

pollev.com/cse333

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

Systems Programming

Buffering, POSIX I/O, System Calls

Instructor: Justin Hsia

Teaching Assistants:

Aadi Adgaonkar

Advaith Vijayakumar

Alice Fu

Andrew Vlahovich

Aviel Wood

Camden Harris

Esha Gore

Hannah Hempstead

Helena Wang

Laura Wang

Megan Frisella

Nina Tharamal

Shrihun Sankepally

Shriya Sateesh

Yusong Yan

Relevant Course Information

- ❖ Exercise 2 grades released
 - Style grading will get stricter: some minor issues will be upgraded to major
- ❖ Exercise 4 posted today, due *next Wednesday* (10/21)
 - Given extra time because HW1 is due and hard(est?) exercise
- ❖ Homework 1 due Thursday night (10/15)
 - Clean up “to do” comments, but leave “STEP #” markers
 - Graded not just on correctness, also code quality (50/50) !
 - Late days counted based on tag commit time; weekend is one day
- ❖ Homework 2 released on Friday

Lecture Outline (1/3)

- ❖ **C Stream Buffering**
- ❖ POSIX Lower-Level I/O
- ❖ System Calls (High-Level View)

Buffering

- ❖ By default, `stdio` uses **buffering** for streams:
 - Data written by **`fwrite()`** is copied into a buffer allocated by `stdio` inside your process' address space
 - As some point, the buffer will be “drained” into the destination:
 - When you explicitly call **`fflush()`** on the stream
 - When the buffer size is exceeded (often 1024 or 4096 bytes)
 - For `stdout` to console, when a newline is written (*“line buffered”*) or when some other function tries to read from the console
 - When you call **`fclose()`** on the stream
 - When your process exits gracefully (**`exit()`** or **`return`** from **`main()`**)

Buffering Example (1/5)

```
int main(int argc, char* argv[]) {  
    → FILE* fout = fopen("test.txt", "wb");  
  
    // write "hi" one char at a time  
    if (fwrite("h", sizeof(char), 1, fout) < 1) {  
        perror("fwrite failed");  
        fclose(fout);  
        return EXIT_FAILURE;  
    }  
  
    if (fwrite("i", sizeof(char), 1, fout) < 1) {  
        perror("fwrite failed");  
        fclose(fout);  
        return EXIT_FAILURE;  
    }  
  
    fclose(fout);  
    return EXIT_SUCCESS;  
}
```

C stdio buffer

test.txt (disk)

buffered_hi.c

Buffering Example (2/5)

```
int main(int argc, char* argv[]) {  
    FILE* fout = fopen("test.txt", "wb");  
  
    // write "hi" one char at a time  
    → if (fwrite("h", sizeof(char), 1, fout) < 1) {  
        perror("fwrite failed");  
        fclose(fout);  
        return EXIT_FAILURE;  
    }  
  
    if (fwrite("i", sizeof(char), 1, fout) < 1) {  
        perror("fwrite failed");  
        fclose(fout);  
        return EXIT_FAILURE;  
    }  
  
    fclose(fout);  
    return EXIT_SUCCESS;  
}
```

C stdio buffer



test.txt (disk)

buffered_hi.c

Buffering Example (3/5)

```
int main(int argc, char* argv[]) {
    FILE* fout = fopen("test.txt", "wb");

    // write "hi" one char at a time
    if (fwrite("h", sizeof(char), 1, fout) < 1) {
        perror("fwrite failed");
        fclose(fout);
        return EXIT_FAILURE;
    }

    → if (fwrite("i", sizeof(char), 1, fout) < 1) {
        perror("fwrite failed");
        fclose(fout);
        return EXIT_FAILURE;
    }

    fclose(fout);
    return EXIT_SUCCESS;
}
```

C stdio buffer



test.txt (disk)

buffered_hi.c

Buffering Example (4/5)

```
int main(int argc, char* argv[]) {  
    FILE* fout = fopen("test.txt", "wb");  
  
    // write "hi" one char at a time  
    if (fwrite("h", sizeof(char), 1, fout) < 1) {  
        perror("fwrite failed");  
        fclose(fout);  
        return EXIT_FAILURE;  
    }  
  
    if (fwrite("i", sizeof(char), 1, fout) < 1) {  
        perror("fwrite failed");  
        fclose(fout);  
        return EXIT_FAILURE;  
    }  
  
    → fclose(fout);  
    return EXIT_SUCCESS;  
}
```

C stdio buffer

'h'	'i'	...
-----	-----	-----

test.txt (disk)

buffered_hi.c

Buffering Example (5/5)

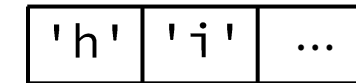
```
int main(int argc, char* argv[]) {
    FILE* fout = fopen("test.txt", "wb");

    // write "hi" one char at a time
    if (fwrite("h", sizeof(char), 1, fout) < 1) {
        perror("fwrite failed");
        fclose(fout);
        return EXIT_FAILURE;
    }

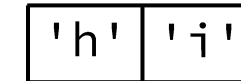
    if (fwrite("i", sizeof(char), 1, fout) < 1) {
        perror("fwrite failed");
        fclose(fout);
        return EXIT_FAILURE;
    }

    fclose(fout);
    return EXIT_SUCCESS;
}
```

C stdio buffer



test.txt (disk)



buffered_hi.c

No Buffering Example (1/5)

```
int main(int argc, char* argv[]) {  
    → FILE* fout = fopen("test.txt", "wb");  
    setbuf(fout, NULL); // turn off buffering  
  
    // write "hi" one char at a time  
    if (fwrite("h", sizeof(char), 1, fout) < 1) {  
        perror("fwrite failed");  
        fclose(fout);  
        return EXIT_FAILURE;  
    }  
  
    if (fwrite("i", sizeof(char), 1, fout) < 1) {  
        perror("fwrite failed");  
        fclose(fout);  
        return EXIT_FAILURE;  
    }  
  
    fclose(fout);  
    return EXIT_SUCCESS;  
}
```

C stdio buffer

test.txt (disk)

unbuffered_hi.c

No Buffering Example (2/5)

```
int main(int argc, char* argv[]) {  
    FILE* fout = fopen("test.txt", "wb");  
    → setbuf(fout, NULL); // turn off buffering  
  
    // write "hi" one char at a time  
    if (fwrite("h", sizeof(char), 1, fout) < 1) {  
        perror("fwrite failed");  
        fclose(fout);  
        return EXIT_FAILURE;  
    }  
  
    if (fwrite("i", sizeof(char), 1, fout) < 1) {  
        perror("fwrite failed");  
        fclose(fout);  
        return EXIT_FAILURE;  
    }  
  
    fclose(fout);  
    return EXIT_SUCCESS;  
}
```

C stdio buffer



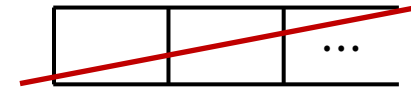
test.txt (disk)

unbuffered_hi.c

No Buffering Example (3/5)

```
int main(int argc, char* argv[]) {  
    FILE* fout = fopen("test.txt", "wb");  
    setbuf(fout, NULL); // turn off buffering  
  
    // write "hi" one char at a time  
    → if (fwrite("h", sizeof(char), 1, fout) < 1) {  
        perror("fwrite failed");  
        fclose(fout);  
        return EXIT_FAILURE;  
    }  
  
    if (fwrite("i", sizeof(char), 1, fout) < 1) {  
        perror("fwrite failed");  
        fclose(fout);  
        return EXIT_FAILURE;  
    }  
  
    fclose(fout);  
    return EXIT_SUCCESS;  
}
```

C stdio buffer



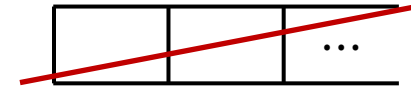
test.txt (disk)

unbuffered_hi.c

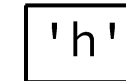
No Buffering Example (4/5)

```
int main(int argc, char* argv[]) {  
    FILE* fout = fopen("test.txt", "wb");  
    setbuf(fout, NULL); // turn off buffering  
  
    // write "hi" one char at a time  
    if (fwrite("h", sizeof(char), 1, fout) < 1) {  
        perror("fwrite failed");  
        fclose(fout);  
        return EXIT_FAILURE;  
    }  
  
    → if (fwrite("i", sizeof(char), 1, fout) < 1) {  
        perror("fwrite failed");  
        fclose(fout);  
        return EXIT_FAILURE;  
    }  
  
    fclose(fout);  
    return EXIT_SUCCESS;  
}
```

C stdio buffer



test.txt (disk)

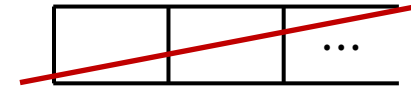


unbuffered_hi.c

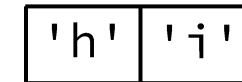
No Buffering Example (5/5)

```
int main(int argc, char* argv[]) {  
    FILE* fout = fopen("test.txt", "wb");  
    setbuf(fout, NULL); // turn off buffering  
  
    // write "hi" one char at a time  
    if (fwrite("h", sizeof(char), 1, fout) < 1) {  
        perror("fwrite failed");  
        fclose(fout);  
        return EXIT_FAILURE;  
    }  
  
    if (fwrite("i", sizeof(char), 1, fout) < 1) {  
        perror("fwrite failed");  
        fclose(fout);  
        return EXIT_FAILURE;  
    }  
  
    → fclose(fout);  
    return EXIT_SUCCESS;  
}
```

C stdio buffer



test.txt (disk)



unbuffered_hi.c

Why Buffer?

❖ Performance – avoid disk accesses

- Group many small writes into a single larger write:
- Disk Latency = 🤯 🤯 🤯
(Jeff Dean from LADIS '09)

❖ Convenience – nicer API

- We'll compare C's **fread()** with POSIX's **read()**

*orders of magnitude
still accurate!*



Numbers Everyone Should Know

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	3,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

Why NOT Buffer?

- ❖ Reliability – the buffer needs to be flushed
 - Loss of computer power = loss of data
 - Writing to a buffer (*i.e.*, return from **fwrite**()) does not mean the data has actually been written to the file/console
 - Segfaults leave buffered data unflushed
- ❖ Performance – buffering takes time
 - Copying data into the `stdio` buffer consumes CPU cycles and memory bandwidth
 - Can potentially slow down high-performance applications, like a web server or database (“zero-copy”)
- ❖ When is buffering faster? / Slower?
 - many small writes* / *large writes*

Lecture Outline (2/3)

- ❖ C Stream Buffering
- ❖ **POSIX Lower-Level I/O**
- ❖ System Calls (High-Level View)

We Need To Go Deeper...



- ❖ So far we've seen the C standard library to access files
 - Use a provided FILE* *stream* abstraction
 - **fopen()**, **fread()**, **fwrite()**, **fclose()**, **fseek()**
- ❖ These are convenient and portable
 - They are buffered (by default, can be disabled)
 - They are implemented using lower-level OS calls

From C to POSIX

- ❖ Most UNIX-like OS support a common set of lower-level 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 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);
```

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

Reading from a File

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

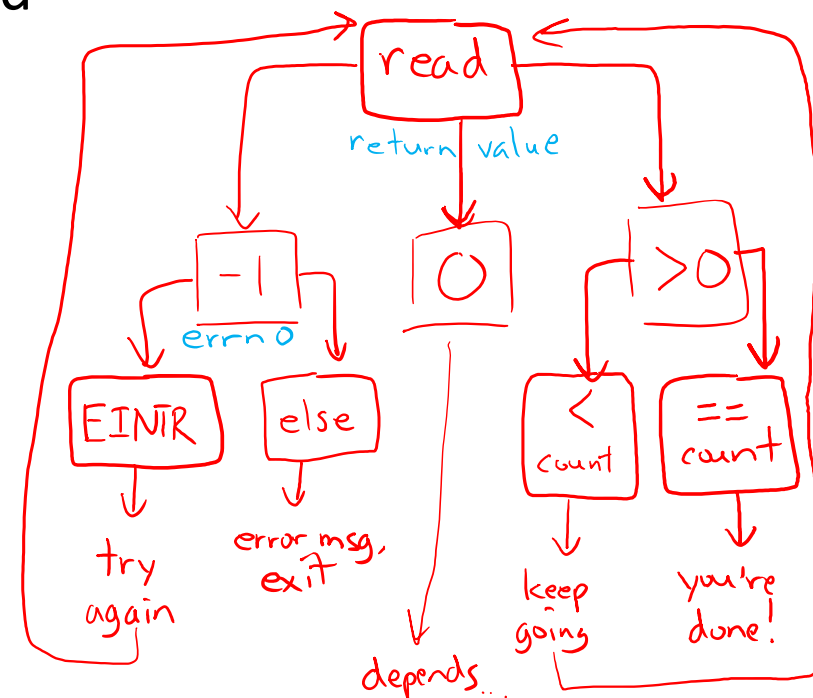
try to read count bytes

- 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`)

errno.h

- `EBADF`: bad file descriptor
- `EFAULT`: output buffer is not a valid address
- `EINTR`: read was interrupted, please try again (ARG! 😡)
- And many others...



pollev.com/cse333

We want to read 'n' bytes. Which is the correct completion of the blank below?

```
char* buf = ...; // buffer of size n
int bytes_left = n;
int result;      // result of read()

while (bytes_left > 0) {
    result = read(fd, _____, 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;
    }
    bytes_left -= result;
}
```

if first read only got $n/3$ bytes:
 $bytes_left = 2n/3$



The diagram shows a horizontal rectangle representing a buffer. The left portion of the rectangle is filled with diagonal hatching. An arrow points from the text "Want next read to start writing here" to the boundary between the hatched area and the empty area on the right.

buf

- A. buf
- B. buf + bytes_left
- C. buf + bytes_left - n
- D. buf + n - bytes_left**
- E. We're lost...

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 disk cache

- `#include <unistd.h>`

- ★ **opendir()**, **readdir()**, **closedir()** – deal with directory listings

- Make sure you read the section 3 version (*e.g.*, `man 3 opendir`)

- Go to section tomorrow to learn more!

- `#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

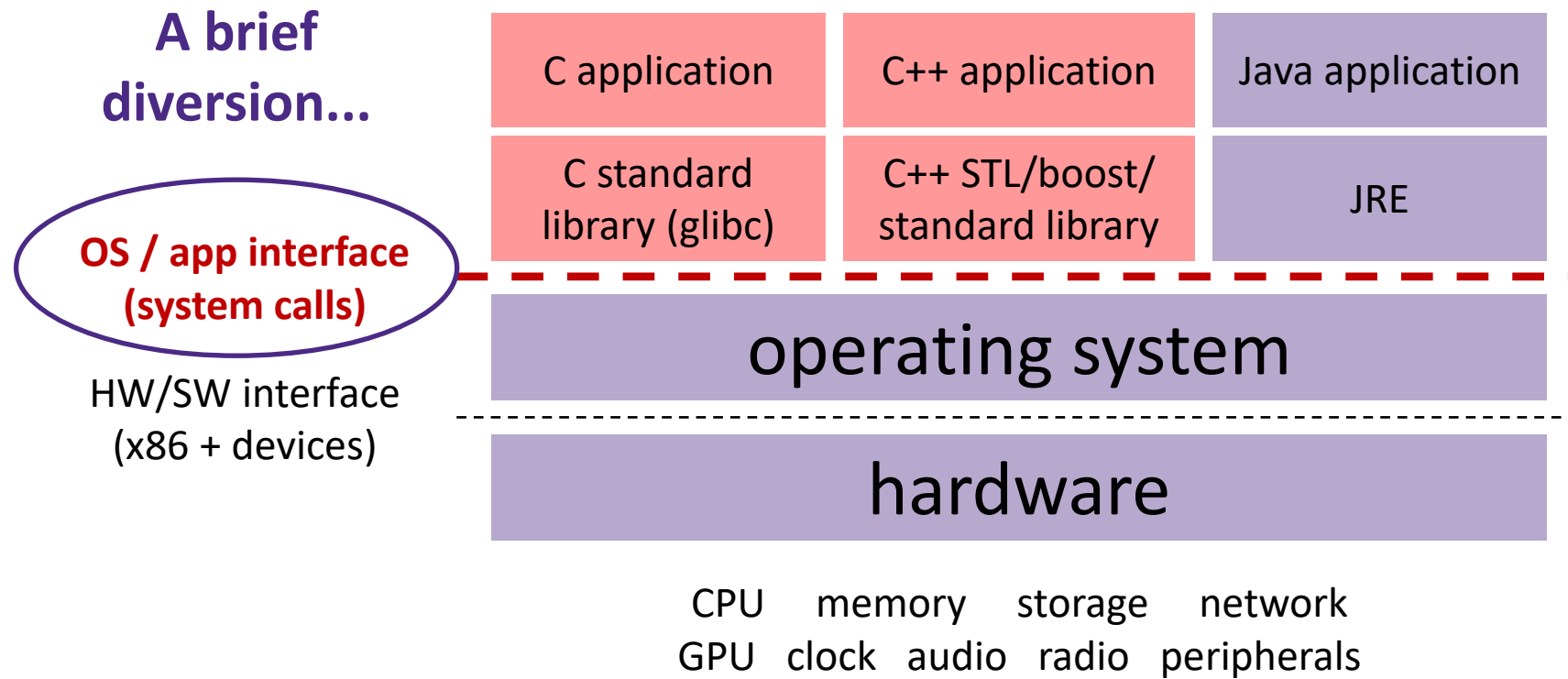
- ❖ C standard library implements a subset of POSIX
 - *e.g.*, POSIX provides directory manipulation that C std lib doesn't
- ❖ C standard library implements automatic buffering
- ❖ C standard library has a nicer API

- ❖ 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 (3/3)

- ❖ C Stream Buffering
- ❖ POSIX Lower-Level I/O
- ❖ **System Calls (High-Level View)**

Remember This Picture?



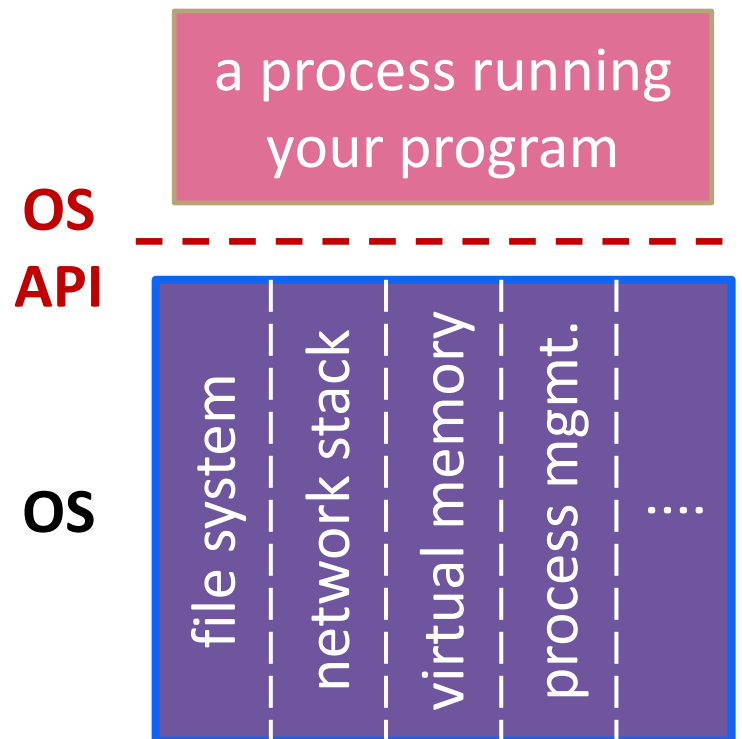
What's an OS?

❖ 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

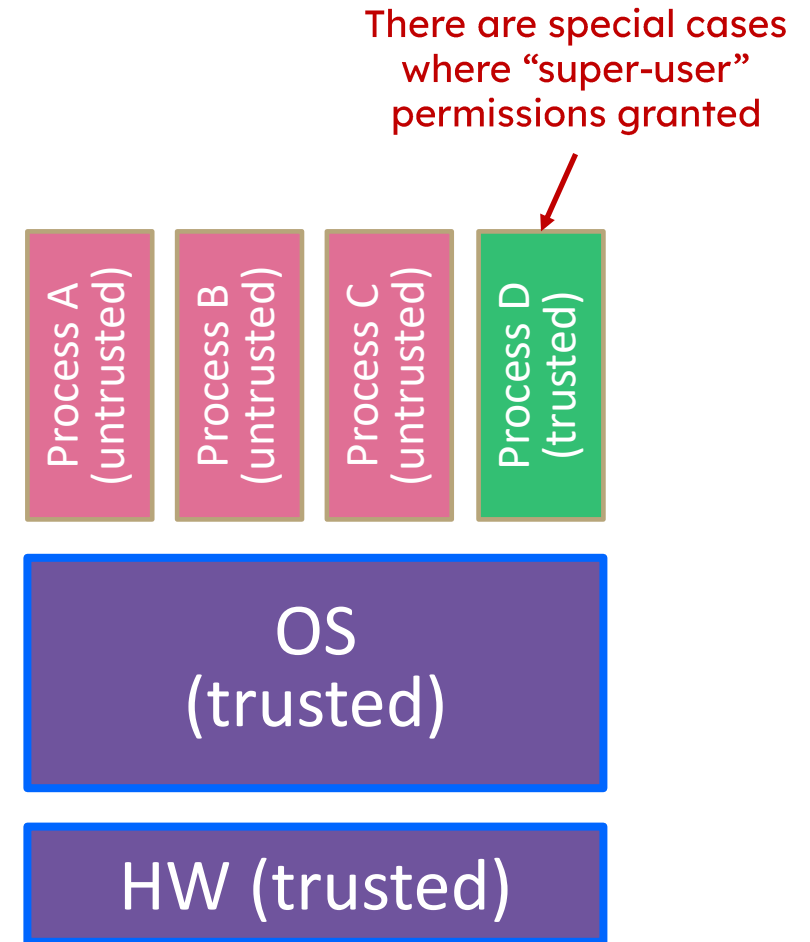
- `brk()`, `shm_open()`, ...

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

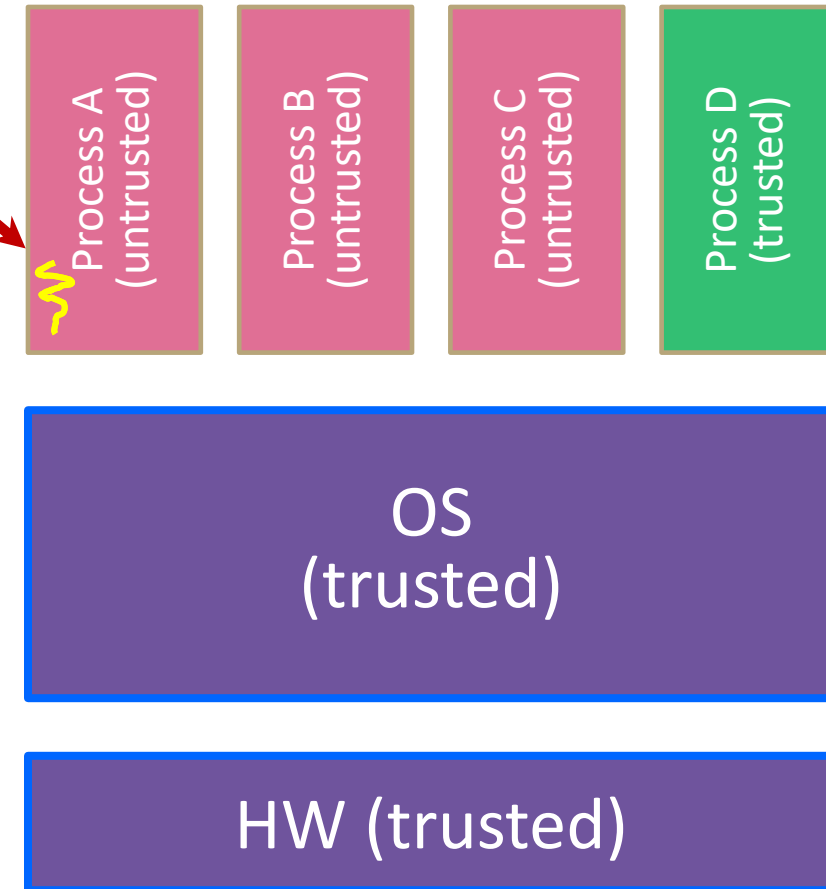


System Calls Simplified Overview

- ❖ 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

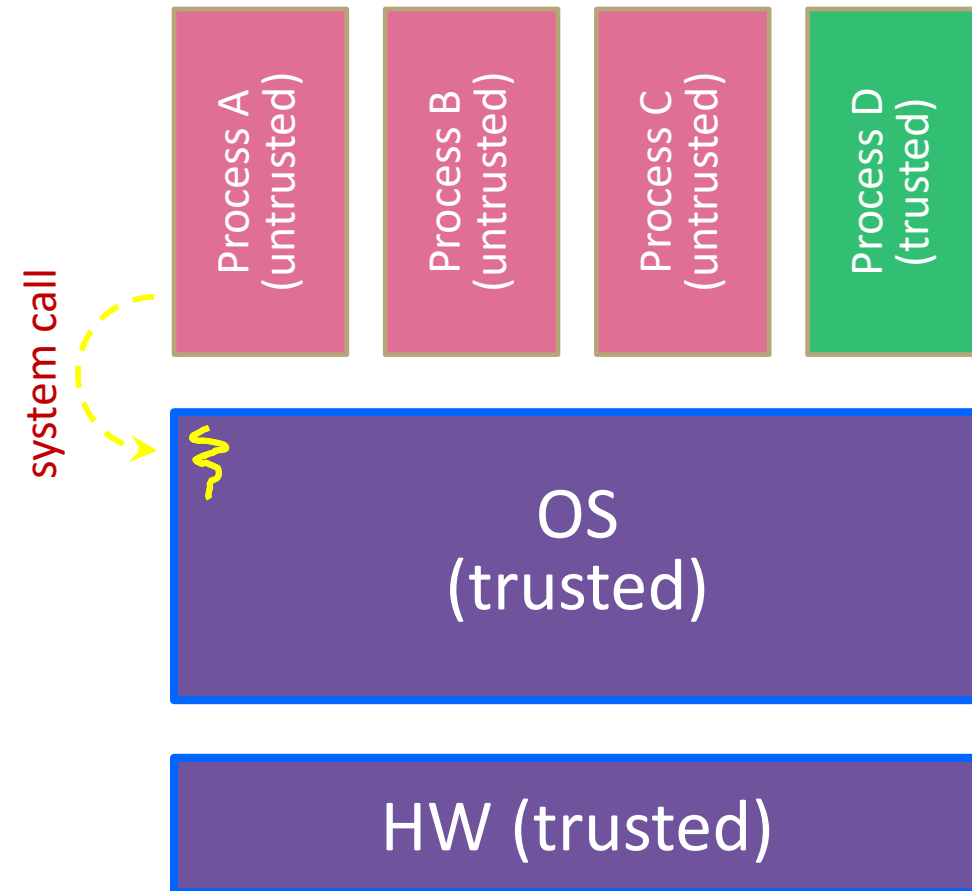
System Call Trace (high-level view, 1/5)

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



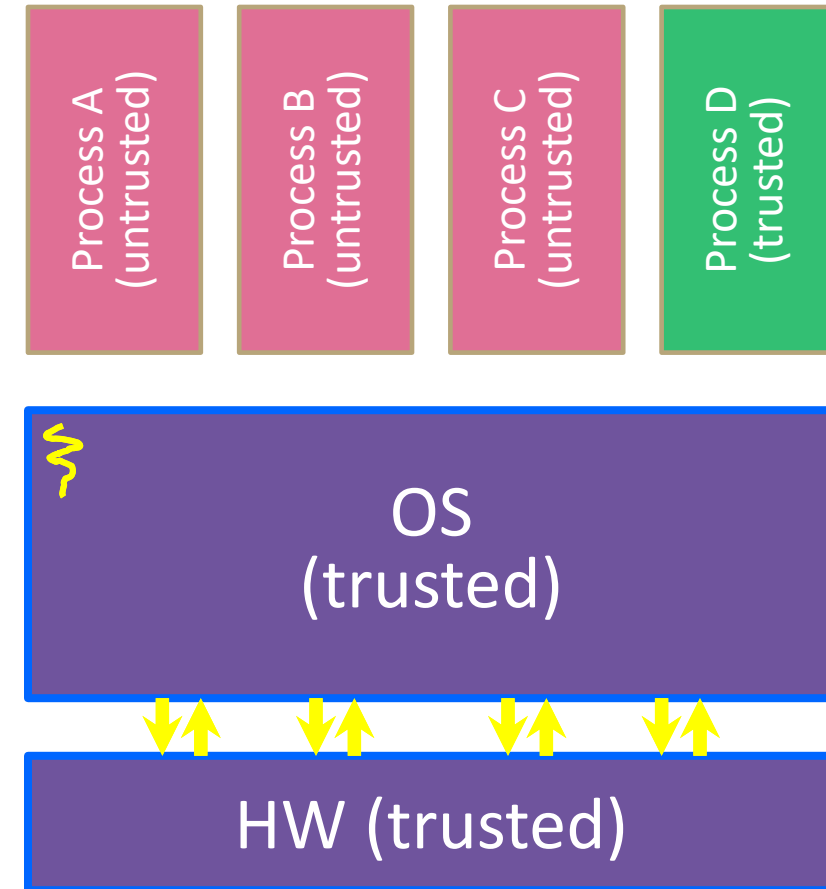
System Call Trace (high-level view, 2/5)

- ❖ 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, 3/5)

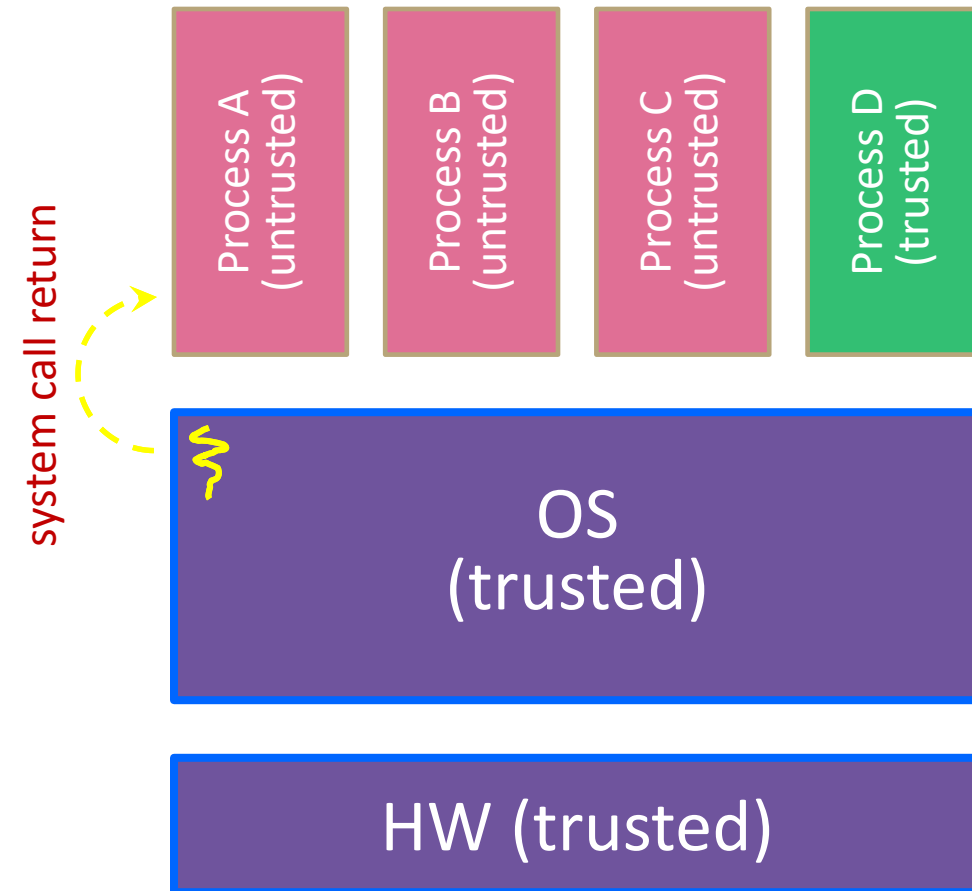
- ❖ Because the CPU executing the thread that's in the OS is in privileged mode, it is able to use *privileged instructions* that interact directly with hardware devices like disks



System Call Trace (high-level view, 4/5)

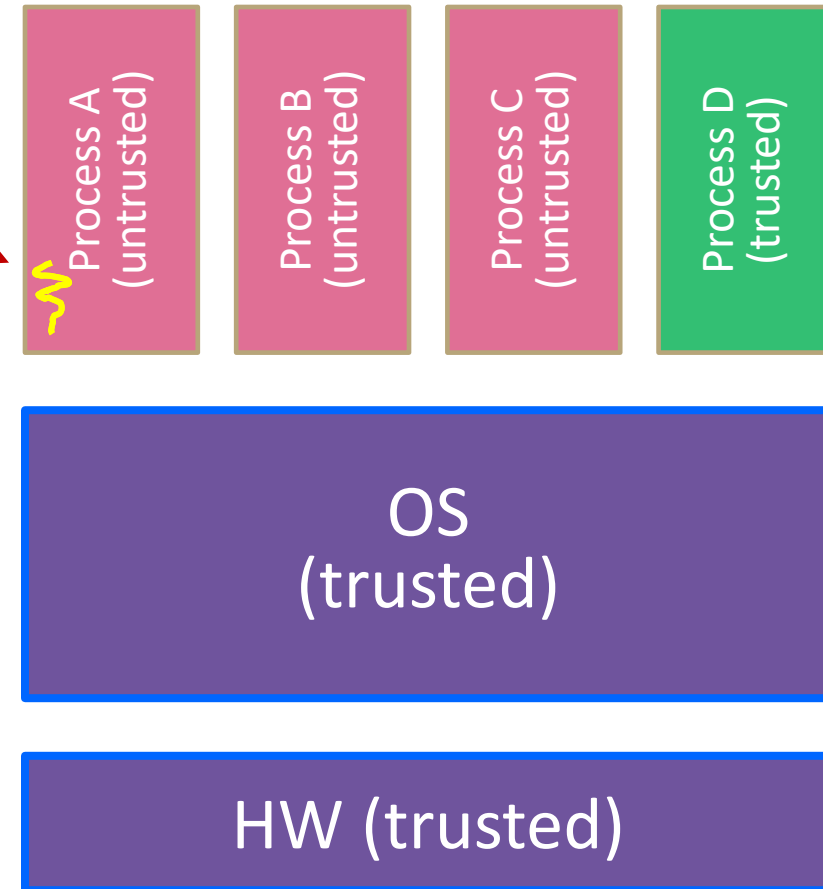
❖ Once the OS has finished servicing the system call, which might involve long waits as it interacts with HW, it:

- 1) Sets the CPU back to unprivileged mode
- 2) Returns out of the system call back to the user-level code in Process A



System Call Trace (high-level view, 5/5)

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

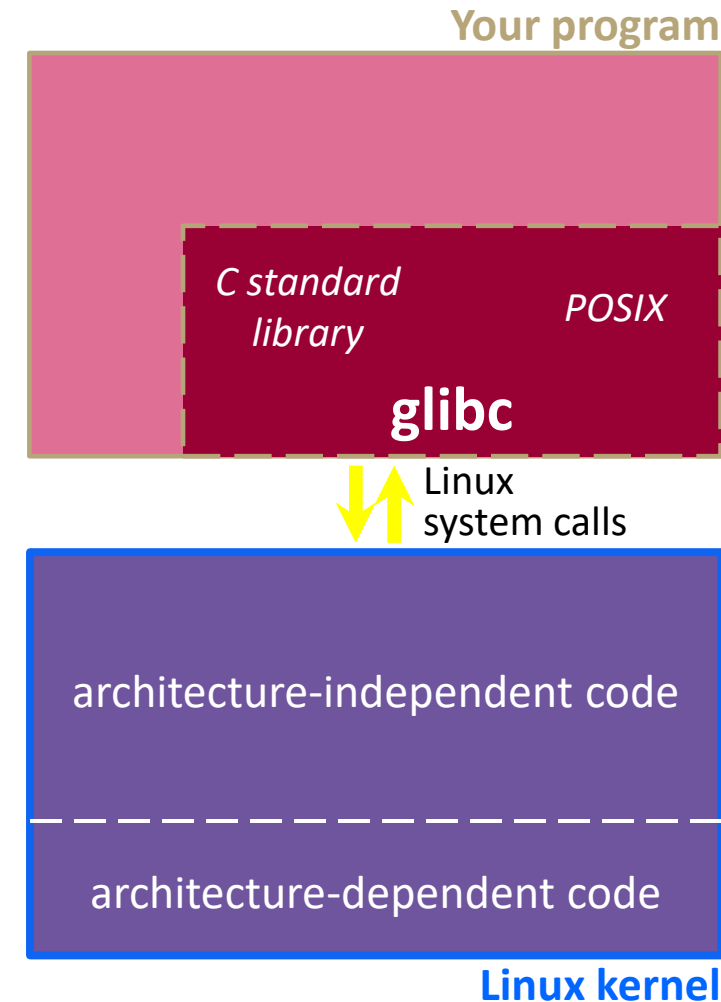


Useful reference:
CSPP § 8.1–8.3
(the 351 book)

“Library calls” on x86/Linux

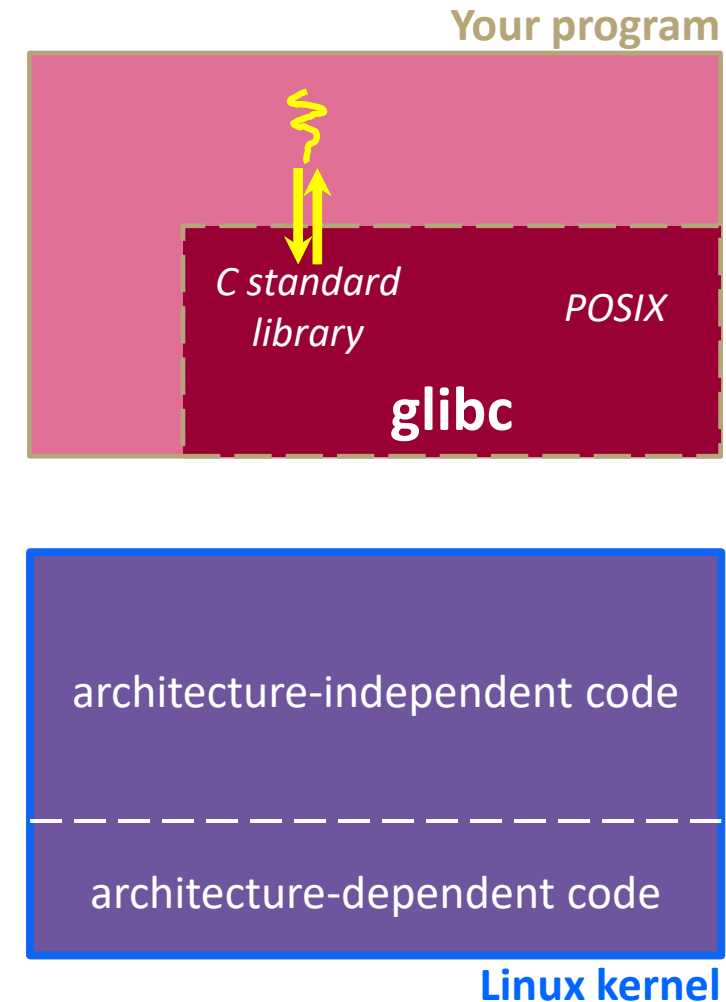
❖ A more accurate picture:

- 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



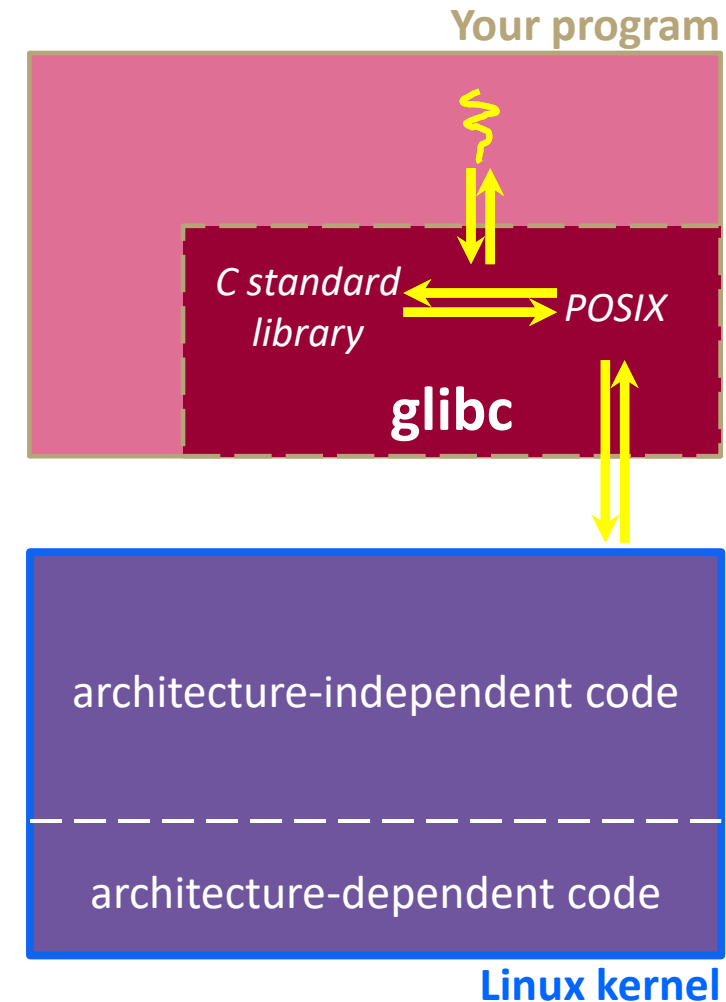
“Library calls” on x86/Linux: Option 1

- ❖ 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!



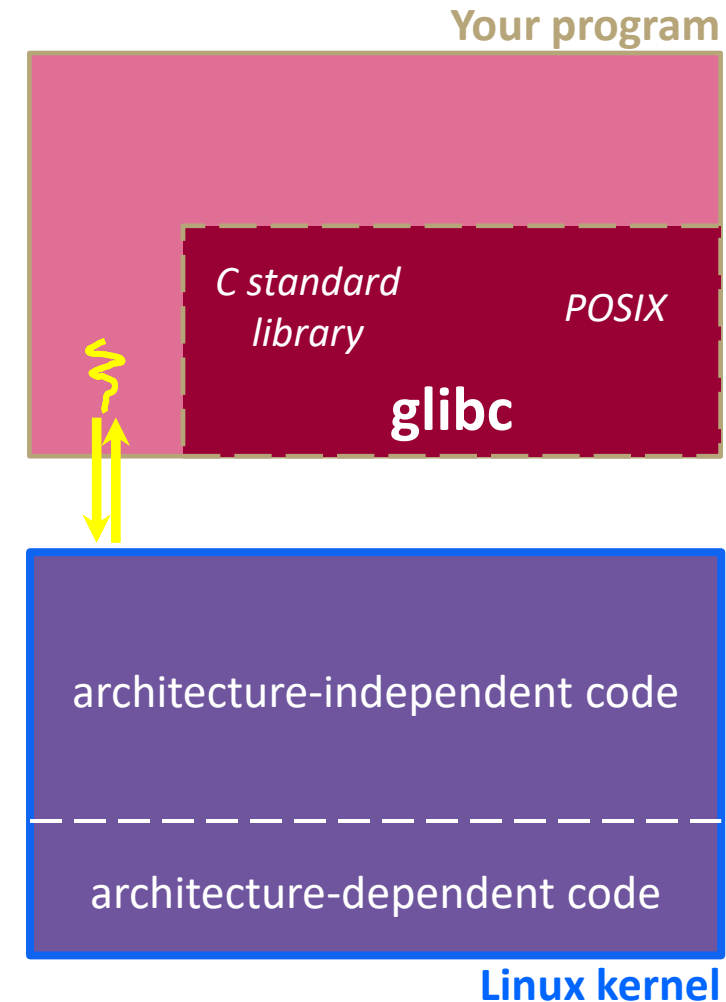
“Library calls” on x86/Linux: Option 2

- ❖ Some routines may be handled by `glibc`, but they in turn invoke Linux system calls
 - *e.g.*, POSIX wrappers around Linux syscalls
 - POSIX **`read`****`ir`**() invokes the underlying Linux **`read`****`ir`**()
 - *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:

pipe both stderr and stdout

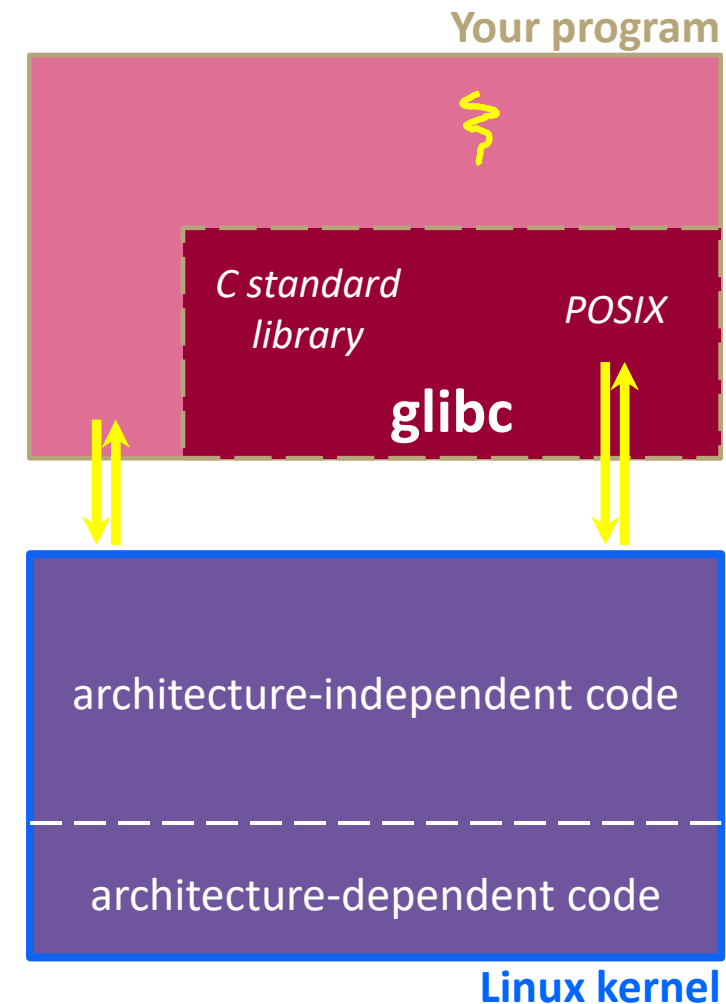
```
$ strace ls |& less
execve("/usr/bin/ls", ["ls"], [/* 41 vars */]) = 0
brk(NULL)                                = 0x15aa000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
    0x7f03bb741000
access("/etc/ld.so.preload", R_OK)        = -1 ENOENT (No such file or directory)
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"...
    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 ...
```

BONUS SLIDES

“Story time” about system calls on x86/Linux

Details on x86/Linux

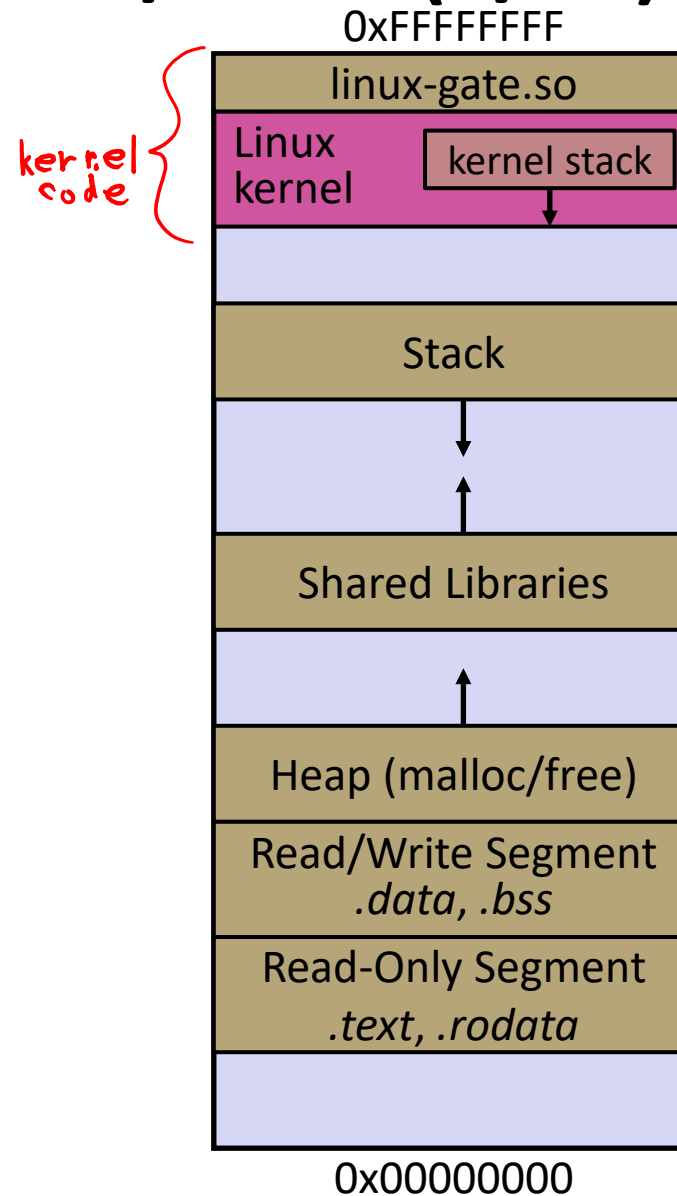
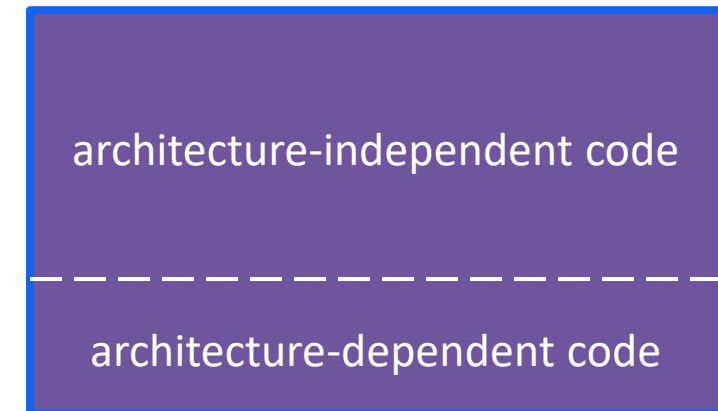
- ❖ Let's walk through how a Linux system call actually works
 - We'll assume *32-bit x86* using the modern SYSENTER / SYSEXIT x86 instructions
 - x86-64 code is similar, though details always change over time, so take this as an example – not a debugging guide



System Calls on x86/Linux (1/11)

❖ Remember our process address space picture?

- Let's add some details

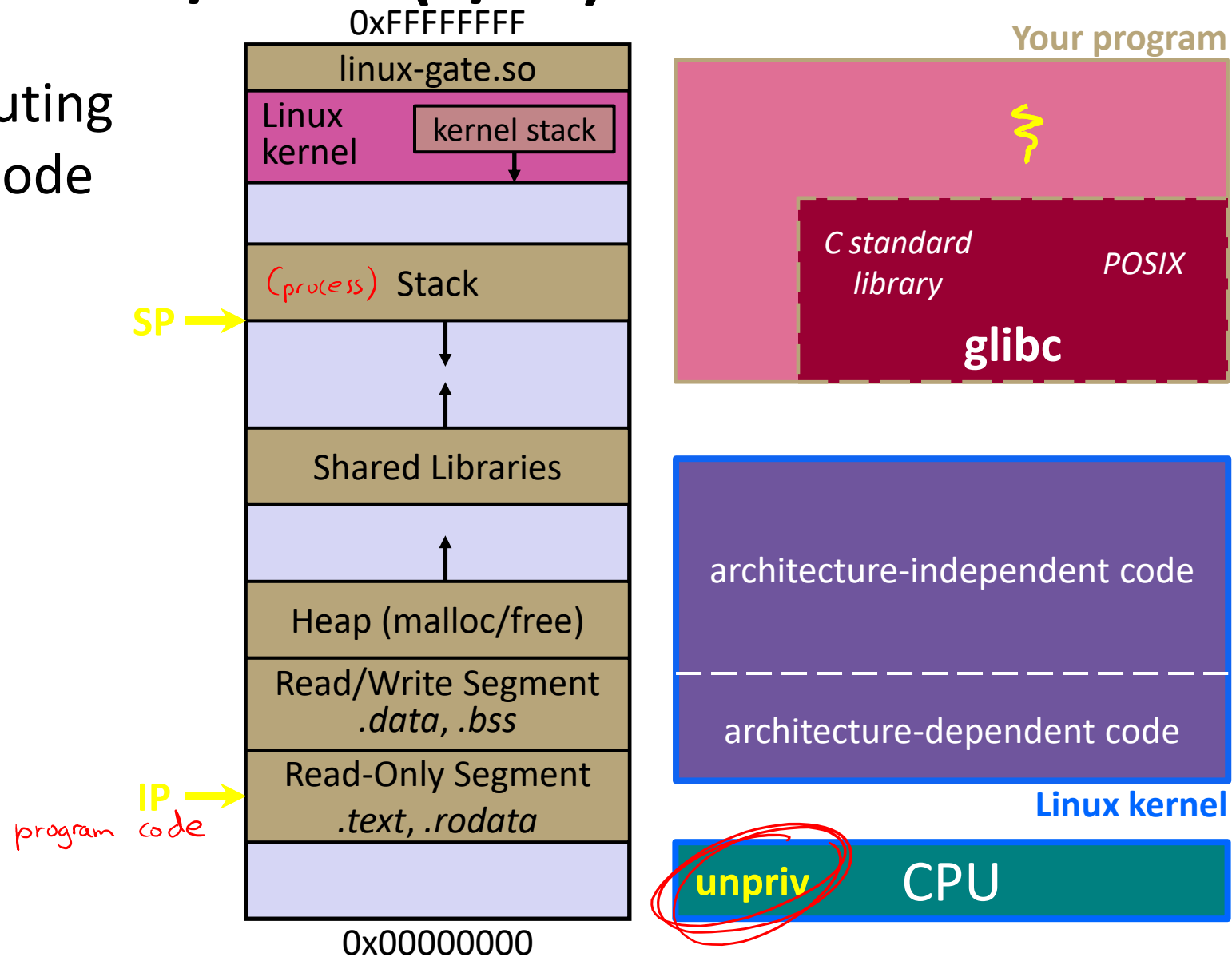


Linux kernel



System Calls on x86/Linux (2/11)

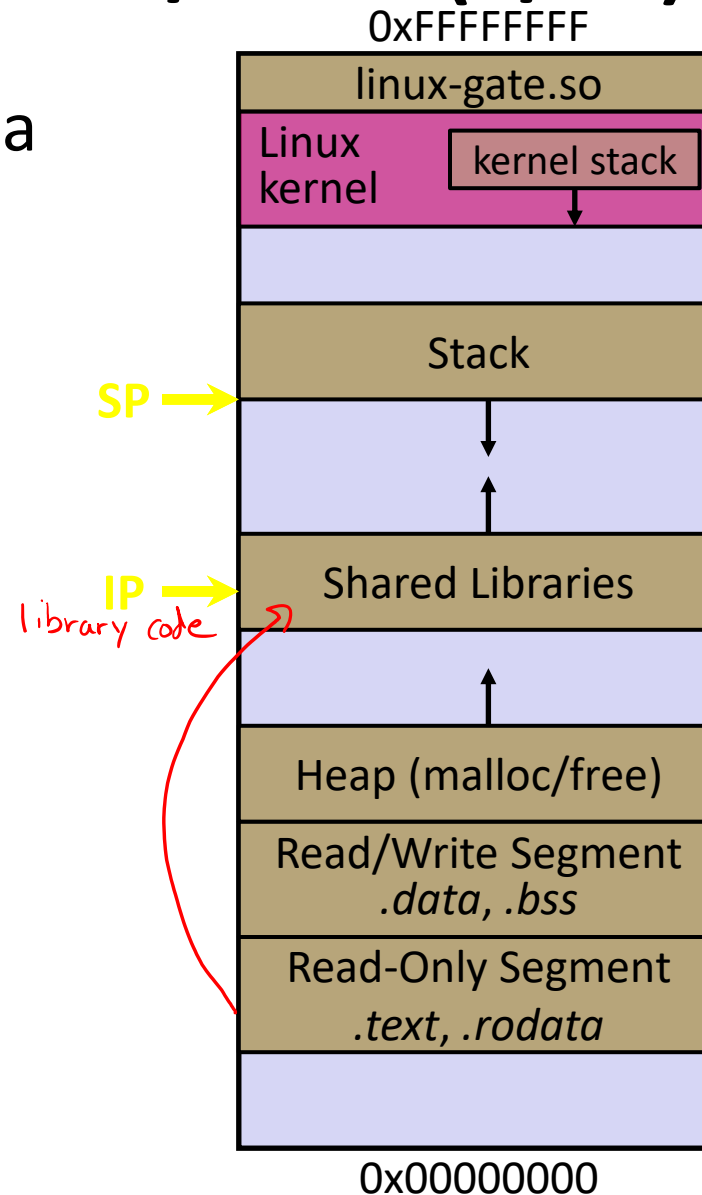
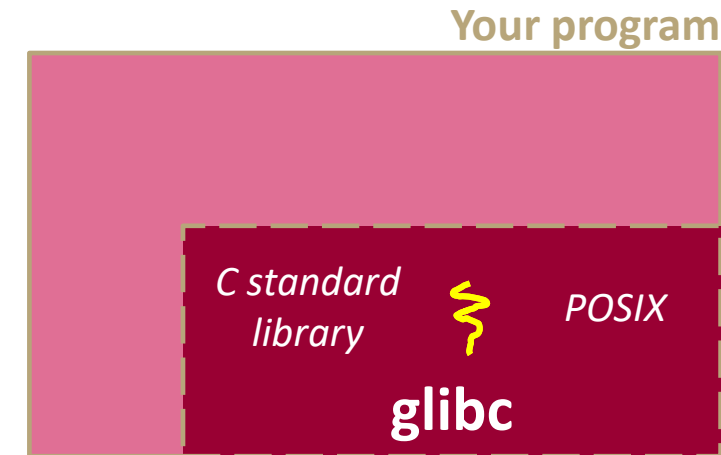
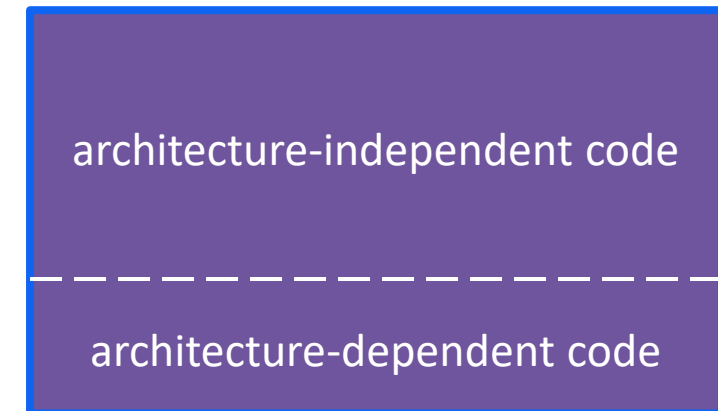
- ❖ Process is executing your program code



System Calls on x86/Linux (3/11)

❖ Process calls into a `glibc` function

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



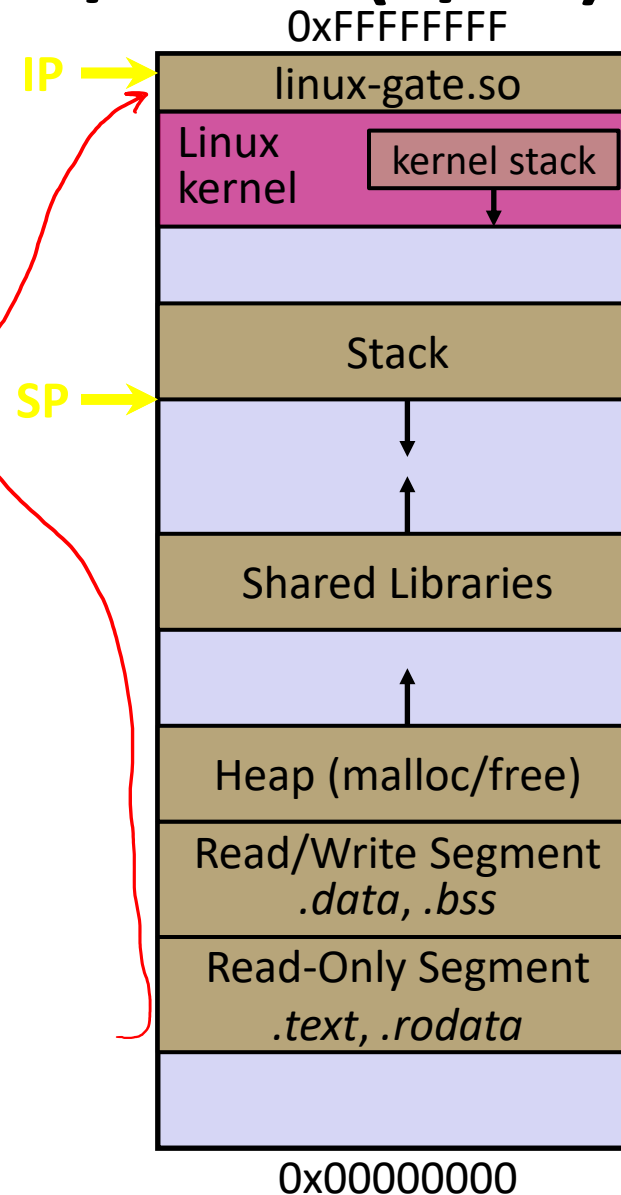
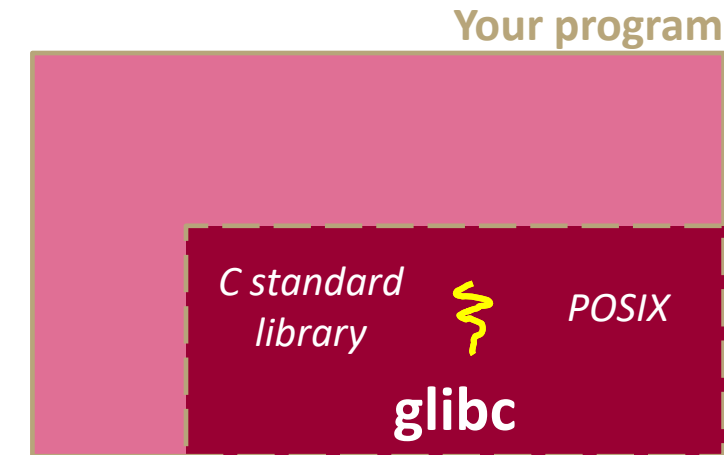
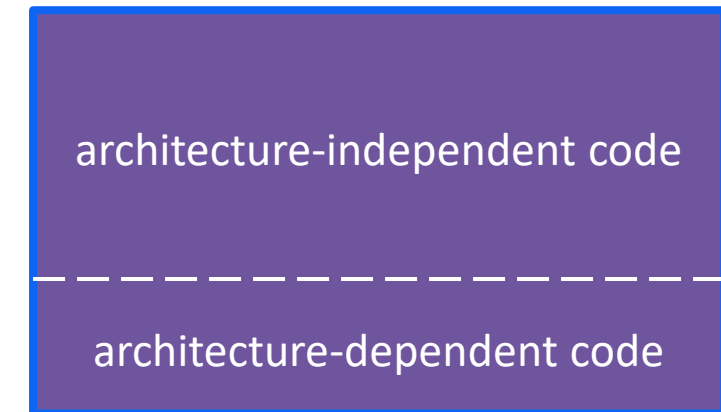
Linux kernel

unpriv

CPU

System Calls on x86/Linux (4/11)

- ❖ 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` (in `linux-gate.so`)



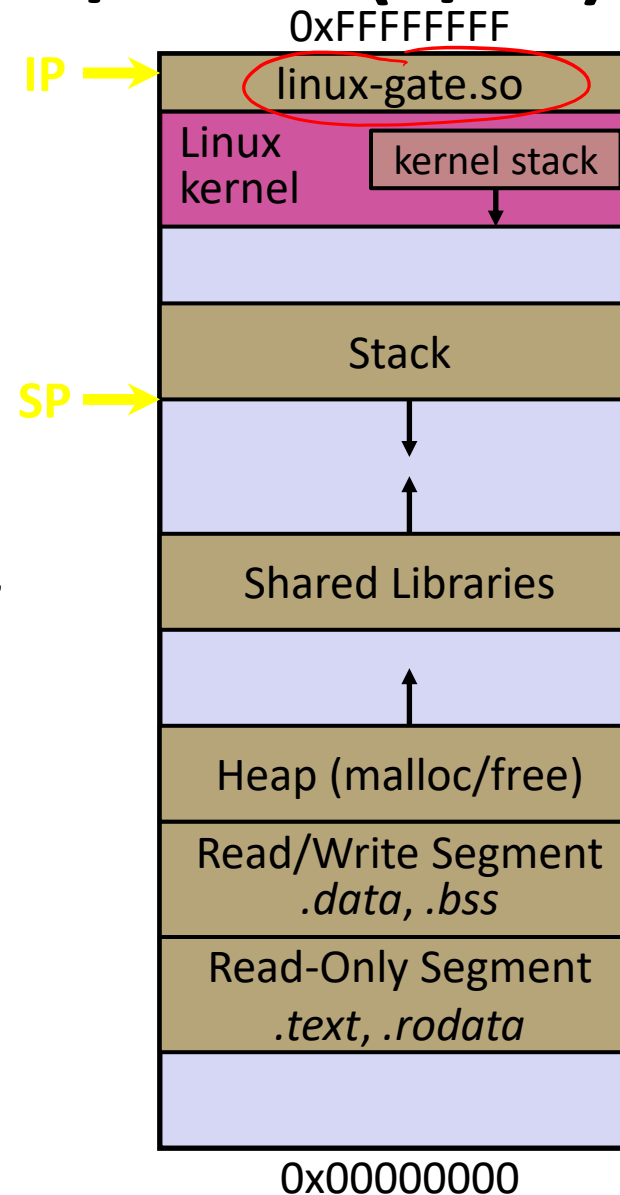
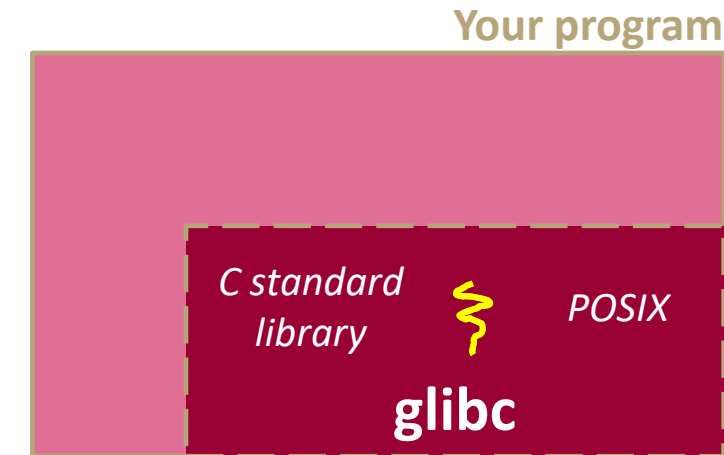
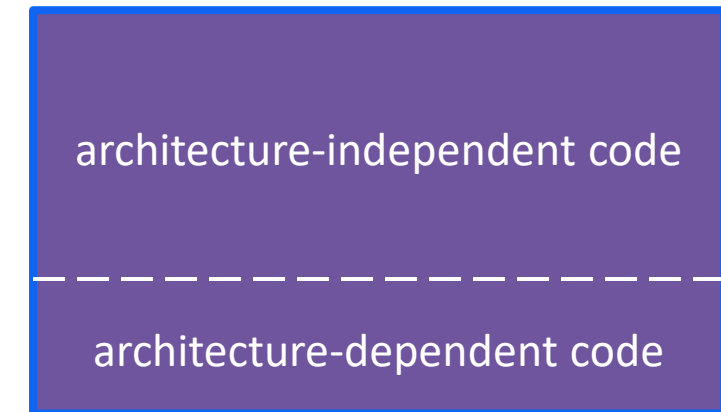
Linux kernel

unpriv CPU

System Calls on x86/Linux (5/11)

❖ linux-gate.so is a **vdso**

- A virtual dynamically-linked shared object
- 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 not important)

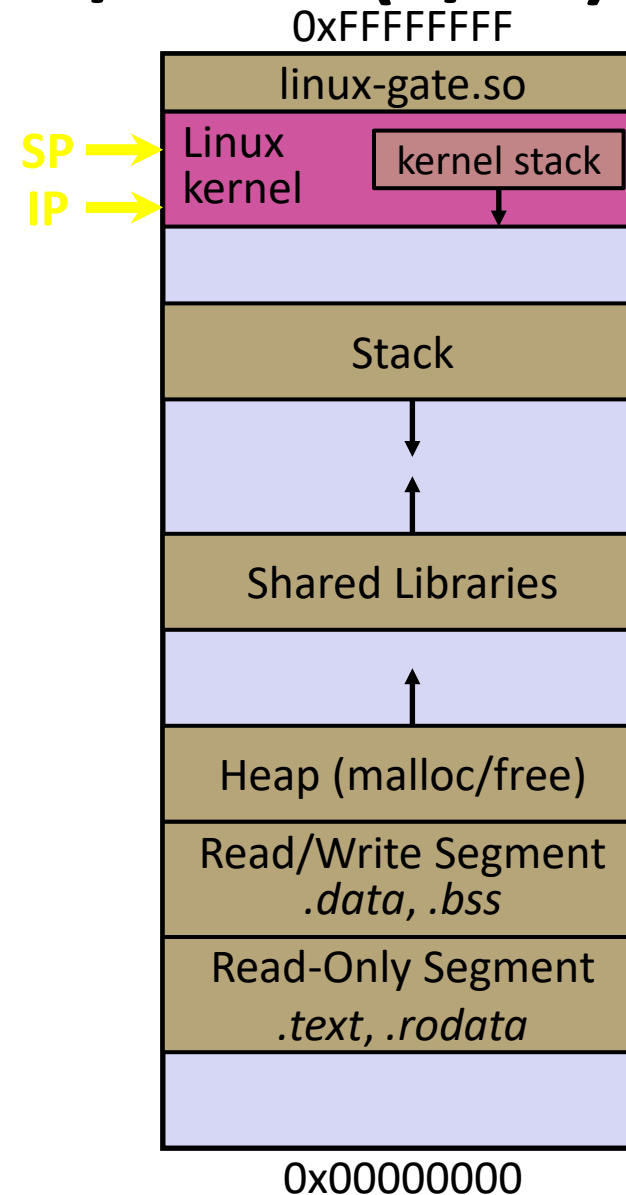
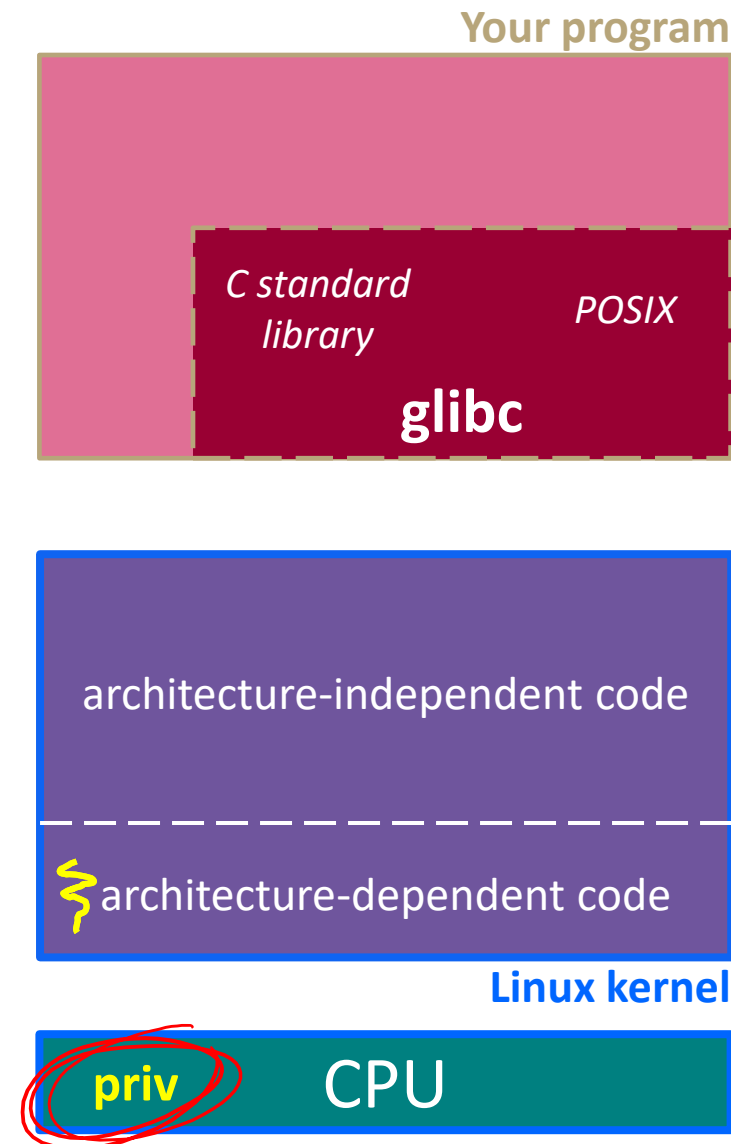


Linux kernel



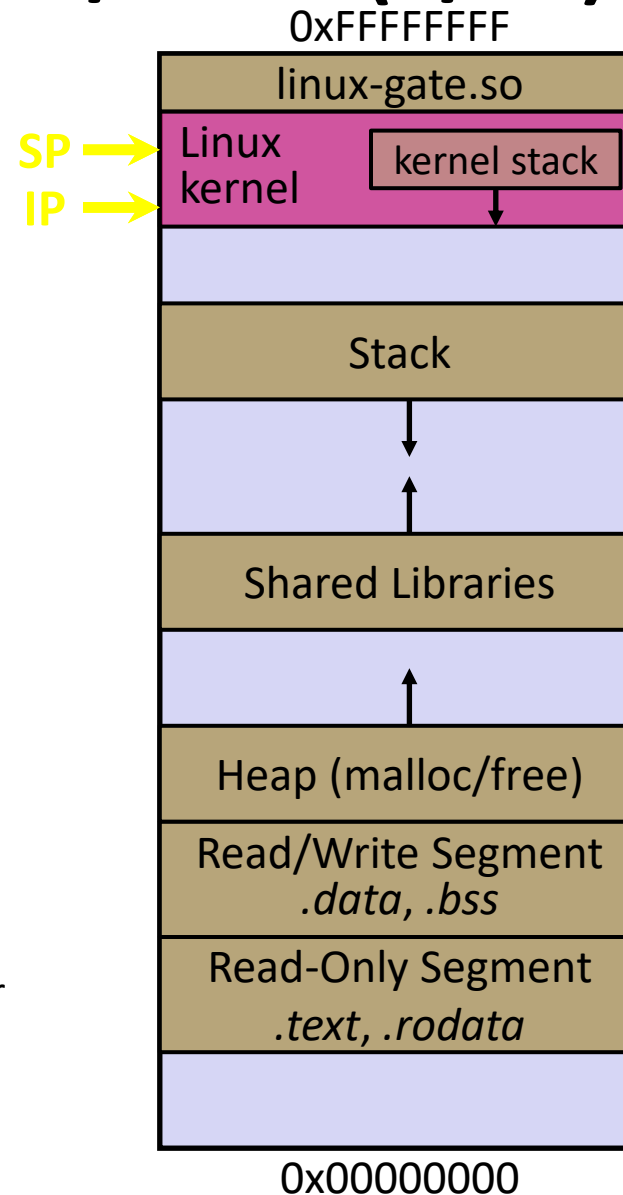
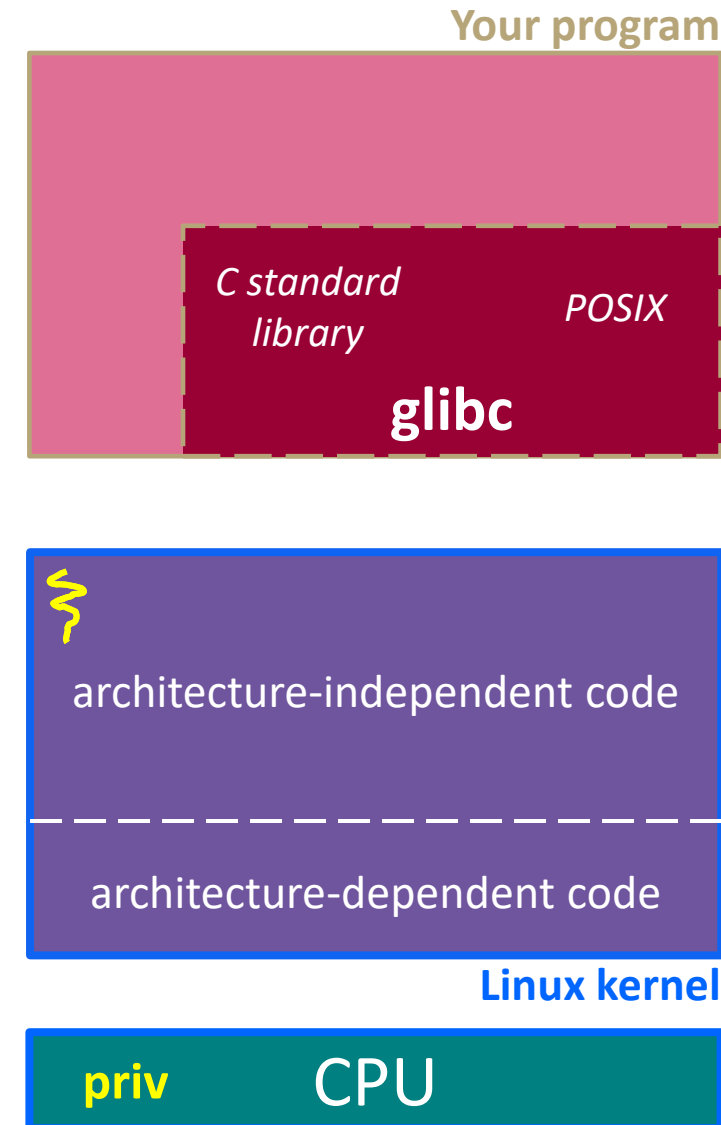
System Calls on x86/Linux (6/11)

- ❖ `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 CSE 451)



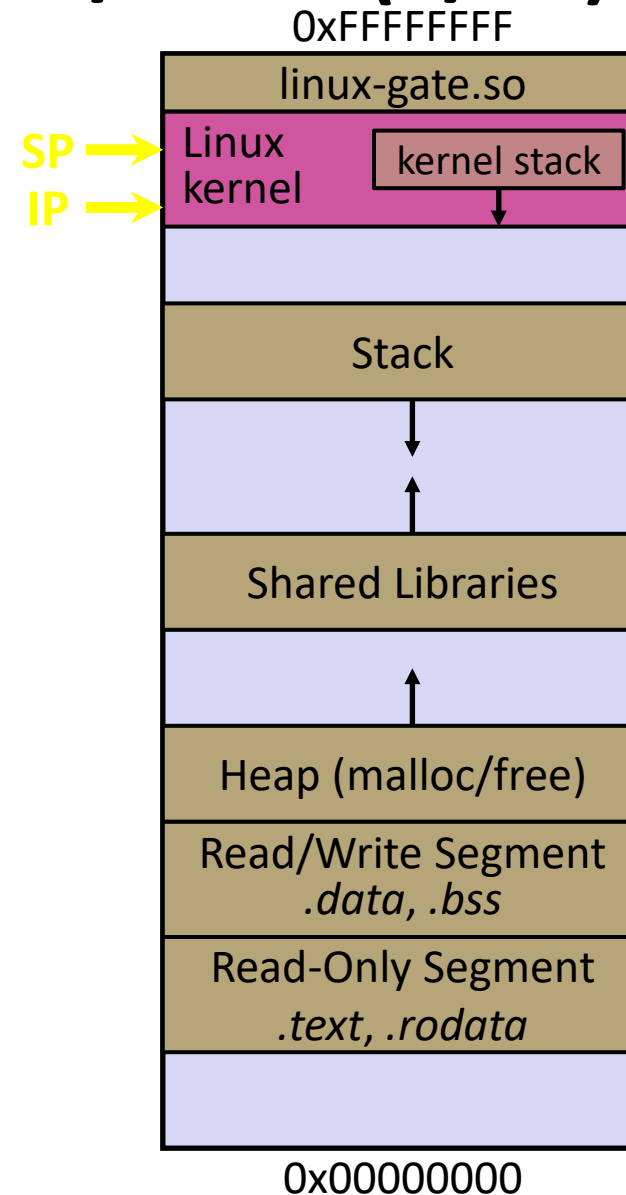
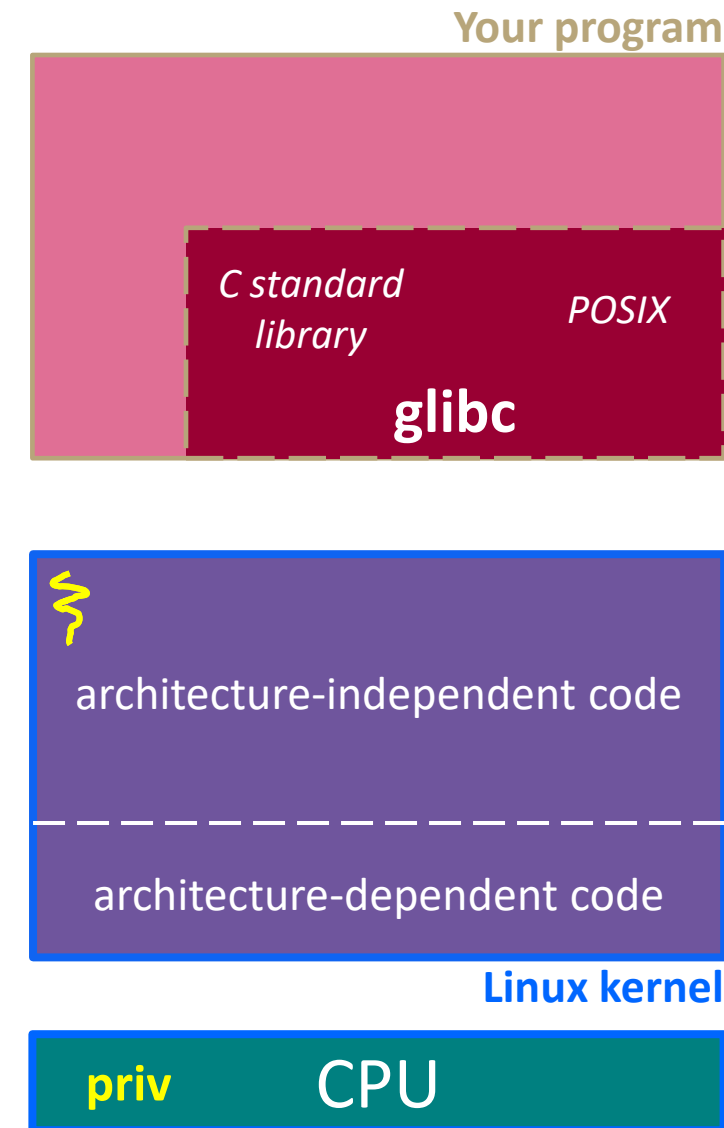
System Calls on x86/Linux (7/11)

- ❖ The kernel begins executing code at the SYSENTER entry point
 - Is in the architecture-dependent part of Linux
 - Its 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



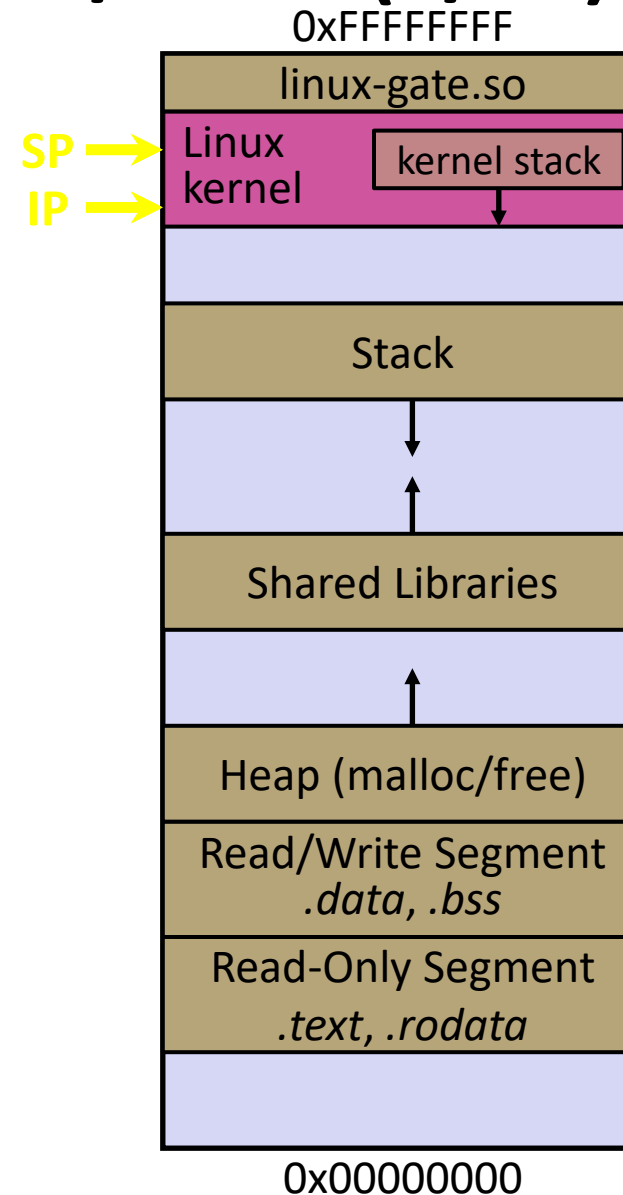
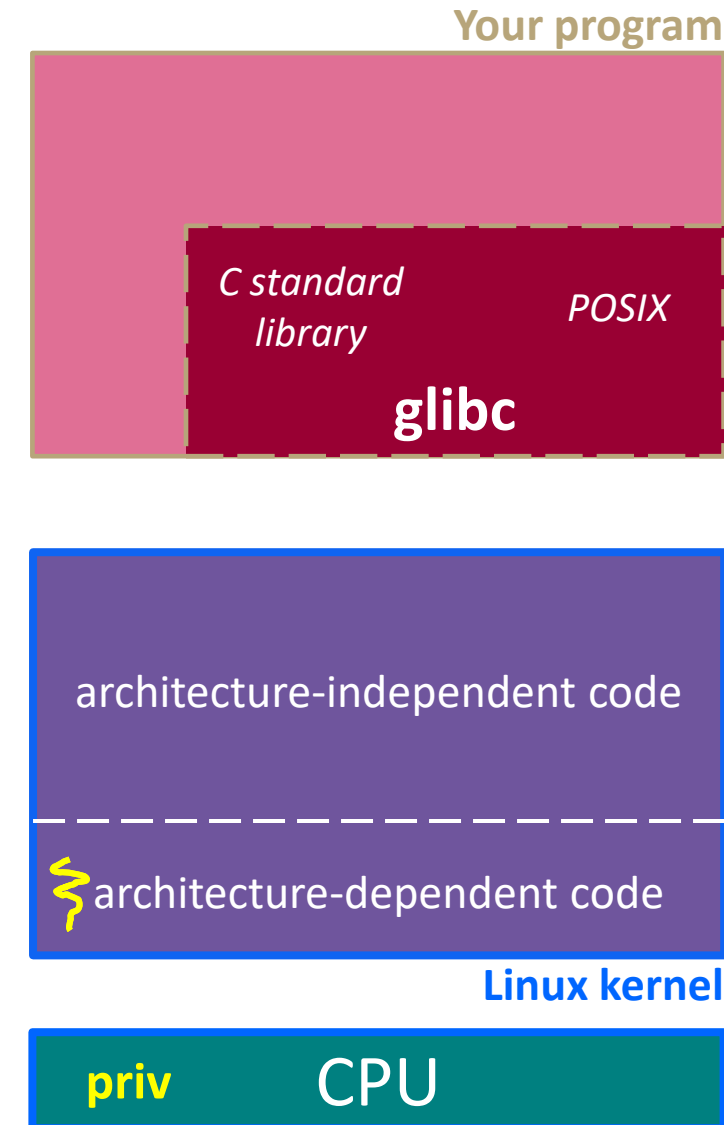
System Calls on x86/Linux (8/11)

- ❖ 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



System Calls on x86/Linux (9/11)

- ❖ 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

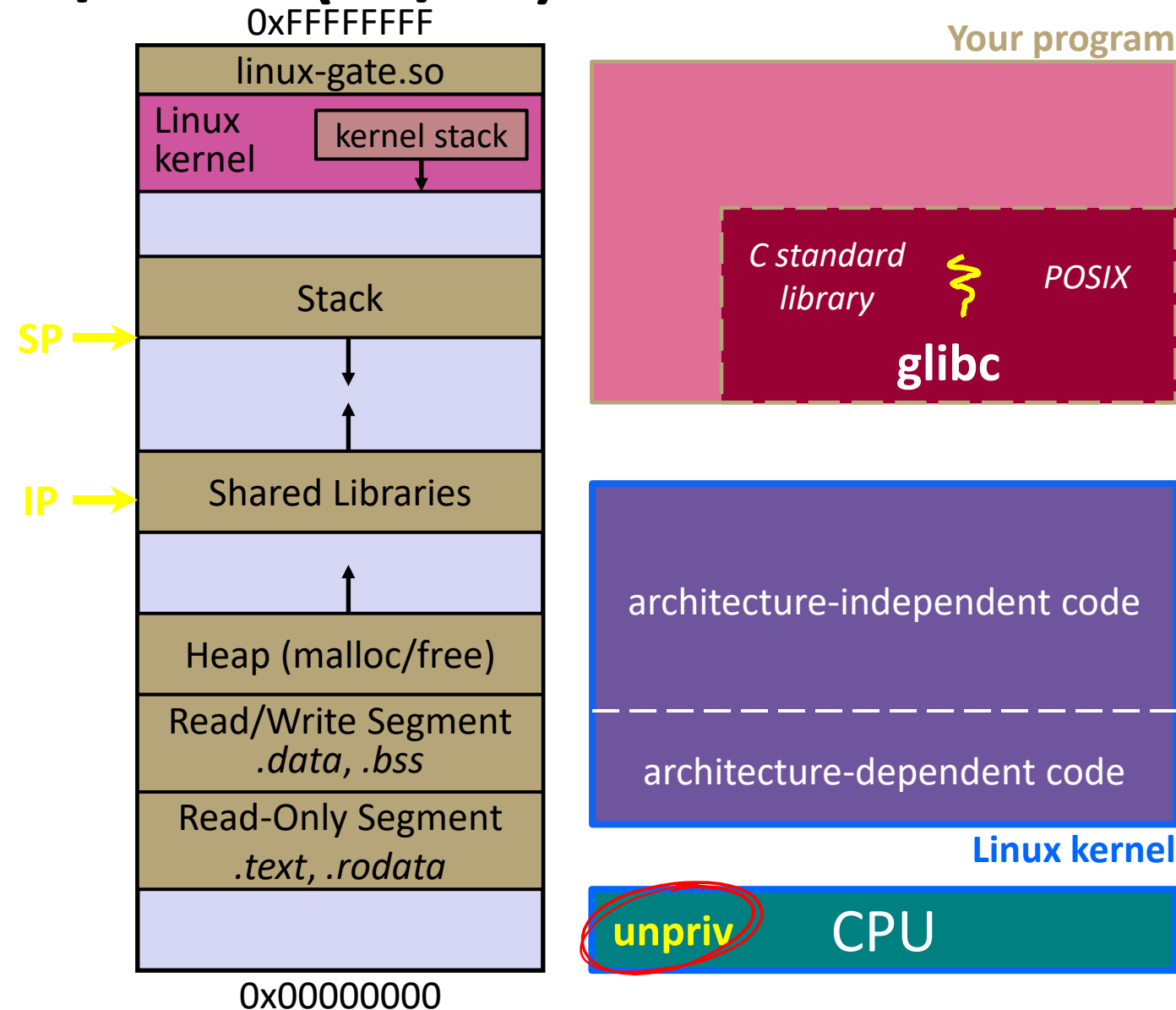


System Calls on x86/Linux (10/11)

❖ 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 CSE 451)
- Returns the processor back to glibc



System Calls on x86/Linux (11/11)

❖ `glibc` continues to execute

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

