UNIVERSITY of WISCONSIN-MADISON
Computer Sciences Department

CS 537
Introduction to Operating Systems

Andrea C. Arpaci-Dusseau
Remzi H. Arpaci-Dusseau

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 in?

Announcements:

- Reading: Chapters 1 – 6
- Begin Project 1 (part a – sorting)

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 efficient access to resources (cost, time, energy)
- Provide fair access to resources

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)

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 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 == PROGRAMS ?

Is a process the same as a program?

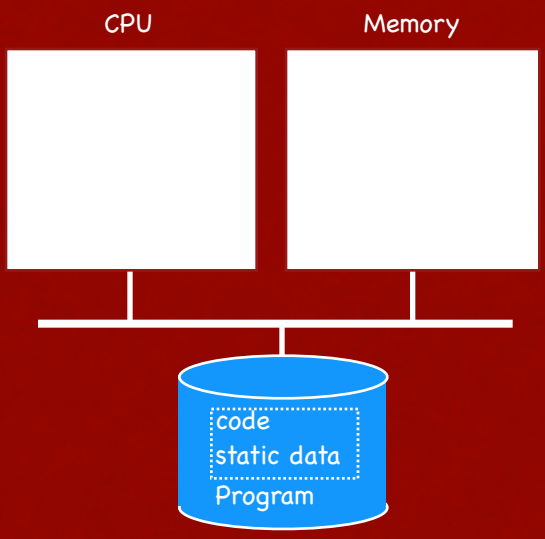
A process is different than a program

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

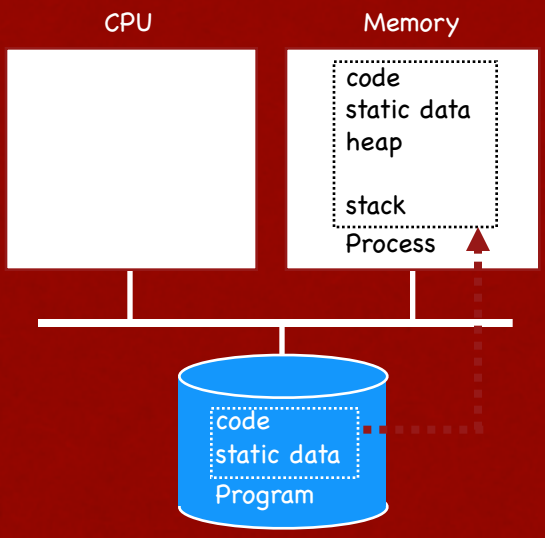
Can have multiple process instances of same program

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

PROCESS CREATION



PROCESS CREATION



Will describe details of process creation later….

PROCESSES == THREADS?

Is a process the same as a thread?

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)

CPU VIRTUALIZATION ATTEMPT #1

Direct execution

- OS allows user process to run directly on CPU (advantage: no overhead!)
- 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 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 and hardware maintain some control

PROBLEM 1: RESTRICTED OPS

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

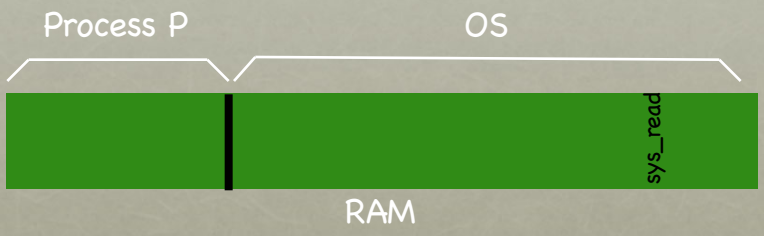
Solution: privilege levels supported by hardware (bit of status)

- User processes run in **user mode** (restricted mode)
- OS runs in **kernel mode** (not restricted)
  - Instructions for interacting with devices and memory
- Could have higher privilege levels too (advanced topic: virtual machines)

How can user process access device?

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

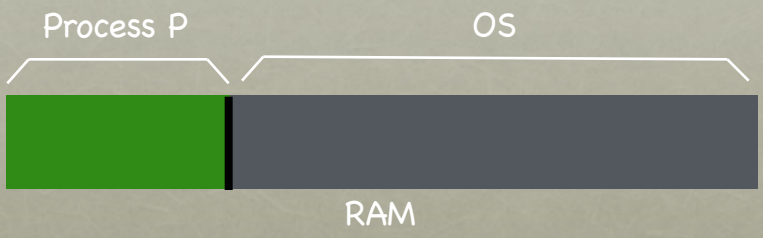
SYSTEM CALL



P wants to call read()

Why can't P just invoke read()??

SYSTEM CALL



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

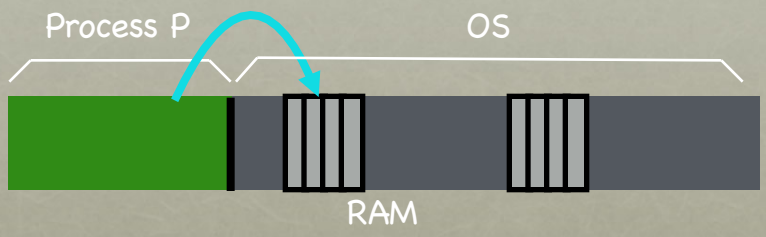
P wants to call read() but no way to call it directly

SYSTEM CALL

read():

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

SYSTEM CALL

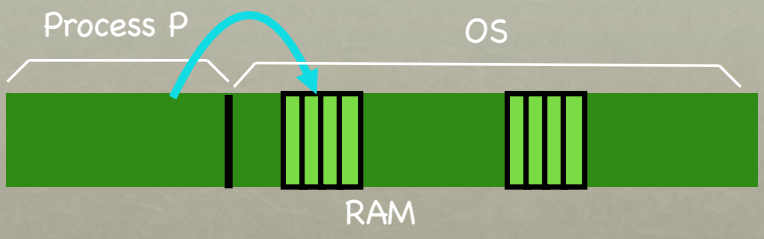



```
movl $6, %eax;    int $64
```

syscall-table index

trap-table index

SYSTEM CALL



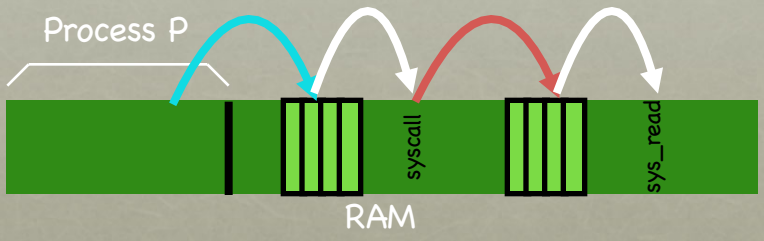

```
movl $6, %eax;    int $64
```

syscall-table index

trap-table index

Kernel mode: we can do anything!

SYSTEM CALL



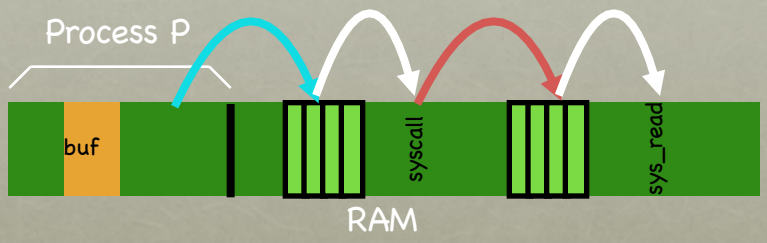

```
movl $6, %eax;    int $64
```

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

WHAT TO LIMIT?

User processes are not allowed to perform:

- General memory access
- Disk I/O
- Special x86 instructions like `lidt`
  (Load Interrupt Descriptor Table Register)

What if process tries to do something restricted?

REPEAT: HOW TO VIRTUALIZE CPU?

Direct execution

- OS allows 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 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 and hardware maintain some control

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

- Reoccuring 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 to stop process A?

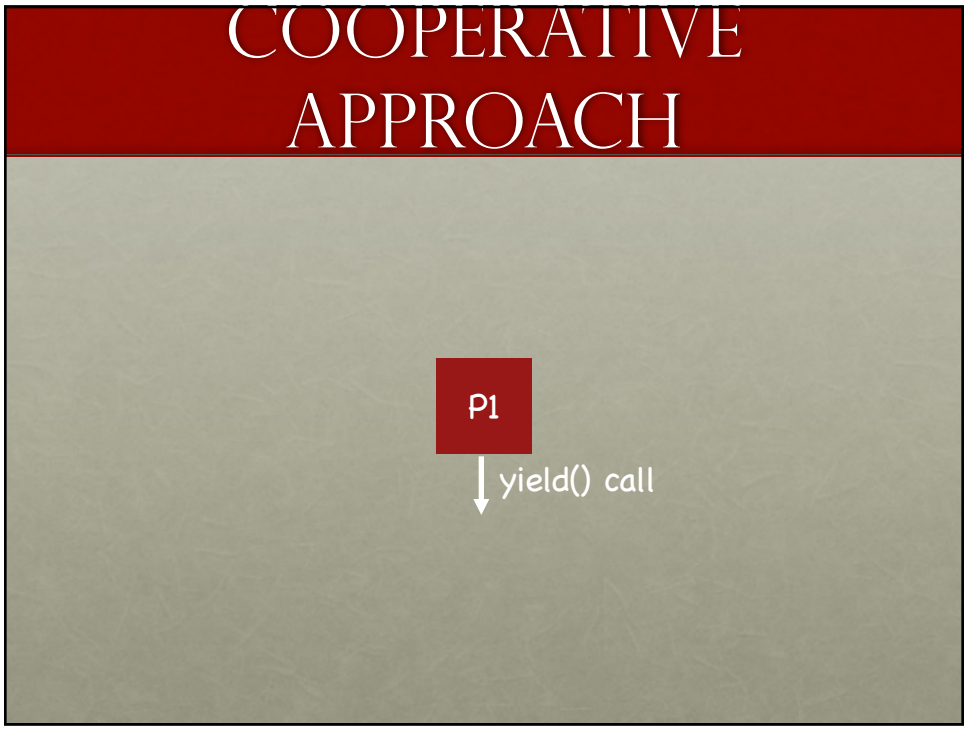
Question 2: What execution context must be saved and restored?

Q1: HOW DOES DISPATCHER GET CONTROL?

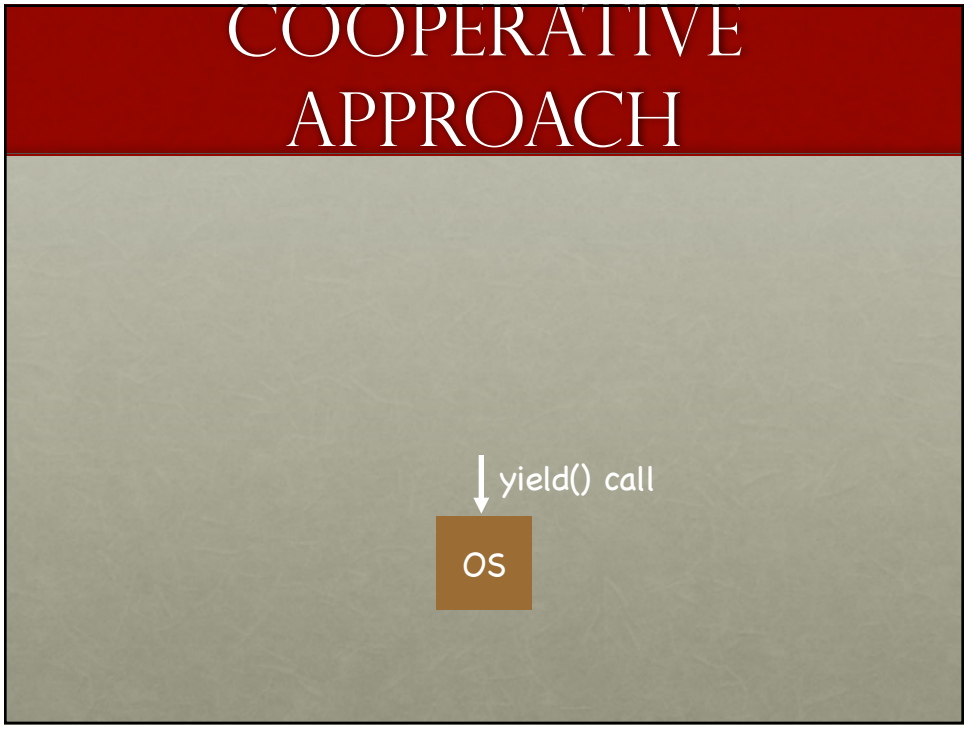
Option 1: Cooperative Multi-tasking

- Trust process to relinquish CPU to OS through traps
  - System calls
  - Page faults (access page not resident in main memory)
  - Errors (illegal instruction or divide by zero)
- Provide new `yield()` system call

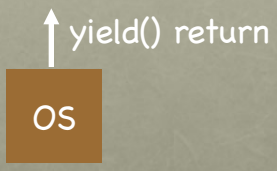
COOPERATIVE APPROACH



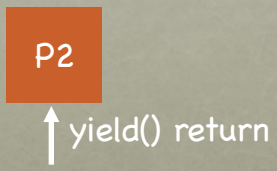
COOPERATIVE APPROACH



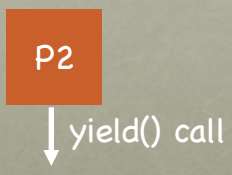
COOPERATIVE APPROACH



COOPERATIVE APPROACH



COOPERATIVE APPROACH



Q1: HOW DOES DISPATCHER RUN?

Problem with cooperative approach?

Disadvantages: Processes can misbehave

- If avoid all traps and perform no I/O, can take over machine!
- Only solution?
  - Reboot!

Not performed in modern operating systems

Q1: HOW DOES DISPATCHER RUN?

Option 2: 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 (turn off) 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

REPEAT: 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 to stop process A?~~

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

Q2: WHAT CONTEXT MUST BE SAVED?

Dispatcher must track context of process when not running

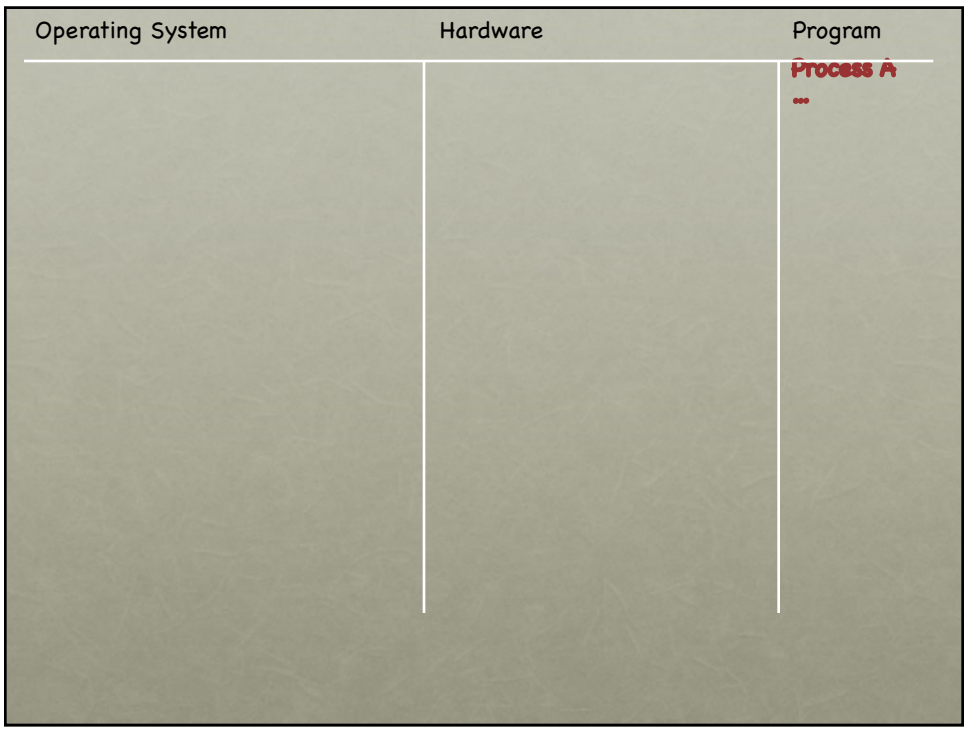
- Save context in process control block (PCB) (or, process descriptor)

What information is stored in PCB?

- Execution state (all registers, program counter (PC), stack ptr, frame ptr)
- PID
- Process state (I.e., running, ready, or blocked)
- 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 (process status register) on interrupts



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

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 PCB(A) restore regs(B) from PCB(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 PCB(A)		
restore regs(B) from PCB(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 PCB(A)		
restore regs(B) from PCB(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
		...

REPEAT: HOW TO VIRTUALIZE CPU?

Direct execution

- OS allows 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 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 and hardware maintain some control

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 later)

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?

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

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);
  }
}
```

SUMMARY

Virtualization:
OS gives each process impression it has its own CPU with time-sharing

Key mechanism: context-switching

Direct execution makes processes fast

Limited execution at key points to ensure OS retains control
- Don't let user process execute all instructions
- Be able to remove CPU from process when needed (true multi-tasking)
- Switch to another process when running process performs I/O

Hardware provides a lot of OS support
- user vs kernel mode
- timer interrupts
- automatic register saving