Poll Everywhere

pollev.com/cse333

Which concept did you find the most difficult in the context of HW1 (so far if not completed)?

A. Pointers
B. Output parameters
C. Dynamic memory allocation
D. Structs
E. GDB
F. Style considerations
G. Prefer not to say

System Call Details, Makefiles

CSE 333 Autumn 2025

Instructors: Naomi Alterman, Chris Thachuk

Teaching Assistants:

Ann Baturytski	Derek de Leuw	Blake Diaz
Rishabh Jain	Chendur Jel Jayavelu	Lucas Kwan
Irene Xin Jie Lau	Nathan Li	Maya Odenheim
Advay Patil	Selim Saridede	Deeksha Vatwani
Angela Wu	Jiexiao Xu	

Relevant Course Information

- Homework 1 was due Thursday (10/9)
  - Still possible to submit late (until Sunday @ 11:59)
- Homework 2 rolling out soon
- Exercise 7 due Monday (10/13)
  - Out this afternoon; practice with POSIX
- Start using C++ on Monday!
  - Many conveniences
  - Many additional concerns (much larger language)

Lecture Outline

- **System Calls (More Detailed View)**
- Make and Build Tools
- Makefile Basics
- C History (for reading, not covered in lecture)

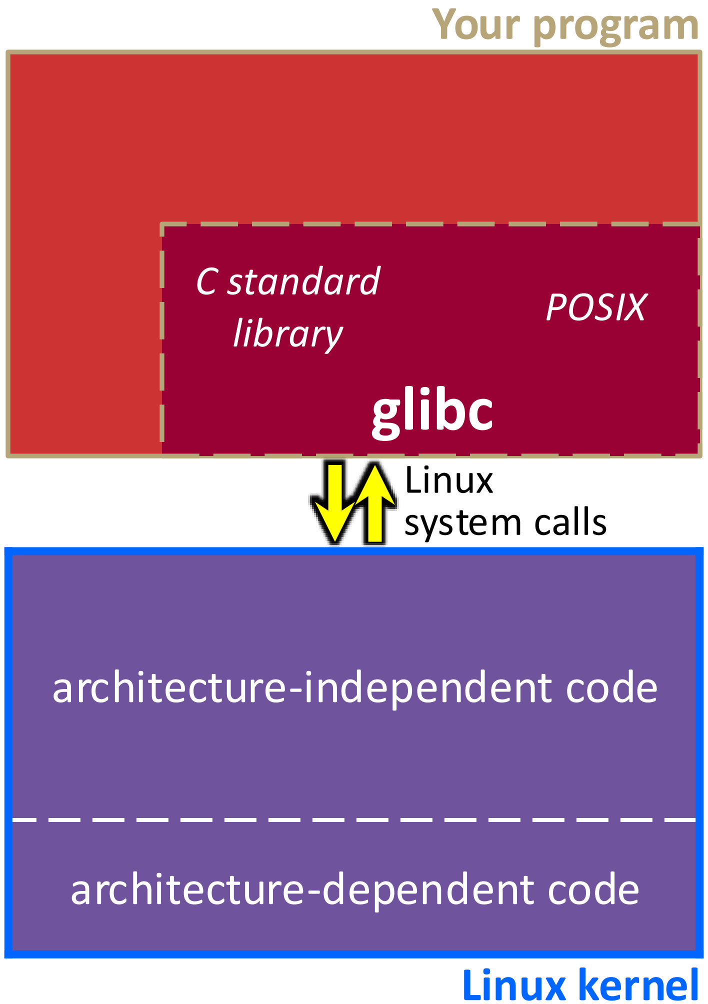
System Call Analogy

- The OS is a bank manager overseeing safety deposit boxes in the vault
  - Is the only one allowed in the vault and has the keys to the safety deposit boxes

- If a client wants to access a deposit box (*i.e.*, add or remove items), they must request that the bank manager do it for them
  - Takes time to locate and travel to box and find the right key
  - Client must wait in the lobby while the bank manager accesses the box – prevents messing with requested box or other boxes
  - Takes time to put box away, return from vault, and let client know that request was fulfilled

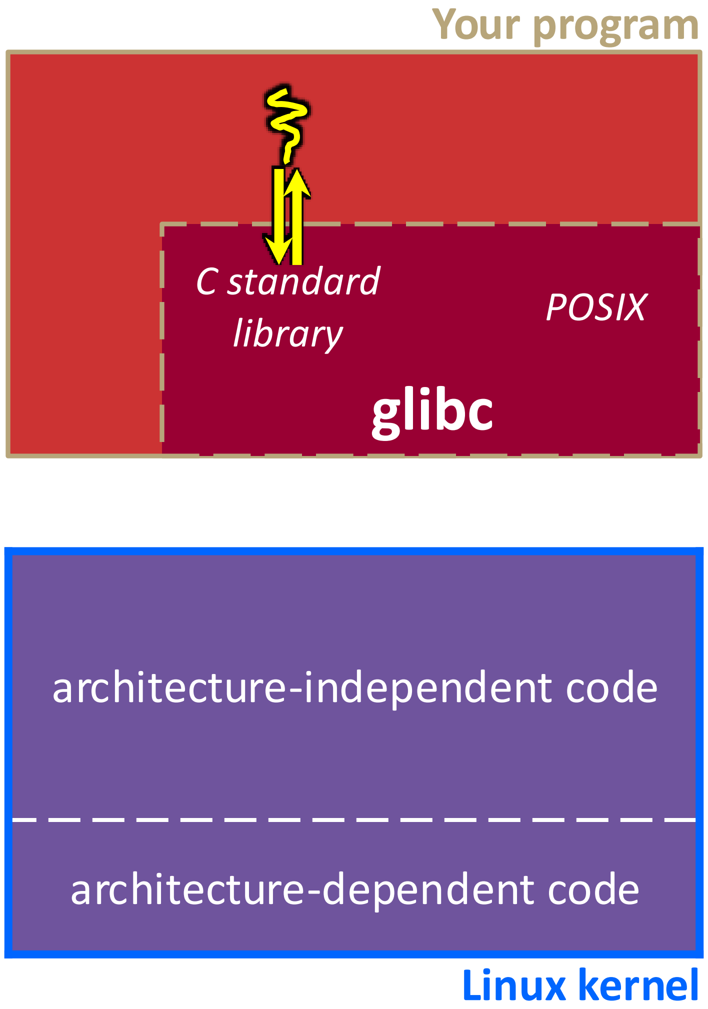
Details on x86/Linux

- A more accurate picture:
  - Consider a typical Linux process
  - Its thread of execution can be in one of several places:
    - In your program's code
    - In `glibc`, a shared library containing the C standard library, POSIX, support, and more
    - In the Linux architecture-independent code
    - In Linux x86-64 code



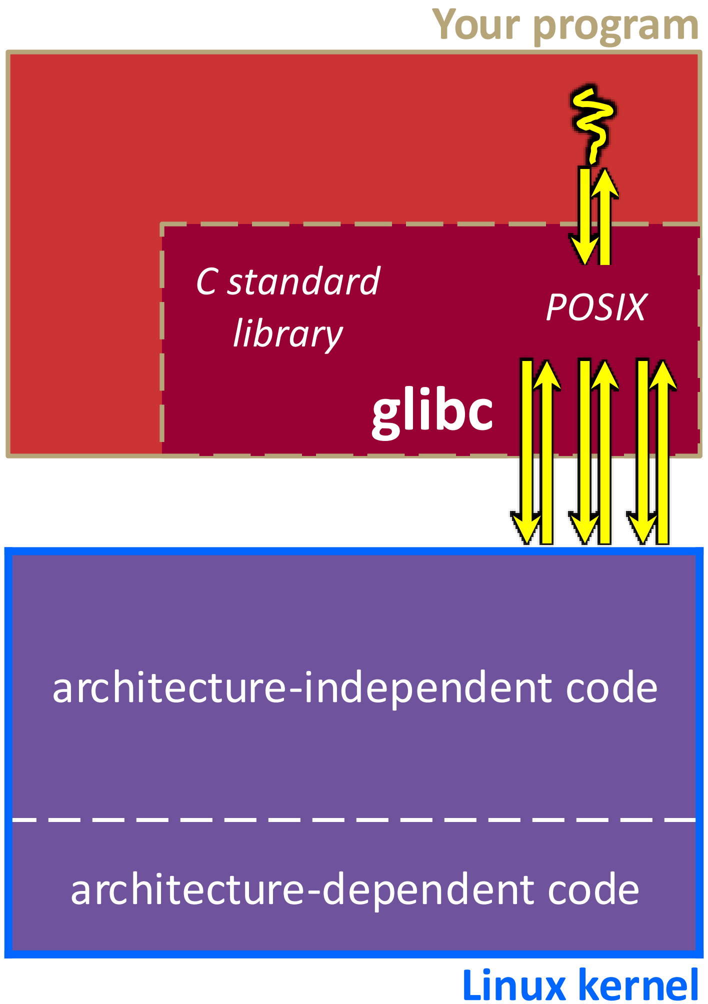
Details on x86/Linux

- Some routines your program invokes may be entirely handled by `glibc` without involving the kernel
  - *e.g.* `strcmp()` from `stdio.h`
  - There is some initial overhead when invoking functions in dynamically linked libraries (during loading)
    - But after symbols are resolved, invoking `glibc` routines is basically as fast as a function call within your program itself!



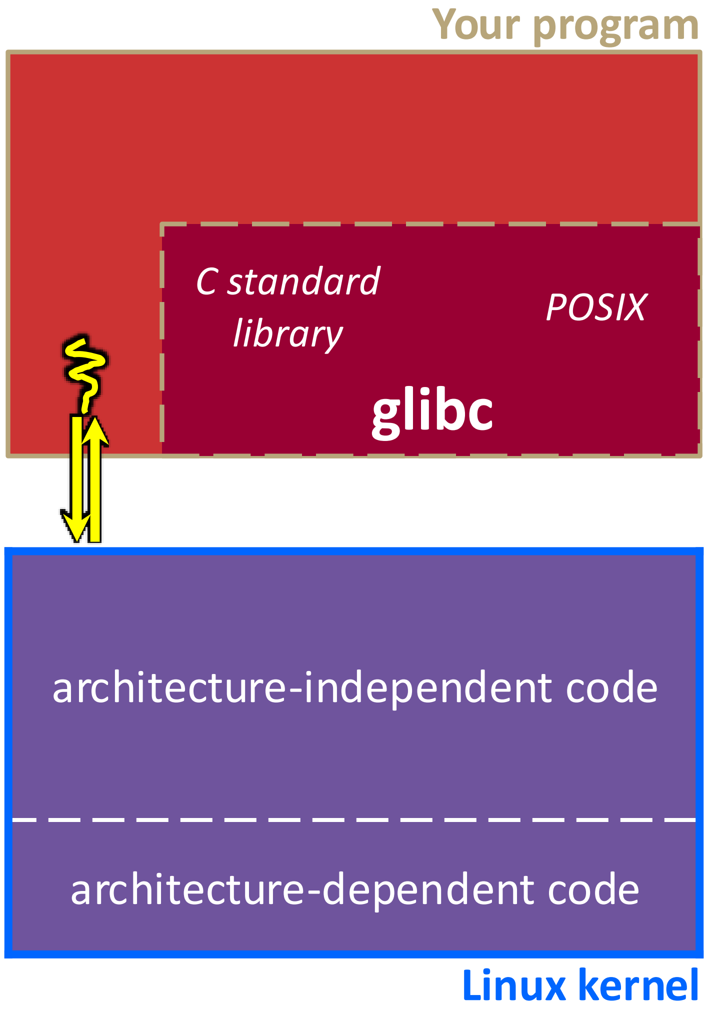
Details on x86/Linux

- Some routines may be handled by `glibc`, but they in turn invoke Linux system calls
  - *e.g.* POSIX wrappers around Linux `syscalls`
    - POSIX `readdir()` invokes the underlying Linux `readdir()`
  - *e.g.* C `stdio` functions that read and write from files
    - `fopen()`, `fclose()`, `fprintf()` invoke underlying Linux `open()`, `close()`, `write()`, etc.



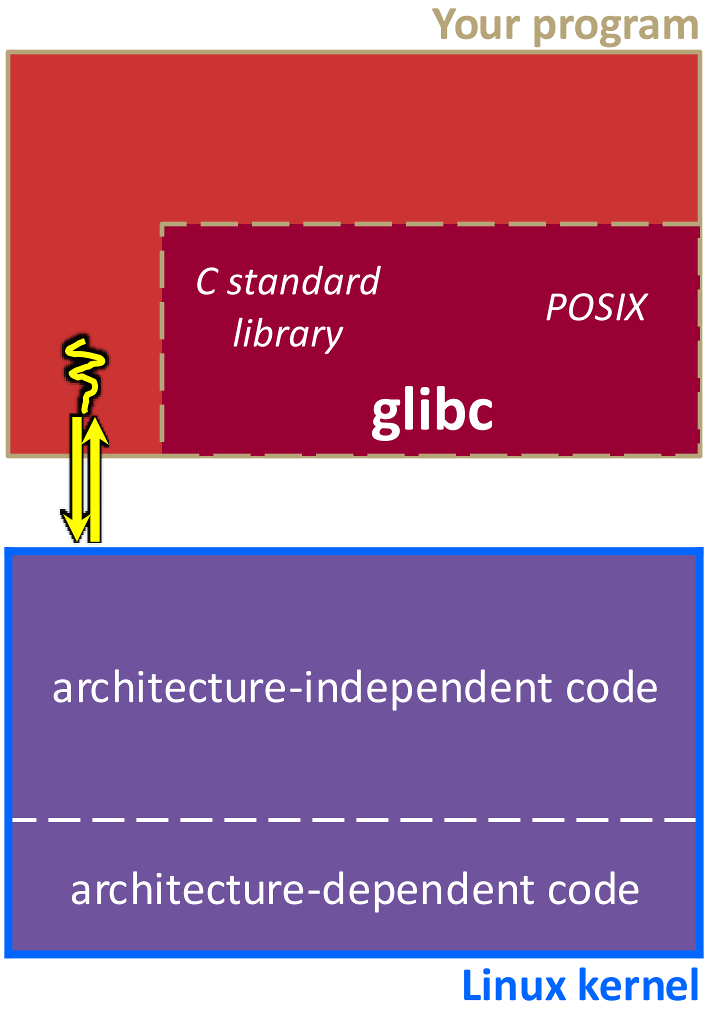
Details on x86/Linux

- Your program can choose to directly invoke Linux system calls as well
  - Nothing is forcing you to link with `glibc` and use it
  - But relying on directly-invoked Linux system calls may make your program less portable across UNIX varieties
    - (And won't be portable to non-Unix systems like Windows that run standard C on top of their own, different syscalls)



Details on x86/Linux

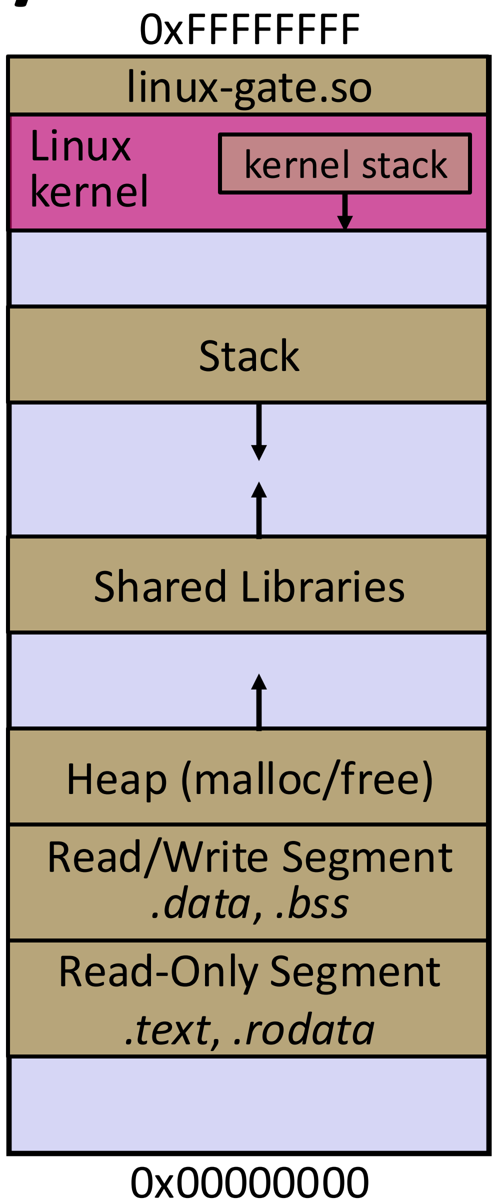
- Let's walk through how a Linux system call actually works
  - We'll assume *32-bit x86* using the modern `SYSENTER` / `SYSEXIT` x86 instructions
    - x86-64 code is similar, though details always change over time, so take this as an example – not a debugging guide

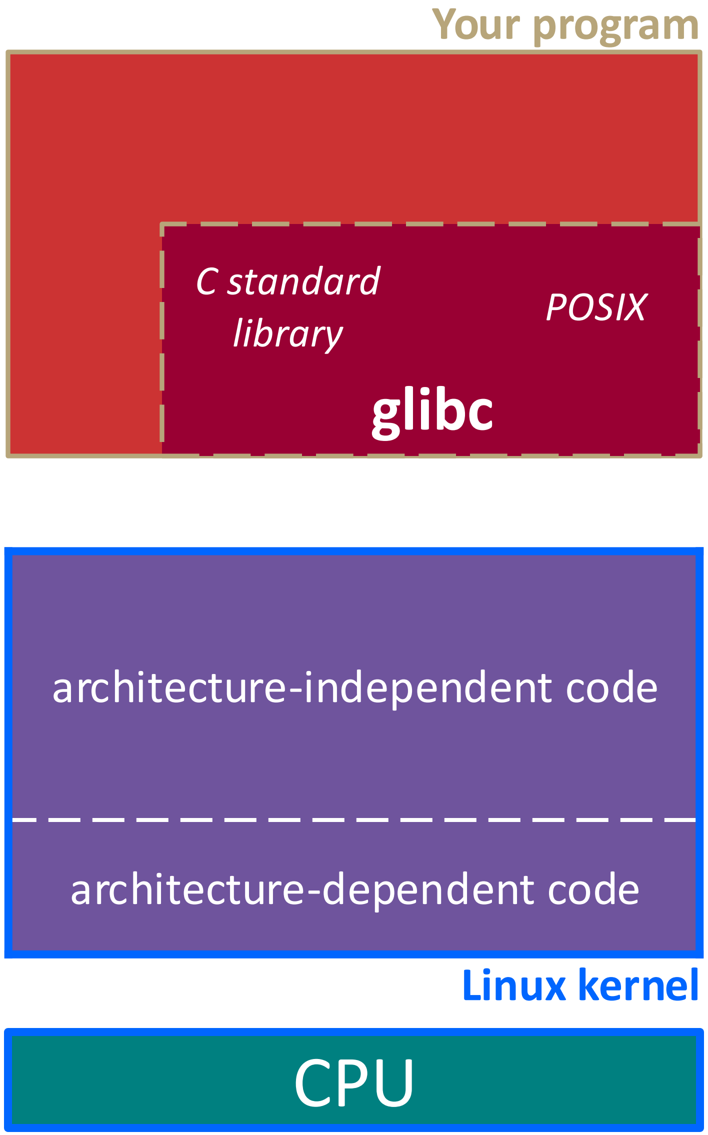


Details on x86/Linux

Remember our process address space picture?

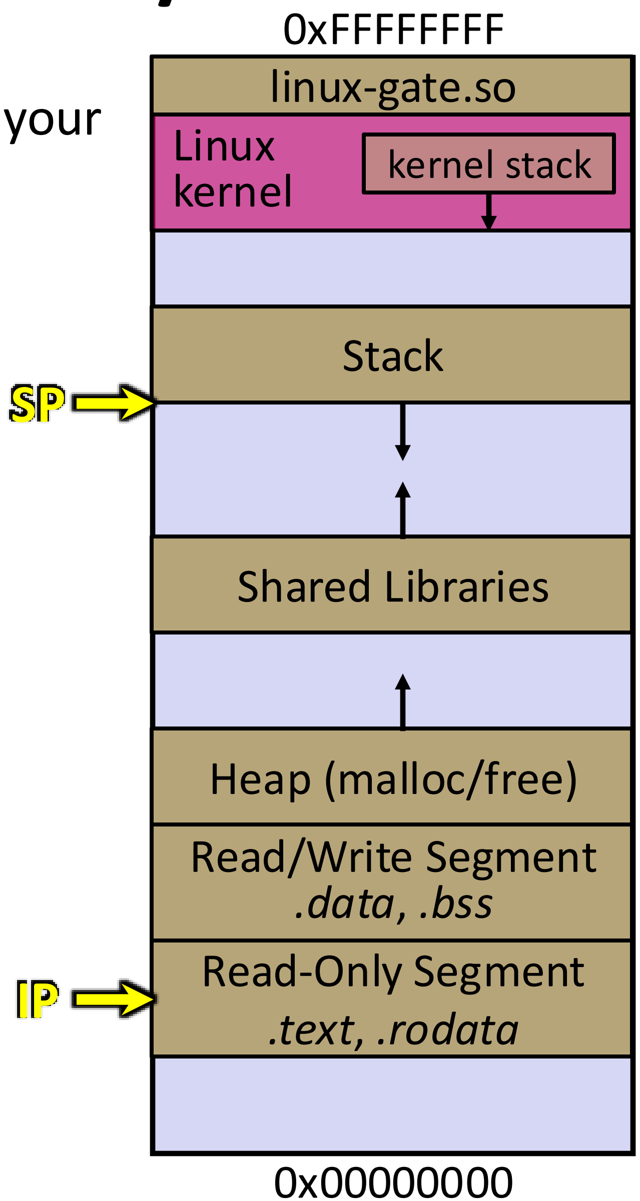
- Let's add some details:

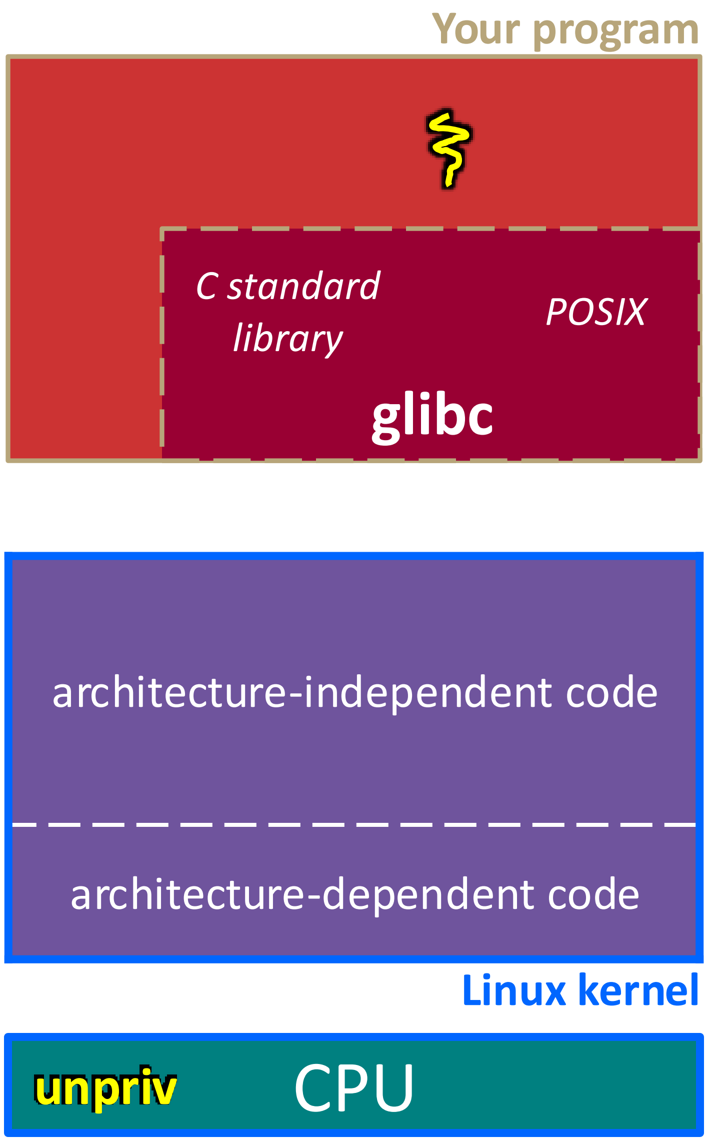




Details on x86/Linux

Process is executing your program code

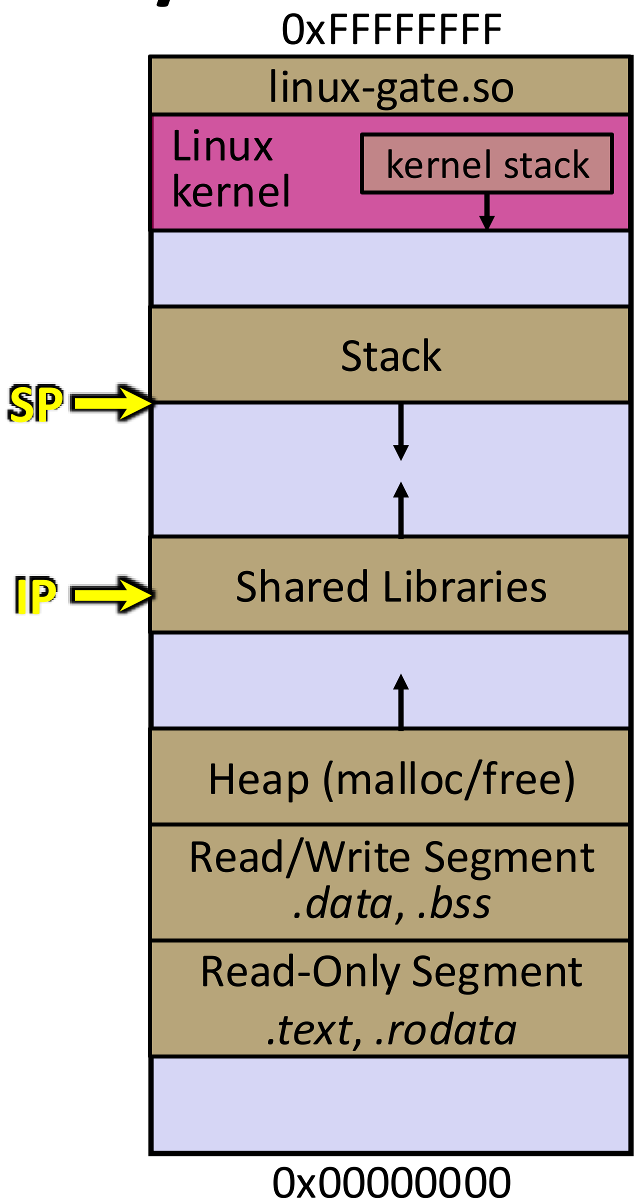


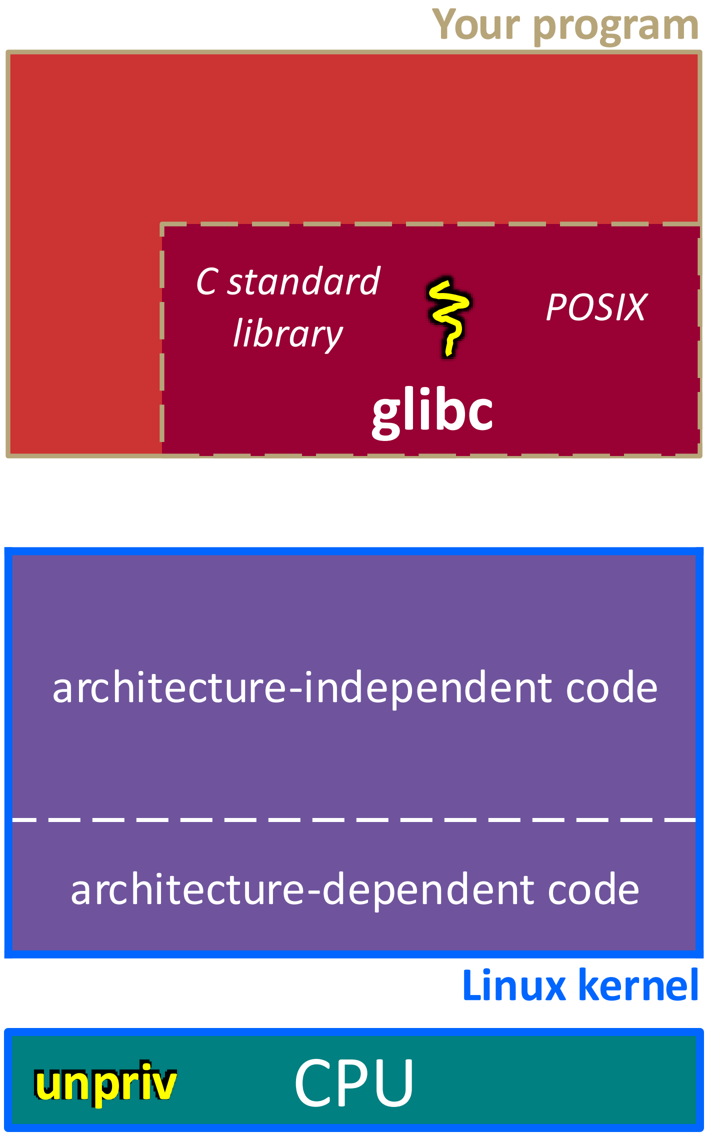


Details on x86/Linux

Process calls into a `glibc` function

- *e.g.* `fopen()`
- We'll ignore the messy details of loading/linking shared libraries

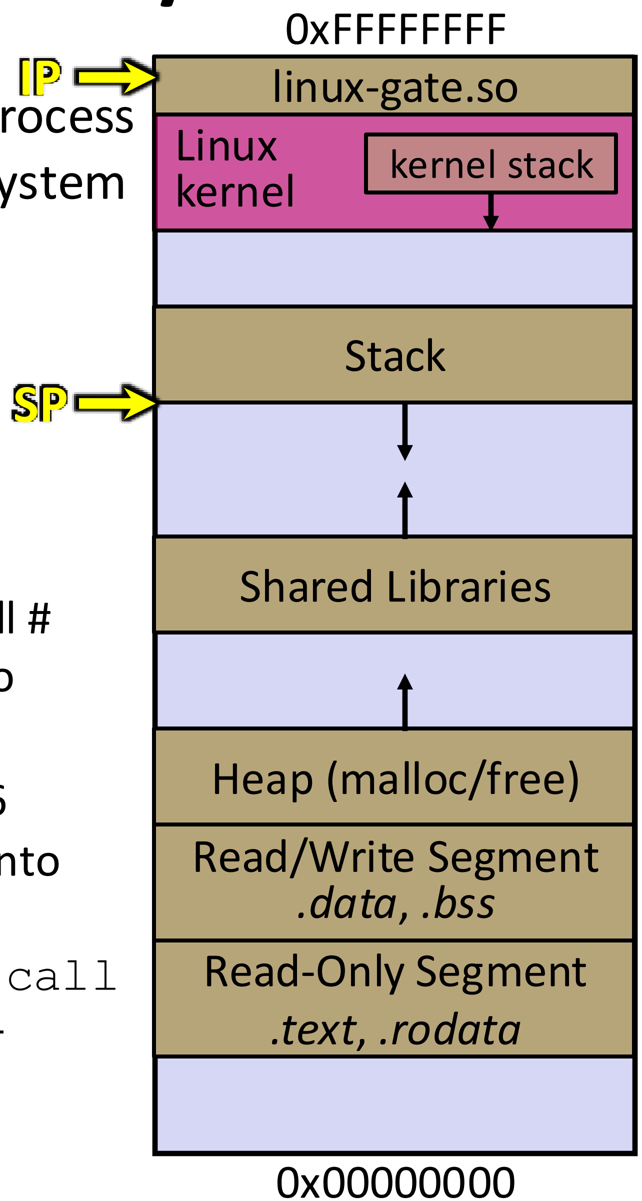


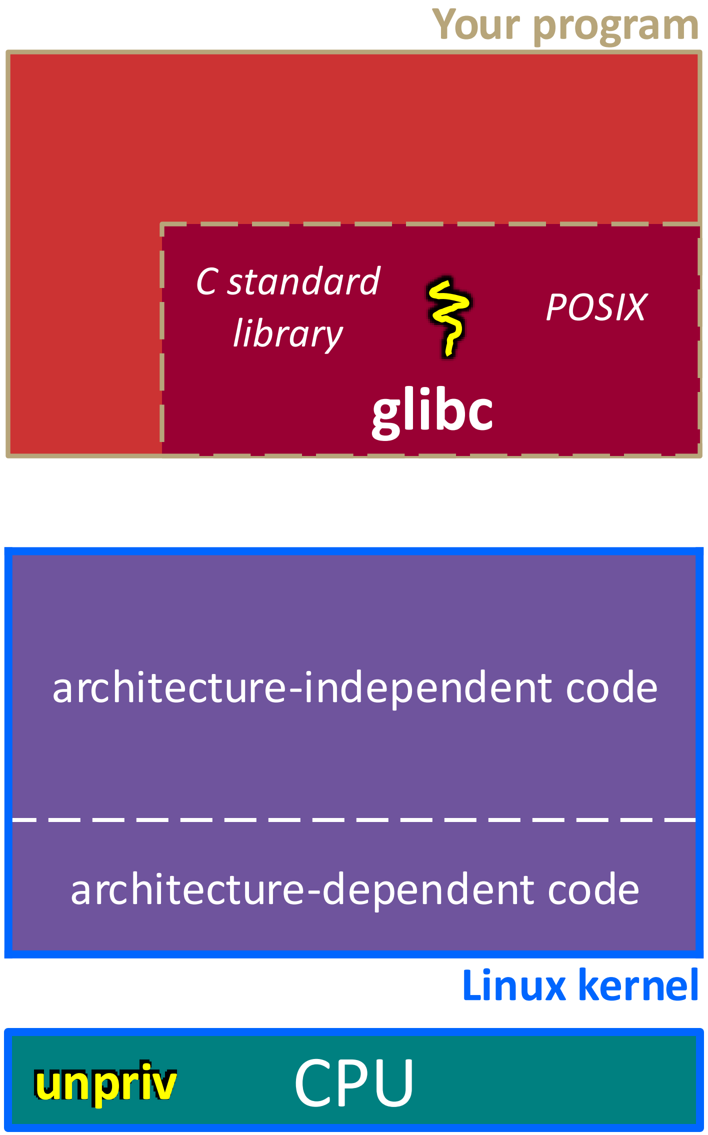


Details on x86/Linux

`glibc` begins the process of invoking a Linux system call

- `glibc`'s `fopen()` likely invokes Linux's `open()` system call
- Puts the system call # and arguments into registers
- Uses the **`call`** x86 instruction to call into the routine `__kernel_vsyscall` located in `linux-gate.so`

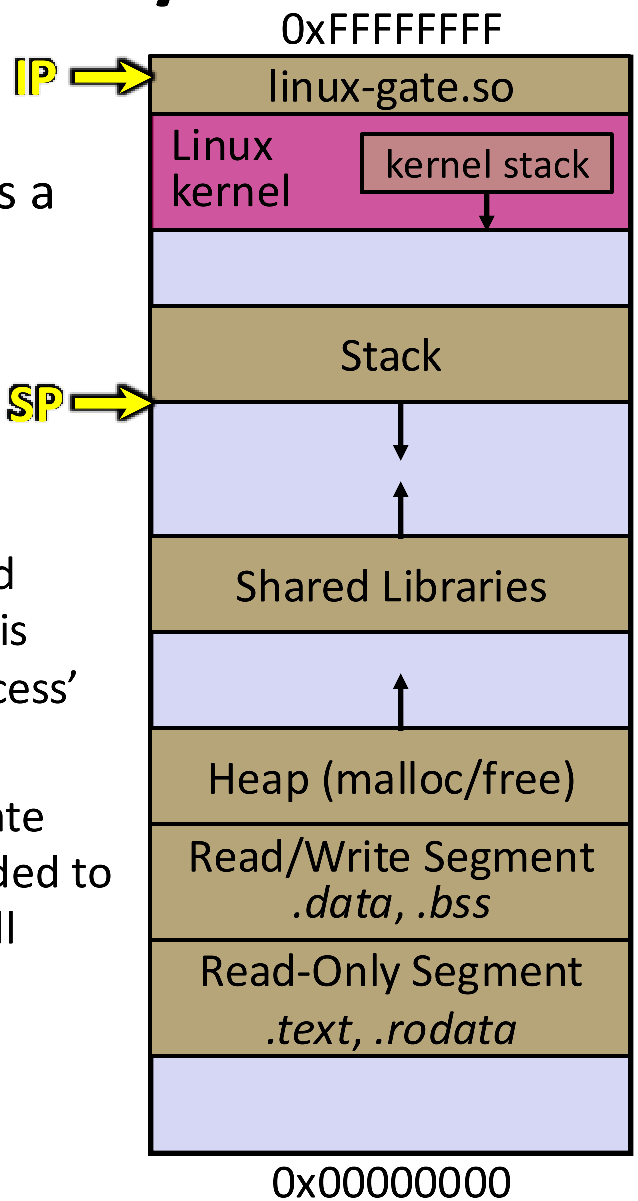


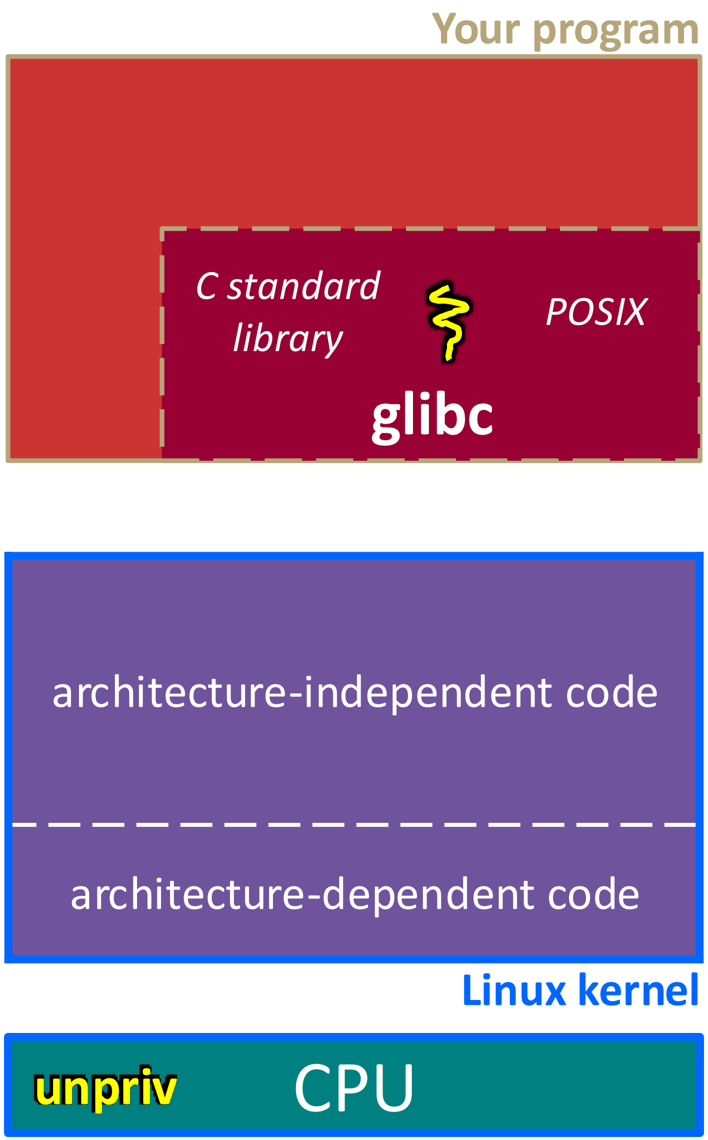


Details on x86/Linux

`linux-gate.so` is a **vdso**

- A virtual dynamically-linked shared object
- Is a kernel-provided shared library that is plunked into a process' address space
- Provides the intricate machine code needed to trigger a system call

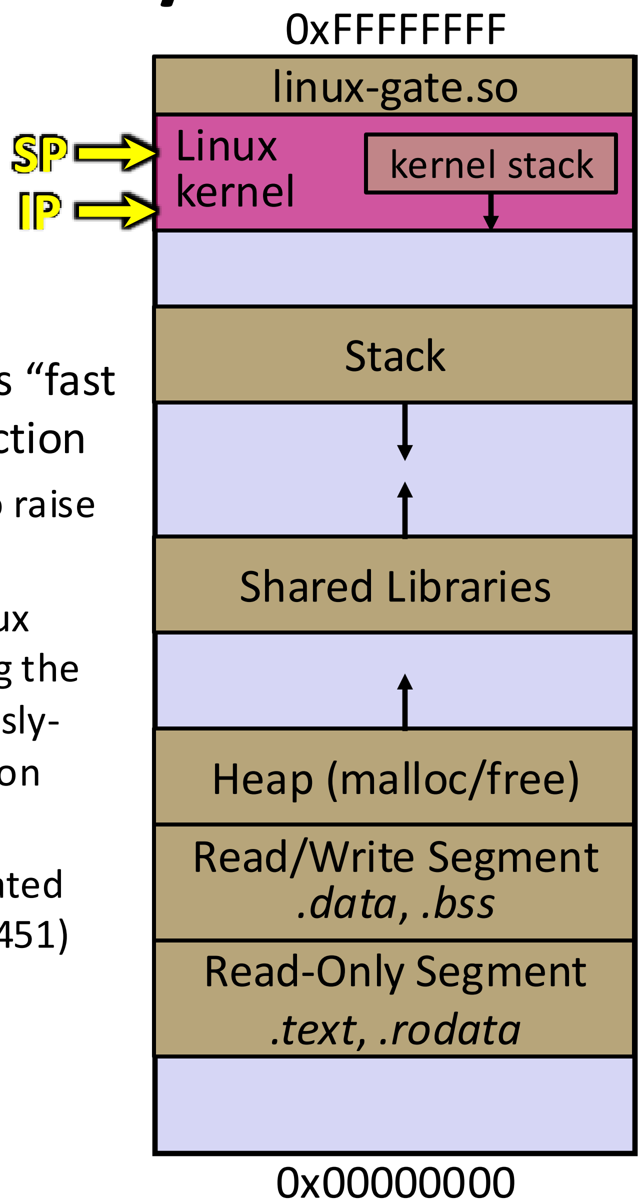


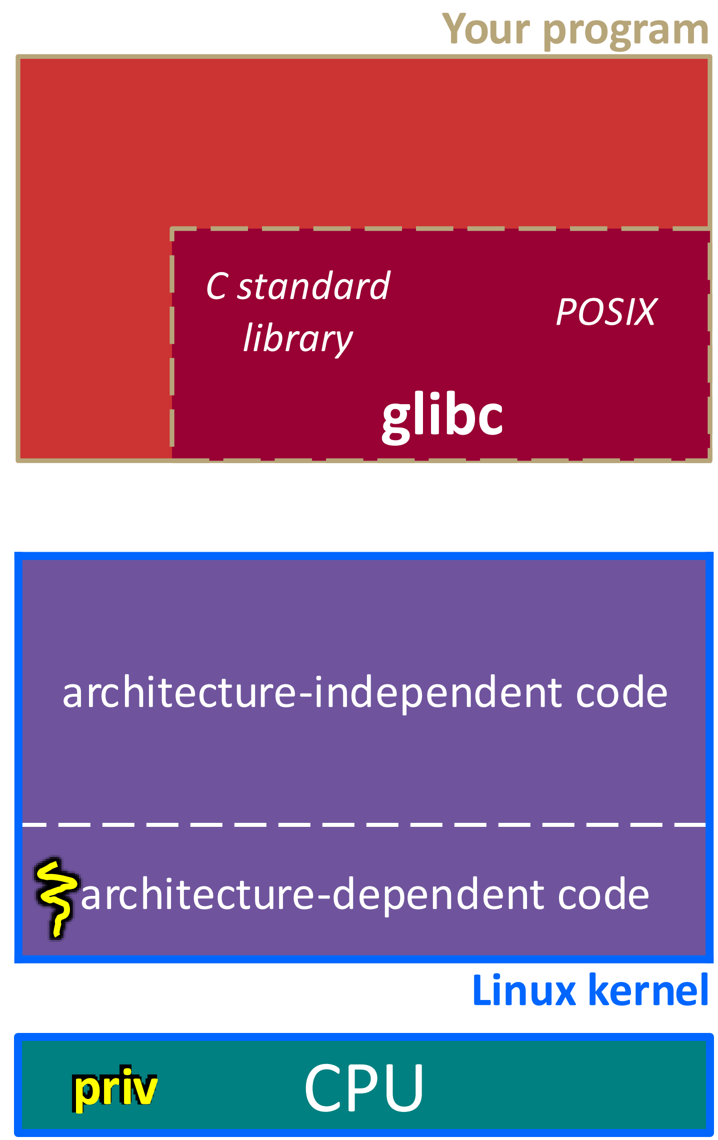


Details on x86/Linux

`linux-gate.so` eventually invokes the `SYSENTER` x86 instruction

- `SYSENTER` is x86's "fast system call" instruction
  - Causes the CPU to raise its privilege level
  - Traps into the Linux kernel by changing the SP, IP to a previously-determined location
  - Changes some segmentation-related registers (see CSE451)

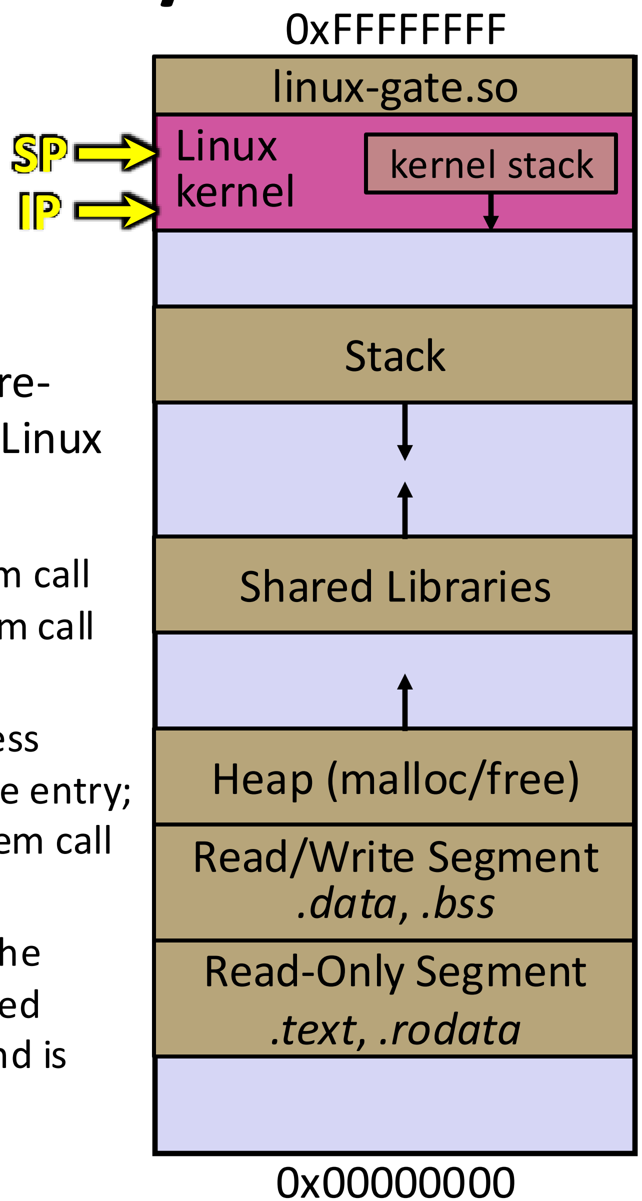


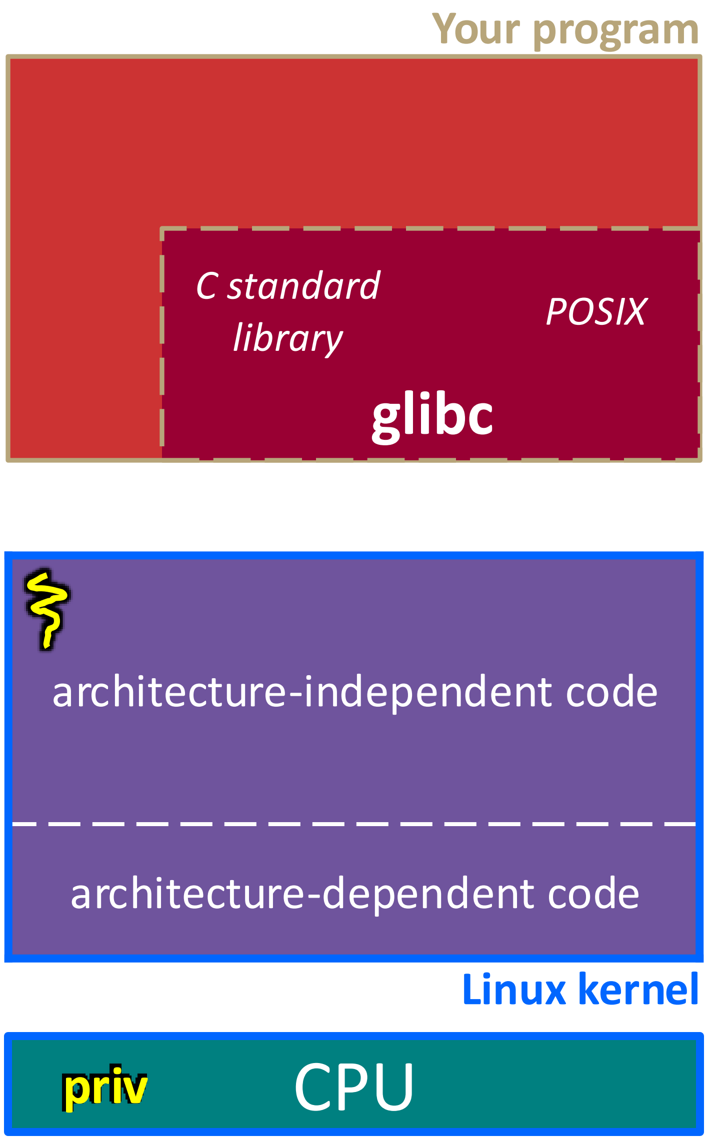


Details on x86/Linux

The kernel begins executing code at the `SYSENTER` entry point

- Is in the architecture-dependent part of Linux
- It's job is to:
  - Look up the system call number in a system call dispatch table
  - Call into the address stored in that table entry; this is Linux's system call handler
    - For `open()`, the handler is named `sys_open`, and is system call #5

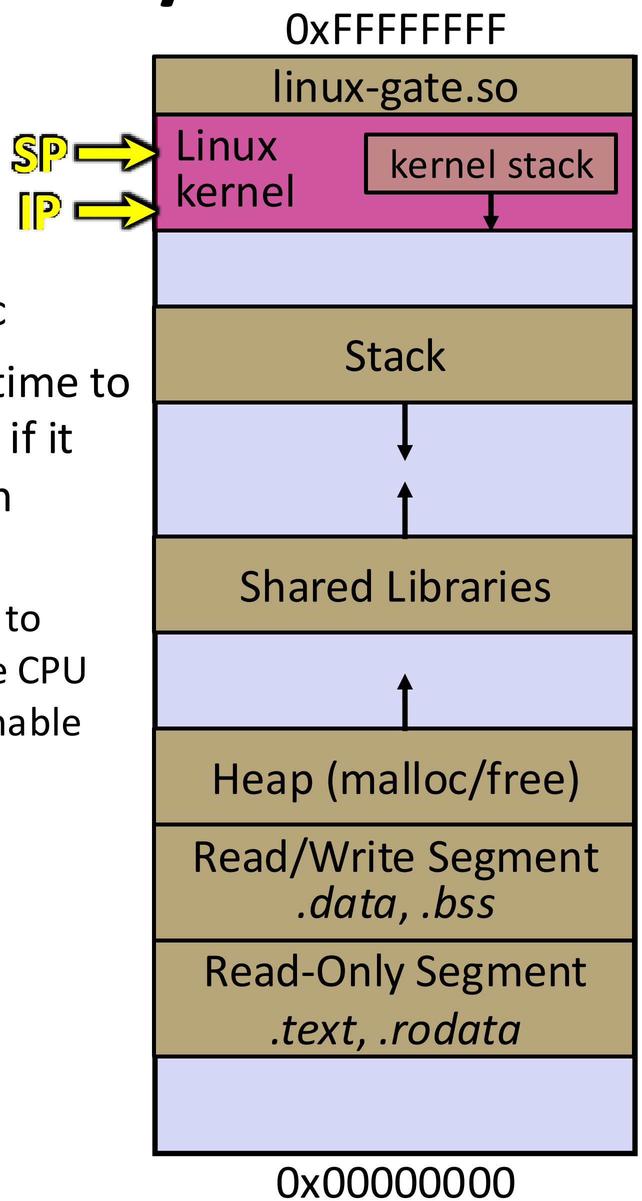


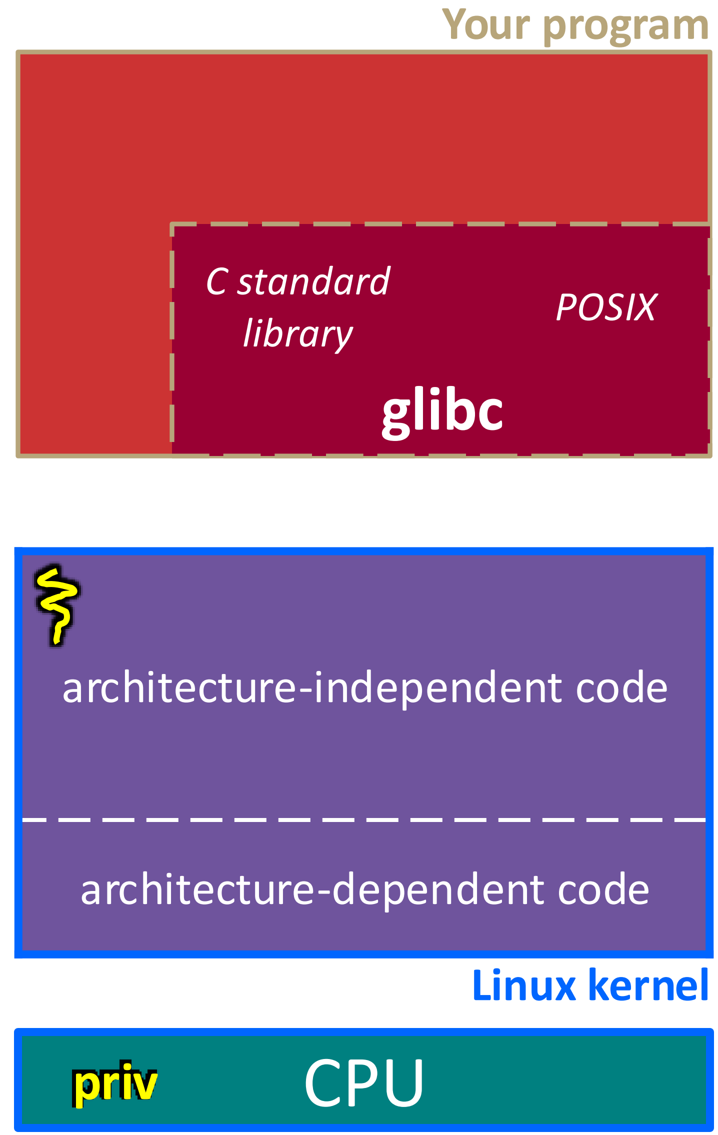


Details on x86/Linux

The system call handler executes

- What it does is system-call specific
- It may take a long time to execute, especially if it has to interact with hardware
  - Linux may choose to context switch the CPU to a different runnable process

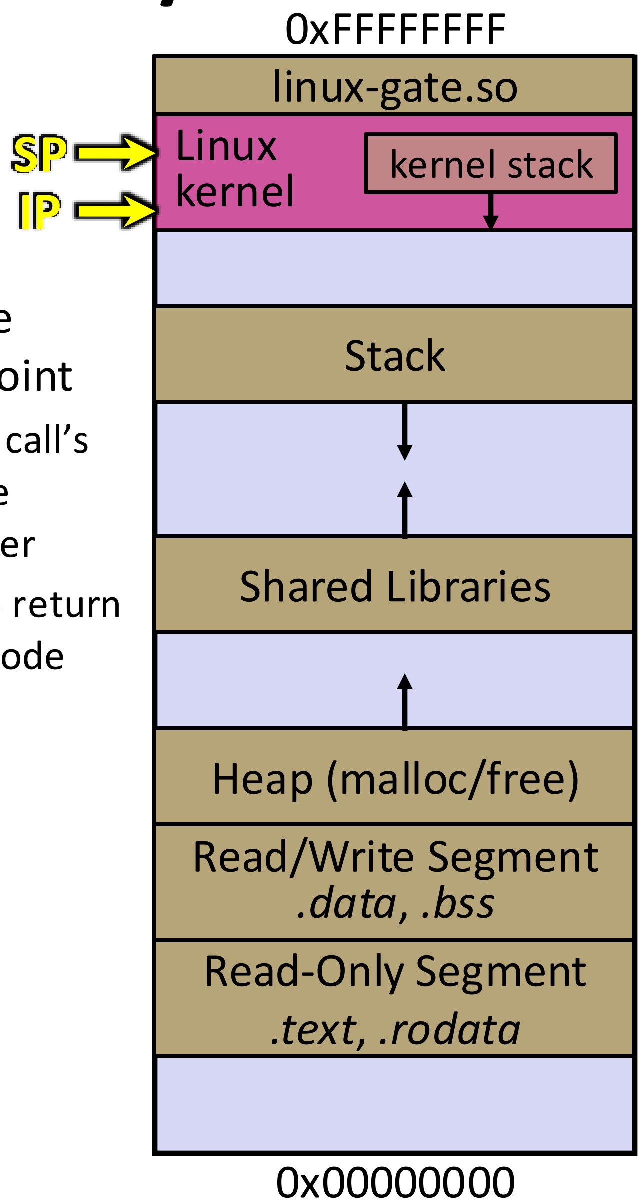


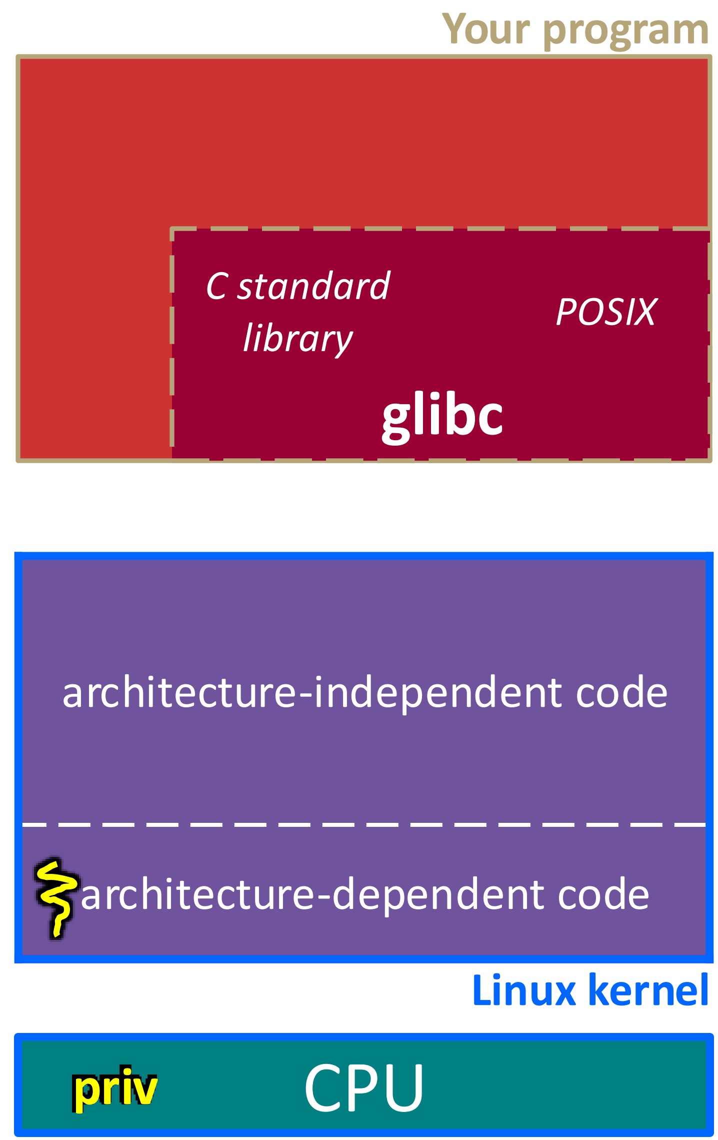


Details on x86/Linux

Eventually, the system call handler finishes

- Returns back to the system call entry point
  - Places the system call's return value in the appropriate register
  - Calls `SYSEXIT` to return to the user-level code

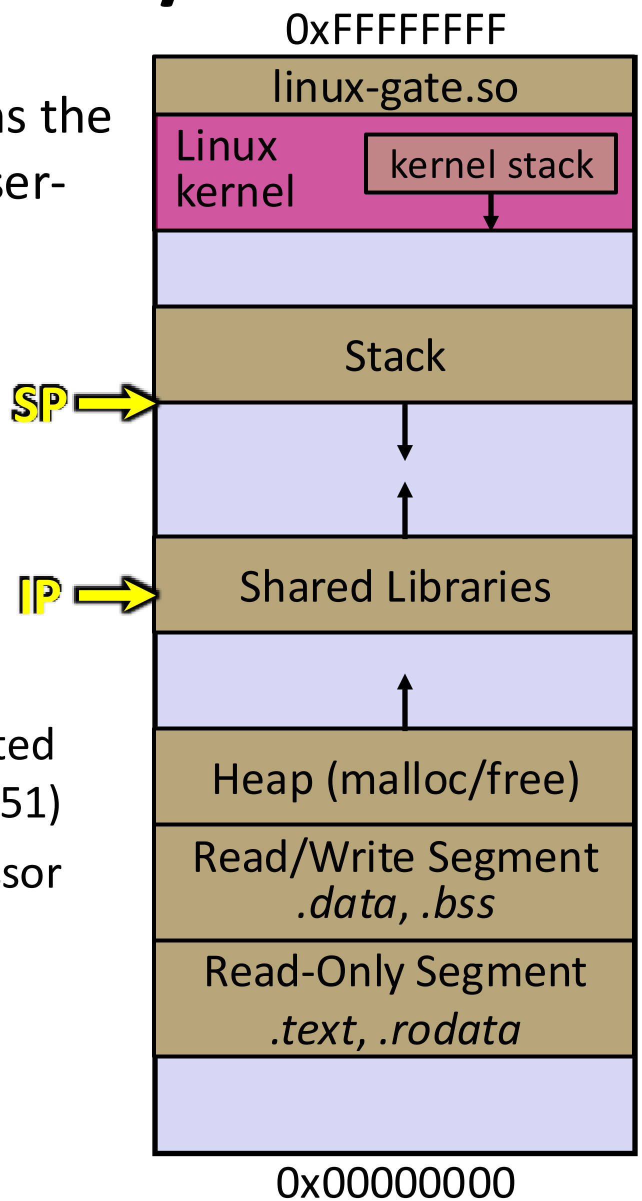


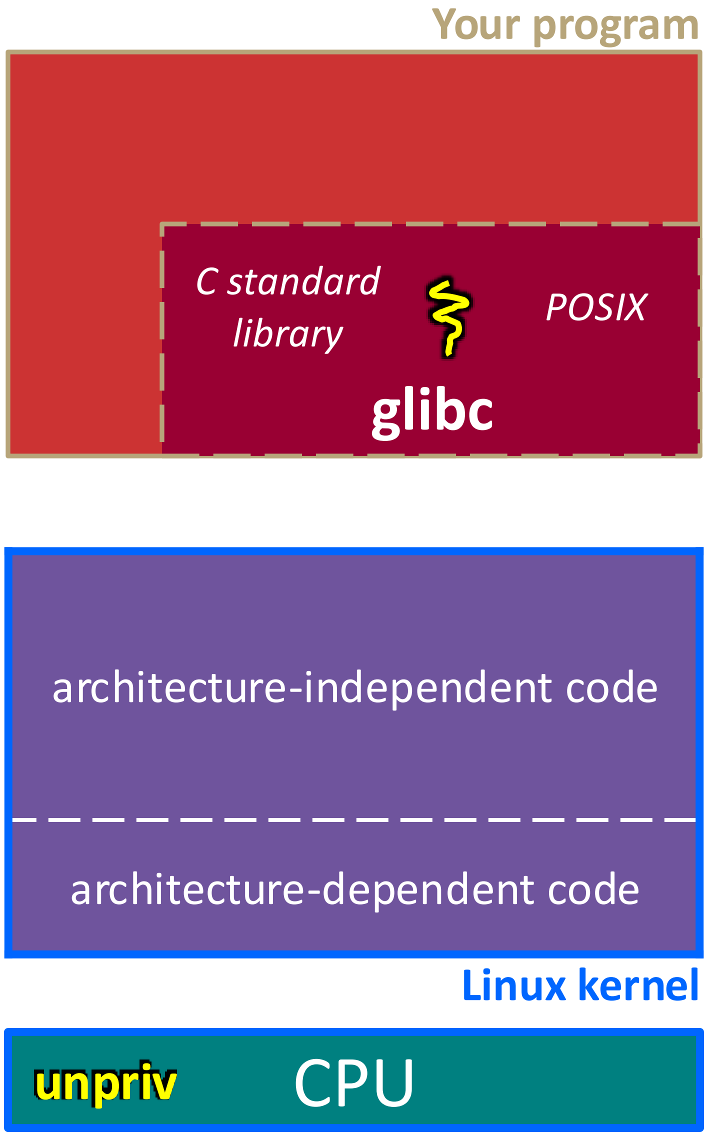


Details on x86/Linux

`SYSEXIT` transitions the processor back to user-mode code

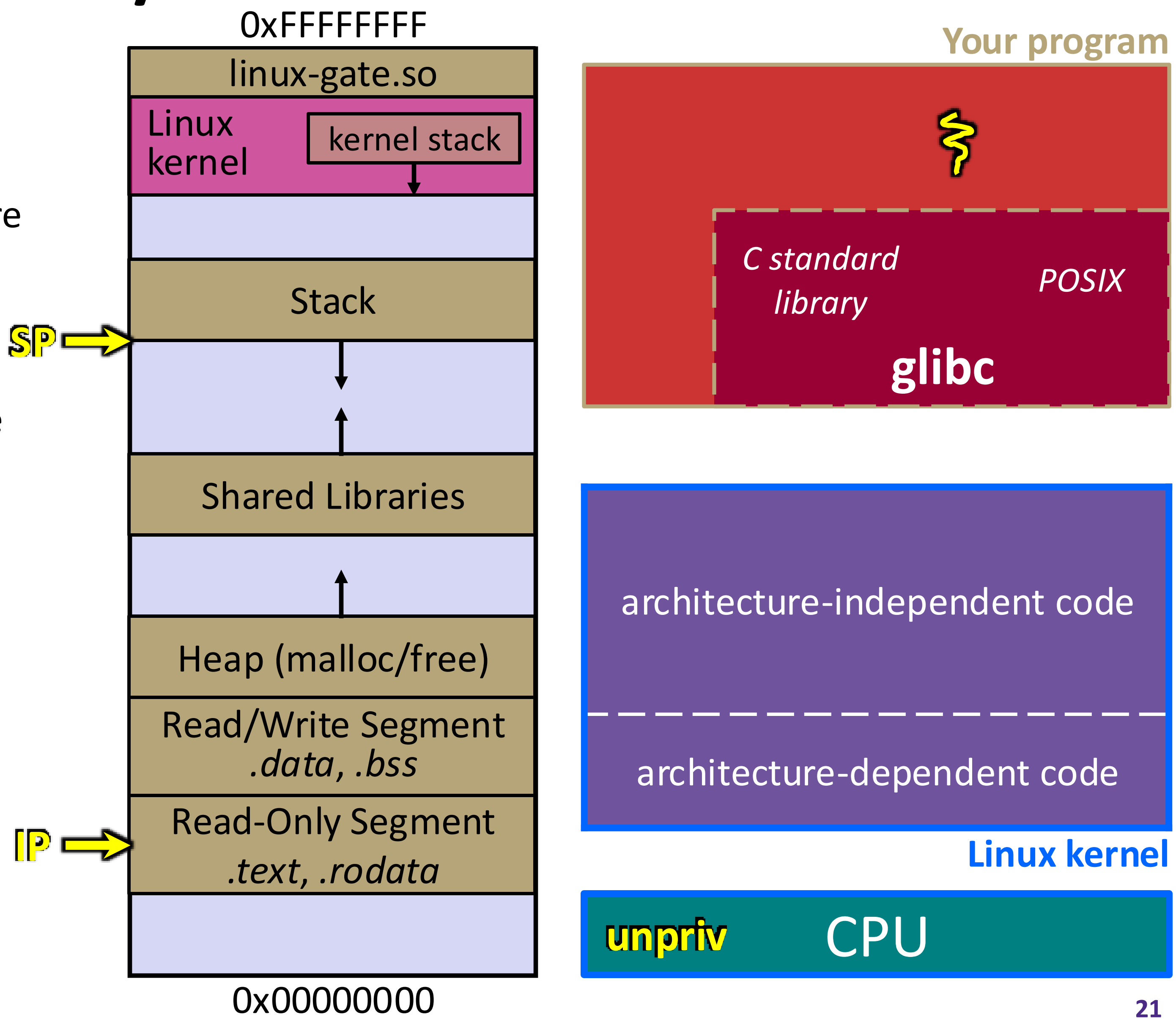
- Restores the IP, SP to user-land values
- Sets the CPU back to unprivileged mode
- Changes some segmentation-related registers (see CSE451)
- Returns the processor back to `glibc`





Details on x86/Linux

`glibc` continues to execute
- Might execute more system calls
- Eventually returns back to your program code



strace

- A useful Linux utility that shows the sequence of system calls that a process makes:

```
bash$ strace ls 2>&1 | less
execve("/usr/bin/ls", ["ls"], [/* 41 vars */]) = 0
brk(NULL)                                        = 0x15aa000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
   0x7f03bb741000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=126570, ...}) = 0
mmap(NULL, 126570, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f03bb722000
close(3)                                         = 0
open("/lib64/libselinux.so.1", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\300j\0\0\0\0\0\0"...,
   832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=155744, ...}) = 0
mmap(NULL, 2255216, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
   0x7f03bb2fa000
mprotect(0x7f03bb31e000, 2093056, PROT_NONE) = 0
mmap(0x7f03bb51d000, 8192, PROT_READ|PROT_WRITE,
   MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x23000) = 0x7f03bb51d000
... etc ...
```

If You're Curious

- Download the Linux kernel source code
  - Available from http://www.kernel.org/
- `man`, section 2: Linux system calls
  - `man 2 intro`
  - `man 2 syscalls`
- `man`, section 3: `glibc`/`libc` library functions
  - `man 3 intro`
- *The* book: *The Linux Programming Interface* by Michael Kerrisk (keeper of the Linux man pages)

Lecture Outline

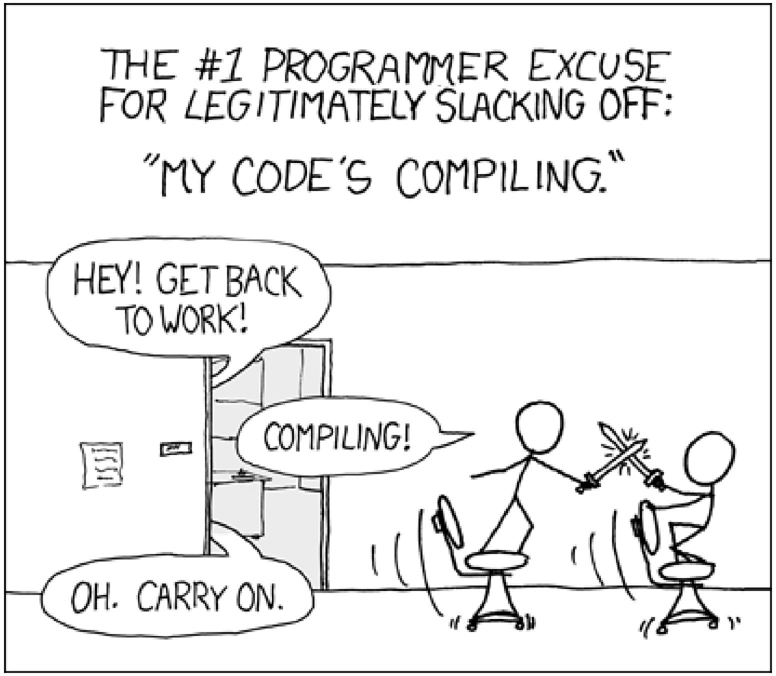
- System Calls (High-Level View)
- **Make and Build Tools**
- Makefile Basics
- C History (for reading, not covered in lecture)

`make`

- `make` is a classic program for controlling what gets (re)compiled and how
  - Many other such programs exist (*e.g.*, `ant`, `maven`, IDE "projects")
- `make` has tons of fancy features, but only two basic ideas:
  1) Scripts for executing commands
  2) Dependencies for avoiding unnecessary work
- To avoid "just teaching `make` features" (boring and narrow), let's focus more on the concepts…

Building Software

- Programmers spend a lot of time "building"
  - Creating programs from source code
  - Both programs that they write and other people write



https://xkcd.com/303/

Building Software

- Programmers spend a lot of time "building"
  - Creating programs from source code
  - Both programs that they write and other people write

- Programmers like to automate repetitive tasks
  - Repetitive: `gcc -Wall -g -std=c17 -o widget foo.c bar.c baz.c`
    - Retype this every time:
    - Use up-arrow or history: (still retype after logout)
    - Have an alias or bash script:
    - Have a Makefile: (you're ahead of us)

"Real" Build Process

- On larger projects, you can't or don't want to have one big (set of) command(s) that are all run every time you change anything. To do things "smarter," consider:
  1) It could be worse: If `gcc` didn't combine steps for you, you'd need to preprocess, compile, and link on your own (along with anything you used to generate the C files)
  2) Source files could have multiple outputs (*e.g.*, `javadoc`). You may have to type out the source file name(s) multiple times
  3) You don't want to have to document the build logic when you distribute source code; make it relatively simple for others to build
  4) You don't want to recompile everything every time you change something (especially if you have 10^5-10^7 files of source code)

- A script can handle 1-3 (use a variable for filenames for 2), but 4 is trickier

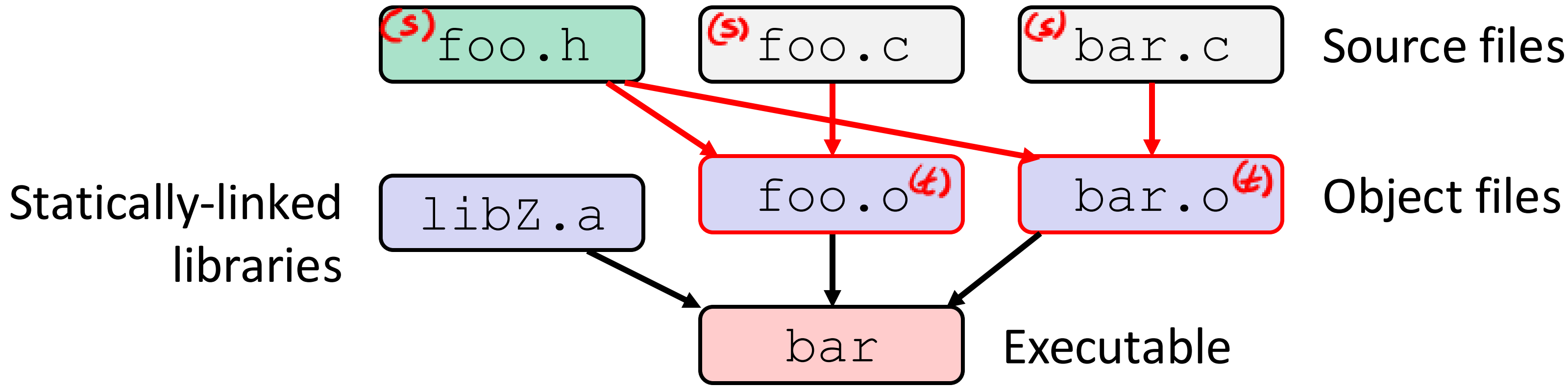
Recompilation Management

- The "theory" behind avoiding unnecessary compilation is a *dependency dag* (**d**irected, **a**cyclic **g**raph)

- To create a target ① t, you need sources ② $s_1, s_2, \dots, s_n$ and a command ③ c that directly or indirectly uses the sources
  - It t is newer than every source (file-modification times), assume there is no reason to rebuild it
  - Recursive building: if some source s_i is itself a target for some other sources, see if it needs to be rebuilt…
  - Cycles "make no sense"!

(s) = source
(t) = target

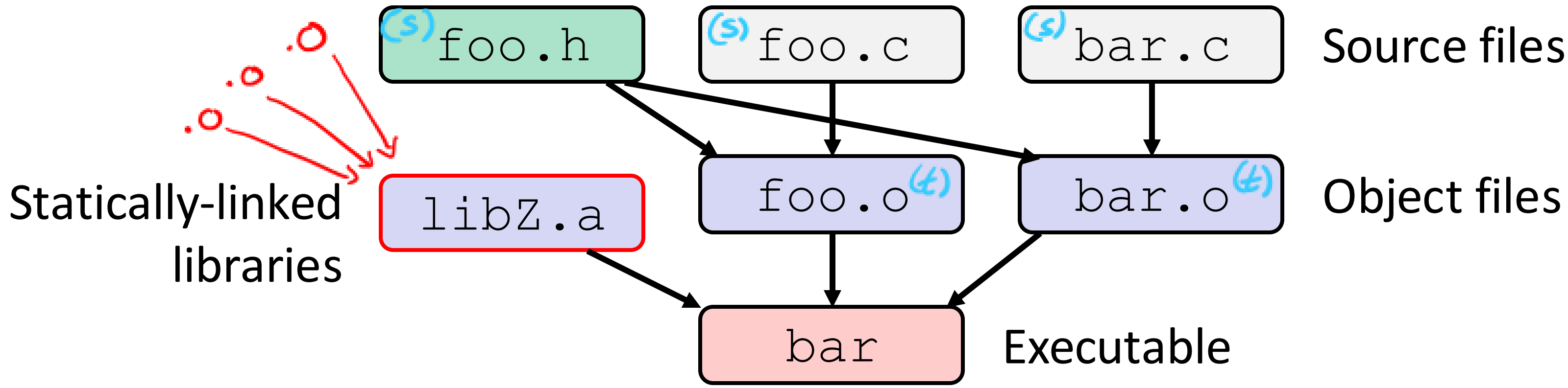
Theory Applied to C



- Compiling a `.c` creates a `.o` – the `.o` depends on the `.c` and all included files (`.h`, recursively/transitively)

(s) = source
(t) = target

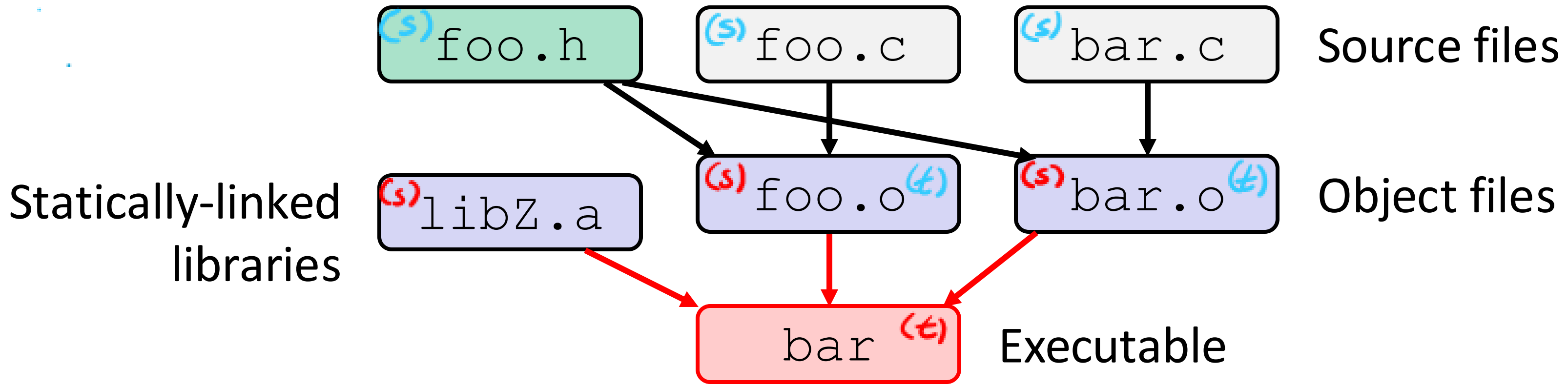
Theory Applied to C



- Compiling a `.c` creates a `.o` – the `.o` depends on the `.c` and all included files (`.h`, recursively/transitively)
- An archive (library, `.a`) depends on included `.o` files

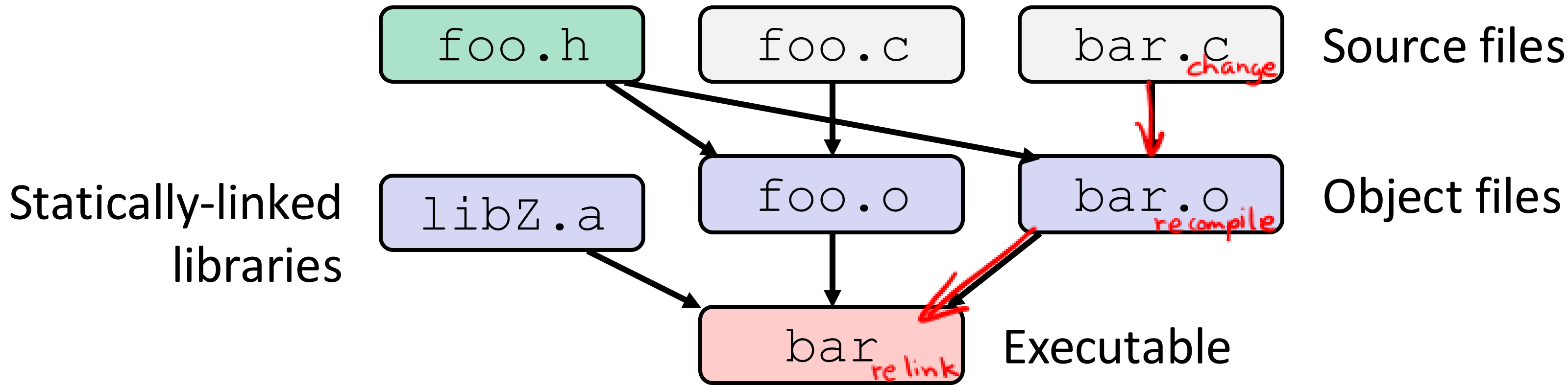
(s) = source
(t) = target

Theory Applied to C



- Compiling a `.c` creates a `.o` – the `.o` depends on the `.c` and all included files (`.h`, recursively/transitively)
- An archive (library, `.a`) depends on included `.o` files
- Creating an executable ("linking") depends on `.o` files and archives
  - Archives linked by `-L<path> -l<name>` (*e.g.*, `-L. -lfoo` to get `libfoo.a` from current directory)

Theory Applied to C



- If one `.c` file changes, just need to recreate one `.o` file, maybe a library, and re-link
- If a `.h` file changes, may need to rebuild more
- Many more possibilities!

Lecture Outline

- System Calls (High-Level View)
- Make and Build Tools
- **Makefile Basics**
- C History (for reading, not covered in lecture)

make Basics

- A makefile contains a bunch of triples:

```
① target: sources ②
← Tab → command ③
```

  - Colon after target is *required*
  - Command lines must start with a **TAB**, NOT SPACES
  - Multiple commands for same target are executed *in order*
    - Can split commands over multiple lines by ending lines with '\'

- Example:

```
foo.o: foo.c foo.h bar.h
	gcc -Wall -o foo.o -c foo.c
```

Using make

```
$ make -f <makefileName> target
```

- Defaults: $ make
  - If no `-f` specified, use a file named `Makefile` in current dir
  - If no `target` specified, will use the first one in the file
  - Will interpret commands in your default shell
    - Set `SHELL` variable in makefile to ensure

- Target execution:
  - Check each source in the source list:
    - If the source is a target in the makefile, then process it recursively
    - If some source does not exist, then error
    - If any source is newer than the target (or target does not exist), run `command` (presumably to update the target)

"Phony" Targets

- A make target whose command does not create a file of the target's name (*i.e.*, a "recipe")
  - As long as target file doesn't exist, the command(s) will be executed because the target must be "remade"
- *e.g.*, target `clean` is a convention to remove generated files to "start over" from just the source

```
clean:
	rm foo.o bar.o baz.o widget *~
```

- *e.g.*, target `all` is a convention to build all "final products" in the makefile
  - Lists all of the "final products" as sources

"all" Example

```
all: prog B.class someLib.a
    # notice no commands this time

prog: foo.o bar.o main.o
      gcc -o prog foo.o bar.o main.o

B.class: B.java
      javac B.java

someLib.a: foo.o baz.o
      ar r foo.o baz.o

foo.o: foo.c foo.h header1.h header2.h
      gcc -c -Wall foo.c

# similar targets for bar.o, main.o, baz.o, etc...
```

make Variables

- You can define variables in a makefile:
  - All values are strings of text, no "types"
  - Variable names are case-sensitive and can't contain ':', '#', '=', or whitespace
- Example:

```
CC = gcc
CFLAGS = -Wall -std=c17
OBJFILES = foo.o bar.o baz.o
widget: $(OBJFILES)
        $(CC) $(CFLAGS) -o widget $(OBJFILES)
```

- Advantages:
  - Easy to change things (especially in multiple commands)
    - It's common to use variables to hold lists of filenames
  - Can also specify/overwrite variables on the command line: (*e.g.*, `make CC=clang CFLAGS=-g`)

Makefile Writing Tips

STYLE TIP

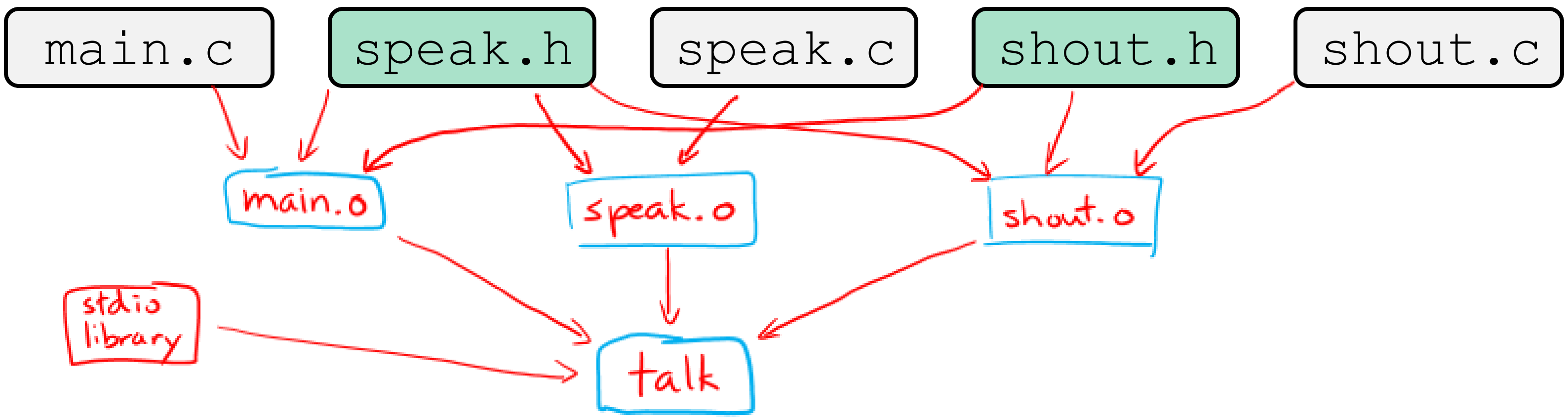
- *When creating a Makefile, first draw the dependencies!!!!*

- C Dependency Rules:
  - `.c` and `.h` files are never targets, only sources.
  - Each `.c` file will be compiled into a corresponding `.o` file
    - Header files will be implicitly used via `#include`
  - Executables will typically be built from one or more `.o` file

- Good Conventions:
  - Include a `clean` rule
  - If you have more than one “final target,” include an `all` rule
  - The first/top target should be your singular “final target” or `all`

Writing a Makefile Example

- "`talk`" program (find files on web with lecture slides)



main.c

```c
#include "speak.h"
#include "shout.h"

int main(int argc, char** argv) {…
```

speak.c

```c
#include "speak.h"
...
```

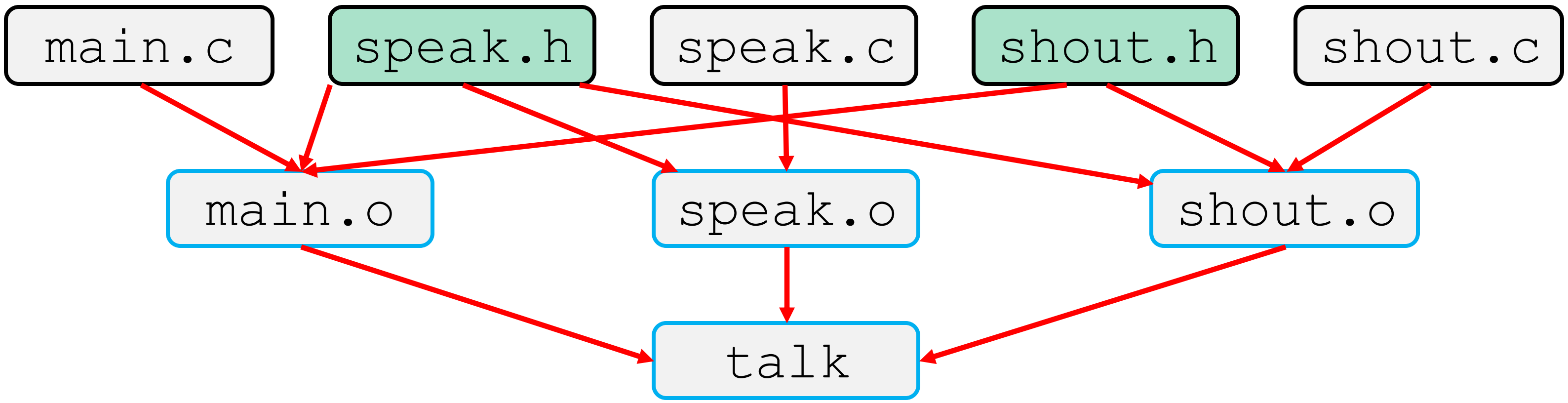
shout.c

```c
#include "speak.h"
#include "shout.h"
...
```

Writing a Makefile Example

target: sources
command

- "`talk`" program (find files on web with lecture slides)




```
talk: main.o speak.o shout.o
    gcc $(CFLAGS) -o talk main.o speak.o shout.o

main.o: main.c speak.h shout.h
    gcc $(CFLAGS) -c main.c
speak.o: speak.c speak.h
    gcc $(CFLAGS) -c speak.c
shout.o: shout.c speak.h shout.h
    gcc $(CFLAGS) -c shout.c
clean:
    rm talk *.o
```

Revenge of the Funny Characters

- Special variables:
  - `$@` for target name
  - `$^` for all sources
  - `$<` for left-most source
  - Lots more! – see the documentation

- Examples:

```
# CC and CFLAGS defined above
widget: foo.o bar.o
        $(CC) $(CFLAGS) -o $@ $^
foo.o: foo.c foo.h bar.h
        $(CC) $(CFLAGS) -c $<
```

And more…

- There are a lot of "built-in" rules – see documentation
- There are "suffix" rules and "pattern" rules
  - Example:

```
%.class: %.java
        javac $<  # we need the $< here
```

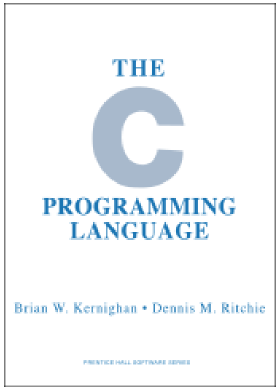
- Remember that you can put *any* shell command – even whole scripts!
- You can repeat target names to add more dependencies
- Often this stuff is more useful for reading makefiles than writing your own (until some day…)

Lecture Outline

- System Calls (High-Level View)
- Make and Build Tools
- Makefile Basics
- **C History (for reading, not covered in lecture)**

Development of the C Language

- Created in 1972
  - BCPL → B → C
  - Designed specifically as a system programming language for Unix
    - Unix was rewritten entirely in C (Version 4 in 1973)
- "Standardized" in 1978 with release of K&R Ed. 1
  - From initial creation, developed in terms of portability and type safety



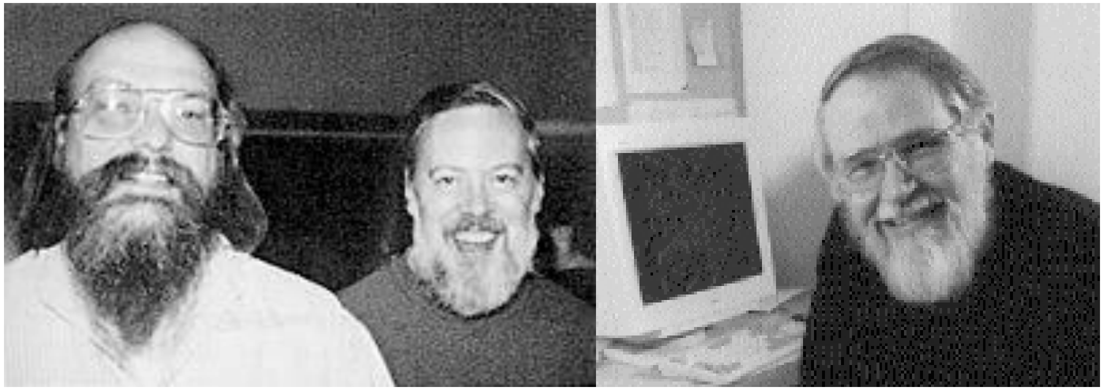
- Formal standardization via American National Standards Institute (ANSI) in 1989 and International Organziation for Standardization (ISO) in 1990
  - Non-portable portion of the Unix C library was the basis for the POSIX standard via IEEE

Development of the C Language

- Development Context:
  - Developed for the PDP-7/PDP-11
    - Very limited memory available for program
  - Improvements over B: data typing, performance, byte addressibility
  - Developed in the context of operating system innovations (Multics, Unix)
    - "Particularly oriented towards system programming, are small and compactly described, and are amenable to translation by simple compilers."
    - "By design, C provides constructs that map efficiently to typical machine instructions. It has found lasting use in applications previously coded in assembly language."
- Who used computers and programming at the time?

Development of the C Language

- Credits:
  - **Dennis Ritchie** designed C
  - **Ken Thompson** designed B and, with Ritchie, were the primary architects of UNIX (in assembly)
  - **Brian Kernighan** helped Ritchie write K&R, the first "standardization" of the C language
- "The development of the C language" (https://dl.acm.org/doi/10.1145/155360.155580)



Ken Thompson | Dennis Ritchie | Brian Kernighan

Principles of C

- Some commonly-held contemporary views:
  - "Since C is relatively small, it can be described in small space and learned quickly."
  - "Shows what's really happening."
  - "Close to the machine/hardware."
  - "Only the bare essentials."
  - "No one to help you."
  - "You're on your own."
  - "I know what I'm doing, get out of my way."