OS Start: Processes & Fork

Computer Systems Programming, Spring 2025

Instructor: Travis McGaha

Teaching Assistants:

Andrew Lukashchuk
Angie Cao
Aniket Ghorpade
Ashwin Alaparthi
Austin Lin
Hassan Rizwan
Lobi Zhao
Pearl Liu
Perrie Quek

Poll Everywhere

pollev.com/tqm

- How are you?

Administrivia

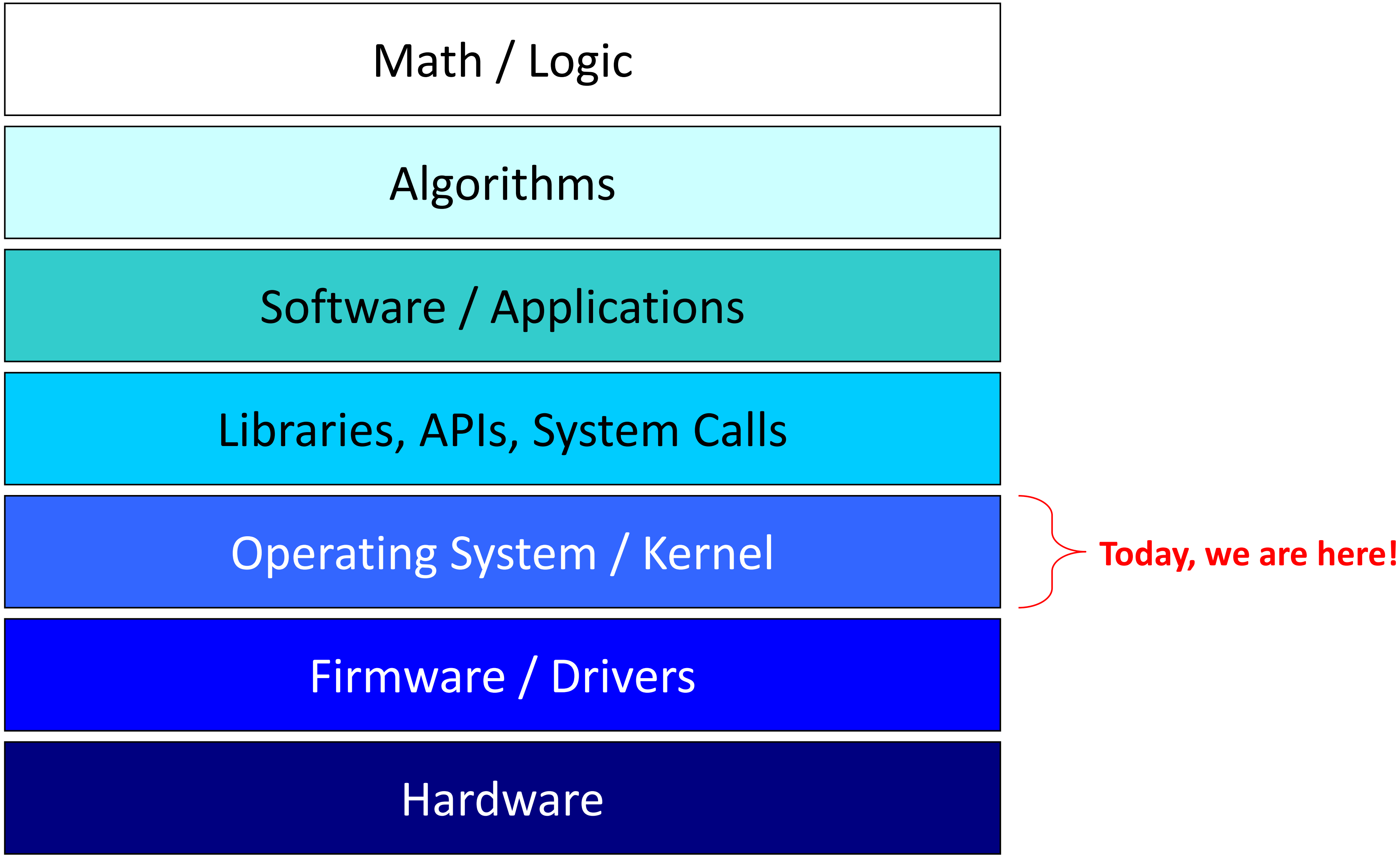
- Simplekv (HW04)
  - Due Friday (2/14)
  - Recommend taking a look sooner rather than later
    - Once you figure out what data members you need, consider talking to a TA or I about it
  - Is more work than previous assignments, not a lot though.

- Check-in 01
  - Was “Due” before this lecture, extended to Wednesday
  - Will re-open assignments soon.

Lecture Outline

- **The OS**
- Processes & fork()
- execvp()

Remember This?



Math / Logic

Algorithms

Software / Applications

Libraries, APIs, System Calls

Operating System / Kernel

Firmware / Drivers

Hardware

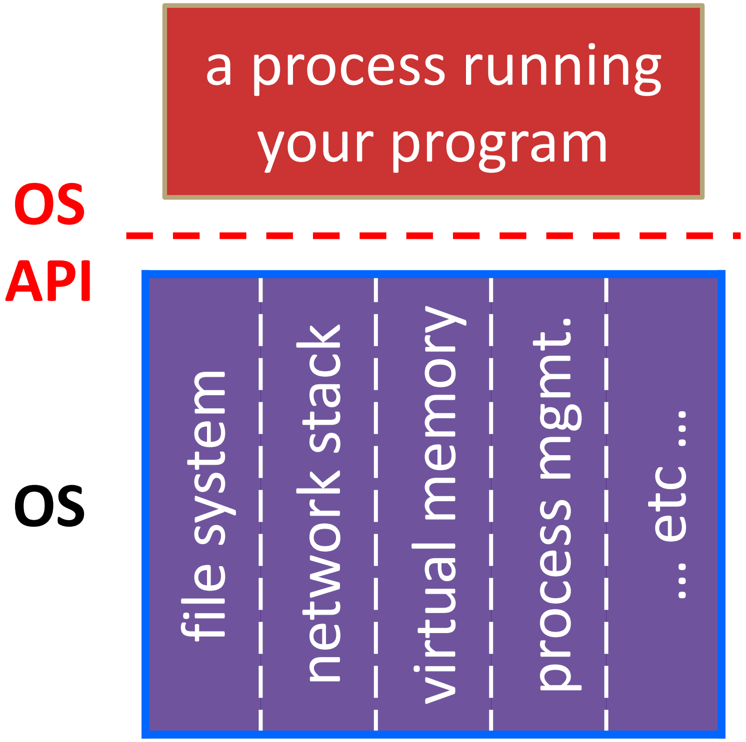
Today, we are here!

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
  - Abstracts away messy hardware devices
    - Provides high-level, convenient, portable abstractions (*e.g.* files, disk blocks)
  - Manages (allocates, schedules, protects) hardware resources
    - Decides which programs have permission to access which files, memory locations, pixels on the screen, etc. and when

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.



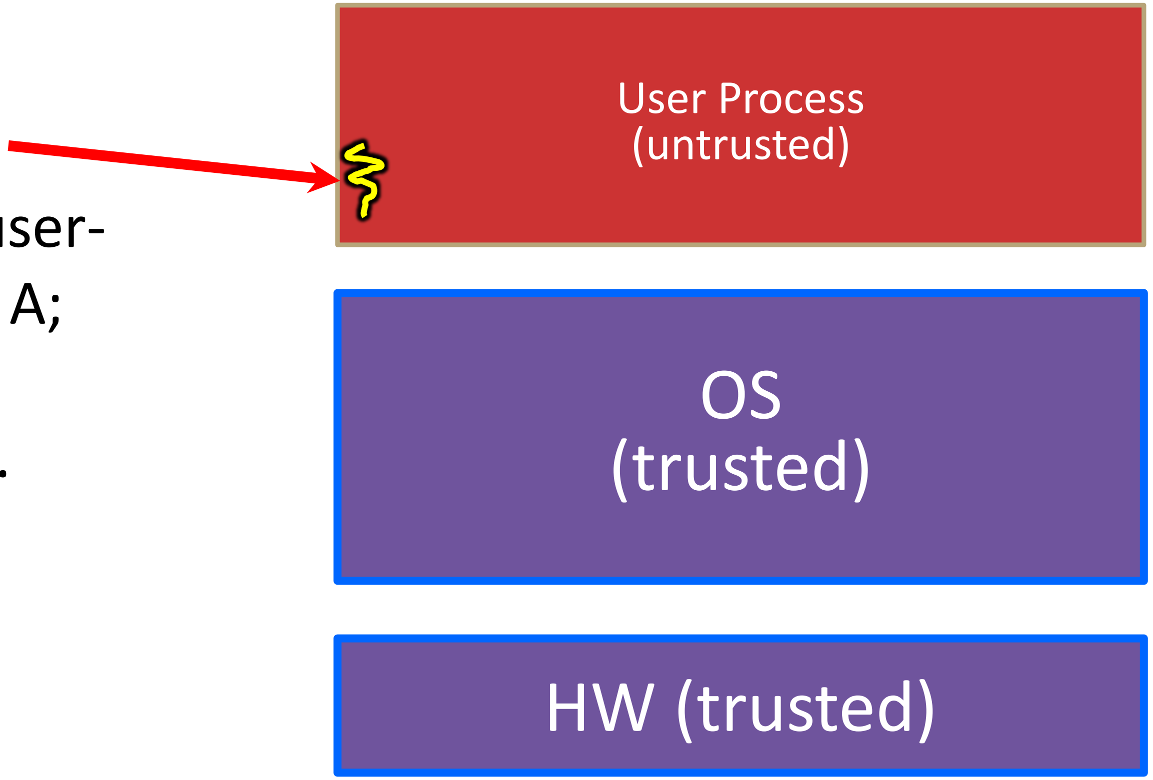
File System
- open(), read(), write(), close(), …

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

Virtual Memory
- brk(), shm_open(), …

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

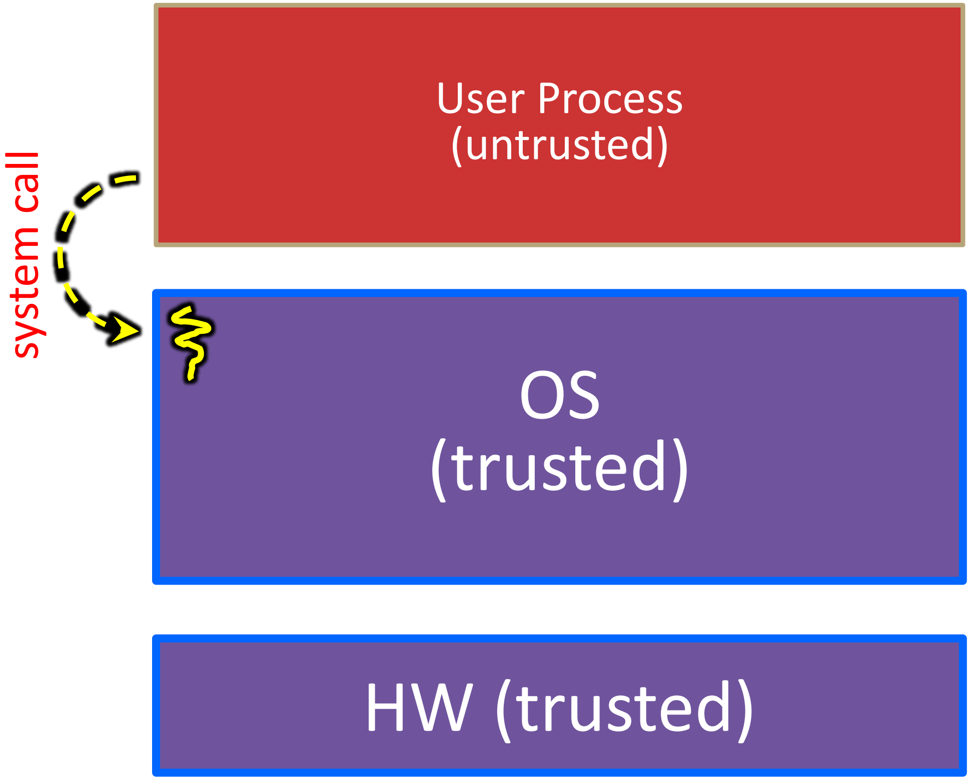
System Call Trace (high-level view)



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

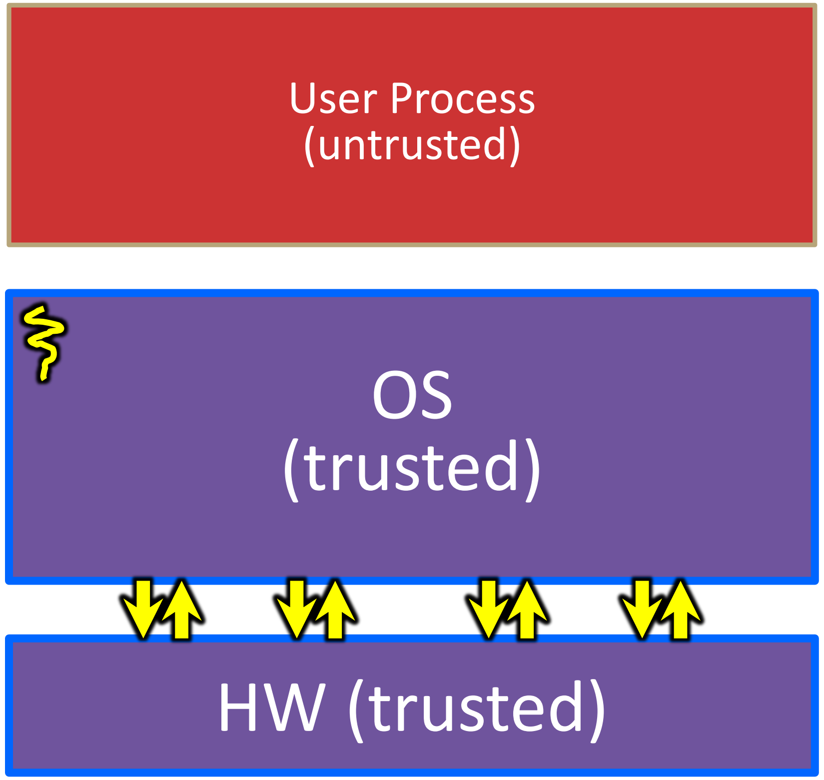
System Call Trace (high-level view)

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.

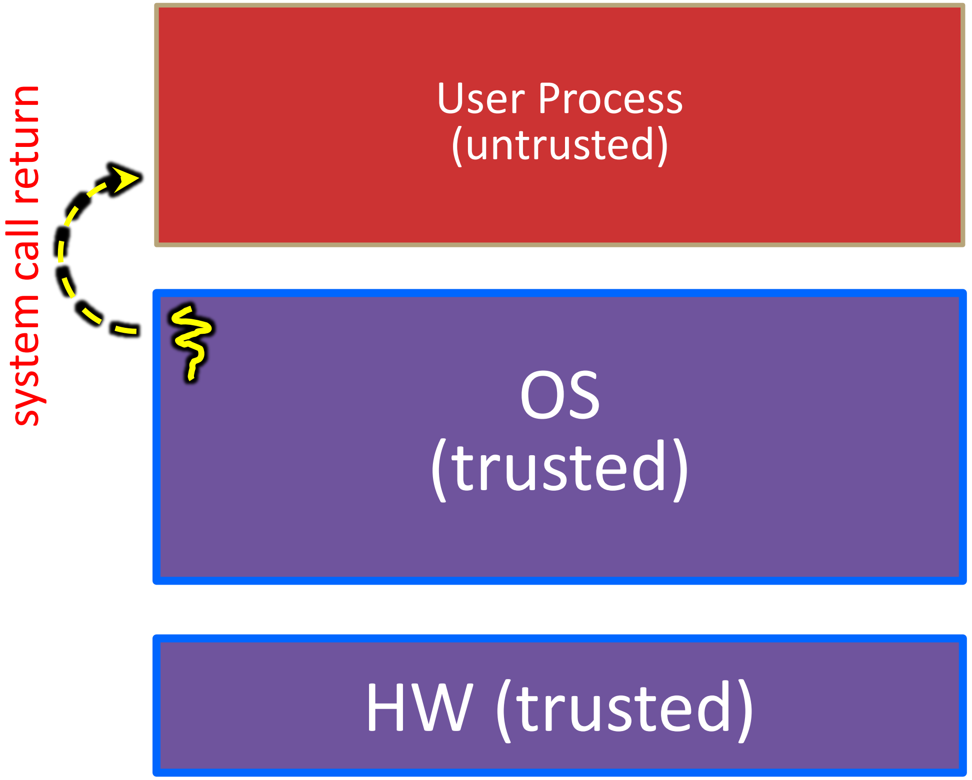


System Call Trace (high-level view)

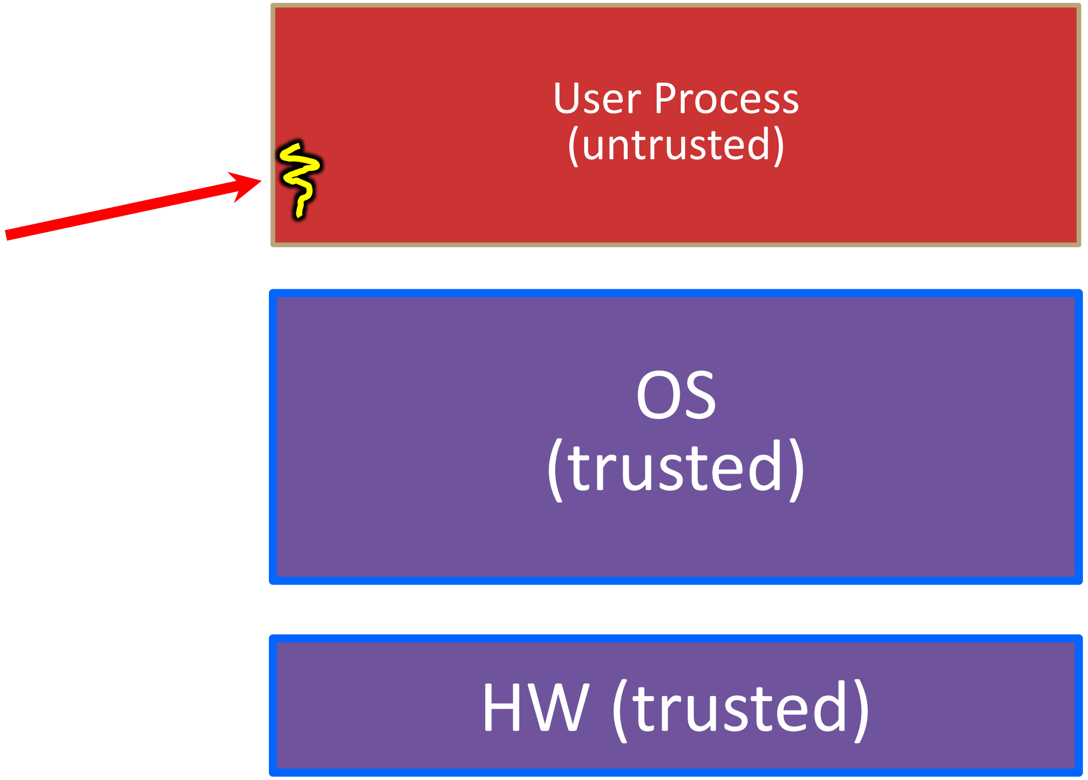
Once the OS has finished servicing the system call, which might involve long waits as it interacts with HW, it:

(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.



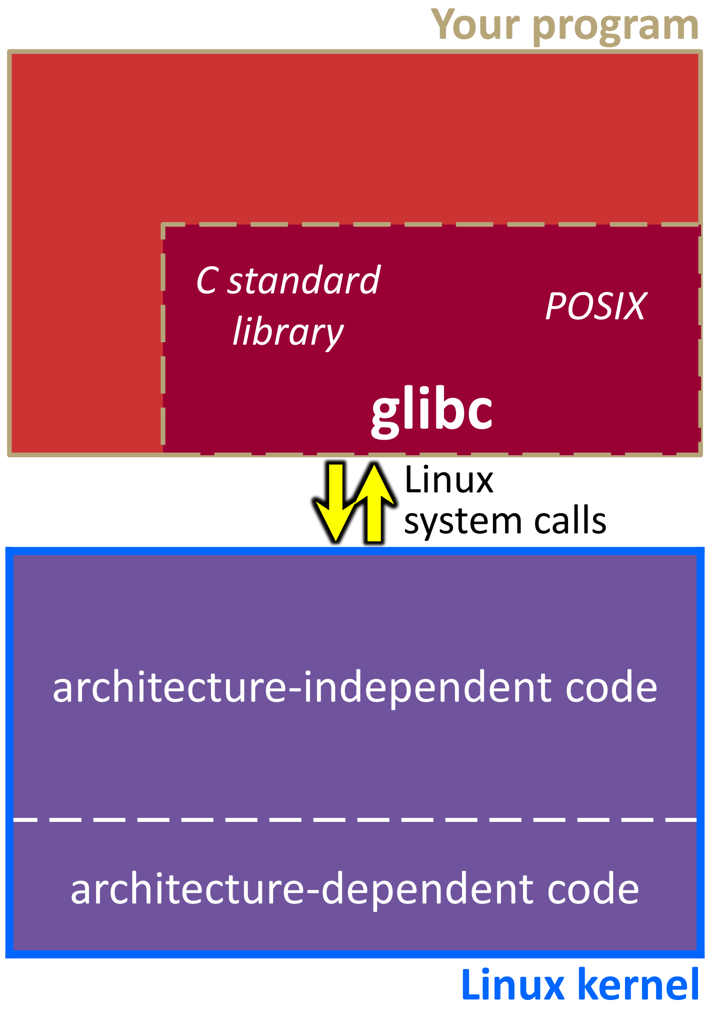
System Call Trace (high-level view)



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

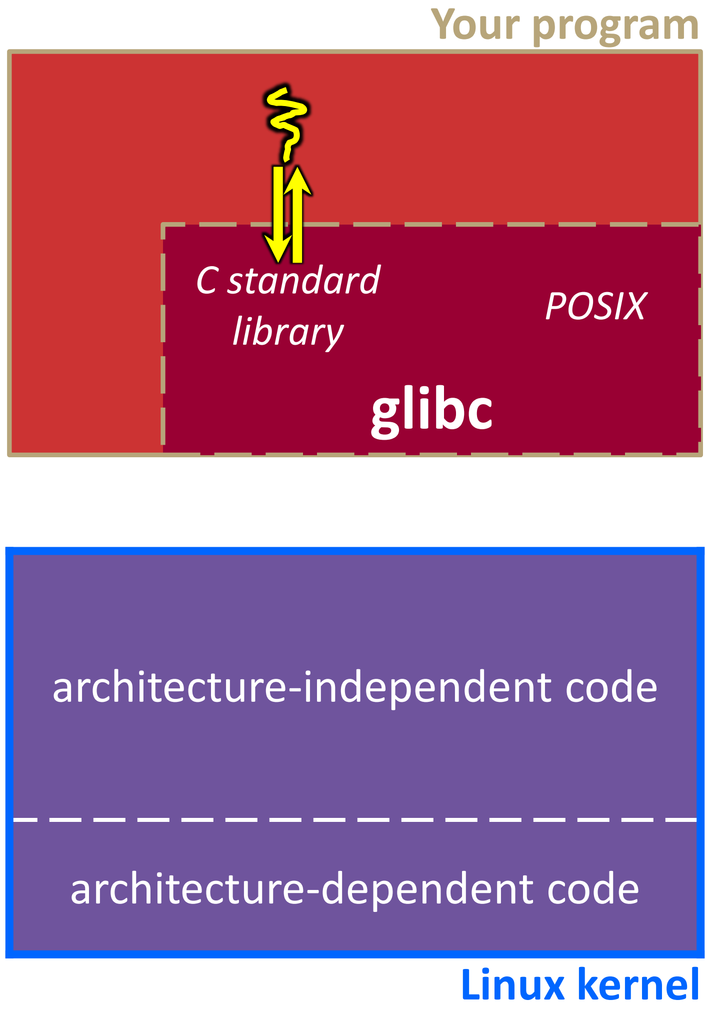
“Library calls” on x86/Linux

- 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



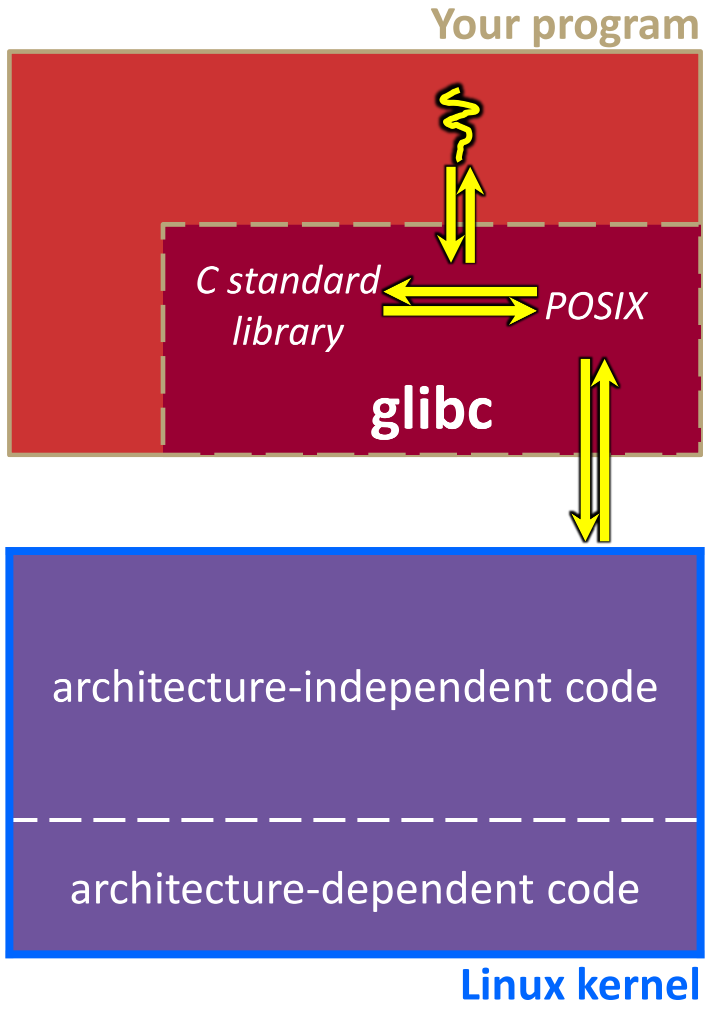
"Library calls" on x86/Linux: Option 1

- 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!



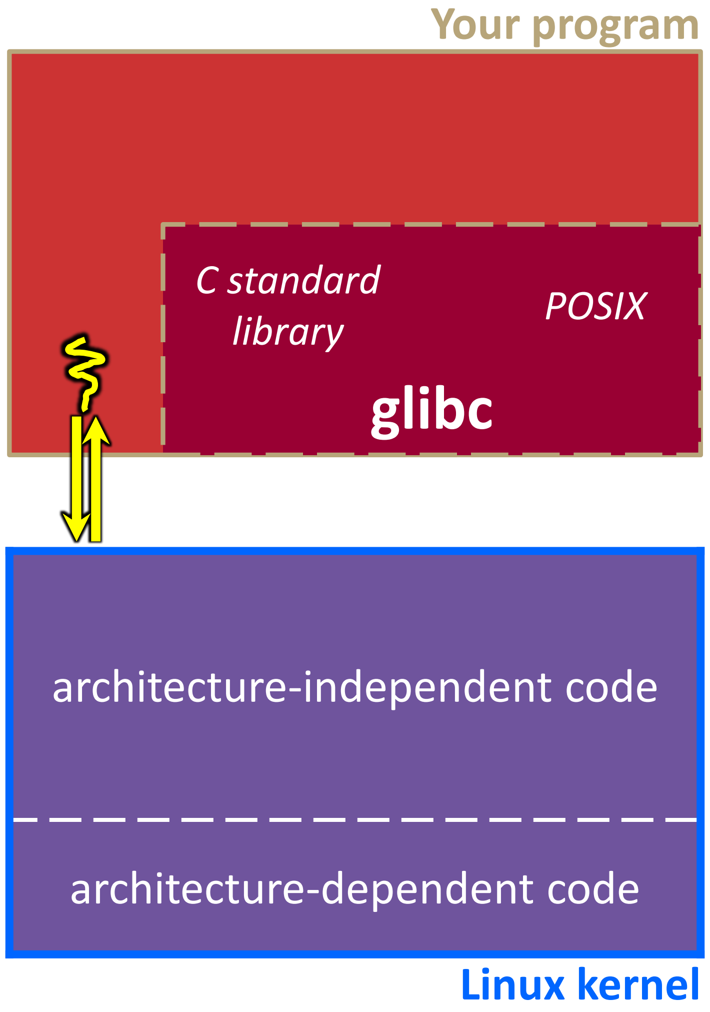
“Library calls” on x86/Linux: Option 2

- Some routines may be handled by `glibc`, but they in turn invoke Linux system calls
  - *e.g.* POSIX wrappers around Linux `syscall`s
    - 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



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.

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)

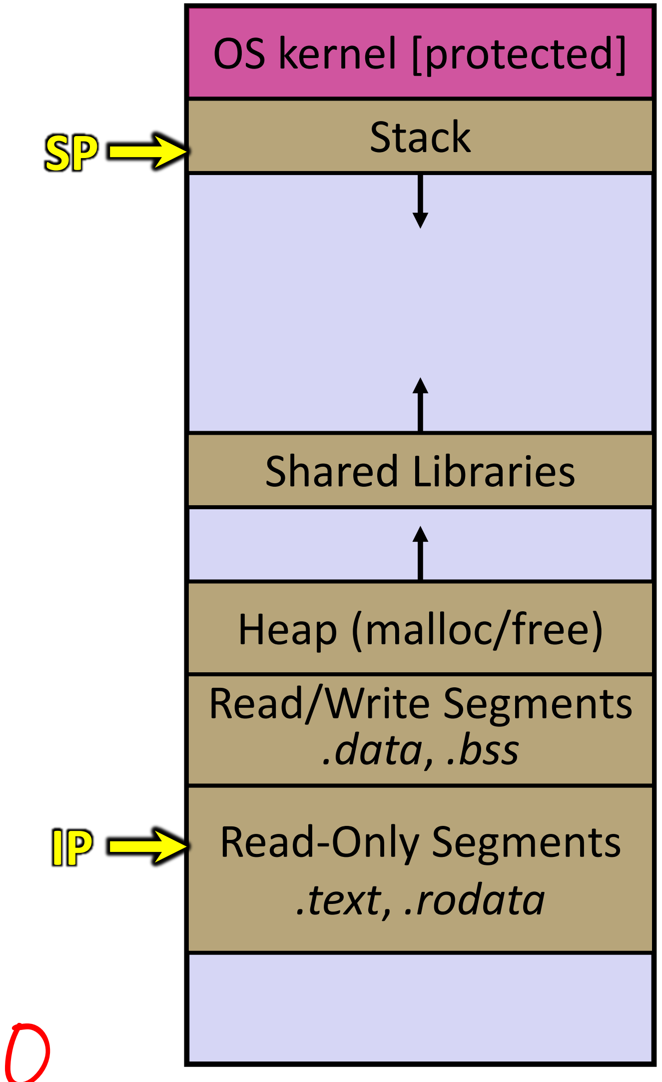
Lecture Outline

- The OS
- **Processes & fork()**
- execvp()

Definition: Process

- Definition: An instance of a program that is being executed (or is ready for execution)

- Consists of:
  - Memory (code, heap, stack, etc)
  - Registers used to manage execution (stack pointer, program counter, ...)
  - Other resources

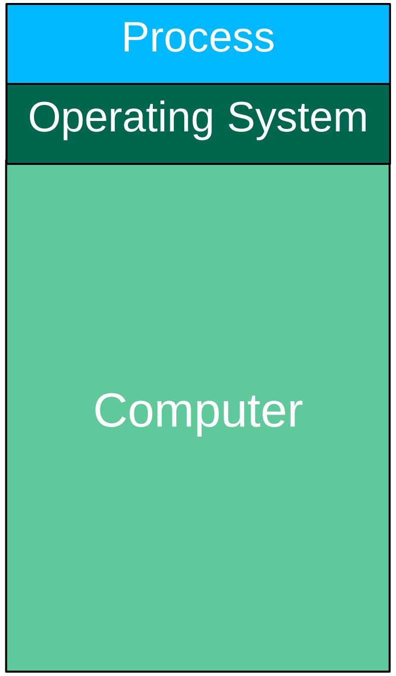


* This isn't quite true
more in CIS 4480/5480

Computers as we know them now

- In CIS 2400, you learned about hardware, transistors, CMOS, gates, etc.
- Once we got to programming, our computer looks something like:

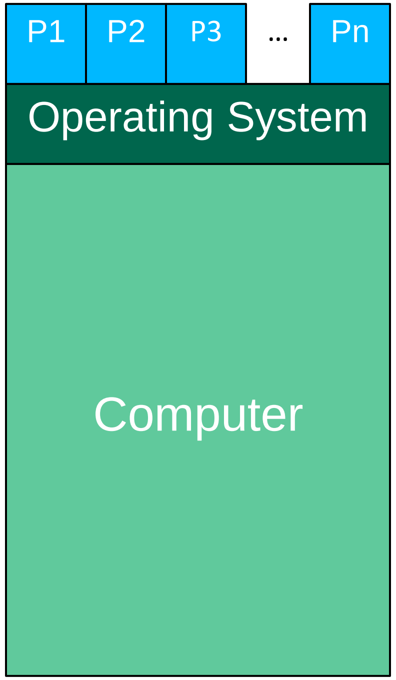
What is missing/wrong with this?



- This model is still useful, and can be used in many settings

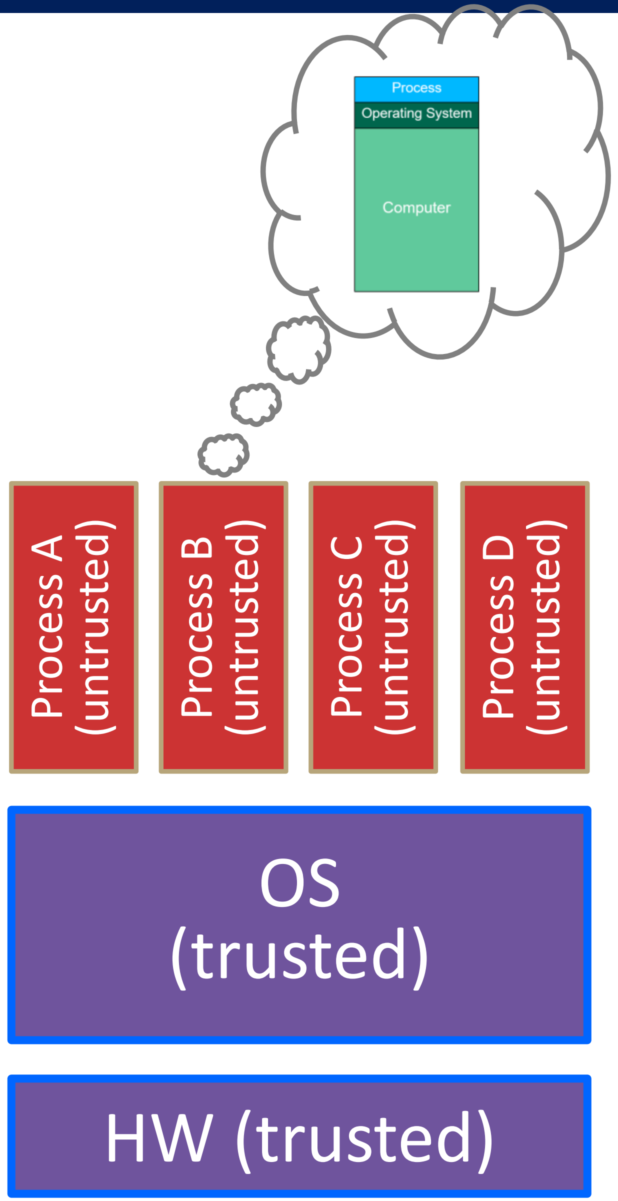
Multiple Processes

- Computers run multiple processes “at the same time”
- One or more processes for each of the programs on your computer
- Each process has its own…
  - Memory space
  - Registers
  - Resources

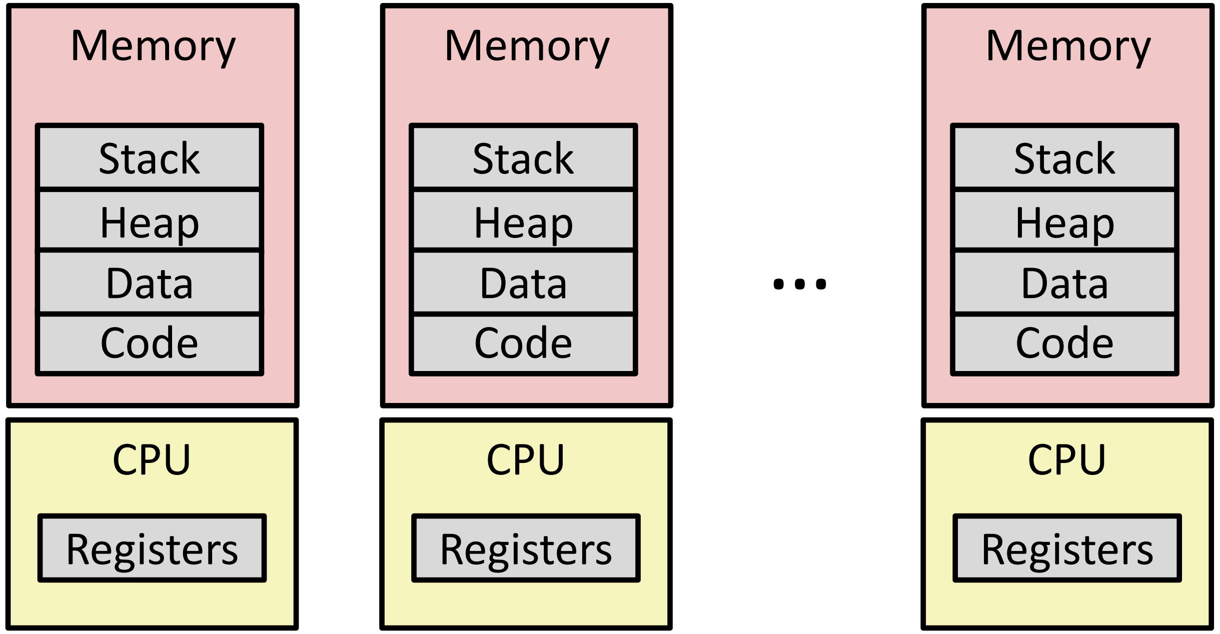


OS: Protection System

- OS isolates process from each other
  - Each process seems to have exclusive use of memory and the processor.
    - This is an **illusion**
    - More on Memory when we talk about virtual memory later in the course
  - OS permits controlled sharing between processes
    - E.g. through files, the network, etc.
- OS isolates itself from processes
  - Must prevent processes from accessing the hardware directly

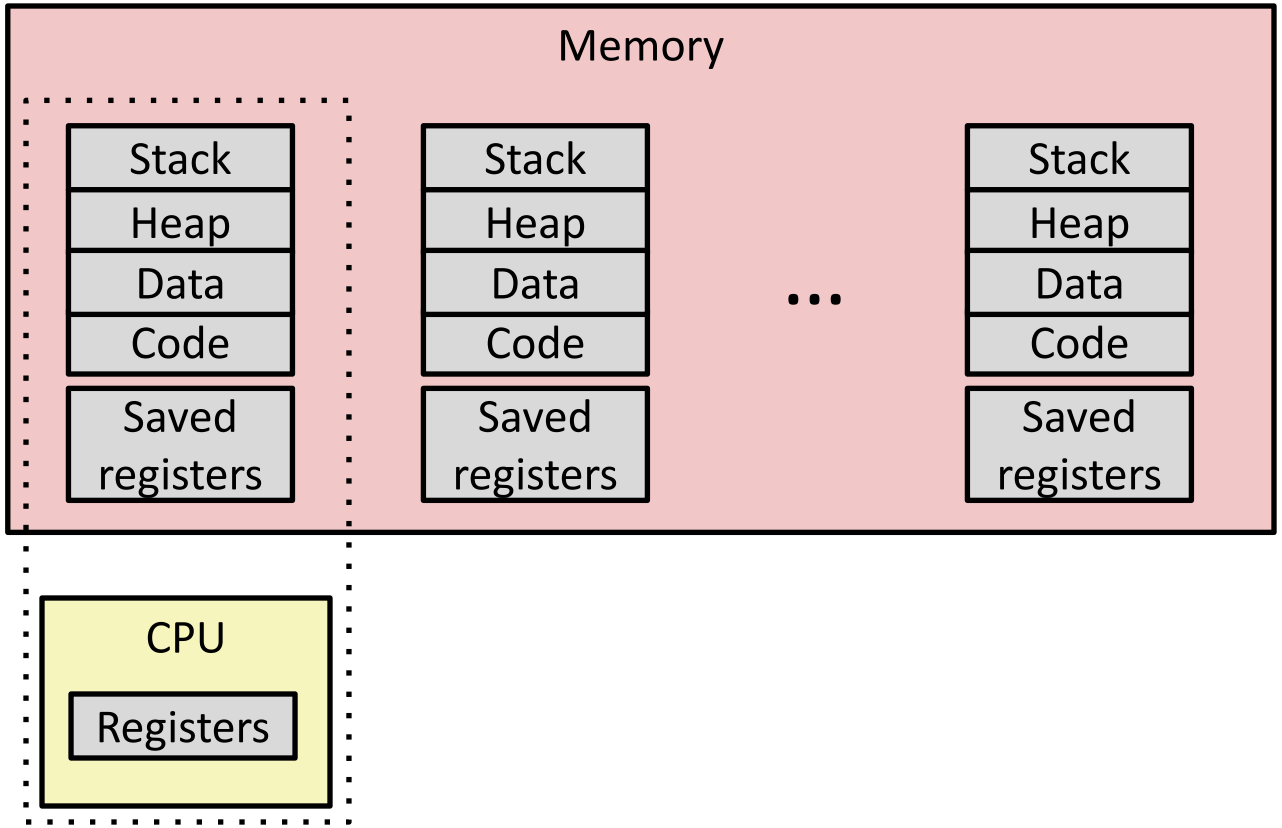


Multiprocessing: The Illusion



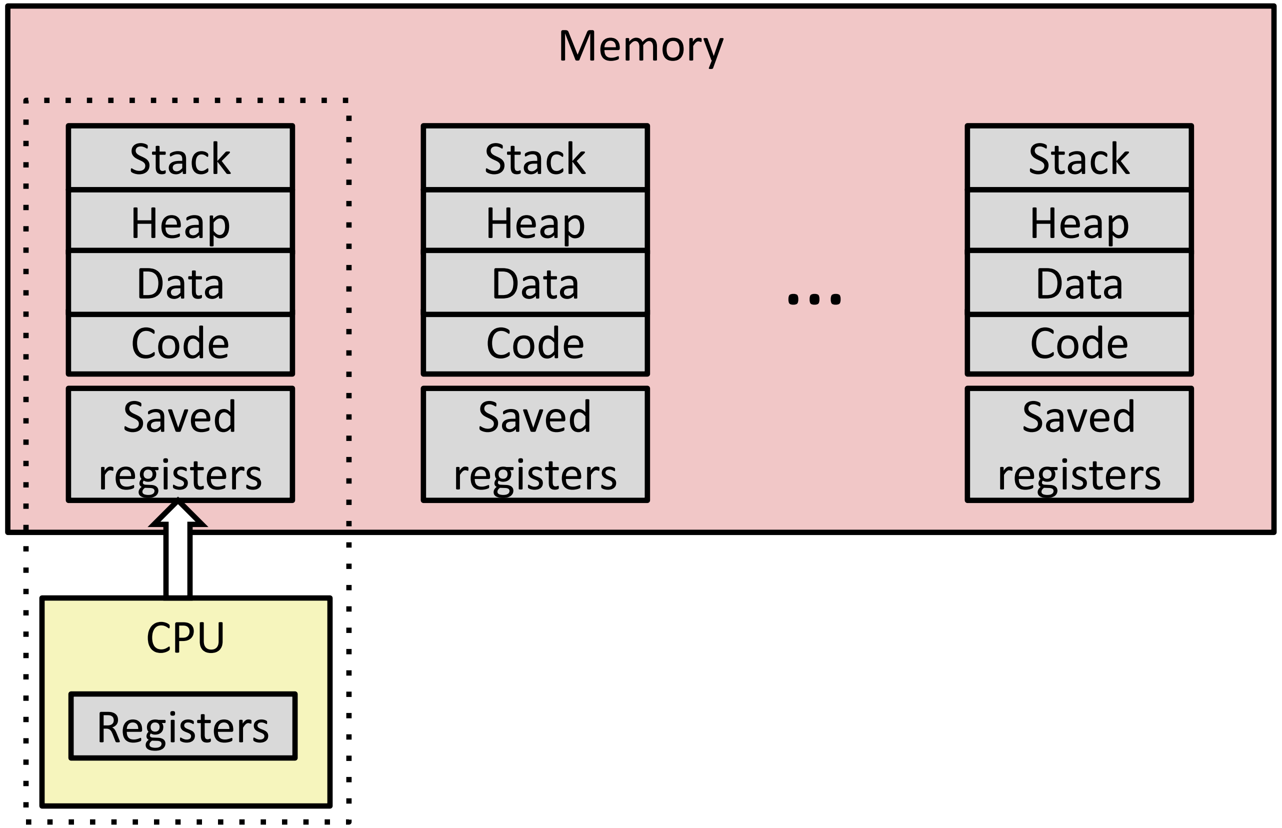
- Computer runs many processes simultaneously
  - Applications for one or more users
    - Web browsers, email clients, editors, …
  - Background tasks
    - Monitoring network & I/O devices

Multiprocessing: The (Traditional) Reality



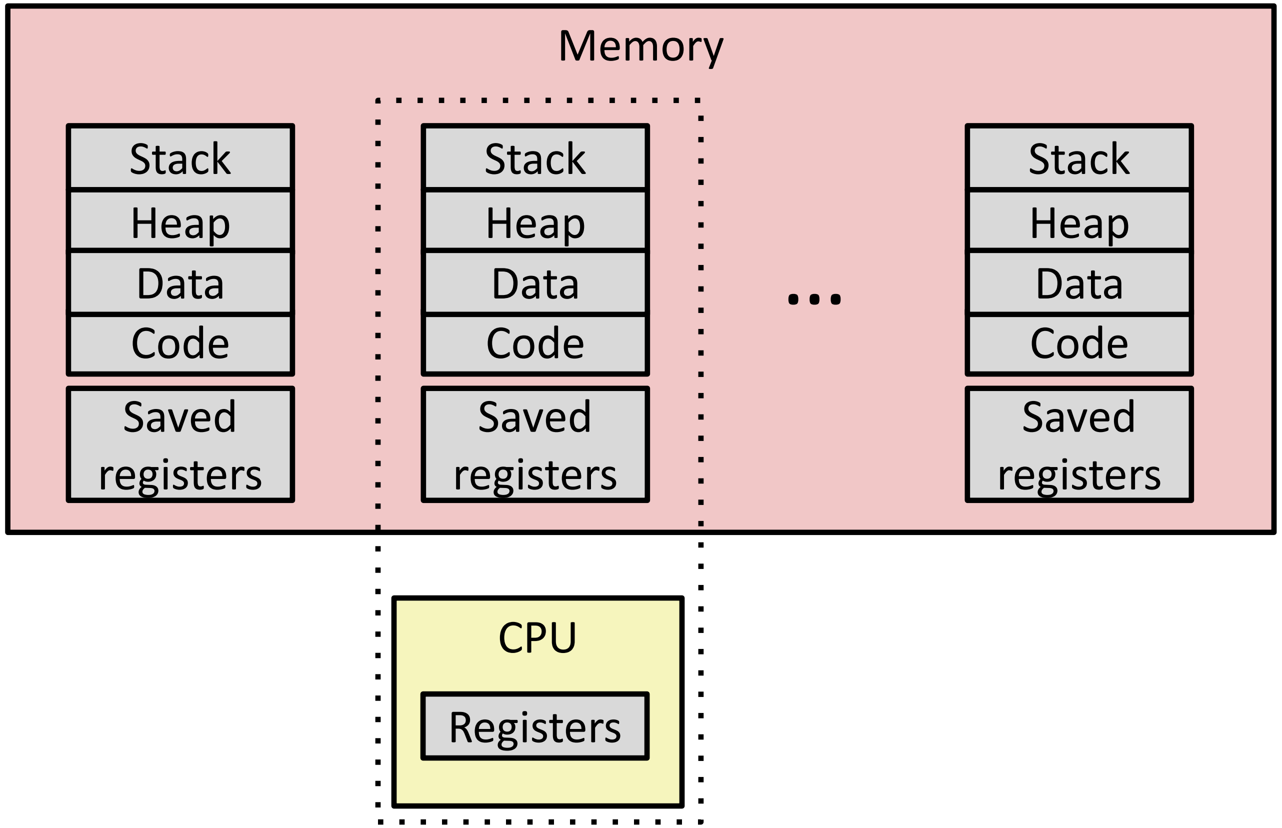
- Single processor executes multiple processes concurrently
  - Process executions interleaved (multitasking)
  - Address spaces managed by virtual memory system (later in course)
  - Register values for nonexecuting processes saved in memory

Multiprocessing: The (Traditional) Reality



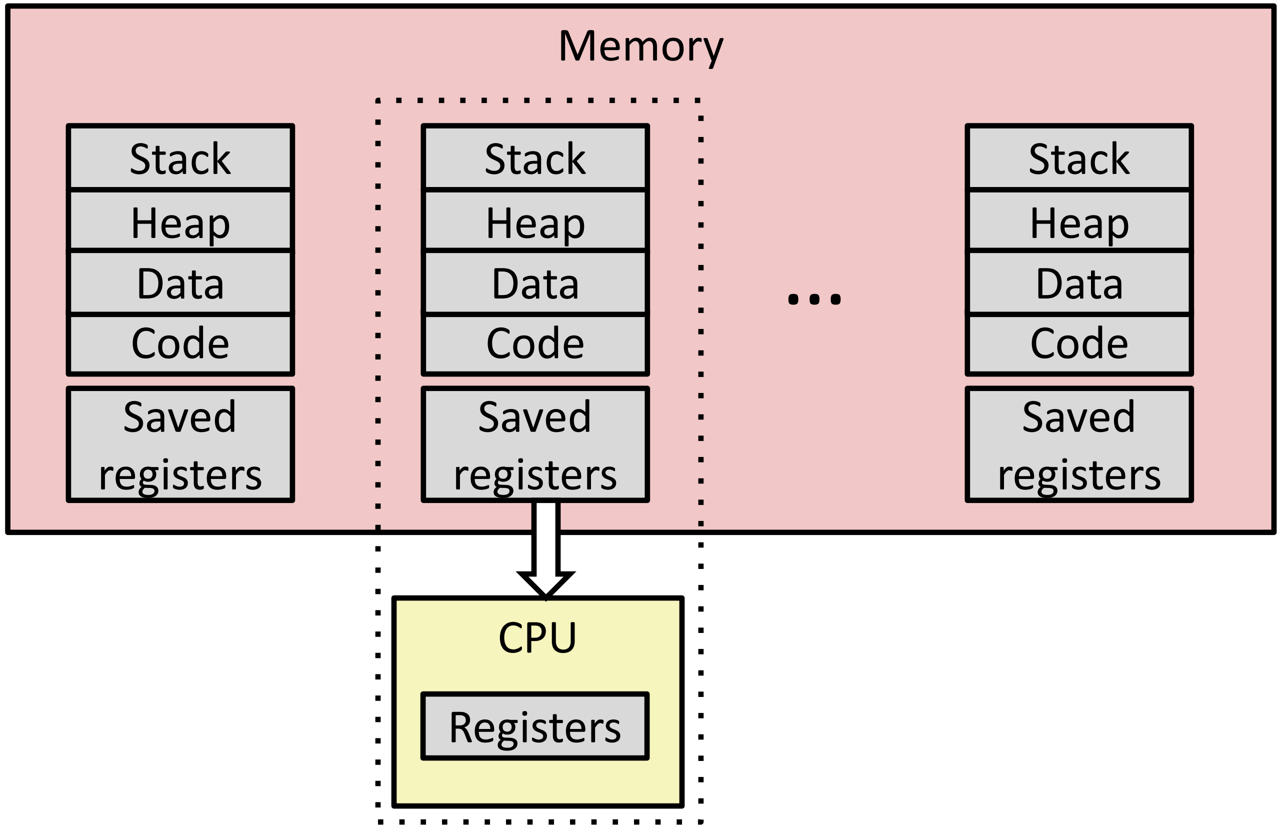
1. Save current registers in memory

Multiprocessing: The (Traditional) Reality



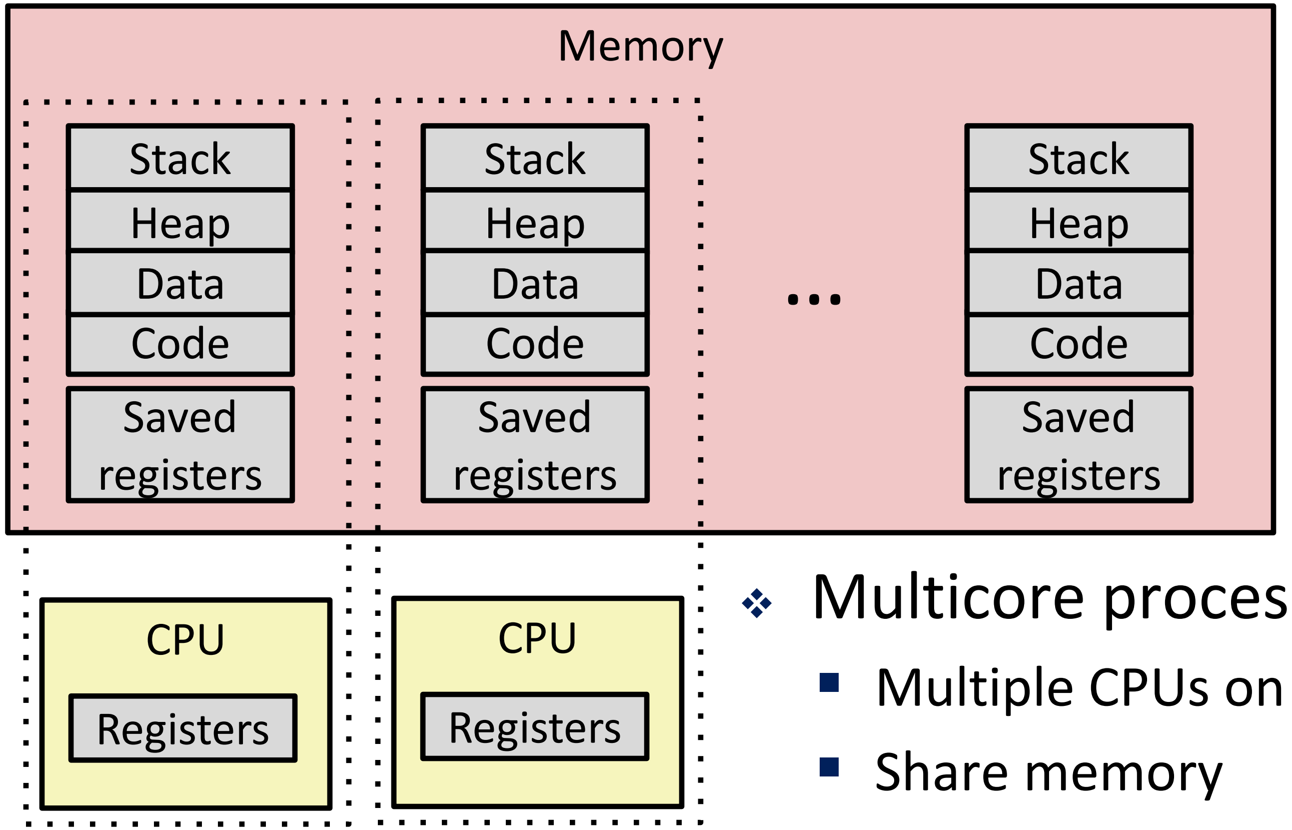
1. Save current registers in memory
2. Schedule next process for execution

Multiprocessing: The (Traditional) Reality



1. Save current registers in memory
2. Schedule next process for execution
3. Load saved registers and switch address space (context switch)

Multiprocessing: The (Modern) Reality



- Multicore processors
  - Multiple CPUs on single chip
  - Share memory
  - Each can execute a separate process
    - Scheduling of processors onto cores done by kernel
  - This is called “Parallelism”

Poll Everywhere

pollev.com/tqm

- What I just went through was the big picture of processes. Many details left, some will be gone over in future lectures

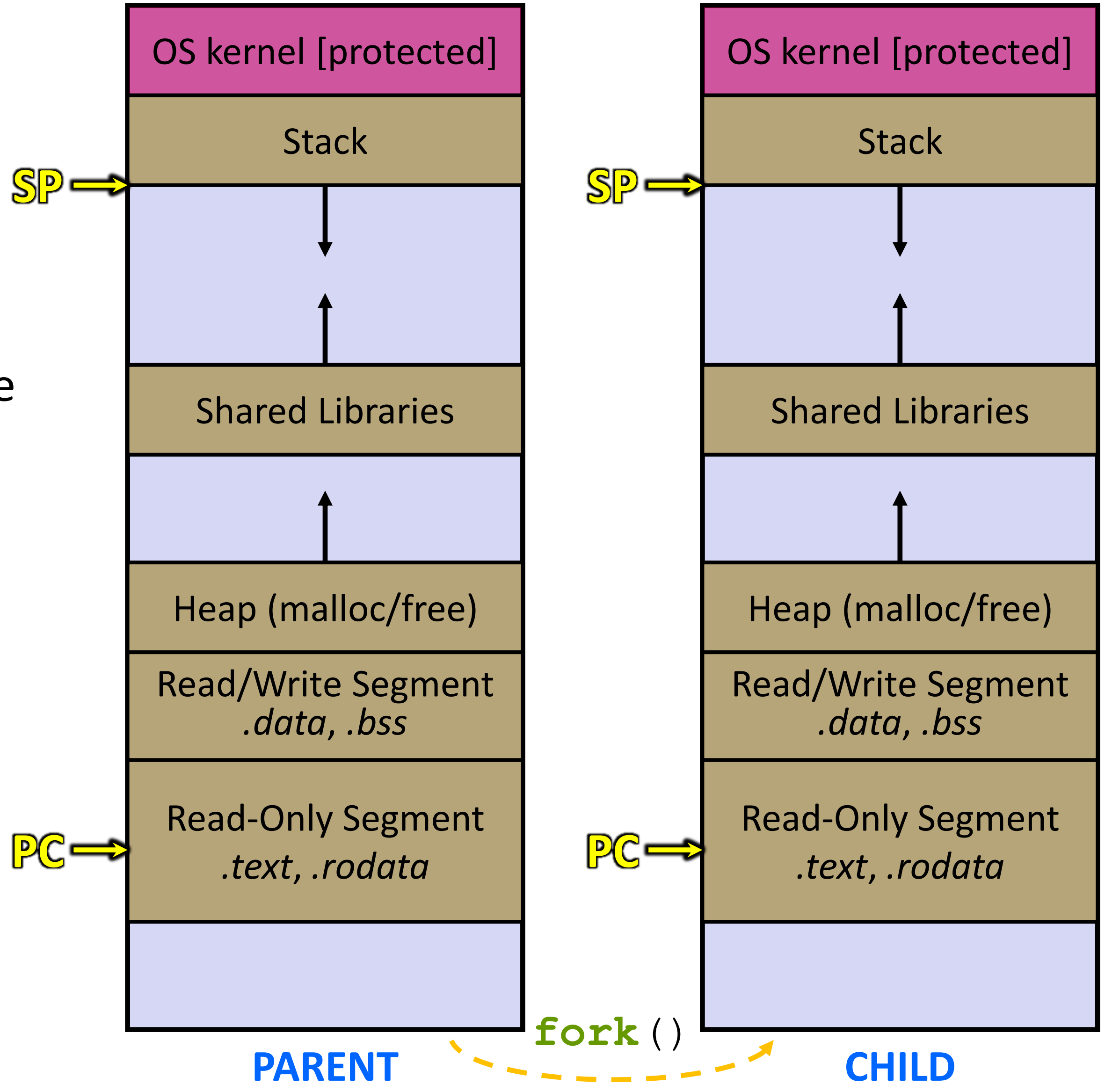
- Any questions, comments or concerns so far?

Creating New Processes

- `pid_t fork();`
  - Creates a new process (the “child”) that is an *exact clone** of the current process (the “parent”)
    - *almost everything
  - The new process has a separate virtual address space from the parent
  - Returns a **`pid_t`** which is an integer type.

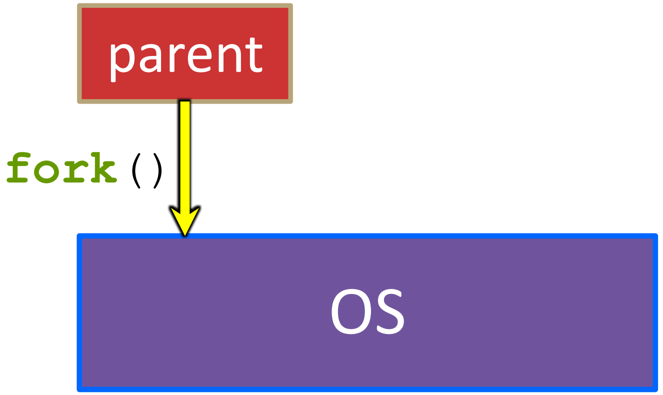
`fork()` and Address Spaces

- Fork causes the OS to clone the address space
  - The *copies* of the memory segments are (nearly) identical
  - The new process has *copies* of the parent's data, stack-allocated variables, open file descriptors, etc.



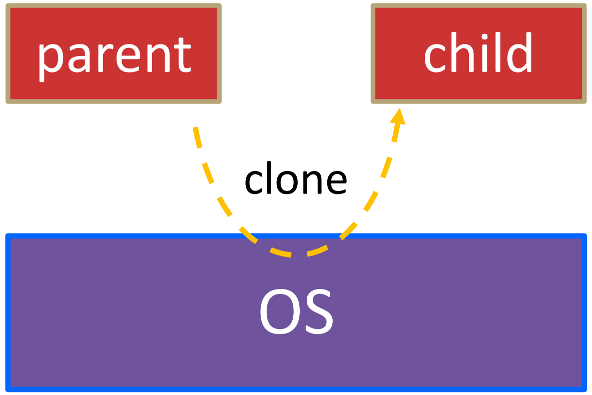
`fork()`

- `fork()` has peculiar semantics
  - The parent invokes `fork()`
  - The OS clones the parent
  - *Both* the parent and the child return from fork
    - Parent receives child's pid
    - Child receives a 0



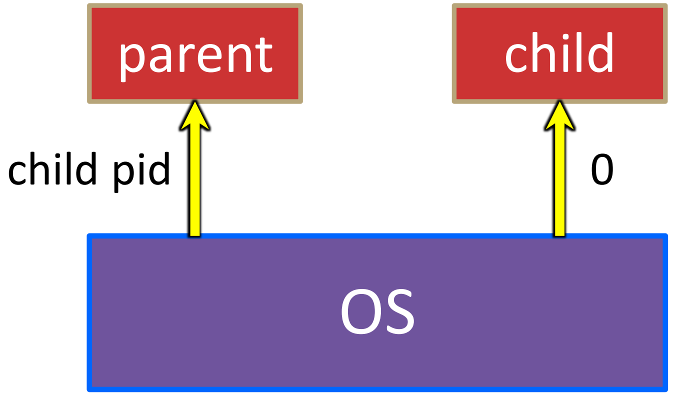
`fork()`

- `fork()` has peculiar semantics
  - The parent invokes `fork()`
  - The OS clones the parent
  - *Both* the parent and the child return from fork
    - Parent receives child's pid
    - Child receives a 0



`fork()`

- `fork()` has peculiar semantics
  - The parent invokes `fork()`
  - The OS clones the parent
  - *Both* the parent and the child return from fork
    - Parent receives child's pid
    - Child receives a 0

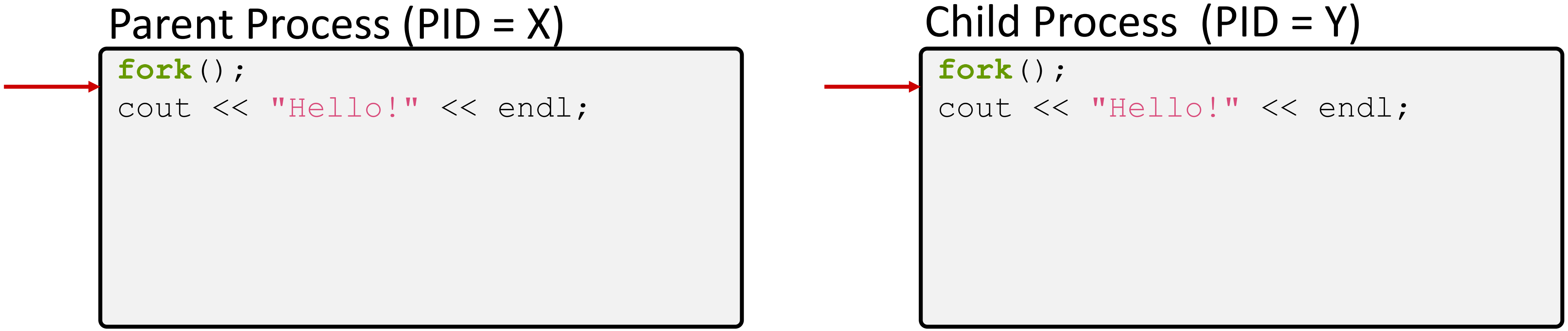


"simple" `fork()` example

```
fork();
cout << "Hello!" << endl;
```

- What does this print?

"simple" `fork()` example



Parent Process (PID = X)

```
fork();
cout << "Hello!" << endl;
```

Child Process (PID = Y)

```
fork();
cout << "Hello!" << endl;
```

- What does this print?

- "Hello!\n" is printed twice

Poll Everywhere

pollev.com/tqm

```
fork();
fork();
cout << "Hello!" << endl;
```

- What does this print?

Poll Everywhere

pollev.com/tqm

```
int x = 3;
fork();
x++;
cout << x << endl;
```

- What does this print?

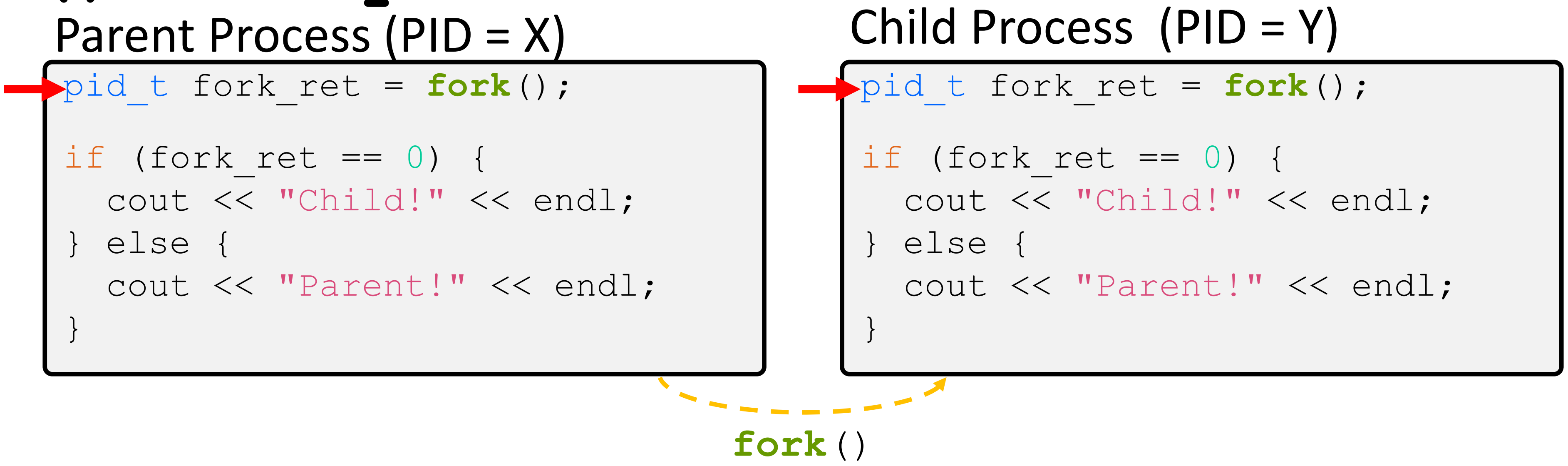
Poll Everywhere

pollev.com/tqm

```
pid_t fork_ret = fork();

if (fork_ret == 0) {
  cout << "Child!" << endl;
} else {
  cout << "Parent!" << endl;
}
```

- What does this print?

`fork()` example



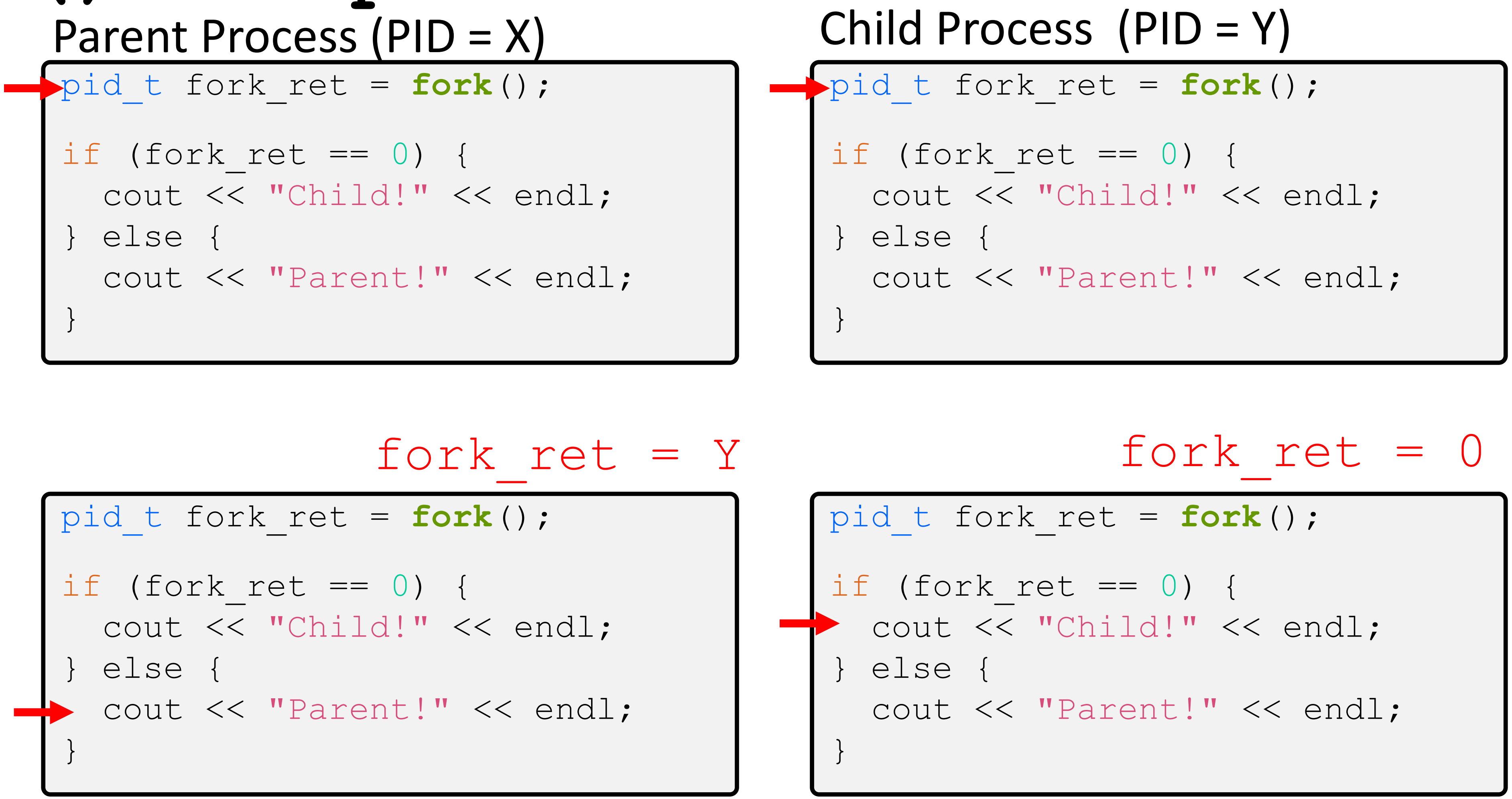
Parent Process (PID = X)

```
pid_t fork_ret = fork();

if (fork_ret == 0) {
  cout << "Child!" << endl;
} else {
  cout << "Parent!" << endl;
}
```

Child Process (PID = Y)

```
pid_t fork_ret = fork();

if (fork_ret == 0) {
  cout << "Child!" << endl;
} else {
  cout << "Parent!" << endl;
}
```

`fork()` example



Parent Process (PID = X)

```
pid_t fork_ret = fork();

if (fork_ret == 0) {
  cout << "Child!" << endl;
} else {
  cout << "Parent!" << endl;
}
```

Child Process (PID = Y)

```
pid_t fork_ret = fork();

if (fork_ret == 0) {
  cout << "Child!" << endl;
} else {
  cout << "Parent!" << endl;
}
```

`fork_ret = Y`

```
pid_t fork_ret = fork();

if (fork_ret == 0) {
  cout << "Child!" << endl;
} else {
  cout << "Parent!" << endl;
}
```

`fork_ret = 0`

```
pid_t fork_ret = fork();

if (fork_ret == 0) {
  cout << "Child!" << endl;
} else {
  cout << "Parent!" << endl;
}
```

Prints "Parent"

Which prints first?

Prints "Child"

Process States (incomplete)

FOR NOW, we can think of a process as being in one of three states:

More states in future lectures

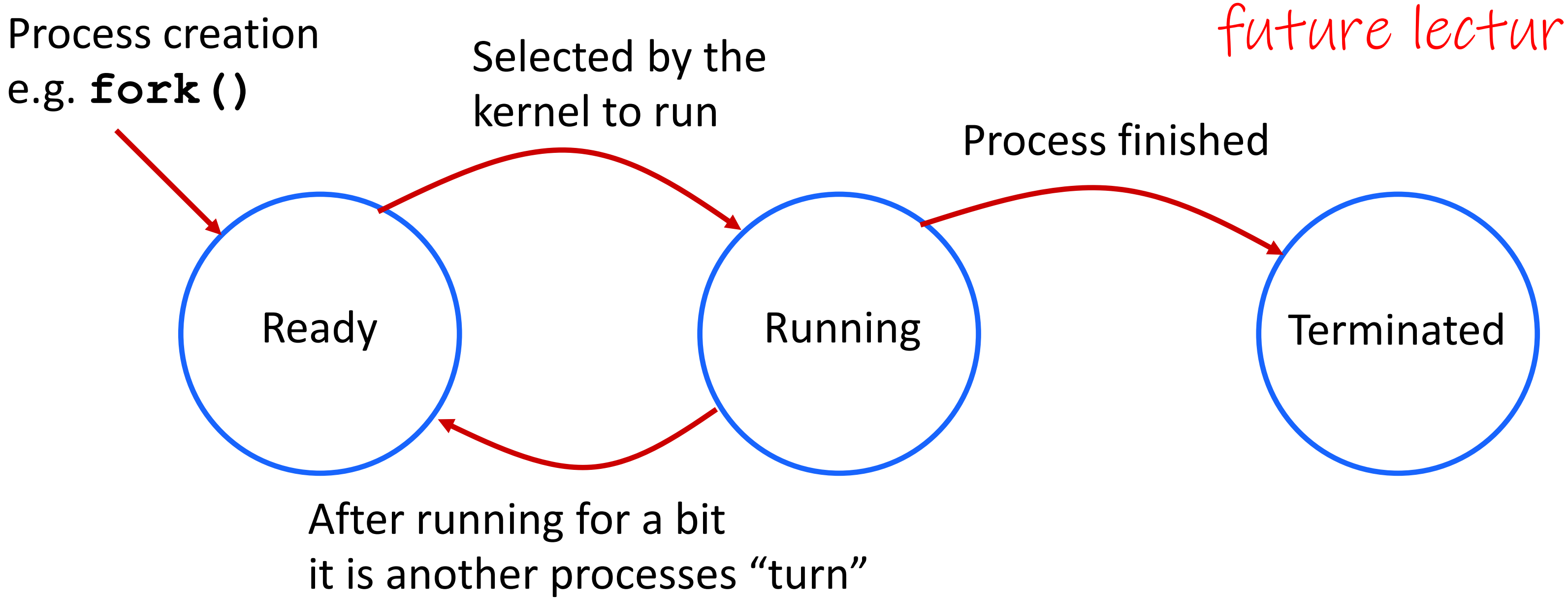
- Running
  - Process is currently executing
- Ready
  - Process is waiting to be executed and will eventually be *scheduled* (i.e., chosen to execute) by the kernel

Scheduler to be covered in a later lecture

- Terminated
  - Process is stopped permanently

Process State Lifetime (incomplete)

More states in future lectures



Process creation e.g. **`fork()`**

Selected by the kernel to run

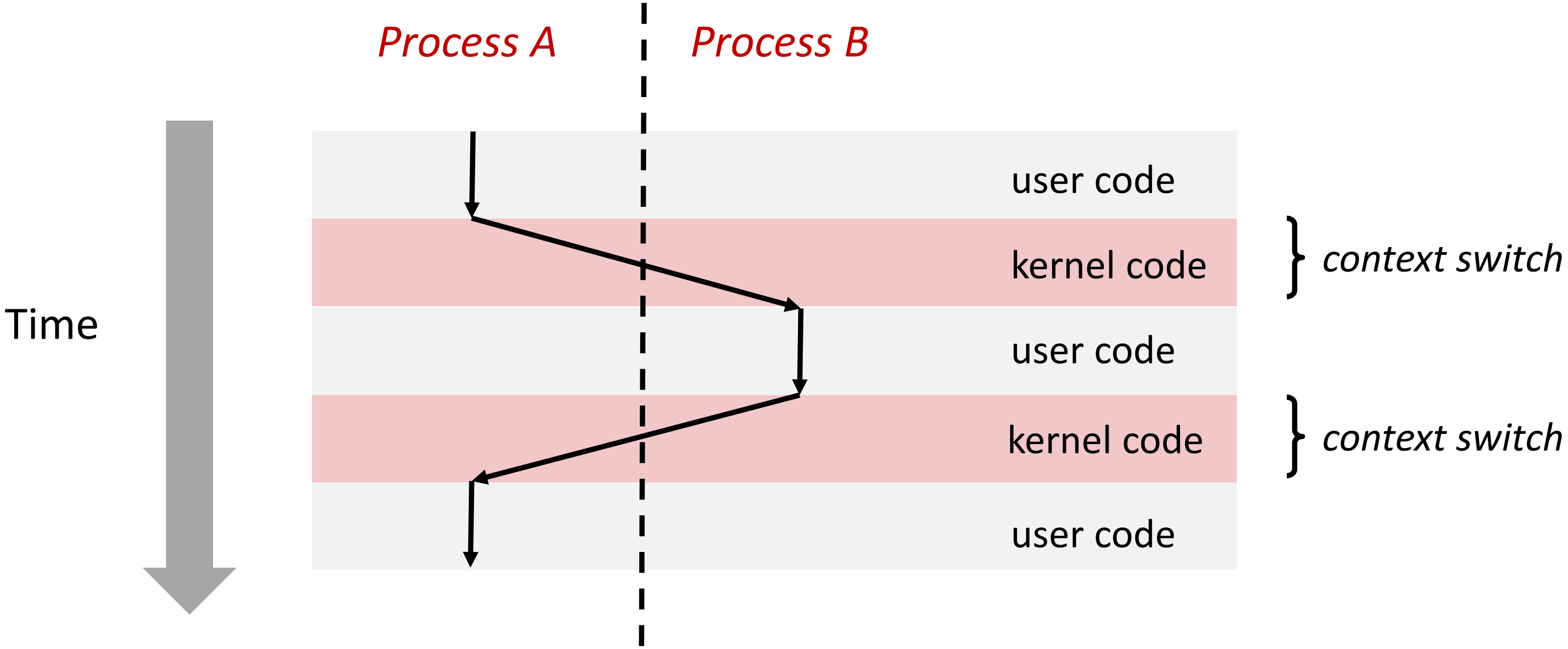
Process finished

After running for a bit it is another processes "turn"

Processes can be "interrupted" to stop running. Through something like a hardware timer interrupt

Context Switching

- Processes are managed by a shared chunk of memory-resident OS code called the *kernel*
  - Important: the kernel is not a separate process, but rather runs as part of some existing process.
- Control flow passes from one process to another via a *context switch*



OS: The Scheduler

- When switching between processes, the OS will run some kernel code called the “Scheduler”
- The scheduler runs when a process:
  - starts (“arrives to be scheduled”),
  - Finishes
  - Blocks (e.g., waiting on something, usually some form of I/O)
  - Has run for a certain amount of time

- It is responsible for scheduling processes
  - Choosing which one to run
  - Deciding how long to run it

Scheduler Considerations

- The scheduler has a scheduling algorithm to decide what runs next.
- Algorithms are designed to consider many factors:
  - Fairness: Every program gets to run
  - Liveness: That “something” will eventually happen
  - Throughput: Number of “tasks” completed over an interval of time
  - Wait time: Average time a “task” is “alive” but not running
  - A lot more...

- More on this later. **For now: think of scheduling as non-deterministic**, details handled by the OS.

fork() example

```
cout << "Hello!" << endl;
pid_t fork_ret = fork();
int x;

if (fork_ret == 0) {
  x = 1234;
} else {
  x = 5678;
}
cout << x << endl;
```

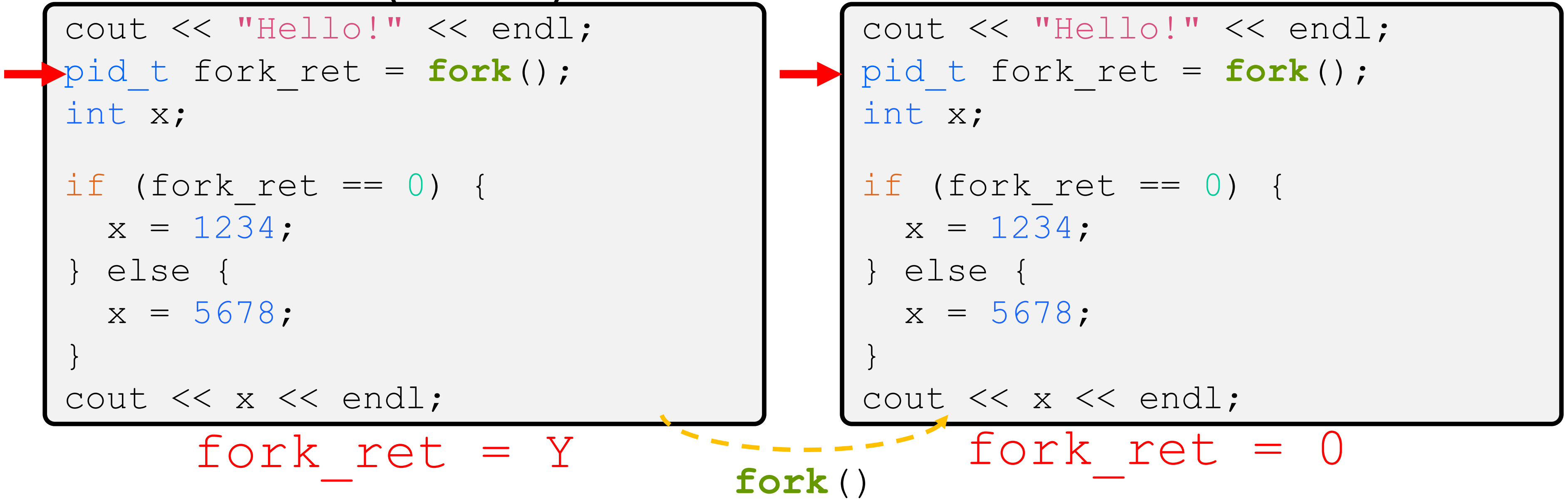
Always prints "Hello"

fork() example

```
cout << "Hello!" << endl;
pid_t fork_ret = fork();
int x;

if (fork_ret == 0) {
  x = 1234;
} else {
  x = 5678;
}
cout << x << endl;
```

Always prints "Hello"

`fork()` example



Parent Process (PID = X)

```
cout << "Hello!" << endl;
pid_t fork_ret = fork();
int x;

if (fork_ret == 0) {
  x = 1234;
} else {
  x = 5678;
}
cout << x << endl;
```

fork_ret = Y

Child Process (PID = Y)

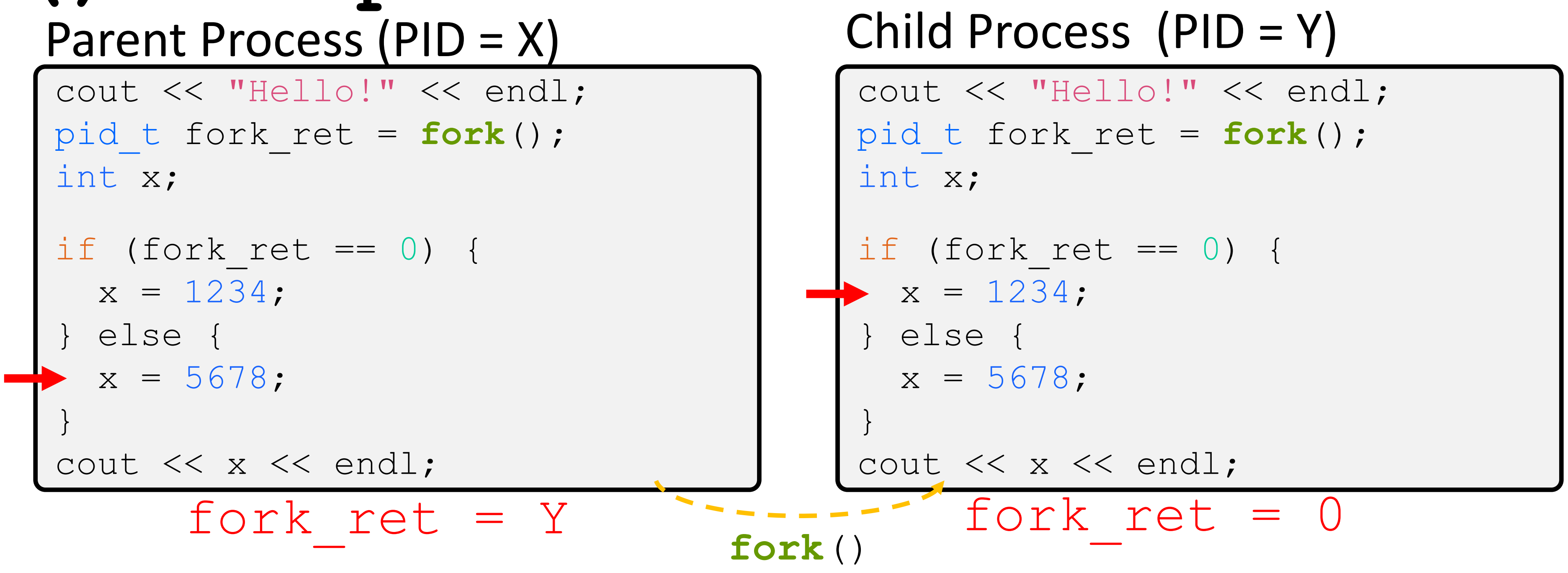
```
cout << "Hello!" << endl;
pid_t fork_ret = fork();
int x;

if (fork_ret == 0) {
  x = 1234;
} else {
  x = 5678;
}
cout << x << endl;
```

fork_ret = 0

Always prints "Hello"

Does NOT print "Hello"

`fork()` example



Parent Process (PID = X)

```
cout << "Hello!" << endl;
pid_t fork_ret = fork();
int x;

if (fork_ret == 0) {
  x = 1234;
} else {
  x = 5678;
}
cout << x << endl;
```

fork_ret = Y

Child Process (PID = Y)

```
cout << "Hello!" << endl;
pid_t fork_ret = fork();
int x;

if (fork_ret == 0) {
  x = 1234;
} else {
  x = 5678;
}
cout << x << endl;
```

fork_ret = 0

Always prints "Hello"

Always prints "5678"

Always prints "1234"

Exiting a Process

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

- Causes the current process to exit normally
- Automatically called by `main()` when main returns
- Exits with a return status (e.g. EXIT_SUCCESS or EXIT_FAILURE)
  - This is the same int returned by `main()`
- The exit status is accessible by the parent process with `wait()` or `waitpid()`.

Poll Everywhere

pollev.com/tqm

```
int global_num = 1;

void function() {
  global_num++;
  cout << global_num << endl;
}

int main() {
  pid_t id = fork();

  if (id == 0) {
    function();
    id = fork();
    if (id == 0) {
      function();
    }
    return EXIT_SUCCESS;
  }

  global_num += 2;
  cout << global_num << endl;
  return EXIT_SUCCESS;
}
```

- How many numbers are printed? What number(s) get printed from each process?

Poll Everywhere

pollev.com/tqm

- How many times is ":)" printed?

```
int main(int argc, char* argv[]) {
  for (int i = 0; i < 4; i++) {
    fork();
  }

  cout << ":)\n"; // "\n" is similar to endl
  return EXIT_SUCCESS;
}
```

Poll Everywhere

pollev.com/tqm

- Are the following outputs possible?

```
pid_t fork_ret = fork();
if (fork_ret == 0) {
  fork_ret = fork();
  if (fork_ret == 0) {
    cout << "Hi 3!" << endl;
  } else {
    cout << "Hi 2!" << endl;
  }
} else {
  cout << "Hi 1!" << endl;
}
cout << "Bye" << endl;
```

Hint 1: there are three processes

Hint 2: Each prints out twice
"Hi" and "Bye"

Sequence 1:	Sequence 2:
Hi 1	**Hi 3**
Bye	**Hi 1**
Hi 2	**Hi 2**
Bye	**Bye**
Bye	**Bye**
Hi 3	**Bye**

A.	No	No
B.	No	Yes
C.	Yes	No
D.	Yes	Yes
E.	We're lost…	

Poll Everywhere

pollev.com/tqm

- Are the following outputs possible?

```
pid_t fork_ret = fork();
if (fork_ret == 0) {
  fork_ret = fork();
  if (fork_ret == 0) {
    cout << "Hi 3!" << endl;
  } else {
    cout << "Hi 2!" << endl;
  }
} else {
  cout << "Hi 1!" << endl;
}
cout << "Bye" << endl;
```

Hint 1: there are three processes

Hint 2: Each prints out twice
"Hi" and "Bye"

Hint 3: Events within a single process
are "ordered normally"

Sequence 1:	Sequence 2:
Hi 1	**Hi 3**
Bye	**Hi 1**
Hi 2	**Hi 2**
Bye	**Bye**
Bye	**Bye**
Hi 3	**Bye**

Hint #2
"Hi 3"
must be
before a "Bye"

A.	[✓ No]	No
B.	[✓ No]	Yes
C.	Yes	No
D.	Yes	Yes
E.	We're lost…	

Poll Everywhere

pollev.com/tqm

- Are the following outputs possible?

```
pid_t fork_ret = fork();
if (fork_ret == 0) {
  fork_ret = fork();
  if (fork_ret == 0) {
    cout << "Hi 3!" << endl;
  } else {
    cout << "Hi 2!" << endl;
  }
} else {
  cout << "Hi 1!" << endl;
}
cout << "Bye" << endl;
```

Hint 1: there are three processes

Hint 2: Each prints out twice
"Hi" and "Bye"

Hint 3: Events within a single process
are "ordered normally"

Sequence 1:	Sequence 2:
Hi 1	**Hi 3**
Bye	**Hi 1**
Hi 2	**Hi 2**
Bye	**Bye**
Bye	**Bye**
Hi 3	**Bye**

OK
Each "hi"
comes
before a
"bye"

A.	No	No
[✓ B.]	No	Yes
C.	Yes	No
D.	Yes	Yes
E.	We're lost…	

Order
across
processes
not
guaranteed

Processes & Fork Summary

- Processes are instances of programs that:
  - Each have their own independent address space

  - Each process is scheduled by the OS
    - Without using some functions we have not talked about (yet), there is no way to guarantee the order processes are executed

  - Processes are created by fork() system call
    - Only difference between processes is their process id and the return value from fork() each process gets

Lecture Outline

- The OS
- Processes & fork()
- **execvp()**

execvp()

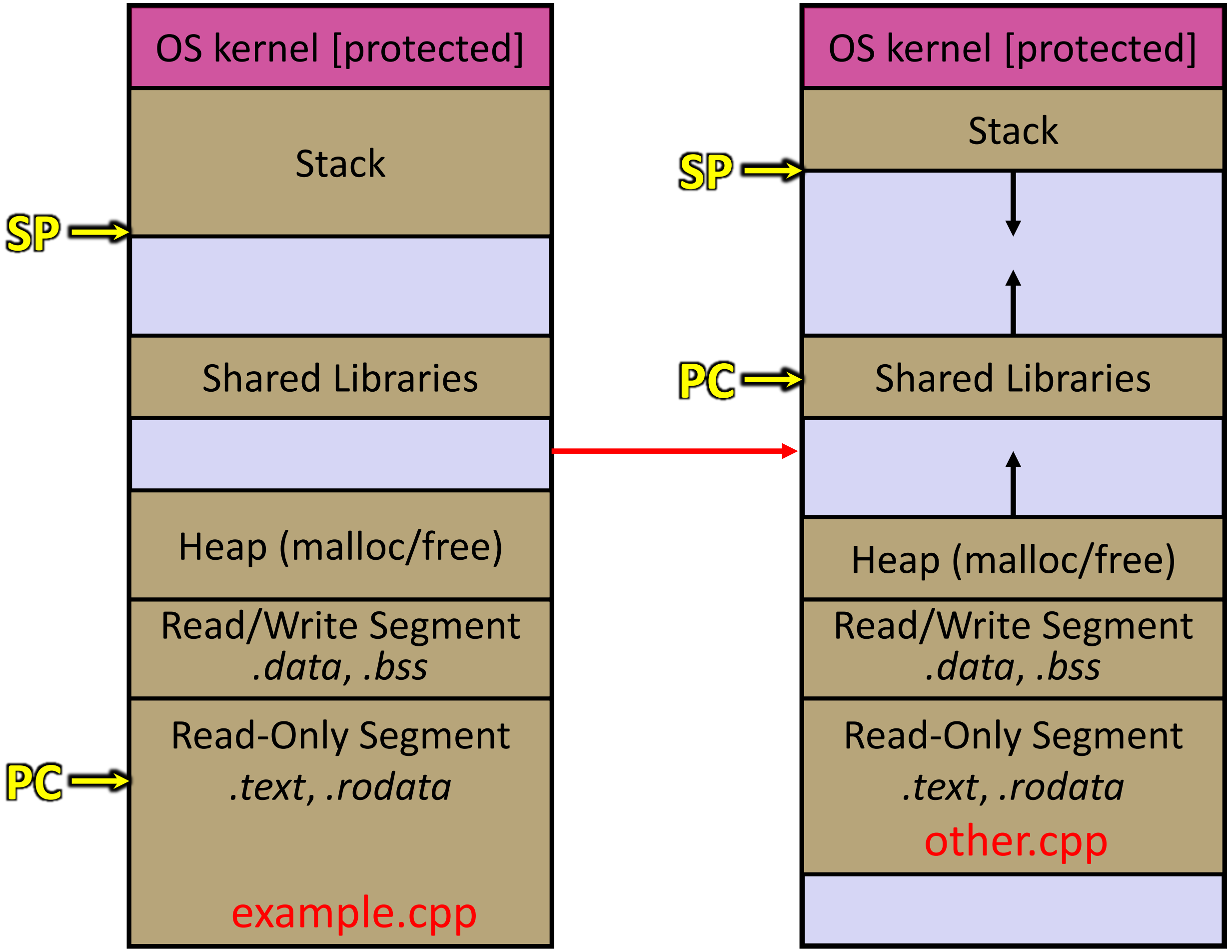
- execvp

```
int execvp(const char *file,
                char* const argv[]);
```

- Duplicates the action of the shell (terminal) in terms of finding the command/program to run
- Argv is an array of **`char*`**, the same kind of argv that is passed to `main()` in a C/C++ program
  - **`argv[0]`** MUST have the same contents as the file parameter
  - **`argv`** must have NULL/nullptr as the last entry of the array
- Returns -1 on error. Does NOT return on success

Exec Visualization

- Exec takes a process and discards or "resets" most of it



NOTE that the following DO change
- The stack
- The heap
- Globals
- Loaded code
- Registers

NOTE that the following do NOT change
- Process ID
- Open files
- The kernel

Exec Demo

- See `exec_example.cpp`
  - Brief code demo to see how exec works
  - What happens when we call exec?
  - What happens if we open some files before exec?
  - What happens if we replace stdout with a file?

- NOTE: When a process exits, then it will close all of its open files by default

Aside: Exiting a Process

- `void exit(int status);`
  - Causes the current process to exit normally
  - Automatically called by `main()` when main returns
  - Exits with a return status (e.g. EXIT_SUCCESS or EXIT_FAILURE)
    - This is the same int returned by `main()`
  - The exit status is accessible by the parent process with `wait()` or `waitpid()`. (more on these functions next lecture)

Exec Demo

- See `exec_example.cpp`
  - Brief code demo to see how exec works
  - What happens when we call exec?
  - What happens to allocated memory when we call exec?

Poll Everywhere

pollev.com/tqm

- In each of these, how often is ":)" printed? Assume functions don't fail

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

  pid_t pid = fork();
  if (pid == 0) {
    // we are the child
    char* argv[] = {"echo",
                    "hello",
                    NULL};
    execvp(argv[0], argv);
  }

  cout << ":)" << endl;

  return EXIT_SUCCESS;
}
```

```
int main(int argc, char* argv[]) {
  char* envp[] = { NULL };

  pid_t pid = fork();
  if (pid == 0) {
    // we are the child
    return EXIT_SUCCESS;
  }

  cout << ":)" << endl;

  return EXIT_SUCCESS;
}
```

That’s it for now!

- More next lecture ☺