

# OPERATING SYSTEMS

## Lecture Notes



**VISHNU**  
UNIVERSAL LEARNING

**DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING  
SHRI VISHNU ENGINEERING COLLEGE FOR WOMEN**  
(Approved by AICTE, Accredited by NBA, Affiliated to JNTU Kakinada)  
**BHIMAVARAM – 534 202**



**UNIT -1**  
**COMPUTER SYSTEM AND OPERATING SYSTEM OVERVIEW**

### **OVER VIEW OF OPERATING SYSTEM**

#### **What is an Operating System?**

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

Operating system goals:

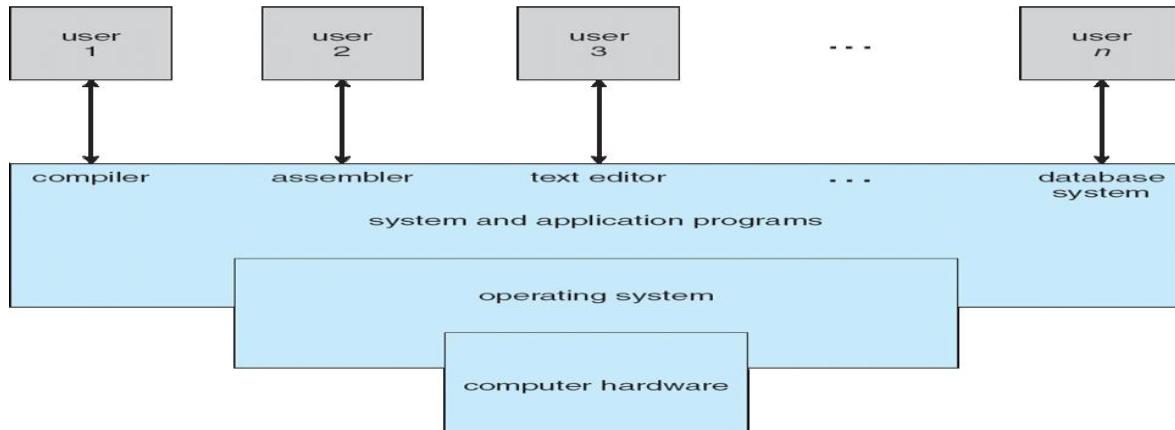
- Execute user programs and make solving user problems easier
- Make the computer system convenient to use
- Use the computer hardware in an efficient manner

### **Computer System Structure**

Computer system can be divided into four components

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

### **Four Components of a Computer System**



### **Operating System Definition**

- OS is a **resource allocator**
- Manages all resources
- Decides between conflicting requests for efficient and fair resource use
- OS is a **control program**
- Controls execution of programs to prevent errors and improper use of the computer
- No universally accepted definition
- Everything a vendor ships when you order an operating system" is good approximation
  - But varies wildly

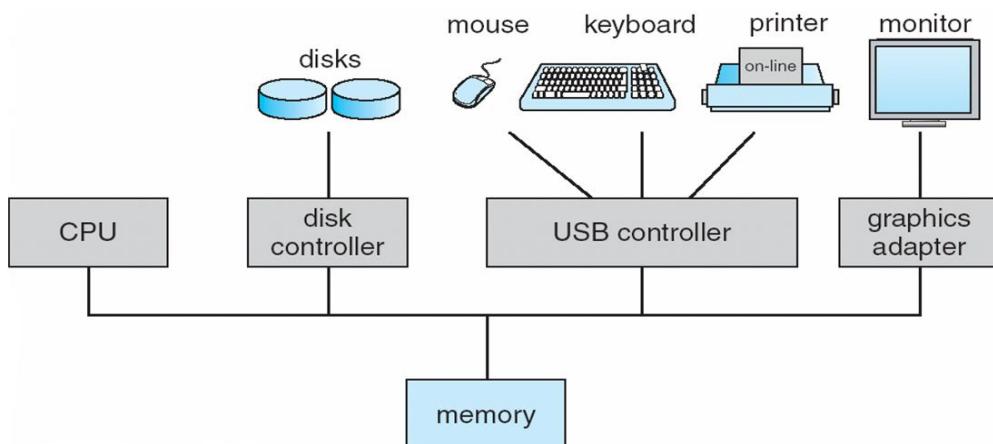
- “The one program running at all times on the computer” is the **kernel**. Everything else is either a system program (ships with the operating system) or an application program

## Computer Startup

- **bootstrap program** is loaded at power-up or reboot
- Typically stored in ROM or EPROM, generally known as **firmware**
- Initializes all aspects of system
- Loads operating system kernel and starts execution

## 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



## 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
- Device controller informs CPU that it has finished its operation by causing An *interrupt*

## Common Functions of Interrupts

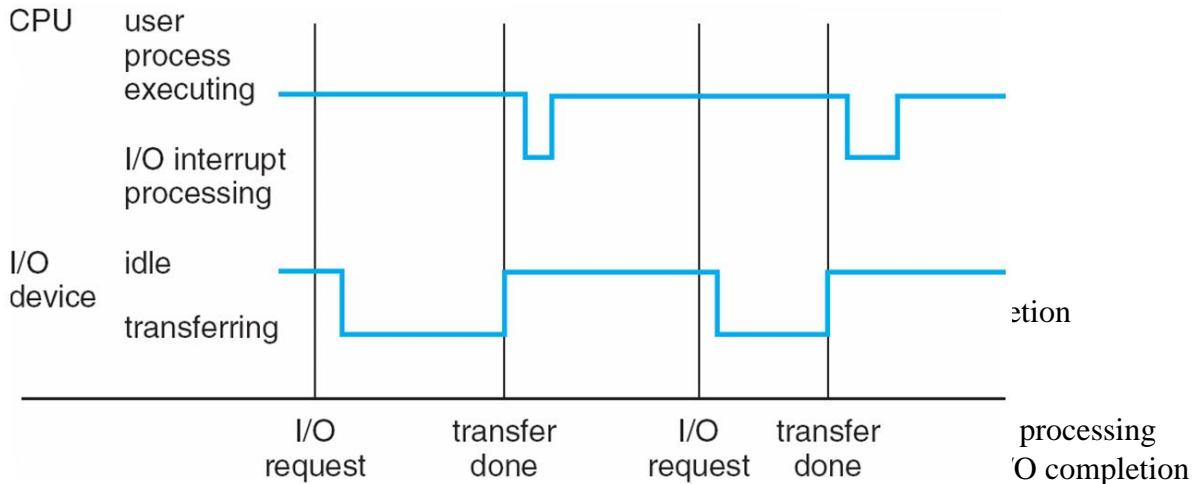
- Interrupt transfers control to the interrupt service routine generally, through the **interrupt vector**, which contains the addresses of all the service routines
- Interrupt architecture must save the address of the interrupted instruction
- Incoming interrupts are *disabled* while another interrupt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

## Interrupt Handling

- The operating system preserves the state of the CPU by storing registers and the program counter
- Determines which type of interrupt has occurred:
- **polling**

- **vectored** interrupt system
- Separate segments of code determine what action should be taken for each type of interrupt

### Interrupt Timeline



- **System call** – request to the operating system to allow user to wait for I/O completion
- **Device-status table** contains entry for each I/O device indicating its type, address, and **state**
- Operating system indexes into I/O device table to determine device status and to modify table entry to include interrupt

### Direct Memory Access Structure

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

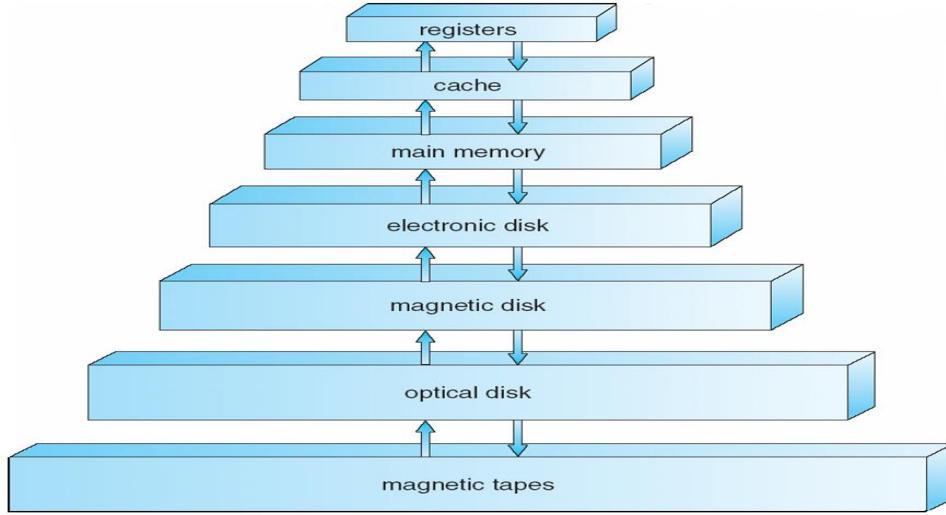
### Storage Structure

- Main memory – only large storage media that the CPU can access directly
- Secondary storage – extension of main memory that provides large nonvolatile storage capacity
- Magnetic disks – rigid metal or glass platters covered with magnetic recording material
- Disk surface is logically divided into **tracks**, which are subdivided into **sectors**
- The **disk controller** determines the logical interaction between the device and the computer

### Storage Hierarchy

- Storage systems organized in hierarchy
- Speed
- Cost
- Volatility

**Caching** – copying information into faster storage system; main memory can be viewed as a last *cache* for secondary storage



### Caching

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

### Computer-System Architecture

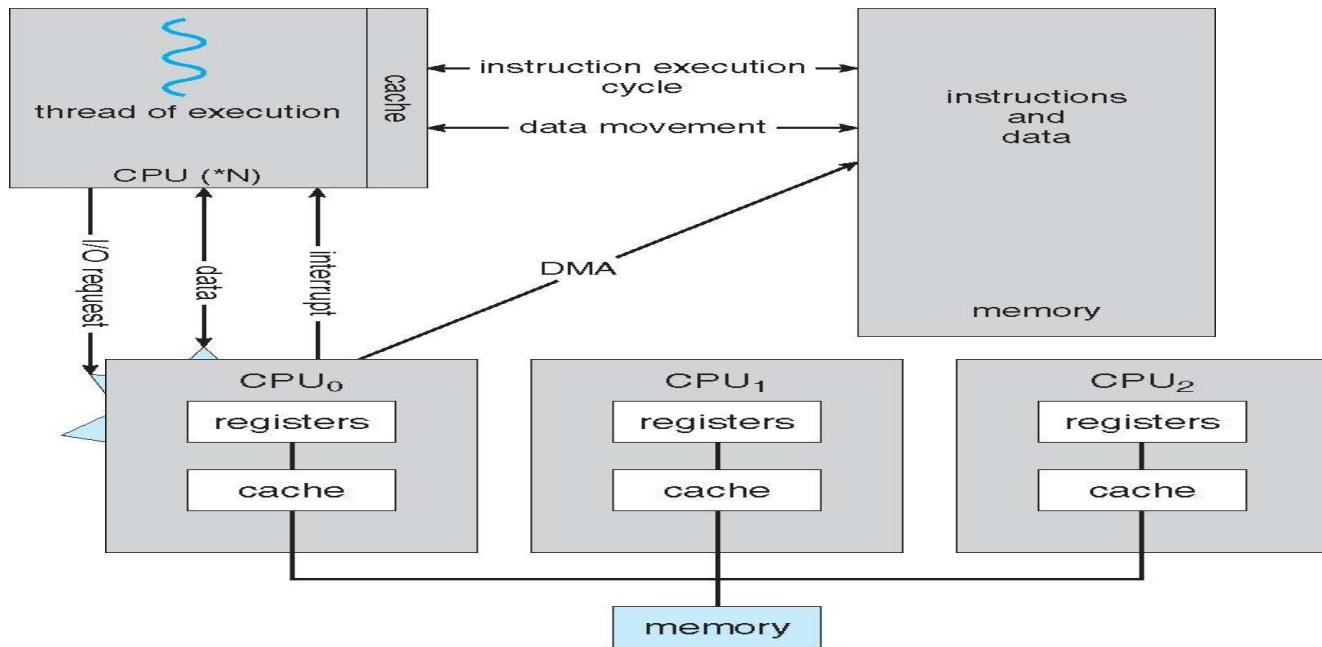
- Most systems use a single general-purpose processor (PDAs through mainframes)
- Most systems have special-purpose processors as well
- Multiprocessor systems growing in use and importance
- Also known as parallel systems, tightly-coupled systems

Advantages include

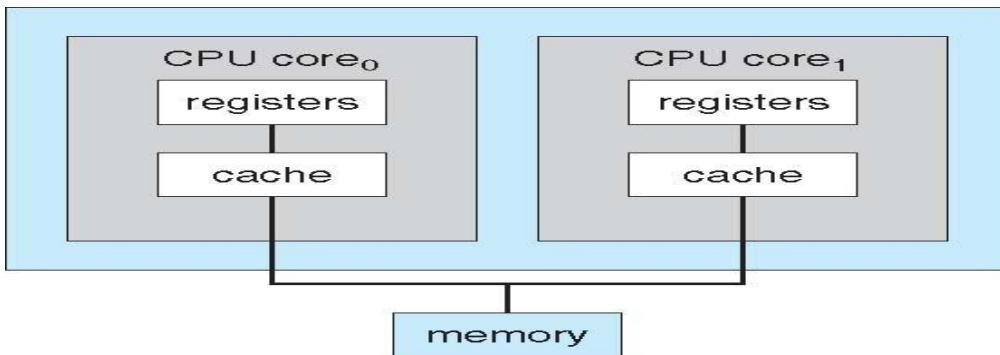
- 1.Increased throughput
- 2.Economy of scale
- 3.Increased reliability – graceful degradation or fault tolerance

Two types

- 1.Asymmetric Multiprocessing
- 2.Symmetric Multiprocessing



### A Dual-Core Design



### Clustered Systems

- Like multiprocessor systems, but multiple systems working together
- Usually sharing storage via a storage-area network (SAN)
- Provides a high-availability service which survives failures
  - Asymmetric clustering has one machine in hot-standby mode
  - Symmetric clustering has multiple nodes running applications, monitoring each other
- Some clusters are for high-performance computing (HPC)
  - Applications must be written to use parallelization

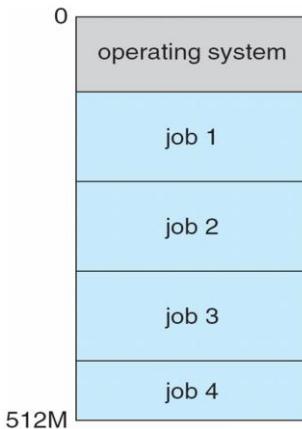
### Operating System Structure

- **Multiprogramming** needed for efficiency
- Single user cannot keep CPU and I/O devices busy at all times
- Multiprogramming organizes jobs (code and data) so CPU always has one to Execute
- A subset of total jobs in system is kept in memory

- One job selected and run via **job scheduling**
- When it has to wait (for I/O for example), OS switches to another job
- **Timesharing (multitasking)** is logical extension in which CPU switches jobs so frequently that users can interact with each job while it is running, creating **interactive** computing
- **Response time** should be < 1 second
- Each user has at least one program executing in memory [**process**]
- If several jobs ready to run at the same time [ **CPU scheduling**]
- If processes don't fit in memory, **swapping** moves them in and out to run

**Virtual memory** allows execution of processes not completely in memory

### Memory Layout for Multiprogrammed System



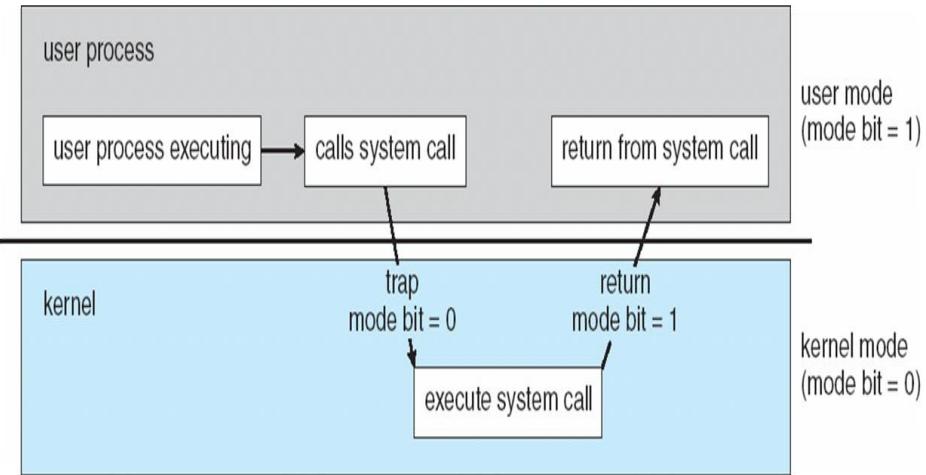
### Operating-System Operations

- Interrupt driven by hardware
- Software error or request creates **exception or trap**
- Division by zero, request for operating system service
- Other process problems include infinite loop, processes modifying each Other or the operating system
- **Dual-mode** operation allows OS to protect itself and other system components
- **User mode and kernel mode**
- **Mode bit** provided by hardware

- Provides ability to distinguish when system is running user code or kernel code
- Some instructions designated as **privileged**, only executable in kernel mode
- System call changes mode to kernel, return from call resets it to user

### Transition from User to Kernel Mode

- Timer to prevent infinite loop / process hogging resources
- Set interrupt after specific period
- Operating system decrements counter
- When counter zero generate an interrupt
- Set up before scheduling process to regain control or terminate program that exceeds allotted time



## OPERATING SYSTEM FUNCTIONS

### Process Management

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

### Process Management Activities

- The operating system is responsible for the following activities in connection with process management:
- Creating and deleting both user and system processes
- Suspending and resuming processes
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication
- Providing mechanisms for deadlock handling

### Memory Management

- All data in memory before and after processing
- All instructions in memory in order to execute
- Memory management determines what is in memory when
- Optimizing CPU utilization and computer response to users

- **Memory management activities**
- Keeping track of which parts of memory are currently being used and by whom
- Deciding which processes (or parts thereof) and data to move into and out of memory
- Allocating and deallocating memory space as needed

### **Storage Management**

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

### **Mass-Storage Management**

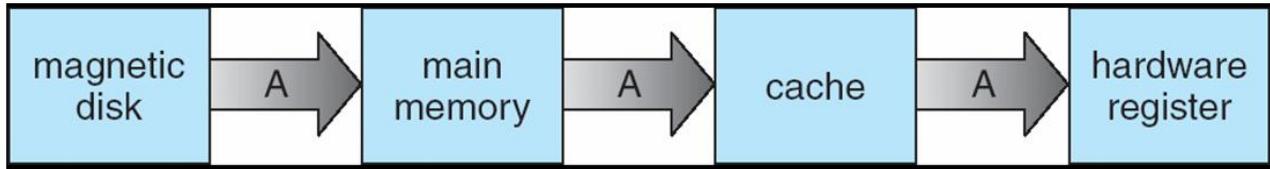
- Usually disks used to store data that does not fit in main memory or data that must be kept for a “long” period of time
- Proper management is of central importance
- Entire speed of computer operation hinges on disk subsystem and its algorithms
- **MASS STORAGE activities**
- Free-space management
- Storage allocation
- Disk scheduling
- Some storage need not be fast
- Tertiary storage includes optical storage, magnetic tape
- Still must be managed
- Varies between WORM (write-once, read-many-times) and RW (read-write)

### **Performance of Various Levels of Storage**

| Level                     | 1                                       | 2                             | 3                | 4                |
|---------------------------|-----------------------------------------|-------------------------------|------------------|------------------|
| Name                      | registers                               | cache                         | main memory      | disk storage     |
| Typical size              | < 1 KB                                  | > 16 MB                       | > 16 GB          | > 100 GB         |
| Implementation technology | custom memory with multiple ports, CMOS | on-chip or off-chip CMOS SRAM | CMOS DRAM        | magnetic disk    |
| Access time (ns)          | 0.25 – 0.5                              | 0.5 – 25                      | 80 – 250         | 5,000.000        |
| Bandwidth (MB/sec)        | 20,000 – 100,000                        | 5000 – 10,000                 | 1000 – 5000      | 20 – 150         |
| Managed by                | compiler                                | hardware                      | operating system | operating system |
| Backed by                 | cache                                   | main memory                   | disk             | CD or tape       |

### Migration of Integer A from Disk to Register

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



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

### I/O Subsystem

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

### Protection and Security

**Protection** – any mechanism for controlling access of processes or users to resources defined by the OS

**Security** – defense of the system against internal and external attacks

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

## DISTRIBUTED SYSTEMS

### Computing Environments

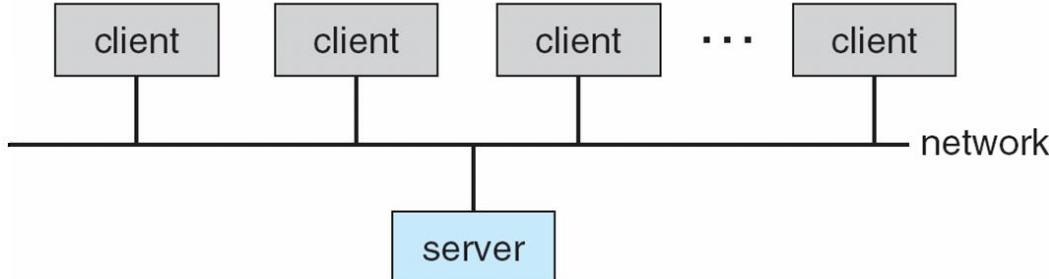
#### Traditional computer

- Blurring over time
- Office environment
  - PCs connected to a network, terminals attached to mainframe or minicomputers providing batch and timesharing
  - Now portals allowing networked and remote systems access to same resources
- Home networks
  - Used to be single system, then modems
  - Now firewalled, networked
- Client-Server Computing

- Dumb terminals supplanted by smart PCs
- Many systems now **servers**, responding to requests generated by **clients**

**Compute-server** provides an interface to client to request services (i.e. database)

**File-server** provides interface for clients to store and retrieve files



## Peer-to-Peer Computing

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

## Web-Based Computing

- Web has become ubiquitous
- PCs most prevalent devices
- More devices becoming networked to allow web access
- New category of devices to manage web traffic among similar servers: **load balancers**
- Use of operating systems like Windows 95, client-side, have evolved into Linux and Windows XP, which can be clients and servers

## Open-Source Operating Systems

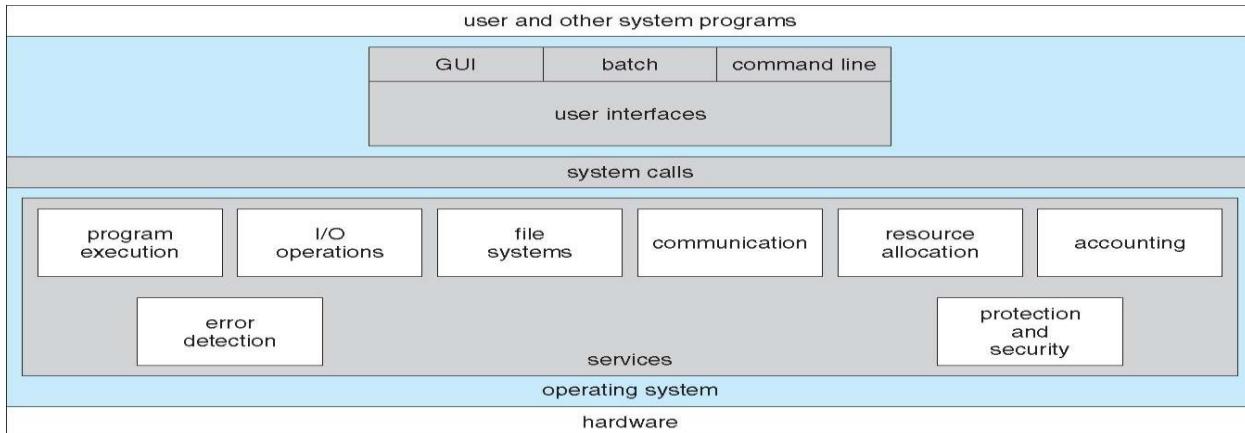
- Operating systems made available in source-code format rather than just binary closed-source
- Counter to the copy protection and Digital Rights Management (DRM) movement
- Started by Free Software Foundation (FSF), which has “copyleft” GNU Public License (GPL)
- Examples include GNU/Linux, BSD UNIX (including core of Mac OS X), and Sun Solaris

## Operating System Services

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

- File-system manipulation - The file system is of particular interest. Obviously, programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.

## A View of Operating System Services



## Operating System Services

- One set of operating-system services provides functions that are helpful to the user  
Communications – Processes may exchange information, on the same computer or between computers over a network Communications may be via shared memory or through message passing (packets moved by the OS)
- Error detection – OS needs to be constantly aware of possible errors May occur in the CPU and memory hardware, in I/O devices, in user program For each type of error, OS should take the appropriate action to ensure correct and consistent computing Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system
- Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing
- **Resource allocation** - When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
- Many types of resources - Some (such as CPU cycles, main memory, and file storage) may have special allocation code, others (such as I/O devices) may have general request and release code
- **Accounting** - To keep track of which users use how much and what kinds of computer resources
- **Protection and security** - The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
- **Protection** involves ensuring that all access to system resources is controlled
- **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts
- If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link.

## User Operating System Interface - CLI

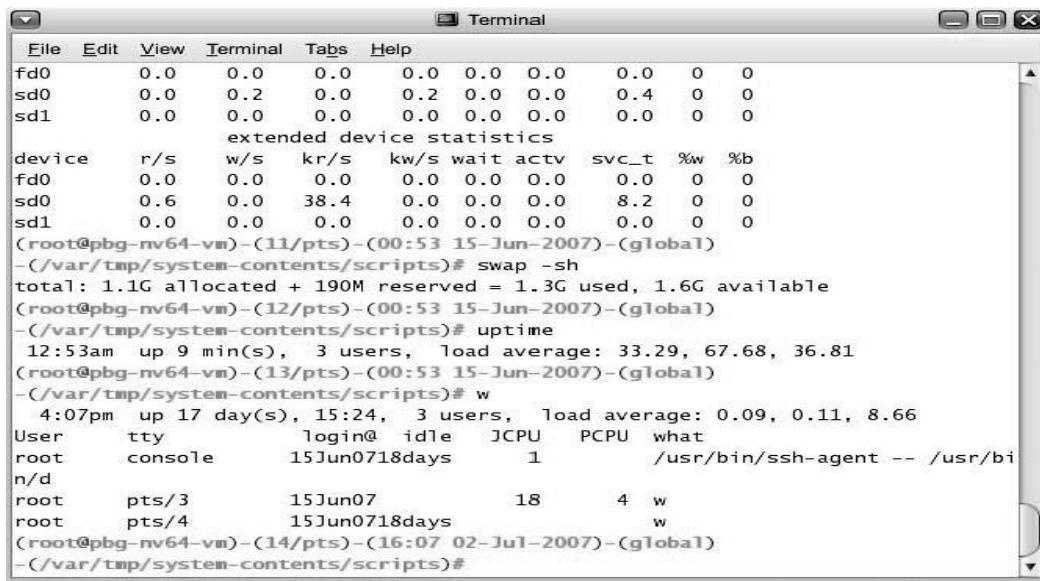
- Command Line Interface (CLI) or command interpreter allows direct command entry

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

## User Operating System Interface - GUI

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

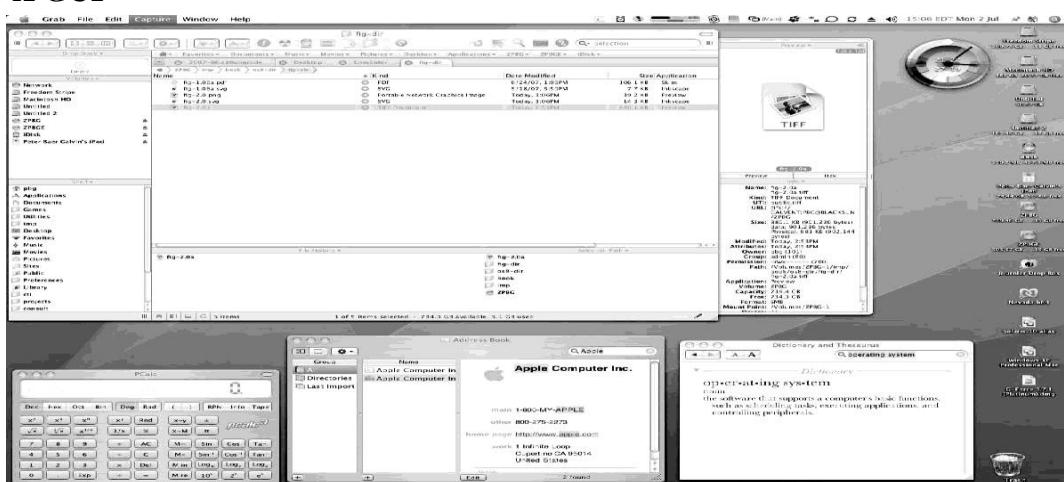
## Bourne Shell Command Interpreter



The screenshot shows a terminal window with the title 'Terminal'. The window contains the following text:

```
File Edit View Terminal Tabs Help
fd0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0 0
sd0 0.0 0.2 0.0 0.2 0.0 0.0 0.4 0 0
sd1 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0 0
extended device statistics
device r/s w/s kr/s kw/s wait actv svc_t %w %b
fd0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0 0
sd0 0.6 0.0 38.4 0.0 0.0 0.0 8.2 0 0
sd1 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0 0
(root@pbg-nv64-vm)-(11/pts)-(00:53 15-Jun-2007)-(global)
-/var/tmp/system-contents/scripts)#
total: 1.1G allocated + 190M reserved = 1.3G used, 1.6G available
(root@pbg-nv64-vm)-(12/pts)-(00:53 15-Jun-2007)-(global)
-/var/tmp/system-contents/scripts)#
 uptime
12:53am up 9 min(s), 3 users, load average: 33.29, 67.68, 36.81
(root@pbg-nv64-vm)-(13/pts)-(00:53 15-Jun-2007)-(global)
-/var/tmp/system-contents/scripts)#
 w
4:07pm up 17 day(s), 15:24, 3 users, load average: 0.09, 0.11, 8.66
User     tty      login@    idle   JCPU   PCPU what
root    console    15Jun0718days      1      /usr/bin/ssh-agent -- /usr/bi
n/d
root    pts/3      15Jun07          18      4  w
root    pts/4      15Jun0718days      1      w
(root@pbg-nv64-vm)-(14/pts)-(16:07 02-Jul-2007)-(global)
-/var/tmp/system-contents/scripts)#
-
```

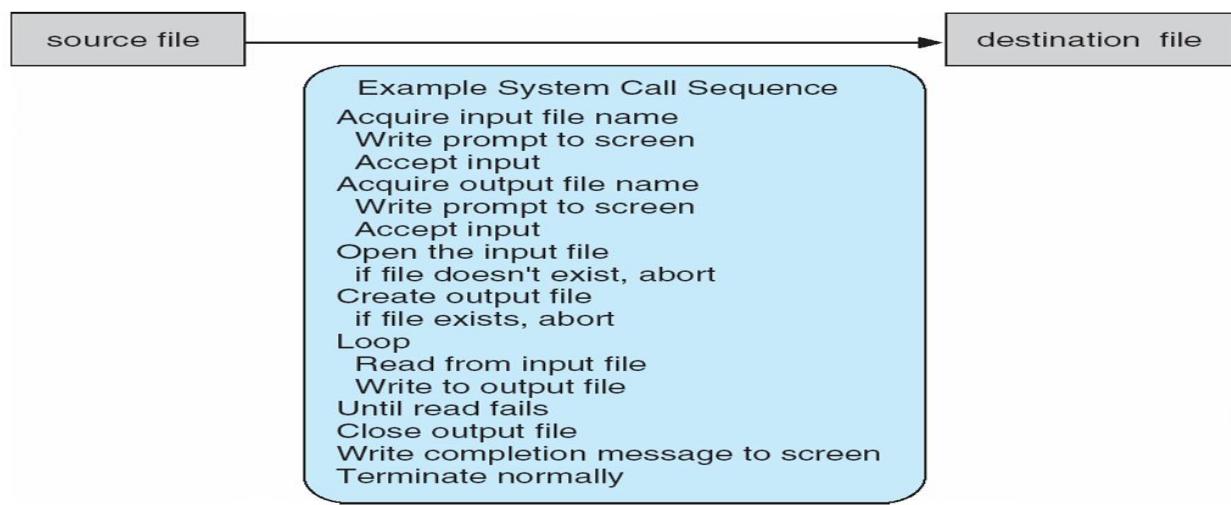
## The Mac OS X GUI



## System Calls

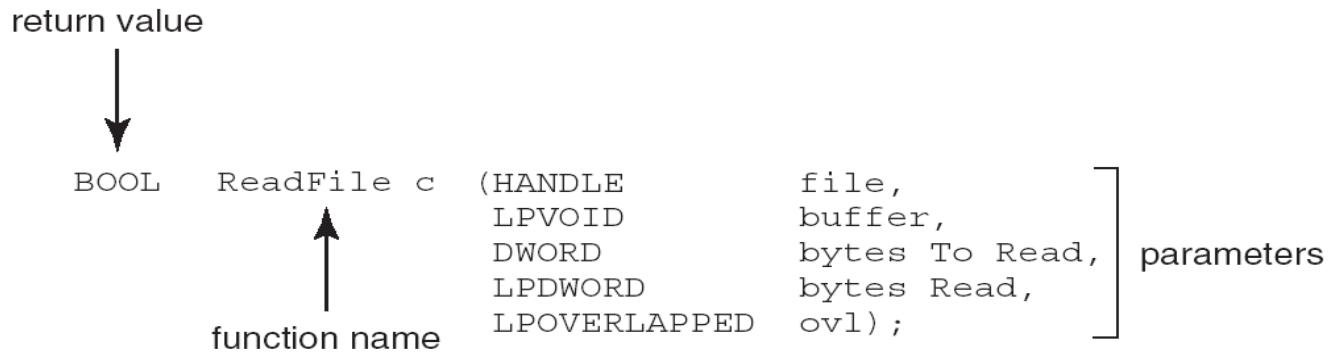
- Programming interface to the services provided by the OS
  - Typically written in a high-level language (C or C++)
  - Mostly accessed by programs via a high-level Application Program Interface (API) rather than direct system call userThree most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)
  - Why use APIs rather than system calls?(Note that the system-call names used throughout this text are generic)

## Example of System Calls



### Example of Standard API

Consider the ReadFile() function in the Win32 API—a function for reading from a file



A description of the parameters passed to ReadFile()

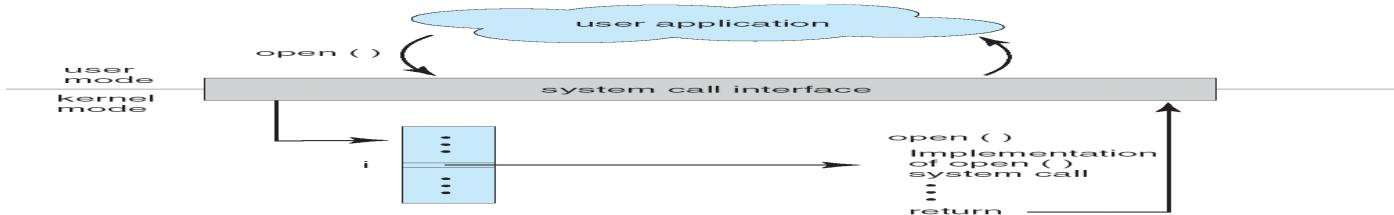
- HANDLE file—the file to be read
- LPVOID buffer—a buffer where the data will be read into and written from
- DWORD bytesToRead—the number of bytes to be read into the buffer
- LPDWORD bytesRead—the number of bytes read during the last read
- LPOVERLAPPED ovl—indicates if overlapped I/O is being used

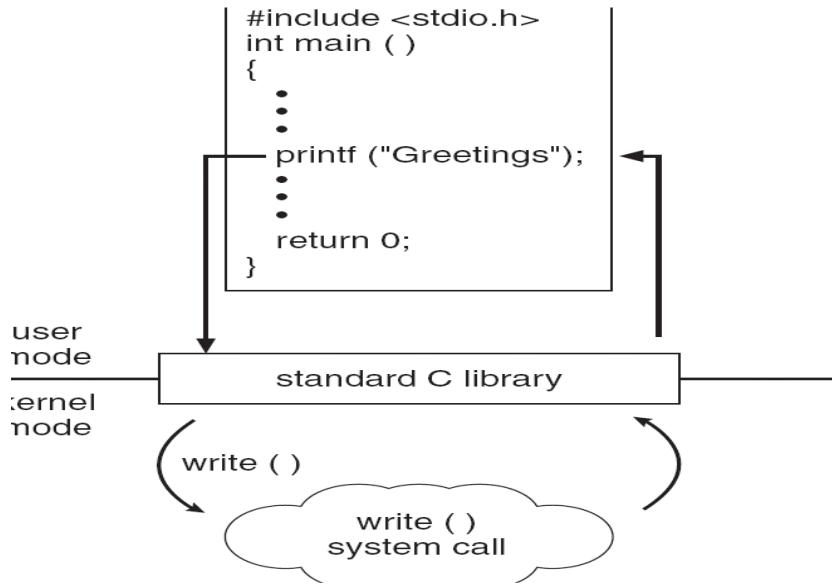
### System Call Implementation

- Typically, a number associated with each system call
- System-call interface maintains a table indexed according to these numbers
- The system call interface invokes 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 call
- Most details of OS interface hidden from programmer by API

Managed by run-time support library (set of functions built into libraries included with compiler)

### API – System Call – OS Relationship

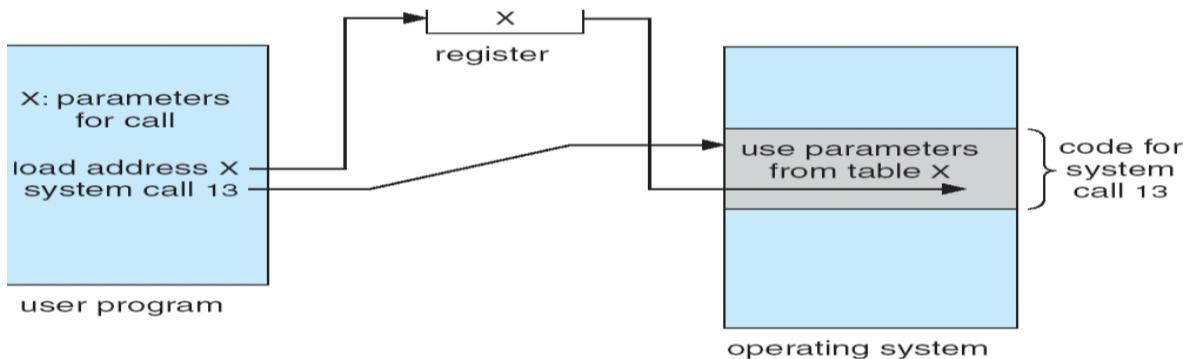




### System Call Parameter Passing

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

### Parameter Passing via Table



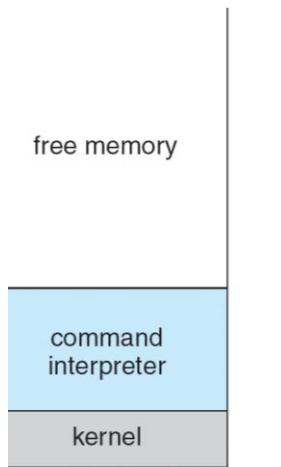
### Types of System Calls

- Process control
- File management
- Device management
- Information maintenance
- Communications
- Protection

### Examples of Windows and Unix System Calls

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

### MS-DOS execution



(a)

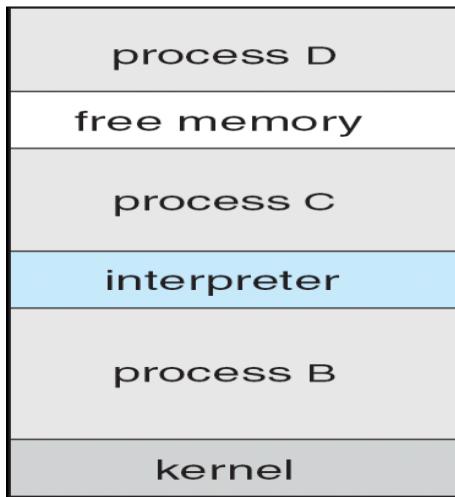
(a) At system startup



(b)

(b) running a program

## FreeBSD Running Multiple Programs



## System Programs

System programs provide a convenient environment for program development and execution. They can be divided into:

- File manipulation
- Status information
- File modification
- Programming language support
- Program loading and execution
- Communications
- Application programs

Most users' view of the operation system is defined by system programs, not the actual system calls

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

File modification

- Text editors to create and modify files
- Special commands to search contents of files or perform transformations of the text
- Programming-language support - Compilers, assemblers, debuggers and interpreters sometimes provided
- Program loading and execution- Absolute loaders, relocatable loaders, linkage editors, and overlay-loaders, debugging systems for higher-level and machine language

- Communications - Provide the mechanism for creating virtual connections among processes, users, and computer systems
- Allow users to send messages to one another's screens, browse web pages, send electronic-mail messages, log in remotely, transfer files from one machine to another

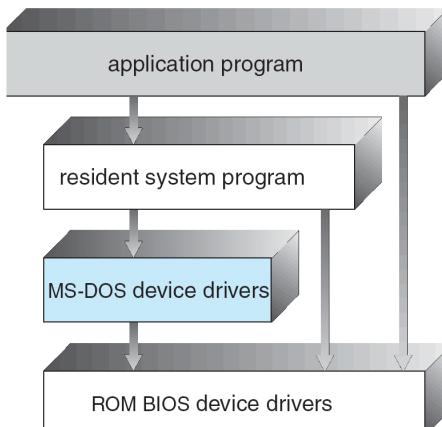
## Operating System Design and Implementation

- Design and Implementation of OS not “solvable”, but some approaches have proven successful
- Internal structure of different Operating Systems can vary widely
- Start by defining goals and specifications
- Affected by choice of hardware, type of system
- *User* goals and *System* goals
- User goals – operating system should be convenient to use, easy to learn, reliable, safe, and fast
- System goals – operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient
- Important principle to separate
- **Policy:** What will be done?  
**Mechanism:** How to do it?
- Mechanisms determine how to do something, policies decide what will be done
- The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later

## Simple Structure

- MS-DOS – written to provide the most functionality in the least space
- Not divided into modules
- Although MS-DOS has some structure, its interfaces and levels of Functionality are not well separated

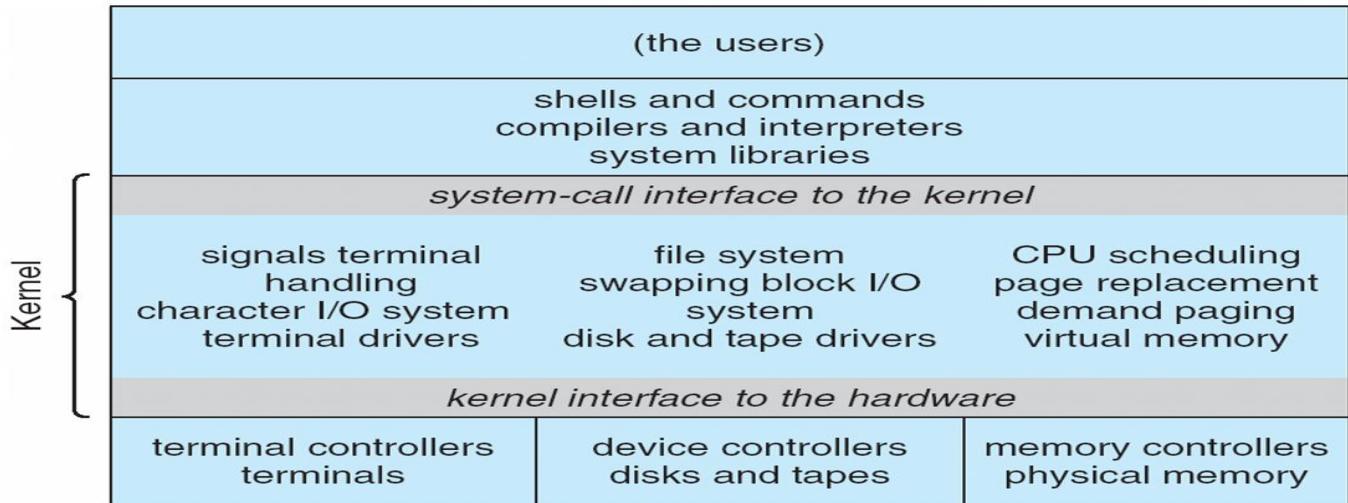
## MS-DOS Layer Structure



## 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

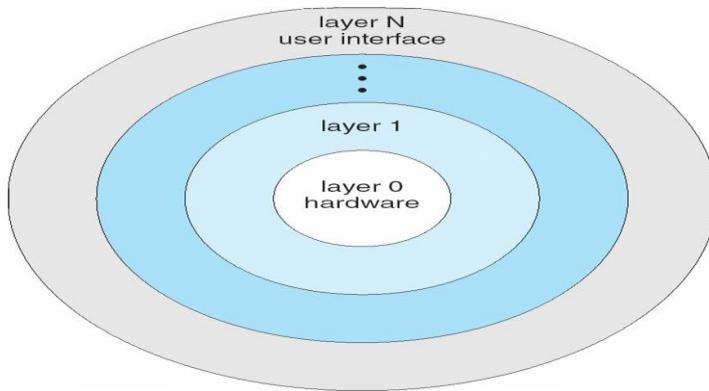
## Traditional UNIX System Structure



## UNIX

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

## Layered Operating System

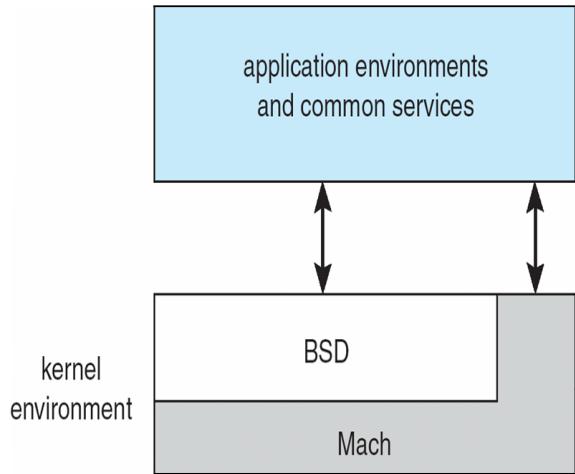


## Micro kernel System Structure

- Moves as much from the kernel into “*user*” space
- Communication takes place between user modules using message passing
- Benefits:
  - Easier to extend a microkernel
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode)

- More secure
- Detriments:
- Performance overhead of user space to kernel space communication

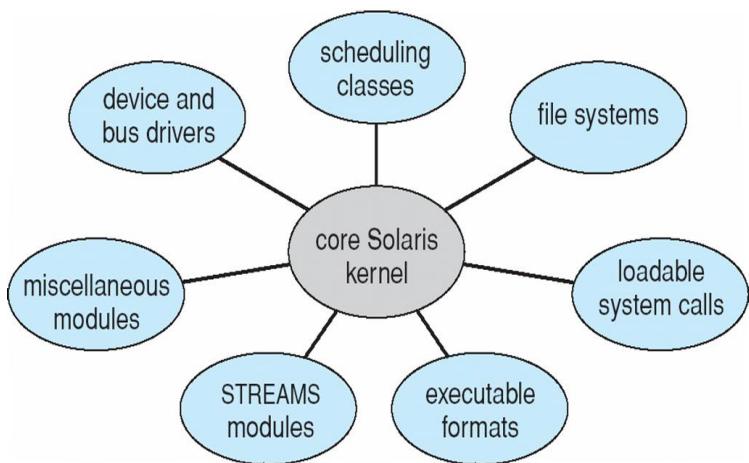
## Mac OS X Structure



## Modules

- Most modern operating systems implement kernel modules
- Uses object-oriented approach
- Each core component is separate
- Each talks to the others over known interfaces
- Each is loadable as needed within the kernel
- Overall, similar to layers but with more flexible

## Solaris Modular Approach



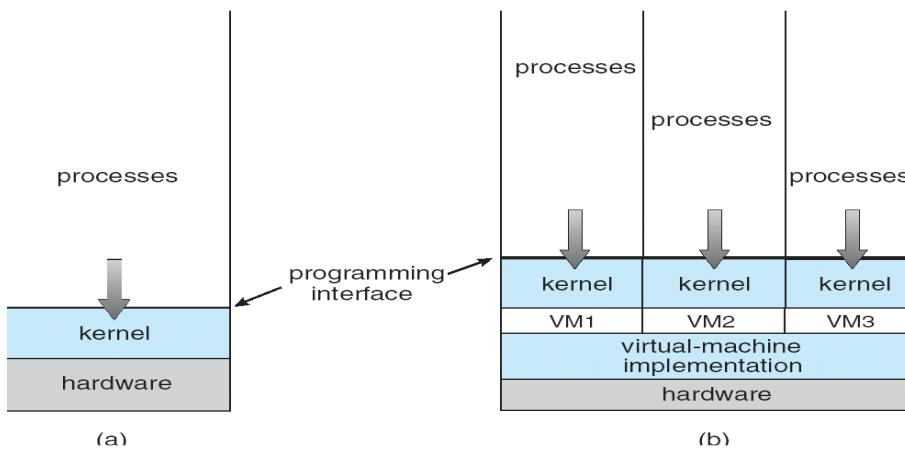
## Virtual Machines

- A virtual machine takes the layered approach to its logical conclusion. It treats hardware and the operating system kernel as though they were all hardware

- A virtual machine provides an interface *identical* to the underlying bare hardware
- The operating system host creates the illusion that a process has its own processor and (virtual) memory
- Each guest provided with a (virtual) copy of underlying computer

### Virtual Machines History and Benefits

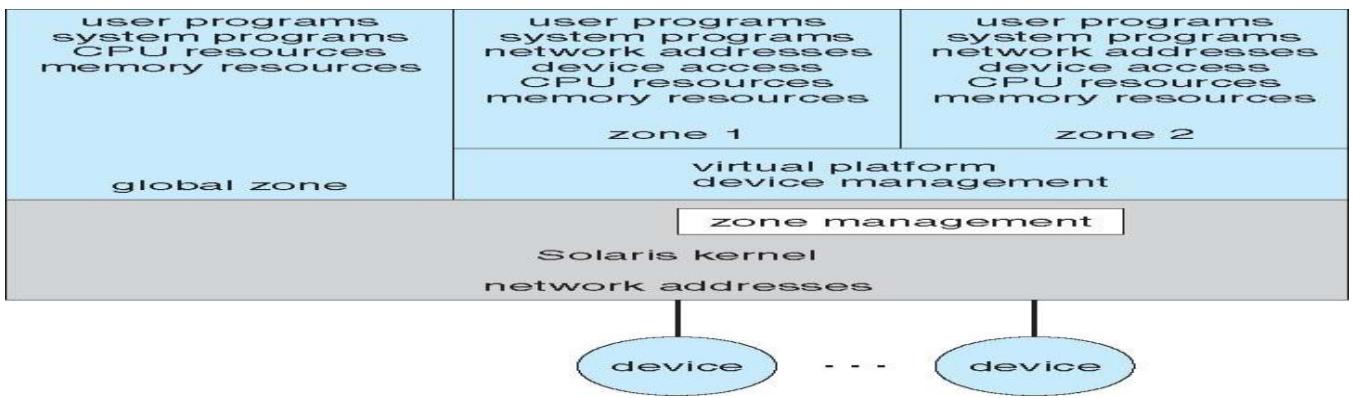
- First appeared commercially in IBM mainframes in 1972
- Fundamentally, multiple execution environments (different operating systems) can share the same hardware
- Protect from each other
- Some sharing of file can be permitted, controlled
- Communicate with each other, other physical systems via networking
- Useful for development, testing
- Consolidation of many low-resource use systems onto fewer busier systems
- “Open Virtual Machine Format”, standard format of virtual machines, allows a VM to run within many different virtual machine (host) platforms



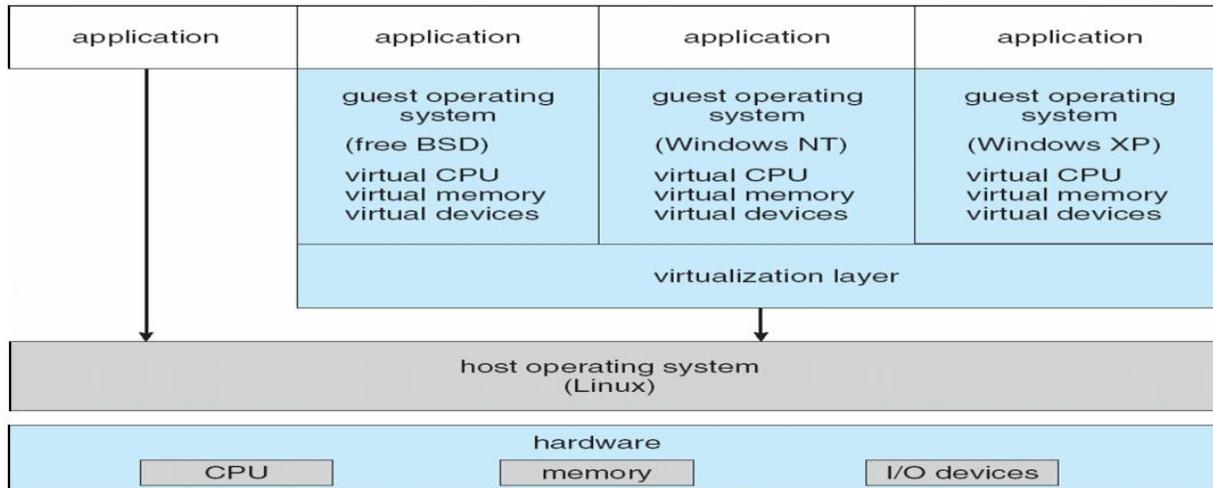
### Para-virtualization

- Presents guest with system similar but not identical to hardware
- Guest must be modified to run on paravirtualized hardware
- Guest can be an OS, or in the case of Solaris 10 applications running in containers

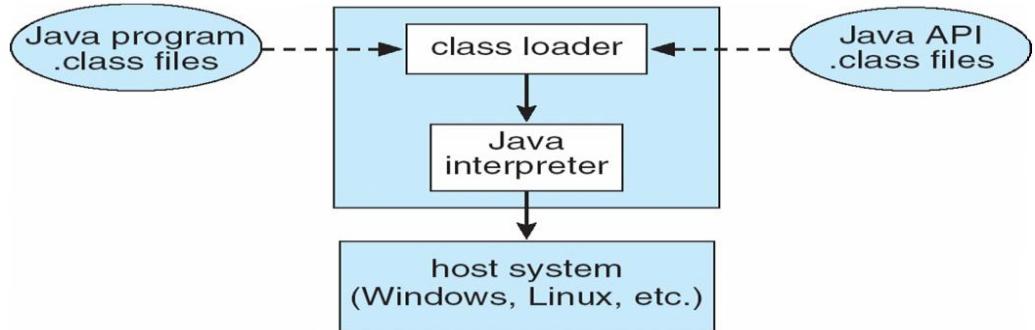
### Solaris 10 with Two Containers



## VMware Architecture



## The Java Virtual Machine



## Operating-System Debugging

- Debugging is finding and fixing errors, or bugs
- OSes generate log files containing error information
- Failure of an application can generate core dump file capturing memory of the process
- Operating system failure can generate crash dump file containing kernel memory
- Beyond crashes, performance tuning can optimize system performance
- Kernighan's Law: "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."
- DTrace tool in Solaris, FreeBSD, Mac OS X allows live instrumentation on production systems
- Probes fire when code is executed, capturing state data and sending it to consumers of those probes

## Solaris 10 dtrace Following System Call

```
# ./all.d 'pgrep xclock' XEventsQueued
dtrace: script './all.d' matched 52377 probes
CPU FUNCTION
  0 -> XEventsQueued                                U
  0  -> _XEventsQueued                            U
  0  -> _X11TransBytesReadable                     U
  0  <- _X11TransBytesReadable                     U
  0  -> _X11TransSocketBytesReadable              U
  0  <- _X11TransSocketBytesReadable              U
  0  -> ioctl                                      U
  0  -> ioctl                                     K
  0  -> getf                                       K
  0    -> set_active_fd                           K
  0    <- set_active_fd                           K
  0  <- getf                                      K
  0  -> get_udatamodel                          K
  0  <- get_udatamodel                          K
...
  0  -> releaseef                               K
  0    -> clear_active_fd                      K
  0    <- clear_active_fd                      K
  0  -> cv_broadcast                           K
  0  <- cv_broadcast                           K
  0  <- releaseef                               K
  0  <- ioctl                                    K
  0  <- ioctl                                   U
  0  <- _XEventsQueued                         U
  0 <- XEventsQueued                           U
```

## Operating System Generation

- Operating systems are designed to run on any of a class of machines; the system must be configured for each specific computer site
- SYSGEN program obtains information concerning the specific configuration of the hardware system
- *Booting* – starting a computer by loading the kernel
- *Bootstrap program* – code stored in ROM that is able to locate the kernel, load it into memory, and start its execution

## System Boot

- Operating system must be made available to hardware so hardware can start it
- Small piece of code – **bootstrap loader**, locates the kernel, loads it into memory, and starts it
- Sometimes two-step process where **boot block** at fixed location loads bootstrap loader
- When power initialized on system, execution starts at a fixed memory location Firmware used to hold initial boot code

## UNIT -2

### PROCESS MANAGEMENT

#### Process Concept

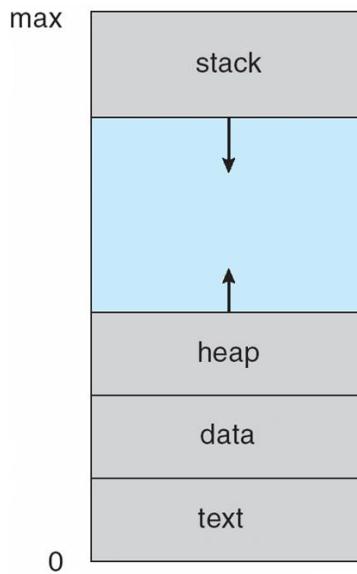
- An operating system executes a variety of programs:
- Batch system – jobs
- Time-shared systems – user programs or tasks
- Textbook uses the terms *job* and *process* almost interchangeably

Process – a program in execution; process execution must progress in sequential fashion

A process includes:

- program counter
- stack
- data section

#### Process in Memory

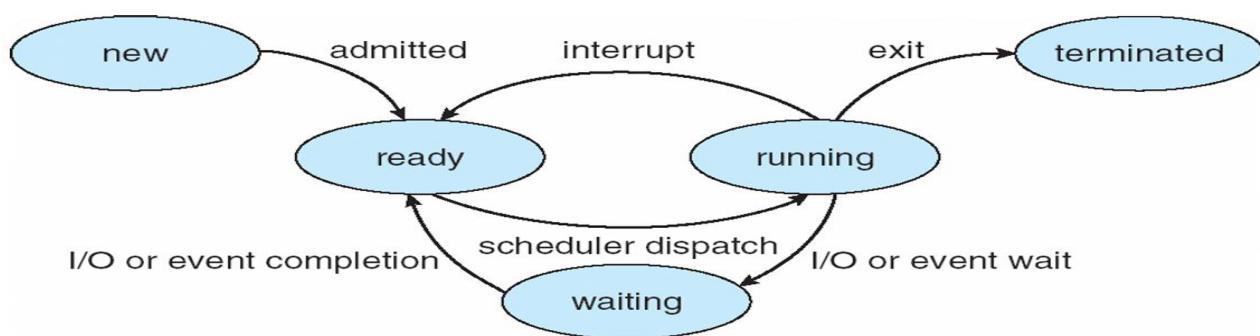


#### Process State

As a process executes, it changes *state*

- **new**: The process is being created
- **running**: Instructions are being executed
- **waiting**: The process is waiting for some event to occur
- **ready**: The process is waiting to be assigned to a processor
- **terminated**: The process has finished execution

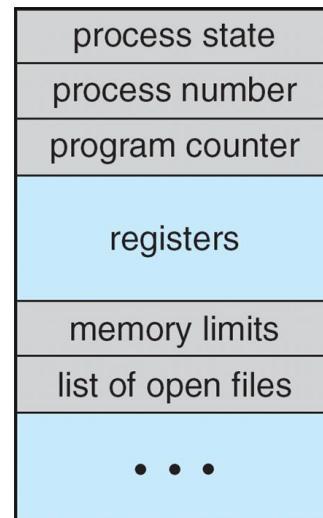
#### Diagram of Process State



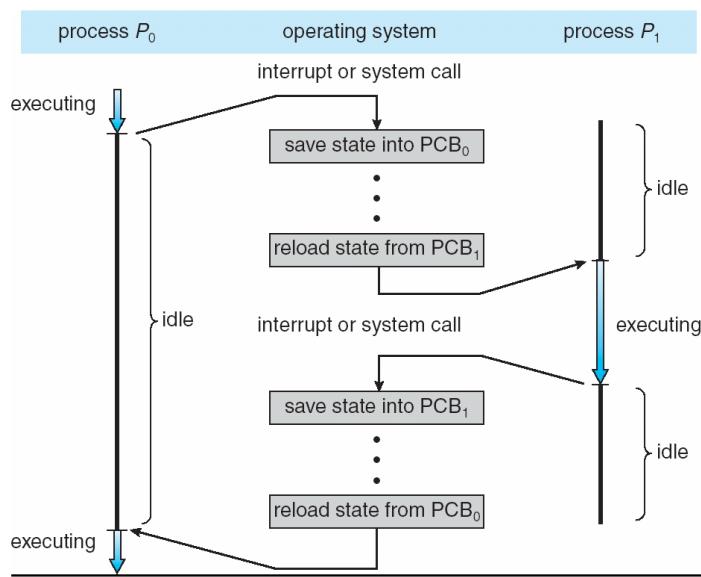
## Process Control Block (PCB)

Information associated with each process

- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information



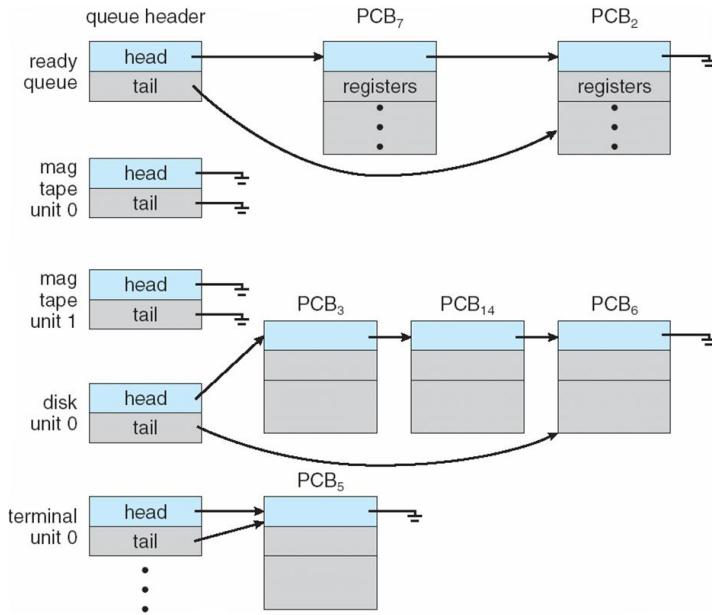
## CPU Switch From Process to Process



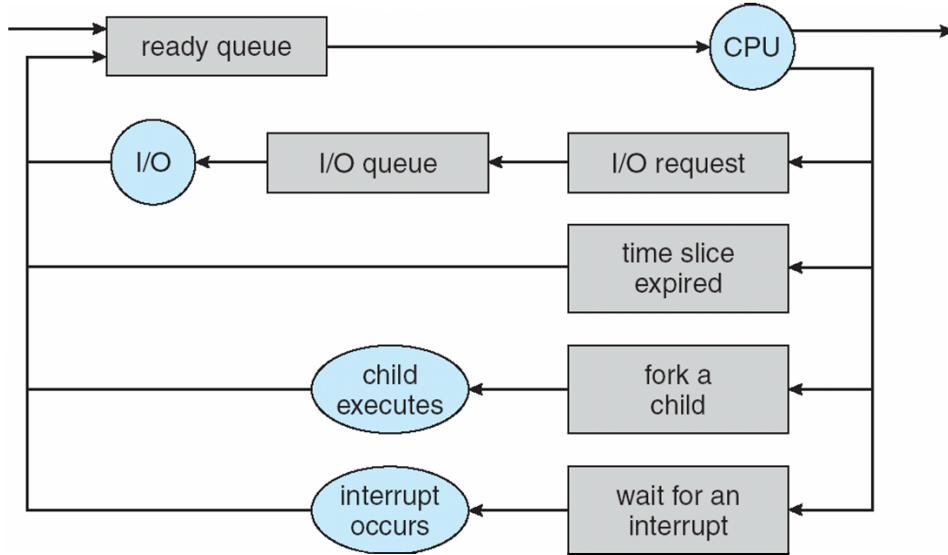
## Process Scheduling Queues

- **Job queue** – set of all processes in the system
- **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
- **Device queues** – set of processes waiting for an I/O device
- Processes migrate among the various queues

## Ready Queue and Various I/O Device Queues



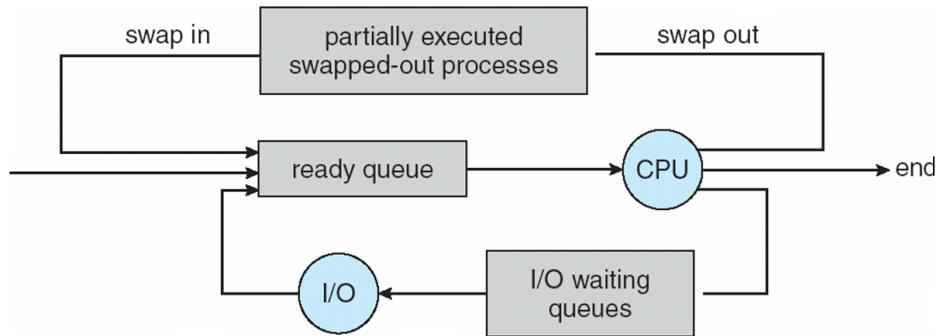
## Representation of Process Scheduling



### Schedulers

- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU

## Addition of Medium Term Scheduling



- Short-term scheduler is invoked very frequently (milliseconds)  $\rightarrow$  (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes)  $\rightarrow$  (may be slow)
- The long-term scheduler controls the *degree of multiprogramming*
- 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

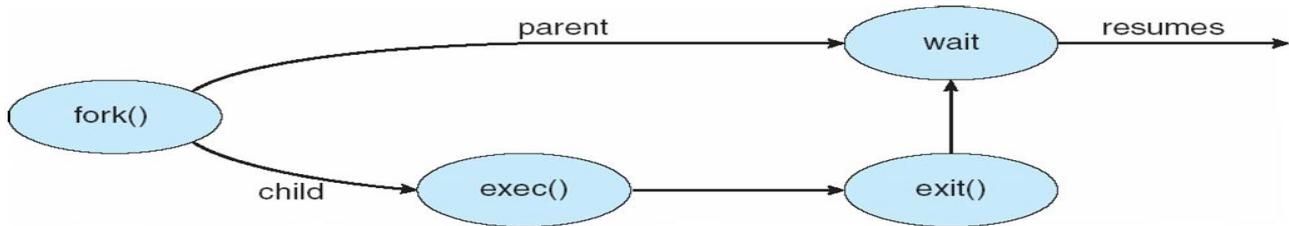
## Context Switch

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch
- Context of a process represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
- Time dependent on hardware support

## Process Creation

- **Parent** process creates **children** processes, which, in turn create other processes, forming a tree of processes
- Generally, process identified and managed via a **process identifier (pid)**
- Resource sharing
- Parent and children share all resources
- Children share subset of parent's resources
- Parent and child share no resources
- Execution
- Parent and children execute concurrently
- Parent waits until children terminate
- Address space
- Child duplicate of parent
- Child has a program loaded into it
- UNIX examples
- **fork** system call creates new process
- **exec** system call used after a **fork** to replace the process' memory space with a new program

## Process Creation

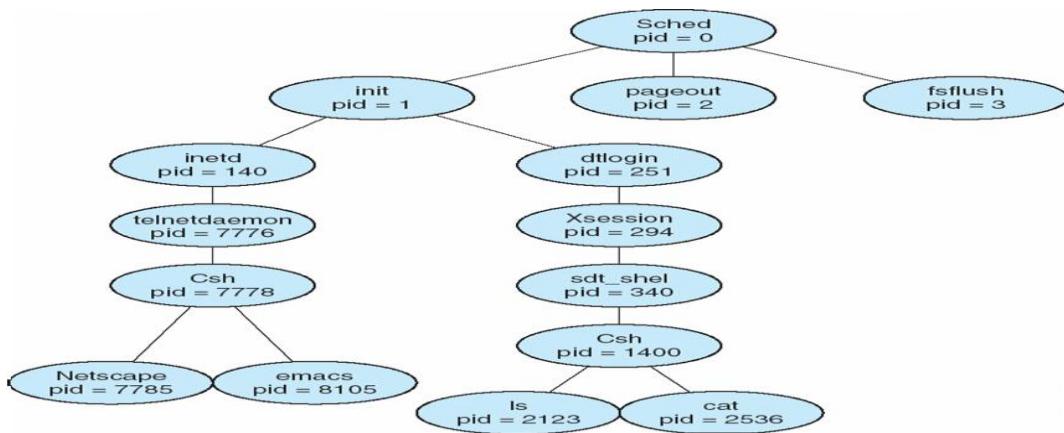


### C Program Forking Separate Process

```

int main()
{
pid_t pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(-1);
    }
    else if (pid == 0) { /* child process */
        execl("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait (NULL);
        printf ("Child Complete");
        exit(0);
    }
}
  
```

### A tree of processes on a typical Solaris



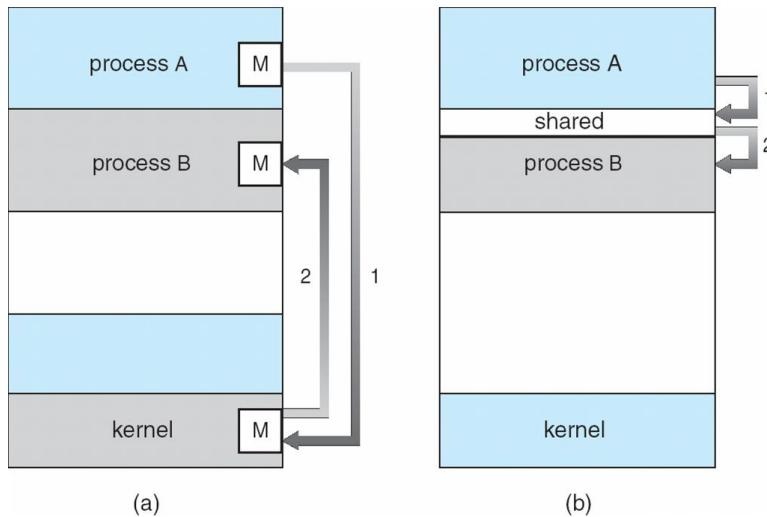
### Process Termination

- Process executes last statement and asks the operating system to delete it (**exit**)
- Output data from child to parent (via **wait**)
- Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (**abort**)
- Child has exceeded allocated resources
- Task assigned to child is no longer required
- If parent is exiting Some operating system do not allow child to continue if its parent terminates  
All children terminated - **cascading termination**

### 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

### Communications Models



### Cooperating Processes

- **Independent** process cannot affect or be affected by the execution of another process
- **Cooperating** process can affect or be affected by the execution of another process

### Advantages of process cooperation

- Information sharing
- Computation speed-up
- Modularity
- Convenience

### 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

### Bounded-Buffer – Shared-Memory Solution

Shared data

```
#define BUFFER_SIZE 10
```

```
typedef struct {
```

```
    ...
```

```
    } item;
```

```
    item buffer[BUFFER_SIZE];
```

```
    int in = 0;
```

```
    int out = 0;
```

Solution is correct, but can only use BUFFER\_SIZE-1 elements

### Bounded-Buffer – Producer

```
    while (true) {
        /* Produce an item */
        while (((in = (in + 1) % BUFFER_SIZE) == out)
            ; /* do nothing -- no free buffers */
            buffer[in] = item;
            in = (in + 1) % BUFFER_SIZE;
    }
```

### Bounded Buffer – Consumer

```
while (true) {
    while (in == out)
        ; // do nothing -- nothing to consume
        // remove an item from the buffer
        item = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        return item;
}
```

### Interprocess Communication – Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
- **send(message)** – message size fixed or variable
- **receive(message)**
- If *P* and *Q* wish to communicate, they need to:
- establish a *communication link* between them
- exchange messages via send/receive
- Implementation of communication link
- physical (e.g., shared memory, hardware bus)
- logical (e.g., logical properties)

### Direct Communication

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

### Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
- Each mailbox has a unique id
- Processes can communicate only if they share a mailbox
- Properties of communication link
- Link established only if processes share a common mailbox
- A link may be associated with many processes
- Each pair of processes may share several communication links
- Link may be unidirectional or bi-directional
- Operations
- create a new mailbox
- 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
- Mailbox sharing
- $P_1, P_2$ , and  $P_3$  share mailbox A
- $P_1$ , sends;  $P_2$  and  $P_3$  receive
- Who gets the message?
- Solutions
- Allow a link to be associated with at most two processes
- Allow only one process at a time to execute a receive operation
- Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

### Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
- **Blocking send** has the sender block until the message is received
- **Blocking receive** has the receiver block until a message is available
- **Non-blocking** is considered **asynchronous**
- **Non-blocking** send has the sender send the message and continue
- **Non-blocking** receive has the receiver receive a valid message or null

### **Buffering**

Queue of messages attached to the link; implemented in one of three ways

1. Zero capacity – 0 messages

Sender must wait for receiver (rendezvous)

2. Bounded capacity – finite length of  $n$  messages

Sender must wait if link full

3. Unbounded capacity – infinite length

Sender never waits

### **Examples of IPC Systems - POSIX**

- POSIX Shared Memory
- Process first creates shared memory segment
- segment id = shmget(IPC\_PRIVATE, size, S\_IRUSR | S\_IWUSR);
- Process wanting access to that shared memory must attach to it
- shared memory = (char \*) shmat(id, NULL, 0);
- Now the process could write to the shared memory
- printf(shared memory, "Writing to shared memory");
- When done a process can detach the shared memory from its address space
- shmdt(shared memory);

### **Examples of IPC Systems - Mach**

- Mach communication is message based
- Even system calls are messages
- Each task gets two mailboxes at creation- Kernel and Notify
- Only three system calls needed for message transfer
- msg\_send(), msg\_receive(), msg\_rpc()
- Mailboxes needed for communication, created via
- port\_allocate()

### **Examples of IPC Systems – Windows XP**

- Message-passing centric via local procedure call (LPC) facility
- Only works between processes on the same system
- Uses ports (like mailboxes) to establish and maintain communication channels
- Communication works as follows:

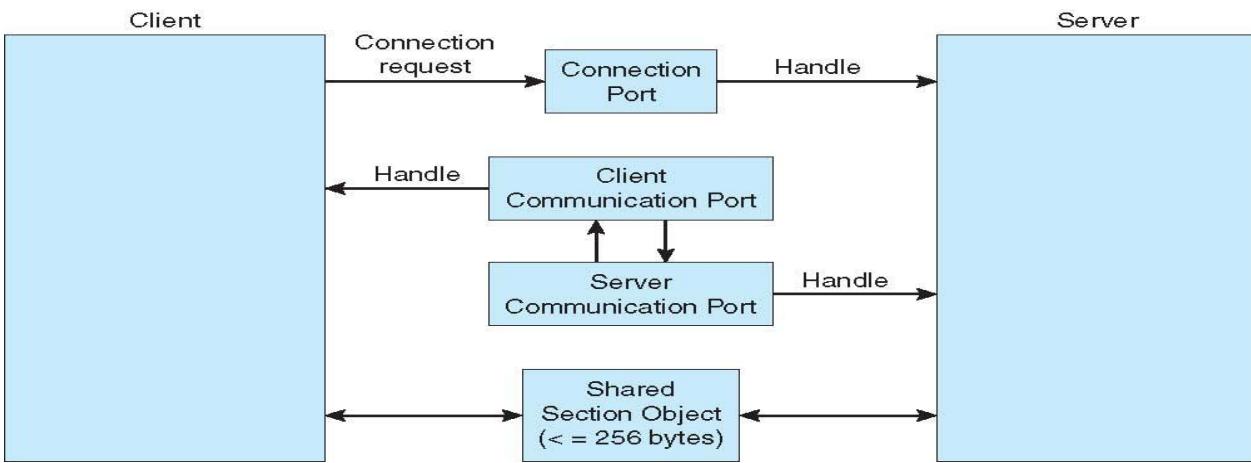
The client opens a handle to the subsystem's connection port object

The client sends a connection request

The server creates two private communication ports and returns the handle to one of them to the client

The client and server use the corresponding port handle to send messages or callbacks and to listen for replies

## Local Procedure Calls in Windows XP



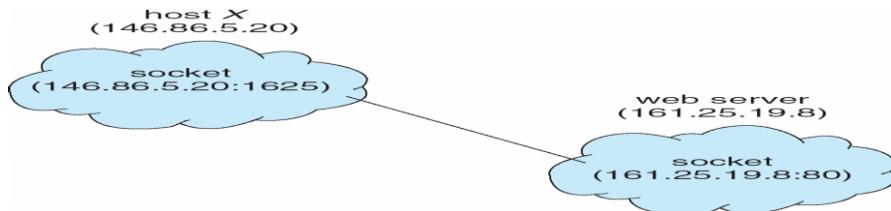
## Communications in Client-Server Systems

- Sockets
- Remote Procedure Calls
- Remote Method Invocation (Java)

### Sockets

- A socket is defined as an *endpoint for communication*
- Concatenation of IP address and port
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets

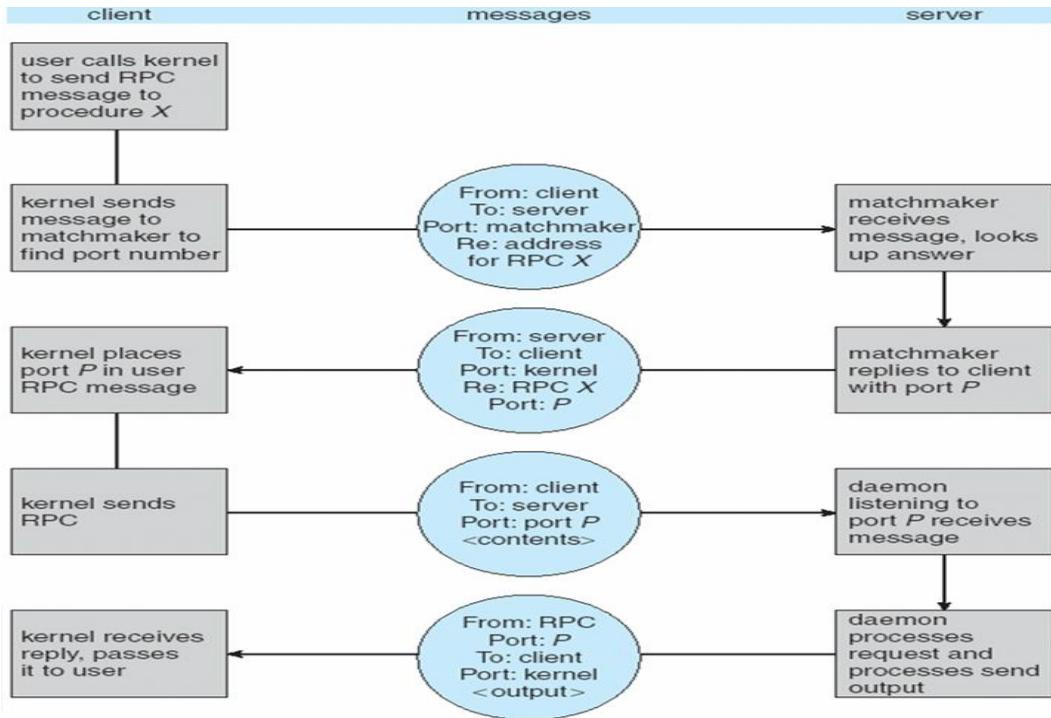
### Socket Communication



## Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
- **Stubs** – client-side proxy for the actual procedure on the server
- The client-side stub locates the server and *marshalls* the parameters
- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server

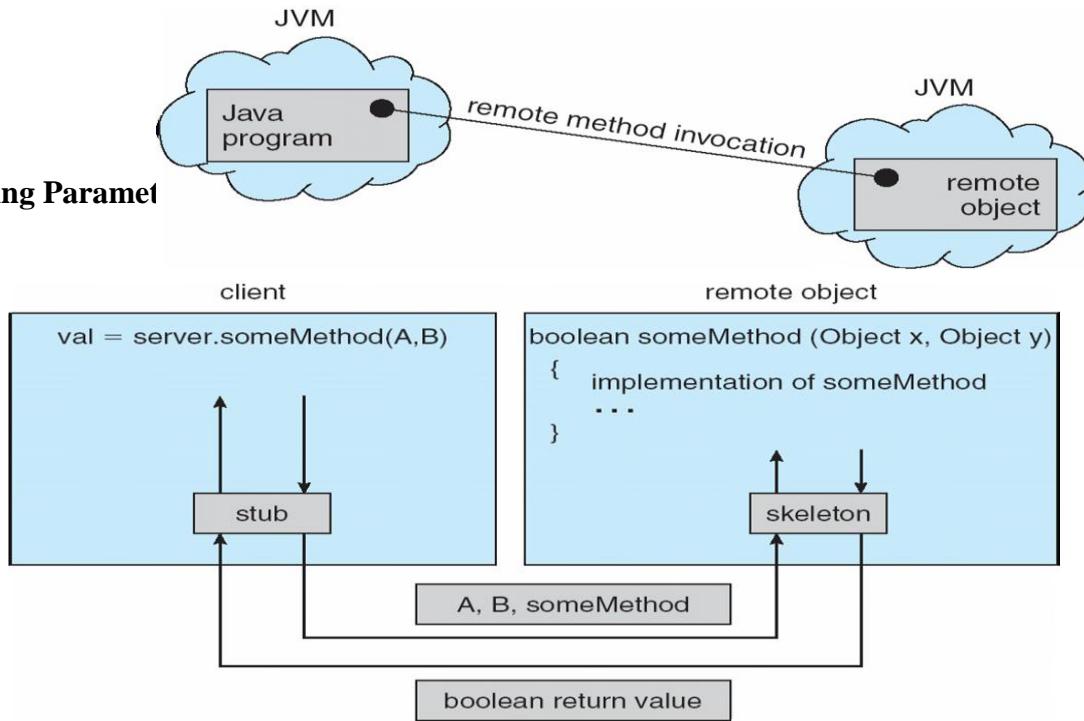
## Execution of RPC



## Remote Method Invocation

- Remote Method Invocation (RMI) is a Java mechanism similar to RPCs
- RMI allows a Java program on one machine to invoke a method on a remote object

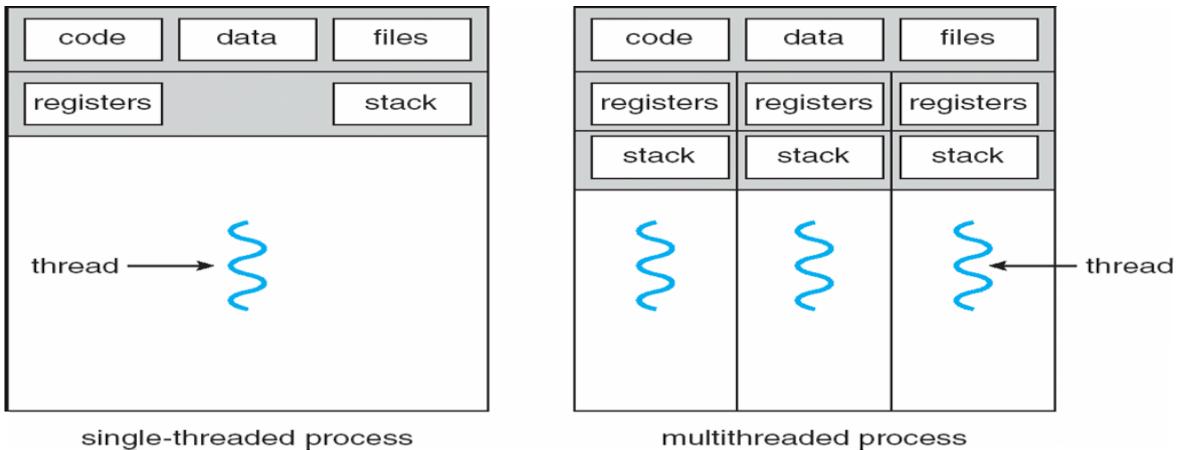
### Marshalling Parameters



## Threads

- To introduce the notion of a thread — a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems
- To discuss the APIs for the Pthreads, Win32, and Java thread libraries
- To examine issues related to multithreaded programming

## Single and Multithreaded Processes



## Benefits

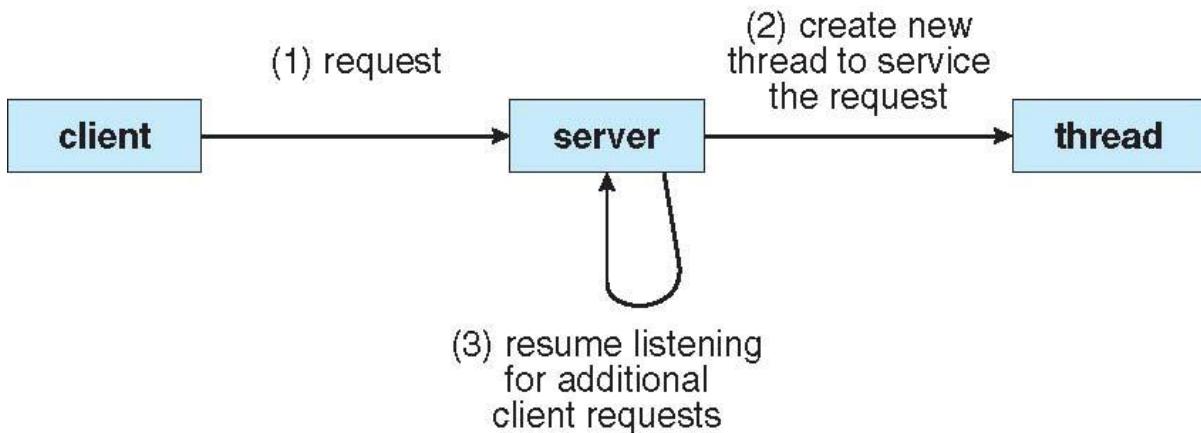
- Responsiveness
- Resource Sharing
- Economy
- Scalability

## Multicore Programming

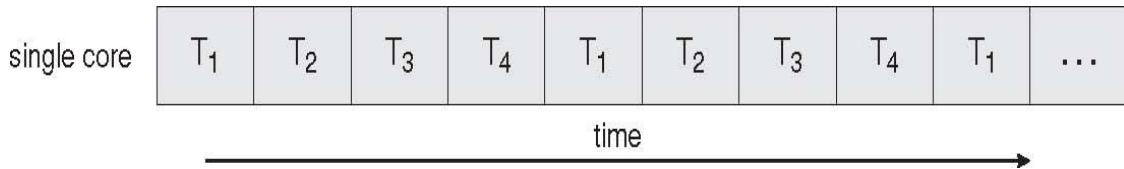
Multicore systems putting pressure on programmers, challenges include

- Dividing activities
- Balance
- Data splitting
- Data dependency
- Testing and debugging

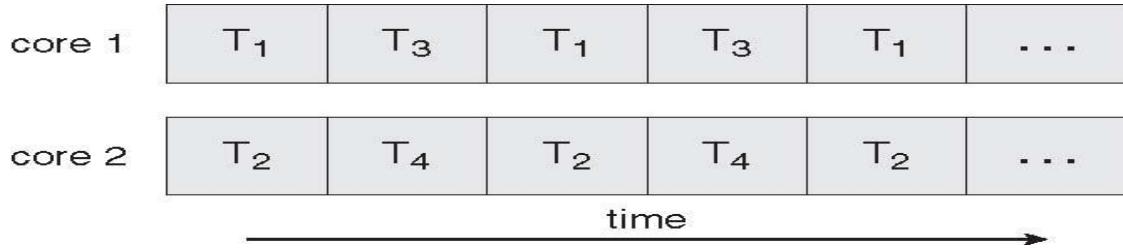
## Multithreaded Server Architecture



## Concurrent Execution on a Single-core System



## Parallel Execution on a Multicore System



### User Threads

- Thread management done by user-level threads library
- Three primary thread libraries:
- POSIX Pthreads
- Win32 threads
- Java threads

### Kernel Threads

Supported by the Kernel

Examples

- Windows XP/2000
- Solaris
- Linux
- Tru64 UNIX
- Mac OS X

### Multithreading Models

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

#### Many-to-One

Many user-level threads mapped to single kernel thread

Examples:

- Solaris Green Threads
- GNU Portable Threads

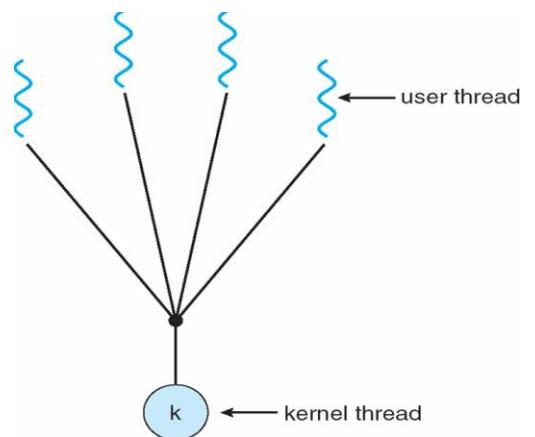
#### One-to-One

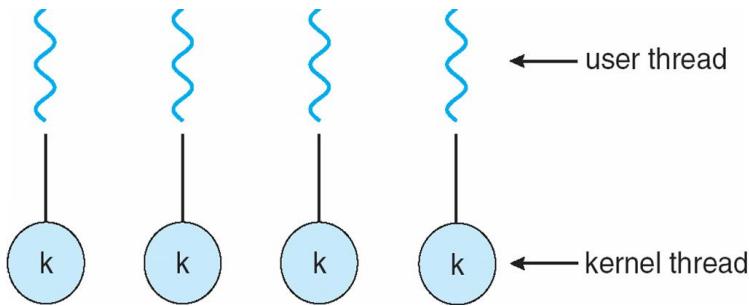
Each user-level thread maps to kernel thread

Examples

Windows NT/XP/2000

Linux

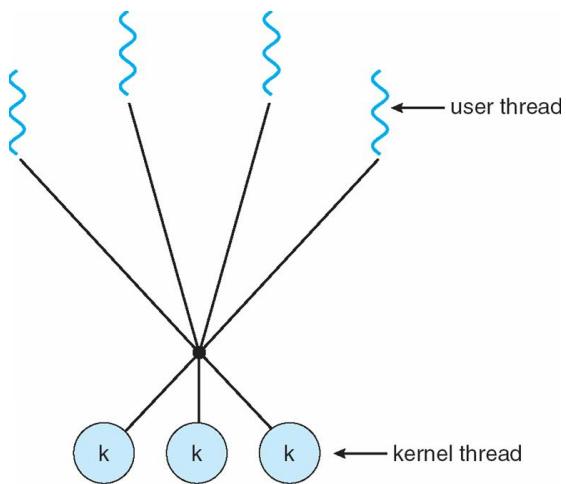




### Many-to-One Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9

Windows NT/2000 with the *ThreadFiber* package

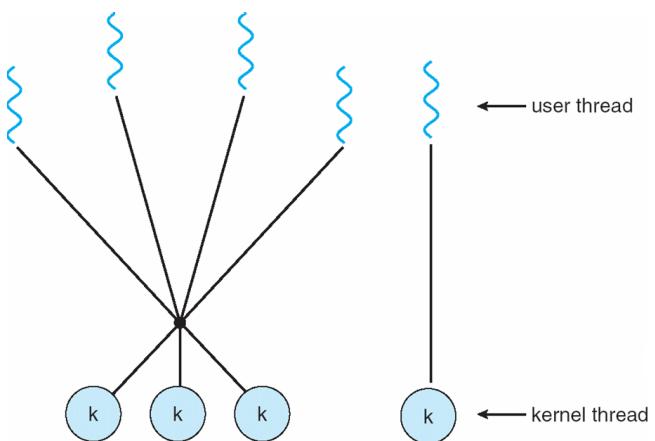


### Two-level Model

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

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
- 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)

### Java Threads

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

### Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Thread cancellation of target thread
- Asynchronous or deferred
- Signal handling
- Thread pools
- Thread-specific data
- Scheduler activations

### Thread Cancellation

- Terminating a thread before it has finished
- Two general approaches:
- **Asynchronous cancellation** terminates the target thread immediately
- **Deferred cancellation** allows the target thread to periodically check if it should be cancelled

### Signal Handling

- Signals are used in UNIX systems to notify a process that a particular event has occurred
- A signal handler is used to process signals
- 1.Signal is generated by particular event
- 2.Signal is delivered to a process
- 3.Signal is handled
- Options:
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread in the process
  - Deliver the signal to certain threads in the process
  - Assign a specific thread to receive all signals for the process

### Thread Pools

- Create a number of threads in a pool where they await work
- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool

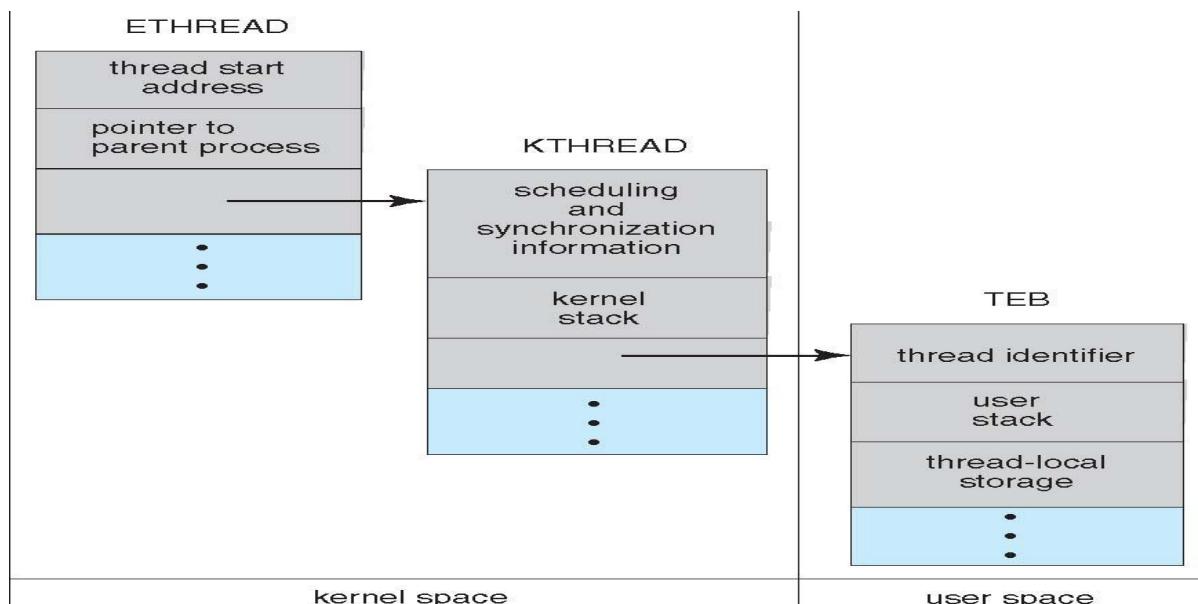
### Thread Specific Data

- Allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)

### Scheduler Activations

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Scheduler activations provide upcalls - a communication mechanism from the kernel to the thread library
- This communication allows an application to maintain the correct number kernel threads

### Windows XP Threads



Implements the one-to-one mapping, kernel-level

- Each thread contains
- A thread id
- Register set
- Separate user and kernel stacks
- Private data storage area
- The register set, stacks, and private storage area are known as the context of the threads
- The primary data structures of a thread include:
- ETHREAD (executive thread block)
- KTHREAD (kernel thread block)
- TEB (thread environment block)

## Linux Threads

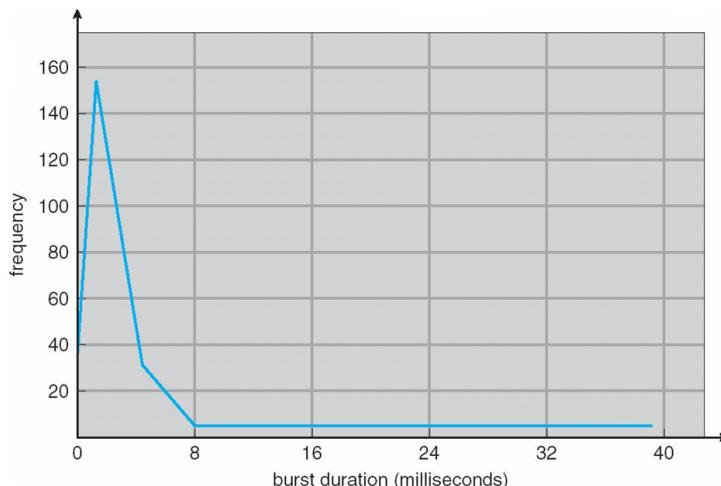
| flag          | meaning                            |
|---------------|------------------------------------|
| CLONE_FS      | File-system information is shared. |
| CLONE_VM      | The same memory space is shared.   |
| CLONE_SIGHAND | Signal handlers are shared.        |
| CLONE_FILES   | The set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

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

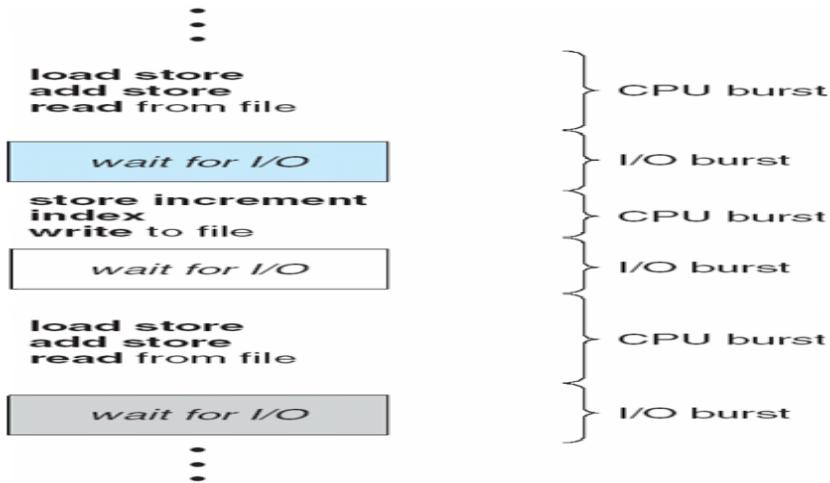
## CPU Scheduling

- To introduce CPU scheduling, which is the basis for multiprogrammed operating systems
- To describe various CPU-scheduling algorithms
- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system
- 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 distribution

## Histogram of CPU-burst Times



## Alternating Sequence of CPU And I/O Bursts



### CPU Scheduler

Selects from among the processes in memory that are ready to execute, 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

Scheduling under 1 and 4 is **nonpreemptive**

All other scheduling is **preemptive**

### Dispatcher

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

### Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process
- **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)
- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

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

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

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

The Gantt Chart for the schedule is:



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

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

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

*Convoy effect* short process behind long process



### Shortest-Job-First (SJF) Scheduling

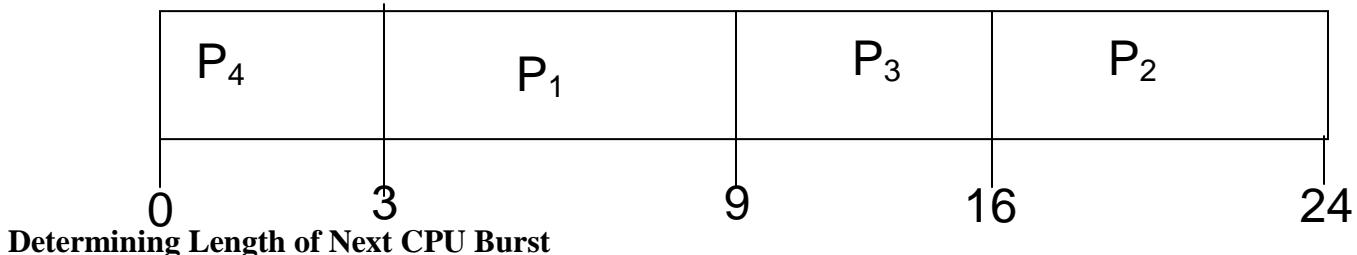
- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes

The difficulty is knowing

| <u>Process</u> | <u>Arrival Time</u> | <u>Burst Time</u> |
|----------------|---------------------|-------------------|
| $P_1$          | 0.0                 | 6                 |
| $P_2$          | 2.0                 | 8                 |
| $P_3$          | 4.0                 | 7                 |
| $P_4$          | 5.0                 | 3                 |

SJF scheduling chart

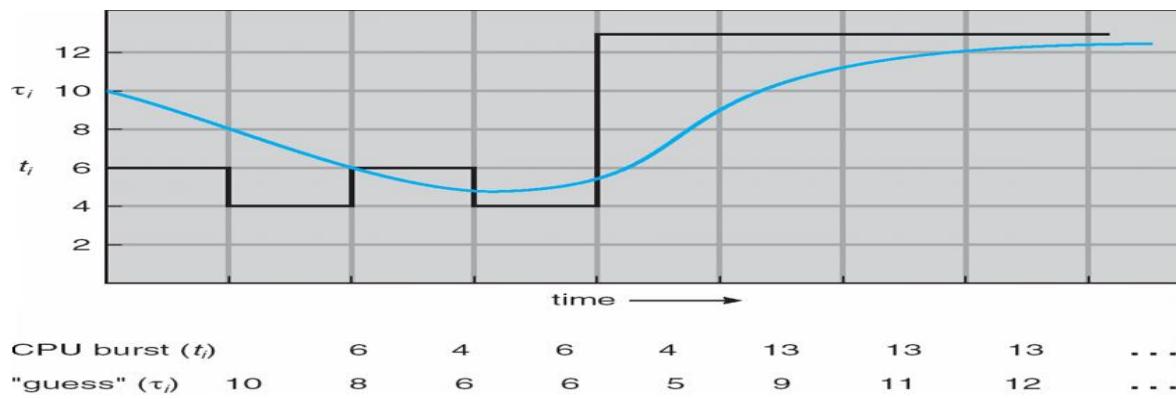
average waiting time =  $(3 + 16 + 9 + 0) / 4 = 7$  the length of the next CPU request



1.  $t_n$  = actual length of  $n^{th}$  CPU burst
2.  $\tau_{n+1}$  = predicted value for the next CPU burst
3.  $\alpha, 0 \leq \alpha \leq 1$
4. Define :

- Can only estimate the length
- Can be done by using the length of previous CPU bursts, using exponential averaging

#### Prediction of the Length of the Next CPU Burst



#### Examples of Exponential Averaging

$a = 0$

$t_{n+1} = t_n$

Recent history does not count

$a = 1$

$t_{n+1} = a t_n$

Only the actual last CPU burst counts

If we expand the formula, we get:

$t_{n+1} = a t_n + (1 - a)a t_{n-1} + \dots$

$+ (1 - a)^j a t_{n-j} + \dots$

$$+(1 - a)^n + 1 t_0$$

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

### Priority Scheduling

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

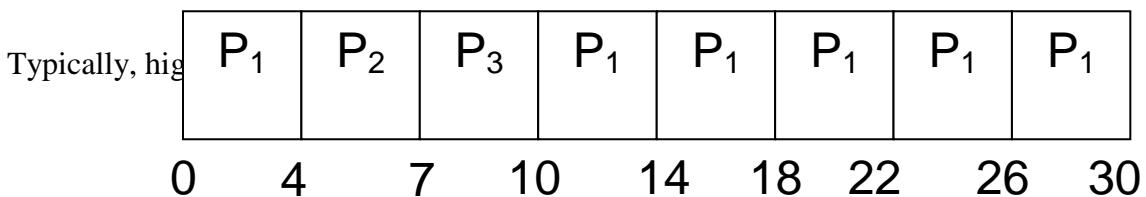
### Round Robin (RR)

- Each process gets a small unit of CPU time (*time quantum*), 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.
- Performance
- $q$  large  $\rightarrow$  FIFO
- $q$  small  $\rightarrow$   $q$  must be large with respect to context switch, otherwise overhead is too high

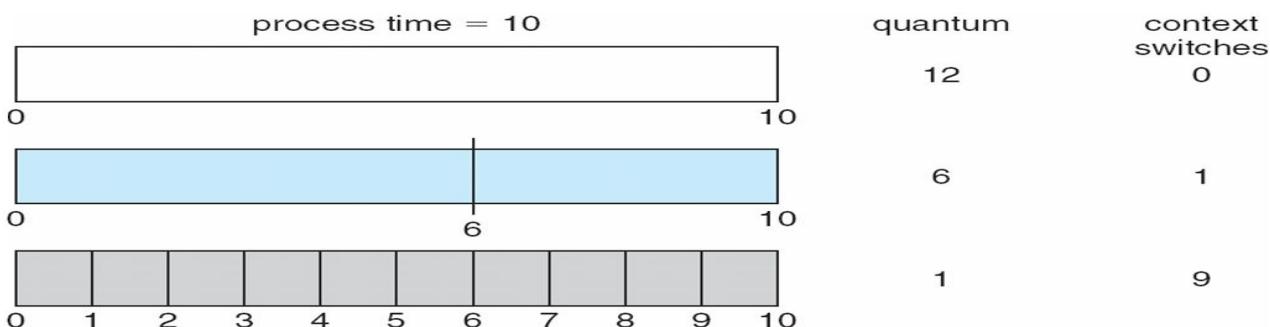
### Example of RR with Time Quantum = 4

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

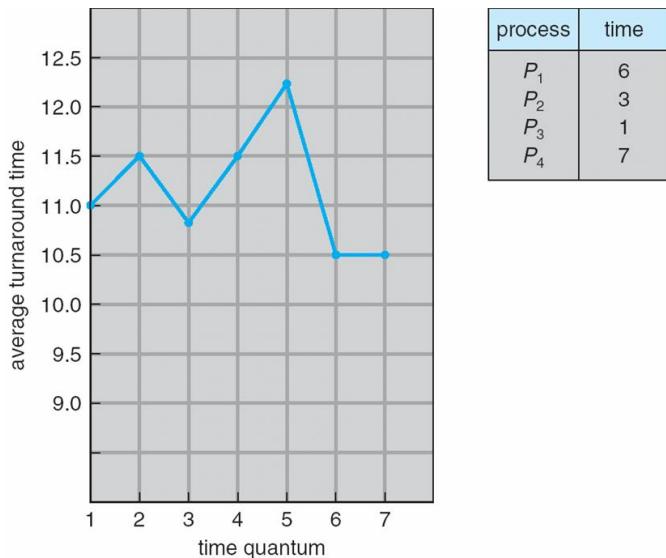
The Gantt chart is:



### Time Quantum and Context Switch Time



### Turnaround Time Varies With The Time Quantum

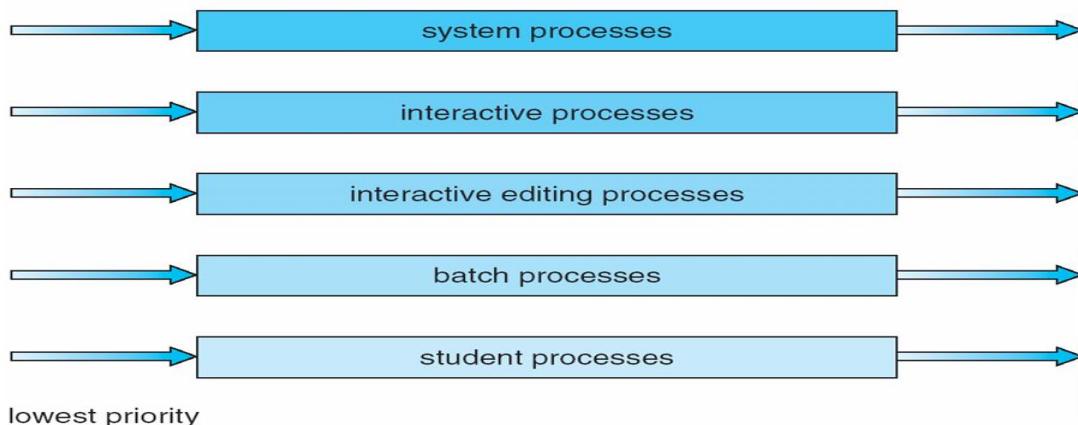


### Multilevel Queue

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

### Multilevel Queue Scheduling

highest priority



### Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process

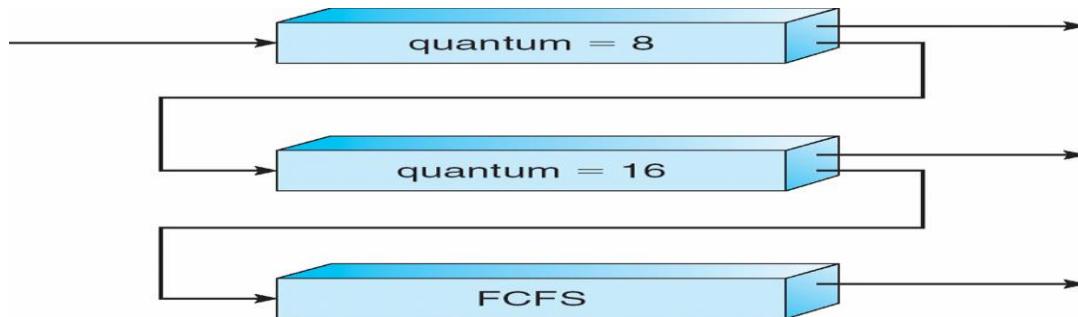
method used to determine which queue a process will enter when that process needs service

### Example of Multilevel Feedback Queue

Three queues:

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

### Multilevel Feedback Queues



### Thread Scheduling

- Distinction between user-level and kernel-level threads
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
- Known as **process-contention scope (PCS)** since scheduling competition is within the process
- Kernel thread scheduled onto available CPU is **system-contention scope (SCS)** – competition among all threads in system

### Pthread Scheduling

- API allows specifying either PCS or SCS during thread creation
- PTHREAD\_SCOPE\_PROCESS schedules threads using PCS scheduling
- PTHREAD\_SCOPE\_SYSTEM schedules threads using SCS scheduling.

### Pthread Scheduling API

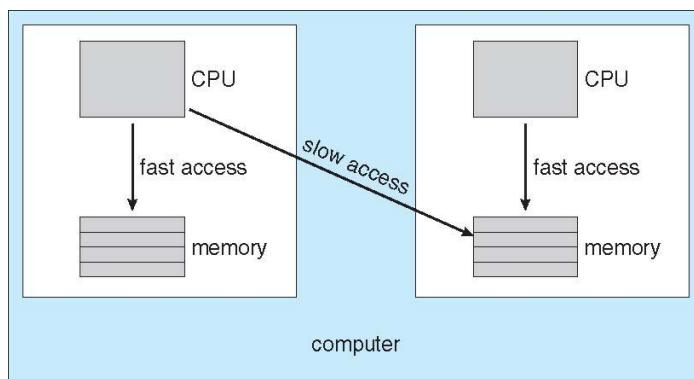
```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[])
```

```
{  
    int i;  pthread_t tid[NUM_THREADS];  
    pthread_attr_t attr;  
    /* get the default attributes */  
    pthread_attr_init(&attr);  
    /* set the scheduling algorithm to PROCESS or SYSTEM */  
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);  
    /* set the scheduling policy - FIFO, RT, or OTHER */  
    pthread_attr_setschedpolicy(&attr, SCHED_OTHER);  
    /* 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)  
{  
    printf("I am a thread\n");  
    pthread_exit(0);  
}
```

### Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- **Homogeneous processors** within a multiprocessor
- **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing
- **Symmetric multiprocessing (SMP)** – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
- **Processor affinity** – process has affinity for processor on which it is currently running
- **soft affinity**
- **hard affinity**

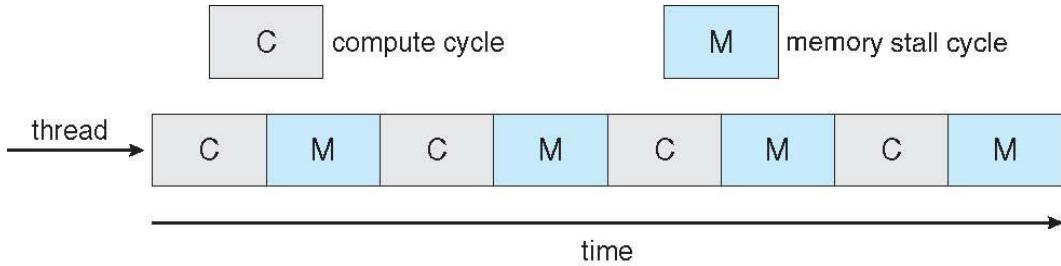
### NUMA and CPU Scheduling



### Multicore Processors

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

### Multithreaded Multicore System



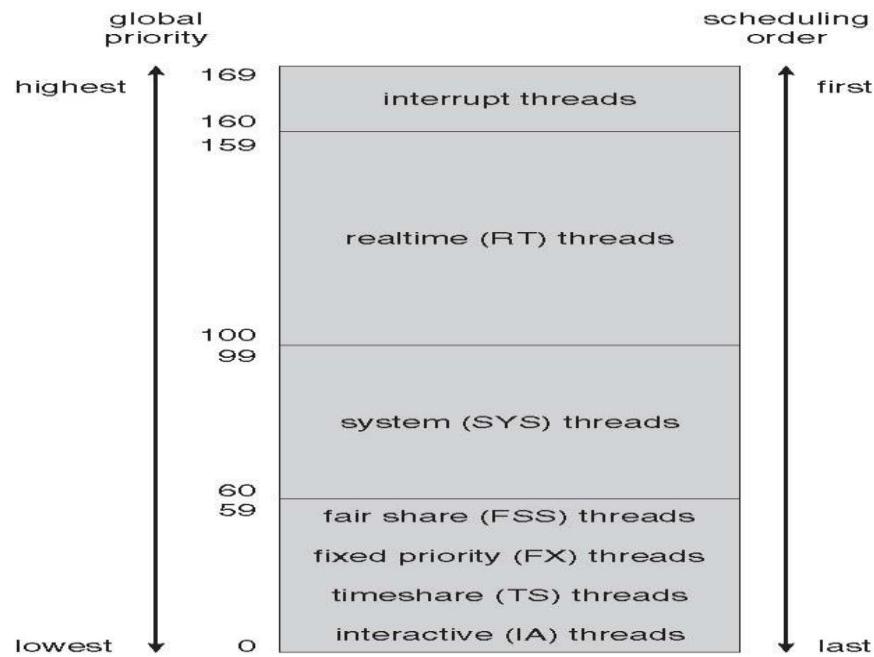
### Operating System Examples

- Solaris scheduling
- Windows XP scheduling
- Linux scheduling

### Solaris Dispatch Table

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

### Solaris Scheduling



### Windows XP Priorities

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

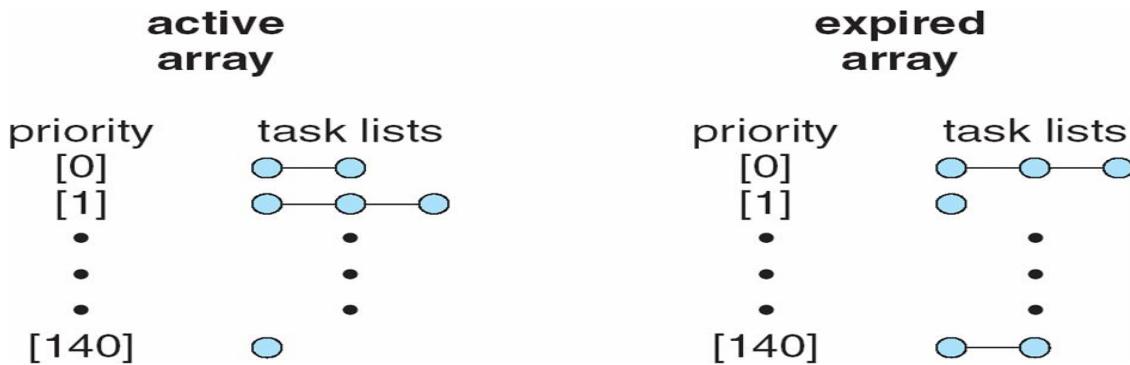
### Linux Scheduling

- Constant order  $O(1)$  scheduling time
- Two priority ranges: time-sharing and real-time
- **Real-time** range from 0 to 99 and **nice** value from 100 to 140

### Priorities and Time-slice length

| numeric priority | relative priority |                 | time quantum |
|------------------|-------------------|-----------------|--------------|
| 0                | highest           | real-time tasks | 200 ms       |
| •                |                   |                 |              |
| •                |                   |                 |              |
| •                |                   |                 |              |
| 99               |                   |                 |              |
| 100              |                   |                 |              |
| •                |                   |                 |              |
| •                |                   |                 |              |
| •                |                   |                 |              |
| 140              | lowest            | other tasks     | 10 ms        |

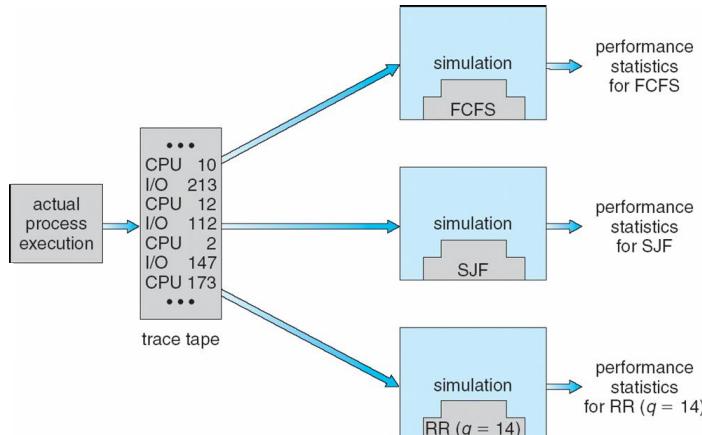
### List of Tasks Indexed According to Priorities



### Algorithm Evaluation

- Deterministic modeling – takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Queuing models
- Implementation

### Evaluation of CPU schedulers by Simulation



## UNIT -3 CONCURRENCY

### **Process Synchronization**

- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem
- To introduce the concept of an atomic transaction and describe mechanisms to ensure atomicity
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having an integer count that keeps track of the number of full buffers. Initially, count is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer

### **Producer**

```
while (true) {
```

```
    /* produce an item and put in nextProduced */  
    while (count == BUFFER_SIZE)  
        ; // do nothing  
    buffer [in] = nextProduced;  
    in = (in + 1) % BUFFER_SIZE;  
    count++;
```

```
}
```

### **Consumer**

```
while (true) {  
    while (count == 0)  
        ; // do nothing  
    nextConsumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    count--;  
    /* consume the item in nextConsumed
```

```
}
```

### **Race Condition**

count++ could be implemented as

```
register1 = count  
register1 = register1 + 1  
count = register1
```

count-- could be implemented as

```
register2 = count  
register2 = register2 - 1  
count = register2
```

Consider this execution interleaving with “count = 5” initially:

```
S0: producer execute register1 = count {register1 = 5}
S1: producer execute register1 = register1 + 1 {register1 = 6}
S2: consumer execute register2 = count {register2 = 5}
S3: consumer execute register2 = register2 - 1 {register2 = 4}
S4: producer execute count = register1 {count = 6 }
S5: consumer execute count = register2 {count = 4}
```

## Solution to Critical-Section Problem

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 there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next 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  
Assume that each process executes at a nonzero speed  
No assumption concerning relative speed of the N processes

## Peterson's Solution

- Two process solution
- Assume that the LOAD and STORE 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.  $\text{flag}[i] = \text{true}$  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);
```

## Synchronization Hardware

- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
- Currently running code would execute without preemption
- Generally too inefficient on multiprocessor systems
  - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
  - Atomic = non-interruptable
- 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);
```

### TestAndSet Instruction

Definition:

```
boolean TestAndSet (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
```

### Solution using TestAndSet

Shared boolean variable lock., initialized to false.

Solution:

```
do {
    while ( TestAndSet (&lock ) )
        ; // do nothing
        // critical section
    lock = FALSE;
        // remainder section
} while (TRUE);
```

### Swap Instruction

Definition:

```
void Swap (boolean *a, boolean *b)
{
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```

### Solution using Swap

Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key

Solution:

```
do {
    key = TRUE;
    while ( key == TRUE)
        Swap (&lock, &key );

        // critical section
    lock = FALSE;
```

```
// remainder section
} while (TRUE);
```

### Bounded-waiting Mutual Exclusion with TestandSet()

```
do {
    waiting[i] = TRUE;
    key = TRUE;
    while (waiting[i] && key)
        key = TestAndSet(&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);
```

### Semaphore

- Synchronization tool that does not require busy waiting
- nSemaphore – integer variable
- Two standard operations modify S: wait() and signal()
- Originally called P() and V()
- Less complicated
- Can only be accessed via two indivisible (atomic) operations

```
wait (S) {
    while S <= 0
        ; // no-op
    S--;
}
signal (S) {
    S++;
}
```

### Semaphore as General Synchronization Tool

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
- Also known as mutex locks
- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusionSemaphore mutex; // initialized to do {  
 wait (mutex);  
 // Critical Section  
 signal (mutex);  
 // remainder section

} while (TRUE);

### Semaphore Implementation

- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section.
- Could now have busy waiting in critical section implementation

But implementation code is short

Little busy waiting if critical section rarely occupied

- Note that applications may spend lots of time in critical sections and therefore this is not a good solution.

### Semaphore Implementation with no Busy waiting

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

Implementation of wait:

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

Implementation of signal:

```
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

|                                          |                                          |
|------------------------------------------|------------------------------------------|
| $P_0$<br>wait (S);<br>wait (Q);<br><br>. | $P_1$<br>wait (Q);<br>wait (S);<br><br>. |
|------------------------------------------|------------------------------------------|

|                            |                            |
|----------------------------|----------------------------|
| signal (S);<br>signal (Q); | signal (Q);<br>signal (S); |
|----------------------------|----------------------------|

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

### Classical Problems of Synchronization

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

#### Bounded-Buffer Problem

- $N$  buffers, each can hold one item
  - Semaphore mutex initialized to the value 1
  - Semaphore full initialized to the value 0
  - Semaphore empty initialized to the value N.
  - The structure of the producer process
- ```
do {           // produce an item in nextp
    wait (empty);
    wait (mutex);
        // add the item to the buffer
    signal (mutex);
    signal (full);
} while (TRUE);
```

The structure of the consumer process

```
do {           wait (full);
    wait (mutex);
        // remove an item from buffer to nextc
    signal (mutex);
    signal (empty);
}
// consume the item in nextc
} 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 – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time
- Shared Data

- Data set
- Semaphore mutex initialized to 1
- Semaphore wrt initialized to 1
- Integer readcount initialized to 0

The structure of a writer process

```
do {           wait (wrt) ;

    // writing is performed
    signal (wrt) ;
} while (TRUE);
```

The structure of a reader process

```
do {
    wait (mutex) ;
    readcount ++ ;
    if (readcount == 1)
        wait (wrt) ;
    signal (mutex)

    // reading is performed
    wait (mutex) ;
    readcount -- ;
    if (readcount == 0)
        signal (wrt) ;
    signal (mutex) ;
} while (TRUE);
```

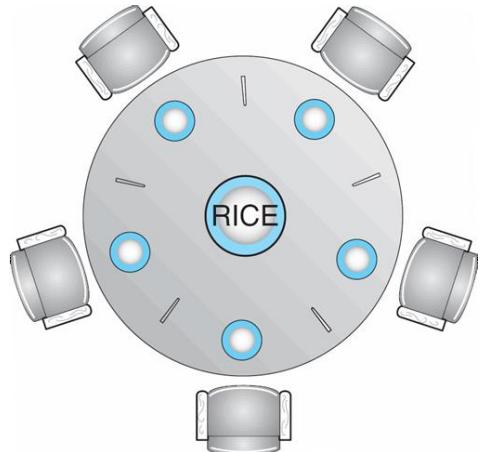
### Dining-Philosophers Problem

- Shared data
- Bowl of rice (data set)
- Semaphore chopstick [5] initialized to 1
- The 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);
```



### Problems with Semaphores

Incorrect use of semaphore operations:

1 signal (mutex)

....

wait (mutex)

wait (mutex) ...

wait (mutex)

Omitting of wait (mutex) or signal (mutex) (or both)

### Monitors

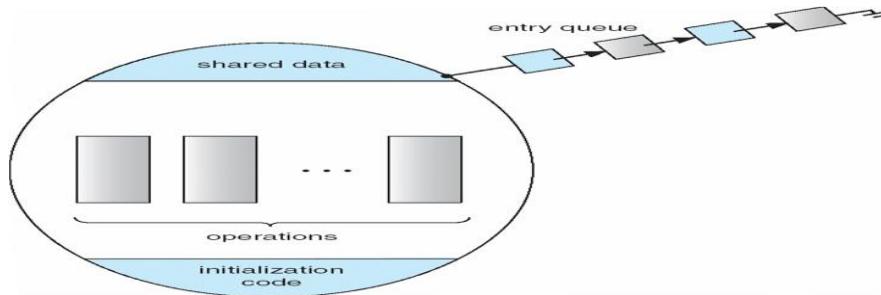
A high-level abstraction that provides a convenient and effective mechanism for process synchronization

Only one process may be active within the monitor at a time

monitor monitor-name

```
{  
    // shared variable declarations  
    procedure P1 (...) { .... }  
    ...  
    procedure Pn (...) {.....}  
    Initialization code ( .... ) { ... }  
    ...  
}
```

### Schematic view of a Monitor



### Condition Variables

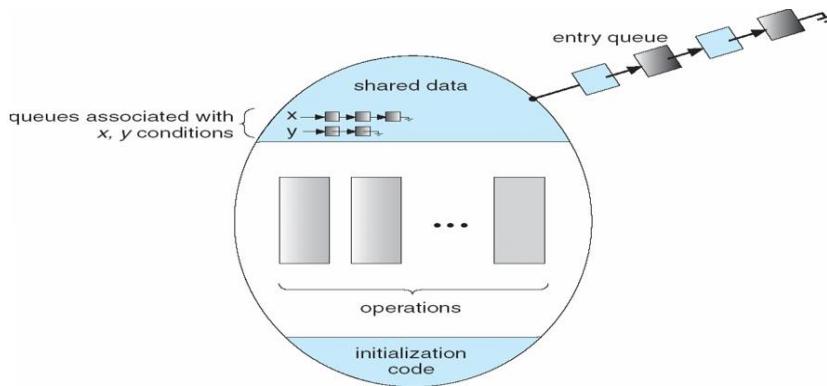
condition x, y;

Two operations on a condition variable:

x.wait () – a process that invokes the operation is suspended.

x.signal () – resumes one of processes (if any) that invoked x.wait ()

## Monitor with Condition Variables



## Solution to Dining Philosophers

```

monitor DP
{
    enum { THINKING; HUNGRY, EATING } state [5] ;
    condition self [5];
    void pickup (int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING) self [i].wait;
    }

    void putdown (int i) {
        state[i] = THINKING;
        // test left and right neighbors
        test((i + 4) % 5);
        test((i + 1) % 5);
    }

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

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

```

Each philosopher  $I$  invokes the operations pickup()  
and putdown() in the following sequence:

```
DiningPhilosophers.pickup (i);
    EAT
DiningPhilosophers.putdown (i);
```

## Monitor Implementation Using Semaphores

### Variables

```
semaphore mutex; // (initially = 1)
semaphore next; // (initially = 0)
int next_count = 0;nEach procedure  $F$  will be replaced by
    wait(mutex);
    ...
body of  $F$ ;
    ...
if (next_count > 0)
    signal(next)
else
    signal(mutex);nMutual exclusion within a monitor is ensured.
```

### Monitor Implementation

For each condition variable  $x$ , we have:

```
semaphore x_sem; // (initially = 0)
int x_count = 0;nThe operation x.wait can be implemented as:
```

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

The operation x.signal can be implemented as:

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

### A Monitor to Allocate Single Resource

monitor ResourceAllocator

```
{
    boolean busy;
    condition x;
    void acquire(int time) {
```

```
        if (busy)
            x.wait(time);
        busy = TRUE;
    }
    void release() {
        busy = FALSE;
        x.signal();
    }
initialization code() {
    busy = FALSE;
}
}
```

### Synchronization Examples

- Solaris
- Windows XP
- Linux
- Pthreads

#### Solaris Synchronization

- **Implements a variety of locks to support** multitasking, multithreading (including real-time threads), and multiprocessing
- Uses adaptive mutexes for efficiency when protecting data from short code segments
- Uses condition variables and readers-writers locks when longer sections of code need access to data
- Uses turnstiles to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock

#### Windows XP Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses spinlocks on multiprocessor systems
- Also provides dispatcher objects which may act as either mutexes and semaphores
- Dispatcher objects may also provide events
- An event acts much like a condition variable

#### Linux Synchronization

- Linux: Prior to kernel Version 2.6, disables interrupts to implement short critical sections
- Version 2.6 and later, fully preemptive
- Linux provides:
- semaphores
- spin locks

#### Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
- mutex locks
- condition variables
- Non-portable extensions include:
- read-write locks

- spin locks

### Atomic Transactions

- System Model
- Log-based Recovery
- Checkpoints
- Concurrent Atomic Transactions

### System Model

- Assures that operations happen as a single logical unit of work, in its entirety, or not at all
- Related to field of database systems
- Challenge is assuring atomicity despite computer system failures
- Transaction - collection of instructions or operations that performs single logical function
- Here we are concerned with changes to stable storage – disk
- Transaction is series of read and write operations
- Terminated by commit (transaction successful) or abort (transaction failed) operation Aborted transaction must be rolled back to undo any changes it performed

### Types of Storage Media

- Volatile storage – information stored here does not survive system crashes
- Example: main memory, cache
- Nonvolatile storage – Information usually survives crashes
- Example: disk and tape
- Stable storage – Information never lost
- Not actually possible, so approximated via replication or RAID to devices with independent failure modes
- Goal is to assure transaction atomicity where failures cause loss of information on volatile storage

### Log-Based Recovery

- Record to stable storage information about all modifications by a transaction
- Most common is write-ahead logging
- Log on stable storage, each log record describes single transaction write operation, including  
    Transaction name  
    Data item name  
    Old value  
    New value
- $\langle T_i \text{ starts} \rangle$  written to log when transaction  $T_i$  starts
- $\langle T_i \text{ commits} \rangle$  written when  $T_i$  commits
- Log entry must reach stable storage before operation on data occurs

### Log-Based Recovery Algorithm

Using the log, system can handle any volatile memory errors

- Undo( $T_i$ ) restores value of all data updated by  $T_i$
- Redo( $T_i$ ) sets values of all data in transaction  $T_i$  to new values
- Undo( $T_i$ ) and redo( $T_i$ ) must be idempotent
- Multiple executions must have the same result as one execution

- If system fails, restore state of all updated data via log
- If log contains  $\langle T_i \text{ starts} \rangle$  without  $\langle T_i \text{ commits} \rangle$ , undo( $T_i$ )
- If log contains  $\langle T_i \text{ starts} \rangle$  and  $\langle T_i \text{ commits} \rangle$ , redo( $T_i$ )

## Checkpoints

Log could become long, and recovery could take long

Checkpoints shorten log and recovery time.

Checkpoint scheme:

1. Output all log records currently in volatile storage to stable storage
2. Output all modified data from volatile to stable storage
3. Output a log record  $\langle \text{checkpoint} \rangle$  to the log on stable storage

Now recovery only includes  $T_i$ , such that  $T_i$  started executing before the most recent checkpoint, and all transactions after  $T_i$  All other transactions already on stable storage

## Concurrent Transactions

- Must be equivalent to serial execution – serializability
- Could perform all transactions in critical section
- Inefficient, too restrictive
- Concurrency-control algorithms provide serializability

## Serializability

- Consider two data items A and B
- Consider Transactions  $T_0$  and  $T_1$
- Execute  $T_0, T_1$  atomically
- Execution sequence called schedule
- Atomically executed transaction order called serial schedule
- For N transactions, there are  $N!$  valid serial schedules

### Schedule 1: $T_0$ then $T_1$

|   | $T_0$                        | $T_1$                                                                 |
|---|------------------------------|-----------------------------------------------------------------------|
|   | $\text{read}(A)$             |                                                                       |
|   | $\text{write}(A)$            |                                                                       |
|   | $\text{read}(B)$             | overlapped execute                                                    |
|   | $\text{write}(B)$            | necessarily incorrect                                                 |
|   | $\text{read}(A)$             | operations $O_i, O_j$                                                 |
|   | $\text{write}(A)$            | a item, with at least one write                                       |
|   | $\text{read}(B)$             | operations of different transactions & $O_i$ and $O_j$ don't conflict |
|   | $\text{write}(B)$            | or $O_j$ $O_i$ equivalent to S                                        |
|   |                              | lapping nonconflicting operations                                     |
| • | $S$ is conflict serializable |                                                                       |

## Schedule 2: Concurrent Serializable Schedule

| $T_0$                             | $T_1$                             |
|-----------------------------------|-----------------------------------|
| <b>read(A)</b><br><b>write(A)</b> | <b>read(A)</b><br><b>write(A)</b> |
| <b>read(B)</b><br><b>write(B)</b> | <b>read(B)</b><br><b>write(B)</b> |

## Locking Protocol

- Ensure serializability by associating lock with each data item
- Follow locking protocol for access control
- Locks
- Shared –  $T_i$  has shared-mode lock (S) on item Q,  $T_i$  can read Q but not write Q
- Exclusive –  $T_i$  has exclusive-mode lock (X) on Q,  $T_i$  can read and write Q
- Require every transaction on item Q acquire appropriate lock
- If lock already held, new request may have to wait
- Similar to readers-writers algorithm

## Two-phase Locking Protocol

- Generally ensures conflict serializability
- Each transaction issues lock and unlock requests in two phases
- Growing – obtaining locks
- Shrinking – releasing locks
- Does not prevent deadlock

## Timestamp-based Protocols

- Select order among transactions in advance – timestamp-ordering
- Transaction  $T_i$  associated with timestamp  $TS(T_i)$  before  $T_i$  starts
- $TS(T_i) < TS(T_j)$  if  $T_i$  entered system before  $T_j$
- $TS$  can be generated from system clock or as logical counter incremented at each entry of transaction
- Timestamps determine serializability order
- If  $TS(T_i) < TS(T_j)$ , system must ensure produced schedule equivalent to serial schedule where  $T_i$  appears before  $T_j$

## Timestamp-based Protocol Implementation

- Data item Q gets two timestamps
- W-timestamp(Q) – largest timestamp of any transaction that executed  $write(Q)$  successfully
- R-timestamp(Q) – largest timestamp of successful  $read(Q)$

- Updated whenever read(Q) or write(Q) executed
- Timestamp-ordering protocol assures any conflicting read and write executed in timestamp order

Suppose  $T_i$  executes read(Q)

If  $TS(T_i) < W\text{-timestamp}(Q)$ ,  $T_i$  needs to read value of Q that was already overwritten read operation rejected and  $T_i$  rolled back

If  $TS(T_i) \geq W\text{-timestamp}(Q)$  read executed,  $R\text{-timestamp}(Q)$  set to  $\max(R\text{-timestamp}(Q), TS(T_i))$

### **Timestamp-ordering Protocol**

Supose  $T_i$  executes write (Q)

If  $TS(T_i) < R\text{-timestamp}(Q)$ , value Q produced by  $T_i$  was needed previously and  $T_i$  assumed it would never be produced Write operation rejected,  $T_i$  rolled back If  $TS(T_i) < W\text{-tiimestamp}(Q)$ ,  $T_i$  attempting to write obsolete value of Q Write operation rejected and  $T_i$  rolled back Otherwise, write executed Any rolled back transaction  $T_i$  is assigned new timestamp and restarted Algorithm ensures conflict serializability and freedom from deadlock

### **Schedule Possible Under Timestamp Protocol**

| $T_2$   | $T_3$               |
|---------|---------------------|
| read(B) | read(B)<br>write(B) |
| read(A) | read(A)<br>write(A) |

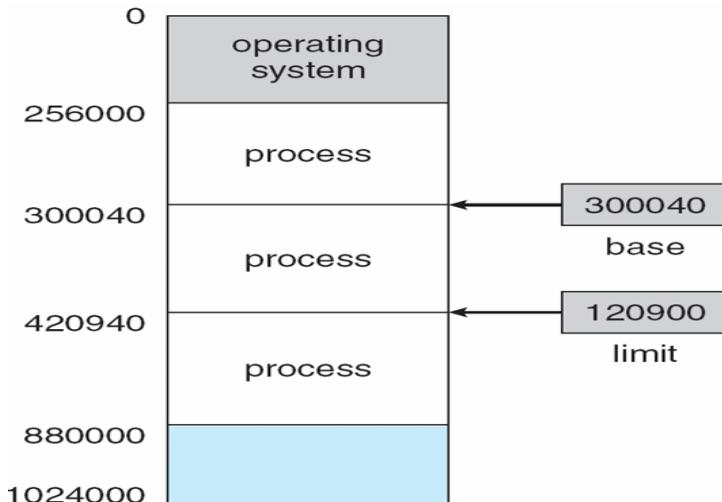
## UNIT IV

### Memory Management

- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques, including paging and segmentation
- To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Register access in one CPU clock (or less)
- Main memory can take many cycles
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

### **Base and Limit Registers**

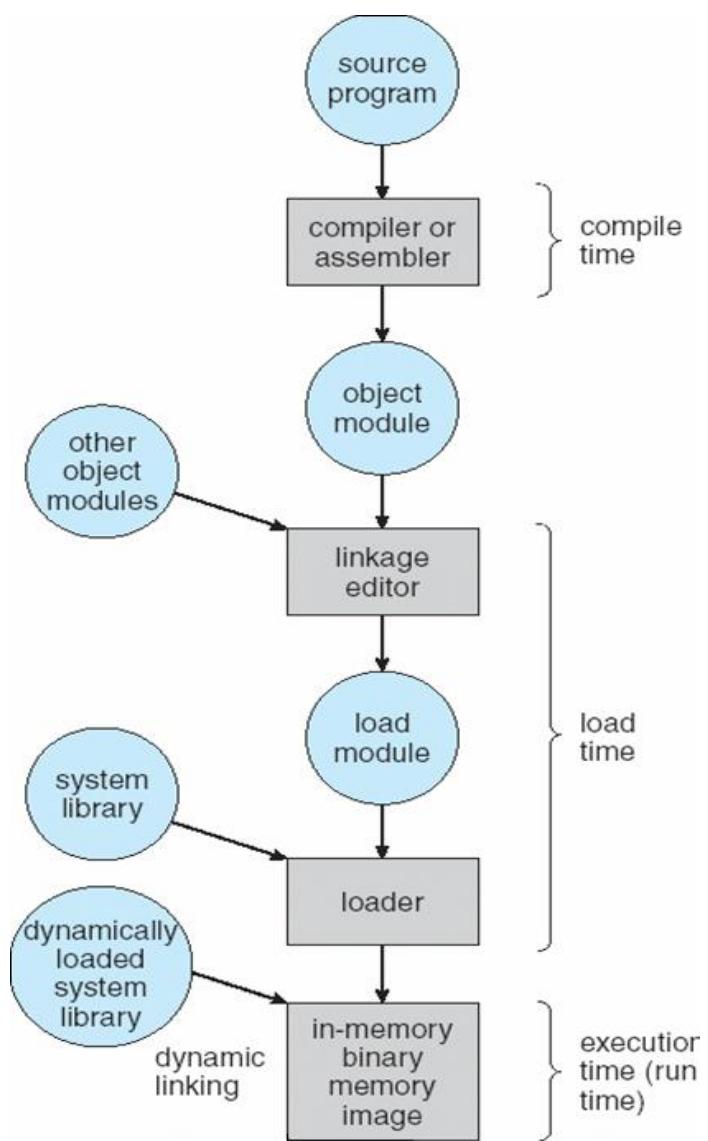
A pair of **base** and **limit** registers define the logical address space



### **Binding of Instructions and Data to Memory**

- Address binding of instructions and data to memory addresses can happen at three different stages
- **Compile time:** If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
- **Load time:** Must generate **relocatable code** if memory location is not known at compile time
- **Execution time:** Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., base and limit registers)

## Multistep Processing of a User Program



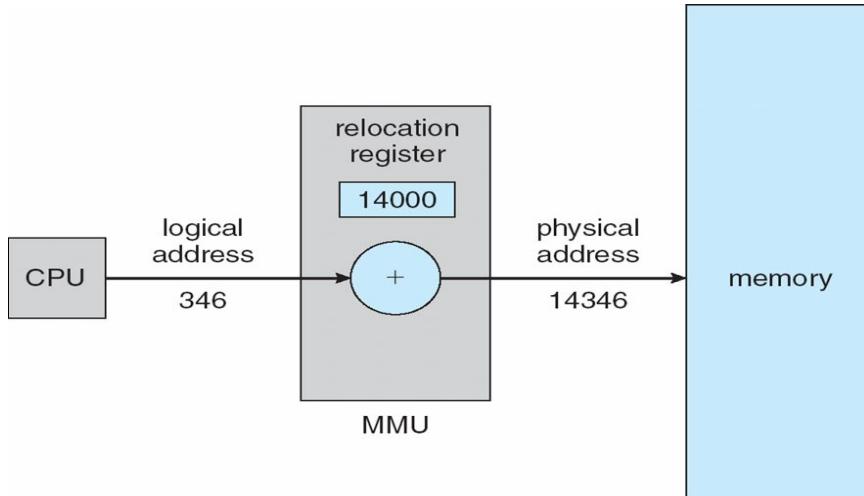
# Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as **virtual address**
  - **Physical address** – address seen by the memory unit
  - Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme

## Memory-Management Unit (MMU)

- Hardware device that maps virtual to physical address
- In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
- The user program deals with *logical* addresses; it never sees the *real* physical addresses

## Dynamic relocation using a relocation register



## Dynamic Loading

- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required implemented through program design

## Dynamic Linking

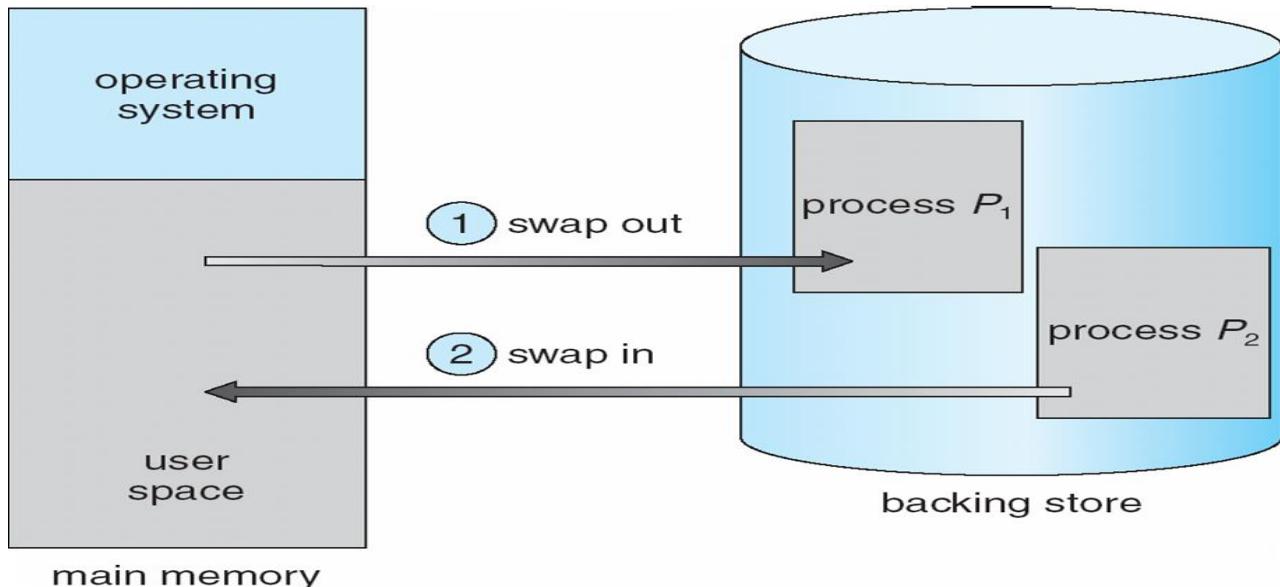
- Linking postponed until execution time
- Small piece of code, *stub*, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system needed to check if routine is in processes' memory address
- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**

## Swapping

A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution. **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images. **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed. Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped. Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows).

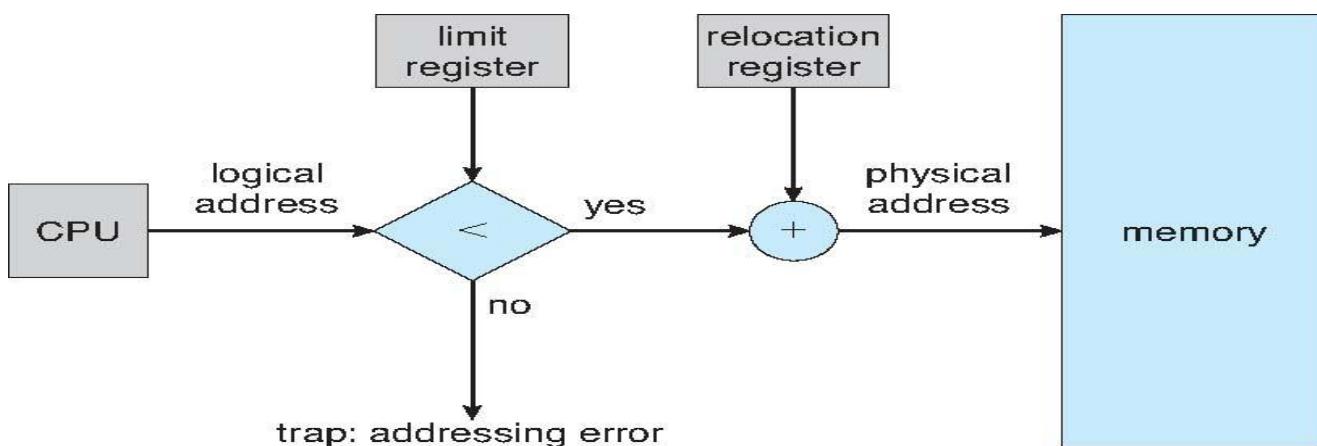
System maintains a **ready queue** of ready-to-run processes which have memory images on disk.

## Schematic View of Swapping



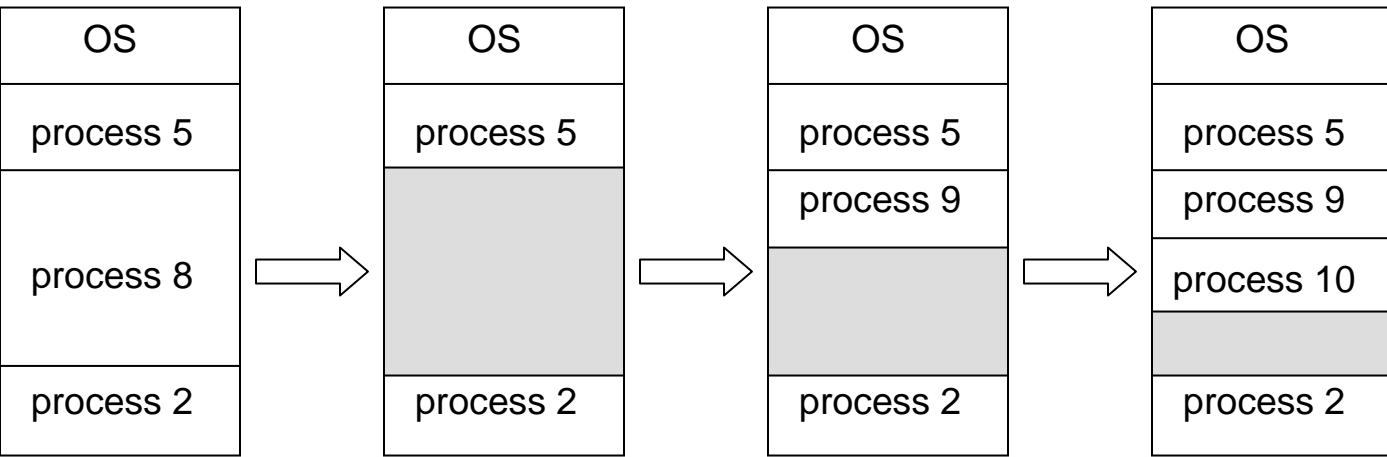
- Main memory usually into two partitions:
- Resident operating system, usually held in low memory with interrupt vector
- User processes then held in high memory
- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
- Base register contains 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*

## Hardware Support for Relocation and Limit Registers



- Multiple-partition allocation
- 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

- Operating system maintains information about:
- a) allocated partitions b) free partitions (hole)



### Dynamic Storage-Allocation Problem

- **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
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- Reduce external fragmentation by **compaction**
- Shuffle memory contents to place all free memory together in one large block
- Compaction is possible *only* if relocation is dynamic, and is done at execution time.
- I/O problem
  - Latch job in memory while it is involved in I/O
  - Do I/O only into OS buffers

### Paging

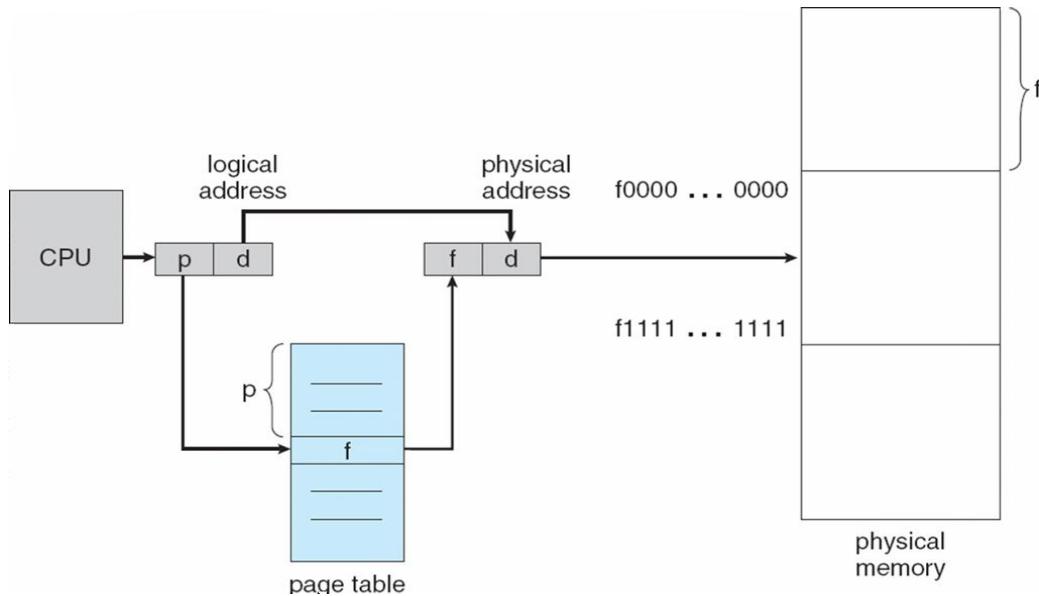
- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
- Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8,192 bytes)
- Divide logical memory into blocks of same size 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
- Internal fragmentation

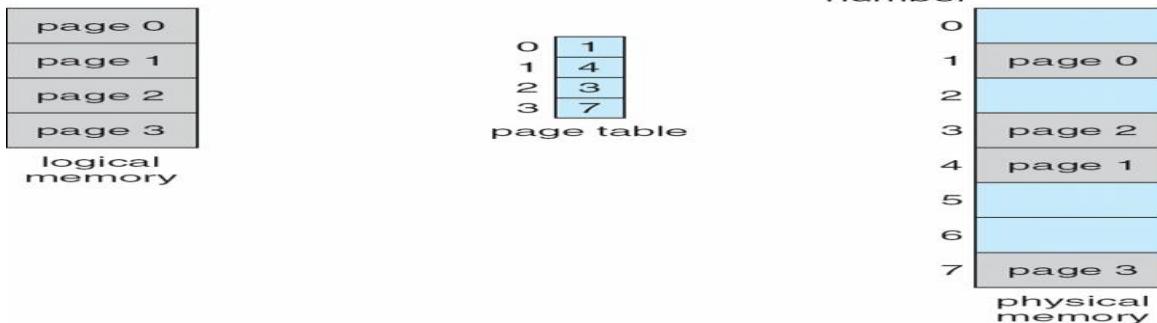
## Address Translation Scheme

- 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
- For given logical address space  $2^m$  and page size  $2^n$

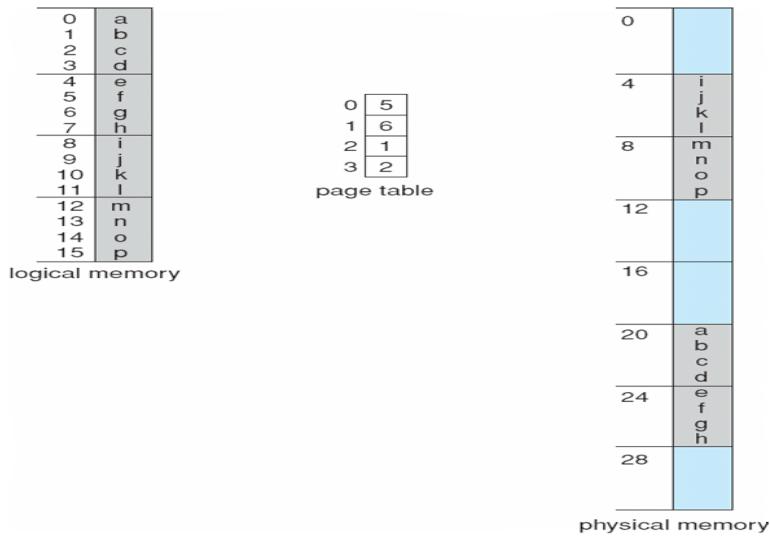
## Paging Hardware



## Paging Model of Logical and Physical Memory

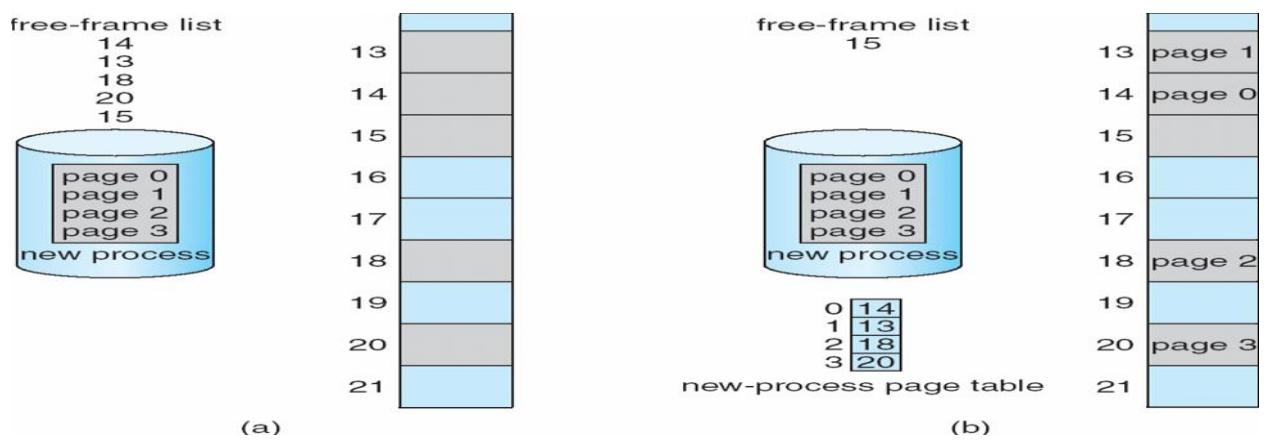


## Paging Example



32-byte memory and 4-byte pages

## Free Frames



## Implementation of Page Table

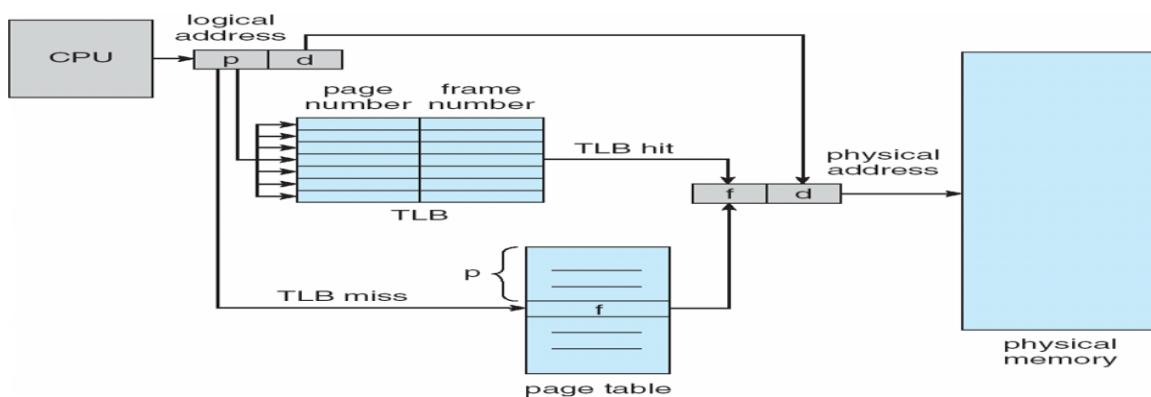
- Page table is kept in main memory
- **Page-table base register (PTBR)** points to the page table
- **Page-table length register (PRLR)** indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.
- 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)**
- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address-space protection for that process

### Associative Memory

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

| Page # | Frame # |
|--------|---------|
|        |         |
|        |         |
|        |         |
|        |         |

### Paging Hardware With TLB



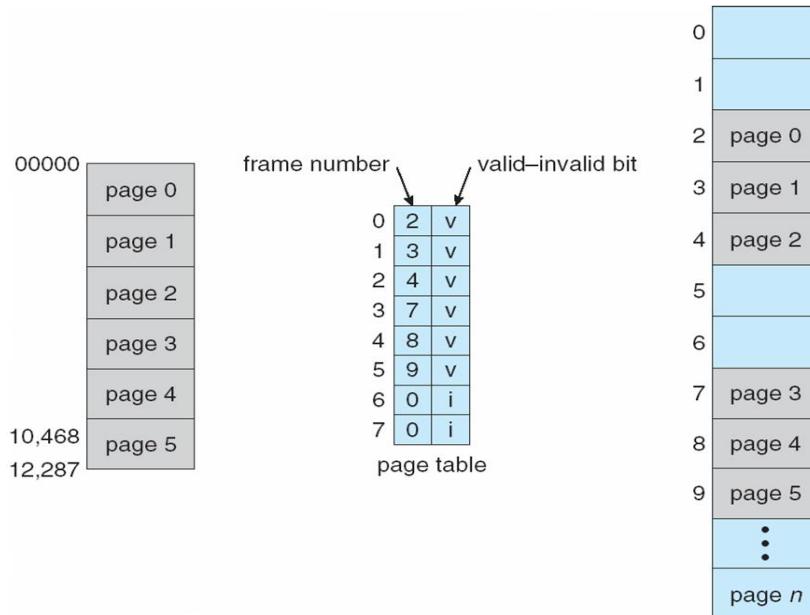
## Effective Access Time

- Associative Lookup =  $e$  time unit
- Assume memory cycle time is 1 microsecond
- Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- Hit ratio = an **Effective Access Time (EAT)**

$$\begin{aligned} \text{EAT} &= (1 + e) a + (2 + e)(1 - a) \\ &= 2 + e - a \end{aligned}$$

## Memory Protection

- Memory protection implemented by associating protection bit with each frame
- **Valid-invalid** bit attached to each entry in the page table:
- “valid” indicates that the associated page is in the process’ logical address space, and is thus a legal page
- “invalid” indicates that the page is not in the process’ logical address space
- **Valid (v) or Invalid (i) Bit In A Page Table**



## Shared Pages

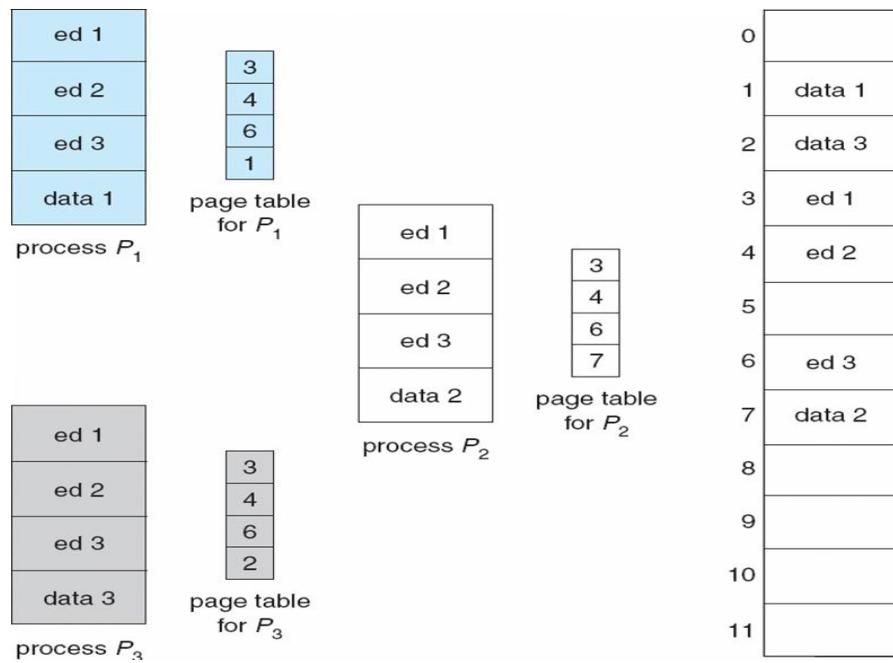
### Shared code

- One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
- Shared code must appear in same location in the logical address space of all processes

### Private code and data

- Each process keeps a separate copy of the code and data
- The pages for the private code and data can appear anywhere in the logical address space

## Shared Pages Example



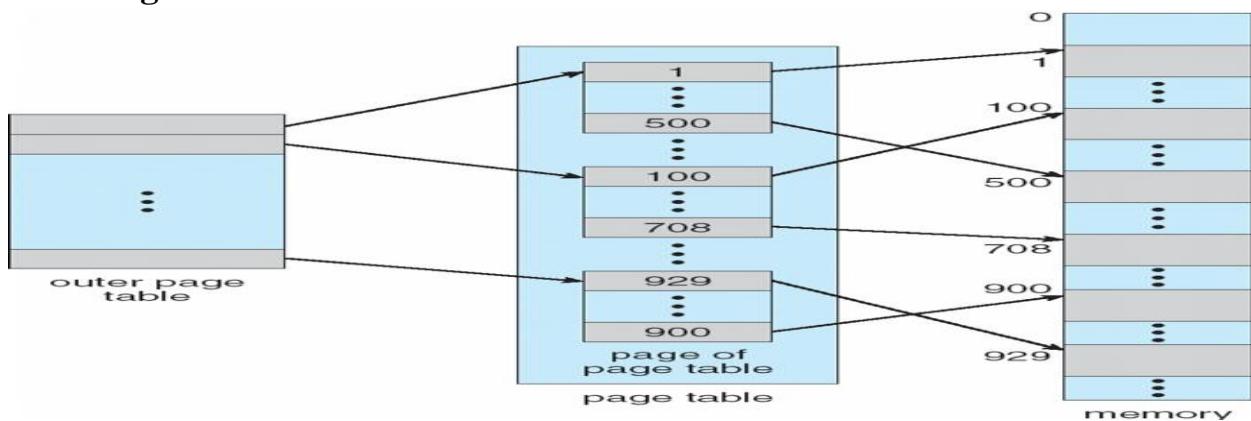
## Structure of the Page Table

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

## Hierarchical Page Tables

- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table

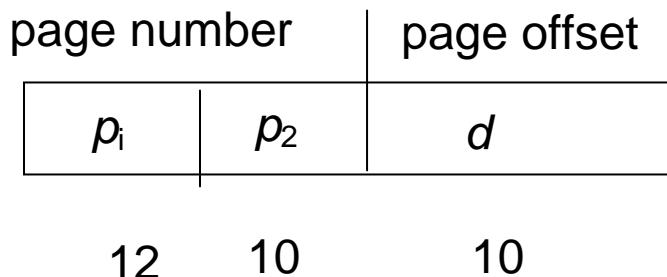
## Two-Level Page-Table Scheme



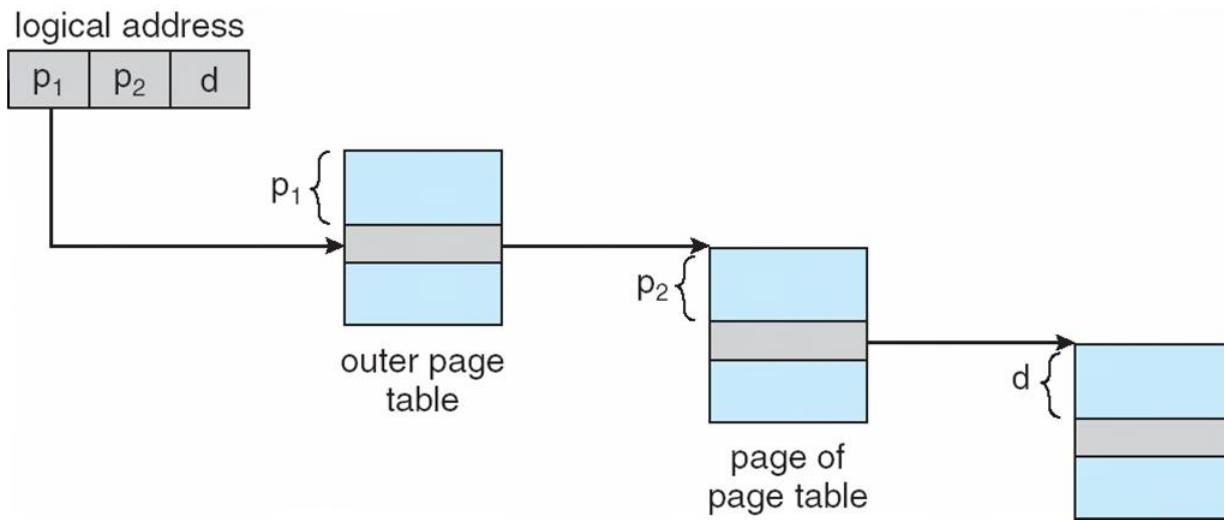
## Two-Level Paging Example

- A logical address (on 32-bit machine with 1K page size) is divided into:
- a page number consisting of 22 bits
- a page offset consisting of 10 bits
- Since the page table is paged, the page number is further divided into:
- a 12-bit page number
- a 10-bit page offset
- Thus, a logical address is as follows:

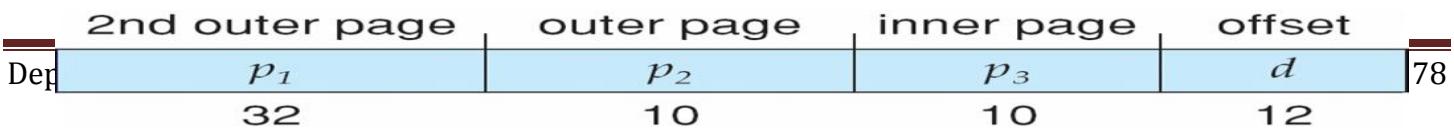
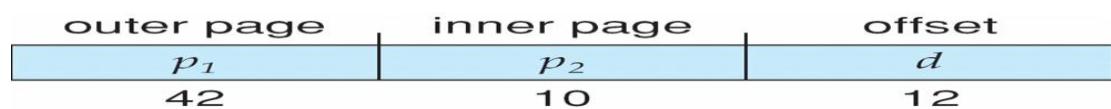
where  $p_1$  is an index into the outer page table, and  $p_2$  is the displacement within the page of the outer page table



## Address-Translation Scheme



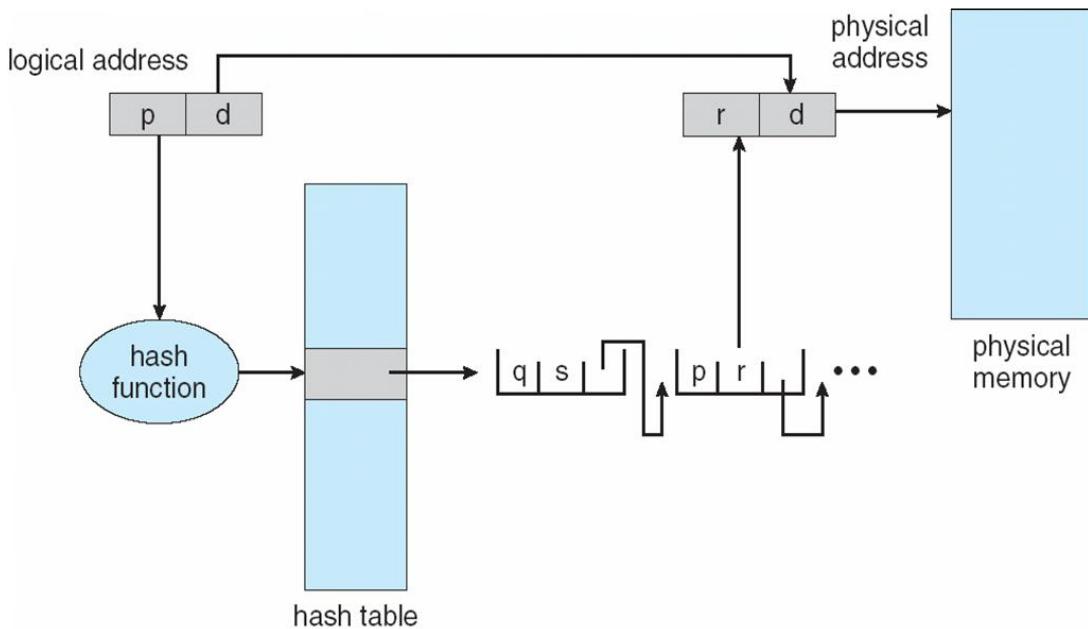
## Three-level Paging Scheme



## Hashed Page Tables

- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
- This page table contains a chain of elements hashing to the same location
- Virtual page numbers are compared in this chain searching for a match
- If a match is found, the corresponding physical frame is extracted

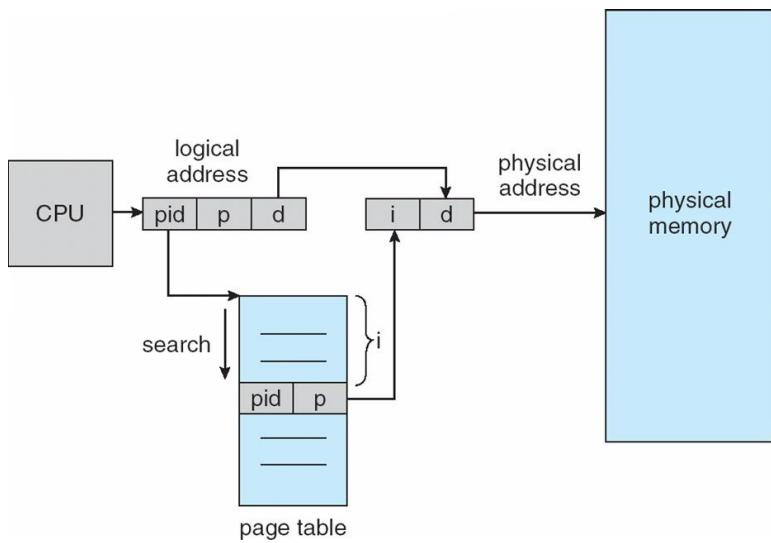
## Hashed Page Table



## Inverted Page Table

- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries

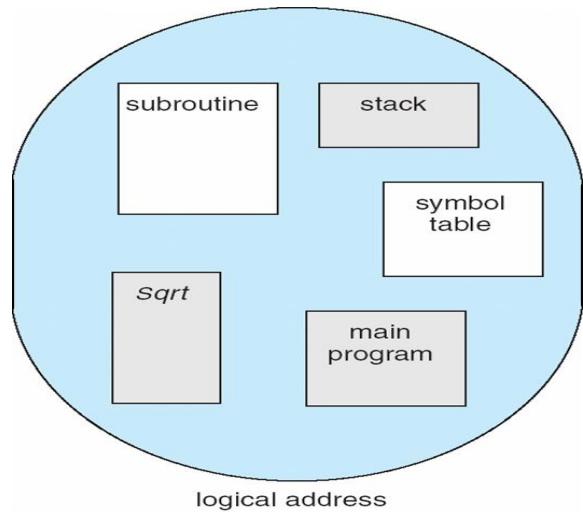
## Inverted Page Table Architecture



## Segmentation

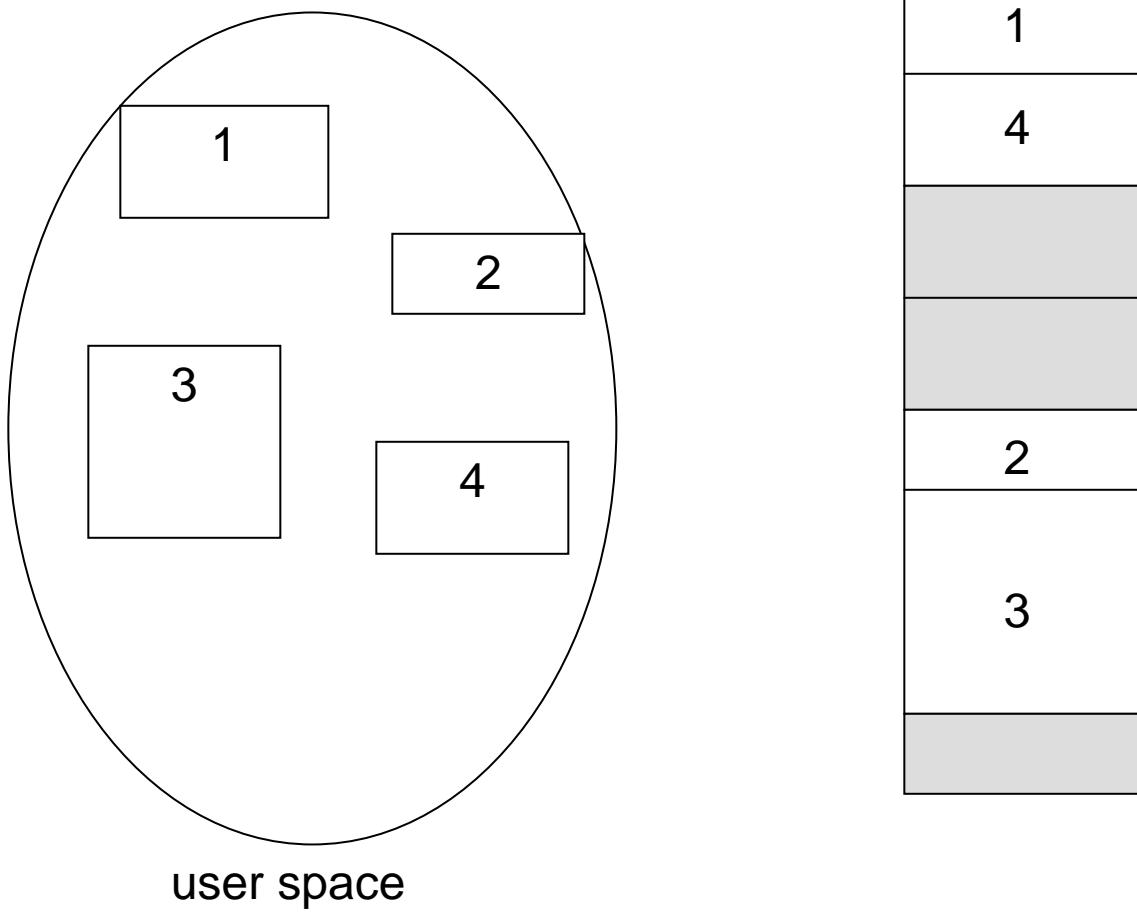
Memory-management scheme that supports user view of memory

- A program is a collection of segments
- A segment is a logical unit such as:
- main program
- procedure function
- method
- object
- local variables, global variables
- common block
- stack
- symbol table
- arrays



## User's View of a Program

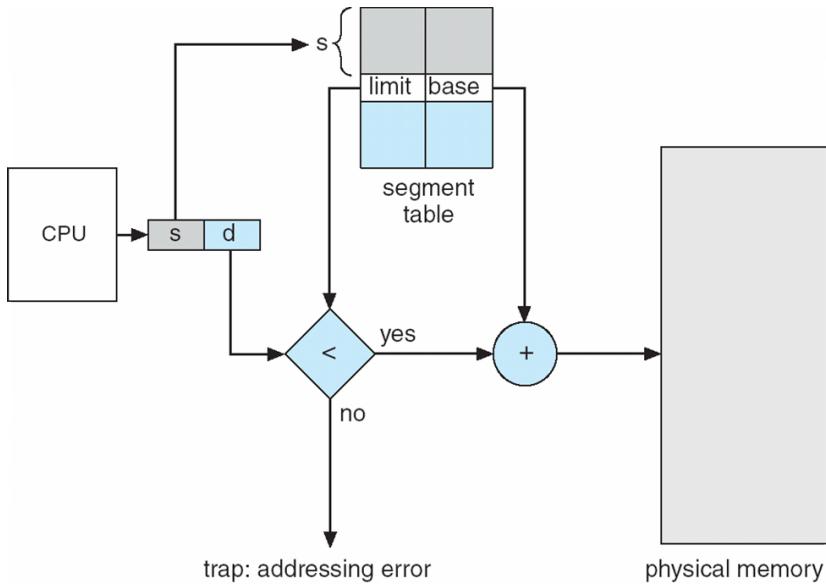
## Logical View of Segmentation



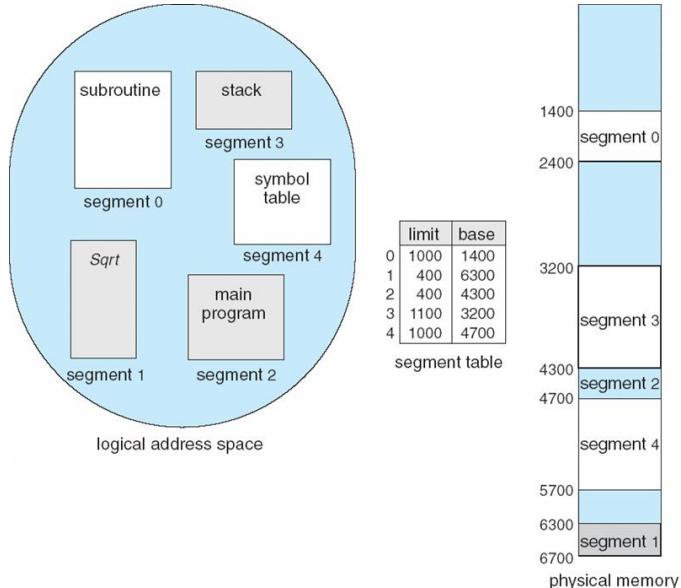
## Segmentation Architecture

- Logical address consists of a two tuple:
  - <segment-number, offset>
- **Segment table** – maps two-dimensional physical addresses; each entry has:
  - base – contains the starting physical address where the segments reside in memory
  - 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; segment number  $s$  is legal if  $s < \text{STLR}$
- Protection
- With each entry in segment table associate:
  - validation bit = 0 → illegal segment
  - read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem
- A segmentation example is shown in the following diagram

## Segmentation Hardware



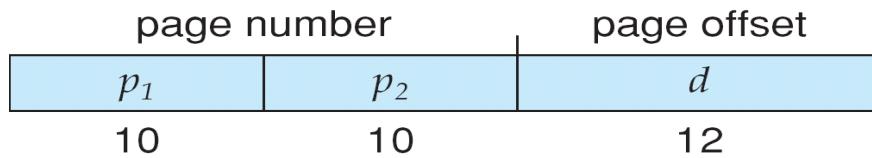
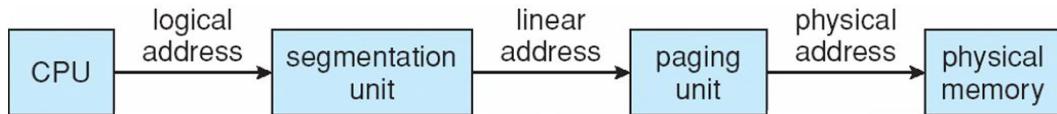
## Example of Segmentation



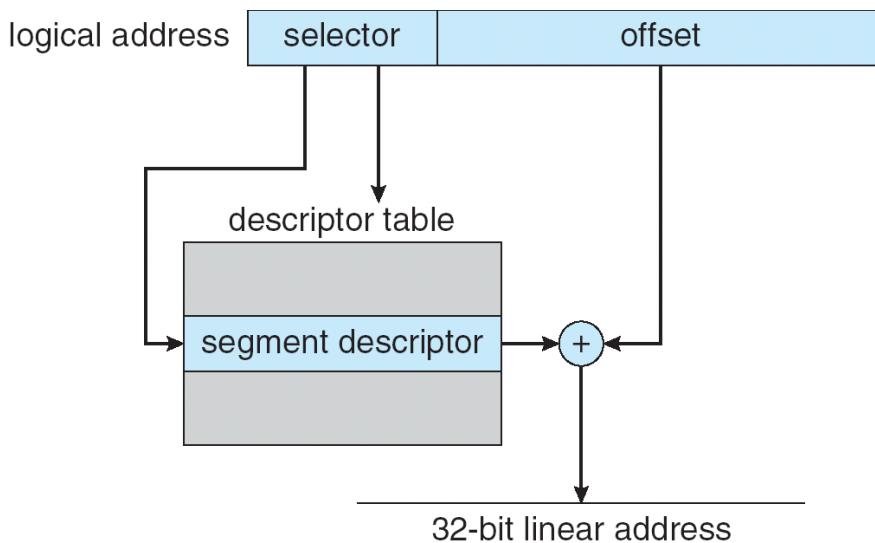
## Example: The Intel Pentium

- Supports both segmentation and segmentation with paging
- CPU generates logical address
- Given to segmentation unit
  - Which produces linear addresses
- Linear address given to paging unit
  - Which generates physical address in main memory
  - Paging units form equivalent of MMU

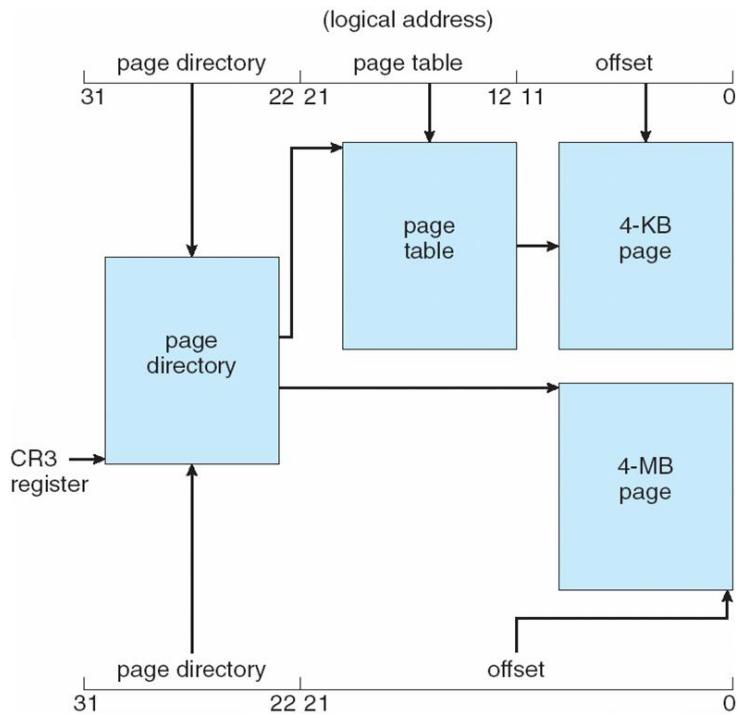
## Logical to Physical Address Translation in Pentium



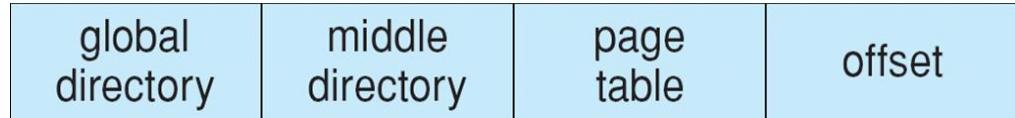
## Intel Pentium Segmentation



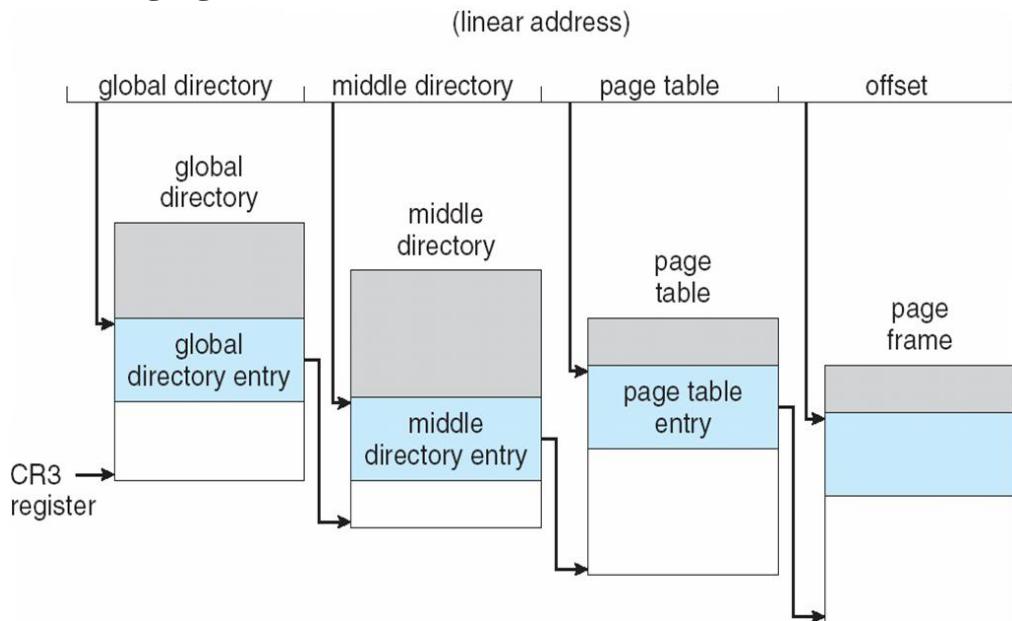
## Pentium Paging Architecture

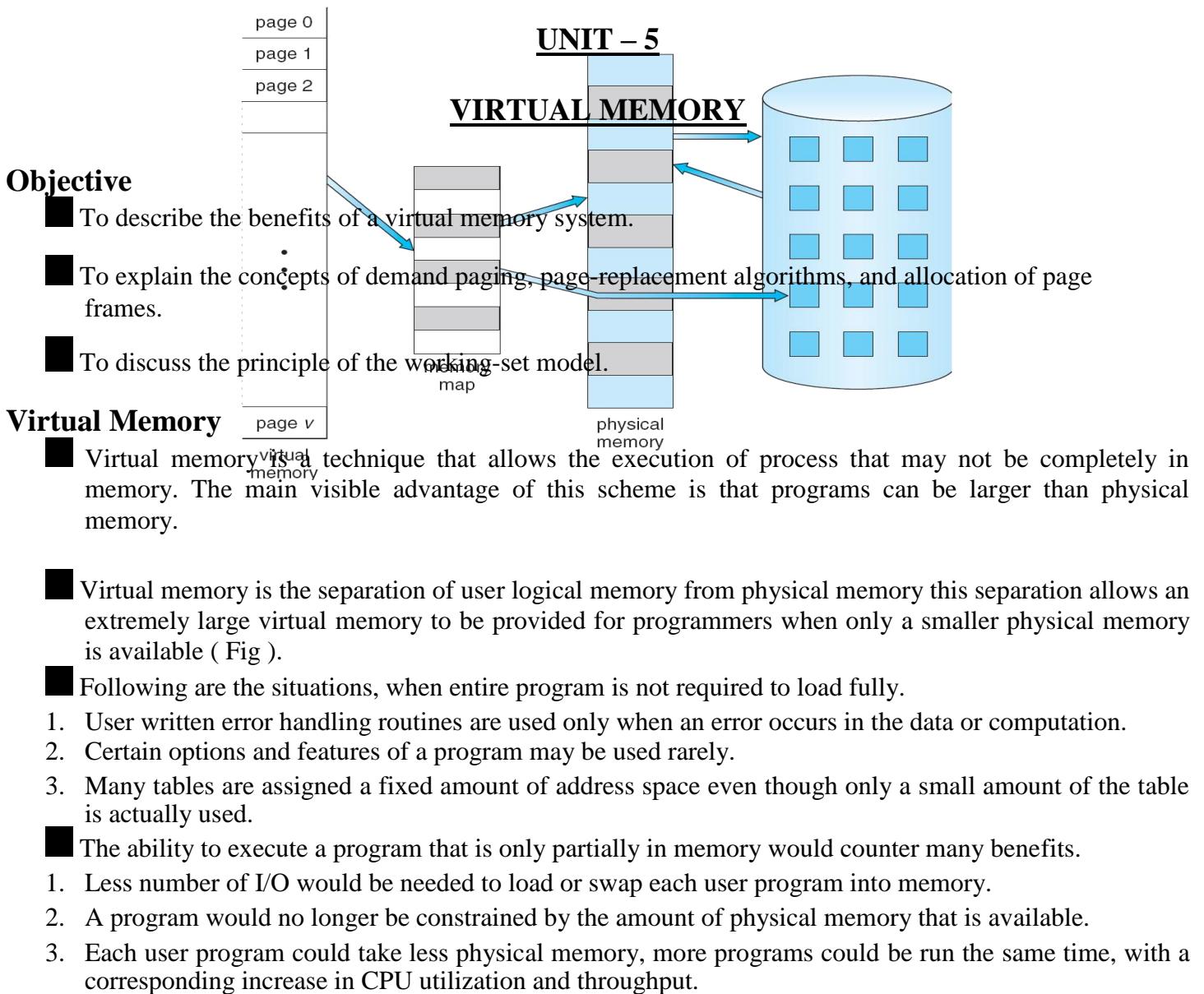


## Linear Address in Linux



## Three-level Paging in Linux





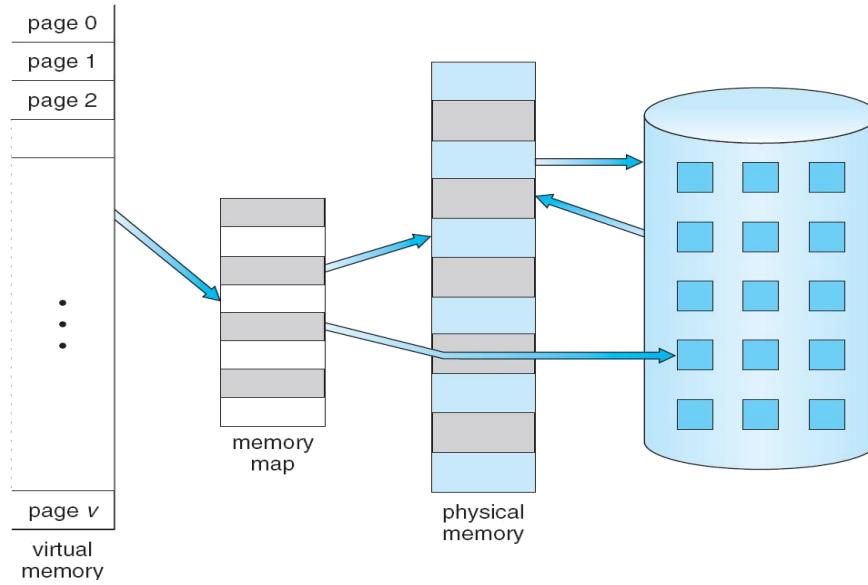


Fig. Diagram showing virtual memory that is larger than physical memory.

Virtual memory is commonly implemented by demand paging. It can also be implemented in a segmentation system. Demand segmentation can also be used to provide virtual memory.

## Demand Paging

A demand paging is similar to a paging system with swapping(Fig 5.2). When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory.

When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again. Instead of swapping in a whole process, the pager brings only those necessary pages into memory. Thus, it avoids reading into memory pages that will not be used in anyway, decreasing the swap time and the amount of physical memory needed.

Hardware support is required to distinguish between those pages that are in memory and those pages that are on the disk using the valid-invalid bit scheme. Where valid and invalid pages can be checked checking the bit and marking a page will have no effect if the process never attempts to access the pages. While the process executes and accesses pages that are memory resident, execution proceeds normally.

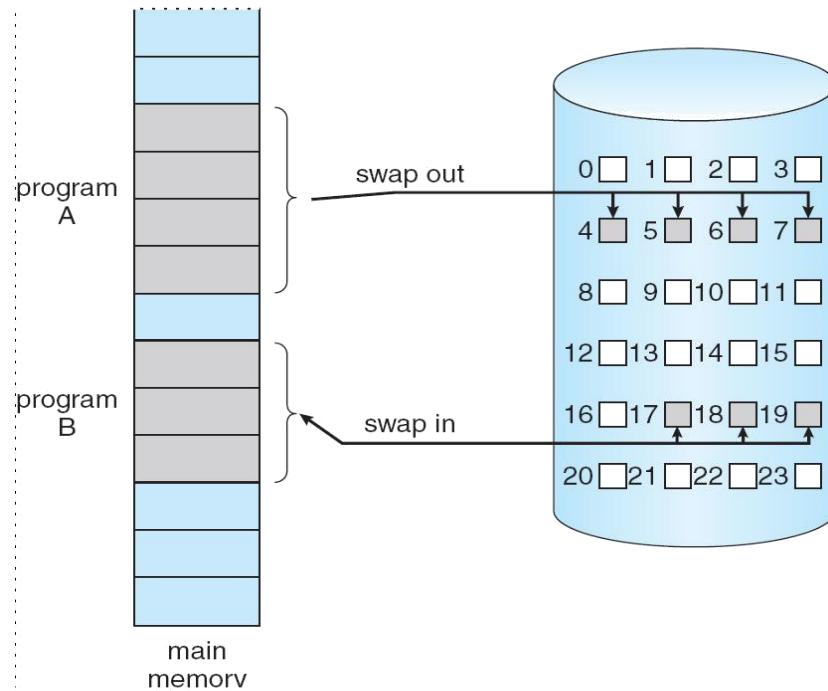


Fig. Transfer of a paged memory to continuous disk space

Access to a page marked invalid causes a page-fault trap. This trap is the result of the operating system's failure to bring the desired page into memory. But page fault can be handled as following (Fig 5.3):

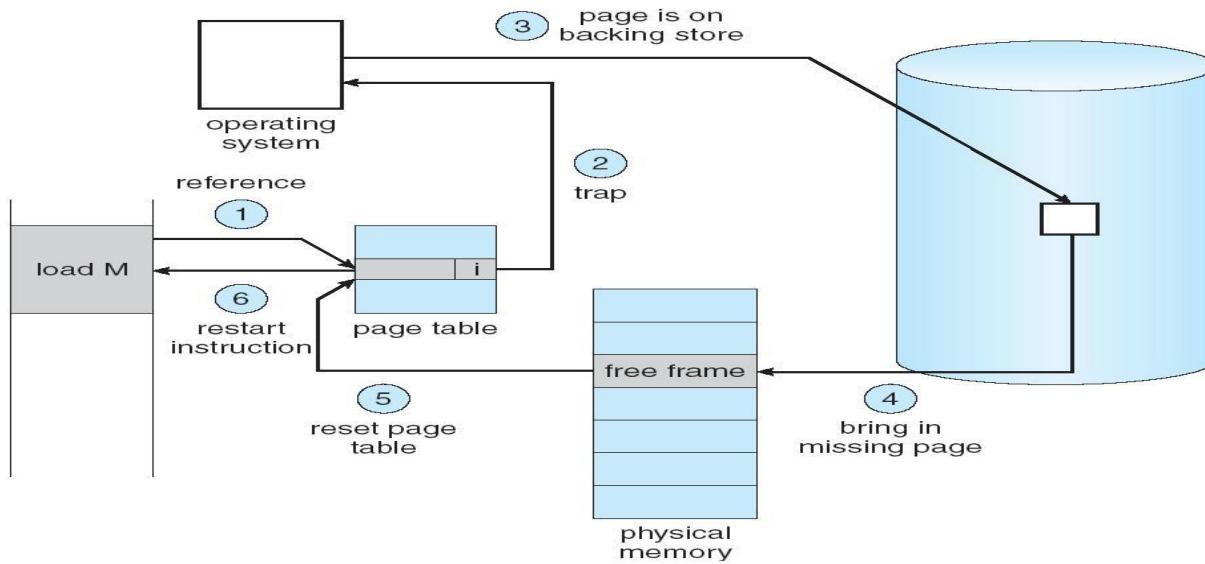


Fig. Steps in handling a page fault

1. We check an internal table for this process to determine whether the reference was a valid or invalid memory access.
2. If the reference was invalid, we terminate the process. If it was valid, but we have not yet brought in that page, we now page in the latter.
3. We find a free frame.
4. We schedule a disk operation to read the desired page into the newly allocated frame.
5. When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.
6. We restart the instruction that was interrupted by the illegal address trap. The process can now access the page as though it had always been memory.

Therefore, the operating system reads the desired page into memory and restarts the process as though the page had always been in memory.

The page replacement is used to make the frame free if they are not in used. If no frame is free then other process is called in.

### **Advantages of Demand Paging:**

1. Large virtual memory.
2. More efficient use of memory.
3. Unconstrained multiprogramming. There is no limit on degree of multiprogramming.

### **Disadvantages of Demand Paging:**

4. Number of tables and amount of processor overhead for handling page interrupts are greater than in the case of the simple paged management techniques.
5. due to the lack of an explicit constraints on a jobs address space size.

### **Page Replacement Algorithm**

There are many different page replacement algorithms. We evaluate an algorithm by running it on a particular string of memory reference and computing the number of page faults. The string of memory references is called reference string. Reference strings are generated artificially or by tracing a given system and recording the address of each memory reference. The latter choice produces a large number of data.

1. For a given page size we need to consider only the page number, not the entire address.
2. if we have a reference to a page p, then any immediately following references to page p will never cause a page fault. Page p will be in memory after the first reference; the immediately following references will not fault.

Eg:- consider the address sequence

0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103, 0104, 0101, 0610, 0102,

0103, 0104, 0104, 0101, 0609, 0102, 0105  
and reduce to 1, 4, 1, 6, 1, 6, 1, 6, 1

To determine the number of page faults for a particular reference string and page replacement algorithm, we also need to know the number of page frames available. As the number of frames available increase, the number of page faults will decrease.

## FIFO Algorithm

The simplest page-replacement algorithm is a FIFO algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. We can create a FIFO queue to hold all pages in memory.

The first three references (7, 0, 1) cause page faults, and are brought into these empty eg. 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1 and consider 3 frames. This replacement means that the next reference to 0 will fault. Page 1 is then replaced by page 0.

## Optimal Algorithm

An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms. An optimal page-replacement algorithm exists, and has been called OPT or MIN. It is simply

Replace the page that will not be used for the longest period of time.

Now consider the same string with 3 empty frames.

The reference to page 2 replaces page 7, because 7 will not be used until reference 15, whereas page 0 will be used at 5, and page 1 at 14. The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again. Optimal replacement is much better than a FIFO.

The optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string.

## LRU Algorithm

The FIFO algorithm uses the time when a page was brought into memory; the OPT algorithm uses the time when a page is to be used. In LRU replace the page that has not been used for the longest period of time.

LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses that page that has not been used for the longest period of time.

Let  $S^R$  be the reverse of a reference string  $S$ , then the page-fault rate for the OPT algorithm on  $S$  is the same as the page-fault rate for the OPT algorithm on  $S^R$ .

## LRU Approximation Algorithms

Some systems provide no hardware support, and other page-replacement algorithm. Many systems provide some help, however, in the form of a reference bit. The reference bit for a page is set, by the hardware, whenever that page is referenced. Reference bits are associated with each entry in the page table. Initially, all bits are cleared (to 0) by the operating system. As a user process executes, the bit associated with each page referenced is set (to 1) by the hardware.

### Additional-Reference-Bits Algorithm

The operating system shifts the reference bit for each page into the high-order or of its 5-bit byte, shifting the other bits right 1 bit, discarding the low-order bit.

These 5-bit shift registers contain the history of page use for the last eight time periods. If the shift register contains 00000000, then the page has not been

used for eight time periods; a page that is used at least once each period would have a shift register value of 11111111.

### Second-Chance Algorithm

The basic algorithm of second-chance replacement is a FIFO replacement algorithm. When a page gets a second chance, its reference bit is cleared and its arrival e is reset to the current time.

### Enhanced Second-Chance Algorithm

The second-chance algorithm described above can be enhanced by considering both the reference bit and the modify bit as an ordered pair.

1. (0,0) neither recently used nor modified best page to replace.
2. (0,1) not recently used but modified not quite as good, because the page will need to be written out before replacement.
3. (1,0) recently used but clean probably will be used again soon.
4. (1,1) recently used and modified probably will be used again, and write out will be needed before replacing it

### Counting Algorithms

There are many other algorithms that can be used for page replacement.

- **LFU Algorithm:** The least frequently used (LFU) page-replacement algorithm requires that the page with the smallest count be replaced. This algorithm suffers from the situation in which a page is used heavily during the initial phase of a process, but then is never used again.

- **MFU Algorithm:** The most frequently used (MFU) page-replacement algorithm is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

### Page Buffering Algorithm

When a page fault occurs, a victim frame is chosen as before. However, the desired page is read into a free frame from the pool before the victim is written out.

This procedure allows the process to restart as soon as possible, without waiting for the victim page to be written out. When the victim is later written out, its frame is added to the free-frame pool.

When the FIFO replacement algorithm mistakenly replaces a page mistakenly replaces a page that is still in active use, that page is quickly retrieved from the free-frame buffer, and no I/O is necessary. The free-frame buffer provides protection against the relatively poor, but simple, FIFO replacement algorithm.

## UNIT VI

### Principles of deadlock

To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks. To present a number of different methods for preventing or avoiding deadlocks in a computer system

#### **The Deadlock Problem**

A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set

Example

System has 2 disk drives

$P_1$  and  $P_2$  each hold one disk drive and each needs another one

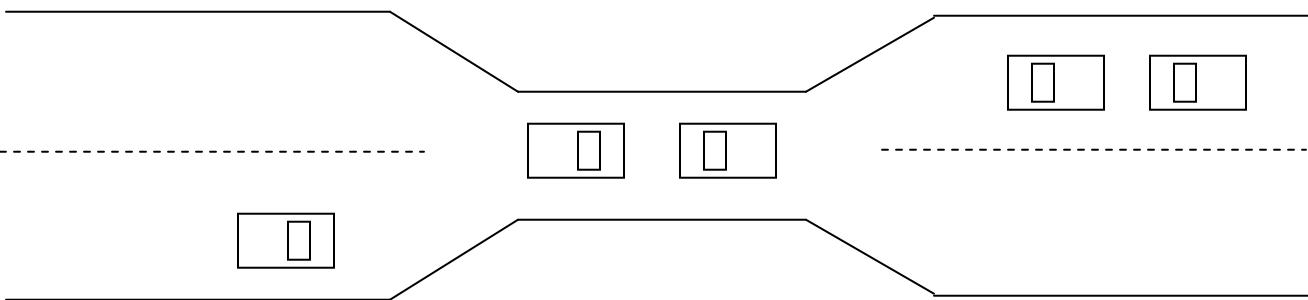
Example

semaphores  $A$  and  $B$ , initialized to 1

$P_0 \quad P_1$

```
wait (A);      wait(B)  
wait (B);      wait(A)
```

#### **Bridge Crossing Example**



Traffic only in one direction

Each section of a bridge can be viewed as a resource

If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)

Several cars may have to be backed up if a deadlock occurs

Starvation is possible

Note – Most OSes do not prevent or deal with deadlocks

## System Model

Resource types  $R_1, R_2, \dots, R_m$

*CPU cycles, memory space, I/O devices*

Each resource type  $R_i$  has  $W_i$  instances.

Each process utilizes a resource as follows:

**request**

**use**

**release**

## Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously

**Mutual exclusion:** only one process at a time can use a resource

**Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes

**No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task

**Circular wait:** there exists a set  $\{P_0, P_1, \dots, P_n\}$  of waiting processes such that  $P_0$  is waiting for a resource that is held by  $P_1$ ,  $P_1$  is waiting for a resource that is held by

$P_2, \dots, P_{n-1}$  is waiting for a resource that is held by

$P_n$ , and  $P_0$  is waiting for a resource that is held by  $P_0$ .

n

## 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  $P_i \circledast R_j$

assignment edge – directed edge  $R_j \circledast P_i$

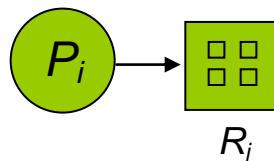
Process



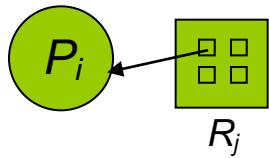
Resource Type with 4 instances



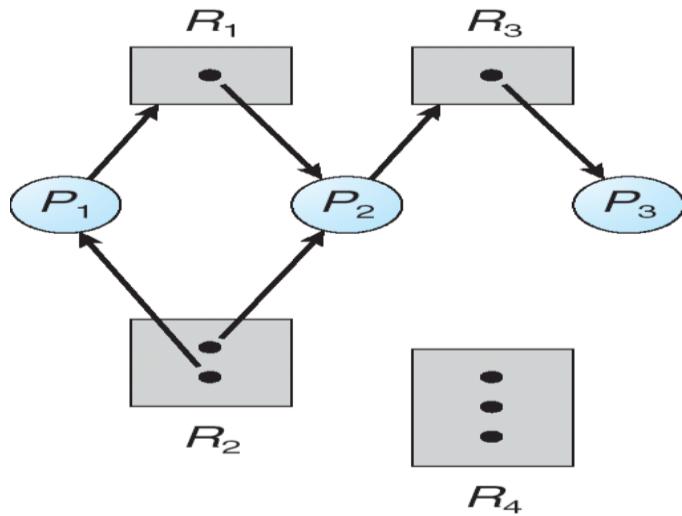
$P_i$  requests instance of  $R_j$



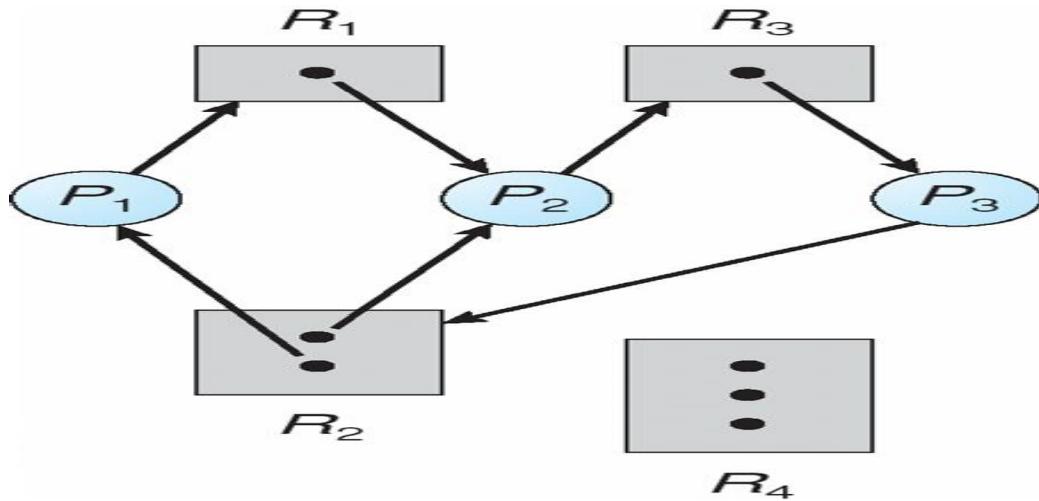
$P_i$  is holding an instance of  $R_j$



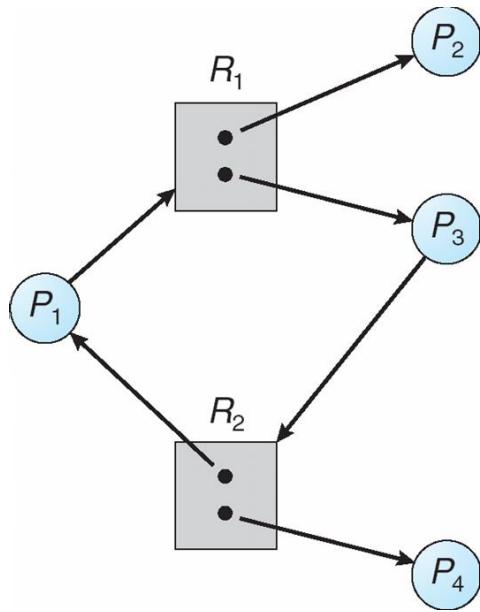
### Example of a Resource Allocation Graph



### Resource Allocation Graph With A Deadlock



## Graph With A Cycle But No Deadlock



### Basic Facts

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

### Methods for Handling Deadlocks

Ensure that the system will *never* enter a deadlock state  
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

### Deadlock Prevention

Restrain the ways request can be made

**Mutual Exclusion** – not required for sharable resources; must hold for nonsharable 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

Low resource utilization; starvation possible

### 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

**Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

## Deadlock Avoidance

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

Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need

The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition

Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes

## Safe State

When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state

System is in safe state if there exists a sequence  $\langle P_1, P_2, \dots, P_n \rangle$  of ALL the processes 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  $P_i$  resource needs are not immediately available, then  $P_i$  can wait until all  $P_j$  have finished

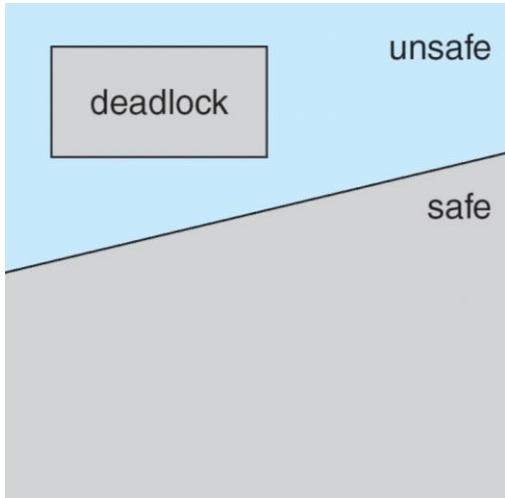
When  $P_j$  is finished,  $P_i$  can obtain needed resources, execute, return allocated resources, and terminate

When  $P_i$  terminates,  $P_{i+1}$  can obtain its needed resources, and so on

## Basic Facts

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.

## Safe, Unsafe , Deadlock State



## Avoidance algorithms

Single instance of a resource type

Use a resource-allocation graph

Multiple instances of a resource type

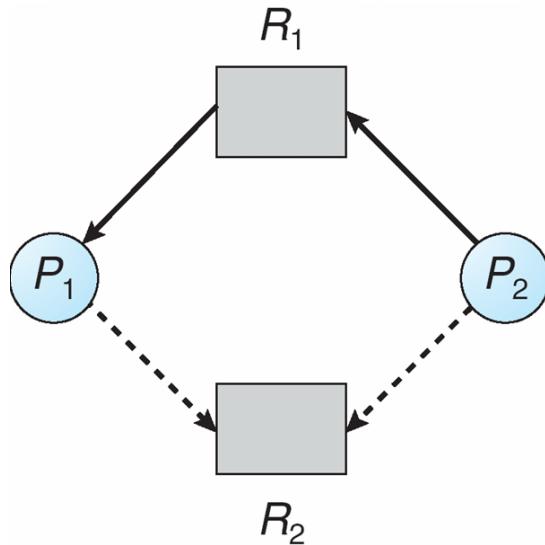
Use the banker's algorithm

### Resource-Allocation Graph Scheme

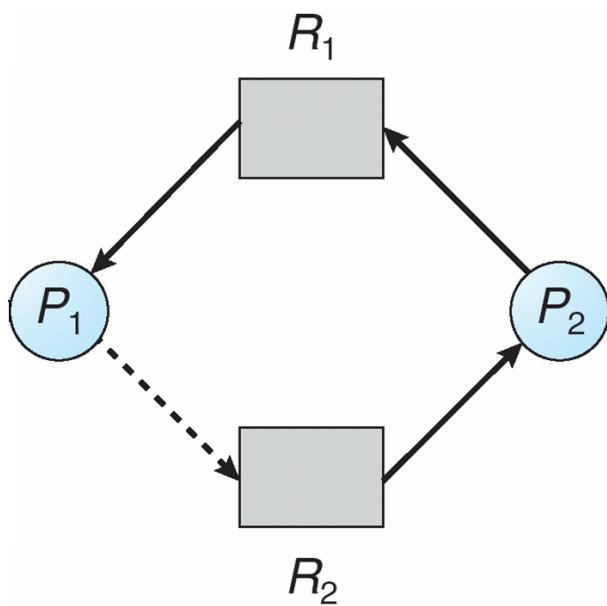
nClaim edge  $P_i \xrightarrow{\text{dashed}} R_j$  indicated that process  $P_j$  may request resource  $R_j$ ; represented by a dashed line  
nConverts to request edge when a process requests a resource  
nRequest edge converted to an assignment edge when the resource is allocated to the process

nWhen a resource is released by a process, assignment edge reconverts to a claim edge  
nResources must be claimed *a priori* in the system

### Resource-Allocation Graph



### Unsafe State In Resource-Allocation Graph



## Resource-Allocation Graph Algorithm

Suppose that process  $P_i$  requests a resource  $R_j$ .

The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph

## Banker's Algorithm

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

## Data Structures for the Banker's Algorithm

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

$\text{Available}$ : Vector of length  $m$ . If available  $[j] = k$ , there are  $k$  instances of resource type  $R_j$  available

$\text{Max}$ :  $n \times m$  matrix. If  $\text{Max}[i,j] = k$ , then process  $P_i$  may request at most  $k$  instances of resource type  $R_j$

$\text{Allocation}$ :  $n \times m$  matrix. If  $\text{Allocation}[i,j] = k$  then  $P_i$  is currently allocated  $k$  instances of  $R_j$

$\text{Need}$ :  $n \times m$  matrix. If  $\text{Need}[i,j] = k$ , then  $P_i$  may need  $k$  more instances of  $R_j$  to complete its task

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

## Safety Algorithm

1. Let  $\text{Work}$  and  $\text{Finish}$  be vectors of length  $m$  and  $n$ , respectively. Initialize:

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

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

2. Find and  $i$  such that both:

(a)  $\text{Finish}[i] = \text{false}$  (b)  $\text{Need}_i \leq \text{Work}$

If no such  $i$  exists, go to step 4

3.  $\text{Work} = \text{Work} + \text{Allocation}_i$

$\text{Finish}[i] = \text{true}$

go to step 2

4. If  $\text{Finish}[i] == \text{true}$  for all  $i$ , then the system is in a safe state

## Resource-Request Algorithm for Process $P_i$

$\text{Request}$  = 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  $\text{Request}_i \not\leq \text{Need}_i$  go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim

2. If  $\text{Request}_i \not\leq \text{Available}$ , 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:

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

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

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

If safe  $P$  the resources are allocated to  $P_i$

If unsafe  $P$   $P_i$  must wait, and the old resource-allocation state is restored

## Example of Banker's Algorithm

5 processes  $P_0$  through  $P_4$ ;

3 resource types:

A (10 instances), B (5 instances), and C (7 instances)

Snapshot at time  $T_0$ :

| <u><math>A\ B\ C</math></u> | <u><math>A\ B\ C</math></u> | <u><math>Allocation</math></u> | <u><math>Max\ Available</math></u> |
|-----------------------------|-----------------------------|--------------------------------|------------------------------------|
| $P_0$                       |                             | 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                              |

The content of the matrix  $Need$  is defined to be  $Max - Allocation$

| <u><math>Need</math></u> | <u><math>A\ B\ C</math></u> |
|--------------------------|-----------------------------|
| $P_0$ 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

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

Check that Request £ Available (that is,  $(1,0,2) \leq (3,3,2)$ )  $\Rightarrow$  true

| <u><math>Available</math></u> | <u><math>A\ B\ C</math></u> | <u><math>Allocation</math></u> | <u><math>Need</math></u>    |
|-------------------------------|-----------------------------|--------------------------------|-----------------------------|
|                               | <u><math>A\ B\ C</math></u> | <u><math>A\ B\ C</math></u>    | <u><math>A\ B\ C</math></u> |
| $P_0$                         | 0 1 0                       | 7 4 3                          | 2 3 0                       |
| $P_1$                         | 3 0 2                       | 0 2 0                          |                             |
| $P_2$                         | 3 0 1                       | 6 0 0                          |                             |
| $P_3$                         | 2 1 1                       | 0 1 1                          |                             |
| $P_4$                         | 0 0 2                       | 4 3 1                          |                             |

Executing safety algorithm shows that sequence  $\langle P_1, P_3, P_4, P_0, P_2 \rangle$  satisfies safety requirement

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

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

### Deadlock Detection

Allow system to enter deadlock state Detection algorithm Recovery scheme

## Single Instance of Each Resource Type

Maintain *wait-for* graph

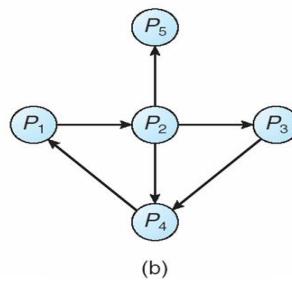
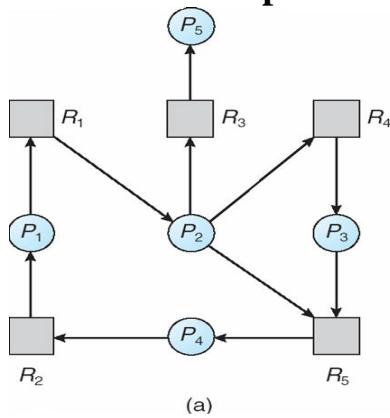
Nodes are processes

$P_i \circledast P_j$  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 an order of  $n^2$  operations, where  $n$  is the number of vertices in the graph

## Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph

Corresponding wait-for graph

## Several Instances of a Resource Type

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

## Detection Algorithm

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

(a)  $Work = Available$  (b) For  $i = 1, 2, \dots, n$ , if  $Allocation_i \neq 0$ , then

$Finish[i] = \text{false}$ ; otherwise,  $Finish[i] = \text{true}$  2. Find an index  $i$  such that both:

(a)  $Finish[i] == \text{false}$  (b)  $Request_i \neq Work$  If no such  $i$  exists, go to step 4

3.  $Work = Work + Allocation_i$

$Finish[i] = \text{true}$

go to step 4. If  $Finish[i] == \text{false}$ , for some  $i$ ,  $1 \leq i \leq n$ , then the system is in deadlock state. Moreover, if  $Finish[i] == \text{false}$ , then  $P_i$  is deadlocked

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

### Example of Detection Algorithm

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

|       | <u>Allocation</u> |   |   | <u>Request</u> |   |   | <u>Available</u> |   |   |
|-------|-------------------|---|---|----------------|---|---|------------------|---|---|
|       | A                 | B | C | A              | B | C | A                | B | C |
| $P_0$ | 0                 | 1 | 0 |                |   |   | 0                | 0 | 0 |
| $P_1$ | 2                 | 0 | 0 |                |   |   | 2                | 0 | 2 |
| $P_2$ | 3                 | 0 | 3 |                |   |   | 0                | 0 | 0 |
| $P_3$ | 2                 | 1 | 1 |                |   |   | 1                | 0 | 0 |
| $P_4$ | 0                 | 0 | 2 |                |   |   | 0                | 0 | 2 |

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

$P_2$  requests an additional instance of type C

|       | <u>Request</u> |   |   |
|-------|----------------|---|---|
|       | A              | B | C |
| $P_0$ | 0              | 0 | 0 |
| $P_1$ | 2              | 0 | 1 |
| $P_2$ | 0              | 0 | 1 |
| $P_3$ | 1              | 0 | 0 |
| $P_4$ | 0              | 0 | 2 |

State of system?

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

### Detection-Algorithm Usage

When, and how often, to invoke depends on:

How often a deadlock is likely to occur?

How many processes will need to be rolled back?

one for each disjoint cycle  
 If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock

### 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

Selecting a victim – minimize cost

Rollback – return to some safe state, restart process for that state

Starvation – same process may always be picked as victim, include number of rollback in cost factor

## UNIT VII

### FILE SYSTEM INTERFACE

#### **7.1 The Concept Of a File**

A file is a named collection of related information that is recorded on secondary storage. The information in a file is defined by its creator. Many different types of information may be stored in a file.

##### **File attributes:-**

A file is named and for the user's convenience is referred to by its name. A name is usually a string of characters. One user might create a file, whereas another user might edit that file by specifying its name. There are different types of attributes.

1)**name**:- the name can be in the human readable form.

2)**type**:- this information is needed for those systems that support different types.

3)**location**:- this information is used to identify the device and to the location of the file on that device.

4)**size**:- this indicates the size of the file in bytes or words.

5)**protection**:-

##### **6)time,date, and user identifications:-**

The information about all files is kept in the directory structure, that also resides on secondary storage.

##### **File operations:-**

###### **Creating a file:-**

Two steps are necessary to create a file first, space in the file system must be found for the file. Second , an entry for the new file must be made in the directory. The directory entry records the name of the file and the location in the system.

###### **Writing a file:-**

To write a file give the name of the file, the system search the directory to find the location of the file. The system must keep the *writer* pointer to the location in the file where the next write is to take place. The write pointer must be updated whenever a write occurs.

**Reading a file**:- to read from a file, specifies the name of the file and directory is searched for the associated directory entry, and the system needs to keep *read* pointer to the location in the file where the next read is to take place. Once the read has taken place, read pointer is updated.

###### **Repositioning with in a file:-**

The directory is searched for the appropriate entry and the current file position is set to given value. this is also known as a file seek.

**Deleting a file**:- to delete a file , we search the directory for the name file. Found that file in the directory entry, we release all file space and erase the directory entry.

**Truncate a file**:- this function allows all attributes to remain unchanged(except for file length) but for the file to be reset to length zero.

**Appending**:- add new information to the end of an existing file .

**Renaming**:- give new name to an existing file.

**Open a file**:-if file need to be used, the first step is to open the file, using the *open* system call.

**Close**:- close is a system call used to terminate the use of an already used file.

### **File Types:-**

A common technique for implementing file type is to include the type as part of the file name. The name is split into two parts

1) the name 2) and an extension .

the system uses the extension to indicate the type of the file and the type of operations that can be done on that file.

### **7.2 : ACCESSMETHODS:-**

There are several ways that the information in the file can be accessed.

**1)sequential method 2) direct access method 3) other access methods.**

1)sequential access method:-

the simplest access method is S.A. information in the file is processed in order, one after the other. the bulk of the operations on a file are reads & writes. It is based on a tape model of a file. Fig 10.3

2)Direct access:- or relative access:-

a file is made up of fixed length records, that allow programs to read and write record rapidly in no particular order. For direct access, file is viewed as a numbered sequence of blocks or records. A direct access file allows, blocks to be read & write. So we may read block15, block 54 or write block10. there is no restrictions on the order of reading or writing for a direct access file. It is great useful for immediate access to large amount of information.

The file operations must be modified to include the block number as a parameter.

We have read n, where n is the block number.

3)other access methods:-

the other access methods are based on the index for the file. The indexed contain pointers to the various blocks. To find an entry in the file , we first search the index and then use the pointer to access the file directly and to find the desired entry. With large files. The index file itself, may become too large to be kept in memory. One solution is to create an index for the index file. The primary index file would contain pointers to secondary index files which would point to the actual data items

### **7.3 Directory Structures:-**

operations that are be on a directory (read in text book)

**single level directory:-**

the simple directory structure is the single level directory. All files are contained in the same directory. Which is easy to understand. Since all files are in same directory, they must have unique names.

In a single level directory there is some limitations. When the no.of files increases or when there is more than one user some problems can occurs. If the no.of files increases, it becomes difficult to remember the names of all the files. FIG 10.7

**Two-level directory:-**

The major disadvantages to a single level directory is the confusion of file names between different users. The standard solution is to create separate directory for each user.

In 2-level directory structure, each user has her own user file directory(ufd). Each ufd has a similar structure, the user first search the master file directory . the mfd is indexed by user name and each entry point to the ufd for that user.fig 10.8

To create a file for a user, the O.S search only that user's ufd to find whether another file of that name exists. To delete a file the O.S only search to the local ufd and it can not accidentally delete another user's file that has the same name.

This solves the name collision problem, but it still have another. This is disadvantages when the user wants to cooperate on some task and to access one another's file . some systems simply do not allow local user files to be accessed by other user.

Any file is accessed by using path name. Here the user name and a file name defines a path name.

Ex:- user1/ob

In MS-DOS a file specification is

C:/directory name/file name

### **Tree structured directory:-**

This allows users to create their own subdirectories and to organize their files accordingly. here the tree has a root directory. And every file in the system has a unique path name. A path name is the path from the root, through all the subdirectories to a specified file.FIG 10.9.

A directory contains a set of subdirectories or files. A directory is simply another file, but it is treated in a special way. Here the path names can be of two types.

1)absolute path and 2) relative path.

An absolute path name begins at the root and follows a path down to the specified file, giving the directory name on the path.

Ex:- root/spell/mail/prt/first.

A relative pathname defines a path from the current directory ex:- prt/first is relative path name.

### **A cyclic- graph directory:-**

Consider two programmers who are working on a joint project. The files associated with that project can be stored in a sub directory , separating them from other projects and files of the two programmers. The common subdirectory is shared by both programmers. A shared directory or file will exist in the file system in two places at once. Notice that a shared file is not the same as two copies of the file with two copies, each programmer can view the copy rather than the original but if one programmer changes the file the changes will not appear in the others copy with a shared file there is only one actual file, so any changes made by one person would be immediately visible to the other.

A tree structure prohibits the sharing of files or directories. An acyclic graph allows directories to have shared subdirectories and files

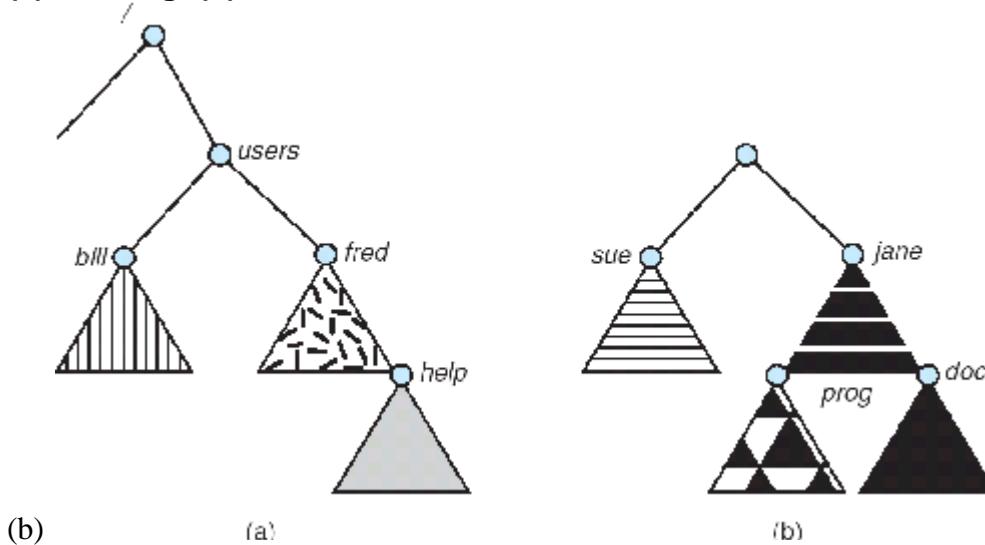
FIG 10.10 . it is more complex and more flexible. Also several problems may occurs at the traverse and deleting the file contents.

#### 7.4:File System Mounting

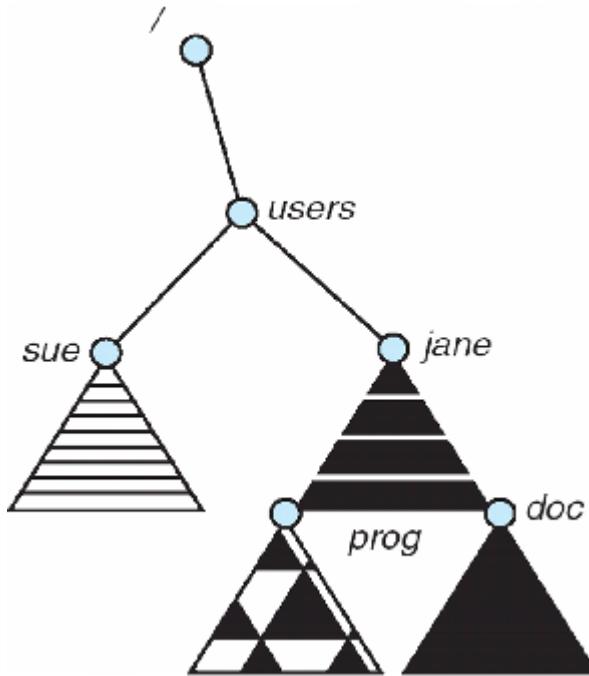
A file system must be **mounted** before it can be accessed

A unmounted file system (i.e. Fig. 11-11(b)) is mounted at a **mount point**

(a) Existing. (b) Unmounted Partition



### Mount Point



## 7.5: File Sharing

Sharing of files on multi-user systems is desirable. Sharing may be done through a **protection scheme**. On distributed systems, files may be shared across a network. Network File System (NFS) is a common distributed filesharing method.

### File Sharing – Multiple Users

User IDs identify users, allowing permissions and protections to be per user. Group IDs allow users to be in groups, permitting group access rights.

### File Sharing – Remote File Systems

- | Uses networking to allow file system access between systems
- | Manually via programs like FTP
- | Automatically, seamlessly using **distributed file systems**
- | Semi automatically via the **world wide web**
- | Client-server model allows clients to mount remote file systems from servers
- | Server can serve multiple clients
- | Client and user-on-client identification is insecure or complicated
- | NFS is standard UNIX client-server file sharing protocol
- | CIFS is standard Windows protocol
- | Standard operating system file calls are translated into remote calls
- | Distributed Information Systems (**distributed naming services**) such as LDAP, DNS, NIS, Active Directory implement unified access to information needed for remote computing

### File Sharing – Failure Modes

Remote file systems add new failure modes, due to network failure, server failure.

Recovery from failure can involve state information about status of each remote request.

Stateless protocols such as NFS include all information in each request, allowing easy recovery but less security

### File Sharing – Consistency Semantics

Consistency semantics specify how multiple users are to access a shared file simultaneously

Similar to Ch 7 process synchronization algorithms

- ▶ Tend to be less complex due to disk I/O and network latency (for remote file systems)

Andrew File System (AFS) implemented complex remote file sharing semantics

Unix file system (UFS) implements:

- ▶ Writes to an open file visible immediately to other users of the same open file
- ▶ Sharing file pointer to allow multiple users to read and write concurrently

AFS has session semantics

- ▶ Writes only visible to sessions starting after the file is closed

## 7.6:Protection

File owner/creator should be able to control:

what can be done

by whom

Types of access

Read

Write

Execute

Append

Delete

List

### Protection:-]

When the information is kept in the system the major worry is its protection from the both physical damage (Reliability) and improper access(Protection).

The reliability is generally provided by duplicate copies of files.

The protection can be provided in many ways . for some single system user, we might provide protection by physically removing the floppy disks . in a multi-user systems, other mechanism are needed.

#### 1) types of access:-

if the system do not permit access to the files of other users, protection is not needed. Protection mechanism provided by controlling accessing. This can be provided by types of file access. Access is permitted or denied depending on several factors. Suppose we mentioned read that file allows only for read .

Read:- read from the file.

Write:- write or rewrite the file.

Execute:- load the file in to memory and execute it.

Append:- write new information at the end of the file.

Delete:- delete the file and free its space for possible reuse.

## FILE SYSTEM IMPLEMENTATION

### 7.7:File allocation methods:-

There are 3 major methods of allocating disk space.

#### 1) Contiguous allocation:-

1) The contiguous allocation method requires each file to occupy a set of contiguous block on the disk.

2) Contiguous allocation of a file is defined by the disk address and length of the first block. If the file is 'n' block long and starts at location 'b', then it occupies blocks  $b, b+1, b+2, \dots, b+n-1$ ;

3) The directory entry for each file indicates the address of the starting block and length of the area allocated for this file. Fig 11.3

4) Contiguous allocation of file is very easy to access. For the *sequential access*, the file system remembers the disk address of the last block referenced and, when necessary read next block. For *direct access* to block 'i' of a file that starts at block 'b', we can immediately access block  $b+i$ . Thus both sequential and direct access can be supported by contiguous allocation.

5) One difficulty with this method is finding space for a new file.

6) Also there are many problems with this method

a) **external fragmentation**:- files are allocated and deleted, the free disk space is broken in to little pieces. The E.F exists when free space is broken in to chunks(large piece) and these chunks are not sufficient for a request of new file.

There is a solution for E.F i.e compaction. All free space compact in to one contiguous space. But the cost of compaction is time.

b) Another problem is determining how much space is needed for a file. When file is created the creator must specifies the size of that file. This becomes to big problem. Suppose if we allocate too little space to a file, some times it may not sufficient.

Suppose if we allocate large space some times space is wasted.

c) Another problem is if one large file is deleted, that large space is becomes to empty. Another file is loaded in to that space whose size is very small then some space is wasted. that wastage of space is called internal fragmentation.

#### 2) Linked allocation:-

1) Linked allocation solves all the problems of contagious allocation. With linked allocation, each file is a linked list of disk blocks, the disk block may be scattered any where on the disk.

2) The directory contains a pointer to the first and last blocks of the file. **Fig11.4**

Ex:- a file have five blocks start at block 9, continue at block 16,then block 1, block 10 and finally block 25. each block contains a ponter to the next block.

These pointers are not available to the user.

3) To create a new file we simply creates a new entry in directory. With linked allocation, each directory entry has a pointer to the first disk block of the file.

3) There is no external fragmentation with linked allocation. Also there is no need to declare the size of a file when that file is created. A file can continue to grows as long as there are free blocks.

- 4) But it have disadvantage. The major problem is that it can be used only for sequential access-files.
- 5) To find the  $I^{th}$  block of a file , we must start at the beginning of that file, and follow the pointers until we get to the  $I^{th}$  block. It can not support the direct access.
- 6) Another disadvantage is it requires space for the pointers. If a pointer requires 4 bytes out of 512 byte block, then 0.78% of disk is being used for pointers, rather than for information.
- 7) The solution to this problem is to allocate blocks in to multiples, called clusters and to allocate the clusters rather than blocks.
- 8) Another problem is reliability. The files are linked together by pointers scattered all over the disk what happen if a pointer were lost or damaged.

**FAT( file allocation table):-**

An important variation on the linked allocation method is the use of a file allocation table.

The table has one entry for each disk block, and is indexed by block number. The FAT is used much as is a linked list.

The directory entry contains the block number of the first block of the file. The table entry contains the block number then contains the block number of the next block in the file. This chain continuous until the last block, which has a special end of file values as the table entry. Unused blocks are indicated by a '0' table value. Allocation a new block to a file is a simple. First finding the first 0-value table entry, and replacing the previously end of file value with the address of the new block. The 0 is then replaced with end of file value.

**Fig 11.5**

**3)Indexed allocation:-**

- 1) linked allocation solves the external fragmentation and size declaration problems of contagious allocation. How ever in the absence of a FAT , linked allocation can not support efficient direct access.
  - 2) The pointers to the blocks are scattered with the blocks themselves all over the disk and need to be retrieved in order.
  - 3) Indexed allocation solves this problem by bringing all the pointers together in to one location i.e *the index block*.
  - 4) Each file has its own index block ,which is an array of disk block addresses. The  $I^{th}$  entry in the index block points to the  $i^{th}$  block of the file.
  - 5) The directory contains the address of the index block. **Fig 11.6**
- To read the  $i^{th}$  block we use the pointer in the  $i^{th}$  index block entry to find and read the desired block.
- 6) When the file is created, all pointers in the index block are set to nil. When the  $i^{th}$  block is first written, a block is obtained from the free space manager, and its address is put in the  $i^{th}$  index block entry.
  - 7) It supports the direct access with out suffering from external fragmentation, but it suffer from the wasted space. The pointer overhead of the index block is generally greater than the pointer over head of linked allocation.

### **7.8:Free space management:-**

- 1) to keep track of free disk space, the system maintains a free space list. The free space list records all disk blocks that are free.
- 2) To create a file we search the free space list for the required amount of space, and allocate that space to the new file. This space is then removed from the free space list.
- 3) When the file is deleted , its disk space is added to the free space list.

There are many methods to find the free space.

#### **1) bit vector:-**

The free space list is implemented as a bit map or bit vector. Each block is represented by 1 bit. If the block is free the bit is 1 if the block is allocated the bit is 0.

**Ex:-** consider a disk where blocks 2,3,4,5,8,9,10,11,12,13,17,18,25, are free and rest of blocks are allocated the free space bit map would be

**00111100111110001100000010000.....**

the main advantage of this approach is that it is relatively simple and efficient to find the first free block or 'n' consecutive free blocks on the disk

#### **2) Linked list:-**

Another approach is to link together all the free disk blocks, keeping a pointer to the first free block in a special location on the disk and caching it in memory. This first block contain a pointer to the next free disk block, and so on.

How ever this scheme is not efficient to traverse the list, we must read each block, which requires I/O time.

Disk space is also wasted to maintain the pointer to next free space.

#### **3) Grouping:-**

Another method is store the addresses of 'n' free blocks in the first free block.

The first (n-1) of these blocks are actually free. The last block contains the addresses of another 'n' free blocks and so on. **Fig 11.8**

**Advantages:-** the main advantage of this approach is that the addresses of a large no.of blocks can be found quickly.

#### **4) Counting:-**

Another approach is counting. Generally several contiguous blocks may be allocated or freed simultaneously. Particularly when space is allocated with the contiguous allocation algorithm rather than keeping a list of 'n' free disk address. We can keep the address of first free block and the number 'n' of free contiguous blocks that follow the first block. Each entry in the free space list then consists of a disk address and a count.

### **7.9:Directory Implementation:-**

#### **1) Linear list:-**

- 1) the simple method of implement ting a directory is to use a linear list of file names with pointers to the data blocks.
- 2) A linear list of directory entries requires a linear search to find a particular entry.
- 3) This method is simple to program but is time consuming to execute.
- 4) To create a new file, we must first search the directory to be sure that no existing file has the same name. Then, we add a new entry at the end of the directory.

## Shri Vishnu Engineering College for Women

5) To delete a file we search the directory for the named file, then release the space allocated to it.

6) To reuse directory entry, we can do one of several things.

7) We can mark the entry as unused or we can attach it to a list of free directory entries.

Disadvantage:- the disadvantage of a linear list of directory entries is the linear search to find a file.

## UNIT VIII

### MASS-STORAGE STRUCTURE

#### **Mass-Storage Systems**

nDescribe the physical structure of secondary and tertiary storage devices and the resulting effects on the uses of the devices  
nExplain the performance characteristics of mass-storage devices  
nDiscuss operating-system services provided for mass storage, including RAID and HSM

#### **8.1: Overview of Mass Storage Structure**

Magnetic disks provide bulk of secondary storage of modern computers

Drives rotate at 60 to 200 times per second

Transfer rate is rate at which data flow between drive and computer

Positioning time (random-access time) is time to move disk arm to desired cylinder (seek time) and time for desired sector to rotate under the disk head (rotational latency)

Head crash results from disk head making contact with the disk surface

► That's bad

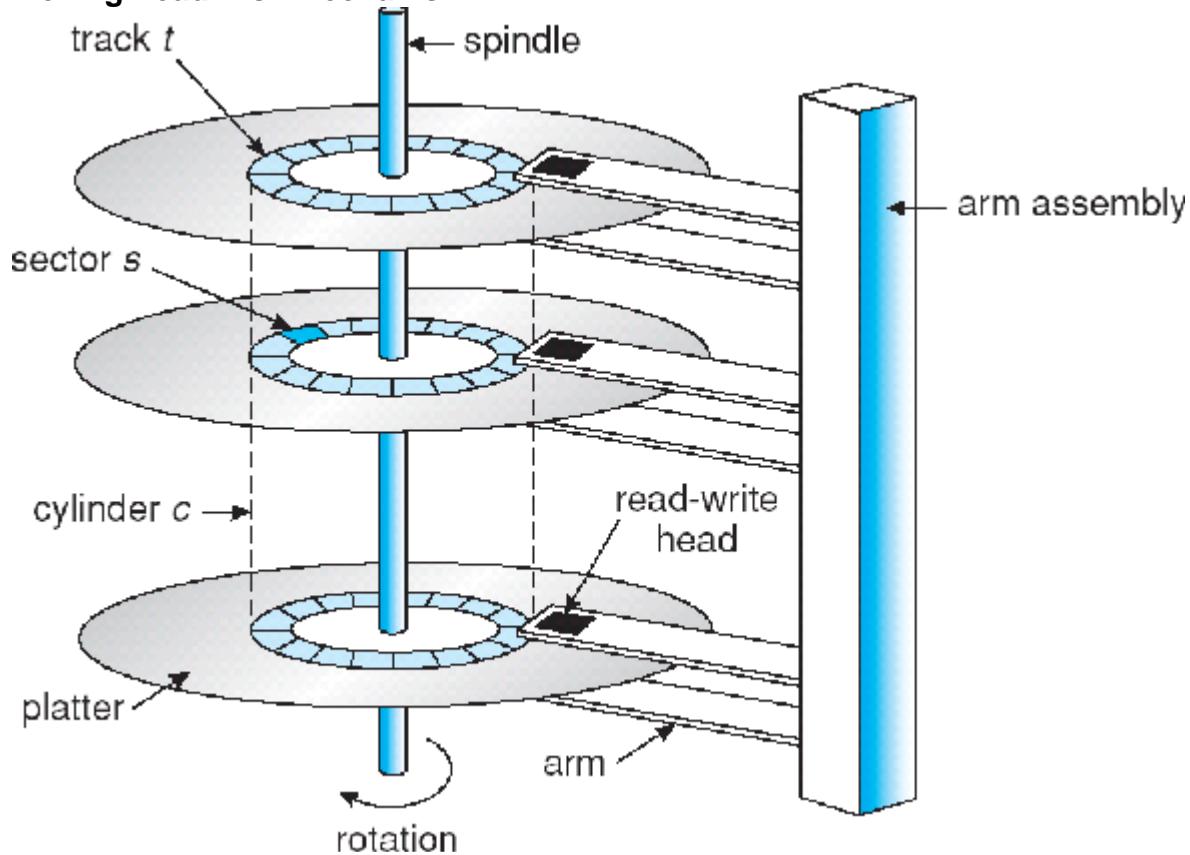
Disks can be removable

Drive attached to computer via I/O bus

Busses vary, including EIDE, ATA, SATA, USB, Fibre Channel, SCSI

Host controller in computer uses bus to talk to disk controller built into drive or storage array

#### **Moving-head Disk Mechanism**



Magnetic tape

Was early secondary-storage medium

Relatively permanent and holds large quantities of data

Access time slow

Random access ~1000 times slower than disk

Mainly used for backup, storage of infrequently-used data, transfer medium between systems

Kept in spool and wound or rewound past read-write head

Once data under head, transfer rates comparable to disk

20-200GB typical storage

Common technologies are 4mm, 8mm, 19mm, LTO-2 and SDLT

## 8.2:Disk Structure

Disk drives are addressed as large 1-dimensional arrays of logical

blocks, where the logical block is the smallest unit of transferThe 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

## 8.3:Disk Attachment

Host-attached storage accessed through I/O ports talking to I/O busses

SCSI itself is a bus, up to 16 devices on one cable, SCSI initiator requests

operation and SCSI targets perform tasks

Each target can have up to 8 logical units (disks attached to device controller

FC is high-speed serial architecture

Can be switched fabric with 24-bit address space – the basis of storage area networks (SANs) in which many hosts attach to many storage units

Can be arbitrated loop (FC-AL) of 126 devices

## Network-Attached Storage

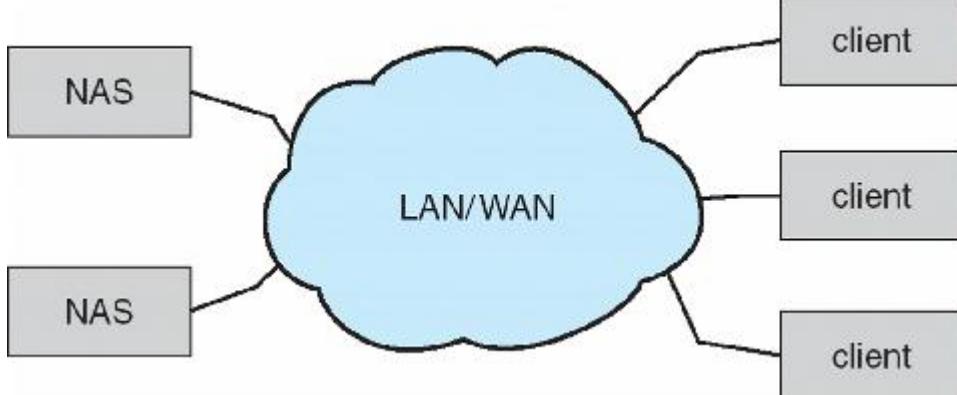
Network-attached storage (NAS) is storage made available over a network

rather than over a local connection (such as a bus)

NFS and CIFS are common protocols

Implemented via remote procedure calls (RPCs) between host and storage

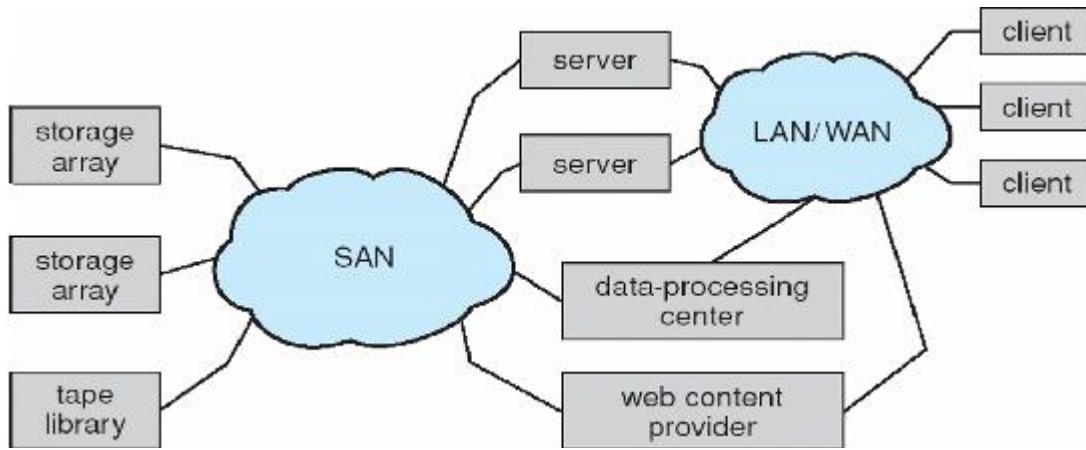
New iSCSI protocol uses IP network to carry the SCSI protocol



## Storage Area Network

Common in large storage environments (and becoming more common)

Multiple hosts attached to multiple storage arrays – flexible



## 8.4:Disk Scheduling

The operating system is responsible for using hardware efficiently — for the disk drives, this means having a fast access time and disk bandwidth

Access time has two major components

Seek time is the time for the disk arm to move the heads to the cylinder containing the desired sector

Rotational latency is the additional time waiting for the disk to rotate the desired sector to the disk head

Minimize seek time

Seek time » seek distance

Disk bandwidth is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer

Several algorithms exist to schedule the servicing of disk I/O requests. We illustrate them with a request queue (0-199)

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

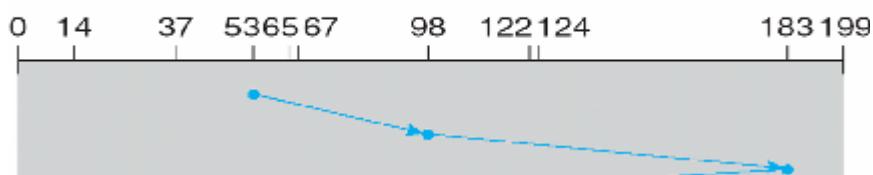
Head pointer 53

### FCFS

Illustration shows total head movement of 640 cylinders

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

head starts at 53



### SSTF

Selects the request with the minimum seek time from the current head position

SSTF scheduling is a form of SJF scheduling; may cause starvation of some requests

Illustration shows total head movement of 236 cylinders

### SCAN

The disk arm starts at one end of the disk, and moves toward the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues.

SCAN algorithm Sometimes called the elevator algorithm

Illustration shows total head movement of 208 cylinders

### **C-SCAN**

Provides a more uniform wait time than SCAN

The head moves from one end of the disk to the other, servicing requests as it goes

When it reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip

Treats the cylinders as a circular list that wraps around from the last cylinder to the first one

### **C-LOOK**

Version of C-SCAN

Arm only goes as far as the last request in each direction, then reverses direction immediately, without first going all the way to the end of the disk

## **Selecting a Disk-Scheduling Algorithm**

SSTF is common and has a natural appeal

SCAN and C-SCAN perform better for systems that place a heavy load on the disk

Performance depends on the number and types of requests

Requests for disk service can be influenced by the file-allocation method

The disk-scheduling algorithm should be written as a separate module of the operating system, allowing it to be replaced with a different algorithm if necessary

Either SSTF or LOOK is a reasonable choice for the default algorithm

### **Disk Management**

Low-level formatting, or physical formatting — Dividing a disk into sectors that the disk controller can read and write

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

Logical formatting or “making a file system”

To increase efficiency most file systems group blocks into clusters

► Disk I/O done in blocks

► File I/O done in clusters

Boot block initializes system

The bootstrap is stored in ROM

Bootstrap loader program

Methods such as sector sparing used to handle bad blocks

## **Booting from a Disk in Windows 2000**

### **8.5:Swap-Space Management**

Swap-space — Virtual memory uses disk space as an extension of main memory

Swap-space can be carved out of the normal file system, or, more commonly, it can be in a separate disk partition

Swap-space management

4.3BSD allocates swap space when process starts; holds text segment (the program) and data segment

Kernel uses swap maps to track swap-space use

Solaris 2 allocates swap space only when a page is forced out of physical

memory, not when the virtual memory page is first created

## **Data Structures for Swapping on Linux**

### **Systems**

#### **RAID Structure**

RAID – multiple disk drives provides reliability via redundancy Increases the mean time to failure Frequently combined with NVRAM to improve write performance

RAID is arranged into six different levels

Several improvements in disk-use techniques involve the use of multiple disks working cooperatively Disk striping uses a group of disks as one storage unit RAID schemes improve performance and improve the reliability of the storage system by storing redundant data

Mirroring or shadowing (RAID 1) keeps duplicate of each disk

Striped mirrors (RAID 1+0) or mirrored stripes (RAID 0+1) provides high performance and high reliability Block interleaved parity (RAID 4, 5, 6) uses much less redundancy RAID within a storage array can still fail if the array fails, so automatic replication of the data between arrays is common

Frequently, a small number of hot-spare disks are left unallocated, automatically replacing a failed disk and having data rebuilt onto them

#### **RAID (0 + 1) and (1 + 0)**

### **Extensions**

RAID alone does not prevent or detect data corruption or other errors, just disk failures

Solaris ZFS adds checksums of all data and metadata

Checksums kept with pointer to object, to detect if object is the right one and whether it changed

Can detect and correct data and metadata corruption

ZFS also removes volumes, partitions

Disks allocated in pools

Filesystems with a pool share that pool, use and release space like “malloc” and “free” memory allocate / release calls

## **ZFS Checksums All Metadata and Data**

### **Traditional and Pooled Storage**

#### **Stable-Storage Implementation**

Write-ahead log scheme requires stable storage To implement stable storage:

Replicate information on more than one nonvolatile storage media with independent failure modes

Update information in a controlled manner to ensure that we can recover the stable data after any failure during data transfer or recovery

#### **Tertiary Storage Devices**

Low cost is the defining characteristic of tertiary storage Generally, tertiary storage is built using removable media Common examples of removable media are floppy disks and CD-ROMs; other types are available

#### **Removable Disks**

Floppy disk — thin flexible disk coated with magnetic material, enclosed in a protective plastic case Most floppies hold about 1 MB; similar technology is used for removable disks that hold more than 1 GB

Removable magnetic disks can be nearly as fast as hard disks, but they are at a greater risk of damage from exposure

# Shri Vishnu Engineering College for Women

A magneto-optic disk records data on a rigid platter coated with magnetic material

Laser heat is used to amplify a large, weak magnetic field to record a bit

Laser light is also used to read data (Kerr effect)

The magneto-optic head flies much farther from the disk surface than a magnetic disk head, and the magnetic material is covered with a protective layer of plastic or glass; resistant to head crashes  
Optical disks do not use magnetism; they employ special materials that are altered by laser light

## **WORM Disks**

The data on read-write disks can be modified over and over

WORM ("Write Once, Read Many Times") disks can be written only once

Thin aluminum film sandwiched between two glass or plastic platters

To write a bit, the drive uses a laser light to burn a small hole through the aluminum; information can be destroyed by not altered

Very durable and reliable

Read-only disks, such as CD-ROM and DVD, come from the factory with the data pre-recorded

## **Tapes**

Compared to a disk, a tape is less expensive and holds more data, but random access is much slower

Tape is an economical medium for purposes that do not require fast random access, e.g., backup copies of disk data, holding huge volumes of data

Large tape installations typically use robotic tape changers that move tapes between tape drives and storage slots in a tape library

stacker – library that holds a few tapes

silo – library that holds thousands of tapes

A disk-resident file can be archived to tape for low cost storage; the computer can stage it back into disk storage for active use

## **Operating System Support**

Major OS jobs are to manage physical devices and to present a virtual machine abstraction to applications  
For hard disks, the OS provides two abstractions:

Raw device – an array of data blocks  
File system – the OS queues and schedules the interleaved requests from

several applications

## **Application Interface**

Most OSs handle removable disks almost exactly like fixed disks — a new cartridge is formatted and an empty file system is generated on the disk

Tapes are presented as a raw storage medium, i.e., an application does not open a file on the tape, it opens the whole tape drive as a raw device

Usually the tape drive is reserved for the exclusive use of that application

Since the OS does not provide file system services, the application must decide how to use the array of blocks

Since every application makes up its own rules for how to organize a tape, a tape full of data can generally only be used by the program that created it

## **Tape Drives**

The basic operations for a tape drive differ from those of a disk drive

locate() positions the tape to a specific logical block, not an entire track  
(corresponds to seek())

The read position() operation returns the logical block number where the

tape head is

The space() operation enables relative motion

Tape drives are “append-only” devices; updating a block in the middle of the tape also effectively erases everything beyond that block

An EOT mark is placed after a block that is written

### **File Naming**

The issue of naming files on removable media is especially difficult when we want to write data on a removable cartridge on one computer, and then use the cartridge in another computer

Contemporary OSs generally leave the name space problem unsolved for removable media, and depend on applications and users to figure out how to access and interpret the data

Some kinds of removable media (e.g., CDs) are so well standardized that all computers use them the same way

### **(Hierarchical Storage Management HSM)**

A hierarchical storage system extends the storage hierarchy beyond primary memory and secondary storage to incorporate tertiary storage — usually implemented as a jukebox of tapes or removable disks

Usually incorporate tertiary storage by extending the file system

Small and frequently used files remain on disk

Large, old, inactive files are archived to the jukebox

HSM is usually found in supercomputing centers and other large installations that have enormous volumes of data

### **Speed**

Two aspects of speed in tertiary storage are bandwidth and latency. Bandwidth is measured in bytes per second

Sustained bandwidth – average data rate during a large transfer; # of bytes/transfer time

Data rate when the data stream is actually flowing

Effective bandwidth – average over the entire I/O time, including seek() or locate(), and cartridge switching

Drive's overall data rate

Access latency – amount of time needed to locate data

Access time for a disk – move the arm to the selected cylinder and wait for the rotational latency; < 35 milliseconds

Access on tape requires winding the tape reels until the selected block reaches the tape head; tens or hundreds of seconds

Generally say that random access within a tape cartridge is about a thousand times slower than random access on disk

The low cost of tertiary storage is a result of having many cheap cartridges share a few expensive drives

A removable library is best devoted to the storage of infrequently used data, because the library can only satisfy a relatively small number of I/O requests per hour

### **Reliability**

A fixed disk drive is likely to be more reliable than a removable disk or tape

An optical cartridge is likely to be more reliable than a magnetic disk or tape

A head crash in a fixed hard disk generally destroys the data, whereas the failure of a tape drive or optical disk drive often leaves the data cartridge

unharmed

**Cost**

Main memory is much more expensive than disk storage. The cost per megabyte of hard disk storage is competitive with magnetic tape if only one tape is used per drive. The cheapest tape drives and the cheapest disk drives have had about the same storage capacity over the years. Tertiary storage gives a cost savings only when the number of cartridges is considerably larger than the number of drives.