The OS & Processes

Computer Systems Programming, Spring 2023

Instructor: Travis McGaha

TAs:

Kevin Bernat
Mati Davis
Chandravaran Kunjeti
Shufan Liu

Jialin Cai
Donglun He
Heyi Liu
Eddy Yang

Logistics

- HW1 (FileReaders) Due Thursday 2/9 @ 11:59 pm
  - Released, autograder coming out later this week
  - You should have everything you need to complete the assignment
  - Recitation should give helpful practice with writing POSIX code

Lecture Outline

- **Control Flow**
- Exceptions
- Processes
- fork()

Control Flow

- Processors do only one thing:
  - From startup to shutdown, a CPU simply reads and executes (interprets) a sequence of instructions, one at a time
  - This sequence is the CPU's *control flow* (or *flow of control*)

Physical control flow

<startup>
inst_1
inst_2
inst_3
…
inst_n
<shutdown>

Altering the Control Flow

- Up to now: two mechanisms for changing control flow:
  - Jumps and branches
  - Call and return

  React to changes in ***program state***

- Insufficient for a useful system:
  Difficult to react to changes in *system state*
  - Data arrives from a disk or a network adapter
  - Instruction divides by zero
  - User hits Ctrl-C at the keyboard
  - System timer expires

- System needs mechanisms for "exceptional control flow"

Exceptional Control Flow

- Exists at all levels of a computer system
- Low level mechanisms
  - 1. **Exceptions**
    - Change in control flow in response to a system event (i.e., change in system state)
    - Implemented using combination of hardware and OS software

what we will be looking at today

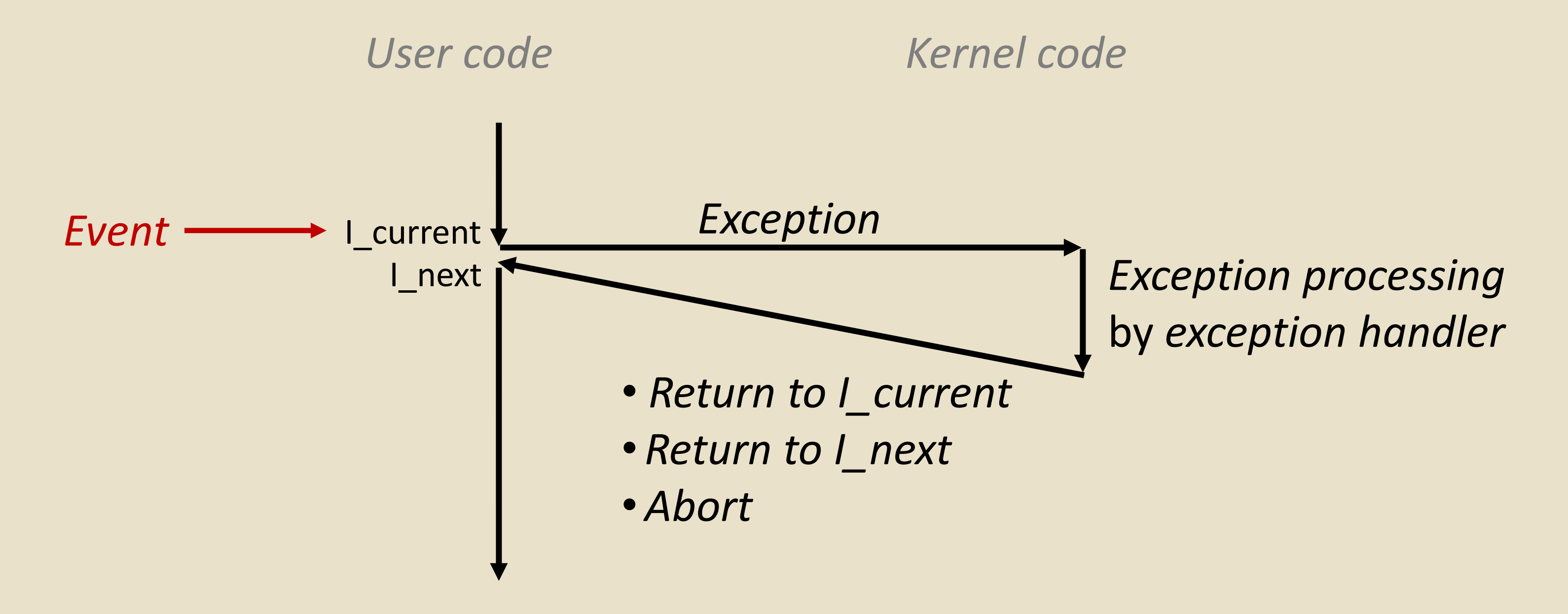
- Higher level mechanisms
  - 2. **Process context switch**
    - Implemented by OS software and hardware timer
  - 3. **Signals**
    - Implemented by OS software

Lecture Outline

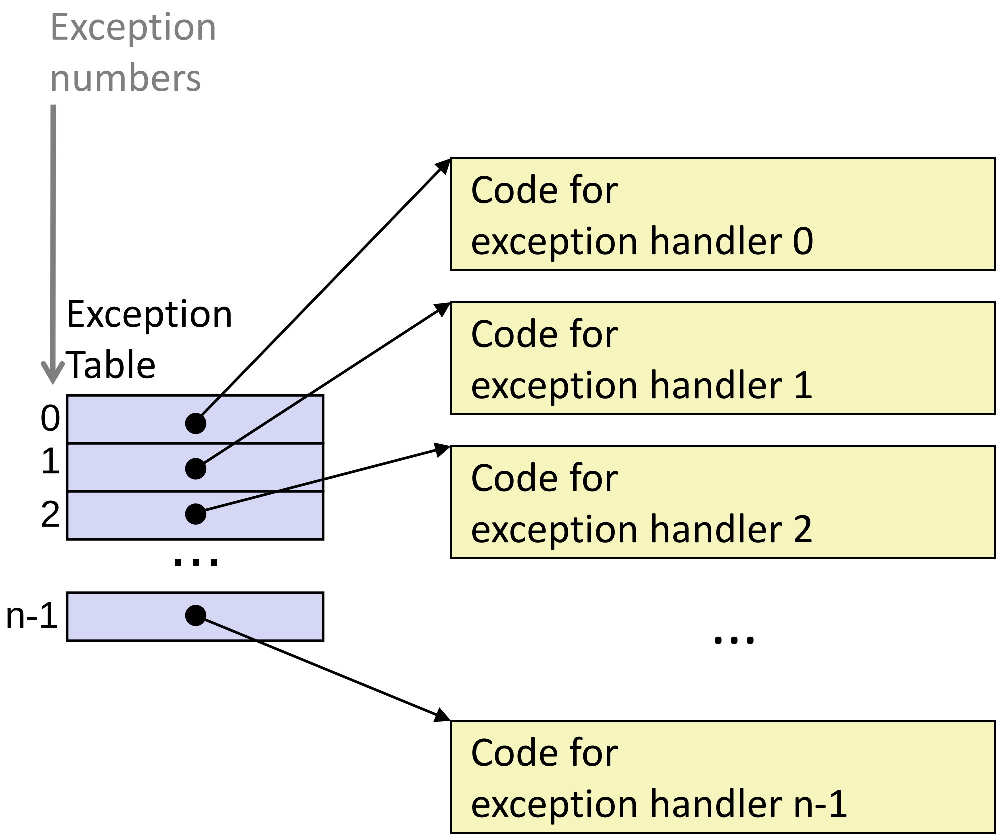
- Control Flow
- **Exceptions**
- Processes
- fork()

Exceptions

- An *exception* is a transfer of control to the OS *kernel* in response to some *event* (i.e., change in processor state)
  - Kernel is the memory-resident part of the OS
  - Examples of events: Divide by 0, arithmetic overflow, page fault, I/O request completes, typing Ctrl-C



Exception Tables



- Each type of event has a unique exception number k
- k = index into exception table (a.k.a. interrupt vector)
- Handler k is called each time exception k occurs

Asynchronous Exceptions (Interrupts)

- Caused by events external to the processor
  - Indicated by setting the processor's *interrupt pin*
  - Handler returns to "next" instruction

- Examples:
  - Timer interrupt
    - Every few ms, an external timer chip triggers an interrupt
    - Used by the kernel to take back control from user programs
  - I/O interrupt from external device
    - Hitting Ctrl-C at the keyboard
    - Arrival of a packet from a network
    - Arrival of data from a disk

Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction:
  - ***Traps***
    - Intentional
    - Examples: ***system calls***, breakpoint traps, special instructions
    - Returns control to “next” instruction
  - ***Faults***
    - Unintentional but possibly recoverable
    - Examples: page faults (recoverable), protection faults (unrecoverable), floating point exceptions
    - Either re-executes faulting (“current”) instruction or aborts
  - ***Aborts***
    - Unintentional and unrecoverable
    - Examples: illegal instruction, parity error, machine check
    - Aborts current program

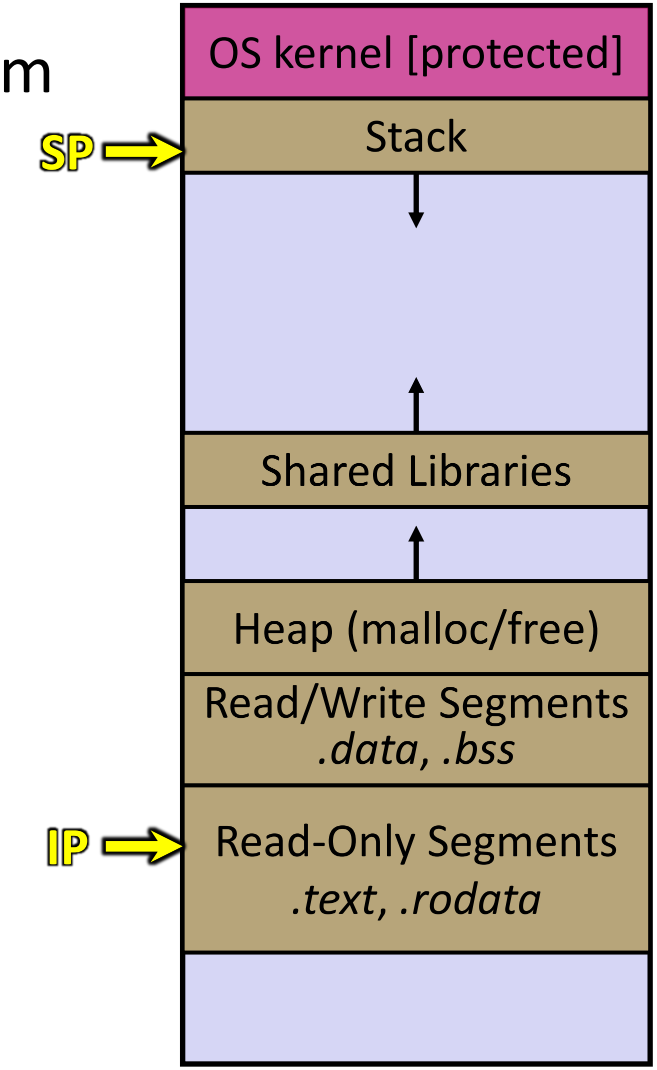
Lecture Outline

- Control Flow
- Exceptions
- **Processes**
- fork()

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


OS kernel [protected]

Stack

Shared Libraries

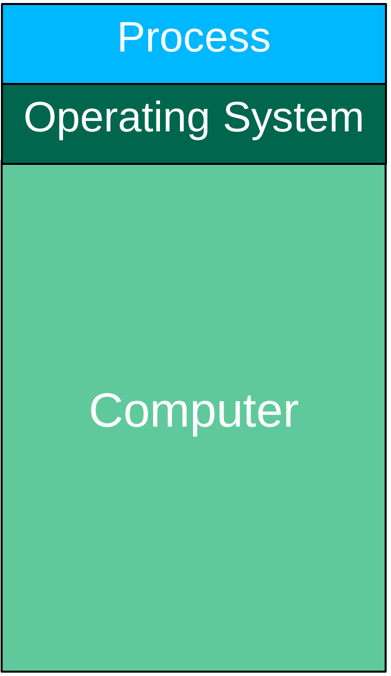
Heap (malloc/free)

Read/Write Segments
.data, .bss

Read-Only Segments
.text, .rodata

Computers as we know them now

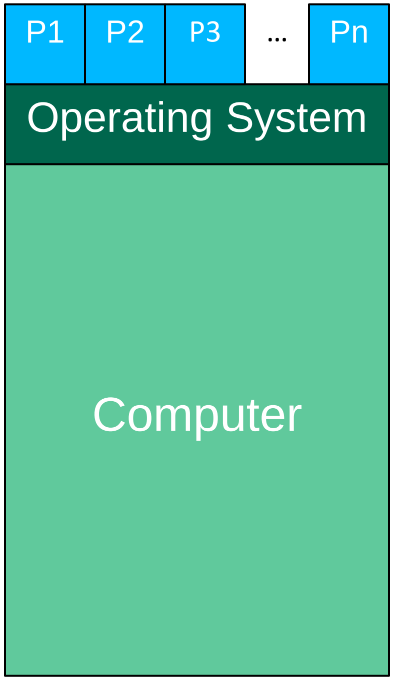
- In CIT 5930, you learned about hardware, transistors, CMOS, gates, etc.
- Once we got to programming, our computer looks something like:



What is missing/wrong with this?

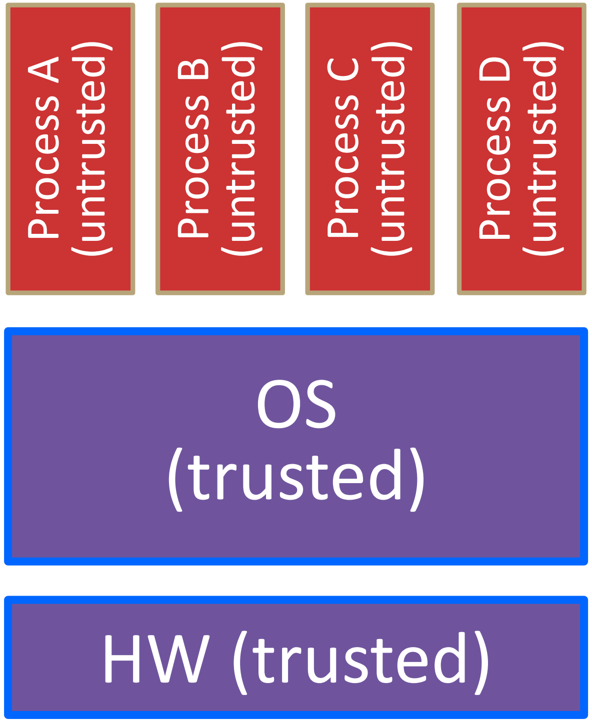
Multiple Processes

- 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

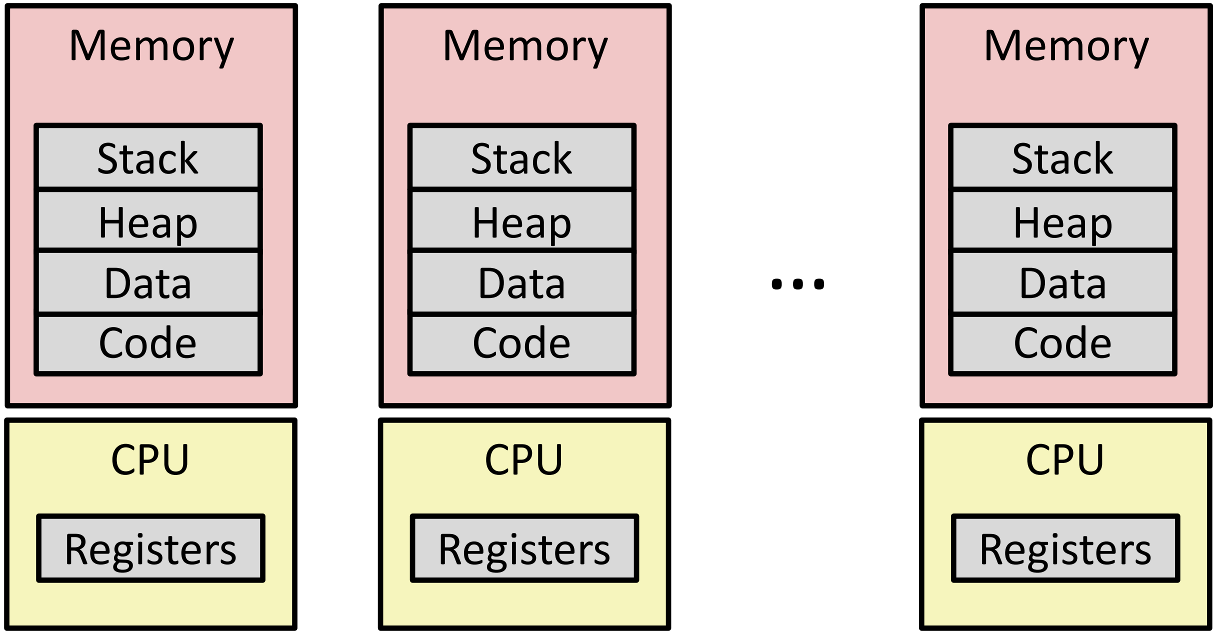


OS: Protection System

- 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

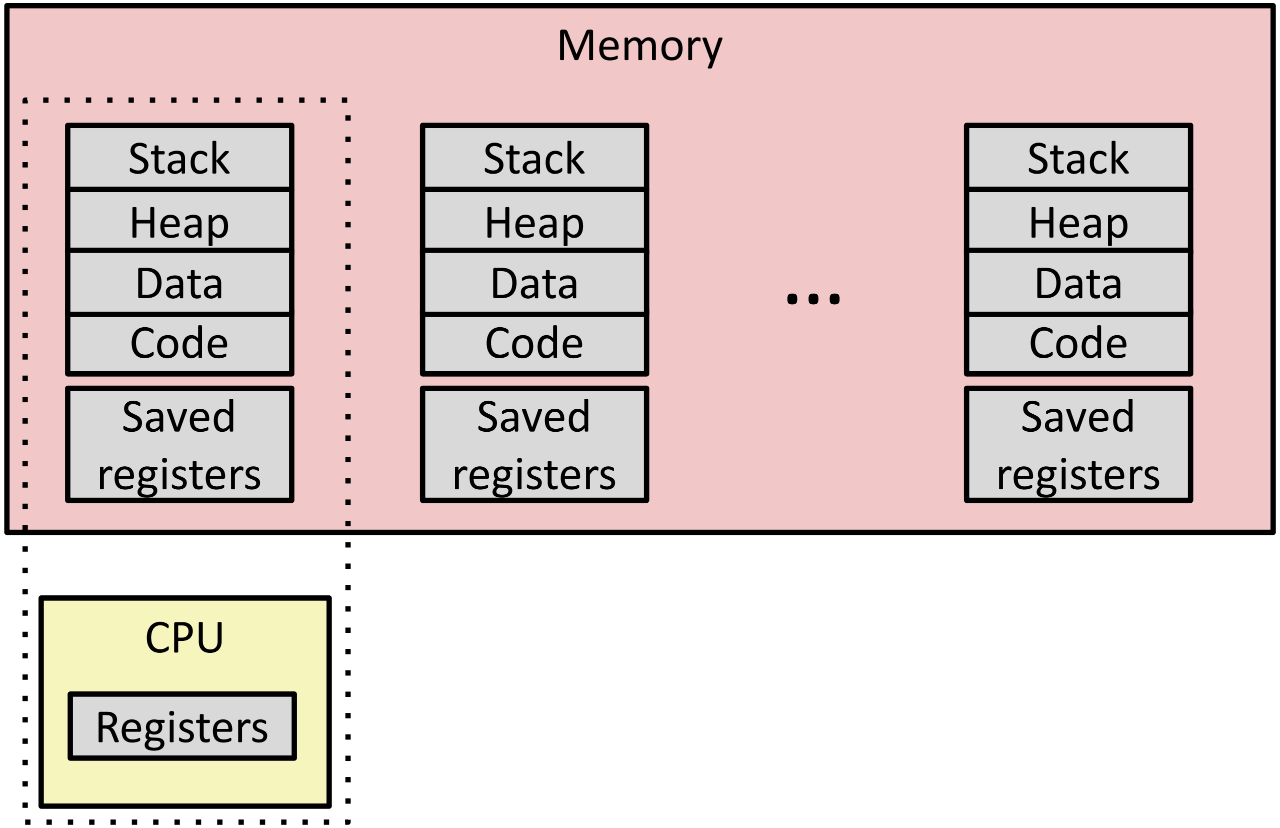


Multiprocessing: The Illusion



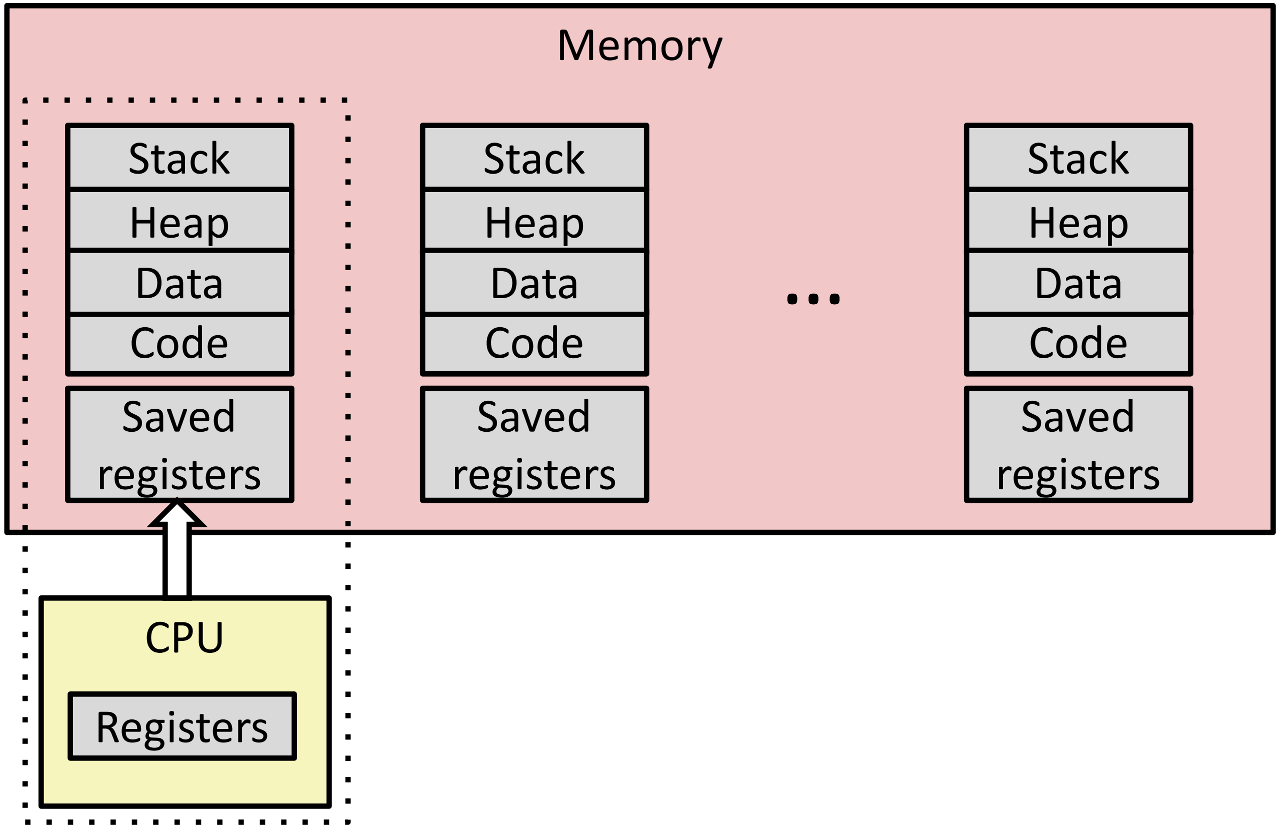
- Computer runs many processes simultaneously
  - Applications for one or more users
    - Web browsers, email clients, editors, …
  - Background tasks
    - Monitoring network & I/O devices

Multiprocessing: The (Traditional) Reality



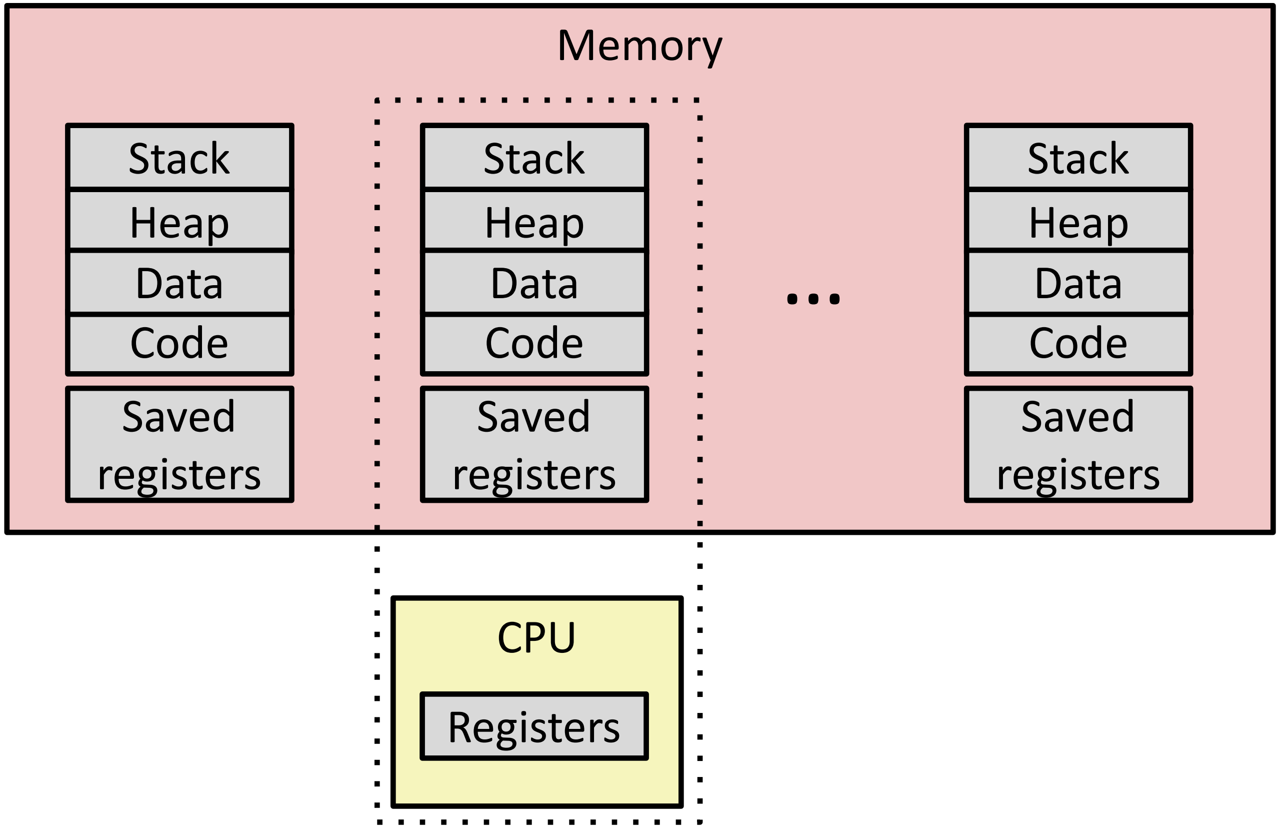
- Single processor executes multiple processes concurrently
  - Process executions interleaved (multitasking)
  - Address spaces managed by virtual memory system (later in course)
  - Register values for nonexecuting processes saved in memory

Multiprocessing: The (Traditional) Reality



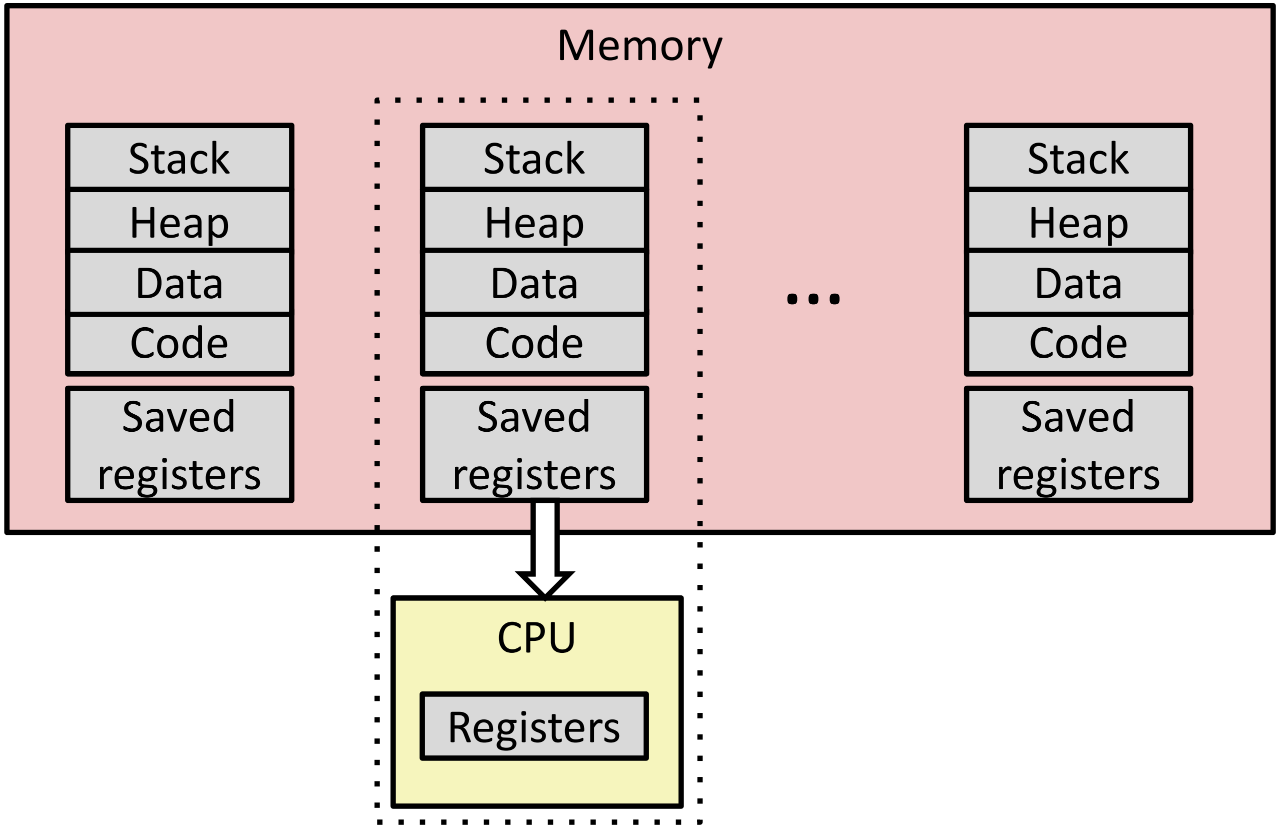
1. Save current registers in memory

Multiprocessing: The (Traditional) Reality



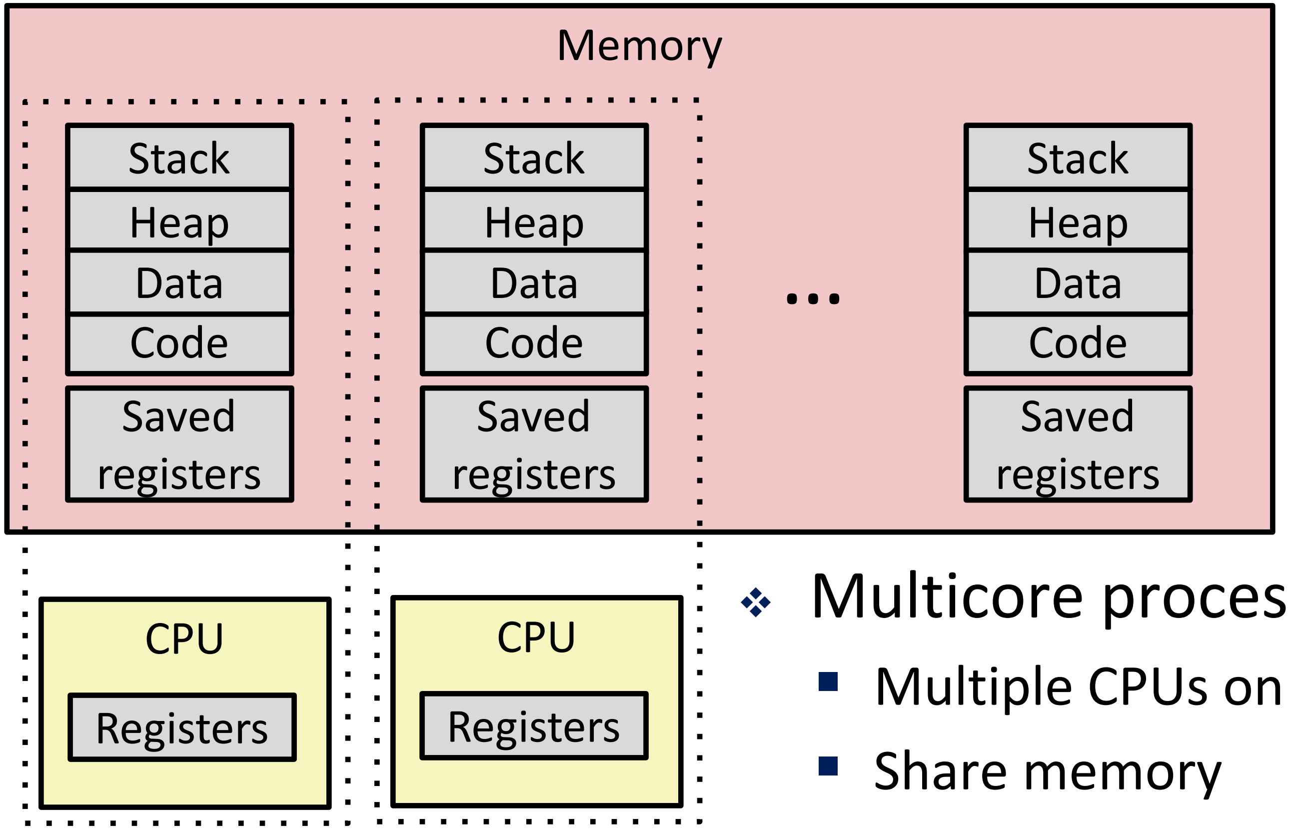
1. Save current registers in memory
2. Schedule next process for execution

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



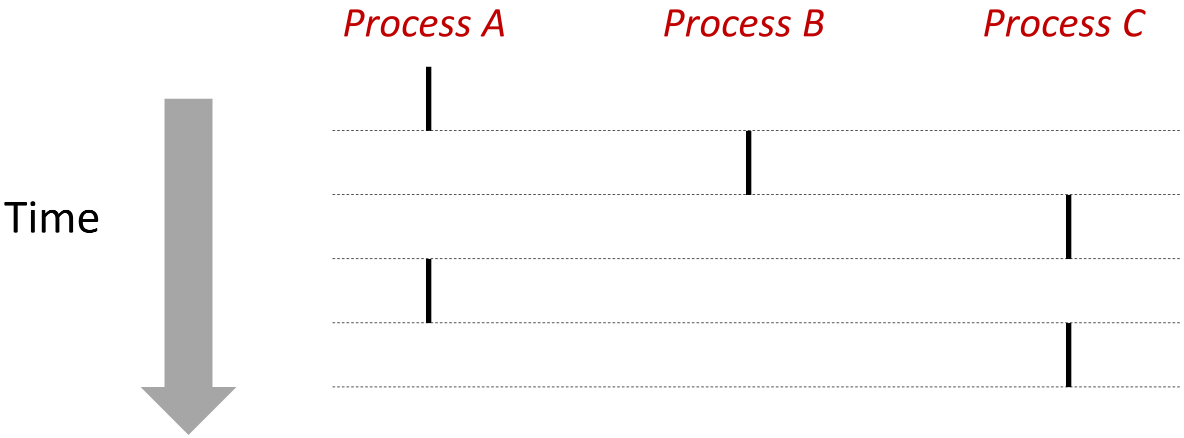
- Multicore processors
  - Multiple CPUs on single chip
  - Share memory
  - Each can execute a separate process
    - Scheduling of processors onto cores done by kernel
  - This is called "Parallelism"

Concurrent Processes

Assuming ONE CPU/Core

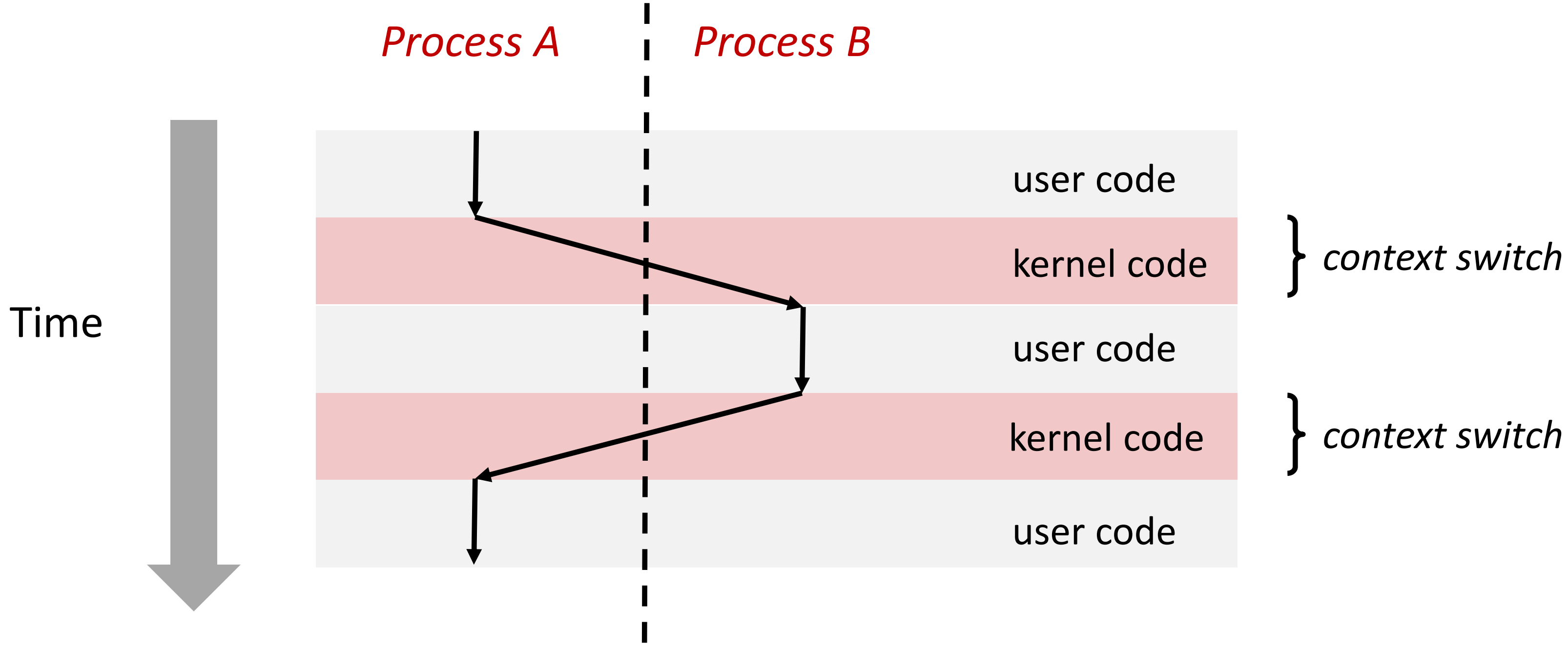
- Each process is a logical control flow.
- Two processes *run concurrently* (*are concurrent)* if their flows overlap in time
- Otherwise, they are *sequential*
- Examples (running on single core):
  - Concurrent: A & B, A & C
  - Sequential: B & C

Note how there is no instant where 2 processes are running



Context Switching

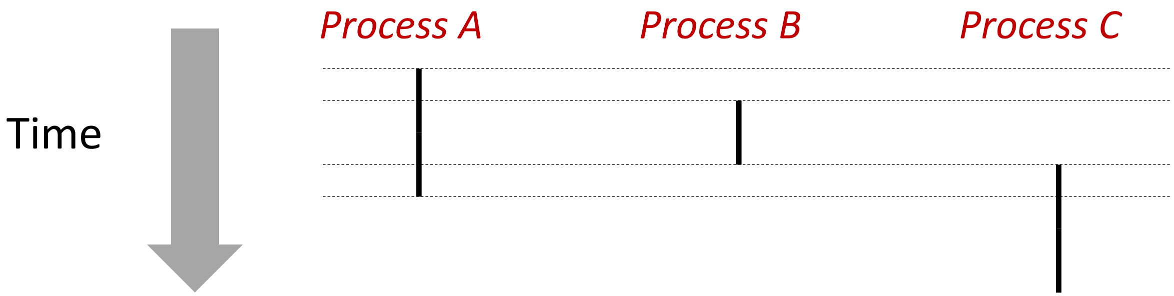
- Processes are managed by a shared chunk of memory-resident OS code called the *kernel*
  - Important: the kernel is not a separate process, but rather runs as part of some existing process.
- Control flow passes from one process to another via a *context switch*



User View of Concurrent Processes

- Control flows for concurrent processes are physically disjoint in time

- However, we can think of concurrent processes as running in parallel with each other

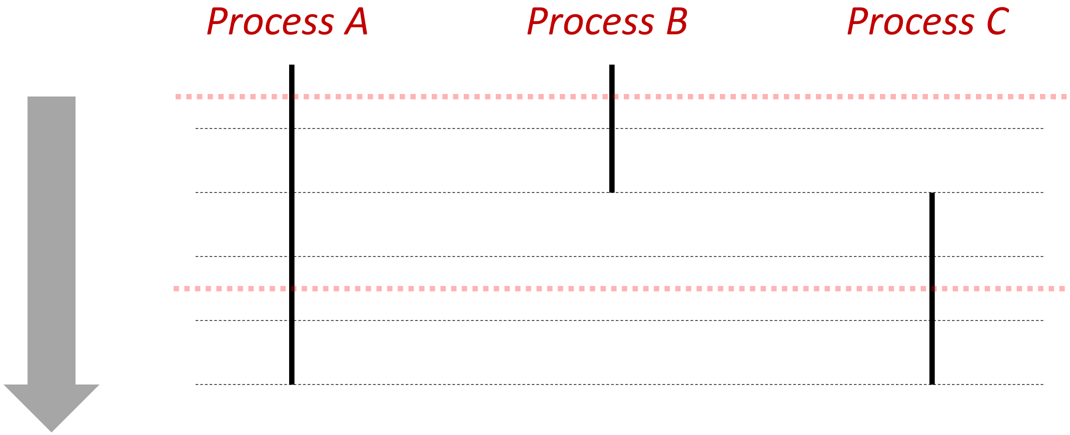


- Above is what a User may **think** is going on. At any moment in time only **one** process has its instructions being executed at a time (ignoring multiple cores).

Parallel Processes

Assuming more than one CPU/Core

- Each process is a logical control flow.
- Two processes *run parallel* if their flows overlap at a specific point in time. (Multiple instructions are performed on the CPU at the same time
- Examples (running on dual core): Dual = 2
  - Parallel: A & B, A & C
  - Sequential: B & C



Note How there is overlap at specific points of time

Lecture Outline

- Control Flow
- Exceptions
- Processes
- fork()

Creating and Terminating Processes

From a programmer's perspective, we can think of a process as being in one of three states

- Running
  - Process is either executing, or waiting to be executed and will eventually be *scheduled* (i.e., chosen to execute) by the kernel
- Stopped
  - Process execution is *suspended* and will not be scheduled until further notice (next lecture when we study signals)
- Terminated
  - Process is stopped permanently

Terminating Processes

- Process becomes terminated for one of three reasons:
  - Receiving a signal whose default action is to terminate (next lecture)
  - Returning from the `main` routine
  - Calling the `exit` function

- `void exit(int status)`
  - Terminates with an *exit status* of `status`
  - Convention: normal return status is 0, nonzero on error
  - Another way to explicitly set the exit status is to return an integer value from the main routine

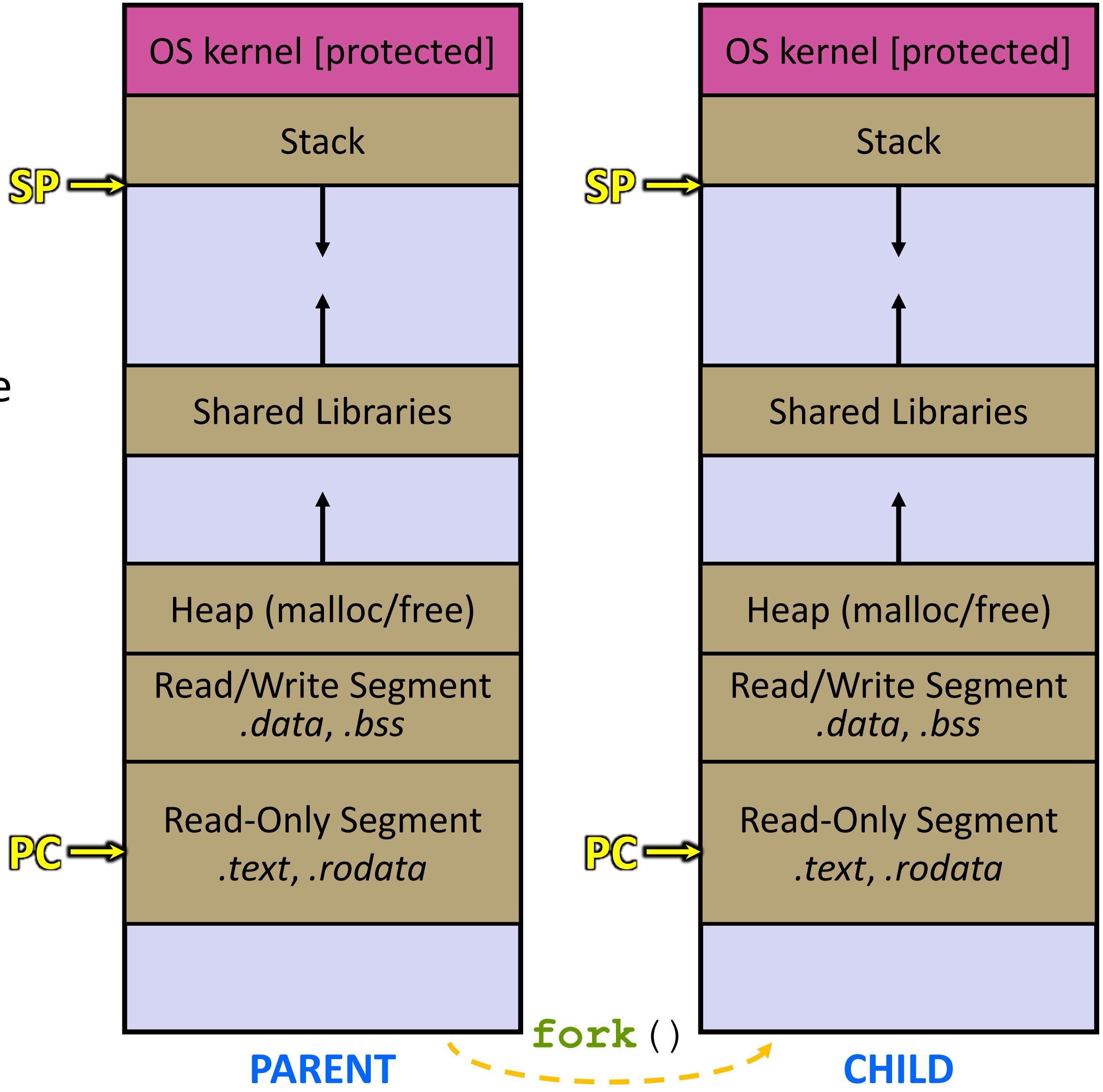
- `exit` is called once but never returns.

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

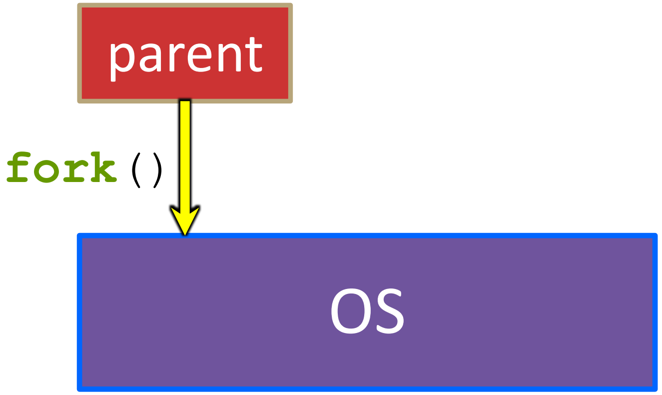
`fork()` and Address Spaces

- 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.



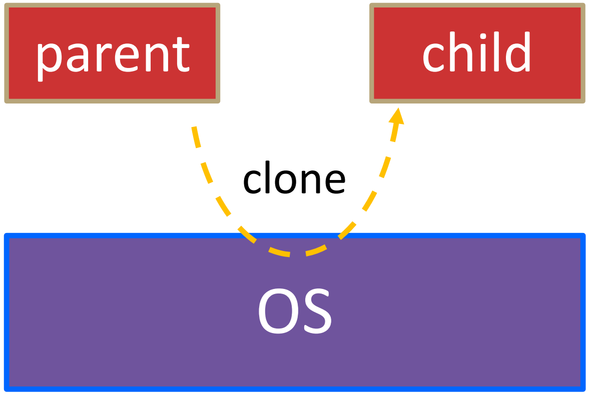
`fork()`

- `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



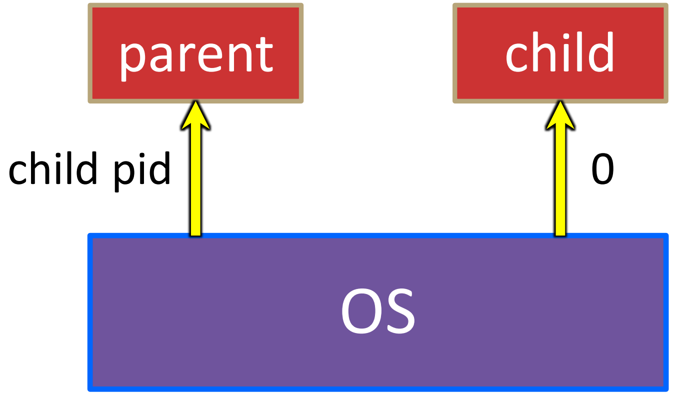
`fork()`

- **`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!\n";
```

OS: The Scheduler

- When switching between processes, the OS will 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 other 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.

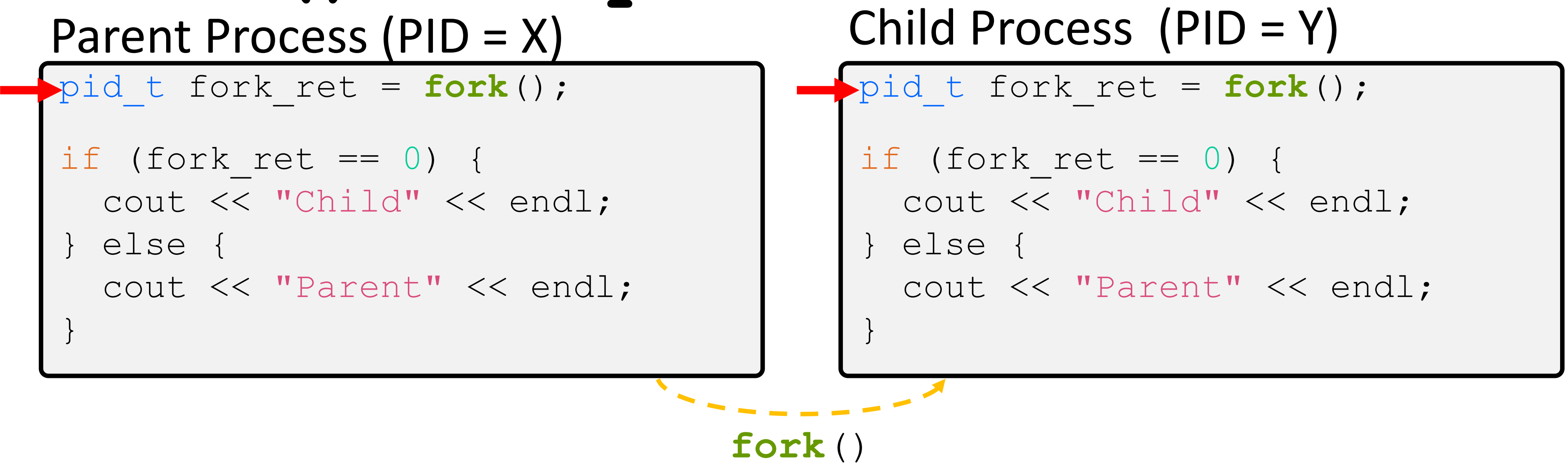
"simple" fork() example

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

`fork()` example

```
pid_t fork_ret = fork();

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

`fork()` example



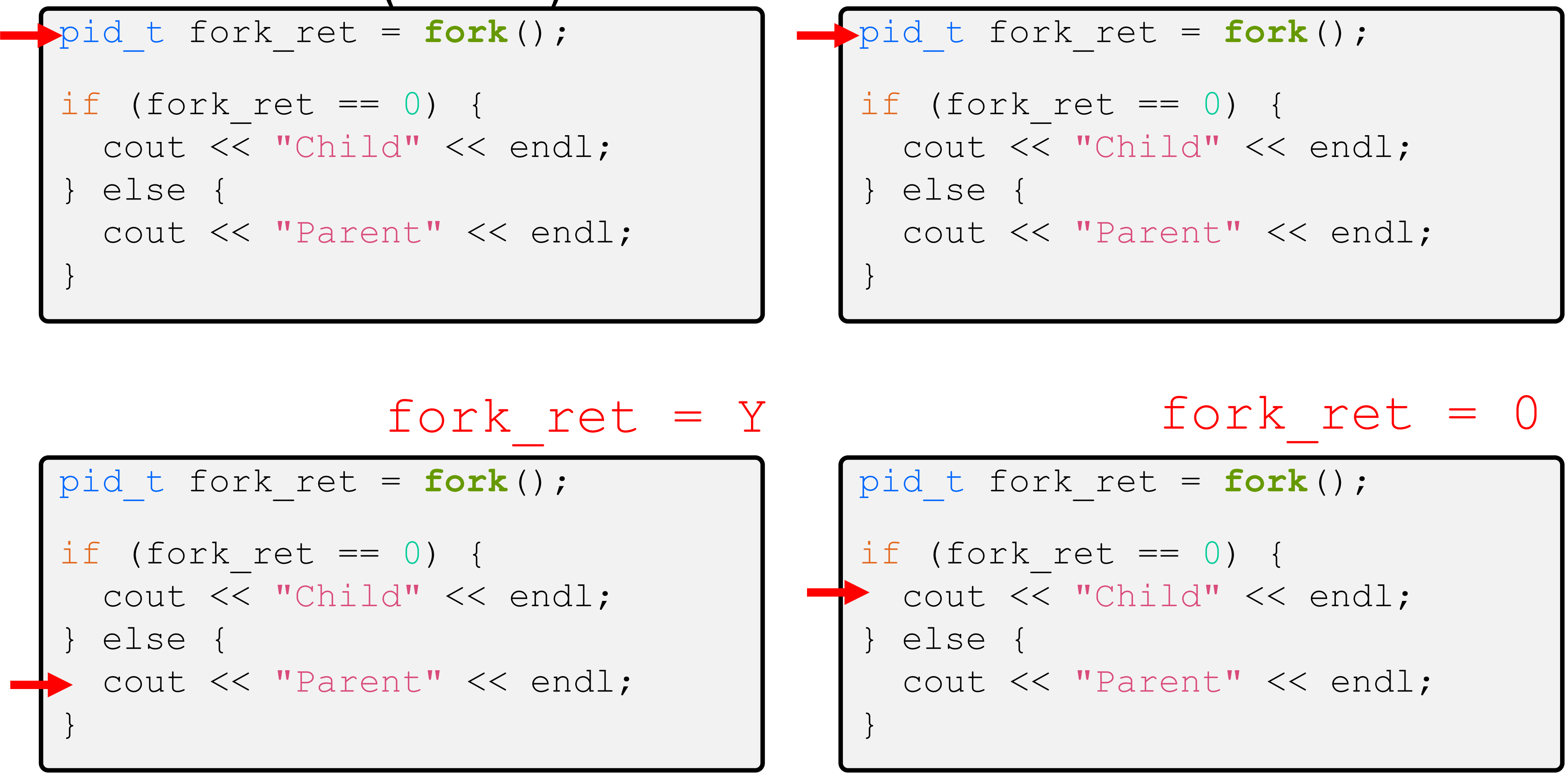
Parent Process (PID = X)

```
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() 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;
}
```

Prints "Parent"

fork_ret = 0

```
pid_t fork_ret = fork();

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

Prints "Child"

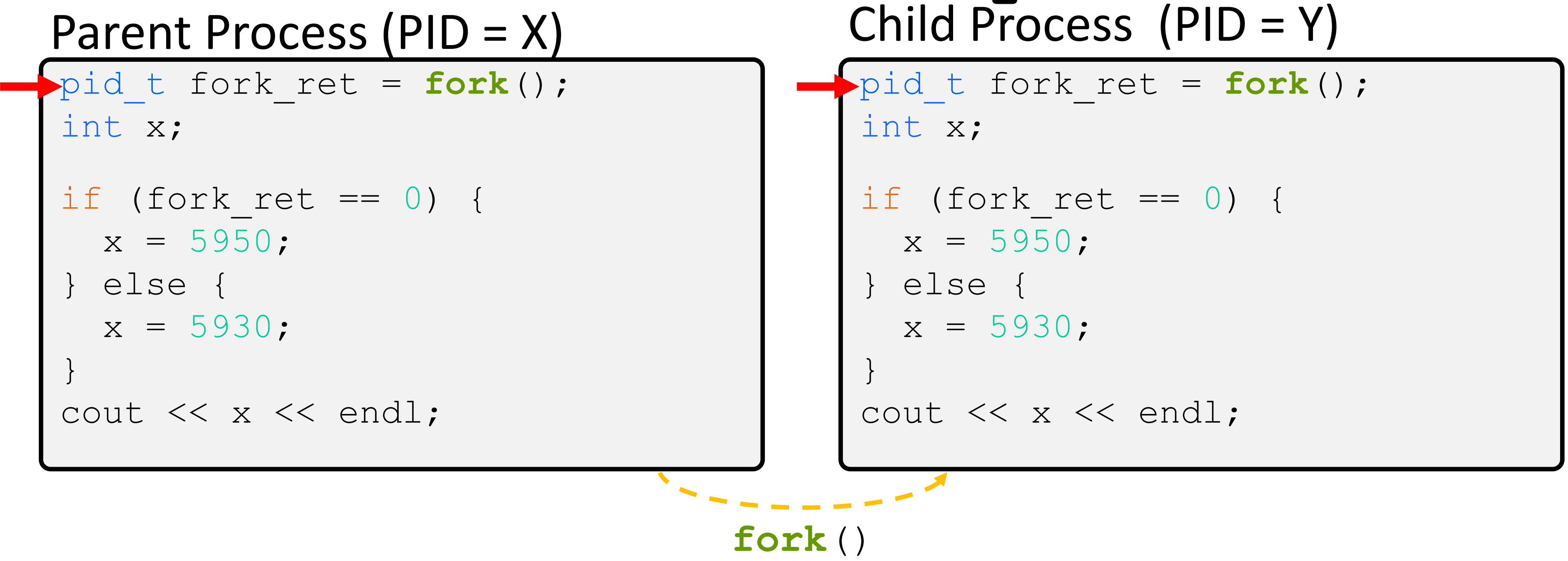
Which prints first?

Non-deterministic

Another fork() example

```
pid_t fork_ret = fork();
int x;

if (fork_ret == 0) {
  x = 5950;
} else {
  x = 5930;
}
cout << x << endl;
```

Another fork() example



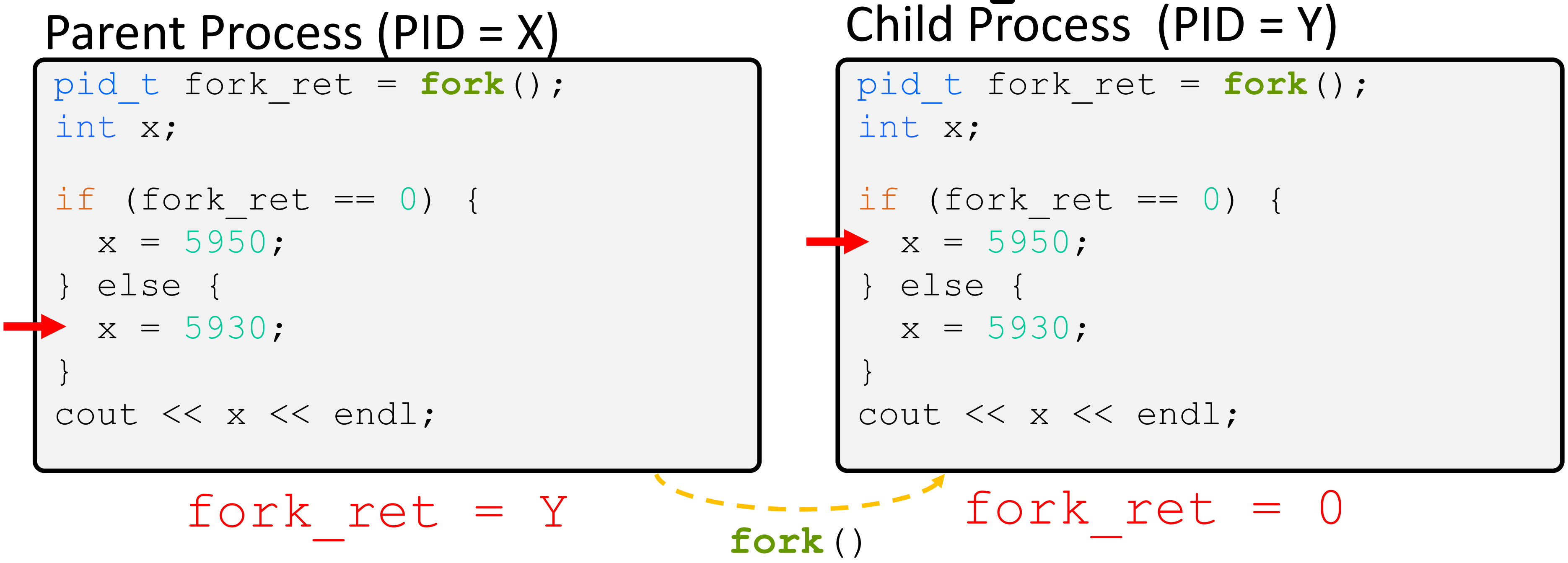
Parent Process (PID = X)

```
pid_t fork_ret = fork();
int x;

if (fork_ret == 0) {
  x = 5950;
} else {
  x = 5930;
}
cout << x << endl;
```

Child Process (PID = Y)

```
pid_t fork_ret = fork();
int x;

if (fork_ret == 0) {
  x = 5950;
} else {
  x = 5930;
}
cout << x << endl;
```

Another `fork()` example



Parent Process (PID = X)

```
pid_t fork_ret = fork();
int x;

if (fork_ret == 0) {
  x = 5950;
} else {
  x = 5930;
}
cout << x << endl;
```

Child Process (PID = Y)

```
pid_t fork_ret = fork();
int x;

if (fork_ret == 0) {
  x = 5950;
} else {
  x = 5930;
}
cout << x << endl;
```

`fork_ret = Y`

`fork()`

`fork_ret = 0`

Always prints "5930"

Always prints "5950"

Reminder: Processes have their own address space (and thus, copies of their own variables)

Order is still nondeterministic!!