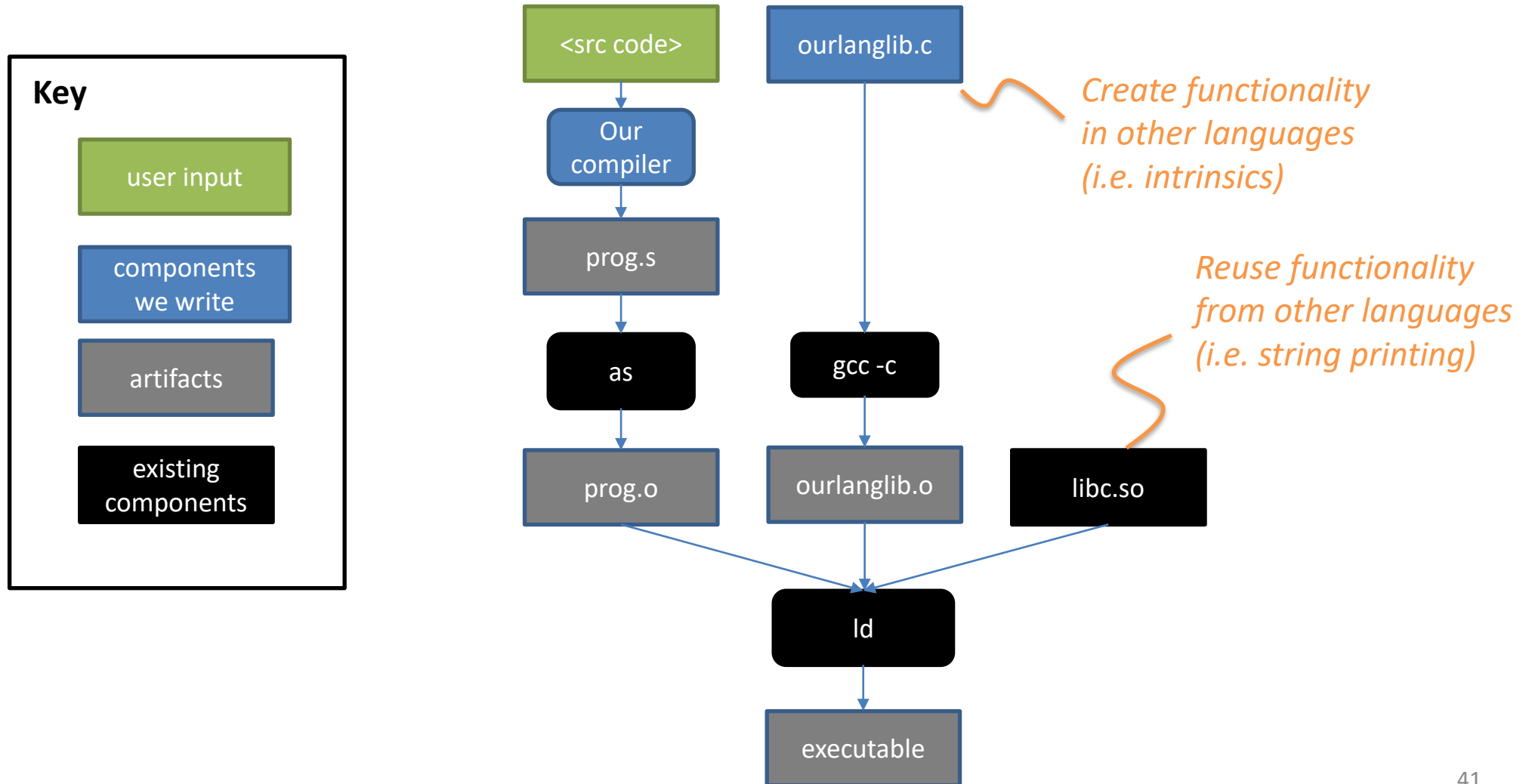
End-to-end compiler toolchain workflow



Cross-language linking

Module Role and Operation

End-to-end compiler toolchain workflow



Benefits of Cross-Linking

Module Role and Operation



**Project
concept**

Inter-language compatibility

- Let each language's strengths shine

Functionality reuse

- Avoid coding against the OS directly



Do not reinvent this

Why use the C Runtime?

Module Role and Operation



Src Code

```
int main() {  
    output 7;  
}
```



https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/

%rax	System call	%rdi	%rsi	%rdx
0	sys_read	unsigned int fd	char *buf	size_t count
1	sys_write	unsigned int fd	const char *buf	size_t count

Why use the C Runtime?

Module Role and Operation



**Project
concept**

Src Code

```
int main() {  
    output 7;  
}
```

Asm Code

```
.globl _start  
.text  
_start: callq fn_main  
        movq %rax, %rdi  
        movq $60, %rax  
        syscall  
fn_main: <fn prologue>  
        # output 7  
        ...  
        syscall  
        <fn epilogue>
```

%rax	System call	%rdi	%rsi	%rdx
0	sys_read	unsigned int fd	char *buf	size_t count
1	sys_write	unsigned int fd	const char *buf	size_t count

Why use the C Runtime?

Module Role and Operation



**Project
concept**

Src Code

```
int main() {  
    output 7;  
}
```

Asm Code

```
.globl _start  
.text  
_start: callq fn_main  
        movq %rax, %rdi  
        movq $60, %rax  
        syscall  
fn_main: <fn prologue>  
        # output 7  
        movq $1, %rax #select sys_write  
        ...  
        syscall  
        <fn epilogue>
```

%rax	System call	%rdi	%rsi	%rdx
0	sys_read	unsigned int fd	char *buf	size_t count
1	sys_write	unsigned int fd	const char *buf	size_t count

Why use the C Runtime?

Module Role and Operation



**Project
concept**

Src Code

```
int main() {  
    output 7;  
}
```

Asm Code

```
.globl _start  
.text  
_start: callq fn_main  
        movq %rax, %rdi  
        movq $60, %rax  
        syscall  
fn_main: <fn prologue>  
        # output 7  
        movq $1, %rax #select sys_write  
        movq $1, %rdi #write to stdout  
        ...  
        syscall  
        <fn epilogue>
```

%rax	System call	%rdi	%rsi	%rdx
0	sys_read	unsigned int fd	char *buf	size_t count
1	sys_write	unsigned int fd	const char *buf	size_t count

Why use the C Runtime?

Module Role and Operation



**Project
concept**

Src Code

```
int main() {  
    output 7;  
}
```

Asm Code

```
.globl _start  
.text  
_start: callq fn_main  
        movq %rax, %rdi  
        movq $60, %rax  
        syscall  
fn_main: <fn prologue>  
        # output 7  
        movq $1, %rax #select sys_write  
        movq $1, %rdi #write to stdout  
        movq ???????, %rsi #buffer address  
        ...  
        syscall  
        <fn epilogue>
```

%rax	System call	%rdi	%rsi	%rdx
0	sys_read	unsigned int fd	char *buf	size_t count
1	sys_write	unsigned int fd	const char *buf	size_t count

Why use the C Runtime?

Module Role and Operation



**Project
concept**

Src Code

```
int main() {  
    output 7;  
}
```

Asm Code

```
.data  
buffer: .asciz "?"  
.globl _start  
.text  
_start: callq fn_main  
        movq %rax, %rdi  
        movq $60, %rax  
        syscall  
fn_main: <fn prologue>  
        # output 7  
        movq $1, %rax #select sys_write  
        movq $1, %rdi #write to stdout  
        movq ???????, %rsi #buffer address  
        ...  
        syscall  
        <fn epilogue>
```

%rax	System call	%rdi	%rsi	%rdx
0	sys_read	unsigned int fd	char *buf	size_t count
1	sys_write	unsigned int fd	const char *buf	size_t count

Why use the C Runtime?

Module Role and Operation



**Project
concept**

Src Code

```
int main() {  
    output 7;  
}
```

Asm Code

```
.data  
buffer: .asciz "?"  
.globl _start  
.text  
_start: callq fn_main  
        movq %rax, %rdi  
        movq $60, %rax  
        syscall  
fn_main: <fn prologue>  
        # output 7  
        movq $1, %rax #select sys_write  
        movq $1, %rdi #write to stdout  
        movq ???????, %rsi #buffer address  
        ...  
        syscall  
        <fn epilogue>
```

%rax	System call	%rdi	%rsi	%rdx
0	sys_read	unsigned int fd	char *buf	size_t count
1	sys_write	unsigned int fd	const char *buf	size_t count

Why use the C Runtime?

Module Role and Operation



**Project
concept**

Src Code

```
int main() {  
    output 7;  
}
```

Asm Code

```
.data  
buffer: .asciz "?"  
.globl _start  
.text  
_start: callq fn_main  
        movq %rax, %rdi  
        movq $60, %rax  
        syscall  
fn_main: <fn prologue>  
        # output 7  
        movq $1, %rax #select sys_write  
        movq $1, %rdi #write to stdout  
        movq $buffer, %rsi #buffer address  
        ...  
        syscall  
        <fn epilogue>
```

%rax	System call	%rdi	%rsi	%rdx
0	sys_read	unsigned int fd	char *buf	size_t count
1	sys_write	unsigned int fd	const char *buf	size_t count

Why use the C Runtime?

Module Role and Operation



**Project
concept**

Src Code

```
int main() {  
    output 7;  
}
```

Asm Code

```
.data  
buffer: .asciz "?"  
.globl _start  
.text  
_start: callq fn_main  
        movq %rax, %rdi  
        movq $60, %rax  
        syscall  
fn_main: <fn prologue>  
        # output 7  
        movq $1, %rax #select sys_write  
        movq $1, %rdi #write to stdout  
        movq $buffer, %rsi #buffer address  
        movq $0x37, (buffer) movq $7, (buffer) #write to buffer  
        ...  
        syscall  
        <fn epilogue>
```

%rax	System call	%rdi	%rsi	%rdx
0	sys_read	unsigned int fd	char *buf	size_t count
1	sys_write	unsigned int fd	const char *buf	size_t count

Why use the C Runtime?

Module Role and Operation



**Project
concept**

Src Code

```
int main() {  
    output 7;  
}
```

Asm Code

```
.data  
buffer: .asciz "?"  
.globl _start  
.text  
_start: callq fn_main  
        movq %rax, %rdi  
        movq $60, %rax  
        syscall  
fn_main: <fn prologue>  
        # output 7  
        movq $1, %rax #select sys_write  
        movq $1, %rdi #write to stdout  
        movq $buffer, %rsi #buffer address  
        movq $0x37, (buffer)#write to buffer  
        ...  
        syscall  
        <fn epilogue>
```

%rax	System call	%rdi	%rsi	%rdx
0	sys_read	unsigned int fd	char *buf	size_t count
1	sys_write	unsigned int fd	const char *buf	size_t count

Why use the C Runtime?

Module Role and Operation



**Project
concept**

Src Code

```
int main() {  
    output 7;  
}
```

Asm Code

```
.data  
buffer: .asciz "?"  
.globl _start  
.text  
_start: callq fn_main  
        movq %rax, %rdi  
        movq $60, %rax  
        syscall  
fn_main: <fn prologue>  
        # output 7  
        movq $1, %rax #select sys_write  
        movq $1, %rdi #write to stdout  
        movq $buffer, %rsi #buffer address  
        movq $0x37, (buffer)#write to buffer  
        movq $1, %rdx #string length  
        syscall  
        <fn epilogue>
```

%rax	System call	%rdi	%rsi	%rdx
0	sys_read	unsigned int fd	char *buf	size_t count
1	sys_write	unsigned int fd	const char *buf	size_t count

Why use the C Runtime?

Module Role and Operation



**Project
concept**

Src Code

```
int main() {  
    output 7;  
}
```

Asm Code

```
.data  
buffer: .asciz "?"  
.globl _start  
.text  
_start: callq fn_main  
        movq %rax, %rdi  
        movq $60, %rax  
        syscall  
fn_main: <fn prologue>  
        # output 7  
        movq $1, %rax #select sys_write  
        movq $1, %rdi #write to stdout  
        movq $buffer, %rsi #buffer address  
        movq $0x37, (buffer)#write to buffer  
        movq $1, %rdx #string length  
        syscall  
        <fn epilogue>
```

**It gets even
worse!**

Why use the C Runtime?

Module Role and Operation



**Project
concept**

Src Code

```
...  
int main() {  
    output gbl;  
}
```

Asm Code

```
fn_main: <fn prologue>  
        # output gbl  
        movq $1, %rax #select sys_write  
        movq $1, %rdi #write to stdout  
        movq $buffer, %rsi #string to print  
        movq ??, (buffer) #write characters to buffer  
        movq ??, %rdx #string length  
        syscall  
        <fn epilogue>
```

Prefill Buffer to all null bytes?

Repeatedly call idivq \$10?

Loop over Buffer counting non-null bytes?

"I miss printf"

Gbl: 173



Buffer:

0x31	0x37	0x33
------	------	------

ASCII 1 ASCII 7 ASCII 3

Use the C Runtime

Module Role and Operation



**Project
concept**

Src Code

```
int main(){  
    output 7;  
}
```

Old assembly

```
.globl _start  
.data  
buffer: .asciz "?"  
.text  
_start: callq fn_main  
        movq %rax, %rdi  
        movq $60, %rax  
        syscall  
fn_main: <fn prologue>  
        # output 7  
        movq $1, %rax #select sys_write  
        movq $1, %rdi #write to stdout  
        movq $buffer, %rsi #string to print  
        movq $31, (buffer) #write to buffer  
        movq $1, %rdx #string length  
        syscall  
        <fn epilogue>
```

New Assembly

```
.text  
main: <fn prologue>  
        # output 7  
        movq $7, %rdi #int to print  
        callq myPrintInt  
        <fn epilogue>
```

myStdLib

```
#include <stdio.h>  
#include <inttypes.h>  
void myPrintInt(int64_t arg){  
    fprintf(stdout, "%ld", arg);  
}
```

Use the C Runtime

Module Role and Operation



**Project
concept**

Src Code

```
int main() {  
    output 7;  
}
```

prog.o

User code
(calls myPrintInt)

mystdlib.o

Shim code
(calls fprintf)

C runtime

fprintf definition
_start label
and so much more!

New Assembly

```
.text  
main: <fn prologue>  
    # output 7  
    movq $7, %rdi #int to print  
    callq myPrintInt  
    <fn epilogue>
```

myStdLib

```
#include <stdio.h>  
#include <inttypes.h>  
void myPrintInt(int64_t arg) {  
    fprintf(stdout, "%ld", arg);  
}
```

Basing our runtime on C's

Module Role and Operation



**Project
concept**

enable dynamic linking

```
SYSPATH=/usr/lib/x86_64-linux-gnu
```

```
ld \
```

```
-dynamic-linker /lib64/ld-linux-x86-64.so.2 \
```

```
$SYSPATH/crt1.o \
```

```
$SYSPATH/crti.o \
```

entrypoint code

```
_start: exit(main)
```

```
-lc \
```

link the libc library

init runtime data structures

```
prog.o \
```

```
$SYSPATH/crtn.o \
```

```
-o prog.exe
```

release runtime data structures

Today's Lecture

Postcompilation

Compiler Toolchains

- Overview

Component Definitions

- Detour - History
- What GCC Does

Component Walkthrough

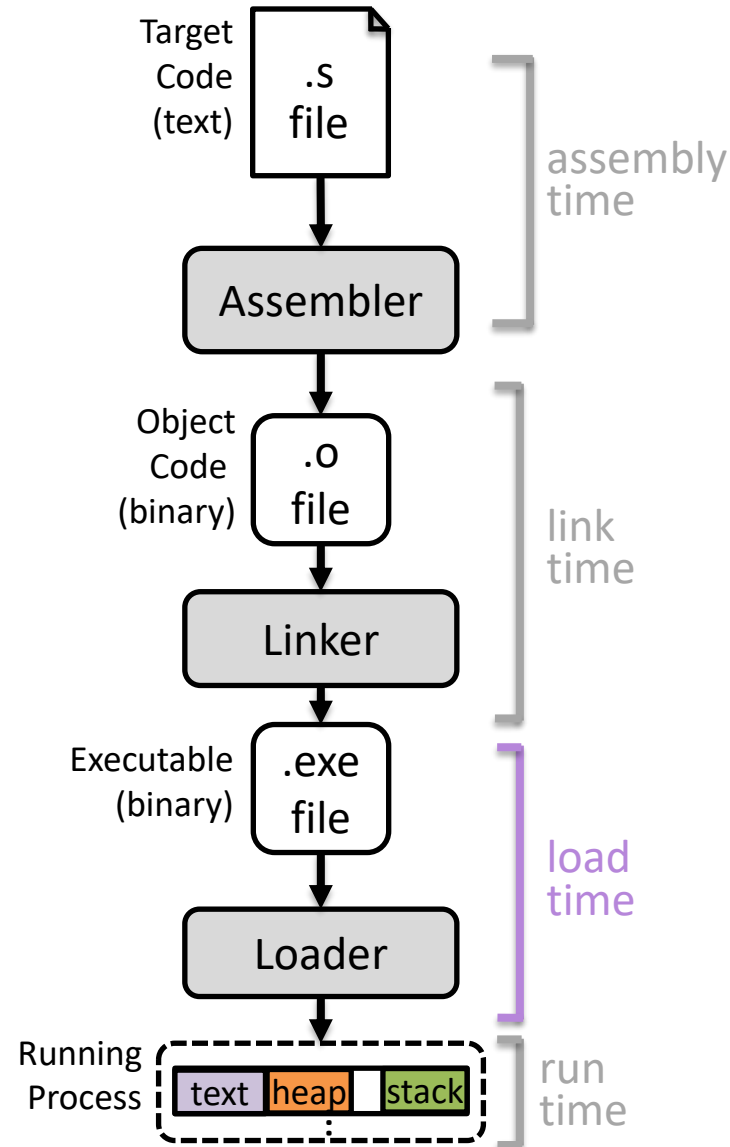
- Assembler
- Linker
- Loader



The Loader

Module Role and Operation

- Move code from secondary storage (disk) into primary storage (RAM)
- Resolves placeholders for dynamically-linked libraries (e.g. .so files)
- Kick off the program



The Loader

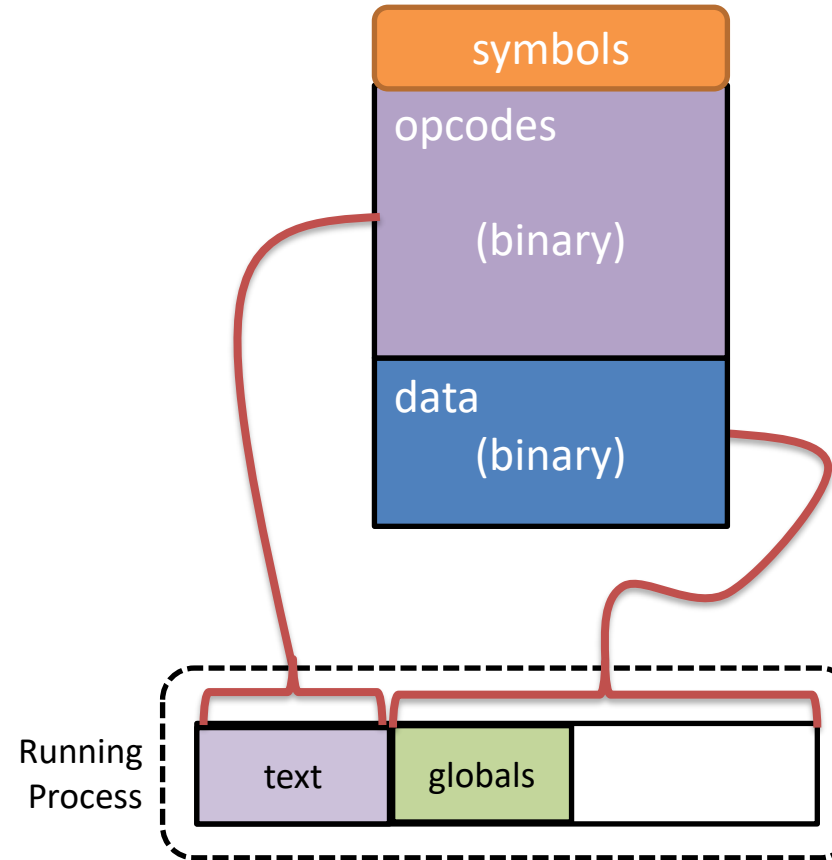
Module Role and Operation

Historically:

- Map program code into free memory
- May need to relocate symbols again

Modern Loaders:

- Simpler, as virtual memory handles physical address mapping
- Resolves dynamically-loaded libraries (.so, .dll)
- Part of the Operating System



That's all the Components!

Postcompilation: Underview

This basic process completes the “lifecycle” from source code to process

- From source to assembly
- From assembly to object code
- From object code to executable
- From executable to running process

Lecture Summary

Postcompilation: Underview

This Lecture

- Conceptual workflow from source text to process
- Tooling to perform this process
- Using cross-language linking to avoid re-implementation



*Project
concept*

Next up: Optimization!

- Improving on the naïve code generation we've performed
 - Better machine code
 - Better intermediate code