

# Processes

CSE 351 Autumn 2019

## Instructor:

Justin Hsia

## Teaching Assistants:

Andrew Hu

Diya Joy

Maurice Montag

Suraj Jagadeesh

Antonio Castelli

Ivy Yu

Melissa Birchfield

Cosmo Wang

Kaelin Laundry

Millicent Li

REFRESH TYPE	EXAMPLE SHORTCUTS	EFFECT
SOFT REFRESH	EMAIL <input type="button" value="REFRESH"/> BUTTON	REQUESTS UPDATE WITHIN JAVASCRIPT
NORMAL REFRESH	F5, CTRL-R, ⌘R	REFRESHES PAGE
HARD REFRESH	CTRL-F5, CTRL-⇧, ⌘⇧R	REFRESHES PAGE INCLUDING CACHED FILES
HARDER REFRESH	CTRL-⇧-HYPER-ESC-R-F5	REMOVELY CYCLES POWER TO DATACENTER
HARDEST REFRESH	CTRL-⌘-⇧-#-R-F5-F5-ESC-O-O-Ø-▲-SCROLL LOCK	INTERNET STARTS OVER FROM ARPANET

# Administrivia

- ❖ hw17 due Friday (11/15)
  - Lab 4 preparation!
- ❖ Lab 4 due next Friday (11/22)

# Roadmap

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

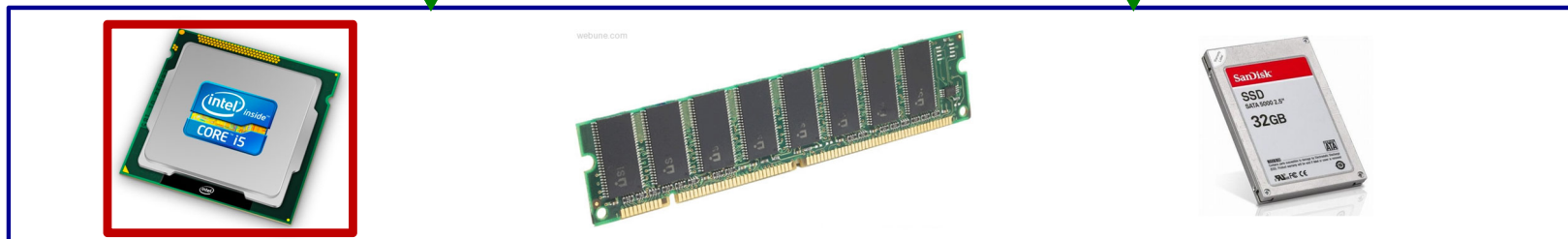
Assembly language:

```
get_mpg:
    pushq    %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

Computer system:



Memory & data  
 Integers & floats  
 x86 assembly  
 Procedures & stacks  
 Executables  
 Arrays & structs  
 Memory & caches  
**Processes**  
 Virtual memory  
 Memory allocation  
 Java vs. C

OS:



# Leading Up to Processes

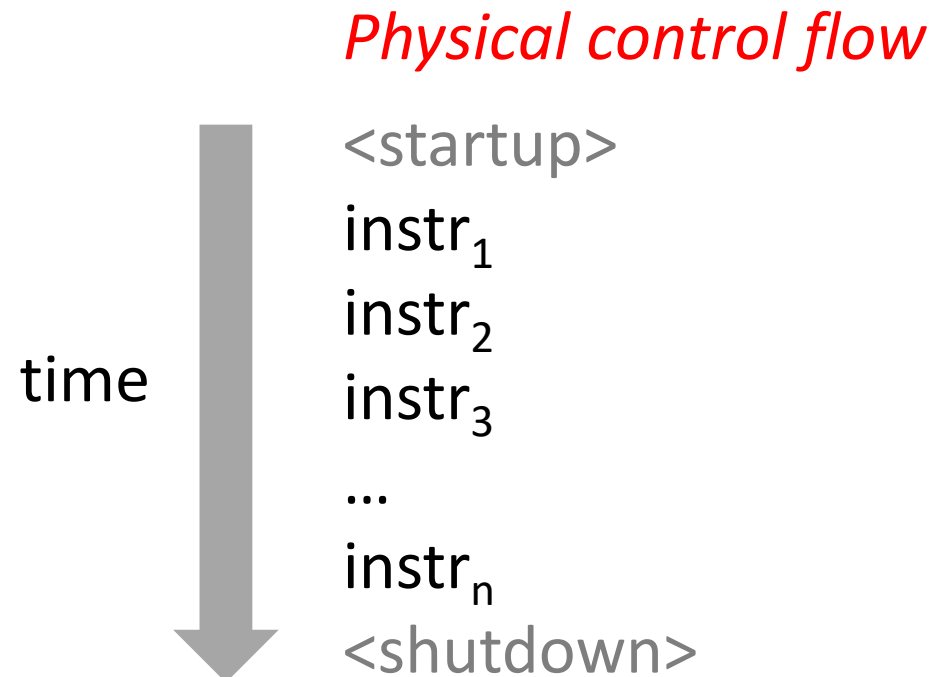
- ❖ System Control Flow
  - **Control flow**
  - **Exceptional control flow**
  - Asynchronous exceptions (interrupts)
  - Synchronous exceptions (traps & faults)

# Control Flow

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



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

# Java Digression

This is extra  
(non-testable)  
material

- ❖ Java has exceptions, but they're *something different*
  - Examples: NullPointerException, MyBadThingHappenedException, ...
  - `throw` statements
  - `try/catch` statements (“throw to youngest matching catch on the call-stack, or exit-with-stack-trace if none”)
- ❖ 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 we already have the CPU features we need
- ❖ System-state changes on previous slide are mostly of a different sort (asynchronous/external except for divide-by-zero) and implemented very differently

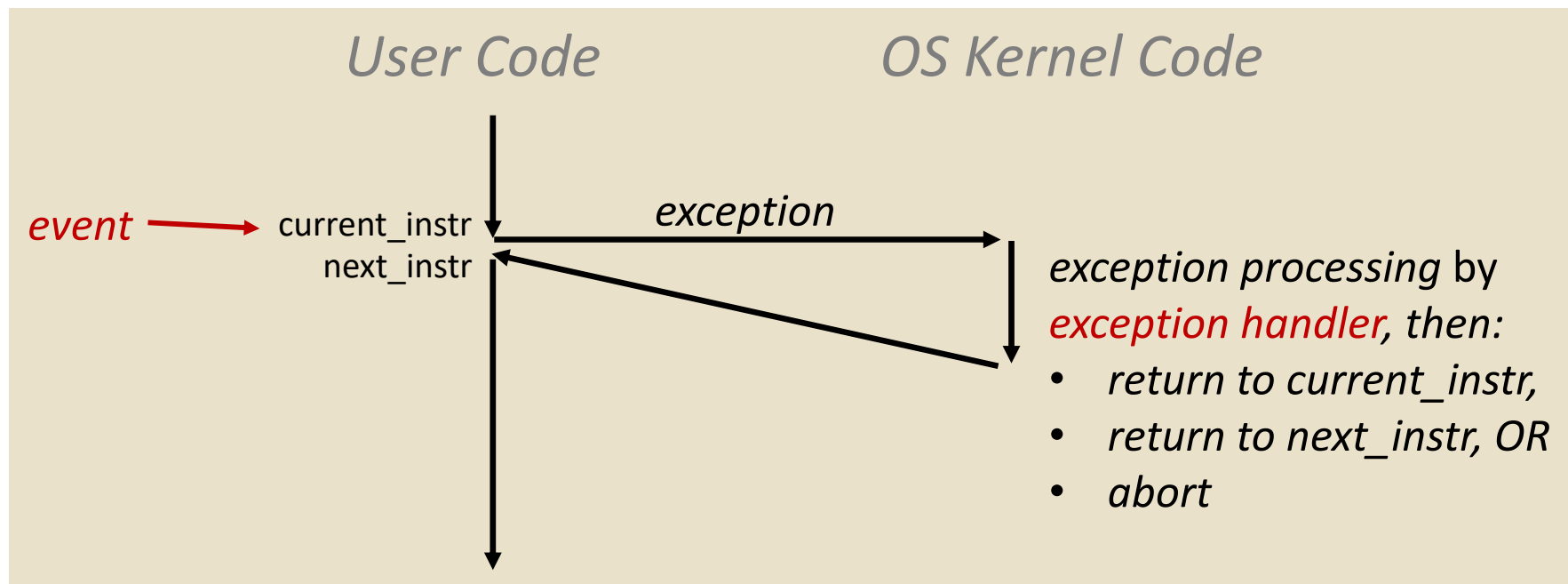


# Exceptional Control Flow

- ❖ 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 – see CSE451 and CSE/EE474

# Exceptions

- ❖ 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 memory-resident part of the OS
  - Examples: division by 0, page fault, I/O request completes, Ctrl-C

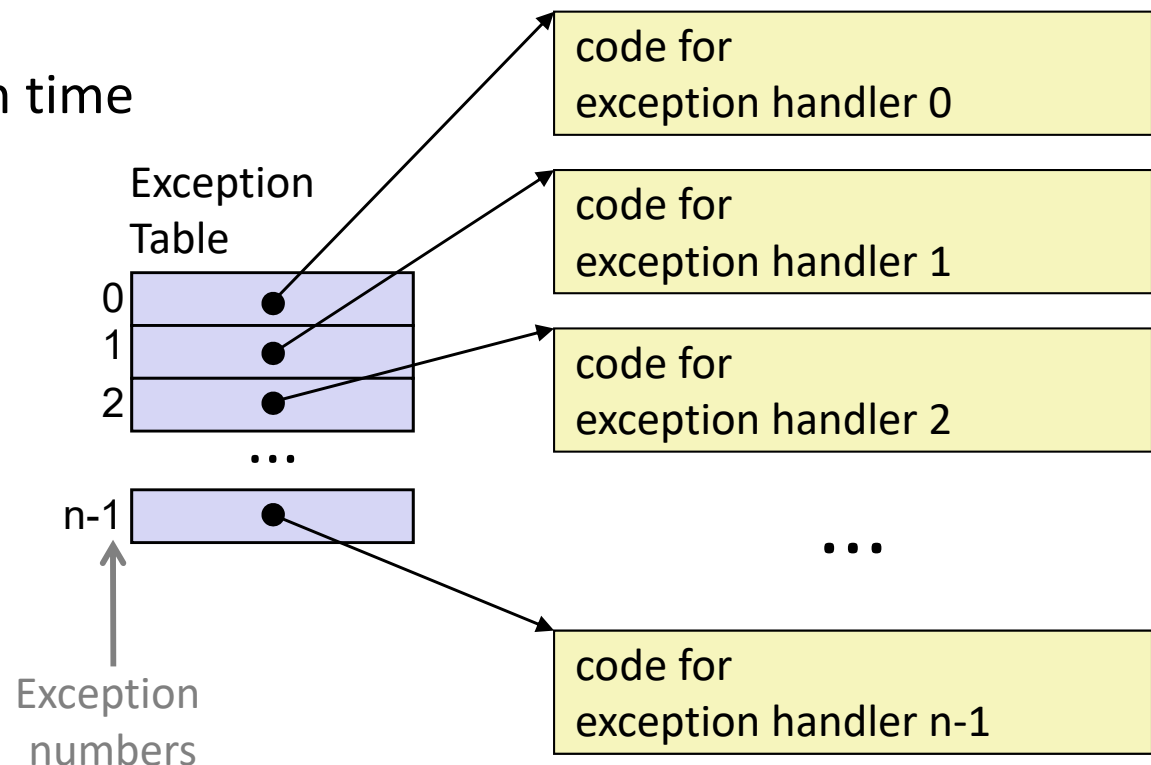


- ❖ *How does the system know where to jump to in the OS?*

# 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

<i>Exception Number</i>	<i>Description</i>	<i>Exception Class</i>
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 (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

- ❖ Caused by events that occur as a result 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
    - 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

# System Calls

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

<i>Number</i>	<i>Name</i>	<i>Description</i>
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



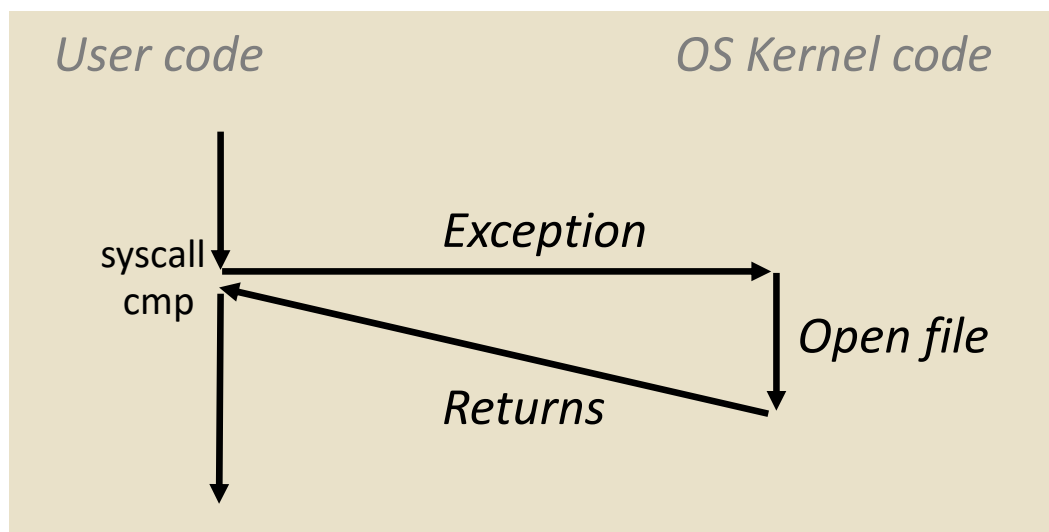
# Traps Example: Opening File

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

```

000000000000e5d70 <__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  $0xffffffffffffffff001,%rax
...
e5dfa:  c3                retq

```



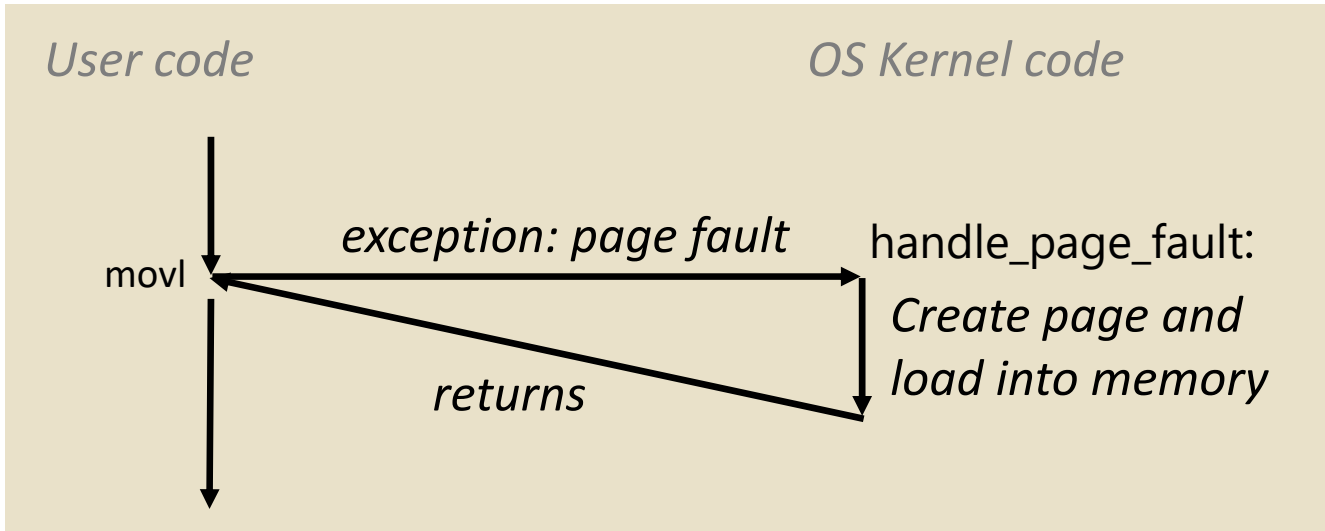
- `%rax` contains syscall number
- Other arguments in `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8`, `%r9`
- Return value in `%rax`
- Negative value is an error corresponding to negative `errno`

# Fault Example: Page Fault

- ❖ User writes to memory location
- ❖ That portion (page) of user's memory is currently on disk

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

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

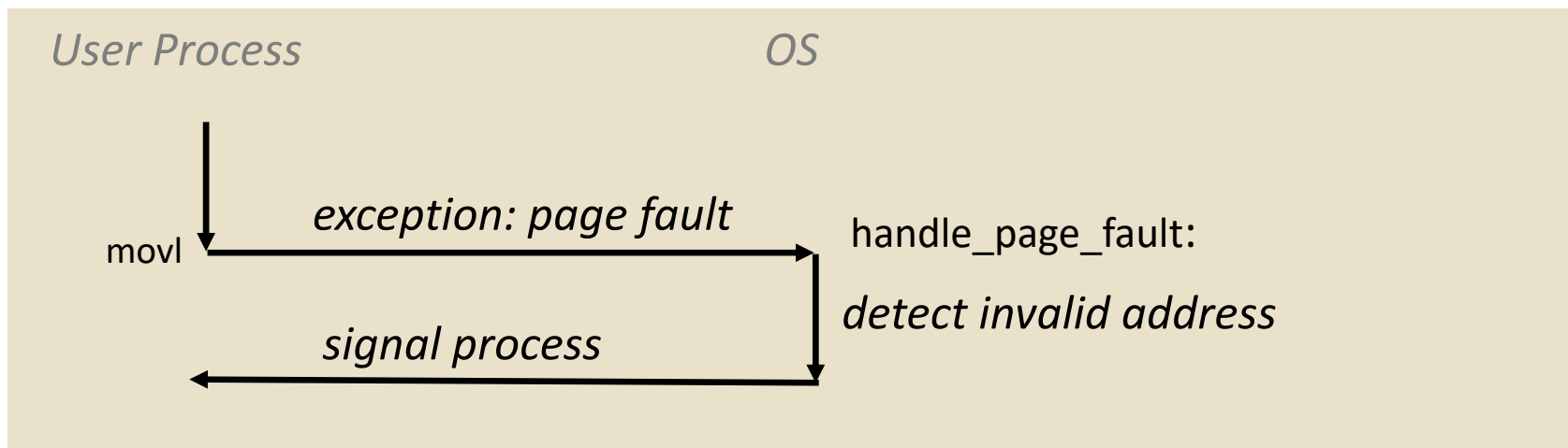


- ❖ Page fault handler must load page into physical memory
- ❖ Returns to faulting instruction: `mov` is executed again!
  - Successful on second try

# Fault Example: Invalid Memory Reference

```
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”

# Summary

## ❖ Exceptions

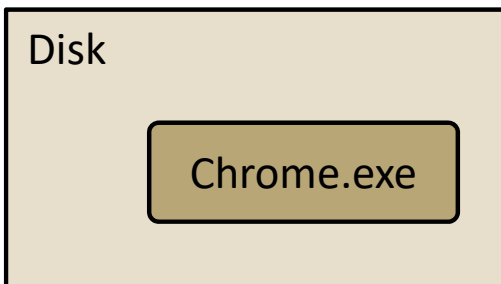
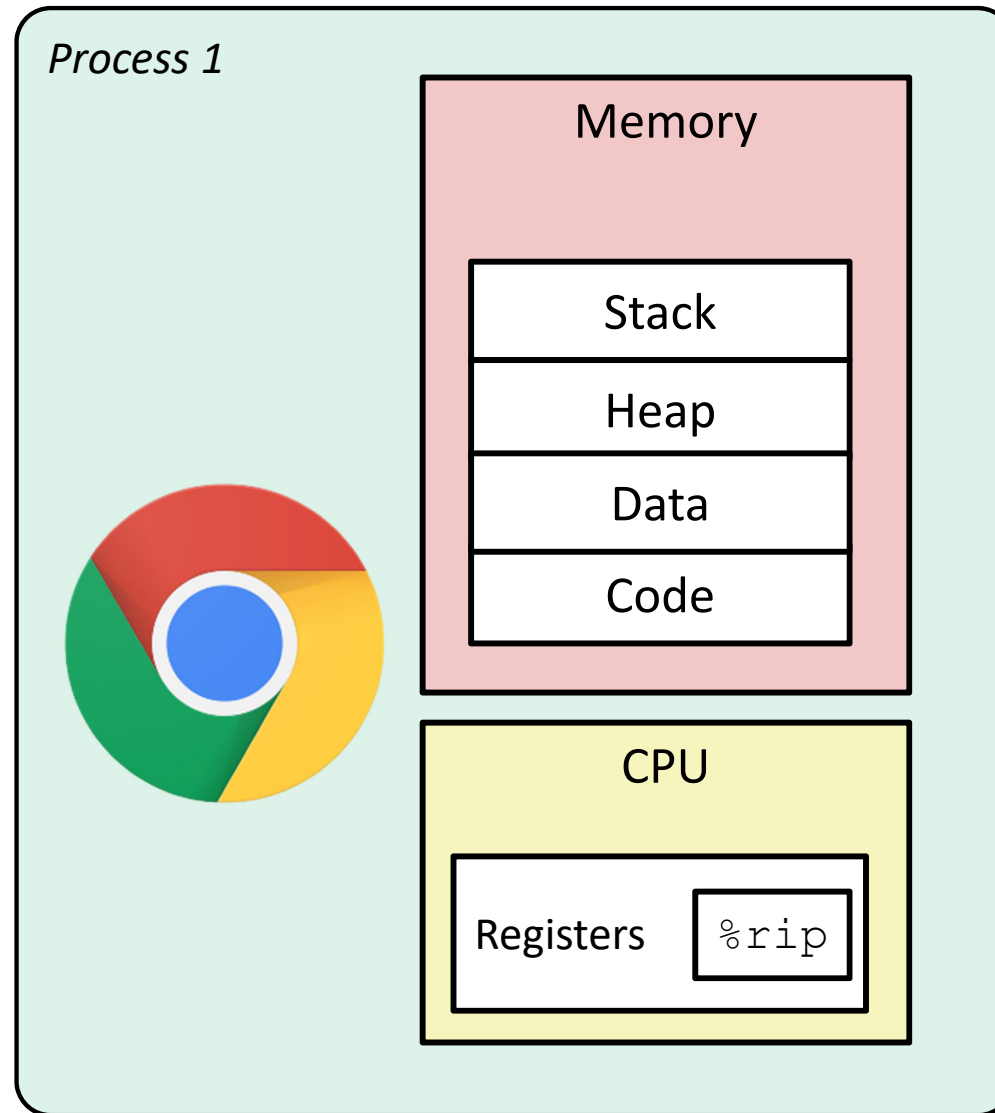
- Events that require non-standard control flow
- Generated externally (interrupts) or internally (traps and faults)
- After an exception is handled, one of three things may happen:
  - Re-execute the current instruction
  - Resume execution with the next instruction
  - Abort the process that caused the exception

# Processes

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

# What is a process?

*It's an illusion!*

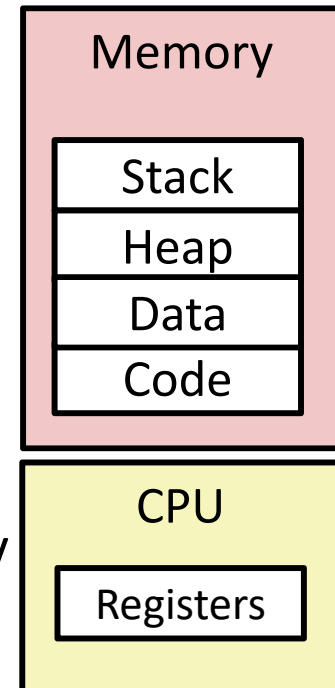


# What is a process?

- ❖ Another *abstraction* in our computer system
  - Provided by the OS
  - OS uses a data structure to represent each process
  - Maintains the *interface* between the program and the underlying hardware (CPU + 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 the difference between:
  - A processor? A program? A process?

# Processes

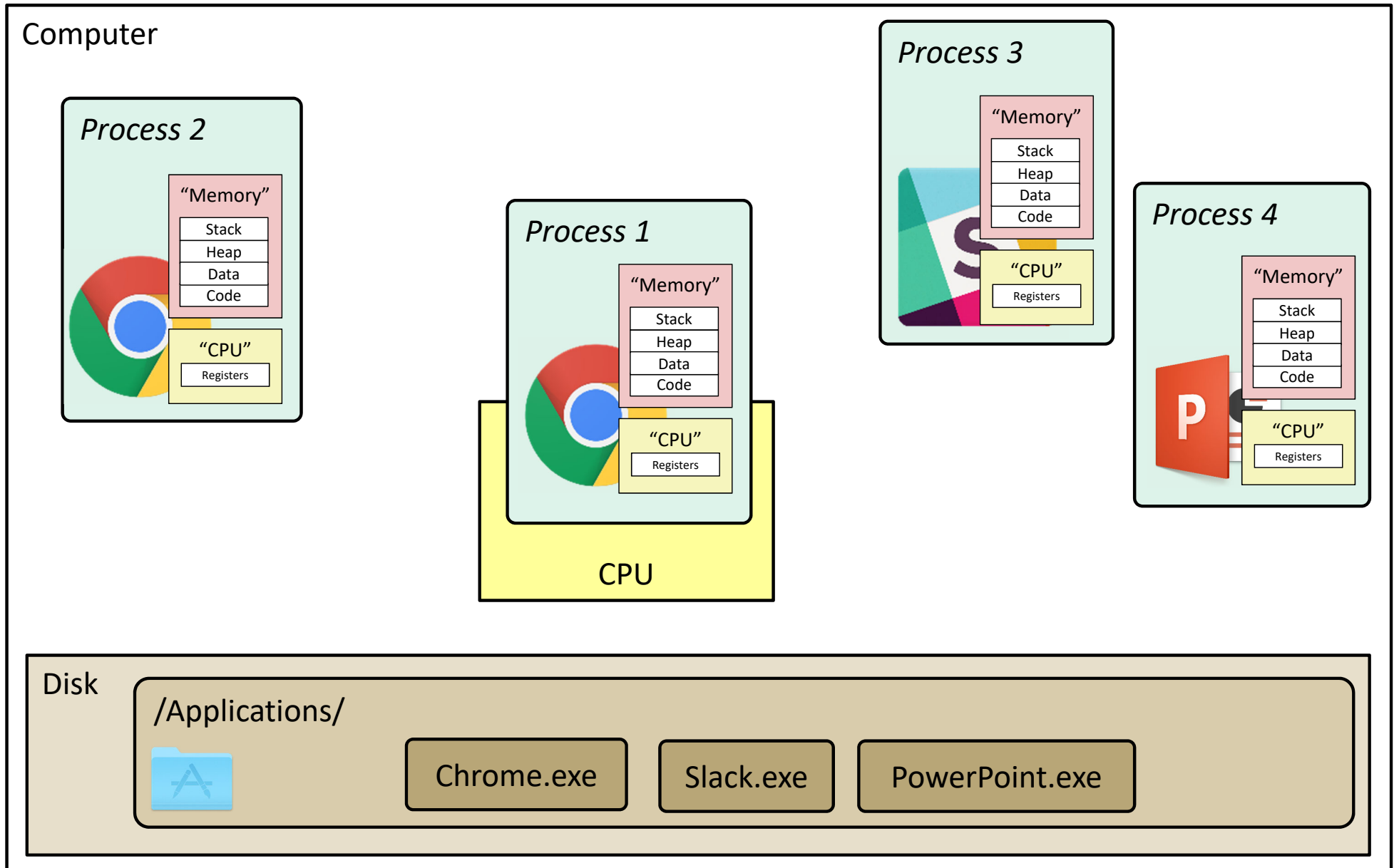
- ❖ A **process** is an instance of a running program
  - One of the most profound ideas in computer science
  - Not the same as “program” or “processor”
- ❖ Process provides each program with two key abstractions:
  - *Logical control flow*
    - Each program seems to have exclusive use of the CPU
    - 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**





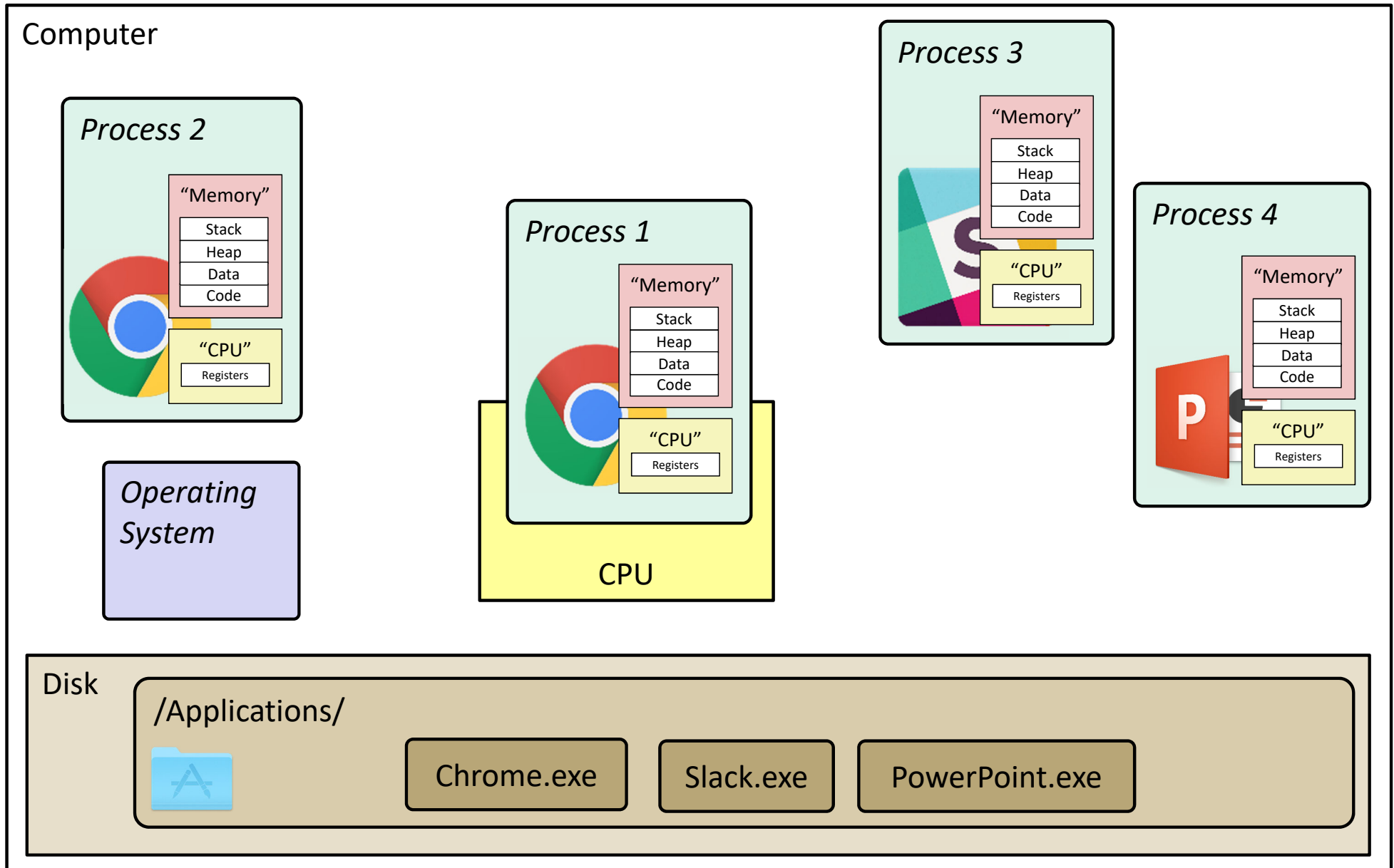
# What is a process?

*It's an illusion!*

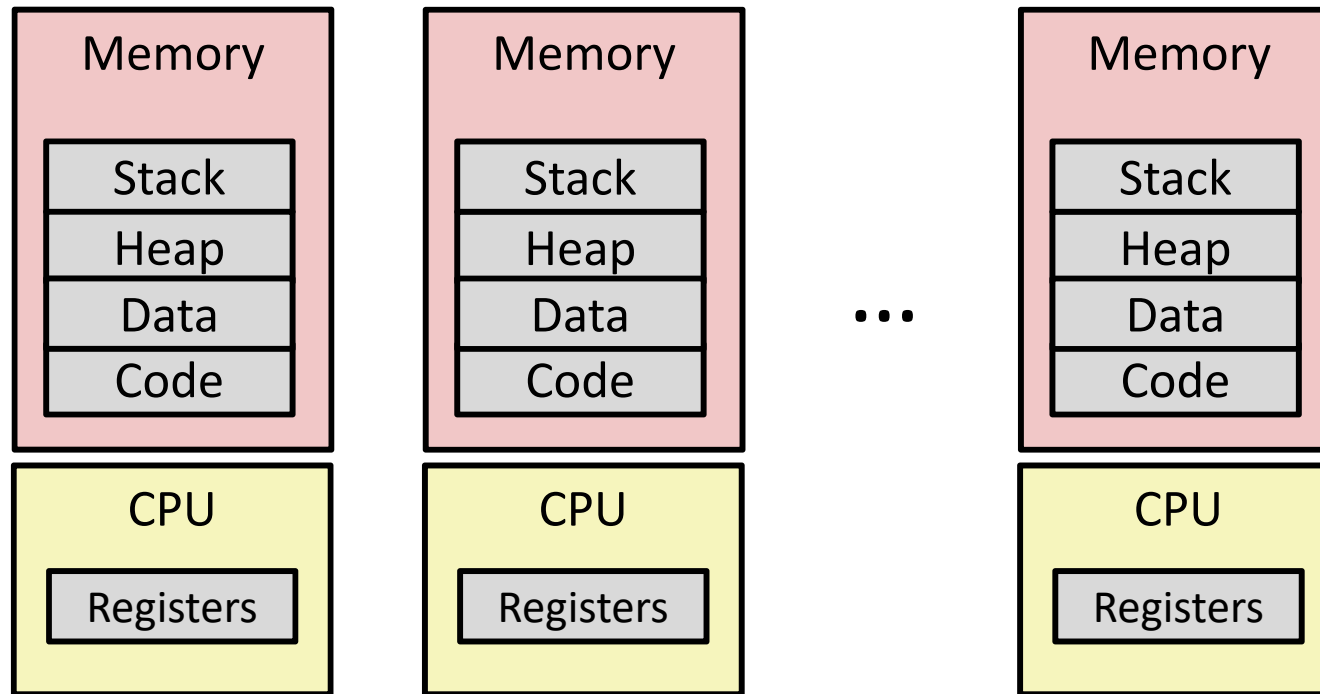


# What is a process?

*It's an illusion!*

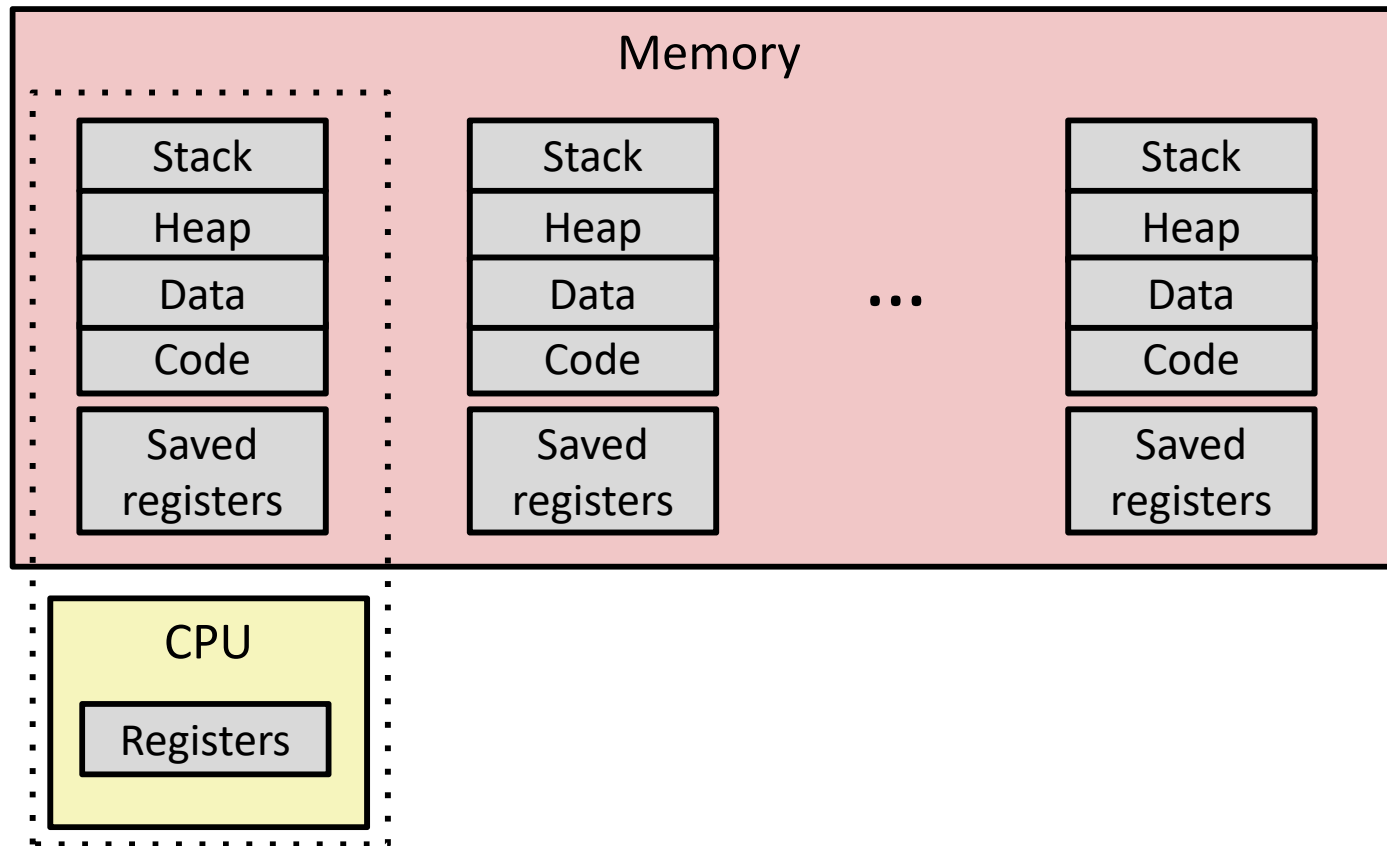


# Multiprocessing: The Illusion



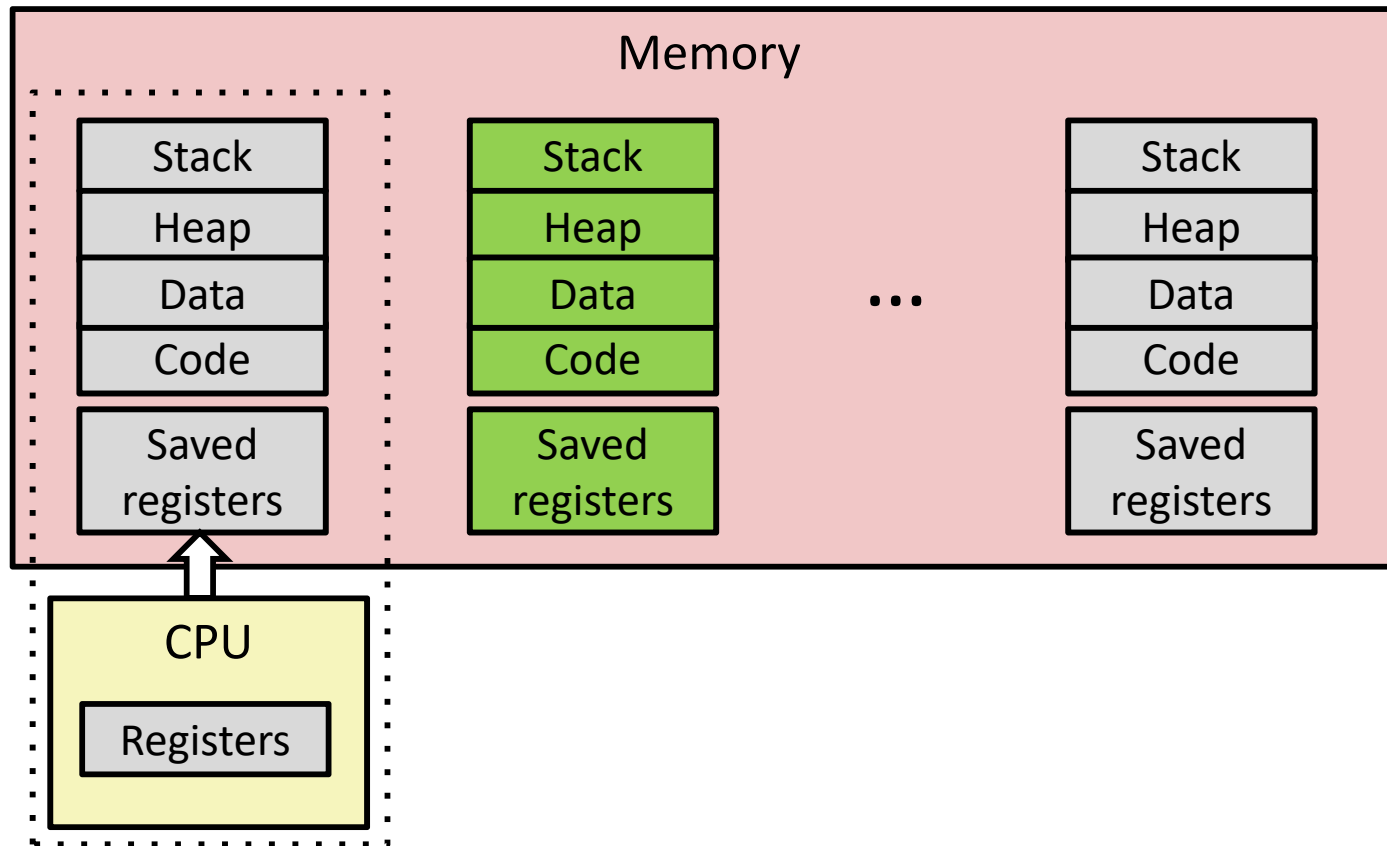
- ❖ Computer runs many processes simultaneously
  - Applications for one or more users
    - Web browsers, email clients, editors, ...
  - Background tasks
    - Monitoring network & I/O devices

# Multiprocessing: The Reality



- ❖ Single processor executes multiple processes *concurrently*
  - Process executions interleaved, CPU runs *one at a time*
  - Address spaces managed by virtual memory system (later in course)
  - *Execution context* (register values, stack, ...) for other processes saved in memory

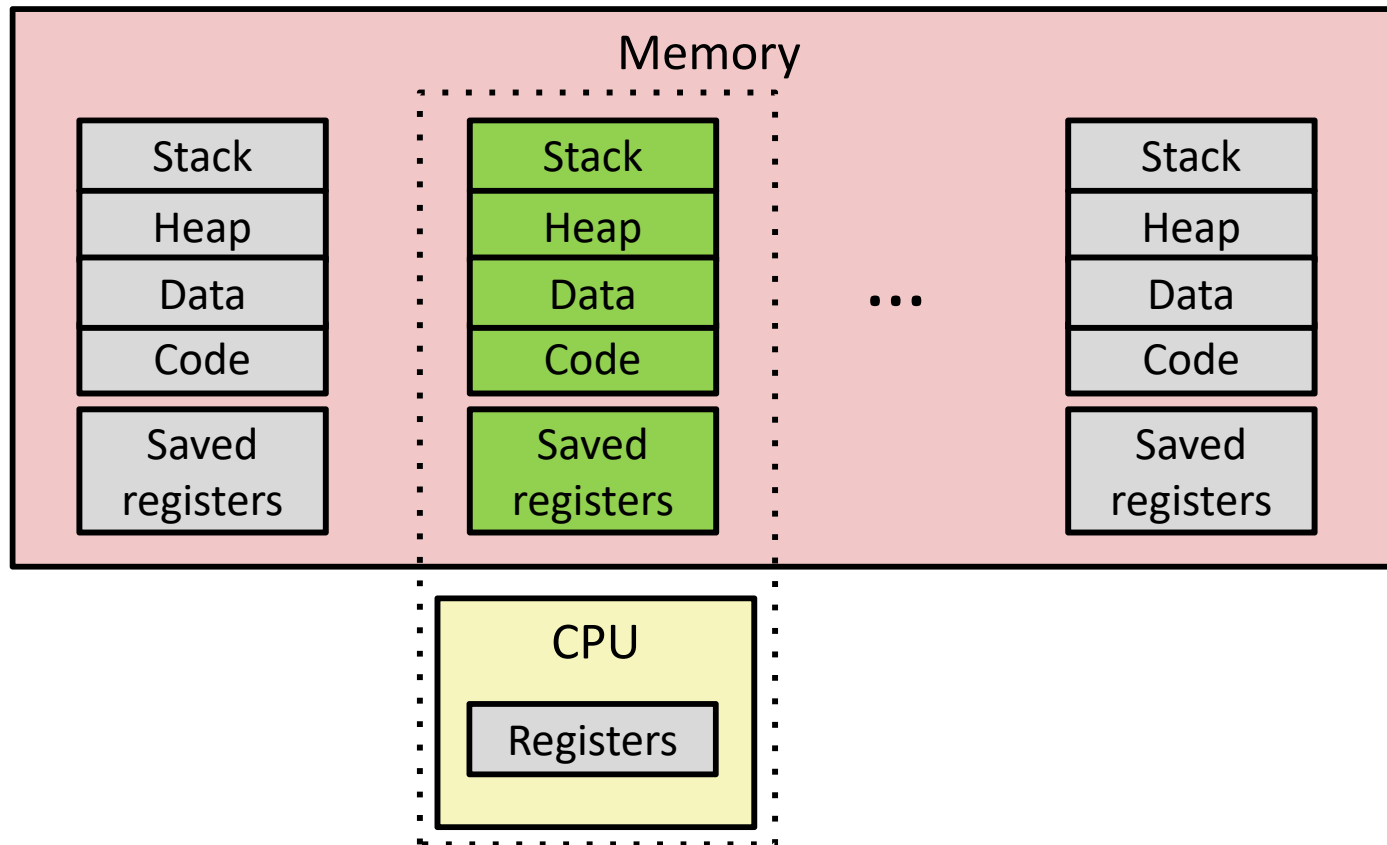
# Multiprocessing



## ❖ Context switch

- 1) Save current registers in memory

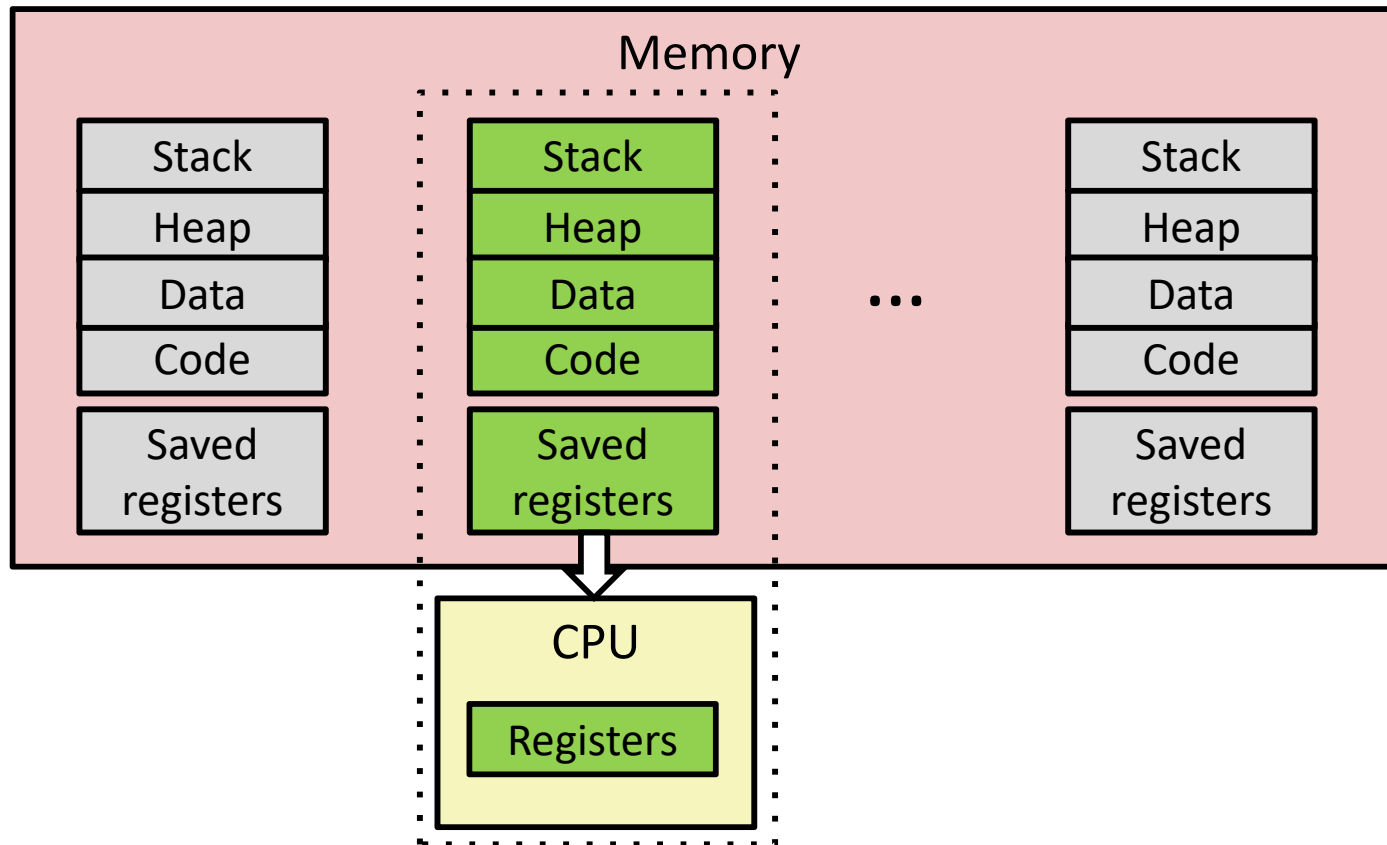
# Multiprocessing



## ❖ Context switch

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

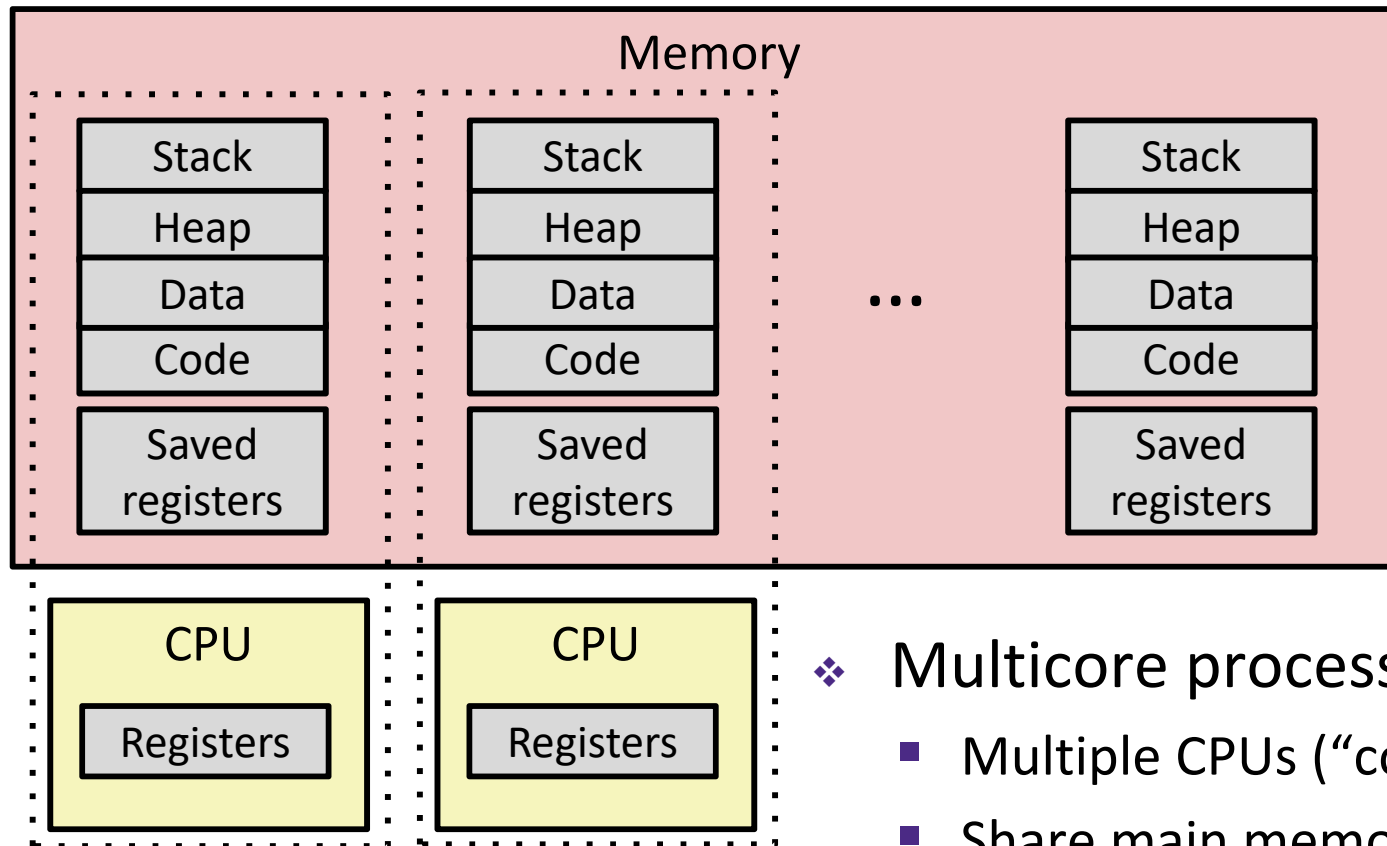
# Multiprocessing



## ❖ 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

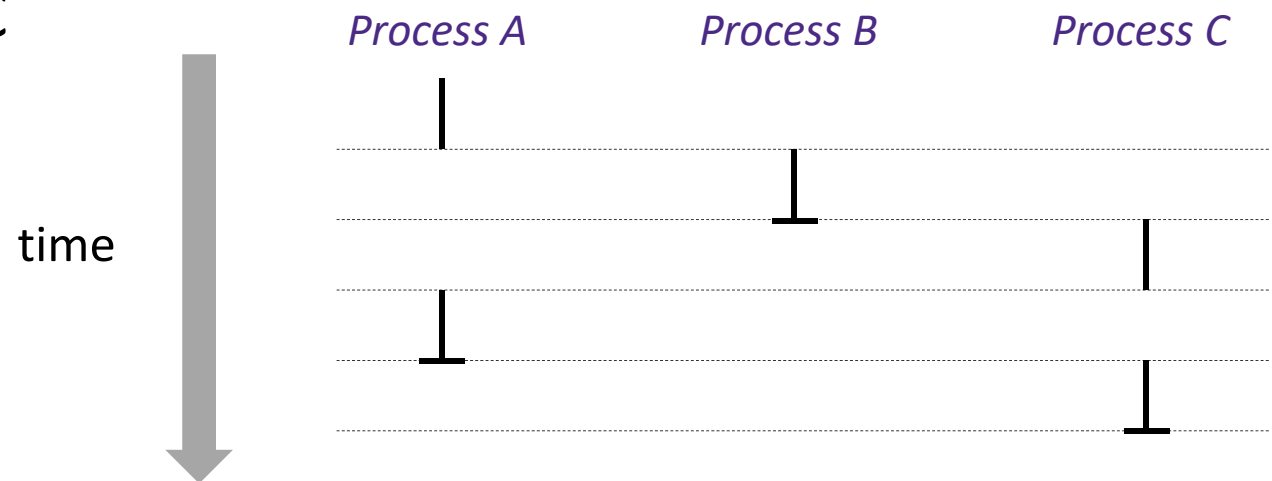
- Multiple CPUs (“cores”) on single chip
- Share main memory (and some of the caches)
- Each can execute a separate process
  - Kernel schedules processes to cores
  - **Still constantly swapping processes**



Assume only one CPU

# Concurrent Processes

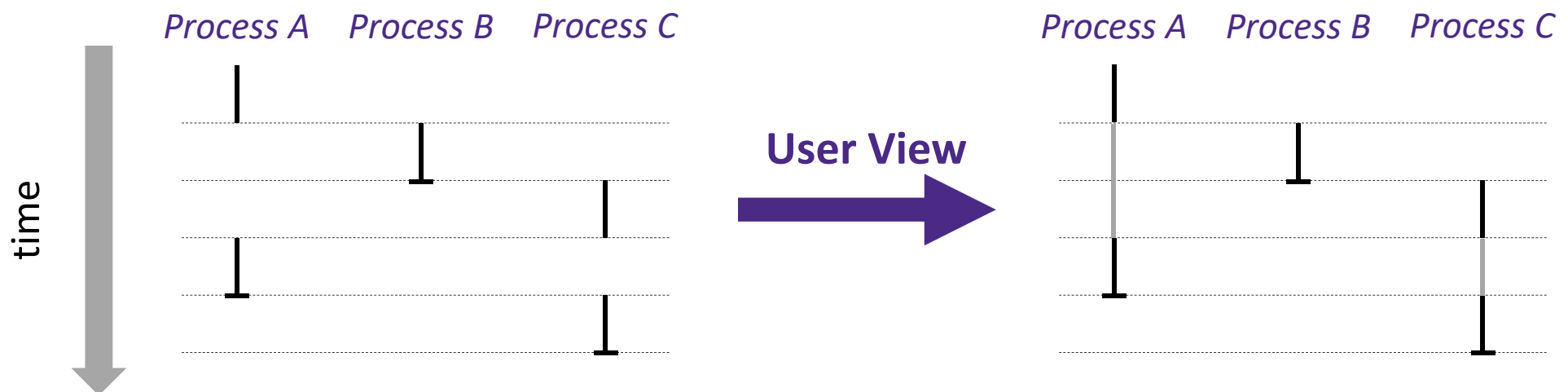
- ❖ Each process is a logical control flow
- ❖ Two processes *run concurrently* (are concurrent) if their instruction executions (flows) overlap in time
  - Otherwise, they are *sequential*
- ❖ Example: (running on single core)
  - Concurrent: A & B, A & C
  - Sequential: B & C



# User's View of Concurrency

Assume only one 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*

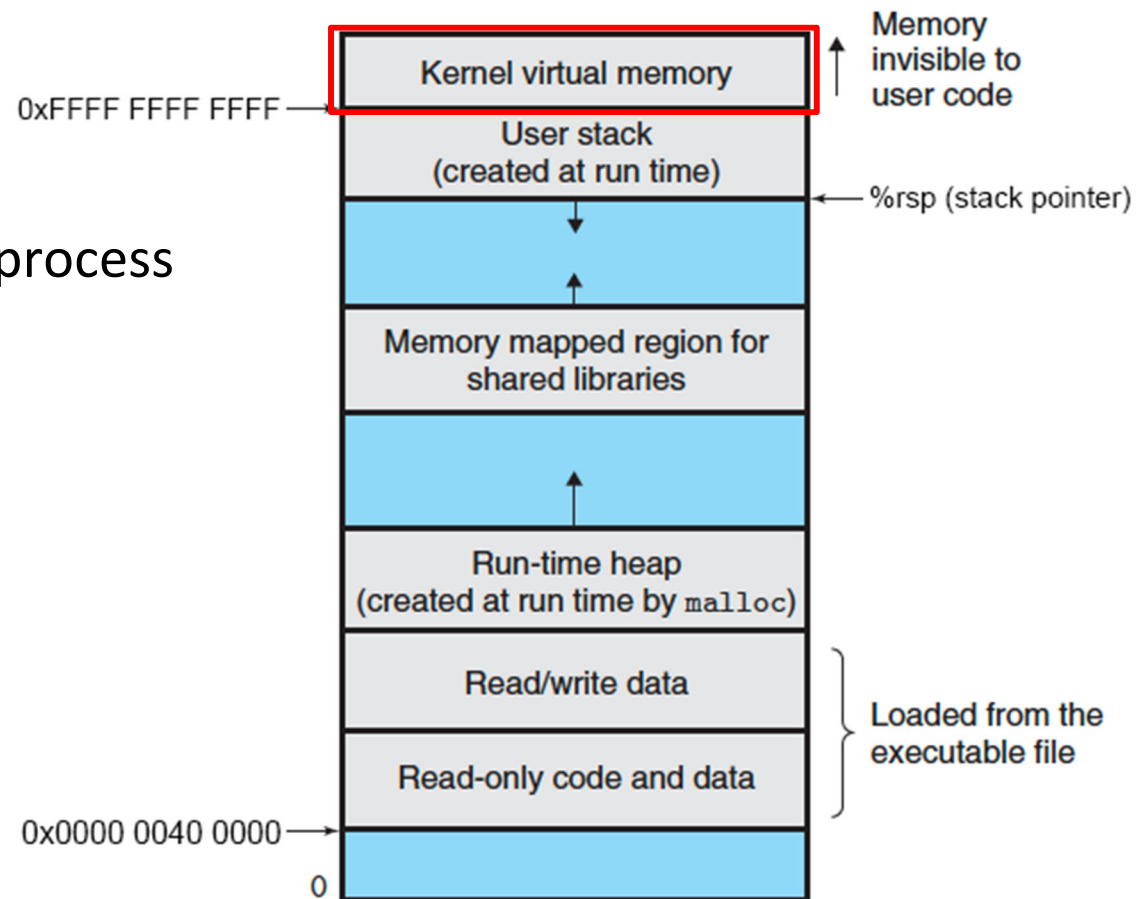


Assume only one CPU

# Context Switching

- ❖ 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

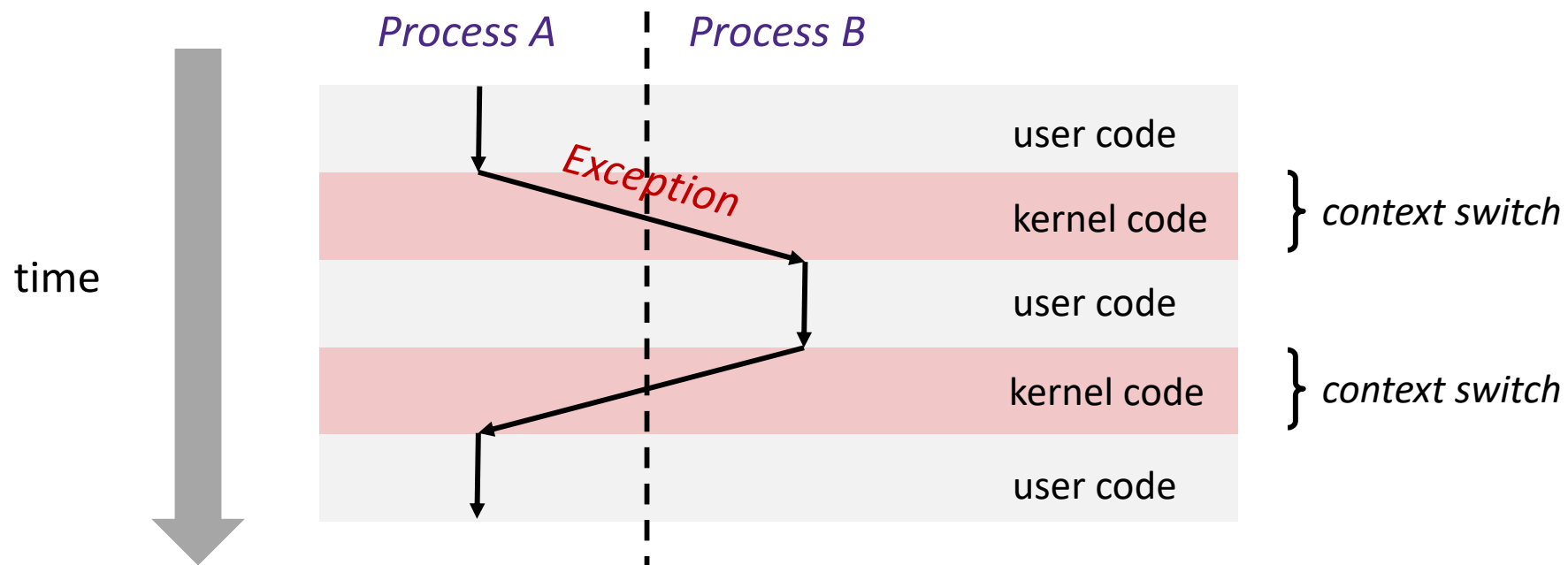
- ❖ In x86-64 Linux:
  - Same address in each process refers to same shared memory location



# Context Switching

Assume only one CPU

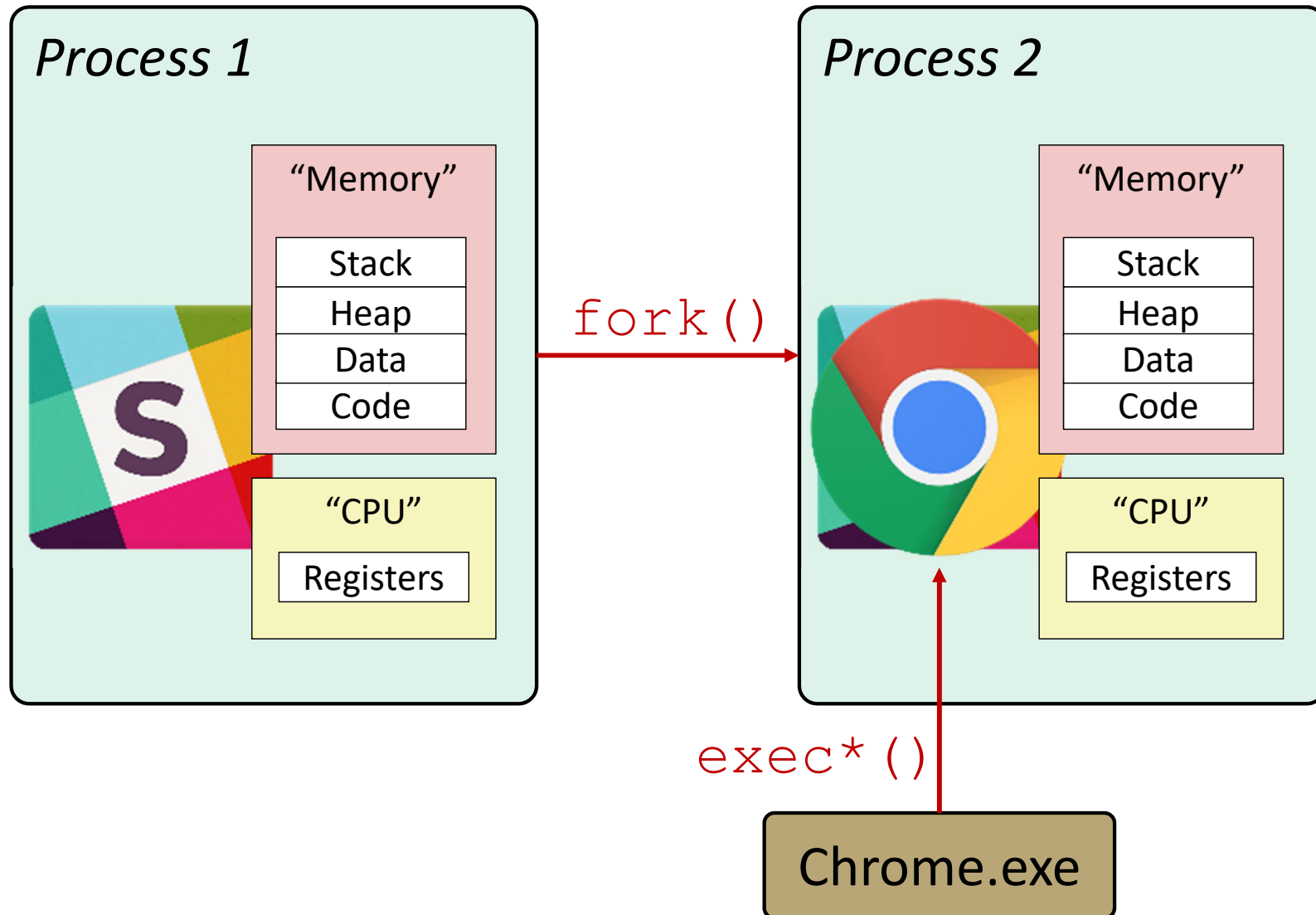
- ❖ Processes are managed by a *shared* chunk of OS code called the **kernel**
  - The kernel is not a separate process, but rather runs as part of a user process
- ❖ Context switch passes control flow from one process to another and is performed using kernel code



# Processes

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

# fork: Creating New Processes

## ❖ `pid_t fork(void)`

- Creates a new “**child**” process that is *identical* to the calling “**parent**” process, including all state (memory, registers, etc.)
- Returns 0 to the **child** process
- Returns child’s **process ID (PID)** to the **parent** process

## ❖ Child is *almost* identical to parent:

- Child gets an identical (but separate) copy of the parent’s virtual address space
- Child has a different PID than the parent

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

- ❖ `fork` is unique (and often confusing) because it is called **once** but returns “**twice**”



# Understanding `fork()`

## *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");  
}
```


# Understanding `fork()`

## 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");
}
```

# Understanding `fork()`

## 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

*Which one appears first?*

# Summary

## ❖ Processes

- At any given time, system has multiple active processes
- On a one-CPU system, only one can execute at a time, but each process appears to have total control of the processor
- OS periodically “context switches” between active processes
  - Implemented using *exceptional control flow*

## ❖ Process management

- `fork`: one call, two returns
- `execve`: one call, usually no return
- `wait` or `waitpid`: synchronization
- `exit`: one call, no return