



BITS Pilani

Hyderabad Campus

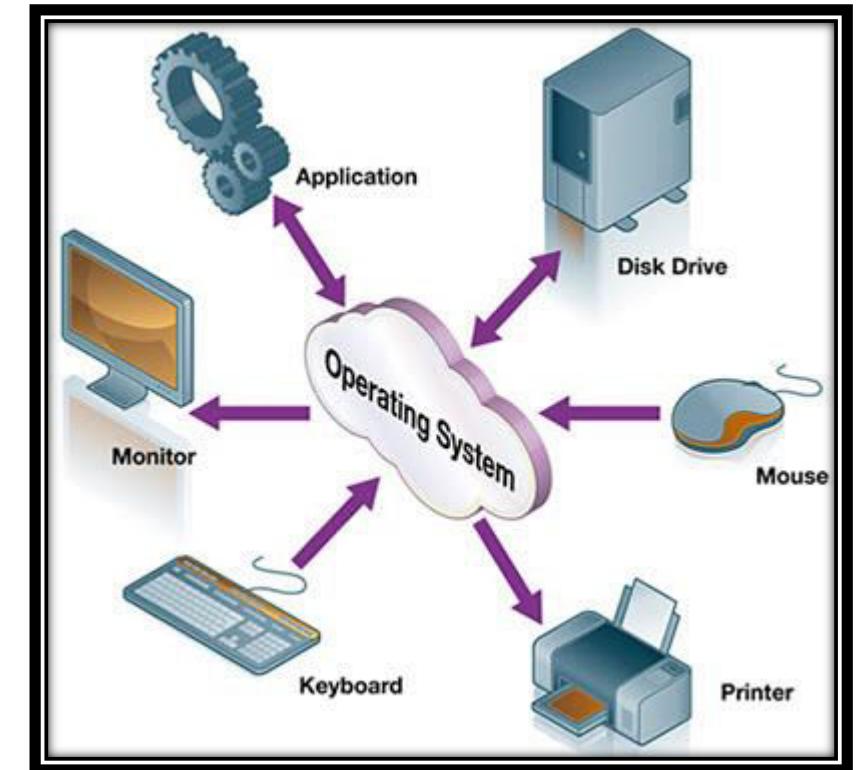
OPERATING SYSTEMS (CS F372)

Introduction

Dr. Barsha Mitra
CSIS Dept., BITS Pilani, Hyderabad Campus



What is an Operating System



Handout Overview

Objectives

- To learn about how process management is carried by the OS. This will include process creation, thread creation, CPU scheduling, process synchronization and deadlocks.
- To learn about memory management carried out by OS. This will include the concepts of paging, segmentation, swapping, and virtual memory.
- To learn how permanent storage like files and disks are managed by OS. This will include topics related to access methods, mounting, disk scheduling, and disk management.
- Hands-on experience

Handout Overview

Text Book:

T1. Silberschatz, Galvin, and Gagne, “Operating System Concepts”, 9th edition, John Wiley & Sons, 2012.

Reference Books:

R1. W. Stallings, “Operating Systems: Internals and Design Principles”, 6th edition, Pearson, 2009.

R2. Tanenbaum, Woodhull, “Operating Systems Design & Implementation”, 3rd edition, Pearson, 2006.

R3. Dhamdhere, “Operating Systems: A Concept based Approach”, 2nd edition, McGrawHill, 2009.

R4. Robert Love, “Linux Kernel Development”, 3rd edition, Pearson, 2010.

Topics to be covered

- Introduction
- OS Structures
- Processes
- Threads
- CPU Scheduling
- Process Synchronization
- Deadlocks
- Main Memory Management
- Virtual Memory
- Mass Storage
- File System Interface
- File System Implementation
- I/O Systems
- Protection

Evaluation

Component	Duration	Weightage (%)	Date & Time	Nature of Component
Mid Semester Examination	90 minutes	30%	As per Time Table	Open Book
Quiz 1	-	10%	TBA	Open Book
Quiz 2	-	10%	TBA	Open Book
Assignment		15%	TBA	Open Book
Comprehensive Examination	120 minutes	35%	As per Time Table	Open Book



Handout Overview

- Chamber Consultation
- Notices
- Make-up Policy

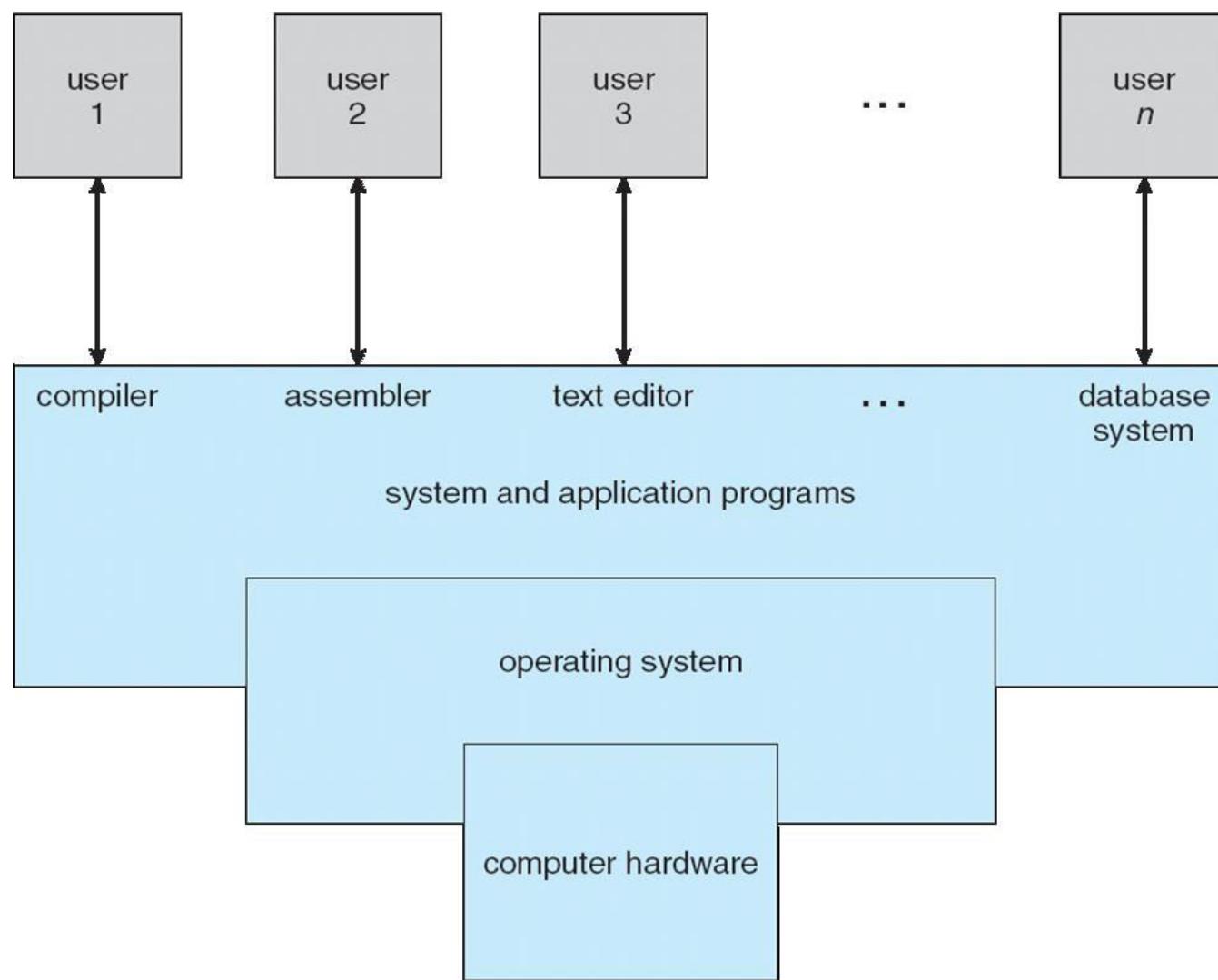
Introduction

- program that manages computer's hardware
- acts as an intermediary between computer user and computer h/w
- mainframe operating systems
- personal computer (PC) operating systems
- operating systems for mobiles

Computer System Architecture

- ❖ **Hardware** – provides basic computing resources
 - ❖ CPU, memory, storage, 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, email, web browsers, database systems, video games, media player
- ❖ **Users**
 - ❖ People, machines, other computers

Computer System Architecture



What OS Does? : User View

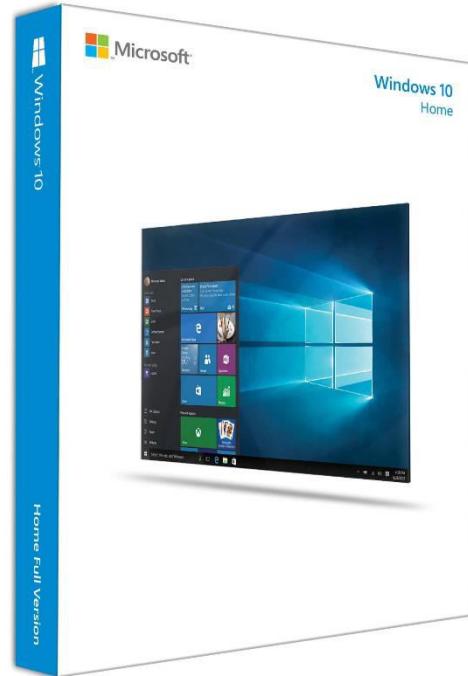
- ❖ Users want **convenience, ease of use** and **good performance**
 - ❖ Don't care about **resource utilization**
- ❖ Shared computer such as **mainframe** or **minicomputer** must keep all users happy
- ❖ Users of dedicated systems such as **workstations** have dedicated resources but frequently use shared resources from **servers**
- ❖ **Handheld computers** are resource poor, optimized for individual usability and battery life
- ❖ Some computers have little or no user interface, such as **embedded computers** in devices and automobiles

What OS Does? : System View

- ❖ OS is a **resource allocator**
 - ❖ Manages all resources
 - ❖ Decides between conflicting requests for efficient and fair resource use
- ❖ OS is a **control program**
 - ❖ Controls execution of user programs to prevent errors and improper use of the computer (like a program should not delete a section of hard-drive, a program should not interfere with other program)

How do we define OS?

- ❖ Everything a vendor ships when you order an operating system is a good approximation

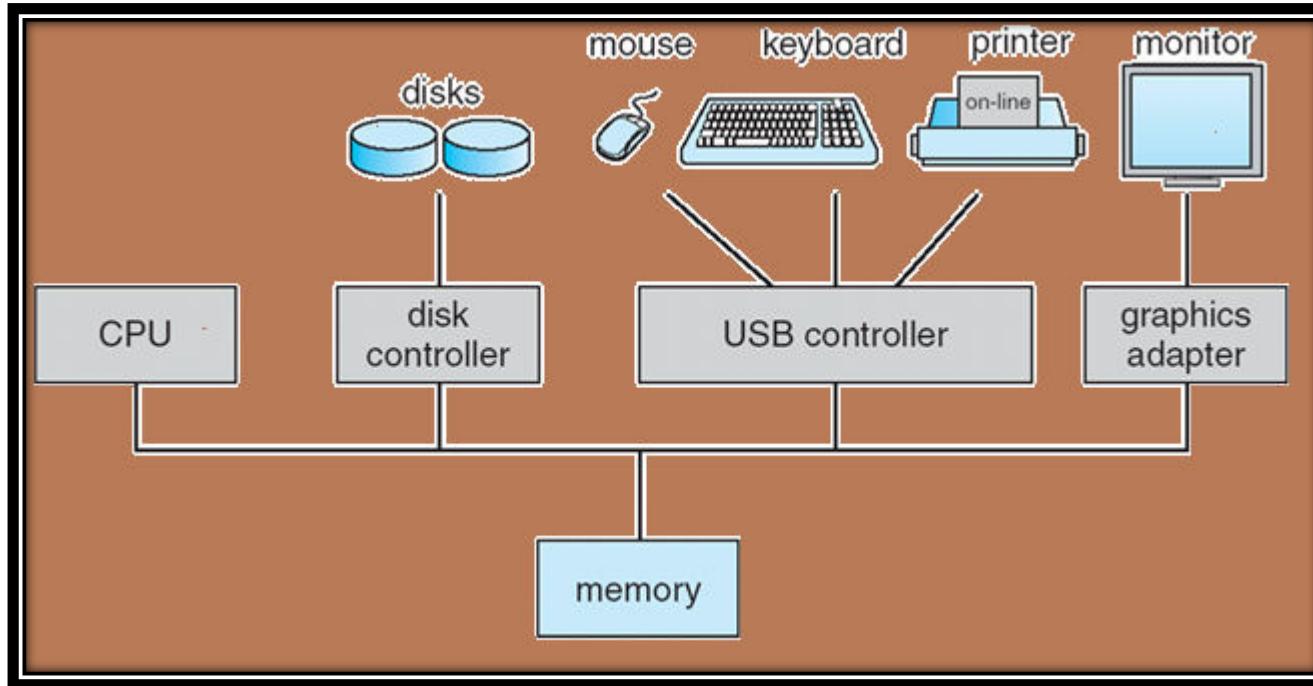


- ❖ “The one program running at all times on the computer”

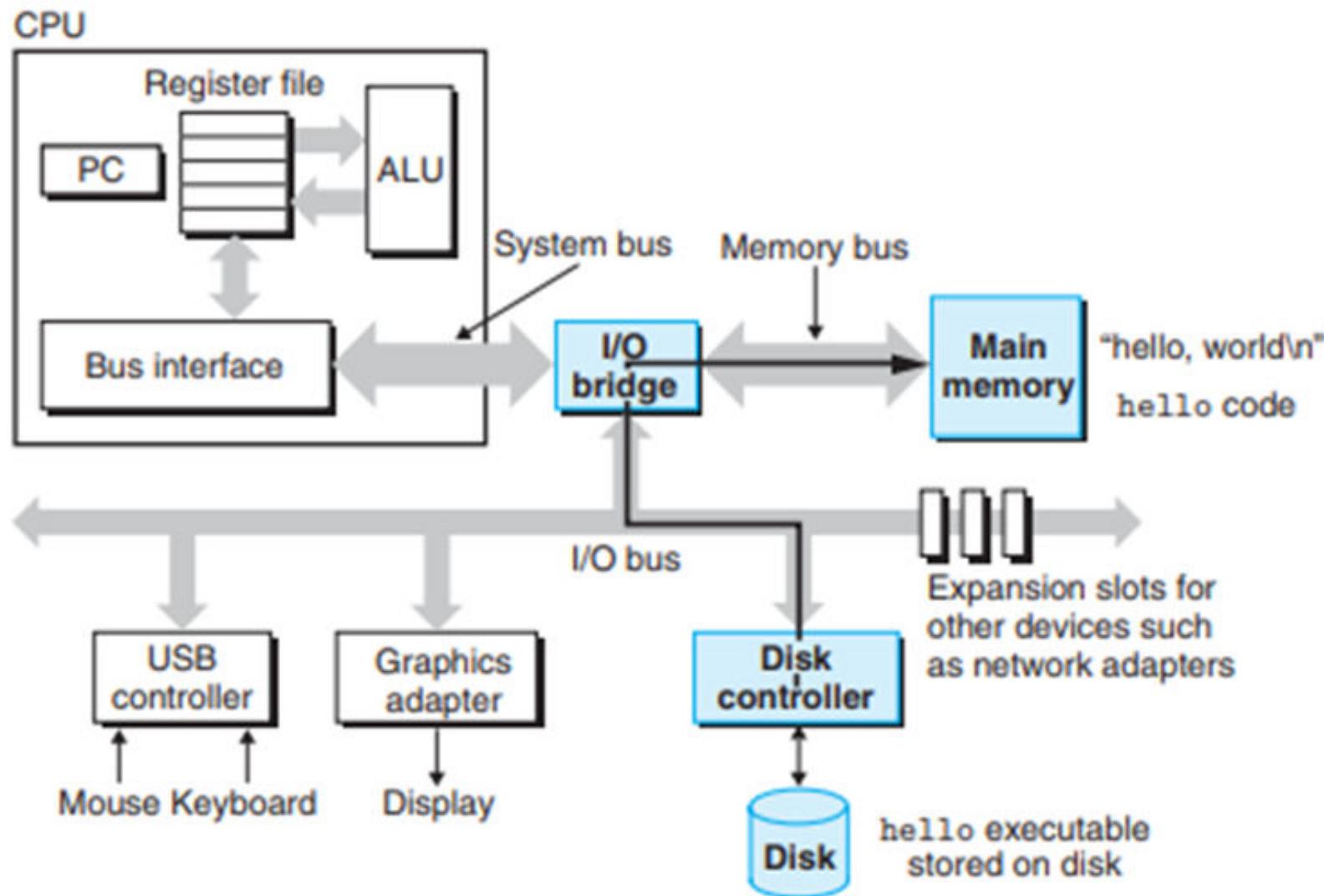
Computer System Organization

- ❖ 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

device controller is a hardware component that works as a bridge between the hardware device and the operating system(device driver of os) or an application program



Computer-System Organization



Computer-System Operation

- ❖ **Bootstrap program** is loaded at power-up or reboot
 - ❖ initial program
 - ❖ stored in ROM or EPROM, generally known as **firmware**
 - ❖ initializes all aspects of system like CPU registers, device controllers, memory contents
 - ❖ locates and loads operating system kernel and starts execution
 - (kernel is one of the **module**/part of OS which is responsible for memory management, it is core component of os basically; kernel is the very first component of the os that is loaded when computer starts)

To summarize, bootstrap is stored in eprom called firmware, bootstrap is the very first thing that runs when computer is started booted ,it initializes all aspects of system like cpu register,device controller, also bootstrap loads os kernel as it knows kernel's location in secondary memory.

Computer-System Operation

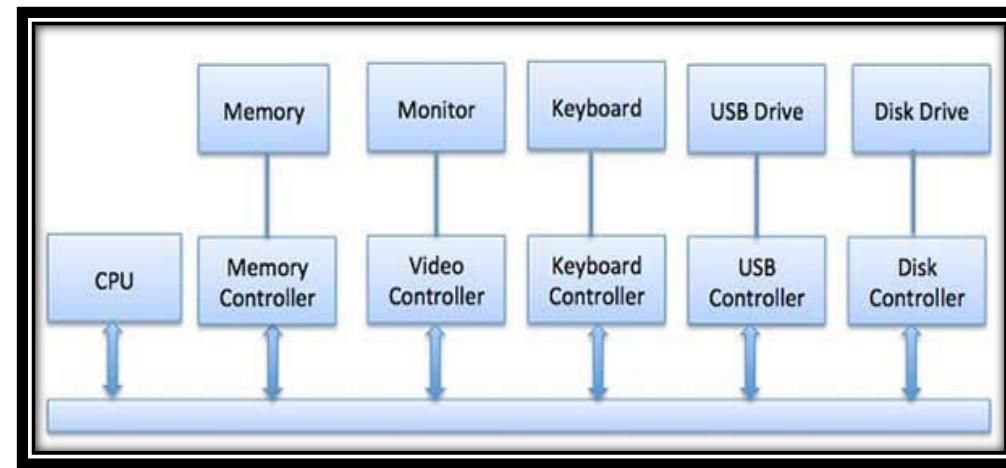
- ❖ I/O devices and the CPU can execute concurrently
- ❖ Each device controller is in charge of a particular device type
- ❖ Each device controller has a local buffer
- ❖ CPU moves data from/to main memory to/from local buffers
- ❖ I/O is from the device to local buffer of controller (ie device talk to local buffer and local buffer talk to cpu)
- ❖ Device controller informs CPU that it has finished its operation by causing an interrupt
- ❖ Basically,local buffer has fast access time than real device, ie, loading and extracting data from local buffer is faster than the device, and since cpu is really fast, local buffer were introduced to maintain efficiency of cpu
- ❖ Local buffer is the part of device controller

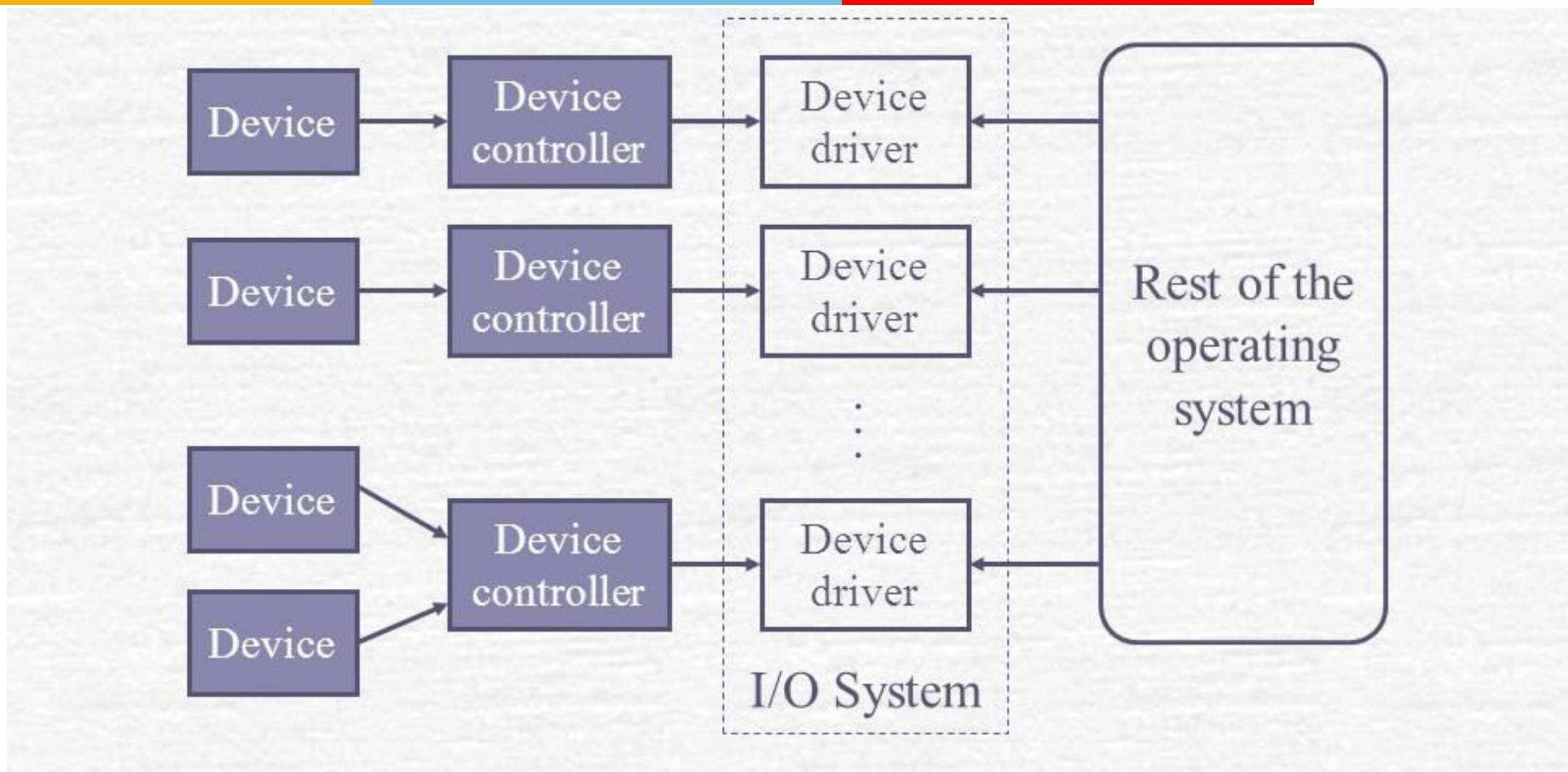
Interrupt Handling

- ❖ interrupt transfers control to the interrupt service routine (stored in a fixed location) through the interrupt vector (address for finding ISR)
 - ❖ IVT – table of pointers containing the addresses of all the interrupt service routines
 - ❖ Basically ivt stores address of all isr, isr have info of what to do when faced with specific interrupt
- ❖ must save the address of the interrupted instruction, system stack
- ❖ trap/exception is a software-generated interrupt caused either by an error or a user request (example arithmetic/filenotfound/segmentation/indexnotfound exception are error type interrupts, user type interrupt are like calling user-defined or any in built function from main function leads to main function getting on hold till called function returns control)
- ❖ operating system is interrupt driven
- ❖ operating system preserves the state of the CPU by storing contents of registers and the program counter , this previous state is stored in system stack

Computer-System Operation

- ❖ computer system consists of CPUs and multiple device controllers that are connected through a common bus
- ❖ each device controller is in charge of a specific type of device
- ❖ device controller
 - ❖ maintains some local buffer storage and a set of special-purpose registers
 - ❖ moves data between the peripheral devices that it controls and its local buffer storage
- ❖ operating systems have a **device driver** for each device controller
- ❖ device driver understands the device controller and provides the rest of the operating system with a uniform interface to the device (generally necessary device drivers come already pre-built inside OS but some others have to be installed by user as per demand)





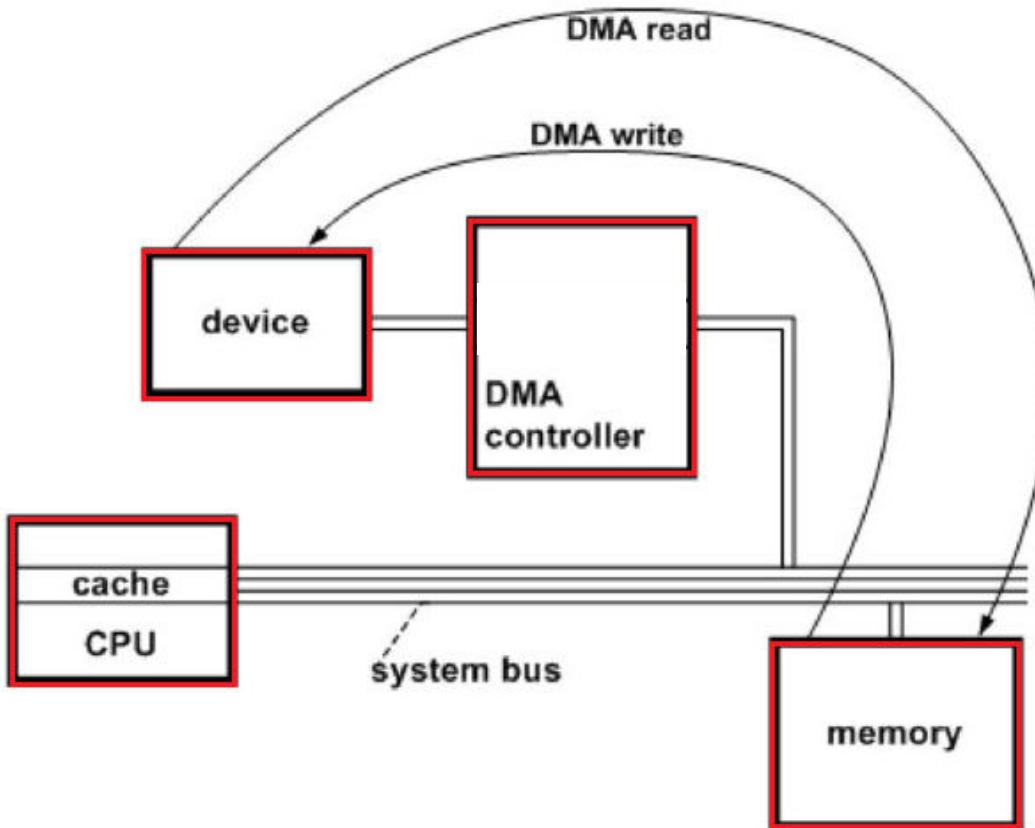
I/O Structure

- ❖ to start an I/O operation, the **device driver loads** the registers within the device controller
 - ❖ **device controller** examines the contents of registers to determine what action to take
 - ❖ controller starts the transfer of data from the device to its **local buffer**
 - ❖ device controller informs the **device driver** via an interrupt that it has finished its operation
 - ❖ device driver then returns control to the operating system, possibly returning the data or a pointer to the data if the operation was a read
 - ❖ for other operations, the device driver returns status information
-

I/O Structure

- ❖ interrupt-driven I/O is fine for moving small amounts of data(usually one interrupt is generated after each byte transfer which makes it slow)
- ❖ can produce high overhead when used for bulk data movement such as disk I/O
- ❖ **direct memory access (DMA)**
- ❖ device controller sets up buffers, pointers, and counters for the I/O device
- ❖ In DMA , device controller **transfers an entire block of data(chunk of many bytes)** directly to or from its own buffer storage to memory, **with no intervention by the CPU**
- ❖ Now **only one interrupt is generated per block**, to tell the device driver that the operation has completed instead of one interrupt per byte generated for low-speed devices
- ❖ Hence now CPU is available to accomplish other work
- ❖ To summarize, in DMA, block of data is transferred and no CPU is involved but direct access to memory which allows cpu to do other work, only one interrupt per block

I/O Structure



Storage Structure

- ❖ Main memory –
 - ❖ only storage media that the CPU can access directly
 - ❖ instruction execution
 - ❖ random access
 - ❖ volatile



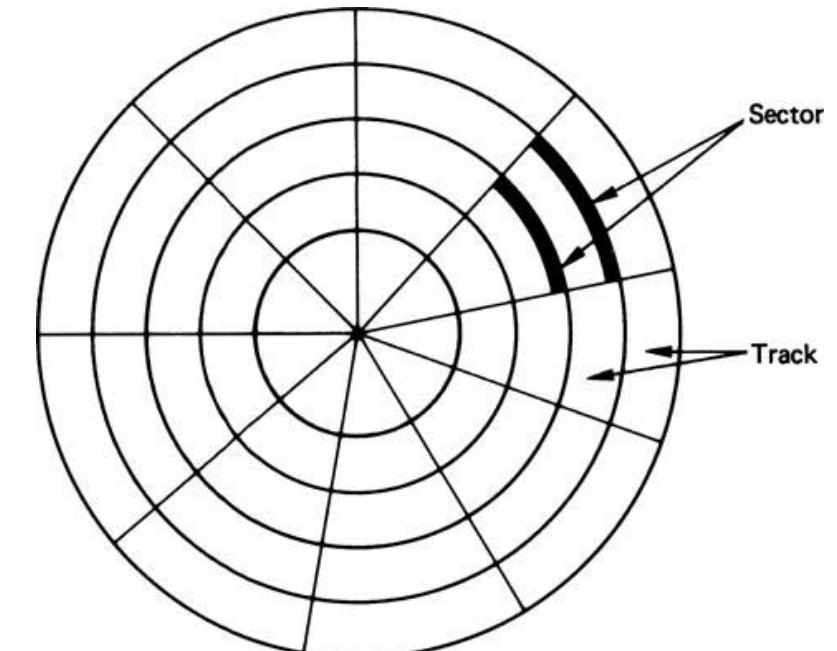
Storage Structure

- ❖ Secondary storage –
 - ❖ extension of main memory that provides large nonvolatile storage capacity
 - ❖ stores both program and data



Storage Structure

- ❖ Hard disks/Magnetic disks –
 - ❖ rigid metal or glass platters covered with magnetic recording material
 - ❖ disk surface is logically divided into tracks, which are subdivided into sectors
 - ❖ disk controller determines the interaction between the device and the computer



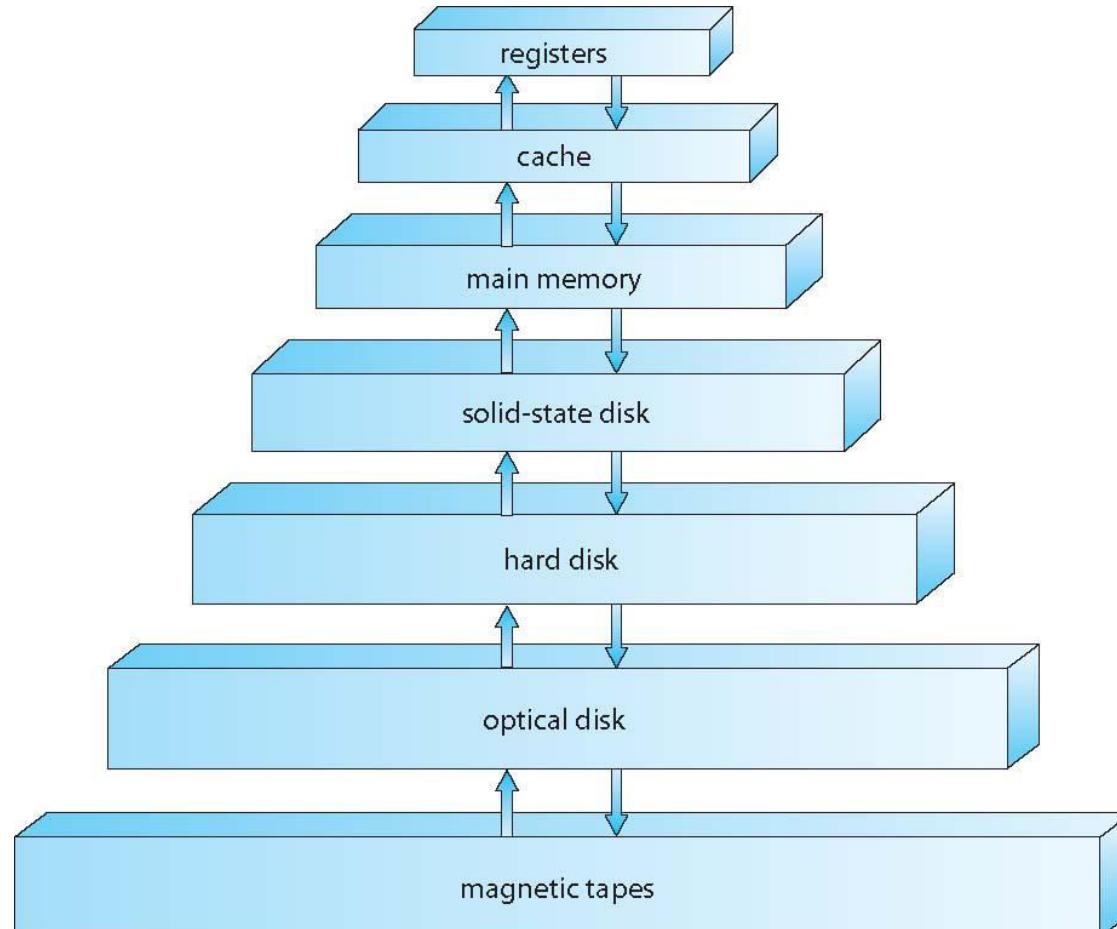
Storage Structure

- ❖ Solid-state disks –
 - ❖ faster than magnetic disks, nonvolatile
 - ❖ becoming more popular

- ❖ stores data in DRAM during normal operation
- ❖ also contains a hidden magnetic hard disk and a battery for backup power
- ❖ if external power is interrupted, solid-state disk's controller copies the data from RAM to the magnetic disk
- ❖ when external power is restored, the controller copies the data back into RAM
- ❖ another form of solid-state disk is flash memory, which is popular in cameras and personal digital assistants (PDAs), slower than DRAM but needs no power to retain its contents



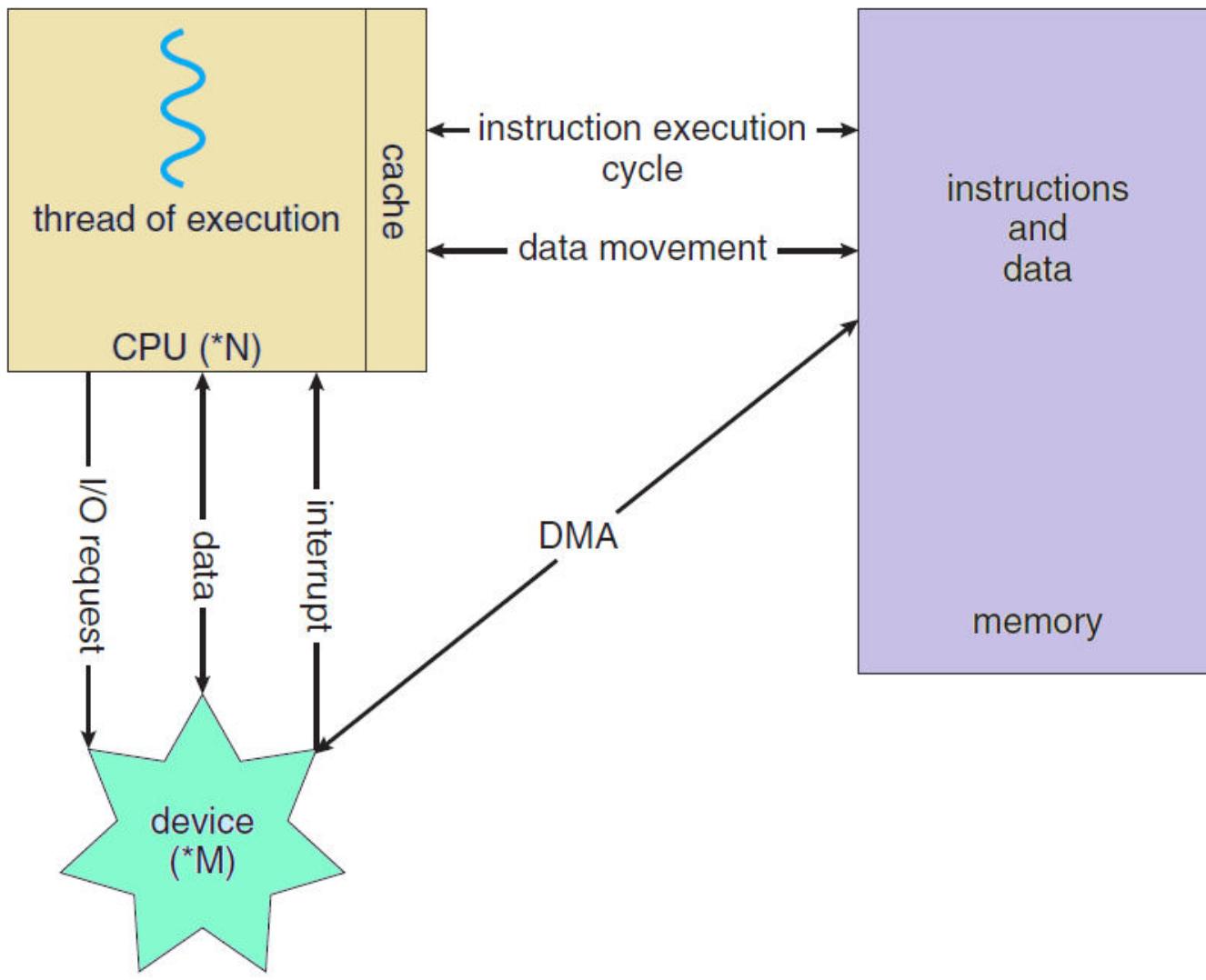
Storage Structure



DOWN THE GRAPH:

- 1) Access speed decreases
- 2) Physical size to store same amount of data increases
- 3) Cost decreases

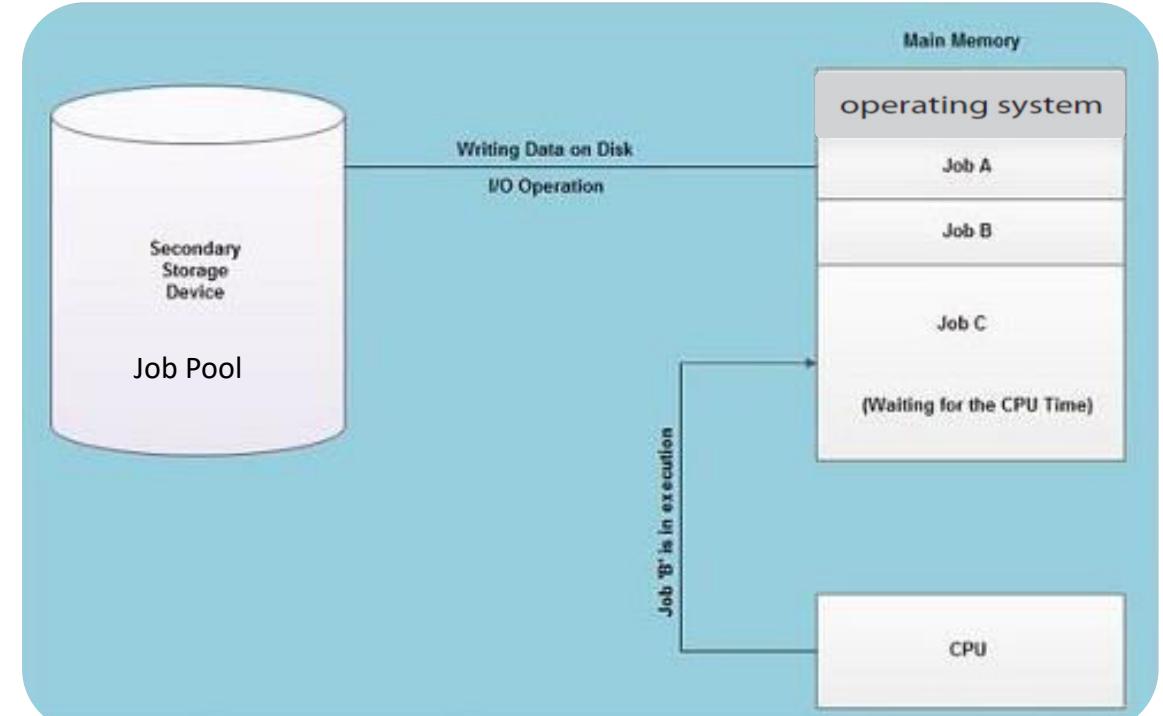
Putting it All Together



Operating System Structure

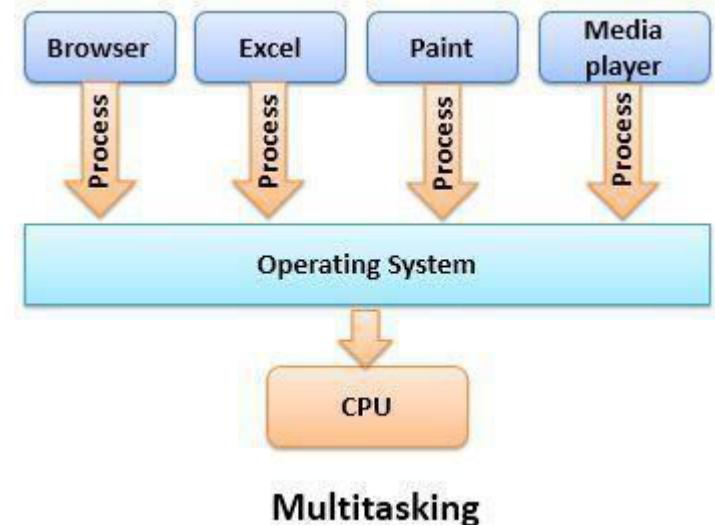
❖ Multiprogramming (Batch system)

- ❖ Needed for efficiency
- ❖ Single process cannot keep CPU and I/O devices busy at all times
- ❖ 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 is selected and run via **job scheduling**
- ❖ When it has to wait for I/O, OS switches to another job



Operating System Structure

- ❖ **Timesharing (multitasking)**: CPU switches jobs so frequently that users can interact with each job while it is running
- ❖ **interactive** computing
 - ❖ User interaction via input devices
 - ❖ **Response time** should be minimal
 - ❖ Each user has at least one program executing in memory \Rightarrow **process (program that is being executed is a process)**
 - ❖ If several processes ready to run at the same time \Rightarrow **CPU scheduling (which process to be given to which CPU)**
 - ❖ If processes don't fit in memory(main memory/RAM), **swapping** moves them in and out to run
 - ❖ **Virtual memory** allows execution of processes larger than physical memory

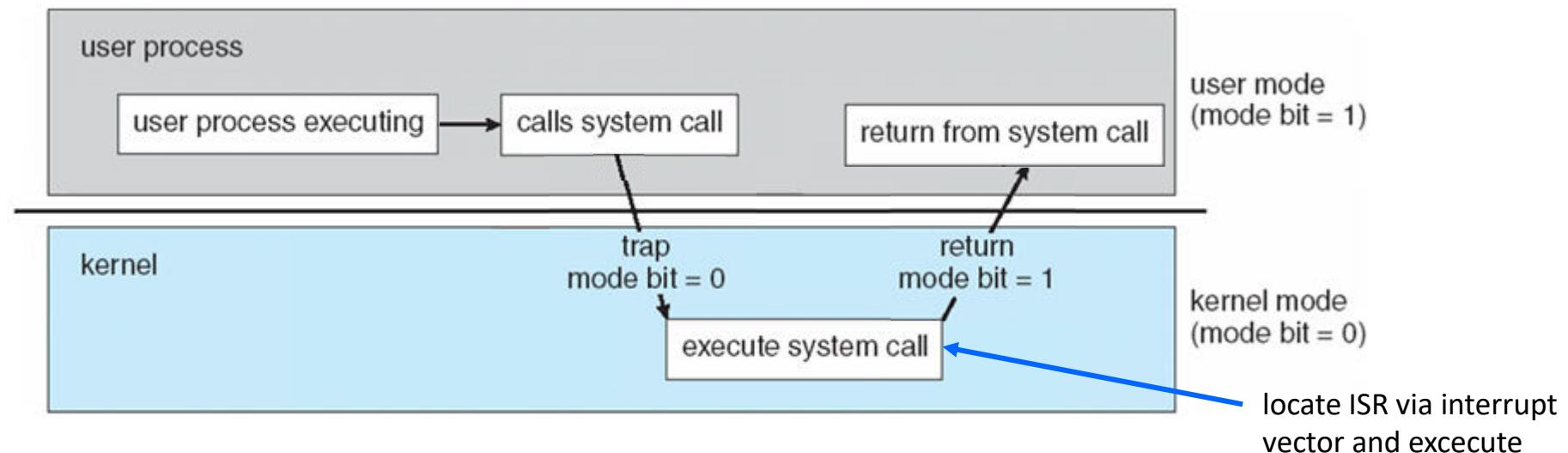


Operating System Operations

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

Operating System Operations

- ❖ **Dual-mode** operation allows OS to protect itself and protect users from one another
- ❖ **User mode** and **Kernel/Supervisor/System/Privileged mode**
- ❖ **Mode bit** provided by hardware
 - ❖ Provides ability to distinguish when system is running user code or kernel code
 - ❖ Some instructions designated as **privileged**, only executable in kernel mode
 - ❖ **System call** changes mode to kernel, return from call resets it to user



Operating System Operations

- ❖ Boot time → hardware starts in kernel mode
 - ❖ After loading OS, user applications are started in user mode
 - ❖ When trap/interrupt occurs, hardware switches from user mode to kernel mode
 - ❖ examples of privileged instructions – switch to kernel mode, I/O control, timer management, interrupt management

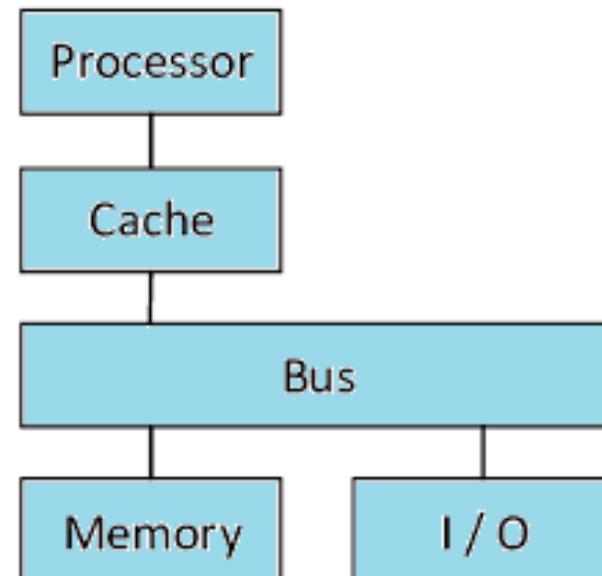
 - ❖ If switching to kernel mode itself is privileged(only executed in kernel mode), how do we to kernel mode switch in the first place?
 - ❖ Sol– in user mode, we can only invoke system call (like printf or scanf which are privileged), automatically it transitions to kernel mode when such invocation happens.
-

Operating System Operations: Timer

- ❖ User processes must return control to OS
 - ❖ Prevent infinite loop / process hogging resources
 - ❖ Set to interrupt the computer after some time period
 - ❖ Keep a counter that is decremented for every physical clock tick
 - ❖ Operating system sets the counter (privileged instruction) before switching to user mode
 - ❖ When counter reaches zero, generate an interrupt
 - ❖ Set up before scheduling process to regain control or terminate program that exceeds allotted time
-

Computing Environments

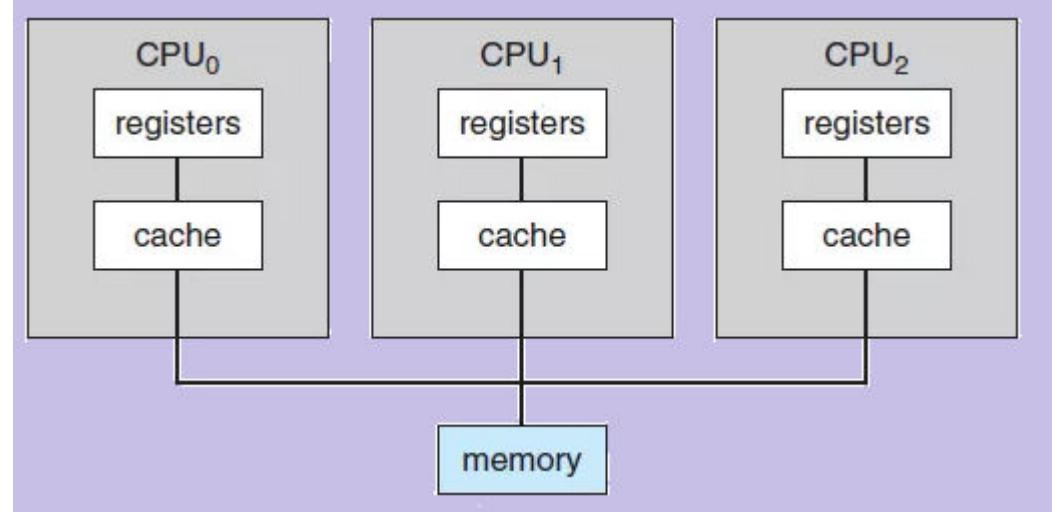
Single-Processor Systems - one main CPU executing instructions, including instructions from user processes, some device-specific processors like disk, keyboard and graphics controller and I/O processor may be present



Computing Environments

❖ Multiprocessors

- ❖ Also known as **parallel systems, multi-core systems**
- ❖ 2 or more processors in close communication, sharing the computer bus and sometimes the clock, memory and peripheral devices
- ❖ Advantages:
 - ❖ **Increased throughput**
 - ❖ **Economy of scale(10 separate processors will cost more than 10-integrated multi-processors as multiprocessors share memory, peripheral devices which reduces cost)**
 - ❖ **Increased reliability** – graceful degradation (ability of system to continue providing service proportional to the degree of healthy components/surviving hardware(here processors) that are present), fault tolerance(even if some components(processors) fail, normal operations can continue as if nothing happened)

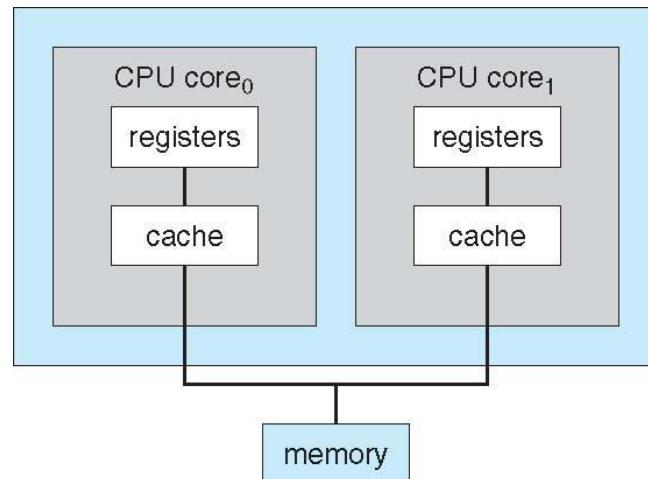


- ❖ Two types:
 - ❖ **Asymmetric Multiprocessing** – each processor is assigned a specific task, boss processor controls worker processors
 - ❖ **Symmetric Multiprocessing** – each processor performs all tasks, peers

Computing Environments

❖ Multicore Systems

- ❖ include multiple computing cores on a single chip
- ❖ more efficient than multiple chips with single cores because on-chip communication is faster than between-chip communication

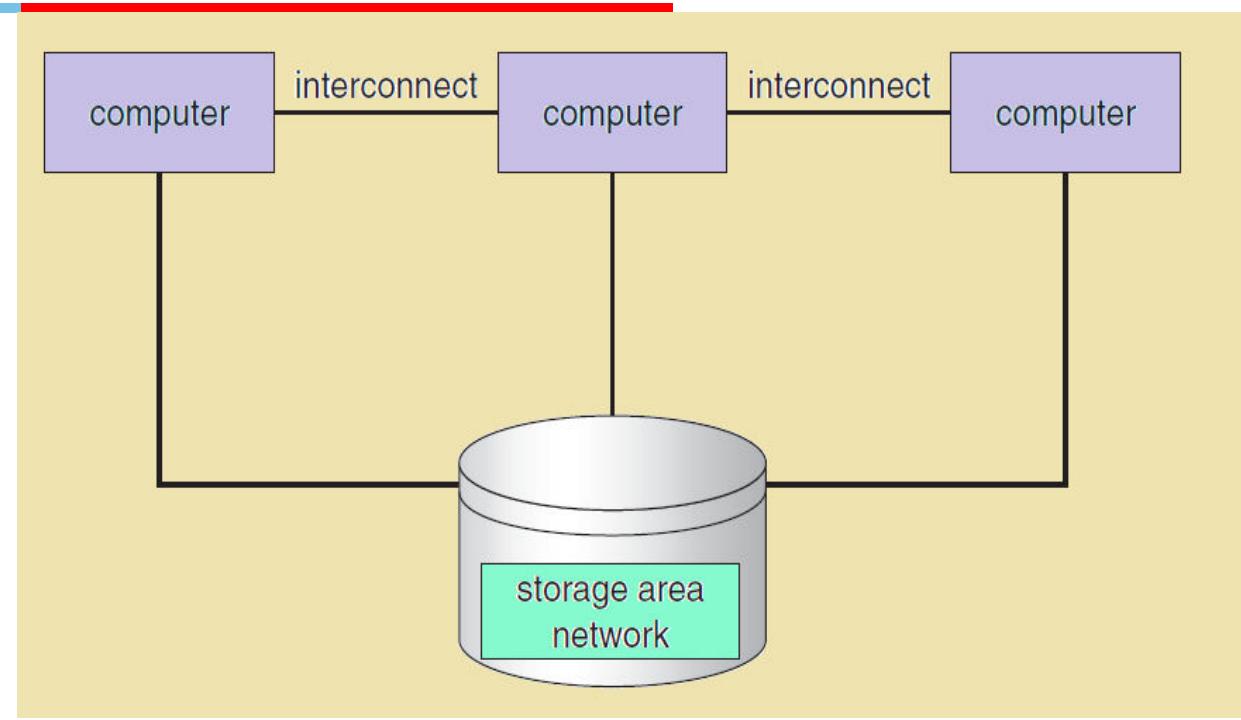


dual-core design with both cores on same chip

Computing Environments

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, users can see only a brief interruption of service
 - ❖ **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**

Computing Environments

Traditional Computing

- ❖ stand-alone general purpose machines
- ❖ most systems interconnect with others (i.e., the Internet)
- ❖ mobile computers interconnect via wireless networks
- ❖ home computers

Computing Environments

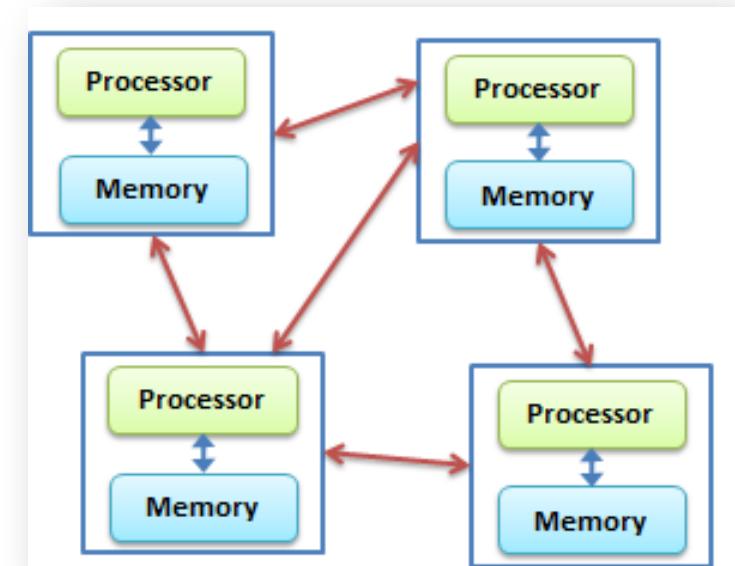
Mobile Computing

- ❖ handheld smartphones, tablets, etc.
- ❖ portable, lightweight
- ❖ allows different types of apps
- ❖ use wireless, or cellular data networks for connectivity
- ❖ Apple iOS, Google Android

Computing Environments

Distributed Computing

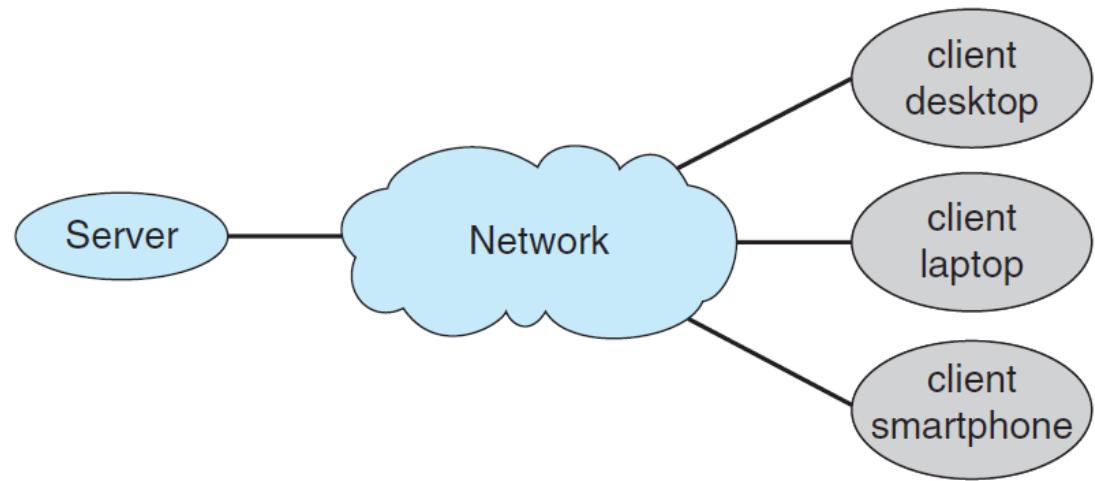
- ❖ collection of separate, possibly heterogeneous, systems networked together
- ❖ access to shared resources
- ❖ They are connected through network, which is a communications path
 - ❖ Local Area Network (LAN)
 - ❖ Wide Area Network (WAN)
 - ❖ Metropolitan Area Network (MAN)
 - ❖ Personal Area Network (PAN)
- ❖ systems exchange messages



Computing Environments

Client Server Computing

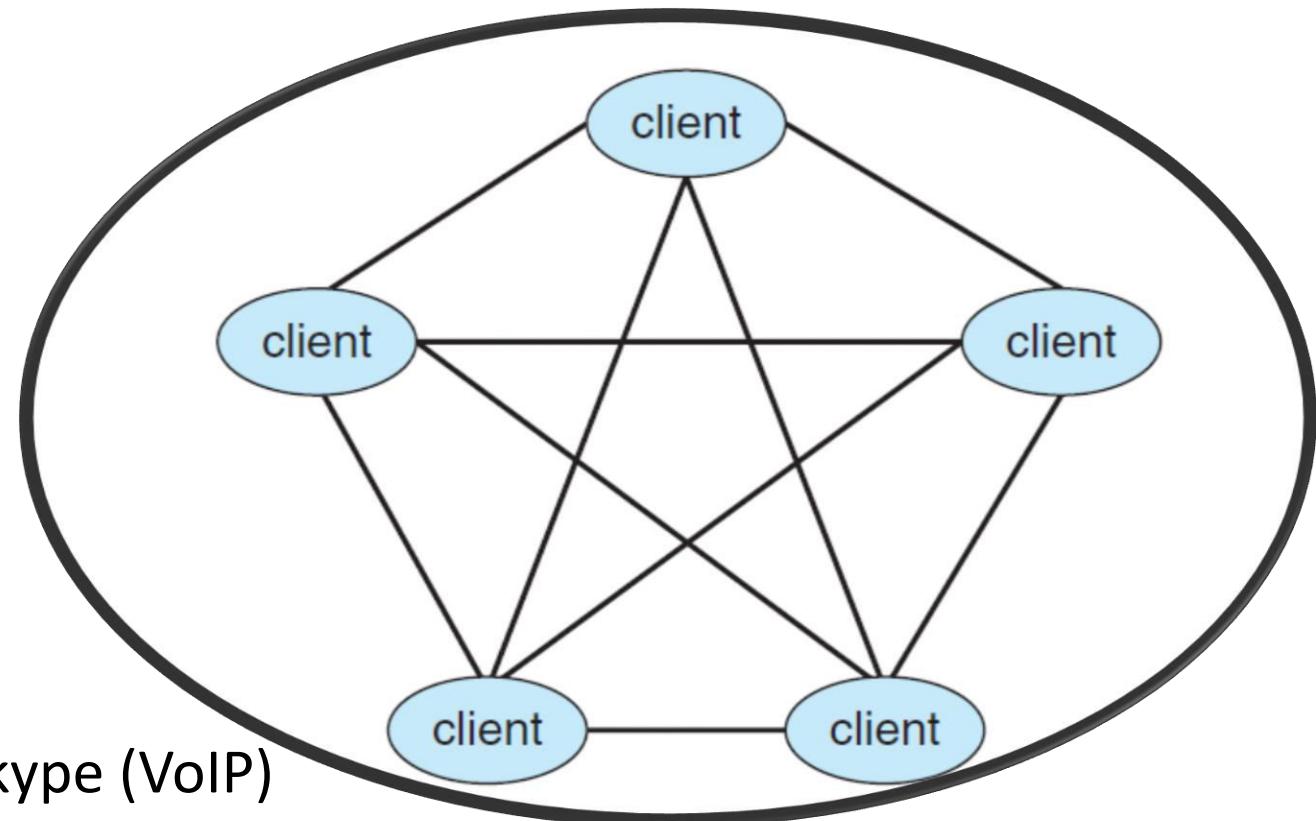
- ❖ terminals are PCs and mobile devices
- ❖ servers respond to requests generated by clients
- ❖ servers can be of 2 types
 - ❖ **compute-server** system provides an interface to client to request services, server executes the action and sends the results to clients
 - ❖ **file-server system** provides interface for clients to create, update, read and delete files



Computing Environments

Peer-to-peer Computing

- ❖ does not distinguish clients and servers
- ❖ nodes join and may also leave P2P network
- ❖ advantage over client server system
- ❖ Napster, Gnutella, BitTorrent, DC++, Skype (VoIP)





Thank You



BITS Pilani
Hyderabad Campus

OPERATING SYSTEMS (CS F372)

OS Structures

Dr. Barsha Mitra
CSIS Dept., BITS Pilani, Hyderabad Campus



Operating System Services

- ❖ **User interface** - almost all operating systems have a user interface (**UI**)
 - ❖ **Command-Line (CLI), Graphics User Interface (GUI)**
- ❖ **Program execution** - 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** - running program may require I/O, which may involve a file or an I/O device
- ❖ **File-system manipulation** - read and write files and directories, create and delete them, search them, list file information, permission management

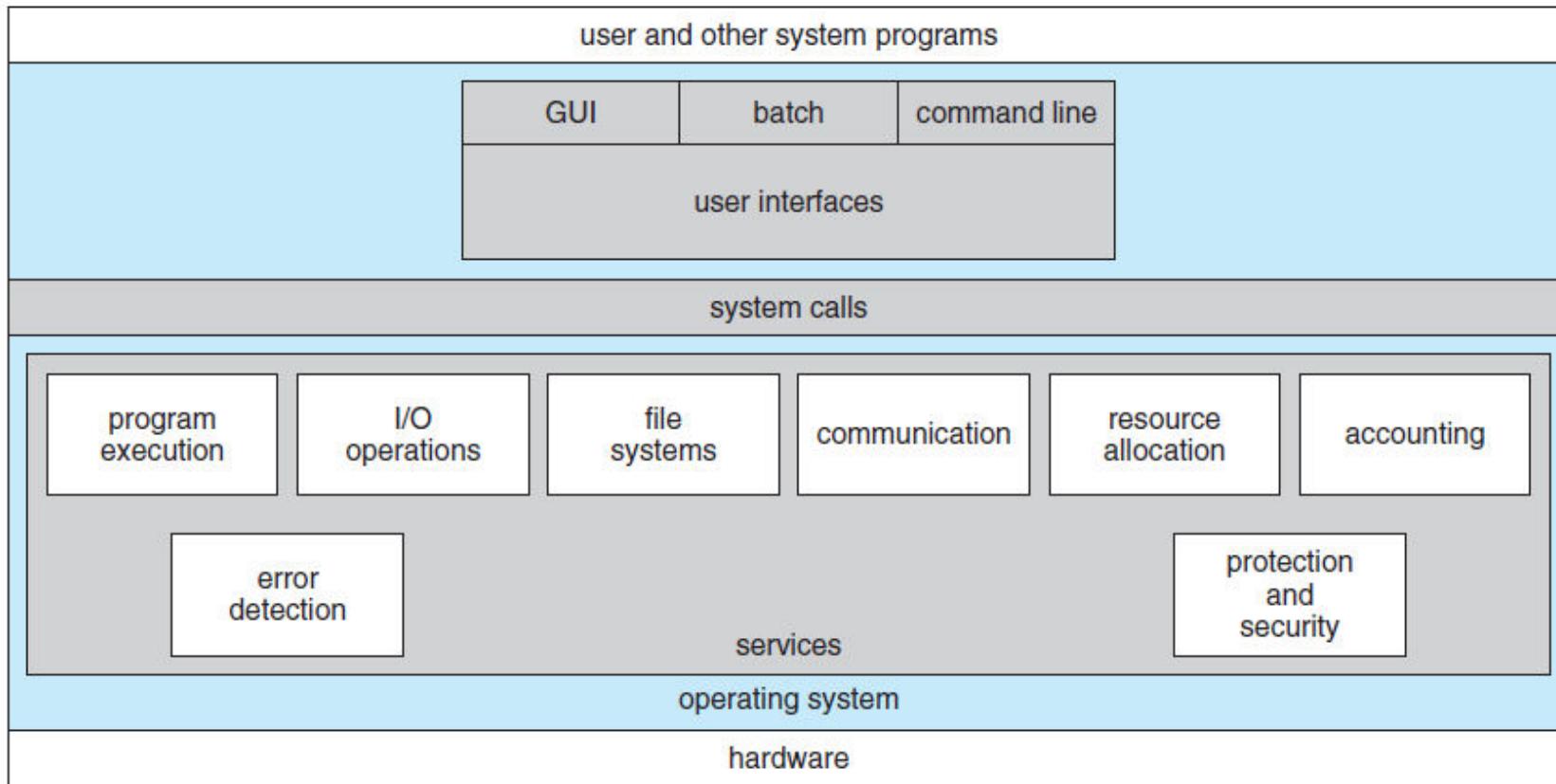
Operating System Services

- ❖ **Communications** – processes may exchange information, on the same computer or between computers over a network, **shared memory or message passing**
- ❖ **Error detection** –
 - ❖ OS needs to be constantly aware of possible errors
 - ❖ May occur in CPU and memory h/w, in I/O devices, in user program
 - ❖ OS should take the appropriate action to ensure correct and consistent computing
 - ❖ Take corrective actions

Operating System Services

- ❖ **Resource allocation** – allocating resources like CPU cycles, main memory, file storage, I/O devices for multiple concurrently executing processes
- ❖ **Accounting** – keep track of which users use how much and what kinds of computer resources
- ❖ **Protection and security** –
 - ❖ owners of information stored in a multiuser or networked computer system want to control use of that information
 - ❖ concurrent processes should not interfere with each other or with OS
 - ❖ ensuring that all accesses 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

Operating System Services



THE ENTIRE BLUE REGION IS THE OS

User and Operating-System Interface: CLI



- ❖ **CLI** or command interpreter
- ❖ Sometimes implemented in kernel, sometimes by separate program (Unix, Windows)
- ❖ Sometimes multiple flavors implemented – shells
- ❖ Primarily fetches a command from user and executes it

User and Operating-System Interface: GUI



- ❖ User-friendly 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))
- ❖ Many systems now include both CLI and GUI interfaces
 - ❖ Microsoft Windows is GUI with CLI “command” shell
 - ❖ Unix and Linux have CLI with optional GUI interfaces (CDE, KDE, GNOME)

User and Operating-System Interface: Touchscreen Interface



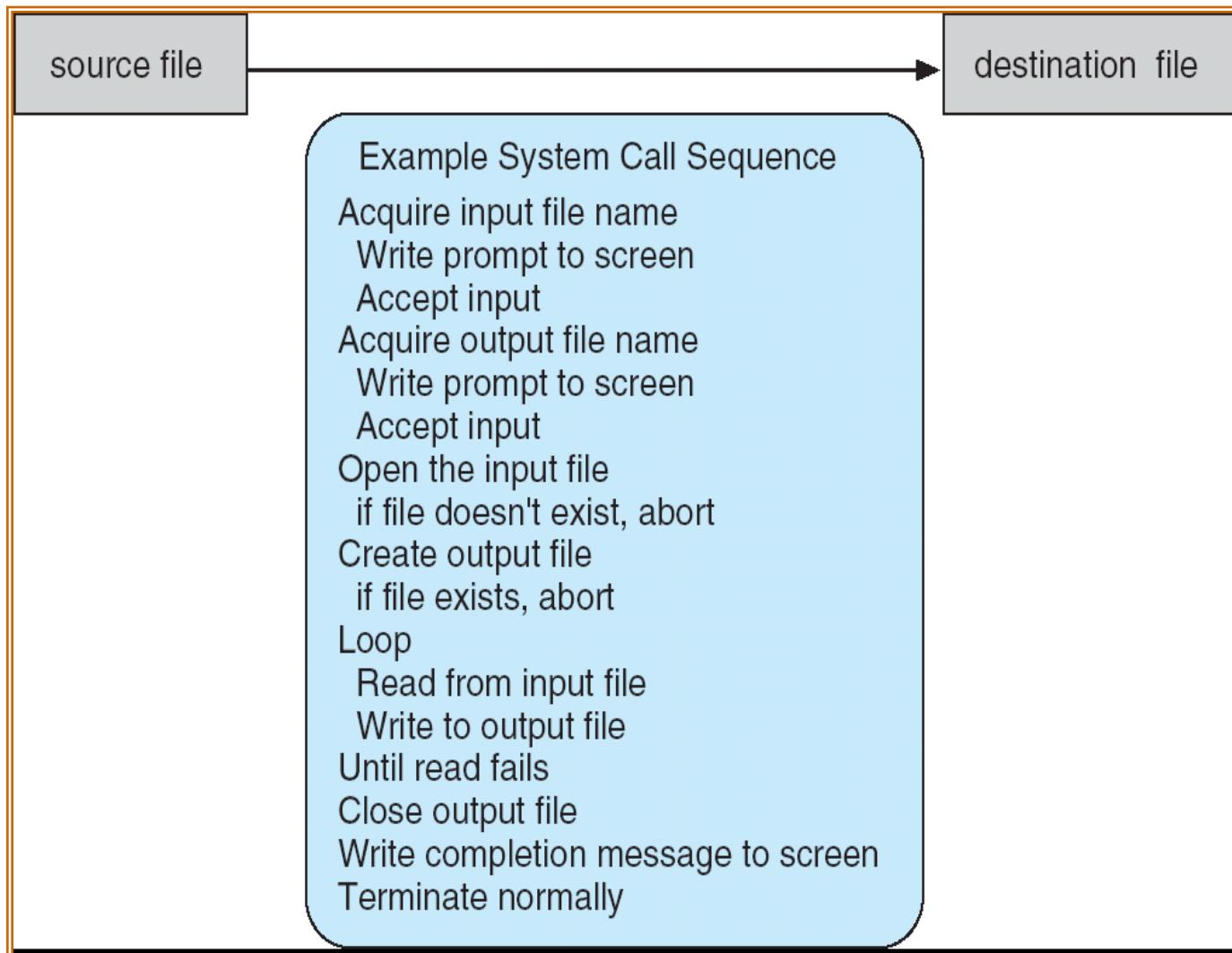
- ❖ Touchscreen devices require new interfaces
- ❖ Mouse not possible or not desired
- ❖ Actions and selection based on gestures
- ❖ Virtual keyboard for text entry
- ❖ Voice commands



Choice of Interface



System Calls



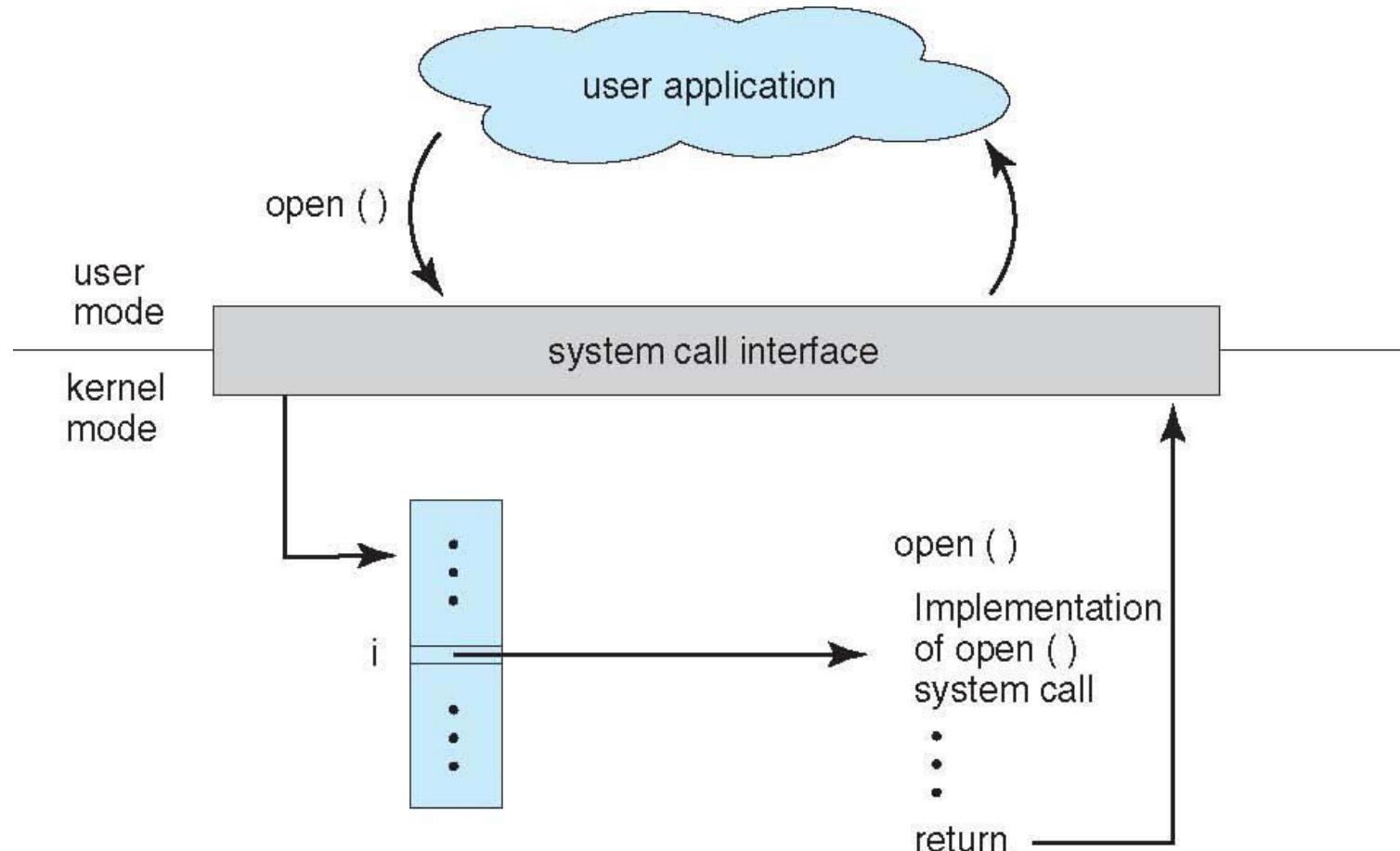
System Calls

- ❖ 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
- ❖ API specifies a set of functions available to application programmers
- ❖ Programmers access API via code library provided by the OS
- ❖ Three most common APIs are
 - ❖ Win32 API for Windows
 - ❖ POSIX API for POSIX-based systems (including all versions of UNIX, Linux, and Mac OS X) (POSIX-portable operating system interface based on unix)
 - ❖ Java API for the Java virtual machine (JVM)

System Calls

- ❖ 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 know nothing about how the system call is implemented
- ❖ Just needs to obey API and understand what OS will do as a result of call execution
- ❖ 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)

System Calls



Types of System Calls

- **Process control**

- create process, terminate process
- end, abort
- load, execute
- get process attributes, set process attributes

- **File management**

- create file, delete file
- open, close file
- read, write file
- get and set file attributes

- **Device management**

- request device, release device
- read, write
- get device attributes, set device attributes
- logically attach or detach devices

- **Information Maintenance**

- **Communication**

- **Protection**

Examples of System Calls

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

OS Structure

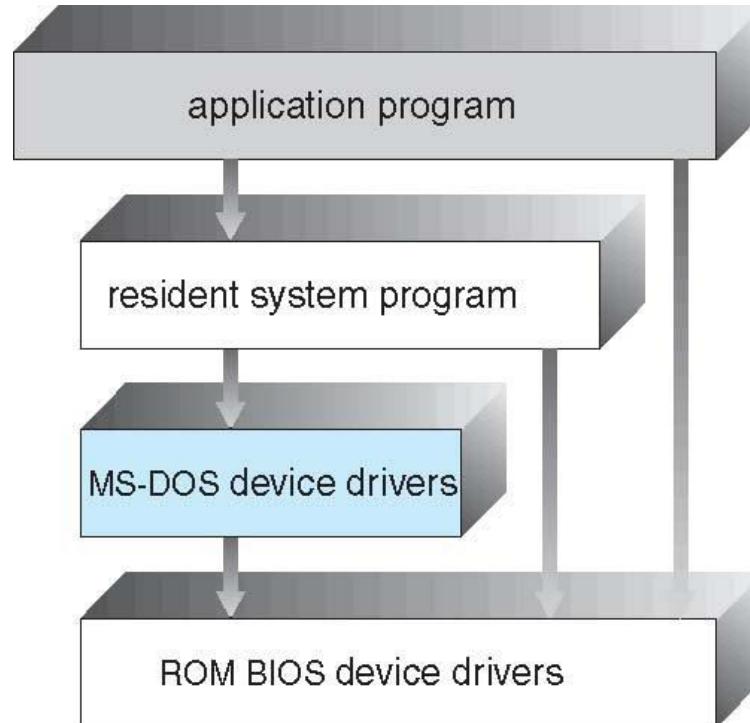


- Simple Structure/ Monolithic Kernel
- Layered Approach
- Microkernels
- Modules
- Hybrid System

Simple Structure

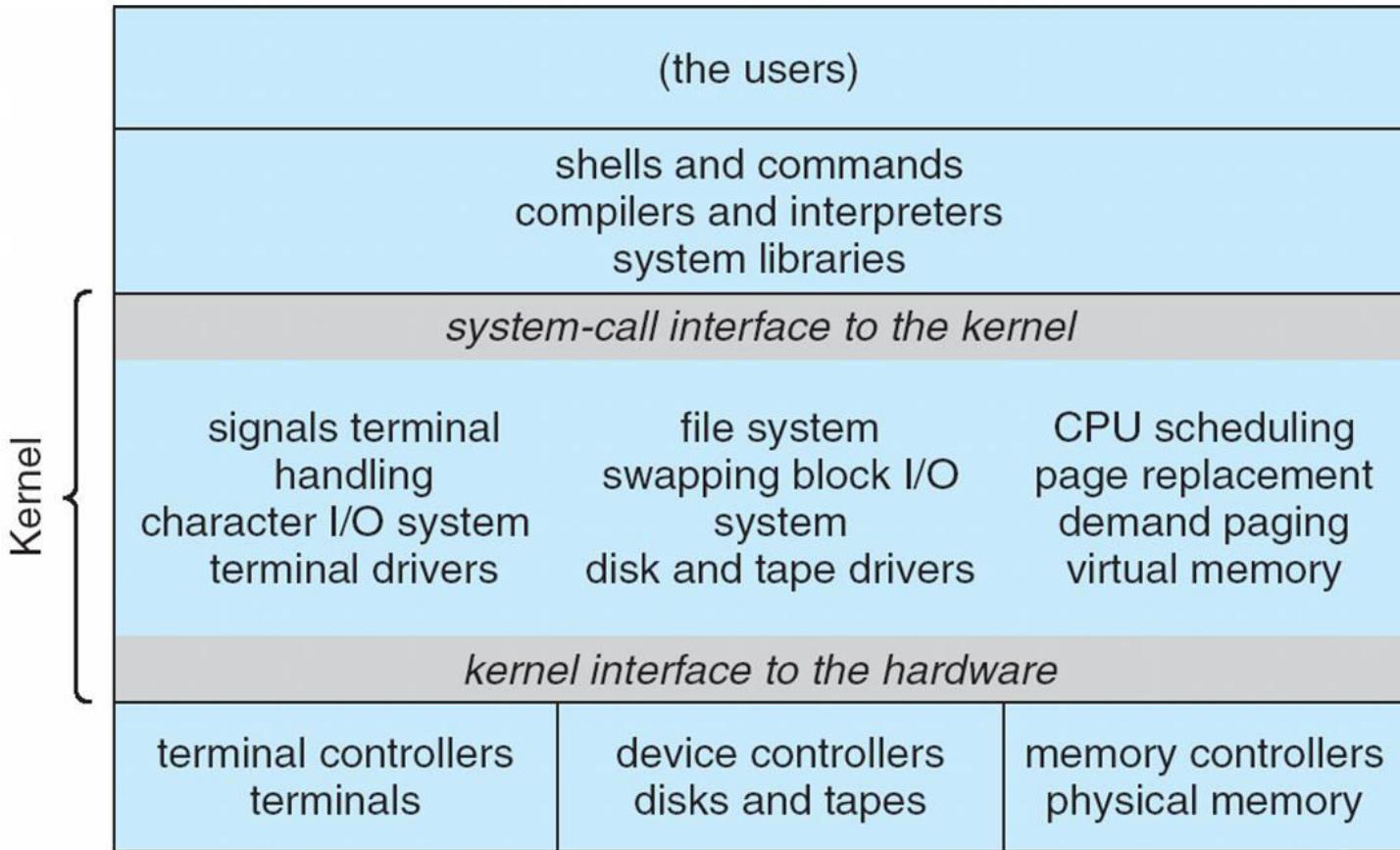


- not divided into modules
- interfaces and levels of functionality are not well separated
- application programs are able to access the basic I/O routines to write directly to the display and disk drives
- vulnerable to malicious programs, causing entire system crashes when user programs fail



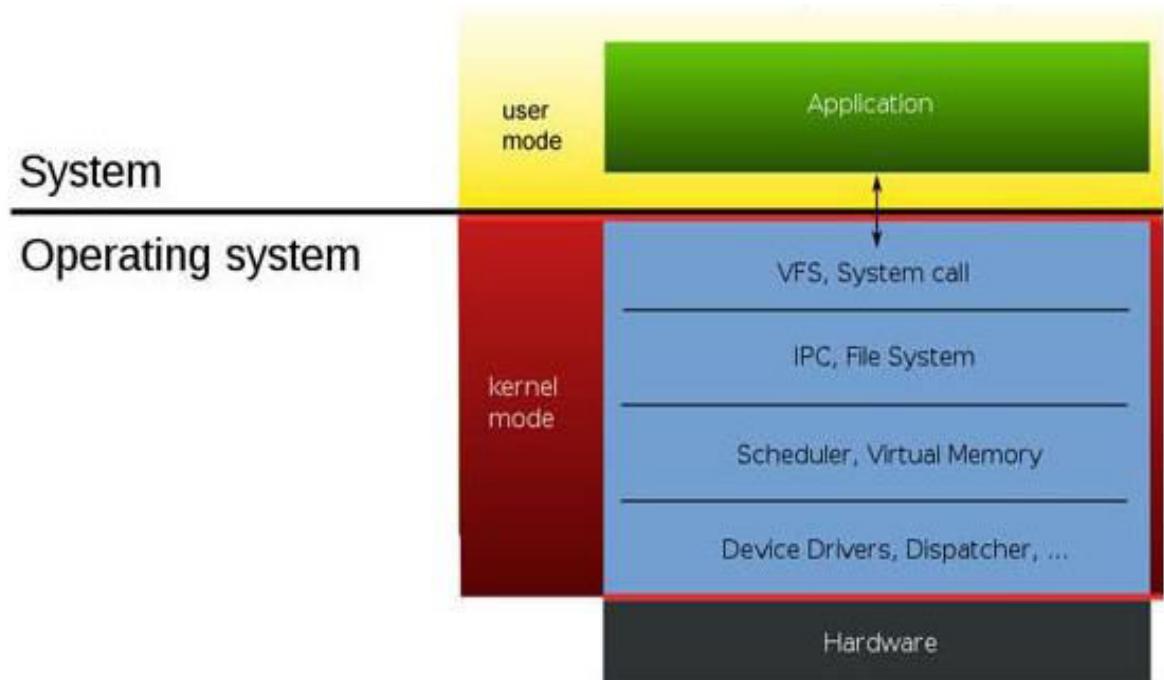
UNIX Architecture

- the original UNIX operating system had limited structuring
- consists of two separable parts
 - Systems programs
 - 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



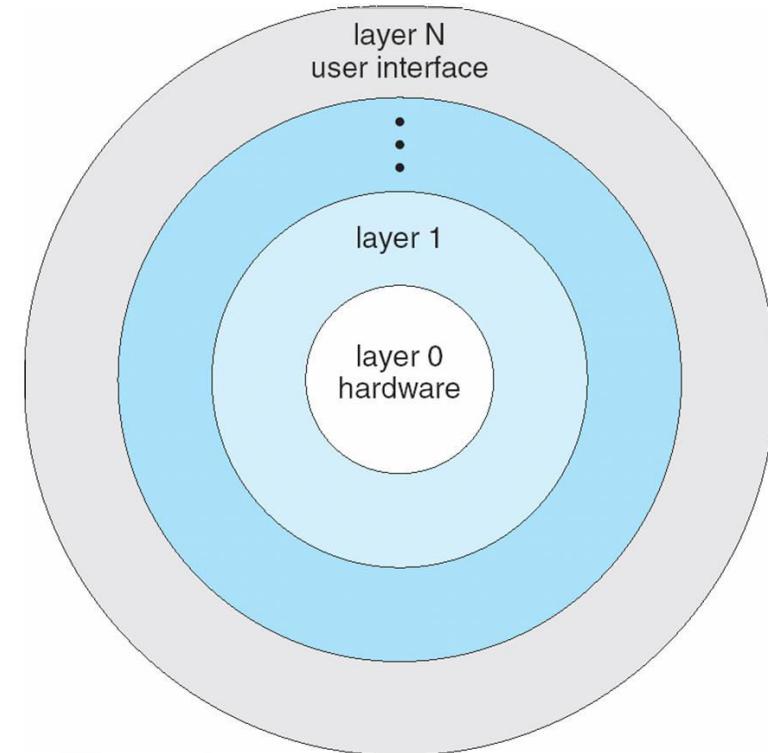
Monolithic Kernel

- entire operating system is working in kernel space
- larger in size
- little overhead in system call interface or in communication within kernel
- faster
- hard to extend
- if a service crashes, whole system is affected
- NOTE:
- There are 2 spaces- user space and kernel space which are basically memory locations/address spaces where user and kernel operations can take place. In monolithic , both these spaces are present together in kernel space making no real distinction between them . user mode is different concept than user space
- Eg., - Linux, Solaris, MS-DOS



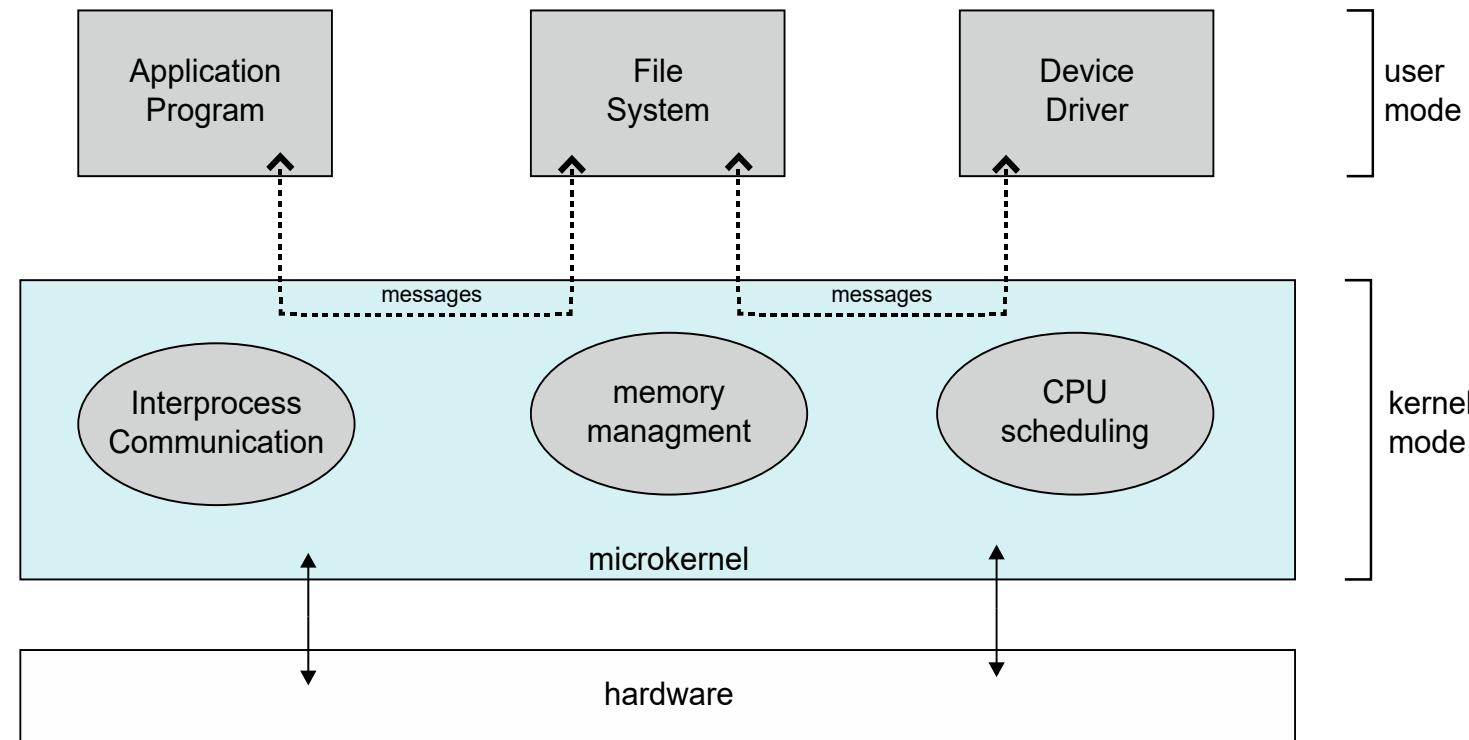
Layered Approach

- The operating system 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
- Advantage- debugging made easy

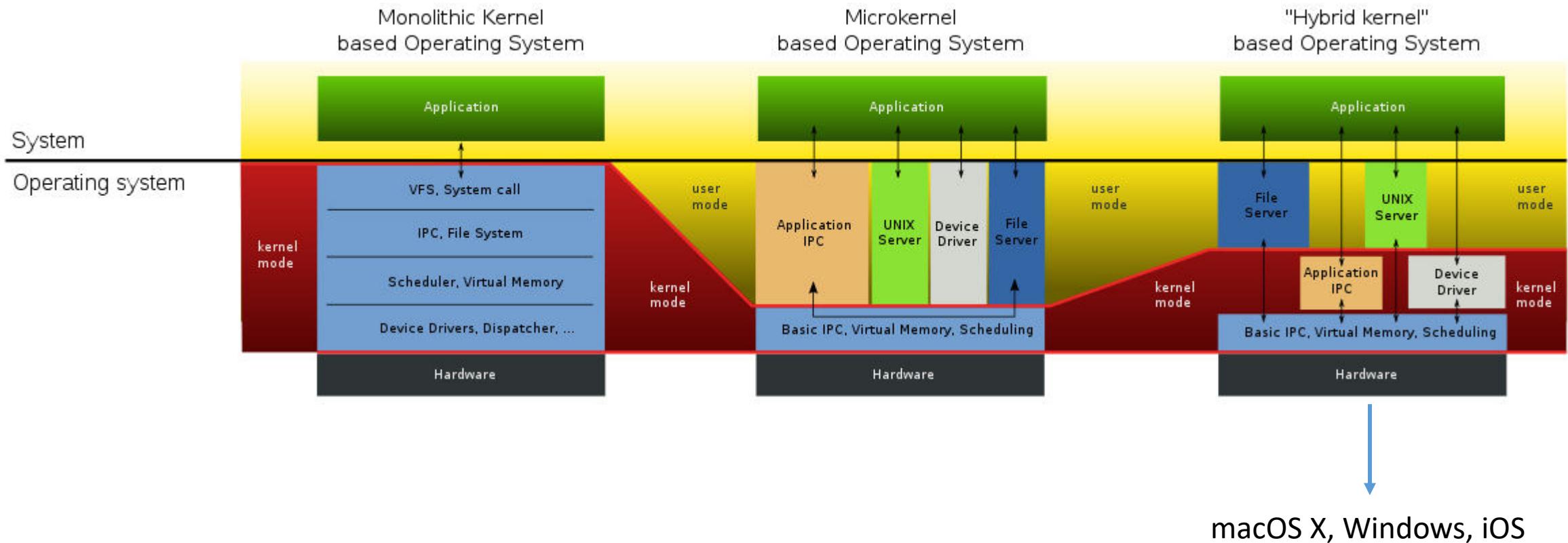


Microkernel

- user services and kernel services are in separate address spaces
- smaller in size
- slower
- extendible, all new services are added to user space
- if a service crashes, working of microkernel is not affected
- more secure and reliable
- eg., Mach, QNX, Windows NT (initial release)
- Drawback ??? *Performance overhead of user space to kernel space communication (even user mode to user mode communication needs to pass through kernel mode)*



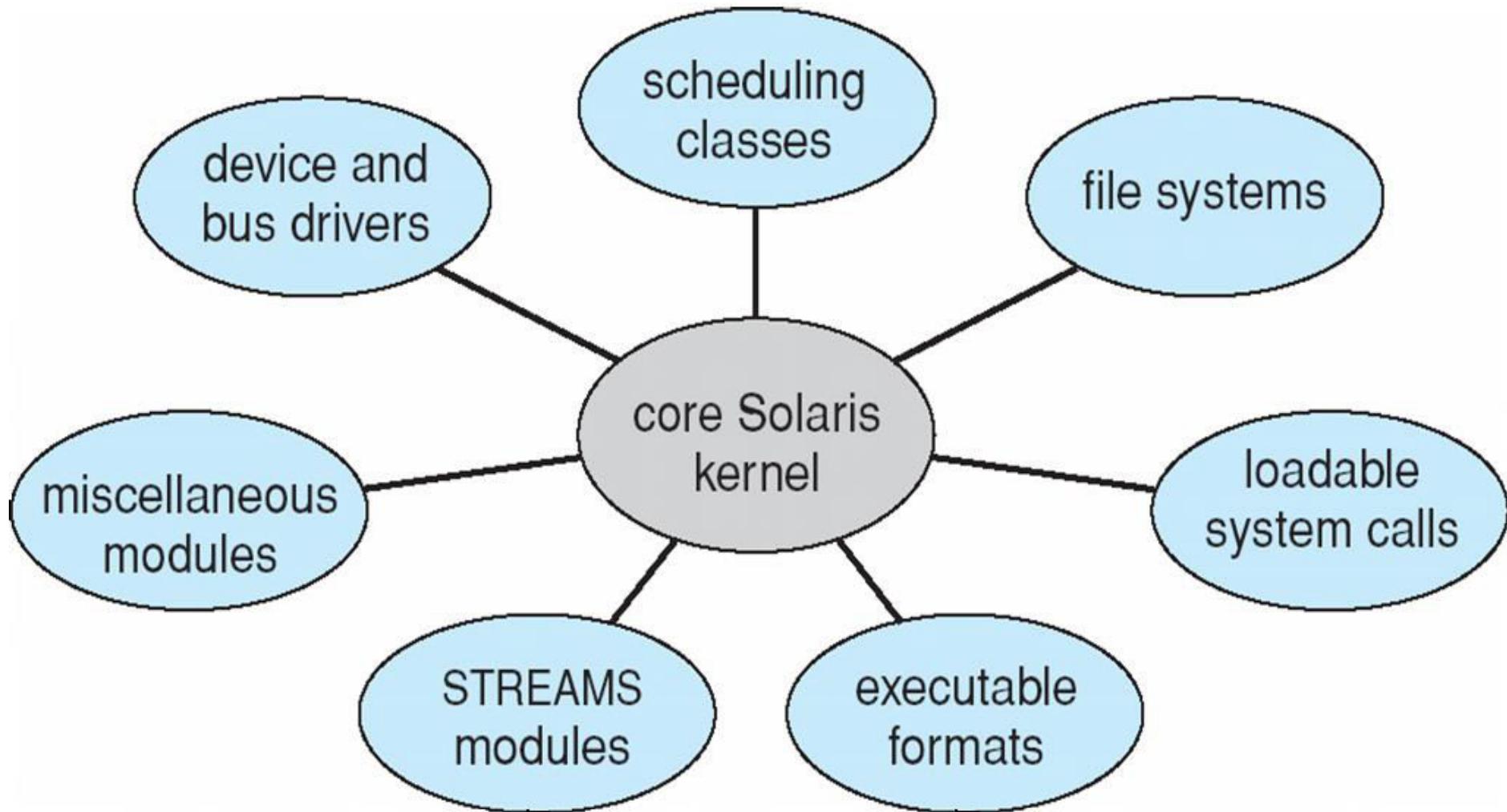
A Comparison



Modules

-
- ❖ loadable kernel modules
 - ❖ kernel has a core set of components
 - ❖ links in additional services via modules, either at boot time or during run time
 - ❖ each module has a well defined interface
 - ❖ dynamically linking services is preferable to adding new features directly to the kernel → does not require recompiling the kernel for every change
 - ❖ better than a layered approach → any module can call any module
 - ❖ better than microkernel → no message passing required to invoke modules

Modules



Operating-System Debugging



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.

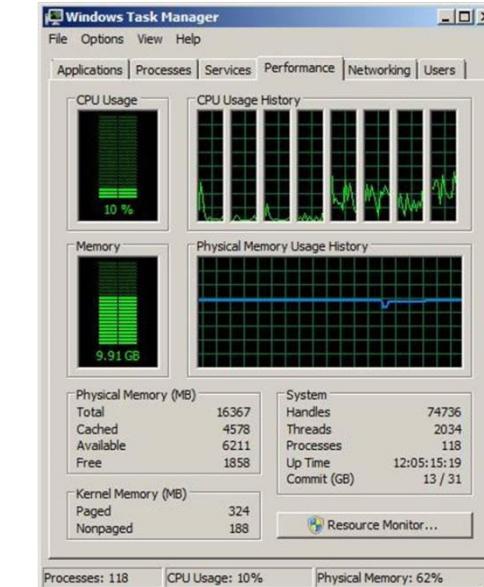
— Brian Kernighan —

Performance Tuning

- Beyond crashes, performance tuning can optimize system performance
- Improve performance by removing bottlenecks
- Sometimes using *trace listings* of activities, recorded for analysis
- OS must provide means of computing and displaying measures of system behavior
- For example, “top” program or Windows Task Manager

```
top - 04:33:30 up 17:16, 1 user, load average: 0.05, 0.01, 0.03
Tasks: 75 total, 2 running, 73 sleeping, 0 stopped, 0 zombie
Cpu(s): 0.0%us, 0.3%sy, 0.0%ni, 88.6%id, 0.0%wa, 0.0%hi, 11.1%si, 0.0%st
Mem: 515348k total, 319956k used, 195392k free, 43432k buffers
Swap: 1048568k total, 0k used, 1048568k free, 201748k cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
4	root	RT	-5	0	0	0	S	0.3	0.0	0:12.90	watchdog/0
32348	root	15	0	2200	996	796	R	0.3	0.2	0:00.04	top
1	root	15	0	2068	612	528	S	0.0	0.1	0:34.24	init
2	root	RT	-5	0	0	0	S	0.0	0.0	0:00.00	migration/0
3	root	34	19	0	0	0	S	0.0	0.0	0:00.04	ksoftirqd/0
5	root	10	-5	0	0	0	S	0.0	0.0	0:04.77	events/0
6	root	10	-5	0	0	0	S	0.0	0.0	0:01.84	khelper
7	root	11	-5	0	0	0	S	0.0	0.0	0:00.06	kthread
10	root	10	-5	0	0	0	S	0.0	0.0	0:00.72	kblockd/0
11	root	20	-5	0	0	0	S	0.0	0.0	0:00.00	kacpid
47	root	20	-5	0	0	0	S	0.0	0.0	0:00.00	cqueue/0
50	root	10	-5	0	0	0	S	0.0	0.0	0:00.00	khubd
52	root	10	-5	0	0	0	S	0.0	0.0	0:00.02	kseriod



System Boot

innovate

achieve

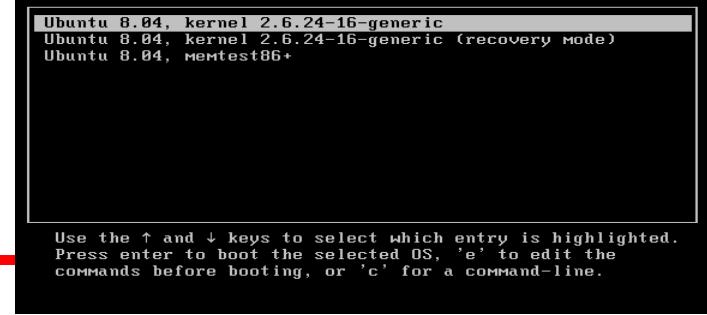
lead

- bootstrap program / bootstrap loader
 - simple bootstrap loader fetches a more complex boot program from disk, which in turn loads kernel
 - instruction register is loaded with a predefined memory location where the initial bootstrap program is located
 - diagnostics to determine the state of the machine
 - *POST (Power-On Self-Test) is the diagnostic testing sequence that a computer's BIOS (basic input/output system) (or "starting program") runs to determine if the computer keyboard, random access memory, disk drives, and other hardware are working correctly*



PCI Devices Listing ...									
	PCI Dev	Fun	Vendor	Device	SVID	SSID	Class	Device Class	I
0	27	0	0006	2668	1458	A005	0403	Multimedia Device	
0	29	0	0006	2658	1458	2658	0C03	USB 1.1 Host Contrlr	
0	29	1	0006	2659	1458	2659	0C03	USB 1.1 Host Contrlr	
0	29	2	0006	2650	1458	2650	0C03	USB 1.1 Host Contrlr	
0	29	3	0006	2656	1458	2656	0C03	USB 1.1 Host Contrlr	
0	29	7	0006	265C	1458	5006	0C03	USB 1.1 Host Contrlr	
0	31	2	0006	2651	1458	2551	0101	IDE Contrlr	
0	31	3	0006	2660	1458	2660	0C03	Serial Bus Controller	
1	0	0	0000	0000	1005	0000	0000	Serial Port(s)	
2	0	0	1203	0212	0000	0000	0100	Mass Storage Contrlr	
2	5	0	11AB	4320	1458	E000	0200	Network Contrlr	
								ACPI Controller	

System Boot



- if the diagnostics pass, the program can continue with the booting steps
- bootstrap will execute the code present in boot block
 - A dedicated **block** usually at the beginning (first block on first track) of a storage medium that holds special data used to start a system
 - Some systems use a boot block of several physical sectors, while some use only one boot sector
 - If a disk contains a boot block it is called a boot disk
 - If a partition contains a boot block it is called a boot partition
- boot block will either contain the remaining bootstrap program or the address on disk and length of the remainder of the bootstrap program
- **GRUB(GRand Unified Bootloader)** is an example of an open-source bootstrap program for Linux systems. Grub provides us with choice to boot from multiple os installed on a computer or select a specific kernel configuration available for a particular OS.
- after the full bootstrap program is loaded, it traverses the file system to locate OS kernel, load kernel into memory and start its execution

Booting process in a gist

- BIOS is stored on the ROM/EPROM called the firmware chip.
- Bootloader is called by the bios. It is basically stored on disk(first sector or block of disk). Bios and bootloader are different.
- When Cpu starts , it needs instruction, ram is empty/undefined at this point,cpu loads instructions from firmware chip where bios is loaded. Firmware chip is present on the motherboard.
- BIOS will perform POST test and do diagnostic checking
- It will initialize different hardware , give success as a beep or failure as a combination of beeps.
- BIOS will now after hardware initialization check different storage media to find bootloader .
Bootloader may be at first sector of disk .
- Now once bootlaoder is located , bios goes out of picture and bootloader takes over.
- Bootloader load remaining part of OS.
- This(bootloader in boot block) is basically first level bootloader which will call the second level bootlaoder,
- Second level bootloader will locate kernel image located on secondary memory. Kernel image will be loaded.
- IN UNIX: INIT process is first process that execute when u power on.



Thank You



BITS Pilani
Hyderabad Campus

OPERATING SYSTEMS (CS F372)

Processes

Dr. Barsha Mitra
CSIS Dept., BITS Pilani, Hyderabad Campus



Process Concept

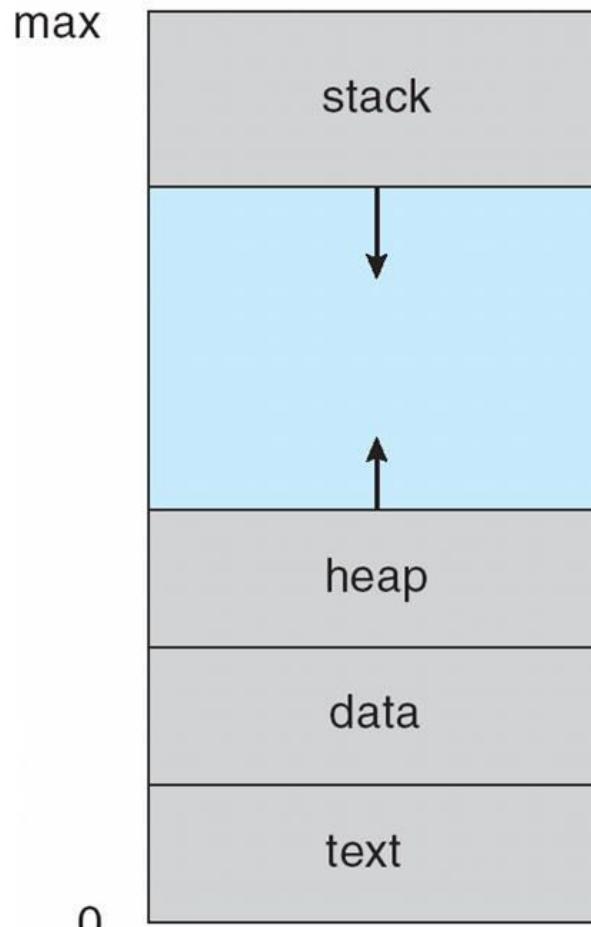
- ❖ An operating system executes a variety of programs
 - ❖ Batch system – **jobs**
 - ❖ Time-shared systems – **user programs** or **tasks**
- ❖ Terms **job** and **process** used interchangeably
- ❖ **Process** – a program in execution; process execution must progress in sequential fashion

Process Concept

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

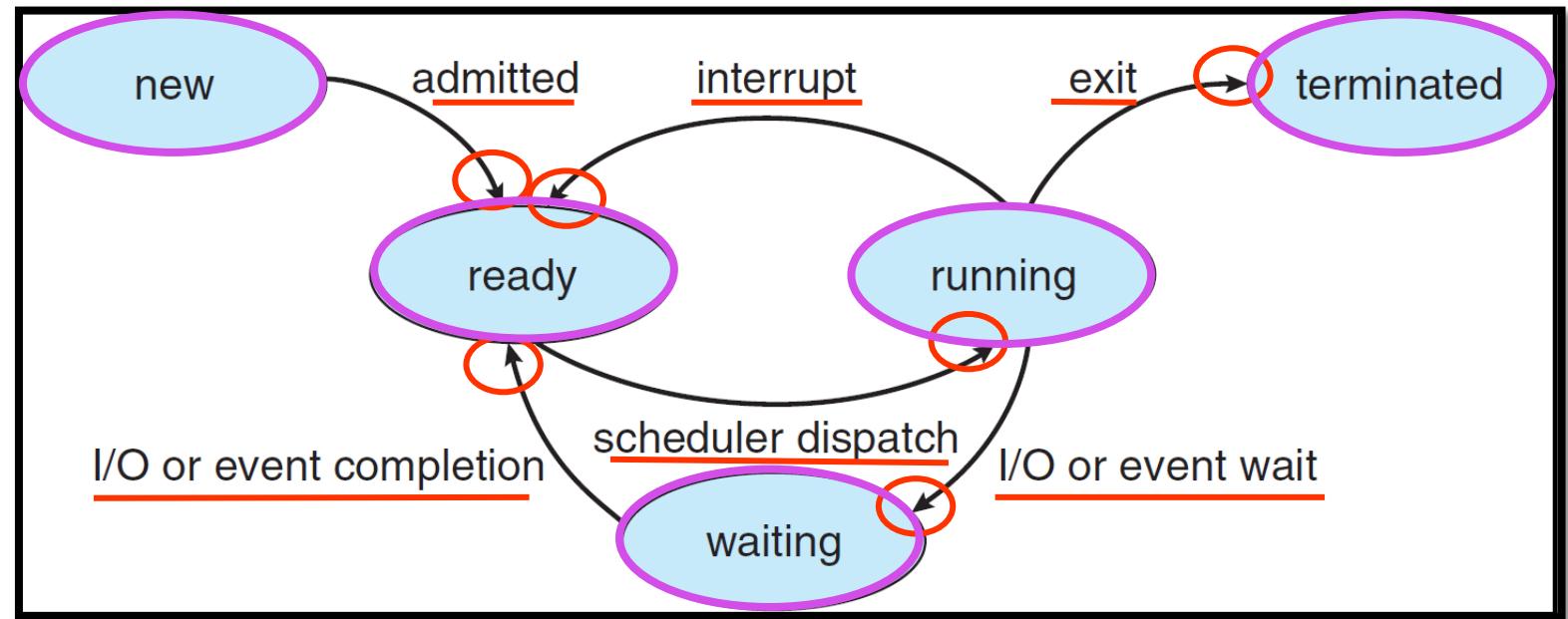
Process Concept

- ❖ Program is ***passive*** entity stored on disk (**executable file**), process is ***active***
 - ❖ Program becomes process when executable file loaded into memory
- ❖ Execution of program started via GUI mouse clicks, command line entry of its name, etc
- ❖ One program can be several processes
 - ❖ Consider multiple users executing the same program



States of Process

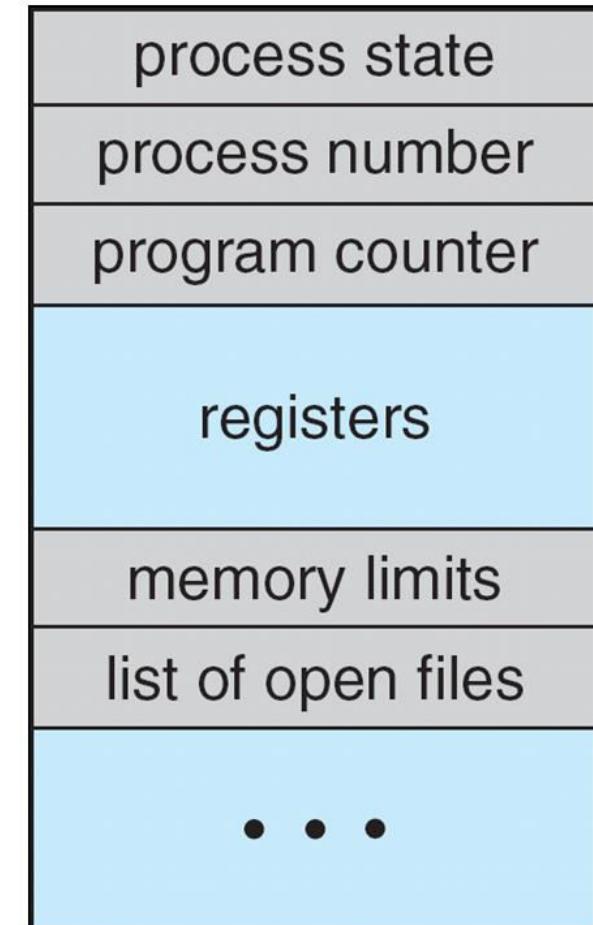
- ❖ **new:** The process is being created(not yet loaded into main memory and not ready for execution)
- ❖ **running:** Instructions are being executed
- ❖ **waiting:** The process is waiting for some event to occur
- ❖ **ready:** The process is waiting to be assigned to a processor
- ❖ **terminated:** The process has finished execution



Process Control Block(PCB)

- ❖ PCB is a data structure which holds several kind of information about a particular process
- ❖ Each process has its own PCB
- ❖ What all PCB holds:-

 1. **Process state** – new, ready, running, waiting, etc.
 2. **Program counter** – location of instruction to next execute
 3. **CPU registers** – contents of all process-centric registers
 4. **CPU scheduling information**- priorities, scheduling queue pointers, scheduling parameters
 5. **Memory-management information** – memory allocated to the process
 6. **Accounting information** – CPU used, clock time elapsed since start, time limits, account nos., process nos.
 7. **I/O status information** – I/O devices allocated to process, list of open files



Process Creation

- ❖ Parent process creates children processes, which, in turn create other processes, forming a tree of processes
- ❖ Ex- In unix, INIT process is the first process to be created, then all others process on the system are either direct children of init or are descendants of init.(NOTE: nowadays init process has been replaced with system D),
- ❖ so ROOT PROCESS of tree in modern unix system is systemD
- ❖ Ex-in mac based system, LAUNCH D is the very first process to be created and executed,so is root process is launchD
- ❖ PID of ROOT PROCESS is 1.
- ❖ Generally, process identified and managed via a process identifier (pid), integer number
- ❖ Resource sharing options (CPU time, memory, files, I/O devices)
 - ❖ 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

- ❖ Address space
 - ❖ Address space of Child is a duplicate of Address space of parent (same program and data) (IMP: It doesn't mean both address space are same , but they are duplicate, ex – CHILD WILL HAVE SAME PROGRAM AND DATA VARIABLES AS PARENT HAS BUT THEY ARE NOT SHARING THESE, if parent has variable x , then child will also have variable x , but these two variables are two different independent variables)
 - ❖ Child has a program loaded into it, that means child can execute different program and perform different task than its parent
- ❖ UNIX examples
 - ❖ fork() system call creates new process
 - ❖ exec() system call used after a fork() to replace the process's memory space with a new program
 - ❖ ps –el command gives all information about all process running currently(in window same is done using tasklist command)

Process Creation

- ❖ **fork()**
 - ❖ address space of child process is a copy of parent process
 - ❖ both child and parent continue execution at the instruction after fork()
 - ❖ return code for fork() is 0 for child
 - ❖ return code for fork() is non-zero (child *pid*) for parent
- ❖ **exec()**
 - ❖ loads a binary file into memory and starts execution
 - ❖ destroys previous memory image
 - ❖ It tells/allows the child to perform some tasks different than its parent
 - ❖ call to exec() does not return unless an error occurs, hence any statement written after execlp() command will not be executed for that process
- ❖ **wait()**
 - ❖ parent can issue wait() to move out of ready queue until the child is done
 - ❖ NOTE: Assume that a parent has 3 child processes, it has called for wait function, so as soon as any one child terminates , parent will resume its execution, ie, wait() doesn't wait for all children to terminate but even 1 child termination is enough for wait() function to return. Incase two or more children terminate, then pid of any of the child process will be randomly selected and returned to the parent

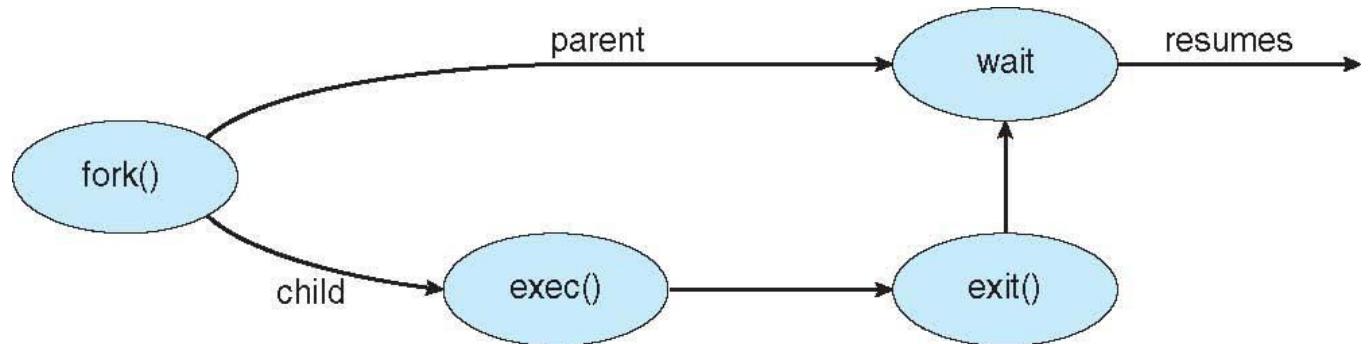
Process Creation

```
#include <sys/types.h> //pid_t datatype is in this header file
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
int main()
{
    pid_t pid;
    pid = fork(); /* fork a child process , this will create a
    child process and return 0 to child process and return pid of child to parent process*/
}
```

//imp: two processes start execution from here now-on

```
if (pid < 0) { /* error occurred */
    fprintf(stderr, "Fork Failed");
    return 1;
}
else if (pid == 0) { /* child process */
    printf("Child Process\n");
    execvp("/bin/ls","ls",NULL);
}

//If exec() executes normally , then anything written in this line,
//ie, below exec() function will never be executed
}
else { /* parent process */
    wait(NULL); /* parent waits for child to complete */
    printf("Child Complete");
}
```



OUTPUT:

Child Process

a.out

Documents examples.desktop MyPrograms Pictures Templates

Desktop Downloads Music parent.c Public Videos

Child Complete

NOTE: examples of using execvp()

- execvp("/home/yashank","./","mycode.","out",NULL);
- execvp("/bin/ls","ls","-al",NULL);

Example 1

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

int main()
{
    fork();
    printf ("hello\n");
    return (0);
}
```

hello
hello

Example 2

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

int main()
{
    int x;
    x = fork();
    if (x == 0 )
        printf ("Child Process :%d", x);
    else
        printf ("I am parent : %d", x);
    return (0);
}
```

Output:

Child Process : 0

I am Parent : 1234

Another Possible output:

I am Parent : 1234

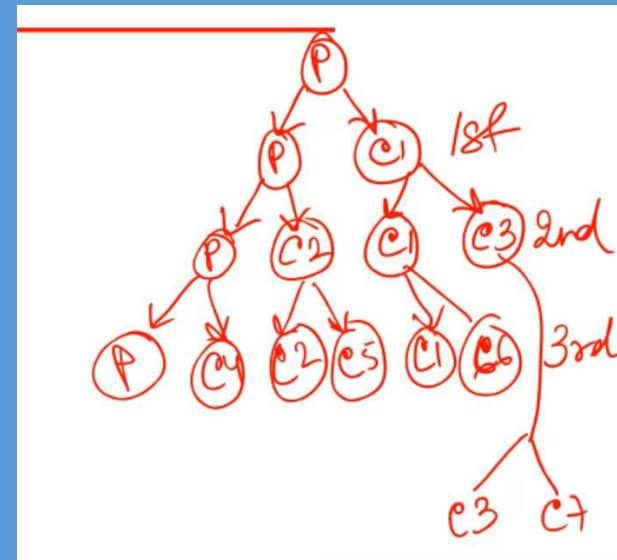
Child Process : 0

Example 3

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

int main()
{
    fork();
    fork();
    fork();
    printf("Hello");
    return (0);
}
```

Output:
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello



Process Creation

```
int main()
{
    pid_t pid;
    pid = fork();
    if (pid < 0) {
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) {
        printf("Child Process\n");
        printf("child pid = %d\n", getpid());
    }
    else {
        printf("parent pid = %d\n", getpid());
        wait(NULL);
        printf("Child Complete");
    }
    return 0;
}
```

OUTPUT:

parent pid = 6597
Child Process
child pid = 6598
Child Complete

Process Creation

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include<sys/wait.h>
int main()
{
    pid_t pid; int x = 10;
    pid = fork();
    if (pid < 0) {
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) {
        printf("Child Process: x = %d\n", x);
        execlp("/bin/ls","ls",NULL);
    }
    else {
        printf("Parent Process: x = %d\n", x);
        wait(NULL);
        printf("Child Complete");
    }
    return 0;
}
```

OUTPUT:

Parent Process: x = 10

Child Process: x = 10

a.out Documents examples.desktop MyPrograms Pictures Templates
Desktop Downloads Music parent.c Public Videos
Child Complete

Process Creation

```
int main()
{
    pid_t pid;
    int x = 10;
    pid = fork();
    if (pid < 0) {
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) {
        x = x + 10;
        printf("Child Process: x = %d\n", x);
    }
    else {
        wait(NULL);
        printf("Parent Process: x = %d\n", x);
        printf("Child Complete");
    }
    return 0;
}
```

OUTPUT:

Child Process: x = 20
Parent Process: x = 10
Child Complete

Process Creation

```
int main()
{
    pid_t pid;
    int x = 10;
    pid = fork();
    if (pid < 0) {
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) {
        for(long i = 0; i < 50000000000; i++);
        printf("Child Process: x = %d\n", x);
    }
    else {
        x = x + 10;
        wait(NULL);
        printf("Parent Process: x = %d\n", x);
        printf("Child Complete");
    }
    return 0;
}
```

OUTPUT:

Child Process: x = 10
Parent Process: x = 20
Child Complete

Process Termination

- ❖ Process executes last statement and then asks the operating system to delete it using the `exit()` system call.
- ❖ May return status data from child to parent (via `wait()`)
- ❖ Process' resources are deallocated by operating system
- ❖ Parent may terminate the execution of children processes because:
 - ❖ Child has exceeded allocated resources limit
 - ❖ Task assigned to child is no longer required
 - ❖ 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 exist if 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_t pid;  
int status;  
pid = wait(&status); //parent can tell which child has terminated
```
- ❖ If no parent waiting (did not invoke **wait()** till then) process is a **zombie**
 $(\text{if child terminates and parent doesn't call wait() or calls wait() after some time;in the meantime the child will be in a state called zombie state, where it has terminated but its entry is still present in process table.When wait() is called,its entry gets deleted and it doesn't remain zombie anymore})$
- ❖ If parent terminated, process is an **orphan**
 $(\text{when child is still running but parent terminates without calling wait(), then that child process is called orphan process. Orphan process will eventually be converted to zombie once it is terminated.})$

HOW TO REMOVE ORPHAN & ZOMBIE RECORD FROM PROCESS TABLE?

Os does it by calling **wait()** in the root process itself(**init ,systemd for unix**) ,so that these zombie and orphan record can be removed from process table

Zombie Process

```
int main()
{
    pid_t child_pid = fork();
    if (child_pid > 0) {
        sleep(10);      → zombie
        wait(NULL);   → no longer a zombie
        sleep(200);
    }
    else{
        printf("\n%d",getpid());
        exit(0);
    }
    return 0;
}
```

	F	S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD
	1	Z	1001	17404	17403	0	80	0	-	0	-	pts/0	00:00:00	a.out <defunct>

Orphan Process

```
int main()
{
    pid_t child_pid = fork();
    if (child_pid > 0){
        printf("\nParent process: %d\n",getpid());
        sleep(6);
    }
    else{
        printf("\nParent PID: %d\n",getppid());
        sleep(20);
        printf("\nChild Process: %d",getpid());
        printf("\nParent PID: %d",getppid());
        exit(0);
    }

    return 0;
}
```

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**
 - ❖ **Message passing**

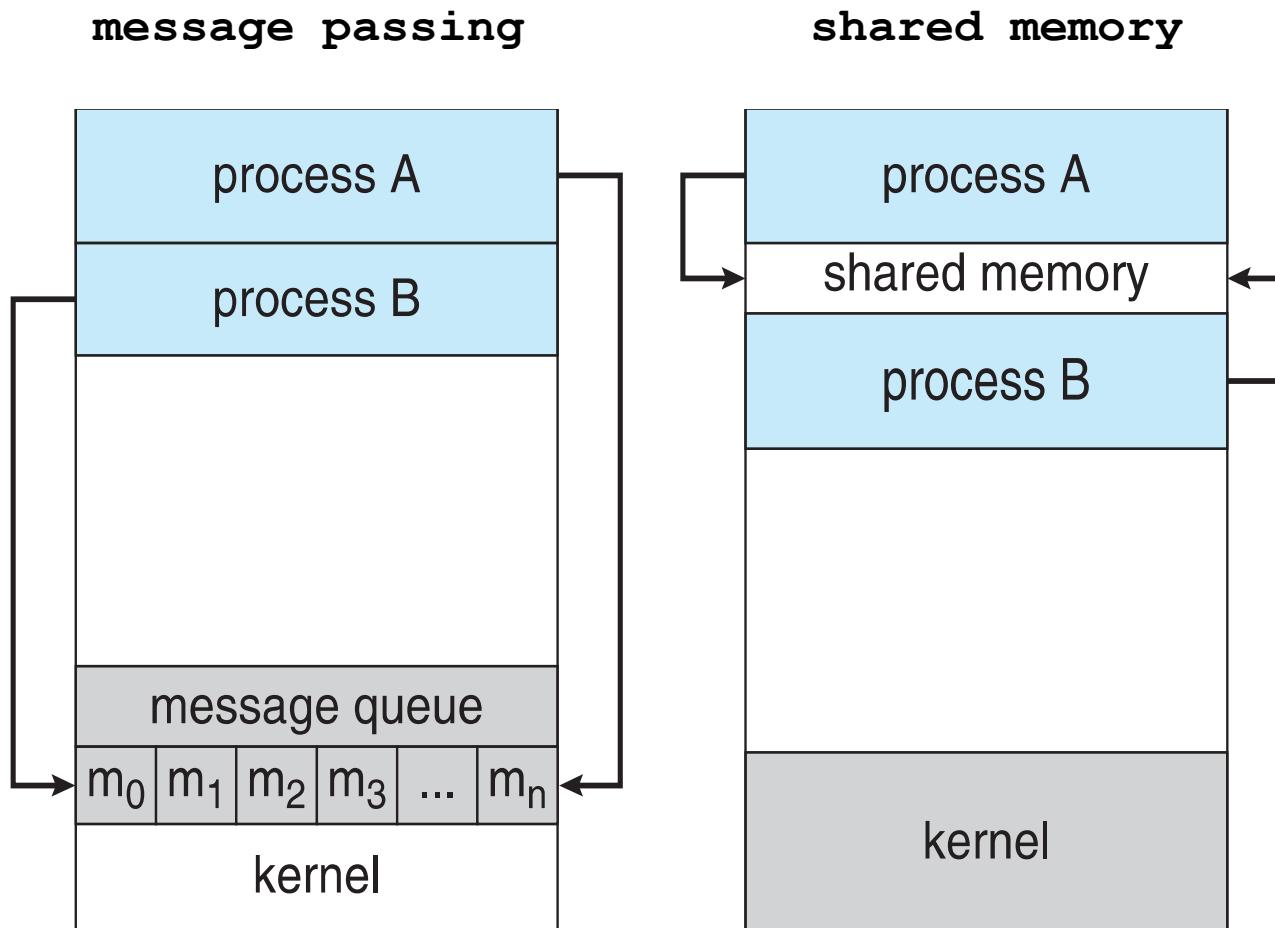
In message passing :

- Only small chunk of data can be communicated

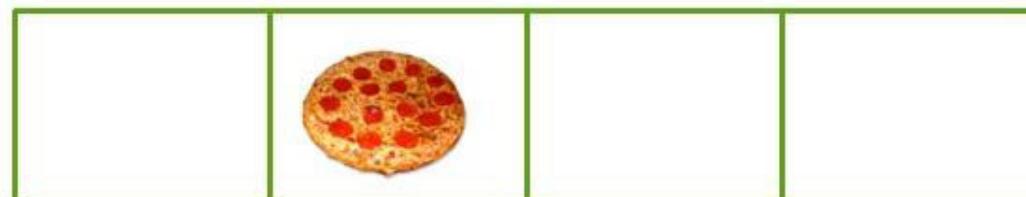
In shared memory:

- It is faster than message passing
- Large amount information can be shared
- One process creates shared memory segment and other processes link/attach to that memory segment to communicate with each other (creation, attaching, detaching will require system call but writing/reading in shared memory are normal/don't require system call)
- Two processes are not allowed to write simultaneously in the shared memory segment, but two/more can read simultaneously

Interprocess Communication



Producer-Consumer Problem



IPC – Shared Memory

- ❖ 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, ie, we don't require system calls to read and write information in shared memory segment
 - ❖ Provide mechanism that will allow the user processes to synchronize their actions when they access shared memory
-

Producer-Consumer Problem

- ❖ Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
 - ❖ **unbounded-buffer** places no practical limit on the size of the buffer
 - ❖ **bounded-buffer** assumes that there is a fixed buffer size
- ❖ Shared data, reside in a region of memory shared by producer & consumer

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

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

- ❖ Solution is correct, but can only use BUFFER_SIZE-1 elements
-

Bounded Buffer - Producer

```

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

```



Consumer

```

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

```

IPC – Message Passing

- ❖ Mechanism for processes to communicate and to synchronize their actions
- ❖ processes communicate with each other without resorting to shared variables, no sharing of address space
- ❖ IPC facility provides two operations:
 - ❖ **send (*message*)**
 - ❖ **receive (*message*)**
- ❖ The *message* size is either fixed or variable
- ❖ If processes P and Q wish to communicate, they need to:
 - ❖ Establish a **communication link** between them
 - ❖ Exchange messages via `send ()` / `receive ()`

Message Passing – 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
 - ❖ Processes only need to know each other's identity
 - ❖ A link is associated with exactly one pair of communicating processes
 - ❖ Between each pair there exists exactly one link

Message Passing – Indirect Communication



- ❖ Messages are directed and received from mailboxes (also referred to as ports)
 - ❖ Each mailbox has a unique ID
 - ❖ Processes can communicate only if they share a mailbox
- ❖ Properties of communication link
 - ❖ Link is 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, each link corresponds to one mailbox

Message Passing – Indirect Communication



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

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 by the receiving process or mailbox
 - ❖ **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 continues
 - ❖ **Non-blocking receive** -- the receiver receives:
 - ❖ A valid message, or
 - ❖ Null message

Buffering

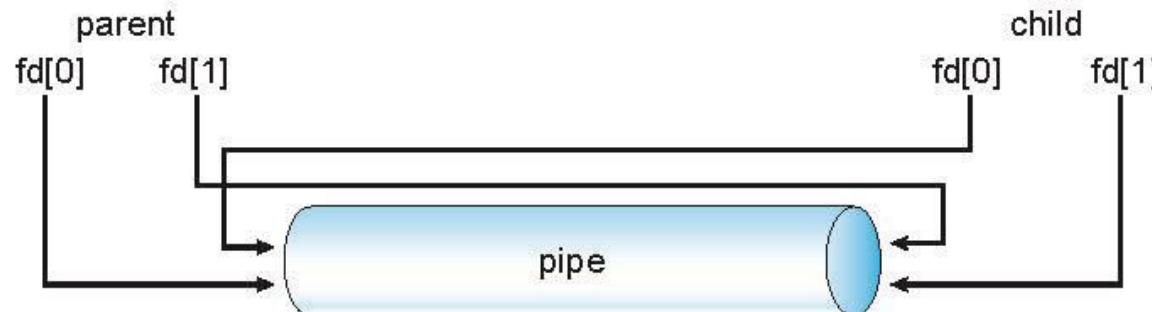
- ❖ messages exchanged by communicating processes reside in a temporary queue
- ❖ implemented in one of three ways
 - ❖ **Zero capacity** – no messages are queued, link can't have any waiting messages, sender must block until receiver receives message
 - ❖ **Bounded capacity** – queue is of finite length of n messages, sender need not block if queue is not full, sender must wait if queue full; receiver need not be blocked irrespective of queue is full or not
 - ❖ **Unbounded capacity** – infinite length queue, sender never blocks

Pipe

- ❖ Acts as a conduit allowing two processes to communicate
- ❖ Issues:
 - ❖ Is communication unidirectional or bidirectional?
 - ❖ In the case of two-way communication, is it half (like walkie-talkie => one way travelling at a given time) or full-duplex (like mobile)?
 - ❖ Must there exist a relationship (i.e., **parent-child**) between the communicating processes?
 - ❖ Can the pipes be used over a network?
- ❖ **Ordinary pipes** –
 - ❖ cannot be accessed from outside the process that created it
 - ❖ parent process creates a pipe and uses it to communicate with a child process that it created
- ❖ **Named pipes** – can be accessed without a parent-child relationship

Ordinary Pipe

- ❖ 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 unidirectional
- ❖ Require parent-child relationship between communicating processes
- ❖ Windows calls these **anonymous pipes**
- ❖ **Fd[0] is read**
- ❖ **Fd[1] is write**



Ordinary Pipe

- ❖ ordinary pipe can't be accessed from outside the process that created it
- ❖ parent process creates a pipe and uses it to communicate with a child process that it creates via fork()
- ❖ child inherits the pipe from its parent process like any other file

Ordinary Pipe

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

#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1

int main(void)
{
char write_msg[BUFFER_SIZE] = "Greetings";
char read_msg[BUFFER_SIZE];
int fd[2];
pid_t pid;

/* create the pipe */
if (pipe(fd) == -1) {
    fprintf(stderr,"Pipe failed");
    return 1;
}

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

if (pid < 0) { /* error occurred */
    fprintf(stderr, "Fork Failed");
    return 1;
}

if (pid > 0) { /* parent process */
    /* close the unused end of the pipe */
    close(fd[READ_END]);

    /* write to the pipe */
    write(fd[WRITE_END], write_msg, strlen(write_msg)+1);

    /* close the write end of the pipe */
    close(fd[WRITE_END]);
}

else { /* child process */
    /* close the unused end of the pipe */
    close(fd[WRITE_END]);

    /* read from the pipe */
    read(fd[READ_END], read_msg, BUFFER_SIZE);
    printf("read %s",read_msg);

    /* close the write end of the pipe */
    close(fd[READ_END]);
}

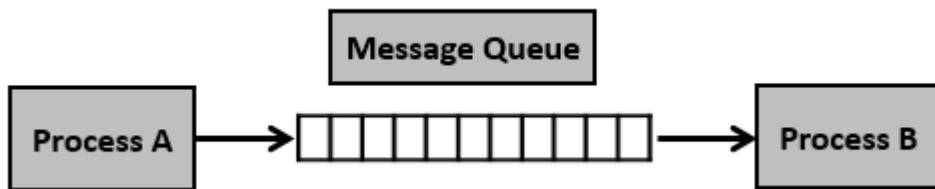
return 0;
}
```

Named Pipe

- ❖ 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
- ❖ Do not cease to exist if the communicating processes have terminated
- ❖ Provided on both UNIX and Windows systems
- ❖ Referred to as FIFOs in UNIX systems

Message Queue

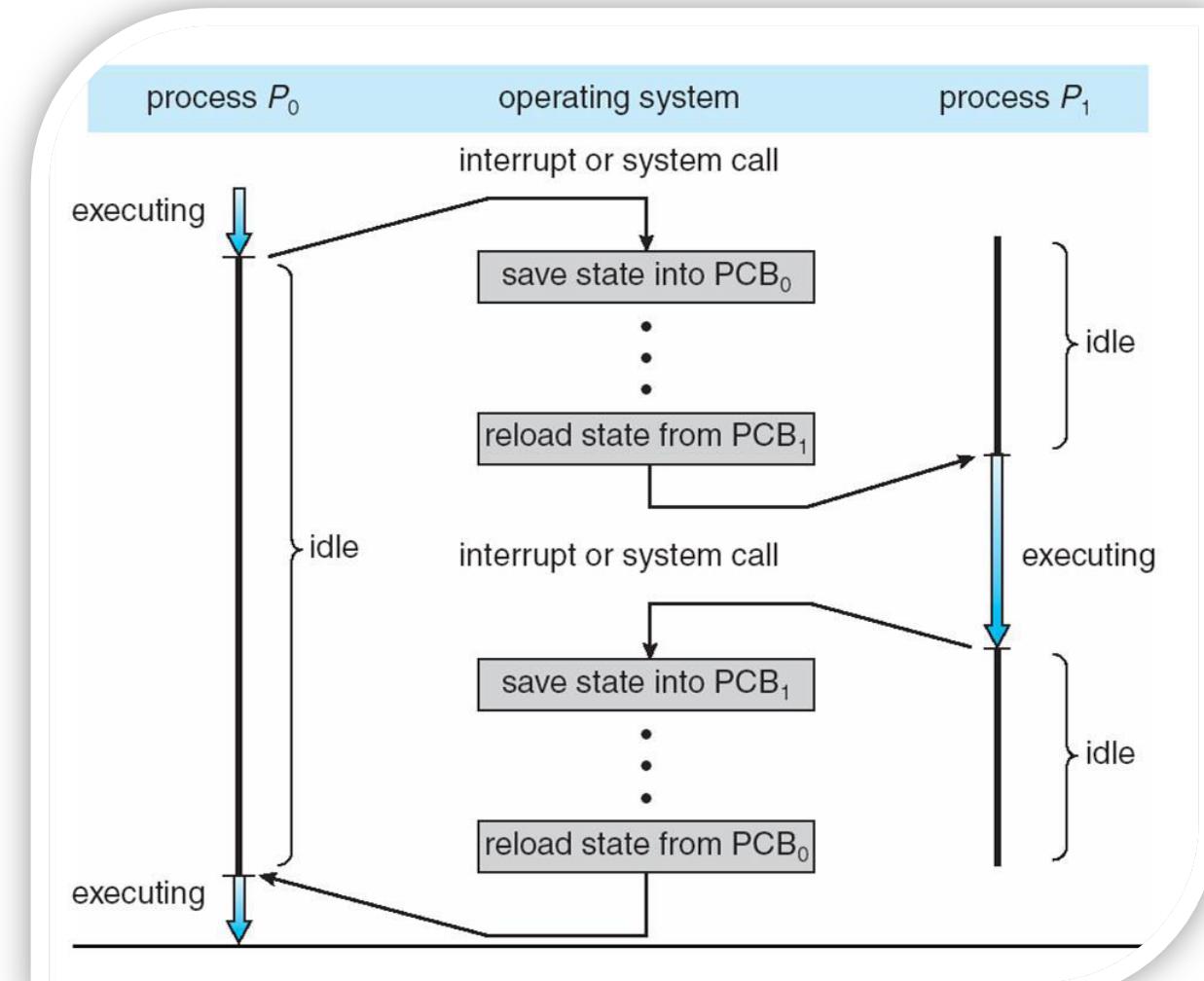
- asynchronous communication
- messages placed onto the queue are stored until the recipient retrieves them



- **Step 1** – Create a message queue or connect to an already existing message queue (`msgget()`)
- **Step 2** – Write into message queue (`msgsnd()`)
- **Step 3** – Read from the message queue (`msgrcv()`)
- **Step 4** – Perform control operations on the message queue (`msgctl()`)

Context Switch

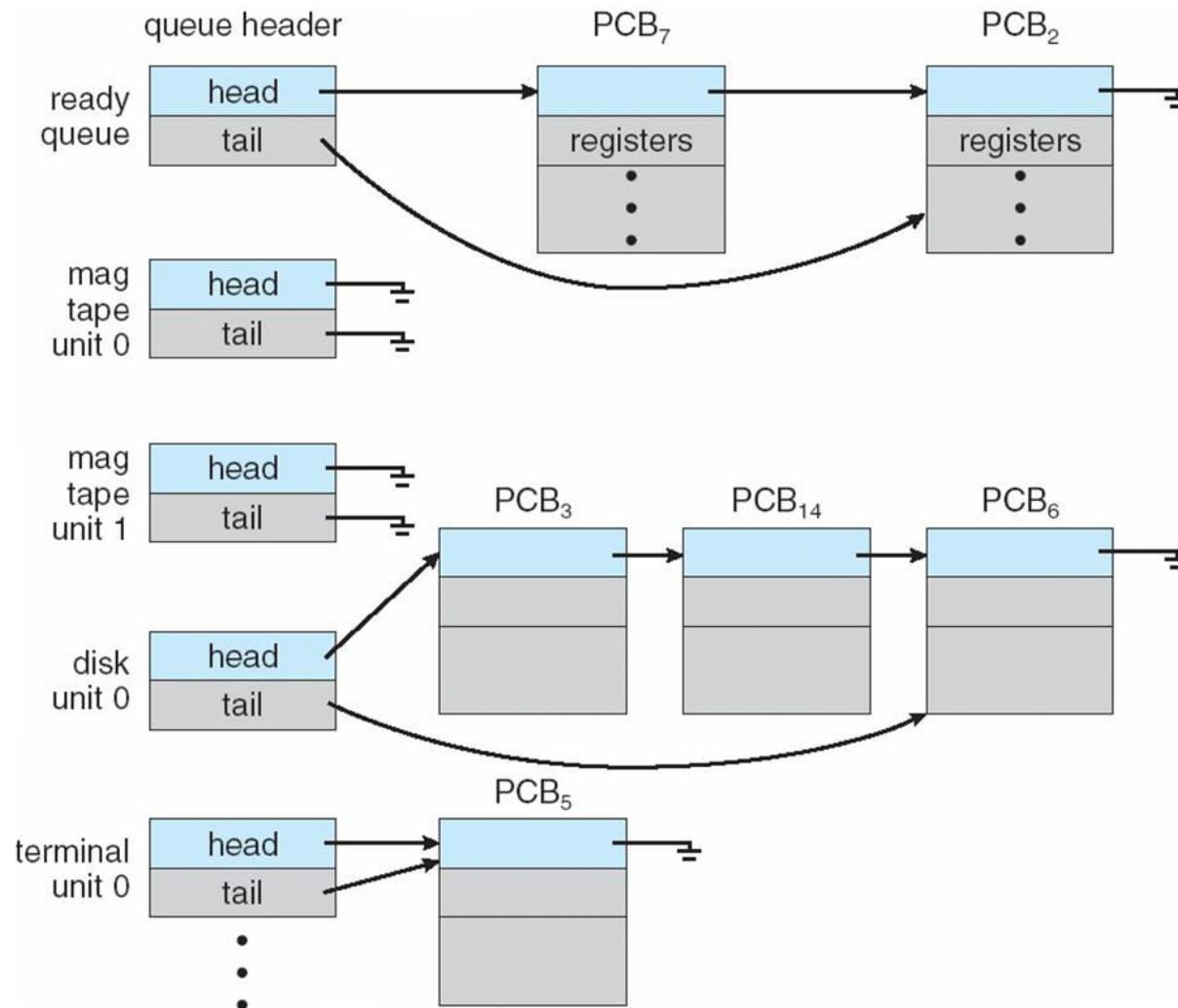
- ❖ When CPU switches to another process, system must **save state** of the old process and load the **state** for the new process via a **context switch**
- ❖ **Context** of a process represented in the PCB (CPU registers contents, process state, memory management info.)
- ❖ Context-switch time is overhead; system does no useful work while switching
- ❖ Context switch Time is dependent on hardware support



Process Scheduling

- ❖ Maximize CPU use, quickly switch processes onto CPU for time sharing
- ❖ **Process scheduler** selects among available processes for next execution on CPU
- ❖ Maintains **scheduling queues** of processes
 - ❖ **Job queue** – set of all processes in the system(set of all process submitted to the system; they are not the process whose image is loaded into the main memory; job queue is stored in secondary queue; initially all queue are present in job queue)
 - ❖ **Ready queue** – set of all processes residing in main memory, ready and waiting to execute, **generally stored as a linked list of PCB** (subset of processes in job queue are brought into main memory, these are called ready queue)
 - ❖ **Device queues** – set of processes waiting for an I/O device(it may happen that many processes require same IO device , so such processes are stored in device queue); device queue is present in the main memory
- ❖ Processes migrate among the various queues

Various Queues

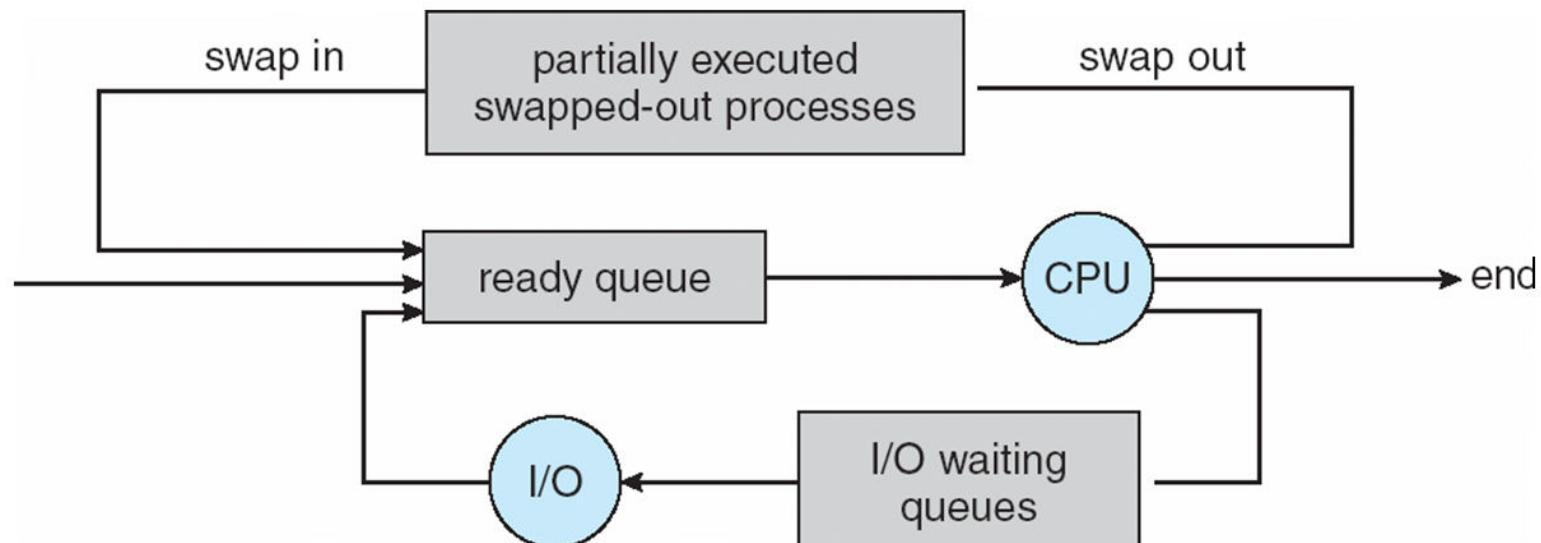


Schedulers

- ❖ **Short-term scheduler (or CPU scheduler)** – selects which process should be executed next and allocates CPU
 - ❖ Sometimes the only scheduler in a system
 - ❖ Short-term scheduler is invoked frequently (milliseconds) \Rightarrow (must be fast)
- ❖ **Long-term scheduler (or job scheduler)** – selects which processes from job queue should be brought into the ready queue
 - ❖ Long-term scheduler is invoked infrequently (seconds, minutes) \Rightarrow (may be slow)
 - ❖ The long-term scheduler controls the **degree of multiprogramming (number of processes in main memory)**
- ❖ Processes can be described as either:
 - ❖ **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
 - ❖ **CPU-bound process** – spends more time doing computations; few very long CPU bursts
- ❖ Long-term scheduler strives for good ***process mix***

Schedulers

- ❖ **Medium-term scheduler** can be added in time sharing systems if degree of multiprogramming needs to decrease
 - ❖ Intermediate level of scheduling
 - ❖ Remove process from memory, store on disk, bring back in from disk to continue execution from where it left off: **swapping**
 - ❖ **Required for improving process mix or for freeing of memory**





Thank You

```
#include <stdio.h>          // READER
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct my_msghdr {
    long mtype;
    char mtext[200];
};

int main(void)
{
    struct my_msghdr buf;
    int msqid;
    long m;
    key_t key;

    if ((key = ftok("writer.c", 'B')) == -1) { //same as writer.c perror("ftok");
        exit(1);
    }

    if ((msqid = msgget(key, 0644)) == -1) { // connect to the queue perror("msgget");
        exit(1);
    }

    printf("Reader: ready to receive messages\n");

    while(1) {
        if (msgrecv(msqid, &buf, sizeof(buf.mtext), buf.mtype, 0) == -1) {
            perror("msgrecv");
            exit(1);
        }
        m=buf.mtype;
        printf("Reader: %ld %s\n", m, buf.mtext);
    }

    return 0;
}
```

```

#include <stdio.h> // WRITER
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct my_msghdr {
    long mtype;
    char mtext[200];
};

int main(void)
{
    struct my_msghdr buf;
    int msqid; /*used by msgget*/
    key_t key; /*used by ftok*/

    /* generate a key*/
    if ((key = ftok("writer.c", 'B')) == -1) {
        perror("ftok");
        exit(1);
    }

    /*creating a msg q*/
    if ((msqid = msgget(key, 0644 | IPC_CREAT)) == -1) {
        perror("msgget");
        exit(1);
    }

    printf("Enter lines of text, ^D to quit:\n");
    /*setting msg type*/
    buf.mtype = 1;

    while(fgets(buf.mtext, sizeof(buf.mtext), stdin) != NULL) {
        int len = strlen(buf.mtext);

        /* ignore newline at end, if it exists */
        if (buf.mtext[len-1] == '\n')
            buf.mtext[len-1] = '\0';

        /*send the msg*/
        if (msgsnd(msqid, &buf, len+1, 0) == -1) /*+1 for '\0' */
            perror("msgsnd");
    }

    /*remove the msg q*/
    if (msgctl(msqid, IPC_RMID, NULL) == -1) {
        perror("msgctl");
        exit(1);
    }

    return 0;
}

```

```

#include <stdio.h> //SHMEM
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>
#include <sys/wait.h>

int main()
{
    int shmid;
    char *shmPtr;
    int n;

    if(fork() == 0) {
        sleep(5); /* To wait for the parent to write */

        if( (shmid = shmget(2041, 32, 0)) == -1 )
            exit(1);

        shmPtr = shmat(shmid, 0, 0);

        if (shmPtr == (char *) -1)
            exit(2);
        printf ("\nChild Reading ....\n\n");

        for (n = 0; n < 26; n++)
            putchar(shmPtr[n]);
        putchar('\n');
    }

    else {
        if( (shmid = shmget(2041, 32, 0666 | IPC_CREAT)) == -1 )
            exit(1);
        shmPtr = shmat(shmid, 0, 0);
        if (shmPtr == (char *) -1)
            exit(2);
        for (n = 0; n < 26; n++)
            shmPtr[n] = 'a' + n;
        printf ("\nParent Writing ....\n\n");
        for (n = 0; n < 26; n++)
            putchar(shmPtr[n]);
        putchar('\n');
        wait(NULL);
        if (shmctl(shmid, IPC_RMID, NULL) == -1 ){
            perror("shmctl");
            exit(-1);
        }
    }
    return 0;
}

```



BITS Pilani
Hyderabad Campus

OPERATING SYSTEMS (CS F372)

Threads

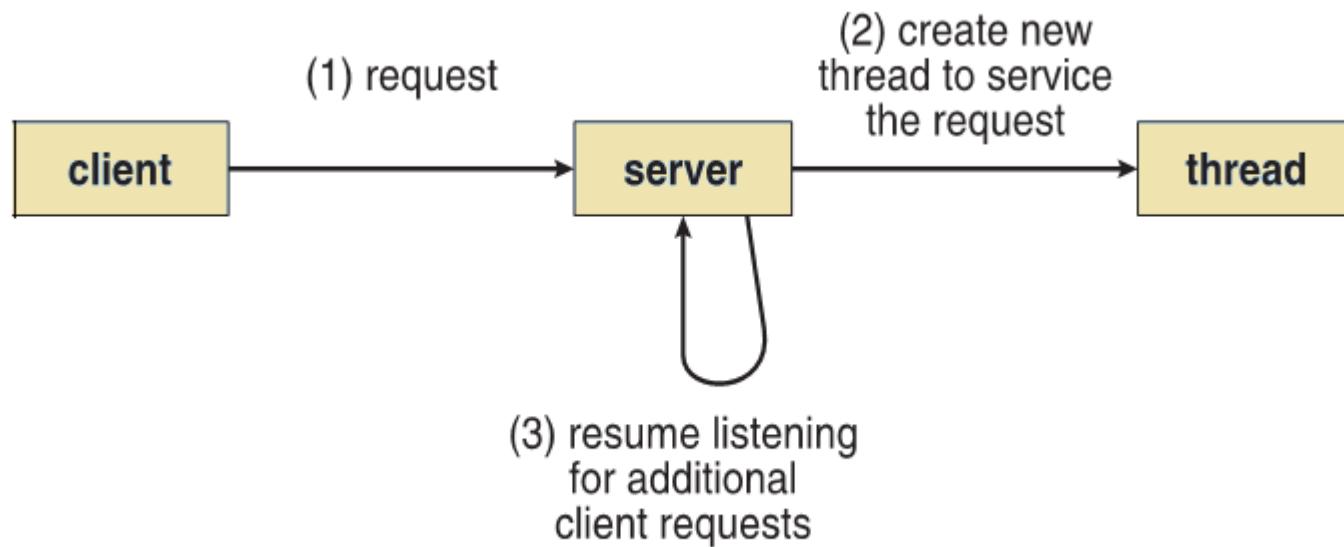
Dr. Barsha Mitra
CSIS Dept., BITS Pilani, Hyderabad Campus



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

Motivation

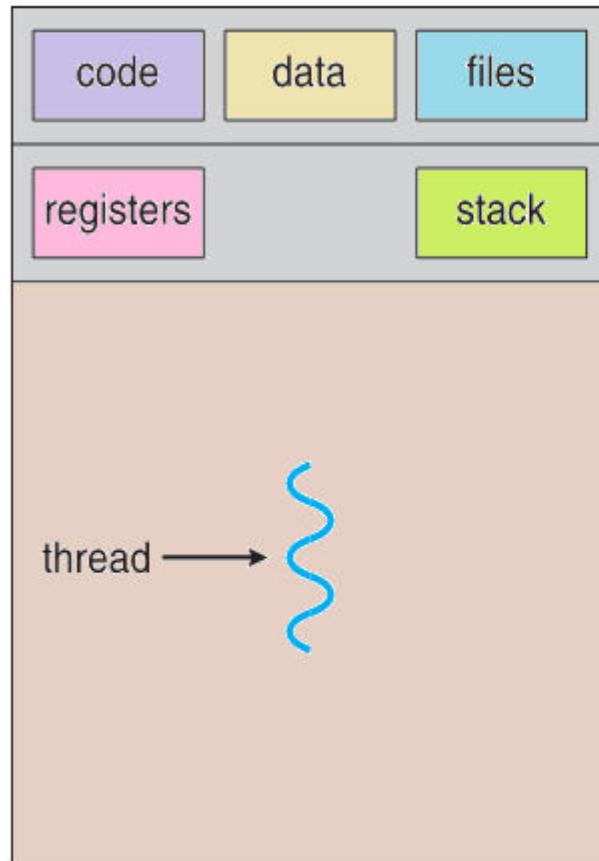


Multithreaded Server Architecture

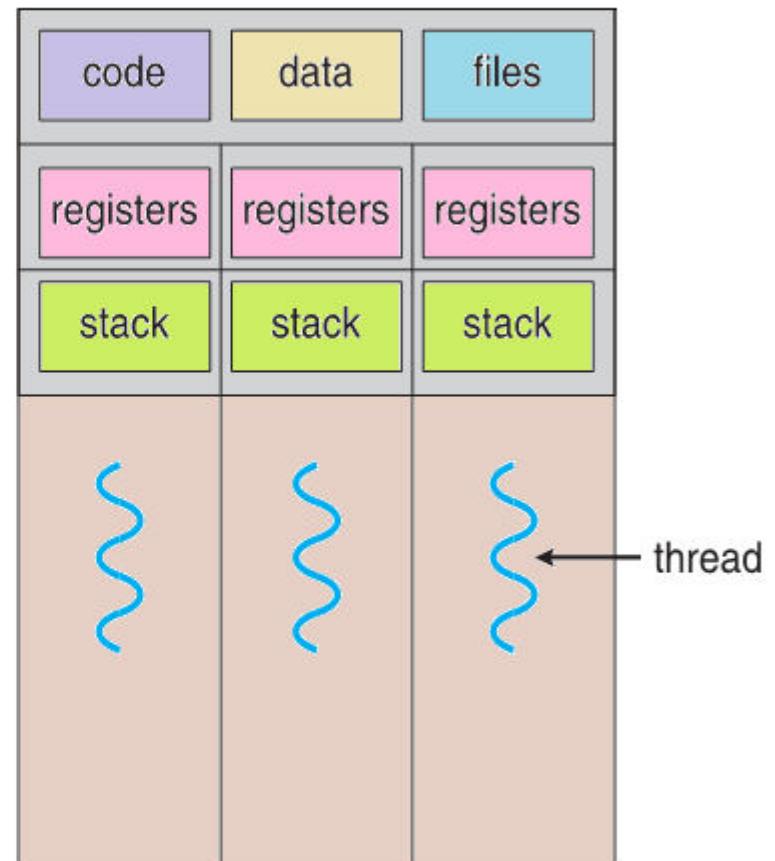
What is Thread?

- ❖ Basic unit of CPU utilization
- ❖ Comprises a thread ID, program counter, registers and stack
- ❖ Shares with other threads belonging to the same program
 - ❖ code section
 - ❖ data section
 - ❖ OS resources like open files

Motivation



single-threaded process



multithreaded process

Benefits

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

Multicore Programming

- ❖ **Multicore or multiprocessor systems** putting pressure on programmers because they have to write multithreaded programs to fully utilize the multiprocessor system , programming challenges include:
 - ❖ Identifying tasks
 - ❖ Balance
 - ❖ Data splitting
 - ❖ Data dependency
 - ❖ Testing and debugging
- ❖ **Parallelism** implies a system can perform more than one task simultaneously; multiprocessor system → each processor does some task at the same time
- ❖ **Concurrency** (illusion of parallelism) supports more than one task making progress, context switching is very fast which looks as if running parallel but not actually
 - ❖ Single processor / core, scheduler providing concurrency but not parallelism

Multicore Programming

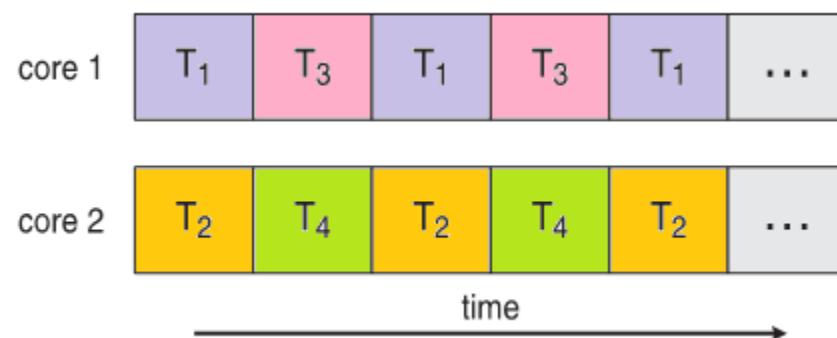
- ❖ Types of parallelism
 - ❖ **Data parallelism** – distributes subsets of the same data across multiple cores, **same operation on each subset**
 - ❖ **Task parallelism** – distributing tasks/threads across cores, **each thread performing unique operation**, threads may be operating on same or different data

Concurrency vs Parallelism

- ❖ Concurrent execution on single-core system:



- ❖ Parallelism on a multi-core system:



User Threads and Kernel Threads



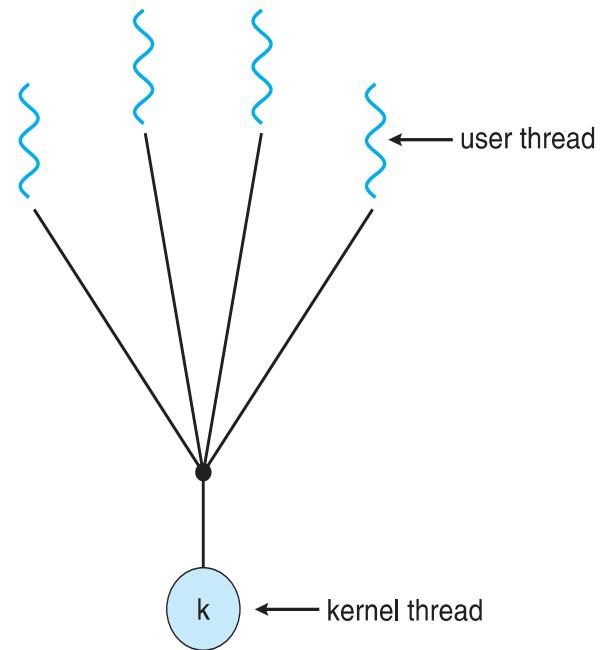
- ❖ **User threads** - management done by user-level threads library without kernel support
- ❖ Three primary thread libraries:
 - ❖ POSIX Pthreads
 - ❖ Windows threads
 - ❖ Java threads
- ❖ **Kernel threads** - Supported and managed by the Kernel
- ❖ **Examples** – virtually all general purpose operating systems support kernel threads, including:
 - ❖ Windows
 - ❖ Solaris
 - ❖ Linux
 - ❖ Tru64 UNIX
 - ❖ Mac OS X
- ❖ Note: each user thread is mapped to kernel thread using LWP, minimum number of lwp required is equal to number concurrent blocking system calls.

Multithreading Models

- ❖ Many-to-One
- ❖ One-to-One
- ❖ Many-to-Many

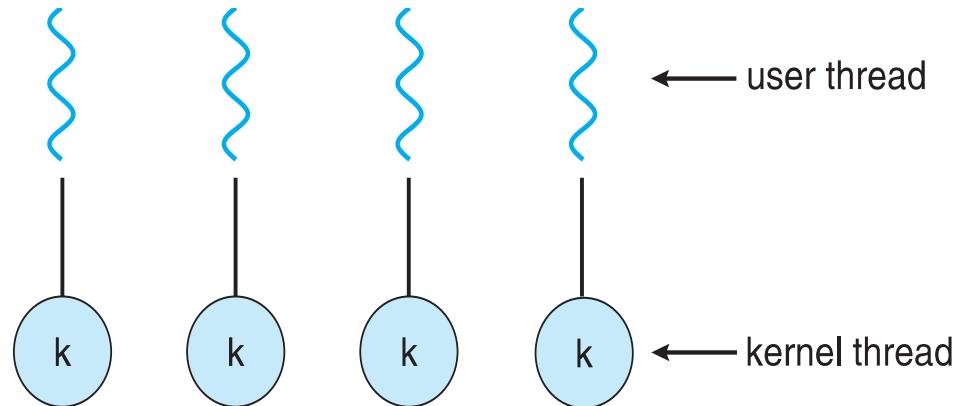
Many-to-One Model

- ❖ Many user-level threads mapped to single kernel thread
- ❖ Thread management done by thread library in user space
- ❖ One thread blocking causes all to block
- ❖ Multiple threads may not run in parallel on multicore system because only one can access kernel at a time
- ❖ Few systems currently use this model
- ❖ Examples:
 - ❖ **Solaris Green Threads**



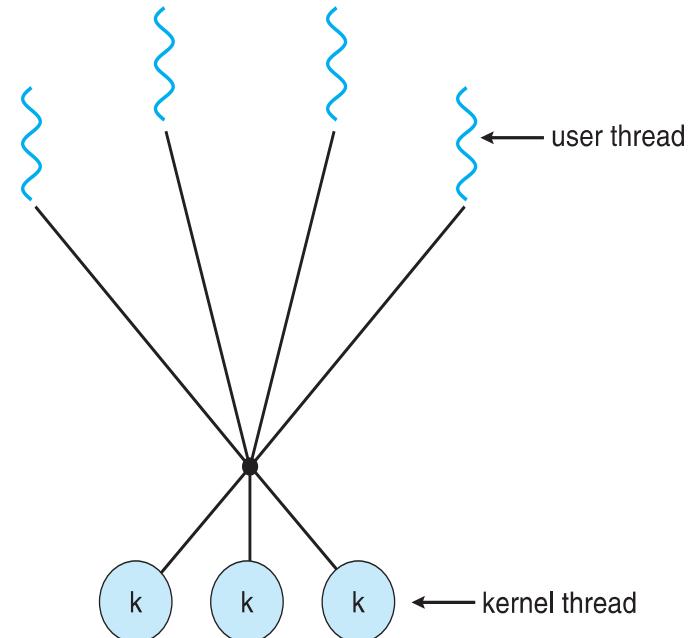
One-to-One Model

- ❖ 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
 - ❖ Solaris 9 and later



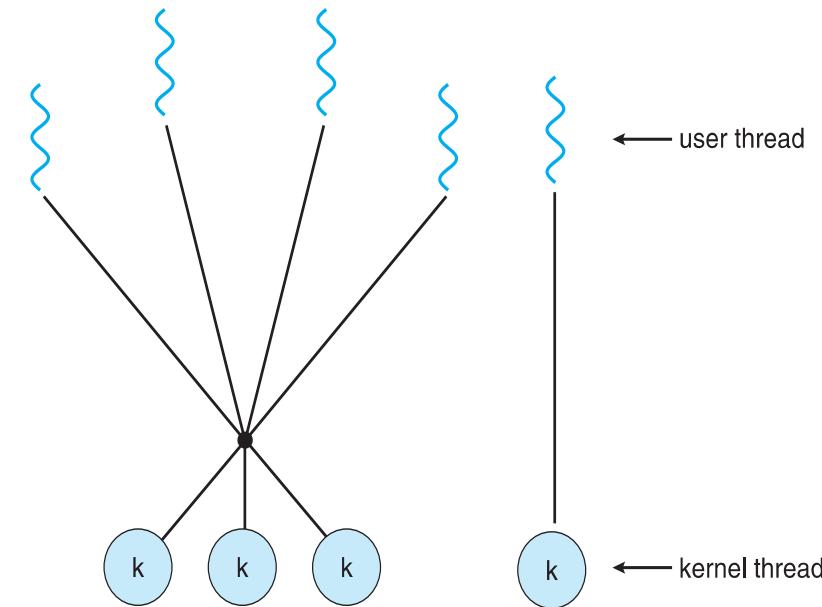
Many-to-Many Model

- ❖ Allows many user level threads to be mapped to many kernel threads
- ❖ Allows the operating system to create a sufficient number of kernel threads
- ❖ When a thread performs a blocking system call, the kernel can schedule another thread for execution
- ❖ Solaris prior to version 9



Two-Level Model

- ❖ Similar to M:M (ie,many to many), except that it allows a user thread to be **bound** to kernel thread
- ❖ Combination of m:m and one to one
- ❖ Examples
 - ❖ IRIX
 - ❖ HP-UX
 - ❖ Tru64 UNIX
 - ❖ Solaris 8 and earlier



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 (Solaris, Linux, Mac OS X)

PThreads

- ❖ `int pthread_attr_init(pthread_attr_t *attr);`
- ❖ `int pthread_create(pthread_t *tid, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);`
- ❖ `int pthread_equal(pthread_t tid1, pthread_t tid2); //0->unequal ids`
- ❖ `int pthread_join(pthread_t tid, void **retval);`
- ❖ `void pthread_exit(void *ptr);`

Pthreads Example

```
#include <pthread.h>
#include <stdio.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 */
    if (argc != 2) {
        fprintf(stderr,"usage: a.out <integer value>\n");
        return -1; }
    if (atoi(argv[1]) < 0) {
        fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
        return -1; }
    pthread_attr_init(&attr); /* set the default attributes */
    pthread_create(&tid, &attr, runner, argv[1]); /* create the thread */
    pthread_join(tid, NULL); /* wait for the thread to exit */
    printf("sum = %d\n", sum);
}
```

```
/* The thread will begin control 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);
}
```

synchronous threading

Pthreads Example

```
#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);

JOINING 10 THREADS
```

Threading Issues

- ❖ fork() and exec() system calls
- ❖ Signal handling
- ❖ Thread cancellation of target thread
- ❖ Thread-local storage

fork() and exec()

- ❖ If one thread in a program calls fork(), does the new process duplicate all threads, or is the new process single-threaded?
 - ❖ Some UNIX systems have chosen to have two versions of fork(), one that duplicates all threads and another that duplicates only the thread that invoked the fork() system call
 - ❖ The exec() system call works in the same way
 - ❖ if a thread invokes the exec() system call, the program specified in the parameter to exec() will replace the entire process—including all threads
 - ❖ If exec() is called immediately after forking, then duplicating all threads is unnecessary, as the program specified in the parameters to exec() will replace the process
 - ❖ duplicating only the calling thread is appropriate
 - ❖ If the separate process does not call exec() after forking, the separate process should duplicate all threads
-

Signal Handling

- ❖ Signals are used in UNIX systems to notify a process that a particular event has occurred
- ❖ The signal is delivered to a process
- ❖ When delivered, signal handler is used to process signals
- ❖ ***Synchronous and asynchronous signals***
- ❖ ***Synchronous signals***
 - ❖ illegal memory access, div. by 0
 - ❖ delivered to the same process that performed the operation generating the signal
- ❖ ***Asynchronous signals***
 - ❖ generated by an event external to a running process
 - ❖ the running process receives the signal asynchronously
 - ❖ Ctrl + C, timer expiration

Signal Handling

- ❖ Signal is handled by one of two signal handlers:
 - ❖ default
 - ❖ user-defined
- ❖ Every signal has default handler that kernel runs when handling signal
- ❖ User-defined signal handler can override default signal handler
- ❖ Some signals can be ignored, others are handled by terminating the process
- ❖ For single-threaded, signal is delivered to process

Signal Handling

❖ *Where should a signal be delivered for multi-threaded process?*

- ❖ 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
- ❖ Method for delivering a signal depends on the type of signal generated
 - ❖ synchronous signals need to be delivered to the thread causing the signal and not to other threads in the process
 - ❖ some asynchronous signals—such as <Ctrl + C> should be sent to all threads
- ❖ Most multithreaded versions of UNIX allow a thread to specify which signals it will accept and which it will block
- ❖ In some cases, an asynchronous signal may be delivered only to those threads that are not blocking it

Thread Cancellation

- ❖ Terminating a thread before it has finished
- ❖ Thread to be canceled is **target thread**
- ❖ Two general approaches:
 - ❖ **Asynchronous cancellation** – one thread terminates the target thread immediately
 - ❖ **Deferred cancellation** allows the target thread to periodically check if it should be cancelled, target thread can terminate itself in orderly fashion
- ❖ ***What about freeing resources??***
- ❖ Pthread code to create and cancel a thread:

```
pthread_t tid;
/* create the thread */
pthread_create(&tid, &attr, worker, NULL);
...
/* cancel the thread */
pthread_cancel(tid);
```

Thread Cancellation

- ❖ Invoking thread cancellation requests cancellation, but actual cancellation depends on how the target thread is set up to handle the request

default type



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**
 - ❖ Establish cancellation point by calling **pthread_testcancel()**
 - ❖ If cancellation request is pending, **cleanup handler** is invoked to release any acquired resources

Thread-Local Storage

- ❖ **Thread-local storage (TLS)** allows each thread to have its own copy of data
- ❖ TLS data are unique to each thread
- ❖ Different from local variables
 - ❖ Local variables visible only during single function invocation
 - ❖ TLS visible across function invocations



Thank You



BITS Pilani
Hyderabad Campus

OPERATING SYSTEMS (CS F372)

CPU Scheduling

Dr. Barsha Mitra
CSIS Dept., BITS Pilani, Hyderabad Campus



Basics

- Maximum CPU utilization obtained with multiprogramming
- **CPU–I/O Burst Cycle** – Process execution consists of a **cycle** of CPU execution and I/O wait
- **CPU burst** followed by **I/O burst**
- More number of short CPU bursts and less number of long CPU bursts

CPU Scheduler

- ❖ **Short-term / CPU scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
- ❖ CPU scheduling decisions may take place when a process:
 1. Switches from running to waiting state
 2. Switches from running to ready state
 3. Switches from waiting to ready
 4. Terminates
- ❖ **Preemptive scheduling** – done in situations 2 and 3, preemption means pausing something forcefully
- ❖ **Nonpreemptive /Cooperative scheduling** – once a process is allocated the CPU it retains the CPU until termination or switching to waiting state

Dispatcher

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

Scheduling Criteria

- ❖ **CPU utilization** – keep the CPU as busy as possible
 - ❖ **Throughput** – no. of processes that complete their execution per time unit
 - ❖ **Turnaround time** – amount of time to execute a particular process, interval from submission time to completion time, sum of durations spent waiting to get into memory, waiting in ready queue, executing on CPU, doing I/O
 - ❖ **Waiting time** – amount of time a process has been waiting in the ready queue
 - ❖ **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment), depends on the speed of output device
-

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



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

❖ Arrival time is 0

❖ *GANTT CHART*



❖ Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$

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

First- Come, First-Served (FCFS) Scheduling



- ❖ Suppose that the processes arrive in the order: P_2, P_3, P_1
- ❖ The Gantt chart for the schedule is:



- ❖ Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- ❖ Average waiting time: $(6 + 0 + 3)/3 = 3$
- ❖ Much better than previous case
- ❖ **Non-preemptive**
- ❖ Not applicable for time sharing systems

FCFS: Example



- Draw Gantt chart
- Compute the average wait time, TAT and RT for processes
- Note: if all arrive at same time, schedule in ascending order of process_id; waiting time = tat - burst time
- TAT = Finish time - arrival time
- Response time = assume = waiting time (when first output received wrt arrival time)

Process	AT	BT	FT	TAT	WT	RT
P1	0	7	7	7	0	0
P2	0	3	10	10	7	7
P3	0	4	14	14	10	10
P4	0	6	20	20	14	14

FCFS: Example



- Draw Gantt chart
- Compute the average wait time, TAT and RT for processes

Process	AT	BT	FT	TAT	WT	RT
P1	0	7	7	7	0	0
P2	8	3	20	12	17-8	9
P3	3	4	11	8	7-3	4
P4	5	6	17	12	11-5	6



FCFS: Example

- Draw Gantt chart
- Compute the average wait time, TAT and RT for processes

Process	AT	BT	FT	TAT	WT	RT
P1	0	2	2	2	0	0
P2	3	1	4	1	0	0
P3	5	5	10	5	0	0

Shortest-Job-First (SJF) Scheduling

- ❖ Associate with each process the length of its next CPU burst
 - ❖ Use these lengths to schedule the process with the shortest time
 - ❖ Use FCFS in case of tie
- ❖ SJF is optimal – gives minimum average waiting time for a given set of processes
 - ❖ The difficulty is knowing the length of the next CPU request
 - ❖ For long-term (job) scheduling in a batch system, use the process time limit that a user specifies when the job is submitted

Determining Length of Next CPU Burst



- ❖ Not possible to implement for short-term scheduling
- ❖ 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:

t_n = actual length of n^{th} CPU burst

τ_{n+1} = predicted value of the next CPU burst,

$0 \leq \alpha \leq 1$, τ_0 = constant or overall system average

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

Prediction of Length of Next CPU Burst



❖ $\alpha = 0$

❖ $\tau_{n+1} = \tau_n$

❖ $\alpha = 1$

❖ $\tau_{n+1} = t_n$

❖ If we expand the formula, we get:

$$\tau_{n+1} = \alpha t_n + (1 - \alpha) \alpha t_{n-1} + \dots + (1 - \alpha)^j \alpha t_{n-j} + \dots + (1 - \alpha)^{n+1} \tau_0$$

❖ τ_0 – constant or system average

❖ Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

❖ Commonly, α set to $\frac{1}{2}$

Prediction of Length of Next CPU Burst

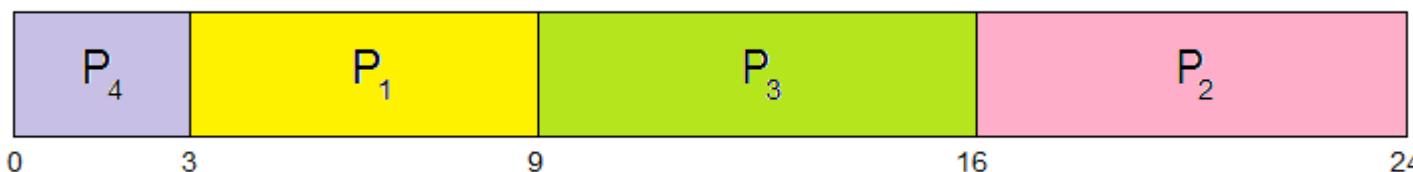


- ❖ Can be nonpreemptive or preemptive
- ❖ The next CPU burst of a newly arrived process may be shorter than what is left of the currently executing process
- ❖ Preemptive version called **shortest-remaining-time-first**

Shortest-Job-First (SJF) Scheduling

Process Burst Time

P_1	6
P_2	8
P_3	7
P_4	3

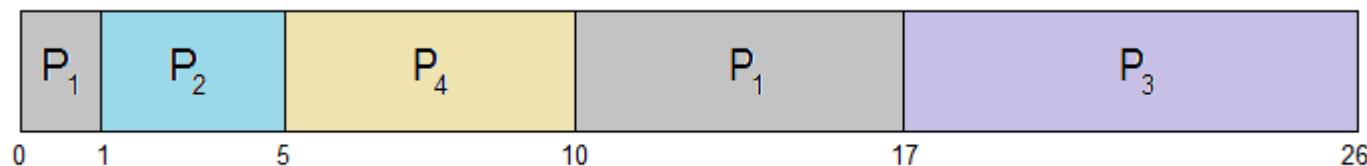


- Assuming same arrival time for all
- ❖ Average waiting time = $(3 + 16 + 9 + 0) / 4 = 7 \text{ ms}$

Shortest-remaining-time-first

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

❖ *Preemptive SJF Gantt Chart*



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

SJF (non-preemptive): Example



- Draw Gantt chart
- Compute the average wait time, TAT and RT for processes

Process	AT	BT	FT	TAT	WT	RT
P1	0	7				
P2	0	3				
P3	0	4				
P4	0	6				

SJF (non-preemptive): Example



- Draw Gantt chart
- Compute the average wait time, TAT and RT for processes

Process	AT	BT	FT	TAT	WT	RT
P1	0	7				
P2	8	3				
P3	3	4				
P4	5	6				



SJF (Preemptive) / SRTF: Example



- Draw Gantt chart
- Compute the average wait time, TAT and RT for processes

Process	AT	BT	FT	TAT	WT	RT
P1	0	7				
P2	8	3				
P3	3	2				
P4	5	6				

Priority Scheduling

- ❖ A priority number (integer) is associated with each process, generally starting from 0
- ❖ The CPU is allocated to the process with the highest priority (smallest integer \equiv highest priority), tie broken using FCFS
 - ❖ Preemptive
 - ❖ Nonpreemptive
- ❖ SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- ❖ Problem \equiv **Starvation/Indefinite blocking** – low priority processes may never execute
- ❖ Solution \equiv **Aging** – as time progresses increase the priority of the process

Nonpreemptive Priority Scheduling

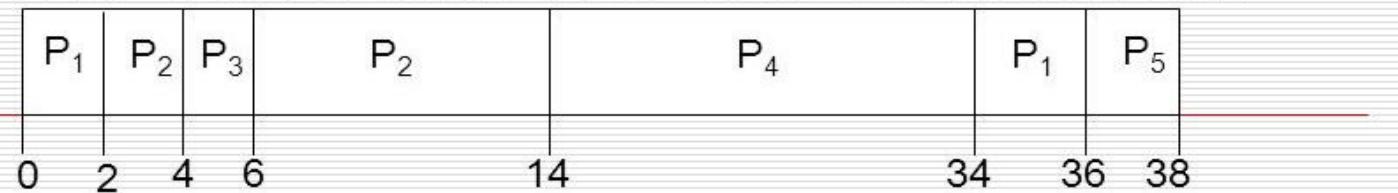
<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- ❖ Arrival time =0 for all
- ❖ Average waiting time = 8.2 msec

Preemptive Priority Scheduling

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>	<u>Priority</u>
P_1	0	4	4
P_2	2	10	2
P_3	4	2	1
P_4	6	20	3
P_5	8	2	5

The Gantt chart for the schedule is:

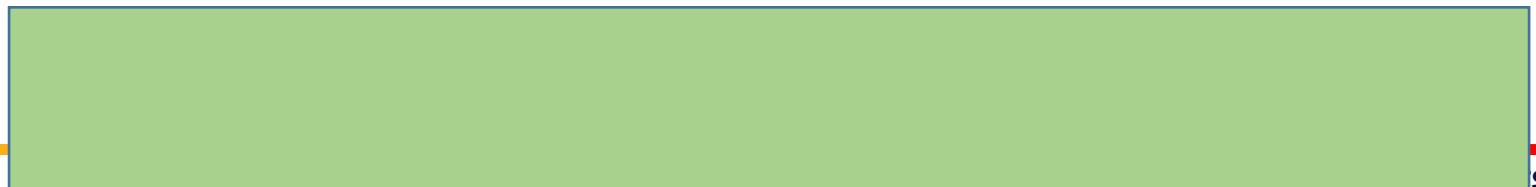


Priority (non-preemptive): Example



- Draw Gantt chart (Lower Number Higher priority,)
- Compute the average wait time, TAT and RT for processes

Process	AT	BT	Pri	FT	TAT	WT	RT
P1	0	3	5				
P2	2	2	3				
P3	3	5	2				
P4	4	4	4				
P5	6	1	1				



Priority (preemptive): Example



- Draw Gantt chart
- Compute the average wait time, TAT and RT for processes

Process	AT	BT	Pri	FT	TAT	WT	RT
P1	0	3	5				
P2	2	2	3				
P3	3	5	2				
P4	4	4	4				
P5	6	1	1				

Round Robin (RR) Scheduling

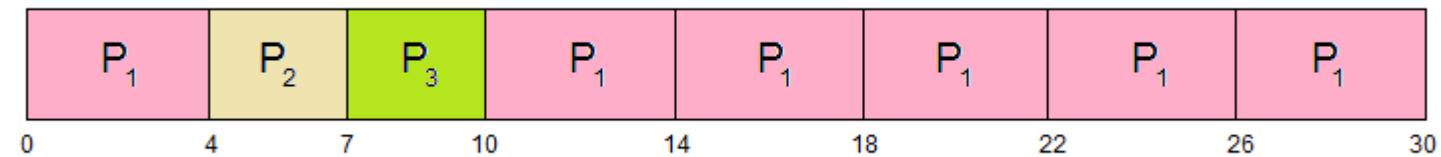
- ❖ Each process gets a small unit of CPU time (**time quantum** q), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- ❖ If there are n processes in the ready queue and the time quantum is q , then each process gets $1/n$ of the CPU time in chunks of at most q time units at once. No process waits more than $(n-1)q$ time units until its next time quantum.
- ❖ If only one process is present in ready queue, preemption will happen but context switching would not happen
- ❖ Timer interrupts every quantum to schedule next process
- ❖ Performance
 - ❖ q large \Rightarrow FCFS
 - ❖ q small \Rightarrow q must be large with respect to context switch, otherwise overhead is too high

Round Robin (RR) Scheduling



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

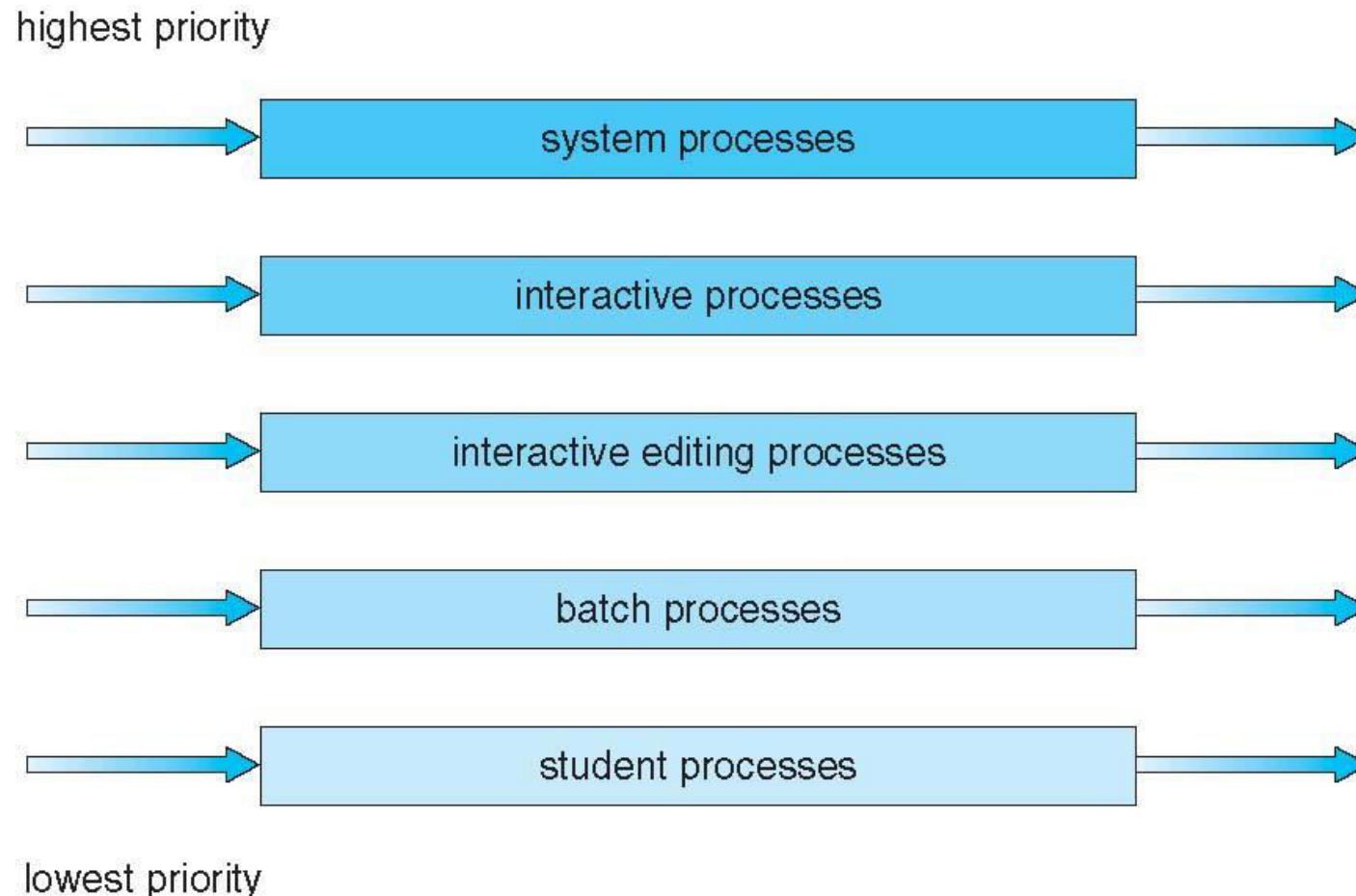
- The Gantt chart is:



Typically, higher average turnaround than SJF, but better **response**

- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch < 10 usec

Multilevel Queue Scheduling



fixed priority
preemptive
scheduling among
queues

Note: only when queue1 is
empty will lower queues be
executed,

process arriving in
q0 can preemp
process running in
lower queues

Multilevel Feedback Queue

- ❖ A process can move between the various queues; aging can be implemented this way
- ❖ Separate processes according to the characteristics of their CPU bursts
- ❖ Multilevel-feedback-queue scheduler is 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

Multilevel Feedback Queue

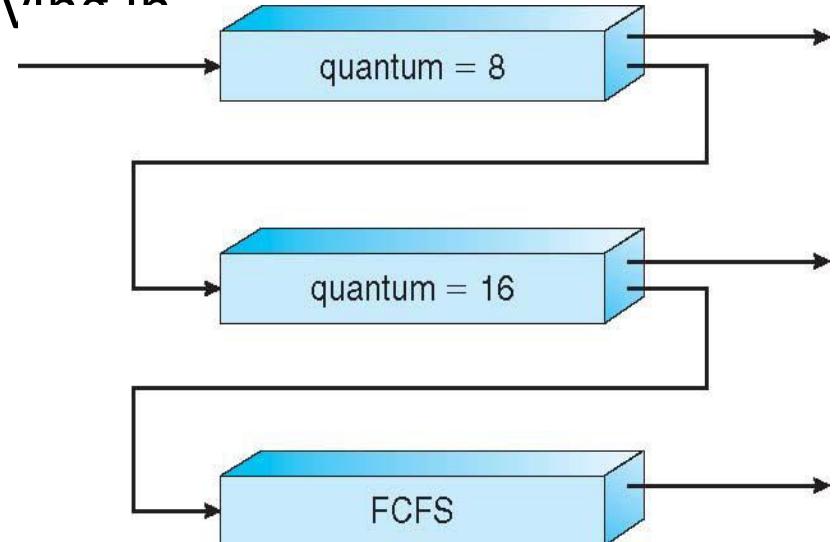
- ❖ Note: only when queue1 is empty will lower queues be executed, process arriving in Q_0 can preempt process running in lower queues

- ❖ 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 uses RR
 - ❖ When it gains CPU, job receives 8 milliseconds
 - ❖ If it does not finish in 8 milliseconds, job is moved to queue tail of Q_1
- ❖ At Q_1 process is again served using RR and receives 16 additional milliseconds
 - ❖ If it still does not complete, it is preempted and moved to tail of queue Q_2



FCFS with I/O



Process	AT	BT & I/O	FT	TAT	WT	RT
P1	0	<u>3</u> , 5, <u>2</u>				
P2	2	<u>4</u> , 3, <u>3</u>				
P3	4	<u>4</u>				
P4	6	<u>3</u> , 2, <u>3</u>				



SJF (Non Pre-emptive) with I/O

Process	AT	CPU- BT & I/O BT	Total BT	FT	TAT	WT	RT
P1	0	(<u>4</u> – 2 – <u>4</u>)					
P2	0	(<u>5</u> – 2 – <u>5</u>)					
P3	0	(<u>2</u> – 2 - <u>2</u>)					



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
 - ❖ Scheme is known as **process-contention scope (PCS)** since scheduling competition is within the process (local contention scope)
 - ❖ Typically done via priority set by programmer
- ❖ Kernel thread scheduled onto available CPU is **system-contention scope (SCS)** – competition among all threads in system (global contention scope)

Pthread Scheduling

- ❖ API allows specifying either PCS or SCS during thread creation, contention scope values
 - ❖ PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling
 - ❖ PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling
- ❖ Can be limited by OS – Linux and Mac OS X only allow PTHREAD_SCOPE_SYSTEM
- On systems implementing the M:M model, PTHREAD_SCOPE_PROCESS policy schedules user-level threads onto available LWPs
- PTHREAD_SCOPE_SYSTEM policy will create and bind an LWP for each user-level thread effectively mapping a user-level thread to a kernel level thread

Pthread Scheduling

- ❖ `pthread_attr_setscope(pthread_attr_t *attr, int scope)`
- ❖ `pthread_attr_getscope(pthread_attr_t *attr, int *scope)`
- ❖ On success, these functions return 0; on error, they return a nonzero error number

Pthread Scheduling

```
#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, "Error\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");
    }
}
```

```
/* 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);
}
```

CFS

- default scheduler of Linux kernel
- red-black tree
- processes are given a fair amount of processor time
- keeps track of the amount of processor time provided to a process – virtual runtime
- smaller a process's virtual runtime implies the process has been permitted to access processor for a smaller amount of time, hence higher is the need for processor



Thank You



BITS Pilani
Hyderabad Campus

OPERATING SYSTEMS (CS F372)

Synchronization

Dr. Barsha Mitra
CSIS Dept., BITS Pilani, Hyderabad Campus



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
 - ❖ Consider the Producer-Consumer problem
-

Producer - Consumer

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

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 counter = 5 initially:

S0	producer execute register1 = counter	register1 = 5
S1	producer execute register1 = register1 + 1	register1 = 6
S2	consumer execute register2 = counter	register2 = 5
S3	consumer execute register2 = register2 - 1	register2 = 4
S4	producer execute counter = register1	counter = 6
S5	consumer execute counter = register2	counter = 4

Critical Section Problem

- ❖ Consider system of n processes $\{p_0, p_1, \dots, p_{n-1}\}$
- ❖ Each process has **critical section** segment of code
 - ❖ Process may be changing common variables, updating table, writing file, etc.
 - ❖ When one process in critical section, no other may be in its critical section
- ❖ ***Critical section problem*** is to design protocol to solve this
- ❖ Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

```
do {  
    entry section  
    critical section  
    exit section  
    remainder section  
} while (true);
```

Requirements

-
1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
 2. **Progress** - If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder section can participate in deciding which will enter its critical section next, and this decision ***cannot be postponed indefinitely***
 3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

Critical-Section Handling in OS

-
- ❖ Two approaches depending on if kernel is preemptive or non-preemptive
 - ❖ **Preemptive** – allows preemption of process when running in kernel mode
 - ❖ **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
 - ❖ Essentially free of race conditions in kernel mode
 - ❖ *Why then anyone would prefer preemptive kernel?????*

Peterson's Solution

- ❖ Two process software based solution
- ❖ Assume that the `load` and `store` machine-language instructions are atomic; that is, cannot be interrupted
- ❖ The two processes share two variables:

```
int turn;
boolean flag[2];
```
- ❖ 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

Algorithm for Process P_i

```
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
        critical section  
    flag[i] = false;  
    remainder section  
} while (true);
```

Provable that the 3 CS requirements are met:

1. Mutual exclusion is preserved
2. Progress requirement is satisfied
3. Bounded-waiting requirement is met

Check 3 Requirements

```
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
        critical section  
    flag[i] = false;  
        remainder section  
} while (true);
```

P_i

```
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
        critical section  
    flag[j] = false;  
        remainder section  
} while (true);
```

P_j

Synchronization Hardware

- ❖ Many systems provide hardware support for implementing the critical section code
- ❖ All solutions based on idea of **locking**
 - ❖ Protecting critical regions via locks
- ❖ Uniprocessors – could disable interrupts
 - ❖ Currently running code would execute without preemption
 - ❖ Generally too inefficient on multiprocessor systems
- ❖ Modern machines provide special atomic hardware instructions
 - ❖ **Atomic** = non-interruptible
 - ❖ Either test memory word and set value
 - ❖ Or swap contents of two memory words

Solution to Critical-section Problem Using Locks



```
do {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
} while (TRUE);
```

test_and_set Instruction

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

1. Executed atomically
2. Returns the original value of passed parameter
3. Sets the new value of passed parameter to true

Solution using `test_and_set` Instruction



Shared Boolean variable `lock`, initialized to false, supports mutual exclusion and progress but not bounded waiting

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

compare_and_swap Instruction

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

1. Executed atomically
2. Returns the original value of passed parameter **value**
3. Sets the variable **value** the value of the passed parameter **new_value** but only if **value == expected**. That is, the swap takes place only under this condition.

Solution using compare_and_swap

- ❖ Shared (global) integer **lock** initialized to 0;
- ❖ Solution:

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

Bounded-waiting Mutual Exclusion with test_and_set



common data structures – boolean waiting[n] , boolean lock, all initialized to false

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

Mutex Locks

- ❖ Hardware solutions are complicated and generally inaccessible to application programmers
- ❖ OS designers build software tools to solve critical section problem
- ❖ Simplest is **mutex** lock
- ❖ Protect a critical section by first `acquire()` a lock then `release()` the lock
 - ❖ mutex lock has a boolean variable **available** indicating if lock is available or not
- ❖ Calls to `acquire()` and `release()` must be atomic
 - ❖ Usually implemented via hardware atomic instructions
- ❖ But this solution requires **busy waiting**
 - ❖ This lock therefore called a **spinlock**

acquire() and release()

```
acquire() {  
    while (!available)  
        ; /* busy wait */  
    available = false;;  
}  
release() {  
    available = true;  
}  
  
do {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
} while (true);
```

Adv. of Spinlock:

1. no context switch is required when a process must wait on a lock
2. useful when locks are to be held for short times

Disadv. of Spinlock: busy waiting wastes CPU cycles

Busy waiting- process although is waiting, but still is present in running state

Semaphore

- ❖ Provides more sophisticated ways (than mutex locks) for process to synchronize
- ❖ Semaphore **S** – integer variable
- ❖ Apart from initialization, can only be accessed via two indivisible (atomic) operations, **wait()** and **signal()**, Originally called **P()** and **V()**
- ❖ Definition of the **wait()** operation

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

- ❖ Definition of the **signal()** operation

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

Semaphore Usage

- ❖ **Counting semaphore** – integer value can range over an unrestricted domain
 - ex- when multiple instance of same resource present
- ❖ **Binary semaphore** – integer value can range only between 0 and 1
- ❖ Must guarantee that no two processes can execute the **wait()** and **signal()** on the same semaphore at the same time
 - ex- when single instance of a resource present

Semaphore Implementation



- ❖ With each semaphore there is an associated waiting queue
- ❖ Each semaphore has:
 - ❖ value (of type integer)
 - ❖ a list of processes
- ❖ Two operations:
 - ❖ **block** – place the process invoking the operation on the appropriate waiting queue
 - ❖ **wakeup** – remove one of processes in the waiting queue and place it in the ready queue

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

Semaphore Implementation

```
wait(semaphore *S) {  
    S->value--;  
    if(S->value < 0) {  
        add this process to S->list;  
        block(); // to take this  
process from running to ready queue  
    }  
}
```

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

Deadlock and Starvation

- ❖ **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

P_0	P_1
<code>wait(S) ;</code>	<code>wait(Q) ;</code>
<code>wait(Q) ;</code>	<code>wait(S) ;</code>
...	...
<code>signal(S) ;</code>	<code>signal(Q) ;</code>
<code>signal(Q) ;</code>	<code>signal(S) ;</code>

- ❖ **Starvation – indefinite blocking** , A process may never be removed from the semaphore queue in which it is suspended

Classical Problems of Synchronization



- ❖ Bounded-Buffer Problem
- ❖ Readers and Writers Problem
- ❖ Dining-Philosophers Problem

Bounded-Buffer Problem

-
- ❖ ***Buffer of length n***, each location can hold one item (`int n`)
 - ❖ Semaphore **mutex** initialized to the value 1
 - ❖ Semaphore **full** initialized to the value 0
 - ❖ Semaphore **empty** initialized to the value *n*

Bounded-Buffer Problem

Structure of the producer process

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

Structure of the consumer process

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

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, no one else should be allowed whenever a writer is accessing critical section
- ❖ ***First readers-writers prob.*** → no reader is kept waiting unless a writer has already obtained the permission to use the shared obj., writers may starve

Readers-Writers Problem

❖ Shared Data

- ❖ Semaphore **rw_mutex** initialized to 1, common to both readers and writers, acts as a mutual exclusion semaphore for writers, used by first reader that enters CS or last reader that exits CS, not used by other readers
- ❖ Integer **read_count** initialized to 0, keeps track of how many processes are reading the shared obj.
- ❖ Semaphore **mutex** initialized to 1, ensures mutual exclusion when **read_count** is updated

Readers-Writers Problem



Structure of a writer process

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

Structure of a reader process

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

Dining-Philosophers Problem

- ❖ Philosophers spend their lives alternating b/w thinking and eating
- ❖ Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
 - ❖ Can pickup only one chopstick at a time
 - ❖ Need both to eat, then release both when done
 - ❖ Each chopstick is binary semaphore as they are not identical
- ❖ In the case of 5 philosophers
 - ❖ Shared data
 - ❖ Bowl of rice (data set)
 - ❖ Semaphore **chopstick [5]** initialized to 1



Dining-Philosophers Problem Algorithm



Structure of Philosopher *i*:

```
do {  
    wait (chopstick[i] );  
    wait (chopstick[ (i + 1) % 5] );  
  
        // eat  
  
    signal (chopstick[i] );  
    signal (chopstick[ (i + 1) % 5] );  
  
        // think  
  
} while (TRUE);
```

What is the problem with this algorithm?

Deadlock can occur

Dining-Philosophers Problem Algorithm



- ❖ Deadlock handling
 - ❖ Allow at most 4 philosophers to be sitting simultaneously at the table.
 - ❖ Allow a philosopher to pick up the chopsticks only if both are available.
 - ❖ Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.

Problems with Semaphores

- ❖ Incorrect use of semaphore operations:
 - ❖ signal (mutex) wait (mutex)
violate mutual exclusiveness
 - ❖ wait (mutex) ... wait (mutex)
leads to deadlock
 - ❖ Omitting of wait (mutex) or signal (mutex) (or both)
deadlock,starvation,mutual exclusiveness violated
- ❖ Deadlock and starvation are possible.



Thank You



BITS Pilani
Hyderabad Campus

OPERATING SYSTEMS (CS F372)

Deadlocks

Dr. Barsha Mitra
CSIS Dept., BITS Pilani, Hyderabad Campus



System Model

- ❖ System consists of resources
- ❖ Resource types R_1, R_2, \dots, R_m
- ❖ Each resource type R_i has W_i instances.
- ❖ Each process utilizes a resource as follows:
 - ❖ request
 - ❖ use
 - ❖ release

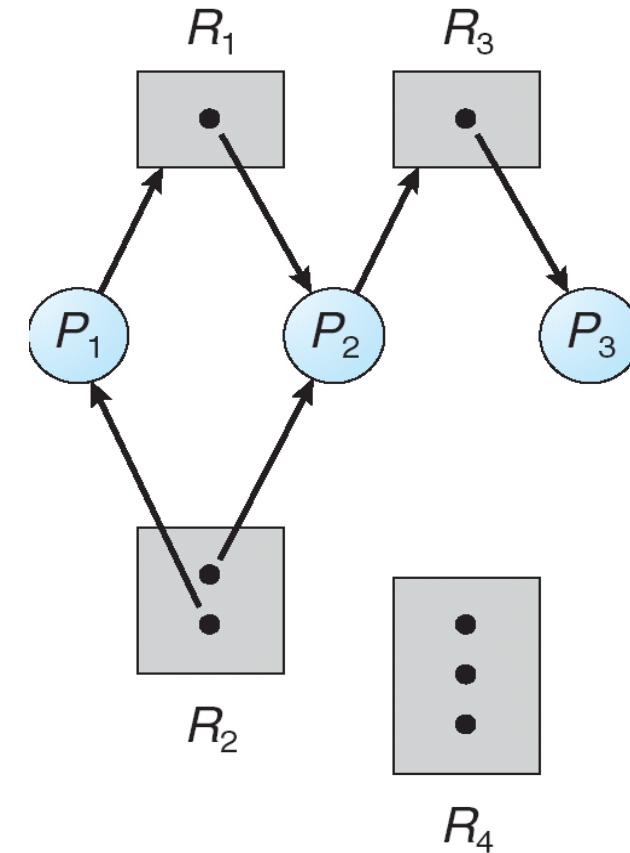
Necessary Conditions

Deadlock can arise if four conditions hold simultaneously

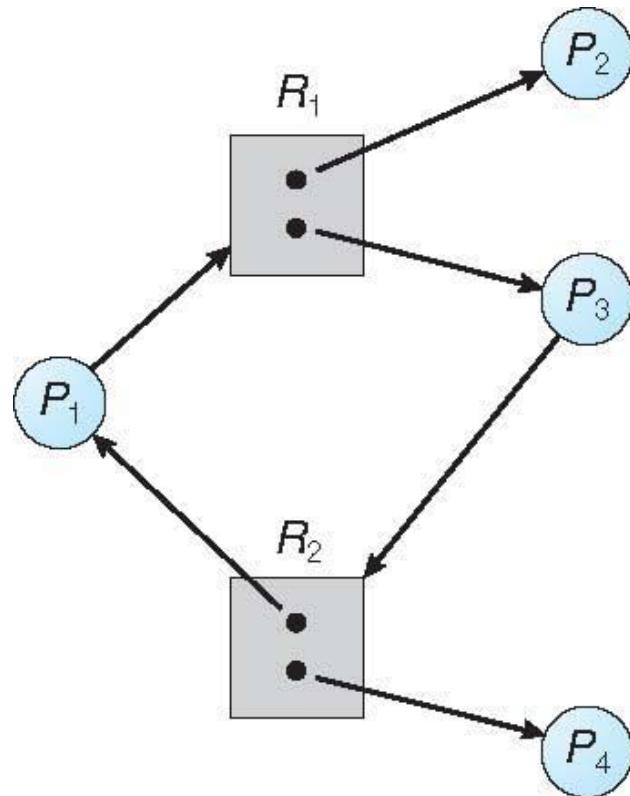
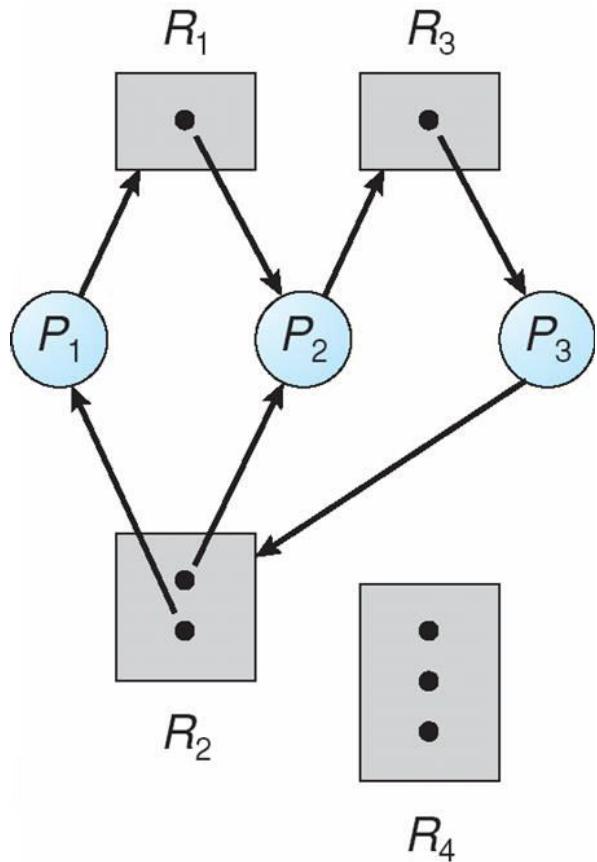
- ❖ **Mutual exclusion:** only one process at a time can use a resource
- ❖ **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
- ❖ **No preemption:** a resource can be released only voluntarily by the process holding it, after 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 .

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 $R_j \rightarrow P_i$



Examples



Inferring Deadlock

- ❖ If graph contains no cycles \Rightarrow no deadlock
- ❖ If graph contains a cycle \Rightarrow
 - ❖ if only one instance per resource type, then deadlock
 - ❖ if several instances per resource type, possibility of deadlock, look for knot
- ❖ What is KNOT??
- ❖ A collection of vertices and edges s.t. every vertex in the knot has outgoing edges that terminate at other vertices in the knot
- ❖ A strongly connected subgraph of a directed graph s.t. starting from any node in the subset it is impossible to leave the knot by following the edges of the graph

Deadlock Handling

- ❖ Ensure that the system will ***never*** enter a deadlock state:
 - ❖ Deadlock prevention
 - ❖ 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; used by most operating systems, including UNIX, application programmers should ensure that deadlocks don't occur

Deadlock Prevention

- ❖ **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
 - ❖ **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
 - ❖ Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none allocated to it
 - ❖ Low resource utilization; starvation possible
-

Deadlock Prevention

❖ No Preemption –

- ❖ If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held 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
- ❖ When a process P_1 requests some resources and they are allocated to some other waiting process P_2 , then preempt the desired resources from P_2 and give them to P_1
- ❖ If the resources are not allocated to a waiting process, then P_1 must wait
- ❖ While waiting P_1 's resources may be preempted

Deadlock Prevention

- ❖ **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration
 - ❖ define a 1:1 function $F : R \rightarrow N$ (N is set of natural nos.)
 - ❖ Say a process P_i has requested a no. of instances of R_i
 - ❖ Later, P_i can request resources of type R_j iff $F(R_j) > F(R_i)$
 - ❖ Alternatively, if P_i requests an instance of R_j then P_i must have released all instances of R_i s.t. $F(R_i) \geq F(R_j)$
 - ❖ Several instances of same resource type must be requested for in a single request
 - ❖ Proof by contradiction
-

Deadlock Avoidance

- Requires that the system has some additional *a priori* information available
- Requires that each process declare the ***maximum number*** of resources of each type that it may need
- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes
- Deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition

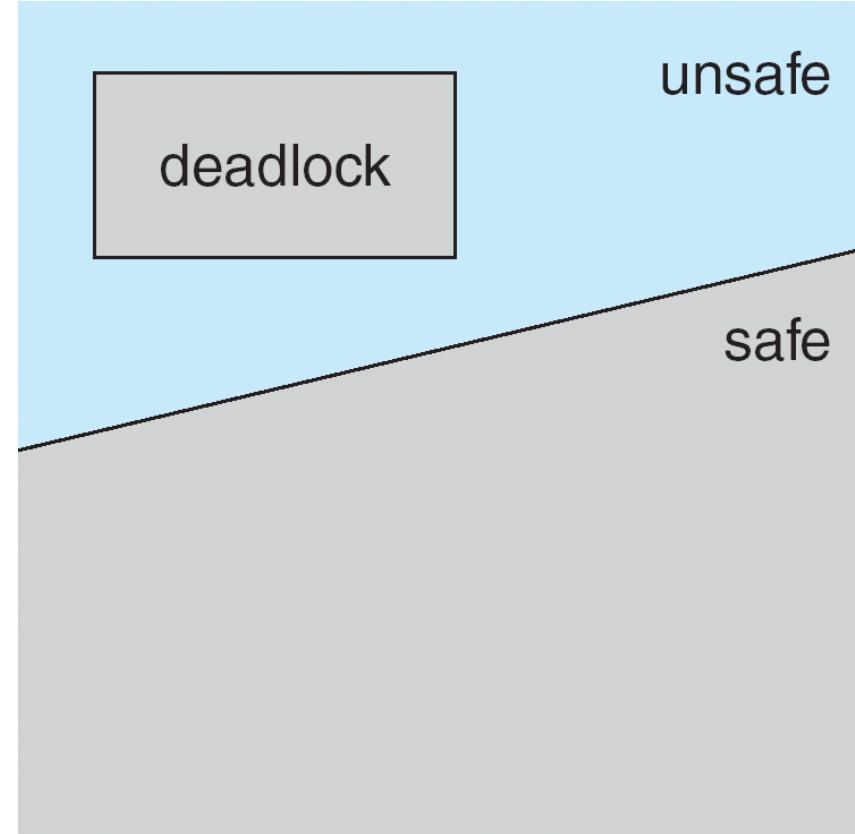
Safe State

- ❖ 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 $\langle P_1, P_2, \dots, P_n \rangle$ 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 $j < i$
- ❖ That is:
 - ❖ If resources needed by P_i 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

Inferences



- ❖ If a system is in safe state \Rightarrow no deadlocks
- ❖ If a system is in unsafe state \Rightarrow possibility of deadlock
- ❖ Avoidance \Rightarrow ensure that a system will never enter an unsafe state



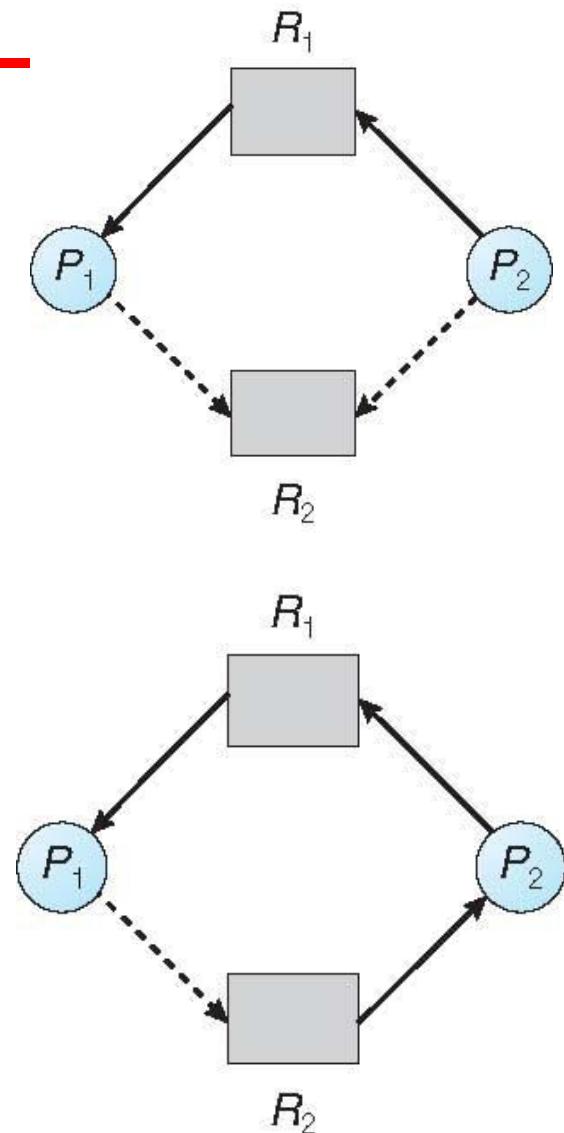
Avoidance Algorithms

- ❖ Single instance of a resource type
 - ❖ Use a resource-allocation graph
- ❖ Multiple instances of a resource type
 - ❖ Use the banker's algorithm

Resource-Allocation-Graph Algorithm



- ❖ **Claim edge** $P_i \rightarrow R_j$ indicated that process P_i may request resource R_j in future; represented by a dashed line
- ❖ Claim edge converts to request edge when a process requests a resource
- ❖ Request edge converted 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
- ❖ 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
- ❖ Each process must a priori declare 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 Banker's Algorithm



- ❖ 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 are available
- ❖ **Max:** $n \times m$ matrix. If 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 Allocation[i][j] = k then P_i is currently allocated k instances of R_j
- ❖ **Need:** $n \times m$ matrix. If Need[i][j] = k , then P_i may need k more instances of R_j to complete its task
$$\text{Need } [i][j] = \text{Max}[i][j] - \text{Allocation } [i][j]$$
- ❖ Each row of Max, Allocation and Need can be treated as vectors
- ❖ If X and Y are 2 vectors, then $X \leq Y$ iff, $X[i] \leq Y[i]$ for all $i = 1, 2, \dots, n$
- ❖ $X < Y$

Safety Algorithm

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

Initialize: **Work** = **Available**, **Finish** [i] = **false** for $i = 0, 1, \dots, n - 1$

2. Find an i such that both:

(a) **Finish** [i] == **false**

(b) **Need** _{i} \leq **Work**

If no such i exists, go to step 4

$O(mn^2)$ operations

3. **Work** = **Work** + **Allocation** _{i}

Finish[i] = **true**

go to step 2

4. If **Finish** [i] == **true** for all i , then the system is in a safe state

Banker's Algorithm Example

- ❖ 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 :

Process	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			

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

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

Resource-Request Algorithm

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

Banker's Algorithm Example:

P₁ Requests (1,0,2)

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

Process	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P ₀	0	1	0	7	5	3	2	3	0
P ₁	3	0	2	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			

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

Deadlock Detection

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

Single Instance of Each Resource Type



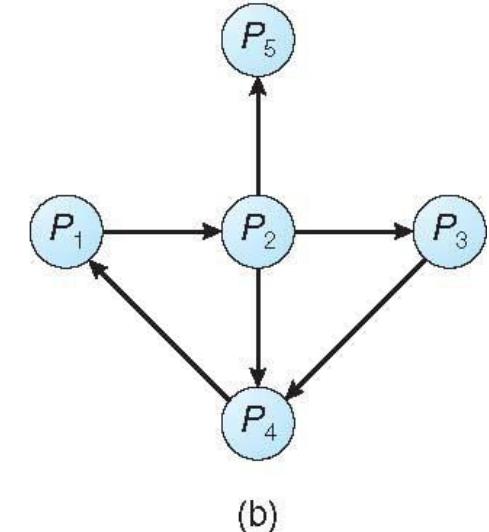
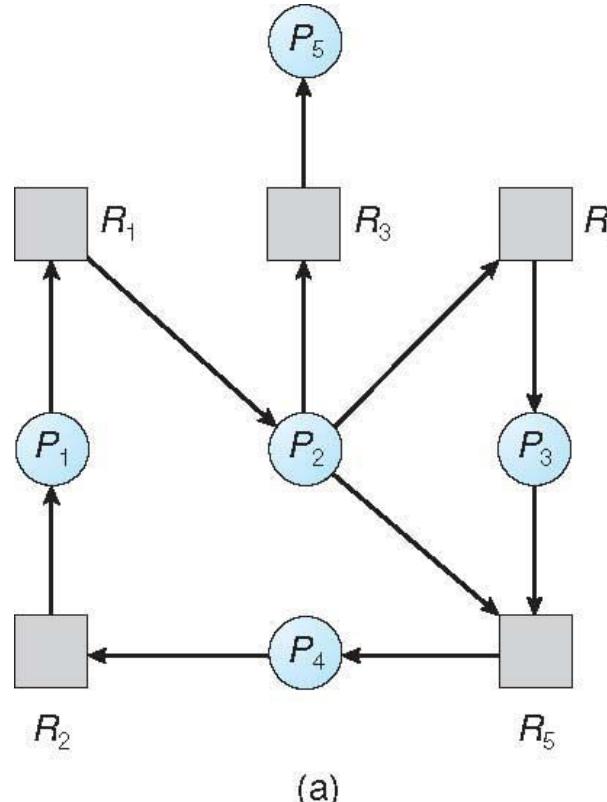
❖ Maintain **wait-for** graph

❖ Nodes are processes

❖ $P_i \rightarrow P_j$ if P_i is waiting for P_j

❖ 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 $O(n^2)$ operations, where n is the number of vertices in the graph



Recovery from Deadlock: Process Termination



- ❖ Abort all deadlocked processes
- ❖ Abort one process at a time until the deadlock cycle is eliminated
- ❖ 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



- ❖ Preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken
 - ❖ **Selecting a victim** – which resources and which processes, minimize cost by deciding order of preemption
 - ❖ **Rollback** – return to some safe state, restart process for that state, total rollback or partials
 - ❖ **Starvation** – same process may always be picked as victim, include number of rollback in cost factor
-



Thank You



BITS Pilani
Hyderabad Campus

OPERATING SYSTEMS (CS F372)

Main Memory Management

Dr. Barsha Mitra
CSIS Dept., BITS Pilani, Hyderabad Campus

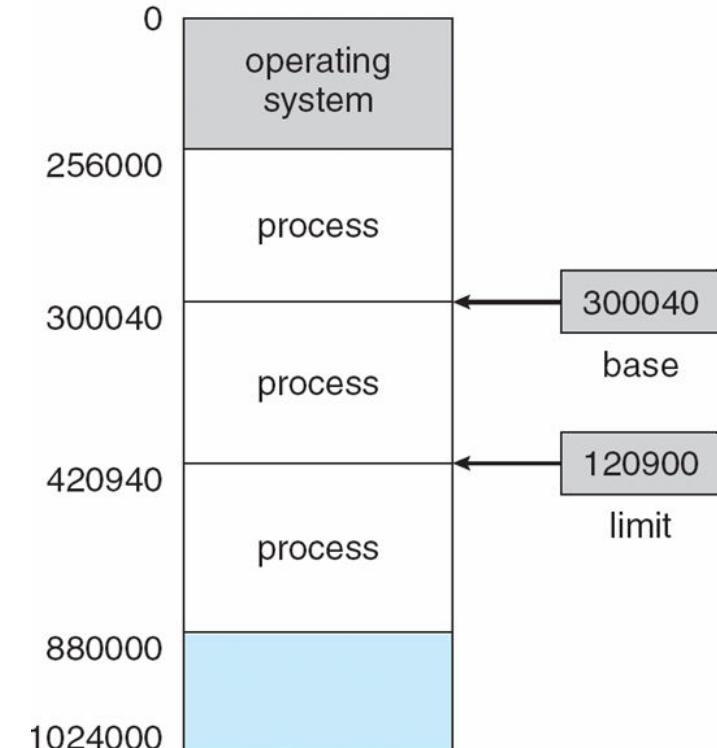


Background

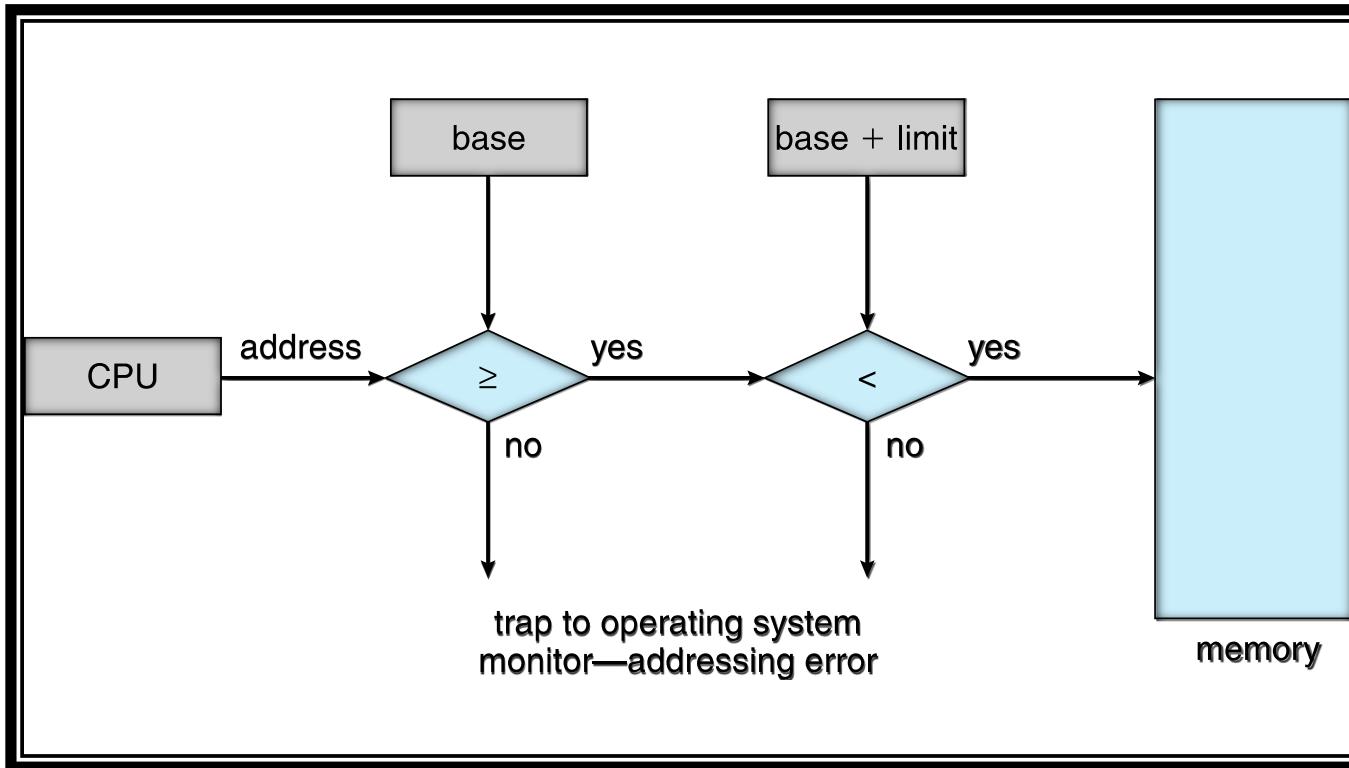
- ❖ Program must be brought (from disk) into memory
 - ❖ Main memory and registers are only storage CPU can access directly
 - ❖ Memory unit only sees a stream of addresses + read requests, or address + data and write requests
 - ❖ Register access in one CPU clock (or less)
 - ❖ Main memory can take many cycles, causing a **stall**
 - ❖ **Cache** sits between main memory and CPU registers
 - ❖ Protection of memory required to ensure correct operation
-

Base and Limit Registers

- ❖ A pair of **base** and **limit registers** define the address space
- ❖ CPU must check every memory access generated in user mode to be sure it is between base and limit for that user
- ❖ Limit will contain the size of range allowed to access by a process
- ❖ Highest address accessible by a process = base + limit - 1



Hardware Address Protection



Address Binding

- ❖ Programs on disk, ready to be brought into memory to execute
 - ❖ Further, addresses represented in different ways at different stages of a program's life
 - ❖ Source code addresses usually symbolic(logical/virtual addresses)
 - ❖ Compiled code addresses **bind** to **relocatable addresses**
 - ❖ like, “14 bytes from beginning of this module”
 - ❖ Linker or loader binds relocatable addresses to absolute addresses(physical addresses)
 - ❖ like, 74014
-

Logical vs. Physical Address Space

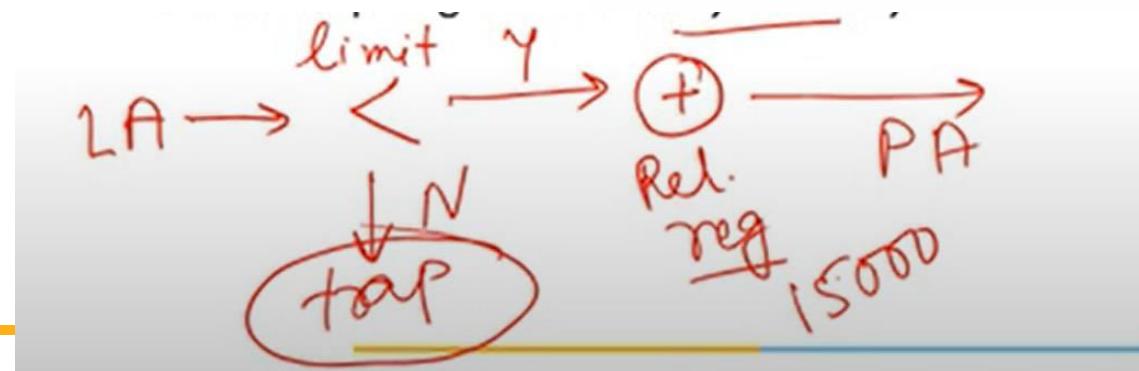
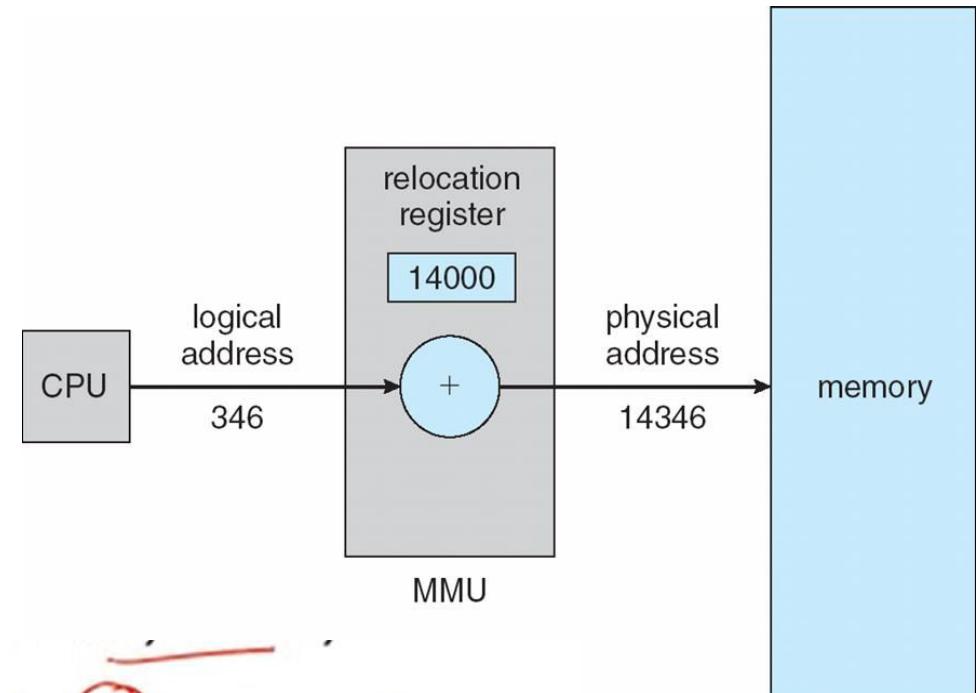
- ❖ **Logical address** – generated by the CPU; also referred to as **virtual address**
 - ❖ **Physical address** – address seen by the memory unit
 - ❖ **Logical address space** is the set of all logical addresses generated by a program
 - ❖ **Physical address space** is the set of all physical addresses corresponding to the logical addresses generated by a program
-

Memory-Management Unit (MMU)

- ❖ Hardware device that at run time maps virtual address to physical address
- ❖ Value in the relocation register is added to every address generated by a user process

❖ Relocation register

- ❖ The user program deals with *logical* addresses; it never sees the *real* physical addresses



Dynamic Loading

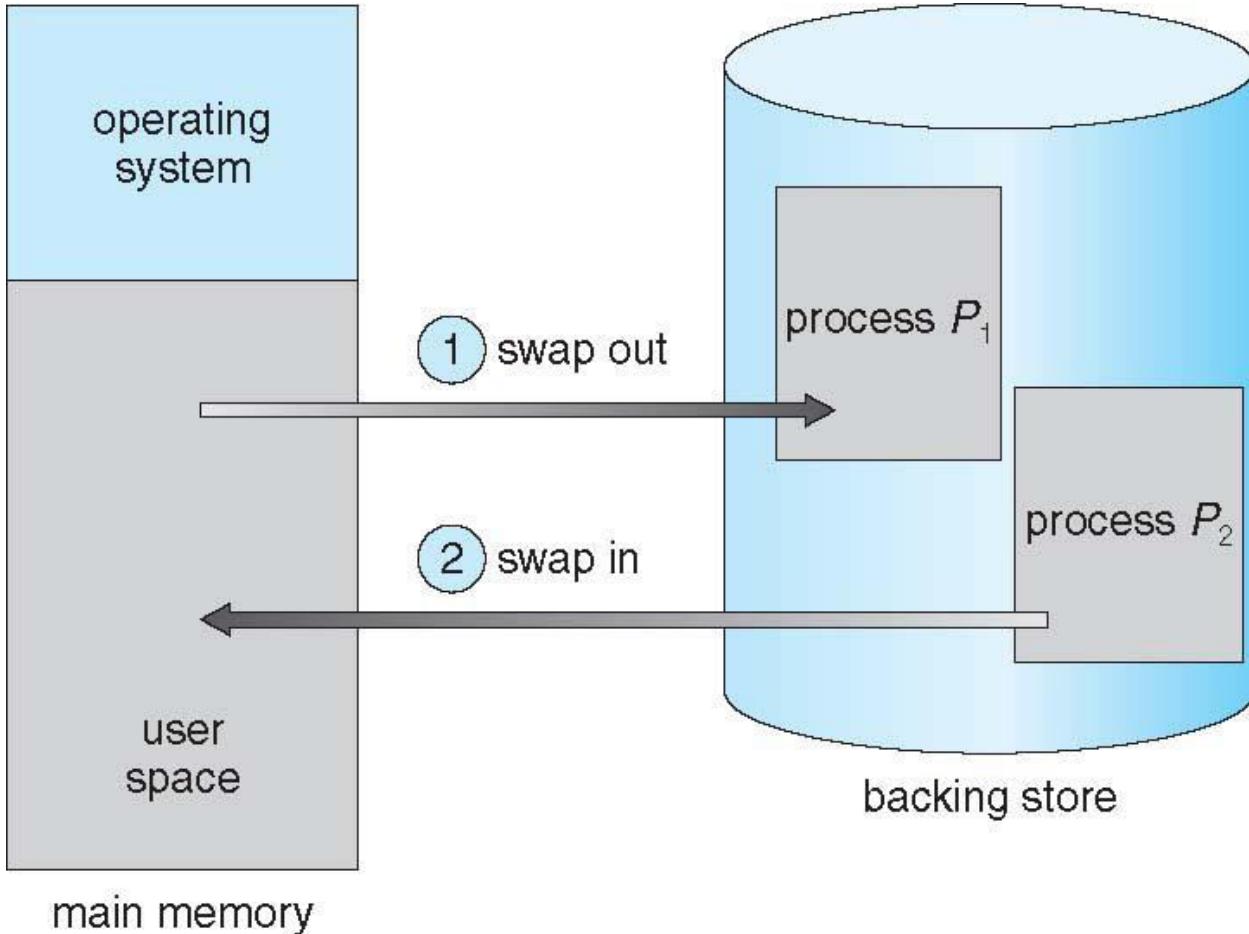
- ❖ Routine is not loaded until it is called
- ❖ Better memory-space utilization; unused routine is never loaded
- ❖ All routines kept on disk
- ❖ Useful when large amounts of code are needed to handle infrequently occurring cases

Swapping

- ❖ A process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution
 - ❖ Total physical memory space of processes can exceed physical memory, in such case swapping is done
- ❖ **Backing store** –(where the temporarily swapped out process are going to be kept) fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- ❖ Major part of swap time is transfer time
- ❖ System maintains a **ready queue** of ready-to-run processes which have memory images on disk
- ❖ Swap only when free memory extremely low
- ❖ Optimizing swapping by Swapping portions of processes

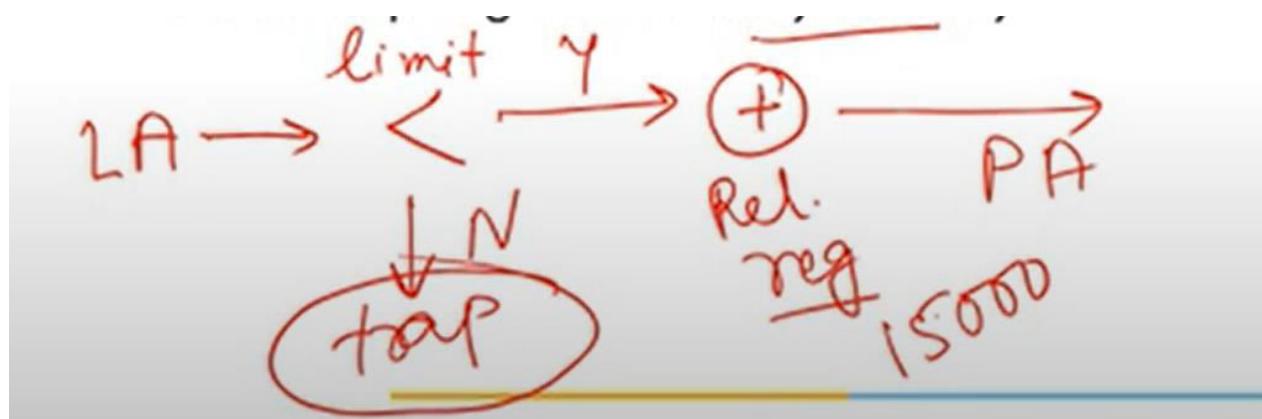
Swapping

- ❖ If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
- ❖ Context switch time can then be very high
- ❖ 100MB process swapping to hard disk with transfer rate of 50MB/sec
 - ❖ Swap out time of 2000 ms
 - ❖ Plus swap in of same sized process
 - ❖ Total context switch swapping component time of 4000ms (4 seconds)



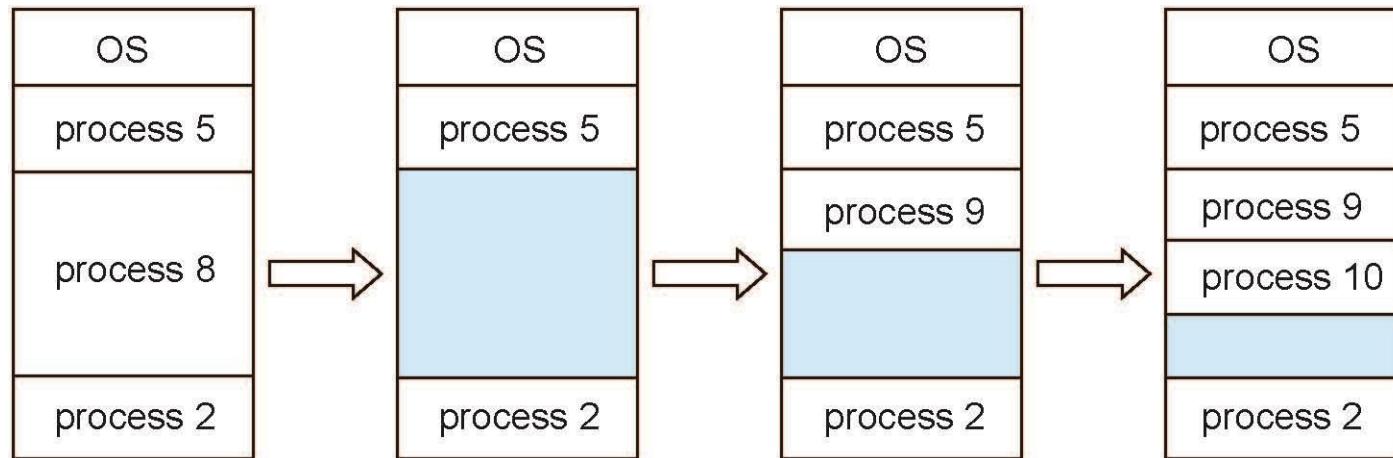
Contiguous Memory Allocation

- ❖ Relocation registers used to protect user processes from each other, and from changing operating-system 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*



Multiple-Partition Allocation

- Fixed size partitions, 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 (hole)



Dynamic Storage-Allocation Problem

How to satisfy a request of size n from a list of free holes?

- ❖ **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

First-fit and best-fit better than worst-fit in terms of speed and storage utilization

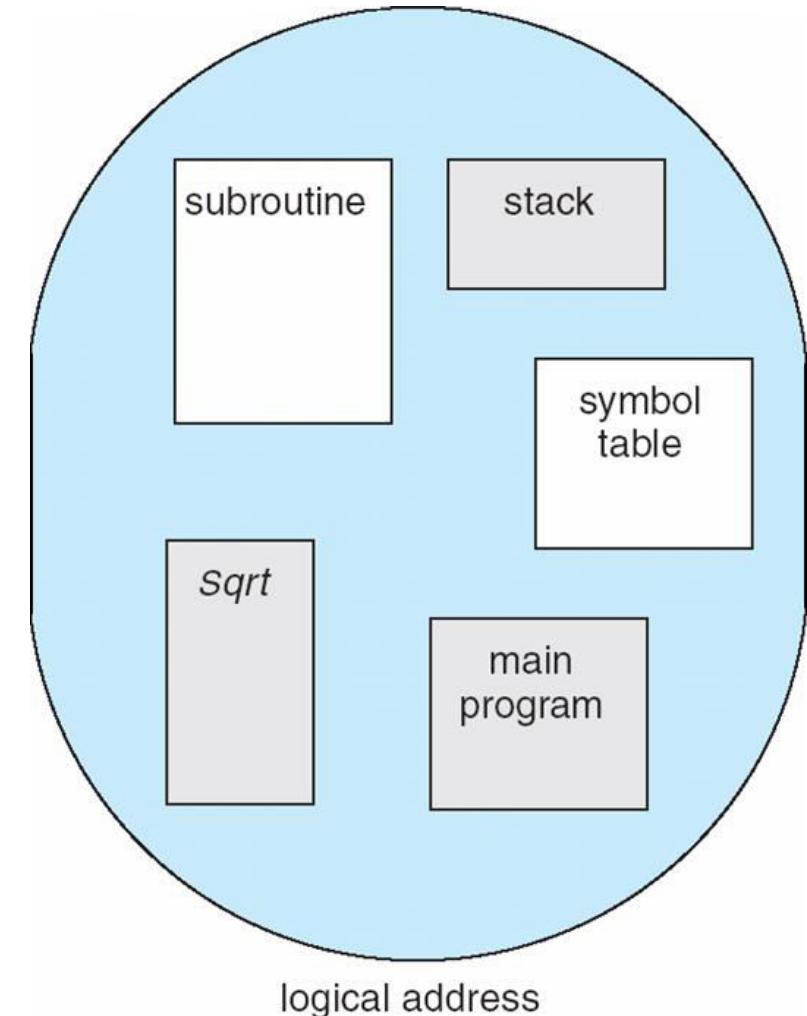
Fragmentation

- ❖ **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- ❖ Reduce external fragmentation by **compaction**
 - ❖ Shuffle memory contents to place all free memory together in one large block

But compaction is expensive, so other method used is : Allow logical address space of processes to be non-contiguous
- ❖ **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used

Segmentation (memory manag. Scheme)

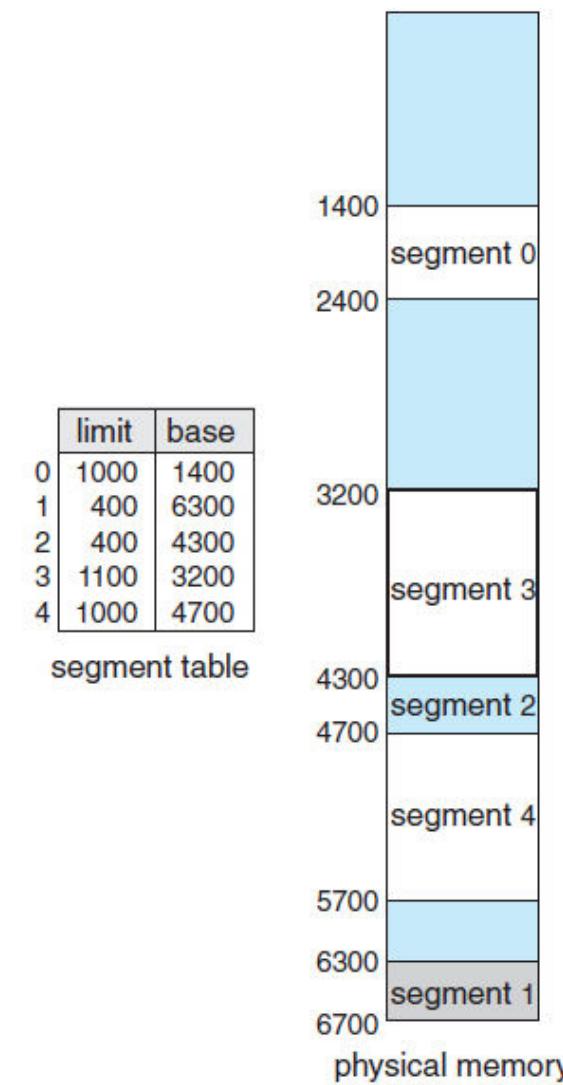
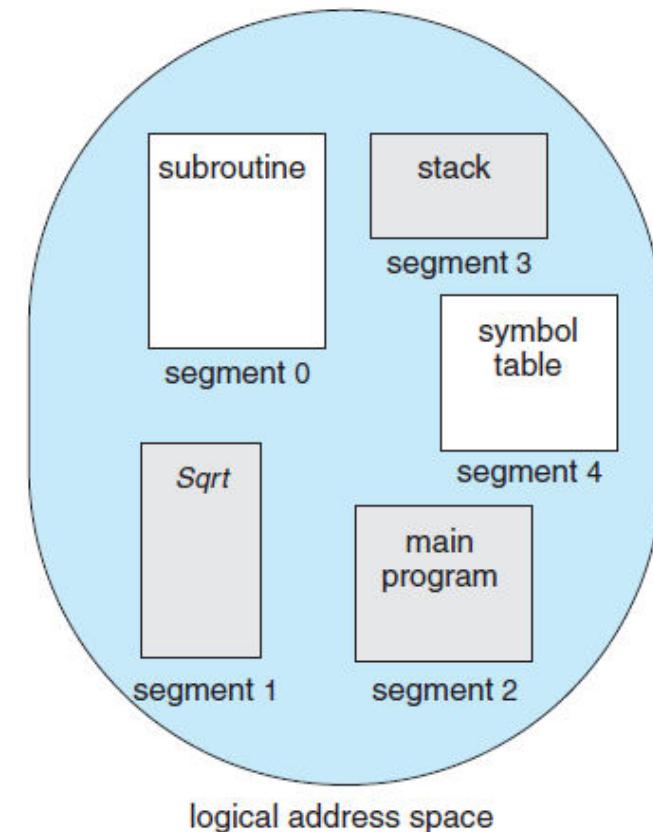
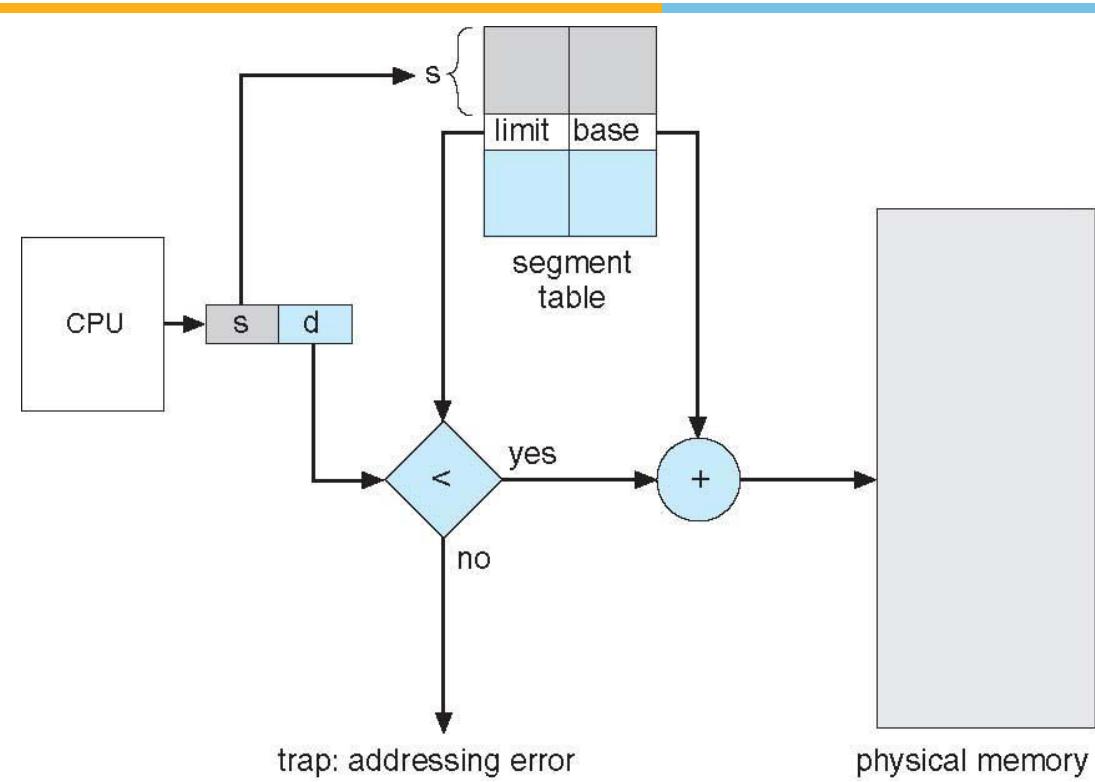
- ❖ A program is a collection of variable sized segments
- ❖ A segment is a logical unit such as:
 - main program
 - procedure
 - function
 - method
 - object
 - local variables, global variables
 - stack
 - symbol table
 - arrays
- ❖ Memory-management scheme that supports user view of memory, logic space for a process broken into segments, these segments are mapped to phy. address
- ❖ Logical address space is a collection of segments



Segmentation Architecture

- ❖ Logical address consists of a two tuple: <segment-number, offset>
- ❖ **Segment table** – maps logical addresses to physical addresses; each table entry has:
 - ❖ **segment base** – contains the starting physical address where the segment resides in memory
 - ❖ **segment limit** – specifies the length of the segment
- ❖ **Segment-table base register (STBR)** points to the segment table's location in memory
- ❖ **Segment-table length register (STLR)** indicates number of segments used by a program
- ❖ Segmentation has problem of external fragmentation, hence compaction can be used

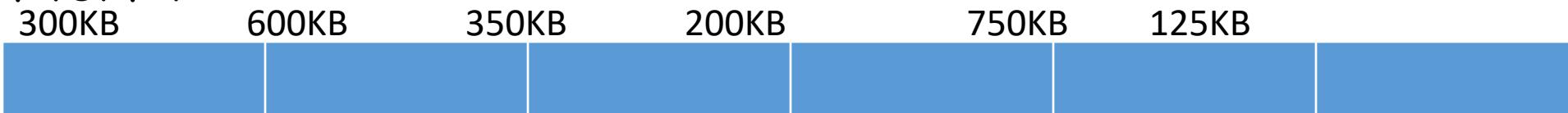
Segmentation Hardware



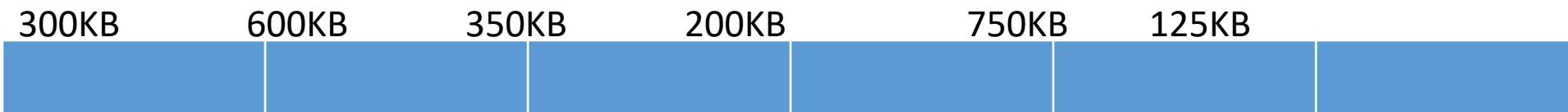
Problem

Given six memory partitions of 300 KB, 600 KB, 350 KB, 200 KB, 750 KB, and 125 KB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of size 115 KB, 500 KB, 358 KB, 200 KB, and 375 KB (in order)? Note: When hole is created it is treated as independent partition.

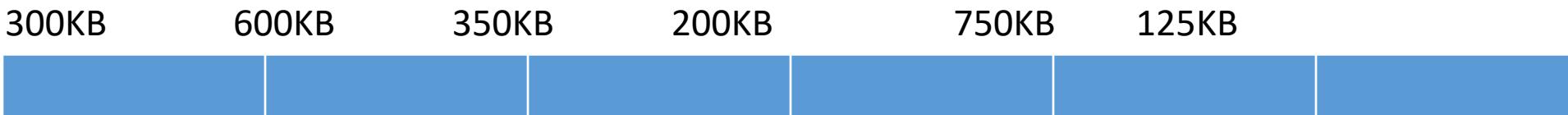
First Fit:



Best Fit:



Worst Fit

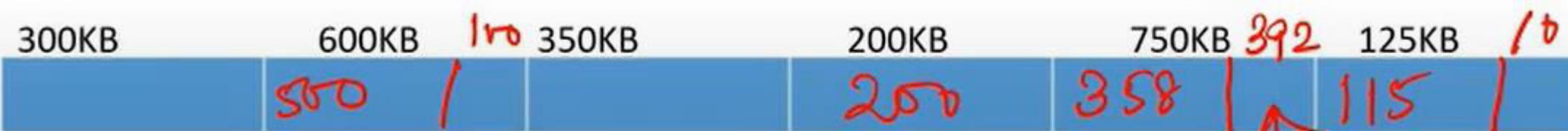


Given six memory partitions of 300 KB, 600 KB, 350 KB, 200 KB, 750 KB, and 125 KB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of size 115 KB, 500 KB, 358 KB, 200 KB, and 375 KB (in order)? Note: When hole is created it is treated as independent partition.

First Fit:



Best Fit:



Worst Fit



Paging (memory management scheme)



- ❖ Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
 - ❖ Avoids external fragmentation
 - ❖ Avoids need for compaction
- ❖ Divide physical memory into fixed-sized blocks called **frames**
 - ❖ Size is power of 2
- ❖ Divide logical memory into blocks of same size as frame-size called **pages**
- ❖ Keep track of all free frames
- ❖ To run a program of size N pages, need to find N free frames and load program
- ❖ Set up a **page table** to translate logical to physical addresses
- ❖ Still have Internal fragmentation, ex- suppose logical memory size=1275KB ,page size=8KB, then pages=160 ;5 KB space of last page wasted

Example

Process A : 4 pages

Process B : 3 pages

Process C : 4 Pages

Process D : 5 Pages

Main Memory : 15 frames

Frame number	Main memory
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	

(a) Fifteen available frames

Frame number	Main memory
0	A.0
1	A.1
2	A.2
3	A.3
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	

(b) Load process A

Frame number	Main memory
0	A.0
1	A.1
2	A.2
3	A.3
4	B.0
5	B.1
6	B.2
7	
8	
9	
10	
11	
12	
13	
14	

(c) Load process B

Frame number	Main memory
0	A.0
1	A.1
2	A.2
3	A.3
4	B.0
5	B.1
6	B.2
7	C.0
8	C.1
9	C.2
10	C.3
11	
12	
13	
14	

(d) Load process C

Frame number	Main memory
0	A.0
1	A.1
2	A.2
3	A.3
4	D.0
5	D.1
6	D.2
7	C.0
8	C.1
9	C.2
10	C.3
11	
12	
13	
14	

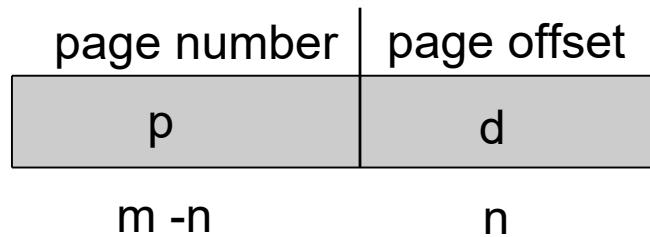
(e) Swap out B

Frame number	Main memory
0	A.0
1	A.1
2	A.2
3	A.3
4	D.0
5	D.1
6	D.2
7	C.0
8	C.1
9	C.2
10	C.3
11	D.3
12	D.4
13	
14	

(f) Load process D

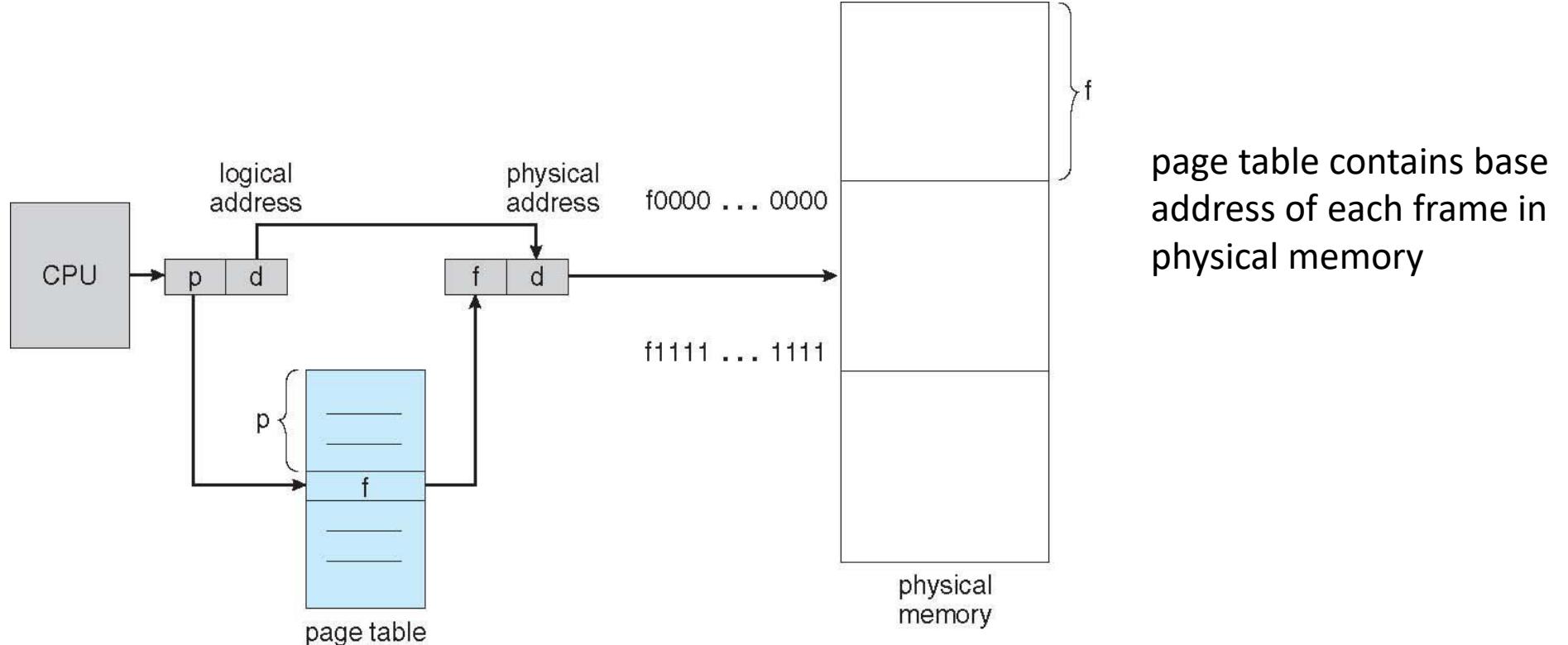
Address Translation Scheme

- ❖ Address generated by CPU is divided into:
 - ❖ **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

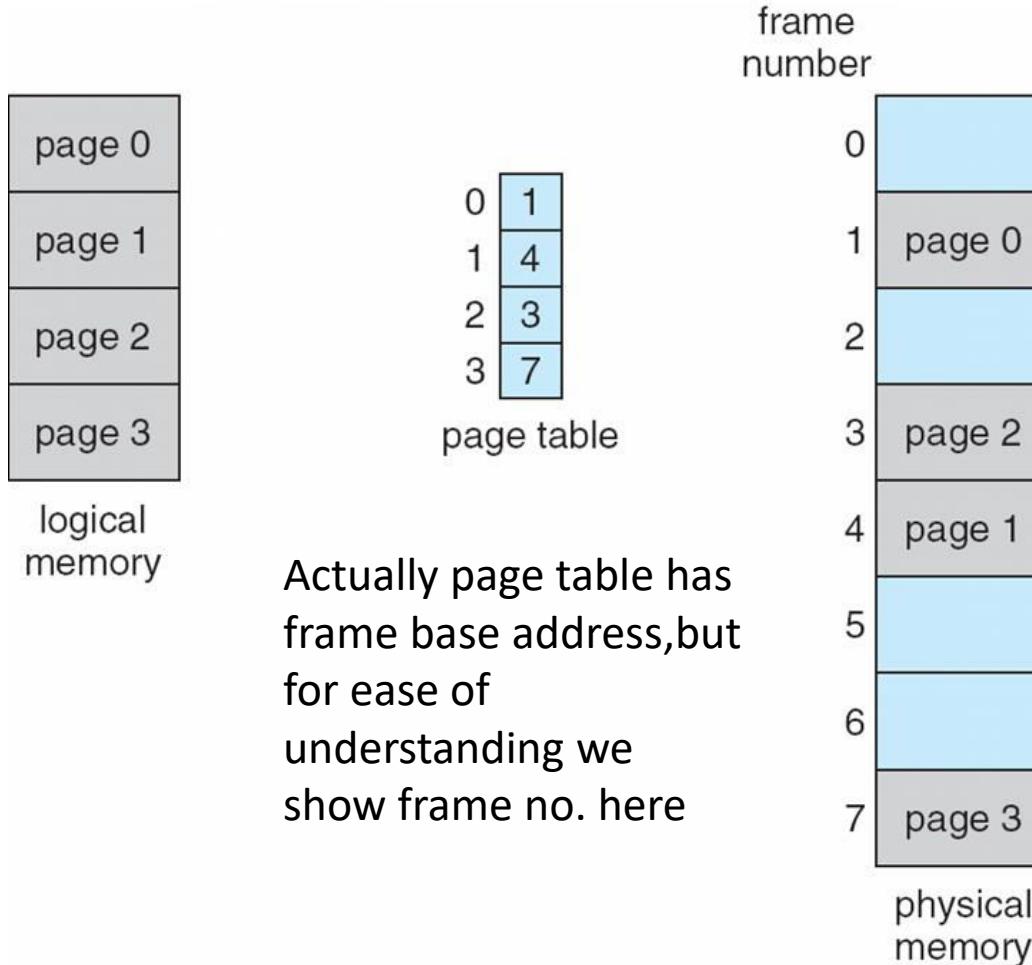


- ❖ logical address space 2^m bytes and page size 2^n bytes
- ❖ Here $(m-n) = \log(\text{total number of pages})$
- ❖ $n = \log(\text{size of one page})$

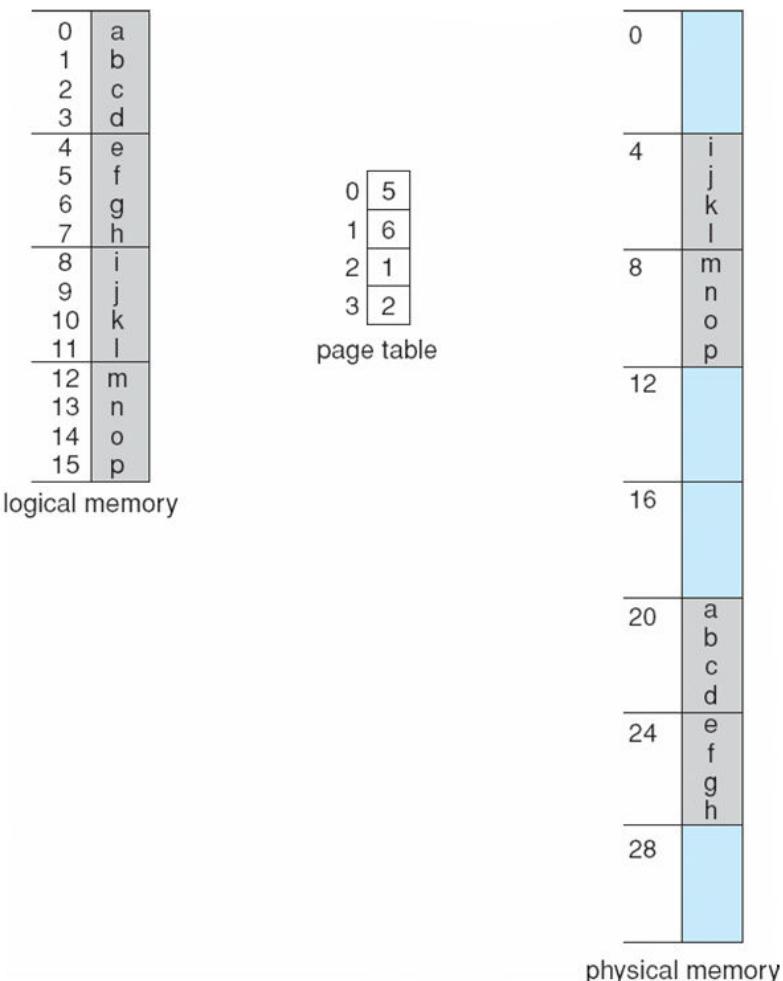
Paging Hardware



Paging Model of Logical and Physical Memory



Paging Example



$n = 2$ and $m = 4$

32-byte memory and 4-byte pages

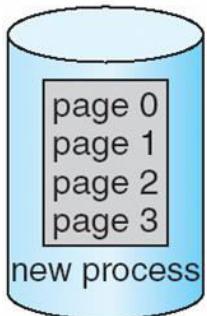
Paging

- Internal fragmentation
- So small frame sizes desirable?
- But each page table entry takes memory to track

Free Frames

free-frame list

14
13
18
20
15

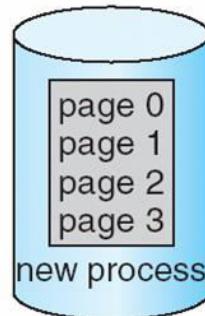


(a)

Before allocation

free-frame list

15



0	14
1	13
2	18
3	20

new-process page table

(b)

After allocation

free-frame list

13
page 1

14
page 0



18	page 2
19	
20	page 3

Frame table – one entry for each frame indicating whether the frame is free or allocated and if allocated, to which page of which process

Page Table Implementation

- Page table is kept in main memory
- **Page-table base register (PTBR)** points to the page table
- **Page-table length register (PTLR)** indicates size of the page table
- In this scheme every data/instruction access requires two main memory accesses
 - One for the page table stored in main memory and one for the data / instruction stored in some frame of main memory
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers (TLBs)**

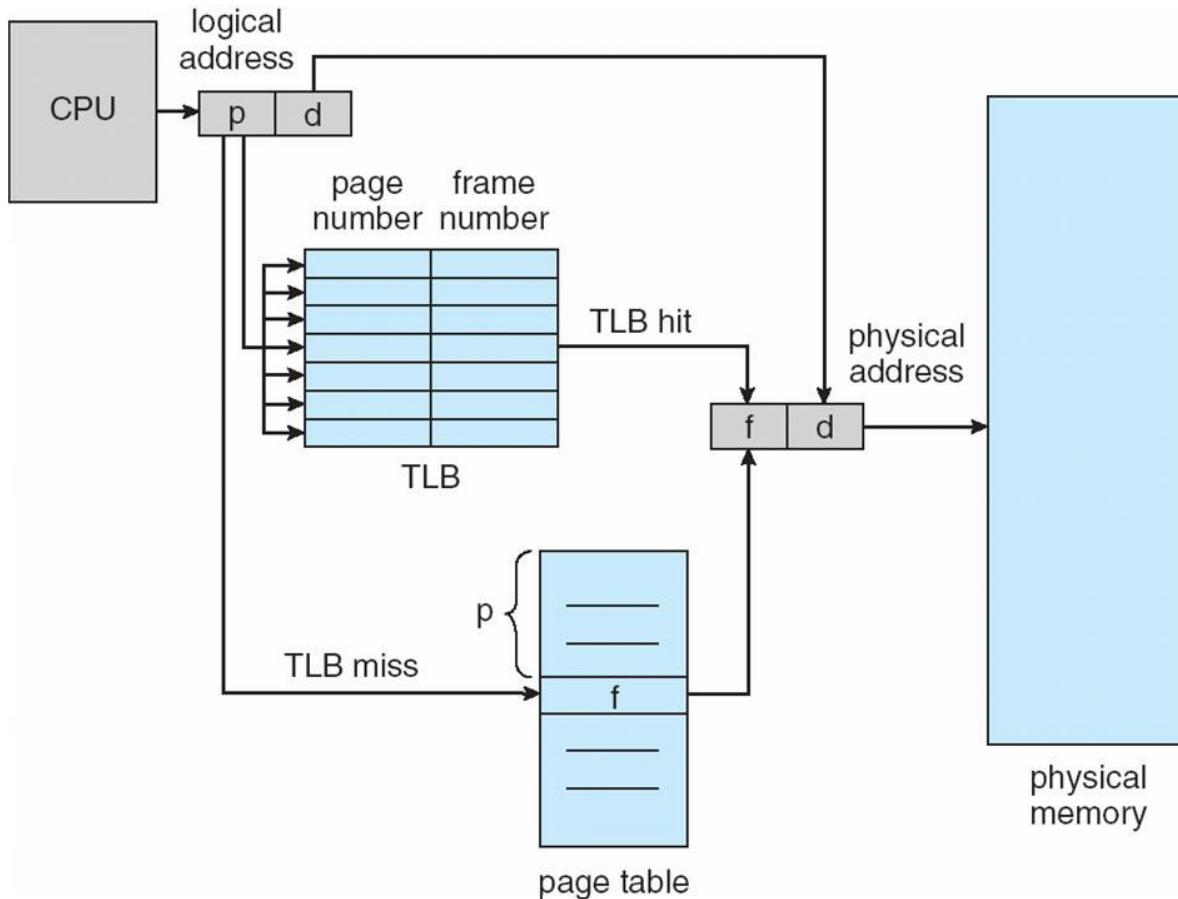
Associative Memory/TLB

- Associative memory – BIG O(1)parallel search
- Key-value pair fashion
- Made from cache-memory
- Size of tlb<<<page table

Page #	Frame #

- Address translation (p, d)
 - If p is in associative memory, get frame # out
 - Otherwise get frame # from page table in memory

Paging Hardware With TLB



Effective Access Time

- Associative/TLB Lookup = ε time unit
- Hit ratio = α
 - Hit ratio – percentage of times that a page number is found in the associative/TLB memory
- Consider $\alpha = 80\%$, $\varepsilon = 20\text{ns}$ for TLB search, 100ns for memory access
- **Effective Access Time (EAT)**

Consider $\alpha = 80\%$, $\varepsilon = 20\text{ns}$ for TLB search, 100ns for memory access

- $EAT = 0.80 \times 120 + 0.20 \times 220$

Problem

Consider a logical address space of 32 pages of 1,024 bytes each, mapped onto a physical memory of 64 frames.

- How many bits are there in the logical address?
- How many bits are there in the physical address?

Problem

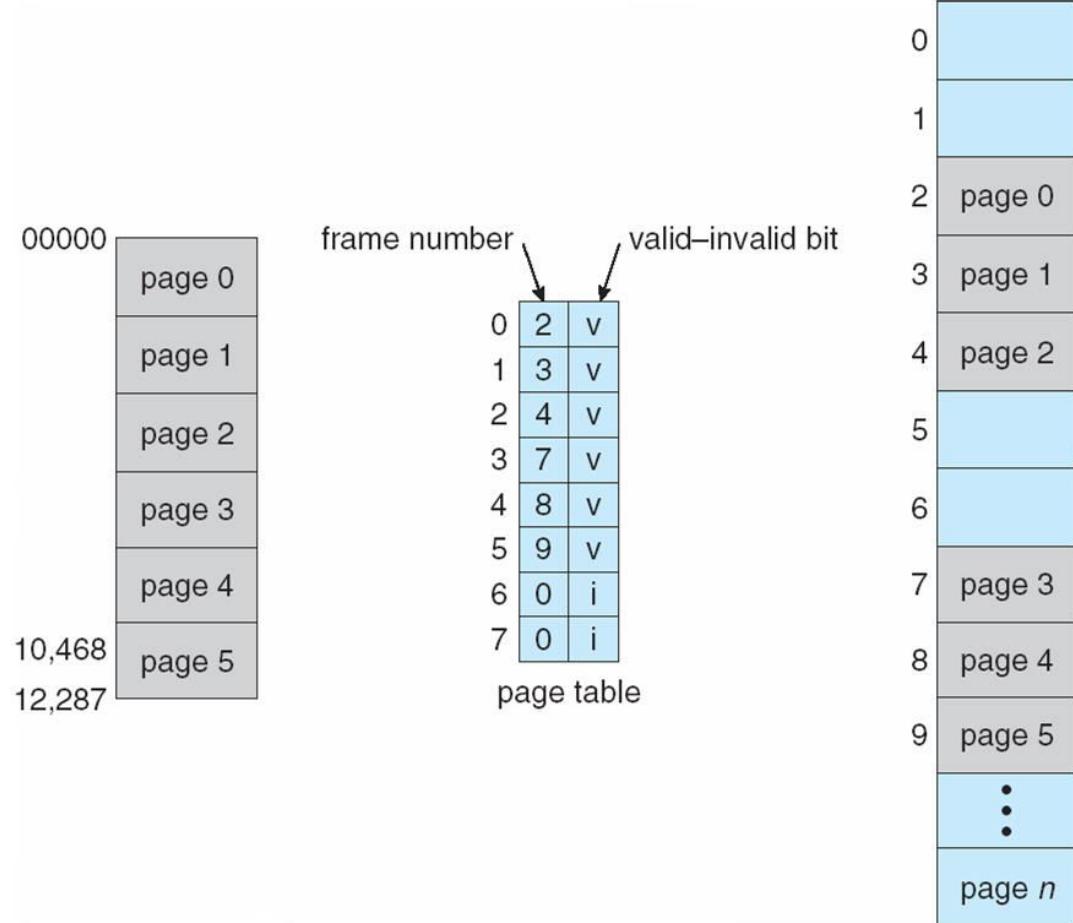
Consider a logical address space of 32 pages of 1,024 bytes each, mapped onto a physical memory of 64 frames.

- How many bits are there in the logical address?
- How many bits are there in the physical address?

$$\begin{array}{l} \xrightarrow{15} d=10 \\ \downarrow \quad \quad \quad 2^5 \times 2^{10} \\ 16 \quad \quad \quad 2^6 \times 2^{10} \end{array}$$

Memory Protection

- Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed
 - Can also add more bits to indicate page execute-only, and so on
- **Valid-invalid** bit attached to each entry in the page table:
 - “valid” indicates that the associated page is in the process’s logical address space, and is thus a legal page
 - “invalid” indicates that the page is not in the process’s logical address space
- Any violations result in a trap to the kernel

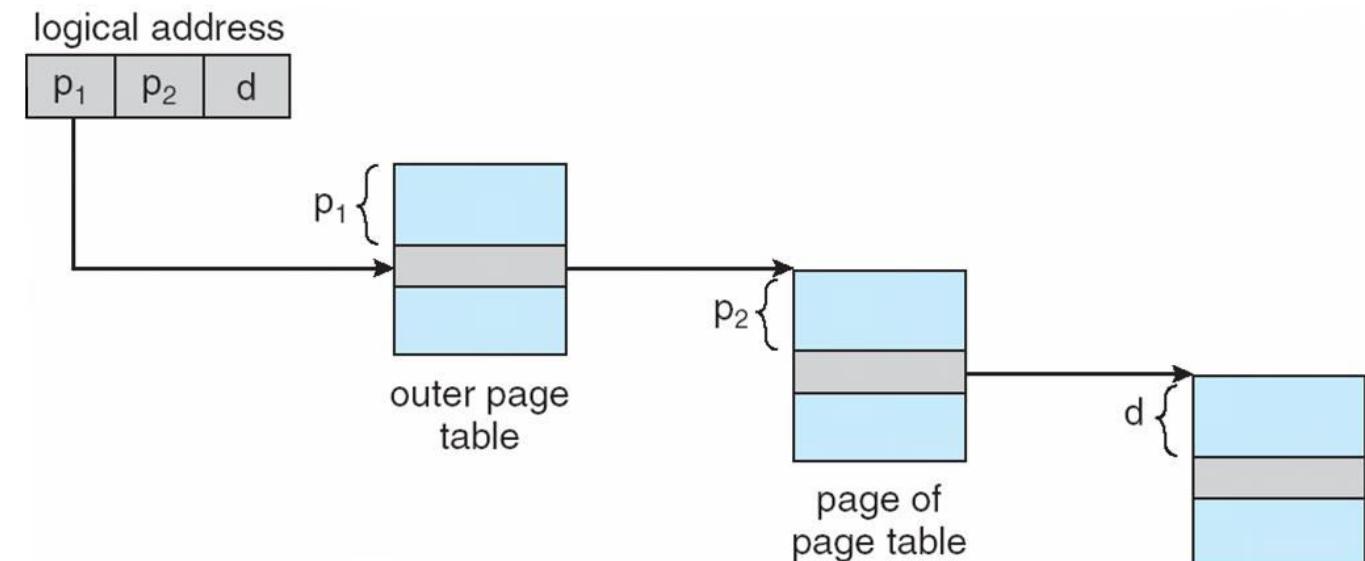
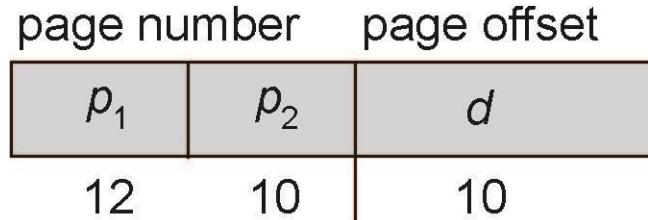


Page Table Structure

- Memory structures for paging can get huge using straight-forward methods
 - Cost a lot
 - Don't want to allocate that contiguously in main memory
- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

Hierarchical Page Tables

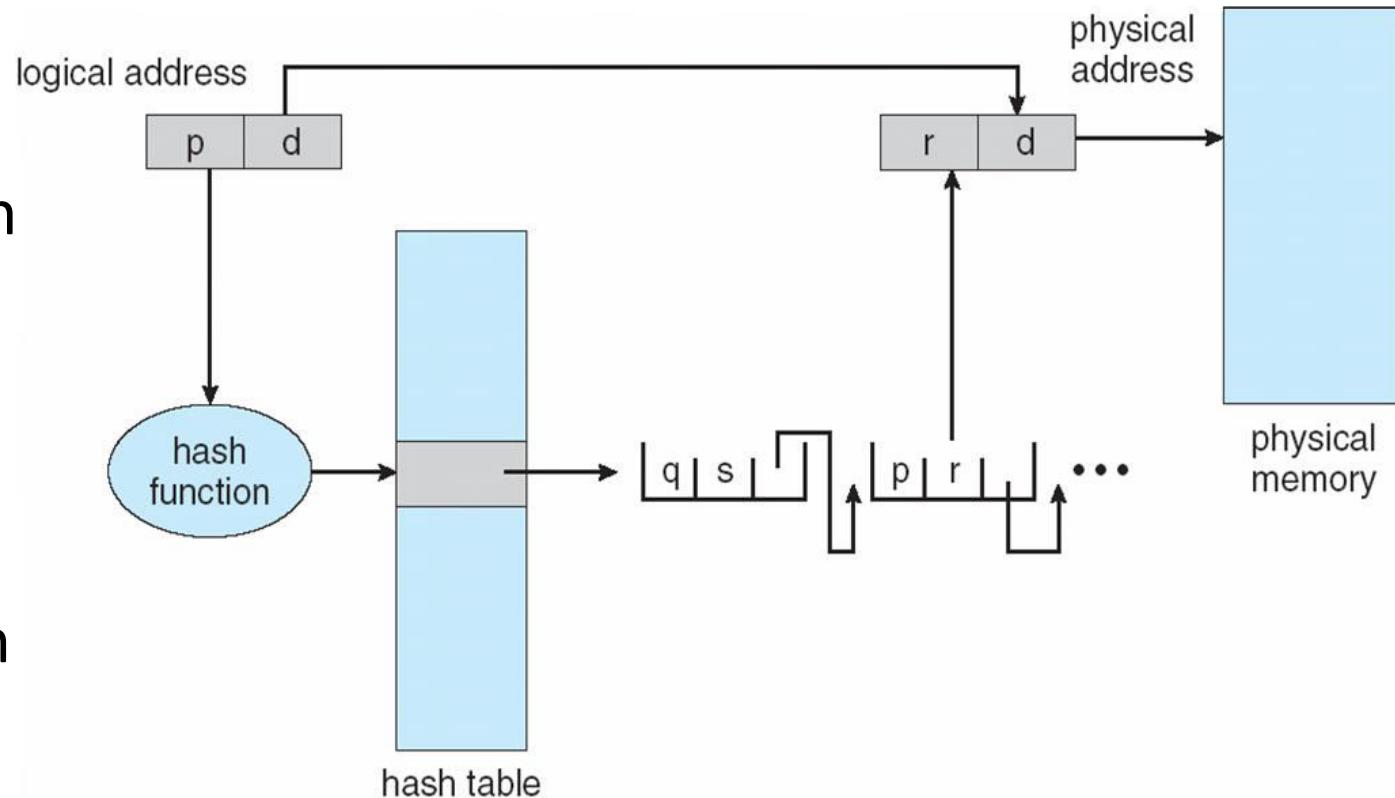
- Break up the logical address space into multiple page tables
- Two-level page table
- We then page the page table



- p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table

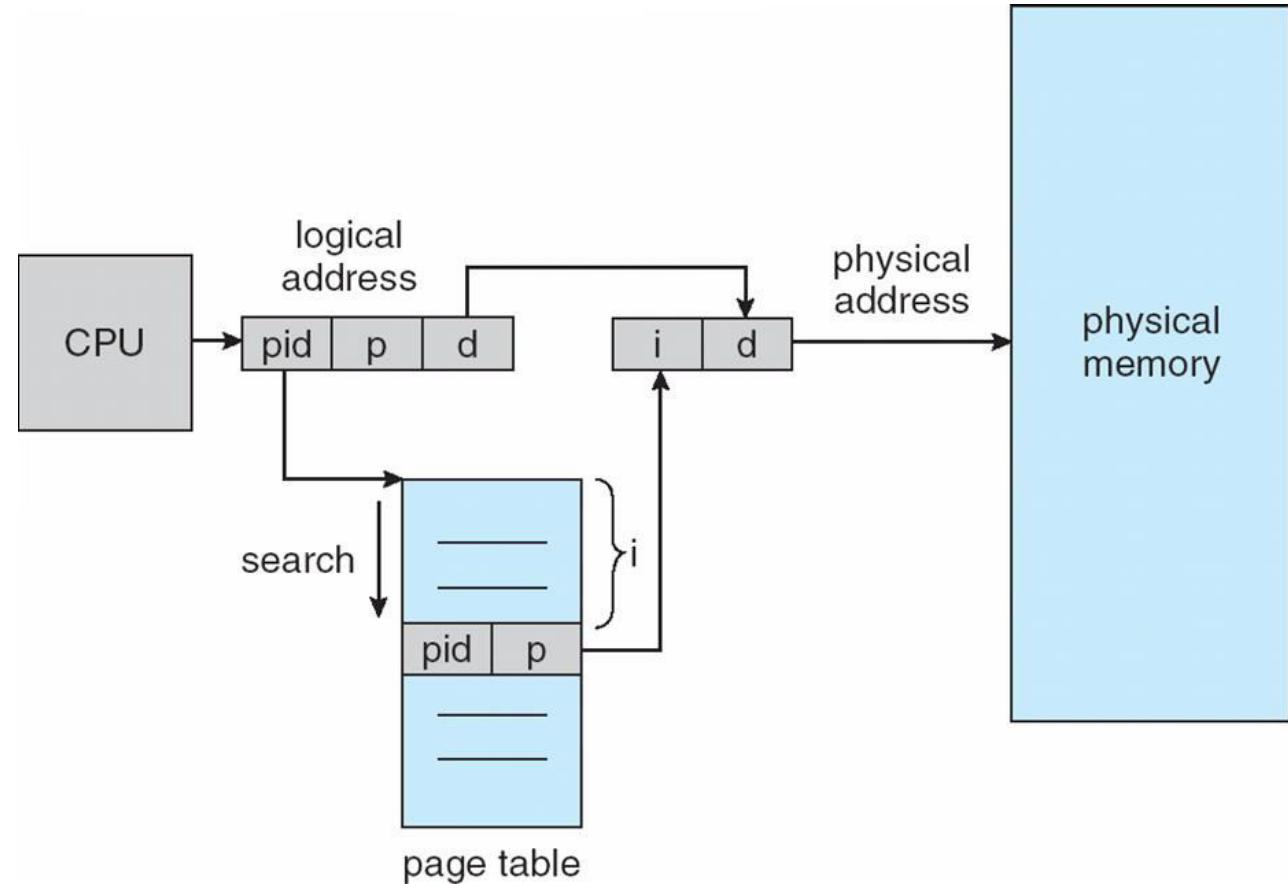
Hashed Page Table

- The virtual page number is hashed into a page table
- Page table contains a chain of elements hashing to the same location
- 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 chain searching for a match
- If a match is found, the corresponding physical frame is extracted



Inverted Page Table

- We will have a single common page table for all processes
- Track all physical pages(frames)
- One entry for each real page of PHY. memory(one entry in page table for one frame in physical memory)
- Pid present in logical address as all process have common page table
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page





Thank You



BITS Pilani
Hyderabad Campus

OPERATING SYSTEMS (CS F372)

Virtual Memory Management

Dr. Barsha Mitra
CSIS Dept., BITS Pilani, Hyderabad Campus



Background

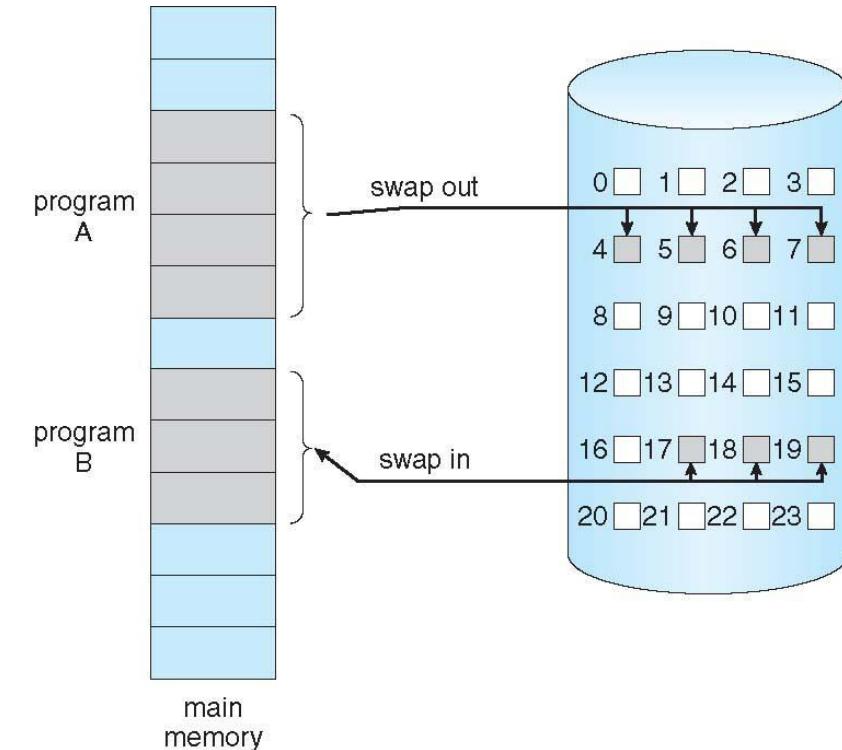
- ❖ Code needs to be in memory to execute, but entire program rarely used
 - ❖ Error code, unusual routines, large data structures
- ❖ Entire program code not needed at same time
- ❖ Consider ability to execute partially-loaded program
 - ❖ Program no longer constrained by limits of physical memory
 - ❖ 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

Background

-
- ❖ **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
 - ❖ 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

Demand Paging(way to implement virtual memory concept)

- ❖ 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
- ❖ Page is needed \Rightarrow reference to it
 - ❖ invalid reference \Rightarrow abort
 - ❖ not-in-memory \Rightarrow bring to memory
- ❖ **Lazy Swapper, Pager**



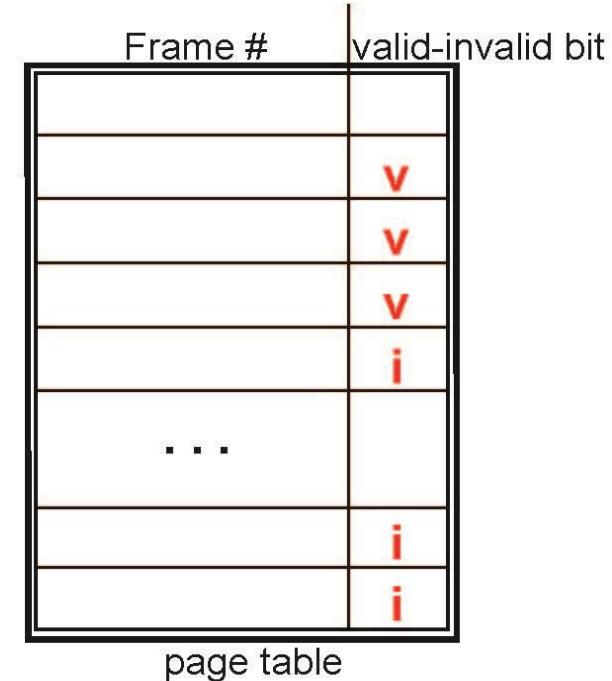
Demand Paging



- ❖ Pager guesses which pages will be used before swapping out again
- ❖ Pager brings in only those pages into memory
- ❖ How to determine that set of pages?
 - ❖ Need new MMU functionality to implement demand paging
- ❖ If pages needed are already **memory resident**
- ❖ If page needed and not memory resident
 - ❖ Need to detect and load the page into memory from storage

Valid-Invalid Bit (for demand paging)

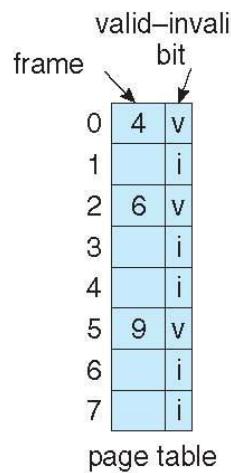
- ❖ With each page table entry a valid–invalid bit is associated
- ❖ **v** ⇒ legal and in-memory – **memory resident**
- ❖ **i** ⇒ either illegal or not-in-memory
- ❖ Initially valid–invalid bit is set to **i** on all entries
- ❖ During MMU address translation, if valid–invalid bit in page table entry is **i** ⇒ page fault



Valid-Invalid Bit

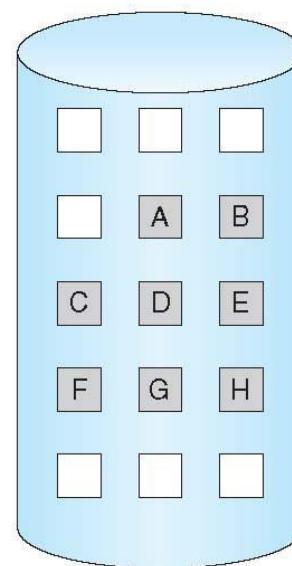
0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H

logical memory



0	
1	
2	
3	
4	A
5	
6	C
7	
8	
9	F
10	
11	
12	
13	
14	
15	

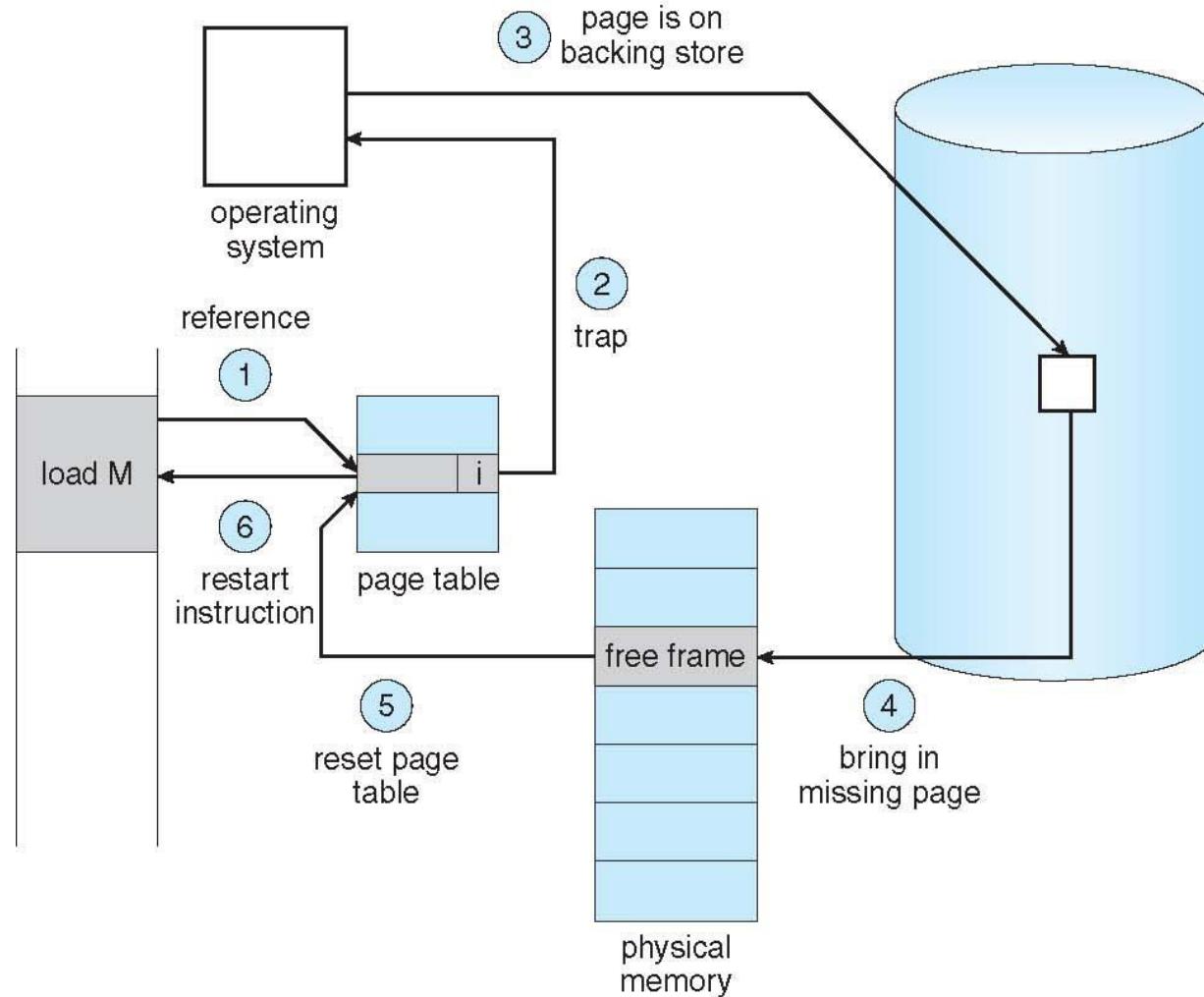
physical memory



Page Fault

- ❖ If there is a reference to a page, first reference to that page will trap to operating system: **page fault**
 - ❖ Operating system looks at an internal table (usually associated with PCB) to decide:
 - ❖ Invalid reference \Rightarrow abort
 - ❖ Just not in memory
 - ❖ Find free frame
 - ❖ Swap page into frame via (secondary memory)disk operation
 - ❖ Reset page table to indicate page now in memory
Set valid/invalid bit = **v**
 - ❖ Restart the instruction that caused the page fault
-

Page Fault



Demand Paging

- ❖ Pure Demand Paging – even at start, apart from bare minimum, don't bring any pages , as in when demand come, only then bring page, no guessing of which page to bring
- ❖ Locality of reference - tendency of a processor to access the same set of memory locations repetitively over a short period of time
- ❖ Hardware support needed for demand paging
 - ❖ Page table with valid / invalid bit
 - ❖ Secondary memory (swap device with **swap space**)
 - ❖ Instruction restart

Performance of Demand Paging

- Three major activities

- Service the interrupt
 - Read the page
 - Restart the process

- 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

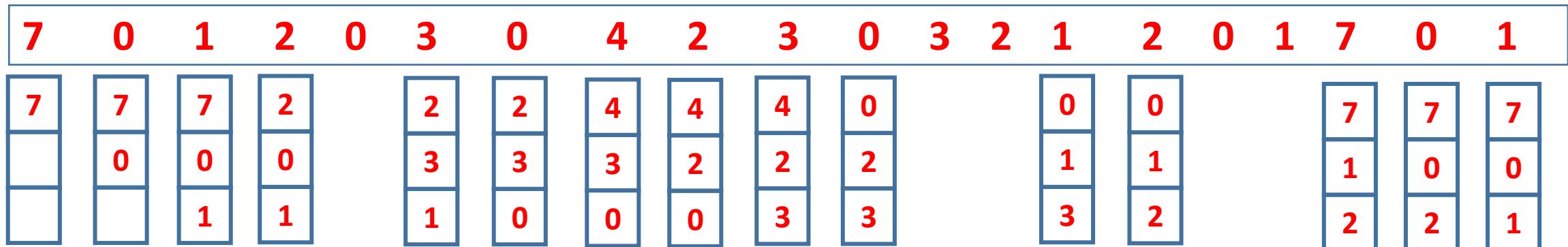
- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- EAT = $(1 - p) \times 200 + p (8 \text{ milliseconds})$
 $= (1 - p) \times 200 + p \times 8,000,000 = 200 + p \times 7,999,800$

Page Replacement

- ❖ Use **modify (dirty) bit** to reduce overhead of page transfers – only modified pages are written to disk
- ❖ Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory
- ❖ Find the location of the desired page on disk
- ❖ Find a free frame:
 - If there is a free frame, use it
 - If there is no free frame, use a page replacement algorithm to select a **victim frame**
 - Write victim frame to disk if dirty
- ❖ Bring the desired page into the (newly) free frame; update the page and frame tables
- ❖ Continue the process by restarting the instruction that caused the trap
- ❖ **Note: reference string never contains same pages adjacent to one another**

First-In-First-Out (FIFO) Algorithm

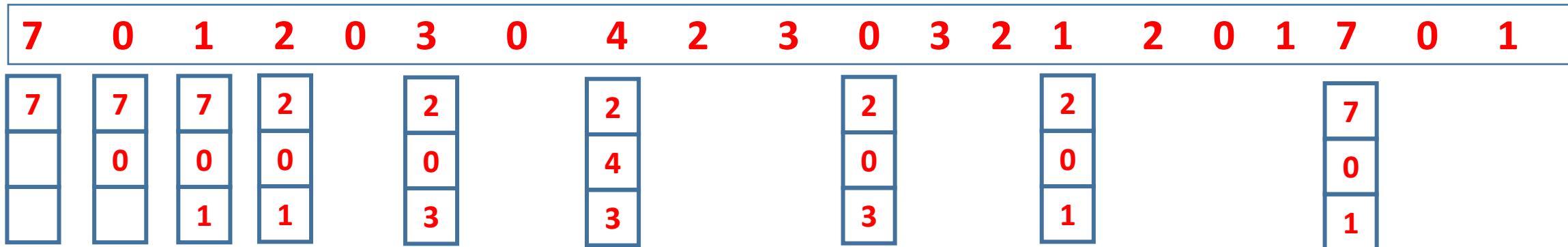
- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)



- In FIFO, replace the one which entered the very earliest, doesn't matter if it was page hit in between, basically follow fixed pointer approach as above.
- Adding more frames can cause more page faults!
 - **Belady's Anomaly**
- How to track ages of pages?
 - Just use a FIFO queue

Optimal Algorithm

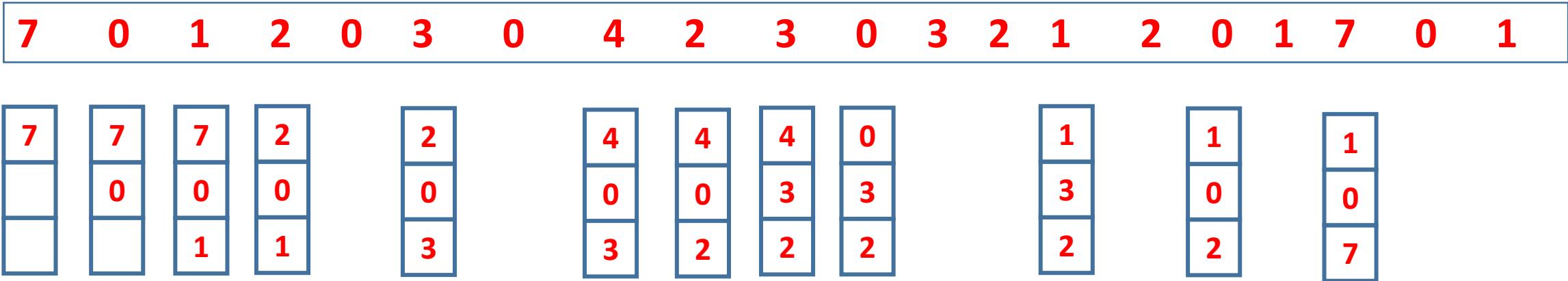
- Replace page that will not be used for longest period of time
 - 9 is optimal for the example
- How do you know this?
 - Can't read the future
- Used for measuring how well your algorithm performs



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



- ❖ 12 faults – better than FIFO but worse than OPT
- ❖ Generally good algorithm and frequently used
- ❖ But how to implement?

Least Recently Used (LRU) Algorithm

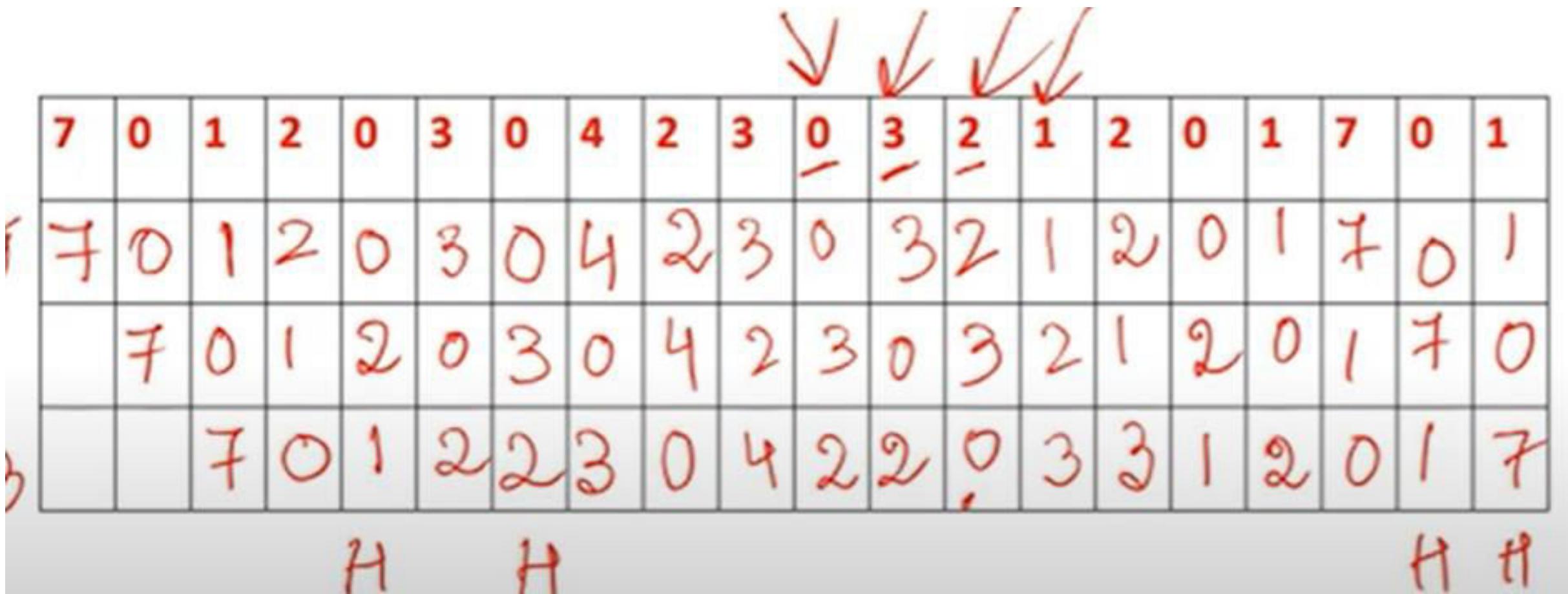


- ❖ Counter implementation
 - ❖ Every page table entry has a counter; every time page is referenced through this entry, copy the clock into the counter
 - ❖ When a page needs to be changed, look at the counters to find smallest value
- ❖ Stack implementation
 - ❖ Keep a stack of page numbers
 - ❖ Page referenced - move it to the top
- ❖ LRU and OPT don't have Belady's Anomaly

Least Recently Used (LRU) Algorithm



Using stack implementation



Allocation of Frames



- ❖ Each process needs ***minimum*** number of frames, decided by computer architecture
- ❖ ***Maximum*** ofcourse is total frames in the system (minus the frames allocated to OS).
- ❖ Two major allocation schemes
 - ❖ Equal allocation
 - ❖ Proportional allocation

Frame Allocation



- ❖ **Equal allocation** – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames
- ❖ **Proportional allocation** – Allocate according to the size of process
 - ❖ Dynamic as degree of multiprogramming, process sizes change

s_i = size of process p_i

$$S = \sum s_i$$

m = total number of frames available after os is given some

$$a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$$

$$m = 64$$

$$s_1 = 10$$

$$s_2 = 127$$

$$a_1 = \frac{10}{137} \quad 62 \quad 4$$

$$a_2 = \frac{127}{137} \quad 62 \quad 57$$

Assume that 2 frames will be allocated for OS out of 64

Global replacement of frame:

If a process wants more frames during execution, it can replace frames of other processes in main memory, i.e., victim frame can be a frame allocated to some other process as well.

Effect of thrashing is more severe when global replacement is used, still thrashing is observed in local replacement also.

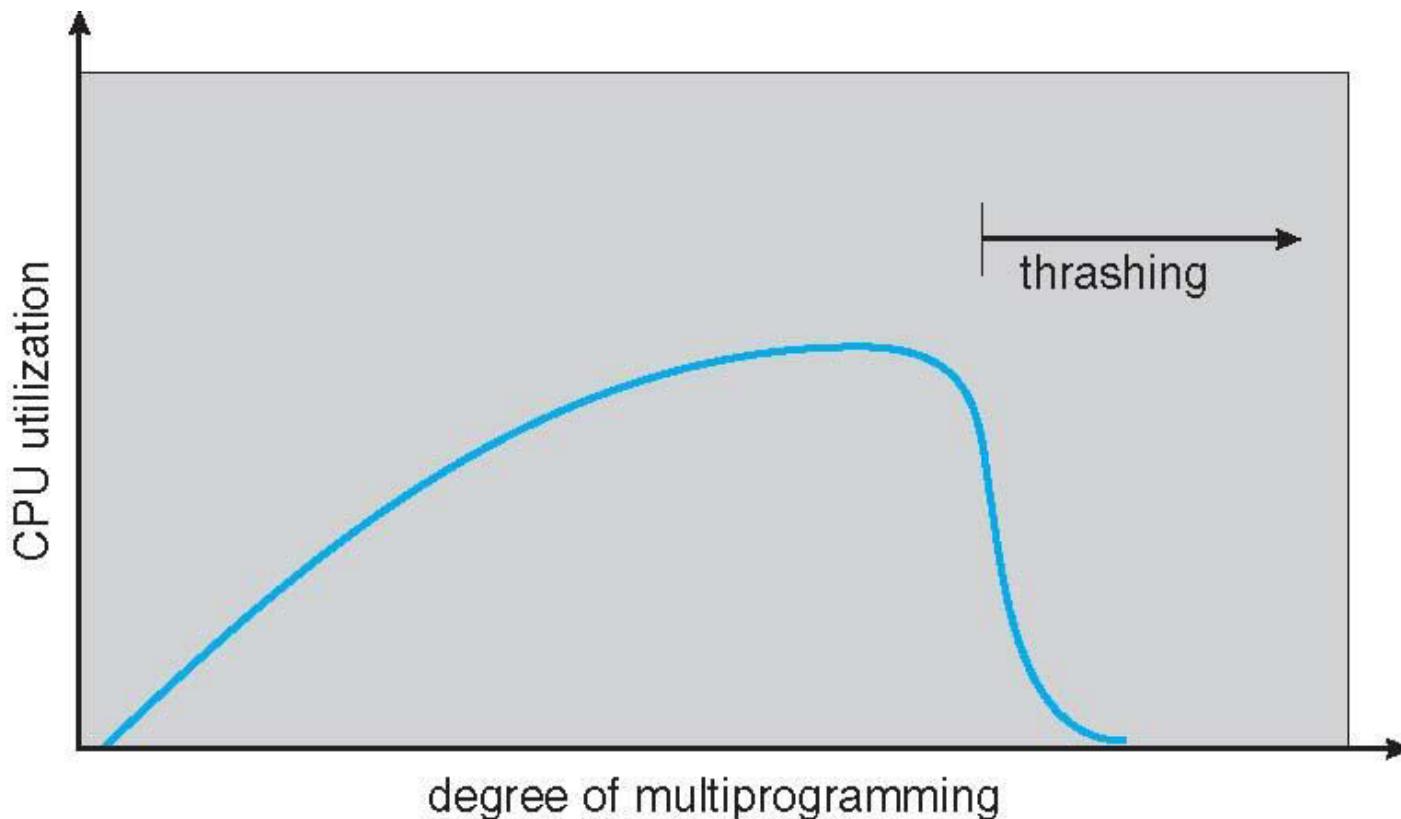
Local replacement of frame:

If a process wants more frames during execution, it can't replace frames of other processes in main memory.

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 replaced frame back
 - ❖ This leads to:
 - ❖ Low CPU utilization
 - ❖ Operating system thinking that it needs to increase the degree of multiprogramming
 - ❖ Another process added to the system
- ❖ **Thrashing** ≡ a process is busy swapping pages in and out, high paging activity

Thrashing



Demand Paging & Thrashing



- ❖ Why does demand paging work?

Locality model

- ❖ Locality is a set of pages that are actively used together
- ❖ Defined by program structure and data structures used
- ❖ Process migrates from one locality to another
- ❖ Contains several localities
- ❖ Localities may overlap

- ❖ Why does thrashing occur?

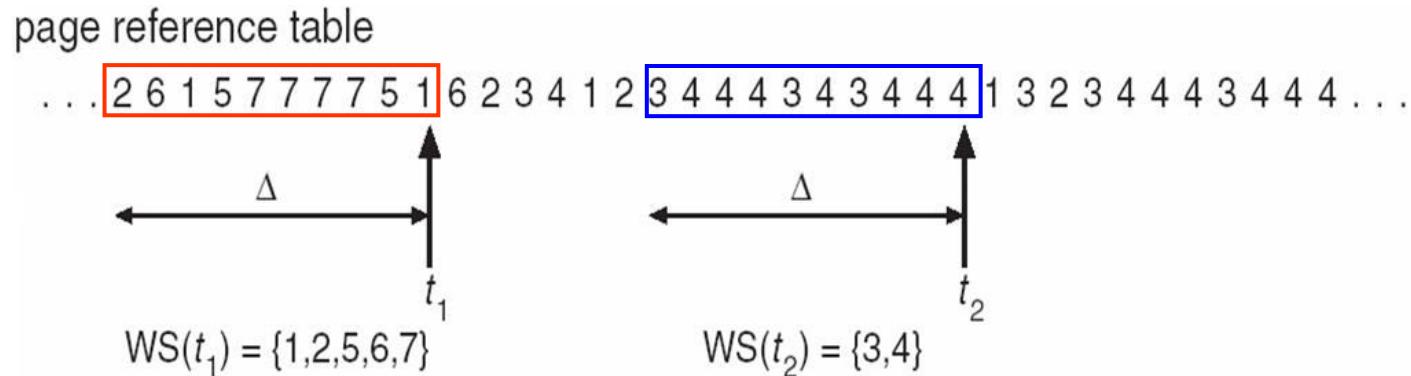
Σ size of locality > total memory size

- ❖ Limit effects by using local or priority page replacement
- ❖ Allocate enough frames for a single locality

Working Set Model



- ❖ $\Delta \equiv$ working-set window \equiv a fixed number of page references (usually most recent), approximation of program's locality



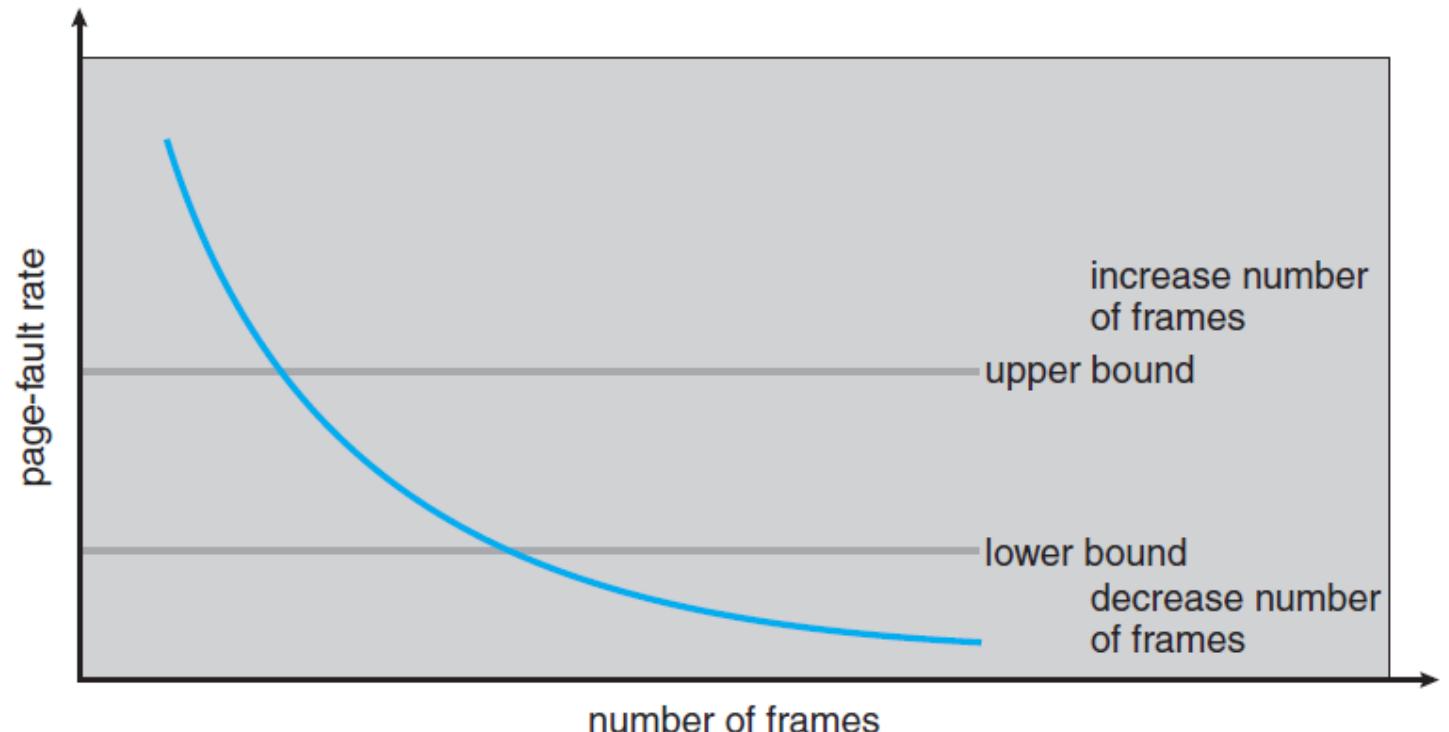
Working Set Model

- ❖ WSS-working set size; Working set model as a solution to thrashing
- ❖ WSS_i (working set of P_i) = total no. of pages referenced in the most recent Δ (varies in time)
 - ❖ if Δ too small will not encompass entire locality
 - ❖ if Δ too large will encompass several localities
 - ❖ if $\Delta = \infty \Rightarrow$ will encompass entire program
- ❖ $D = \sum WSS_i \equiv$ total demand for frames
 - ❖ Approximation of locality
- ❖ m = no. of available frames
- ❖ if $D > m \Rightarrow$ Thrashing
- ❖ Policy if $D > m$, then suspend or swap out one of the processes
- ❖ Keeps the degree of multiprogramming as high as possible
- ❖ Optimizes CPU utilization
- ❖ Difficulty is to keep track of the working set

Page Fault Rates (another solution to thrashing)



- ❖ Page-Fault Frequency (PFF)
- ❖ Define an upper and lower limits of page fault rate
- ❖ If page fault rate is too low → take away a frame
- ❖ If page fault rate is too high → allocate one more frame



Allocating Kernel Memory

- ❖ Treated differently from user memory
- ❖ Often allocated from a free-memory pool
 - ❖ Kernel requests memory for data structures of varying sizes
 - ❖ Some kernel memory needs to be contiguous
 - ❖ i.e., for device I/O, h/w devices interact directly with physical memory

Buddy System

- ❖ Allocates memory from fixed-size segment consisting of physically-contiguous blocks
- ❖ Define a maximum and minimum block size
- ❖ Memory allocated using **power-of-2 allocator**
 - ❖ Satisfies requests in units sized as power of 2
 - ❖ Request rounded up to next highest power of 2
 - ❖ When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2
 - ❖ Continue until appropriate sized chunk available

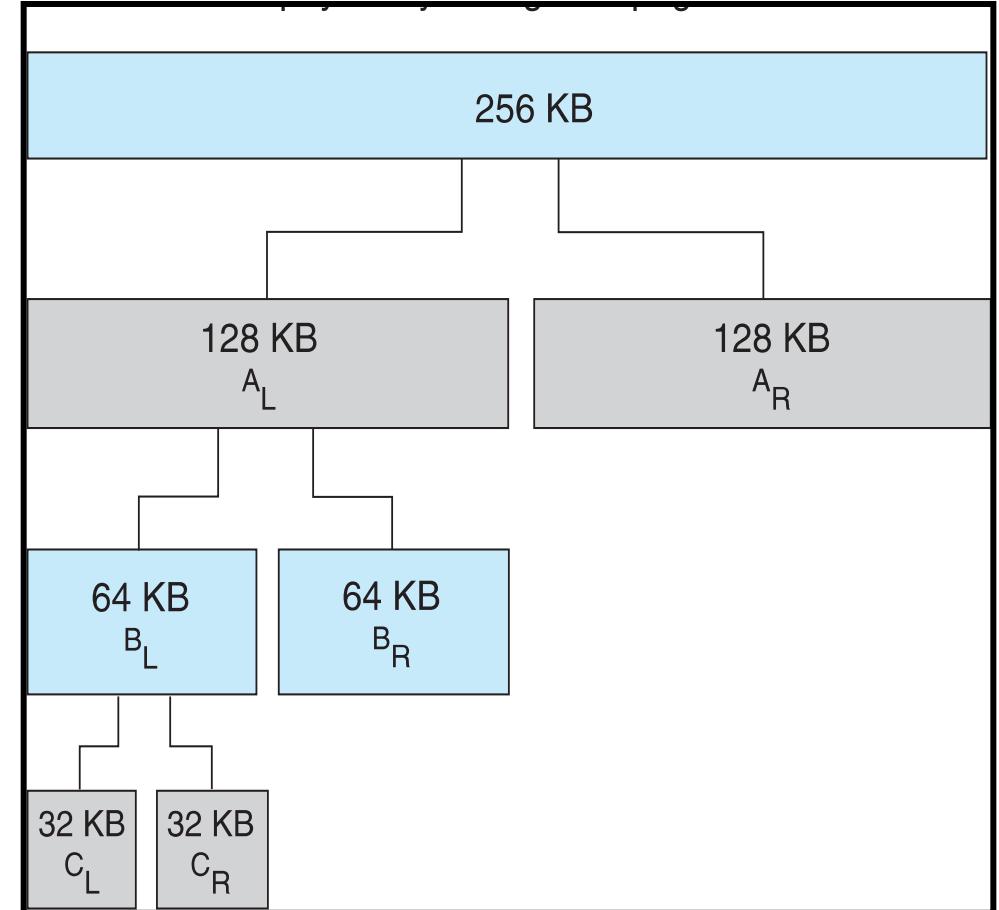
Buddy System

innovate

achieve

lead

- ❖ **Advantage** – quickly **coalesce** smaller chunks into larger chunk, reduces external fragmentation
- ❖ **Disadvantage** – internal fragmentation



Buddy System

- The **buddy system** is a memory allocation and management algorithm
- It manages memory in **power of two** increments
- Splitting memory into halves and to try to give a best fit

Contd...

- Provides two operations:
 - $\text{Allocate}(A, 2^k)$: Allocates a block of 2^k and marks it as allocated
 - $\text{Free}(A)$: Marks the previously allocated block A as free and merge it with other blocks to form a larger block
- Algorithm: Assume that a process P of size “X” needs to be allocated
 - **If $2^{K-1} < X \leq 2^K$:** Allocate the entire block 2^K
 - **Else:** Recursively divide the block equally and test the condition at each time, when it satisfies, allocate the block.

Note:

- 1) allocate buddy best fit, then from left chunk to right chunk
- 2) If u have a appropriate size chunk currently available, do not split or merge any other chunks ,just allocate the available chunk, example in next slide when allocate(F)

Problem 1

Consider a memory block of 16K. Perform the following:

Allocate (A: 3.5K)

Allocate (B: 1.2K)

Allocate (C: 1.3K)

Allocate (D: 1.9K)

Allocate (E: 3.2K)

Free (C)

Free (B)

Allocate (F: 1.6K)

Allocate (G: 1.8K)

Consider a memory block of 16K. Perform the following:

Allocate (A: 3.5K)	Free (C)
Allocate (B: 1.2K)	Free (B)
Allocate (C: 1.3K)	Allocate (F: 1.6K)
Allocate (D: 1.9K)	Allocate (G: 1.8K)
Allocate (E: 3.2K)	

lead

- 16K Memory Block



- Allocate (A: 3.5K)



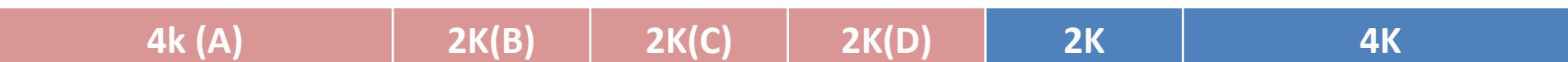
- Allocate (B: 1.2K)



- Allocate (C: 1.3K)



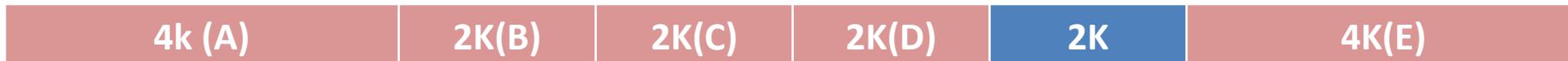
- Allocate (D: 1.9K)



Consider a memory block of 16K. Perform the following:

Allocate (A: 3.5K)	Free (C)
Allocate (B: 1.2K)	Free (B)
Allocate (C: 1.3K)	Allocate (F: 1.6K)
Allocate (D: 1.9K)	Allocate (G: 1.8K)
Allocate (E: 3.2K)	

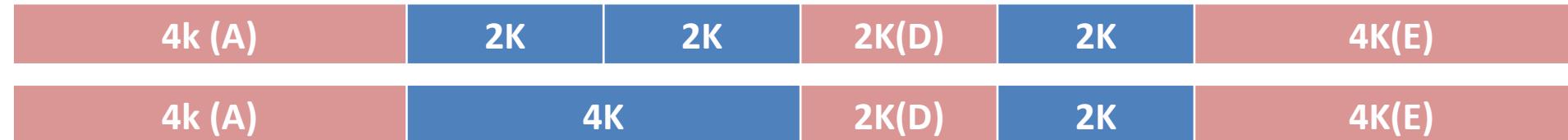
- Allocate (E: 3.2K)



- Free (C)



- Free (B)



- Allocate (F: 1.6K)



- Allocate (G: 1.8K)



Tree Structure

Consider a memory block of 16K. Perform the following:

Allocate (A: 3.5K)

Free (C)

Allocate (B: 1.2K)

Free (B)

Allocate (C: 1.3K)

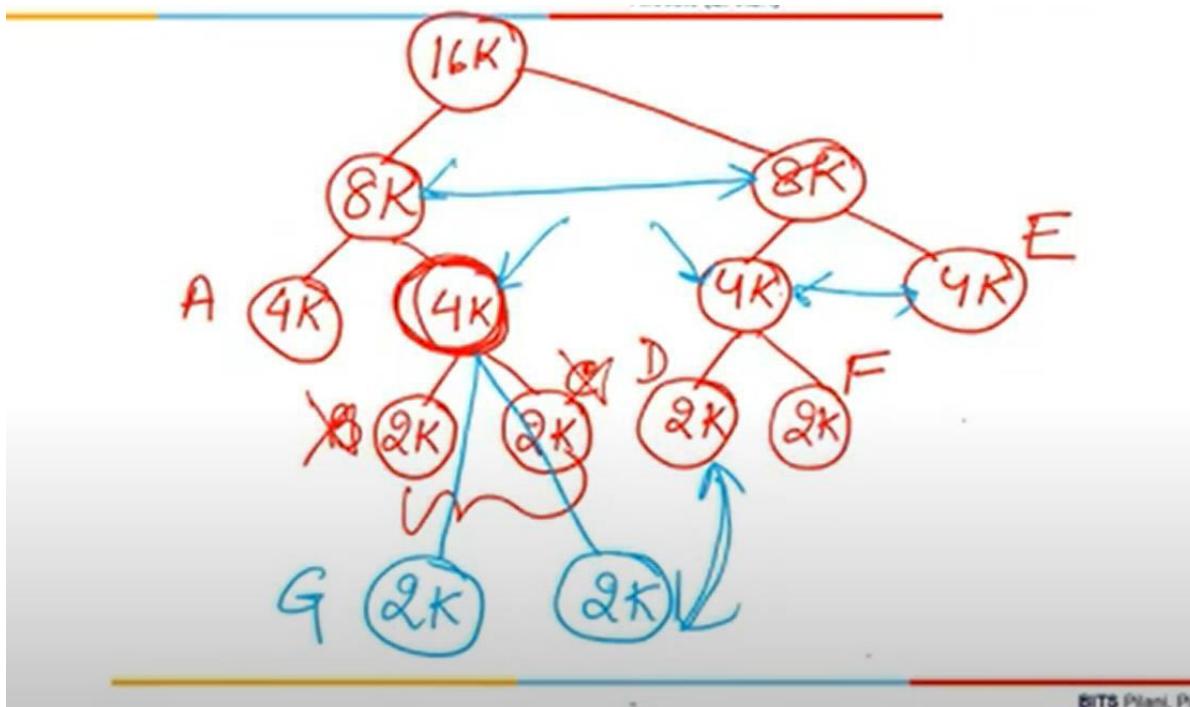
Allocate (F: 1.6K)

Allocate (D: 1.9K)

Allocate (G: 1.8K)

lead

2 buddies belonging to 2 different subtrees cannot be fused



BITS Pilani, Pilani

Problem 2

- Consider the state of memory after allocating 5 processes A, B, C, D and E



- What is the state of memory after freeing process D?



- What is the state of memory after freeing Process C?



Advantages and Disadvantages

Advantage -

- Easy to implement a buddy system (Linux)
- Allocates block of correct size
- It is easy to merge adjacent holes
- Fast to allocate memory and de-allocating memory

Disadvantage -

- It requires all allocation unit to be powers of two
- It leads to internal fragmentation



Thank You



BITS Pilani
Hyderabad Campus

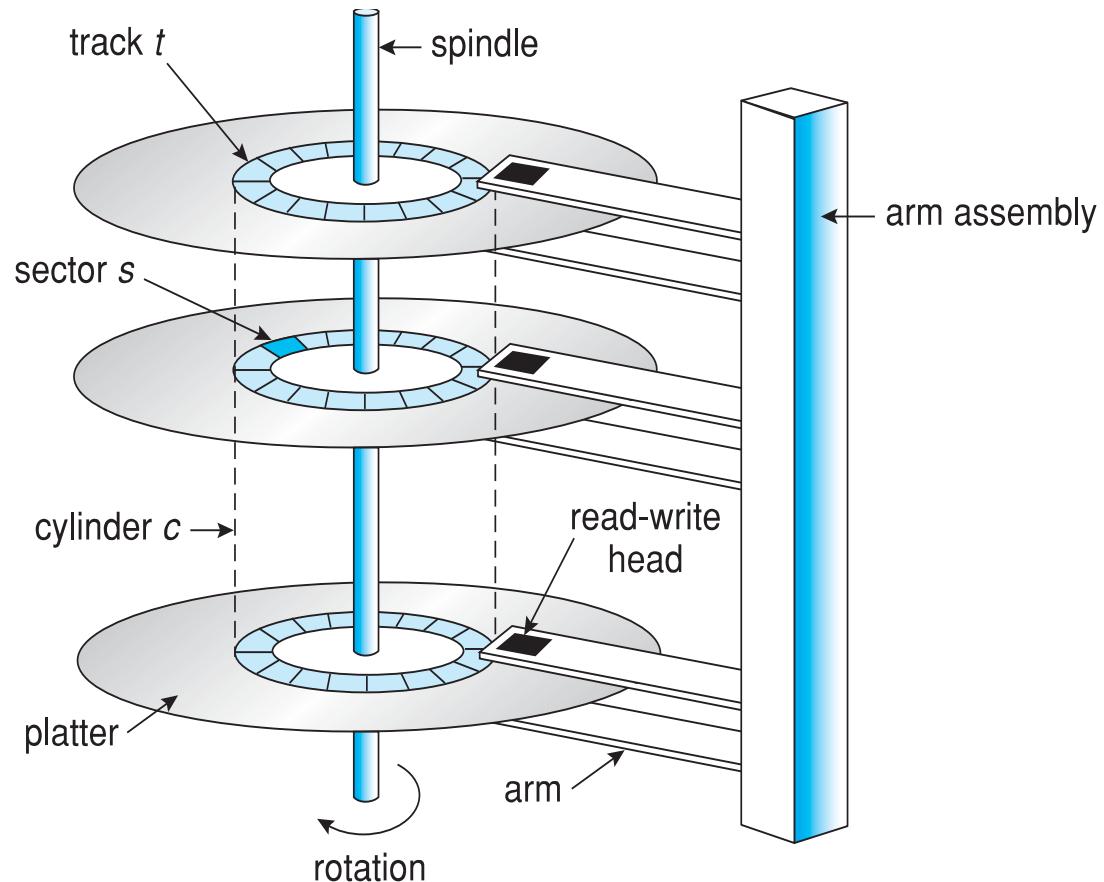
OPERATING SYSTEMS (CS F372)

Mass Storage

Dr. Barsha Mitra
CSIS Dept., BITS Pilani, Hyderabad Campus



Magnetic Disks



Mass Storage Structure

- **Magnetic disks** provide bulk of secondary storage of modern computers
 - Disk in use rotates at 60 to 250 times per second (RPM)
 - **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
- Disks can be removable
- Disk drive is attached to computer via **I/O bus**
 - Buses vary, including **ATA, SATA, USB, Fibre Channel (FC), etc.**

Disk Structure

- ❖ Disk drives are addressed as large 1-dimensional arrays of **logical blocks**, where the logical block is the smallest unit of transfer
- ❖ 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
 - ❖ Physical Address – cylinder number, track number, sector number

Disk Scheduling



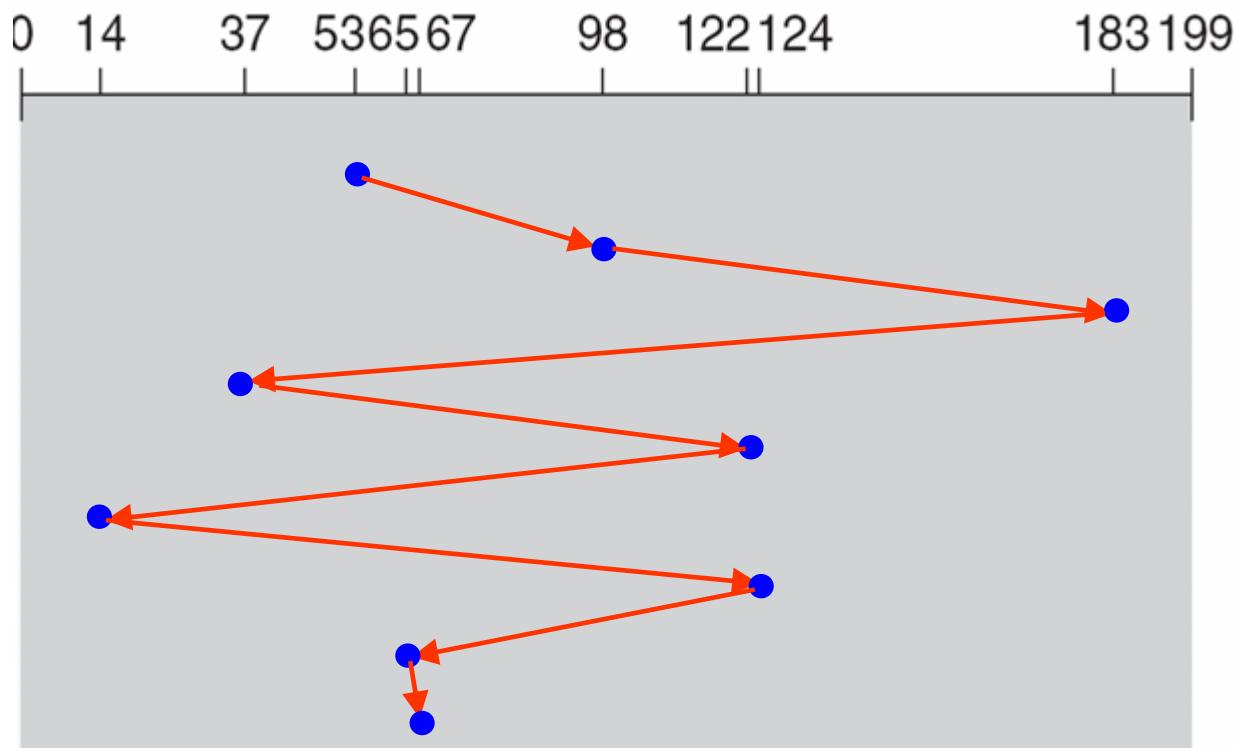
- ❖ Minimize seek time
- ❖ Disk **bandwidth** - total no. of bytes transferred divided by the total time between the first request for service and the completion of the last transfer
- ❖ OS maintains queue of I/O requests, per disk or device
- ❖ Idle disk can immediately work on I/O request, busy disk means request must be queued
- ❖ Next we look at some disk scheduling algorithms, basically how to move arm handler when io request pile up

FCFS

Total head movement of 640 cylinders

queue = 98, 183, 37, 122, 14, 124, 65, 67

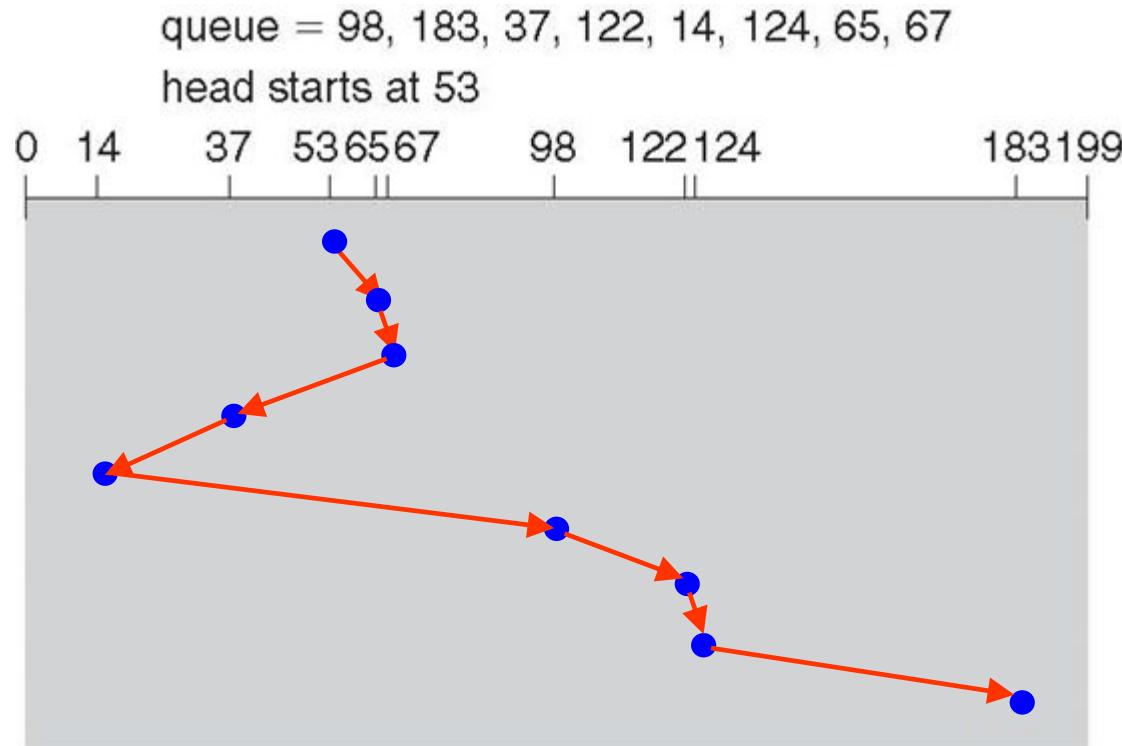
head starts at 53



SSTF(Shortest Seek Time First)



Total head movement of 236 cylinders



- ❖ Shortest Seek Time First selects the request with the minimum seek time from the current head position
- ❖ May cause starvation of some requests

SCAN /ELEVATOR Algorithm

innovate

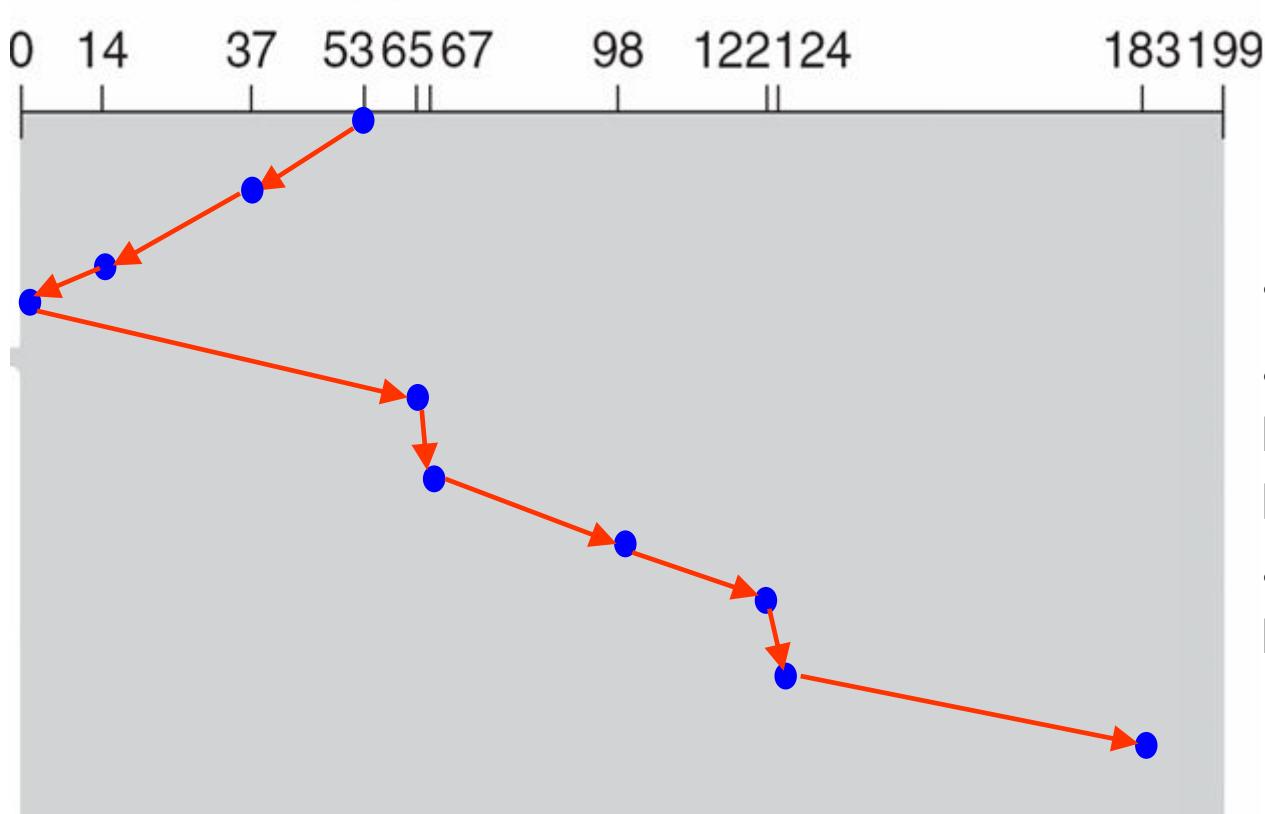
achieve

lead

❖ ELEVATOR Algorithm

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



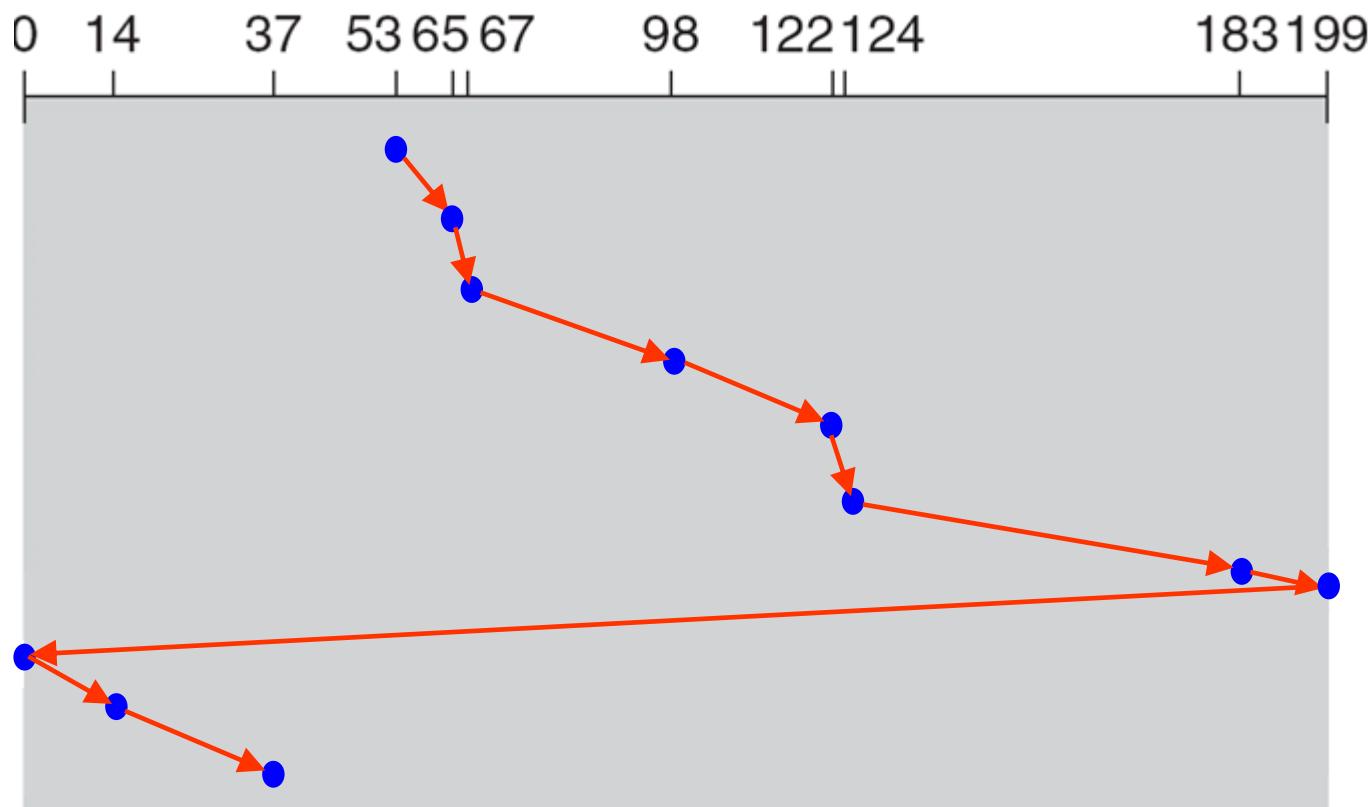
236 cylinders

- Direction of arm movement is needed
- If moving in a direction, move till u hit boundary if some more request are pending,
- if last request is done, no need to goto boundary

C-SCAN (circular-scan)

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



$$\begin{aligned}
 & (199 - 53) + \\
 & (199 - 0) + \\
 & (37 - 0) = \\
 & 382 \text{ cylinders}
 \end{aligned}$$

Usually Better
performance/r
esponse time
than scan

Disk Management

- ❖ 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**- creation of a file system, OS stores the initial file system data structures on the disk, data structures include maps of free and allocated space and an initial empty directory

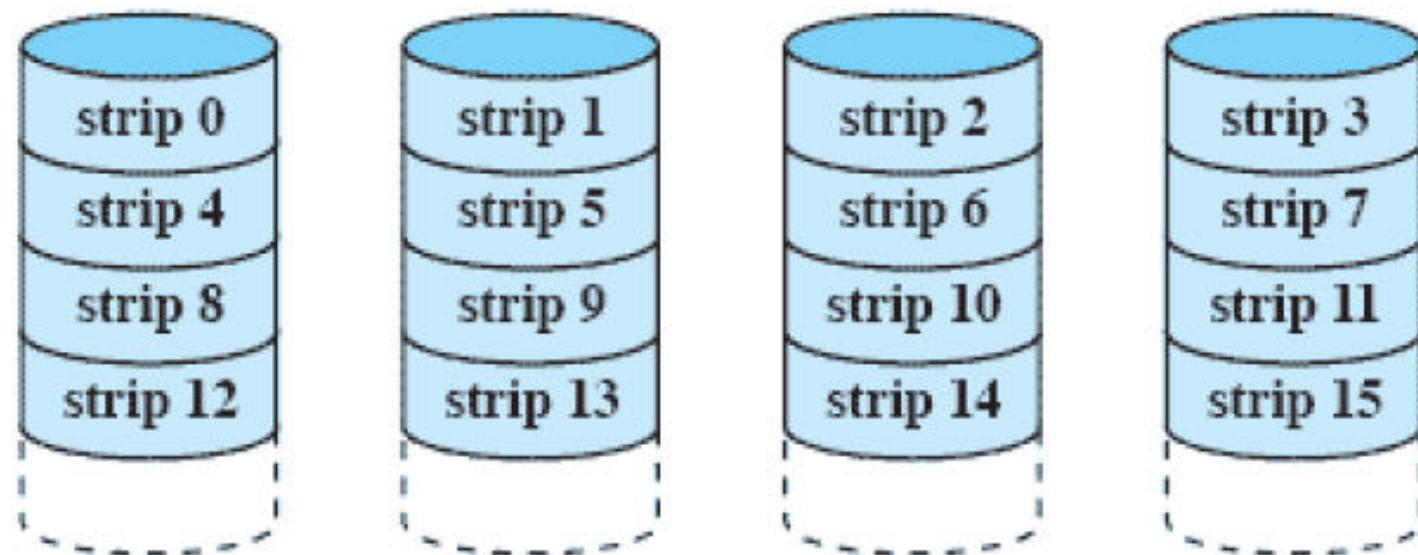
RAID

- ❖ RAID – redundant arrays of independent disks
 - ❖ multiple disk drives provides reliability via **redundancy**
- ❖ Increases the **mean time to failure**
- ❖ **Mean time to repair** – avg. time to replace a failed disk and restore the data on it
- ❖ **Mean time to data loss** based on above factors
- ❖ Mirrored Disk (volume)
- ❖ Data Stripping – bit-level stripping, byte-level stripping, block-level stripping
- ❖ RAID 0 ❖ RAID 3
- ❖ RAID 1 ❖ RAID 4
- ❖ RAID 2 ❖ RAID 5
- ❖ RAID 6

RAID 0

Level 0: Non redundant

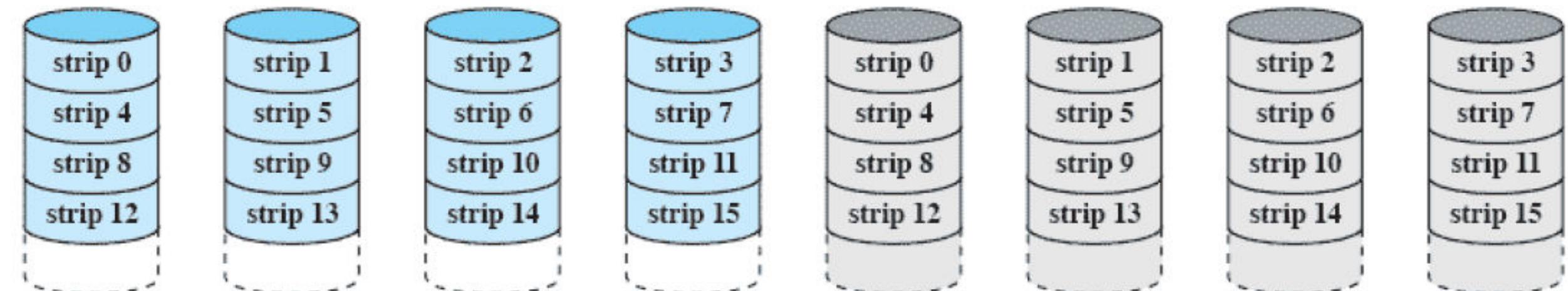
- Data striping is used for increased performance but no redundant information is maintained.
- Striping is done at block level but without any redundancy.
- Writing performance is best in this level because due to absence of redundant information there is no need to update redundant information



RAID 1

❖ Level 1: Mirrored

Same data is copied on two different disks. This type of redundancy is called **mirroring**. It is the most expensive system. Because two copies of same data are available in two different disks, it allows parallel read

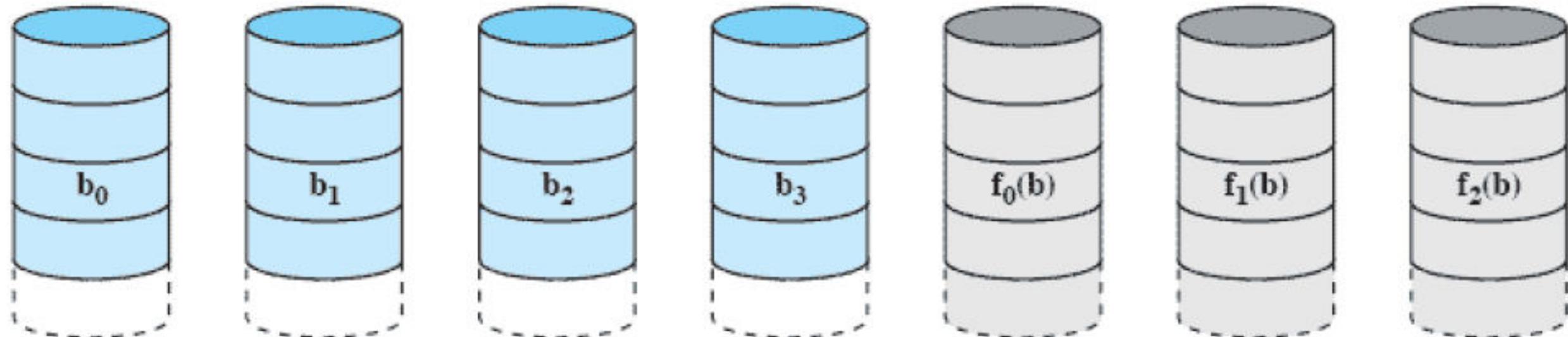


RAID 2

$2^R - 1 \geq m+k$
k = no. of drives. of data disks
block level. It is used with error-correcting codes (ECC)

❖ Level 2: Error correcting codes

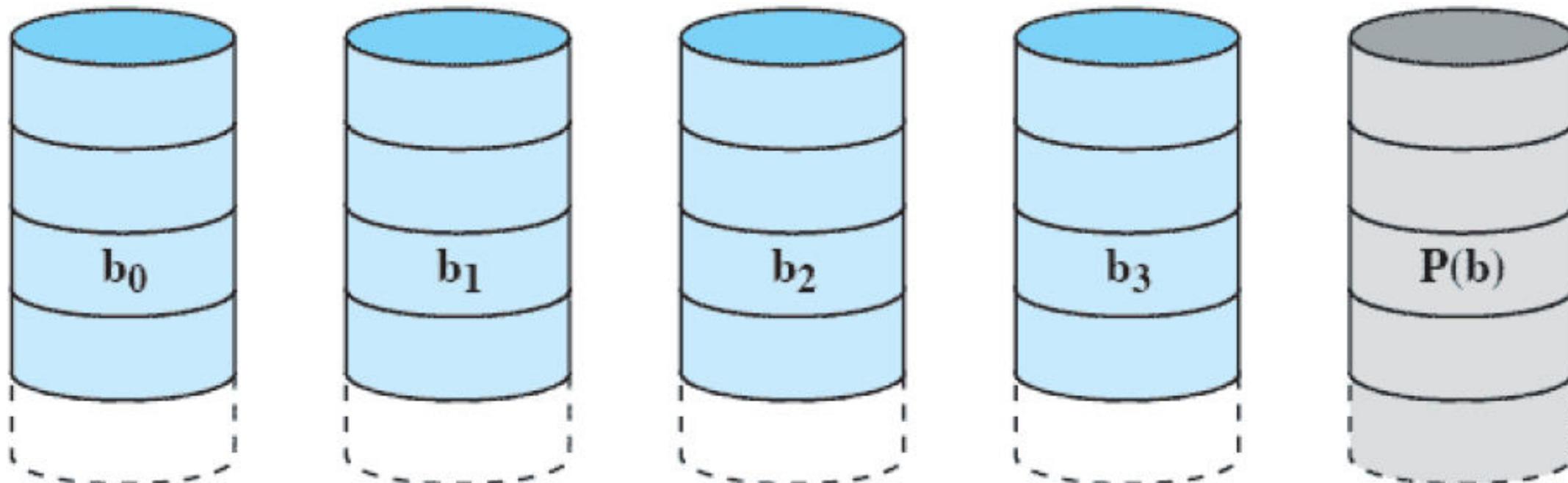
This level uses bit-level data stripping in place of block level. It is used with drives with no built in error detection technique. Error-correcting codes (ECC) store two or more extra bits and it is used for reconstruction of the data if a single bit is damaged.



RAID 3

❖ Level 3: Bit-Interleaved parity

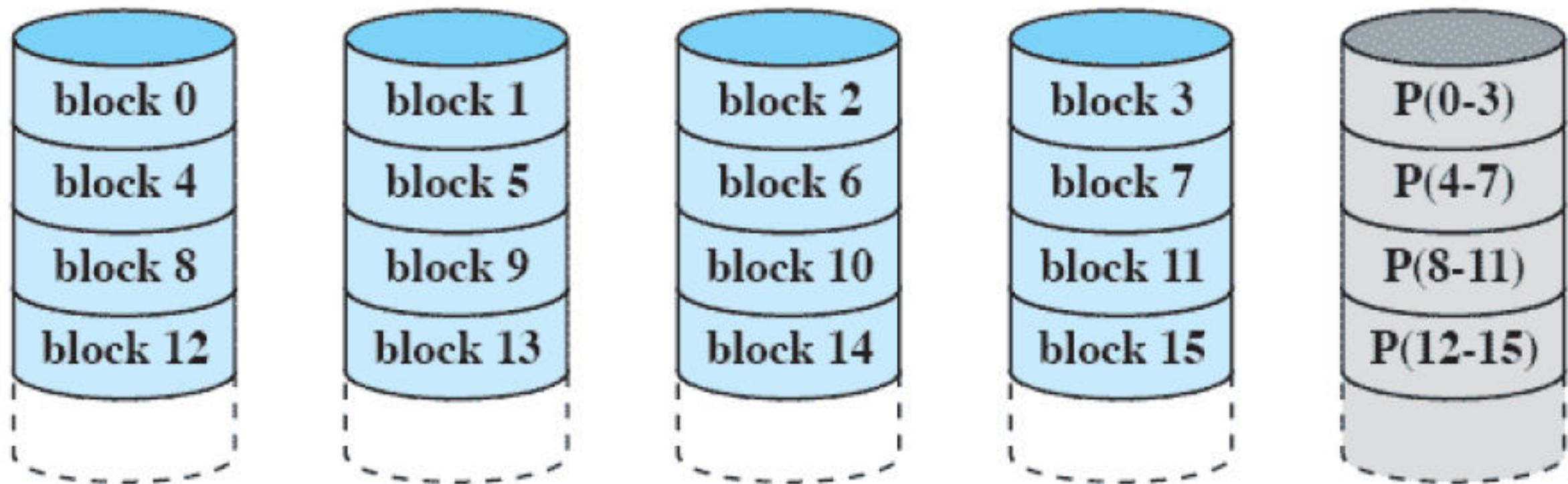
Data stripping is used and a single parity bit is used for error correction as well as for detection. Systems have disk controller that detects which disk has failed. RAID level 3 has a single check disk with parity bit.



RAID 4

❖ Level 4: Block-Interleaved parity

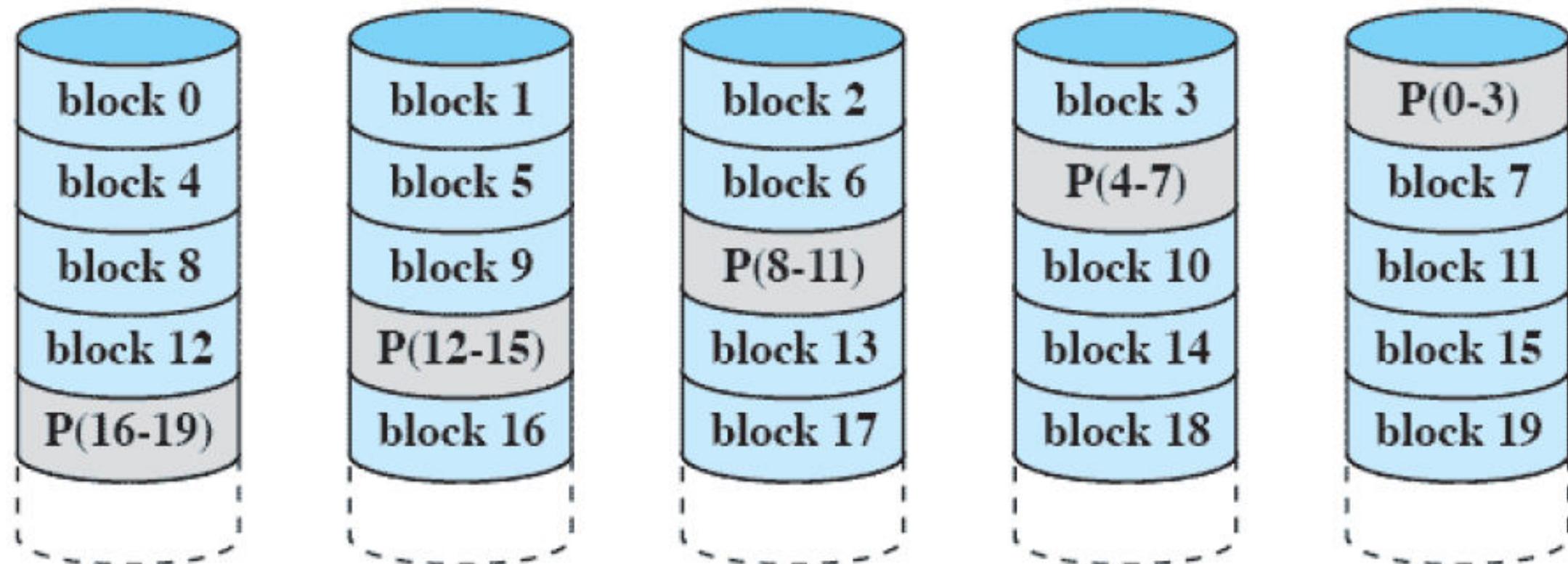
RAID level 4 is similar as RAID level 3 but it has Block-Interleaved parity instead of bit parity. You can access data independently so read performance is high.



RAID 5

❖ Level 5: Block-Interleaved distributed parity

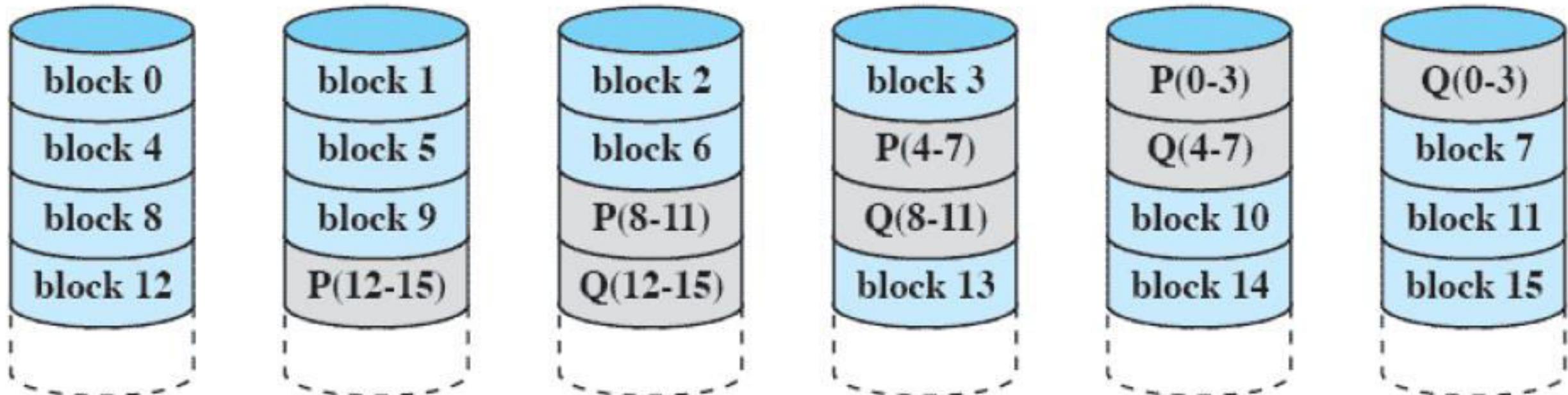
RAID level 5 distributes the parity block and data on all disks. For each block, one of the disks stores the parity and the others store data. RAID level 5 gives best performance for large read and write.



RAID 6

❖ Level 6: P+Q Redundancy Scheme

What happens if more than one disk fails at a time? This level stores extra redundant information to save the data against multiple disk failures. It uses Reed-Solomon codes (ECC) for data recovery. Two different algorithms are employed





Thank You



BITS Pilani
Hyderabad Campus

OPERATING SYSTEMS (CS F372)

File System

Dr. Barsha Mitra
CSIS Dept., BITS Pilani, Hyderabad Campus



File Attributes

- ❖ **Name** – only information kept in human-readable form
- ❖ **Identifier** – unique tag (number) identifies file within file system, non-human-readable
- ❖ **Type** – needed for systems that support different types
- ❖ **Location** – pointer to file location on device
- ❖ **Size** – current file size
- ❖ **Protection** – controls who can do reading, writing, executing
- ❖ **Time, date, and user identification** – data for protection, security, and usage monitoring
- ❖ Information about files are kept in directory structure maintained on the disk

File Operations

- ❖ File is an **abstract data type**
- ❖ **Create**
- ❖ **Write** – at **write pointer** location
- ❖ **Read** – at **read pointer** location
- ❖ **Reposition within file - seek**
- ❖ **Delete**
- ❖ **Truncate**
- ❖ OS maintains an open-file table
- ❖ For a requested file operation, file is specified via an index into the table
- ❖ When file is closed, OS removes entry from file table

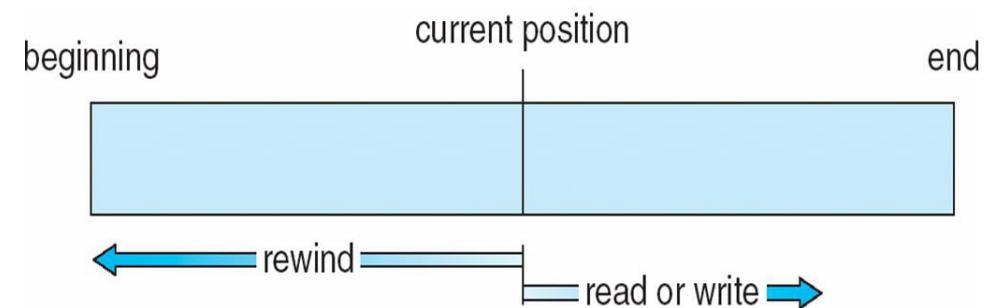
File Types

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, perl, asm	source code in various languages
batch	bat, sh	commands to the command interpreter
markup	xml, html, tex	textual data, documents
word processor	xml, rtf, docx	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	gif, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	rar, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, mp3, mp4, avi	binary file containing audio or A/V information

Access Methods

- **Sequential Access**

`read_next()`
`write_next()`
`reset()`



- **Direct Access/Relative Access** – file consists of fixed length logical records

`read(n)`
`write(n)`
`position_file(n)`

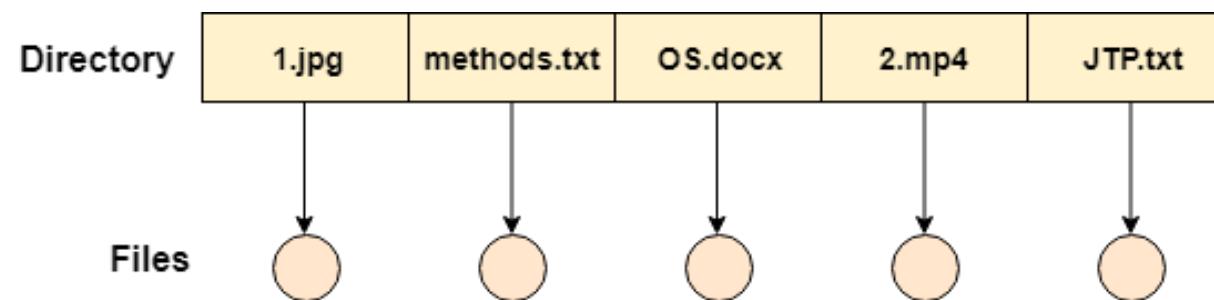
n = relative block number w.r.t beginning of file

Directory

- Any entity containing a file system is called a volume
- The directory is organized logically to obtain
 - Efficiency – locating a file quickly
 - Naming – convenient to users
 - Two users can have same name for different files
 - The same file can have several different names
 - Grouping – logical grouping of files by properties
- Directory Operations
 - Search for a file
 - Create a file
 - Delete a file
 - List a directory
 - Rename a file
 - Traverse the file system

Single-Level Directory

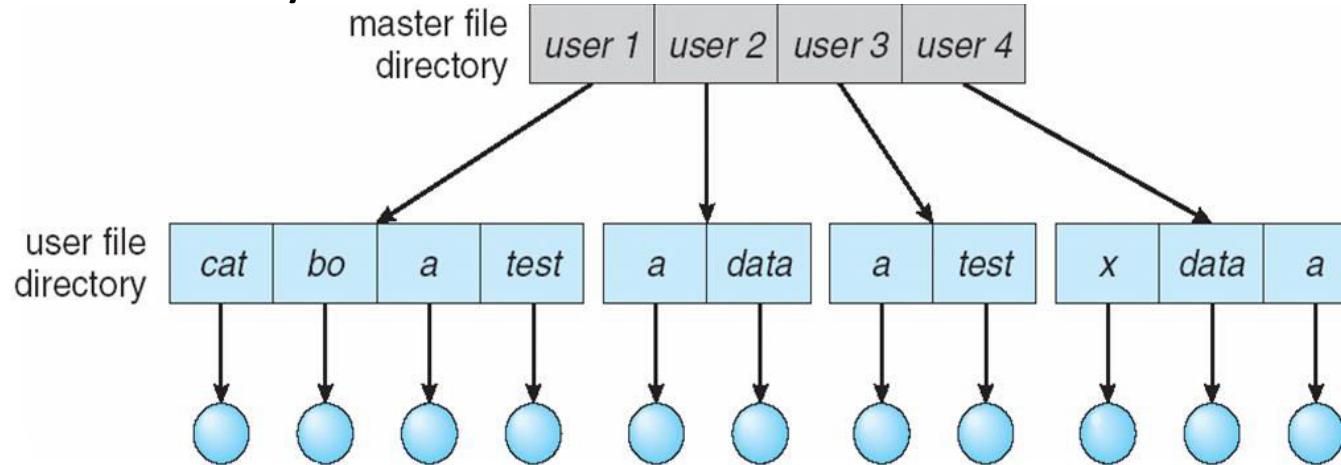
- A single directory for all users
- Entire system will contain only one directory which is supposed to mention all the files present in the file system
- Directory contains one entry for each file present on the file system



- Naming problem
- Grouping problem

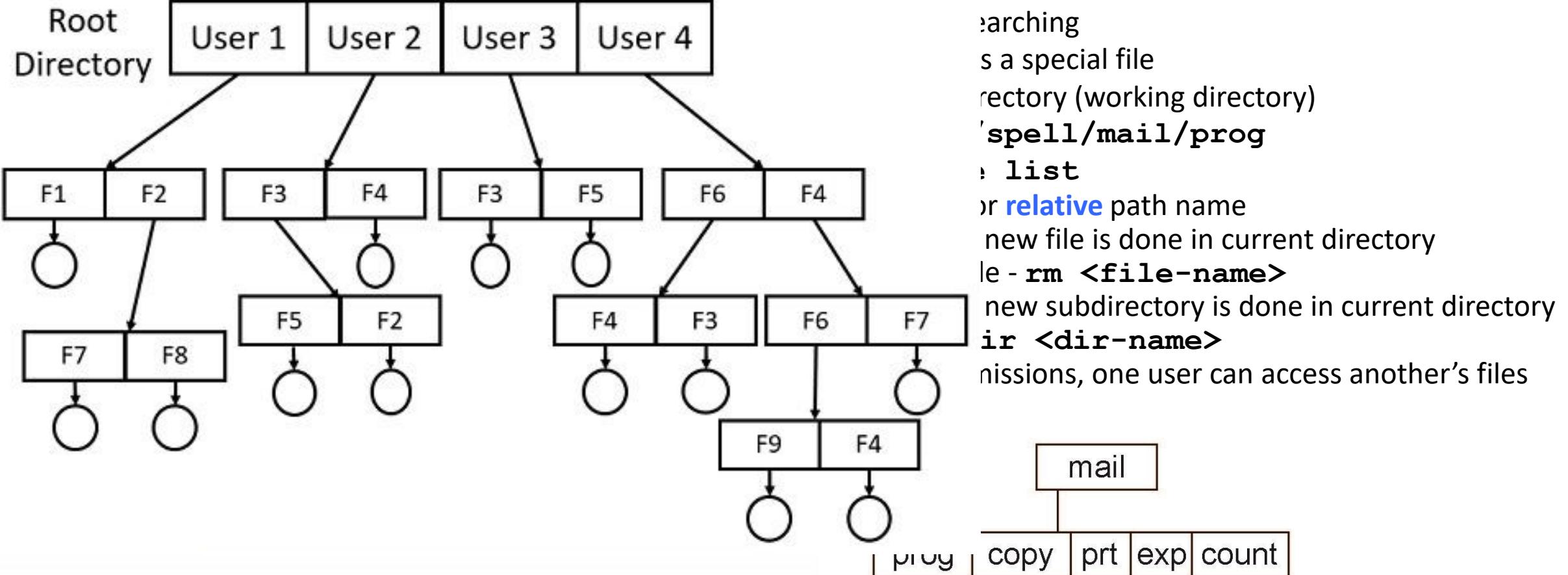
Two-Level Directory

- ❖ Separate directory for each user



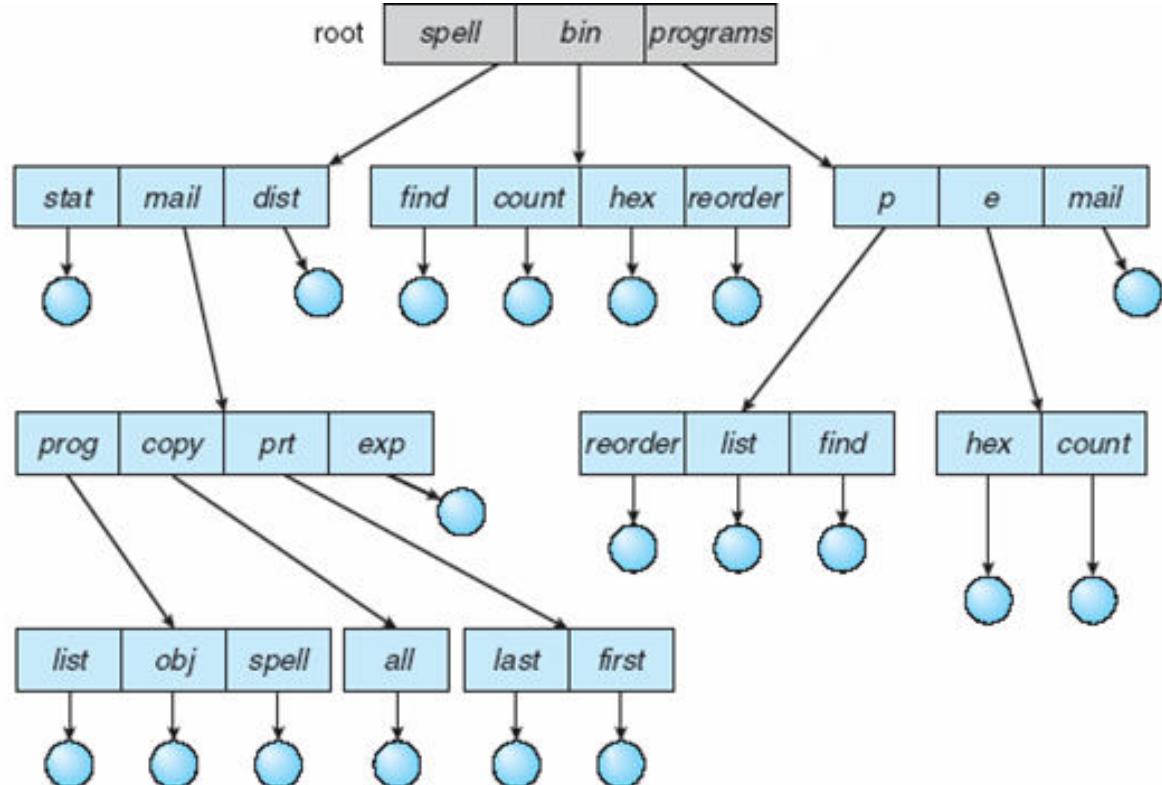
- ❖ User name and file name define a path
- ❖ MFD is indexed by user name/ account number
- ❖ Can have the same file name for different user
- ❖ Efficient searching, only UFD is searched for creation or deletion
- ❖ Creation and deletion of user directories – admin
- ❖ Sharing not possible

Tree-Structured Directory



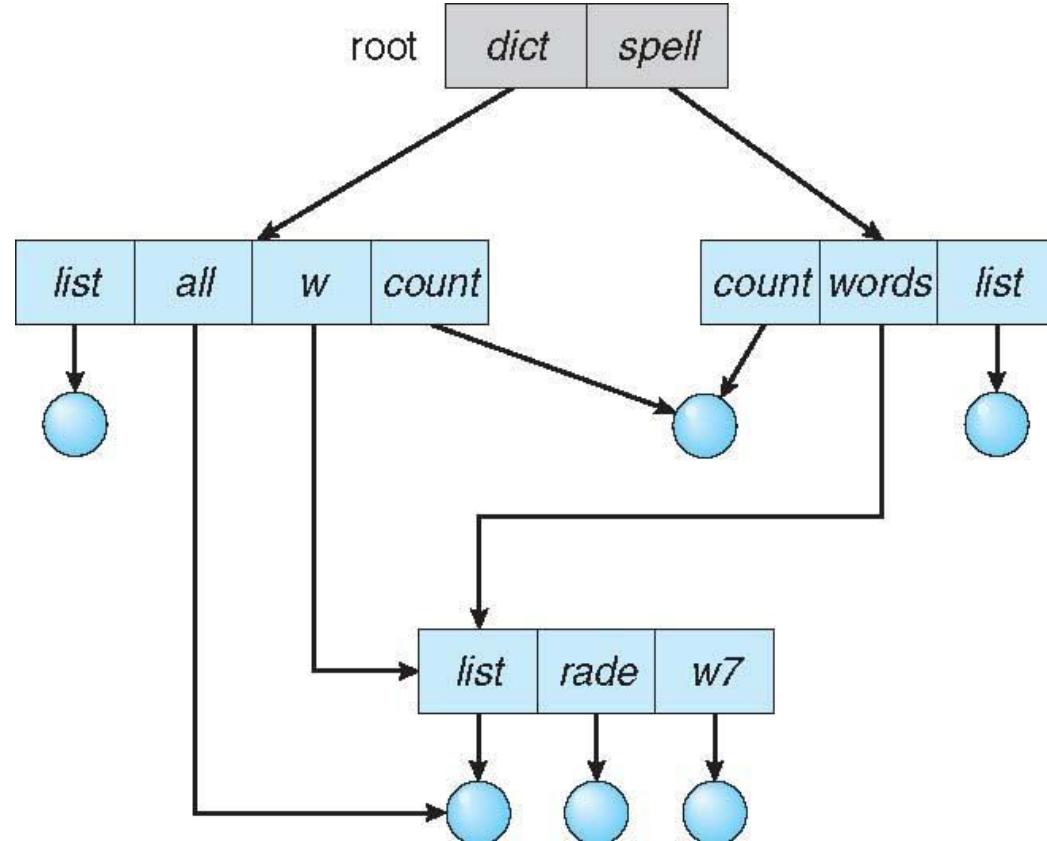
Deleting “mail” \Rightarrow deleting the entire subtree rooted by “mail”

Tree-Structured Directory



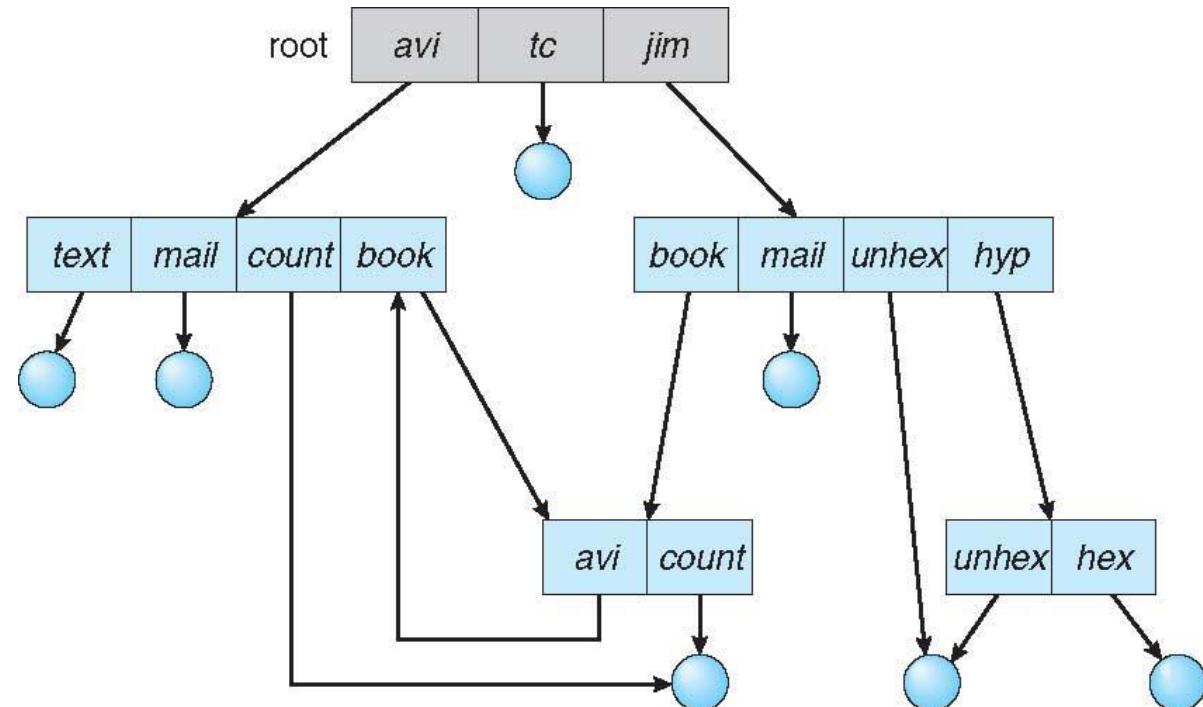
- ❖ Absolute path name – **root/spell/mail/copy/all**
- ❖ If you are inside root/spell/mail, then relative path name for all is **copy/all**

Acyclic-Graph Directory



- ❖ Have shared subdirectories and files
- ❖ Two different absolute path names
- ❖ Dangling pointer

General Graph Directory



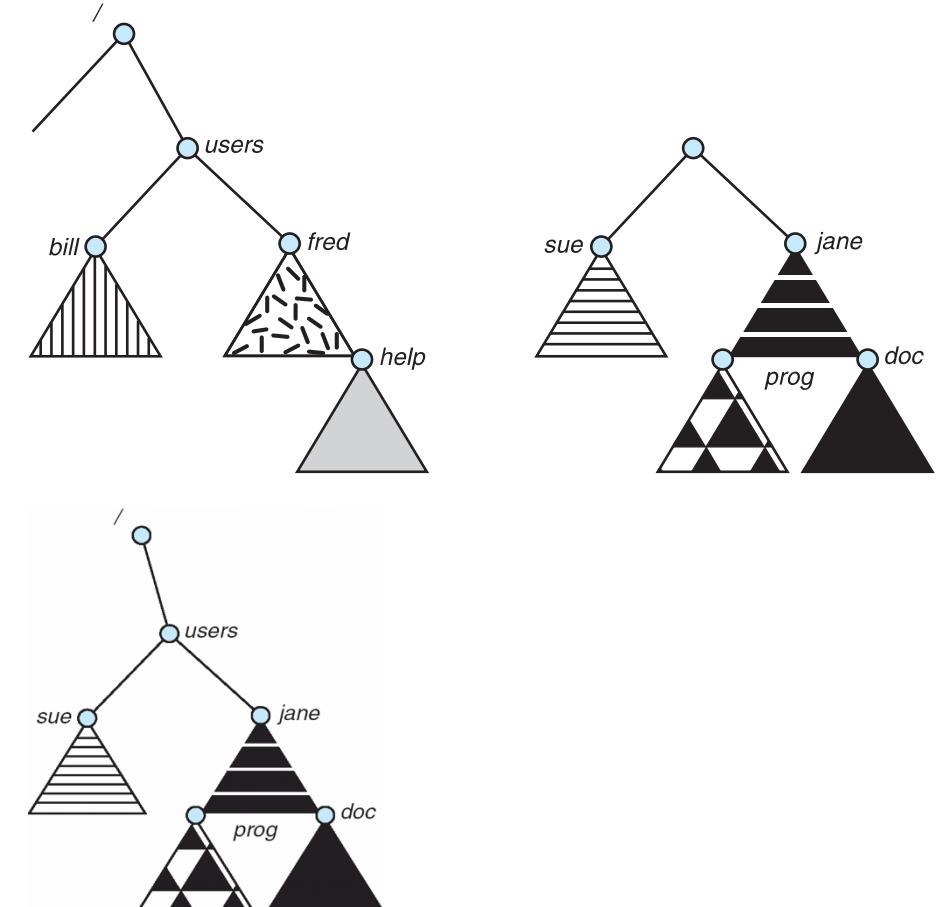
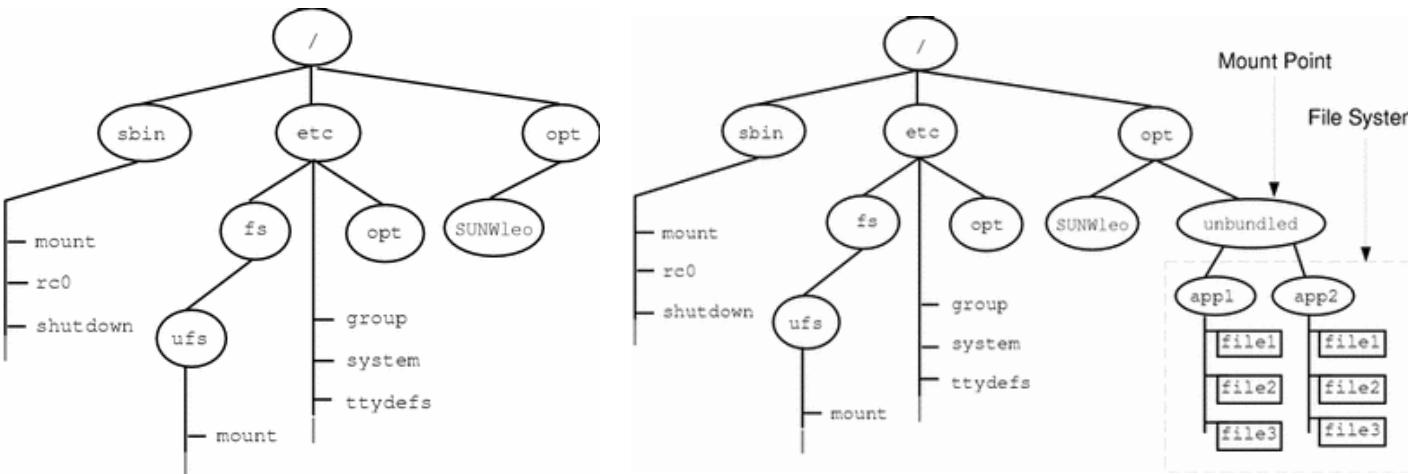
- How do we guarantee no cycles?
 - Allow only links to file not subdirectories
 - **Garbage collection - mechanism to determine when the last of the references of a file has been deleted and the disk space can be reallocated**
 - Every time a new link is added use a cycle detection algorithm to determine whether it is OK

File System Mounting

A file system must be **mounted** before it can be accessed

Requires the device name(pendrive,cd) and mount point (location within the file structure where the file system is to be attached)

An unmounted file system is mounted at a **mount point**





Thank You



BITS Pilani
Hyderabad Campus

OPERATING SYSTEMS (CS F372)

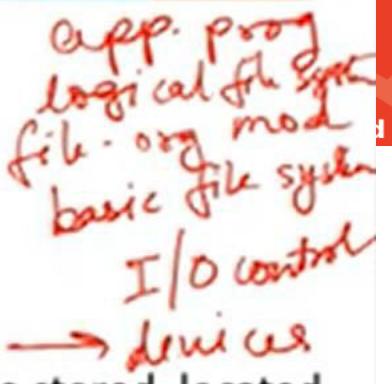
File System Implementation

Dr. Barsha Mitra
CSIS Dept., BITS Pilani, Hyderabad Campus



File-System Structure

- File structure
 - Logical storage unit
 - Collection of related information
- **File system** resides on secondary storage (disks)
 - 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** – storage structure consisting of information about a file,in unix it is inode
- **Device driver** controls the physical device
- File system organized into layers



File System Layers

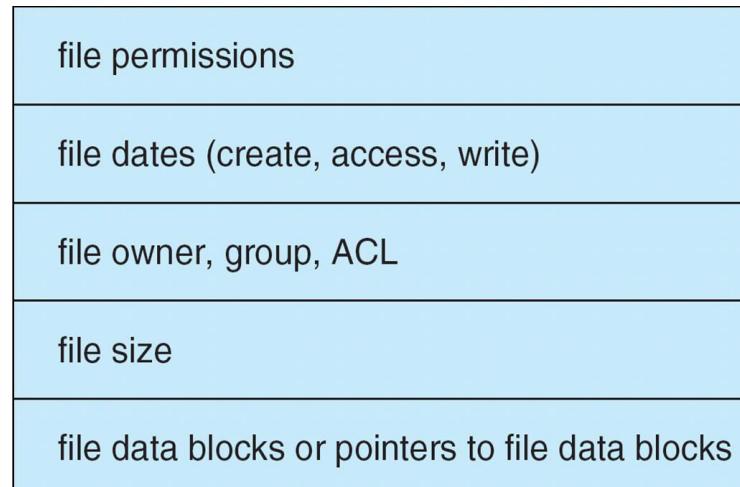
- **Device drivers** manage I/O devices at the I/O control layer
- **Basic file system** – gives generic commands to device driver to read and write physical disk blocks
 - Also manages memory buffers and caches (allocation, freeing, replacement)
 - Buffers hold data in transit
 - Caches hold frequently used file system metadata
- **File organization module** understands files, logical blocks, and physical blocks, maps logical block address to physical block address
- **Logical file system** manages metadata information
 - Translates file name into file no., file handle, location by maintaining fcb/file control blocks
 - Directory management
 - Protection

File System Implementation

- **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/partition details
 - Total # of blocks, # of free blocks, block size, free block pointers or array
 - Directory structure organizes the files

File System Implementation

- Per-file **File Control Block (FCB)** contains many details about the file
 - unique identifier to allow association with a directory entry, permissions, size, dates

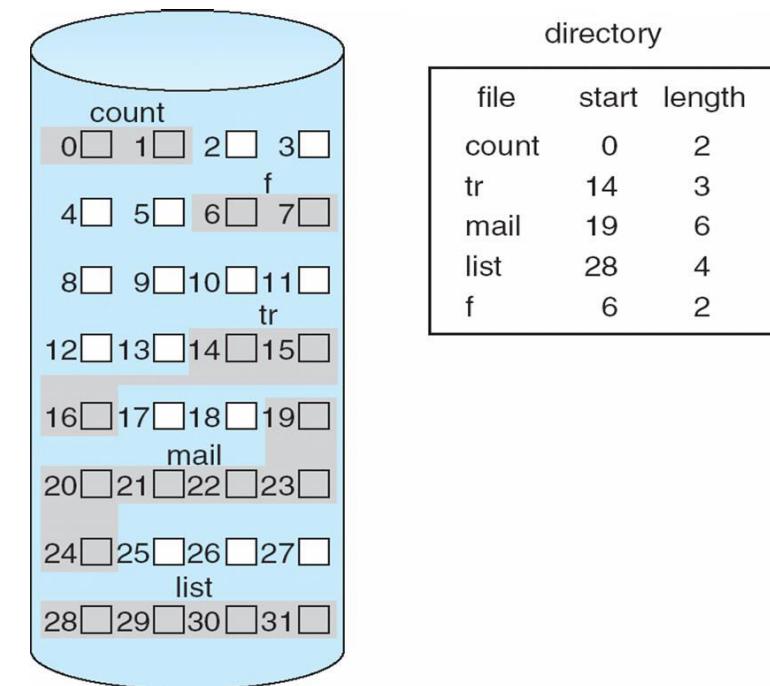


Directory Implementation

- **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 (for directory entries) 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

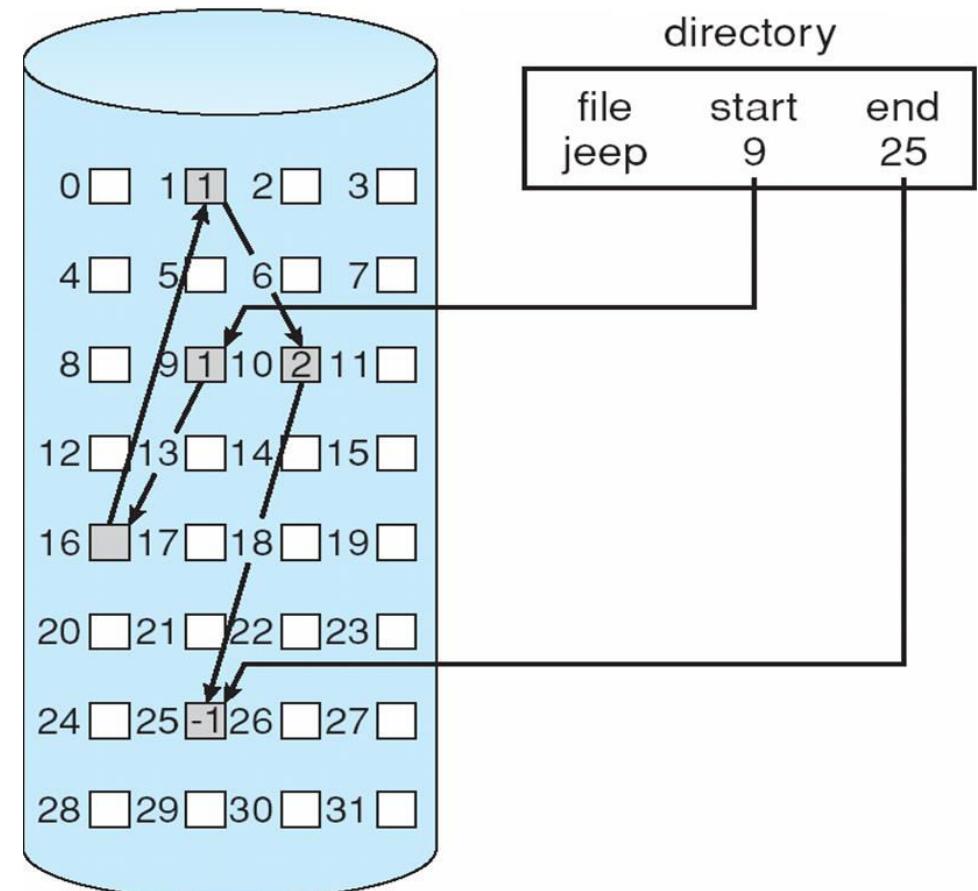
- Allocation method refers to how disk blocks are allocated for files
- Contiguous allocation** – each file occupies set of contiguous blocks
 - Best performance in most cases
 - Simple – only starting location (block #) and length (number of blocks) are required
 - Problems include finding space for file, knowing file size, external fragmentation, need for **compaction off-line (downtime)** or **on-line**
- Mapping from logical to physical



Allocation Methods - Linked

- **Linked allocation** – each file a linked list of blocks

- Space wastage as next-file pointer also stored for every file/cluster of files
- File ends at nil pointer
- Each block contains pointer to next block
- No compaction, no external fragmentation
- Free space management system called when new block needed
- Improve efficiency by clustering blocks into groups
- Reliability can be a problem
- Locating a block can take many I/Os and disk seeks



Allocation Methods - File Allocation Table



FAT (File Allocation Table)

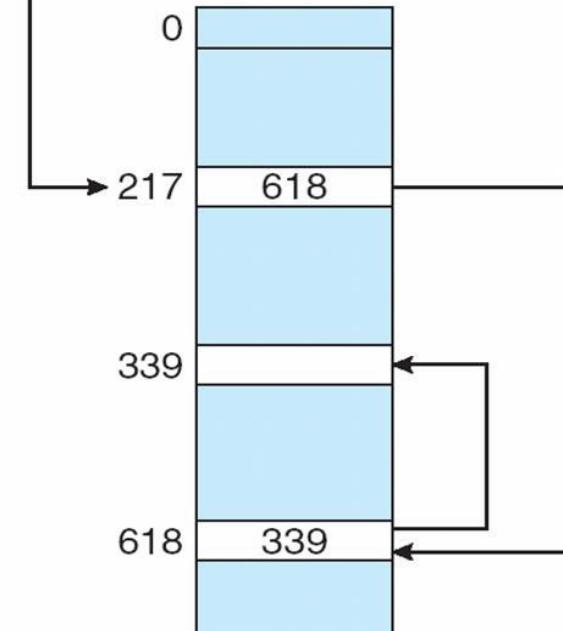
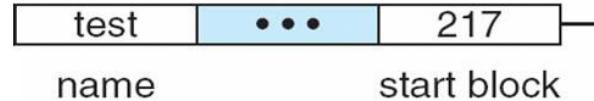
Beginning of volume has table, indexed by block number

Much like a linked list allocation but without the overhead of pointers

Next Block-number stored instead of pointer

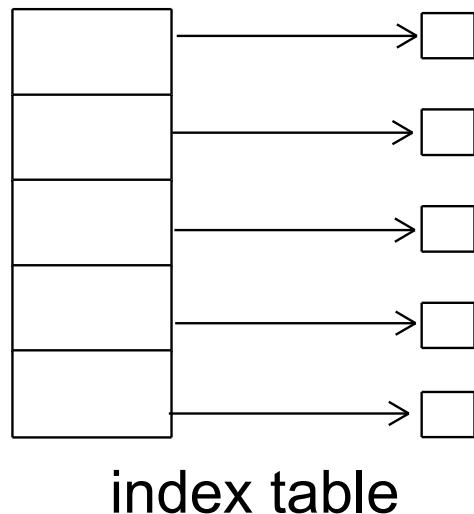
New block allocation simple

directory entry



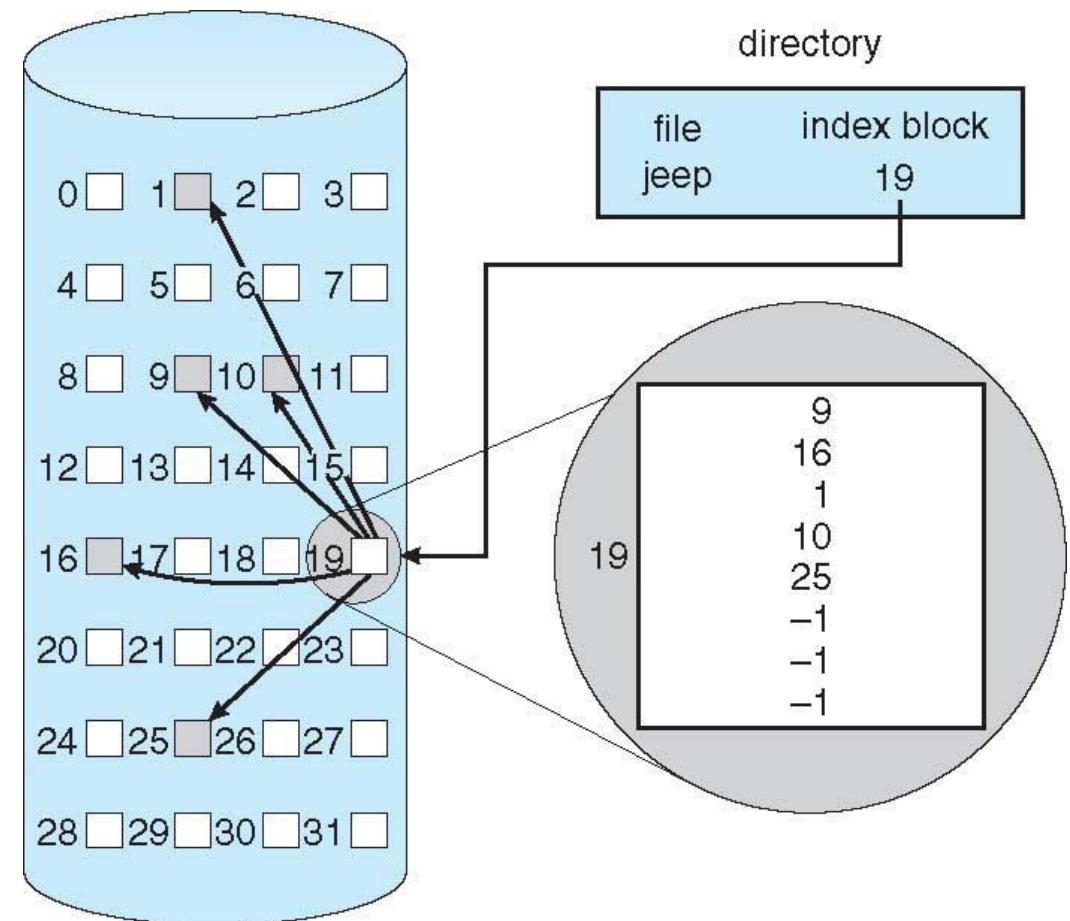
Allocation Methods - Indexed

- ❖ **Indexed allocation** - Each file has its own **index block** of pointers to its data blocks



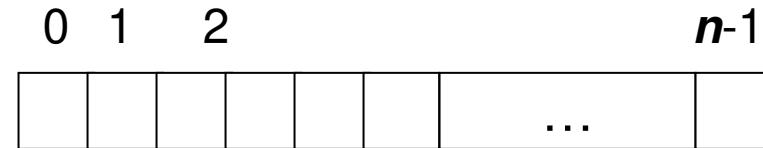
Cluster pointers into single block called index block

- ❖ Need index table
- ❖ Random access
- ❖ Dynamic access without external fragmentation, but have overhead of index block



Free Space Management (bitmap & linked list)

- ❖ File system maintains **free-space list** to track available blocks/clusters
 - ❖ (Using term “block” for simplicity)
 - ❖ **Bit vector** or **bit map** (n blocks)

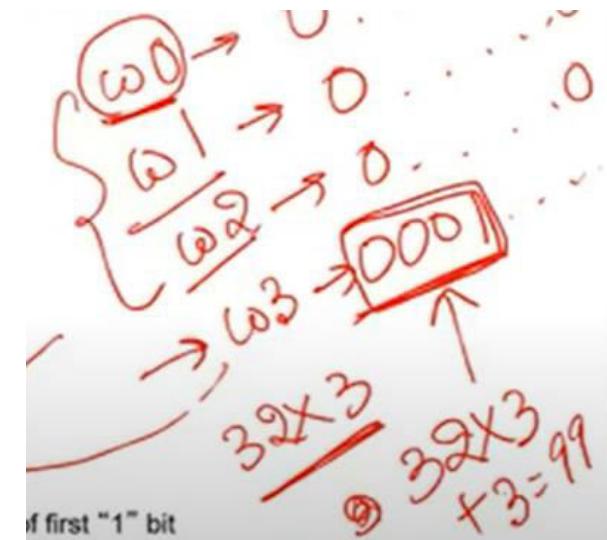


$$\text{bit}[i] = \begin{cases} 1 & \Rightarrow \text{block}[i] \text{ free} \\ 0 & \Rightarrow \text{block}[i] \text{ occupied} \end{cases}$$

First free Block number calculation

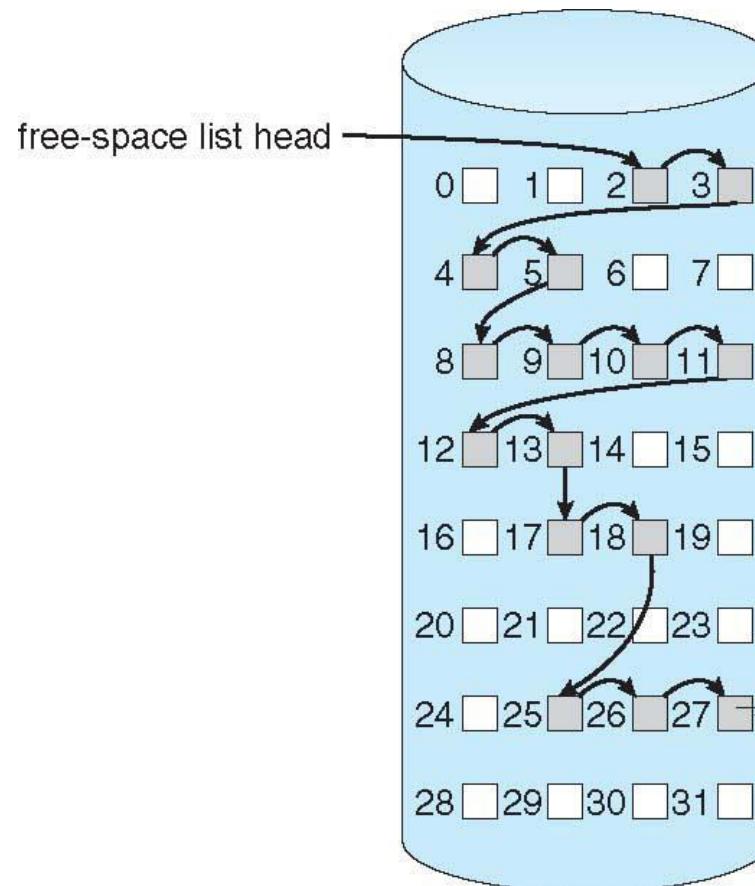
(number of bits per word) *
(number of 0-value words) +
offset of first 1 bit

CPUs have instructions to return offset within word of first “1” bit



Linked List

- ❖
- ❖ Linked list (free list of free blocks)
 - ❖ Cannot get contiguous space easily
 - ❖ No waste of space
 - ❖ No need to traverse the entire list (if # free blocks recorded)
 - ❖ Some space wasted in each block (to hold pointer to next free block)



Free Space Management

IMPROVEMENT IN LINKED LIST APPROACH:

- ❖ **Grouping**
 - ❖ 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**
 - ❖ Because space is frequently contiguously used and freed
 - ❖ Keep address of first free block and count of following free blocks
 - ❖ Free space list then has entries containing addresses and counts



Thank You



BITS Pilani
Hyderabad Campus

OPERATING SYSTEMS (CS F372)

I/O Systems

Dr. Barsha Mitra
CSIS Dept., BITS Pilani, Hyderabad Campus



Overview

- I/O management is a major component
- Ports, buses, device controllers connect to various devices
- **Device drivers** encapsulate device details
- Common concepts – signals from I/O devices interface with computer
 - **Port** – connection point for device
 - **Bus - daisy chain(Device a connected to b,b to c ,c to comp.)** or shared direct access
 - **PCI** bus common in PCs
 - **expansion bus** connects relatively slow devices
 - SCSI BUS
 - **Controller (host adapter)** – electronics that operate port, bus, device
 - Sometimes integrated on a chip
 - Sometimes separate circuit board plugging into the computer, contains processor, microcode, private memory

I/O Hardware

- I/O instructions control devices
 - Controllers usually have registers where device driver places processor commands, addresses, and data to write, or read data from registers after command execution
 - **Data-in register** – read by host/CPU to get i/p
 - **Data-out register** – written by host to send o/p
 - **Status register** – contains bits that can be read by host, bits indicate states like completion of current command, availability of byte to be read from data-in register or occurrence of device error
 - **Control register** – written by host to start a command
-

IO protocol Polling

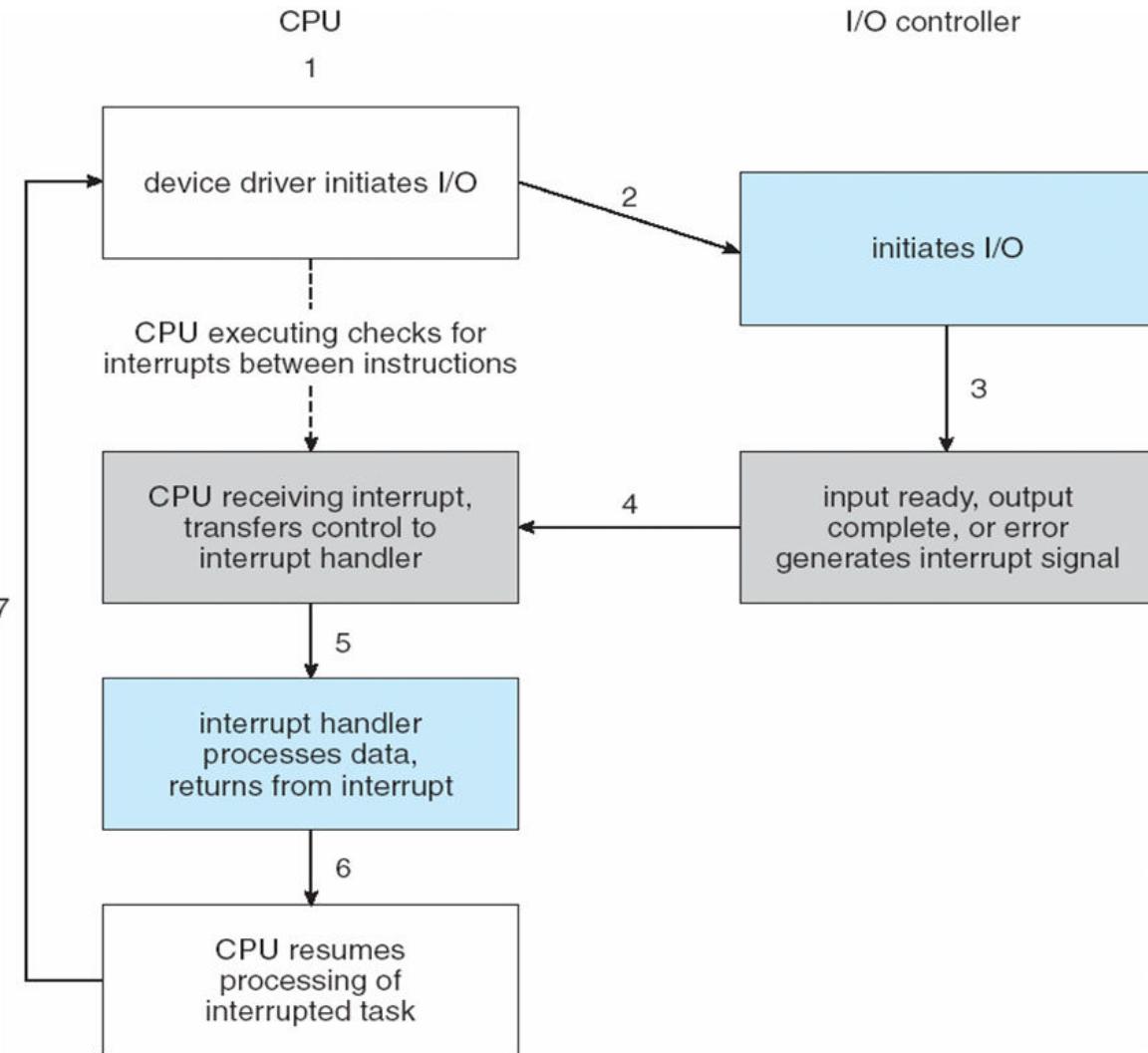
1. Host reads busy bit from status register until made 0 by controller
 2. Host sets read or write bit in command register and if write bit is set, writes a data byte into data-out register
 3. Host sets command-ready bit in command register
 4. Controller sees command-ready bit and sets busy bit
 5. Controller sees write command in command register, reads byte from data-out register and complete I/O
 6. Controller clears busy bit, error bit (status reg.), clears command-ready bit when transfer done
- Step 1 is **busy-waiting or polling**
 - Reasonable if device is fast, but inefficient if device slow;cpu cycle wasted in polling
 - CPU switches to other tasks?
 - How does CPU know controller is idle, controller buffer may overflow;solution to this == interrupt

Interrupts

Interrupt driven IO:

- CPU **Interrupt-request line** triggered by I/O device
 - Checked by processor after each instruction
 - 2 types of interrupts- maskable(ignorable) & non-maskable(very imp./non-ignorable by cpu)
- **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
 - Some **nonmaskable ;ex-unrecoverable memory errors)**

Interrupts



- Interrupt mechanism also used for **exceptions**
 - Terminate process, system crash due to hardware error
- Page fault executes when memory access error
- System call executes via **trap** to trigger kernel to execute request

Direct Memory Access (IO-method)



- Used for large data movement
- Requires **DMA** controller (special-purpose processor)
- Bypasses CPU to transfer data directly between I/O device and memory
- OS writes DMA command block(a datastructure) into memory ;dma command block contains:
 - Source and destination addresses
 - Read or write mode
 - Count of bytes
- CPU writes location of command block to DMA controller
- DMA performs transfer without help of CPU

Direct Memory Access

- Handshaking b/w DMA controller and device controller is done via DMA-request and DMA-acknowledge wires
- Device controller places a signal on the DMA-request wire when a data word is available for transfer
- DMA controller takes control of memory bus, places the desired address on memory-address wires and places a signal on DMA-acknowledge wire
- Device controller on receiving DMA-acknowledge signal, transfers data word to memory and removes DMA-request signal
- When transfer is complete, DMA controller interrupts CPU
- Cycle stealing(for some cycle ,cpu might not be able to access memory bus as dma is using it)

I/O Interface

- I/O system calls encapsulate device behaviors in generic classes
- Device-driver layer hides differences among I/O controllers from I/O subsystem of kernel
- Devices vary in many dimensions
 - **Character-stream**(transfer 1 byte of data at a time like monitor) or **block**(hard disk)
 - **Sequential**(modem) or **random-access**(cd,bluray disk)
 - **Synchronous** or **asynchronous** (or both)
 - **Sharable** or **dedicated**
 - **Speed of operation**
 - **read-write, read only, or write only**

I/O Interface

Block & Character Devices

- Block devices include disk drives
 - Commands include read, write, seek
- Character devices include keyboards, mice, modems, printers
 - Commands include `get()`, `put()`

Types of I/O

- Blocking
- Nonblocking – input from keyboard and mouse
- Asynchronous – disk and n/w I/O

Other Aspects

- **Error Handling**
 - OS can recover from disk read, device unavailable, transient write failures
 - Retry a read or write, for example
 - 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



Thank You



BITS Pilani
Hyderabad Campus

OPERATING SYSTEMS (CS F372)

Protection

Dr. Barsha Mitra
CSIS Dept., BITS Pilani, Hyderabad Campus



Goals of Protection

- Computer consists of a collection of objects, hardware or software
- Each object can be accessed through a well-defined set of operations
- **Protection problem** - ensure that each object is accessed correctly and only by those processes that are allowed to do so
- policy and mechanism

Principles of Protection

- Guiding principle – **principle of least privilege**
 - Programs, users and systems should be given just enough **privileges** to perform their tasks
 - Limits damage if entity has a bug, gets abused
 - Can be static (during life of system, during life of process)
 - Or dynamic (changed by process as needed) – **privilege escalation**
 - “Need to know” a similar concept regarding access to data

Access Matrix

- View protection as a matrix (**access matrix**)
- Rows represent domains
- Columns represent objects
- **Access** (i, j) is the set of operations that a process executing in Domain_i can invoke on Object_j

domain \ object	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	



Thank You