What is an OS?

Operating Systems, Fall 2026

Instructors: Travis McGaha

Head TAs: Shriya Sane

Arnuv Batra
Khoi Dinh
Sydney Fitzgerald
Rania Souissi
Jeffrey Oduman
Abdullah Goher
Monisha Krothapalli
Veer Kakar
Kevin Xue
Shreya Mukunthan
Daniel Price
Wesley Liu
Leo Zhang

pollev.com/tqm

- Which operating system do you think is the most popular?

Administrivia

- Second Assignment (penn-shell)
  - Milestone 0 due Friday 9/11 // you have everything you need for this
  - Milestone 1 due Friday 9/18 // you have everything you need for this
  - Full assignment due Wednesday 9/23 // you have everything you need for this
  - Basic shell implementation
    - System calls, processes, I/O redirection, pipes, job control, and more!
  - Autograders are posted

- Information about interviews will be today or tomorrow!

Lecture Outline

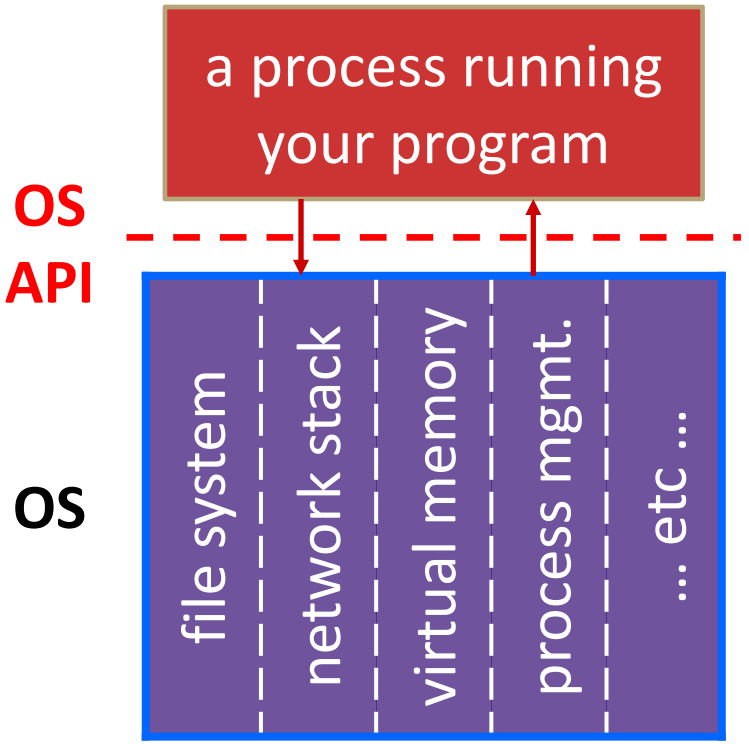
- **Kernel vs Operating System**
- What is Linux?
- System Calls & POSIX
- Buffering

pollev.com/tqm

- What comes to your mind when I say “Operating System”?

OS as we've seen so far: 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.



OS provides access to many services on the computer:

- **File System**
- **Network Stack**
- **Virtual Memory**
- **Process Management**
- **Threads**
- **…**

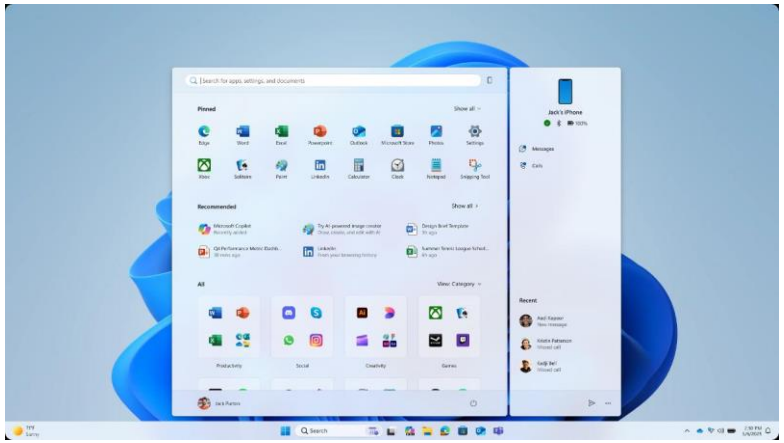
Kernel vs Operating System: What is it to you?

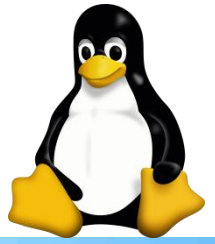
- Sometimes OS and Kernel are used interchangeably, but they mean different things.

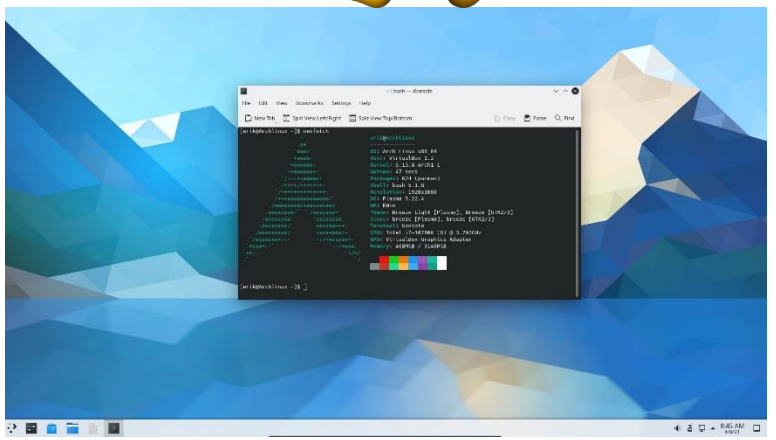
- Outside of this class, when you hear “Operating System” what first comes to mind?

macOS

Windows 11







Kernel vs Operating System

- Sometimes OS and Kernel are used interchangeably, but they mean different things.

- Outside of this class, when you hear “Operating System” what first comes to mind?
  - User interface
  - How files are structured
  - Built-in applications (notepad, safari, candy crush, the terminal, etc)

- You may also associate an operating system with what hardware (laptops) you can run those OS’s on.

What is the kernel?

- An operating system kernel is the “core” of the operating system that handles various core components of the system

- It manages all hardware resources
  - I/O: file system & network
  - Arbitrates conflicts between processes on these resources
  - Thread/processes creation & scheduling
  - Virtual memory
  - …

- When you start your computer, the kernel* is (typically) the first* thing run and it is responsible for setting up the rest of the OS

What is the kernel?

- The programs we write (for the most part) are “user-level” and usually only have basic permissions. The OS has enhanced permissions.

- The kernel is Software that runs at elevated permissions
  - It is allowed to directly interact with the hardware
  - It is allowed to access memory of any process
  - It isn’t restricted on which instructions it can run like the user is
  - In general, it is trusted to do anything that can be done on that architecture (hardware)

Lecture Outline

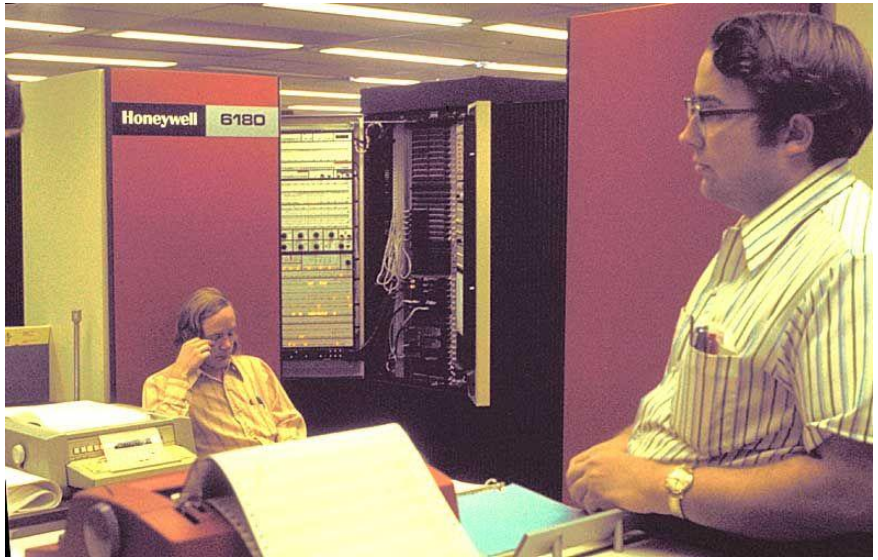
- Kernel vs Operating System
- **What is Linux?**
- System Calls & POSIX
- Buffering

Multics: The Precursor

- **Multiplexed Information and Computing Service**

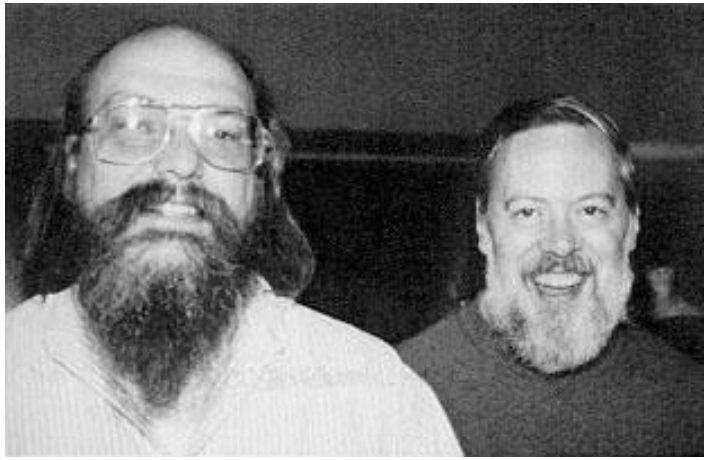
- Early time-sharing operating system
  - Time sharing: the sharing of a computer (mainframe) across multiple users at the same time
  - Necessary pre – personal computers (~1975)

- Started development in 1964
  - funded in part by Bell labs
- Bell Labs pulls out of Multics in 1969



"Unics"

- Ken Thompson and Dennis Ritchie lead the development of Unix
  - Both worked on Multics under Bell Labs



- Took some inspiration from Multics
  - Hierarchical file system
  - Text command line shell
  - The name:
    - Multics: Multiplexed Information and Computing Service
    - Unics: Uniplexed Information and Computing Service
    - At some point "Unics" became "Unix“
  - Unix rejected the overcomplexity of Multics

UNIX

- Originally (1970) was a singletasking system, without name or backing, and written in PDP assembly

- Functionality and multitasking added as other departments in Bell Labs needed them

- Departments kept adopting UNIX instead of built in OS's.
  - As a result, a support team was created, a UNIX Programmer's Manual was written, and man pages were created

UNIX and C

- B programming language by Ken Thompson
  - Was intended for writing UNIX utilities
- Dennis Ritchie modified B to make New B
  - Added things like types! (int, char, etc.)
- More features were added to New B, heavily influenced by its use in UNIX
- UNIX was soon re-written in C
  - One of the first operating systems (re)written in a higher-level-language (aka, not assembly)

Unix Adoption

- 1973: Unix was first presented formally outside of Bell Labs. Leading to many requests for the system

- Due to a 1956 decree, Bell System could not turn UNIX into a commercial product.
  - Bell had to license the product to anyone who asked
  - Code was “open source” of sorts.

- UNIX was continually updated, and C was as well.
  - Included the addition of pipes and other features
  - These updates made UNIX more portable to other systems.

UNIX Design Philosophy

- Philosophy behind development of UNIX that spread to standards for developing software generally.
  - Arguable more influential than UNIX itself

- Short version:
  - Programs should "Do One Thing And Do It Well."
  - Programs should be written to work together
  - Write programs that handle text streams, since text streams is a universal* interface.

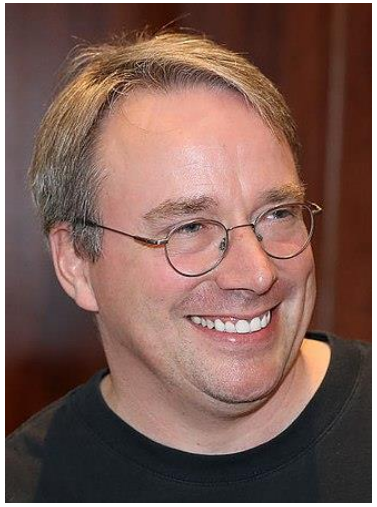
- Extra short version: "Keep it Simple, Stupid."

GNU

- In 1983, Bell Systems split up due to anti-trust laws.
  - A successor (AT&T) then turned UNIX into a commercial product, limiting rights to distribute/change/adapt/etc. UNIX

- Later that year, GNU is founded by Richard Stallman
  - GNU Not Unix
  - Copyleft
  - Goal: create a complete UNIX compatible system composed entirely of free software
  - Developed many required programs (libraries, editors, shell, compilers …) but missing low level elements like the kernel

Linux



- By 1991, a UNIX-like kernel that was
  Free Software did not exist

- Linus Torvalds was studying operating systems and wrote his own called Linux
  - This would be published under GPL 2 (GNU Public License)

- Blew up in popularity due to being free and open source

Unix-Like

- Almost all operating systems are UNIX related
  - "Genetically" related with historical connection to the original code base
  - Through the UNIX trademark once a system meets the Single UNIX Specification and is certified
  - Through "functionally" being UNIX-like. Behaving in a manner that is consistent with UNIX design and specification
    - Linux falls under this one
- Most Operating systems are Unix Like
  - Linux, macOS, iOS, Chrome OS, Android, etc.
  - Pretty much everything that is not Windows lol

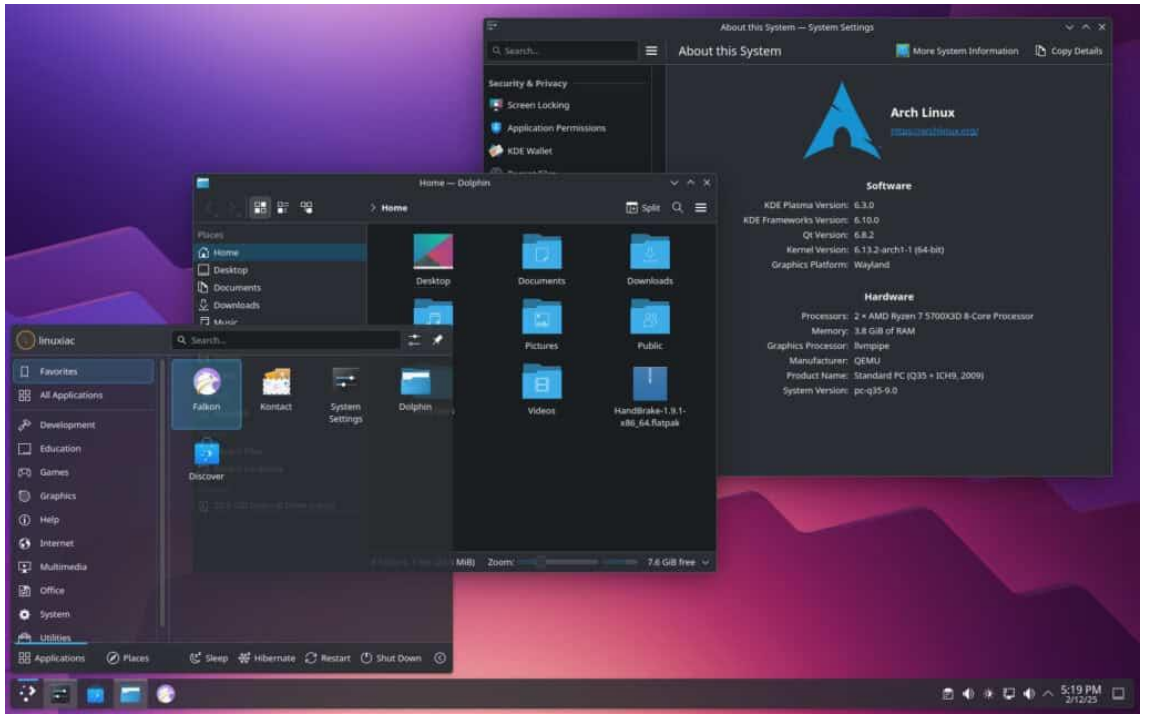
Linux Distributions

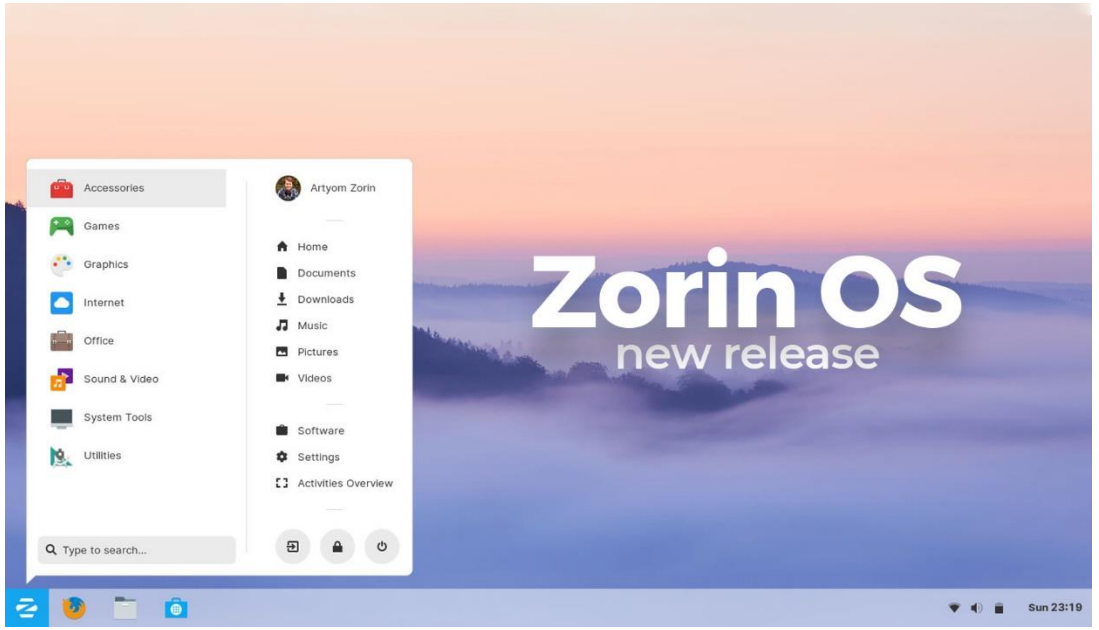
- When someone has a computer that is “Linux” there are many different variants called “Distributions” or shortened to “distros”

- Each Linux distribution includes the Linux kernel at its core, but have different choices for the built-in:
  - Package manager
  - Init system (system to dynamically start/stop/enable/disable services)
  - Network configuration
  - Display system (manages how application windows are presented on your screen)
  - Desktop environment
  - Default installed applications & utilities

Linux Distro Examples

Ubuntu





Choosing a Linux Distribution

- There is an online quiz to help you decide which distro to use if you are unsure: https://distrochooser.de/
  - This is how I decided to use arch when I got sick of windows over 2025 winter break

- For beginners, I would recommend either:
  - ZorinOS
  - Linux Mint
- These ones are usually pretty beginner friendly

Linux Basics

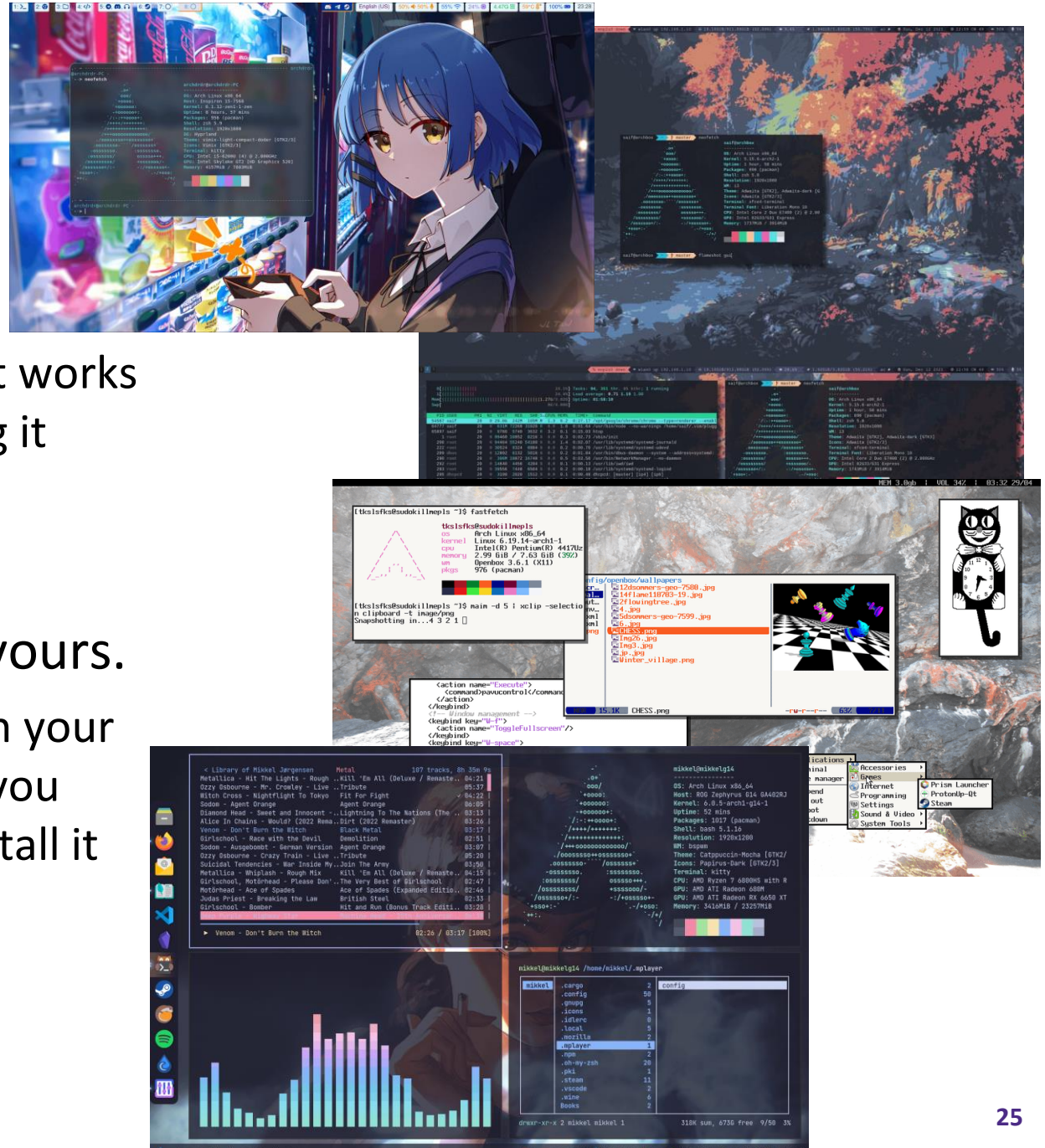
- By doing things in this class, you already should have some familiarity with linux basics

- The file system structure and the hierarchical file system
  - . and ..
  - overall root /
  - Etc.

- The terminal basics
  - cd, mkdir, ls, etc.

Why Linux?

- Linux is free & open-source software.
  - Being free is pretty nice ☺
  - There is a very passionate community that works to keep making linux better and improving it
- Linux distros aren't controlled by companies. Your OS & device is truly yours.
  - You don't have to worry about spyware on your computer or any dumb shit (copilot) that you don't want. If you don't want it, just uninstall it
- Its also very customizable (aesthetically and functionally)



Curious About Linux?

- A youtuber (diinki) has made some videos on Linux and switching to Linux that I think are pretty good. I recommend them if you want to know more.

- The Art of using Linux and why YOU should too
  https://www.youtube.com/watch?v=z4iSZetVkRg

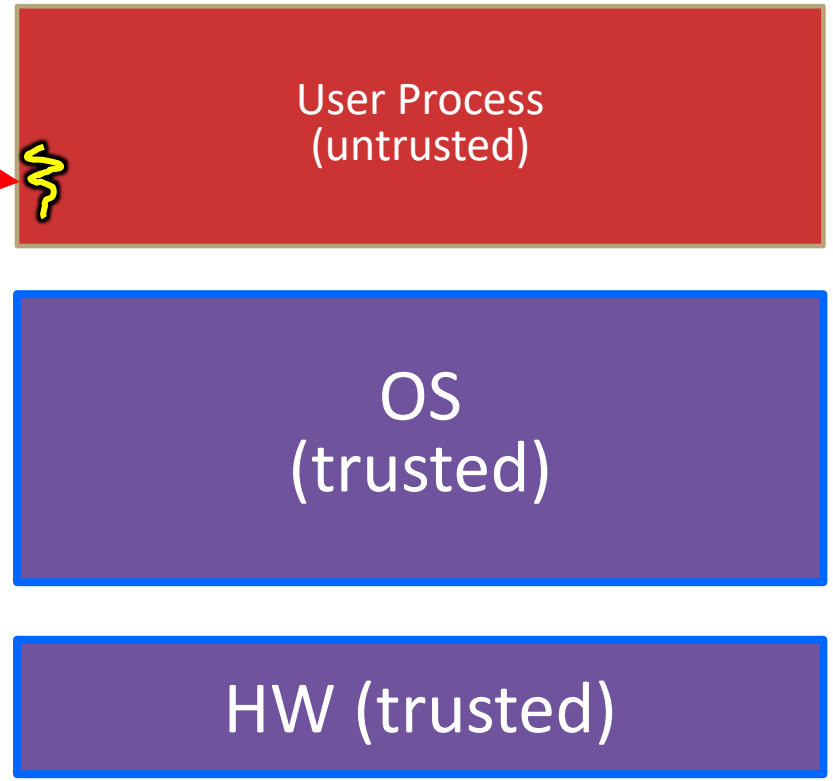
- The Perfect Linux Beginners Guide
  https://www.youtube.com/watch?v=uXvT7hHJfIE

Operating System

- Kernel vs Operating System
- What is Linux?
- **System Calls & POSIX**
- Buffering

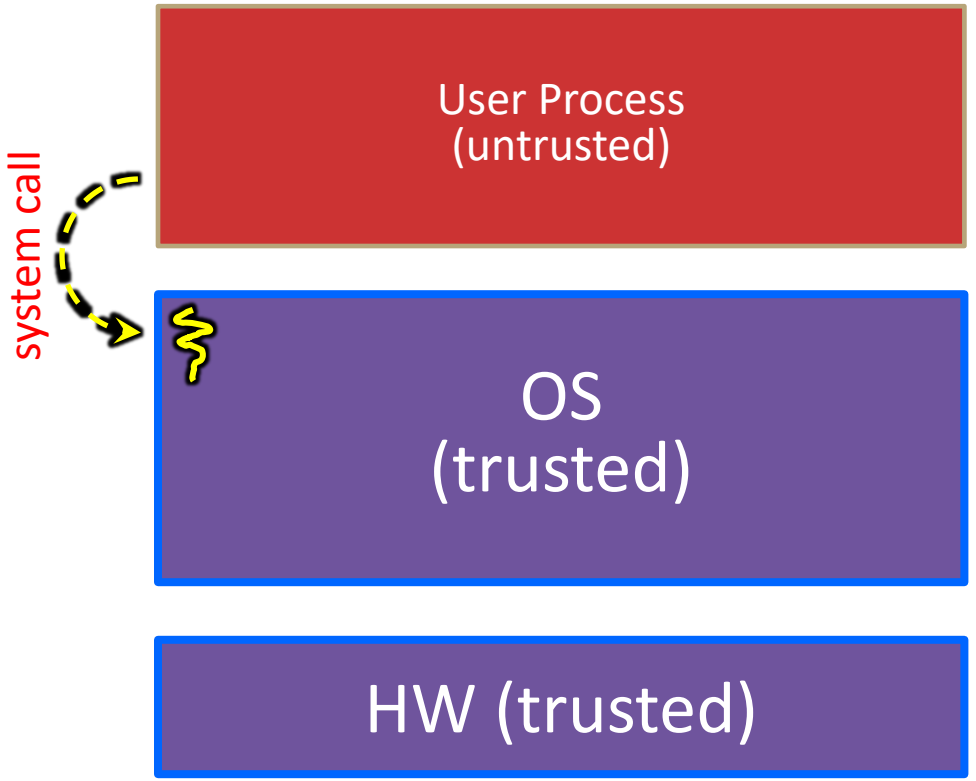
System Call Trace (high-level view)

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



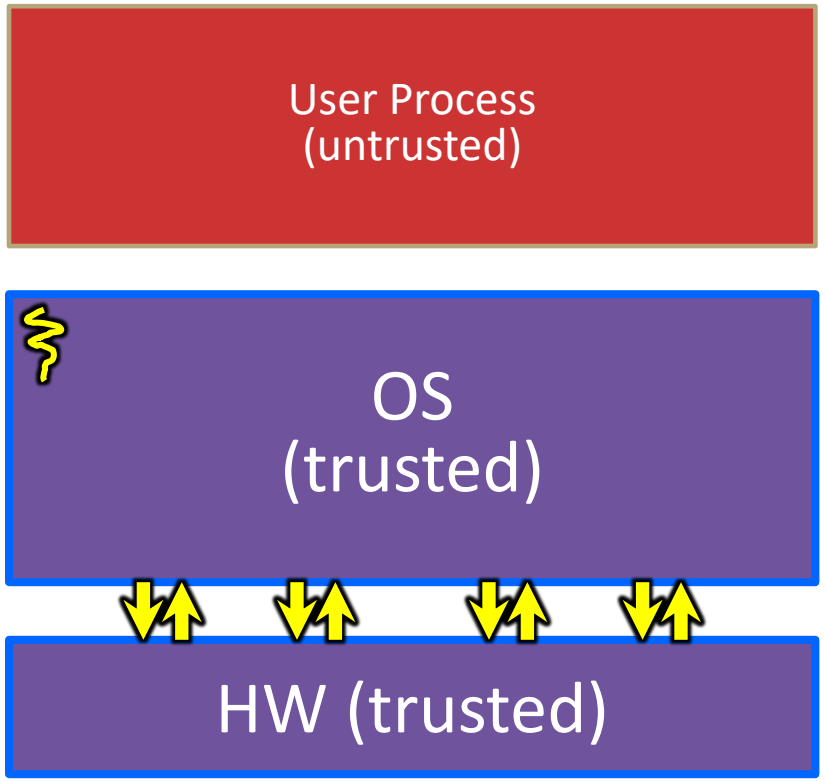
System Call Trace (high-level view)

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



System Call Trace (high-level view)

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

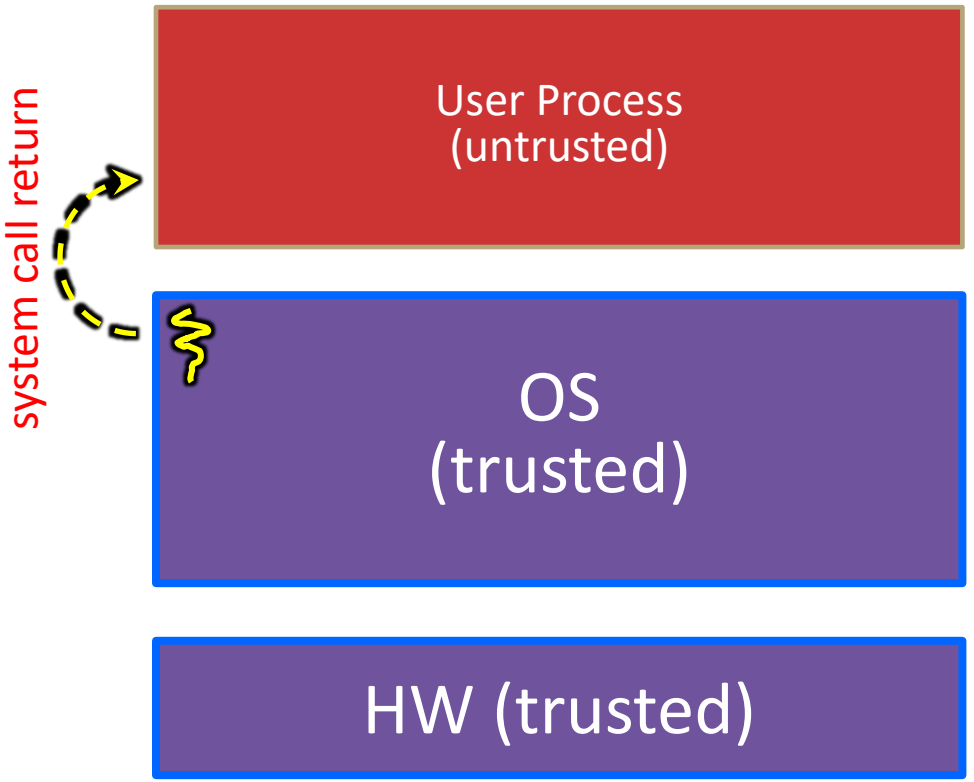


System Call Trace (high-level view)

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

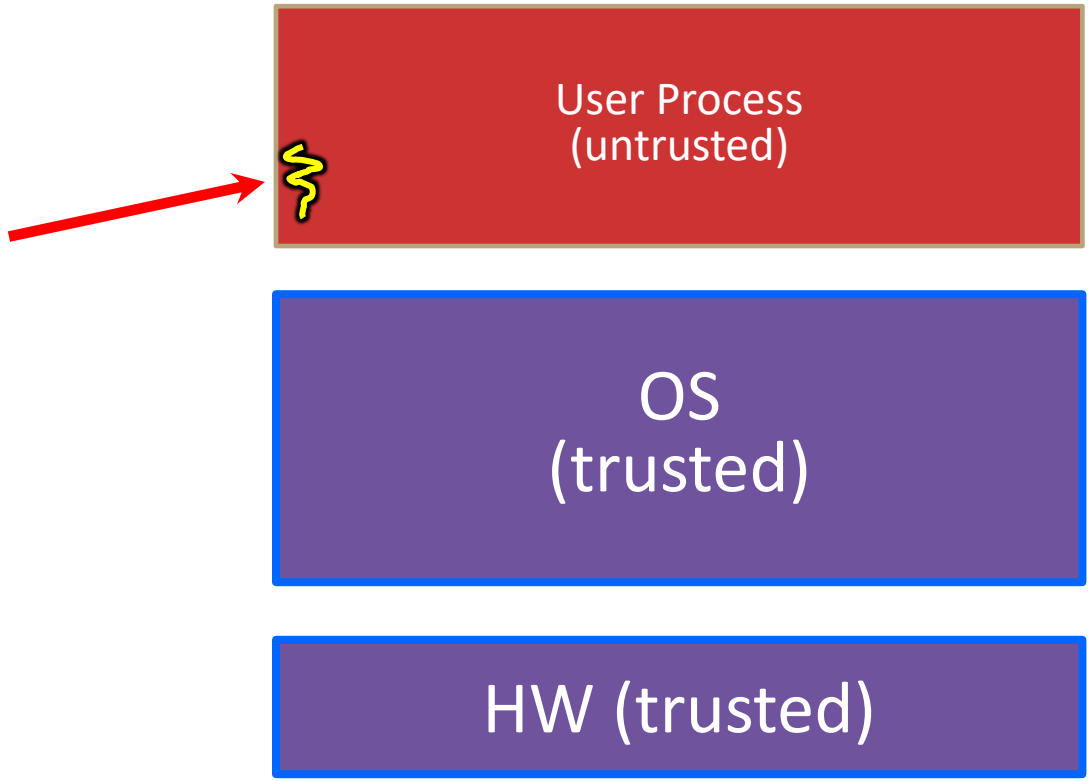
(1) Sets the CPU back to unprivileged mode and

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



System Call Trace (high-level view)

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



A System Call Analogy

- The OS is a very wise and knowledgeable wizard
  - It has many dangerous and powerful artifacts, but it doesn't trust others to use them. Will perform tasks on request.
- If a civilian wants to access a "magical" feature, they must fill out a request to the wizard.
  - It takes some time for the wizard to start processing the request, they must ensure they do everything safely
  - The wizard will handle the powerful artifacts themselves. The user WILL NOT TOUCH ANYTHING.
  - Wizard will take a second to analyze results and put away artifacts before giving results back to the user.

What are system calls really?

- System calls are how we invoke the functionality of the operating system.
  - Boils down to assembly
  - We put a special number in a register to specify what "function" we want the OS to run.
  - Put arguments in correct registers
  - Then run a special instruction similar to calling a function, except transfer is controlled to the OS.
    - X86/64 – `syscall`
    - ARM – `svc #0`
    - riscv – `ecall`
    - LC4 – `TRAP`
- When the OS runs, it runs in a special "protected/higher-privilege" mode. It takes time for these abstractions to work.

How are they implemented?

- Well, we can always invoke our system call via in-line assembly!
  - Yes, this is legal C and C++, though your machine may use a different ASM ISA

```
int main() {
  int fd = -1;
  int mode = O_RDONLY;
  int flags = 0;
  const char* fname = "example.txt";

  __asm__("movl $2, %%eax\n"
          "movq %1, %%rdi\n"
          "movl %2, %%esi\n"
          "movl %3, %%edx\n"
          "syscall\n"
          "movl %%eax, %0"
          : "=r" (fd)
          : "r"(fname), "r"(flags), "r"(mode)
          : "rax", "rbx", "rdi", "rsi", "rdx");
```



```
        movl    $-1, -16(%rbp)
        movl    $0, -20(%rbp)
        movl    $0, -24(%rbp)
        leaq    .L.str(%rip), %rax
        movq    %rax, -32(%rbp)
        movq    -32(%rbp), %rcx
        movl    -24(%rbp), %r8d
        movl    -20(%rbp), %r9d
        #APP
        movl    $2, %eax
        movq    %rcx, %rdi
        movl    %r8d, %esi
        movl    %r9d, %edx
        syscall
        movl    %eax, %ecx
        #NO_APP
```

System Call Portability

- There are many different assembly architectures. Whenever someone wants to interact with the OS, do they need to write asm specific to that architecture?

- In the past: yes!
  - In the past (1970s and prior) ASM was how you programmed many things. C was considered a “high level” language.

- In the present: no!
  - You can use **`long syscall(long number, …)`** and it will handle the ASM.
  - To help make an OS more portable, there is a nice C level interface available to user level programs that serves as a very thin wrapper around the assembly: POSIX API

From C to POSIX

- Most UNIX-en support a common set of lower-level file access APIs: POSIX – **P**ortable **O**perating **S**ystem **I**nterface (for Uni**x**)
  - `open()`, `read()`, `write()`, `close()`, `lseek()`
    - Similar in spirit to their `f*()` counterparts from the C std lib
    - Lower-level and unbuffered compared to their C and C++ std lib counterparts
    - Also less convenient
    - Better than using assembly!
  - C and C++ stdlib doesn't provide everything POSIX does
    - You will have to use these or a third-party library for network I/O

C Standard Library I/O

- We've seen the C standard library to access files
  - Uses a *stream* abstraction
  - `fstream`, `ifstream`, `ofstream`, `getline()`, etc.

- These are convenient and portable
  - They are buffered*
  - They are implemented using lower-level OS calls

ALL FILE I/O IS BUILT ON TOP OF LOWER-LEVEL OS CALLS

open()/close()

- To open a file:
  - Pass in the filename and access mode
  - Get back a "file descriptor"
    - Similar to `FILE*` from `fopen()`, but is just an `int`
      - Returns `-1` to indicate error
    - Must manually close file when done

Used to identify a file w/ the OS

```
#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

Stores read result in buf

Number of bytes

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

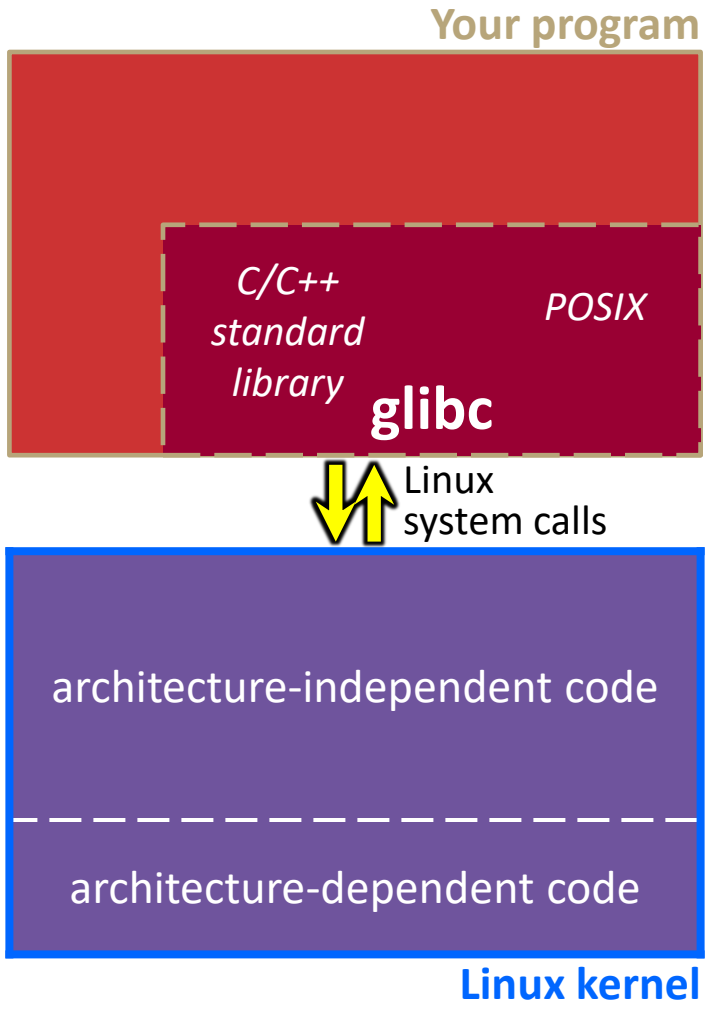
signed

- Function is written in C: follows C design
  - Takes in a file descriptor
  - Takes in an array and length of where to store the results of the read
  - Returns number of bytes read* (see next slide)

- **EVERY TIME we read from a file, this function is getting called somewhere**
  - Even in Java or Python
  - There are wrappers around this, but they are all implemented on top of these system calls
  - The OS doesn't speak java or python, it "speaks" assembly and C so all languages must have a way to invoke these C functions.

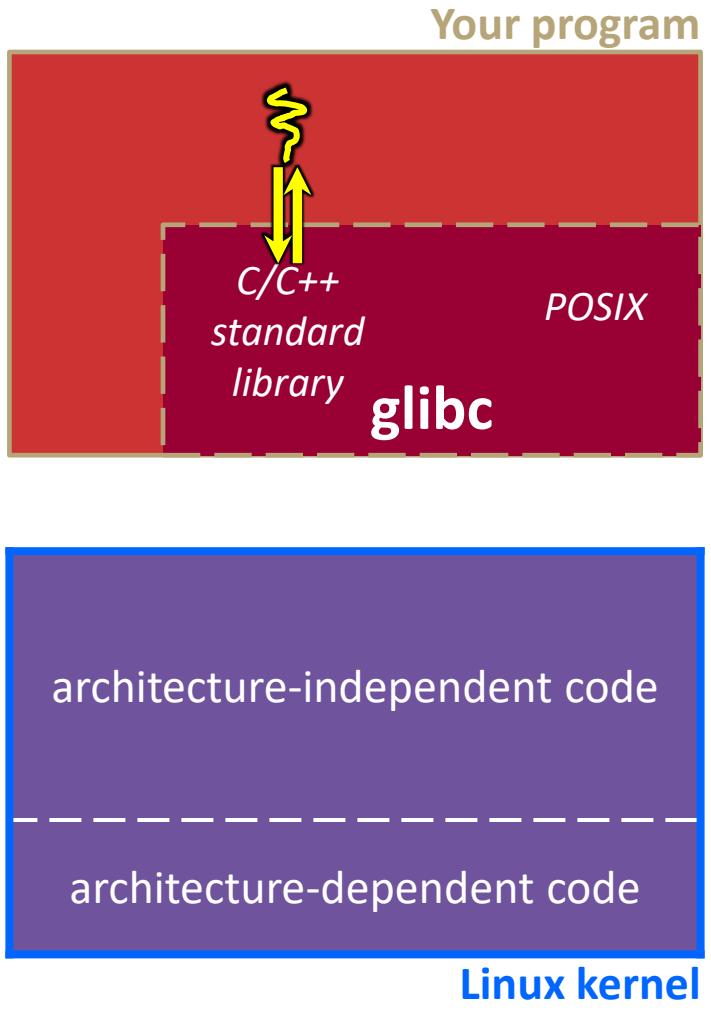
"Library calls" on x86/Linux

- A more accurate picture of what happens when we invoke a function
- The C standard library and POSIX are written as "user-level" code.
- The OS has both code that depends on which architecture we are on, and codes that is independent of that.



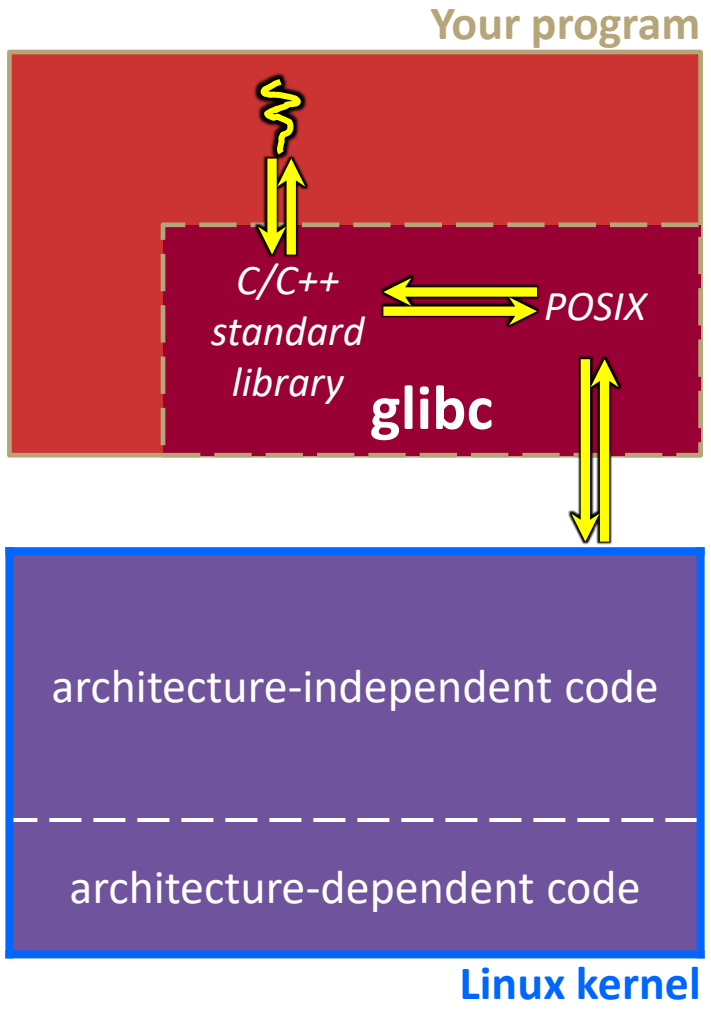
"Library calls" on x86/Linux: Option 1

- Some routines your program invokes may be entirely handled by `glibc` without involving the kernel
  - *e.g.* **`strcmp`**`()` from `stdio.h`



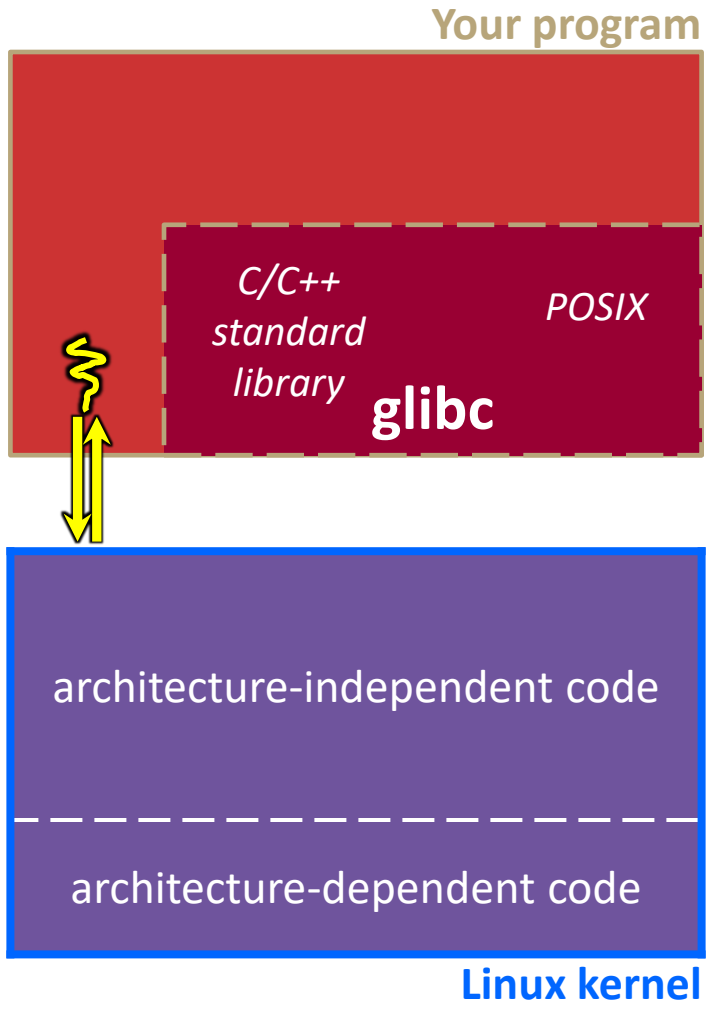
"Library calls" on x86/Linux: Option 2

- 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`**`()` will invoke POSIX, which invokes underlying Linux **`open`**`()`, **`close`**`()`, **`write`**`()`, etc.



"Library calls" on x86/Linux: Option 3

- We can also do the gross "direct" system call via assembly.
- Has its uses, but you don't want to do this 99.999999% of the time



Other Low-Level Functions

- Read man pages to learn about:
  - **`write`**`()` – write data
    - `#include <unistd.h>`
  - **`lseek`**`()` – reposition and/or get file offset
    - `#include <unistd.h>`
  - **`opendir`**`()`, **`readdir`**`()`, **`closedir`**`()` – deal with directory listings
    - Make sure you read the section 3 version (*e.g.* `man 3 opendir`)
    - `#include <dirent.h>`

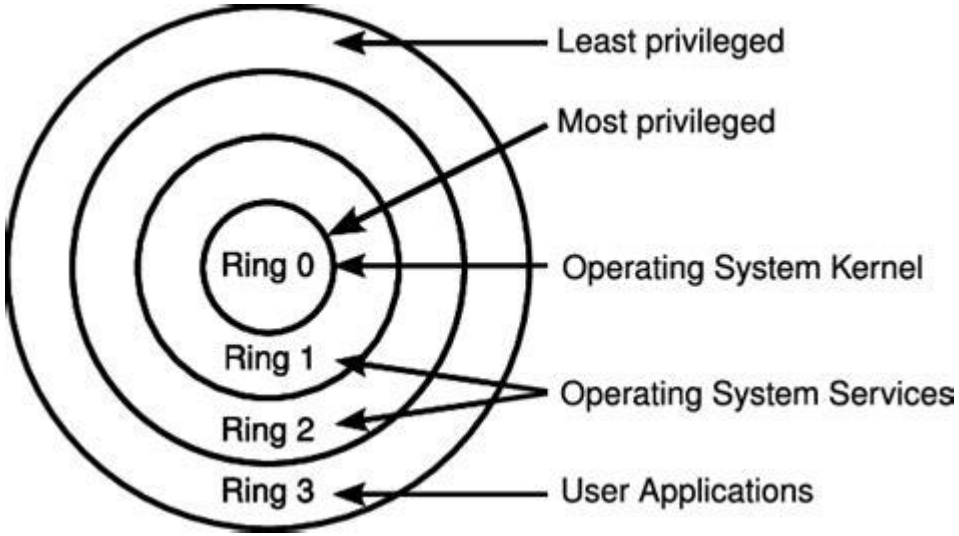
- A useful shortcut sheet (from CMU):
  http://www.cs.cmu.edu/~guna/15-123S11/Lectures/Lecture24.pdf

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)

Aside: How do we "switch" into the operating system

- Like calling a function, a return address and some registers are stored on the caller's stack.
- An instruction is executed to interrupt normal control flow, and switch to the kernel
- The process then changes privilege, the stack pointer is relocated to a stack in the kernel, and some information needs to be cleared (e.g. TLB which we will talk about later)



Aside: How do we "switch" into the operating system

- When in the operating system, the user args need to be checked to make sure they are legal and then the specified system call can be run
- Invoking a system call ends up taking more time than calling a normal function due to the isolation that must be enforced.
- Various mechanisms exist to try to reduce system call cost.
  - Look into VDSO if you want to learn more

Lecture Outline

- Kernel vs Operating System
- What is Linux?
- System Calls & POSIX
- **Buffering**

Poll Everywhere

pollev.com/tqm

- If we compile this and run it, how many times is hello printed?

```
int main() {
  if (fork() == 0) {
    write(STDOUT_FILENO, "hello", 5);
  }
  if (fork() == 0) {
    write(STDOUT_FILENO, "hello", 5);
  }
  return EXIT_SUCCESS;
}
```

Poll Everywhere

Raise Your Hands

- If we compile this and run it, how many times is hello printed?

```
int main() {
  if (fork() == 0) {
    printf("hello");
  }
  if (fork() == 0) {
    printf("hello");
  }
  return EXIT_SUCCESS;
}
```

Poll Everywhere

Raise Your Hands

- If we compile this and run it, how many times is hello printed?

```
int main() {
  if (fork() == 0) {
    printf("hello\n");
  }
  if (fork() == 0) {
    printf("hello\n");
  }
  return EXIT_SUCCESS;
}
```

C stdio vs POSIX

- Why are we getting these different outputs?

- Let's start with the first two examples. Both use different ways of writing to standard out.

  - C stdio : user level portable library for **st**andar**d** **i**nput/**o**utput. Should work on any environment that has the C standard library
    - E.g. printf, fprintf, fputs, getline, etc.

  - POSIX C API: **P**ortable **O**perating **S**ystem **I**nterface. Functions that are supported by many operating systems to support many OS-level concepts (Input/Output, networking, processes, threads…)

Buffered writing

- By default, C `stdio` uses buffering on top of POSIX:
  - When one writes with `fwrite()`, the data being written is copied into a buffer allocated by `stdio` inside your process' address space
  - As some point, once enough data has been written, the buffer will be "flushed" to the operating system.
    - When the buffer fills (often 1024 or 4096 bytes)
  - This prevents invoking the write system call and going to the filesystem too often
    - System calls and the file system are slower than normal functions and memory accesses

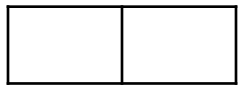
Buffered Writing Example

Arrow shows what has just finished executing.

```
int main(int argc, char** argv) {
  char *h = "hi"
  char *i = h + 1;

  FILE* fout = fopen("hi.txt", "w");

  // write "hi" one char at a time
  fwrite(h, sizeof(char), 1, fout);
  fwrite(i, sizeof(char), 1, fout);

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

buf

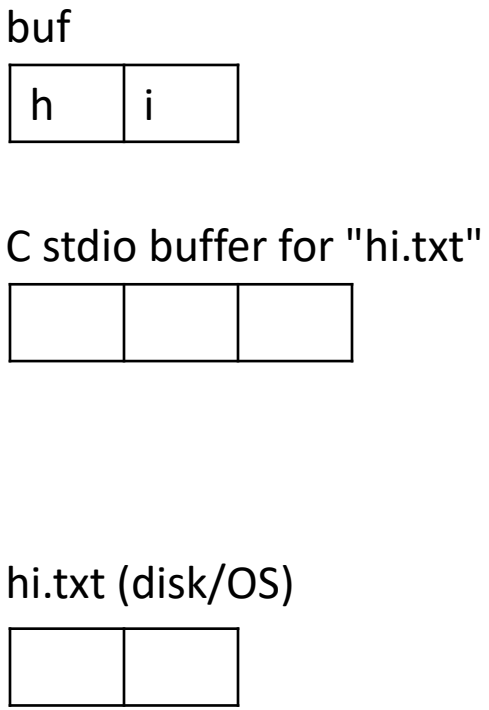
hi.txt (disk/OS)



Buffered Writing Example

Arrow shows what has just finished executing.

```
int main(int argc, char** argv) {
  char *h = "hi"
  char *i = h + 1;

  FILE* fout = fopen("hi.txt", "w");

  // write "hi" one char at a time
  fwrite(h, sizeof(char), 1, fout);
  fwrite(i, sizeof(char), 1, fout);

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

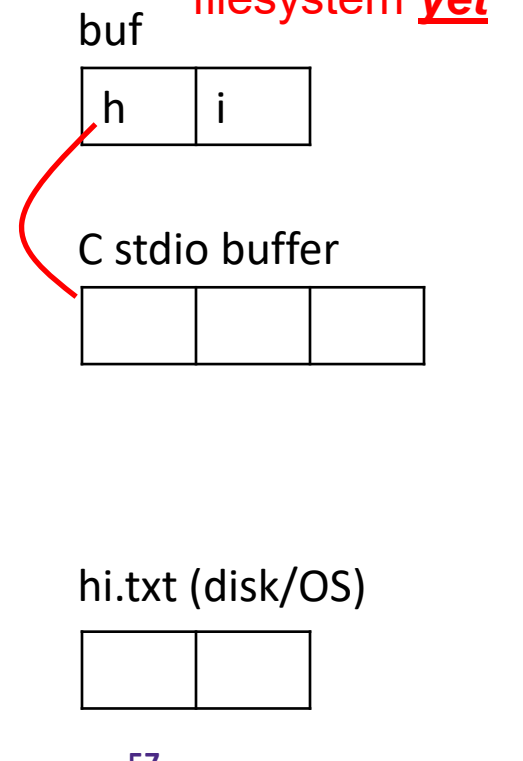


Buffered Writing Example

Arrow shows what has just finished executing.

```
int main(int argc, char** argv) {
  char *h = "hi"
  char *i = h + 1;

  FILE* fout = fopen("hi.txt", "w");

  // write "hi" one char at a time
  fwrite(h, sizeof(char), 1, fout);
  fwrite(i, sizeof(char), 1, fout);

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

Store ‘h’ into buffer, so that we do not go to filesystem ***yet***

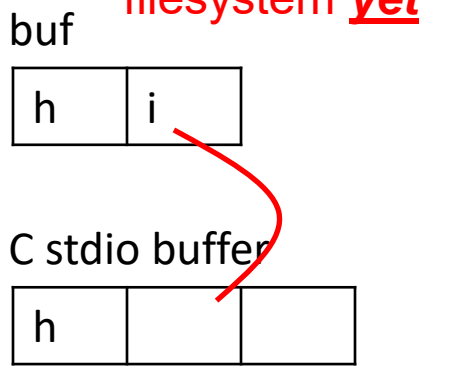


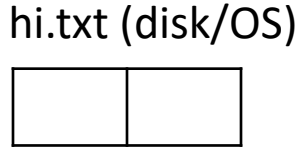
Buffered Writing Example

Arrow shows what has just finished executing.

```
int main(int argc, char** argv) {
  char *h = "hi"
  char *i = h + 1;

  FILE* fout = fopen("hi.txt", "w");

  // write "hi" one char at a time
  fwrite(h, sizeof(char), 1, fout);
  fwrite(i, sizeof(char), 1, fout);

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

Store 'i' into buffer, so that we do not go to filesystem ***yet***

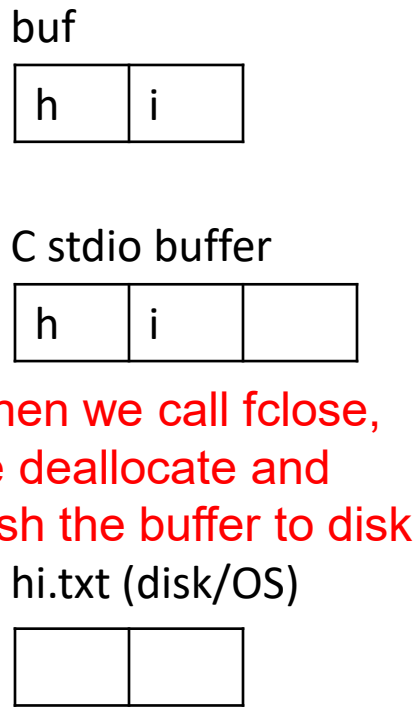




Buffered Writing Example

Arrow shows what has just finished executing.

```
int main(int argc, char** argv) {
  char *h = "hi"
  char *i = h + 1;

  FILE* fout = fopen("hi.txt", "w");

  // write "hi" one char at a time
  fwrite(h, sizeof(char), 1, fout);
  fwrite(i, sizeof(char), 1, fout);

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



Buffered Writing Example

Arrow shows what has just finished executing.

```
int main(int argc, char** argv) {
  char *h = "hi"
  char *i = h + 1;

  FILE* fout = fopen("hi.txt", "w");

  // write "hi" one char at a time
  fwrite(h, sizeof(char), 1, fout);
  fwrite(i, sizeof(char), 1, fout);

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

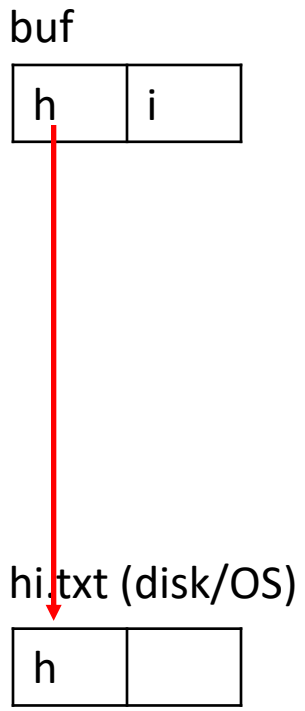
buf

hi.txt (disk/OS)

Unbuffered Writing Example

Arrow shows what has just finished executing.

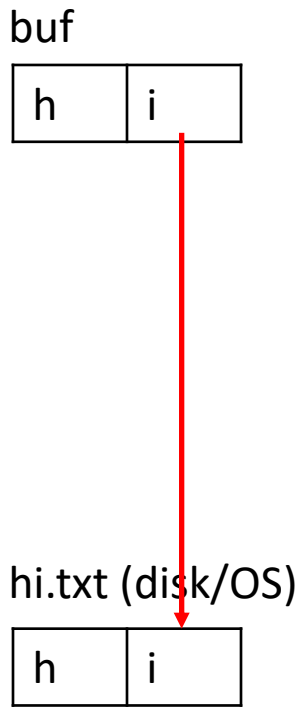
```c
int main(int argc, char** argv) {
  char buf[2] = {'h', 'i'};
  int fd = open("hi.txt", O_WRONLY | O_CREAT);

  // read "hi" one char at a time
  write(fd, &buf, sizeof(char));

  write(fd, &buf+1, sizeof(char));

  close(fd);
  return EXIT_SUCCESS;
}
```



Unbuffered Writing Example

Arrow shows what has just finished executing.

```c
int main(int argc, char** argv) {
  char buf[2] = {'h', 'i'};
  int fd = open("hi.txt", O_WRONLY | O_CREAT);

  // read "hi" one char at a time
  write(fd, &buf, sizeof(char));

  write(fd, &buf+1, sizeof(char));

  close(fd);
  return EXIT_SUCCESS;
}
```



Unbuffered Writing Example

Arrow shows what has just finished executing.

```
int main(int argc, char** argv) {
  char buf[2] = {'h', 'i'};
  int fd = open("hi.txt", O_WRONLY | O_CREAT);

  // read "hi" one char at a time
  write(fd, &buf, sizeof(char));

  write(fd, &buf+1, sizeof(char));

  close(fd);
  return EXIT_SUCCESS;
}
```

buf

h	i

Two OS/File system accesses instead of one ☹

hi.txt (disk/OS)

h	i

Buffered Reading

- By default, C `stdio` uses buffering on top of POSIX:
  - When one reads with `fread()`, a lot of data is copied into a buffer allocated by `stdio` inside your process' address space
  - Next time you read data, it is retrieved from the buffer
    - This avoids having to invoke a system call again
  - As some point, the buffer will be "refreshed":
    - When you process everything in the buffer (often 1024 or 4096 bytes)
  - Similar thing happens when you write to a file

Quick Aside: strace

- Basic usage: **`strace ./program [args]`**
- strace exec's a program and "traces" the system calls that it utilizes to execute.
- Will show you the system call, its arguments, return values and more!
  - "Arguments are printed in symbolic form with passion."
- Really useful tool. See example usage below.

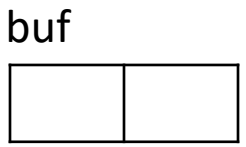
```
$ strace cat /dev/null
…
open("/dev/null", O_RDONLY) = 3
…
```

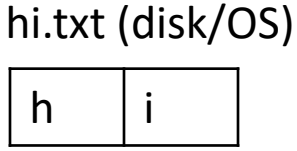
- We can use this to see if the past couple of slides are true or not.

Buffered Reading Example

Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  char buf[2];
  FILE* fin = fopen("hi.txt", "rb");

  // read "hi" one char at a time
  fread(&buf, sizeof(char), 1, fin);

  fread(&buf+1, sizeof(char), 1, fin);

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

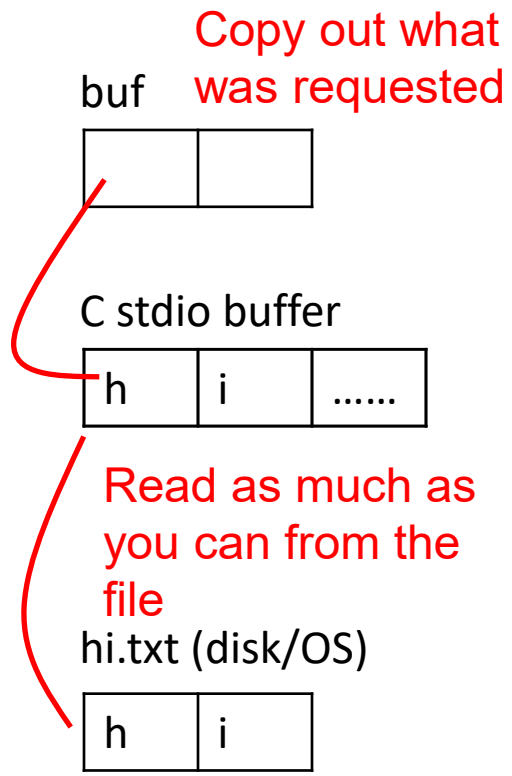




Buffered Reading Example

Arrow signifies what will be executed next

```c
int main(int argc, char** argv) {
  char buf[2];
  FILE* fin = fopen("hi.txt", "rb");

  // read "hi" one char at a time
  fread(&buf, sizeof(char), 1, fin);

  fread(&buf+1, sizeof(char), 1, fin);

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

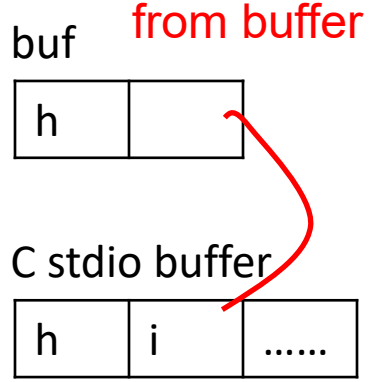


Buffered Reading Example

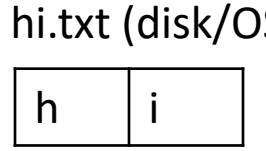
Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  char buf[2];
  FILE* fin = fopen("hi.txt", "rb");

  // read "hi" one char at a time
  fread(&buf, sizeof(char), 1, fin);

  fread(&buf+1, sizeof(char), 1, fin);

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

Get next char from buffer



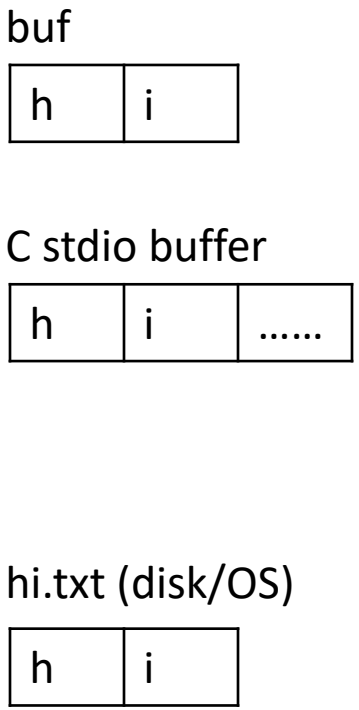
No need to go to file!



Buffered Reading Example

Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  char buf[2];
  FILE* fin = fopen("hi.txt", "rb");

  // read "hi" one char at a time
  fread(&buf, sizeof(char), 1, fin);

  fread(&buf+1, sizeof(char), 1, fin);

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



Buffered Reading Example

Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  char buf[2];
  FILE* fin = fopen("hi.txt", "rb");

  // read "hi" one char at a time
  fread(&buf, sizeof(char), 1, fin);

  fread(&buf+1, sizeof(char), 1, fin);

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

buf

h	i

hi.txt (disk/OS)

h	i

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

- 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
Singular write

Quick Aside: exit vs _exit

```
int exit(int status);
```

- Causes the process to terminate, but before it does it calls all “installed” exit functions.
- Additionally, it flushes and closes all `stdio` buffers before the process terminates.
  - By default, “return” calls exit() somewhere down the line. If you’d like to see this, examine the obj file created by a binary.

```
int _exit(int status);
```

- Causes the process to terminate immediately. There is no flushing.
  - Check it out for yourself. Everything that uses fwrite will not actually write to the file.

Fork Problem Explained

Arrow signifies what will be executed next.
I execute processes in parallel and "in sync" for demonstration purposes

- Remember: printf (and stdio) buffers input in the programs address space

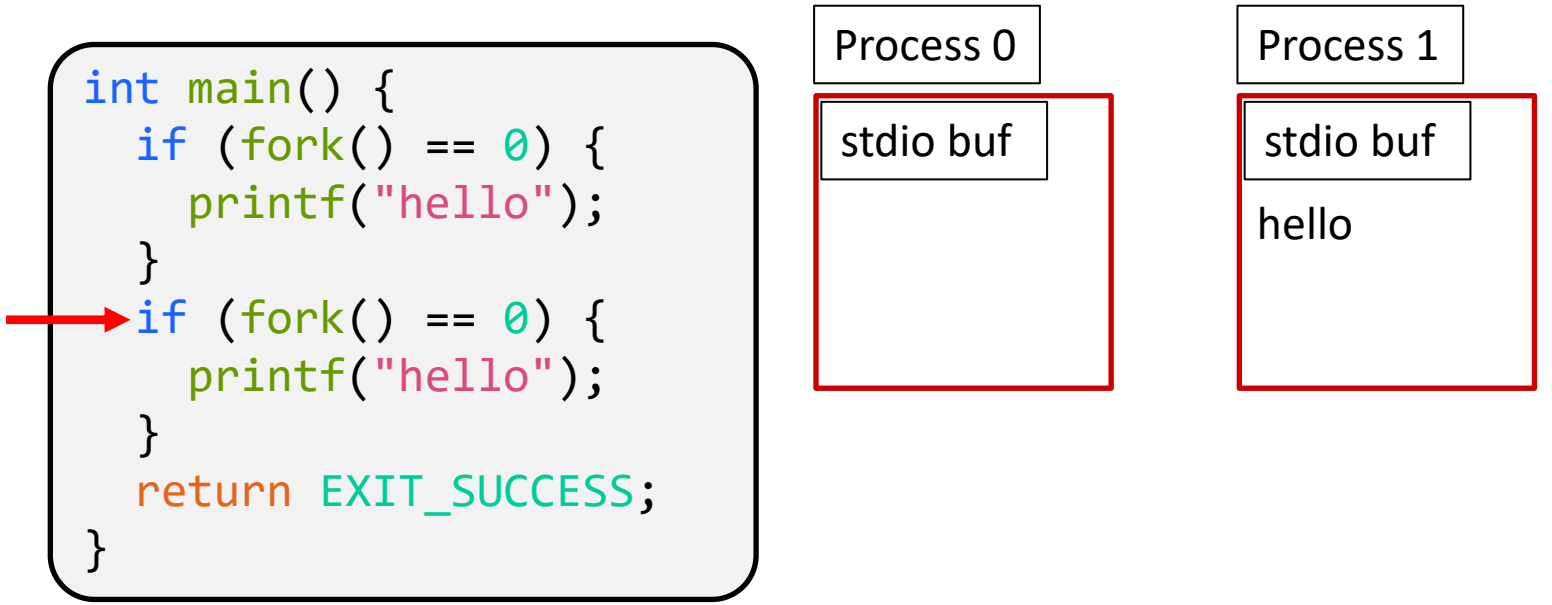
```
int main() {
  if (fork() == 0) {
    printf("hello");
  }
  if (fork() == 0) {
    printf("hello");
  }
  return EXIT_SUCCESS;
}
```

Fork Problem Explained

Arrow signifies what will be executed next.
I execute processes in parallel and "in sync" for demonstration purposes

- Remember: printf (and stdio) buffers input in the programs address space

```
int main() {
  if (fork() == 0) {
    printf("hello");
  }
  if (fork() == 0) {
    printf("hello");
  }
  return EXIT_SUCCESS;
}
```

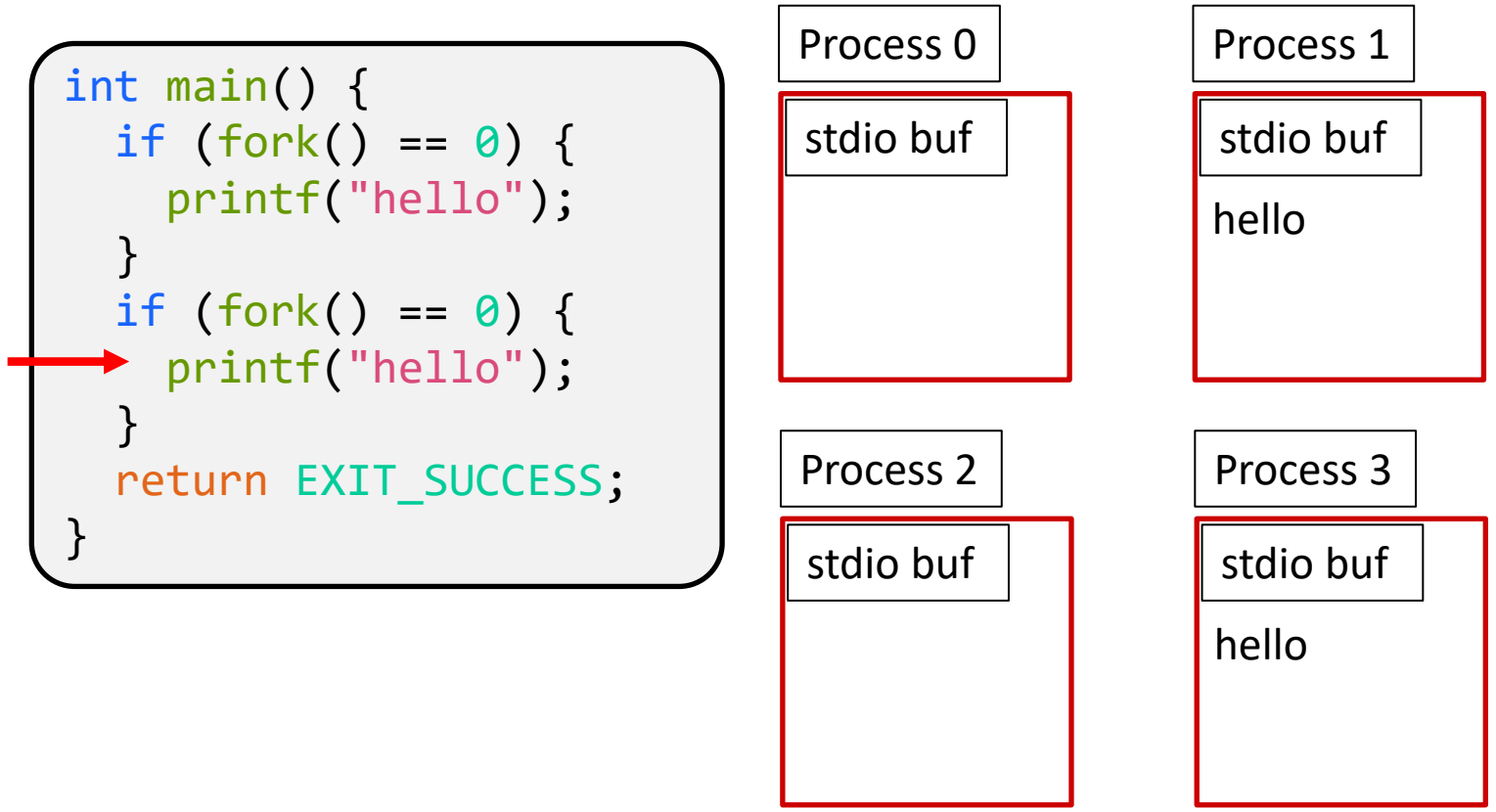


Fork Problem Explained

Arrow signifies what will be executed next.
I execute processes in parallel and "in sync" for demonstration purposes

- Remember: printf (and stdio) buffers input in the programs address space

```
int main() {
  if (fork() == 0) {
    printf("hello");
  }
  if (fork() == 0) {
    printf("hello");
  }
  return EXIT_SUCCESS;
}
```

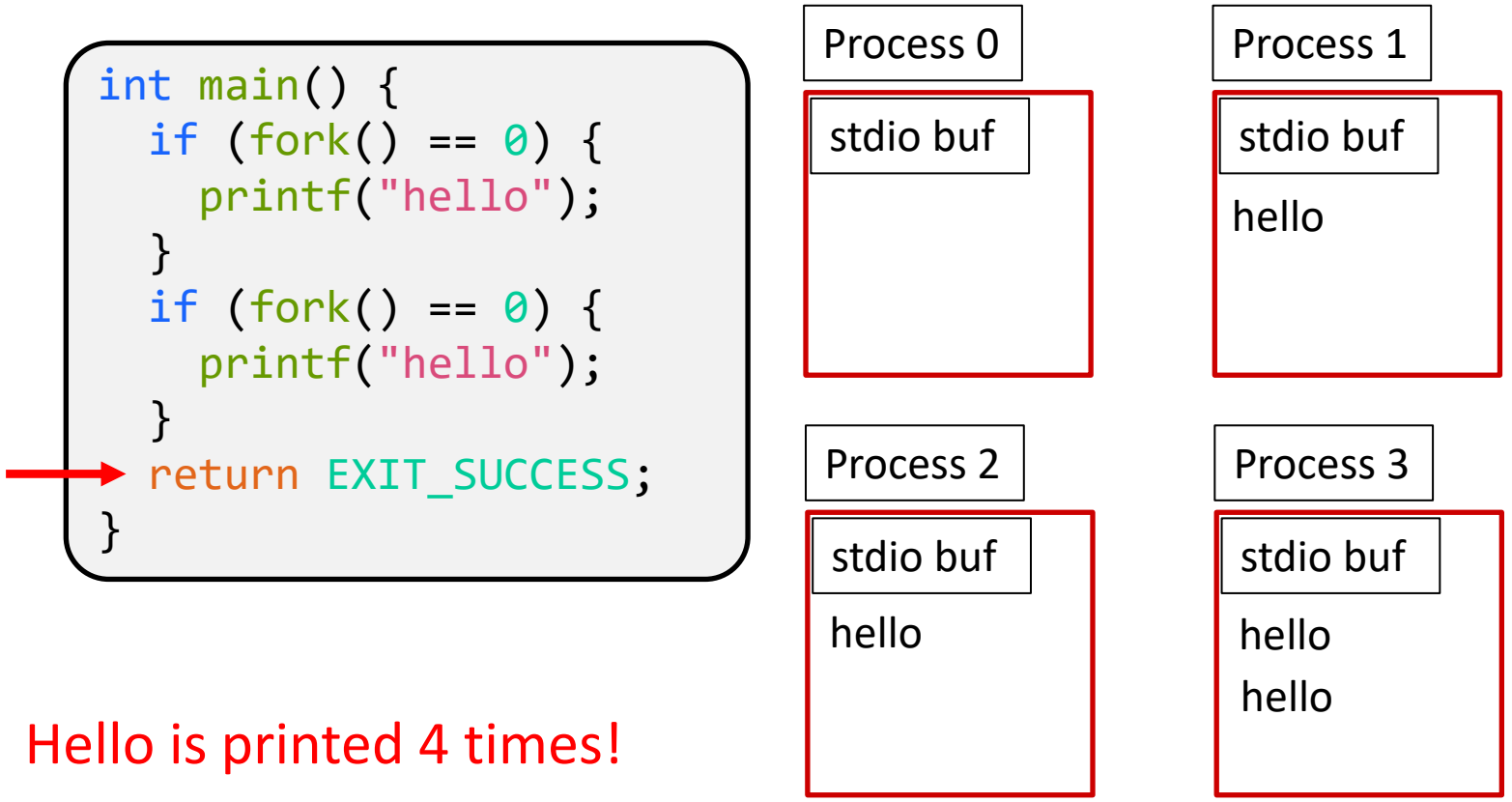


Fork Problem Explained

Arrow signifies what will be executed next.
I execute processes in parallel and "in sync" for demonstration purposes

- Remember: printf (and stdio) buffers input in the programs address space

```
int main() {
  if (fork() == 0) {
    printf("hello");
  }
  if (fork() == 0) {
    printf("hello");
  }
  return EXIT_SUCCESS;
}
```



Hello is printed 4 times!

Fork Problem Explained (pt.2)

- Why did we get different outputs when printf printed a newline character after hello?
  - Only difference was:

```
printf("hello");
```
vs
```
printf("hello\n");
```

- All we needed to do to get the expected output was add a \n. why?

- `printf` prints to `stdout` and by default `stdout` is line buffered because it refers to a TTY. Meaning it flushes the buffer on a newline character
  - If we ran `./prog > out.txt` (redirect the output of the program to a file), we would get different output since buffering policy changes.

How to flush/modify the cstdio buffer

- For C stdio:
  - ```
    int fflush(FILE* stream);
    ```
  - Flushes the stream to the OS/filesystem
  - ```
    int setvbuf(FILE* stream, char* buf,
                int mode, size_t size);
    ```
  - Has a family of related functions like setbuf(), setbuffer(), setlinebuf();
  - Can set the stream to be unbuffered or a specified buffer

How to flush POSIX?

- When we write to a file with POSIX it is sent to the filesystem, is it immediately sent to disc? No
  - Well, we do have the block cache… so it may not be written to disc
  - Since all File I/O requests go to the file system, if another process accesses the same file, then it should see the data even if it is the block cache and not in disc.
  - If we lose power though…

How to flush POSIX to disk

- Two functions
  - `int fsync(int fd);`
  - Flushes all in-core data and metadata to the storage medium
  - `int fdatasync(int fd);`
  - Sends the file data to disk
  - Does not flush modified metadata unless necessary for data.

That's all!

- Next time: The File System ☺