Intro to File I/O, System Calls

CSE 333 Spring 2020

Instructor: Hal Perkins

Teaching Assistants:

Ramya Challa
Mengqui Chen
John Depaszthory
Greg Guo
Zachary Keyes
CJ Lin
Travis McGaha
Arjun Singh
Guramrit Singh
Cosmo Wang
Yifan Xu
Robin Yang
Haoran Yu
Velocity Yu

Administrivia

- I/O and System Calls – this lecture and next
  - Essential material for next part of the project (hw2)

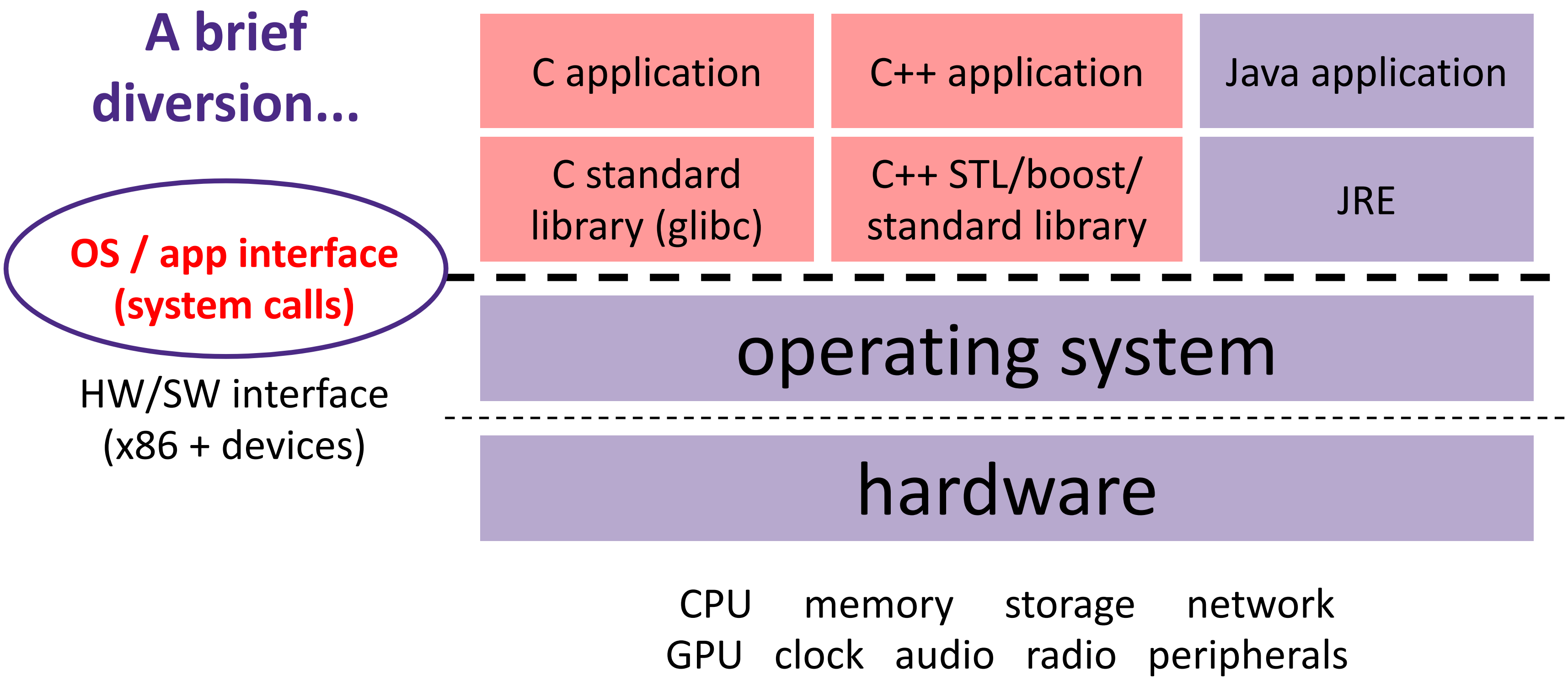
- Exercise 6 out today, due Wednesday morning 4/17
- Homework 1 due Thursday (4/18) at 11 pm
  - Submit via *GitLab* (i.e., commit/push changes, then push tag)
  - No exercise due Friday 4/19. Exercise 7 will be released on Thursday (based on section material), due following Monday

- MacOS 10.15.4 update: there are reports on the interwebs that this update can cause real problems especially with newer macs with T2 security chip. Best to hold off upgrading if you haven't yet. (And have a backup before you upgrade!)

Lecture Outline

- **File I/O with the C standard library**
- System Calls

Remember This Picture?


A brief diversion...
C application
C++ application
Java application
C standard library (glibc)
C++ STL/boost/ standard library
JRE
OS / app interface (system calls)
operating system
HW/SW interface (x86 + devices)
hardware
CPU memory storage network
GPU clock audio radio peripherals

File I/O

- We'll start by using C's standard library
  - These functions are part of `glibc` on Linux
  - They are implemented using Linux system calls

- C's `stdio` defines the notion of a stream
  - A way of reading or writing a sequence of characters to and from a device
  - Can be either *text* or *binary*; Linux does not distinguish
  - Is *buffered* by default; `libc` reads ahead of your program
  - Three streams provided by default: `stdin`, `stdout`, `stderr`
    - You can open additional streams to read and write to files
  - C streams are manipulated with a `FILE*` pointer, which is defined in `stdio.h`

C Stream Functions

- Some stream functions (complete list in `stdio.h`):

  -
    ```
    FILE* fopen(filename, mode);
    ```
    - Opens a stream to the specified file in specified file access mode

  -
    ```
    int fclose(stream);
    ```
    - Closes the specified stream (and file)

  -
    ```
    size_t fwrite(ptr, size, count, stream);
    ```
    - Writes an array of *count* elements of *size* bytes from *ptr* to *stream*

  -
    ```
    size_t fread(ptr, size, count, stream);
    ```
    - Reads an array of *count* elements of *size* bytes from *stream* to *ptr*

C Stream Functions

- Formatted I/O stream functions (more in in `stdio.h`):

  -
    ```
    int fprintf(stream, format, ...);
    ```
    - Writes a formatted C string
      - `printf(...);` is equivalent to `fprintf(stdout, ...);`

  -
    ```
    int fscanf(stream, format, ...);
    ```
    - Reads data and stores data matching the format string

Error Checking/Handling

- Some error functions (complete list in `stdio.h`):

  -
    ```
    void perror(message);
    ```
    - Prints `message` and error message related to `errno` to `stderr`

  -
    ```
    int ferror(stream);
    ```
    - Checks if the error indicator associated with the specified stream is set

  -
    ```
    int clearerr(stream);
    ```
    - Resets error and eof indicators for the specified stream

C Streams Example

cp_example.c

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#define READBUFSIZE 128

int main(int argc, char** argv) {
  FILE *fin, *fout;
  char readbuf[READBUFSIZE];
  size_t readlen;

  if (argc != 3) {
    fprintf(stderr, "usage: ./cp_example infile outfile\n");
    return EXIT_FAILURE;       // defined in stdlib.h
  }

  // Open the input file
  fin = fopen(argv[1], "rb");  // "rb" -> read, binary mode
  if (fin == NULL) {
    fprintf(stderr, "%s -- ", argv[1]);
    perror("fopen for read failed");
    return EXIT_FAILURE;
  }
  ...
```

C Streams Example

cp_example.c

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

  ...   // previous slide’s code

  // Open the output file
  fout = fopen(argv[2], "wb");  // "wb" -> write, binary mode
  if (fout == NULL) {
    fprintf(stderr, "%s -- ", argv[2]);
    perror("fopen for write failed");
    return EXIT_FAILURE;
  }

  // Read from the file, write to fout
  while ((readlen = fread(readbuf, 1, READBUFSIZE, fin)) > 0) {
    if (fwrite(readbuf, 1, readlen, fout) < readlen) {
      perror("fwrite failed");
      return EXIT_FAILURE;
    }
  }

  ...   // next slide’s code

}
```

C Streams Example

cp_example.c

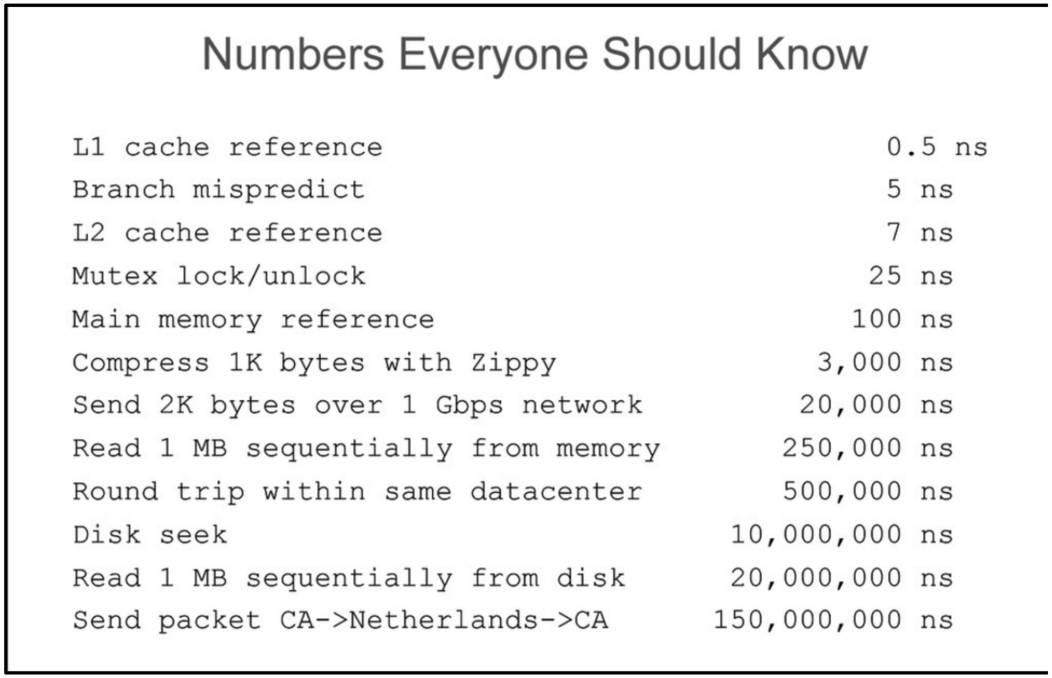
```
int main(int argc, char** argv) {

  ...   // Slide 7’s code

  ...   // Slide 8’s code

  // Test to see if we encountered an error while reading
  if (ferror(fin)) {
    perror("fread failed");
    return EXIT_FAILURE;
  }

  fclose(fin);
  fclose(fout);

  return EXIT_SUCCESS;
}
```

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`**`()`)

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()` shortly



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
  - "Completion" of a write (*i.e.* return from **`fwrite`**`()`) does not mean the data has actually been written
    - What if you signal another process to read the file you just wrote to?

- 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?

Disabling C's Buffering

- Explicitly turn off with `setbuf(stream, NULL)`
  - But potential performance problems: lots of small writes triggers lots of slower system calls instead of a single system call that writes a large chunk

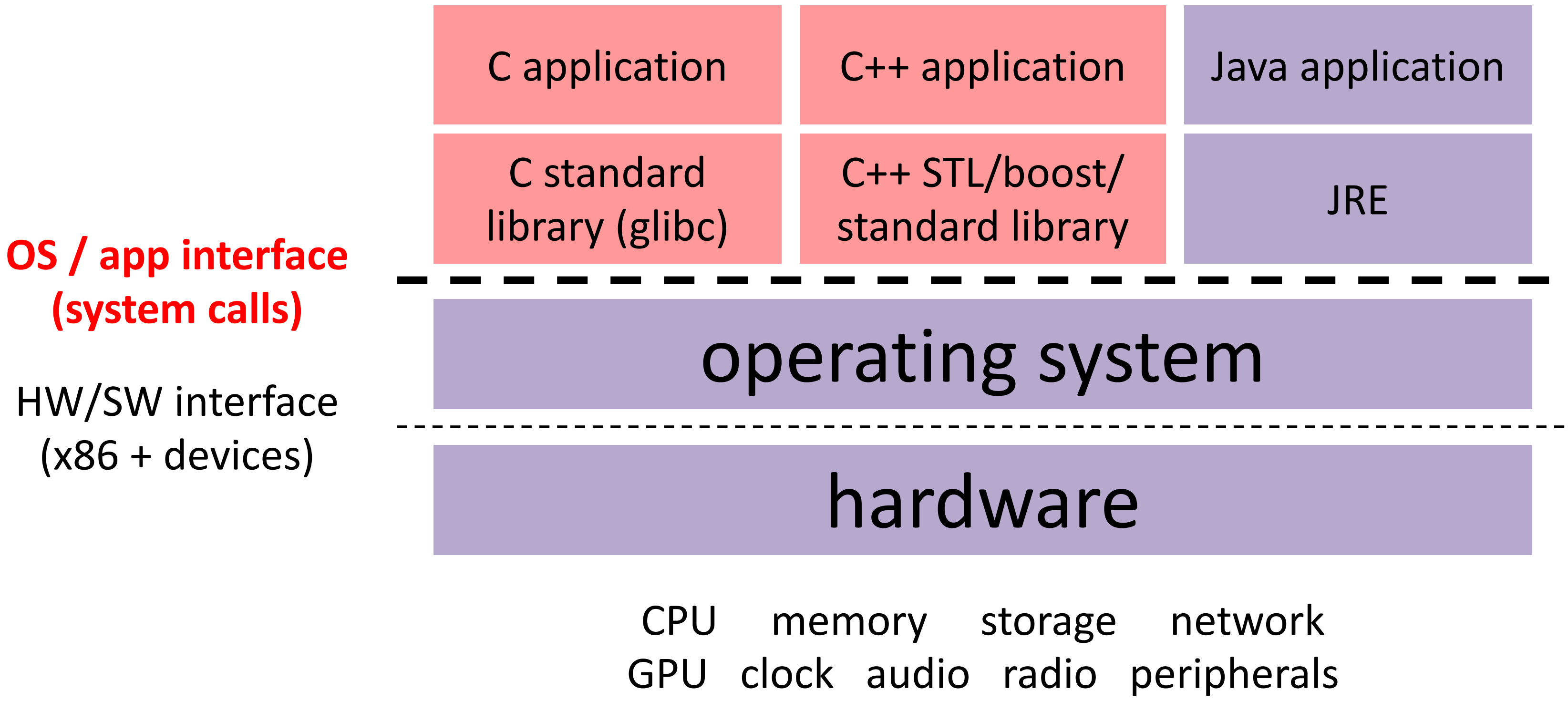
- Use POSIX APIs instead of C's
  - No buffering is done at the user level
  - We'll see these soon

- But… what about the layers below?
  - The OS caches disk reads and writes in the file system *buffer* cache
  - Disk controllers have caches too!

Lecture Outline

- File I/O with the C standard library
- **System Calls**

What's an OS?



C application

C++ application

Java application

C standard library (glibc)

C++ STL/boost/ standard library

JRE

OS / app interface (system calls)

operating system

HW/SW interface (x86 + devices)

hardware

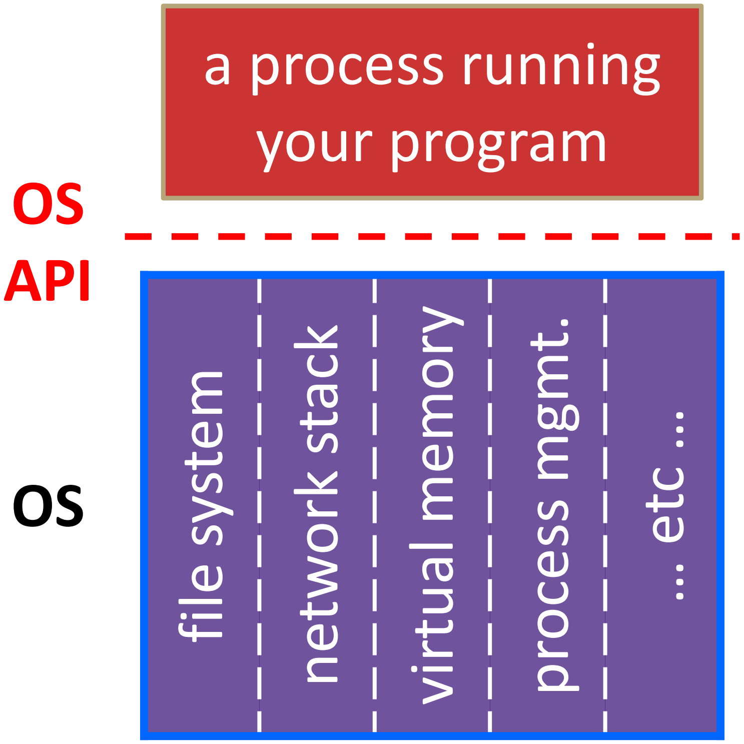
CPU memory storage network
GPU clock audio radio peripherals

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

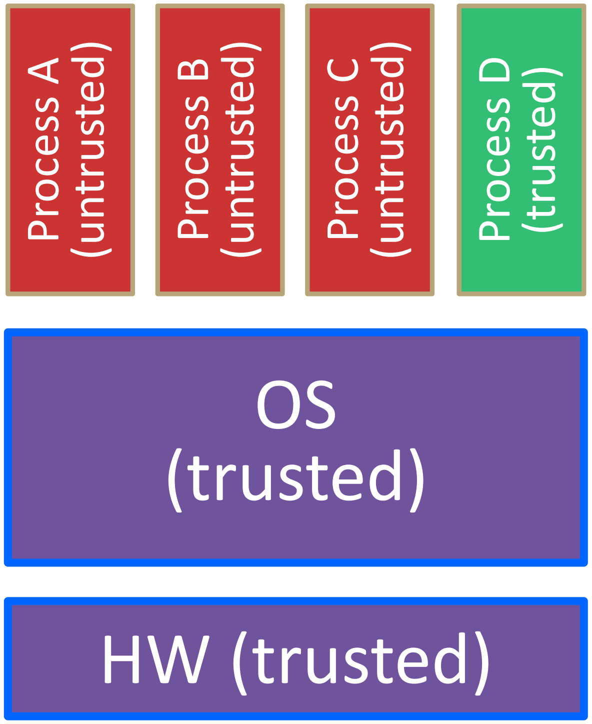
- brk(), shm_open(), …

Process Management

- fork(), wait(), nice(), …

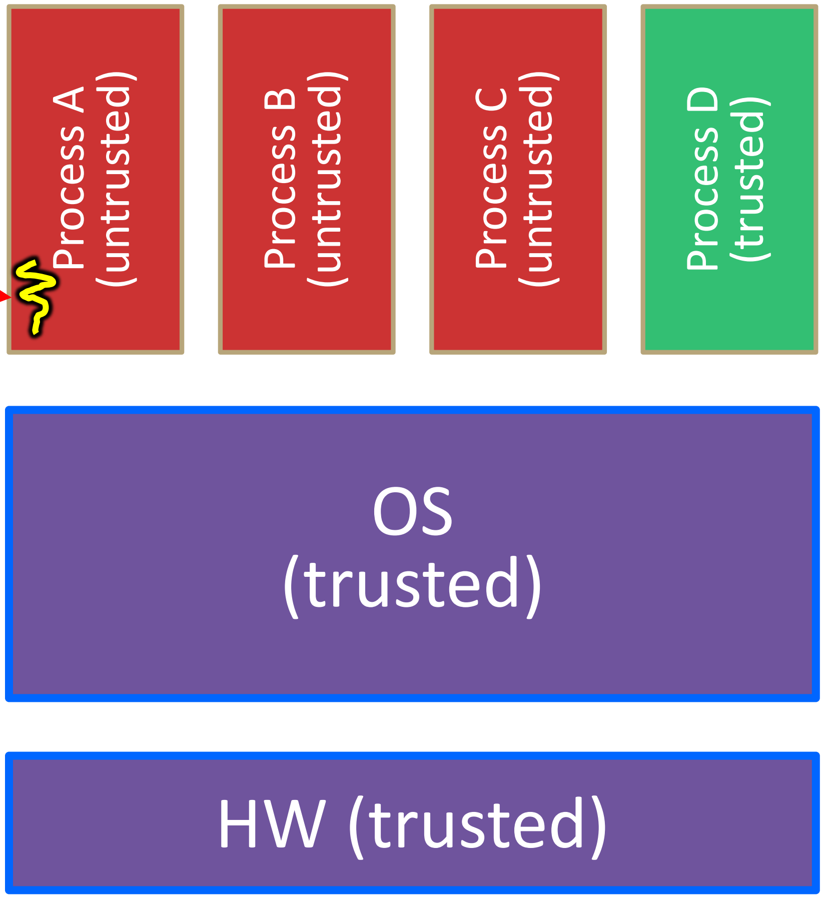
OS: Protection System

- 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



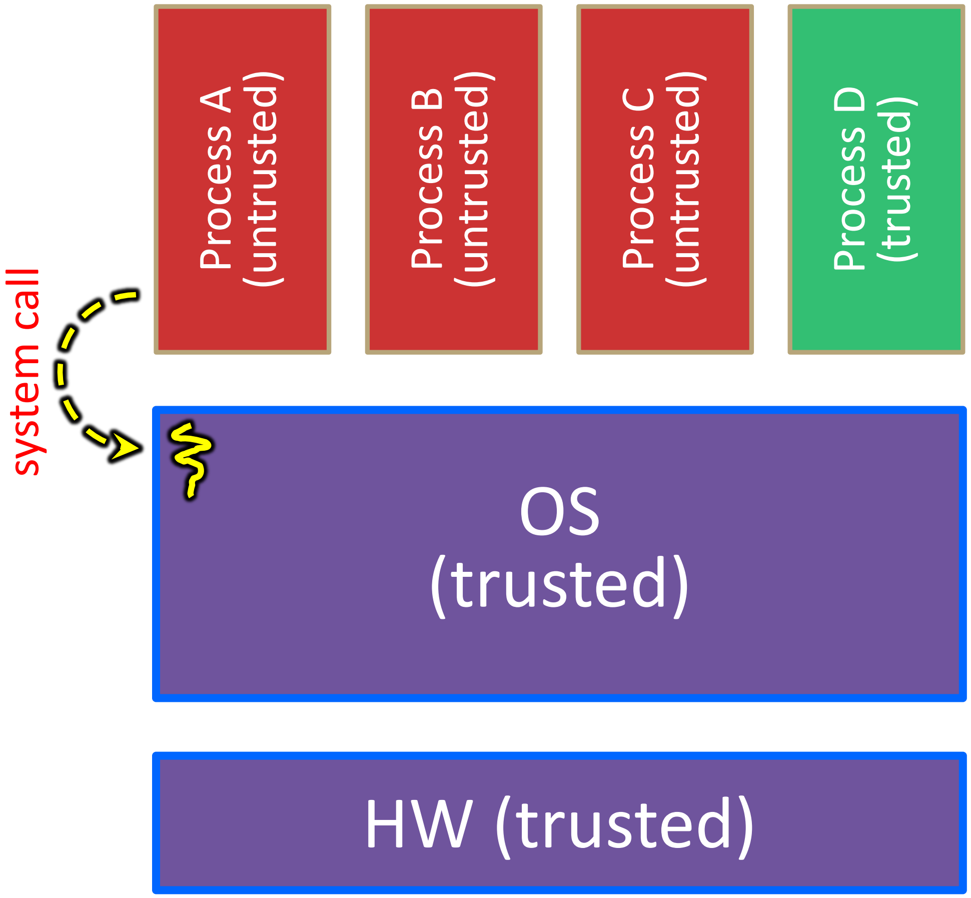
System Call Trace

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



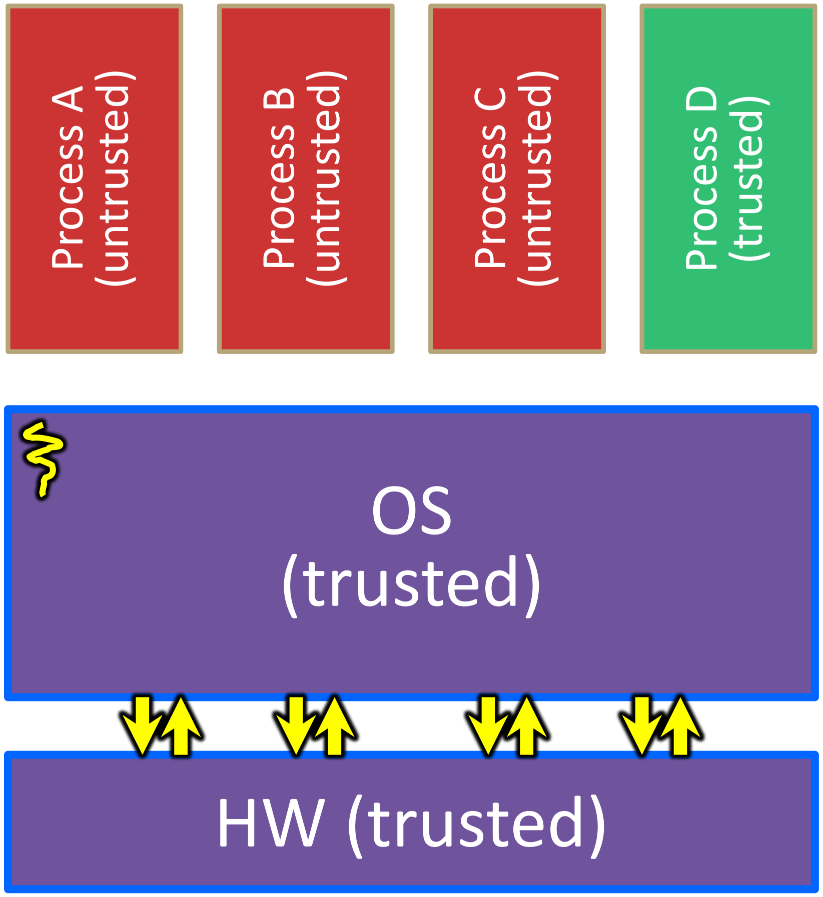
System Call Trace

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

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.

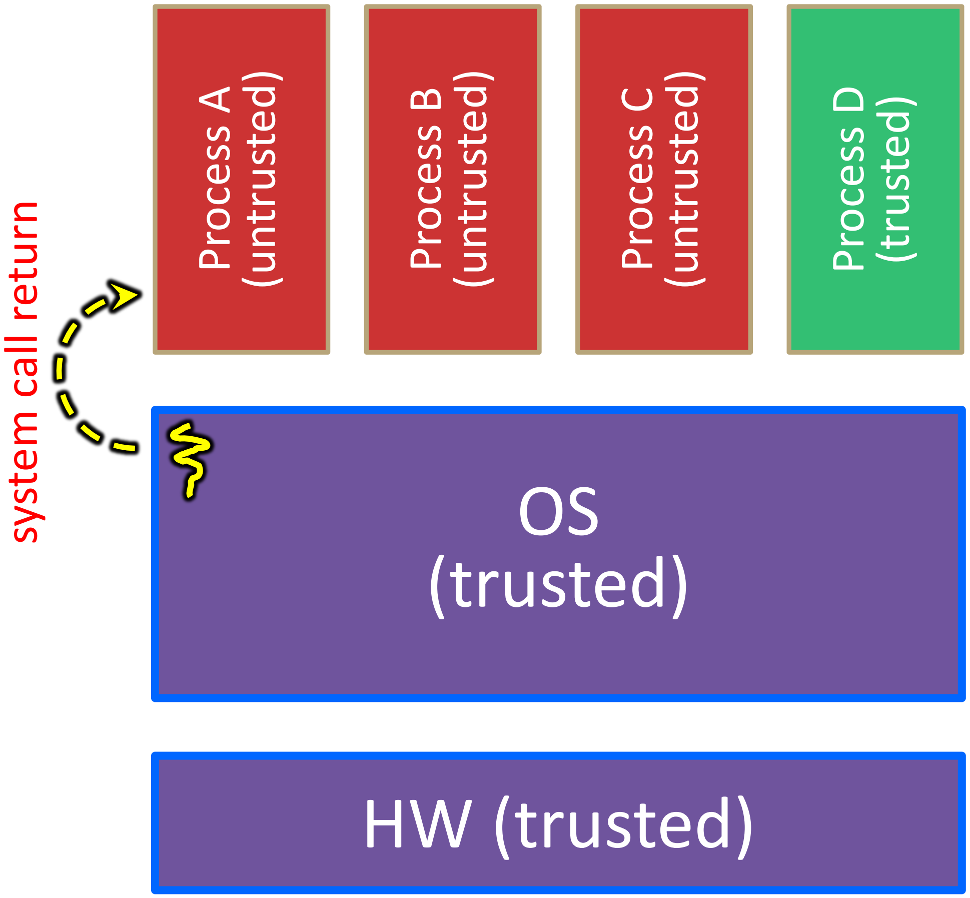


System Call Trace

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

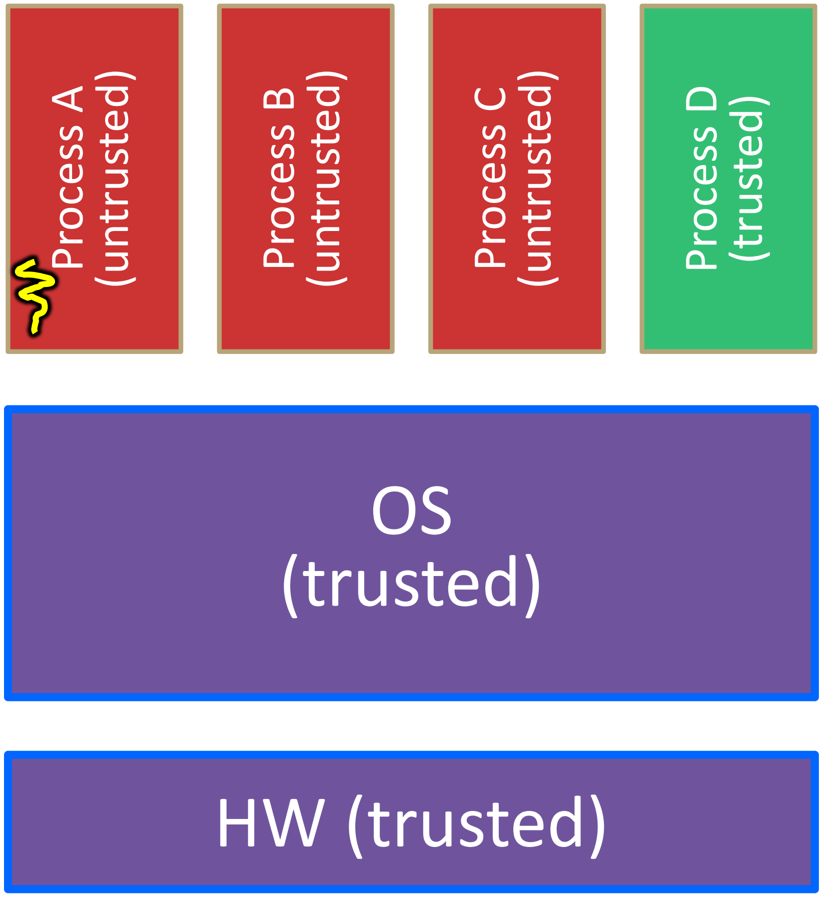
(1) Sets the CPU back to unprivileged mode and

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



System Call Trace

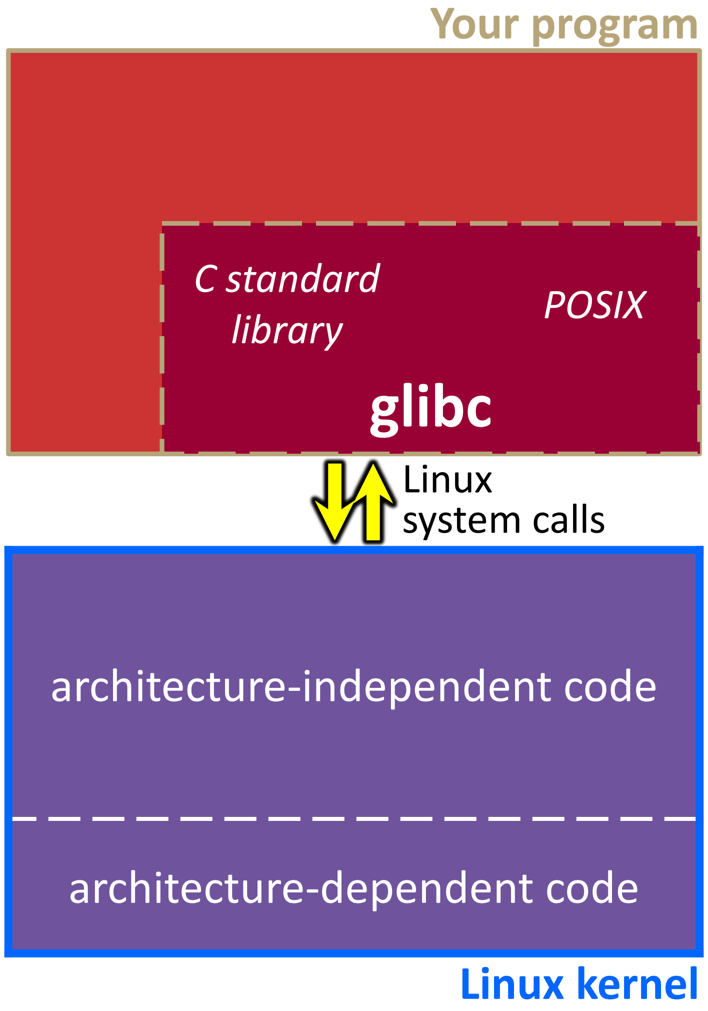
The process continues executing whatever code is next after the system call invocation.



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

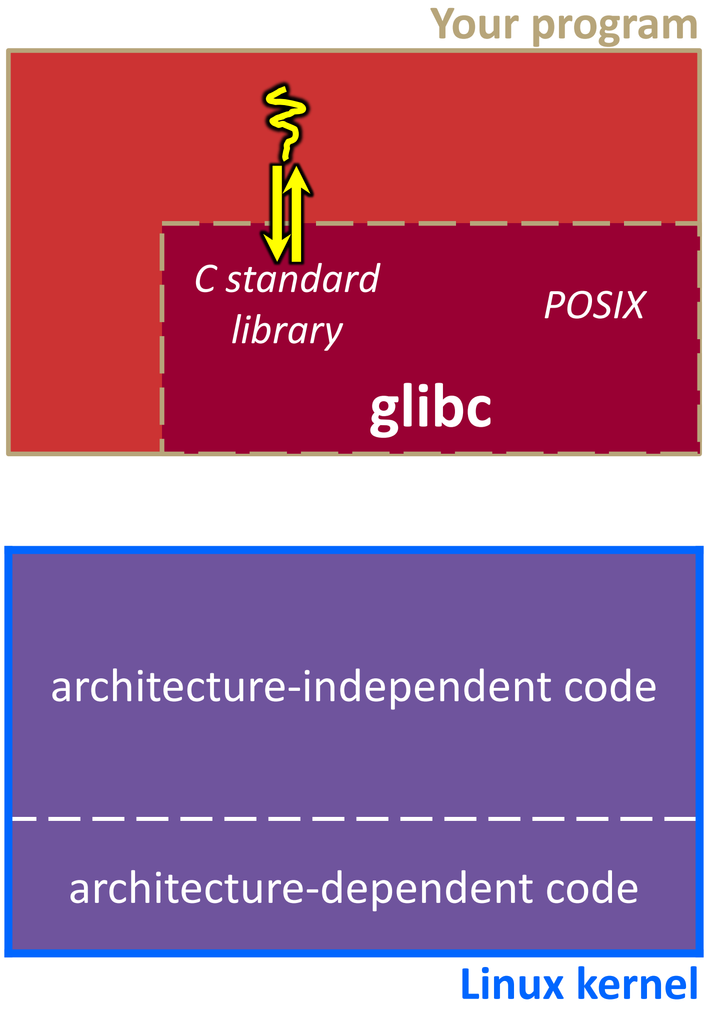
Details on x86/Linux

- 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



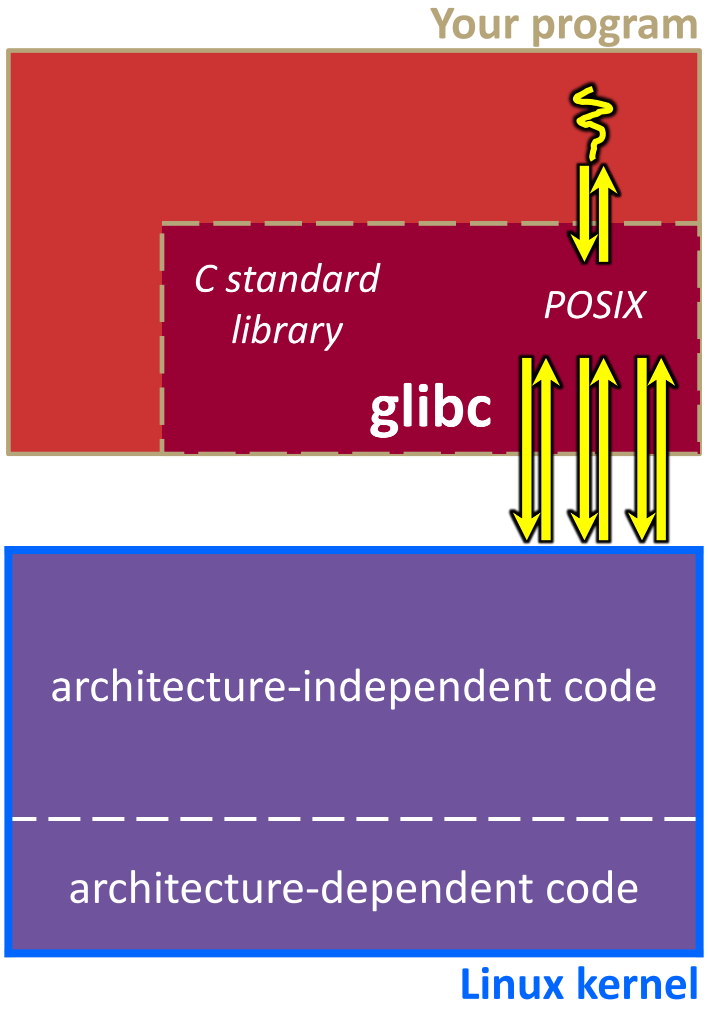
Details on x86/Linux

- 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!



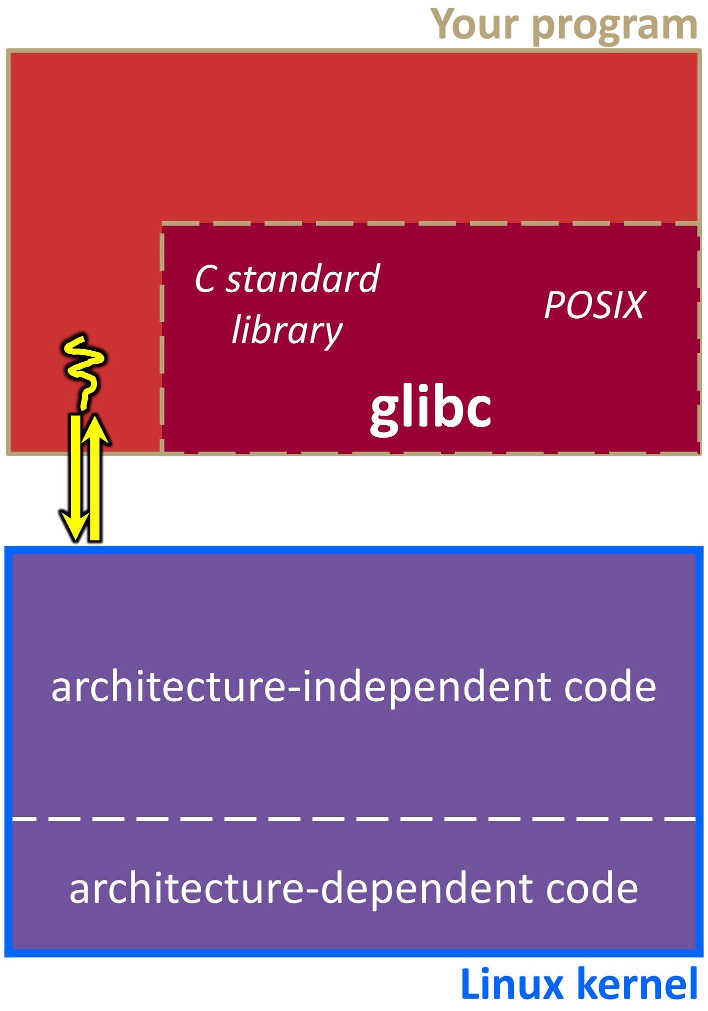
Details on x86/Linux

- 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.



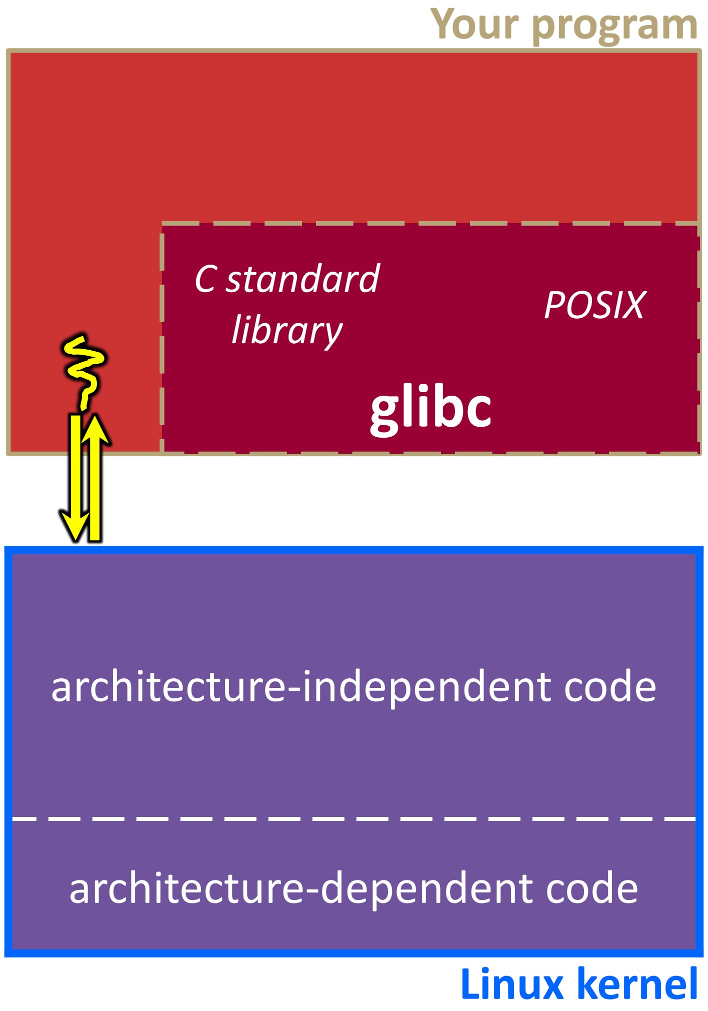
Details on x86/Linux

- 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



Details on x86/Linux

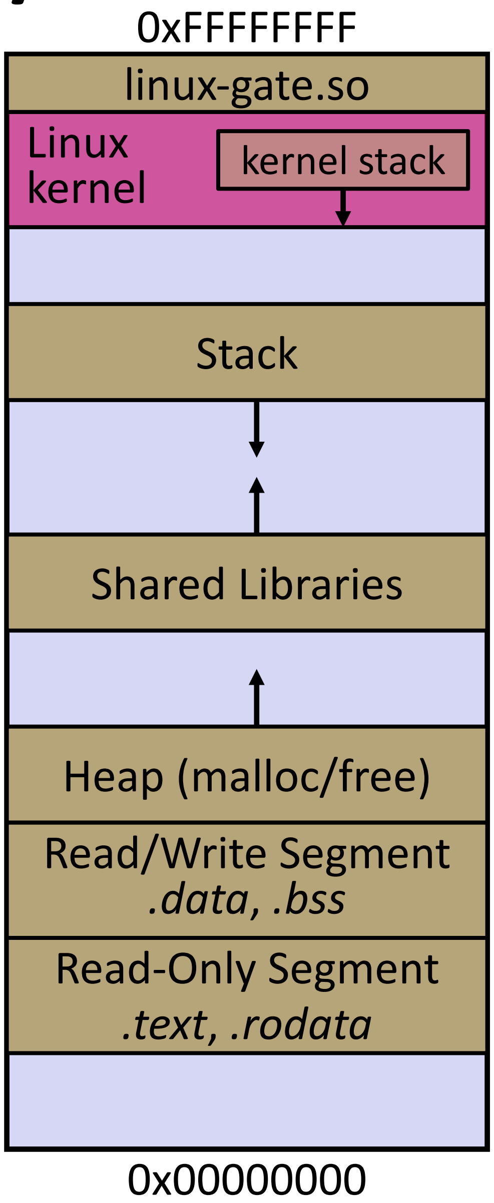
- Let’s walk through how a Linux system call actually works
  - We’ll assume *32-bit x86* using the modern `SYSENTER` / `SYSEXIT` x86 instructions
    - x86-64 code is similar, though details always change over time, so take this as an example – not a debugging guide

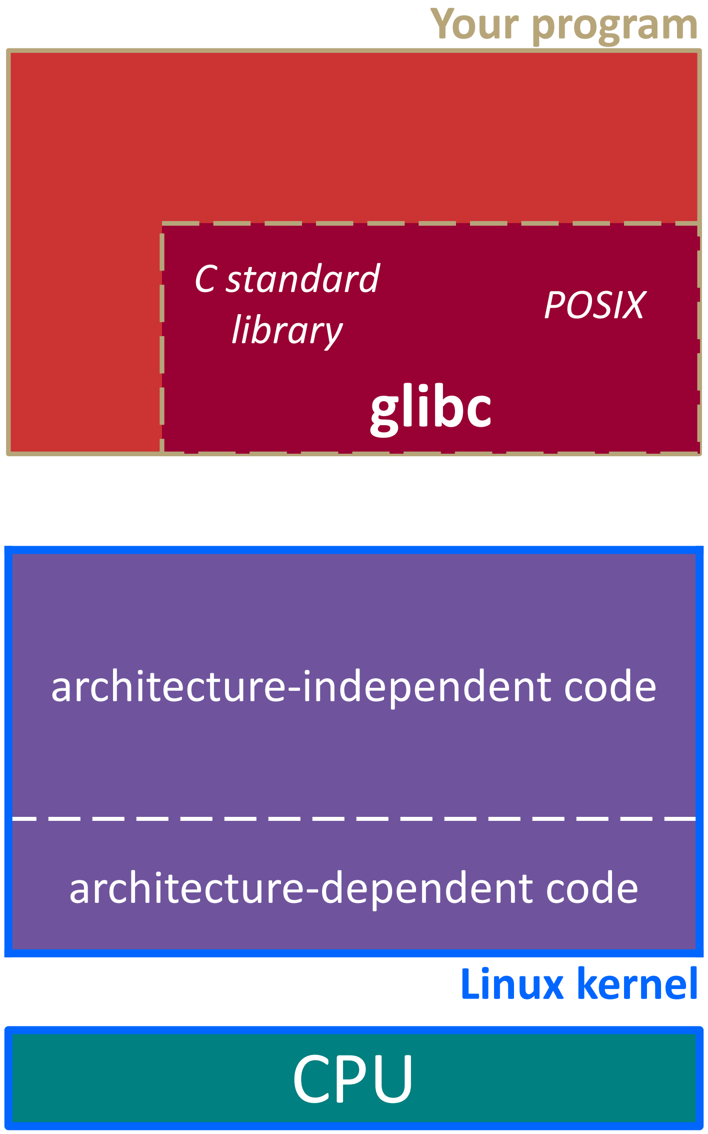


Details on x86/Linux

Remember our process address space picture?

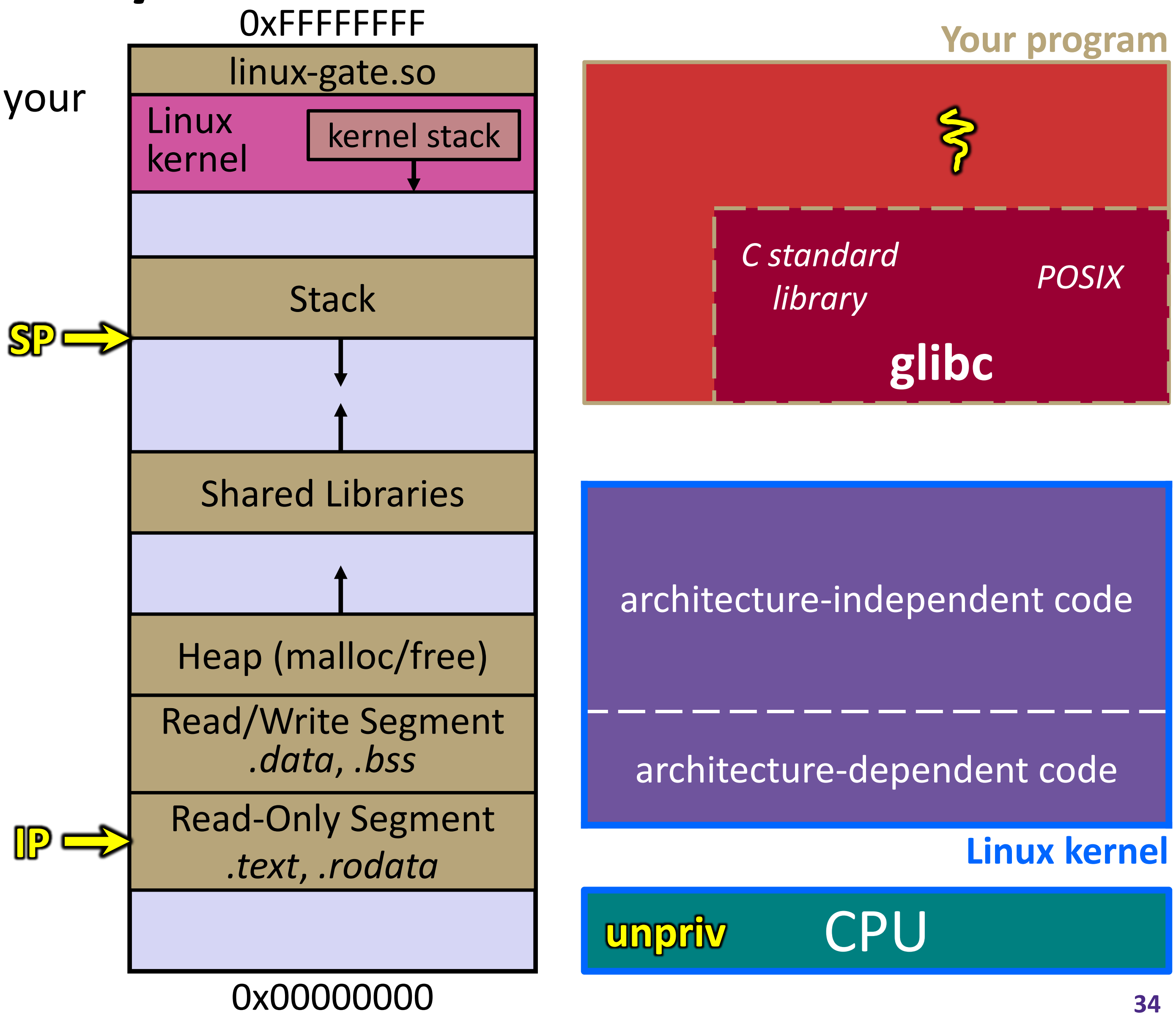
- Let's add some details:





Details on x86/Linux

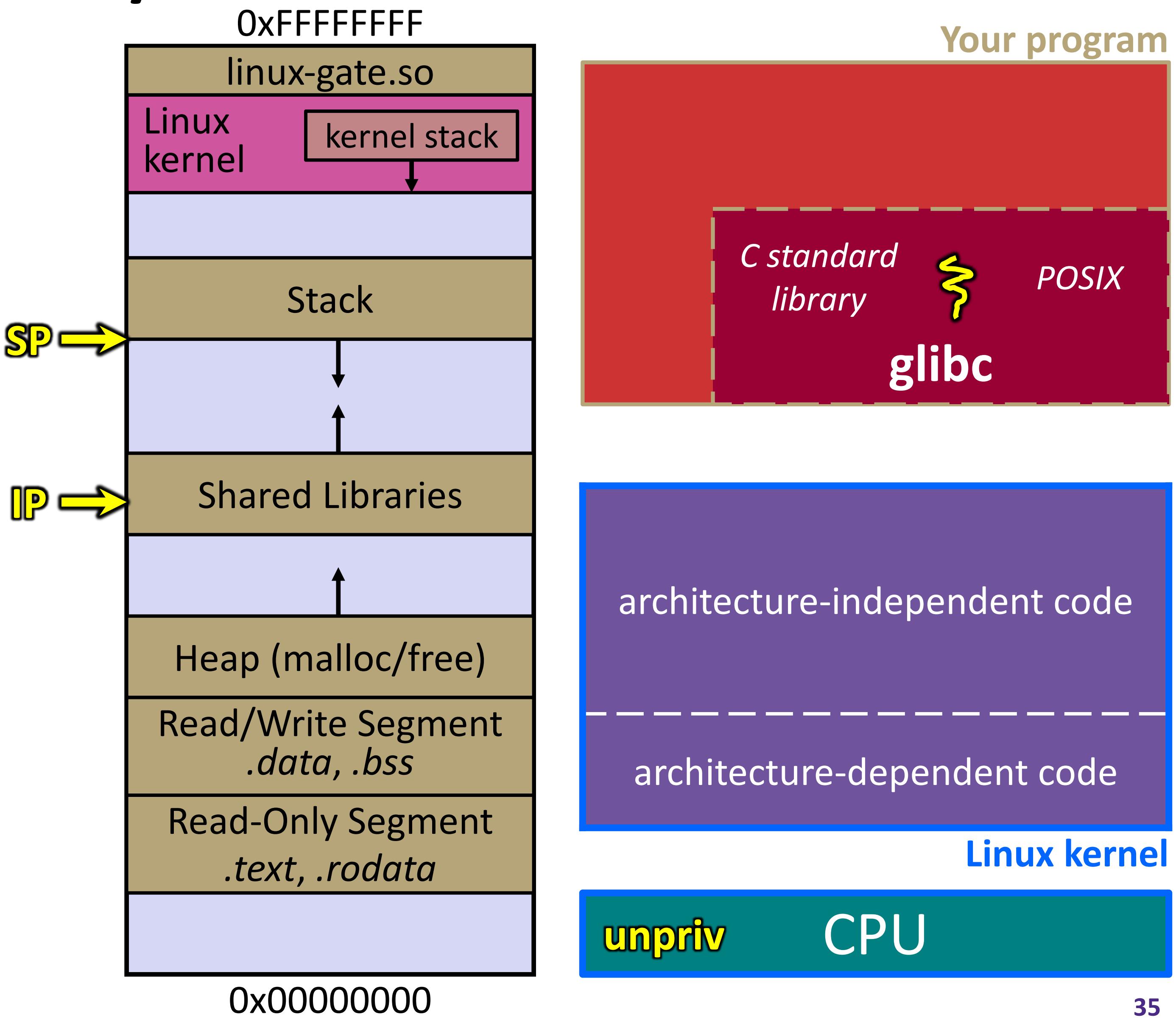
Process is executing your program code



Details on x86/Linux

Process calls into a `glibc` function

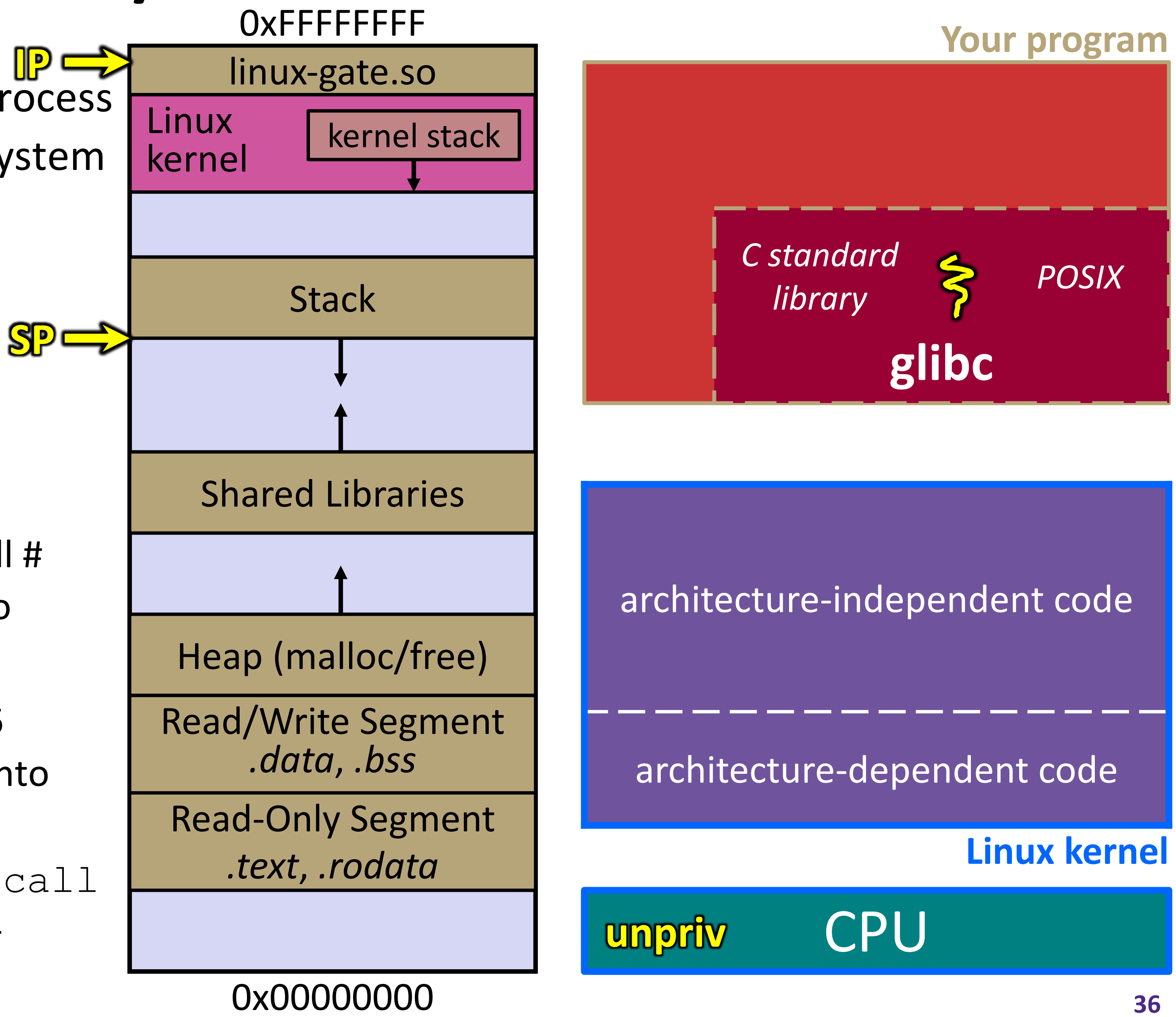
- *e.g.* `fopen()`
- We'll ignore the messy details of loading/linking shared libraries



Details on x86/Linux

glibc begins the process of invoking a Linux system call

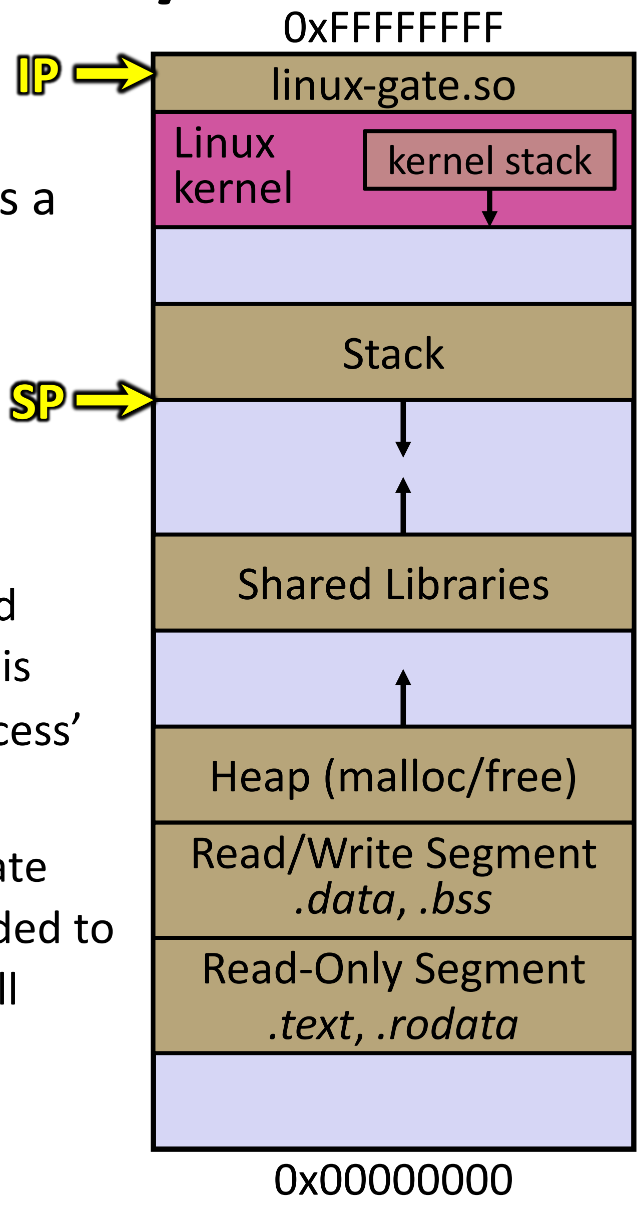
- glibc's fopen() likely invokes Linux's open() system call
- Puts the system call # and arguments into registers
- Uses the **call** x86 instruction to call into the routine __kernel_vsyscall located in linux-gate.so

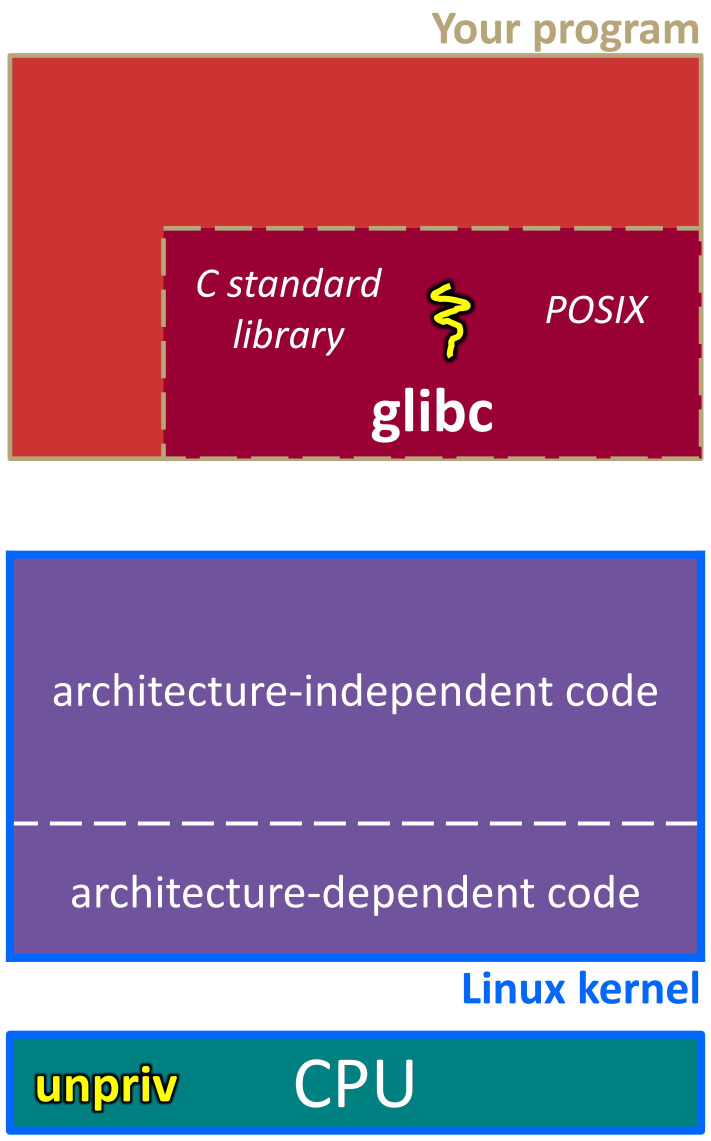


Details on x86/Linux

`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

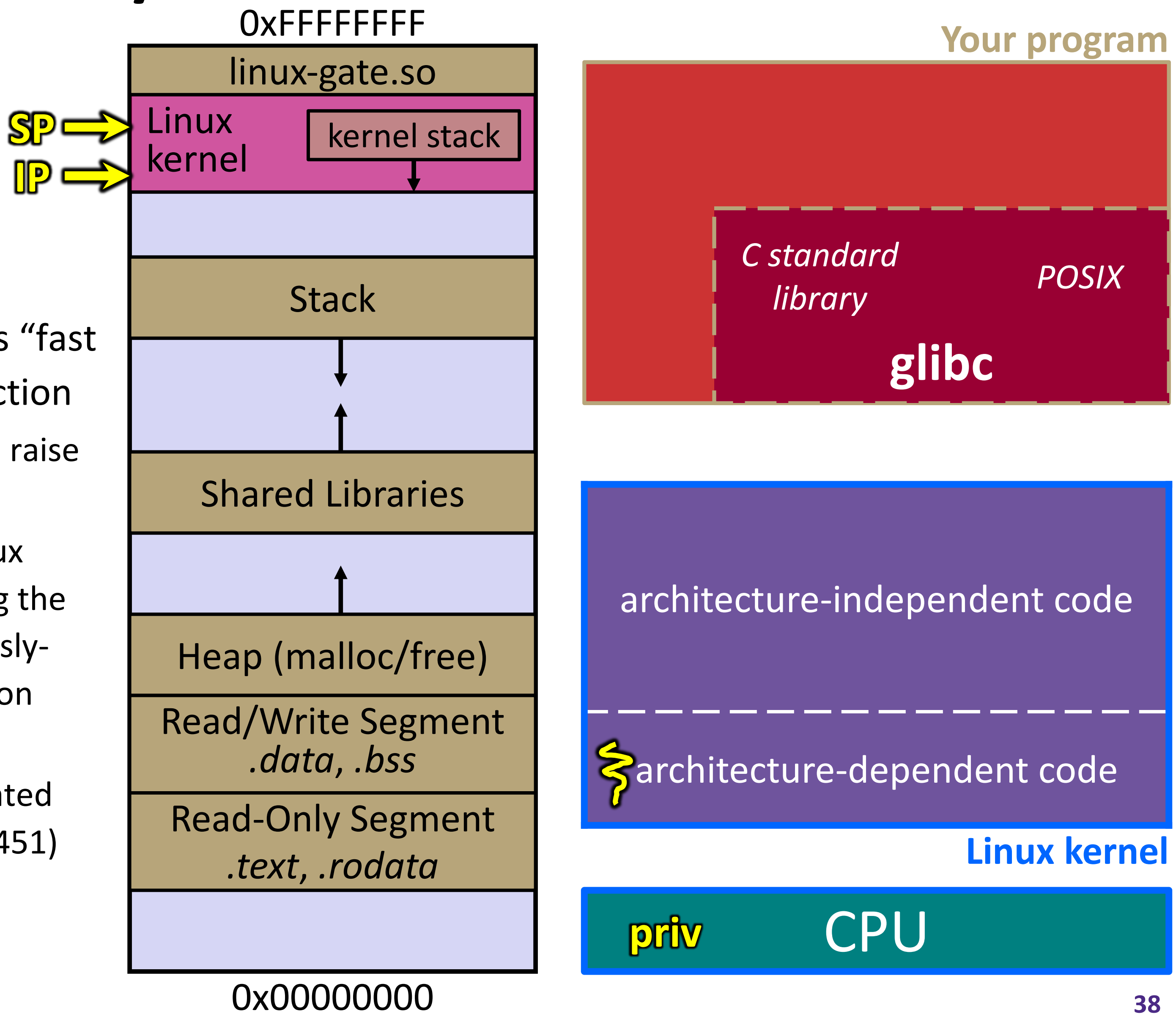




Details on x86/Linux

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

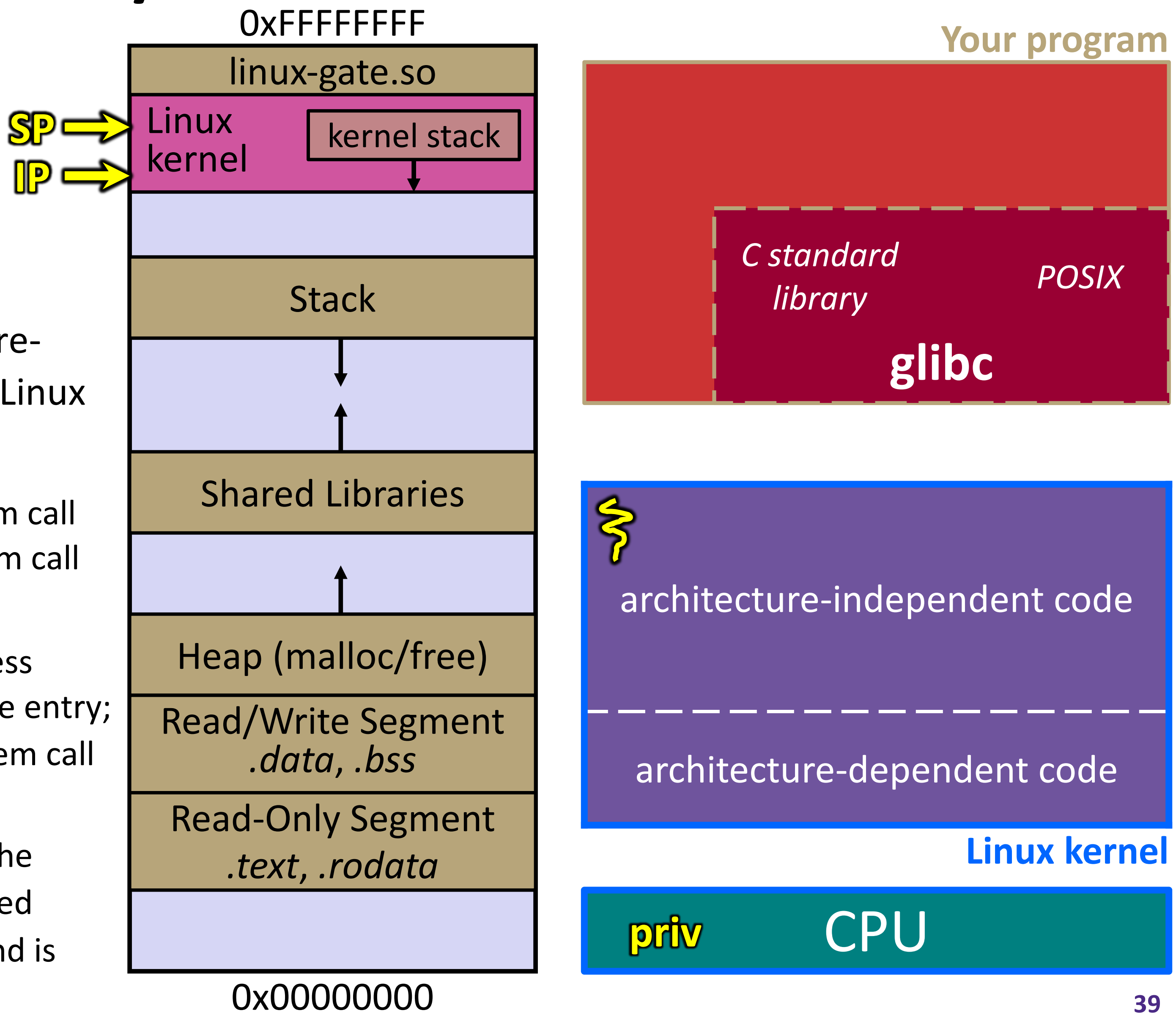
- `SYSENTER` is x86's "fast system call" instruction
  - Causes the CPU to raise its privilege level
  - Traps into the Linux kernel by changing the SP, IP to a previously-determined location
  - Changes some segmentation-related registers (see CSE451)



Details on x86/Linux

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

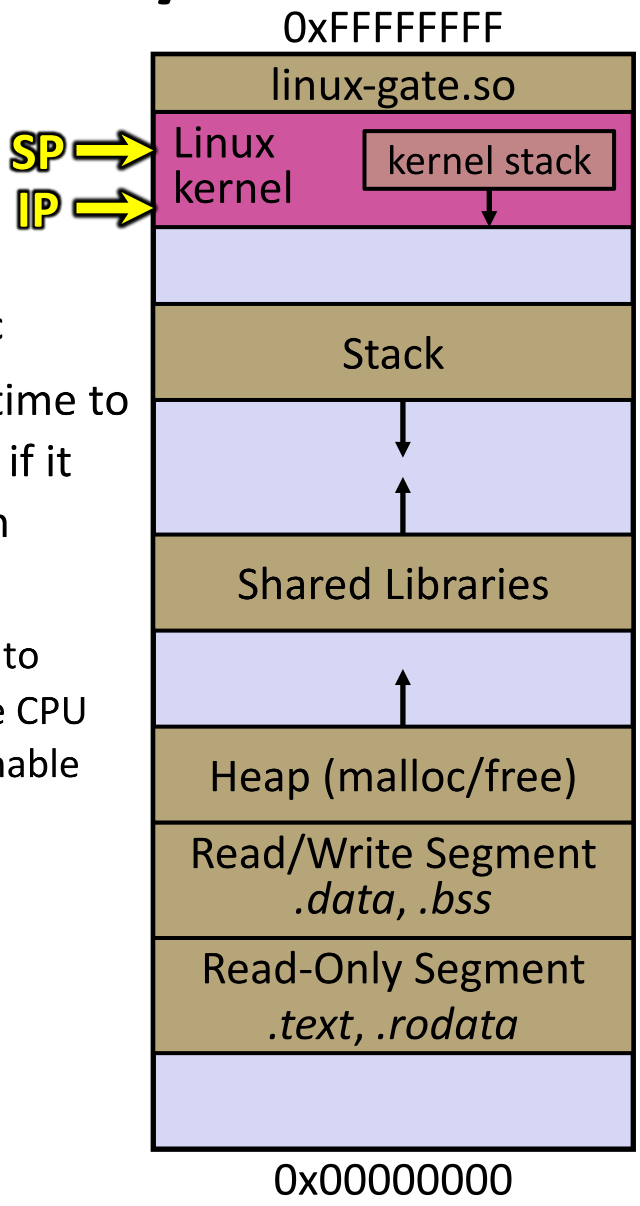
- Is in the architecture-dependent part of Linux
- It's job is to:
  - Look up the system call number in a system call dispatch table
  - Call into the address stored in that table entry; this is Linux's system call handler
    - For `open()`, the handler is named `sys_open`, and is system call #5

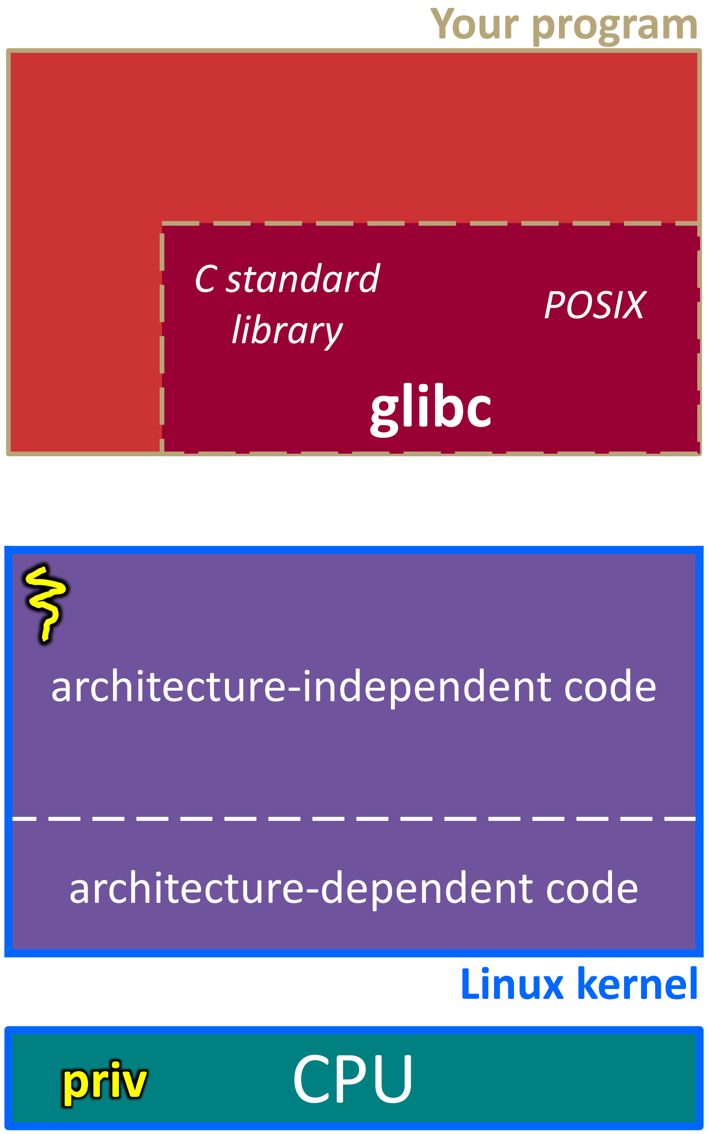


Details on x86/Linux

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

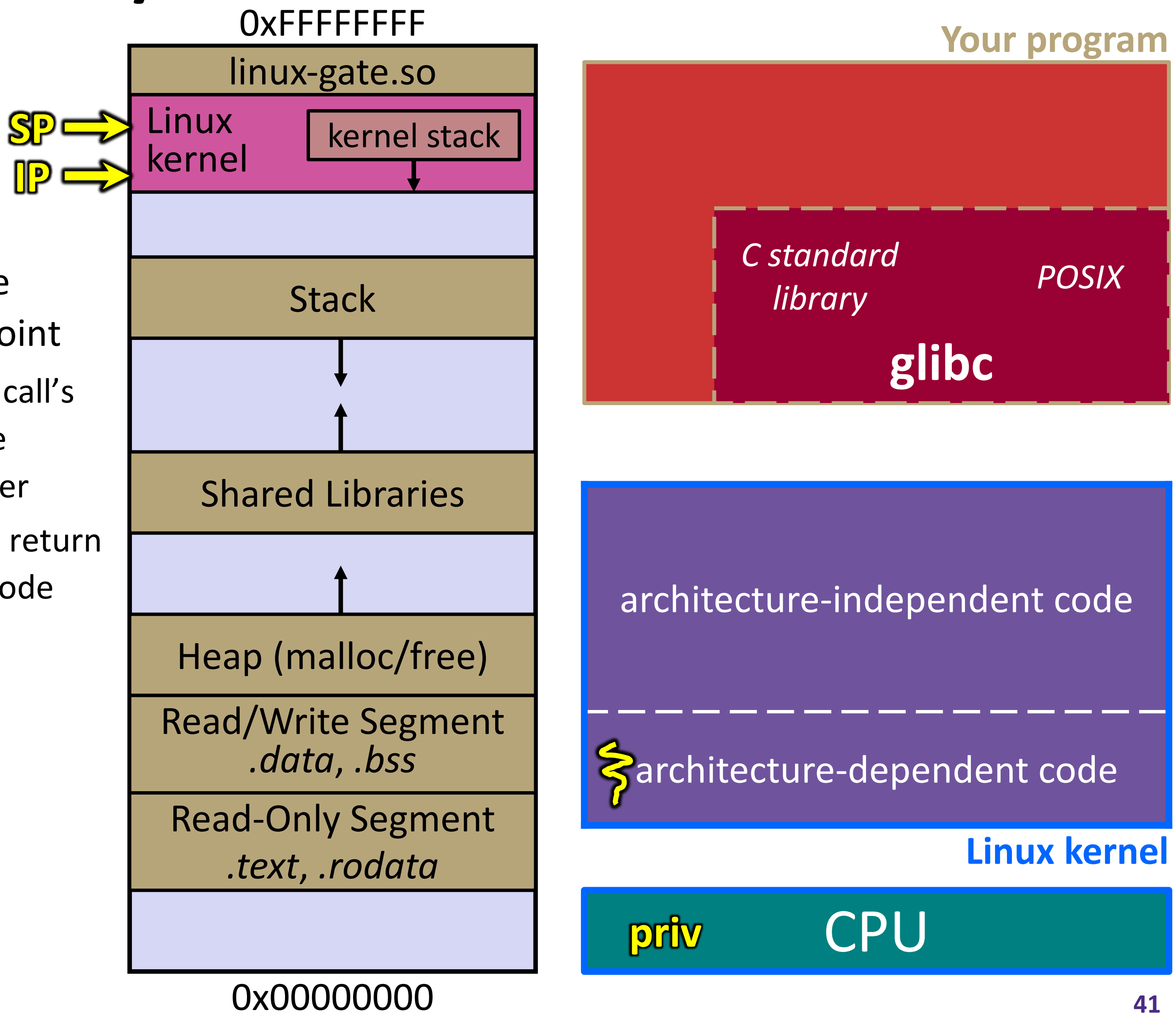




Details on x86/Linux

Eventually, the system call handler finishes

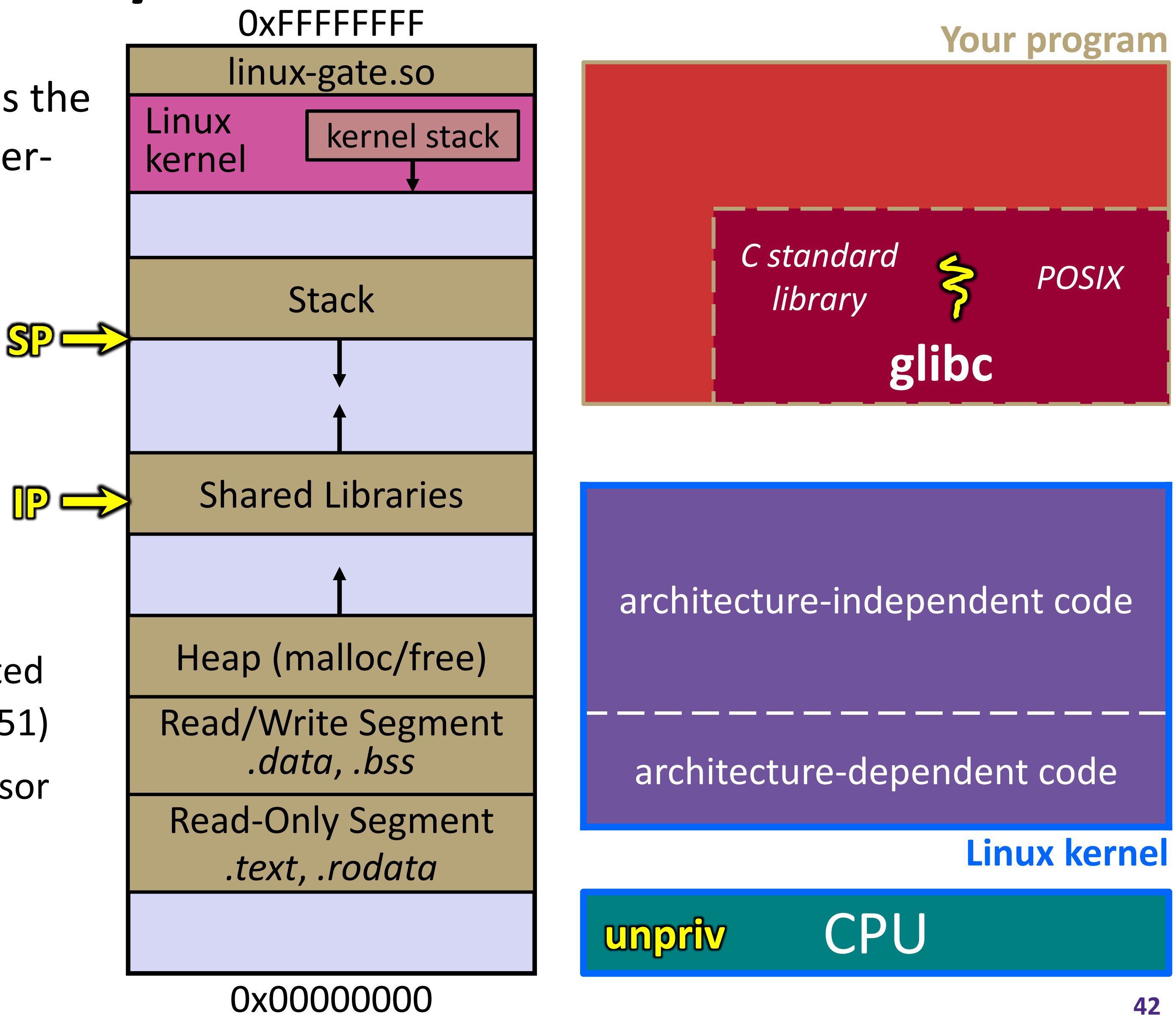
- Returns back to the system call entry point
  - Places the system call's return value in the appropriate register
  - Calls `SYSEXIT` to return to the user-level code



Details on x86/Linux

SYSEXIT transitions the processor back to user-mode code

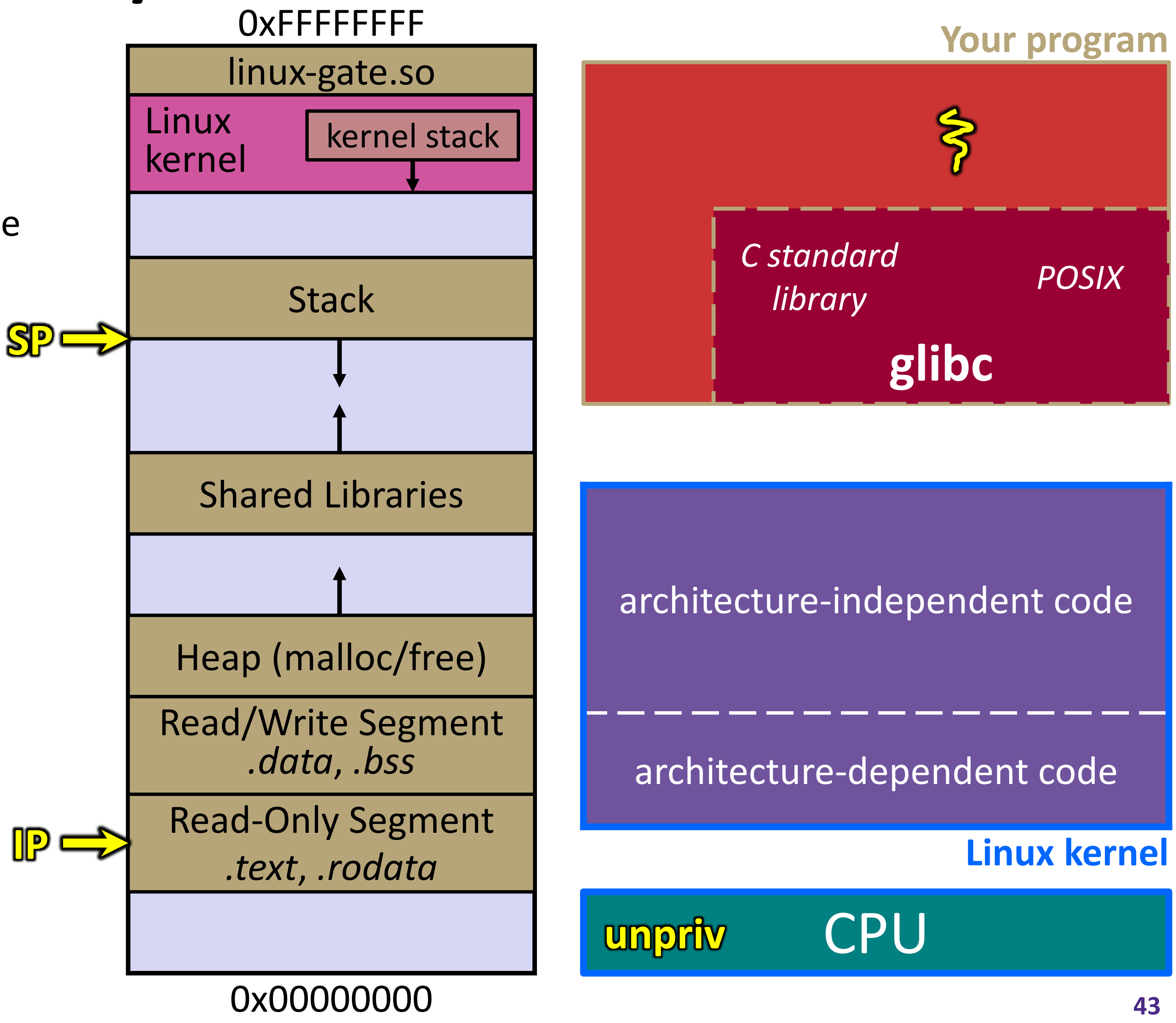
- Restores the IP, SP to user-land values
- Sets the CPU back to unprivileged mode
- Changes some segmentation-related registers (see CSE451)
- Returns the processor back to glibc



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)

Extra Exercise #1

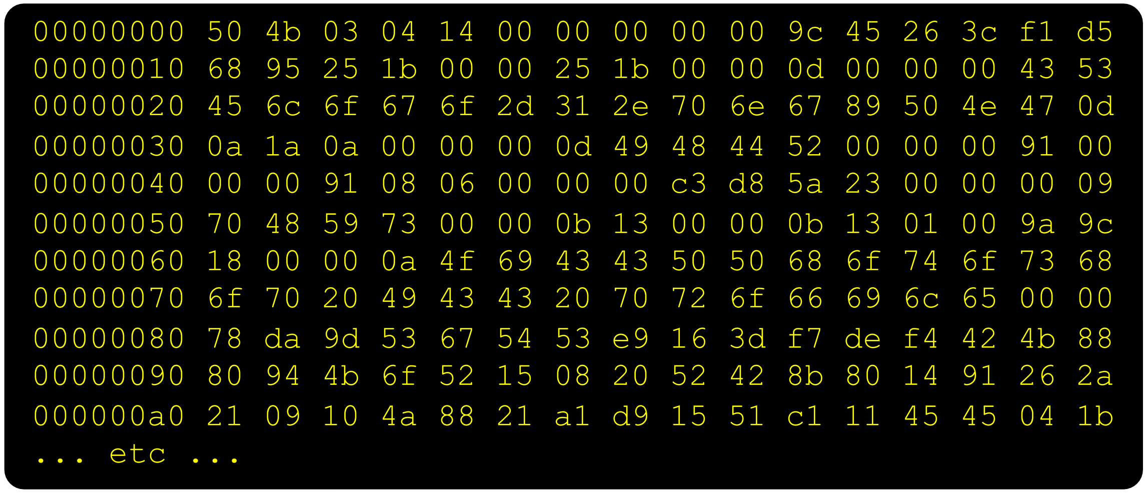
- Write a program that:
  - Uses `argc`/`argv` to receive the name of a text file
  - Reads the contents of the file a line at a time
  - Parses each line, converting text into a `uint32_t`
  - Builds an array of the parsed `uint32_t`’s
  - Sorts the array
  - Prints the sorted array to `stdout`

```
bash$ cat in.txt
1213
3231
000005
52
bash$ ./extra1 in.txt
5
52
1213
3231
bash$
```

- Hint: use `man` to read about `getline`, `sscanf`, `realloc`, and `qsort`

Extra Exercise #2

- Write a program that:
  - Loops forever; in each loop:
    - Prompt the user to input a filename
    - Reads a filename from `stdin`
    - Opens and reads the file
    - Prints its contents to `stdout` in the format shown:



```
00000000 50 4b 03 04 14 00 00 00 00 00 9c 45 26 3c f1 d5
00000010 68 95 25 1b 00 00 25 1b 00 00 0d 00 00 00 43 53
00000020 45 6c 6f 67 6f 2d 31 2e 70 6e 67 89 50 4e 47 0d
00000030 0a 1a 0a 00 00 00 0d 49 48 44 52 00 00 00 91 00
00000040 00 00 91 08 06 00 00 00 c3 d8 5a 23 00 00 00 09
00000050 70 48 59 73 00 00 0b 13 00 00 0b 13 01 00 9a 9c
00000060 18 00 00 0a 4f 69 43 43 50 50 68 6f 74 6f 73 68
00000070 6f 70 20 49 43 43 20 70 72 6f 66 69 6c 65 00 00
00000080 78 da 9d 53 67 54 53 e9 16 3d f7 de f4 42 4b 88
00000090 80 94 4b 6f 52 15 08 20 52 42 8b 80 14 91 26 2a
000000a0 21 09 10 4a 88 21 a1 d9 15 51 c1 11 45 45 04 1b
... etc ...
```

- Hints:
  - Use `man` to read about `fgets`
  - Or, if you're more courageous, try `man 3 readline` to learn about `libreadline.a` and Google to learn how to link to it