Poll Everywhere

pollev.com/cse333sp

About how long did Exercise 3 take you?

A. [0, 2) hours
B. [2, 4) hours
C. [4, 6) hours
D. [6, 8) hours
E. 8+ Hours
F. I didn't submit / I prefer not to say

System Calls, Makefiles
CSE 333 Spring 2023

Instructor: Chris Thachuk

Teaching Assistants:

Byron Jin	CJ Reith
Deeksha Vatwani	Edward Zhang
Humza Lala	Lahari Nidadavolu
Noa Ferman	Saket Gollapudi
Seulchan (Paul) Han	Timmy Yang
Tim Mandzyuk	Wui Wu

Relevant Course Information

- Homework 1 due Thursday night (4/13)
  - Clean up “to do” comments, but leave “STEP #” markers
  - Graded not just on correctness, also code quality
  - OH get crowded – come prepared to describe your incorrect behavior and what you think the issue is and what you’ve tried
  - Late days: don’t tag `hw1-final` until you are really ready
    - Please use them if you need to!
- Homework 2 (and next exercise) released today
  - Partner declaration form and matching form will be released after the spec is released

Cont'd from previous lecture

- File I/O with the C standard library
- C Stream Buffering
- **POSIX Lower-Level I/O**

From C to POSIX

- Most UNIX-en support a common set of lower-level file access APIs: POSIX – Portable Operating System Interface
  - **`open`**`()`, **`read`**`()`, **`write`**`()`, **`close`**`()`, **`lseek`**`()`
    - Similar in spirit to their `f*()` counterparts from the C std lib
    - Lower-level and unbuffered compared to their counterparts
    - Also less convenient
  - You will have to use these to read file system directories and for network I/O, so we might as well learn them now
    - These are functionalities that C stdio *doesn't* provide!

open/close

- To open a file:
  - Pass in the filename and access mode (similar to **fopen**)
  - Get back a "file descriptor"
    - Similar to FILE* from **fopen**, but is just an int
    - **-1** indicates an error

```
#include <fcntl.h>     // for open()
#include <unistd.h>    // for close()
  ...
  int fd = open("foo.txt", O_RDONLY);
  if (fd == -1) {
    perror("open failed");
    exit(EXIT_FAILURE);
  }
  ...
  close(fd);
```

filename access mode file descriptor

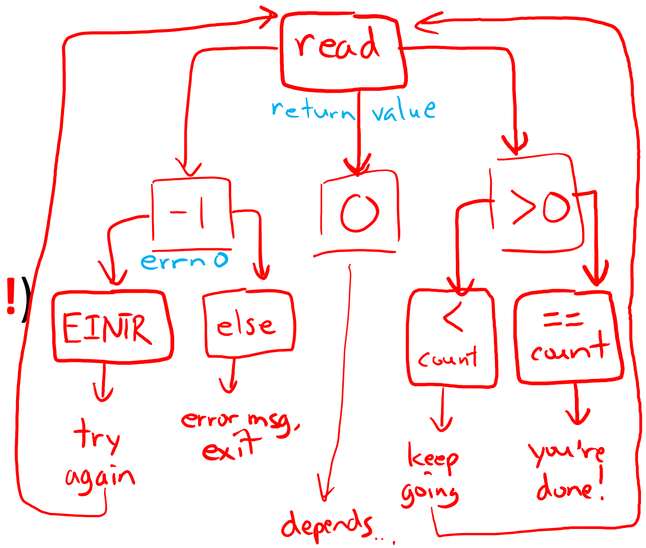
- Open descriptors: **0** (stdin), **1** (stdout), **2** (stderr)

Reading from a File

try to read count bytes

```
ssize_t read(int fd, void* buf, size_t count);
```

- Advances forward in the file by number of bytes read
- Returns the number of bytes read
  - Might be fewer bytes than you requested (!!!)
  - Returns 0 if you're already at the end-of-file
  - Returns -1 on error (and sets `errno`)



- There are some surprising error modes (check `errno`)
  - `EBADF`: bad file descriptor
  - `EFAULT`: output buffer is not a valid address
  - `EINTR`: read was interrupted, please try again (ARGH!!!! 😤 😠)
  - And many others…

these are defined in errno.h

One method to `read()` n bytes

```
int fd = open(filename, O_RDONLY);
char* buf = ...;  // buffer of appropriate size
int bytes_left = n;
int result;

while (bytes_left > 0) {
  result = read(fd, buf + (n - bytes_left), bytes_left);
  if (result == -1) {
    if (errno != EINTR) {
      // a real error happened, so return an error result
    }
    // EINTR happened, so do nothing and try again
    continue;
  } else if (result == 0) {
    // EOF reached, so stop reading
    break;
  }
  bytes_left -= result;
}

close(fd);
```

prevent infinite loop if EOF reached

readN.c

Other Low-Level Functions

- Read man pages to learn about:
  - `write()` – write data
    - `#include <unistd.h>`
  - `fsync()` – flush data to the underlying device
    - `#include <unistd.h>`
  - `opendir()`, `readdir()`, `closedir()` – deal with directory listings
    - Make sure you read the section 3 version (*e.g.*, `man 3 opendir`)
    - `#include <dirent.h>`

- A useful shortcut sheet (from CMU): http://www.cs.cmu.edu/~guna/15-123S11/Lectures/Lecture24.pdf

C Standard Library vs. POSIX

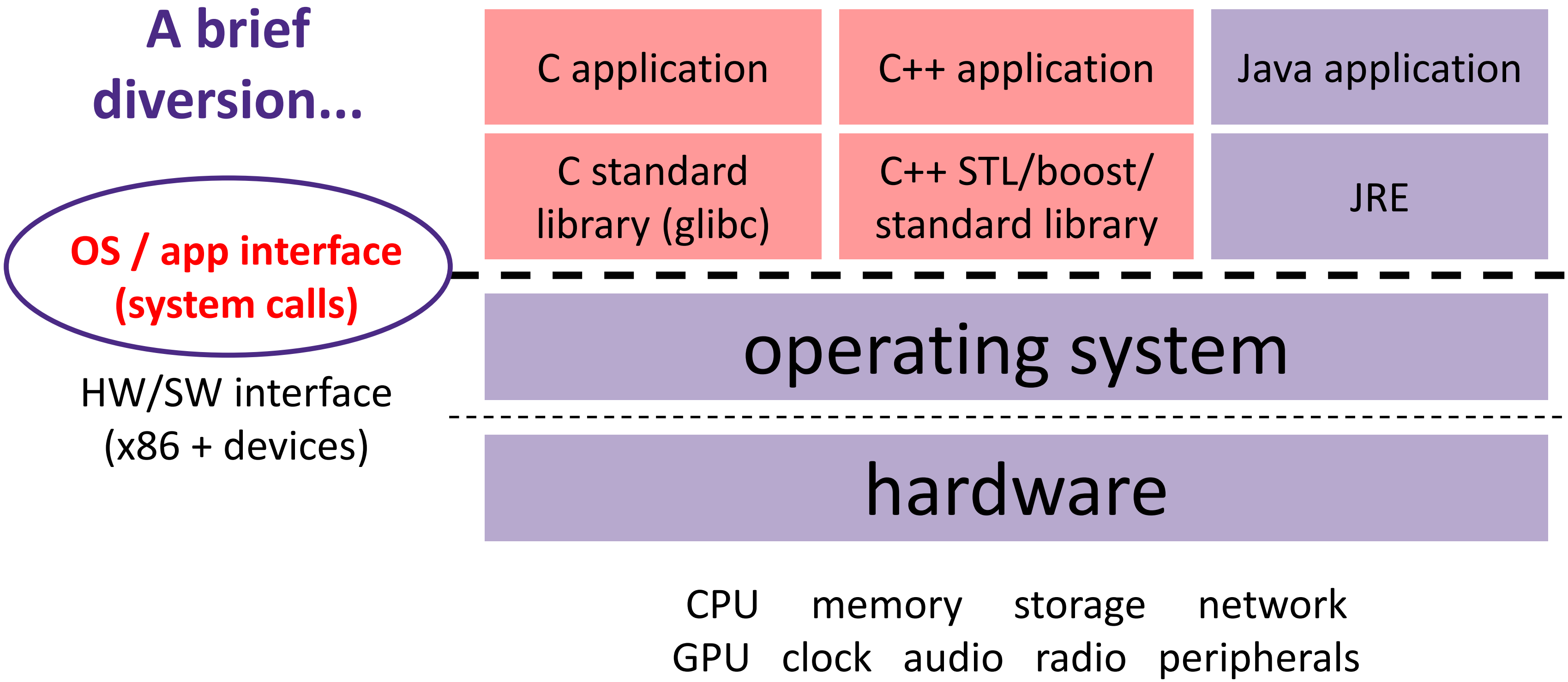
- C standard library implements a <u>subset of POSIX</u>
  - *e.g.*, POSIX provides directory manipulation that C std lib doesn't
- C standard library implements <u>automatic buffering</u>
- C standard library has a <u>nicer API</u>

- The two are similar but C standard library builds on top of POSIX
  - Choice between high-level and low-level
  - Will depend on the requirements of your application
  - You will explore this relationship in Exercise 4!

Lecture Outline

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

Remember This Picture?

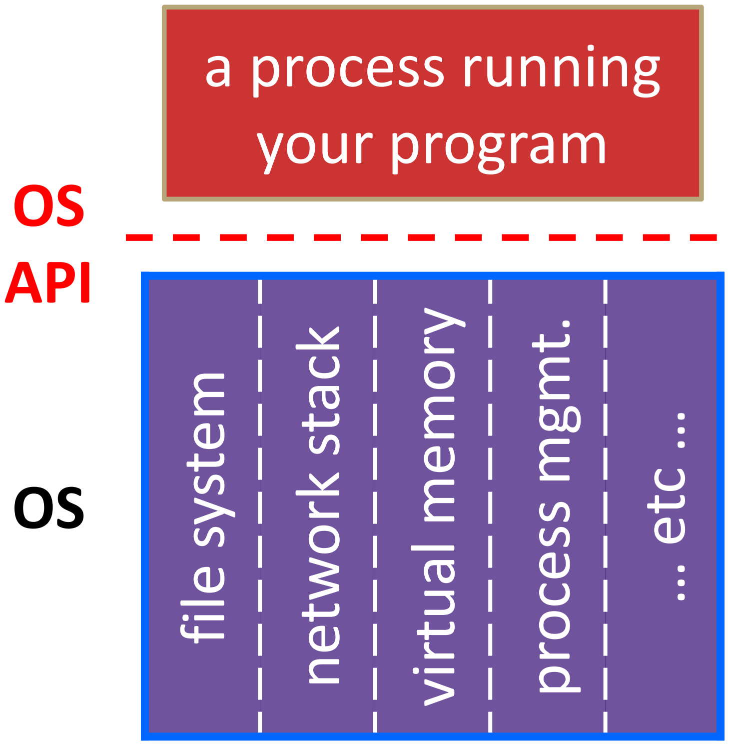


What's an OS?

- Software that:
  - Directly interacts with the hardware
    - OS is trusted to do so; user-level programs are not
    - OS must be ported to new hardware; user-level programs are portable
  - Manages (allocates, schedules, protects) hardware resources
    - Decides which programs can access which files, memory locations, pixels on the screen, etc. and when
  - Abstracts away messy hardware devices
    - Provides high-level, convenient, portable abstractions (*e.g.*, files, disk blocks)

OS: Abstraction Provider

- The OS is the "layer below"
  - A module that your program can call (with system calls)
  - Provides a powerful OS API – POSIX, Windows, etc.



File System

- open(), read(), write(), close(), ...

Network Stack

- connect(), listen(), read(), write(), ...

Virtual Memory

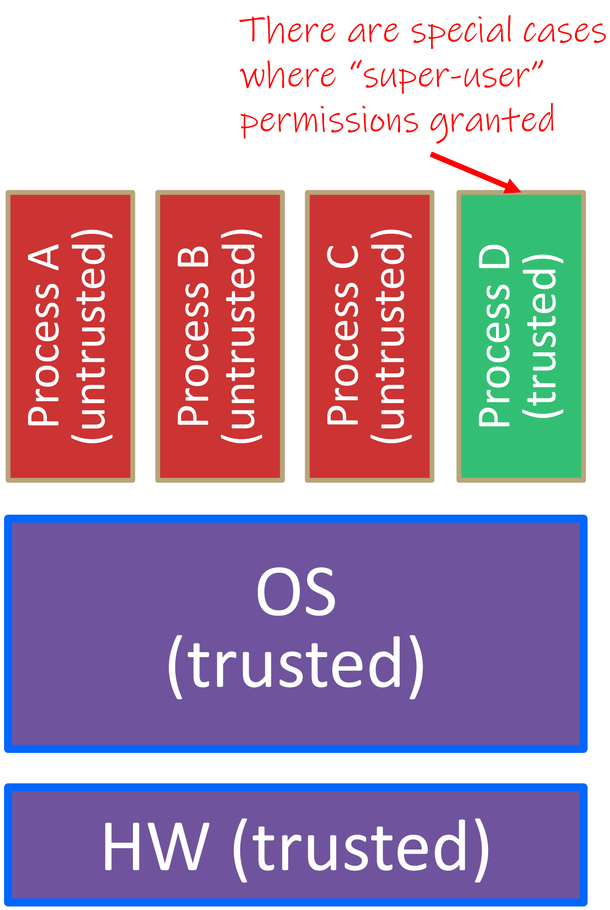
- brk(), shm_open(), ...

Process Management

- fork(), wait(), nice(), ...

OS: Protection System

- OS isolates process from each other
  - But permits controlled sharing between them
    - Through shared name spaces (*e.g.*, file names)
- OS isolates itself from processes
  - Must prevent processes from accessing the hardware directly
- OS is allowed to access the hardware
  - User-level processes run with the CPU (processor) in unprivileged mode
  - The OS runs with the CPU in privileged mode
  - User-level processes invoke **system calls** to safely enter the OS



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

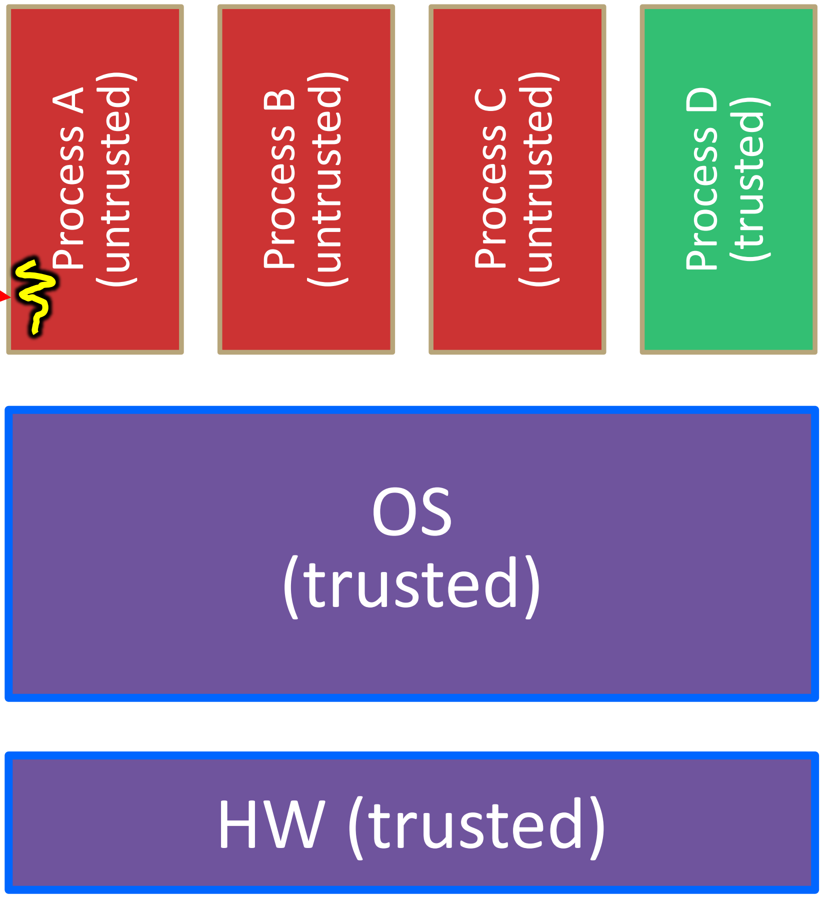
System Calls Simplified Overview

- The operating system (OS) is a super complicated "program overseer" program for the computer
  - The only software that is directly trusted with hardware access

- If a user process wants to access an OS feature, they must invoke a system call
  - A system call involves context switching into the OS/kernel, which has some overhead
  - The OS will handle hardware/special functionality directly (in privileged mode) while user processes wait and don't touch anything themselves
  - OS will eventually finish, return result to user process, and context switch back

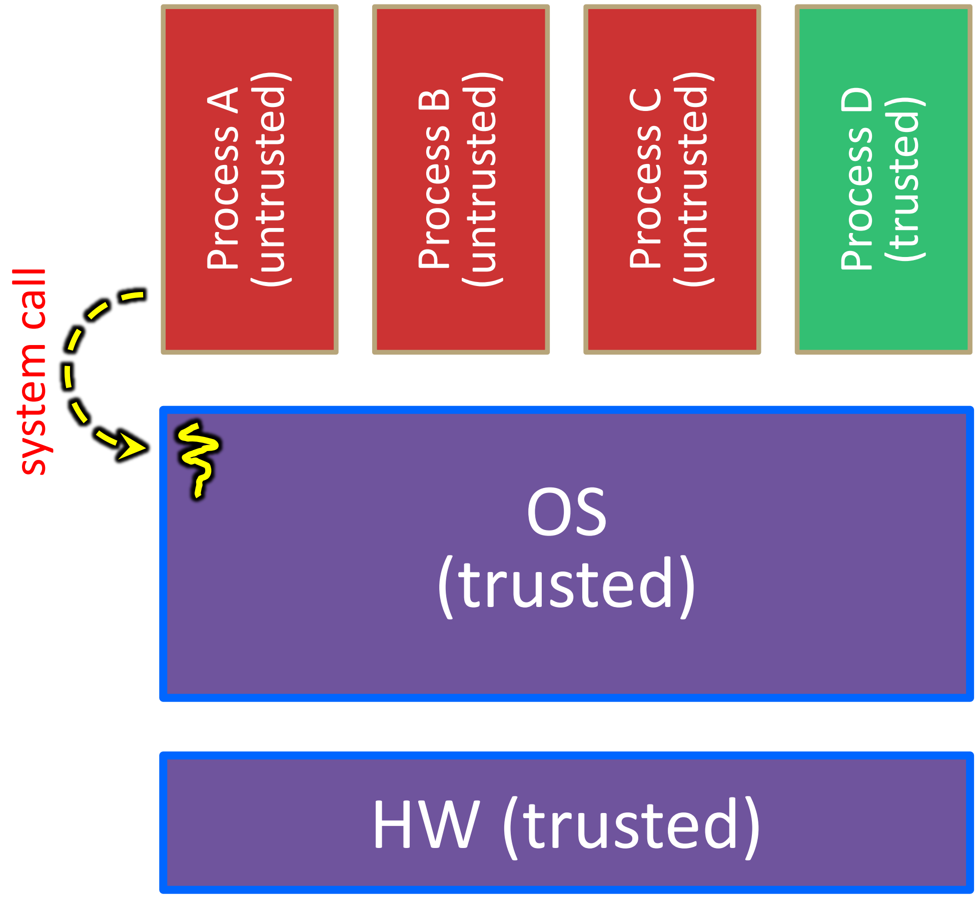
System Call Trace (high-level view)

A CPU (thread of execution) is running user-level code in Process A; the CPU is set to *unprivileged mode*.



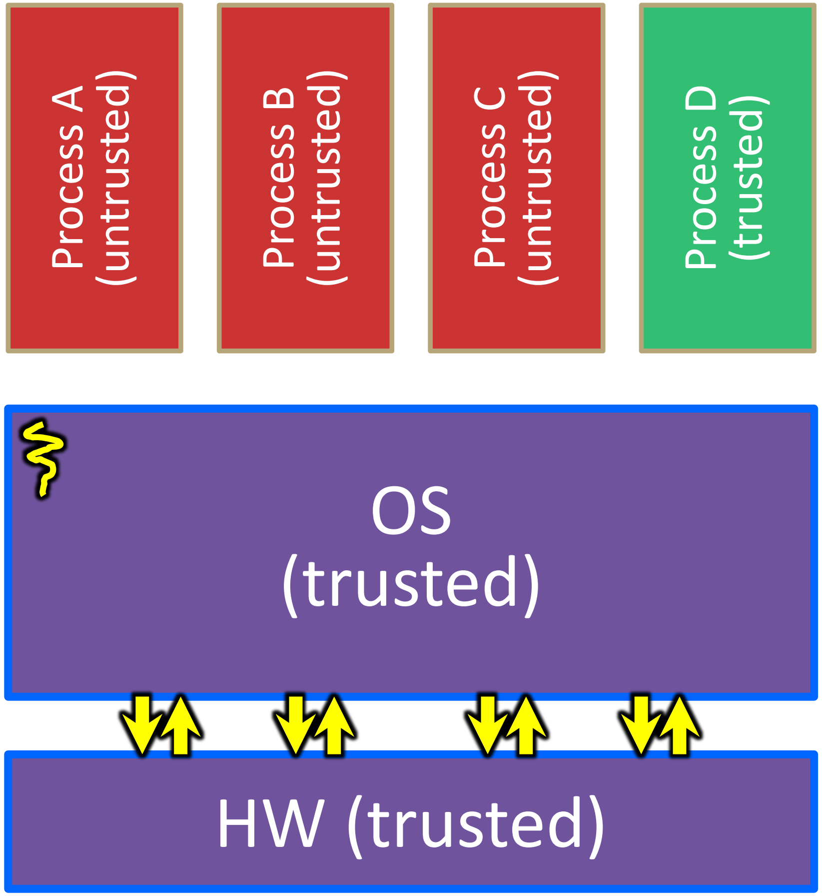
System Call Trace (high-level view)

Code in Process A invokes a system call; the hardware then sets the CPU to *privileged mode* and traps into the OS, which invokes the appropriate system call handler.



System Call Trace (high-level view)

Because the CPU executing the thread that's in the OS is in privileged mode, it is able to use *privileged instructions* that interact directly with hardware devices like disks.

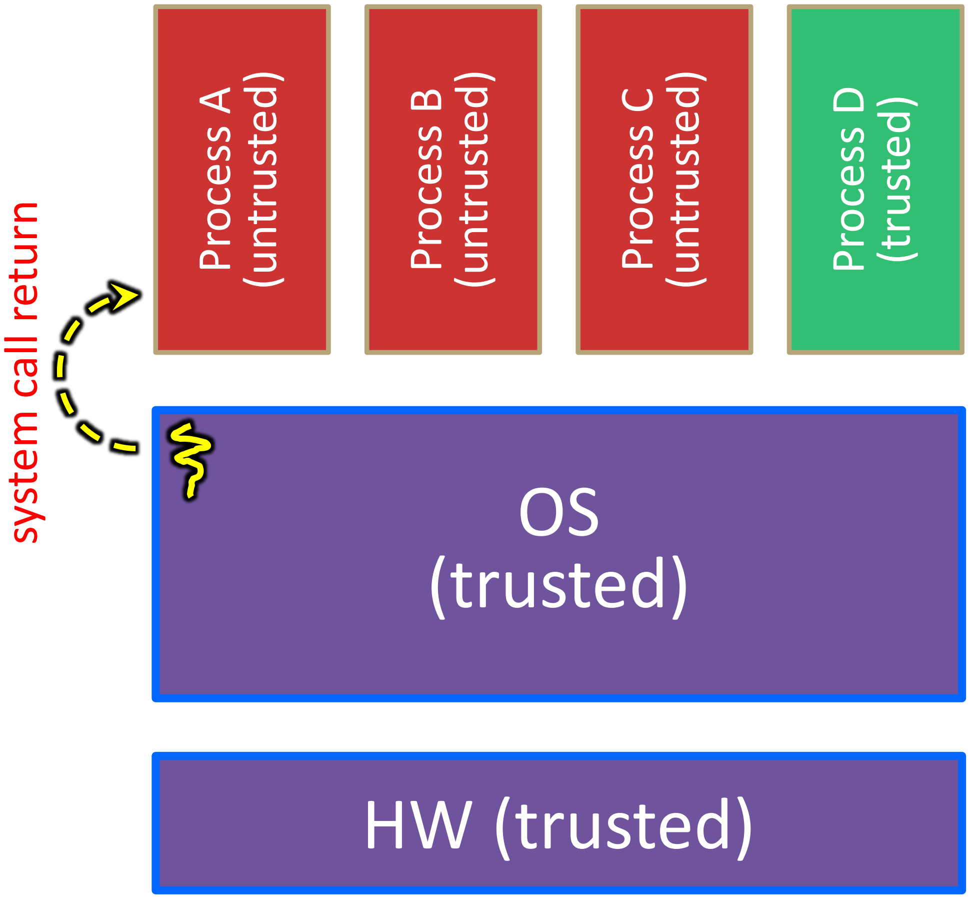


System Call Trace (high-level view)

Once the OS has finished servicing the system call, which might involve long waits as it interacts with HW, it:

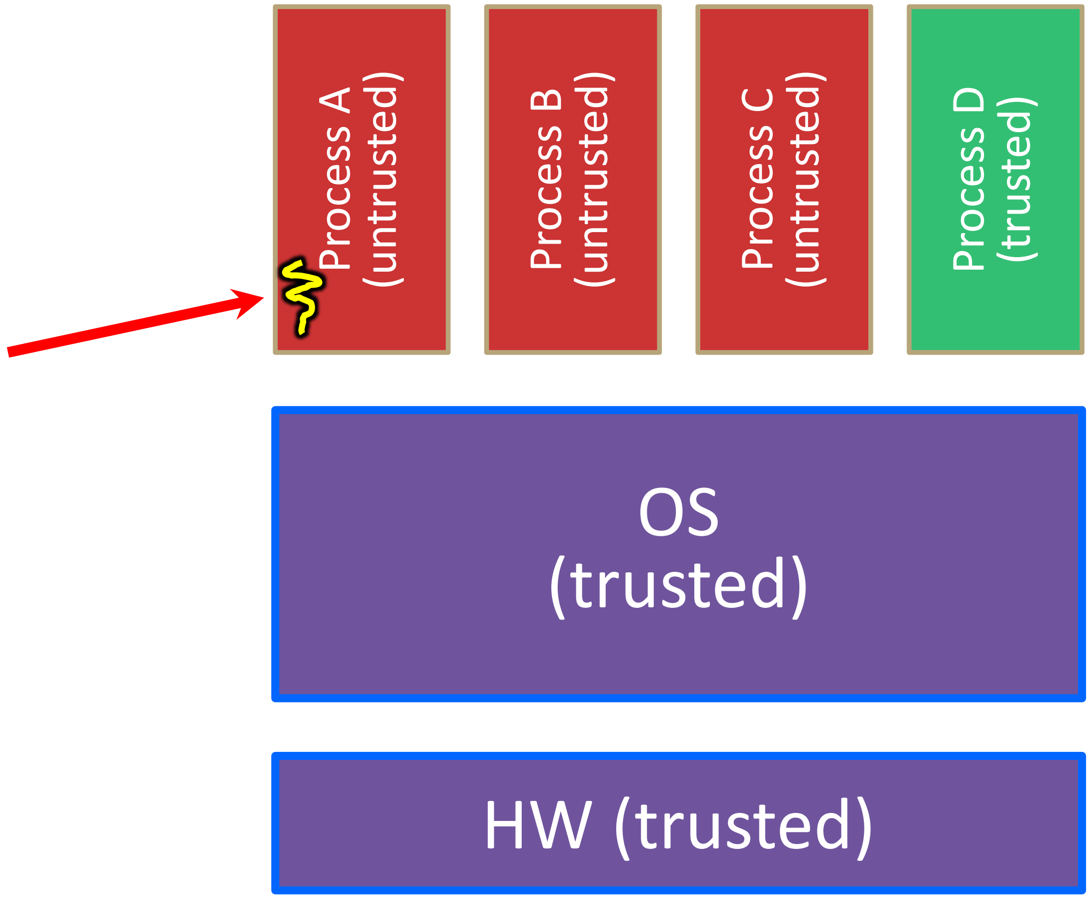
(1) Sets the CPU back to unprivileged mode and

(2) Returns out of the system call back to the user-level code in Process A.



System Call Trace (high-level view)

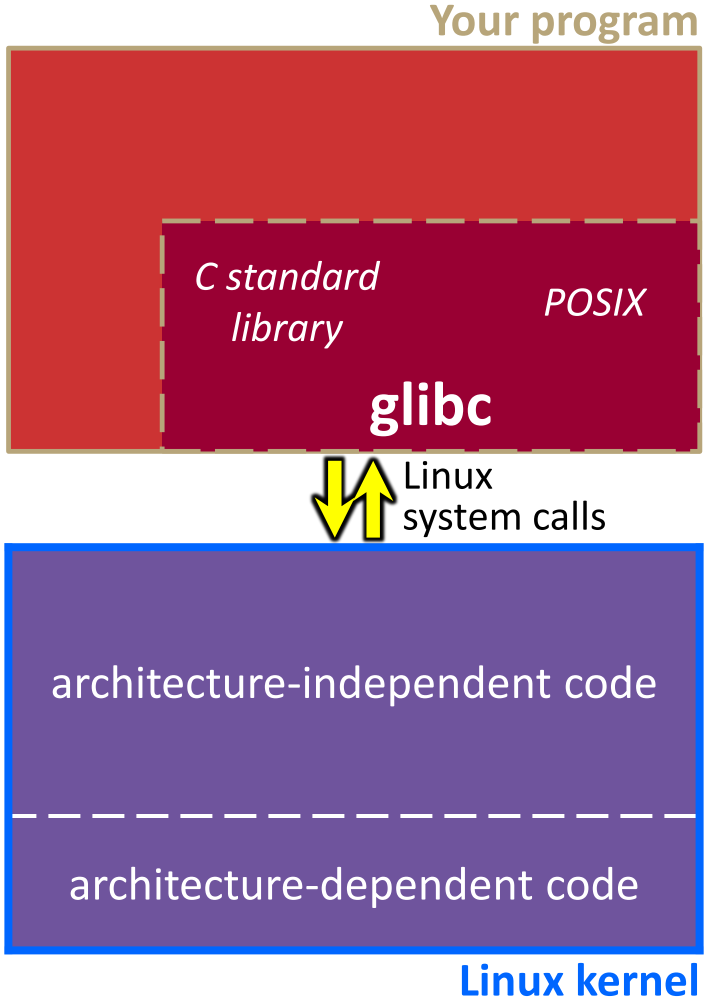
The process continues executing whatever code is next after the system call invocation.



Useful reference:
CSPP § 8.1–8.3
(the 351 book)

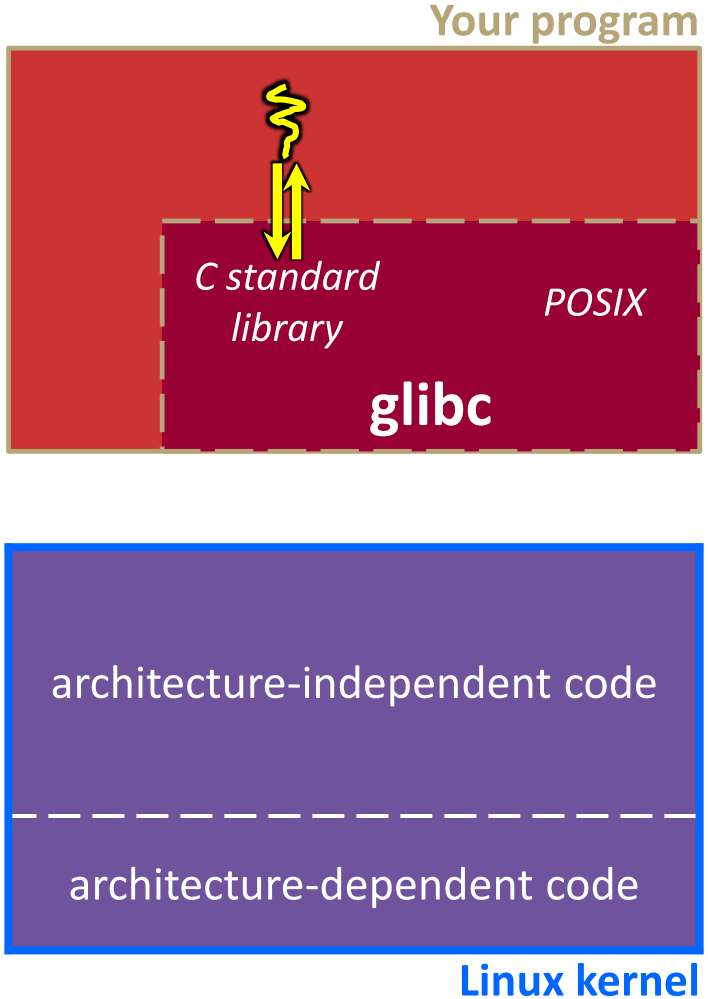
"Library calls" on x86/Linux

- 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



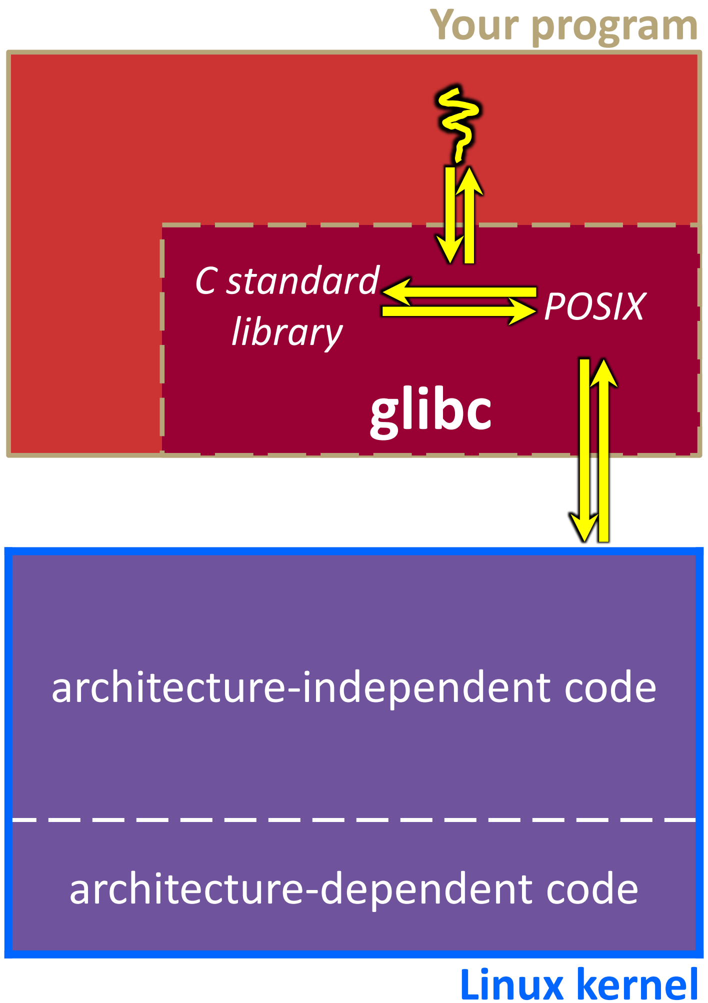
"Library calls" on x86/Linux: Option 1

- 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!



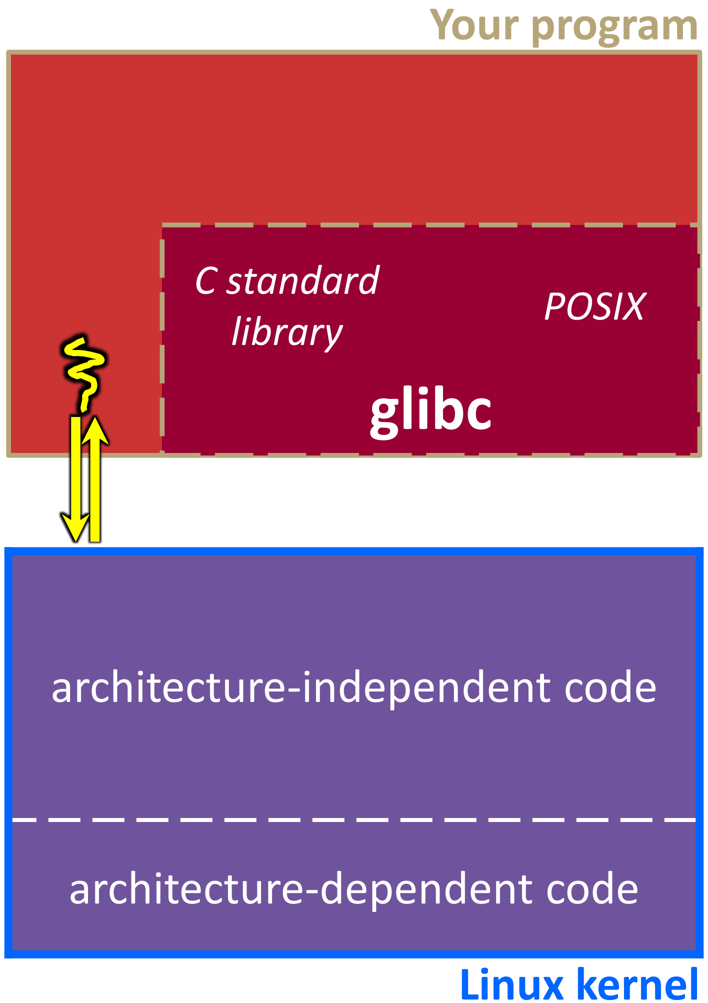
"Library calls" on x86/Linux: Option 2

- 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.



"Library calls" on x86/Linux: Option 3

- 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



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 ...
```

Lecture Outline

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

`make`

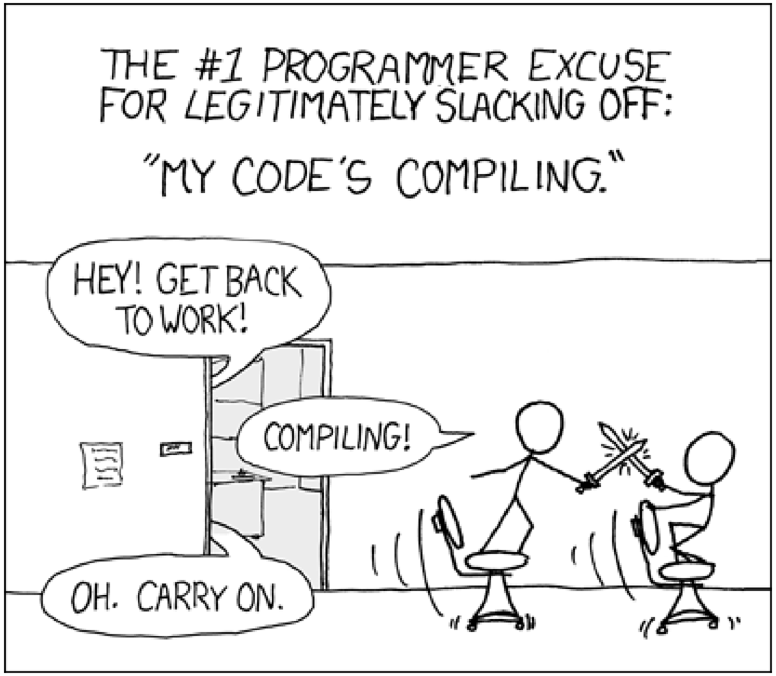
- `make` is a classic program for controlling what gets (re)compiled and how
  - Many other such programs exist (*e.g.*, `ant`, `maven`, IDE "projects")

- `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

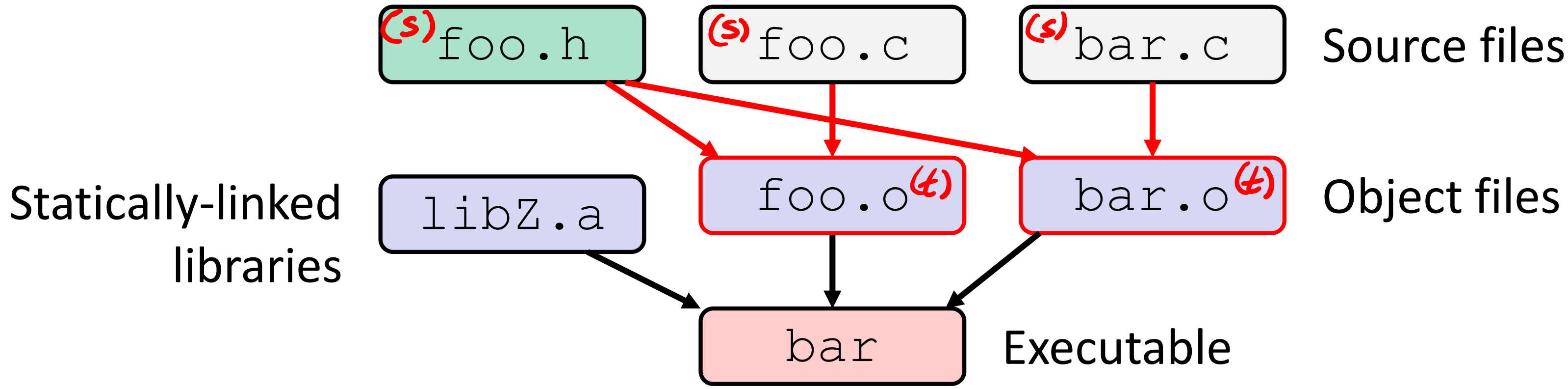
Recompilation Management

- 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 $s_1, s_2, \ldots, s_n$ 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"!

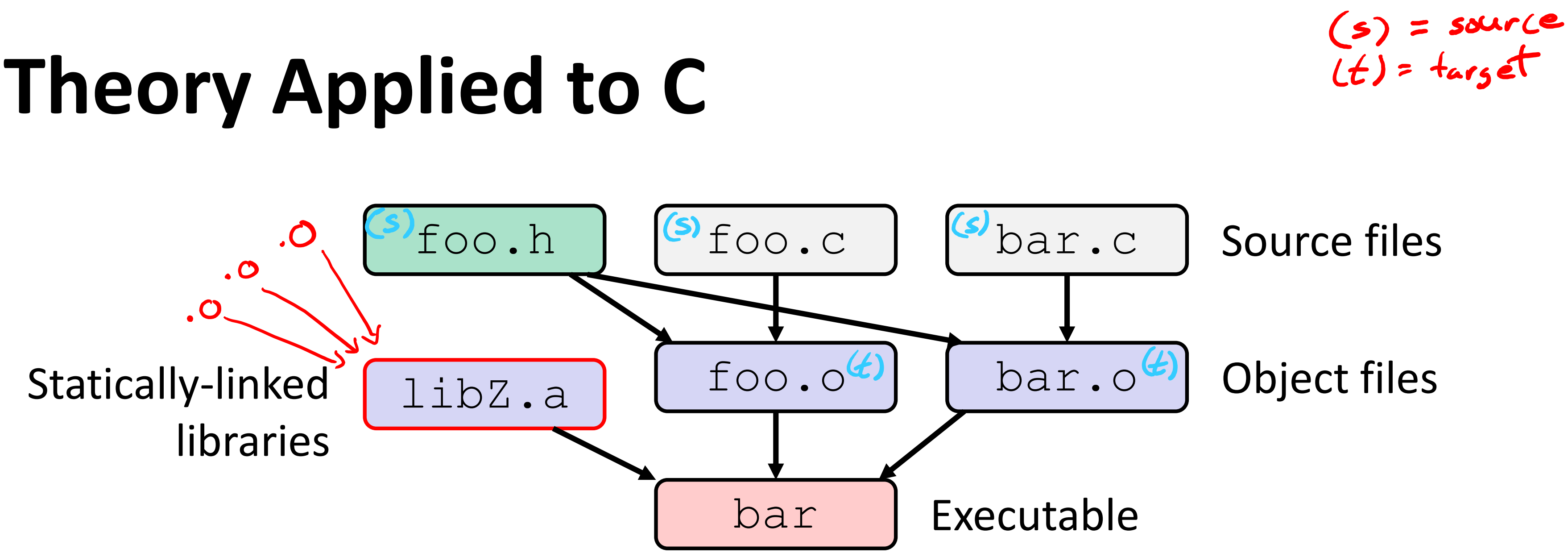
(s) = source
(t) = target

Theory Applied to C



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

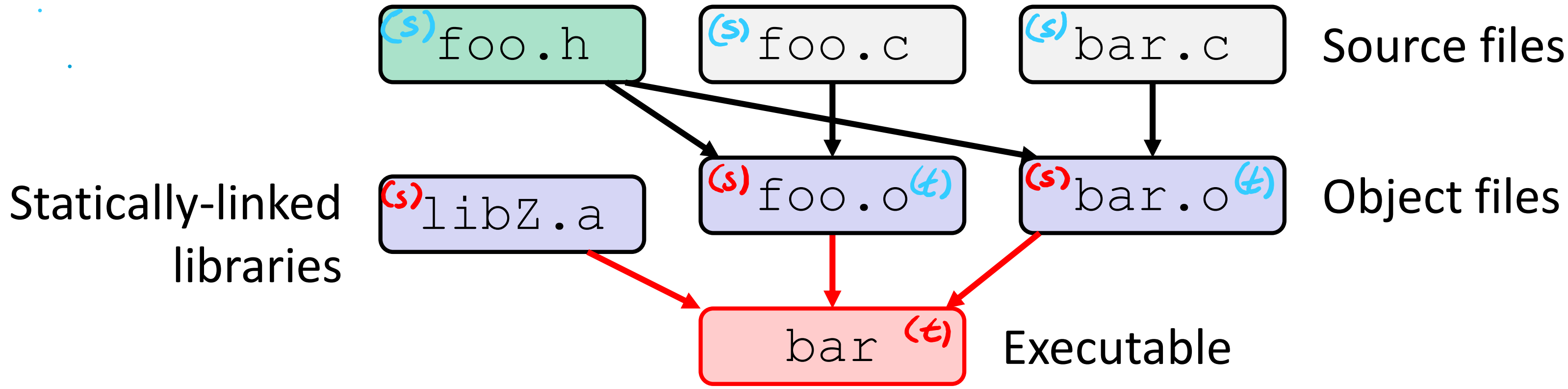
Theory Applied to C



- 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

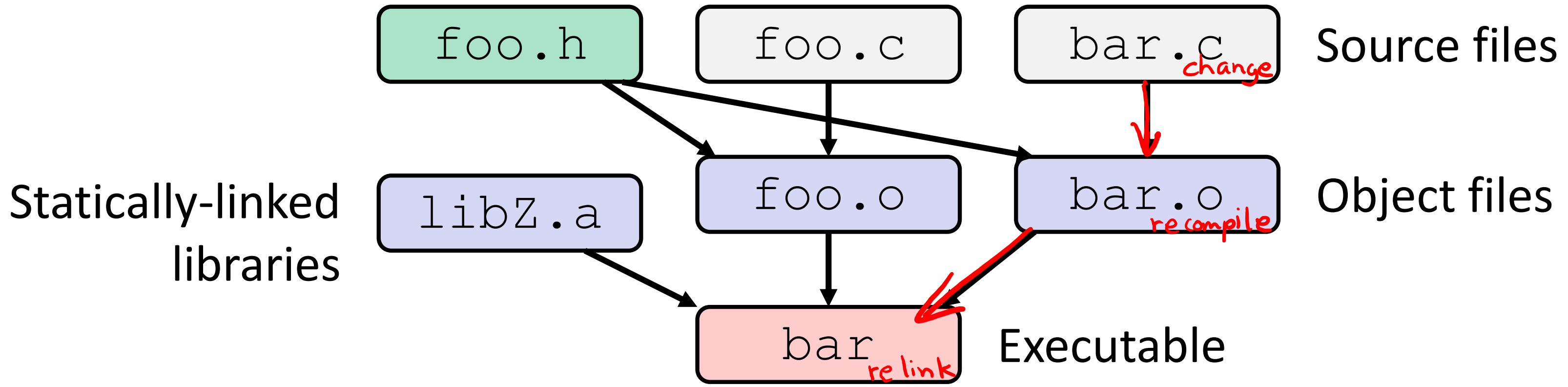
(s) = source
(t) = target

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

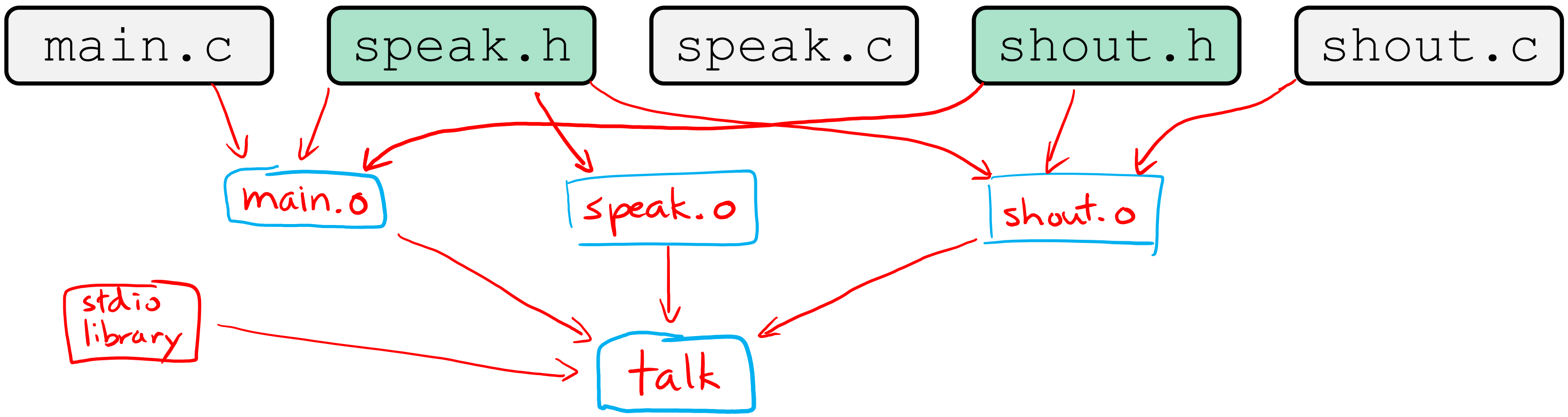
- *When creating a Makefile, first draw the dependencies!!!!*

- 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

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

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

speak.c

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

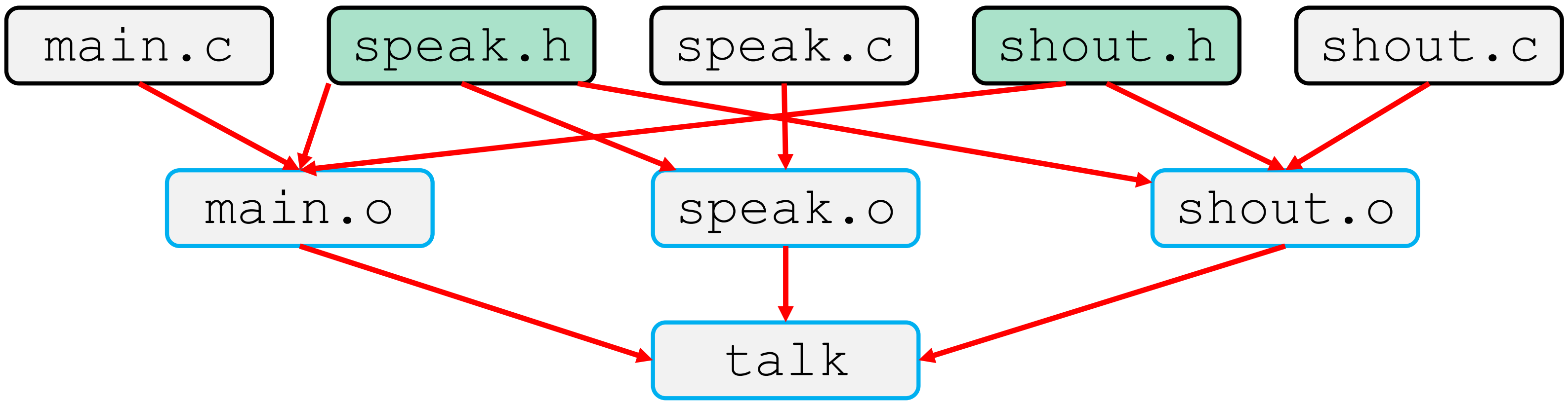
shout.c

```
#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
```

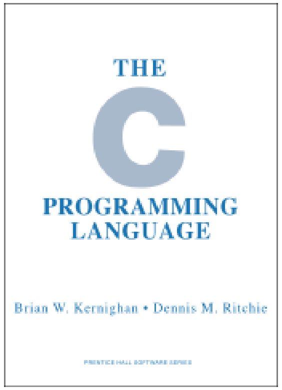
- Remember that you can put *any* shell command – even whole scripts!
- You can repeat target names to add more dependencies
- Often this stuff is more useful for reading makefiles than writing your own (until some day…)

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."