VIRTUALIZATION: THE CPU

Andrea Arpaci-Dusseau

CS 537, Fall 2019

ADMINISTRIVIA

- Project 1 is out! Due Monday, 9/16 before midnight
  - Solo, but later ones will involve project partners
  - Handin directories and test cases available
- Discussion sections on Wednesday
  - Can attend others if room (last two have most space)
- Sign up for Piazza
- Lecture recordings
- Still on waitlist? Sign attendance sheet at end of lecture
- Lecture ends at 12:15pm
- Microphone: Let me know if you can't hear

AGENDA / OUTCOMES

Abstraction for CPU

- What is a Process? What is its lifecycle?

Mechanism

- How does a process interact with the OS?
- How does the OS switch between processes?

REVIEW

What is an Operating System?

- Software that converts hardware into a useful form for applications

What abstraction does the OS provide for the CPU?

  - Process or thread

For memory?

  - Virtual address space

What are some advantages of OS providing resource management?

- Protect applications from one another
- Provide fair and efficient access to resources (cost, time, energy)

VIRTUALIZING THE CPU

High-level Goal:

Give each "process" impression it alone is actively using CPU

Resources can be shared in **time** and/or **space**

Assume single uniprocessor

- Time-sharing (multi-processors: advanced issue with space-sharing)

Memory?

- Space-sharing (later)

Disk?

- Space-sharing (later)

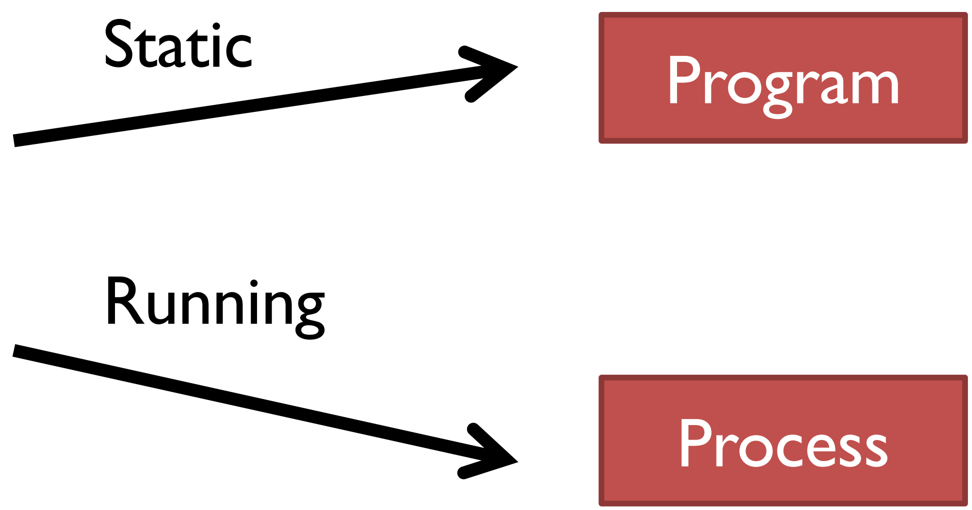
ABSTRACTION: PROCESS

PROGRAM VS PROCESS

```c
#include <stdio.h>
#include <stdlib.h>
#include "common.h"

int main(int argc, char *argv[]) {
    char *str = argv[1];

    while (1) {
      printf("%s\n", str);
      Spin(1);
    }
    return 0;
}
```



WHAT IS A PROCESS?

Stream of executing instructions and their "context" in address space

Instruction Pointer

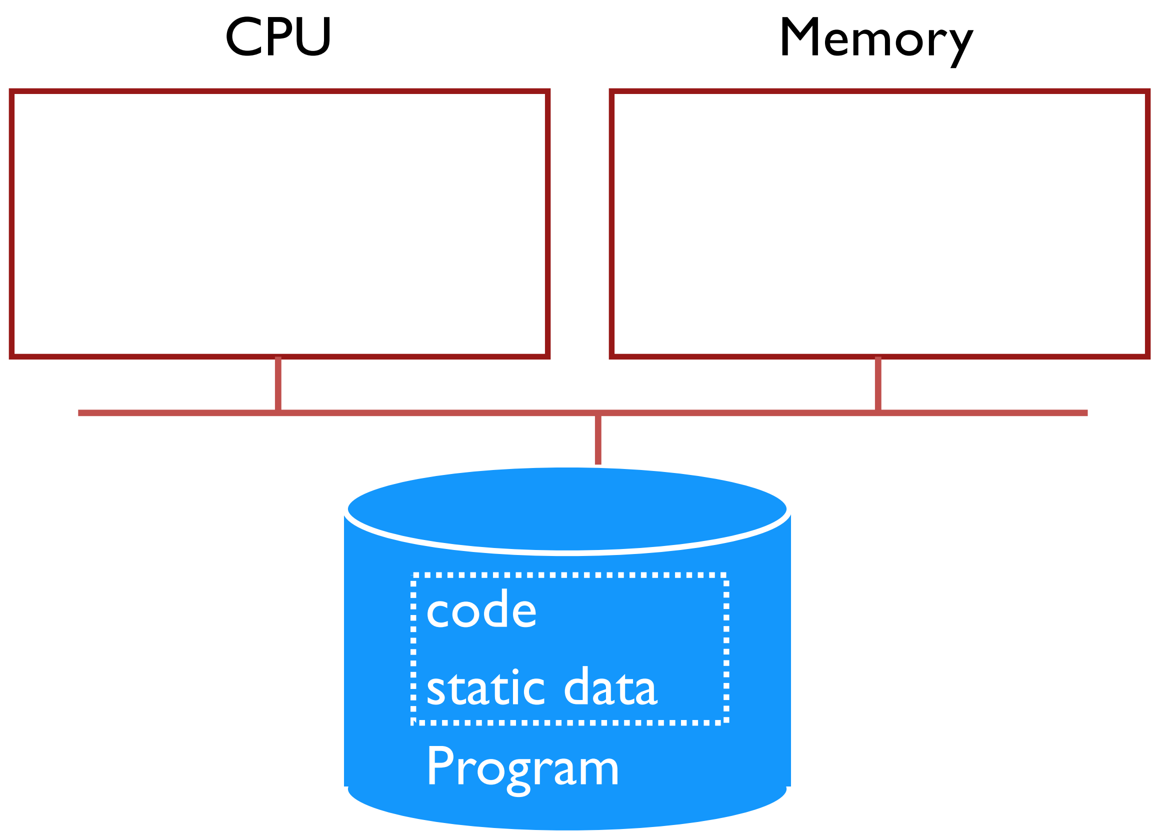
```
pushq   %rbp
movq    %rsp, %rbp
subq    $32, %rsp
movl    $0, -4(%rbp)
movl    %edi, -8(%rbp)
movq    %rsi, -16(%rbp)
cmpl    $2, -8(%rbp)
je      LBB0_2
```

Registers
Memory addrs

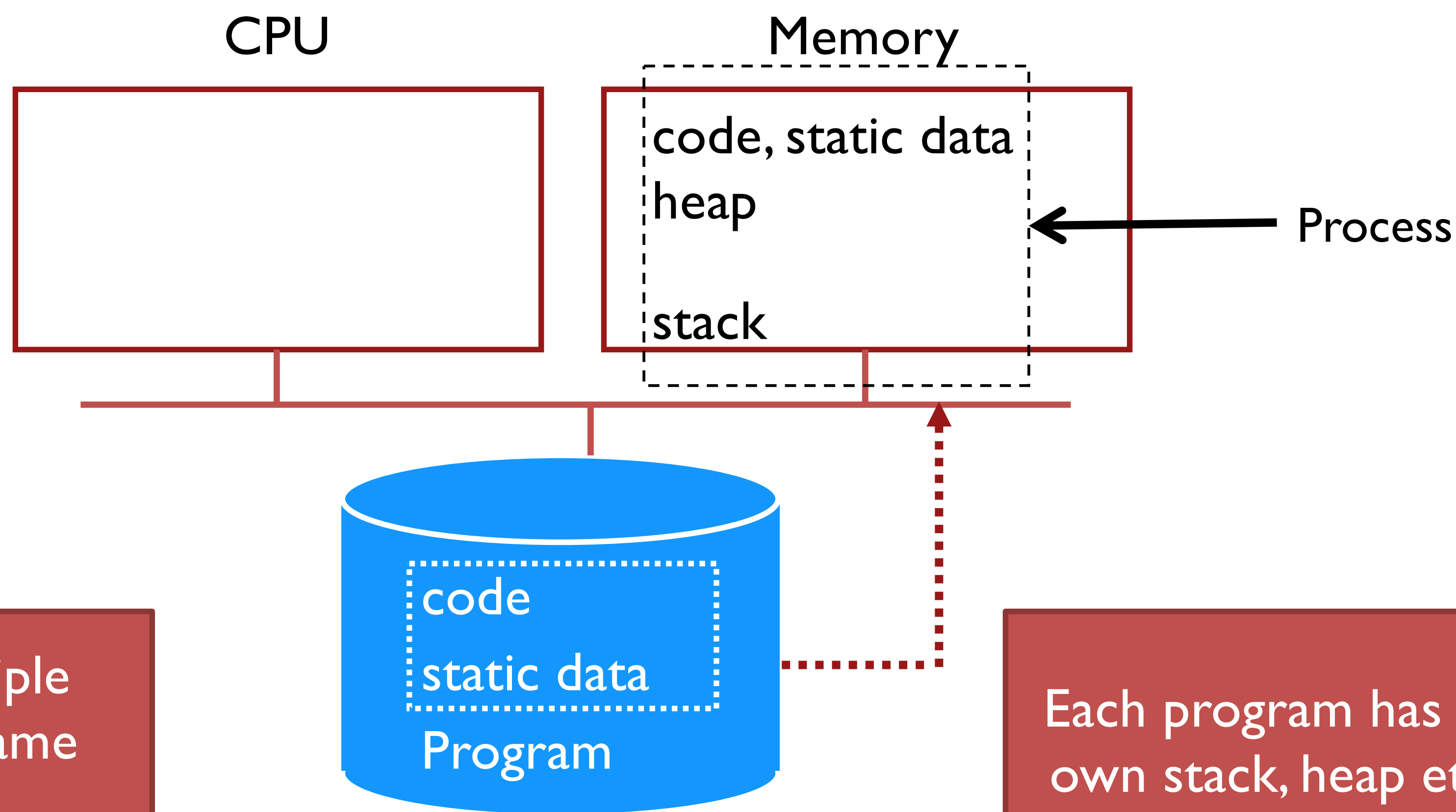
Stack pointer

File descriptors

PROCESS CREATION



PROCESS CREATION



Can run multiple instances of same program

Each program has its own stack, heap etc.

PROCESS VS THREAD DEMO

- Two **processes** examining same memory address see **different** values (I.e., different contents)
  - Different isolated address spaces

- Two **threads** examining memory address see **same** value (I.e., same contents)
  - Share same address space

PROCESS VS THREAD

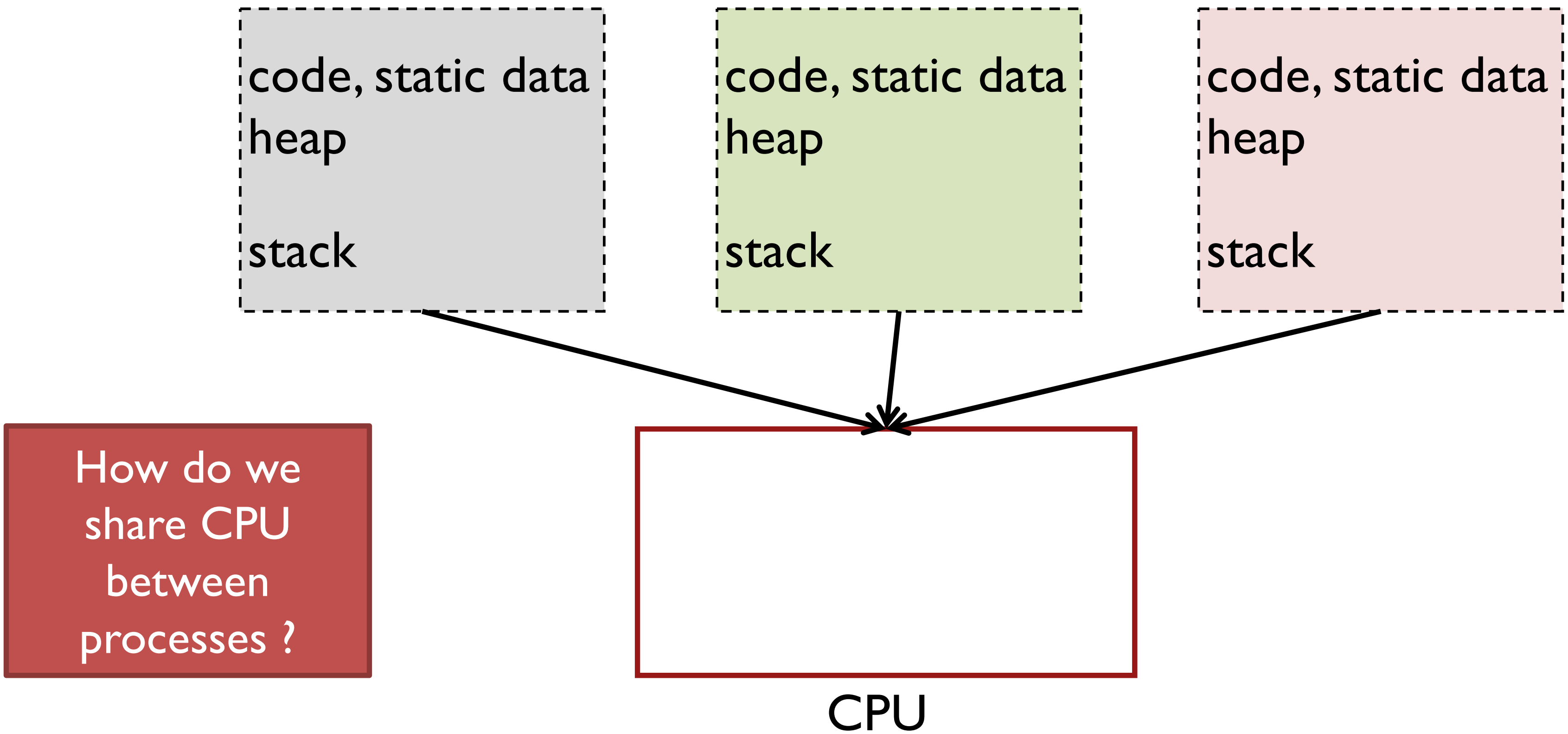
Threads: "Lightweight process"

Execution streams that share an address space
Can directly read / write same memory

Can have multiple threads within a single process

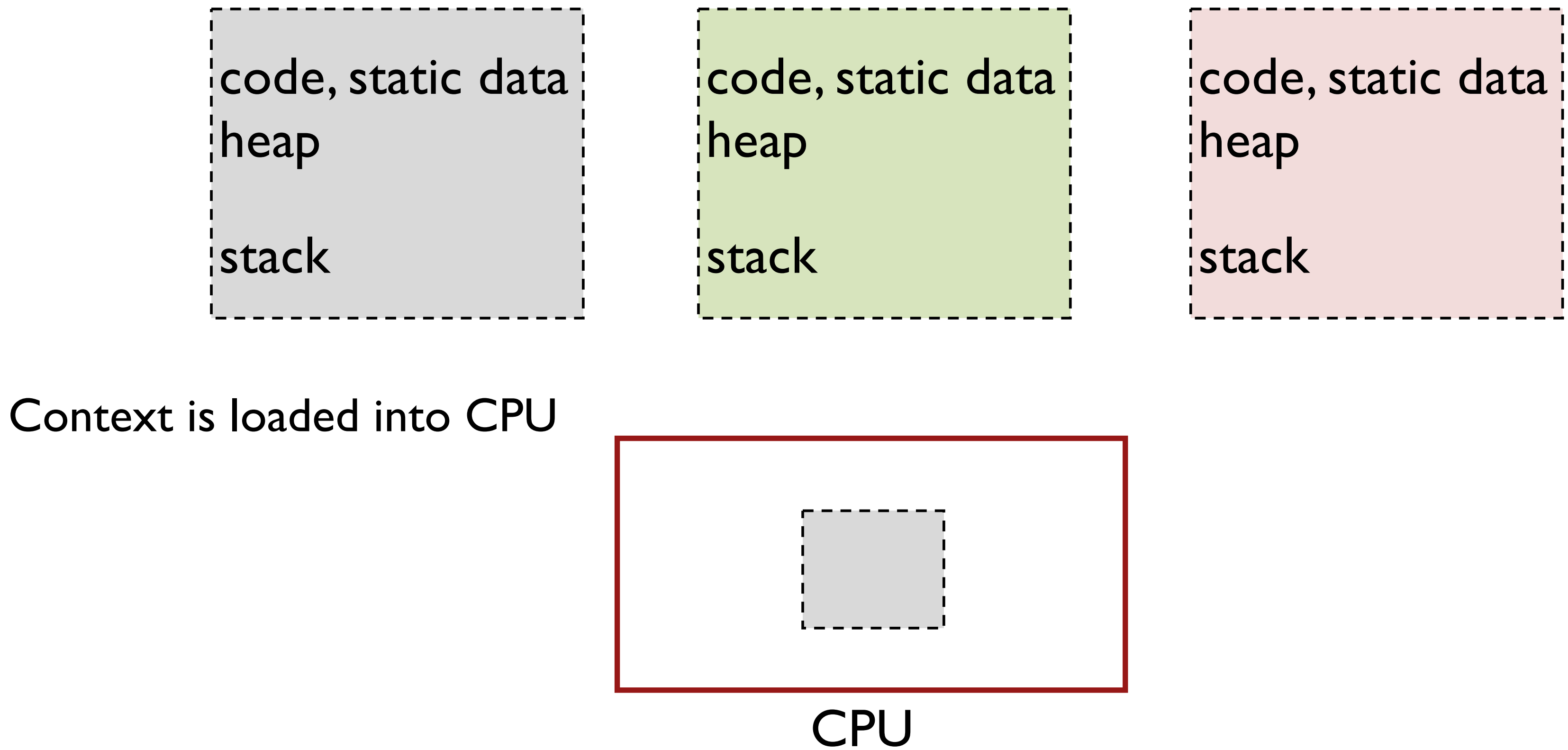
WHY DO WE NEED PROCESSES ?

SHARING CPU

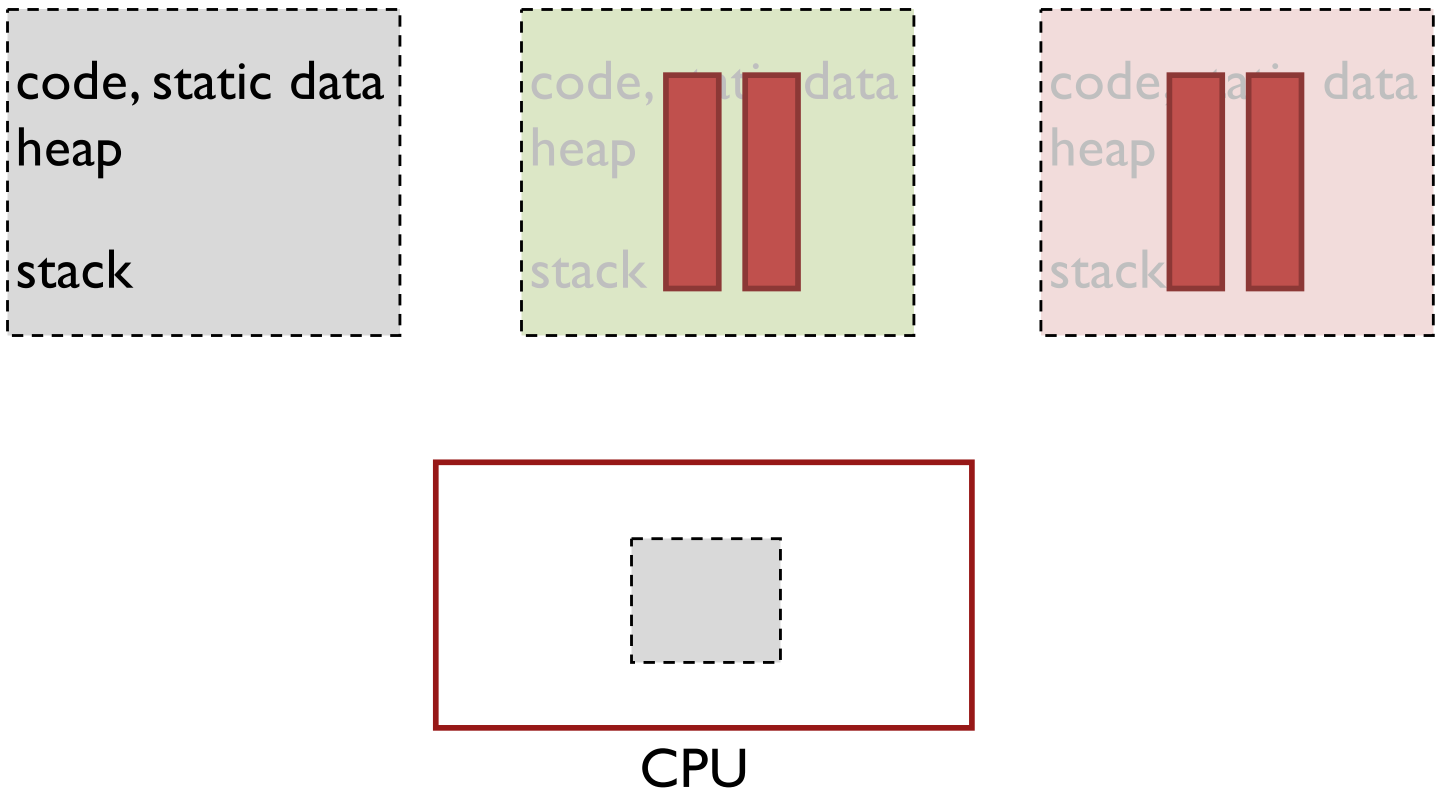


code, static data
heap
stack
code, static data
heap
stack
code, static data
heap
stack
How do we share CPU between processes ?
CPU

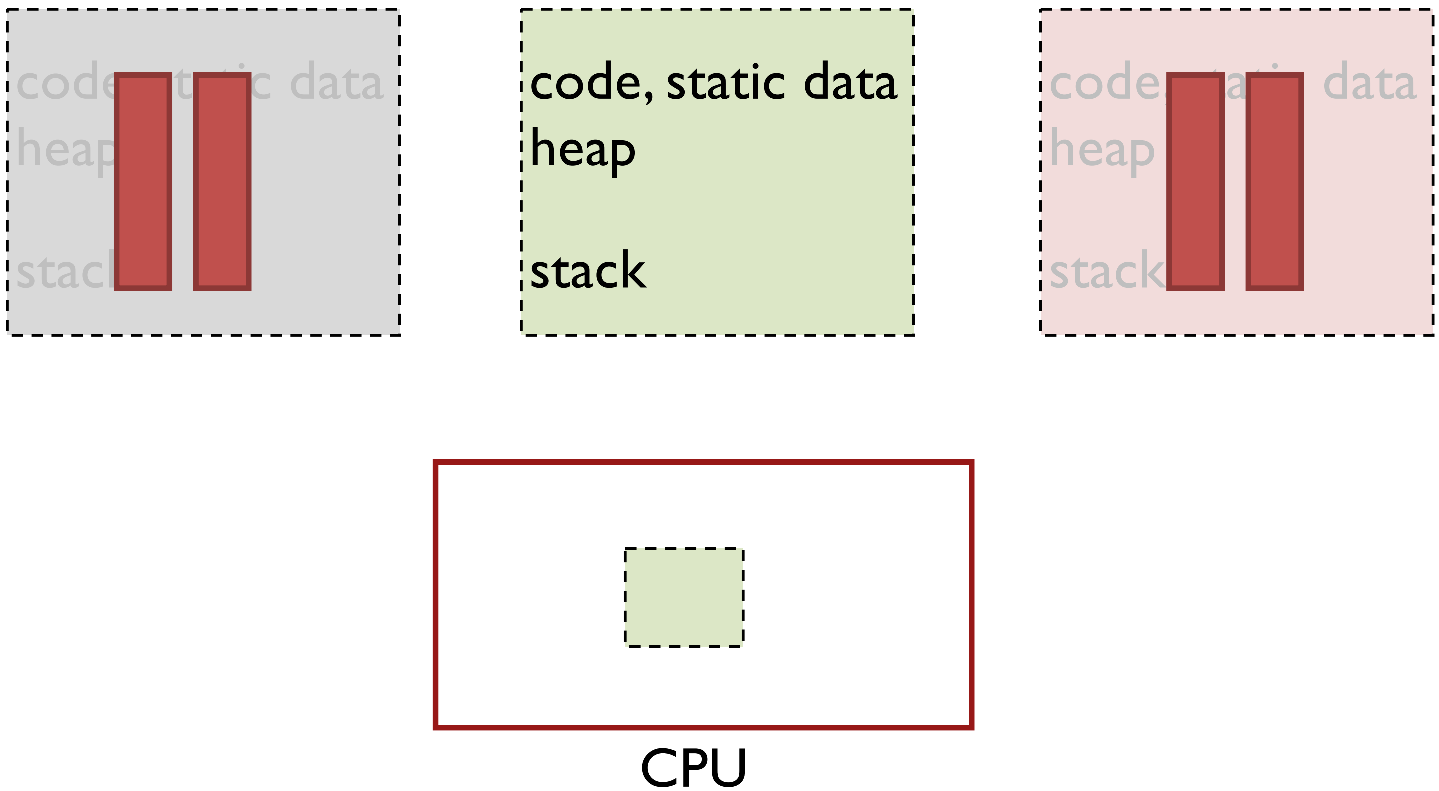
TIME SHARING



TIME SHARING



TIME SHARING

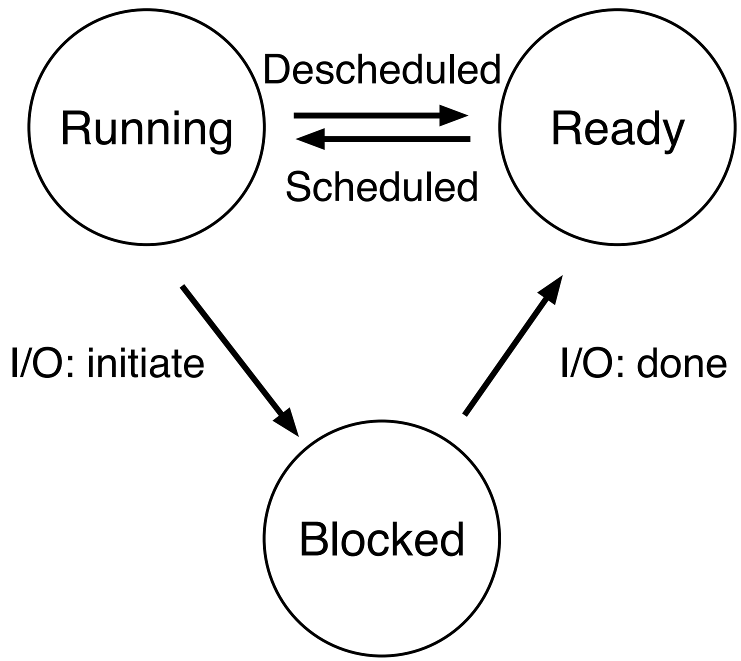


WHAT TO DO WITH PROCESSES THAT ARE NOT RUNNING ?

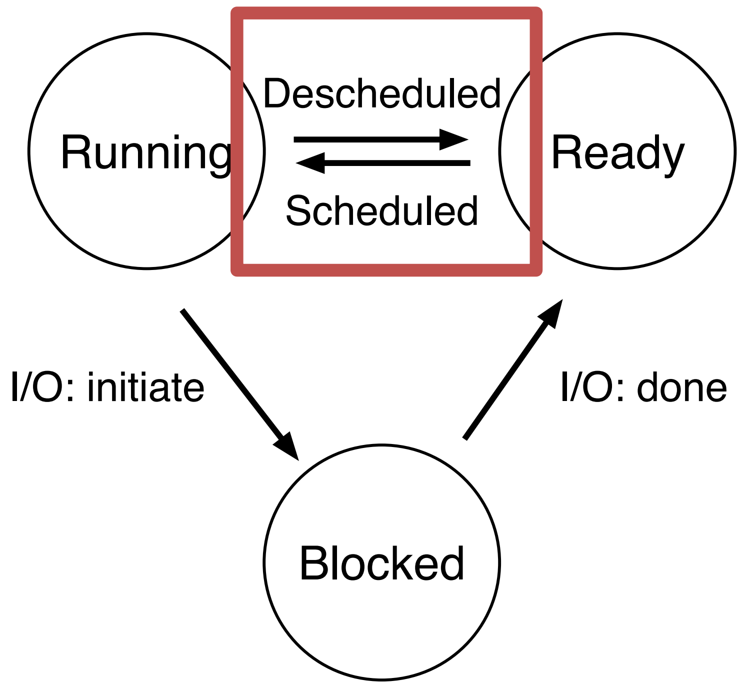
OS Scheduler

- Save context when process is paused
- Restore context on resumption

STATE TRANSITIONS



STATE TRANSITIONS



ASIDE: OSTEP HOMEWORKS!

- Optional homeworks corresponding to each chapter in book
- Little simulators to help you understand
- Can generate problems and solutions!

http://pages.cs.wisc.edu/~remzi/OSTEP/Homework/homework.html

PROCESS HW

Run ./process_run.py –l 2:100,2:0

QUIZ

```
≥ ./process-run.py -l 3:50,3:40
Process 0
  io
  io
  cpu

Process 1
  cpu
  io
  io
```

CPU TIME SHARING

Mechanism goals: Be able to run processes

  Efficiency: Time sharing should not add overhead

  Control: OS should be able to intervene when required

Policy goals: Pick the "best" process to schedule

  Reschedule process for fairness? efficiency ?

Separate mechanism from policy for clean OS design

How to have efficient mechanism??

EFFICIENT EXECUTION

Simple answer !?: Direct Execution

Allow user process to run directly

Create process and transfer control to main()

Challenges

1) What if the process wants to do something restricted ? Access disk ?

2) What if the process runs forever ? Buggy ? Malicious ?

Solution: Limited Direct Execution (LDE)

CHALLENGE 1: RESTRICTED OPS

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

Solution: privilege levels supported by hardware (bit of status)

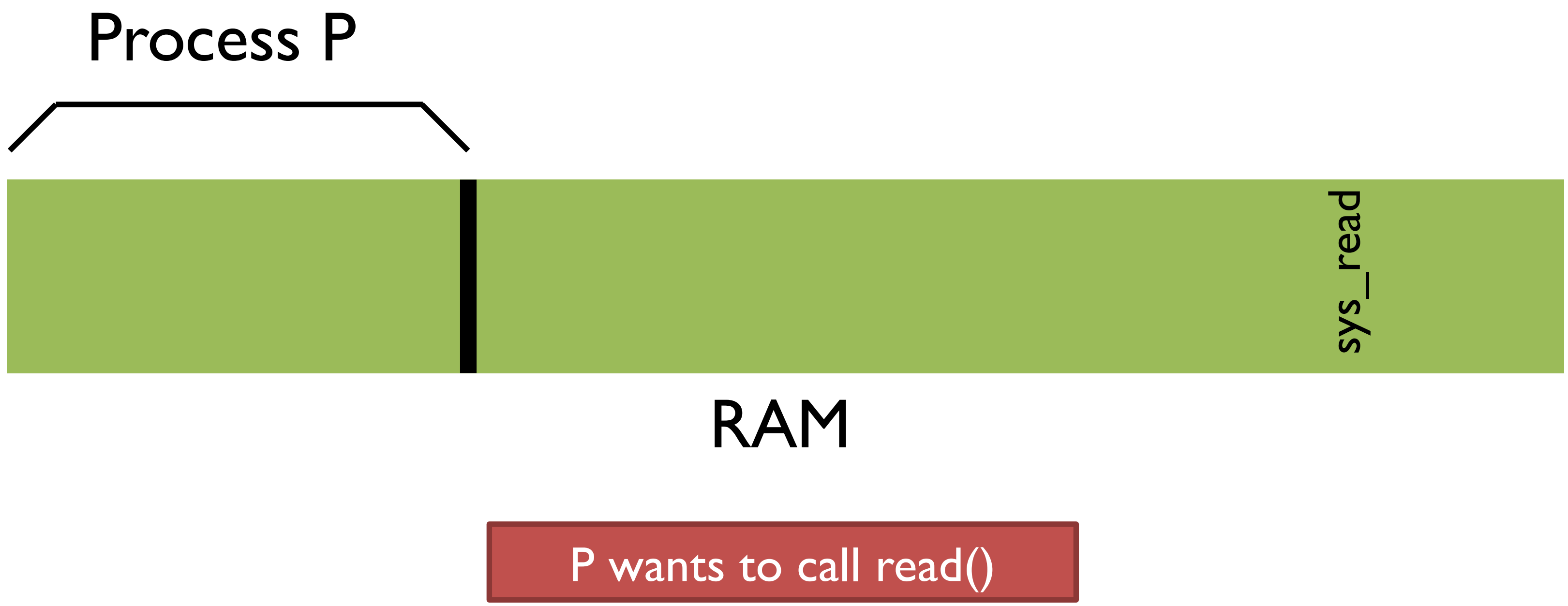
- User processes run in user mode (restricted mode)
- OS runs in kernel mode (not restricted)

How can process access devices?

- System calls (function call implemented by OS)

SYSTEM CALL

SYSTEM CALL



SYSTEM CALL

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

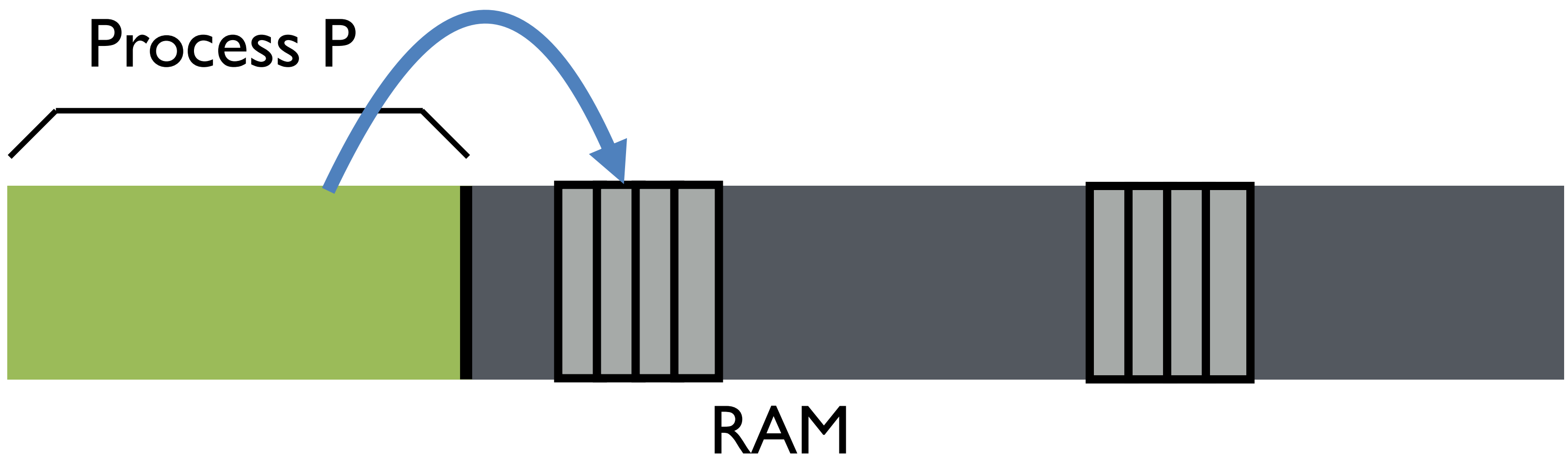
SYSTEM CALL

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

SYSTEM CALL

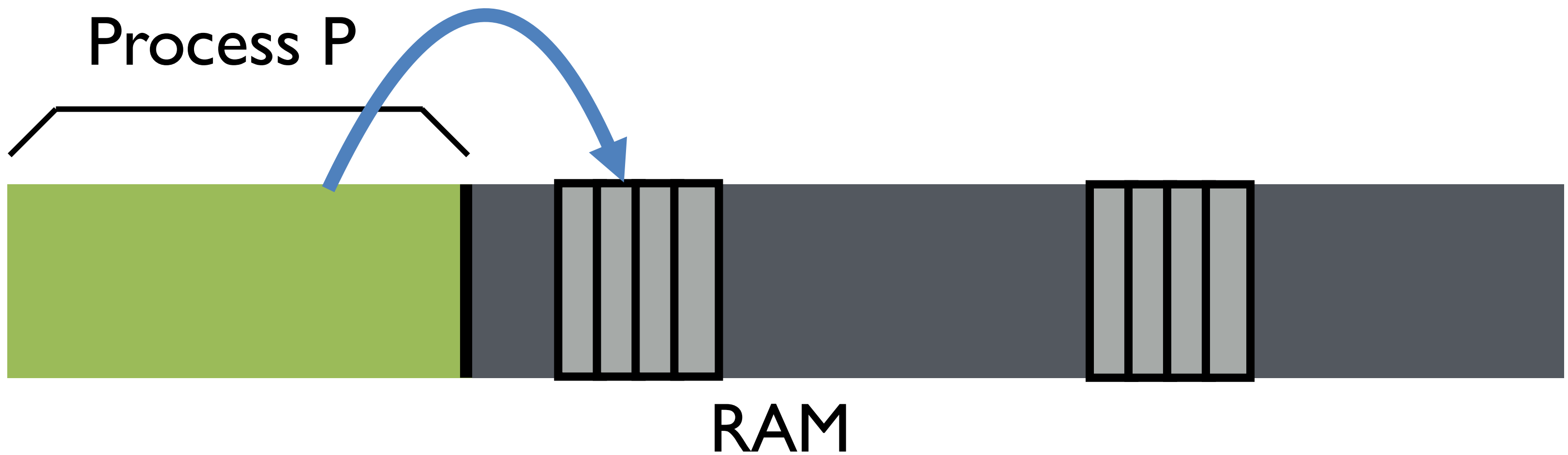

```
movl $6, %eax;   int $64
```

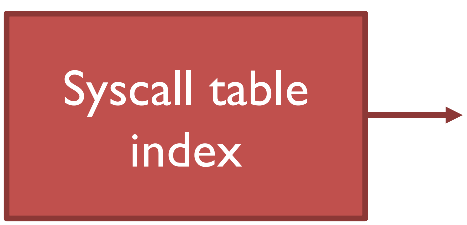
SYSTEM CALL




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

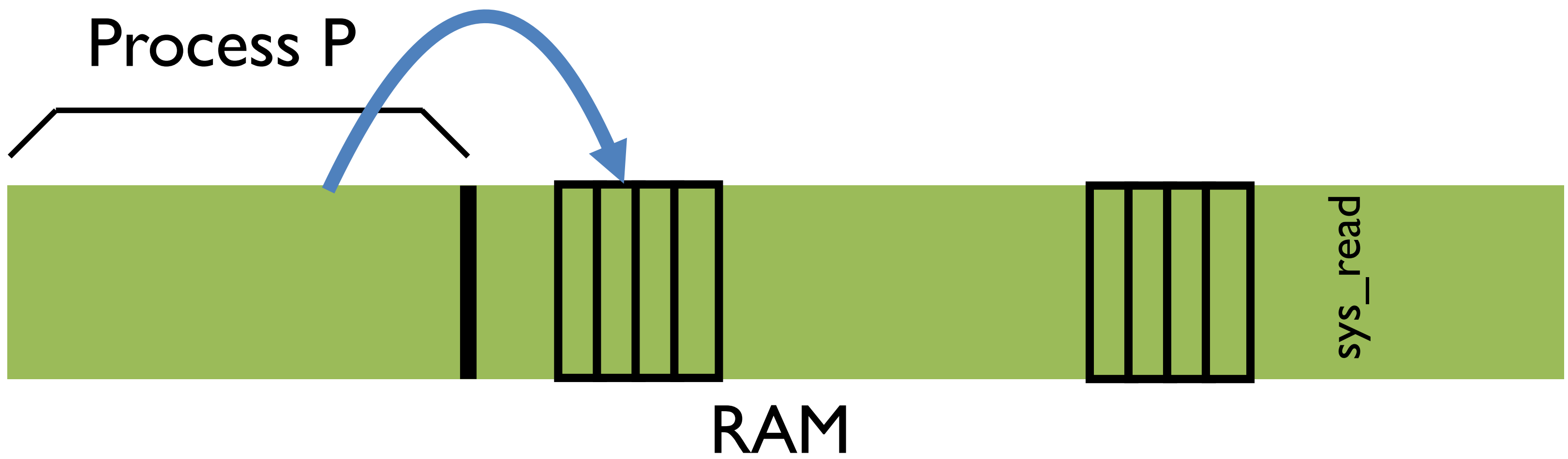
SYSTEM CALL





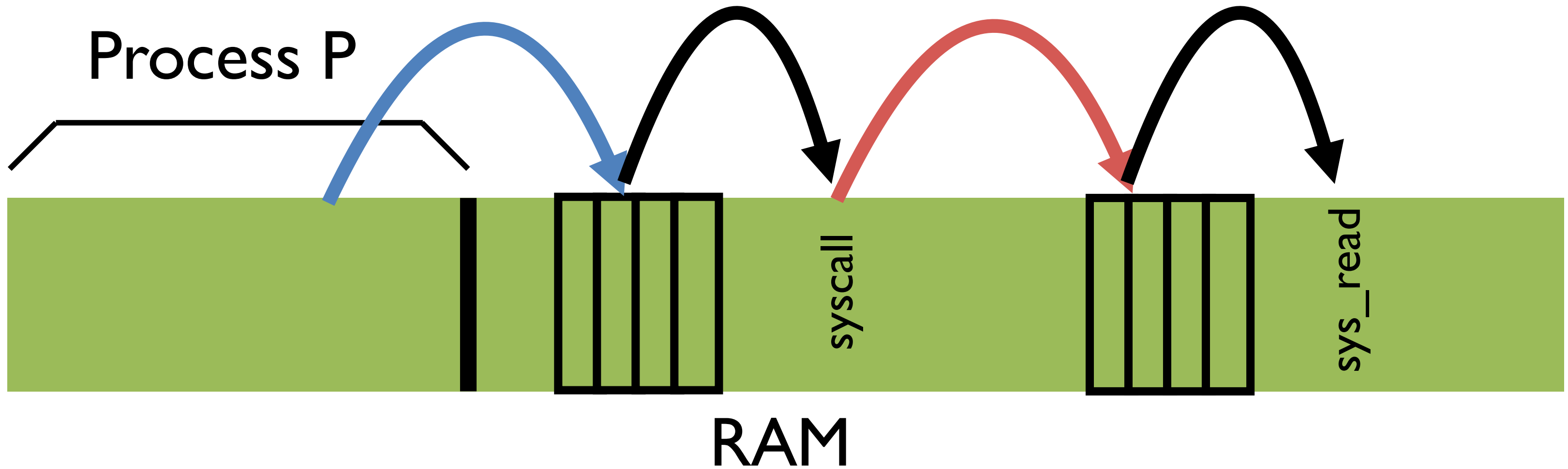

```
movl $6, %eax;    int $64
```

SYSTEM CALL




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

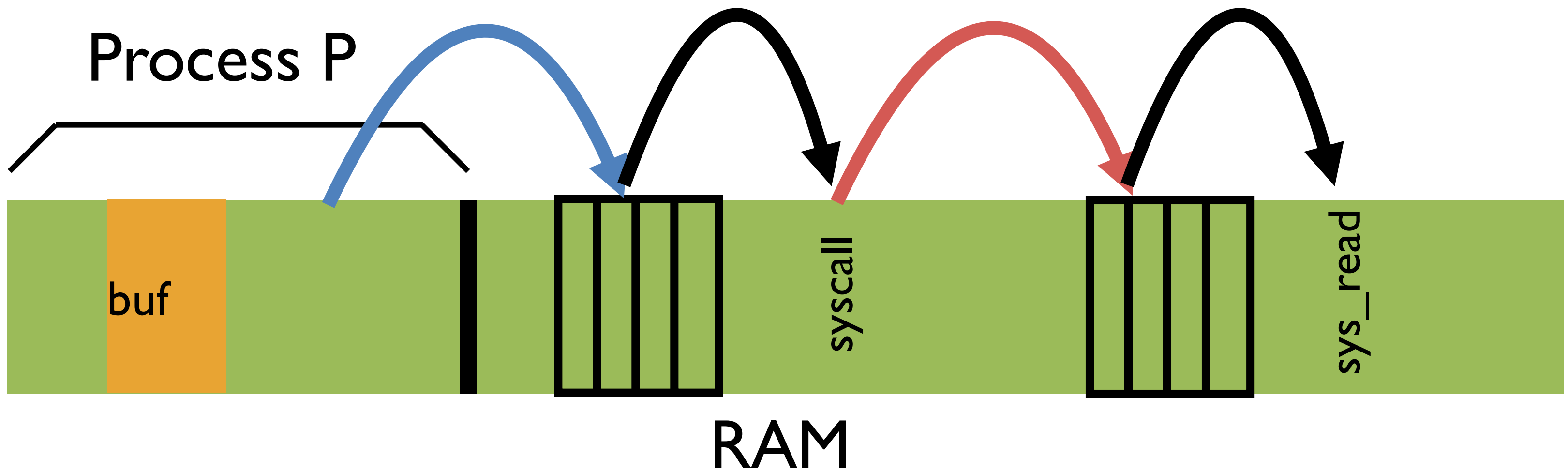
SYSTEM CALL




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

Follow entries to correct system call code

SYSTEM CALL



Kernel can access user memory to fill in user buffer
return-from-trap at end to return to Process P

SYSCALL SUMMMARY

Separate user-mode from kernel mode for security

Syscall: call kernel mode functions

- Transfer from user-mode to kernel-mode (trap)
- Return from kernel-mode to user-mode (return-from-trap)

5 MINUTE BREAK!

Talk with at least two neighbors

- What has been your favorite course in CS? What did you like most about it?
- Favorite course outside of CS? Why?

REPEAT: EFFICIENT EXECUTION

Simple answer !?: Direct Execution

Allow user process to run directly

Create process and transfer control to main()

Challenges

1) What if the process wants to do something restricted ? Access disk ?

2) What if the process runs forever ? Buggy ? Malicious ?

Solution: Limited Direct Execution (LDE)

CHALLENGE 2: HOW TO TAKE CPU AWAY

Policy

- To decide which process to schedule when
- Decision-maker to optimize some workload performance metric

Mechanism

- To switch between processes
- Low-level code that implements the decision

Separation of policy and mechanism: Recurring theme in OS

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

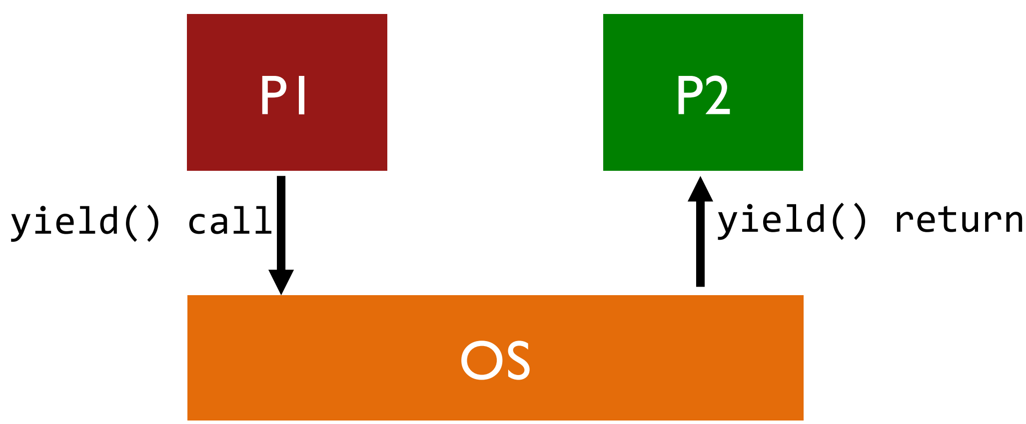
Question 1: How does dispatcher gain control?

Question 2: What must be saved and restored?

HOW DOES DISPATCHER GET CONTROL?

Option 1: Cooperative Multi-tasking: Trust process to relinquish CPU 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



PROBLEMS WITH COOPERATIVE ?

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

TIMER-BASED INTERRUPTS

Option 2: Timer-based Multi-tasking (True multi-tasking)

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

Operating System | Hardware | Program

Process A

Operating System	Hardware	Program
		Process A
	timer interrupt save regs(A) to k-stack(A) move to kernel mode jump to trap handler for timer	

Operating System | Hardware | Program

Process A

timer interrupt
save regs(A) to k-stack(A)
move to kernel mode
jump to trap handler for timer

Handle the trap for timer
Call switch() routine
 save regs(A) to proc-struct(A)
 restore regs(B) from proc-struct(B)
 switch to k-stack(B)
 return-from-trap (into B)

Operating System	Hardware	Program
		Process A
	timer interrupt save regs(A) to k-stack(A) move to kernel mode jump to trap handler for timer	
Handle the trap for timer Call switch() routine save regs(A) to proc-struct(A) restore regs(B) from proc-struct(B) switch to k-stack(B) return-from-trap (into B)		
	restore regs(B) from k-stack(B) move to user mode jump to B's IP	

Operating System	Hardware	Program
		Process A
	timer interrupt save regs(A) to k-stack(A) move to kernel mode jump to trap handler for timer	
Handle the trap for timer Call switch() routine save regs(A) to proc-struct(A) restore regs(B) from proc-struct(B) switch to k-stack(B) return-from-trap (into B)		
	restore regs(B) from k-stack(B) move to user mode jump to B's IP	
		Process B

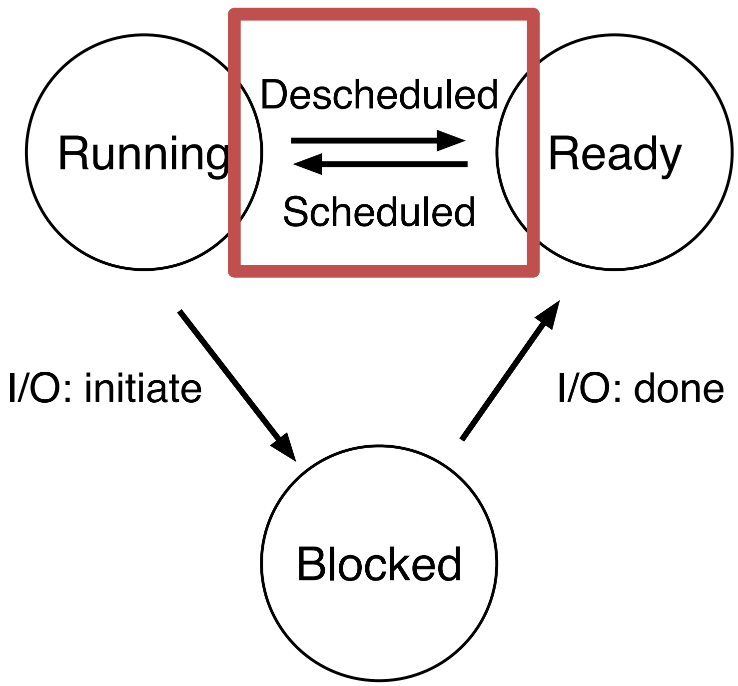
SUMMARY

Process: Abstraction to virtualize CPU

Use time-sharing in OS to switch between processes

Key aspects

- Use system calls to run access devices etc. from user mode
- Context-switch using interrupts for multi-tasking



POLICY ?

NEXT CLASS!

ADMINISTRIVIA

- Lecture ends at 12:15pm
- Project 1 is out! Due Monday, 9/16 before midnight
  - Handin directories and test cases available
- Discussion sections on Wednesday
  - Can attend others if room (last two have most space)
- Sign up for Piazza
- Still on waitlist? Sign attendance sheet at end of lecture