University of Kansas | Drew Davidson

EECS 665 COMPILER CONSTRUCTION

Practical x64

Previously…

Review – x64 Memory

Memory Layout

- Static Allocation
- The heap and the stack

<u>You should know</u>

- 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]


```
.text
.globl _start
_start:
        movq $60, %rax
        movq $4, %rdi
        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

```
.data
myStr: .asciz "hello\n"
.text
.globl _start
_start:
        #write to console
        movq $1, %rax           #Select sys_write
        movq $1, %rdi           #Choose file descriptor 1
        movq $myStr, %rsi       #Point to the string address
        movq $6, %rdx           #Write 6 characters
        syscall                 #Invoke OS

        #Exit
        movq $60, %rax
        movq $7, %rdi
        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

```
.data
myStr: .asciz "hello\n"
myFile: .asciz "newFile"
.text
.globl _start
_start:
        movq $2, %rax           # select sys_open
        movq $myFile, %rdi      # filename
        movq $0101, %rsi        # flags
        movq $0777, %rdx        # Access mode
        syscall                 # Invoke OS
        movq %rax, %r11         # retrieve new fd

        movq $1, %rax           # select sys_write
        movq %r11, %rdi         # file descriptor
        movq $myStr, %rsi       # string to write
        movq $6, %rdx           # length to write
        syscall                 # Invoke OS

        movq $3, %rax           # select sys_close
        movq %r11, %rdi         # file descriptor
        syscall                 # Invoke OS

        #Exit
        movq $60, %rax
        movq $7, %rdi
        syscall
```

O_CREAT | O_WRONLY

wrx for all

Don't Invoke Syscalls Directly!

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~~ How Use a Standard Library?

Practical x64 – More Capable Programs

Handle tedious, error-prone functionality

- Example: printf

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

- libc contains native x64 string handling functions for ~~C~~ our programs

~~Why~~ How Use a Standard Library?

Practical x64 – More Capable Programs

Be compatible with the System V ABI

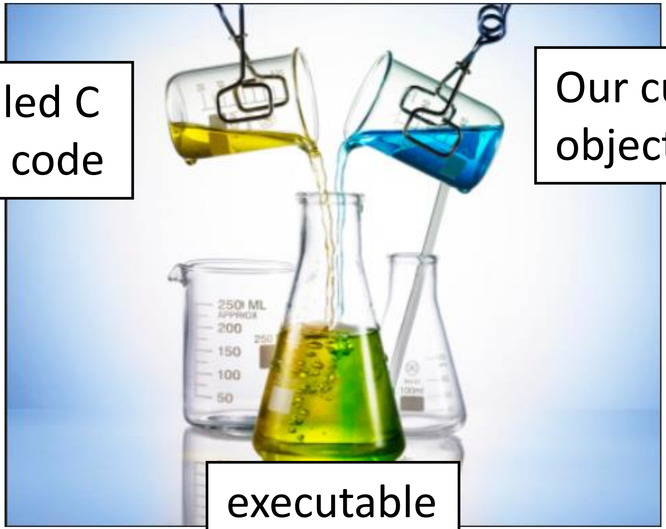
1. ~~"Masquerade" as a C program~~

- Follow the conventions of the libc native code:

  Adhere to the *Application Binary Interface*

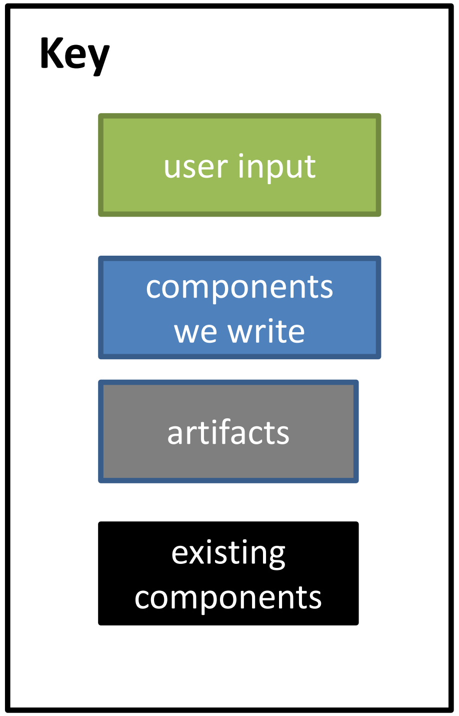
2. Combine object code we write with compiled C code

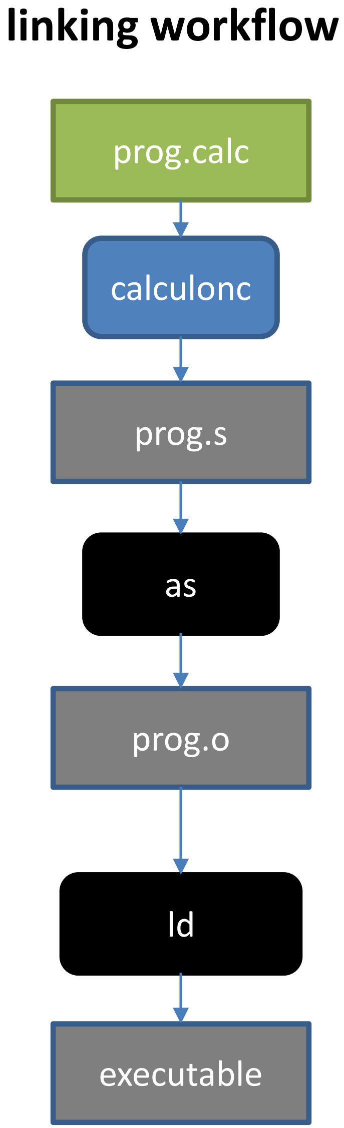
- Good news: linking is already a well-supported operation



Cross-language linking

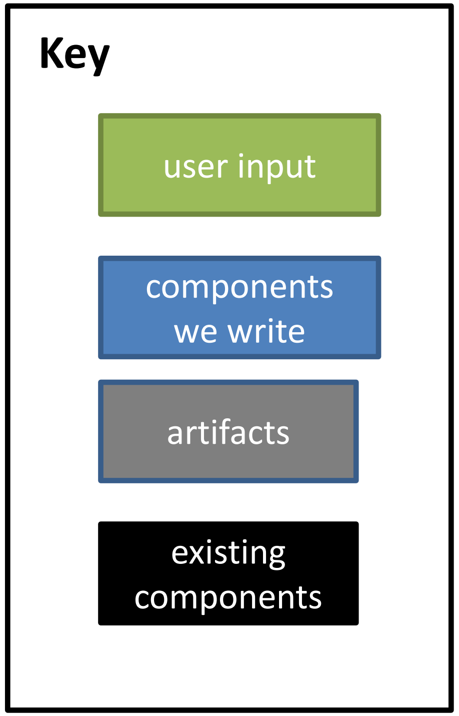
Practical x64 – More Capable Programs



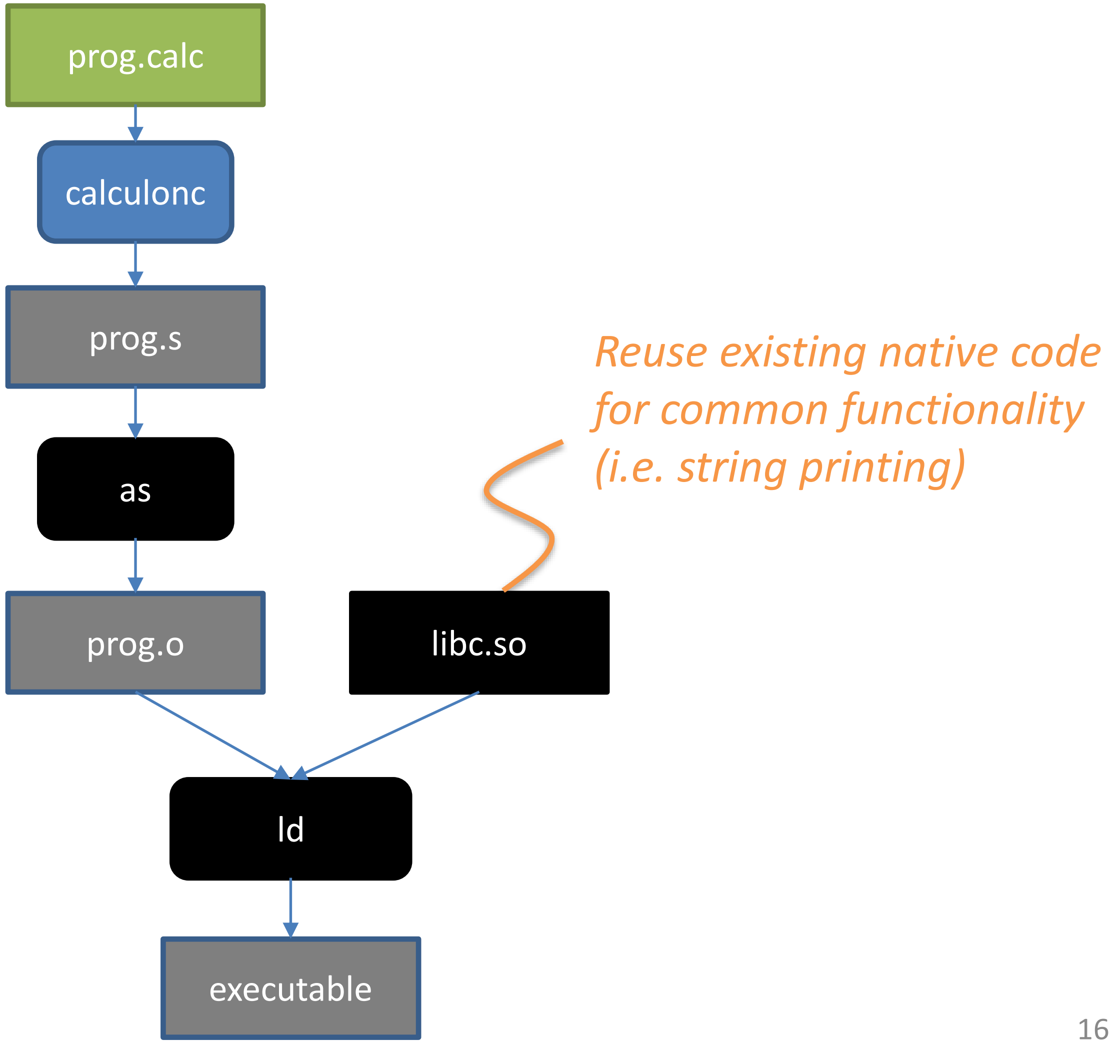


Cross-language linking

Practical x64 – More Capable Programs



linking workflow



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

entrypoint code

`_start: exit(main)`

init runtime data structures

link the libc library

release runtime data structures

Basing our runtime on C's

Practical x64 – More Complicated Programs

Standalone

Source code (prog.s)

```
.text
.globl _start
_start:
        movq $60, %rax
        movq $6, %rdi
        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)

```
.text
.globl _start
_start:
        movq $60, %rax
        movq $6, %rdi
        syscall
```

Assembler command

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

Linker command

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

Using libc

Source code (prog.s)

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

```
.text
.globl main
main:
        retq
```

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

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.

Assembly code

ascii code for a

```
.text
.globl main
main:
        movq $97, %rdi
        callq putchar

        movq $10, %rdi
        callq putchar

        retq
```

\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

```
.text
.globl main
main:
        movq $90, %rdi
        addq $7, %rdi
        callq putchar

        retq
```

output

a

Shimming the C library

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

<cstdio>

```
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?

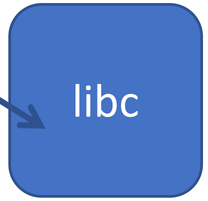
prog.s

```
.text
.globl main
main:
        movq $90, %rdi
        addq $7, %rdi
        callq printInt

        retq
```

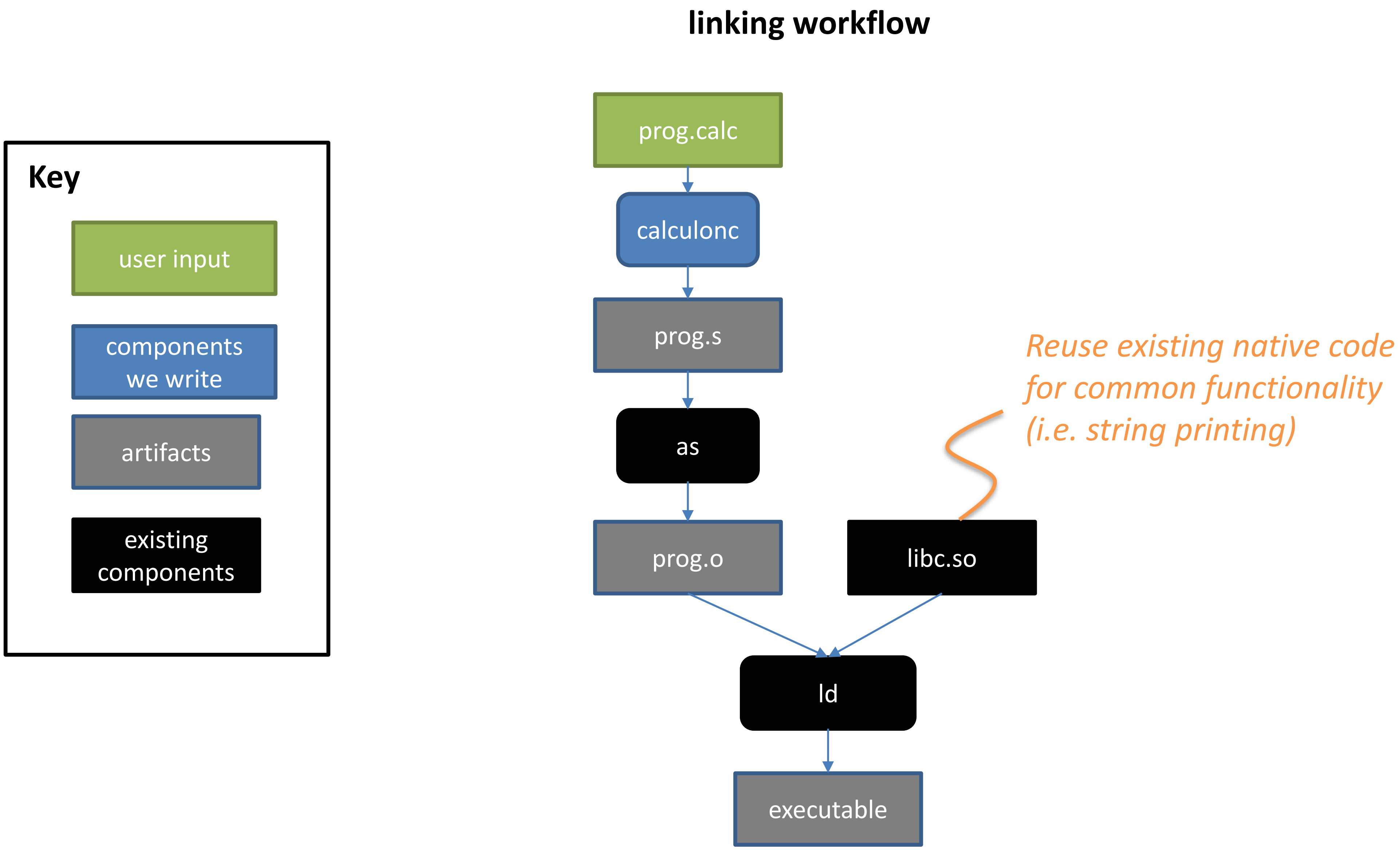
shim.c

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



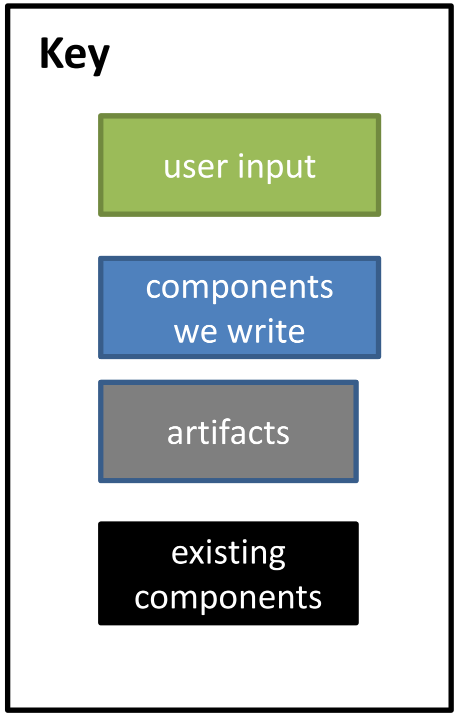
Cross-language linking

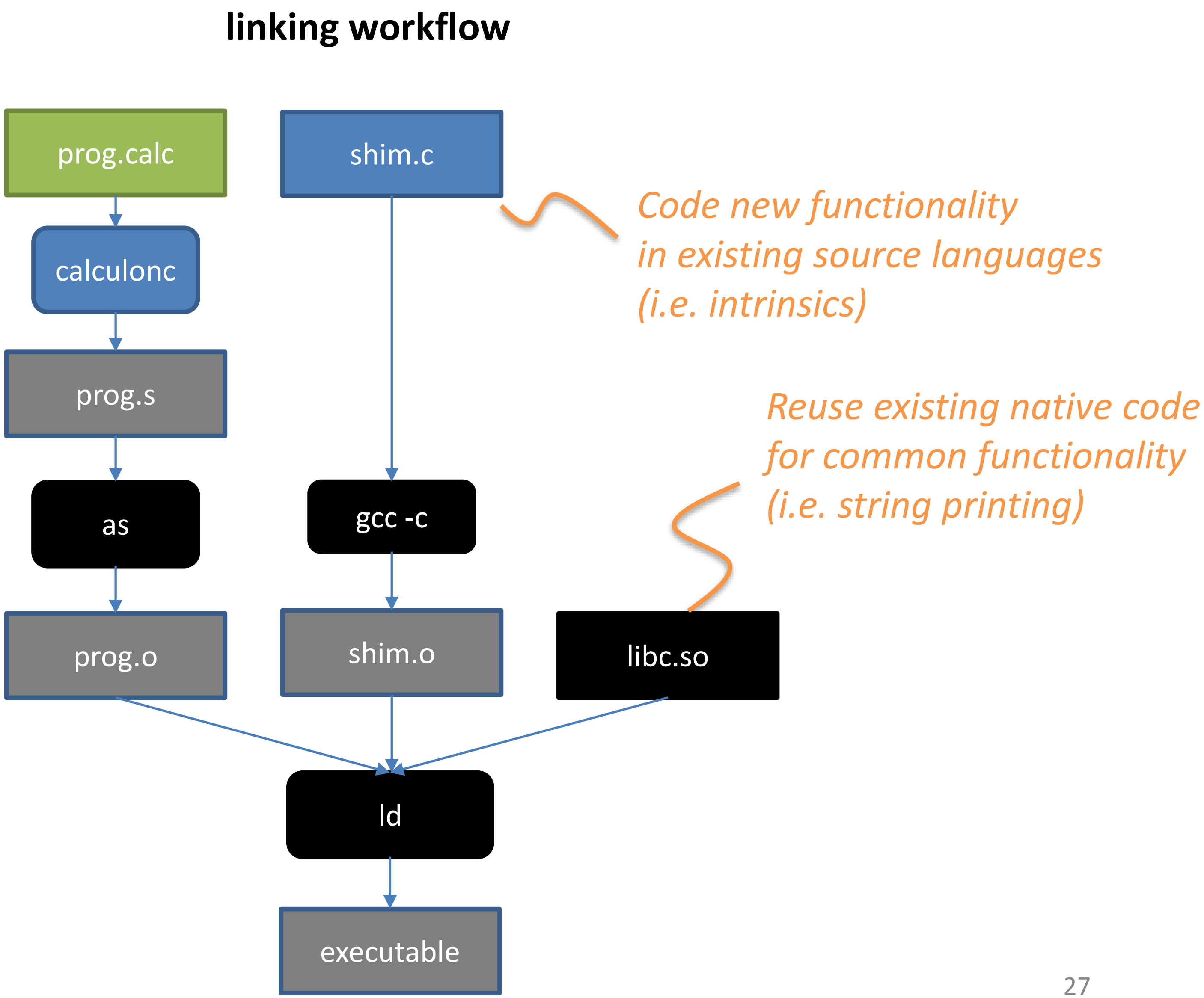
Practical x64 – More Complicated Programs



Cross-language linking

Practical x64 – More Complicated Programs





Today's Lecture

Practical x64

Create more capable programs

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

Practicing basic x64 instruction use

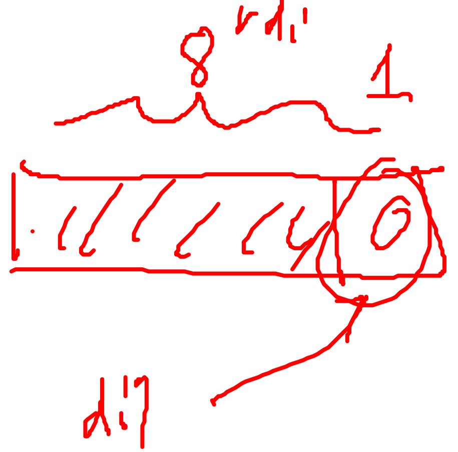
Architecture

Using cmpq and set<CC>

Practical x64 – Basic instruction use

setle

setg




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

.globl _start
.text
_start:
        movq $0, %rdi           #Prepare rcx to receive the comparison result
        movq (VAR1), %rax       #Load Var1 value into %rax
        movq (VAR2), %rbx       #Load Var2 value int %rbx
        cmpq %rax, %rbx         # Actually do the comparison
        setle %dil              # Set the lowest byte of %rdi to 1 if rbx <= rax
                                # AKA VAR2 <= VAR1

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

20

10

$op_2 \; ? \; op_1 \rightarrow$ rflags

$op_2 \le op_1 \rightarrow 0$

if (a < b && c < d){

}

$\Downarrow$

[tmp1] := [a] LT_{64} [b]

[tmp2] := [c] LT_{64} [d]

[tmp3] := [tmp1] AND_{64} [tmp2]

if z [tmp3] goto LBL

Using idivq and cqto

x64 Practice – Basic instruction use

to

convert

quadword

octoword

rdx : rax

8 8

r8

num

```
movq $6, %rax   #lower 64 bits of divisor
movq $0, %rdx    #upper 64 bits of divisor
movq $2, %r8     # The denominator
idivq %r8        # Do the deed
movq %rdx, %r13 # move remainder into r13
```

r13 = 3

-2

rdx

rax

```
movq $-2, %rax   #lower 64 bits of divisor
movq $0, %rdx    #upper 64 bits of divisor
movq $2, %r8     # The denominator
idivq %r8        # Do the deed
movq %rdx, %r13 # move remainder into r13
```

```
movq $-2, %rax   #lower 64 bits of divisor
cqto             #upper 64 bits of divisor
movq $2, %r8     # The denominator
idivq %r8        # Do the deed
movq %rdx, %r13 # move remainder into r13
```