Operating Systems

Processes

ENCE 360

Outline

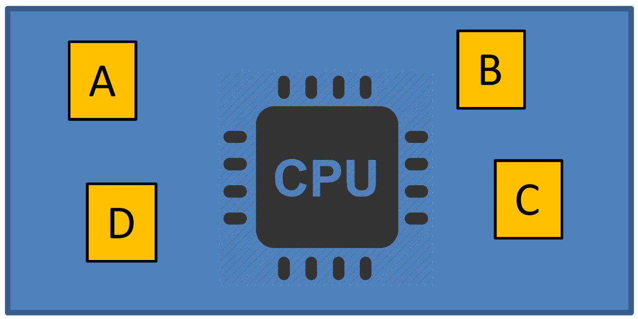
- Motivation
- Control block
- Switching
- Control

Chapter 2
MODERN OPERATING SYSTEMS (MOS)
By Andrew Tanenbaum

Chapter 4
OPERATING SYSTEMS: THREE EASY PIECES
By Arpaci-Dusseau and Arpaci-Dusseau

The Problem

- Remember “CPU” program from day 1?
  - Each ran as if was *only* program on computer

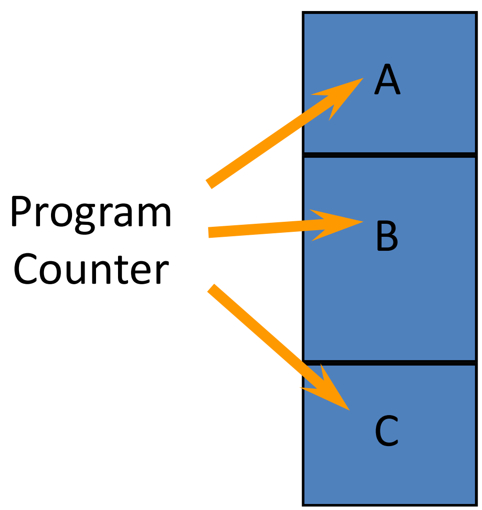


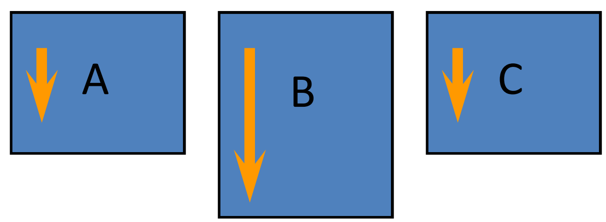
THE CRUX OF THE PROBLEM:
HOW TO PROVIDE ILLUSION OF MANY CPUS?

Few physical CPUs available, so how can OS provide illusion of nearly-endless supply of said CPUs?

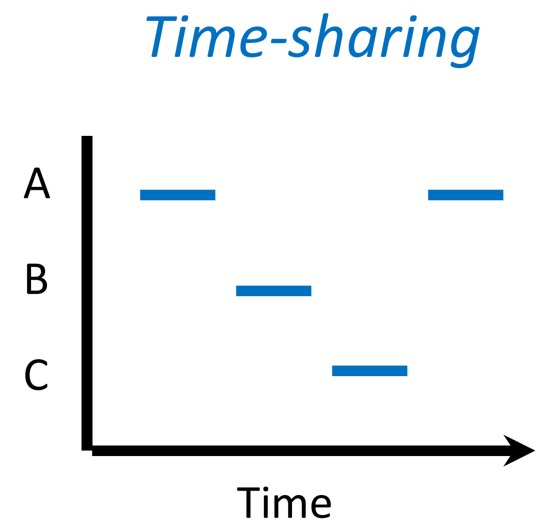
The Solution – The Process

- "A program in execution"
- Running several at once provides pseudo-parallelism





Conceptual View



- Low-level machinery (mechanisms)

  Answer question of *how*. E.g., how to keep program context
- High-level intelligence (policies)

  Answer question of *which*. E.g., which process to run

Note: good design to separate!

Process States

- Consider the shell command:

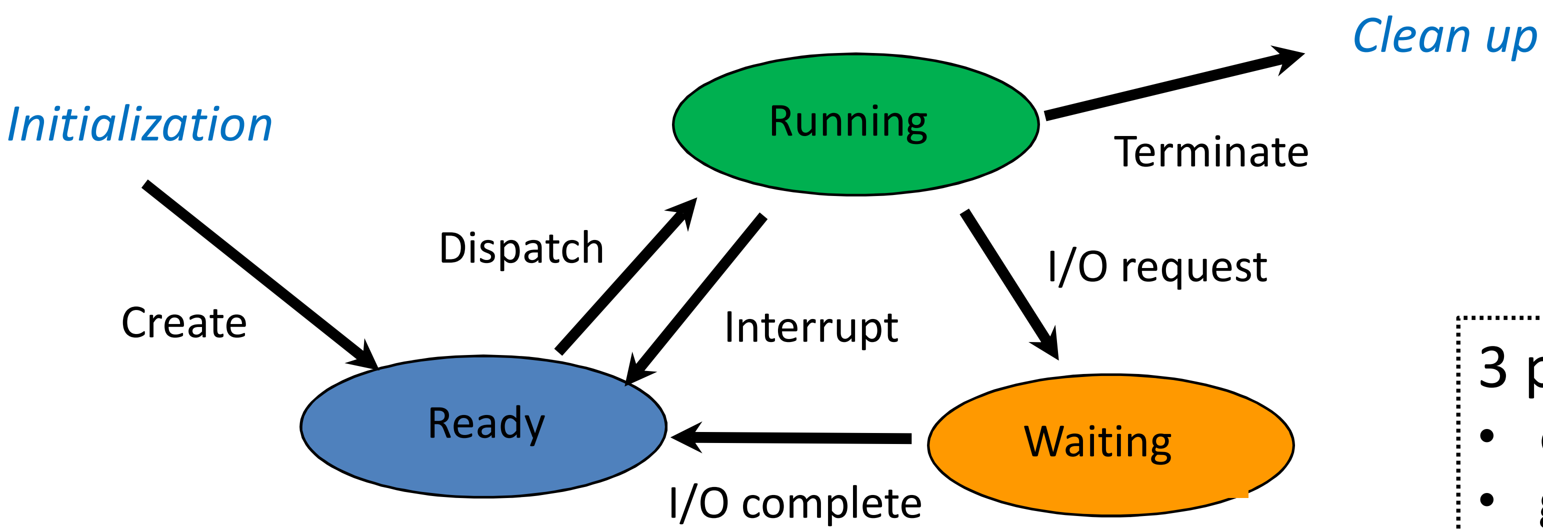
  `cat /etc/passwd | grep claypool`

1. What is this command doing?
2. How many processes are involved?

Process States

- Consider the shell command:

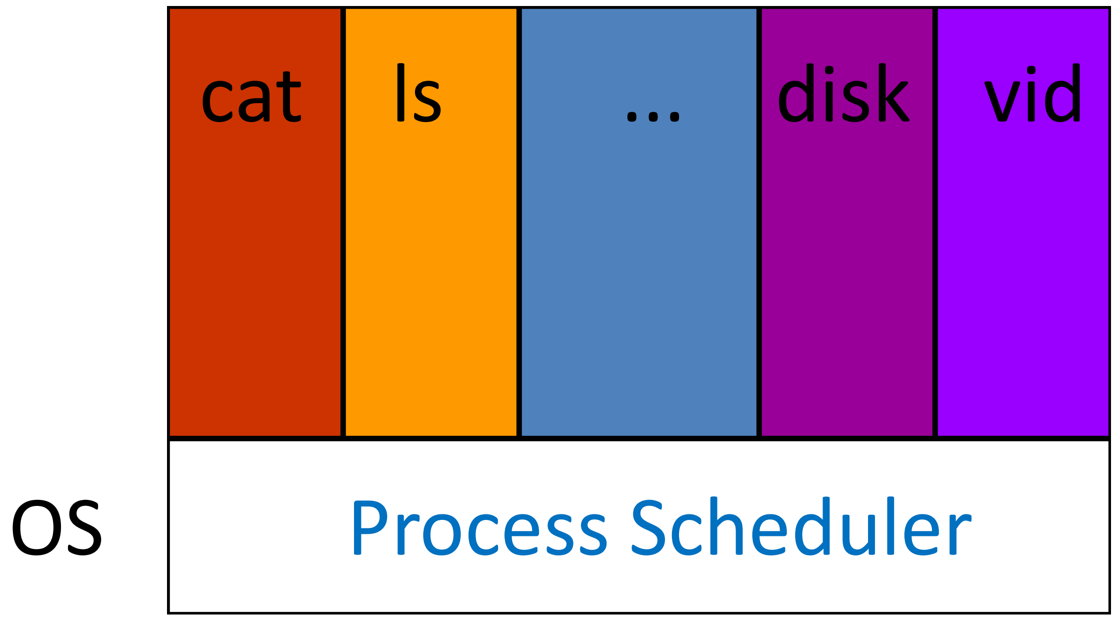
```
cat /etc/passwd | grep claypool
```



3 processes
- `cat`
- `grep`
- `bash`

(See process states with `top`)

OS as a Process Scheduler

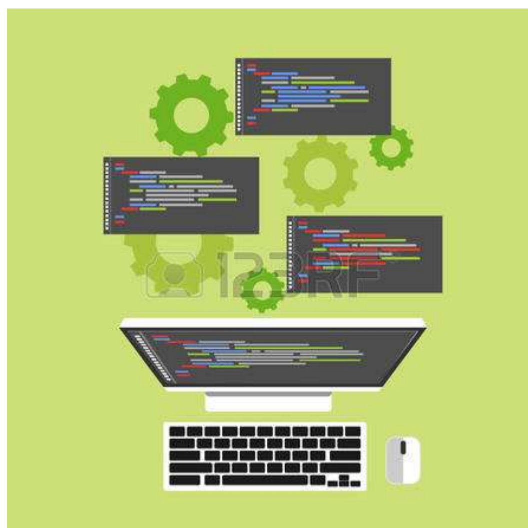


- Simple OS view – just schedule processes! Even OS services (e.g., file system) are just processes
- Small scheduler handles interrupts, stopping and starting processes (policy decides when)
- Ok, what are mechanisms needed to make this happen?

Program → Process

```
int g_x
main() {
...
}
A() {
  f = open()
  ...
}
```

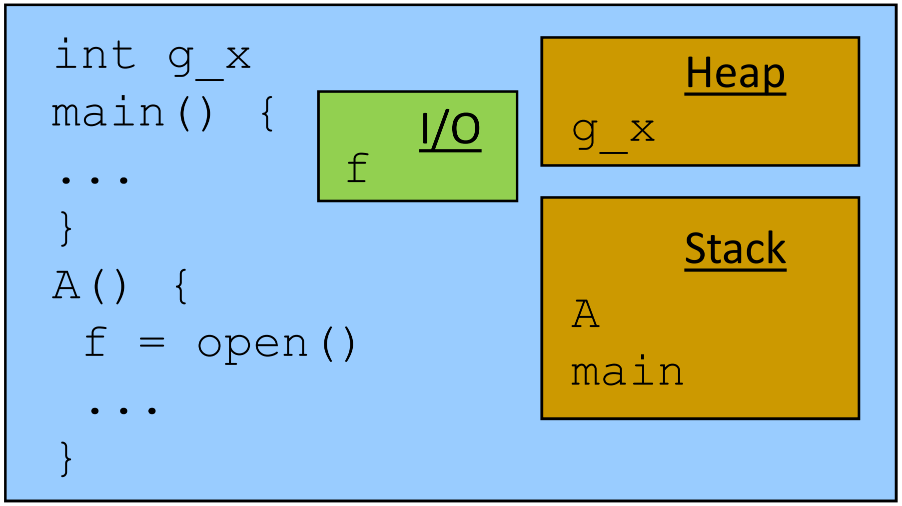
- What information do we need to keep track of a process (i.e., a running program)?



?

Program → Process

```
int g_x
main() {
...
}
A() {
  f = open()
  ...
}
```



- Low-level machinery (*mechanisms*) – to store program context
  - (Discuss policies later in scheduling)
  - Current execution location
  - Intermediate computations (heap and stack)
  - Access to resources (e.g., I/O and files open)

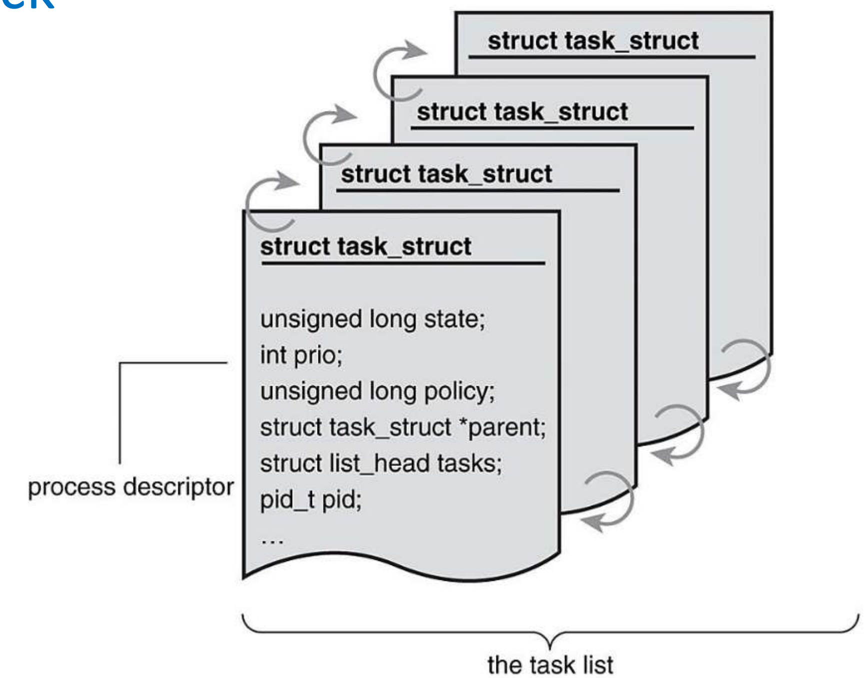
Process Control Block (PCB)

Outline

- Motivation (done)
- Control block (next)
- Switching
- Control

Process Control Block

- OS keeps one Process Control Block (PCB) for each process
  - process state
  - program counter
  - registers
  - memory management
  - open devices
  - …
- OS keeps list/table of PCB's for all processes (use when scheduling)
- Code examples:
  - SOS "pcb.h": `ProcessControlBlock`
  - Xv6 "proc.h": `proc`
  - Linux "sched.h": `task_struct`



Process Control Block – Summary Info

Process management	Memory management	File management
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

List of typical attributes in PCB

Outline

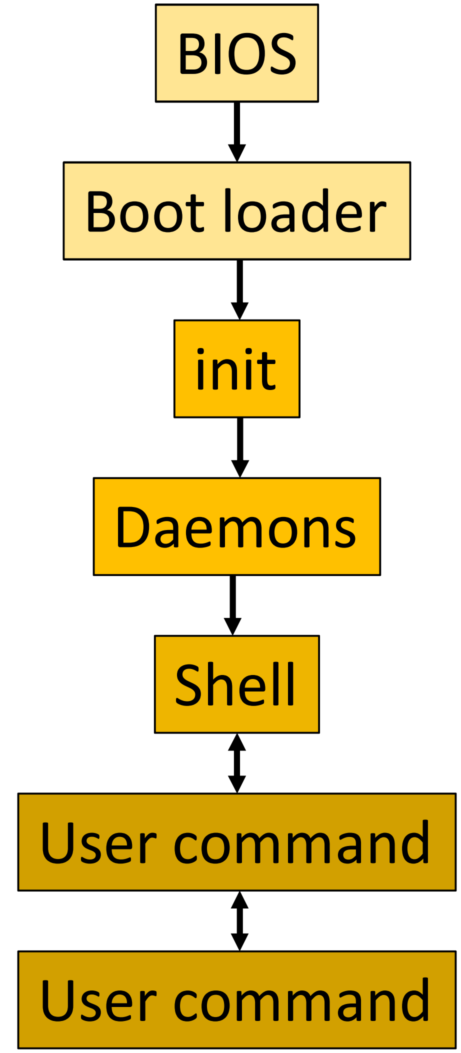
- Motivation (done)
- Control block (done)
- Switching (next)
- Control

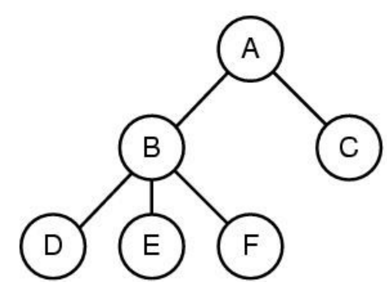
Process Creation

- When are processes created?

Process Creation

- System initialization
  - When OS boots, variety of system processes created
  - `init` – parent of all processes (pid 1)
  - Background, don't need to interact with user (*daemons* for "guiding spirit")
    - Note, foreground processes get input from user
- Created on demand by user
  - Shell command or, e.g., double clicking icon
- Execution of system call
  - Process itself may create other processes to complete task
- Created by batch job
  - Queued awaiting necessary resources. When available, create process(es)



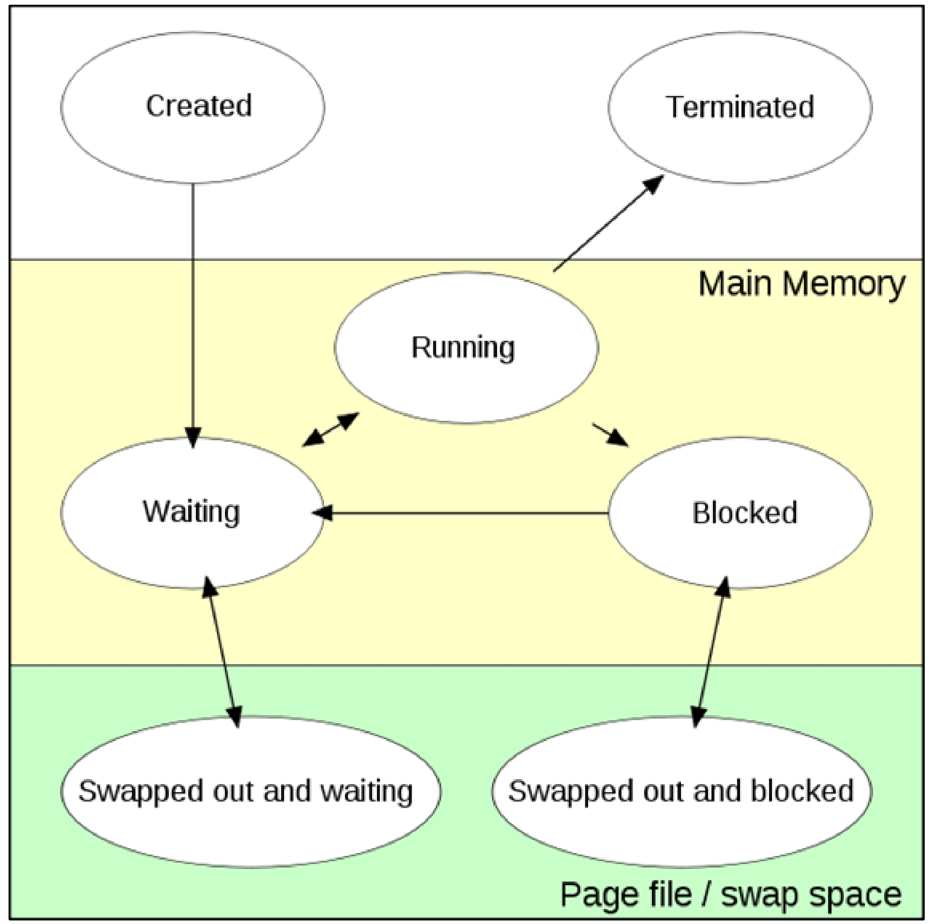


Process Termination

- When are processes terminated?

Process Termination

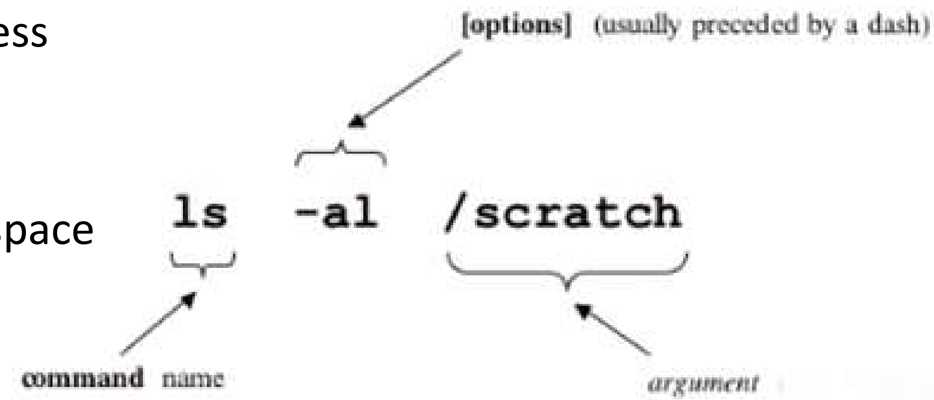
- Voluntarily
  - Make system call to `exit()` or `return` from `main()`
- Involuntarily
  - By OS if "misbehave" – e.g., divide by zero, invalid memory access
  - By another process (e.g., `kill` or `signal()`)

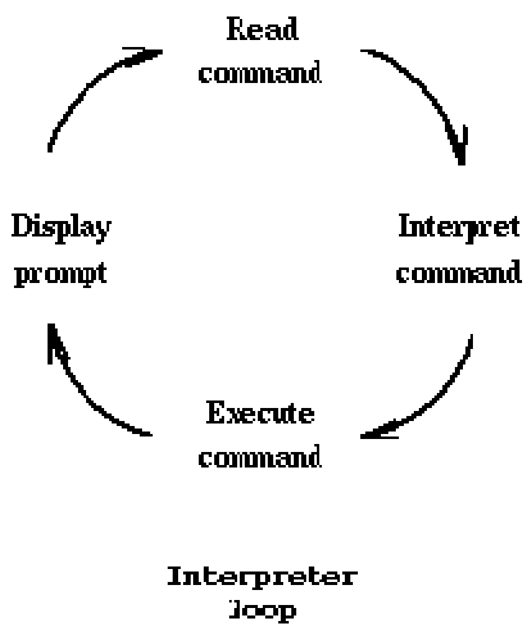


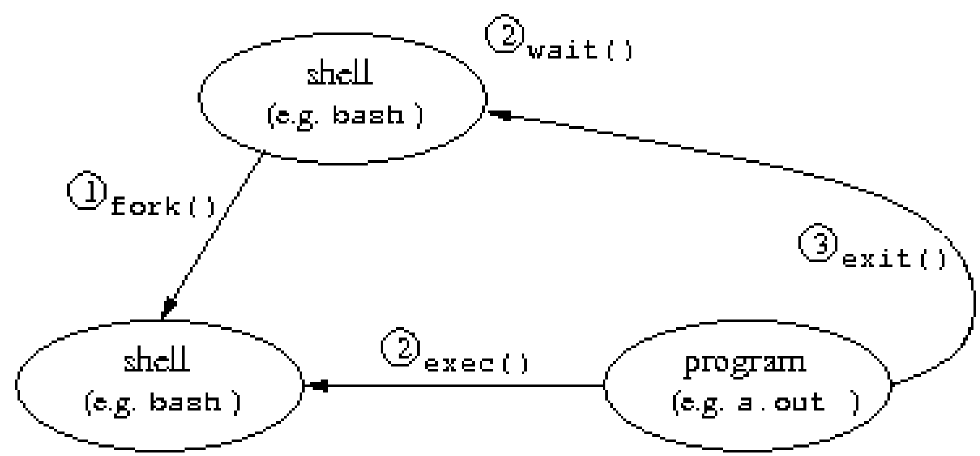
Creation/Termination Example – Unix Shell

See: “`shell-v0.c`”

- System call: `fork()`
  - Creates (nearly) identical copy of process
  - Return value different for child/parent
- System call: `exec()`
  - Over-write with new process address space
- Shell
  - Uses `fork()` and `exec()`
  - → Simple!



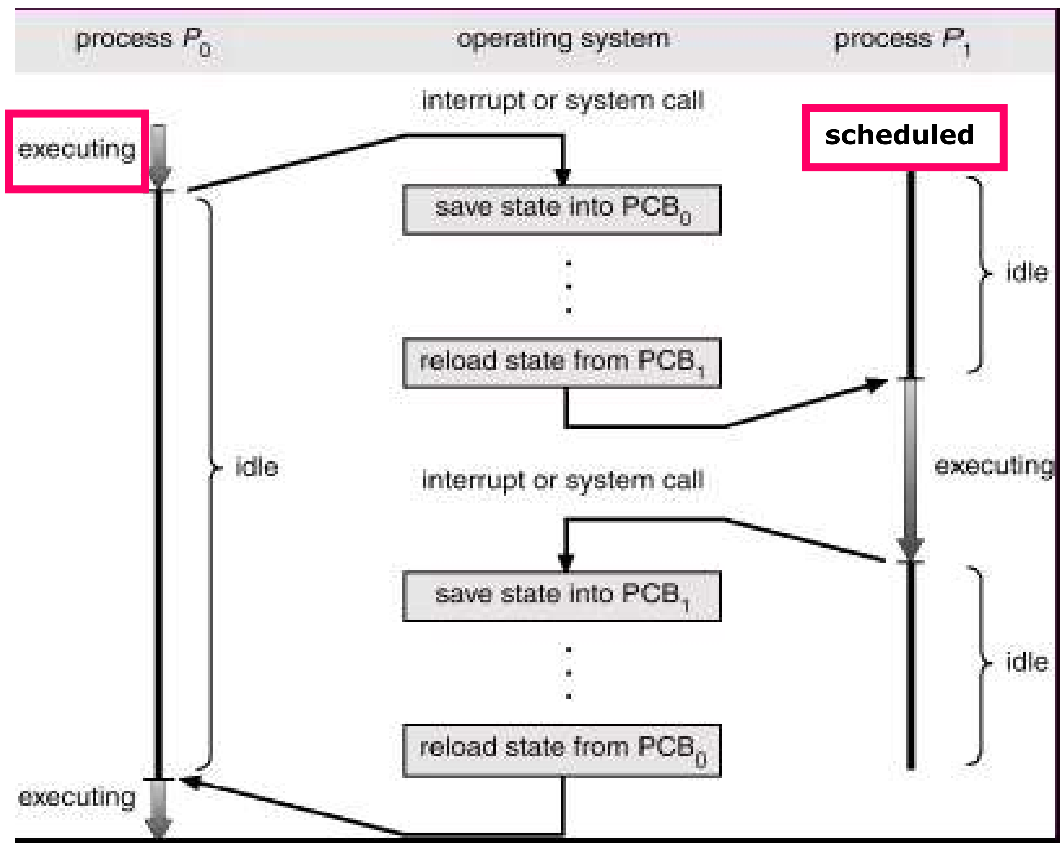




See: “`shell-v1.c`”

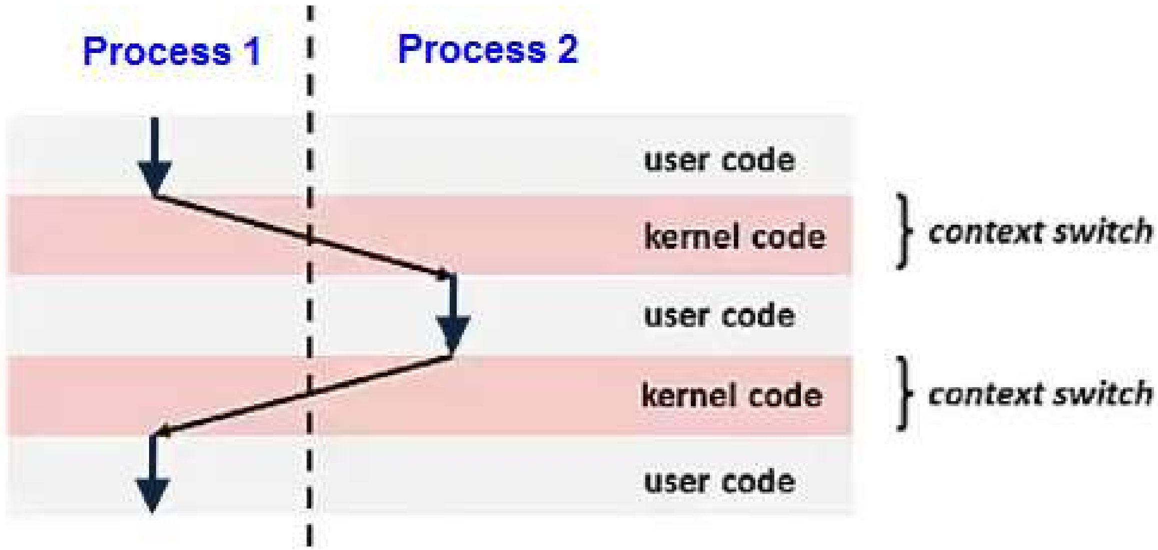
Model for Multiprogramming

- CPU switches from process to process
  - Each runs for 10s or 100s of milliseconds
  - Block for I/O
    - E.g., disk read
  - Other interrupt
    - E.g., I/O complete
  - “timeslice” is over (configurable parameter)



Silberschatz & Galvin, 5th Ed, Wiley, Fig 4.3
Operating System Concepts

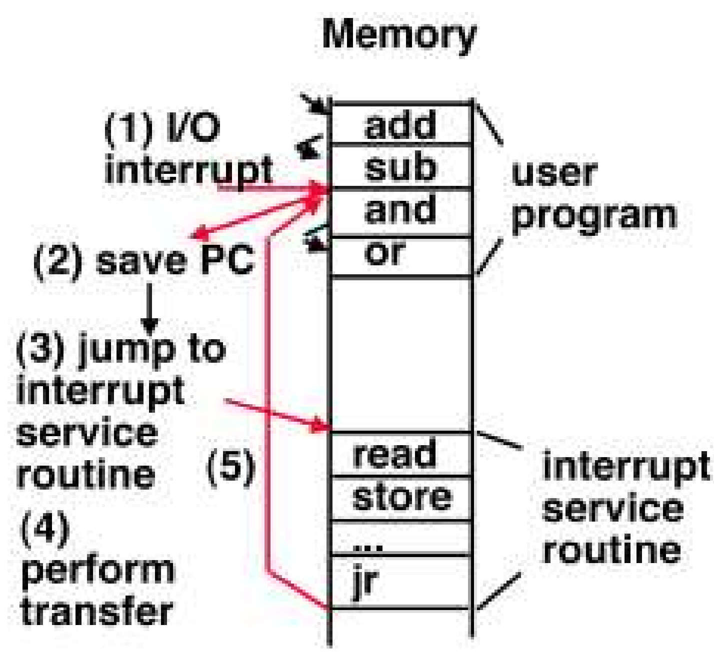
Context Switch



- Pure overhead
- So … want it to be fast, fast, fast
  – typically 1 to 1000 microseconds
- Sometimes special hardware to speed up
  – Real-time wants worst case (e.g., max 20 microseconds)
- When to switch contexts to another process is *process scheduling*

Interrupt Handling Mechanism

- Store PC (hardware)
- Load new PC (hardware)
  - Jump to interrupt service procedure
- Save PCB information (assembly)
- Set up new stack (assembly)
- Set "*waiting*" proc to "*ready*" (C)
- Service interrupt (C and assembly)
- Invoke scheduler (C)
  - Newly awakened process (*context-switch*)
  - Previously running process

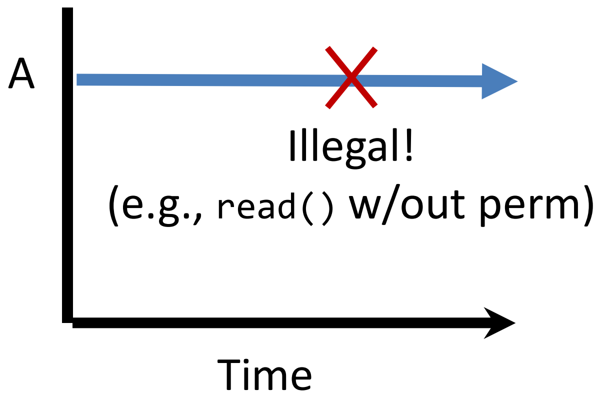


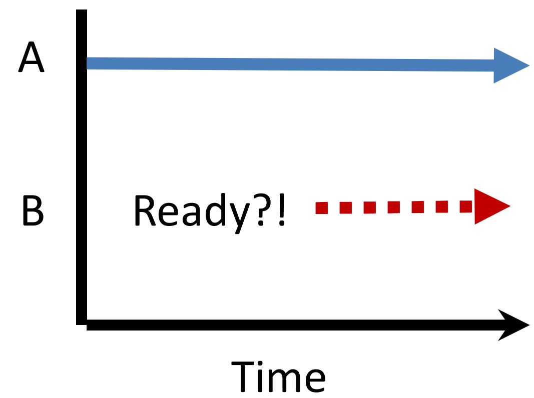
Outline

- Motivation (done)
- Control block (done)
- Switching (done)
- Control (next)

Chapter 6
OPERATING SYSTEMS: THREE EASY PIECES
http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-mechanisms.pdf

The Problem – Virtualizing CPU with Control





THE CRUX OF THE PROBLEM:
HOW TO EFFICIENTLY VIRTUALIZE CPU WITH CONTROL?

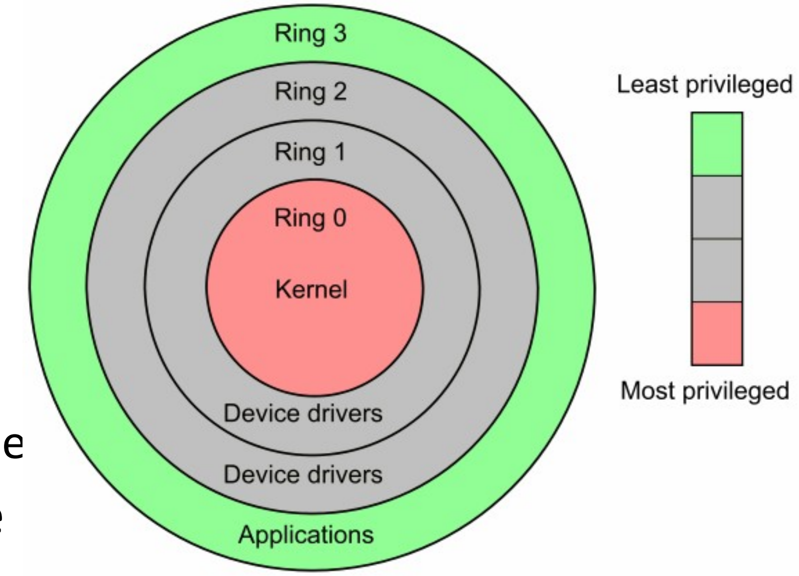
OS must virtualize CPU efficiently while retaining control over system. Note: hardware support required!

Solution – Limited Direct Execution

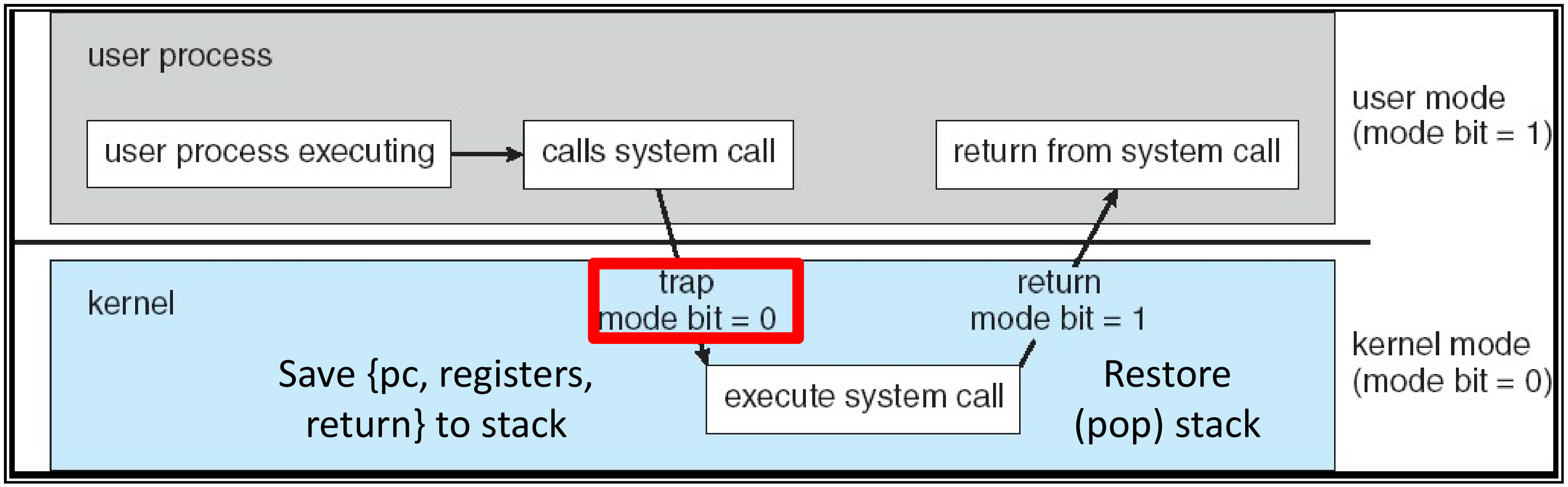
- Hardware provides two (sometimes more) modes
  - User mode – certain operations/access not allowed
  - Kernel mode – full access allowed
- Allows OS to protect against
  - Faulty processes
  - Malicious processes
- Some instructions and memory locations are designated as privileged
  - Only executable or accessible in kernel mode
- System calls, traps, and interrupts change mode from *user* to *kernel*
  - Return from system call resets mode to user

Still allow programs to directly run (e.g., on CPU) – i.e., no "sandbox" interpretation

But *limit* permissions

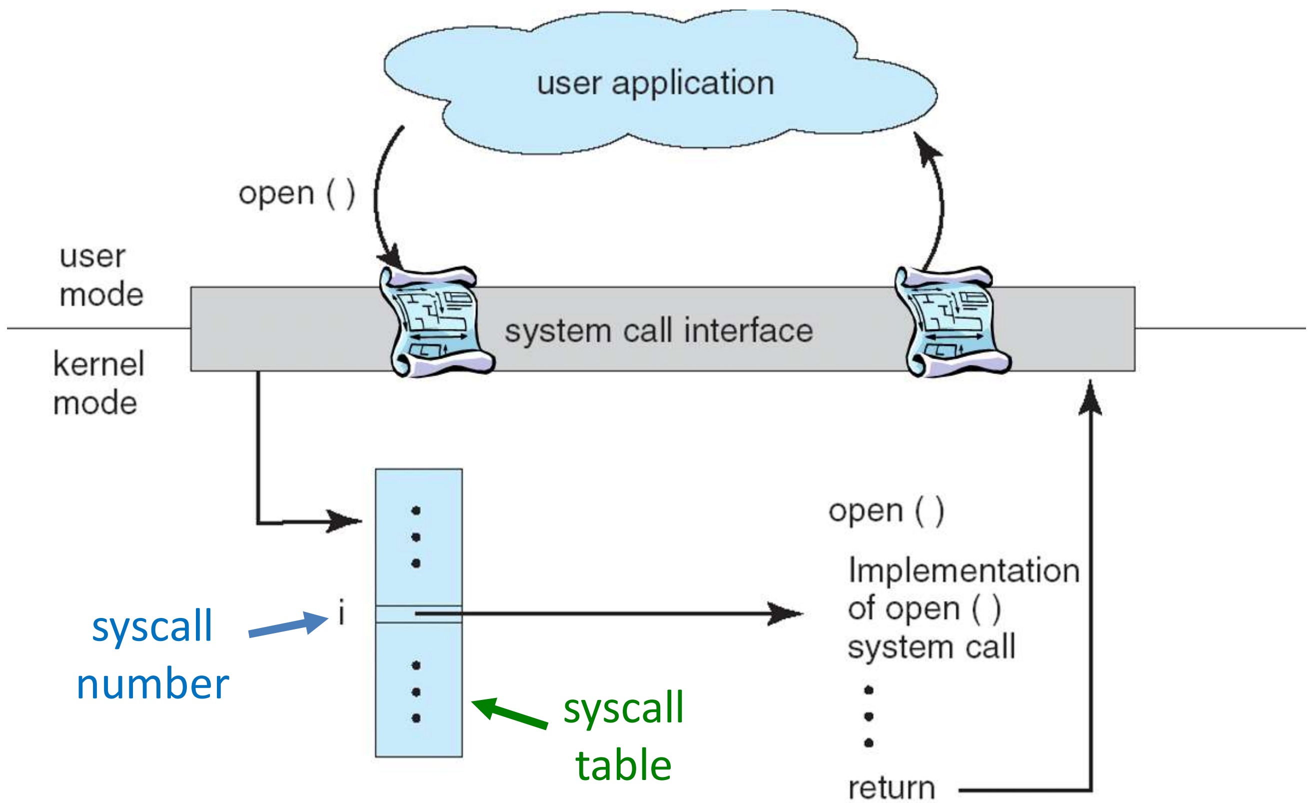


Trap – Transition User to Kernel Mode



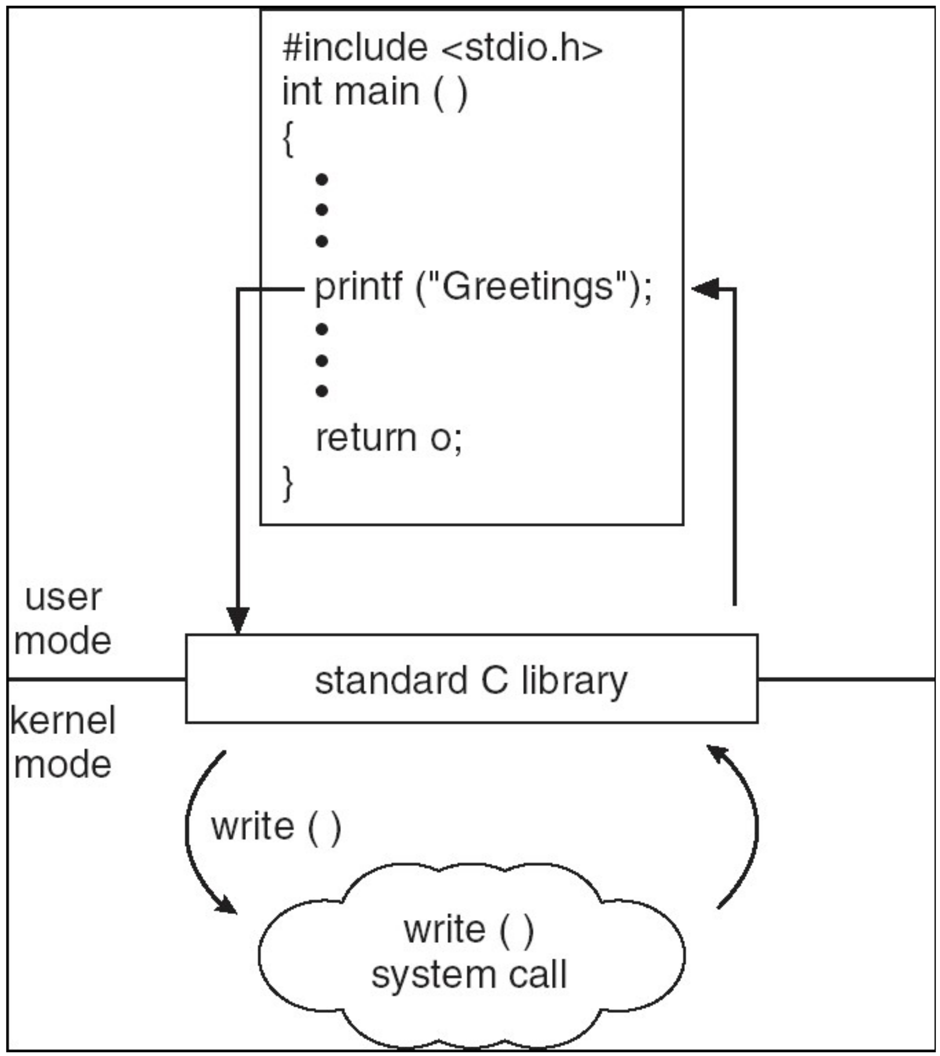
- But … wow to know what code to execute for system call? i.e., how to know where system call is?

Trap – System Call Lookup Table



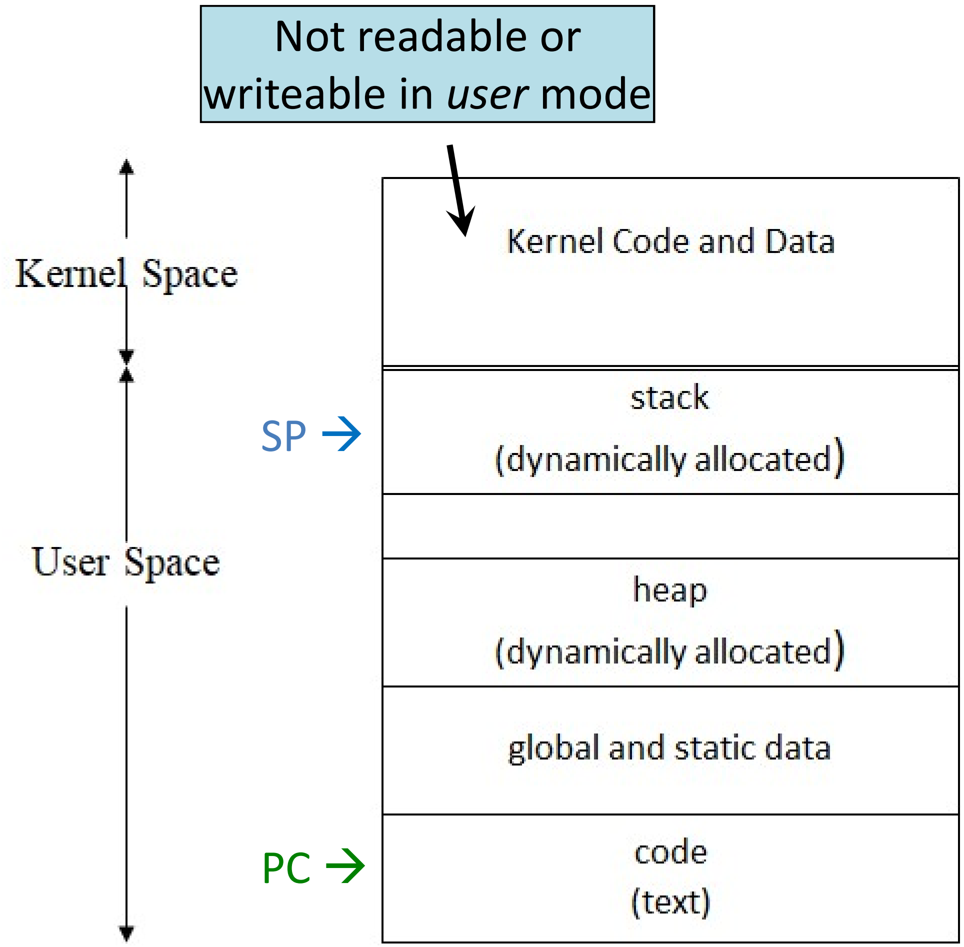
- Each system call has own number/identity
  - Initialized at boot time
- Kernel trap handler uses syscall number to index into table of syscall routines
  - Unique to each OS

E.g., Accessing Kernel via Library



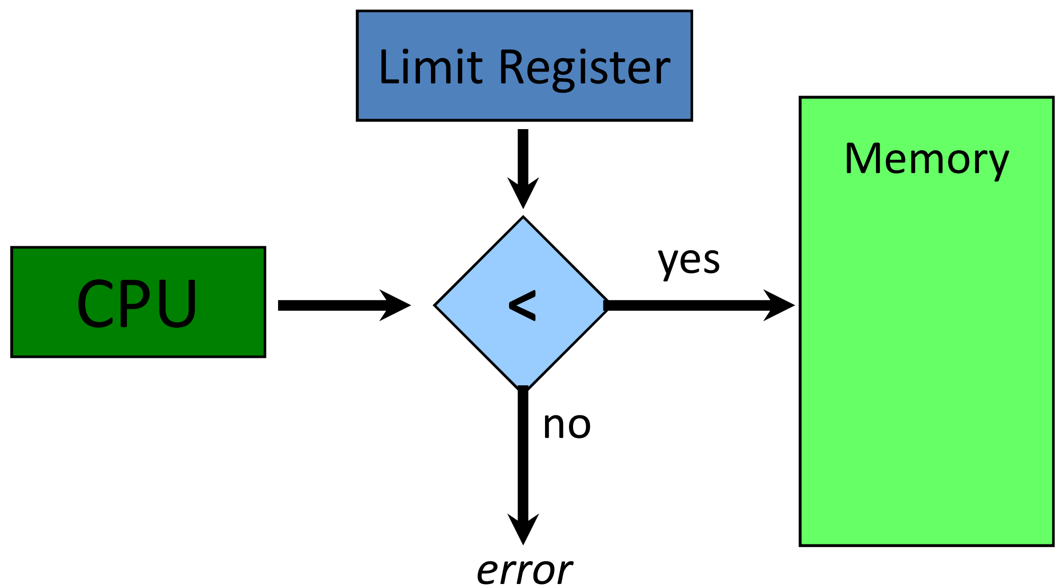
Inside Kernel Mode, OS can …

- Read and modify data structures not in user address space
- Control devices and hardware settings forbidden to user processes
- Invoke operating system functions not available to user processes
- Access address of space of invoking process



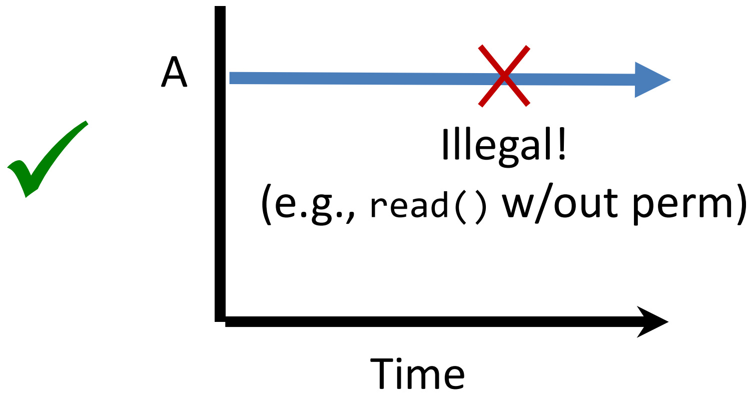
Involuntary Transition User to Kernel Mode

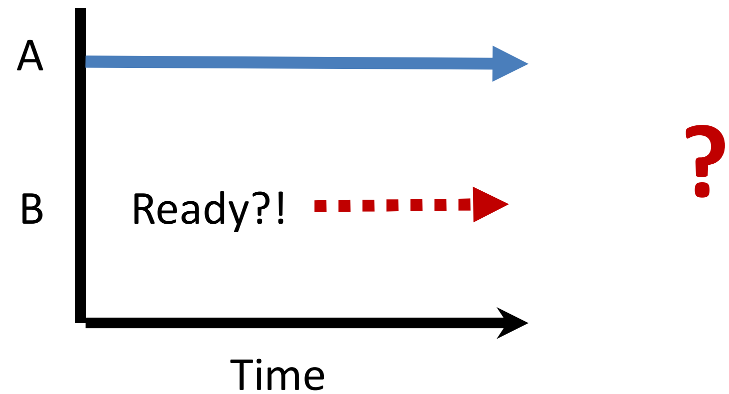
- E.g., in user mode, memory violation generates interrupt



Switch to kernel mode
Handle error (e.g., terminate process)

The Problem – Virtualizing the CPU



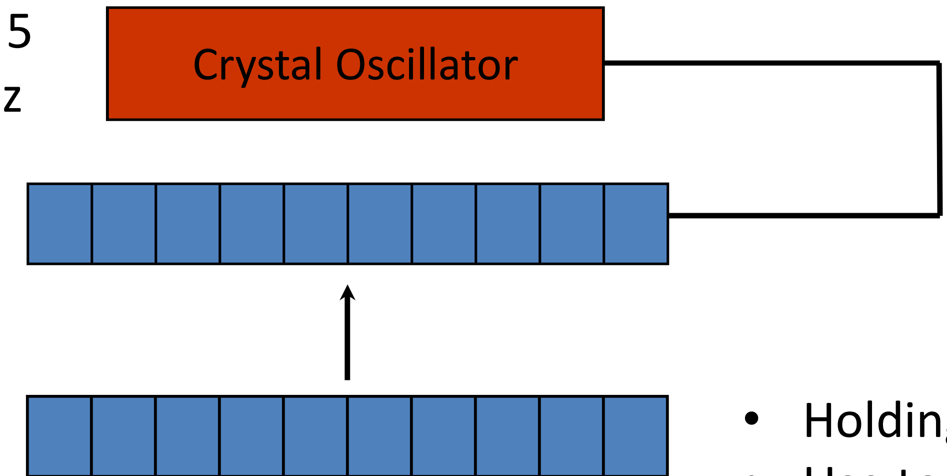


THE CRUX OF THE PROBLEM:
HOW TO EFFICIENTLY VIRTUALIZE CPU WITH CONTROL?

What if process doesn't voluntarily give up control? It doesn't make a system call (so, can't check) and it doesn't make a violation. e.g., `while(1) {}`

Solution – Special Timer Hardware

Pulse from 5 to 300 MHz



Decrement counter when == 0
→ generate interrupt

- Holding register to load counter
- Use to control clock ticks (i.e., length of timer)

- When timer interrupt occurs, OS regains control
- E.g., can run scheduler to pick new process

Outline

- Motivation (done)
- Control block (done)
- Switching (done)
- Control (done)