System Control Flow & Processes

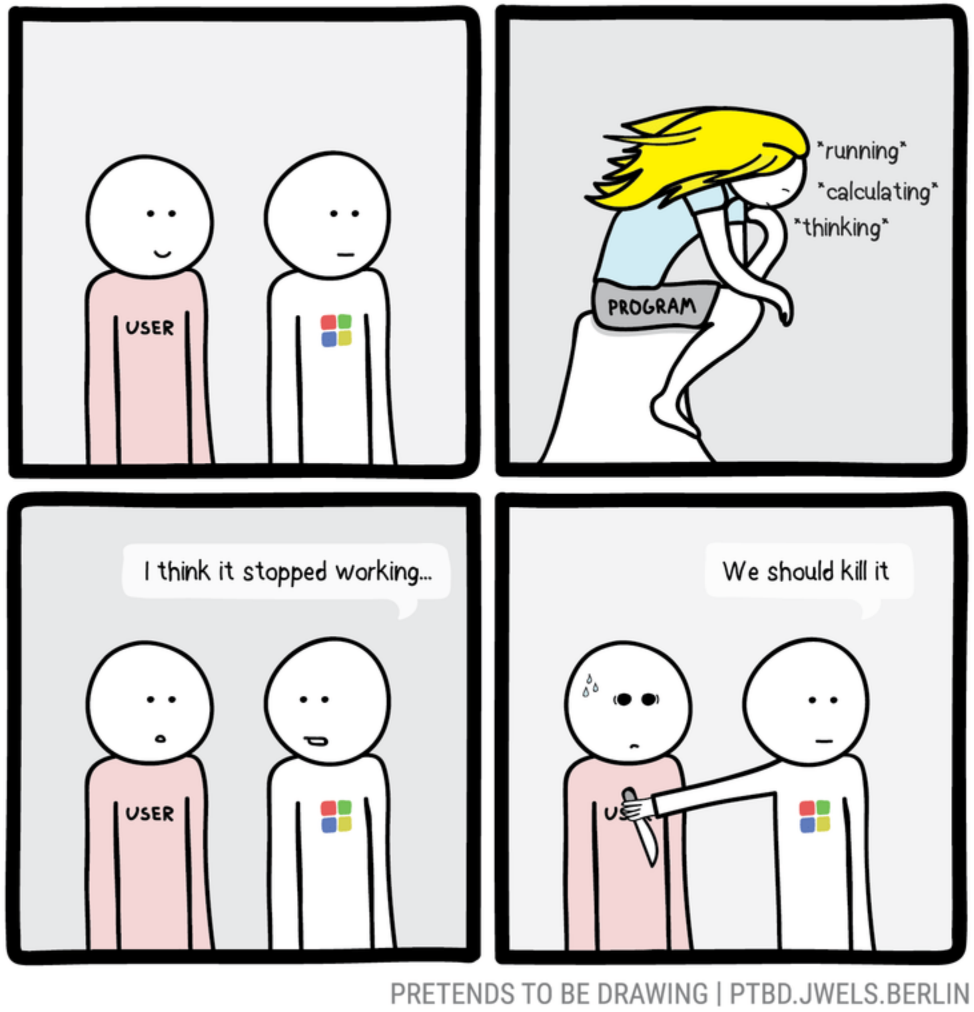
CSE 351 Spring 2024

Instructor:

Elba Garza

Teaching Assistants:

Ellis Haker
Adithi Raghavan
Aman Mohammed
Brenden Page
Celestine Buendia
Chloe Fong
Claire Wang
Hamsa Shankar
Maggie Jiang
Malak Zaki
Naama Amiel
Nikolas McNamee
Shananda Dokka
Stephen Ying
Will Robertson



Playlist: [CSE 351 24Sp Lecture Tunes!]

Announcements, Reminders

- HW19 due tonight!
  - HW20 due Wednesday (15 May)
  - HW21 due Friday (17 May)
  - HW22 due Monday (20 May)
- Lab 4 due Friday
  - Use any late days left on Lab 4!
- Lab 5 releasing Friday!
- Guest lectures by Prof. Kelly Shaw on Wednesday & Friday 😎

Midterm Grades: A Reminder

Your **success in life** is not defined by grades.

You are not defined by grades.

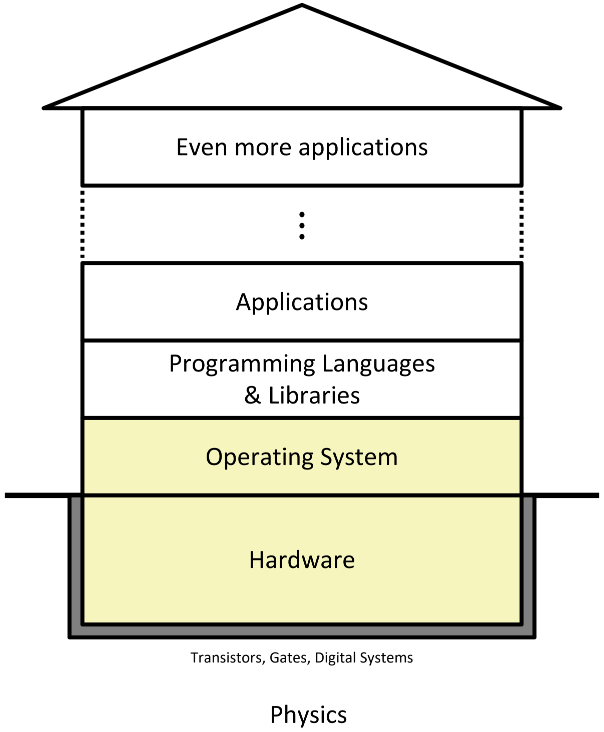
- We know all of this seems critically important right now, but we promise, the numbers on a transcript will fade with time. I (Elba) personally got both the highest and lowest midterm grades in my classes at some point in my college career.

- **We'll release grades later this week!**
  - Regrade requests will be open for a week – please let us know if anything looks amiss!

The Hardware/Software Interface

- Topic Group 3: **Scale & Coherence**
  - Caches, **Processes**, Virtual Memory, Memory Allocation

So far we've been viewing concepts from the perspective of a single program…



- How do we maintain logical consistency in the face of more data and more processes?
  - How do we support control flow both within many processes and things external to the computer?
  - How do we support data access, including dynamic requests, across multiple processes? (Hint: Virtual memory next time!)

Reading Review

- Terminology:
  - Exceptional control flow, event handlers
  - Operating system kernel
  - Exceptions: interrupts, traps, faults, aborts
  - Processes: concurrency, context switching, fork-exec model, process ID

Leading Up to Processes

- System Control Flow
  - **Control flow**
  - **Exceptional control flow**
  - Asynchronous exceptions (interrupts)
  - Synchronous exceptions (traps & faults)

Control Flow

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

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**

Physical control flow

time ↓

<startup>
$instr_1$
$instr_2$
$instr_3$
…
$instr_n$
<shutdown>

Altering the Control Flow

- Up to now, two ways to change control flow:
  - Jumps (conditional and unconditional)
  - Call and return
  - Both react to changes in program state

Not the same!

- Processor also needs to react to changes in system state:
  - 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 (important later!)
- Can jumps and procedure calls achieve this?
  - No – the system needs mechanisms for "*exceptional*" control flow!

Before, we've been operating in a world where everything comes from within a program, but now we have to think about what happens outside the program.

These are unrelated to Java's exceptions.

Exceptional Control Flow

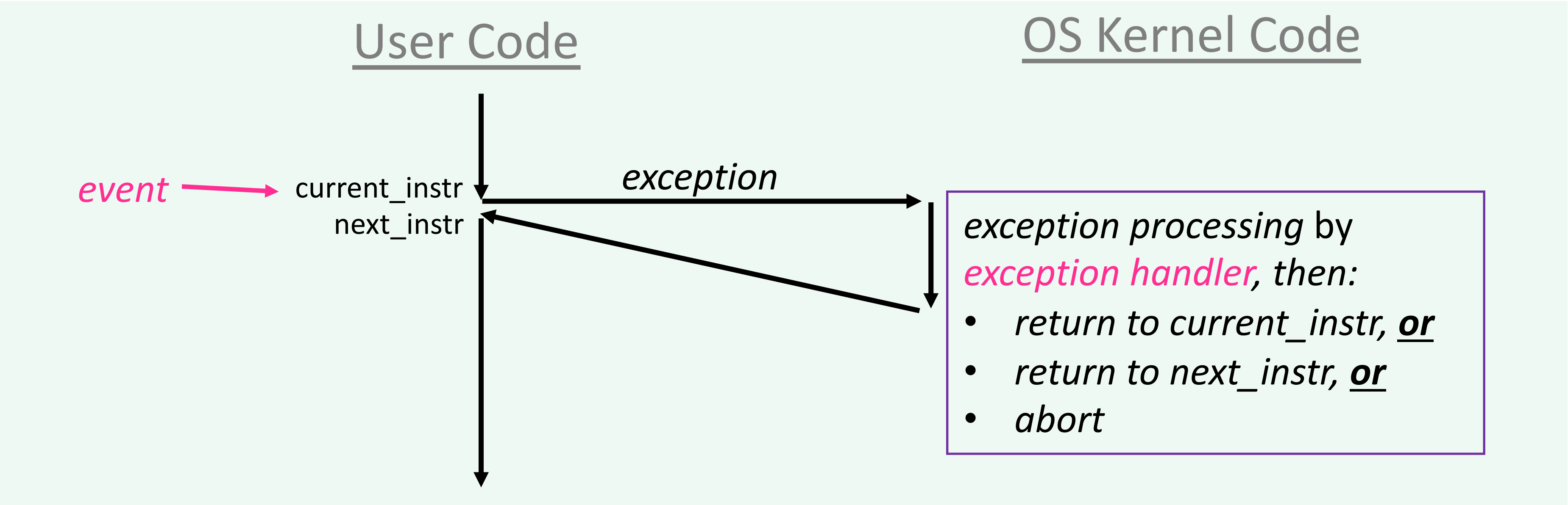
Exists at **all levels** of a computer system:

- Low level mechanisms
  - **Exceptions**
    - Change in processor's control flow in response to a system event (*i.e.*, change in system state, user-generated interrupt)
    - Implemented using a combination of hardware and OS software
- Higher level mechanisms
  - **Process context switch**
    - Implemented by OS software and hardware timer
  - **Signals**
    - Implemented by OS software
    - We won't cover these in detail—see CSE 451 and EE/CSE 474

Not covered here!

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, very VIP**
  - Examples: division by 0, page fault, I/O request completes, Ctrl-C

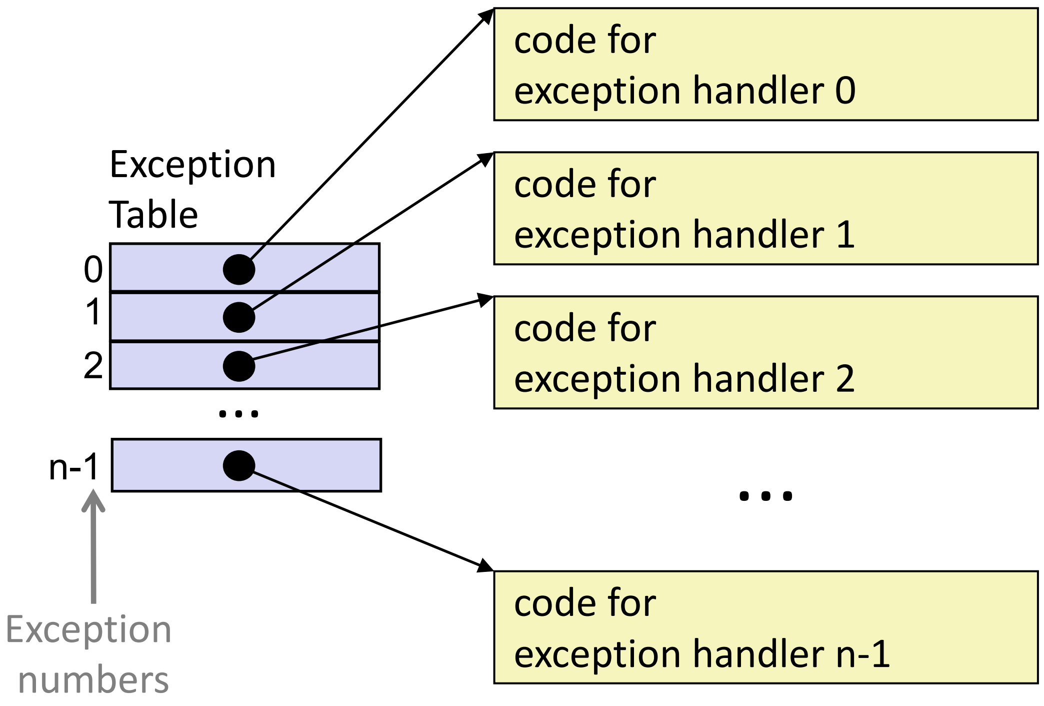


- How does the system know where to jump to in the OS?

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

Leading Up to Processes

- System Control Flow
  - Control flow
  - Exceptional control flow
  - **Asynchronous exceptions (interrupts)**
  - **Synchronous exceptions (traps & faults)**

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
    - Every few milliseconds, an external timer chip triggers an interrupt
    - Used by the OS kernel to take back control from user programs

Synchronous Exceptions (Review)

- Caused by events that occur as a result of executing an instruction:
  - Traps *(why is it called this?)*
    - **Intentional**: transfer control to OS to perform some function
    - Examples: *system calls*, breakpoint traps, special instructions
    - Returns control to "next" instruction, because we wanted it to happen ❤️
  - Faults
    - **Unintentional** but possibly recoverable
    - 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 the current program

System Calls

These are **not the same** as exception numbers!

- Each system call has a unique ID number
- Examples for Linux on x86-64:

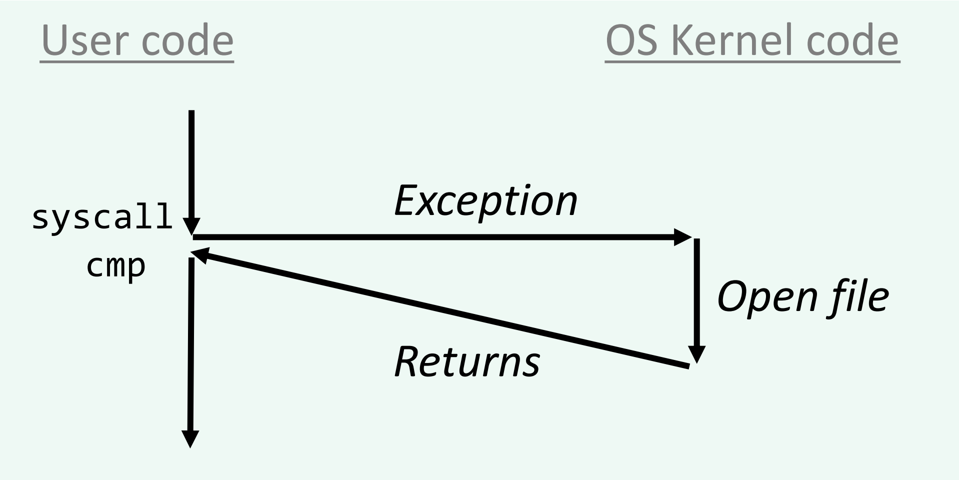
Group	Number	Name	Description
Files	0	`read`	Read file
Files	1	`write`	Write file
Files	2	`open`	Open file
Files	3	`close`	Close file
Files	4	`stat`	Get info about file
Processes	57	`fork`	Create process
Processes	59	`execve`	Execute a program
Processes	60	`_exit`	Terminate process
Processes	62	`kill`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

Traps Example: Opening File

The OS has to help the user program to do this.

- 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
```



- `%(r|e)ax` contains syscall number (weird…)
- Other arguments in `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8`, `%r9`
- Return value in `%rax`
- Negative value in %rax is an error corresponding to negative `errno`

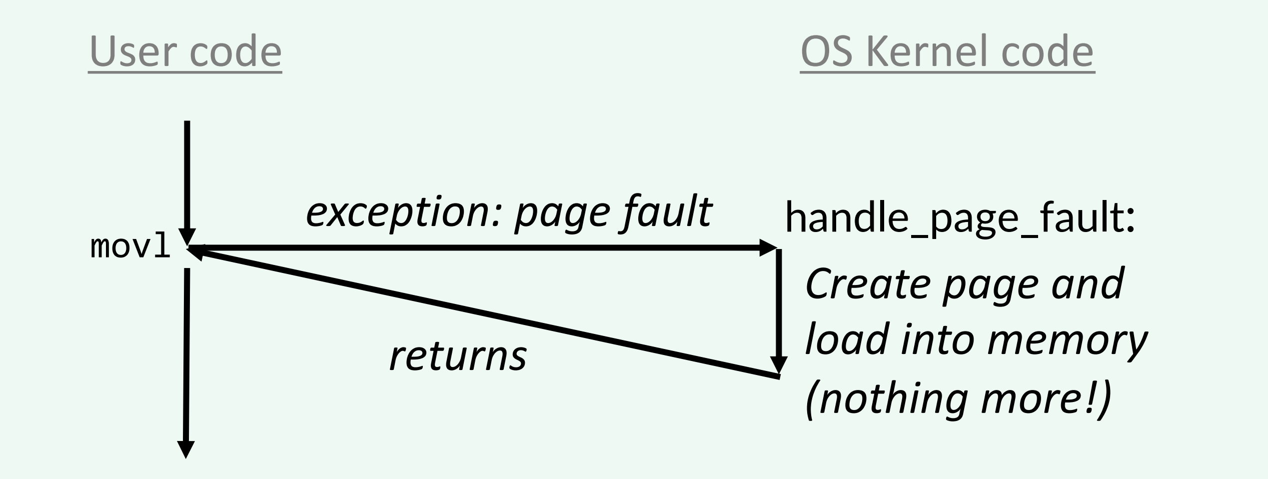
Fault Example: Page Fault

Code:

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

- User writes to memory location
- That portion (page) of user's memory is currently on disk and not in memory

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



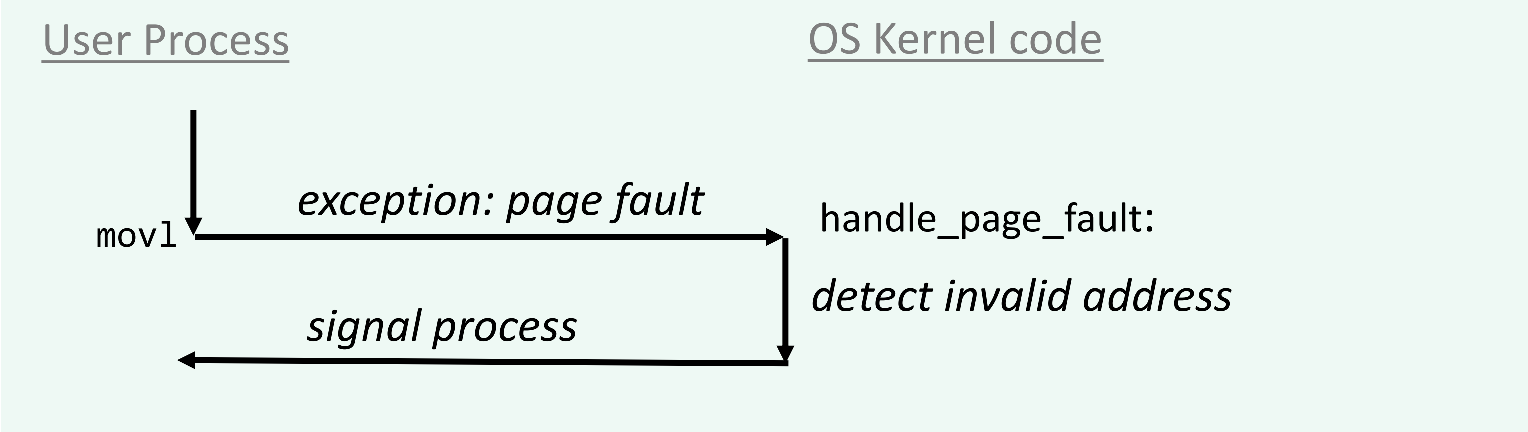
- Page fault handler must load page into physical memory
- Returns to faulting instruction: `mov` **is executed again**!
  - Successful on second try

Abort Example: Invalid Memory Reference

Code:

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

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

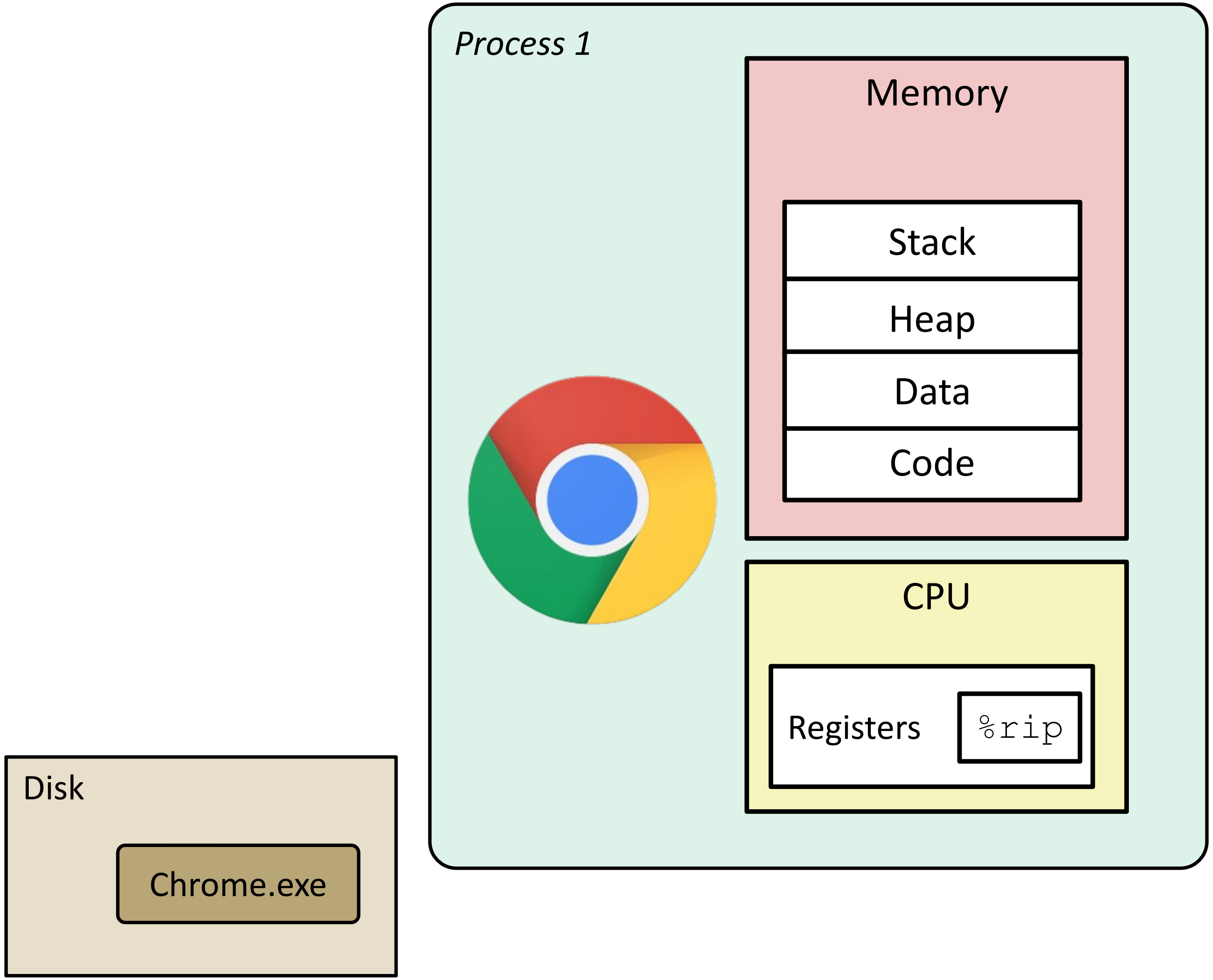


- Page fault handler detects invalid address
- Sends `SIGSEGV` signal to user process
- User process exits with "segmentation fault"

Processes

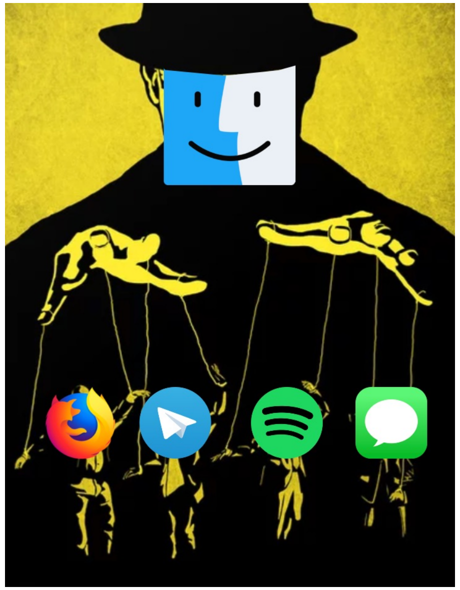
- **Processes and context switching**
- Creating new processes
  - `fork()`, `exec*()`, and `wait()`
- Zombies

What is a process? (Review)



It's an illusion, though!

What is a process? (Review)

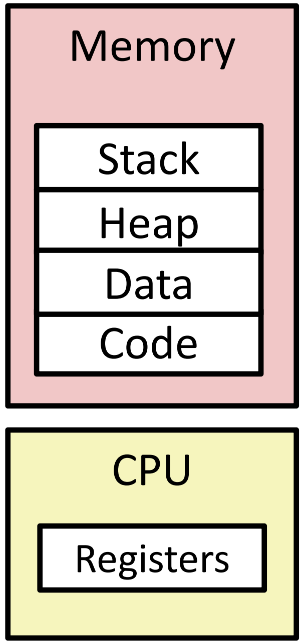


- A **process** is an **instance of a running program**
  - One of the most profound ideas in computer science
- Another abstraction in our computer
  - Provided by the OS
  - OS uses a data structure to represent each process (contains process ID (PID), etc.)
  - Maintains the **interface** between the program and the underlying hardware (CPU + memory)
- What is the difference between:
  - A processor? A program? A process?

Hardware "Blueprint" → an instance!

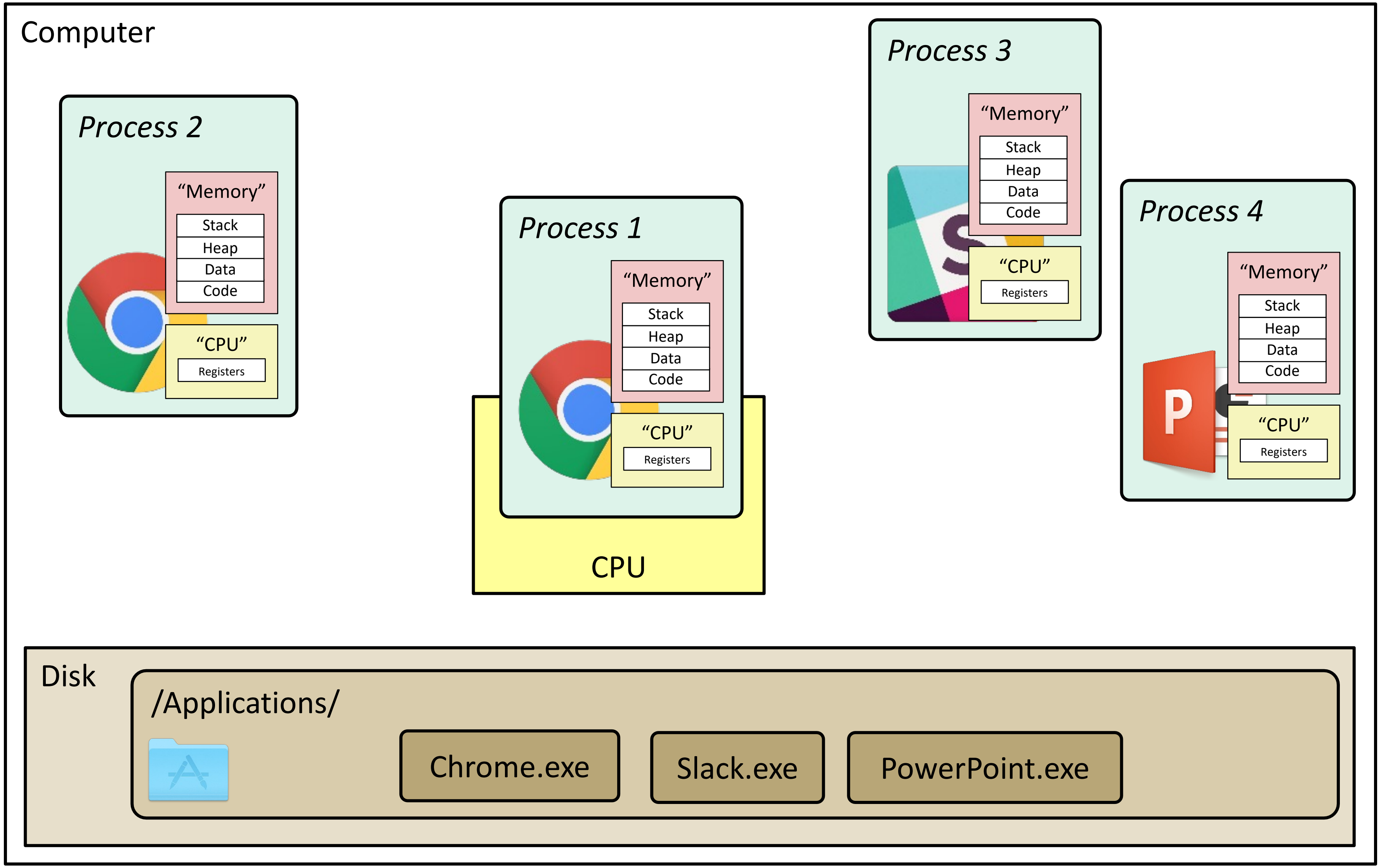
Processes (Review)

- 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***
- What do processes have to do with exceptional control flow?
  - Exceptional control flow is the *mechanism* the OS uses to enable multiple processes to run on the same system



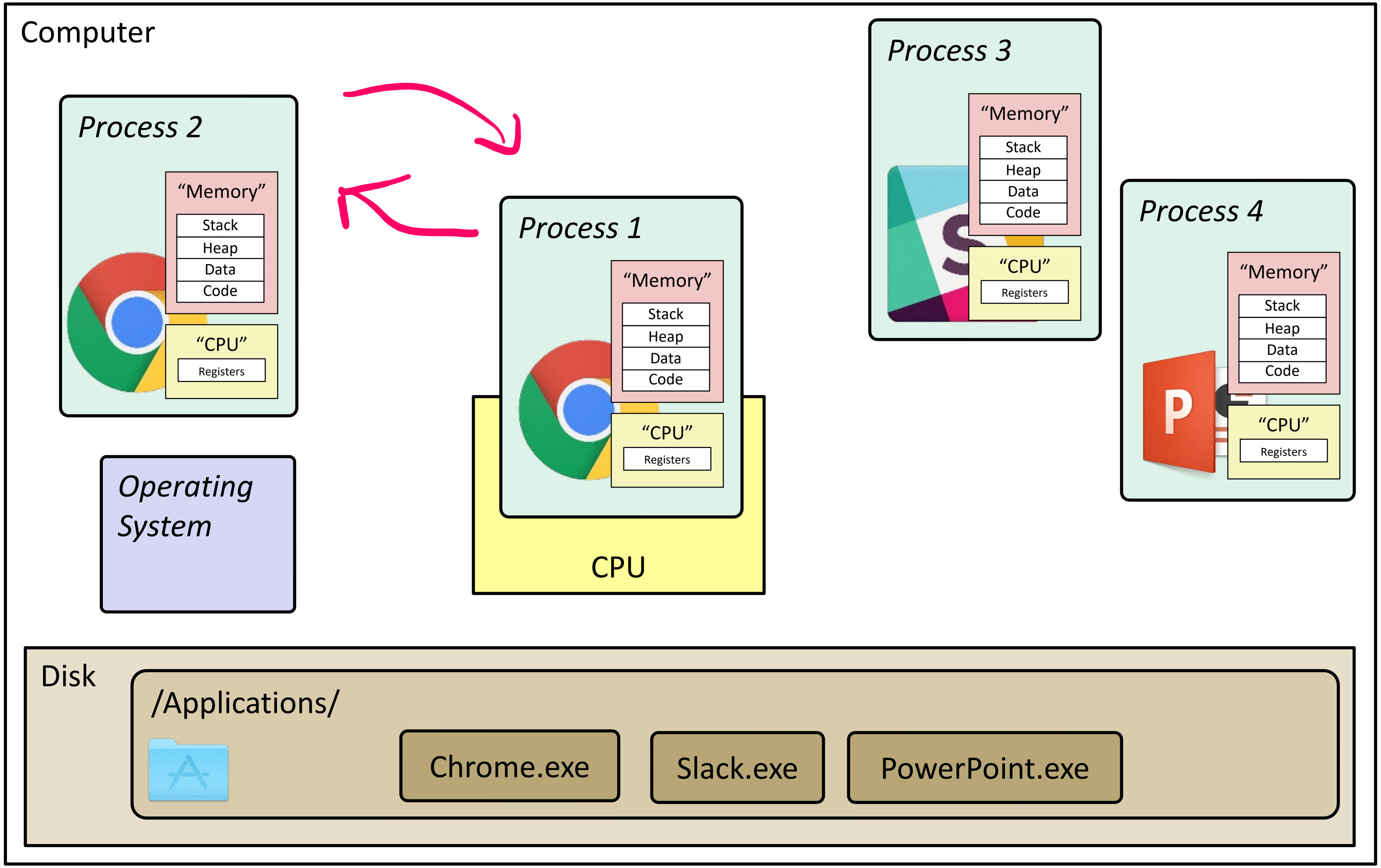
What is a process?

It's an *illusion*!



What is a process?

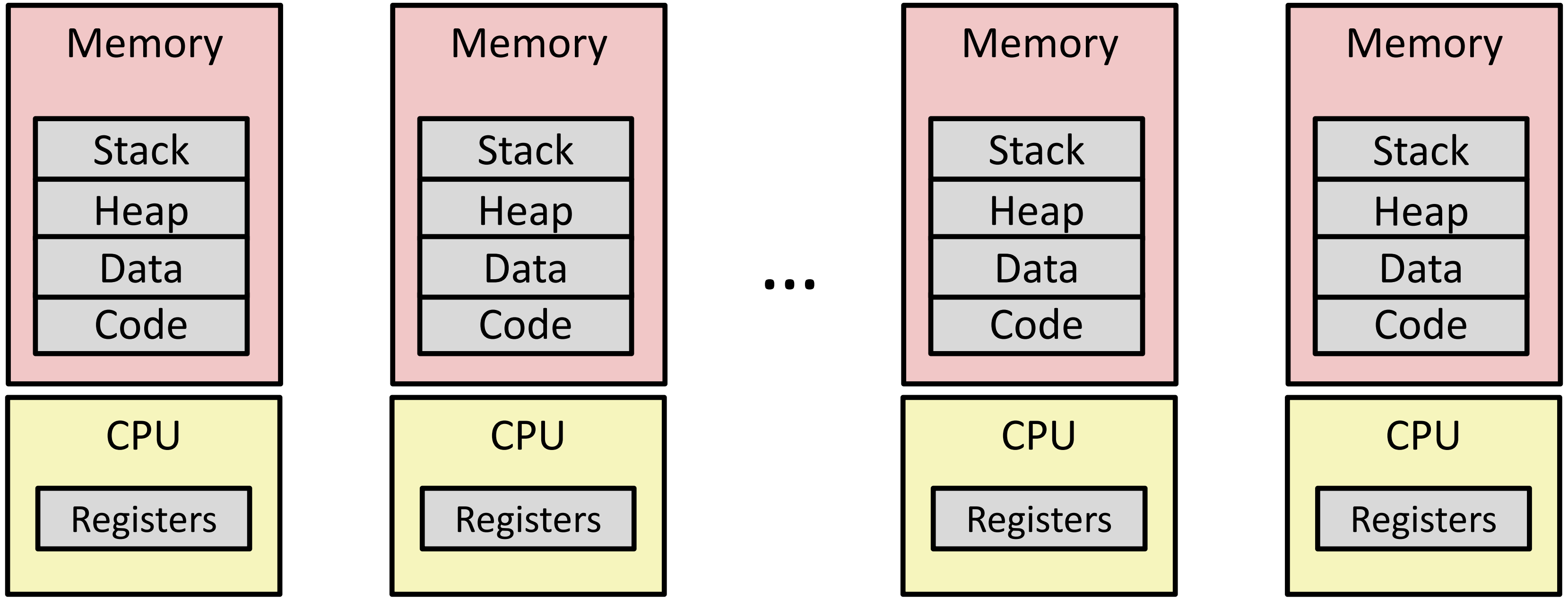
It's an *illusion*!



Multiprocessing: The Illusion

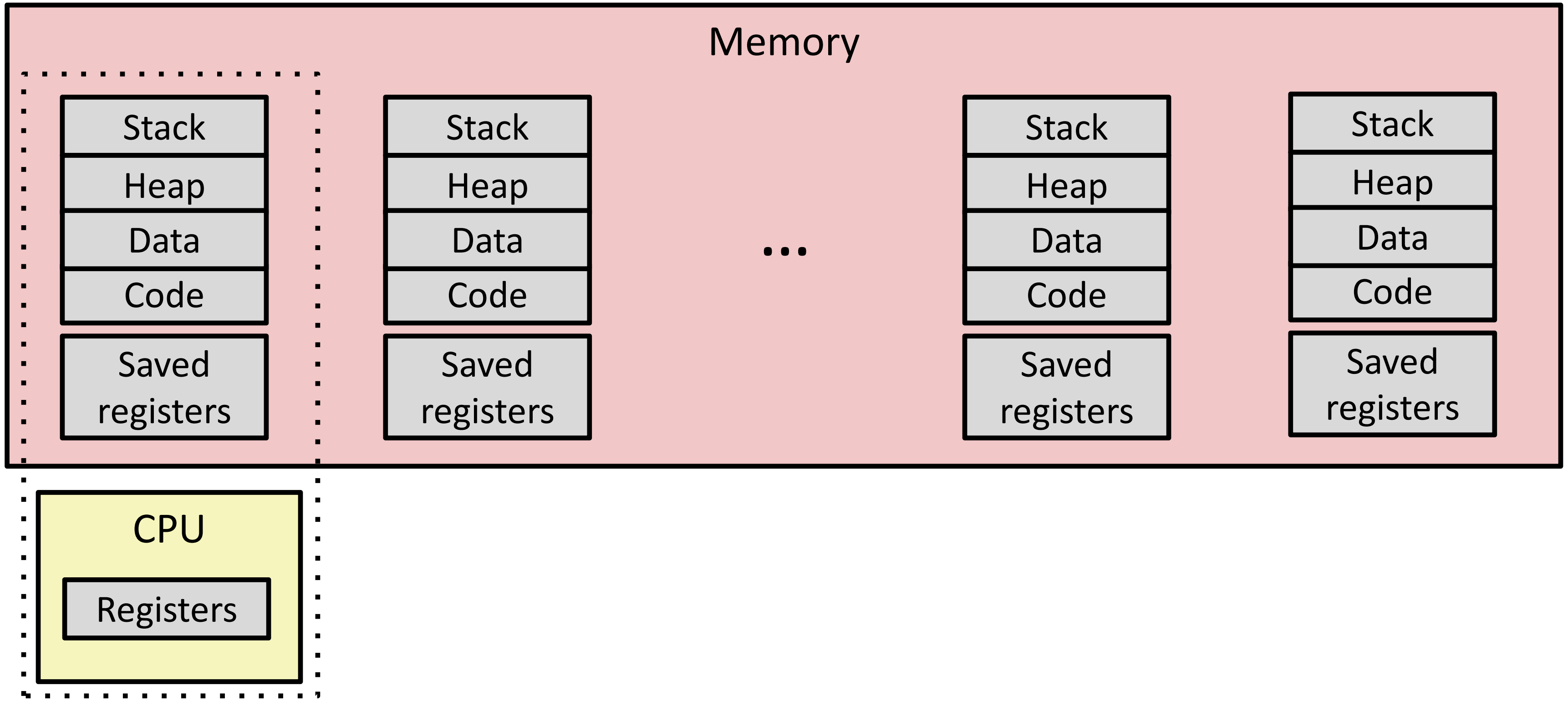
So much memory? No...

So many CPUs? No...



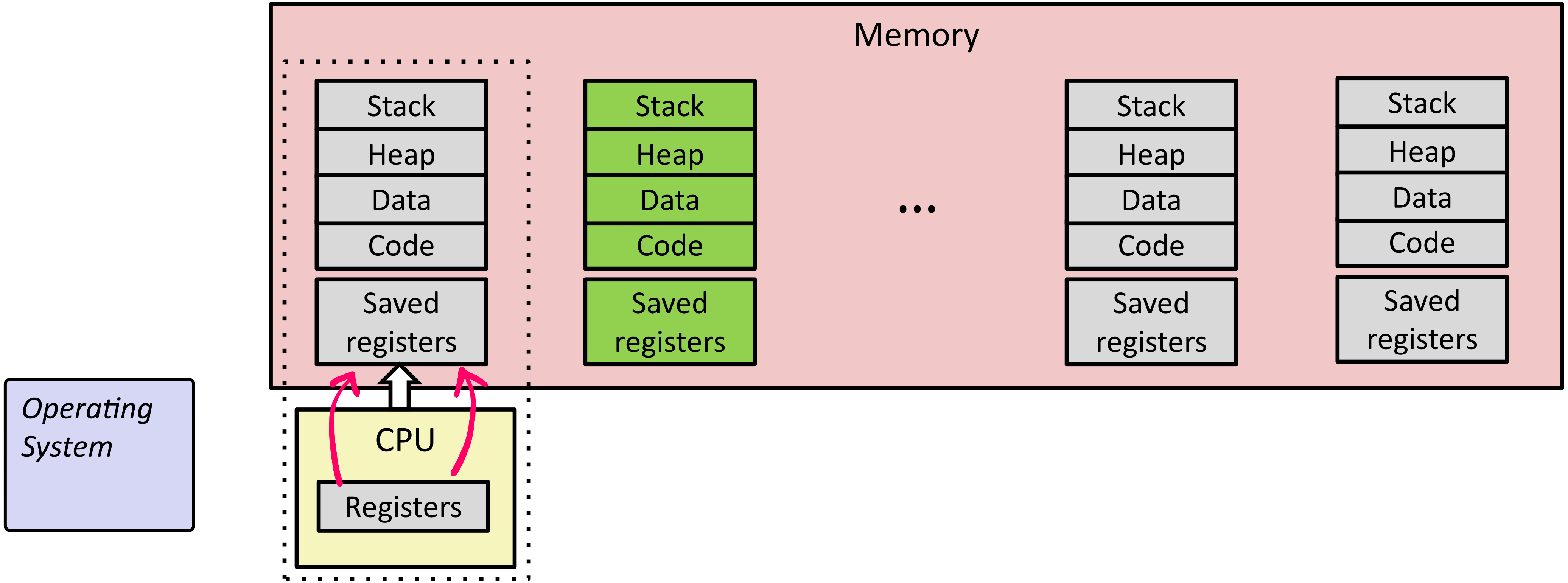
- Computer runs many processes simultaneously
  - Applications for one or more users
    - Web browsers, email clients, editors, …
  - Background tasks
    - Monitoring network & I/O devices

Multiprocessing: The Reality



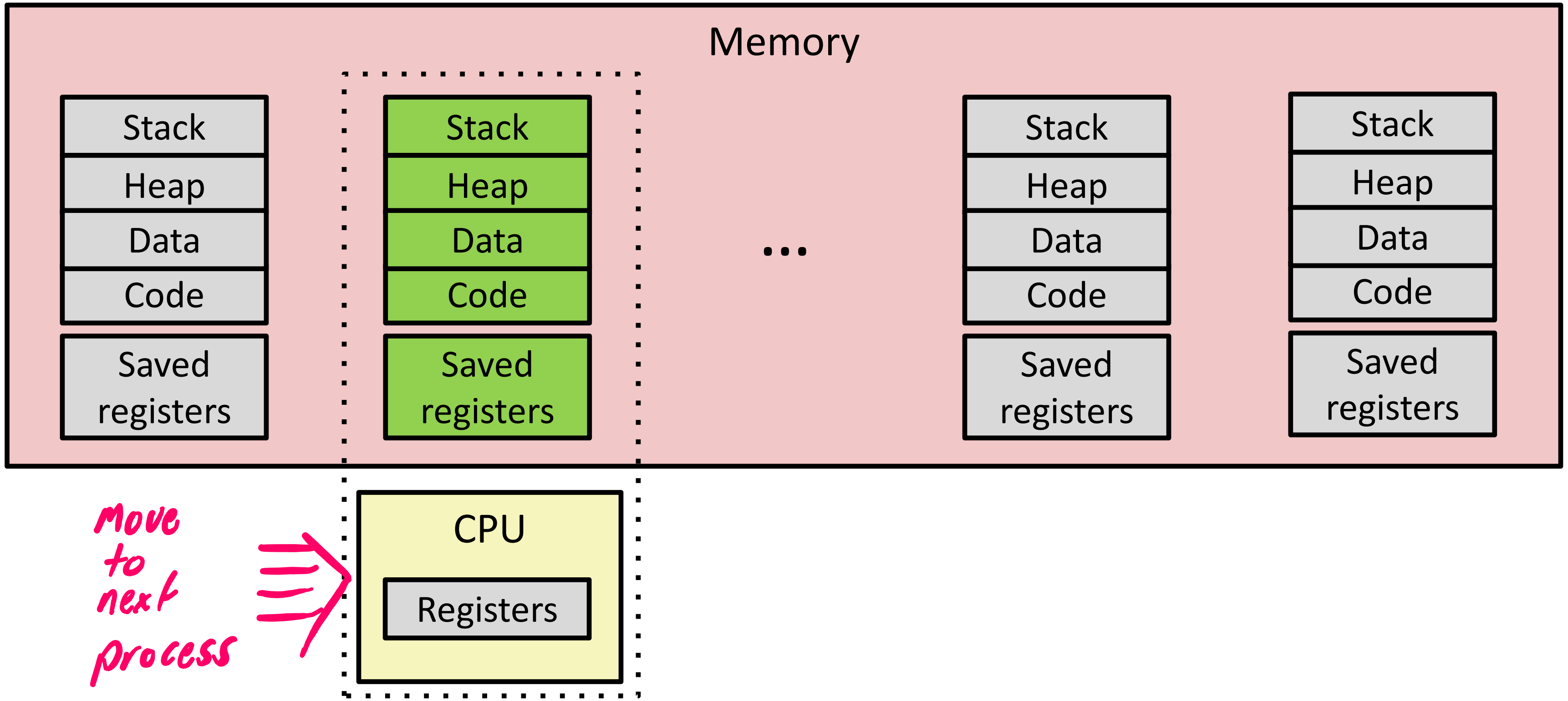
- Single processor executes multiple processes concurrently
  - Process executions interleaved, CPU runs *one at a time*
  - Address spaces managed by virtual memory system (we'll get to it!)
  - **Execution context** (register values, stack, …) for other processes saved in memory

Multiprocessing (Review)



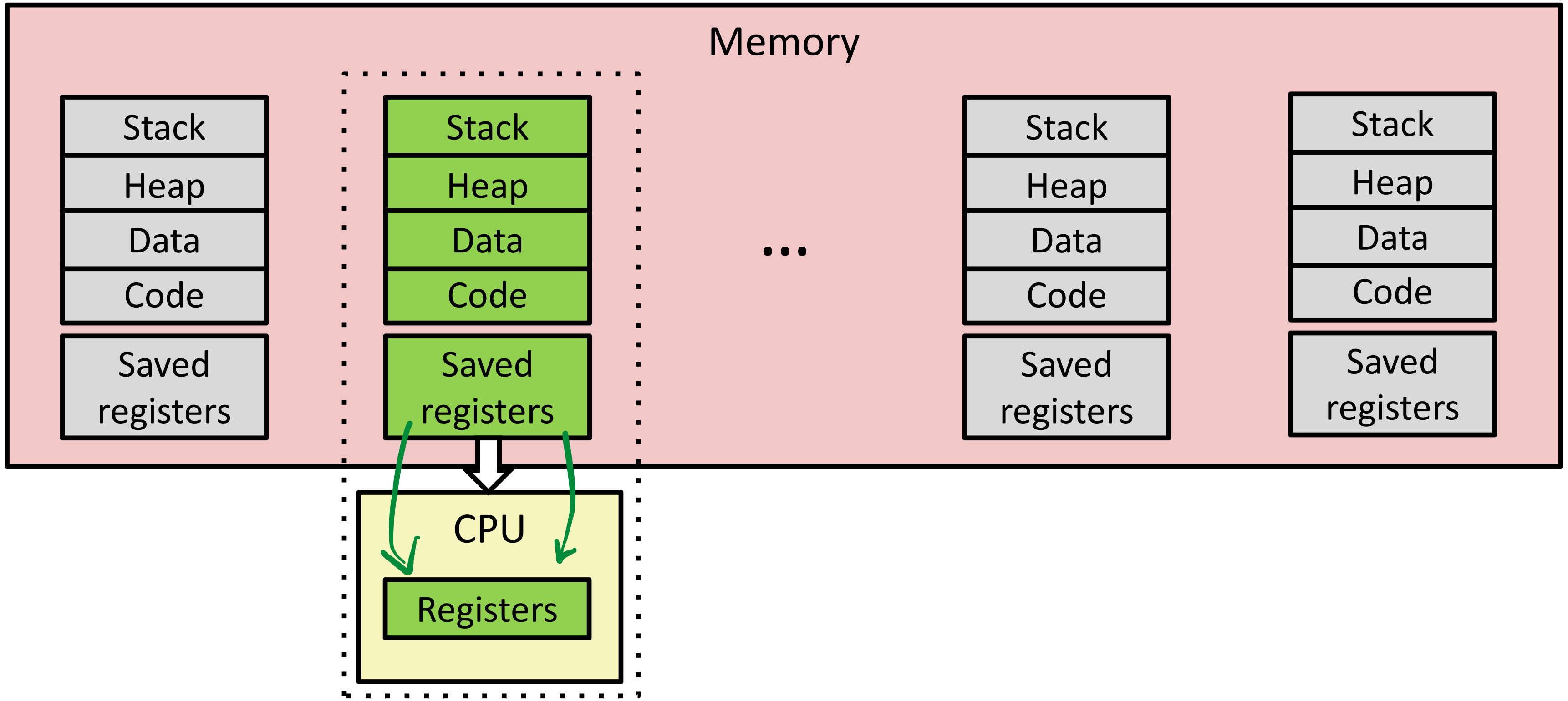
- Context switch
  1) **Save current registers in memory**

Multiprocessing (Review)



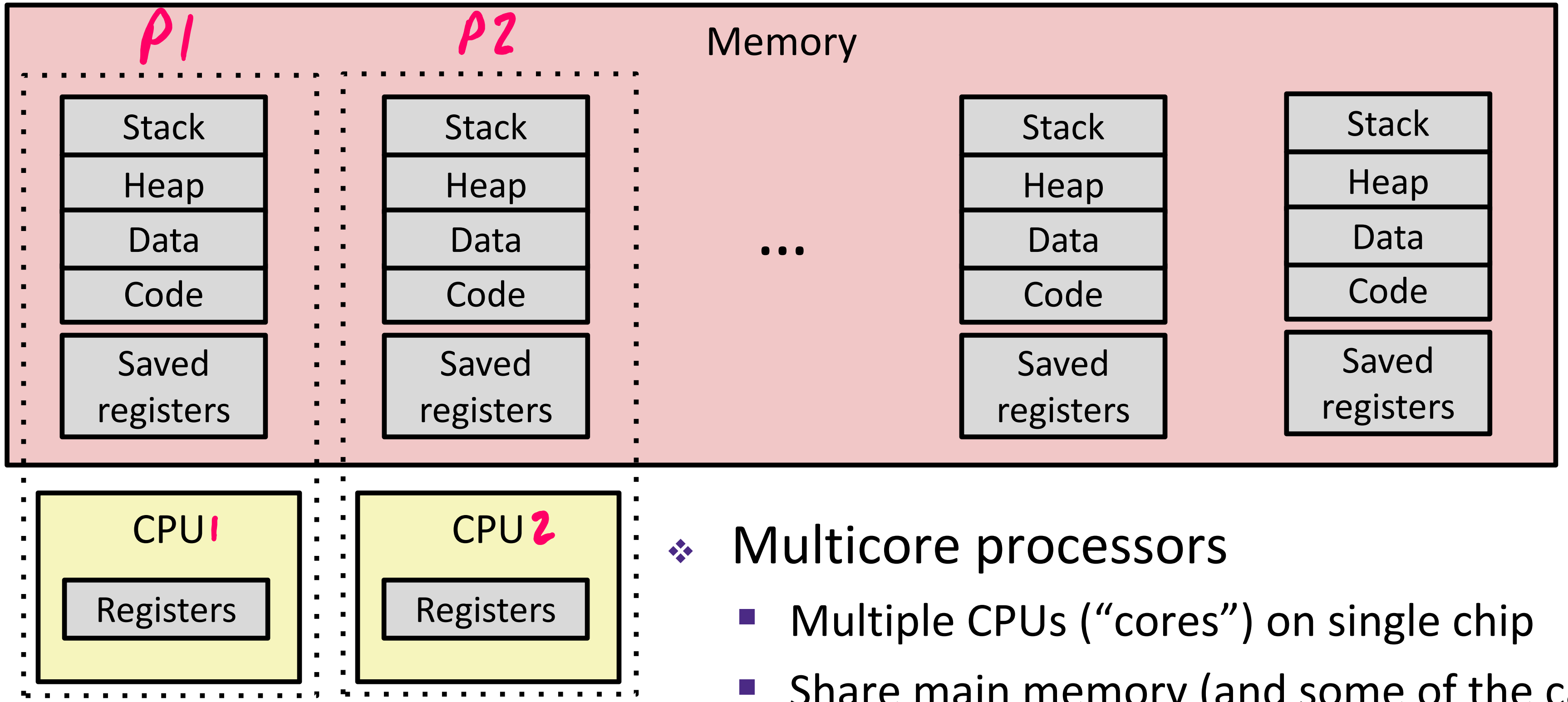
- Context switch
  1) Save current registers in memory
  2) **Schedule next process for execution**

Multiprocessing (Review)



- 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

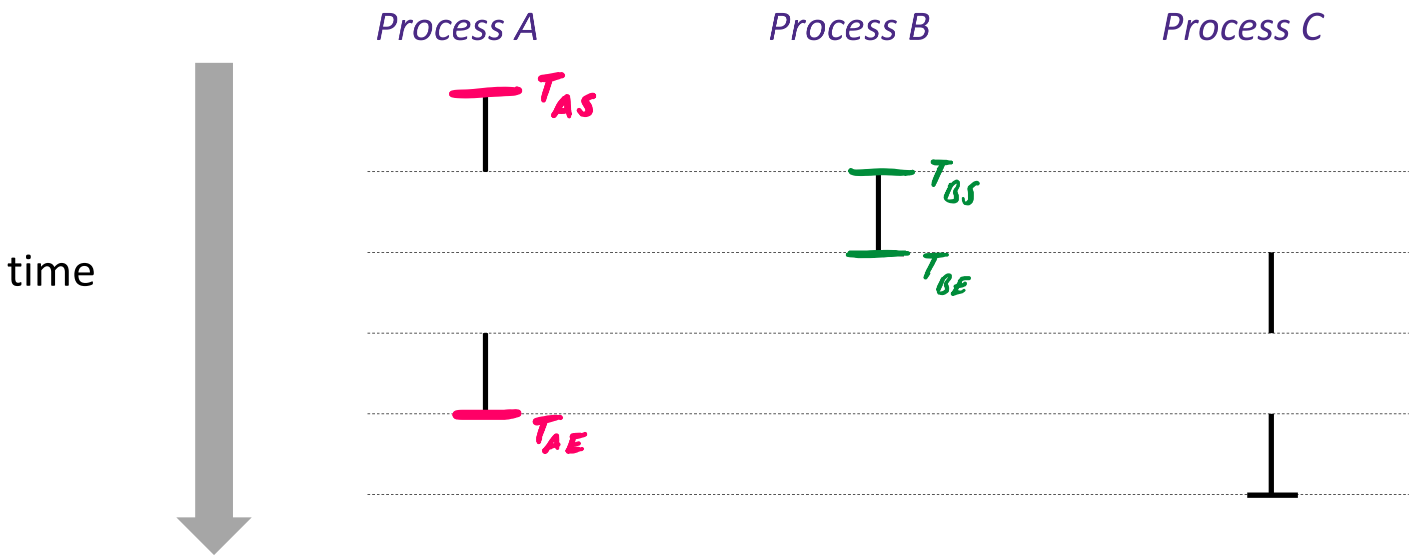


- Multicore processors
  - Multiple CPUs ("cores") on single chip
  - Share main memory (and some of the caches)
  - Each can execute a separate process
    - Kernel schedules processes to cores
    - **<u>Still</u> constantly swapping processes**

Concurrent Processes

Assume only one CPU core

- Each process is a logical control flow
- Two processes run **concurrently** (are concurrent) if their instruction executions/flows overlap in time → $[T_{AS} < T_{BS}$ && $T_{AE} > T_{BE}]$
  - Otherwise, they are **sequential**
- **Example**: (running on single core)
  - Concurrent: A & B, A & C
  - Sequential: B & C

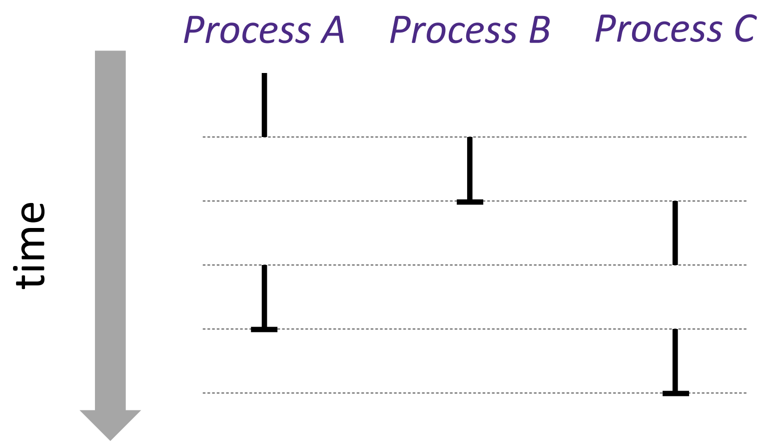


User's View of Concurrency

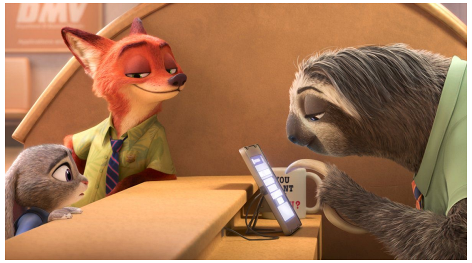
Assume only one CPU core

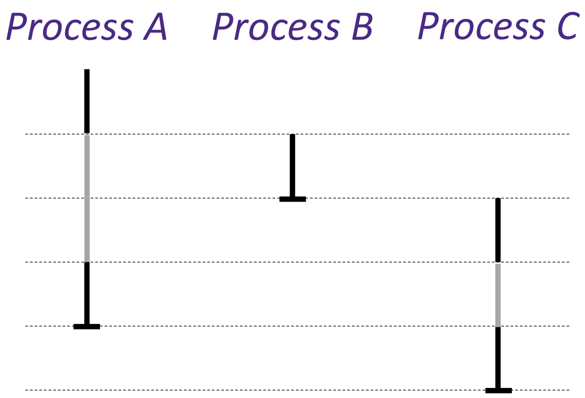
- Control flows for concurrent processes are physically disjoint in time
  - CPU only executes instructions for one process at a time
- However, the user can ***think** of* concurrent processes as executing at the same time, in parallel

Because we humans can't really tell!



User View

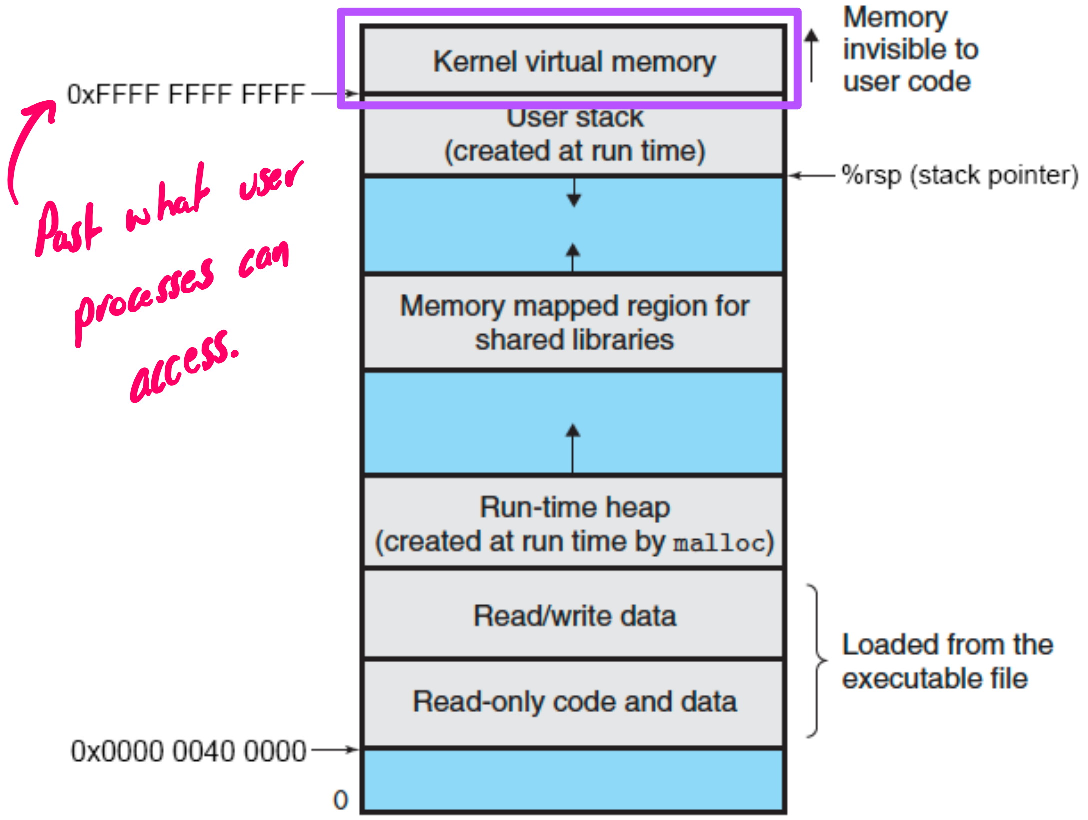




Context Switching

Assume only one CPU core

- 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*

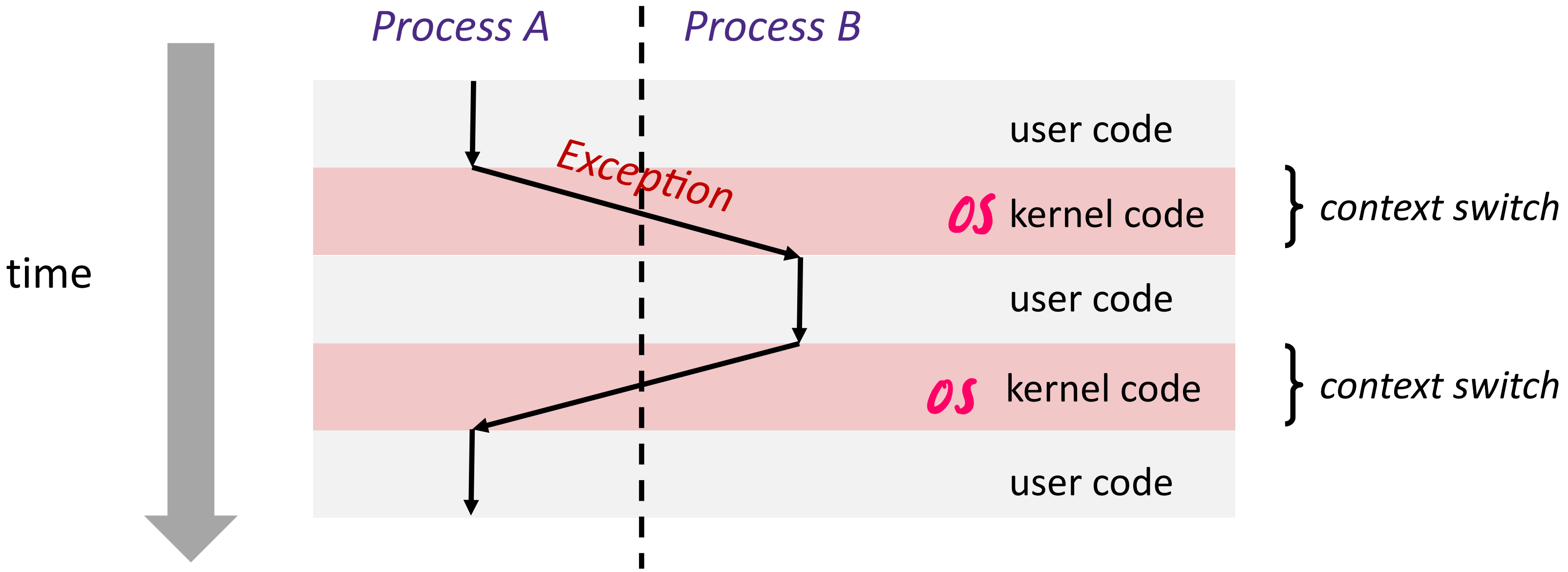


* sort of, the story here became more complicated after Meltdown and Spectre… Roughly since 2018…

Context Switching (Review)

Assume only one CPU core

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



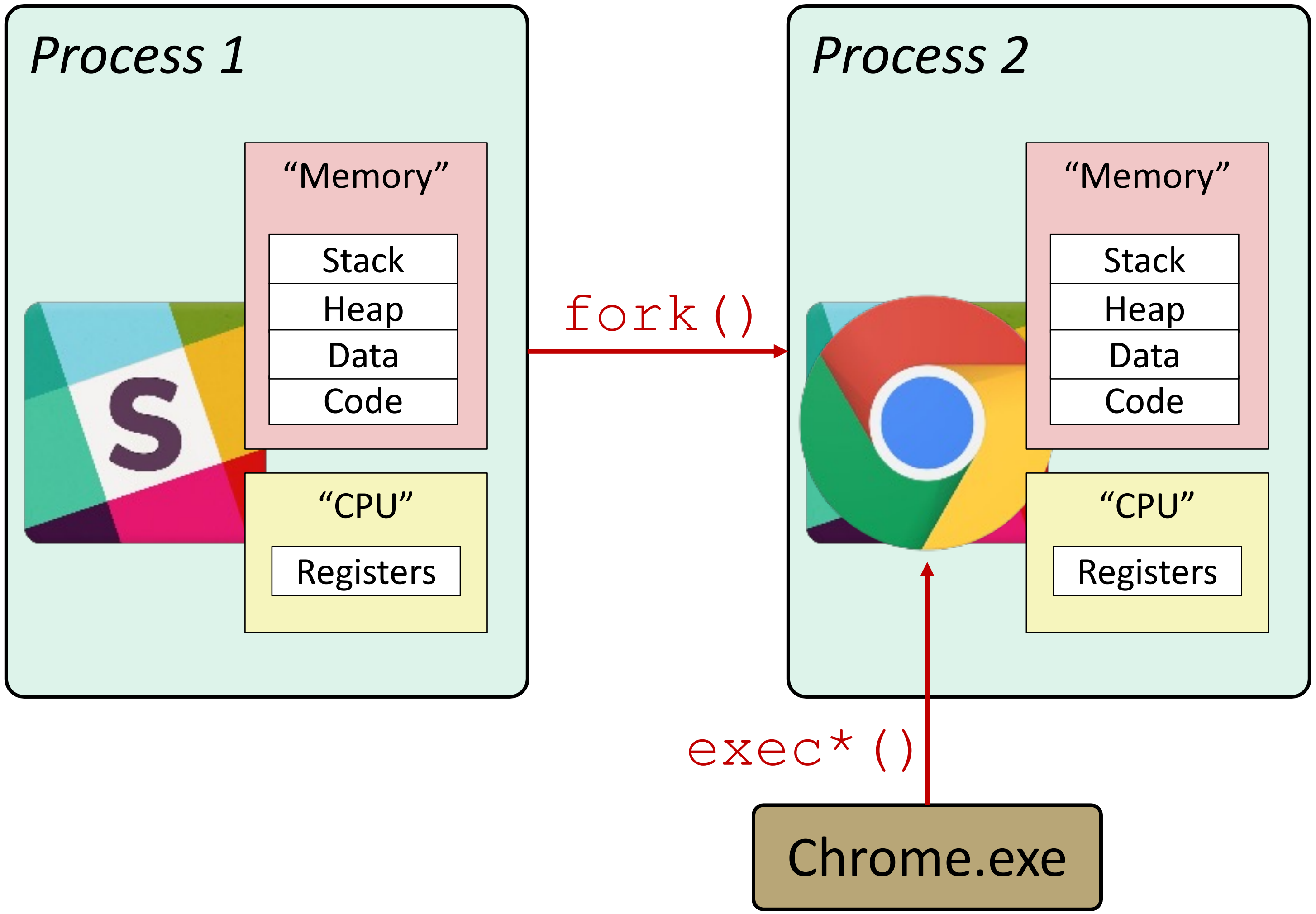
Processes & Context Switching Summary

- Exceptions
  - Events that require non-standard control flow
  - Generated asynchronously (interrupts) or synchronously (traps and faults)
  - After an exception is handled, either:
    - Re-execute the current instruction
    - Resume execution with the next instruction
    - Abort the process that caused the exception
- Processes
  - Only one of many active processes executes at a time on a CPU, but each appears to have total control of the processor
  - OS periodically "context switches" between active processes

Processes

- Processes and context switching
- **Creating new processes**
  - **`fork()` and `exec*()`**
- Ending a process
  - `exit()`, `wait()`, `waitpid()`
  - Zombies

Creating New Processes & Programs



Creating New Processes & Programs

- fork-exec model (Linux):
  - `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()`

Summary

- Exceptions
  - Events that require non-standard control flow
  - Generated asynchronously (interrupts) or synchronously (traps and faults)
  - After an exception is handled, either:
    - Re-execute the current instruction
    - Resume execution with the next instruction
    - Abort the process that caused the exception
- Processes
  - Only one of many active processes executes at a time on a CPU, but each appears to have total control of the processor
  - OS periodically "context switches" between active processes