Virtualization:
The CPU

Questions answered in this lecture:

What is a process?

Why is limited direct execution a good approach for virtualizing the CPU?

What execution state must be saved for a process?

What 3 modes could a process be in?

Announcements:

Read chapters 1 – 6

*Slide content borrowed from
Andrea Arpaci-Dusseau @UW

ILLINOIS INSTITUTE
OF TECHNOLOGY

What is a Process?

Process: An execution stream in the context of a process state

What is an execution stream?

- Stream of executing instructions
- Running piece of code
- “thread of control”

What is process state?

- Everything that the running code can affect or be affected by
- Registers
- General purpose, floating point, status, program counter, stack pointer
- Address space
- Heap, stack, and code
- Open files

Processes vs. Programs

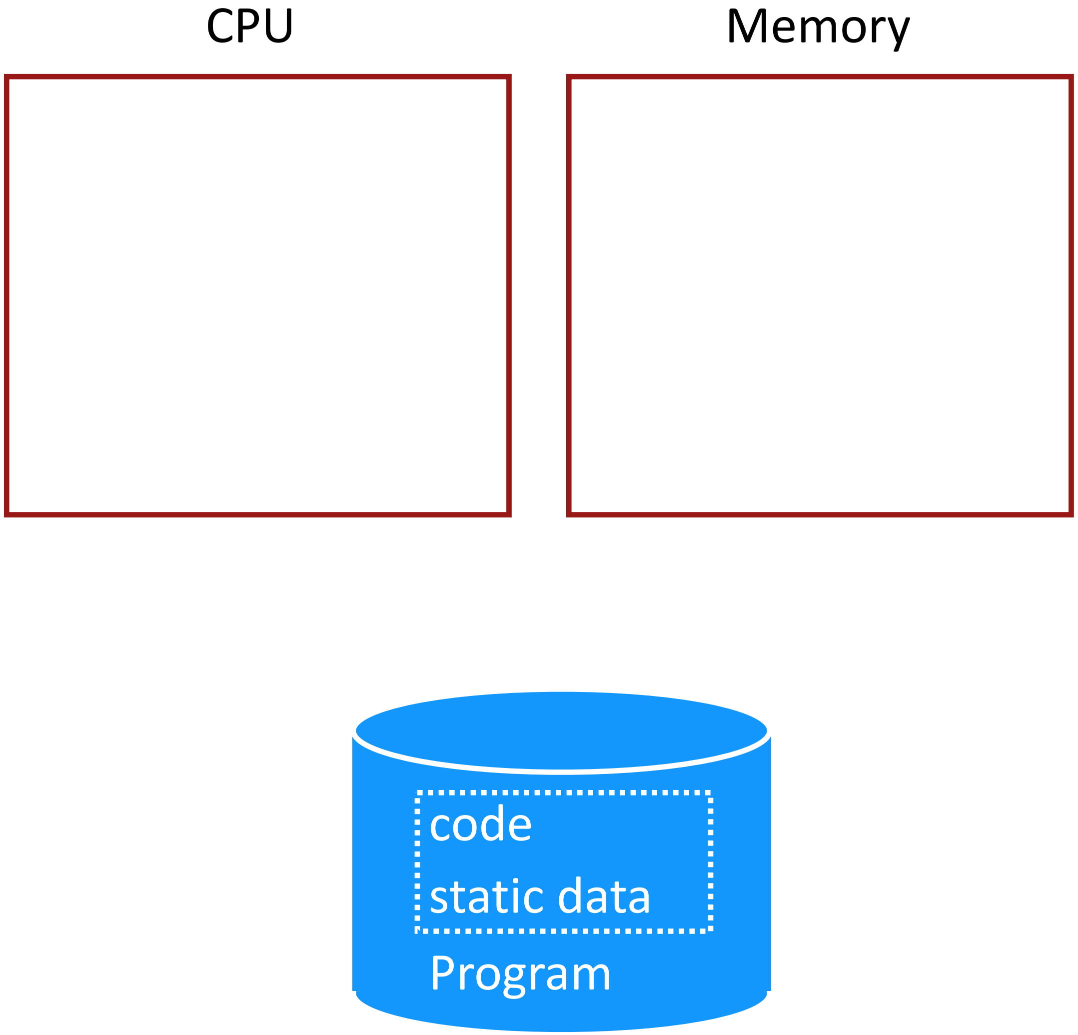
A process is different than a program

- Program: Static code and static data
- Process: Dynamic instance of code and data

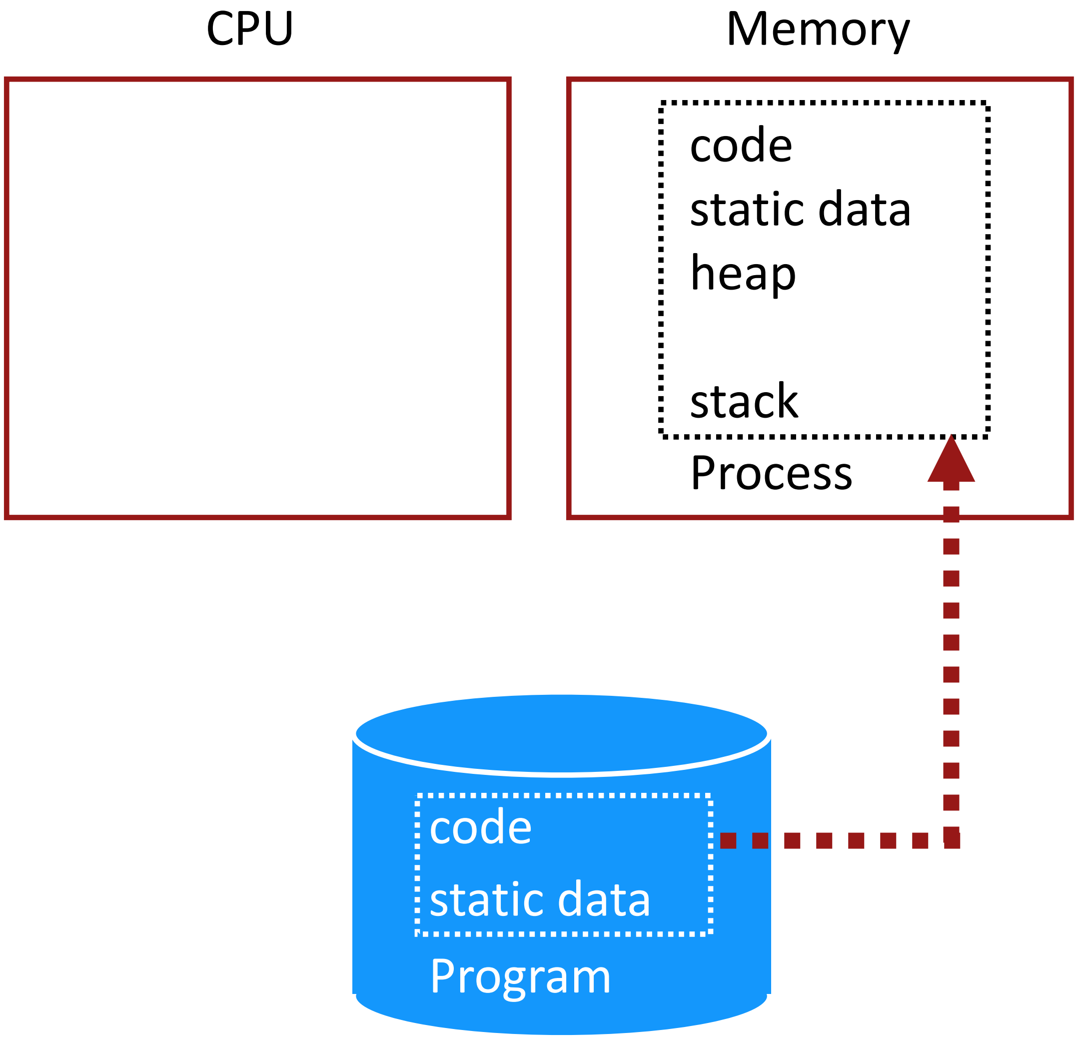
Can have multiple process instances of same program

- Can have multiple processes of the same program
  Example: many users can run “ls” at the same time

Process Creation



Process Creation



CPU

Memory

code
static data
heap

stack
Process

code
static data
Program

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
  - These days processes are “made of” threads
- 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:
Give each process impression it 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)

How to Provide Good CPU Performance?

Direct execution

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

Problems with direct execution?

1. Process could do something restricted

   Could read/write other process data (disk, memory) or restricted device

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 and hardware maintain some control

Problem 1:
Restricted Operations

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)
  - Instructions for interacting with devices
  - 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 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

System Call


Process P

RAM

① ②

read():

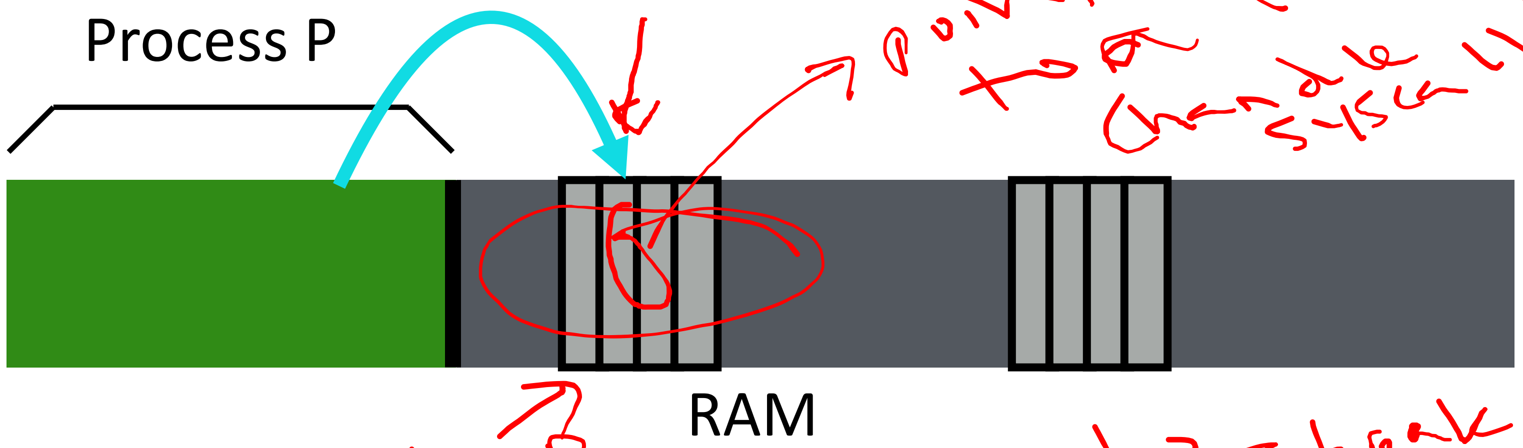
```
movl $6, %eax;     int $128
```

vector

ST

SW interrupt

System Call



128

pointer to a function (handle syscall)

trap table

$3 = break pt trap

```
movl $6, %eax;    int $128
```

syscall-table index

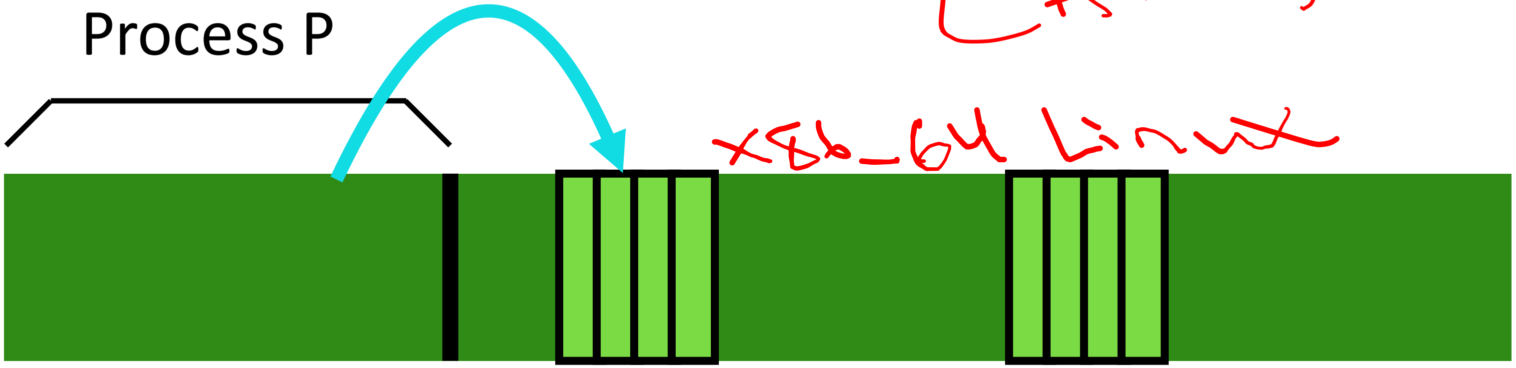
which syscall do I want?

trap-table index

which trap do I mean?

System Call

Application Binary Interface (ABI)



x86_64 Linux

RAM

movl $7, %edi

```
movl $6, %eax;   int $128
```

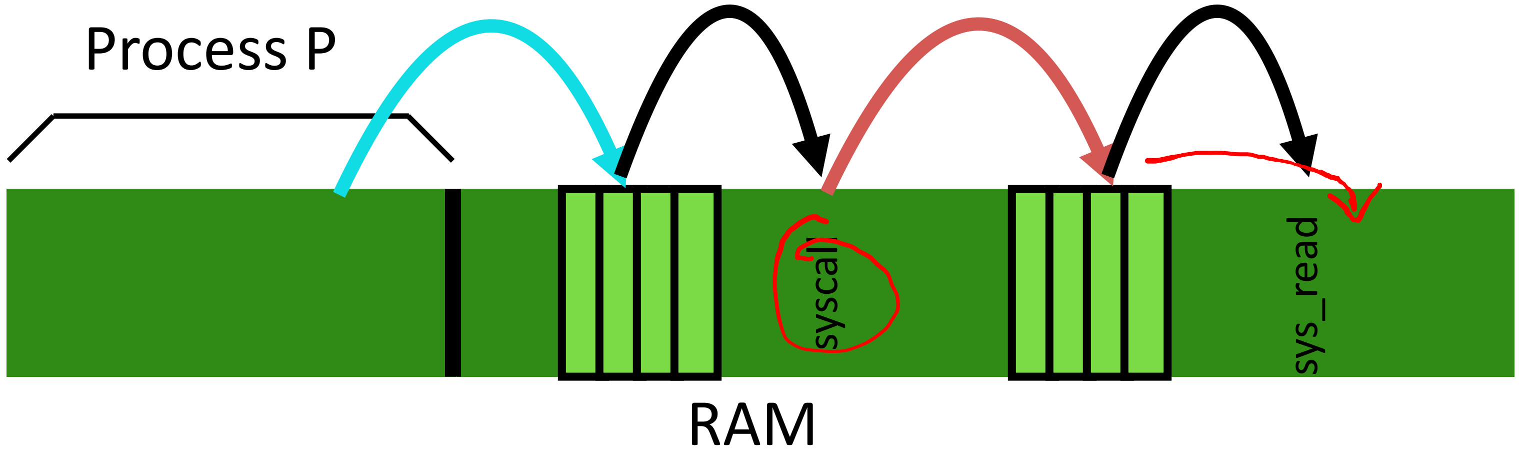
%ebx

syscall-table index

trap-table index

Kernel mode: we can do anything!

System Call

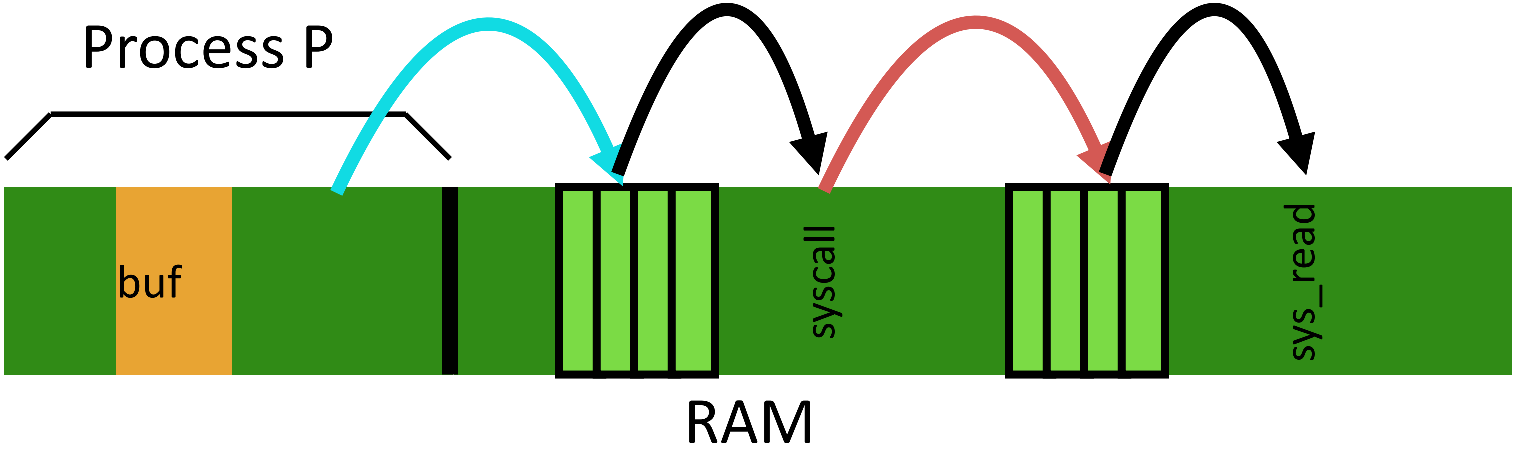


syscall-table index

trap-table index

Follow entries to correct system call code

System Call



```
movl $6, %eax;    int $128
```

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 to limit?

User processes are not allowed to perform:

- General memory access
- Disk I/O
- Special hardware instructions (e.g. on x86, `lidt`)

What if process tries to do something restricted?

Problem 2: How to take CPU away?

OS requirements for **multiprogramming** (or multitasking)

- Mechanism
  - To switch between processes
- Policy
  - To decide which process to schedule when

Separation of policy and mechanism

- Recurring theme in OS
- 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?

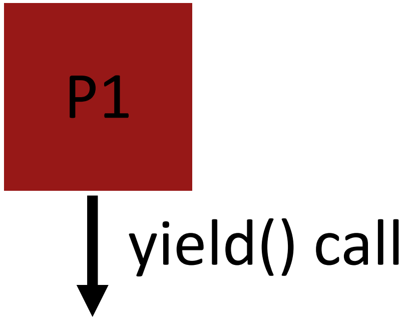
Question 2: What execution context must be saved and restored?

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

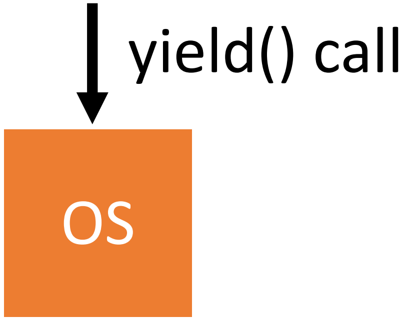
Cooperative Approach



P1

yield() call

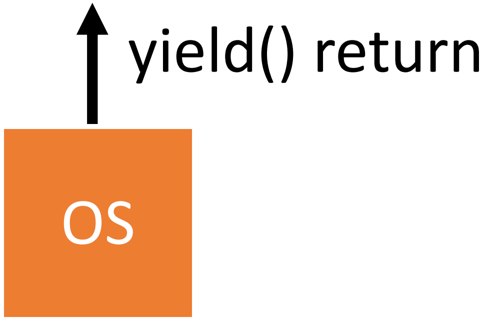
Cooperative Approach



yield() call

OS

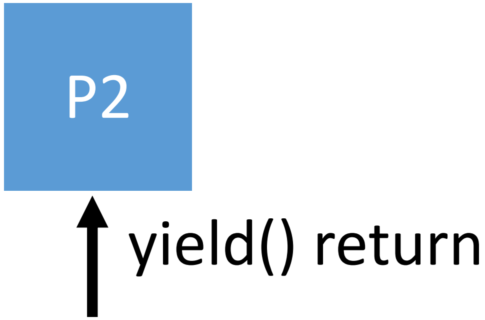
Cooperative Approach



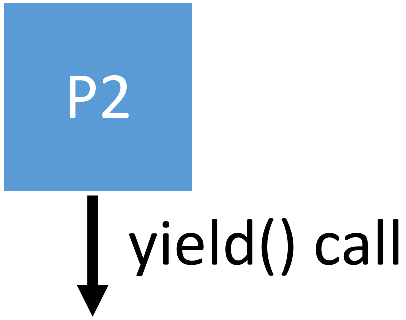
yield() return

OS

Cooperative Approach



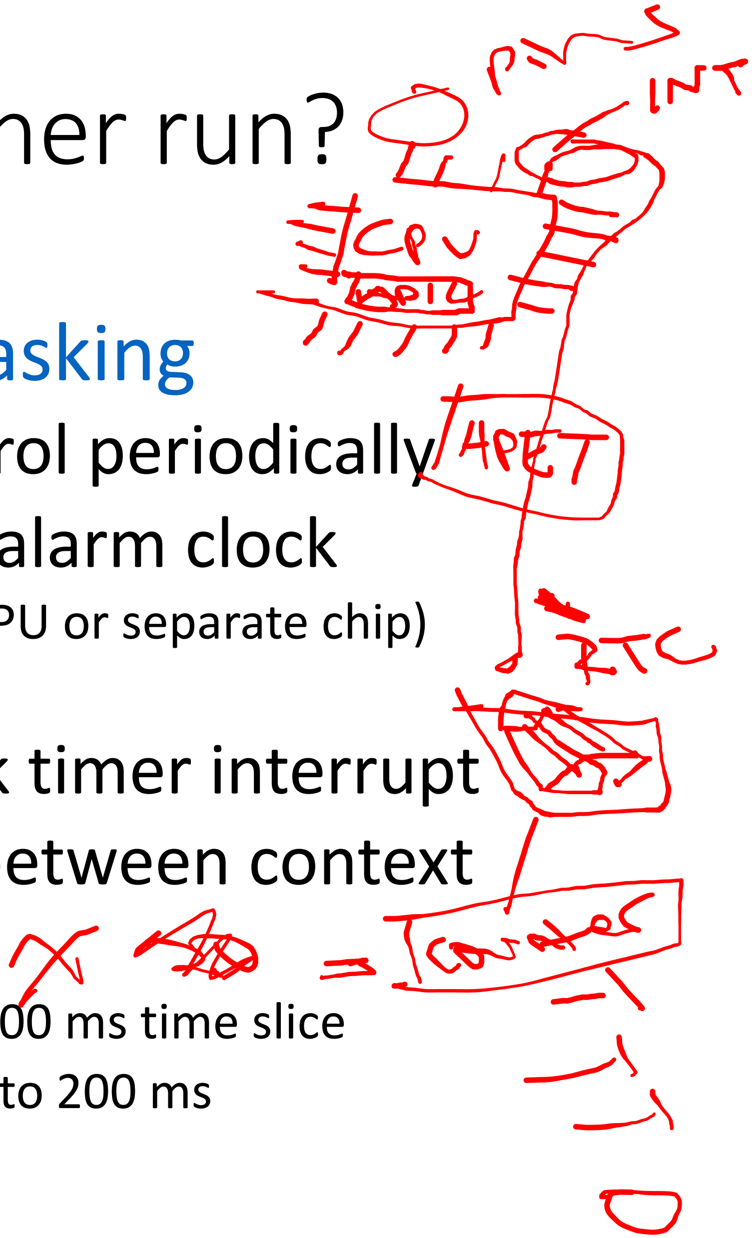
Cooperative Approach



Q1: How Does Dispatcher Run?

- 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 (most) modern operating systems

Q1: How does Dispatcher run?



Option 2: Preemptive 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
- Dispatcher counts interrupts between context switches
  - Example: Waiting 20 timer ticks gives 200 ms time slice
  - Common time slices range from 10 ms to 200 ms

Q2: What Context must be Saved?

Dispatcher must track context of process when not running

- Save context in process control block (PCB) (or, process descriptor)

What information is stored in PCB?

- PID
- Process state (I.e., running, ready, or blocked)
- Execution state (all registers, PC, 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)

Requires special hardware support

- Hardware saves process PC and PSR on interrupts

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	
	jmp to OS	

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

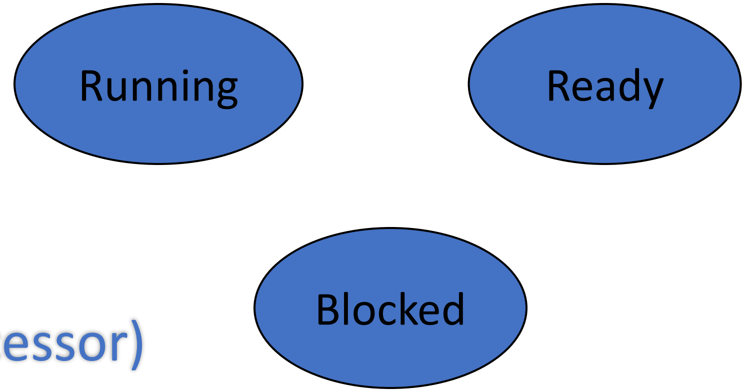
Operating System	Hardware	Program
		Process A …
	timer interrupt save regs(A) to k-stack(A) move to kernel mode jump to trap handler	
Handle the trap **Call switch() routine** **save regs(A) to proc-struct(A)** **restore regs(B) from proc-struct(B)** **switch to k-stack(B)** **return-from-trap (into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

CONTEXT SWITCH

Problem 3:
Slow Ops such as I/O?

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



Transitions?

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

Problem 3:
Slow ops such as I/O?

FIFO

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

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) {
       // 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);
 }
}
```