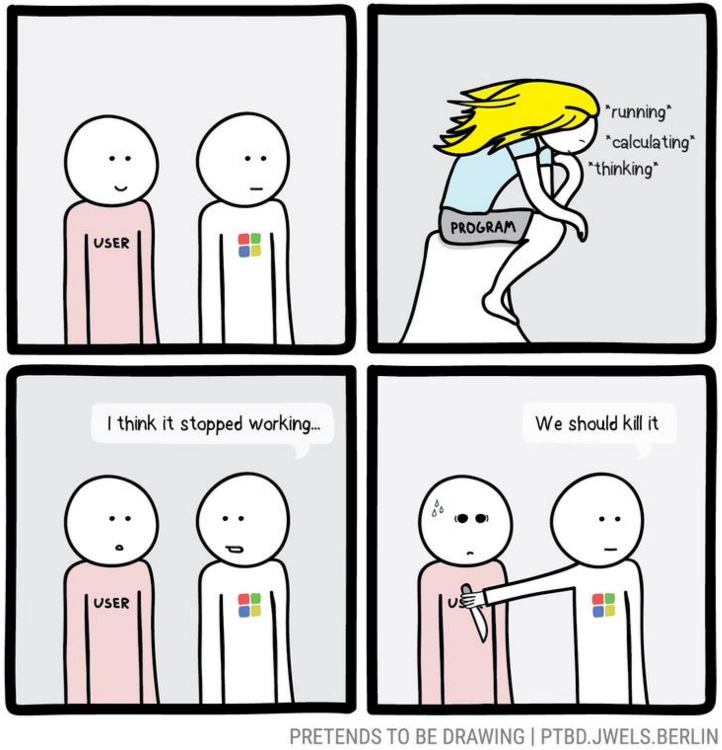
System Control Flow & Processes I

CSE 351 Summer 2024

Instructor:
Ellis Haker

Teaching Assistants:
Naama Amiel
Micah Chang
Shananda Dokka
Nikolas McNamee
Jiawei Huang

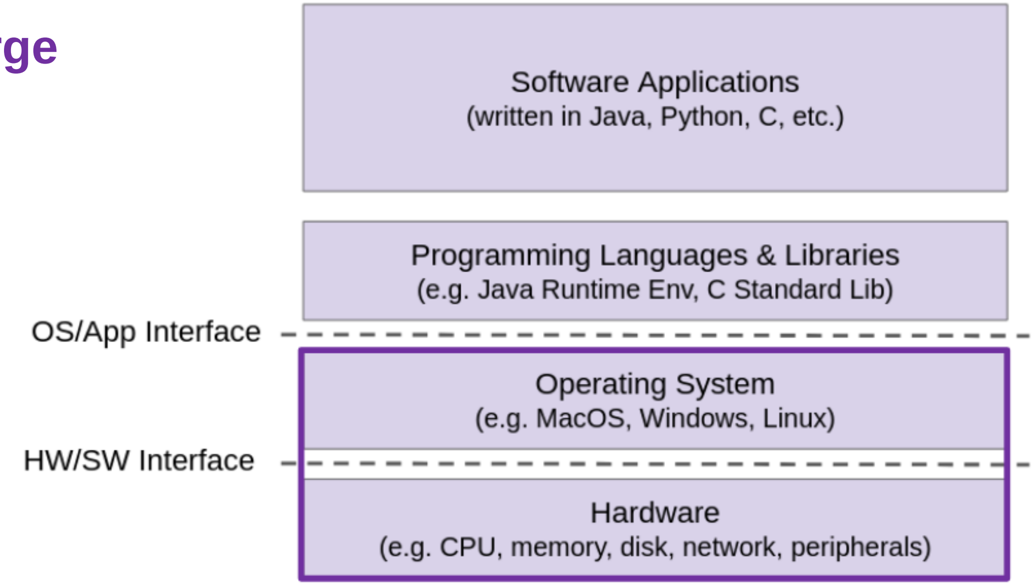


Administrivia

- Today
  - HW 19 due (11:59pm)
- Wednesday, 5/7
  - RD 22 due (1pm)
  - HW 20 due (11:59pm)
  - **Lab 4 due (11:59pm)**
- Friday, 5/9
  - RD 23 due (1pm)
  - HW 21 due (11:59pm)
- **Quiz 3 released on Monday, 5/12**
  - **Due Friday, 5/16**

Topic Group 3: Scale & Coherence

- How do we make memory accesses faster?
- **How do programs manage large amounts of memory?**
  - How can we allocate memory *dynamically* (i.e. at runtime)
- **How does your computer run multiple programs at once?**



Lecture Topics

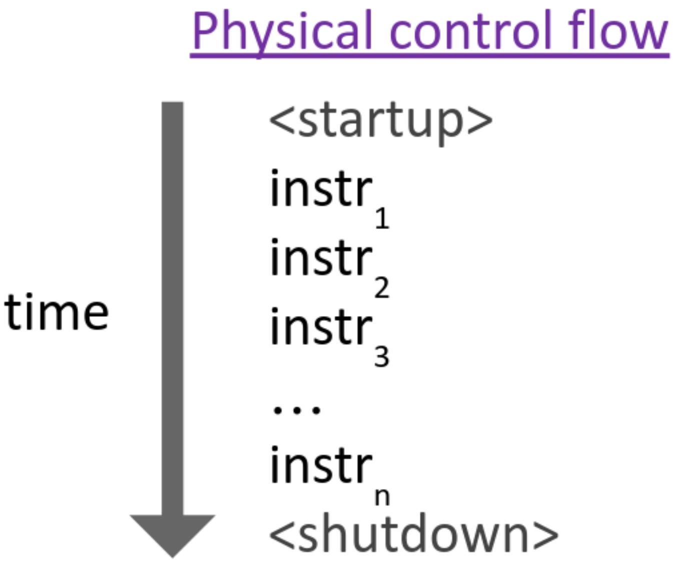
- **System Control Flow**
  - **Control flow**
  - **Exceptional control flow**
  - Asynchronous exceptions (interrupts)
  - Synchronous exceptions (traps & faults)
- OS History
- Processes and context switching
  - Creating new processes
    - fork() and exec*()
  - Ending a process
    - exit(), wait(), waitpid()
    - Zombies

Control Flow

- **So far:** we've seen how the flow of control changes as a single program executes, within that 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 (pt 2)

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



Altering Control Flow

Before, we were only thinking about what happens **within** a program. Now, we have to think about what happens **outside** a program!

- 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” on their 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!)

Exceptional Control Flow

Note: these are unrelated to Java's exceptions

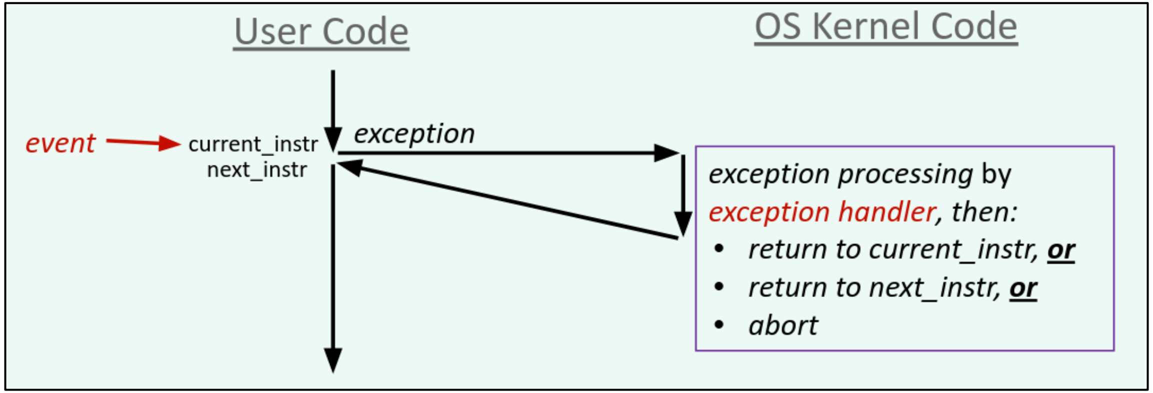
Exists at all levels of a computer system

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

Exceptions

Note: these are unrelated to Java's exceptions

- Transfer of control to the OS kernel in response to some event
  - Kernel is the operating system code that lives in memory (**very VIP!**)

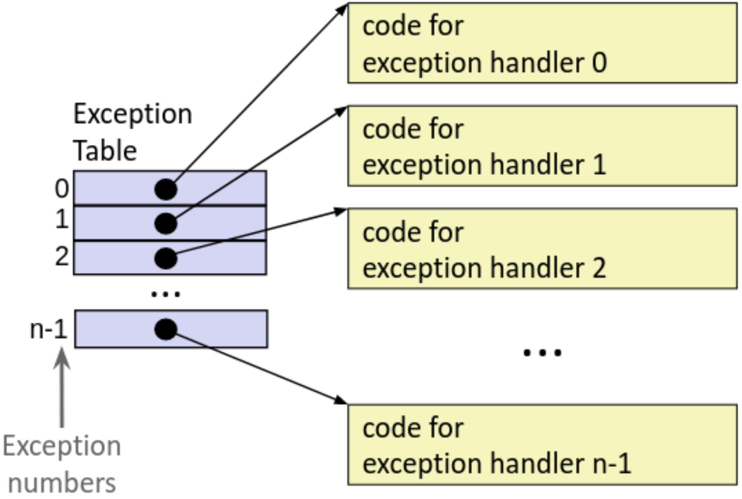


- *How does the system know where to jump to in the OS?*

Exception Table

This is extra (non-testable) material

- A jump table for exceptions (also called the **Interrupt Vector Table**)
  - Each type of event has a unique exception number *k*
  - *k* = index in the exception table, which points to the corresponding **exception handler**



Exception Table Excerpt

This is extra (non-testable) material

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

Lecture Topics

- **System Control Flow**
  - Control flow
  - Exceptional control flow
  - **Asynchronous exceptions (interrupts)**
  - **Synchronous exceptions (traps & faults)**
- OS History
- Processes and context switching
  - Creating new processes
    - fork() and exec*()
  - Ending a process
    - exit(), wait(), waitpid()
    - Zombies

Asynchronous Exceptions

- **Interrupts**: caused by events <u>external</u> 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
- <u>Examples</u>:
  - 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 <u>as a result of executing an instruction</u>:
    - **Traps:** Intentional (*why is it called this?*)
        - Transfer control to OS to perform some function
        - <u>Ex</u>: **system calls**, breakpoint traps, special instructions
        - After handler runs, returns control to “next” instruction
    - **Faults:** Unintentional, but possibly recoverable
        - <u>Ex</u>: *page fault*, segment protection faults, integer divide-by-zero exceptions
        - Either re-executes failing (“current”) instruction, or aborts
    - **Aborts:** Unintentional and unrecoverable
        - <u>Ex</u>: parity error, machine check (hardware failure detected)
        - Abort program

System Calls

- Each system call has a unique ID number
  - **Note**: this is *separate* from the exception number!
- Examples on Linux for x86-64:

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 program
60	`_exit`	Terminate process
62	`kill`	Send signal to process

files

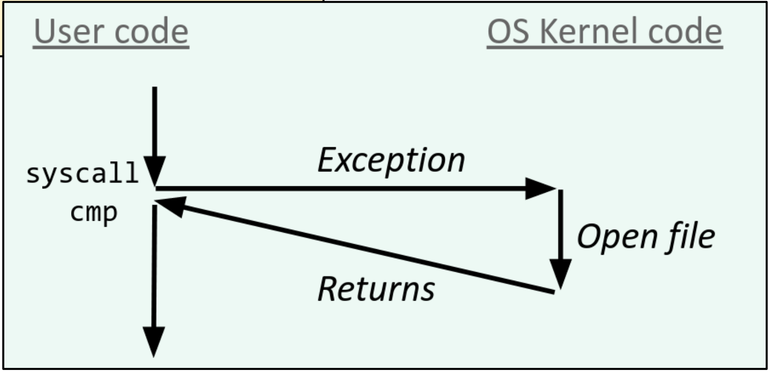
processes

Trap Example: Opening File

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

```
__open:
    ...
    mov $0x2,%eax                 # open is syscall 2
    syscall                       # return value in %rax
    cmp $0xfffffffffffff001,%rax  # check for error
    ...
    retq
```

see prev slide

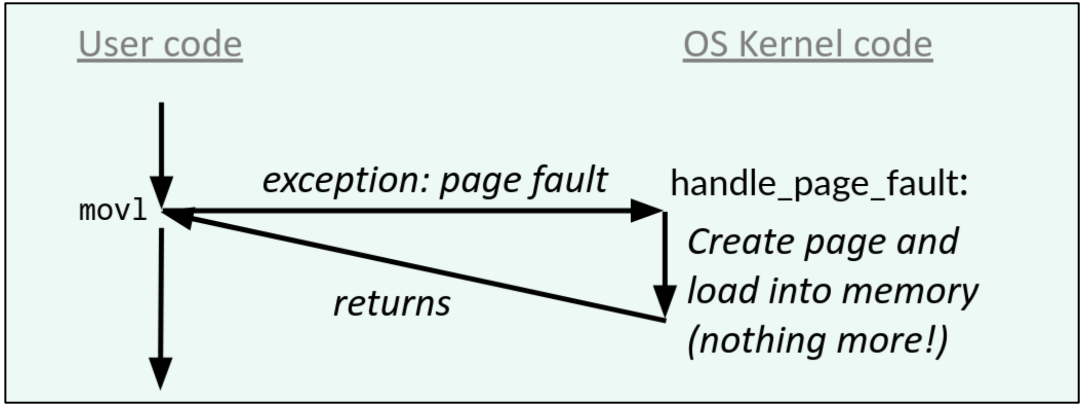


- Syscall number stored in `%rax` (weird)
  - Other arguments in regular arg registers
- Check return value for negative `errno`

Fault Example: Page Fault (future lecture topic!)

- Program writes to some location
  - That portion (page) of user's memory is currently on disk and <u>not in memory</u>

```
movl   $0xd, 0x8049d10
```

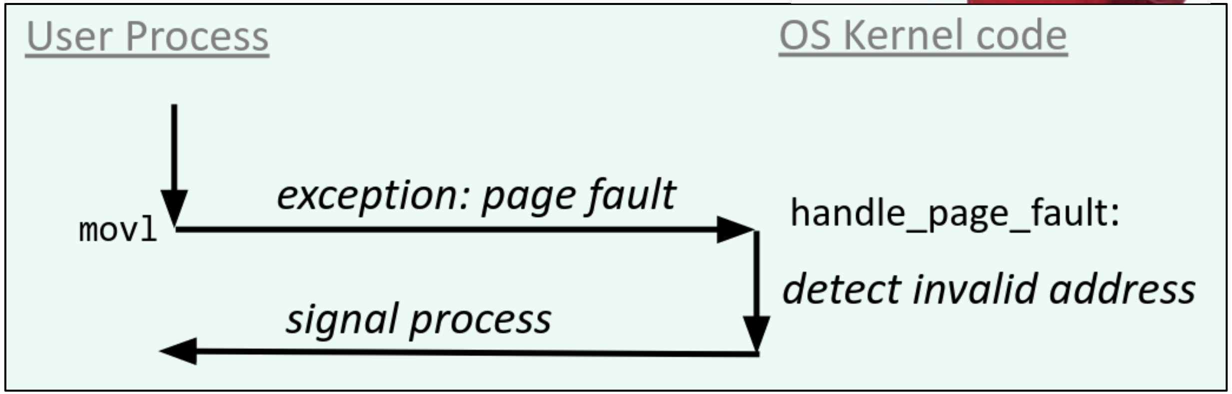


- Page fault handler loads page into memory
- Returns to faulting instruction: mov is executed again!
  - Successful on second try

Abort Example: Invalid Memory Reference

```
movl $0xd, 0x00000000
```

- Page fault handler detects invalid address
- Sends `SIGSEGV` signal to user process
- Process exits with “segmentation fault”



Lecture Topics

- System Control Flow
  - Control flow
  - Exceptional control flow
  - Asynchronous exceptions (interrupts)
  - Synchronous exceptions (traps & faults)
- **OS History**
- Processes and context switching
  - Creating new processes
    - fork() and exec*()
  - Ending a process
    - exit(), wait(), waitpid()
    - Zombies

Early Operating Systems

- Before there were operating systems, there was **computer operator**
  - Person who loaded in punch cards to begin running a program
- Operating Systems were developed to replace the human operator
- Eventually, magnetic tape replaced punch cards, allowing for **multiprocessing**
  - Programming by typing into machine

This is part of a larger trend of computers being used to automate labor.

The First Computers

- **Computer**: "a person who computes"
  - Doing calculations by hand quickly for aeronautics, warfare, science, etc.

Human Computers at NACA Credit: NASA

The women of Bletchley Park, Credit: BBC

Legacy of Computing

- Computers **augment** the abilities of humans
  - Makes the labor of boring, repetitive work more widely available
  - Highly valued, but generally exclusively available
- Computers **automate** the boring, repetitive work
  - Culturally, we are conditioned to believe that such work should be automated
  - Has consistently led to job elimination
    - e.g., ENIAC's calculation speed could displace 2,400 human computers
- Both narratives are simultaneously true, even today!
  - Underlying goal is **efficiency of labor** (usually for profit)
  - Take CSE480: Computer Ethics Seminar & CSE478: Autonomous Robotics for more

Legacy of Computing (pt 2)

- Where are we now?
  - AI being used to automate all kinds of jobs:
    - Elimination of customer service jobs, especially in developing nations.
    - Not just affecting “blue-collar” jobs, but high-paying ones too.
    - Creative jobs too? Depends on whom you ask.

other ones from class:
- Paralegals
- writers
- cashiers

Quick Discussion

- (On Ed) What jobs have you heard about that might be in imminent danger of automation? Who tends to hold these jobs?

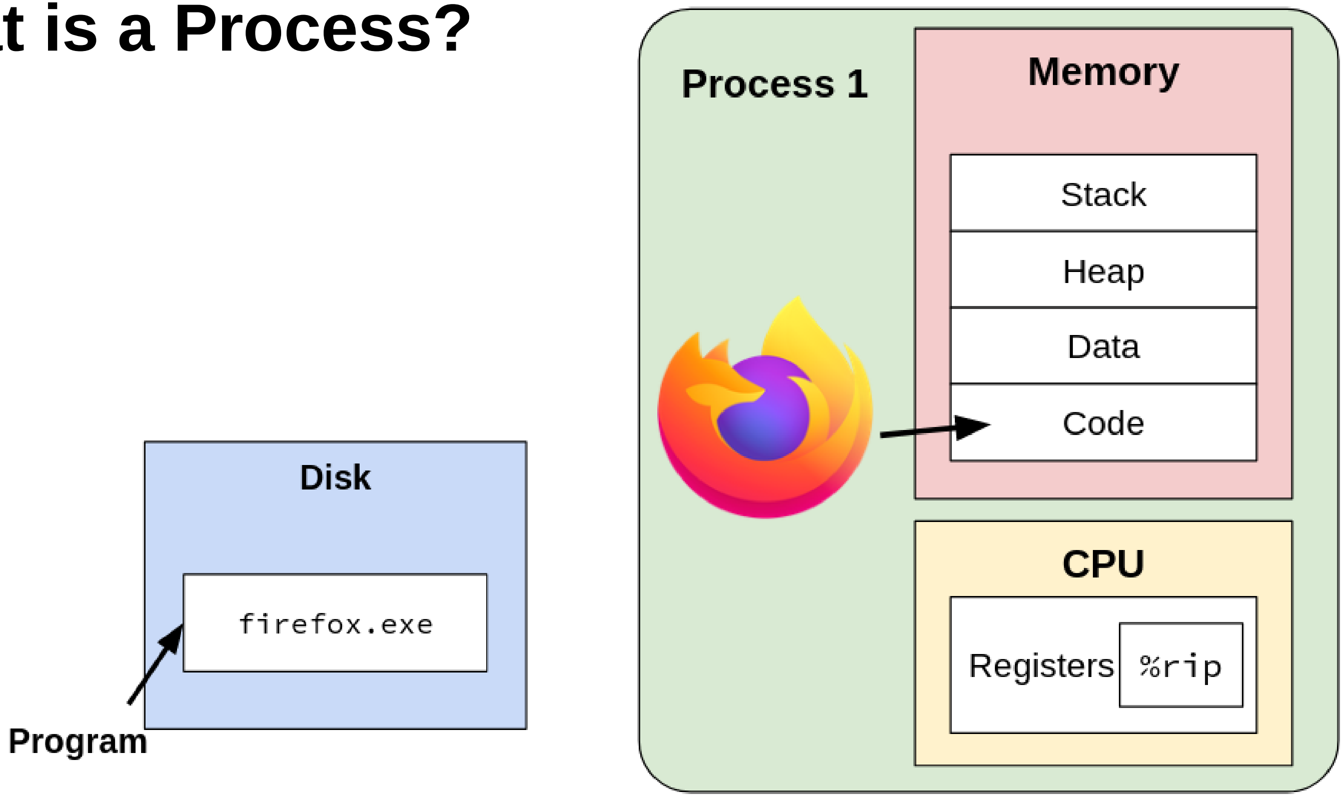
Musicians’ Strike (1940s)

Writers’ Strike (2023)

Lecture Topics

- System Control Flow
  - Control flow
  - Exceptional control flow
  - Asynchronous exceptions (interrupts)
  - Synchronous exceptions (traps & faults)
- OS History
- **Processes and context switching**
  - Creating new processes
    - fork() and exec*()
  - Ending a process
    - exit(), wait(), waitpid()
    - Zombies

What is a Process?



What is a Process? (pt 2)

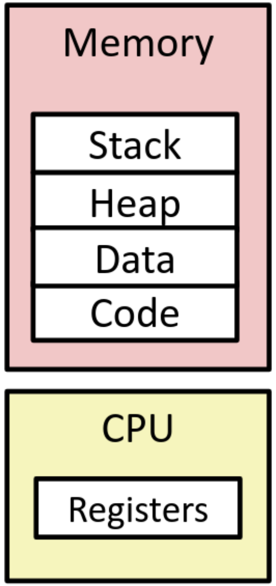
- A **process** is an **instance of a running program**
  - One of the most profound ideas in computer science!
- Another *abstraction* in our computer
  - Provided by the OS
    - Uses a data structure to keep track of each process (ID, open files, etc.)
  - Maintains the **interface** between the program and the underlying hardware

- What is the difference between:
  - A processor? A program? A process?

CPU: hardware .exe: code context code runs in (memory, register values, etc.)

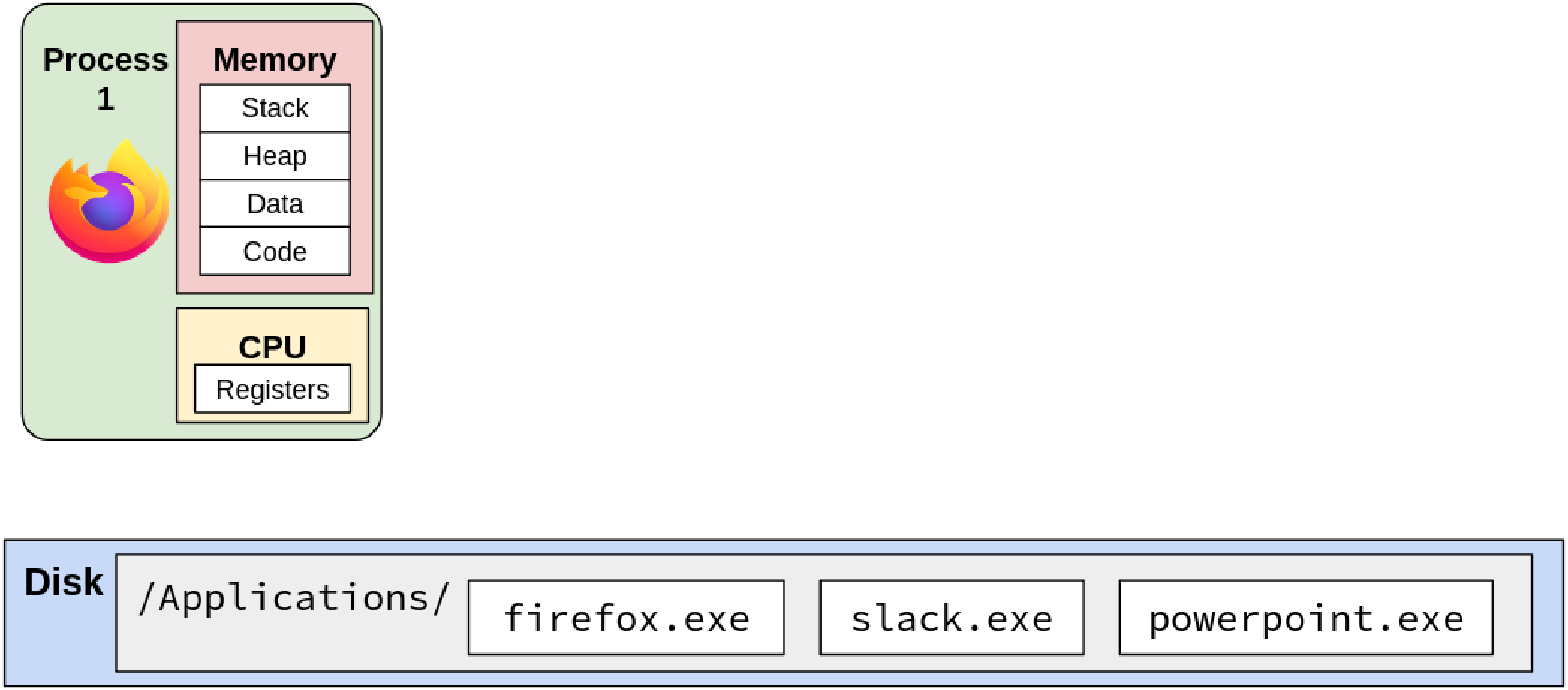
Processes

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

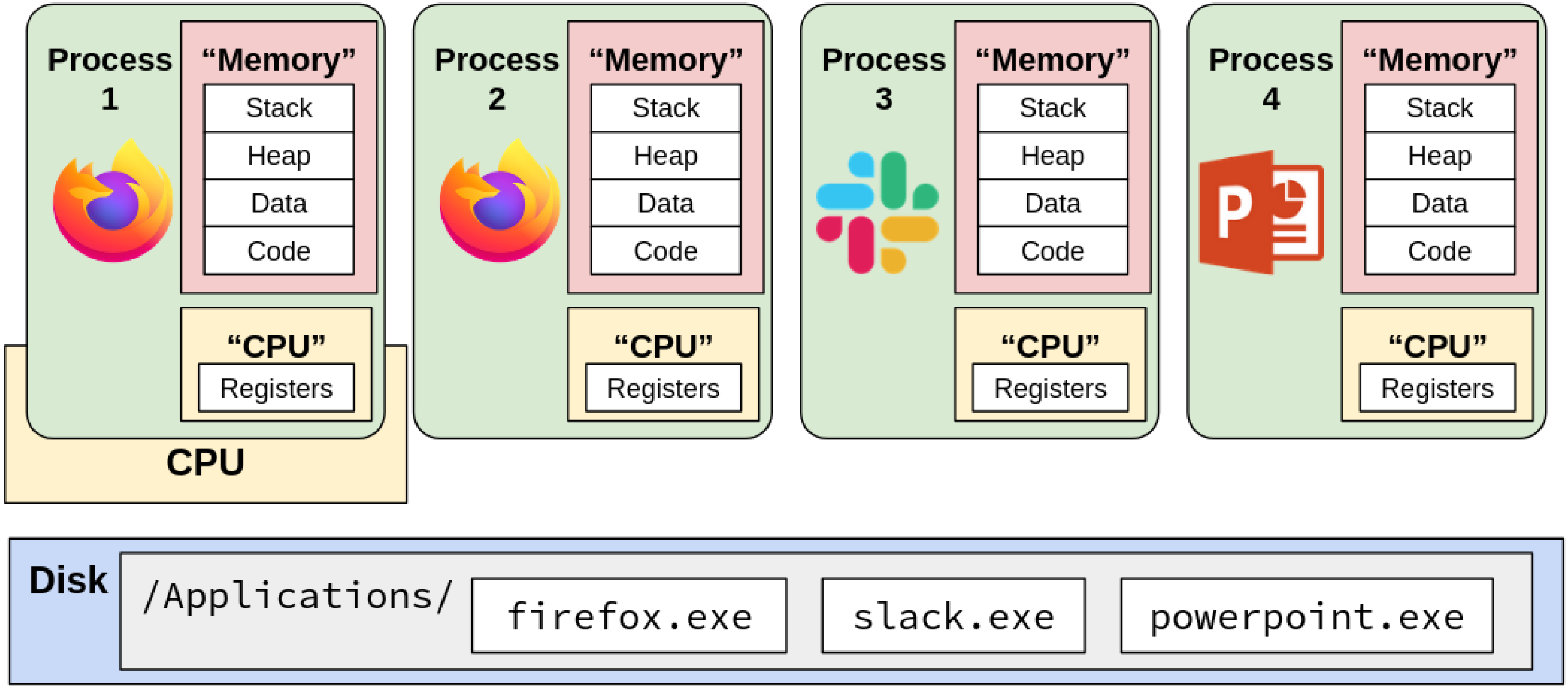


- What do processes have to do with exceptional control flow?
  - Exceptional control flow is the mechanism the OS uses to enable multiple processes to run on the same system

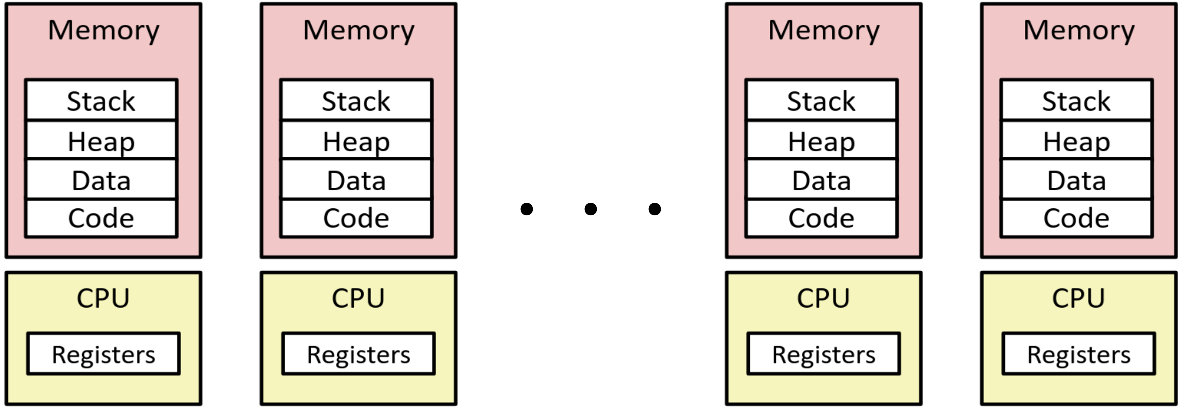
Processes (pt 2)



Processes (pt 3)

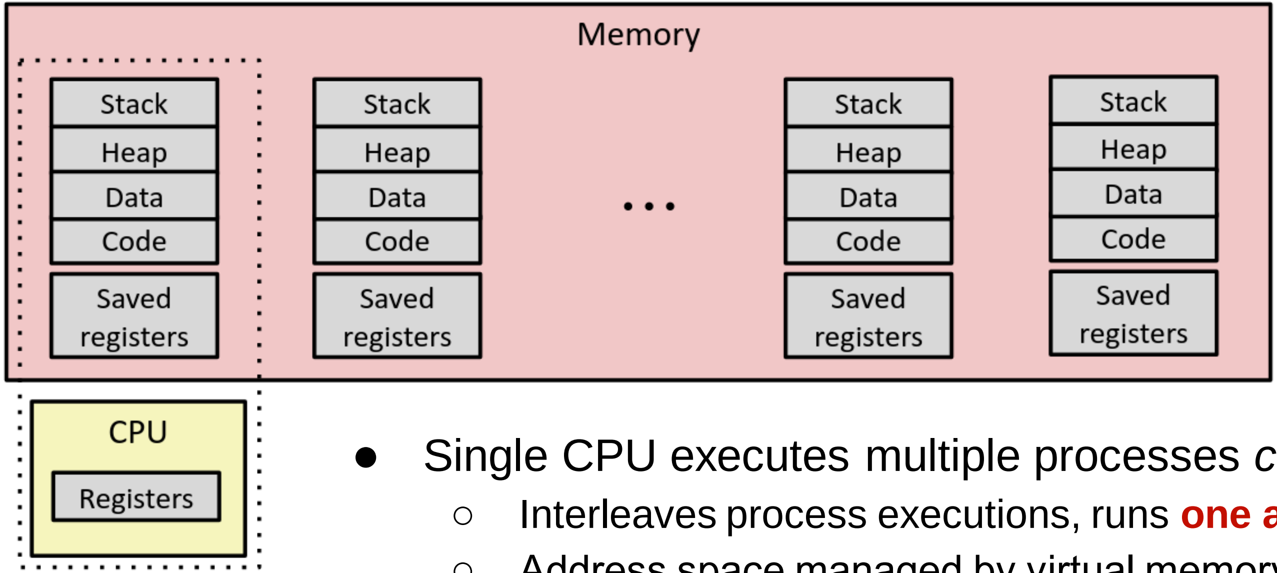


Multiprocessing: the Illusion 🧙



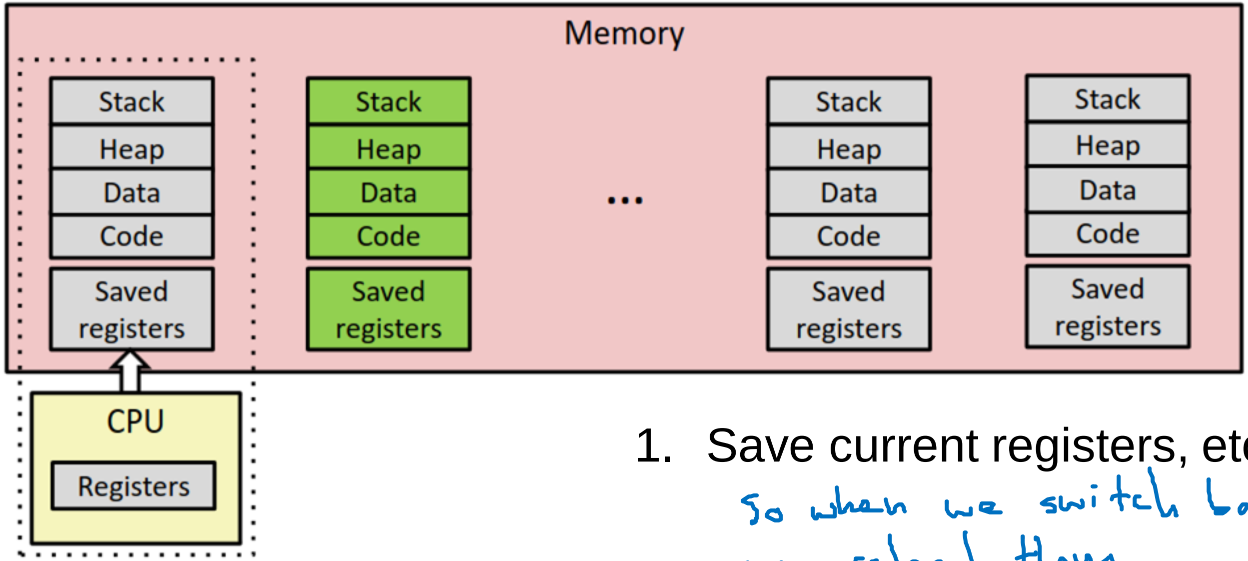
- Computer runs many processes simultaneously
  - Applications for one or more users
    - Web browsers, email clients, editors, …
- Background tasks
  - Monitoring network & I/O devices

Multiprocessing: the Reality



- Single CPU executes multiple processes *concurrently*
  - Interleaves process executions, runs **one at a time**
  - Address space managed by virtual memory system (we'll get to it!)
  - **Execution context** (register values, stack, etc.) saved in memory when process isn't running

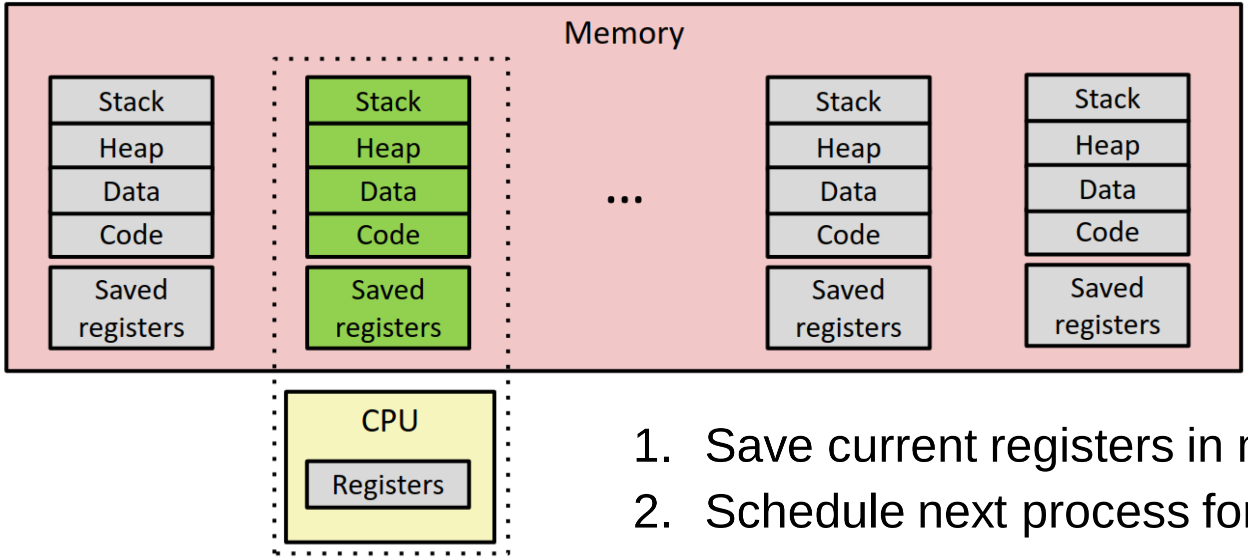
Context Switch steps



1. Save current registers, etc. in memory

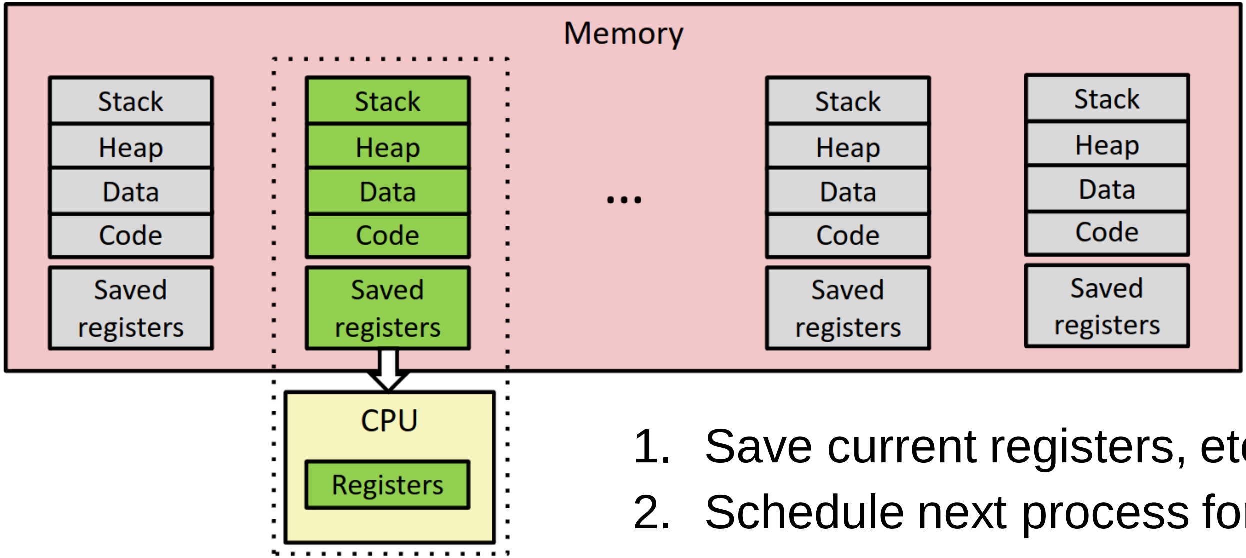
So when we switch back, we can reload them

Context Switch steps (pt 2)



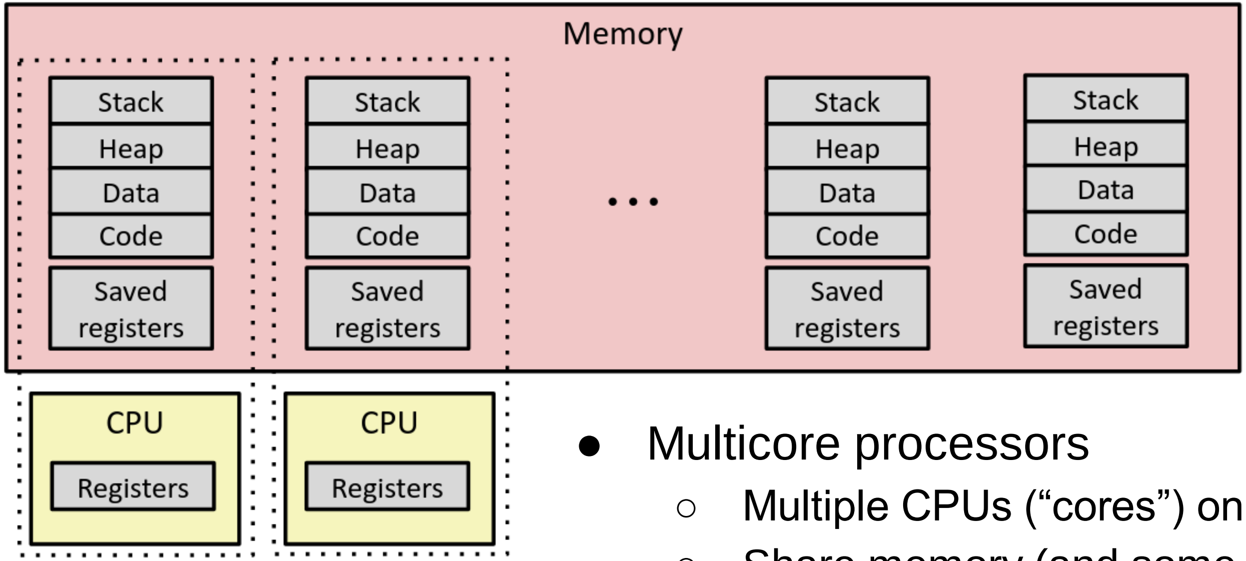
1. Save current registers in memory
2. Schedule next process for execution

Context Switch steps (pt 3)



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

Multiprocessing: the (Modern) Reality



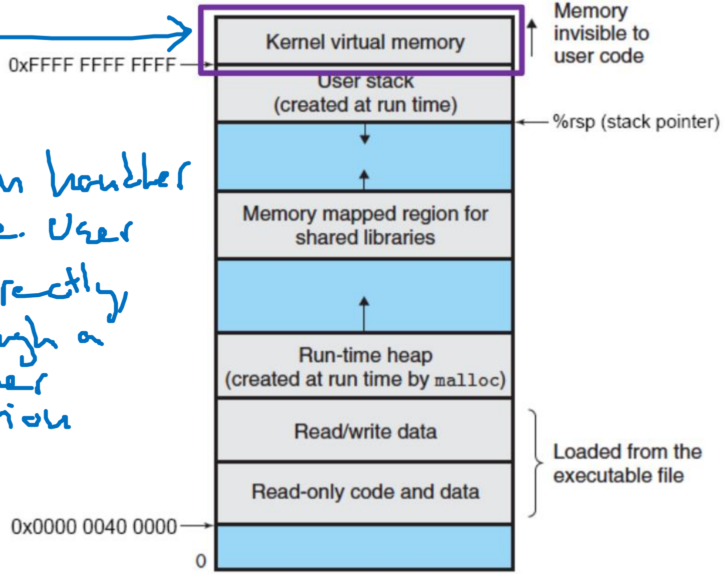
- Multicore processors
  - Multiple CPUs (“cores”) on one chip
  - Share memory (and some caches)
  - Each can execute a separate process
    - **Still constantly switching**

Context Switching

Assume only <u>one</u> 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 each user process
- In x86-64 Linux:
  - Same address in each process' kernel memory refers to the same shared memory location*

exception handler code goes here. User can't run it directly, has to go through a syscall or other exception

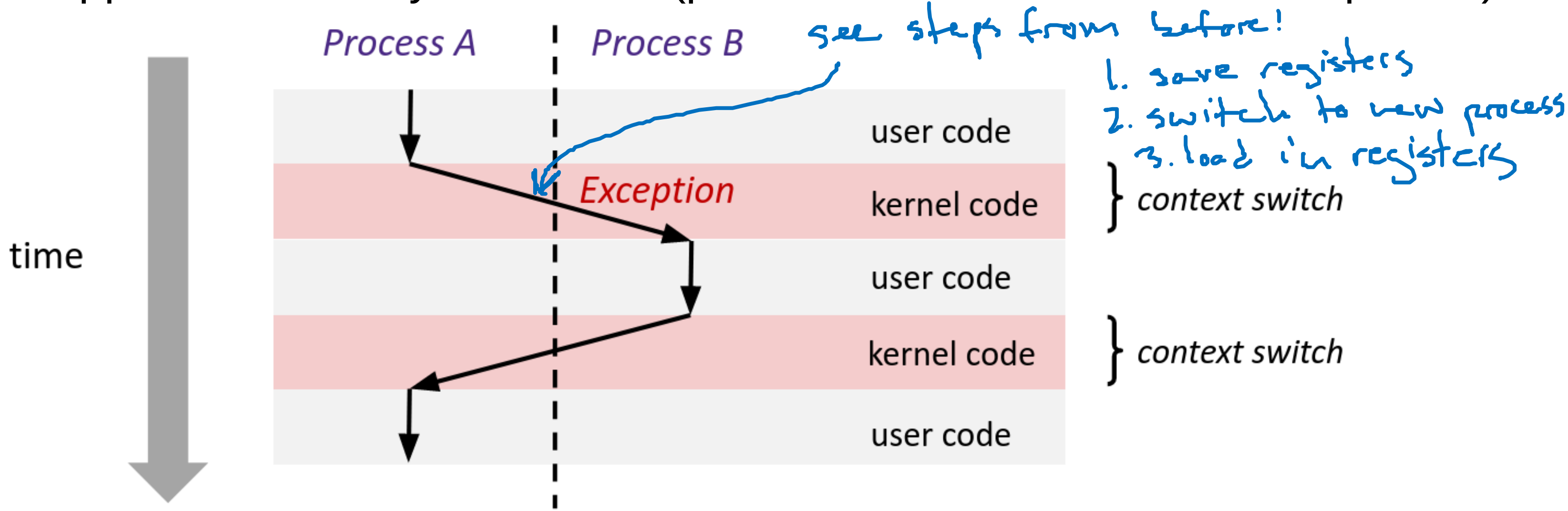


*sort of, the story here became more complicated after Meltdown and Spectre…

Context Switching (pt 2)

Assume only <u>one</u> CPU core

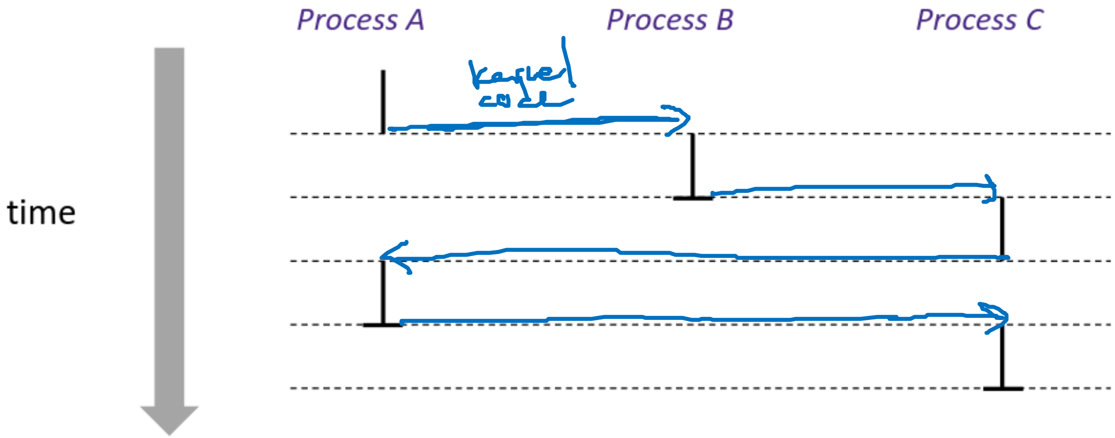
- Context switch **passes control flow** from one process to another and is performed using kernel code
- Can happen for a variety of reasons (process terminated, timer interrupt, etc.)



Concurrency

Assume only <u>one</u> CPU core

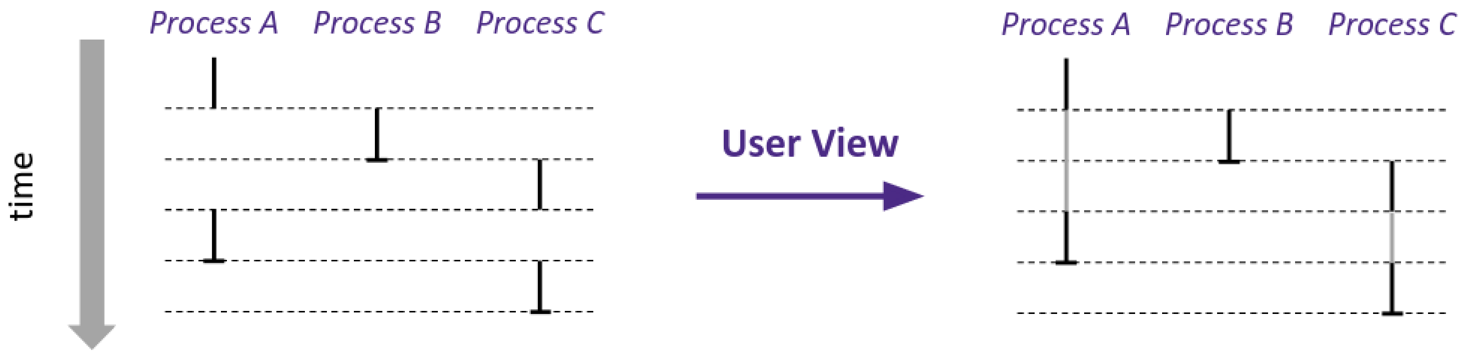
- Two processes are **concurrent** if their instruction executions/flows overlap in time
  - i.e. one starts before the other has *completely* finished executing
  - Otherwise, they are **sequential**
- <u>Example:</u>
  - Concurrent: A&B, A&C
  - Sequential: B&C



User's View of Concurrency

Assume only <u>one</u> CPU core

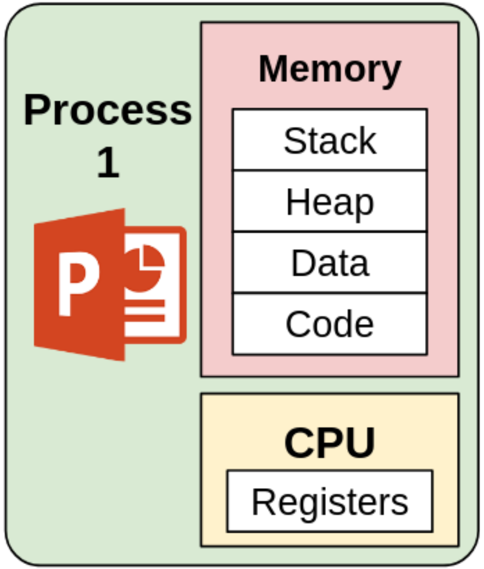
- Control flows for concurrent processes are physically disjoint in time
    - CPU executes instructions for one process at a time
- However, we can ***think*** of them as if they're running at the same time, in <u>parallel</u>

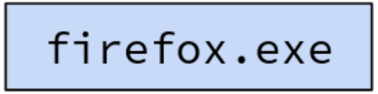


Lecture Topics

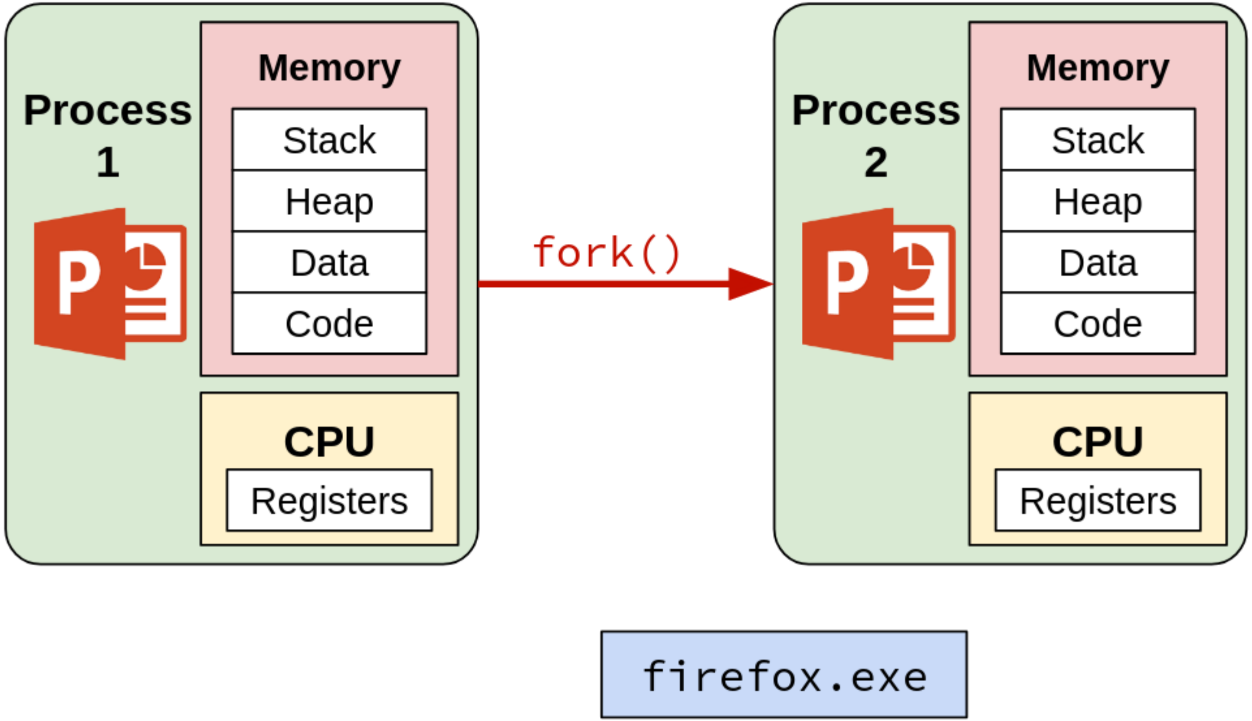
- System Control Flow
  - Control flow
  - Exceptional control flow
  - Asynchronous exceptions (interrupts)
  - Synchronous exceptions (traps & faults)
- OS History
- **Processes and context switching**
  - **Creating new processes**
    - **fork() and exec*()**
  - Ending a process
    - exit(), wait(), waitpid()
    - Zombies

Creating New Processes and Programs

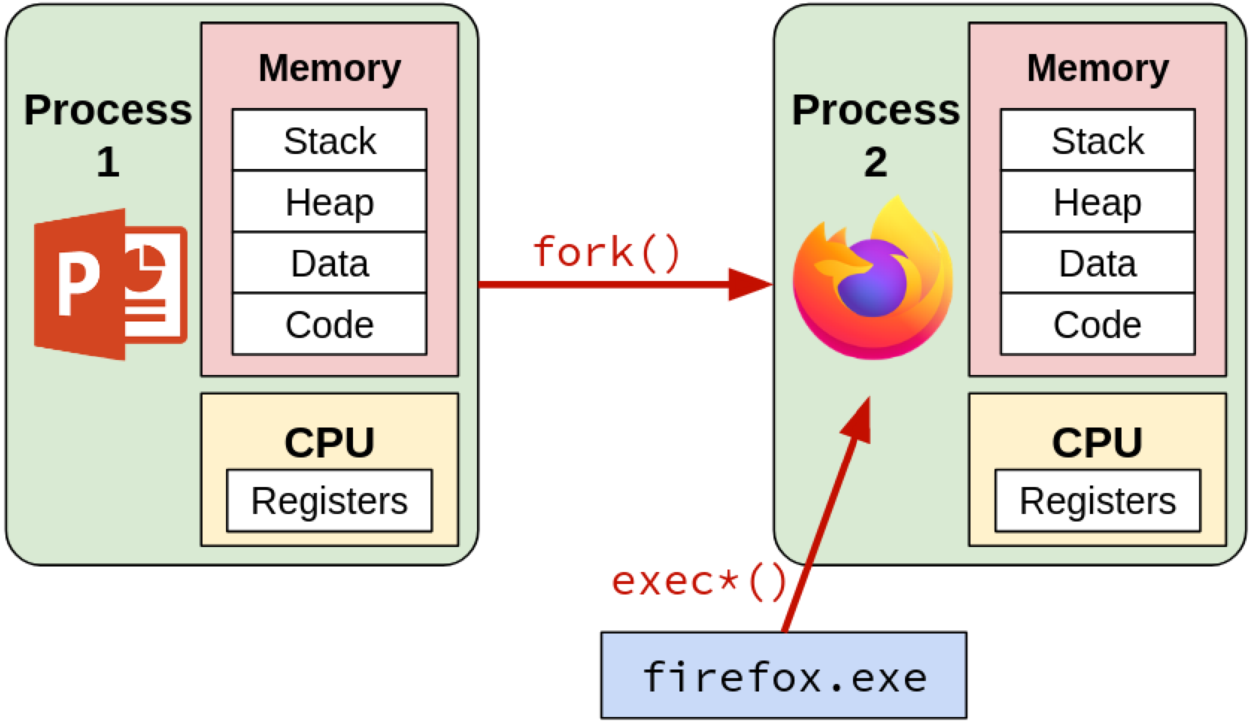




Creating New Processes and Programs (pt 2)



Creating New Processes and Programs (pt 3)

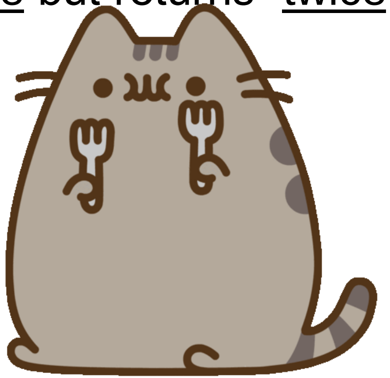


Creating New Processes and Programs

- **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`
  - Both `fork()` and `exec*()` are **system calls**
- Other system calls for process management:
  - `getpid()`
  - `exit()`
  - `wait()`, `waitpid()`

`fork`: Creating New Processes

- **`pid_t`** `fork()`
    - Creates a new "child" process that is *a copy* of 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
- `fork` is unique (and often confusing) because it is called once but returns "twice"



- Child is *almost identical* to the parent
    - Gets an identical (but separate) copy of parent's address space
    - Register `%rax` is 0 on return
    - Has a different PID than the parent

fork Example

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

- Child has the same memory and register values as the parent (except `%rax`)
    - This includes the code in memory and `%rip`, so child will start executing the same code as parent, right after `fork()` returns
- Can use the return values from `fork()` to distinguish between parent and child

Why have `fork` and `exec`?

- Why make a copy of the parent's data if we're just going to throw it away in `exec()`?
  - Easier to implement
  - Useful if you want the new process to execute the same code
- Not all systems do it this way!
  - Windows uses **`spawn`**
  - Optional reading: "A Fork in the Road"

Summary: Exceptional Control Flow

- **Exceptional control flow** allows the OS to interrupt a currently running program
  - **Interrupts** are **asynchronous** (i.e. they come from outside the program)
  - **Traps** are **synchronous**, purposefully invoked by the user application
    - Includes **system calls**: OS services that user programs can invoke
  - **Faults** are **synchronous**, unintentional, but possibly recoverable
  - **Aborts** are **synchronous** and occur in response to unrecoverable errors

Summary: Processes

- A **process** is a single instance of a running program
  - Keeps track of the **context** the program is being run in (register values, memory state, etc.)
- A computer can have multiple **concurrent** processes, but can only **execute one at a time**
  - Performs a **context switch** to move between processes
- Processes are created using the **`fork`** system call
  - Creates a ***copy*** of the parent process
  - The **`exec`** system call throws out the old context and starts running a new program