# OSTEP
# CPU Virtualization
# Process

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?

# 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

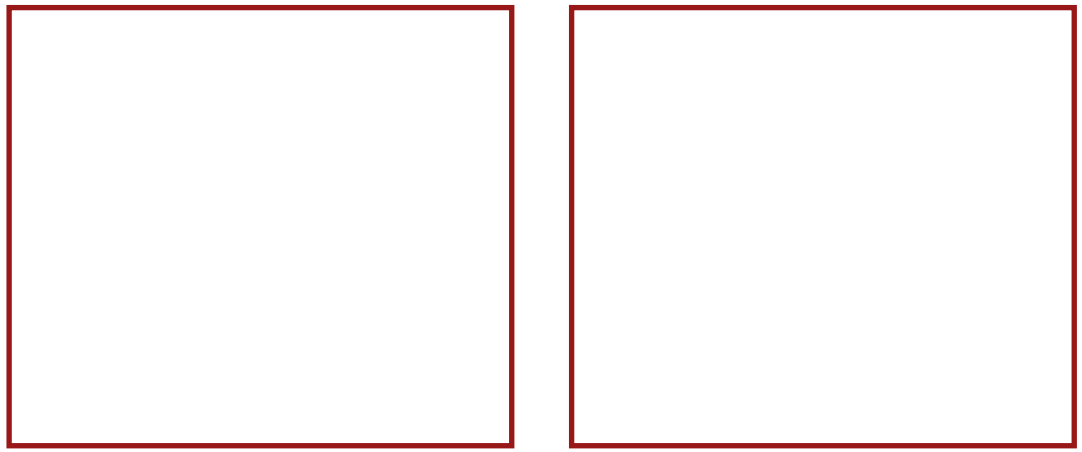
# Processes vs. Programs

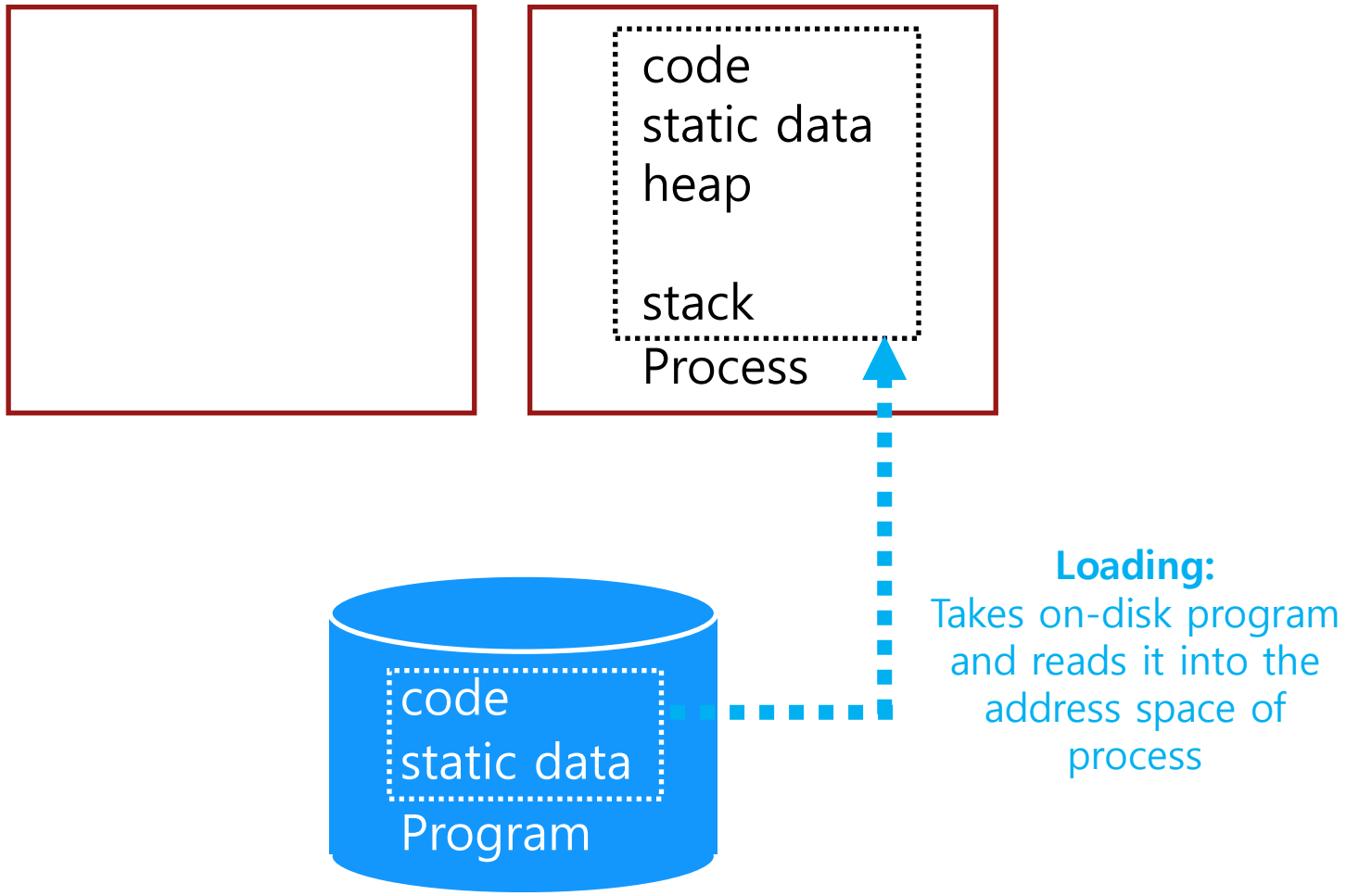
A process is different from a program

- 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


code
static data
Program
Disk


code
static data
heap
stack
Process
code
static data
Program
Disk
Loading:
Takes on-disk program and reads it into the address space of process

# Processes vs. Threads

A process is different from 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: 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())

| **OS** | **Program** |
| --- | --- |
| 1. Create entry for process list<br>2. Allocate memory for program<br>3. Load program into memory<br>4. Set up stack with `argc` / `argv`<br>5. Clear registers<br>6. Execute call `main()` | |
| | 7. Run `main()`<br>8. Execute `return` from `main()` |
| 9. Free memory of process<br>10. Remove from process list | |

# How to Provide Good CPU Performance? (CONT.)

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

Without *limits* on running programs,
the OS wouldn't be in control of anything and
thus would be "just a library"

Solution:

**Limited direct execution** – OS and hardware maintain some control

# Problem 1: Restricted OPs

How can we ensure user process can't harm others?
What if a process wishes to perform some kind of restricted operation such as …

- Issuing an I/O request to a disk
- Gaining access to more system resources such as CPU or memory

**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 a process access device?

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

# System Call

Allow the kernel to carefully expose certain key pieces of functionality to user programs, such as …

- Accessing the file system
- Creating and destroying processes
- Communicating with other processes
- Allocating more memory

# System Call (Cont.)

**Trap** instruction (int 0x00)

- Jump into the kernel
- Raise the privilege level to kernel mode

**Return-from-trap** instruction

- Return into the calling user program
- Reduce the privilege level back to user mode

# System Call

Process P


RAM

P wants to call read()

# System Call

Process P

RAM

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

# System Call

Process P

RAM

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

# System Call

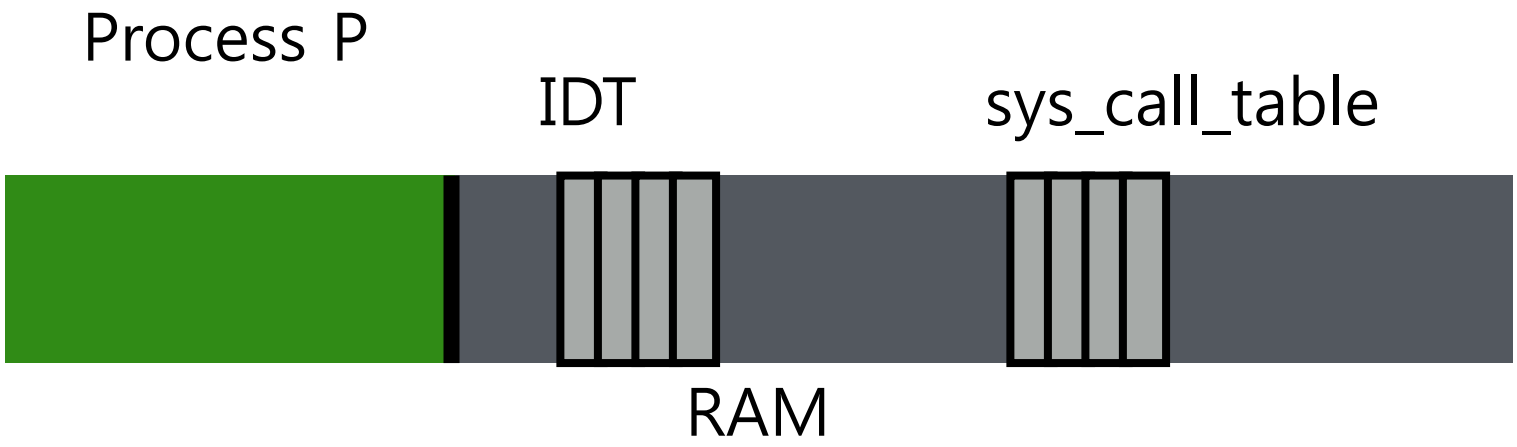
Process P

RAM

read():

```
movl $3, %eax;   int 0x80
```

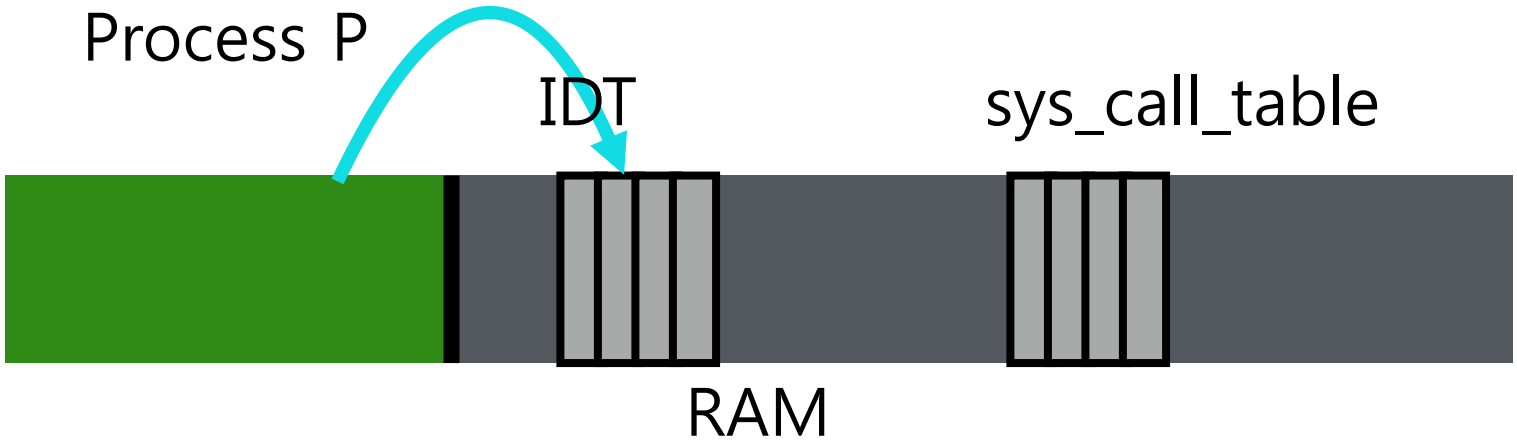
# System Call




movl $3, %eax;  int 0x80

syscall-table index

# System Call

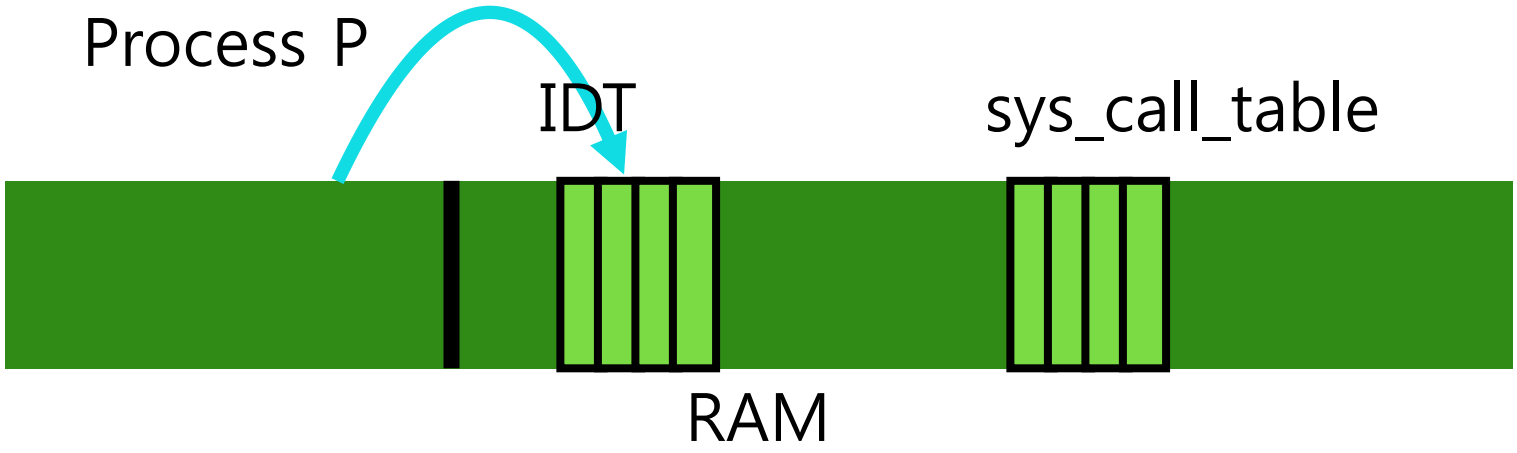



movl $3, %eax;  int 0x80

syscall-table index

trap-table index

# System Call



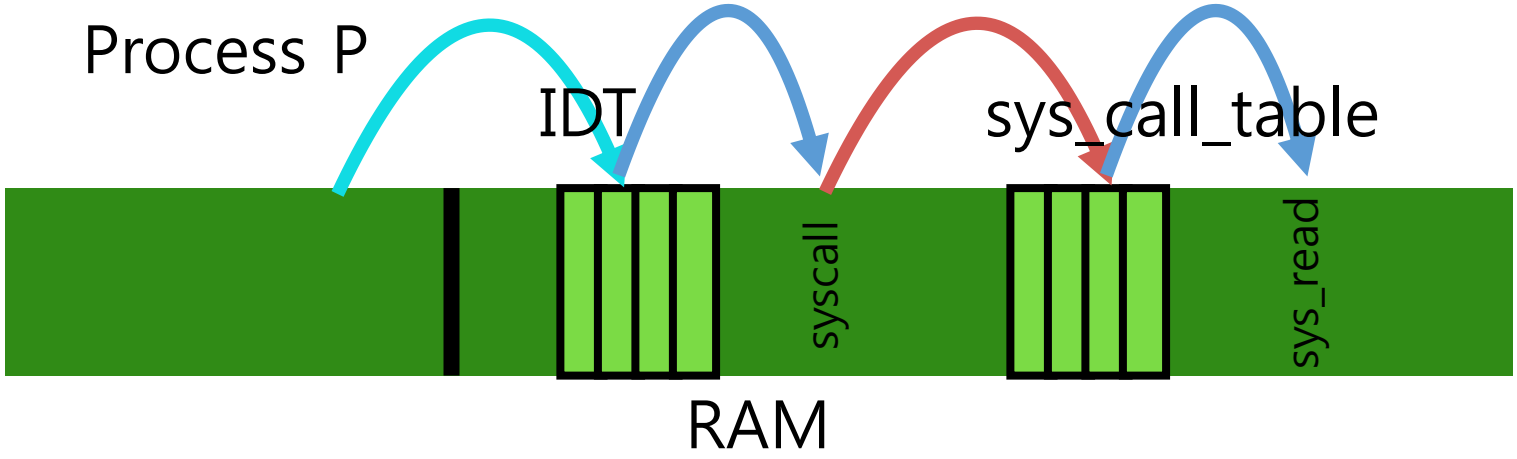

movl $3, %eax;  int 0x80

syscall-table index

trap-table index

Kernel mode: we can do anything!

# System Call



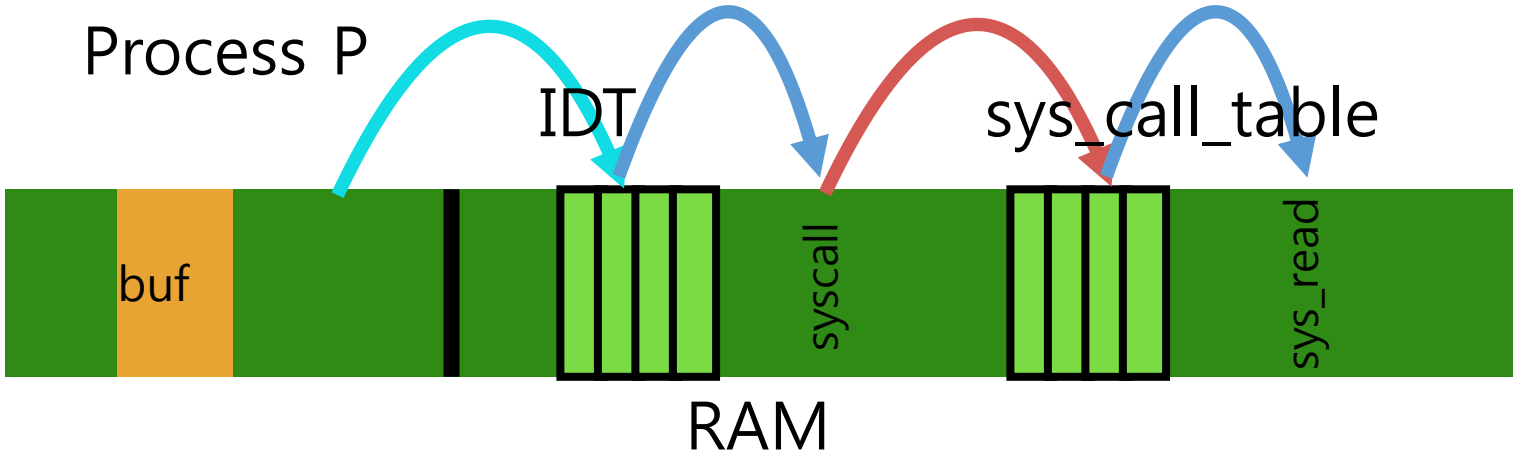

movl $3, %eax;  int 0x80

syscall-table index

trap-table index

Follow entries to correct system call code

# System Call




movl $3, %eax;  int 0x80

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

# Limited Direct Execution

# Limited Direction Execution Protocol

| OS @ boot (kernel mode) | Hardware |
|---|---|
| **initialize trap table** | remember address of … syscall handler |

| OS @ run (kernel mode) | Hardware | Program (user mode) |
|---|---|---|
| Create entry for process list<br>Allocate memory for program<br>Load program into memory<br>Setup user stack with argv<br>Fill kernel stack with reg/PC<br>**return-from -trap** | | |
| | restore regs from kernel stack<br>move to user mode<br>jump to main | |
| | | Run main()<br>…<br>Call system<br>**trap** into OS |

# Limited Direction Execution Protocol (Cont.)

| OS @ run (kernel mode) | Hardware | Program (user mode) |
|---|---|---|
| | *(Cont.)* | |
| | save regs to kernel stack<br>move to kernel mode<br>jump to trap handler | |
| Handle trap<br>Do work of syscall<br>**return-from-trap** | | |
| | restore regs from kernel stack<br>move to user mode<br>jump to PC after trap | |
| | | ...<br>return from main<br>trap (via `exit()`) |
| Free memory of process<br>Remove from process list | | |

# What to limit?

User processes are not allowed to perform:

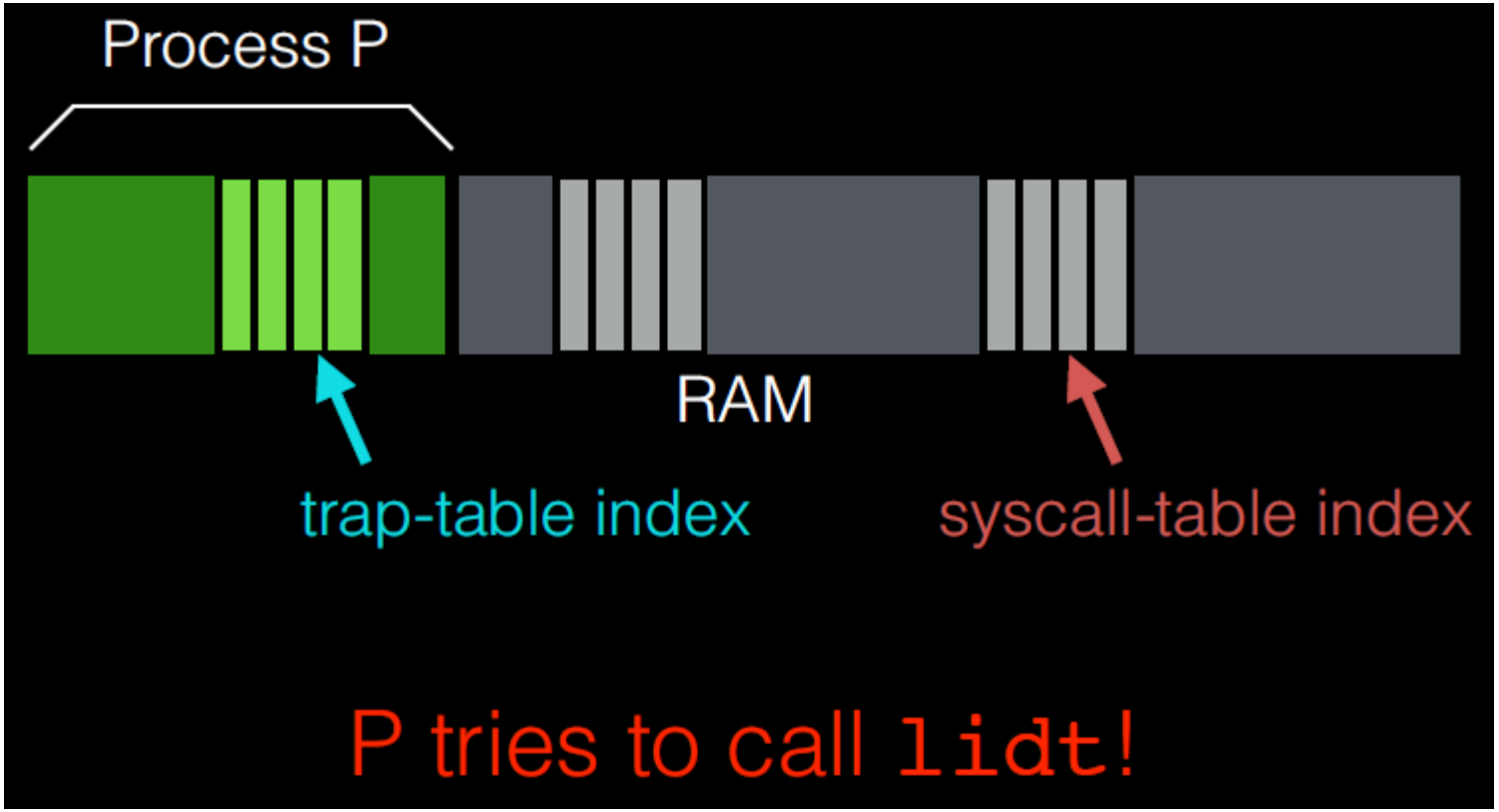
- General memory access
- Disk I/O
- Special x86 instructions like lidt

  load interrupt descriptor table register

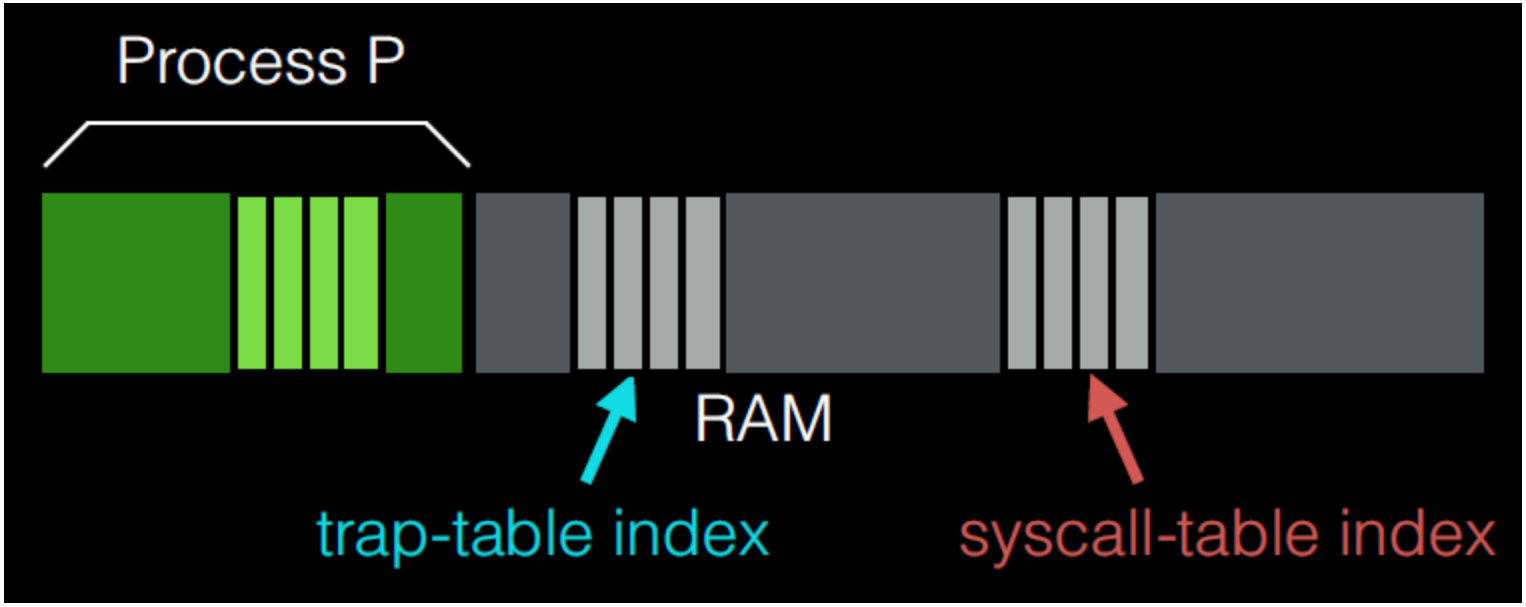
What if process tries to do something restricted?

Get HW help, put processes in “user mode”

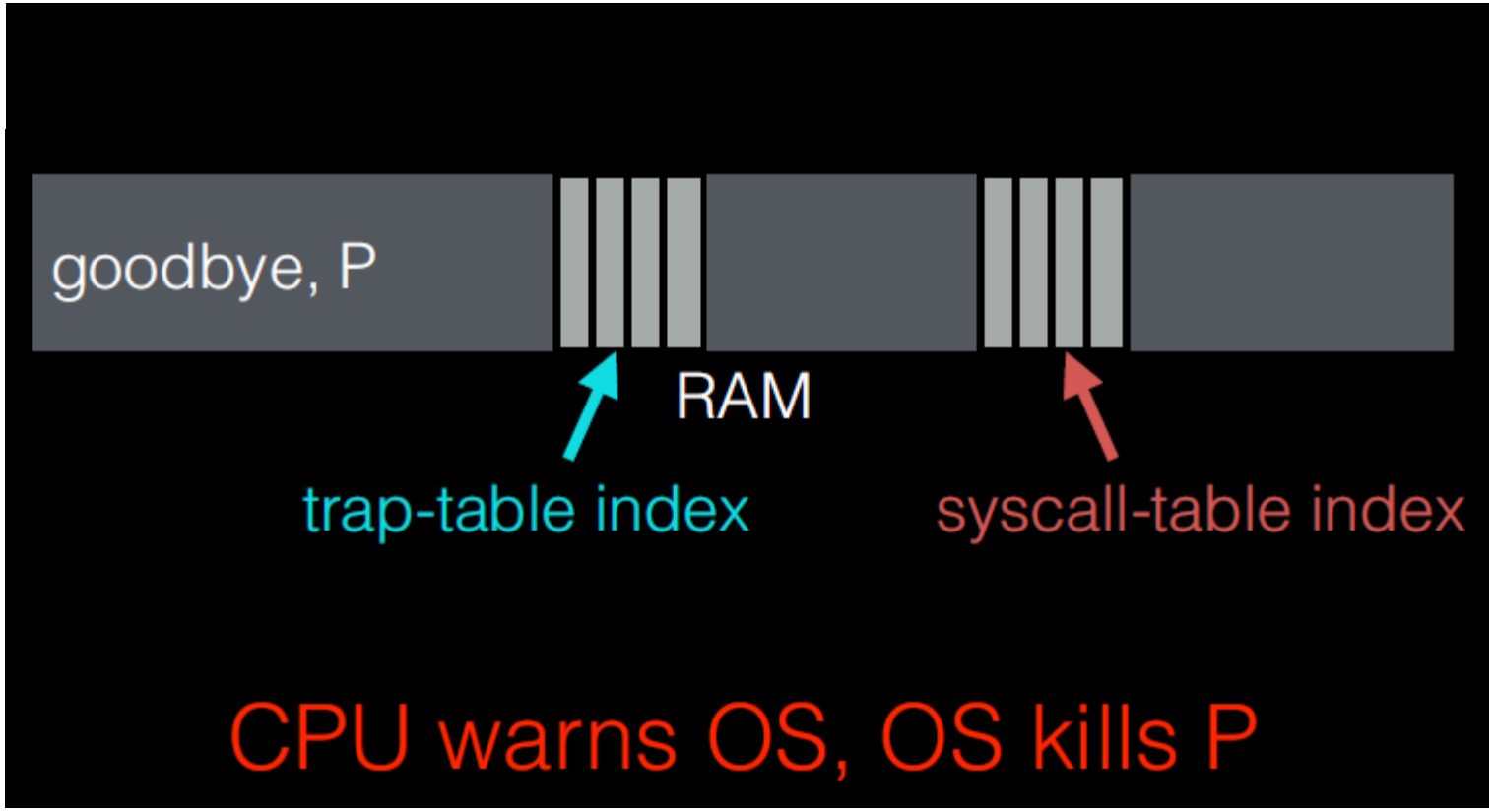
# lidt example



# lidt example(cont.)



# lidt example(cont.)



# 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

- Reoccurring 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
}
```


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

Processes periodically give up the CPU by making **system calls** such as `yield`.

- The OS decides to run some other task.
- Application also transfer control to the OS when they do something illegal.
  - Divide by zero
  - Try to access memory that it shouldn't be able to access

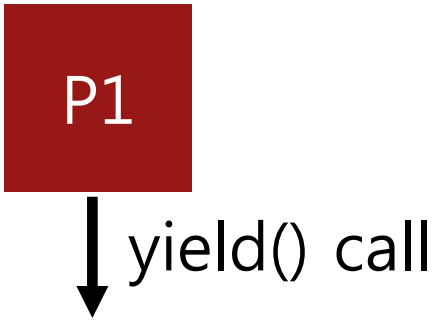
- Ex) Early versions of the Macintosh OS, The old Xerox Alto system

**A process gets stuck in an infinite loop.**
→ **Reboot the machine**

# A cooperative Approach: Wait for system calls

Switch contexts for syscall interrupt.

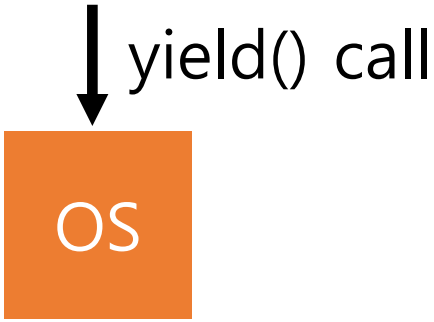
Provide special yield() system call.



# A cooperative Approach: Wait for system calls

Switch contexts for syscall interrupt.

Provide special yield() system call.



# A cooperative Approach: Wait for system calls

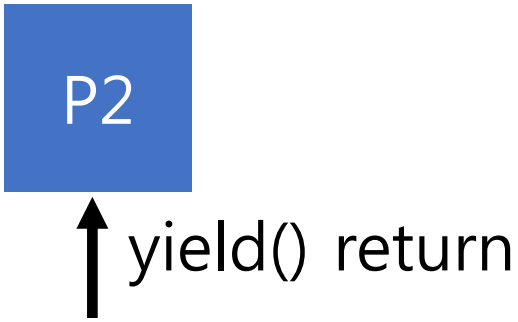
Switch contexts for syscall interrupt.

Provide special yield() system call.

# A cooperative Approach: Wait for system calls

Switch contexts for syscall interrupt.

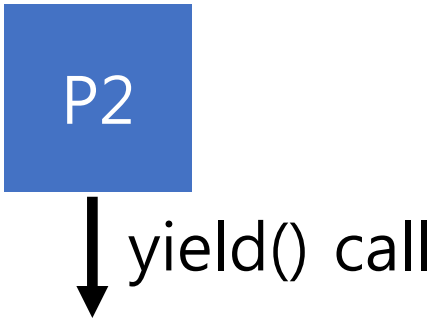
Provide special yield() system call.



# A cooperative Approach: Wait for system calls

Switch contexts for syscall interrupt.

Provide special yield() system call.



# A Non-Cooperative Approach: OS Takes Control

## A timer interrupt

- During the boot sequence, the OS start the timer.
- The timer raise an interrupt every so many milliseconds.
- When the interrupt is raised :
  - The currently running process is halted.
  - Save enough of the state of the program
  - A pre-configured interrupt handler in the OS runs.

**A timer interrupt gives OS the ability to run again on a CPU.**

# A Non-Cooperative Approach: OS Takes Control(CONT.)

Switch contexts on timer interrupt.

Set up before running any processes.

HW does not let processes prevent this.

*Is it better to be cooperative or non-cooperative?*

# Saving and Restoring Context

Scheduler makes a decision:

- Whether to continue running the **current process**, or switch to a **different one**.
- If the decision is made to switch, the OS executes context switch.

# Context Switch

A low-level piece of assembly code

- **Save a few register values** for the current process onto its kernel stack
  - General purpose registers
  - PC
  - kernel stack pointer
- **Restore a few** for the soon-to-be-executing process from its kernel stack
- **Switch to the kernel stack** for the soon-to-be-executing process

# Limited Direction Execution Protocol (Timer interrupt)

| OS @ boot (kernel mode) | Hardware |
|---|---|
| **initialize trap table** | |
| | remember address of …<br>syscall handler<br>timer handler |
| **start interrupt timer** | |
| | start timer<br>interrupt CPU in X ms |

| OS @ run (kernel mode) | Hardware | Program (user mode) |
|---|---|---|
| | | Process A<br>… |
| | **timer interrupt**<br>save regs(A) to k-stack(A)<br>move to kernel mode<br>jump to trap handler | |

# Limited Direction Execution Protocol (Timer interrupt)

| OS @ run (kernel mode) | Hardware | Program (user mode) |
|---|---|---|
| | *(Cont.)* | |
| Handle the trap<br>Call switch() routine<br>save regs(A) to proc-struct(A)<br>restore regs(B) from proc-struct(B)<br>switch to k-stack(B)<br>**return-from-trap (into B)** | | |
| | restore regs(B) from k-stack(B)<br>move to user mode<br>jump to B's PC | |
| | | Process B<br>… |

# The xv6 Context Switch Code

```
1  # void swtch(struct context **old, struct context *new);
2  #
3  # Save current register context in old
4  # and then load register context from new.
5  .globl swtch
6  swtch:
7          # Save old registers
8          movl 4(%esp), %eax            # put old ptr into eax
9          popl 0(%eax)                  # save the old IP
10         movl %esp, 4(%eax)            # and stack
11         movl %ebx, 8(%eax)            # and other registers
12         movl %ecx, 12(%eax)
13         movl %edx, 16(%eax)
14         movl %esi, 20(%eax)
15         movl %edi, 24(%eax)
16         movl %ebp, 28(%eax)
17
18         # Load new registers
19         movl 4(%esp), %eax            # put new ptr into eax
20         movl 28(%eax), %ebp           # restore other registers
21         movl 24(%eax), %edi
22         movl 20(%eax), %esi
23         movl 16(%eax), %edx
24         movl 12(%eax), %ecx
25         movl 8(%eax), %ebx
26         movl 4(%eax), %esp            # stack is switched here
27         pushl 0(%eax)                 # return addr put in place
28         ret                           # finally return into new ctxt
```

# Worried About Concurrency?

What happens if, during interrupt or trap handling, another interrupt occurs?

OS handles these situations:

- **Disable interrupts** during interrupt processing
- Use a number of sophisticate **locking** schemes to protect concurrent access to internal data structures.

# 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 entir e 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 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 | |

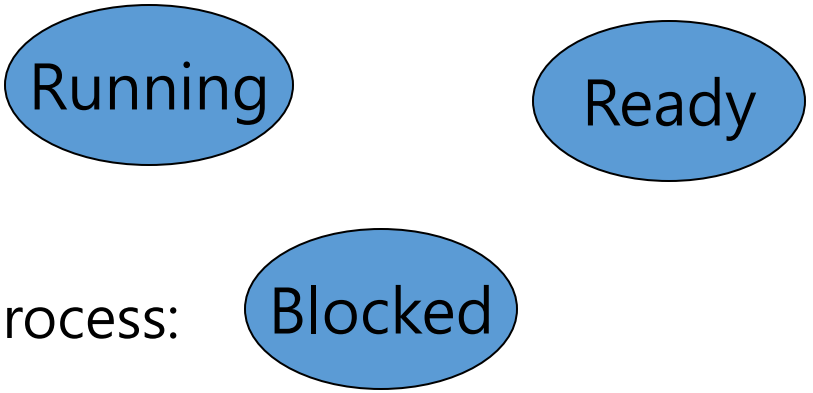
| Operating System | Hardware | Program |
| --- | --- | --- |
| | | Process A |
| | | ... |
| | timer interrupt<br>save regs(A) to k-stack(A)<br>move to kernel mode<br>jump to trap handler | |
| Handle the trap<br>Call **switch()** routine<br>save regs(A) to proc-struct(A)<br>restore regs(B) from proc-struct(B)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

# 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)




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

Transitions?

# 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

# 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