# The Hardware/Software Interface

## Processes I

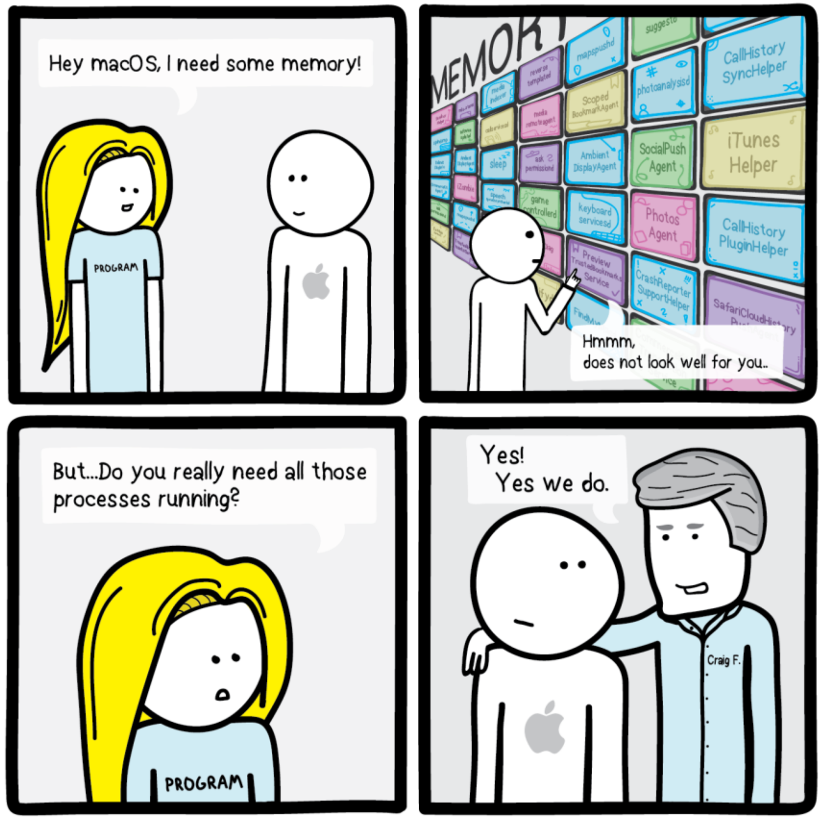
No handout today

**Instructors:**

Amber Hu, Justin Hsia

**Teaching Assistants:**

Anthony Mangus
Grace Zhou
Jiuyang Lyu
Kurt Gu
Mendel Carroll
Naama Amiel
Rose Maresh
Violet Monserate
Divya Ramu
Jessie Sun
Kanishka Singh
Liander Rainbolt
Ming Yan
Pollux Chen
Soham Bhosale




PRETENDS TO BE DRAWING | PTBD.JWELS.BERLIN

https://ptbd.jwels.berlin/comic/21/

# Relevant Course Information

- HW 22 due Friday (11/21)
- HW 23 due Monday (11/24)
- Lab 4 due Friday (11/21)
- Lab 5 due 12/4

- Section on Memory Allocation/Lab 5 tomorrow

- Final review session 12/5 (Fri)
- Final 12/10 (Wed)

# Lecture Outline (1/3)

- **System Control Flow**
- Processes
- Process Management (x86-64 Linux)

# Control Flow

- **So far:** we've seen how the flow of control changes as a *single program* executes
- **Reality:** multiple programs running *concurrently*
  - How does control flow across the many components of the system?
  - In particular, more programs running than CPUs

many applications open at once:
browser, text editor, music, file explorer, etc.

- *Exceptional* control flow is basic mechanism used for:
  - Transferring control between *processes* and OS
  - Handling *I/O* and *virtual memory* within the OS
  - Implementing multi-process apps like shells and web servers
  - Implementing concurrency

# CPU 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*

time ↓

<startup>
$instr_1$ mov
$instr_2$ add
$instr_3$ jmp
…
$instr_n$ call
<shutdown>

# Altering the Control Flow

- Up to now, two ways to change control flow (caused by changes in *program state*):
  - Jumps (conditional and unconditional)
  - Procedures: `call` and `ret`
- Processor also needs to react to changes in *system state*
  - *e.g.*, Unix/Linux user hits "Ctrl-C" at the keyboard, user clicks on a different application's window on the screen, data arrives from a disk or a network adapter, instruction divides by zero, system timer expires

kills active process

- Can jumps and procedure calls achieve this?
  - No – the system needs mechanisms for *"exceptional"* control flow!

# Exceptional Control Flow

- Mechanisms exist at all levels of a computer system
  - **Exceptions** (low-level) are changes in processor's control flow in response to a system event (*e.g.*, change in system state, user-generated interrupt)
    - Implemented using a combination of hardware and OS software
  - **Process context switches** (higher-level) pass execution on the CPU from one process to another
    - Implemented by OS software and hardware timer
  - **Signals** (higher-level) are standardized inter-process messages to trigger specific behavior
    - Implemented by OS software
    - We won't cover these in detail – see CSE 451 and EE/CSE 474

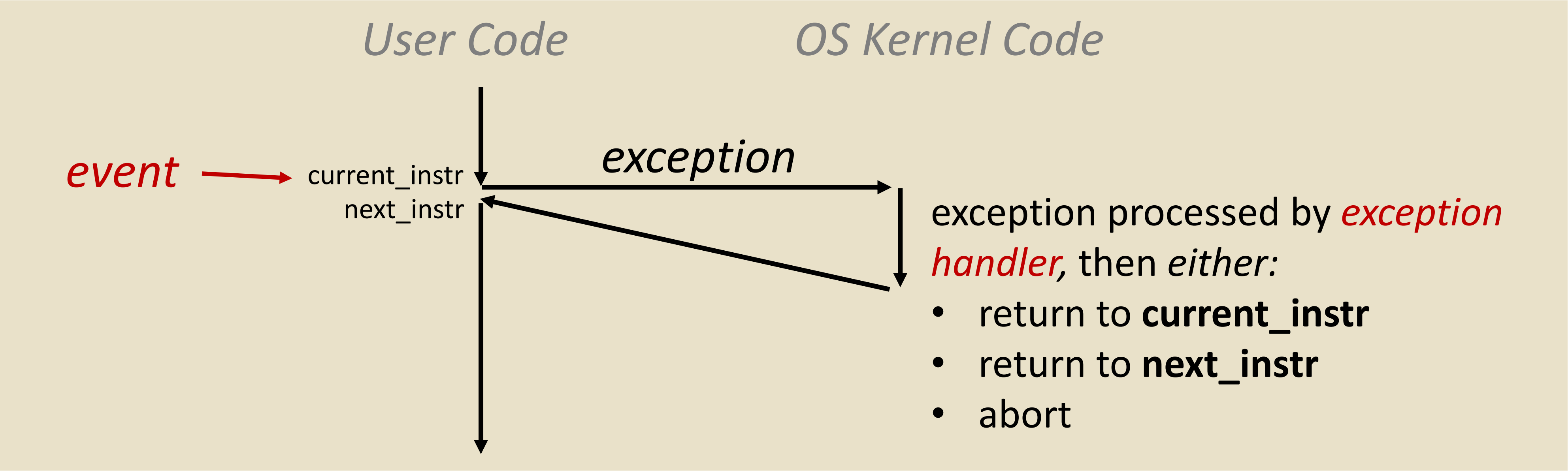
# Aside: Java Exceptions (Review)

This is extra (non-testable) material

- Java has exceptions, but they're *something different*
  - *e.g.*, `NullPointerException`, `MyBadThingHappenedException`
  - `throw` statements
  - `try`/`catch` statements
- Java exceptions are for reacting to (unexpected) program state
  - Can be implemented with stack operations and conditional jumps
  - A mechanism for "many call-stack returns at once"
  - Requires additions to the calling convention, but no additional CPU features
- System-state changes on previous slide are mostly of a different sort (asynchronous/external except for divide-by-zero) and implemented very differently

# Exceptions (Review)

- An *exception* is transfer of control to the operating system (OS) kernel in response to some *event* (*i.e.*, change in processor state)
  - Kernel is the operating system code that **lives in memory**
  - *e.g.*, division by 0, page fault, I/O request completes, Ctrl-C

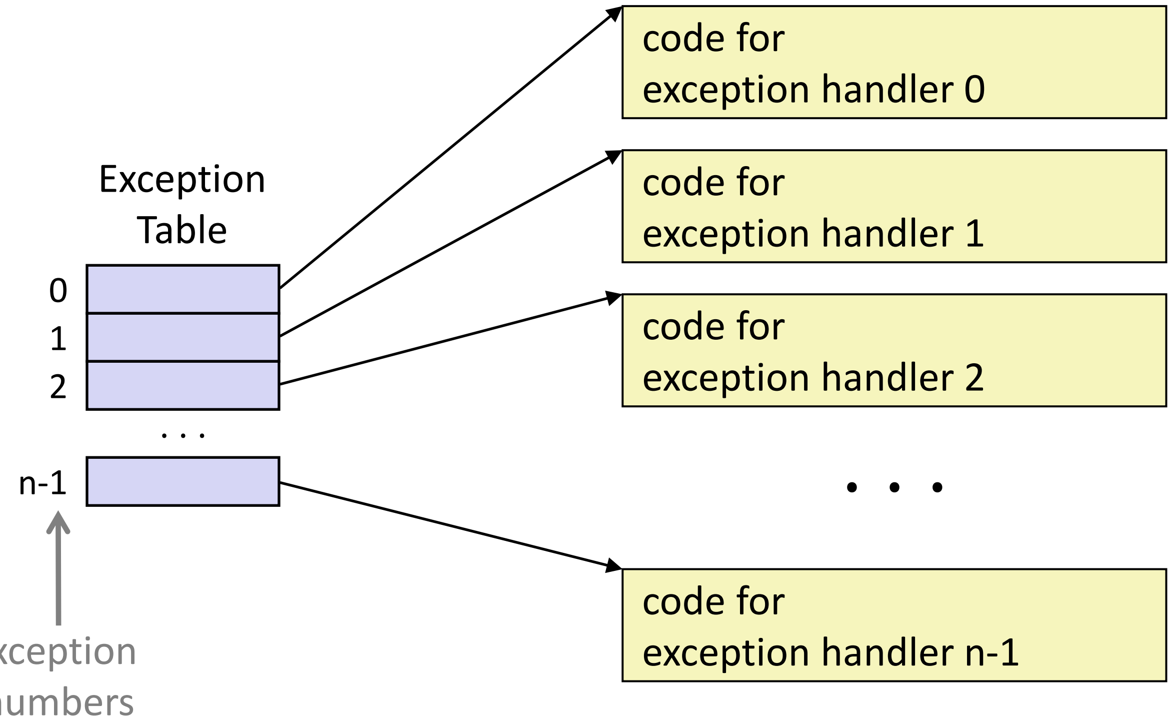



- *How does the system know where to jump to in the OS?*

# Exception Table

This is extra (non-testable) material

- A jump table for exceptions (also called *Interrupt Vector Table*)
  - 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



# Exception Table Excerpt

This is extra (non-testable) material

| *Exception Number* | *Description* | *Exception Class* |
|---|---|---|
| 0 | Divide error | Fault |
| 13 | General protection fault | Fault |
| 14 | Page fault | Fault |
| 18 | Machine check | Abort |
| 32-255 | OS-defined | Interrupt or trap |

Different on Windows vs. Linux, etc.

# *Asynchronous* Exceptions (Review)

- ***Interrupts***: caused by events **external to the processor**
  - Indicated by setting the processor's interrupt pin(s) (wire into CPU)
  - After interrupt handler runs, the handler returns to "next" instruction

- Examples:
  - I/O interrupts: hitting Ctrl-C on the keyboard, clicking a mouse button or tapping a touchscreen, arrival of a packet from a network, arrival of data from a disk
  - Timer interrupt: an external timer chip triggers an interrupt every few milliseconds
    - Used by the OS kernel to take back control from user programs

Context switches depend upon timer interrupts

# *Synchronous* Exceptions (Review)

- Caused by events that occur **because of executing an instruction**:
  - ***Traps***
    - **Intentional**: transfer control to OS to perform some function
    - Examples: *system calls*, breakpoint traps, special instructions
    - Returns control to "next" instruction
  - ***Faults***
    - **Unintentional** but possibly recoverable sometimes
    - Examples: *page faults*, segment protection faults, integer divide-by-zero exceptions
    - Either re-executes faulting ("current") instruction or aborts
  - ***Aborts***
    - **Unintentional** and unrecoverable
    - Examples: parity error, machine check (hardware failure detected)
    - Aborts current program

# Traps: System Calls

- Each system call has a unique ID number
  - NOT the same as exception numbers!
- Examples for x86-64 Linux :

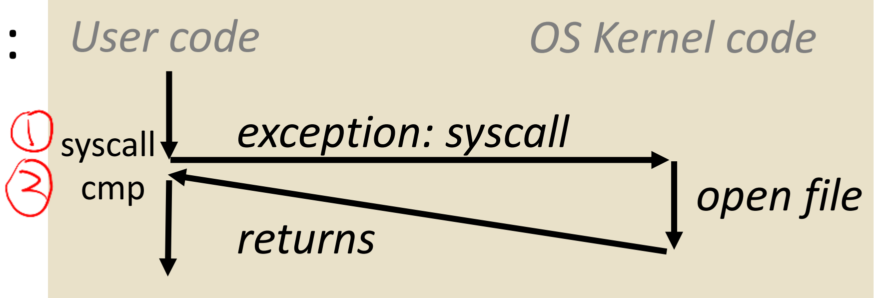
| Number | Name | Description |
|---|---|---|
| 0 | `read` | Read file |
| 1 | `write` | Write file |
| 2 | `open` | Open file |
| 3 | `close` | Close file |
| 4 | `stat` | Get info about file |
| 57 | `fork` | Create process |
| 59 | `execve` | Execute a program |
| 60 | `_exit` | Terminate process |
| 62 | `kill` | Send signal to process |

# Trap Example: Opening File

- User calls `open(filename, options)`
  - Calls `__open` function, which invokes system call instruction `syscall`:

```
0000000000e5d70 <__open>:
 ...
e5d79:   b8 02 00 00 00      mov   $0x2,%eax   # open is syscall 2
e5d7e:   0f 05               syscall          # return value in %rax
e5d80:   48 3d 01 f0 ff ff   cmp   $0xfffffffffffff001,%rax
 ...
e5dfa:   c3                  retq
```

    - `%rax` contains syscall number
    - Other arguments in `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8`, `%r9`
    - Return value in `%rax`, negative value indicates an error
  - Execution resumes at next instruction:



# Fault Example: Page Fault

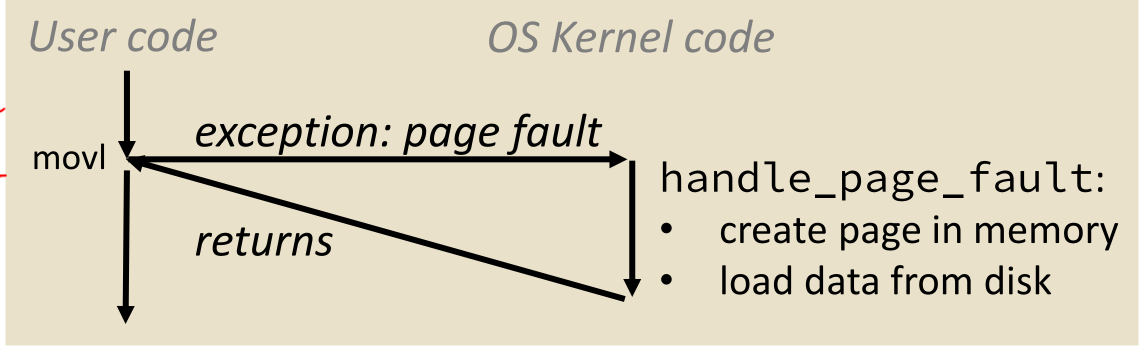
- User writes to memory location:

```
int a[1000];
int main () {
  a[500] = 13;
}
```

```
80483b7:        c7 05 10 9d 04 08 0d    movl    $0xd,0x8049d10
```

  - That location happens to be currently on disk (not in memory)
    - Page fault handler must load page into physical memory
  - Execution returns to faulting instruction: (mov is executed again)
    - Successful on second try

1st try = fault
2nd try = success



# Fault Example: Invalid Memory Reference

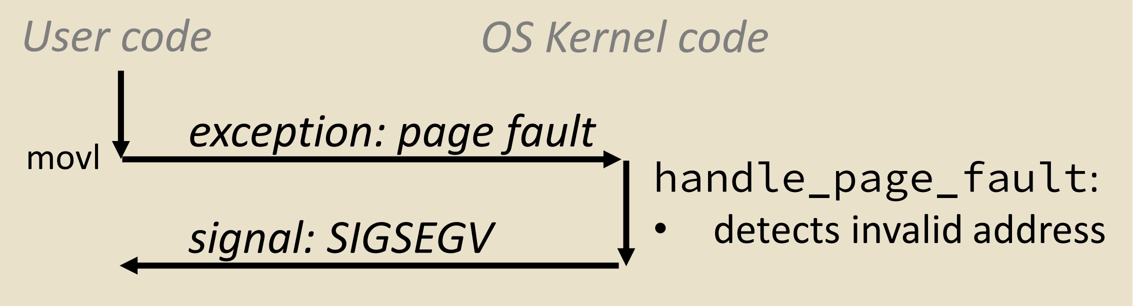
- User writes to memory location:

```
int a[1000];
int main () {
  a[5000] = 13;
}
```

```
80483b7:        c7 05 60 e3 04 08 0d    movl    $0xd,0x804e360
```

- That location is past allocated memory in an invalid region
  - Page fault handler detects invalid address
- `SIGSEGV` signal sent to user process:
  - User process exits with "segmentation fault"

Signal Segment Violation



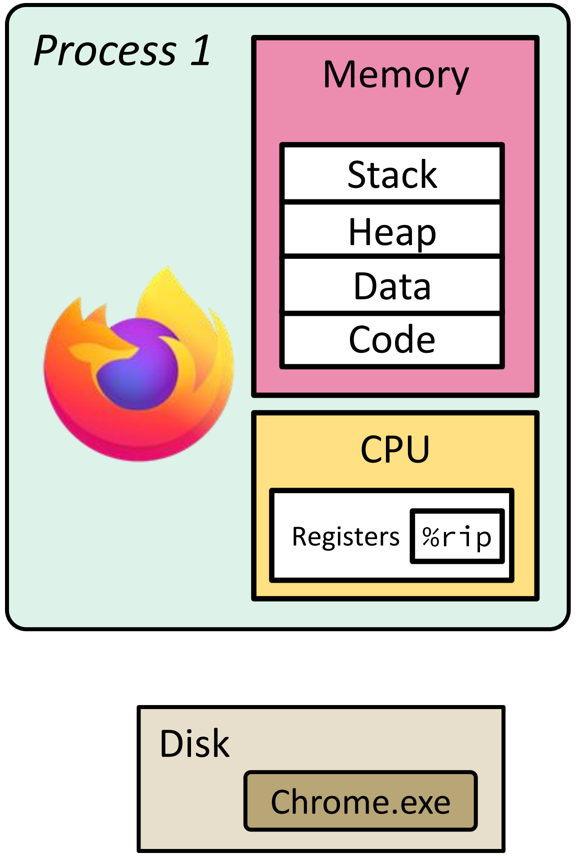
# Lecture Outline (2/3)

- System Control Flow
- **Processes**
- Process Management (x86-64 Linux)

# What is a Process?

- **It's an *illusion/abstraction*!**
  - The OS uses a data structure to represent each process
  - Maintains the *interface* between the program and the underlying hardware (CPU + memory)

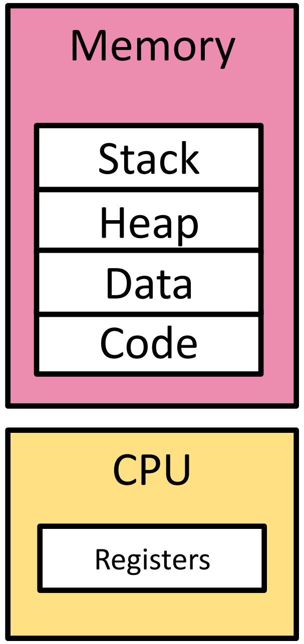
- Exceptional control flow is the *mechanism* the OS uses to enable **multiple processes** to run on the same system



# Processes (Review)

- A ***process*** is an instance of a running program
  - One of the most profound ideas in computer science

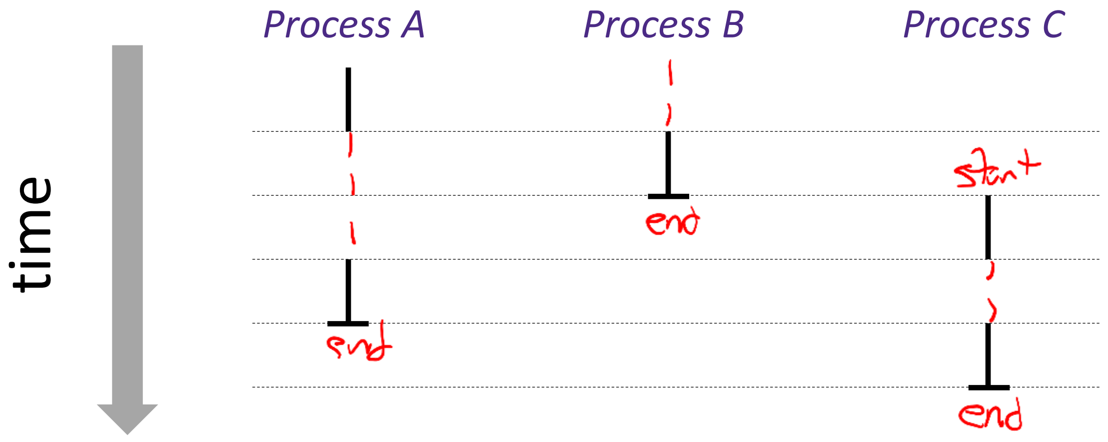
- Process provides each program with two key abstractions:
  - *Logical control flow*
    - Each program *seems to* have exclusive use of the CPU
    - Provided by kernel mechanism called ***context switching***
  - *Private address space*
    - Each program *seems to* have exclusive use of main memory
    - Provided by kernel mechanism called ***virtual memory***



# Concurrent Processes (Review)

Assume only one CPU

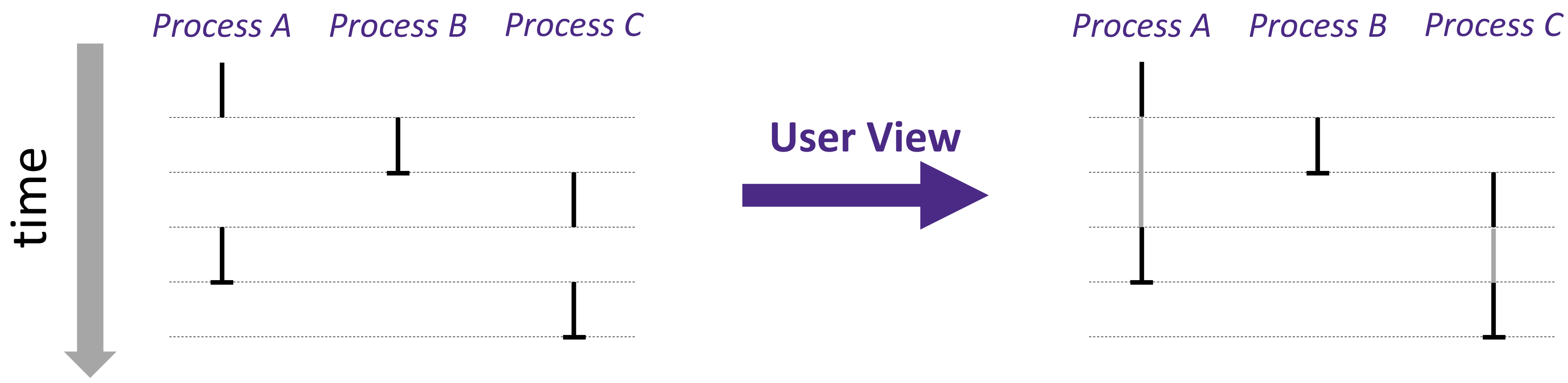
- Each process is a logical control flow
- Two processes *run concurrently* (are concurrent) if their instruction executions (flows) overlap in time
  - Otherwise, they are *sequential*
- Example: (running on single core)
  - Concurrent: A & B, A & C
  - Sequential: B & C



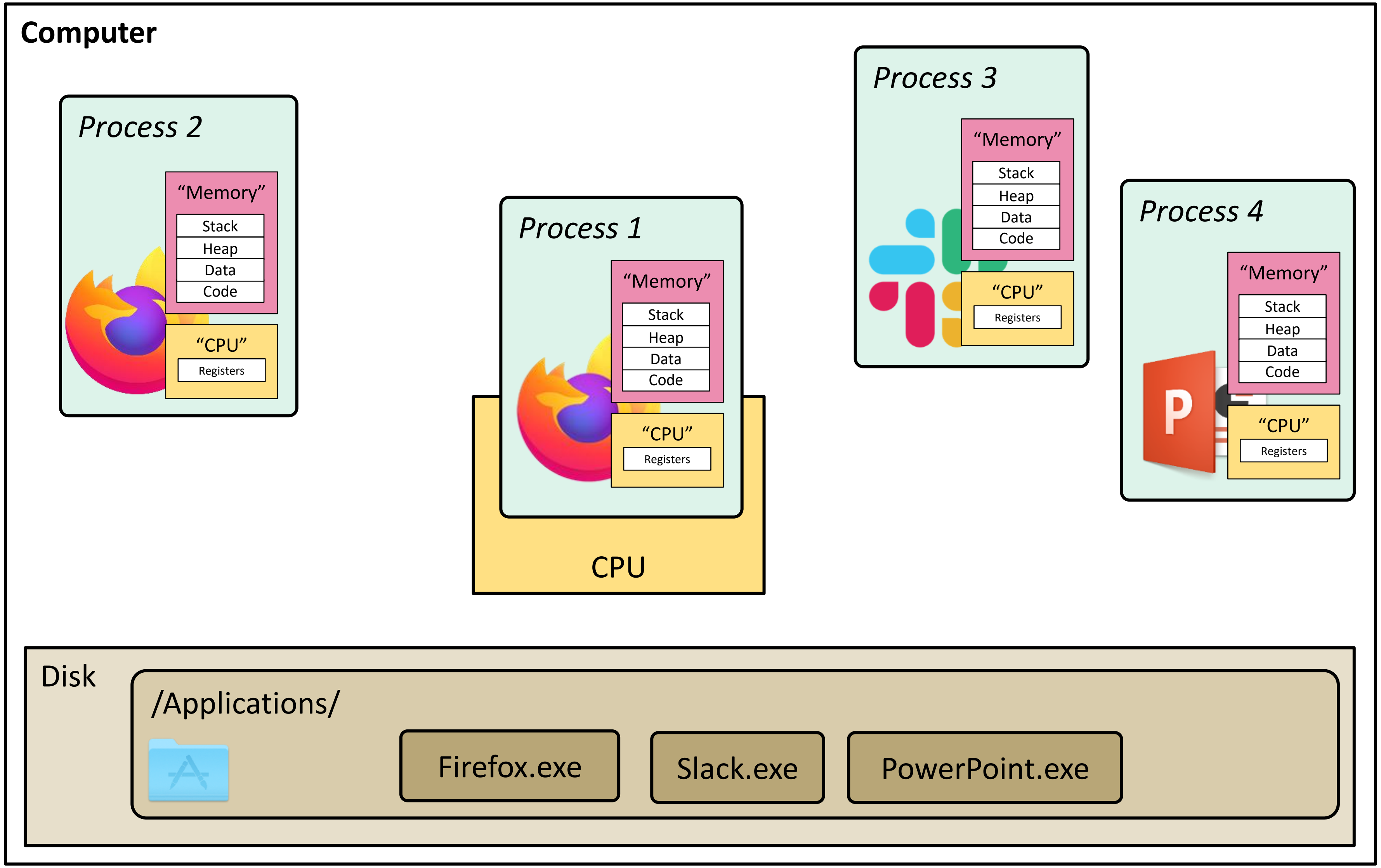
# User's View of Concurrency

Assume only <u>one</u> CPU

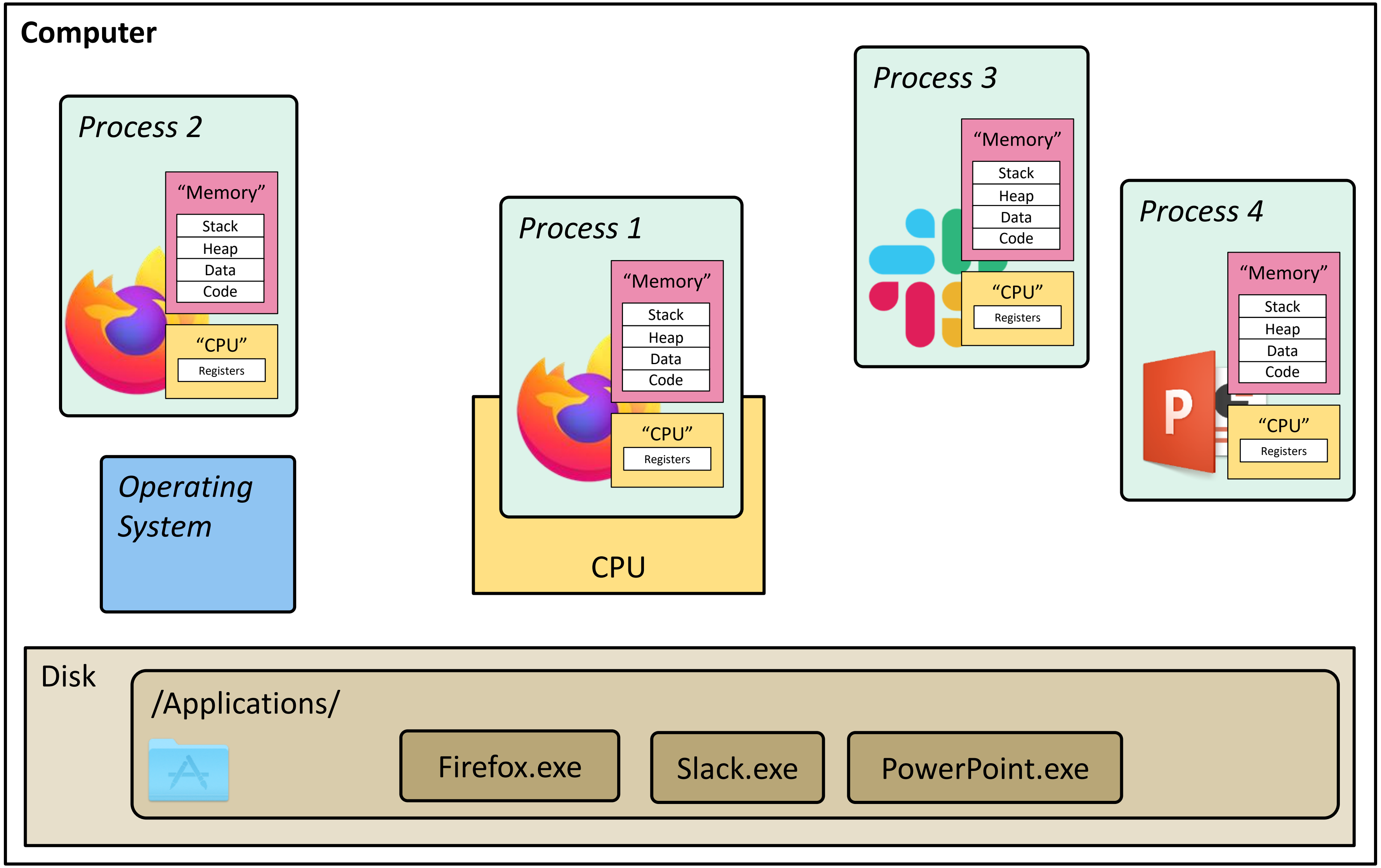
- Control flows for concurrent processes are physically disjoint in time
  - CPU only executes instructions for one process at a time
- However, the user can *think of* concurrent processes as executing at the same time, in *parallel*



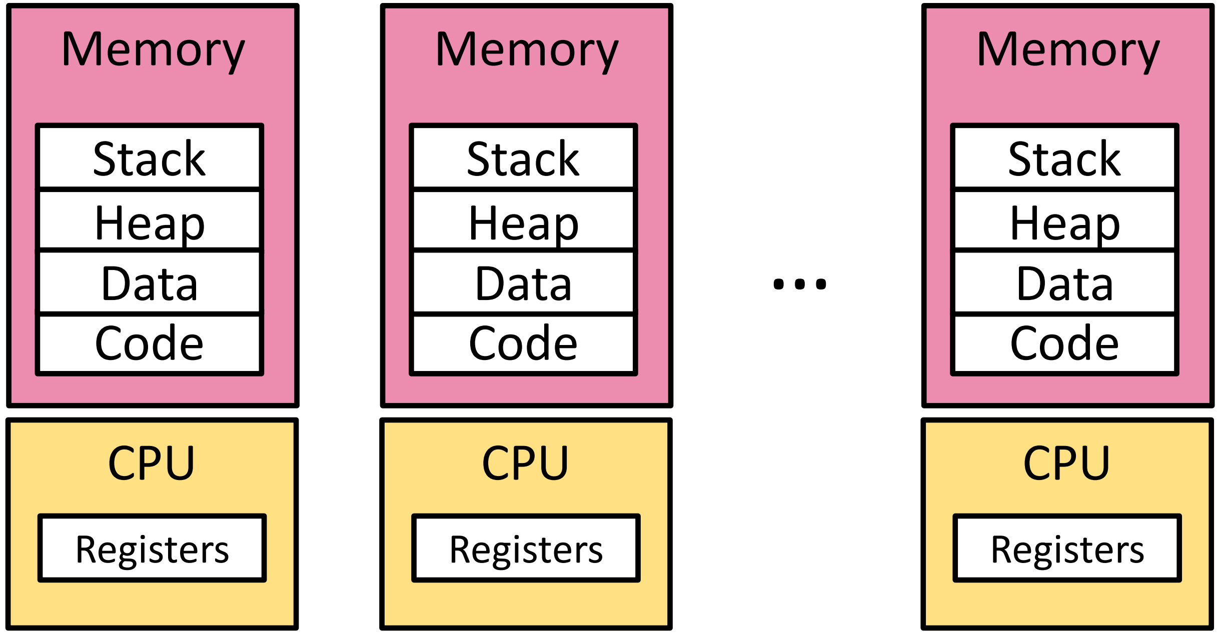
# Multiple Processes



# Multiple Processes: Context Switching

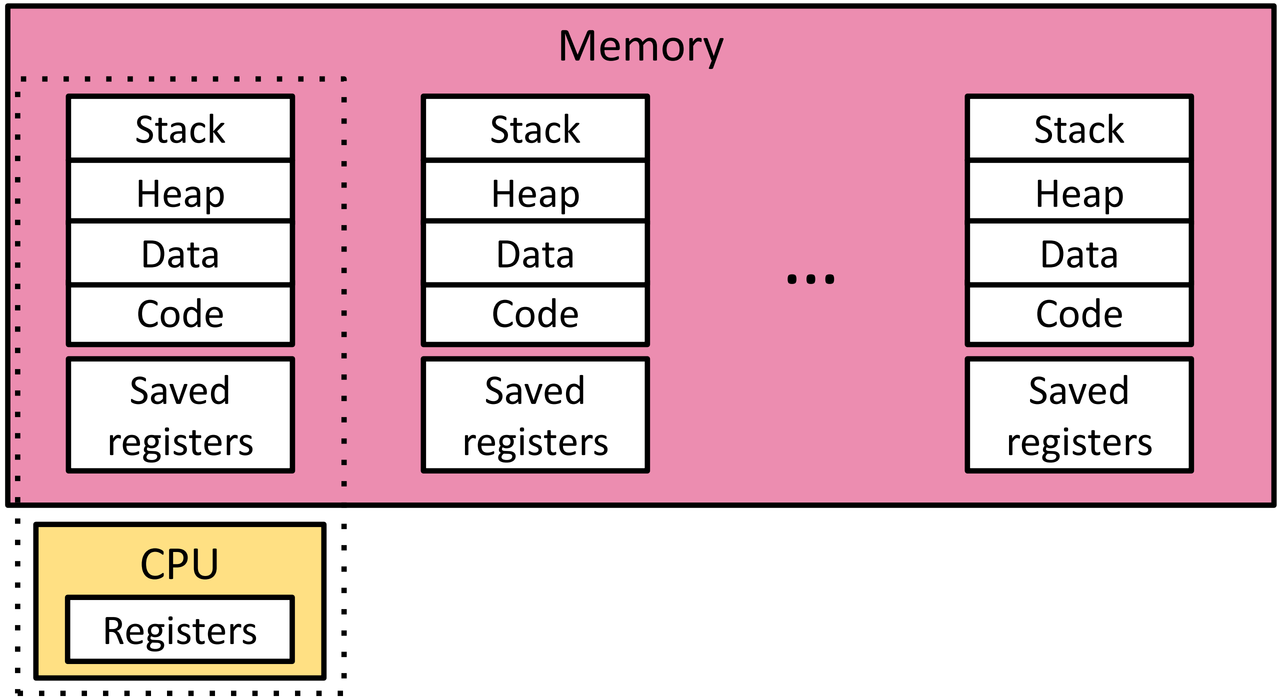


# Multiprocessing: The Illusion



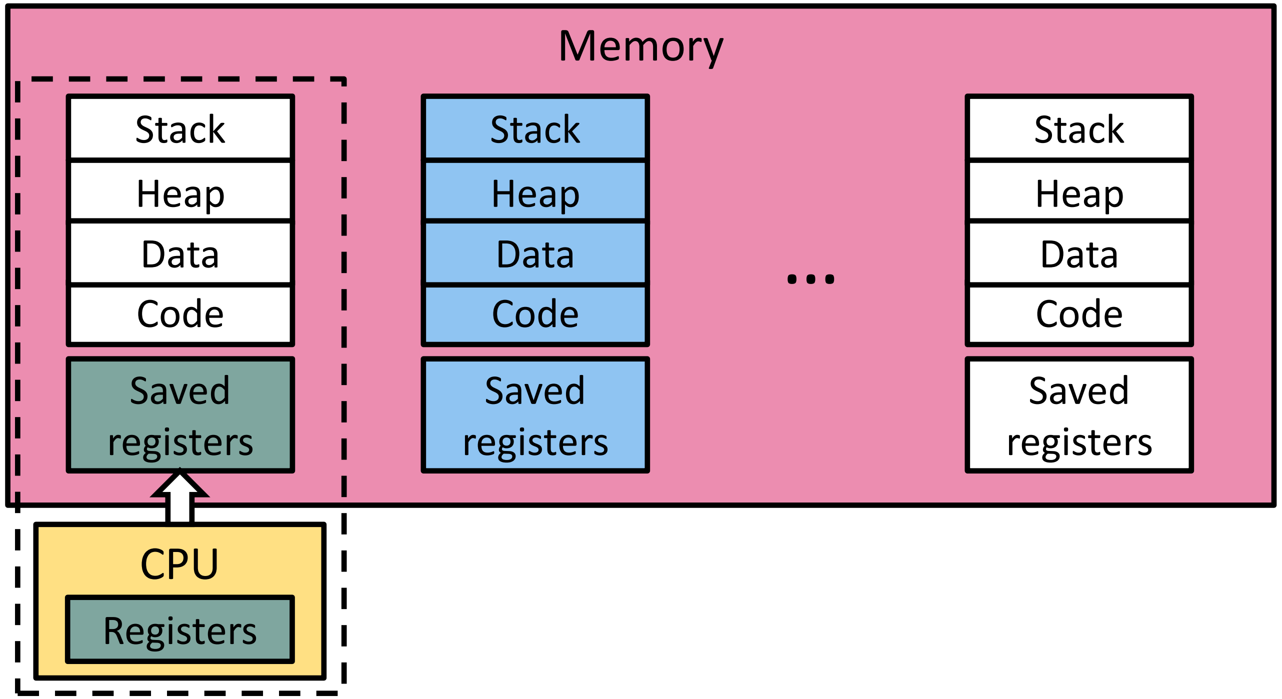

- A computer runs many processes simultaneously
  - Applications for one or more users (*e.g.*, web browsers, email clients, text editors)
  - Background tasks (*e.g.*, monitoring network & I/O devices)

# Multiprocessing: The Reality



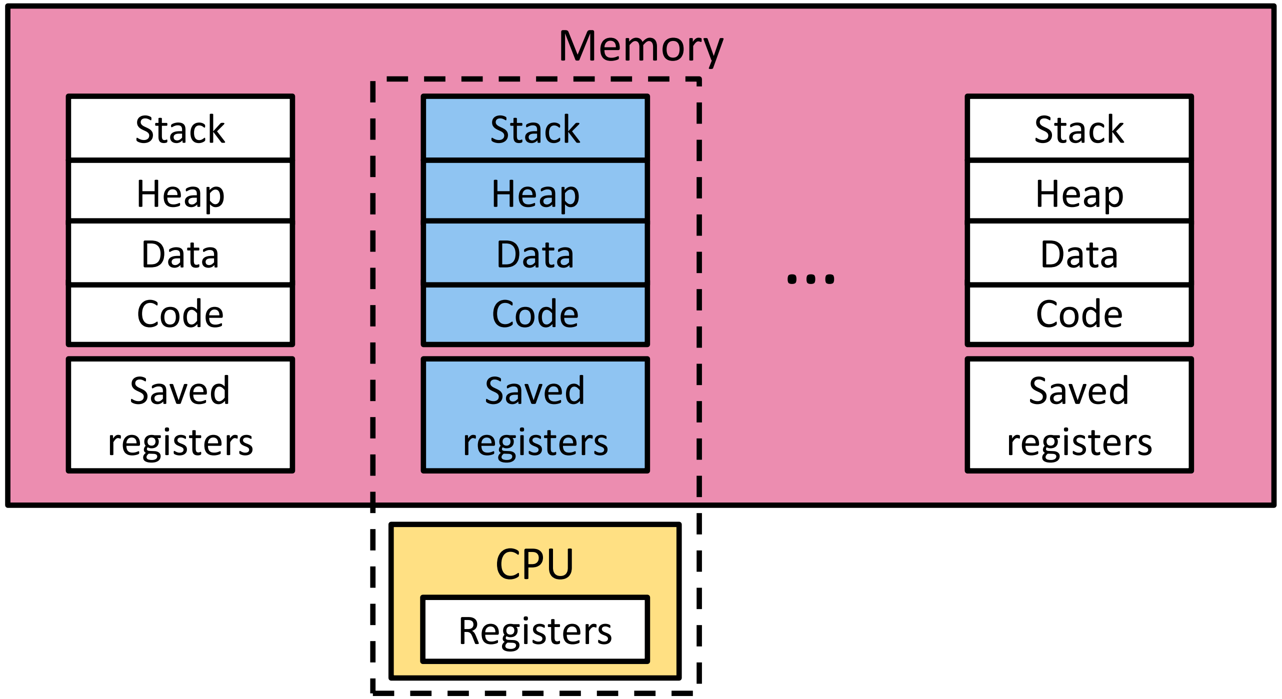

- Single processor executes multiple processes *concurrently*
  - Process executions are interleaved – CPU only runs *one at a time*
  - Address spaces managed by virtual memory system (we'll get to it!)
  - *Execution context* (register values, stack, …) for other processes saved in memory

# Multiprocessing: Context Switching (Review, 1/3)



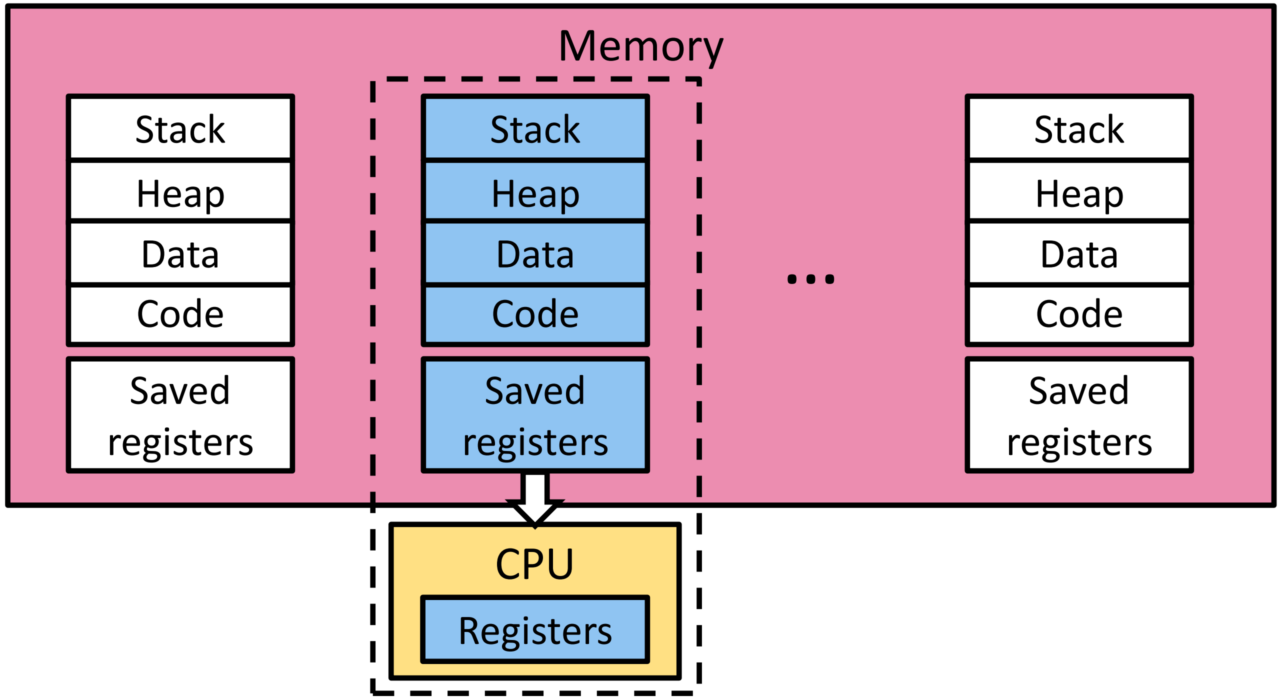

- Context switch
  1) **Save current registers in memory**

# Multiprocessing: Context Switching (Review, 2/3)



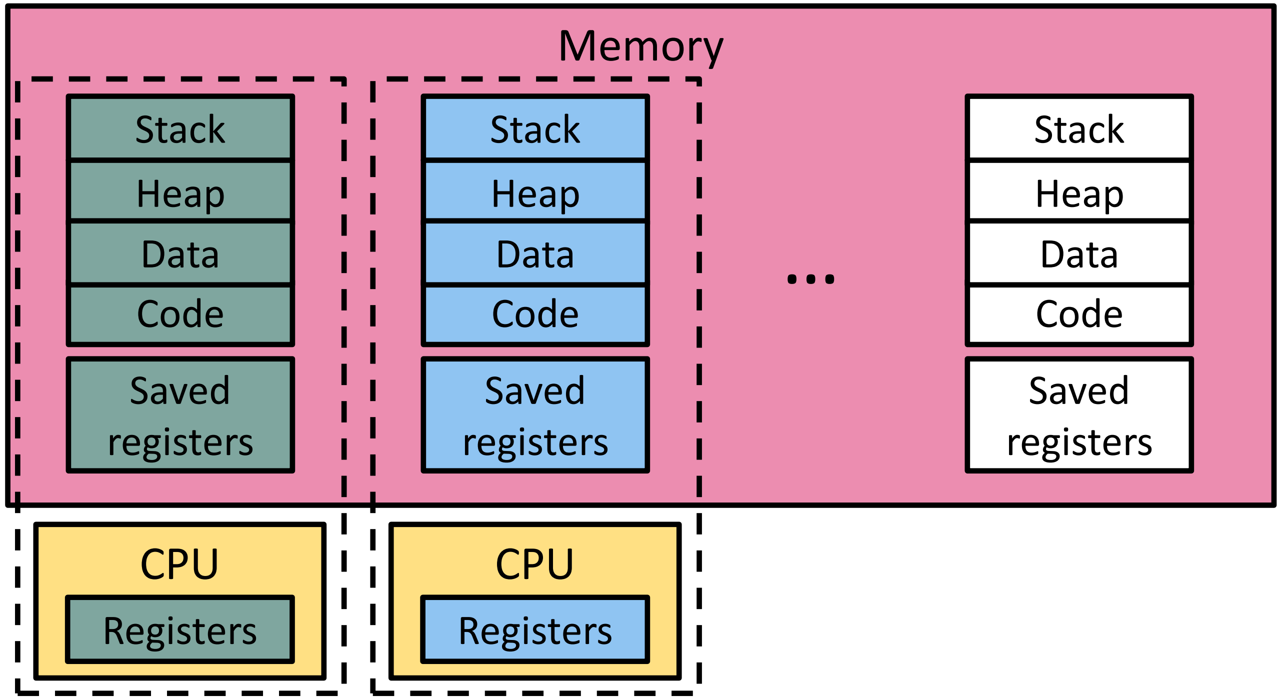

- Context switch
  1) Save current registers in memory
  2) **Schedule next process for execution**

# Multiprocessing: Context Switching (Review, 3/3)




- Context switch
  1) Save current registers in memory
  2) Schedule next process for execution
  3) **Load saved registers and switch address space**

# Multiprocessing: The Modern Reality

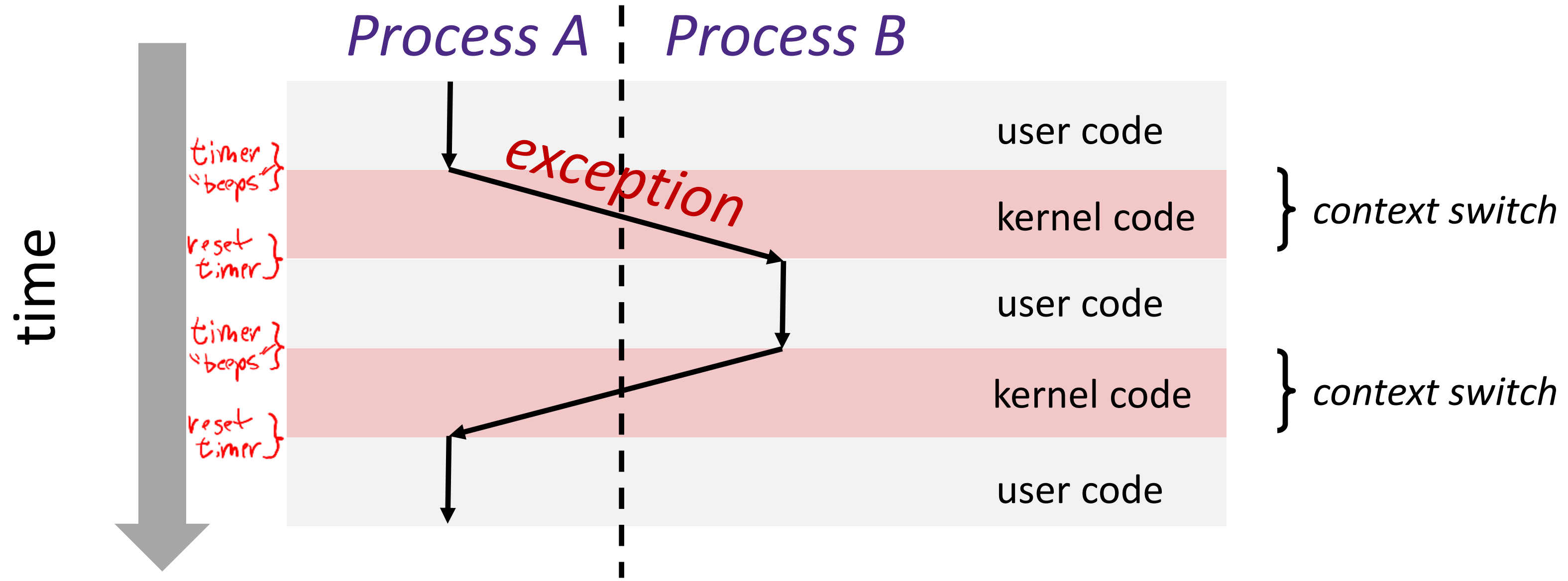



- Multicore processors have multiple CPUs ("cores") on a single chip
  - Each can execute a separate process, but ***still*** **constantly swapping processes**
    - Kernel schedules processes to cores
  - Share main memory (and some of the caches)

# Context Switching via Exceptions (Review)

Assume only one CPU

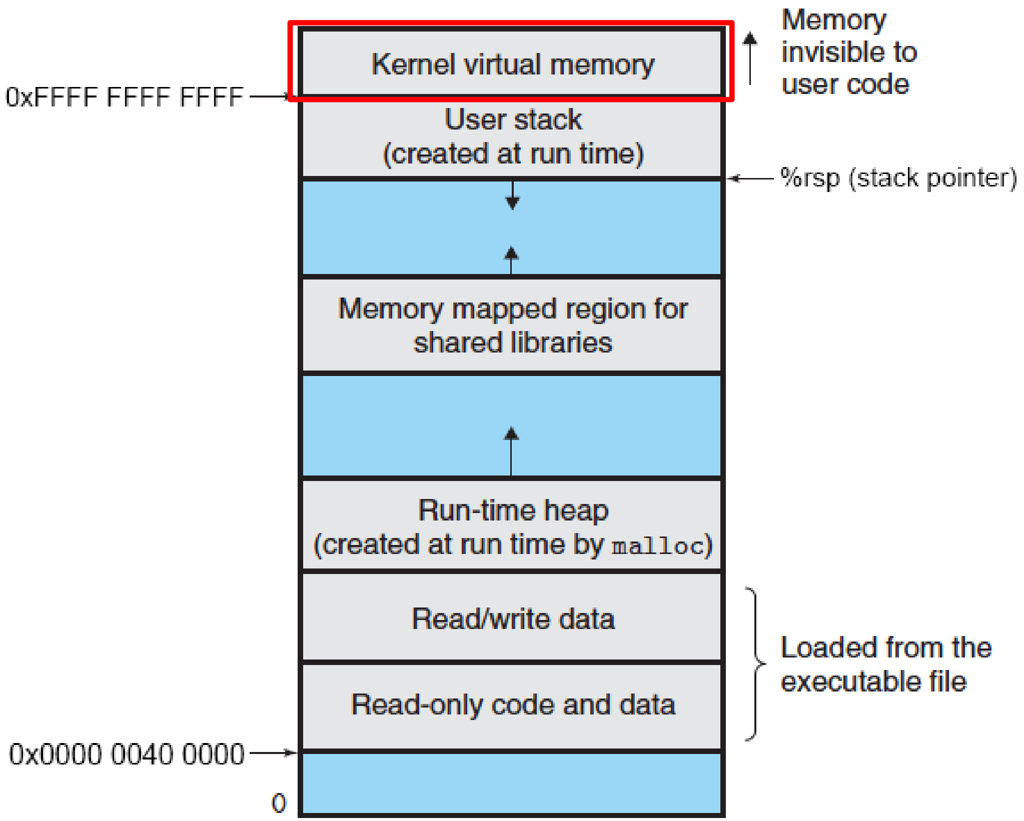
- Context switch passes control flow from one process to another and is performed using kernel code



# Context Switching: The Kernel

Assume only one CPU

- Processes are managed by a *shared* chunk of OS code called the kernel
  - The kernel is not a separate process, but rather runs as part of a user process
- In x86-64 Linux:
  - Same address in each process refers to same shared memory location*

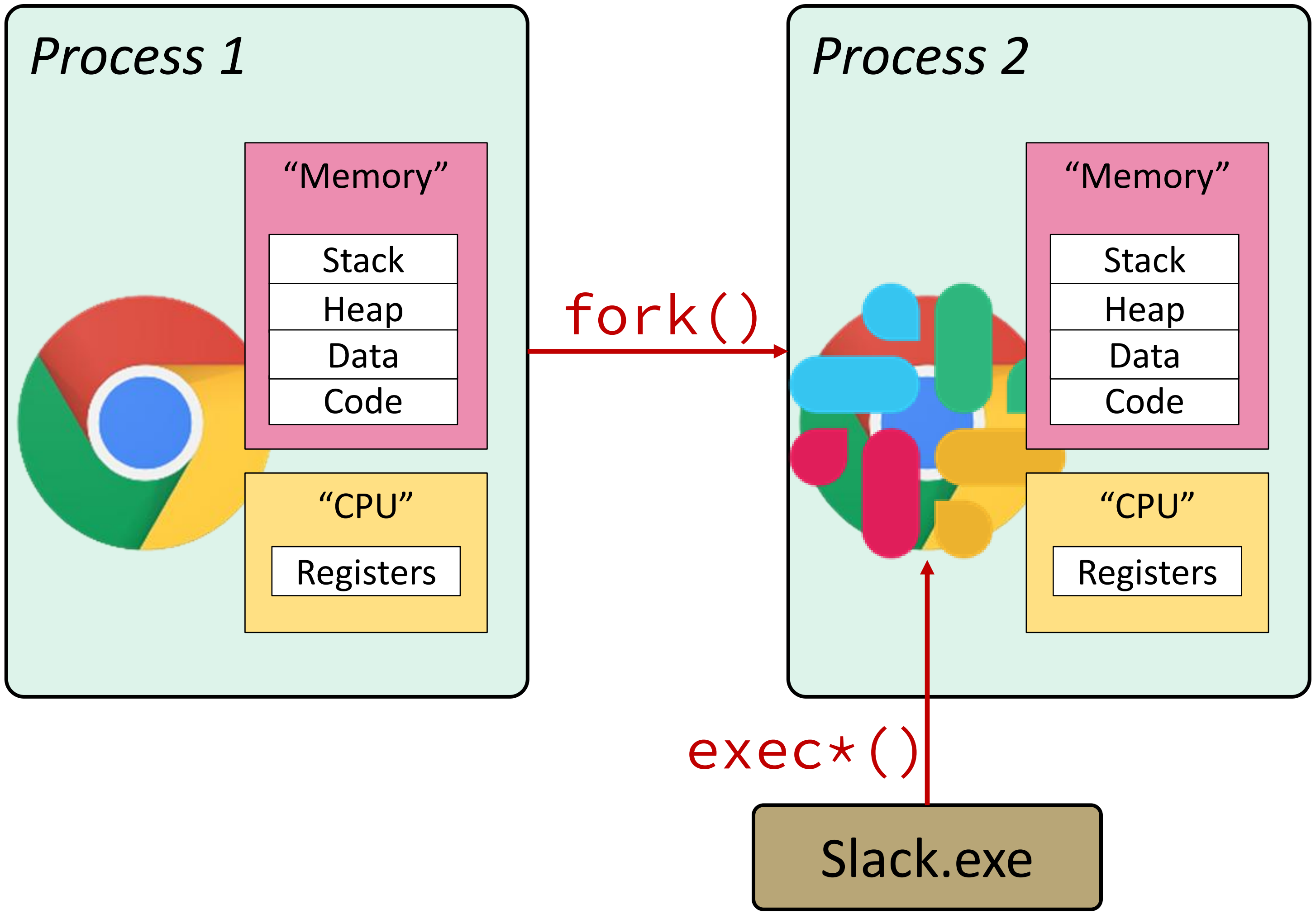



* sort of, the story became more complicated recently with Meltdown and Spectre (out of scope here)

# Lecture Outline (3/3)

- System Control Flow
- Processes
- **Process Management (x86-64 Linux)**

# Creating New Processes & Programs



# Process Management in Linux (Mostly Review)

- fork-exec model:
  - `fork()` creates a copy of the current process
  - `exec*()` replaces the current process' code and address space with the code for a different program
    - Family: `execv`, `execl`, `execve`, `execle`, `execvp`, `execlp`
  - `fork()` and `execve()` are *system calls*

- Other system calls for process management:
  - `getpid()`
  - `exit()`
  - `wait()`,`waitpid()`

# fork: Creating New Processes

- `pid_t fork(void)`
  - Creates a "**child**" process from the calling "**parent**" process that is *almost* identical
    - Child has a newly assigned *process ID (PID)* that is different than the parent's PID
    - Child gets an identical, but separate, copy of the parent's **state** (*e.g.*, memory, registers)
  - Both start/resume execution at the return from `fork`
    - Returns `0` to the **child** process, and the *child's PID* to the **parent** process

```
pid_t fork_ret = fork();
if (fork_ret == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

- ⚠ Unique (and confusing) because it is called **once** but returns **"twice"**

# fork Illustration (1/3)

**Process X** (parent; PID X)

```
pid_t fork_ret = fork();
if (fork_ret == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

**Process Y** (child; PID Y)

```
pid_t fork_ret = fork();
if (fork_ret == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

# `fork` Illustration (2/3)

**Process X** (parent; PID X)

```
pid_t fork_ret = fork();
if (fork_ret == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

fork_ret = Y

```
pid_t fork_ret = fork();
if (fork_ret == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

**Process Y** (child; PID Y)

```
pid_t fork_ret = fork();
if (fork_ret == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

fork_ret = 0

```
pid_t fork_ret = fork();
if (fork_ret == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

# fork Illustration (3/3)

## Process X (parent; PID X)

```
pid_t fork_ret = fork();
if (fork_ret == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

fork_ret = Y

```
pid_t fork_ret = fork();
if (fork_ret == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

hello from parent

## Process Y (child; PID Y)

```
pid_t fork_ret = fork();
if (fork_ret == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

fork_ret = 0

```
pid_t fork_ret = fork();
if (fork_ret == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

hello from child

***We don't know which will appear first!***

# fork Example

```
void fork1() {
  int x = 1;
  pid_t fork_ret = fork();
  if (fork_ret == 0)
    printf("Child has x = %d\n", ++x);
  else
    printf("Parent has x = %d\n", --x);
  printf("Bye from process %d with x = %d\n", getpid(), x);
}
```

- Notes/Reminders:
  - Both processes continue/start execution after fork
    - Can't predict execution order between parent and child
  - Both processes start with x = 1
    - However, subsequent changes to x are independent
  - Shared open files: stdout is the same in both parent and child

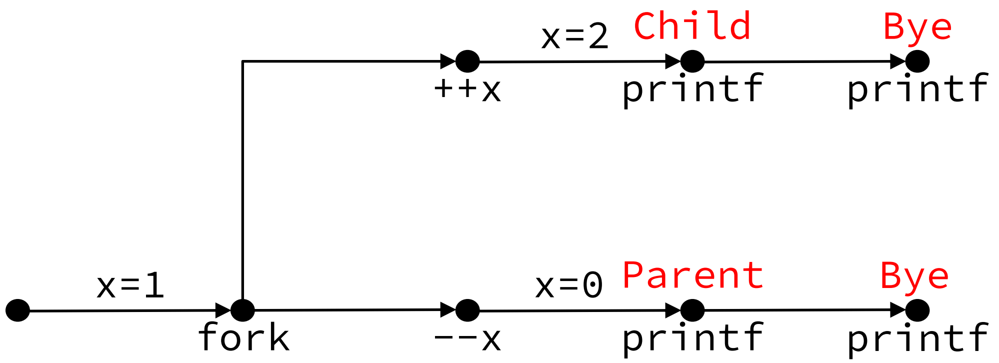
# Modeling Concurrency with Process Graphs

- A *process graph* is a useful tool for capturing the partial ordering of statements in a concurrent program
  - Each vertex indicates the execution of a notable statement
  - Edges (a → b) indicate sequential ordering of statements within a process
    - *i.e.*, a must happen before b
  - Vertices and edges can be labeled with important notes
    - *e.g.*, updated variable values on edges, program output on `printf` vertices
  - Each graph begins with a vertex with no in-edges

- Any *topological sort* of the graph corresponds to a feasible total ordering
  - An ordering of nodes that contains every node, and only follows edges (lines between nodes) in the direction of the arrows

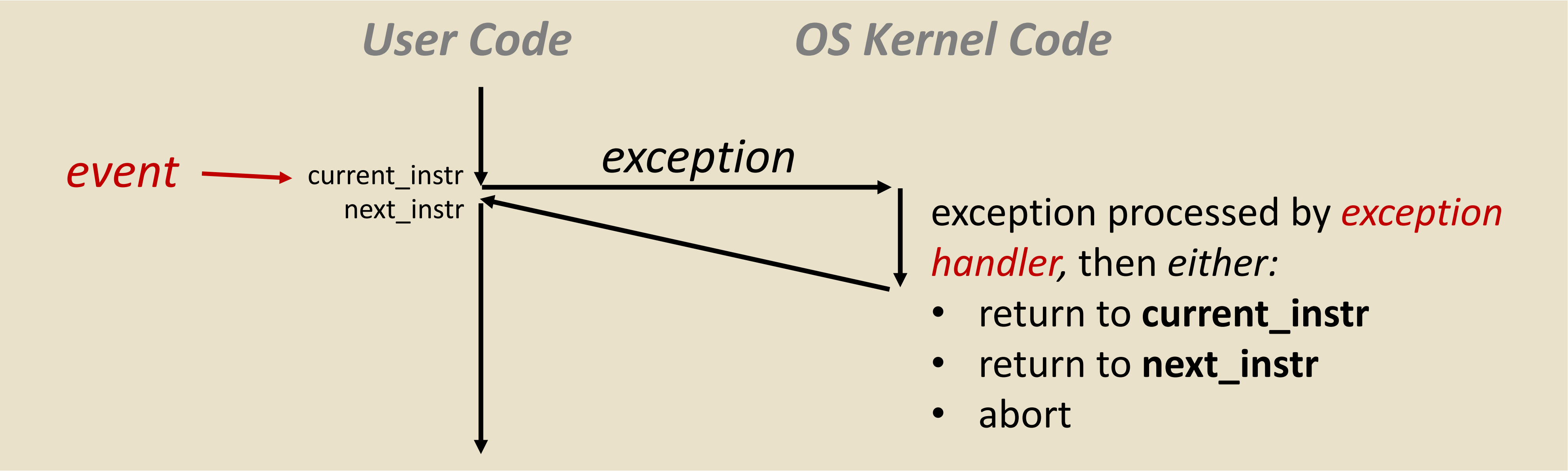
# fork Example: Process Graph

```
void fork1() {
  int x = 1;
  pid_t fork_ret = fork();
  if (fork_ret == 0)
    printf("Child has x = %d\n", ++x);
  else
    printf("Parent has x = %d\n", --x);
  printf("Bye from process %d with x = %d\n", getpid(), x);
}
```



# Summary (1/3)

- ***Exceptional control flow*** enables a computer to respond/react to system *events* that can be external to the running process
  - The event generates an ***exception*** that transfers control to ***exception handler*** in operating system kernel, which will have 1 of 3 possible outcomes:



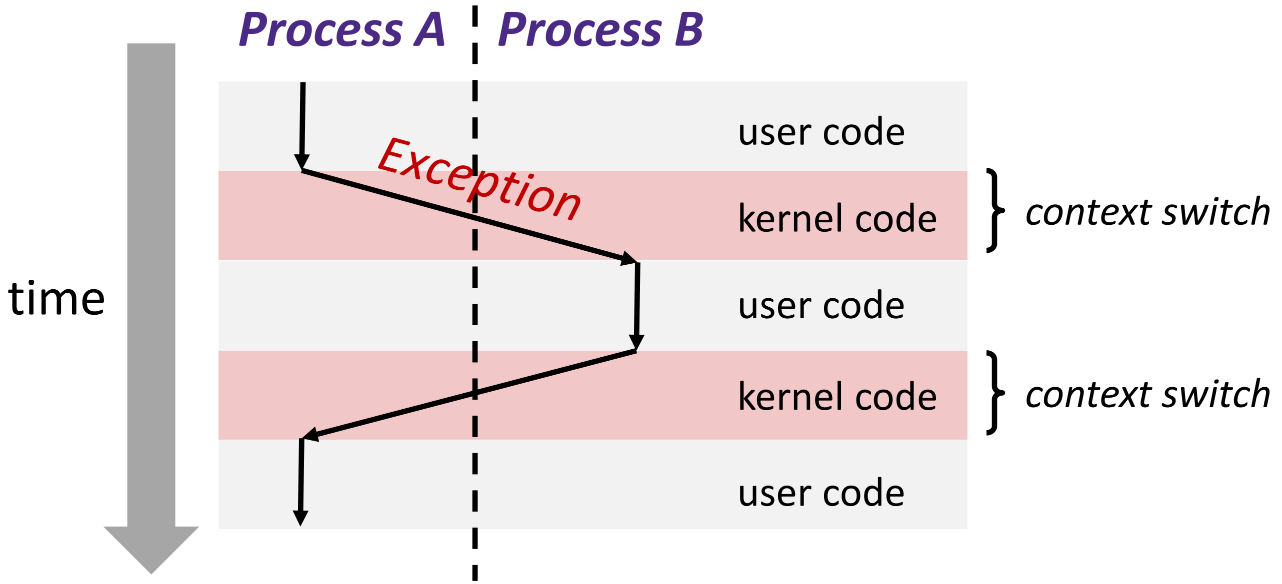
# Summary (2/3)

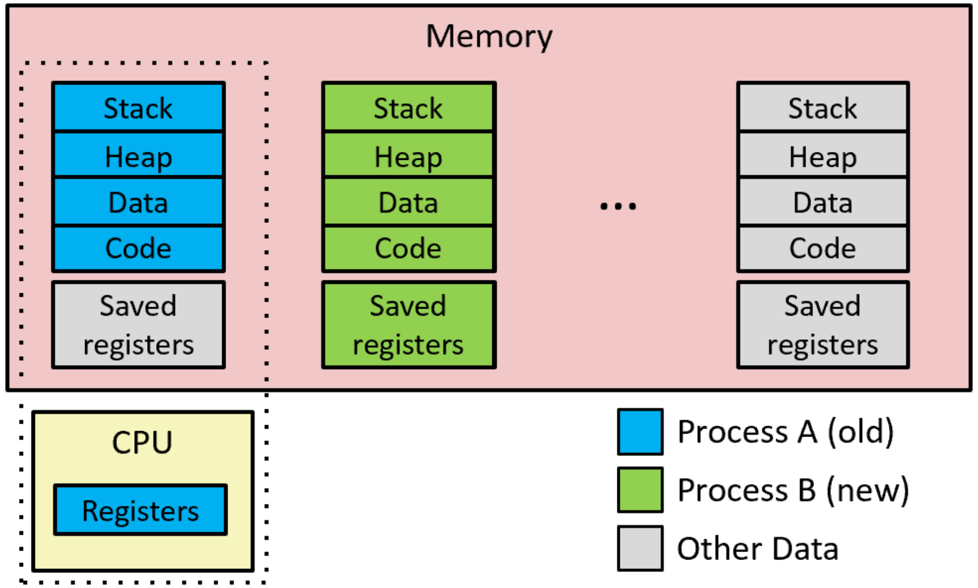
- *Asynchronous exceptions* (external to running process)
  - ***Interrupts*** don't affect the currently running process
- *Synchronous exceptions* (internal to running process)
  - ***Traps*** are intentional – asking the operating system to do something for you
  - ***Faults*** are unintentional but possibly recoverable
  - ***Aborts*** are unintentional and unrecoverable

- A ***process*** is an instance of an running program and provides two key abstractions: logical control flow and private address space
  - Concurrently executing processes are scheduled non-deterministically by the operating system

# Summary (3/3)

- Multiple running processes can be run *concurrently* via ***context switching***




- The ***fork-exec model***
  - Every process is assigned a unique ***process ID*** (pid)
  - `fork()` returns 0 to child, child's PID to parent