CS5460: Operating Systems

Lecture 2: Processes

(Chapters 4, 5, 6)

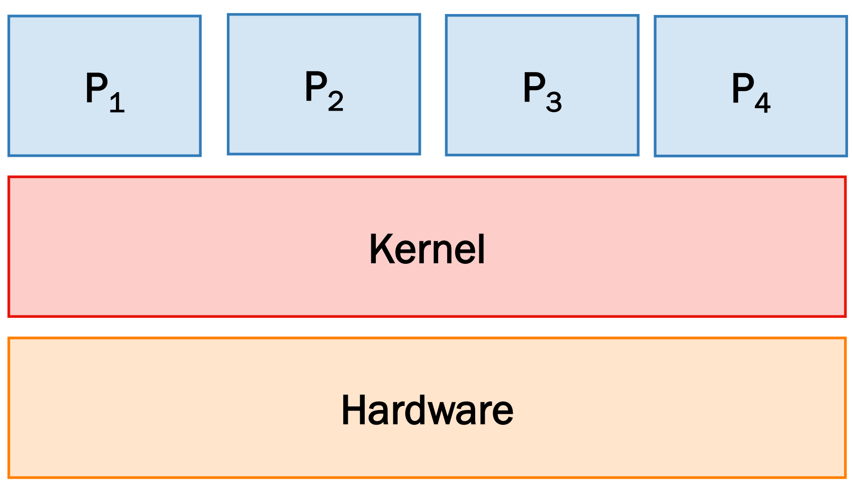
Slide Credit: Andrea Arpaci-Dusseau

Assignment 1

- Due Tue Feb 2

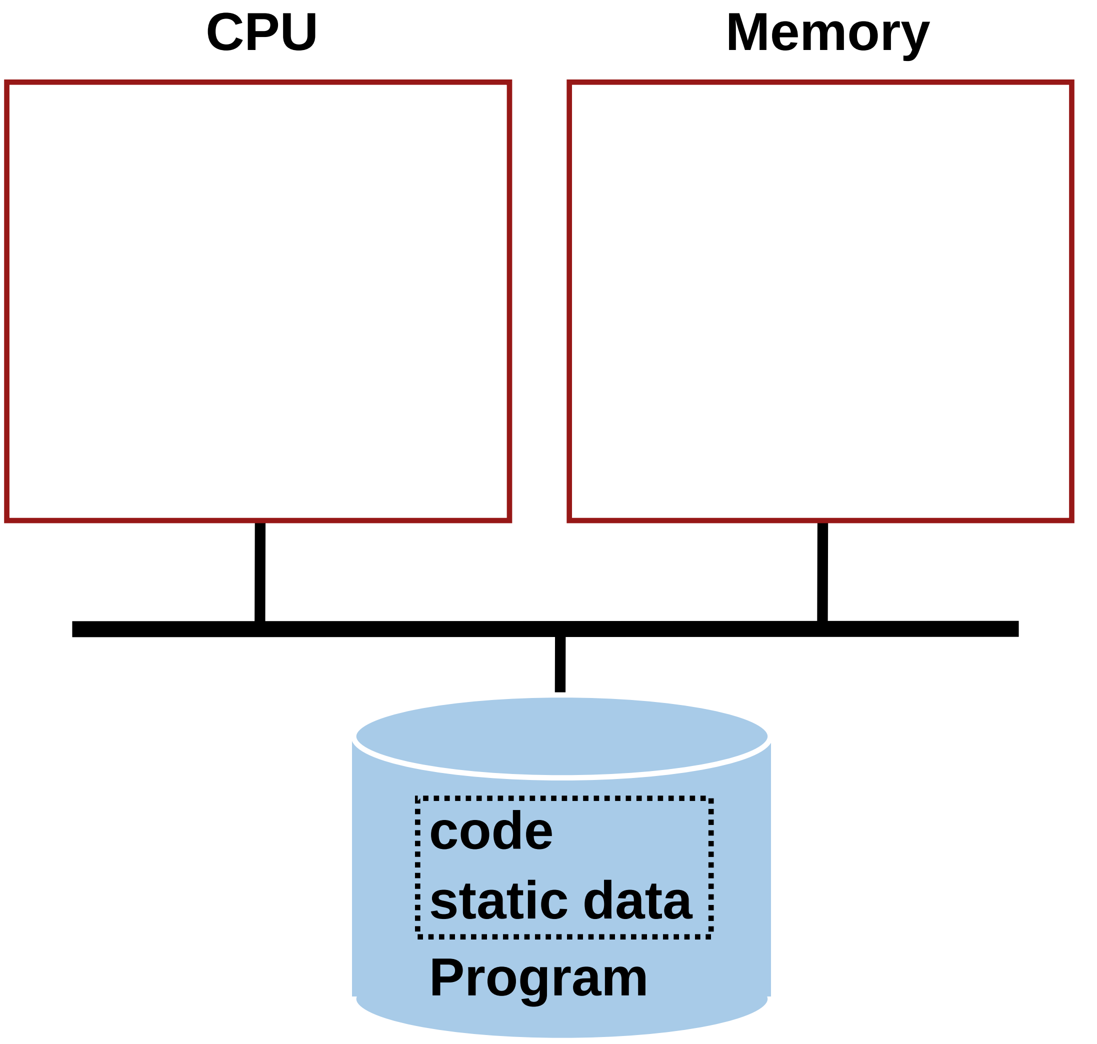
Isolating Processes

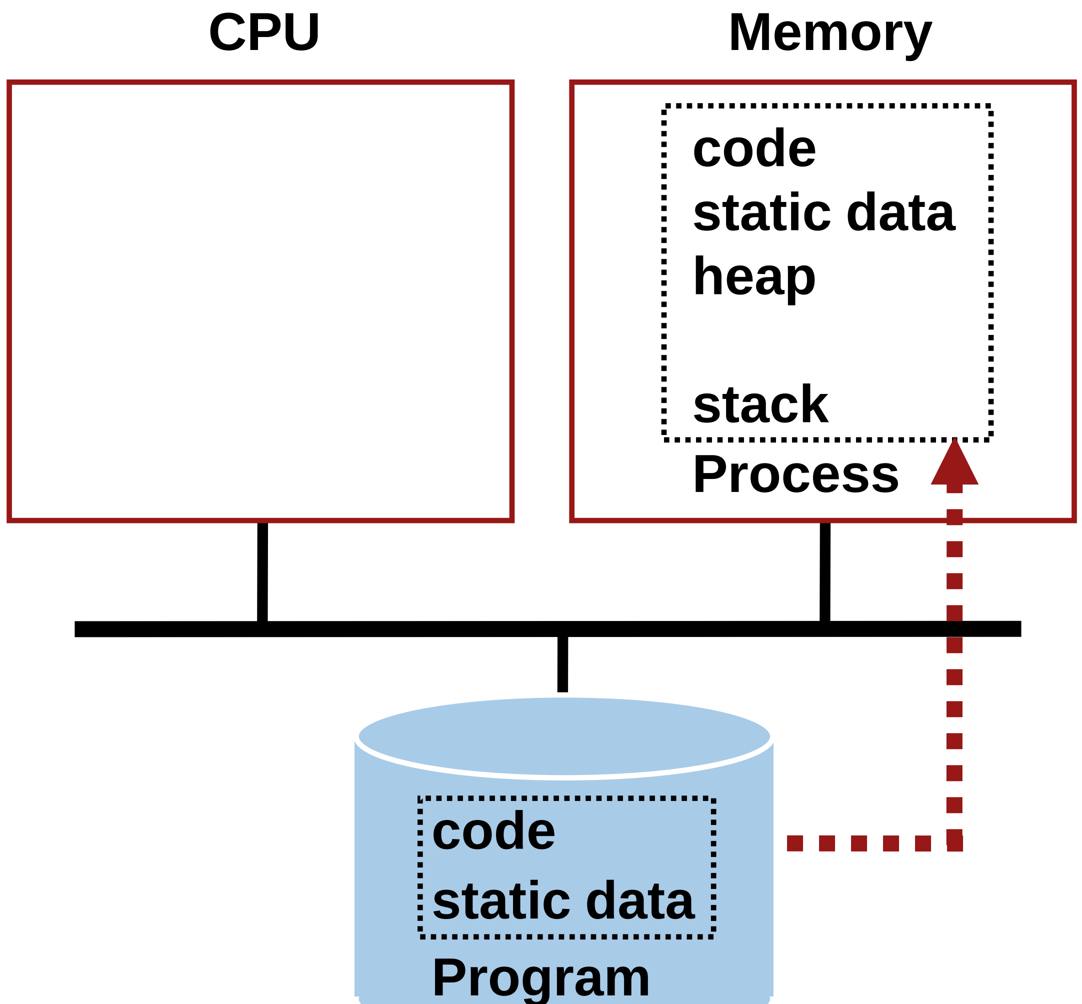
- Lots of running processes
- Each with own code, data
- Each need to interact with devices, memory, CPU
- How do we multiplex the hardware among them?
- How do we make this safe? Efficient?



What is a Process?

- Process: execution context of running program
- A process does not equal a program!
  - Process is an *instance* of a program
  - Many copies of same program can be running at same time
- OS manages a variety of activities
  - User programs
  - Batch jobs and scripts
  - System programs – print spool, file servers, net daemons
- Each of these activities is encapsulated in a process
- Everything happens either in kernel or a process
  - (Generally) the OS kernel is a program but not a process

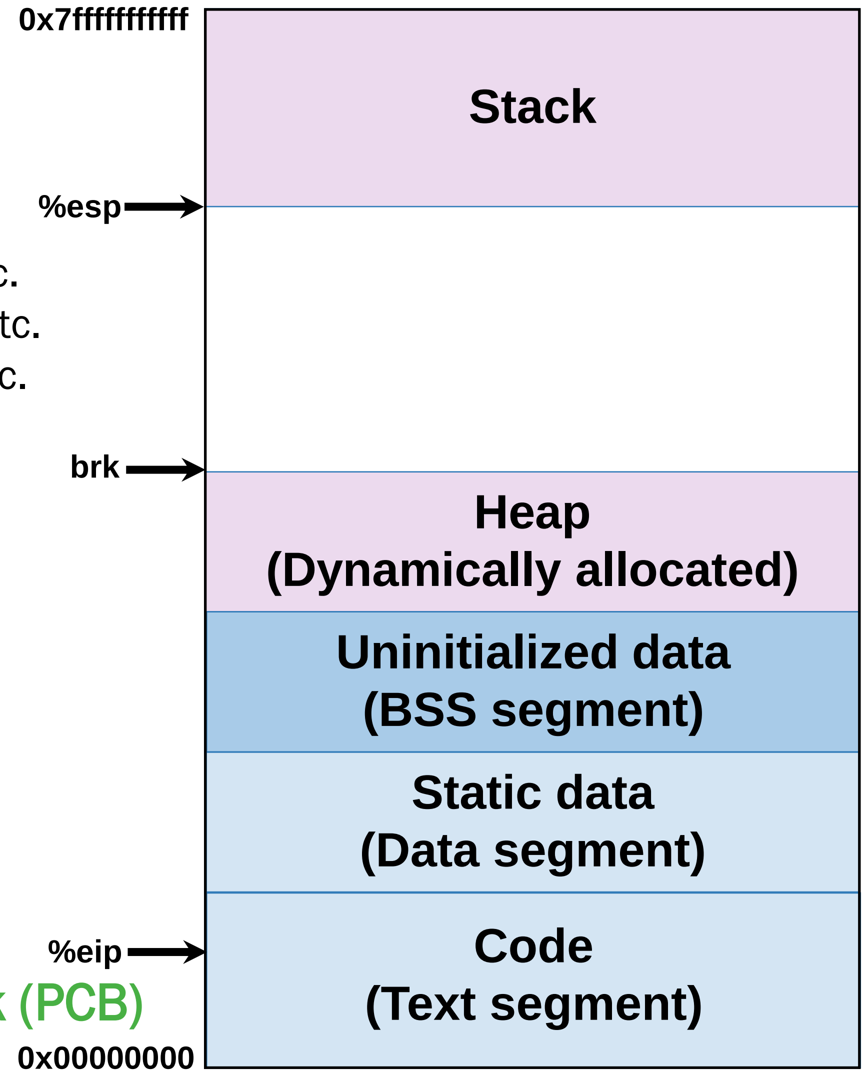

CPU
Memory
code
static data
Program


CPU
Memory
code
static data
heap
stack
Process
code
static data
Program

What is in a Process?

- Process state consists of:
  - Memory: code, data, heap, stack
  - Processor state: IP, registers, etc.
  - Kernel state:
    - Process state: ready, running, etc.
    - Resources: open files/sockets, etc.
    - Scheduling: priority, CPU time, etc.
- Address space consists of:
  - Code
  - Static data (data and BSS)
  - Dynamic data (heap and stack)
  - See: Unix "`size`" command
- Special pointers:
  - IP: current instruction being executed
  - brk: top of heap (explicitly moved)
  - SP: bottom of stack (implicitly moved)

All tracked in a Process Control Block (PCB)



Processes vs. Threads

- A process is different than a thread
- Thread: “Lightweight process” (LWP)
  - An execution stream that **shares an address space**
  - Multiple threads within a single process
- Example:
  - Two **processes** examining same memory address 0xffe84264 see ***different*** values (i.e., different contents)
  - Two **threads** examining memory address 0xffe84264 see ***same*** value (i.e., same contents)

Virtualizing the CPU

Goal:
Each process thinks it is alone is actively using CPU

Resources can be shared in time and space

Assume single uniprocessor
- Time-sharing (multi-processors: advanced issue)

Memory?
- Space-sharing (later)

Disk?
- Space-sharing (later)

Providing Good CPU Performance?

Direct execution

- Allow user process to run directly on hardware
- OS creates process, transfers control to start point (i.e., main())

Problems with direct execution?

1. Process could do something restricted
   Could read/write other process data (disk or memory)
2. Process could run forever (slow, buggy, or malicious)
   OS needs to be able to switch between processes
3. Process could do something slow (like I/O)
   OS wants to use resources efficiently and switch CPU to other process

Solution:

Limited direct execution – OS & hw maintain some control

Problem #1: Restricted Ops

How can we ensure user process can't harm others?

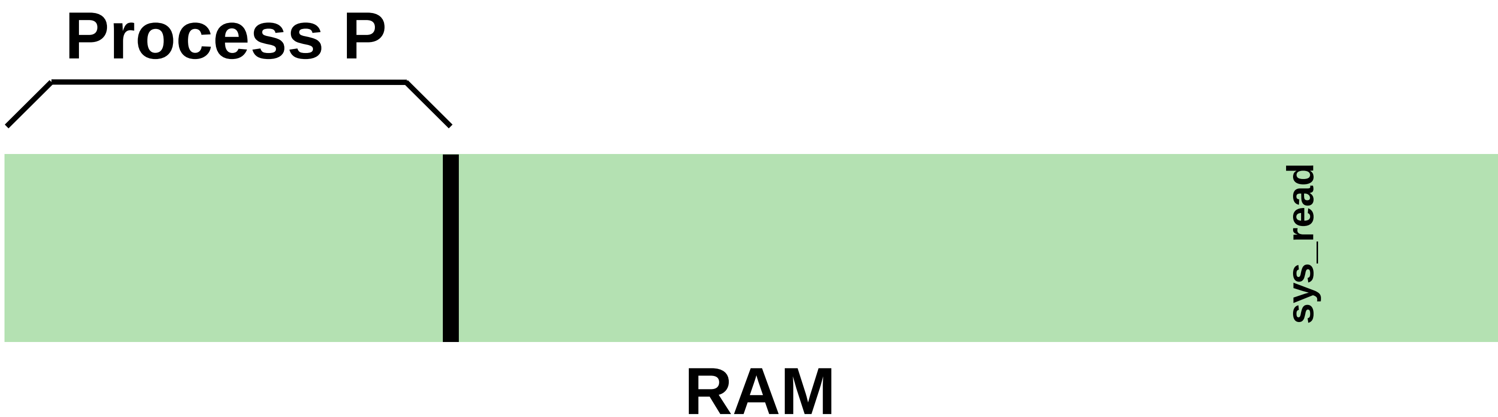
Solution: privilege levels supported by hw (status bit)

- User processes run in user mode (restricted mode) (Ring 3)
- OS runs in kernel mode (not restricted) (Ring 0)
  - Instructions for interacting with devices
  - Access to all memory
  - Ability to reconfigure CPU control registers (IDT, PTBR/CR3)

How can processes access devices?

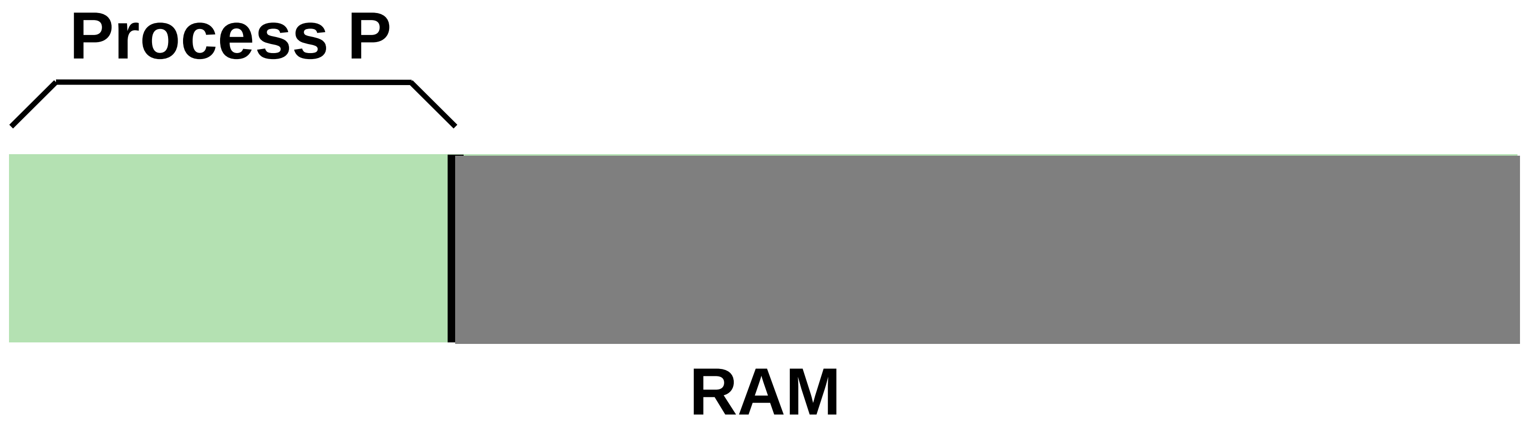
- System calls (function call implemented by OS)
- Change privilege level through system call (trap)

System Call



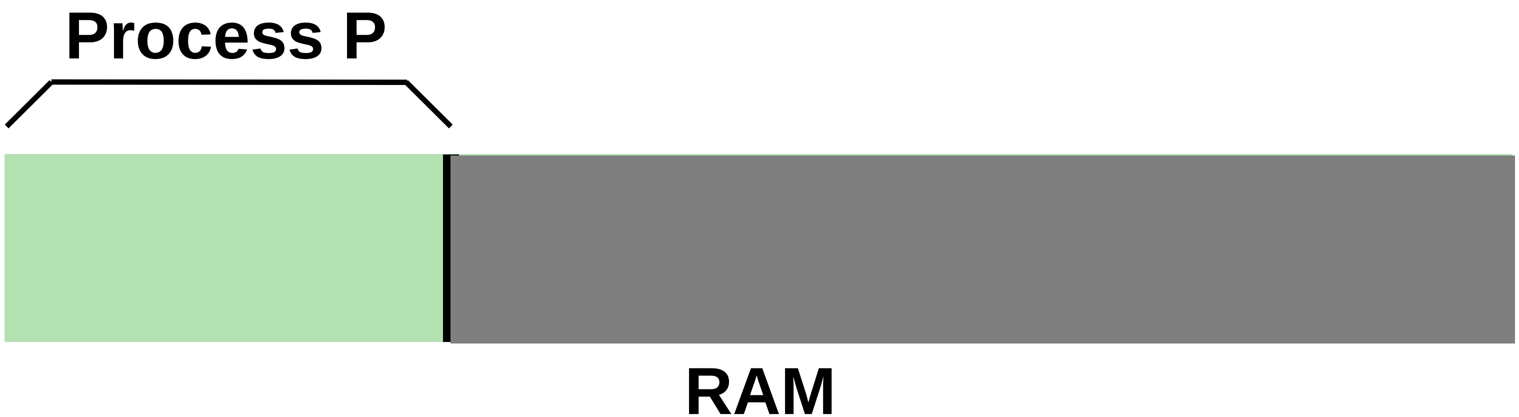
P wants to call read()

System Call



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

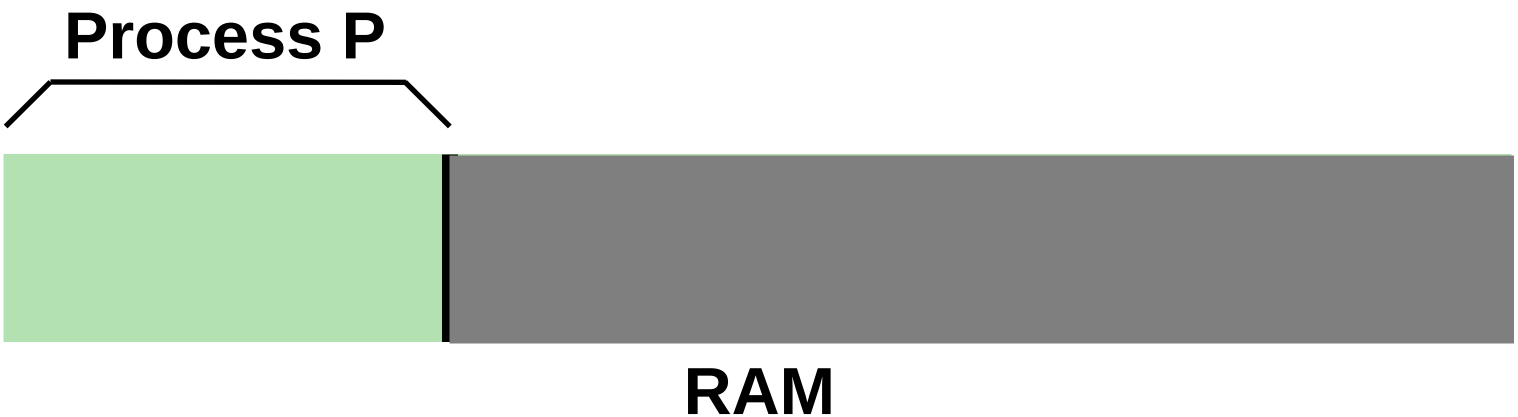
System Call



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

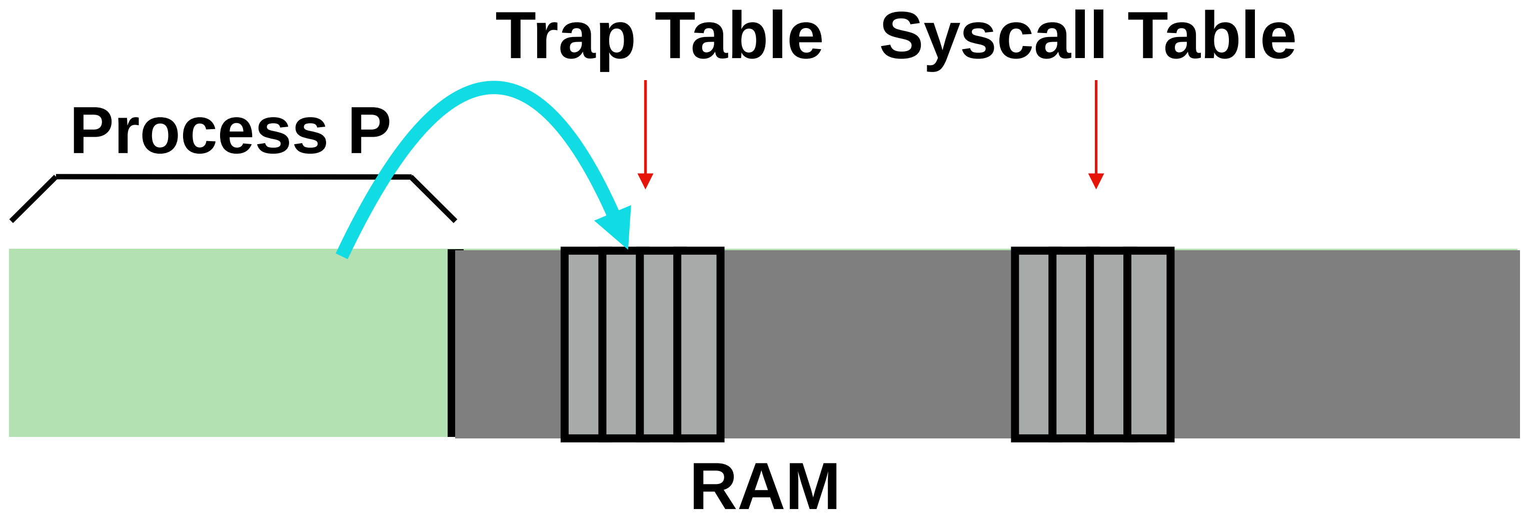
~~`callq sys_read`~~

System Call



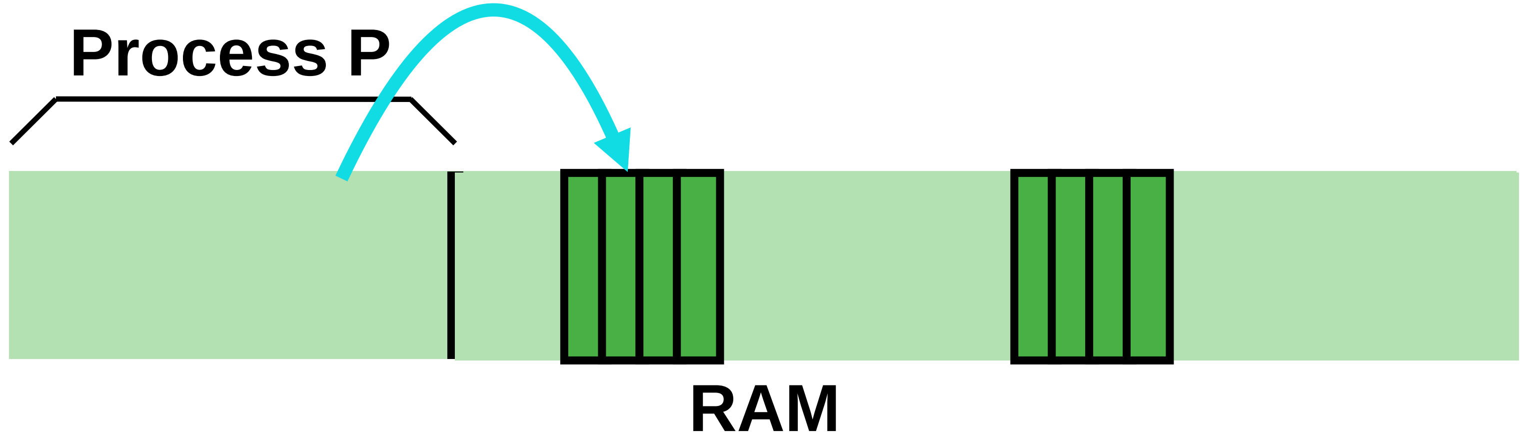

```
movl $6, %eax;   int $64
```

System Call




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

System Call



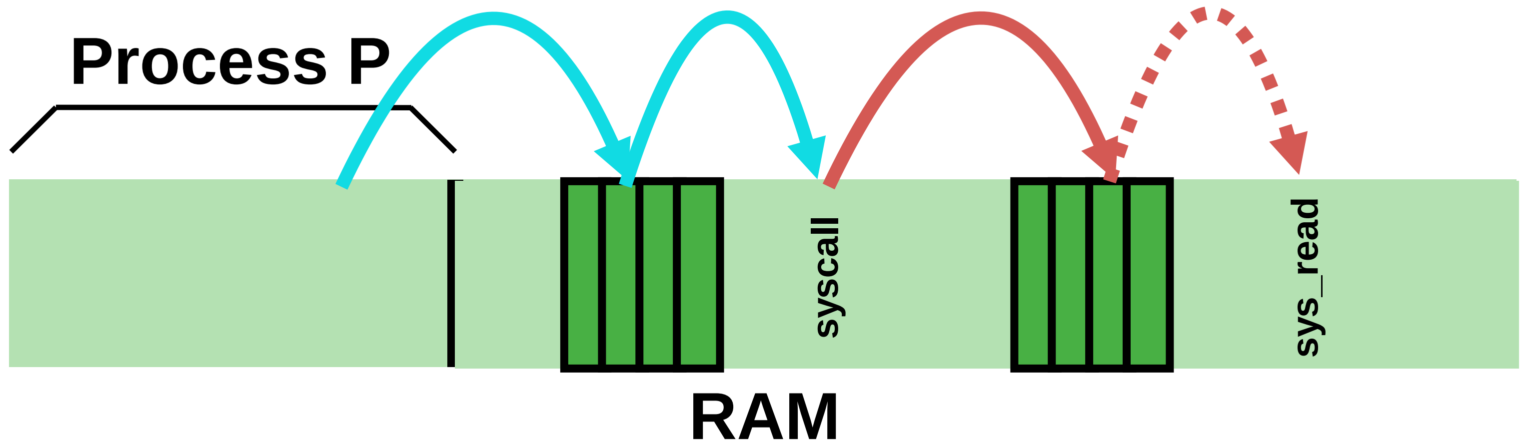

```
movl $6, %eax;    int $64
```

syscall-table index   trap-table index

Trap instruction → kernel mode, vectors to trap handler

Kernel mode: we can do anything!

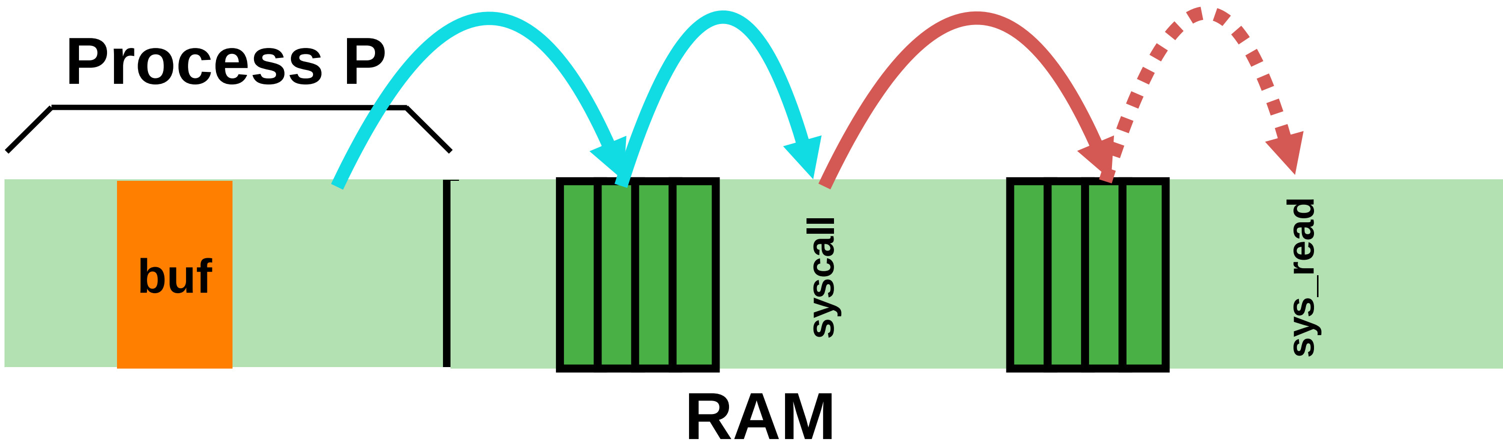
System Call




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

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

What do we need to limit?

User processes are not allowed to perform:

- General memory access
- Disk I/O
- Special x86 instructions like `lidt`

What if process tries to do something restricted?

Problem #2: Take CPU Away?

OS requirements for multiprogramming (multitasking):

- Policy: Decision-maker optimizing a performance metric
  - Process Scheduler: Which process when?
- Mechanism: Low-level code that implements the decision
  - Dispatcher and Context Switch: How?

Example of separation of policy and mechanism

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

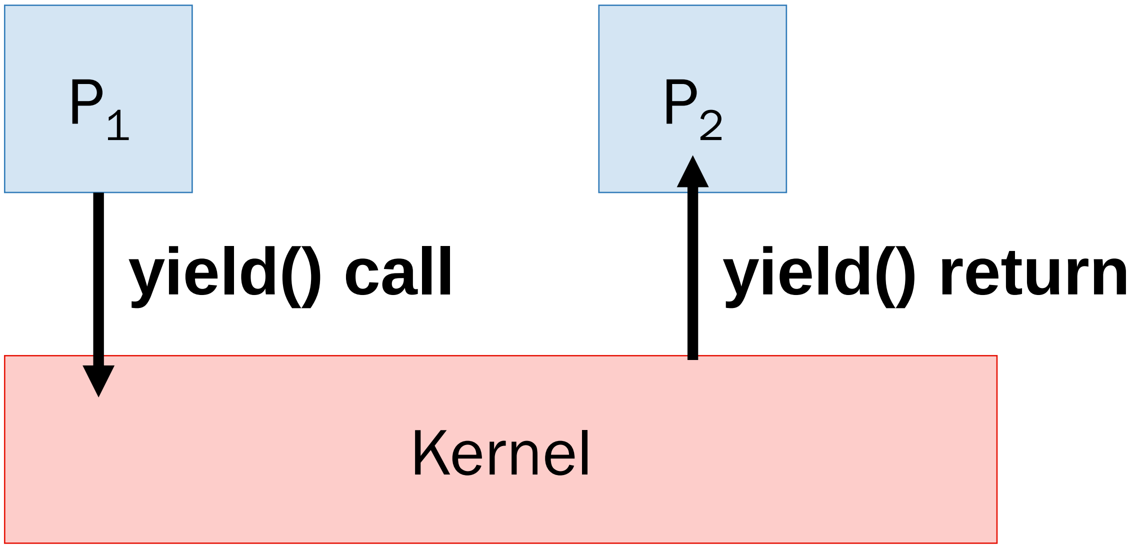
Question 1: How does OS/dispatcher gain control?

Question 2: What execution context must be saved /restored?

Q1: How does OS get control?

Option 1: Cooperative Multitasking

- 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



Q1: How does OS get control?

- Problem with cooperative approach?
- Disadvantages: Processes can misbehave
  - By avoiding all traps and performing no I/O, can take over entire machine
  - Only solution: Reboot!
- Not performed in modern operating systems

Q1: How does OS get control?

Option 2: Preemptive Multitasking

- 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 mask timer interrupt
- Dispatcher counts interrupts between context switches
  - Example: Waiting 20 timer ticks gives 200 ms time slice
  - Common time slices range from 4 ms to a few hundred ms

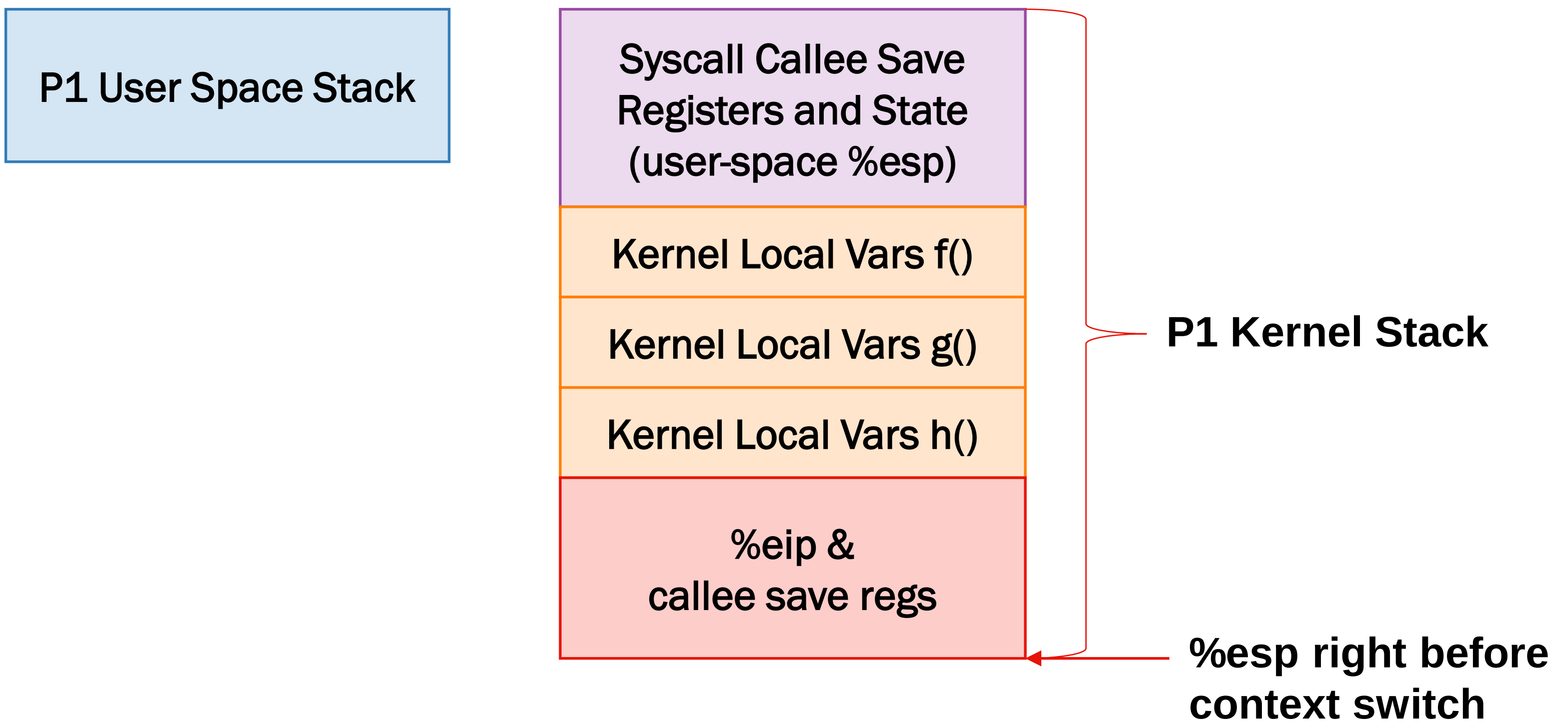
Q2: What context to save?

- Process Control Block: where dispatcher stores context of process when not running; contains
  - PID
  - Process state (i.e., running, ready, or blocked)
  - **Execution state (all registers, instruction ptr, stack ptr)**
  - Scheduling priority
  - Accounting information (parent and child processes)
  - Credentials (which resources can be accessed, owner)
  - Pointers to other allocated resources (e.g., open files)

How this stuff is handled is a bit tricky

- On fork: allocate PCB, initialize, put on ready queue (queue of runnable processes)
- On exit: clean up all process state (close files, release memory, page tables, etc)

Zooming In: Register & KStack



Store this pointer as *context in PCB

Operating System	Hardware	Program
		Process A …
	Syscall or timer interrupt Hw switches to kstack Raises to kernel mode Save regs(A) to kstack(A) Jump to trap handler	
Handle the trap Call **swtch()** routine Save regs(A) to PCB(A) Restore regs(B) from PCB(B) Switch to kstack(B) Return-from-trap (into B)		
	Restore regs(B) from kstack(B) Move to user mode Jump to B's IP	
		Process B …

xv6 PCB

```
struct context
{
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};
```

```
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };
struct proc {
  uint sz;                      // Proc mem size (bytes)
  pde_t* pgdir;                 // Page table
  char* kstack;                 // Bottom of kstack
  enum procstate state;         // Process state
  int pid;                      // Process ID
  struct proc* parent;          // Parent process
  struct trapframe* tf;         // Trap frm for syscall
  struct context* context;      // swtch() here to run
  void* chan;                   // If !0, sleep on chan
  int killed;                   // If !0, been killed
  struct file* ofile[NOFILE];   // Open files
  struct inode* cwd;            // Current directory
  char name[16];                // Process name
};
```

Context Switch

- Context switches are fairly expensive
  - Time sharing systems do 100-1000 context switches per second
  - When? Timer interrupt, packet arrives on network, disk I/O completes, user moves mouse, …

- lab2-15   3.8 μs
- gamow   1.6 μs
- home   1.0 μs

- How might one go about measuring this?

Problem #3: Slow Ops (I/O)?

On op that does not use CPU,
OS switches to other processes

OS must track process states:

Running:

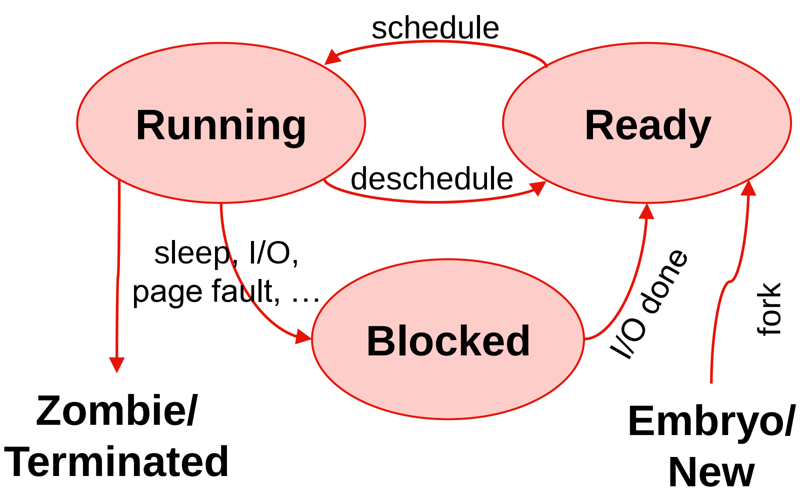
On the CPU (1 on a uniprocessor)

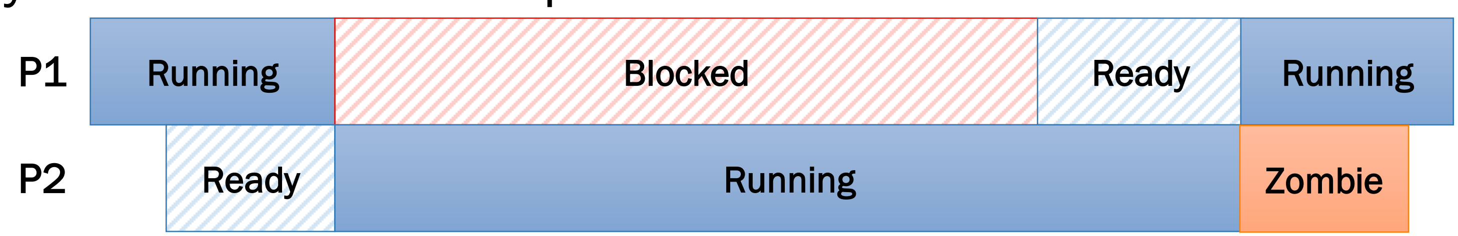
Ready:

Waiting for the CPU

Blocked:

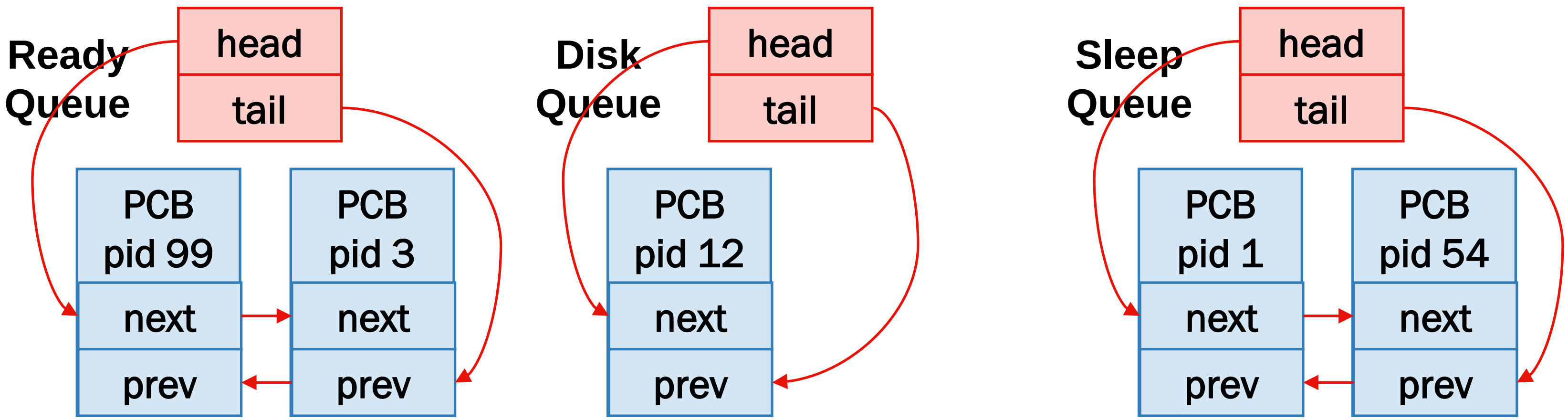
Asleep: Waiting for I/O or synchronization to complete





Problem #3: Slow Ops (I/O)?

- OS maintains queues of all PCBs
  - 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
- Invariant: each process in 1 state and on 1 queue



Summary

- Virtualization:
  Context switching gives each process impression it has its own CPU
- Direct execution makes processes fast
- Limited execution at key points to ensure OS retains control
- Hardware provides a lot of OS support
  - user vs kernel mode
  - timer interrupts
  - automatic register saving

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 call 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

Unix Process Creation

How are Unix shells implemented?

```
while (1) {
  char *cmd = getcmd();
  int retval = fork();
  if (retval == 0) {
        // 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 {
        // Parent process; Wait for child to finish
        int pid = retval;
        wait(pid);
  }
}
```

Next Time

- OSTEP Chapters 7, 8, 9

Important Terms and Ideas

- Process, programs
- Address space
- Process Control Blocks
- Process State Machine
- New, Ready, Running, Blocked, Terminated
- fork(), wait(), exec()