

Check-In

Review – x64 Memory

Write x64 code for the following snippet. Assume a, b, and c are global variables

```
if (a == 2) {  
    a = b - c;  
}
```

University of Kansas | Drew Davidson

ECCS 665
COMPILER
CONSTRUCTION

The title is presented in a multi-tiered, stylized font. The top tier features 'ECCS 665' in a pink, italicized, outlined font. The middle tier features 'COMPILER' in a large, bold, metallic 3D font with a blue and purple glow. The bottom tier features 'CONSTRUCTION' in a white, italicized, outlined font. The background is a dark blue space with stars and a glowing purple grid floor.

Practical x64

Previously...

Review – x64 Memory

.data
var-a: .quad 8

Memory Layout

- Static Allocation
- The heap and the stack

You should know

- What a static allocation scheme is (and its limitations)
- How to do static allocation in x64 assembly
- The concepts of the stack and heap



Architecture

Today's Lecture

Practical x64

Create more capable programs

- More (Linux) syscalls
- Library linking
- Library shims

Practicing basic x64 instruction use



Architecture

Syscalls

Practical x64 – More Capable Programs

You're already familiar with `sys_exit`

- Put 60 in `%rax`
- Put the value to return in `%rdi`

There's are many syscalls

- Officially documented in `syscall_64.tbl` in the [Linux source git](#)

They adhere to a protocol

- Lots of documents on the web
- I like [one by Ryan A. Chapman](#)



```
1 .text
2 .globl _start
3 _start:
4     movq $60, %rax
5     movq $4, %rdi
6     syscall
```

Syscall Reference

Practical x64 – More Capable Programs

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

%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
0	sys_read	unsigned int fd	char *buf	size_t count			
1	sys_write	unsigned int fd	const char *buf	size_t count			
2	sys_open	const char *filename	int flags	int mode			
3	sys_close	unsigned int fd					

⋮

60	sys_exit	int error_code					
----	----------	----------------	--	--	--	--	--

⋮

Syscall Example: Write to Console

Practical x64 – More Capable Programs

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

%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
0	sys_read	unsigned int fd	char *buf	size_t count			
1	sys_write	unsigned int fd	const char *buf	size_t count			
2	sys_open	const char *filename	int flags	int mode			
3	sys_close	unsigned int fd					

⋮

60	sys_exit	int error_code					
----	----------	----------------	--	--	--	--	--

⋮

Syscall Example: Write to Console

Practical x64 – More Capable Programs

%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
1	sys_write	unsigned int fd	const char *buf	size_t count			

fd 1 is stdout

```
1 .data
2 myStr: .asciz "hello\n"
3 .text
4 .globl _start
5 _start:
6     #write to console
7     movq $1, %rax      #Select sys_write
8     movq $1, %rdi      #Choose file descriptor 1
9     movq $myStr, %rsi  #Point to the string address
10    movq $6, %rdx      #Write 6 characters
11    syscall            #Invoke OS
12
13    #Exit
14    movq $60, %rax
15    movq $7, %rdi
16    syscall
```

Syscall Reference

Practical x64 – More Capable Programs

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

%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
0	sys_read	unsigned int fd	char *buf	size_t count			
1	sys_write	unsigned int fd	const char *buf	size_t count			
2	sys_open	const char *filename	int flags	int mode			
3	sys_close	unsigned int fd					

⋮

60	sys_exit	int error_code					
----	----------	----------------	--	--	--	--	--

⋮

Syscall Example: Write to File

Practical x64 – More Capable Programs

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

%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
0	sys_read	unsigned int fd	char *buf	size_t count			
1	sys_write	unsigned int fd	const char *buf	size_t count			
2	sys_open	const char *filename	int flags	int mode			
3	sys_close	unsigned int fd					

⋮

60	sys_exit	int error_code					
----	----------	----------------	--	--	--	--	--

⋮

Syscall Example: Write to File

Practical x64 – More Capable Programs

```
1 .data
2 myStr: .asciz "hello\n"
3 myFile: .asciz "newFile"
4 .text
5 .globl _start
6 _start:
7     movq $2, %rax           # select sys_open
8     movq $myFile, %rdi      # filename
9     movq $0101, %rsi        # flags
10    movq $0777, %rdx        # Access mode
11    syscall                 # Invoke OS
12    movq %rax, %r11         # retrieve new fd
13
14    movq $1, %rax           # select sys_write
15    movq %r11, %rdi          # file descriptor
16    movq $myStr, %rsi       # string to write
17    movq $6, %rdx           # length to write
18    syscall                 # Invoke OS
19
20    movq $3, %rax           # select sys_close
21    movq %r11, %rdi          # file descriptor
22    syscall                 # Invoke OS
23
24    #Exit
25    movq $60, %rax
26    movq $7, %rdi
27    syscall
```

O_CREAT | O_WRONLY

wrx for all

Don't Invoke Syscalls Directly!

Practical x64 – More Capable Programs

**Programmers are discouraged
from directly invoking syscalls**

- Prefer to interact with the OS through the language runtime library
- But we don't have that option...

OR DO WE?!?!?!?!?

(yes, we do)



Today's Lecture

Practical x64

Create more capable programs

- More (Linux) syscalls
- Library linking
- Library shims

Practicing basic x64 instruction use



Architecture

~~Why~~ Use a Standard Library?

How Practical x64 – More Capable Programs

Handle tedious, error-prone functionality

12

- Example: printf

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

- libc contains native string handling functions for ~~C~~ programs

our

x64

~~Why~~ Use a Standard Library?

How Practical x64 – More Capable Programs

Be compatible with the System V ABI

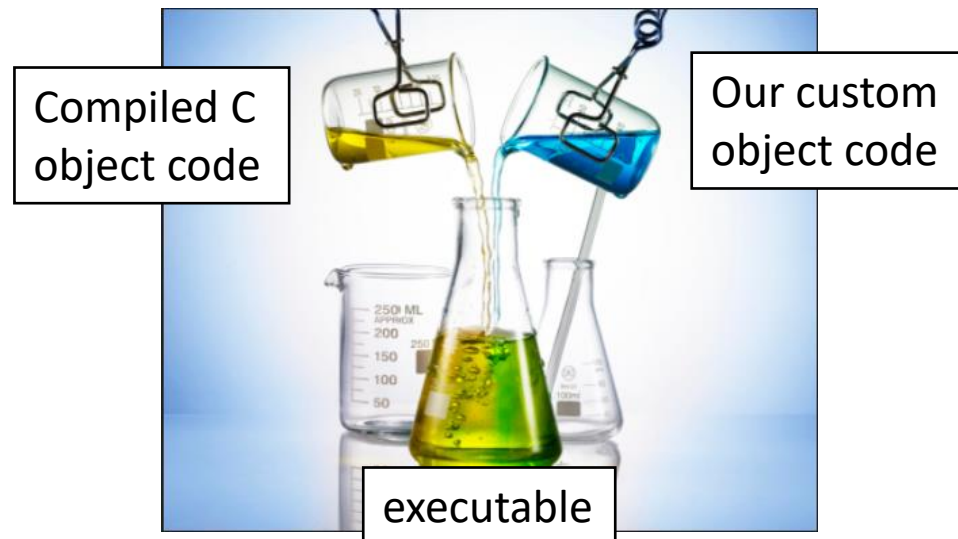
1. ~~“Masquerade” as a C program~~

- Follow the conventions of the libc native code:

Adhere to the *Application Binary Interface*

2. Combine object code we write with compiled C code

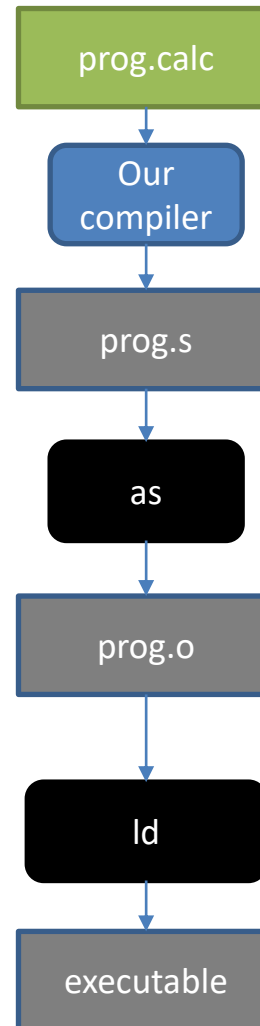
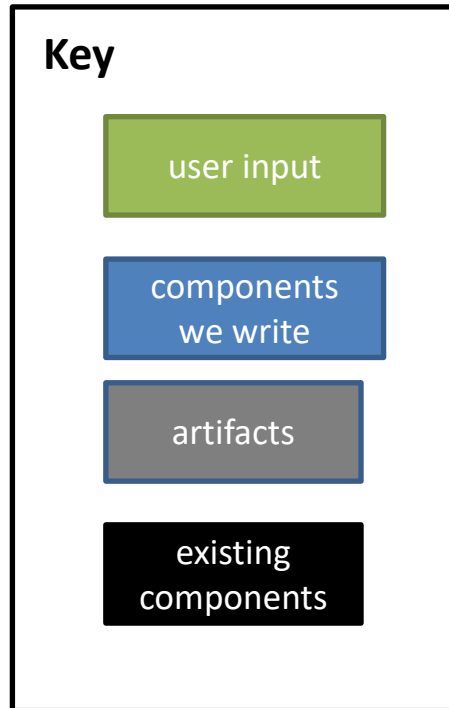
- Good news: linking is already a well-supported operation



Cross-language linking

Practical x64 – More Capable Programs

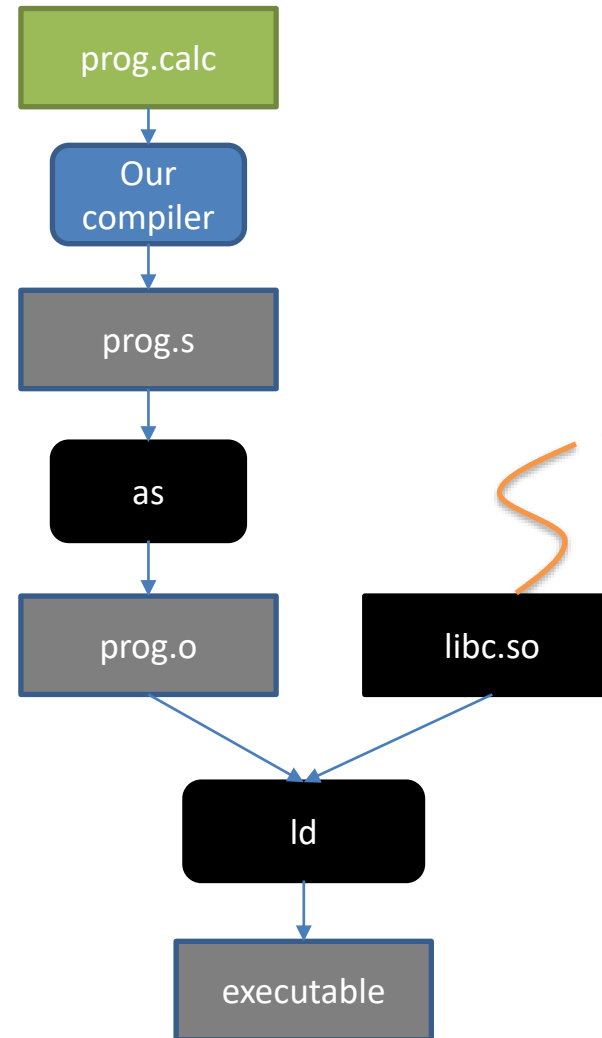
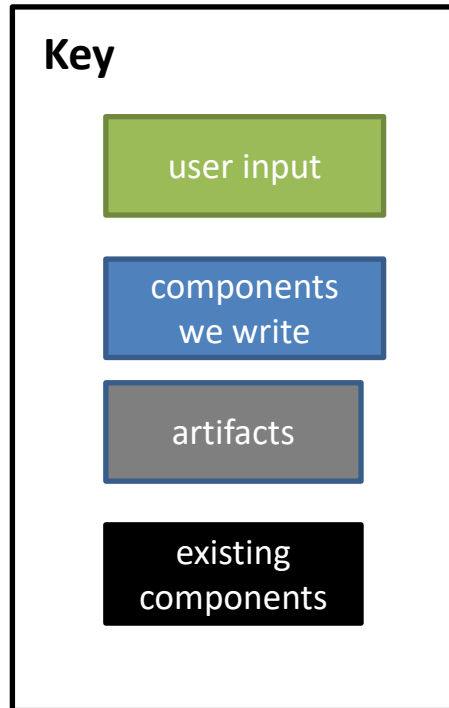
linking workflow



Cross-language linking

Practical x64 – More Capable Programs

linking workflow



prog.s
as prog.s -o prog.o
ld prog.o -o prog.exe

*Reuse existing native code
for common functionality
(i.e. string printing)*

Basing our runtime on C's

Practical x64 – More Complicated Programs

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 \

-lc \

link the libc library

prog.o \

\$SYSPATH/crtn.o \

-o prog.exe

release runtime data structures

entrypoint code

`_start: exit(main)`

init runtime data structures

Basing our runtime on C's

Practical x64 – More Complicated Programs

Standalone

Source code (prog.s)

```
1 .text
2 .globl _start
3 _start:
4     movq $60, %rax
5     movq $6, %rdi
6     syscall
```



Assembler command

```
as prog.s -o prog.o
```



Linker command

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

Using libc



Clever C code disguise

Basing our runtime on C's

Practical x64 – More Complicated Programs

Standalone

Source code (prog.s)

```
1 .text
2 .globl _start
3 _start:
4     movq $60, %rax
5     movq $6, %rdi
6     syscall
```

Assembler command

```
as prog.s -o prog.o
```

Linker command

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

Using libc

Source code (prog.s)

```
1 .text
2 .globl main
3 main:
4     retq
```

* caveat: we'll create a more elaborate main function in later lectures

Assembler command

```
as prog.s -o prog.o
```

Linker command

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

```
ld -dynamic-linker /lib64/ld-linux-x86-64.so.2
$SYSPATH/crt1.o $SYSPATH/crti.o -lc prog.o
$SYSPATH/crtn.o -o prog.exe
```

Calling a C library function

Practical x64 – More Complicated Programs

function

putchar

```
int putchar ( int character );
```

Write character to stdout

Writes a *character* to the *standard output* (*stdout*).

It is equivalent to calling `putc` with `stdout` as second argument.

Parameters

character

The int promotion of the character to be written.

The value is internally converted to an unsigned char when written.

Some System V ABI Facts:

- First argument is in %rdi
- Second argument is in %rsi
- Return value is in %rax

Calling a C library function

Practical x64 – More Complicated Programs

Assembly code

function
putchar
`int putchar ( int character );`
Write character to stdout
Writes a *character* to the *standard output* (stdout).
It is equivalent to calling `putc` with `stdout` as second argument.

 **Parameters**

character
The int promotion of the character to be written.
The value is internally converted to an unsigned char when written.

```
1 .text
2 .globl main
3 main:
4     movq $97, %rdi
5     callq putchar
6
7     movq $10, %rdi
8     callq putchar
9
10    retq
```

ASCII code for a

ASCII code for \n

```
as prog.s -o prog.o
```

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

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

```
$SYSPATH/crt1.o $SYSPATH/crti.o -lc prog.o
```

```
$SYSPATH/crtn.o -o prog.exe
```

output

a(newline)

Today's Lecture

Practical x64

Create more capable programs

- More (Linux) syscalls
- Library linking
- Library shims

Practicing basic x64 instruction use



Architecture

Shimming the C library

Practical x64 – More Complicated Programs

How would we print an int?

- Sure would love to call printf!
- Calling printf seems... complicated

```
1 .text
2 .globl main
3 main:
4     movq $90, %rdi
5     addq $7, %rdi
6     callq putchar
7
8     retq
```

output

a

Shimming the C library

Practical x64 – More Complicated Programs

How would we print an int?

- Sure would love to call printf!
- Calling printf seems... complicated
- Maybe we could create a simpler interface to printf?

function

printf

<stdio>

```
int printf ( const char * format, ... );
```

Print formatted data to stdout

Writes the C string pointed by *format* to the standard output (*stdout*). If *format* includes *format specifiers* (subsequences beginning with %), the additional arguments following *format* are formatted and inserted in the resulting string replacing their respective specifiers.

Shimming the C library

Practical x64 – More Complicated Programs

How would we print an int?

- Sure would love to call printf!
- Calling printf seems... complicated
- Maybe we could create a simpler interface to printf?

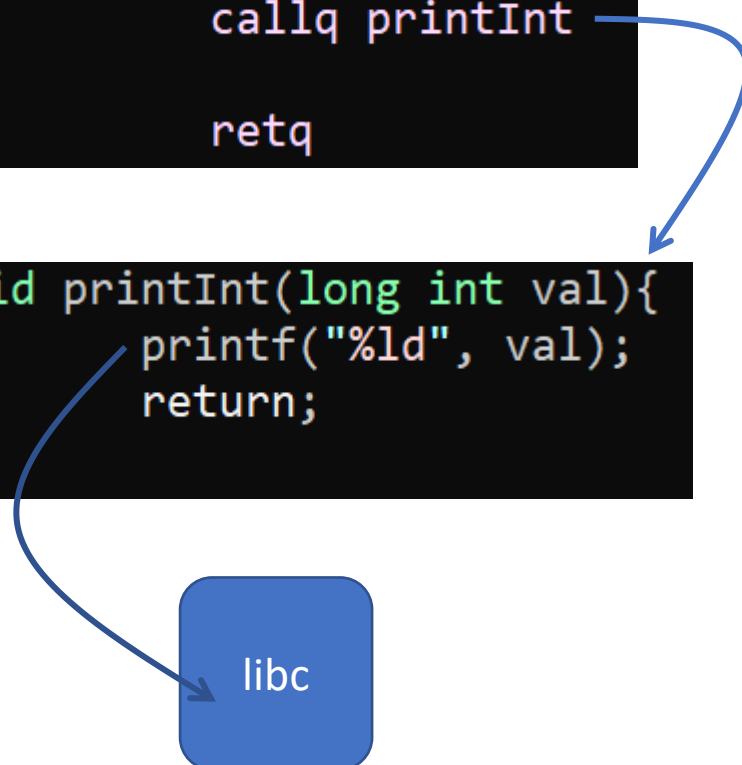
prog.s

```
1 .text
2 .globl main
3 main:
4     movq $90, %rdi
5     addq $7, %rdi
6     callq printInt
7
8     retq
```

shim.c

```
1 void printInt(long int val){
2     printf("%ld", val);
3     return;
4 }
```

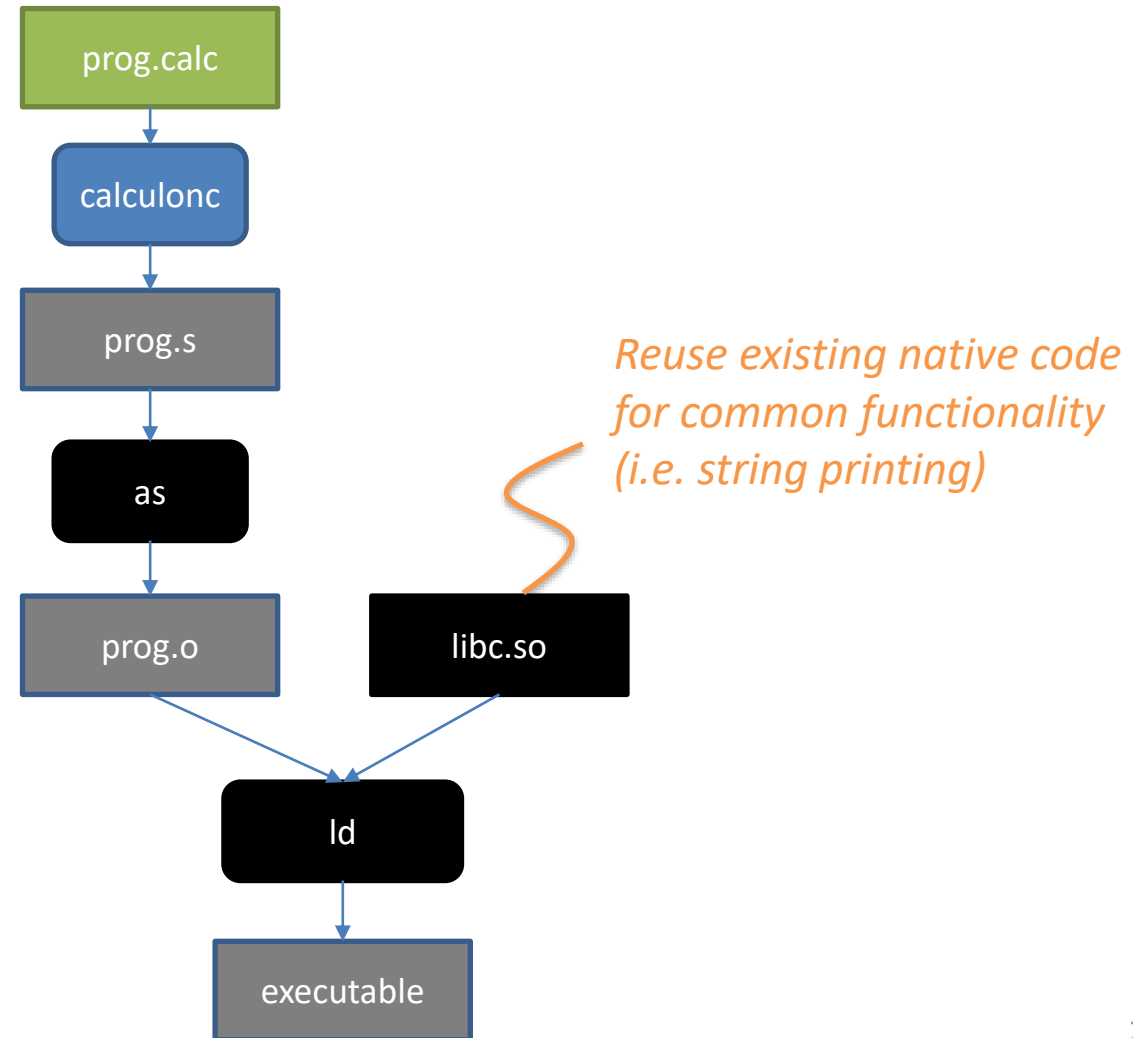
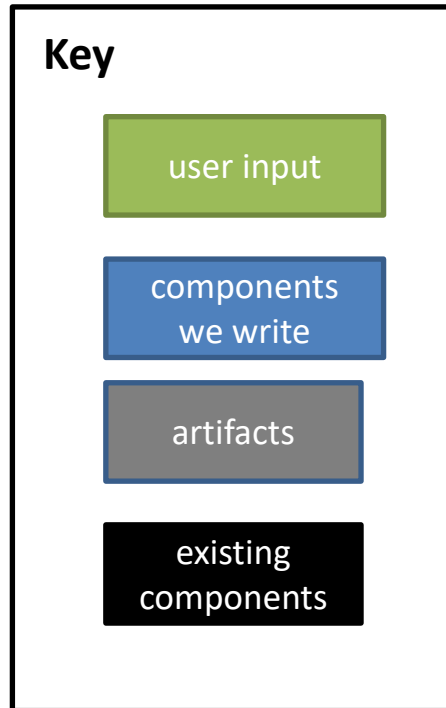
libc



Cross-language linking

Practical x64 – More Complicated Programs

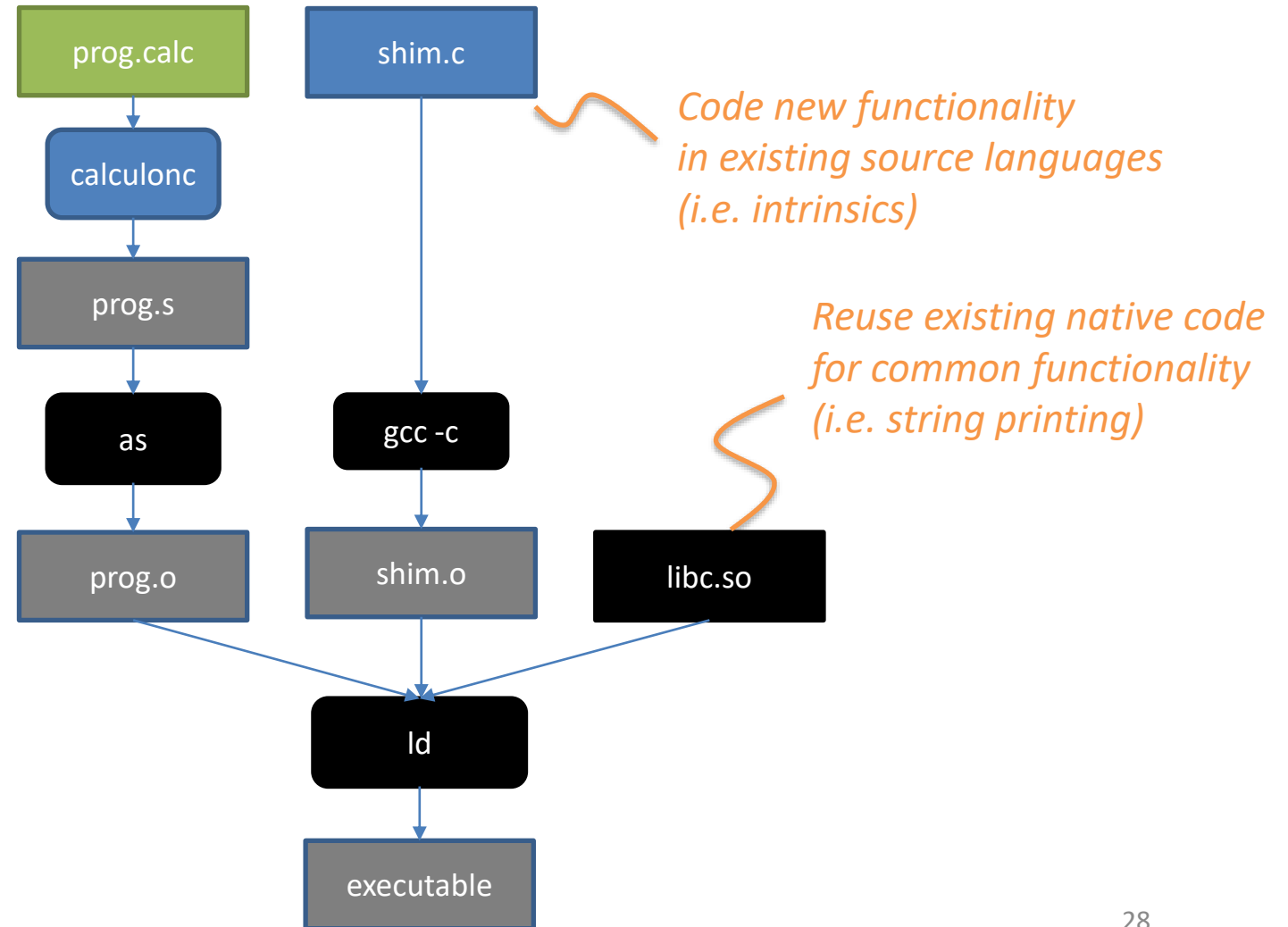
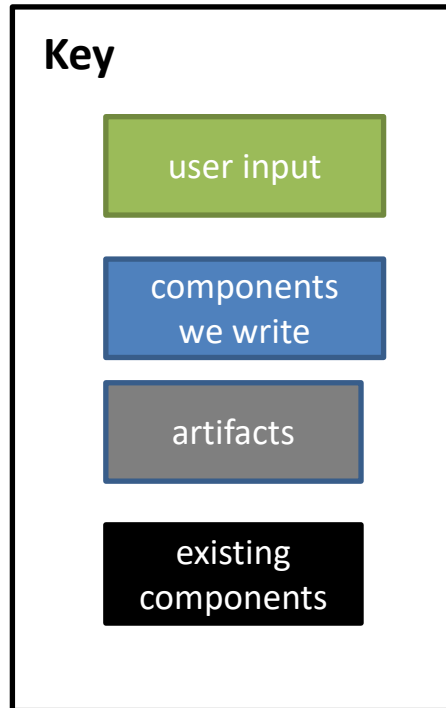
linking workflow



Cross-language linking

Practical x64 – More Complicated Programs

linking workflow



Today's Lecture

Practical x64

Create more capable programs

- More (Linux) syscalls
- Library linking
- Library shims

Practicing basic x64 instruction use



Architecture

Using cmpq and set<CC>

Practical x64 – Basic instruction use

Translating `a = VAR2 <= VAR1` into code

```
.data
VAR1: .quad 20
VAR2: .quad 10

.globl _start
.text
_start:
    movq $0, %rdi        #Prepare rdi to receive comparison result
    movq (VAR1), %rax     #Load var1 value into %rax
    movq (VAR2), %rbx     #Load var2 value into %rbx
    cmpq %rax, %rbx       #Actually do the comparison
    setle %dil            #Set the lowest byte of %rdi to 1 if rbx <= rax

    movq $60, %rax        # Select the exit syscall
    nop                  # %rdi is already set as the return value
    syscall              # Invoke the selected syscall
```

idivq semantics

$\%rax = \%rdx:\%rax / o_1$

$\%rdx = \%rdx:\%rax \% o_1$

Using idivq and cqto

x64 Practice – Basic instruction use

convert quadword to octoword

Naïve Division (a case that happens to work out)

```
movq $6, %rax    #lower 64 bits of number
movq $0, %rdx    #upper 64 bits of number
movq $2, %r8     #denominator
idivq %r8        #Do the division
```

$\%rdx: 0$

0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
------	------	------	------	------	------	------	------

octoword

$\%rax: 6$

0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x06
------	------	------	------	------	------	------	------

rdx:rax 6

$\%r8: 2$

0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x02
------	------	------	------	------	------	------	------

quotient 3

Naïve Division (a case that doesn't work out)

```
movq $-2, %rax   #lower 64 bits of number
movq $0, %rdx    #upper 64 bits of number
movq $2, %r8     #denominator
idivq %r8        #Do the division
```

$\%rdx: 0$

0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
------	------	------	------	------	------	------	------

octoword

$\%rax: -2$

0xFF	0xFF	0xFF	0xFF	0xFF	0xFF	0xFF	0xFE
------	------	------	------	------	------	------	------

rdx:rax ~~-2~~

$\%r8: 2$

0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x02
------	------	------	------	------	------	------	------

quotient -1

18446744073709551614

9223372036854775808

```
movq $-2, %rax   #lower 64 bits of number
cqto             #upper 64 bits of number
movq $2, %r8     #denominator
idivq %r8        #Do the division
```

$\%rdx: -1$

0xFF	0xFF	0xFF	0xFF	0xFF	0xFF	0xFF	0xFF
------	------	------	------	------	------	------	------

octoword

$\%rax: -2$

0xFF	0xFF	0xFF	0xFF	0xFF	0xFF	0xFF	0xFE
------	------	------	------	------	------	------	------

rdx:rax -2

$\%r8: 2$

0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x02
------	------	------	------	------	------	------	------

quotient -1