OPERATING SYSTEMS

Introduction, Computer System Organization

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Slides Credits for all the PPTs of this course

- The slides/diagrams in this course are an **adaptation**, **combination**, and **enhancement** of material from the following resources and persons:

1. Slides of Operating System Concepts, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne - 9th edition 2013 and some slides from 10th edition 2018
2. Some conceptual text and diagram from Operating Systems - Internals and Design Principles, William Stallings, 9th edition 2018
3. Some presentation transcripts from A. Frank – P. Weisberg
4. Some conceptual text from Operating Systems: Three Easy Pieces, Remzi Arpaci-Dusseau, Andrea Arpaci Dusseau

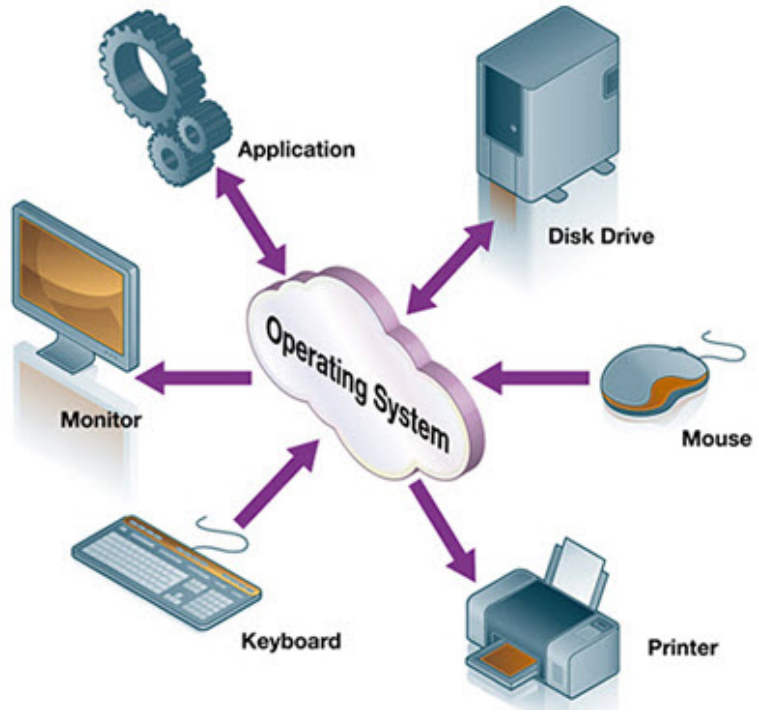
OPERATING SYSTEMS

Introduction, Computer System Organization

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Need for Operating System

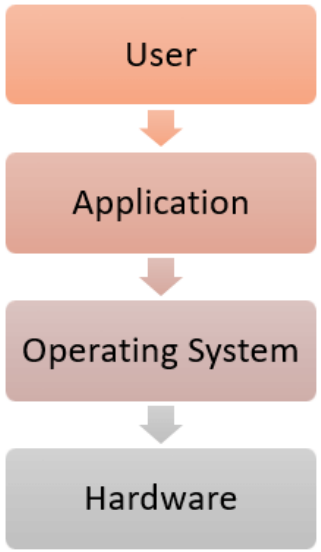


If OS is not developed then how will the application developers access the hardware?

OPERATING SYSTEMS

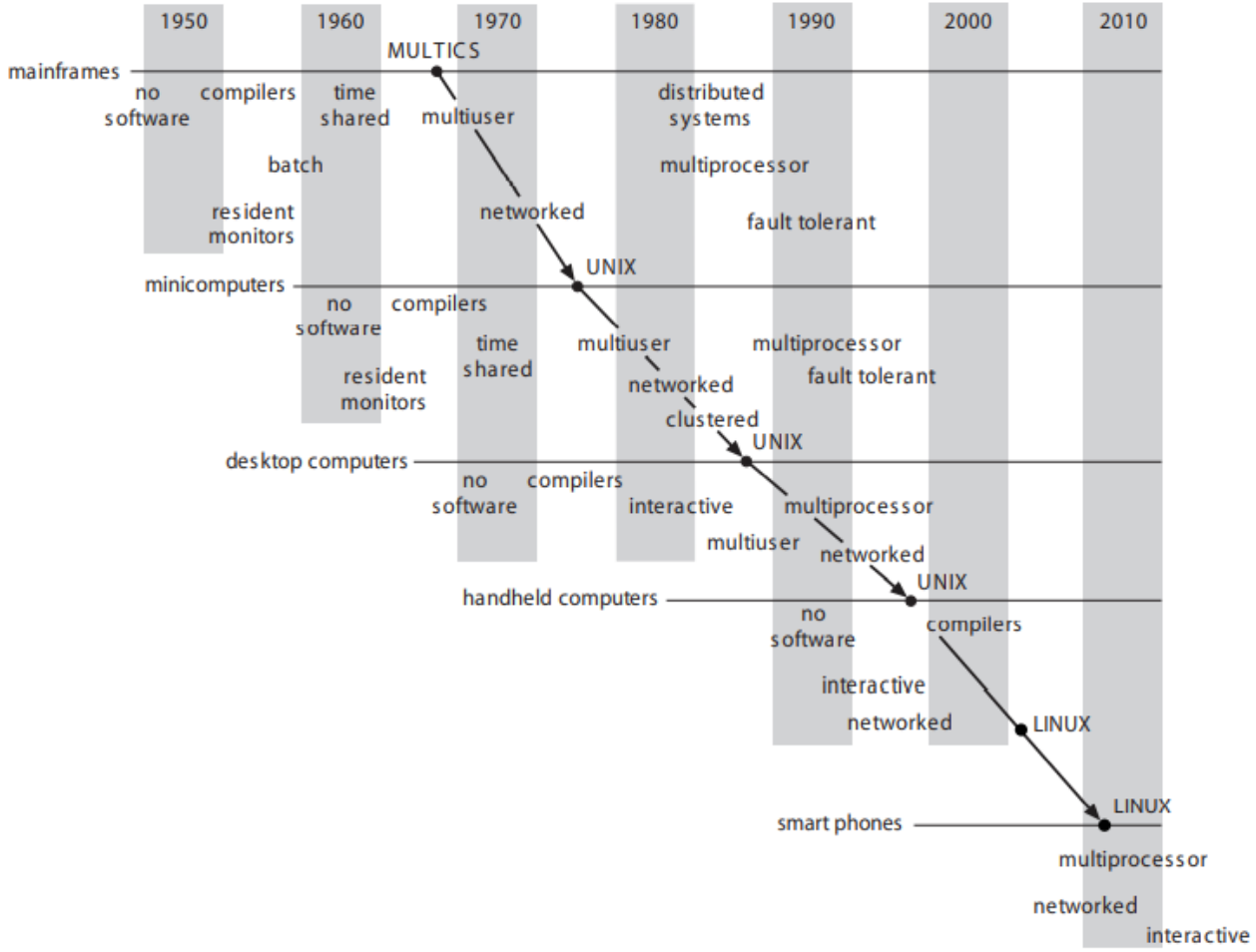
General Definition

- An Operating System is a program that acts as an intermediary between a user of a computer and the computer hardware
- It provides a user-friendly environment in which a user may easily develop and execute programs. Otherwise, hardware knowledge would be mandatory for computer programming.



OPERATING SYSTEMS

Genesis



Operating System Goals

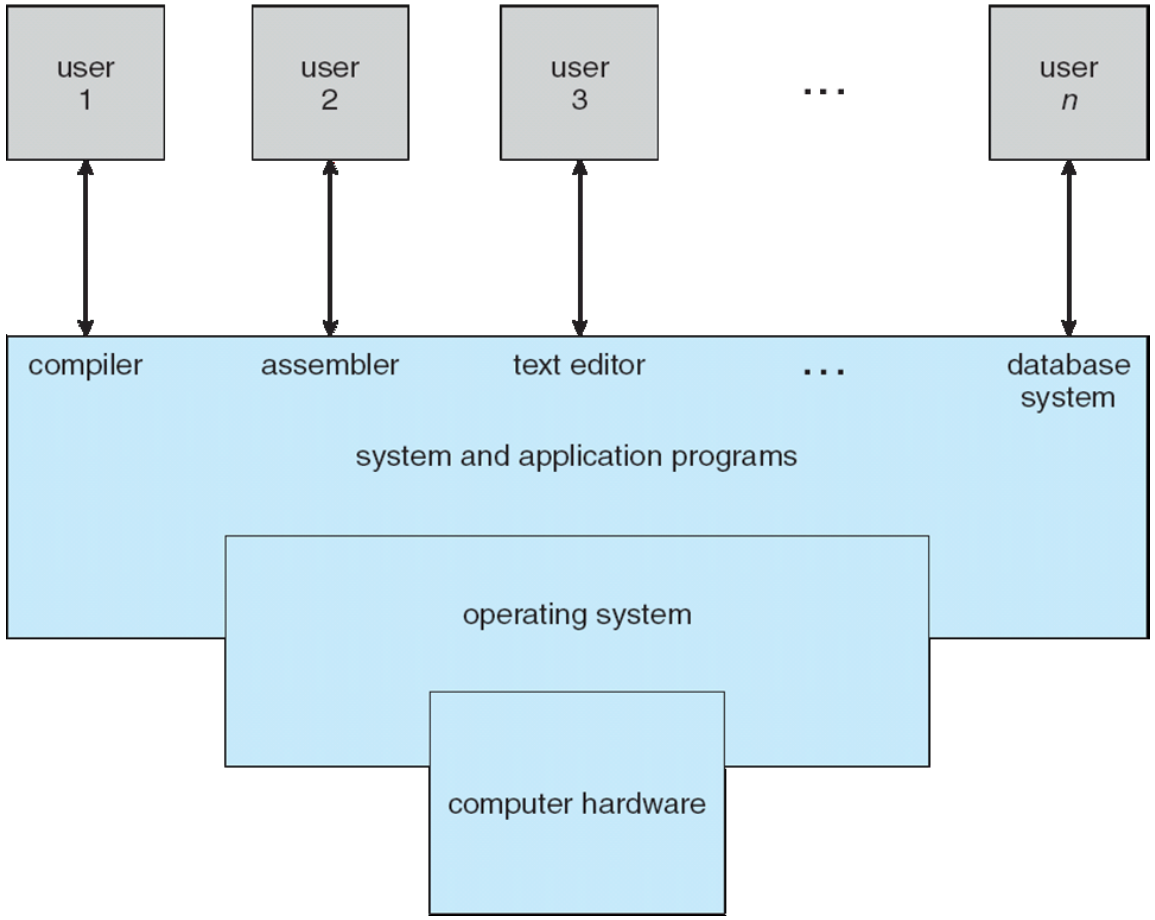
- Execute user programs and make solving user problems easier
- Make the computer system convenient to use
- Use the computer hardware in an efficient manner
- Manage resources such as
  - Memory
  - Processor(s)
  - I/O Devices

Why Study Operating Systems?

- Only a small percentage of computer practitioners will be involved in the creation or modification of an operating system.
- However almost all code runs on top of an operating system, and thus knowledge of how operating systems work is crucial to proper, efficient, effective, and secure programming.
- Understanding the fundamentals of operating systems

OPERATING SYSTEMS

Four Components of a Computer System (Abstract view)



OPERATING SYSTEMS

Computer System Structure

- Hardware – provides basic computing resources
  - CPU, memory, I/O devices
- Operating system
  - Controls and coordinates use of hardware among various applications and users
- Application programs – define the ways in which the system resources are used to solve the computing problems of the users
  - Word processors, compilers, web browsers, database systems, video games
- Users
  - People, machines, other computers

What Operating Systems Do

- Depends on the point of view user and system
- Users want convenience, **ease of use** and **good performance**
  - Don’t care about **resource utilization**
- But shared computer such as **mainframe** or **minicomputer** must keep all users happy.
  - Maximize resource utilization.
  - Available CPU time, memory, and I/O are used efficiently and that no individual user takes more than her fair share

What Operating Systems Do

- Users of dedicated systems such as workstations have dedicated resources but frequently use shared resources from servers.
  - resources like file, compute, and print servers are shared.
  - operating system is designed to compromise between individual usability and resource utilization
- Handheld computers are resource poor, optimized for usability and battery life.
- Some computers have little or no user interface, such as embedded computers in devices and automobiles

OPERATING SYSTEMS

System View

- OS is a **resource allocator**
  - Manages all resources
  - Decides between conflicting requests for efficient and fair resource use
- OS is a **control program**
  - Controls execution of programs to prevent errors and improper use of the computer

OPERATING SYSTEMS

Defining Operating System

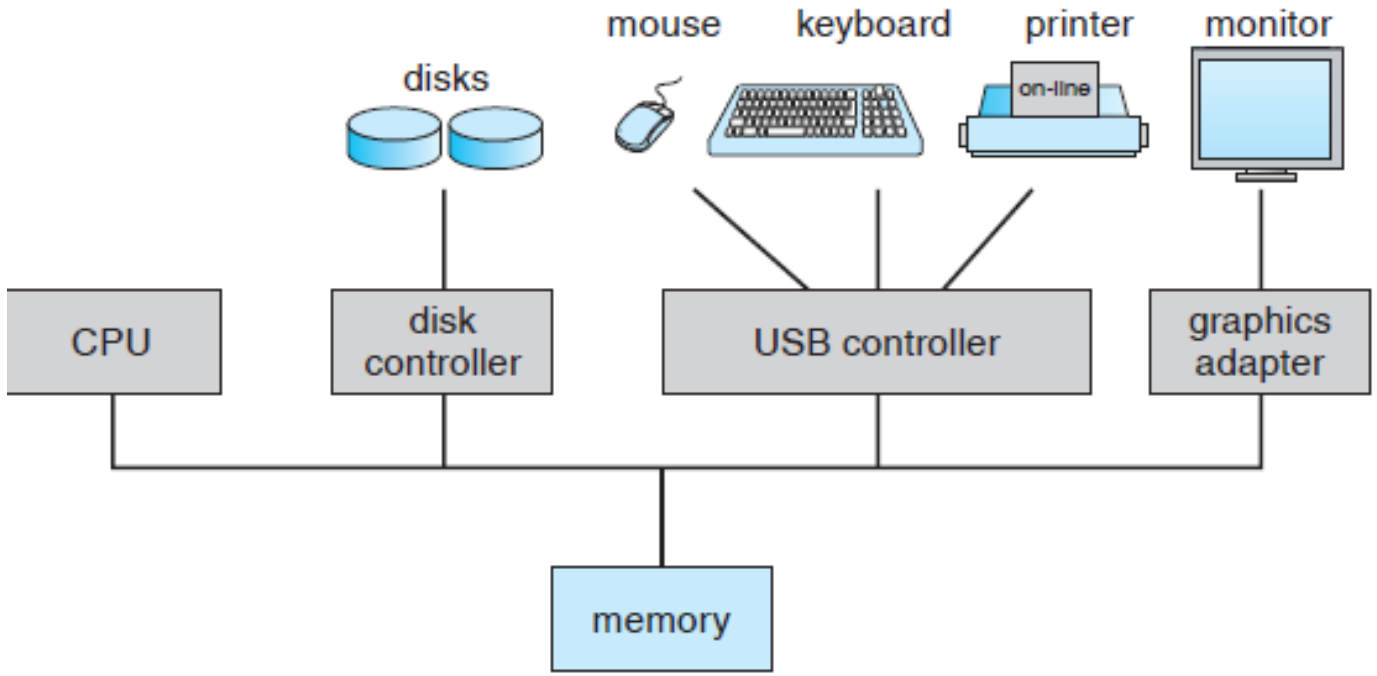
- The OS has many roles and functions
- The fundamental goal of computer systems is to execute user programs and to make solving user problems easier.
- The common functions of controlling and allocating resources are then brought together into one piece of software: the **operating system**
- “The one program running at all times on the computer” is the **kernel**.
- Everything else is either
  - a system program (ships with the operating system) , or
  - an application program.

Computer System Organization

- Computer-system consists of,
  - One or more CPUs, device controllers connect through common bus providing access to shared memory
  - The CPU and the device controllers can execute concurrently, competing for memory cycles.
    - memory controller is provided to synchronize access to the memory.

OPERATING SYSTEMS

Computer System Organization



A modern computer system

OPERATING SYSTEMS

Computer System Operation

- I/O devices and the CPU can execute concurrently
- Each device controller is in charge of a particular device type
- Each device controller has a local buffer
- Each device controller has registers for action (like “read character from keyboard”) to take
- CPU moves data from/to main memory to/from local buffers
- I/O is from the device to local buffer of controller
- Device controller informs CPU that it has finished its operation by causing an interrupt

OPERATING SYSTEMS

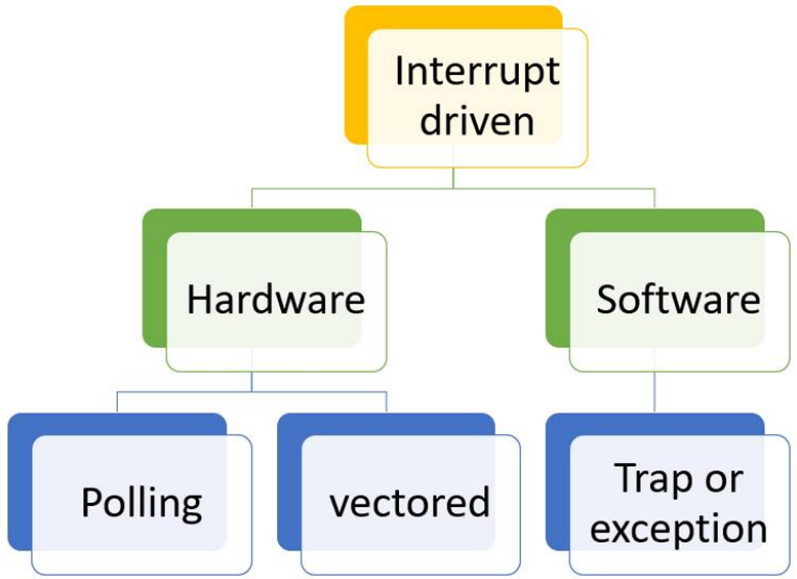
Computer System Organization

- When the system is booted, the first program that starts running is a Boostrap.
- It is stored in read-only memory (ROM) or electrically erasable programmable read-only memory (EEPROM).
- Bootstrap is known by the general term firmware, within the computer hardware.
- It initializes all aspects of the system, from CPU registers to device controllers to memory contents.
- The bootstrap program must know how to load the operating system and how to start executing that system.
- The bootstrap program must locate and load into memory the operating system kernel.
- The first program that is created is **init**, after the OS is booted. It waits for the occurrence of event.

OPERATING SYSTEMS

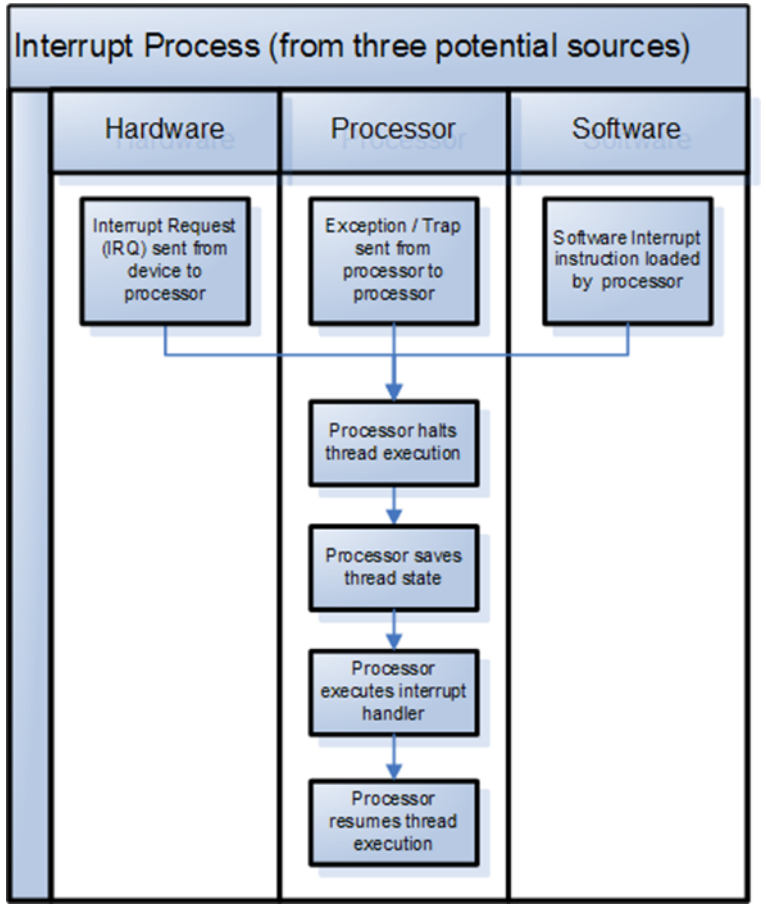
Common Functions of Interrupts

- An operating system is **interrupt driven**
- Interrupt transfers control to the interrupt service routine generally, through the **interrupt vector**, which contains the addresses of all the service routines
- Interrupt architecture must save the address of the interrupted instruction
- A **trap** or **exception** is a software-generated interrupt caused either by an error or a user request

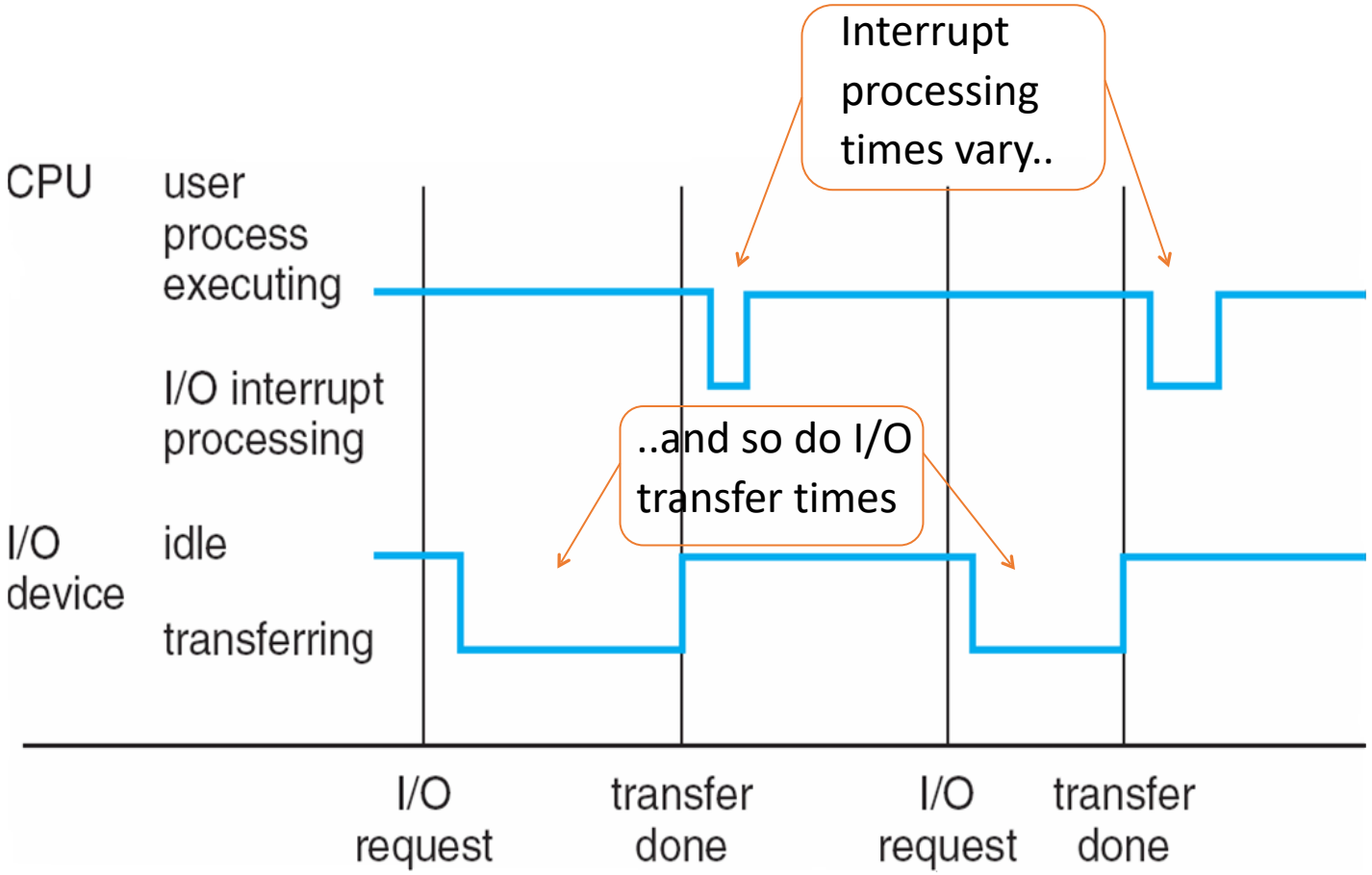


Interrupt Handling

- The operating system saves the state of the CPU by storing registers and the program counter
- Determines which type of interrupt has occurred:
  - polling
  - vectored interrupt system
- Separate segments of code determine what action should be taken for each type of interrupt



Interrupt Timeline for a single process doing output



OPERATING SYSTEMS

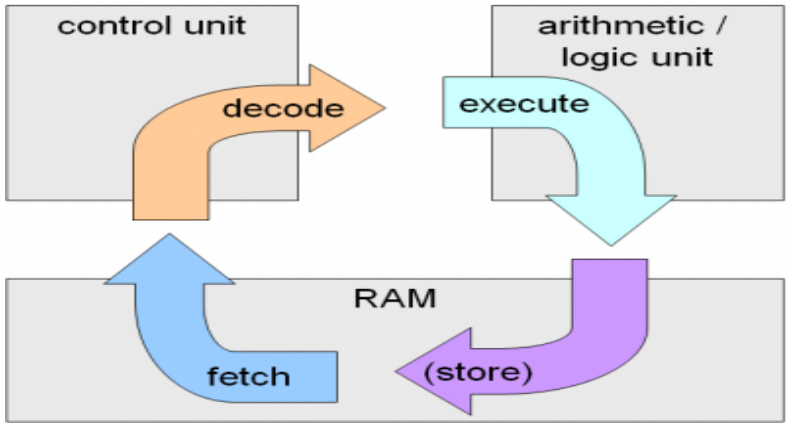
Storage Structure

- Main memory – only large storage media that the CPU can access directly (**Random access memory** and typically **volatile)**
  - Implemented with semiconductor technology called DRAM
- Computers use other forms of memory like ROM, EEPROM
- Smart phones have EEPROM to store factory installed programs.

OPERATING SYSTEMS

Program Execution Model

- Typical instruction execution



- The processor fetches instructions from memory, decodes and executes them.
- The Fetch, Decode and Execute cycles are repeated until the program terminates.
- This is called the Von Neumann model of computing.

Image source :https://computersciencewiki.org/index.php/The_machine_instruction_cycle

OPERATING SYSTEMS

Storage Structure

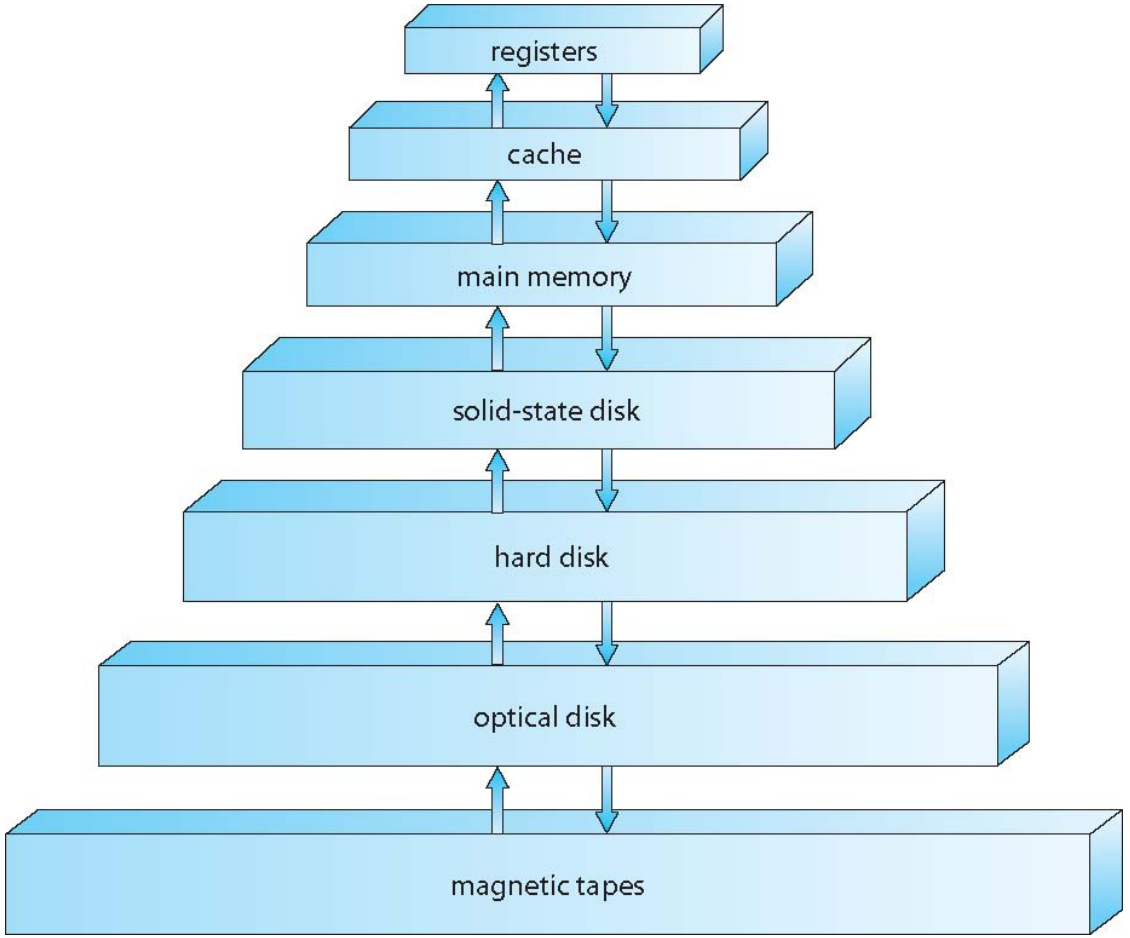
- Secondary storage – extension of main memory that provides large **nonvolatile** storage capacity
- Hard disks – rigid metal or glass platters covered with magnetic recording material
  - Disk surface is logically divided into **tracks**, which are subdivided into **sectors**
  - The **disk controller** determines the logical interaction between the device and the computer
- **Solid-state disks** – faster than hard disks, nonvolatile
  - Various technologies and becoming more popular
  - Flash memory used in camera's PDA's

Storage Hierarchy

- Storage systems organized in hierarchy
  - Speed
  - Cost
  - Volatility
- **Caching** – copying information into faster storage system; main memory can be viewed as a cache for secondary storage

OPERATING SYSTEMS

Storage-Device Hierarchy



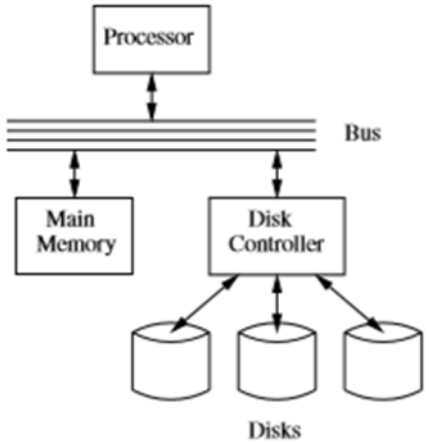
Caching

- Important principle, performed at many levels in a computer (in hardware, operating system, software)
- Information in use copied from slower to faster storage temporarily
- Faster storage (cache) checked first to determine if information is there
  - If it is, information used directly from the cache (fast)
  - If not, data copied to cache and used there
- Cache smaller than storage being cached
  - Cache management important design problem
  - Cache size and replacement policy

OPERATING SYSTEMS

I/O Structure

- Storage is a type of I/O device
- A large portion of operating-system code is dedicated to managing I/O
    - As reliability and performance of a system is the main concern.
- General-purpose computer system consists of CPUs and multiple device controllers that are connected through a common bus.
- Each device controller is in charge of a specific type of device.
- **Device Driver** for each device controller to manage I/O
    - Provides uniform interface between controller and kernel
- **Small Computer-Systems Interface (SCSI)** controller enables to connect more devices.



I/O Structure

- A device controller maintains some local buffer storage and a set of special-purpose registers.
- The device controller is responsible for moving the data between the peripheral devices.

OPERATING SYSTEMS

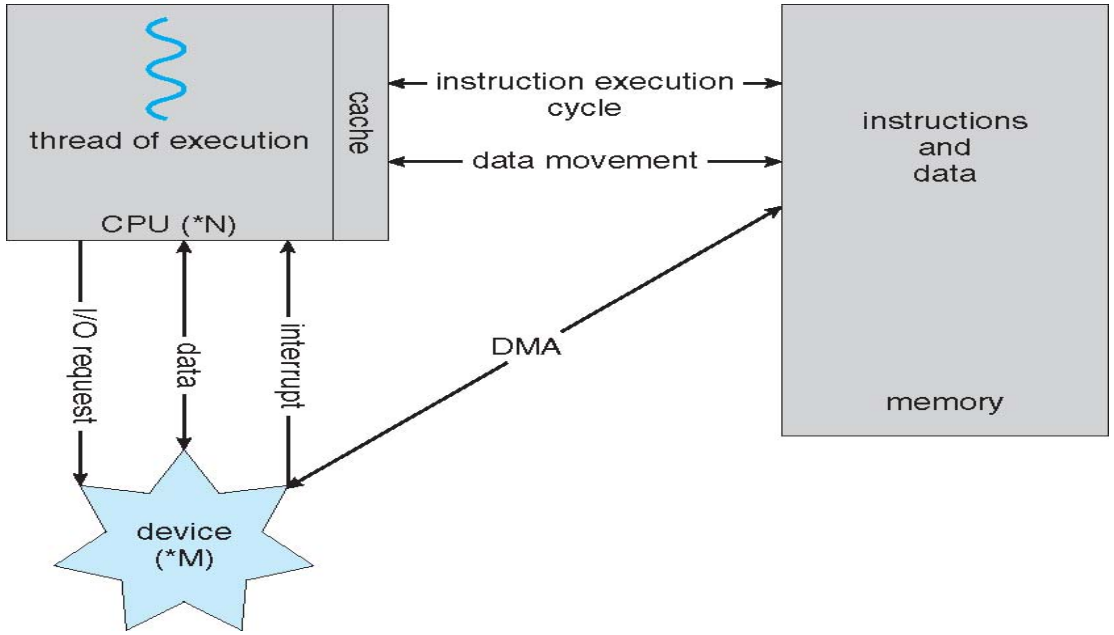
I/O Structure

- After I/O starts, control returns to user program only upon I/O completion
  - Wait instruction idles the CPU until the next interrupt
  - Wait loop (contention for memory access)
  - At most one I/O request is outstanding at a time, no simultaneous I/O processing
- After I/O starts, control returns to user program without waiting for I/O completion
  - **System call** – request to the OS to allow user to wait for I/O completion
  - **Device-status table** contains entry for each I/O device indicating its type, address, and state
  - OS indexes into I/O device table to determine device status and to modify table entry to include interrupt.

OPERATING SYSTEMS

Direct Memory Access Structure

- Used for high-speed I/O devices able to transmit information at close to memory speeds
- Device controller transfers blocks of data from buffer storage directly to main memory without CPU intervention
- Only one interrupt is generated per block, rather than the one interrupt per byte



THANK YOU

Suresh Jamadagni
Department of Computer Science Engineering

sureshjamadagni@pes.edu

OPERATING SYSTEMS

Computer-System Architecture, OS Structure and Operations

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Slides Credits for all the PPTs of this course

- The slides/diagrams in this course are an **adaptation**, **combination**, and **enhancement** of material from the following resources and persons:

1. Slides of Operating System Concepts, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne - 9th edition 2013 and some slides from 10th edition 2018
2. Some conceptual text and diagram from Operating Systems - Internals and Design Principles, William Stallings, 9th edition 2018
3. Some presentation transcripts from A. Frank – P. Weisberg
4. Some conceptual text from Operating Systems: Three Easy Pieces, Remzi Arpaci-Dusseau, Andrea Arpaci Dusseau

OPERATING SYSTEMS

Computer-System Architecture, OS Structure & Operations

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Computer-System Architecture

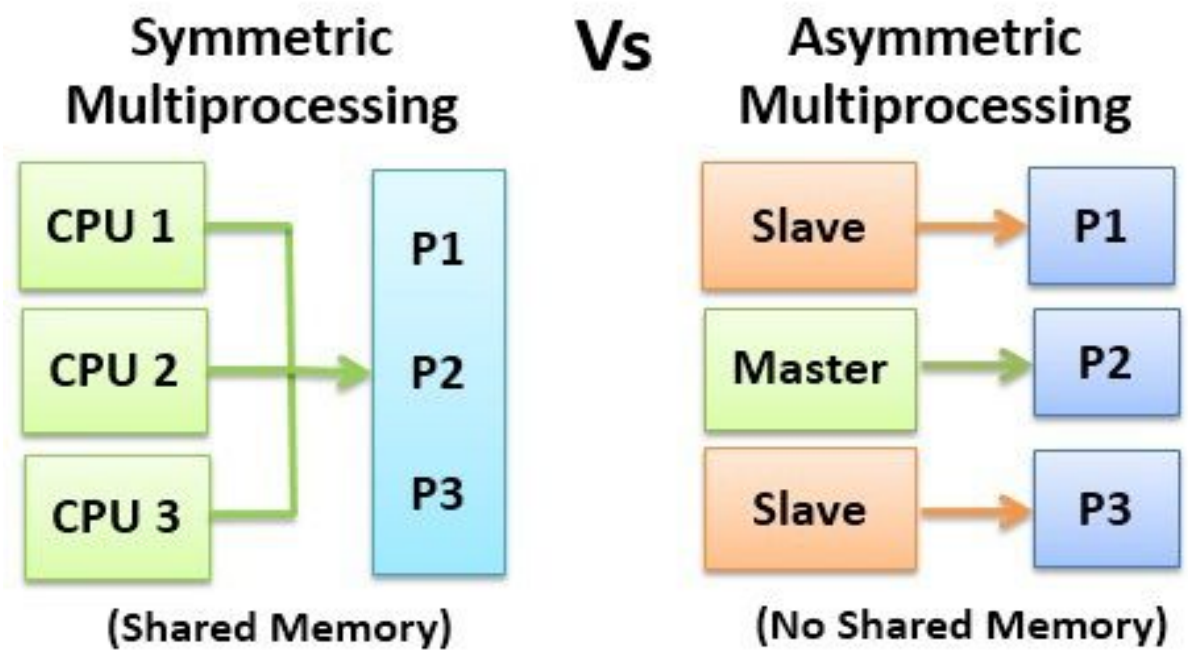
- Most systems use a single general-purpose processor
  - Most systems have other special-purpose processors as well.
  - Device specific processors like disk, keyboard, graphic controller
  - Special-purpose processors run a limited number of instructions
  - Special-purpose processors are low-level components built into the hardware
  - Managed by OS.
  - OS monitors the status.
- Example Disk controller microprocessor
  - Receives sequence of requests from CPU.
  - Implements its own disk queue and scheduling algorithm
  - Relieves the main CPU of the overhead of disk scheduling.

Computer-System Architecture

- **Multiprocessors** systems growing in use and importance
  - Also known as **parallel systems**, **tightly-coupled systems**
  - Advantages include:
    - **Increased throughput**
    - **Economy of scale**
    - **Increased reliability** – graceful degradation or fault tolerance

Two types of Multiprocessor Systems

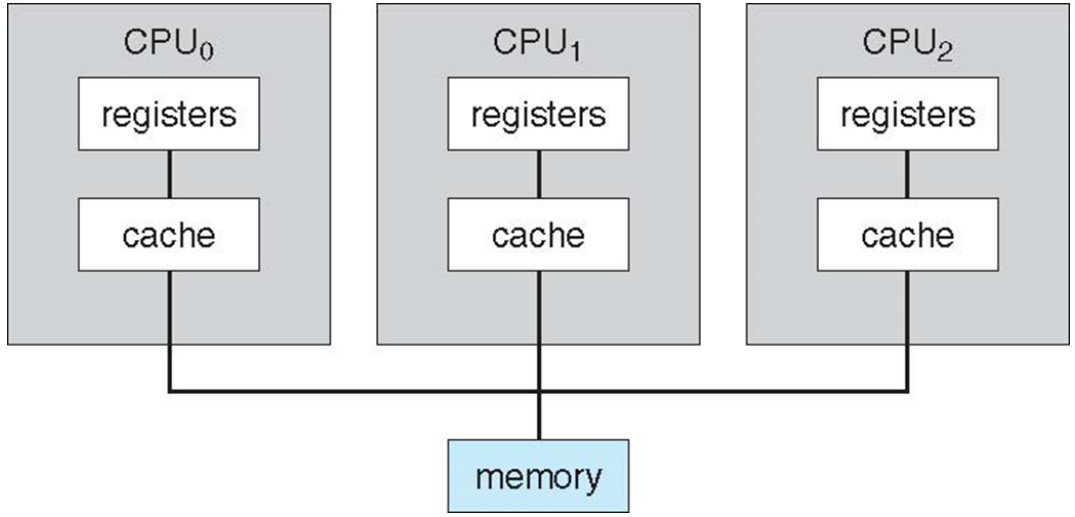
1. **Asymmetric Multiprocessing** – each processor is assigned a specific task.
2. **Symmetric Multiprocessing** – each processor performs all tasks



OPERATING SYSTEMS

Symmetric Multiprocessing Architecture

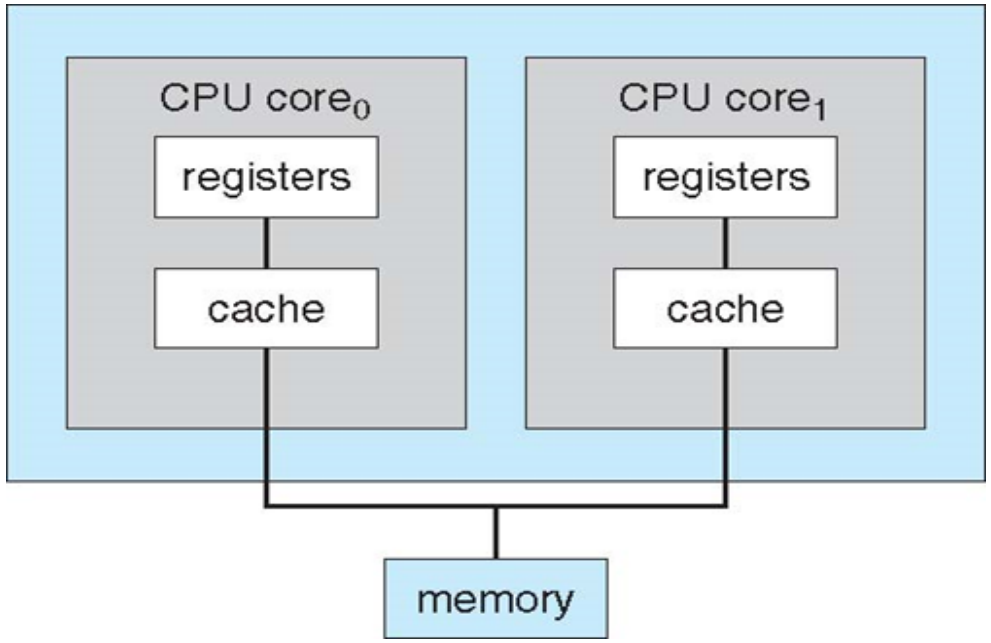
- In SMP all processors are peers; no boss–worker relationship exists between processors.
- Each processor has its own set of registers, as well as a private or local cache.
- All processors share physical memory.



OPERATING SYSTEMS

Multi-Core Design

- A recent trend in CPU design is to include multiple computing cores on a single chip. Such multiprocessor systems are termed **multicore**.
- More efficient than multiple chips with single cores because on-chip communication is faster than between-chip communication.
- One chip with multiple cores uses significantly less power than multiple single-core chips.



A dual-core design with two cores placed on the same chip

Command to know the number of cores, cache details
$cat /proc/cpuinfo|more

OPERATING SYSTEMS

Multi-Core Design

Command to know the number of cores, cache details

$more /proc/cpuinfo

```
 rtm mpx rdseed adx smap clflushopt intel_pt xsaveopt xsavec xgetbv1 xsaves dtherm ida arat pln pts hw
p hwp_notify hwp_act_window hwp_epp md_clear flush_l1d arch_capabilities
bugs            : spectre_v1 spectre_v2 spec_store_bypass mds
bogomips        : 7200.00
clflush size    : 64
cache_alignment : 64
address sizes   : 39 bits physical, 48 bits virtual
power management:

┌[bwang UNIX] [/dev/pts/5]
└[~]> lscpu
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              16
On-line CPU(s) list: 0-15
Thread(s) per core:  2
Core(s) per socket:  8
Socket(s):           1
NUMA node(s):        1
Vendor ID:           GenuineIntel
CPU family:          6
Model:               158
Model name:          Intel(R) Core(TM) i9-9900K CPU @ 3.60GHz
Stepping:            12
CPU MHz:             4784.069
CPU max MHz:         5000.0000
CPU min MHz:         800.0000
BogoMIPS:            7200.00
Virtualization:      VT-x
L1d cache:           32K
L1i cache:           32K
L2 cache:            256K
L3 cache:            16384K
NUMA node0 CPU(s):   0-15
Flags:               fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush d
ts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs
 bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf tsc_known_freq pni pclmulqdq dtes64 monitor
ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadli
ne_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb invpcid_single ssbd ibrs
ibpb stibp tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invp
cid rtm mpx rdseed adx smap clflushopt intel_pt xsaveopt xsavec xgetbv1 xsaves dtherm ida arat pln pts
 hwp hwp_notify hwp_act_window hwp_epp md_clear flush_l1d arch_capabilities
┌[bwang UNIX] [/dev/pts/5]
└[~]>
```

Blade servers

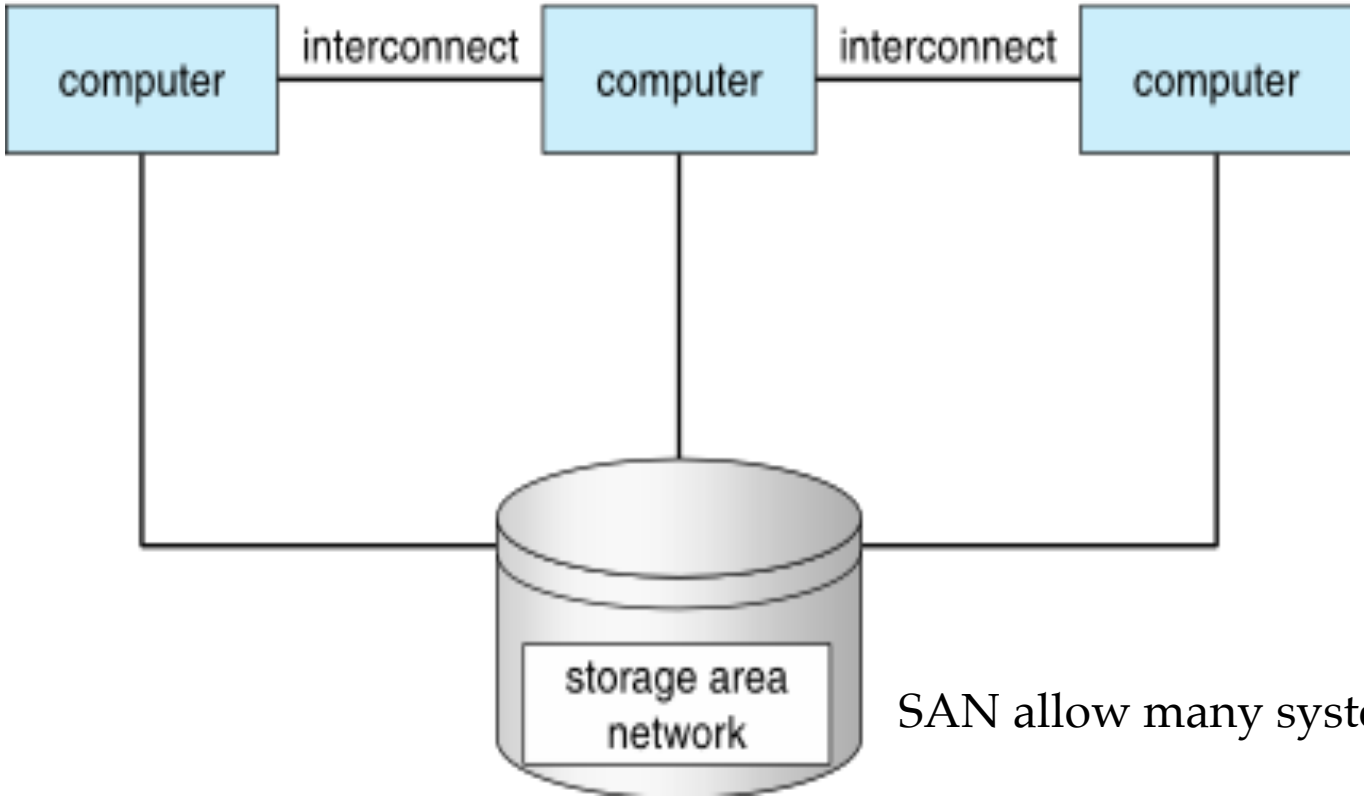
- **Blade servers** are a recent development in which multiple processor boards, I/Oboards, and networking boards are placed in the same chassis.
    - blade-processor board boots independently and runs its own operating system.
    - Some blade-server boards are multiprocessor as well, which blurs the lines between types of computers.
    - In essence, these servers consist of multiple independent multiprocessor systems.

OPERATING SYSTEMS

Clustered Systems

- Like multiprocessor systems, but multiple systems working together
  - Usually sharing storage via a **storage-area network (SAN)**
  - Provides a **high-availability** service which survives failures
    - **Asymmetric clustering** has one machine in hot-standby mode
    - **Symmetric clustering** has multiple nodes running applications, monitoring each other
  - Some clusters are for **high-performance computing (HPC)**
    - Applications must be written to use **parallelization**
  - Some have **distributed lock manager** (**DLM**) to avoid conflicting operations (Ex: when multiple hosts access the same data on shared storage)

Clustered Systems



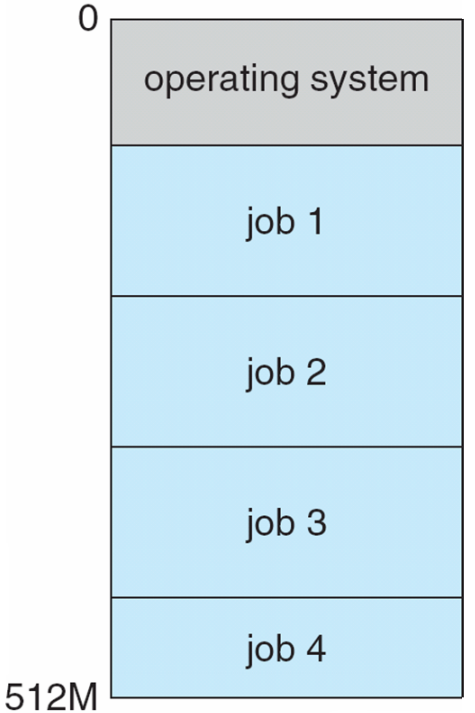
SAN allow many systems to attach to a pool of storage

General structure of a clustered system.

OPERATING SYSTEMS

Operating-System Structure - Multiprogramming

- **Multiprogramming** (**Batch system**) needed for efficiency
  - Single user cannot keep CPU and I/O devices busy at all times
  - Multiprogramming organizes jobs (code and data) so CPU always has one to execute
  - A subset of total jobs in system is kept in memory
  - One job selected and run via **job scheduling**
  - When it has to wait (for I/O for example), OS switches to another job



Operating-System Structure - Multitasking

- **Timesharing** (**multitasking**) is logical extension in which CPU switches jobs so frequently that users can interact with each job while it is running, creating **interactive** computing
  - **Response time** should be < 1 second
  - Each user has at least one program executing in memory ⇨ **process**
  - If several jobs ready to run at the same time ⇨ **CPU scheduling**
  - If processes don't fit in memory, **swapping** moves them in and out to run
  - **Virtual memory** allows execution of processes not completely in memory

Operating-System Operations

- **Interrupt driven** (hardware and software)
  - Hardware interrupt by one of the devices
  - Software interrupt (**exception** or **trap):**
    - Software error (e.g., division by zero)
    - Request for operating system service
    - Other process problems include infinite loop, processes modifying each other or the operating system

OPERATING SYSTEMS

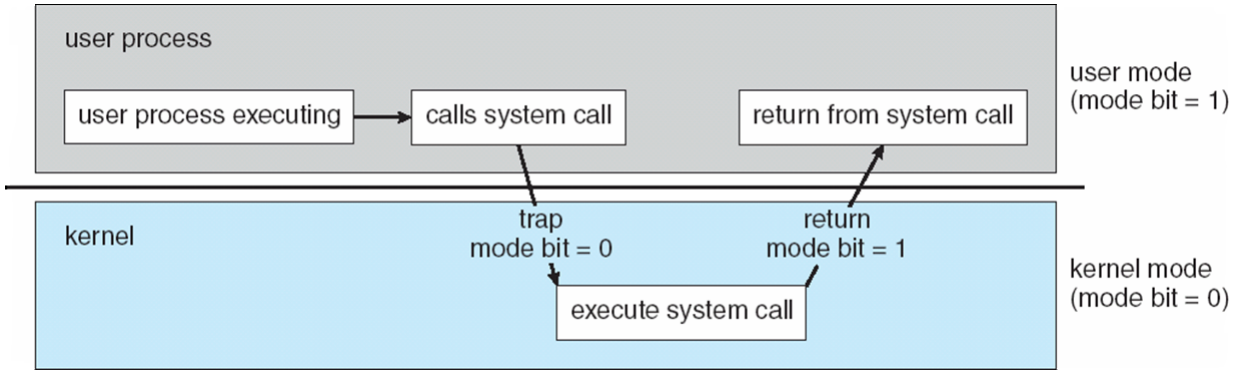
Dual-Mode and Multimode Operation

- **Dual-mode** operation allows OS to protect itself and other system components
  - **User mode** and **kernel mode**
  - **Mode bit** provided by hardware
    - Provides ability to distinguish when system is running user code or kernel code
    - Some instructions designated as **privileged**, only executable in kernel mode
    - System call changes mode to kernel, return from call resets it to user
- Increasingly CPUs support multi-mode operations
  - i.e. **virtual machine manager** (**VMM**) mode for guest **VMs**

OPERATING SYSTEMS

Transition from user to kernel mode

- When a trap or interrupt occurs, hardware switches from user mode to kernel mode (changes the state of the mode bit to 0).
- When the request is fulfilled, the system always switches to user mode (by setting the mode bit to 1) before passing control to a user program.



OPERATING SYSTEMS

Timer

- Timer to prevent infinite loop / process hogging resources
  - Timer is set to interrupt the computer after a specified period (fixed 1/60 sec or variable 1 msec to 1 sec)
  - A **variable timer** is generally implemented by a fixed-rate clock and a counter.
  - Operating system sets the counter (privileged instruction)
  - Every time the clock ticks, the counter is decremented.
  - When counter reaches zero, an interrupt occurs
  - Timer can be used to prevent a user program from running too long (terminate the program)

THANK YOU

Suresh Jamadagni
Department of Computer Science Engineering

sureshjamadagni@pes.edu

OPERATING SYSTEMS

Kernel Data Structures and Computing Environments

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Slides Credits for all the PPTs of this course

- The slides/diagrams in this course are an **adaptation**, **combination**, and **enhancement** of material from the following resources and persons:

1. Slides of Operating System Concepts, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne - 9th edition 2013 and some slides from 10th edition 2018
2. Some conceptual text and diagram from Operating Systems - Internals and Design Principles, William Stallings, 9th edition 2018
3. Some presentation transcripts from A. Frank – P. Weisberg
4. Some conceptual text from Operating Systems: Three Easy Pieces, Remzi Arpaci-Dusseau, Andrea Arpaci Dusseau

OPERATING SYSTEMS

Kernel Data Structures

Suresh Jamadagni
Department of Computer Science

Kernel Data Structures

Array:

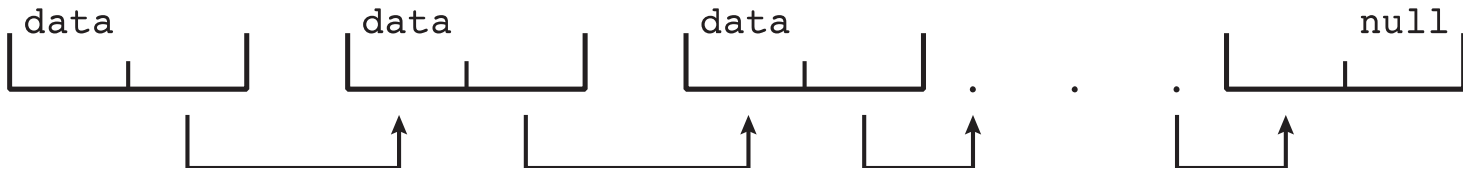
- An array is a simple data structure in which each element can be accessed directly.
- Main Memory constructed with array.
- How the data is accessed?
- Items with multiple bytes are accessed as item number × item size
- But what about storing an item whose size may vary?
- what about removing an item if the relative positions of the remaining items must be preserved?

Lists

- Standard programming data structures are used extensively in OS

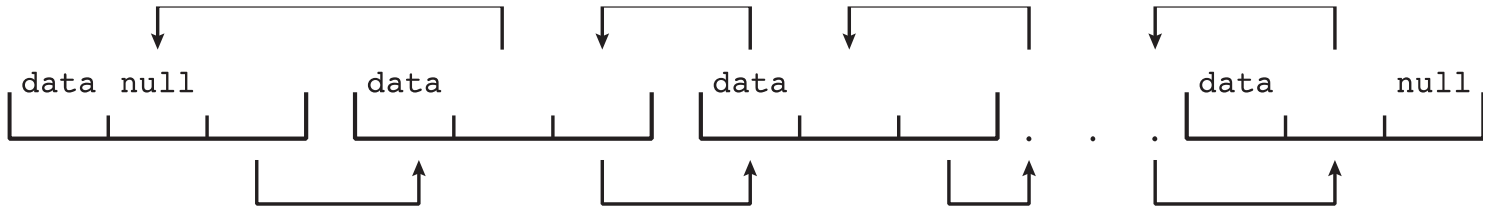
Singly linked list

- The items in a list must be accessed in a particular order.
- common method for implementing this structure is a linked list



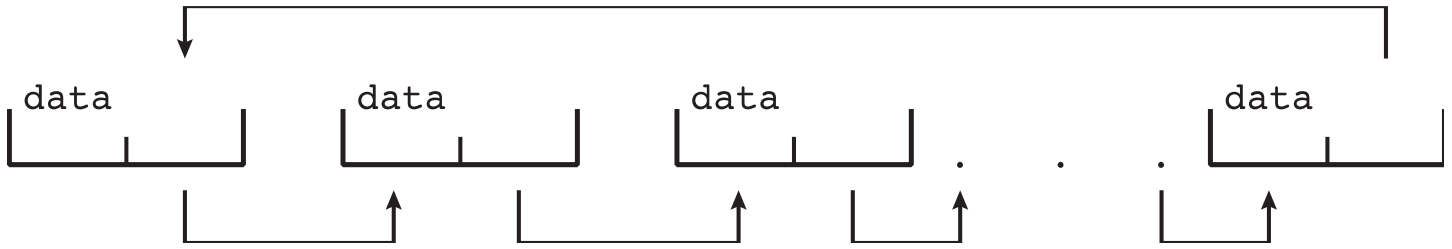
- In a **singly linked list,** each item points to its successor.

Doubly linked list



In a **doubly linked list**, a given item can refer either to its predecessor or to its successor.

Circular linked list



In a **circularly linked list**, the last element in the list refers to the first element, rather than to null.

Lists

Advantages:

- Linked lists accommodate items of varying sizes.
- Allow easy insertion and deletion of items

Disadvantages:

- Performance for retrieving a specified item in a list of size n is linear — O(n) , as it requires potentially traversing all n elements in the worst case.

Usage:

- Lists are used by the some of the kernel algorithms
- Constructing more powerful data structures such as stacks and queues

OPERATING SYSTEMS

Stacks & Queues

Stack - a sequentially ordered data structure that uses **LIFO** principle for adding and removing items

- OS often uses a stack when involving function calls.
- Parameters, local variables and the return address are **push**ed onto the stack when a function is called
- Return from the function call **pop**s those items off the stack

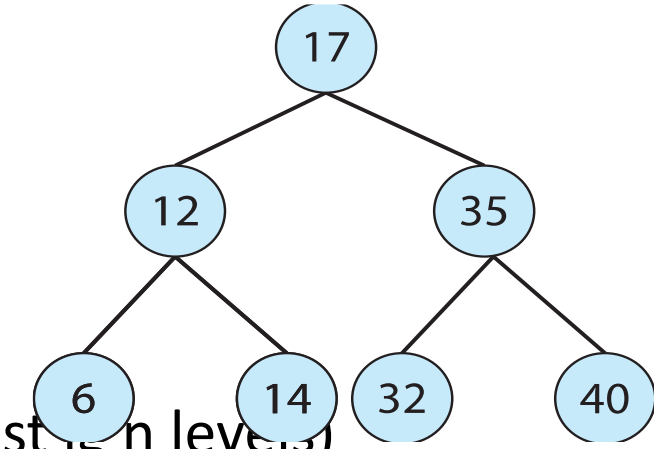
Queue - a sequentially ordered data structure that uses **FIFO** principle for adding and removing items

- Tasks waiting to be run on an available CPU are organized in queues
- Print jobs sent to a printer are printed in the order of submission

OPERATING SYSTEMS

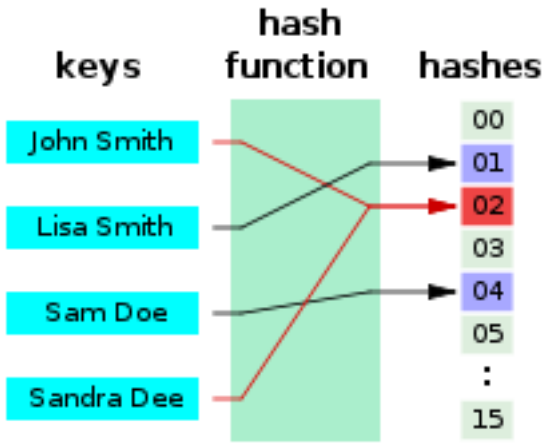
Trees

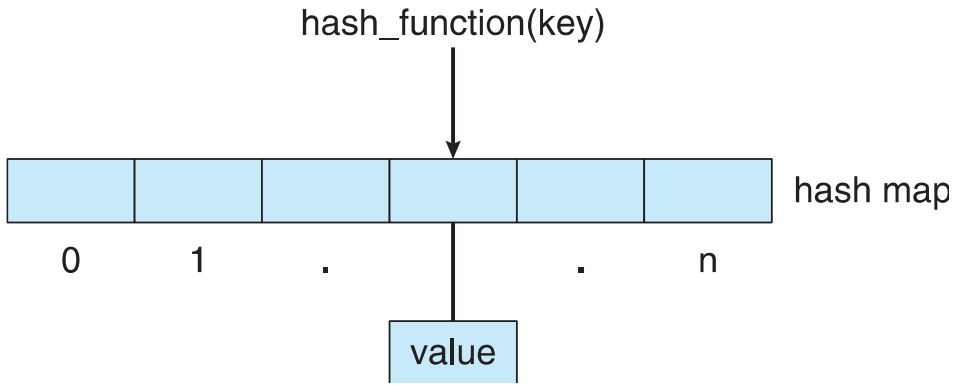
- Data structure used to represent data hierarchically.
- Data values in a tree structure are linked through parent–child relationships
- **Binary search tree**
  - ordering between 2 children: left <= right
  - Search performance is *O(n)*
- **Balanced binary search tree –** (a tree containing n items has at most lg n levels)
  - Search performance is *O(lg n)*
  - Used by Linux for selecting which task to run next (CPU-Scheduling algorithm)



Hash Functions and Maps

- Hash functions can result in the same output value for 2 inputs
- **Hash function** can be used to implement a **hash map**
  - Maps or associates key:value pairs using a hash function
  - Search performance is *O(1)*





Bitmaps

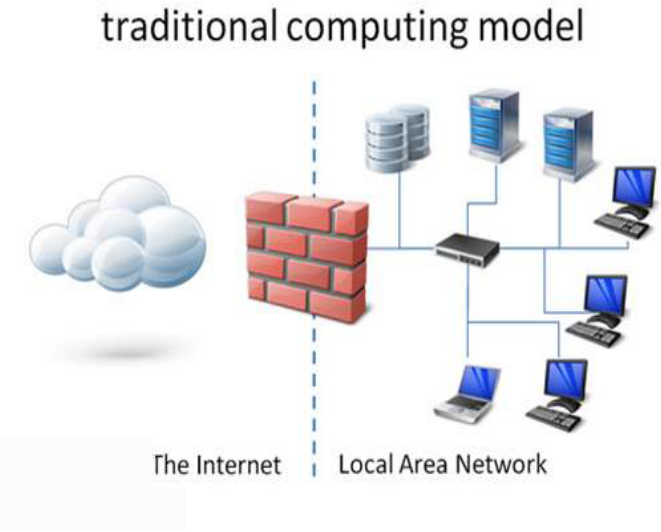
Bitmap - string of *n* binary digits representing the status of *n* items

- Availability of each resource is indicated by the value of a binary digit
  - 0 – resource is available
  - 1 – resource is unavailable
- Value of the **i**th position in the bitmap is associated with the **i**th resource
  - Example: bitmap 001011101 shows resources 2, 4, 5, 6, and 8 are unavailable; resources 0, 1, 3, and 7 are available
- Commonly used to represent the availability of a large number of resources (disk blocks)

OPERATING SYSTEMS

Computing Environments – Traditional

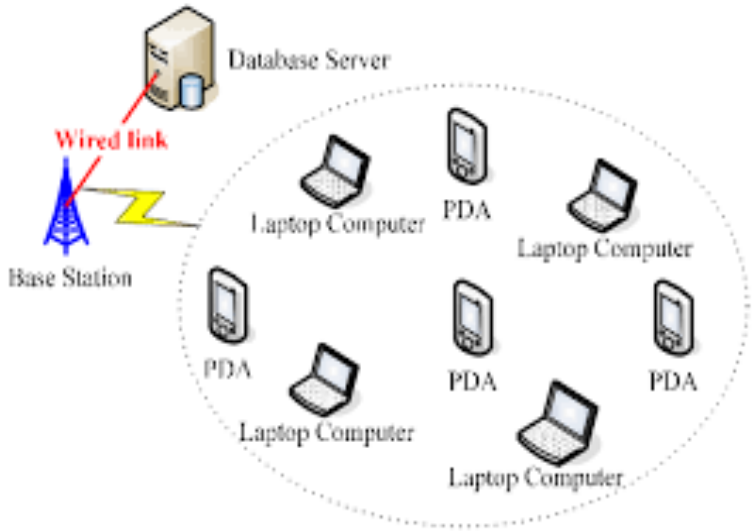
- Stand-alone general purpose machines
- But blurred as most systems interconnect with others (i.e., the Internet)
- **Portals** provide web access to internal systems
- **Network computers** (**thin clients**) are like Web terminals
- Mobile computers interconnect via **wireless networks**
- Networking becoming ubiquitous – even home systems use **firewalls** to protect home computers from Internet attacks



OPERATING SYSTEMS

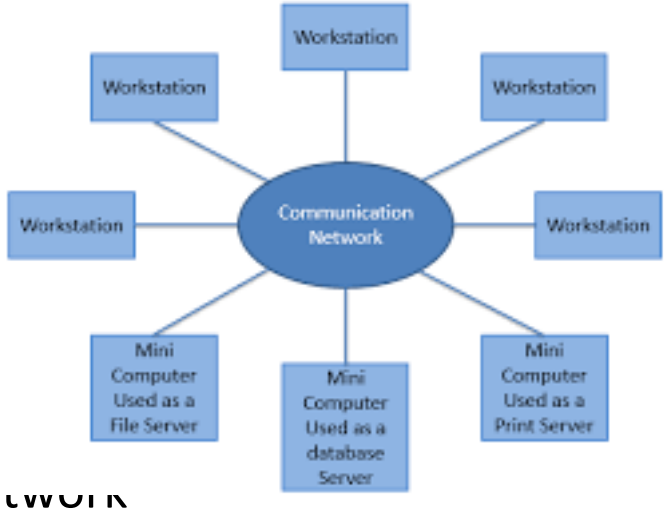
Computing Environments – Mobile

- Handheld smartphones, tablets, etc
- What is the functional difference between them and a “traditional” laptop?
- Extra feature – more OS features (GPS, gyroscope)
- Allows new types of apps like ***augmented reality***
- Use IEEE 802.11 wireless, or cellular data networks for connectivity
- Leaders are **Apple iOS** and **Google Android**



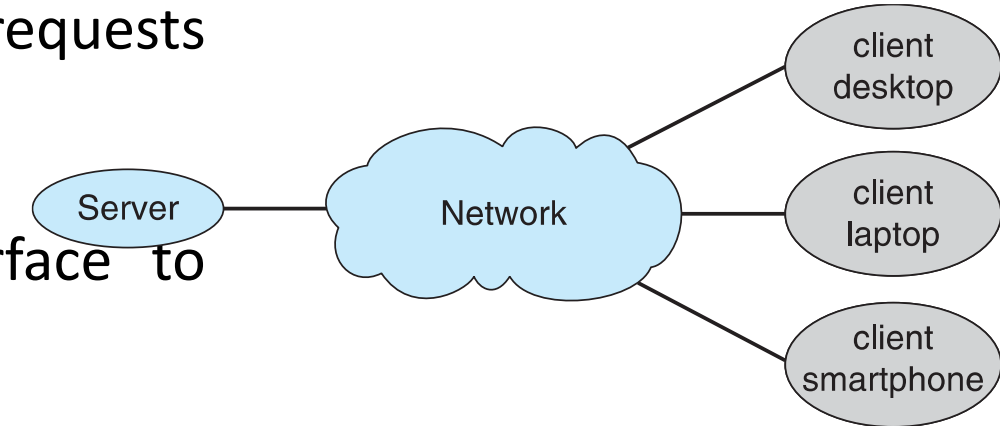
Distributed computing

- Collection of separate, possibly heterogeneous systems networked together
  - **Network** is a communications path, **TCP/IP** most common
    - **Local Area Network** (**LAN**)
    - **Wide Area Network** (**WAN**)
    - **Metropolitan Area Network** (**MAN**)
    - **Personal Area Network** (**PAN**)
- **Network Operating System** provides features between systems across network
  - Communication scheme allows systems to exchange messages
  - Illusion of a single system



Client-Server Computing

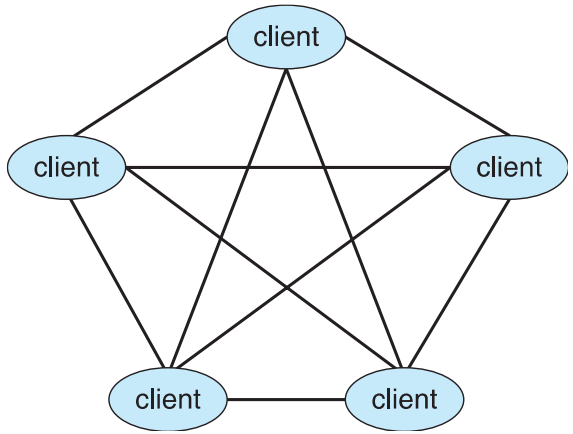
- Dumb terminals replaced by smart PCs
- Many systems now **servers**, responding to requests generated by **clients**
  - **Compute-server system** provides an interface to client to request services (i.e., database)
  - **File-server system** provides interface for clients to store and retrieve files



OPERATING SYSTEMS

Computing Environments – Peer-to-Peer

- Another model of distributed system
  - P2P does not distinguish clients and servers
  - Instead all nodes are considered peers
  - May each act as client, server or both
  - Node must join P2P network
    - Registers its service with central lookup service on network, or
    - Broadcast request for service and respond to requests for service via ***discovery protocol***
  - Examples include Napster and Gnutella, **Voice over IP** (**VoIP**) such as Skype



Computing Environments – Virtualization

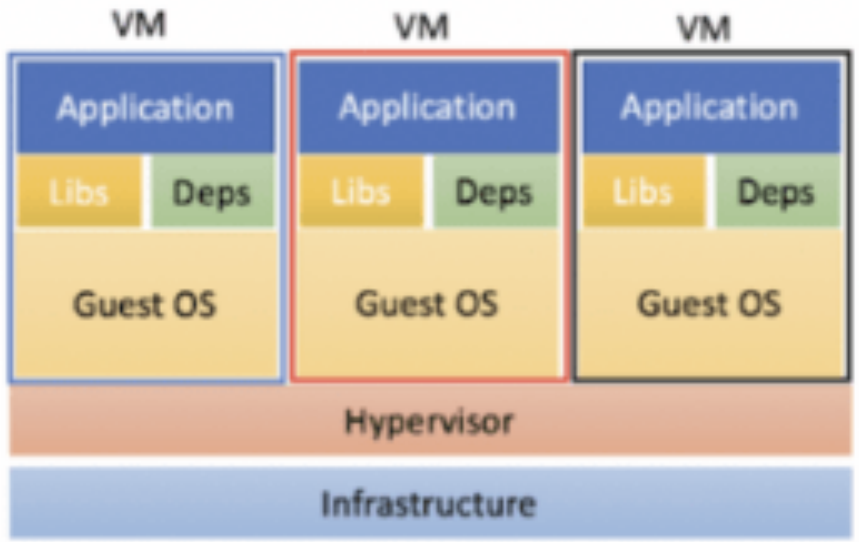
- Allows operating systems to run applications within other OSes
  - Vast and growing industry
- **Emulation** used when source CPU type different from target type (i.e. PowerPC to Intel x86)
  - Generally slowest method
  - When computer language not compiled to native code – **Interpretation**
- **Virtualization** – OS natively compiled for CPU, running **guest** OSes also natively compiled
  - Consider VMware running WinXP guests, each running applications, all on native WinXP **host** OS
  - **VMM** (virtual machine Manager) provides virtualization services

Computing Environments – Virtualization

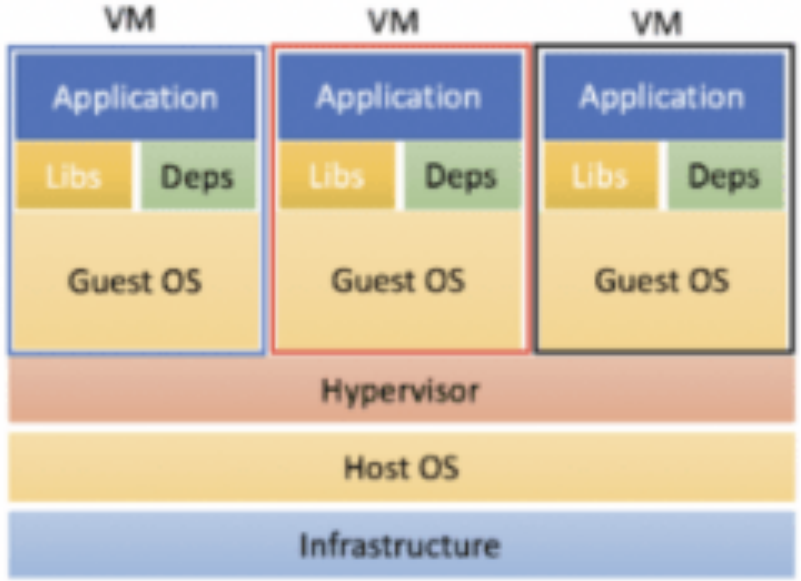
- Use cases involve laptops and desktops running multiple OSes for exploration or compatibility
  - Apple laptop running Mac OS X host, Windows as a guest
  - Developing apps for multiple OSes without having multiple systems
  - QA testing applications without having multiple systems
  - Executing and managing compute environments within data centers
- VMM can run natively, in which case they are also the host

OPERATING SYSTEMS

Computing Environments – Virtualization



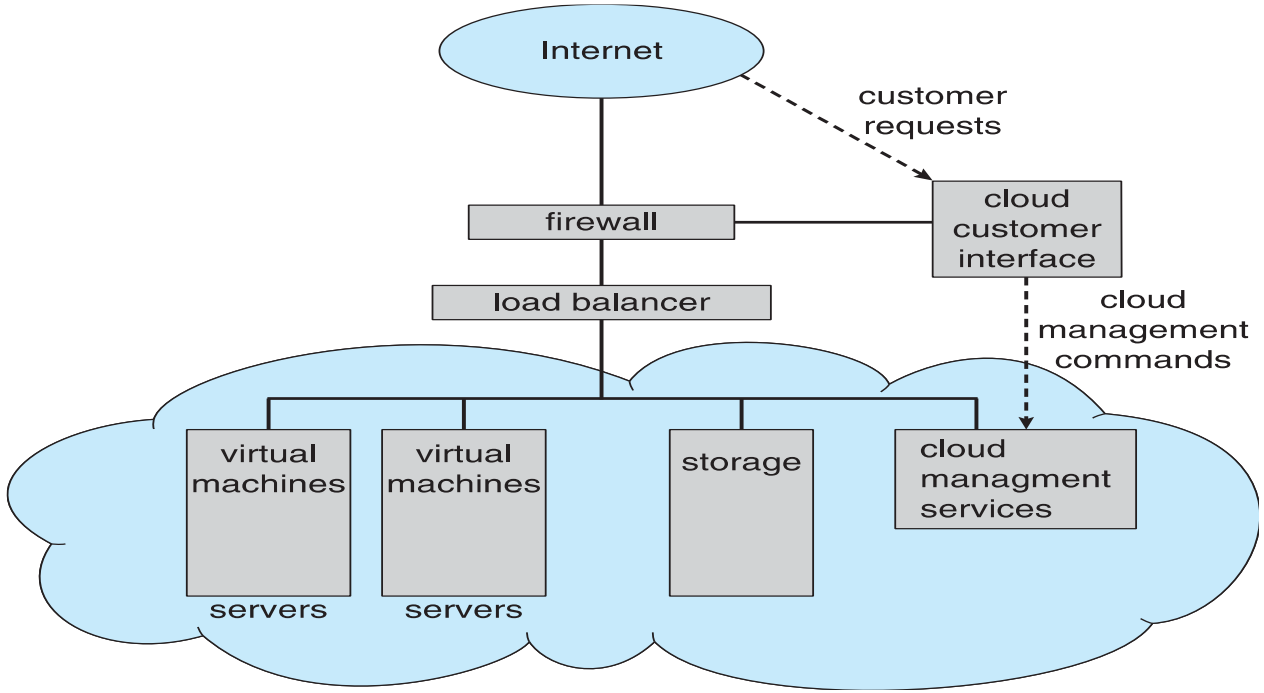
Type-1



Type-2

Computing Environments – Cloud Computing

- Cloud computing environments composed of traditional OSes, plus VMMs, plus cloud management tools
  - Internet connectivity requires security like firewalls
  - Load balancers spread traffic across multiple applications



Computing Environments – Cloud Computing

- Delivers computing, storage, even apps as a service across a network
- Logical extension of virtualization because it uses virtualization as the base for it functionality.
  - Amazon **EC2** has thousands of servers, millions of virtual machines, petabytes of storage available across the Internet, pay based on usage
- Many types
  - **Public cloud** – available via Internet to anyone willing to pay
  - **Private cloud** – run by a company for the company's own use
  - **Hybrid cloud** – includes both public and private cloud components

Computing Environments – Cloud Computing

- Software as a Service (**SaaS**) – one or more applications available via the Internet (i.e., word processor)
- Platform as a Service (**PaaS**) – software stack ready for application use via the Internet (i.e., a database server)
- Infrastructure as a Service (**IaaS**) – servers or storage available over Internet (i.e., storage available for backup use)

Computing Environments – Real-Time Embedded Systems

- Real-time embedded systems most prevalent form of computers
  - Vary considerable, special purpose, limited purpose OS, **real-time OS**
  - Use expanding
- Many other special computing environments as well
  - Some have OSes, some perform tasks without an OS
- Real-time OS has well-defined fixed time constraints
  - Processing ***must*** be done within constraint
  - Correct operation only if constraints met

THANK YOU

Suresh Jamadagni
Department of Computer Science Engineering

sureshjamadagni@pes.edu

OPERATING SYSTEMS

Process Management

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Slides Credits for all the PPTs of this course

- The slides/diagrams in this course are an **adaptation**, **combination**, and **enhancement** of material from the following resources and persons:

1. Slides of Operating System Concepts, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne - 9th edition 2013 and some slides from 10th edition 2018
2. Some conceptual text and diagram from Operating Systems - Internals and Design Principles, William Stallings, 9th edition 2018
3. Some presentation transcripts from A. Frank – P. Weisberg
4. Some conceptual text from Operating Systems: Three Easy Pieces, Remzi Arpaci-Dusseau, Andrea Arpaci Dusseau

OPERATING SYSTEMS

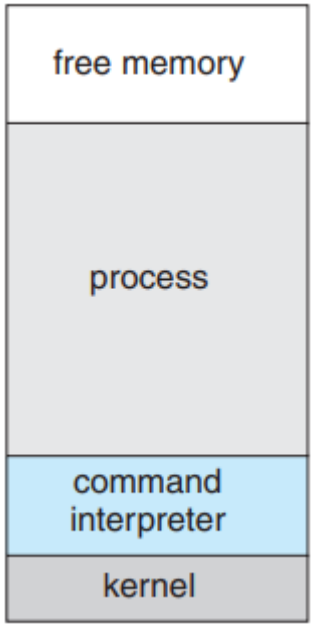
Process Concept

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

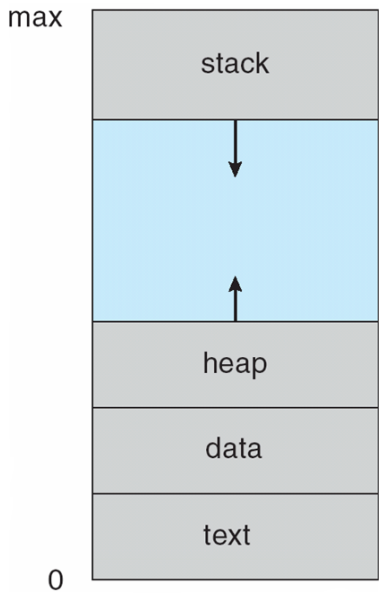
Process Concept

- An operating system executes a variety of programs:
  - Batch system – **jobs**
  - Time-shared systems – **user programs** or **tasks**
- Textbook uses the terms ***job*** and ***process*** almost interchangeably
- **Process** – a program in execution; process execution must progress in sequential fashion
- Program is ***passive*** entity stored on disk (**executable file**), process is ***active***
  - Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc
- One program can be several processes
  - Consider multiple users executing the same program



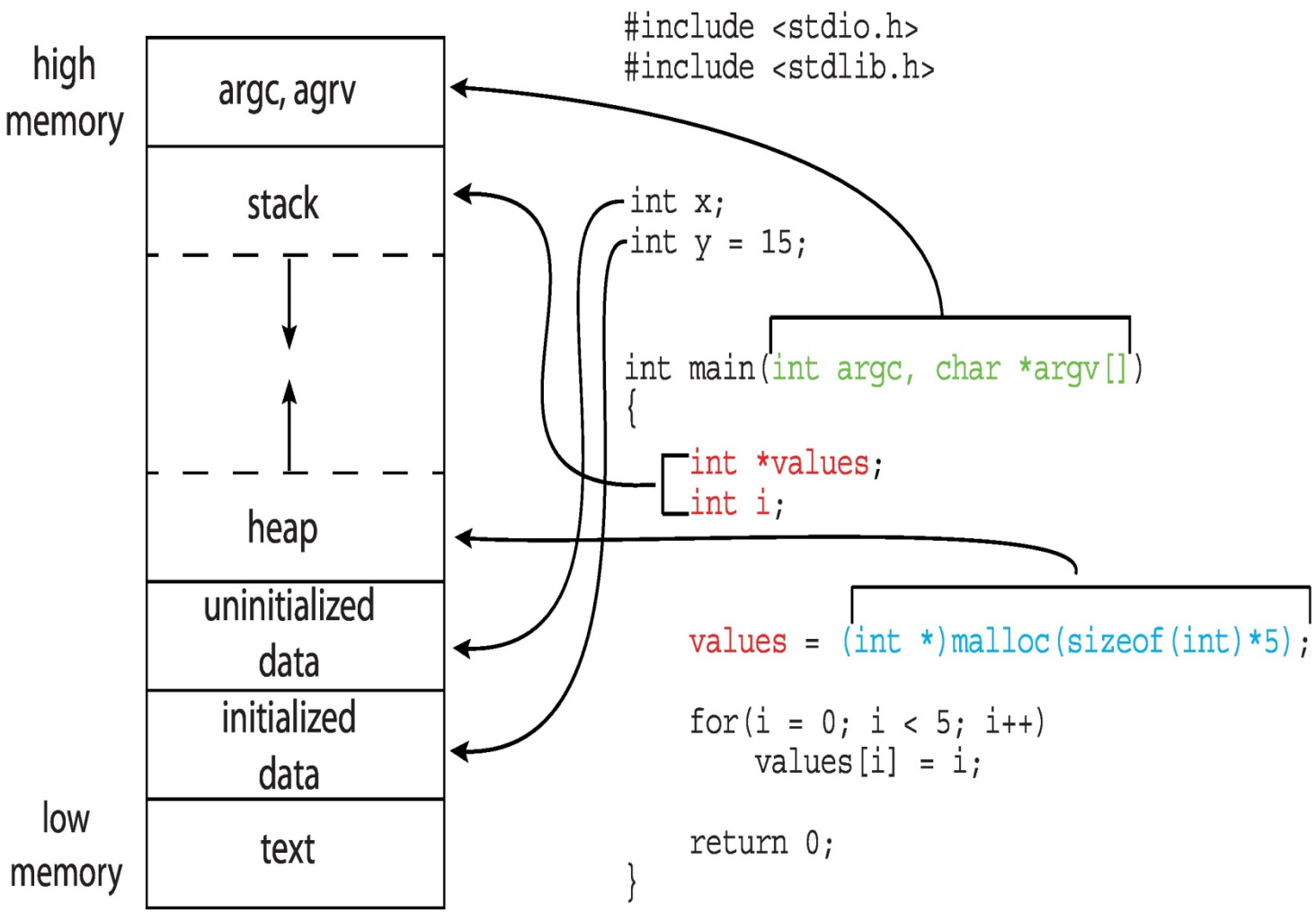
Structure of a process in memory

- The program code, also called **text section.**
  - Includes current activity including **program counter**, processor registers
- **Stack** containing temporary data
  - Function parameters, return addresses, local variables
- **Data section** containing global variables
- **Heap** containing memory dynamically allocated during run time



OPERATING SYSTEMS

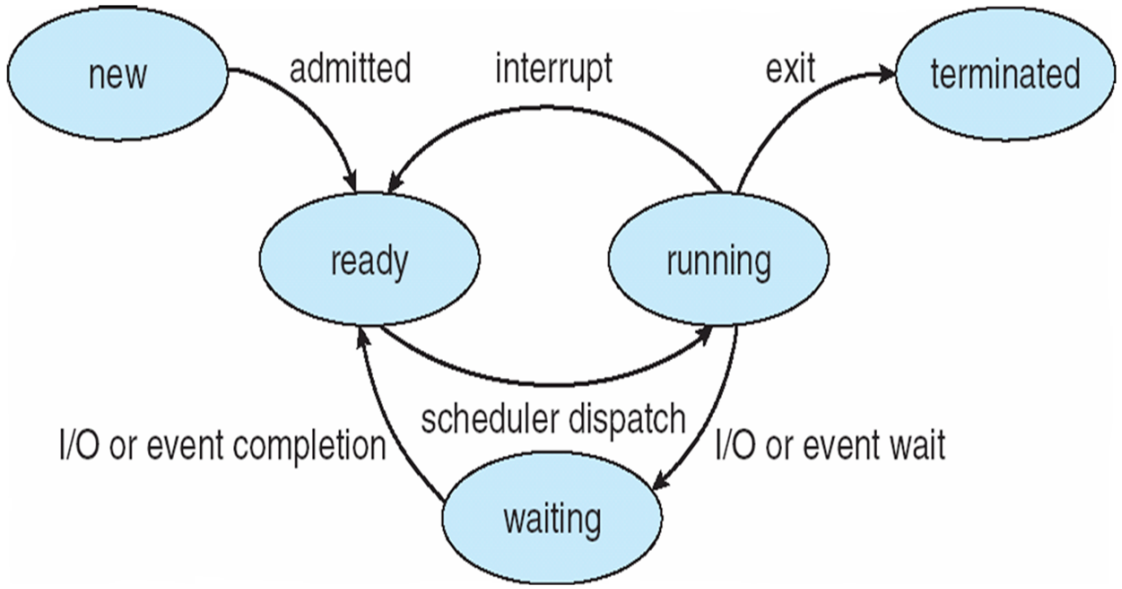
Memory Layout of a C Program



Process State

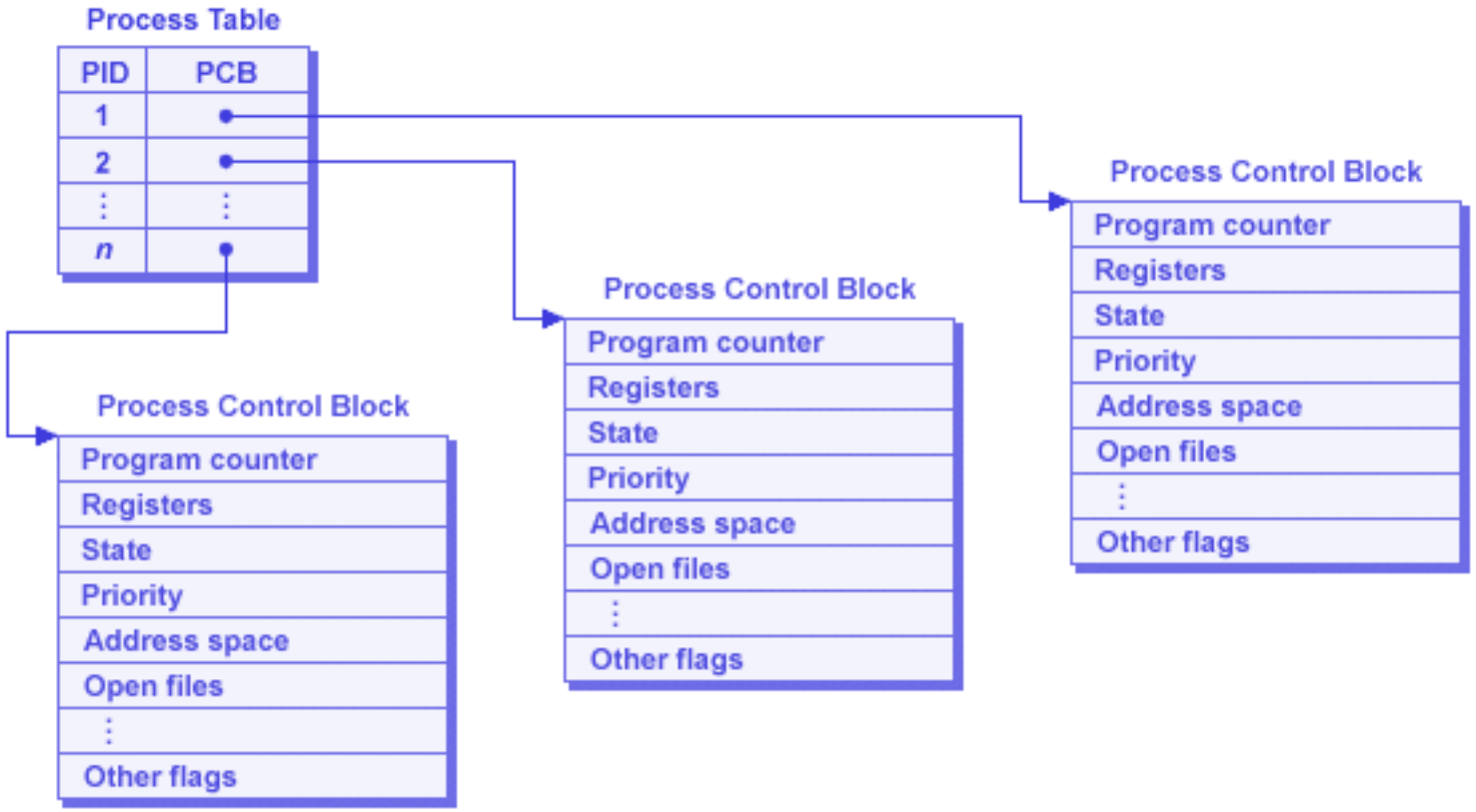
As a process executes, it changes **state**

- **New**: The process is being created
- **Running**: Instructions are being executed
- **Waiting**: The process is waiting for some event to occur
- **Ready**: The process is waiting to be assigned to a processor
- **Terminated**: The process has finished execution



OPERATING SYSTEMS

Process Control Block (PCB)

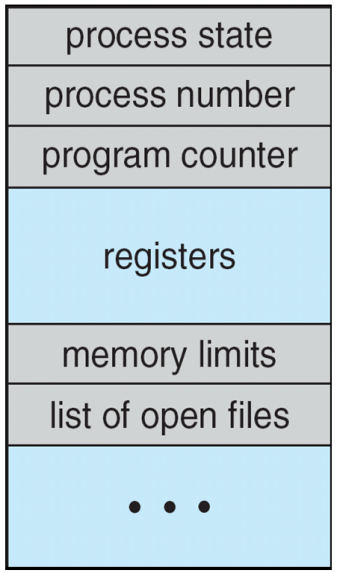


OPERATING SYSTEMS

Process Control Block (PCB)

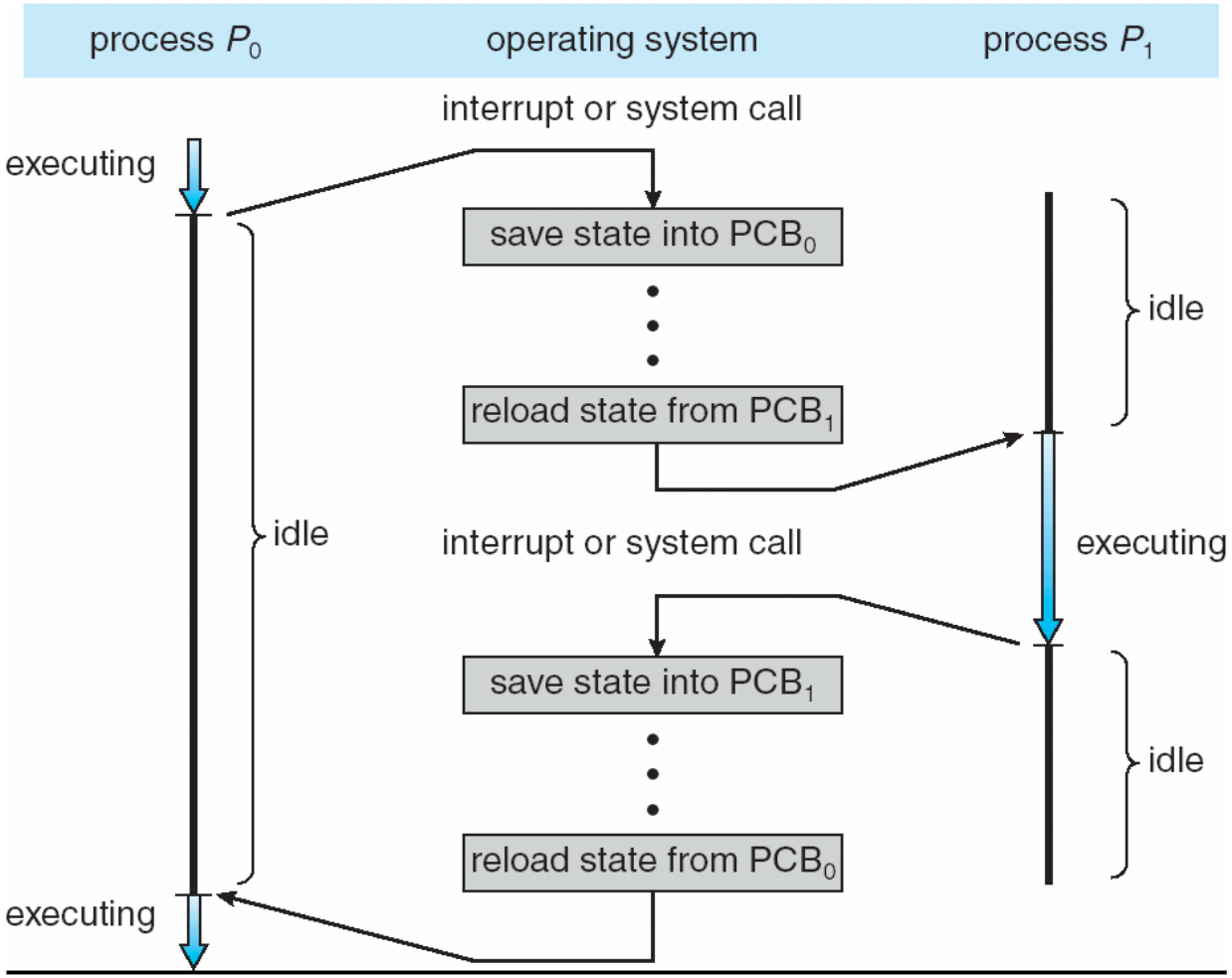
Each process is represented in the operating system by a Process Control Block (also called **task control block**)

- Process state – running, waiting, etc
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files



OPERATING SYSTEMS

CPU switch from process to process



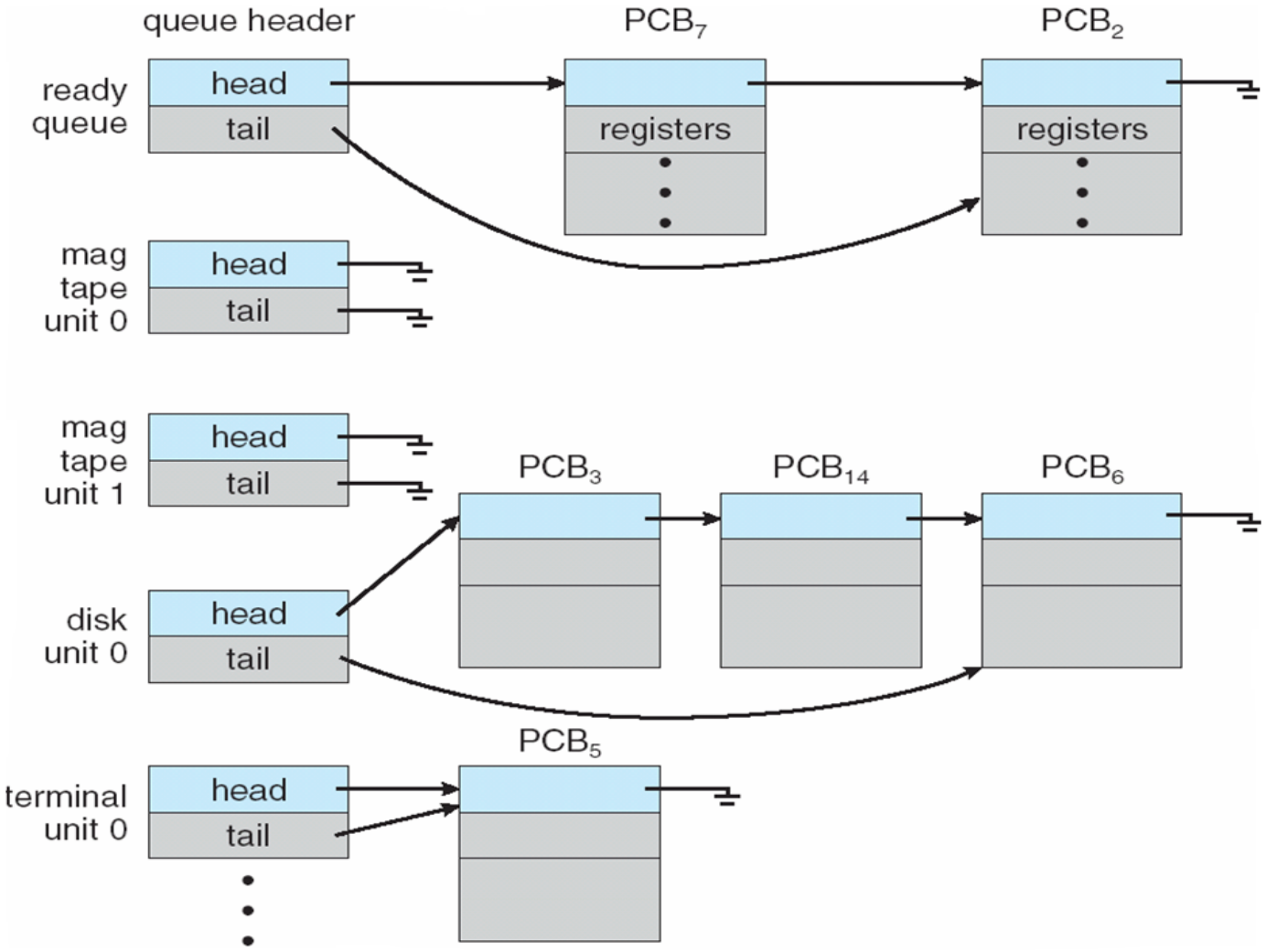
OPERATING SYSTEMS

Process Scheduling

- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Device queues** – set of processes waiting for an I/O device
  - Processes migrate among the various queues

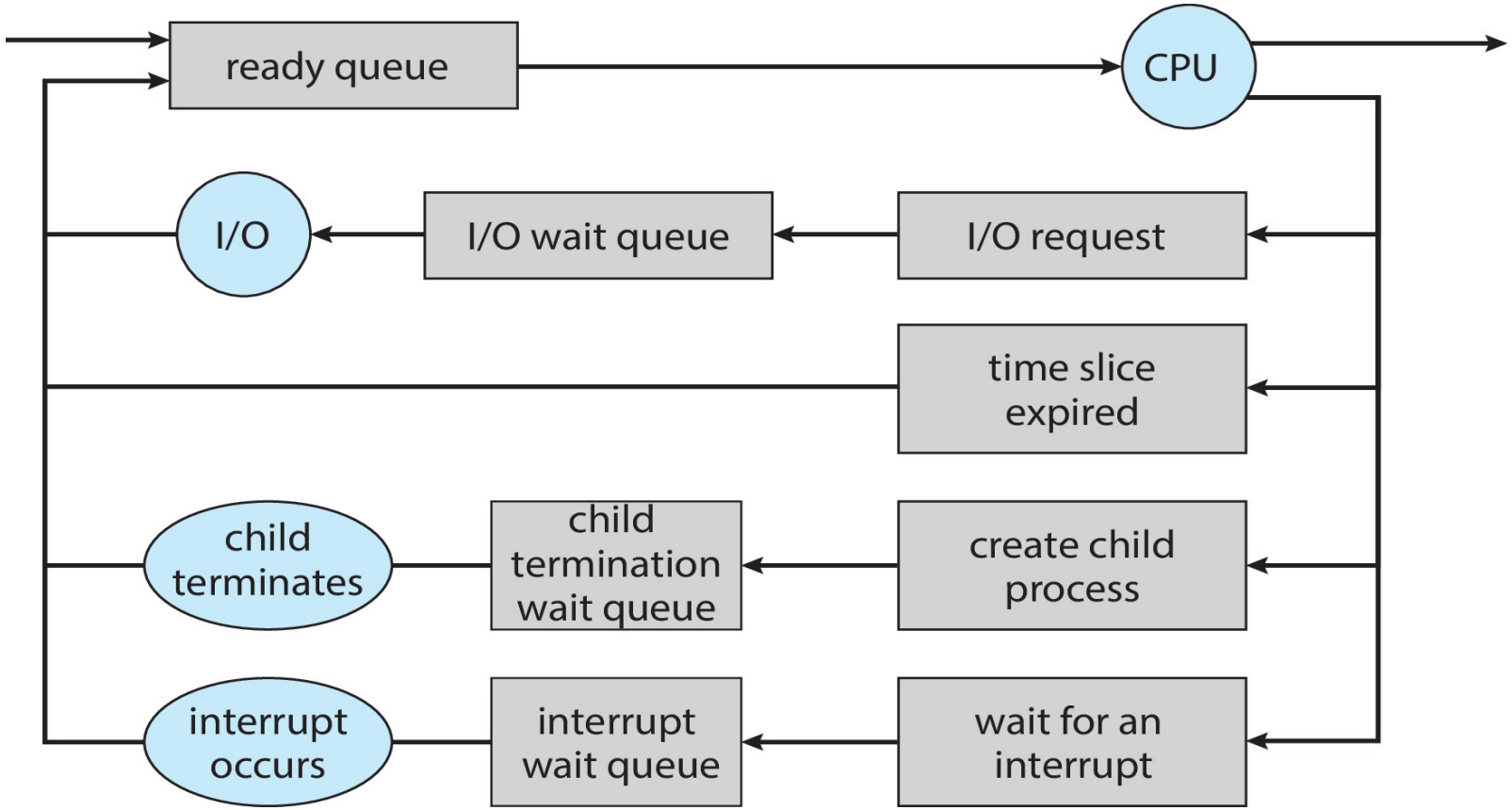
OPERATING SYSTEMS

Ready Queue And Various I/O Device Queues



OPERATING SYSTEMS

Representation of Process Scheduling



OPERATING SYSTEMS

Schedulers

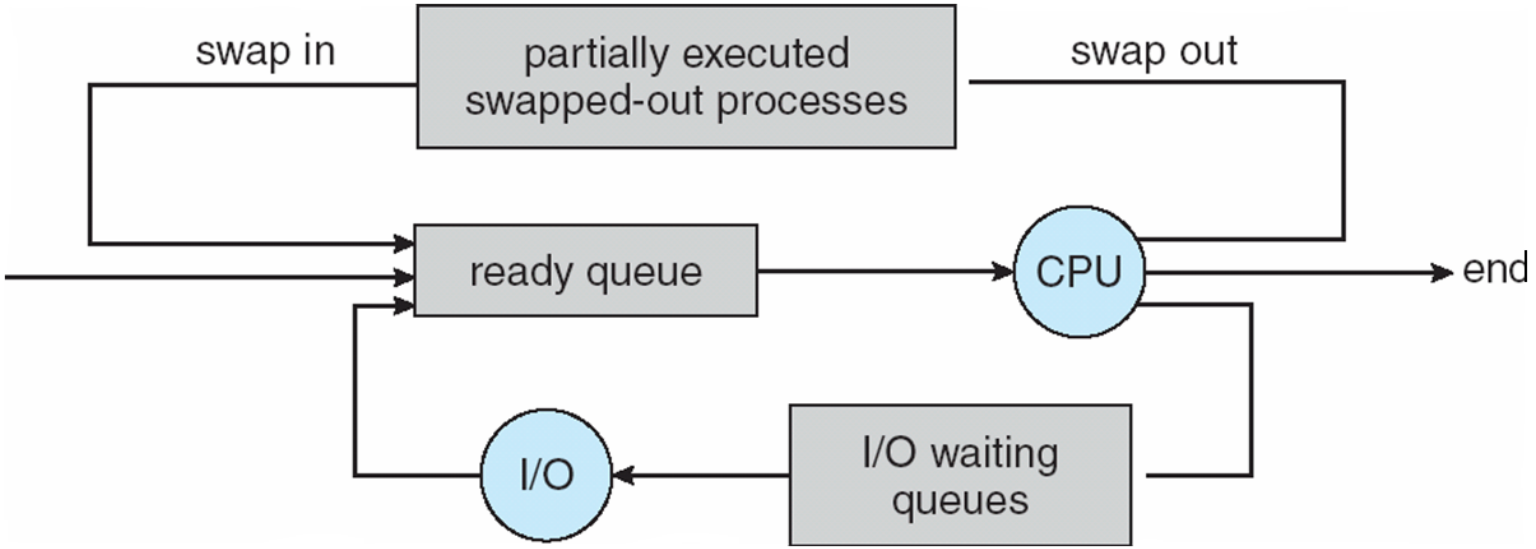
- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) ⇒ (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes) ⇒ (may be slow)
  - The long-term scheduler controls the **degree of multiprogramming**

Schedulers

- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
- Long-term scheduler strives for good ***process mix***

Representation of Process Scheduling

- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
  - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**



OPERATING SYSTEMS

Context Switch

- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**
- **Context** of a process represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
  - The more complex the OS and the PCB ➔ the longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU ➔ multiple contexts loaded at once

Operations on Processes

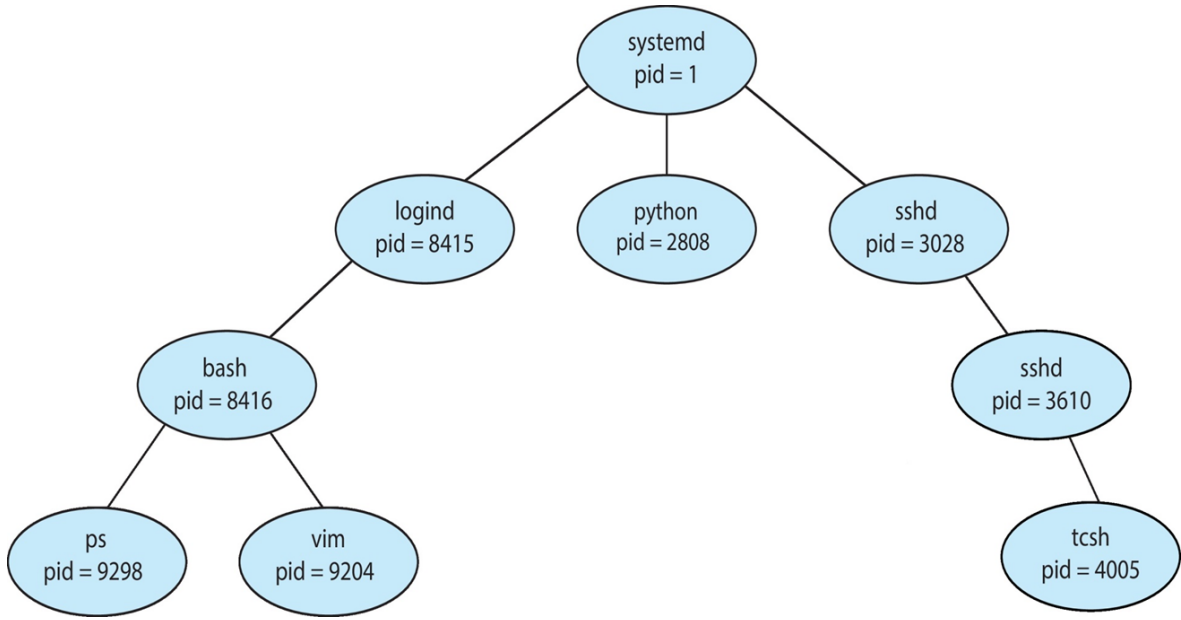
System must provide mechanisms for:

- process creation
- process termination

OPERATING SYSTEMS

Process Creation

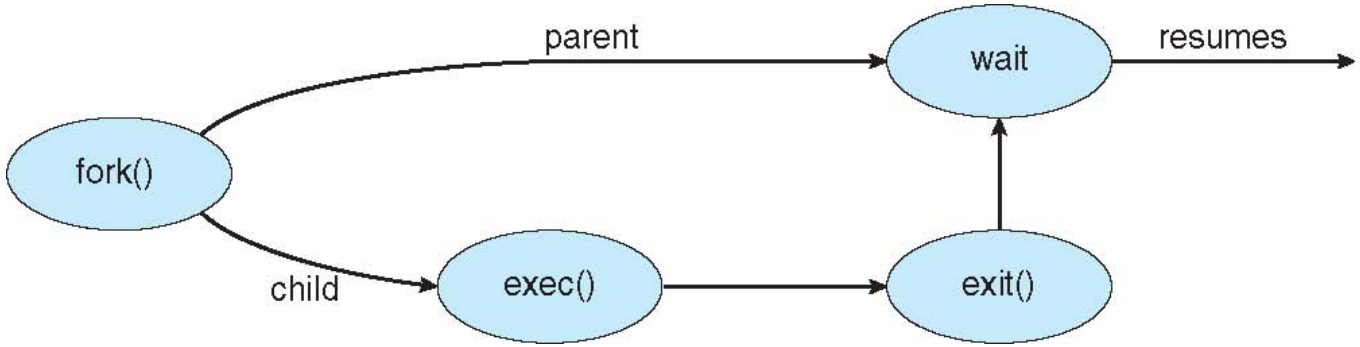
- **Parent** process creates **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier** (**pid**)
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate



OPERATING SYSTEMS

Process creation using fork()

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - **fork()** system call creates new process
  - **exec()** system call used after a **fork()** to replace the process' memory space with a new program



OPERATING SYSTEMS

C Program forking Separate Process

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```

OPERATING SYSTEMS

Process Termination

- Process executes last statement and then asks the operating system to delete it using the **exit()** system call.
  - Returns status data from child to parent (via **wait()**)
  - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the **abort()** system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

OPERATING SYSTEMS

Process Termination

- Some operating systems do not allow child to exist if its parent has terminated. If a process terminates, then all its children must also be terminated.
  - **cascading termination.** All children, grandchildren, etc. are terminated.
  - The termination is initiated by the operating system.
- The parent process may wait for termination of a child process by using the **wait()** system call**.** The call returns status information and the pid of the terminated process
  - **pid = wait(&status);**
- If no parent waiting (did not invoke **wait()**) process is a **zombie**
- If parent terminated without invoking **wait** , process is an **orphan**

OPERATING SYSTEMS

Process Termination

- What happens if parent process terminates before child process
- What happens if child process terminates before parent(i.e When the parent process in sleep)
- How to know the state of the process?

THANK YOU

Suresh Jamadagni
Department of Computer Science Engineering

sureshjamadagni@pes.edu

UE20CS254

Operating Systems

Suresh Jamadagni

Department of Computer Science and Engineering

Operating Systems

System calls for Process Management

Suresh Jamadagni

Department of Computer Science and Engineering

Process Identifiers

- Every process has a unique **process ID**, a non-negative integer
- The process ID is the only well-known identifier of a process that is always unique
- It is often used as a piece of other identifiers, to guarantee uniqueness.
- Although unique, process IDs are reused. As processes terminate, their IDs become candidates for reuse.
- Most UNIX systems implement algorithms to delay reuse, so that newly created processes are assigned IDs different from those used by processes that terminated recently

OPERATING SYSTEMS

fork()

- An existing process can create a new one by calling the **fork** function

```
#include <unistd.h>
pid_t fork(void);
```

- The new process created by fork is called the child process
- Return value in the child is 0
- Return value in the parent is the process ID of the new child
- Return value is -1 on error
- The child is a copy of the parent. The child gets a copy of the parent's data space, heap, and stack

OPERATING SYSTEMS

exit()

- A process can terminate normally in five ways
    1. Executing a return from the main function
    2. Calling the **exit** function.
    3. Calling the _exit or _Exit function.
    4. Executing a return from the start routine of the last thread in the process.
    5. Calling the pthread_exit function from the last thread in the process

A process can terminate abnormally in three ways

1. Calling abort
2. When the process receives certain signals
3. The last thread responds to a cancellation request

- Regardless of how a process terminates, the same code in the kernel is eventually executed.
- This kernel code closes all the open descriptors for the process, releases the memory that it was using

OPERATING SYSTEMS

wait() and waitpid()

- A process that calls wait or waitpid can
  - Block, if all of its children are still running
  - Return immediately with the termination status of a child, if a child has terminated and is waiting for its termination status to be fetched
  - Return immediately with an error, if it doesn't have any child processes
- If the process is calling wait because it received the SIGCHLD signal, wait will return immediately. But if we call it at any random point in time, it can block.

```
#include <sys/wait.h>
pid_t wait(int *statloc);
pid_t waitpid(pid_t pid, int *statloc, int options);
```

asn't changed or −1 on failure

OPERATING SYSTEMS

waitid()

- waitid() allows a process to specify which children to wait for.
- Instead of encoding this information in a single argument combined with the process ID or process group ID, two separate arguments are used

```
#include <sys/wait.h>

int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

- Returns: 0 if OK, −1 on error

OPERATING SYSTEMS

exec()

- When a process calls one of the exec functions, that process is completely replaced by the new program, and the new program starts executing at its main function.
- The process ID does not change across an exec, because a new process is not created;
- exec merely replaces the current process — its text, data, heap, and stack segments — with a brand-new program from disk.

```
#include <unistd.h>

int execl(const char *pathname, const char *arg0, ... /* (char *)0 */ );

int execv(const char *pathname, char *const argv[]);

int execle(const char *pathname, const char *arg0, ...
           /* (char *)0, char *const envp[] */ );

int execve(const char *pathname, char *const argv[], char *const envp[]);

int execlp(const char *filename, const char *arg0, ... /* (char *)0 */ );

int execvp(const char *filename, char *const argv[]);

int fexecve(int fd, char *const argv[], char *const envp[]);
```

- Return: −1 on error, no return on success

OPERATING SYSTEMS

getpid() and getppid()

```
#include <unistd.h>
pid_t getpid(void);


pid_t getppid(void);
```

- **getpid**() returns the process ID (PID) of the calling process.

- **getppid**() returns the process ID of the parent of the calling process.
- This will be either the ID of the process that created this process using **fork**() or if that process has already terminated, the ID of the process to which this process has been re-parented

OPERATING SYSTEMS

Race condition

- A race condition occurs when multiple processes are trying to do something with shared data and the final outcome depends on the order in which the processes run.
- The fork function is a lively breeding ground for race conditions, if the logic after the fork either explicitly or implicitly depends on whether the parent or child runs first after the fork.
- In general, which process runs first cannot be predicted
- A process that wants to wait for a child to terminate must call one of the wait functions.
- If a process wants to wait for its parent to terminate, a loop of the following form could be used:

```
while (getppid() != 1)
    sleep(1);
```

- The problem with this type of loop, called polling, is that it wastes CPU time, as the caller is awakened every second to test the condition.
- To avoid race conditions and to avoid polling, some form of signaling is used between multiple processes

THANK YOU

Suresh Jamadagni
Department of Computer Science and Engineering

sureshjamadagni@pes.edu

OPERATING SYSTEMS

CPU Scheduling

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Slides Credits for all the PPTs of this course

- The slides/diagrams in this course are an **adaptation**, **combination**, and **enhancement** of material from the following resources and persons:

1. Slides of Operating System Concepts, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne - 9th edition 2013 and some slides from 10th edition 2018
2. Some conceptual text and diagram from Operating Systems - Internals and Design Principles, William Stallings, 9th edition 2018
3. Some presentation transcripts from A. Frank – P. Weisberg
4. Some conceptual text from Operating Systems: Three Easy Pieces, Remzi Arpaci-Dusseau, Andrea Arpaci Dusseau

OPERATING SYSTEMS

CPU Scheduling

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

CPU Scheduling - Basic Concepts

- In a system with a single CPU core, only one process can run at a time. Others must wait until the CPU is free and can be rescheduled.
- The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.
- Several processes are kept in memory at one time.
- When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process. This pattern continues.
- Every time one process has to wait, another process can take over use of the CPU. On a multicore system, this concept of keeping the CPU busy is extended to all processing cores on the system.

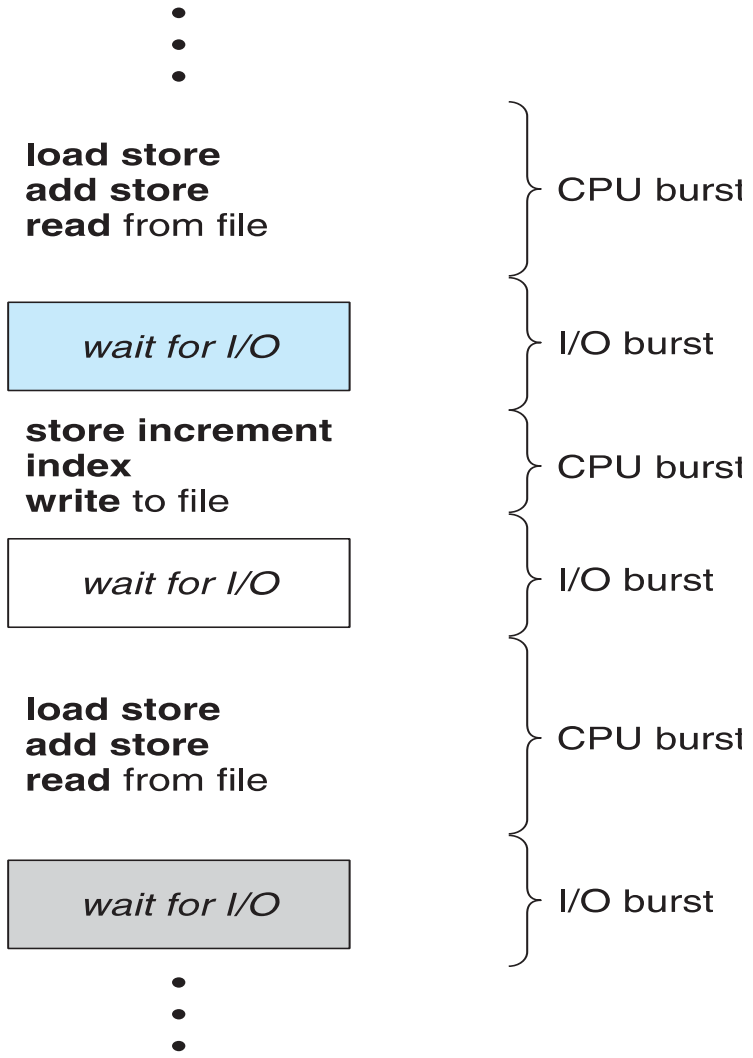
CPU Scheduling - Basic Concepts (Contd)

- A process is executed until it must wait, typically for the completion of some I/O request.
- In a simple computer system, the CPU then just sits idle. All this waiting time is wasted; no useful work is accomplished. With multiprogramming, we try to use this time productively.
- Scheduling of this kind is a fundamental operating-system function.

OPERATING SYSTEMS

Alternating Sequence of CPU and I/O bursts

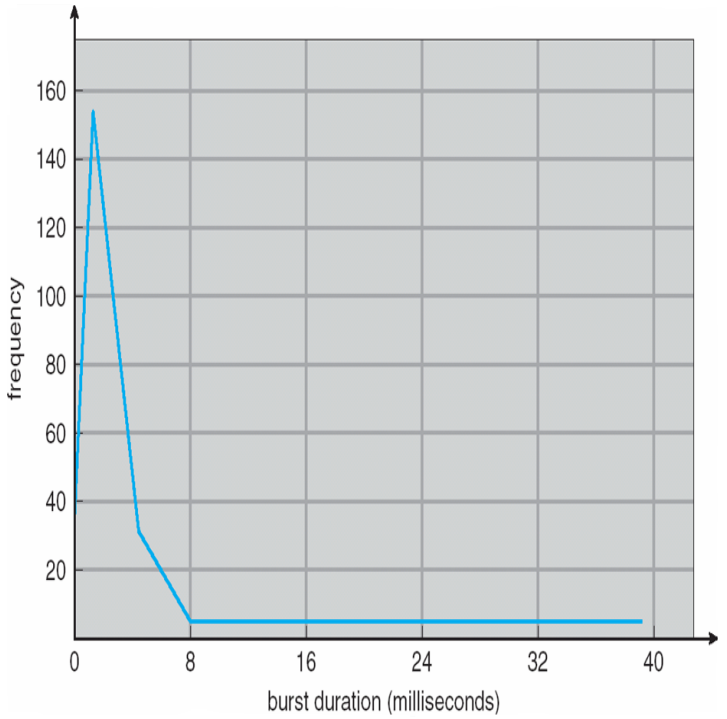
- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a **cycle** of CPU execution and I/O wait
- **CPU burst** followed by **I/O burst**
- CPU burst distribution is of main concern



OPERATING SYSTEMS

Histogram of CPU-burst Times

- The durations of CPU bursts have been measured extensively. Although they vary greatly from process to process and from computer to computer, they tend to have a frequency curve similar to that shown in the Figure.
- An I/O-bound program typically has many short CPU bursts. A CPU-bound program might have a few long CPU bursts. This distribution can be important when implementing a CPU-scheduling algorithm.



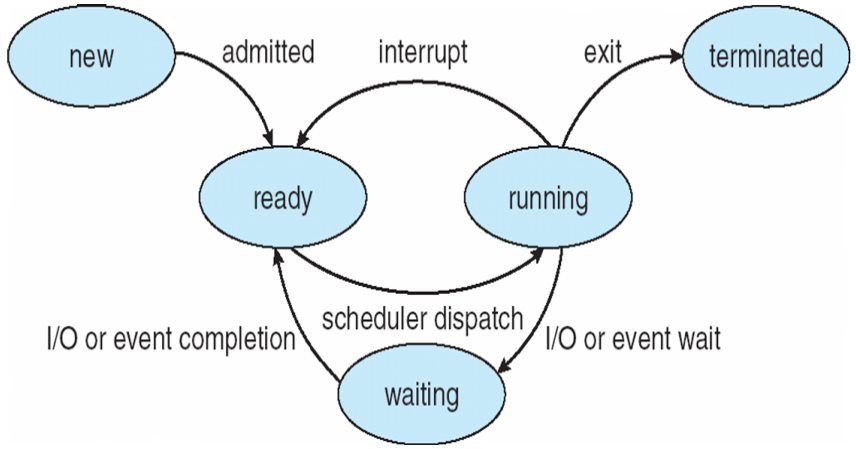
CPU Scheduler

- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
  - Queue may be ordered in various ways
  - It can be FIFO, Priority, queue, tree, unordered linked list
  - The records in the queue are PCB's of the processes

OPERATING SYSTEMS

Preemptive Scheduling

- CPU scheduling decisions may take place when a process
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- Scheduling under 1 and 4 is **non-preemptive**
- All other scheduling is **preemptive**
  - Consider access to shared data
  - Consider preemption while in kernel mode
  - Consider interrupts occurring during crucial OS activities
- Scheduling algorithms used in windows3.x, non-preemptive
- Win 95 onwards used preemptive
- Preemptive Scheduling Algorithm is used in Macintosh OS



Preemptive vs Non-Preemptive Scheduling

- Unfortunately, pre-emptive scheduling can result in race conditions when data are shared among several processes. Ex: While one process is updating the shared data, it is pre-empted so that the second process can run. The second process then tries to read the data, which are in an inconsistent state.

- A pre-emptive kernel requires mechanisms such as mutex locks to prevent race conditions when accessing shared kernel data structures. Most modern operating systems are now fully pre-emptive when running in kernel mode.

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running

Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process (performance metric)
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

OPERATING SYSTEMS

Scheduling Algorithm Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

THANK YOU

Suresh Jamadagni
Department of Computer Science Engineering

sureshjamadagni@pes.edu

OPERATING SYSTEMS

Scheduling Algorithms

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Slides Credits for all PPTs of this course

- The slides/diagrams in this course are an **adaptation**, **combination**, and **enhancement** of material from the following resources and persons:

1. Slides of Operating System Concepts, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne - 9th edition 2013 and some slides from 10th edition 2018
2. Some conceptual text and diagram from Operating Systems - Internals and Design Principles, William Stallings, 9th edition 2018
3. Some presentation transcripts from A. Frank – P. Weisberg
4. Some conceptual text from Operating Systems: Three Easy Pieces, Remzi Arpaci-Dusseau, Andrea Arpaci Dusseau

OPERATING SYSTEMS

FCFS and SJF Scheduling

Suresh Jamadagni
Department of Computer Science

First- Come, First-Served (FCFS) Scheduling

Process	Burst Time
P1	24
P2	3
P3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3

  The Gantt Chart for the schedule is:

Waiting time for P_1 = 0, P_2 = 24, P_3 = 27

- Average waiting time: (0 + 24 + 27)/3 = 17

FCFS Scheduling (Cont.)

- Suppose that the processes arrive in the order: P_2 , P_3 , P_1

  The Gantt Chart for the schedule is:

- Waiting time for $P_1 = 6$; $P_2 = 0$, $P_3 = 3$
- Average waiting time: (6 + 0 + 3)/3 = 3
- ***The average waiting time under an FCFS policy is generally not minimal and may vary substantially if the processes' CPU burst times vary greatly***

FCFS Scheduling (Cont.)

- **Convoy effect** - short process behind long process
  - Consider one CPU-bound and many I/O-bound processes
- FCFS scheduling algorithm is non-preemptive. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU
- FCFS algorithm is thus particularly troublesome for time-sharing systems, where it is important that each process to get a share of the CPU at regular intervals.
- It is not desirable to allow one process to keep the CPU for an extended period

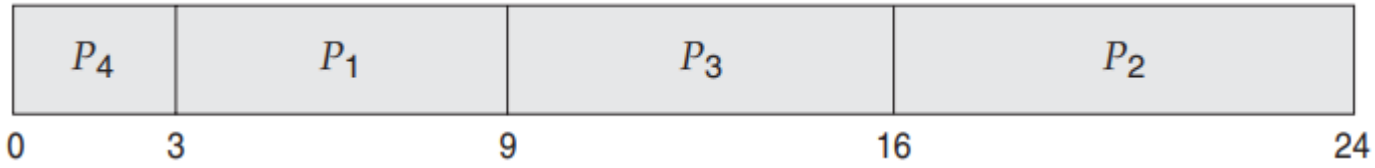
Shortest-Job-First (SJF) Scheduling

- Associate with each process the **length of its next CPU burst**
  - Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request
  - Compute an approximation of the length of the next CPU burst

Example of SJF Scheduling

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

- SJF scheduling chart



- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

Note: If FCFS scheduling is used, average waiting time = (0 + 6 + 14 + 21) / 4 = 10.25 ms.

Determining Length of Next CPU Burst

- Can be estimated using the length of previous CPU bursts, using exponential averaging

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n$$

where t_{n+1} is the predicted value for the next CPU burst,

The parameter α controls the relative weight of recent and past history in the prediction .

Commonly, α = 1/2, so recent history and past history are equally weighted

The value of t_n contains most recent information

The value τ_n stores the past history

- Preemptive version is called **shortest-remaining-time-first**

Determining Length of Next CPU Burst

Calculate the exponential averaging with $T_1 = 10$, $\alpha = 0.5$ and the algorithm is SJF with previous runs as 8, 7, 4, 16.

Initially $T_1 = 10$ and $\alpha = 0.5$ and the run times given are 8, 7, 4, 16 as it is shortest job first,

So the possible order in which these processes would serve will be 4, 7, 8, 16 since SJF is a non-preemptive technique.

So, using formula: $T_2 = \alpha * t_1 + (1-\alpha)T_1$

we have,

$T_2 = 0.5*4 + 0.5*10 = 7$, here $t_1 = 4$ and $T_1 = 10$

$T_3 = 0.5*7 + 0.5*7 = 7$, here $t_2 = 7$ and $T_2 = 7$

$T_4 = 0.5*8 + 0.5*7 = 7.5$, here $t_3 = 8$ and $T_3 = 7$

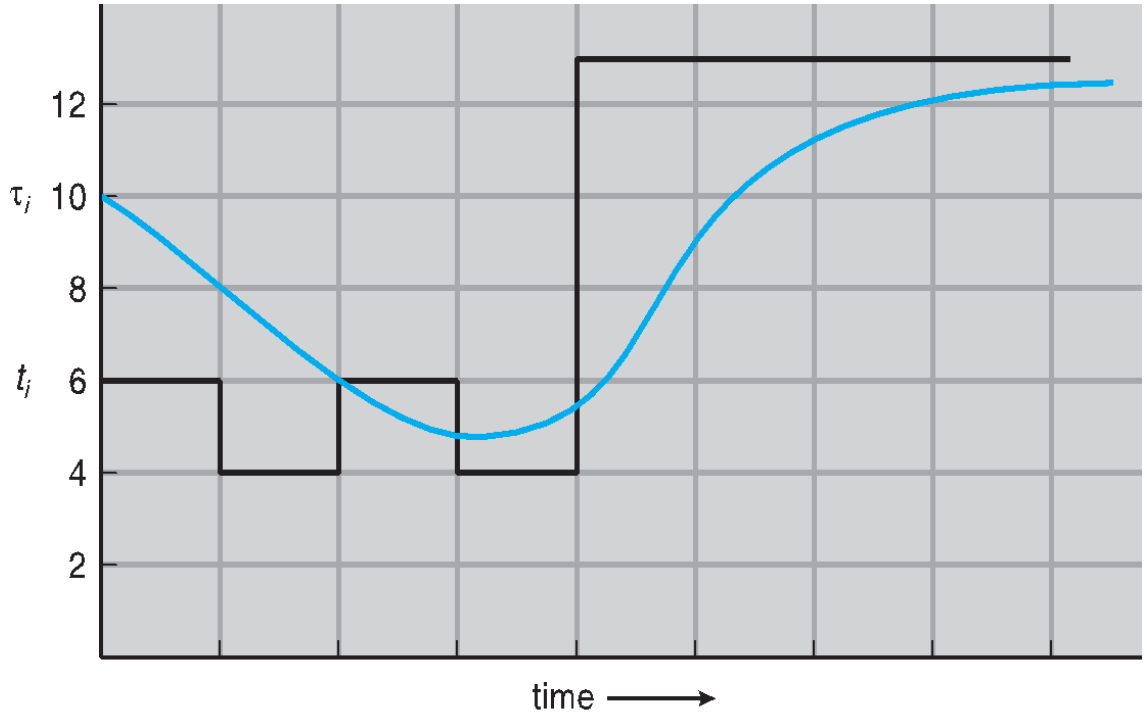
$T_5 = 0.5*16 + 0.5*7.5 = 11.8$, here $t_4 = 16$ and $T_4 = 7.5$

So the prediction for 5th burst time will be $T_5 = 11.8$

Src: https://www.geeksforgeeks.org/shortest-job-first-cpu-scheduling-with-predicted-burst-time/

OPERATING SYSTEMS

Prediction of the Length of the Next CPU Burst



$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\tau_n$$

CPU burst (t_i)		6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

OPERATING SYSTEMS

Examples of Exponential Averaging

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n$$

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Most recent CPU burst does not count
- $\alpha = 1$
  - $\tau_{n+1} = \alpha\, t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:
  - $\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\alpha\, t_{n-1} + \ldots$
  - $+(1 - \alpha)^j \alpha\, t_{n-j} + \ldots$
  - $+(1 - \alpha)^{n+1} \tau_0$
- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

OPERATING SYSTEMS

More examples

Suppose that the following processes arrive for execution at the times indicated. Each process will run for the amount of time listed. In answering the questions, use non-preemptive scheduling, and base all decisions on the information you have at the time the decision must be made.

Process	Burst Time
P_1	8
P_2	4
P_3	1

- What is the average wait time for these processes with the FCFS scheduling algorithm?

  Average wait time = (0 + 8 + 12)/3 = 6.67

- What is the average wait time for these processes with the SJF scheduling algorithm?

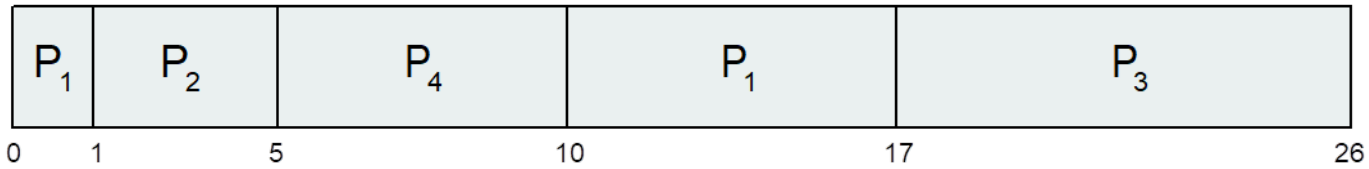
  Average wait time = (5 + 1 + 0)/3 = 2

Example of Shortest-remaining-time-first

- Preemptive SJF Scheduling is sometimes called SRTF
- Now we add the concepts of varying arrival times and preemption to the analysis

Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- *Preemptive* SJF Gantt Chart



- Average waiting time = [(10-1)+(1-1)+(17-2)+(5-3)]/4 = 26/4 = 6.5 msec

THANK YOU

Suresh Jamadagni
Department of Computer Science Engineering

sureshjamadagni@pes.edu

OPERATING SYSTEMS

Scheduling Algorithms

Chandravva Hebbi
Department of Computer Science

OPERATING SYSTEMS

Slides Credits for all PPTs of this course

- The slides/diagrams in this course are an **adaptation**, **combination**, and **enhancement** of material from the following resources and persons:

1. Slides of Operating System Concepts, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne - 9th edition 2013 and some slides from 10th edition 2018
2. Some conceptual text and diagram from Operating Systems - Internals and Design Principles, William Stallings, 9th edition 2018
3. Some presentation transcripts from A. Frank – P. Weisberg
4. Some conceptual text from Operating Systems: Three Easy Pieces, Remzi Arpaci-Dusseau, Andrea Arpaci Dusseau

OPERATING SYSTEMS

Priority and Round Robin Scheduling

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

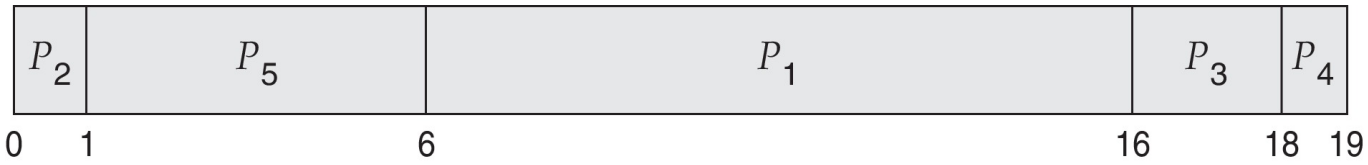
Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of the process

Example of Priority Scheduling

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Priority Scheduling Gantt chart



- Average waiting time = (6 + 0 + 16 + 18 + 1) / 5 = 41/5 = 8.2

Round-Robin Scheduling

- Round-robin (RR) scheduling algorithm is designed especially for timesharing systems.
- It is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes
- A small unit of time, called a time quantum or time slice, is defined.
- A time quantum is generally from 10 to 100 milliseconds in length
- The ready queue is treated as a circular queue
- The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum

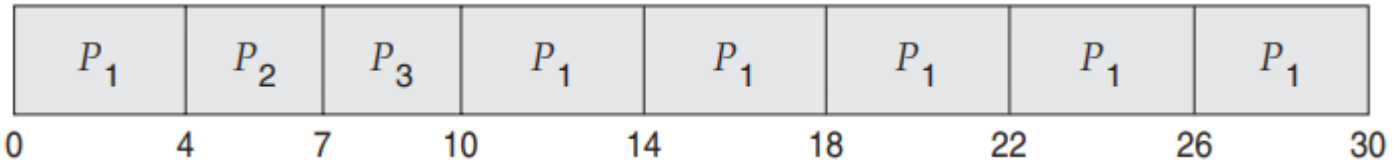
OPERATING SYSTEMS

Round-Robin Scheduling Example

- Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	24
P_2	3
P_3	3

- RR scheduling Gantt chart using a time quantum of 4 milliseconds



- P1 wa
- P2 waits for 4 milliseconds
- P3 waits for 7 milliseconds.

- The average waiting time = (6 + 4 + 7)/3 = 5.66 milliseconds

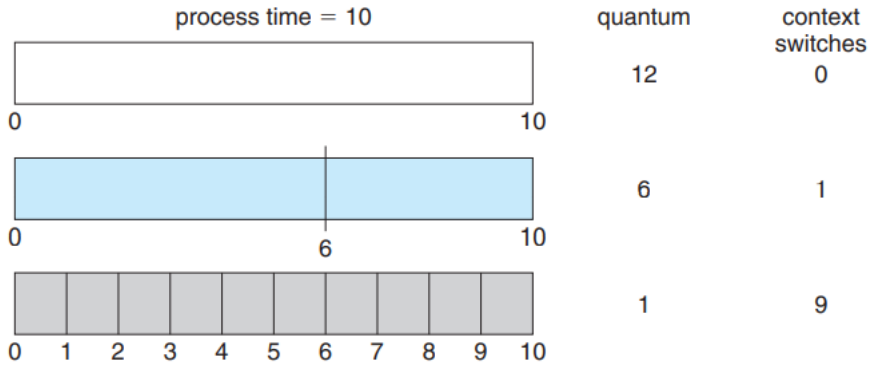
Round-Robin Scheduling Performance

- If there are n processes in the ready queue and the time quantum is q, then each process gets 1/n of the CPU time in chunks of at most q time units.
- Each process must wait no longer than (n − 1) × q time units until its next time quantum.
  - For example, with five processes and a time quantum of 20 milliseconds, each process will get up to 20 milliseconds every 100 milliseconds.
- Performance of the RR algorithm depends heavily on the size of the time quantum
- If the time quantum is extremely large, the RR policy is the same as the FCFS policy
- If the time quantum is extremely small, the RR approach can result in a large number of context switches

OPERATING SYSTEMS

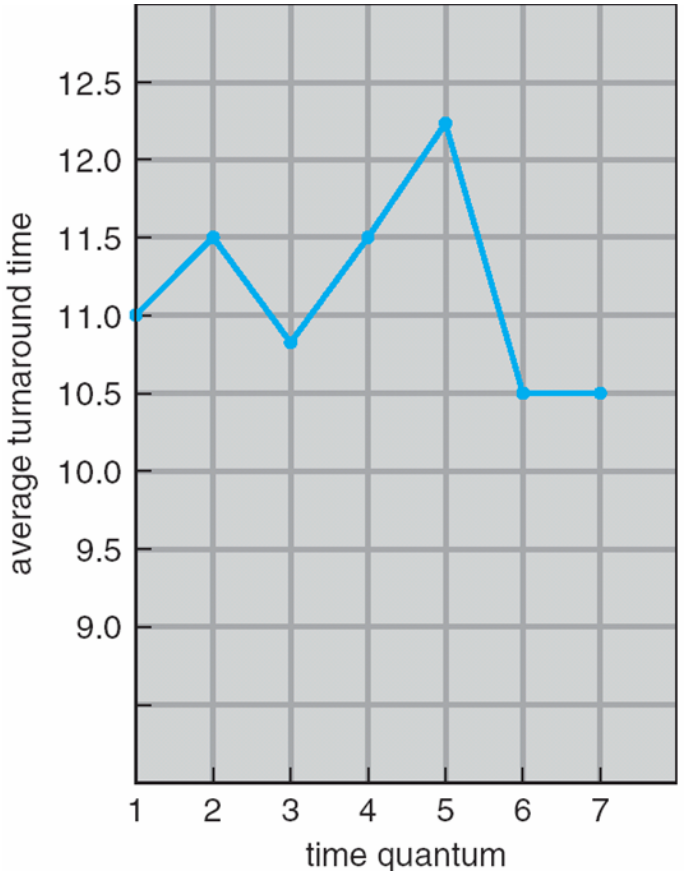
Example of Round-Robin Scheduling Performance

- Consider only one process of 10 time units.
- If the quantum is 12 time units, the process finishes in less than 1 time quantum, with no overhead.
- If the quantum is 6 time units, the process requires 2 quanta, resulting in one context switch.
- If the time quantum is 1 time unit, then nine context switches will occur, slowing the execution of the process accordingly



- In practice, most modern systems have time quanta ranging from 10 to 100 milliseconds.
- The time required for a context switch is typically less than 10 microseconds. Thus, the context-switch time is a small fraction of the time quantum.

Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

80% of CPU bursts should be shorter than the time quantum

THANK YOU

Suresh Jamadagni
Department of Computer Science Engineering

sureshjamadagni@pes.edu

OPERATING SYSTEMS

Scheduling Algorithms

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Slides Credits for all the PPTs of this course

- The slides/diagrams in this course are an **adaptation**, **combination**, and **enhancement** of material from the following resources and persons:

1. Slides of Operating System Concepts, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne - 9th edition 2013 and some slides from 10th edition 2018
2. Some conceptual text and diagram from Operating Systems - Internals and Design Principles, William Stallings, 9th edition 2018
3. Some presentation transcripts from A. Frank – P. Weisberg
4. Some conceptual text from Operating Systems: Three Easy Pieces, Remzi Arpaci-Dusseau, Andrea Arpaci Dusseau

OPERATING SYSTEMS

Multi-level Queue & Feedback Queue Scheduling

Suresh Jamadagni

Department of Computer Science

OPERATING SYSTEMS

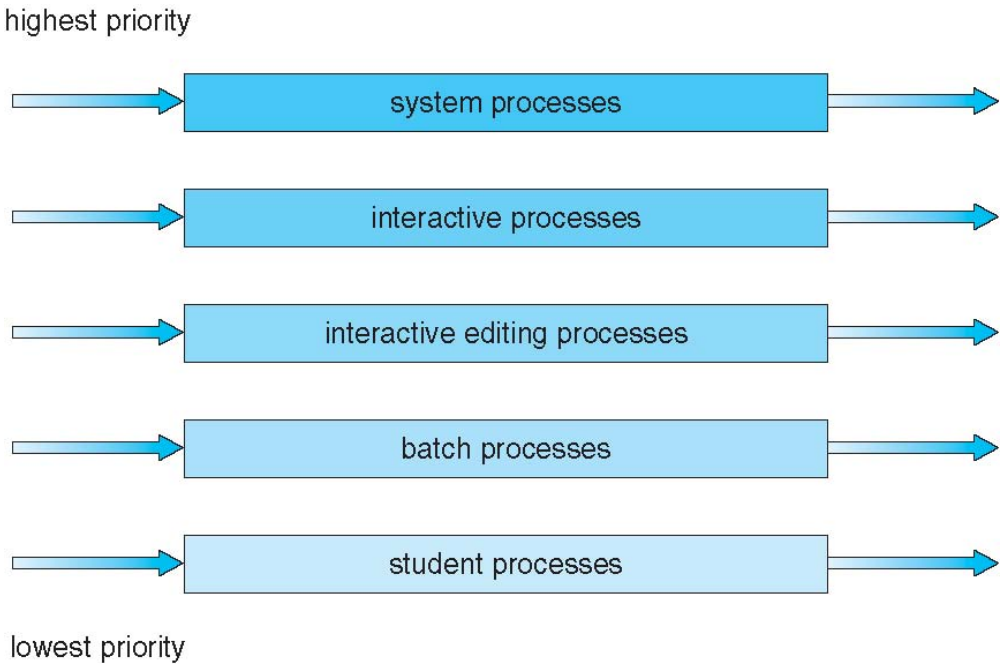
Multilevel Queue

- Ready queue is partitioned into separate queues:
  - **foreground** (interactive)
  - **background** (batch)
- Process permanently assigned to a queue
- Each queue has its own scheduling algorithm:
  - foreground – RR
  - background – FCFS
- In addition, scheduling must be done between the queues
  - Fixed priority scheduling; (serve all from foreground then from background). Possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR, 20% to background in FCFS

OPERATING SYSTEMS

Multilevel Queue Scheduling

- Each queue has absolute priority over lower-priority queues
- No process in the batch queue, for example, could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty
- If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted.



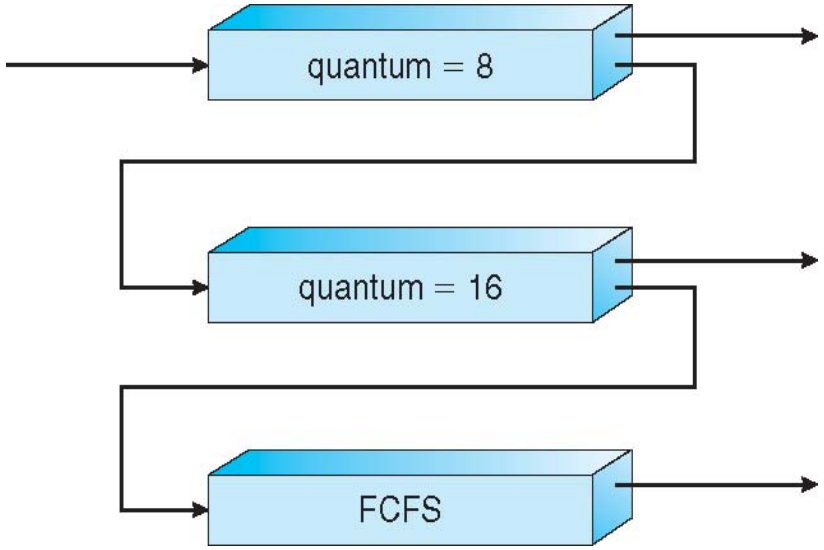
Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - Number of queues
  - Scheduling algorithms for each queue
  - Method used to determine when to upgrade a process
  - Method used to determine when to demote a process
  - Method used to determine which queue a process will enter when that process needs service

OPERATING SYSTEMS

Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS



- Scheduling
  - A new job enters queue Q_0 which is served FCFS
    - When it gains CPU, job receives 8 milliseconds
    - If it does not finish in 8 milliseconds, job is moved to queue Q_1
  - At Q_1 job is again served FCFS and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_2

OPERATING SYSTEMS

Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS

  - The scheduler first executes all processes in queue 0.
  - Only when queue 0 is empty will it execute processes in queue 1.
  - Processes in queue 2 will be executed only if queues 0 and 1 are empty.
  - A process that arrives for queue 1 will preempt a process in queue 2.
  - A process in queue 1 will in turn be preempted by a process arriving in queue 0.

OPERATING SYSTEMS

Multiple-processor Scheduling

If multiple CPUs are available, **load sharing** becomes possible, but scheduling problems become correspondingly more complex.

1. **Asymmetric multiprocessing**
   - CPU scheduling in a multiprocessor system has all scheduling decisions, I/O processing, and other system activities handled by a single processor—the master server.
   - The other processors execute only user code.
   - Asymmetric multiprocessing is simple because only one processor accesses the system data structures, reducing the need for data sharing.
2. **Symmetric multiprocessing (SMP)**
   - Each processor is self-scheduling. All processes may be in a common ready queue, or each processor may have its own private queue of ready processes.
   - All modern operating systems support SMP

OPERATING SYSTEMS

Processor Affinity

- When a process has been running on a specific processor, the data most recently accessed by the process populate the cache of the processor.
- Successive memory accesses by the process are often satisfied in cache memory.
- If the process migrates to another processor, the contents of cache memory must be invalidated for the first processor and the cache for the second processor must be repopulated.
- Because of the high cost of invalidating and repopulating caches, most SMP systems try to avoid migration of processes from one processor to another and instead attempt to keep a process running on the same processor. This is known as **processor affinity**
- **Soft affinity** - OS will attempt to keep a process on a single processor, but it is possible for a process to migrate between processors
- **Hard affinity** – OS provides system calls for process to specify a subset of processors on which it may run

OPERATING SYSTEMS

Load Balancing

- On SMP systems, it is important to keep the workload balanced among all processors to fully utilize the benefits of having more than one processor.
- **Load balancing** attempts to keep the workload evenly distributed across all processors in an SMP system.
- Load balancing is typically necessary only on systems where each processor has its own private queue of eligible processes to execute

1. **Push migration** - a specific task periodically checks the load on each processor and if it finds an imbalance, evenly distributes the load by moving (or pushing) processes from overloaded to idle or less-busy processors
2. **Pull migration** occurs when an idle processor pulls a waiting task from a busy processor.

- Push and pull migration need not be mutually exclusive

THANK YOU

Suresh Jamadagni
Department of Computer Science Engineering

sureshjamadagni@pes.edu

OPERATING SYSTEMS

Case Study: Linux/ Windows Scheduling Policies.

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Slides Credits for all PPTs of this course

- The slides/diagrams in this course are an **adaptation**, **combination**, and **enhancement** of material from the following resources and persons:

1. Slides of Operating System Concepts, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne - 9th edition 2013 and some slides from 10th edition 2018
2. Some conceptual text and diagram from Operating Systems - Internals and Design Principles, William Stallings, 9th edition 2018
3. Some presentation transcripts from A. Frank – P. Weisberg
4. Some conceptual text from Operating Systems: Three Easy Pieces, Remzi Arpaci-Dusseau, Andrea Arpaci Dusseau

OPERATING SYSTEMS

Case Study: Linux/ Windows Scheduling Policies

Suresh Jamadagni

Department of Computer Science

OPERATING SYSTEMS

Linux Scheduling Through Version 2.5

- Process Scheduling in Linux
- Linux kernel ran a variation of standard UNIX scheduling algorithm
- It did not support for SMP systems
- It had poor performance for larger processes

Linux Scheduling Version 2.5

- Kernel moved to constant order *O*(1) scheduling time
- Supported SMP systems - processor affinity and load balancing between processors with good performance
- Poor response times for the interactive processes that are common on many desktop computer systems

OPERATING SYSTEMS

Linux Scheduling in Version 2.6.23 +

- Completely Fair Scheduler (CFS) is the default scheduling algorithm.
- Based on **Scheduling classes**
  - Each class is assigned a specific priority.
  - Using different scheduling classes, the kernel can accommodate different scheduling algorithms
  - Scheduler picks the highest priority task in the highest scheduling class
  - Two scheduling classes are included, others can be added
    1. A scheduling class with CFS algorithm
    2. A real-time scheduling class

Linux Scheduling in Version 2.6.23 + (Cont.)

Completely Fair Scheduler (CFS)

- CFS scheduler assigns a **proportion of CPU processing time** to each task.
- Proportion calculated based on **nice value** which ranges from -20 to +19
  - A numerically lower nice value indicates a higher relative priority.
  - Tasks with lower nice values receive a higher proportion of CPU processing time than tasks with higher nice values
- Calculates **target latency** – interval of time during which task should run at least once
  - Proportions of CPU time are allocated from the value of targeted latency computed.
  - Target latency can increase if number of active tasks increase

Linux Scheduling in Version 2.6.23 + (Cont.)

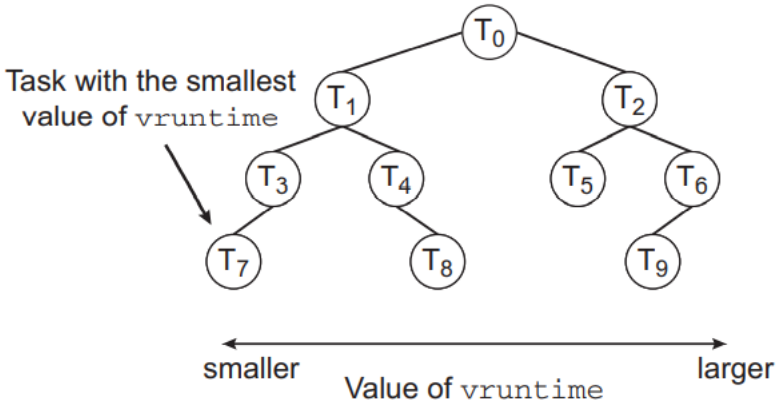
Completely Fair Scheduler (CFS)

- CFS scheduler doesn't directly assign priorities
- CFS scheduler maintains per task **virtual run time** in variable **vruntime**
  - Associated with decay factor based on priority of task – lower priority is higher decay rate
  - Normal default priority yields virtual run time = actual run time
- To decide next task to run, scheduler picks task with lowest virtual run time

Linux Scheduling in Version 2.6.23 + (Cont.)

Completely Fair Scheduler (CFS)

- Each runnable task is placed in a balanced binary search tree whose key is based on the value of **vruntime**
- When a task becomes runnable, it is added to the tree
- When a task is not runnable, it is deleted from the tree
- Navigating the tree to discover the task to run (leftmost node) will require O(lg N) operations (where N is the number of nodes in the tree).



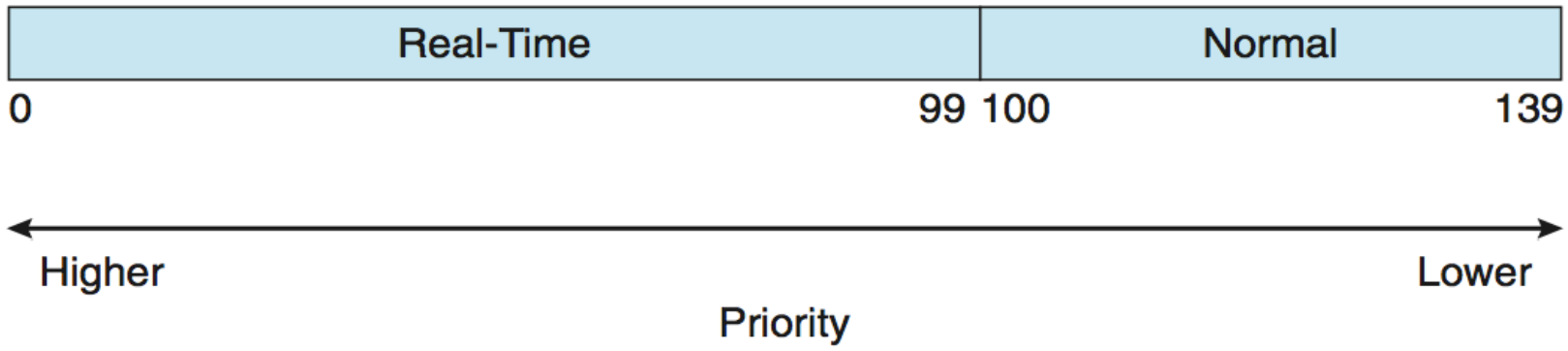
Linux Scheduling in Version 2.6.23 + (Cont.)

Completely Fair Scheduler (CFS)

- Assume that two tasks have the same nice values.
- One task is I/O-bound and the other is CPU-bound
- The value of **vruntime** will be lower for the I/O-bound task than for the CPU-bound task, giving the I/O-bound task higher priority than the CPU-bound task.
- If the CPU-bound task is executing when the I/O-bound task becomes eligible to run, the I/O-bound task will preempt the CPU-bound task.

Linux Real Time Scheduling

- Real-time scheduling according to POSIX.1b
  - Real-time tasks have static priorities (0-99)
- Real-time plus normal map into global priority scheme
- Nice value of -20 maps to global priority 100
- Nice value of +19 maps to priority 139



OPERATING SYSTEMS

Windows Scheduling

- Uses priority based pre-emptive scheduling algorithm
- Scheduler ensures that the **highest-priority thread** will always run
- Windows kernel that handles scheduling is called the **dispatcher**
- Thread selected by the dispatcher runs until it is preempted by a higher-priority thread or until it terminates or until its time quantum ends or until it calls a blocking system call
- Dispatcher uses a 32-level priority scheme
  - **Variable class** is 1-15, **real-time class** is 16-31
  - Priority 0 is memory-management thread
- Queue for each priority class
- If no run-able thread, runs **idle thread**

Windows Priority Classes

- Windows API identifies several priority classes to which a process can belong
  - REALTIME_PRIORITY_CLASS, HIGH_PRIORITY_CLASS, ABOVE_NORMAL_PRIORITY_CLASS,NORMAL_PRIORITY_CLASS, BELOW_NORMAL_PRIORITY_CLASS, IDLE_PRIORITY_CLASS
  - All are variable except REALTIME
- A thread within a given priority class has a relative priority
  - TIME_CRITICAL, HIGHEST, ABOVE_NORMAL, NORMAL, BELOW_NORMAL, LOWEST, IDLE
- Priority class and relative priority combine to give numeric priority
- Base priority is NORMAL within the class
- If quantum expires, priority lowered, but never below base

OPERATING SYSTEMS

Windows Thread Priorities

- A thread within a given priority class also has a relative priority
- Priority of each thread is based on both the priority class it belongs to (top row in the diagram) and its relative priority within that class (left column in the diagram)

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Windows Priority Classes (Cont.)

- If wait occurs, priority boosted depending on what was waited for
- Windows distinguishes between foreground process and background processes
- Foreground processes given 3x priority boost
- This priority boost gives the foreground process three times longer to run before a time-sharing preemption occurs.

THANK YOU

Suresh Jamadagni
Department of Computer Science Engineering

sureshjamadagni@pes.edu

Operating Systems

UE22CS242B

Suresh Jamadagni
Department of Computer Science and Engineering

Operating Systems

Bash shell and cron

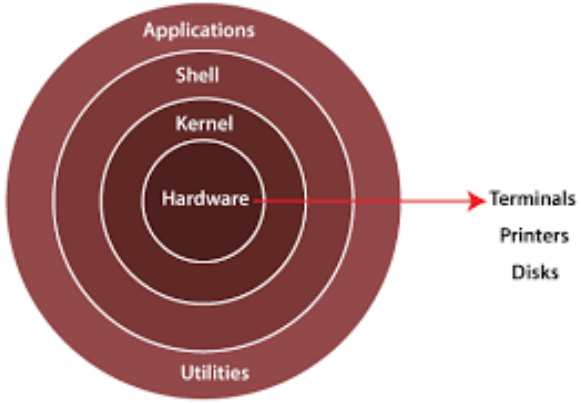
Suresh Jamadagni

Department of Computer Science and Engineering

OPERATING SYSTEMS

What is a shell?

- The **shell** is the Linux command line interpreter.
- Shell provides an interface between the user and the kernel and executes programs called commands.
- For example, if a user enters ***ls*** then the shell executes the *ls* command.
- The shell can also execute other programs such as applications, scripts, and user programs



OPERATING SYSTEMS

What is a shell?

- Instructions entered in response to the shell prompt have the following syntax: **command [arg1] [arg2] .. [argn]**
- The brackets [] indicate that the arguments are optional. Many commands can be executed with or without arguments
- The shell parses the words or tokens (commandname , options, filesnames[s]) and gets the kernel to execute the commands assuming the syntax is correct.
- Typically, the shell processes the complete line after a carriage return (cr) (carriage return) is entered and finds the program that the command line wants executing.
- The shell looks for the command to execute either in the specified directory if given (./mycommand) or it searches through a list of directories depending on your $PATH variable.

OPERATING SYSTEMS

Various shells

- There are a number of shells available to a Linux user:

  sh - This is known as the Borne Shell and is the original shell

  csh, tcsh - These are well-known and widely used derivatives of the Borne shell

  ksh - The popular Korn shell

  bash - The Borne Again SHell is the most popular shell used for linux and developed by GNU

echo $SHELL

```
ubuntu@ip-172-31-41-208:~$ echo $SHELL
/bin/bash
```

OPERATING SYSTEMS

Environment variables

- **Environmental variables** are variables that are defined for the current shell and are inherited by any child shells or processes.
- Environmental variables are used to pass information into processes that are spawned from the shell.
- **Shell variables** are variables that are contained exclusively within the shell in which they were set or defined.
- They are often used to keep track of ephemeral data, like the current working directory.
- By convention, these types of variables are usually defined using all capital letters. This helps users distinguish environmental variables within other contexts.
- We can see a list of all of our environmental variables by using the ***env*** or ***printenv***

OPERATING SYSTEMS

Common Environment variables

- **SHELL**: This describes the shell that will be interpreting any commands you type in. In most cases, this will be bash by default, but other values can be set if you prefer other options.
- **TERM**: This specifies the type of terminal to emulate when running the shell. Different hardware terminals can be emulated for different operating requirements. You usually won't need to worry about this though.
- **USER**: The current logged in user.
- **PWD**: The current working directory.
- **OLDPWD**: The previous working directory. This is kept by the shell in order to switch back to your previous directory by running cd -.
- **PATH**: A list of directories that the system will check when looking for commands. When a user types in a command, the system will check directories in this order for the executable.
- **HOME**: The current user's home directory.

OPERATING SYSTEMS

Common SHELL variables

- **BASHOPTS**: The list of options that were used when bash was executed. This can be useful for finding out if the shell environment will operate in the way you want it to.
- **BASH_VERSION**: The version of bash being executed, in human-readable form.
- **BASH_VERSINFO**: The version of bash, in machine-readable output.

```
ubuntu@ip-172-31-41-208:~$ echo $BASH_VERSION
5.0.17(1)-release
```

OPERATING SYSTEMS

SHELL basics

- Creating shell variables:

```
ubuntu@ip-172-31-41-208:~$ MY_VAR="Hello world"
```

- Accessing the value of any shell or environmental variable:

```
ubuntu@ip-172-31-41-208:~$ echo $MY_VAR
Hello world
```

- Spawning a child shell process

```
ubuntu@ip-172-31-41-208:~$ bash
```

- In the child shell process, the shell variable defined in the parent is not available

```
ubuntu@ip-172-31-41-208:~$ echo $MY_VAR
```

- To terminate the child shell process,

```
ubuntu@ip-172-31-41-208:~$ exit
```

- To pass the shell variables to child shell processes, you need to **export** the variable

```
ubuntu@ip-172-31-41-208:~$ export MY_VAR="Hello world"
ubuntu@ip-172-31-41-208:~$ bash
ubuntu@ip-172-31-41-208:~$ echo $MY_VAR
Hello world
```

PES UNIVERSITY
CELEBRATING 50 YEARS

OPERATING SYSTEMS

SHELL basics

- Removing a shell variable

```
ubuntu@ip-172-31-41-208:~$ unset MY_VAR
ubuntu@ip-172-31-41-208:~$ echo $MY_VAR

ubuntu@ip-172-31-41-208:~$
```

- For setting environment variables at login, edit the **.profile** file in the **$HOME** directory and add the export command

```
ubuntu@ip-172-31-41-208:~$ cd $HOME
ubuntu@ip-172-31-41-208:~$ vi .profile
```

OPERATING SYSTEMS

SHELL control flow - if

```
if commands; then
    commands
[elif commands; then
    commands...]
[else
    commands]
fi
```

Example: To check if the file exists

```
#!/bin/bash

if [ -f welcome.txt ]; then
   echo "File exists"
else
   echo "File does not exist"
fi
```

OPERATING SYSTEMS

SHELL control flow - loops

```
#!/bin/bash

for i in {1..5}
do
   echo "i = $i"
done
```

Nested loop:

```
#!/bin/bash

for i in {1..5}
do

   for j in {1..3}
   do
      echo "i = $i and j = $j"
   done
done
```

OPERATING SYSTEMS

SHELL control flow – continue and break

```
#!/bin/bash

for i in {1..5}
do

   if [ $i -eq 3 ]
   then
      continue
   fi

   for j in {1..3}
   do
      echo "i = $i and j = $j"
   done
done
```

```
#!/bin/bash

for i in {1..5}
do

   if [ $i -eq 3 ]
   then
      break
   fi

   for j in {1..3}
   do
      echo "i = $i and j = $j"
   done
done
```

OPERATING SYSTEMS

cron

- In Unix and Linux, **cron** is a daemon, which is an unattended program that runs continuously in the background and wakes up (executes) to handle periodic service requests when required. The daemon runs when the system boots
- Cron is a job scheduling utility present in Unix like systems. The **crond** daemon enables cron functionality and runs in background.
- The cron reads the **crontab** (cron tables) for running predefined scripts.
- By using a specific syntax, you can configure a cron job to schedule scripts or other commands to run automatically.
- Checking for jobs scheduled in **cron**

```
ubuntu@ip-172-31-41-208:~$ crontab -l
no crontab for ubuntu
```

OPERATING SYSTEMS

cron

- Starting/checking status of **cron**

```
ubuntu@ip-172-31-41-208:~$ sudo systemctl status cron.service
● cron.service - Regular background program processing daemon
     Loaded: loaded (/lib/systemd/system/cron.service; enabled; vendor preset: enabled)
     Active: active (running) since Wed 2024-01-24 06:44:21 UTC; 1h 8min ago
```

- Adding jobs to the scheduler
- **crontab -e**: edits crontab entries to add, delete, or edit cron jobs.

```
*   *   *   *   *  sh /path/to/script/script.sh
|   |   |   |   |             |
|   |   |   |   |      Command or Script to Execute
|   |   |   |   |
|   |   |   |   |
|   |   |   |   |
|   |   |   | Day of the Week(0-6)
|   |   |   |
|   |   | Month of the Year(1-12)
|   |   |
|   | Day of the Month(1-31)
|   |
| Hour(0-23)
|
Min(0-59)
```

OPERATING SYSTEMS

cron examples

SCHEDULE	SCHEDULED VALUE
5 0 * 8 *	At 00:05 in August.
5 4 * * 6	At 04:05 on Saturday.
0 22 * * 1-5	At 22:00 on every day-of-week from Monday through Friday.

PES UNIVERSITY
CELEBRATING 50 YEARS

OPERATING SYSTEMS

cron examples

Executing a program every minute

crontab -l

```
# m h  dom mon dow   command
*/1 * * * * /bin/bash $HOME/OS/hello.exe &> $HOME/OS/cron_output.log
```

```
ubuntu@ip-172-31-41-208:~/OS$ tail /var/log/syslog
Jan 24 08:38:01 ip-172-31-41-208 CRON[2923]: (ubuntu) CMD ($HOME/OS/hello.exe)
Jan 24 08:38:01 ip-172-31-41-208 CRON[2920]: (CRON) info (No MTA installed, discarding output)
Jan 24 08:39:01 ip-172-31-41-208 CRON[2927]: (ubuntu) CMD ($HOME/OS/hello.exe)
Jan 24 08:39:01 ip-172-31-41-208 CRON[2926]: (CRON) info (No MTA installed, discarding output)
Jan 24 08:39:09 ip-172-31-41-208 crontab[2911]: (ubuntu) REPLACE (ubuntu)
Jan 24 08:39:09 ip-172-31-41-208 crontab[2911]: (ubuntu) END EDIT (ubuntu)
Jan 24 08:39:41 ip-172-31-41-208 crontab[2932]: (ubuntu) LIST (ubuntu)
Jan 24 08:40:01 ip-172-31-41-208 cron[665]: (ubuntu) RELOAD (crontabs/ubuntu)
Jan 24 08:40:01 ip-172-31-41-208 CRON[2935]: (ubuntu) CMD (/bin/bash $HOME/OS/hello.exe &> $HOME/OS/cron_output.log)
Jan 24 08:41:01 ip-172-31-41-208 CRON[2939]: (ubuntu) CMD (/bin/bash $HOME/OS/hello.exe &> $HOME/OS/cron_output.log)
```

THANK YOU

Suresh Jamadagni
Department of Computer Science and Engineering

sureshjamadagni@pes.edu