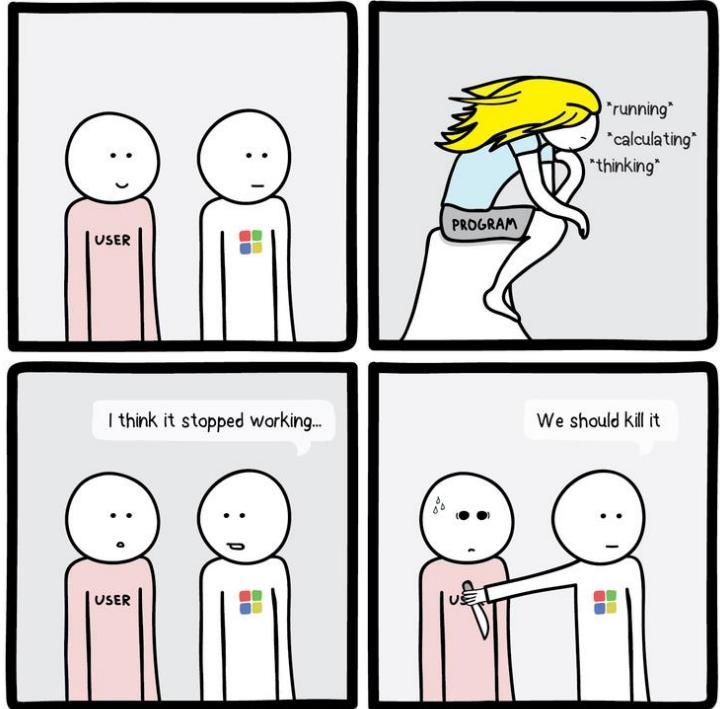


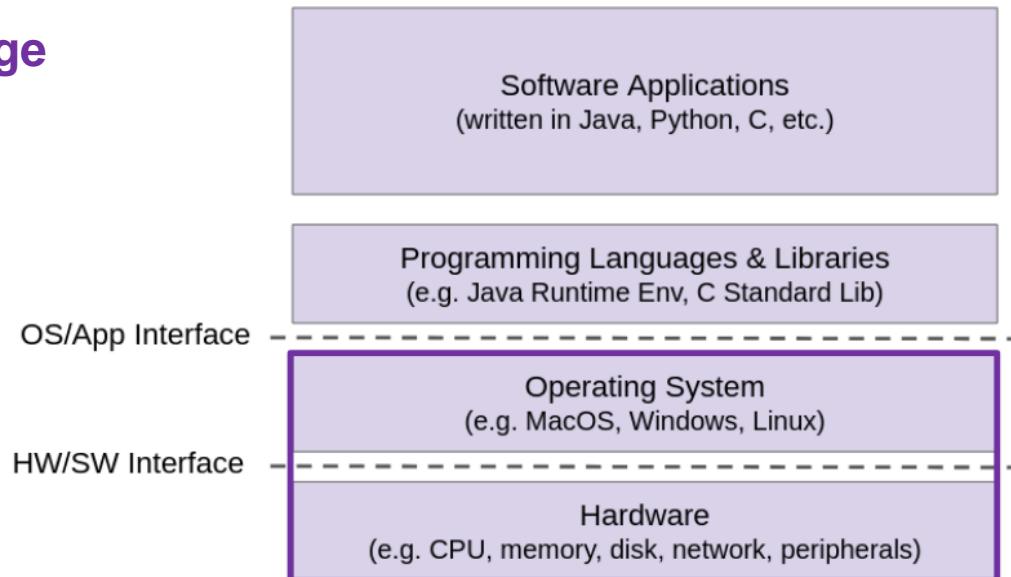
System Control Flow & Processes I



PRETENDS TO BE DRAWING | PTBD.JWELS.BERLIN

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

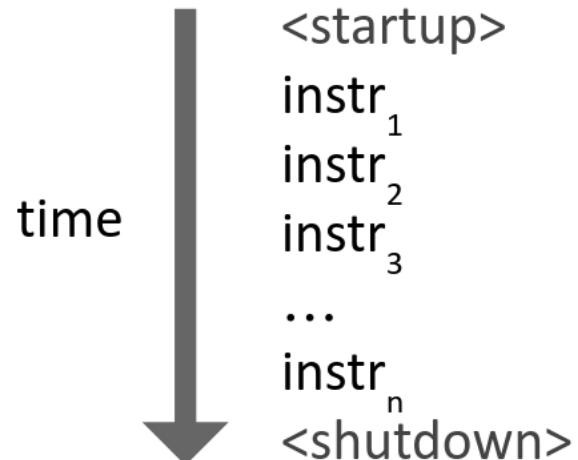
Control Flow

- **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**

Physical control flow



Altering Control Flow

- Up to now, two ways to change control flow:
 - Jumps (conditional (if/else) and unconditional (infinite loops))
 - 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!)

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

Exceptional Control Flow

Note: these are unrelated to Java's exceptions

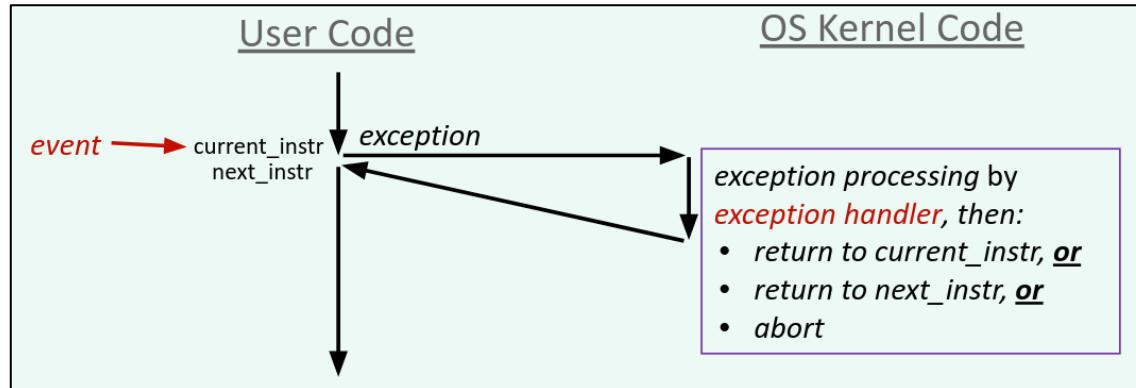
Exists at all levels of a computer system

- 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!**)



Lecture Topics

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

Asynchronous Exceptions

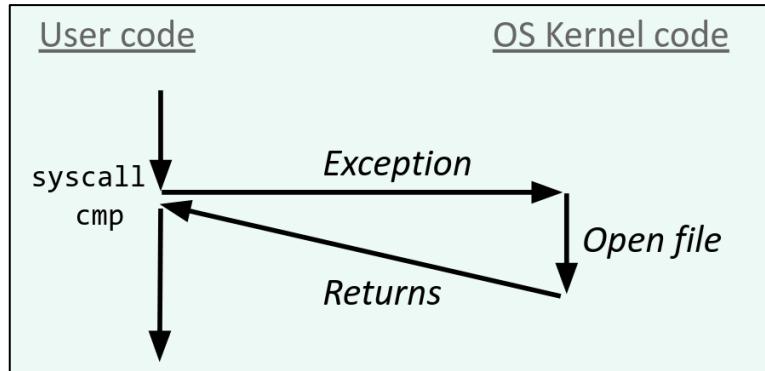
- **Interrupts:** caused by events external to the processor
 - After interrupt handler runs, the handler returns to “next” instruction
- Examples:
 - I/O interrupts
 - Hitting Ctrl-C on the keyboard
 - Clicking a mouse button or tapping a touchscreen
 - Arrival of a packet from a network
 - Arrival 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
 - Ex: system calls, breakpoint traps, special instructions
 - After handler runs, returns control to “next” instruction
 - **Faults:** Unintentional, but possibly recoverable
 - Ex: page fault, segment protection faults, integer divide-by-zero exceptions
 - Either re-executes failing (“current”) instruction, or aborts
 - **Aborts:** Unintentional and unrecoverable
 - Ex: parity error, machine check (hardware failure detected)
 - Abort program

Trap Example: Opening File

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

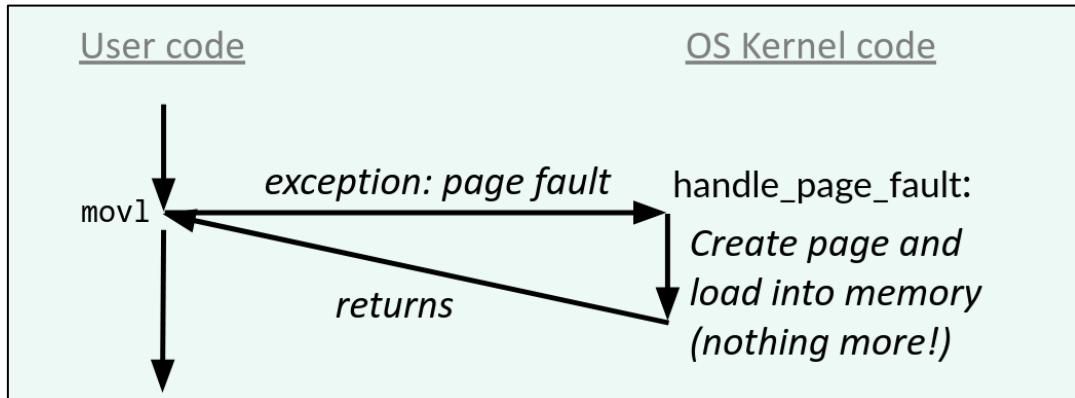


Fault Example: Page Fault

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

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

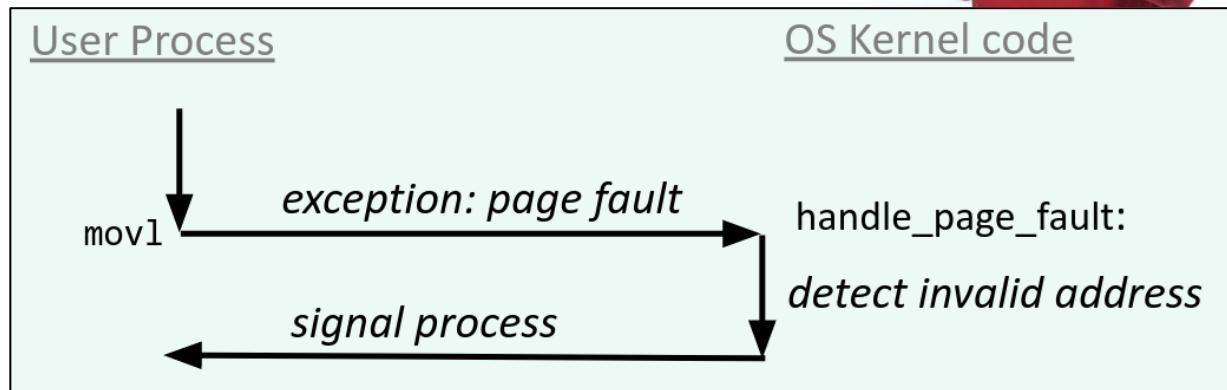
- Page fault handler loads page into memory
- Returns to faulting instruction: mov is executed again!
 - Successful on second try



Abort Example: Invalid Memory Reference

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

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



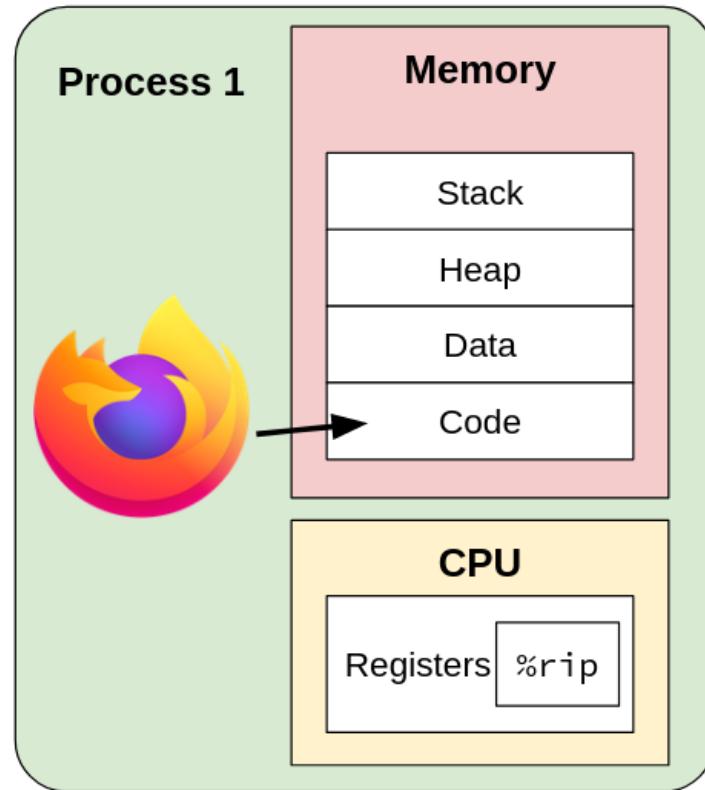
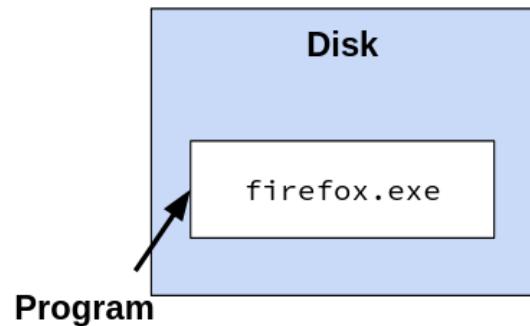
Lecture Topics

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

Lecture Topics

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

What is a Process?



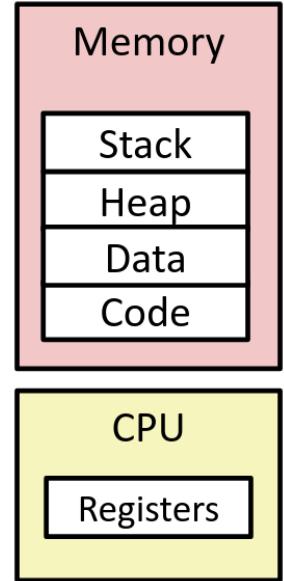
What is a Process? (pt 2)

- 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?

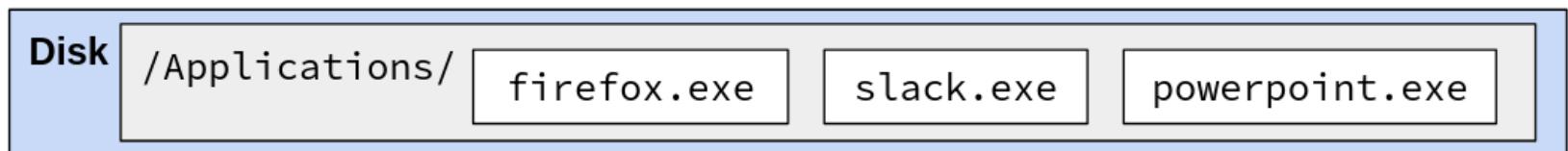
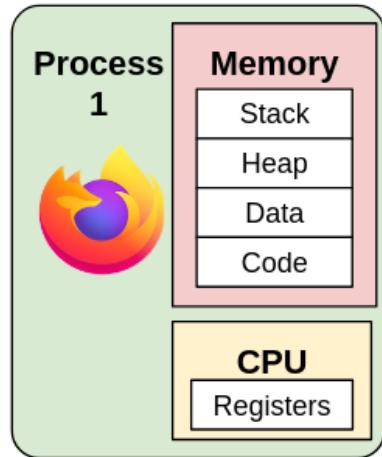


Processes

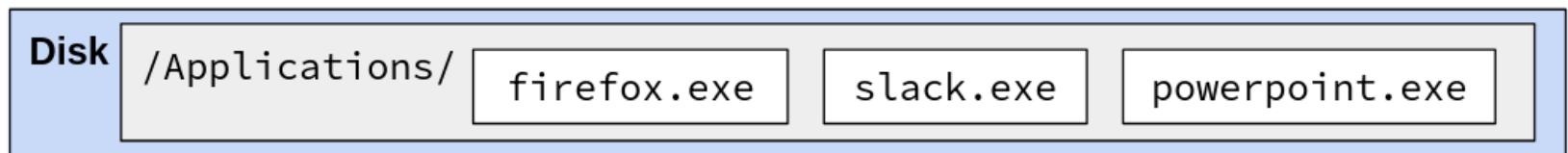
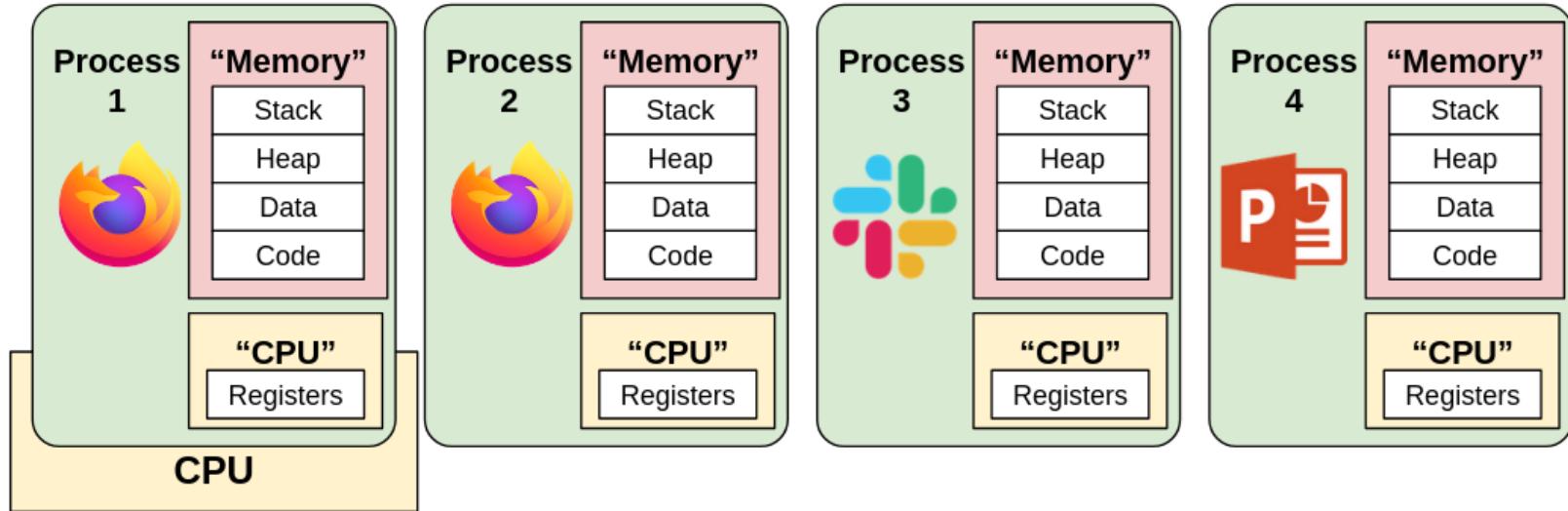
- Provides each program with two key abstractions:
 - Logical control flow
 - Each program seems to have exclusive use of the CPU
 - Provided by a kernel mechanism called **context switching**
 - Private address space
 - Each program seems to have exclusive use of memory
 - Provided by a kernel mechanism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



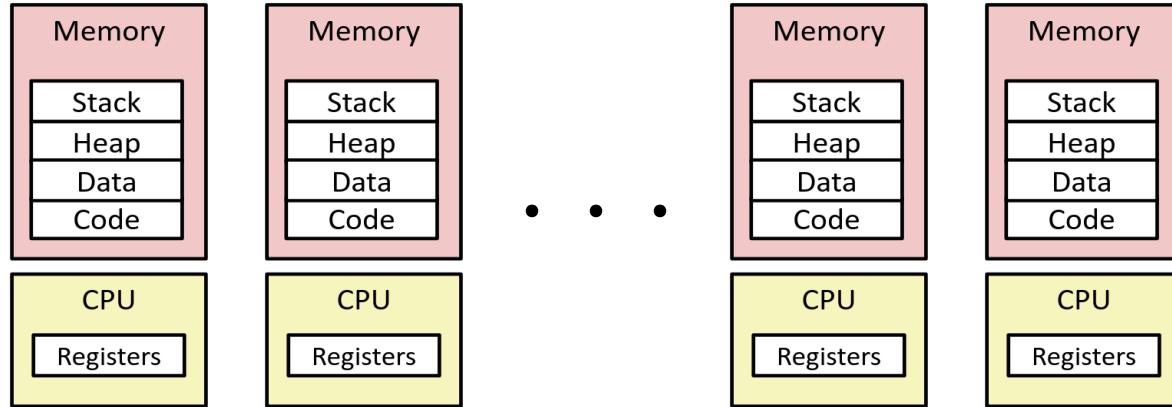
Processes (pt 2)



Processes (pt 3)

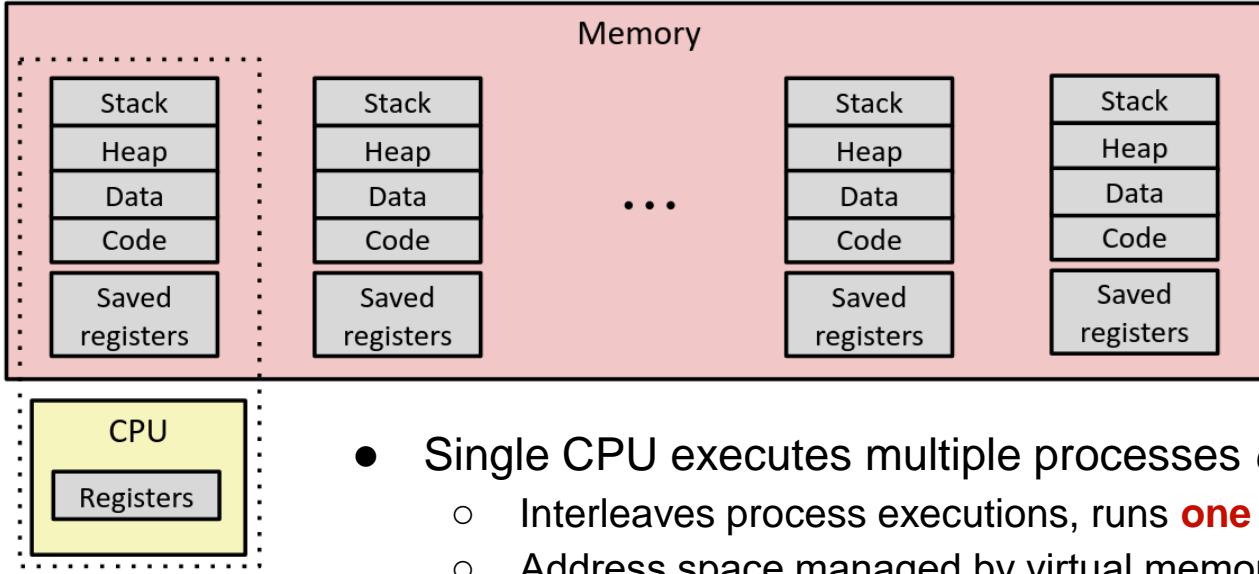


Multiprocessing: the Illusion



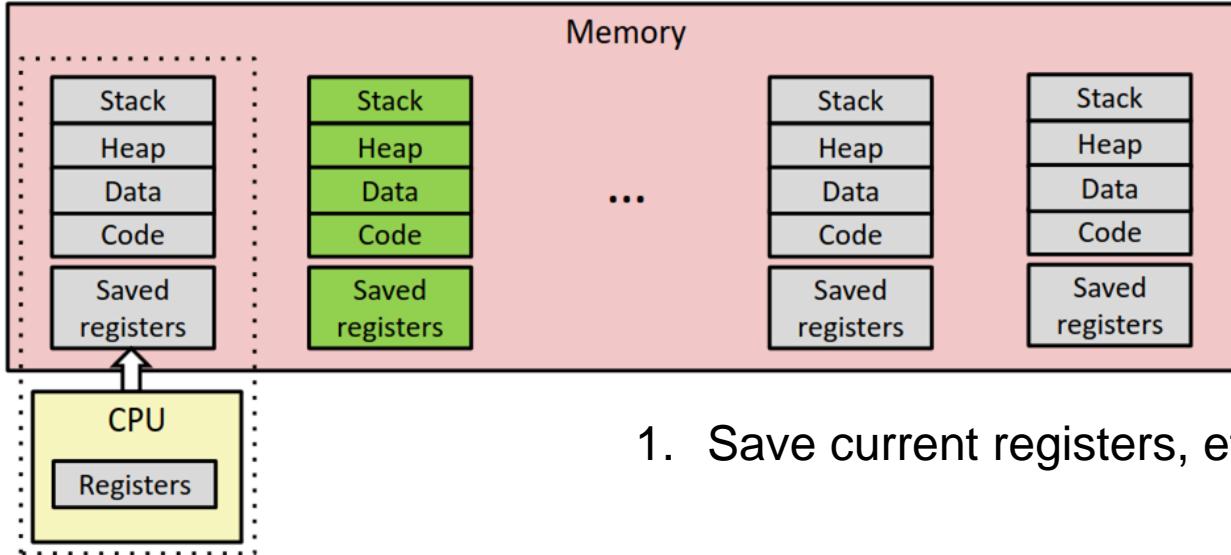
- Computer runs many processes simultaneously
 - Applications for one or more users
 - Web browsers, email clients, editors, ...
- Background tasks
 - Monitoring network & I/O devices

Multiprocessing: the Reality

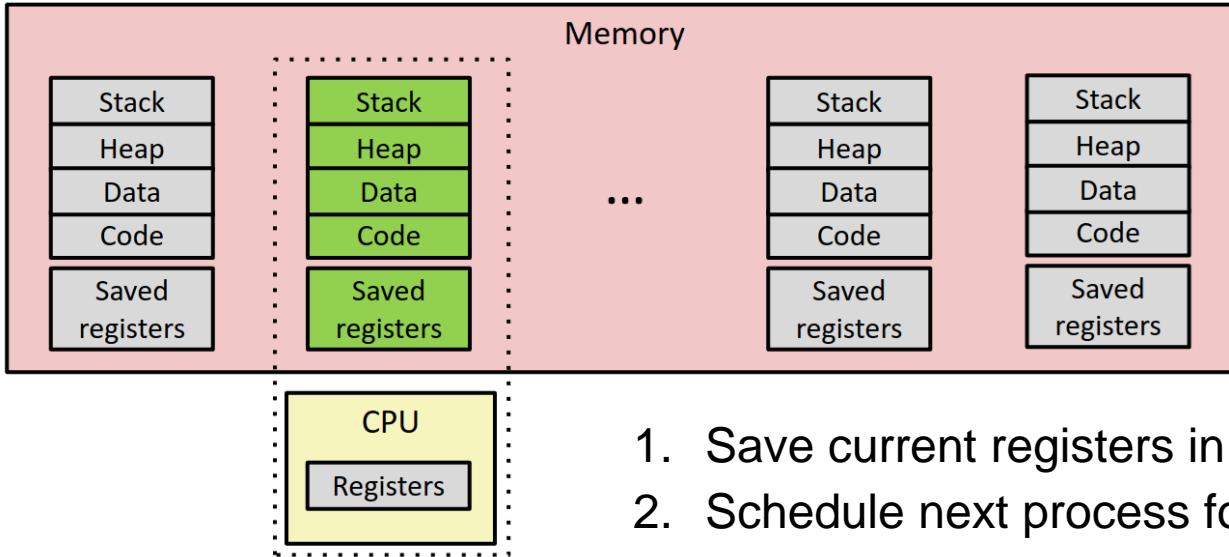


- Single CPU executes multiple processes *concurrently*
 - Interleaves process executions, runs **one at a time**
 - Address space managed by virtual memory system (we'll get to it!)
 - **Execution context** (register values, stack, etc.) saved in memory when process isn't running

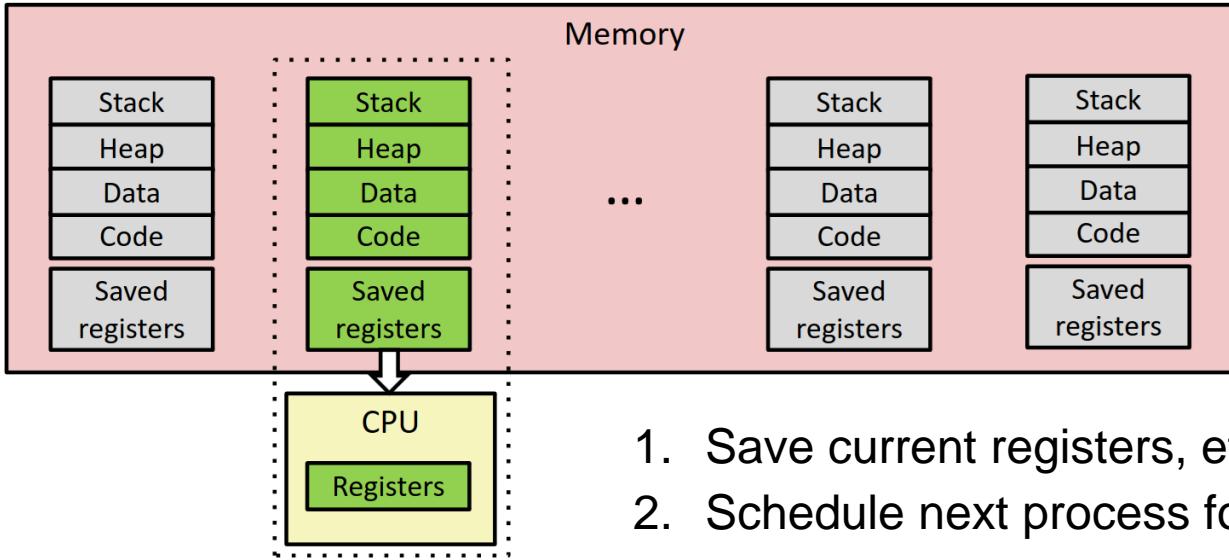
Context Switch steps



Context Switch steps (pt 2)

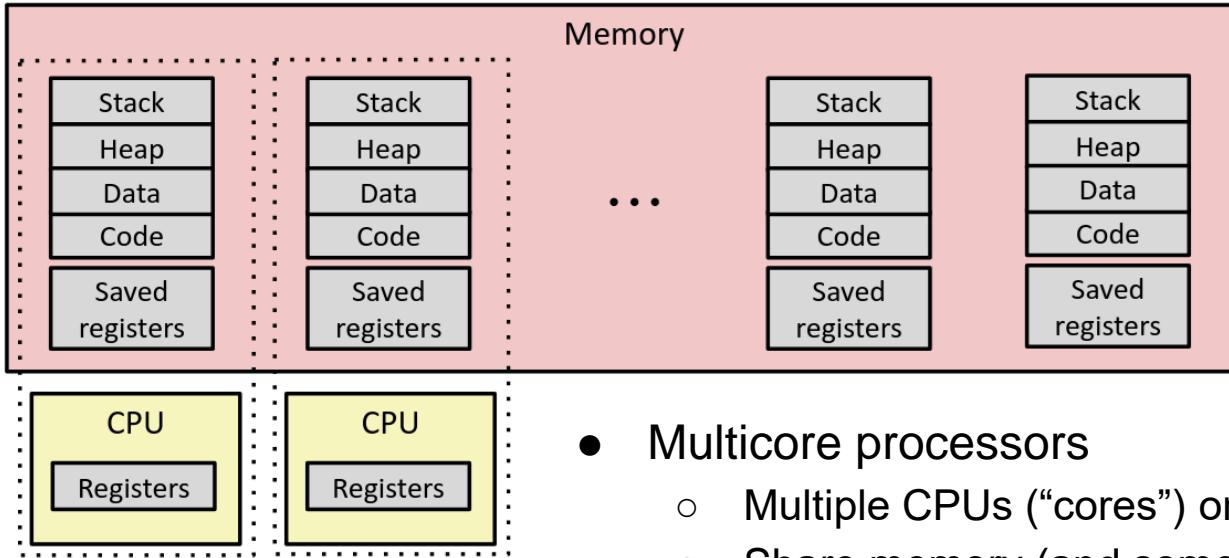


Context Switch steps (pt 3)



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

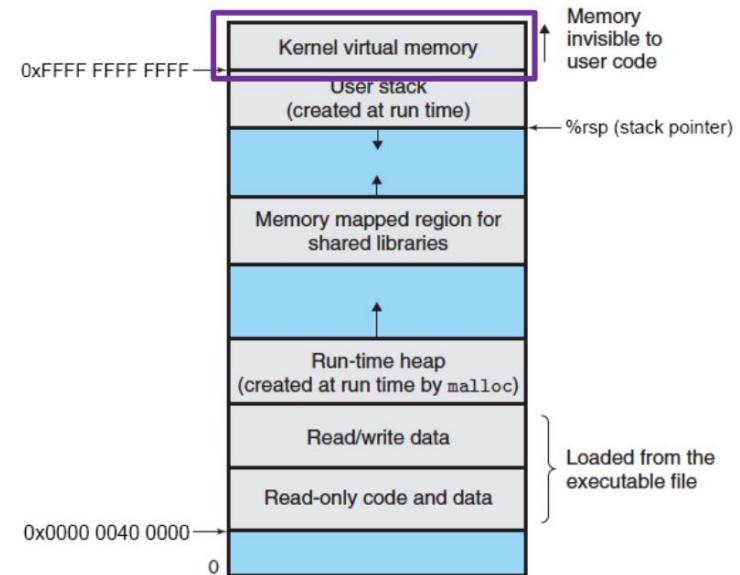


- Multicore processors
 - Multiple CPUs (“cores”) on one chip
 - Share memory (and some caches)
 - Each can execute a separate process
 - **Still constantly switching**

Context Switching

Assume only one CPU core

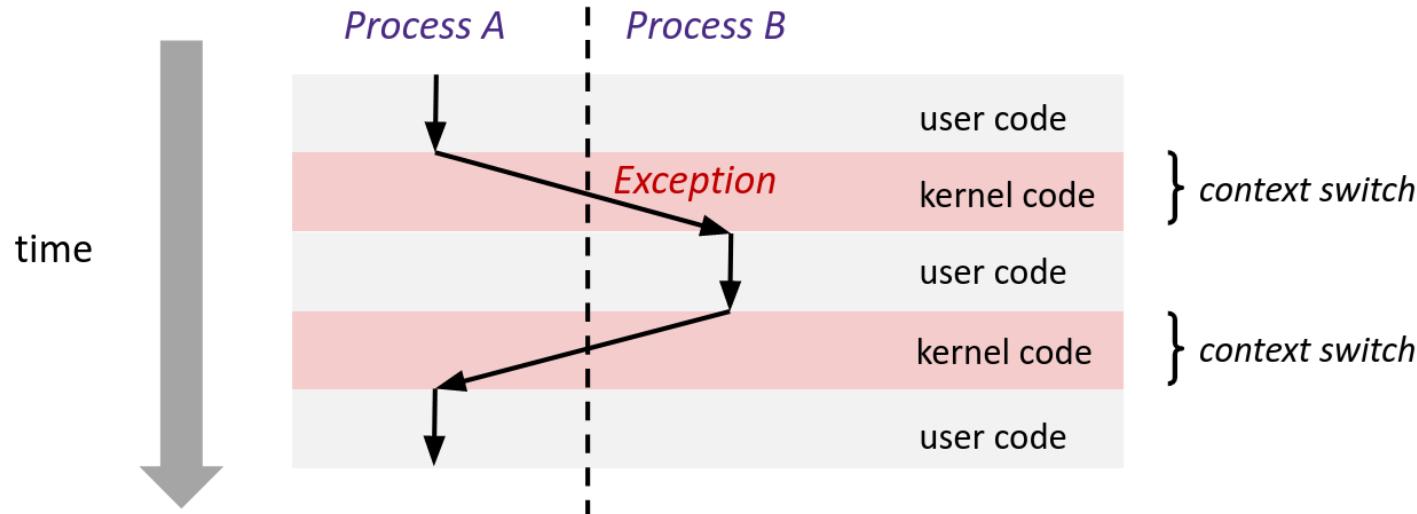
- Processes are managed by a shared chunk of OS code called the **kernel**
 - The kernel is not a separate process, but rather runs as part of each user process
- In x86-64 Linux:
 - Same address in each process' kernel memory refers to the same shared memory location*



Context Switching (pt 2)

Assume only one CPU core

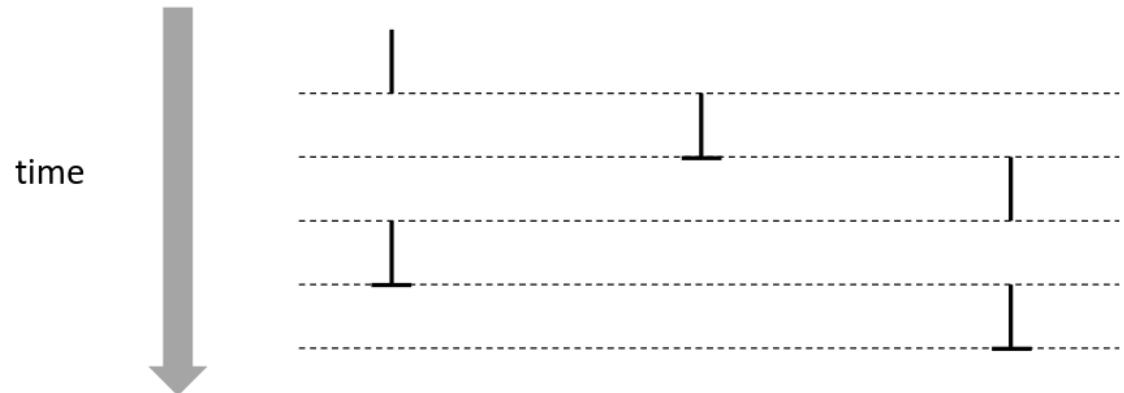
- Context switch **passes control flow** from one process to another and is performed using kernel code
- Can happen for a variety of reasons (process terminated, timer interrupt, etc.)



Concurrency

Assume only one CPU core

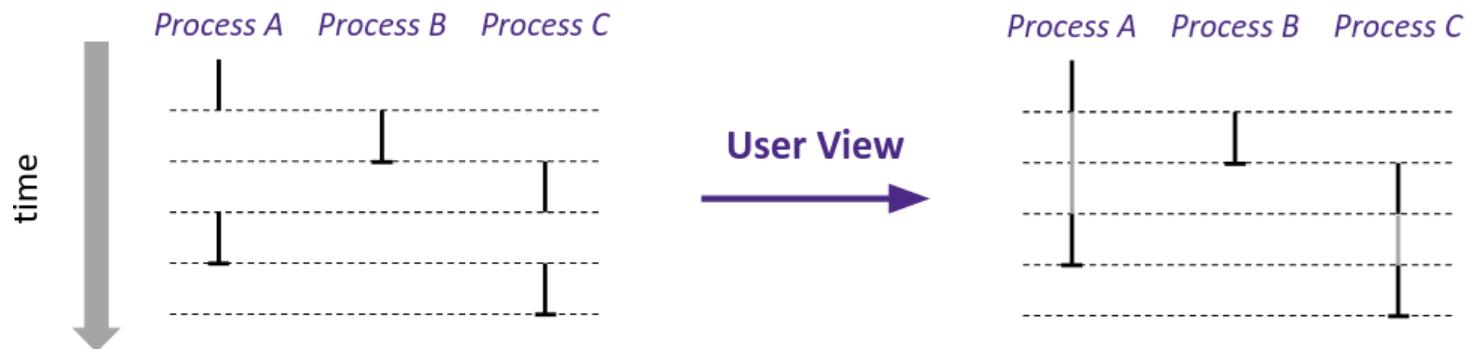
- Two processes are **concurrent** if their instruction executions/flows overlap in time
 - i.e. one starts before the other has *completely* finished executing
 - Otherwise, they are **sequential**
- Example:
 - Concurrent: A&B, A&C
 - Sequential: B&C



User's View of Concurrency

Assume only one CPU core

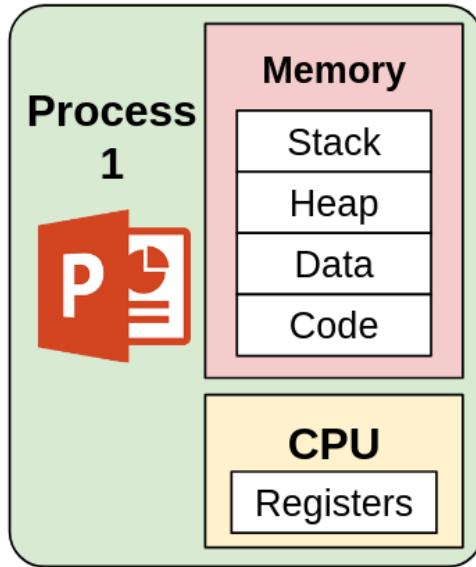
- Control flows for concurrent processes are physically disjoint in time
 - CPU executes instructions for one process at a time
- However, we can *think* of them as if they're running at the same time, in parallel



Lecture Topics

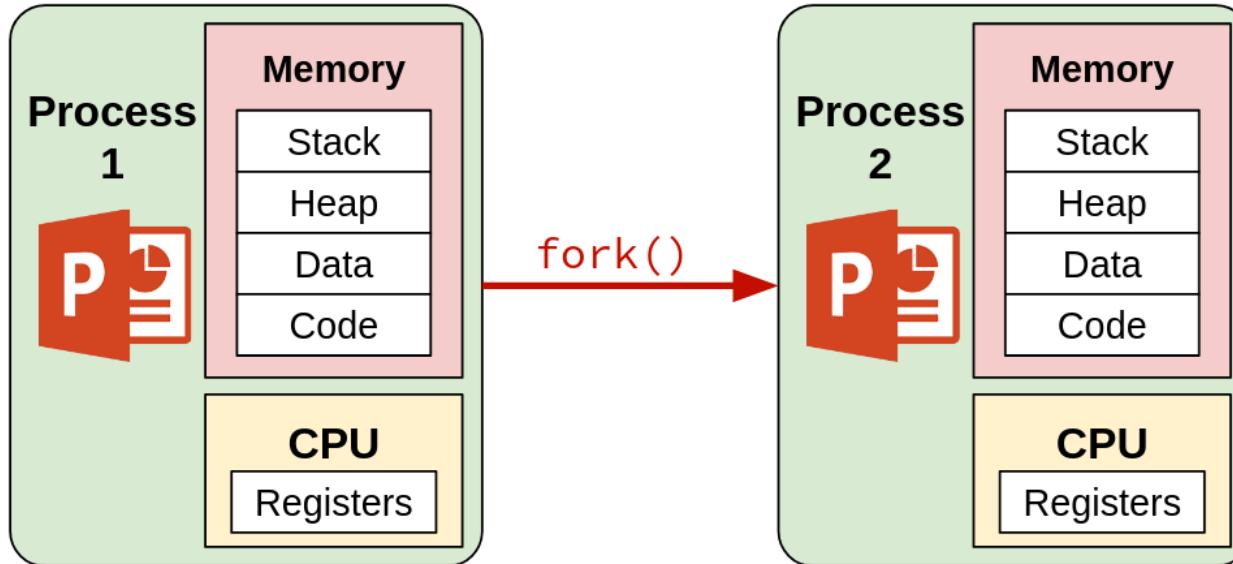
- System Control Flow
 - Control flow
 - Exceptional control flow
 - Asynchronous exceptions (interrupts)
 - Synchronous exceptions (traps & faults)
- Processes and context switching
 - Creating new processes
 - fork() and exec*()
 - Ending a process
 - exit(), wait(), waitpid()
 - Zombies

Creating New Processes and Programs



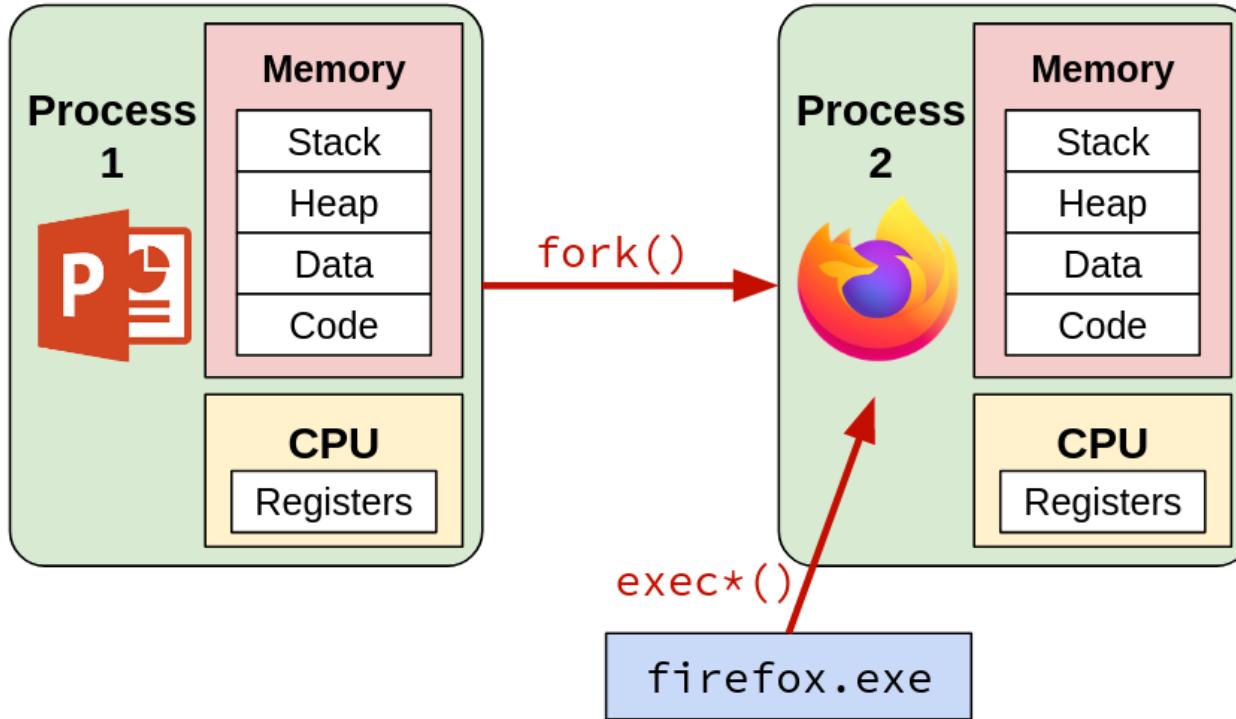
firefox.exe

Creating New Processes and Programs (pt 2)



firefox.exe

Creating New Processes and Programs (pt 3)

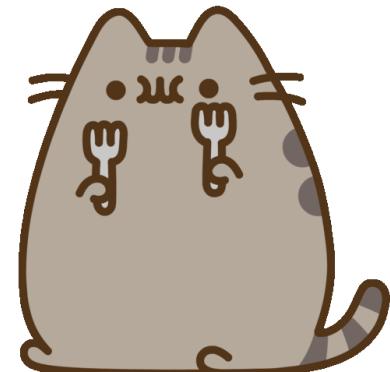


Creating New Processes and Programs

- **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