

## Part I: Overview

# Chapter 1: Introduction



Operating System Concepts – 10<sup>th</sup> Edition

Silberschatz, Galvin and Gagne ©2018



## Outline

- What Operating Systems Do
- Computer-System Organization
- Computer-System Architecture
- Operating-System Operations
- Resource Management
- Security and Protection
- Virtualization
- Distributed Systems
- Kernel Data Structures
- Computing Environments
- Free/Libre and Open-Source Operating Systems



Operating System Concepts – 10<sup>th</sup> Edition

1.2

Silberschatz, Galvin and Gagne ©2018



## Objectives

- Describe the general organization of a computer system and the role of interrupts
- Describe the components in a modern, multiprocessor computer system
- Illustrate the transition from user mode to kernel mode
- Discuss how operating systems are used in various computing environments
- Provide examples of free and open-source operating systems



## What Does the Term Operating System Mean?

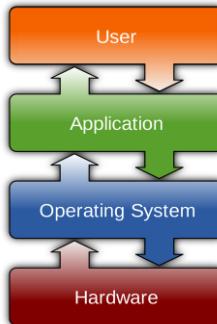
- An operating system is “fill in the blanks”
- What about:
  - Program
  - Hardware
  - Compiler
  - Etc.





## What is an Operating System?

- A program that acts as an intermediary between a user of a computer and the computer hardware
- 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



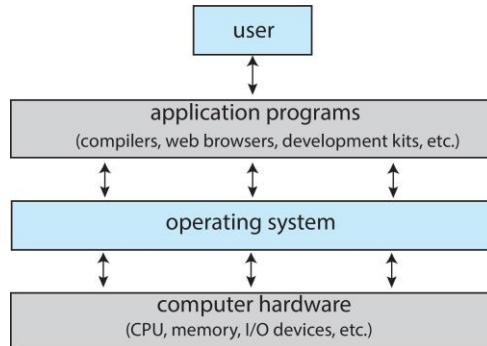
## Computer System Structure

- Computer system can be divided into four components:
  - 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





## Abstract View of Components of Computer



## What Operating Systems Do

- Depends on the point of view
- 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
  - Operating system is a **resource allocator** and **control program** making efficient use of HW and managing execution of user programs





## What Operating Systems Do (Cont.)

- Users of dedicated systems such as **workstations** have dedicated resources but frequently use shared resources from **servers**
- Mobile devices like smartphones and tablets are resource poor, optimized for usability and battery life
  - Mobile user interfaces such as touch screens, voice recognition
- Some computers have little or no user interface, such as embedded computers in devices and automobiles
  - Run primarily without user intervention



## Term OS Covers Many Roles

- Because of myriad designs and uses of OSes
- Present in toasters through ships, spacecraft, game machines, TVs and industrial control systems
- Born when fixed use computers for military became more general purpose and needed resource management and program control





## Operating System Definition

- No universally accepted definition
- “Everything a vendor ships when you order an operating system” is a good approximation
  - But varies wildly
- “The one program running at all times on the computer” is the **kernel**, which is part of the operating system
- Everything else is either
  - A **system program** (ships with the operating system, but not part of the kernel) , or
  - An **application program**, all programs not associated with the operating system
- Today's OSes for general purpose and mobile computing also include **middleware** – a set of software frameworks that provide addition services to application developers such as databases, multimedia, graphics



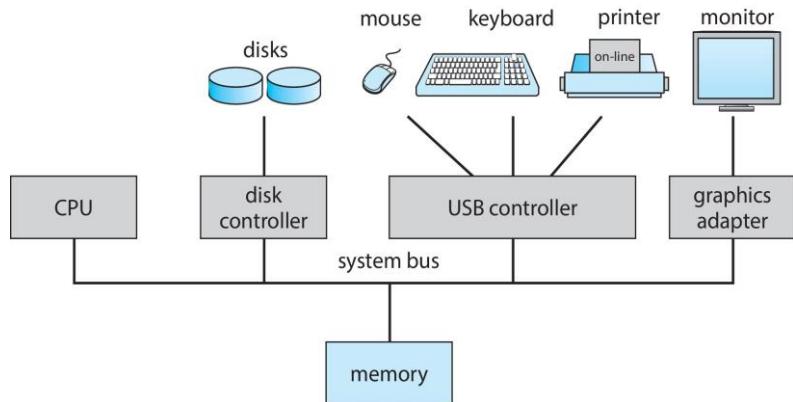
## Overview of Computer System Structure





## Computer System Organization

- Computer-system operation
  - One or more CPUs, device controllers connect through common **bus** providing access to shared memory
  - Concurrent execution of CPUs and devices competing for memory cycles



Operating System Concepts – 10<sup>th</sup> Edition

1.13

Silberschatz, Galvin and Gagne ©2018



## 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 type has an operating system **device driver** to manage it
- 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 System Concepts – 10<sup>th</sup> Edition

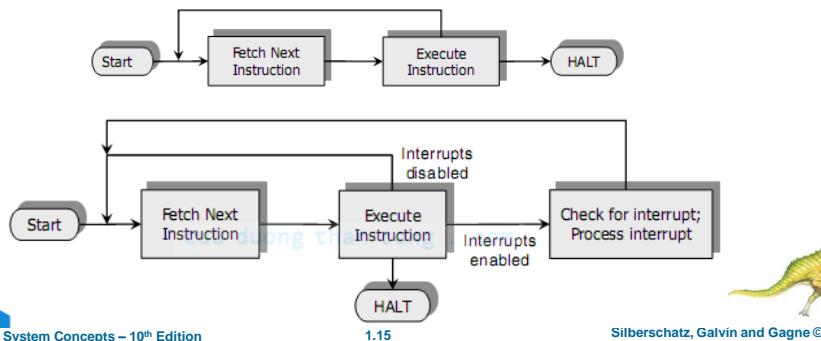
1.14

Silberschatz, Galvin and Gagne ©2018



## Common Functions of Interrupts

- 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
- An operating system is **interrupt driven**



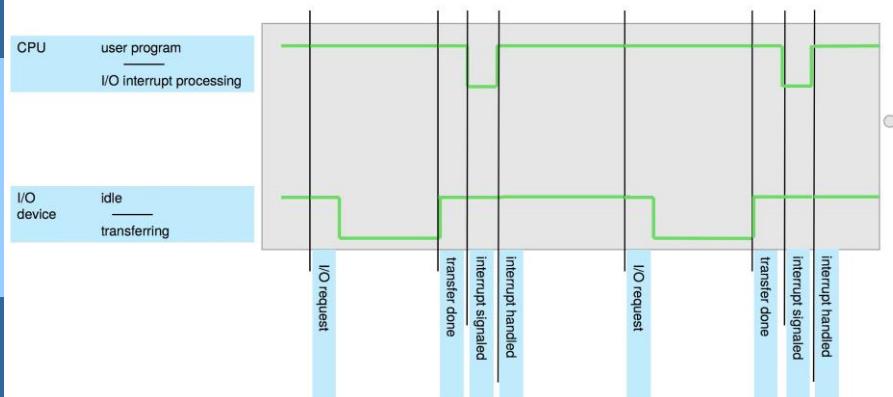
Operating System Concepts – 10<sup>th</sup> Edition

1.15

Silberschatz, Galvin and Gagne ©2018



## Interrupt Timeline



Operating System Concepts – 10<sup>th</sup> Edition

1.16

Silberschatz, Galvin and Gagne ©2018



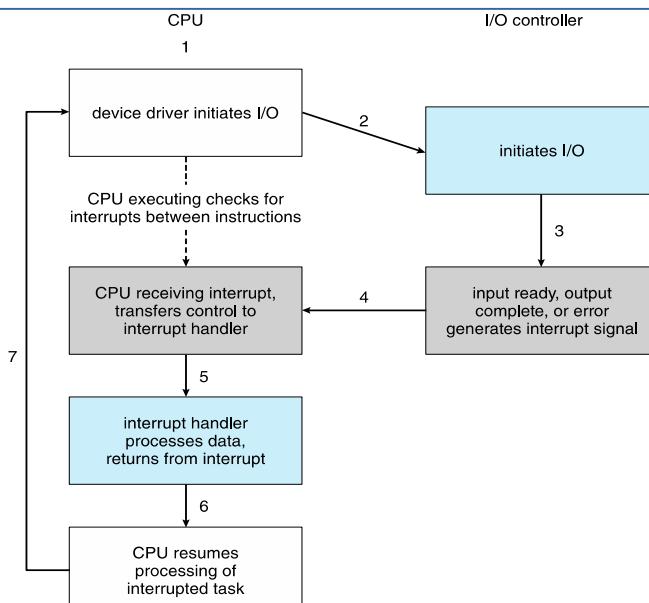


## Interrupt Handling

- The operating system preserves the state of the CPU by storing the registers and the program counter
- Determines which type of interrupt has occurred:
  - Overload (div/0)
  - Timer
  - I/O
  - Hardware failure
  - Trap (software interrupt)
- Separate segments of code determine what action should be taken for each type of interrupt



## Interrupt-drive I/O Cycle





## I/O Structure

- Two methods for handling I/O
  - After I/O starts, control returns to user program only upon I/O completion
  - After I/O starts, control returns to user program without waiting for I/O completion



## I/O Structure (Cont.)

- 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





## Storage Structure

Operating System Concepts – 10<sup>th</sup> Edition

1.22

Silberschatz, Galvin and Gagne ©2018



## Storage Structure

- Main memory – only large storage media that the CPU can access directly
  - Typically, **volatile**
  - Typically, **random-access memory** in the form of **Dynamic Random-access Memory (DRAM)**
- Secondary storage – extension of main **memory** that provides large **nonvolatile** storage capacity

Operating System Concepts – 10<sup>th</sup> Edition

1.23

Silberschatz, Galvin and Gagne ©2018





## Storage Structure (Cont.)

- **Hard Disk Drives (HDD)** – 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
- **Non-volatile memory (NVM)** devices – faster than hard disks, nonvolatile
  - Various technologies
  - Becoming more popular as capacity and performance increases, price drops



## Storage Definitions and Notation Review

The basic unit of computer storage is the **bit**. A bit can contain one of two values, 0 and 1. All other storage in a computer is based on collections of bits. Given enough bits, it is amazing how many things a computer can represent: numbers, letters, images, movies, sounds, documents, and programs, to name a few. A **byte** is 8 bits, and on most computers, it is the smallest convenient chunk of storage. For example, most computers don't have an instruction to move a bit but do have one to move a byte. A less common term is **word**, which is a given computer architecture's native unit of data. A word is made up of one or more bytes. For example, a computer that has 64-bit registers and 64-bit memory addressing typically has 64-bit (8-byte) words. A computer executes many operations in its native word size rather than a byte at a time.

Computer storage, along with most computer throughput, is generally measured and manipulated in bytes and collections of bytes. A **kilobyte**, or KB, is 1,024 bytes; a **megabyte**, or MB, is  $1,024^2$  bytes; a **gigabyte**, or GB, is  $1,024^3$  bytes; a **terabyte**, or TB, is  $1,024^4$  bytes; and a **petabyte**, or PB, is  $1,024^5$  bytes. Computer manufacturers often round off these numbers and say that a megabyte is 1 million bytes and a gigabyte is 1 billion bytes. Networking measurements are an exception to this general rule; they are given in bits (because networks move data a bit at a time).



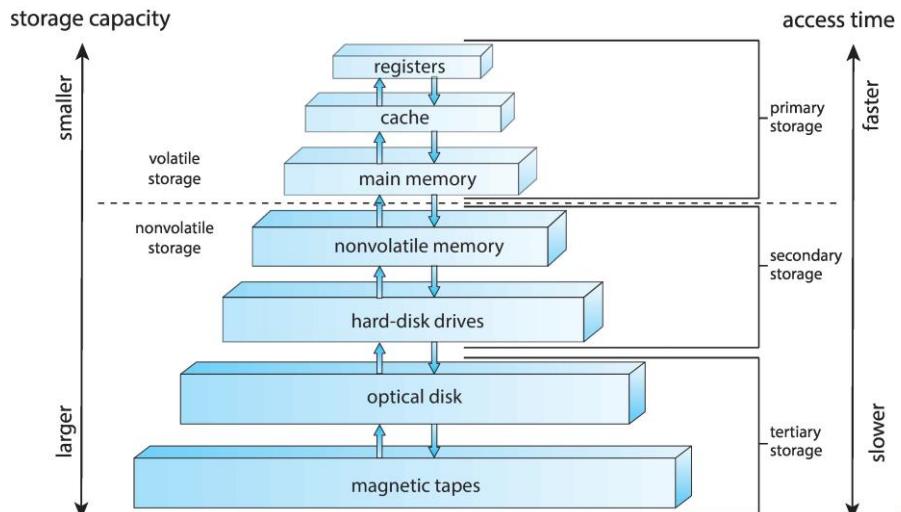


## Storage Hierarchy

- 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
- **Device Driver** for each device controller to manage I/O
  - Provides uniform interface between controller and kernel

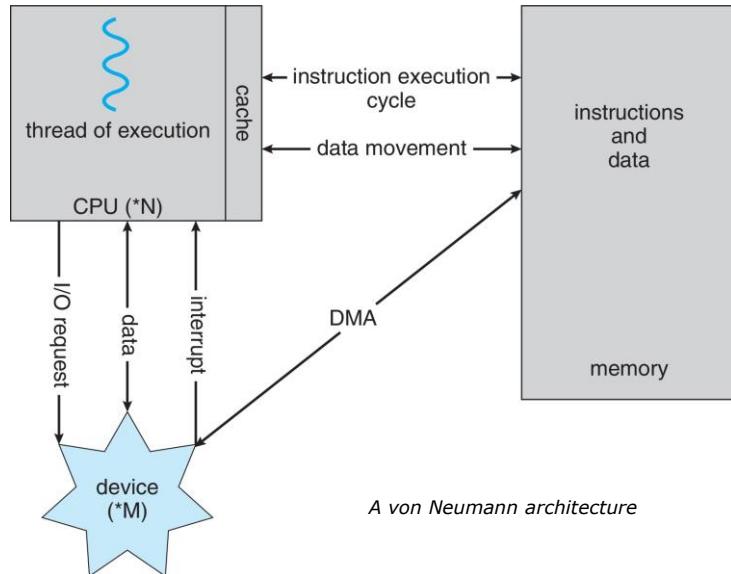


## Storage-Device Hierarchy





## How a Modern Computer Works



Operating System Concepts – 10<sup>th</sup> Edition

1.28

Silberschatz, Galvin and Gagne ©2018



## 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



Operating System Concepts – 10<sup>th</sup> Edition

1.29

Silberschatz, Galvin and Gagne ©2018



## Operating-System Operations

- Bootstrap program – simple code to initialize the system, load the kernel
- Kernel loads
- Starts **system daemons** (services provided outside of the kernel)
- Kernel **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 – **system call**
    - Other process problems include infinite loop, processes modifying each other or the operating system



## Multiprogramming (Batch system)

- Single user cannot always keep CPU and I/O devices busy
- 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 job has to wait (for I/O for example), OS switches to another job



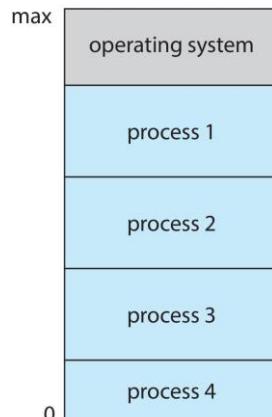


## Multitasking (Timesharing)

- A logical extension of Batch systems – the 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, which is called **process**
  - If several jobs ready to run at the same time  $\Rightarrow$  **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



## Memory Layout for Multiprogrammed System





## Dual-mode 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.
  - When a user is running → mode bit is “user”
  - When kernel code is executing → mode bit is “kernel”
- Some instructions designated as **privileged**, only executable in kernel mode



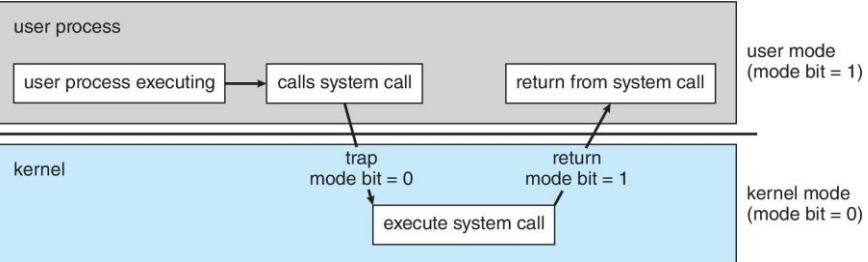
## Dual-mode Operation (Cont.)

- How do we guarantee that user does not explicitly set the mode bit to “kernel”?
- When the system starts executing it is in kernel mode
- When control is given to a user program the mode-bit changes to “user mode”.
- When a user issues a system call it results in an interrupt, which trap to the operating system. At that time, the mode-bit is set to “kernel mode”.





## Transition from User to Kernel Mode



## Timer

- Timer to prevent infinite loop (or process hogging resources)
  - Timer is set to interrupt the computer after some time period
  - Keep a counter that is decremented by the physical clock
  - Operating system set the counter (privileged instruction)
  - When counter zero generate an interrupt
  - Set up before scheduling process to regain control or terminate program that exceeds allotted time





## Process Management

- A process is a program in execution. It is a unit of work within the system. Program is a **passive entity**; process is an **active entity**.
- Process needs resources to accomplish its task
  - CPU, memory, I/O, files
  - Initialization data
- Process termination requires reclaim of any reusable resources
- Single-threaded process has one **program counter** specifying location of next instruction to execute
  - Process executes instructions sequentially, one at a time, until completion
- Multi-threaded process has one program counter per thread
- Typically, system has many processes, some user, some operating system running concurrently on one or more CPUs
  - Concurrency by multiplexing the CPUs among the processes / threads



## Process Management Activities

The operating system is responsible for the following activities in connection with process management:

- Creating and deleting both user and system processes
- Suspending and resuming processes
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication
- Providing mechanisms for deadlock handling





## Memory Management

- To execute a program all (or part) of the instructions must be in memory
- All (or part) of the data that is needed by the program must be in memory
- Memory management determines what is in memory and when
  - Optimizing CPU utilization and computer response to users
- Memory management activities
  - Keeping track of which parts of memory are currently being used and by whom
  - Deciding which processes (or parts thereof) and data to move into and out of memory
  - Allocating and deallocating memory space as needed



## File-system Management

- OS provides uniform, logical view of information storage
  - Abstracts physical properties to logical storage unit - **file**
  - Each medium is controlled by device (i.e., disk drive, tape drive)
    - Varying properties include access speed, capacity, data-transfer rate, access method (sequential or random)
- File-System management
  - Files usually organized into directories
  - Access control on most systems to determine who can access what
  - OS activities include
    - Creating and deleting files and directories
    - Primitives to manipulate files and directories
    - Mapping files onto secondary storage
    - Backup files onto stable (non-volatile) storage media





## Mass-Storage Management

- Usually, disks used to store data that does not fit in main memory or data that must be kept for a “long” period of time
- Proper management is of central importance
- Entire speed of computer operation hinges on disk subsystem and its algorithms
- OS activities
  - Mounting and unmounting
  - Free-space management
  - Storage allocation
  - Disk scheduling
  - Partitioning
  - Protection



## 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





## Characteristics of Various Types of Storage

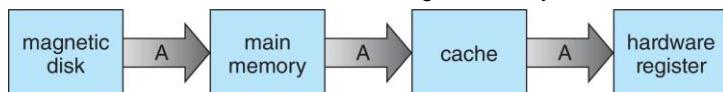
Level	1	2	3	4	5
Name	registers	cache	main memory	solid-state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25-0.5	0.5-25	80-250	25,000-50,000	5,000,000
Bandwidth (MB/sec)	20,000-100,000	5,000-10,000	1,000-5,000	500	20-150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

Movement between levels of storage hierarchy can be explicit or implicit



## Migration of data “A” from Disk to Register

- Multitasking environments must be careful to use most recent value, no matter where it is stored in the storage hierarchy



- Multiprocessor environment must provide **cache coherency** in hardware such that all CPUs have the most recent value in their cache
- Distributed environment situation even more complex
  - Several copies of a datum can exist
  - Various solutions covered in Chapter 19





## I/O Subsystem

- One purpose of OS is to hide peculiarities of hardware devices from the user
- I/O subsystem responsible for
  - Memory management of I/O including buffering (storing data temporarily while it is being transferred), caching (storing parts of data in faster storage for performance), spooling (the overlapping of output of one job with input of other jobs)
  - General device-driver interface
  - Drivers for specific hardware devices



## Protection and Security

- **Protection** – mechanism for controlling access of processes or users to resources defined by the OS
- **Security** – defense of the system against internal and external attacks
  - Huge range, including denial-of-service, worms, viruses, identity theft, theft of service





## Protection

- Systems generally first distinguish among users, to determine who can do what
  - User identities (**user IDs**, security IDs) include name and associated number, one per user
  - User ID then associated with all files, processes of that user to determine access control
  - Group identifier (**group ID**) allows set of users to be defined and controls managed, then also associated with each process, file
  - **Privilege escalation** allows user to change to effective ID with more rights



## 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



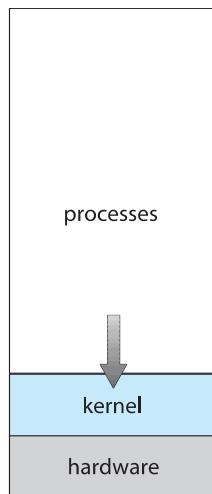


## Virtualization (cont.)

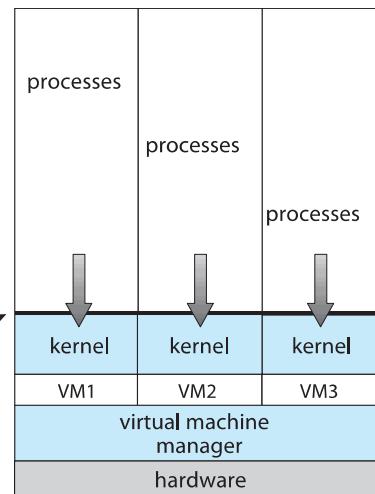
- Use cases involve laptops and desktops running multiple OSes for exploration or compatibility
  - Apple laptop running Mac OS X host, Windows as a guest
  - Developing apps for multiple OSes without having multiple systems
  - Quality assurance testing applications without having multiple systems
  - Executing and managing compute environments within data centers
- VMM can run natively, in which case they are also the host
  - There is no general-purpose host then (VMware ESX and Citrix XenServer)



## Virtualization Illustration



(a)



(b)





## Distributed Systems

- 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



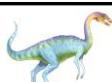
## Computer-System Architecture



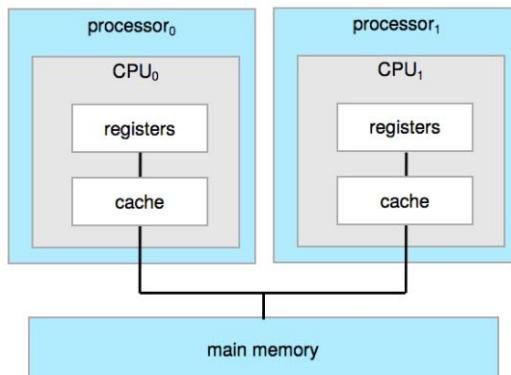


# Computer-System Architecture

- Most systems use a single general-purpose processor
  - Most systems have special-purpose processors as well
- **Multiprocessor's** systems growing in use and importance
  - Also known as **parallel systems, tightly-coupled systems**
  - Advantages include:
    1. **Increased throughput**
    2. **Economy of scale**
    3. **Increased reliability** – graceful degradation or fault tolerance
  - Two types:
    1. **Asymmetric Multiprocessing** – each processor is assigned a specific task.
    2. **Symmetric Multiprocessing** – each processor performs all tasks



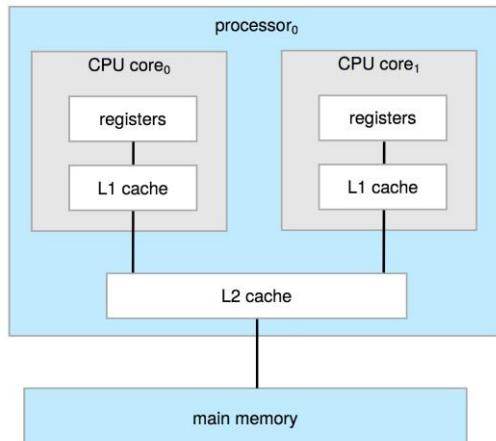
## Symmetric Multiprocessing Architecture





## Dual-Core Design

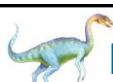
- Multi-chip and **multicore**
- Systems containing all chips
  - Chassis containing multiple separate systems



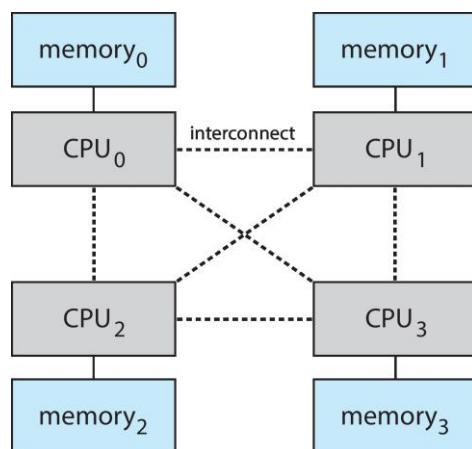
Operating System Concepts – 10<sup>th</sup> Edition

1.56

Silberschatz, Galvin and Gagne ©2018



## Non-Uniform Memory Access System



Operating System Concepts – 10<sup>th</sup> Edition

1.57

Silberschatz, Galvin and Gagne ©2018

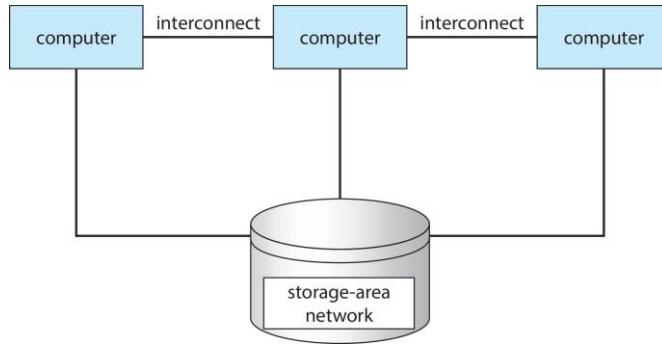


## Clustered Systems

- 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



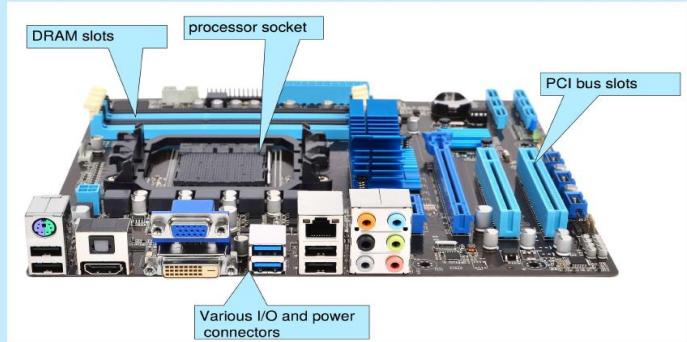
## Clustered Systems





## PC Motherboard

Consider the desktop PC motherboard with a processor socket shown below:



This board is a fully-functioning computer, once its slots are populated. It consists of a processor socket containing a CPU, DRAM sockets, PCIe bus slots, and I/O connectors of various types. Even the lowest-cost general-purpose CPU contains multiple cores. Some motherboards contain multiple processor sockets. More advanced computers allow more than one system board, creating NUMA systems.



## Computer System Environments





## Computing Environments

- Traditional
- Mobile
- Client Server
- Peer-to-Peer
- Cloud computing
- Real-time Embedded



## 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





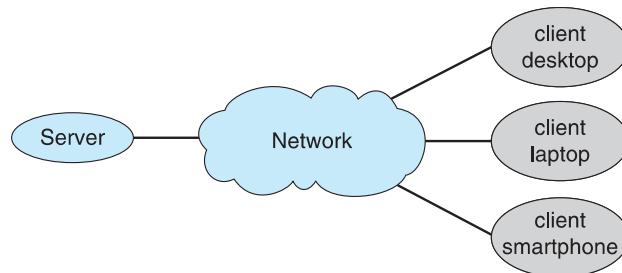
## Mobile Computing

- 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**



## Client Server Computing

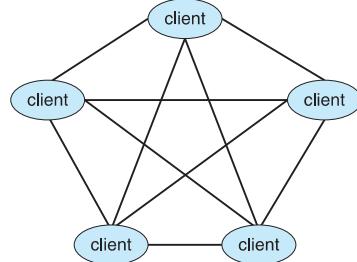
- Dumb terminals supplanted by smart PCs
- Many systems now **servers**, responding to requests generated by **clients**
  - **Compute-server system** provides an interface to client to request services (i.e., database)
  - **File-server system** provides interface for clients to store and retrieve files





## 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



## 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 its functionality.
  - Amazon **EC2** has thousands of servers, millions of virtual machines, petabytes of storage available across the Internet, pay based on usage





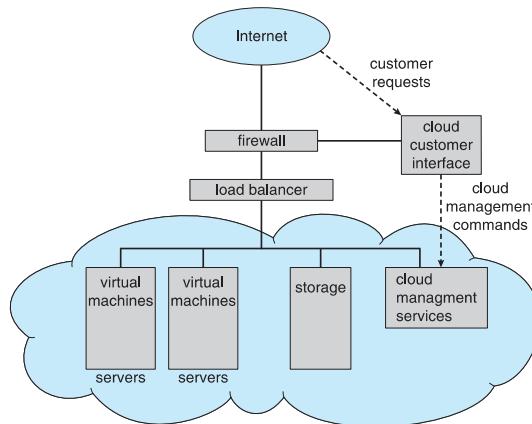
## Cloud Computing – Many Types

- **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
- 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)



## Cloud Computing (cont.)

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





## 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



## Free and Open-Source Operating Systems

- Operating systems made available in source-code format rather than just binary **closed-source** and **proprietary**
- Counter to the **copy protection** and **Digital Rights Management (DRM)** movement
- Started by **Free Software Foundation (FSF)**, which has “copyleft” **GNU Public License (GPL)**
  - Free software and open-source software are two different ideas championed by different groups of people
    - ▶ <http://gnu.org/philosophy/open-source-misses-the-point.html/>
- Examples include **GNU/Linux** and **BSD UNIX** (including core of **Mac OS X**), and many more
- Can use VMM like VMware Player (Free on Windows), Virtualbox (open source and free on many platforms - <http://www.virtualbox.com>)
  - Use to run guest operating systems for exploration





## The Study of Operating Systems

There has never been a more interesting time to study operating systems, and it has never been easier. The open-source movement has overtaken operating systems, causing many of them to be made available in both source and binary (executable) format. The list of operating systems available in both formats includes Linux, BSD UNIX, Solaris, and part of macOS. The availability of source code allows us to study operating systems from the inside out. Questions that we could once answer only by looking at documentation or the behavior of an operating system we can now answer by examining the code itself.

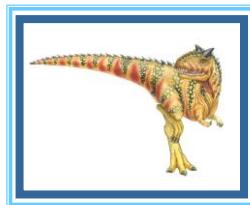
Operating systems that are no longer commercially viable have been open-sourced as well, enabling us to study how systems operated in a time of fewer CPU, memory, and storage resources. An extensive but incomplete list of open-source operating-system projects is available from [https://curlie.org/Computers/Software/Operating\\_Systems/Open\\_Source/](https://curlie.org/Computers/Software/Operating_Systems/Open_Source/)

In addition, the rise of virtualization as a mainstream (and frequently free) computer function makes it possible to run many operating systems on top of one core system. For example, VMware (<http://www.vmware.com>) provides a free "player" for Windows on which hundreds of free "virtual appliances" can run. VirtualBox (<http://www.virtualbox.com>) provides a free, open-source virtual machine manager on many operating systems. Using such tools, students can try out hundreds of operating systems without dedicated hardware.

The advent of open-source operating systems has also made it easier to make the move from student to operating-system developer. With some knowledge, some effort, and an Internet connection, a student can even create a new operating-system distribution. Just a few years ago, it was difficult or impossible to get access to source code. Now, such access is limited only by how much interest, time, and disk space a student has.



## End of Chapter 1

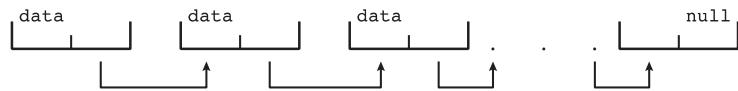




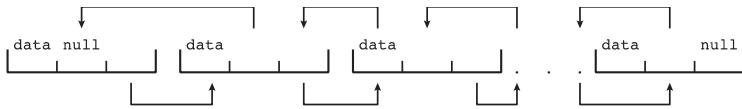
## Kernel Data Structures

- Many similar to standard programming data structures

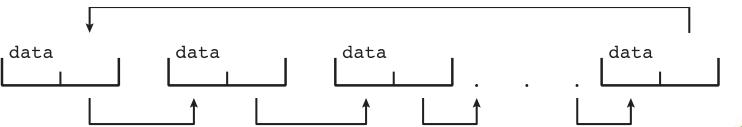
- **Singly linked list**



- **Doubly linked list**



- **Circular linked list**



Operating System Concepts – 10<sup>th</sup> Edition

1.74

Silberschatz, Galvin and Gagne ©2018

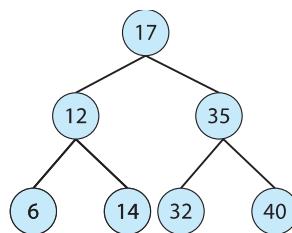


## Kernel Data Structures

- **Binary search tree**

left  $\leq$  right

- Search performance is  $O(n)$
- **Balanced binary search tree** is  $O(\lg n)$



Operating System Concepts – 10<sup>th</sup> Edition

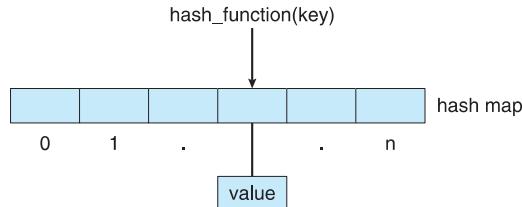
1.75

Silberschatz, Galvin and Gagne ©2018



## Kernel Data Structures

- Hash function can create a hash map



- Bitmap – string of  $n$  binary digits representing the status of  $n$  items
- Linux data structures defined in **include** files `<linux/list.h>`, `<linux/kfifo.h>`, `<linux/rbtree.h>`



## Characteristics of Various Types of Storage

Level	1	2	3	4	5
Name	registers	cache	main memory	solid-state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25-0.5	0.5-25	80-250	25,000-50,000	5,000,000
Bandwidth (MB/sec)	20,000-100,000	5,000-10,000	1,000-5,000	500	20-150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

Movement between levels of storage hierarchy can be explicit or implicit



# Chapter 1: Overview

## 1.2. Operating-System Services



Operating System Concepts – 10<sup>th</sup> Edition

Silberschatz, Galvin and Gagne ©2018



## Outline

- Operating System Services
- User and Operating System-Interface
- System Calls
- System Services
- Linkers and Loaders
- Why Applications are Operating System Specific

Operating System Concepts – 10<sup>th</sup> Edition

2a.2

Silberschatz, Galvin and Gagne ©2018





## Objectives

- Identify services provided by an operating system
- Illustrate how system calls are used to provide operating system services



## Operating System Services

- Operating systems provide an environment for execution of programs and services to programs and users
- One set of operating-system services provides functions that are helpful to the user:
  - **User interface** - Almost all operating systems have a user interface (**UI**). Varies between
    - **Command-Line (CLI)**,
    - **Graphics User Interface (GUI)**,
    - **touch-screen**,
    - **Batch**
  - **Program execution** - The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
  - **I/O operations** - A running program may require I/O, which may involve a file or an I/O device





## Operating System Services (Cont.)

- One set of operating-system services provides functions that are helpful to the user (Cont.):
  - **File-system manipulation** - The file system is of particular interest. Programs need to read and write files and directories, create and delete them, search them, list file information, permission management.
  - **Communications** – Processes may exchange information, on the same computer or between computers over a network
    - ▶ Communications may be via shared memory or through message passing (packets moved by the OS)



## Operating System Services (Cont.)

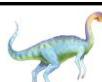
- One set of operating-system services provides functions that are helpful to the user (Cont.):
  - **Error detection** – OS needs to be constantly aware of possible errors
    - ▶ May occur in the CPU and memory hardware, in I/O devices, in user program
    - ▶ For each type of error, OS should take the appropriate action to ensure correct and consistent computing
    - ▶ Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system



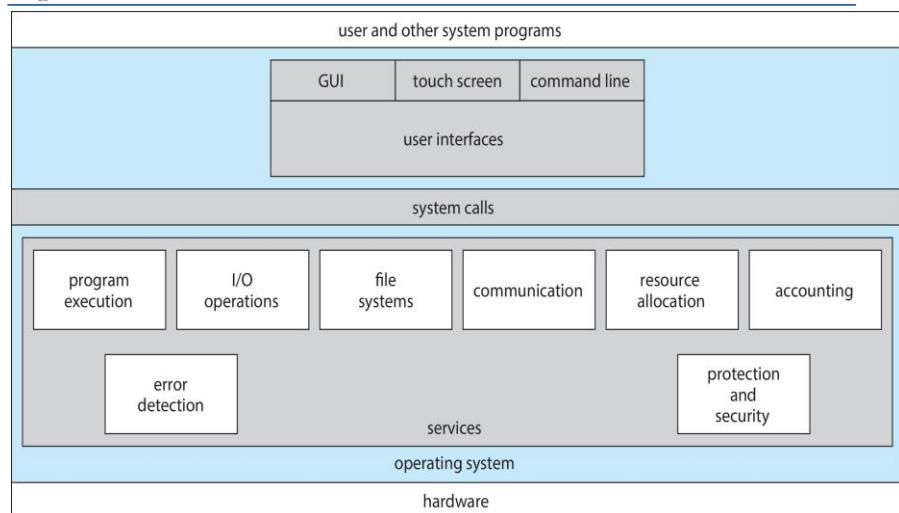


## Operating System Services (Cont.)

- Another set of OS function exists for ensuring the efficient operation of the system itself via resource sharing
  - **Resource allocation** - When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
    - Many types of resources - CPU cycles, main memory, file storage, I/O devices.
  - **Logging** - To keep track of which users use how much and what kinds of computer resources
  - **Protection and security** - The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
    - **Protection** involves ensuring that all access to system resources is controlled
    - **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts



## A View of Operating System Services





## User Operating System Interface

- CLI -- command line interpreter
  - allows direct command entry
- GUI – graphical user interface
- Touchscreen Interfaces
- Batch



## CLI

- Sometimes implemented in kernel, sometimes by systems program
- Sometimes multiple flavors implemented – **shells**
- Primarily fetches a command from user and executes it
- Sometimes commands built-in, sometimes just names of programs
  - If the latter, adding new features doesn't require shell modification





## Bourne Shell Command Interpreter

```
1. root@r6181-d5-us01:~ (ssh)
X root@r6181-d5-us01... ● 8%  ssh  ⚡ 8%  × root@r6181-d5-us01... 83
Last login: Thu Jul 14 08:47:01 on ttys002
iMacPro:~ pb9$ ssh root@r6181-d5-us01
root@r6181-d5-us01's password:
Last login: Thu Jul 14 06:01:11 2016 from 172.16.16.162
[root@r6181-d5-us01 ~]# uptime
 06:57:48 up 16 days, 10:52, 3 users, load average: 129.52, 80.33, 56.55
[root@r6181-d5-us01 ~]# df -kh
Filesystem      Size  Used Avail Use% Mounted on
/dev/mapper/vg_ks-lv_root
                  50G   19G   28G  41% /
tmpfs           127G  520K  127G  1% /dev/shm
/dev/sda1        477M  71M  381M  16% /boot
/dev/dssd0000    1.0T  480G  545G  47% /dssd_xfs
tcp://192.168.150.1:3334/oranefs
                  12T  5.7T  6.4T  47% /mnt/oranefs
/dev/gpfs-test   23T  1.1T  22T  5% /mnt/gpfs
[root@r6181-d5-us01 ~]#
[root@r6181-d5-us01 ~]# ps aux | sort -nrk 3,3 | head -n 5
root    97653 11.2  6.6 42665344 17520636 ? S<Ll Jul13 166:23 /usr/lpp/mmfs/bin/mmfsd
root    69849  6.6  0.0     0  0 ? S Jul12 181:54 [vpthread-1-1]
root    69850  6.4  0.0     0  0 ? S Jul12 177:42 [vpthread-1-2]
root    3829  3.0  0.0     0  0 ? S Jun27 730:04 [rp_thread 7:0]
root    3826  3.0  0.0     0  0 ? S Jun27 728:08 [rp_thread 6:0]
[root@r6181-d5-us01 ~]# ls -l /usr/lpp/mmfs/bin/mmfsd
-r-x----- 1 root root 20667161 Jun  3  2015 /usr/lpp/mmfs/bin/mmfsd
[root@r6181-d5-us01 ~]#
```



## GUI

- User-friendly **desktop** metaphor interface
  - Usually mouse, keyboard, and monitor
  - **Icons** represent files, programs, actions, etc.
  - Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a **folder**)
  - Invented at Xerox PARC
- Many systems now include both CLI and GUI interfaces
  - Microsoft Windows is GUI with CLI “command” shell
  - Apple Mac OS X is “Aqua” GUI interface with UNIX kernel underneath and shells available
  - Unix and Linux have CLI with optional GUI interfaces (CDE, KDE, GNOME)





## Touchscreen Interfaces

- Touchscreen devices require new interfaces
  - Mouse not possible or not desired
  - Actions and selection based on gestures
  - Virtual keyboard for text entry
- Voice commands



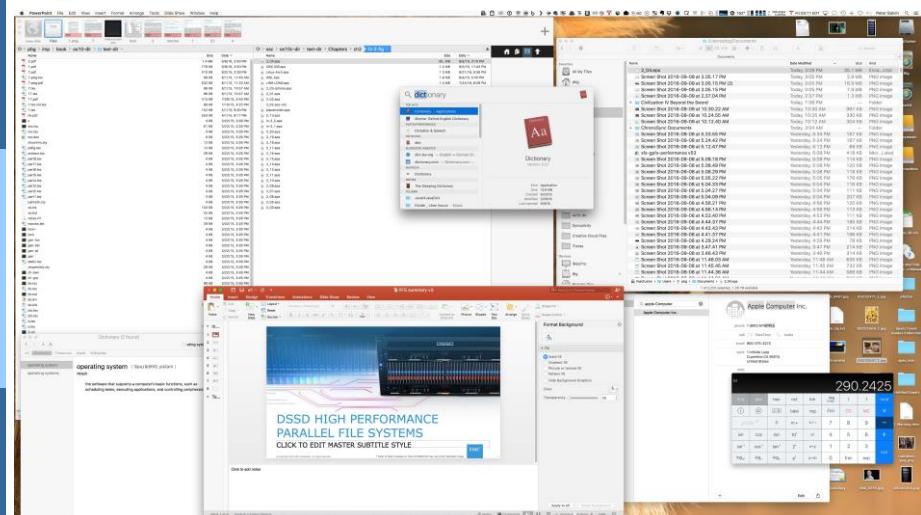
Operating System Concepts – 10<sup>th</sup> Edition

2a.13

Silberschatz, Galvin and Gagne ©2018



## The Mac OS X GUI



Operating System Concepts – 10<sup>th</sup> Edition

2a.14

Silberschatz, Galvin and Gagne ©2018



## System Calls

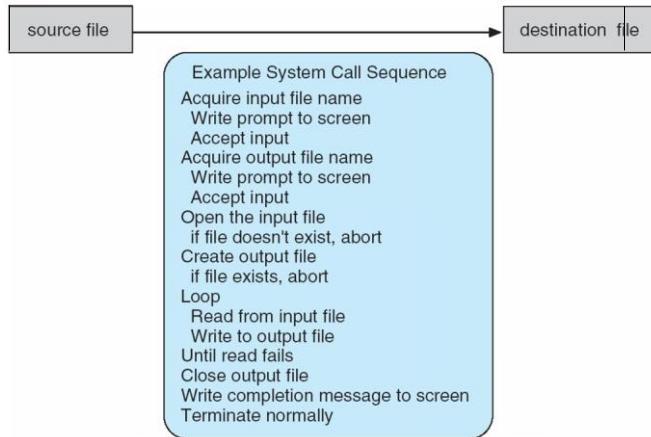
- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level **Application Programming Interface (API)** rather than direct system call use
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)

Note that the system-call names used throughout this text are generic



## Example of System Calls

- System call sequence to copy the contents of one file to another file





## Example of Standard API

### EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count)
```

return value      function name      parameters

A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `int fd`—the file descriptor to be read
- `void *buf`—a buffer into which the data will be read
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns -1.



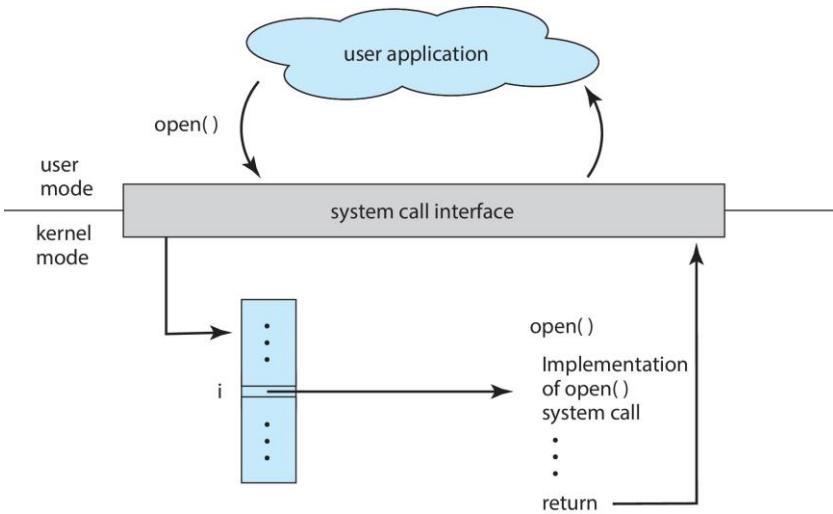
## System Call Implementation

- Typically, a number is associated with each system call
  - **System-call interface** maintains a table indexed according to these numbers
- The system call interface invokes the intended system call in OS kernel and returns status of the system call and any return values
- The caller need not know anything about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call
  - Most details of OS interface hidden from programmer by API
    - ▶ Managed by run-time support library (set of functions built into libraries included with compiler)





## API: System Call to Open a File

Operating System Concepts – 10<sup>th</sup> Edition

2a.19

Silberschatz, Galvin and Gagne ©2018

**ex**

Function	System call
open a file	open
close a file	close
perform I/O	read/write
send a signal	kill
create a pipe	pipe
create a socket	socket
duplicate a process	fork/clone
overlay a process	execl/execv
terminate a process	exit

User process

```
....  
result = open ("~/home/glass/file.txt", O_RDONLY);  
....
```

open (char\* name, int mode)  
{  
 <Place parameters in registers>  
 <Execute trap instruction, switching to kernel code and kernel mode>  
 <Return result of system call>  
}

User  
codeC runtime  
library

Kernel

```
Address of kernel close ()  
Address of kernel open ()  
Address of kernel write ()  
....  
kernel code for open ()  
{  
    <Manipulate kernel data structures>  
    ....  
    <Return to user code and user mode>  
}
```

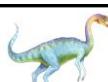
System-call  
vector tableKernel  
system-call  
codeOperating System Concepts – 10<sup>th</sup> Edition

8

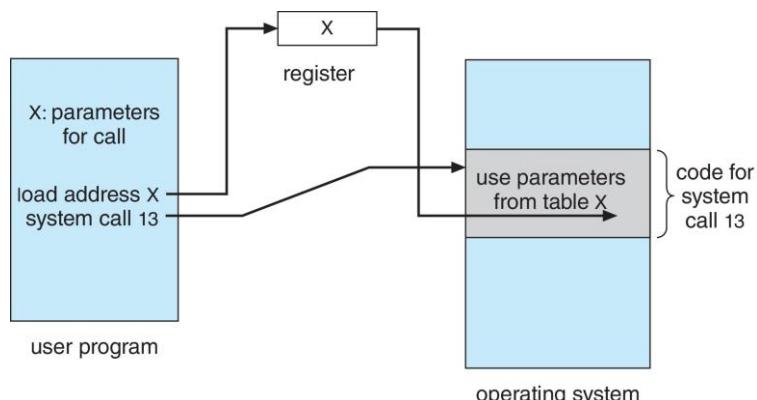


## System Call Parameter Passing

- Often, more information is required than simply identity of desired system call
  - Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
  - Pass the parameters in registers
    - In some cases, there may be more parameters than registers
  - Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register
    - This approach taken by Linux and Solaris
  - Parameters placed, or **pushed**, onto the **stack** by the program and **popped** off the stack by the operating system
  - Block and stack methods do not limit the number or length of parameters being passed

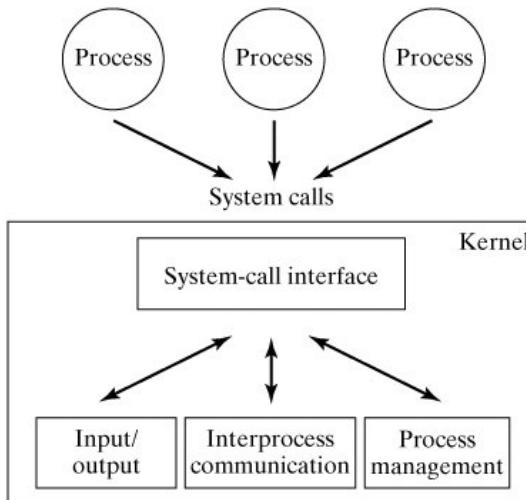


## Parameter Passing via Table





## Types of System Calls



Operating System Concepts – 10<sup>th</sup> Edition

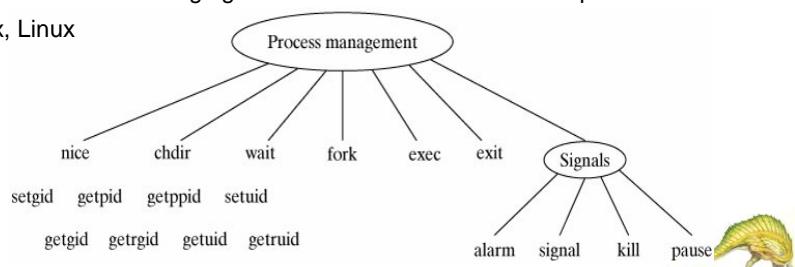
2a.23

Silberschatz, Galvin and Gagne ©2018



## Types of System Calls

- Process control
  - create process, terminate process
  - end, abort, load, execute
  - get process attributes, set process attributes
  - wait for time, wait event, signal event
  - allocate and free memory, dump memory if error
  - **Debugger** for determining **bugs**, **single step** execution
  - **Locks** for managing access to shared data between processes
- Ex, Linux



Operating System Concepts – 10<sup>th</sup> Edition

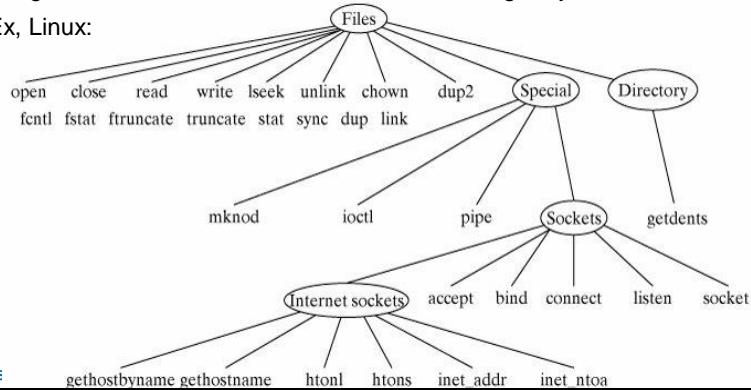
2a.24

Silberschatz, Galvin and Gagne ©2018



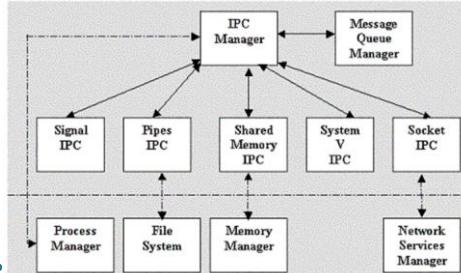
## Types of System Calls (Cont.)

- File management
  - create file, delete file, open, close file, read, write, reposition
  - get and set file attributes
- Device management
  - request device, release device, read, write, reposition
  - get device attributes, set device attributes, logically attach or detach devices
- Ex, Linux:



## Types of System Calls (Cont.)

- Information maintenance
  - get time or date, set time or date, get system data, set system data
  - get and set process, file, or device attributes
- Communications
  - create, delete communication connection, send, receive messages if **message passing model** to **host name** or **process name**
    - ▶ From **client** to **server**
  - **shared-memory model** create and gain access to memory regions
  - transfer status information, attach and detach remote devices
- Ex, Linux





## Types of System Calls (Cont.)

- Protection

- control access to resources
- get and set permissions
- allow and deny user access



## Examples of Windows and Unix System Calls

### EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

The following illustrates various equivalent system calls for Windows and UNIX operating systems.

	Windows	Unix
Process control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File management	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device management	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communications	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shm_open() mmap()
Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()



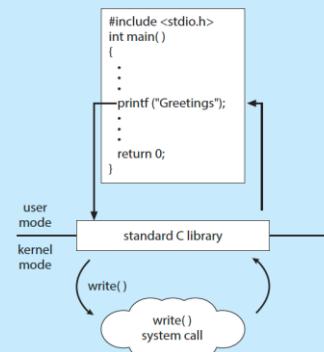


## Standard C Library Example

- C program invoking printf() library call, which calls write() system call

### THE STANDARD C LIBRARY

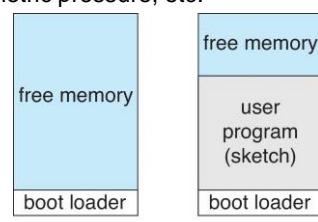
The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux. As an example, let's assume a C program invokes the `printf()` statement. The C library intercepts this call and invokes the necessary system call (or calls) in the operating system—in this instance, the `write()` system call. The C library takes the value returned by `write()` and passes it back to the user program:



## Example: Arduino

The Arduino is a simple hardware platform consisting of a microcontroller along with input sensors that respond to a variety of events, such as changes to light, temperature, and barometric pressure, etc.

- Single-tasking
- No operating system
- Programs (sketch) loaded via USB into flash memory
- Single memory space
- Boot loader loads program
- Program exit -> shell reloaded



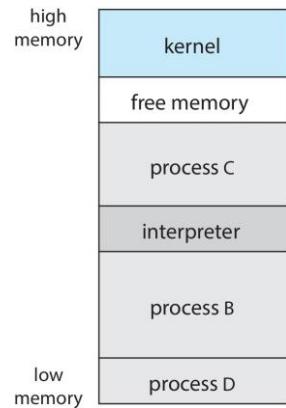
(a) At system startup      (b) running a program





## Example: FreeBSD

- Unix variant
- Multitasking
- User login -> invoke user's choice of shell
- Shell executes fork() system call to create process
  - Executes exec() to load program into process
  - Shell waits for process to terminate or continues with user commands
- Process exits with:
  - code = 0 – no error
  - code > 0 – error code



## System Services

- System programs provide a convenient environment for program development and execution. They can be divided into:
  - File manipulation (handling/use)
  - Status information sometimes stored in a file
  - Programming language support
  - Program loading and execution
  - Communications
  - Background services
  - Application programs
- Most users' view of the operation system is defined by system programs, not the actual system calls





## System Services (Cont.)

- Provide a convenient environment for program development and execution
  - Some of them are simply user interfaces to system calls; others are considerably more complex
- **File management** - Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories
- **Status information**
  - Some ask the system for info - date, time, amount of available memory, disk space, number of users
  - Others provide detailed performance, logging, and debugging information
  - Typically, these programs format and print the output to the terminal or other output devices
  - Some systems implement a **registry** - used to store and retrieve configuration information



## System Services (Cont.)

- **File modification**
  - Text editors to create and modify files
  - Special commands to search contents of files or perform transformations of the text
- **Programming-language support** - Compilers, assemblers, debuggers and interpreters sometimes provided
- **Program loading and execution** - Absolute loaders, relocatable loaders, linkage editors, and overlay-loaders, debugging systems for higher-level and machine language
- **Communications** - Provide the mechanism for creating virtual connections among processes, users, and computer systems
  - Allow users to send messages to one another's screens, browse web pages, send electronic-mail messages, log in remotely, transfer files from one machine to another





## System Services (Cont.)

- **Background Services**

- Launch at boot time
  - Some for system startup, then terminate
  - Some from system boot to shutdown
- Provide facilities like disk checking, process scheduling, error logging, printing
- Run in user context not kernel context
- Known as **services, subsystems, daemons**

- **Application programs**

- Don't pertain to system
- Run by users
- Not typically considered part of OS
- Launched by command line, mouse click, finger poke



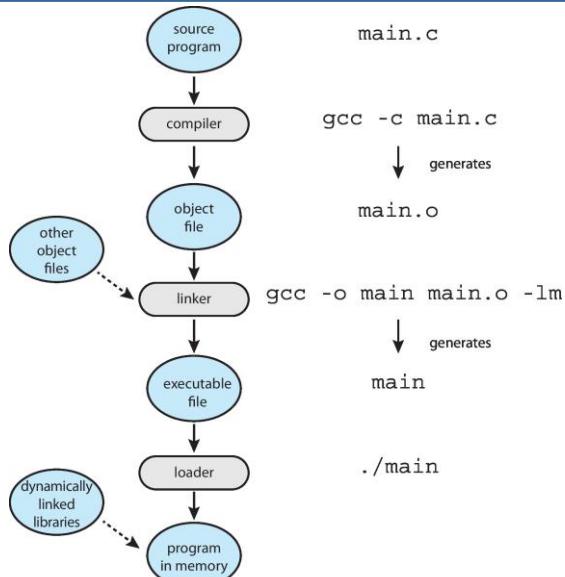
## Linkers and Loaders

- Source code compiled into object files designed to be loaded into any physical memory location – **relocatable object file**
- **Linker** combines these into single binary **executable** file
  - Also brings in libraries
- Program resides on secondary storage as binary executable
- Must be brought into memory by **loader** to be executed
  - **Relocation** assigns final addresses to program parts and adjusts code and data in program to match those addresses
- Modern general-purpose systems don't link libraries into executables
  - Rather, **dynamically linked libraries** (in Windows, **DLLs**) are loaded as needed, shared by all that use the same version of that same library (loaded once)
- Object, executable files have standard formats, so operating system knows how to load and start them





## The Role of the Linker and Loader



Operating System Concepts – 10<sup>th</sup> Edition

2a.37

Silberschatz, Galvin and Gagne ©2018



## Why Applications are Operating System Specific

- Apps compiled on one system usually not executable on other operating systems
- Each operating system provides its own unique system calls
  - Own file formats, etc.
- Apps can be multi-operating system
  - Written in interpreted language like Python, Ruby, and interpreter available on multiple operating systems
  - App written in language that includes a VM containing the running app (like Java)
  - Use standard language (like C), compile separately on each operating system to run on each
- **Application Binary Interface (ABI)** is architecture equivalent of API, defines how different components of binary code can interface for a given operating system on a given architecture, CPU, etc.

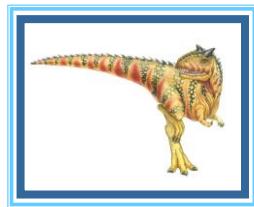


Operating System Concepts – 10<sup>th</sup> Edition

2a.38

Silberschatz, Galvin and Gagne ©2018

## End of Chapter 1.2



# Chapter I: Overview

## 1.3. Operating-System Design and Implementation



OS Concepts – 10<sup>th</sup> Edition

Silberschatz, Galvin and Gagne ©2018



## Outline

- Design and Implementation
- OS Structure
- Building and Booting an OS
- OS Debugging

OS Concepts – 10<sup>th</sup> Edition

2b.2

Silberschatz, Galvin and Gagne ©2018





## Objectives

- Compare and contrast monolithic, layered, microkernel, modular, and hybrid strategies for designing OSs
- Illustrate the process for booting an OS
- Apply tools for monitoring OS performance
- Design and implement kernel modules for interacting with a Linux kernel



## Design and Implementation

- Design and Implementation of OS is not “solvable”, but some approaches have proven successful
- Internal structure of different OSs can vary widely
- Start the design by defining goals and specifications
- Affected by choice of hardware, type of system
- **User** goals and **System** goals
  - User goals – OS should be convenient to use, easy to learn, reliable, safe, and fast
  - System goals – OS should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient
- Specifying and designing an OS is highly creative task of **software engineering**





## Policy and Mechanism

- **Policy:** What needs to be done?
  - Example: Interrupt after every 100 seconds
- **Mechanism:** How to do something?
  - Example: timer
- Important principle: separate policy from mechanism
- The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later.
  - Example: change 100 to 200



## Implementation

- Much variation
  - Early OSes in assembly language
  - Then system programming languages like Algol, PL/I
  - Now C, C++
- Actually, usually a mix of languages
  - Lowest levels in assembly
  - Main body in C
  - Systems programs in C, C++, scripting languages like PERL, Python, shell scripts
- More high-level language easier to **port** to other hardware
  - But slower
- **Emulation** can allow an OS to run on non-native hardware





## OS Structure

- General-purpose OS is very large program
- Various ways to structure ones
  - Layered – an abstraction
  - Simple structure – MS-DOS
  - More complex – UNIX
  - Microkernel – Mach

OS Concepts – 10<sup>th</sup> Edition

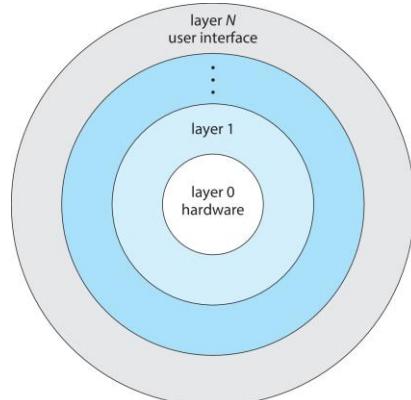
2b.7

Silberschatz, Galvin and Gagne ©2018



## Layered Approach

- The OS is divided into a number of layers (levels),
  - each built on top of lower layers.
  - The bottom layer (layer 0), is the hardware;
  - the highest (layer N) is the user interface.
- With modularity, layers are selected such that
  - each uses functions (operations) and
  - services of only lower-level layers



OS Concepts – 10<sup>th</sup> Edition

2b.8

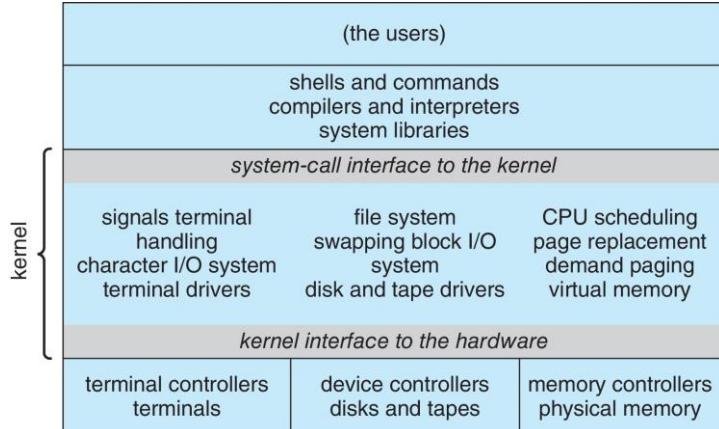
Silberschatz, Galvin and Gagne ©2018





## Traditional UNIX System Structure

Beyond simple but not fully layered



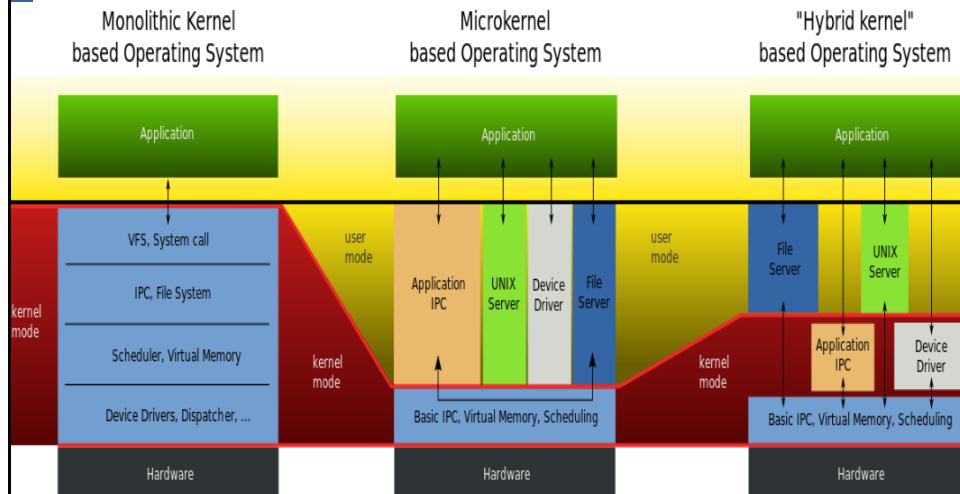
OS Concepts – 10<sup>th</sup> Edition

2b.9

Silberschatz, Galvin and Gagne ©2018



## Comparison kernel types



OS Concepts – 10<sup>th</sup> Edition

2b.10

Silberschatz, Galvin and Gagne ©2018





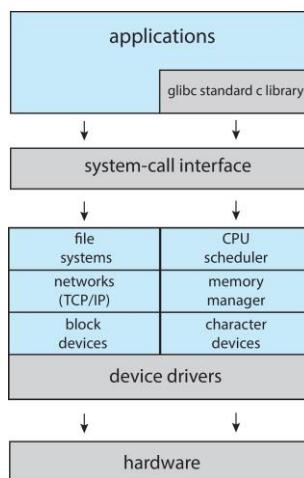
## Monolithic Structure – Original UNIX

- UNIX – limited by hardware functionality, the original UNIX OS had limited structuring.
- The UNIX OS consists of two separable parts
  - Systems programs
  - The kernel
    - Consists of everything below the system-call interface and above the physical hardware
    - Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level



## Linux System Structure

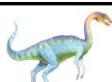
Monolithic plus modular design



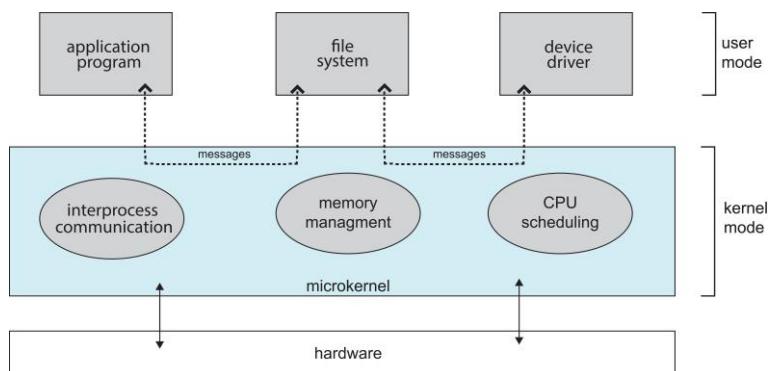


## Microkernels

- Moves as much from the kernel into user space
- **Mach** is an example of **microkernel**
  - Mac OS X kernel (**Darwin**) partly based on Mach
- Communication takes place between user modules using **message passing**
- Benefits:
  - Easier to extend a microkernel
  - Easier to port the OS to new architectures
  - More reliable (less code is running in kernel mode)
  - More secure
- Detriments:
  - Performance overhead of user space to kernel space communication



## Microkernel System Structure





## Modules

- Many modern OSs implement **loadable kernel modules (LKMs)**
  - Uses object-oriented approach
  - Each core component is separate
  - Each talks to the others over known interfaces
  - Each is loadable as needed within the kernel
- Overall, similar to layers but with more flexible
  - Linux, Solaris, etc.



## Hybrid Systems

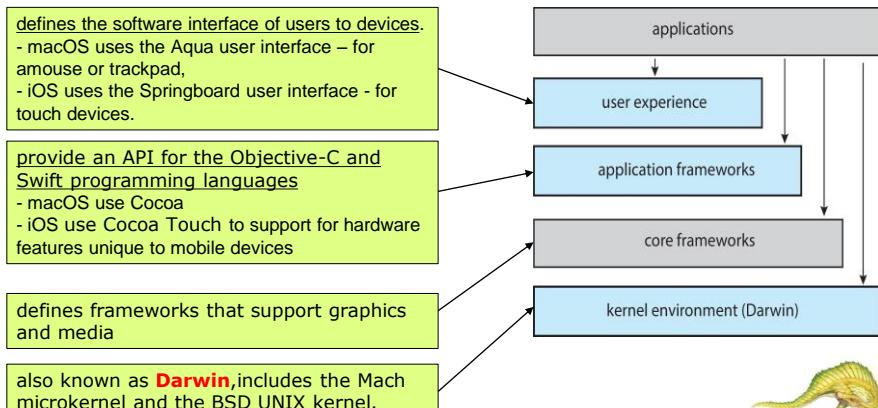
- Most modern OSs are not one pure model
  - Hybrid combines multiple approaches to address performance, security, usability needs
  - Linux and Solaris kernels in kernel address space, so monolithic, plus modular for dynamic loading of functionality
  - Windows mostly monolithic, plus microkernel for different subsystem **personalities**
- Apple Mac OS X hybrid, layered, **Aqua** UI plus **Cocoa** programming environment
  - Below is kernel consisting of Mach microkernel and BSD Unix parts, plus I/O kit and dynamically loadable modules (called **kernel extensions**)





## macOS and iOS Structure

- Apple's macOS: is designed to run primarily on desktop and laptop
- iOS is a mobile operating system designed for the iPhone smartphone and iPad tablet computer.
- Architecturally, macOS and iOS have much in common



OS Concepts – 10<sup>th</sup> Edition

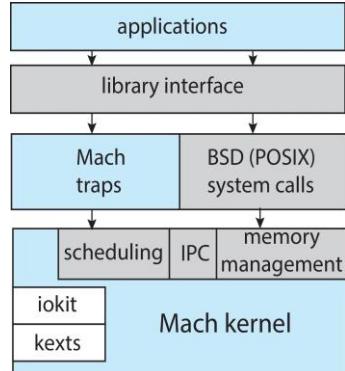
2b.17

Silberschatz, Galvin and Gagne ©2018



## Darwin - a hybrid structure

- Darwin is a layered system that consists
  - the Mach microkernel
  - the BSD UNIX kernel.
- Darwin provides two system-call interfaces:
  - Mach system calls (known as traps):
    - ▶ provides fundamental OS
  - BSD system calls
    - ▶ provide POSIX functionality
  - I/O kit:
    - ▶ for development of device drivers and dynamically loadable modules



OS Concepts – 10<sup>th</sup> Edition

2b.18

Silberschatz, Galvin and Gagne ©2018





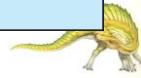
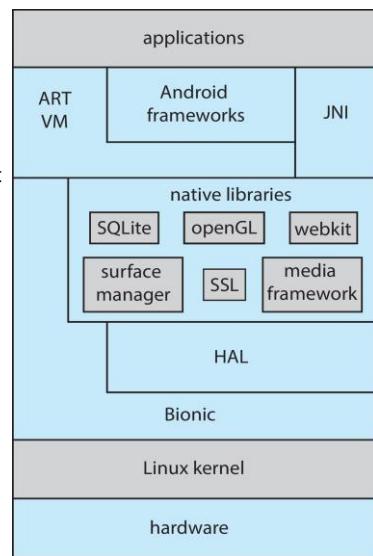
## Android

- Developed by Open Handset Alliance (mostly Google)
  - Open Source
- Similar to IOS
- Based on Linux kernel but modified
  - Provides process, memory, device-driver management
  - Adds power management
- Runtime environment includes core set of libraries and Dalvik virtual machine
  - Apps developed in Java plus Android API
    - Java class files compiled to Java bytecode then translated to executable than runs in Dalvik VM
- Libraries include frameworks for web browser (webkit), database (SQLite), multimedia, smaller libc



## Android Architecture

- Google has chosen to abstract the physical hardware through the hard-ware abstraction layer, or HAL.
  - By abstracting all hardware, such as the camera, GPS chip, and other sensors, the HAL provides applications with a consistent view independent of specific hardware.
  - => allows developers to write programs that are portable across different hardware platforms.





## Building and Booting an OS

- OSs generally designed to run on a class of systems with variety of peripherals
- Commonly, OS already installed on purchased computer
  - But can build and install some other OSs
  - If generating an OS from scratch
    - Write the OS source code
    - Configure the OS for the system on which it will run
    - Compile the OS
    - Install the OS
    - Boot the computer and its new OS



## Building and Booting Linux

- Download Linux source code (<http://www.kernel.org>)
- Configure kernel via “make menuconfig”
- Compile the kernel using “make”
  - Produces `vmlinuz`, the kernel image
  - Compile kernel modules via “make modules”
  - Install kernel modules into `vmlinuz` via “make modules\_install”
  - Install new kernel on the system via “make install”

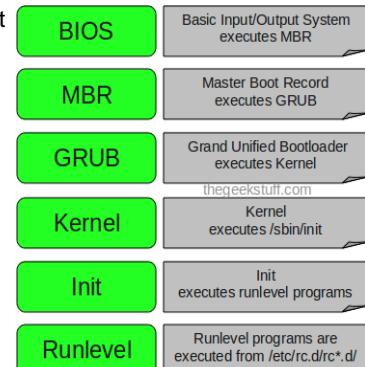




## System Boot

- Process
  - Small piece of code – **bootstrap loader**, **BIOS**, stored in **ROM** or **EEPROM** locates the kernel, loads it into memory, and starts it
  - Sometimes two-step process where **boot block** at fixed location loaded by ROM code, which loads bootstrap loader from disk
  - Modern systems replace BIOS with **Unified Extensible Firmware Interface (UEFI)**
- Common bootstrap loader, **GRUB**, allows selection of kernel from multiple disks, versions, kernel options
- Kernel loads and system is then **running**
- Boot loaders frequently allow various boot states, such as single user mode

Example: Linux boot



## Operating-System Debugging

- **Debugging** is finding and fixing errors, or **bugs**
- OS generate **log files** containing error information
- Failure of an application can generate **core dump** file capturing memory of the process
- OS failure can generate **crash dump** file containing kernel memory

Kernighan's Law: "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."



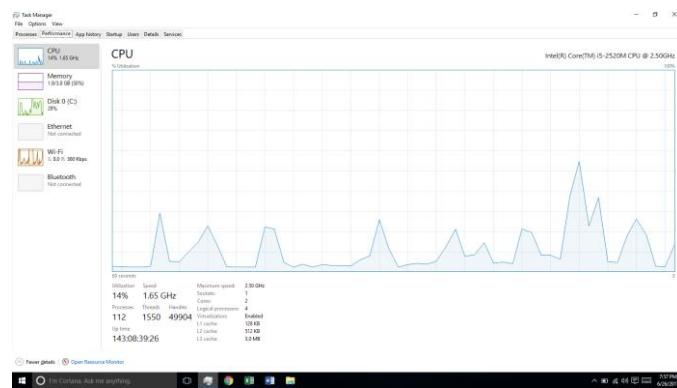


## Performance Tuning

- Improve performance by removing bottlenecks
- OS must provide means of computing and displaying measures of system behavior
- Sometimes using **trace listings** of activities, recorded for analysis
- **Profiling** is periodic sampling of instruction pointer to look for statistical trends
- Per-Process Tool
  - ps—reports information for a single process or selection of processes
  - top—reports real-time statistics for current processes
- System-Wide Tool
  - vmstat—reports memory-usage statistics
  - netstat—reports statistics for network interfaces
  - iostat—reports I/O usage for disks



## Performance Tuning





## Tracing

- Collects data for a specific event, such as steps involved in a system call invocation
- Tools include
  - strace – trace system calls invoked by a process
  - gdb – source-level debugger
  - perf – collection of Linux performance tools
  - tcpdump – collects network packets



## BCC

- Debugging interactions between user-level and kernel code nearly impossible without toolset that understands both and can instrument their actions
- BCC (BPF Compiler Collection) is a rich toolkit providing tracing features for Linux
  - See also the original DTrace
- For example, disksnoop.py traces disk I/O activity

TIME(s)	T	BYTES	LAT(ms)
1946.29186700	R	8	0.27
1946.33965000	R	8	0.26
1948.34585000	W	8192	0.96
1950.43251000	R	4096	0.56
1951.74121000	R	4096	0.35

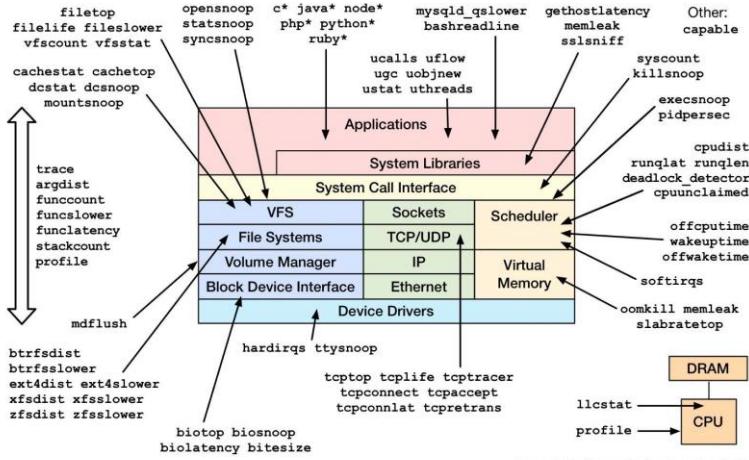
- Many other tools (next slide)





## Linux bcc/BPF Tracing Tools

Linux bcc/BPF Tracing Tools



<https://github.com/iovisor/bcc#tools> 2017

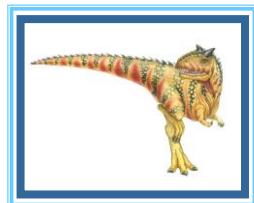


OS Concepts – 10<sup>th</sup> Edition

2b.30

Silberschatz, Galvin and Gagne ©2018

## End of Chapter 1.3



OS Concepts – 10<sup>th</sup> Edition

Silberschatz, Galvin and Gagne ©2018

## Chapter 2: Process Management

### 2.1 Processes



GV: Nguyễn Thị Thanh Vân

Operating System Concepts – 10<sup>th</sup> Edition

Silberschatz, Galvin and Gagne ©2018



### Objectives

- Identify the separate components of a process and illustrate how they are represented and scheduled in an operating system.
- Describe how processes are created and terminated in an operating system, including developing programs using the appropriate system calls that perform these operations.
- Describe and contrast interprocess communication using shared memory and message passing.
- Design programs that use pipes and POSIX shared memory to perform interprocess communication.
- Describe client-server communication using sockets and remote procedure calls.
- Design kernel modules that interact with the Linux operating system.



Operating System Concepts – 10<sup>th</sup> Edition

3.2

Silberschatz, Galvin and Gagne ©2018



## Outlines

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- IPC in Shared-Memory Systems
- IPC in Message-Passing Systems
- Examples of IPC Systems
- Communication in Client-Server Systems



## Process Concept

- An OS executes a variety of programs that run as a process.
- **Process**
  - a program in execution - must progress in sequential fashion.
  - No parallel execution of instructions of a single process
- One program can be several processes
  - Consider multiple users executing the same program
    - 4 Compiler, Text editor
- The memory layout of a process is typically divided into multiple parts
  - The program code, also called **text section**
  - Current activity including **program counter**, processor registers
  - **Stack** containing temporary data
    - 4 Function parameters,
    - 4 return addresses,
    - 4 local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time





# Process in Memory

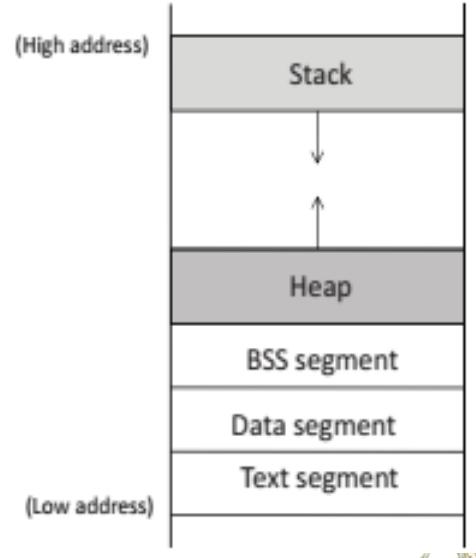
**Stack:** store local variables in func,  
store data related to function calls:  
return address, arguments, (LIFO)

**Heap:** provide space for dynamic memory allocation. This area is managed by malloc, calloc ...

**BSS segment:** stores uninitialized static/global variables (zero)

**Data segment** : stores static/global variables that are initialized by the programmer

**Text:** stores the executable code of the program (read-only)



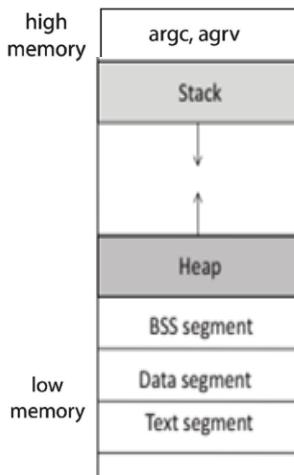
Operating System Concepts – 10<sup>th</sup> Edition

3.5

Silberschatz, Galvin and Gagne ©2018



# Memory Layout of a C Program



```
#include <stdio.h>
#include <stdlib.h>

int x;
int y = 15;

int main(int argc, char *argv[])
{
    int *values;
    int i;

    values = (int *)malloc(sizeof(int)*5);

    for(i = 0; i < 5; i++)
        values[i] = i;

    return 0;
}
```



Operating System Concepts – 10<sup>th</sup> Edition

37

Silberschatz, Galvin and Gagne ©2018



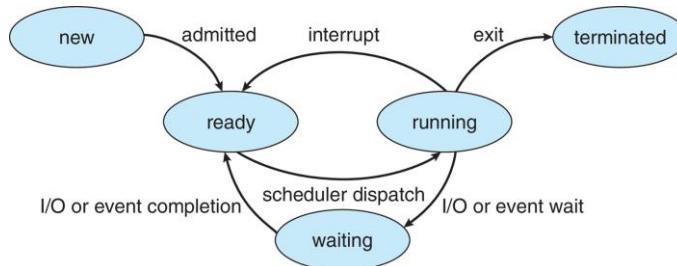
## Process

- A process includes:
  - Text
  - Data
  - Heap
  - Stack
  - PC – Program counter: a register that manages the memory address of the instruction to be executed next
  - PSW – Program status word: a register that performs the function of a status register and program counter
  - SP – Stack pointer
  - Registers
  
- Four principal events cause processes to be created:
  - System initialization.
  - Execution of a process-creation system call by a running process.
  - A user request to create a new process.
  - Initiation of a batch job.



## 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

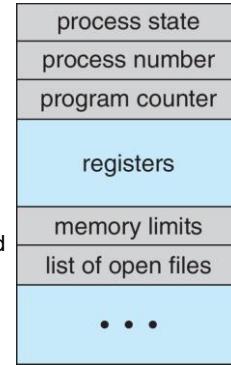




## Process Control Block (PCB)

- ❖ PCB: a data structure used by computer OS to store all the information about a process.
- ❖ Each process is represented in OS by PCB, also called **task control block**
- ❖ It contains many pieces of information associated with a specific process:

- 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



## Threads

- So far, process has a single thread of execution
- Most modern OSs have extended the process concept to allow a process to have multiple threads of execution
  - thus to perform more than one task at a time
  - Multiple threads can run in parallel
- The PCB is expanded to include information for each thread.
  - Must then have storage for thread details,
- Explore in detail later – next chapter

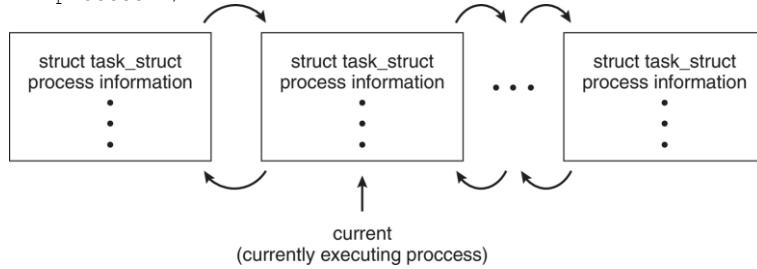




## PCB in Linux OS

PCB is represented by the C structure `task_struct`, which is found in `<include/linux/sched.h>`

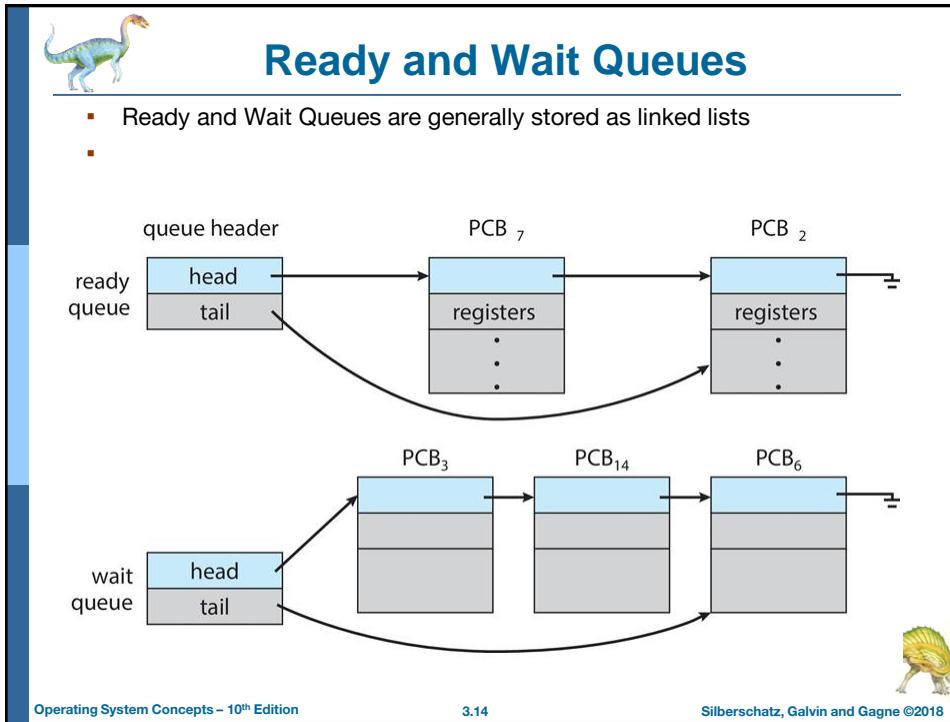
```
pid t_pid;           /* process identifier */
long state;          /* state of the process */
unsigned int time_slice /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm;      /* address space of this
process */
```



## Process Scheduling

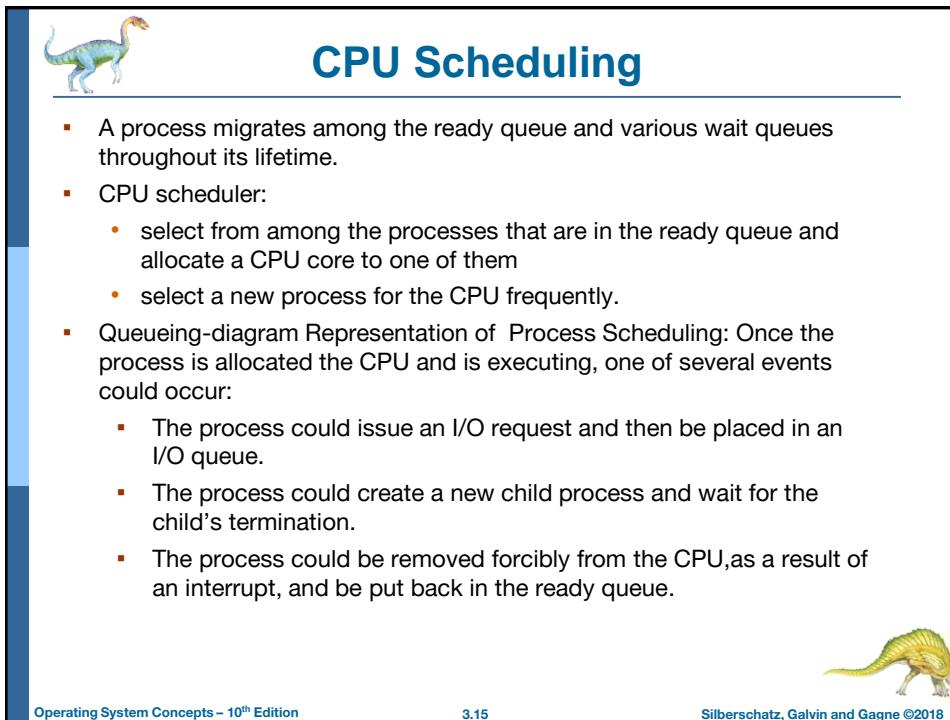
- Goals of:
  - Multiprogramming: some process running at all times so as to maximize CPU utilization
  - Time sharing: switch a CPU core among processes so frequently that users can interact with each program while it is running
- => **Process scheduler** selects among available processes for next execution on CPU core
- The number of processes currently in memory is known as the **degree of multiprogramming**
- Maintains **scheduling queues** of processes
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Wait queues** – set of processes waiting for an event (i.e., I/O)
  - Processes migrate among the various queues



Operating System Concepts – 10<sup>th</sup> Edition

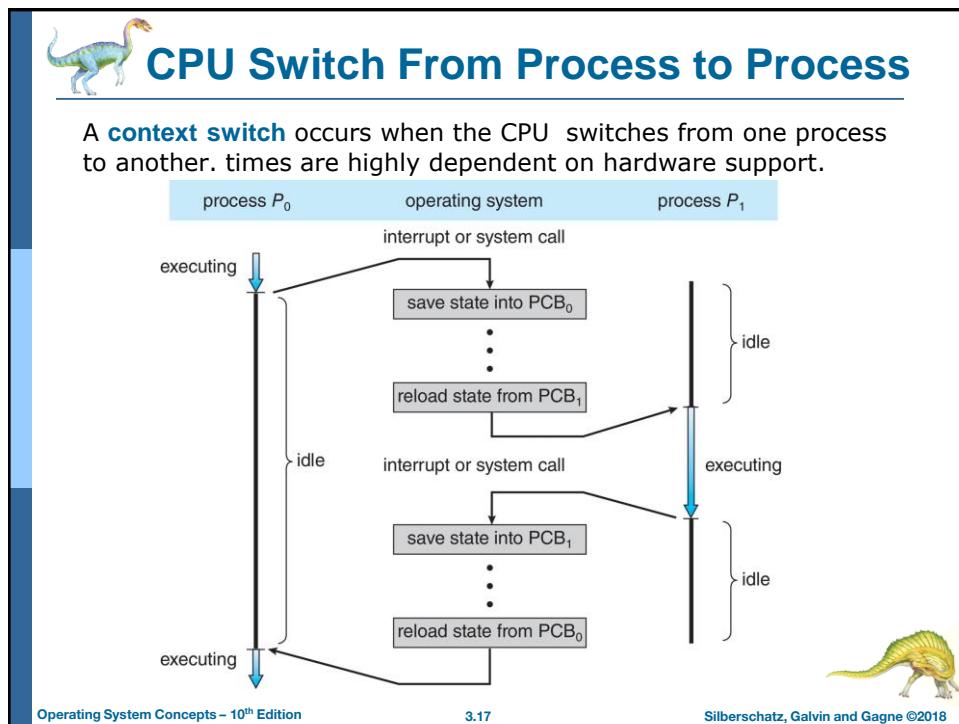
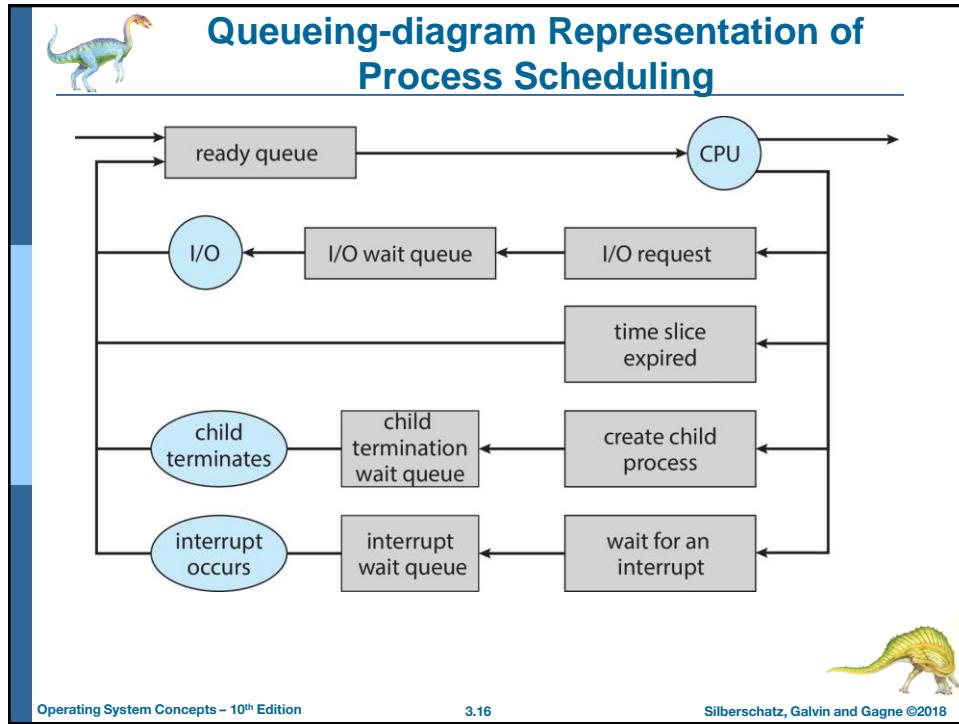
3.14

Silberschatz, Galvin and Gagne ©2018

Operating System Concepts – 10<sup>th</sup> Edition

3.15

Silberschatz, Galvin and Gagne ©2018





## Context Switching

- 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 pure 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



## Multitasking in Mobile Systems

- Some mobile systems (e.g., early version of iOS) allow only one process to run, others suspended
- Due to screen real estate, user interface limits iOS provides for a
  - Single **foreground** process- controlled via user interface
  - Multiple **background** processes- in memory, running, but not on the display, and with limits
  - Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback
- Android runs foreground and background, with fewer limits
  - Background process uses a **service** to perform tasks
  - Service can keep running even if background process is suspended
  - Service has no user interface, small memory use





## Operations on Processes

- System must provide mechanisms for:
  - Process creation
  - Process termination



## Process Creation

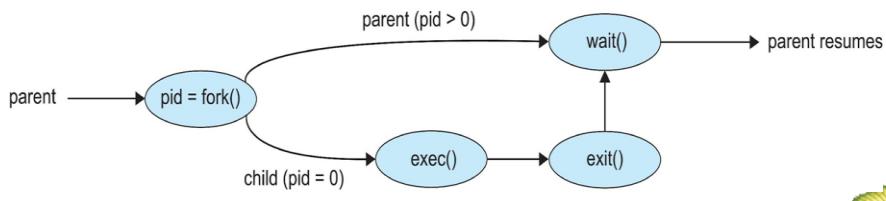
- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier (pid)**
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of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





## Process Creation (Cont.)

- 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
  - Parent process calls **wait()** waiting for the child to terminate

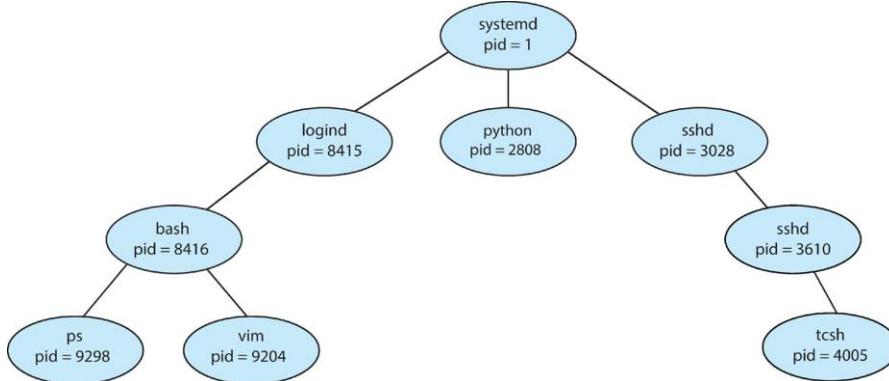
Operating System Concepts – 10<sup>th</sup> Edition

3.22

Silberschatz, Galvin and Gagne ©2018



## A Tree of Processes in Linux

Operating System Concepts – 10<sup>th</sup> Edition

3.23

Silberschatz, Galvin and Gagne ©2018





## 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 */
        execvp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
    }

    return 0;
}
```

Operating System Concepts - 10<sup>th</sup> Edition

3.24

Silberschatz, Galvin and Gagne ©2018



## Creating a Separate Process via Windows API

```
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    /* allocate memory */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
                      "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
                      NULL, /* don't inherit process handle */
                      NULL, /* don't inherit thread handle */
                      FALSE, /* disable handle inheritance */
                      0, /* no creation flags */
                      NULL, /* use parent's environment block */
                      NULL, /* use parent's existing directory */
                      &si,
                      &pi))
    {
        fprintf(stderr, "Create Process Failed");
        return -1;
    }
    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

Operating System Concepts - 10<sup>th</sup> Edition

3.25

Silberschatz, Galvin and Gagne ©2018



## 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



## Process Termination

- Some operating systems do not allow child to exists 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**





## Ex

- Detail in Process syscal expl file



## Android Process Importance Hierarchy

- Mobile operating systems often have to terminate processes to reclaim system resources such as memory. From **most** to **least** important:
  - Foreground process
  - Visible process
  - Service process
  - Background process
  - Empty process
- Android will begin terminating processes that are least important.





## Multiprocess Architecture – Chrome Browser

- Many web browsers ran as single process (some still do)
  - If one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser is multiprocess with 3 different types of processes:
  - **Browser** process manages user interface, disk and network I/O
  - **Renderer** process renders web pages, deals with HTML, Javascript. A new renderer created for each website opened
    - 4 Runs in **sandbox** restricting disk and network I/O, minimizing effect of security exploits
  - **Plug-in** process for each type of plug-in



Each tab represents a separate process.



## Interprocess Communication

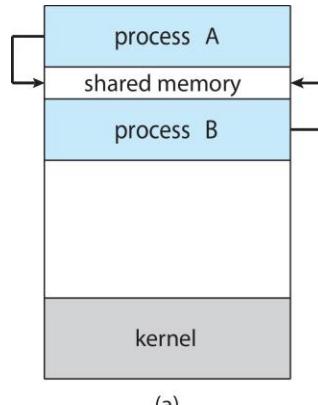
- Processes within a system may be **independent** or **cooperating**
- **Cooperating process** can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication (IPC)**
- Two models of IPC
  - **Shared memory** (under the control of users)
  - **Message passing** (under the control of OS)



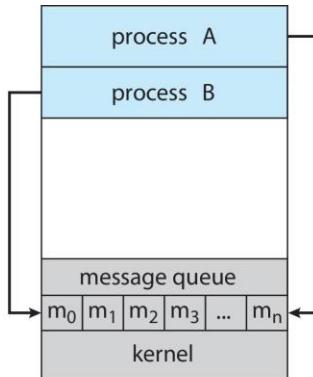


## Communications Models

(a) Shared memory.

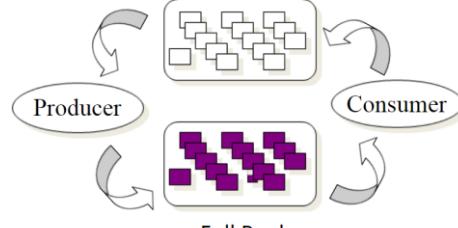


(b) Message passing.



## Producer-Consumer Problem

- Paradigm for cooperating processes:
  - **producer** process produces information that is consumed by a **consumer** process



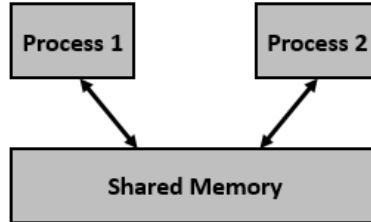
- Two variations:
  - **unbounded-buffer** places no practical limit on the size of the buffer:
    - 4 Producer never waits
    - 4 Consumer waits if there is no buffer to consume
  - **bounded-buffer** assumes that there is a fixed buffer size
    - 4 Producer must wait if all buffers are full
    - 4 Consumer waits if there is no buffer to consume





## Shared Memory Solution

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
- Synchronization is discussed in great details in Chapters 6 & 7.



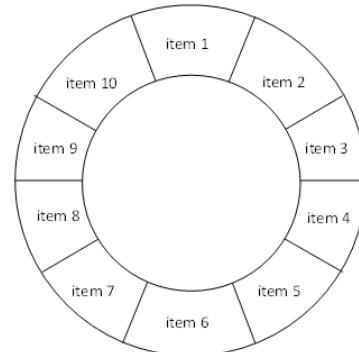
## Bounded-Buffer – Shared-Memory Solution

- Shared data

```

#define BUFFER_SIZE 10
typedef struct {
    . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
  
```



- Solution presented in next slides is correct, but can only use **BUFFER\_SIZE-1** items; that is: 9 items





## Producer/ Consumer Process – Shared Memory

**Producer**

```

item next_produced;
while (true) {
    /* produce an item in next produced */
    while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}

```

**Customer**

```

item next_consumed;
while (true) {
    while (in == out)
        ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    /* consume the item in next consumed */
}

```



Operating System Concepts – 10<sup>th</sup> Edition      3.36      Silberschatz, Galvin and Gagne ©2018



## What about Filling all the Buffers?

- Suppose that we wanted to provide a solution to the consumer-producer problem that fills **all** the buffers.
- We can do so by having an integer **counter** that keeps track of the number of full buffers.
- Initially, **counter** is set to 0.
- The integer **counter** is incremented by the producer after it produces a new buffer.
- The integer **counter** is and is decremented by the consumer after it consumes a buffer.



Operating System Concepts – 10<sup>th</sup> Edition      3.37      Silberschatz, Galvin and Gagne ©2018



## Producer/ Consumer Process

---

**Producer**

```

while (true) {
    /* produce an item in next produced
    while (counter == BUFFER_SIZE)
        ; /* do nothing */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}

```

**Customer**

```

while (true) {
    while (counter == 0)
        ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
    /* consume the item in next consumed */
}

```

Operating System Concepts - 10<sup>th</sup> Edition
3.30
Silberschatz, Galvin and Gagne ©2018




## Race Condition

---

- **counter++** could be implemented as
 

```

register1 = counter
register1 = register1 + 1
counter = register1
      
```
- **counter--** could be implemented as
 

```

register2 = counter
register2 = register2 - 1
counter = register2
      
```
- Consider this execution interleaving with “count = 5” initially:

S0: producer execute <b>register1 = counter</b>	{register1 = 5}
S1: producer execute <b>register1 = register1 + 1</b>	{register1 = 6}
S2: consumer execute <b>register2 = counter</b>	{register2 = 5}
S3: consumer execute <b>register2 = register2 - 1</b>	{register2 = 4}
S4: producer execute <b>counter = register1</b>	{counter = 6 }
S5: consumer execute <b>counter = register2</b>	{counter = 4}

- **Question** – why was there no race condition in the first solution (where at most N - 1) buffers can be filled?

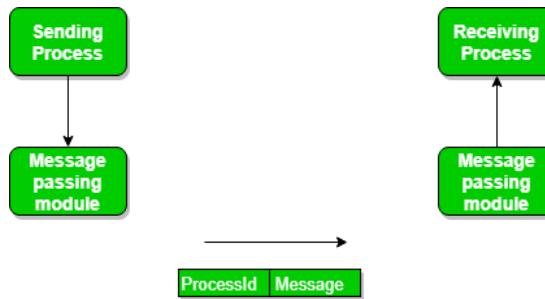
Operating System Concepts - 10<sup>th</sup> Edition
3.39
Silberschatz, Galvin and Gagne ©2018





## IPC – Message Passing

- Processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - `send(message)`
  - `receive(message)`
- The *message size* is either fixed or variable

Operating System Concepts – 10<sup>th</sup> Edition

3.40

Silberschatz, Galvin and Gagne ©2018



## Message Passing (Cont.)

- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a **communication link** between them
  - Exchange messages via send/receive
- Implementation issues:
  - How are **links** established?
  - Can a link be associated with more than two processes?
  - How many links can there be between every pair of communicating processes?
  - What is the capacity of a link?
  - Is the size of a message that the link can accommodate fixed or variable?
  - Is a link unidirectional or bi-directional?

Operating System Concepts – 10<sup>th</sup> Edition

3.41

Silberschatz, Galvin and Gagne ©2018



## Implementation of Communication Link

- Physical:
  - Shared memory
  - Hardware bus
  - Network
- Logical:
  - Direct or indirect
  - Synchronous or asynchronous
  - Automatic or explicit buffering



## Direct Communication

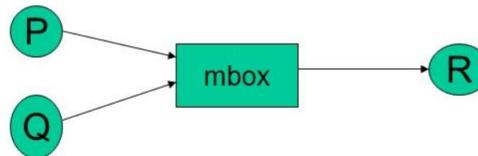
- Processes must name each other explicitly:
  - **send**(*P, message*) – send a message to process P
  - **receive**(*Q, message*) – receive a message from process Q
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional





## Indirect Communication

- Messages are directed and received from mailboxes (also is ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional

Operating System Concepts – 10<sup>th</sup> Edition

3.44

Silberschatz, Galvin and Gagne ©2018



## Indirect Communication

- Operations
  - Create a new mailbox (port)
  - Send and receive messages through mailbox
  - Delete a mailbox
- Primitives are defined as:
  - **Send(A, message)** – send a message to mailbox A
  - **receive(A, message)** – receive a message from mailbox A

Operating System Concepts – 10<sup>th</sup> Edition

3.45

Silberschatz, Galvin and Gagne ©2018

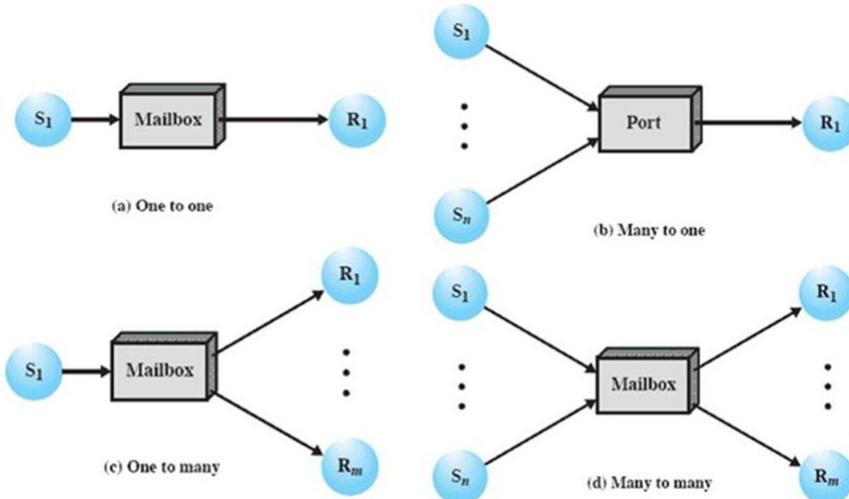


## Indirect Communication (Cont.)

- Mailbox sharing
  - $P_1$ ,  $P_2$ , and  $P_3$  share mailbox A
  - $P_1$ , sends;  $P_2$  and  $P_3$  receive
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.



## Indirect Communication (Cont.)





## Synchronization

Message passing may be either blocking or non-blocking

- **Blocking** is considered **synchronous**
  - **Blocking send** -- the sender is blocked until the message is received
  - **Blocking receive** -- the receiver is blocked until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** -- the sender sends the message and continue
  - **Non-blocking receive** -- the receiver receives:
    - 4 A valid message, or
    - 4 Null message
- Different combinations possible
  - If both send and receive are blocking, we have a **rendezvous**



## Producer-Consumer: Message Passing

- Producer
 

```
message next_produced;
while (true) {
    /* produce an item in next_produced */

    send(next_produced);
}
```
- Consumer
 

```
message next_consumed;
while (true) {
    receive(next_consumed)

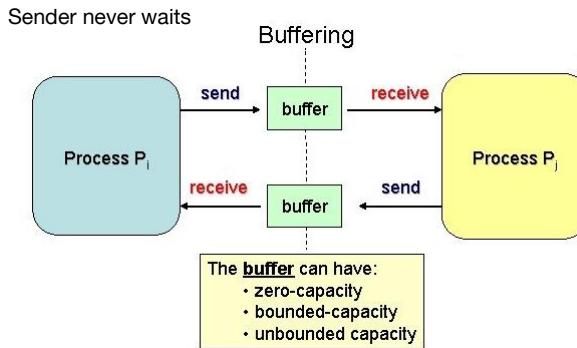
    /* consume the item in next_consumed */
}
```





## Buffering

- Queue of messages attached to the link.
- Implemented in one of three ways
  1. Zero capacity – no messages are queued on a link.  
Sender must wait for receiver (rendezvous)
  2. Bounded capacity – finite length of  $n$  messages  
Sender must wait if link full
  3. Unbounded capacity – infinite length

Operating System Concepts – 10<sup>th</sup> Edition

3.50

Silberschatz, Galvin and Gagne ©2018



## Examples of IPC Systems - POSIX

- POSIX Shared Memory
  - Process first creates shared memory segment
 

```
shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
```
  - Also used to open an existing segment
  - Set the size of the object
 

```
ftruncate(shm_fd, 4096);
```
  - Use `mmap()` to memory-map a file pointer to the shared memory object
  - Reading and writing to shared memory is done by using the pointer returned by `mmap()`.

Operating System Concepts – 10<sup>th</sup> Edition

3.51

Silberschatz, Galvin and Gagne ©2018





## IPC POSIX Producer

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
    /* the size (in bytes) of shared memory object */
    const int SIZE = 4096;
    /* name of the shared memory object */
    const char *name = "OS";
    /* strings written to shared memory */
    const char *message_0 = "Hello";
    const char *message_1 = "World!";

    /* shared memory file descriptor */
    int shm_fd;
    /* pointer to shared memory object */
    void *ptr;

    /* create the shared memory object */
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

    /* configure the size of the shared memory object */
    ftruncate(shm_fd, SIZE);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

    /* write to the shared memory object */
    sprintf(ptr,"%s",message_0);
    ptr += strlen(message_0);
    sprintf(ptr,"%s",message_1);
    ptr += strlen(message_1);

    return 0;
}
```

Operating System Concepts – 10<sup>th</sup> Edition

3.52

Silberschatz, Galvin and Gagne ©2018



## IPC POSIX Consumer

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
    /* the size (in bytes) of shared memory object */
    const int SIZE = 4096;
    /* name of the shared memory object */
    const char *name = "OS";
    /* shared memory file descriptor */
    int shm_fd;
    /* pointer to shared memory object */
    void *ptr;

    /* open the shared memory object */
    shm_fd = shm_open(name, O_RDONLY, 0666);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

    /* read from the shared memory object */
    printf("%s",(char *)ptr);

    /* remove the shared memory object */
    shm_unlink(name);

    return 0;
}
```

Operating System Concepts – 10<sup>th</sup> Edition

3.53

Silberschatz, Galvin and Gagne ©2018





## Examples of IPC Systems - Mach

- Mach communication is message based
  - Even system calls are messages
  - Each task gets two ports at creation- Kernel and Notify
  - Messages are sent and received using the `mach_msg()` function
  - Ports needed for communication, created via `mach_port_allocate()`
  - Send and receive are flexible, for example four options if mailbox full:
    - 4 Wait indefinitely
    - 4 Wait at most n milliseconds
    - 4 Return immediately
    - 4 Temporarily cache a message



## Mach Messages

```
#include<mach/mach.h>

struct message {
    mach_msg_header_t header;
    int data;
};

mach_port_t client;
mach_port_t server;
```





## Mach Message Passing – Client/Server

```

/* Client Code */

struct message message;

// construct the header
message.header.msgh_size = sizeof(message);
message.header.msgh_remote_port = server;
message.header.msgh_local_port = client;

// send the message
mach_msg(&message.header, // message header
         MACH_SEND_MSG, // sending a message
         sizeof(message), // size of message sent
         0, // maximum size of received message - unnecessary
         MACH_PORT_NULL, // name of receive port - unnecessary
         MACH_MSG_TIMEOUT_NONE, // no time outs
         MACH_PORT_NULL // no notify port
);

/* Server Code */

struct message message;

// receive the message
mach_msg(&message.header, // message header
         MACH_RCV_MSG, // sending a message
         0, // size of message sent
         sizeof(message), // maximum size of received message
         server, // name of receive port
         MACH_MSG_TIMEOUT_NONE, // no time outs
         MACH_PORT_NULL // no notify port
);

```

Operating S

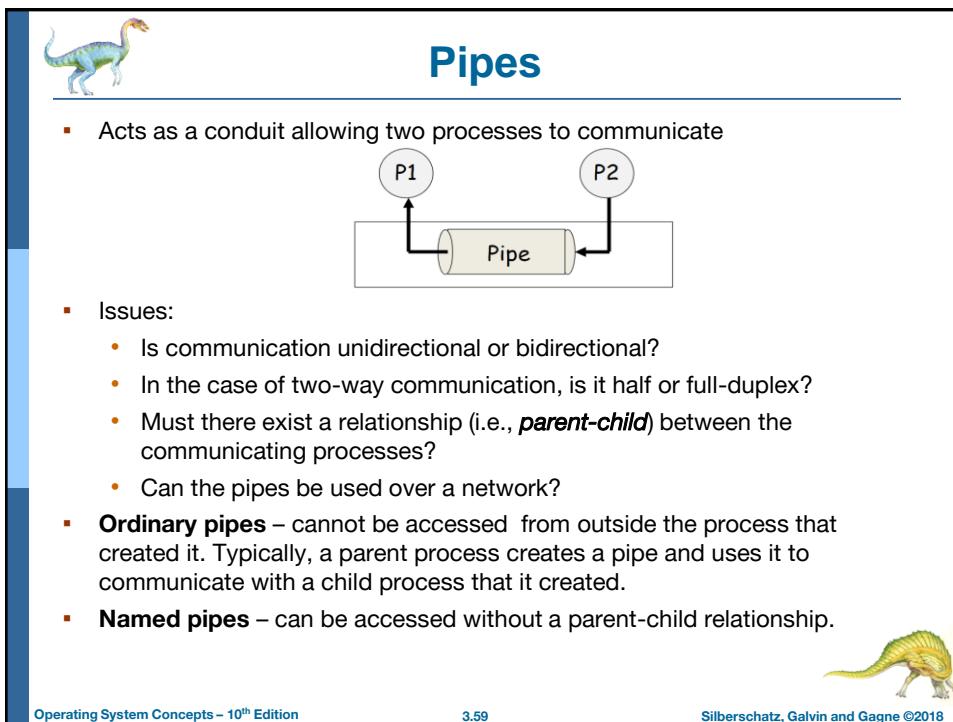
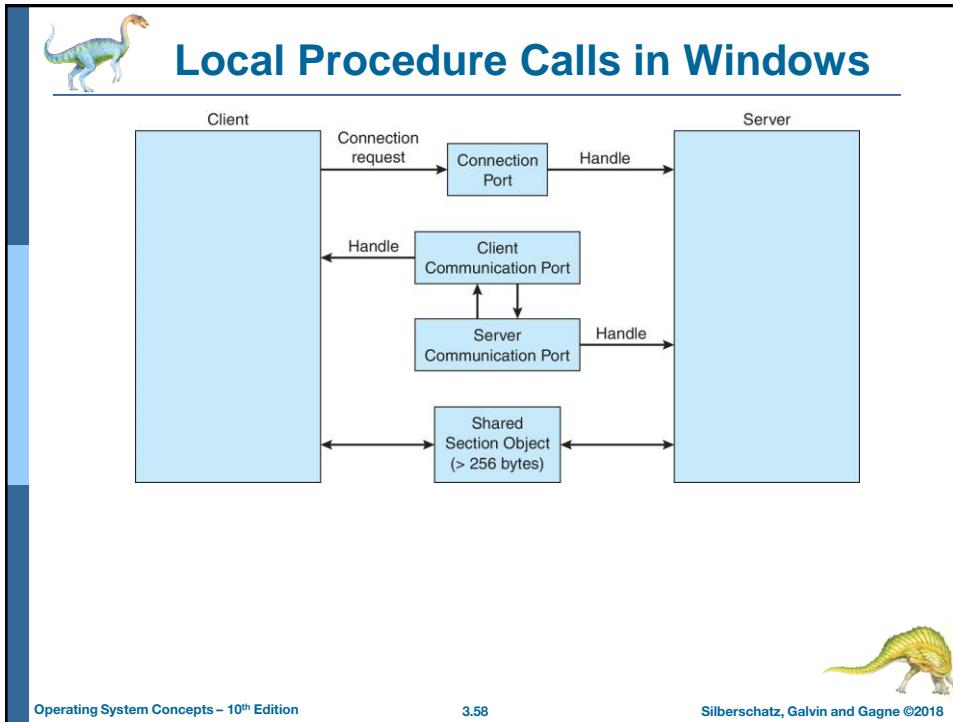
Galvin and Gagne ©2018



## Examples of IPC Systems – Windows

- Message-passing centric via **advanced local procedure call (LPC)** facility
  - Only works between processes on the same system
  - Uses ports (like mailboxes) to establish and maintain communication channels
  - Communication works as follows:
    - 4 The client opens a handle to the subsystem's **connection port** object.
    - 4 The client sends a connection request.
    - 4 The server creates two private **communication ports** and returns the handle to one of them to the client.
    - 4 The client and server use the corresponding port handle to send messages or callbacks and to listen for replies.

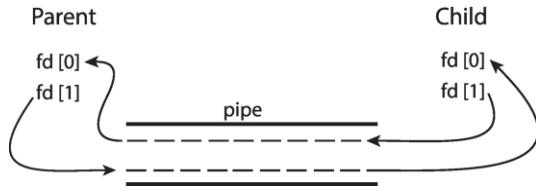






## Ordinary Pipes

- Ordinary Pipes allow communication in standard producer-consumer style
- Producer writes to one end (the **write-end** of the pipe)
- Consumer reads from the other end (the **read-end** of the pipe)
- Ordinary pipes are therefore unidirectional
- Require parent-child relationship between communicating processes

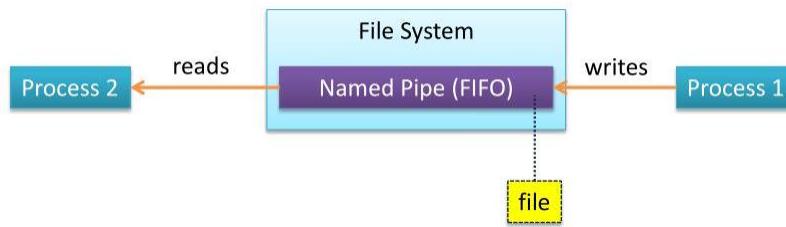


- Windows calls these **anonymous pipes**



## Named Pipes

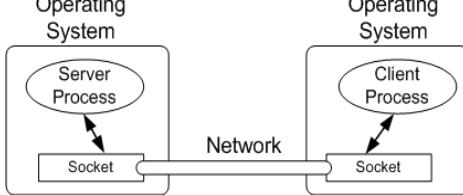
- Named Pipes are more powerful than ordinary pipes
- Communication is bidirectional
- No parent-child relationship is necessary between the communicating processes
- Several processes can use the named pipe for communication
- Provided on both UNIX and Windows systems





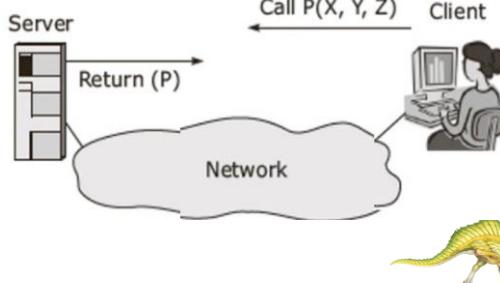
## Communications in Client-Server Systems

- Sockets



The diagram shows two separate boxes representing 'Operating System'. Each box contains an oval labeled 'Server Process' or 'Client Process' and a rectangle below it labeled 'Socket'. A horizontal line labeled 'Network' connects the two sockets.

- Remote Procedure Calls



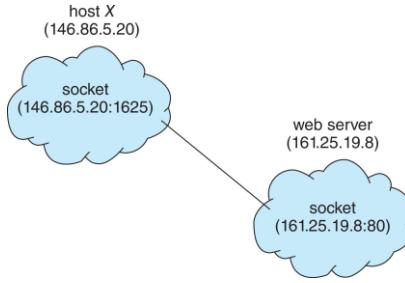
The diagram illustrates an RPC interaction. On the left, a server is shown as a vertical stack of rectangles. An arrow labeled 'Call P(X, Y, Z)' points from a 'Client' (represented by a person at a desk) towards the server. Another arrow labeled 'Return (P)' points back from the server to the client. The entire interaction is enclosed in a cloud-like shape labeled 'Network'.

**Operating System Concepts – 10<sup>th</sup> Edition**      3.62      Silberschatz, Galvin and Gagne ©2018



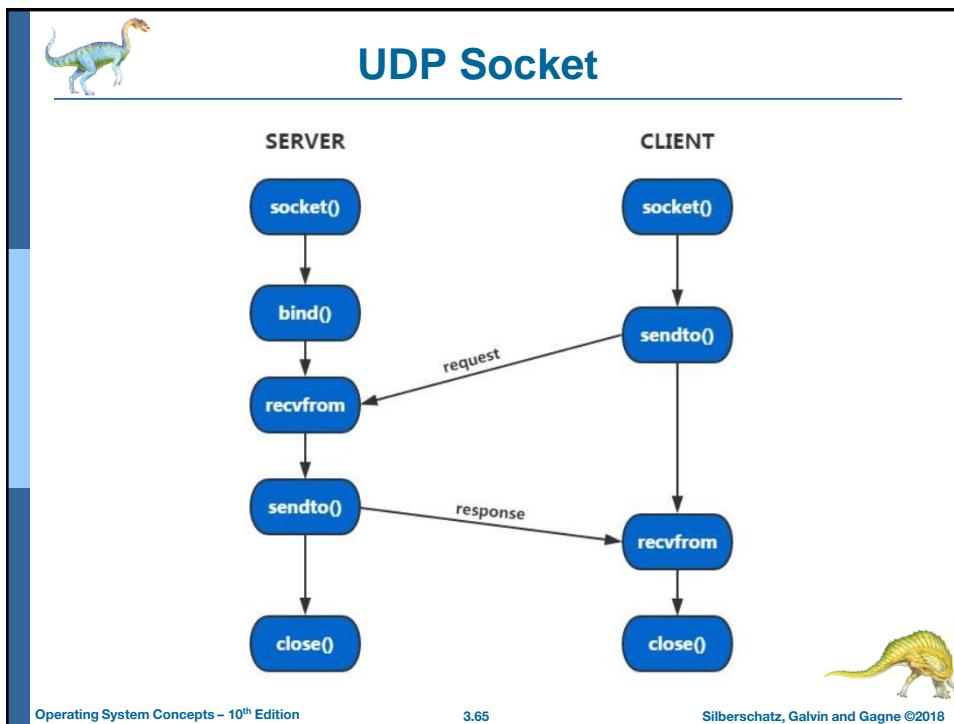
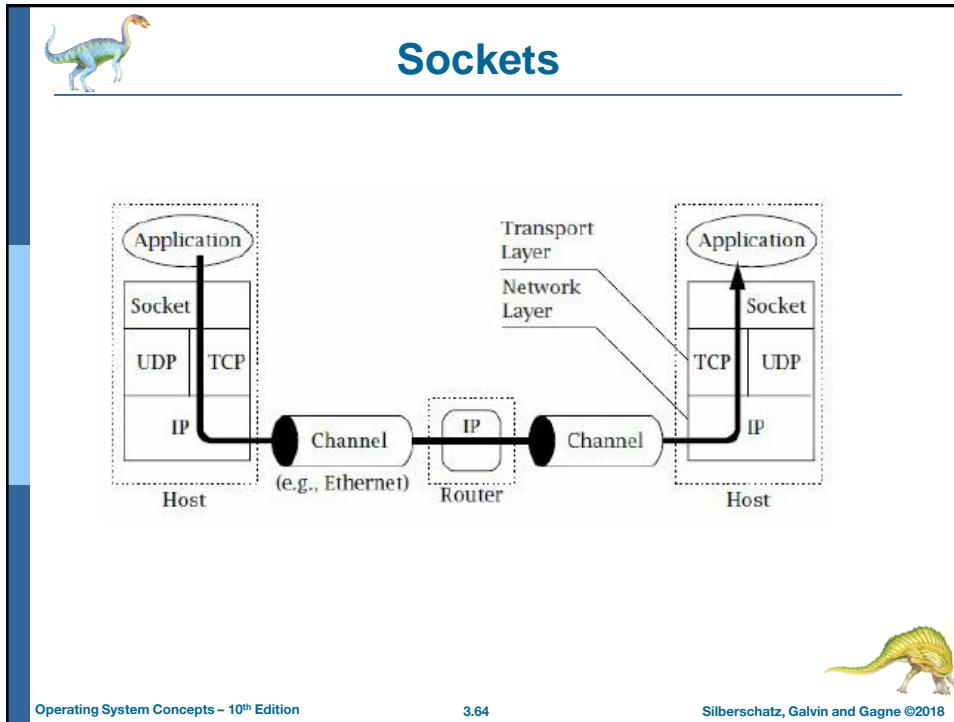
## Sockets

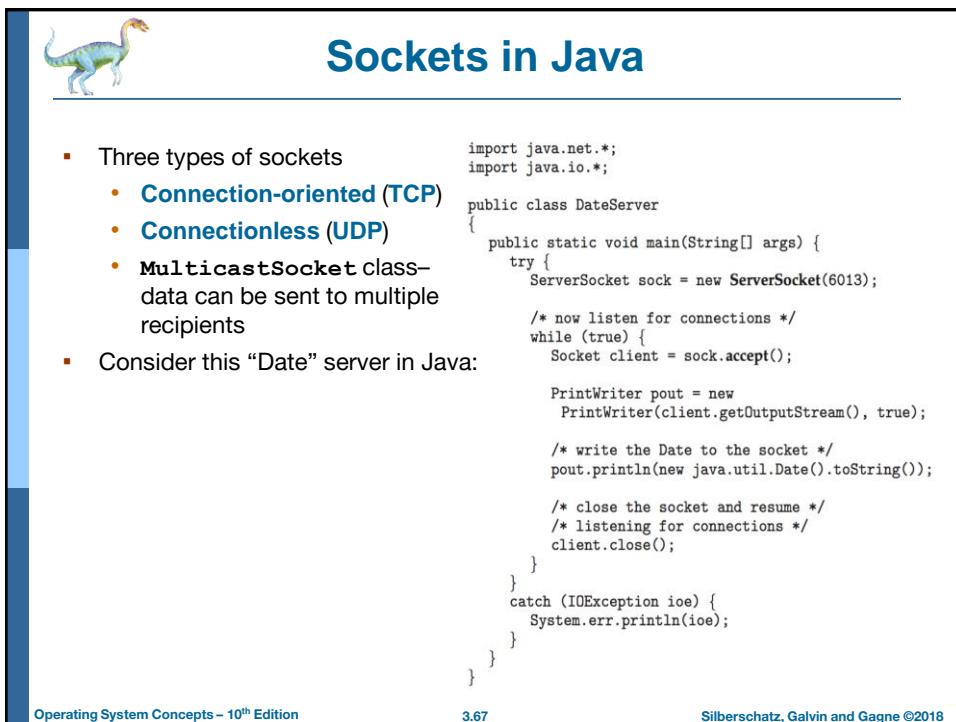
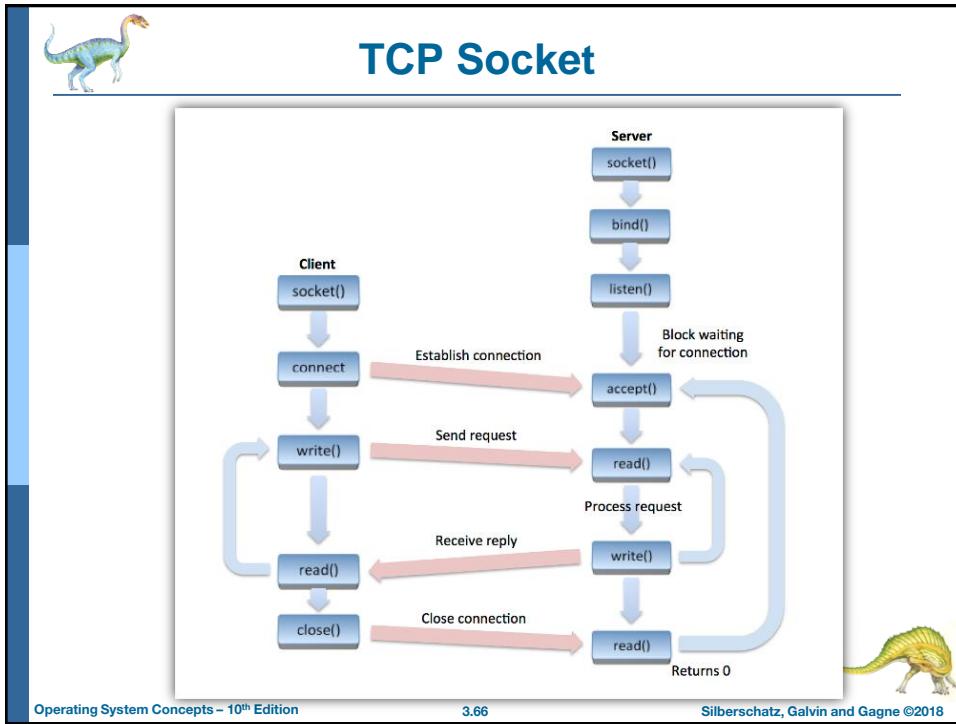
- A **socket** is defined as an endpoint for communication
- Concatenation of IP address and **port**
  - port is a number included at start of message packet to differentiate network services on a host
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets
- All ports below 1024 are **well known**, used for standard services
- Special IP address 127.0.0.1 (**loopback**) to refer to system on which process is running
- Communication:



The diagram shows two blue cloud shapes representing hosts. The top host is labeled 'host X (146.86.5.20)' and contains a box labeled 'socket (146.86.5.20:1625)'. An arrow points from this socket to a second host labeled 'web server (161.25.19.8)' which contains a box labeled 'socket (161.25.19.8:80)'. A small green dinosaur icon is located at the bottom right.

**Operating System Concepts – 10<sup>th</sup> Edition**      3.63      Silberschatz, Galvin and Gagne ©2018







## Sockets in Java

The equivalent Date client

```
import java.net.*;
import java.io.*;

public class DateClient
{
    public static void main(String[] args) {
        try {
            /* make connection to server socket */
            Socket sock = new Socket("127.0.0.1",6013);

            InputStream in = sock.getInputStream();
            BufferedReader bin = new
                BufferedReader(new InputStreamReader(in));

            /* read the date from the socket */
            String line;
            while ( (line = bin.readLine()) != null)
                System.out.println(line);

            /* close the socket connection*/
            sock.close();
        }
        catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}
```



Operating System Concepts – 10<sup>th</sup> Edition

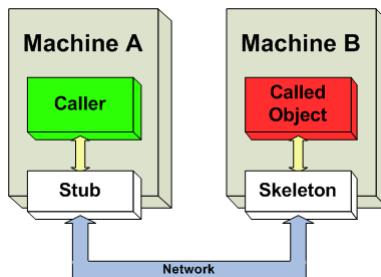
3.68

Silberschatz, Galvin and Gagne ©2018



## Remote Procedure Calls

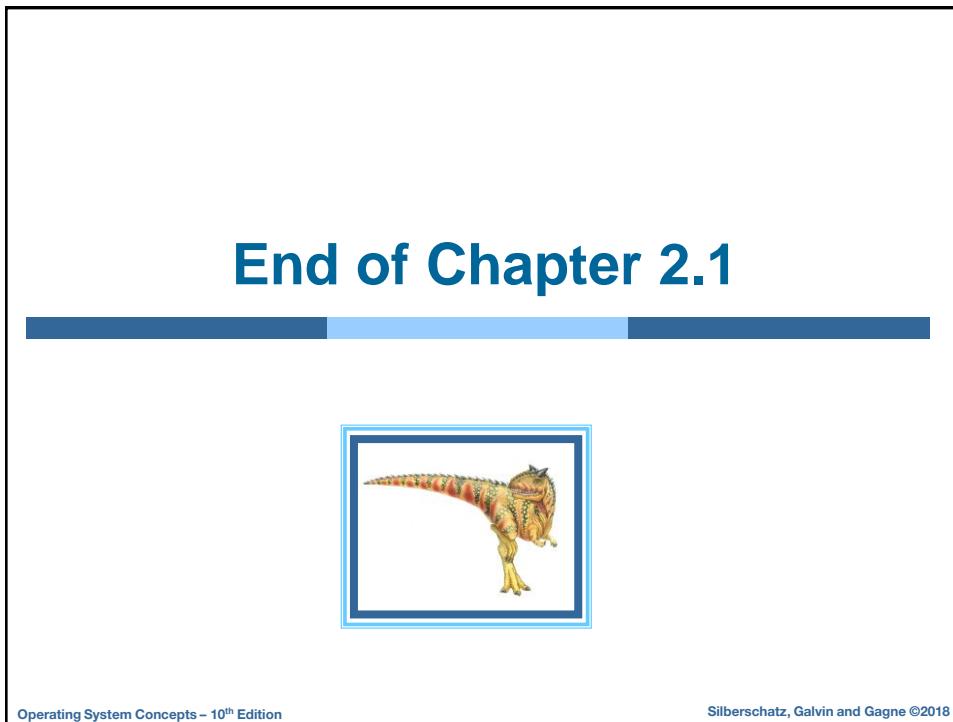
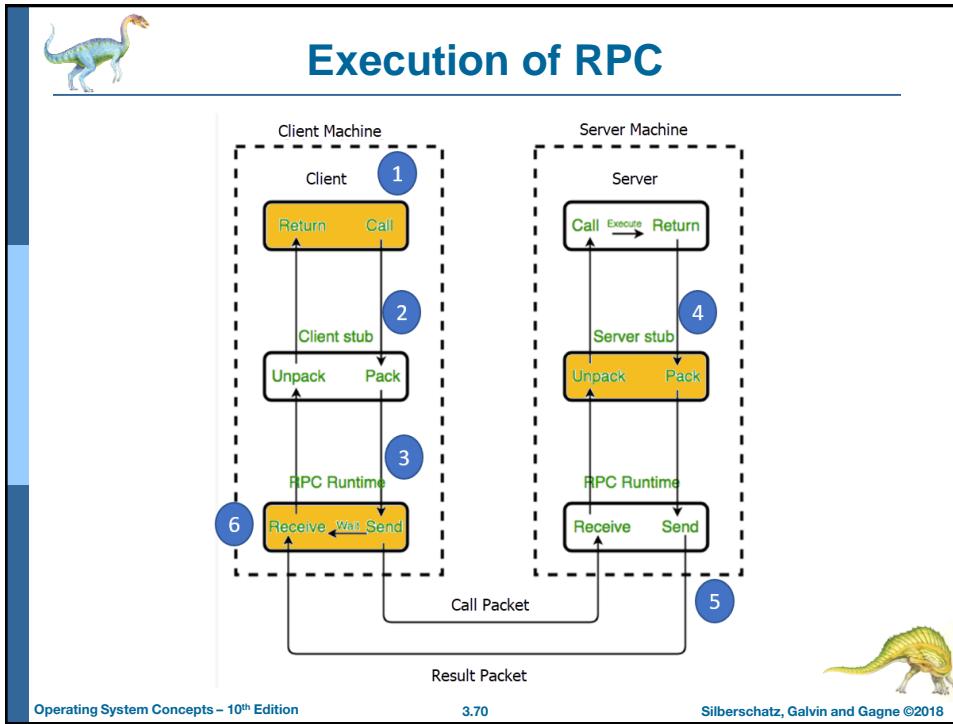
- RPC abstracts procedure calls between processes on networked systems
  - Again, uses ports for service differentiation
- **Stubs** – client-side proxy for the actual procedure on the server
- The client-side stub locates the server and **marshalls** the parameters
- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server



Operating System Concepts – 10<sup>th</sup> Edition

3.69

Silberschatz, Galvin and Gagne ©2018



## Chapter 2: Process Management

### 2.2. Threads & Concurrency



GV: Nguyễn Thị Thanh Vân

Operating System Concepts – 10<sup>th</sup> Edition

Silberschatz, Galvin and Gagne ©2018



### Outline

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues
- Operating System Examples



Operating System Concepts – 10<sup>th</sup> Edition

4.2

Silberschatz, Galvin and Gagne ©2018



## Objectives

- Identify the basic components of a thread, and contrast threads and processes
- Describe the benefits and challenges of designing multithreaded applications
- Illustrate different approaches to implicit threading including thread pools, fork-join, and Grand Central Dispatch
- Describe how the Windows and Linux operating systems represent threads
- Design multithreaded applications using the Pthreads, Java, and Windows threading APIs



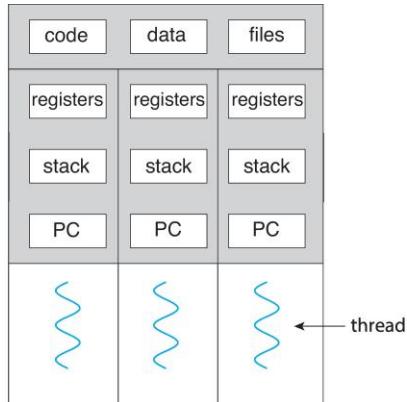
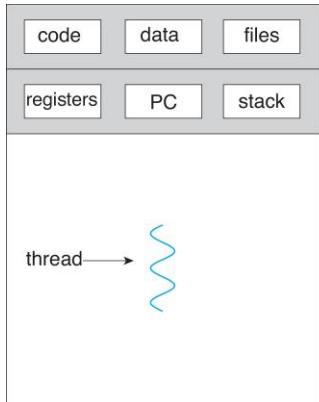
## Motivation

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks within the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

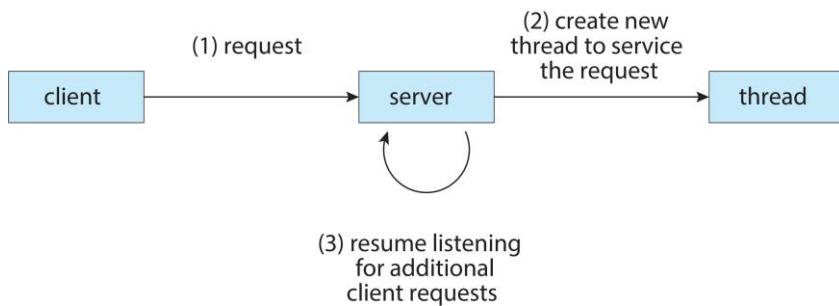




## Single and Multithreaded Processes



## Multithreaded Server Architecture





## Benefits

- **Responsiveness** – may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing** – threads share resources of process, easier than shared memory or message passing
- **Economy** – cheaper than process creation, thread switching lower overhead than context switching
- **Scalability** – process can take advantage of multicore architectures



## Multicore Programming

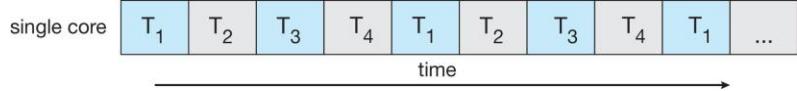
- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**
- **Parallelism** implies a system can perform more than one task simultaneously
- **Concurrency** supports more than one task making progress
  - Single processor / core, scheduler providing concurrency



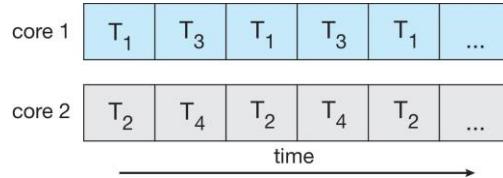


## Concurrency vs. Parallelism

- Concurrent execution on single-core system:



- Parallelism on a multi-core system:

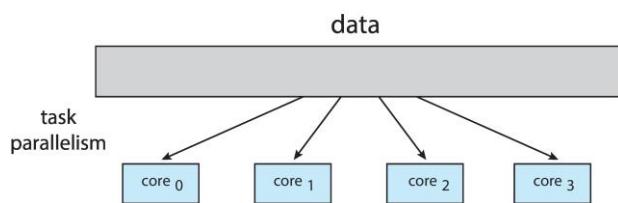
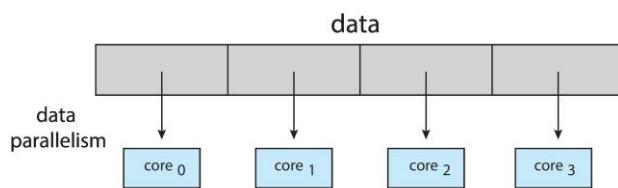


- Types of parallelism

- Data parallelism – distributes subsets of the same data across multiple cores, same operation on each
- Task parallelism – distributing threads across cores, each thread performing unique operation



## Data and Task Parallelism





## Amdahl's Law

- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
- S is serial portion
- N processing cores

$$\text{speedup} \leq \frac{1}{S + \frac{(1-S)}{N}}$$

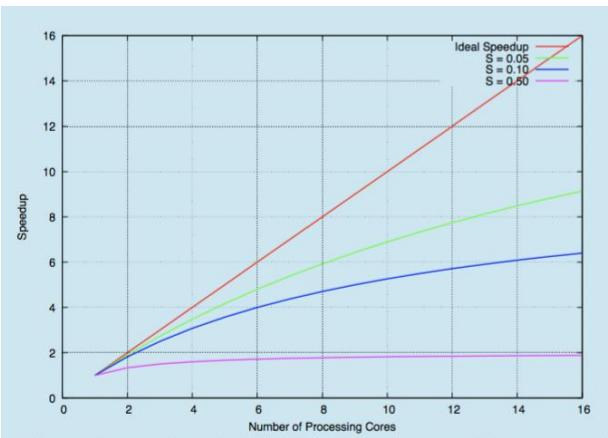
- That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
- As N approaches infinity, speedup approaches 1 / S

**Serial portion of an application has disproportionate effect on performance gained by adding additional cores**

- But does the law take into account contemporary multicore systems?



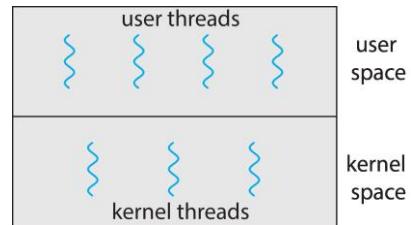
## Amdahl's Law





## User Threads and Kernel Threads

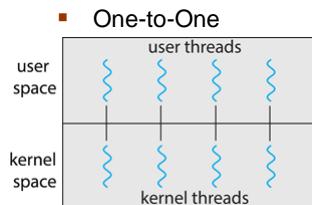
- **User threads** - management done by user-level threads library
- Three primary thread libraries:
  - POSIX **Pthreads**
  - Windows threads
  - Java threads



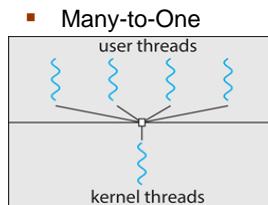
- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general-purpose operating systems, including:
  - Windows, Linux, Mac OS X, iOS, Android



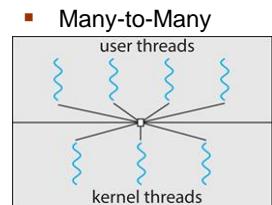
## Multithreading Models



- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - **Windows**
  - **Linux**



- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**



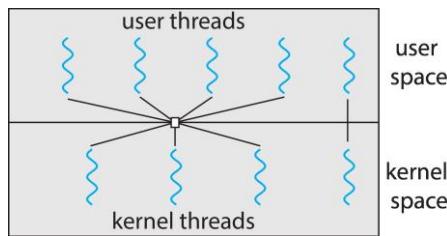
- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Windows with the ThreadFiber package
- Otherwise not very common





## Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread



## Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS





## Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- **Specification, not implementation**
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Linux & Mac OS X)



## Pthreads Example

```
#include <pthread.h>
#include <stdio.h>

#include <stdlib.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
    pthread_t tid; /* the thread identifier */
    pthread_attr_t attr; /* set of thread attributes */

    /* set the default attributes of the thread */
    pthread_attr_init(&attr);
    /* create the thread */
    pthread_create(&tid, &attr, runner, argv[1]);
    /* wait for the thread to exit */
    pthread_join(tid,NULL);

    printf("sum = %d\n",sum);
}
```





## Pthreads Example (Cont.)

```
/* The thread will execute in this function */
void *runner(void *param)
{
    int i, upper = atoi(param);
    sum = 0;

    for (i = 1; i <= upper; i++)
        sum += i;

    pthread_exit(0);
}
```



## Pthreads Code for Joining 10 Threads

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
    pthread_join(workers[i], NULL);
```





## Windows Multithreaded C Program

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */

/* The thread will execute in this function */
DWORD WINAPI Summation(LPVOID Param)
{
    DWORD Upper = *(DWORD*)Param;
    for (DWORD i = 1; i <= Upper; i++)
        Sum += i;
    return 0;
}
```



## Windows Multithreaded C Program (Cont.)

```
int main(int argc, char *argv[])
{
    DWORD ThreadId;
    HANDLE ThreadHandle;
    int Param;

    Param = atoi(argv[1]);
    /* create the thread */
    ThreadHandle = CreateThread(
        NULL, /* default security attributes */
        0, /* default stack size */
        Summation, /* thread function */
        &Param, /* parameter to thread function */
        0, /* default creation flags */
        &ThreadId); /* returns the thread identifier */

    /* now wait for the thread to finish */
    WaitForSingleObject(ThreadHandle, INFINITE);

    /* close the thread handle */
    CloseHandle(ThreadHandle);

    printf("sum = %d\n", Sum);
}
```





## Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface

```
public interface Runnable
{
    public abstract void run();
}
```

- Standard practice is to implement Runnable interface



## Java Threads

### Implementing Runnable interface:

```
class Task implements Runnable
{
    public void run() {
        System.out.println("I am a thread.");
    }
}
```

### Creating a thread:

```
Thread worker = new Thread(new Task());
worker.start();
```

### Waiting on a thread:

```
try {
    worker.join();
}
catch (InterruptedException ie) { }
```





## Java Executor Framework

- Rather than explicitly creating threads, Java also allows thread creation around the Executor interface:

```
public interface Executor
{
    void execute(Runnable command);
}
```

- The Executor is used as follows:

```
Executor service = new Executor;
service.execute(new Task());
```



## Java Executor Framework

```
import java.util.concurrent.*;

class Summation implements Callable<Integer>
{
    private int upper;
    public Summation(int upper) {
        this.upper = upper;
    }

    /* The thread will execute in this method */
    public Integer call() {
        int sum = 0;
        for (int i = 1; i <= upper; i++)
            sum += i;

        return new Integer(sum);
    }
}
```





## Java Executor Framework (Cont.)

```
public class Driver
{
    public static void main(String[] args) {
        int upper = Integer.parseInt(args[0]);

        ExecutorService pool = Executors.newSingleThreadExecutor();
        Future<Integer> result = pool.submit(new Summation(upper));

        try {
            System.out.println("sum = " + result.get());
        } catch (InterruptedException | ExecutionException ie) { }
    }
}
```



## Implicit Threading

- Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads
- Creation and management of threads done by compilers and run-time libraries rather than programmers
- Five methods explored
  - Thread Pools
  - Fork-Join
  - OpenMP
  - Grand Central Dispatch
  - Intel Threading Building Blocks





## Thread Pools

- Create a number of threads in a pool where they await work
- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool
  - Separating task to be performed from mechanics of creating task allows different strategies for running task
    - ▶ i.e., Tasks could be scheduled to run periodically
- Windows API supports thread pools:

```
DWORD WINAPI PoolFunction(VOID Param) {  
    /*  
     * this function runs as a separate thread.  
     */  
}
```



## Java Thread Pools

- Three factory methods for creating thread pools in Executors class:
  - static ExecutorService newSingleThreadExecutor()
  - static ExecutorService newFixedThreadPool(int size)
  - static ExecutorService newCachedThreadPool()





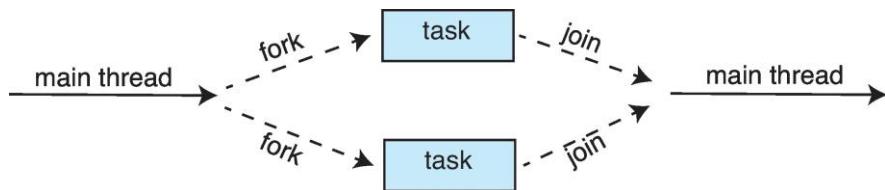
## Java Thread Pools (Cont.)

```
import java.util.concurrent.*;  
  
public class ThreadPoolExample  
{  
    public static void main(String[] args) {  
        int numTasks = Integer.parseInt(args[0].trim());  
  
        /* Create the thread pool */  
        ExecutorService pool = Executors.newCachedThreadPool();  
  
        /* Run each task using a thread in the pool */  
        for (int i = 0; i < numTasks; i++)  
            pool.execute(new Task());  
  
        /* Shut down the pool once all threads have completed */  
        pool.shutdown();  
    }  
}
```



## Fork-Join Parallelism

- Multiple threads (tasks) are **forked**, and then **joined**.





## Fork-Join Parallelism

- General algorithm for fork-join strategy:

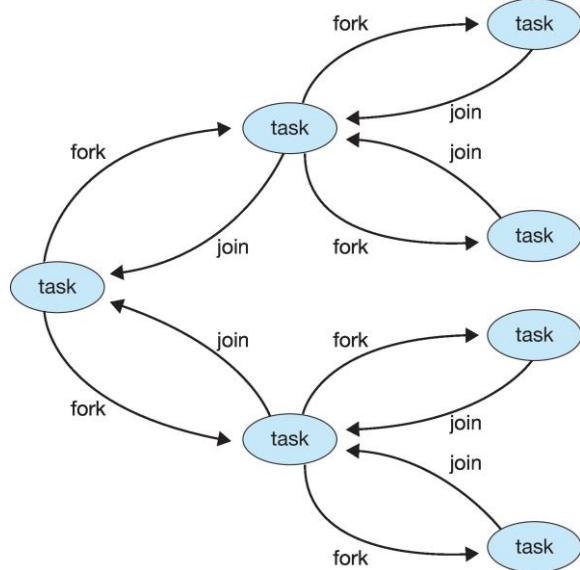
```
Task(problem)
    if problem is small enough
        solve the problem directly
    else
        subtask1 = fork(new Task(subset of problem)
        subtask2 = fork(new Task(subset of problem)

        result1 = join(subtask1)
        result2 = join(subtask2)

        return combined results
```



## Fork-Join Parallelism





## Fork-Join Parallelism in Java

```
ForkJoinPool pool = new ForkJoinPool();
// array contains the integers to be summed
int[] array = new int[SIZE];

SumTask task = new SumTask(0, SIZE - 1, array);
int sum = pool.invoke(task);
```



## Fork-Join Parallelism in Java

```
import java.util.concurrent.*;

public class SumTask extends RecursiveTask<Integer>
{
    static final int THRESHOLD = 1000;

    private int begin;
    private int end;
    private int[] array;

    public SumTask(int begin, int end, int[] array) {
        this.begin = begin;
        this.end = end;
        this.array = array;
    }

    protected Integer compute() {
        if (end - begin < THRESHOLD) {
            int sum = 0;
            for (int i = begin; i <= end; i++)
                sum += array[i];
            return sum;
        } else {
            int mid = (begin + end) / 2;

            SumTask leftTask = new SumTask(begin, mid, array);
            SumTask rightTask = new SumTask(mid + 1, end, array);

            leftTask.fork();
            rightTask.fork();

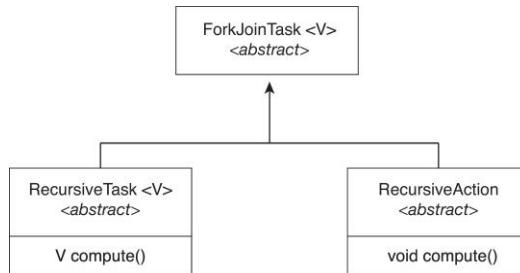
            return rightTask.join() + leftTask.join();
        }
    }
}
```





## Fork-Join Parallelism in Java

- The `ForkJoinTask` is an abstract base class
- `RecursiveTask` and `RecursiveAction` classes extend `ForkJoinTask`
- `RecursiveTask` returns a result (via the return value from the `compute()` method)
- `RecursiveAction` does not return a result



## OpenMP

- Set of compiler directives and an API for C, C++, FORTRAN
- Provides support for parallel programming in shared-memory environments
- Identifies **parallel regions** – blocks of code that can run in parallel

`#pragma omp parallel`

Create as many threads as there are cores

```
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    /* sequential code */

    #pragma omp parallel
    {
        printf("I am a parallel region.");
    }

    /* sequential code */

    return 0;
}
```





## Run the Loop in Parallel

- Run the for loop in parallel

```
#pragma omp parallel for
for (i = 0; i < N; i++) {
    c[i] = a[i] + b[i];
}
```



## Grand Central Dispatch

- Apple technology for macOS and iOS operating systems
- Extensions to C, C++ and Objective-C languages, API, and run-time library
- Allows identification of parallel sections
- Manages most of the details of threading
- Block is in “^{ }” :

```
^{ printf("I am a block"); }
```

- Blocks placed in dispatch queue
  - Assigned to available thread in thread pool when removed from queue





## Grand Central Dispatch

- Two types of dispatch queues:
  - **serial** – blocks removed in FIFO order, queue is per process, called **main queue**
    - ▶ Programmers can create additional serial queues within program
  - **concurrent** – removed in FIFO order but several may be removed at a time
    - ▶ Four system wide queues divided by quality of service:
      - QOS\_CLASS\_USER\_INTERACTIVE
      - QOS\_CLASS\_USER\_INITIATED
      - QOS\_CLASS\_USER.Utility
      - QOS\_CLASS\_USER\_BACKGROUND



## Grand Central Dispatch

- For the Swift language a task is defined as a closure – similar to a block, minus the caret
- Closures are submitted to the queue using the `dispatch_async()` function:

```
let queue = dispatch_get_global_queue
           (QOS_CLASS_USER_INITIATED, 0)

dispatch_async(queue, { print("I am a closure.") })
```





## Intel Threading Building Blocks (TBB)

- Template library for designing parallel C++ programs
- A serial version of a simple for loop

```
for (int i = 0; i < n; i++) {  
    apply(v[i]);  
}
```

- The same for loop written using TBB with `parallel_for` statement:

```
parallel_for (size_t(0), n, [=](size_t i) {apply(v[i]);});
```



## Threading Issues

- Semantics of `fork()` and `exec()` system calls
- Signal handling
  - Synchronous and asynchronous
- Thread cancellation of target thread
  - Asynchronous or deferred
- Thread-local storage
- Scheduler Activations





## Semantics of fork() and exec()

- Does `fork()` duplicate only the calling thread or all threads?
  - Some UNIXes have two versions of fork
- `exec()` usually works as normal – replace the running process including all threads



## Signal Handling

- **Signals** are used in UNIX systems to notify a process that a particular event has occurred.
- A **signal handler** is used to process signals
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal is handled by one of two signal handlers:
    1. default
    2. user-defined
- Every signal has **default handler** that kernel runs when handling signal
  - **User-defined signal handler** can override default
  - For single-threaded, signal delivered to process





## Signal Handling (Cont.)

- Where should a signal be delivered for multi-threaded?
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread in the process
  - Deliver the signal to certain threads in the process
  - Assign a specific thread to receive all signals for the process



## Thread Cancellation

- Terminating a thread before it has finished
- Thread to be canceled is **target thread**
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
- Pthread code to create and cancel a thread:

```
pthread_t tid;  
  
/* create the thread */  
pthread_create(&tid, 0, worker, NULL);  
  
...  
  
/* cancel the thread */  
pthread_cancel(tid);  
  
/* wait for the thread to terminate */  
pthread_join(tid, NULL);
```





## Thread Cancellation (Cont.)

- Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state

Mode	State	Type
Off	Disabled	—
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

- If thread has cancellation disabled, cancellation remains pending until thread enables it
- Default type is deferred
  - Cancellation only occurs when thread reaches **cancellation point**
    - i.e., `pthread_cancel()`
    - Then **cleanup handler** is invoked
- On Linux systems, thread cancellation is handled through signals



## Thread Cancellation in Java

- Deferred cancellation uses the `interrupt()` method, which sets the interrupted status of a thread.

```
Thread worker;  
.  
.  
.  
/* set the interruption status of the thread */  
worker.interrupt();
```

- A thread can then check to see if it has been interrupted:

```
while (!Thread.currentThread().isInterrupted()) {  
    .  
}
```





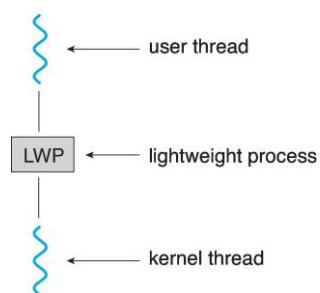
## Thread-Local Storage

- **Thread-local storage (TLS)** allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- Different from local variables
  - Local variables visible only during single function invocation
  - TLS visible across function invocations
- Similar to **static** data
  - TLS is unique to each thread



## Scheduler Activations

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Typically use an intermediate data structure between user and kernel threads – **lightweight process (LWP)**
  - Appears to be a virtual processor on which process can schedule user thread to run
  - Each LWP attached to kernel thread
  - How many LWPs to create?
- Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the **upcall handler** in the thread library
- This communication allows an application to maintain the correct number kernel threads





## Operating System Examples

- Windows Threads
- Linux Threads



## Windows Threads

- Windows API – primary API for Windows applications
- Implements the one-to-one mapping, kernel-level
- Each thread contains
  - A thread id
  - Register set representing state of processor
  - Separate user and kernel stacks for when thread runs in user mode or kernel mode
  - Private data storage area used by run-time libraries and dynamic link libraries (DLLs)
- The register set, stacks, and private storage area are known as the **context** of the thread



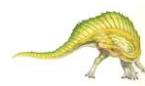
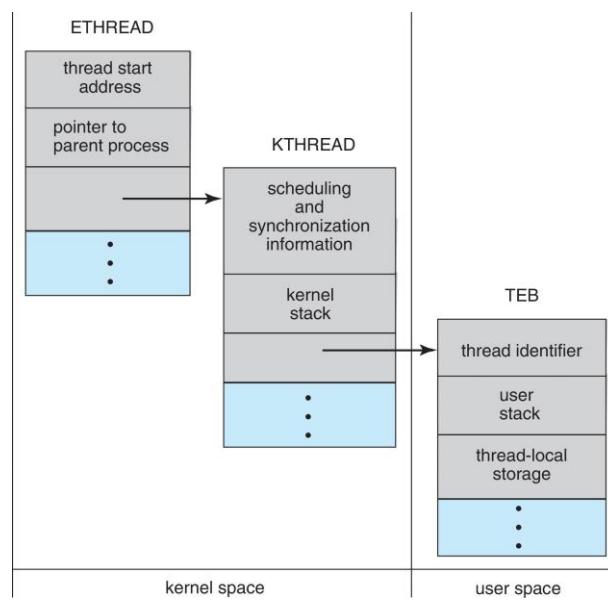


## Windows Threads (Cont.)

- The primary data structures of a thread include:
  - ETHREAD (executive thread block) – includes pointer to process to which thread belongs and to KTHREAD, in kernel space
  - KTHREAD (kernel thread block) – scheduling and synchronization info, kernel-mode stack, pointer to TEB, in kernel space
  - TEB (thread environment block) – thread id, user-mode stack, thread-local storage, in user space



## Windows Threads Data Structures





## Linux Threads

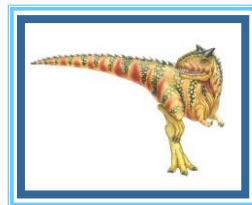
- Linux refers to them as **tasks** rather than **threads**
- Thread creation is done through `clone()` system call
- `clone()` allows a child task to share the address space of the parent task (process)
  - Flags control behavior

flag	meaning
<code>CLONE_FS</code>	File-system information is shared.
<code>CLONE_VM</code>	The same memory space is shared.
<code>CLONE_SIGHAND</code>	Signal handlers are shared.
<code>CLONE_FILES</code>	The set of open files is shared.

- `struct task_struct` points to process data structures (shared or unique)



## End of Chapter 2.2



## Chapter 2: Process Management

### 2.3. CPU Scheduling



GV: Nguyễn Thị Thanh Vân

Operating System Concepts – 10<sup>th</sup> Edition

Silberschatz, Galvin and Gagne ©2018

### Outline

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms



Operating System Concepts – 10<sup>th</sup> Edition

5a.2

Silberschatz, Galvin and Gagne ©2018



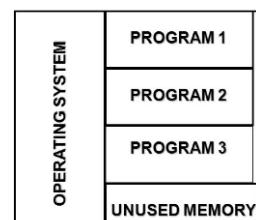
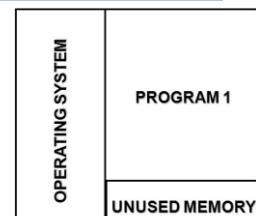
## Objectives

- Describe various CPU scheduling algorithms
- Assess CPU scheduling algorithms based on scheduling criteria
- Explain the issues related to multiprocessor and multicore scheduling
- Describe various real-time scheduling algorithms
- Describe the scheduling algorithms used in the Windows, Linux, and Solaris operating systems
- Apply modeling and simulations to evaluate CPU scheduling algorithms



## Basic Concepts

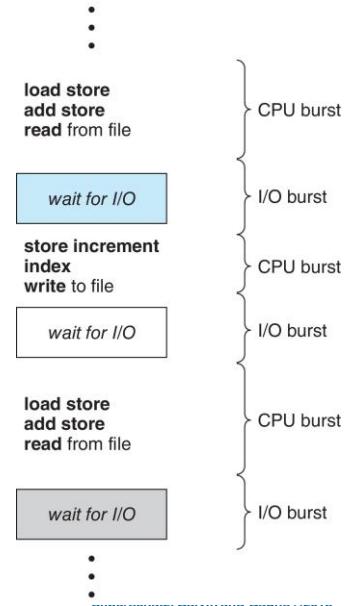
- Single program: A process is executed until it must wait for some I/O request.
  - => the CPU then just sits idle.
  - waiting time is wasted; no useful work is accomplished
- Multiprogramming: to maximize CPU utilization.
  - Processes are kept in memory at one time.
  - When one process has to wait, the OS takes the CPU away from that process and gives the CPU to another process.
  - => keeping the CPU busy is extended to all processing cores on the system.
- **CPU scheduling:** central to OS design
  - Almost all computer resources are scheduled
  - CPU is one of the primary





## Basic Concepts - CPU–I/O Burst Cycle

- **CPU scheduling:** Process execution consists of
  - a **cycle** of CPU execution and I/O wait
- Processes have two states (alternate): in fig
  - **CPU burst** followed by **I/O burst**
- Execution
  - begins with a CPU burst.,
  - I/O burst
  - ...
  - the final CPU burst ends with a system request to terminate execution
- The durations of CPU bursts have been measured extensively



Operating System Concepts – 10<sup>th</sup> Edition

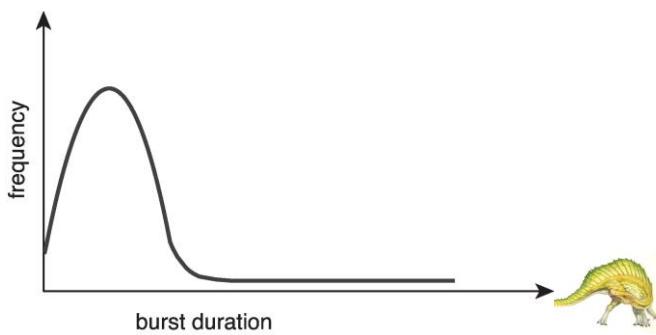
5a.5

Silberschatz, Galvin and Gagne ©2018



## Histogram of CPU-burst Times

- CPU burst distribution is of main concern
  - An I/O-bound program typically has many short CPU bursts.
  - A CPU-bound program might have a few long CPU bursts.
- The curve is with a large number of short CPU bursts and a small number of long CPU bursts.
- This distribution can be important when implementing a CPU-scheduling
- Histogram



Operating System Concepts – 10<sup>th</sup> Edition

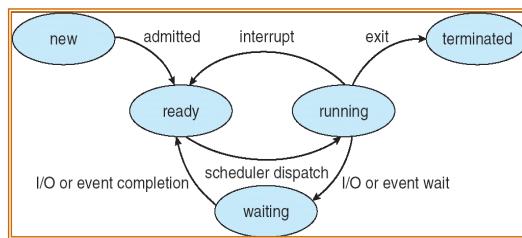
5a.6

Silberschatz, Galvin and Gagne ©2018



## CPU Scheduler

- The **CPU scheduler** selects from among the processes in ready queue, and allocates a CPU core to one of them
  - The ready queue may be ordered in various ways: FIFO, FCFS, SJF
  - The records in the queues are generally process control blocks (PCBs) of the processes
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state => need schedule
  2. Switches from running to ready state => a choice
  3. Switches from waiting to ready => a choice
  4. Terminates => need schedule



Operating System Concepts – 10<sup>th</sup> Edition

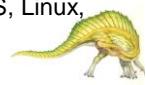
5a.7

Silberschatz, Galvin and Gagne ©2018



## Preemptive and Nonpreemptive Scheduling

- For situations 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution.
  - the scheduling scheme is **nonpreemptive** or cooperative.
- For situations 2 and 3, however, there is a choice.
  - **Preemptive**
- **Nonpreemptive (không được ưu tiên trước)**
  - a process terminates, or a process switches running to waiting state
  - once the resources (CPU cycles) is allocated to a process, the process holds the CPU till it gets terminated or it reaches a waiting
- **Preemptive** (có ưu tiên)
  - Once the CPU has been allocated, the process keeps the CPU until it releases it either by terminating or by switching to the waiting state.
  - What is the potential problem?
- Virtually all modern operating systems including Windows, MacOS, Linux, and UNIX use preemptive scheduling algorithms.



Operating System Concepts – 10<sup>th</sup> Edition

5a.8

Silberschatz, Galvin and Gagne ©2018



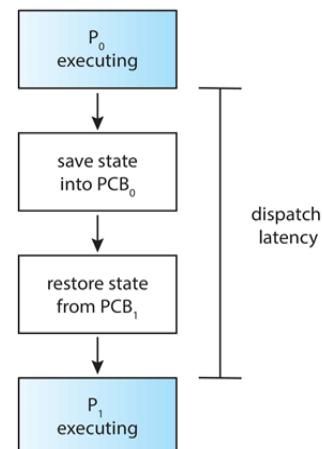
## Preemptive Scheduling and Race Conditions

- Preemptive scheduling can result in race conditions when data are shared among several processes.
- Consider the case of two processes that share data (Race Conditions).
  - While one process is updating the data, it is preempted so that the second process can run.
  - The second process then tries to read the data, which are in an inconsistent state.
- We saw this in the bounded buffer example
- This issue will be explored in detail in Chapter 6.



## Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the CPU 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

- Max ▪ **CPU utilization** – keep the CPU as busy as possible
- Max ▪ **Throughput** – # of processes that complete their execution per time unit
- Min ▪ **Turnaround time** – amount of time to execute a particular process
- Min ▪ **Waiting time** – amount of time a process has been waiting in the ready queue
- Min ▪ **Response time** – amount of time it takes from when a request was submitted until the first response is produced.

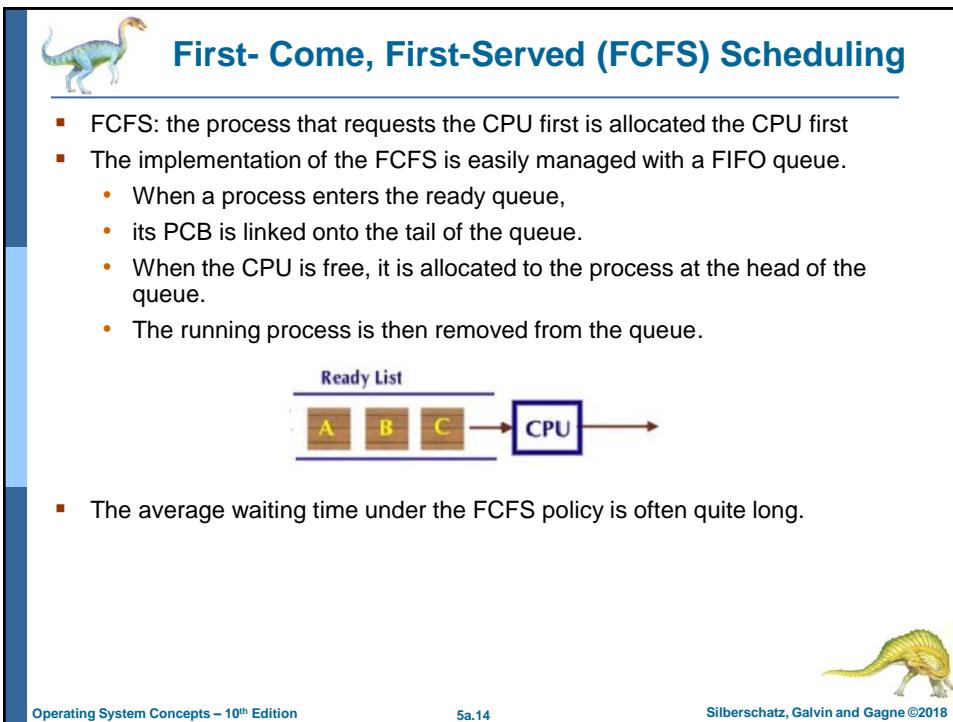
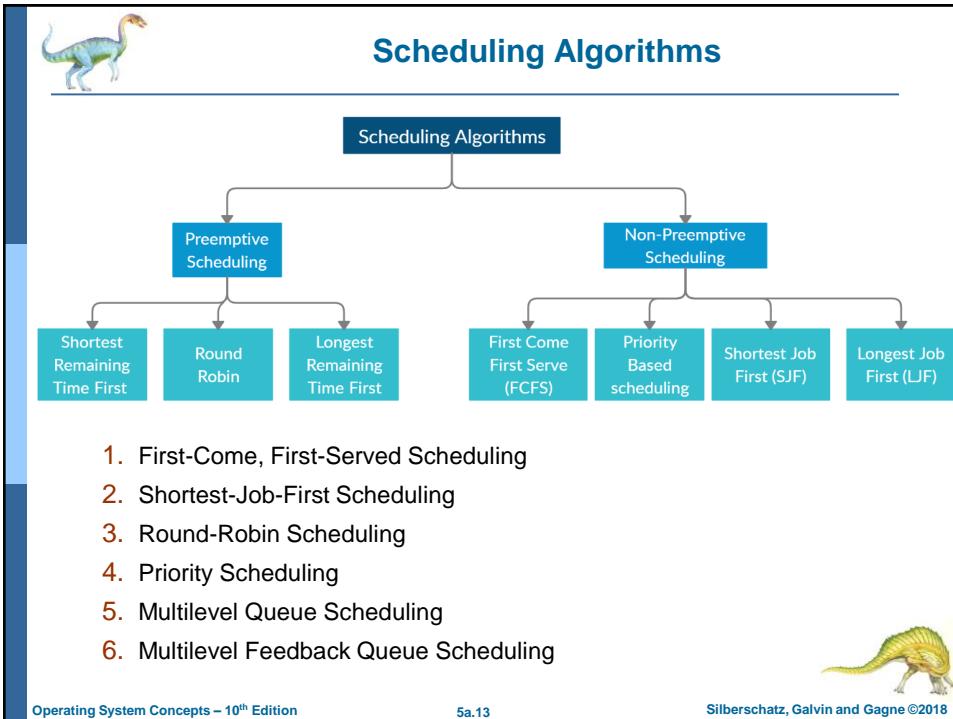
### Optimization Criteria for Scheduling Algorithms



## Outline

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms







## First-Come, First-Served (FCFS) Scheduling

- Example with 3 processes

Process	Burst Time
$P_1$	24
$P_2$	3
$P_3$	3

- Suppose that the processes arrive in the order:  $P_1, P_2, P_3$   
The Gantt Chart for the above schedule is:



- Waiting time for  $P_1 = 0; P_2 = 24; P_3 = 27$
- Average waiting time:  $(0 + 24 + 27)/3 = 17$
- Operation: Once the CPU has been allocated to a process  $\Rightarrow$  it keeps the CPU until it releases the CPU, either by terminating or by requesting I/O.
- The FCFS particularly troublesome for interactive systems, where it is important that each process get a share of the CPU at regular intervals.
  - FCFS: Nonpreemptive



## FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order:  $P_2, P_3, P_1$ . The Gantt chart :



- Waiting time for  $P_1 = 6; P_2 = 0; P_3 = 3$
- Average waiting time:  $(6 + 0 + 3)/3 = 3 \Rightarrow$  Much better than previous case
- Consider one CPU-bound and many I/O-bound processes
  - The CPU-bound process will get and hold the CPU.
  - all the other processes will finish their I/O and will move into the ready queue, waiting for the CPU. (I/O devices are idle)
  - $\Rightarrow$  the CPU-bound process finishes its CPU burst and moves to an I/O device. All the I/O-bound processes, execute quickly and move back to the I/O queues (CPU sits idle)
- Convoys effect** - as all the other processes wait for the one big process to get off the CPU. (longer processes were allowed to go first): **better**
  - This effect results in lower CPU and device utilization than might be possible if the shorter processes were allowed to go first.





## Scheduling Algorithms

1. First-Come, First-Served Scheduling
2. Shortest-Job-First Scheduling
3. Round-Robin Scheduling
4. Priority Scheduling
5. Multilevel Queue Scheduling
6. Multilevel Feedback Queue Scheduling



## Shortest-Job-First (SJF) Scheduling

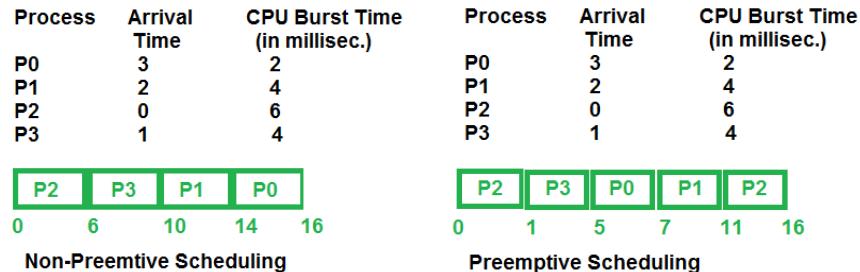
- SJF associates with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
  - If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie
- SJF is optimal – gives minimum average waiting time for a given set of processes:
  - Moving a short price before a long one
- How do we determine the length of the next CPU burst?
  - Could ask the user
  - Estimate





## Shortest Remaining Time First Scheduling

- The SJF algorithm can be either preemptive or nonpreemptive.
  - Non-preemptive SJF algorithm will allow the currently running process to finish its CPU burst.
  - A preemptive SJF algorithm will preempt the **currently executing** process, (a new process arrives with less work than the remaining time of currently executing proc



- Detail, later =>



Operating System Concepts – 10<sup>th</sup> Edition

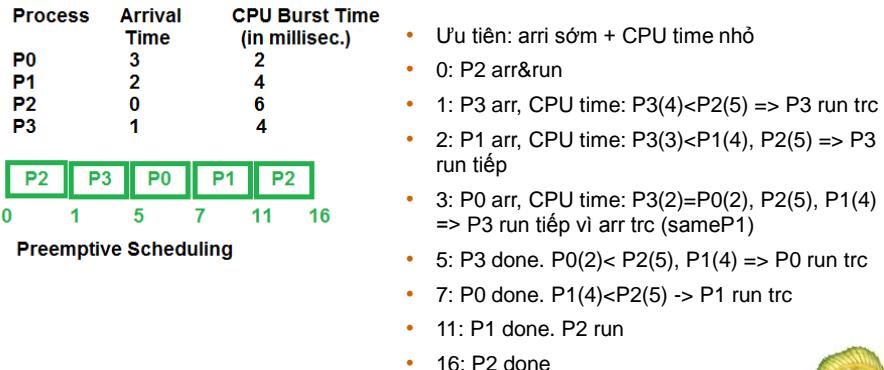
5a.19

Silberschatz, Galvin and Gagne ©2018



## Shortest Remaining Time First Scheduling

- Preemptive SJF:** called shortest-remaining-time-first scheduling
  - Whenever a new process arrives in the ready queue, the decision on which process to schedule next is redone using the SJN algorithm.
  - Is SRT more “optimal” than SJN in terms of the minimum average waiting time for a given set of processes?



Operating System Concepts – 10<sup>th</sup> Edition

5a.20

Silberschatz, Galvin and Gagne ©2018



## Example of Shortest-remaining-time-first

- Preemption to the analysis - arr early + CPU time small

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 -



- Analysis:

- 0: P1 arr & run
- 1: P2 arr. CPU Time:  $P_2(4) < P_1(7)$  => P2 run
- 2: P3 arr. CPU time:  $P_2(3) < P_3(9)$  &  $P_1(7)$  => P2 run
- 3: P4 arr. CPU time:  $P_2(2) < P_3(9) & P_1(7) & P_4(5)$  => P2 run
- 5: P2 done. CPU time:  $P_4(5) < P_3(9) & P_1(7)$  => P4 run
- 10: P4 done. CPU time:  $P_1(7) < P_3(9)$  => P1 run
- 17: P1 done. P3 run; 26: P3 done

- Average waiting time =  $[(10-1)+(1-1)+(17-2)+(5-3)]/4 = 26/4 = 6.5\text{ms}$



## Example of Shortest-remaining-time-first

- NonPreemption to the analysis – no break

Process    Arrival Time    Burst Time

$P_1$	0	8
$P_2$	1	4
$P_3$	2	9
$P_4$	3	5

- NonPreemptive SJF Gantt Chart



- Analysis:

- 0: P1 arr & run
- 8: P1 done. Queue: P2, P3, P4. CPU time:  $P_2(4) < P_3(9) & P_4(5)$  => P2 run
- 12: P2 done. Queue: P3, P4. CPU time:  $P_4(5) < P_3(9)$  => P4 run
- 17: P4 done. => P3 run
- 26: P3 done

- average waiting time of 7.75 milliseconds =  $(0+ (8-1)+(12-3)+(17-2))/4$

- **FCFS** result in an average waiting time  $8.75= (0+7+10+18)/4$





## Determining Length of Next CPU Burst

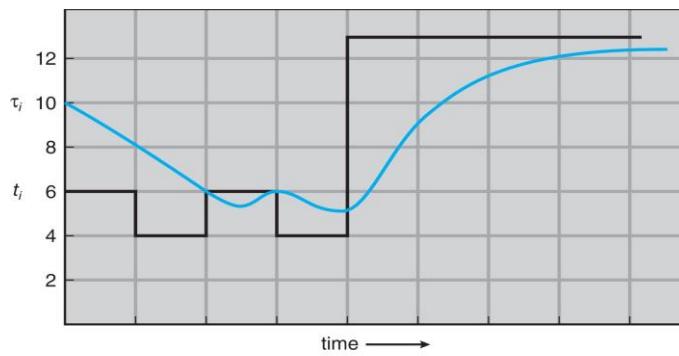
- Can only estimate the length – should be similar to the previous one
  - Then pick process with shortest predicted next CPU burst
- Can be done by using the length of previous CPU bursts, using exponential averaging
- Equation:

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

- $\tau_{n+1}$  = giá trị dự đoán cho thời gian sử dụng CPU tiếp sau
- $t_n$  = thời gian thực tế của sự sử dụng CPU thứ n
- $\alpha$ ,  $0 \leq \alpha \leq 1$
- $\tau_0$  là một hằng số



## Prediction of the Length of the Next CPU Burst



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

Figure shows an exponential average with  $\alpha = 1/2$  and  $\tau_0 = 10$





## Examples of Exponential Averaging

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



## Scheduling Algorithms

1. First-Come, First-Served Scheduling
2. Shortest-Job-First Scheduling
3. Round-Robin Scheduling
4. Priority Scheduling
5. Multilevel Queue Scheduling
6. Multilevel Feedback Queue Scheduling





## Round Robin (RR)

- RR: similar to FCFS scheduling, but preemption is added to enable the system to switch between processes
- Each process gets a small unit of CPU time (**time quantum  $q$**  - *định lượng thời gian*),
  - usually 10-100 milliseconds.
  - After this time has elapsed, the process is preempted and added to the end of the ready queue.
- 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 at once.
  - No process waits more than  $(n-1)q$  time units.
- Timer interrupts every quantum to schedule next process



## Example of RR with Time Quantum = 4

<u>Process</u>	<u>Burst Time</u>
$P_1$	24
$P_2$	3
$P_3$	3

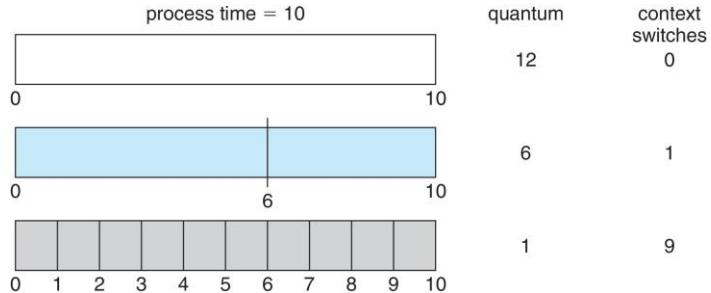
- The Gantt chart is:

$P_1$	$P_2$	$P_3$	$P_1$	$P_1$	$P_1$	$P_1$	$P_1$
0	4	7	10	14	18	22	26
- Waiting time for
  - $P_1: 3(p_2)+3(p_3)=6$
  - $P_2: 4(p_1)=4$
  - $P_3: 4(p_1)+3(p_2)=7$
- average waiting time  $=(6+4+7)/3 = 5.67\text{ms}$
- Typically, higher average turnaround than SJF, but better **response**
- $q$  should be large compared to context switch time
  - $q$  usually 10 milliseconds to 100 milliseconds,
  - Context switch < 10 microseconds





## Time Quantum and Context Switch Time

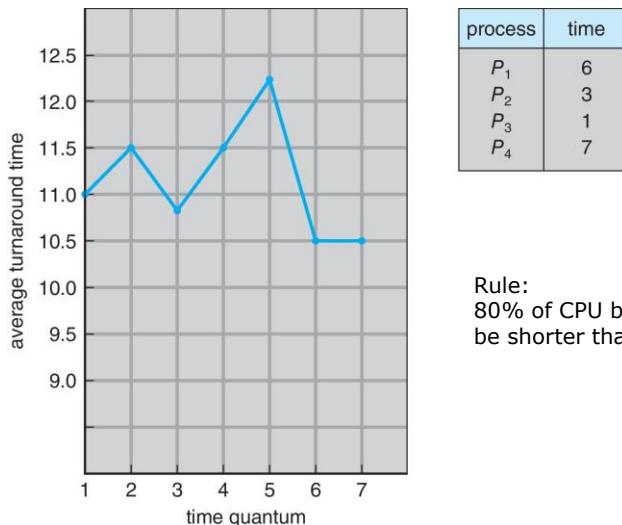


### Performance

$q$  large  $\Rightarrow$  FIFO (same FCFS)  
 $q$  small  $\Rightarrow$   $q$  must be large with respect to context switch, otherwise overhead is too high,  
=> performance decrease



## Turnaround Time Varies With The Time Quantum



Rule:  
80% of CPU bursts should  
be shorter than  $q$





## Scheduling Algorithms

1. First-Come, First-Served Scheduling
2. Shortest-Job-First Scheduling
3. Round-Robin Scheduling
4. **Priority Scheduling**
5. Multilevel Queue Scheduling
6. Multilevel Feedback Queue Scheduling



## Priority Scheduling

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





## 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  $(0 + 1 + 6 + 16 + 18)/5 = 8.2$

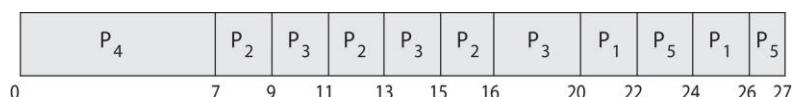


## Priority Scheduling w/ Round-Robin

- Run the process with the **highest priority**. Processes with the **same priority run round-robin**
- Example:

Process	Burst Time	Priority
$P_1$	4	3
$P_2$	5	2
$P_3$	8	2
$P_4$	7	1
$P_5$	3	3

- Gantt Chart with time quantum = 2





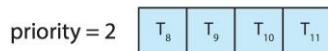
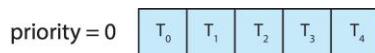
## Scheduling Algorithms

1. First-Come, First-Served Scheduling
2. Shortest-Job-First Scheduling
3. Round-Robin Scheduling
4. Priority Scheduling
5. Multilevel Queue Scheduling
6. Multilevel Feedback Queue Scheduling



## Multilevel Queue

- The ready queue consists of multiple queues
- Example:
  - Priority scheduling, where each priority has its separate queue.
  - Schedule the process in the highest-priority queue!



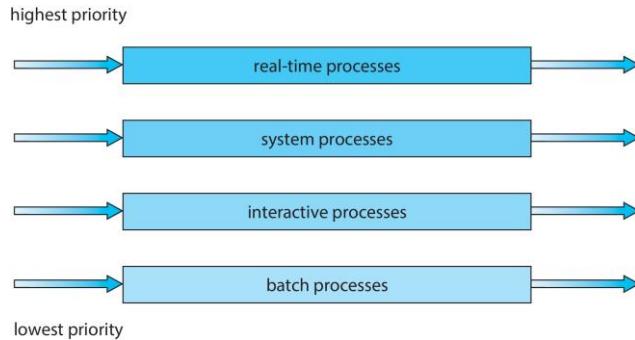
●  
●  
●





## Multilevel Queue

- Prioritization based upon process type



## Scheduling Algorithms

1. First-Come, First-Served Scheduling
2. Shortest-Job-First Scheduling
3. Round-Robin Scheduling
4. Priority Scheduling
5. Multilevel Queue Scheduling
6. Multilevel Feedback Queue Scheduling





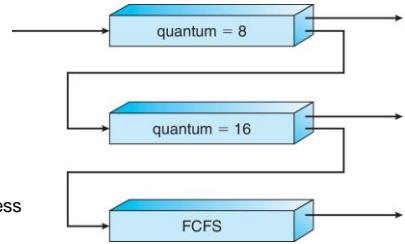
## Multilevel Feedback Queue

- A process can move between the various queues.
- 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
- Aging can be implemented using multilevel feedback queue



## 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 process enters queue  $Q_0$  which is served in RR
    - When it gains CPU, the process receives 8 milliseconds
    - If it does not finish in 8 milliseconds, the process is moved to queue  $Q_1$
  - At  $Q_1$  job is again served in RR and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue  $Q_2$



## Chapter 2: Process Management

### Advanced CPU Scheduling



Operating System Concepts – 10<sup>th</sup> Edition

Silberschatz, Galvin and Gagne ©2018



### Outline

- Thread Scheduling
- Multi-Processor Scheduling
- Real-Time CPU Scheduling
- Operating Systems Examples
- Algorithm Evaluation

Operating System Concepts – 10<sup>th</sup> Edition

5a.42

Silberschatz, Galvin and Gagne ©2018





## Objectives

- Describe various CPU scheduling algorithms
- Assess CPU scheduling algorithms based on scheduling criteria
- Explain the issues related to multiprocessor and multicore scheduling
- Describe various real-time scheduling algorithms
- Describe the scheduling algorithms used in the Windows, Linux, and Solaris operating systems
- Apply modeling and simulations to evaluate CPU scheduling algorithms



## Thread Scheduling

- Distinction between user-level and kernel-level threads
- When threads supported, threads scheduled, not processes
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
  - Known as **process-contention scope (PCS)** since scheduling competition is within the process
  - Typically done via priority set by programmer
- Kernel thread scheduled onto available CPU is **system-contention scope (SCS)** – competition among all threads in system





## Thread Scheduling

- Lập lịch cục bộ (Local Scheduling)
  - Bằng cách nào Thư viện luồng quyết định chọn luồng nào để đặt vào một CPU ảo khả dụng:
    - Thường chọn luồng có mức ưu tiên cao nhất
  - Sự cạnh tranh CPU diễn ra giữa các luồng của cùng một tiến trình.
  - Trong các HĐH sử dụng mô hình Many-to-one, Many-to-many.
- Lập lịch toàn cục (Global Scheduling)
  - Bằng cách nào kernel quyết định kernel thread nào để lập lịch CPU chạy tiếp.
  - Sự cạnh tranh CPU diễn ra giữa tất cả các luồng trong hệ thống.
  - Trong các HĐH sử dụng mô hình One-to-one (Windows XP, Linux, Solaris 9)



## Pthread Scheduling

- API allows specifying either PCS or SCS during thread creation
  - PTHREAD\_SCOPE\_PROCESS schedules threads using PCS scheduling
  - PTHREAD\_SCOPE\_SYSTEM schedules threads using SCS scheduling
- Can be limited by OS – Linux and macOS only allow PTHREAD\_SCOPE\_SYSTEM





## Pthread Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[]) {
    int i, scope;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* first inquire on the current scope */
    if (pthread_attr_getscope(&attr, &scope) != 0)
        fprintf(stderr, "Unable to get scheduling scope\n");
    else {
        if (scope == PTHREAD_SCOPE_PROCESS)
            printf("PTHREAD_SCOPE_PROCESS");
        else if (scope == PTHREAD_SCOPE_SYSTEM)
            printf("PTHREAD_SCOPE_SYSTEM");
        else
            fprintf(stderr, "Illegal scope value.\n");
    }
}
```

Operating System Concepts – 10<sup>th</sup> Edition

5a.47

Silberschatz, Galvin and Gagne ©2018



## Pthread Scheduling API

```
/* set the scheduling algorithm to PCS or SCS */
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
/* create the threads */
for (i = 0; i < NUM_THREADS; i++)
    pthread_create(&tid[i], &attr, runner, NULL);
/* now join on each thread */
for (i = 0; i < NUM_THREADS; i++)
    pthread_join(tid[i], NULL);
}
/* Each thread will begin control in this function */
void *runner(void *param)
{
    /* do some work ... */
    pthread_exit(0);
}
```

Operating System Concepts – 10<sup>th</sup> Edition

5a.48

Silberschatz, Galvin and Gagne ©2018





## Outline

- Thread Scheduling
- **Multi-Processor Scheduling**
- Real-Time CPU Scheduling
- Operating Systems Examples
- Algorithm Evaluation



## Multiple-Processor Scheduling

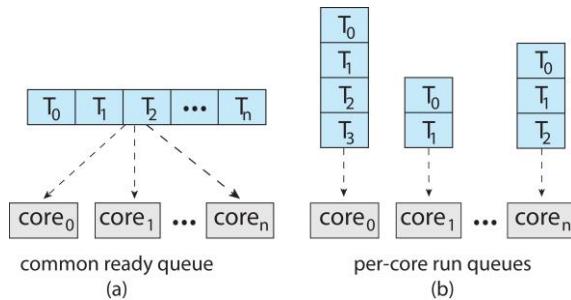
- CPU scheduling more complex when multiple CPUs are available
- Multiprocess may be any one of the following architectures:
  - Multicore CPUs
  - Multithreaded cores
  - NUMA systems
  - Heterogeneous multiprocessing





## Multiple-Processor Scheduling

- Symmetric multiprocessing (SMP) is where each processor is self scheduling.
- All threads may be in a common ready queue (a)
- Each processor may have its own private queue of threads (b)



Operating System Concepts – 10<sup>th</sup> Edition

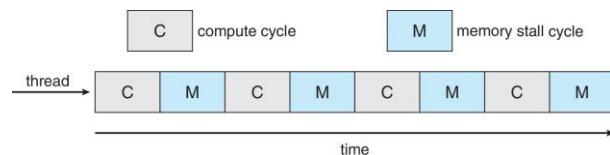
5a.51

Silberschatz, Galvin and Gagne ©2018



## Multicore Processors

- Recent trend to place multiple processor cores on same physical chip
- Faster and consumes less power
- Multiple threads per core also growing
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens
- Figure



Operating System Concepts – 10<sup>th</sup> Edition

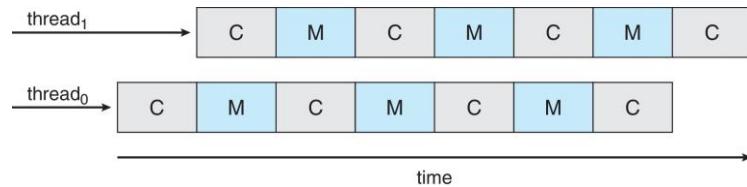
5a.52

Silberschatz, Galvin and Gagne ©2018



## Multithreaded Multicore System

- Each core has > 1 hardware threads.
- If one thread has a memory stall, switch to another thread!
- Figure



Operating System Concepts – 10<sup>th</sup> Edition

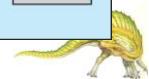
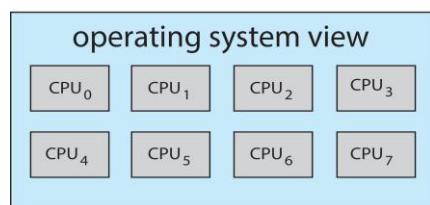
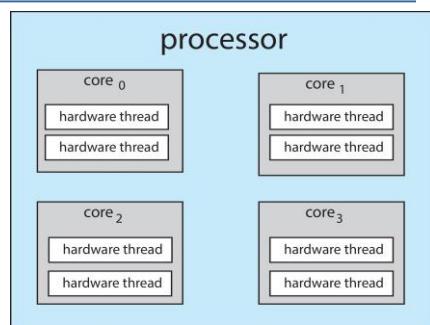
5a.53

Silberschatz, Galvin and Gagne ©2018



## Multithreaded Multicore System

- **Chip-multipathing (CMT)** assigns each core multiple hardware threads. (Intel refers to this as **hyperthreading**.)
- On a quad-core system with 2 hardware threads per core, the operating system sees 8 logical processors.



Operating System Concepts – 10<sup>th</sup> Edition

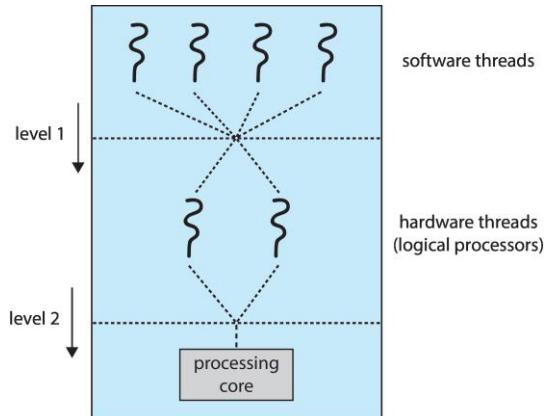
5a.54

Silberschatz, Galvin and Gagne ©2018



## Multithreaded Multicore System

- Two levels of scheduling:
  1. The operating system deciding which software thread to run on a logical CPU
  2. How each core decides which hardware thread to run on the physical core.



## Multiple-Processor Scheduling – Load Balancing

- If SMP, need to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep workload evenly distributed
- **Push migration** – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
- **Pull migration** – idle processors pulls waiting task from busy processor





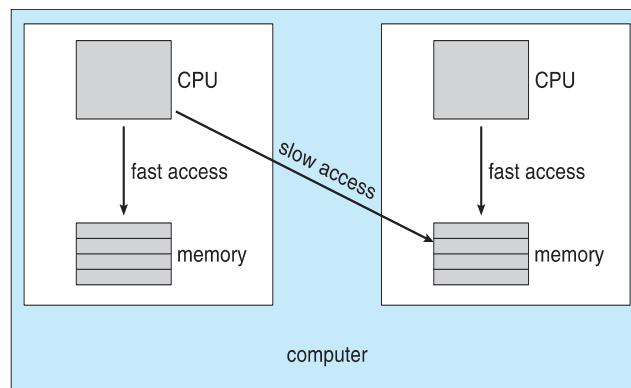
## Multiple-Processor Scheduling – Processor Affinity

- When a thread has been running on one processor, the cache contents of that processor stores the memory accesses by that thread.
- We refer to this as a thread having affinity for a processor (i.e., “processor affinity”)
- Load balancing may affect processor affinity as a thread may be moved from one processor to another to balance loads, yet that thread loses the contents of what it had in the cache of the processor it was moved off of.
- **Soft affinity** – the operating system attempts to keep a thread running on the same processor, but no guarantees.
- **Hard affinity** – allows a process to specify a set of processors it may run on.



## NUMA and CPU Scheduling

If the operating system is **NUMA-aware**, it will assign memory closer to the CPU the thread is running on.





## Outline

- Thread Scheduling
- Multi-Processor Scheduling
- Real-Time CPU Scheduling
- Operating Systems Examples
- Algorithm Evaluation



## Real-Time CPU Scheduling

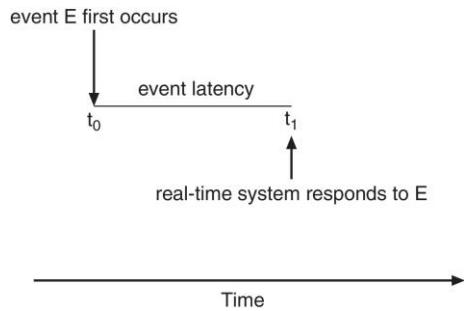
- Can present obvious challenges
- **Hard real-time systems –**
  - task must be serviced by its deadline
- **Soft real-time systems –**
  - Critical real-time tasks have the highest priority, but no guarantee as to when tasks will be scheduled



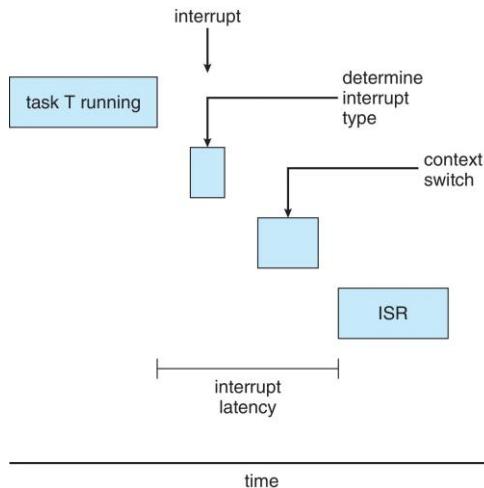


## Real-Time CPU Scheduling

- Event latency – the amount of time that elapses from when an event occurs to when it is serviced.
- Two types of latencies affect performance
  1. **Interrupt latency** – time from arrival of interrupt to start of routine that services interrupt
  2. **Dispatch latency** – time for scheduler to take current process off CPU and switch to another



## Interrupt Latency

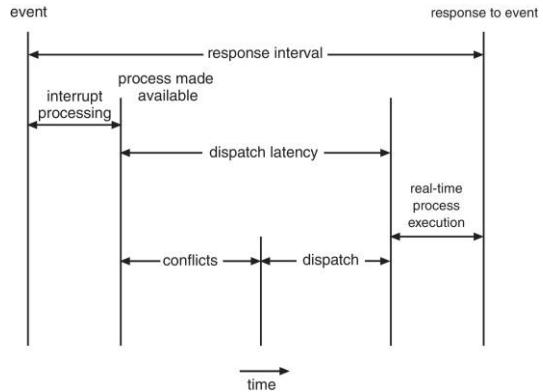




## Dispatch Latency

- Conflict phase of dispatch latency:

- Preemption of any process running in kernel mode
- Release by low-priority process of resources needed by high-priority processes



Operating System Concepts – 10<sup>th</sup> Edition

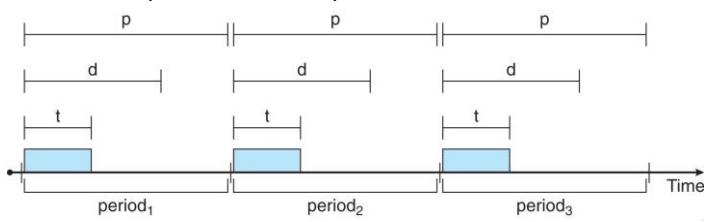
5a.63

Silberschatz, Galvin and Gagne ©2018



## Priority-based Scheduling

- For real-time scheduling, scheduler must support preemptive, priority-based scheduling
  - But only guarantees soft real-time
- For hard real-time must also provide ability to meet deadlines
- Processes have new characteristics: **periodic** ones require CPU at constant intervals
  - Has processing time  $t$ , deadline  $d$ , period  $p$
  - $0 \leq t \leq d \leq p$
  - Rate** of periodic task is  $1/p$



Operating System Concepts – 10<sup>th</sup> Edition

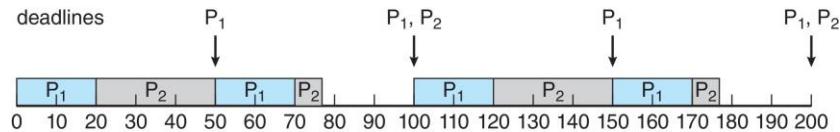
5a.64

Silberschatz, Galvin and Gagne ©2018



## Rate Monotonic Scheduling

- A priority is assigned based on the inverse of its period
- Shorter periods = higher priority;
- Longer periods = lower priority
- $P_1$  is assigned a higher priority than  $P_2$ .



Operating System Concepts – 10<sup>th</sup> Edition

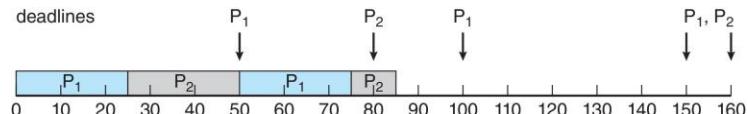
5a.65

Silberschatz, Galvin and Gagne ©2018



## Missed Deadlines with Rate Monotonic Scheduling

- Process  $P_2$  misses finishing its deadline at time 80
- Figure



Operating System Concepts – 10<sup>th</sup> Edition

5a.66

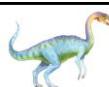
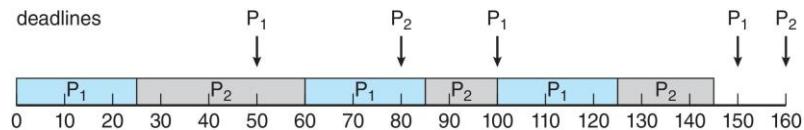
Silberschatz, Galvin and Gagne ©2018





## Earliest Deadline First Scheduling (EDF)

- Priorities are assigned according to deadlines:
  - The earlier the deadline, the higher the priority
  - The later the deadline, the lower the priority
- Figure



## Proportional Share Scheduling

- $T$  shares are allocated among all processes in the system
- An application receives  $N$  shares where  $N < T$
- This ensures each application will receive  $N / T$  of the total processor time





## POSIX Real-Time Scheduling

- The POSIX.1b standard
- API provides functions for managing real-time threads
- Defines two scheduling classes for real-time threads:
  1. SCHED\_FIFO - threads are scheduled using a FCFS strategy with a FIFO queue. There is no time-slicing for threads of equal priority
  2. SCHED\_RR - similar to SCHED\_FIFO except time-slicing occurs for threads of equal priority
- Defines two functions for getting and setting scheduling policy:
  1. `pthread_attr_getsched_policy(pthread_attr_t *attr, int *policy)`
  2. `pthread_attr_setsched_policy(pthread_attr_t *attr, int policy)`



## POSIX Real-Time Scheduling API

```
#include <pthread.h>
#include <stdio.h>

#define NUM_THREADS 5
int main(int argc, char *argv[])
{
    int i, policy;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* get the current scheduling policy */
    if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
        fprintf(stderr, "Unable to get policy.\n");
    else {
        if (policy == SCHED_OTHER) printf("SCHED_OTHER\n");
        else if (policy == SCHED_RR) printf("SCHED_RR\n");
        else if (policy == SCHED_FIFO) printf("SCHED_FIFO\n");
    }
}
```





## POSIX Real-Time Scheduling API (Cont.)

```
/* set the scheduling policy - FIFO, RR, or OTHER */
if (pthread_attr_setschedpolicy(&attr, SCHED_FIFO) != 0)
    fprintf(stderr, "Unable to set policy.\n");
/* create the threads */
for (i = 0; i < NUM_THREADS; i++)
    pthread_create(&tid[i], &attr, runner, NULL);
/* now join on each thread */
for (i = 0; i < NUM_THREADS; i++)
    pthread_join(tid[i], NULL);
}

/* Each thread will begin control in this function */
void *runner(void *param)
{
    /* do some work ... */
    pthread_exit(0);
}
```



## Outline

- Thread Scheduling
- Multi-Processor Scheduling
- Real-Time CPU Scheduling
- Operating Systems Examples
- Algorithm Evaluation





## Operating System Examples

- Linux scheduling
- Windows scheduling
- Solaris scheduling



## Linux Scheduling Through Version 2.5

- Prior to kernel version 2.5, ran variation of standard UNIX scheduling algorithm
- Version 2.5 moved to constant order  $O(1)$  scheduling time
  - Preemptive, priority based
  - Two priority ranges: time-sharing and real-time
  - **Real-time** range from 0 to 99 and **nice** value from 100 to 140
  - Map into global priority with numerically lower values indicating higher priority
  - Higher priority gets larger  $q$
  - Task run-able as long as time left in time slice (**active**)
  - If no time left (**expired**), not run-able until all other tasks use their slices
  - All run-able tasks tracked in per-CPU **runqueue** data structure
    - Two priority arrays (active, expired)
    - Tasks indexed by priority
    - When no more active, arrays are exchanged
  - Worked well, but poor response times for interactive processes





## Linux Scheduling in Version 2.6.23 +

- Completely Fair Scheduler (CFS)
- Scheduling classes
  - Each has specific priority
  - Scheduler picks highest priority task in highest scheduling class
  - Rather than quantum based on fixed time allotments, based on proportion of CPU time
  - Two scheduling classes included, others can be added
    1. default
    2. real-time



## Linux Scheduling in Version 2.6.23 + (Cont.)

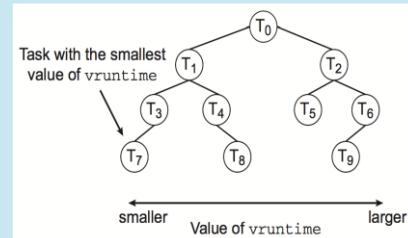
- Quantum calculated based on **nice value** from -20 to +19
  - Lower value is higher priority
  - Calculates **target latency** – interval of time during which task should run at least once
  - Target latency can increase if say number of active tasks increases
- CFS scheduler maintains per task **virtual 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





## CFS Performance

The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:

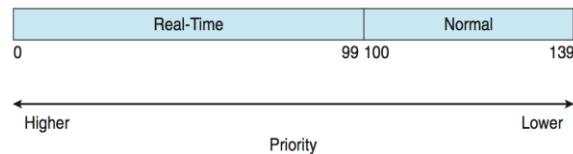


When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and tasks that have been given more processing time are on the right side. According to the properties of a binary search tree, the leftmost node has the smallest key value, which for the sake of the CFS scheduler means that it is the task with the highest priority. Because the red-black tree is balanced, navigating it to discover the leftmost node will require  $O(\lg N)$  operations (where  $N$  is the number of nodes in the tree). However, for efficiency reasons, the Linux scheduler caches this value in the variable `rb_leftmost`, and thus determining which task to run next requires only retrieving the cached value.



## Linux Scheduling (Cont.)

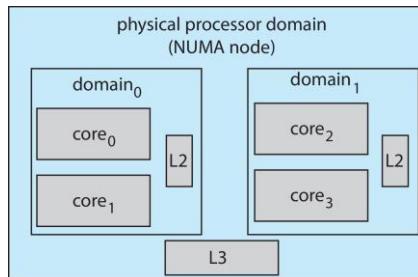
- Real-time scheduling according to POSIX.1b
  - Real-time tasks have static priorities
- Real-time plus normal map into global priority scheme
- Nice value of -20 maps to global priority 100
- Nice value of +19 maps to priority 139





## Linux Scheduling (Cont.)

- Linux supports load balancing, but is also NUMA-aware.
- **Scheduling domain** is a set of CPU cores that can be balanced against one another.
- Domains are organized by what they share (i.e., cache memory.) Goal is to keep threads from migrating between domains.



## Windows Scheduling

- Windows uses priority-based preemptive scheduling
- Highest-priority thread runs next
- **Dispatcher** is scheduler
- Thread runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
- Real-time threads can preempt non-real-time
- 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
- If no run-able thread, runs **idle thread**





## Windows Priority Classes

- Win32 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



## Windows Priority Classes (Cont.)

- If wait occurs, priority boosted depending on what was waited for
- Foreground window given 3x priority boost
- Windows 7 added **user-mode scheduling (UMS)**
  - Applications create and manage threads independent of kernel
  - For large number of threads, much more efficient
  - UMS schedulers come from programming language libraries like C++ **Concurrent Runtime** (ConcRT) framework





## Windows Priorities

	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



## Solaris

- Priority-based scheduling
- Six classes available
  - Time sharing (default) (TS)
  - Interactive (IA)
  - Real time (RT)
  - System (SYS)
  - Fair Share (FSS)
  - Fixed priority (FP)
- Given thread can be in one class at a time
- Each class has its own scheduling algorithm
- Time sharing is multi-level feedback queue
  - Loadable table configurable by sysadmin





## Solaris Dispatch Table

priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59



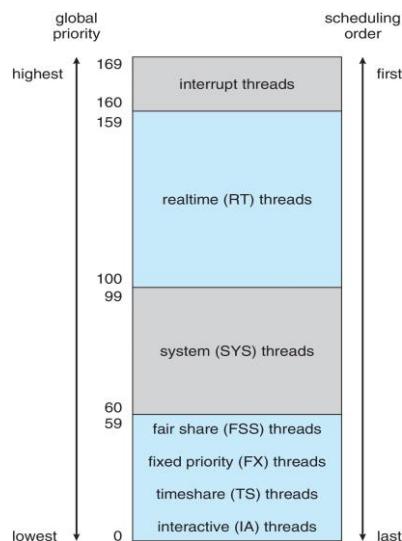
Operating System Concepts – 10<sup>th</sup> Edition

5a.85

Silberschatz, Galvin and Gagne ©2018



## Solaris Scheduling



Operating System Concepts – 10<sup>th</sup> Edition

5a.86

Silberschatz, Galvin and Gagne ©2018



## Solaris Scheduling (Cont.)

- Scheduler converts class-specific priorities into a per-thread global priority
  - Thread with highest priority runs next
  - Runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
  - Multiple threads at same priority selected via RR



## Outline

- Thread Scheduling
- Multi-Processor Scheduling
- Real-Time CPU Scheduling
- Operating Systems Examples
- Algorithm Evaluation





## Algorithm Evaluation

- How to select CPU-scheduling algorithm for an OS?
- Determine criteria, then evaluate algorithms
- **Deterministic modeling**
  - Type of **analytic evaluation**
  - Takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Consider 5 processes arriving at time 0:

Process	Burst Time
$P_1$	10
$P_2$	29
$P_3$	3
$P_4$	7
$P_5$	12



Operating System Concepts – 10<sup>th</sup> Edition

5a.89

Silberschatz, Galvin and Gagne ©2018



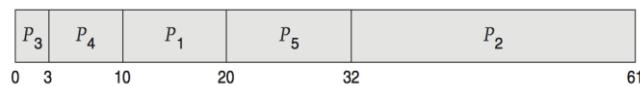
## Deterministic Evaluation

- For each algorithm, calculate minimum average waiting time
- Simple and fast, but requires exact numbers for input, applies only to those inputs
  - FCS is 28ms:

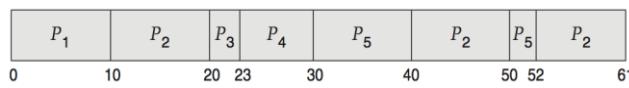
Process	Burst Time
$P_1$	10
$P_2$	29
$P_3$	3
$P_4$	7
$P_5$	12



- Non-preemptive SJF is 13ms:



- RR is 23ms:



Operating System Concepts – 10<sup>th</sup> Edition

5a.90

Silberschatz, Galvin and Gagne ©2018



## Queueing Models

- Describes the arrival of processes, and CPU and I/O bursts probabilistically
  - Commonly exponential, and described by mean
  - Computes average throughput, utilization, waiting time, etc.
- Computer system described as network of servers, each with queue of waiting processes
  - Knowing arrival rates and service rates
  - Computes utilization, average queue length, average wait time, etc.



## Little's Formula

- $n$  = average queue length
- $W$  = average waiting time in queue
- $\lambda$  = average arrival rate into queue
- Little's law – in steady state, processes leaving queue must equal processes arriving, thus:  
$$n = \lambda \times W$$
  - Valid for any scheduling algorithm and arrival distribution
- For example, if on average 7 processes arrive per second, and normally 14 processes in queue, then average wait time per process = 2 seconds



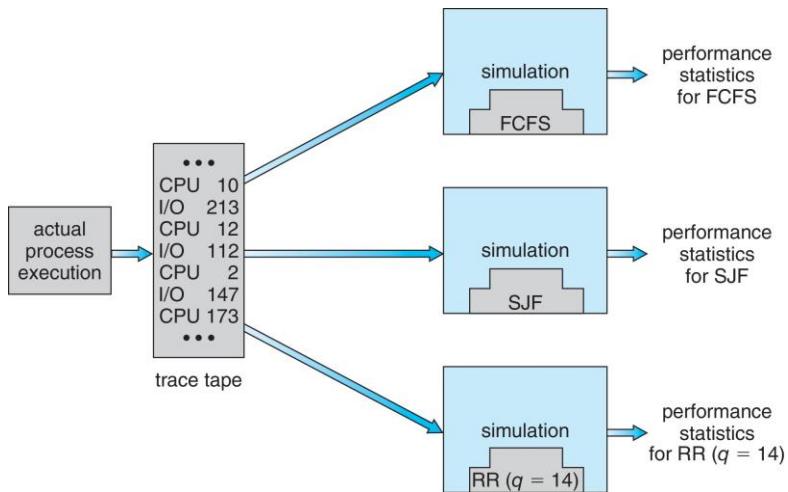


## Simulations

- Queueing models limited
- **Simulations** more accurate
  - Programmed model of computer system
  - Clock is a variable
  - Gather statistics indicating algorithm performance
  - Data to drive simulation gathered via
    - Random number generator according to probabilities
    - Distributions defined mathematically or empirically
    - Trace tapes record sequences of real events in real systems



## Evaluation of CPU Schedulers by Simulation



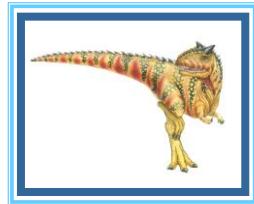


## Implementation

- Even simulations have limited accuracy
- Just implement new scheduler and test in real systems
  - High cost, high risk
  - Environments vary
- Most flexible schedulers can be modified per-site or per-system
- Or APIs to modify priorities
- But again environments vary



## End of Chapter 5



## Chapter 2: Process Synchronization

### 2.4 Process Synchronization



GV: Nguyễn Thị Thanh Vân

Operating System Concepts – 10<sup>th</sup> Edition

Silberschatz, Galvin and Gagne ©2018



## Outline

- Background
- The Critical-Section Problem
- Peterson's Solution
- Hardware Support for Synchronization
- Mutex Locks
- Semaphores
- Monitors
- Liveness
- Evaluation



Operating System Concepts – 10<sup>th</sup> Edition

6.2

Silberschatz, Galvin and Gagne ©2018



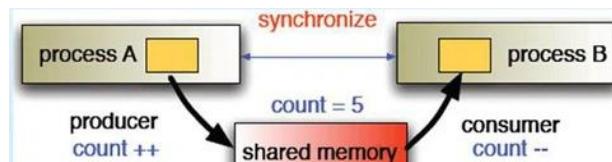
## Objectives

- Describe the critical-section problem and illustrate a race condition
- Illustrate hardware solutions to the critical-section problem using memory barriers, compare-and-swap operations, and atomic variables
- Demonstrate how mutex locks, semaphores, monitors, and condition variables can be used to solve the critical section problem
- Evaluate tools that solve the critical-section problem in low-, Moderate-, and high-contention scenarios



## Background

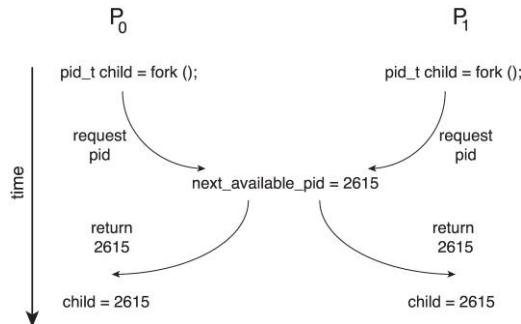
- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- The Bounded Buffer problem with use of a counter that is updated concurrently by the producer and consumer,. Which lead to race condition.





## Race Condition

- Processes  $P_0$  and  $P_1$  are creating child processes using the `fork()` system call
- Race condition on kernel variable `next_available_pid` which represents the next available process identifier (pid)



- Unless there is a mechanism to prevent  $P_0$  and  $P_1$  from accessing the variable `next_available_pid` the same pid could be assigned to two different processes!



## Critical Section Problem

- Process synchronization (no race condition) = critical-section problem (*phần quan trọng, phần tranh chấp, miền găng*)
- Consider system of  $n$  processes  $\{p_0, p_1, \dots p_{n-1}\}$
- Each process has **critical section (CS)** segment of code
  - Process may be changing common variables, updating table, writing file, etc.
  - When 1 process in CS , no other may be in its CS
- => CS is the code of the process that accesses shared resources
  - it needs to be run atomically (indivisible, can't be seen from the outside, can't stop suddenly,...)
  - Atomicity also means that 2 or more CS segments of many processes accessing the same resource will never run concurrently, but must be sequential.
- **Critical section problem** is to design protocol to solve this





## Critical Section

- General structure of process  $P_i$

```
do {
```

```
entry section
```

```
critical section
```

```
exit section
```

```
remainder section
```

```
} while (true);
```

- Two general approaches are used to handle critical sections in OS:

- preemptive kernel: allows a process to be preempted while it is running in kernel mode.
- nonpreemptive kernel: does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU.



## Critical-Section Problem (Cont.)

Requirements for solution to critical-section problem

1. **Mutual Exclusion** - If process  $P_i$  is executing in its CS, then no other processes can be executing in their critical sections

(Không có hai tiến trình cùng ở trong miền critical-section cùng lúc)

2. **Progress** - If no process is executing in its CS and there exist some processes that wish to enter their CS, then the selection of the process that will enter the critical section next cannot be postponed indefinitely

(Một tiến trình tạm dừng bên ngoài miền critical-section không được ngăn cản các tiến trình khác vào miền critical-section)

3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their CSs after a process has made a request to enter its CS and before that request is granted

(Không có tiến trình nào phải chờ vô hạn để được vào miền critical-section)

- Assume that each process executes at a nonzero speed
- No assumption concerning **relative speed** of the  $n$  processes





## Classification of solutions

- Interrupt-based Solution
- Software Solution
  - Software Solution 1
  - Software Solution 2
  - Peterson's Algorithm
- Hardware support for synchronization
  - Memory barriers
  - Special hardware instructions
    - ▶ **Test-and-Set** instruction
    - ▶ **Compare-and-Swap** instruction
- Mutex lock
- Semaphores
- Monitors
- Liveness



## Interrupt-based Solution

- The critical-section problem could be solved simply in a single-core environment if we could prevent interrupts from occurring while a shared variable was being modified.
- Entry section: disable interrupts
- Exit section: enable interrupts
- Will this solve the problem?
  - What if the critical section is code that runs for an hour?  
Can some processes starve – never enter their critical section.
  - What if there are two CPUs?  
The interrupt prohibition will only work on the processor that is handling the process, while processes executing on other processors can still access the CS!





## Software Solution 1

- Two process solution
- Assume that the **load** and **store** machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share one variable:
  - **int turn;**
- The variable **turn** indicates whose turn it is to enter the critical section
- Initially **turn = 0**
- Algorithm for Process  $P_i$

```
do {  
    while (turn != i);  
    critical section  
    turn = j;  
    remainder section  
} while (1);
```

- Meet 3 requirements of CS?



## Correctness of the Software Solution1

- (1) Mutual exclusion is preserved – **is satisfied**  
 $P_i$  enters critical section if and only if:  
**turn = i**  
and **turn** cannot be both 0 and 1 at the same time
- (2) What about the Progress requirement? – **is not satisfied**
  - If Process 0 wants to enter the CS and Process 1 is not interested in entering the CS, can Process 0 enter?

```
Process P0:  
do  
    while (turn != 0);  
    critical section  
    turn := 1;  
    remainder section  
while (1);
```

```
Process P1:  
do  
    while (turn != 1);  
    critical section  
    turn := 0;  
    remainder section  
while (1);
```

- (3) What about the Bounded-waiting requirement? – **is not satisfied**





## Software Solution -- Peterson's Algorithm

- Peterson's solution:
  - software-based solution to the critical-section problem
  - Involves designing software that addresses the requirements of CS:
    - mutual exclusion,
    - progress,
    - and bounded waiting
  - is not guaranteed to work on modern computer architecture
- Two process solution:
  - alternate execution between their critical sections and remainder sections,
  - And share two variables:
    - `int turn;`
    - `boolean flag[2]`



## Algorithm for Process $P_i$

- The variable `turn` indicates whose turn it is to enter the critical section
- The `flag` array is used to indicate if a process is ready to enter the critical section.
  - `flag[i] = true` implies that process  $P_i$  is ready!

```
while (true){  
  
    flag[i] = true; /*Pi ready */  
    turn = j;      /*preemptive Pj */  
    while (flag[j] && turn == j)  
        ;  
  
    /* critical section */  
  
    flag[i] = false;  
  
    /* remainder section */  
}
```





## Peterson's Algorithm

```
while (true){      /* 0 wants in */
    flag[0] = true;
    turn = 1;      /* 0 gives a change to 1 */
    while (flag[1] && turn == 1)
        ;
    /* critical section */
    /* 0 no longer wants in */
    flag[0] = false;
    /* remainder section */
}

while (true){      /* 1 wants in */
    flag[1] = true;
    turn = 0;      /* 1 gives a change to 0 */
    while (flag[0] && turn == 0)
        ;
    /* critical section */
    /* 1 no longer wants in */
    flag[1] = false;
    /* remainder section */
}
```

Operating System Concepts - 10<sup>th</sup> Edition

6.17

Silberschatz, Galvin and Gagne ©2018



## Correctness of Peterson's Solution

- Three CS requirement are met:

1. Mutual exclusion is preserved

$P_i$  enters CS only if: either `flag[j] = false` or `turn = i`

2. Progress requirement is satisfied

3. Bounded-waiting requirement is met

- Proving:

1.  $P_i$  and  $P_j$  are same time if: `flag[i] = flag[j] == true` and `turn == i`.  $\Rightarrow$  cannot occur

2,3. Note that a process can be prevented from entering the CS only if it is stuck in the while loop with the condition `flag[i] = true and turn = j`, this loop is the only one possible.

- If  $P_j$  is not ready to CS, then `flag[j] == false`, and  $P_i$  can enter its CS.
- If  $P_j$  has set `flag[j] = true` and is also executing in its while statement, then either `turn == i` or `j`.
  - If `turn == i`, then  $P_i$  will enter the CS.
  - If `turn == j`, then  $P_j$  will enter the CS. ( $P_i$  continue)
- However, once  $P_j$  exits its CS, it will reset `flag[j] = false`, allowing  $P_i$  to enter its CS.
- If  $P_j$  resets `flag[j]` to true, it must also set `turn to i`.
- Thus, since  $P_i$  does not change the value of the variable `turn` while executing the while statement,  $P_i$  will enter the CS (**progress**) after at most one entry by  $P_j$  (**bounded waiting**)



Operating System Concepts - 10<sup>th</sup> Edition

6.18

Silberschatz, Galvin and Gagne ©2018



## Peterson's Solution

- Solution works for 2 process.
- What about modifying it to handle 10 processes?
- Solution requires **busy waiting**
  - Processes waste CPU cycles to ask if they can enter the critical section
- Although useful for demonstrating an algorithm, Peterson's Solution is not guaranteed to work on modern architectures.
  - To improve performance, processors and/or compilers may reorder operations that have no dependencies
- Understanding why it will not work is useful for better understanding race conditions.
- For single-threaded this is OK as the result will always be the same.
- For multithreaded the reordering may produce inconsistent or unexpected results!



## Modern Architecture Example

- Two threads share the data:

```
boolean flag = false;
int x = 0;
```
- Thread 1 performs

```
while (!flag)
;
print x
```
- Thread 2 performs

```
x = 100;
flag = true
```
- What is the expected output (Thread1)? 100
- However, since the variables **flag** and **x** are independent of each other, the instructions: (reorder the instructions for Thread 2)

```
flag = true;
x = 100;
```
- If this occurs, the output may be (Thread1): 0



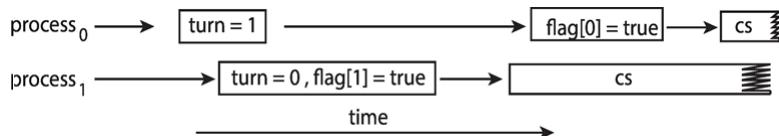


## Peterson's Solution Revisited

- The effects of instruction reordering in Peterson's Solution: reordered 2 lines

```
while (true) {
    flag[i] = true; /*Pi ready */
    turn = j;        /*preemptive Pj */
    while (flag[j] && turn == j)
        ;
    /* critical section */
    flag[i] = false;
    /* remainder section */
}
```

- Result:



- This allows both processes to be in their critical section at the same time!
- To ensure that Peterson's solution will work correctly on modern computer architecture we must use **Memory Barrier**



## Hardware Support for Synchronization

- Many systems provide hardware support for implementing the critical section code.
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
    - Is this practical?
  - Generally, too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- We will look at **three** forms of hardware support:
  - Memory Barriers
  - Hardware instructions:
    - Test-and-Set** instruction
    - Compare-and-Swap** instruction
  - Atomic Variables





## Memory Barrier Instructions

- A **memory barrier** instruction:
  - is used to ensure that all **loads** and **stores** instructions are completed before any subsequent load or store operations are performed.
- Therefore, even if instructions were reordered, the memory barrier ensures that
  - the store operations are completed in memory and visible to other processors before future load or store operations are performed.
- Note that memory barriers are considered very low-level operations and are typically only used by kernel developers when writing specialized code that ensures mutual exclusion



## Memory Barrier Example

- We could add a memory barrier to the following instructions to ensure Thread 1 outputs 100:
- Thread 1 now performs

```
while (!flag)
    memory_barrier();
    print x
```
- Thread 2 now performs

```
x = 100;
memory_barrier();
flag = true
```
- For Thread 1 we are guaranteed that the value of **flag** is loaded before the value of **x**.
- For Thread 2 we ensure that the assignment to **x** occurs before the assignment **flag**.





## Hardware Instructions

- Special hardware instructions that allow us to either *test-and-modify* the content of a word, or to *swap* the contents of two words atomically (uninterruptedly.)
- Use these special instructions to solve the critical-section problem in a relatively simple manner.
- types of instructions:
  - **Test-and-Set** instruction
  - **Compare-and-Swap** instruction



## The test\_and\_set Instruction

- Definition

```
boolean test_and_set (boolean *target)
{
    boolean rv = *target;
    *target = true;
    return rv;
}
```

- Properties

- Executed atomically
- Returns the original value of passed parameter
- Set the new value of passed parameter to **true**





## Solution Using test\_and\_set()

- Shared Boolean variable **lock**, initialized to **false**
- Solution: using non-preemptive (ko đọc quyền) access with TSL can be implemented using lock variable

```
do {  
    while (test_and_set(&lock))  
        ; /* do nothing */  
  
    /* critical section */  
    lock = false;  
    /* remainder section */  
} while (true);
```

- Based on busy waiting
- Does it solve the critical-section problem?
  - Mutual exclusion - ok
  - bounded wait - ok



## The compare\_and\_swap Instruction

- just like the test and set(), operates on two words atomically, but uses a different mechanism that is based on **swapping** the content of two words
- . Definition

```
int compare_and_swap(int *value, int expected, int new_value)  
{  
    int temp = *value;  
    if (*value == expected)  
        *value = new_value;  
    return temp;  
}
```

- Properties:
  - Executed atomically => 2 CAS run: in some arbitrary order
  - If instruction: the swap takes place only under this condition.
  - Returns the original value of passed parameter **value**





## Solution using compare\_and\_swap

- Shared integer **lock** initialized to 0;
- Solution:

```
while (true){  
    while (compare_and_swap(&lock, 0, 1) != 0)  
        ; /* do nothing */  
  
    /* critical section */  
  
    lock = 0;  
  
    /* remainder section */  
}
```

- Based on busy waiting
- Does it solve the critical-section problem?
  - Mutual exclusion - ok
  - does not satisfy the bounded-waiting



## Bounded-waiting with compare-and-swap

```
while (true) {  
    waiting[i] = true;  
    key = 1;  
    while (waiting[i] && key == 1)  
        key = compare_and_swap(&lock, 0, 1);  
    waiting[i] = false;  
    /* critical section */  
    j = (i + 1) % n;  
    while ((j != i) && !waiting[j])  
        j = (j + 1) % n;  
    if (j == i)  
        lock = 0;  
    else  
        waiting[j] = false;  
    /* remainder section */  
}
```





## Atomic Variables

- Typically, instructions such as compare-and-swap are used as building blocks for other synchronization tools (not used to provide mutual exclusion)
- One tool is an **atomic variable** that provides *atomic* (uninterruptible) updates on basic data types such as integers and Booleans.
- For example:
  - Let **sequence** be an atomic variable
  - Let **increment()** be operation on the atomic variable **sequence**
  - The Command: **increment(&sequence);**

ensures **sequence** is incremented without interruption:

```
void increment(atomic_int *v)
{
    int temp;
    do {
        temp = *v;
    }
    while (temp !=
        (compare_and_swap(v,temp,temp+1));
    }
```

Operating System



Gagne ©2018



## Classification of solutions

- Interrupt-based Solution
- Software Solution
  - Software Solution 1
  - Software Solution 2
  - Peterson's Algorithm
- Hardware support for synchronization
  - Memory barriers
  - Special hardware instructions
    - ▶ **Test-and-Set** instruction
    - ▶ **Compare-and-Swap** instruction
- **Mutex lock**
- Semaphores
- Monitors
- Liveness



Operating System Concepts – 10<sup>th</sup> Edition

6.32

Silberschatz, Galvin and Gagne ©2018



## Mutex Locks

- Previous hardware-based solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve CS problem
  - Simplest of these tools is mutex lock
  - a process must acquire the lock before entering a CS; it releases the lock when it exits the CS.
- A mutex lock:
  - has boolean **available**: the lock is available or not
  - Two operations: **acquire()** a lock; **release()** a lock
  - The **acquire()** and **release()** are executed atomically
  - Usually implemented via hardware atomic instructions such as compare-and-swap.
- But this solution requires **busy waiting**. => called a **spinlock**
  - no context switch is required when a process must wait on a lock
  - On multicore, spinlocks are widely used



## Solution to CS Problem Using Mutex Locks

```
while (true) {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
}
```

```
acquire() {  
    while (!available)  
        ; /* busy wait */  
    available = false;  
}
```

```
release() {  
    available = true;  
}
```





## Semaphore

- A robust tool that can behave similarly to a mutex lock but can also provide more sophisticated ways for processes to synchronize their activities.
- Semaphores were introduced by the Dutch computer scientist Edsger Dijkstra
- Semaphore **S** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **wait()** and **signal()**
    - ▶ Originally called **P()** and **V()**
- Definition of the two operation

```
wait(S) {  
    while (S <= 0)  
        ; // busy wait  
    S--;  
}
```

```
signal(S) {  
    S++;  
}
```

- The semaphore helps the process avoid busy waiting:
  - when waiting, the process will be placed in a blocked queue, which contains processes that are waiting for the same event.
- But busy waiting may be needed to implement the semaphore itself



## Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
  - can be used to control access to a given resource consisting of a finite number of instances
    - ▶ a Process uses resource: performs a wait() operation (S--)
    - ▶ a process releases a resource: performs a signal() operation (S++)
    - ▶ the count for the semaphore goes to 0, all resources are being used.
    - ▶ processes that wish to use a resource will block until the count becomes greater than 0
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can implement a counting semaphore **S** using binary semaphores
- With semaphores we can solve various synchronization problems





## Semaphore Usage Example (Cont.)

- Solution to the CS Problem.
  - Create a semaphore “`mutex`” initialized to 1
  - Code:

```
wait(mutex);
CS
signal(mutex);
```
- Consider two concurrently running processes:  $P_1$  and  $P_2$  that with two statements  $S_1$  and  $S_2$  and the requirement that  $S_1$  to happen before  $S_2$ 
  - Create a semaphore “`synch`” initialized to 0
  - Code: to synchronization

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```



## Semaphore Implementation

- Must guarantee that no two processes can execute the `wait()` and `signal()` on the same semaphore at the same time
- Can be implemented using any of the critical sections solutions - where the `wait` and `signal` code are placed in the critical section
- Could now have **busy waiting** in critical section implementation
  - But implementation code is short
  - Little busy waiting if CS rarely occupied
- Note that for “regular” applications, where the application may spend lots of time in critical sections this is not a good solution
- To overcome **busy waiting**, we can modify the definition of the `wait()` and `signal()` operations



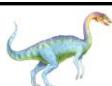


## Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue.
- Each entry in a waiting queue has two data items:
  - Value (of type integer)
  - Pointer to next record in the list
- Waiting queue (FIFO queue)

```
typedef struct {  
    int value;  
    struct process *list;  
} semaphore;
```

- OS provides 2 tasks:
  - The **wakeup (P)** operation resumes the execution of process P
  - The **sleep()** suspends the process that invoked it



## Implementation of the wait/ signal operation

- The **wait** operation:

```
wait(semaphore *S) {  
    S->value--;  
  
    if (S->value < 0) {  
        add this P to S->list;  
        sleep();  
    }  
}
```

- The **sleep()** suspends the process that invoked it.

- The **signal** operation:

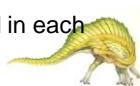
```
signal(semaphore *S) {  
    S->value++;  
  
    if (S->value <= 0) {  
        remove a P from S->list;  
        wakeup (P);  
    }  
}
```

- The **wakeup (P)** operation resumes the execution of process P

- S.value<0: The number of process in queue: |S.value|

- S.value >=0: The number of process execute wait(S), without blocked: S.value

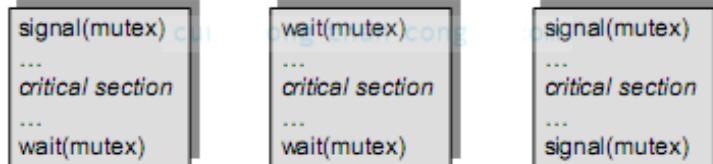
- The list of waiting processes can be easily implemented by a link field in each process control block (PCB)





## Problems with Semaphores

- Incorrect use of semaphore operations:
- Ex: interchanges the order in which the wait() and signal() operations on the semaphore mutex are executed



- Omitting of **wait(mutex)** and/or **signal(mutex)**
- Result: either mutual exclusion is violated or the process will permanently block.
- These – and others – are examples of what can occur when semaphores and other synchronization tools are used incorrectly.
- Solution: introduce high-level programming constructs - **Monitor**



## Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- Pseudocode syntax of a monitor:

```
monitor monitor-name
{
    // shared variable declarations
    function P1 (...) { ... }

    function P2 (...) { ... }

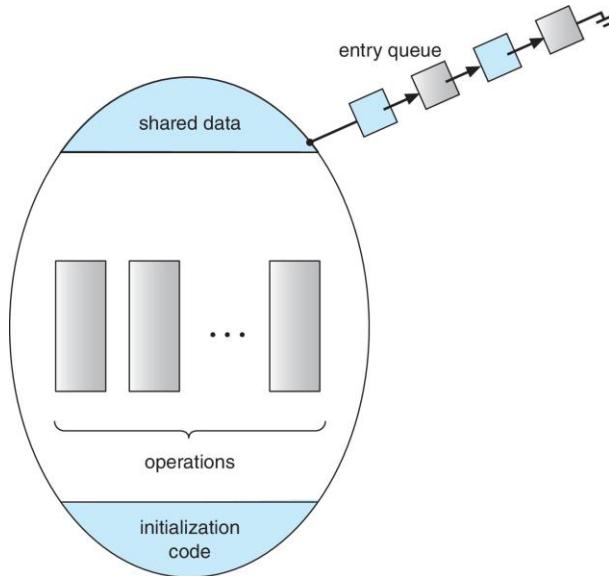
    function Pn (...) { ..... }

    initialization code (...) { ... }
}
```





## Schematic view of a Monitor



Operating System Concepts – 10<sup>th</sup> Edition

6.43

Silberschatz, Galvin and Gagne ©2018



## Condition Variables

- **condition x, y;**
- Two operations are allowed on a condition variable:
  - **x.wait()** – a process that invokes the operation is suspended until **x.signal()**
  - **x.signal()** – resumes one of processes (if any) that invoked **x.wait()**
    - ▶ If no **x.wait()** on the variable, then it has no effect on the variable

Operating System Concepts – 10<sup>th</sup> Edition

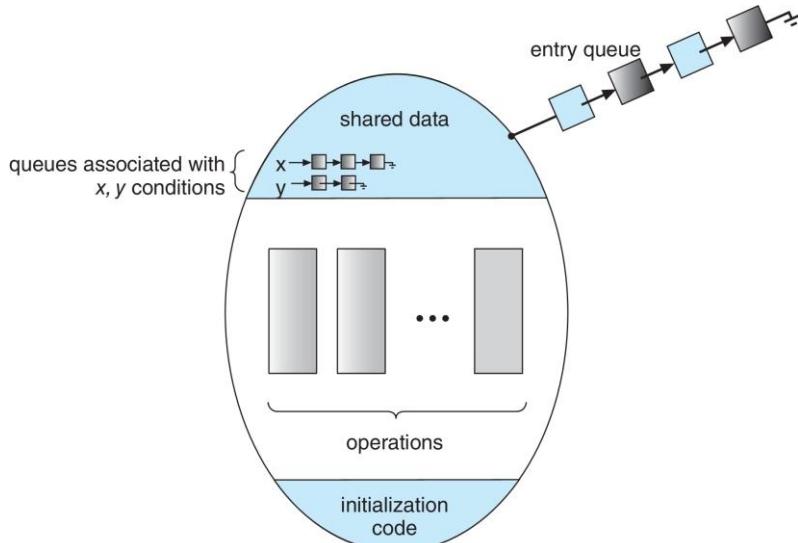
6.44

Silberschatz, Galvin and Gagne ©2018



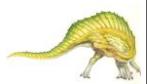


## Monitor with Condition Variables



## Condition Variables Choices

- If process P invokes **x.signal()**, and process Q is suspended in **x.wait()**, what should happen next?
  - Both Q and P cannot execute in parallel. If Q is resumed, then P must wait
- Options include
  - **Signal and wait** – P waits until Q either leaves the monitor or it waits for another condition
  - **Signal and continue** – Q waits until P either leaves the monitor or it waits for another condition
  - Both have pros and cons – language implementer can decide
  - Monitors implemented in Concurrent Pascal compromise
    - P executing signal immediately leaves the monitor, Q is resumed
  - Implemented in other languages including Mesa, C#, Java





## Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex; // (initially = 1)
semaphore next; // (initially = 0)
int next_count = 0; // number of processes waiting
                     inside the monitor
```

- Each function *F* will be replaced by

```
wait(mutex);
...
body of F;
...
if (next_count > 0)
    signal(next)
else
    signal(mutex);
```

- Mutual exclusion within a monitor is ensured



## Implementation – Condition Variables

- For each condition variable *x*, we have:

```
semaphore x_sem; // (initially = 0)
int x_count = 0;
```

- The operation *x.wait()* can be implemented as:

```
x_count++;
if (next_count > 0)
    signal(next);
else
    signal(mutex);
wait(x_sem);
x_count--;
```

- The operation *x.signal()* can be implemented as:

```
if (x_count > 0) {
    next_count++;
    signal(x_sem);
    wait(next);
    next_count--;
}
```





## Resuming Processes within a Monitor

- If several processes are queued on condition variable **x**, and **x.signal()** is executed, which process should be resumed?
- One simple solution is to use a FCFS ordering,
  - the process that has been waiting the longest is resumed first.
  - => simple scheduling scheme is not adequate.
  - In these circumstances, the conditional wait construct can be used
- **conditional-wait** construct of the form **x.wait(c)**
  - Where **c** is **priority number**
  - Process with lowest number (highest priority) is scheduled next



## Single Resource allocation Example

- Allocate a single resource among competing processes using priority numbers that specify the maximum amount of time a process plans to use the resource

```
R.acquire(t);  
...  
access the resource;  
...  
  
R.release;
```

- Where R is an instance of type **ResourceAllocator** (shown in the next slides)





## A Monitor to Allocate Single Resource

```
monitor ResourceAllocator
{
    boolean busy;
    condition x;
    void acquire(int time) {
        if (busy)
            x.wait(time);
        busy = true;
    }
    void release() {
        busy = FALSE;
        x.signal();
    }
    initialization code() {
        busy = false;
    }
}
```

- Usage:

- acquire

- ...

- release

- Incorrect use of monitor operations

- release() ... acquire()
  - acquire() ... acquire()
  - Omitting of acquire() and/or release()



## Liveness

- Processes may have to wait indefinitely while trying to acquire a synchronization tool such as a mutex lock or semaphore.
- Waiting indefinitely violates the progress and bounded-waiting criteria discussed at the beginning of this chapter.
- **Liveness** refers to a set of properties that a system must satisfy to ensure processes make progress.
- Indefinite waiting is an example of a liveness failure.





## Deadlock

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let  $s$  and  $q$  be two semaphores initialized to 1

```
P0
    wait(S);
    wait(Q);
    ...
    signal(S);
    signal(Q);

P1
    wait(Q);
    wait(S);
    ...
    signal(Q);
    signal(S);
```

- Consider if  $P_0$  executes `wait(S)` and  $P_1$  `wait(Q)`. When  $P_0$  executes `wait(Q)` – it is blocked, it must wait until  $P_1$  executes `signal(Q)`
- However,  $P_1$  is waiting – `wait(S)` is blocked, until  $P_0$  execute `signal(S)`.
- Since these `signal()` operations will never be executed,  $P_0$  and  $P_1$  are **deadlocked**.



## Other Forms of Deadlock

- **Starvation** – indefinite blocking
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
- Solved via **priority-inheritance protocol**



## Chapter 2: Process Synchronization

### Synchronization Examples



GV: Nguyễn Thị Thanh Vân

Operating System Concepts – 10<sup>th</sup> Edition

Silberschatz, Galvin and Gagne ©2018



### Outline

- Explain the bounded-buffer synchronization problem
- Explain the readers-writers synchronization problem
- Explain and dining-philosophers synchronization problems

Operating System Concepts – 10<sup>th</sup> Edition

6.56

Silberschatz, Galvin and Gagne ©2018





## Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem (Producer – Consumer)
  - Readers and Writers Problem
  - Dining-Philosophers Problem



## Bounded-Buffer Problem

- **n** buffers, each can hold one item
- Semaphore **mutex** initialized to the value 1
- Semaphore **full** initialized to the value 0
- Semaphore **empty** initialized to the value n

```
BufferSize = n; // số chỗ trong bộ đệm
semaphore mutex = 1; // kiểm soát truy xuất độc quyền
semaphore empty = BufferSize; // số chỗ trống
semaphore full = 0; // số chỗ đầy
```





## Bounded Buffer Problem (Cont.)

- The structure of the **producer** process

```
while (true) {
    ...
    /* produce an item in next_produced */
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add next produced to the buffer */
    ...
    signal(mutex);
    signal(full);
}
```



## Bounded Buffer Problem (Cont.)

- The structure of the **consumer** process

```
while (true) {
    wait(full);
    wait(mutex);
    ...
    /* remove an item from buffer to next_consumed */
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume the item in next consumed */
    ...
}
```





## Use Monitor

```
monitor ProducerConsumer
condition full, empty;
int count;
procedure enter();
{ if (count == N)
wait(full); // nếu bộ đệm đầy, phải chờ
enter_item(item); // đặt dữ liệu vào bộ đệm
count++; // tăng số chỗ đầy
if (count == 1) // nếu bộ đệm không trống
signal(empty); // thì kích hoạt Consumer
}

procedure remove();
{ if (count == 0)
wait(empty) // nếu bộ đệm trống, chờ
remove_item(&item); // lấy dữ liệu từ bộ đệm
count--; // giảm số chỗ đầy
if (count == N-1) // nếu bộ đệm không đầy
signal(full); // thì kích hoạt Producer
}
count = 0;
end monitor;
```

Operating System Concepts – 10<sup>th</sup> Edition



## Use Monitor

```
Producer();
{
while (TRUE)
{
produce_item(&item);
ProducerConsumer.enter;
}
}
```

```
Consumer();
{
while (TRUE)
{
ProducerConsumer.remove;
consume_item(item);
}
}
```



Operating System Concepts – 10<sup>th</sup> Edition

6.62

Silberschatz, Galvin and Gagne ©2018



## Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - **Readers** – only read the data set; they do **not** perform any updates
  - **Writers** – can both read and write
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are considered – all involve some form of priorities



## Readers-Writers: Shared Data

- Data set
- Semaphore **rw\_mutex** initialized to 1
- Semaphore **mutex** initialized to 1
- Integer **read\_count** initialized to 0





## The Structure of a Writer Process

```
while (true) {  
    wait(rw_mutex);  
    ...  
    /* writing is performed */  
    ...  
    signal(rw_mutex);  
}
```



## The Structure of a Reader process

```
while (true){  
    wait(mutex);  
    read_count++;  
    if (read_count == 1) /* first reader */  
        wait(rw_mutex);  
        signal(mutex);  
    ...  
    /* reading is performed */  
    ...  
    wait(mutex);  
    read_count--;  
    if (read_count == 0) /* last reader */  
        signal(rw_mutex);  
        signal(mutex);  
}
```





## Readers-Writers Problem Variations

- The solution in previous slide can result in a situation where a writer process never writes. It is referred to as the “First reader-writer” problem.
- The “Second reader-writer” problem is a variation the first reader-writer problem that state:
  - Once a writer is ready to write, no “newly arrived reader” is allowed to read.
- Both the first and second may result in starvation, leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks



## Dining-Philosophers Problem

- N philosophers sit at a round table with a bowel of rice in the middle.
- They spend their lives alternating between thinking and eating.
- They do not interact with their neighbors.
- Occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both chopsticks to eat, then release both when done
- In the case of 5 philosophers, the shared data
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1





## Semaphore Solution to Dining-Philosophers

- The structure of Philosopher *i*:

```
while (true){  
    wait (chopstick[i] );  
    wait (chopStick[ (i + 1) % 5] );  
  
    /* eat for awhile */  
  
    signal (chopstick[i] );  
    signal (chopstick[ (i + 1) % 5] );  
  
    /* think for awhile */  
}
```
- What is the problem with this algorithm?



## Monitor Solution to Dining Philosophers

```
monitor DiningPhilosophers  
{  
    enum { THINKING; HUNGRY, EATING} state [5] ;  
    condition self [5];  
  
    void pickup (int i) {  
        state[i] = HUNGRY;  
        test(i);  
        if (state[i] != EATING) self[i].wait;  
    }  
  
    void putdown (int i) {  
        state[i] = THINKING;  
        // test left and right neighbors  
        test((i + 4) % 5);  
        test((i + 1) % 5);  
    }  
}
```





## Solution to Dining Philosophers (Cont.)

```
void test (int i) {
    if ((state[(i + 4) % 5] != EATING) &&
        (state[i] == HUNGRY) &&
        (state[(i + 1) % 5] != EATING) ) {
        state[i] = EATING ;
        self[i].signal () ;
    }
}

initialization_code() {
    for (int i = 0; i < 5; i++)
        state[i] = THINKING;
}
}
```



## Solution to Dining Philosophers (Cont.)

- Each philosopher “i” invokes the operations `pickup()` and `putdown()` in the following sequence:

```
DiningPhilosophers.pickup(i);

/** EAT **/

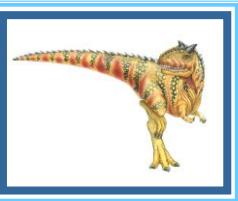
DiningPhilosophers.putdown(i);
```

- No deadlock, but starvation is possible



# End of Chapter 2

## 2.4



## Chapter 2: Process Synchronization

### 2.5: Deadlocks



GV: Nguyễn Thị Thanh Vân

Operating System Concepts – 10<sup>th</sup> Edition

Silberschatz, Galvin and Gagne ©2018



### Outline

- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

Operating System Concepts – 10<sup>th</sup> Edition

8.2

Silberschatz, Galvin and Gagne ©2018





## Chapter Objectives

- Illustrate how deadlock can occur when mutex locks are used
- Define the four necessary conditions that characterize deadlock
- Identify a deadlock situation in a resource allocation graph
- Evaluate the four different approaches for preventing deadlocks
- Apply the banker's algorithm for deadlock avoidance
- Apply the deadlock detection algorithm
- Evaluate approaches for recovering from deadlock



## System Model

- System consists of resources
- Resource types  $R_1, R_2, \dots, R_m$ 
  - CPU cycles, memory space, I/O devices
- Each resource type  $R_i$  has  $W_i$  instances.
- Each process utilizes a resource as follows:
  - **Request:** The thread requests the resource
  - **Use:** The thread can operate on the resource
  - **Release:** The thread releases the resource.





## Deadlock Example with Semaphores

- Data:
  - A semaphore **s1** initialized to 1
  - A semaphore **s2** initialized to 1
- Two processes P1 and P2
- P1:

```
wait(s1)  
wait(s2)
```
- P2:

```
wait(s2)  
wait(s1)
```

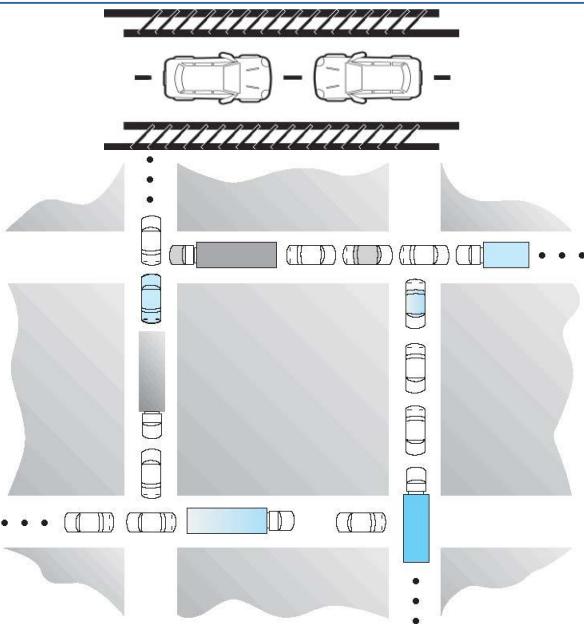
Operating System Concepts – 10<sup>th</sup> Edition

8.5

Silberschatz, Galvin and Gagne ©2018



## Deadlock Example on a one-way Bridge



Operating System Concepts

Galvin and Gagne ©2018





## Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion:** only one process at a time can use a resource
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait:** there exists a set  $\{P_0, P_1, \dots, P_n\}$  of waiting processes such that
  - $P_0$  is waiting for a resource that is held by  $P_1$ ,
  - $P_1$  is waiting for a resource that is held by  $P_2$ , ...,
  - $P_{n-1}$  is waiting for a resource that is held by  $P_n$ , and
  - $P_n$  is waiting for a resource that is held by  $P_0$ .



- Coffman, Elphick và Shoshani đã đưa ra 4 điều kiện cần có để làm xuất hiện tắc nghẽn:
- Có sử dụng tài nguyên không thể chia sẻ (Mutual exclusion): Mỗi thời điểm, một tài nguyên không thể chia sẻ được hệ thống cấp phát chỉ cho một tiến trình , khi tiến trình sử dụng xong tài nguyên này, hệ thống mới thu hồi và cấp phát tài nguyên cho tiến trình khác.
- Sự chiếm giữ và yêu cầu thêm tài nguyên (Wait for): Các tiến trình tiếp tục chiếm giữ các tài nguyên đã cấp phát cho nó trong khi chờ được cấp phát thêm một số tài nguyên mới.
- Không thu hồi tài nguyên từ tiến trình đang giữ chúng (No preemption-độc quyền): Tài nguyên không thể được thu hồi từ tiến trình đang chiếm giữ chúng trước khi tiến trình này sử dụng chúng xong.
- Tồn tại một chu kỳ trong đồ thị cấp phát tài nguyên ( Circular wait): có ít nhất hai tiến trình chờ đợi lẫn nhau : tiến trình này chờ được cấp phát tài nguyên đang bị tiến trình kia chiếm giữ và ngược lại.
- Khi có đủ 4 điều kiện này, thì tắc nghẽn xảy ra. Nếu thiếu một trong 4 điều kiện trên thì không có tắc nghẽn.

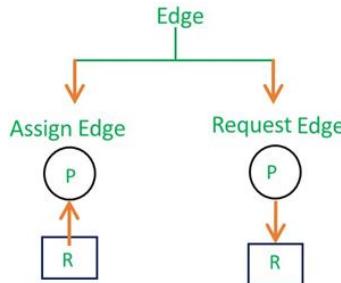




## Resource-Allocation Graph

A set of vertices  $V$  and a set of edges  $E$ .

- $V$  is partitioned into two types:
  - $P = \{P_1, P_2, \dots, P_n\}$ , the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \dots, R_m\}$ , the set consisting of all resource types in the system
- **request edge** – directed edge  $P_i \rightarrow R_j$
- **assignment edge** – directed edge  $R_j \rightarrow P_i$



Operating System Concepts – 10<sup>th</sup> Edition

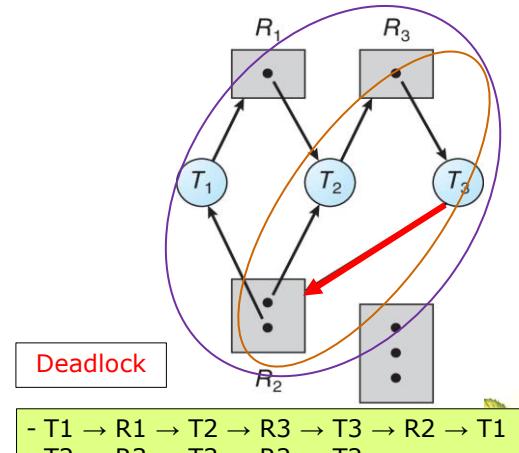
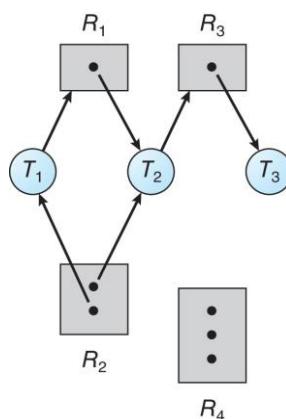
8.9

Silberschatz, Galvin and Gagne ©2018



## Resource Allocation Graph Example

- One instance of  $R_1$  ; Two instances of  $R_2$  ; One instance of  $R_3$  ; Three instance of  $R_4$
- $T_1$  holds one instance of  $R_2$  and is waiting for an instance of  $R_1$
- $T_2$  holds one instance of  $R_1$ , one instance of  $R_2$ , and is waiting for an instance of  $R_3$
- $T_3$  holds one instance of  $R_3$



Operating System Concepts – 10<sup>th</sup> Edition

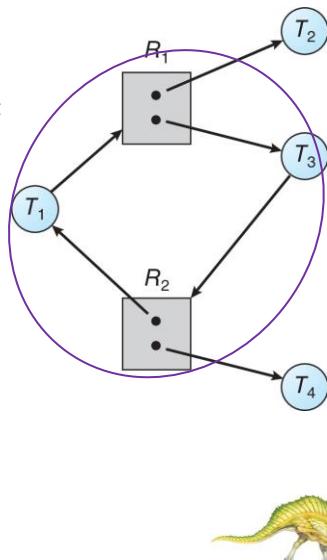
8.10

Silberschatz, Galvin and Gagne ©2018



## Graph with a Cycle But no Deadlock

- $T_1 \rightarrow R_1 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1$
- Observe that:
  - Thread  $T_4$  (holding one instance of  $R_2$ ) may **release its instance of resource type  $R_2$** . That resource can then be allocated to  $T_3$ , breaking the cycle => no deadlock



### In summary

- If graph contains no cycles  $\Rightarrow$  no deadlock
  - If graph contains a cycle
    - If only one instance per resource type, then deadlock
    - If several instances per resource type, possibility of deadlock
- => This observation is important when we deal with the deadlock problem.



## Methods for Handling Deadlocks

- Ensure that the system will **never** enter a deadlock state:
  - Deadlock prevention
    - **Mutual Exclusion:** One or more than one resource are non-shareable (Only one process can use at a time)
    - **Hold and Wait:** A process is holding at least one resource and waiting for resources.
    - **No Preemption:** A resource cannot be taken from a process unless the process releases the resource.
    - **Circular Wait:** A set of processes are waiting for each other in circular form.
  - Deadlock avoidance
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system.
  - What is the consequence?



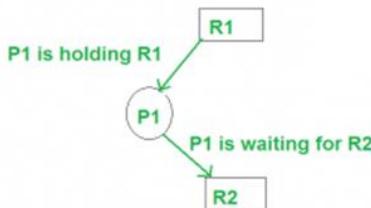


## Deadlock Prevention

Invalidate one of the four necessary conditions for deadlock:

- **Mutual Exclusion –**
  - sharable resources (e.g., read-only files): not required
  - non-sharable resources (e.g: printer): do not
- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Require processes to request and be allocated all its resources before it begins execution, or
  - Allow processes to request resources only when the process has none allocated to it.

Limit: Low resource utilization; starvation possible



Operating System Concepts – 10<sup>th</sup> Edition

Silberschatz, Galvin and Gagne ©2018



## Deadlock Prevention (Cont.)

- **No Preemption:**
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all the resources currently being held by the process are released
  - Preempted resources are added to the list of resources for which the process is waiting
  - Process will be **restarted** only when it can regain its old resources, as well as the new ones that it is requesting
- **Circular Wait:**
  - Impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration



Operating System Concepts – 10<sup>th</sup> Edition

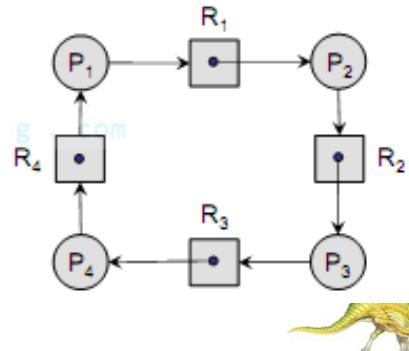
8.14

Silberschatz, Galvin and Gagne ©2018



## Circular Wait

- Cách 1: mỗi process yêu cầu thực thể của tài nguyên theo thứ tự tăng dần (định nghĩa bởi hàm F) của loại tài nguyên. Ví dụ
  - Chuỗi yêu cầu thực thể hợp lệ: tape drive -> disk drive -> printer
  - Chuỗi yêu cầu thực thể không hợp lệ: disk drive -> tape drive
- Cách 2: Khi một process yêu cầu một thực thể của loại tài nguyên  $R_j$  thì nó phải trả lại các tài nguyên  $R_i$  với  $F(R_i) > F(R_j)$ .
- Ví dụ
  - “Chứng minh” cho cách 1: phản chứng
    - P1:  $F(R_4) < F(R_1)$
    - P2:  $F(R_1) < F(R_2)$
    - P3:  $F(R_2) < F(R_3)$
    - P4:  $F(R_3) < F(R_4)$
  - Vậy  $F(R_4) < F(R_4)$ , mâu thuẫn!
  - P1 y/c tài nguyên  $R_1$  thì P1 phải trả tài nguyên  $R_4$



Operating System Concepts – 10<sup>th</sup> Edition

8.15

Silberschatz, Galvin and Gagne ©2018



## Circular Wait

- Invalidating the circular wait condition is most commonly used.
- Simply assign each resource (i.e., mutex locks) a unique number.
- Resources must be acquired in order.
- Example: two mutex locks

```
first_mutex = 1  
second_mutex = 5
```

code for **thread\_two** could not be written as follows:

```
/* thread.one runs in this function */  
void *do_work_one(void *param)  
{  
    pthread_mutex_lock(&first_mutex);  
    pthread_mutex_lock(&second_mutex);  
    /*  
     * Do some work  
     */  
    pthread_mutex_unlock(&second_mutex);  
    pthread_mutex_unlock(&first_mutex);  
  
    pthread_exit(0);  
}  
  
/* thread.two runs in this function */  
void *do_work_two(void *param)  
{  
    pthread_mutex_lock(&second_mutex);  
    pthread_mutex_lock(&first_mutex);  
    /*  
     * Do some work  
     */  
    pthread_mutex_unlock(&first_mutex);  
    pthread_mutex_unlock(&second_mutex);  
  
    pthread_exit(0);  
}
```

Operating System Concepts – 10<sup>th</sup> Edition

8.16

Silberschatz, Galvin and Gagne ©2018



## Deadlock Avoidance

Requires that the system has some additional **a priori** information available

- Simplest and most useful model requires that each process declare the **maximum number** of resources of each type that it may need
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes



## Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- System is in **safe state** if there exists a sequence

$$P_1, P_2, \dots, P_n$$

of ALL the processes in the systems such that for each  $P_i$ , the resources that  $P_i$  can still request can be satisfied by currently available resources + resources held by all the  $P_j$  with  $j < i$

**Need [i,j] = Max[i,j] – Allocation [i,j]**

**Available= Available + Allocation [i,j]**

- That is:
  - If  $P_i$  resource needs are not immediately available, then  $P_i$  can wait until all  $P_j$  have finished
  - When  $P_j$  is finished,  $P_i$  can obtain needed resources, execute, return allocated resources, and terminate
  - When  $P_i$  terminates,  $P_{i+1}$  can obtain its needed resources, and so on





## Safe State ex

- Ví dụ: Hệ thống có **12** tape drive và 3 quá trình P0, P1, P2
- Tại thời điểm t0, dữ liệu cho như bảng,
  - => hệ thống còn **3** tape drive sẵn sàng (12-5-2-2).
  - Sequence **(P1, P0, P2)**: safe?:
    - ▶ P1: get and return => avail: **3+2=5**
    - ▶ P0: get and return => avail: **5+5=10**
    - ▶ P2: get and return => avail: **10+2=12**

Process	Max - need	Allocation
P0	10	5
P1	4	2
P2	9	2

⇒ system is in a **safe state**

- Tại thời điểm t1, P2 yêu cầu và được cấp phát 1 tape drive còn **2** tape drive sẵn sàng
  - ▶ P1: get and return => avail: **2+2=4**
  - ▶ P0: get 5: wait
  - ▶ P2: get

⇒ Sequence **(P1, P0, P2)**: no safe

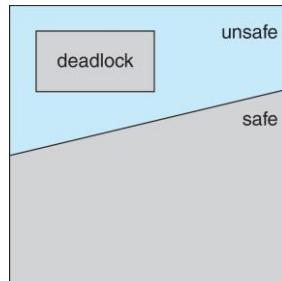
⇒ system is in a **no safe state**

Process	Max - need	Allocation
P0	10	5
P1	4	2
P2	9	<b>3</b>



## Safe, Unsafe, Deadlock State

- If a system is in safe state ⇒ no deadlocks
- If a system is in unsafe state ⇒ possibility of deadlock
- Avoidance algorithm:
  - Ensure that a system will never enter an unsafe state





## Avoidance Algorithms

- Single instance of a resource type
  - Use a modified resource-allocation graph
- Multiple instances of a resource type
  - Use the Banker's Algorithm



## Modified Resource-Allocation Graph Scheme

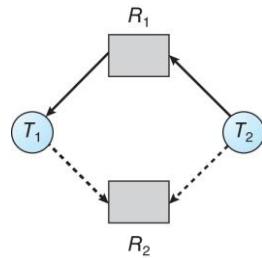
- **Claim edge**  $P_i \dashrightarrow R_j$  indicates that process  $P_i$  may request resource  $R_j$
- **Request edge**  $P_i \rightarrow R_j$  indicates that process  $P_i$  requests resource  $R_j$ 
  - Claim edge converts to request edge when a process requests a resource
- **Assignment edge**  $R_j \rightarrow P_i$  indicates that resource  $R_j$  was allocated to process  $P_i$ 
  - Request edge converts to an assignment edge when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Resources must be claimed **a priori** in the system



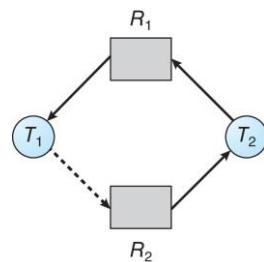


## Resource-Allocation Graph

- Modified Resource-Allocation Graph Example



- Unsafe State In Resource-Allocation Graph



## Resource-Allocation Graph Algorithm

- Suppose that process  $P_i$  requests a resource  $R_j$
- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph

- **Banker's Algorithm**

- Multiple instances of resources
- Each process must a priori claim maximum use
- When a process requests a resource, it may have to wait
- When a process gets all its resources it must return them in a finite amount of time





## Data Structures for the Banker's Algorithm

Let  $n$  = number of processes, and  $m$  = number of resources types.

- **Available:** Vector of length  $m$ . If available  $[j] = k$ , there are  $k$  instances of resource type  $R_j$  available
- **Max:**  $n \times m$  matrix. If  $\text{Max}[i,j] = k$ , then process  $P_i$  may request at most  $k$  instances of resource type  $R_j$
- **Allocation:**  $n \times m$  matrix. If  $\text{Allocation}[i,j] = k$  then  $P_i$  is currently allocated  $k$  instances of  $R_j$
- **Need:**  $n \times m$  matrix. If  $\text{Need}[i,j] = k$ , then  $P_i$  may need  $k$  more instances of  $R_j$  to complete its task

$$\text{Need}[i,j] = \text{Max}[i,j] - \text{Allocation}[i,j]$$



## Safety Algorithm

1. Let **Work** and **Finish** be vectors of length  $m$  and  $n$ , respectively.  
Initialize:

$$\text{Work} = \text{Available}$$

$$\text{Finish}[i] = \text{false} \text{ for } i = 0, 1, \dots, n-1$$

2. Find an  $i$  such that both:
  - (a)  $\text{Finish}[i] = \text{false}$
  - (b)  $\text{Need}_i \leq \text{Work}$If no such  $i$  exists, go to step 4
3.  $\text{Work} = \text{Work} + \text{Allocation}_i$   
 $\text{Finish}[i] = \text{true}$   
go to step 2
4. If  $\text{Finish}[i] == \text{true}$  for all  $i$ , then the system is in a **safe state**





## Example of Banker's Algorithm

- 5 processes  $P_0$  through  $P_4$ :
- 3 resource types: A (10 instances), B (5 instances), and C (7 instances)
- Snapshot at time  $T_0$ :

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
$P_0$	0	1	0	7	5	3	3	3	2
$P_1$	2	0	0	3	2	2			
$P_2$	3	0	2	9	0	2			
$P_3$	2	1	1	2	2	2			
$P_4$	0	0	2	4	3	3			

Need		
A	B	C
7	4	3
1	2	2
6	0	0
0	1	1
4	3	1

$$\text{Need}[i,j] = \text{Max}[i,j] - \text{Allocation}[i,j]$$

	Allocation	Need	Work
	A B C	A B C	A B C
$P_0$	0 1 0	7 4 3	3 3 2
$P_1$	2 0 0	1 2 2	5 3 2
$P_2$	3 0 2	6 0 0	7 4 3
$P_3$	2 1 1	0 1 1	7 4 3
$P_4$	0 0 2	4 3 1	7 4 5

10 4 7 → 10 5 7

The system is in a safe state since the sequence  $\langle P_1, P_3, P_4, P_0, P_2 \rangle$  satisfies safety criteria

Silberschatz, Galvin and Gagne ©2018



## Expl: based on Need

### Tính toán an toàn dựa vào ma trận Need:

- Khởi tạo work=avail => Work(3,3,2)
- Nếu Need <=Work: Work= Work+Allo (so sánh với Work cập nhật mới nhất theo Pi)
  - P0: Need\_P0(7,4,3) > Work(3,3,2) => ko update Work
  - P1: Need\_P1(1,2,2) < Work(3,3,2) => Work = Work(3,3,2)+Allo\_P1(2,0,0) => update: **Work(5,3,2)**
  - P2: Need\_P2(6,0,0) > Work(5,3,2) => ko update Work
  - P3: Need\_P3(0,1,1) < Work(5,3,2) => Work= Work(5,3,2) + Allo\_P3(2,1,1) => update Work(7,4,3)
  - P4: Need\_P4(4,3,1) < Work(7,4,3) => Work= Work(7,4,3) + Allo\_P4(0,0,2) => update Work(7,4,5)
- Còn P0, P2 chưa cấp phát. Finish = True thì quay lại xem xét cấp phát lại (Step 2)
  - P0: Need\_P0(7,4,3) < Work(7,4,5) => Work= Work(7,4,5) + Allo\_P0(0,1,0) => update Work(7,5,5)
  - P2: Need\_P2(6,0,0) < Work(7,5,5) => Work= Work(7,5,5) + Allo\_P2(3,0,2) => update Work(10,5,7)
- Hệ thống trên ở trạng thái an toàn vì nó tồn tại thứ tự an toàn P1,3,4,0,2
- Cách check:
  - So sánh Work ở trạng thái cuối với đề ra có bao nhiêu instant, nếu khớp thì OK
  - Nếu ko cho instant ban đầu thì so Work ở trạng thái cuối với tổng từng kiểu instant ở các Process + instant ở Available, nếu khớp thì OK





## Resource-Request Algorithm for Process $P_i$

$\text{Request}_i$  = request vector for process  $P_i$

If  $\text{Request}_i[j] = k$  then process  $P_i$  wants  $k$  instances of resource type  $R_j$

1. If  $\text{Request}_i \leq \text{Need}_i$  go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If  $\text{Request}_i \leq \text{Available}$ , go to step 3. Otherwise  $P_i$  must wait, since resources are not available
3. Pretend to allocate requested resources to  $P_i$  by modifying the state as follows:

$\text{Available} = \text{Available} - \text{Request}_i;$

$\text{Allocation}_i = \text{Allocation}_i + \text{Request}_i;$

$\text{Need}_i = \text{Need}_i - \text{Request}_i;$

- If safe  $\Rightarrow$  the resources are allocated to  $P_i$
- If unsafe  $\Rightarrow P_i$  must wait, and the old resource-allocation state is restored

$\text{Available} = \text{Available} + \text{Request}_i;$

$\text{Allocation}_i = \text{Allocation}_i - \text{Request}_i;$

$\text{Need}_i = \text{Need}_i + \text{Request}_i;$



## Example: $P_1$ Request (1,0,2)

- $P_1$  Request (1,0,2): Check that  $\text{Request} \leq \text{Available}$  (that is,  $(1,0,2) \leq (3,3,2) \Rightarrow$  true)

- Before:

	Allocation			Available			Need		
	A	B	C	A	B	C	A	B	C
$P_0$	0	1	0	3	3	2	7	4	3
$P_1$	2	0	0				1	2	2
$P_2$	3	0	2				6	0	0
$P_3$	2	1	1				0	1	1
$P_4$	0	0	2				4	3	1

$\text{Avail} = \text{Avail} - \text{Request}_i;$

$\text{Alloc}_i = \text{Alloc}_i + \text{Request}_i;$

$\text{Need}_i = \text{Need}_i - \text{Request}_i;$

④  
①  
⑤  
②  
③

- Update:

	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
$P_0$	0	1	0	7	4	3	2	3	0
$P_1$	3	0	2	0	2	0			
$P_2$	3	0	2	6	0	0			
$P_3$	2	1	1	0	1	1			
$P_4$	0	0	2	4	3	1			

Pro_	Work:		
	A	B	C
$P_1$	2	3	0
$P_3$	5	3	2
$P_4$	7	4	3
$P_0$	7	5	5
$P_2$	10	5	7

- Sequence  $< P_1, P_3, P_4, P_0, P_2 >$  satisfies safety requirement

- $\Rightarrow P_1$  can be allocated resource





## Example:

- Can request for (3,3,0) by  $P_4$  be granted?
- Can request for (0,2,0) by  $P_0$  be granted?



## Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme





## Single Instance of Each Resource Type

- a deadlock detection algorithm that uses a variant of the resource-allocation graph, called **wait-for** graph

- Nodes are processes
  - $P_i \rightarrow P_j$  if  $P_i$  is waiting for  $P_j$  to release a resource  $R_i$  needs

We obtain this graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges

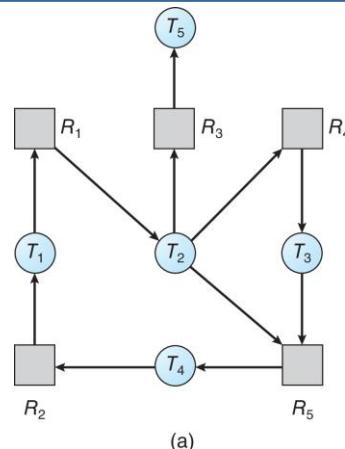
- To detect deadlocks:

- Maintain the wait-for graph
  - Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock

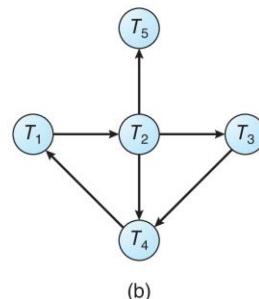
- An algorithm to detect a cycle in a graph requires an order of  $n^2$  operations, where  $n$  is the number of vertices in the graph



## Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph



Corresponding wait-for graph





## Several Instances of a Resource Type

- **Available:** A vector of length  $m$  indicates the number of available resources of each type
- **Allocation:** An  $n \times m$  matrix defines the number of resources of each type currently allocated to each process
- **Request:** An  $n \times m$  matrix indicates the current request of each process.
  - If  $\text{Request}[i][j] = k$ , then process  $P_i$  is requesting  $k$  more instances of resource type  $R_j$ .



## Detection Algorithm

1. Let  $Work$  and  $Finish$  be vectors of length  $m$  and  $n$ , respectively Initialize:
  - a)  $Work = Available$
  - b) For  $i = 1, 2, \dots, n$ , if  $\text{Allocation}_{i,i} \neq 0$ , then  $Finish[i] = \text{false}$ ; otherwise,  $Finish[i] = \text{true}$
2. Find an index  $i$  such that both:
  - a)  $Finish[i] == \text{false}$
  - b)  $\text{Request}_{i,i} \leq Work$If no such  $i$  exists, go to step 4
3.  $Work = Work + Allocation_i$ ,  
 $Finish[i] = \text{true}$   
go to step 2
4. If  $Finish[i] == \text{false}$ , for some  $i$ ,  $1 \leq i \leq n$ , then the system is in deadlock state. Moreover, if  $Finish[i] == \text{false}$ , then  $P_i$  is deadlocked  
If  $Finish[i] == \text{true}$ , then the system is in safe

*Algorithm requires an order of  $O(m \times n^2)$  operations to detect whether the system is in deadlocked state*





## Example of Detection Algorithm

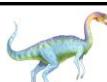
- Five processes  $P_0$  through  $P_4$ ; three resource types A (7 instances), B (2 instances), and C (6 instances)
- Snapshot at time  $T_0$ :

	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
$P_0$	0	1	0	0	0	0	0	0	0
$P_1$	2	0	0	2	0	2			
$P_2$	3	0	3	0	0	0			
$P_3$	2	1	1	1	0	0			
$P_4$	0	0	2	0	0	2			

- Sequence  $\langle P_0, P_2, P_3, P_4, P_1 \rangle$  or
- Sequence  $\langle P_0, P_2, P_3, P_1, P_4 \rangle$
- will result in  $Finish[ij] = \text{true}$  for all  $i$

=> The system is in safe

Process	A	B	C
$P_0$	0	0	0
$P_2$	3	1	3
$P_3$	5	2	4
$P_4$	5	2	6
$P_1$	7	2	6



## Example (Cont.)

- $P_2$  requests an additional instance of type C

Request		
A	B	C
0	0	0
2	0	2
0	0	1
1	0	0
0	0	2

- State of system?

- Can reclaim resources held by process  $P_0$ , but insufficient resources to fulfill other processes' requests
- Deadlock exists, consisting of processes  $P_1$ ,  $P_2$ ,  $P_3$ , and  $P_4$





## Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - ▶ One for each disjoint cycle
- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.



## When Deadlock

- When a detection algorithm determines that a deadlock exists, several alternatives are available.
  - One possibility is to inform the operator
  - Another possibility is to let the system recover from the deadlock automatically.
- recover from breaking a deadlock
  - Abort all deadlocked processes
  - Abort one process at a time until the deadlock cycle is eliminated





## Recovery from Deadlock: Process Termination

- In which order should we choose to abort?
  - Priority of the process
  - How long process has computed, and how much longer to completion
  - Resources the process has used
  - Resources process needs to complete
  - How many processes will need to be terminated
  - Is process interactive or batch?

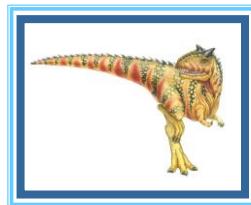


## Recovery from Deadlock: Resource Preemption

- **Selecting a victim** – minimize cost
- **Rollback** – return to some safe state, restart process for that state
- **Starvation** – same process may always be picked as victim, include number of rollback in cost factor



# End of Chapter 2



# Chapter 3: Memory Management

## 3.1. Main Memory



GV: Nguyễn Thị Thanh Vân

Operating System Concepts – 10<sup>th</sup> Edition

Silberschatz, Galvin and Gagne ©2018



## Outline

- Background
- Contiguous Memory Allocation
  - Fixed partition allocation
  - Variable Partition Allocation
- Non-contiguous Memory Allocation
  - Paging
  - Segmentation
  - Segmentation with paging
- Structure of the Page Table. Techniques:
  - Hierarchical Paging
  - Hashed Page Tables
  - Inverted Page Tables
- Swapping
- Example: The Intel 32 and 64-bit Architectures



Operating System Concepts – 10<sup>th</sup> Edition

9.2

Silberschatz, Galvin and Gagne ©2018

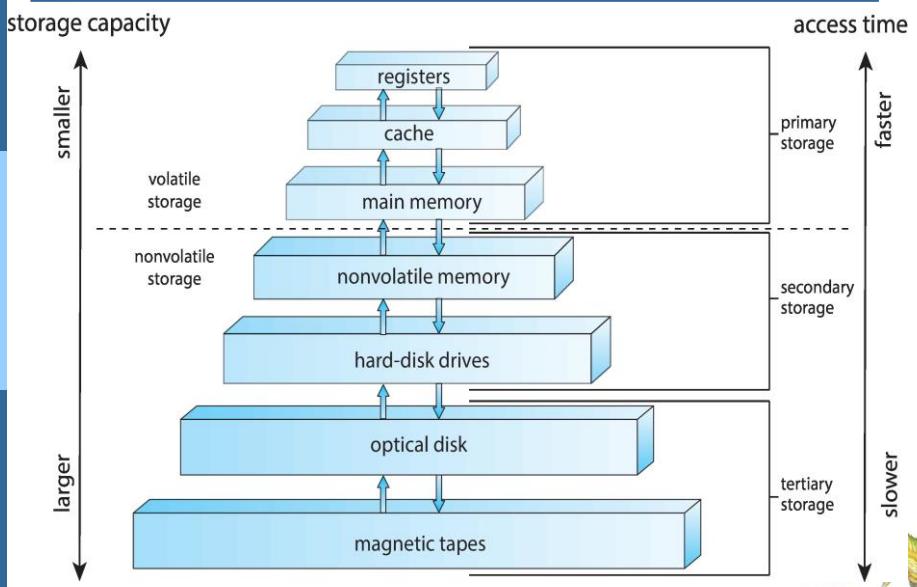


## Objectives

- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques,
- To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging



## Storage-Device Hierarchy





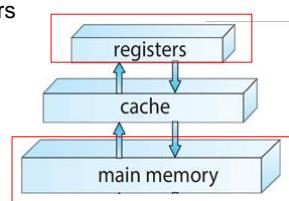
## OS with memory management

- OS chế độ đa nhiệm nhằm nâng cao hiệu suất sử dụng CPU.
  - nảy sinh nhu cầu chia sẻ bộ nhớ giữa các tiến trình khác nhau .
  - Vẫn đề nằm ở chỗ : « bộ nhớ thì hữu hạn và các yêu cầu bộ nhớ thì vô hạn ».
- OS chịu trách nhiệm cấp phát vùng nhớ cho các tiến trình có yêu cầu. Để thực hiện tốt nhiệm vụ này, OS cần phải xem xét nhiều khía cạnh :
  - Sự tương ứng giữa địa chỉ logic và địa chỉ vật lý (physic) : làm cách nào để chuyển đổi một địa chỉ tượng trưng (symbolic) trong chương trình thành một địa chỉ thực trong bộ nhớ chính?
  - Quản lý bộ nhớ vật lý: làm cách nào để mở rộng bộ nhớ có sẵn nhằm lưu trữ được nhiều tiến trình đồng thời?
  - Chia sẻ thông tin: làm thế nào để cho phép hai tiến trình có thể chia sẻ thông tin trong bộ nhớ?
  - Bảo vệ: làm thế nào để ngăn chặn các tiến trình xâm phạm đến vùng nhớ được cấp phát cho tiến trình khác?
- Các giải pháp quản lý bộ nhớ phụ thuộc rất nhiều vào đặc tính phần cứng và trải qua nhiều giai đoạn cải tiến để trở thành những giải pháp khá ổn như hiện nay.



## Background

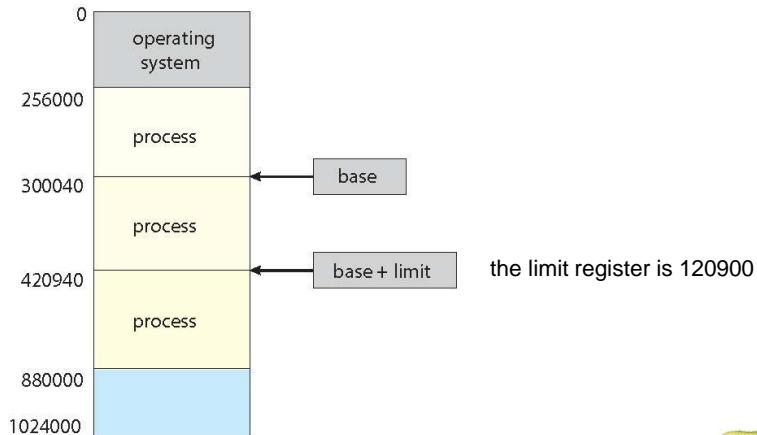
- Program is permanently kept on **backing store** (disks)
- When run: it must be brought from disk into memory and placed within a process
- CPU can access directly to main memory and registers
- Memory unit only sees a stream of:
  - addresses + read requests, or
  - address + data and write requests
- Register access is done in one CPU clock (or less)
- Main memory access can take many cycles, causing a **stall**, since it does not have the data required to complete the instruction that it is executing
  - Solution: add fast memory between the CPU and main memory, typically on the CPU chip for fast access
- **Cache** sits between main memory and CPU registers
- Protection of memory is required to ensure correct operation





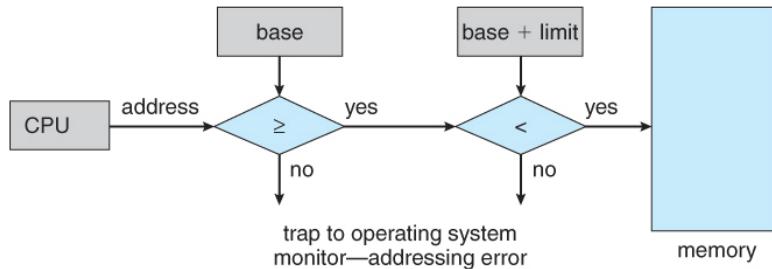
## Memory Protection

- Need to ensure that a process can access only those addresses in its address space
- We can provide this protection by using a pair of **base** and **limit registers** to define the logical address space of a process



## Hardware Address Protection

- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user
- Hardware address protection with base and limit registers:



- The OS loads the base and limit registers
  - uses a special privileged instruction, executes only in kernel mode,
- OS can change the value of the registers but prevents user programs from changing the registers' contents





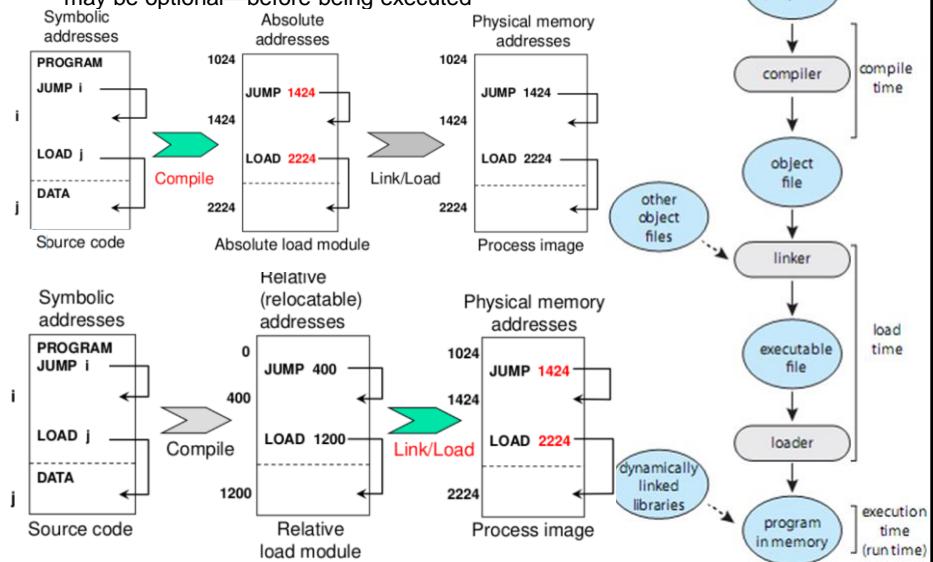
## Address Binding

- Programs on disk, ready to be brought into memory to execute, are placed in an **input queue**
  - Without support, must be loaded into address 0000
- Inconvenient to have first user process physical address always at 0000
  - How can it not be?  
user program actually places a process in physical memory.
- Addresses represented in different ways at different stages of a program's life
  - Source code addresses are usually symbolic
  - Compiled code addresses **bind** to relocatable addresses
    - i.e., "14 bytes from beginning of this module"
  - Linker or loader will bind relocatable addresses to absolute addresses
    - i.e., 74014
  - Each binding maps one address space to another



## Multistep Processing of a User Program

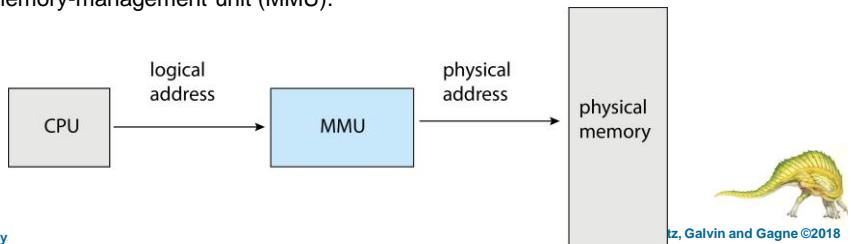
- a user program goes through several steps—some of which may be optional—before being executed





## Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
- Logical address (bit)** generated by the CPU; or **virtual address**
  - Logical address space (byte):** set of all logical addresses generated by a program
- Physical address (bit)** – address seen by the memory unit (actually available)
  - Physical address space** set of all physical addresses generated by a program
- Logical and physical addresses are:
  - the **same** in compile-time and load-time address-binding schemes
  - differ** in execution-time address-binding scheme
- Hardware device that at run time maps virtual to physical address => called memory-management unit (MMU).



Operating Sy

iz, Galvin and Gagne ©2018



## Compare LA & PA

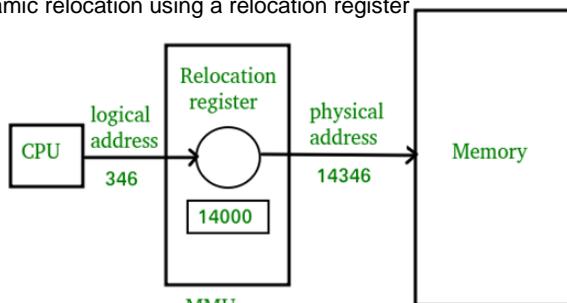
Paramenter	Logical Address	Physical Address
Basic	generated by CPU	location in a memory unit
Address Space	Logical Address Space is set of all logical addresses generated by CPU in reference to a program.	Physical Address is set of all physical addresses mapped to the corresponding logical addresses.
Visibility	User can view the logical address of a program.	User can never view physical address of program.
Generation	generated by the CPU	Computed by MMU
Access	The user can use the logical address to access the physical address.	The user can indirectly access physical address but not directly.
Editable	Logical address can be change.	Physical address will not change.
Also called	virtual address.	real address.





## Relocation Register

- A simple MMU scheme is a generalization of the base-register scheme.
- The base register now called **relocation register**
- The value in the relocation register is added to every address generated by a user process at the time it is sent to memory
- The user program deals with *logical* addresses; it never sees the *real* physical addresses
  - Execution-time binding occurs when reference is made to location in memory
  - Logical address bound to physical addresses
- Ex: Dynamic relocation using a relocation register



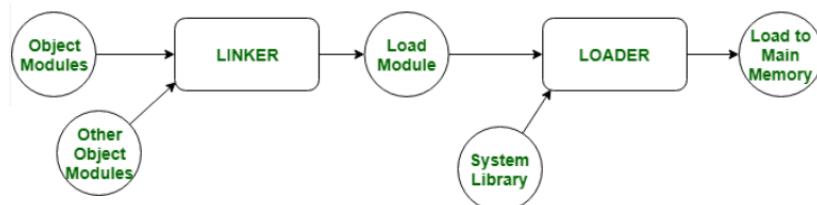
Operating System Concepts – 10<sup>th</sup> Edition

9.14

Silberschatz, Galvin and Gagne ©2018



## Execution of a program



- **Linking** and **Loading** are the utility programs that play a important role in the execution of a program.
  - Linking intakes the object codes generated by the assembler and combines them to generate the executable module.
  - Loading loads this executable module to the main memory for execution.



Operating System Concepts – 10<sup>th</sup> Edition

9.15

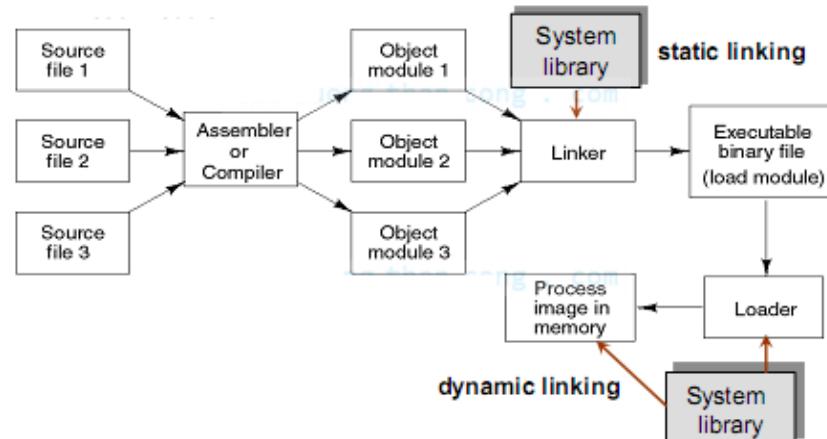
Silberschatz, Galvin and Gagne ©2018



## Loading

- Bộ linker:

- Tái định vị địa chỉ tương đối và phân giải các external reference
- Kết hợp các object module thành một file nhị phân khả thi gọi là **load module**



Operating System Concepts – 10<sup>th</sup> Edition

9.16

Silberschatz, Galvin and Gagne ©2018



## Dynamic Linking

- **Static linking** – system libraries and program code combined by the loader into the binary program image.
- **Dynamic linking** – system libraries that are linked to user programs when the programs are run, is postponed until execution time
- Small piece of code, called **stub**, is used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system checks if routine is in processes' memory address
  - If not in address space, add to address space
- Dynamic linking is particularly useful for libraries -
- System also known as **shared libraries**



Operating System Concepts – 10<sup>th</sup> Edition

9.17

Silberschatz, Galvin and Gagne ©2018



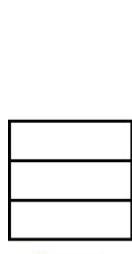
## Dynamic Loading

- The program consist of main part and a number of routines
- The entire program does need to be in memory to execute
- Routine is not loaded until it is called
- To obtain better memory-space utilization, we can use **dynamic loading**.
- All routines kept on disk in relocatable load format
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required
  - Implemented through program design
  - OS can help by providing libraries to implement dynamic loading



## Memory Allocation

- How to allocate available memory to the processes that are waiting to be brought into memory
- Two methods:



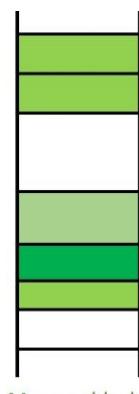
Process



Contiguous Memory Allocation



Process



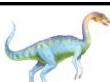
Noncontiguous Memory Allocation



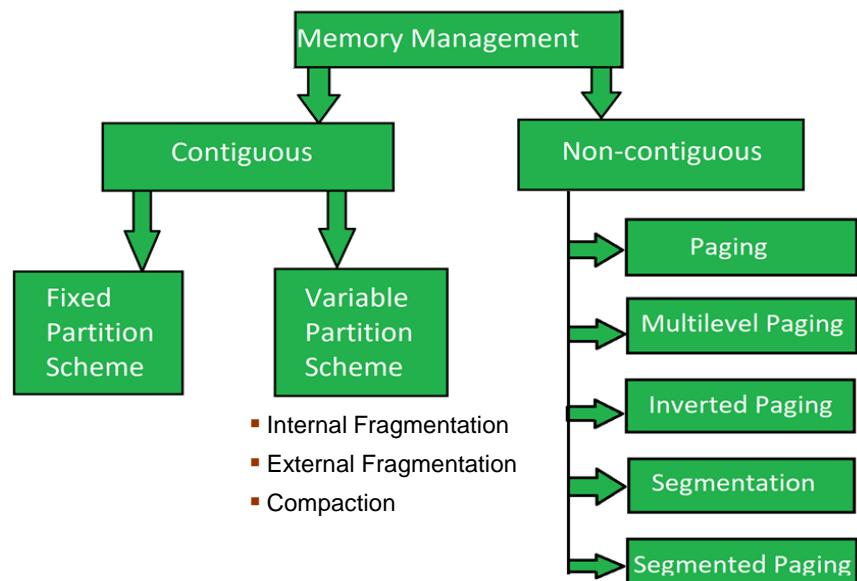


## Memory Allocation

N.	Contiguous Memory Allocation	Non-Contiguous Memory Allocation
1.	Contiguous memory allocation allocates consecutive blocks of memory to a file/process.	Non-Contiguous memory allocation allocates separate blocks of memory to a file/process.
2.	Faster in Execution.	Slower in Execution.
3.	It is easier for the OS to control.	It is difficult for the OS to control.
4.	Overhead is minimum as not much address translations are there while executing a process.	More Overheads are there as there are more address translations.
5.	Internal fragmentation occurs in Contiguous memory allocation method.	External fragmentation occurs in Non-Contiguous memory allocation method.
6.	It includes single partition allocation and multi-partition allocation.	It includes paging and segmentation.
7.	Wastage of memory is there.	No memory wastage is there.
8.	In contiguous memory allocation, swapped-in processes are arranged in the originally allocated space.	In non-contiguous memory allocation, swapped-in processes can be arranged in any place in the memory.



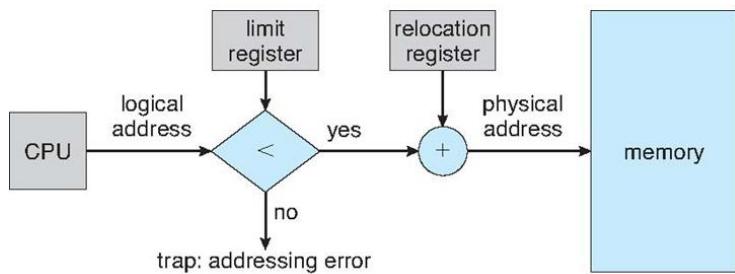
## Types of memory management





## Memory Protection

- Relocation registers used to protect user processes from each other, and from changing OS code and data
  - Relocation register contains value of **smallest** physical address
  - Limit register contains **range** of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address *dynamically* by adding the value in the relocation register
  - Can then allow actions such as kernel code being **transient** and kernel changing size
- Hardware support for relocation and limit registers:



Operating System

igne ©2018



## Memory Allocation - Contiguous allocation

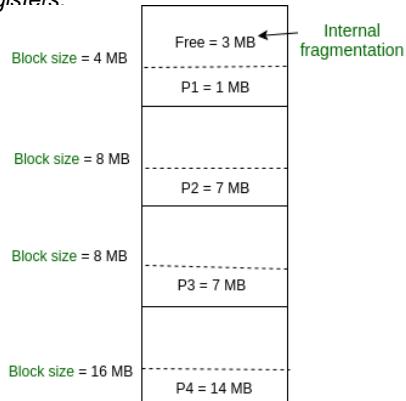
- **Contiguous allocation** is one early method
  - Main memory usually consists of two **partitions**:
    - Resident operating system, usually held in low memory with interrupt vector
    - User processes then held in high memory
  - Each process contained in single contiguous section of memory
- 2 types:
  - Fixed partition allocation
  - Variable Partition Allocation





## Fixed Partitioning Allocation

- Main memory is divided into many partitions (same size or different size)
- The simplest technique used to put more than 1 processes in the main memory
  - In every partition only one process will be accommodated.
- Degree of multi-programming is restricted by number of partitions in the memory.
  - Maximum size of the process is restricted by maximum size of the partition.
- Every partition is associated with the *limit registers*.
- **Limit: Internal Fragmentation:**  
Any program, no matter how small, occupies an entire partition.  
=> Main memory use is inefficient.

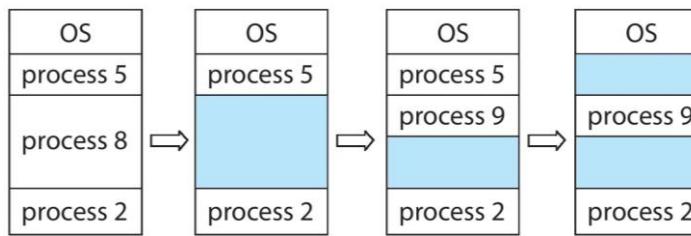


Sum of Internal Fragmentation in every block  
 $(4-1)+(8-7)+(8-7)+(16-14)= 3+1+1+2 = 7\text{MB}$



## Variable Partition Allocation

- Degree of multiprogramming limited by number of partitions
- **Variable-partition** sizes for efficiency (sized to a given process' needs)
- **Hole** – block of available memory; holes of various size are scattered throughout memory
- When a process arrives, it is allocated memory from a hole large enough to accommodate it
- Process exiting frees its partition, adjacent free partitions combined
- Operating system maintains information about:
  - (a) allocated partitions
  - (b) free partitions (holes)





## Dynamic Storage-Allocation Problem

- This problem concerns:
  - How to satisfy a request of size  $n$  from a list of free holes? ...
- Solution for selecting a free hole from the set of available holes. Strategy:
  - **First-fit:** Allocate the *first* hole that is big enough
  - **Best-fit:** Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size
    - ▶ Produces the smallest leftover hole
  - **Worst-fit:** Allocate the *largest* hole; must also search entire list
    - ▶ Produces the largest leftover hole
  - **Next Fit:** similar to the first fit but it will search for the first sufficient partition from the last allocation point.
- First-fit and best-fit better than worst-fit in terms of speed and storage utilization
- Limit: External Fragmentation



## Ex

### ▪ First-fit

PROCESS A  
= 25 KB



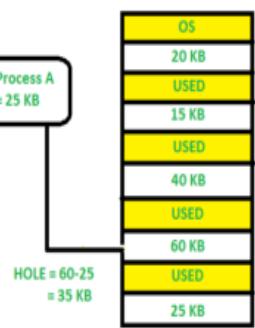
### Best-fit

PROCESS A  
= 25 KB



### Worst-fit

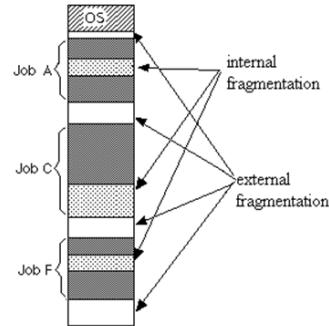
Process A  
= 25 KB





## Fragmentation

- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- **External Fragmentation** – total memory space available to satisfy a request, but it is not contiguous
- First fit analysis reveals that given  $N$  blocks allocated,  $0.5 N$  blocks lost to fragmentation:  $1/3$  may be unusable  $\rightarrow$  **50-percent rule**
- Can reduce external fragmentation by **compaction**



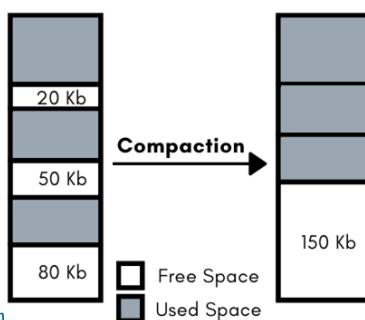
Operating System Concepts

Silberschatz, Galvin and Gagne ©2018



## Compaction

- One solution to the problem of **external fragmentation**
- Shuffle memory contents to place all free memory together in one large block
- Compaction is possible *only* if relocation is dynamic, and is done at execution time
- I/O problem. I/O done while compaction. Data arrives in wrong place.
- Solutions to I/O problem
  - Latch process in memory while it is involved in I/O
  - Do I/O only into OS buffers
- Now consider that **backing store** has same fragmentation problems but on disks



Operating System Concepts – 10<sup>th</sup> Edition

Silberschatz, Galvin and Gagne ©2018



## Memory Allocation - Non-contiguous Allocation

- Allow a process to reside in different locations on memory
  - Segmentation
    - Segmentation
    - Program and segmentation
    - Segmentation Hardware
    - Segmentation: Adv and Disadv
  - Paging
    - Paging
    - Logical Address & Physical Address
    - Paging Hardware
    - Paging Model
    - Paging -- Calculating internal fragmentation
    - Allocating Free Frames
    - Implementation of Page Table, using PTBR
    - Paging Hardware With TLB
  - Paged segmentation



## Segmentation

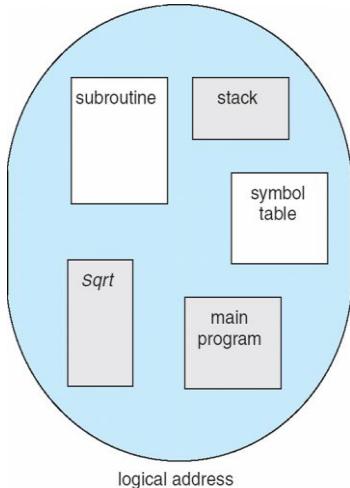
- Memory-management scheme that supports user view of memory
- A program is a collection of segments
- A segment is a logical unit such as:
  - main program
  - procedure
  - function
  - method
  - object
  - local variables, global variables
  - common block
  - stack
  - symbol table
  - arrays



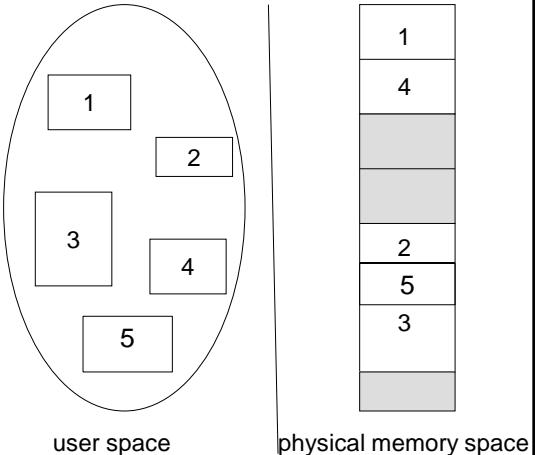


## Program and segmentation

### User's View of a Program



### Logical View of Segmentation



Operating System Concepts – 10<sup>th</sup> Edition

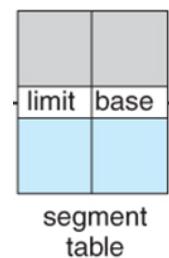
9.32

Silberschatz, Galvin and Gagne ©2018



## Segmentation Architecture

- **Segment table** – maps two-dimensional physical addresses; include:
  - **base** – contains the starting physical address where the segments reside in mem
  - **limit** – specifies the length of the segment
- **Segment-table base register (STBR)** points to the segment tables location in mem
- **Segment-table length register (STLR)** indicates number of segments used by a program.
- A logical address: <s,d>
  - s - segment number : index in segment table
  - d: an offset into that segment,
- Note:
  - Segment number **s** is legal if **s < STLR**
  - an offset is legal: **d < limit**



Operating System Concepts – 10<sup>th</sup> Edition

9.33

Silberschatz, Galvin and Gagne ©2018



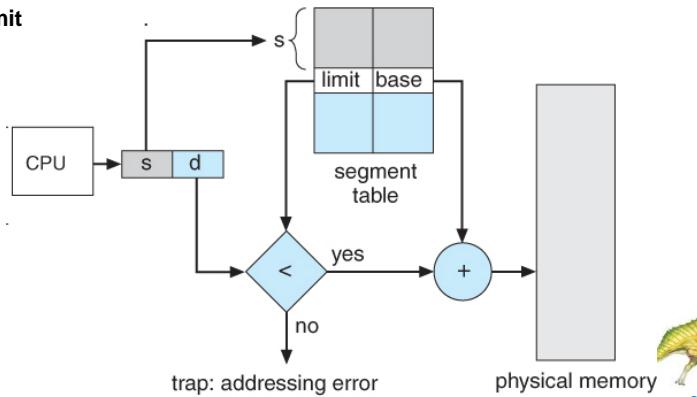


## Segmentation Hardware

- A logical address consists of two parts: **<segment-number, offset>**
- an offset into that segment, d - be between 0 and the segment limit
  - If it is not, we trap to the OS
  - an offset is legal, it is added to the segment base to produce the address in physical memory: **correct offset+ base (in table segment)**

► **base** is mapped from **s**

► **d < limit**



Operating System Concepts – 10<sup>th</sup> Edition

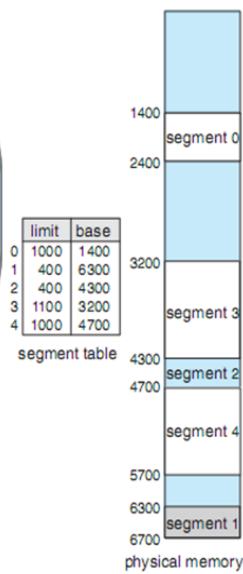
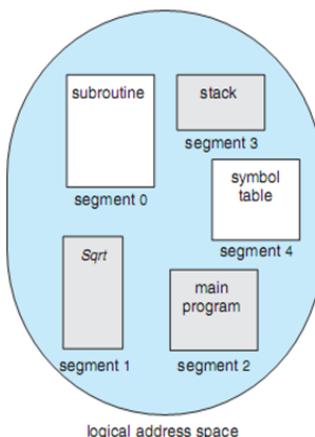
9.34

Silberschatz, Galvin and Gagne ©2018



## Segmentation, ex

- Logical Addr (s,d), **check:**
  - s in seg table, d<limit
- Seg2=400bytes, at 4300
- Thus: CPU want to
  - A ref to byte 53 of seg 2 (s=2, d=53) is mapped onto location:  $4300 + 53 = 4353$
  - A ref to byte 852 of seg 3 (s=3, d=852) is mapped onto location:  $3200 + 852 = 4052$
  - A ref to byte 1222 of seg 0: would result in a trap to the OS, as this segment is only 1,000 bytes
- Physical address space of a process can be noncontiguous;
  - Varying size memory chunks
  - External fragmentation



Operating System Concepts – 10<sup>th</sup> Edition

9.35

Silberschatz, Galvin and Gagne ©2018



## Segmentation: Adv and Disadv

### ▪ Advantages of Segmentation

- Segmentation is more close to the programmer's view of physical memory.
- Segmentation prevents internal fragmentation.
- Segmentation prevents the CPU overhead as the segment contain an entire module of at once.

### ▪ Disadvantages of Segmentation

- The segmentation leads to **external fragmentation**.



## Paging

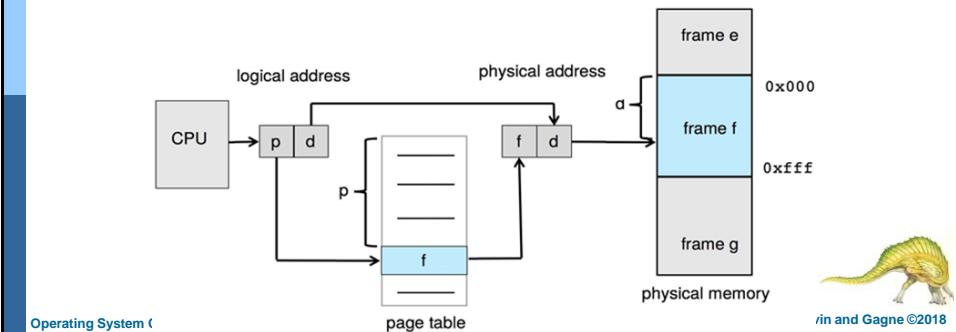
- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
  - Avoids **external fragmentation**
  - Avoids problem of varying sized memory chunks
- **Paging** is implemented through cooperation between the operating system and the computer hardware
- Implementing paging involves:
  - breaking physical memory into fixed-sized blocks called **frames** and
  - breaking logical memory into blocks of the same size called **pages**
- To run a program of size  $N$  pages, need to find  $N$  free frames and load program
- Backing store likewise split into **blocks**
- Still have **Internal fragmentation**



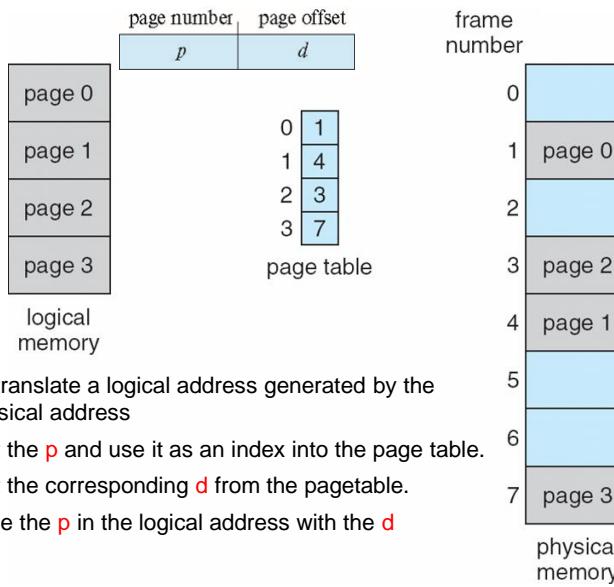


## Paging Hardware

- Set up a page table to translate logical to physical addresses. Table contains:
  - The **base address** of each frame in physical memory.
  - The **offset** is the location in the frame being referenced
- Every address (logic) generated by the CPU is divided into two parts: **p & d**
  - **Page number (p)** – used as an index into a page table which contains base address of each page in physical memory
  - **Page offset (d)** – combined with **base address** to define the physical memory address that is sent to the memory unit



## Paging Model of Logical and Physical Memory



- The MMU to translate a logical address generated by the CPU to a physical address
  - 1. Extract the **p** and use it as an index into the page table.
  - 2. Extract the corresponding **d** from the pagetable.
  - 3. Replace the **p** in the logical address with the **d**



## Logical Address & Physical Address

### ▪ Logical Address

- LA Space -> blocks (pages), is divided into 2 parts p&d,
  - ▶ **Page number (p)** an index contains base address
  - ▶ **Page offset (d)** combined with **base address** => physical memory address

### ▪ Physical Address

- PA Space -> blocks (frames), is divided into 2 parts f&d
  - ▶ **Frame number(f):** Number of bits required to represent the frame of Physical Address Space or Frame number.
  - ▶ **Frame offset(d):** Number of bits required to represent particular word in a frame or frame size of Physical Address Space or word number of a frame

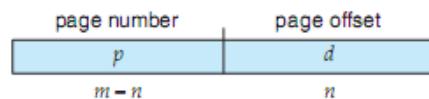
### ■ Ex:

- If LA= 31bit, then LA Space =  $2^{31}$ words = 2GB words ( $1G=2^{30}$ )
- If LA Space = 128 MB words =  $2^7 * 2^{20}$  words, then LA=  $\log_2 2^{27} = 27$  bits
- If PA = 22 bit, then PA Space =  $2^{22}$  words = 4 MB words ( $1M = 2^{20}$ )
- If PA Space = 16 MB words =  $2^4 * 2^{20}$  words, then PA =  $\log_2 2^{24} = 24$  bits



## Logical address and logical address space

- For given logical address space size  $2^m$  and page size  $2^n$  bytes, we have



- Ex: In the logical address, n=2 and m=4
  - ⇒ a page size:  $2^n = 4$  bytes (= frame sz)
  - ⇒ Number of page:  $2^{m-n} = 4$

0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p

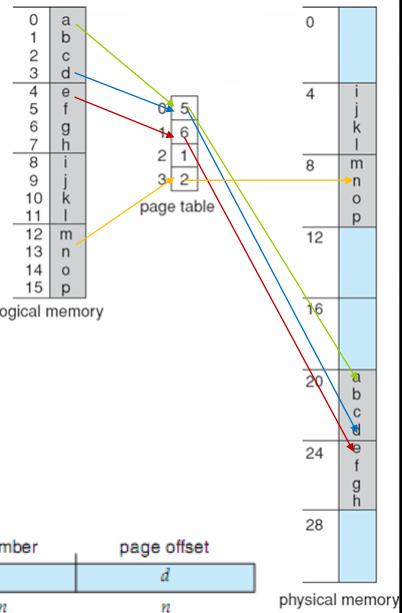
logical memory





## Paging Example

- Consider the memory: 1 page has  $x$  of size)
  - In the logical address,  $n=2$  and  $m=4$
  - => a page size:  $2^n = 4$  bytes (= frame sz)
  - => Number of page:  $2^{m-n} = 4$
  - A physical memory of 32 bytes: (sz=4)  
 $32/4=8$  frames
- The programmer's view of memory can be mapped into physical memory:
  - LA 0 (page 0 (frame 5), offset 0) maps to PA: 20 [=  $(5 \times 4) + 0$ ].
  - LA 3 (page 0 (frame 5), offset 3) maps to PA: 23 [=  $(5 \times 4) + 3$ ].
  - LA 4 (page 1 (frame 6), offset 0) maps to PA: 24 [=  $(6 \times 4) + 0$ ].
  - LA 13 (page 3 (frame 2), offset 1) maps to PA: 9 [=  $(2 \times 4) + 1$ ]



Operating System Concepts – 10<sup>th</sup> Edition

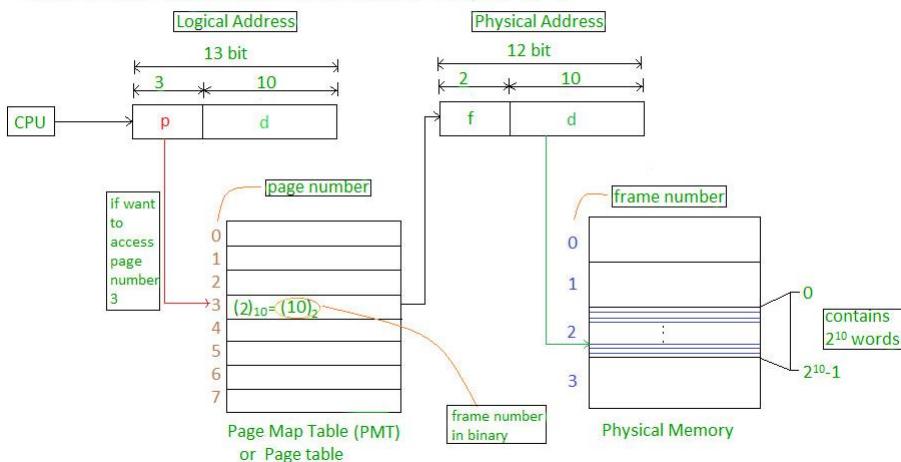


## Ex

- Physical Address = 12 bits => Physical Address Space =  $2^{12} = 4.2^{10}$  B = 4K words
- Logical Address = 13 bits => Logical Address Space =  $2^{13} = 8.2^{10}$  B = 8 K words
- Page size = frame size = 1 K words (assumption)

$$\text{Number of frames} = \text{Physical Address Space} / \text{Frame size} = 4 \text{ K} / 1 \text{ K} = 4 = 2^2$$

$$\text{Number of pages} = \text{Logical Address Space} / \text{Page size} = 8 \text{ K} / 1 \text{ K} = 8 = 2^3$$





## Paging -- Calculating internal fragmentation

- For example, if
  - Page size = 2,048 bytes
  - Process size = 72,766 bytes (=  $35 \times 2048 + 1086$ )
- Then, will need: 35 pages + 1,086 bytes => will be allocated 36 frames
- Resulting in Internal fragmentation of  $2,048 - 1,086 = 962$  bytes
- **Worst case fragmentation:** a process need n pages + 1byte
  - It would be allocated:  $n + 1$  frames => internal fragmentation of almost 1 frame.
  - Ex:  $35 \times 2048 + 1 = 72765$ bytes
- On average fragmentation =  $1 / 2$  frame size
- This consideration suggests that small page sizes are desirable?
  - However, overhead is involved in each page-table entry, it reduced as the size of the pages increases
  - disk I/O is more efficient when the amount of data being transferred is larger
- But each page table entry takes memory to track
- Page sizes growing over time
  - Solaris supports two-page sizes – 8 KB and 4 MB

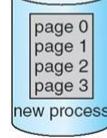


## Allocating Free Frames

- OS is managing physical memory: Keep a list of free frames – called **frame table**:
  - has 1 entry for 1 physical page frame - indicating the latter is free or allocated
- Example of frame allocation
  - New process arrives: (a) before allocation (b) after allocation

frame table

free-frame list  
14  
13  
18  
20  
15



(a)

frame table

free-frame list  
15  
page 0  
page 1  
page 2  
page 3  
new process

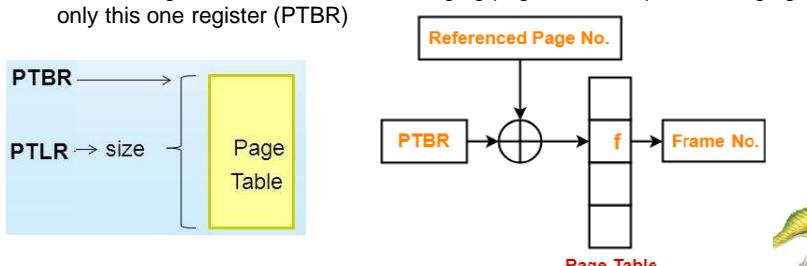
new-process page table  
(b)





## Implementation of Page Table

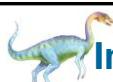
- The hardware implementation of the page table:
  - Use set of dedicated high-speed hardware registers (suitable **small page**)
  - makes the page-address translation very efficient.
  - However, it increases context-switch time, as each one of these registers must be exchanged during a context switch
- **Large page** table: page table is kept in main memory using
  - **Page-table base register (PTBR)** points to the page table
  - **Page-table length register (PTLR)** indicates size of the page table
  - => reducing context-switch time: Changing page tables requires changing only this one register (PTBR)



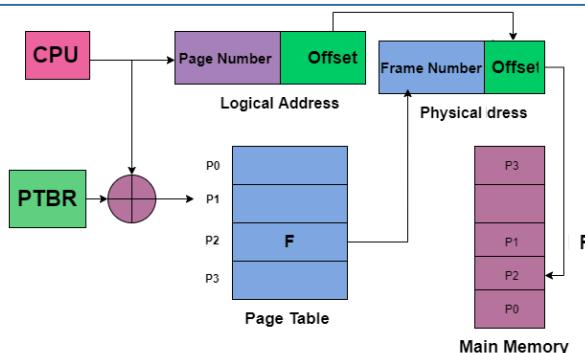
Operating System Concepts – 10<sup>th</sup> Edition

9.46

Silberschatz, Galvin and Gagne ©2018



## Implementation of Page Table using PTBR



- In this scheme every data/instruction access requires two memory accesses
  - 1 for the page table: find the index into the page table by PTBR offset = pagenum
  - 1 for the data / instruction: the frame number & page offset = actual address
  - => Run at half a speed!
- The two-memory access problem can be solved by the use of a special fast-lookup hardware cache called **translation look-aside buffers (TLBs)** (also called **associative memory**).



Operating System Concepts – 10<sup>th</sup> Edition

9.47

Silberschatz, Galvin and Gagne ©2018



## Associative Memory Hardware

- The TLB is associative, high-speed memory.
- Each entry in the TLB consists of two parts: a key (or tag) and a value.
- Associative memory – parallel search: the item is compared with all keys
  - If the item is found, the corresponding value field is returned

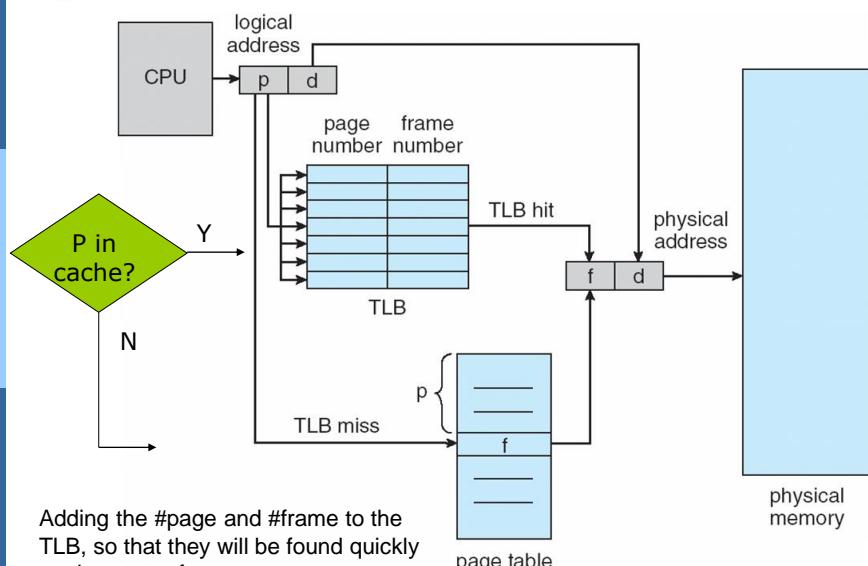
- TLB:

Page #	Frame #

- In paging, TLBs are used to store the most recently accessed memory pages.
  - When the CPU generates an address, the page number of the address is compared with the elements in the TLBs,
  - Ex, Address translation (p, d)
    - If p is in associative register, get frame # out
    - Otherwise get frame # from page table in memory



## Paging Hardware With TLB





## Translation Look-Aside Buffer - TLB

- The TLB contains only a few of the page-table entries, typically 32 -1,024 entries
- When a logical address is generated by the CPU,
  - the MMU checks if its page number is present in the TLB.
  - If the page number is found, its frame number is immediately available and is used to access memory (pipeline within the CPU)
  - If the page number is not in the TLB (**TLB miss**), addr translation proceeds
- **TLB miss:** **update TLB:** value is loaded into the TLB for faster access next time
  - Replacement policies must be considered when TLB is already full of entries:
    - an existing entry must be selected for replacement.
    - Range from least recently used (LRU) through round-robin to random.
  - Some entries can be **wired down** for permanent fast access
    - Means they cannot be removed from the TLB.
    - Typically, TLB entries for key kernel code are wired down.
- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry:
  - identifies each process to provide address-space protection for that process
  - Otherwise need to flush the TLB at every context switch



## Effective Access Time

- If the page is found in the TLB  
$$\text{TLB\_hit\_time} := \text{TLB\_search\_time} + \text{memory\_access\_time}$$
- If the page is not found in the TLB  
$$\text{TLB\_miss\_time} := \text{TLB\_search\_time} + \text{memory\_access\_time (get p&f)} + \text{memory\_access\_time (get data)}$$
- An average measure of the TLB performance: the Effective Access Time
  - $$\text{EAT} := \text{TLB\_miss\_time} * (1 - \text{hit\_ratio}) + \text{TLB\_hit\_time} * \text{hit\_ratio}.$$
  
$$\Leftrightarrow \text{EAT} := (\text{TLB\_search\_time} + 2 * \text{memory\_access\_time}) * (1 - \text{hit\_ratio}) + (\text{TLB\_search\_time} + \text{memory\_access\_time}) * \text{hit\_ratio}.$$
  - **Hit ratio** – percentage of times that a page number is found in the TLB
    - Ex: An 80% hit ratio means that we find the desired page number in the TLB 80% of the time.
- Ex: Suppose that it takes 10 nanoseconds to access memory. 80% hit ratio
  - If we find the desired page in TLB then a mapped-memory access take 10ns
  - Otherwise, we need two memory access, so it is 20ns
  - $$\text{EAT} = 0.80 * 10 + 0.20 * 20 = 12 \text{ ns} \Rightarrow 20\% \text{ slowdown in access time}$$





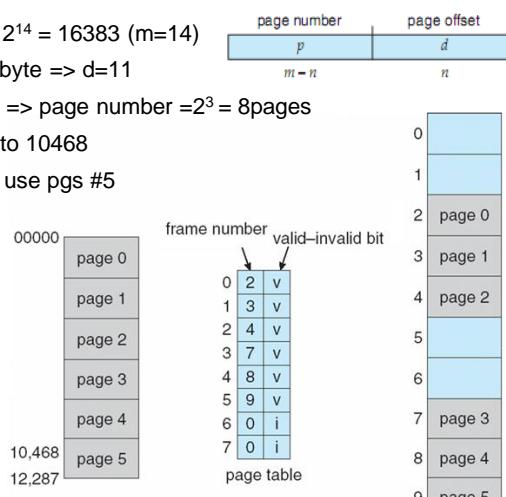
## Memory Protection

- Memory protection implemented by associating protection bit with each frame to indicate if access is allowed
  - **Valid-invalid** bit attached to each entry in the page :
    - “valid” indicates that the associated page is in the process’ logical address space, and is thus a legal page => allow access to the page
    - “invalid” indicates that the page is not in the process’ logical address space => disallow access to the page
    - Or use **page-table length register (PTLR)**
  - 1 entry:
- |      |                      |
|------|----------------------|
| Page | Bit<br>valid/invalid |
|------|----------------------|
- Any violations result in a trap to the kernel
  - Can also add more bits to indicate if read-only, read-write, execute-only is allowed.



## Valid (v) or Invalid (i) Bit In A Page Table

- a system with a 14-bit address space:  $2^{14} = 16383$  ( $m=14$ )
- Given a page size of  $2KB = 2^10 \times 1024 = 2^{11}$  byte =>  $d=11$
- =>  $p = \text{Page number space} = 14-11=3 \Rightarrow \text{page number} = 2^3 = 8\text{pages}$
- Ex, a program need only addresses 0 to 10468
  - $10468 > 2048 \times 5 = 10240 \Rightarrow \text{need use pgs } \#5$



- Any attempt to generate an address in:
  - pages 0, 1, 2, 3, 4, and 5 are mapped normally: **Valid**
  - pages 6 or 7: **invalid** => computer will trap to the OS



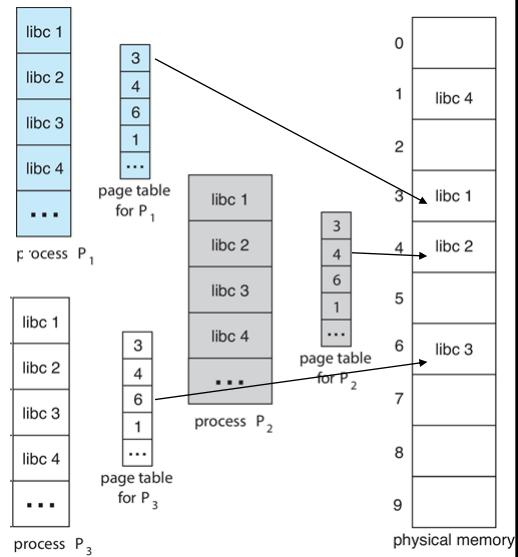
## Shared Pages

- An advantage of paging is the possibility of sharing common code
- **Shared code**
  - One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
  - Similar to multiple threads sharing the same process space
  - Also useful for interprocess communication if sharing of read-write pages is allowed
- **Private code and data**
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space



## Shared Pages Example

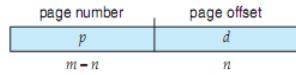
- P<sub>1</sub>,P<sub>2</sub>,P<sub>3</sub> sharing the pages for libc
- Each process has its own copy of registers and data storage
- Only 1 copy of the standard C library need be kept in physical memory,
- The page table for each user process maps onto the same physical copy of libc
- ➔ Saving!!!





## Structure of the Page Table

- Memory structures for paging can get huge using straight-forward methods
  - Modern computer systems support a large 32-bit logical address space
    - $2^{32}$  physical page frames ( $m=32$ )
  - Page size of 4 KB ( $2^{12}$ Byte),  $d=12$
  - => Page table would have 1 million entries:  $2^{20} = 2^{32} / 2^{12}$  ( $p=32-12=20$ )
  - If each entry is 4 bytes ➔ each process requires:  $2^{20} * 4\text{bytes} = 4\text{MB}$  of physical address space for the page table alone – **high cost**
    - Do not want to allocate that contiguously in main memory
  - One simple solution is to divide the page table into smaller units.
    - We can accomplish this division in several ways
- The most common techniques for structuring the page table
  - Hierarchical Paging
  - Hashed Page Tables
  - Inverted Page Tables



## The size of the page table

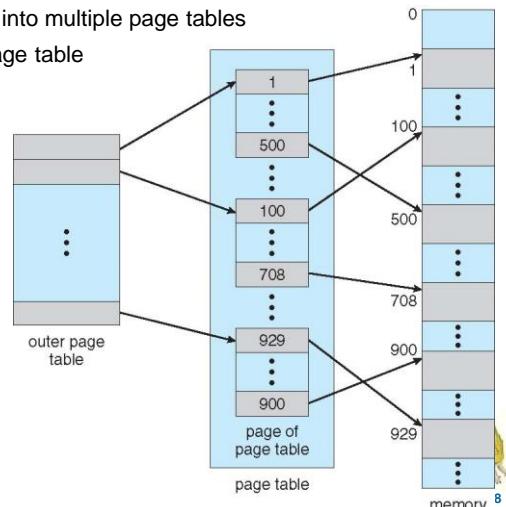
- The size of the page table depends upon the number of entries in the table and the bytes stored in one entry.
  - the number of entries are numbers of pages, ex  $2^{20}$   
(Number of pages is calculated by logical address space and Page size)
  - Give size of a page table entry, ex: 4 Byte
- Many fields in each page table entry
  - Page frame number (virtual addresses)
  - Page number (physical or real address)
  - Present/absent bit (is this page frame currently loaded?)
  - Protection bit (read/write vs. read only)
  - Dirty bit (has the data been modified?)
  - Referenced bit (has this page been recently referenced?)
  - Caching disabled bit (Can this data be cached?)
- Therefore, the size of the page table:  $2^{20} * 4\text{bytes} = 4\text{MB}$   
=> each process requires 4MB of physical address space to store the page table alone (each process use 1 page table for mapping and store it on main mem)





## Hierarchical Page Tables

- Most modern computer systems support a large logical address space ( $2^{32} \rightarrow 2^{64}$ ).
  - =>the page table itself becomes excessively large
- Noncontiguous => divide the page table into **smaller pieces**
- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table



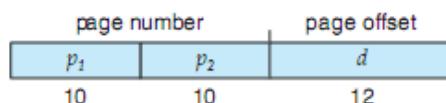
Operating System Concepts – 10<sup>th</sup> Edition

8



## Two-Level Paging Example

- A logical address (on 32-bit machine with 4KB page size) is divided into:
  - A page offset consisting of 12 bits ( $4KB = 2^{12}$  byte =>  $d=12$ )
  - A page number consisting of 20 bits ( $p=32-12$ )
- Since the page table is paged, the page number is further divided into:
  - A 10-bit page number (outer page)
  - A 12-bit page offset
- Thus, a logical address is as follows:



- where  $p_1$  is an index into the outer page table, and  $p_2$  is the displacement (moving) within the page of the inner page table
- Because address translation works from the outer page table inward, this scheme is also known as a **forward-mapped page table**



Operating System Concepts – 10<sup>th</sup> Edition

9.60

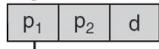
Silberschatz, Galvin and Gagne ©2018



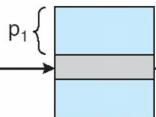
## Address-Translation Scheme

Address translation for a two-level 32-bit paging architecture with 4KB page size

logical address

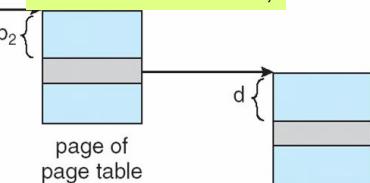


The outer page table consists of  $2^{10}$  entries,

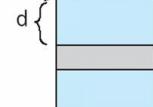


outer page table

The inner page table consists of  $2^{10}$  entries,



page of page table



page table entry

4 bytes per page table entry

The size of first level (outer) page table =  $2^{10} \times 4 \text{ B} = 4\text{KB}$

The size of each second level (inner) page table:  $2^{10} \times 4\text{B} = 4\text{KB}$

Each second level page table addresses  $2^{10}$  entries and so it requires:

**$2^{10}$  second-level page tables**

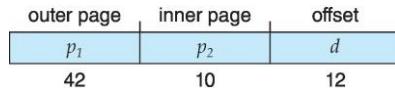
If a process size is 1GB ( $2^{30}\text{B}$ ) need  $2^{30}/2^{12}=2^{18}$  pages use a two-level:  
total memory requirement:  **$4\text{KB} + 4\text{KB} * 2^{18}/2^{10} = 260\text{KB} << 4\text{MB} \rightarrow \text{save cost}$**



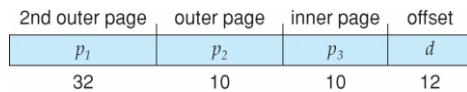
## 64-bit Logical Address Space

- A system with a 64-bit logical address space, a two-level paging scheme is no longer appropriate.
- To illustrate, If page size is 4 KB ( $2^{12}$ ), means  $d=12$ bit space for page\_sz

- Then page table has  $2^{52}$  entries ( $p=64-12=52$ bit for space of #page)
- If two level scheme, inner page tables could be  $2^{10}$  4-byte entries
- Address would look like



- Outer page table has  $2^{42}$  entries or  $2^{44}$  bytes
- One solution is to add a 2<sup>nd</sup> outer page table
- But in the following example the 2<sup>nd</sup> outer page table is still  $2^{34}$  bytes (16GB)
  - And possibly 4 memory access to get to one physical memory location
- Three-level Paging Scheme





## Hashed Page Tables

- Used in architecture with address spaces > 32 bits
- The virtual page number is **hashed into a page table**
  - Each entry in the hash table has a **linked list** of elements hashed to the same location (to avoid collisions – as we can get the same value of a hash function for different page numbers).
  - The hash value is virtual page number - is all the bits that are not a part of the page offset
- Each element contains
  1. The virtual page number
  2. The value of the mapped page frame
  3. A pointer to the next element
- Virtual page numbers are compared in this linked list searching for a match
  - If a match is found, the corresponding physical frame is extracted
  - Otherwise, subsequent entries in the linked list are checked until the virtual page number matches.



## Expl: Hash Table Chaining

- The idea is to make each cell of hash table point to a linked list of records that have same hash function value.

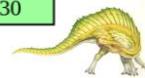
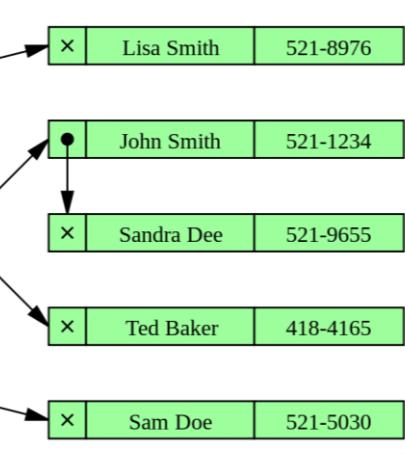
keys

John Smith  
Lisa Smith  
Sam Doe  
Sandra Dee  
Ted Baker

buckets

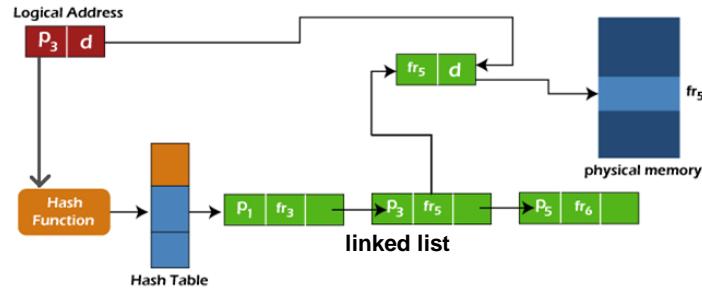
000	x
001	●
002	x
:	:
151	x
152	●
153	●
154	x
:	:
253	x
254	●
255	x

entries





## Hashed Page Table Hardware

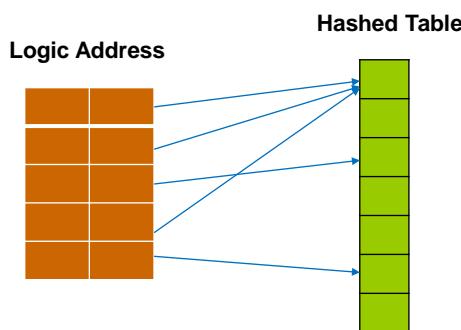


- The virtual page number in the virtual address is hashed into the hash table.
- The virtual page number ( $p_3$ ) is compared with field 1 ( $p_1$ ) in the **first** element in the linked list
  - does not match the first element of the link list
  - move ahead and check the next element ( $P_3$ ): match
  - check the frame entry of the element, which is  $fr_5$ .
  - append the offset provided ( $d$ ) in the logical address to this frame number to reach the page's physical address.



## Clustered Hashed Page Tables

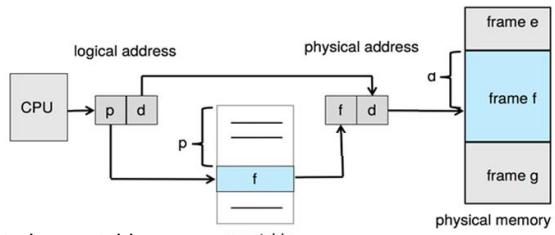
- Variation for 64-bit addresses uses clustered page tables - similar to hashed:
  - Hashed table: each entry refers to a single page-table entry
  - Clustered Hashed: each entry refers to **several** pages (such as 16), can store the mappings for **multiple physical-page frames**.
  - Especially useful for **sparse** address spaces (where memory references are non-contiguous and scattered)





## Inverted Page Table

- Normally



- each process has an associated page table
- each page table may consist of millions of entries.  
=> consume large amounts of physical memory just to keep track of how other physical memory is being used

- Solution: Rather than having each process keep a page table and track of all possible logical pages, track all physical pages => **Inverted Page Table:**

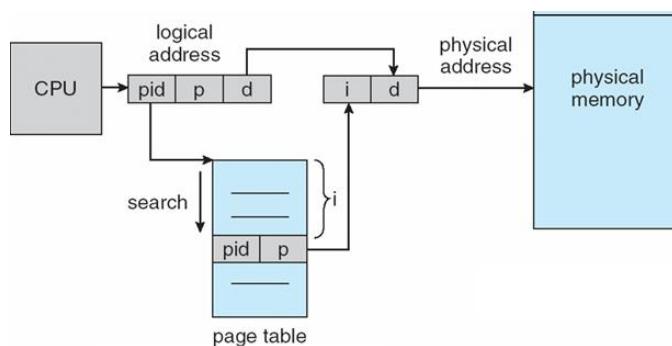
- Use only 1 IPT for all processes – 1 entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Each inverted page-table entry is a pair <process-id, page-number>
  - ▶ process-id: role of the address-space identifier.



## Inverted Page Table

- When a memory reference occurs

- part of the virtual address, consisting of <**process-id, page-number**>, is presented to the memory subsystem.
- The inverted page table is then searched for a match.
  - ▶ If a match is found at entry i then the physical address <i, offset> is generated.
  - ▶ If no match is found, then an illegal address access has been attempted.





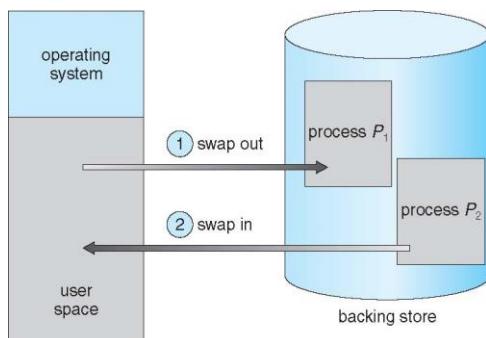
## Inverted Page Table (Cont.)

- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one (or at most a few) page-table entries
  - TLB can accelerate access (TLB is searched before the hash table is consulted)
- Implement shared memory in inverted page tables?
  - One mapping of a virtual address to the shared physical address
  - A reference by another process sharing the memory will result in a page fault and will replace the mapping with a different virtual address



## Swapping

- Process instructions and their data must be in memory to be executed.
- However, a process can be **swapped** temporarily out of memory to a backing store, and then brought **back** into memory for continued execution
  - increasing the degree of multiprogramming in a system
- **Backing store** is commonly fast secondary storage.
  - It must be large enough to accommodate whatever parts of processes need to be stored and retrieved, and it must provide direct access to these memory images.



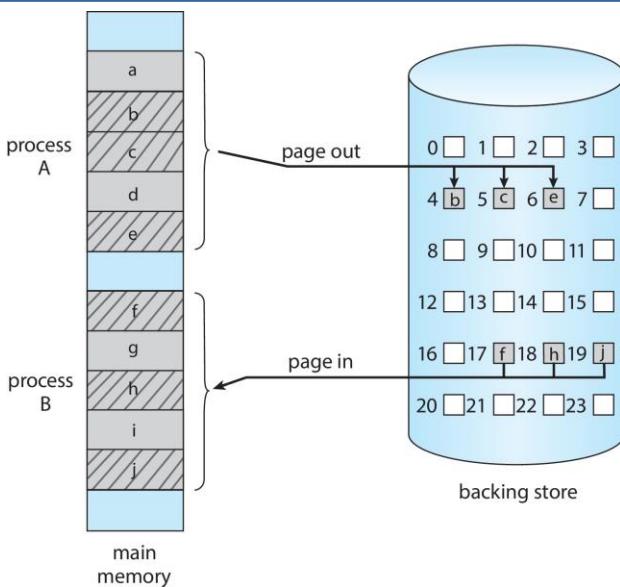


## Swapping

- Standard swapping is generally no longer used in contemporary OSs,
  - because the amount of time required to move entire processes between memory and the backing store is prohibitive.
  - Context switch time can then be very high
- a variation of swapping in which pages of a process can be swapped (no: an entire process)
  - still allows physical memory to be oversubscribed, but does not incur the cost of swapping entire processes, (only a small number of pages will be involved in swapping).
  - A **page out** operation moves a page from memory to the backing store; the reverse process is known as a **page in**.
  - swapping with paging works well in conjunction with virtual memory.



## Swapping with Paging





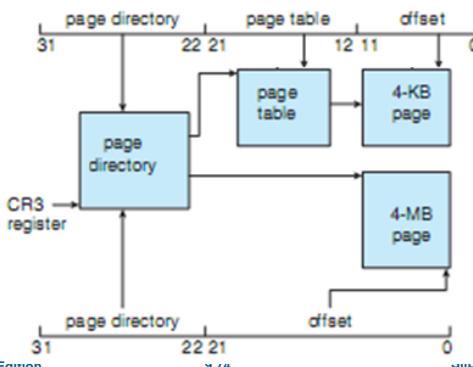
## Swapping on Mobile Systems

- Not typically supported -- Flash memory based
  - Small amount of space
  - Limited number of write cycles
  - Poor throughput between flash memory and CPU on mobile platform
- Instead use other methods to free memory if low
  - iOS **asks** apps to voluntarily relinquish allocated memory
    - Read-only data thrown out and reloaded from flash if needed
    - Failure to free can result in termination
  - Android terminates apps if low free memory, but first writes **application state** to flash for fast restart
  - Both OSes support paging as discussed below



## Example: Intel 32- and 64-bit Architectures

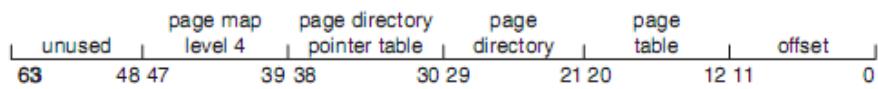
- Paging in the IA-32 architecture.
  - Address translation two-level (4KB)
  - One entry in the page directory is the Page Size flag, if set—indicates that the size of the page frame is 4 MB and not the standard 4 KB
    - the page directory points directly to the 4-MB page frame, bypassing the inner page table; and the 22 low-order bits in the linear address refer to the offset in the 4-MB page frame.





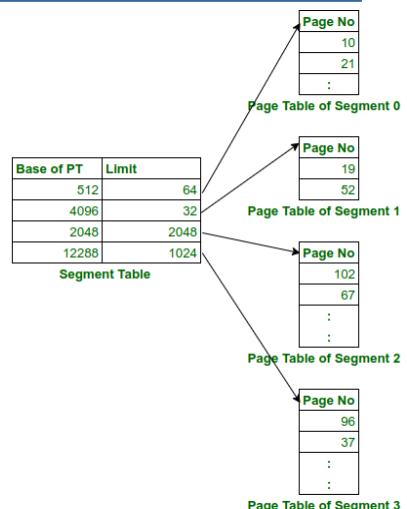
## Example: Intel 32- and 64-bit Architectures

- Intel 64-bit Architectures based on the AMD's x86-64 architecture.
  - The x86-64 supported much larger logical and physical address spaces, as well as several other architectural advances.
  - Support for a 64-bit address space =  $2^{64}$  bytes of addressable memory—a number greater than 16 quintillion (or 16 exabytes).
  - In practice far fewer than 64 bits are used for address representation in current designs.
  - The x86-64 architecture currently provides a 48-bit virtual address with support for page sizes of 4 KB, 2 MB, or 1 GB using 4 levels of paging hierarchy.
- x86-64 linear address.



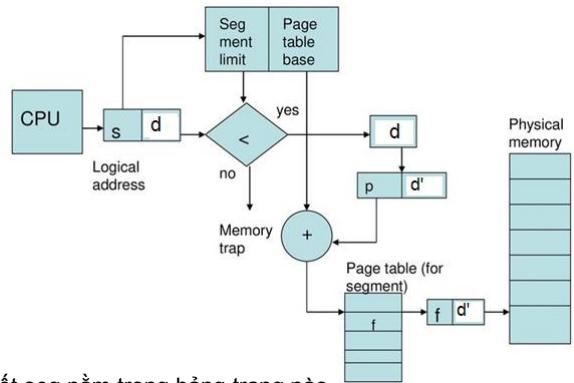
## Segmented Paging

- use segmentation along with paging to reduce the size of page table.
  - creating a page table for each segment.
  - hardware support is required.
  - The address provided by CPU will now be partitioned into segment no., page no. and offset.
- MMU will use the segment table which will contain the address of page table(base) and limit.
- The page table will point to the page frames of the segments in main memory.





## Segmented Paging Scheme

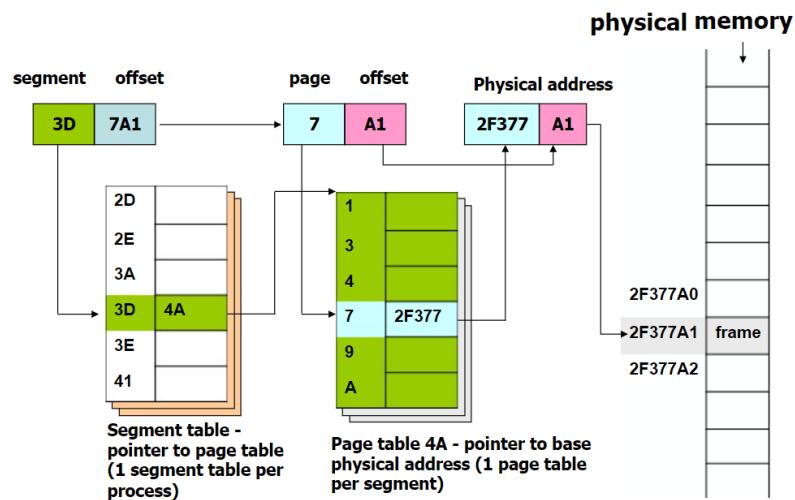


s: segment number  
 d: logical offset in segment  
 p: page  
 d': offset in page  
 f: frame

- Xét đ/chỉ logic: (s,d)
  - tra **s** trong bảng seg để biết seg nằm trong bảng trang nào.
  - Offset **d** cho biết:
    - Số trang **p** trong bảng trang (đã dc xđinh từ s trong bảng seg);
    - Offset **d'** tương ứng với trang **p** trong bảng trang
  - Kết hợp offset **d'** (tách từ d), và frame **f** tại trang **p** trong bảng trang
  - => đ/chỉ vật lý (**f, d'**)



## Segmented Paging, ex





## Segmented Paging

- Advantages

- Reduces external fragmentation
- The page table size is reduced as pages are present only for data of segments, hence reducing the memory requirements.
- the swapping out into virtual memory becomes easier .

- Disadvantages

- Internal fragmentation still exists in pages.
- Extra hardware is required
- increasing the memory access time.
- External fragmentation occurs

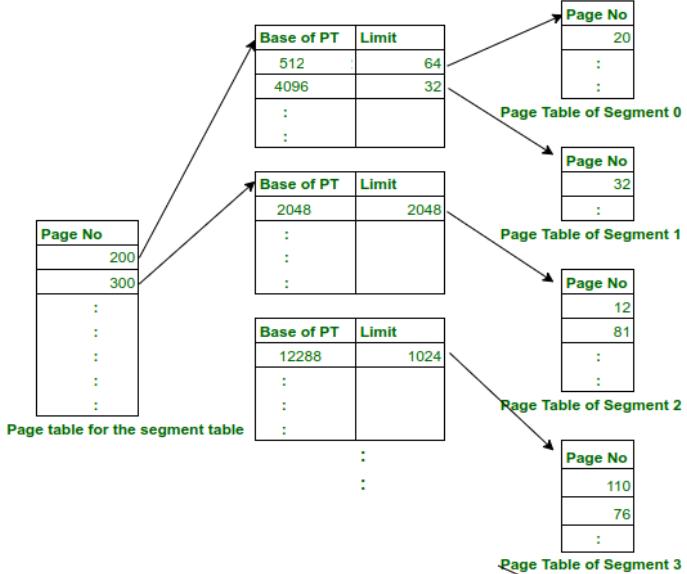


## Paged Segmentation

- the segment table to be paged

- logical address

- page no #1,
- segment no,
- page no #2
- offset



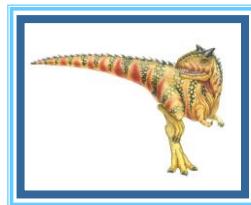


## Paged Segmentation

- Advantages of Paged Segmentation
  - No external fragmentation
  - Reduced memory requirements as no. of pages limited to segment size.
  - Page table size is smaller just like segmented paging,
  - Similar to segmented paging, the entire segment need not be swapped out.
- Disadvantages of Paged Segmentation
  - Internal fragmentation remains a problem.
  - Hardware is complexer than segmented paging.
  - Extra level of paging at first stage adds to the delay in memory access.



## End of Chapter 3.1



# Chapter 3: Memory Management

## 3.2. Virtual Memory



GV: Nguyễn Thị Thanh Vân

Operating System Concepts – 10<sup>th</sup> Edition

Silberschatz, Galvin and Gagne ©2018



## Outline

- Background
- Demand Paging
- Page Replacement
- Allocation of Frames
- Thrashing
- Other Considerations



Operating System Concepts – 10<sup>th</sup> Edition

10.2

Silberschatz, Galvin and Gagne ©2018



## Objectives

- Define virtual memory and describe its benefits.
- Illustrate how pages are loaded into memory using demand paging.
- Apply the FIFO, optimal, and LRU page-replacement algorithms.
- Describe the working set of a process and explain how it is related to program locality.
- Describe how Linux, Windows 10, and Solaris manage virtual memory.
- Design a virtual memory manager simulation in the C programming language.



## Background

- Code needs to be in memory to execute, but entire program rarely used
  - Error code, unusual routines, large data structures
- Entire program code not needed at the same time
- Consider ability to execute partially-loaded program
  - Program no longer constrained by limits of physical memory
  - Each program takes less memory while running -> more programs run at the same time
    - ▶ Increased CPU utilization and throughput with no increase in response time or turnaround time
  - Less I/O needed to load or swap programs into memory -> each user program runs faster



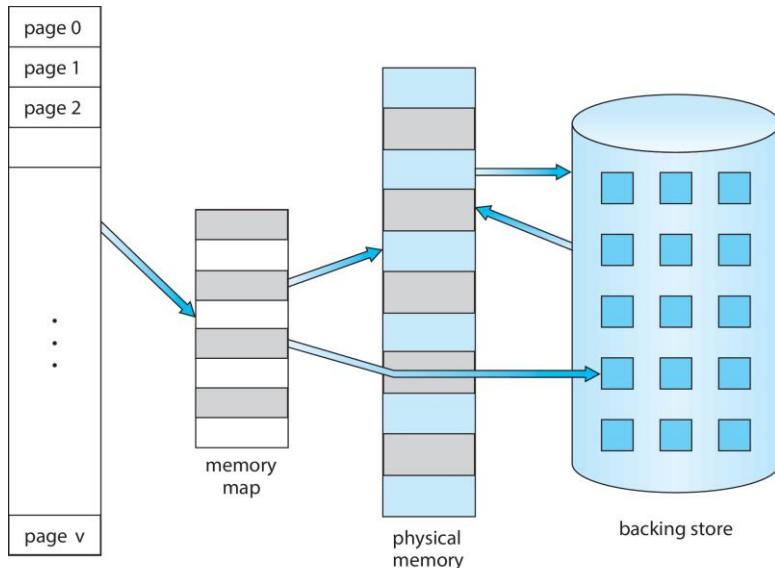


## Virtual memory

- **Virtual memory** – separation of user logical memory from physical memory
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
  - More programs running concurrently
  - Less I/O needed to load or swap processes
- **Virtual address space** – logical view of how process is stored in memory
  - Usually start at address 0, contiguous addresses until end of space
  - Meanwhile, physical memory organized in page frames
  - MMU must map logical to physical
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation



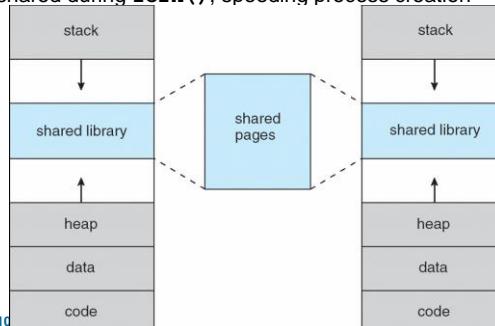
## Virtual Memory That is Larger Than Physical Memory





## Virtual-address Space

- Usually design logical address space for the stack to start at Max logical address and grow "down" while heap grows "up"
  - Maximizes address space use
  - Unused address space between the two is hole
    - No physical memory needed until heap or stack grows to a given new page
- Enables **sparse** address spaces with holes left for growth, dynamically linked libraries...
- System libraries shared via mapping into virtual address space
- Shared memory by mapping pages read-write into virtual address space
- Pages can be shared during `fork()`, speeding process creation



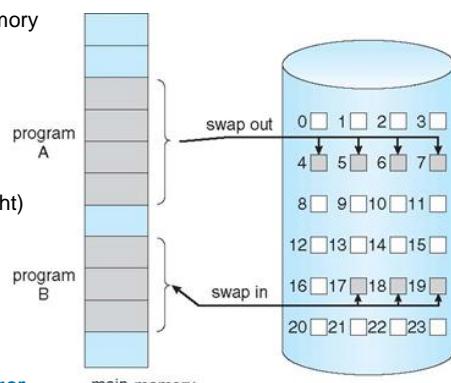
Operating System Concepts - 10<sup>th</sup> Edition

Silberschatz, Galvin and Gagne ©2018



## Demand Paging

- Instead of bringing the entire program into memory at load time, bring a page into memory only when it is needed
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
- Similar to paging system with swapping (right)
- Page reference
  - Invalid reference  $\Rightarrow$  abort
  - Not-in-memory  $\Rightarrow$  bring to memory
- Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**



Operating System Concepts - 10<sup>th</sup> Edition

10.8

Silberschatz, Galvin and Gagne ©2018

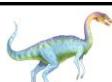


## Basic Concepts

- With swapping, the pager guesses which pages will be used before swapping them out again
  - How to determine that set of pages?
- Need new MMU functionality to implement demand paging
- If pages needed are already **memory resident**
  - No difference from non-demand paging
- If page needed and not memory resident
  - Need to detect and load the page into memory from storage
    - ▶ Without changing program behavior and needing to change code
- Use page table with valid-invalid bit:
  - (**v** ⇒ in-memory, **i** ⇒ not-in-memory)
  - Initially valid–invalid bit is set to **i** on all entries
  - During MMU address translation,
    - ▶ if valid–invalid bit in the page table entry is **i**
      - ▶ ⇒ page fault

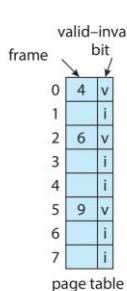
Frame #	valid-invalid bit
	v
	v
	v
	i
...	
	i
	i

page table



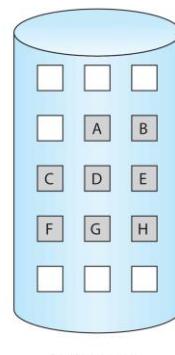
## Page Table When Some Pages Are Not in Main Memory

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H



0	
1	
2	
3	
4	A
5	
6	C
7	
8	
9	F
10	
11	
12	
13	
14	
15	

physical memory



backing store



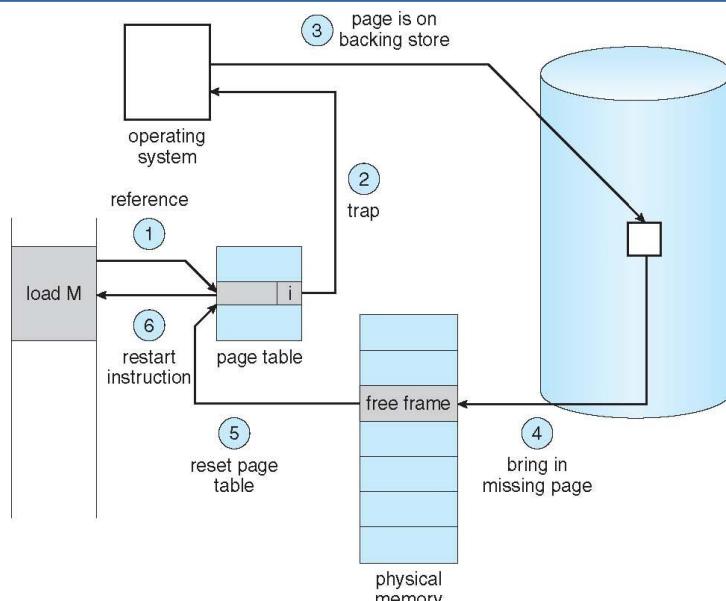


## Steps in Handling Page Fault

1. The first reference to a page will trap to operating system.
  1. Page fault
2. Operating system looks at another table to decide:
  - Invalid reference  $\Rightarrow$  abort
  - Just not in memory (go to step 3)
3. Find free frame (what if there is none?)
4. Swap page into frame via scheduled disk operation
5. Reset tables to indicate page now in memory.
  1. Set validation bit = v
6. Restart the instruction that caused the page fault



## Steps in Handling a Page Fault (Cont.)





## Performance of Demand Paging

- Three major activities
  - Service the interrupt – careful coding means just several hundred instructions needed
  - Input the page from disk – lots of time
  - Restart the process – again just a small amount of time
- Page Fault Rate  $0 \leq p \leq 1$ 
  - If  $p = 0$  no page faults
  - If  $p = 1$ , every reference is a fault
- Effective Access Time (EAT)
$$\text{EAT} = (1 - p) \times \text{memory access} + p (\text{page fault overhead} + \text{swap page out} + \text{swap page in})$$



## Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- $$\begin{aligned}\text{EAT} &= (1 - p) \times 200 + p (8 \text{ milliseconds}) \\ &= (1 - p) \times 200 + p \times 8,000,000 \\ &= 200 + p \times 7,999,800\end{aligned}$$
- If one access out of 1,000 causes a page fault, then
$$\text{EAT} = 8.2 \text{ microseconds.}$$
This is a slowdown by a factor of 40!!
- If want performance degradation < 10 percent
  - $$\begin{aligned}220 &> 200 + 7,999,800 \times p \\ 20 &> 7,999,800 \times p\end{aligned}$$
  - $p < .0000025$ 
    - ▶ one page fault in every 400,000 memory accesses





## Demand Paging Optimizations

- Swap space I/O faster than file system I/O even if on the same device
  - Swap allocated in larger chunks; less management needed than file system
- Copy entire process image to swap space at process load time
  - Then page in and out of swap space
  - Used in older BSD Unix
- Demand page in from program binary on disk, but discard rather than paging out when freeing frame
  - Used in Solaris and current BSD
  - Still need to write to swap space
    - ▶ Pages not associated with a file (like stack and heap) – **anonymous memory**
    - ▶ Pages modified in memory but not yet written back to the file system
- Mobile systems
  - Typically, don't support swapping
  - Instead, demand page from file system and reclaim read-only pages (such as code)



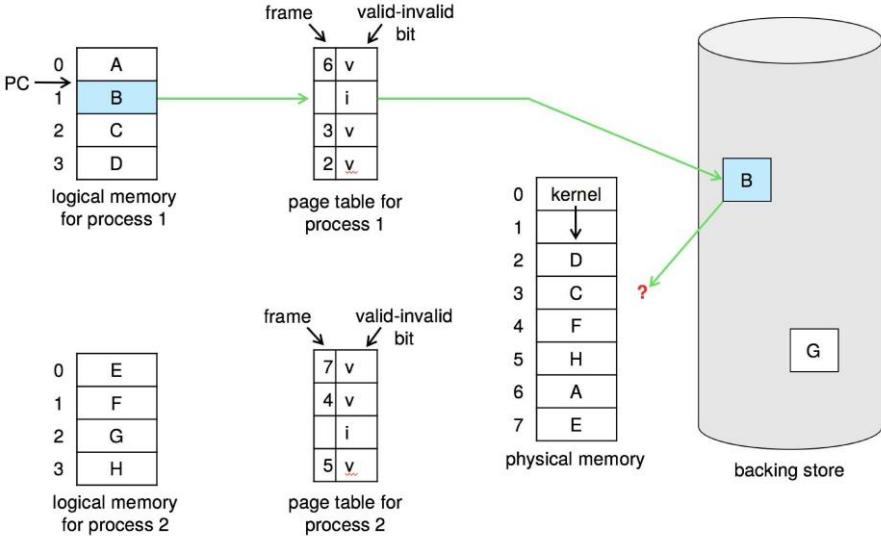
## What Happens if There is no Free Frame?

- Page replacement – find some page in memory, but not really in use, page it out
  - Algorithm – terminate? swap out? replace the page?
  - Performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times
- Page replacement
  - Prevent **over-allocation** of memory by modifying page-fault service routine to include page replacement
  - Use **modify (dirty) bit** to reduce overhead of page transfers – only modified pages are written to disk
  - completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory





## Need For Page Replacement

Operating System Concepts – 10<sup>th</sup> Edition

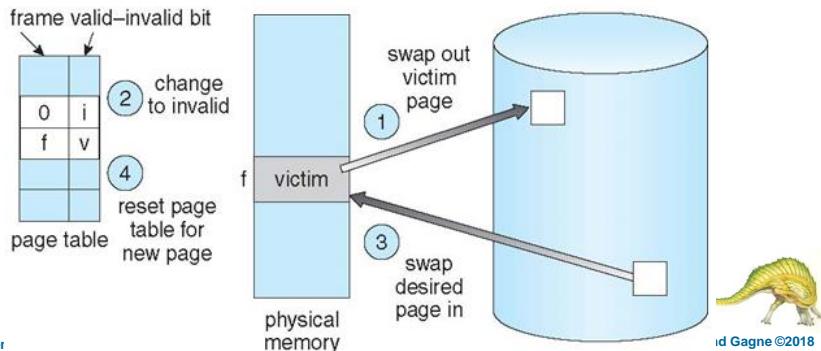
10.17

Silberschatz, Galvin and Gagne ©2018



## Basic Page Replacement

1. Find the location of the desired page on disk
  2. Find a free frame:
    - If there is a free frame, use it
    - If there is no free frame, use a page repl algorithm to select a **victim frame**
    - Write victim frame to disk if dirty
  3. Bring the desired page into the (newly) free frame; update the page & frame tables
  4. Continue the process by restarting the instruction that caused the trap
- Note now potentially 2 page transfers for page fault – increasing EAT



Operating System

Galvin and Gagne ©2018



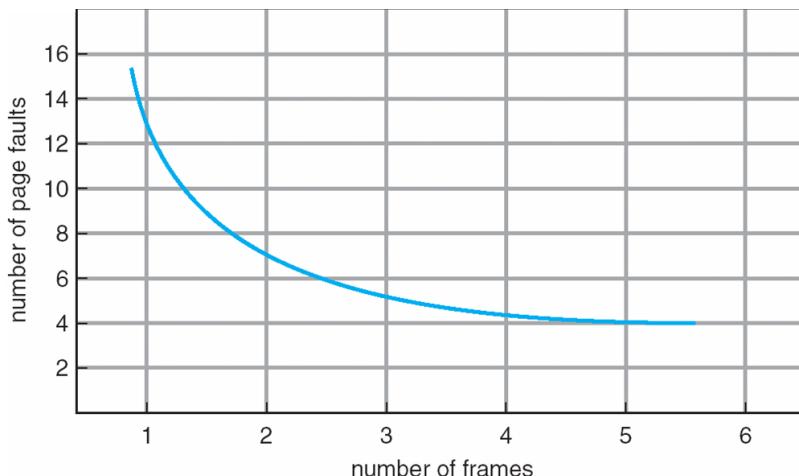
## Page and Frame Replacement Algorithms

- **Frame-allocation algorithm** determines
  - How many frames to give each process
  - Which frames to replace
- **Page-replacement algorithm**
  - Want lowest page-fault rate on both first access and re-access
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
  - String is just page numbers, not full addresses
  - Repeated access to the same page does not cause a page fault
  - Results depend on number of frames available
- In all our examples, the **reference string** of referenced page numbers is

**7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**



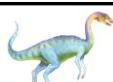
## Graph of Page Faults Versus the Number of Frames





## Replacement Algorithms

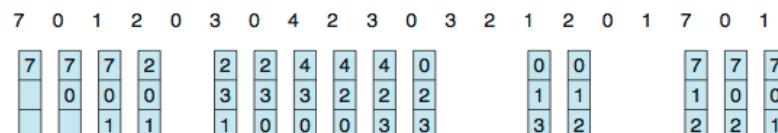
- First-In-First-Out (FIFO)
- Optimal Algorithm
- Least Recently Used (LRU)
- LRU Approximation
  - Additional-Reference-Bits Algorithm
  - Enhanced Second-Chance Algorithm
  - Second-Chance Algorithm
- Counting Algorithms
- Page-Fault Frequency Algorithm



## First-In-First-Out (FIFO) Algorithm

- Replace page that **FIFO**
- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)

reference string



page frames

15 page faults

- How to track ages of pages?
  - Just use a FIFO queue



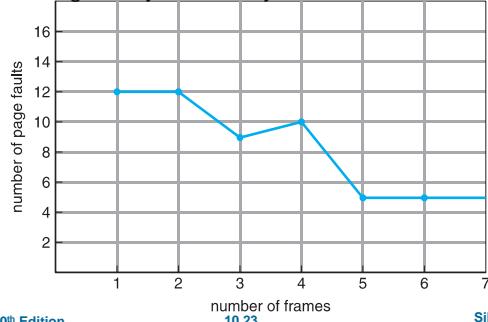


## Belady's Anomaly

- Consider the string 1,2,3,4,1,2,5,1,2,3,4,5
  - Adding more frames can cause more page faults!

1	2	3	4	<b>1</b>	2	5	1	2	3	4	5
1	1	1	4	4	4	5	5	5	5	5	5
2	2	2	<b>1</b>	1	1	1	1	1	3	3	3
	3	3	3	2	2	2	2	2	4	4	
*	*	*	*	*	*	*	*	*	*	*	

- Graph illustrating Belady's Anomaly



Operating System Concepts – 10<sup>th</sup> Edition

Silberschatz, Galvin and Gagne ©2018



## Optimal Algorithm

- Replace page that **will not be used for longest period of time**
- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
  - Optimal algorithm
- reference string
  - 7 0 1 2 0 3 0 4 2 3 0 3 0 3 2 1 2 0 1 7 0 1
  - page frames
    - 7 7 7 2 2 2 4 2 3 2 0 2 0 2 7 0 1
  - page faults
    - 9 page faults
- How do you implement this?
  - Can't read the future
- Used for measuring how well your algorithm performs
- Optimal is an example of **stack algorithms** that don't suffer from Belady's Anomaly

Operating System Concepts – 10<sup>th</sup> Edition

10.24

Silberschatz, Galvin and Gagne ©2018

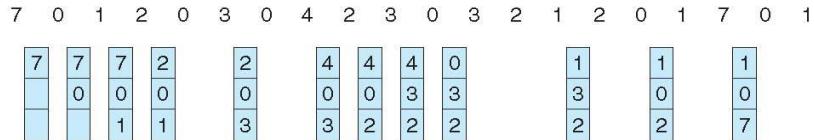




## Least Recently Used (LRU) Algorithm

- Use past knowledge rather than future
- Replace page that has not been used in the most amount of time
- Associate time of last use with each page

reference string



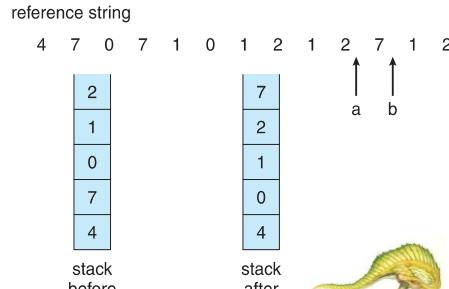
page frames

- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- LRU is another example of stack algorithms; thus, it does not suffer from Belady's Anomaly



## LRU Algorithm Implementation

- Time-counter implementation
  - Every page entry has a time-counter variable; every time a page is referenced through this entry, copy the value of the clock into the time-counter
  - When a page needs to be changed, look at the time-counters to find smallest value
    - Search through a table is needed
- Stack implementation
  - Keep a stack of page numbers in a double link form:
  - Page referenced:
    - Move it to the top
    - Requires 6 pointers to be changed
  - But each update more expensive
  - No search for replacement
- Use of a stack to record most recent page references





## LRU Approximation Algorithms

- Needs special hardware
  - **Reference bit**
    - With each page associate a bit, initially = 0 (by OS)
    - When page is referenced (a process executes), bit set to 1 (by HW)
  - 1 entry in page table
- | Page# | Bit valid /invalid | Dirty bit | Reference bit |
|-------|--------------------|-----------|---------------|
|-------|--------------------|-----------|---------------|
- After some time, we can determine which pages have been used and which have not been used by examining the reference bits
    - although we do not know the order of use
  - When a page needs to be replaced, replace any with reference bit = 0 (if exists)
  - To know the order of page, LRU appr use:
    - Additional-Reference-Bits Algorithm
    - Enhanced Second-Chance Algorithm
    - Second-Chance Algorithm



## LRU Approximation Algorithms

- **Additional-Reference-Bits Algorithm**
  - keep an 8-bit byte for each page in a table - history of page use for the last eight time periods.
  - At regular intervals (every 100 ms), a timer interrupt transfers control OS
  - HR=00000000: the page has not been used for 8 time periods
  - HR=11111111: A page that is used at least once in each period
  - Ex: A page with a HR1 has been used more recently than HR3

	0	0	1	0	0	0	1	1	1	0
HR =11000100										
HR =11100010										
HR =01110001										





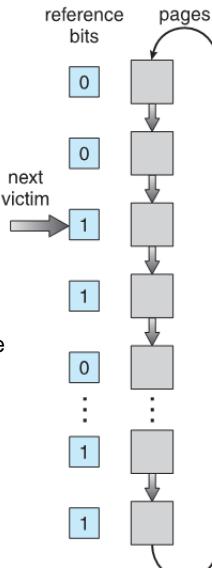
## LRU Approximation Algorithms

- **Second-chance algorithm**

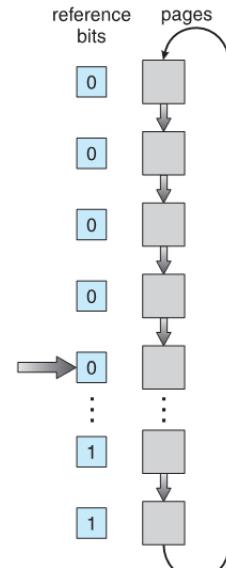
- Generally, **FIFO**, plus hardware-provided reference bit

- **Clock** replacement (circular queue)

- If page to be replaced has
  - ▶ Reference bit = 0 -> replace it
  - ▶ Reference bit = 1 then:
    - Set reference bit 0, leave page in memory
    - Replace next page (victim), subject to same rules



(a) circular queue of pages



(b) circular queue of pages

Operating System Concepts – 10<sup>th</sup> Edition

## LRU Approximation Algorithms

- **Second-chance algorithm**

- Reference string: **7,0,1,2,0,3,0,4,2,4,0,3,2**

	7	0	1		2	0		3	0		4	2	4	0		3	2
fr1	7	7	7	7	2	2	2	2	2	2	4	4	4	4	4	3	3
	(1)	(1)	(1)	(0)	(1)	(1)	(1)	(1)	(1)	(0)	(1)	(1)	(1)	(1)	(0)	(1)	(1)
fr2	(0)	0	0	0	0	0	0	0	0	0	2	2	2	2	2	2	2
	(1)	(1)	(1)	(0)	(0)	(1)	(0)	(0)	(1)	(0)	(1)	(1)	(1)	(1)	(0)	(1)	(1)
fr3		(0)	1	1	1	1	1	3	3	3	3	3	3	0	0	0	0
		(1)	(0)	(0)	(0)	(0)	(0)	(1)	(1)	(0)	(0)	(0)	(0)	(1)	(0)	(0)	(0)
	x	x	x		x			x			x	x		x		x	

Operating System Concepts – 10<sup>th</sup> Edition

10.30

Silberschatz, Galvin and Gagne ©2018



## LRU Approximation Algorithms

- Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bit (if available) in concert
- Take ordered pair (reference, modify):
  - (0, 0) neither recently used nor modified – best page to replace
  - (0, 1) not recently used but modified – not quite as good, must write out before replacement
  - (1, 0) recently used but clean – probably will be used again soon
  - (1, 1) recently used and modified – probably will be used again soon and need to write out before replacement

- When page replacement called for, use the clock scheme but use the four classes replace page in lowest non-empty class

- Might need to search circular queue several times



## Counting Algorithms

- Keep a counter of the number of references that have been made to each page

- Not common

- Lease Frequently Used (LFU) Algorithm:

- Replaces page with **smallest count**

1	2	3	4	2	1	5	6	2	1
1	1	1	1			1	1		
2	2	2				2	2		
3	3					6	5		
	4					4	6		

- pg5: f1=2, f2=2, f3=1, f4=1

- Pg3 came first => remove, f3=0

- Add pg5, f5=1

- pg6: f1=2, f2=2, f5=1, f4=1:

- Pg4 came first => remove pg4, f4=0

- Add pg6, f6=1

- Most Frequently Used (MFU) Algorithm:

1	2	3	4	2	1	5	6	2	1
1	1	1	1			5	5	5	5
2	2	2				2	6	6	6
3	3					3	3	2	1
	4					4	4	4	4

- pg5: f1=2, f2=2, f3=1, f4=1

- Remove, f1=1. Add pg5, f5=1

- pg6: f5=1, f2=2, f3=1, f4=1:

- Remove pg2, f2=1. Add pg6, f6=1

- pg2: f5=1, f6=1, f3=1, f4=1:

- Remove pg3, f3=0. Add pg2, f2=2

- pg1: f5=1, f6=1, f2=2, f4=1:

- Remove pg2, f2=1. Add pg1, f1=1



## Allocation of Frames

- How do we allocate the fixed amount of free memory among the various processes
  - 93 free frames and 2 processes, ?frames/process
  - => free-frame list was exhausted
- Strategy: Processs need minimum Number of Frames
  - is defined by the computer architecture
  - as the number of frames allocated to each process decreases, the page-fault rate increases, slowing process execution.
  - when a page fault occurs before an executing instruction is complete, the instruction must be restarted.
  - Have enough frames to hold all the different pages
  - **Maximum** of course is total frames in the system
- Two major allocation schemes
  - Fixed allocation
  - Priority allocation
- Many variations



## Fixed Allocation

- **Equal allocation** – all processes gets the same number of frames.
  - For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames
  - Keep some as free frame buffer pool
- **Proportional allocation** – Allocate according to the size of process
  - Dynamic as degree of multiprogramming, process sizes change
  - Example

$$s_i = \text{size of process } p_i$$

$$m = 64$$

$$S = \sum s_i$$

$$s_1 = 10$$

$$m = \text{total number of frames}$$

$$s_2 = 127$$

$$a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$$

$$a_1 = \frac{10}{137} \cdot 62 \gg 4$$

$$a_2 = \frac{127}{137} \cdot 62 \gg 57$$





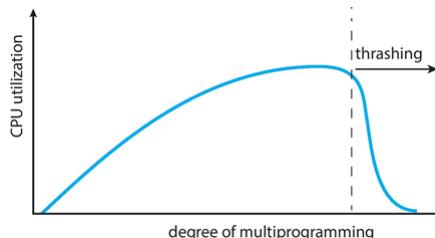
## Global vs. Local Allocation

- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
  - Process execution time can vary greatly
  - Greater throughput so more commonly used
- **Local replacement** – each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - But possibly underutilized memory
  - What if a process does not have enough frames?



## Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need the replaced frame back
- This leads to:
  - Low CPU utilization
  - OS thinking that it needs to increase the degree of multiprogramming
  - Another process added to the system
  - Which results in even higher page fault rate
- **Thrashing**. A process is busy swapping pages in and out





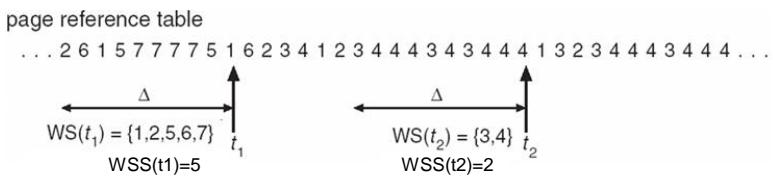
## Preventing thrashing

- we must provide a process with as many frames as it needs.
  - But how do we know how many frames it “needs”?
- One strategy starts by looking at howmany frames a process is actually using.
- => **Locality model**
  - Process migrates from one locality to another
  - Localities may overlap
- Why does thrashing occur?
  - Σ size of locality > total memory size
- To avoid thrashing:
  - Calculate the Σ size of locality
  - Policy: if Σ size of locality > total memory size
    - ▶ suspend or swap out one of the processes
- Issue: how to calculate “Σ size of locality”



## Working-Set Model

- $\Delta$  = working-set window = a fixed number of page references, Ex: 10,000 instructions
- $WSS_i$  (working set of Process  $P_i$ ) = total number of pages referenced in the most recent  $\Delta$  (varies in time)
- Example: if  $\Delta=10$



- Observation
  - if  $\Delta$  too small will not encompass the entire locality
  - if  $\Delta$  too large will encompass several localities
  - if  $\Delta = \infty \Rightarrow$  will encompass entire program
- $D = \sum WSS_i$  = total demand frames (process i request WSSi)
  - Approximation of locality
  - Let  $m$  = total number of frames
  - If  $D > m \Rightarrow$  Thrashing
  - Policy if  $D > m$ , then suspend or swap out one of the processes





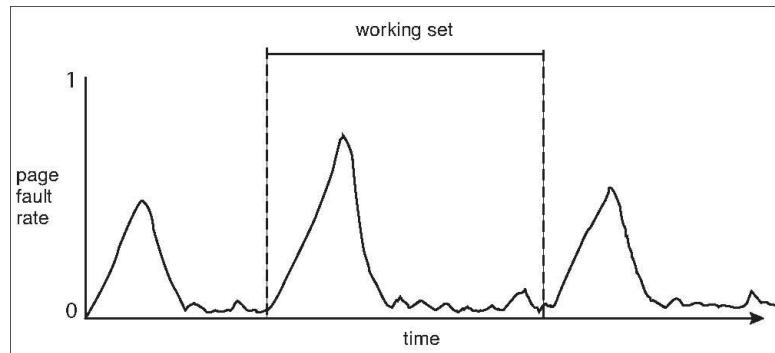
## Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example:  $\Delta = 10,000$ 
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page  $i$ 
    - $B1_i$  and  $B2_i$
  - Whenever a timer interrupts copy the reference to one of the  $B_j$  and sets the values of all reference bits to 0
  - If either  $B1_i$  or  $B2_i = 1$ , it implies that Page  $i$  is in the working set
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units



## Working Sets and Page Fault Rates

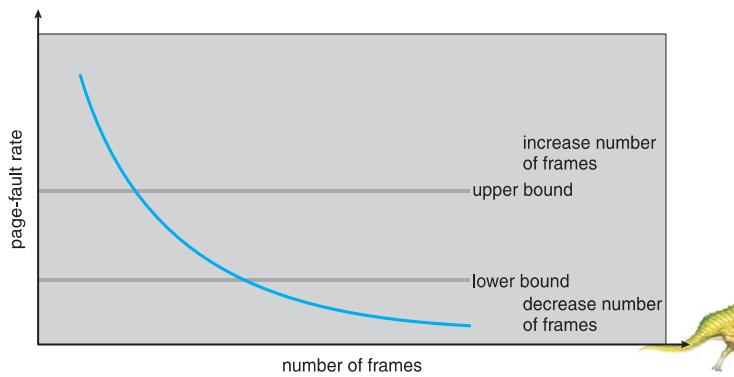
- Direct relationship between working set of a process and its page-fault rate
- Working set changes over time
- Peaks and valleys over time





## Page-Fault Frequency Algorithm

- More direct approach than WSS
- Establish “acceptable” **page-fault frequency (PFF)** rate and use local replacement policy
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame
- Example

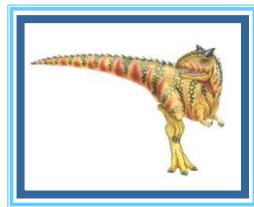


## Other Considerations

- Prepaging
- Page size
- TLB reach
- Inverted page table
- Program structure
- I/O interlock and page locking



## End of Chapter 3.2



## Chapter 4: File System Management and I/O System

### 4.1. Mass-Storage Systems



GV: Nguyễn Thị Thanh Vân

Operating System Concepts – 10<sup>th</sup> Edition

Silberschatz, Galvin and Gagne ©2018



### Mass-Storage Systems

- Overview of Mass Storage Structure
- HDD Scheduling
- NVM Scheduling
- Error Detection and Correction
- Storage Device Management
- Swap-Space Management
- Storage Attachment
- RAID Structure



Operating System Concepts – 10<sup>th</sup> Edition

11.2

Silberschatz, Galvin and Gagne ©2018



## Objectives

- Describe the physical structure of secondary storage devices and the effect of a device's structure on its uses
- Explain the performance characteristics of mass-storage devices
- Evaluate I/O scheduling algorithms
- Discuss operating-system services provided for mass storage, including RAID



## Overview of Mass Storage Structure

- Bulk of secondary storage for modern computers is **hard disk drives (HDDs)** and **nonvolatile memory (NVM)** devices
- **HDDs** spin platters of magnetically-coated material under moving r/w heads
  - The two surfaces of a platter are covered with a magnetic material.
  - store information by recording it magnetically on the platters,
  - read information by detecting the magnetic pattern on the platters
  - Drives rotate at 60 to 250 times per second
  - **Transfer rate (Rotation speed)** is rate at which data flow between drive and computer
  - **Positioning time (random-access time)** is:
    - ▶ time to move disk arm to desired cylinder (**seek time**) and
    - ▶ time for desired sector to rotate under the disk head (**rotational latency**)
  - **Head crash** results from disk head making contact with the disk surface
    - That's bad
- Disks can be removable





## Moving-head Disk Mechanism

Positioning time:

Rotational latency

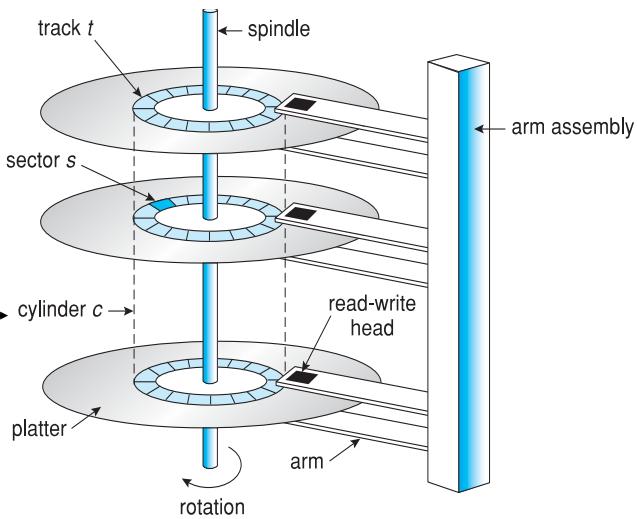
Seek time

Transfer rate  
(Rotation speed)

11.5

Operating System Concepts – 10<sup>th</sup> Edition

Silberschatz, Galvin and Gagne ©2018



## Hard Disk Drives

- Platters range from .85" to 14" (historically)
  - Commonly 3.5", 2.5", and 1.8"
- Range from 30GB to 3TB per drive
- Performance
  - Transfer Rate – theoretical – 6 Gb/sec
  - Effective Transfer Rate – real – 1Gb/sec
  - Seek time from 3ms to 12ms – 9ms common for desktop drives
  - Average seek time measured or calculated based on 1/3 of tracks
  - Latency based on spindle speed
    - ▶  $1 / (\text{RPM} / 60) = 60 / \text{RPM}$
  - Average latency =  $\frac{1}{2}$  latency



\* RPM: Revolutions per minute, số vòng quay  
trên 1 phút.

Operating System Concepts – 10<sup>th</sup> Edition

11.6

Silberschatz, Galvin and Gagne ©2018

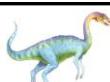




## Hard Disk Performance

- **Access Latency** = **Average access time** = average seek time + average latency (milliseconds)
  - For fastest disk 3ms + 2ms = 5ms
  - For slow disk 9ms + 5.56ms = 14.56ms
- Average I/O time = average access time + (amount to transfer / transfer rate) + controller overhead
- For example to transfer a 4KB block on a 7200 RPM disk with a 5ms average seek time, 1Gb/sec transfer rate with a 0.1ms controller overhead
  - **Average latency** =  $(60 \times 1000 \div 7200) \div 2 = 4.17\text{ms}$
  - Transfer time =  $4\text{KB} \div 1\text{Gb/s} = (4 \times 1024 \times 8)\text{b} \div 10^9 = 0.033\text{ms}$
  - Average I/O time =  $(5\text{ms} + 4.17\text{ms}) + 0.1\text{ms} + 0.033\text{ms} = 9.303\text{ms}$
  - Average I/O time for 4KB block = 9.303ms

\* *RPM: Revolutions per minute, số vòng quay trên 1 phút.*



## The First Commercial Disk Drive



1956

IBM RAMDAC computer included the IBM Model 350 disk storage system

5M (7 bit) characters

50 x 24" platters

Access time = < 1 second





## Nonvolatile Memory (NVM) Devices

- NVM devices are electrical, is composed of a controller and **flash NAND** die semiconductor chips, which are used to store data.
  - It is used in a disk-drive-like container, called **solid-state disks (SSDs)**
- Other forms include: **USB drives** (thumb drive, flash drive), surface-mounted on motherboards, and main storage in devices like smartphones
- NVM
  - Can be more reliable than HDDs
  - More expensive per MB
  - Maybe have shorter life span – need careful management
  - Less capacity
  - But much faster
  - Busses can be too slow -> connect directly to PCI for example
  - No moving parts, so no seek time or rotational latency



## Flash NAND vs NOR

- Bộ nhớ flash NAND (Not And) – on NVM devices:
  - là ổ nhớ flash thể rắn, dung lượng cao, rất nhanh.
  - sử dụng rất ít năng lượng, thuận tiện cho máy tính xách tay,
  - các chip nhỏ hơn, cho phép nhiều bộ nhớ được đóng gói vào cùng một không gian kích thước.
  - xóa nhanh hơn, cho nên NAND ghi nhanh hơn (vì bộ nhớ flash phải xóa các khối bộ nhớ trước khi ghi vào chúng).
  - NAND có tuổi thọ dài hơn với nhiều chu kỳ xóa hơn.
  - ưu điểm hơn so với các loại bộ nhớ flash khác, ex flash NOR (Not Or)
- Bộ nhớ flash NOR
  - thường được sử dụng trong điện thoại di động, PDA và các thiết bị khác, trong đó các chương trình nhỏ được thực thi tại chỗ thay vì ghi vào RAM.
  - Bộ nhớ flash NOR là một lựa chọn tốt cho các ứng dụng yêu cầu ít bộ nhớ và thực hiện hầu hết công việc của chúng trên mã không thay đổi, chẳng hạn như hướng khởi động hoặc hệ điều hành.





## Nonvolatile Memory Devices

- Have characteristics that present challenges
- Read and written in “page” increments (think sector) but can’t overwrite in place
  - Must first be erased, and erases happen in larger “block” increments
  - Can only be erased a limited number of times before worn out – ~ 100,000
  - Life span measured in **drive writes per day (DWPD)**
    - ▶ A 1TB NAND drive with rating of 5DWPD is expected to have 5TB per day written within warranty period without failing
- Ex, A 3.5-inch SSD circuit board



Operating System Concepts – 10<sup>th</sup> Edition

11.11

Silberschatz, Galvin and Gagne ©2018



## SSD

- Chi phí của NAND flash đã giảm đáng kể => các thiết bị lưu trữ chính mới như ổ cứng thẻ rắn SSD trở thành phổ biến thay thế cho HDD truyền thống
- Ổ cứng SSD
  - lợi thế đáng kể về hiệu năng và độ bền so với ổ cứng tiêu chuẩn.
  - không có bộ phận chuyển động, nên không gặp phải tình trạng trễ cơ học như ổ cứng, có thể chịu được va đập và rung lắc tốt hơn
  - thiết bị này chỉ có các thiết bị bán dẫn NAND.



Operating System Concepts – 10<sup>th</sup> Edition

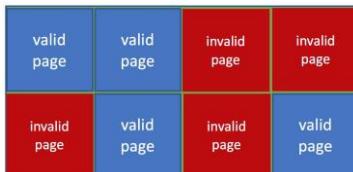
11.12

Silberschatz, Galvin and Gagne ©2018



## NAND Flash Controller Algorithms

- With no overwrite, pages end up with mix of valid and invalid data
- To track which logical blocks are valid, controller maintains **flash translation layer (FTL)** table
- Also implements **garbage collection** to free invalid page space
- Allocates **overprovisioning** to provide working space for GC
- Each cell has lifespan, so **wear leveling** needed to write equally to all cells



NAND block with valid and invalid pages



## Volatile Memory

- DRAM frequently used as mass-storage device
  - Not technically secondary storage because volatile, but can have file systems, be used like very fast secondary storage
- **RAM drives** (with many names, including RAM disks) present as raw block devices, commonly file system formatted
- Computers have buffering, caching via RAM, so why RAM drives?
  - Caches / buffers allocated / managed by programmer, operating system, hardware
  - RAM drives under user control
  - Found in all major operating systems
    - Linux `/dev/ram`, macOS `diskutil` to create them, Linux `/tmp` of file system type `tmpfs`
- Used as high speed temporary storage
  - Programs could share bulk data, quickly, by reading/writing to RAM drive





## Magnetic Tape

**Magnetic tape** was used as an early secondary-storage medium. Although it is nonvolatile and can hold large quantities of data, its access time is slow compared with that of main memory and drives. In addition, random access to magnetic tape is about a thousand times slower than random access to HDDs and about a hundred thousand times slower than random access to SSDs so tapes are not very useful for secondary storage. Tapes are used mainly for backup, for storage of infrequently used information, and as a medium for transferring information from one system to another.

A tape is kept in a spool and is wound or rewound past a read–write head. Moving to the correct spot on a tape can take minutes, but once positioned, tape drives can read and write data at speeds comparable to HDDs. Tape capacities vary greatly, depending on the particular kind of tape drive, with current capacities exceeding several terabytes. Some tapes have built-in compression that can more than double the effective storage. Tapes and their drivers are usually categorized by width, including 4, 8, and 19 millimeters and 1/4 and 1/2 inch. Some are named according to technology, such as LTO-6 (Figure 11.5) and SDLT.



Figure 11.5 An LTO-6 Tape drive with tape cartridge inserted.



## Disk Attachment

- A secondary storage device is attached to a computer by the system bus or an I/O bus
- Several busses available, including **advanced technology attachment (ATA)**, **serial ATA (SATA)**, **eSATA**, **serial attached SCSI (SAS)**, **universal serial bus (USB)**, and **fibre channel (FC)**.
- Most common is SATA
- Because NVM much faster than HDD, new fast interface for NVM called **NVM express (NVMe)**, connecting directly to PCI bus
- Data transfers on a bus carried out by special electronic processors called **controllers** (or **host-bus adapters, HBAs**)
  - Host controller on the computer end of the bus, while device controller on device end
  - Computer places command on host controller, using memory-mapped I/O ports
    - Host controller sends messages to device controller
    - Data transferred via DMA between device and computer DRAM





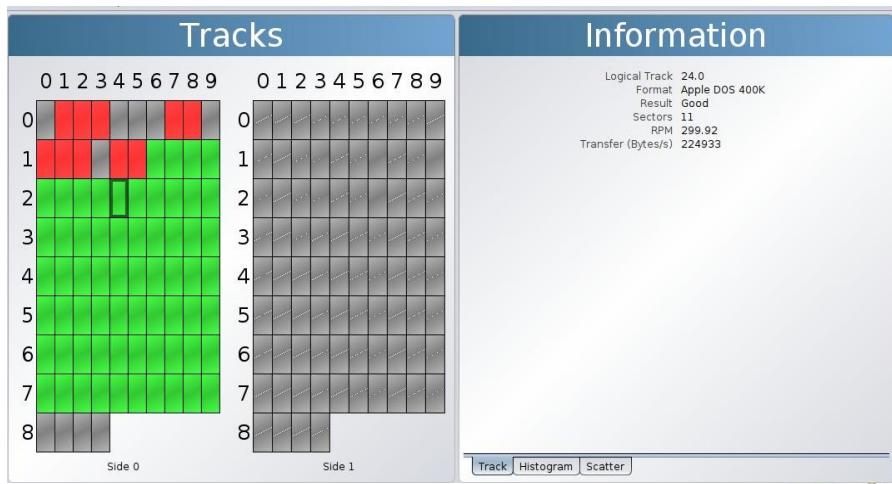
## Address Mapping

- Disk drives are addressed as large 1-dimensional arrays of **logical blocks**, where the logical block is the smallest unit of transfer
  - Low-level formatting creates **logical blocks** on physical media
- The 1-dimensional array of logical blocks is mapped into the sectors of the disk sequentially
  - Sector 0 is the first sector of the first track on the outermost cylinder
  - Mapping proceeds in order through that track, then the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost
  - Logical to physical address should be easy
    - Except for bad sectors
    - Non-constant # of sectors per track via constant angular velocity



## Ex

- 1-dimensional array of logical blocks is mapped into the sectors





## HDD Scheduling

- The operating system is responsible for using hardware efficiently — for the disk drives, this means having a fast access time and disk bandwidth
  - Minimize seek time
  - Seek time  $\approx$  seek distance
- Disk **bandwidth** is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer
- Whenever a process needs I/O to or from the drive, it issues a system call to the operating system.
  - If the desired drive and controller are available, It can be serviced
  - If the drive or controller is busy, It will be placed in the queue
- OS maintains queue of requests, per disk or device
  - Optimization algorithms only make sense when a queue exists



## Disk Scheduling (Cont.)

- In the past, operating system responsible for queue management, disk drive head scheduling
  - Now, built into the storage devices, controllers
  - Just provide Logical Block Addressing (LBA), handle sorting of requests
    - ▶ Some of the algorithms they use described next
- Note that drive controllers have small buffers and can manage a queue of I/O requests (of varying “depth”)
- Several algorithms exist to schedule the servicing of disk I/O requests
- The analysis is true for one or many platters
- We illustrate scheduling algorithms with a request queue (0-199)
  - 98, 183, 37, 122, 14, 124, 65, 67

Head pointer 53

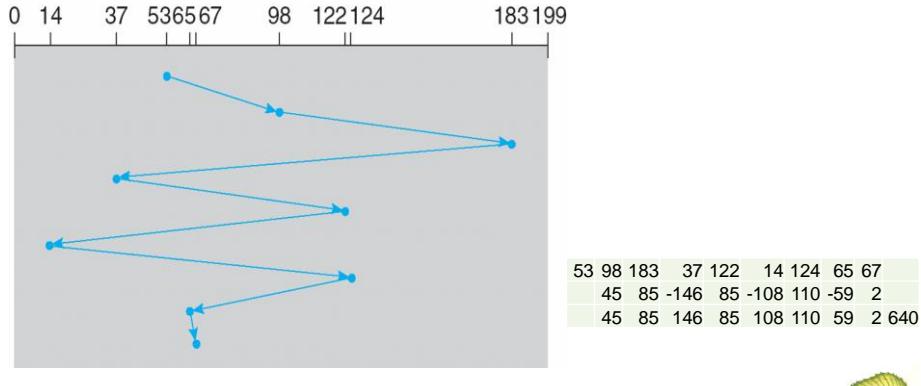




## FCFS - first-come, first-served

- Illustration shows total head movement of 640 cylinders

queue = 98, 183, 37, 122, 14, 124, 65, 67  
head starts at 53



Operating System Concepts – 10<sup>th</sup> Edition

11.21

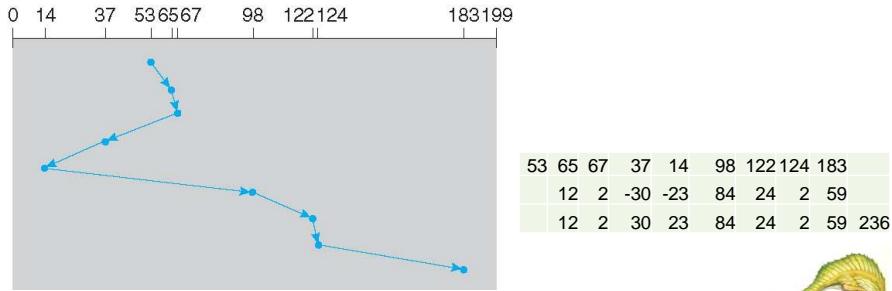
Silberschatz, Galvin and Gagne ©2018



## SSTF

- Shortest Seek Time First selects the request with the minimum seek time from the current head position
- SSTF scheduling is a form of SJF scheduling; may cause starvation of some requests
- Illustration shows total head movement of 236 cylinders

queue = 98, 183, 37, 122, 14, 124, 65, 67  
head starts at 53



Operating System Concepts – 10<sup>th</sup> Edition

11.22

Silberschatz, Galvin and Gagne ©2018

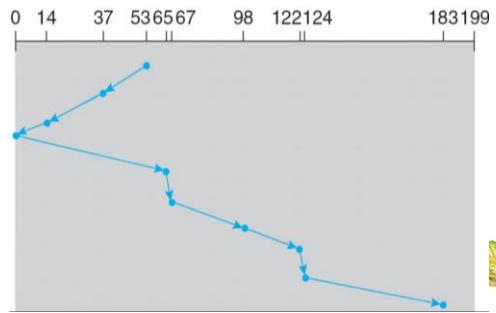




## SCAN

- The disk arm starts at one end of the disk, and moves toward the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues.
- SCAN algorithm** Sometimes called the **elevator algorithm**
- Illustration shows total head movement of 208 cylinders
- But note that if requests are uniformly dense, largest density at other end of disk and those wait the longest

queue = 98, 183, 37, 122, 14, 124, 65, 67  
head starts at 53



Operating System Concepts – 10<sup>th</sup> Edition

11.23

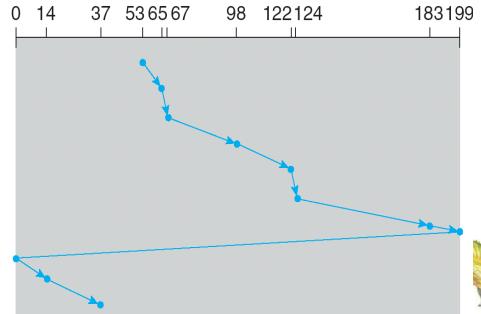
Silberschatz, Galvin and Gagne ©2018



## Circular-SCAN (C-SCAN)

- Provides a more uniform wait time than SCAN
- The head moves from one end of the disk to the other, servicing requests as it goes
  - When it reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip
- Treats the cylinders as a circular list that wraps around from the last cylinder to the first one
- Total number of cylinders?

queue = 98, 183, 37, 122, 14, 124, 65, 67  
head starts at 53



Operating System Concepts – 10<sup>th</sup> Edition

11.24

Silberschatz, Galvin and Gagne ©2018



## Selecting a Disk-Scheduling Algorithm

- SSTF is common and has a natural appeal
- SCAN and C-SCAN perform better for systems that place a heavy load on the disk
  - Less starvation, but still possible
- To avoid starvation Linux implements **deadline** scheduler
  - Maintains separate read and write queues, gives read priority
    - Because processes more likely to block on read than write
  - Implements four queues: 2 x read and 2 x write
    - 1 read and 1 write queue sorted in LBA order, essentially implementing C-SCAN
    - 1 read and 1 write queue sorted in FCFS order
    - All I/O requests sent in batch sorted in that queue's order
    - After each batch, checks if any requests in FCFS older than configured age (default 500ms)
      - If so, Logical Block Addressing (LBA) queue containing that request is selected for next batch of I/O
- In Red Hat (RHEL 7) also **NOOP** and **completely fair queueing** scheduler (**CFQ**) also available, defaults vary by storage device



## NVM Scheduling

- NVM devices do not contain moving disk heads and commonly use a simple FCFS policy.
- For example, the Linux NOOP scheduler uses an FCFS policy but modifies it to merge adjacent requests.
- In NVM devices:
  - the time required to service reads is uniform
  - write service time is not uniform - slower than reading, decreasing the adv
  - I/O can occur sequentially or randomly, HDD at sequential
  - Throughput can be similar
  - Random-access I/O (is measured in Input/Output operations per second - **IOPS**): much higher with NVM than HDD
  - **Write amplification** (one write, causing garbage collection and many read/writes) can decrease the performance advantage





## Error Detection and Correction

- **Error detection** determines if there a problem has occurred (for example a bit flipping)
  - If detected, can halt the operation
  - Detection frequently done via parity bit
- Parity is one form of **checksum** – uses modular arithmetic to compute, store, compare values of fixed-length words
- Another error-detection method common in networking is **cyclic redundancy check (CRC)** which uses hash function to detect multiple-bit errors
- **Error-correction code (ECC)** not only detects, but can correct some errors
  - Soft errors correctable, while hard errors detected but not corrected



## Storage Device Management

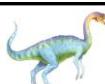
- **Low-level formatting**, or **physical formatting** — Dividing a disk into sectors that the disk controller can read and write
  - Each sector can hold header information, plus data, plus error correction code (**ECC**)
  - Usually 512 bytes of data but can be selectable
- To use a disk to hold files, the operating system still needs to record its own data structures on the disk
  - **Partition** the disk into one or more groups of cylinders, each treated as a logical disk
  - **Logical formatting** or “making a file system”
  - To increase efficiency most file systems group blocks into **clusters**
    - ▶ Disk I/O done in blocks
    - ▶ File I/O done in clusters





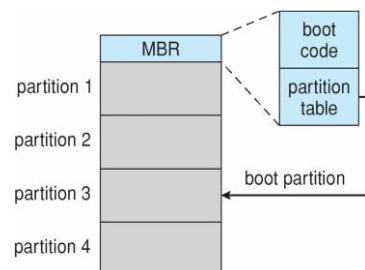
## Storage Device Management (cont.)

- **Root partition** contains the OS, other partitions can hold other Oses, other file systems, or be raw
  - **Mounted** at boot time
  - Other partitions can mount automatically or manually
- At mount time, file system consistency checked
  - Is all metadata correct?
    - ▶ If not, fix it, try again
    - ▶ If yes, add to mount table, allow access
- Boot block can point to boot volume or boot loader set of blocks that contain enough code to know how to load the kernel from the file system
  - Or a boot management program for multi-os booting



## Device Storage Management (Cont.)

- Raw disk access for apps that want to do their own block management, keep OS out of the way (databases for example)
- Boot block initializes system
  - The bootstrap is stored in ROM, firmware
  - **Bootstrap loader** program stored in boot blocks of boot partition
- Methods such as **sector sparing** used to handle bad blocks



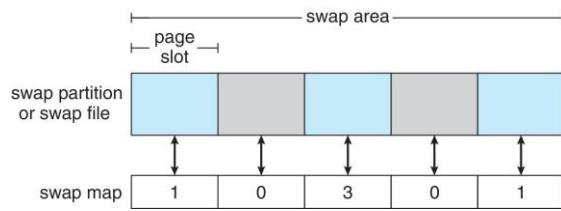
Booting from secondary storage in Windows





## Swap-Space Management

- Used for moving entire processes (swapping), or pages (paging), from DRAM to secondary storage when DRAM not large enough for all processes
- Operating system provides **swap space management**
  - Secondary storage slower than DRAM, so important to optimize performance
  - Usually multiple swap spaces possible – decreasing I/O load on any given device
  - Best to have dedicated devices
  - Can be in raw partition or a file within a file system (convenience of adding)
  - Data structures for swapping on Linux systems:



## Storage Attachment

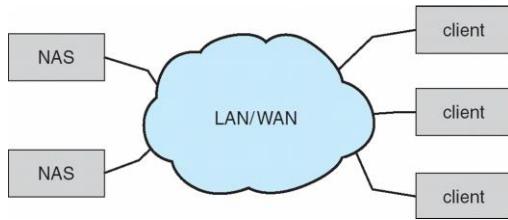
- Computers access storage in three ways
  - Host-attached
  - Network-attached
  - Cloud
- Host attached access through local I/O ports, using one of several technologies
  - To attach many devices, use storage busses such as USB, firewire, thunderbolt
  - High-end systems use **fibre channel (FC)**
    - High-speed serial architecture using fibre or copper cables
    - Multiple hosts and storage devices can connect to the FC fabric





## Network-Attached Storage

- Network-attached storage (**NAS**) is storage made available over a network rather than over a local connection (such as a bus)
  - Remotely attaching to file systems
- NFS and CIFS are common protocols
- Implemented via remote procedure calls (RPCs) between host and storage over typically TCP or UDP on IP network
- **iSCSI** protocol uses IP network to carry the SCSI protocol
  - Remotely attaching to devices (blocks)



## Cloud Storage

- Similar to NAS, provides access to storage across a network
  - Unlike NAS, accessed over the Internet or a WAN to remote data center
- NAS presented as just another file system, while cloud storage is API based, with programs using the APIs to provide access
  - Examples include Dropbox, Amazon S3, Microsoft OneDrive, Apple iCloud
  - Use APIs because of latency and failure scenarios (NAS protocols wouldn't work well)





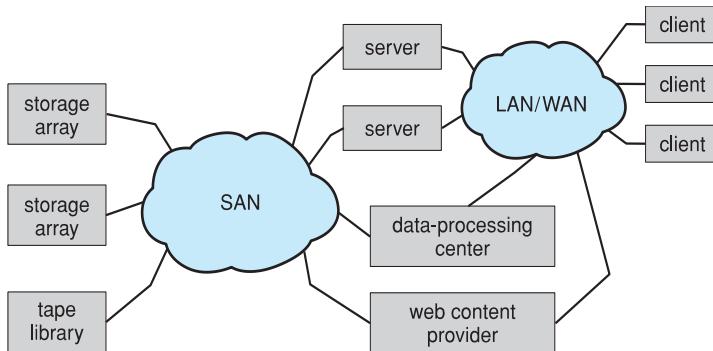
## Storage Array

- Can just attach disks, or arrays of disks
- Avoids the NAS drawback of using network bandwidth
- Storage Array has controller(s), provides features to attached host(s)
  - Ports to connect hosts to array
  - Memory, controlling software (sometimes NVRAM, etc.)
  - A few to thousands of disks
  - RAID, hot spares, hot swap (discussed later)
  - Shared storage -> more efficiency
  - Features found in some file systems
    - ▶ Snapshots, clones, thin provisioning, replication, deduplication, etc



## Storage Area Network

- Common in large storage environments
- Multiple hosts attached to multiple storage arrays – flexible





## Storage Area Network (Cont.)

- SAN is one or more storage arrays
  - Connected to one or more Fibre Channel switches or **InfiniBand (IB)** network
- Hosts also attach to the switches
- Storage made available via **LUN Masking** from specific arrays to specific servers
- Easy to add or remove storage, add new host and allocate it storage
- Why have separate storage networks and communications networks?
  - Consider iSCSI, FCOE



A Storage Array



## RAID Structure

- **RAID – redundant array of inexpensive disks**
  - multiple disk drives provides reliability via **redundancy**
- Increases the **mean time to failure**
- **Mean time to repair** – exposure time when another failure could cause data loss
- **Mean time to data loss** based on above factors
- If mirrored disks fail independently, consider disk with 1300,000 **mean time to failure** and 10 hour mean time to repair
  - Mean time to data loss is  $100,000^2 / (2 * 10) = 500 * 10^6$  hours, or 57,000 years!
- Frequently combined with **NVRAM** to improve write performance
- Several improvements in disk-use techniques involve the use of multiple disks working cooperatively



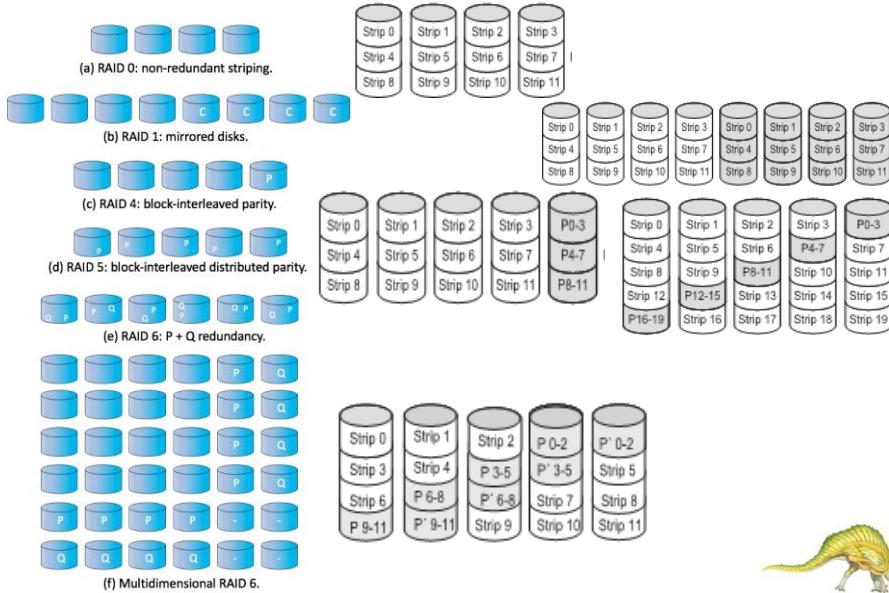


## RAID (Cont.)

- Disk **striping** uses a group of disks as one storage unit
- RAID is arranged into six different levels
- RAID schemes improve performance and improve the reliability of the storage system by storing redundant data
  - **Mirroring or shadowing (RAID 1)** keeps duplicate of each disk
  - Striped mirrors (**RAID 1+0**) or mirrored stripes (**RAID 0+1**) provides high performance and high reliability
  - **Block interleaved parity (RAID 4, 5, 6)** uses much less redundancy
- RAID within a storage array can still fail if the array fails, so automatic **replication** of the data between arrays is common
- Frequently, a small number of **hot-spare** disks are left unallocated, automatically replacing a failed disk and having data rebuilt onto them

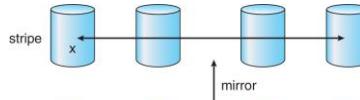


## RAID Levels

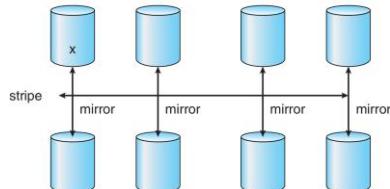




## RAID (0 + 1) and (1 + 0)



a) RAID 0 + 1 with a single disk failure.



b) RAID 1 + 0 with a single disk failure.



## Other Features

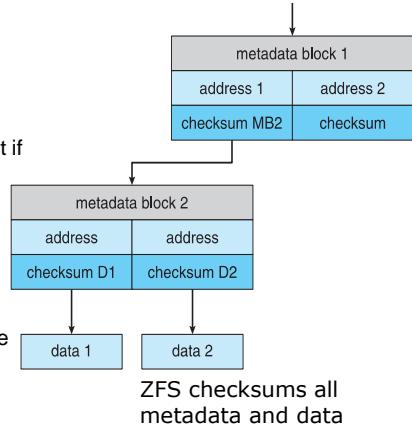
- Regardless of where RAID implemented, other useful features can be added
- **Snapshot** is a view of file system before a set of changes take place (i.e., at a point in time)
- Replication is automatic duplication of writes between separate sites
  - For redundancy and disaster recovery
  - Can be synchronous or asynchronous
- Hot spare disk is unused, automatically used by RAID production if a disk fails to replace the failed disk and rebuild the RAID set if possible
  - Decreases average of the time to repair



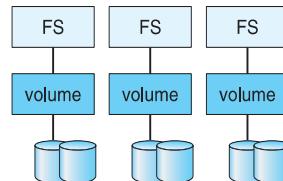


## Extensions

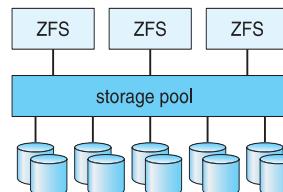
- RAID alone does not prevent or detect data corruption or other errors, just disk failures
- Solaris ZFS adds **checksums** of all data and metadata
- Checksums kept with pointer to object, to detect if object is the right one and whether it changed
- Can detect and correct data and metadata corruption
- ZFS also removes volumes, partitions
  - Disks allocated in **pools**
  - Filesystems with a pool share that pool, use and release space like `malloc()` and `free()` memory allocate / release calls



## Traditional and Pooled Storage



(a) Traditional volumes and file systems.



(b) ZFS and pooled storage.



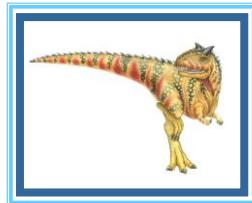


## Object Storage

- General-purpose computing, file systems not sufficient for very large scale
- Another approach – start with a storage pool and place objects in it
  - Object just a container of **data**
  - No way to navigate the pool to find objects (no directory structures, few services)
  - Computer-oriented, not user-oriented
- Typical sequence
  - Create an object within the pool, receive an object ID
  - Access object via that ID
  - Delete object via that ID
- Object storage management software like **Hadoop file system (HDFS)** and **Ceph** determine where to store objects, manages protection
  - Typically by storing N copies, across N systems, in the object storage cluster
  - **Horizontally scalable**
  - **Content addressable, unstructured**



## End of Chapter 4.1



## Chapter 4: File System Management & I/O System

### 4.2. File System Implementation



GV: Nguyễn Thị Thanh Vân

Operating System Concepts – 10<sup>th</sup> Edition

Silberschatz, Galvin and Gagne ©2018

## Outline

- File-System Structure
- File-System Operations
- Directory Implementation
- Allocation Methods
- Free-Space Management
- Efficiency and Performance
- Recovery
- Example: WAFL File System



Operating System Concepts – 10<sup>th</sup> Edition

14.2

Silberschatz, Galvin and Gagne ©2018



## Objectives

- Describe the details of implementing local file systems and directory structures
- Discuss block allocation and free-block algorithms and trade-offs
- Explore file system efficiency and performance issues
- Look at recovery from file system failures
- Describe the WAFL file system as a concrete example



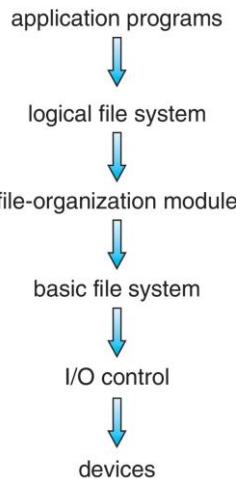
## File-System Structure

- File structure
  - Logical storage unit
  - Collection of related information
- **File system** resides on secondary storage (disks)
  - Provided user interface to storage, mapping logical to physical
  - Provides efficient and convenient access to disk by allowing data to be stored, located retrieved easily
- Disk provides in-place rewrite and random access
  - I/O transfers performed in **blocks** of **sectors** (usually 512 bytes)
- **File control block (FCB)** – storage structure consisting of information about a file
- **Device driver** controls the physical device
- File system organized into layers





## Layered File System



## File System Layers

- **Device drivers** manage I/O devices at the I/O control layer
  - Given commands like “read drive1, cylinder 72, track 2, sector 10, into memory location 1060” outputs low-level hardware specific commands to hardware controller
- **Basic file system** given command like “retrieve block 123” translates to device driver
- Also manages memory buffers and caches (allocation, freeing, replacement)
  - Buffers hold data in transit
  - Caches hold frequently used data
- **File organization module** understands files, logical address, and physical blocks
- Translates logical block # to physical block #
- Manages free space, disk allocation





## File System Layers (Cont.)

- **Logical file system** manages metadata information
  - Translates file name into file number, file handle, location by maintaining file control blocks (**inodes** in UNIX)
  - Directory management
  - Protection
- Layering useful for reducing complexity and redundancy, but adds overhead and can decrease performance
- Logical layers can be implemented by any coding method according to OS designer



## File System Layers (Cont.)

- Many file systems, sometimes many within an operating system
  - Each with its own format:
  - CD-ROM is ISO 9660;
  - Unix has **UFS**, FFS;
  - Windows has FAT, FAT32, NTFS as well as floppy, CD, DVD Blu-ray,
  - Linux has more than 130 types, with **extended file system** ext3 and ext4 leading; plus distributed file systems, etc.)
  - New ones still arriving – ZFS, GoogleFS, Oracle ASM, FUSE





## File-System Operations

- We have system calls at the API level, but how do we implement their functions?
  - On-disk and in-memory structures
- **Boot control block** contains info needed by system to boot OS from that volume
  - Needed if volume contains OS, usually first block of volume
- **Volume control block (superblock, master file table)** contains volume details
  - Total # of blocks, # of free blocks, block size, free block pointers or array
- Directory structure organizes the files
  - Names and inode numbers, master file table



## File-System Implementation (Cont.)

- Per-file **File Control Block (FCB)** contains many details about the file
  - Typically inode number, permissions, size, dates
  - NFTS stores into in master file table using relational DB structures

file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks





## In-Memory File System Structures

- **Mount table** storing file system mounts, mount points, file system types
- **System-wide open-file table** contains a copy of the FCB of each file and other info
- **Per-process open-file table** contains pointers to appropriate entries in system-wide open-file table as well as other info
- The following figure illustrates the necessary file system structures provided by the operating systems
- Plus buffers hold data blocks from secondary storage
- Open returns a file handle for subsequent use
- Data from read eventually copied to specified user process memory address



Operating System Concepts – 10<sup>th</sup> Edition

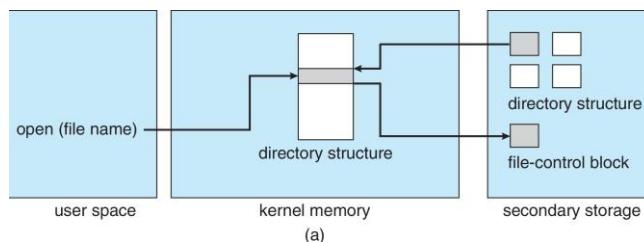
14.11

Silberschatz, Galvin and Gagne ©2018

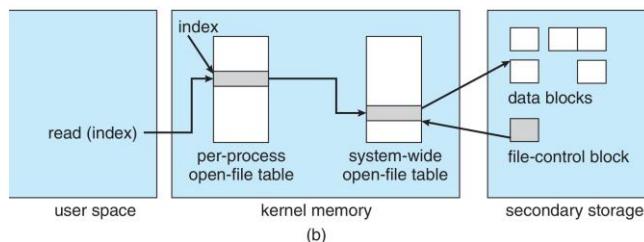


## In-Memory File System Structures

Figure (a) refers to opening a file  
Figure (b) refers to reading a file



(a)



(b)



Operating System Concepts – 10<sup>th</sup> Edition

14.12

Silberschatz, Galvin and Gagne ©2018



## Directory Implementation

- **Linear list** of file names with pointer to the data blocks
  - Simple to program
  - Time-consuming to execute
    - ▶ Linear search time
  - Could keep ordered alphabetically via linked list or use B+ tree
- **Hash Table** – linear list with hash data structure
  - Decreases directory search time
  - **Collisions** – situations where two file names hash to the same location
  - Only good if entries are fixed size, or use chained-overflow method



## Allocation Methods - Contiguous

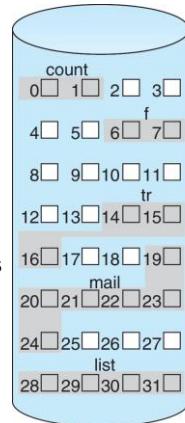
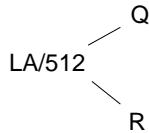
- An allocation method refers to how disk blocks are allocated for files:
- Each file occupies set of contiguous blocks
  - Best performance in most cases
  - Simple – only starting location (block #) and length (number of blocks) are required
  - Problems include:
    - ▶ Finding space for file,
    - ▶ Knowing file size,
    - ▶ External fragmentation, need for **compaction off-line (downtime)** or **on-line**





## Contiguous Allocation

- Mapping from logical to physical (block size =512 bytes)



directory		
file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

- Block to be accessed = starting address + Q
- Displacement into block = R
- Contiguous allocation is easy to implement but has limitations, and is therefore not used in modern file systems.



## Extent-Based Systems

- Many newer file systems (i.e., Veritas File System) use a modified contiguous allocation scheme
- Extent-based file systems allocate disk blocks in extents
- An **extent** is a contiguous block of disks
  - Extents are allocated for file allocation
  - A file consists of one or more extents





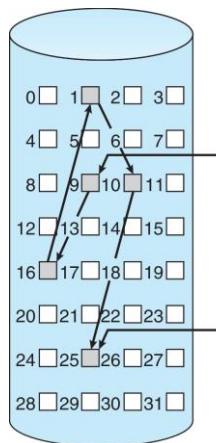
## Allocation Methods - Linked

- Each file a linked list of blocks
  - File ends at nil pointer
  - No external fragmentation
  - Each block contains pointer to next block
  - No compaction, external fragmentation
  - Free space management system called when new block needed
  - Improve efficiency by clustering blocks into groups but increases internal fragmentation
  - Reliability can be a problem
  - Locating a block can take many I/Os and disk seeks



## Linked Allocation

- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk
- Scheme. Ex: a file of 5 blocks: start at block 9, and continue at block 16, then block 1, then block 10, and finally block 25



directory		
file	start	end
jeep	9	25

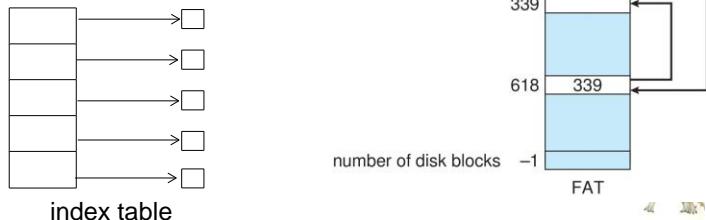
- Mapping      LA/511 → Q  
LA/511 → R
- Block to be accessed is the Qth block in the linked chain of blocks representing the file.
- Displacement into block = R + 1





## Allocation Methods – FAT

- FAT (File Allocation Table) - An important variation on linked allocation
  - Beginning of volume has table, indexed by block number
  - Much like a linked list, but faster on disk and cacheable
  - New block allocation simple
- Ex:
  - a file consisting of disk blocks 217, 618, and 339
- Each file has its own **index block(s)** of pointers to its data blocks
- Logical view



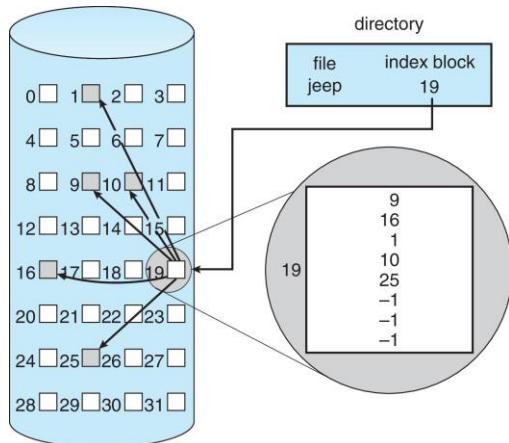
Operating System Concepts – 10<sup>th</sup> Edition

14.19

Silberschatz, Galvin and Gagne ©2018



## Example of Indexed Allocation



Operating System Concepts – 10<sup>th</sup> Edition

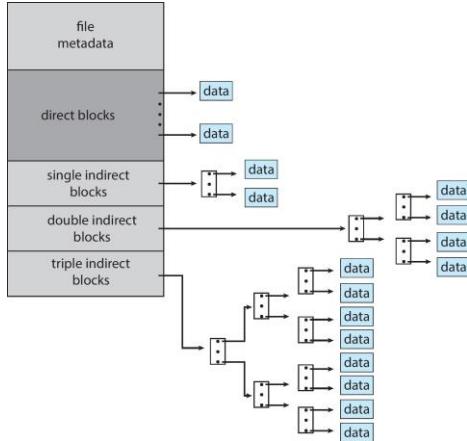
14.20

Silberschatz, Galvin and Gagne ©2018



## Combined Scheme: UNIX UFS

4K bytes per block, 32-bit addresses



More index blocks than can be addressed with 32-bit file pointer



## Performance

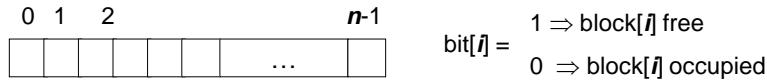
- Best method depends on file access type
  - Contiguous great for sequential and random
- Linked good for sequential, not random
- Declare access type at creation -> select either contiguous or linked
- Indexed more complex
  - Single block access could require 2 index block reads then data block read
  - Clustering can help improve throughput, reduce CPU overhead
- For NVM, no disk head so different algorithms and optimizations needed
  - Using old algorithm uses many CPU cycles trying to avoid non-existent head movement
  - With NVM goal is to reduce CPU cycles and overall path needed for I/O





## Free-Space Management

- File system maintains **free-space list** to track available blocks/clusters (Using term “block” for simplicity)
  - the **free-space list** is implemented as a bitmap or bit vector.
- Bit vector** or **bit map** ( $n$  blocks)



- Ex: 00111100111111000110000011100000 ...
- Block number calculation:

(number of bits per word) \* (number of 0-value words) + offset of first 1 bit  
CPUs have instructions to return offset within word of first “1” bit

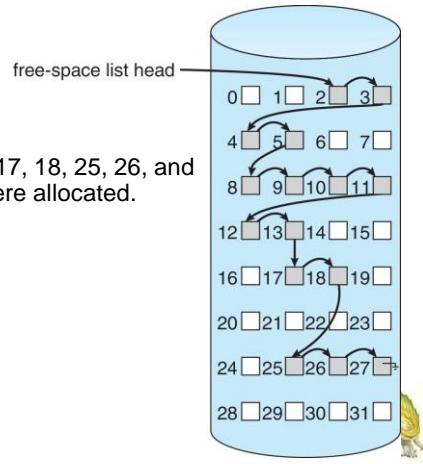
- Bit map requires extra space  
Example:

block size = 4KB =  $2^{12}$  bytes  
disk size =  $2^{40}$  bytes (1 terabyte)  
 $n = 2^{40}/2^{12} = 2^{28}$  bits (or 32MB)  
if clusters of 4 blocks -> 8MB of memory



## Linked Free Space List on Disk

- Linked list (free list)** - Another approach to free-space management. Advantage:
  - Cannot get contiguous space easily
  - No waste of space
  - No need to traverse the entire list (if # free blocks recorded)
- Ex: blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 were free and the rest of the blocks were allocated.





## Free-Space Management (Cont.)

- **Grouping:** A modification of the free-list approach
  - Modify linked list to store address of next  $n-1$  free blocks in first free block, plus a pointer to next block that contains free-block-pointers (like this one)
- **Counting:** Another approach
  - Because space is frequently contiguously used and freed, with contiguous-allocation allocation, extents, or clustering
    - Keep address of first free block and count of following free blocks
    - Free space list then has entries containing addresses and counts



## Free-Space Management (Cont.)

- **Space Maps**
  - Used in **ZFS** (Solaris, OS: huge numbers of files, directories)
  - Consider meta-data I/O on very large file systems
    - Full data structures like bit maps couldn't fit in memory -> thousands of I/Os
  - Divides device space into **metaslab** units and manages metaslabs
    - Given volume can contain hundreds of metaslabs
  - Each metaslab has associated space map
    - Uses counting algorithm
  - But records to log file rather than file system
    - Log of all block activity, in time order, in counting format
  - Metaslab activity -> load space map into memory in balanced-tree structure, indexed by offset
    - Replay log into that structure
    - Combine contiguous free blocks into single entry





## Free-Space Management (Cont.)

### TRIMing Unused Blocks

- HDDS overwrite in place so need only free list
- Blocks not treated specially when freed
  - Keeps its data but without any file pointers to it, until overwritten
- Storage devices not allowing overwrite (like NVM) suffer badly with same algorithm
  - Must be erased before written, erases made in large chunks (blocks, composed of pages) and are slow
  - TRIM is a newer mechanism for the file system to inform the NVM storage device that a page is free
    - Can be garbage collected or if block is free, now block can be erased



## Efficiency and Performance

- Efficiency dependent on:
  - Disk allocation and directory algorithms
  - Types of data kept in file's directory entry
  - Pre-allocation or as-needed allocation of metadata structures
  - Fixed-size or varying-size data structures





## Efficiency and Performance (Cont.)

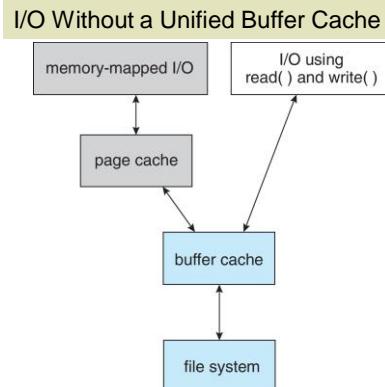
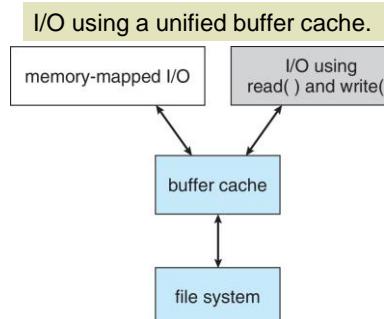
- Performance

- Keeping data and metadata close together
- **Buffer cache** – separate section of main memory for frequently used blocks
- **Synchronous** writes sometimes requested by apps or needed by OS
  - No buffering / caching – writes must hit disk before acknowledgement
  - **Asynchronous** writes more common, buffer-able, faster
- **Free-behind** and **read-ahead** – techniques to optimize sequential access
- Reads frequently slower than writes



## Page Cache

- A **page cache** caches pages rather than disk blocks using virtual memory techniques and addresses
- Memory-mapped I/O uses a page cache
- Routine I/O through the file system uses the buffer (disk) cache
- This leads to the following figure





## Recovery

- **Consistency checking** – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies
  - Can be slow and sometimes fails
- Use system programs to **back up** data from disk to another storage device (magnetic tape, other magnetic disk, optical)
- Recover lost file or disk by **restoring** data from backup



## Log Structured File Systems

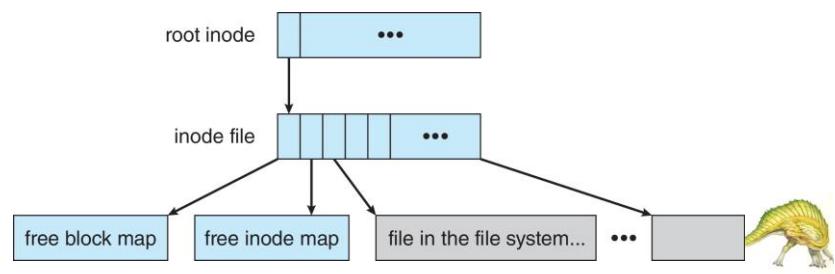
- **Log structured** (or **journaling**) file systems record each metadata update to the file system as a **transaction**
- All transactions are written to a log
  - A transaction is considered committed once it is written to the log (sequentially)
  - Sometimes to a separate device or section of disk
  - However, the file system may not yet be updated
- The transactions in the log are asynchronously written to the file system structures
  - When the file system structures are modified, the transaction is removed from the log
- If the file system crashes, all remaining transactions in the log must still be performed
- Faster recovery from crash, removes chance of inconsistency of metadata





## Example: WAFL File System

- Used on Network Appliance “Filers” – distributed file system appliances
- “Write-anywhere file layout”
- Serves up NFS, CIFS, http, ftp
- Random I/O optimized, write optimized
  - NVRAM for write caching
- Similar to Berkeley Fast File System, with extensive modifications
- The WAFL File Layout



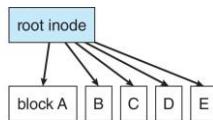
Operating System Concepts – 10<sup>th</sup> Edition

14.39

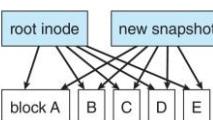
Silberschatz, Galvin and Gagne ©2018



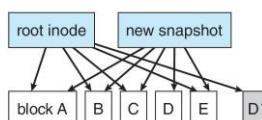
## Snapshots in WAFL



(a) Before a snapshot.



(b) After a snapshot, before any blocks change.



(c) After block D has changed to D'.



Operating System Concepts – 10<sup>th</sup> Edition

14.40

Silberschatz, Galvin and Gagne ©2018



## The Apple File System

In 2017, Apple, Inc., released a new file system to replace its 30-year-old HFS+ file system. HFS+ had been stretched to add many new features, but as usual, this process added complexity, along with lines of code, and made adding more features more difficult. Starting from scratch on a blank page allows a design to start with current technologies and methodologies and provide the exact set of features needed.

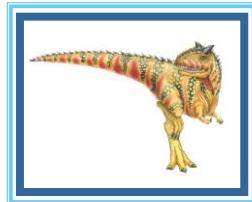
**Apple File System (APFS)** is a good example of such a design. Its goal is to run on all current Apple devices, from the Apple Watch through the iPhone to the Mac computers. Creating a file system that works in watchOS, I/Os, tvOS, and macOS is certainly a challenge. APFS is feature-rich, including 64-bit pointers, clones for files and directories, snapshots, space sharing, fast directory sizing, atomic safe-save primitives, copy-on-write design, encryption (single- and multi-key), and I/O coalescing. It understands NVM as well as HDD storage.

Most of these features we've discussed, but there are a few new concepts worth exploring. **Space sharing** is a ZFS-like feature in which storage is available as one or more large free spaces (**containers**) from which file systems can draw allocations (allowing APFS-formatted volumes to grow and shrink). **Fast directory sizing** provides quick used-space calculation and updating. **Atomic safe-save** is a primitive (available via API, not via file-system commands) that performs renames of files, bundles of files, and directories as single atomic operations. I/O coalescing is an optimization for NVM devices in which several small writes are gathered together into a large write to optimize write performance.

Apple chose not to implement RAID as part of the new APFS, instead depending on the existing Apple RAID volume mechanism for software RAID. APFS is also compatible with HFS+, allowing easy conversion for existing deployments.



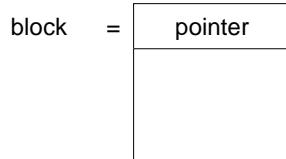
## End of Chapter 4.2



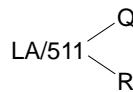


## Linked Allocation

- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk



- Mapping



Block to be accessed is the Qth block in the linked chain of blocks representing the file.

Displacement into block = R + 1



## Chapter 4: File System Management & I/O System

### 4.3. File System Internals



GV: Nguyễn Thị Thanh Vân

Operating System Concepts – 10<sup>th</sup> Edition

Silberschatz, Galvin and Gagne ©2018



### Outline

- File Systems
- File-System Mounting
- Partitions and Mounting
- File Sharing
- Virtual File Systems
- Remote File Systems
- Consistency Semantics
- NFS

Operating System Concepts – 10<sup>th</sup> Edition

15.2

Silberschatz, Galvin and Gagne ©2018





## Objectives

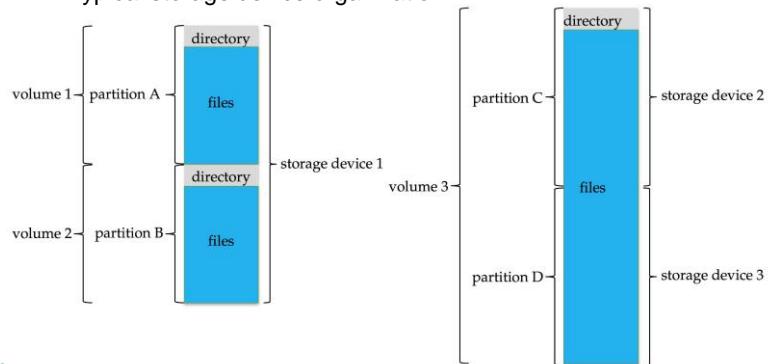
- Delve into the details of file systems and their implementation
- Explore booting and file sharing
- Describe remote file systems, using NFS as an example



## File System

- General-purpose computers can have multiple storage devices
  - Devices can be sliced into partitions, which hold volumes
  - Volumes can span multiple partitions
  - Each volume usually formatted into a file system
  - # of file systems varies, typically dozens available to choose from

Typical storage device organization:





## Example Mount Points and File Systems - Solaris

/	ufs
/devices	devfs
/dev	dev
/system/contract	ctfs
/proc	proc
/etc/mnttab	mntfs
/etc/svc/volatile	tmpfs
/system/object	objfs
/lib/libc.so.1	lofs
/dev/fd	fd
/var	ufs
/tmp	tmpfs
/var/run	tmpfs
/opt	ufs
/zpbge	zfs
/zpbge/backup	zfs
/export/home	zfs
/var/mail	zfs
/var/spool/mqueue	zfs
/zpbg	zfs
/zpbg/zones	zfs



## Partitions and Mounting

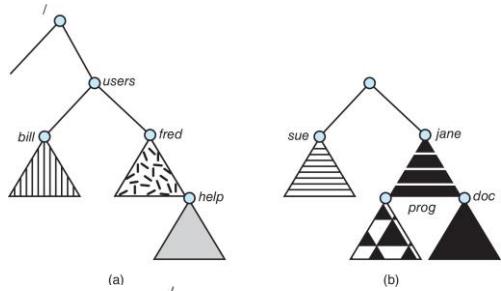
- Partition can be a volume containing a file system (“cooked”) or **raw** – just a sequence of blocks with no file system
- Boot block can point to boot volume or boot loader set of blocks that contain enough code to know how to load the kernel from the file system
  - Or a boot management program for multi-os booting
- **Root partition** contains the OS, other partitions can hold other OSes, other file systems, or be raw
  - Mounted at boot time
  - Other partitions can mount automatically or manually on **mount points** – location at which they can be accessed
- At mount time, file system consistency checked
  - Is all metadata correct?
    - ▶ If not, fix it, try again
    - ▶ If yes, add to mount table, allow access



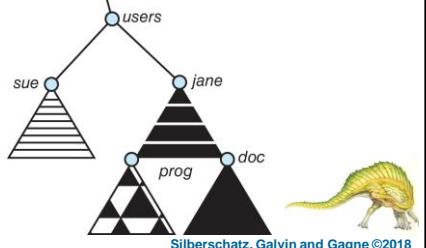


## File Systems and Mounting

- (a) Unix-like file system directory tree  
(b) Unmounted file system



- (c) After mounting (b) into the existing directory tree



## File Sharing

- Allows multiple users / systems access to the same files
- Permissions / protection must be implemented and accurate
  - Most systems provide concepts of owner, group member
  - Must have a way to apply these between systems





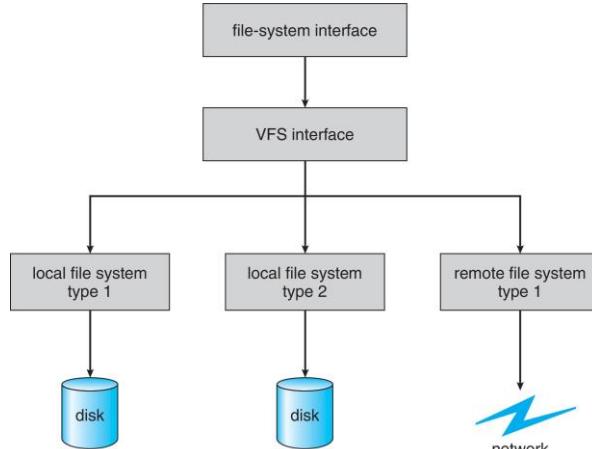
## Virtual File Systems

- **Virtual File Systems (VFS)** on Unix provide an object-oriented way of implementing file systems
- VFS allows the same system call interface (the API) to be used for different types of file systems
  - Separates file-system generic operations from implementation details
  - Implementation can be one of many file systems types, or network file system
    - Implements **vnodes** which hold inodes or network file details
  - Then dispatches operation to appropriate file system implementation routines



## Virtual File Systems (Cont.)

- The API is to the VFS interface, rather than any specific type of file system





## Virtual File System Implementation

- For example, Linux has four object types:
  - **inode, file, superblock, dentry**
- VFS defines set of operations on the objects that must be implemented
  - Every object has a pointer to a function table
    - Function table has addresses of routines to implement that function on that object
    - For example:
      - • `int open(...)`—Open a file
      - • `int close(...)`—Close an already-open file
      - • `ssize_t read(...)`—Read from a file
      - • `ssize_t write(...)`—Write to a file
      - • `int mmap(...)`—Memory-map a file



## Remote File Systems

- Sharing of files across a network
- **First** method involved manually sharing each file – programs like `ftp`
- **Second** method uses a **distributed file system (DFS)**
  - Remote directories visible from local machine
- **Third** method – **World Wide Web**
  - A bit of a revision to first method
  - Use browser to locate file/files and download /upload
  - **Anonymous** access doesn't require authentication





## Client-Server Model

- Sharing between a server (providing access to a file system via a network protocol) and a client (using the protocol to access the remote file system)
- Identifying each other via network ID can be spoofed, encryption can be performance expensive
- NFS an example
  - User auth info on clients and servers must match (UserIDs for example)
  - Remote file system mounted, file operations sent on behalf of user across network to server
  - Server checks permissions, file handle returned
  - Handle used for reads and writes until file closed



## Distributed Information Systems

- Aka **distributed naming services**, provide unified access to info needed for remote computing
- **Domain name system (DNS)** provides host-name-to-network-address translations for the Internet
- Others like **network information service (NIS)** provide user-name, password, userID, group information
- Microsoft's **common Internet file system (CIFS)** network info used with user auth to create network logins that server uses to allow to deny access
  - **Active directory** distributed naming service
  - **Kerberos-derived** network authentication protocol
- Industry moving toward **lightweight directory-access protocol (LDAP)** as secure distributed naming mechanism





## Consistency Semantics

- Important criteria for evaluating file sharing-file systems
- Specify how multiple users are to access shared file simultaneously
  - When modifications of data will be observed by other users
  - Directly related to process synchronization algorithms, but atomicity across a network has high overhead (see Andrew File System)
- The series of accesses between file open and closed called **file session**
- UNIX semantics
  - Writes to open file immediately visible to others with file open
  - One mode of sharing allows users to share pointer to current I/O location in file
  - Single physical image, accessed exclusively, contention causes process delays
- Session semantics (Andrew file system (OpenAFS))
  - Writes to open file not visible during session, only at close
  - Can be several copies, each changed independently



## The Sun Network File System (NFS)

- An implementation and a specification of a software system for accessing remote files across LANs (or WANs)
- The implementation originally part of SunOS operating system, now industry standard / very common
- Can use unreliable datagram protocol (UDP/IP) or TCP/IP, over Ethernet or other network





## NFS (Cont.)

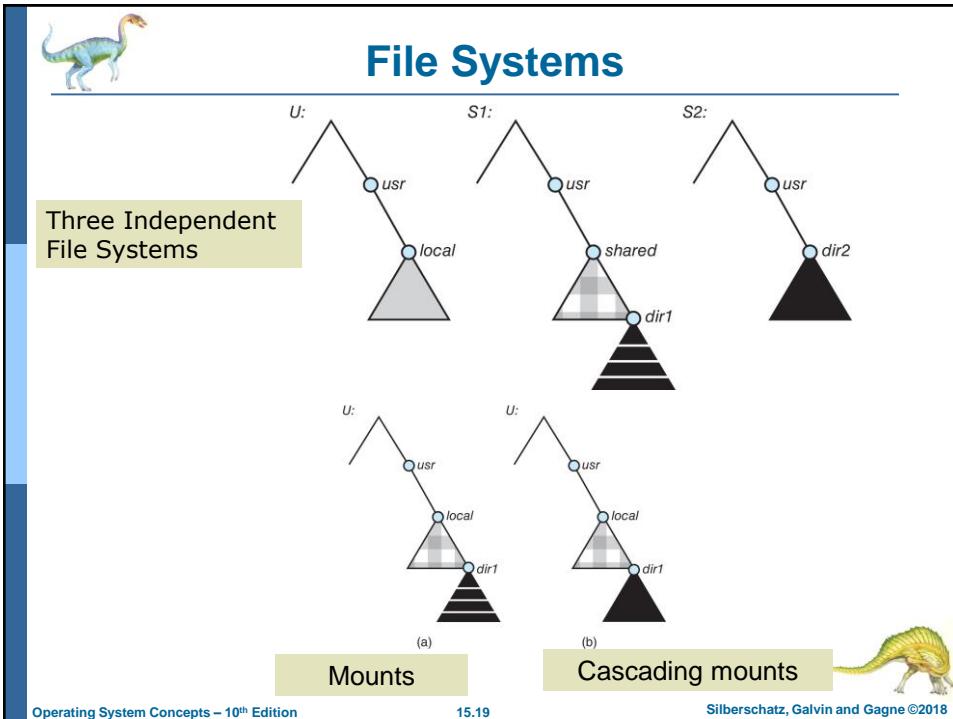
- Interconnected workstations viewed as a set of independent machines with independent file systems, which allows sharing among these file systems in a transparent manner
  - A remote directory is mounted over a local file system directory
    - ▶ The mounted directory looks like an integral subtree of the local file system, replacing the subtree descending from the local directory
  - Specification of the remote directory for the mount operation is nontransparent; the host name of the remote directory has to be provided
    - ▶ Files in the remote directory can then be accessed in a transparent manner
  - Subject to access-rights accreditation, potentially any file system (or directory within a file system), can be mounted remotely on top of any local directory



## NFS (Cont.)

- NFS is designed to operate in a heterogeneous environment of different machines, operating systems, and network architectures; the NFS specifications independent of these media
- This independence is achieved through the use of RPC primitives built on top of an External Data Representation (XDR) protocol used between two implementation-independent interfaces
- The NFS specification distinguishes between the services provided by a mount mechanism and the actual remote-file-access services





## NFS Mount Protocol

- Establishes initial logical connection between server and client
- Mount operation includes name of remote directory to be mounted and name of server machine storing it
  - Mount request is mapped to corresponding RPC and forwarded to mount server running on server machine
  - Export list – specifies local file systems that server exports for mounting, along with names of machines that are permitted to mount them
- Following a mount request that conforms to its export list, the server returns a file handle—a key for further accesses
- File handle – a file-system identifier, and an inode number to identify the mounted directory within the exported file system
- The mount operation changes only the user's view and does not affect the server side





## NFS Protocol

- Provides a set of remote procedure calls for remote file operations. The procedures support the following operations:
  - searching for a file within a directory
  - reading a set of directory entries
  - manipulating links and directories
  - accessing file attributes
  - reading and writing files
- NFS servers are **stateless**; each request has to provide a full set of arguments (NFS V4 is newer, less used – very different, stateful)
- Modified data must be committed to the server's disk before results are returned to the client (lose advantages of caching)
- The NFS protocol does not provide concurrency-control mechanisms



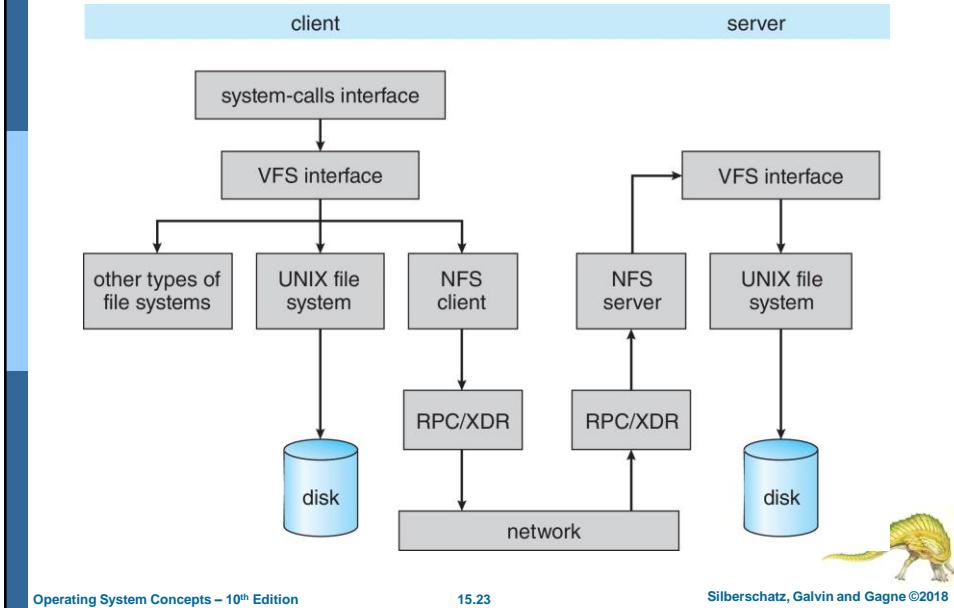
## Three Major Layers of NFS Architecture

- UNIX file-system interface (based on the **open**, **read**, **write**, and **close** calls, and **file descriptors**)
- Virtual File System (VFS) layer – distinguishes local files from remote ones, and local files are further distinguished according to their file-system types
  - The VFS activates file-system-specific operations to handle local requests according to their file-system types
  - Calls the NFS protocol procedures for remote requests
- NFS service layer – bottom layer of the architecture
  - Implements the NFS protocol





## Schematic View of NFS Architecture



## NFS Path-Name Translation

- Performed by breaking the path into component names and performing a separate NFS lookup call for every pair of component name and directory vnode
- To make lookup faster, a directory name lookup cache on the client's side holds the vnodes for remote directory names



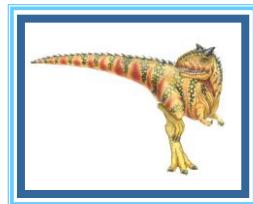


## NFS Remote Operations

- Nearly one-to-one correspondence between regular UNIX system calls and the NFS protocol RPCs (except opening and closing files)
- NFS adheres to the remote-service paradigm, but employs buffering and caching techniques for the sake of performance
- File-blocks cache – when a file is opened, the kernel checks with the remote server whether to fetch or revalidate the cached attributes
  - Cached file blocks are used only if the corresponding cached attributes are up to date
- File-attribute cache – the attribute cache is updated whenever new attributes arrive from the server
- Clients do not free delayed-write blocks until the server confirms that the data have been written to disk



## End of Chapter 4.3



## Chapter 4: File System Management and I/O System

### 4.4. Mass-Storage Systems



GV: Nguyễn Thị Thanh Vân

Operating System Concepts – 10<sup>th</sup> Edition

Silberschatz, Galvin and Gagne ©2018



### Mass-Storage Systems

- Overview of Mass Storage Structure
- HDD Scheduling
- NVM Scheduling
- Error Detection and Correction
- Storage Device Management
- Swap-Space Management
- Storage Attachment
- RAID Structure



Operating System Concepts – 10<sup>th</sup> Edition

11.2

Silberschatz, Galvin and Gagne ©2018



## Objectives

- Describe the physical structure of secondary storage devices and the effect of a device's structure on its uses
- Explain the performance characteristics of mass-storage devices
- Evaluate I/O scheduling algorithms
- Discuss operating-system services provided for mass storage, including RAID



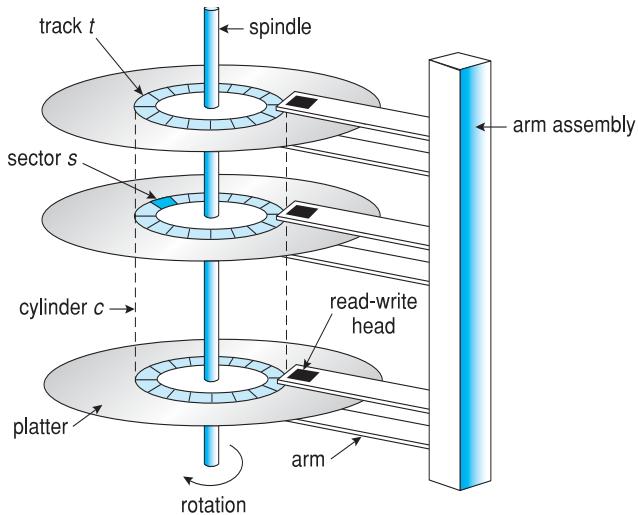
## Overview of Mass Storage Structure

- Bulk of secondary storage for modern computers is **hard disk drives (HDDs)** and **nonvolatile memory (NVM)** devices
- **HDDs** spin platters of magnetically-coated material under moving read-write heads
  - Drives rotate at 60 to 250 times per second
  - **Transfer rate** is rate at which data flow between drive and computer
  - **Positioning time (random-access time)** is time to move disk arm to desired cylinder (**seek time**) and time for desired sector to rotate under the disk head (**rotational latency**)
  - **Head crash** results from disk head making contact with the disk surface -- That's bad
- Disks can be removable





## Moving-head Disk Mechanism



Operating System Concepts – 10<sup>th</sup> Edition

11.5

Silberschatz, Galvin and Gagne ©2018



## Hard Disk Drives

- Platters range from .85" to 14" (historically)
  - Commonly 3.5", 2.5", and 1.8"
- Range from 30GB to 3TB per drive
- Performance
  - Transfer Rate – theoretical – 6 Gb/sec
  - Effective Transfer Rate – real – 1Gb/sec
  - Seek time from 3ms to 12ms – 9ms common for desktop drives
  - Average seek time measured or calculated based on 1/3 of tracks
  - Latency based on spindle speed
    - ▶  $1 / (\text{RPM} / 60) = 60 / \text{RPM}$
  - Average latency =  $\frac{1}{2}$  latency



Operating System Concepts – 10<sup>th</sup> Edition

11.6

Silberschatz, Galvin and Gagne ©2018



## Hard Disk Performance

- **Access Latency** = **Average access time** = average seek time + average latency (milliseconds)
  - For fastest disk 3ms + 2ms = 5ms
  - For slow disk 9ms + 5.56ms = 14.56ms
- Average I/O time = average access time + (amount to transfer / transfer rate) + controller overhead
- For example to transfer a 4KB block on a 7200 RPM disk with a 5ms average seek time, 1Gb/sec transfer rate with a .1ms controller overhead =
  - 5ms + 4.17ms + 0.1ms + transfer time =
  - Transfer time =  $4\text{KB} / 1\text{Gb/s} * 8\text{Gb} / \text{GB} * 1\text{GB} / 1024^2\text{KB} = 32 / (1024^2)$  = 0.031 ms
  - Average I/O time for 4KB block = 9.27ms + .031ms = 9.301ms



## The First Commercial Disk Drive



1956

IBM RAMDAC computer included the IBM Model 350 disk storage system

5M (7 bit) characters

50 x 24" platters

Access time = < 1 second





## Nonvolatile Memory (NVM) Devices

- NVM devices are electrical, is composed of a controller and lash NAND die semiconductor chips, which are used to store data.
  - It is used in a disk-drive-like container, called **solid-state disks (SSDs)**
- Other forms include **USB drives** (thumb drive, flash drive), DRAM disk replacements, surface-mounted on motherboards, and main storage in devices like smartphones
- Can be more reliable than HDDs
- More expensive per MB
- Maybe have shorter life span – need careful management
- Less capacity
- But much faster
- Busses can be too slow -> connect directly to PCI for example
- No moving parts, so no seek time or rotational latency



## Nonvolatile Memory Devices

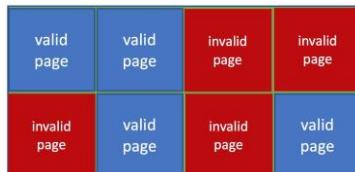
- Have characteristics that present challenges
- Read and written in “page” increments (think sector) but can’t overwrite in place
  - Must first be erased, and erases happen in larger “block” increments
  - Can only be erased a limited number of times before worn out – ~ 100,000
  - Life span measured in **drive writes per day (DWPD)**
    - ▶ A 1TB NAND drive with rating of 5DWPD is expected to have 5TB per day written within warranty period without failing





## NAND Flash Controller Algorithms

- With no overwrite, pages end up with mix of valid and invalid data
- To track which logical blocks are valid, controller maintains **flash translation layer (FTL)** table
- Also implements **garbage collection** to free invalid page space
- Allocates **overprovisioning** to provide working space for GC
- Each cell has lifespan, so **wear leveling** needed to write equally to all cells



NAND block with valid and invalid pages



## Volatile Memory

- DRAM frequently used as mass-storage device
  - Not technically secondary storage because volatile, but can have file systems, be used like very fast secondary storage
- **RAM drives** (with many names, including RAM disks) present as raw block devices, commonly file system formatted
- Computers have buffering, caching via RAM, so why RAM drives?
  - Caches / buffers allocated / managed by programmer, operating system, hardware
  - RAM drives under user control
  - Found in all major operating systems
    - Linux `/dev/ram`, macOS `diskutil` to create them, Linux `/tmp` of file system type `tmpfs`
- Used as high speed temporary storage
  - Programs could share bulk data, quickly, by reading/writing to RAM drive





## Magnetic Tape

**Magnetic tape** was used as an early secondary-storage medium. Although it is nonvolatile and can hold large quantities of data, its access time is slow compared with that of main memory and drives. In addition, random access to magnetic tape is about a thousand times slower than random access to HDDs and about a hundred thousand times slower than random access to SSDs so tapes are not very useful for secondary storage. Tapes are used mainly for backup, for storage of infrequently used information, and as a medium for transferring information from one system to another.

A tape is kept in a spool and is wound or rewound past a read–write head. Moving to the correct spot on a tape can take minutes, but once positioned, tape drives can read and write data at speeds comparable to HDDs. Tape capacities vary greatly, depending on the particular kind of tape drive, with current capacities exceeding several terabytes. Some tapes have built-in compression that can more than double the effective storage. Tapes and their drivers are usually categorized by width, including 4, 8, and 19 millimeters and 1/4 and 1/2 inch. Some are named according to technology, such as LTO-6 (Figure 11.5) and SDLT.

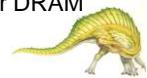


Figure 11.5 An LTO-6 Tape drive with tape cartridge inserted.



## Disk Attachment

- Host-attached storage accessed through I/O ports talking to **I/O busses**
- Several busses available, including **advanced technology attachment (ATA)**, **serial ATA (SATA)**, **eSATA**, **serial attached SCSI (SAS)**, **universal serial bus (USB)**, and **fibre channel (FC)**.
- Most common is SATA
- Because NVM much faster than HDD, new fast interface for NVM called **NVM express (NVMe)**, connecting directly to PCI bus
- Data transfers on a bus carried out by special electronic processors called **controllers** (or **host-bus adapters, HBAs**)
  - Host controller on the computer end of the bus, device controller on device end
  - Computer places command on host controller, using memory-mapped I/O ports
    - Host controller sends messages to device controller
    - Data transferred via DMA between device and computer DRAM





## Address Mapping

- Disk drives are addressed as large 1-dimensional arrays of **logical blocks**, where the logical block is the smallest unit of transfer
  - Low-level formatting creates **logical blocks** on physical media
- The 1-dimensional array of logical blocks is mapped into the sectors of the disk sequentially
  - Sector 0 is the first sector of the first track on the outermost cylinder
  - Mapping proceeds in order through that track, then the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost
  - Logical to physical address should be easy
    - Except for bad sectors
    - Non-constant # of sectors per track via constant angular velocity



## HDD Scheduling

- The operating system is responsible for using hardware efficiently — for the disk drives, this means having a fast access time and disk bandwidth
- Minimize seek time
- Seek time  $\approx$  seek distance
- Disk **bandwidth** is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer





## Disk Scheduling (Cont.)

- There are many sources of disk I/O request
  - OS
  - System processes
  - Users processes
- I/O request includes input or output mode, disk address, memory address, number of sectors to transfer
- OS maintains queue of requests, per disk or device
- Idle disk can immediately work on I/O request, busy disk means work must queue
  - Optimization algorithms only make sense when a queue exists



## Disk Scheduling (Cont.)

- In the past, operating system responsible for queue management, disk drive head scheduling
  - Now, built into the storage devices, controllers
  - Just provide Logical Block Addressing (LBA), handle sorting of requests
    - ▶ Some of the algorithms they use described next
- Note that drive controllers have small buffers and can manage a queue of I/O requests (of varying “depth”)
- Several algorithms exist to schedule the servicing of disk I/O requests
- The analysis is true for one or many platters
- We illustrate scheduling algorithms with a request queue (0-199)

98, 183, 37, 122, 14, 124, 65, 67

Head pointer 53

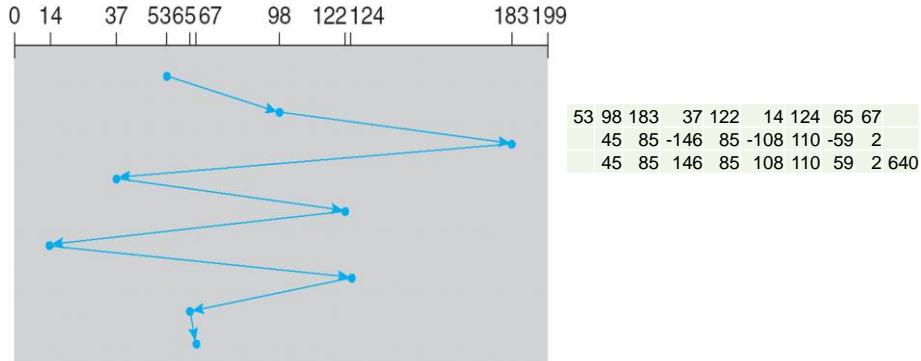




## FCFS - first-come, first-served

Illustration shows total head movement of 640 cylinders

queue = 98, 183, 37, 122, 14, 124, 65, 67  
head starts at 53



Operating System Concepts – 10<sup>th</sup> Edition

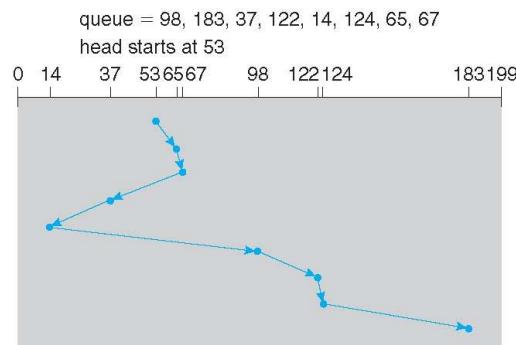
11.20

Silberschatz, Galvin and Gagne ©2018



## SSTF

- Shortest Seek Time First selects the request with the minimum seek time from the current head position
- SSTF scheduling is a form of SJF scheduling; may cause starvation of some requests
- Illustration shows total head movement of 236 cylinders



Operating System Concepts – 10<sup>th</sup> Edition

11.21

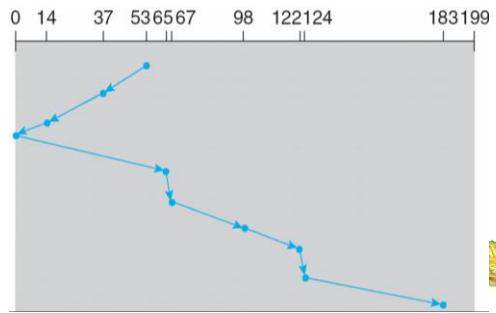
Silberschatz, Galvin and Gagne ©2018



## SCAN

- The disk arm starts at one end of the disk, and moves toward the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues.
- SCAN algorithm** Sometimes called the **elevator algorithm**
- Illustration shows total head movement of 208 cylinders
- But note that if requests are uniformly dense, largest density at other end of disk and those wait the longest

queue = 98, 183, 37, 122, 14, 124, 65, 67  
head starts at 53



Operating System Concepts – 10<sup>th</sup> Edition

11.22

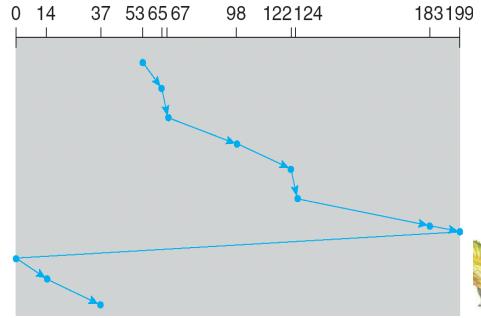
Silberschatz, Galvin and Gagne ©2018



## C-SCAN

- Provides a more uniform wait time than SCAN
- The head moves from one end of the disk to the other, servicing requests as it goes
  - When it reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip
- Treats the cylinders as a circular list that wraps around from the last cylinder to the first one
- Total number of cylinders?

queue = 98, 183, 37, 122, 14, 124, 65, 67  
head starts at 53



Operating System Concepts – 10<sup>th</sup> Edition

11.23

Silberschatz, Galvin and Gagne ©2018



## Selecting a Disk-Scheduling Algorithm

- SSTF is common and has a natural appeal
- SCAN and C-SCAN perform better for systems that place a heavy load on the disk
  - Less starvation, but still possible
- To avoid starvation Linux implements **deadline** scheduler
  - Maintains separate read and write queues, gives read priority
    - Because processes more likely to block on read than write
  - Implements four queues: 2 x read and 2 x write
    - 1 read and 1 write queue sorted in LBA order, essentially implementing C-SCAN
    - 1 read and 1 write queue sorted in FCFS order
    - All I/O requests sent in batch sorted in that queue's order
    - After each batch, checks if any requests in FCFS older than configured age (default 500ms)
      - If so, Logical Block Addressing (LBA) queue containing that request is selected for next batch of I/O
- In Red Hat (RHEL 7) also **NOOP** and **completely fair queueing** scheduler (**CFQ**) also available, defaults vary by storage device



## NVM Scheduling

- No disk heads or rotational latency but still room for optimization
- In RHEL 7 **NOOP** (no scheduling) is used but adjacent LBA requests are combined
  - NVM best at random I/O, HDD at sequential
  - Throughput can be similar
  - **Input/Output operations per second (IOPS)** much higher with NVM (hundreds of thousands vs hundreds)
  - But **write amplification** (one write, causing garbage collection and many read/writes) can decrease the performance advantage





## Error Detection and Correction

- Fundamental aspect of many parts of computing (memory, networking, storage)
- **Error detection** determines if there a problem has occurred (for example a bit flipping)
  - If detected, can halt the operation
  - Detection frequently done via parity bit
- Parity is one form of **checksum** – uses modular arithmetic to compute, store, compare values of fixed-length words
- Another error-detection method common in networking is **cyclic redundancy check (CRC)** which uses hash function to detect multiple-bit errors
- **Error-correction code (ECC)** not only detects, but can correct some errors
  - Soft errors correctable, hard errors detected but not corrected



## Storage Device Management

- **Low-level formatting**, or **physical formatting** — Dividing a disk into sectors that the disk controller can read and write
  - Each sector can hold header information, plus data, plus error correction code (**ECC**)
  - Usually 512 bytes of data but can be selectable
- To use a disk to hold files, the operating system still needs to record its own data structures on the disk
  - **Partition** the disk into one or more groups of cylinders, each treated as a logical disk
  - **Logical formatting** or “making a file system”
  - To increase efficiency most file systems group blocks into **clusters**
    - ▶ Disk I/O done in blocks
    - ▶ File I/O done in clusters





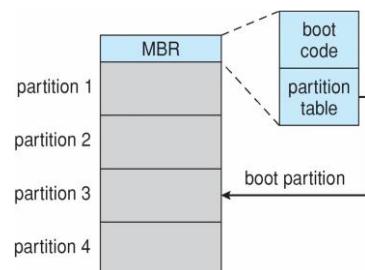
## Storage Device Management (cont.)

- **Root partition** contains the OS, other partitions can hold other Oses, other file systems, or be raw
  - **Mounted** at boot time
  - Other partitions can mount automatically or manually
- At mount time, file system consistency checked
  - Is all metadata correct?
    - ▶ If not, fix it, try again
    - ▶ If yes, add to mount table, allow access
- Boot block can point to boot volume or boot loader set of blocks that contain enough code to know how to load the kernel from the file system
  - Or a boot management program for multi-os booting



## Device Storage Management (Cont.)

- Raw disk access for apps that want to do their own block management, keep OS out of the way (databases for example)
- Boot block initializes system
  - The bootstrap is stored in ROM, firmware
  - **Bootstrap loader** program stored in boot blocks of boot partition
- Methods such as **sector sparing** used to handle bad blocks



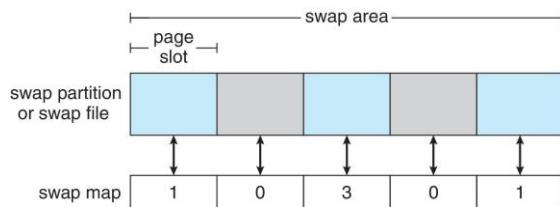
Booting from secondary storage in Windows





## Swap-Space Management

- Used for moving entire processes (swapping), or pages (paging), from DRAM to secondary storage when DRAM not large enough for all processes
- Operating system provides **swap space management**
  - Secondary storage slower than DRAM, so important to optimize performance
  - Usually multiple swap spaces possible – decreasing I/O load on any given device
  - Best to have dedicated devices
  - Can be in raw partition or a file within a file system (convenience of adding)
  - Data structures for swapping on Linux systems:



## Storage Attachment

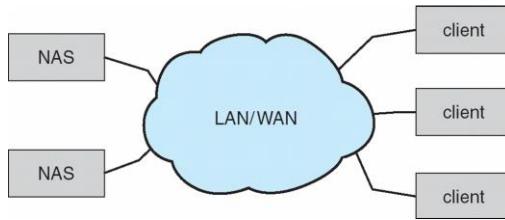
- Computers access storage in three ways
  - Host-attached
  - Network-attached
  - Cloud
- Host attached access through local I/O ports, using one of several technologies
  - To attach many devices, use storage busses such as USB, firewire, thunderbolt
  - High-end systems use **fibre channel (FC)**
    - High-speed serial architecture using fibre or copper cables
    - Multiple hosts and storage devices can connect to the FC fabric





## Network-Attached Storage

- Network-attached storage (**NAS**) is storage made available over a network rather than over a local connection (such as a bus)
  - Remotely attaching to file systems
- NFS and CIFS are common protocols
- Implemented via remote procedure calls (RPCs) between host and storage over typically TCP or UDP on IP network
- **iSCSI** protocol uses IP network to carry the SCSI protocol
  - Remotely attaching to devices (blocks)



## Cloud Storage

- Similar to NAS, provides access to storage across a network
  - Unlike NAS, accessed over the Internet or a WAN to remote data center
- NAS presented as just another file system, while cloud storage is API based, with programs using the APIs to provide access
  - Examples include Dropbox, Amazon S3, Microsoft OneDrive, Apple iCloud
  - Use APIs because of latency and failure scenarios (NAS protocols wouldn't work well)





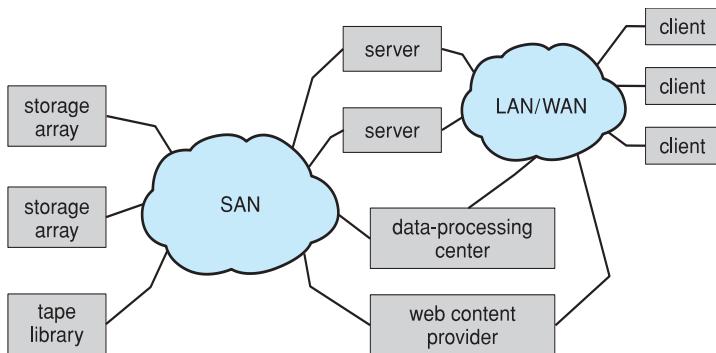
## Storage Array

- Can just attach disks, or arrays of disks
- Avoids the NAS drawback of using network bandwidth
- Storage Array has controller(s), provides features to attached host(s)
  - Ports to connect hosts to array
  - Memory, controlling software (sometimes NVRAM, etc.)
  - A few to thousands of disks
  - RAID, hot spares, hot swap (discussed later)
  - Shared storage -> more efficiency
  - Features found in some file systems
    - ▶ Snapshots, clones, thin provisioning, replication, deduplication, etc



## Storage Area Network

- Common in large storage environments
- Multiple hosts attached to multiple storage arrays – flexible





## Storage Area Network (Cont.)

- SAN is one or more storage arrays
  - Connected to one or more Fibre Channel switches or **InfiniBand (IB)** network
- Hosts also attach to the switches
- Storage made available via **LUN Masking** from specific arrays to specific servers
- Easy to add or remove storage, add new host and allocate it storage
- Why have separate storage networks and communications networks?
  - Consider iSCSI, FCOE



A Storage Array



## RAID Structure

- **RAID – redundant array of inexpensive disks**
  - multiple disk drives provides reliability via **redundancy**
- Increases the **mean time to failure**
- **Mean time to repair** – exposure time when another failure could cause data loss
- **Mean time to data loss** based on above factors
- If mirrored disks fail independently, consider disk with 1300,000 **mean time to failure** and 10 hour mean time to repair
  - Mean time to data loss is  $100,000^2 / (2 * 10) = 500 * 10^6$  hours, or 57,000 years!
- Frequently combined with **NVRAM** to improve write performance
- Several improvements in disk-use techniques involve the use of multiple disks working cooperatively



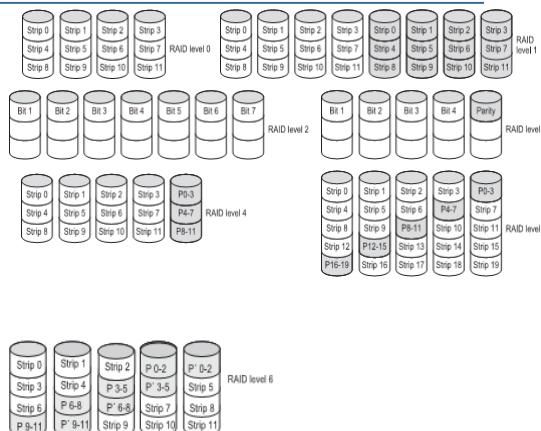
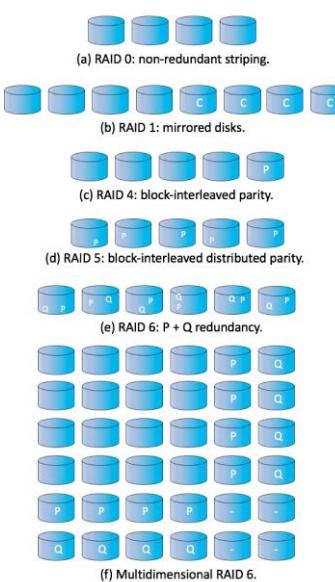


## RAID (Cont.)

- Disk **striping** uses a group of disks as one storage unit
- RAID is arranged into six different levels
- RAID schemes improve performance and improve the reliability of the storage system by storing redundant data
  - **Mirroring or shadowing (RAID 1)** keeps duplicate of each disk
  - Striped mirrors (**RAID 1+0**) or mirrored stripes (**RAID 0+1**) provides high performance and high reliability
  - **Block interleaved parity (RAID 4, 5, 6)** uses much less redundancy
- RAID within a storage array can still fail if the array fails, so automatic **replication** of the data between arrays is common
- Frequently, a small number of **hot-spare** disks are left unallocated, automatically replacing a failed disk and having data rebuilt onto them

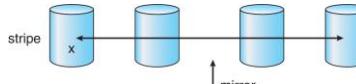


## RAID Levels

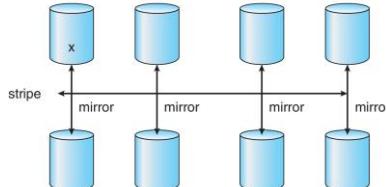




## RAID (0 + 1) and (1 + 0)



a) RAID 0 + 1 with a single disk failure.



b) RAID 1 + 0 with a single disk failure.



## Other Features

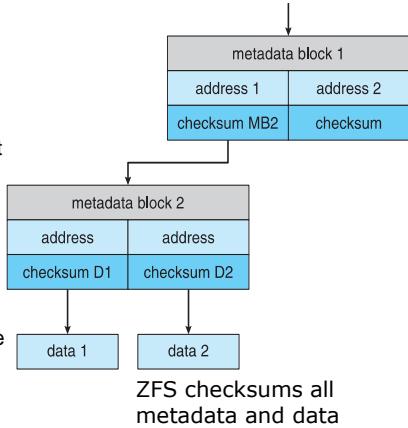
- Regardless of where RAID implemented, other useful features can be added
- **Snapshot** is a view of file system before a set of changes take place (i.e., at a point in time)
  - More in Ch 12
- Replication is automatic duplication of writes between separate sites
  - For redundancy and disaster recovery
  - Can be synchronous or asynchronous
- Hot spare disk is unused, automatically used by RAID production if a disk fails to replace the failed disk and rebuild the RAID set if possible
  - Decreases mean time to repair



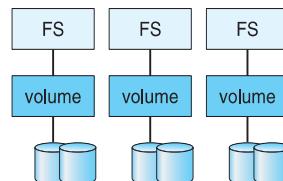


## Extensions

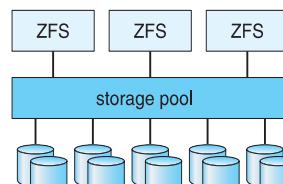
- RAID alone does not prevent or detect data corruption or other errors, just disk failures
- Solaris ZFS adds **checksums** of all data and metadata
- Checksums kept with pointer to object, to detect if object is the right one and whether it changed
- Can detect and correct data and metadata corruption
- ZFS also removes volumes, partitions
  - Disks allocated in **pools**
  - Filesystems with a pool share that pool, use and release space like `malloc()` and `free()` memory allocate / release calls



## Traditional and Pooled Storage



(a) Traditional volumes and file systems.



(b) ZFS and pooled storage.



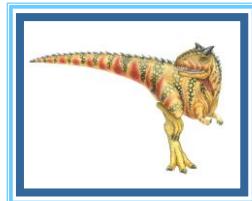


## Object Storage

- General-purpose computing, file systems not sufficient for very large scale
- Another approach – start with a storage pool and place objects in it
  - Object just a container of **data**
  - No way to navigate the pool to find objects (no directory structures, few services)
  - Computer-oriented, not user-oriented
- Typical sequence
  - Create an object within the pool, receive an object ID
  - Access object via that ID
  - Delete object via that ID
- Object storage management software like **Hadoop file system (HDFS)** and **Ceph** determine where to store objects, manages protection
  - Typically by storing N copies, across N systems, in the object storage cluster
  - **Horizontally scalable**
  - **Content addressable, unstructured**



## End of Chapter 4.4



## Chapter 4: File System Management & I/O System

### 4.5. I/O Systems



GV: Nguyễn Thị Thanh Vân

Operating System Concepts – 10<sup>th</sup> Edition

Silberschatz, Galvin and Gagne ©2018



### Chapter 12: I/O Systems

- Overview
- I/O Hardware
- Application I/O Interface
- Kernel I/O Subsystem
- Transforming I/O Requests to Hardware Operations
- STREAMS
- Performance



Operating System Concepts – 10<sup>th</sup> Edition

12.2

Silberschatz, Galvin and Gagne ©2018



## Objectives

- Explore the structure of an operating system's I/O subsystem
- Discuss the principles and complexities of I/O hardware
- Explain the performance aspects of I/O hardware and software



## Overview

- I/O management is a major component of operating system design and operation
  - Important aspect of computer operation
  - I/O devices vary greatly
  - Various methods to control them
  - Performance management
  - New types of devices frequent
- Ports, busses, device controllers connect to various devices
- **Device drivers** encapsulate device details
  - Present uniform device-access interface to I/O subsystem



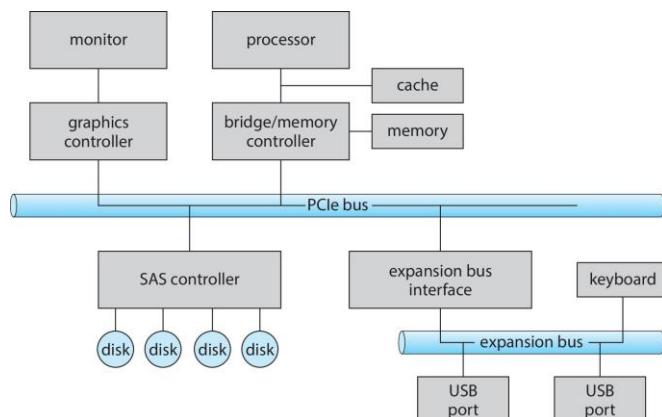


## I/O Hardware

- Incredible variety of I/O devices
  - Storage
  - Transmission
  - Human-interface
- Common concepts – signals from I/O devices interface with computer
  - **Port** – connection point for device
  - **Bus - daisy chain** or shared direct access
    - PCI bus common in PCs and servers, PCI Express (**PCIe**)
    - **expansion bus** connects relatively slow devices
    - **Serial-attached SCSI (SAS)** common disk interface
  - **Controller (host adapter)** – electronics that operate port, bus, device
    - Sometimes integrated
    - Sometimes separate circuit board (host adapter)
    - Contains processor, microcode, private memory, bus controller, etc.
      - Some talk to per-device controller with bus controller, microcode, memory, etc.



## A Typical PC Bus Structure





## I/O Hardware (Cont.)

- **Fibre channel (FC)** is complex controller, usually separate circuit board (**host-bus adapter, HBA**) plugging into bus
- I/O instructions control devices
- Devices usually have registers where device driver places commands, addresses, and data to write, or read data from registers after command execution
  - Data-in register, data-out register, status register, control register
  - Typically 1-4 bytes, or FIFO buffer
- Devices have addresses, used by
  - Direct I/O instructions
  - **Memory-mapped I/O**
    - Device data and command registers mapped to processor address space
    - Especially for large address spaces (graphics)



## Device I/O Port Locations on PCs (partial)

I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)





## Polling

- The complete protocol for interaction between the host and a controller can be used by handshaking. For each byte of I/O
  - 1. The host repeatedly reads the busy bit until that bit becomes clear.
  - 2. Host sets read or write bit and if write copies data into data-out register
  - 3. Host sets command-ready bit
  - 4. Controller sets busy bit, and executes transfer
  - 5. Controller clears busy bit, error bit, command-ready bit when transfer done
- In step 1, host is **busy-wait or polling**: cycle to wait for I/O from device
  - Reasonable if device is fast
  - But inefficient if device slow
  - CPU switches to other tasks?
    - ▶ But if miss a cycle data overwritten / lost



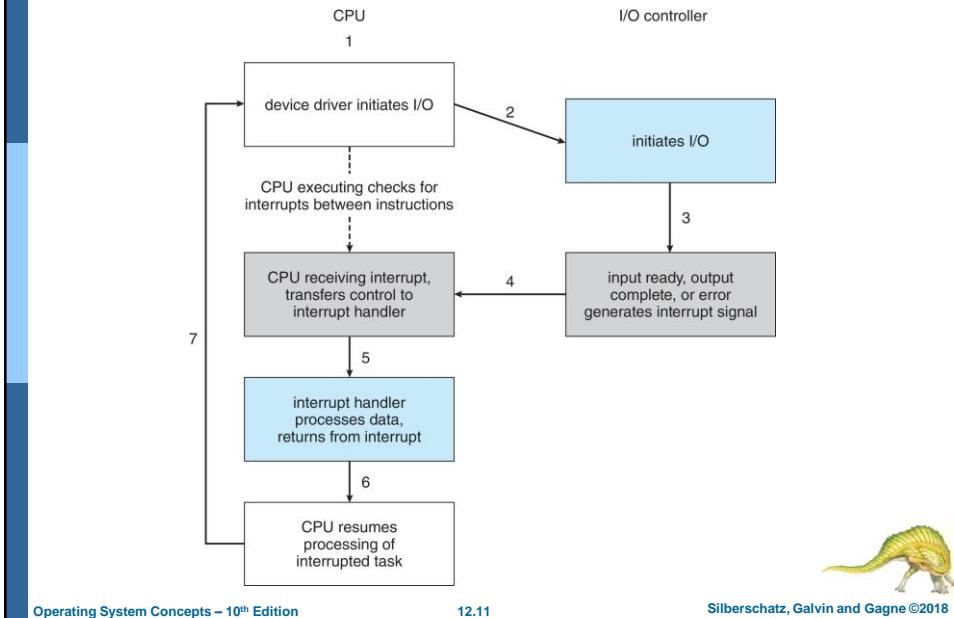
## Interrupts

- Polling can happen in 3 instruction cycles
  - Read status, logical-and to extract status bit, branch if not zero
  - How to be more efficient if non-zero infrequently?
- CPU **Interrupt-request line** triggered by I/O device
  - Checked by processor after each instruction
- **Interrupt handler** receives interrupts
  - **Maskable** to ignore or delay some interrupts
- **Interrupt vector** to dispatch interrupt to correct handler
  - Context switch at start and end
  - Based on priority
  - Some **nonmaskable**
  - Interrupt chaining if more than one device at same interrupt number





## Interrupt-Driven I/O Cycle



## Interrupts (Cont.)

- Interrupt mechanism also used for **exceptions**
  - Terminate process, crash system due to hardware error
- Page fault executes when memory access error
- System call executes via **trap** to trigger kernel to execute request
- Multi-CPU systems can process interrupts concurrently
  - If operating system designed to handle it
- Used for time-sensitive processing, frequent, must be fast





## Latency

- Stressing interrupt management because even single-user systems manage hundreds of interrupts per second and servers hundreds of thousands per second
- For example, a quiet macOS desktop generated 23,000 interrupts over 10 seconds

```
Fri Nov 25 13:55:59          0:00:10
SCHEDULER      INTERRUPTS
-----
total_samples      13      22998
delays < 10 usecs    12     16243
delays < 20 usecs     1      5312
delays < 30 usecs     0      473
delays < 40 usecs     0      590
delays < 50 usecs     0       61
delays < 60 usecs     0      317
delays < 70 usecs     0       2
delays < 80 usecs     0       0
delays < 90 usecs     0       0
delays < 100 usecs    0       0
total < 100 usecs    13     22998
```



## Intel Pentium Processor Event-Vector Table

- Interrupt vector: contains the memory addresses of specialized interrupt handlers. => to reduce the need for a single interrupt handler to search all possible sources of interrupts to determine which one needs service.
- Ex: Interrupt vector for the Intel Pentium processor

vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts



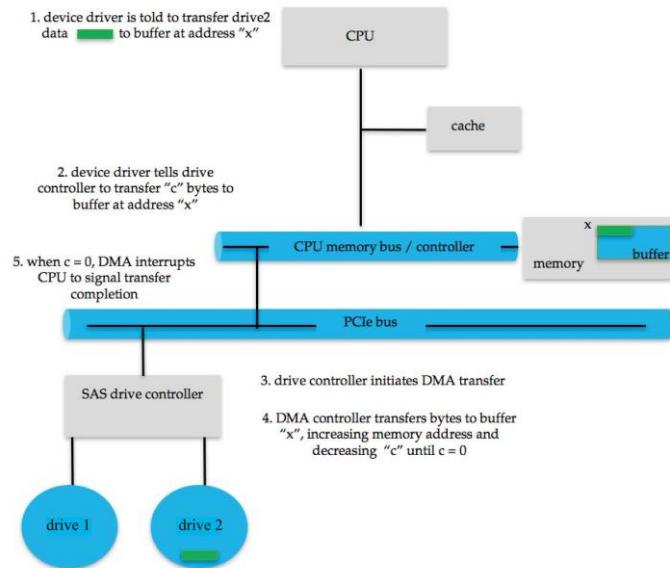


## Direct Memory Access

- DMA is used to avoid **programmed I/O** (one byte at a time) for large data movement
- Requires **DMA** controller: Proceeds to operate the memory bus directly
  - transfer data directly between I/O device and memory (without CPU)
- Handshaking between the DMA controller and the device controller
  - is performed via a pair of wires called DMA-request and DMA- acknowledge
- OS writes DMA command block into memory
  - Source and destination addresses
  - Read or write mode
  - Count of bytes
  - Writes location of command block to DMA controller
  - Bus mastering of DMA controller – grabs bus from CPU
    - **Cycle stealing** from CPU but still much more efficient
  - When done, interrupts to signal completion
- Version that is aware of virtual addresses can be even more efficient – **DVMA - direct virtual memory access**



## Six Step Process to Perform DMA Transfer



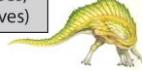
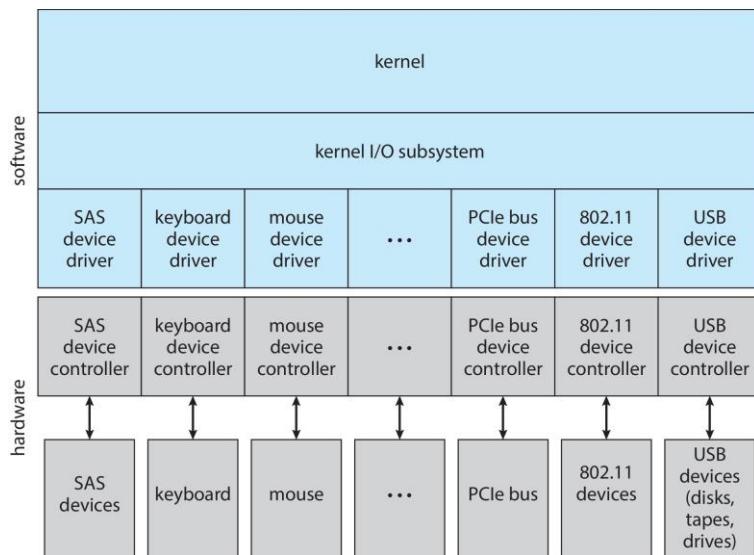


## Application I/O Interface

- I/O system calls encapsulate device behaviors in generic classes
- Device-driver layer hides differences among I/O controllers from kernel
- New devices talking already-implemented protocols need no extra work
- Each OS has its own I/O subsystem structures and device driver frameworks
- Devices vary in many dimensions
  - **Character-stream** or **block**: transfers bytes one by one
  - **Sequential** or **random-access**: transfers data in a fixed order determined
  - **Synchronous** or **asynchronous** (or both)
  - **Sharable** or **dedicated**
  - **Speed of operation**
  - **read-write**, **read only**, or **write only**



## A Kernel I/O Structure





## Characteristics of I/O Devices

Devices vary on many dimensions

aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only read-write	CD-ROM graphics controller disk



## Characteristics of I/O Devices (Cont.)

- Subtleties of devices handled by device drivers
- I/O devices can be grouped by the OS into
  - Block I/O
  - Character I/O (Stream)
  - Memory-mapped file access
  - Network sockets
- For direct manipulation of I/O device specific characteristics, usually an escape / back door
  - Unix `ioctl()` call to send arbitrary bits to a device control register and data to device data register
- UNIX and Linux use tuple of “major” and “minor” device numbers to identify type and instance of devices (here major 8 and minors 0-4)

```
% ls -l /dev/sda*
```

```
brw-rw---- 1 root disk 8, 0 Mar 16 09:18 /dev/sda
brw-rw---- 1 root disk 8, 1 Mar 16 09:18 /dev/sda1
brw-rw---- 1 root disk 8, 2 Mar 16 09:18 /dev/sda2
brw-rw---- 1 root disk 8, 3 Mar 16 09:18 /dev/sda3
```





## Block and Character Devices

- Block devices include disk drives
  - Commands include read, write, seek
  - **Raw I/O, direct I/O**, or file-system access
  - Memory-mapped file access possible
    - File mapped to virtual memory and clusters brought via demand paging
  - DMA
- Character devices include keyboards, mice, serial ports
  - Commands include `get()`, `put()`
  - Libraries layered on top allow line editing



## Network Devices

- Varying enough from block and character to have own interface
- Linux, Unix, Windows and many others include **socket** interface
  - Separates network protocol from network operation
  - Includes `select()` functionality
- Approaches vary widely (pipes, FIFOs, streams, queues, mailboxes)





## Clocks and Timers

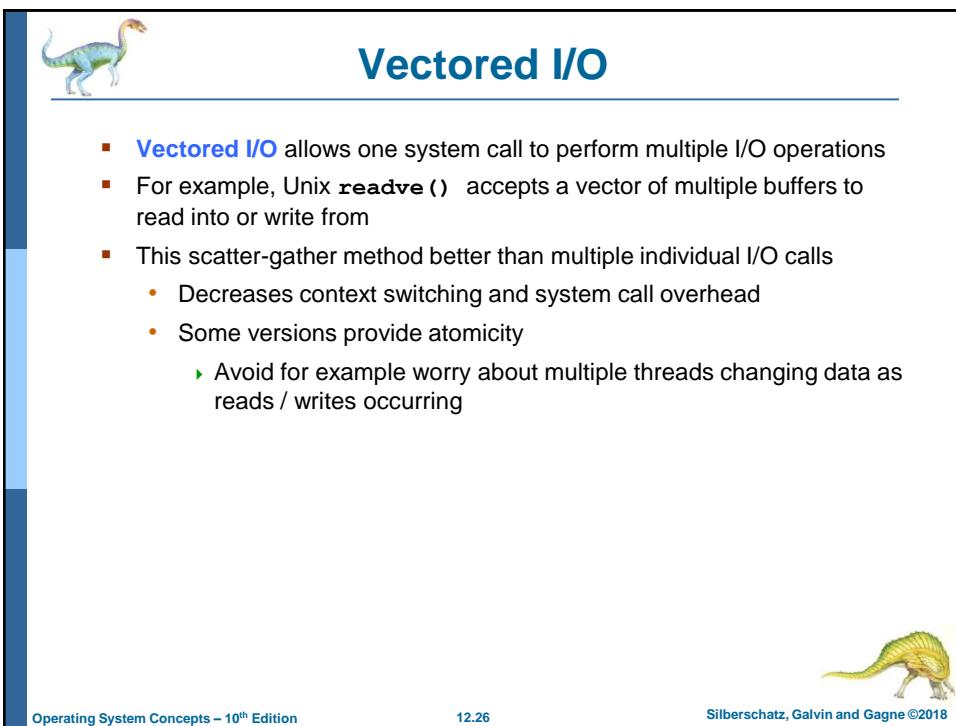
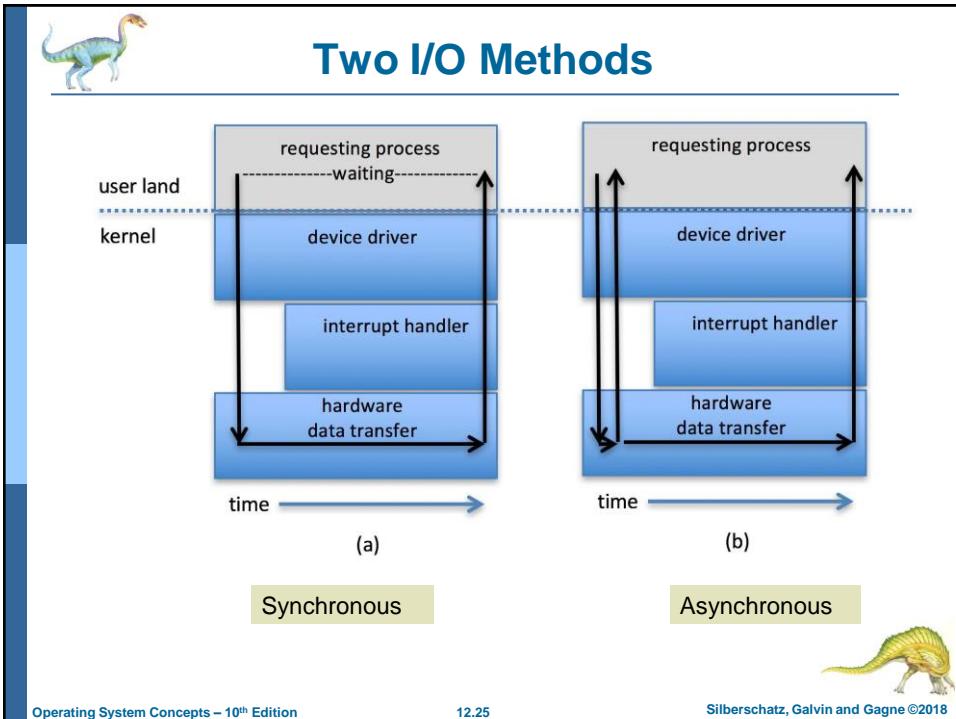
- Provide current time, elapsed time, timer
- Normal resolution about 1/60 second
- Some systems provide higher-resolution timers
- **Programmable interval timer** used for timings, periodic interrupts
- `ioctl()` (on UNIX) covers odd aspects of I/O such as clocks and timers



## Nonblocking and Asynchronous I/O

- **Blocking** - process suspended until I/O completed
  - Easy to use and understand
  - Insufficient for some needs
- **Nonblocking** - I/O call returns as much as available
  - User interface, data copy (buffered I/O)
  - Implemented via multi-threading
  - Returns quickly with count of bytes read or written
  - `select()` to find if data ready then `read()` or `write()` to transfer
- **Asynchronous** - process runs while I/O executes
  - Difficult to use
  - I/O subsystem signals process when I/O completed







## Kernel I/O Subsystem

- **Scheduling**
- **Buffering**
- **Caching**
- **Spooling and device Reservation**
- **Error Handling**
- **I/O Protection**
- **Kernel Data structure**
- **Power management**



## Kernel I/O Subsystem

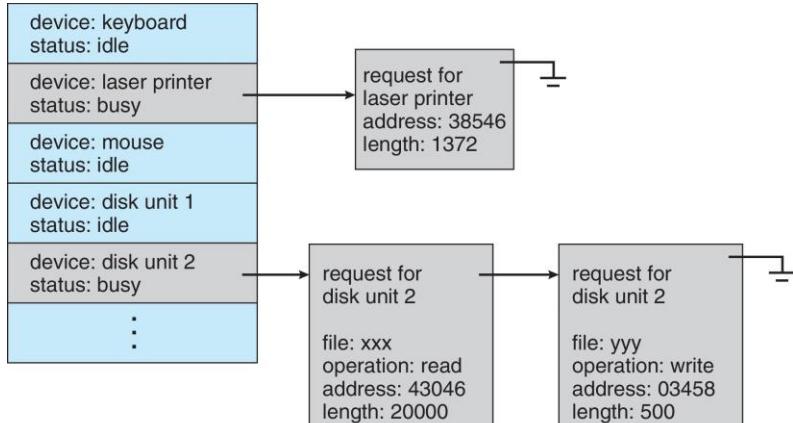
- **Scheduling** a set of I/O requests: determine a good order to execute
  - Some I/O request ordering via per-device queue
  - Some OSs try fairness among processes
  - Some implement Quality Of Service (i.e. IPQOS)
- **Buffering** - store data in memory while transferring between devices
  - To cope with device speed mismatch
  - To cope with device transfer size mismatch
  - To maintain “copy semantics”
  - **Double buffering** – two copies of the data
    - ▶ Kernel and user
    - ▶ Varying sizes
    - ▶ Full / being processed and not-full / being used
    - ▶ Copy-on-write can be used for efficiency in some cases





## Device-status Table

Device-status table: a place that OS might attach the wait queue to keep track of many I/O requests at the same time



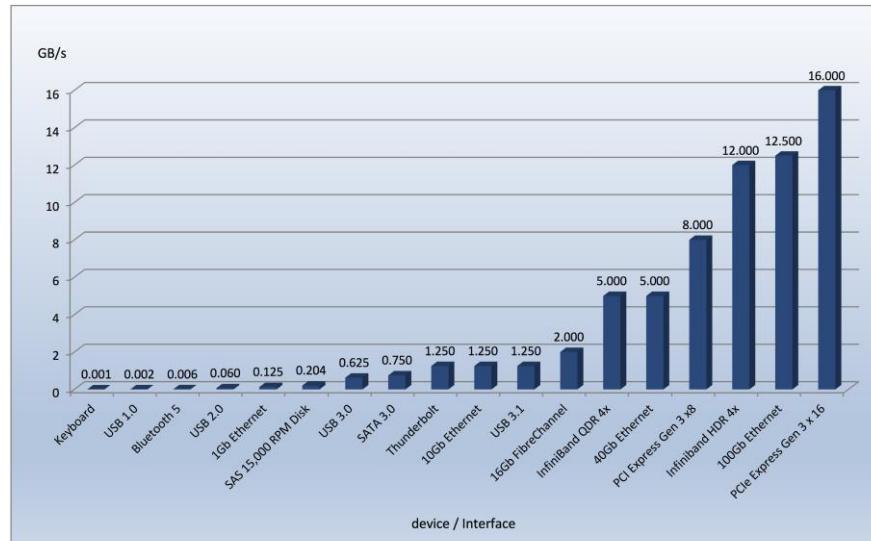
Operating System Concepts – 10<sup>th</sup> Edition

12.29

Silberschatz, Galvin and Gagne ©2018



## Common PC and Data-center I/O devices and Interface Speeds



Operating System Concepts – 10<sup>th</sup> Edition

12.30

Silberschatz, Galvin and Gagne ©2018





## Kernel I/O Subsystem

- **Caching** - faster holding copy of data - Access to the cached copy is more efficient than access to the original.
  - Always just a copy
  - Key to performance
  - Sometimes combined with buffering
- **Spooling** - a buffer holds output for a device
  - If device can serve only one request at a time
  - i.e., Printing
- **Device reservation** - provides exclusive access to a device
  - System calls for allocation and de-allocation
  - Watch out for deadlock



## Error Handling

- OS can recover from disk read, device unavailable, transient write failures
  - Retry a read or write, for example
  - Some systems more advanced – Solaris FMA, AIX
    - ▶ Track error frequencies, stop using device with increasing frequency of retry-able errors
- Most return an error number or code when I/O request fails
- System error logs hold problem reports



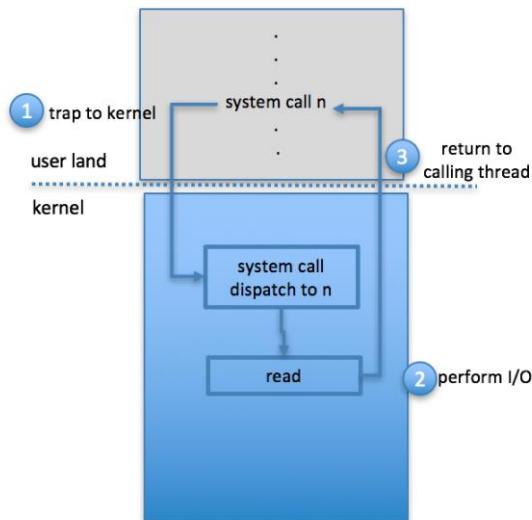


## I/O Protection

- User process may accidentally or purposefully attempt to disrupt normal operation via illegal I/O instructions
  - All I/O instructions defined to be privileged
  - I/O must be performed via system calls
    - Memory-mapped and I/O port memory locations must be protected too



## Use of a System Call to Perform I/O



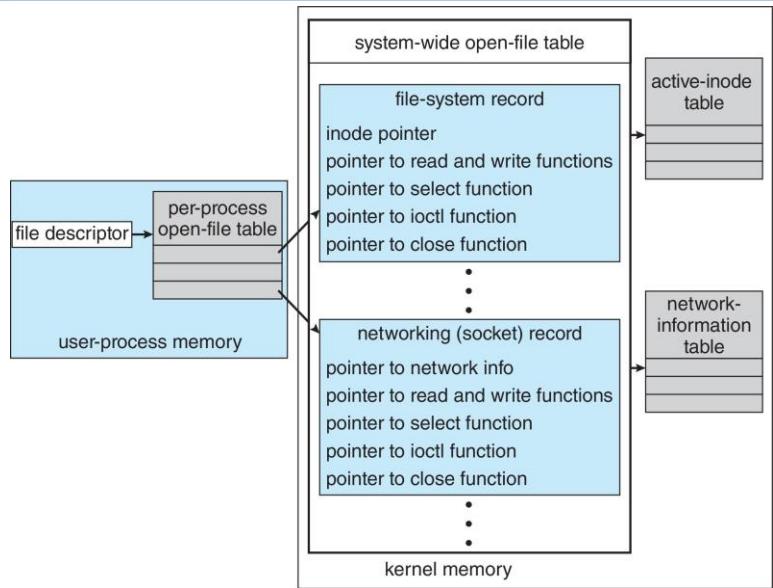


## Kernel Data Structures

- Kernel keeps state info for I/O components, including open file tables, network connections, character device state
- Many, many complex data structures to track buffers, memory allocation, “dirty” blocks
- Some use object-oriented methods and message passing to implement I/O
  - Windows uses message passing
    - Message with I/O information passed from user mode into kernel
    - Message modified as it flows through to device driver and back to process
    - Pros / cons?



## UNIX I/O Kernel Structure





## Power Management

- Not strictly domain of I/O, but much is I/O related
- Computers and devices use electricity, generate heat, frequently require cooling
- OSes can help manage and improve use
  - Cloud computing environments move virtual machines between servers
    - Can end up evacuating whole systems and shutting them down
- Mobile computing has power management as first class OS aspect



## Power Management (Cont.)

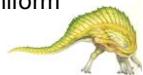
- For example, Android implements
  - Component-level power management
    - Understands relationship between components
    - Build device tree representing physical device topology
    - System bus -> I/O subsystem -> {flash, USB storage}
    - Device driver tracks state of device, whether in use
    - Unused component – turn it off
    - All devices in tree branch unused – turn off branch
  - Wake locks – like other locks but prevent sleep of device when lock is held
  - Power collapse – put a device into very deep sleep
    - Marginal power use
    - Only awake enough to respond to external stimuli (button press, incoming call)
- Modern systems use **advanced configuration and power interface (ACPI)** firmware providing code that runs as routines called by kernel for device discovery, management, error and power management





## Kernel I/O Subsystem Summary

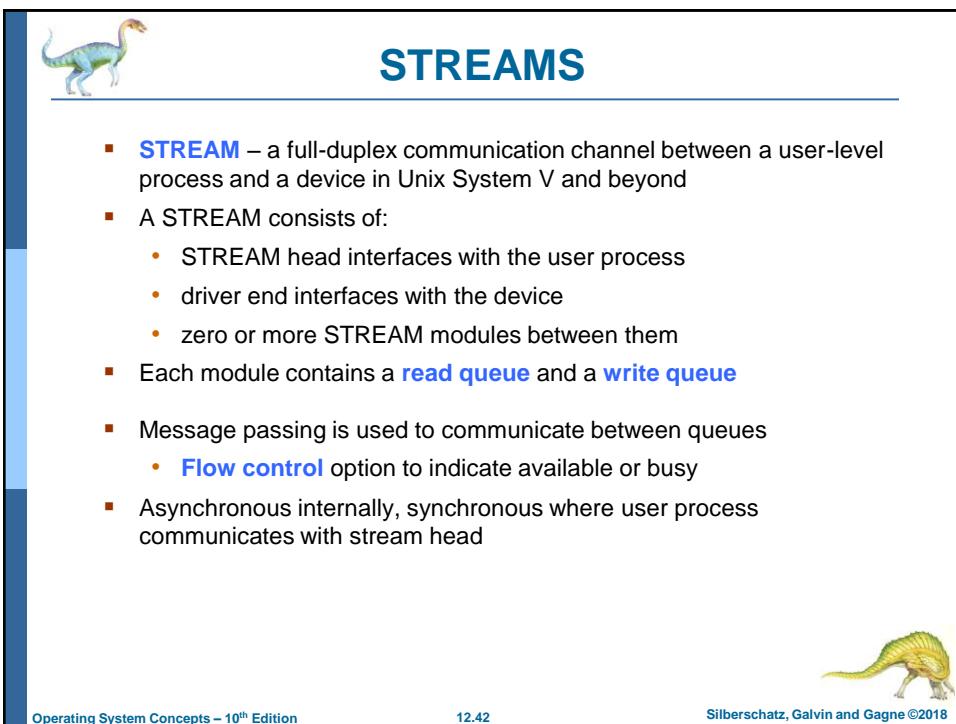
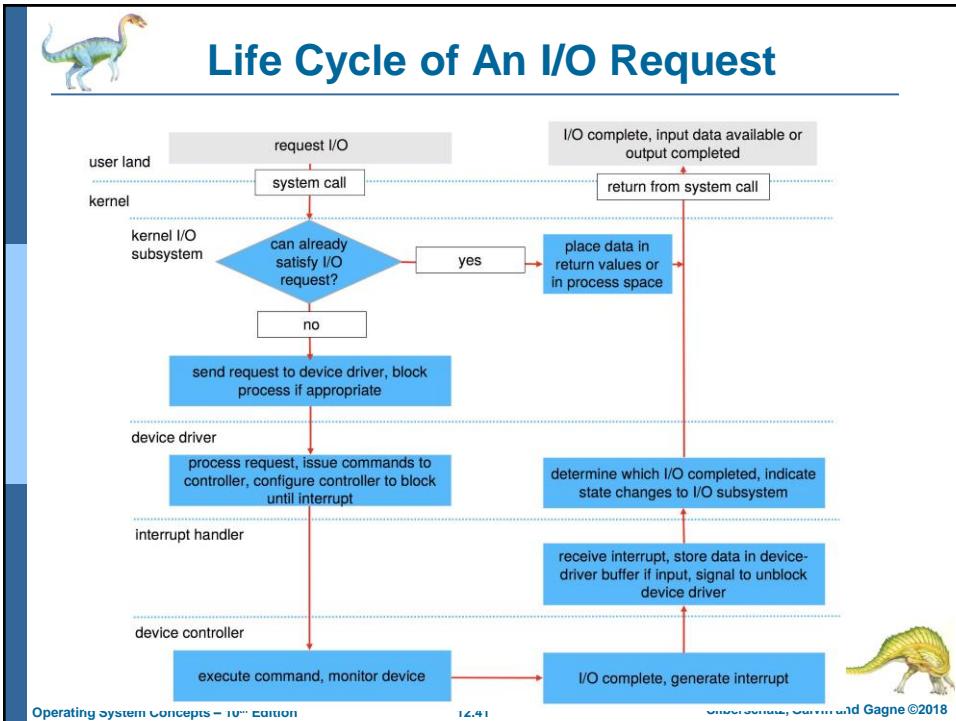
- In summary, the I/O subsystem coordinates an extensive collection of services that are available to applications and to other parts of the kernel
  - Management of the name space for files and devices
  - Access control to files and devices
  - Operation control (for example, a modem cannot seek())
  - File-system space allocation
  - Device allocation
  - Buffering, caching, and spooling
  - I/O scheduling
  - Device-status monitoring, error handling, and failure recovery
  - Device-driver configuration and initialization
  - Power management of I/O devices
- The upper levels of the I/O subsystem access devices via the uniform interface provided by the device drivers



## Transforming I/O Requests to Hardware Operations

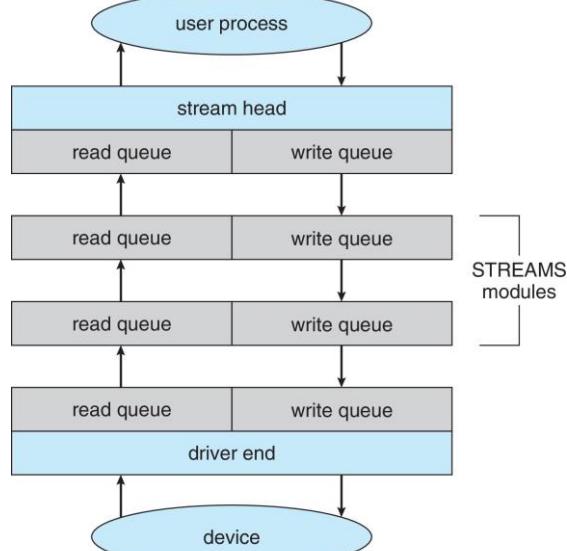
- Consider reading a file from disk for a process:
  - Determine device holding file
  - Translate name to device representation
  - Physically read data from disk into buffer
  - Make data available to requesting process
  - Return control to process







## The STREAMS Structure



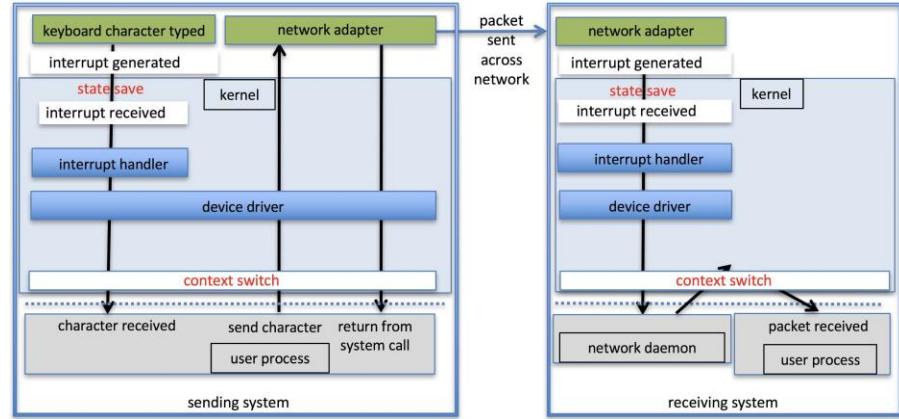
## Performance

- I/O a major factor in system performance:
  - Demands CPU to execute device driver, kernel I/O code
  - Context switches due to interrupts
  - Data copying
  - Network traffic especially stressful





## Intercomputer Communications



Operating System Concepts – 10<sup>th</sup> Edition

12.45

Silberschatz, Galvin and Gagne ©2018



## Improving Performance

- Reduce number of context switches
- Reduce data copying
- Reduce interrupts by using large transfers, smart controllers, polling
- Use DMA
- Use smarter hardware devices
- Balance CPU, memory, bus, and I/O performance for highest throughput
- Move user-mode processes / daemons to kernel threads

Operating System Concepts – 10<sup>th</sup> Edition

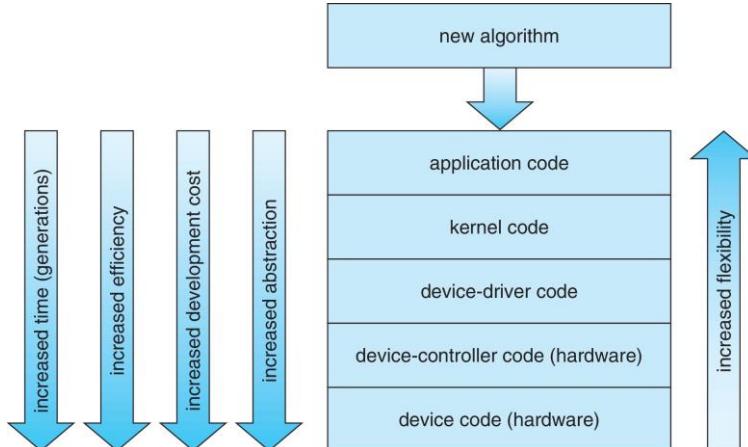
12.46

Silberschatz, Galvin and Gagne ©2018





## Device-Functionality Progression



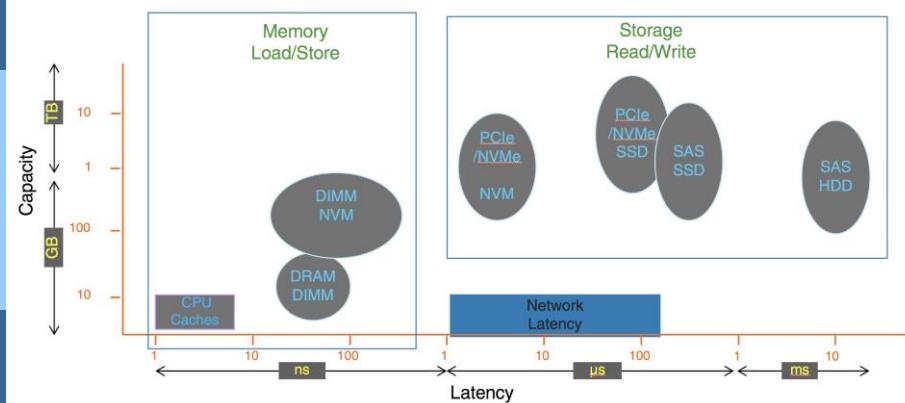
Operating System Concepts – 10<sup>th</sup> Edition

12.47

Silberschatz, Galvin and Gagne ©2018



## I/O Performance of Storage (and Network Latency)



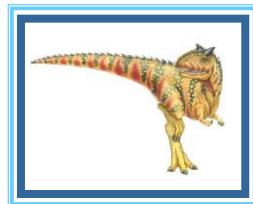
Operating System Concepts – 10<sup>th</sup> Edition

12.48

Silberschatz, Galvin and Gagne ©2018



## End of Chapter 4.5



Operating System Concepts – 10<sup>th</sup> Edition

Silberschatz, Galvin and Gagne ©2018