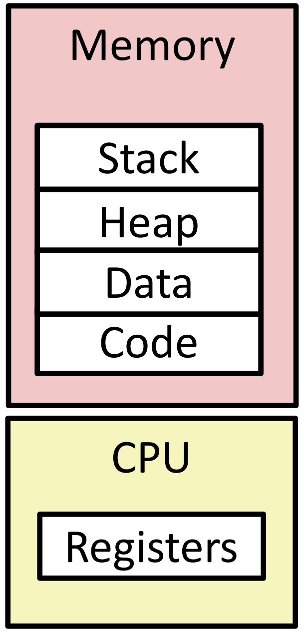
Process

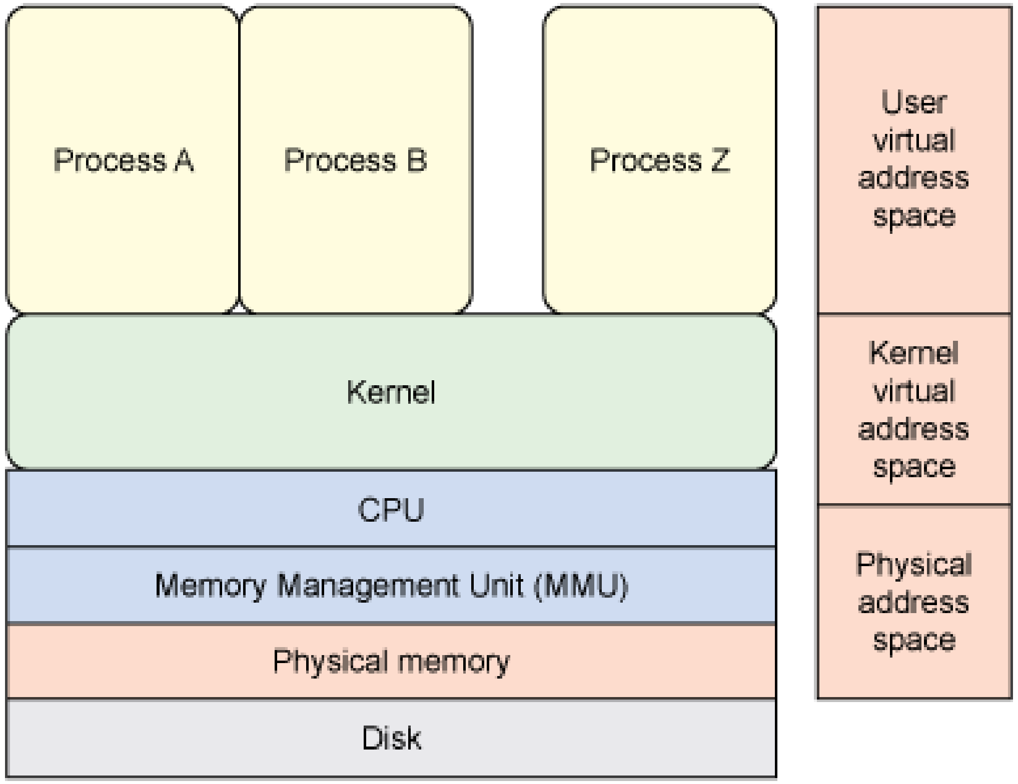
Process

- Definition: 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”

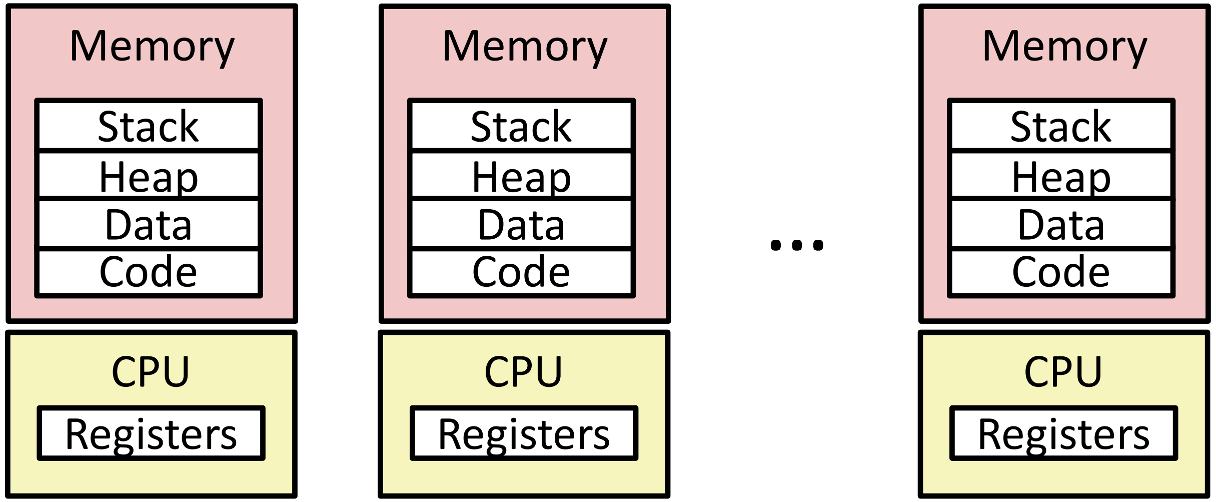
- Process provides each program with two key abstractions:
  - *Logical control flow*
    - Each program seems to have exclusive use of the **CPU**
    - Provided by kernel mechanism called *context switching*
  - *Private address space*
    - Each program seems to have exclusive use of **main memory**.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



Address Space



Multi-processing Illusion

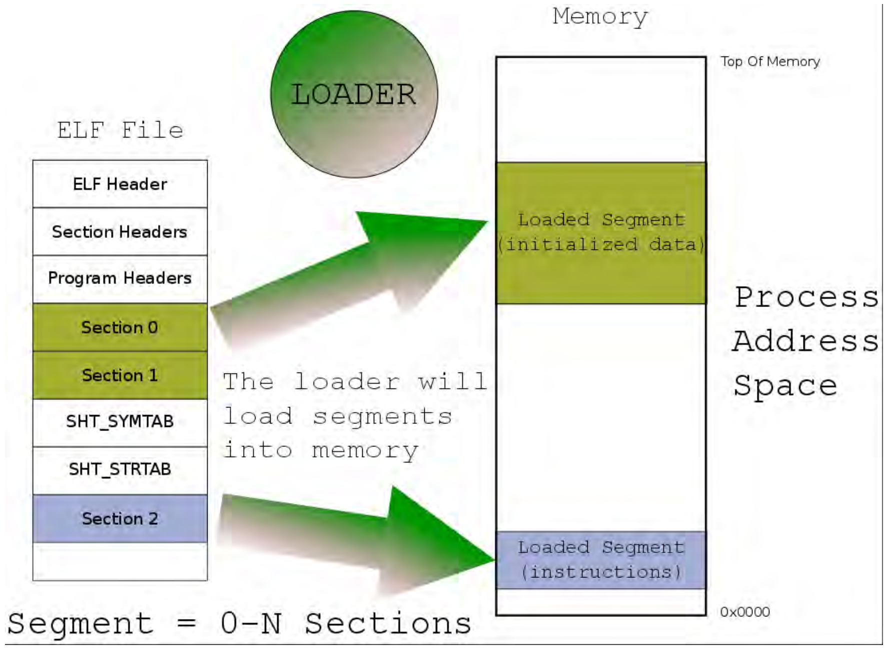


- Computer runs many processes simultaneously
  - Applications for one or more users
    - Web browsers, email clients, editors, …
  - Background tasks
    - Monitoring network & I/O devices

Processes vs. Programs

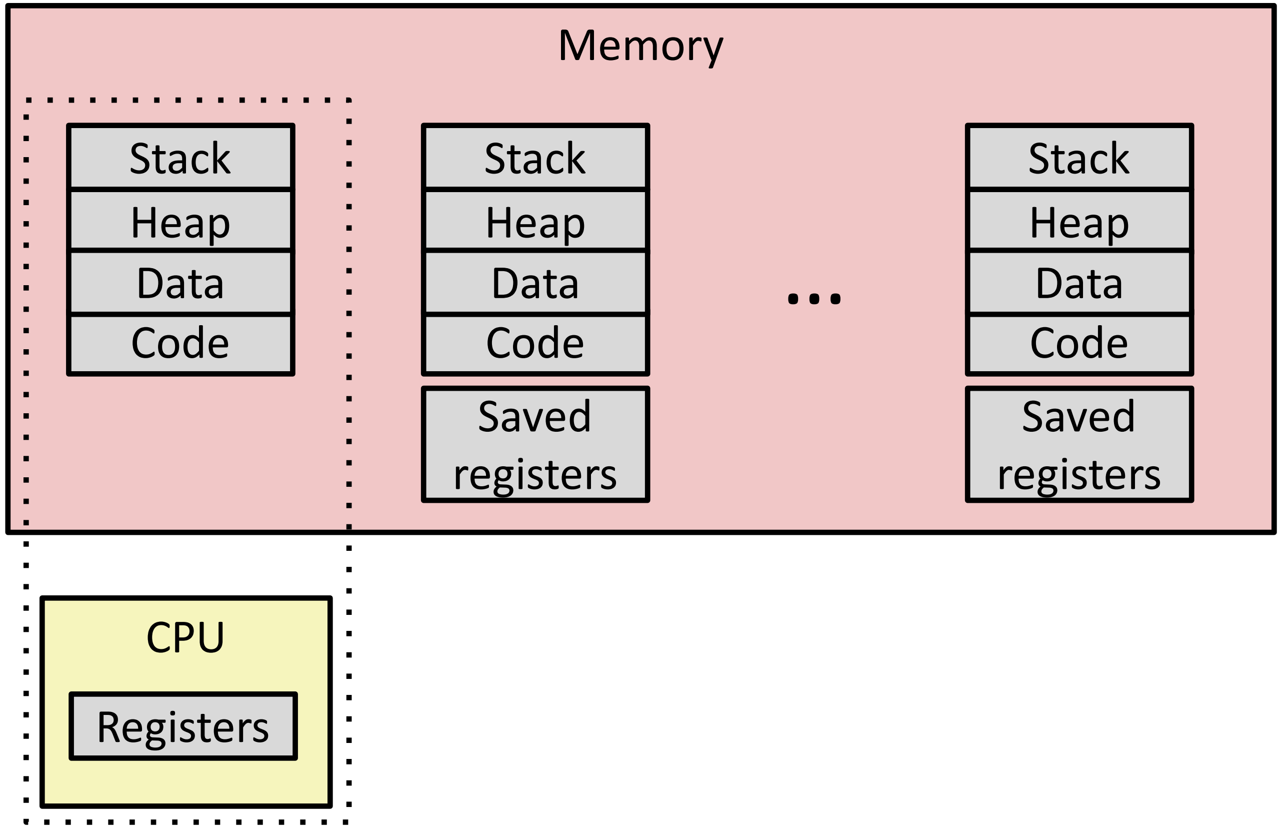
- A process is different than a program
  - Program: Static code and static data
  - Process: Dynamic instance of code and data
- Can have multiple process instances of same program
  - Can have multiple processes of the same program
    Example: many users can run “ls” at the same time

Turn Program into Process



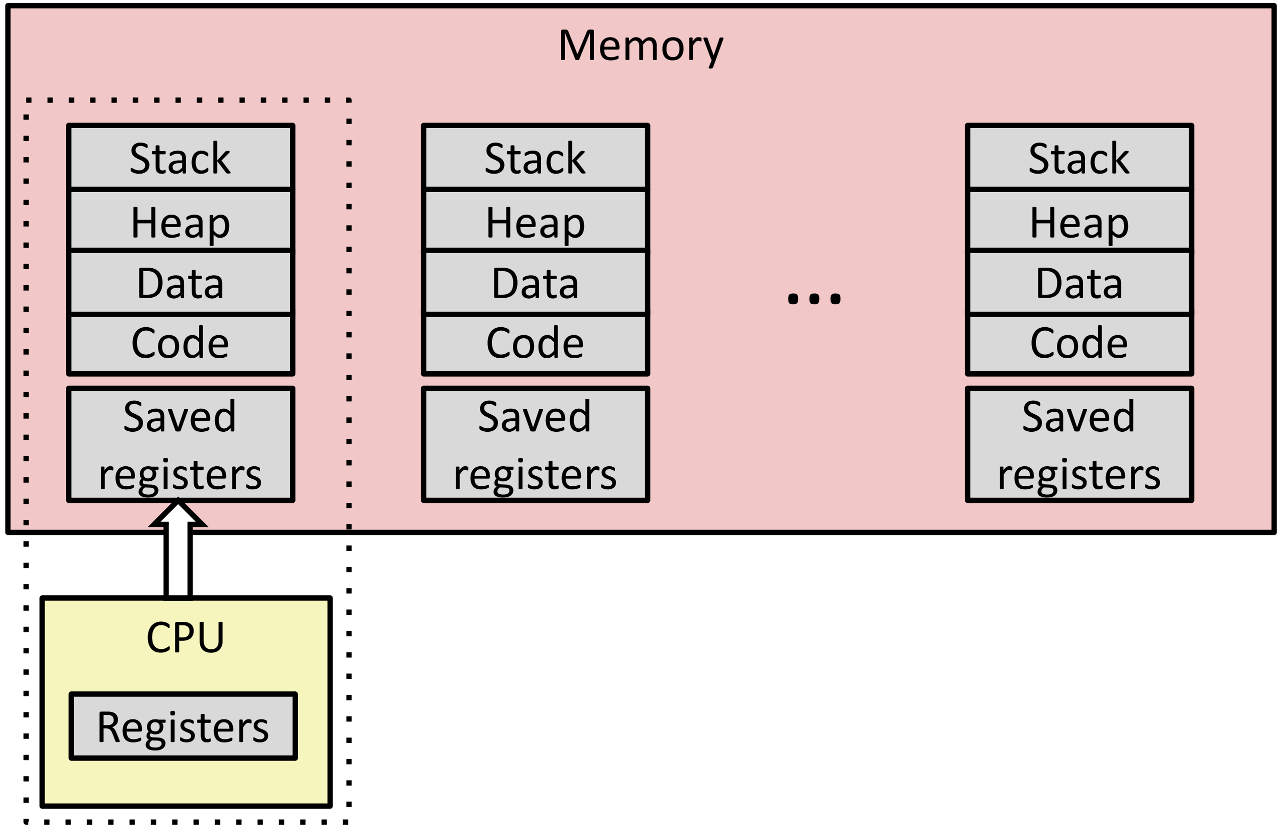
When loading an executable

Multiprocessing



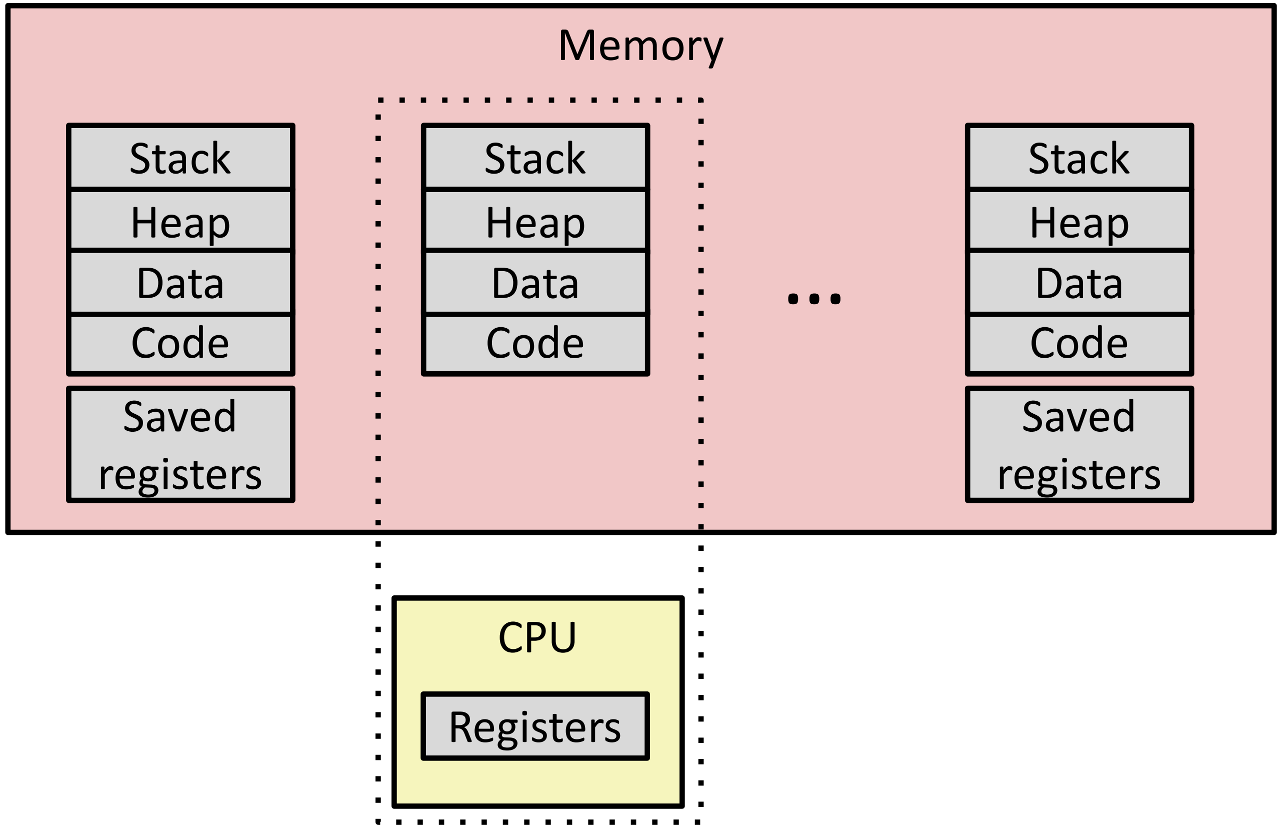
- Single processor executes multiple processes concurrently
  - Process executions interleaved (multitasking)
  - Address spaces managed by virtual memory system (later in course)
  - Register values for nonexecuting processes saved in memory

Multiprocessing



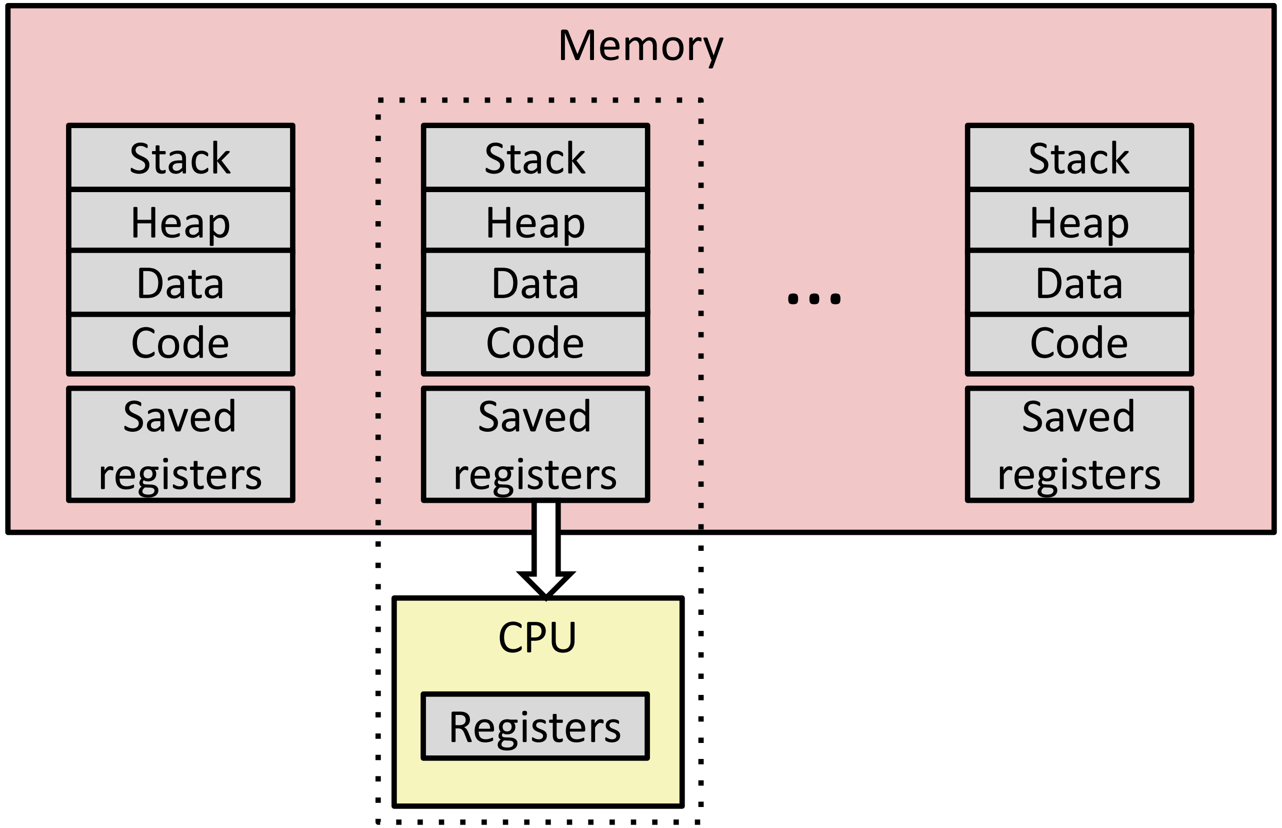
- Save current registers in memory

Multiprocessing



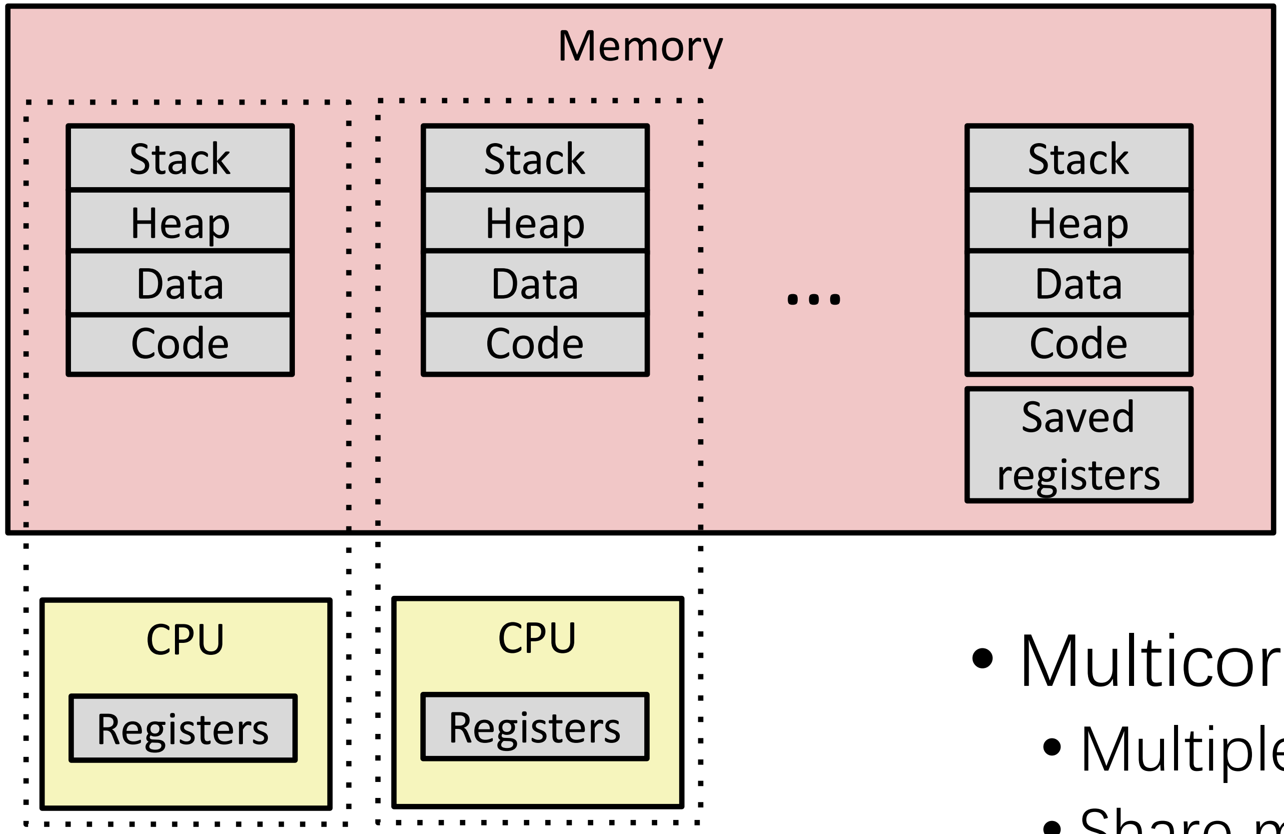
- Schedule next process for execution

Multiprocessing



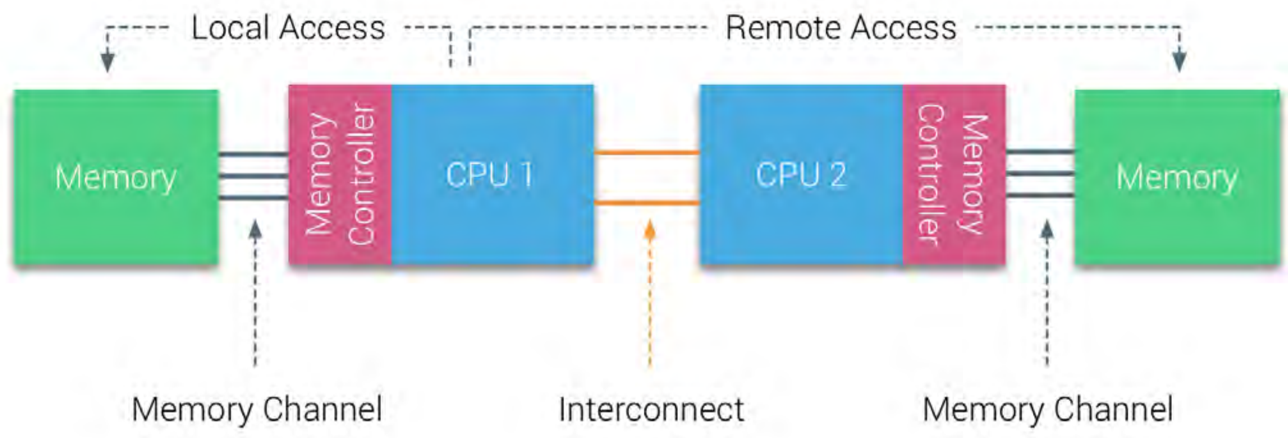
- Load saved registers and switch address space (context switch)

Multiprocessing



- Multicore processors (Symmetrical)
  - Multiple CPU cores on single chip
  - Share main memory (and some caches)
  - Each can execute a separate process
    - Scheduling of processors onto cores done by kernel

UMA (SMP) vs NUMA



UMA vs. NUMA

Uniform memory access	Non-uniform memory access
Offers limited bandwidth.	Offers relatively more bandwidth than UMA.
Since it uses a single memory controller, it is slower than NUMA.	It uses multiple memory controls to speed up operations compared to UMA.
Memory access time is equal or balanced.	Memory access time is unequal.
It is mainly used in time-sharing and general-purpose applications.	It offers better performance and speed, making it suitable for real-time and time-critical applications.

Process Creation

Two ways to create a process

- Build a new empty process from scratch
- Copy an existing process and change it appropriately

Option 1: New process from scratch

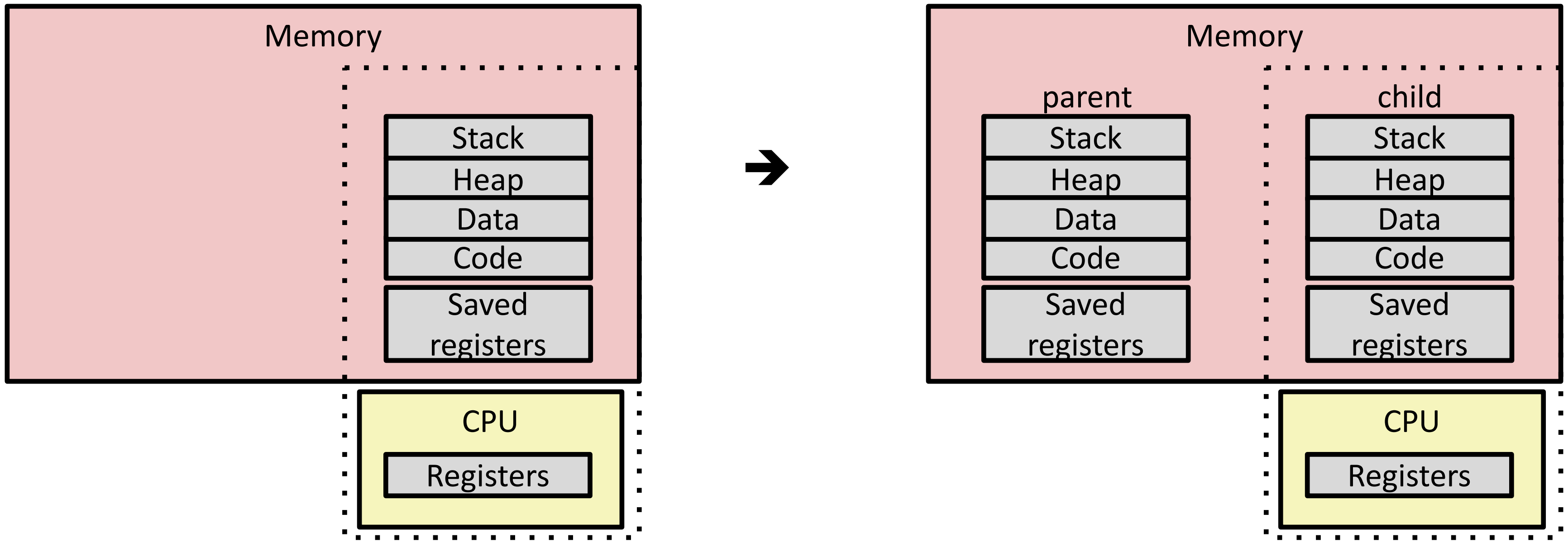
- Steps
  - Load specified code and data into memory;
  - Create empty stack
  - Create and initialize PCB (make look like context-switch)
  - Put process on ready list
- Advantages: No wasted work
- Disadvantages: Difficult to setup process correctly and to express all possible options
  - Process permissions, where to write I/O, environment variables
  - Example: WindowsNT has call with 10 arguments

Process Creation

Option 2: Clone existing process and change

- Example: Unix fork() and exec()
  - Fork(): Clones calling process
  - Exec(char *file): Overlays file image on calling process
- fork()
  - Stop current process and save its state
  - Make copy of code, data, stack, and PCB
  - Add new PCB to ready list
  - Any changes needed to child process?
- exec(char *file)
  - Replace current data and code segments with those in specified file
- Advantages: Flexible, clean, simple
- Disadvantages: Wasteful to perform copy and then overwrite of memory

Conceptual View of `fork`



- Make complete copy of execution state
  - Designate one as parent and one as child
  - Resume execution of parent or child

Creating Processes

- *Parent process* creates a new running *child process* by calling `fork`

- `int fork(void)`
  - Returns 0 to the child process, child's PID to parent process
  - Child is *almost* identical to parent:
    - Child get an identical (but separate) copy of the parent's virtual address space.
    - Child gets identical copies of the parent's open file descriptors
    - Child has a different PID than the parent

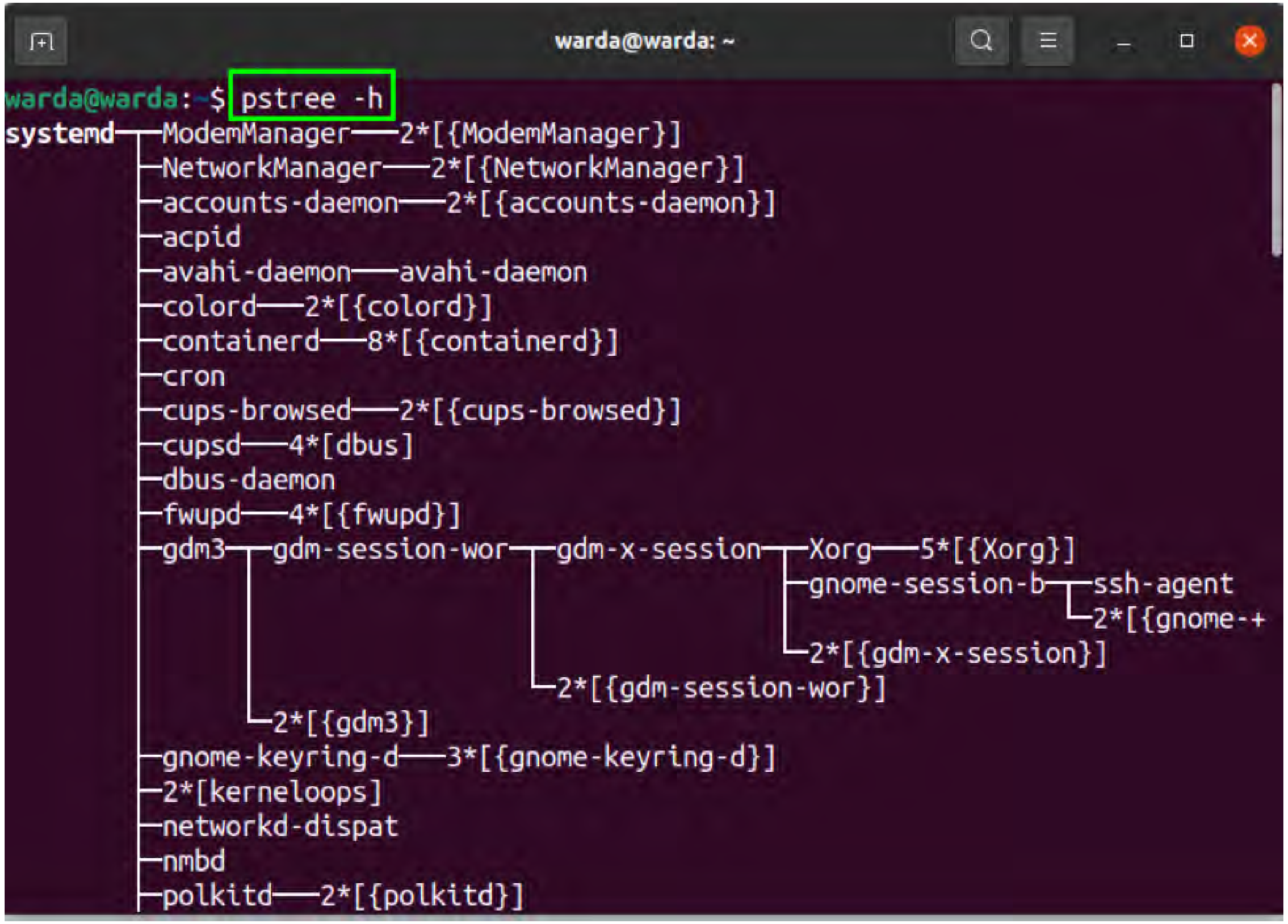
- `fork` is interesting (and often confusing) because
  it is called *once* but returns *twice*

Unix Process Creation

How are Unix shells implemented?

```
While (1) {
  Char *cmd = getcmd();
  Int retval = fork();
  If (retval == 0) {
       // This is the child process
       // Setup the child’s process environment here
       // E.g., where is standard I/O, how to handle signals?
       exec(cmd);
       // exec does not return if it succeeds
       printf(“ERROR: Could not execute %s\n”, cmd);
       exit(1);
  } else {
       // This is the parent process; Wait for child to finish
       int pid = retval;
       wait(pid);
  }
}
```

Modeling `fork` with Process Graphs



Process Termination

- By invoking exit(0), the process notifies OS kernel to terminate.
- When to recycle resources inside kernel?
  - Just like when you graduate from the university.
  - When the parent call wait() to wait for the child's termination, it can reap all the resources corresponding to the child
  - What if the parent never wait?
  - The child becomes zombie, and the init process (pid = 1) will reap all.

```
linux> ./forks 7 &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
  PID TTY          TIME CMD
 6639 ttyp9    00:00:03 forks
 6640 ttyp9    00:00:00 forks <defunct>
 6641 ttyp9    00:00:00 ps
```

`wait`: Synchronizing with Children

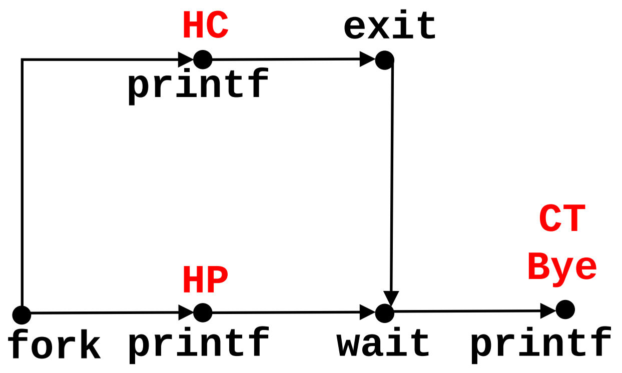
- Parent reaps a child by calling the `wait` function
- `int wait(int *child_status)`
  - Suspends current process until one of its children terminates
  - Return value is the **`pid`** of the child process that terminated
  - If **`child_status != NULL`**, then the integer it points to will be set to a value that indicates reason the child terminated and the exit status:
    - Checked using macros defined in `wait.h`
      - `WIFEXITED, WEXITSTATUS, WIFSIGNALED, WTERMSIG, WIFSTOPPED, WSTOPSIG, WIFCONTINUED`
      - See textbook for details

Common Usage of wait()

```c
void fork9() {
    int child_status;

    if (fork() == 0) {
        printf("HC: hello from child\n");
       exit(0);
    } else {
        printf("HP: hello from parent\n");
        wait(&child_status);
        printf("CT: child has terminated\n");
    }
    printf("Bye\n");
}
```

forks.c



Feasible output(s):

HC	HP
HP	HC
CT	CT
Bye	Bye

Infeasible output:

HP
CT
Bye
HC

`execve:` Loading and Running Programs

- `int execve(char *filename, char *argv[], char *envp[])`
- Loads and runs in the current process:
  - Executable file **`filename`**
    - Can be object file or script file beginning with `#!interpreter` (e.g., `#!/bin/bash`)
  - …with argument list **`argv`**
    - By convention **`argv[0]==filename`**
  - …and environment variable list **`envp`**
    - "name=value" strings (e.g., `USER=droh`)
    - `getenv, putenv, printenv`
- Overwrites code, data, and stack
  - Retains PID, open files and signal context
- Called once and never returns
  - …except if there is an error

CPU Virtualization

- How to convince a process that it exclusively uses the CPU core assigned to it?
  - Just like using wifi router to share a broadband.
  - A CPU core can be shared in time and space manner.
- How to start execution?
  - Just jump to the pre-defined entrance of a program (i.e., main())
  - What if the process runs while(1);?
  - How to manage resources?

Restricting Process

- How can we ensure a process can't harm others?
- **Solution: privilege levels supported by**
  - User processes run in user mode (restricted mode)
  - OS runs in kernel mode (not restricted)
    - Instructions for interacting with devices
    - Instructions for resource management
    - Could have many privilege levels (advanced topic)
- How can process access device?
  - System calls (function call implemented by OS)
  - Change privilege level through system call (trap)

System Call

P can only see its own memory because of **user mode**
(other areas, including kernel, are hidden)

System Call

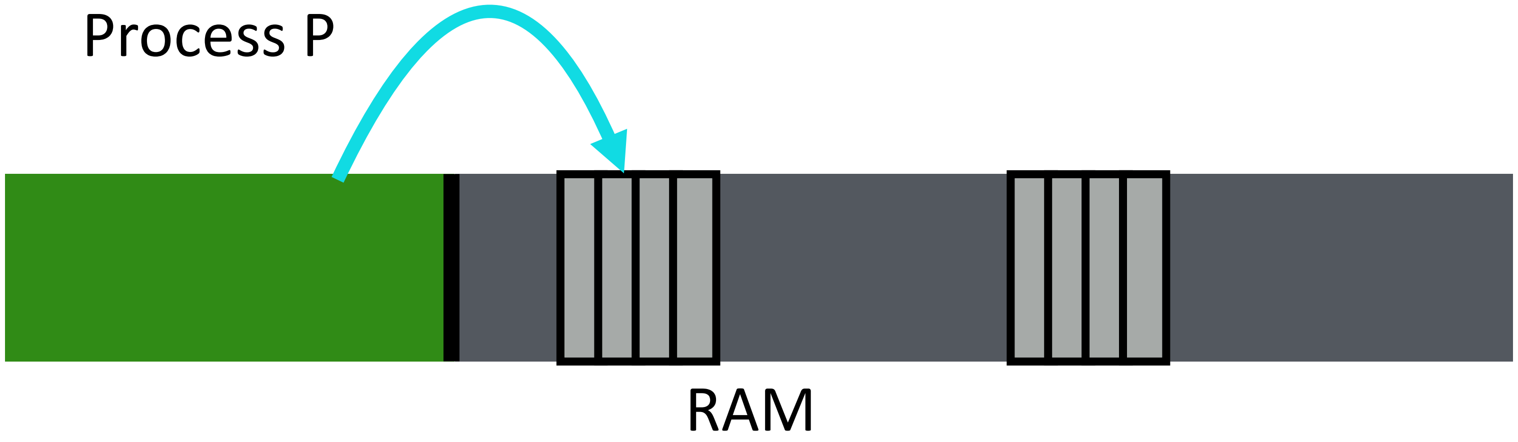
P wants to call read() but no way to call it directly

System Call

read():

```
movl $6, %eax;    int $64
```

System Call

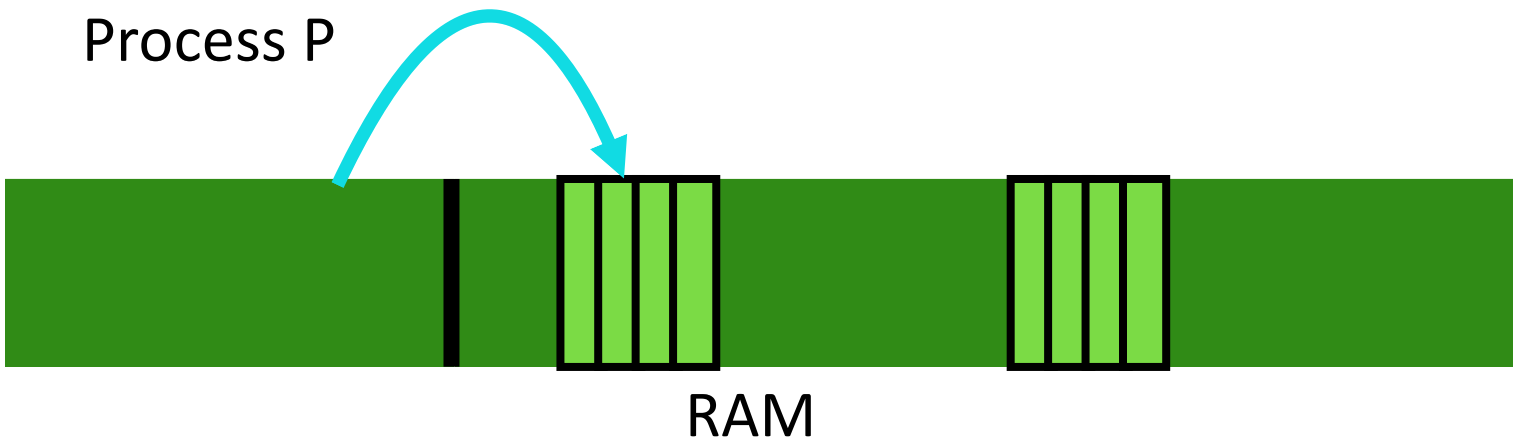



```
movl $6, %eax;    int $64
```

syscall-table index

trap-table index

System Call



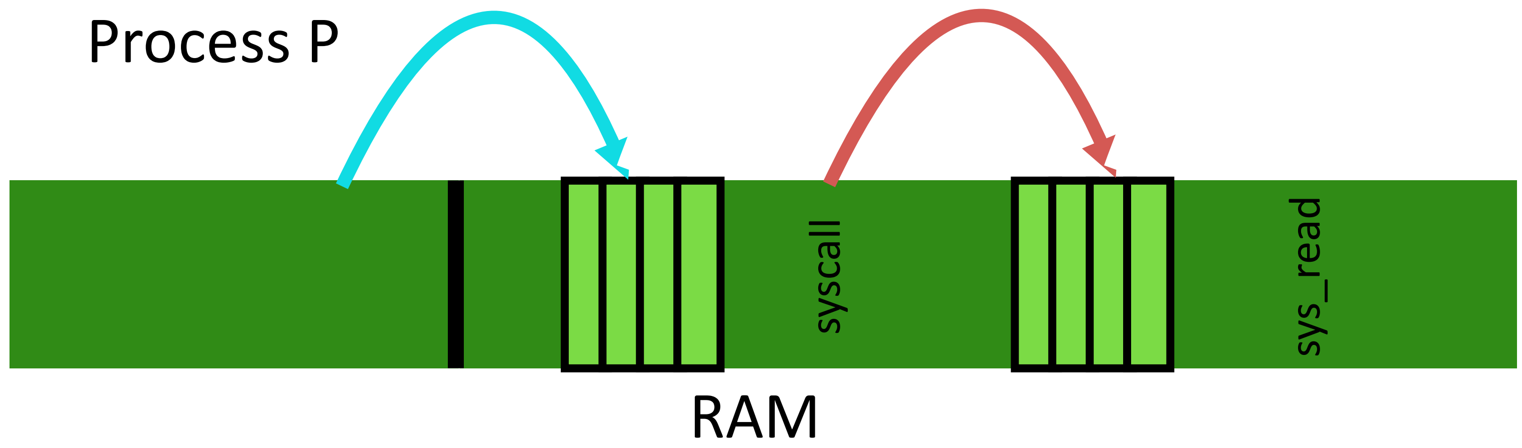

```
movl $6, %eax;    int $64
```

syscall-table index

trap-table index

Kernel mode: we can do anything!

System Call



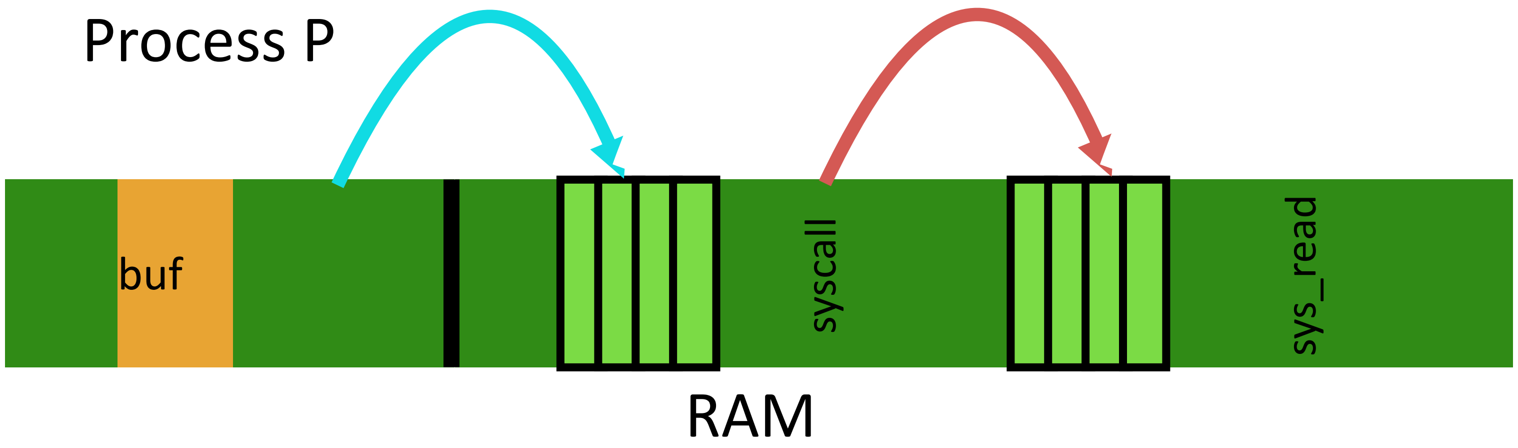

```
movl $6, %eax;    int $64
```

syscall-table index

trap-table index

Follow entries to correct system call code

System Call




```
movl $6, %eax;    int $64
```

syscall-table index

trap-table index

Kernel can access user memory to fill in user buffer

return-from-trap at end to return to Process P

Problem 2: How to take CPU away?

- OS requirements for multi-tasking
  - Mechanism
    - To switch between processes
  - Policy
    - To decide which process to schedule when
- Separation of policy and mechanism
  - Policy: Decision-maker to optimize some workload performance metric
    - Which process when?
    - Process **Scheduler**: Future lecture
  - Mechanism: Low-level code that implements the decision
    - How?
    - Process **Dispatcher**: Today's lecture

Dispatch Mechanism

- OS runs dispatch loop

```
while (1) {
        run process A for some time-slice
        stop process A and save its context
        load context of another process B
    }
```

Context-switch

- Question 1: How does dispatcher gain control?
- Question 2: What execution context must be saved and restored?

Q1: How does Dispatcher get control?

- Option 1: Cooperative Multi-tasking
  - Trust process to relinquish CPU to OS through traps
    - Examples: System call, page fault (access page not in main memory), or error (illegal instruction or divide by zero)
    - Provide special `yield()` system call
- Option 2: Preemptive
  - Guarantee OS can obtain control periodically
  - Enter OS by enabling periodic alarm clock
    - Hardware generates timer interrupt (CPU or separate chip)
    - Example: Every 10ms
  - User must not be able to disable timer interrupt

Q2: What Context must be saved?

- Dispatcher must track context of process when not running
  - Save context in process control block (PCB)
  - task_struct for Linux, or the marco current
- What information is stored in PCB?
  - Metainfo: PID, Process state (i.e., running, ready, or blocked)…
  - Execution state (all registers, PC, stack ptr)
  - Scheduling priority
  - Credentials (which resources can be accessed, owner)
  - Pointers to other allocated resources (e.g., open files)
  - …
- Requires special hardware support
  - Hardware saves process PC and PSR on interrupts

How does it work in x86

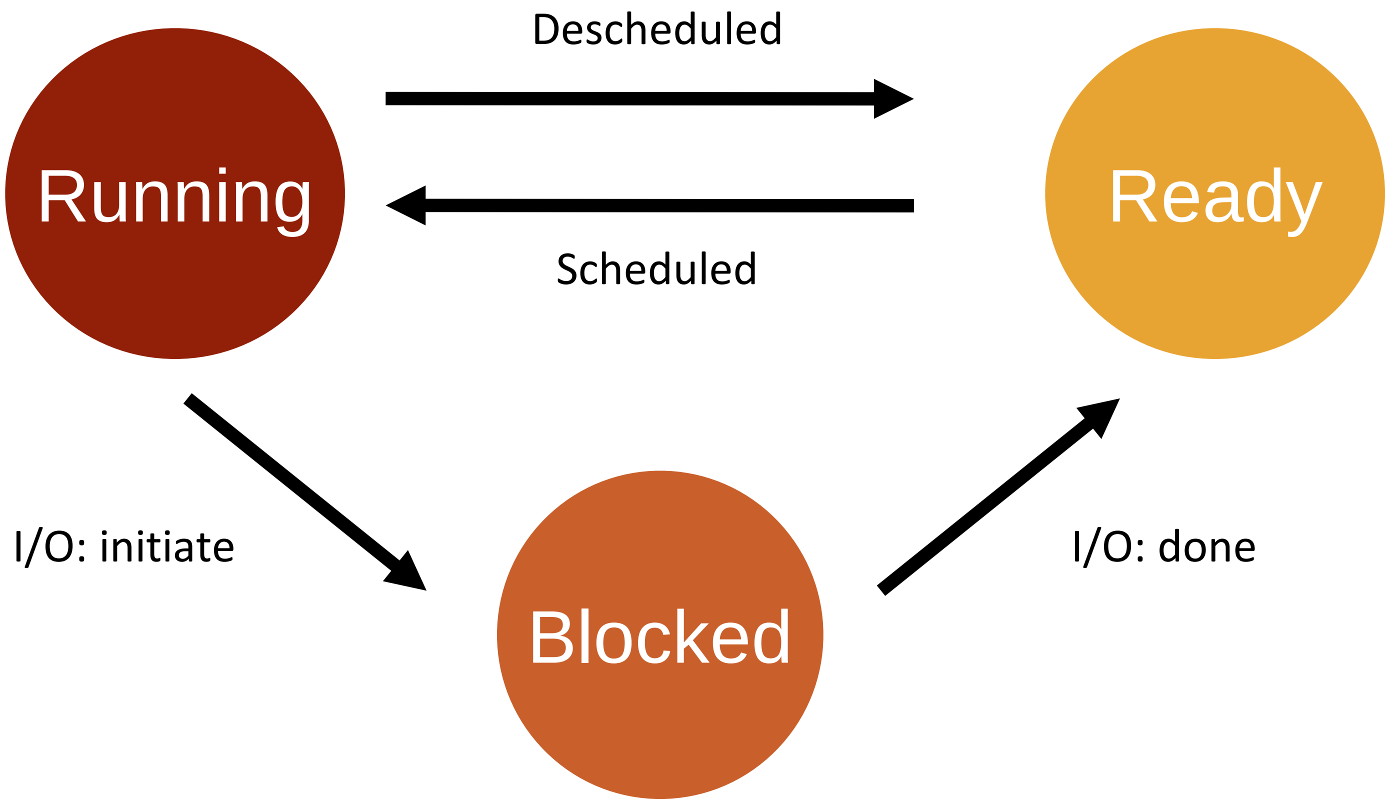
- Need for store old state
  - The processor needs a place to save the *old* processor state, such as EIP and CS, before the processor invoked the exception handler
  - This cannot be done by software
- Kernel stack for each process
  - A structure called the ***task state segment* (TSS)** specifies the segment selector and address where this stack lives.
  - The processor pushes (on this new stack) SS, ESP, EFLAGS, CS, EIP, and an optional error code.
  - Then it loads the CS and EIP from the interrupt descriptor, and sets the ESP and SS to refer to the new stack.

Problem 3: Slow Ops such as I/O?

- When running process performs a op that does not use CPU, OS switches to process that needs CPU (policy issues)

- OS must track mode of each process:
  - Running:
    - On the CPU (only one on a uniprocessor)
  - Ready:
    - Waiting for the CPU
  - Blocked:
    - Asleep: Waiting for I/O or synchronization to complete

State Transitions



Problem 3: Slow Ops such as I/O?

- OS must track every process in system
  - Each process identified by unique Process ID (PID)
- OS maintains queues of all processes
  - Ready queue: Contains all ready processes
  - Event queue: One logical queue per event
    - e.g., disk I/O and locks
    - Contains all processes waiting for that event to complete

- Next Topic: Policy for determining which **ready** process to run