Operating System Introduction

Sunbeam Infotech

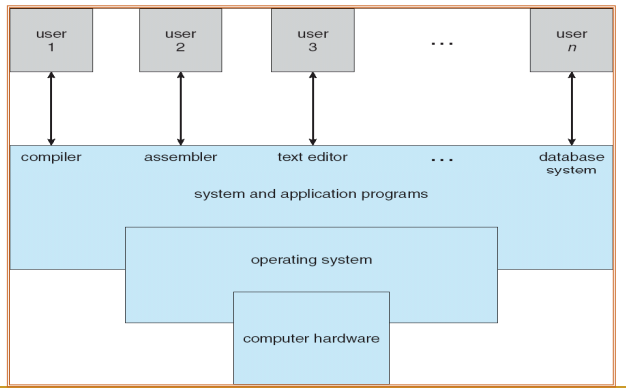
What is an Operating System?

- 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

- 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 Computer System



Operating System Definition

- OS is a **resource allocator**
  - Manages all resources and
  - 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
- Everything a vendor ships when you order an operating system
- "The one program running at all times on the computer" is the **kernel.** Everything else is either a system program (ships with the operating system) or an application program

Installing OS on PC

Sunbeam Infotech

Hard disks

- There are two types of hard disks
  - PATA / IDE
  - SATA
- SATA drives are faster as compared to PATA drives.
- These disks are connected to motherboard using different kind of connectors.
- Today's motherboards have one (or more) port/s for PATA, while more than four ports for SATA.
- These ports can be used to connect hard disks, CD/DVD R/W or printers.

Hard disk settings

- The SATA hard disk can be directly connected to motherboard using special SATA cable.
- The PATA hard disk can be connected to motherboard using other type of cable.
- PATA drive must be configured using jumper settings in one of the following modes:
  - Primary Master
  - Primary Slave
  - Secondary Master
  - Secondary Slave

BIOS settings

- Disk configurations
  - Many of the older OS do not install on SATA disks.
  - For installing such OS, in BIOS, configure disks as IDE or PATA instead of AHCI mode.
- Boot options:
  - Decides the device from which PC should try to boot.
  - The recommended order for boot devices is
    - CD/DVD drive
    - Disk drive
    - USB drive
    - Network boot

Partitions

- The hard disk can be split into multiple partitions, so that each partition could be treated as one disk.
- These partitions can be used to install operating systems and to store data/files.
- One disk can have different types of partitions:
  - Primary partition
    - Maximum four can be created on a disk.
  - Extended partition
    - Special type of primary partition
    - Only one extended partition is allowed on a disk
  - Logical partition
    - Multiple can be created within extended partition

File system

- Each partition must have some file system.
- Different operating systems can be installed on different types of file systems.
- Example file systems are:
  - FAT / FAT32 (Windows OS)
  - NTFS (Windows Server OS)
  - EXT2 / EXT3 (Linux OS)
  - HFS (MAC OS)
- FAT or NTFS partitions have drive letters that are used by Windows OS to specify file paths.

Installing Windows OS

- Insert Windows OS (Windows XP, Windows 2003 Server, Windows Vista, etc.) CD/DVD into CD/DVD drive and boot from it.
- The basic OS setup will be loaded into the memory.
- Windows can be installed on first primary partition (OR any primary or logical partition in special case).
- Create first primary partition (OR logical partition) of appropriate size for the operating system. If partitions are already created only select the partition. The partition will be formatted and OS files will be installed on it.
- Follow the steps as given in the setup program.

Installing Linux System

- Insert Linux OS (Open SuSE, RHEL, UBUNTU, etc.) CD/DVD into CD/DVD drive, boot from it and select installation option.
- The basic OS setup will be loaded into the memory.
- Select fresh/new installation option.
- Linux can be installed on any partition.
- Select the **Custom Partition setup** and Create appropriate (mostly logical) partition of appropriate size for the OS. If partitions are already created only select the partition. The partition will be formatted and OS files will be installed on it.
- Follow the steps as given by the setup program.

System Booting

Sunbeam Infotech

Bootable device

- The procedure of starting a computer system by loading the kernel is known as **booting** the system.
- The first sector of any storage device is known as **boot sector**.
- If boot sector of any storage device contains a special program called **bootstrap program**, then that device is said to be a **bootable device**.
- Similarly if boot sector of the partition of any disk contains this program, partition is said to be a **boot partition**.

bootstrap program

- Bootstrap program locates the kernel, loads it into main memory and starts its execution.
- Sometimes two-step process where **boot block** at fixed location loads complex bootstrap program.
- There is a separate bootstrap program for each operating system.
- If multiple operating systems are installed on the computer, the boot loader encompasses the bootstrap/boot loader for other operating systems.
- NTLDR, GRUB, LILO, etc are few examples of boot loaders.

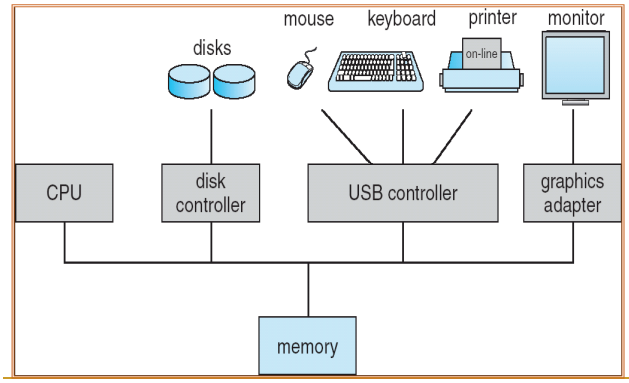
Booting process

- When computer is started, the instruction pointer (program counter) is loaded with a predefined memory location and execution starts there.
- The initial boot program is kept here in ROM, as RAM is in an unknown state at system startup.
- The bootstrap program run diagnostics to determine the state of machine (hardware).
- It also initialize CPU registers, device controllers and such things so that basic system is initialized.
- It loads kernel of OS from the disk/device to the memory and starts the first process of that kernel.
- Finally one or more system processes are created which functions as an operating system.

Computer System

Sunbeam Infotech

Computer System Structure



Computer-System Operation

- One or more CPUs, device controllers connect through common bus providing access to shared memory.
- I/O devices and the CPU can execute concurrently.
- Each device controller is in charge of a particular device type and has a local buffer.
- CPU moves data from/to main memory to/from local buffers.
- Actual input and output occurs between the device to local buffer of controller.
- Device controller informs CPU that it has finished its operation by causing an *interrupt*.

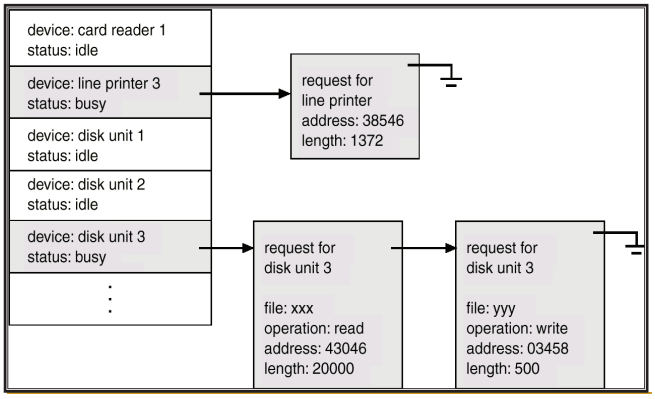
Common Functions of Interrupts

- Interrupt transfers control to the interrupt service routine generally, through the *interrupt vector*, which contains the addresses of all service routines.
- Interrupt architecture must save the address of the interrupted instruction. The OS preserves the state of the CPU by storing registers and the program counter.
- 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.

I/O Structure

- In **synchronous** I/O method, after I/O starts, control returns to user program only upon completing I/O
  - wait instruction or loop idles the CPU until the next interrupt
  - At most one I/O request is outstanding at a time, no simultaneous I/O processing.
- In **asynchronous** method, after I/O starts, control return to user program without waiting for I/O completion
  - *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.

Device-Status Table



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.

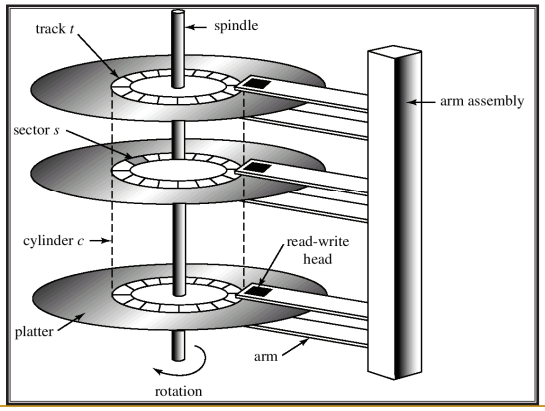
Storage Structure

- 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 systems organized in hierarchy.
  - Speed, Cost, Volatility

Main Memory

- The main memory and processor registers are directly accessed by the CPU.
- The machine instructions take the addresses from main memory or registers, not the disk.
- In concept of memory mapped I/O, a set of memory addresses are reserved to tie with device registers.
- Even I/O ports are mapped with some addresses.
- The user program or OS writes data on these addresses and then set control register to send the data.
- Access to main memory is slower than the registers.

Magnetic Disks



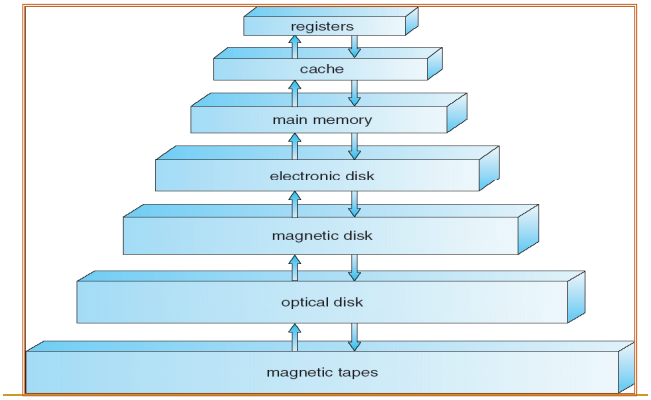
Magnetic Disks

- Magnetic disk speed is based on two factors
  - Transfer rate: data flow rate bet^{n} drive and computer
  - Positioning time or Random access time
    - Seek time: move disk arm to desired cylinder
    - Rotation latency: rotate disk head to desired sector
- Drive is attached to computer through I/O bus
- Different types of buses are
  - Enhanced integrated drive electronics (EIDE)
  - Advanced technology attachment (ATA) : (PATA/SATA)
  - Small computer systems interface (SCSI)
- Host controller is connected at computer end of bus connecting to drive and do the I/O.

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

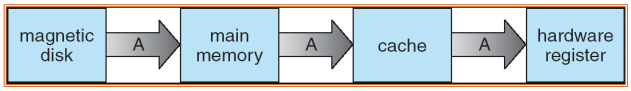
Storage-Device Hierarchy



Caching

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

Migration of int 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

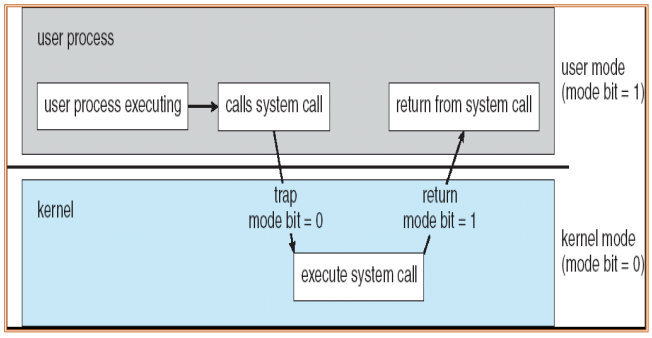
Hardware Protection

- Early operating systems work as **resident monitors**.
- Then OS start doing additional jobs like I/O, resource allocator, etc.
- In multiprogramming environment, one program could disturb other program in memory by corrupting its data.
- The programming errors are detected by hardware and conveyed to operating system via interrupt. OS should take appropriate action like terminating victim program.
- The following protection mechanisms are available:
  - Dual-Mode Operation
  - I/O Protection
  - Memory Protection
  - CPU Protection

Dual-Mode Operation

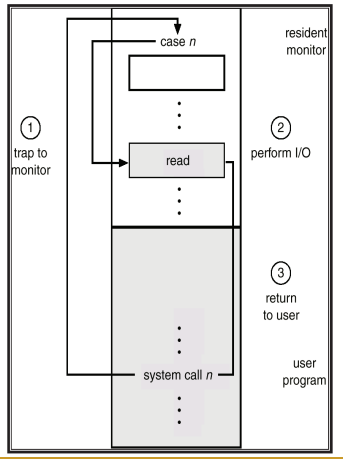
- Sharing system resources requires operating system to ensure that an incorrect program cannot cause other programs to execute incorrectly.
- Provide hardware support to differentiate between at least two modes of operations.
  - User mode – execution done on behalf of a user.
  - Monitor mode (also kernel mode or system mode) – execution done on behalf of operating system.
- *Mode bit* added to computer hardware to indicate the current mode: monitor (0) or user (1).
- When an interrupt or fault occurs hardware switches to monitor mode.

User mode and Kernel mode



I/O Protection

- All I/O instructions are privileged instructions.
- Since all I/O operations are done through IVT, it must be protected from user programs.
- I/O must be done via system calls.
- Must ensure that a user program could never gain control of the computer in monitor mode.

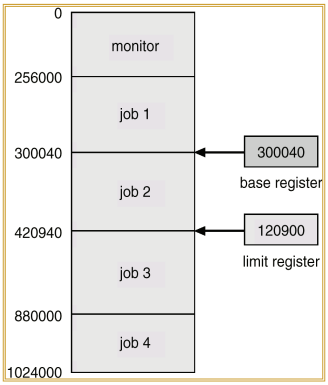


Memory Protection

- Must provide memory protection at least for the interrupt vector and the interrupt service routines.
- A user program must be protected from the other user program.
- In order to have memory protection, add two registers that determine the range of legal addresses a program may access:
  - **Base register** – holds the smallest legal physical memory address.
  - **Limit register** – contains the size of the range
- Memory outside the defined range is protected.

Use of Base and Limit Register

- When executing in monitor mode, the operating system has unrestricted access to both monitor and user's memory.
- The load instructions for the *base* and *limit* registers are privileged instructions.



CPU Protection

- User program may stuck up in infinite loop and may not return control to the operating system.
- Timer interrupts computer after specified period to ensure operating system maintains control.
  - Timer is decremented every clock tick.
  - When timer reaches the value 0, an interrupt occurs.
  - Set up before scheduling process to regain control or terminate program that exceeds allotted time
- Timer is commonly used to implement time sharing.
- Changing timer values is a privileged instruction.
- Computers may have time-of-day clock that is independent of operating system.

Operating System

Sunbeam Infotech

Multiprogramming

- 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 any job has to wait (for I/O for example), OS switches to another job
- **Multiprogramming** needed to use the system in efficient way.

Multitasking

- **Multitasking (Time sharing)** is logical extension in which CPU switches jobs so frequently that users can interact with each job while it is running
- **Response time** should be < 1 second
- If several jobs ready to run at the same time **CPU scheduling** is necessary to schedule them all.
- Multitasking can be achieved in one of the ways:
  - multi-processing
  - multi-threading
- **Virtual memory** allows execution of processes not completely in memory. If processes don't fit in memory, they are **swapped** in and out to run.

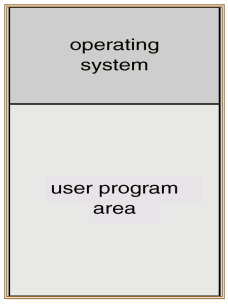
Types of operating systems

- Few operating systems are designed to be **efficient** while other are designed to be **convenient**.
- Depending on design goals and hardware constraints different types of operating systems exists.
  - Mainframe Systems
  - Desktop Systems
  - Multiprocessor Systems
  - Distributed Systems
  - Real -Time Systems
  - Handheld Systems

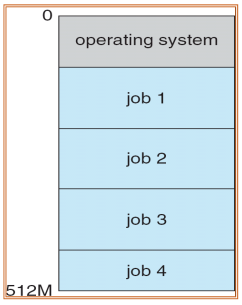
Mainframe Systems

- Batch systems
  - Reduce setup time by batching similar jobs
  - Transfers control from one job to another.
- Resident monitor
  - initial control in monitor
  - control transfers to job
  - when job completes control transfers back to monitor
- Multiprogrammed systems
  - Multiple jobs in memory at a time
- Time sharing systems
  - Executes multiple jobs using time sharing concept

Mainframe Systems



- Simple Batch System



- Multiprogramming System

Desktop Systems

- *Personal computers* – computer system dedicated to a single user.
- I/O devices – keyboards, mice, display screens, small printers.
- User convenience and responsiveness.
- May run several different types of operating systems (Windows, MacOS, UNIX, Linux)

Multiprocessor Systems

- Multiprocessor systems are also called as parallel systems.
- Multiprocessor systems with more than one CPU in close communication.
- Advantages of parallel system:
  - Increased throughput, Economical and Increased reliability
- **Symmetric multiprocessing (SMP)**
  - Each processor runs and identical copy of the operating system.
  - Most modern operating systems support SMP
- **Asymmetric multiprocessing**
  - Each processor is assigned a specific task; master processor schedules and allocated work to slave processors.
  - More common in extremely large systems

Distributed Systems

- Distribute computation among several physical processors.
- It is also called as **Loosely coupled system**. Because each processor has its own local memory; processors communicate with one another through various communications lines, such as high-speed buses or telephone lines.
- Advantages of distributed systems.
  - Resources Sharing, Load balancing, Reliability
- Requires networking infrastructure.
- Local area networks (LAN) or Wide area networks (WAN)
- May be either client-server or peer-to-peer systems.

Real -Time Systems

- Often used as a control device in a dedicated application such as controlling scientific experiments, medical imaging systems, industrial control systems, and some display systems.
- Well-defined fixed-time constraints.
- These systems may be either *hard* or *soft* real-time.
- Hard real-time
  - Secondary storage limited or absent, data stored in short term memory, or read-only memory (ROM)
- Soft real-time
  - More flexible and hence widely used.

Handheld Systems

- These systems are used for mobile hardware.
  - Personal Digital Assistants (PDAs)
  - Cellular phones
  - Portable multimedia systems
- Issues:
  - Limited memory
  - Slow processors
  - Small display screens

What Operating System does?

- There are different types of operating systems, however all these systems have some core services
  - Process Management
  - Memory Management
  - Storage Management
  - File System Management
  - I/O System Management
  - Protection and Security
  - User interfacing

Process Management

- A process is a program in execution. It is a unit of work within the system. Program is **passive**, process is **active**.
- Process needs resources to accomplish its task
  - CPU, memory, I/O, files
  - Initialization data
- Reusable resources are released as process terminates
- Process management activities done by OS:
  - Creating and deleting both user and system processes
  - Suspending and resuming processes
  - Providing mechanisms for process synchronization
  - Providing mechanisms for process communication

Memory Management

- Memory is a large array of words or bytes, each with its own address. It is a quickly accessible by the CPU and I/O devices.
- All data is in memory before and after processing.
- All instructions are in memory for execution.
- 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

- 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
- Most modern computer systems use disks as the principle on-line storage medium, for both programs and data.
- Entire speed of computer operation hinges on disk subsystem and its algorithms
- OS activities
  - Free-space management
  - Storage allocation
  - Disk scheduling
- Some storage need not be fast
  - Tertiary storage includes optical storage, magnetic tape
  - WORM (write-once, read-many-times) and RW (read-write)

File Management

- A file is a collection of related information defined by its creator. Commonly, files represent programs (both source and object forms) and data.
- Files usually organized into directories
- Access control on most systems to determine who can access what
- OS activities include
  - Creating and deleting files and directories
  - Primitives to manipulate files and directories
  - Mapping files onto secondary storage
  - Backup files onto stable (non-volatile) storage media

I/O Management

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

Operating System Interface - CLI

- User communicates with the core OS graphical or command line interface.
- CLI allows direct command entry
- Sometimes implemented in kernel, sometimes by systems program
- Sometimes multiple flavors implemented – **shells**
  - Primarily fetches a command from user and execute it
    - Sometimes commands built-in, sometimes just names of programs
    - If the latter, adding new features doesn't require shell modification

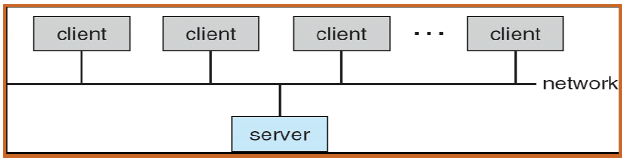
Operating System Interface - GUI

- 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 (Java desktop, KDE)

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 fire walled, 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

Operating System Design

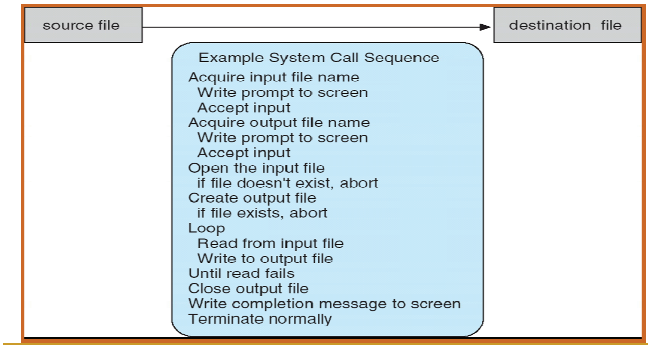
Sunbeam Infotech

System Calls

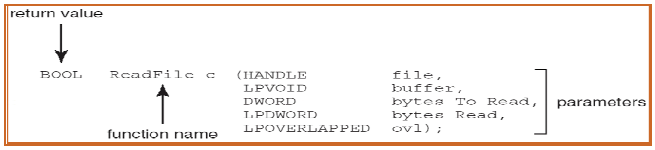
- System calls provide the interface between a running program and the operating system.
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level **Application Program Interface (API)** rather than direct system call use
  - Win32 API for Windows
  - POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X)
  - Java API for the Java virtual machine (JVM)
- Why use APIs rather than system calls?

System Call Example

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



Standard API example

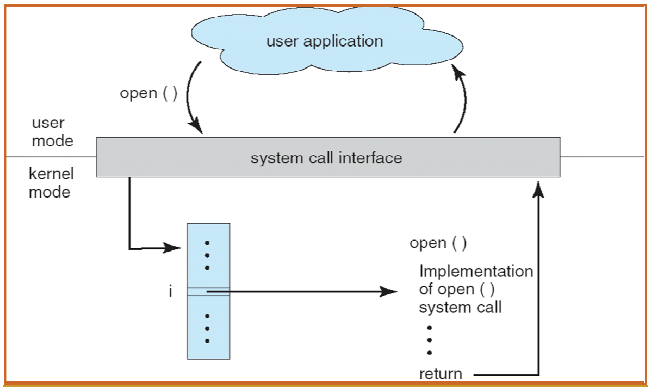


- A description of the parameters passed to ReadFile()
  - HANDLE file—the file to be read
  - LPVOID buffer—a buffer where the data will be read into and written from
  - DWORD bytesToRead—the number of bytes to be read into the buffer
  - LPDWORD bytesRead—the number of bytes read during the last read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

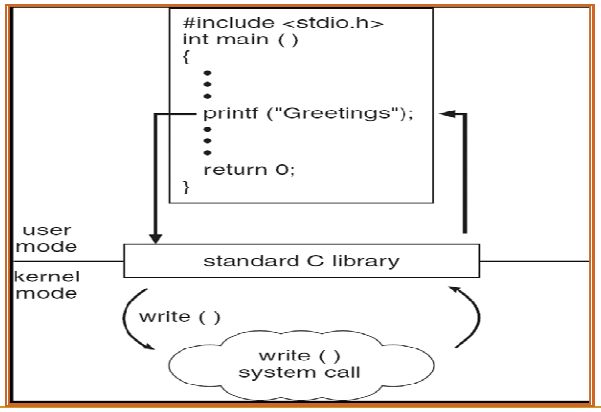
System Call Implementation

- 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 system call and any return values
- The caller need not be aware of system call implementation
- Just needs to obey API and understand what OS will do as a result call
  - Details of OS interface hidden from programmer by API
  - Managed by run-time support library

API – System Call – OS Relationship



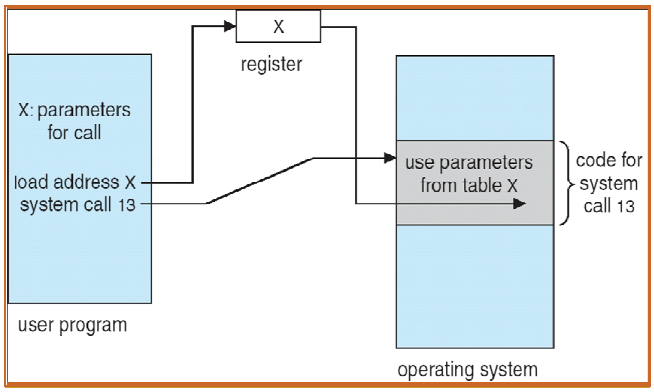
Standard C Library Example: printf()



System Call Parameters

- More information is required than identity of system call
  - Exact type and amount of information vary with OS and call
- Three 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

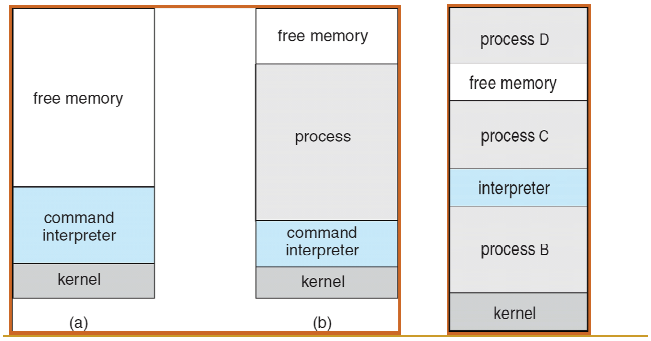


Types of System Calls

- Operating system does different activities for the services that are provided by it
- There are system calls corresponding to most of these activities
- The types of system calls will depend on the services
  - Process control
  - File management
  - Device management
  - Information maintenance
  - Communications

Running Operating Systems

- MS-DOS Execution
  - startup (a) and running program (b)
- Free-BSD UNIX
  - multiprogramming



System programs

- System programs provide a convenient environment for program development and execution. They are classified as:
  - File manipulation
  - File modification
  - Status information
  - Programming language support
  - Program loading and execution
  - Communications
  - Application programs
- Most user's view of the operation system is defined by system programs, not the actual system calls
- Some of the system programs are simply user interfaces to system calls; others are considerably more complex.

System programs (cont'd)

- File management: to manipulate files and directories
  - Text editors to create and modify files
  - Special commands to search contents of files and do transformations
- Status information
  - Some ask the system for info - date, time, amount of available memory, disk space, number of users
  - Others provide detailed performance, logging, and debugging information
  - Typically, these programs format and print the output to the terminal or other output devices
  - Some systems implement a registry - used to store and retrieve configuration information

System programs (cont'd)

- Programming-language support
  - Compilers, assemblers, debuggers and interpreters
- Program loading and execution
  - Absolute, relocatable or overlay loaders
  - debugging systems for high-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

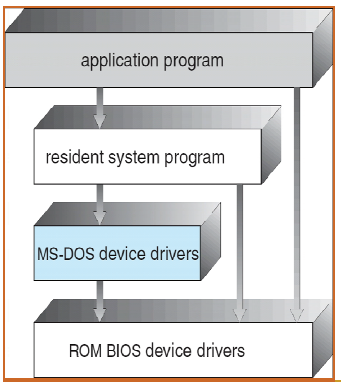
OS Design and Implementation

- Design and Implementation of OS not "solvable", but some approaches have proven successful
- Internals 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

OS Design and Implementation

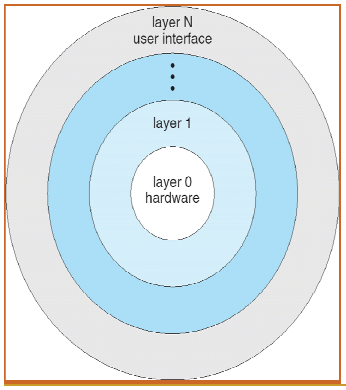
- Important principle to separate
  - **Policy**: What will be done?
  - **Mechanism**: How to do it?
- Mechanisms determine how to do something, policies decide what will be done.
- The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later
- There are different examples of operating systems depending on the goals of designs and the policies and mechanisms.

Simple Structure : MS-DOS



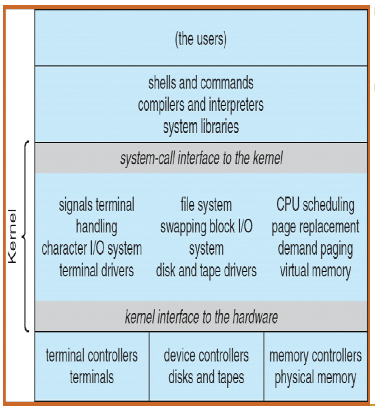
- MS-DOS – written to provide the most functionality in the least space
  - Not divided into modules
  - Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated

Layered Structure



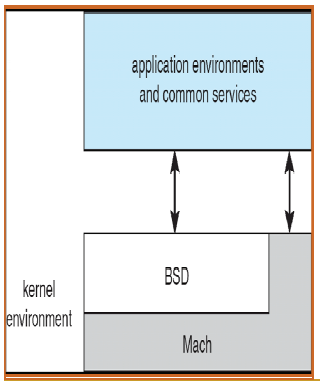
- The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level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

UNIX System



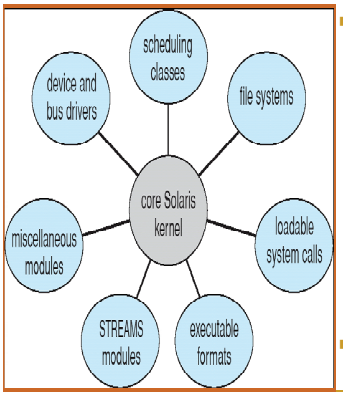
- UNIX is limited by hardware functionality, the original UNIX had limited structuring.
- UNIX consists two separate parts
  - Systems programs
  - The kernel
    - Consists of everything below the system-call interface and above the physical hardware
    - Provides the file system, CPU scheduling, memory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

Microkernel Structure: MAC OS X



- Moves as much from the kernel into "*user*" space
- Communication takes place between user modules using message passing
  - Easier to extend a microkernel
  - Easier to port on new architecture
  - More reliable (less code running in kernel mode)
  - More secure
  - Performance overhead of user space to kernel space comm^{n}

Modular Structure: Solaris

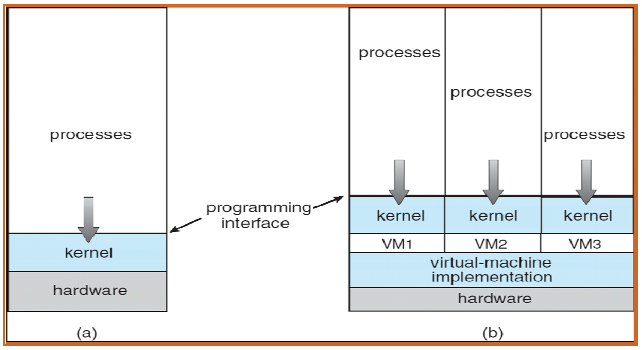


- Most modern operating systems implement kernel modules
  - Uses object-oriented approach
  - Each core component is separate
  - Each talks to the others over known interfaces
  - Each is loadable as needed within the kernel
- Overall, similar to layers but with more flexible

Virtual Machines

- **Virtual machine** takes layered approach for 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 creates the illusion of multiple processes, each executing on its own processor with its own (virtual) memory
- The resources of the physical computer are shared to create the virtual machines
  - CPU scheduling can create the appearance that users have their own processor
  - Spooling and a file system can provide virtual card readers and virtual line printers
  - A normal user time-sharing terminal serves as the virtual machine operator's console

Virtual Machine (cont'd)

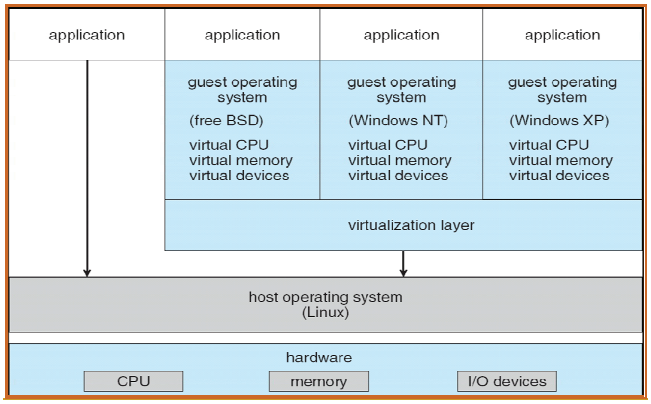


(a) Non-Virtual Machine AND (b) Virtual Machine

Virtual Machine (cont'd)

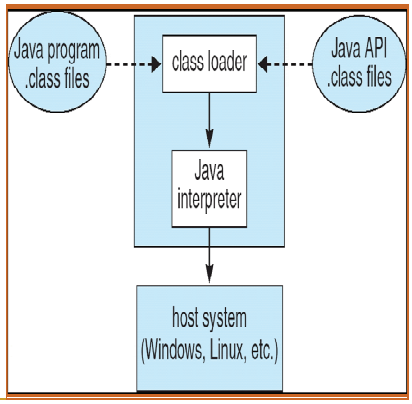
- The virtual-machine concept provides complete protection of system resources since each virtual machine is isolated from all other virtual machines. This isolation, however, permits no direct sharing of resources.
- A virtual-machine system is a perfect vehicle for operating-systems research and development. System development is done on the virtual machine, instead of on a physical machine and so does not disrupt normal system operation.
- The virtual machine concept is difficult to implement due to the effort required to provide an *exact* duplicate to the underlying machine

VMware Architecture



Java Virtual Machine

- While generating machine code, compiler assumes certain machine configuration. This machine assumed as *virtual machine*.
- JAVA compiler assumes JVM and creates byte code, that is executed on JVM installed on host system.

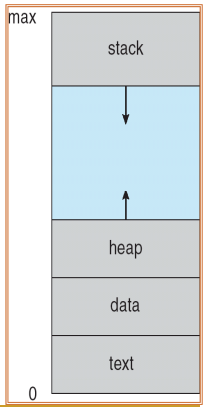


Processes

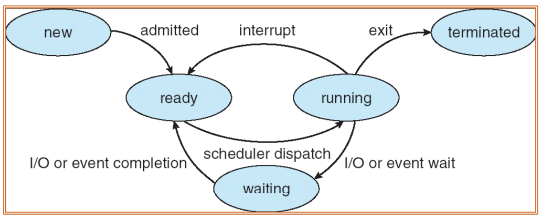
SunBeam Infotech

Process Concept

- An OS executes a variety of programs:
  - Batch system: jobs
  - Time-shared systems: user programs or tasks
- Process – a program in execution; process execution must progress in sequential fashion
- A process includes:
  - program counter
  - stack
  - data section



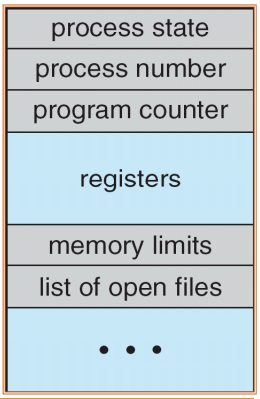
Process State



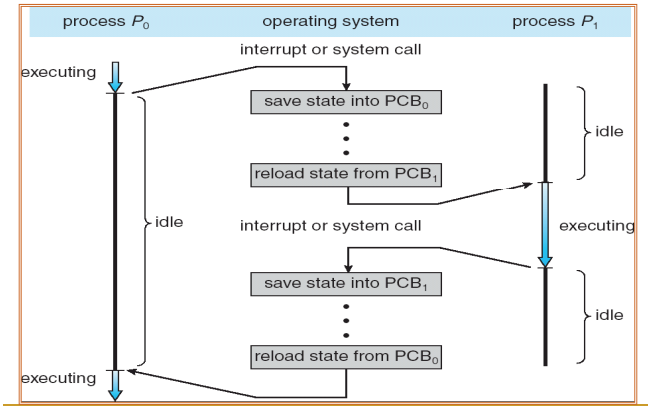
- As a process executes, it changes state
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process has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

Process Control Block (PCB)

- Information associated with each process
  - Process state
  - Program counter
  - CPU registers
  - CPU scheduling information
  - Memory-management information
  - Accounting information
  - I/O status information



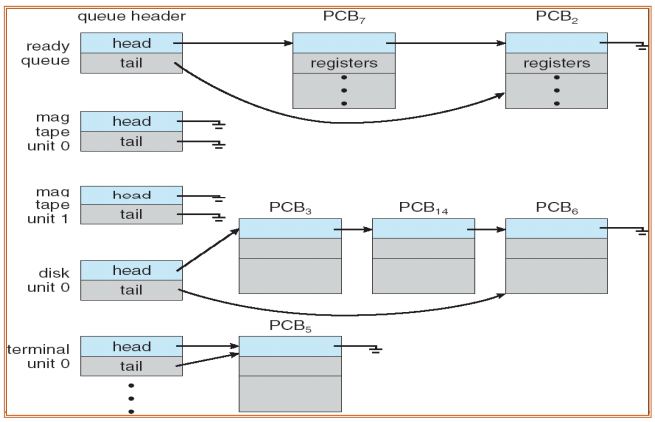
CPU Switch From Process to Process



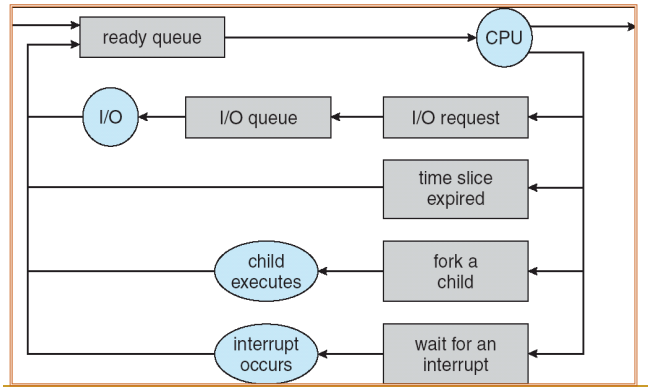
Process Scheduling Queues

- **Job queue** – set of all processes in 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 & I/O Device Queues



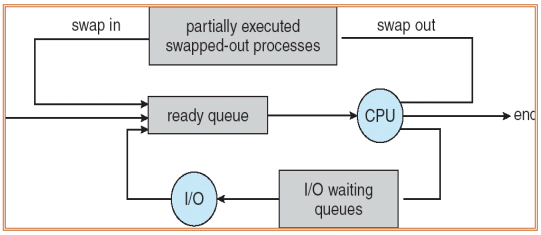
Representation of process scheduling



Schedulers

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

Schedulers & Medium term schedulers



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

Context Switch

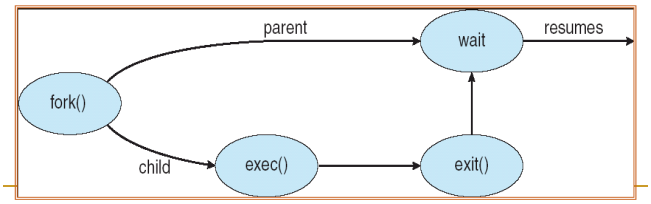
- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process
- Context-switch time is overhead; the system does no useful work while switching
- Time dependent on hardware support

Process Creation

- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- 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

Process Creation (Cont.)

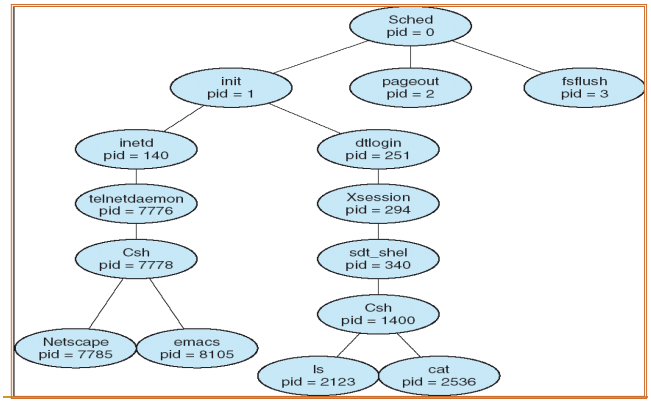
- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - **fork** system call creates new process
  - **exec** system call used after a **fork** to replace the process' memory space with a new program



C Program Fork() Separate Process

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

A processes tree on 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*

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
- Paradigm for cooperating processes, *producer* process produces info that is consumed by a *consumer* process
  - *unbounded-buffer* places no practical limit on size of the buffer
  - *bounded-buffer* assumes that there is a fixed buffer size

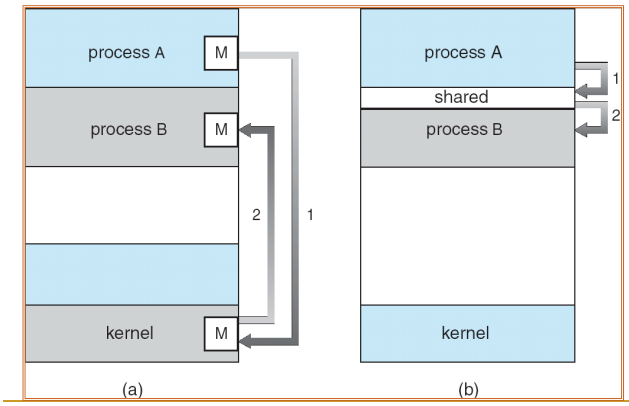
Interprocess Communication (IPC)

- 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

Implementing IPC

- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)
- Implementation questions:
  - How are links established?
  - Can a link be associated with more than two processes?
  - How many links can there be between every pair of communicating processes?
  - What is the capacity of a link?
  - Is the size of a message that the link can accommodate fixed or variable?
  - Is a link unidirectional or bi-directional?

Communications Models



Direct Communication

- Processes must name each other explicitly:
  - **send** (*P, message*) – send a message to the process P
  - **receive**(*Q, message*) – receive a message from the 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

Indirect Communication

- **Operations**
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox
- **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

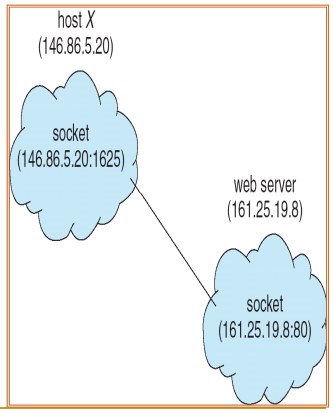
- Queue of messages attached to the link; implemented in one of three ways
  - Zero capacity – 0 messages
    - Sender must wait for receiver (rendezvous)
  - Bounded capacity – finite length of *n* messages
    - Sender must wait if link full
  - Unbounded capacity – infinite length
    - Sender never waits

Client-Server Communication

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

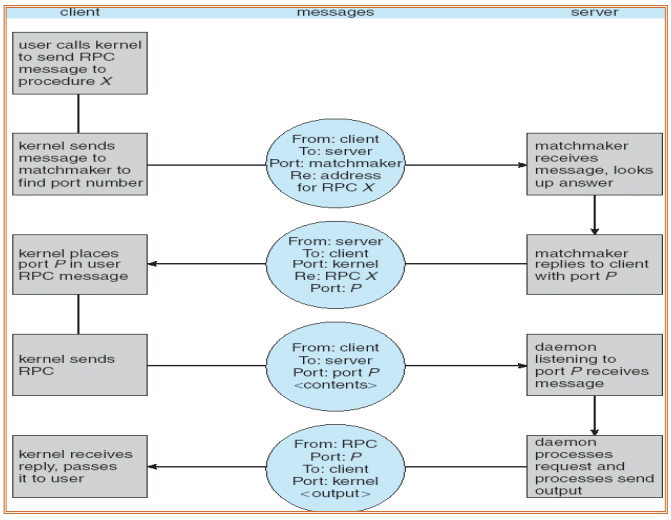
Sockets

- 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



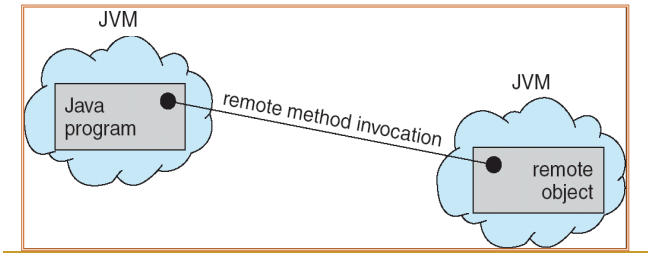
Remote Procedure Calls

- 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 peforms the procedure on the server.

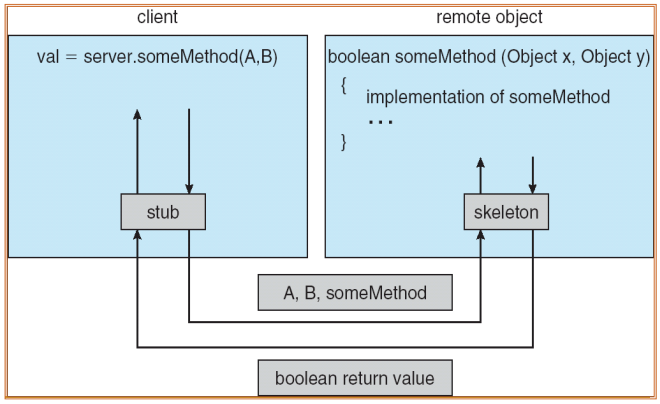


Remote Method Invocation

- 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.



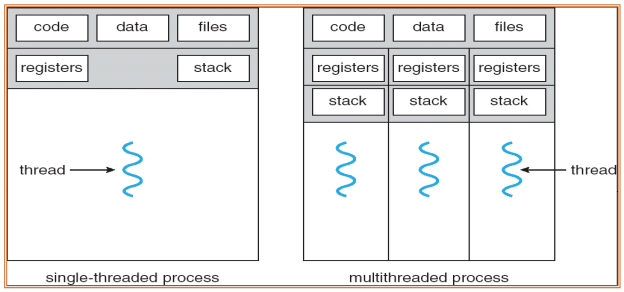
Marshalling Parameters



Threads

SunBeam Infotech

Single & Multithreaded Processes



- Benefits of Multi-threading
  - Responsiveness
  - Resource Sharing
  - Economy
  - Utilization of MP Architecture

Single & Multithreaded Processes

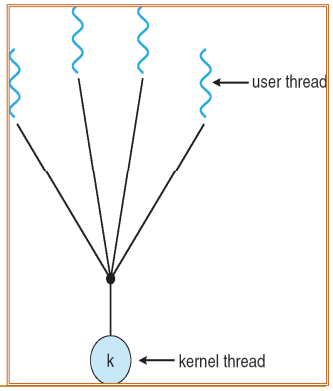
- Single-threaded process has one **program counter** specifying location of next instruction to execute
  - Process executes instructions sequentially, one at a time, until completion
- Multi-threaded process has one program counter per thread
- Typically system has many processes, some user, some operating system running concurrently on one or more CPUs
  - Concurrency by multiplexing the CPUs among the processes / threads

User threads Vs Kernel threads

- Thread management done by user-level threads library
- Three primary thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java threads
- Supported by the Kernel
- Examples
  - Windows XP/2000
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

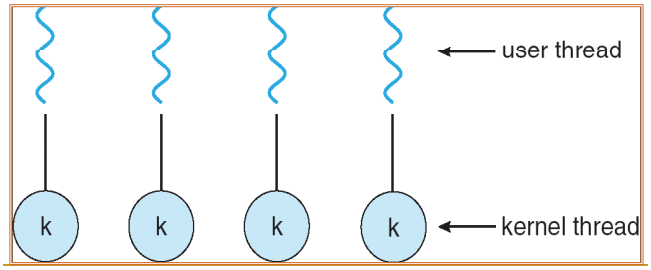
Multithreading: Many-to-One Model

- Many user-level threads mapped to single kernel thread
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads



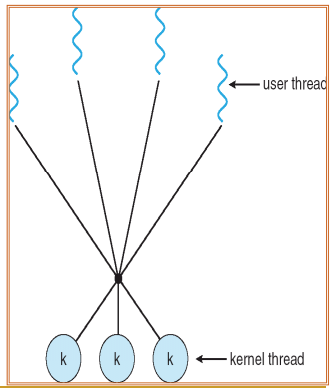
Multithreading: One-to-one Model

- Each user-level thread maps to kernel thread
- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later



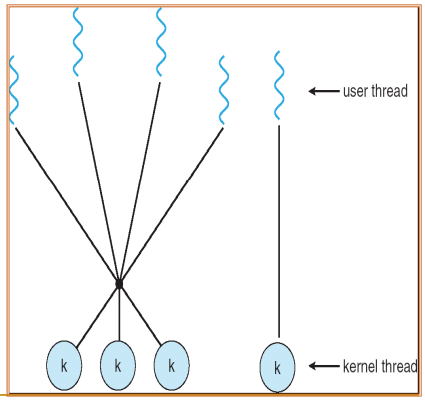
Multithreading: Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows NT/2000 with the *ThreadFiber* package



Multithreading: 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



Threading Issues

- Semantics of **fork()** and **exec()** system calls
  - Does **fork()** duplicate only the calling thread or all threads?
- 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

Threading Issues : 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
  - Signal is generated by particular event
  - Signal is delivered to a process
  - Signal is handled
- Options:
  - Deliver the signal to 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 rocess

Threading Issues

- 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)

Threading Issues

- 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

P-threads & Linux threads

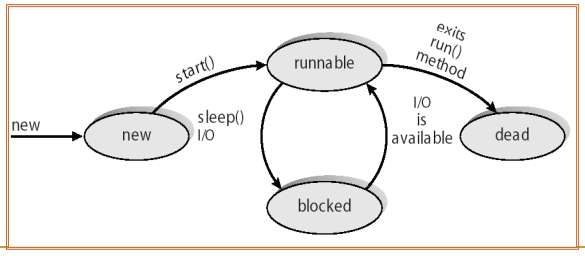
- A **POSIX** standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)
- **Linux** refers to them as *tasks* rather than *threads*
- Thread creation is done through **clone()** system call
- **clone()** allows a child task to share the address space of the parent task (process)

Windows XP Threads

- Implements the one-to-one mapping
- 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)

Java Threads

- Java threads are managed by the JVM
- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface

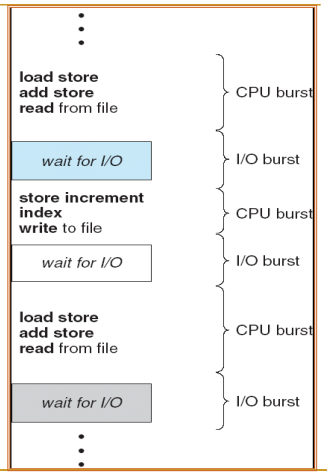


CPU Scheduling

SunBeam Infotech

Basic Concepts

- 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



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 *non-preemptive*
- All other scheduling is *preemptive*

Dispatcher

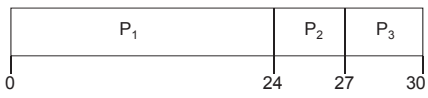
- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- *Dispatch latency* – time it takes for the dispatcher to stop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

Scheduling Criteria

- CPU utilization – keep the CPU as busy as possible (Ideally Max)
- Throughput – # of processes that complete their execution per time unit (Ideally Max)
- Turnaround time – amount of time to execute a particular process (Ideally Min)
- Waiting time – amount of time a process has been waiting in the ready queue (Ideally Min)
- 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) (Ideally Min)

FCFS Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3



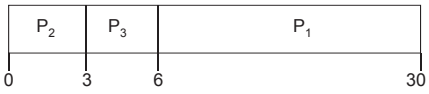
- If processes arrive in the order: P_1 , P_2 , P_3 ,
  The Gantt Chart for the schedule is given above:

  Waiting time for *P1* = 0; *P2* = 24; *P3* = 27
  Average waiting time: (0 + 24 + 27)/3 = 17

FCFS Scheduling (Cont.)

- If processes arrive in the order P_2 , P_3 , P_1

  The Gantt chart for the schedule is:



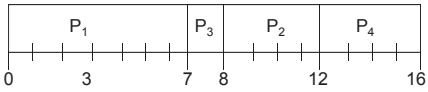
Waiting time for *P1* = 6*; P2* = 0*; P3* = 3
Average waiting time: (6 + 0 + 3)/3 = 3
Much better than previous case
Convoy effect short process behind long process

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
- Two schemes:
  - nonpreemptive – once CPU given to the process it cannot be preempted until completes its CPU burst
  - preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is know as the Shortest-Remaining-Time-First (SRTF)
- SJF is optimal – gives minimum average waiting time for a given set of processes

Example of Non-Preemptive SJF

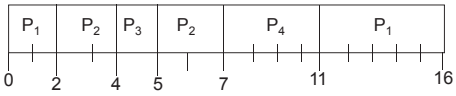
Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4



- SJF (non-preemptive)
- Average waiting time = (0 + 6 + 3 + 7)/4 = 4

Example of Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4



- SJF (non-preemptive)
- Average waiting time = (9 + 1 + 0 +2)/4 = 3

determining length of next CPU burst

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

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

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

Examples of Exponential Averaging

- α =0
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- α =1
  - $\tau_{n+1} = \alpha\, t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:

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

  $+ (1 - \alpha)^j \alpha\, t_{n-j} + \ldots$

  $+ (1 - \alpha)^{n+1} \tau_0$
- Since both α and (1 - α) are less than or equal to 1, each successive term has less weight than its predecessor

Priority Scheduling

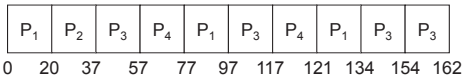
- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - nonpreemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem ≡ Starvation – low priority processes may never execute
- Solution ≡ 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 ⇒ FIFO
  - *q* small ⇒ *q* must be large with respect to context switch, otherwise overhead is too high

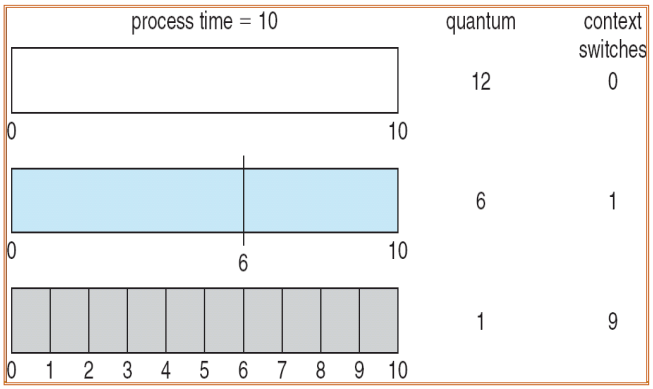
R-R Scheduling : Time Quantum=20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24



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

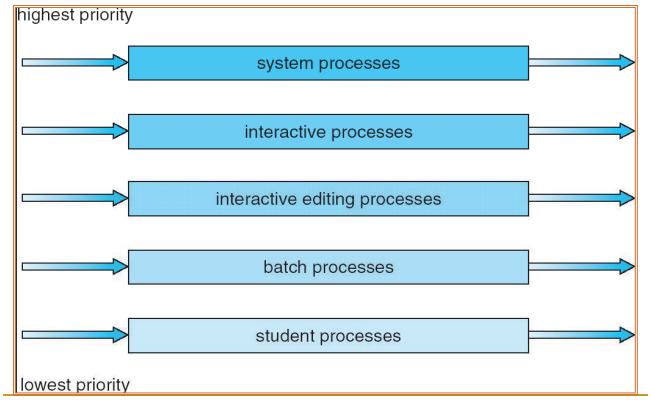
time quantum & context switch time



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



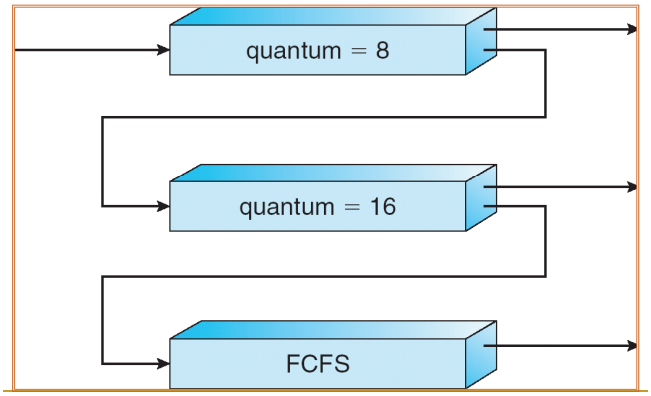
Multilevel Feedback Queue

- 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: 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



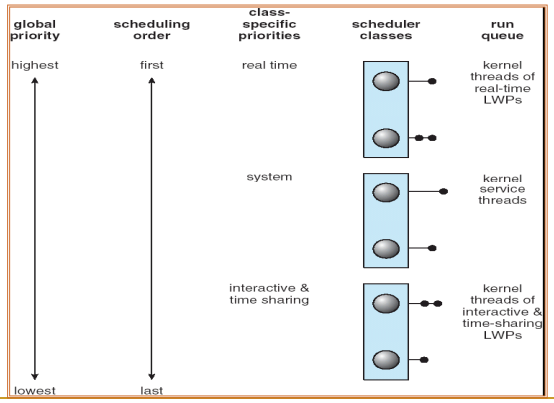
Scheduling Environments

- Multi processor scheduling
  - CPU scheduling more complex when multiple CPUs are available
  - *Homogeneous processors* within a multiprocessor
  - *Load sharing*
  - *Asymmetric multiprocessing* – only one processor accesses the system data structures, alleviating the need for data sharing

Scheduling Environments

- Real Time Scheduling
  - *Hard real-time* systems – required to complete a critical task within a guaranteed amount of time
  - *Soft real-time* computing – requires that critical processes receive priority over less fortunate ones
- Thread Scheduling
  - Local Scheduling – How the threads library decides which thread to put onto an available LWP
  - Global Scheduling – How the kernel decides which kernel thread to run next

OS Example: Solaris 2 Scheduling



OS Example: 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

OS Example: Linux Scheduling

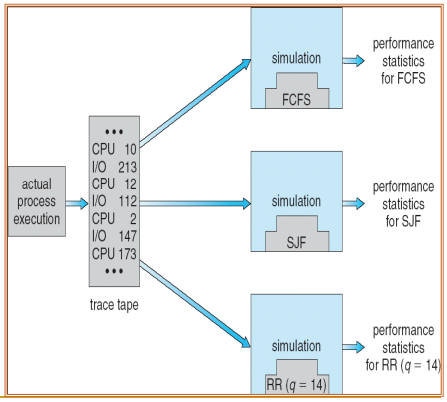
- Two algorithms: time-sharing and real-time
- Time-sharing
  - Prioritized credit-based – process with most credits is scheduled next
  - Credit subtracted when timer interrupt occurs
  - When credit = 0, another process chosen
  - When all processes have credit = 0, recrediting occurs
    - Based on factors including priority and history
- Real-time
  - Soft real-time
  - Posix.1b compliant – two classes
    - FCFS and RR
    - Highest priority process always runs first

Java Thread Scheduling

- JVM Uses a Preemptive, Priority-Based Scheduling Algorithm
- FIFO Queue is Used if There Are Multiple Threads With the Same Priority
- JVM Schedules a Thread to Run When:
  - The Currently Running Thread Exits the Runnable State
  - A Higher Priority Thread Enters the Runnable State
  - Note – the JVM Does Not Specify Whether Threads are Time-Sliced or Not
- Since the JVM Doesn't Ensure Time-Slicing, the yield() method may be used. This Yields Control to Another Thread of Equal Priority.

Algorithm Evaluation

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



Process Synchronization and Deadlocks

SunBeam Infotech

Background

- 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 regs1 = count {regs1 = 5}
  - S1: producer execute regs1 = regs1 + 1 {regs1 = 6}
  - S2: consumer execute regs2 = count {regs2 = 5}
  - S3: consumer execute regs2 = regs2 - 1 {regs2 = 4}
  - S4: producer execute count = regs1 {count = 6 }
  - S5: consumer execute count = regs2 {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 & 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. flag[i] = true implies that process P_i is ready!

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

Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore *S* – 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 Sync 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 exclusion
  - Semaphore S; // initialized to 1
  - wait (S);
    Critical Section
    signal (S);

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 crtical 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.

Deadlock and Starvation

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

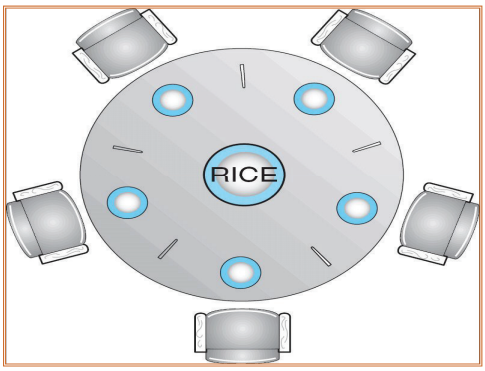
P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

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

Classical problems of synchronization

- Bounded-Buffer Problem
  - *N* buffers, each can hold one item
  - ??
- Readers and Writers Problem
  - A data set is shared among a number of concurrent processes
    - Readers – only read the data set
    - Writers – can both read and write.
  - Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time.
- Dining-Philosophers Problem
  - ??

Dining-Philosophers Problem



Problems with Semaphores

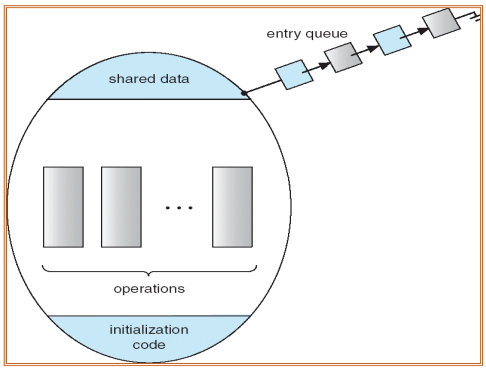
- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)

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

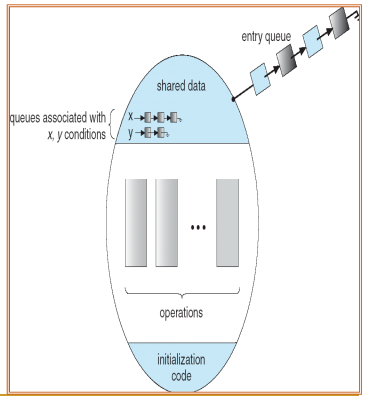
```
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (…) { …. }
            …
    procedure Pn (…) {……}
    Initialization code ( ….) { … }
            …
    }
}
```

Schematic view of a Monitor



Monitor with 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 ()



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:
  - disables interrupts to implement short critical sections
- 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

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

Log-Based Recovery

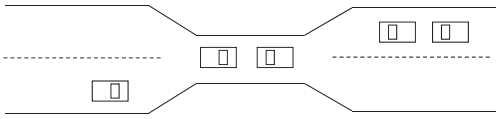
- Record to stable storage information about all modifications by a transaction
- Most common is write-ahead logging
  - Log on stable storage, each log record describes single transaction write operation, including
    - Transaction name
    - Data item name
    - Old value
    - New value
  - $<T_i$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

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. For 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	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.

Deadlock Characterization

- **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 $\{P_0, P_1, \ldots, P_0\}$ 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

  $P_2, \ldots, P_{n-1}$ 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.

Deadlock Prevention

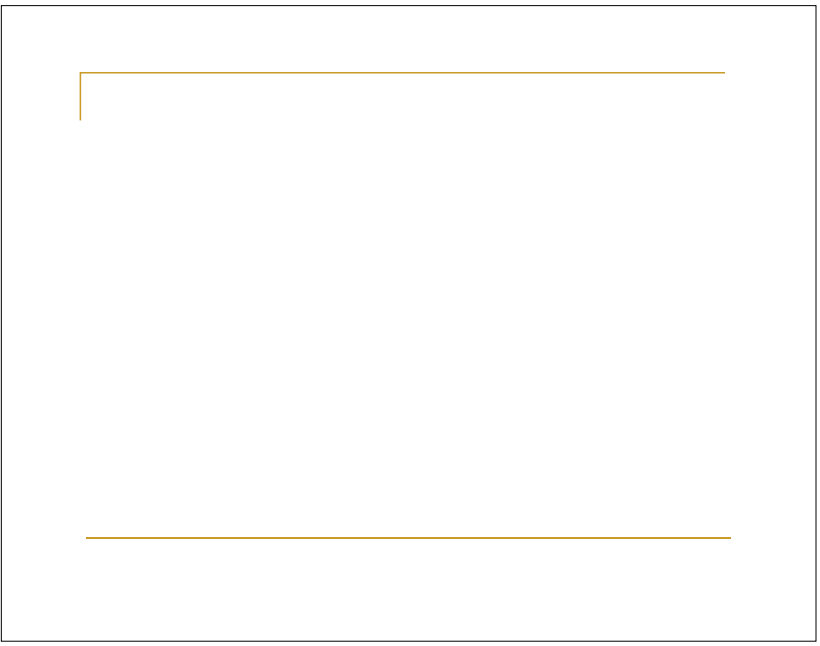
- **Mutual Exclusion** – not required for sharable resources; must hold for nonsharable resources.
- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources.
  - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none.
  - Low resource utilization; starvation possible.
- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.

Deadlock Prevention (Cont.)

- **No Preemption** –
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all 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.

Deadlock Avoidance

- 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.
- Avoidance Algorithm 1: Single instance of a resource type. Use a resource-allocation graph
- Avoidance Algorithm 2: Multiple instances of a resource type. Use the banker's algorithm



Main Memory

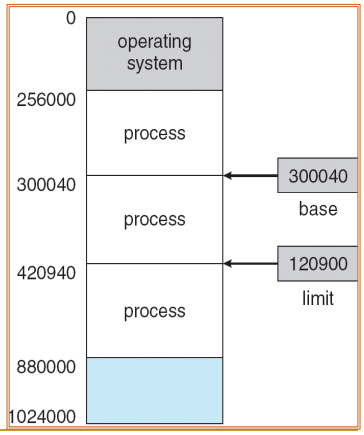
SunBeam Infotech

Background

- 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

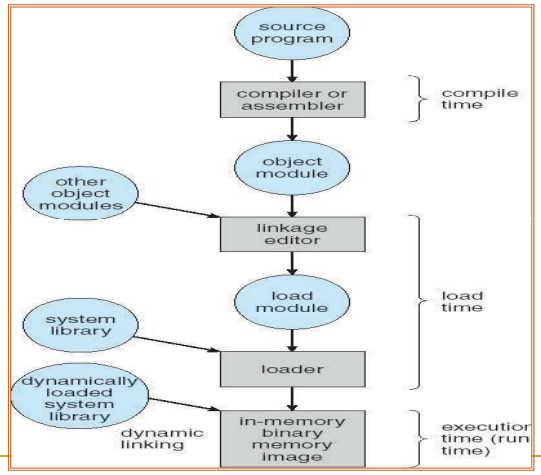
- A pair of **base** and **limit** registers define the logical address space



Binding of instructions & 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



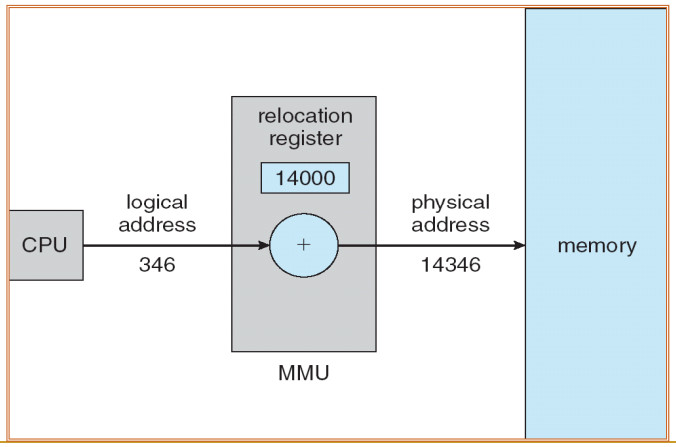
Logical vs. Physical Address Space

- 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 relocation regs



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

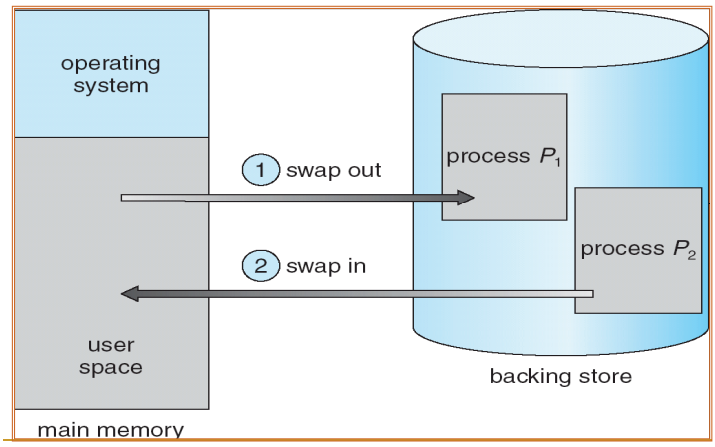
Dynamic Linking

- 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

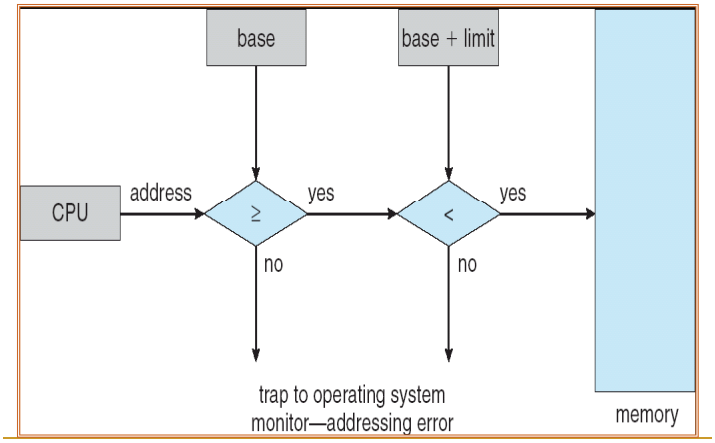
Schematic View of Swapping



Contiguous Allocation

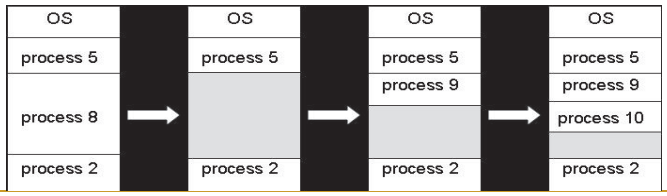
- Main memory usually into two partitions:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Base register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

HW address protection with base & limit regs



Contiguous Allocation (Cont.)

- 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

- How to satisfy a request of size n from a list of free holes
  - **First-fit**: Allocate the *first* hole that is big enough
  - **Best-fit**: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size
    - Produces the smallest leftover hole
  - **Worst-fit**: Allocate the *largest* hole; must also search entire list
    - Produces the largest leftover hole
- First-fit and best-fit better than worst-fit in terms of speed and storage utilization

Fragmentation

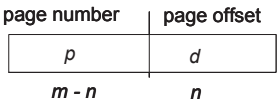
- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- Reduce external fragmentation 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 0.5 KB and 8 KB)
- 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

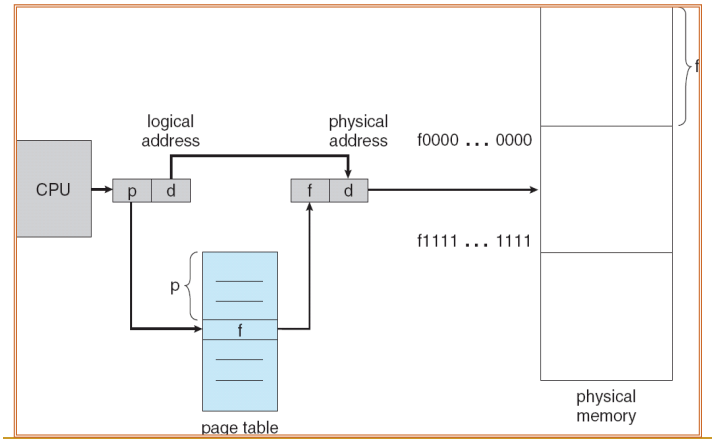
Address Translation Scheme

- 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

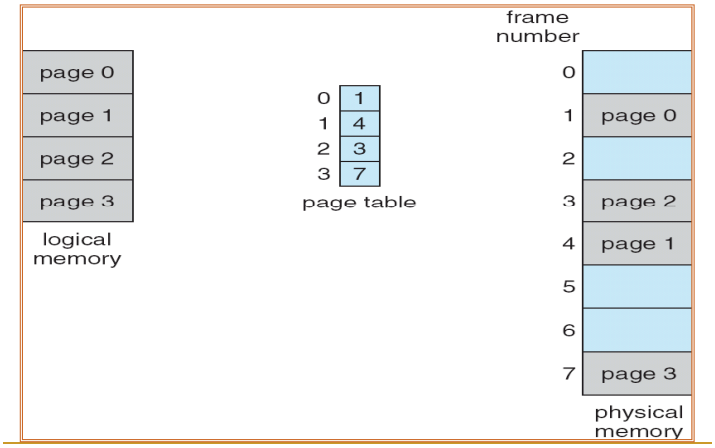


For given logical address space 2*m and page size* 2n

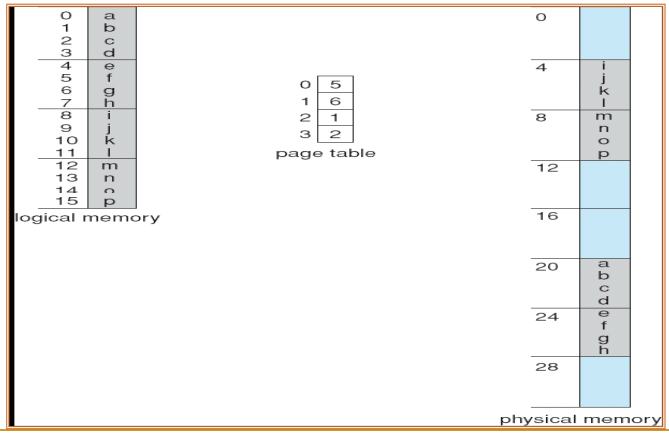
Paging Hardware



Paging model of logical & physical memory



Paging Example (32-byte memory and 4-byte pages)



Free Frames

(a) Before allocation (b) After allocation

Implementation of Page Table

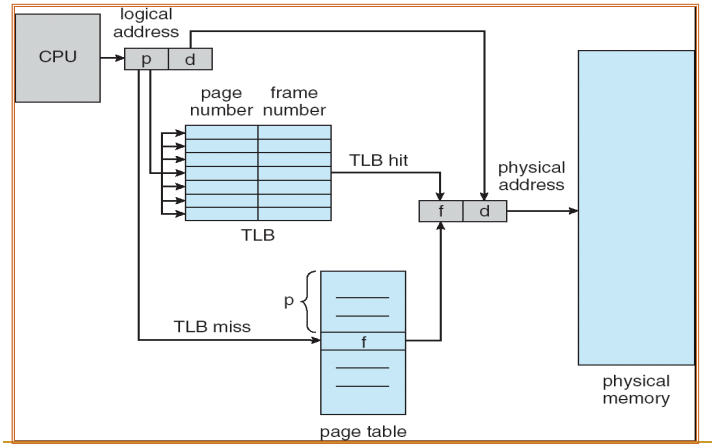
- Page table is kept in main memory
- **Page-table base register (PTBR)** points to the page table
- **Page-table length register (PRLR)** indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.
- 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

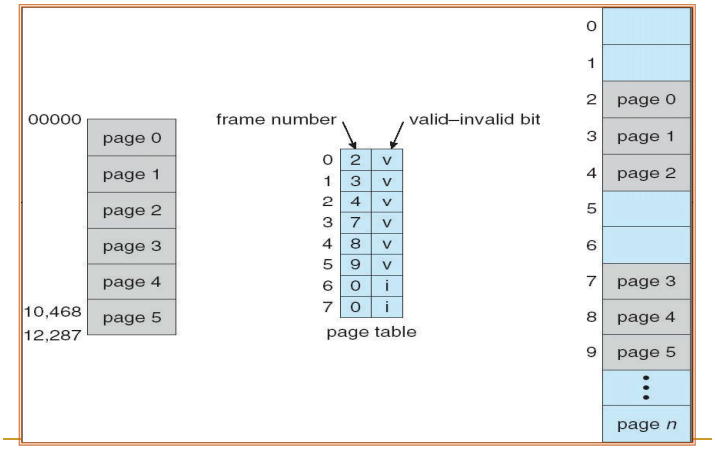
- Associative Lookup = ε time unit
- Assume memory cycle time is 1 microsecond
- Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- Hit ratio = α
- **Effective Access Time** (EAT)

$$EAT = (1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha)$$
$$= 2 + \varepsilon - \alpha$$

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

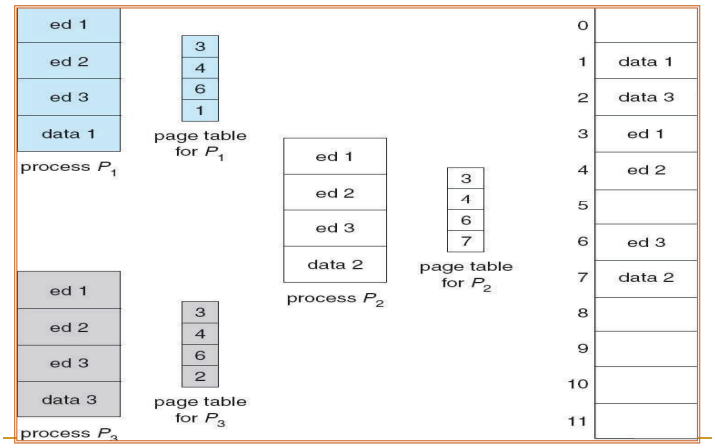
Valid (v) / Invalid (i) Bit in page table



Shared Pages

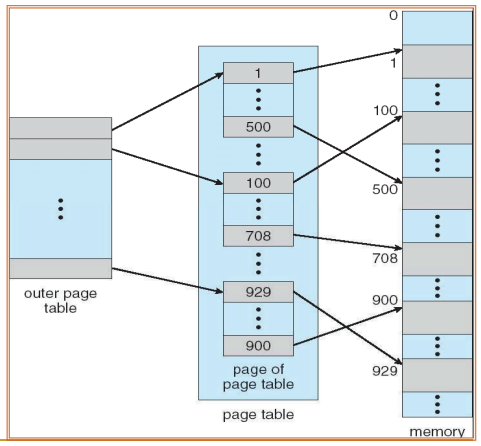
- **Shared code**
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
  - Shared code must appear in same location in the logical address space of all processes
- **Private code and data**
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space

Shared Pages Example



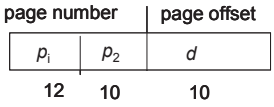
Structure of page table : Hierarchical

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



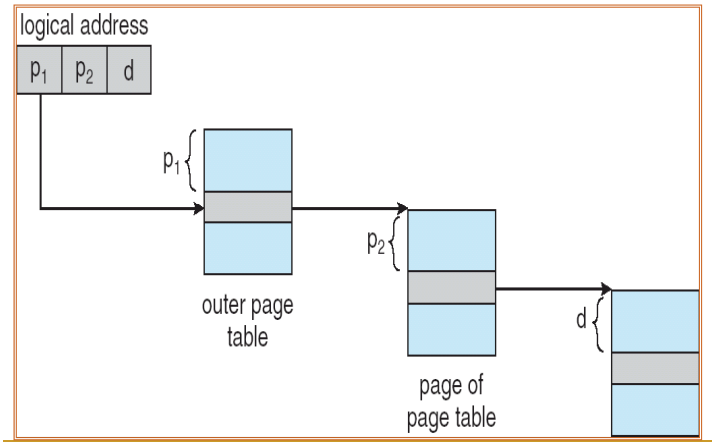
Two-Level Paging Example

- 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 *pi* is an index into the outer page table,
and *p2* is the displacement within the page of the outer page table

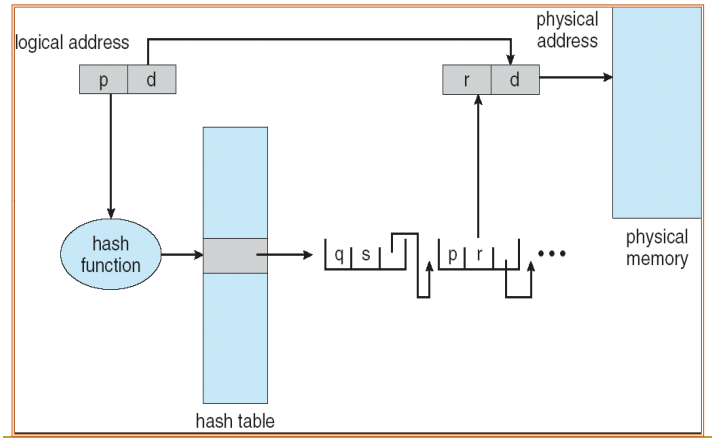
Address-Translation Scheme



Structure of page table : Hashed

- 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.

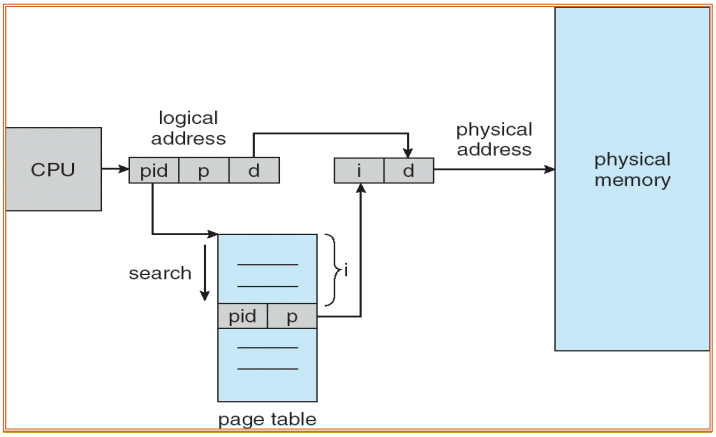
Hashed Page Table



Structure of page table : Inverted

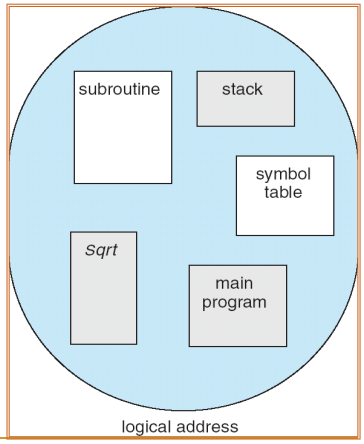
- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference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

Inverted Page Table Architecture

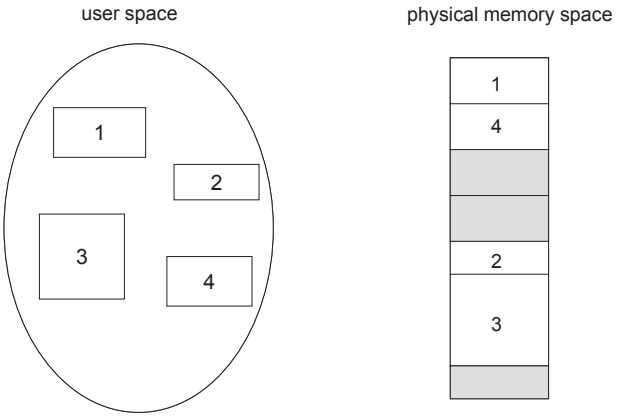


Segmentation

- 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



Logical View of Segmentation



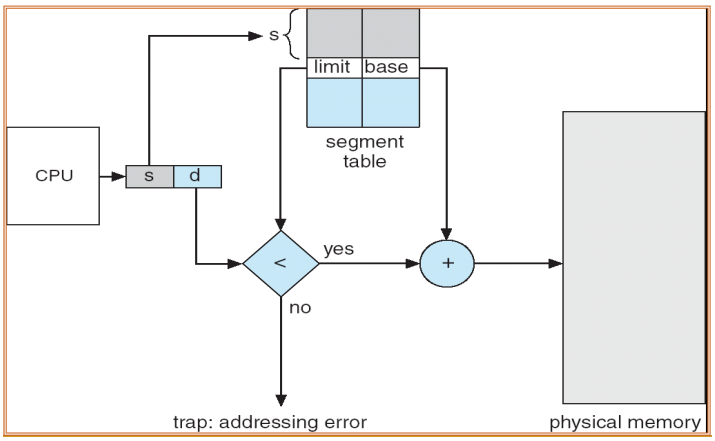
Segmentation Architecture

- Logical address consists of a two tuple:
  - <segment-number, offset>
- **Segment table** – maps two-dimensional physical addresses; each table 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*** < **STLR**

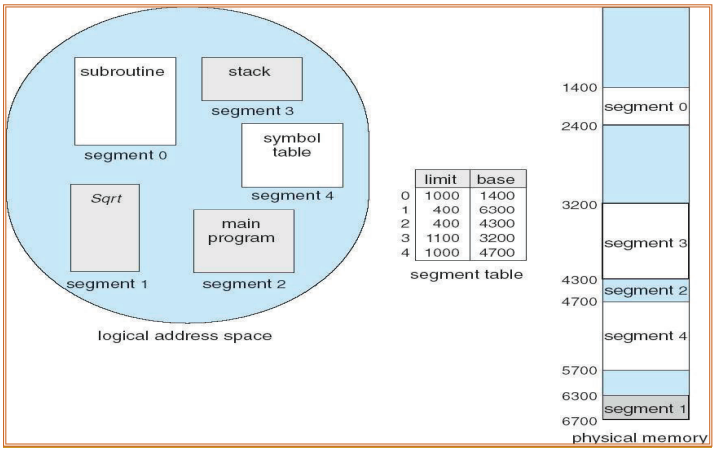
Segmentation Architecture (Cont.)

- 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

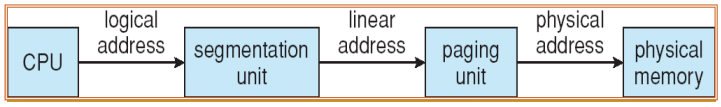


Example of Segmentation

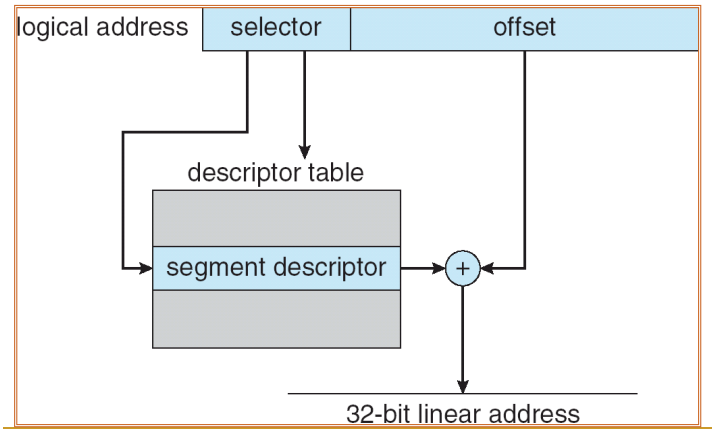


Example: The Intel Pentium

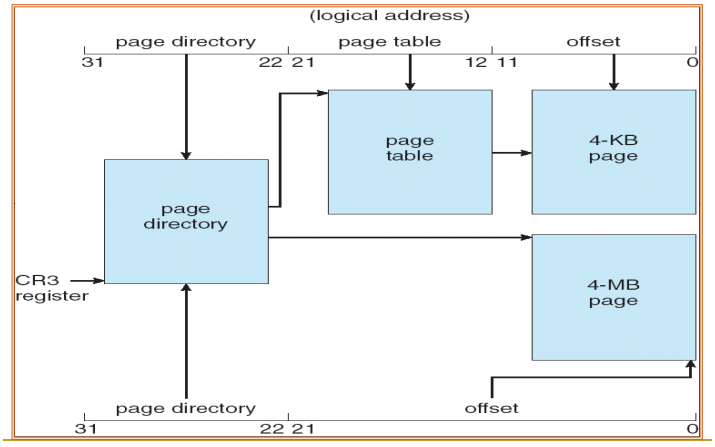
- Supports both segmentation and segmentation with paging
- CPU generates logical address
  - Given to segmentation unit
    - Which produces linear addresses
  - Linear address given to paging unit
    - Which generates physical address in main memory
    - Paging units form equivalent of MMU



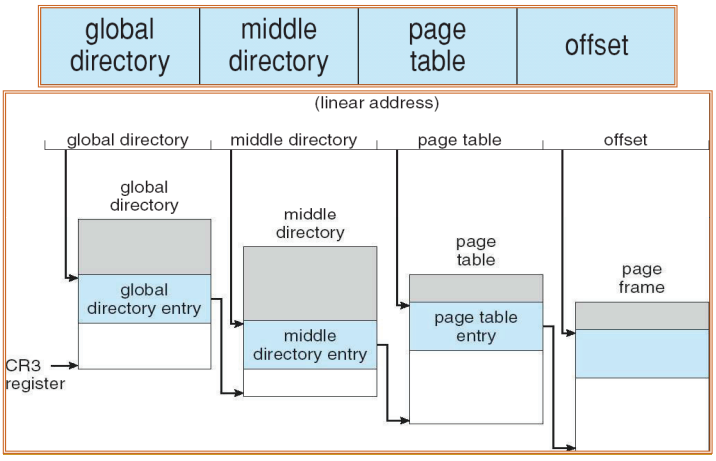
Intel Pentium Segmentation



Pentium Paging Architecture



Linux : linear address & 3 level paging



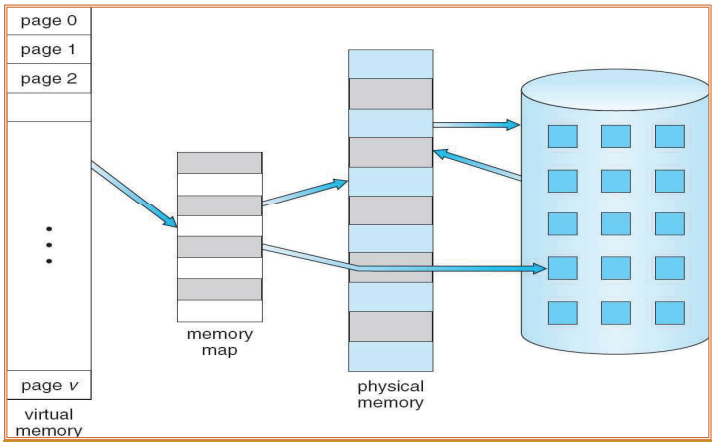
Virtual Memory

SunBeam Infotech

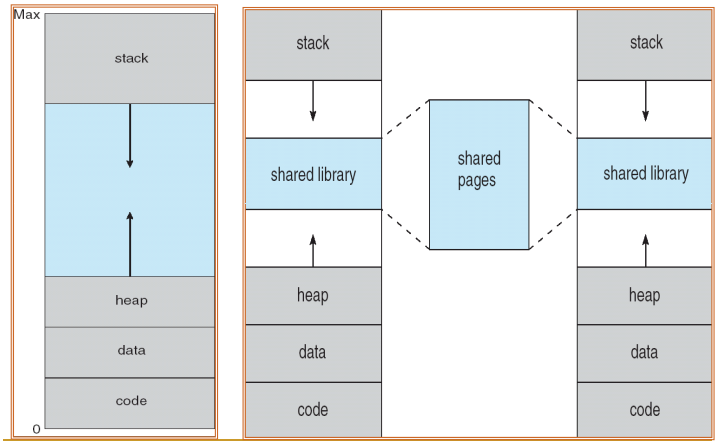
Background

- **Virtual memory** – separation of user logical memory from physical memory.
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

virtual memory larger than physical memory



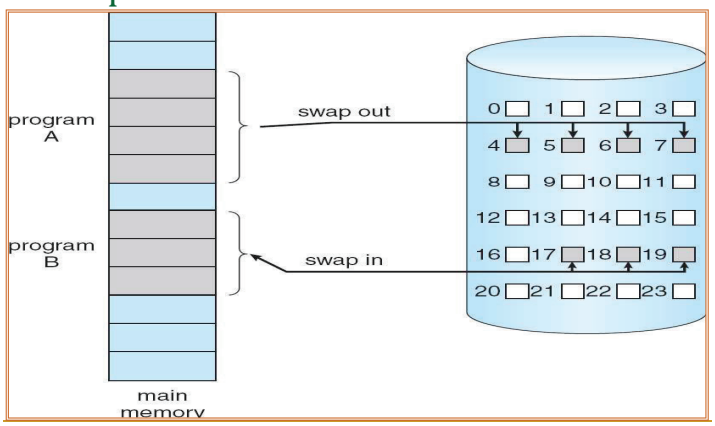
Virtual-address Space



Demand Paging

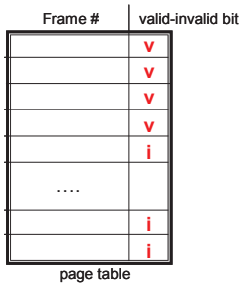
- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users
- Page is needed => reference to it
  - invalid reference => abort
  - not-in-memory => bring to memory
- Lazy swapper – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a pager

Transfer of a Paged Memory to Contiguous Disk Space

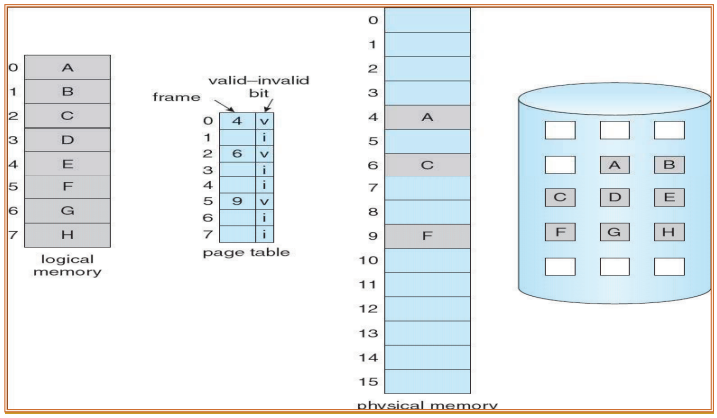


Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated
  - (**v** ⇒ in-memory, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:
- During address translation, if valid–invalid bit in page table entry is **I** ⇒ page fault



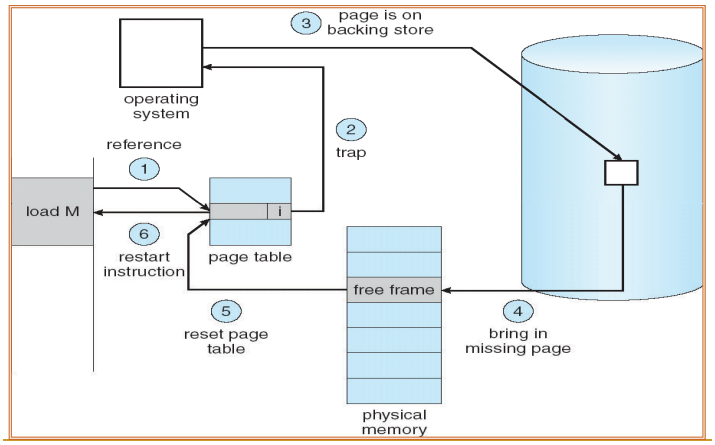
Page Table When Some Pages Are Not in Main Memory



Page Fault

- If there is a reference to a page, first reference to that page will trap to operating system:
  page fault
1. Operating system looks at another table to decide:
   - Invalid reference ⇒ abort
   - Just not in memory
2. Get empty frame
3. Swap page into frame
4. Reset tables
5. Set validation bit = **v**
6. Restart the instruction that caused the page fault

Steps in Handling a Page Fault



Performance of Demand Paging

- Page Fault Rate 0 => p => 1.0
  - if p = 0 no page faults
  - if p = 1, every reference is a fault
- Effective Access Time (EAT)

  EAT = (1 – p) x memory access
  + p (page fault overhead
  + swap page out
  + swap page in
  + restart overhead
)

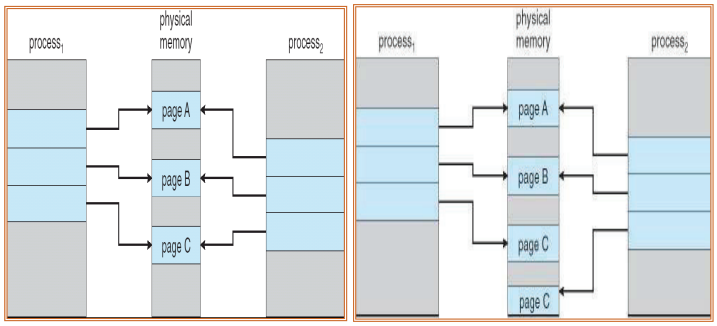
Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- EAT = (1 – p) x 200 + p (8 milliseconds)
  = (1 – p x 200 + p x 8,000,000
  = 200 + p x 7,999,800
- If one access out of 1,000 causes a page fault, then EAT = 8.2 microseconds.
  This is a slowdown by a factor of 40!!

Copy-on-Write

- One more benefit of virtual memory is **Copy-on-Write**
- Copy-on-Write (COW) allows both parent and child processes to initially *share* the same pages in memory
- If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied
- Free pages are allocated from a **pool** of zeroed-out pages

Page C is Modified, then Copied

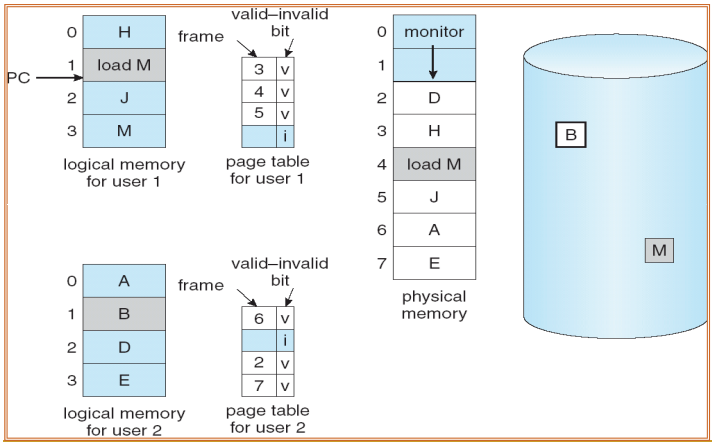


- If process1 modifies page C, then a copy is created for that page and thus each page will use separate copy of the page

Page Replacement

- If there is no free frame available, page is replaced.
- It finds some page in memory, but not really in use, swap it out using some algorithm, which will result in minimum number of page faults
- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement
- Use modify (dirty) bit to reduce overhead of page transfers – only modified pages are written to disk
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory

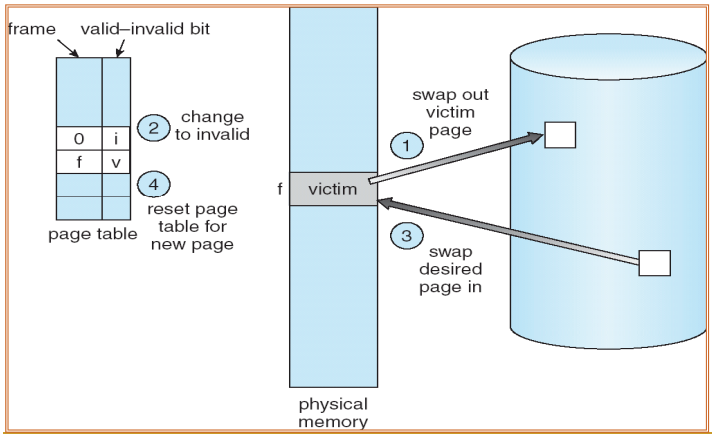
Need For Page Replacement



Basic Page Replacement

- Step 1: Find the location of the desired page on disk
- Step 2: Find a free frame:
  - If there is a free frame, use it
  - If there is no free frame, use a page replacement algorithm to select a victim frame
- Step 3: Bring the desired page into the (newly) free frame; update the page and frame tables
- Step 4: Restart the process

Page Replacement

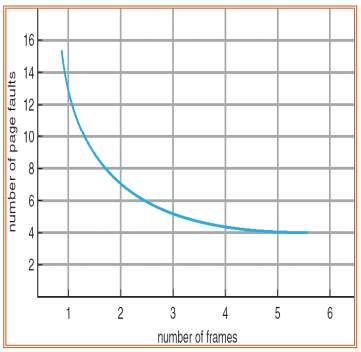


Page Replacement Algorithms

- Want lowest page-fault rate
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
- In all our examples, the reference string is

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Graph of Page Faults Versus The Number of Frames



First-In-First-Out (FIFO) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process)

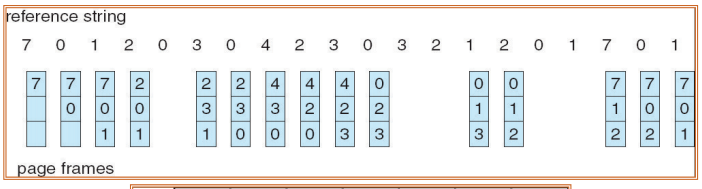
1	1	4	5		
2	2	1	3	9 page faults	
3	3	2	4		

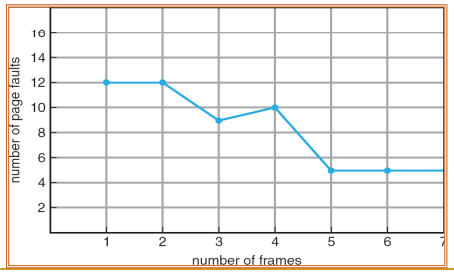
- 4 frames

1	1	5	4		
2	2	1	5	10 page faults	
3	3	2			
4	4	3			

- Belady's Anomaly: more frames ⇒ more page faults

FIFO Page Replacement



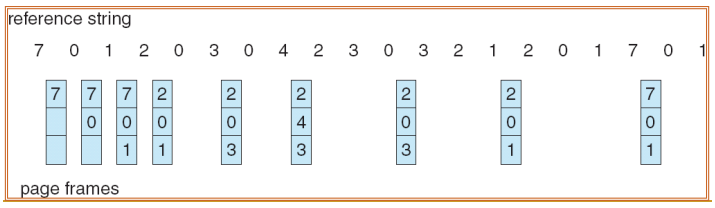


Optimal Algorithm

- Replace page that will not be used for longest period of time
- 4 frames example

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

- How do you know this?
- Used for measuring how well your algorithm performs



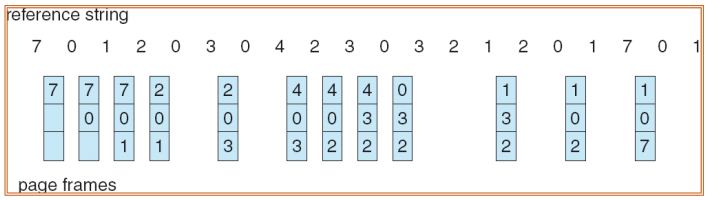
Least Recently Used (LRU) Algo

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

1	1	1	1	**5**
2	2	2	2	2
3	**5**	5	**4**	4
4	4	**3**	3	3

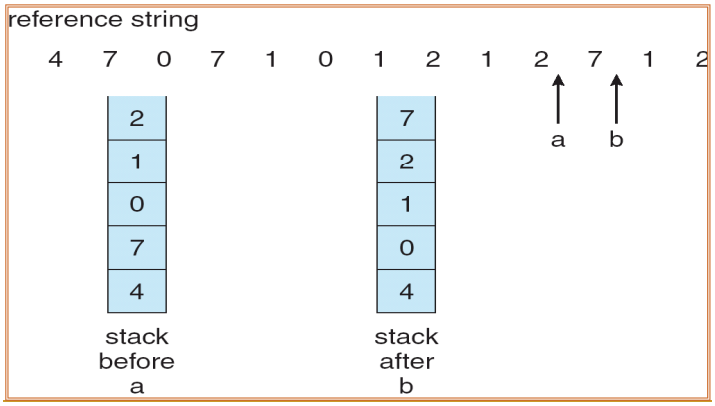
- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to determine which are to change

LRU Page Replacement



- Stack implementation – keep a stack of page numbers in a double link form:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - No search for replacement

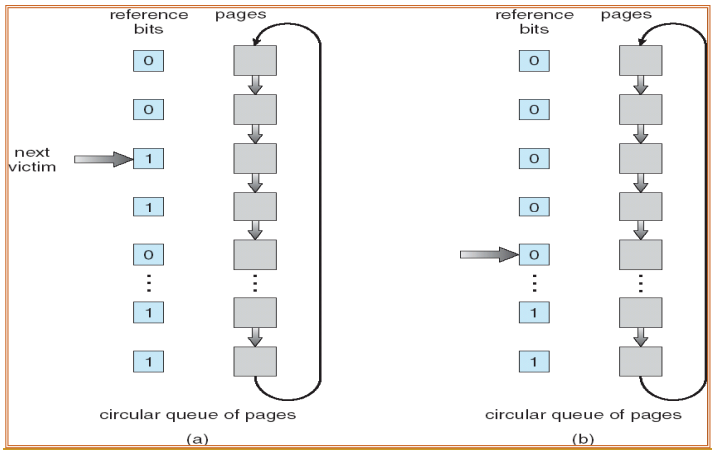
Use of Stack to record most recent page references



LRU Approximation Algorithms

- Reference bit
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace the one which is 0 (if one exists)
    - We do not know the order, however
- Second chance
  - Need reference bit
  - Clock replacement
  - If page to be replaced (in clock order) has reference bit = 1 then:
    - set reference bit 0
    - leave page in memory
    - replace next page (in clock order), subject to same rules

2nd Chance (clock) Page-Replacement Algo



Counting Algorithms

- Keep a counter of the number of references that have been made to each page
- **LFU Algorithm**: replaces page with smallest count
- **MFU Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used

Allocation of Frames

- Each process needs minimum number of pages
- Example: IBM 370 – 6 pages to handle SS MOVE instruction:
  - instruction is 6 bytes, might span 2 pages
  - 2 pages to handle from
  - 2 pages to handle to
- Two major allocation schemes
  - Fixed allocation
    - Equal or proportional allocation
  - priority allocation
    - Use a proportional allocation using priorities size

Priority Allocation

- Use a proportional allocation scheme using priorities
- If process P_i generates a page fault,
  - select for replacement one of its frames
  - select for replacement a frame from a process with lower priority number
- Page replacement policies
  - **Global replacement:** process selects a replacement frame from the set of all frames; one process can take a frame from another
  - **Local replacement:** each process selects from only its own set of allocated frames

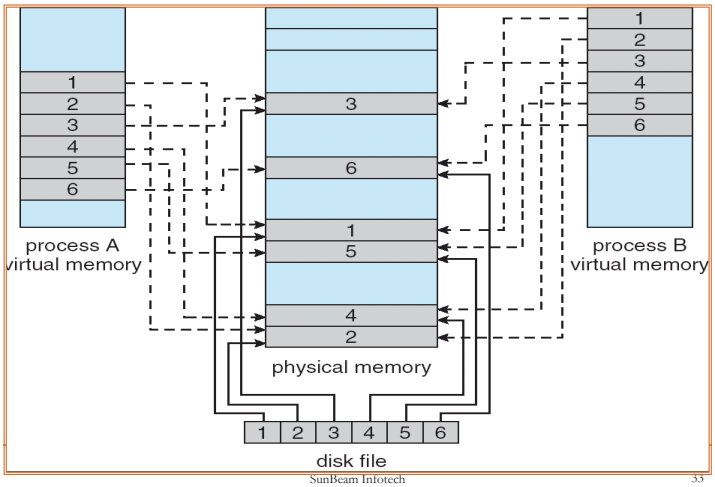
Thrashing

- If a process does not have "enough" pages, the page-fault rate is very high. This leads to:
  - low CPU utilization
  - operating system thinks that it needs to increase the degree of multiprogramming
  - another process added to the system
- **Thrashing** ≡ a process is busy swapping pages in & out
- Why does demand paging work?
  - Process migrates from one locality to another
  - Localities may overlap
- Why does thrashing occur?
  - Σ size of locality > total memory size

Memory-Mapped Files

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by **mapping** a disk block to a page in memory
- A file is initially read using demand paging. A page-sized portion of the file is read from the file system into a physical page. Subsequent reads/writes to/from the file are treated as ordinary memory accesses.
- Simplifies file access by treating file I/O through memory rather than **read() write()** system calls
- Also allows several processes to map the same file allowing the pages in memory to be shared

Memory Mapped Files



Windows XP : Memory

- Uses demand paging with **clustering**. Clustering brings in pages surrounding the faulting page.
- Processes are assigned **working set minimum** and **working set maximum**
- Working set minimum is the minimum number of pages the process is guaranteed to have in memory
- A process may be assigned as many pages up to its working set maximum
- When the amount of free memory in the system falls below a threshold, **automatic working set trimming** is performed to restore the amount of free memory
- Working set trimming removes pages from processes that have pages in excess of their working set minimum

Solaris

- Maintains a list of free pages to assign faulting processes
- *Lotsfree* – threshold parameter (amount of free memory) to begin paging
- *Desfree* – threshold parameter to increasing paging
- *Minfree* – threshold parameter to being swapping
- Paging is performed by *pageout* process
- Pageout scans pages using modified clock algorithm
- *Scanrate* is the rate at which pages are scanned. This ranges from *slowscan* to *fastscan*
- Pageout is called more frequently depending upon the amount of free memory available

File-System Interface

SunBeam Infotech

File Concept

- Named collection of related information that is recorded on secondary storage device.
- Contiguous logical address space
- Types:
  - Data : numeric, character, binary
  - Program
- File has certain defined structure according to its type.
- At a advanced step, every thing in the system (except process) can be understood as file.

File Structure

- None - sequence of words, bytes
- Simple record structure
  - Fixed length
  - Variable length
- Complex Structures
  - Formatted document
  - Relocatable load file
- Who decides file structure:
  - Operating system
  - Program

File Attributes

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

File Operations

- File is an **abstract data type**
- Regular file operations
  - Create
  - Write
  - Read
  - Reposition within file
  - Delete
  - Truncate
- *Open*(F_i) – search the directory structure on disk for entry F_i, and move the content of entry to memory
- *Close* (F_i) – move the content of entry F_i in memory to directory structure on disk

Opening Files

- Several pieces of data are needed to manage open files:
  - File pointer: pointer to last read/write location, per process that has the file open
  - File-open count: counter of number of times a file is open – to allow removal of data from open-file table when last processes closes it
  - Disk location of the file: cache of data access information
  - Access rights: per-process access mode information
- File locking is provided by some operating systems and file systems and it mediates access to a file

File Types – Name, Extension

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

Access Methods

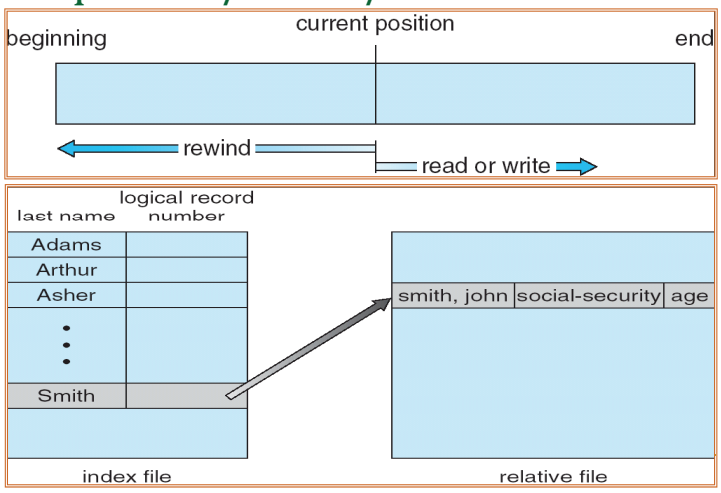
- **Sequential Access**

  read next
  write next
  reset
  no read after last write
  (rewrite)

- **Direct Access**

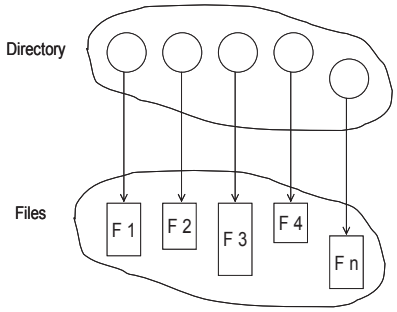
  read *n*
  write *n*
  position to *n*
  read next
  write next
  rewrite *n*
  (*n* = relative block number)

Sequential/ Index/ Relative files

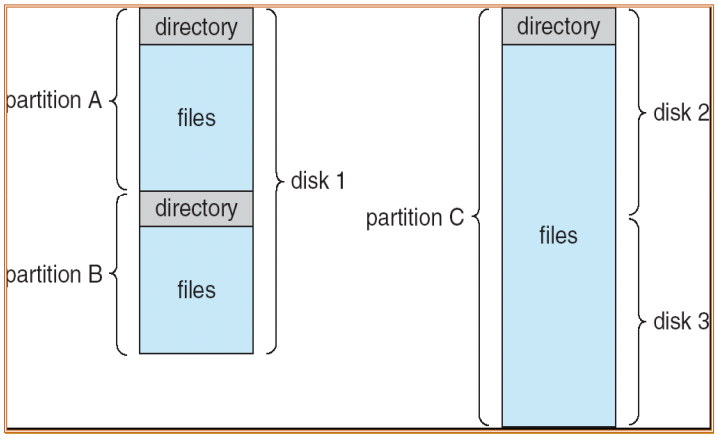


Directory Structure

- A collection of nodes containing information about all files
- Both the directory structure and the files reside on disk
- Backups of these two structures are kept on tapes (magnetic or optic)



A Typical File-system Organization



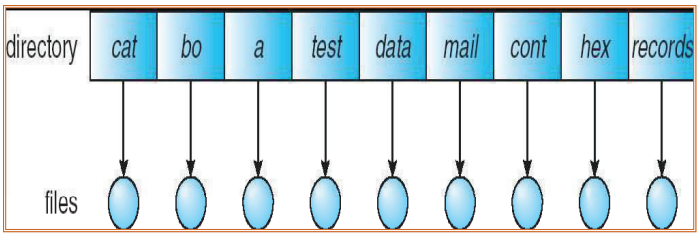
Directory Operations

- Directory is an **abstract data type**
- Regular operations on directories
  - Search for a file
  - Create a file
  - Delete a file
  - List a directory
  - Rename a file
  - Traverse the file system
- From the OS view, directory is a file that store information about its sub directories and content files. However this file cannot be edited by user.

Organize the Directory For

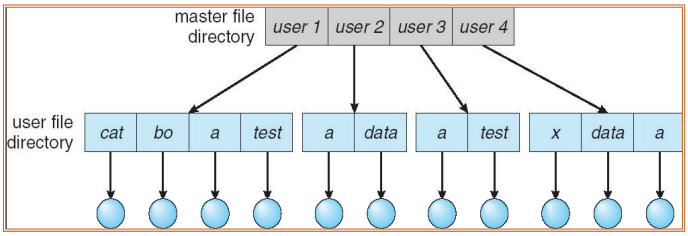
- Efficiency – locating a file quickly
- Naming – convenient to users
  - Two users can have same name for different files
  - The same file can have several different names
- Grouping – logical grouping of files by properties, (e.g., all Java programs, all games, …)

Single-Level Directory



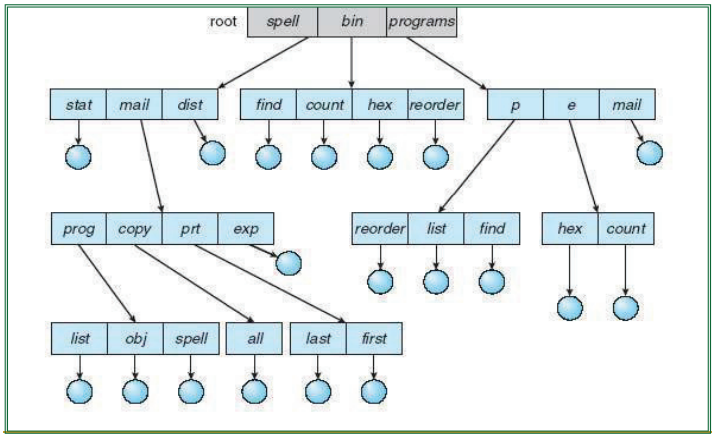
- A single directory for all users
- Naming problem
- Grouping problem

Two-Level Directory



- Separate directory for each user
- Path name
- Can have the same file name for different user
- Efficient searching
- No grouping capability

Tree-Structured Directories

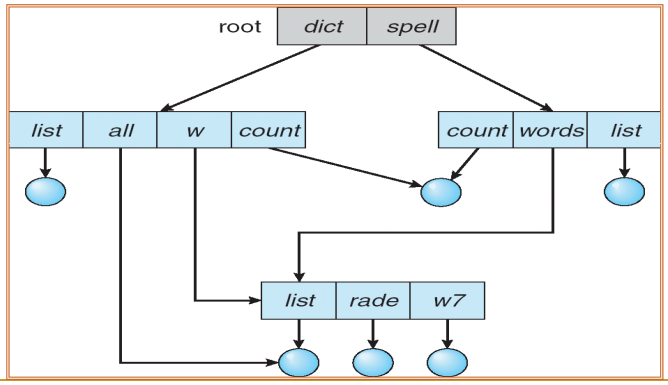


Tree-Structured Directories (Cont)

- Efficient searching
- Grouping Capability
- Current directory (working directory)
  - cd /spell/mail/prog
- **Absolute** or **relative** path name
- Creating a new file is done in current directory
- Creating a new subdirectory is done in current directory
- Delete a file from current directory

Acyclic-Graph Directories

- Have shared subdirectories and files

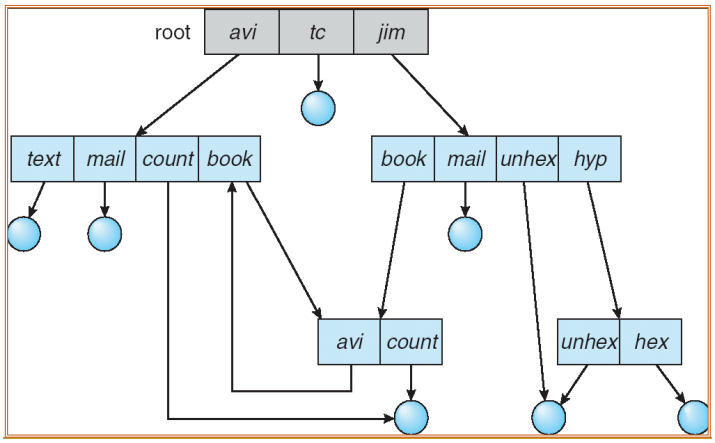


Acyclic-Graph Directories (Cont.)

- Two different names (aliasing)
- If *dict* deletes *list* ⇒ dangling pointer

  Solutions:
  - Backpointers, so we can delete all pointers
    Variable size records a problem
  - Backpointers using a daisy chain organization
  - Entry-hold-count solution
- New directory entry type
  - **Link** – another name (pointer) to an existing file
  - **Resolve the link** – follow pointer to locate the file

General Graph Directory

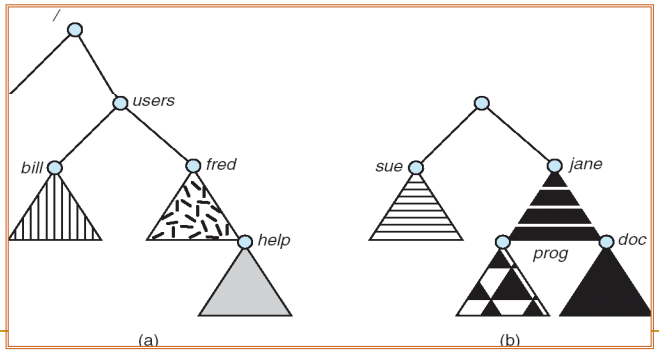


General Graph Directory (Cont.)

- How do we guarantee no cycles?
  - Allow only links to file not subdirectories
  - Garbage collection
  - Every time a new link is added use a cycle detection algorithm to determine whether it is OK

File System Mounting

- A file system must be **mounted** before it can be accessed
- A unmounted file system (b) is mounted at a **mount point**



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 file-sharing method
- **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
- 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

Protection

- File owner/creator should be able to control:
  - what can be done
  - by whom
- Types of access
  - **Read**
  - **Write**
  - **Execute**
  - **Append**
  - **Delete**
  - **List**

Access Lists and Groups

- Mode of access: read, write, execute
- Three classes of users

			RWX
a) **owner access**	7	⇒	1 1 1
b) **group access**	6	⇒	1 1 0
c) **public access**	1	⇒	0 0 1

- Ask manager to create a group (unique name), say G, and add some users to the group.
- For a particular file (say *game*) or subdirectory, define an appropriate access.

A Sample UNIX Directory Listing

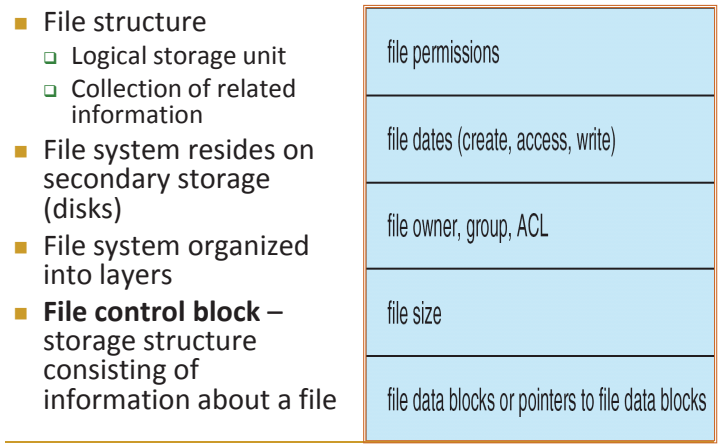
-rw-rw-r--	1 pbg	staff	31200	Sep 3 08:30	intro.ps
drwx------	5 pbg	staff	512	Jul 8 09.33	private/
drwxrwxr-x	2 pbg	staff	512	Jul 8 09:35	doc/
drwxrwx---	2 pbg	student	512	Aug 3 14:13	student-proj/
-rw-r--r--	1 pbg	staff	9423	Feb 24 2003	program.c
-rwxr-xr-x	1 pbg	staff	20471	Feb 24 2003	program
drwx--x--x	4 pbg	faculty	512	Jul 31 10:31	lib/
drwx------	3 pbg	staff	1024	Aug 29 06:52	mail/
drwxrwxrwx	3 pbg	staff	512	Jul 8 09:35	test/

File System Implementation

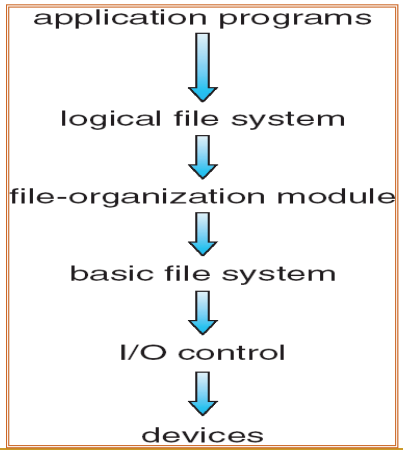
SunBeam Infotech

File-System Structure

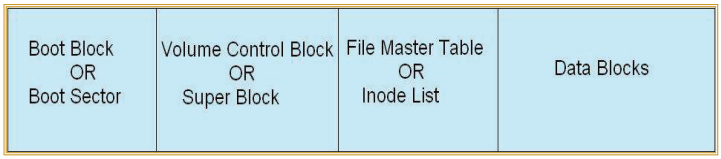
- File structure
  - Logical storage unit
  - Collection of related information
- File system resides on secondary storage (disks)
- File system organized into layers
- **File control block** – storage structure consisting of information about a file



Layered File System

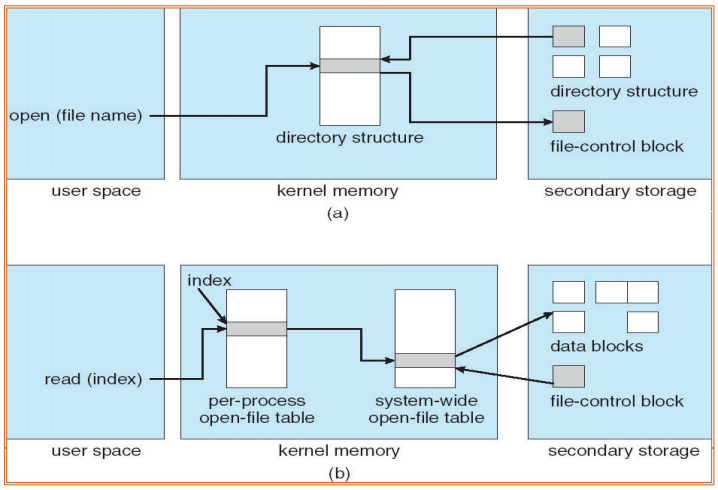


File System Implementation



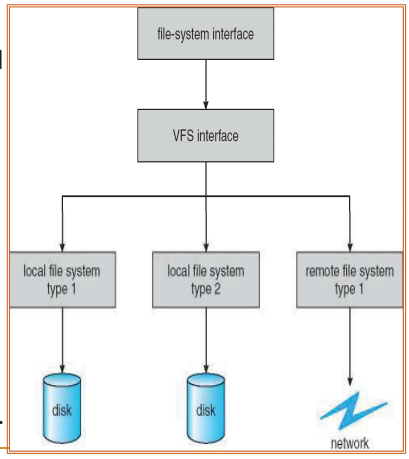
- **Boot Block** may contains bootstrap program to boot OS.
- **Volume Control Block** contains information about the blocks in that volume. It also contains info / pointers to free blocks.
- **File Master table** contains **File control blocks** for all files. File control block keeps info about files and pointer to data blocks.
- **Data block** contains data of the file.

In-Memory File System Structures



Virtual File Systems

- Virtual File Systems (VFS) provide an object-oriented way of implementing file systems.
- VFS allows the same system call interface (the API) to be used for different types of file systems.
- The API is to the VFS interface, rather than any specific type of file system.



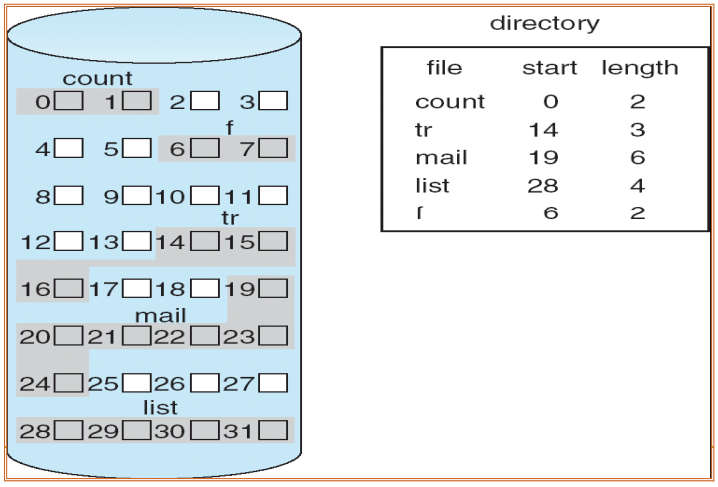
Directory Implementation

- **Linear list** of file names with pointer to the data blocks.
  - simple to program
  - time-consuming to execute
- **Hash Table** – linear list with hash data structure.
  - decreases directory search time
  - **collisions** – situations where two file names hash to the same location
  - fixed size

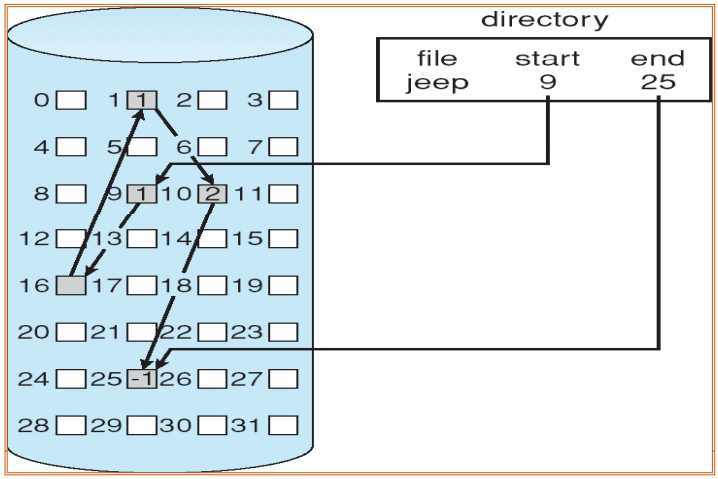
Allocation Methods

- An allocation method refers to how disk blocks are allocated for files.
- Contiguous allocation
  - Each file occupies a set of contiguous blocks on the disk
  - Starting location and size stored, random access
  - Wastage of space, File cannot grow
- Linked allocation
  - Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk.
  - starting address stored, no wastage, no random access
- Indexed allocation
  - Brings all pointers together into the **index block**.
  - Index table, Random access, index table blocks linking

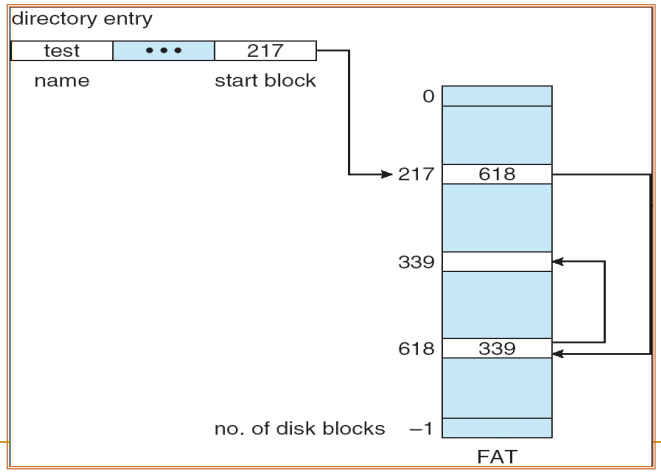
Contiguous Allocation of Disk Space



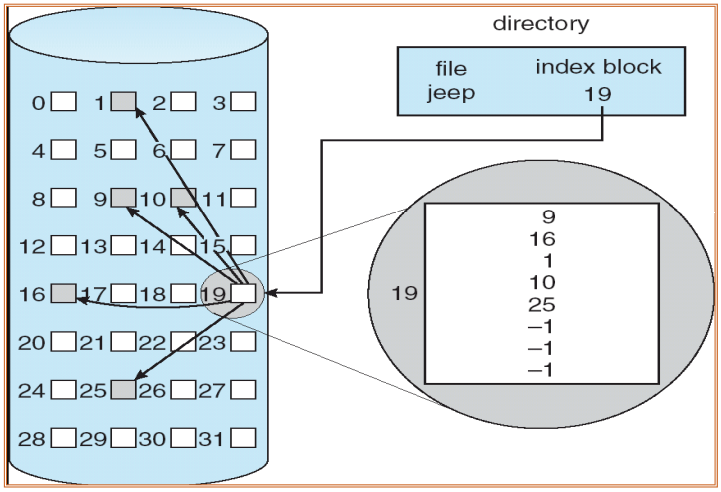
Linked Allocation



File-Allocation Table

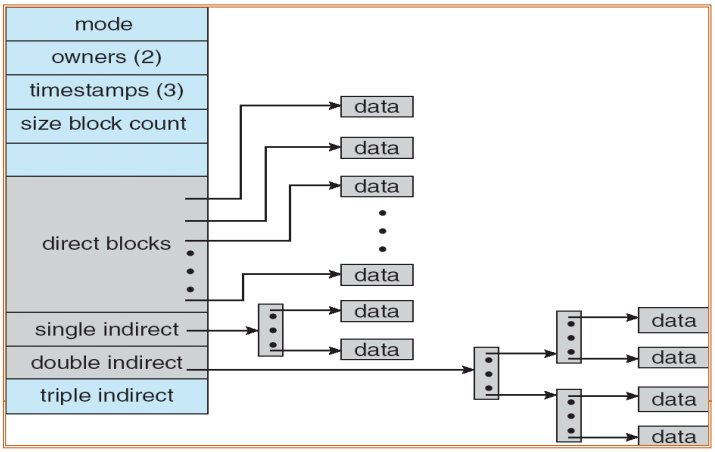


Indexed Allocation



Combined Scheme

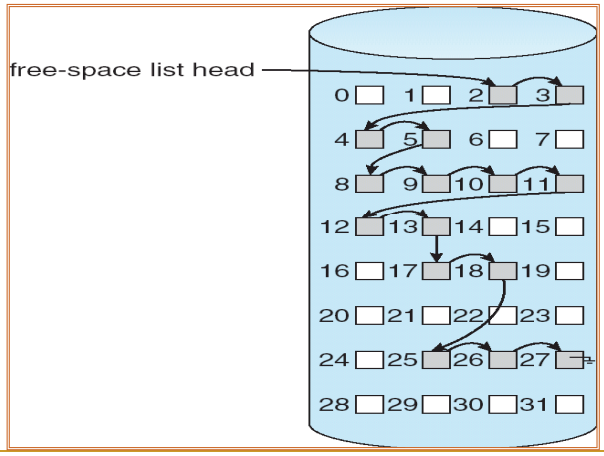
UNIX (4K bytes per block)



Free-Space Management

- To reuse the space of deleted files, it is necessary to maintain free space list.
- Bit vector or Bit map
  - Each block is represented as a bit (0: empty, 1: filled).
- Linked list
  - Each free block keep the address of next free block.
- Grouping
  - The addresses of group of free blocks kept in last block.
- Counting
  - The address of free block and count is kept in last block.

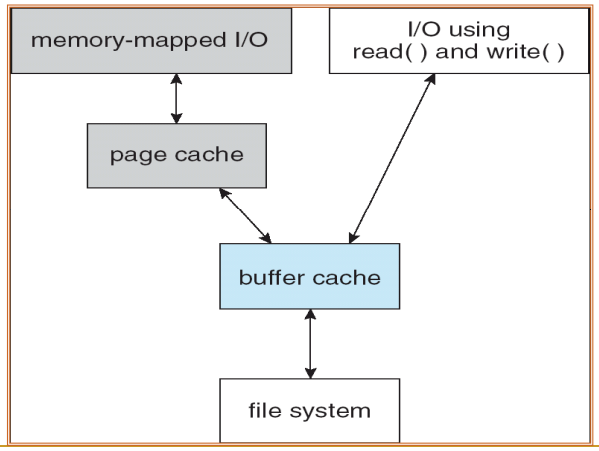
Linked Free Space List on Disk



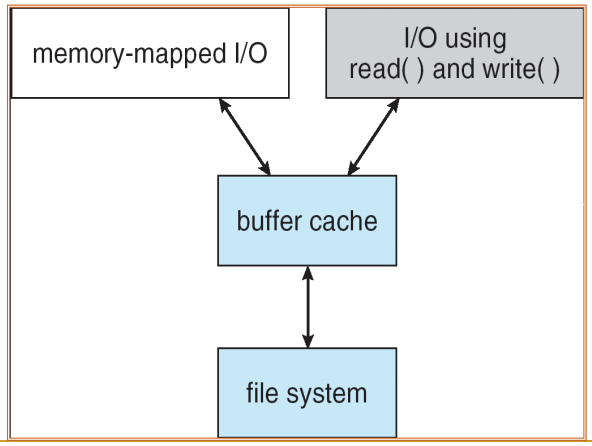
Efficiency and Performance

- Efficiency dependent on:
  - disk allocation and directory algorithms
  - types of data kept in file's directory entry
- Performance
  - disk cache – separate section of main memory for frequently used blocks
  - free-behind and read-ahead – techniques to optimize sequential access
  - improve PC performance by dedicating section of memory as virtual disk, or RAM disk

I/O Without a Unified Buffer Cache



I/O Using a Unified Buffer Cache



Recovery

- Consistency checking – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies
- Use system programs to **back up** data from disk to another storage device (floppy disk, magnetic tape, other magnetic disk, optical)
- Recover lost file or disk by **restoring** data from backup

Log Structured File Systems

- Log structured (or journaling) file systems record each update to the file system as a transaction
- All transactions are written to a log
  - A transaction is considered committed once it is written to the log
  - However, the file system may not yet be updated
- The transactions in the log are asynchronously written to the file system
  - When the file system is modified, the transaction is removed from the log
- If the file system crashes, all remaining transactions in the log must still be performed

The Sun Network File System (NFS)

- An implementation and a specification of a software system for accessing remote files across LANs
- The implementation is part of the Solaris and SunOS operating systems running on Sun workstations using an unreliable datagram protocol (UDP/IP)
- Interconnected workstations viewed as a set of independent machines with independent file systems, which allows sharing (mounting) among these file systems in a transparent manner
- NFS is designed to operate in a heterogeneous environment of different machines, operating systems, and network architectures; the NFS specifications independent of these media

Three Major Layers of NFS Architecture

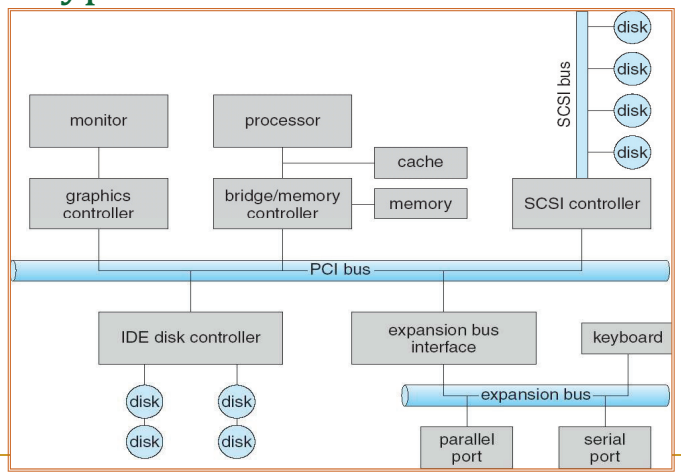
- UNIX file-system interface (based on the **open, read, write**, and **close** calls, and **file descriptors**)
- *Virtual File System* (VFS) layer – distinguishes local files from remote ones, and local files are further distinguished according to their file-system types
  - The VFS activates file-system-specific operations to handle local requests according to their file-system types
  - Calls the NFS protocol procedures for remote requests
- NFS service layer – bottom layer of the architecture
  - Implements the NFS protocol

I/O Systems

I/O Hardware

- Incredible variety of I/O devices
- Common concepts
  - **Port**
  - **Bus** (**daisy chain** or shared direct access)
  - **Controller** (**host adapter**)
- I/O instructions control devices
- Devices have addresses, used by
  - Direct I/O instructions
  - **Memory-mapped** I/O

A Typical PCI Bus Structure



Device I/O Port Locations on PCs

I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

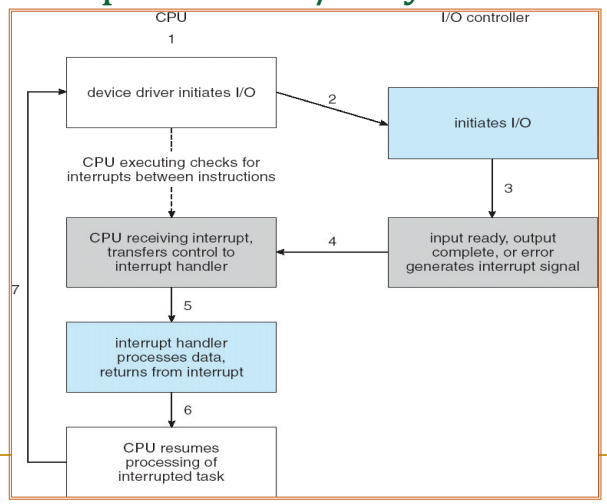
Polling

- Determines state of device
  - command-ready
  - busy
  - Error
- **Busy-wait** cycle to wait for I/O from device

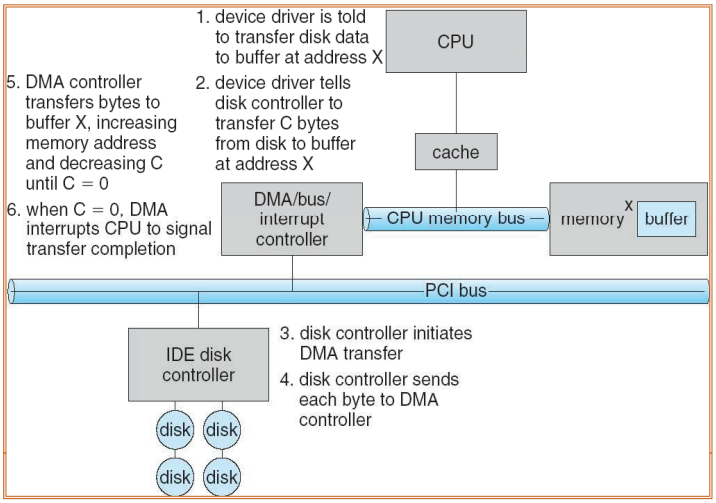
Interrupts

- CPU **Interrupt-request line** triggered by I/O device
- **Interrupt handler** receives interrupts
- **Maskable** to ignore or delay some interrupts
- Interrupt vector to dispatch interrupt to correct handler
  - Based on priority
  - Some **nonmaskable**
- Interrupt mechanism also used for exceptions

Interrupt-Driven I/O Cycle



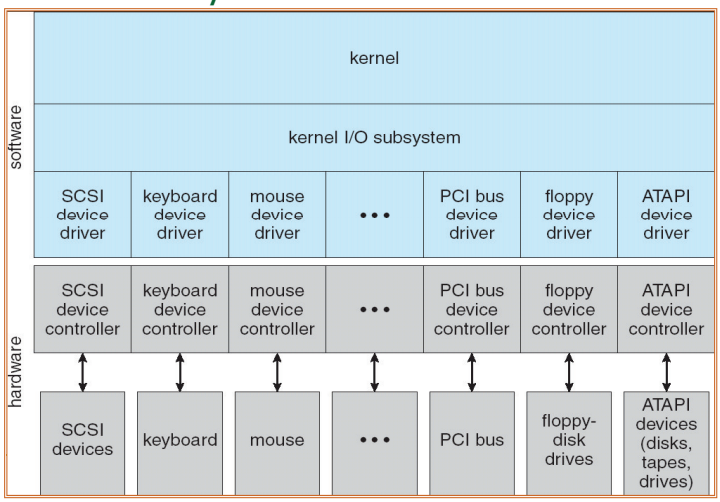
6 step process to perform DMA transfer



Application I/O Interface

- I/O system calls encapsulate device behaviors in generic classes
- Device-driver layer hides differences among I/O controllers from kernel
- Devices vary in many dimensions
  - **Character-stream** or **block**
  - **Sequential or random-access**
  - **Sharable or dedicated**
  - **Speed of operation**
  - **read-write, read only,** or **write only**

A Kernel I/O Structure



Characteristics of I/O Devices

aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only read–write	CD-ROM graphics controller disk

Block and Character Devices

- Block devices include disk drives
  - Commands include read, write, seek
  - Raw I/O or file-system access
  - Memory-mapped file access possible
- Character devices include keyboards, mice, serial ports
  - Commands include `get, put`
  - Libraries layered on top allow line editing

Network Devices

- Varying enough from block and character to have own interface
- Unix and Windows NT/9*x*/2000 include socket interface
  - Separates network protocol from network operation
  - Includes `select` functionality
- Approaches vary widely (pipes, FIFOs, streams, queues, mailboxes)

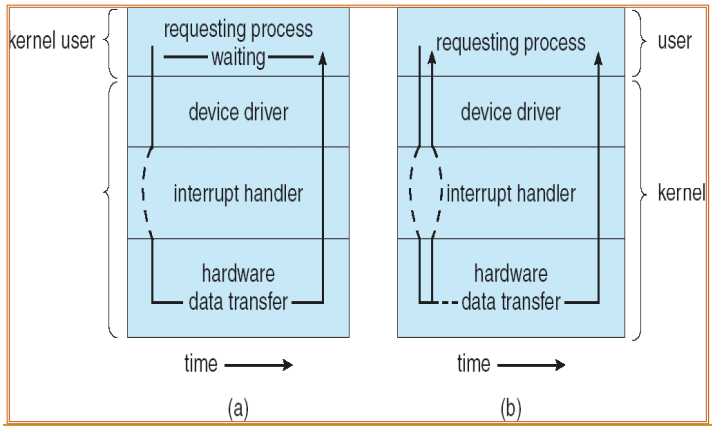
Clocks and Timers

- Provide current time, elapsed time, timer
- **Programmable interval timer** used for timings, periodic interrupts
- `ioctl` (on UNIX) covers odd aspects of I/O such as clocks and timers

Blocking and Nonblocking I/O

- **Blocking** - process suspended until I/O completed
  - Easy to use and understand
  - Insufficient for some needs
- **Nonblocking** - I/O call returns as much as available
  - User interface, data copy (buffered I/O)
  - Implemented via multi-threading
  - Returns quickly with count of bytes read or written
- **Asynchronous** - process runs while I/O executes
  - Difficult to use
  - I/O subsystem signals process when I/O completed

Two I/O Methods: (a) sync (b) async



Kernel I/O Subsystem

- Scheduling
  - Some I/O request ordering via per-device queue
  - Some OSs try fairness
- Buffering - store data in memory while transferring between devices
  - To cope with device speed mismatch
  - To cope with device transfer size mismatch
  - To maintain “copy semantics”

Kernel I/O Subsystem

- **Caching** - fast memory holding copy of data
  - Always just a copy
  - Key to performance
- **Spooling** - hold output for a device
  - If device can serve only one request at a time
  - i.e., Printing
- **Device reservation** - provides exclusive access to a device
  - System calls for allocation and deallocation
  - Watch out for deadlock

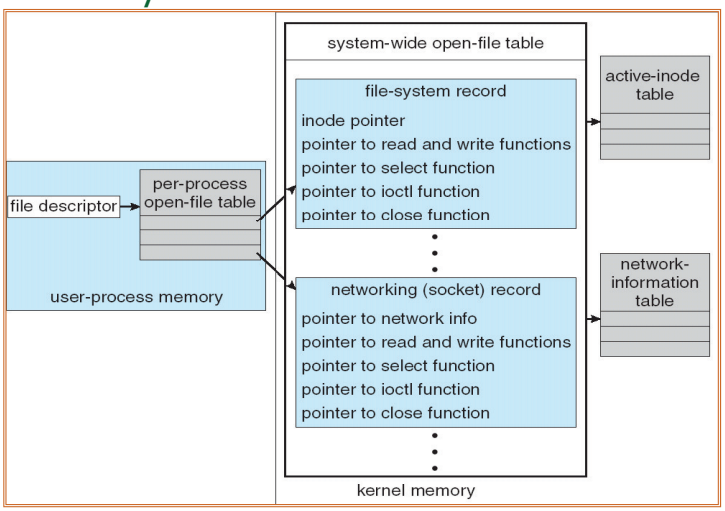
Error Handling

- OS can recover from disk read, device unavailable, transient write failures
- Most return an error number or code when I/O request fails
- System error logs hold problem reports

Kernel Data Structures

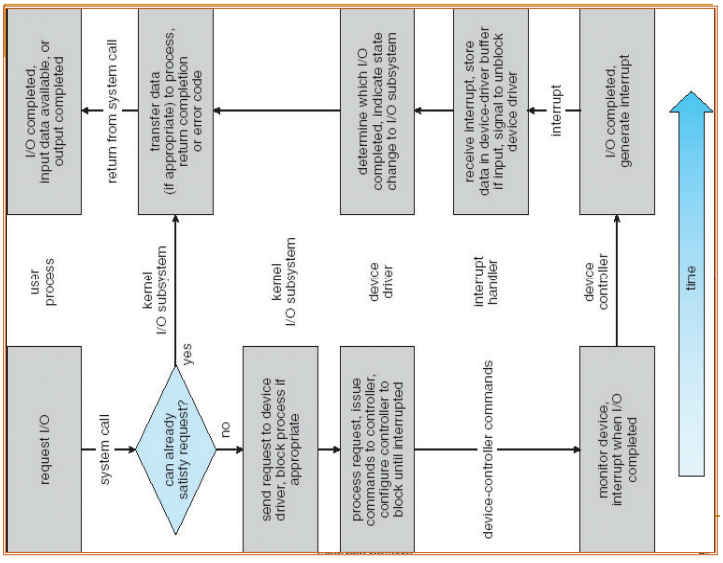
- Kernel keeps state info for I/O components, including open file tables, network connections, character device state
- Many, many complex data structures to track buffers, memory allocation, “dirty” blocks
- Some use object-oriented methods and message passing to implement I/O

UNIX I/O Kernel Structure



I/O Requests to H/W Operations

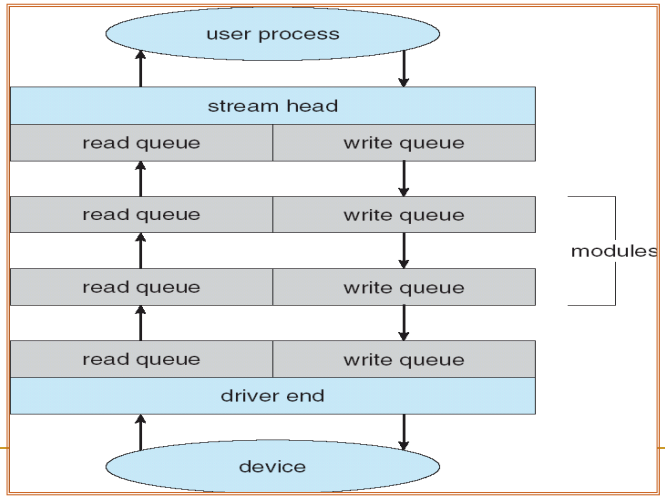
- Consider reading a file from disk for a process:
  - Determine device holding file
  - Translate name to device representation
  - Physically read data from disk into buffer
  - Make data available to requesting process
  - Return control to process



STREAMS

- **STREAM** – a full-duplex communication channel between a user-level process and a device in Unix System V and beyond
- A STREAM consists of:
  - STREAM head interfaces with the user process
  - driver end interfaces with the device
  - zero or more STREAM modules between them.
- Each module contains a **read queue** and a **write queue**
- Message passing is used to communicate between queues

The STREAMS Structure



Performance

- I/O a major factor in system performance:
  - Demands CPU to execute device driver, kernel I/O code
  - Context switches due to interrupts
  - Data copying
  - Network traffic especially stressful

Device-Functionality Progression