Department of Physics,
Computer Science & Engineering

CPSC 410 – Operating Systems I

# Process Description & Control

Keith Perkins

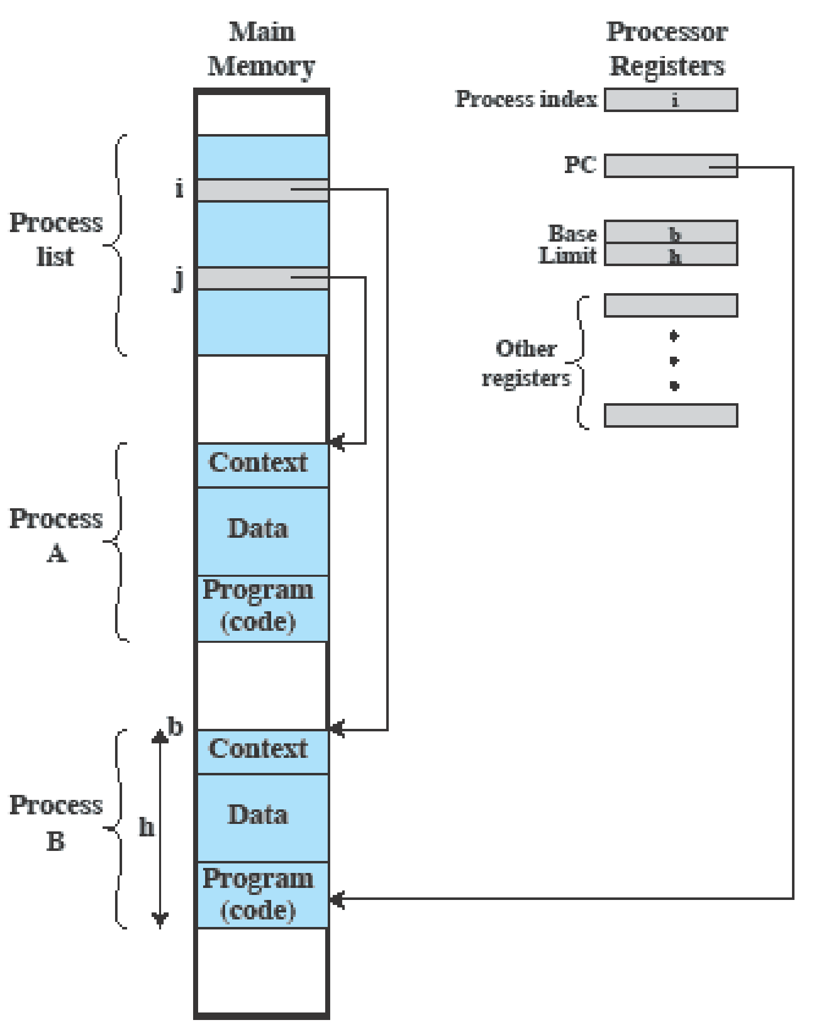
Adapted from original slides by Dr. Roberto A. Flores
Also from “CS 537 Introduction to Operating Systems” Arpaci-Dusseau

# Chapter 3 Topics

- Everything about Processes
  - Control blocks
  - States
  - Description
  - Control
- OS Execution
- Security Issues

# Revisit - Process Management

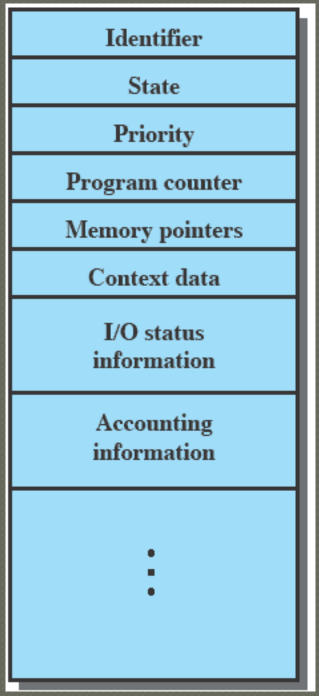
- Scheduler chooses a process to run (more later)
- Dispatcher runs it
- How?  What's in the Process List?
- BTW this list is a simplification



# Processes

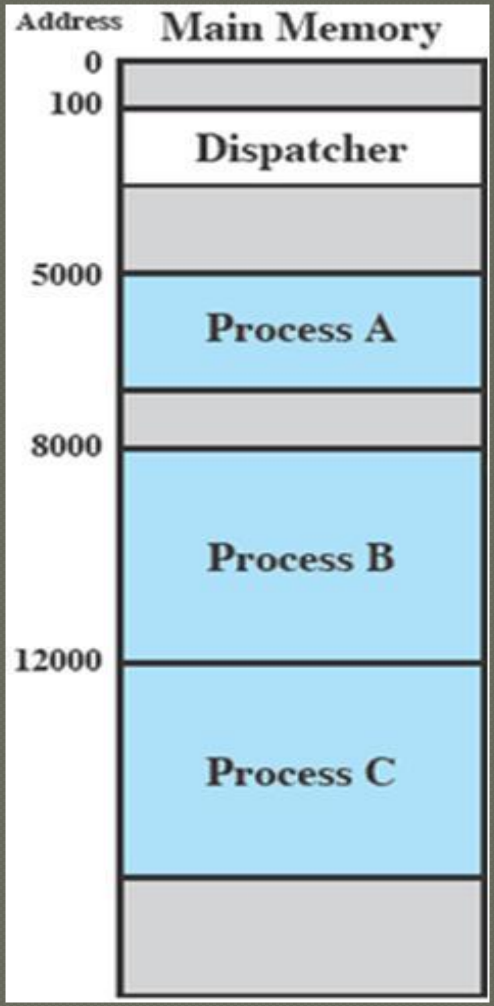
## Control Blocks

- data structure created & managed by OS
  - Identifier: unique ID
  - State: (e.g., running, blocked)
  - Priority: relative to other processes
  - Program counter: address of next instruction
  - Memory pointers: to code & data
  - I/O status: I/O in use/pending
  - Accounting: CPU time used, IDs, …
- data to hold/restore process state on interrupt/resume
  - key to support multiprocessing



# Processes

- Dispatcher
  - Program that switches processes in/out of the CPU



# Processes

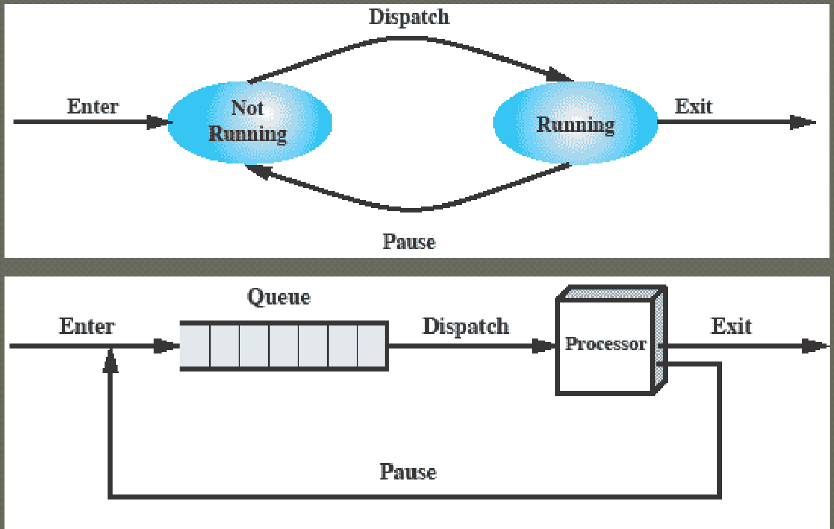
- States
  - Trace
    - Instructions executed by a process
    - In multiprogramming:
      - interleaving of instructions as processes alternate using the CPU
  - The pale blue lower right is dispatcher code
- Process switches because of Interrupts (timer, I/O)

| (a) Trace of Process A | (b) Trace of Process B | (c) Trace of Process C |
|---|---|---|
| 5000 | 8000 | 12000 |
| 5001 | 8001 | 12001 |
| 5002 | 8002 | 12002 |
| 5003 | 8003 | 12003 |
| 5004 | | 12004 |
| 5005 | | 12005 |
| 5006 | | 12006 |
| 5007 | | 12007 |
| 5008 | | 12008 |
| 5009 | | 12009 |
| 5010 | | 12010 |
| 5011 | | 12011 |

| | |
|---|---|
| 1 | 5000 |
| 2 | 5001 |
| 3 | 5002 |
| 4 | 5003 |
| 5 | 5004 |
| 6 | 5005 |
| ------------------- Timeout | |
| 7 | 100 |
| 8 | 101 |
| 9 | 102 |
| 10 | 103 |
| 11 | 104 |
| 12 | 105 |
| 13 | 8000 |
| 14 | 8001 |
| 15 | 8002 |
| 16 | 8003 |
| ---------------- I/O Request | |
| 17 | 100 |
| 18 | 101 |
| 19 | 102 |
| 20 | 103 |
| 21 | 104 |
| 22 | 105 |
| 23 | 12000 |
| 24 | 12001 |
| 25 | 12002 |
| 26 | 12003 |
| 27 | 12004 |
| 28 | 12005 |
| ------------------- Timeout | |
| 29 | 100 |
| 30 | 101 |
| 31 | 102 |
| 32 | 103 |
| 33 | 104 |
| 34 | 105 |
| 35 | 5006 |
| 36 | 5007 |
| 37 | 5008 |
| 38 | 5009 |
| 39 | 5010 |
| 40 | 5011 |
| ------------------- Timeout | |
| 41 | 100 |
| 42 | 101 |
| 43 | 102 |
| 44 | 103 |
| 45 | 104 |
| 46 | 105 |
| 47 | 12006 |
| 48 | 12007 |
| 49 | 12008 |
| 50 | 12009 |
| 51 | 12010 |
| 52 | 12011 |
| ------------------- Timeout | |

# Processes

- States (2 states)
  - One CPU
  - Round-robin (timeout)
  - Running: CPU time!
  - Not running: or not



(b) Queuing diagram

- Where do processes come from?
- When do they stop?

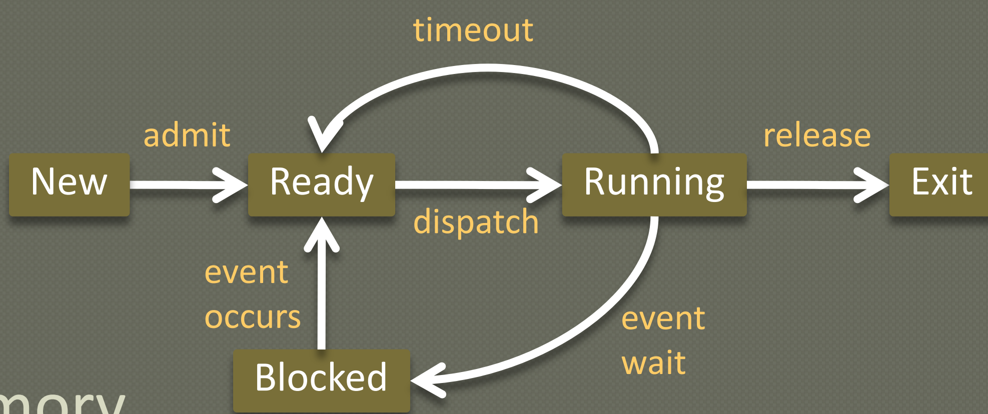
# Processes

- Where do processes come from? (start)
  - New batch job: Next job in the incoming batch stream
  - Interactive logon: User in terminal logs in
  - OS service: OS-provided service (e.g., print spooler)
  - Spawned by process: uses parallelism (parent spawns child)

- When do they end? (termination)
  - Normal
    - Job finishes, user logs off, OS shutting down, etc.
  - Abnormal
    - Timeout: running too long
    - Resource error: out of memory, I/O device unresponsive, deadlock
    - Runtime error: arithmetic operation, uninitialized variable
    - Authorization error: memory out of bounds, resource/instruction privilege

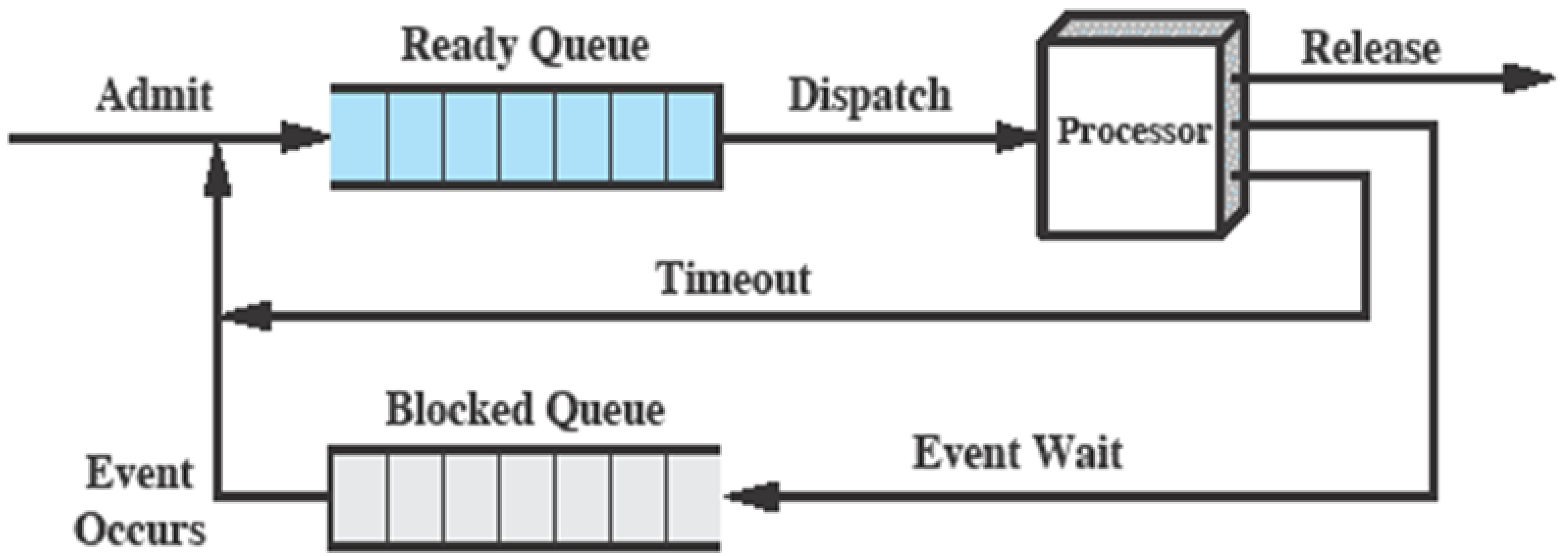
# Processes

## States (5 states)




- New: not yet in memory
- Ready: awaiting its turn
- Running: CPU time!
- Blocked: waiting for I/O
- Exit: done & gone

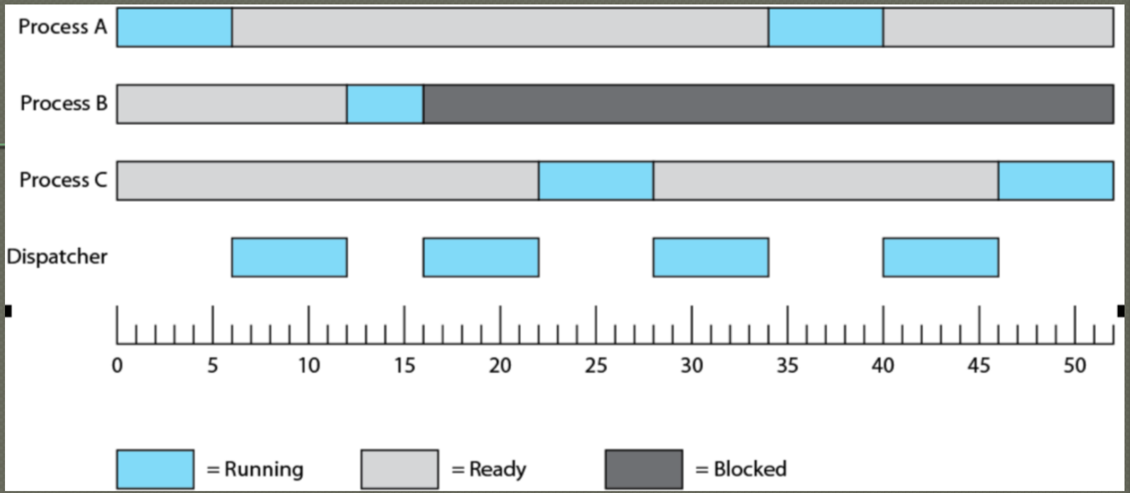
# Using Two Queues

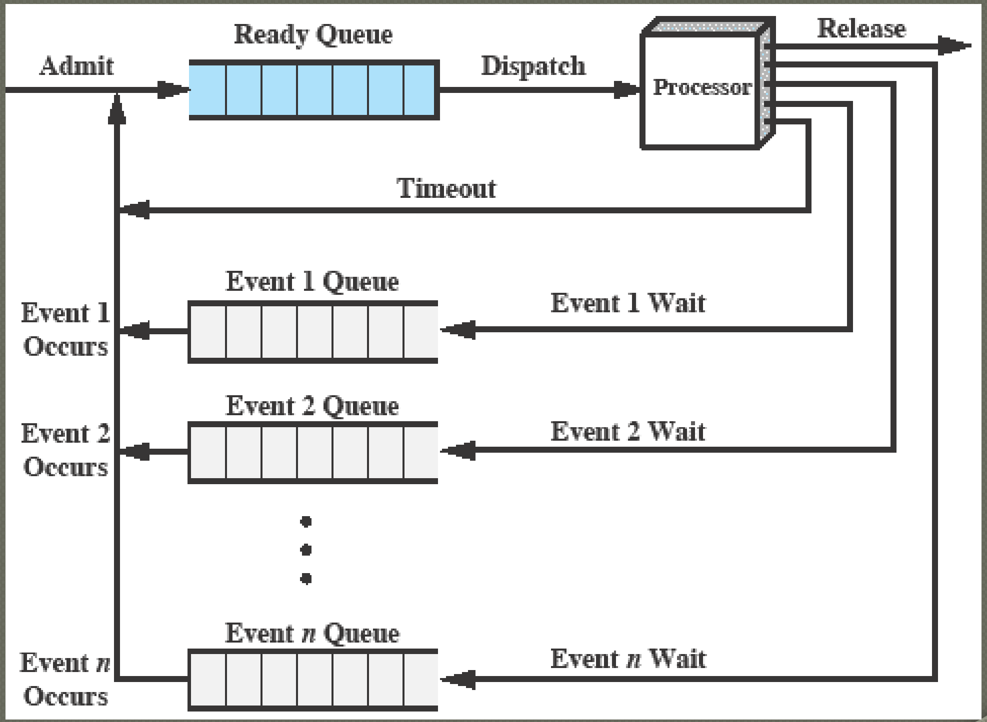


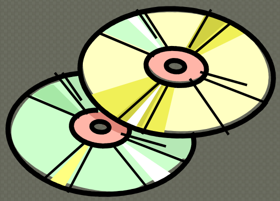
(a) Single blocked queue

Control blocks
States
Description
Control

- States (5 states)
  - e.g., Processes A, B & C
  - Multiple block queues (1 per I/O device)

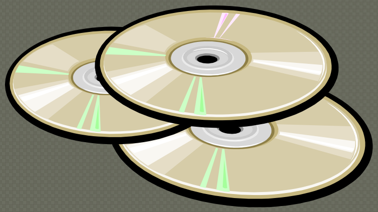




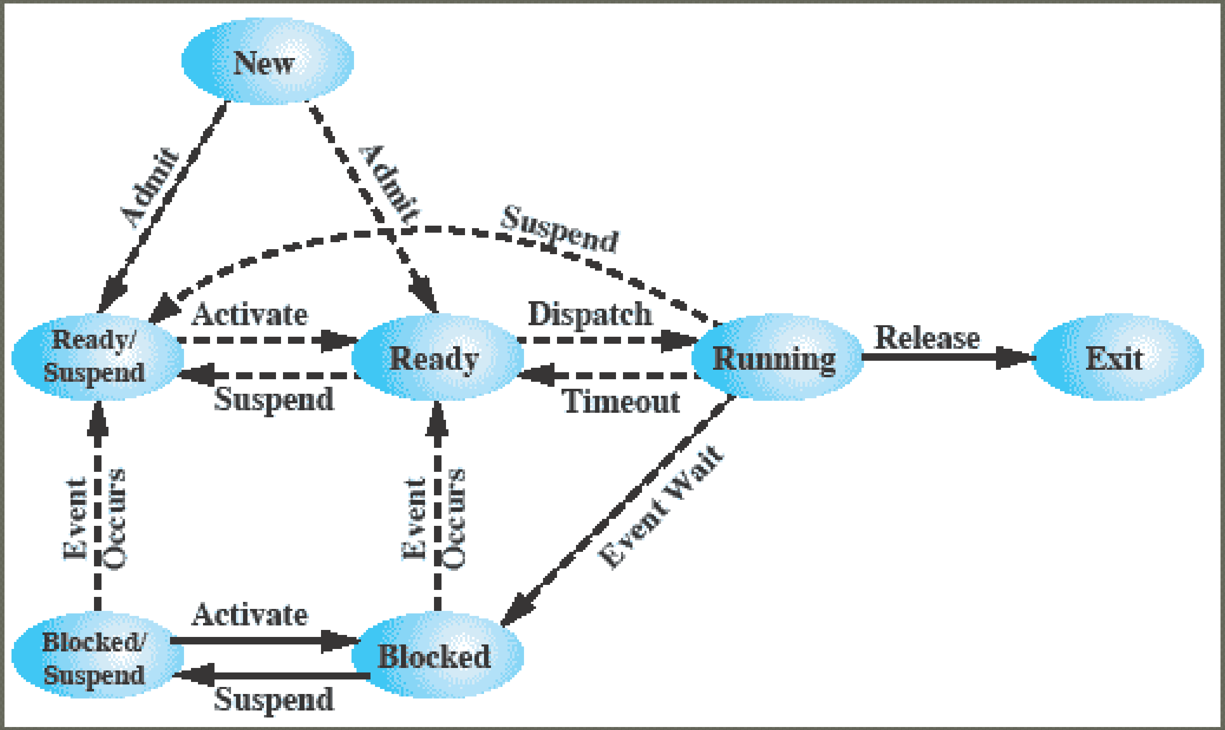
# Suspended Processes

- Swapping
  - involves moving part of all of a process from main memory to disk
  - when none of the processes in main memory is in the Ready state, the OS swaps one of the blocked processes out on to disk into a suspend queue



# Processes

- States (7 states)
  - What if not all processes fit in memory at once?
    - Suspended: when a process has been swapped to disk



# Structure of Process

## Images in Virtual Memory

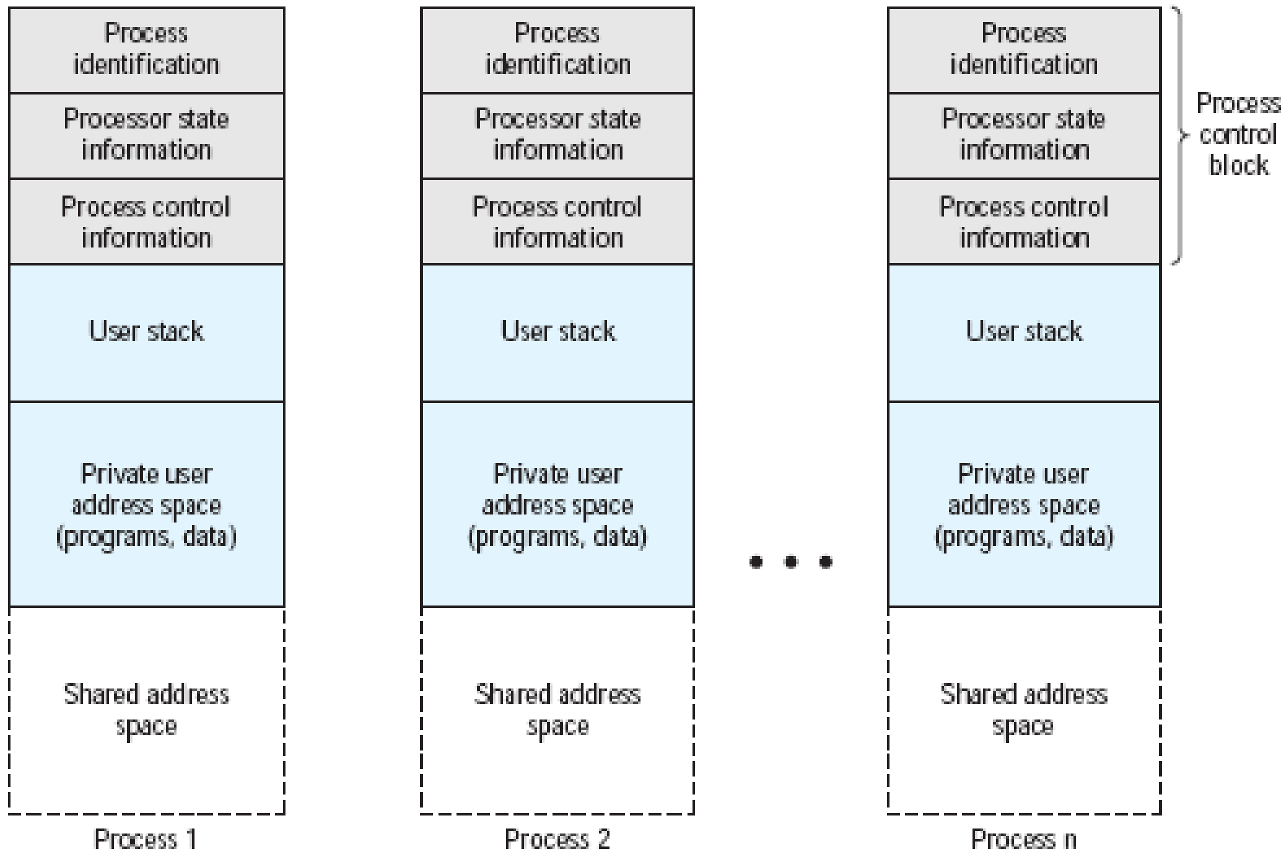



Figure 3.13 User Processes in Virtual Memory

# Process List Structures

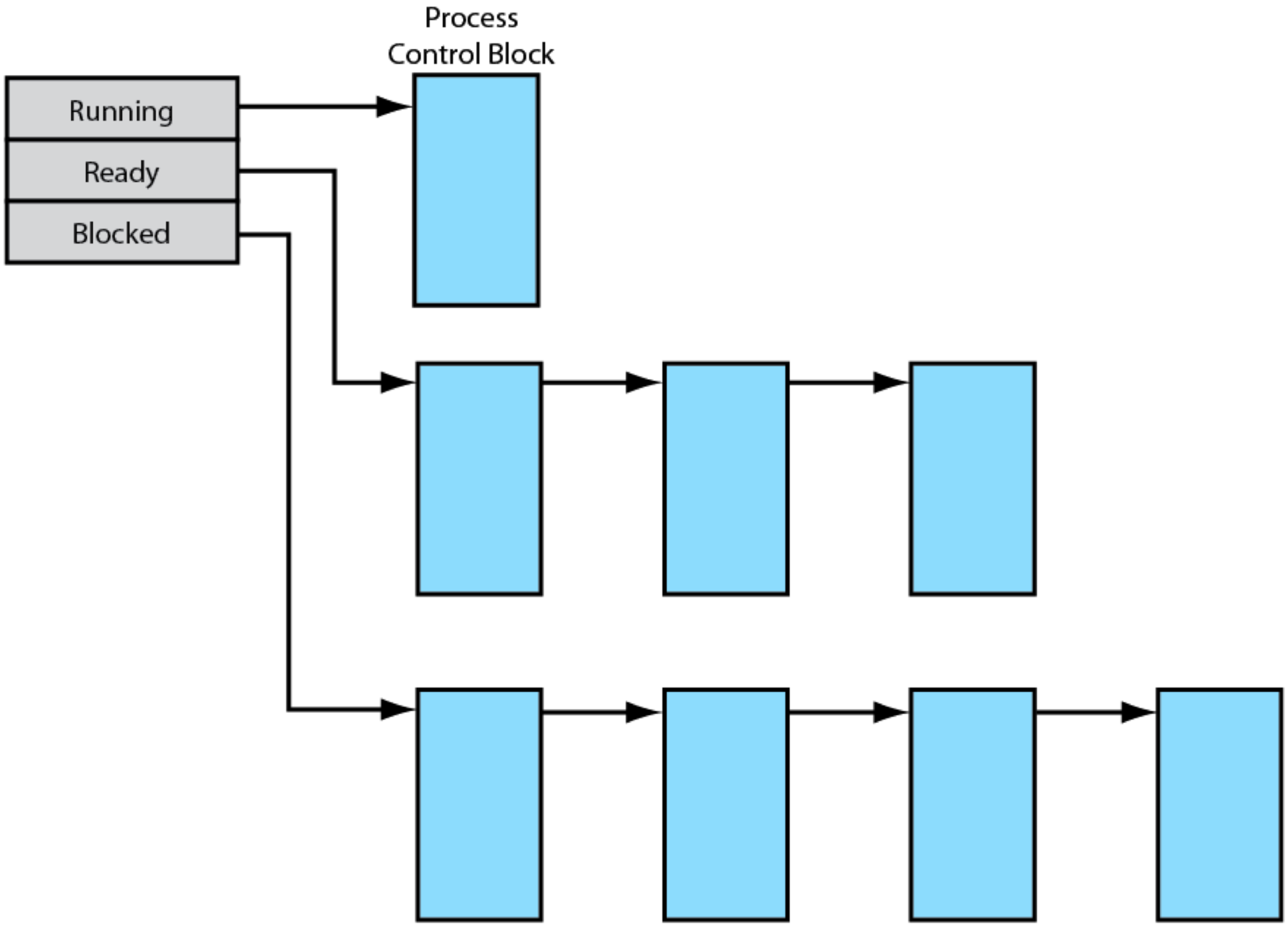



Figure 3.14 Process List Structures

Control blocks
States
Description
Control

# Processes

- Process tables
  - keep data about each process (process image)
    - user data: modifiable part of program, e.g., variables
    - user program: program to execute
    - stack: stores method calls & parameters
    - process control block (PCB): data OS uses to control process
      - process identification: process/parent/user ID
      - processor state information: user/control registers, stack pointers
      - process control information: scheduling, inter-process comms, …
  - reference (directly/indirectly) memory, I/O & file tables

# Processes

## Process tables

**Process identification**

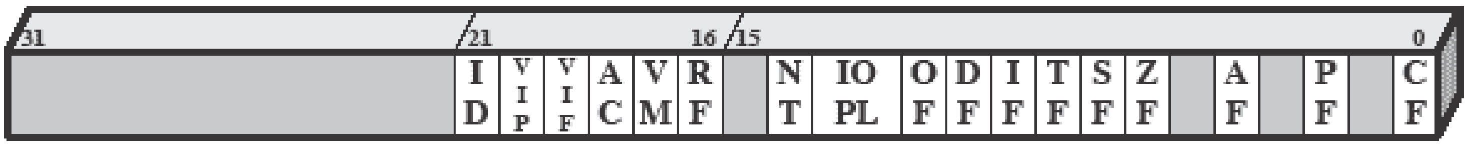
- Each process has a unique ID
- IDs are used for reference:
    - in other tables
    - in inter-process communication
    - when a parent spawns a child process

- process identification: process/parent/user ID
- processor state information: user/control registers, stack pointers
- process control information: scheduling, inter-process comms, …

- reference (directly/indirectly) memory, I/O & file tables

Process state information

- stack pointers
- user-visible registers
- control & status registers
  - program status word (PSW), e.g., EFLAGS in x86 processors




| | | | | | |
|---|---|---|---|---|---|
| ID | = | Identification flag | DF | = | Direction flag |
| VIP | = | Virtual interrupt pending | IF | = | Interrupt enable flag |
| VIF | = | Virtual interrupt flag | TF | = | Trap flag |
| AC | = | Alignment check | SF | = | Sign flag |
| VM | = | Virtual 8086 mode | ZF | = | Zero flag |
| RF | = | Resume flag | AF | = | Auxiliary carry flag |
| NT | = | Nested task flag | PF | = | Parity flag |
| IOPL | = | I/O privilege level | CF | = | Carry flag |
| OF | = | Overflow flag | | | |

S

- processor state information: user/control registers, stack pointers
- process control information: scheduling, inter-process comms, …
- reference (directly/indirectly) memory, I/O & file tables

# Processes

## Control

- Modes of execution
  - User mode (-privileged) … Kernel mode (+privileged)
  - User processes are not allowed to perform;
    - General Memory Access
    - Disk I/O
    - Special x86 instructions like lidt
    - A little background first

# Kernel Mode verses User Mode

- Can be implemented with 1 bit
- Kernel Mode
  - Most privileged
  - Access to entire file system, memory space, all hardware
  - OS runs in this mode
- User Mode
  - Least privileged
  - Access to resources (like files and memory) that belong to current user
  - User processes run in this mode

- What mode is process in?
  - Architecture typically supports:
    - Status bit in protected processor register indicating mode
    - Restricts ability to perform certain instructions if not in kernel mode

# Interrupts-types

- Types
- Hardware
  - Raised by hardware devices
  - Can occur at any time (Asynchronous)
  - Examples: timer (process switch), I/O signals
- Traps:
  - Software interrupts (Synchronous)
  - Raised by user programs to invoke OS functionality
- Exceptions
  - Generated by processor as a result of illegal action (Synchronous)
  - Faults: recoverable (page fault)
  - Aborts: difficult to recover (divide by 0)

# Processes

## Control

- Modes of execution
  - User mode (-privileged) … Kernel mode (+privileged)
  - User processes are not allowed to perform;
    - General Memory Access
    - Disk I/O
    - Special x86 instructions like lidt

# Processes

## Interrupts-IDT

So how does a process perform restricted operation (like read/write to disk)?

by trapping into the OS and running a handler in kernel mode
Address of handler is IDT + interrupt number

OS sets up an Interrupt Descriptor Table (IDT) at boot (in memory, its base pointed to by IDT register(IDTR) in CPU)
Each entry is address of an interrupt handler (known as Interrupt Service Routine ISR)
Each ISR handles a particular type of interrupt.

# Processes

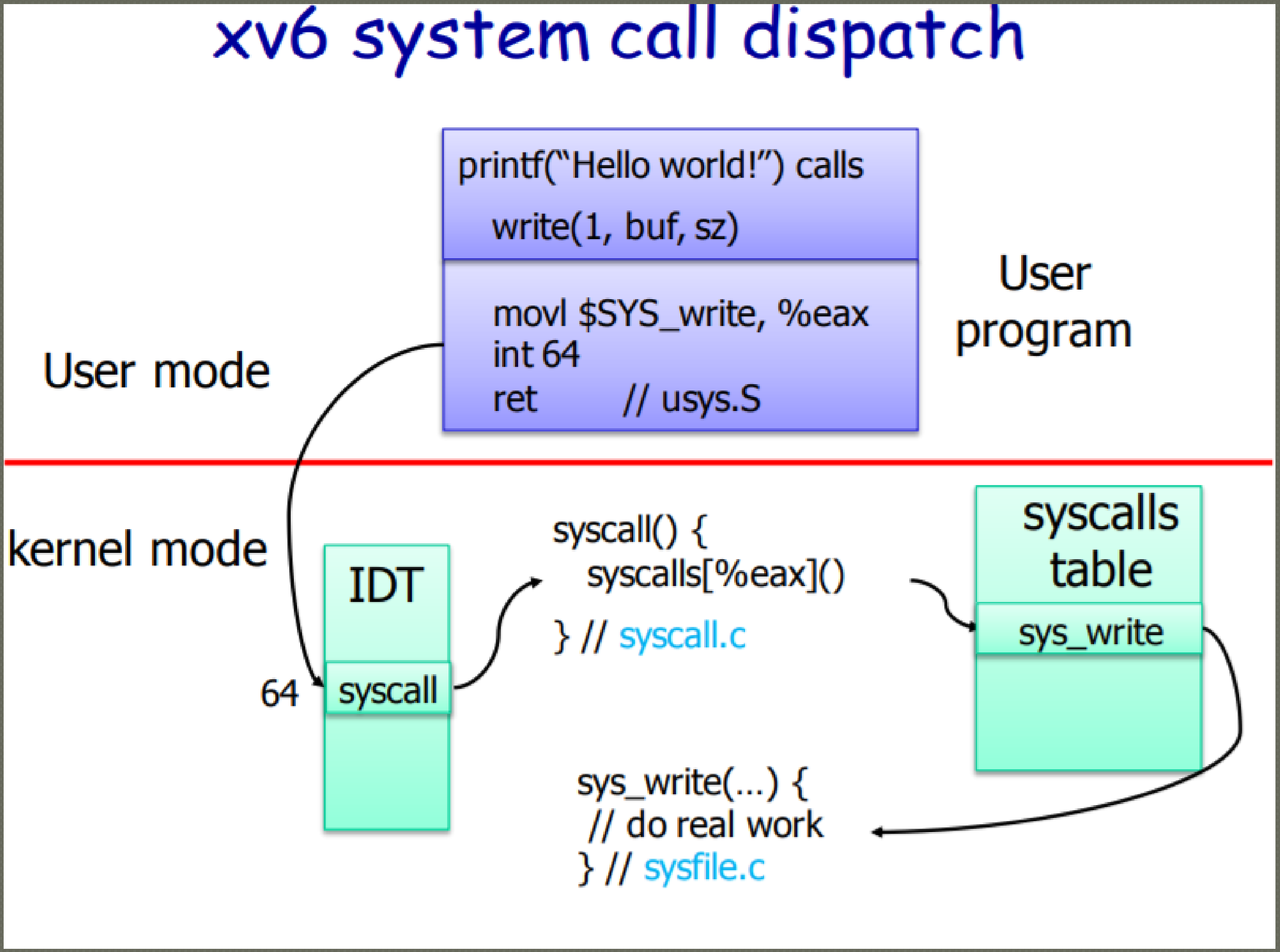
## Interrupts-IDT

### System call dispatch

1. Kernel assigns system call type a system call number
2. Kernel initializes system call table, mapping system call number to function implementing the system call
   - Also called system call vector
3. User process sets up system call number and arguments
4. User process runs int X
5. Hardware switches to kernel mode and invokes kernel's interrupt handler for X (interrupt dispatch)
6. Kernel looks up syscall table using system call number
7. Kernel invokes the corresponding function
8. Kernel returns by running iret (interrupt return)

# Processes

## Interrupts-IDT

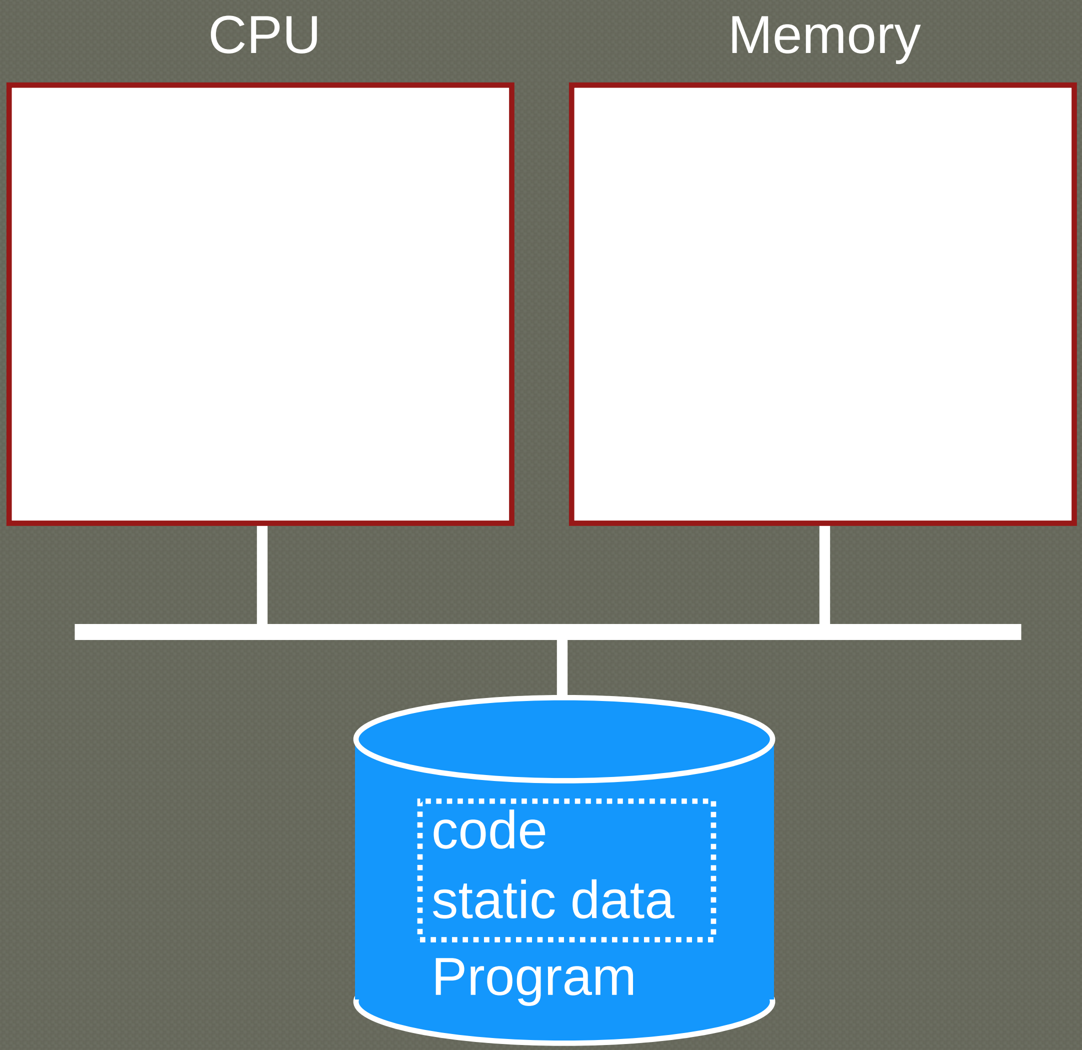


# Processes

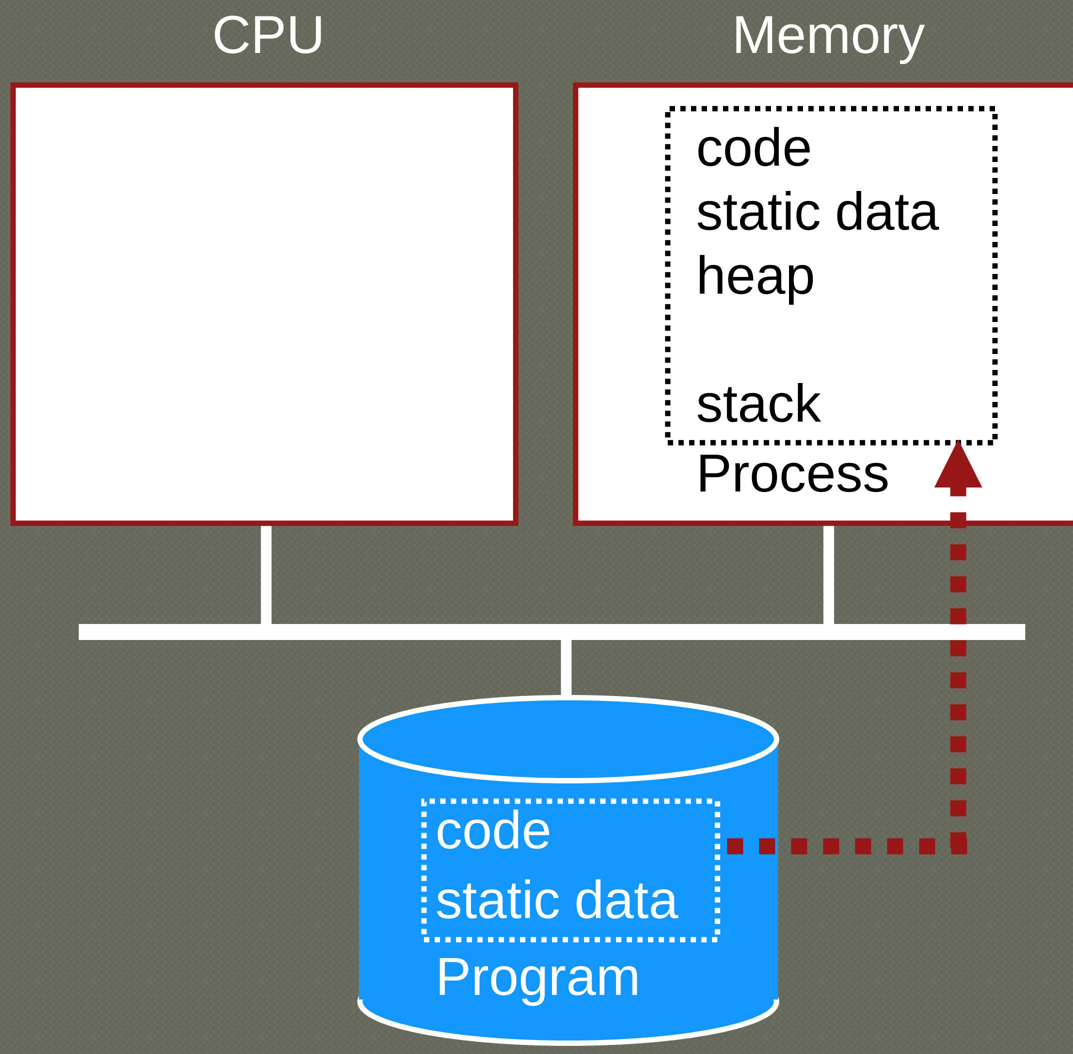
## Control

- Process creation
  - What does OS do when a process is created?
    - assigns a new unique ID
    - allocates space for the process in memory
    - initializes its process control block & sets it in place (e.g. in process list)

# Process Creation



# Process Creation



# Processes

## Dispatch Mechanism

Process is running- how to switch to other process?

# Processes
## Dispatch Mechanism

### Process is running- how to switch to other process?

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?

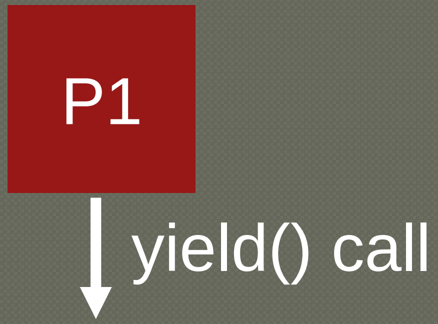
# Processes

## Q1: How does Dispatcher get CONTROL?

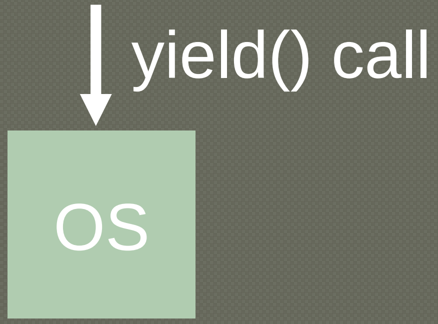
### Option 1: Cooperative Multi-tasking

- 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

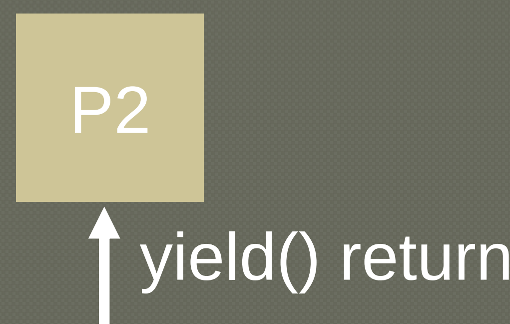


# Cooperative Approach

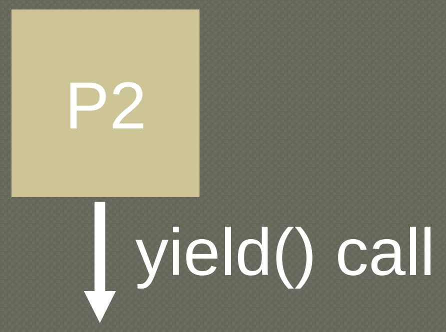


# Cooperative Approach

# Cooperative Approach



# Cooperative Approach



# Processes

## Q1: How does Dispatcher get CONTROL?

- Problem with cooperative approach? YES
- Disadvantages: Processes can misbehave
  - By avoiding all traps and performing no I/O, can take over entire machine
  - Only solution: Reboot (windows 95)!
- Not performed in modern operating systems

# Processes

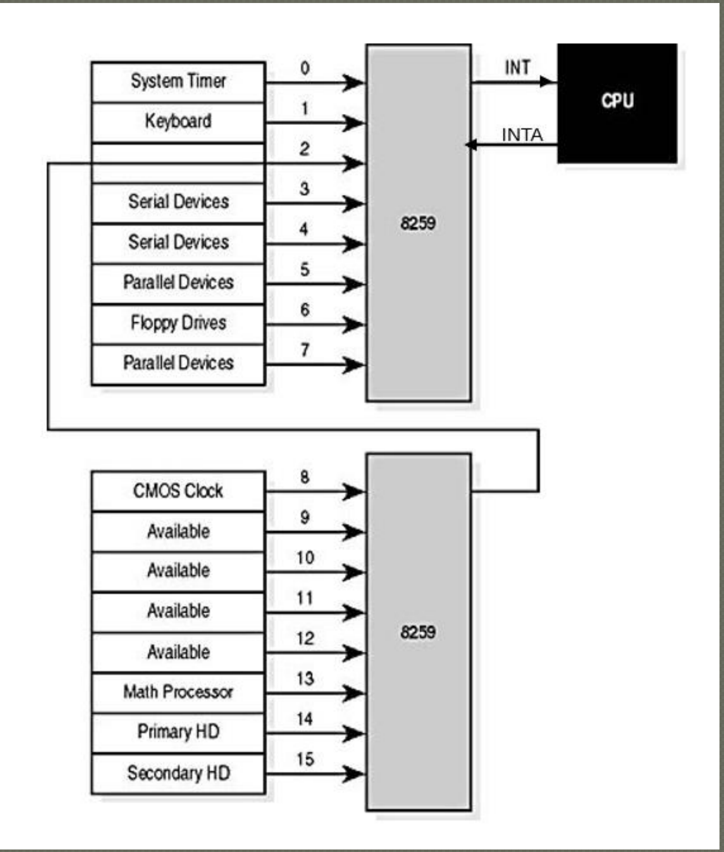
## Q1: How does Dispatcher get CONTROL?

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

# Interrupts-HW- timer example

- 8259 (Programmable interrupt controller or PIC) relays up to 8 interrupt to CPU
- Devices raise interrupts by an 'interrupt request' (IRQ)
- CPU acknowledges and queries the 8259 to determine which device interrupted (int#)
- Priorities can be assigned to each IRQ line
- 8259s can be cascaded to support more interrupts



# Processes

## What context to save?

Dispatcher must track context of process when not running

- Save context in process control block (PCB)

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<br>… |

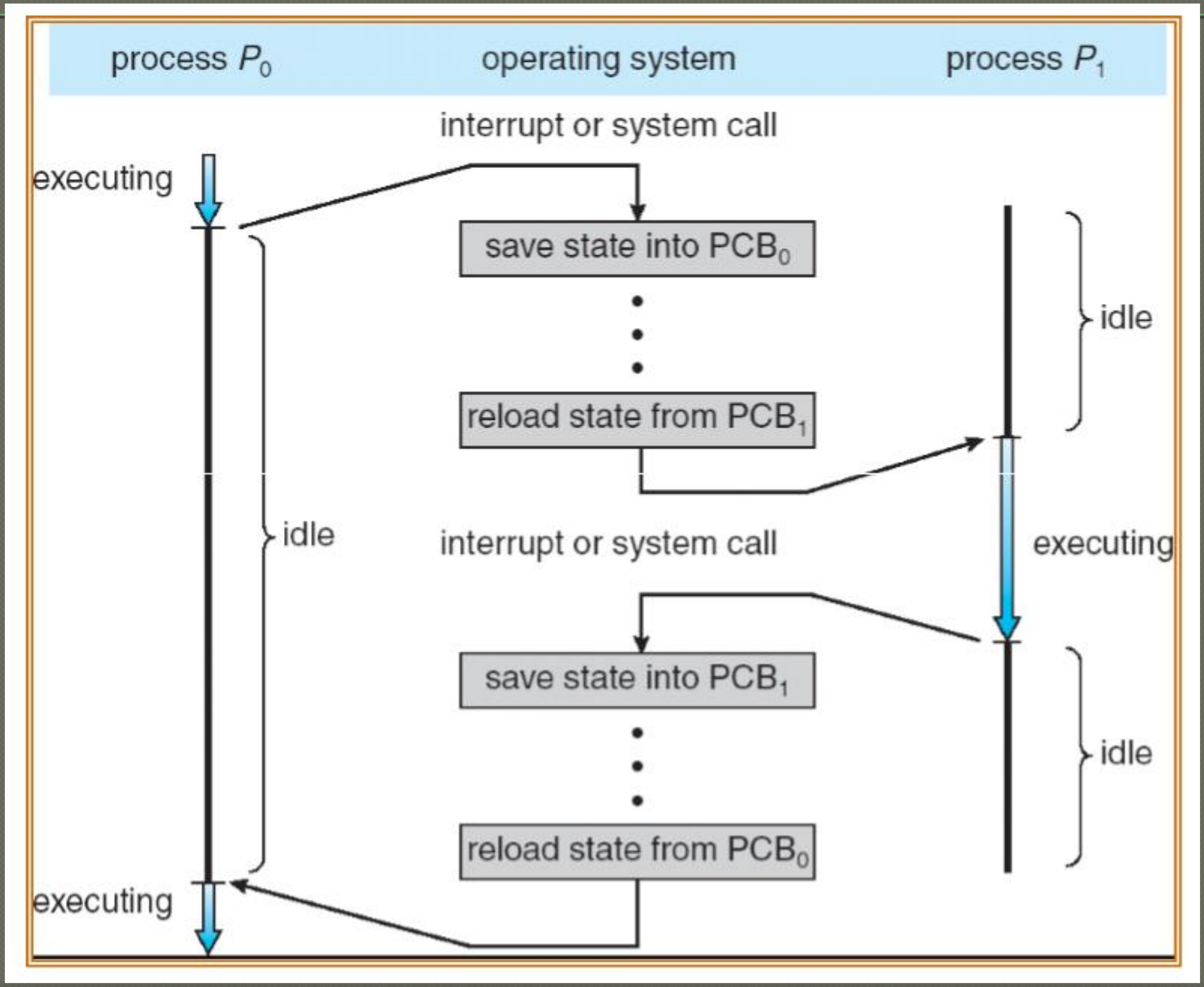
| Operating System | Hardware | Program |
|---|---|---|
| | | Process A<br>… |
| | timer interrupt<br>save regs(A) to k-stack(A)<br>move to kernel mode<br>jump to trap handler | |

| Operating System | Hardware | Program |
|---|---|---|
| | | Process A<br>… |
| | 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<br>… |
| | 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) | | |
| | restore regs(B) from k-stack(B)<br>move to user mode<br>jump to B's IP | |

| Operating System | Hardware | Program |
| --- | --- | --- |
| | | Process A<br>… |
| | 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)** | | |
| | restore regs(B) from k-stack(B)<br>move to user mode<br>jump to B's IP | |
| | | Process B<br>… |

# Interrupts



process $P_0$
operating system
process $P_1$
interrupt or system call
executing
save state into $PCB_0$
idle
reload state from $PCB_1$
idle
interrupt or system call
executing
save state into $PCB_1$
idle
reload state from $PCB_0$
executing

# Change of Process State

- The steps in a full process switch are:

save the context of the processor

update the process control block of the process currently in the Running state

move the process control block of this process to the appropriate queue

select another process for execution

update the process control block of the process selected

update memory management data structures

restore the context of the processor to that which existed at the time the selected process was last switched out

If the currently running process is to be moved to another state (Ready, Blocked, etc.), then the OS must make substantial changes in its environment

# Mode Switching

**If no interrupts are pending the processor:**

proceeds to the fetch stage and fetches the next instruction of the current program in the current process

**If an interrupt is pending the processor:**

sets the program counter to the starting address of an interrupt handler program

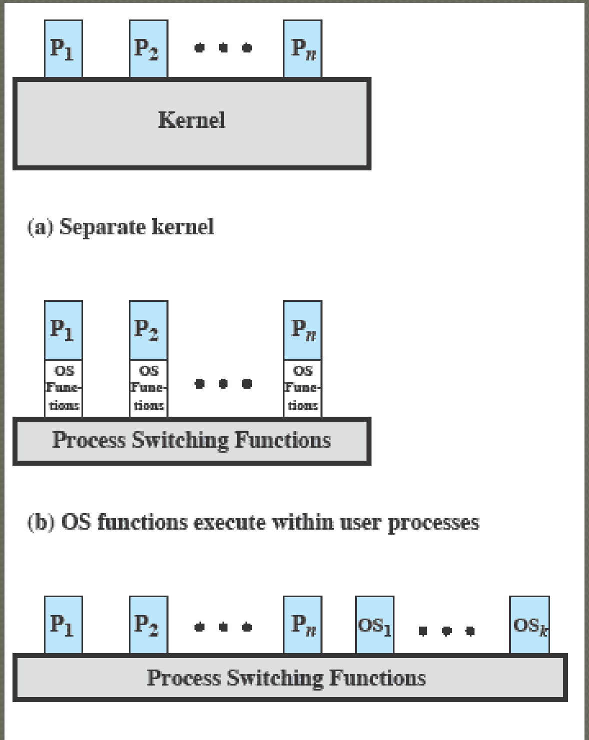
switches from user mode to kernel mode so that the interrupt processing code may include privileged instructions

# Chapter 3 Topics

- Everything about Processes
  - Elements
  - Control blocks
  - States
  - Description
  - Control
- OS Execution
- Security Issues

# OS Execution

- OS is software, right?
  - How is it different from just another process?
  - How is it controlled?



(a) Separate kernel
(b) OS functions execute within user processes
(c) OS functions execute as separate processes

a) Non-process Kernel
  - Processes are processes.
    The kernel is the kernel.

b) Execution within user processes
  - OS is a bare process
    switching mechanism
  - OS routines are linked to
    user programs (OS data is shared)

c) Process-based OS
  - OS routines run as independent processes
  - Modular approach for parallelism (e.g., OS in one CPU, user processes in another)

# Execution *Within* User Processes

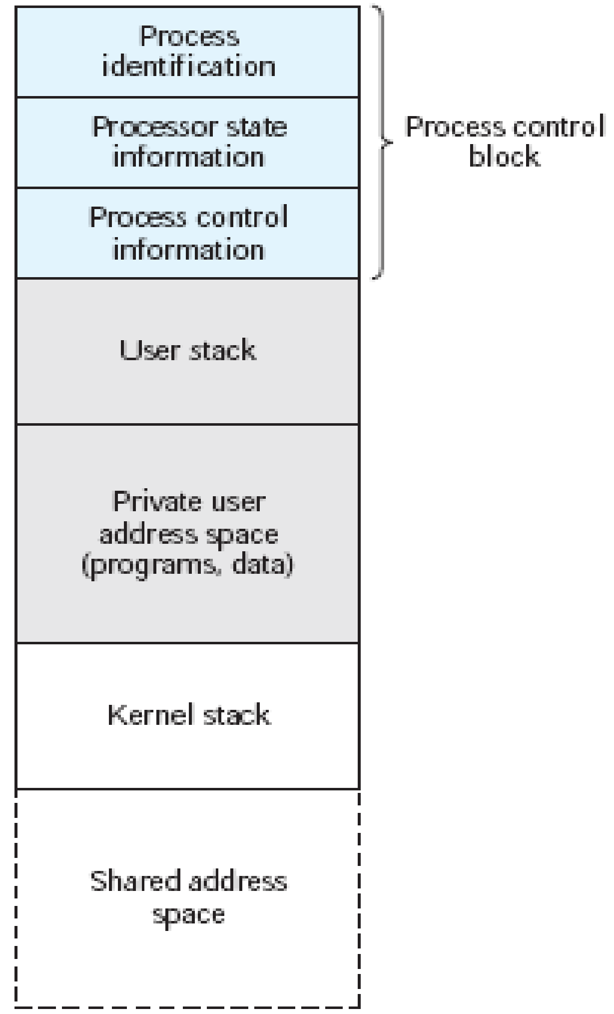



**Figure 3.16** **Process Image: Operating System Executes within User Space**

# Chapter 3 Topics

- Everything about Processes
  - Elements
  - Control blocks
  - States
  - Description
  - Control
- OS Execution
- Security Issues

# Security

- Protecting computer resources
  - OS should prevent (or at least detect) users/malware attempts to gain unauthorized access
  - Privileges
    - Users have privilege levels (highest: administrator/root)
    - Processes have (at most) the same privilege as their user

- Threats
  - A potential violation of security, given a circumstance/capability/action/event breaching security and causing harm.
- Countermeasures
  - An action/technique that eliminates/prevents/minimizes/reports a threat.

[RFC 4949, 2007, http://tools.ietf.org/html/rfc4949]

# Security

- Threats
  - Goal: gain access to / increase privileges in system
  - Intruders (hacker | cracker)
    - Misfeasor: user seeking more than allowed | misusing resources
    - Masquerader: non-user posing as legitimate user
    - Clandestine user: (non-) user seeking root privilege
  - Malicious software (malware)
    - Sophisticated (harmless -> crippling)
    - Parasitic (needs host program)
      - virus: self-replicating code embedded into another program
      - logic bomb: routine activated under certain conditions
      - backdoor: non-regular access to system (left by designers)
    - Independent: worm (virus-minus-host)

# Security

- Countermeasures
  - Intrusion detection
    - Service monitoring system events, warning about attempts to access resources in an unauthorized manner.
    - 3 logical components
      - sensing >> analyzing >> reporting (UI)
  - Authentication
    - Process of verifying an identity claimed by a system entity.
      - Identification: representative token
      - Verification: examining token
  - Firewalls
    - Computer controlling network traffic (based on policies)

# Chapter 3 Topics

- Everything about Processes
  - Elements
  - Control blocks
  - States
  - Description
  - Control
- OS Execution
- Security Issues

Done!