Buffering, Syscalls, Make

CSE 333 Autumn 2019

Instructor: Hannah C. Tang

Teaching Assistants:

Dao Yi	Farrell Fileas	Lukas Joswiak
Nathan Lipiarski	Renshu Gu	Travis McGaha
Yibo Cao	Yifan Bai	Yifan Xu

Poll Everywhere

pollev.com/cse333

About how long did Homework 1 take?

A. 0-3 Hours
B. 3-6 Hours
C. 6-9 Hours
D. 9-12 Hours
E. 12+ Hours
F. I haven’t finished yet / I prefer not to say

Administrivia

- Homework 2 released Monday (10/14)
- Exercise 6 posted NOW (← *anon. f/b!* ☺), due Monday (10/14)
- Late policy reminder:
  - Max of two days per HW; weekends count as 1 day
  - 1 late day = tonight @ 8:59pm, 2 late days = Sunday @ 8:59pm
- Extra OH w/Travis today! 3-5pm @ 4th floor breakout

Lecture Outline

- **Another Difference: C Stream Buffering**
- Another Difference: What is a System Call?
- Make

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?

- Nicer API!
  - Compare C's fread() vs POSIX's read(); no need to handle `EINTR`
- Performance!
  - Grouping small writes into a larger write = fewer disk accesses

Disk Latency = 😱😱😱

- Jeff Dean's "Numbers Everyone Should Know" (from LADIS '09)

Numbers Everyone Should Know

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	100 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	10,000 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 network	10,000,000 ns
Read 1 MB sequentially from disk	30,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

Google

the ~10 slides or so after this one are great system design slides

Constants no longer accurate, but orders of magnitude are

Why NOT buffer?

- Reliability!
  - Your computer loses power before the buffer is flushed
  - Your program assumes data is written to a file and signals another program to read it
- Performance!
  - Data is *copied* 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

Disabling C's Buffering

- Explicitly turn off with **`setbuf`**`(stream, NULL)`
- Use POSIX APIs instead of C's
  - No buffering is done at the user level
- But… what about the layers below?
  - The OS caches disk reads and writes in the FS *buffer* cache
  - Disk controllers have caches too!

Lecture Outline

- Another Difference: C Stream Buffering
- **Another Difference: What is a System Call?**
- Make

C Standard Lib vs POSIX

Choosing between the two isn't choosing between two equiv. options; it's choosing between high level vs low level

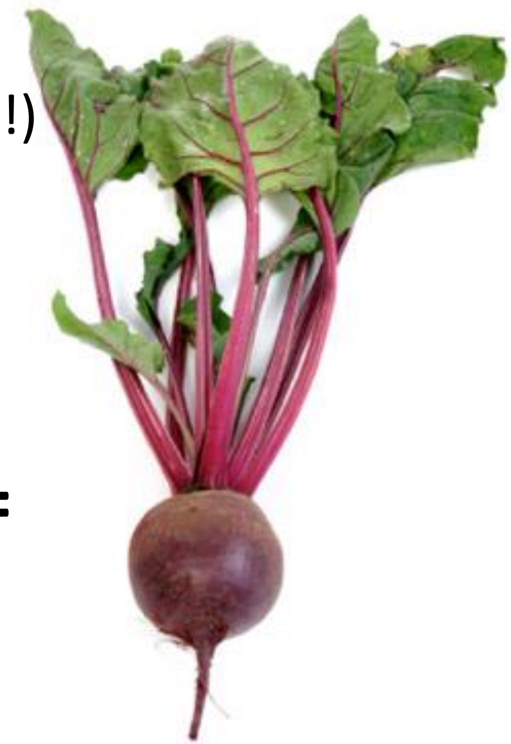
- Thus far, we know:
  - C standard library implements a subset of POSIX (eg, POSIX provides directory manipulation)
  - C standard library implements buffering
  - C standard library has a nicer API (WTF EINTR?!?!)

Posix

+

C stdlib

=



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, Win32, etc.

File System

- open(), read(), write(), close(), …

Network Stack

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

Virtual Memory

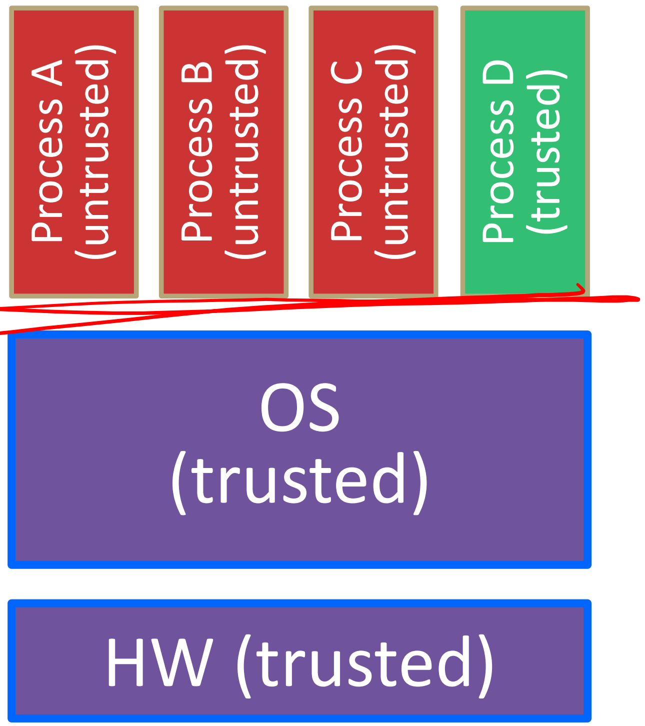
- brk(), shm_open(), …

Process Management

- fork(), wait(), nice(), …

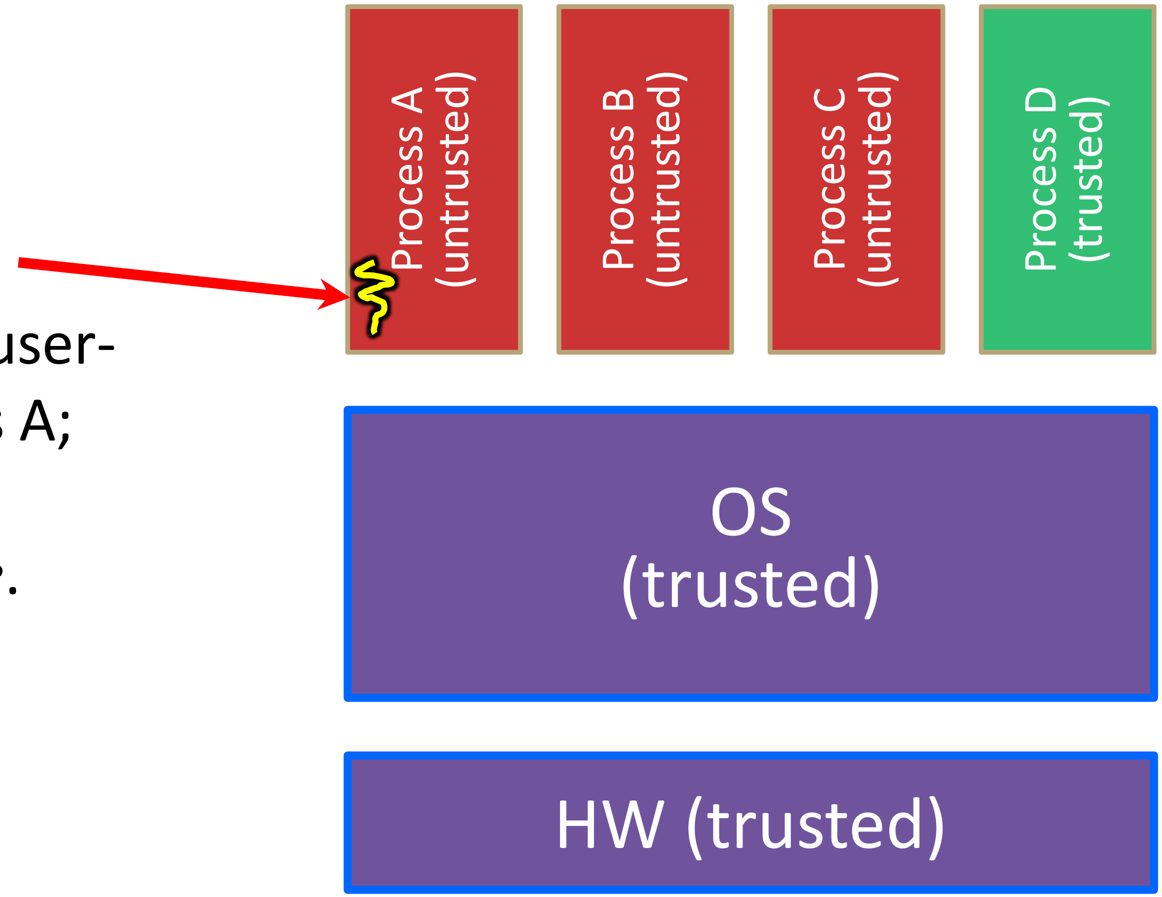
OS: Protection System

- 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



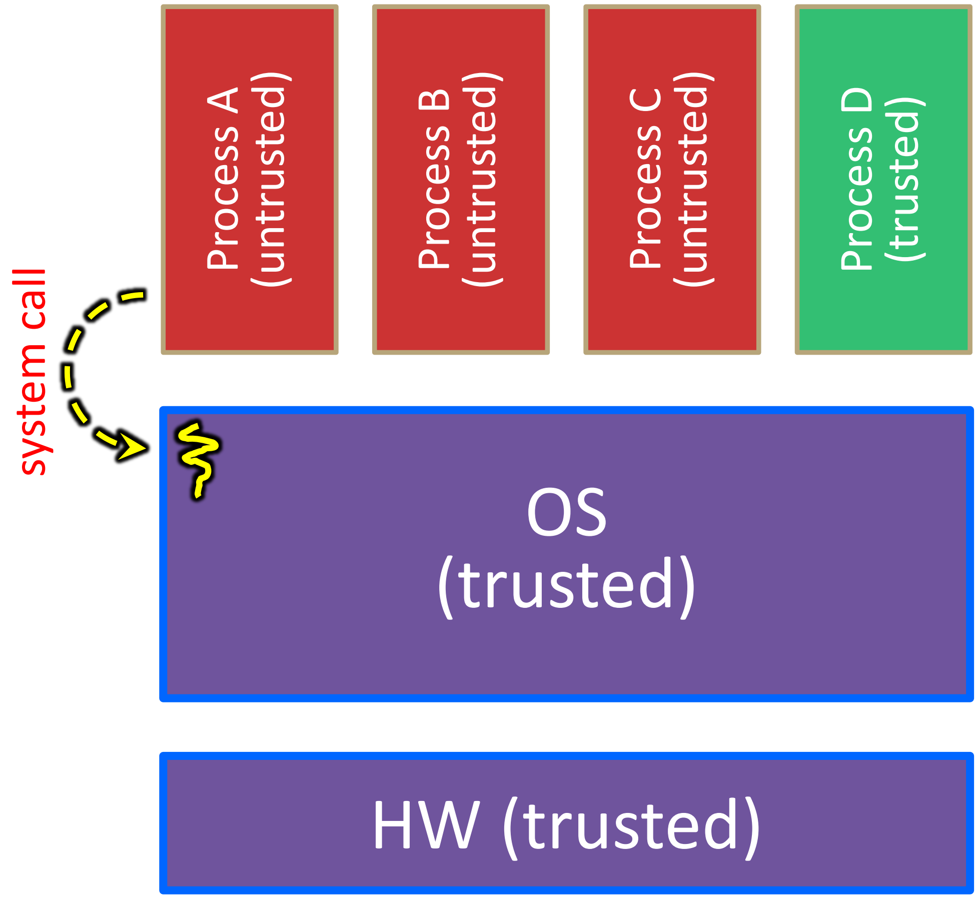
System Call Trace (high level)

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



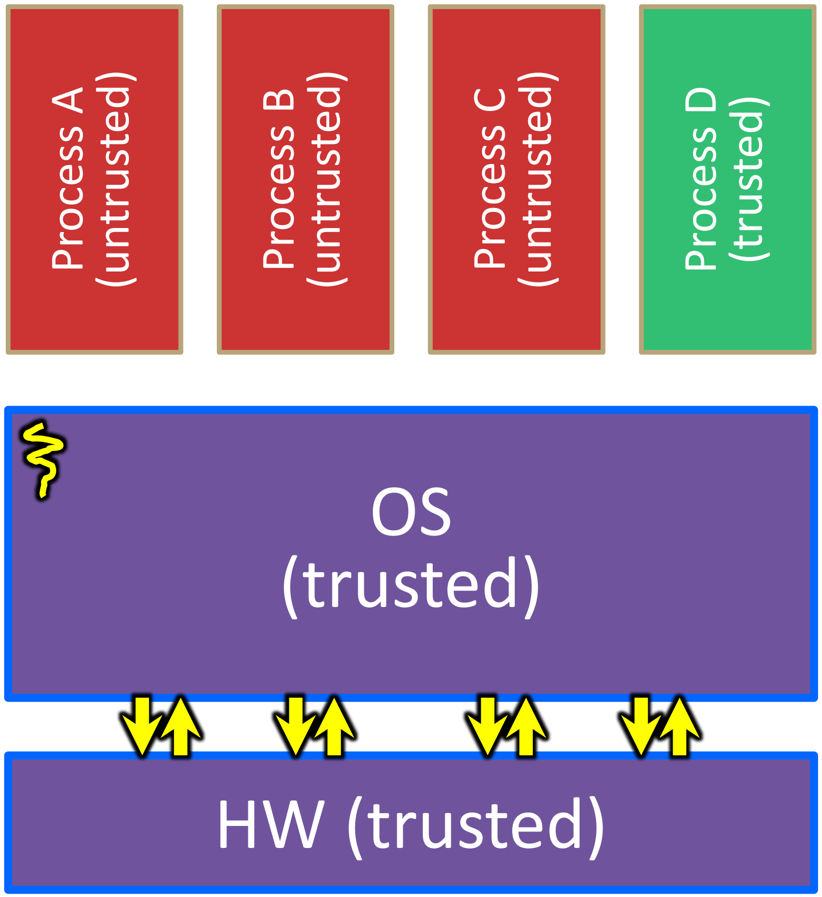
System Call Trace (high level)

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)

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.

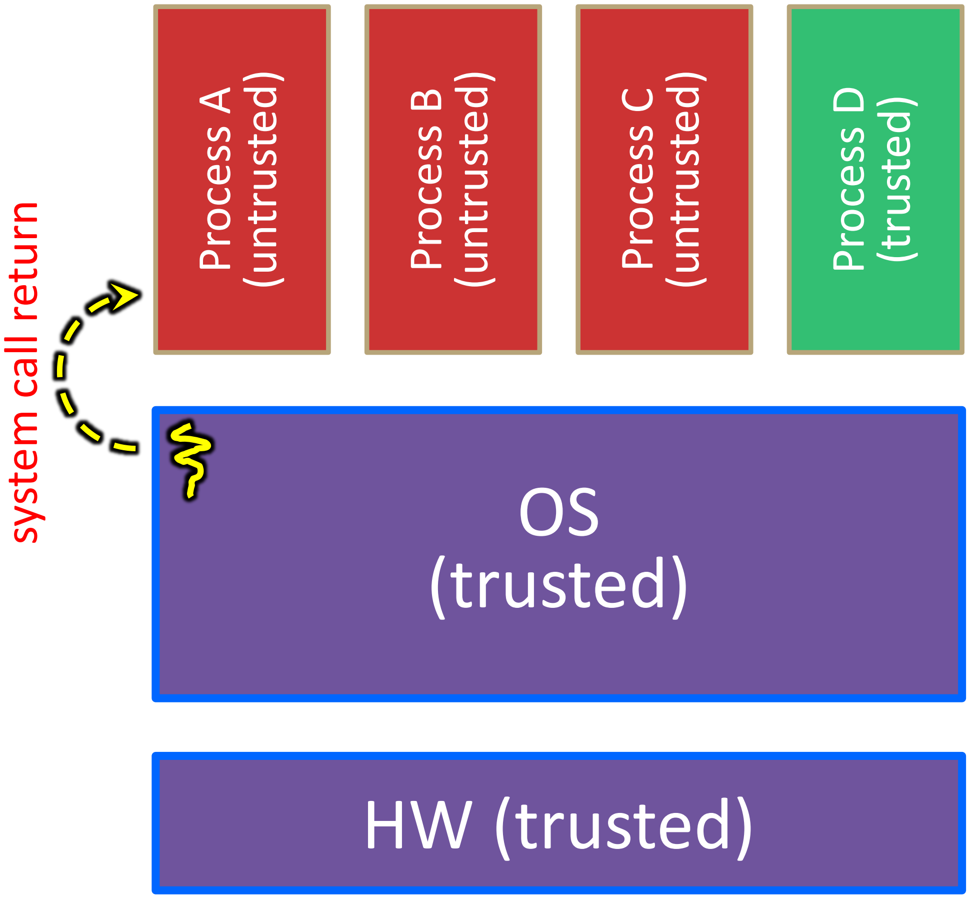


System Call Trace (high level)

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

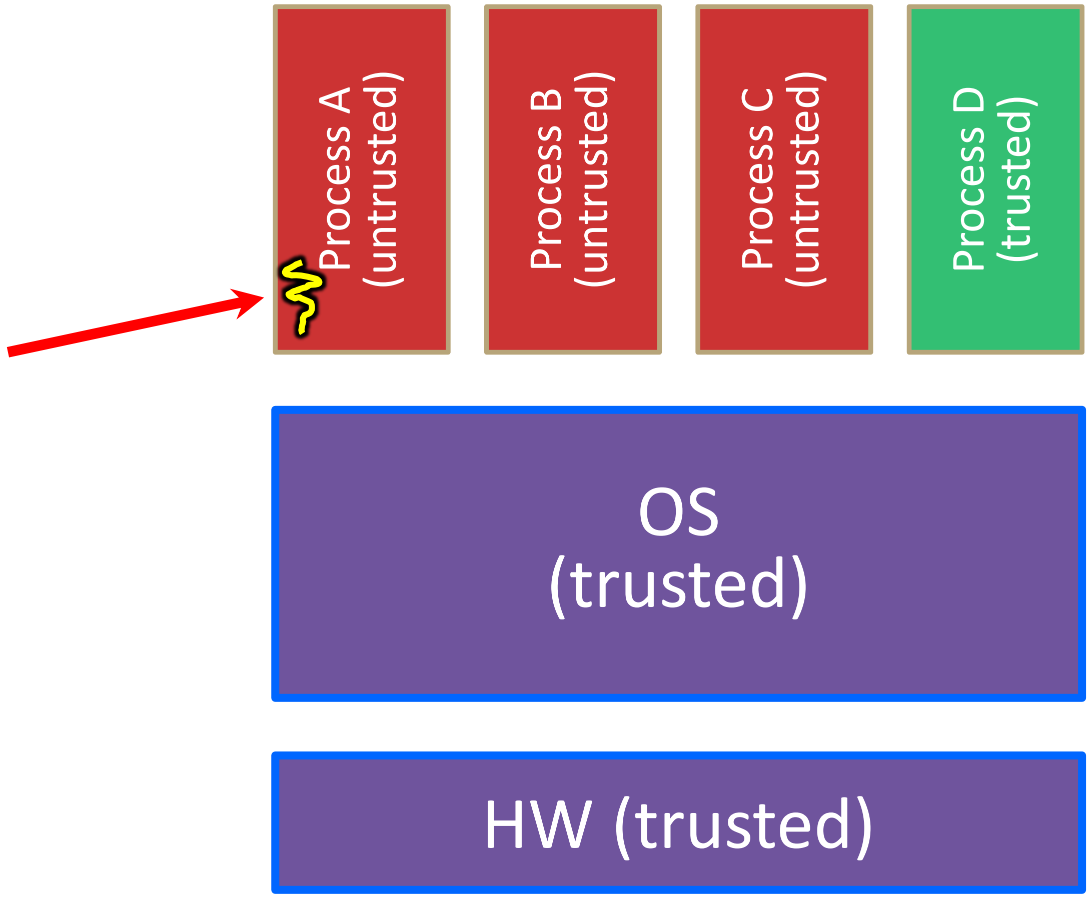
(1) Sets the CPU back to unprivileged mode and

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



System Call Trace (high level)

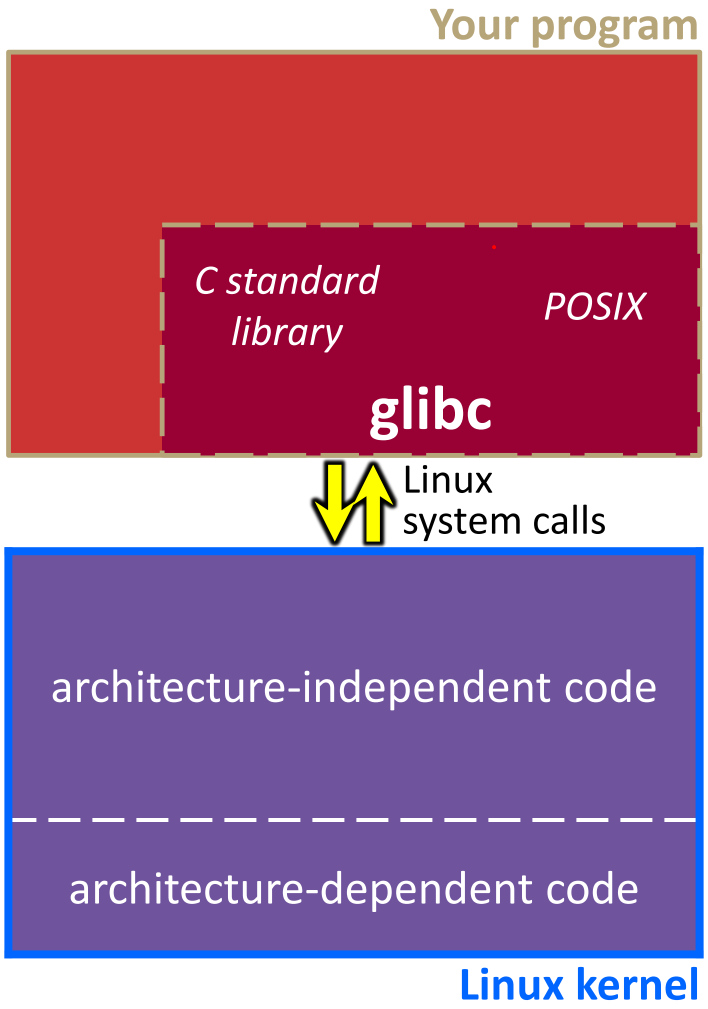
The process continues executing whatever code is next after the system call invocation.



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

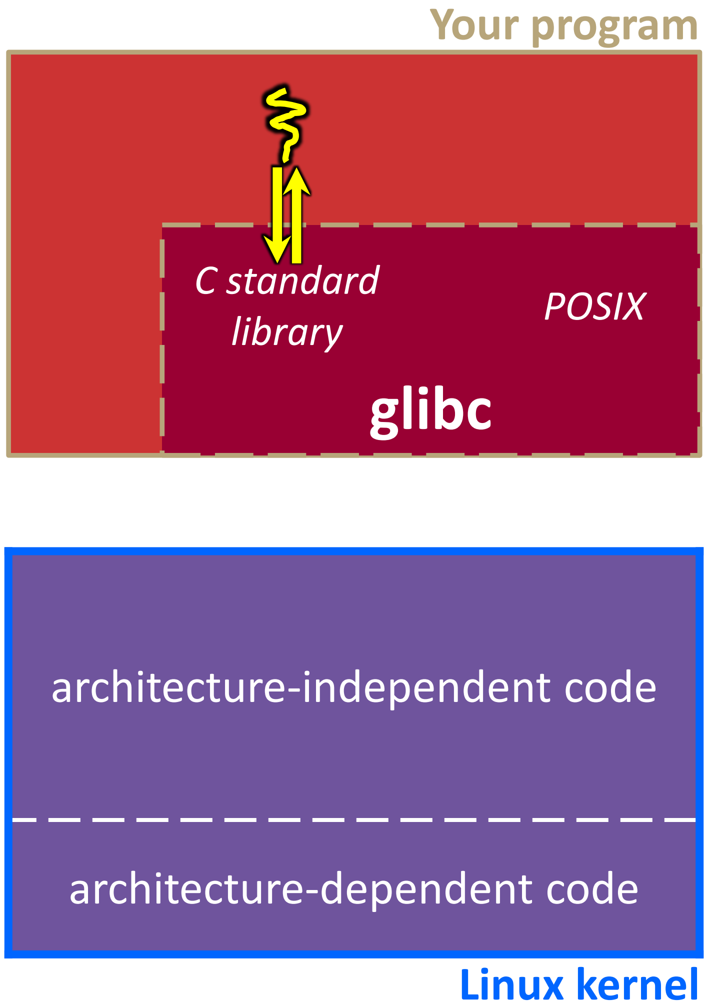
"Library calls" on x86/Linux

- 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



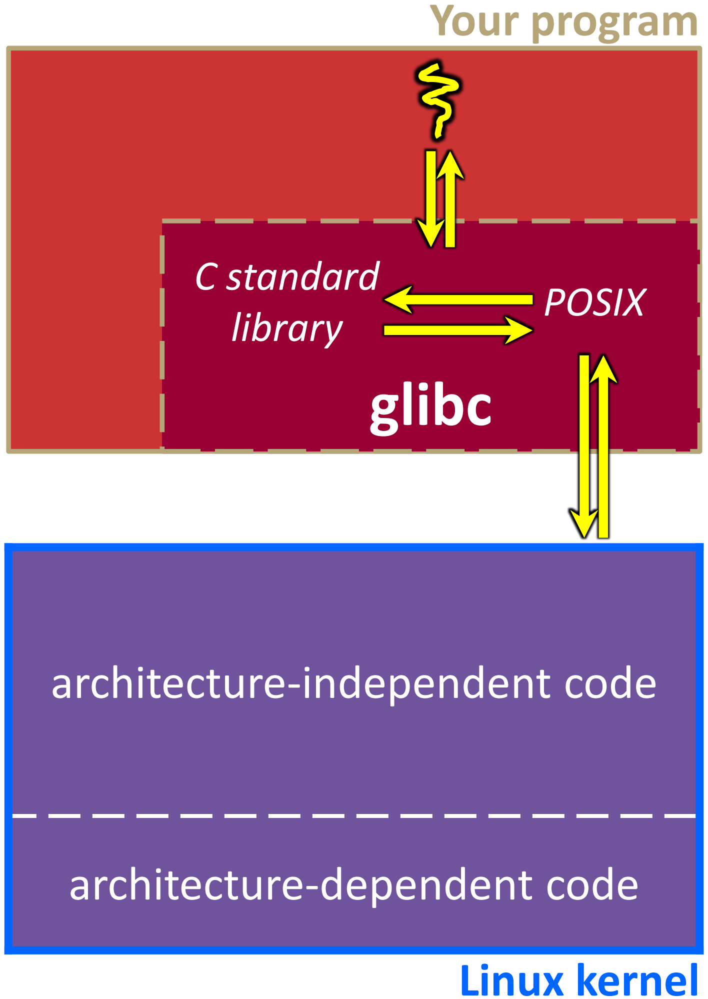
"Library calls" on x86/Linux: Option 1

- 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!



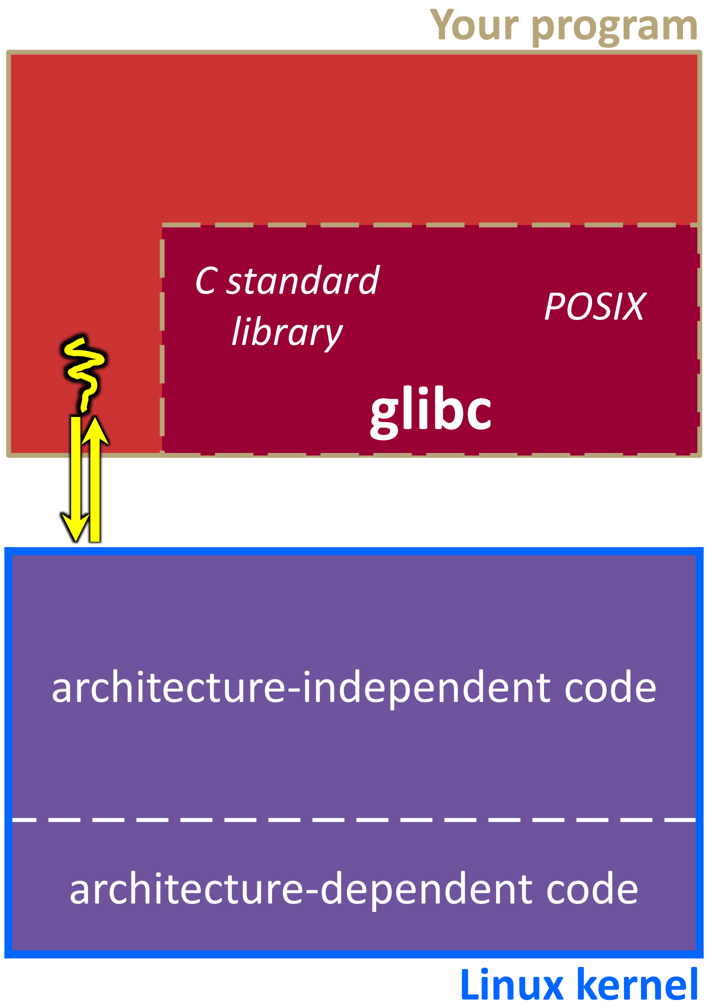
"Library calls" on x86/Linux: Option 2

- 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.



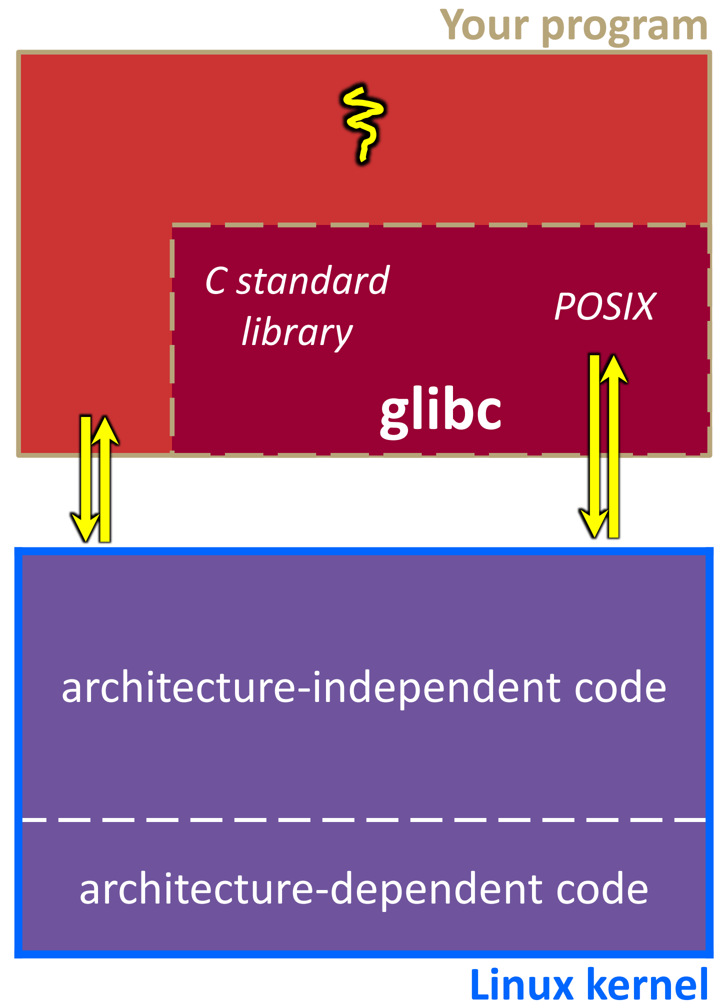
"Library calls" on x86/Linux: Option 3

- 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



System Calls on x86/Linux

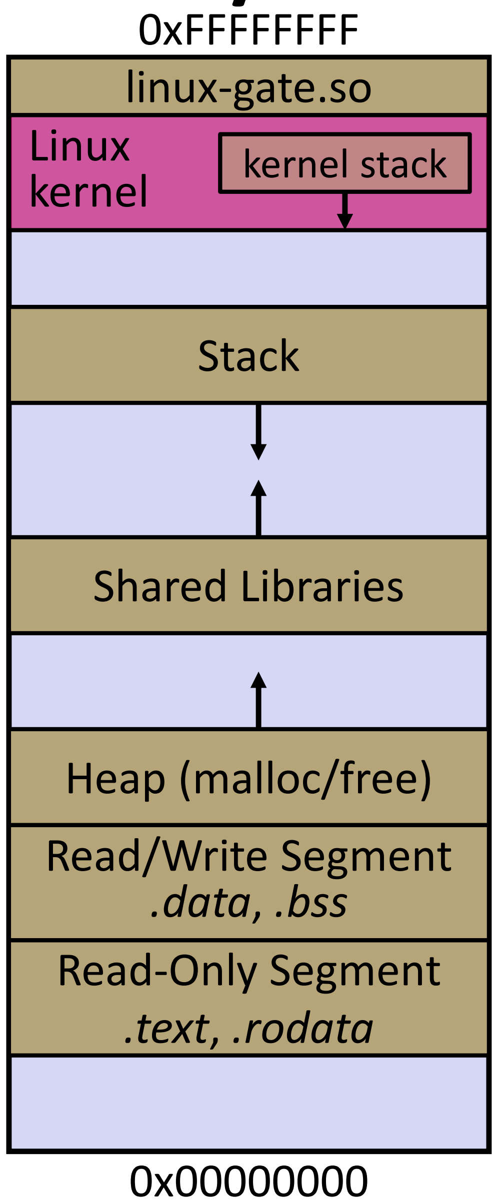
- Let's walk through how a Linux system call actually works
  - We'll assume *32-bit x86* using the modern SYSENTER / SYSEXIT x86 instructions
    - x86-64 code is similar, though details always change over time, so take this as an example – not a debugging guide

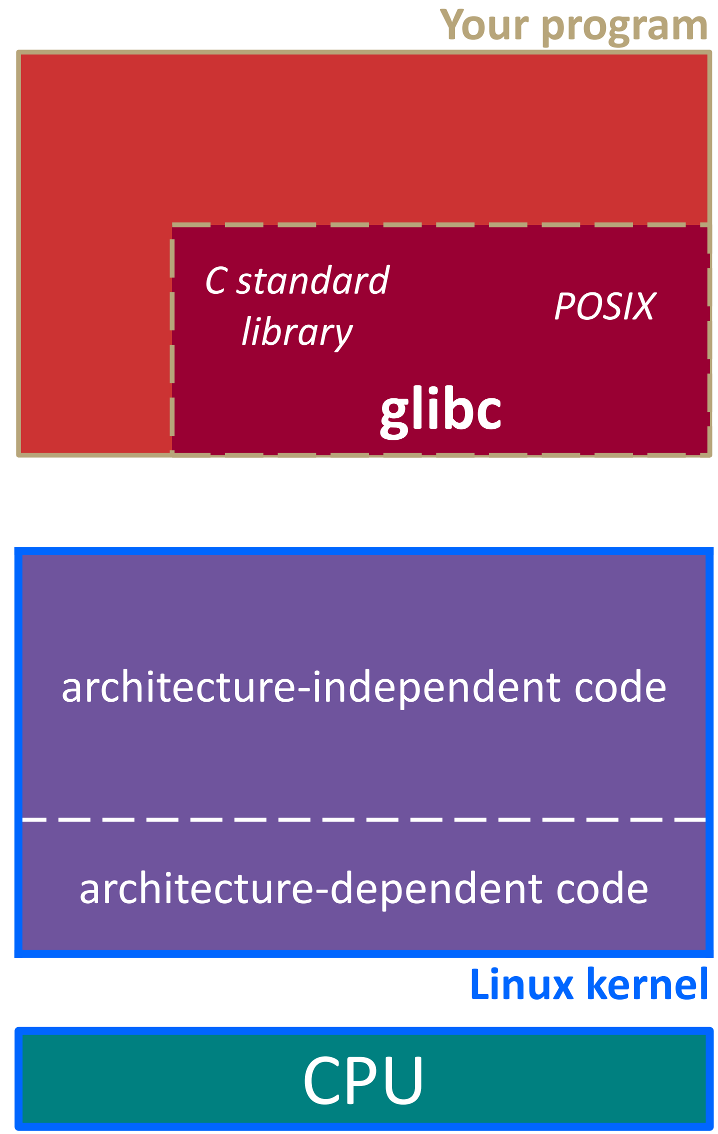


System Calls on x86/Linux

Remember our process address space picture?

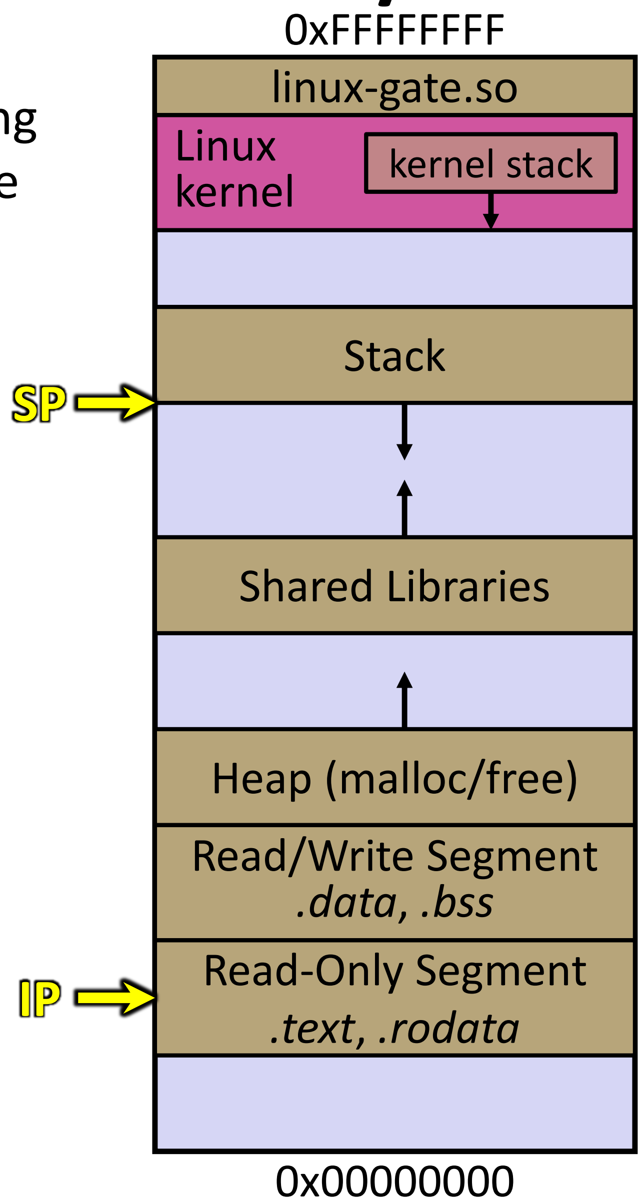
- Let's add some details:

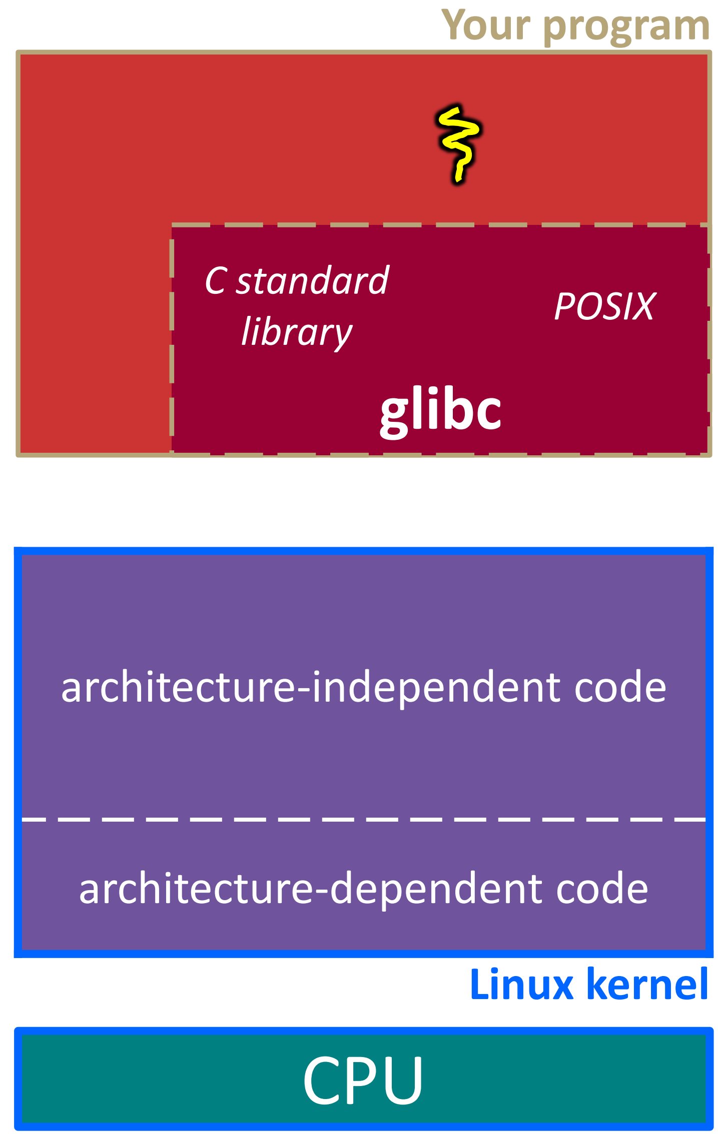




System Calls on x86/Linux

Process is executing your program code

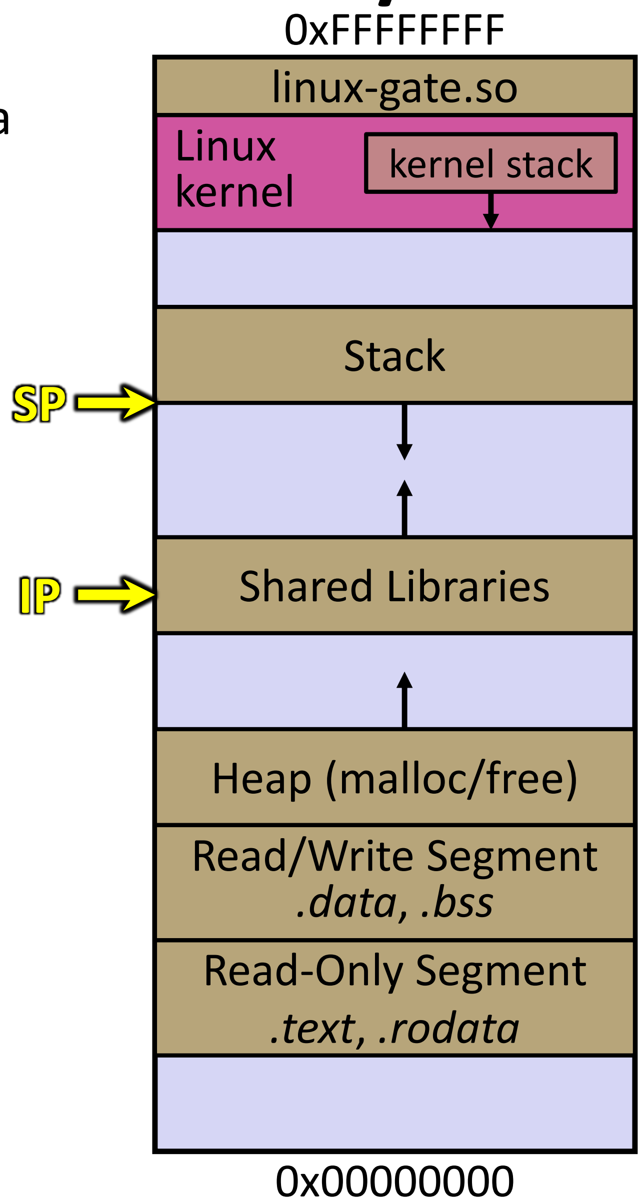


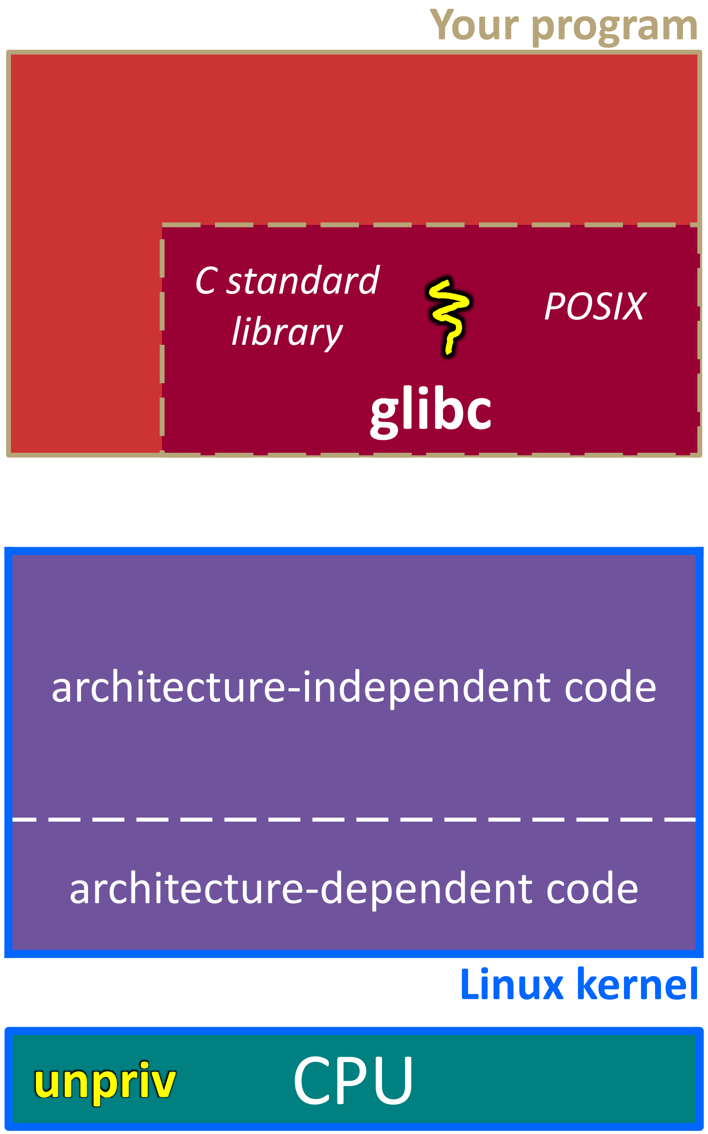


System Calls on x86/Linux

Process calls into a `glibc` function

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

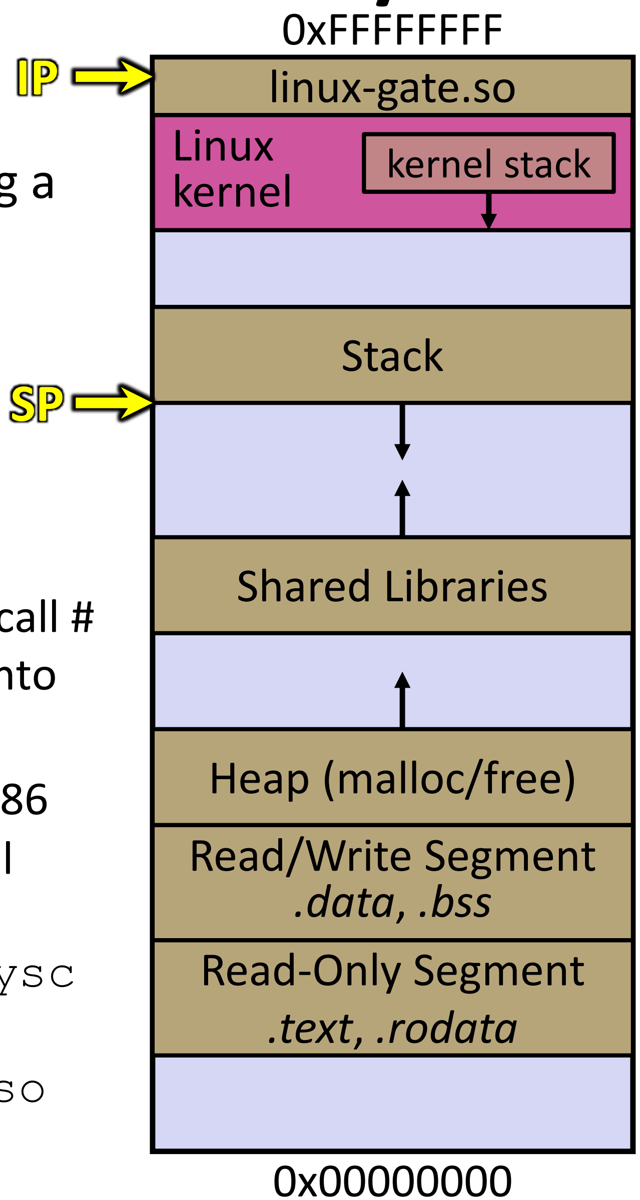


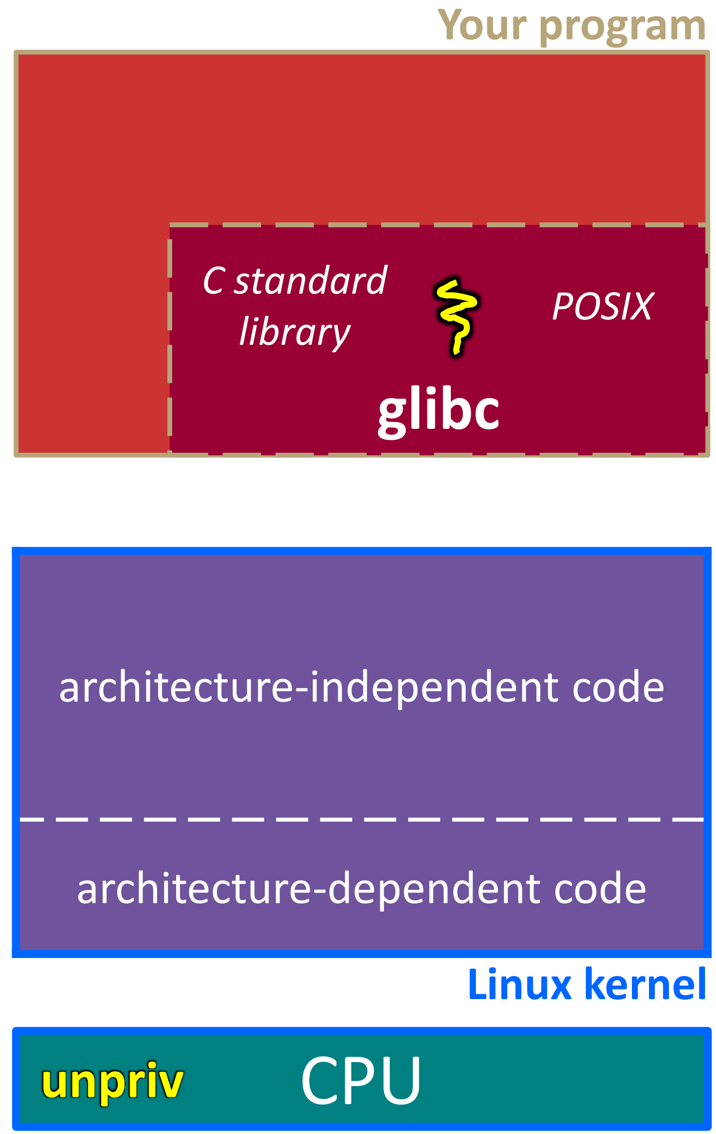


System Calls on x86/Linux

`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`

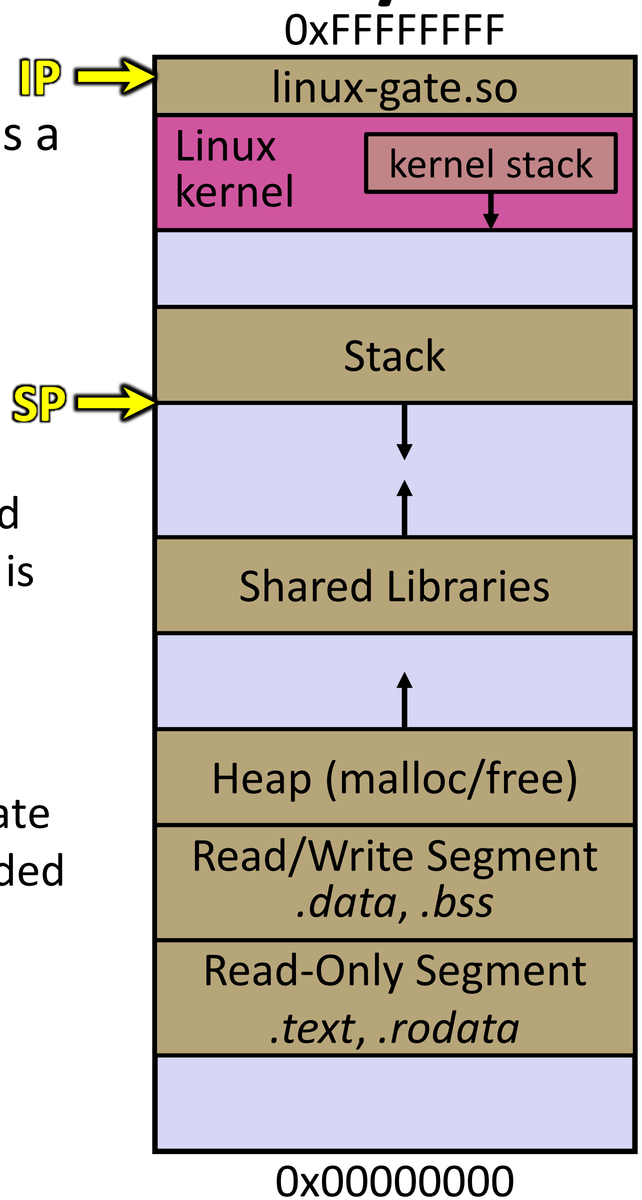


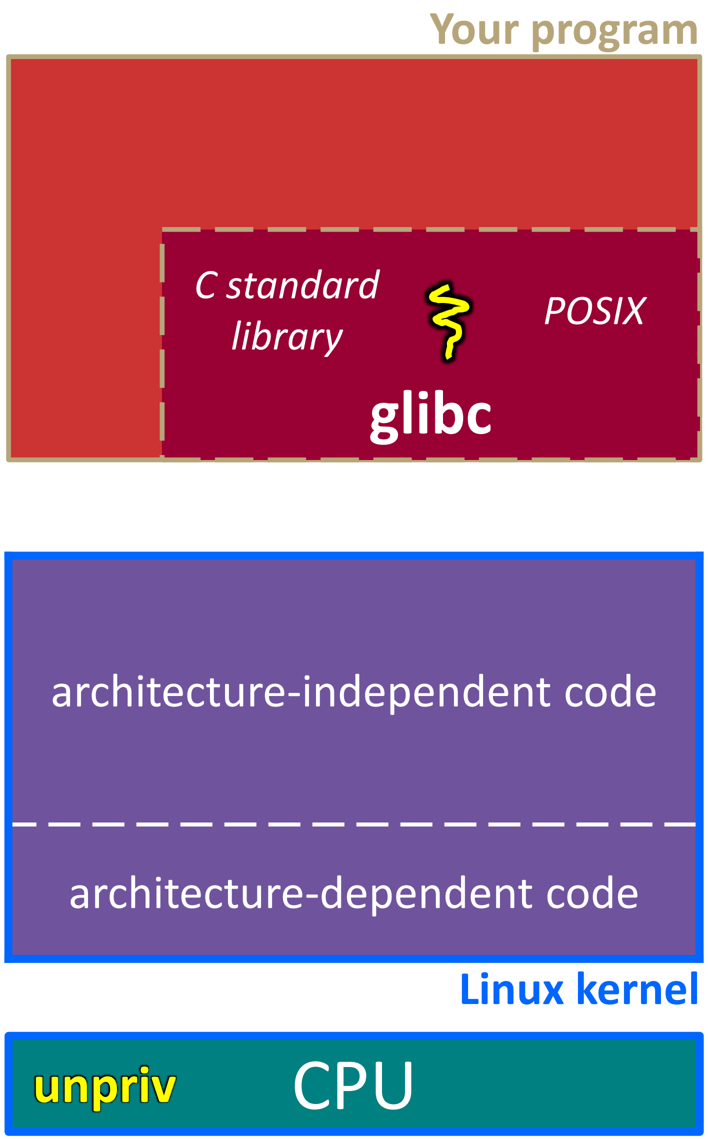


System Calls on x86/Linux

`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

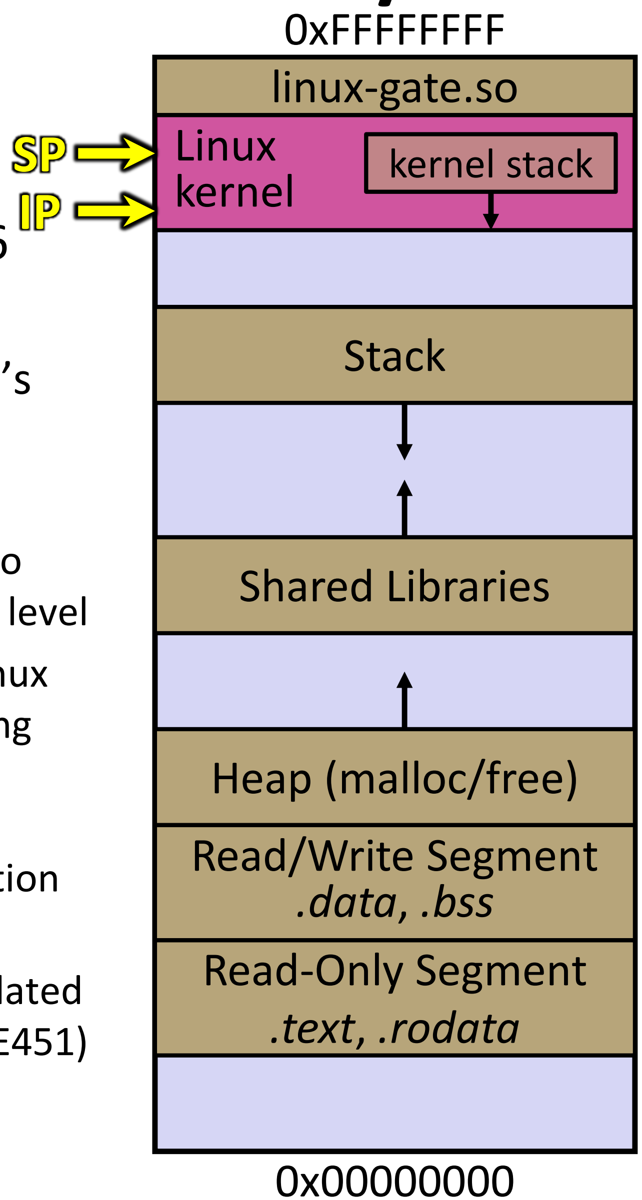


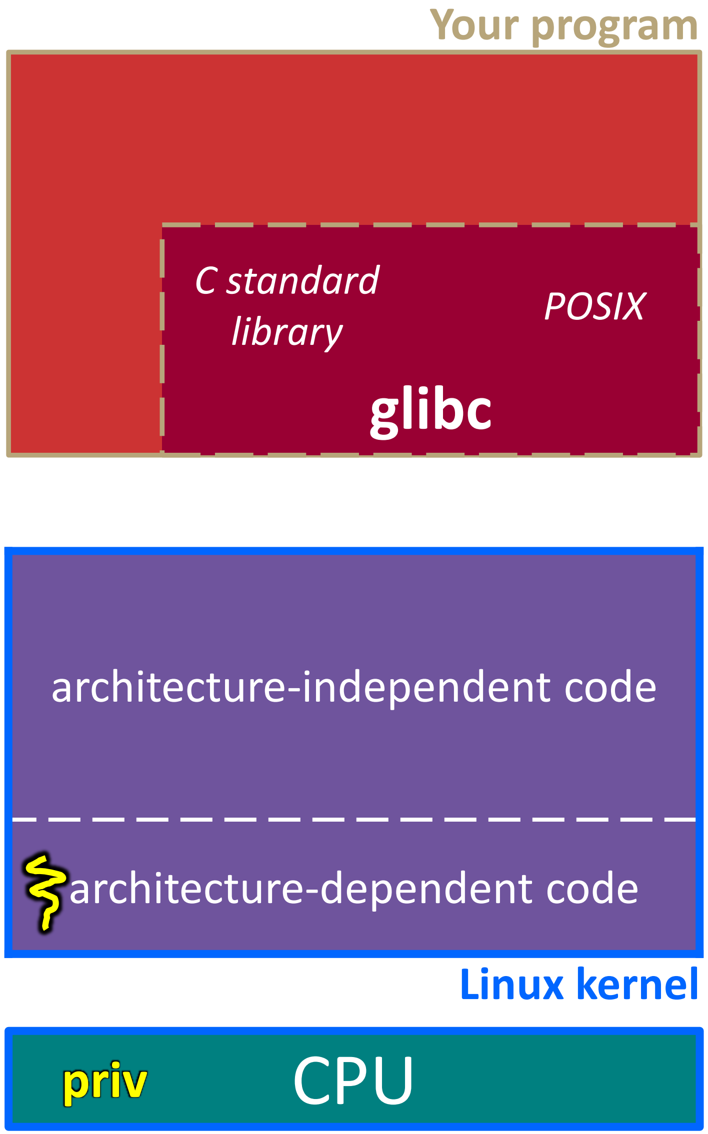


System Calls on x86/Linux

`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)

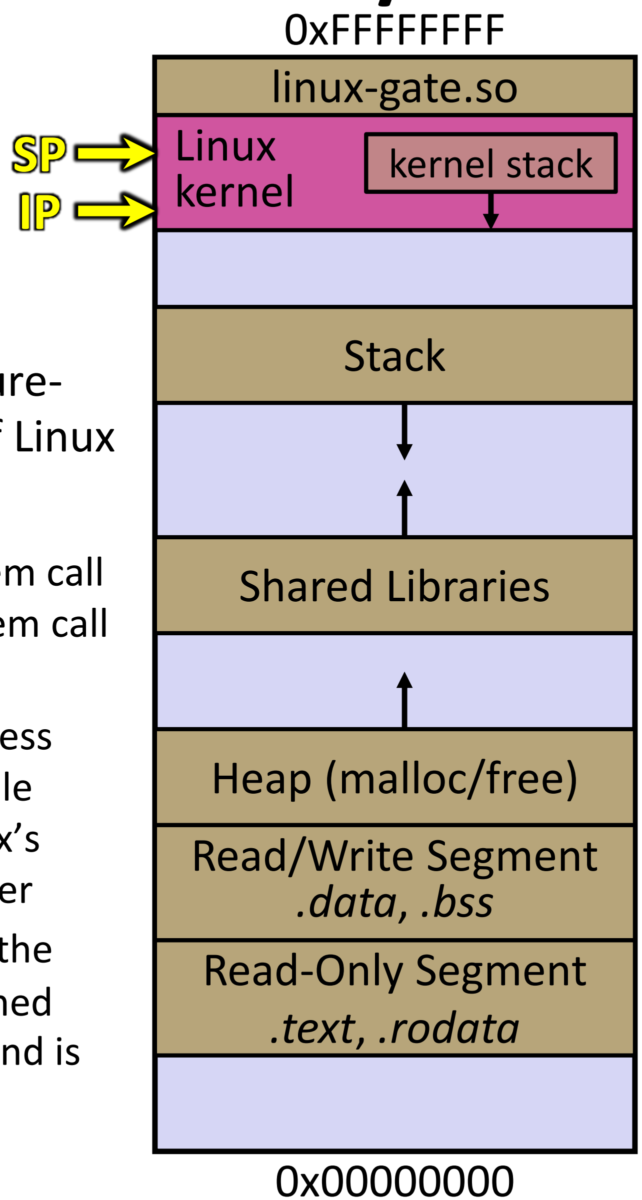


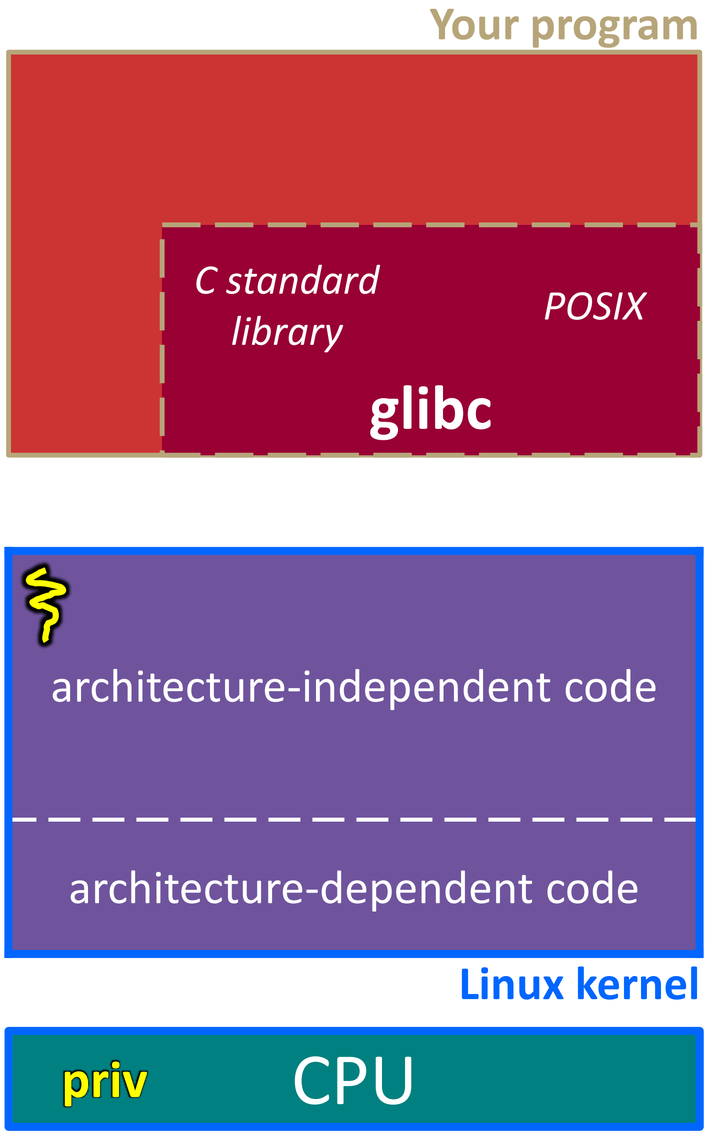


System Calls on x86/Linux

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

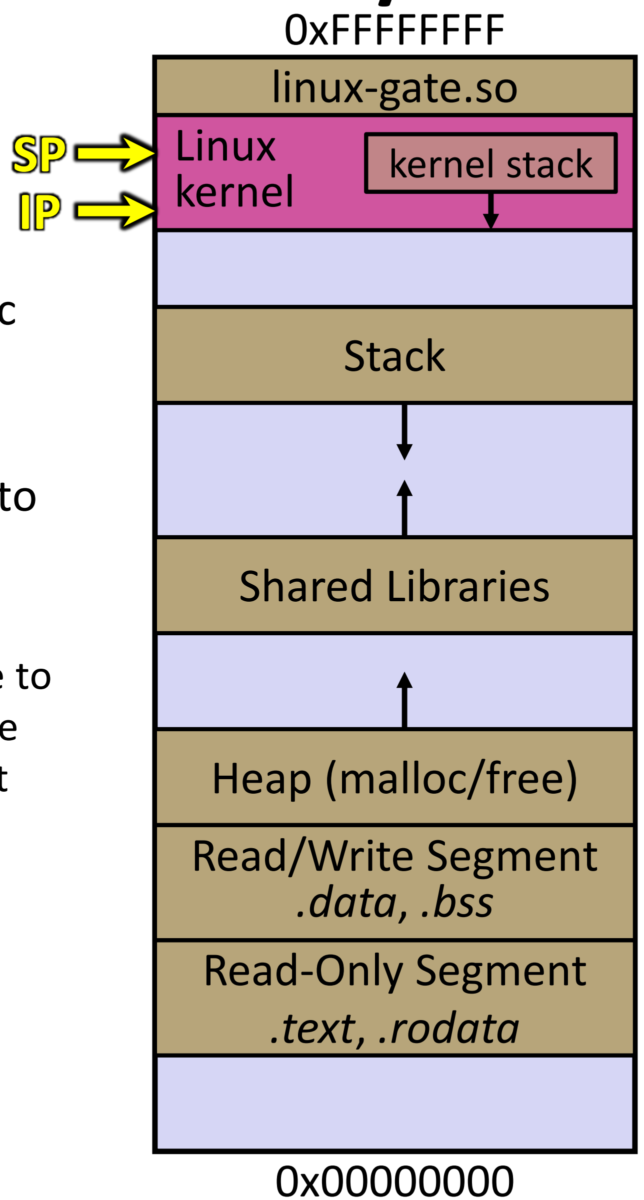


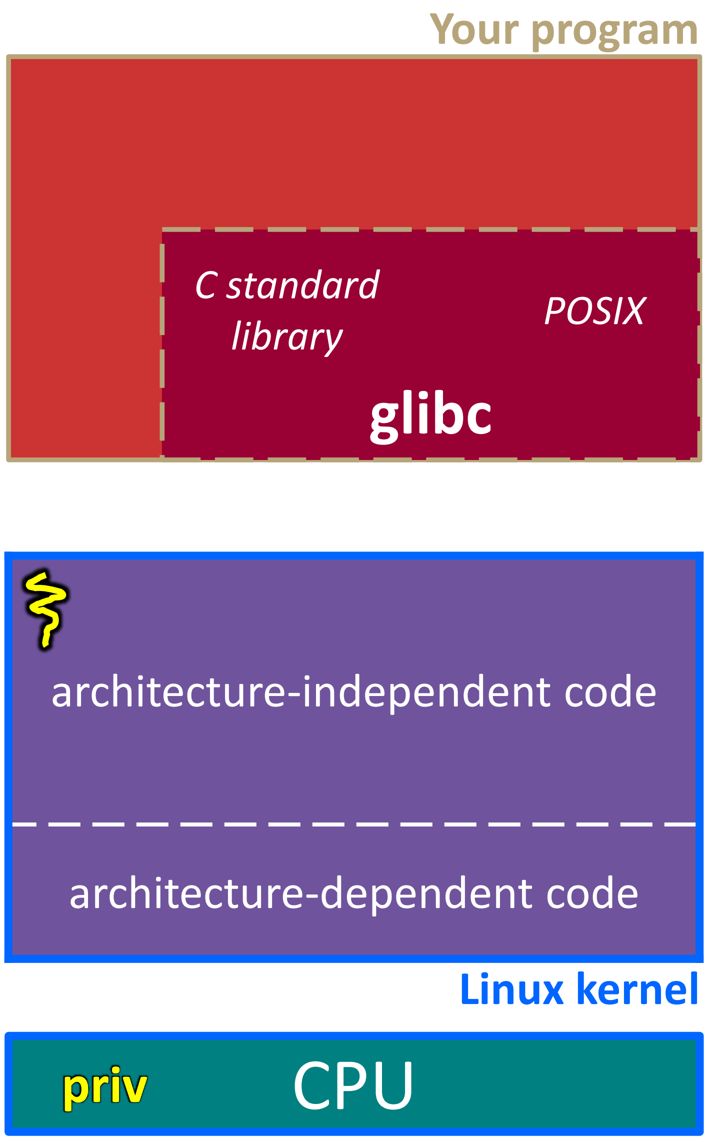


System Calls on x86/Linux

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

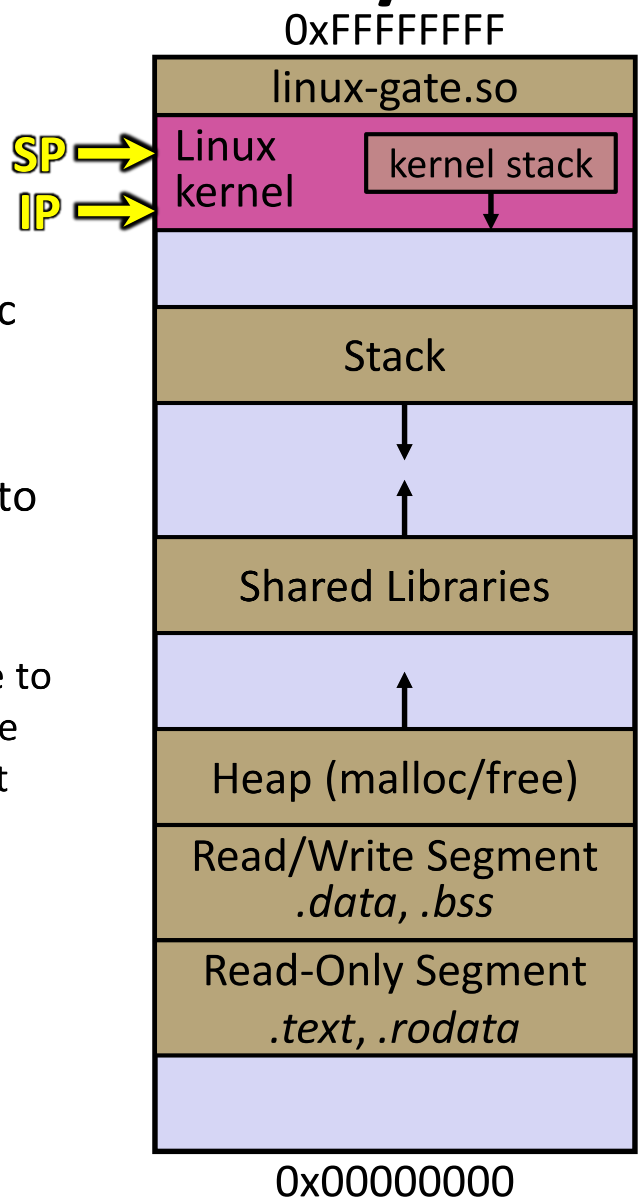


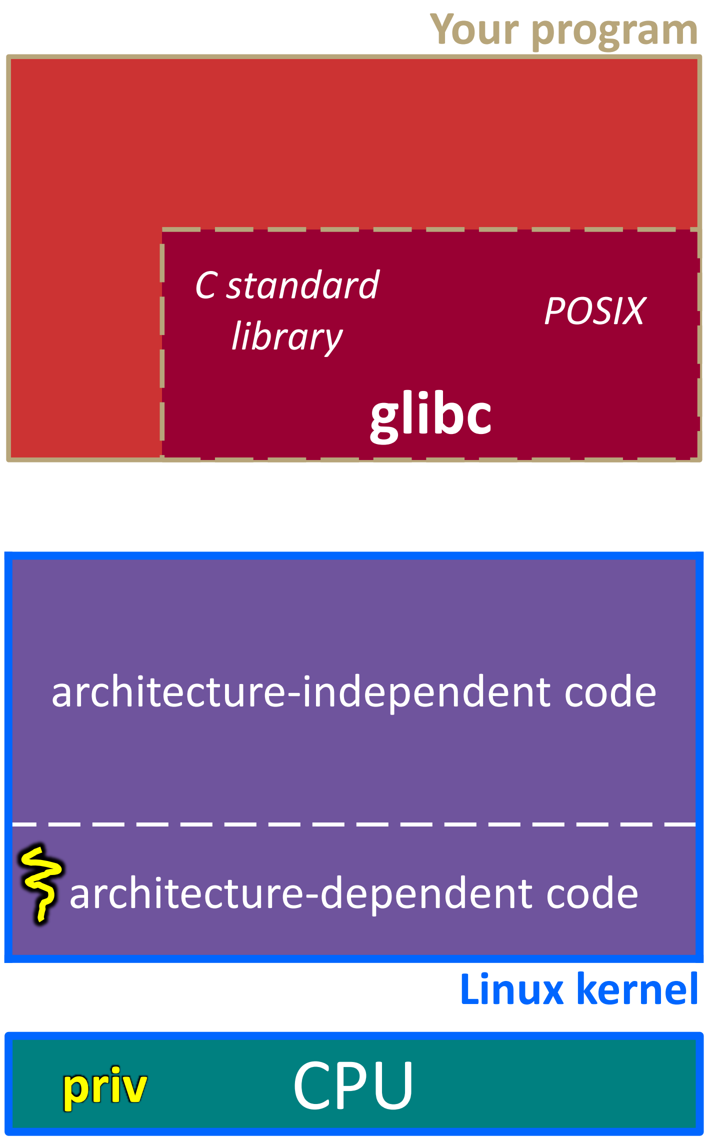


System Calls on x86/Linux

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

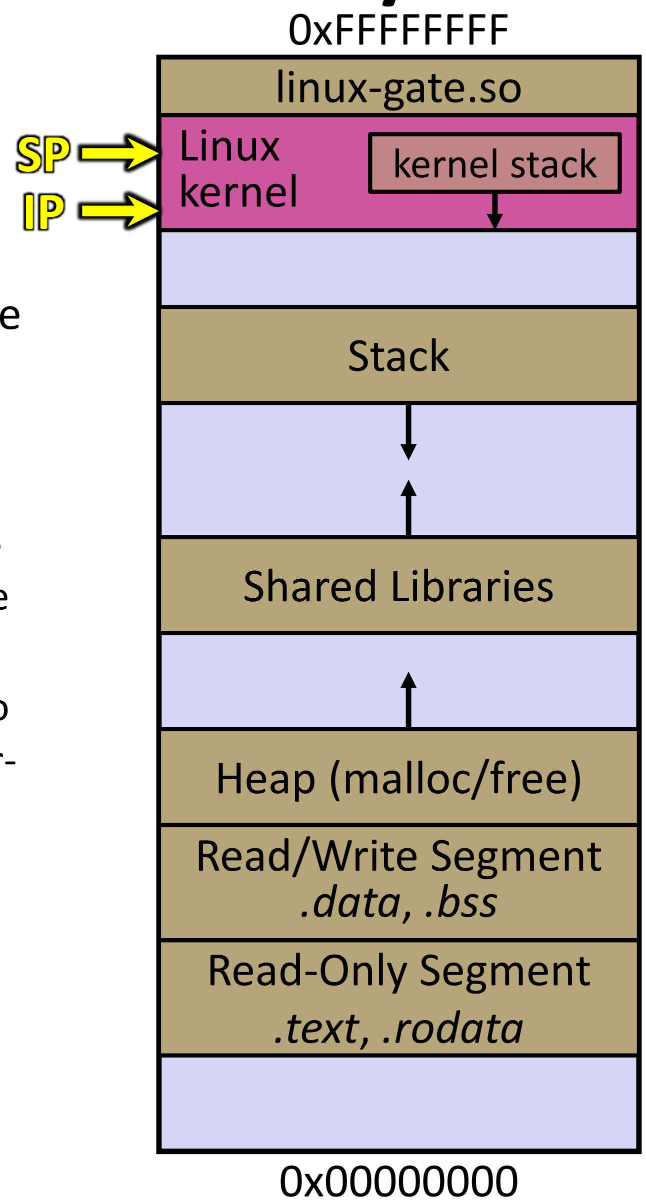


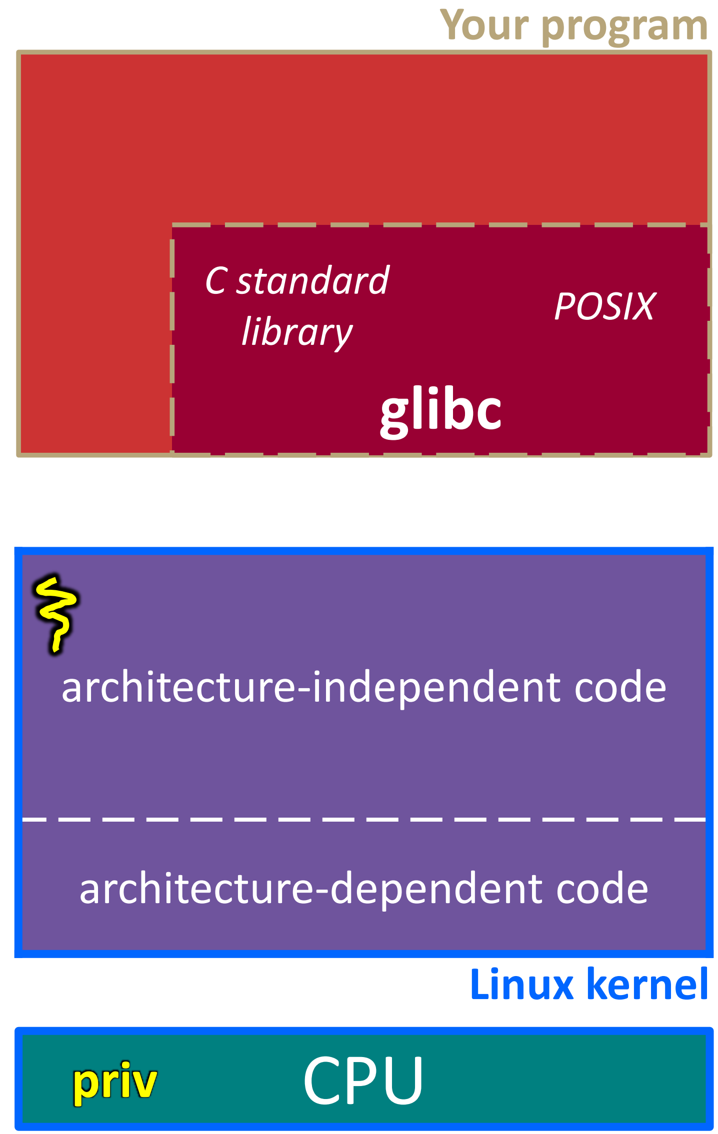


System Calls on x86/Linux

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

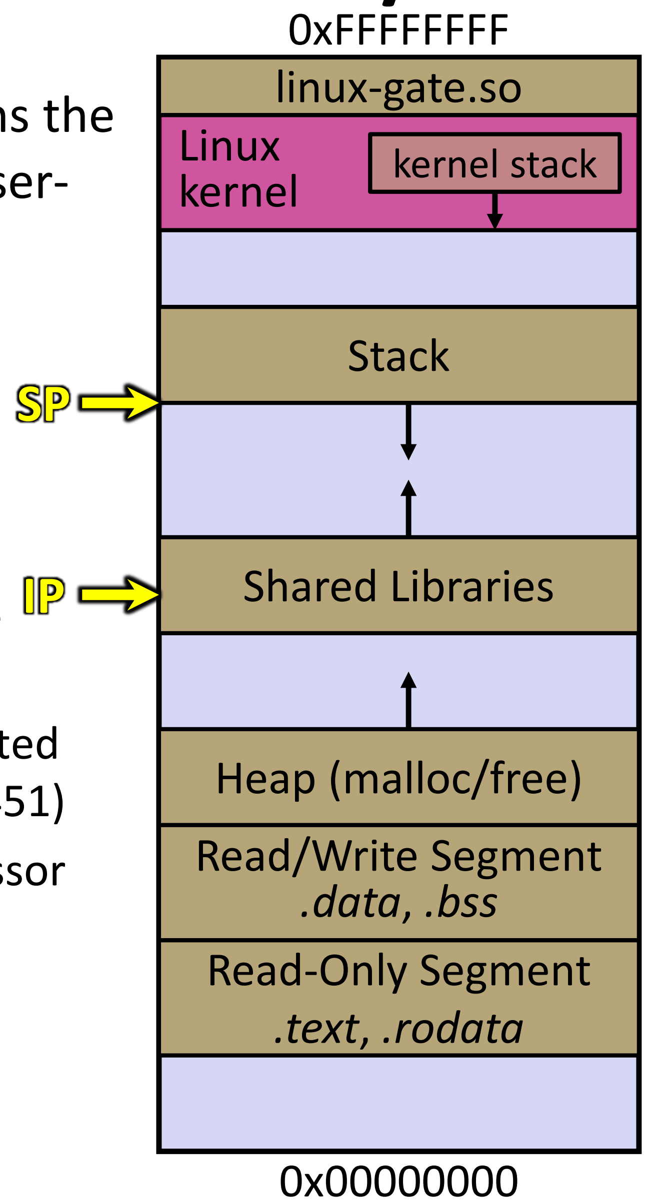


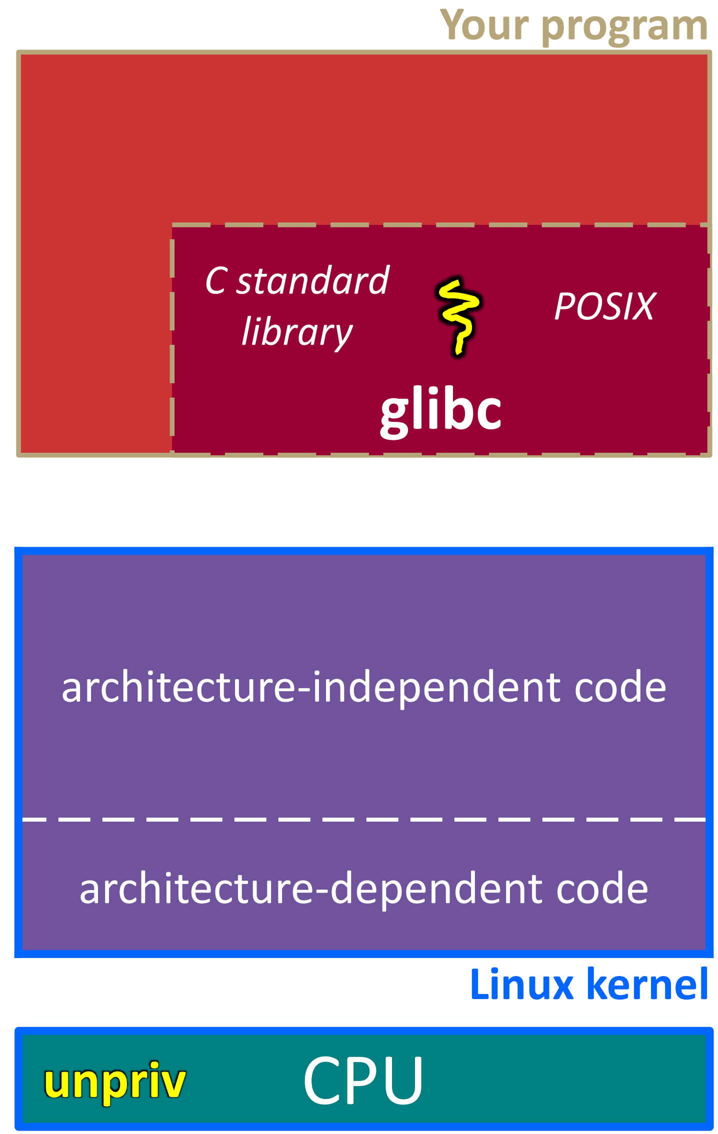


System Calls on x86/Linux

`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`

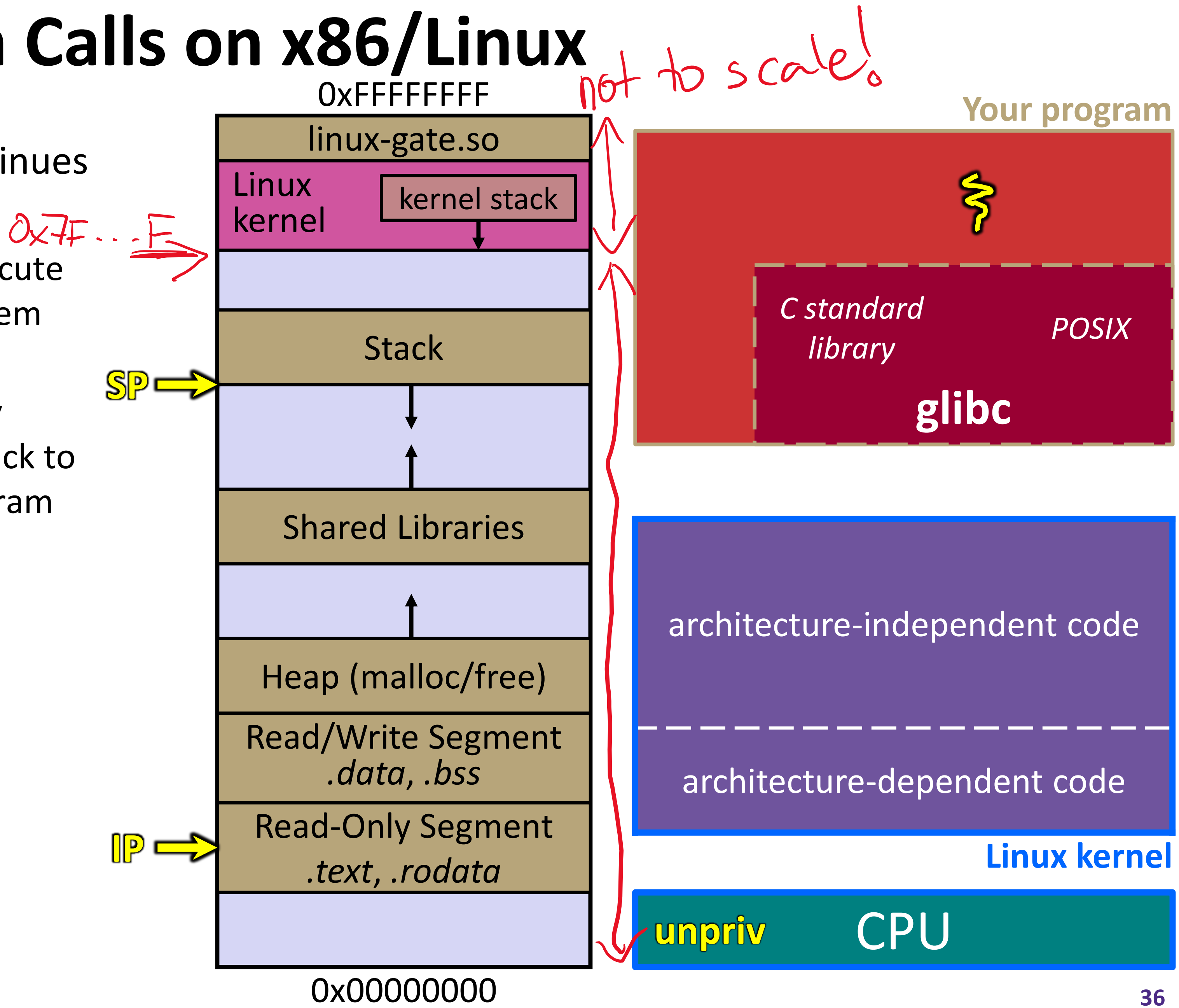




System Calls on x86/Linux

`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

- Another Difference: C Stream Buffering
- Another Difference: What is a System Call?
- **Make**

`make`

- `make` is a classic program for controlling what gets (re)compiled and how
  - Many options (*e.g.* `ant`, `maven`, `bazel`, `gradle`, 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 executables from source code …
  - … that they and other people write

- Programmers like to automate repetitive tasks
  - Repetitive: gcc -Wall -g -std=c11 -o widget foo.c bar.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

1. A single logical step may require lots of actual commands
   - Preprocess, compile, link; generate language bindings (eg, protobuf/thrift)

2. One input may be referenced by multiple outputs
   - *e.g.* Javadoc, .po (for gettext/internationalization)

3. Don't want to document build logic when distributing code

4. Don't want to recompile everything whenever one thing changes
   - Especially if you have 10^5-10^7 files of source code!

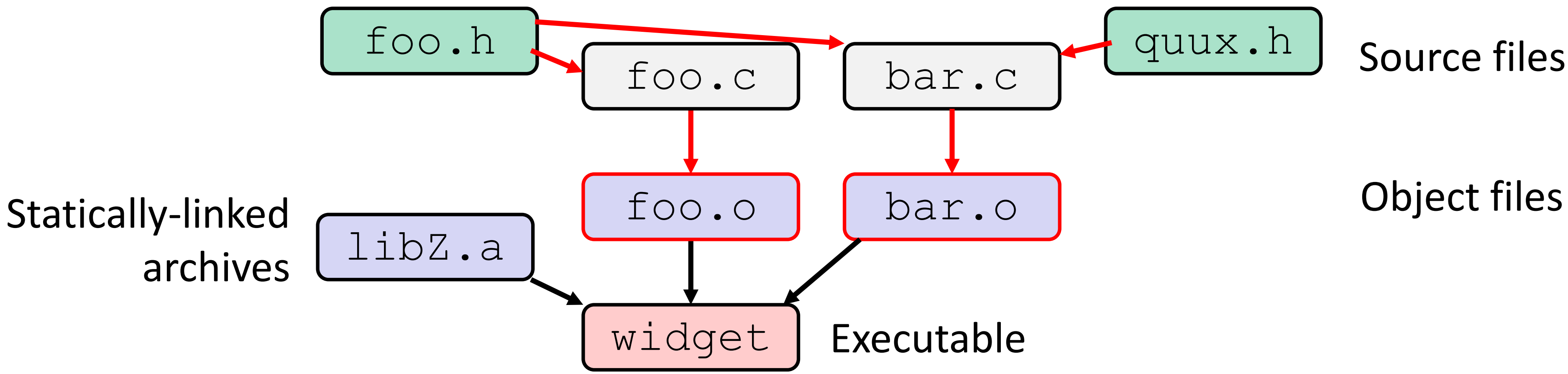
A script can handle #1-3 (use variables for filenames in #2), but #4 is trickier

Recompilation Management

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

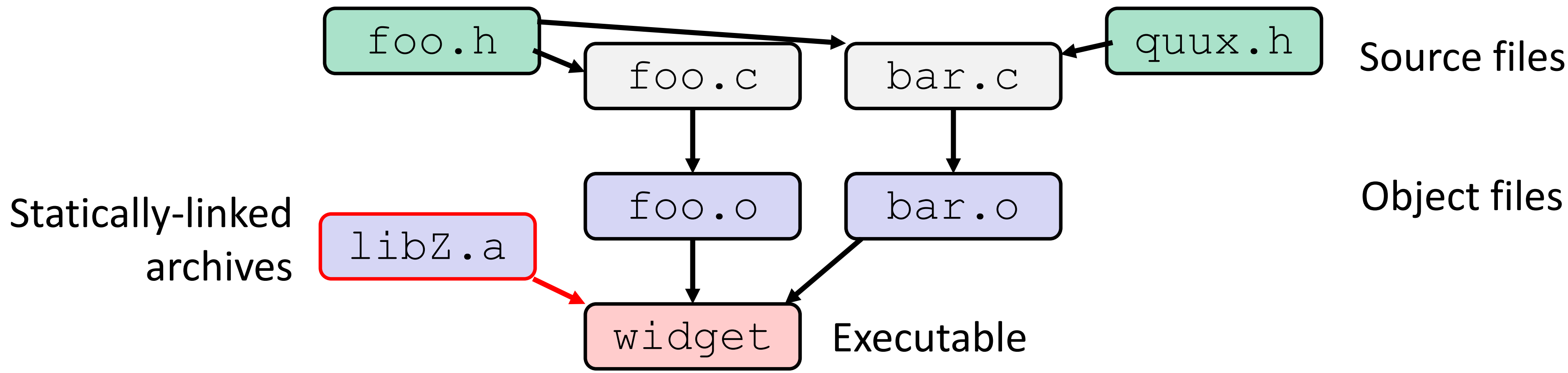
- To create a target t, you need sources $s_1, s_2, \ldots, s_n$ and a command c that directly or indirectly uses the sources
  - It t is newer than every source (file-modification times, content hash, etc), assume there is no reason to rebuild it
  - Recursive building: if some source s_i is itself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

Theory Applied to C



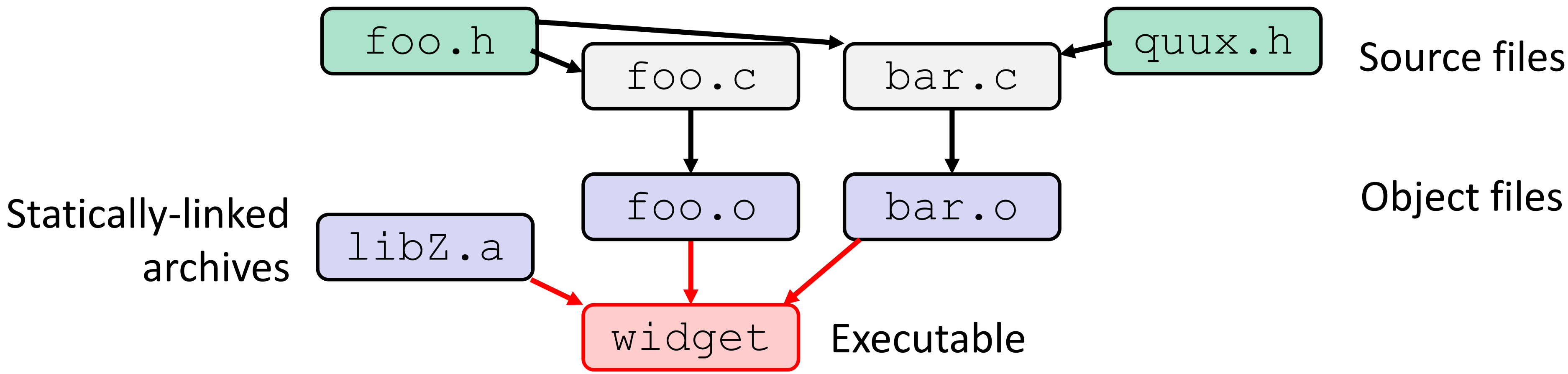
- Compiling a `.c` creates a `.o`
- The `.o` depends on the `.c` and all included files (`.h`'s, recursively/transitively)

Theory Applied to C



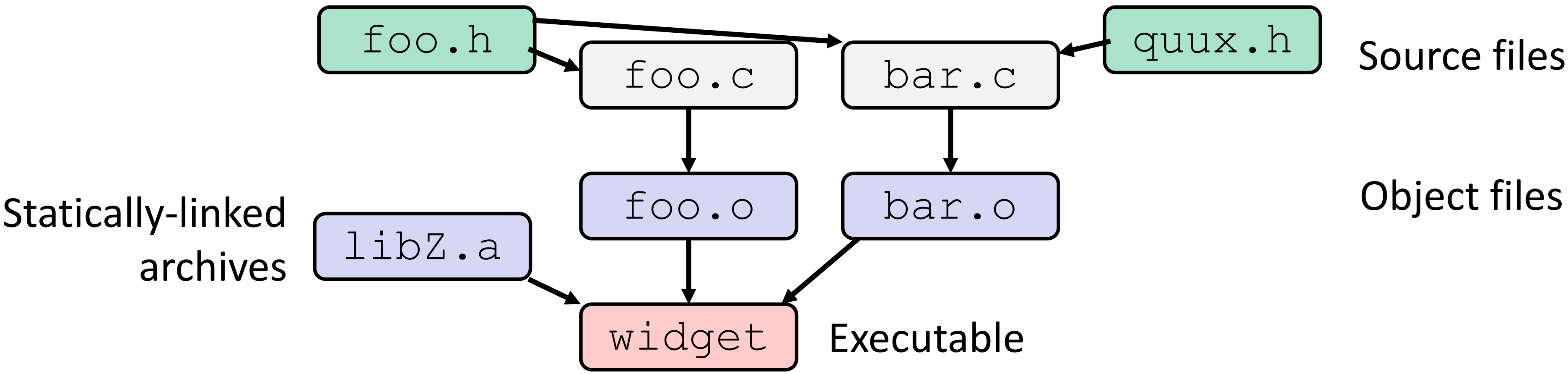
- Compiling a `.c` creates a `.o`
- The `.o` depends on the `.c` and all included files (`.h`'s, recursively/transitively)
- An archive (library, `.a`) depends on included `.o` files

Theory Applied to C



- Compiling a `.c` creates a `.o`
- The `.o` depends on the `.c` and all included files (`.h`'s, recursively/transitively)
- An archive (library, `.a`) depends on included `.o` files
- An executable ("linking") depends on `.o` and `.a` files
  - 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 an archive, and re-link

- If a `.h` file changes, may need to rebuild more

- Many more possibilities!

make Basics

- A makefile contains a bunch of triples:

```
target: sources
← Tab → command
```

  - 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

```
bash% make -f <makefileName> target
```

- Defaults:
  - If no `-f` specified, use a file named `Makefile`
  - 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)

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=c11
foo.o: foo.c foo.h bar.h
	$(CC) $(CFLAGS) -o foo.o -c foo.c
```

- Advantages:
  - Easy to change things (especially in multiple commands)
  - Can also specify on the command line (`CFLAGS=-g`)

More Variables

- It's common to use variables to hold list of filenames:

```
OBJFILES = foo.o bar.o baz.o
widget: $(OBJFILES)
	gcc -o widget $(OBJFILES)
clean:
	rm $(OBJFILES) widget *~
```

- `clean` is a convention
  - Remove generated files to "start over" from just the source
  - It's "funny" because the target doesn't exist and there are no sources, but it works because:
    - The target doesn't exist, so it must be "remade" by running the command
    - These "phony" targets have several uses, such as "`all`"…

"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...
```

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…)

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 `int`
  - Builds an array of the parsed `int`'s
  - Sorts the array
  - Prints the sorted array to `stdout`

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

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

Extra Exercise #2

- Modify the linked list code from Lecture 5 ("Designing C Modules") Extra Exercise #1
  - Add static declarations to any internal functions you implemented in `linkedlist.h`
  - Add a header guard to the header file
  - Write a `Makefile`
    - Use Google to figure out how to add rules to the `Makefile` to produce a library (`liblinkedlist.a`) that contains the linked list code