System Calls & POSIX I/O

CSE 333 Spring 2025

Instructors: Naomi Alterman, Chris Thachuk

Teaching Assistants:

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

Poll Everywhere

pollev.com/naomila

How do you trust a total stranger to care for your dog 🐶 / cat 😽 / bunny 🐰 while you're away on vacation 😎 ?

How do you give them instructions? How do you make sure they do what you asked? What precautions do you take?

(open-ended survey question)

2

Administrivia

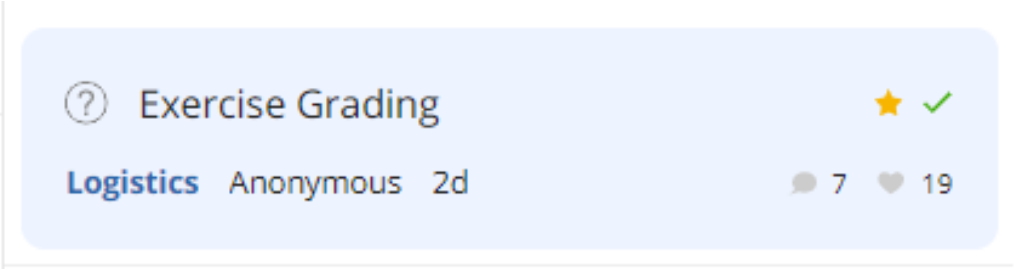
- EX6 due
- EX7 out on Friday, take a breather
  - Involves material from lecture today **and section tomorrow**!
- HW1 due at midnight tomorrow
  - Submission time is the timestamp of the git commit tagged `hw1-final`
  - Late day tokens will get "automatically" used if needed, you don't need to let us know

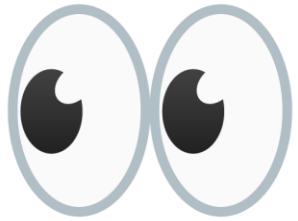
Administrivia

- Midterm!
  - **Monday October 27th** from **5:30p – 6:20p (50 minutes)** in **Smith Hall 120**
    - If you absolutely can't make this time, please let us know in a private Ed post now!
  - Written exam
  - Closed book
  - Allowed one 5x8" card of handwritten notes
  - Will cover material up through (and possibly including) C++ Templates
  - You got this 👊 👊

Administrivia

- Exercise grading and feedback





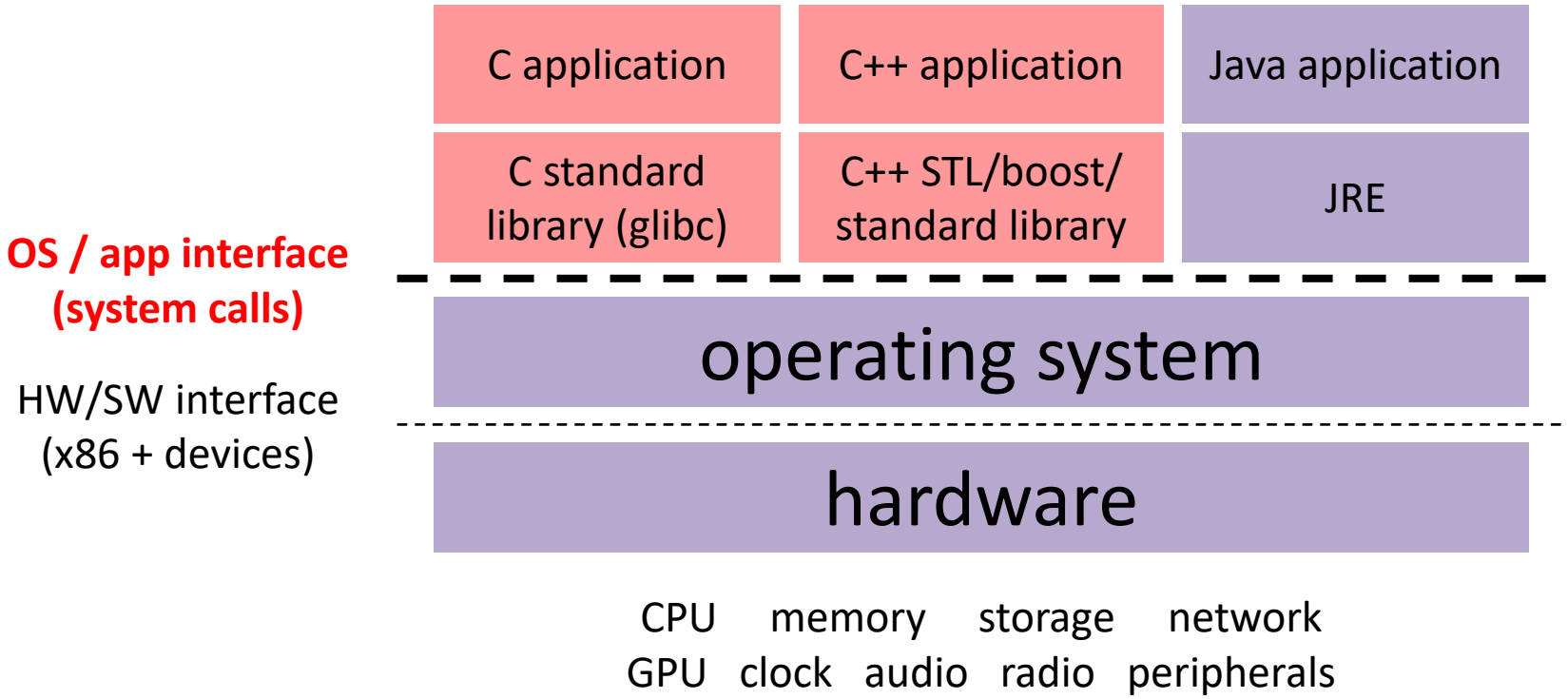
  - Exercise scores do not operate linearly; that is 2/3 != 66.67% and 1/3 != 33.3%.
  - Advay: “The number one thing I can recommend is keeping track of all the feedback you've received on exercises in a document. You can also include feedback from the Gradescope emails, which have common mistakes people made.”

- I apologize for misleading comments earlier in the quarter about how we’d be grading style 🙏 u_u

Lecture Outline

- **System Calls (theoretical)**
- POSIX I/O System Calls

What's an OS?

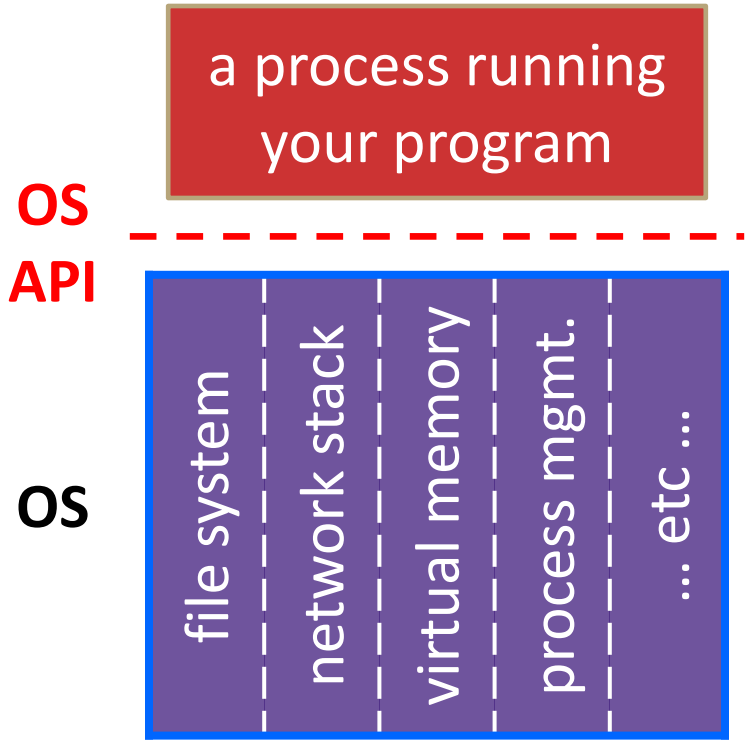

C application
C++ application
Java application
C standard library (glibc)
C++ STL/boost/ standard library
JRE
OS / app interface (system calls)
operating system
HW/SW interface (x86 + devices)
hardware
CPU memory storage network
GPU clock audio radio peripherals

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(), …

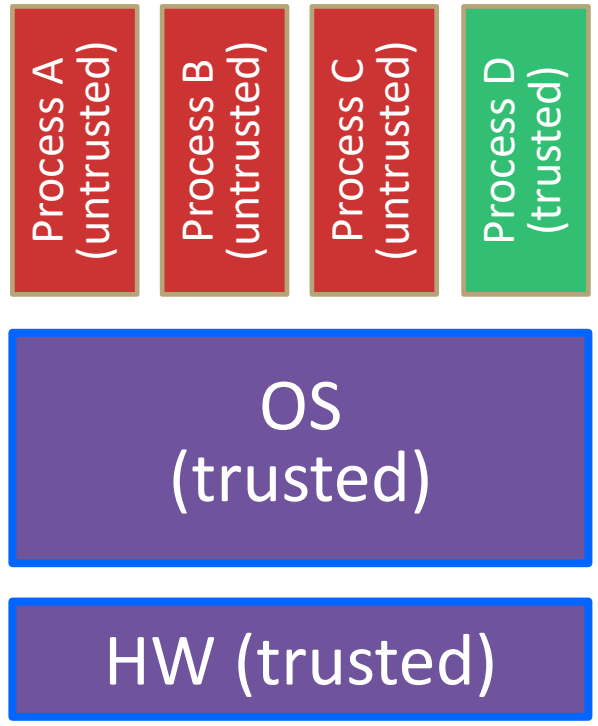
Network Stack
- connect(), listen(), read(), write(), ...

Virtual Memory
- brk(), shm_open(), …

Process Management
- fork(), wait(), nice(), …

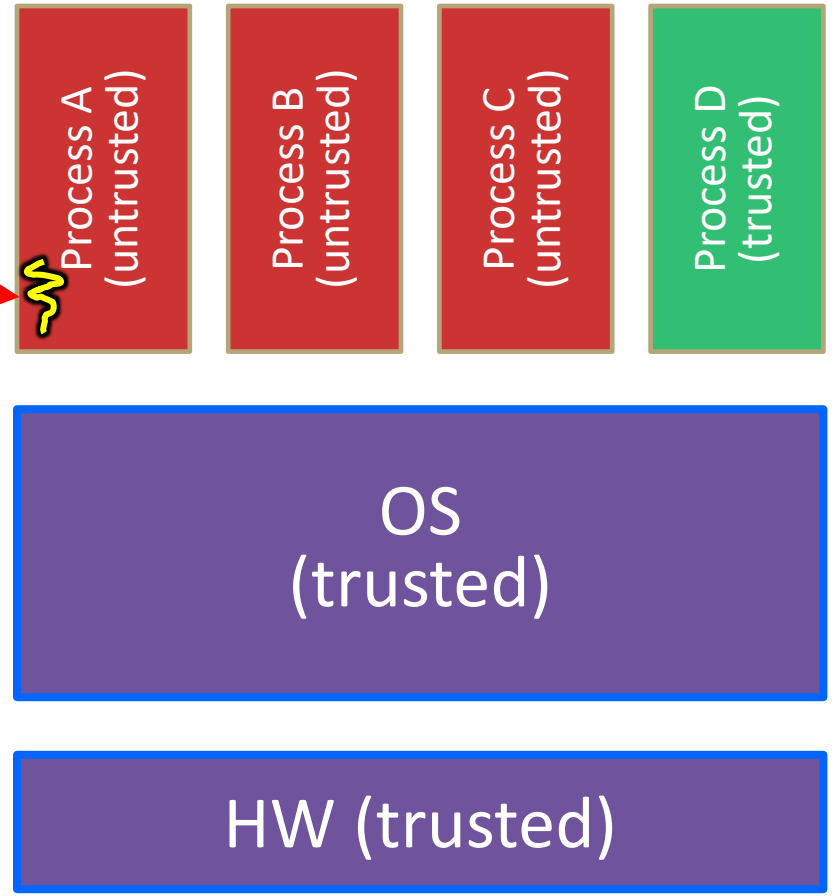
OS: Protection System

- 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



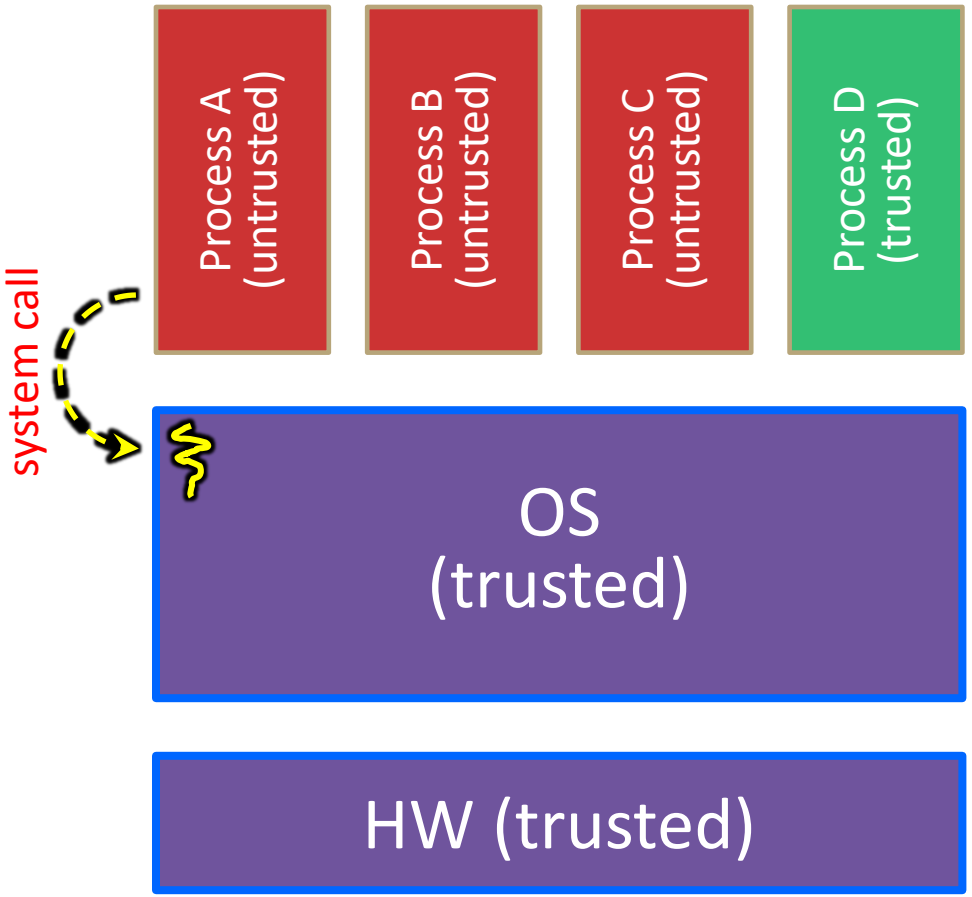
System Call Trace

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



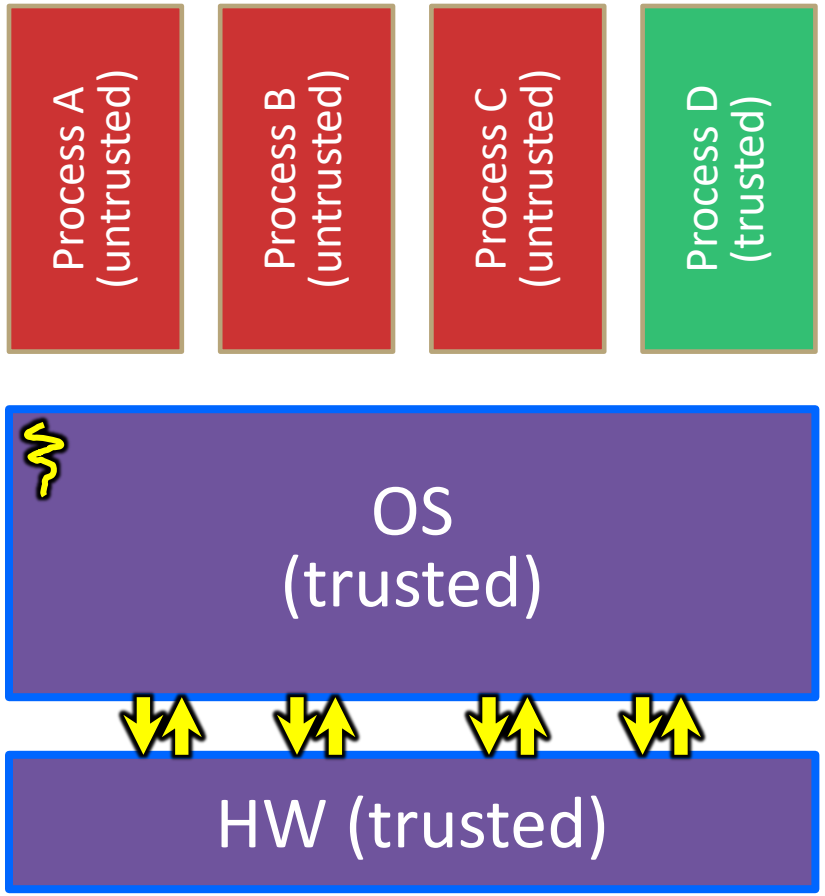
System Call Trace

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.

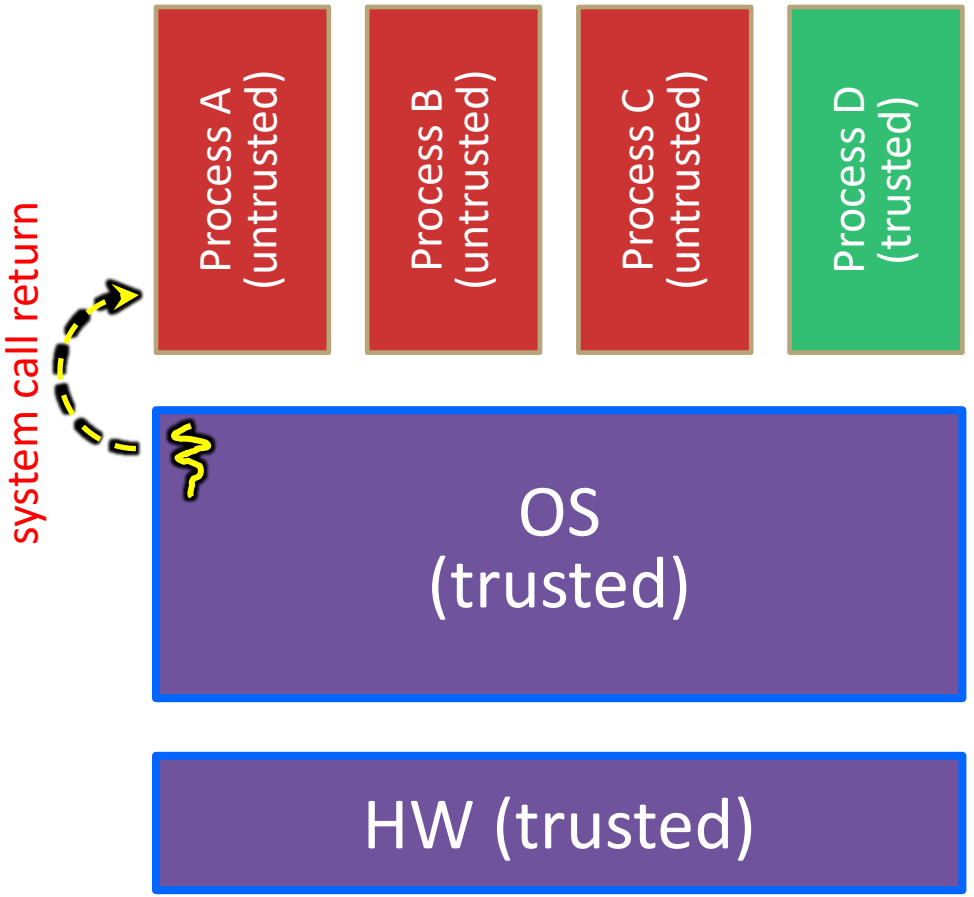


System Call Trace

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

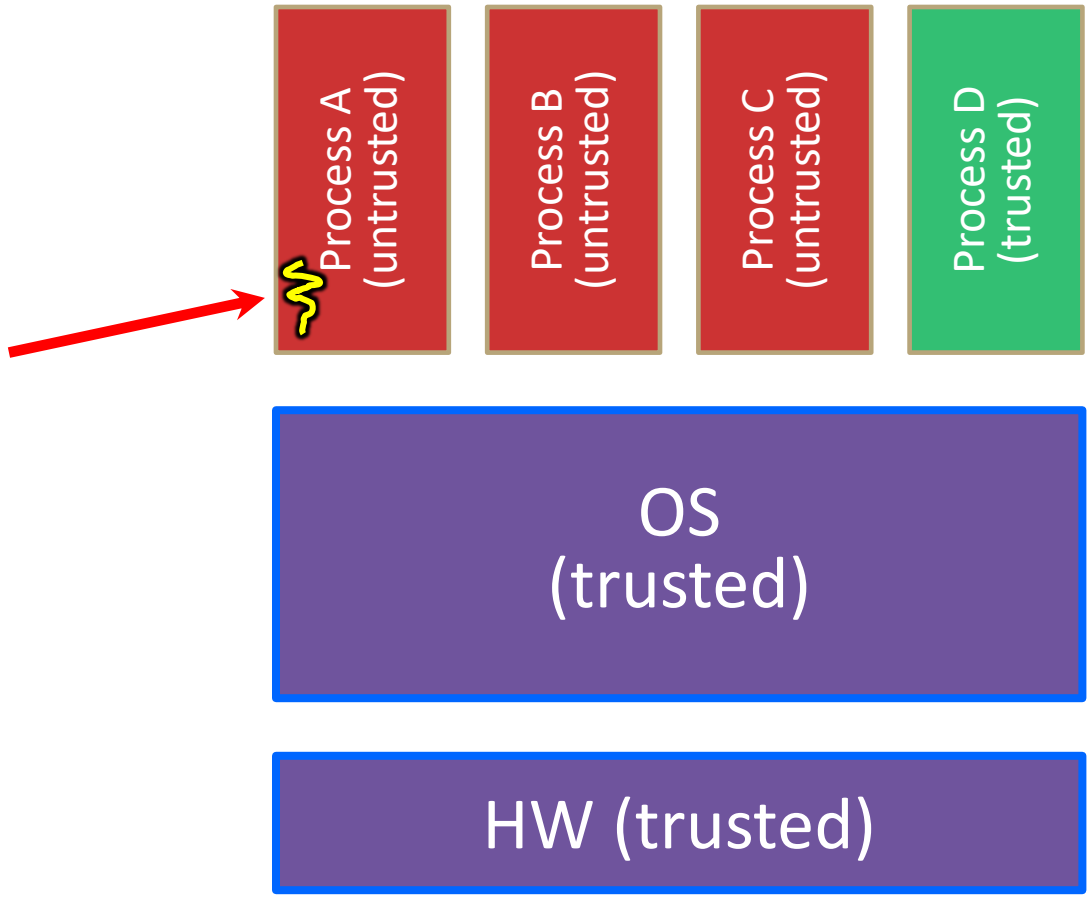
(1) Sets the CPU back to unprivileged mode and

(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)

Lecture Outline

- System Calls (theoretical)
- **POSIX I/O System Calls**

How do you trust a total stranger to care for your dog🐶/ cat 😽/ bunny 🐰 while you're away on vacation 😎 ?

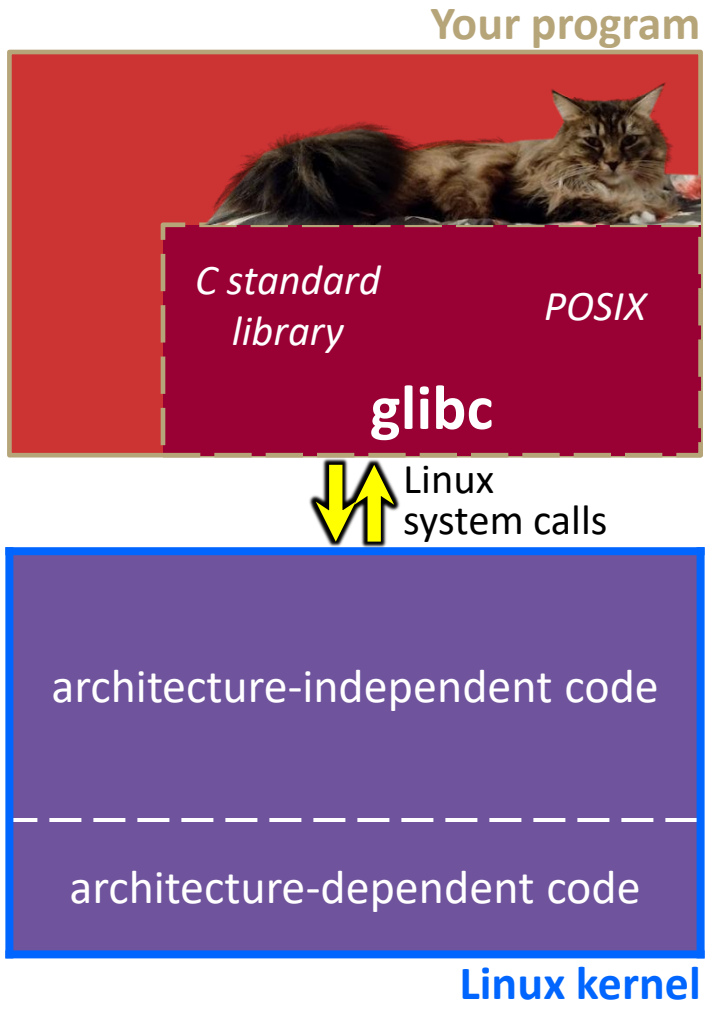
0

Nobody has responded yet.

Hang tight! Responses are coming in.

Remember This Picture?

- Your program can access many layers of APIs:
  - C standard library
    - Some are just ordinary functions (<string.h>, for example)
    - Some also call OS-level (POSIX) functions (<stdio.h>, for example)
  - POSIX compatibility API
    - C-language interface to OS system calls (fork(), read(), etc.)
  - Underlying OS system calls
    - Assembly language ☺



C Standard Library File I/O

- So far you've used 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
  - They are implemented using lower-level OS calls

Lower-Level File Access

- 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 C std lib
    - Lower-level and unbuffered compared to their counterparts
    - Also less convenient
  - We will have to use these to read file system directories and for network I/O, so we might as well learn them now

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`
    - Defaults: **0** is `stdin`, **1** is `stdout`, **2** is `stderr`

```
#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);
```

Reading from a File

- `ssize_t` `ssize_t read(int fd, void* buf, size_t count);`
  - 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
  - **`read`** has some surprising error modes…

Read error modes

- `ssize_t` `ssize_t read(int fd, void* buf, size_t count);`
  - On error, `read` returns -1 and sets the global **`errno`** variable
  - You need to check **`errno`** to see what kind of error happened
    - `EBADF:` bad file descriptor
    - `EFAULT:` output buffer is not a valid address
    - `EINTR:` read was interrupted, please try again (ARGH!!!! 😤 😠)
    - And many others…

One way to `read()` n bytes

- Which is the correct completion of the blank below?
  - Vote at http://PollEv.com/naomila

```
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;
}
```

A. buf

B. buf + bytes_left

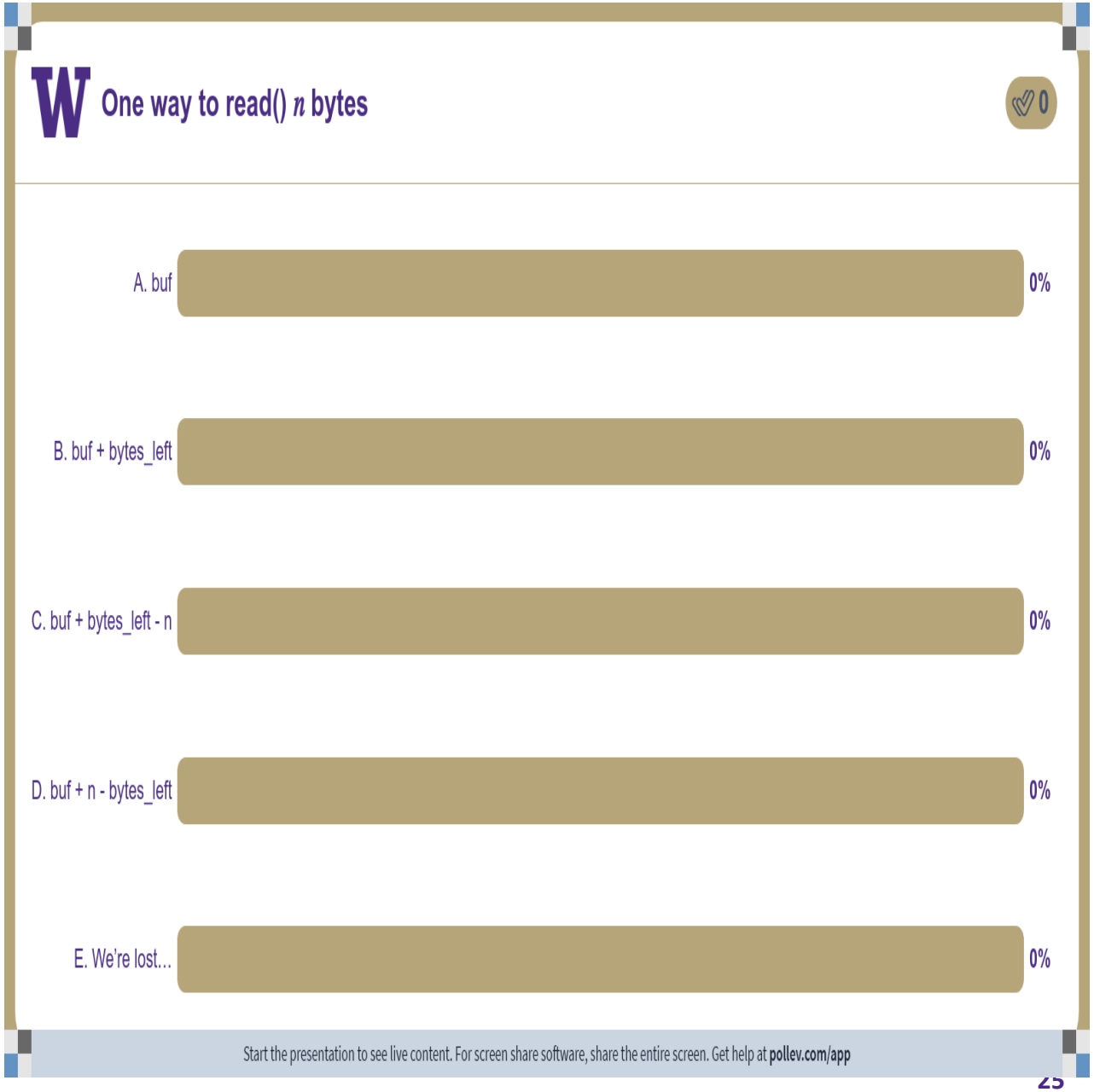
C. buf + bytes_left - n

D. buf + n - bytes_left

E. We're lost…

```
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;
}
```



One way 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);
```

readN.c

Other Low-Level Functions

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

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

- More in sections this week…. (as in, *tomorrow!*)

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`

- 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

- 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