UNIT-I

Chapter-1 OVERVIEW OF OPERATING SYSTEM

1.1 What is an Operating System?

A program that acts as an intermediary between a user of a computer and the computer hardware
Operating system goals:

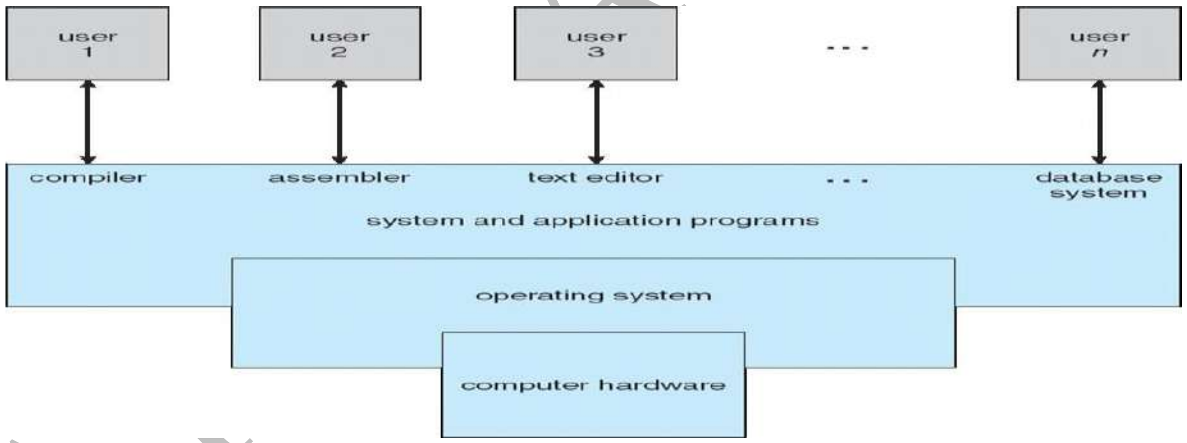
- Execute user programs and make solving user problems easier
- Make the computer system convenient to use
- Use the computer hardware in an efficient manner

Computer System Structure

Computer system can be divided into four components

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

Four Components of a Computer System



Operating System Definition

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

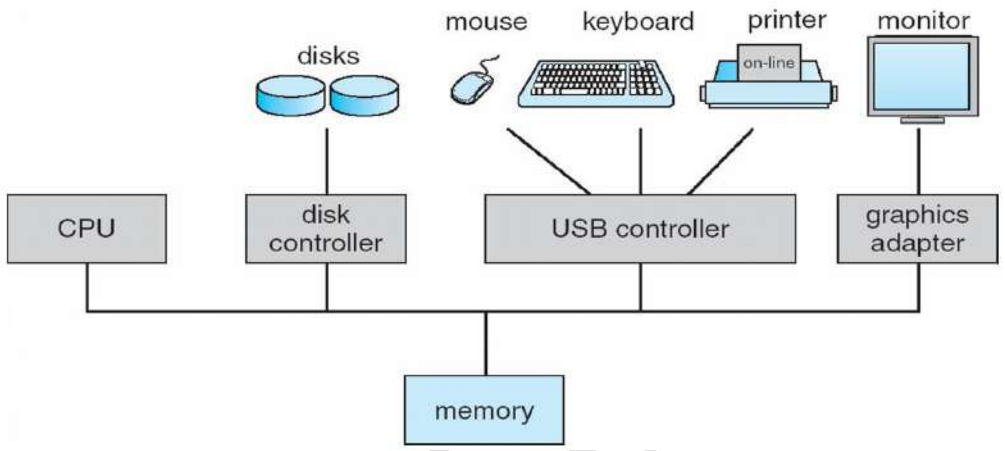
- "The one program running at all times on the computer" is the **kernel.** Everything else is either a system program (ships with the operating system) or an application program

Computer Startup

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

1.2 Computer System Organization

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



Computer-System Operation

- I/O devices and the CPU can execute concurrently
- Each device controller is in charge of a particular device type
- Each device controller has a local buffer
- CPU moves data from/to main memory to/from local buffers
- I/O is from the device to local buffer of controller
- Device controller informs CPU that it has finished its operation by causing An *interrupt*

Common Functions of Interrupts

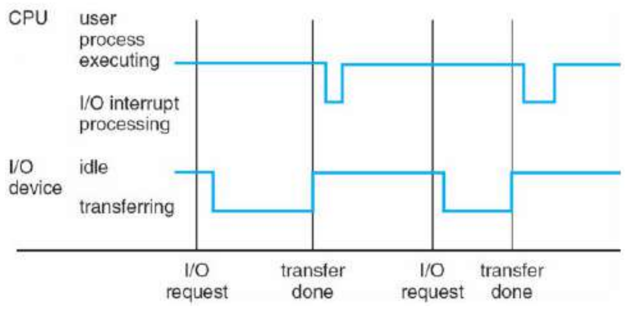
- Interrupt transfers control to the interrupt service routine generally, through the **interrupt vector**, which contains the addresses of all the service routines
- Interrupt architecture must save the address of the interrupted instruction
- Incoming interrupts are *disabled* while another interrupt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

Interrupt Handling

- The operating system preserves the state of the CPU by storing registers and the program counter

- Determines which type of interrupt has occurred using:
    - **polling**
    - **vectored** interrupt system
- Separate segments of code determine what action should be taken for each type of interrupt

Interrupt Timeline



I/O Structure

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

Direct Memory Access Structure

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

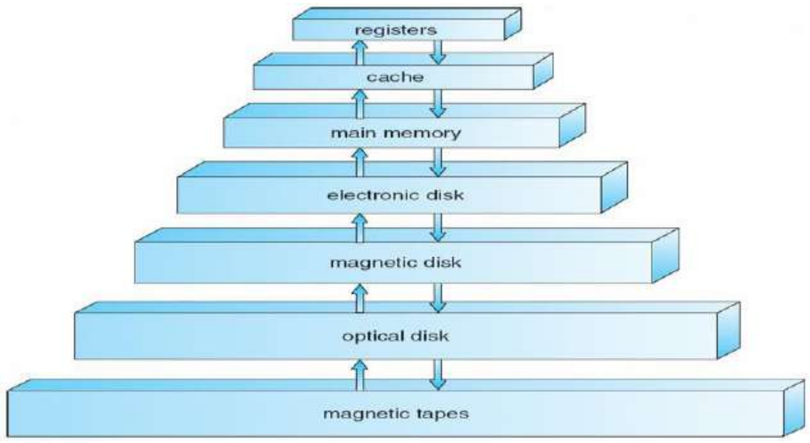
Storage Structure

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

Storage Hierarchy

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

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



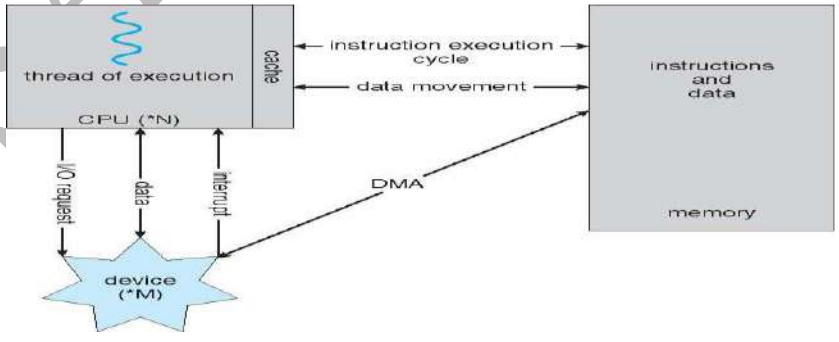
Storage Device Hierarchy

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

1.3 Computer-System Architecture

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



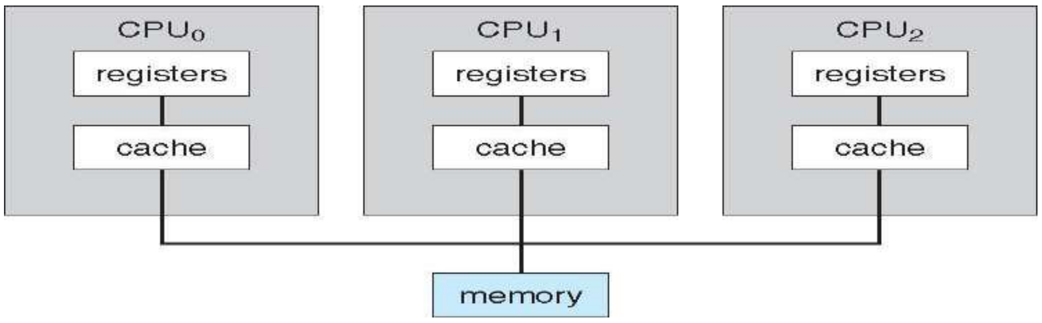


Fig: Symmetric Multiprocessing Architecture

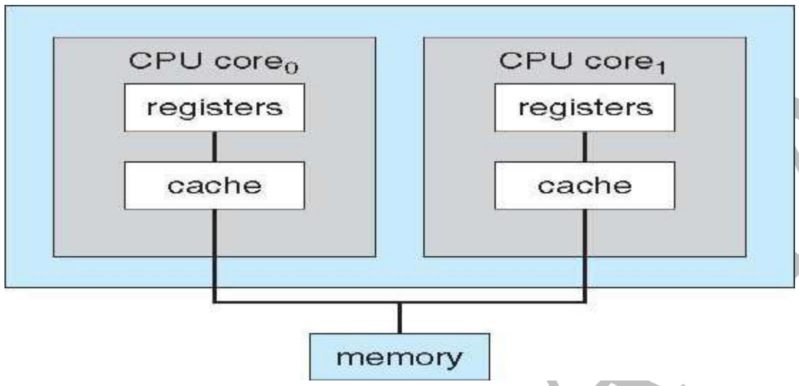
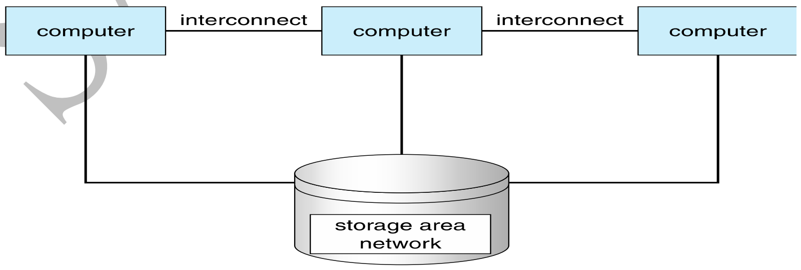


Fig: A Dual-Core Design with two cores placed on the same chip

Clustered Systems

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



1.4 Operating System Structure

- **Multiprogramming** needed for efficiency
    - Single user cannot keep CPU and I/O devices busy at all times
    - Multiprogramming organizes jobs (code and data) so CPU always has one to Execute
    - A subset of total jobs in system is kept in memory
    - One job selected and run via **job scheduling**
    - When it has to wait (for I/O for example), OS switches to another job
- **Timesharing (multitasking)** is logical extension in which CPU switches jobs so frequently that users can interact with each job while it is running, creating **interactive** computing
    - **Response time** should be < 1 second
    - Each user has at least one program executing in memory [**process]**
    - If several jobs ready to run at the same time [**CPU scheduling]**
    - If processes don't fit in memory, **swapping** moves them in and out to run
    - **Virtual memory** allows execution of processes not completely in memory

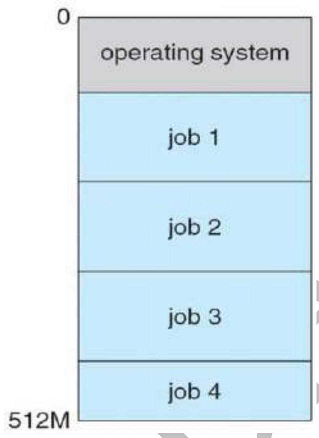


Fig: Memory Layout for Multiprogrammed System

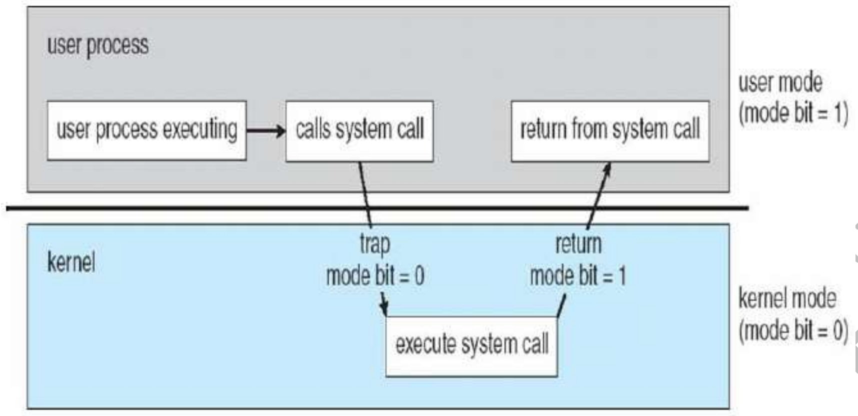
1.5 Operating-System Operations

- Interrupt driven by hardware and software
    - Hardware interrupt is caused by one of the devices
    - Software error or request creates **exception** or **trap** like a division by zero, request for operating system service
    - Other process problems include infinite loop, processes modifying each Other or the operating system
- **Dual-mode** operation allows OS to protect itself and other system components
    - **User mode** and **kernel mode**
    - **Mode bit** provided by hardware (mode bit 0 kernel mode; mode bit 1 user mode)
      Provides ability to distinguish when system is running user code or kernel code
      Some instructions designated as **privileged**, only executable in kernel mode
      System call changes mode to kernel, return from call resets it to user

Transition from User to Kernel Mode

- Timer to prevent infinite loop / process hogging resources
    - Sets interrupt after specific period
    - Keep a counter that is decremented by the physical clock.

- Operating system sets the counter (privileged instruction)
- When counter becomes zero it generates an interrupt
- Set up before scheduling process to regain control or terminate program that exceeds allotted time



OPERATING SYSTEM FUNCTIONS

1.6 Process Management

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

Process Management Activities

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

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

1.7 Memory Management

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

Memory management activities

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

1.8 Storage Management

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

- **OS activities include**
    - Creating and deleting files and directories
    - Primitives to manipulate files and dirs
    - Mapping files onto secondary storage
    - Backup files onto stable (non-volatile) storage media

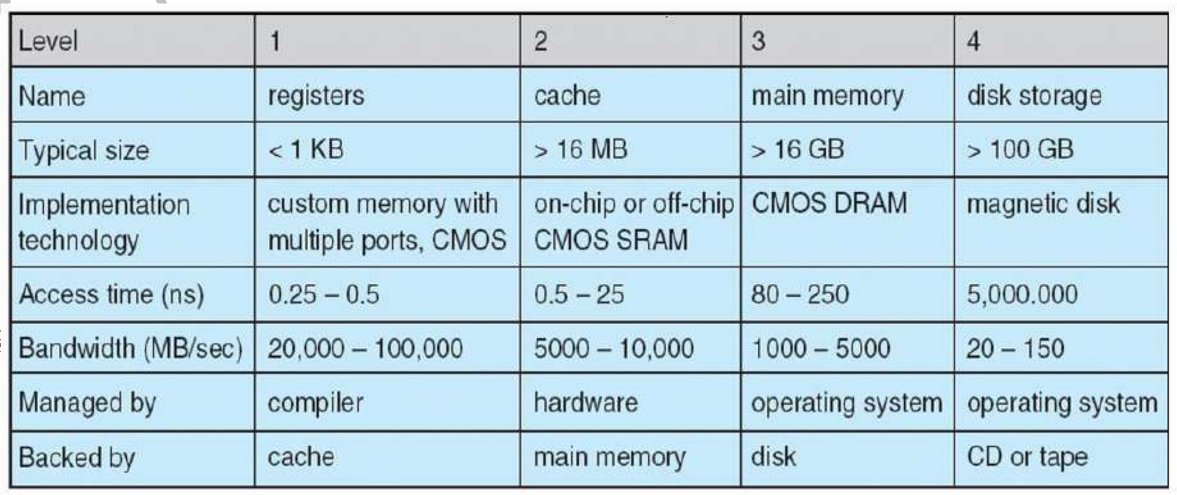
Mass-Storage Management

- Usually disks used to store data that does not fit in main memory or data that must be kept for a "long" period of time
- Proper management is of central importance
- Entire speed of computer operation hinges on disk subsystem and its algorithms

OS activities

- Free-space management
- Storage allocation
- Disk scheduling
- Some storage need not be fast
- Tertiary storage includes optical storage, magnetic tape
- Still must be managed
- Varies between WORM (write-once, read-many-times) and RW (read-write)

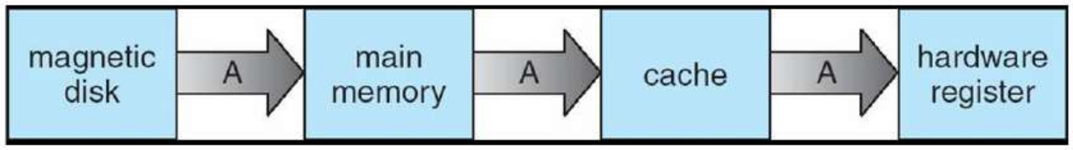
Performance of Various Levels of Storage



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

Mig

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



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

I/O Systems

- 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

1.9 Protection and Security

- **Protection** – any mechanism for controlling access of processes or users to resources defined by the OS
- **Security** – defense of the system against internal and external attacks
    - Huge range, including denial-of-service, worms, viruses, identity theft, theft of service
- Systems generally first distinguish among users, to determine who can do what
    - User identities (**user IDs**, security IDs) include name and associated number, one per user
    - User ID then associated with all files, processes of that user to determine access control
    - Group identifier (**group ID**) allows set of users to be defined and controls managed, then also associated with each process, file
    - **Privilege escalation** allows user to change to effective ID with more rights

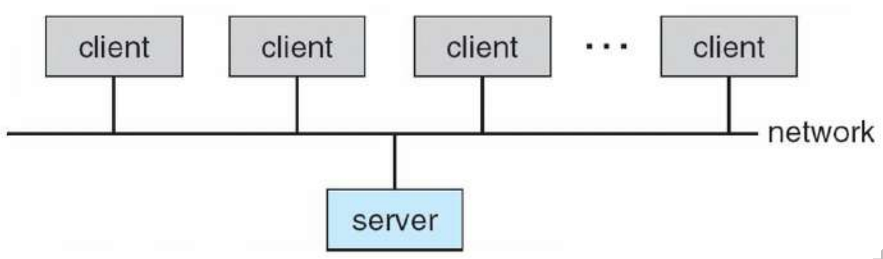
1.10 Computing Environments

Traditional computing

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

Client-Server Computing

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



Peer-to-Peer Computing

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

Web-Based Computing

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

1.11 Open-Source Operating Systems

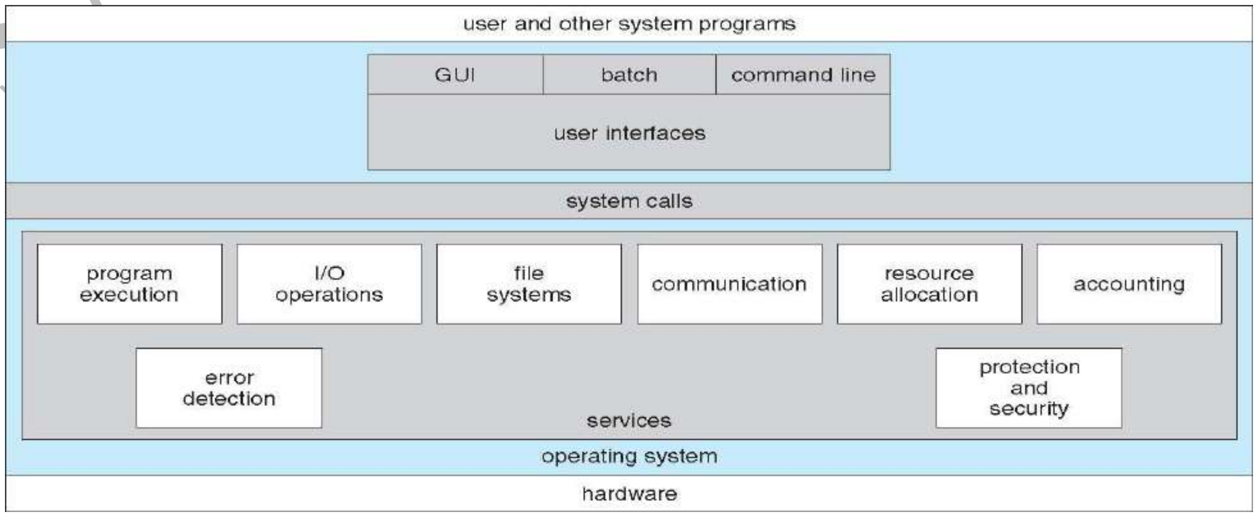
- Operating systems made available in source-code format rather than just binary closed-source
- Counter to the copy protection and Digital Rights Management (DRM) movement
- Started by Free Software Foundation (FSF), which has "copyleft" GNU Public License (GPL)
- Examples include GNU/Linux, BSD UNIX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

Chapter -2 SYSTEM STRUCTURES

2.1 Operating System Services

- Operating systems provide an environment for execution of programs and services to programs and users
- One set of operating-system services provides functions that are helpful to the user:
  - **User interface -** Almost all operating systems have a user interface (UI) and Varies between Command-Line (CLI), Graphics User Interface (GUI), Batch interface.
  - **Program execution -** The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
  - **I/O operations -** A running program may require I/O, which may involve a file or an I/O device
  - **File-system manipulation -** The file system is of particular interest. Obviously, programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.
  - **Communications** – Processes may exchange information, on the same computer or between computers over a network. Communications may be via shared memory or through message passing (packets moved by the OS)
  - **Error detection** – OS needs to be constantly aware of possible errors.
    - May occur in the CPU and memory hardware, in I/O devices, in user program
    - For each type of error, OS should take the appropriate action to ensure correct and consistent computing
    - Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system
- Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing
  - **Resource allocation** - When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
    - Many different types of resources are managed by OS - CPU cycles, main memory, file storage, I/O devices.
  - **Accounting** - To keep track of which users use how much and what kinds of computer resources
  - **Protection and security** - The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
    - Protection involves ensuring that all access to system resources is controlled
    - Security of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts

A View of Operating System Services



2.2 User Operating System Interface

Command interpreter- CLI

- Command Line Interface (CLI) or command interpreter allows direct command entry
  - Sometimes implemented in kernel, sometimes by systems program
  - Sometimes multiple flavors implemented – shells
  - Primarily fetches a command from user and executes it
  - Sometimes commands built-in, sometimes just names of programs. If the latter, adding new features doesn't require shell modification

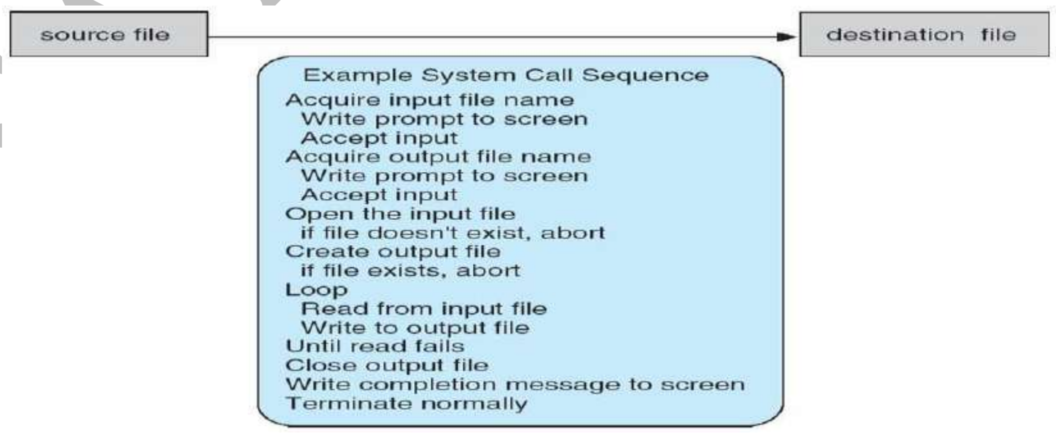
Graphical User Interface - GUI

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

2.3 System Calls

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

Example of how System Calls are used: System call sequence to copy the contents of one file to another file



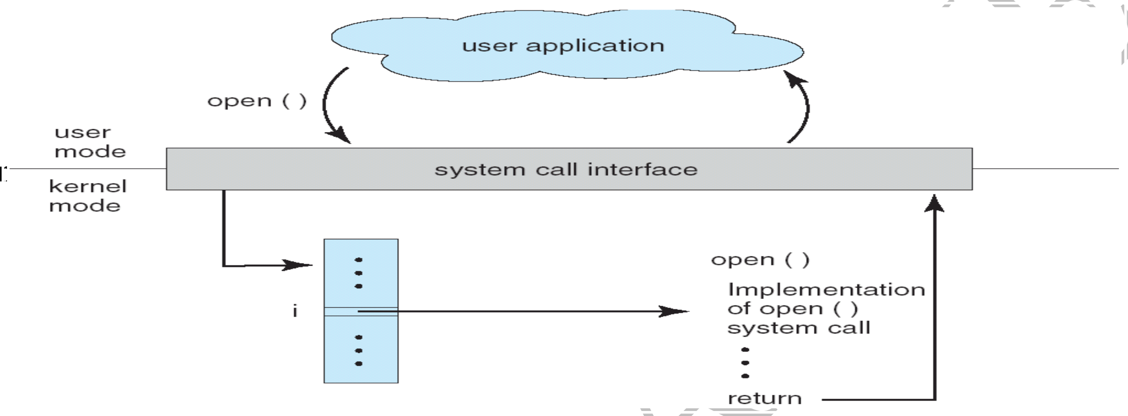
```
Example System Call Sequence
Acquire input file name
  Write prompt to screen
  Accept input
Acquire output file name
  Write prompt to screen
  Accept input
Open the input file
  if file doesn't exist, abort
Create output file
  if file exists, abort
Loop
  Read from input file
  Write to output file
Until read fails
Close output file
Write completion message to screen
Terminate normally
```

System Call Implementation

- Typically, a number associated with each system call. System-call interface maintains a table indexed according to these Numbers
- The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
    - Just needs to obey API and understand what OS will do as a result call
    - Most details of OS interface hidden from programmer by API and Managed by run-time support library (set of functions built into libraries included with compiler)

API – System Call – OS Relationship

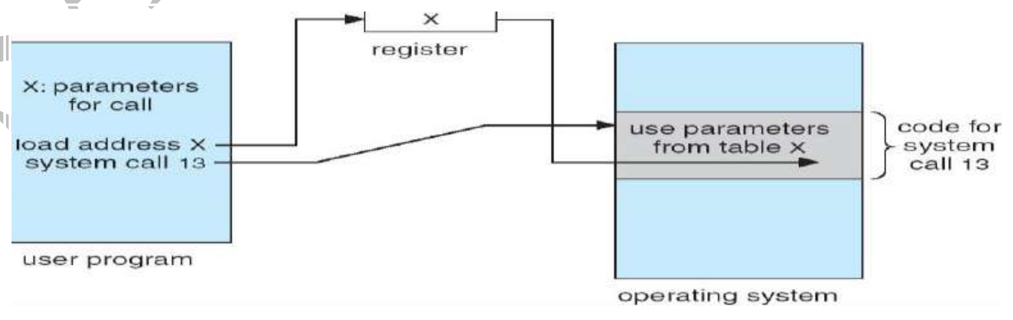
Star



System Call Parameter Passing

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

Parameter Passing via Table



2.4 Types of System Calls

Process control

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

File management

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

Device management

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

Information maintenance

- get time or date, set time or date
- get system data, set system data
- get and set process, file, or device attributes

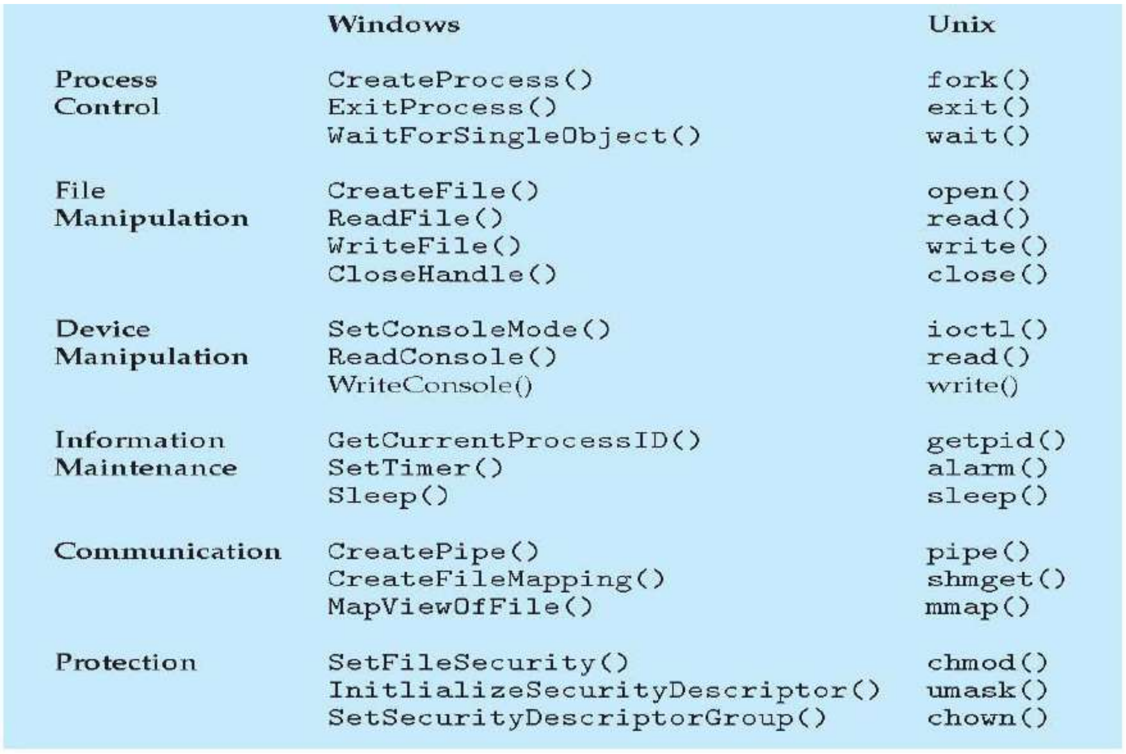
Communications

- create, delete communication connection
- send, receive messages
- transfer status information
- attach and detach remote devices

Protection

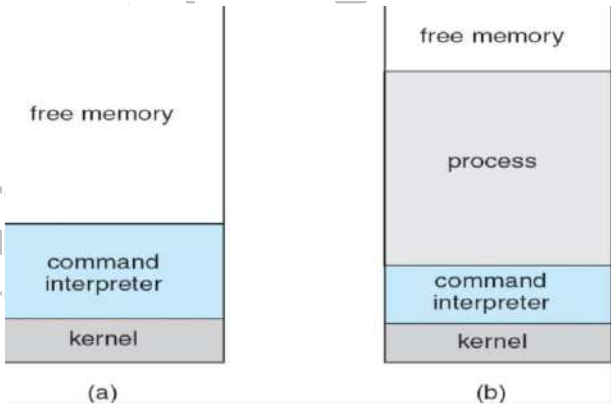
- Control access to resources
- Get and set permissions
- Allow and deny user access

Examples of Windows and Unix System Calls



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

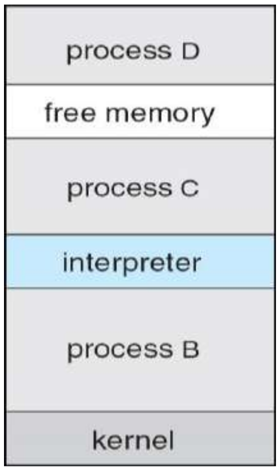
Process Control
There are so many facets of and variations in process and job control that we next use two examples-one involving a single-tasking system and the other a multitasking system -to clarify these concepts. The MS-DOS operating system is an example of a single-tasking system. It has a command interpreter that is invoked when the computer is started (Figure 2.10(a)). Because MS-DOS is single-tasking, it uses a simple method to run a program and does not create a new process. It loads the program into memory, writing over most of itself to give the program as much memory as possible (Figure 2.10(b)). Next, it sets the instruction pointer to the first instruction of the program. The program then runs, and either an error cause a trap, or the program executes a system call to terminate. In either case, the error code is saved in the system memory for later use.
MS-DOS execution



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

FreeBSD (derived from Berkely UNIX) is an example of a multitasking system. When a user logs on to the system, the shell of the user's choice is run. This shell is similar to the MS-DOS shell in that it accepts commands and executes programs that the user requests. However, since FreeBSD is a multitasking system, the command interpreter may continue running while another program is executed (Figure 2.11). To start a new process, the shell executes a fork() system call. Then, the selected program is loaded into memory via an exec() system call, and the program is executed. Depending on the way the command was issued, the shell then either waits for the process to finish or runs the process "in the background."

FreeBSD Running Multiple Programs



2.5 System Programs

System Programs are available just above Operating System and below Application Programs. System programs provide a convenient environment for program development and execution. They can be divided into:

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

Most users' view of the operating system is defined by application and system programs, not the actual system calls

– Provide a convenient environment for program development and execution
– Some of them are simply user interfaces to system calls; others are considerably more complex

- **File management -** Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories
- **Status information** –Some programs simply ask the system for info - date, time, amount of available memory, disk space, number of users. Others provide detailed performance, logging, and debugging information. Typically, these programs format and print the output to the terminal or other output devices. Some systems implement a registry - used to store and retrieve configuration information
- **File modification-**Text editors available to create and modify files stored on disks or other storage devices. there are Special commands to search contents of files or perform transformations of the text.
- **Programming-language support** - Compilers, assemblers, debuggers and interpreters provided by OS

- **Program loading and execution-** Absolute loaders, relocatable loaders, linkage editors, and overlay-loaders, debugging systems for higher-level and machine languages
- **Communications -** Provide the mechanism for creating virtual connections among processes, users, and computer systems. They 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.

2.6 Operating System Design and Implementation

- Design and Implementation of OS not "solvable", but some approaches have proven successful
- Internal structure of different Operating Systems can vary widely
- Start by defining goals and specifications
- Affected by choice of hardware, type of system

Policies and Mechanisms

- Requirements divided into 2 basic groups: 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
- One Important principle in the design is to separate policy from mechanism
  - **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
- Specifying and designing an OS is highly creative task of **software engineering**

Implementation

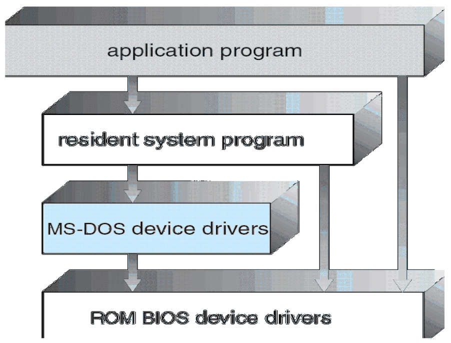
- Much variation
  - Early OSes in assembly language
  - Then system programming languages like Algol, PL/1
  - Now C, C++
- Actually usually a mix of languages
  - Lowest levels in assembly
  - Main body in C
  - Systems programs in C, C++, scripting languages like PERL, Python, shell scripts
- More high-level language easier to **port** to other hardware but slower
- **Emulation** can allow an OS to run on non-native hardware

2.7 Operating – System structure

Simple Structure

- Many commercial systems do not have well-defined structures.
- MS-DOS is an example written to provide the most functionality in the least space
- Not divided into modules
- Although MS-DOS has some structure, its interfaces and levels of Functionality are not well separated

MS-DOS Layer Structure

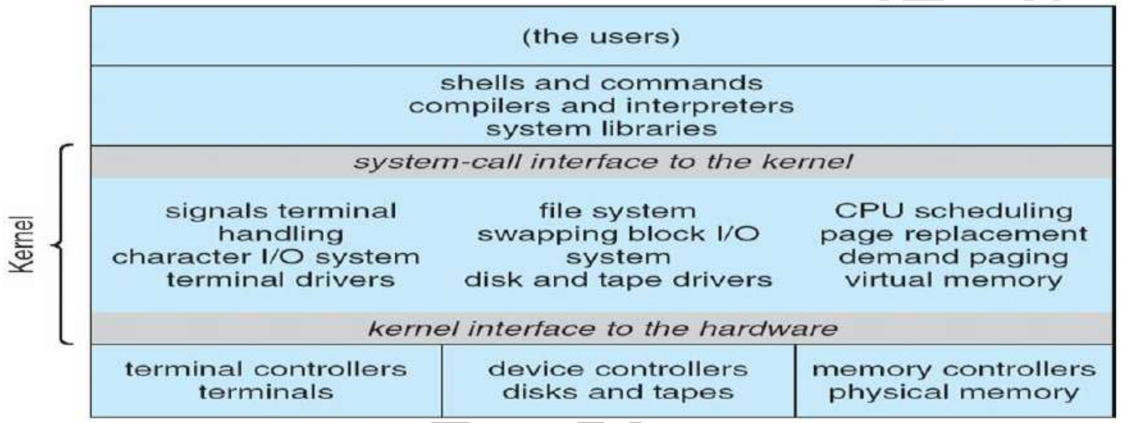


Another example of limited structuring is the original UNIX operating system. Like MS-DOS, UNIX initially was limited· by hardware functionality. It consists of two separable parts: the kernel and the system prograrns. The kernel is further separated into a series of interfaces and device drivers, which have been added and expanded over the years as UNIX has evolved. We can view the traditional UNIX operating system as being layered, as shown in Figure 2.13. Everything below the system-call interface and above the physical hardware is the kernel. The kernel provides the file system, CPU scheduling, memory management, and other operating-system functions through system calls.

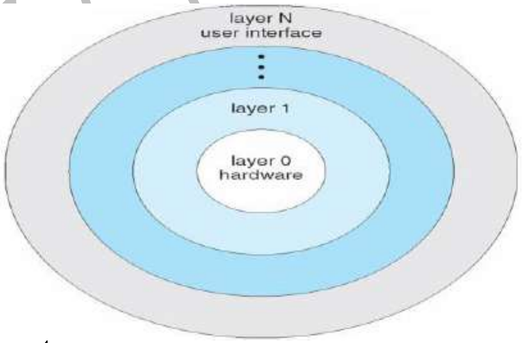
Layered Approach

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

Traditional UNIX System Structure



Layered Operating System



Micro kernel System Structure
Researchers developed an operating system called Mach that modularized the kernel using the microkernel approach. This method structures the operating system by removing all nonessential components from the kernel and implementing them as system and user-level programs. the result is a smaller kernel. There is little consensus regarding which services should remain in the kernel and which should be implemented in user space. Typically, however, micro kernels provide minimal process and memory management, in addition to a communication facility.

- The main function of the micro kernel is to provide Communication facility between user modules using message passing.

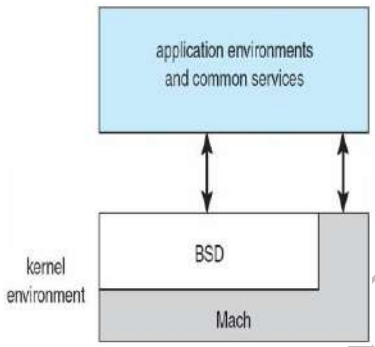
Benefits:

- Easier to extend a microkernel
- Easier to port the operating system to new architectures
- More reliable (less code is running in kernel mode)
- More secure

Detriments:

- Performance decreases due to increased system function overhead of communication.

Mach OS X Structure



Modules

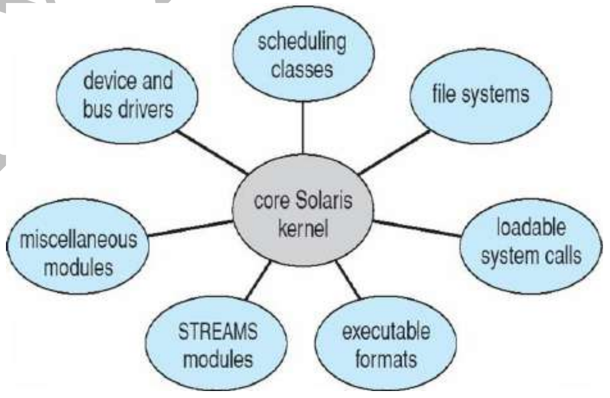
Most modern operating systems implement kernel modules

- Uses object-oriented approach
- Each core component is separate
- Each talks to the others over known interfaces
- Each is loadable as needed within the kernel

Overall, similar to layers but with more flexible Linux, Solaris etc

Solaris Modular Approach

The Solaris Operating system structure is organized around a core kernel with seven types of loadable kernel modules.



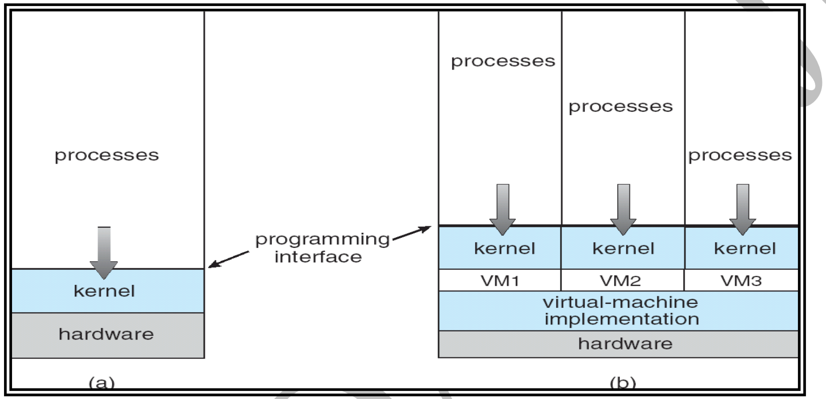
2.8 Virtual Machines

- A virtual machine takes the layered approach to its logical conclusion. It treats hardware and the operating system kernel as though they are all hardware
- A virtual machine provides an interface *identical* to the underlying bare hardware
- The operating system host creates the illusion that a process has its own processor and (virtual memory)

- Each guest provided with a (virtual) copy of underlying computer

Virtual Machines History and Benefits

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

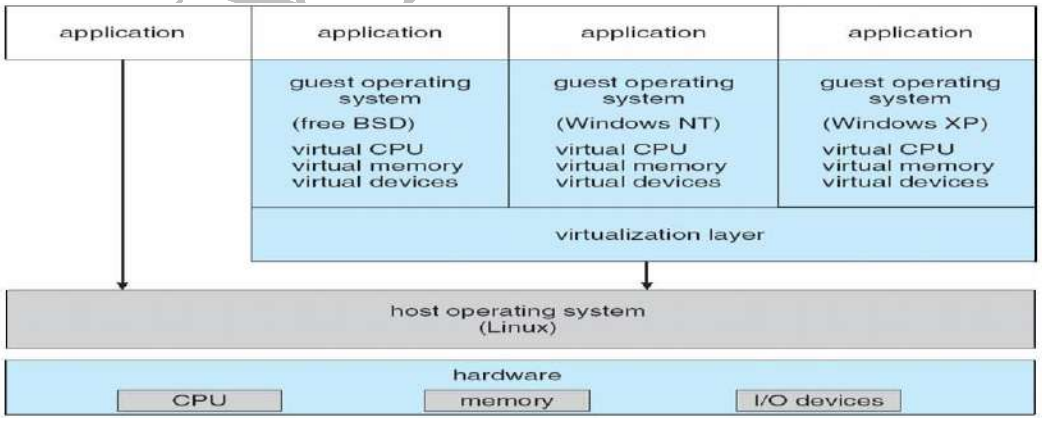


(a) Non virtual machine (b)virtual machine

Examples:

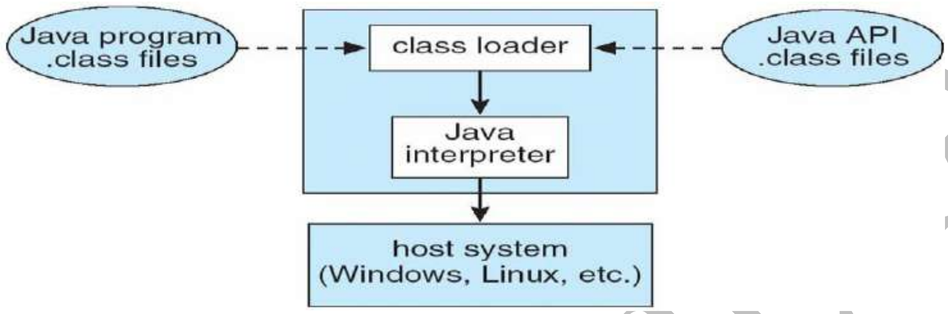
VMware Architecture

VMware is a popular commercial application that runs on a host operating system such as Windows or linux and allows this host system to concurrently run several different guest operating systems as independent virtual machines.



The Java Virtual Machine

The JVM is a specification for an abstract computer. It consists of a class loader and a Java interpreter that executes the architecture-neutral bytecodes, as diagrammed in Figure 2.20. The class loader loads the compiled . class files from both the Java program and the Java API for execution by the Java interpreter. After a class is loaded, the verifier checks that the . class file is valid Java bytecode and does not overflow or underflow the stack It also ensures that the bytecode does not perform pointer arithmetic, which could provide illegal memory access. If the class passes verification, it is run by the Java interpreter. The JVM also automatically manages memory by performing garbage collection -the practice of reclaiming memory from objects no longer in use and returning it to the system.



2.9 Operating System Generation

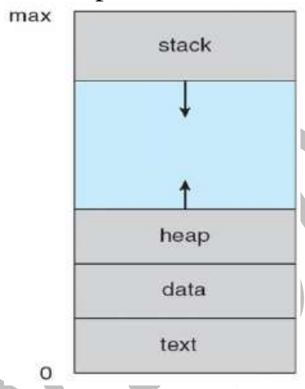
- Operating systems are designed to run on any class of machines; the system must then be configured for each specific computer site
- to generate a system we use a special program called SYSGEN (system generation) program which obtains information concerning the specific configuration of the hardware system or probes the hardware directly to determine what components are there.
- The following kinds of information must be determined:
    - what CPU is to be used?
    - how much memory is available?
    - what devices are available?
    - what operating system options are desired?

2.10 System Boot

- When power initialized on system, execution starts at a fixed memory location and Firmware is used to hold initial boot code
- Operating system must be made available to hardware so that hardware can start it
    - Small piece of code – **bootstrap loader**, stored in **ROM** or **EEPROM** locates the kernel, loads it into memory, and starts it
    - Sometimes two-step process where **boot block** at fixed location is loaded by ROM code, which loads bootstrap loader from disk
- Common bootstrap loader, **GRUB**, allows selection of kernel from multiple disks, versions and kernel options
- Kernel loads and system starts **running**

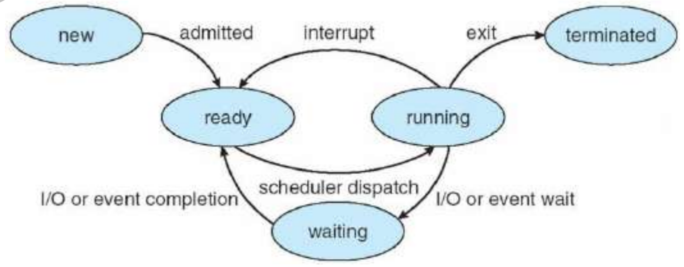
Chapter-3 PROCESS CONCEPT

- An operating system executes a variety of programs:
  - Batch system – **jobs**
  - Time-shared systems – **user programs** or **tasks**
- Textbook uses the terms ***job*** and ***process*** almost interchangeably
- **Process** – a program in execution; process execution must progress in sequential fashion
- Multiple parts
  - The program code, also called **text section**
  - Current activity including **program counter**, processor registers
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time
- Program is ***passive*** entity stored on disk (**executable file**), process is ***active***
  - Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc
- One program can be several processes
  - Consider multiple users executing the same program



Process State

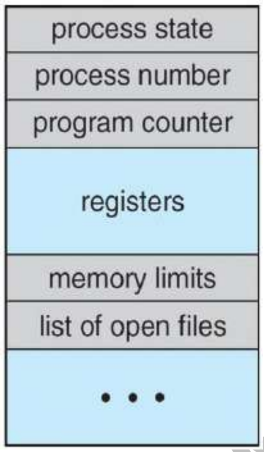
- As a process executes, it changes **state**
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process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



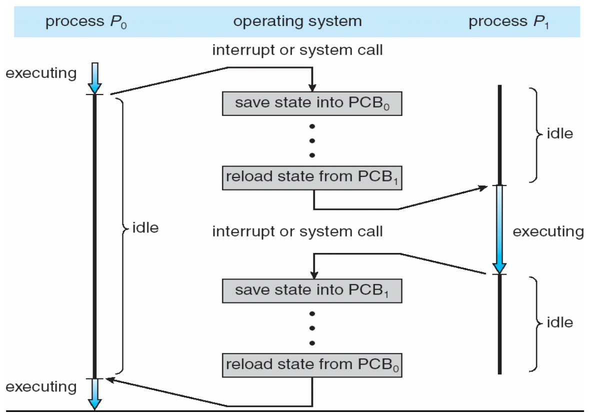
Process Control Block (PCB)

Information associated with each process (also called **task control block**)

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



CPU Switch from Process to Process

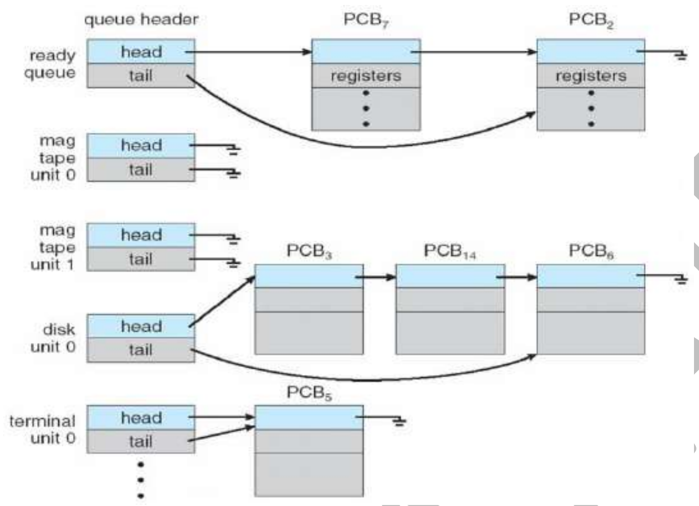


Threads

- A process is a program that performs a single thread of execution.
- This single thread of control allows the process to perform only one task at a time.
- If multiple tasks are allowed to execute simultaneously then we have multiple threads of execution.

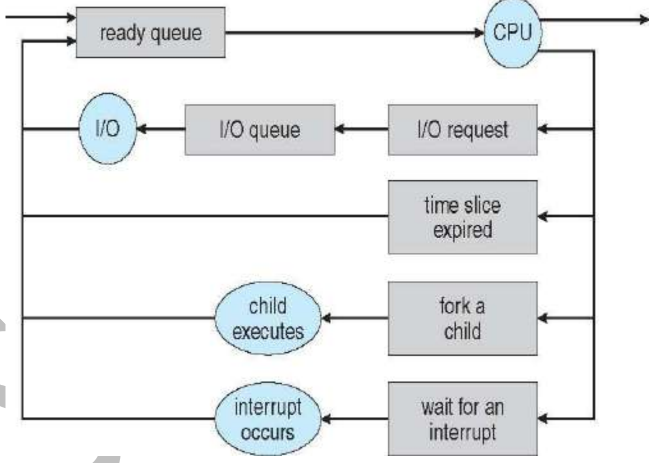
3.2 Process Scheduling

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



Ready Queue And Various I/O Device Queues

Representation of Process Scheduling

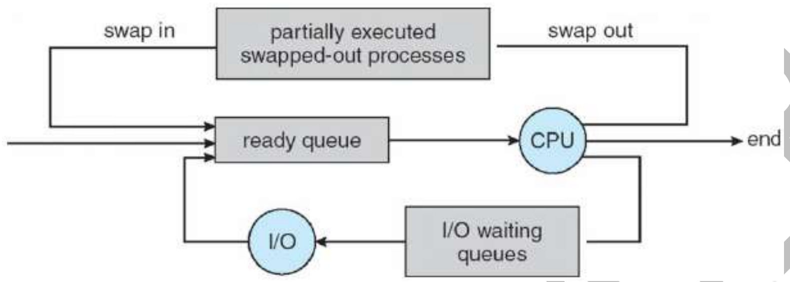


Schedulers

- Short-term scheduler (or CPU scheduler) – selects which process should be executed next from ready queue and allocates CPU
    - Sometimes the only scheduler in a system
    - Short-term scheduler is invoked frequently (milliseconds) ⇒ (must be fast)
- Long-term scheduler (or job scheduler) – selects which processes should be brought into the ready queue

- Long-term scheduler is invoked infrequently (seconds, minutes) ⇒ (may be slow)
- The long-term scheduler controls the degree of multiprogramming

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



Addition of Medium Term Scheduling

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

Context Switch

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

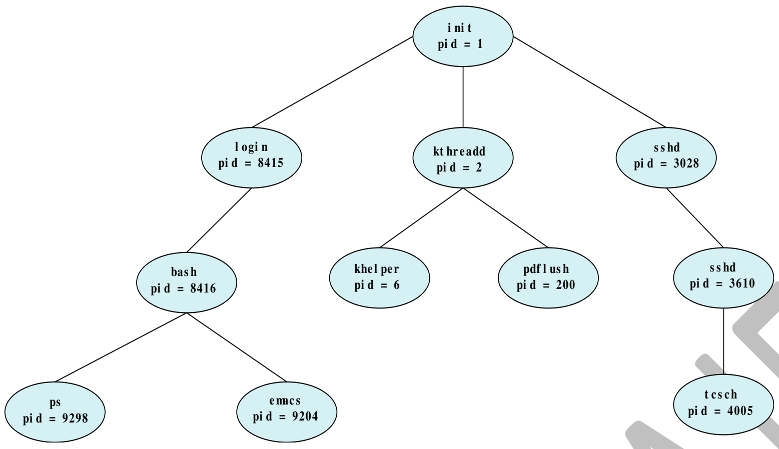
3.3 Operations on Processes

- System must provide mechanisms for:
  - process creation,
  - process termination,

Process creation

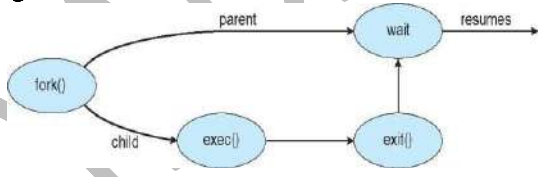
- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier** (**pid**)
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources

    - Parent and child share no resources
- Execution options
    - Parent and children execute concurrently
    - Parent waits until children terminate



A Tree of Processes in Linux

- There are 2 possibilities in terms of the address space
    - Child process is duplicate of parent process(has same data as parent)
    - Child has a new program loaded into it
- UNIX examples
    - **fork()** system call creates new process
    - **exec()** system call used after a **fork()** to replace the process' memory space with a new program



Process Creation

Process Termination

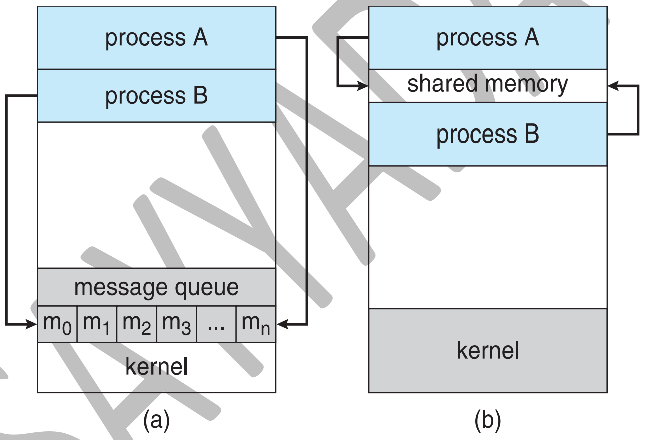
- Process executes last statement and then asks the operating system to delete it using the **exit()** system call.
    - Returns status data from child to parent (via **wait()**)
    - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using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

- Some operating systems do not allow child to exists if its parent has terminated. If a process terminates, then all its children must also be terminated.
    - **cascading termination.** All the child processes are terminated.
    - The termination is initiated by the operating system.
- The parent process may wait for termination of a child process by using the **wait()** system call**.** The call returns status information and the pid of the terminated process

  pid = wait(&status);
- If parent terminated without invoking **wait** , process is an **orphan**

3.4 Interprocess Communication

- Processes within a system may be ***independent*** or ***cooperating***
- ***Independent*** process cannot affect or be affected by the execution of another process
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
    - Information sharing
    - Computation speedup
    - Modularity
    - Convenience
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC are
    - **Shared memory**
    - **Message passing**



(a) Message Passing (b) Shared Memory

Shared Memory Systems

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issue is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
- Synchronization is discussed in great details in Chapter 5.

Producer-Consumer Problem

- Producer-consumer problem is a common Paradigm for cooperating processes.
    - Example: server is the producer and client is the consumer in client-server architecture.
    - Example: web server produces files which are consumed by the client browser.
- *Producer-consumer problem uses shared memory and runs concurrently using a buffer.*
- *Producer* process produces information that is consumed by a *consumer* process
    - **unbounded-buffer** places no practical limit on the size of the buffer
    - **bounded-buffer** assumes that there is a fixed buffer size
    - The producer and consumer must be synchronized, so that consumer does not try to consume an item that has not yet been produced.

Message-Passing Systems

- 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*)
    - **receive**(*message*)
- The *message* size is either fixed or variable
- If processes *P* and *Q* wish to communicate, they need to:
    - Establish a ***communication link*** between them
    - Exchange messages via send/receive
- Implementation issues:
    - How are links established?
    - Can a link be associated with more than two processes?
    - How many links can there be between every pair of communicating processes?
    - What is the capacity of a link?
    - Is the size of a message that the link can accommodate fixed or variable?
    - Is a link unidirectional or bi-directional?
- Implementation of communication link
    - Physical:
        - Hardware bus
        - Network
    - Logical:
        - Direct or indirect
        - Synchronous or asynchronous
        - Automatic or explicit buffering

Direct communication

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

- The link may be unidirectional, but is usually bi-directional

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional
- Operations
  - create a new mailbox (port)
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:
  send(*A, message*) – send a message to mailbox A
  receive(*A, message*) – receive a message from mailbox A
- 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** -- the sender is blocked until the message is received
  - **Blocking receive** -- the receiver is blocked until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** -- the sender sends the message and continue
  - **Non-blocking receive** -- the receiver receives:
    - A valid message, or
    - Null message
- Different combinations possible
  - If both send and receive are blocking, we have a **rendezvous**

Buffering

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

1. Zero capacity – no messages are queued on a link.
   Sender must wait for receiver (rendezvous)
2. Bounded capacity – finite length of *n* messages
   Sender must wait if link full

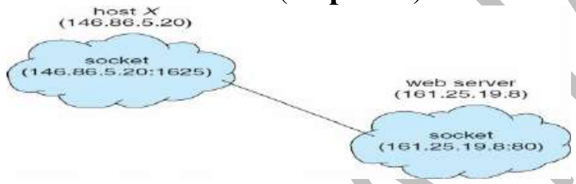
3. Unbounded capacity – infinite length
   Sender never waits

3.6 Communications in Client-Server Systems

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

Sockets

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



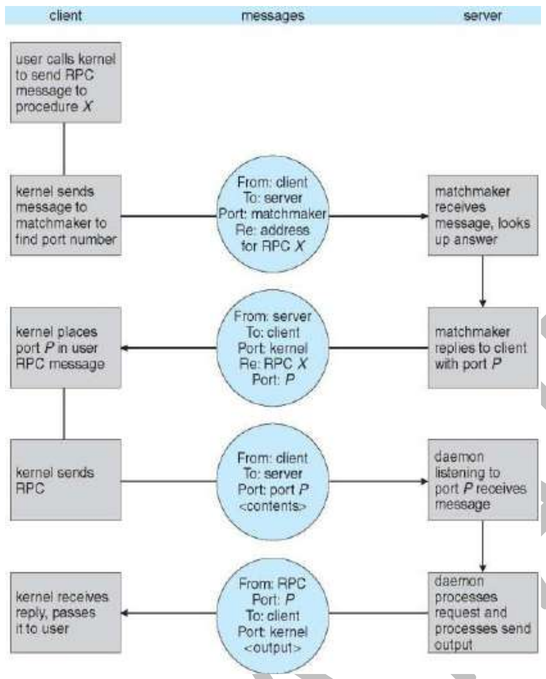
Three types of sockets

- Connection-oriented (TCP)
- Connectionless (UDP)
- MulticastSocket class– data can be sent to multiple recipients

Remote Procedure Calls

- Remote procedure call (RPC)is designed as a way to abstract procedure calls between processes on networked systems.
- Uses ports for service differentiation
- RPC allows a client to invoke a procedure on a remote host as it would invoke a procedure locally.
- The RPC system hides the details that allow communication to take place by providing a stub on the client side.
- **Stubs** – client-side proxy for the actual procedure on the server
- The client-side stub locates the port on the server and **marshalls** the parameters
- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server
- Data representation differences on client and server is handled via External Data Representation (XDR) format to account for different architectures
  - o Big-endian (use high memory address to store the most significant byte)
  - o Little-endian (store least significant byte in the high memory address)
- Remote communication has more failure scenarios than local procedure calls due to network errors.

- One way to address this problem is to ensure that Messages can be delivered *exactly once* rather than *at most once*
- OS typically provides a rendezvous (or matchmaker) service requesting the port address of the RPC to connect client and server



client
messages
server
user calls kernel to send RPC message to procedure X
kernel sends message to matchmaker to find port number
From: client To: server Port: matchmaker Re: address for RPC X
matchmaker receives message, looks up answer
kernel places port P in user RPC message
From: server To: client Port: kernel Re: RPC X Port: P
matchmaker replies to client with port P
kernel sends RPC
From: client To: server Port: port P <contents>
daemon listening to port P receives message
kernel receives reply, passes it to user
From: RPC Port: P To: client Port: kernel <output>
daemon processes request and processes send output

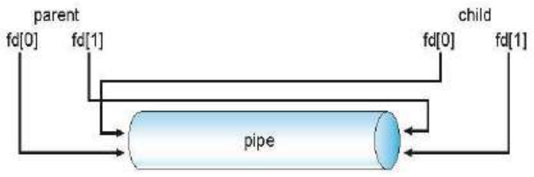
Execution of RPC

Pipes

- Acts as a channel allowing two processes to communicate
- Issues:
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - There must exist a relationship (i.e., ***parent-child***) between the communicating processes?
  - Can the pipes be used over a network?
- Ordinary pipes – cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it created.
- Named pipes – can be accessed without a parent-child relationship.

Ordinary Pipes

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

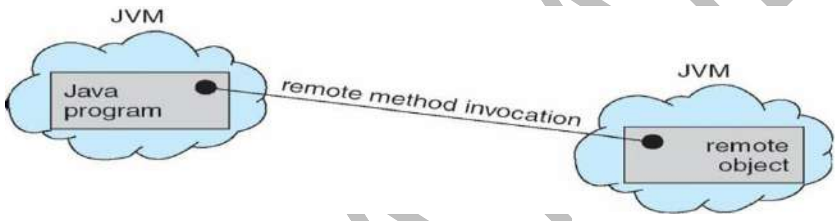


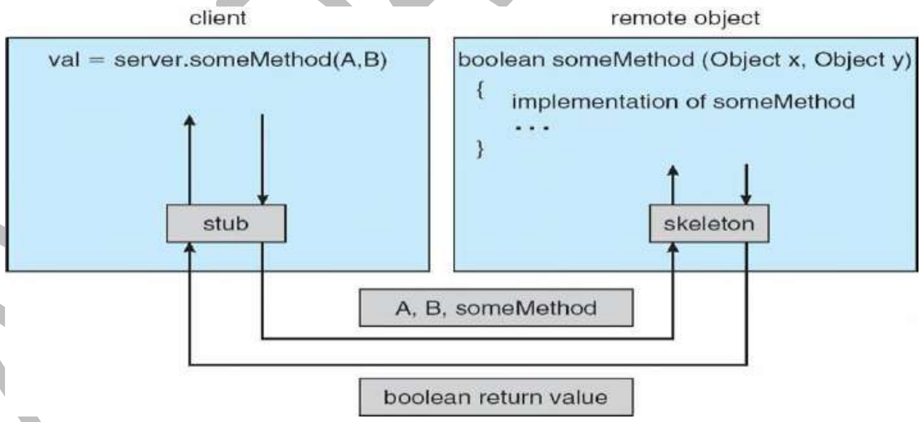
Named Pipes

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

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

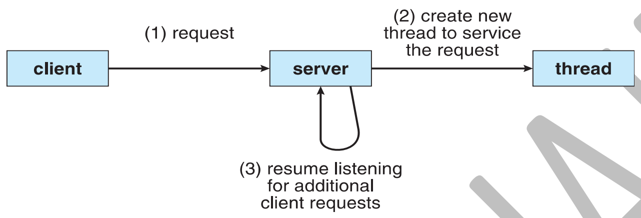


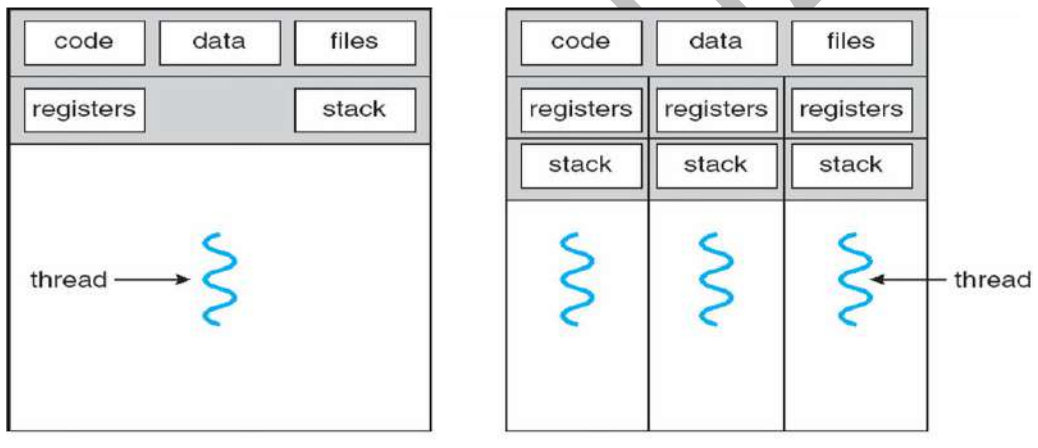

client
val = server.someMethod(A,B)
stub
remote object
boolean someMethod (Object x, Object y)
{
implementation of someMethod
. . .
}
skeleton
A, B, someMethod
boolean return value

Marshalling Parameters

Chapter-4 MULTITHREADED PROGRAMMING

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


(1) request
(2) create new thread to service the request
client
server
thread
(3) resume listening for additional client requests



single-threaded process multithreaded process

Benefits

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

4.2 Multithreading Models

- **User threads** - management done by user-level threads library
- Three primary thread libraries:

- POSIX **Pthreads**
- Windows threads
- Java threads

- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general purpose operating systems, including:
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

Relationship between user threads and kernel threads can be established using any of the below models:

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

Many-to-one Model

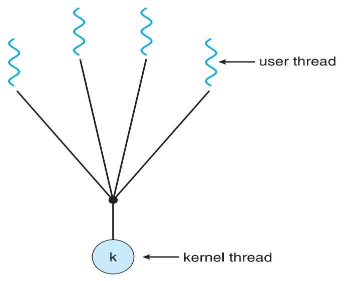
- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable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

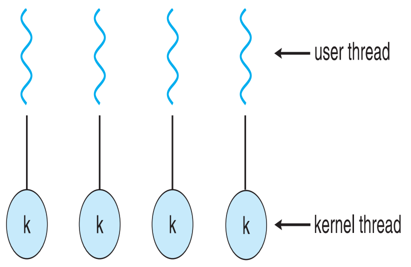
One-to-one Model

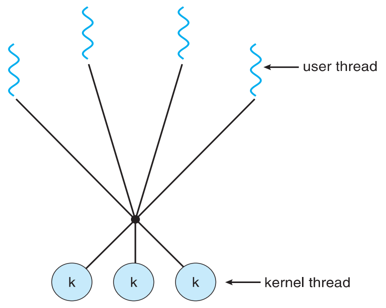
- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later

Many-to-Many Model

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

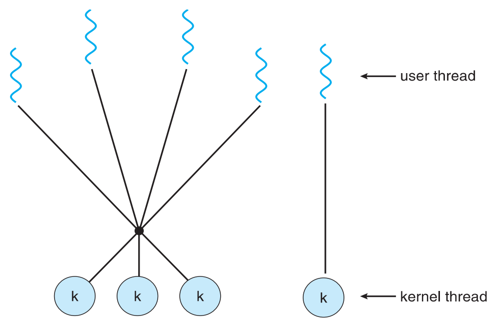






Two-Level Model

- Similar to M:M, except that it allows a user thread to be bound to kernel thread
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier



4.3 Thread Libraries

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

Pthreads

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

Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by:

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

  - Extending Thread class
  - Implementing the Runnable interface

4.4 Threading Issues

- Semantics of fork() and exec() system calls
- Signal handling
  - Synchronous and asynchronous

- Thread cancellation of target thread
    - Asynchronous or deferred
- Thread-local storage
- Scheduler Activations

The fork() and exec() system calls

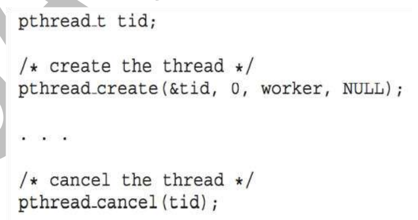
- Does fork()duplicate only the calling thread or all threads?
    - Some UNIXes have two versions of fork
- exec() usually works as normal – replace the running process including all threads

Signal Handling

- Signals are used in UNIX systems to notify a process that a particular event has occurred.
- A signal handler is used to process signals
    - Signal is generated by particular event
    - Signal is delivered to a process
    - Signal is handled by one of two signal handlers:
        - default
        - user-defined
- Every signal has default handler that kernel runs when handling signal
    - User-defined signal handler can override default
    - For single-threaded, signal delivered to process
- Where should a signal be delivered for multi-threaded?
    - Deliver the signal to the thread to which the signal applies
    - Deliver the signal to every thread in the process
    - Deliver the signal to certain threads in the process
    - Assign a specific thread to receive all signals for the process

Thread Cancellation

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



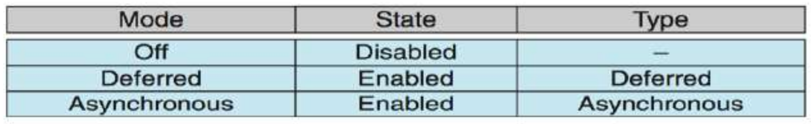
```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

. . .

/* cancel the thread */
pthread_cancel(tid);
```

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



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

- If thread has cancellation disabled, cancellation remains pending until thread enables it

- Default type is deferred
  - Cancellation only occurs when thread reaches cancellation point
    - I.e. pthread_testcancel()
    - Then cleanup handler is invoked
- On Linux systems, thread cancellation is handled through signals

Thread Pools

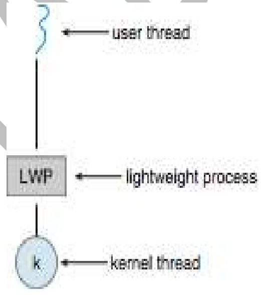
- Create a number of threads in a pool where they await work
- Advantages:
- Usually slightly faster to service a request with an existing thread than create a new thread
- Allows the number of threads in the application(s) to be bound to the size of the pool

Thread Specific Data

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

Scheduler Activations

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



UNIT II
Chapter 5 Process scheduling

CPU scheduling is the basis of multiprogrammed operating systems. By switching the CPU among processes, the operating system can make the computer more productive.

BASIC CONCEPTS

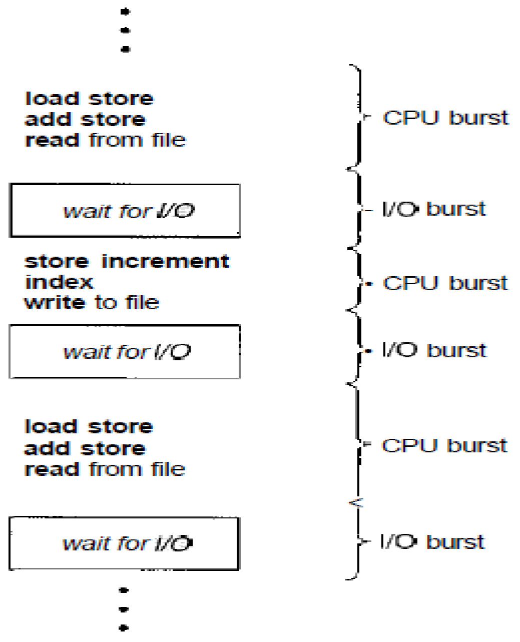
In a single-processor system, only one process can run at a time; any others must wait until the CPU is free and can be rescheduled. The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.

With multiprogramming, we try to use this time productively. Several processes are kept in memory at one time.

CPU-I/O BURST CYCLE

The success of CPU scheduling depends on an observed property of processes: Process execution consists of a **1) cycle of CPU execution** and **2) I/O wait**. Processes alternate between these two states.

The durations of CPU bursts have been measured extensively. Although they vary greatly from process to process and from computer to computer. they tend to have a frequency curve similar to that shown in Figure 5.2.



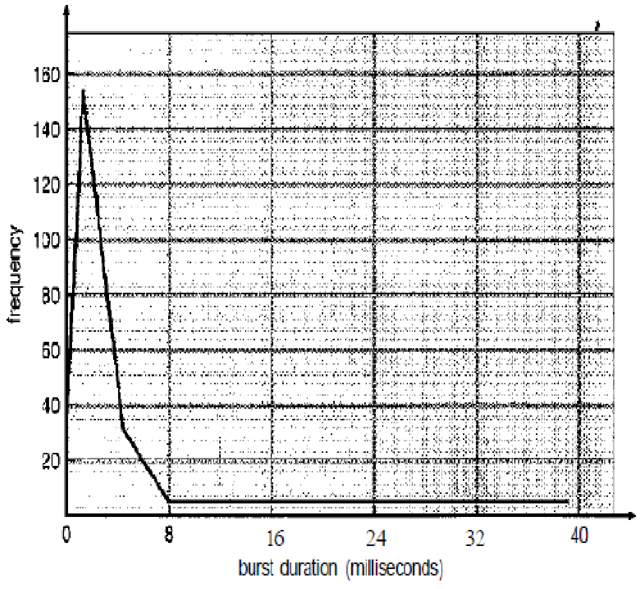


Figure 5.2 Histogram of CPU-burst durations.

gure 5.1 Alternating sequence of CPU and I/O bursts.

CPU Scheduler : Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the **short-term scheduler.**

Note that the ready queue is not necessarily a first-in, first-out (FIFO) queue.

<u>Preemptive Scheduling:</u> CPU-scheduling decisions may take place under the following four circumstances:

1. When a process switches from the running state to the waiting state (for example, as the result of an I/O request or an invocation of wait for the termination of one of the child processes)
2. When a process switches from the running state to the ready state *(ioi* example, when an interrupt occurs)
3. When a process switches from the waiting state to the ready state (for example, at completion of I/O)
4. When a process terminates.

NOTE: For situations 1 and 4, there is no choice in terms of scheduling.

When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is **non-preemptive** or **cooperative;** otherwise, it is **preemptive.**

<u>Dispatcher</u>
Another component involved in the CPU-scheduling function is the **dispatcher.**
The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler.

This function involves the following:

• Switching context
• Switching to user mode
• Jumping to the proper location in the user program to restart that program

The dispatcher should be as fast as possible, since it is invoked during every process switch.

Dispatch latency: The time it takes for the dispatcher to stop one process and start another running

<u>Scheduling Criteria</u> : Different CPU scheduling algorithms have different properties, and the choice of a particular algorithm depends on following…

• **CPU utilization:** Utilizing the services of CPU to maximum extent. it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).

• **Throughput:** The number of processes that are completed per time unit, called *throughput.* For long processes, this rate may be one process per hour; for short transactions, it may be 10 processes per second.

• **Turnaround time:** The interval from the time of submission of a process to the time of completion is the *turnaround time.* Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.

• **Waiting time:** The amount of time that a process spends waiting in the ready queue. OR *Waiting time* is the sum of the periods spent waiting in the ready queue.

NOTE: The CPU scheduling algorithm does not affect the amount of time during which a process executes or does I/O

• **Response time:** The time from the submission of a request until the first response is produced. This measure, called *response time,* is the time it takes to start responding, not the time it takes to output the response.

Scheduling Algorithms: CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU.

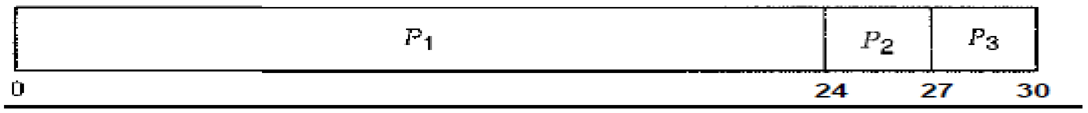
First-Come, First-Served Scheduling

1. The process that requests the CPU first is allocated the CPU first. It is managed with a FIFO queue.
2. When a process enters the ready queue, its PCB(process control block) is linked onto the tail of the queue.
3. When the CPU is free, it is allocated to the process at the head of the queue.
4. The running process is then removed from the queue.
5. The code for FCFS scheduling is simple to write and understand.
6. The average waiting time under the FCFS policy, however, is often quite long.

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

Process	**Burst Time**
P0	**24**
P 1	**3**
***p* 2**	**3**

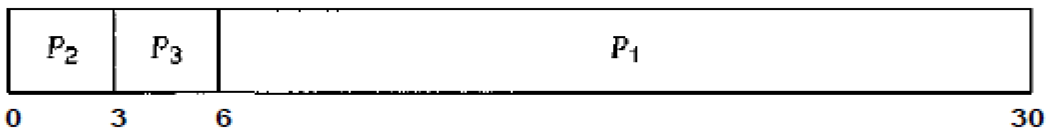
If the processes arrive in the order P1, *P2, P3,* and are served in FCFS order, we get the result shown in the following **Gantt chart:**



1. a. The waiting time is 0 milliseconds for process P1,
2. b. The waiting time is 24 milliseconds for process *P2,* and
3. c. The waiting time is 27 milliseconds for process P3

Thus, the average waiting time is (0 + 24 + 27) / 3 = 17 milliseconds.

If the processes arrive in the order *P2,* P3, P1, however, the results will be as shown in the following Gantt chart:



4. The average waiting time is now (6 + 0 + 3)/3 = 3 milliseconds.
5. This reduction is substantial. Thus, the average waiting time under an FCFS policy is generally not minimal and may vary substantially if the process's CPU burst times vary greatly.

Assume we have one CPU-bound process and many I/O-bound processes. The CPU-bound process will get and hold the CPU. During this time, all the other processes will finish their I/O and will move into the ready queue, waiting for the CPU. While the processes wait in the ready queue, the I/O devices are idle. Eventually, the CPU-bound process finishes its CPU burst and moves to an I/O device.

6. The FCFS scheduling algorithm is non-preemptive.
7. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O.
8. It create trouble for time-sharing systems, where it is important that each user get a share of the CPU at regular intervals.

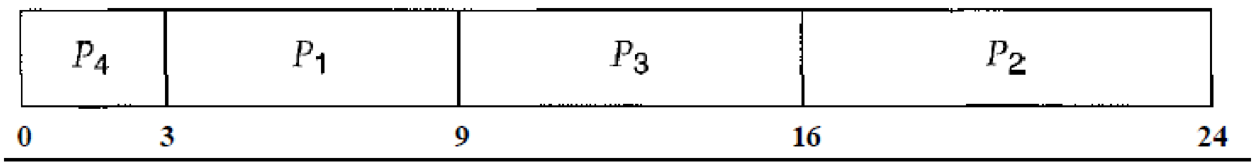
Shortest-Job-First Scheduling:

1. This algorithm associates with each process the length of the process's next CPU burst.
2. When the CPU is available, it is assigned to the process that has the smallest next CPU burst.
3. If the next CPU bursts of two processes are same, FCFS scheduling is used to break the tie.
4. The more appropriate term for this scheduling method would be **the *shortest-next-CPU-burst algorithm,*** because scheduling depends on the length of the next CPU burst of a process rather than its total length.

Consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	**Burst Time**
P1	**6**
P2	***8***
P3	**7**
P4	***3***

Gantt chart:



1. The waiting time is 3 milliseconds for process *p1,*
2. The waiting time is 16 milliseconds for process *P2*
3. The waiting time is 9 milliseconds for process *P3* and
4. The waiting time is 0 milliseconds for process P4.
5. Thus, the average waiting time is (3 + 16 + 9 + 0)/4 = *7* milliseconds.
6. By comparison, if we were using the FCFS scheduling scheme, the average waiting time would be 10.25 milliseconds.

FCFS Gantt chart

P1	P2	P3	P4
6	14	21	41

(Waiting time for p1, p2, p3, p4 = 0+6+14+21/4 = 41/4 =10.25)

7. The SJF scheduling algorithm is provably *optimal,* in that it gives the minimum average waiting time for a given set of processes.
8. The real difficulty with the SJF algorithm is knowing the length of the next CPU request.
9. The next CPU burst is generally predicted as an exponential average of the measured lengths of previous CPU bursts. Let *tn* be the length of the nth CPU burst_burst, and let Tn+1 be our predicted value for the next CPU burst. Then, for a, 0 < a < 1, define

T n + 1 =*a tn* + *(1 - a)tn.*

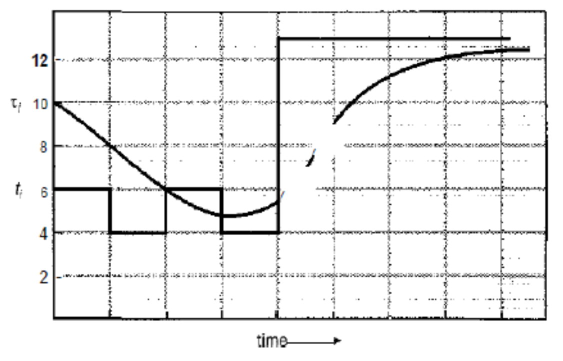
This formula defines an **exponential average.** the value of Tn contain our most recent information (Tn stores past history) The parameter " a" controls relative weight of recent history in our prediction. If a =0 then Tn+1 = Tn and recent history has no effect.
If a= 1 then Tn+1 = Tn i.e (Tn+1= 1. Tn +(1-1)Tn).

More commonly, a = 1/2, so recent history and past history are equally weighted.
we can expand the formula for T,,+I by substituting for TH, to find

$$\tau_{n+1} = at_{,,} + (1 - \alpha)\alpha t_{n-1} + \cdots + (1-\alpha)^{j}\alpha t_{n-j} + \cdots + (1-\alpha)^{n-1}\tau_0.$$

10. The SJF algorithm can be either preemptive or non-preemptive.

11. The choice arises when a new process arrives at the ready queue while a previous process is still executing. The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process.



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

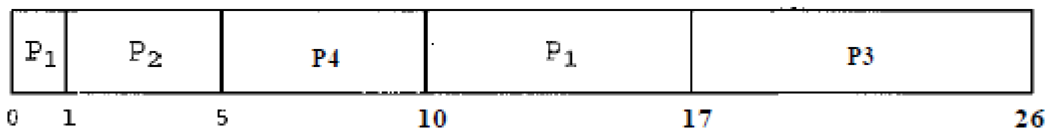
Figure 5.3 Prediction of the length of the next CPU burst.

12. A preemptive SJF Algorithm will preempt the currently executing process, whereas a nonpreemptive SJF algorithm will allow the currently running process to finish its CPU burst.
13. Preemptive SJF scheduling is sometimes called **shortest-remaining-time-first scheduling.**

Consider the following four processes, with the length of the CPU burst given in milliseconds

PROCESSES	**ARRIVAL TIME**	**BURST TIME**
P1	0	8
P2	1	4
P3	2	9
P4	3	5

If the processes arrive at the ready queue at the times shown and need the indicated burst times, then the resulting preemptive SJF schedule is as depicted in the following **Gantt chart**:



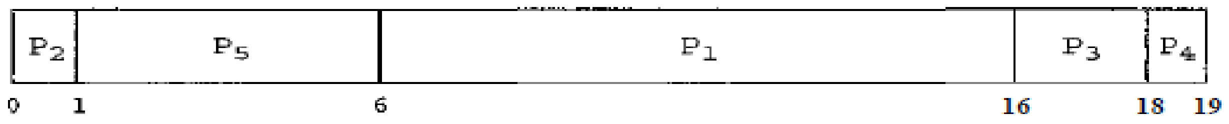
1. Process P1 is started at time 0, since it is the only process in the queue.
2. Process *P2* arrives at time 1. The remaining time for process P1 (7 milliseconds) is larger than the time required by process P2 (4 milliseconds), so process P1 is preempted, and process P2 is scheduled. This process continues predicting The average waiting time for this example is ((10 - 1) + (1 - 1) + (17 - 2) + (5 - 3))/4 = 26/4 = 6.5 milliseconds. Non-preemptive SJF scheduling would result in an average waiting time of 7.75 milliseconds.

Priority Scheduling :

1. A priority is associated with each process, and the CPU is allocated to the process with the highest priority.
2. Equal-priority processes are scheduled in FCFS order.
3. An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst.
4. The larger the CPU burst, the lower the priority, and vice versa.
5. scheduling is done in terms of *high* priority and *low* priority.
6. Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095.
7. However, there is no general agreement on whether 0 is the highest or lowest priority.

consider the following set of processes, assumed to have arrived at time 0, in the order below…

PROCESSES	**BURST TIME**	**PRIORITY**
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2



1. The average waiting time is 8.2 milliseconds.

   Waiting time for P1=6 ,p2= 0 ,p3= 16 , p4= 18 , p5= 1

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

2. Priorities can be defined either internally or externally.
3. Internally defined priorities use some measurable quantity or quantities to compute the priority of a process.
4. External priorities are set by criteria outside the operating system, such as the importance of the process, the type and amount of funds being paid for computer use, the department sponsoring the work, and other, often political, factors.
5. Priority scheduling can be either preemptive or non-preemptive. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process, based on which it will follow up with preemptive or non-preemptive.

Indefinite blocking or **starvation:** In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU. Here two possible condition will occur either process will get the CPU and accomplish the task or gets crash waiting for CPU.

Aging : The problem to indefinite blocking is overcome with aging, It is technique of gradually increasing the priority of processes that wait in system for long time…..

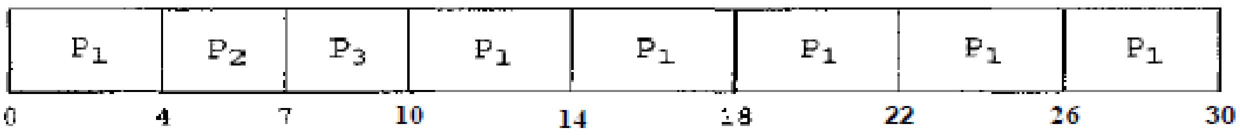
Round-Robin Scheduling

1. The **round-robin (RR) scheduling algorithm** is designed especially for timesharing systems.
2. It is similar to FCFS scheduling, but preemption is added to switch between processes.
3. A small unit of time, called a **time quantum** or time slice, is defined The ready queue is treated as a circular queue.
4. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval To implement RR scheduling, we keep the ready queue as a FIFO queue of processes.
5. New processes are added to the tail of the ready queue.
6. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.
7. A context switch will be executed, and the process will be put at the **tail** of the ready queue.

The average waiting time under the RR policy is often long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

PROCESSES	**BURST TIME**
P1	24
P2	3
P3	3

1. If we use a time quantum of 4 milliseconds, then process P1 gets the first 4 milliseconds.
2. Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process *P2.* Since process P2 does not need 4 milliseconds, it quits before its time quantum expires.
3. The CPU is then given to the next process, process P3. Once each process has received 1 time quantum, the CPU is returned to process P1 for an additional time quantum.



4. The average waiting time is 17/3 = 5.66 milliseconds.
5. In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row.
6. If processes process's CPU burst exceeds 1 time quantum, that process is *preempted* and is put back in the ready queue. **The RR scheduling algorithm is thus preemptive**.

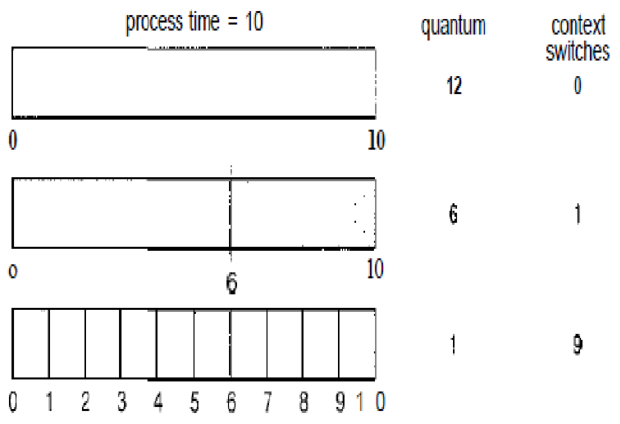
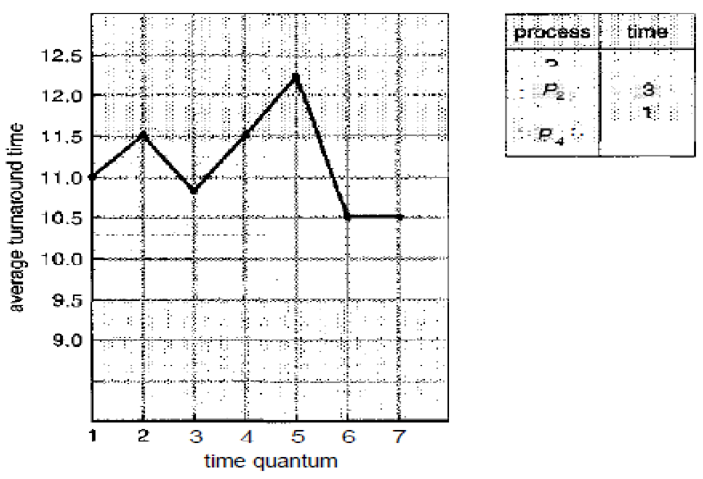


Figure 5.4 The way in which a smaller time quantum increases context switches.



jure 5.5 The way in which turnaround time varies with the time quantum.

7. Turnaround time also depends on the size of the time quantum. As we can see from Figure 5.5, the average turnaround time of a set of processes does not necessarily improve as the time-quantum size increases.
8. In general, the average turnaround time can be improved if most processes finish their next CPU burst in a single time quantum.

<u>**Multilevel Queue Scheduling :**</u> Another scheduling method where processes are classified as **foreground** (interactive) processes and **background** (batch) processes. These two types of processes have different response-time requirements and so may have different scheduling needs.

1. **Foreground** processes have high priority over **Background** processes.
2. It partitions the ready queue into several separate queues (Below Figure)
3. The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type.
4. Each queue has its own scheduling algorithm. For example, separate queues might be used for foreground and background processes.
5. The foreground queue might be scheduled by an RR algorithm, while the background queue is scheduled by an FCFS algorithm.

Let's look at an example of a multilevel queue scheduling algorithm with five queues, listed below in order of priority:
1. System processes
2. Interactive processes
3. Interactive editing processes
4. Batch processes
5. Student processes

NOTE: If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted. Another possibility is to time-slice among the queues. Here, each queue gets a certain portion of the CPU time, which it can then schedule among its various processes.

Multilevel Feedback-Queue Scheduling : It allows a process to move between queues(system ,interactive, interactive editing, Batch and student).

1. The idea is to separate processes (foreground and background) according to the characteristics of their CPU bursts.
2. If a process uses too much CPU time, it will be moved to a lower-priority queue.
3. This scheme leaves I/O-bound and interactive processes in the higher-priority queues.

Example: A process entering the ready queue is put in queue 0. A process in queue 0 is given a time quantum of 8 milliseconds. If it does not finish within this time, it is moved to the tail of queue 1. If queue 0 is empty, the process at the head of queue 1 is given a quantum of 16 milliseconds. If it does not complete, it is preempted and is put into queue 2. Processes in queue 2 are run on an FCFS basis but are run only when queues 0 and 1 are empty.

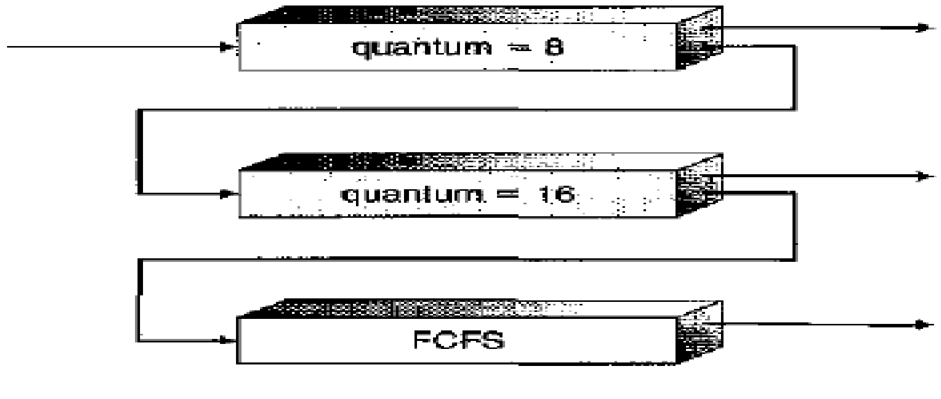


Figure 5.7 Multilevel feedback queues.

Parameter defining multilevel feedback queues are…..

• The number of queues
• The scheduling algorithm for each queue
• The method used to determine when to upgrade a process to a higher priority queue
• The method used to determine when to demote a process to a lower priority queue
• The method used to determine which queue a process will enter when that process needs service

Multiple-Processor Scheduling : with homogeneous multiprocessors, there are sometimes limitations on scheduling. Consider a system with an I/O device attached to a private bus of one processor. Processes that wish to use that device must be scheduled to run on that processor.

Approaches to Multiple-Processor Scheduling : Two major Approaches are as follows

1. First A**symmetric multiprocessing** is simple because only one processor accesses the system data structures, reducing the need for data sharing.
2. Second S**ymmetric multiprocessing (SMP),** where each processor is self-scheduling. All processes may be in a common ready queue, or each processor may have its own private queue of ready processes.

Processor Affinity : To overcome cache problem, Most SMP systems try to avoid migration of processes (Task) from one processor to another and instead attempt to keep a process running on the same processor. This is known as **processor affinity.** It is of Two types **1. Soft Affinity 2. Hard Affinity.**

1. **Soft Affinity:** Operating System attempting to keep process running on same processor, but not guaranteeing weather assigned task will get accomplish.
2. **Hard Affinity:** Operating system allows process to migrate to other processor, using standard system calls.
3. **Load balancing :** attempts to keep the workload evenly distributed across all processors in an SMP system. It is only necessary on systems where each processor has its own private queue of eligible processes to execute.

There are two general approaches to **Load Balancing**:

1. **push migration** and
2. **Pull migration.**

push migration,- a specific task periodically checks the load on each processor and—if it finds an imbalance—-evenly distributes the load by moving (or pushing) processes from overloaded to idle or less-busy processors.

Pull migration- occurs when an idle processor pulls a waiting task from a busy processor. Push and pull migration need not be mutually exclusive and are in fact often implemented in parallel on load-balancing systems.

Symmetric Multithreading: SMP an alternative strategy is to provide multiple *logical*— rather than *physical*—processors. It has also been termed **hyper-threading technology** on Intel processors.

The idea behind SMT is to create multiple logical processors on the same physical processor,. Each logical processor has its own **architecture state,** which includes general-purpose and machine-state registers.

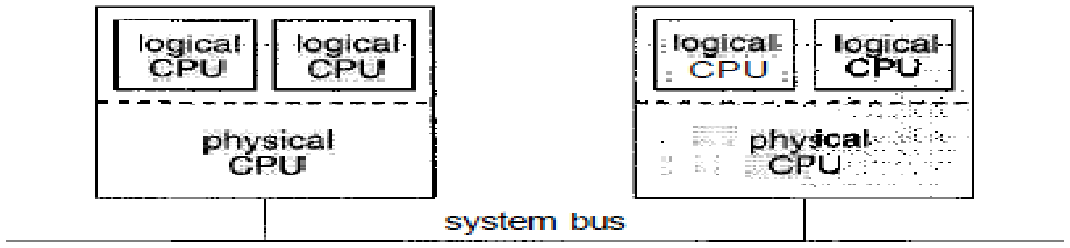


Figure 5.8 A typical SMT architecture

<u>**Thread Scheduling**</u> threads to the process model are distinguished between

1. *user-level* and 2. *kernel-level* threads.
2. Operating systems that support is kernel-level threads—not processes—that are being scheduled by the operating system.
3. User-level threads are managed by a thread library, and the kernel is unaware of them. To run on a CPU, user-level threads must ultimately be mapped to an associated kernel-level thread, although this mapping may be indirect and may use a lightweight process (LWP).
4. The thread library schedules user-level threads to run on an available LWP, a scheme known as **process-contention scope (PCS),** since competition for the CPU takes place among threads belonging to the same process.
5. To decide which kernel thread to schedule onto a CPU, the kernel uses **system-contention scope** (SCS). Competition for the CPU with SCS scheduling takes place among all threads in the system.

<u>**Pthread Scheduling**</u> The POSIX Pthread API that allows specifying either PCS or SCS during thread creation. Pthreads identifies the following contention scope values:

- PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling.
- PTHREAD-SCOPE_SYSTEM schedules threads using SCS scheduling
- On systems implementing the many-to-many model the PTHREAD_SCOPE_PROCESS policy schedules user-level threads onto available LVVPs.
- The Pthread IPC provides the following two functions for getting—and setting—-the contention scope policy:

• pthread_attr_setscope(pthread_attr_t *attr, int scope)
• pthread_attr_getscope(pthread_attr_t *attr, int *scope)

The first parameter for both functions contains a pointer to the attribute set for the thread. The second parameter for the pthread_attr_setscope 0 function is passed either the PTHREAD.SCOPE..SYSTEM or PTHREAD_5COPE_PROCESS value, indicating how the contention scope is to be set. In the case of pthread_attr_getscope(), this second parameter contains a pointer to an int value that is set to the current value of the contention scope. If an error occurs, each of these functions returns non-zero values.

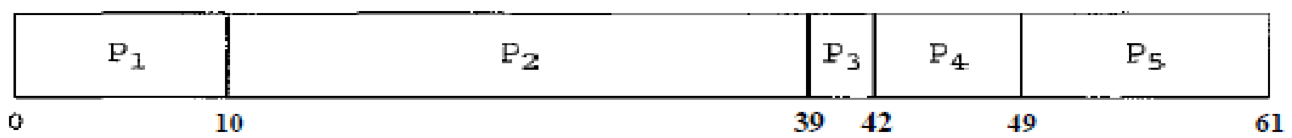
Operating System Examples The scheduling policies of the Solaris, Windows XP, and Linux operating systems. It is important to remember that we are describing the scheduling of kernel threads with Solaris and Linux.

Example: Solaris Scheduling
Solaris uses priority-based thread scheduling. It has defined four classes of scheduling, which are, in order of priority:
1. Real time
2. System
3. Time sharing
4. Interactive

Deterministic Modeling This method takes a particular predetermined workload and defines the performance of each algorithm for that workload. For example, assume that we have the workload shown below. All five processes arrive at time 0, in the order given, with the length of the CPU burst given in milliseconds:

PROCESSES	**BURST TIME**
P1	10
P2	29
P3	3
P4	7
P5	12

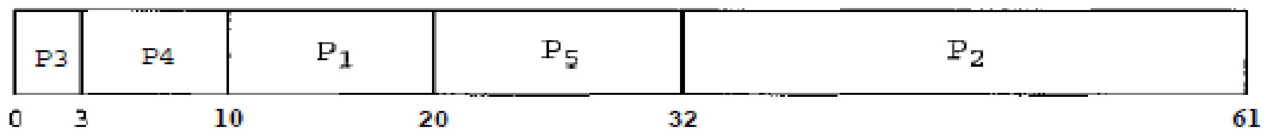
Consider the FCFS, SJF, and RR (quantum = 10 milliseconds) scheduling algorithms for this set of processes. Which algorithm would give the minimum average waiting time?
For the FCFS algorithm, we would execute the processes as



1. The waiting time is 0 milliseconds for process P1,
2. The waiting time is 10 milliseconds for process p2
3. The waiting time is 39 milliseconds for process P3,
4. The waiting time is 42 milliseconds for process P4, and
5. The waiting time is 49 milliseconds for process P5.

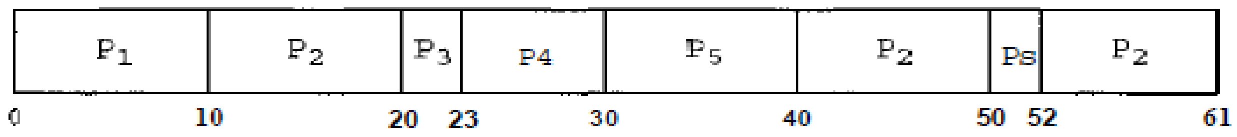
6. Thus, the average waiting time is (0 + 10 + 39 + 42 + *49)/5* = 28 milliseconds.

With non-preemptive SJF scheduling, we execute the processes as



1. The waiting time is 10 milliseconds for process *P1*
2. The waiting time is 32 milliseconds for process *P2,*
3. The waiting time is 0 milliseconds for process P3,
4. The waiting time is 3 milliseconds for process P4, and
5. The waiting time is 20 milliseconds for process P5.
6. Thus, the average waiting time is (10 + 32 + 0 + 3 + 20)/5 = 13 milliseconds.

With the RR algorithm, we execute the processes as



1. The waiting time is 0 milliseconds
2. for process P1,
3. 32 milliseconds for process *P2,*
4. 20 milliseconds for process P3,
5. 23 milliseconds for process P4, and
6. 40 milliseconds for process P5.
7. Thus, the average waiting time is (0 + 32 + 20+ 23 + 40)/5 = 23 milliseconds.

   We see that, *in this case,* the average waiting time obtained with the SJF policy is less than half that obtained with FCFS scheduling; the RR algorithm gives us an intermediate value.

NOTE: SJF policy will always result in the minimum waiting time.

let *n* be the average queue length (excluding the process being serviced), let W be the average waiting time in the queue, and let λ be the average arrival rate for new processes in the queue (such as three processes per second). the number of processes leaving the queue must be equal to the number of
processes that arrive. Thus, This equation, known as **Little's formula,** **n= λ xW.**

Chapter 6 Process Synchronization

A cooperating process is one that can affect or be affected by other processes executing in the system can either directly share a logical address space (that is, both code and data) or be allowed to share data only through files or messages.

The producer and consumer routines are correct separately, they may not function correctly when executed concurrently, suppose that the value of the variable counter is currently 5 and that the producer and consumer processes execute the statements "counter++"& "counter--" concurrently.

Following the execution of these two statements, the value of the variable counter may be 4, 5, or 6! The only correct result, though, is counter == 5, which is generated correctly if the producer and consumer execute separately.

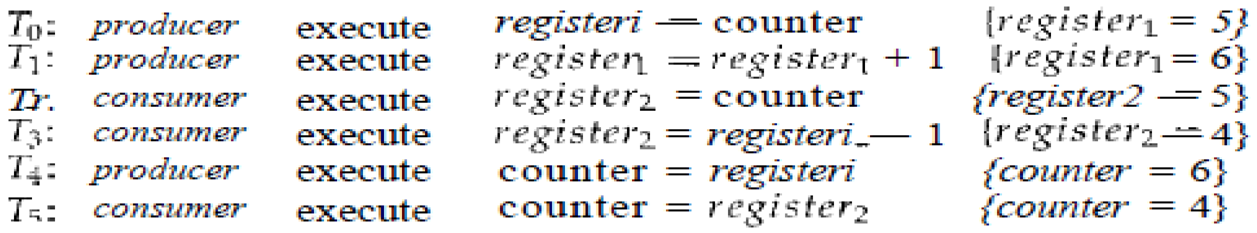
We can show that the value of counter may be incorrect as follows. Note that the statement "counter++" may be implemented in machine language (on a typical machine) as

Register1 =- counter
Register1 = *register1* + 1
counter =*register1*

where *register1* is a local CPU register. Similarly, the statement "counter--" is implemented as follows:
register2=counter
register2= *register2* -1
counter = *register2*

where again *register2* is a local CPU register. Even though *register}* and *register^* may be the same physical register (an accumulator, say), that the contents of this register will be saved and restored by the interrupt handler.

The concurrent execution of "counter++" and "counter- -" is equivalent to a sequential execution



T_0:	*producer*	execute	*registeri* — counter	{*register*$_1$ = *5*}
T_1:	*producer*	execute	*registen*$_1$ = *register*$_1$ + 1	{*register*$_1$ = 6}
Tr.	*consumer*	execute	*register*$_2$ = counter	{*register2* — = 5}
T_3:	*consumer*	execute	*register*$_2$ = *registeri*_— 1	{*register*$_2$ —= 4}
T_4:	*producer*	execute	counter = *registeri*	{*counter* = 6}
T_5:	*consumer*	execute	counter = *register*$_2$	{*counter* = 4}

Notice that we have arrived at the incorrect state "counter == 4", indicatingthat four buffers are full, when, in fact, five buffers are full. If we reversed the order of the statements at T4 and T5, we would arrive at the incorrect state "counter --= 6".

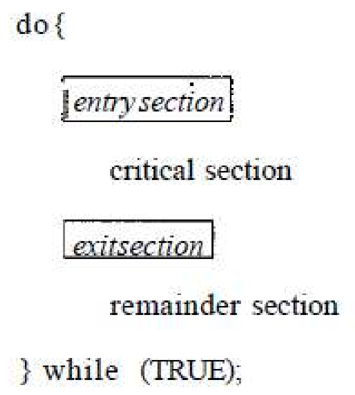
** A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **Race Condition.**

To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the variable counter.

<u>The Critical-Section Problem</u>

1. Each process has a segment of code, called a **critical section,** in which the process may be changing common variables, updating a table, writing a file, and so on.
2. No two processes should enter their critical section at same time.

3. A protocol is used to cooperate among processes ,i.e must request permission to enter its critical section.
4. The section of code implementing this request is called E**ntry section.**
5. The critical section may be followed by an **Exit section.**
6. The remaining code is the **remainder section.**



```
do{
    entry section
        critical section
    exitsection
        remainder section
} while  (TRUE);
```

General structure of a typical process P_i .

A solution to the critical-section problem must satisfy the following three requirements:
1. Mutual exclusion. If process Pi is executing in its critical section, then no other processes should execute in their critical sections.
2. **Progress.** If no process is executing in its critical section and some processes wish to enter their critical sections, (and not executing in their remainder sections) can participate in the decision on which process will enter its critical section next, and this selection cannot be postponed indefinitely.
3. Bounded waiting. There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request.

Two general approaches are used to handle critical sections in operating systems:
(1) **preemptive kernels :** A preemptive kernel allows a process to be preempted while it is running in kernel mode.

(2) Non-preemptive kernels : A Non-preemptive kernel does not allow a process running in kernel mode to be preempted, a kernel-mode process will run until it exits kernel mode or blocks. Obviously, a non-preemptive kernel is essentially free from race conditions on kernel data structures, as only one process is active at a time in kernel mode.
Peterson's Solution : It addresses the requirements of mutual exclusion, progress, and bounded waiting requirements.

Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. The processes are numbered Po and P1. For convenience, when presenting Pi we use *Pj* to denote the other process; that is, j equals 1 = i.

Peterson's solution requires two data items to be shared between the two processes:

int turn ;
bolean f l a g [2] ;

1. The variable turn indicates whose turn it is to enter its critical section. That is, if turn == i, then process Pi is allowed to execute in its critical section.
2. The flag array is used to indicate if a process *is ready* to enter its critical section. For example, if f lag[i] is true, this value indicates that Pi, is ready to enter its critical section.
3. To enter the critical section, process Pi first sets flag[i] to be true and then sets turn to the value j, thereby asserting that if the other process wishes to enter the critical section, it can do so.The value of turn will decide which of these two processes will enter into critical section...

```
do {

     flag[i] = TRUE;
     turn = j;
     while (flag[j] && turn == j);

        critical section

     flag[i] = FALSE;

        remainder section

} while (TRUE);
```

6.2 The structure of process P_i in Peterson's solution

We now prove that this solution is correct. We need to show that:
1. Mutual exclusion is preserved.
2. The progress requirement is satisfied.
3. The bounded-waiting requirement is met.

To prove property of **Mutual Exclusion** , we note that each P; enters its critical section only if either flag[j] == false or turn -- i. Also note that, if both processes can be executing in their critical sections at the same time, then flag [0] == flag [1] == true. These two observations imply that Po and Pi could not have successfully executed their while statements at about the same time, since the value of turn can be either 0 or 1 but cannot be both.

To prove propertys Progress and bounded waiting , If Pj , is not ready to enter the critical section, then flag [j] == false, and Pi can enter its critical section. If *Pj* has set flag [j] to true and is also executing in its while statement, then either turn == i or turn == j . If turn == i, then Pi will enter the critical section. If turn == j, then *Pj* will enter the critical section. However, once *Pj* exits its critical section, it will reset f lag[j] to false, allowing Pi to enter its critical section.

```
do {

    acquire lock

        critical section

    releaselock

        remainder section

} while  (TRUE);
```

Solution to the critical-section problem using locks.

Synchronization Hardware: we can state that any solution to the critical-section problem requires a simple tool—a lock.

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

igure 6.4 The definition of the TestAndSet() instruction.

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

Figure 6.6 The definition of the Swap () instruction.

The TestAndSet() instruction can execute simultaneously at different computers. But not allow to process to enter their critical section using lock tool which is set to false.

```
do {
   while (TestAndSetLock(&lock))
      ; // do nothing

      // critical section

   lock = FALSE;

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

.5 Mutual-exclusion implementation with TestAndSet ().

The above condition is possible with only two processes if and only if they execute within the test And set ( ) range..

The Swap( ) instruction, in contrast to the TestAndSet0 instruction,operates on the contents of two words.

<u>Semaphores</u> : A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: **wait ( ) and signal ( ).**

1. The **wait ( )** operation was originally termed P (from the Dutch *probercn,* "to test");
2. **signal ( )** was originally called V (from *verhogen,* "to increment"). The definition of wait 0 is as follows:

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

The definition of signal() is as follows:

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

1. All the modifications to the integer value of the semaphore in the wait ( ) and signal( ) operations must be executed indivisibly.
2. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.
3. Operating systems often distinguish between counting and binary semaphores.
4. The value of a **counting semaphore** can range over an unrestricted domain.
5. The value of a **binary semaphore** can range only between 0 and 1. On some systems, binary semaphores are known as **mutex locks,** as they are locks that provide *mutual* exclusion.
6. We can use binary semaphores to deal with the critical-section problem, The *n* processes share a semaphore, mutex, initialized to 1.
7. **<u>Counting semaphores</u>** can be used to control access to a given resource consisting of a finite number of instances.
8. Each process that wishes to use a resource performs a waitQ operation on the semaphore (thereby decrementing the count).
9. When a process releases a resource, it performs a signal () operation (incrementing the count). When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

For example, consider two concurrently running processes: *P1* with a statement S1 and *P2* with a statement *S2.* Suppose we require that *S2* be executed only after S1 has completed. We can implement this scheme readily by letting P1 and *P2* share a common semaphore synch, initialized to 0, and by inserting the
statements

```
S1;                /*process p1 with s1 instance*/
signal(synch);
```

```
wait(synch);   /*process p2 with s2 instance*/
    S2;
```

Implementation: The main disadvantage of the semaphore definition given here is that it requires **busy waiting. i.e** While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code.

Busy waiting wastes CPU cycles that some other process might be able to use productively. This type of semaphore is also called a **spinlock** because the process "spins" while waiting for the lock.

1. When a process executes the wait () operation and finds that the semaphore value is not positive, it must wait. the process can *block* itself.
2. A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal() operation.
3. The process is restarted by a wakeup () operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue

Deadlocks and Starvation. The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event and waiting long processes will enter into situation called deadlock.

OR

Two or more processes waiting for an event and enter a situation called deadlock. Another problem related to deadlocks is **indefinite blocking,** or **starvation,** a situation in which processes wait indefinitely within the semaphore.

Classic Problems of Synchronization

A data base is to be shared among several concurrent processes. Some of these processes may want only to read the data base, whereas other may want to update (i.e., to read and write) the data base we distinguish between these two types of processes by referring to the former as readers and to the latter as writers. Obviously, if two readers access the shared data simultaneously, number of adverse affects will result.

1. To ensure that these difficulties do not arise we require that the writers have exclusive access to the shared data base. This synchronization problem is referred to as **the readers-writers problems.**
2. The readers-writers problem has several variations, all involving priorities. The simplest one, referred to as the first readers-writers problem, requires that no reader will be kept waiting unless a writer has already obtain permission to use the shared object.
3. In other words, no reader should wait for other readers to finish simply because a writer is writing.
4. The second readers-writers problem requires that, once a writer is ready that writer performs its write as soon as possible. In other words, if a writer is waiting to access the object, no new reader may start reading.

A solution to either problem may result in starvation. (i.e., situation I which processes wait in definitely within the semaphore). In the first case, writers may starve. In the second case, readers may starve.
In the solution to the first readers-writers problem, the reader processes share the following data structure:

```
Semaphore mutex,wrt;
Int readcount;
```

Mutex –locks that provide mutual exclusion (i.e. if process Pi isexecuting in its critical section, then on other processes can be executing in their critical section).

1. In the above data structure, semaphore mutex and wrt are initialized to 1. Readcount is initialized to 0.
2. The semaphore wrt is common to both reader and writer processes. The mutex semaphore is used to ensure mutual exclusion when the variable readcount is updated.
3. The readcount variable keep tracks ofhow many processes are currently reading the object.
4. The semaphore wrt function as a mutual- exclusion semaphore for the writer. It is also used by the first or last reader that enters or exits the critical section.
5. It is not used by the readers who enters or exits while other readers are in their critical sections.

The code for a writer process is shown in figure mention below:

```
do
{
wait(wrt);
………
//writing is performed
………
signal(wrt);
}while(TRUE);
```

Fig: Structure of a writer process.

The code for a reader process is show in figure mention below:

```
do
{
wait(mutex);
readcount++;
if(readcount==1)
wait (wrt);
signal(mutex);
………..
//reading is performed
………..
wait(mutex);
readcount--;
if(readcount==0)
signal(wrt);
signal(mutex);
}while(TRUE);
```

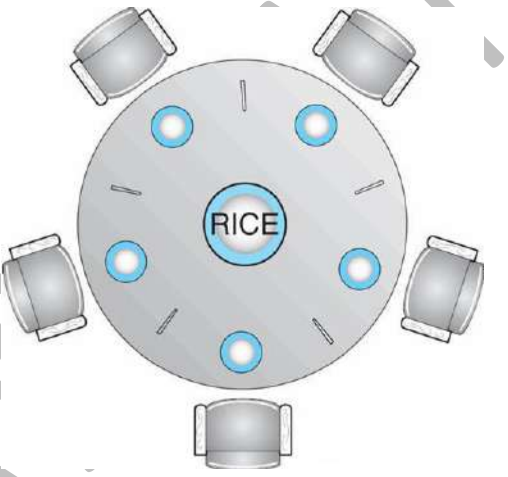
Fig: Structure of a reader process

Note that, if a writer is in the critical section and n reader are waiting, then one reader is queued on wrt, and n-1 readers are queued on mute.

Also observe that, when a writer executes signal (wr), we may resume the execution of either the waiting readers on a signal waiting writing. The selection is made by the scheduler.

DINNING – PHILOSPHERS PROBLEM :-

Consider five PHILOSPHERS who spend their lives thinking and eating. The PHILOSPHERS share a circular table surrounded by five chairs, each belonging to one PHILOSPHER. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks as shown in figure shown below:



1. When a philosopher thinks, he does not interact with his colleagues; a philosopher gets hungry and tries to pick up the two chopsticks that are closest to him.
2. A philosopher may pick up only one chopstick at a time. He cannot pick up a chopstick that is already in the hand of a neighbor.
3. When a hungry philosopher has both of his chopsticks at the same time, he eats releasing his chopsticks,when he finished eating.he puts down both of his chopsticks and start thinking again.
4. The dinning-philosophers problem is consider a classic synchronization problem because it is an example of a large class of concurrency-control problem.
5. One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a wait ()

6. Operation on that semaphore, he release his chopstick by executing the signal () operation on the appropriate semaphore. Thus, the shared data are:

```
Semaphore chopstick[5];
```

Where all the elements of the chopstick are initialized to 1. The structure of philosopher I is shown in figure below:

```
do{
Wait(chopsticki]);
Wait(chopstick(i+1)%5);
……..
//eat
……..
Signal(chopstick[i]);
Signal(chopstick[i+1]%5);
……….
//think
……….
}while(TRUE);
```

1. Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it could create a deadlock.
2. Suppose that all five philosophers become hungry simultaneously and each grabs his left chopstick.
3. All the elements of the chopstick will now be equal to 0.
4. When each philosopher tries to grab his right chopstick he will be delayed forever.
5. Allow at most four philosophers to be sitting simultaneously at the table.
6. Allow a philosopher to pick up his chopsticks only if both chopsticks are available (to do this he must pick them up in a critical section).
7. Use an asymmetric solution; that is, an odd philosopher picks up first his left chopstick and then his right chopstick, whereas an even philosopher picks up his right chopstick and then his left chopstick.

Monitor: A monitor encapsulates some state, and provides methods that operate on that state. 2. In addition, it guarantees that these methods are executed in a mutually exclusive manner. 3.Thus if a process tries to invoke a method from a specific monitor, and some method of that monitor is already being executed by another process, the first process is blocked until the monitor becomes available.
4.A procedure defined within a monitor can access only those variables declared locally within the monitor and its formal parameters.

```
monitor monitor name f
{
        /* shared variable declarations*/
procedure PI ( . . . ) {
}
p r o c e d u r e P 2 ( . . . )
{
p r o c e d u r e P n ( . . . )
 {
i n i t i a l i z a t i o n c o d e ( . . . )
 {
}
}
```

The monitor construct ensures that only one process at a time can be active within the monitor.
Additional support is provided by the condition construct the synchronization scheme can define one or more variables of type *condition:*

condition x, y;

The only operations that can be invoked on a condition variable are wait () and signal(). The operation

x.waitO ;

means that the process invoking this operation is suspended until another process invokes

x . s i g n a l ( ) ;

The x. signal () operation resumes exactly one suspended process If no process is suspended, then the signal () operation has no effect; that is, the state of x is the same as if the operation had never been executed

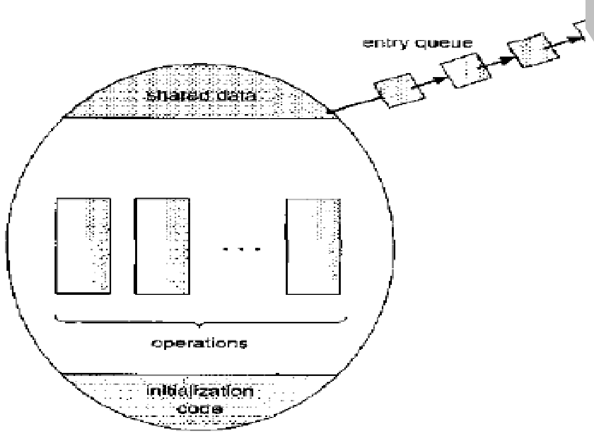
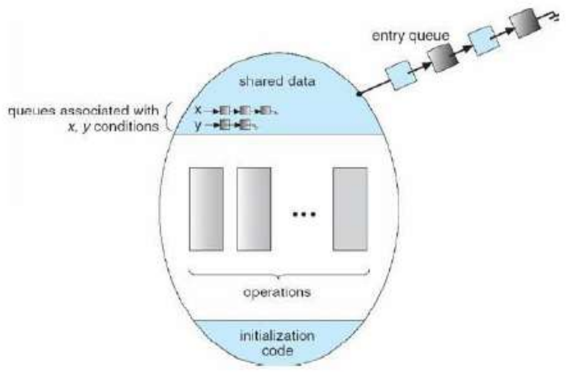


Figure 6.17 Schematic view of a monitor.

Now suppose tht, when the x.**s** ignal () operation is invoked by a process P, there is suspended process Q associated with condition x.
Two possibilities exist:
1. Signal and wait. *P* either waits until Q leaves the monitor or waits for another condition.
2. Signal and continue. *Q* either waits until *P* leaves the monitor or waits for another condition.

Monitor with Condition Variables



Solution to Dining Philosophers

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

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


void test (int i) {
              if ( (state[(i + 4) % 5] != EATING) &&
              (state[i] == HUNGRY) &&
              (state[(i + 1) % 5] != EATING) ) {
                   state[i] = EATING ;
                     self[i].signal () ;
               }
         }
     initialization_code() {
             for (int i = 0; i < 5; i++)
             state[i] = THINKING;
        }
}
```

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

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

Monitor Implementation Using Semaphores

Variables

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

Monitor Implementation

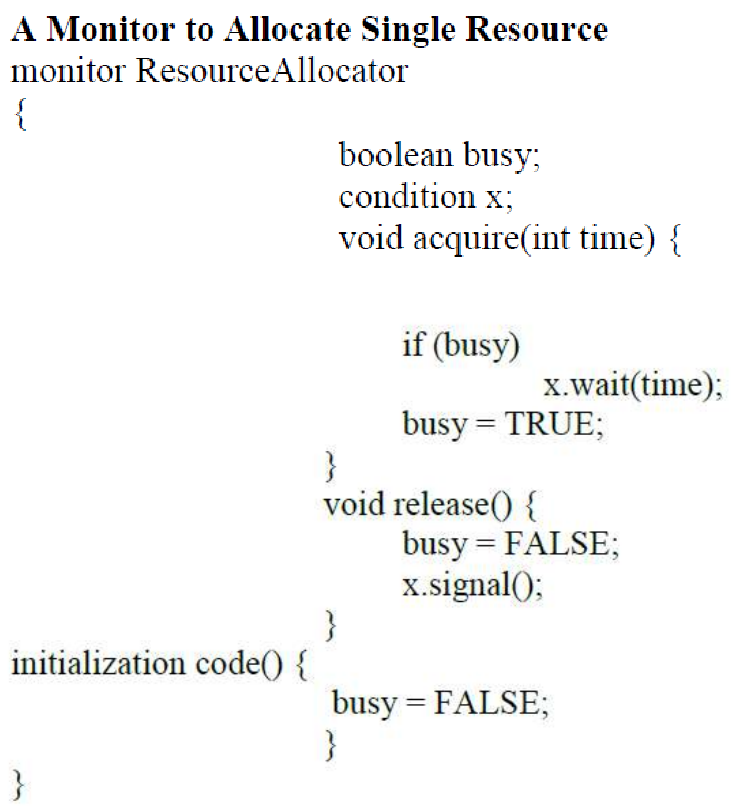
For each condition variable ***x***, we have:

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

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

The operation x.signal can be implemented as:

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



A Monitor to Allocate Single Resource

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

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

Chapter 7 DEADLOCKS

Sometimes, a waiting process will never again able to change state, because the resources it has requested are held by other waiting processes. This situation of process is called a deadlock.

Under the normal mode of operation, a process may utilize a resource in only the following sequence:

1. Request. If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.

2. Use, The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).

3. Release. The process releases the resource.

A deadlock, processes never finish executing, and system resources are tied up, preventing other jobs from starting

Necessary Conditions

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

1.Mutual exclusion. At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

2. Hold and wait. A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
3. **No preemption.** Resources cannot be preempted.; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.
4. Circular wait. A set {*P1, P2, ..., Pn*\ of waiting processes must exist such that *P1* is waiting for a resource held by *P2, P2* is waiting for a resource held by P3, •••, Pn-1, is waiting for a resource held by *Pn,* and *1,* is waiting for a resource held by Pn.

We emphasize that all four conditions must hold for a deadlock to occur.

7.2.1 Resource-Allocation Graph

Deadlocks can be described more precisely in terms of a directed graph called a system resource allocation graph.

This graph consist of a set of vertices *V* and a set of edges *E.*

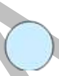
V is partitioned into two types:

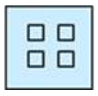
$P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system

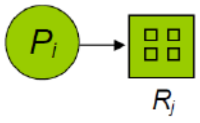
$R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system

request edge – directed edge $P_i \rightarrow R_j$

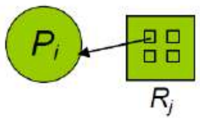
assignment edge – directed edge $R_j \rightarrow P_i$

Process 

Resource Type with 4 instances 

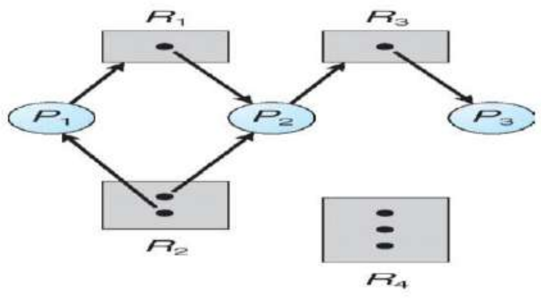


P_i requests instance of R_j

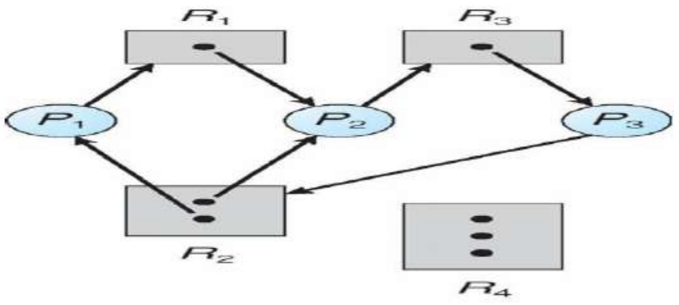


P_i is holding an instance of R_j

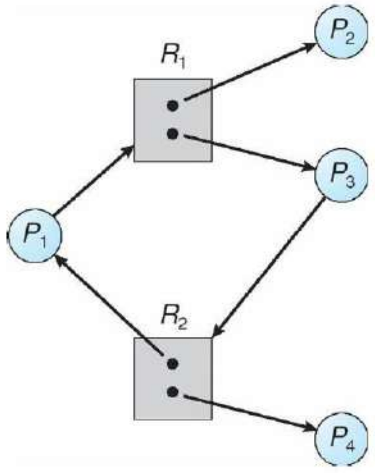
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



Graph With A Cycle But No Deadlock



Basic Facts

If graph contains no cycles ⇒ no deadlock

If graph contains a cycle ⇒

if only one instance per resource type, then deadlock if several instances per resource type, possibility of deadlock

7.3 Methods for Handling Deadlocks

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

Deadlock prevention

Deadlock avoidence

Allow the system to enter a deadlock state and then recover

Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX

7.4 Deadlock Prevention

Restrain the ways request can be made

Mutual Exclusion – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources

Hold and Wait – must guarantee that whenever a process requests a resource, it does not hold any other resources

- Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none allocated to it.
- Low resource utilization; starvation possible

No Preemption –

- If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
- Preempted resources are added to the list of resources for which the process is waiting
- Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

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

7.5 Deadlock Avoidance

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

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

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

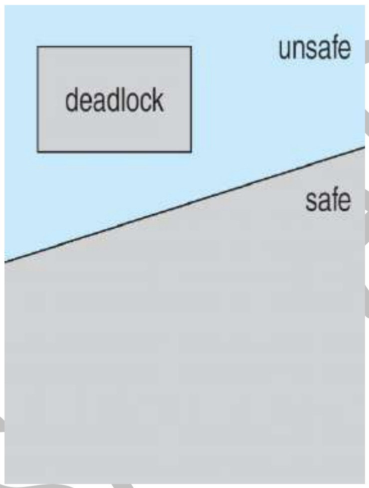
Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- System is in safe state if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of ALL the processes in the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$
- That is:
    - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
    - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
    - When P_i terminates, P_{i+1} can obtain its needed resources, and so on

Basic Facts

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

Safe, Unsafe and deadlock state spaces



Avoidance Algortithms

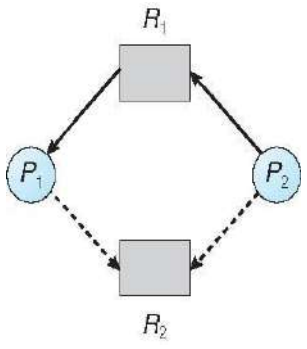
- Single instance of a resource type
  Use a resource-allocation graph

- Multiple instances of a resource type
  Use the banker's algorithm

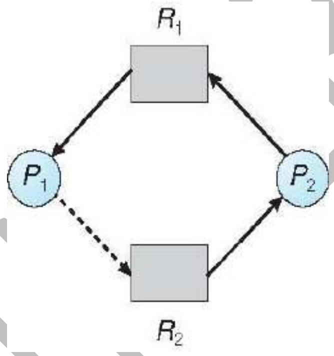
Resource-Allocation Graph Scheme

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

Resource Allocation Graph



Unsafe State in Resource-Allocation Graph



Resource-Allocation Graph Algorithm

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

Banker's Algorithm

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

Data Structures for the Banker's Algorithm

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

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

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

Safety Algorithm

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

   Work = Available

   ***Finish* [*i*] = *false* for *i* = 0, 1, …, *n*- 1**

2. Find an ***i*** such that both:

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

   (b) $\boldsymbol{Need_i \leq Work}$

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

3. $\boldsymbol{Work = Work + Allocation_i}$
   Finish*[*i*] = *true
   go to step 2

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

Resource-Request Algorithm for Process P_i

$\boldsymbol{Request_i}$ = request vector for process $\boldsymbol{P_i}$. If $\boldsymbol{Request_i\ [j] = k}$ then process $\boldsymbol{P_i}$ wants ***k*** instances of resource type $\boldsymbol{R_j}$

1. If $\boldsymbol{Request_i \leq Need_i}$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim

2. If $\boldsymbol{Request_i \leq Available}$, go to step 3. Otherwise $\boldsymbol{P_i}$ must wait, since resources are not available

3. Pretend to allocate requested resources to $\boldsymbol{P_i}$ by modifying the state as follows:

   $\boldsymbol{Available = Available - Request_i;}$

   $\boldsymbol{Allocation_i = Allocation_i + Request_i;}$

   $\boldsymbol{Need_i = Need_i - Request_i;}$

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

Example of Banker's Algorithm
5 processes *P*0 through *P*4;
3 resource types:
A (10 instances), B (5instances), and *C* (7 instances)

Snapshot at time T_0:

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

The content of the matrix *Need* is defined to be *Max* – *Allocation*

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

The system is in a safe state since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria

Example: P_1 Request (1,0,2)

Check that Request £ Available (that is, (1,0,2) £ (3,3,2) Þ true

Available

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

Executing safety algorithm shows that sequence < P_1, P_3, P_4, P_0, P_2> satisfies safety requirement
Can request for (3,3,0) by P_4 be granted?
Can request for (0,2,0) by P_0 be granted?

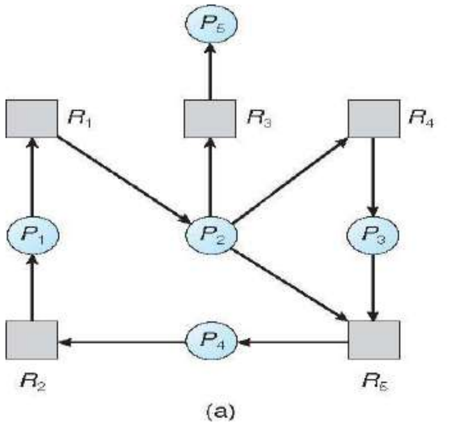
7.6 Deadlock Detection

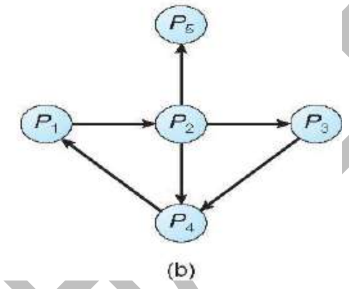
- Allow system to enter a deadlock state
- Detection algorithm examines the state of the system to determine whether a deadlock has occurred.
- Recovery scheme helps in recovering from the deadlock.

Single Instance of Each Resource Type
- Maintain **wait-for** graph

- Nodes are processes
- $P_i \rightarrow P_j$ if P_i is waiting for P_j

- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph





(a) resource allocation graph (b) corresponding wait-for graph

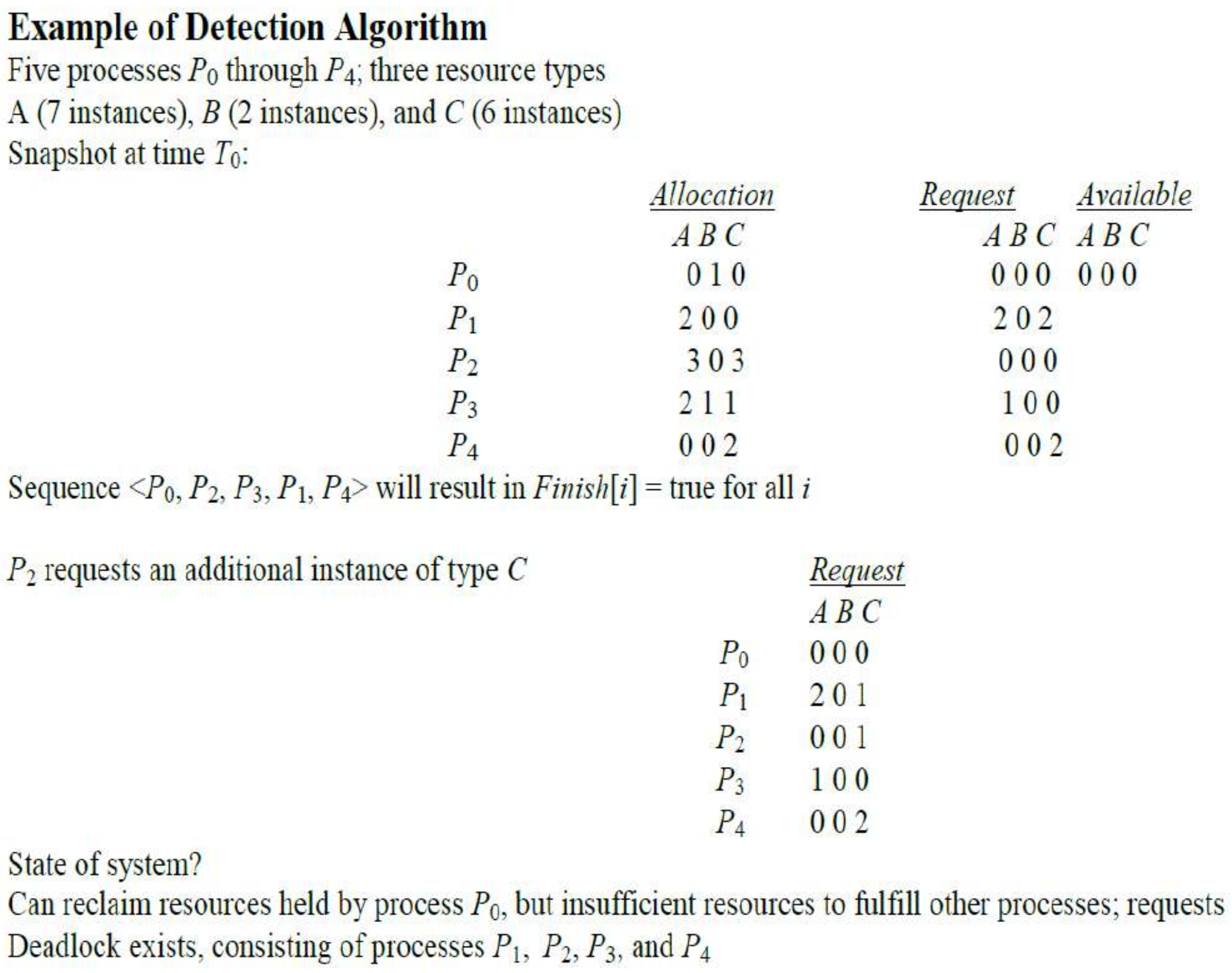
Several Instances of a Resource Type

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

Detection Algorithm

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

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



Example of Detection Algorithm

Five processes P_0 through P_4; three resource types

A (7 instances), B (2 instances), and C (6 instances)

Snapshot at time T_0:

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

Sequence <P_0, P_2, P_3, P_1, P_4> will result in *Finish*[*i*] = true for all *i*

P_2 requests an additional instance of type C

	Request
	A B C
P_0	0 0 0
P_1	2 0 1
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

State of system?

Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests

Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4

Detection-Algorithm Usage

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

7.7 Recovery from Deadlock: Process Termination

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
    1. Priority of the process
    2. How long process has computed, and how much longer to completion
    3. Resources the process has used
    4. Resources process needs to complete
    5. How many processes will need to be terminated
    6. Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

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

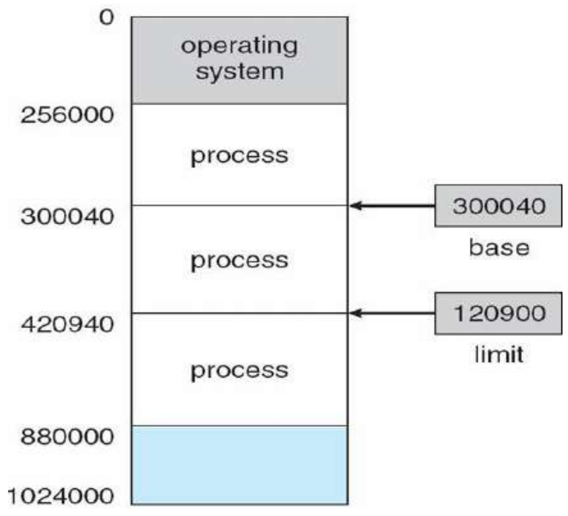
UNIT III

Part-I - Memory Management

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

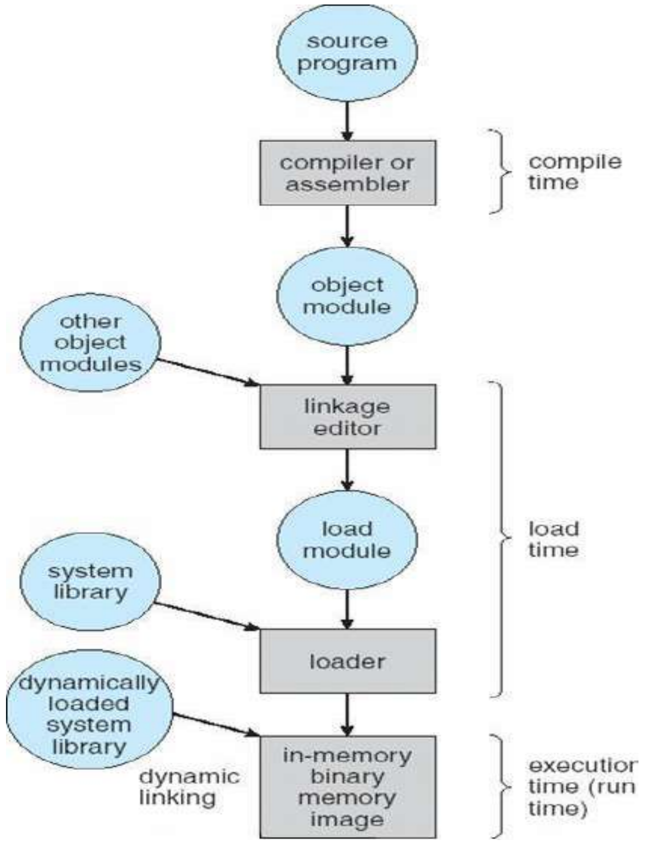
Base and Limit Registers

A pair of **base** and **limit** registers define the logical address space
Base Register: Holds smallest legal physical memory address.
Limit Register: Specifies the size of the range.



Binding of Instructions and Data to Memory

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



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

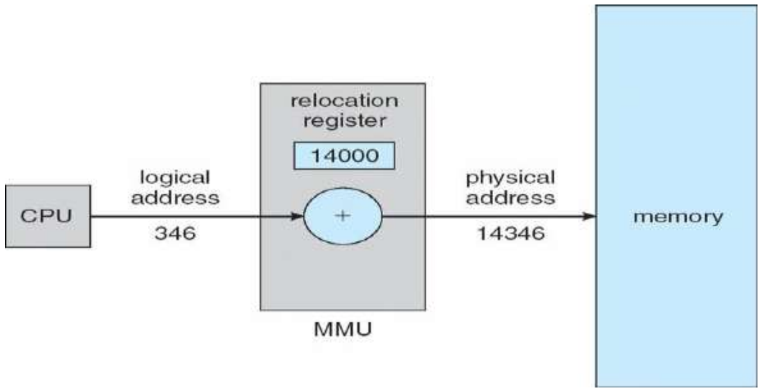
Memory-Management Unit (MMU)

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 register



Dynamic Loading

Routine is not loaded until it is called

Better memory-space utilization; unused routine is never loaded

Useful when large amounts of code are needed to handle infrequently occurring cases

No special support from the operating system is required implemented through program design

Dynamic Linking

Linking postponed until execution time

Small piece of code, *stub*, used to locate the appropriate memory-resident library routine Stub replaces itself with the address of the routine, and executes the routine

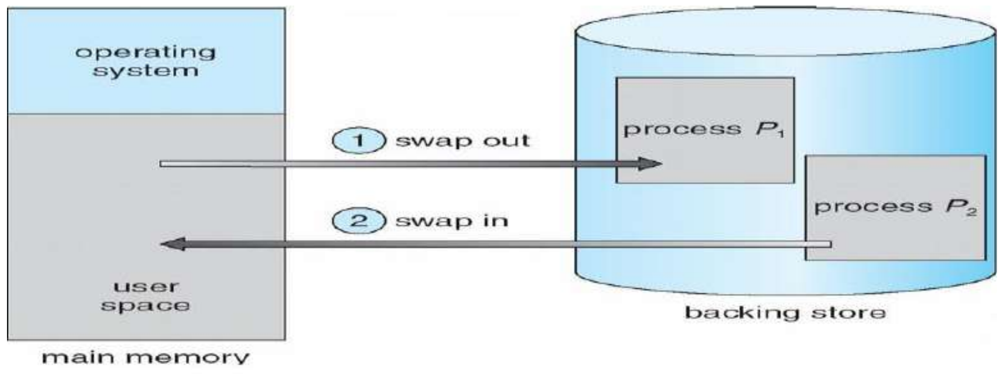
Operating system needed to check if routine is in processes' memory address

Dynamic linking is particularly useful for libraries
System also known as **shared libraries**

Swapping

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

Schematic View of Swapping



Contiguous Allocation

Main memory usually into two partitions:

Resident operating system, usually held in low memory with interrupt vector

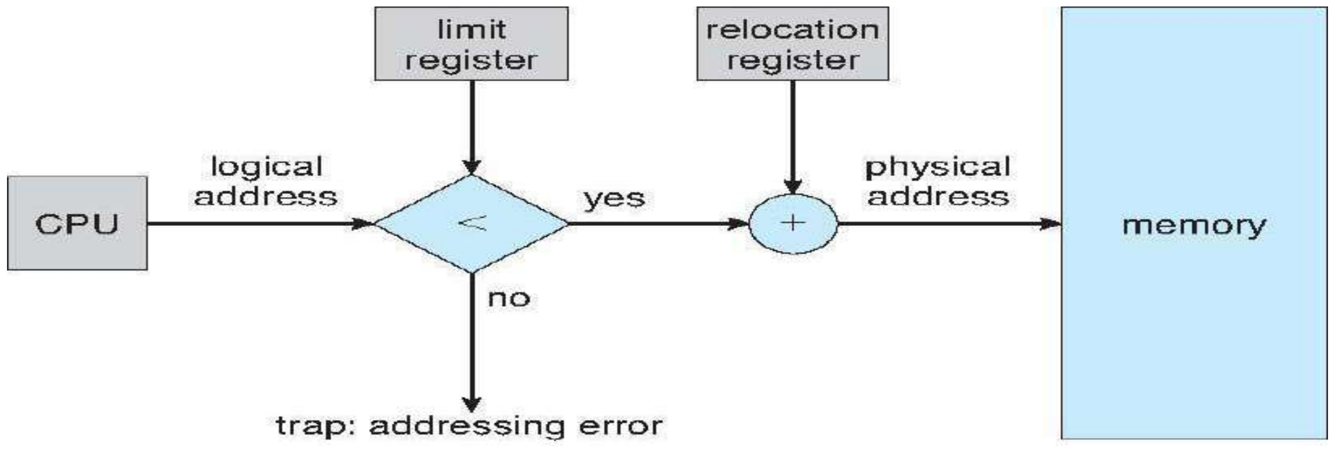
User processes then held in high memory Relocation registers used to protect user processes from each other, and from changing operating-system code and data

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*

Hardware Support for Relocation and Limit Registers

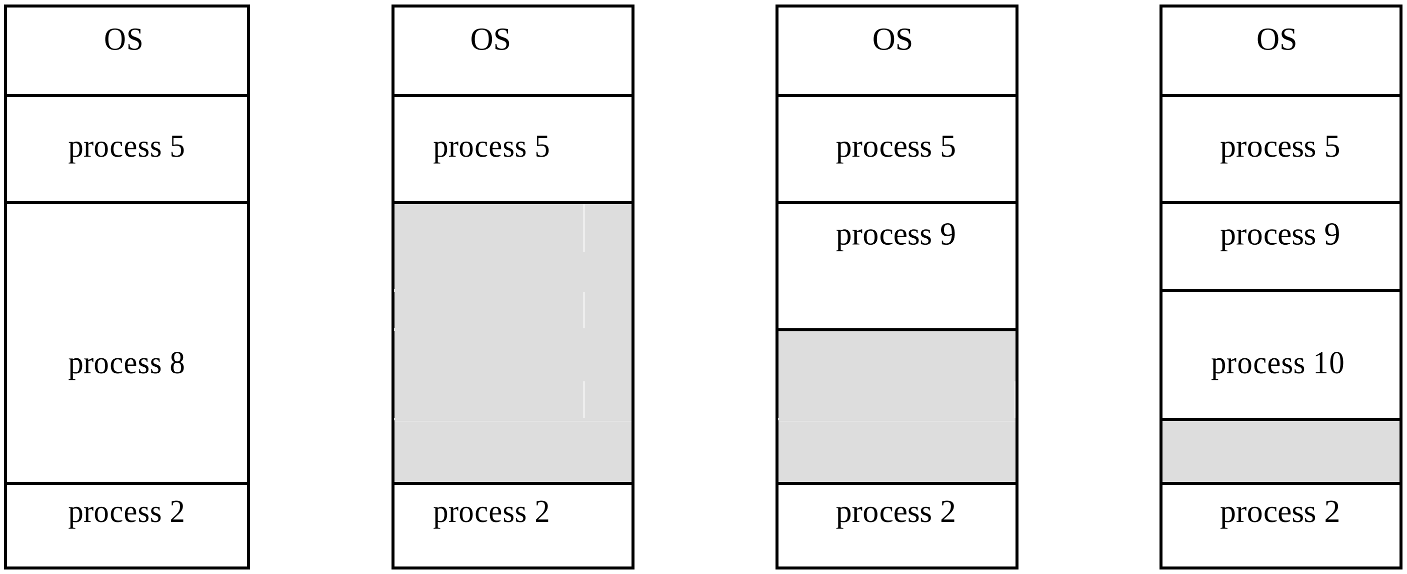


Multiple-partition allocation

Hole – block of available memory; holes of various size are scattered throughout memory
When a process arrives, it is allocated memory from a hole large enough to accommodate it

Operating system maintains information about:

a) allocated partitions b) free partitions (hole)



Dynamic Storage-Allocation Problem

First-fit: Allocate the *first* hole that is big enough

Best-fit: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size Produces the smallest leftover hole

Worst-fit: Allocate the *largest* hole; must also search entire list Produces the largest leftover hole

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

Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- Reduce external fragmentation by **compaction**
- Shuffle memory contents to place all free memory together in one large block Compaction is possible *only* if relocation is dynamic, and is done at execution time.
- I/O problem

  Latch job in memory while it is involved in I/O
  Do I/O only into OS buffers

Paging

Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available

Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8,192 bytes)

Divide logical memory into blocks of same size called **pages** Keep track of all free frames

- To run a program of size ***n*** pages, need to find *n* free frames and load program
- Set up a page table to translate logical to physical addresses

Internal fragmentation

Address Translation Scheme

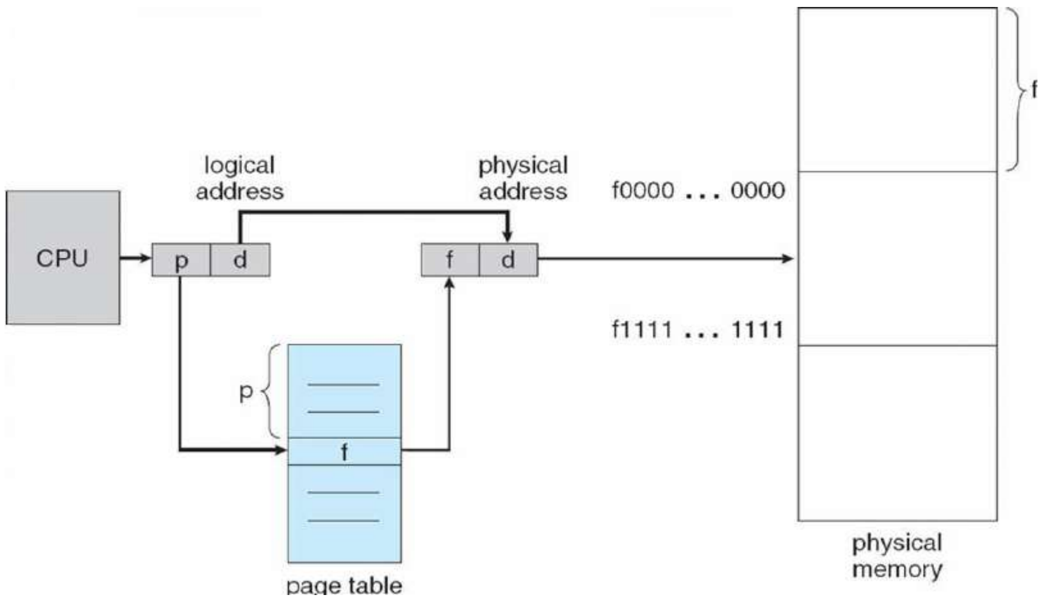
Address generated by CPU is divided into

Page number (*p*) – used as an index into a *page table* which contains base address of each page in physical memory

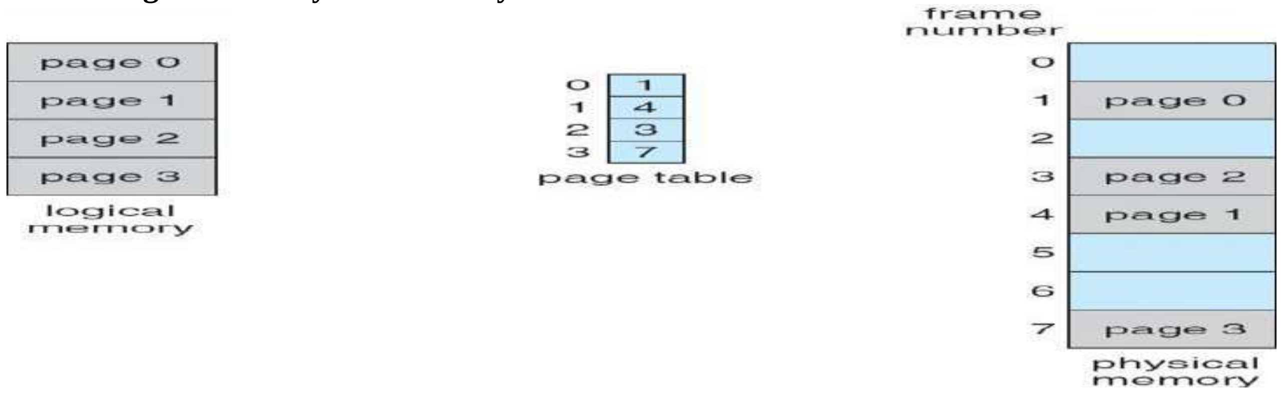
Page offset (d) – combined with base address to define the physical memory address that is sent to the memory unit

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

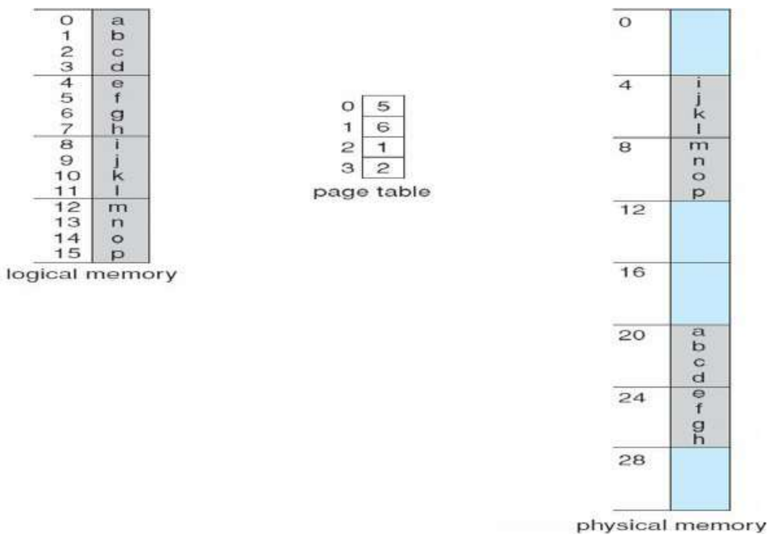
Paging Hardware



Paging Model of Logical and Physical Memory

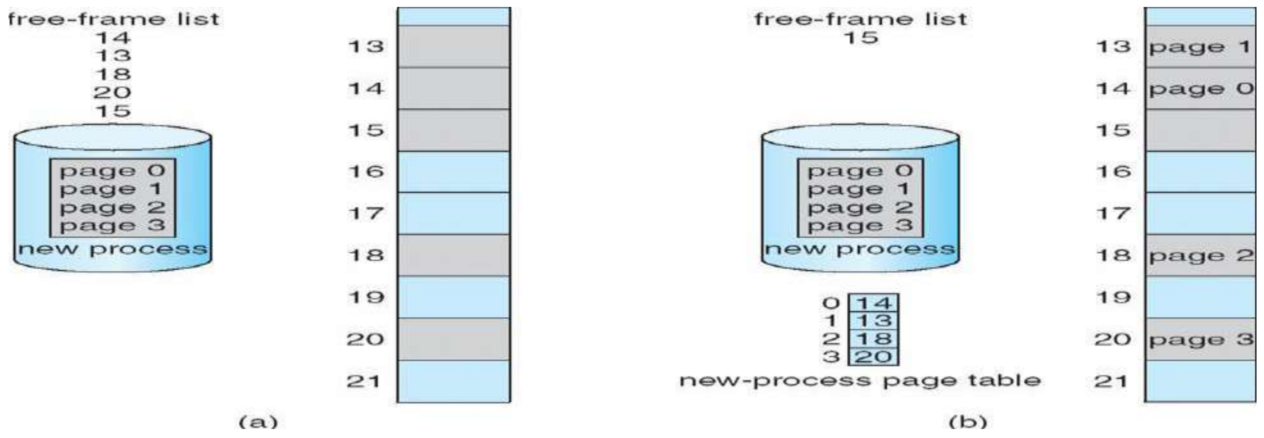


Paging Example



32-byte memory and 4-byte pages

Free Frames



Implementation of Page Table

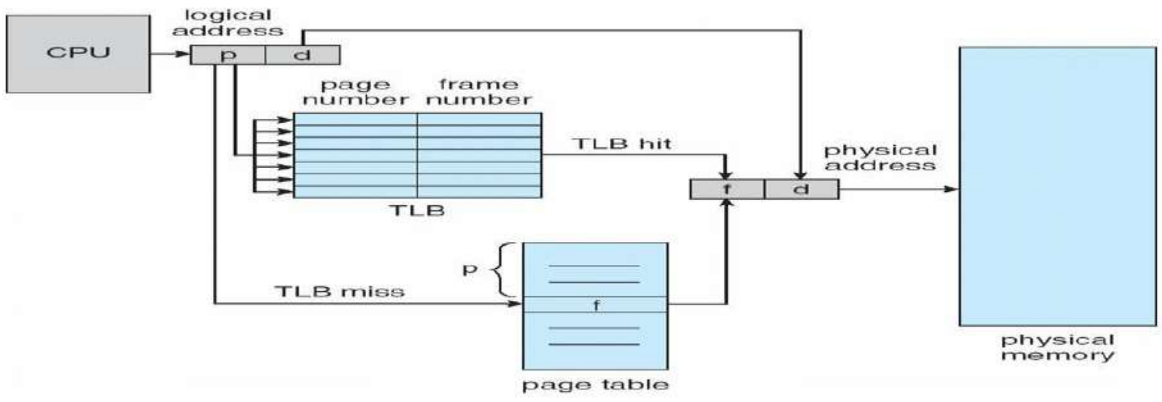
- Page table is kept in main memory
- **Page-table base register (PTBR)** points to the page table
  Page-table length register (PRLR) indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers (TLBs)**
- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address-space protection for that process

Associative Memory

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

Page #	Frame #

Paging Hardware with TLB



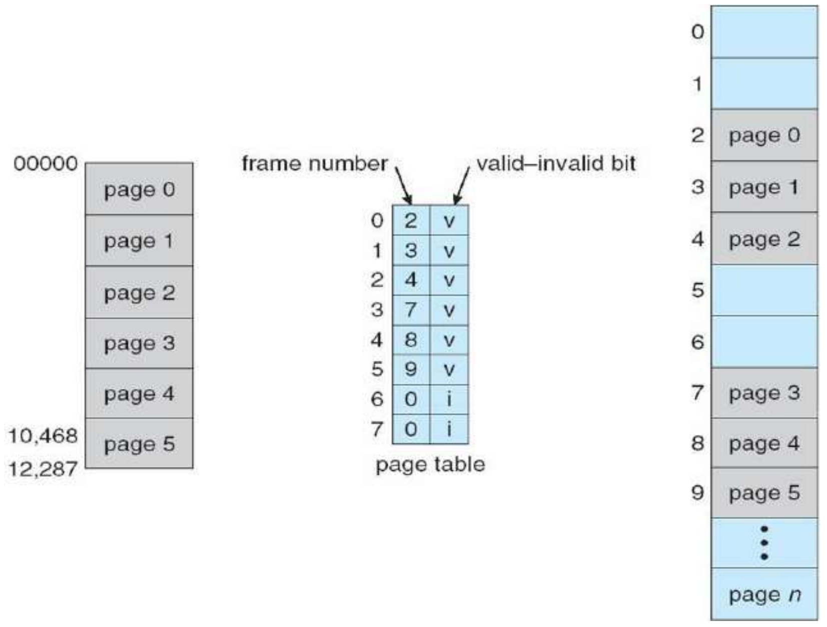
Effective Access Time

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

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

Memory Protection

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



Shared Pages

Shared code

One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).

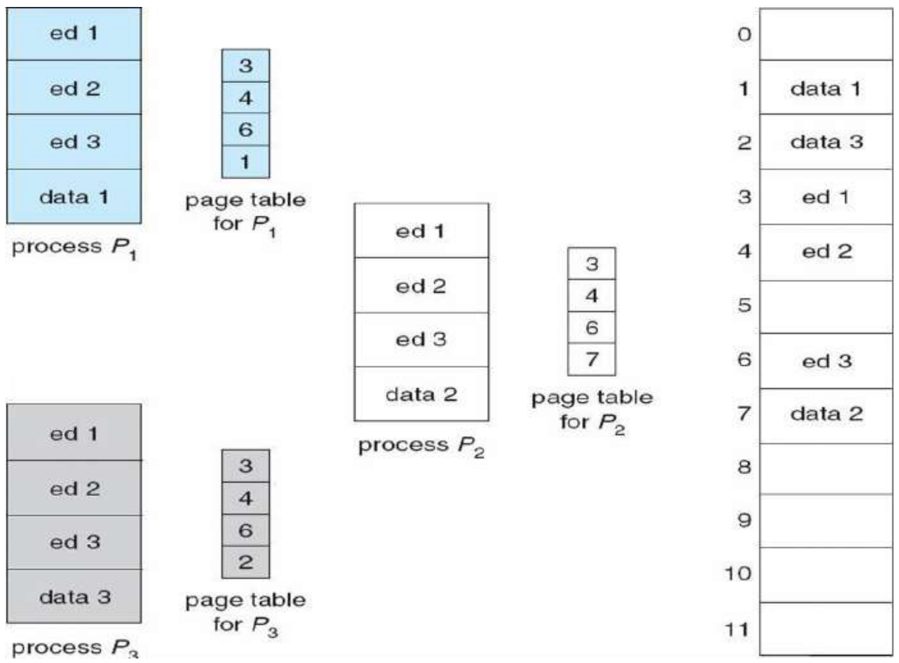
Shared code must appear in same location in the logical address space of all processes

Private code and data

Each process keeps a separate copy of the code and data

The pages for the private code and data can appear anywhere in the logical address space

Shared Pages Example



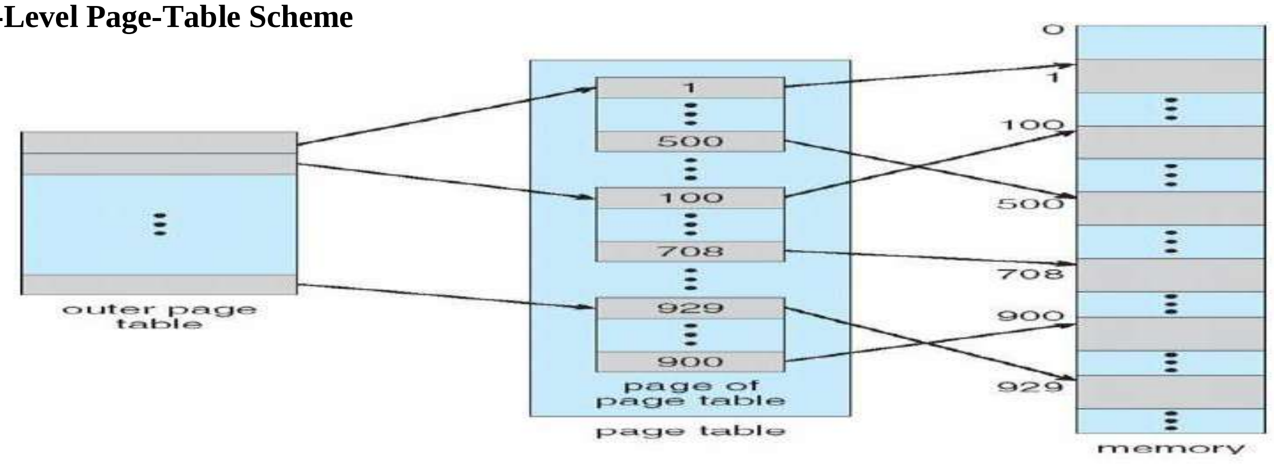
Structure of the Page Table

a. Hierarchical Paging
b. Hashed Page Tables
c. Inverted Page Tables

Hierarchical Page Tables

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

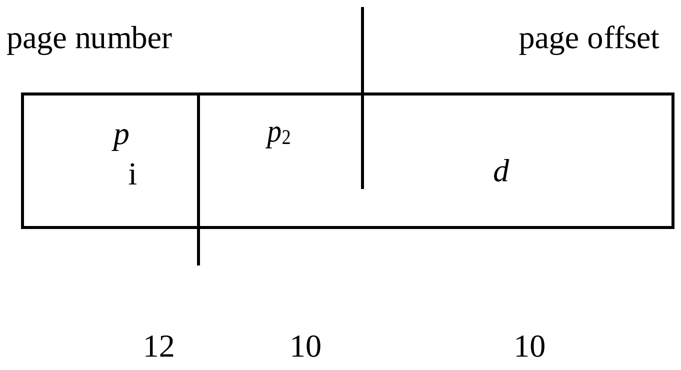
Two-Level Page-Table Scheme



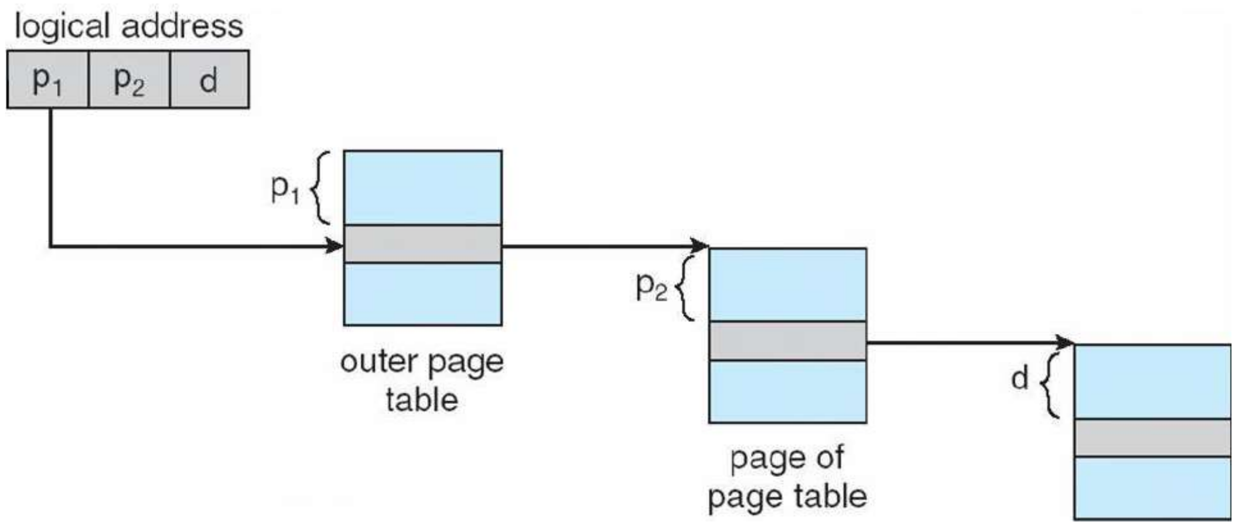
Two-Level Paging Example

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

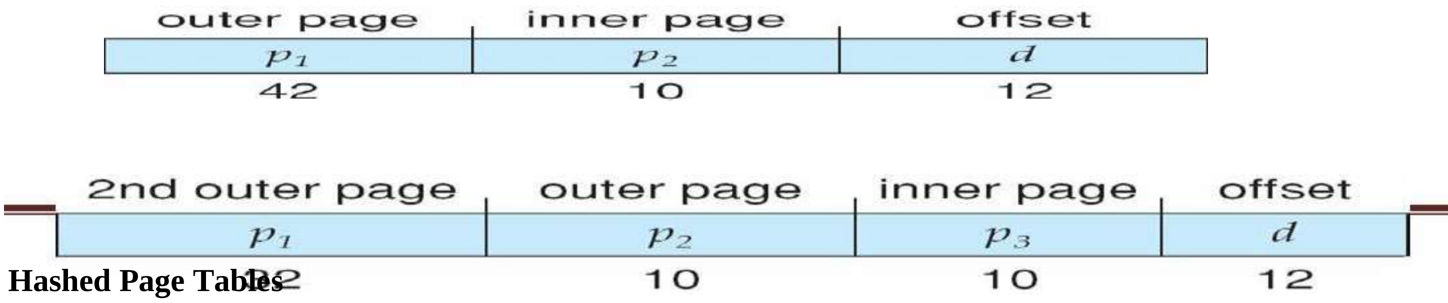
where *pi* is an index into the outer page table, and *p2* is the displacement within the page of the outer page table



Address-Translation Scheme



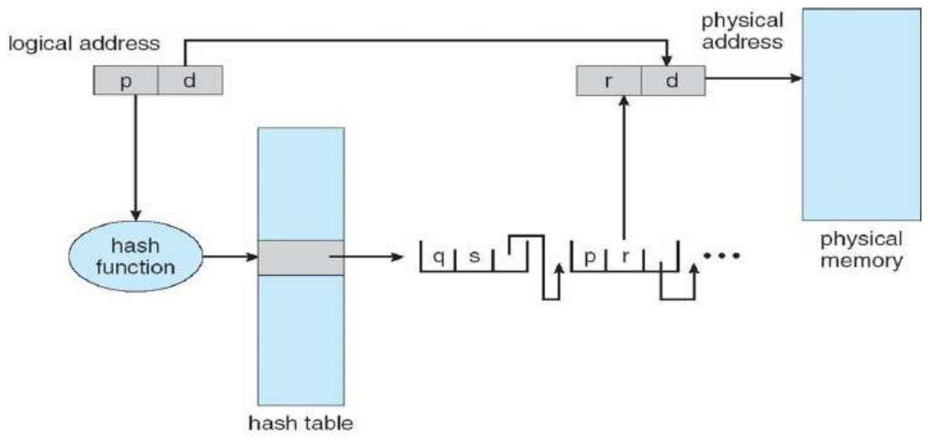
Three-level Paging Scheme



Hashed Page Tables

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

Hashed Page Table



Inverted Page Table

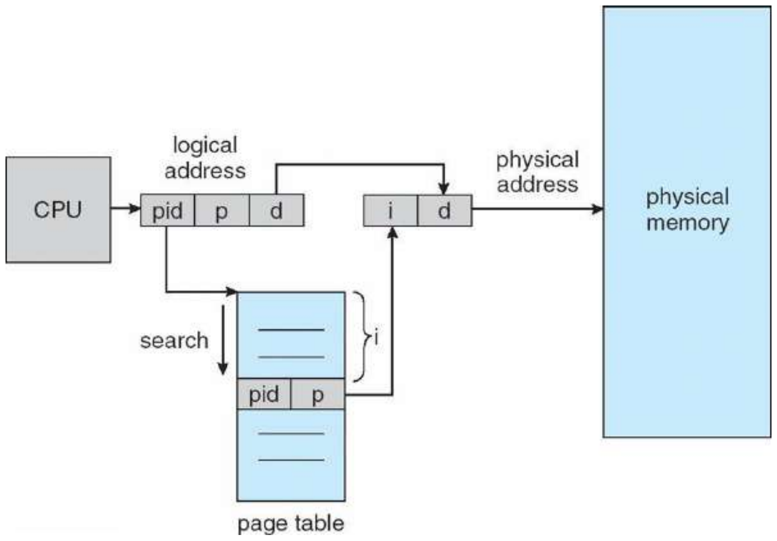
One entry for each real page of memory

Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page

Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs

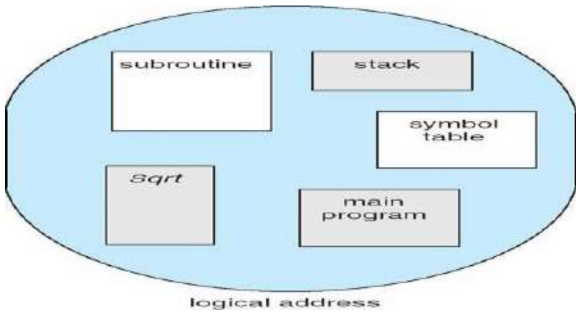
Use hash table to limit the search to one — or at most a few — page-table entries

Inverted Page Table Architecture



Segmentation

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



User's View of a Program

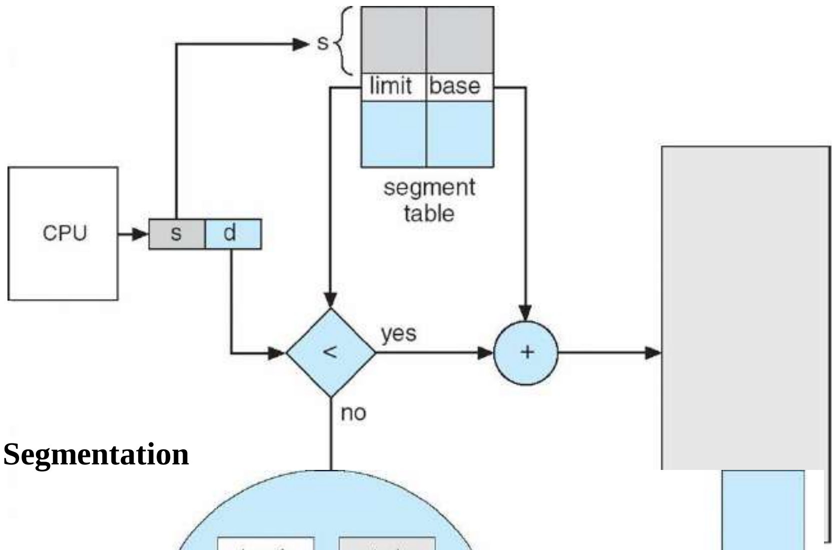
Segmentation Architecture

Logical address consists of a two tuple:
<segment-number, offset>,
Segment table – maps two-dimensional physical addresses physical; each table memory entry has: space
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**
Protection
With each entry in segment table associate: Validation bit=0 illegal segment
Reaad/write/execute privileges
potection bits associated with segments; code sharing occurs at segment level

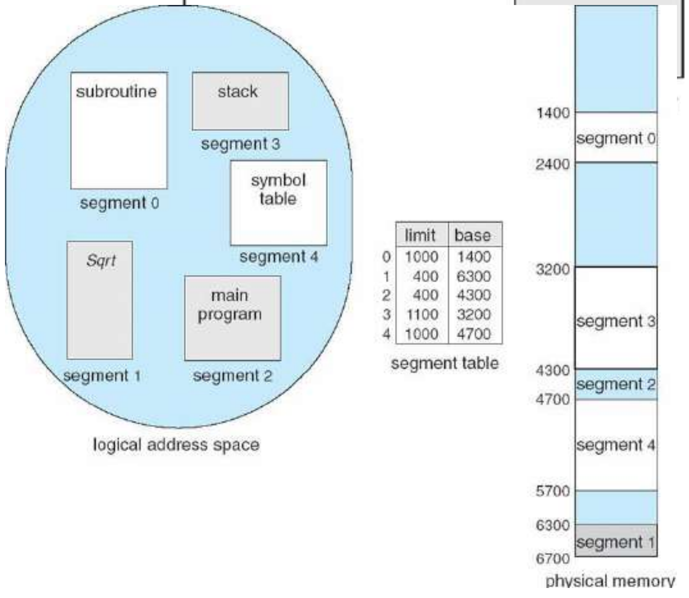
Since segments vary in length, memory allocation is a dynamic storage-allocation problem

A segmentation example is shown in the following diagram

Segmentation Hardware



Example of Segmentation



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

Example: The Intel Pentium

Supports both segmentation and segmentation with paging

CPU generates logical address

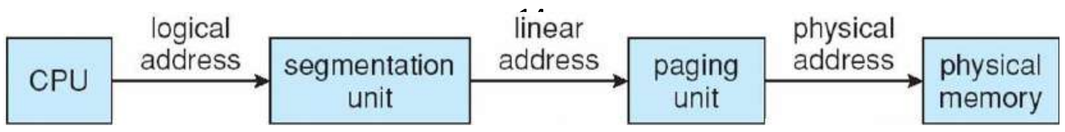
Given to segmentation unit which produces linear addresses

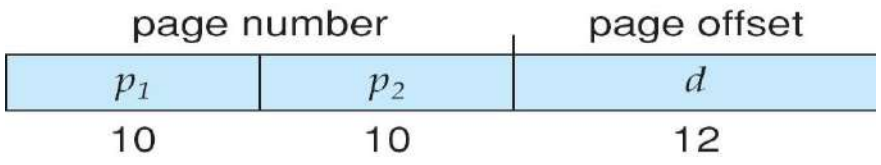
Linear address given to paging unit

Which generates physical address in main memory

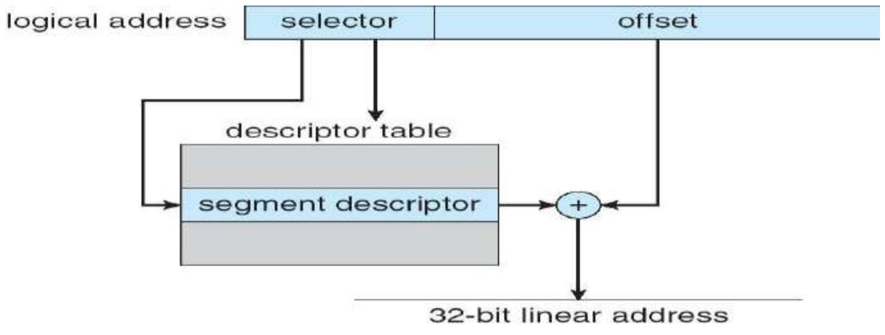
Paging units form equivalent of MMU

Logical to Physical Address Translation in Pentium

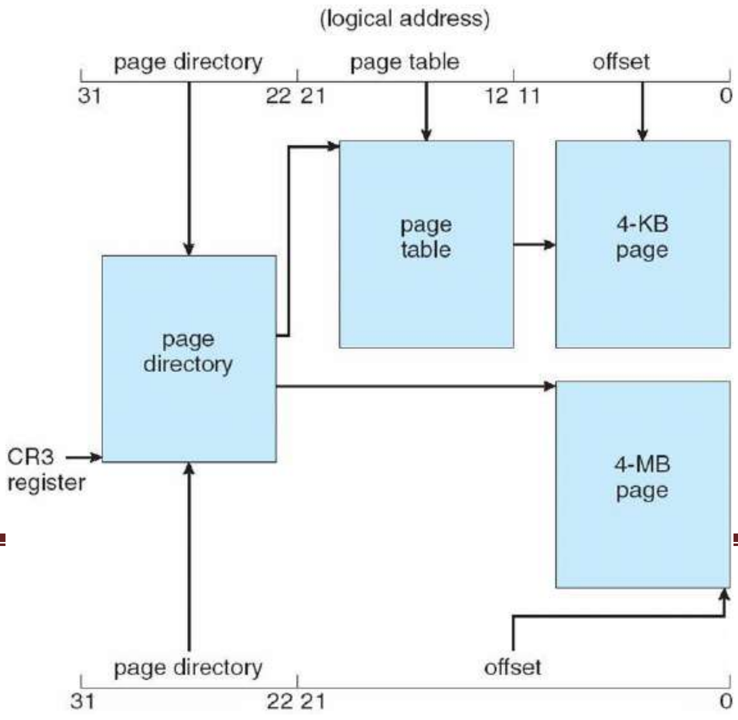




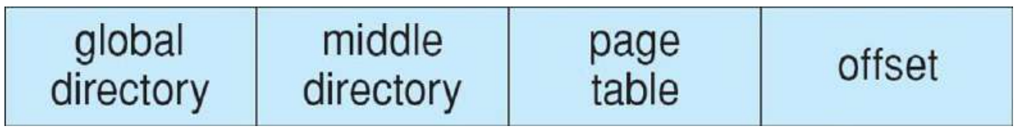
Intel Pentium Segmentation



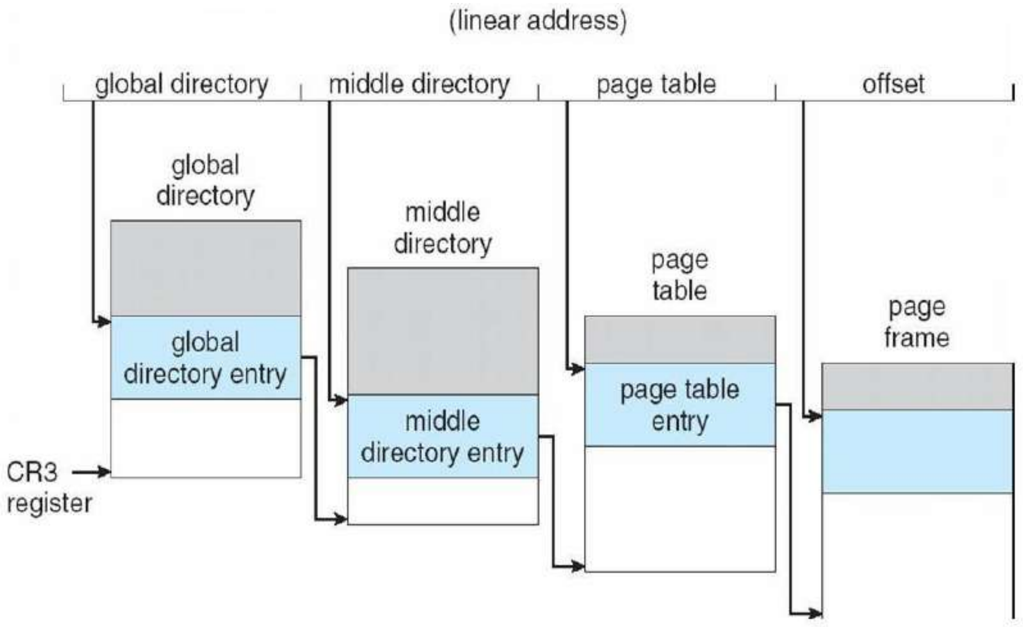
Pentium Paging Architecture



Linear Address in Linux



Three-level Paging in Linux



Part-2 - VIRTUAL MEMORY

Objective

- To describe the benefits of a virtual memory system.
- To explain the concepts of demand paging, page-replacement algorithms, and allocation of page frames.
- To discuss the principle of the working-set model.

Virtual Memory

Virtual memory is a technique that allows the execution of process that may not be completely in memory. The main visible advantage of this scheme is that programs can be larger than physical memory.

- Virtual memory is the separation of user logical memory from physical memory this separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available (Fig).

Following are the situations, when entire program is not required to load fully.

1. User written error handling routines are used only when an error occurs in the data or computation.
2. Certain options and features of a program may be used rarely.
3. Many tables are assigned a fixed amount of address space even though only a small amount of the table is actually used.

The ability to execute a program that is only partially in memory would counter many benefits.

1. Less number of I/O would be needed to load or swap each user program into memory.
2. A program would no longer be constrained by the amount of physical memory that is available.
3. Each user program could take less physical memory, more programs could be run the same time, with a corresponding increase in CPU utilization and throughput.

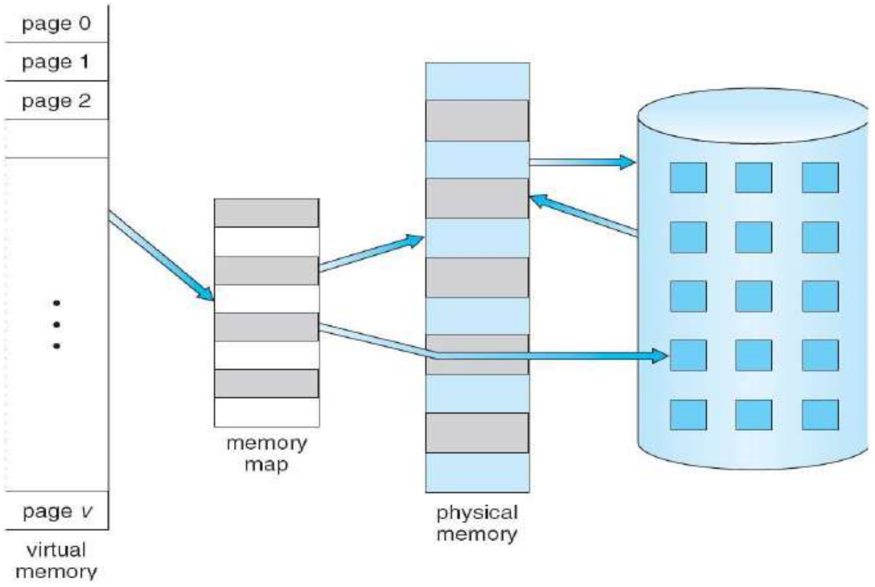


Fig. Diagram showing virtual memory that is larger than physical memory.

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

Demand Paging

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

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

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

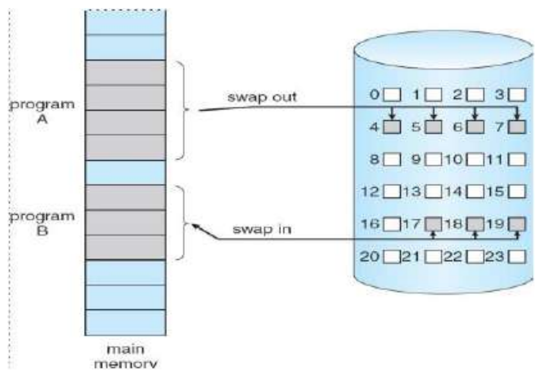


Fig. Transfer of a paged memory to continuous disk space

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

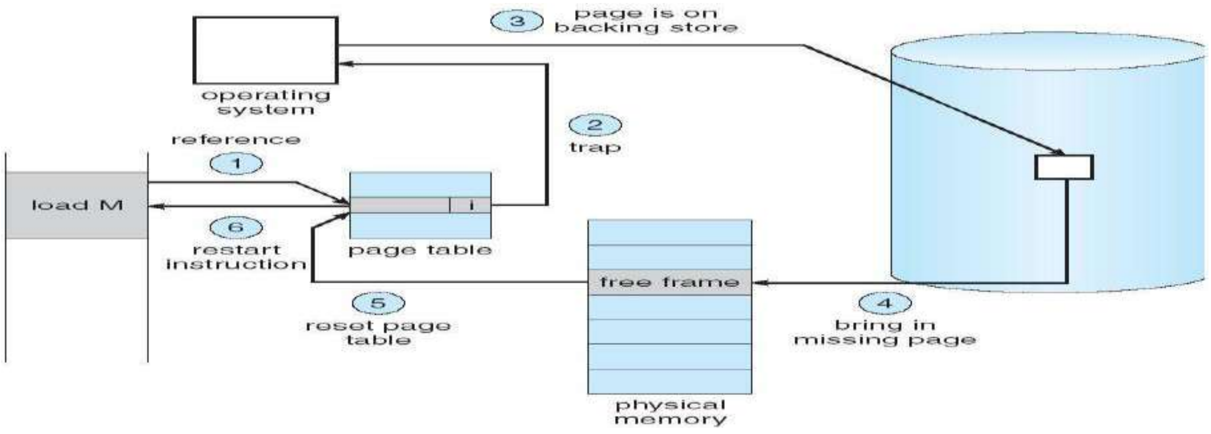


Fig. Steps in handling a page fault

We check an internal table for this process to determine whether the reference was a valid or invalid memory access.

1. If the reference was invalid, we terminate the process. If .it was valid, but we have not yet brought in that page, we now page in the latter.
2. We find a free frame.

3. We schedule a disk operation to read the desired page into the newly allocated frame.

4. When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.

5. We restart the instruction that was interrupted by the illegal address trap. The process can now access the page as though it had always been memory.

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

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

Advantages of Demand Paging:

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

Disadvantages of Demand Paging:

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

Process Creation

Virtual memory enhances the performance of creating and running processes:
- Copy-on-Write
fork() creates a child process as a duplicate of the parent process & it worked by creating copy of the parent address space for child, duplicating the pages belonging to the parent. Copy-on-Write (COW) allows both parent and child processes to initially
Share the same pages in memory. These shared pages are marked as Copy-on-Write pages, meaning that if either process modifies a shared page, a copy of the shared page is created.

vfork():

With this the parent process is suspended & the child process uses the address space of the parent.

Because vfork() does not use Copy-on-Write, if the child process changes any pages of the parent‘s address space, the altered pages will be visible to the parent once it resumes. Therefore, vfork() must be used with cauti on, ensuring that the child process does not modify the address space of the parent.

Page Replacement Algorithm

- ✓ If no frames are free, we could find one that is not currently being used & free it.
- ✓ We can free a frame by writing its contents to swap space & changing the page table to indicate that the page is no longer in memory.
- ✓ Then we can use that freed frame to hold the page for which the process faulted.

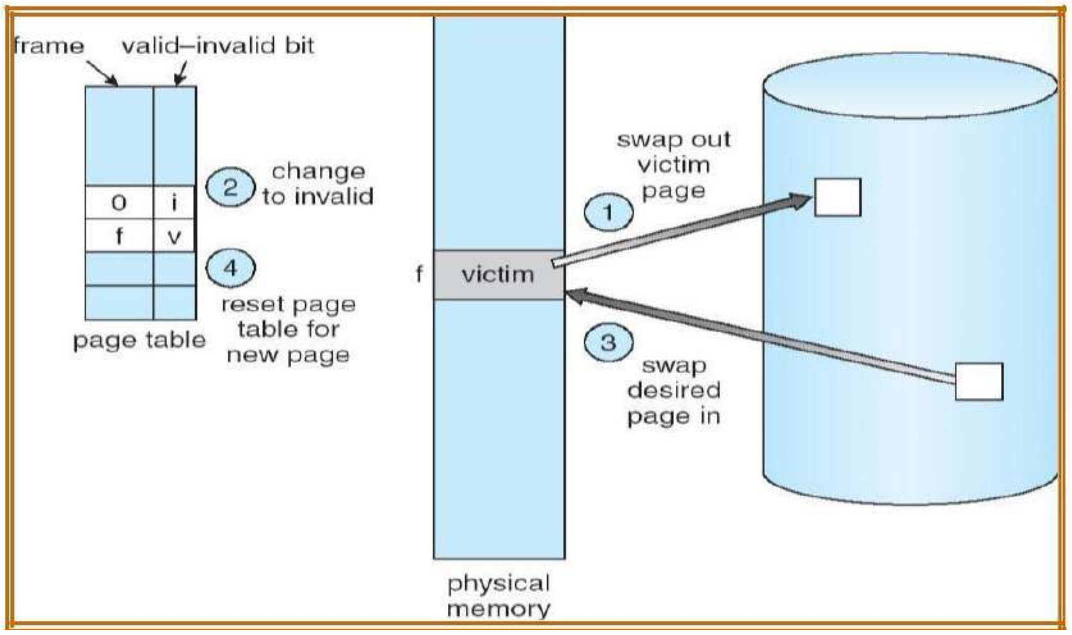
Basic Page Replacement

1. Find the location of the desired page on disk
2. Find a free frame
- If there is a free frame, then use it.
- If there is no free frame, use a page replacement algorithm to select a **victim** frame
- Write the victim page to the disk, change the page & frame tables accordingly.
3. Read the desired page into the (new) free frame. Update the page and frame tables.
4. Restart the process

If no frames are free, two page transfers are required & this situation effectively doubles the page- fault service time.

Modify (dirty) bit:

✓ It indicates that any word or byte in the page is modified.
✓ When we select a page for replacement, we examine its modify bit.
If the bit is set, we know that the page has been modified & in this case we must write that page to the disk. If the bit is not set, then if the copy of the page on the disk has not been overwritten, then we can avoid writing the memory page on the disk as it is already there.



Page Replacement Algorithms

1. FIFO Page Replacement

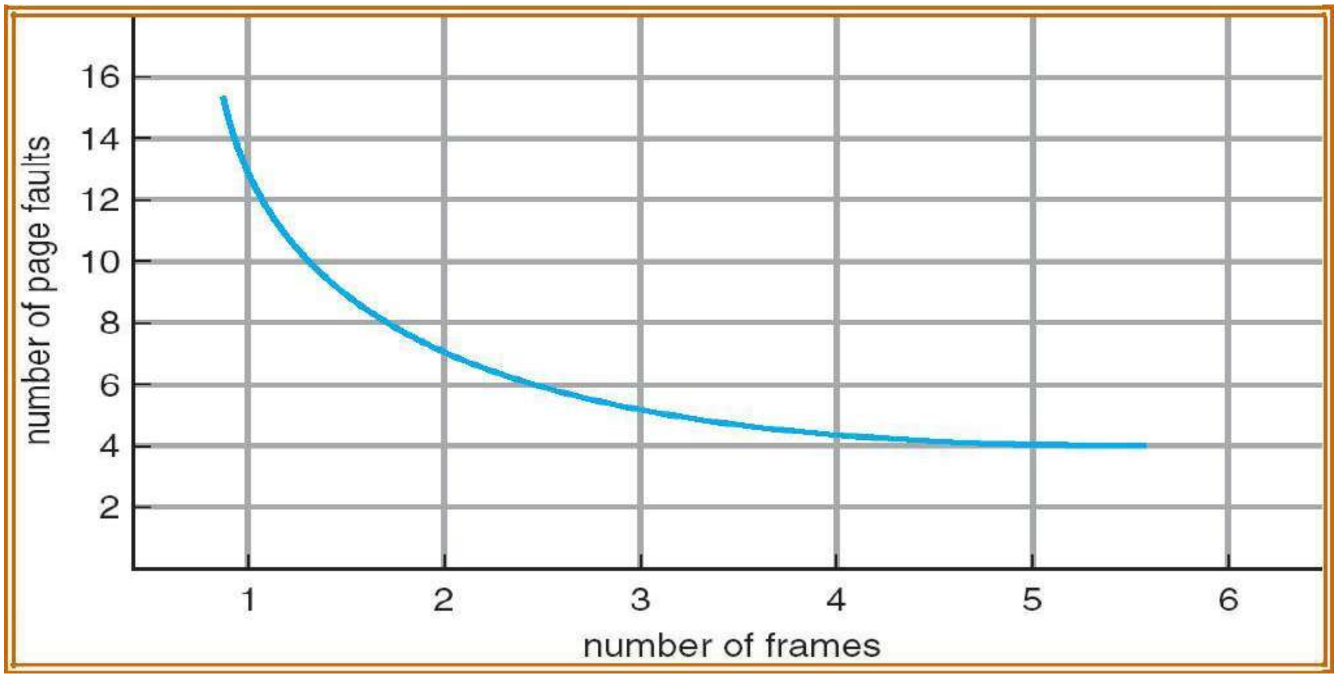
2. Optimal Page Replacement

3. LRU Page Replacement

4. LRU Approximation Page Replacement

5. Counting-Based Page Replacement

We evaluate an algorithm by running it on a particular string of memory references & computing the number of page faults. The string of memory reference is called a ―reference string‖.The algorithm that provides less number of page faults is termed to be a good one



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

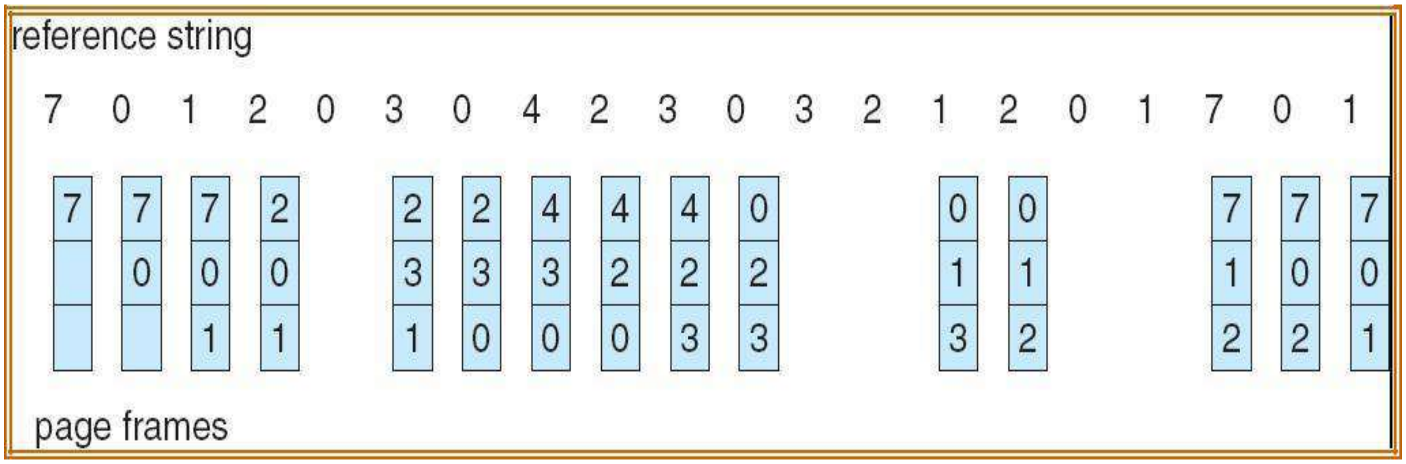
(a) FIFO page replacement algorithm

- ✓ **Replace the oldest page.**
- ✓ This algorithm associates with each page, the time when that page was brought in.

Example:

Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1

No.of available frames = 3 (3 pages can be in memory at a time per process)



No. of page faults = 15.

Drawback:

- ✓ FIFO page replacement algorithm s performance is not always good.
- ✓ To illustrate this, consider the following example:
  - o **Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**
- ✓ If No. of available frames -= 3 then the no. of page faults =9
- ✓ If No. of available frames =4 then the no. of page faults =10
- ✓ Here the no. of page faults increases when the no. of frames increases .This is called as **Belady's Anomaly.**

(b) Optimal page replacement algorithm

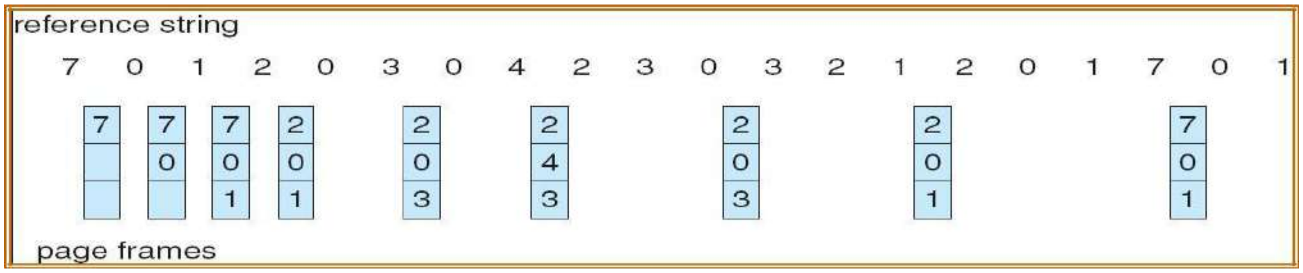
- ✓ **Replace the page that will not be used for the longest period of time. Example:**
- ✓ **Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1**
- ✓ No. of available frames = 3

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

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

Now consider the same string with 3 empty frames.

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



No. of page faults = 9

Drawback:

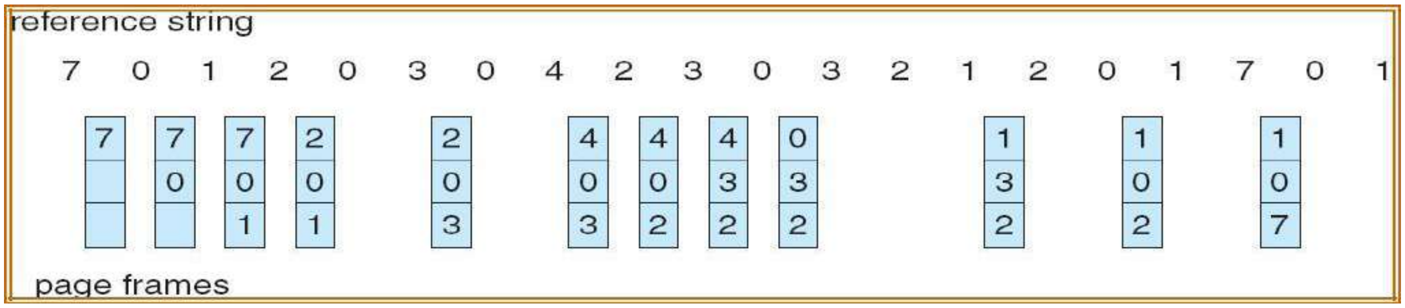
It is difficult to implement as it requires future knowledge of the reference string.

(c) LRU(Least Recently Used) page replacement algorithm

- ✓ **Replace the page that has not been used for the longest period of time.**
- ✓ **Example:**

Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1

No.of available frames = 3



No. of page faults = 12

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

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

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

- **LRU page replacement can be implemented using**

1. Counters

Every page table entry has a time-of-use field and a clock or counter is associated with the CPU.

The counter or clock is incremented for every memory reference.

Each time a page is referenced, copy the counter into the time-of-use field. When a page needs to be replaced, replace the page with the smallest counter value.

2. Stack

Keep a stack of page numbers whenever a page is referenced, remove the page from the stack and put it on

top of the stack. When a page needs to be replaced, replace the page that is at the bottom of the stack.(LRU page)

Use of A Stack to Record The Most Recent Page References

(d) LRU Approximation Page Replacement

- ✓ Reference bit
  - With each page associate a reference bit, initially set to 0
  - When page is referenced, the bit is set to 1
- ✓ When a page needs to be replaced, replace the page whose reference bit is 0

- ✓ The order of use is not known, but we know which pages were used and which were not used.

(i) Additional Reference Bits Algorithm

- ✓ Keep an 8-bit byte for each page in a table in memory.
- ✓ At regular intervals , a timer interrupt transfers control to OS.
- ✓ The OS shifts reference bit for each page into higher- order bit shifting the other bits right 1 bit and discarding the lower-order bit.

Example:

- ✓ If reference bit is 00000000 then the page has not been used for 8 time periods.
- ✓ If reference bit is 11111111 then the page has been used atleast once each time period.
- ✓ If the reference bit of page 1 is 11000100 and page 2 is 0111011 then page 2 is the LRU page.

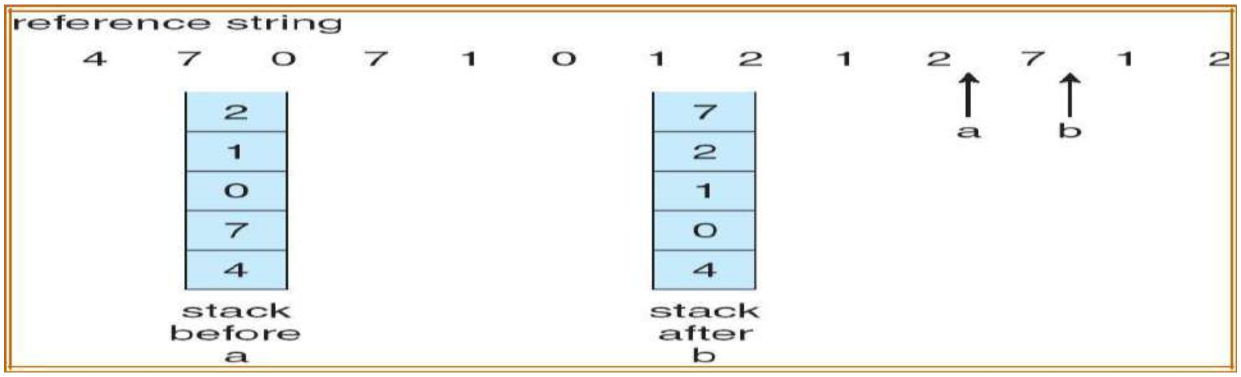


Fig: Use of a stack to record the most recent page references.

(ii) Second Chance Algorithm

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

- Basic algorithm is FIFO
- When a page has been selected, check its reference bit.
    - If 0 proceed to replace the page

- If 1 give the page a second chance and move on to the next FIFO page.

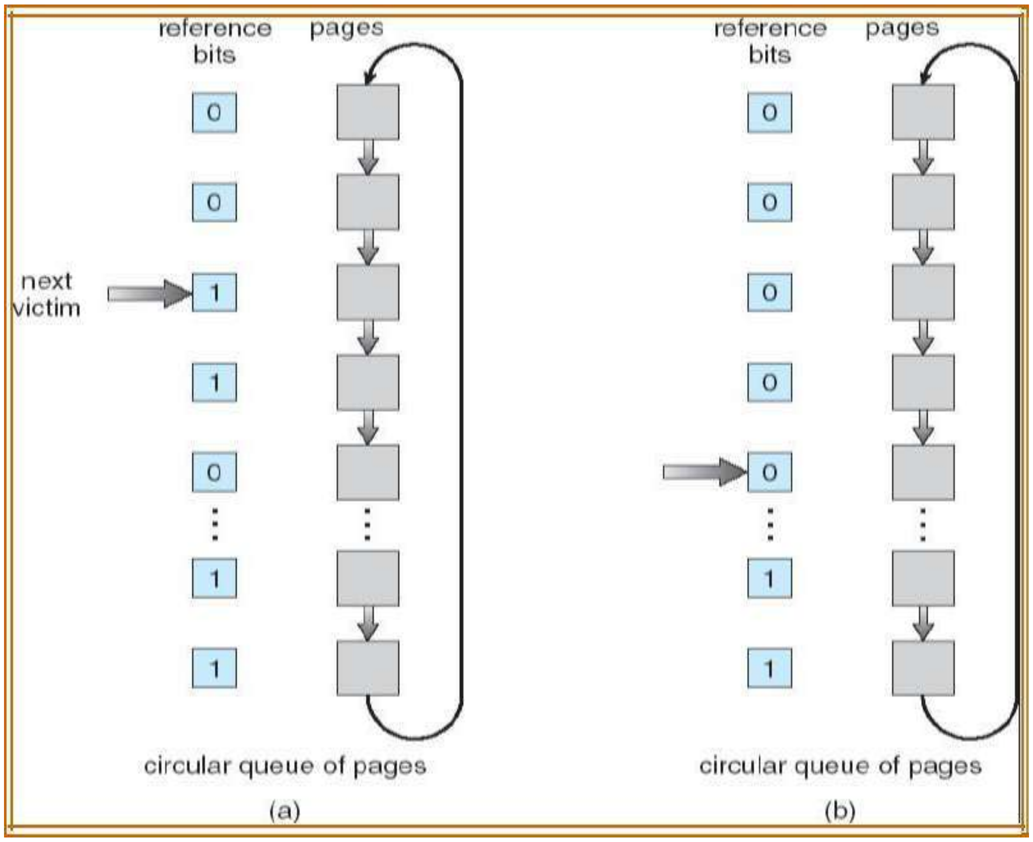


Fig: Second-chance (clock) page- replacement algorithm.

When a page gets a second chance, its reference bit is cleared and arrival time is reset to current time.

Hence a second chance page will not be replaced until all other pages are replaced.

(iii) Enhanced Second Chance Algorithm:

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

- Consider both reference bit and modify bit
- There are four possible classes

1. (0,0) – neither recently used nor modified Best page to replace
2. (0,1) – not recently used but modified page has to be written out before replacement.
3. (1,0) - recently used but not modified page may be used again

4. (1,1) – recently used and modified page may be used again and page has to be written to disk

(e) Counting-Based Page Replacement

Keep a counter of the number of references that have been made to each page

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

2. Most Frequently Used (MFU) Algorithm: replaces page with largest count It is based on the argument that the page with the smallest count was probably just brought in and has yet to be used

Page Buffering Algorithm

When a page fault occurs, a victim frame is chosen as before. However, the desired page is read into a free frame from the pool before the victim is written out.
This procedure allows the process to restart as soon as possible, without waiting for the victim page to be written out. When the victim is later written out, its frame is added to the free-frame pool.
When the FIFO replacement algorithm mistakenly replaces a page mistakenly replaces a page that is still in active use, that page is quickly retrieved from the free-frame buffer, and no I/O is necessary. The free-frame buffer provides protection against the relatively poor, but simple, FIFO replacement algorithm.

These are used along with page replacement algorithms to improve their performance

Technique 1:

- A pool of free frames is kept.
- When a page fault occurs, choose a victim frame as before.
- Read the desired page into a free frame from the pool
- The victim frame is written onto the disk and then returned to the pool of
  - free frames.

Technique 2:

- Maintain a list of modified pages.
  - Whenever the paging device is idles, a modified is selected and written to disk and it's modify bit is reset.

Technique 3:

- A pool of free frames is kept.
- Remember which page was in each frame.
  - If frame contents are not modified then the old page can be reused directly from the free frame pool when needed

Allocation of Frames

There are two major allocation schemes

Equal Allocation, Proportional Allocation

Equal allocation

If there are n processes and m frames then allocate m/n frames to each process.

Example: If there are 5 processes and 100 frames, give each process 20 frames.

Proportional allocation

Allocate according to the size of process

Let si be the size of process i. Let m be the total no. of frames

Then S = ∑ si ai = si / S * m

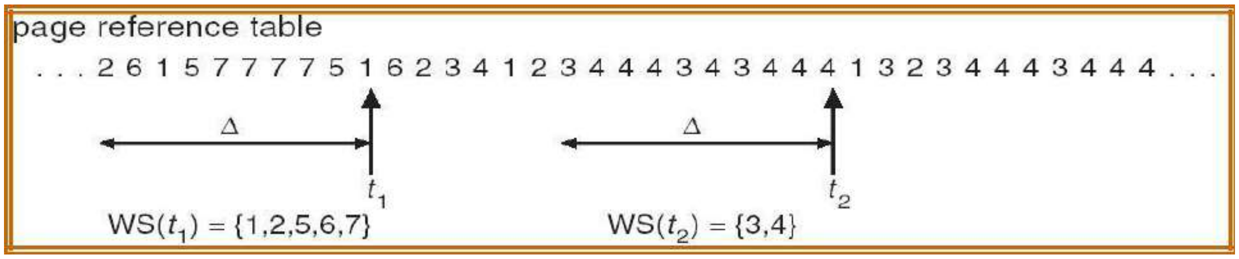
Where ai is the no. of frames allocated to process i.

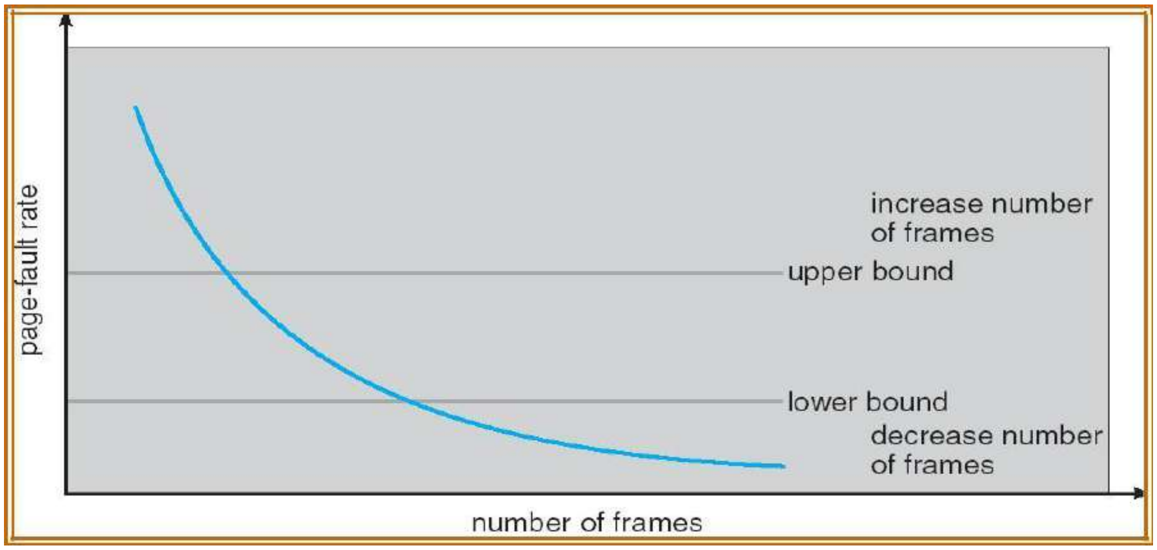
Global vs. Local Replacement

- **Global replacement** – each 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

- High paging activity is called **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 is added to the system
- When the CPU utilization is low, the OS increases the degree of multiprogramming.
- If global replacement is used then as processes enter the main memory they tend to steal frames belonging to other processes.
  - Eventually all processes will not have enough frames and hence the page fault rate becomes very high.
- Thus swapping in and swapping out of pages only takes place.
- This is the cause of thrashing.
- To **limit thrashing**, we can use a **local replacement** algorithm.
- To prevent thrashing, there are two methods namely , Working Set Strategy





Page Fault Frequency

1. Working-Set Strategy

- It is based on the assumption of the model of locality.
- Locality is defined as the set of pages actively used together.
- Working set is the set of pages in the most recent page references is the working set window.

  if too small , it will not encompass entire locality if too large ,it will encompass several localities ,if it will encompass entire program *D* = *WSSi* Where WSSi is the working set size for process i. D is the total demand of frames , if *D* > *m* then Thrashing will occur.

2. Page-Fault Frequency Scheme

If actual rate too low, process loses frame
If actual rate too high, process gains frame

Prepaging

To reduce the large number of page faults that occurs at process startup prepage all or some of the pages a process will need, before they are referenced .But if prepaged pages are unused, I/O and memory are wasted.

Page Size

Page size selection must take into consideration:

- fragmentation table size
- I/O overhead
- locality

TLB Reach

TLB Reach - The amount of memory accessible from the TLB
TLB Reach = (TLB Size) X (Page Size)
Ideally, the working set of each process is stored in the TLB. Otherwise there is a high degree of page faults. Increase the Page Size. This may lead to an increase in fragmentation as not all applications require a large page size Provide Multiple Page Sizes. This allows applications that require largerpage sizes the opportunity to use them without an increase infragmentation.

I/O interlock

Pages must sometimes be locked into memory

Consider I/O. Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm.

Unit-III

Part-3 - Storage Management

The file system is the most visible aspect of an OS. It provides the mechanism for online storage of and access to both data and programs of OS and all the users of the computer system. The file system consists of two distinct parts: a collection of files – each storing related data and a directory structure which organizes and provides information about all the files in the system.

File Concept

Computers can store information on various storage media such as magnetic disks, magnetic tapes and optical disks. OS provides a uniform logical view of information storage. OS abstracts from the physical properties of its storage devices to define a logical storage unit called a **file**. Files are mapped by OS onto physical devices. These storage devices are non volatile so the contents are persistent through power failures and system reboots.

A file is a named collection of related information that is recorded on secondary storage. A file is the smallest allotment of logical secondary storage; that is data cannot be written to secondary storage unless they are within a file. Files represent programs and data. Data files may be numeric, alphabetic, alphanumeric or binary. Files may be free form such as text files or may be formatted rigidly. A file is a sequence of bits, bytes, lines or records.

Information in a file is defined by its creator. Many different types of information may be stored in a file – source programs, object programs, executable programs, numeric data, text etc. A file has a certain defined structure which depends on its type.

- Text file –sequence of characters organized into lines
- Source file – sequence of sub routines and functions each of which is further organized as declarations followed by executable statements.
- Object file – sequence of bytes organized into blocks understandable by the system's linker
- Executable file – series of code sections that the loader can bring into memory and execute.

<u>File Attributes</u>

A file is referred to by its name. A name is usually a string of characters. When a file is named, it becomes independent of the process, the user and even the system that created it.

A file's attributes vary from one OS to another but consist of these –

Name: symbolic file name is the only information kept in human readable form

- **Identifier:** number which identifies the file within the file system; it is the non human readable name for the file.
- **Type**: information is needed for systems that support different types of files.
- **Location**: this information is a pointer to a device and to the location of the file on that device.
- **Size:** the current size of the file
- **Protection**: Access control information determines who can do reading, writing, executing etc.
- **Time, date and user identification**: This information may be kept for creation, last modification and last use.

The information about all files is kept in the directory structure which resides on secondary storage. A directory entry consists of the file's name and its unique identifier. The identifier in turn locates the other file attributes.

<u>File Operations</u>

A file is an abstract data type. OS can provide system calls to create, write, read, reposition, delete and truncate files.

- **Creating a file** – First space in the file system must be found for the file. Second, an entry for the new file must be made in the directory.
- **Writing a file** – To write a file, specify both the name of the file and the information to be written to the file. The system must keep a write pointer to the location in the file where the next write is to take place.

- **Reading a file** – To read from a file, directory is searched for the associated entry and the system needs to keep a read pointer to the location in the file where the next read is to take place. Because a process is either reading from or writing to a file, the current operation location can be kept as a per process current file position pointer.
- **Repositioning within a file** – Directory is searched for the appropriate entry and the current file position pointer is repositioned to a given value. This operation is also known as files seek.
- **Deleting a file** – To delete a file, search the directory for the named file. When found, releases all file space and erase the directory entry.
- **Truncating a file** – User may want to erase the contents of a file but keep its attributes. This function allows all attributes to remain unchanged except for file length.

Other common operations include appending new information to the end of an existing file and renaming an existing file. We may also need operations that allow the user to get and set the various attributes of a file.

Most of the file operations mentioned involve searching the directory for the entry associated

call be made before a file is first used actively. OS keeps a small table called the **open file table** containing information about all open files. When a file operation is requested, the file is specified via an index into this table so no searching is required.

When the file is no longer being actively used, it is closed by the process and the OS removes its entry from the open file table. Create and delete are system calls that work with closed files.

The open () operation takes a file name and searches the directory copying the directory entry into the open file table. The open () call can also accept access mode information – create, read – only, read – write, append – only, etc. This mode is checked against file's permissions. If the request mode is allowed, the file is opened for the process. The open () system call returns a pointer to the entry in the open file table. This pointer is used in all I/O operations avoiding any further searching and simplifying the system call interface.

OS uses two levels of internal tables – a per process table and a system wide table. The per process table tracks all files that a process has open. Stored in this table is information regarding the use of the file by the process.

Each entry in the per process table points to a system wide open file table. The system wide table contains process independent information. Once a file has been opened by one process, the system wide table includes an entry for the file. The open file table also has an open count associated with each file to indicate how many processes have the file open.

To summarize, several pieces of information are associated with an open file.

- o File pointer – System must keep track of the last read – write location as a current file position pointer.

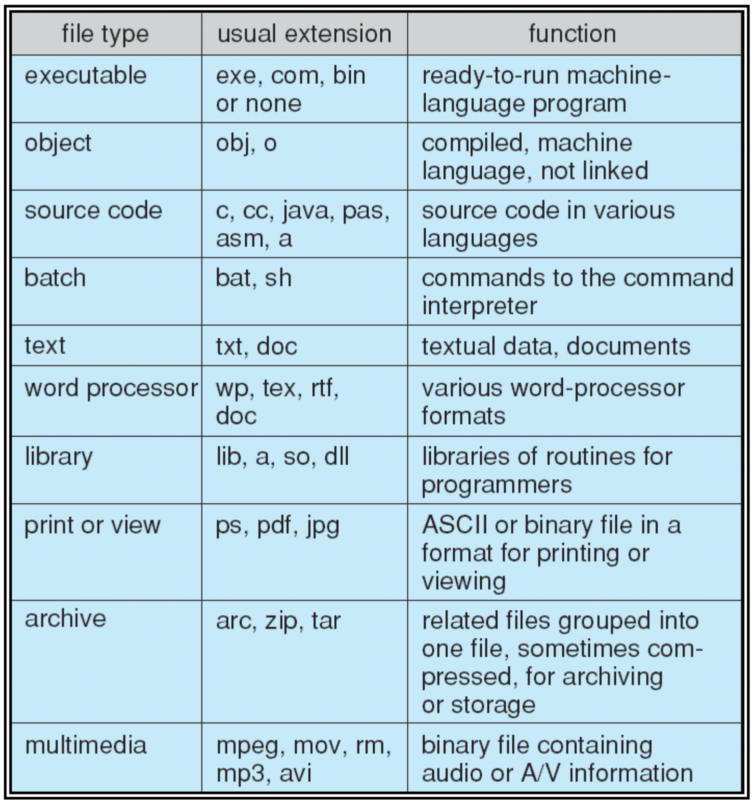
- File open count – As files are closed, OS must reuse its open file entries or it could run out of space in the table. File open counter tracks the number of opens and closes and reaches zero on the last close.
- Disk location of the file – The information needed to locate the file on disk is kept in memory so that the system does not have to read it from disk for each operation.
- Access rights – Each process opens a file a file in an access mode. This information is stored on the per process table so the OS can allow or deny subsequent I/O requests.

Some OS's provide facilities for locking an open file. File locks allow one process to lock a file and prevent other processes from gaining access to it. File locks are useful for files that are shared by several processes. A **shared lock** is where several processes can acquire the lock concurrently. An **exclusive lock** is where only one process at a time can acquire such a lock.

Also some OS's may provide either mandatory or advisory file locking mechanisms. If a lock is mandatory, then once a process acquires an exclusive lock, the OS will prevent any other process from accessing the locked file. If the lock scheme is mandatory, OS ensures locking integrity.
For advisory locking, it is upto software developers to ensure that locks are appropriately acquired and released.

File types

A common technique for implementing file types is to include the type as part of the file name. The name is split into two parts – a name and an extension separated by a period character. The system uses the extension to indicate the type of the file and the type of operations that can be done on that file.



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

Fig: Common files types

File structure

File types can be used to indicate the internal structure of the file. Source and object files have structures that match the expectations of the programs that read them. Certain files conform to a required structure that is understood by OS. But the disadvantage of having the OS support multiple file structures is that the resulting size of the OS is cumbersome. If the OS contains five different file structures, it needs to contain the code to support these file structures. Hence some OS's impose a minimal number of file structures. MAC OS also supports a minimal number of file structures. It expects files to contain two parts – a resource fork and a data fork. The resource fork contains information of interest to the user. The data fork contains program code or data – traditional file contents.

Internal file structure

Internally locating an offset within a file can be complicated for the OS. Disk systems have a well defined block size determined by the size of the sector. All disk I/O is performed in units of one block and all blocks are the same size. Since it is unlikely that the physical record size will exactly match the length of the desired logical record, and then logical records may even vary in length, packing a number of logical records into physical blocks is a solution.

The logical record size, physical block size and packing technique determine how many logical records are in each physical block. The packing can be done either by the user's application program or by the OS. Hence the file may be considered to be a sequence of blocks. All the basic I/O functions operate in terms of blocks.

Access methods

Files store information. When it is used, this information must be accessed and read into computer memory. The information in the file can be accessed in several ways.

They are –

1. Sequential access: Simplest method. Information in the file is processed in order that is one record after the other. This method is based on a tape model of a file and works as well on sequential access devices as it does on random access

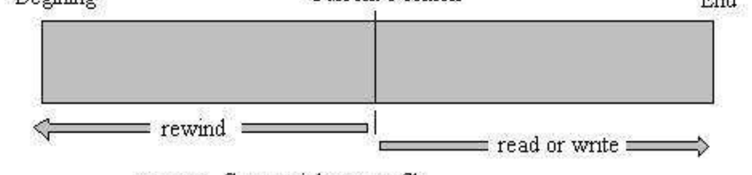
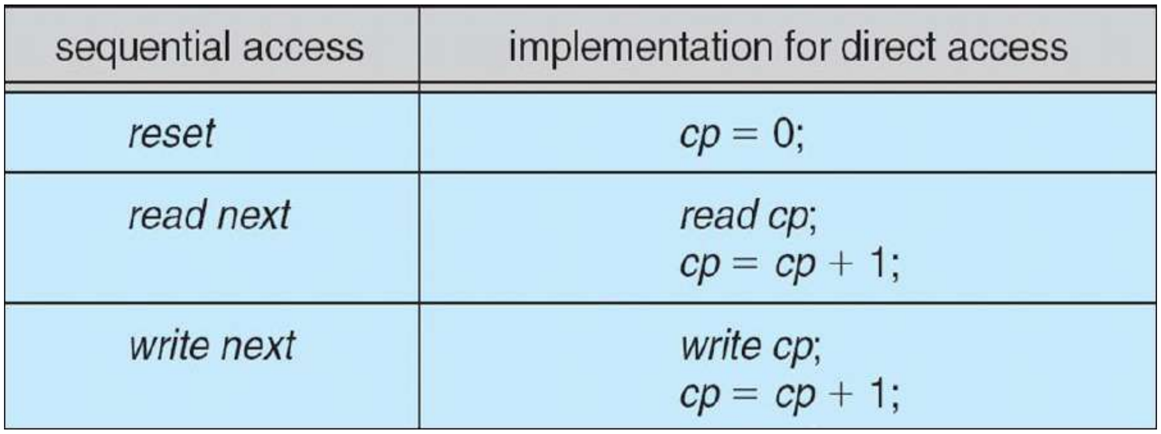


Fig 4.10 Sequential-access file

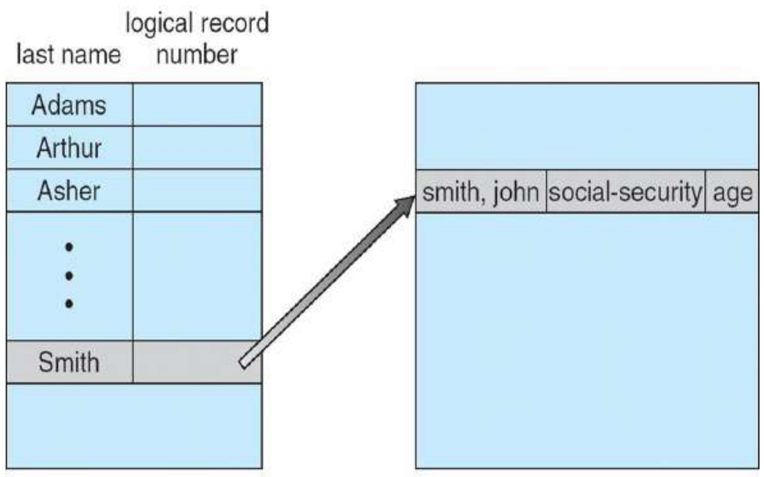
2. Direct access: Another method is direct access or relative access. A file is made up of fixed length logical records that allow programs to read and write records rapidly in no particular order. The direct access method is based on a disk model of a file since disks allow random access to any file block. Direct access files are of great use for immediate
access to large amounts of information. In this method, file operations must be modified to include block number as a parameter.
The block number provided by the user to the OS is a relative block number. A relative block number is an index relative to the beginning of the file. The use of relative block numbers allows the OS to decide where the file should be placed and helps to prevent the user from accessing portions of the file system that may not be a part of the file.
Some systems allow only sequential file access; others allow only direct access.



sequential access	implementation for direct access
reset	*cp* = 0;
read next	*read cp*; *cp* = *cp* + 1;
write next	*write cp*; *cp* = *cp* + 1;

Fig: Simulation of sequential access on a direct-access file

3. Other Access Methods: Other access methods can be built on top of a direct access method. These methods generally involve the construction of an index for the file. This index contains pointers to the various blocks. To find a record in the file, first search the index and then use the pointer to access the file directly and to find the desired record.
But with large files, the index file itself may become too large to be kept in memory. One solution is to create an index for the index file. The primary index file would contain pointers to secondary index files which would point to actual data items.



Directory Structure

Systems may have zero or more file systems and the file systems may be of varying types. Organizing millions of files involves use of directories.

Storage Structure

A disk can be used in its entirety for a file system. But at times, it is desirable to place multiple file systems on a disk or to use parts of a disk for a file system and other parts for other things. These parts are known variously as **partitions, slices or minidisks**. A file system can be created on each of these parts of the disk. These parts can be combined together to form larger structures known as **volumes** and file systems can be created on these too. Each volume can be thought of as a virtual disk. Volumes can also store multiple OS's allowing a system to boot and run more than one.

Each volume that contains a file system must also contain information about the files in the system. This information is kept in entries in a **device directory or volume table of contents**. The device directory/directory records information for all files on that volume

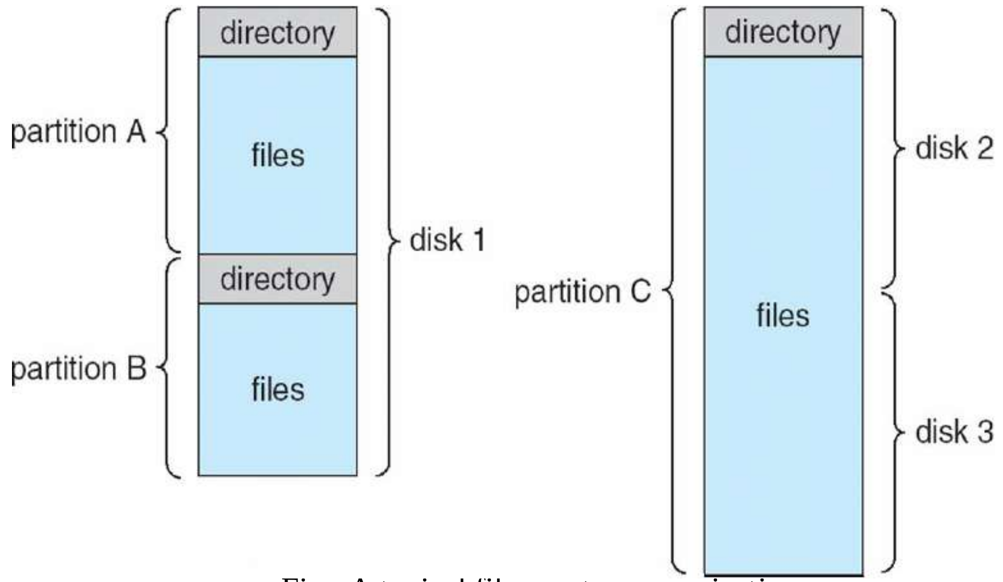


Fig : A typical file –system organization.

Directory Overview

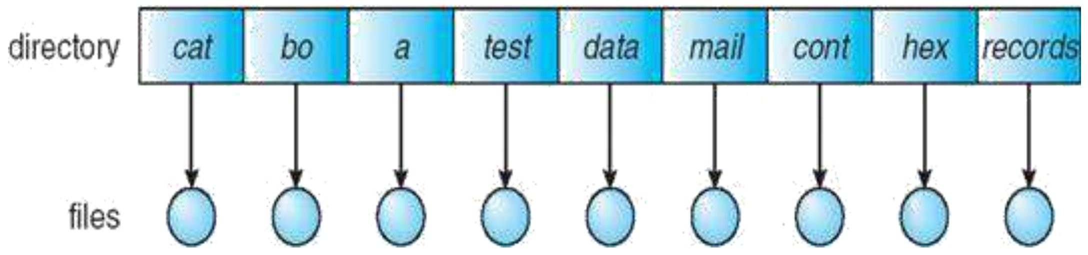
The directory can be viewed as a symbol table that translates file names into their directory entries. The operations that can be performed on the directory are:

- ✓ Search for a file
- ✓ Create a file

- ✓ Delete a file
- ✓ List a directory
- ✓ Rename a file
- ✓ Traverse the file system

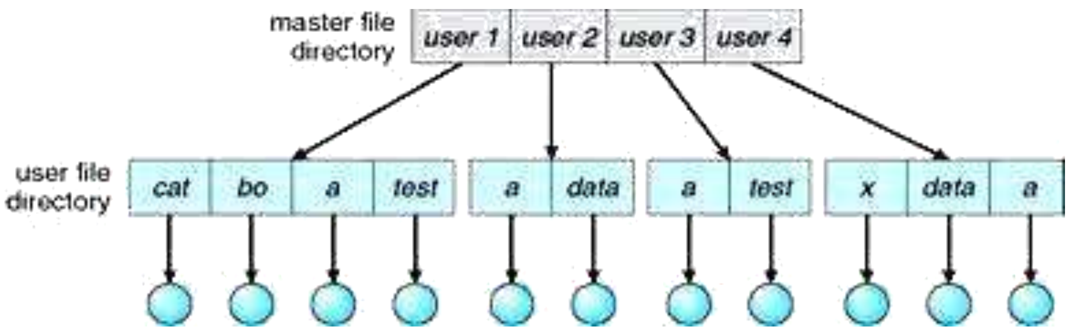
Single level directory

The simplest directory structure is the single level directory. All files are contained in the same directory which is easy to support and understand. But this implementation has limitations when the number of files increases or when the system has more than one user. Since all files are in same directory, all files names must be unique. Keeping track of so many files is a difficult task. A single user on a single level directory may find it difficult to remember the names of all the files as the number of files increases.



Two level directory

In the two level directory structure, each user has his own **user file directory** (UFD). The UFD's have similar structures but each lists only the files of a single user. When a user job starts or a user logs in, the system's **master file directory** (MFD) is searched. The MFD is indexed by user name or account number and each entry points to the UFD for that user. When a user refers to a particular file, only his own UFD is searched. Different users may have files with the same name as long as all the files names within each UFD are unique.



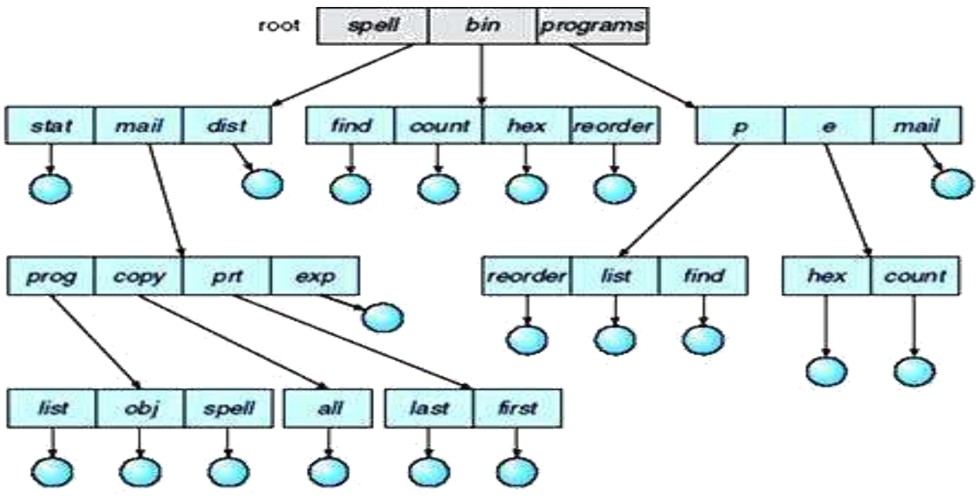
Root of the tree is MFD. Its direct descendants are UFDs. The descendants of the UFDs are the files themselves. The files are the leaves of the tree.

The sequence of directories searched when a file is names is called the **search path**.

Although the two level directory structure solves the name collision problem, it still has disadvantages. This structure isolates on user from another. Isolation is an advantage when the users are completely independent but a disadvantage when the users want to cooperate on some task and to access one another's files.

Tree Structured Directories

Here, we extend the two level directory to a tree of arbitrary height. This generalization allows users to create their own subdirectories and to organize their files accordingly. A tree is the most common directory structure. The tree has a root directory and every file in the system has a unique path name. A directory contains a set of files or sub directories. All directories have the same internal format. One bit in each directory entry defines the entry as a file (0) or as a subdirectory (1).



Each process has a current directory. The current directory should contain most of the files that are of current interest to the process.

Path names can be of two types – absolute and relative. An absolute path name begins at the root and follows a path down to the specified file giving the directory names on the path. A relative path name defines a path from the current directory.

Deletion of directory under tree structured directory – If a directory is empty, its entry in the directory that contains it can simply be deleted. If the directory to be deleted is not empty, then use one of the two approaches –

User must first delete all the files in that directory

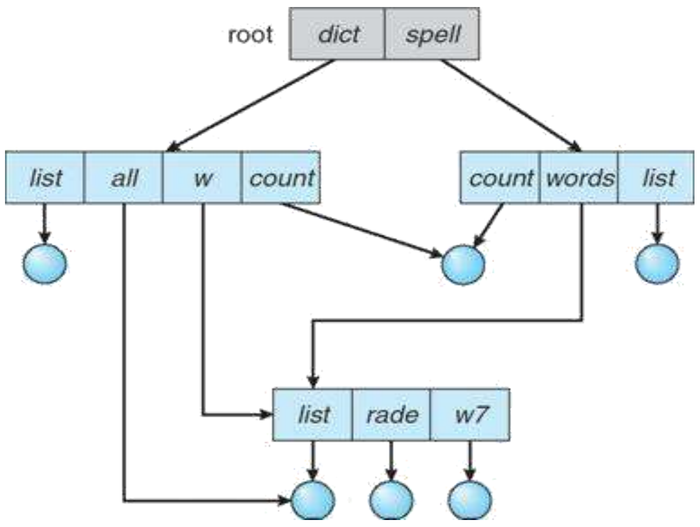
If a request is made to delete a directory, all the directory's files and sub directories are also to be deleted.

A path to a file in a tree structured directory can be longer than a path in a two level directory.

Acyclic graph directories

A tree structure prohibits the sharing of files and directories. An acyclic graph i.e. a graph with no cycles allows directories to share subdirectories and files. The same file or subdirectory may

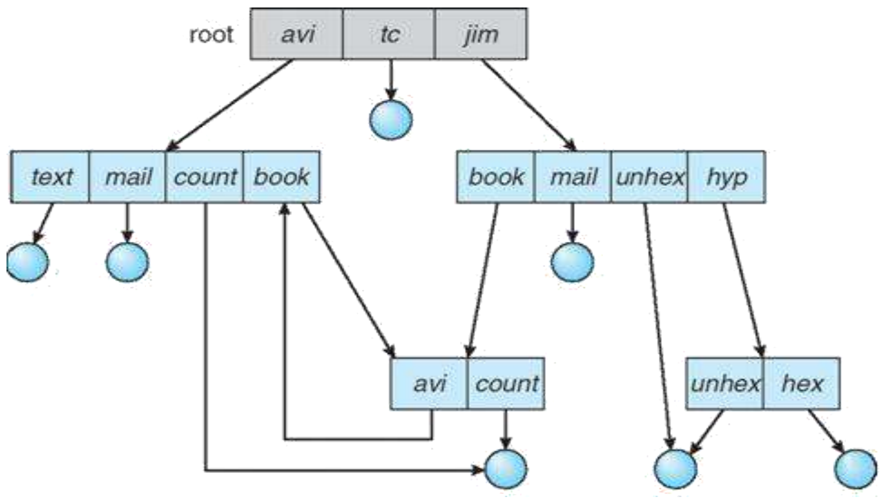
With a shared file, only one actual file exists. Sharing is particularly important for subdirectories. Shared files and subdirectories can be implemented in several ways. One way is to create a new directory entry called a link. A link is a pointer to another file or subdirectory. Another approach in implementing shared files is to duplicate all information about them in both sharing directories.



An acyclic graph directory structure is flexible than a tree structure but it is more complex. Several problems may exist such as multiple absolute path names or deletion.

General graph directory

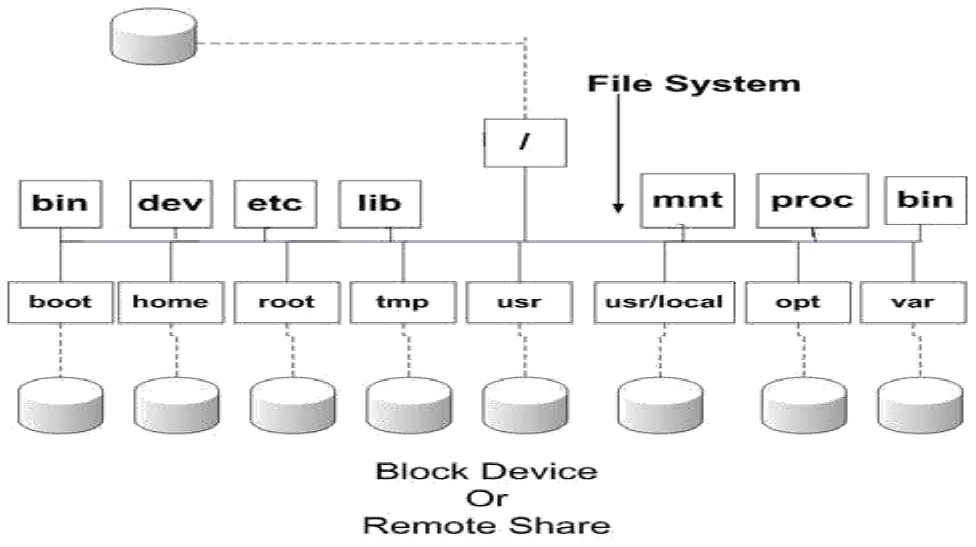
A problem with using an acyclic graph structure is ensuring that there are no cycles.



The primary advantage of an acyclic graph is the relative simplicity of the algorithms to traverse the graph and to determine when there are no more references to a file. If cycles are allowed to exist in the directory, avoid searching any component twice. A similar problem exists when we are trying to determine when a file can be deleted. The difficulty is to avoid cycles as new links are added to the structure.

File System Mounting

A file system must be mounted before it can be available to processes on the system. OS is given the name of the device and a mount point – the location within the file structure where the file system is to be attached. This mount point is an empty directory. Next, OS verifies that the device contains a valid file system. It does so by asking the device driver to read the device directory and verifying that the directory has the expected format. Finally OS notes in its directory structure that a file system is mounted at the specified mount point.



File Sharing

File sharing is desirable for users who want to collaborate and to reduce the effort required to achieve a computing goal.

Multiple users

When an OS accommodates multiple users, the issues of file sharing, file naming and file protection become preeminent. System mediates file sharing. The system can either allow a user to access the files of other users by default or require that a user specifically grant access to the files.

Remote File Systems

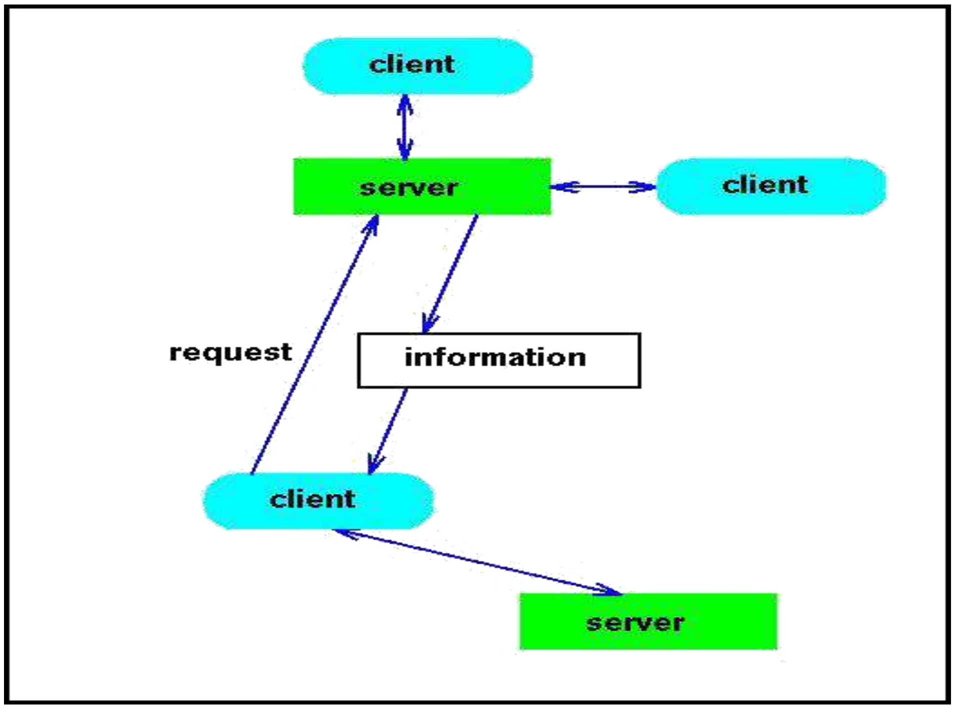
Networking allows sharing of resources spread across a campus or even around the world. One obvious resource to share is data in the form of files.

The first implemented file sharing is method involves manually transferring files between machines via programs like ftp. The second major method uses a distributed file system in which remote directories are visible from a local machine. The third method is through WWW.

ftp is used for both anonymous and authenticated access. Anonymous access allows a user to transfer files without having an account on the remote system. WWW uses anonymous files exchange almost exclusively. DFS involves a much tighter integration between the machine that is accessing the remote files and the machine providing the files.

Client Server Model

Remote file systems allow a computer to mount one or more file systems from one or more remote machines. Here the machine containing the files is the server and the machine seeking access to the files is the client. A server can serve multiple clients and a client can use multiple servers depending on the implementation details of a given client server facility. Once the remote file system is mounted, file operation requests are sent on behalf of the user across the network to the server via the DFS protocol.



Distributed Information Systems

To make client server systems easier to manage, distributed information systems also known as distributed naming services provide unified access to the information needed for remote computing. The domain name system provides host name to network address translations for the entire Internet.

Distributed information systems used by some companies –

Sun Microsystems – Network Information Service or NIS

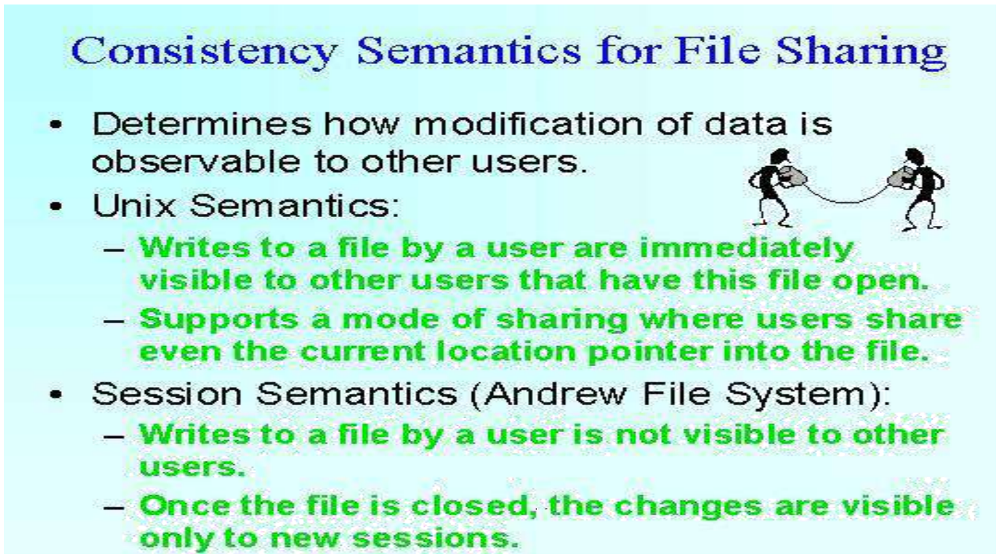
Microsoft – Common internet file system or CIFS

Failure Modes

Local file systems can fail for a variety of reasons including failure of the disk containing the file system, corruption of the delivery structure or other disk management information, disk controller failure, cable failure and host adapter failure. User or system administrator failure can also cause files to be lost or entire directories or volumes to be deleted. Many of these failures will cause a host to crash and an error condition to be displayed and human intervention will be required to repair the damage.
Remote fail systems have even more failure modes. In the case of networks, the network can be interrupted between two hosts. Such interruption can result from hardware failure, poor hardware configuration or networking implementation issues.
For a recovery from a failure, some kind of state information may be maintained on both the client and server.

Consistency semantics

These represent an important criterion for evaluating any file system that supports file sharing. These semantics specify how multiple users of a system are to access a shared file simultaneously. These are typically implemented as code with the file system.



Consistency Semantics for File Sharing

- Determines how modification of data is observable to other users.
- Unix Semantics:
  - Writes to a file by a user are immediately visible to other users that have this file open.
  - Supports a mode of sharing where users share even the current location pointer into the file.
- Session Semantics (Andrew File System):
  - Writes to a file by a user is not visible to other users.
  - Once the file is closed, the changes are visible only to new sessions.

Protection

When information is stored in a computer system, it should be kept safe from physical damage (reliability) and improper access (protection).

Reliability is provided by duplicate copies of files.

Protection can be provided in many ways such as physically removing the floppy disks and locking them up.

Types of Access

Complete protection to files can be provided by prohibiting access. Systems that do not permit access to the files of other users do not need protection. Both these approaches are extreme. Hence **controlled access** is required.

Protection mechanisms provide controlled access by limiting the types of file access that can be made. Access is permitted or denied depending on many factors. Several different types of operations may be controlled –

i. Read
ii. Write
iii. Execute
iv. Append
v. Delete
vi. List

Other operations such as renaming, copying etc may also be controlled.

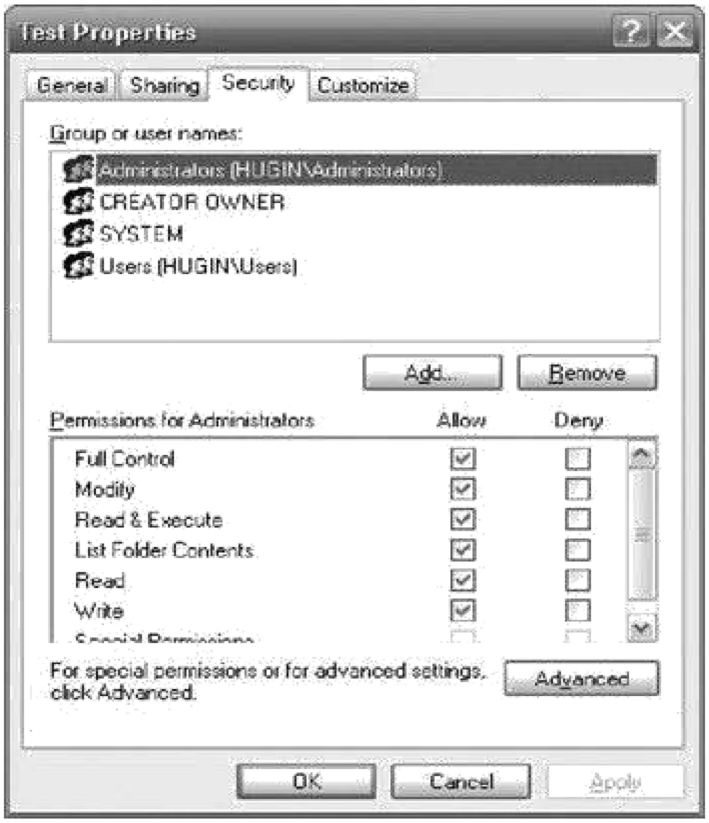
Access Control

The most common approach to the protection problem is to make access dependent on the identity of the user. The most general scheme to implement identity- dependent access is to associate with each file and directory an access- control list (ACL) specifying user names and the types of access allowed for each user.

This approach has the advantage of enabling complex access methodologies. The main problem with access lists is their length. To condense the length of the access control list, many systems recognize three classifications of users in connection with each file:

a) Owner – user who created the file
b) Group – set of users who are sharing the file and need similar access
c) Universe – all other users in the system

With the more limited protection classification, only three fields are needed to define protection. Each field is a collection of bits and each bit either allows or prevents the access associated with it. A separate field is kept for the file owner for the file's group and for all the other users.


Test Properties

General | Sharing | Security | Customize

Group or user names:

- Administrators (HUGIN\Administrators)
- CREATOR OWNER
- SYSTEM
- Users (HUGIN\Users)

Permissions for Administrators	Allow	Deny
Full Control	✓	
Modify	✓	
Read & Execute	✓	
List Folder Contents	✓	
Read	✓	
Write	✓	
Special Permissions		

For special permissions or for advanced settings, click Advanced.

Other Protection Approaches

Another approach to protection problem is to associate a password with each file. If the passwords are chosen randomly and changed often, this scheme may be effective in limiting access to a file.

Use of passwords has certain disadvantages –

1. The number of passwords that a user needs to remember may become large making the scheme impractical.
2. If only one password is used for all the files, then once it is discovered, all files are accessible.

UNIT -4 - IMPLEMENTING FILE SYSTEM

The file system provides the mechanism for on line storage and access to file contents including data and programs. The file system resides permanently on secondary storage which is designed to hold a large amount of data permanently.

11.1 File System Structure

A file is a logical storage unit or a collection of related information. File system resides on secondary storage (disks).
Disks provide the bulk of secondary storage on which a file system is maintained. They have two characteristics that make them a convenient medium for storing multiple files:

- A disk can be rewritten in place; it is possible to read a block from the disk, modify the block and write it back into the same place.
- A disk an access directly any given block of information it contains. It is simple to access any file sequentially or randomly and switching from one file to another requires only moving the read – write heads and waiting for the disk to rotate.

To improve I/O efficiency, I/O transfers between memory and disk are performed in units of blocks. Each block has one or more sectors.

To provide efficient and convenient access to the disk, OS imposes one or more file systems to allow the data to be stored, located and retrieved easily. The file system is composed of many different levels –

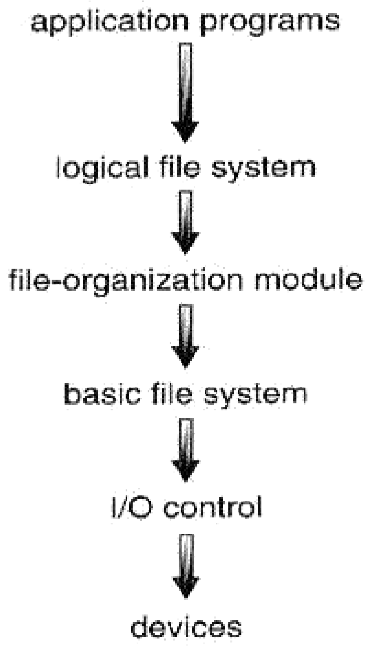


Fig: Layered file system

Each level in the design uses the features of lower levels to create new features for use by higher levels.

The lowest level, I/O control consists of device drivers and interrupts handlers to transfer information between the main memory and the disk system. The basic file system needs to issue generic commands to appropriate device driver to read and write physical blocks on the disk.

The file organization module knows about files and their logical blocks as well as physical blocks.

The logical file system manages metadata information. Metadata includes all of the file system structure except the actual data.

A file control block contains information about the file including ownership, permissions and location of the file contents.

11.2 File System Implementation

OS's implement open() and close() system calls for processes to request access to file contents.

Overview

Several on disk and in memory structures are used to implement a file system. These structures vary depending on the OS and the file system. File system may contain information such as:

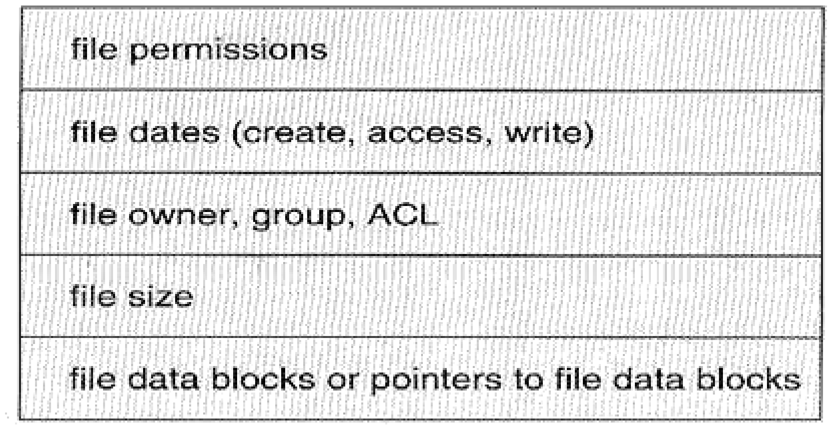
- Boot control block - In UFS, it is called the boot block; in NTFS it is partition boot sector.
- Volume control block – In UFS, it is called a super block; in NTFS it is stored in the master file table
- A directory structure per file system is used to organize the files. In UFS, this includes file names and associated inode numbers. In NTFS, it is stored in master file table.
- A per fie FCB contains many details about the file, including file permissions, ownership, size and location of data blocks. In UFS, it is called the inode. In NTFS this is stored within the master file table which uses a relational database structure.

The structures may include the ones described below –

✓ An in memory mount table contains information about each mounted volume

✓ An in memory directory structure cache holds the directory information of recently accessed directories.

- ✓ The system wide open file table contains a copy of the FCB of each open file
- ✓ The per process open file table contains a pointer to the appropriate entry in the system wide open file table.



file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

A typical file-control block.

Partitions and Mounting

The layout of a disk can have many variations depending on the OS. A disk can be sliced into multiple partitions or a volume can span multiple partitions on multiple disks. Each partition can be either raw containing no file system or may contain a file system. Raw disk is used where no file system is appropriate.

The root partition which contains OS kernel and sometimes other system files is mounted at boot time. As part of successful mount operation, OS verifies that the device contains a valid file system. OS finally notes in its in-memory mount table structure that a file system is mounted along with the type of the file system.

Virtual File Systems

An optimal method of implementing multiple types of file systems is to write directory and file routines for each type. Most operating systems use object oriented techniques to simplify, organize and modularize the implementation. Data structures and procedures are used to isolate the basic system call functionality from the implementation details. Thus, file system implementation consists of three major layers –

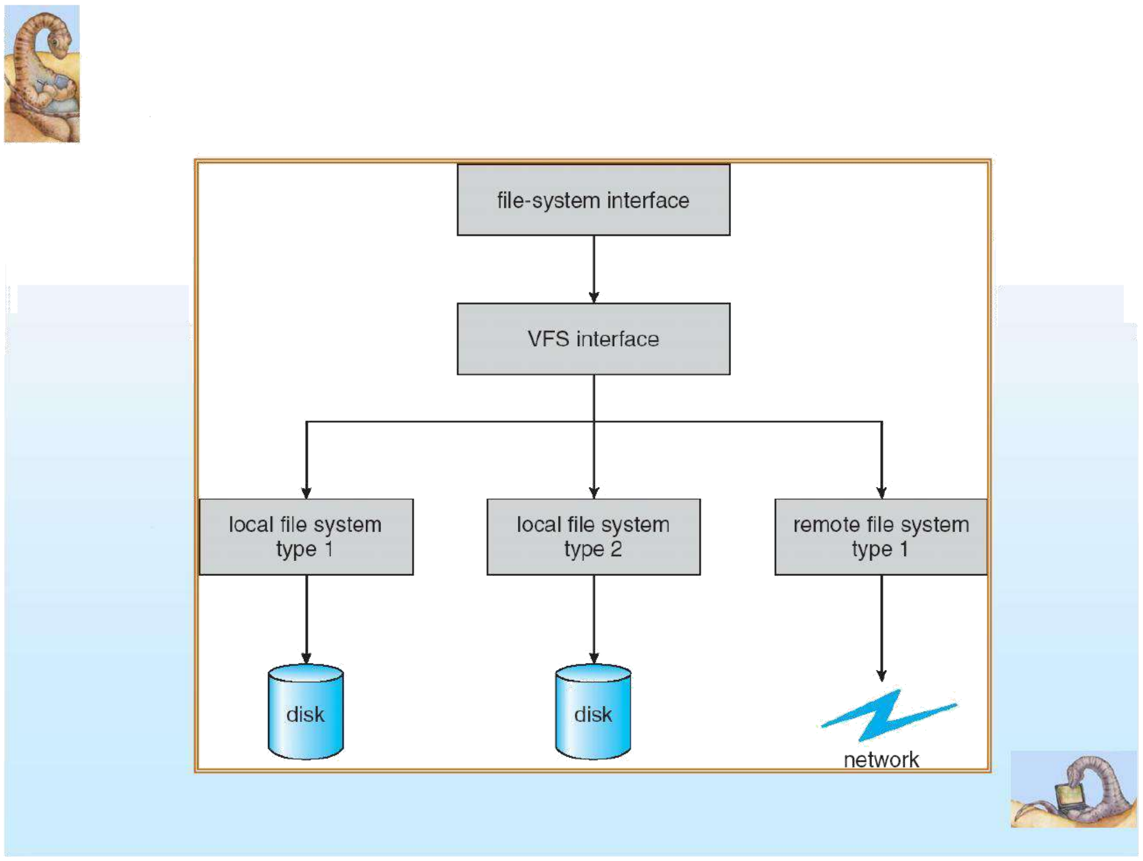


Figure: Schematic view of a virtual file system.

The first layer is the file system interface based on system calls and on file descriptors.

The second layer is called virtual file system layer which serves two important functions:

- ❖ Separates file system generic operations from their implementation by defining a clean VFS interface.
- ❖ VFS provides a mechanism for uniquely representing a file throughout a network.VFS is based on a file representation structure called vnode that contains a numerical designator for a network wide unique file.

Thus, VFS distinguishes local files from remote ones and local files are further distinguished according to their file system types.

VFS architecture in linux.The four main object types defined by the linux VFS are:

- The **inode object**,which represents an open file

- The **file object**, which represents an individual file
- The **super block**, which represents an entire file system
- The **dentry object,** which represents an individual directory entry

For each of these four object types, the VFS defines a set of operation that must be implemented.

11.3 Directory Implementation

The selection of directory allocation and directory management algorithms significantly affects the efficiency, performance and reliability of the file system.

Linear List

The simplest method of implementing a directory is to use a linear list of file names with pointers to the data blocks. This method is simple to program but time consuming to execute. The real disadvantage of a linear list of directory entries is that finding a file requires a linear search.

Hash Table

Another data structure used for a file directory is a **hash table.** With this method, a linear list stores the directory entries but a hash data structure is also used. The hash table takes a value computed from the file name and returns a pointer to the file name in the linear list.

The major difficulties with a hash table are its generally fixed size and the dependence of the hash function on that size.

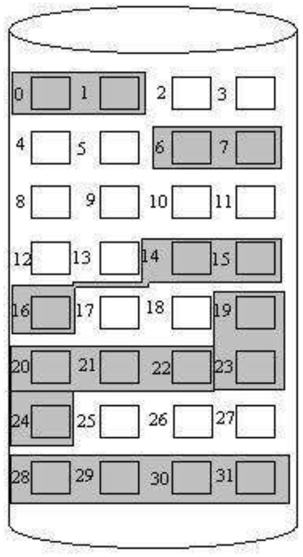
11.4 Allocation Methods

The direct access nature of disks allows flexibility in the implementation of files. The main problem here is how to allocate space to these files so that disk space is utilized effectively and files can be accessed quickly. Three major methods of allocating disk space are:

i. **Contiguous**
ii. **Linked**
iii. **Indexed**

Contiguous Allocation

This allocation requires that each file occupy a set of contiguous blocks on the disk. The number of disk seeks required for accessing contiguously allocated files is minimal. Contiguous allocation of a file is defined by the disk address and length of the first block. Accessing a file that has been contiguously allocated is easy. Both sequential and direct access can be supported by contiguous allocation. Disadvantage is finding space for a new file.



Directory

file	start	length
Count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

This problem can be seen as a particular application of general dynamic storage allocation problem which involves how to satisfy a request of size n form a list of free holes. First fit and best fit are the most common strategies used to select a free hole from the set of available holes.

These algorithms suffer from external fragmentation. As files are allocated and deleted, the free disk space is broken into little pieces. External fragmentation exists whenever free space is broken into chunks. For solving the fragmentation problem, compact all free space into one contiguous space.

Another problem with contiguous allocation is determining how much space is needed for a file.

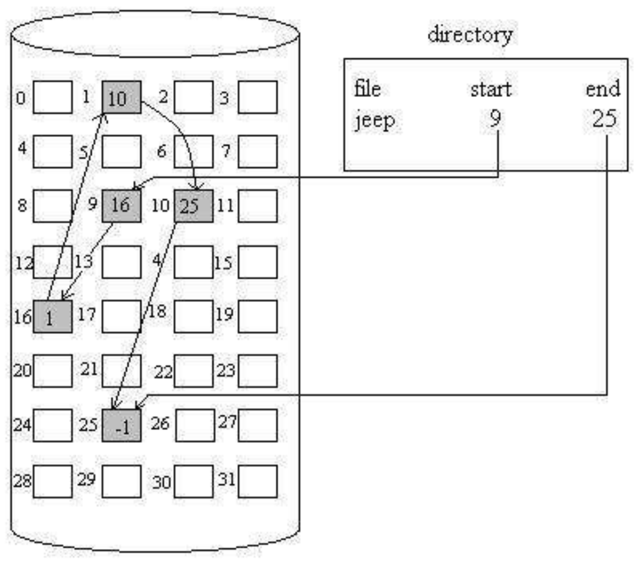
Pre allocation of memory space to a file may be insufficient. A file may be allocated space for its final size but large amount of that space will remain unused for a long time. The file therefore has a large amount of internal fragmentation.

To minimize these drawbacks, some operating systems use a modified contiguous allocation scheme. Here a contiguous chunk of space is allocated initially and if that amount proves not to be large enough another chunk of contiguous space called **extent** is added. Internal fragmentation can still be a problem if the extents are too large and external fragmentation can become a problem as extents of varying sizes are allocated and de allocated.

Linked Allocation

This solves all problems of contiguous allocation. Each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file. Each block contains a pointer to the next block. There is no external fragmentation with linked allocation and any free block on the free space list can be used to satisfy a request. But the major problem is that it can be used **effectively only for sequential access files**. It is inefficient to support a direct access capability for linked allocation files. Another disadvantage is **space required for pointers**.

Solution to this problem is to collect blocks into multiples called clusters and to allocate clusters rather than blocks. This method allows logical to physical block mapping to remain simple but improved disk through put and decreases the space needed for block allocation and free list management. This increases internal fragmentation because more space is wasted when a cluster is partially full than when a block is partially full. Another problem of linked allocation is reliability.



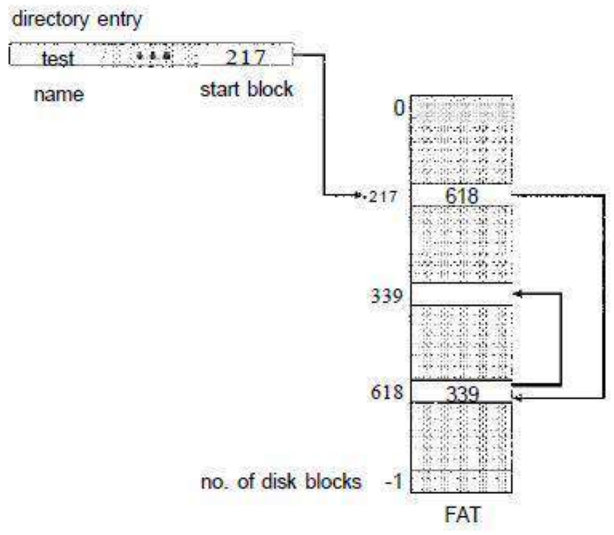
File Allocation Table(FAT)

- An important variation on the linked allocation method is the use of a file allocation table(FAT).
- This simple but efficient method of disk- space allocation is used by the MS-

DOS and OS/2 operating systems.

- A section of disk at beginning of each partition is set aside to contain the table.
- The table has entry for each disk block, and is indexed by block number.
- The FAT is used in much the same way as is a linked list.
- The directory entry contains the block number of the first block of the file.
- The table entry indexed by that block number contains the block number of the next block in the file.

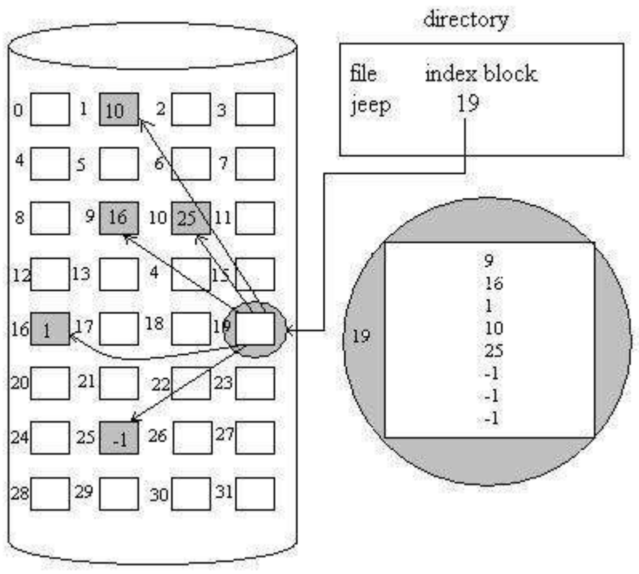
This chain continues until the last block which has a special end – of – file value as the table entry. Unused blocks are indicated by a 0 table value. Allocating a new block file is a simple matter of finding the first 0 – valued table entry, and replacing the previous end of file value with the address of the new block.The 0 is replaced with the end – of – file value, an illustrative example is the FAT structure for a file consisting of disk blocks 217,618, and 339.



__Indexed Allocation__

Linked allocation solves external fragmentation and size declaration problems of contiguous allocation. In the absence of FAT, linked allocation cannot support efficient direct access since the pointers to the blocks are scattered with the blocks themselves all over the disk and must be retrieved in order. Indexed allocation solves this problem by bringing all pointers together into one location – index block.

Each file has its own index block which is an array of disk block addresses.



Indexed allocation supports direct access without suffering from external fragmentation because any free block on the disk can satisfy a request for more space. But indexed allocation suffers from wasted space. Every file must have an index block so it should be as small as possible. But if it is too small, it will not be able to hold enough pointers for a large file and a mechanism will have to be available to deal with this issue. Mechanisms for this purpose include –

Linked Scheme: An index block is normally one disk block. Thus, it can be read and written directly by itself. To allow for large files, we may link together several index blocks.

Multilevel index: A variant of the linked representation is to use a first level index block to point to a set of second – level index blocks.

Combined scheme: Another alternative, used in the UFS, is to keep the first, say, 15 pointers of the index block in the file's inode.

- The first 12 of these pointers point to direct blocks; that is for small (no more than 12 locks) files do not need a separate index block
- The next pointer is the address of a single indirect block.
- The single indirect block is an index block, containing not data, but rather the addresses of blocks that do contain data

- Then there is a double indirect block pointer, which contains the address of a block that contain pointers to the actual data blocks. The last pointer would contain pointers to the actual data blocks.
- The last pointer would contain the address of a triple indirect block.

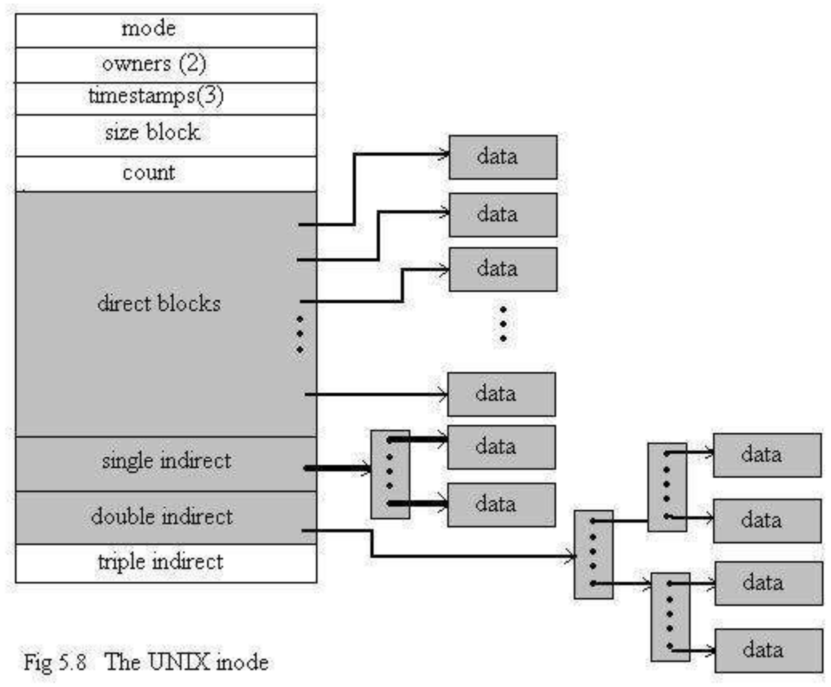


Fig 5.8 The UNIX inode

<u>Performance</u>

The allocation methods vary in their storage efficiency and data block access times. Both are important in selecting the proper method for an operating system to implement. Before selecting an allocation method, determine how systems will be used.

For any type of access, contiguous allocation requires only one access to get a disk block. For linked allocation, we can keep the address of the next block in memory and read it directly. This method is fine for sequential access. Hence some systems support direct access files by using contiguous allocation and sequential access by linked allocation.

11.5 Free Space Management

Since disk space is limited, we should reuse the space from deleted files for new files. To keep track of free disk space, the system maintains a **free space list**. The free space list records all free disk blocks – those not allocated to some file or directory. This free space list can be implemented as one of the following:

a) **Bit vector** – free space list is implemented as a bit map or a bit vector. Each block is represented by one bit. If the block is free, bit is 1, if the block is allocated, bit is 0.

   The main advantage of this approach is its relative simplicity and its efficiency in finding the first free block or n consecutive free blocks on the disk. The calculation of the block number is

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

b) **Linked list** – Another approach to free space management is to link together all the free disk blocks keeping a pointer to the first free block in a special location on the disk and caching it in memory. The first block contains a pointer to the next free disk block.

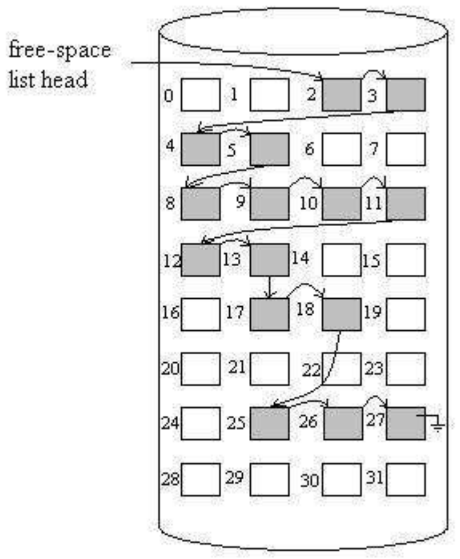


fig: Linked free-space list on disk.

c) **Grouping** – A modification of the free linked list approach is to store the addresses of n free blocks in the first free block.

d) **Counting** – Another approach is to take advantage of the fact that several contiguous blocks may be allocated or freed simultaneously when space is allocated with the contiguous allocation algorithm or clustering. Thus rather than keeping a list of n free disk addresses, we can keep the address of the first free block and the number n of free contiguous blocks that follow the first block.

Efficiency and Performance

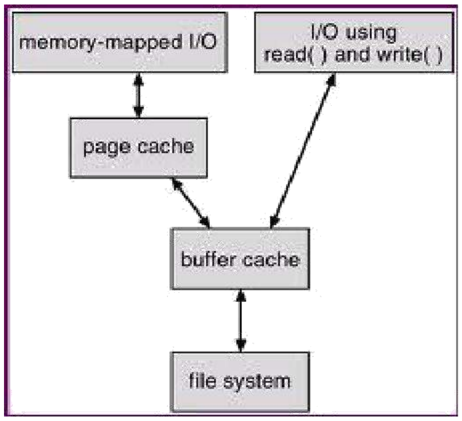
Disks tend to represent a major bottleneck in system performance since they are the slowest main computer component.

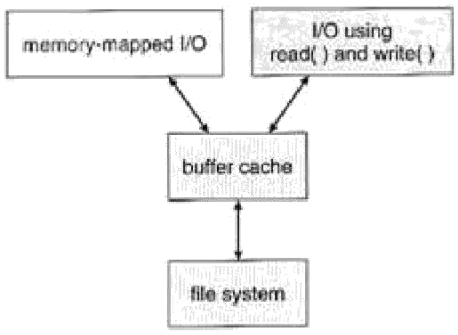
Efficiency

- The efficient use of disk space depends heavily on the disk allocation and directory algorithms in use.
- Types of data kept in files directory entry
- Pre-allocation or as needed allocation of metadata structures.
- Fixed size or varying size data structures.

Performance

Most disk controllers include local memory to form an on board cache that is large enough to store entire tracks at a time. Once a seek is performed, the track is read into the disk cache starting at the sector under the disk head. The disk controller then transfers any sector requests to OS. Some systems maintain a separate section of main memory for a **buffer cache** where blocks are kept under the assumption that they will be used again. Other systems cache file data using a **page cache**. The page cache uses virtual memory techniques to cache file data as pages rather than as a file system oriented blocks. Caching file data using virtual addresses is more efficient than caching through physical disk blocks as accesses interface with virtual memory rather than the file system. Several systems use page caching to cache both process pages and file data. This is known as **unified buffer cache**.





I/O using a unified buffer cache.

There are other issues that can affect the performance of I/O such as whether writes to the file system occur synchronously or asynchronously. Synchronous writes occur in the order in which the disk subsystem receives them and the writes are not buffered. Asynchronous writes are done the majority of the time.

Some systems optimize their page cache by using different replacement algorithms depending on the access type of the file. Sequential access can be optimized by techniques known as free behind and read ahead. Free behind removes a page from buffer as soon as the next page is requested. With read ahead, a requested page and several subsequent pages are read and cached.
Although each entry requires more space than would a simple disk address, the overall list will be shorter, as long as the count is generally greater than 1.

11.7 Recovery:

Files and directories are kept both in main memory and on disk, and care must be taken to ensure that system failure does not result in loss of data or in data inconsistency.

1.Consistency Checking

1. The directory information in main memory is generally more up to date than is the corresponding information on the disk, because cached directory information is not necessarily written to disk as soon as the update takes place
2. Frequently, a special program is run at reboot time to check for and correct disk inconsistencies.
3. The consistency checker—a systems program such as fcsk in UNIX or chkdsk in MS-DOS—compares the data in the directory structure with the data blocks on disk and tries to fix any inconsistencies it finds. The allocation and free-space-management algorithms dictate what types of problems the checker can find and how successful it will be in fixing them.

2. Backup and Restore

- Magnetic disks sometimes fail, and care must be taken to ensure that thedata lost in such a failure are not lost forever. To this end, system programscan be used to **back** up data from disk to another storage device, such as a floppy disk, magnetic tape, optical disk, or other hard disk.
- Recovery from the loss of an individual file, or of an entire disk, may then be a matter of **restoring** the data from backup.

A **typical backup schedule** may then be as follows:

Day 1: Copy to a backup medium all files from the disk. This is called a **full backup.**

Day 2: Copy to another medium all files changed since day 1. This is an **incremental backup.**

Day 3: Copy to another medium all files changed since day 2

.

.

Day N: Copy to another medium all files changed since day N— 1. Then go back to Day 1.

11.8 Log-Structured File Systems

- Computer scientists often find that algorithms and technologies originally used in one area are equally useful in other areas.
- These logging algorithms have been applied successfully to the problem of consistency checking.
- The resulting implementations are known as **log-based transaction-oriented** (or **journaling)** file systems.
- Fundamentally, all metadata changes are written sequentially to a log.
- Each set of operations for performing a specific task is a transaction.
- Once the changes are written to this log, they are considered to be committed, and the system call can return to the user process, allowing it tocontinue execution.
- As the changes are made, a pointer is updated to indicate which actions have completed and which are still incomplete.
- When an entire committed transaction is completed, it is removed from the log file, which is actually a circular buffer.
- A circular buffer writes to the end of its space and then continues at the beginning, overwriting older values as it goes. If the system crashes, the log file will contain zero or more transactions.

UNIT-4-PART II-SECONDARY STORAGE STRUCTURE

Mass-Storage Systems

- Describe the physical structure of secondary and tertiary storage devices and the resulting effects on the uses of the devices.
- To explain the performance characteristics of mass-storage devices.
- To discuss operating-system services provided for mass storage, including RAID and HSM

12.1 Overview of Mass Storage Structure

Magnetic disks provide bulk of secondary storage of modern computers:

- Drives **rotate** at 60 to 200 times per second
- **Transfer rate** is the rate at which data flow between drive and computer
- **Positioning time** (random-access time) is time to move disk arm to desired cylinder (**seek time**)

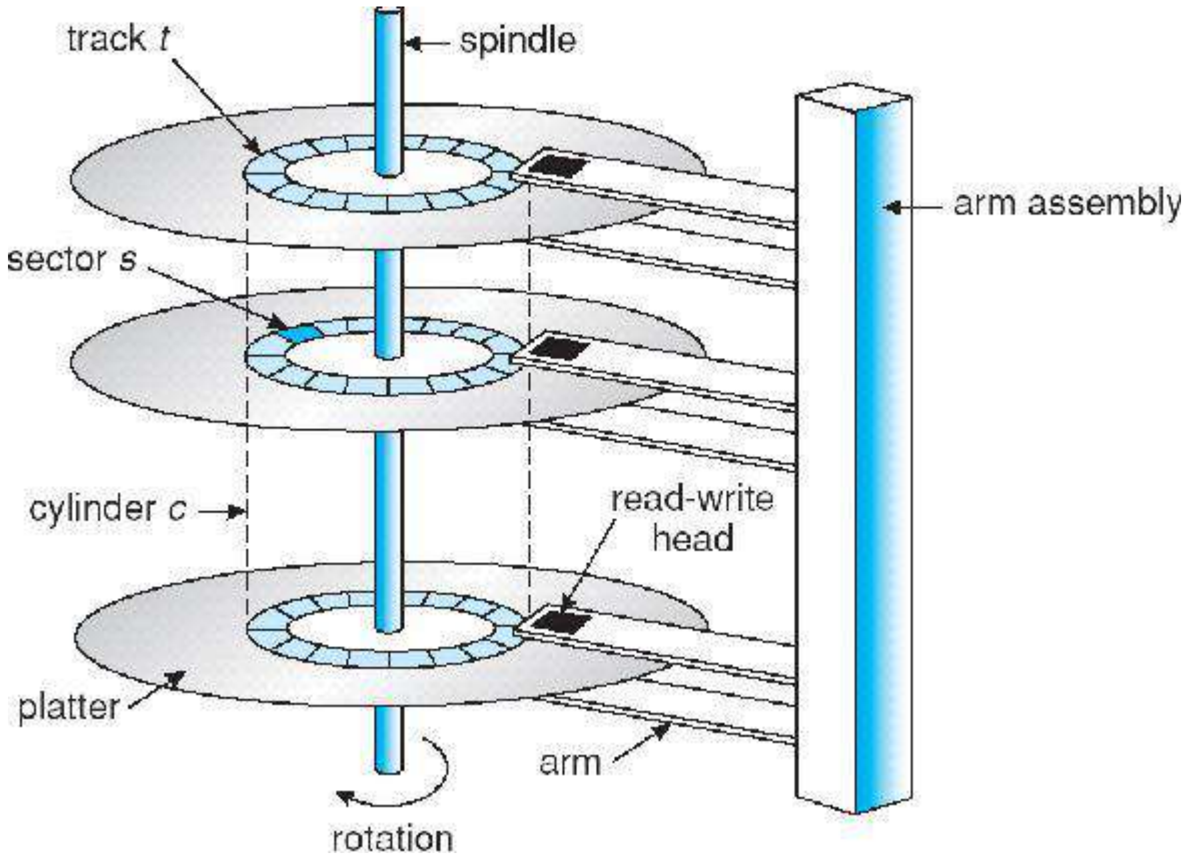
and time for desired sector to rotate under the disk head (**rotational latency**)

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

12.2 Disk Structure

- Disks can be **removable**
- Disk drives are addressed as large **1-dimensional arrays** of **logical blocks**, where the logical block is the **smallest unit of transfer**
- The 1-dimensional array of logical blocks is mapped into the **sectors** of the disk sequentially.**Sector 0** is the first sector of the **first track** on the **outermost cylinder.**Mapping proceeds in order through that track, then the rest of the tracks in that cylinder andthen through the rest of the cylinders from **outermost to innermost.**

Moving-head Disk Mechanism



12.3 Disk Attachment

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

SCSI is a bus architecture and can support up to 16 devices on one cable, SCSI initiator requests operation and SCSI targets perform tasks. The devices include one controller card in the host (SCSI initiator) and upto 15 storage devices (SCSI targets). A SCSI disk is a common SCSI target and each target can address up to 8 logical units (disks attached to device controller

FC is high-speed serial architecture that operates over optical fiber. It has 2 variants.

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

Other is an arbitrated loop (FC-AL) which can address 126 devices

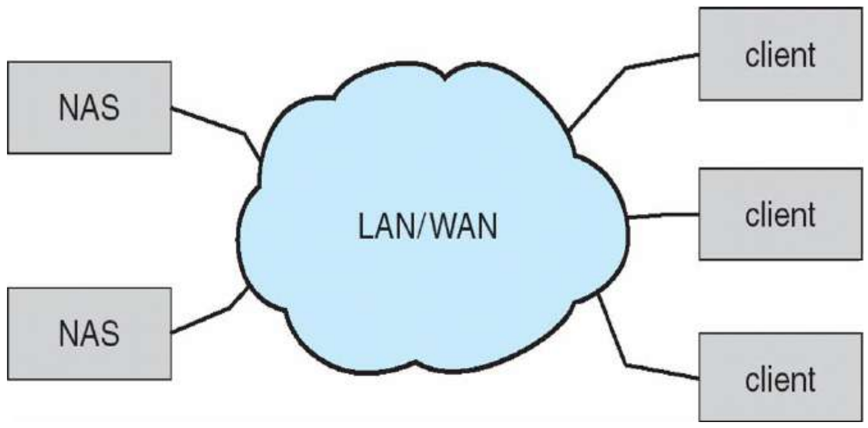
Network-Attached Storage

Network-attached storage (NAS) is storage made available over a network rather than over a local connection (such as a bus)

NFS and CIFS are common protocols

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

New iSCSI protocol uses IP network to carry the SCSI protocol



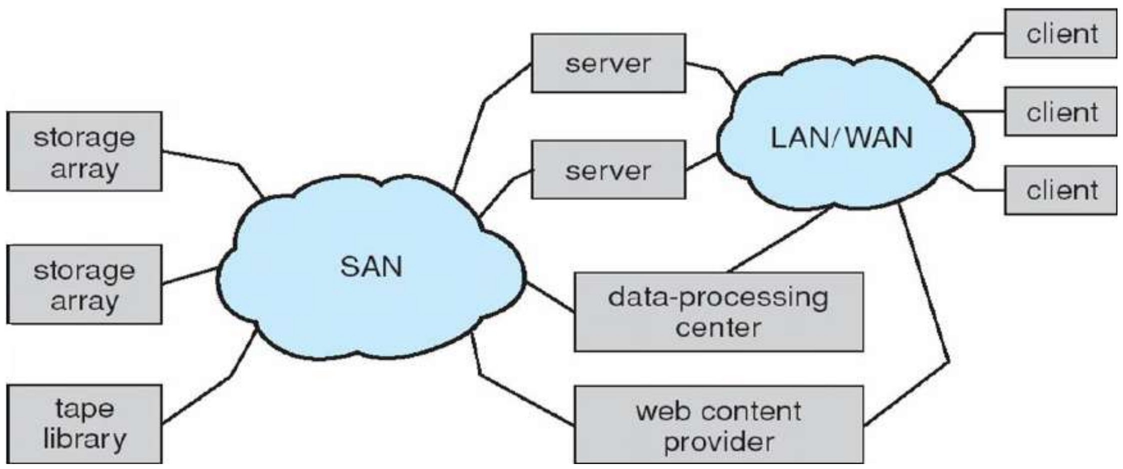
Storage Area Network

Common in large storage environments (and becoming more common)

SAN is a private network connecting servers and storage units.

Multiple hosts attached to multiple storage arrays – flexible

A SAN switch allows or prohibits access between the hosts and the storage.



12.4 **Disk Scheduling**

The operating system is responsible for using hardware efficiently.
For the disk drives, this means having a fast ***access time*** & ***disk bandwidth***.

- ✓ **Access time** has two major components:
  Seek time is the time for the disk to **move the heads** to the cylinder containing the desired sector
  Rotational latency time waiting for the disk to rotate the desired sector to the disk head

  We like to ***minimize seek time***.

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

  Several algorithms exist to schedule the servicing of disk I/O requests.
  We illustrate them with a **Request Queue** (cylinder range 0-199):

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

Head pointer: cylinder 53

FCFS (first come first serve)

Illustration shows total head movement of **640** cylinders

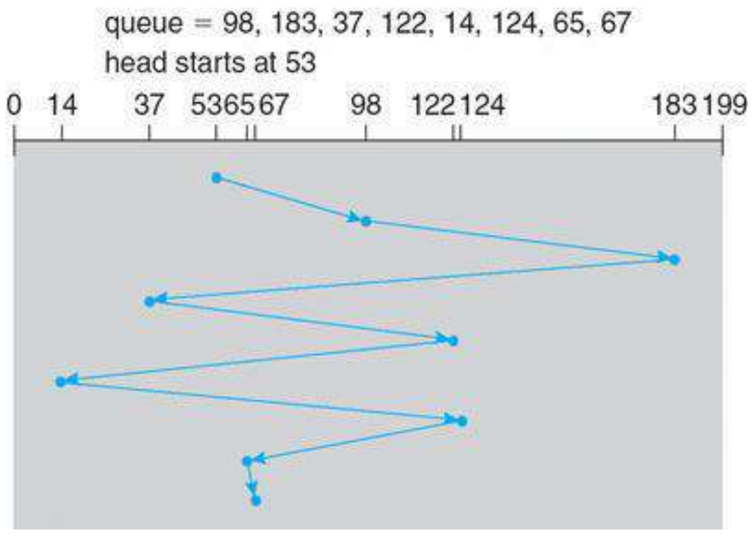


Fig: FCFS disk scheduling .

SSTF (shortest seek time first)

Selects the request with the **minimum seek time** from the current head position
SSTF scheduling may cause **starvation** of some requests

Illustration shows total head movement of **236** cylinders

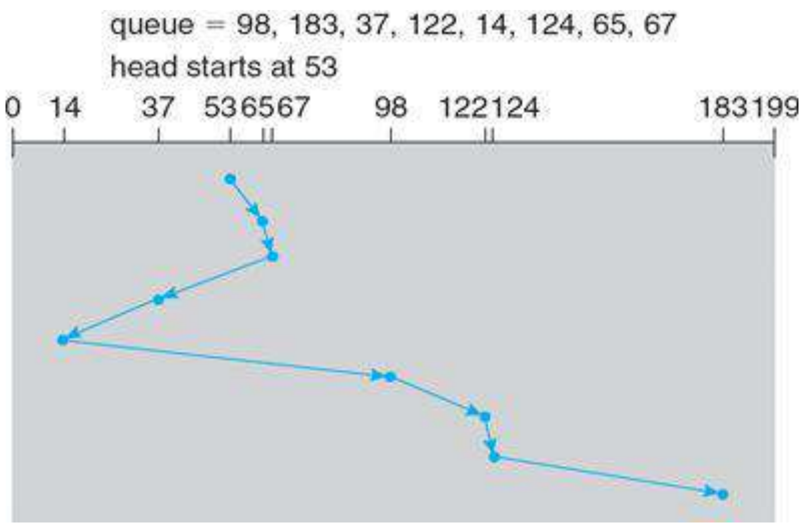


Fig: SSTF disk scheduling.

SCAN

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

SCAN algorithm sometimes called the **elevator** algorithm.
We need to know the direction of head movement in addition to the heads current position.

Illustration shows total head movement of **208** cylinders

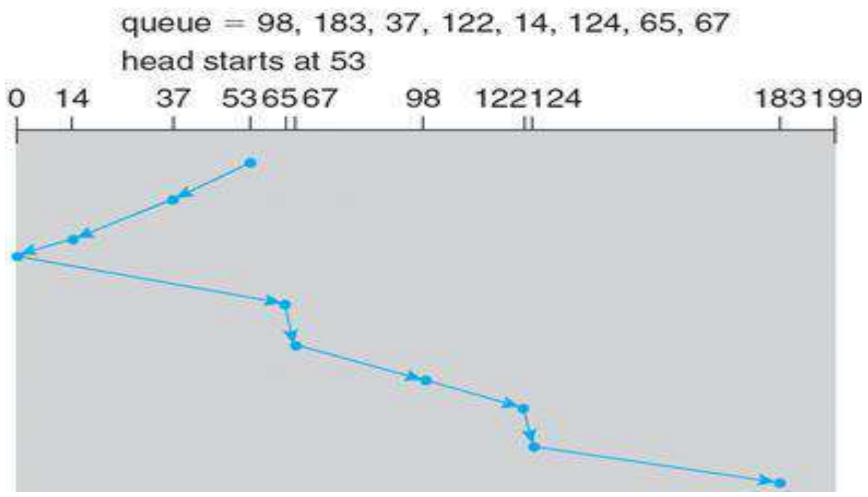


Fig: SCAN disk scheduling.

C-SCAN (circular SCAN)

Provides a more uniform wait time than SCAN

- ✓ The head moves from one end of the disk to the other, servicing requests as it goes
- ✓ When it reaches the other end, however, it immediately **returns to the beginning of the disk**,without servicing any requests on the return trip.

Treats the cylinders as a **C**ircular list that wraps around from the last cylinder to the first one

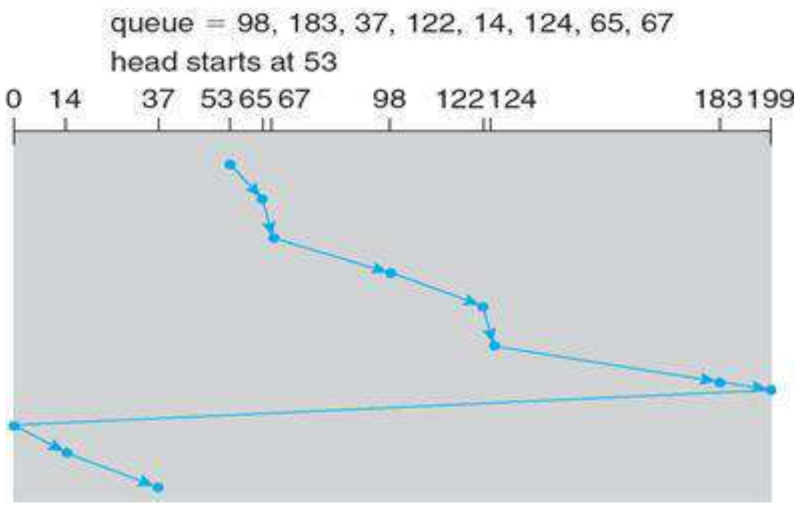


Fig:C-SCAN disk scheduling

C-LOOK

- ✓ Version of C-SCAN
- ✓ Arm only goes as far as the **last request** in each direction,

then reverses direction immediately, without first going all the way to the end of the disk.

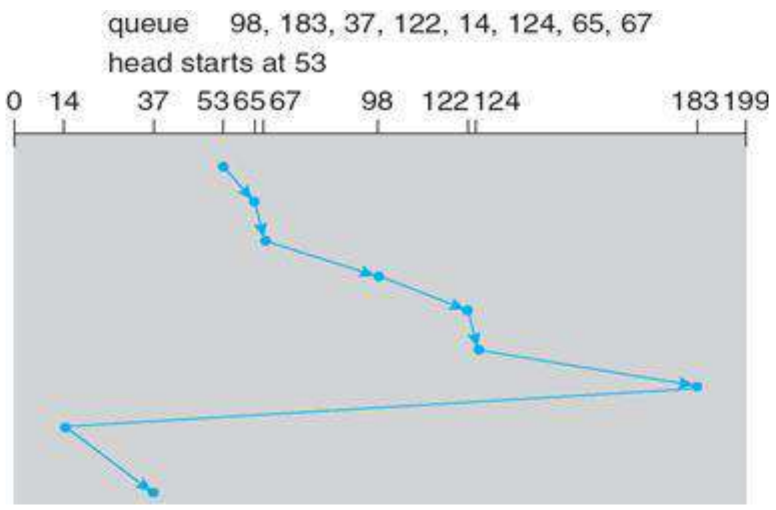


Figure: C-LOOK disk scheduling.

LOOK : version of SCAN (refer notes)

Selecting a Disk-Scheduling Algorithm

- ✓ **SSTF** is common and has a natural appeal
- ✓ **SCAN** and **C-SCAN** perform better for systems that place a heavy load on the disk

Performance depends on the number and types of requests

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

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

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

12. 5 Disk Management

Disk Formatting

- Before a disk can be used to store data, it has to be ***low-level formatted***, which means laying down all of the headers and trailers demarking the beginning and ends of each sector. Included in the header and trailer are the linear sector numbers, and ***error-correcting codes, ECC,*** which allow damaged sectors to not only be detected, but in many cases for the damaged data to be recovered (depending on the extent of the damage.) Sector sizes are traditionally 512 bytes, but may be larger, particularly in larger drives.
- ECC calculation is performed with every disk read or write, and if damage is detected but the data is recoverable, then a ***soft error*** has occurred. Soft errors are generally handled by the on-board disk controller, and never seen by the OS. (See below.)
- Once the disk is low-level formatted, the next step is to partition the drive into one or more separate partitions. This step must be completed even if the disk is to be used as a single large partition, so that the partition table can be written to the beginning of the disk.
- After partitioning, then the filesystems must be ***logically formatted,*** which involves laying down the master directory information (FAT table or inode structure), initializing free lists, and creating at least the root directory of the filesystem. (Disk partitions which are to be used as raw devices are not logically formatted. This saves the overhead and disk space of the filesystem structure, but requires that the application program manage its own disk storage requirements.)

Boot Block

- Computer ROM contains a ***bootstrap*** program (OS independent) with just enough code to find the first sector on the first hard drive on the first controller, load that sector into memory, and transfer control over to it. (The ROM bootstrap program may look in floppy and/or CD drives before accessing the hard drive, and is smart enough to recognize whether it has found valid boot code or not.)
- The first sector on the hard drive is known as the ***Master Boot Record, MBR,*** and contains a very small amount of code in addition to the ***partition table.*** The partition table documents how the disk is partitioned into logical disks, and indicates specifically which partition is the ***active*** or ***boot*** partition.
- Boot code instructs disk controller to read the boot blocks.
- The boot program then looks to the active partition to find an operating system, possibly loading up a slightly larger / more advanced boot program along the way.
- In a ***dual-boot*** (or larger multi-boot) system, the user may be given a choice of which operating system to boot, with a default action to be taken in the event of no response within some time frame.
- Once the kernel is found by the boot program, it is loaded into memory and then control is transferred over to the OS. The kernel will normally continue the boot process by

initializing all important kernel data structures, launching important system services (e.g. network daemons, sched, init, etc.), and finally providing one or more login prompts. Boot options at this stage may include ***single-user*** a.k.a. ***maintenance*** or ***safe*** modes, in which very few system services are started - These modes are designed for system administrators to repair problems or otherwise maintain the system.

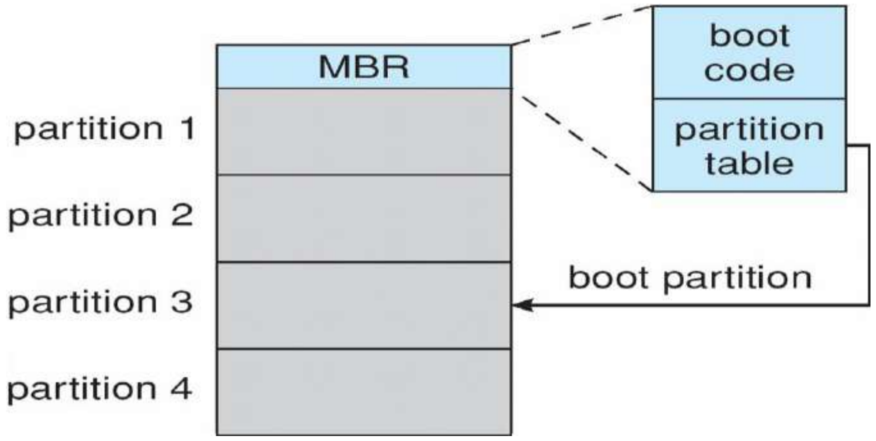


Fig: Booting from disk in windows 2000

Bad Blocks

- No disk can be manufactured to 100% perfection, and all physical objects wear out over time. For these reasons all disks are shipped with a few bad blocks, and additional blocks can be expected to go bad slowly over time. If a large number of blocks go bad then the entire disk will need to be replaced, but a few here and there can be handled through other means.
- In the old days, bad blocks had to be checked for manually. Formatting of the disk or running certain disk-analysis tools would identify bad blocks, and attempt to read the data off of them one last time through repeated tries. Then the bad blocks would be mapped out and taken out of future service. Sometimes the data could be recovered, and sometimes it was lost forever. (Disk analysis tools could be either destructive or non-destructive.)
- Modern disk controllers make much better use of the error-correcting codes, so that bad blocks can be detected earlier and the data usually recovered. (Recall that blocks are tested with every write as well as with every read, so often errors can be detected before the write operation is complete, and the data simply written to a different sector instead.)
- Bad blocks can be handled by sector sparing or sector slipping.
- Sector sparing is replacing bad sector logically with one of the spare sectors.
- Note that re-mapping of sectors from their normal linear progression can throw off the disk scheduling optimization of the OS, especially if the replacement sector is physically far away from the sector it is replacing. For this reason most disks normally keep a few spare sectors on each cylinder, as well as at least one spare cylinder. Whenever possible a

bad sector will be mapped to another sector on the same cylinder, or at least a cylinder as close as possible. ***Sector slipping*** may also be performed, in which all sectors between the bad sector and the replacement sector are moved down by one, so that the linear progression of sector numbers can be maintained.

- If the data on a bad block cannot be recovered, then a ***hard error*** has occurred., which requires replacing the file(s) from backups, or rebuilding them from scratch.

12.6 Swap-Space Management

- Modern systems typically swap out pages as needed, rather than swapping out entire processes. Hence the swapping system is part of the virtual memory management system.
- Managing swap space is obviously an important task for modern OSes.

Swap-Space Use

- Swap space or a swap file is a space on hard disk used as virtual memory extension of a computers real memory.
- The amount of swap space needed by an OS varies greatly according to how it is used. Some systems require an amount equal to physical RAM; some want a multiple of that; some want an amount equal to the amount by which virtual memory exceeds physical RAM, and some systems use little or none at all!
- Some systems support multiple swap spaces on separate disks in order to speed up the virtual memory system.
- Varies from megabytes to gigabytes.

Swap-Space Location

Swap space can be physically located in one of two locations:

- As a large file which is part of the regular filesystem. This is easy to implement, but inefficient. Not only must the swap space be accessed through the directory system, the file is also subject to fragmentation issues. Caching the block location helps in finding the physical blocks, but that is not a complete fix.
- As a raw partition, possibly on a separate or little-used disk. This allows the OS more control over swap space management, which is usually faster and more efficient. Fragmentation of swap space is generally not a big issue, as the space is re-initialized every time the system is rebooted. The downside of keeping swap space on a raw partition is that it can only be grown by repartitioning the hard drive.

Swap-Space Management: An Example

- Historically OSes swapped out entire processes as needed. Modern systems swap out only individual pages, and only as needed. (For example process code blocks and other blocks that have not been changed since they were originally loaded are normally just freed from the virtual memory system rather than copying them to swap space, because it is faster to go find them again in the filesystem and read them back in from there than to write them out to swap space and then read them back.)

- In the mapping system shown below for Linux systems, a map of swap space is kept in memory, where each entry corresponds to a 4K block in the swap space. Zeros indicate free slots and non-zeros refer to how many processes have a mapping to that particular block (>1 for shared pages only.)
- Value>0 indicates that the page slot is occupied by a swap space
- Value>1 indicates that the page slot is mapped to multiple processes (shared pages)

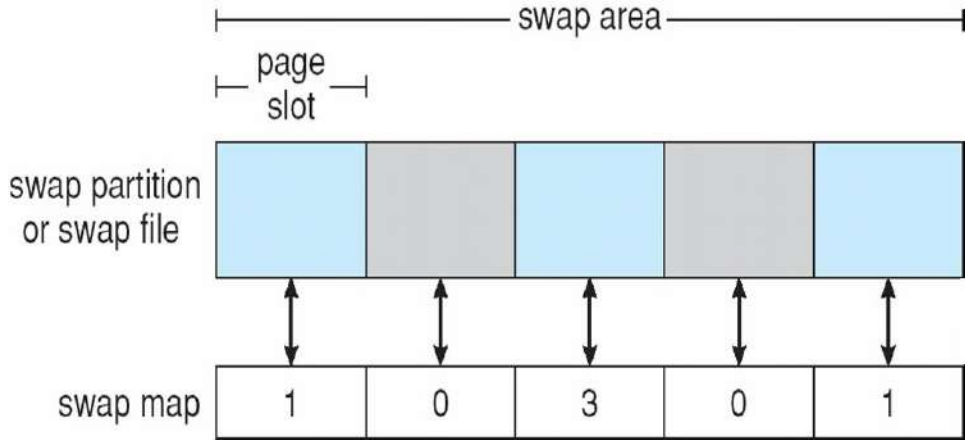


Figure: The data structures for swapping on Linux systems

12.7 RAID Structure

- The general idea behind RAID is to employ a group of hard drives together with some form of duplication, either to increase reliability or to speed up operations, (or sometimes both.)
- ***RAID*** originally stood for ***Redundant Array of Inexpensive Disks,*** and was designed to use a bunch of cheap small disks in place of one or two larger more expensive ones. Today RAID systems employ large possibly expensive disks as their components, switching the definition to ***Independent*** disks.

Improvement of Reliability via Redundancy

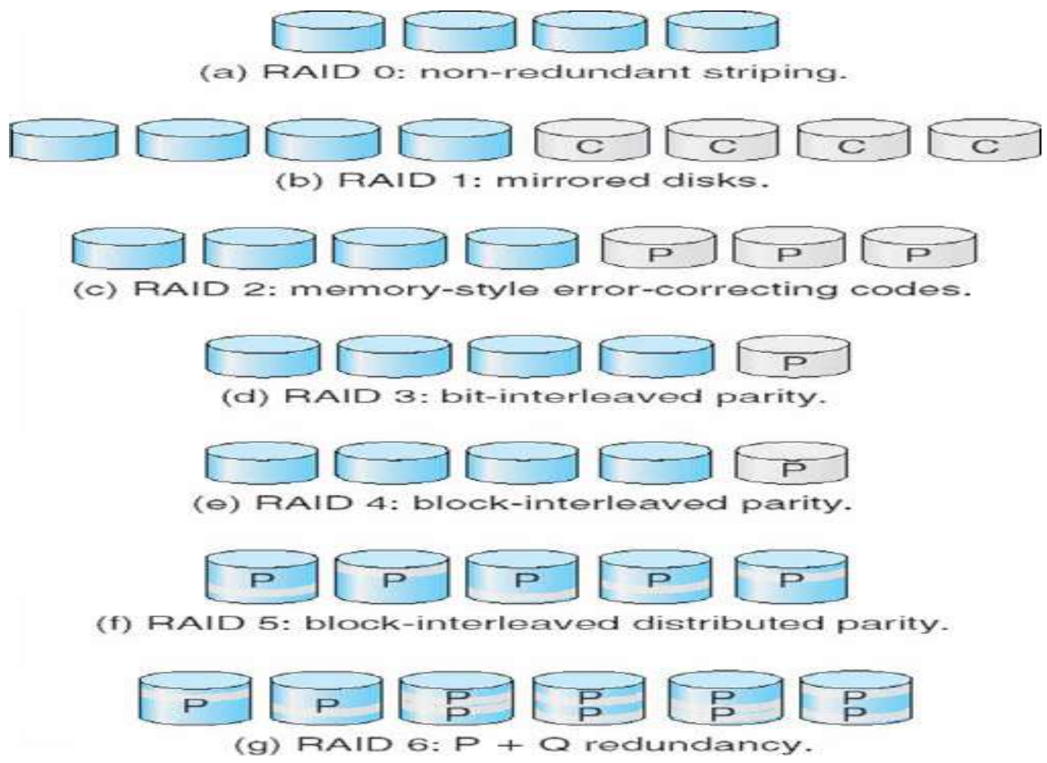
- The more disks a system has, the greater the likelihood that one of them will go bad at any given time. Hence increasing disks on a system actually ***decreases*** the **Mean Time To Failure, MTTF** of the system.

- If, however, the same data was copied onto multiple disks, then the data would not be lost unless **both** (or all) copies of the data were damaged simultaneously, which is a **MUCH** lower probability than for a single disk going bad. More specifically, the second disk would have to go bad before the first disk was repaired, which brings the **Mean Time To Repair** into play. For example if two disks were involved, each with a MTTF of 100,000 hours and a MTTR of 10 hours, then the **Mean Time to Data Loss** would be 500 * 10^6 hours, or 57,000 years!
- This is the basic idea behind disk ***mirroring***, in which a system contains identical data on two or more disks.
    - Note that a power failure during a write operation could cause both disks to contain corrupt data, if both disks were writing simultaneously at the time of the power failure. One solution is to write to the two disks in series, so that they will not both become corrupted (at least not in the same way) by a power failure. And alternate solution involves non-volatile RAM as a write cache, which is not lost in the event of a power failure and which is protected by error-correcting codes.

Improvement in Performance via Parallelism

- There is also a performance benefit to mirroring, particularly with respect to reads. Since every block of data is duplicated on multiple disks, read operations can be satisfied from any available copy, and multiple disks can be reading different data blocks simultaneously in parallel. (Writes could possibly be sped up as well through careful scheduling algorithms, but it would be complicated in practice.)
- Another way of improving disk access time is with ***striping***, which basically means spreading data out across multiple disks that can be accessed simultaneously.
    - ✓ With ***bit-level striping*** the bits of each byte are striped across multiple disks. For example if 8 disks were involved, then each 8-bit byte would be read in parallel by 8 heads on separate disks. A single disk read would access 8 * 512 bytes = 4K worth of data in the time normally required to read 512 bytes. Similarly if 4 disks were involved, then two bits of each byte could be stored on each disk, for 2K worth of disk access per read or write operation.
    - ✓ ***Block-level striping*** spreads a file system across multiple disks on a block-by-block basis, so if block N were located on disk 0, then block N + 1 would be on disk 1, and so on. This is particularly useful when file systems are accessed in ***clusters*** of physical blocks. Other striping possibilities exist, with block-level striping being the most common.

RAID Levels



C
(a) RAID 0: non-redundant striping.
(b) RAID 1: mirrored disks.
P
(c) RAID 2: memory-style error-correcting codes.
(d) RAID 3: bit-interleaved parity.
(e) RAID 4: block-interleaved parity.
(f) RAID 5: block-interleaved distributed parity.
(g) RAID 6: P + Q redundancy.

Figure: RAID levels

- Mirroring provides reliability but is expensive; Striping improves performance, but does not improve reliability. Accordingly there are a number of different schemes that combine the principals of mirroring and striping in different ways, in order to balance reliability versus performance versus cost. These are described by different ***RAID levels***, as follows: (In the diagram that follows, "C" indicates a copy, and "P" indicates parity, i.e. checksum bits.)
  1. ***Raid Level 0 -*** This level includes striping only, with no mirroring.
  2. ***Raid Level 1 -*** This level includes mirroring only, no striping.
  3. ***Raid Level 2 -*** This level stores error-correcting codes on additional disks, allowing for any damaged data to be reconstructed by subtraction from the remaining undamaged data. Note that this scheme requires only three extra disks to protect 4 disks worth of data, as opposed to full mirroring. (The number of disks required is a function of the error-correcting algorithms, and the means by which the particular bad bit(s) is(are) identified.)
  4. ***Raid Level 3 -*** This level is similar to level 2, except that it takes advantage of the fact that each disk is still doing its own error-detection, so that when an error occurs, there is no question about which disk in the

array has the bad data. As a result a single parity bit is all that is needed to recover the lost data from an array of disks. Level 3 also includes striping, which improves performance. The downside with the parity approach is that every disk must take part in every disk access, and the parity bits must be constantly calculated and checked, reducing performance. Hardware-level parity calculations and NVRAM cache can help with both of those issues. In practice level 3 is greatly preferred over level 2.

5. ***Raid Level 4*** **-** This level is similar to level 3, employing block-level striping instead of bit-level striping. The benefits are that multiple blocks can be read independently, and changes to a block only require writing two blocks (data and parity) rather than involving all disks. Note that new disks can be added seamlessly to the system provided they are initialized to all zeros, as this does not affect the parity results.
6. ***Raid Level 5*** **-** This level is similar to level 4, except the parity blocks are distributed over all disks, thereby more evenly balancing the load on the system. For any given block on the disk(s), one of the disks will hold the parity information for that block and the other N-1 disks will hold the data. Note that the same disk cannot hold both data and parity for the same block, as both would be lost in the event of a disk crash.
7. ***Raid Level 6*** **-** This level extends raid level 5 by storing multiple bits of error-recovery codes, (such as the **Reed-Solomon codes**), for each bit position of data, rather than a single parity bit. In the example shown below 2 bits of ECC are stored for every 4 bits of data, allowing data recovery in the face of up to two simultaneous disk failures. Note that this still involves only 50% increase in storage needs, as opposed to 100% for simple mirroring which could only tolerate a single disk failure.

There are also two RAID levels which combine RAID levels 0 and 1 (striping and mirroring) in different combinations, designed to provide both performance and reliability at the expense of increased cost.

- **RAID level 0 + 1** disks are first striped, and then the striped disks mirrored to another set. This level generally provides better performance than RAID level 5.
- **RAID level 1 + 0** mirrors disks in pairs, and then stripes the mirrored pairs. The storage capacity, performance, etc. are all the same, but there is an advantage to this approach in the event of multiple disk failures, as illustrated below:.
  - In diagram (a) below, the 8 disks have been divided into two sets of four, each of which is striped, and then one stripe set is used to mirror the other set.
    - If a single disk fails, it wipes out the entire stripe set, but the system can keep on functioning using the remaining set.
    - However if a second disk from the other stripe set now fails, then the entire system is lost, as a result of two disk failures.
  - In diagram (b), the same 8 disks are divided into four sets of two, each of which is mirrored, and then the file system is striped across the four sets of mirrored disks.

- If a single disk fails, then that mirror set is reduced to a single disk, but the system rolls on, and the other three mirror sets continue mirroring.
- Now if a second disk fails, (that is not the mirror of the already failed disk), then another one of the mirror sets is reduced to a single disk, but the system can continue without data loss.
- In fact the second arrangement could handle as many as four simultaneously failed disks, as long as no two of them were from the same mirror pair.

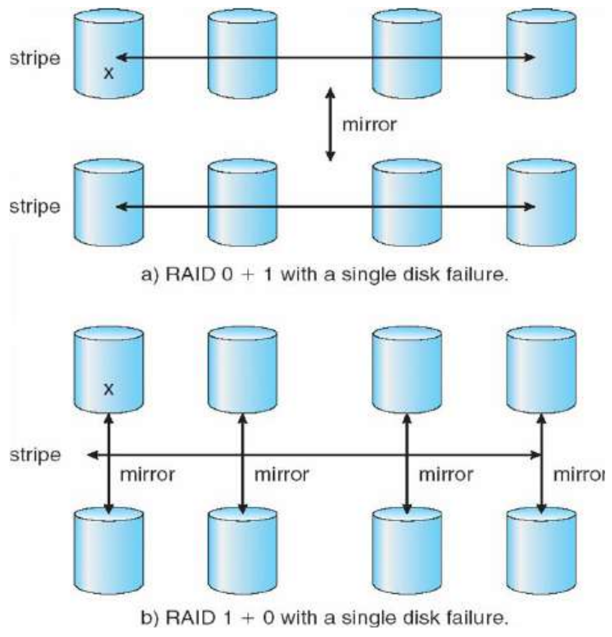

stripe
x
mirror
stripe
a) RAID 0 + 1 with a single disk failure.
x
stripe
mirror
mirror
mirror
mirror
b) RAID 1 + 0 with a single disk failure.

Figure: RAID 0+1 and 1+0

Selecting a RAID Level

- Trade-offs in selecting the optimal RAID level for a particular application include cost, volume of data, need for reliability, need for performance, and rebuild time, the latter of which can affect the likelihood that a second disk will fail while the first failed disk is being rebuilt.
- Other decisions include how many disks are involved in a RAID set and how many disks to protect with a single parity bit. More disks in the set increases performance but increases cost. Protecting more disks per parity bit saves cost, but increases the likelihood that a second disk will fail before the first bad disk is repaired.

Extensions

- RAID concepts have been extended to tape drives (e.g. striping tapes for faster backups or parity checking tapes for reliability), and for broadcasting of data.

Problems with RAID

- RAID protects against physical errors, but not against any number of bugs or other errors that could write erroneous data.
- ZFS adds an extra level of protection by including data block checksums in all inodes along with the pointers to the data blocks. If data are mirrored and one copy has the correct checksum and the other does not, then the data with the bad checksum will be replaced with a copy of the data with the good checksum. This increases reliability greatly over RAID alone, at a cost of a performance hit that is acceptable because ZFS is so fast to begin with.

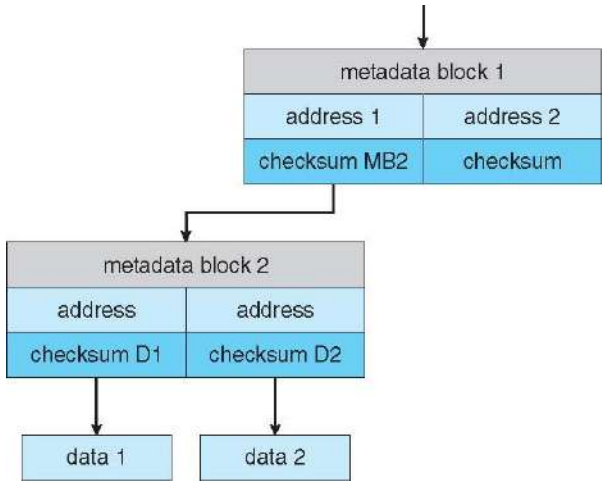
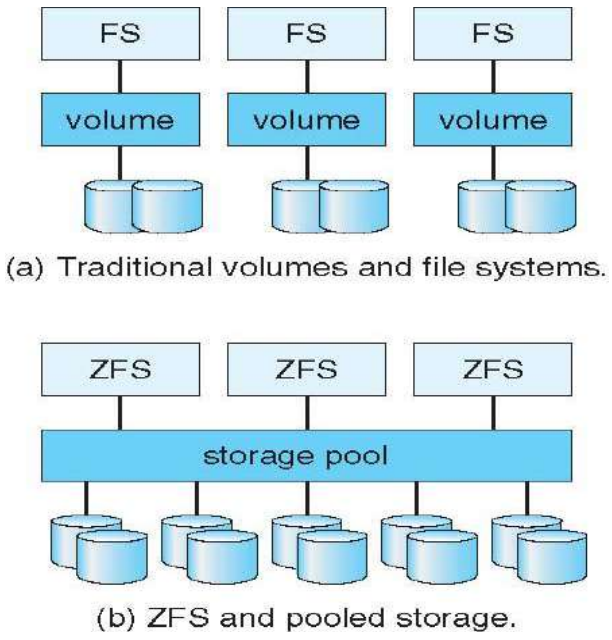


Figure: ZFS checksums all metadata and data.

- Another problem with traditional filesystems is that the sizes are fixed, and relatively difficult to change. Where RAID sets are involved it becomes even harder to adjust filesystem sizes, because a filesystem cannot span across multiple filesystems.
- ZFS solves these problems by pooling RAID sets, and by dynamically allocating space to filesystems as needed. Filesystem sizes can be limited by quotas, and space can also be reserved to guarantee that a filesystem will be able to grow later, but these parameters can be changed at any time by the filesystem's owner. Otherwise filesystems grow and shrink dynamically as needed.



FS
FS
FS
volume
volume
volume
(a) Traditional volumes and file systems.
ZFS
ZFS
ZFS
storage pool
(b) ZFS and pooled storage.

12.8 Stable-Storage Implementation

- The concept of stable storage (first presented in chapter 6) involves a storage medium in which data is ***never*** lost, even in the face of equipment failure in the middle of a write operation.
- To implement this requires two (or more) copies of the data, with separate failure modes.
- An attempted disk write results in one of three possible outcomes:
    1. The data is successfully and completely written.
    2. The data is partially written, but not completely. The last block written may be garbled.
    3. No writing takes place at all.
- Whenever an equipment failure occurs during a write, the system must detect it, and return the system back to a consistent state. To do this requires two physical blocks for every logical block, and the following procedure:
    1. Write the data to the first physical block.
    2. After step 1 had completed, then write the data to the second physical block.
    3. Declare the operation complete only after both physical writes have completed successfully.
- During recovery the pair of blocks is examined.
    - If both blocks are identical and there is no sign of damage, then no further action is necessary.
    - If one block contains a detectable error but the other does not, then the damaged block is replaced with the good copy. (This will either undo the operation or complete the operation, depending on which block is damaged and which is undamaged.)
    - If neither block shows damage but the data in the blocks differ, then replace the data in the first block with the data in the second block. (Undo the operation.)
- Because the sequence of operations described above is slow, stable storage usually includes NVRAM as a cache, and declares a write operation complete once it has been written to the NVRAM.

12.9 Tertiary-Storage Structure

- Primary storage refers to computer memory chips; Secondary storage refers to fixed-disk storage systems (hard drives); And ***Tertiary Storage*** refers to ***removable media,*** such as tape drives, CDs, DVDs, and to a lesser extend floppies, thumb drives, and other detachable devices.
- Tertiary storage is typically characterized by large capacity, low cost per MB, and slows access times, although there are exceptions in any of these categories.
- Tertiary storage is typically used for backups and for long-term archival storage of completed work. Another common use for tertiary storage is to swap large little-used files (or groups of files) off of the hard drive, and then swap them back in as needed in a fashion similar to secondary storage providing swap space for primary storage..

Tertiary-Storage Devices

Removable Disks

- Removable magnetic disks (e.g. floppies) can be nearly as fast as hard drives, but are at greater risk for damage due to scratches. Variations of removable magnetic disks up to a GB or more in capacity have been developed. (Hot-swappable hard drives?)
- A ***magneto-optical*** disk uses a magnetic disk covered in a clear plastic coating that protects the surface.
    - The heads sit a considerable distance away from the magnetic surface, and as a result do not have enough magnetic strength to switch bits ***at normal room temperature.***
    - For writing, a laser is used to heat up a specific spot on the disk, to a temperature at which the weak magnetic field of the write head is able to flip the bits.
    - For reading, a laser is shined at the disk, and the ***Kerr effect*** causes the polarization of the light to become rotated either clockwise or counter-clockwise depending on the orientation of the magnetic field.
- ***Optical disks*** do not use magnetism at all, but instead use special materials that can be altered (by lasers) to have relatively light or dark spots.
    - For example the ***phase-change disk*** has a material that can be frozen into either a crystalline or an amorphous state, the latter of which is less transparent and reflects less light when a laser is bounced off a reflective surface under the material.
        - Three powers of lasers are used with phase-change disks: (1) a low power laser is used to read the disk, without effecting the materials. (2) A medium power erases the disk, by melting and re-freezing the medium into a crystalline state, and (3) a high power writes to the disk by melting the medium and re-freezing it into the amorphous state.
        - The most common examples of these disks are ***re-writable*** CD-RWs and DVD-RWs.
- An alternative to the disks described above are ***Write-Once Read-Many, WORM*** drives.
    - The original version of WORM drives involved a thin layer of aluminum sandwiched between two protective layers of glass or plastic.
        - Holes were burned in the aluminum to write bits.
        - Because the holes could not be filled back in, there was no way to re-write to the disk. (Although data could be erased by burning more holes.)

    - WORM drives have important legal ramifications for data that must be stored for a very long time and must be provable in court as unaltered since it was originally written. (Such as long-term storage of medical records.)
    - Modern CD-R and DVD-R disks are examples of WORM drives that use organic polymer inks instead of an aluminum layer.
- Read-only disks are similar to WORM disks, except the bits are pressed onto the disk at the factory, rather than being burned on one by one.

Tapes

- Tape drives typically cost more than disk drives, but the cost per MB of the tapes themselves is lower.
- Tapes are typically used today for backups, and for enormous volumes of data stored by certain scientific establishments. (E.g. NASA's archive of space probe and satellite imagery, which is currently being downloaded from numerous sources faster than anyone can actually look at it.)
- Robotic tape changers move tapes from drives to archival tape libraries upon demand.
- (Never underestimate the bandwidth of a station wagon full of tapes rolling down the highway!)

Future Technology

- ***Solid State Disks, SSDs,*** are becoming more and more popular.
- ***Holographic storage*** uses laser light to store images in a 3-D structure, and the entire data structure can be transferred in a single flash of laser light.
- ***Micro-Electronic Mechanical Systems, MEMS,*** employs the technology used for computer chip fabrication to create VERY tiny little machines. One example packs 10,000 read-write heads within a square centimeter of space, and as media are passed over it, all 10,000 heads can read data in parallel.

Operating-System Support

- The OS must provide support for tertiary storage as removable media, including the support to transfer data between different systems.

Application Interface

- File systems are typically not stored on tapes. (It might be technically possible, but it is impractical.)
- Tapes are also not low-level formatted, and do not use fixed-length blocks. Rather data is written to tapes in variable length blocks as needed.

- Tapes are normally accessed as raw devices, requiring each application to determine how the data is to be stored and read back. Issues such as header contents and ASCII versus binary encoding (and byte-ordering) are generally application specific.
- Basic operations supported for tapes include locate( ), read( ), write( ), and read_position( ).
- (Because of variable length writes), writing to a tape erases all data that follows that point on the tape.
    - Writing to a tape places the End of Tape (EOT) marker at the end of the data written.
    - It is not possible to locate( ) to any spot past the EOT marker.

File Naming

- File naming conventions for removable media are not entirely uniquely specific, nor are they necessarily consistent between different systems. (Two removable disks may contain files with the same name, and there is no clear way for the naming system to distinguish between them.)
- Fortunately music CDs have a common format, readable by all systems. Data CDs and DVDs have only a few format choices, making it easy for a system to support all known formats.

Hierarchical Storage Management

- Hierarchical storage involves extending file systems out onto tertiary storage, swapping files from hard drives to tapes in much the same manner as data blocks are swapped from memory to hard drives.
- A placeholder is generally left on the hard drive, storing information about the particular tape (or other removable media) on which the file has been swapped out to.
- A robotic system transfers data to and from tertiary storage as needed, generally automatically upon demand of the file(s) involved.

Performance Issues

Speed

- ***Sustained Bandwidth*** is the rate of data transfer during a large file transfer, once the proper tape is loaded and the file located.
- ***Effective Bandwidth*** is the effective overall rate of data transfer, including any overhead necessary to load the proper tape and find the file on the tape.
- ***Access Latency*** is all of the accumulated waiting time before a file can be actually read from tape. This includes the time it takes to find the file on

the tape, the time to load the tape from the tape library, and the time spent waiting in the queue for the tape drive to become available.

- Clearly tertiary storage access is much slower than secondary access, although removable disks (e.g. a CD jukebox) have somewhat faster access than a tape library.

Reliability

- Fixed hard drives are generally more reliable than removable drives, because they are less susceptible to the environment.
- Optical disks are generally more reliable than magnetic media.
- A fixed hard drive crash can destroy all data, whereas an optical drive or tape drive failure will often not harm the data media, (and certainly can't damage any media not in the drive at the time of the failure.)
- Tape drives are mechanical devices, and can wear out tapes over time, (as the tape head is generally in much closer physical contact with the tape than disk heads are with platters.)
    - Some drives may only be able to read tapes a few times whereas other drives may be able to re-use the same tapes millions of times.
    - Backup tapes should be read after writing, to verify that the backup tape is readable. (Unfortunately that may have been the LAST time that particular tape was readable, and the only way to be sure is to read it again, . . .)
    - Long-term tape storage can cause degradation, as magnetic fields "drift" from one layer of tape to the adjacent layers. Periodic fast-forwarding and rewinding of tapes can help, by changing which section of tape lays against which other layers.

Cost

- The cost per megabyte for removable media is its strongest selling feature, particularly as the amount of storage involved (i.e. the number of tapes, CDs, etc) increases.
- However the cost per megabyte for hard drives has dropped more rapidly over the years than the cost of removable media, such that the currently most cost-effective backup solution for many systems is simply an additional (external) hard drive.
- (One good use for old unwanted PCs is to put them on a network as a backup server and/or print server. The downside to this backup solution is that the backups are stored on-site with the original data, and a fire, flood, or burglary could wipe out both the original data and the backups.)

UNIT-4-PART III-I/O SYSTEMS

I/O Systems

- Explore the structure of an operating system’s I/O subsystem
- Discuss the principles of I/O hardware and its complexity
- Provide details of the performance aspects of I/O hardware and software

I/O Hardware

The role of the operating system in computer I/O is to manage and control I/O operations and I/O devices. A device communicates with a computer system by sending signals over a cable or even through the air.

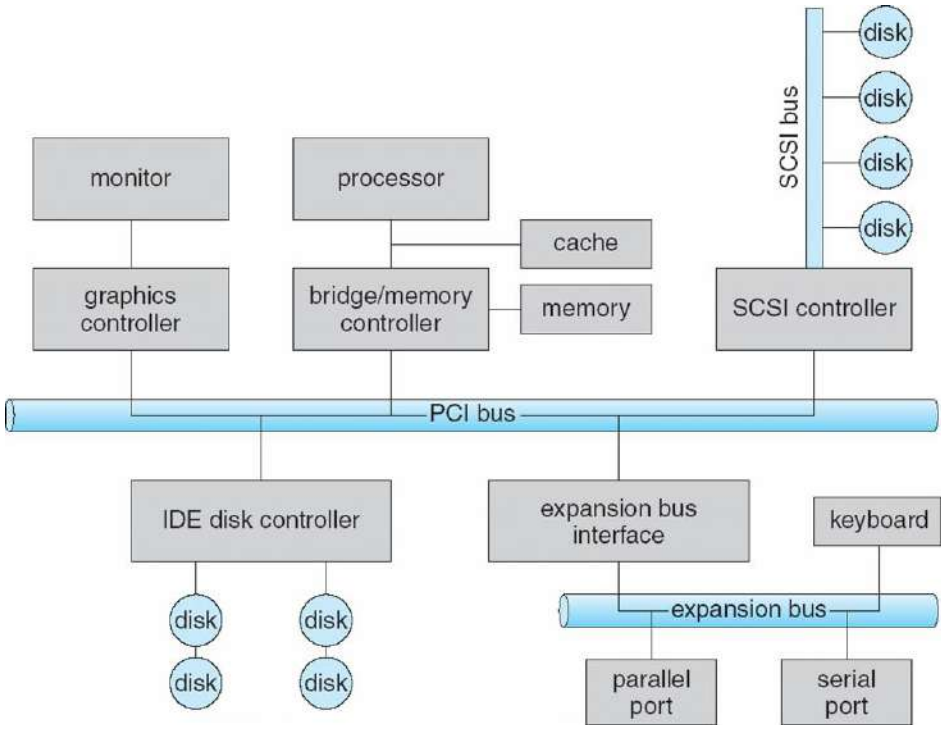
Port: The device communicates with the machine via a connection point (or port), for example, a serial port.

Bus: If one or more devices use a common set of wires, the connection is called a bus.When device Daisy chain: Device _A‘ has a cable that plugs into device _B‘, and device _B‘ has a cable that plugs into device _C‘, and device _C‘ plugs into a port on the computer, this arrangement is called a **daisy chain**. A daisy chain usually operates as a bus.

PC bus structure:

A PCI bus that connects the processor-memory subsystem to the fast devices, and an expansion bus that connects relatively slow devices such as the keyboard and serial and parallel ports. In the upper-right portion of the figure, four disks are connected together a SCSI bus plugged into a SCSI controller.

A **controller or host adapter** is a collection of electronics that can operate a port, a bus, or a device. A serial-port controller is a simple device controller. It is a single chip in the computer that controls the signals on the wires of a serial port. By contrast, a SCSI bus controller is notsimple. Because the SCSI protocol is complex, the SCSI bus controller is often implemented as a separate circuit board. It typically contains a processor, microcode, and some private memory.Some devices have their own built-in controllers



How can the processor give commands and data to a controller to accomplish an I/O transfer?

- Direct I/O instructions
- Memory-mapped I/O

Direct I/O instructions

Use special I/O instructions that specify the transfer of a byte or word to an I/O port address.

The I/O instruction triggers bus lines to select the proper device and to move bits into or out of a device register .

Memory-mapped I/O

The device-control registers are mapped into the address space of the processor. The CPU executes I/O requests using the standard data-transfer instructions to read and write the device-control registers. An I/O port typically consists of four registers: status, control, data-in, and data-out registers.

• the ***data-in register*** is read by the host to get input from the device.
• the ***data-out register*** is written by the host to send output.
• the ***status register*** has bits read by the host to ascertain the status of the device, such as idle, ready for input, busy, error, transaction complete, etc.

• the ***control register*** has bits written by the host to issue commands or to change settings of the device such as parity checking, word length, or full- versus half-duplex operation

Polling

Interaction between the host and a controller

The controller sets the busy bit when it is busy working, and clears the busy bit when it is ready to accept the next command.

The host sets the command ready bit when a command is available for the controller to execute. Coordination between the host & the controller is done by handshaking as follows:

1. The host repeatedly reads the busy bit until that bit becomes clear.

2. The host sets the write bit in the command register and writes a byte into the data-out register.

3. The host sets the command-ready bit.

4. When the controller notices that the command-ready bit is set, it sets the busy bit.

5. The controller reads the command register and sees the write command. It reads the data-out register to get the byte, and does the I/O to the device.

6. The controller clears the command-ready bit, clears the error bit in the status register to indicate that the device I/O succeeded, and clears the busy bit to indicate that it is finished.

In step 1, the host is **—busy-waiting or polling**: It is in a loop, reading the status register over and over until the busy bit becomes clear.

Interrupts

The CPU hardware has a wire called the —interrupt-request lin. The basic interrupt mechanism works as follows;

1. Device controller raises an interrupt by asserting a signal on the interrupt request line.

2. The CPU catches the interrupt and dispatches to the interrupt handler and

3. The handler clears the interrupt by servicing the device. Two interrupt request lines:

1. **Nonmaskable interrupt**: which is reserved for events such as unrecoverable memory errors? 2. **Maskable interrupt**: Used by device controllers to request service

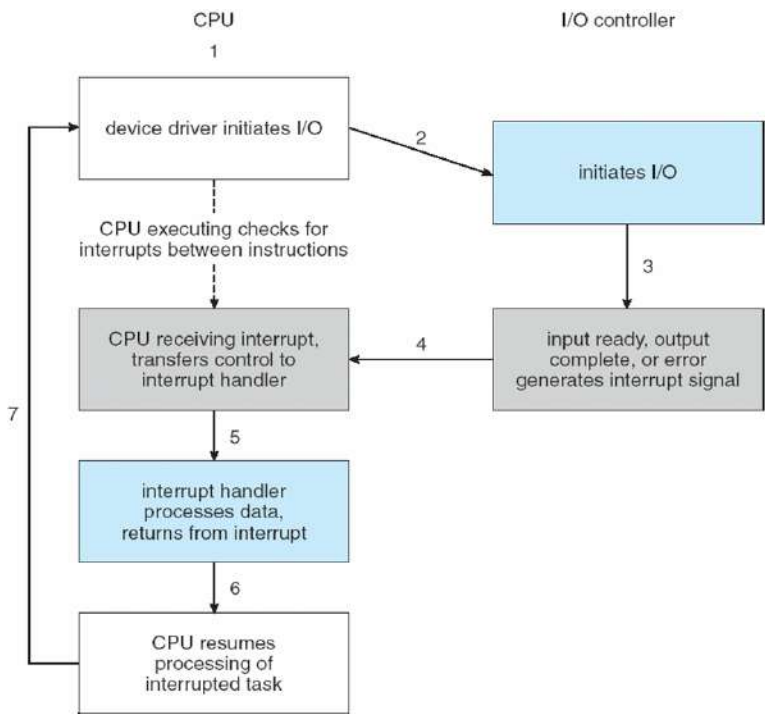
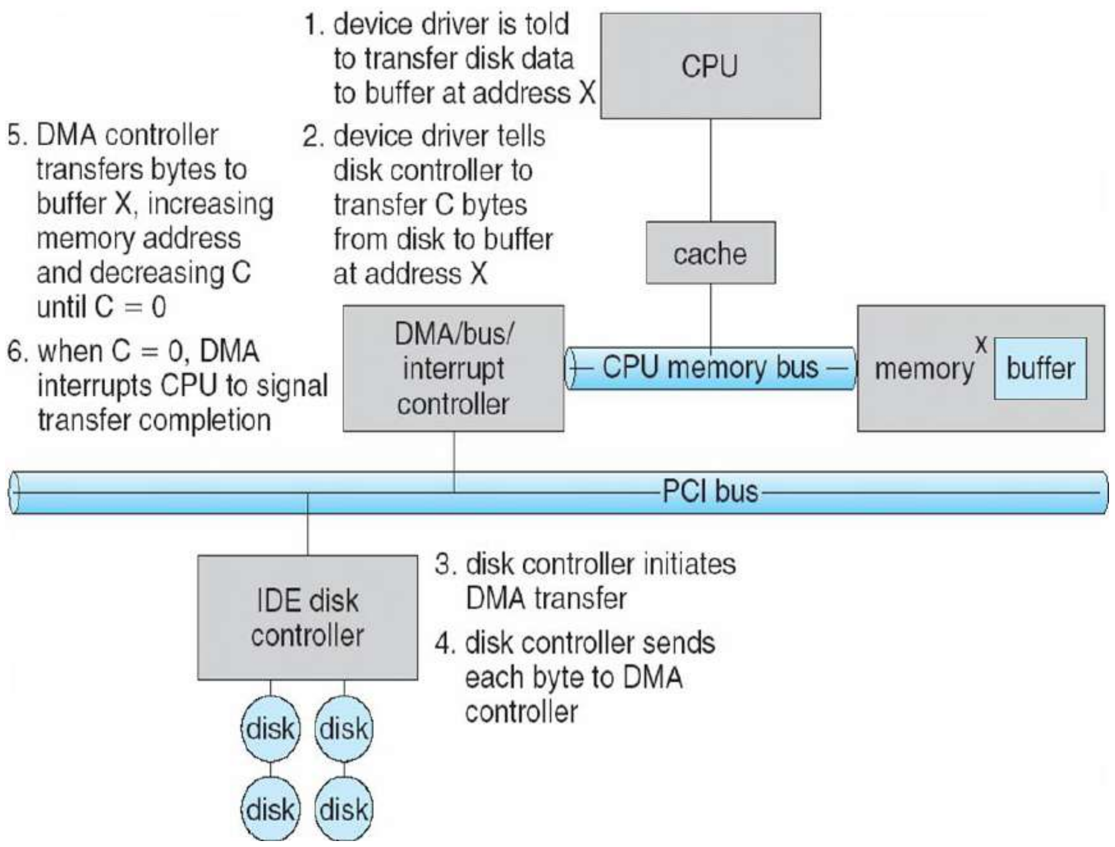

CPU
I/O controller
1
device driver initiates I/O
2
initiates I/O
CPU executing checks for interrupts between instructions
3
CPU receiving interrupt, transfers control to interrupt handler
4
input ready, output complete, or error generates interrupt signal
7
5
interrupt handler processes data, returns from interrupt
6
CPU resumes processing of interrupted task

Figure: - Interrupt-driven I/O cycle

Direct Memory Access (DMA)

In general it is tough for the CPU to do the large transfers between the memory buffer &disk; because it is already equipped with some other tasks, then this will create overhead. So a special-purpose processor called a direct memory- access **(DMA)** controller is used.

• While the DMA transfer is going on the CPU does not have access to the PCI bus (including main memory), but it does have access to its internal registers and primary and secondary caches.
• DMA can be done in terms of either physical addresses or virtual addresses that are mapped to physical addresses. The latter approach is known as ***Direct Virtual Memory Access, DVMA,*** and allows direct data transfer from one memory-mapped device to another without using the main memory chips.
• Direct DMA access by user processes can speed up operations, but is generally forbidden by modern systems for security and protection reasons. (i.e. DMA is a kernel-mode operation.


1. device driver is told to transfer disk data to buffer at address X
CPU
5. DMA controller transfers bytes to buffer X, increasing memory address and decreasing C until C = 0
2. device driver tells disk controller to transfer C bytes from disk to buffer at address X
cache
6. when C = 0, DMA interrupts CPU to signal transfer completion
DMA/bus/interrupt controller
CPU memory bus
memory
X
buffer
PCI bus
IDE disk controller
3. disk controller initiates DMA transfer
4. disk controller sends each byte to DMA controller
disk
disk
disk
disk

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

1. Character-stream or block
2. Sequential or random-access
3. Sharable or dedicated
4. Speed of operation
5. read-write, read only, or writes only

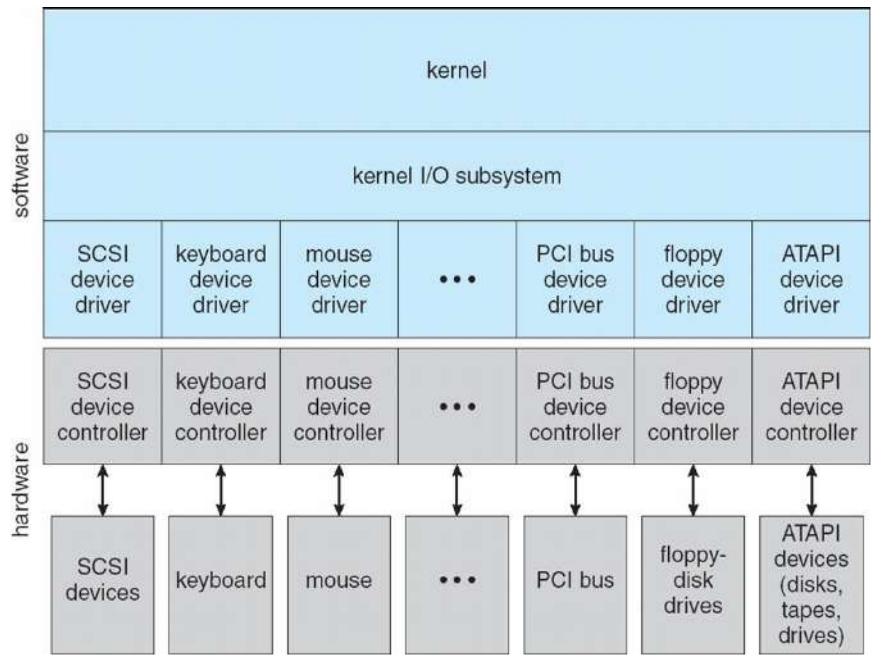
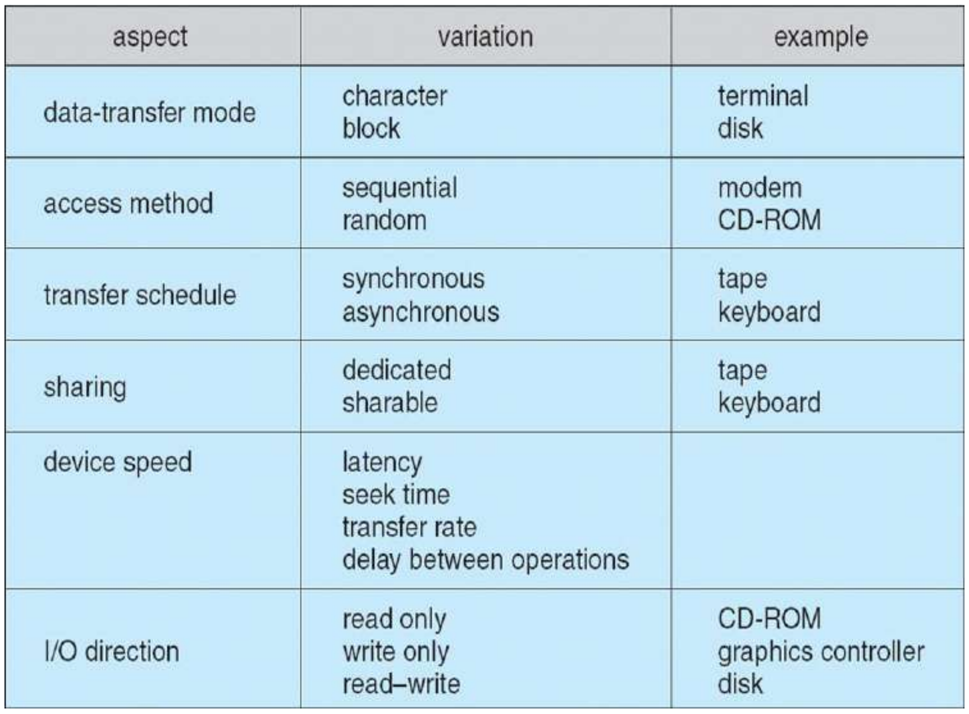


Figure: A kernel I/O structure

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-device: The block-device interface captures all the aspects necessary for accessing disk drives and other block-oriented devices. The device should understand the commands such as read () & write (), and if it is a random access device, it has a seek() command to specify which block to transfer next.

Applications normally access such a device through a file-system interface. The OS itself, and special applications such as database-management systems, may prefer to access a block device as a simple linear array of blocks. This mode of access is sometimes called **raw I/O.**

Memory-mapped file access can be layered on top of block-device drivers. Rather than offering read and write operations, a memory-mapped interface provides access to disk storage via an array of bytes in main memory.

Character Devices:

A keyboard is an example of a device that is accessed through a character stream interface. The basic system calls in this interface enable an application to get() or put() one character.On top of this interface, libraries can be built that offer line-at-a-time access, with buffering and editing services. This style of access is convenient for input devices where it produce input "spontaneously". This access style is also good for output devices such as printers or audio boards, which naturally fit the concept of a linear stream of bytes.

Network Devices

Because the performance and addressing characteristics of network I/O differsignificantly from those of disk I/O, most operating systems provide a network I/O interface that is different from the read0 -write() -seek() interface used for disks.

Windows NT provides one interface to the network interface card, and a second interface to the network protocols. In UNIX, we find half-duplex pipes, full-duplex FIFOs, full-duplex STREAMS, message queues and sockets.

Clocks and Timers

Most computers have hardware clocks and timers that provide three basic functions:

1. Give the current time

2. Give the elapsed time

3. Set a timer to trigger operation X at time T

These functions are used by the operating system & also by time sensitive applications.

Blocking and Non-blocking I/O (or) synchronous & asynchronous:

Blocking I/O: When an application issues a blocking system call;

- ✓ The execution of the application is suspended.
- ✓ The application is moved from the operating system's run queue to a wait queue.
- ✓ After the system call completes, the application is moved back to the run queue, where it is eligible to resume execution, at which time it will receive the values returned by the system call

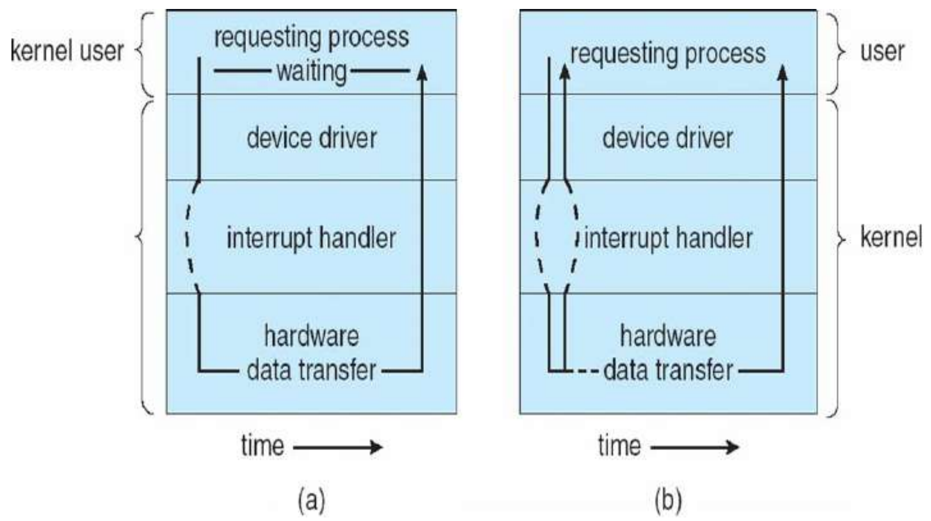


Figure: Two I/O methods: (a) synchronous and (b) asynchronous.

Non-blocking I/O: Some user-level processes need non-blocking I/O.

Examples: 1. User interface that receives keyboard and mouse input while processing and displaying data on the screen.

2. Video application that reads frames from a file on disk while simultaneously decompressing and displaying the output on the display.

Kernel I/O Subsystem Kernels provide many services related to I/O.

One way that the I/O subsystem improves the efficiency of the computer is by scheduling I/O operations.

Another way is by using storage space in main memory or on disk, via techniques called buffering, caching, and spooling.

Services include;

I/O Scheduling:

To determine a good order in which to execute the set of I/O requests.

Uses:

a) It can improve overall system performance,

b) It can share device access fairly among processes, and

c) It can reduce the average waiting time for 1/0 to complete.

Implementation: OS developers implement scheduling by maintaining a ―queue of requests for each device.

1. When an application issues a blocking I/O system call,

2. The request is placed on the queue for that device.

3. The I/O scheduler rearranges the order of the queue to improve the overall system efficiency and the average response time experienced by applications.

Buffering:

Buffer: A memory area that stores data while they are transferred between two devices or between a device and an application.

Reasons for buffering:

a) To cope with a speed mismatch between the producer and consumer of a data stream.

b) To adapt between devices that have different data-transfer sizes.

c) To support copy semantics for application I/O.

Copy semantics: Suppose that an application has a buffer of data that it wishes to write to disk. It calls the write () system call, providing a pointer to the buffer and an integer specifying the number of bytes to write.

After the system call returns, what happens if the application changes the contents of the buffer? With copy semantics, the version of the data written to disk is guaranteed to be the version at the time of the application system call, independent of any subsequent changes in the application's buffer.

A simple way that the operating system can guarantee copy semantics is for the write()system call to copy the application data into a kernel buffer before returning control to the application. The disk write is performed from the kernel buffer, so that subsequent changes to theapplication buffer have no effect.

Caching A cache is a region of fast memory that holds copies of data.

Access to the cached copy is more efficient than access to the original

Cache vs buffer: A buffer may hold the only existing copy of a data item, whereas a cache just holds a copy on faster storage of an item that resides elsewhere.

When the kernel receives a file I/O request,

1. The kernel first accesses the buffer cache to see whether that region of the file is already available in main memory.

2. If so, a physical disk I/O can be avoided or deferred. Also, disk writes are accumulated in the buffer cache for several seconds, so that large transfers are gathered to allow efficient write schedules.

Spooling and Device Reservation:

Spool: A buffer that holds output for a device, such as a printer, that cannot accept interleaved data streams.

A printer can serve only one job at a time, several applications may wish to print their output concurrently, without having their output mixed together

The os provides a control interface that enables users and system administrators;

a) To display the queue,

b) To remove unwanted jobs before those jobs print,

c) To suspend printing while the printer is serviced, and so on. Device reservation - provides exclusive access to a device System calls for allocation and de-allocation

Watch out for deadlock

Error Handling:

An operating system that uses protected memory can guard against many kinds of hardware and application errors.

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

I/O Protection

- ✓ User process may accidentally or purposefully attempt to disrupt normal operation via illegal I/O instructions.
- ✓ All I/O instructions defined to be privileged
- ✓ .I/O must be performed via system calls
- ✓ Memory-n tected too.

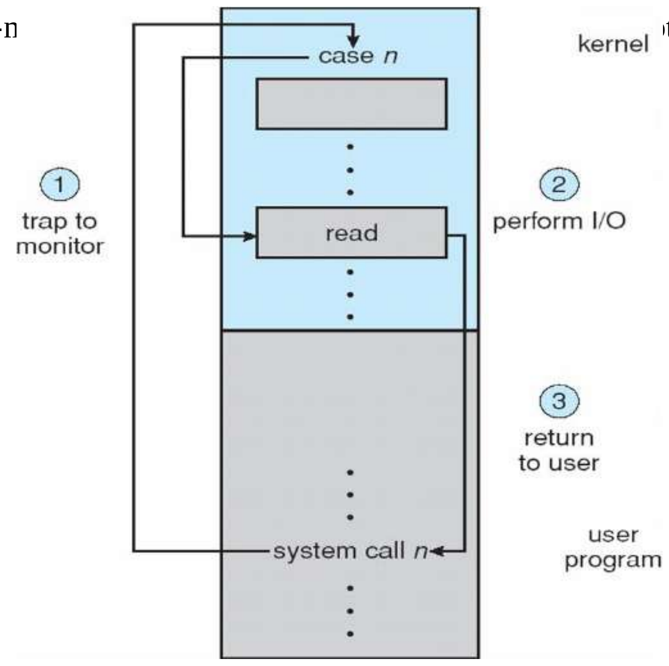
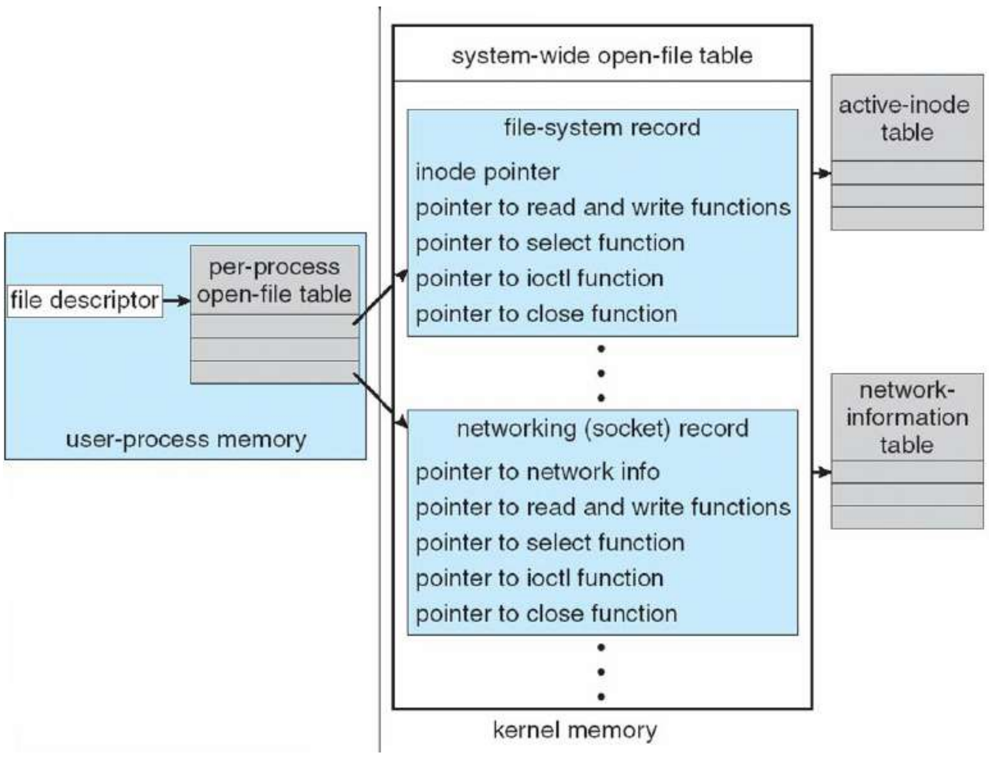


Figure: Use of a system call to perform I/O.

Kernel Data Structures

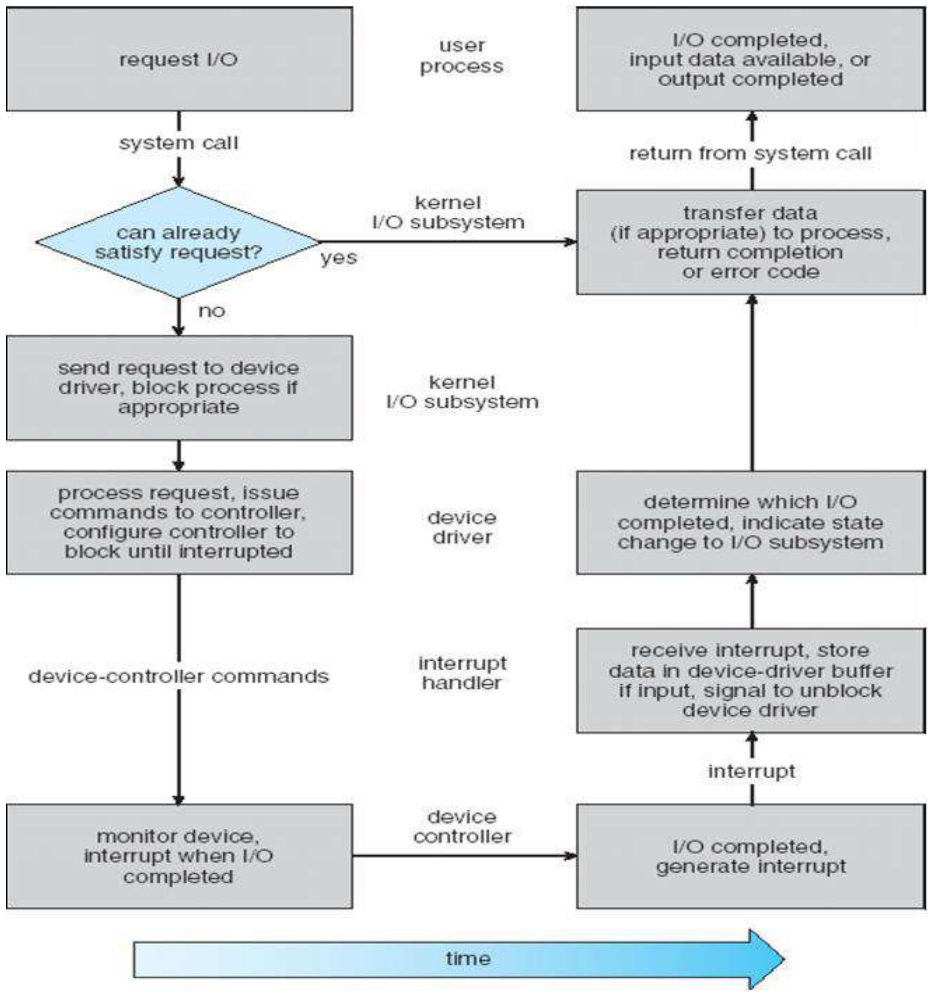
Kernel keeps state info for I/O components, including open file tables, network connections, character device state Many, many complex data structures to track buffers, memory allocation, “dirty” blocks .Some use object-oriented methods and message passing to implement I/O


system-wide open-file table
file-system record
inode pointer
pointer to read and write functions
pointer to select function
pointer to ioctl function
pointer to close function
networking (socket) record
pointer to network info
pointer to read and write functions
pointer to select function
pointer to ioctl function
pointer to close function
active-inode table
network-information table
file descriptor
per-process open-file table
user-process memory
kernel memory

Transforming I/O Requests to Hardware Operations

Consider reading a file from disk for a process:

- ✓ Determine device holding file
- ✓ Translate name to device representation
- ✓ Physically read data from disk into buffer
- ✓ Make data available to requesting process
- ✓ Return control to process


request I/O
user process
I/O completed, input data available, or output completed
system call
return from system call
can already satisfy request?
yes
kernel I/O subsystem
transfer data (if appropriate) to process, return completion or error code
no
send request to device driver, block process if appropriate
kernel I/O subsystem
process request, issue commands to controller, configure controller to block until interrupted
device driver
determine which I/O completed, indicate state change to I/O subsystem
device-controller commands
interrupt handler
receive interrupt, store data in device-driver buffer if input, signal to unblock device driver
interrupt
monitor device, interrupt when I/O completed
device controller
I/O completed, generate interrupt
time

STREAMS:

STREAM: a full-duplex communication channel between a user-level process and a device in Unix System V and beyond

<u>A STREAM consists of:</u>

a) STREAM head interfaces with the user process

b) Driver end interfaces with the device

c) 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
- ✓ Modules provide the functionality of STREAMS processing and they are pushed onto a stream using the ioct () system call.

Flow control:

Because messages are exchanged between queues in adjacent modules, a queue in one module may overflow an adjacent queue. To prevent this from occurring, a queue may support flow control.

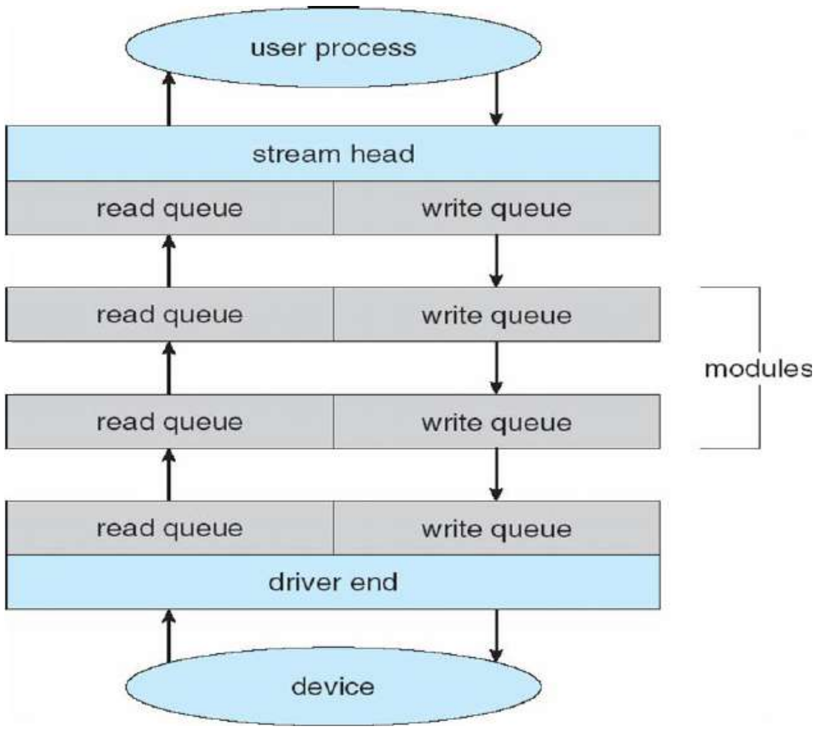


Figure: The STREAMS structure.

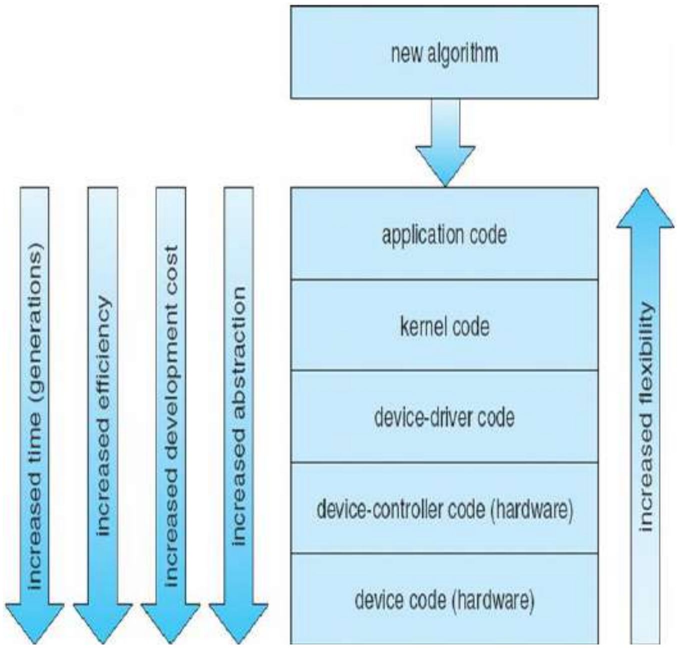
Performance:

I/O a major factor in system performance:

- ✓ Heavy demands on CPU to execute device driver, kernel I/O code. So contextswitches occur due to interrupts.
- ✓ Interrupt handling is a relatively expensive task: Each interrupt causes the system to perform a state change, to execute the interrupt handler & then to restore state
- ✓ Network traffic especially stressful.
- ✓ Systems use separate **—front-end processors”** for terminal I/O, to reduce theinterrupt burden on the main CPU.

We can employ several principles to improve the efficiency of I/O:

1. Reduce the number of context switches
2. Reduce the number of times that data must be copied in memory while passing between device and application.
3. Reduce the frequency of interrupts by using large transfers, smart controllers & polling.
4. Increase concurrency by using DMA-knowledgeable controllers or channels to offload simple data copying from the CPU.
5. Move processing primitives into hardware, to allow their operation in device controllers concurrent with the CPU and bus operation.
6. Balance CPU, memory subsystem, bus, and I/O performance, because an overload in any one area will cause idleness in others.



Unit-V Part-I System Protection

Protection

- Discuss the goals and principles of protection in a modern computer system
- Explain how protection domains combined with an access matrix are used to specify the resources a process may access
- Examine capability and language-based protection systems

Goals of Protection

Operating system consists of a collection of objects, hardware or softwarenEach object has a unique name and can be accessed through a well-defined set of operationsnProtection problem - ensure that each object is accessed correctly and only by those processes that are allowed to do so.

- To ensure that each shared resource is used only in accordance with system *policies*, which may be set either by system designers or by system administrators.
- To ensure that errant programs cause the minimal amount of damage possible.
- Note that protection systems only provide the *mechanisms* for enforcing policies and ensuring reliable systems. It is up to administrators and users to implement those mechanisms effectively.

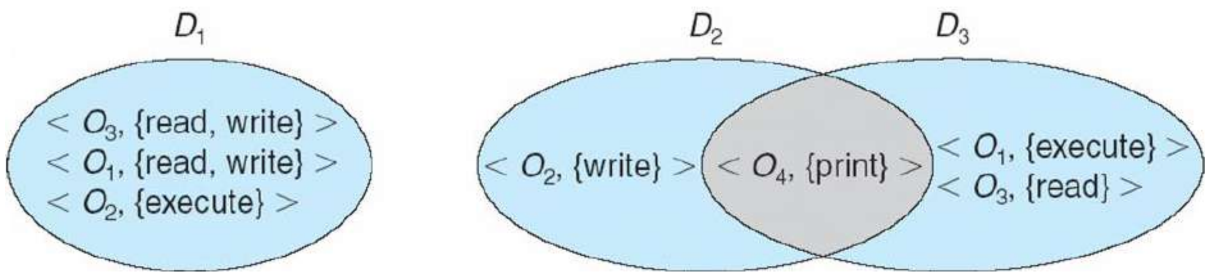
Principles of Protection

- The ***principle of least privilege*** dictates that programs, users, and systems be given just enough privileges to perform their tasks.
- This ensures that failures do the least amount of harm and allow the least of harm to be done.
- For example, if a program needs special privileges to perform a task, it is better to make it a SGID program with group ownership of "network" or "backup" or some other pseudo group, rather than SUID with root ownership. This limits the amount of damage that can occur if something goes wrong.
- Typically each user is given their own account, and has only enough privilege to modify their own files.
- The root account should not be used for normal day to day activities - The System Administrator should also have an ordinary account, and reserve use of the root account for only those tasks which need the root privileges

Domain of Protection

- A computer can be viewed as a collection of *processes* and *objects* (both HW & SW).

- The ***need to know principle*** states that a process should only have access to those objects it needs to accomplish its task, and furthermore only in the modes for which it needs access and only during the time frame when it needs access.
- The modes available for a particular object may depend upon its type.
- Access-right = <*object-name*, *rights-set*>
  where *rights-set* is a subset of all valid operations that can be performed on the object.
- Domain = set of access-rights



System consists of 2 domains:

User

Supervisor UNIX

Domain = user-id

Domain switch accomplished via file system

✓ Each file has associated with it a domain bit (setuid bit)

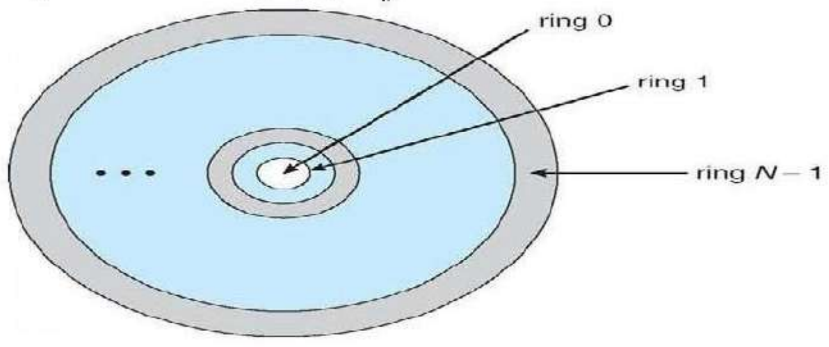
✓ When file is executed and setuid = on

Domain Implementation

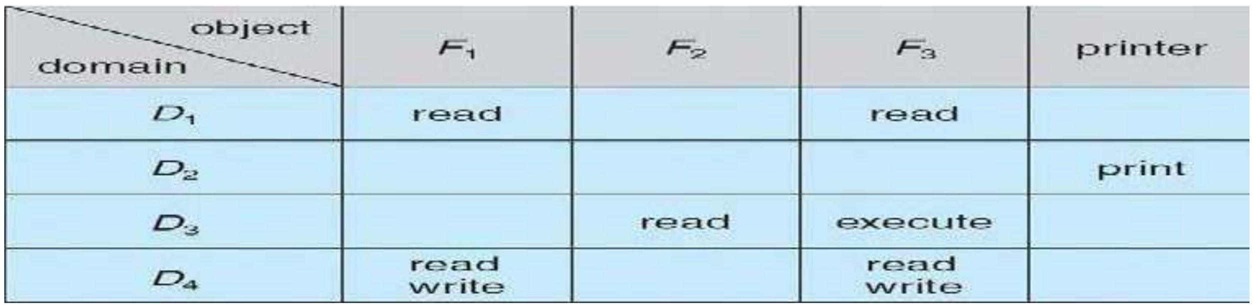
Each file has associated with it a domain bit (setuid bit)
• When file is executed and setuid = on, then user-id is set to owner of the file being executed. When execution completes user-id is reset •

Domain Implementation (MULTICS)

- Let D_i and D_j be any two domain rings
- If $j < I \Rightarrow D_i \subseteq D_j$



ACCESS MATRIX
• View protection as a matrix (*access matrix*)
• Rows represent domains
• Columns represent objects
• *Access(i, j)* is the set of operations that a process executing in Domaini can invoke on Objectj



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

- Use of Access Matrix

  - If a process in Domain Di tries to do "op" on object Oj, then "op" must be in the access matrix
  - Can be expanded to dynamic protection
  - Operations to add, delete access rights
  - Special access rights: • owner of Oi
  - copy op from Oi to Oj
  - control – Di can modify Dj access rights
  - transfer – switch from domain Di to Dj

- Access matrix design separates mechanism from policy
- Mechanism
- Operating system provides access-matrix + rules
- If ensures that the matrix is only manipulated by authorized agents and that rules are strictly enforced
- Policy
- User dictates policy
- Who can access what object and in what mode

ACCESS MATRIX

- Each column = Access-control list for one object Defines who can perform what operation.

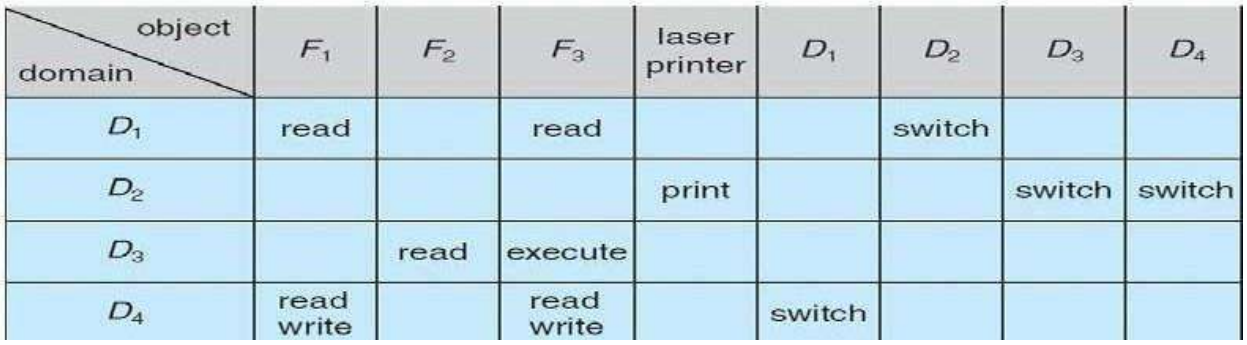
✓ Domain 1 = Read, Write Domain 2 Read Domain 3 Read

- Each Row = Capability List (like a key)

✓ Fore each domain, what operations allowed on what objects. Object 1 – Read Object 4 – Read, Write,

✓ Execute

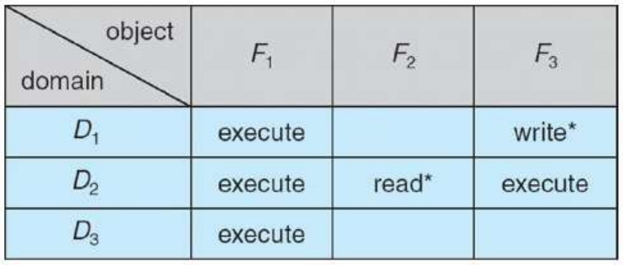
✓ Object 5 – Read, Write, Delete, Copy



object / domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch
D_3		read	execute					
D_4	read write		read write		switch			

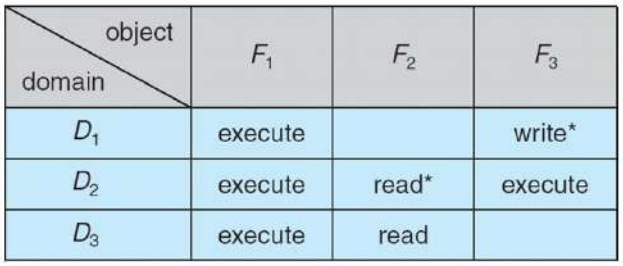
ACCESS MATRIX OF FIGURE A WITH DOMAINS AS OBJECTS

Access Matrix with *Copy* Rights



object domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute		

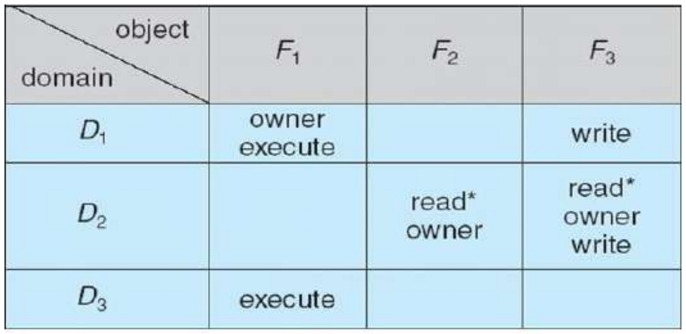
(a)



object domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute	read	

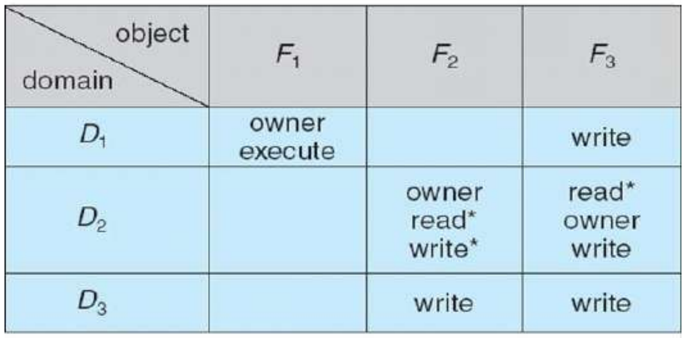
(b)

Access Matrix With *Owner* Rights



object domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		read* owner	read* owner write
D_3	execute		

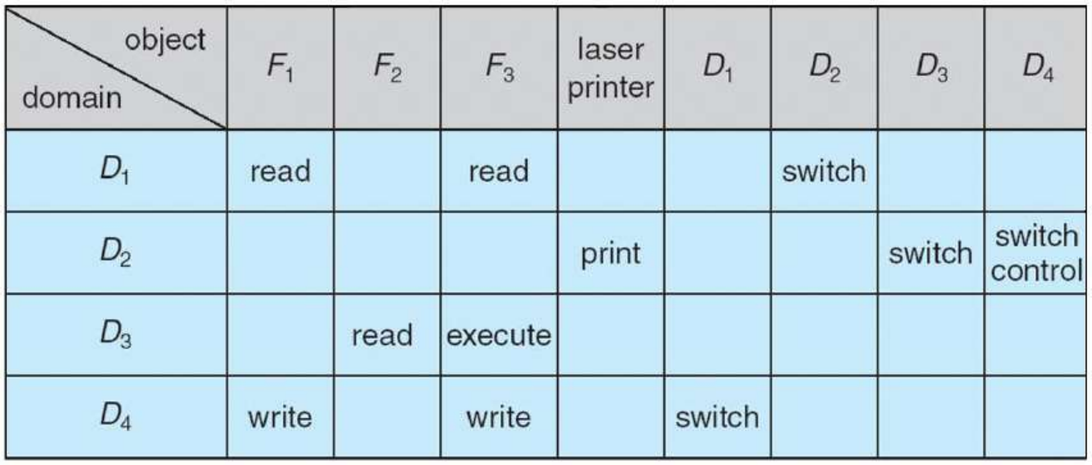
(a)



object domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		owner read* write*	read* owner write
D_3		write	write

(b)

Modified Access Matrix of Figure B



object / domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch control
D_3		read	execute					
D_4	write		write		switch			

Implementation of Access Matrix

Global Table

- The simplest approach is one big global table with < domain, object, rights > entries.
- Unfortunately this table is very large (even if sparse) and so cannot be kept in memory (without invoking virtual memory techniques.)
- There is also no good way to specify groupings - If everyone has access to some resource, then it still needs a separate entry for every domain.

Access Lists for Objects

- Each column of the table can be kept as a list of the access rights for that particular object, discarding blank entries.
- For efficiency a separate list of default access rights can also be kept, and checked first.

Capability Lists for Domains

- In a similar fashion, each row of the table can be kept as a list of the capabilities of that domain.
- Capability lists are associated with each domain, but not directly accessible by the domain or any user process.
- Capability lists are themselves protected resources, distinguished from other data in one of two ways:
    - A ***tag,*** possibly hardware implemented, distinguishing this special type of data. (Other types may be floats, pointers, booleans, etc.)

- The address space for a program may be split into multiple segments, at least one of which is inaccessible by the program itself, and used by the operating system for maintaining the process's access right capability list.

A Lock-Key Mechanism

- Each resource has a list of unique bit patterns, termed locks.
- Each domain has its own list of unique bit patterns, termed keys.
- Access is granted if one of the domain's keys fits one of the resource's locks.
- Again, a process is not allowed to modify its own keys.

Comparison

- Each of the methods here has certain Advantage or Disadvantage, depending on the particular situation and task at hand.
- Many systems employ some combination of the listed methods

Access Control

Protection can be applied to non-file resources

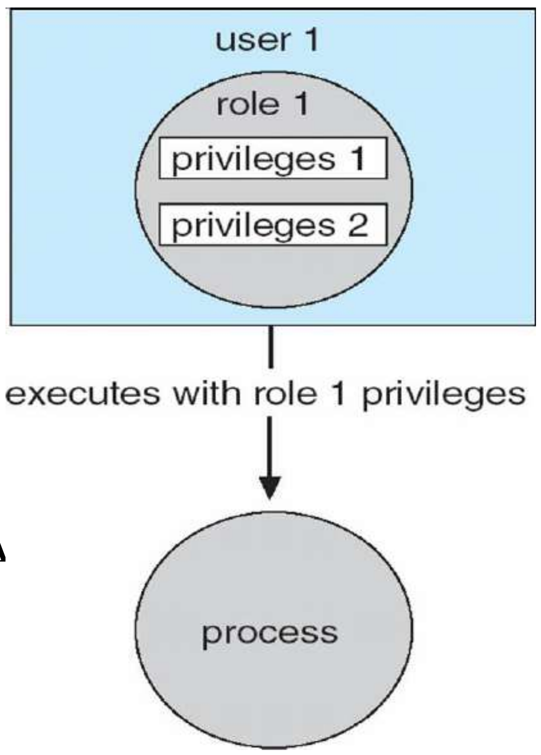
Solaris 10 provides role-based access control **(**RBAC**)** to implement least privilege

Privilege is right to execute system call or use an option within a system call

Can be assigned to processes

Users assigned roles granting access to privileges and programs

Role-based Access Control in Solaris 10



Revocation of A

Capability-Based Systems (Optional)

An Example: Hydra

- Hydra is a capability-based system that includes both system-defined ***rights*** and user-defined rights. The interpretation of user-defined rights is up to the specific user programs, but the OS provides support for protecting access to those rights, whatever they may be
- Operations on objects are defined procedurally, and those procedures are themselves protected objects, accessed indirectly through capabilities.
- The names of user-defined procedures must be identified to the protection system if it is to deal with user-defined rights.
- When an object is created, the names of operations defined on that object become ***auxiliary rights,*** described in a capability for an ***instance*** of the type. For a process to act on an object, the capabilities it holds for that object must contain the name of the operation being invoked. This allows access to be controlled on an instance-by-instance and process-by-process basis.
- Hydra also allows ***rights amplification,*** in which a process is deemed to be ***trustworthy,*** and thereby allowed to act on any object corresponding to its parameters.
- Programmers can make direct use of the Hydra protection system, using suitable libraries which are documented in appropriate reference manuals.

An Example: Cambridge CAP System

- The CAP system has two kinds of capabilities:
    - ***Data capability,*** used to provide read, write, and execute access to objects. These capabilities are interpreted by microcode in the CAP machine.
    - ***Software capability,*** is protected but not interpreted by the CAP microcode.
        - Software capabilities are interpreted by protected (privileged) procedures, possibly written by application programmers.
        - When a process executes a protected procedure, it temporarily gains the ability to read or write the contents of a software capability.
        - This leaves the interpretation of the software capabilities up to the individual subsystems, and limits the potential damage that could be caused by a faulty privileged procedure.
        - Note, however, that protected procedures only get access to software capabilities for the subsystem of which they are a part. Checks are made when passing software capabilities to protected procedures that they are of the correct type.
        - Unfortunately the CAP system does not provide libraries, making it harder for an individual programmer to use than the Hydra system.

Language-Based Protection (Optional)

- As systems have developed, protection systems have become more powerful, and also more specific and specialized.
- To refine protection even further requires putting protection capabilities into the hands of individual programmers, so that protection policies can be implemented on the application level, i.e. to protect resources in ways that are known to the specific applications but not to the more general operating system.

Compiler-Based Enforcement

- In a compiler-based approach to protection enforcement, programmers directly specify the protection needed for different resources at the time the resources are declared.
- This approach has several advantages:
    1. Protection needs are simply declared, as opposed to a complex series of procedure calls.
    2. Protection requirements can be stated independently of the support provided by a particular OS.
    3. The means of enforcement need not be provided directly by the developer.
    4. Declarative notation is natural, because access privileges are closely related to the concept of data types.
- Regardless of the means of implementation, compiler-based protection relies upon the underlying protection mechanisms provided by the underlying OS, such as the Cambridge CAP or Hydra systems.
- Even if the underlying OS does not provide advanced protection mechanisms, the compiler can still offer some protection, such as treating memory accesses differently in code versus data segments. (E.g. code segments cant be modified, data segments can't be executed.)
- There are several areas in which compiler-based protection can be compared to kernel-enforced protection:
    - **Security.** Security provided by the kernel offers better protection than that provided by a compiler. The security of the compiler-based enforcement is dependent upon the integrity of the compiler itself, as well as requiring that files not be modified after they are compiled. The kernel is in a better position to protect itself from modification, as well as protecting access to specific files. Where hardware support of individual memory accesses is available, the protection is stronger still.
    - **Flexibility.** A kernel-based protection system is not as flexible to provide the specific protection needed by an individual programmer, though it may provide support which the programmer may make use of. Compilers are more easily changed and updated when necessary to change the protection services offered or their implementation.
    - **Efficiency.** The most efficient protection mechanism is one supported by hardware and microcode. Insofar as software based protection is concerned, compiler-based systems have the advantage that many checks can be made off-line, at compile time, rather that during execution.

- The concept of incorporating protection mechanisms into programming languages is in its infancy, and still remains to be fully developed. However the general goal is to provide mechanisms for three functions:
    0. Distributing capabilities safely and efficiently among customer processes. In particular a user process should only be able to access resources for which it was issued capabilities.
    1. Specifying the ***type*** of operations a process may execute on a resource, such as reading or writing.
    2. Specifying the ***order*** in which operations are performed on the resource, such as opening before reading.

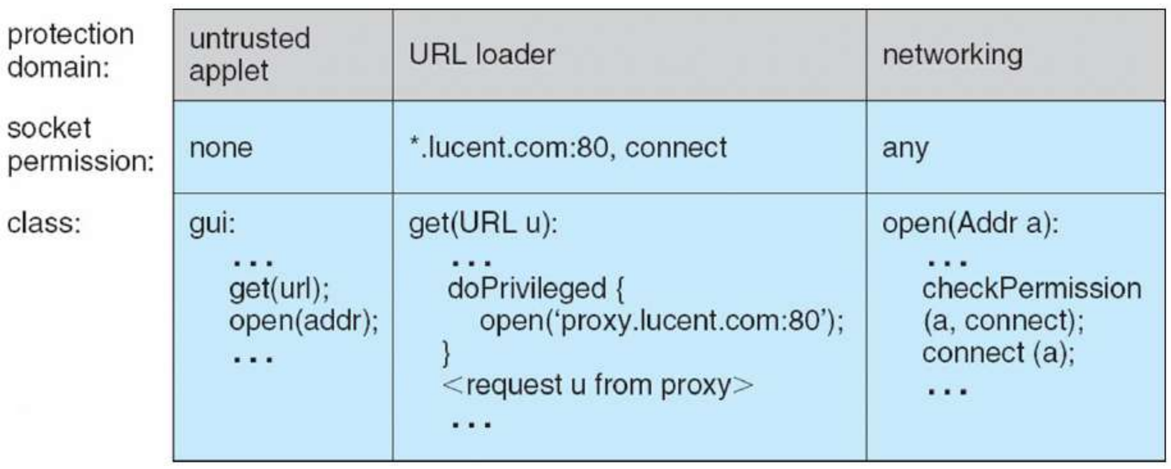
Language-Based Protection

Specification of protection in a programming language allows the high-level description of policies for the allocation and use of resources Language implementation can provide software for protection enforcement when automatic hardware-supported checking is unavailable Interpret protection specifications to generate calls on whatever protection system is provided by the hardware and the operating system

Protection in Java 2

Protection is handled by the Java Virtual Machine (JVM) A class is assigned a protection domain when it is loaded by the JVM The protection domain indicates what operations the class can (and cannot) perform If a library method is invoked that performs a privileged operation, the stack is inspected to ensure the operation can be performed by the library

Stack Inspection



protection domain:	untrusted applet	URL loader	networking
socket permission:	none	*.lucent.com:80, connect	any

class:	gui:
	...
	get(url);
	open(addr);
	...
		get(URL u):
		...
		doPrivileged {
		open('proxy.lucent.com:80');
		}
		<request u from proxy>
		...
			open(Addr a):
			...
			checkPermission
			(a, connect);
			connect (a);
			...

<u>Unit-V Part-II System Security</u>

The Security Problem

- Protection dealt with protecting files and other resources from accidental misuse by cooperating users sharing a system, generally using the computer for normal purposes.
- Security deals with protecting systems from deliberate attacks, either internal or external, from individuals intentionally attempting to steal information, damage information, or otherwise deliberately wreak havoc in some manner.
- Some of the most common types of ***violations*** include:
  - ✓ ***Breach of Confidentiality -*** Theft of private or confidential information, such as credit-card numbers, trade secrets, patents, secret formulas, manufacturing procedures, medical information, financial information, etc.
  - ✓ ***Breach of Integrity -*** Unauthorized ***modification*** of data, which may have serious indirect consequences. For example a popular game or other program's source code could be modified to open up security holes on users systems before being released to the public.
  - ✓ ***Breach of Availability -*** Unauthorized ***destruction*** of data, often just for the "fun" of causing havoc and for bragging rites. Vandalism of web sites is a common form of this violation.
  - ✓ ***Theft of Service -*** Unauthorized use of resources, such as theft of CPU cycles, installation of daemons running an unauthorized file server, or tapping into the target's telephone or networking services.
  - ✓ ***Denial of Service, DOS -*** Preventing legitimate users from using the system, often by overloading and overwhelming the system with an excess of requests for service.
- One common attack is ***masquerading,*** in which the attacker pretends to be a trusted third party. A variation of this is the ***man-in-the-middle,*** in which the attacker masquerades as both ends of the conversation to two targets.
- A ***replay attack*** involves repeating a valid transmission. Sometimes this can be the entire attack, (such as repeating a request for a money transfer), or other times the content of the original message is replaced with malicious content.

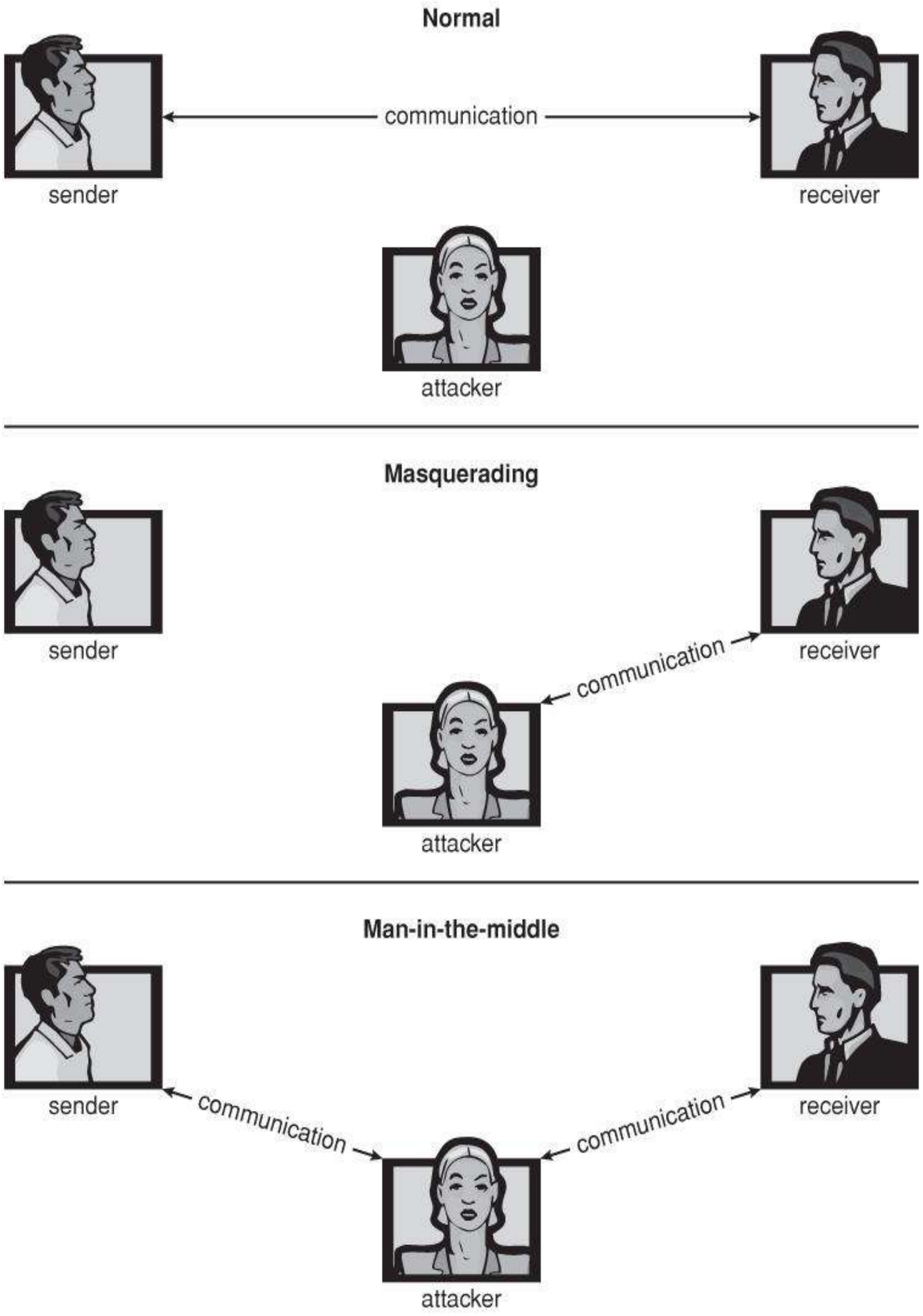


Figure - Standard security attacks.

- There are four levels at which a system must be protected:
    1. **Physical -** The easiest way to steal data is to pocket the backup tapes. Also, access to the root console will often give the user special privileges, such as rebooting the system as root from removable media. Even general access to terminals in a computer room offers some opportunities for an attacker, although today's modern high-speed networking environment provides more and more opportunities for remote attacks.
    2. **Human -** There is some concern that the humans who are allowed access to a system be trustworthy, and that they cannot be coerced into breaching security. However more and more attacks today are made via ***social engineering,*** which basically means fooling trustworthy people into accidentally breaching security.
        - **Phishing** involves sending an innocent-looking e-mail or web site designed to fool people into revealing confidential information. E.g. spam e-mails pretending to be from e-Bay, PayPal, or any of a number of banks or credit-card companies.
        - **Dumpster Diving** involves searching the trash or other locations for passwords that are written down. (Note: Passwords that are too hard to remember, or which must be changed frequently are more likely to be written down somewhere close to the user's station.)
        - **Password Cracking** involves divining user's passwords, either by watching them type in their passwords, knowing something about them like their pet's names, or simply trying all words in common dictionaries. (Note: "Good" passwords should involve a minimum number of characters, include non-alphabetical characters, and not appear in any dictionary (in any language), and should be changed frequently. Note also that it is proper etiquette to look away from the keyboard while someone else is entering their password.)
    3. **Operating System -** The OS must protect itself from security breaches, such as runaway processes (denial of service), memory-access violations, stack overflow violations, the launching of programs with excessive privileges, and many others.
    4. **Network -** As network communications become ever more important and pervasive in modern computing environments, it becomes ever more important to protect this area of the system. (Both protecting the network itself from attack, and protecting the local system from attacks coming in through the network.) This is a growing area of concern as wireless communications and portable devices become more and more prevalent.

Program Threats

- There are many common threats to modern systems. Only a few are discussed here.

Trojan Horse

- A ***Trojan horse*** is a program that secretly performs some maliciousness in addition to its visible actions.

- Some Trojan horses are deliberately written as such, and others are the result of legitimate programs that have become infected with ***viruses,***
- One dangerous opening for Trojan horses is long search paths, and in particular paths which include the current directory (".") as part of the path. If a dangerous program having the same name as a legitimate program (or a common mis-spelling, such as "sl" instead of "ls") is placed anywhere on the path, then an unsuspecting user may be fooled into running the wrong program by mistake.
- Another classic Trojan Horse is a login emulator, which records a users account name and password, issues a "password incorrect" message, and then logs off the system. The user then tries again (with a proper login prompt), logs in successfully, and doesn't realize that their information has been stolen.
- Two solutions to Trojan Horses are to have the system print usage statistics on logouts, and to require the typing of non-trappable key sequences such as Control-Alt-Delete in order to log in. (This is why modern Windows systems require the Control-Alt-Delete sequence to commence logging in, which cannot be emulated or caught by ordinary programs. I.e. that key sequence always transfers control over to the operating system.)
- ***Spyware*** is a version of a Trojan Horse that is often included in "free" software downloaded off the Internet. Spyware programs generate pop-up browser windows, and may also accumulate information about the user and deliver it to some central site. (This is an example of ***covert channels,*** in which surreptitious communications occur.) Another common task of spyware is to send out spam e-mail messages, which then purportedly come from the infected user.

Trap Door

- A ***Trap Door*** is when a designer or a programmer (or hacker) deliberately inserts a security hole that they can use later to access the system.
- Because of the possibility of trap doors, once a system has been in an untrustworthy state, that system can never be trusted again. Even the backup tapes may contain a copy of some cleverly hidden back door.
- A clever trap door could be inserted into a compiler, so that any programs compiled with that compiler would contain a security hole. This is especially dangerous, because inspection of the code being compiled would not reveal any problems.

Logic Bomb

- A ***Logic Bomb*** is code that is not designed to cause havoc all the time, but only when a certain set of circumstances occurs, such as when a particular date or time is reached or some other noticeable event.

Stack and Buffer Overflow

- This is a classic method of attack, which exploits bugs in system code that allows buffers to overflow. Consider what happens in the following code, for example, if argv[1] exceeds 256 characters:

- The strcpy command will overflow the buffer, overwriting adjacent areas of memory.
- (The problem could be avoided using str***n***cpy, with a limit of 255 characters copied plus room for the null byte.)

```
#include
#define BUFFER_SIZE 256

int main( int argc, char * argv[ ] )
{
   char buffer[ BUFFER_SIZE ];

   if( argc < 2 )
      return -1;
   else {
      strcpy( buffer, argv[ 1 ] );
      return 0;
   }
}
```

Figure - C program with buffer-overflow condition.

- So how does overflowing the buffer cause a security breach? Well the first step is to understand the structure of the stack in memory:
  - The "bottom" of the stack is actually at a high memory address, and the stack grows towards lower addresses.
  - However the address of an array is the lowest address of the array, and higher array elements extend to higher addresses. (I.e. an array "grows" towards the bottom of the stack.
  - In particular, writing past the top of an array, as occurs when a buffer overflows with too much input data, can eventually overwrite the return address, effectively changing where the program jumps to when it returns.

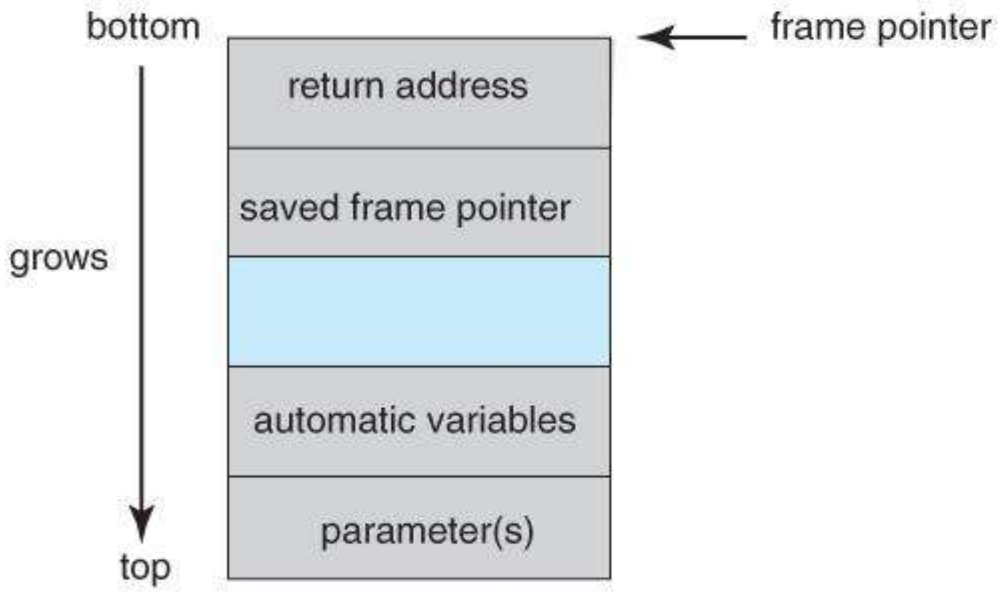


Figure - The layout for a typical stack frame.

- Now that we know how to change where the program returns to by overflowing the buffer, the second step is to insert some nefarious code, and then get the program to jump to our inserted code.
- Our only opportunity to enter code is via the input into the buffer, which means there isn't room for very much. One of the simplest and most obvious approaches is to insert the code for "exec (/bin/sh)". To do this requires compiling a program that contains this instruction, and then using an assembler or debugging tool to extract the minimum extent that includes the necessary instructions.
- The bad code is then padded with as many extra bytes as are needed to overflow the buffer to the correct extent, and the address of the buffer inserted into the return address location. (Note, however, that neither the bad code nor the padding can contain null bytes, which would terminate the strcpy.)
- The resulting block of information is provided as "input", copied into the buffer by the original program, and then the return statement causes control to jump to the location of the buffer and start executing the code to launch a shell.

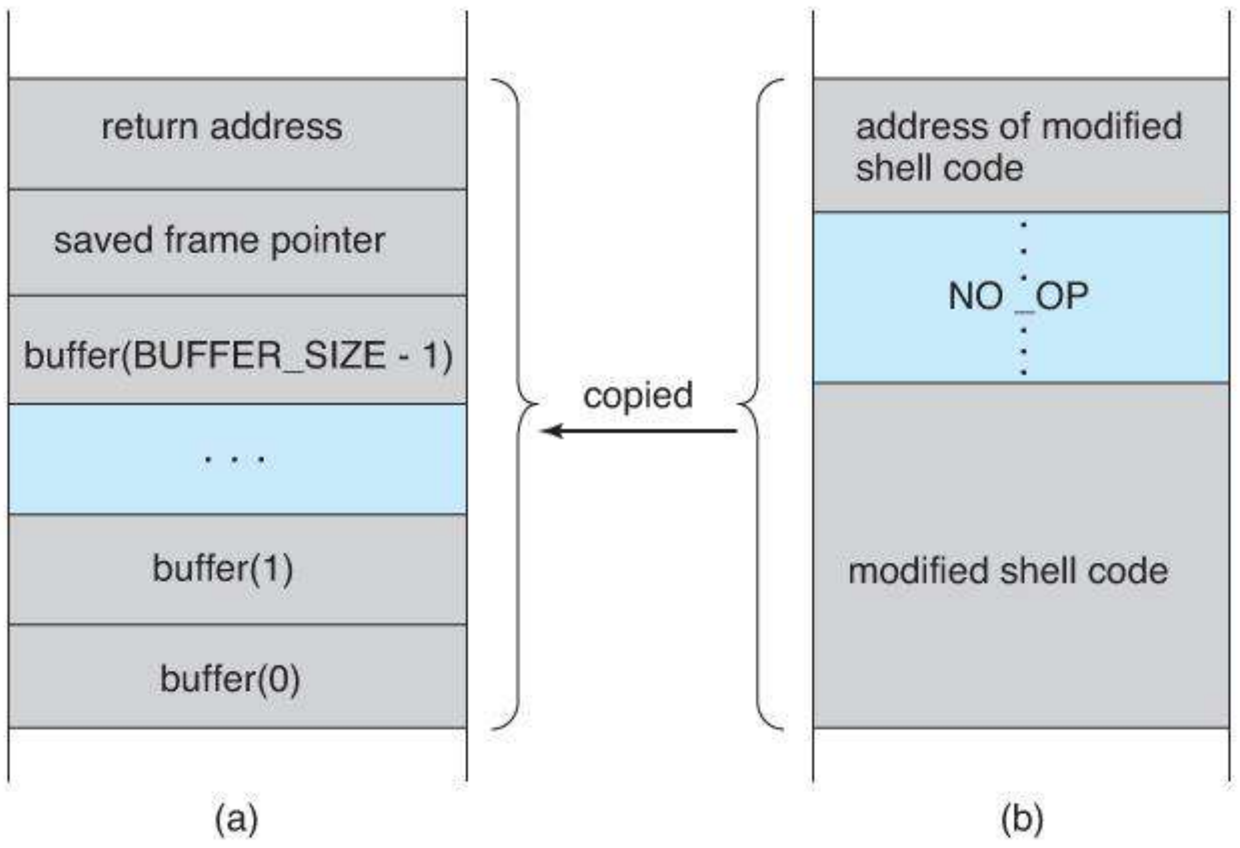


Figure : Hypothetical stack frame for (a) before and (b) after.

- Fortunately modern hardware now includes a bit in the page tables to mark certain pages as non-executable. In this case the buffer-overflow attack would work up to a point, but as soon as it "returns" to an address in the data space and tries executing statements there, an exception would be thrown crashing the program.

Viruses

- A virus is a fragment of code embedded in an otherwise legitimate program, designed to replicate itself (by infecting other programs), and (eventually) wreaking havoc.
- Viruses are more likely to infect PCs than UNIX or other multi-user systems, because programs in the latter systems have limited authority to modify other programs or to access critical system structures (such as the boot block.)
- Viruses are delivered to systems in a ***virus dropper,*** usually some form of a Trojan Horse, and usually via e-mail or unsafe downloads.
- Viruses take many forms (see below.) Figure shows typical operation of a boot sector virus:

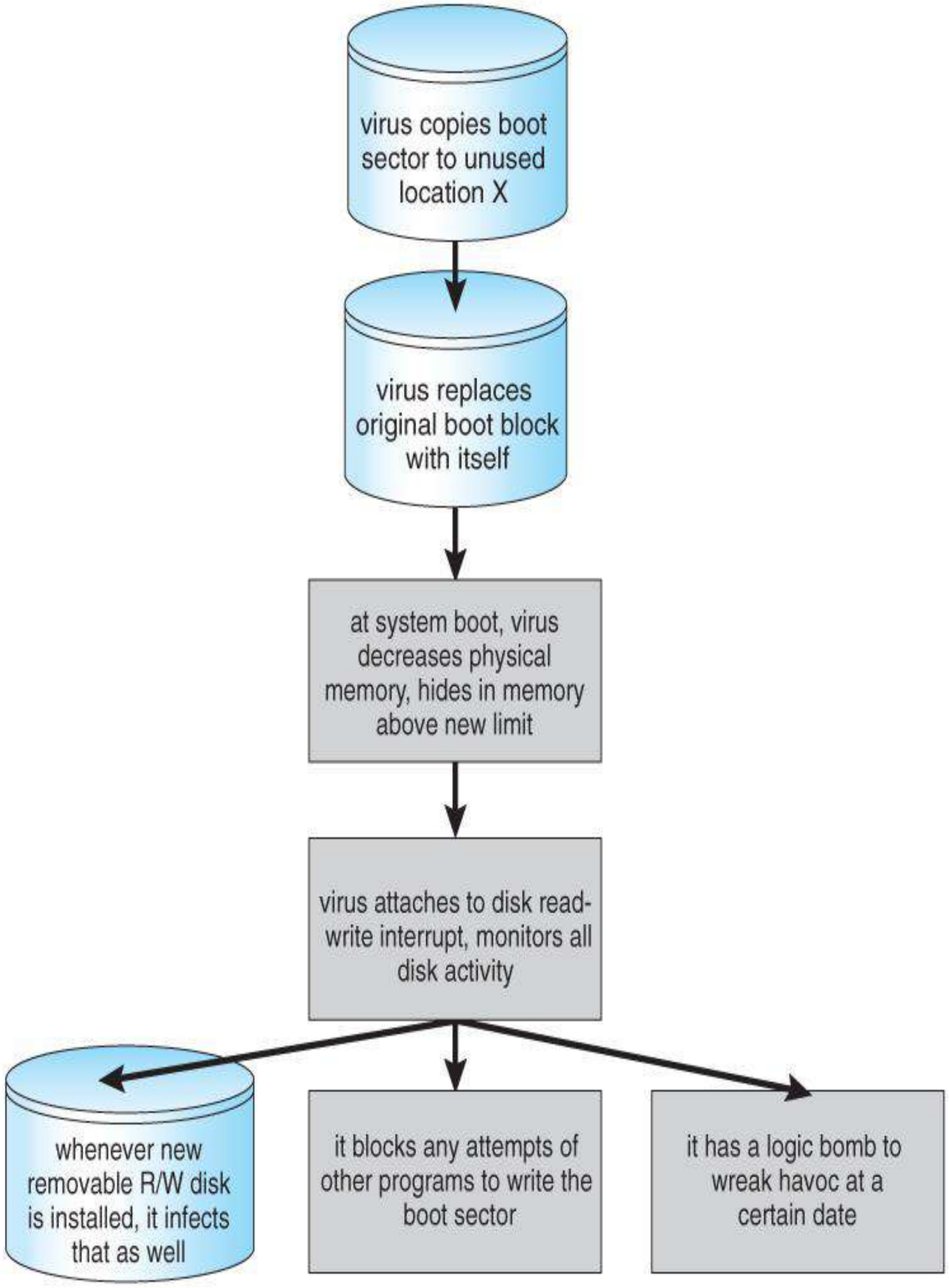

virus copies boot sector to unused location X
virus replaces original boot block with itself
at system boot, virus decreases physical memory, hides in memory above new limit
virus attaches to disk read-write interrupt, monitors all disk activity
whenever new removable R/W disk is installed, it infects that as well
it blocks any attempts of other programs to write the boot sector
it has a logic bomb to wreak havoc at a certain date

Figure - A boot-sector computer virus.

- Some of the forms of viruses include:
  - ✓ **File -** A file virus attaches itself to an executable file, causing it to run the virus code first and then jump to the start of the original program. These viruses are termed ***parasitic,*** because they do not leave any new files on the system, and the original program is still fully functional.
  - ✓ **Boot -** A boot virus occupies the boot sector, and runs before the OS is loaded. These are also known as ***memory viruses***, because in operation they reside in memory, and do not appear in the file system.
  - ✓ **Macro -** These viruses exist as a macro (script) that is run automatically by certain macro-capable programs such as MS Word or Excel. These viruses can exist in word processing documents or spreadsheet files.
  - ✓ **Source code** viruses look for source code and infect it in order to spread.
  - ✓ **Polymorphic** viruses change every time they spread - Not their underlying functionality, but just their ***signature,*** by which virus checkers recognize them.
  - ✓ **Encrypted** viruses travel in encrypted form to escape detection. In practice they are self-decrypting, which then allows them to infect other files.
  - ✓ **Stealth** viruses try to avoid detection by modifying parts of the system that could be used to detect it. For example the read ( ) system call could be modified so that if an infected file is read the infected part gets skipped and the reader would see the original unadulterated file.
  - ✓ **Tunneling** viruses attempt to avoid detection by inserting themselves into the interrupt handler chain, or into device drivers.
  - ✓ **Multipartite** viruses attack multiple parts of the system, such as files, boot sector, and memory.
  - ✓ **Armored** viruses are coded to make them hard for anti-virus researchers to decode and understand. In addition many files associated with viruses are hidden, protected, or given innocuous looking names such as "...".
- In 2004 a virus exploited three bugs in Microsoft products to infect hundreds of Windows servers (including many trusted sites) running Microsoft Internet Information Server, which in turn infected any Microsoft Internet Explorer web browser that visited any of the infected server sites. One of the back-door programs it installed was a ***keystroke logger,*** which records user's keystrokes, including passwords and other sensitive information.
- There is some debate in the computing community as to whether a ***monoculture,*** in which nearly all systems run the same hardware, operating system, and applications, increases the threat of viruses and the potential for harm caused by them.

System and Network Threats

- Most of the threats described above are termed ***program threats***, because they attack specific programs or are carried and distributed in programs. The threats in this section attack the operating system or the network itself, or leverage those systems to launch their attacks.

Worms

- A ***worm*** is processes that uses the fork / spawn process to make copies of itself in order to wreak havoc on a system. Worms consume system resources, often blocking out other, legitimate processes. Worms that propagate over networks can be especially problematic, as they can tie up vast amounts of network resources and bring down large-scale systems.
- One of the most well-known worms was launched by Robert Morris, a graduate student at Cornell, in November 1988. Targeting Sun and VAX computers running BSD UNIX version 4, the worm spanned the Internet in a matter of a few hours, and consumed enough resources to bring down many systems.
- This worm consisted of two parts:
    1. A small program called a ***grappling hook,*** which was deposited on the target system through one of three vulnerabilities, and
    2. The main worm program, which was transferred onto the target system and launched by the grappling hook program.

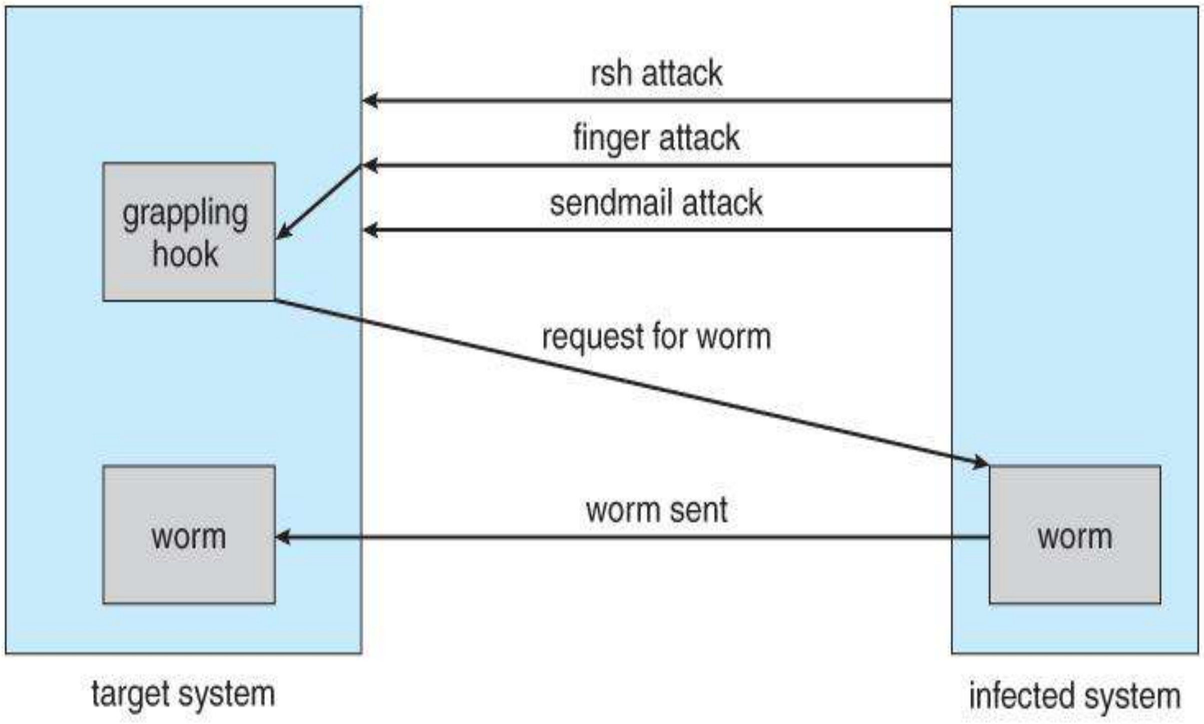


Figure - The Morris Internet worm.

- The three vulnerabilities exploited by the Morris Internet worm were as follows:
    1. **rsh (remote shell)** is a utility that was in common use at that time for accessing remote systems without having to provide a password. If a user had an account on two different computers (with the same account name on both systems), then the system could be configured to allow that user to remotely connect from one system to the other without having to provide a password. Many systems were

configured so that ***any*** user (except root) on system A could access the same account on system B without providing a password.

2. **finger** is a utility that allows one to remotely query a user database, to find the true name and other information for a given account name on a given system. For example "finger joeUser@somemachine.edu" would access the finger daemon at somemachine.edu and return information regarding joeUser. Unfortunately the finger daemon (which ran with system privileges) had the buffer overflow problem, so by sending a special 536-character user name the worm was able to fork a shell on the remote system running with root privileges.
3. **sendmail** is a routine for sending and forwarding mail that also included a debugging option for verifying and testing the system. The debug feature was convenient for administrators, and was often left turned on. The Morris worm exploited the debugger to mail and execute a copy of the grappling hook program on the remote system.

- Once in place, the worm undertook systematic attacks to discover user passwords:
    1. First it would check for accounts for which the account name and the password were the same, such as "guest", "guest".
    2. Then it would try an internal dictionary of 432 favorite password choices. (I'm sure "password", "pass", and blank passwords were all on the list.)
    3. Finally it would try every word in the standard UNIX on-line dictionary to try and break into user accounts.
- Once it had gotten access to one or more user accounts, then it would attempt to use those accounts to rsh to other systems, and continue the process.
- With each new access the worm would check for already running copies of itself, and 6 out of 7 times if it found one it would stop. (The seventh was to prevent the worm from being stopped by fake copies.)
- Fortunately the same rapid network connectivity that allowed the worm to propagate so quickly also quickly led to its demise - Within 24 hours remedies for stopping the worm propagated through the Internet from administrator to administrator, and the worm was quickly shut down.
- There is some debate about whether Mr. Morris's actions were a harmless prank or research project that got out of hand or a deliberate and malicious attack on the Internet. However the court system convicted him, and penalized him heavy fines and court costs.
- There have since been many other worm attacks, including the W32.Sobig.F@mm attack which infected hundreds of thousands of computers and an estimated 1 in 17 e-mails in August 2003. This worm made detection difficult by varying the subject line of the infection-carrying mail message, including "Thank You!", "Your details", and "Re: Approved".

Port Scanning

- ***Port Scanning*** is technically not an attack, but rather a search for vulnerabilities to attack. The basic idea is to systematically attempt to connect to every known (or common or possible) network port on some remote machine, and to attempt to make contact. Once it is determined that a particular computer is listening to a particular port, then the next step is to determine what daemon is listening, and whether or not it is a version containing a known security flaw that can be exploited.

- Because port scanning is easily detected and traced, it is usually launched from ***zombie systems,*** i.e. previously hacked systems that are being used without the knowledge or permission of their rightful owner. For this reason it is important to protect "innocuous" systems and accounts as well as those that contain sensitive information or special privileges.
- There are also port scanners available that administrators can use to check their own systems, which report any weaknesses found but which do not exploit the weaknesses or cause any problems. Two such systems are ***nmap*** (http://www.insecure.org/nmap) and ***nessus*** (http://www.nessus.org). The former identifies what OS is found, what firewalls are in place, and what services are listening to what ports. The latter also contains a database of known security holes, and identifies any that it finds.

Denial of Service

- ***Denial of Service (DOS)*** attacks do not attempt to actually access or damage systems, but merely to clog them up so badly that they cannot be used for any useful work. Tight loops that repeatedly request system services are an obvious form of this attack.
- DOS attacks can also involve social engineering, such as the Internet chain letters that say "send this immediately to 10 of your friends, and then go to a certain URL", which clogs up not only the Internet mail system but also the web server to which everyone is directed. (Note: Sending a "reply all" to such a message notifying everyone that it was just a hoax also clogs up the Internet mail service, just as effectively as if you had forwarded the thing.)
- Security systems that lock accounts after a certain number of failed login attempts are subject to DOS attacks which repeatedly attempt logins to all accounts with invalid passwords strictly in order to lock up all accounts.
- Sometimes DOS is not the result of deliberate maliciousness.

Cryptography as a Security Tool

- Within a given computer the transmittal of messages is safe, reliable and secure, because the OS knows exactly where each one is coming from and where it is going.
- On a network, however, things aren't so straightforward - A rogue computer (or e-mail sender) may spoof their identity, and outgoing packets are delivered to a lot of other computers besides their (intended) final destination, which brings up two big questions of security:
    - **Trust -** How can the system be sure that the messages received are really from the source that they say they are, and can that source be trusted?
    - **Confidentiality -** How can one ensure that the messages one is sending are received only by the intended recipient?
- Cryptography can help with both of these problems, through a system of **secrets** and **keys.** In the former case, the key is held by the sender, so that the recipient knows that only the authentic author could have sent the message; In the latter, the key is

held by the recipient, so that only the intended recipient can receive the message accurately.

- Keys are designed so that they cannot be divined from any public information, and must be guarded carefully. (***Asymmetric encryption*** involve both a public and a private key.)

Encryption

- The basic idea of encryption is to encode a message so that only the desired recipient can decode and read it. Encryption has been around since before the days of Caesar, and is an entire field of study in itself. Only some of the more significant computer encryption schemes will be covered here.
- The basic process of encryption is shown in Figure and will form the basis of most of our discussion on encryption. The steps in the procedure and some of the key terminology are as follows:
    1. The **sender** first creates a **message, m** in plaintext.
    2. The message is then entered into an **encryption algorithm, E,** along with the **encryption key, Ke.**
    3. The encryption algorithm generates the **ciphertext, c, = E(Ke)(m).** For any key k, E(k) is an algorithm for generating ciphertext from a message, and both E and E(k) should be efficiently computable functions.
    4. The ciphertext can then be sent over an unsecure network, where it may be received by **attackers.**
    5. The **recipient** enters the ciphertext into a **decryption algorithm, D,** along with the **decryption key, Kd.**
    6. The decryption algorithm re-generates the plaintext message, m, = D(Kd)(c). For any key k, D(k) is an algorithm for generating a clear text message from a ciphertext, and both D and D(k) should be efficiently computable functions.
    7. The algorithms described here must have this important property: Given a ciphertext c, a computer can only compute a message m such that c = E(k)(m) if it possesses D(k). (In other words, the messages can't be decoded unless you have the decryption algorithm and the decryption key.)

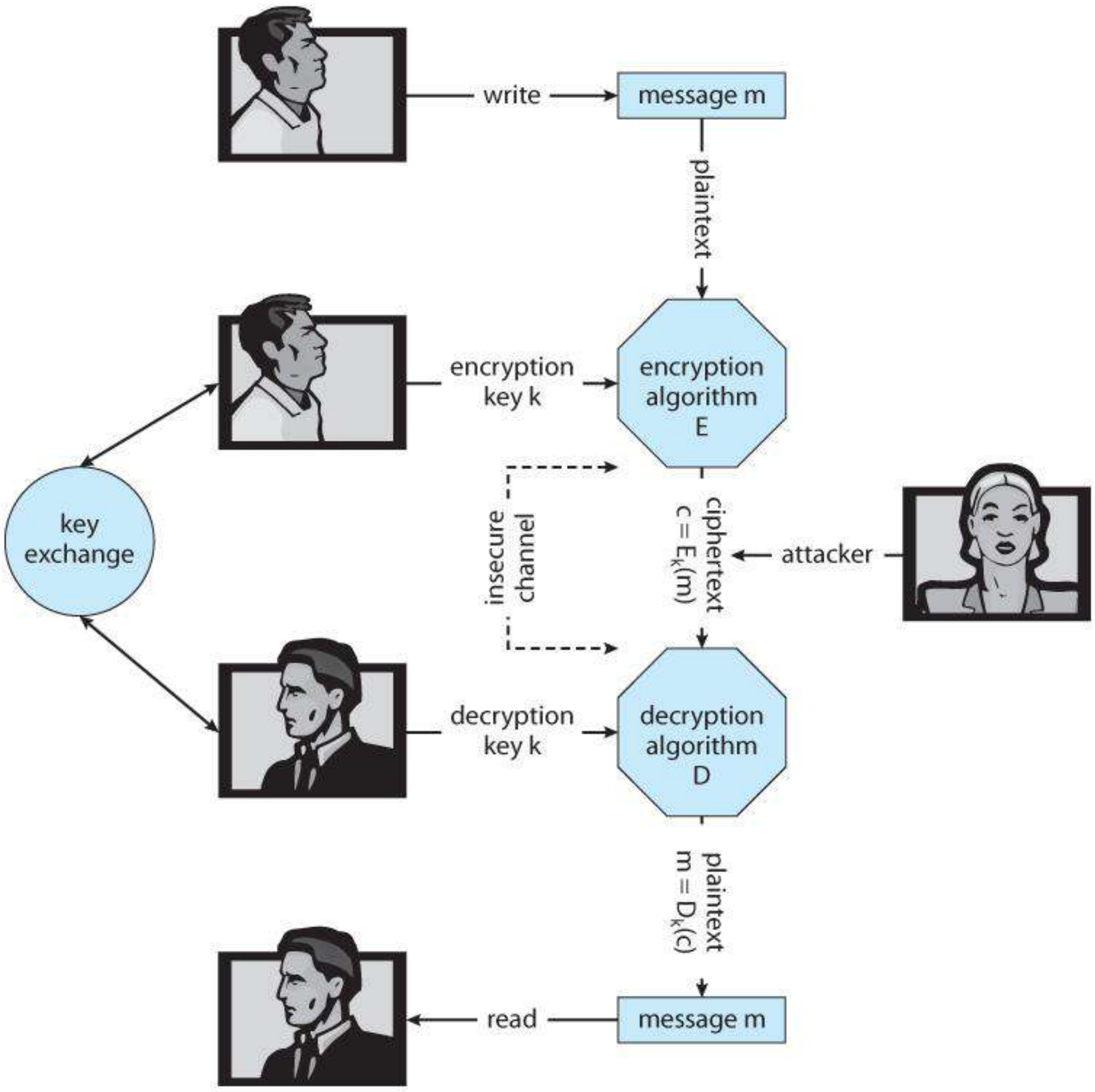


Figure - A secure communication over an insecure medium.

Symmetric Encryption

- With ***symmetric encryption*** the same key is used for both encryption and decryption, and must be safely guarded. There are a number of well-known symmetric encryption algorithms that have been used for computer security:
  - The ***Data-Encryption Standard, DES,*** developed by the National Institute of Standards, NIST, has been a standard civilian encryption standard for over 20 years. Messages are broken down into 64-bit chunks, each of which are encrypted

using a 56-bit key through a series of substitutions and transformations. Some of the transformations are hidden (black boxes), and are classified by the U.S. government.

- DES is known as a ***block cipher,*** because it works on blocks of data at a time. Unfortunately this is a vulnerability if the same key is used for an extended amount of data. Therefore an enhancement is to not only encrypt each block, but also to XOR it with the previous block, in a technique known as ***cipher-block chaining.***
- As modern computers become faster and faster, the security of DES has decreased, to where it is now considered insecure because its keys can be exhaustively searched within a reasonable amount of computer time. An enhancement called ***triple DES*** encrypts the data three times using three separate keys (actually two encryptions and one decryption) for an effective key length of 168 bits. Triple DES is in widespread use today.
- The ***Advanced Encryption Standard, AES,*** developed by NIST in 2001 to replace DES uses key lengths of 128, 192, or 256 bits, and encrypts in blocks of 128 bits using 10 to 14 rounds of transformations on a matrix formed from the block.
- The ***twofish algorithm,*** uses variable key lengths up to 256 bits and works on 128 bit blocks.
- ***RC5*** can vary in key length, block size, and the number of transformations, and runs on a wide variety of CPUs using only basic computations.
- ***RC4*** is a ***stream cipher,*** meaning it acts on a stream of data rather than blocks. The key is used to seed a pseudo-random number generator, which generates a ***keystream***of keys. RC4 is used in ***WEP,*** but has been found to be breakable in a reasonable amount of computer time.

Asymmetric Encryption

- With ***asymmetric encryption,*** the decryption key, Kd, is not the same as the encryption key, Ke, and more importantly cannot be derived from it, which means the encryption key can be made publicly available, and only the decryption key needs to be kept secret. (or vice-versa, depending on the application.)
- One of the most widely used asymmetric encryption algorithms is ***RSA,*** named after its developers - Rivest, Shamir, and Adleman.
- RSA is based on two large prime numbers, ***p*** and ***q,*** (on the order of 512 bits each), and their product ***N.***
  - Ke and Kd must satisfy the relationship:
    (Ke * Kd) % [(p - 1) * (q - 1)] = = 1
  - The encryption algorithm is:
    c = E(Ke)(m) = m^Ke % N
  - The decryption algorithm is:
    m = D(Kd)(c) = c^Kd % N

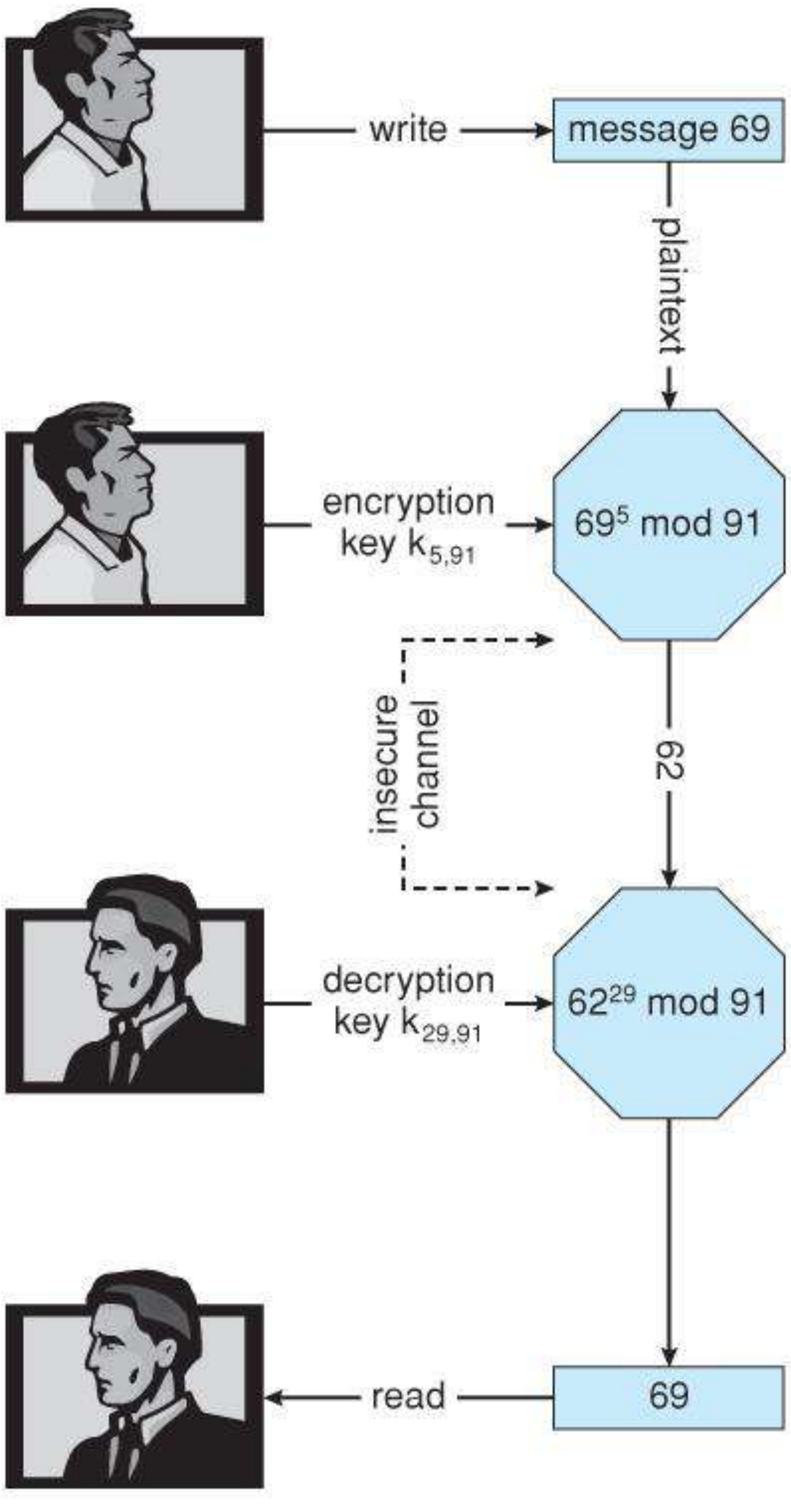


Figure - Encryption and decryption using RSA asymmetric cryptography

- Note that asymmetric encryption is much more computationally expensive than symmetric encryption, and as such it is not normally used for large transmissions.

Asymmetric encryption is suitable for small messages, authentication, and key distribution, as covered in the following sections.

Authentication

- Authentication involves verifying the identity of the entity who transmitted a message.
- For example, if D(Kd)(c) produces a valid message, then we know the sender was in possession of E(Ke).
- This form of authentication can also be used to verify that a message has not been modified
- Authentication revolves around two functions, used for ***signatures*** (or ***signing***), and ***verification:***
    - A signing function, ***S(Ks)*** that produces an ***authenticator, A,*** from any given message m.
    - A Verification function, ***V(Kv,m,A)*** that produces a value of "true" if A was created from m, and "false" otherwise.
    - Obviously S and V must both be computationally efficient.
    - More importantly, it must not be possible to generate a valid authenticator, A, without having possession of S(Ks).
    - Furthermore, it must not be possible to divine S(Ks) from the combination of (m and A), since both are sent visibly across networks.
- Understanding authenticators begins with an understanding of hash functions, which is the first step:
    - ***Hash functions, H(m)*** generate a small fixed-size block of data known as a ***message digest,*** or ***hash value*** from any given input data.
    - For authentication purposes, the hash function must be ***collision resistant on m.*** That is it should not be reasonably possible to find an alternate message m' such that H(m') = H(m).
    - Popular hash functions are **MD5,** which generates a 128-bit message digest, and **SHA-1,** which generates a 160-bit digest.
- Message digests are useful for detecting (accidentally) changed messages, but are not useful as authenticators, because if the hash function is known, then someone could easily change the message and then generate a new hash value for the modified message. Therefore authenticators take things one step further by encrypting the message digest.
- A ***message-authentication code, MAC,*** uses symmetric encryption and decryption of the message digest, which means that anyone capable of verifying an incoming message could also generate a new message.
- An asymmetric approach is the ***digital-signature algorithm,*** which produces authenticators called ***digital signatures.*** In this case Ks and Kv are separate, Kv is the public key, and it is not practical to determine S(Ks) from public information. In practice the sender of a message signs it (produces a digital signature using S(Ks)), and the receiver uses V(Kv) to verify that it did indeed come from a trusted source, and that it has not been modified.
- There are three good reasons for having separate algorithms for encryption of messages and authentication of messages:

1. Authentication algorithms typically require fewer calculations, making verification a faster operation than encryption.
2. Authenticators are almost always smaller than the messages, improving space efficiency. (?)
3. Sometimes we want authentication only, and not confidentiality, such as when a vendor issues a new software patch.

- Another use of authentication is ***non-repudiation,*** in which a person filling out an electronic form cannot deny that they were the ones who did so.

Key Distribution

- Key distribution with symmetric cryptography is a major problem, because all keys must be kept secret, and they obviously can't be transmitted over unsecure channels. One option is to send them ***out-of-band,*** say via paper or a confidential conversation.
- Another problem with symmetric keys, is that a separate key must be maintained and used for each correspondent with whom one wishes to exchange confidential information.
- Asymmetric encryption solves some of these problems, because the public key can be freely transmitted through any channel, and the private key doesn't need to be transmitted anywhere. Recipients only need to maintain one private key for all incoming messages, though senders must maintain a separate public key for each recipient to which they might wish to send a message. Fortunately the public keys are not confidential, so this ***key-ring*** can be easily stored and managed.
- Unfortunately there are still some security concerns regarding the public keys used in asymmetric encryption. Consider for example the following man-in-the-middle attack involving phony public keys:

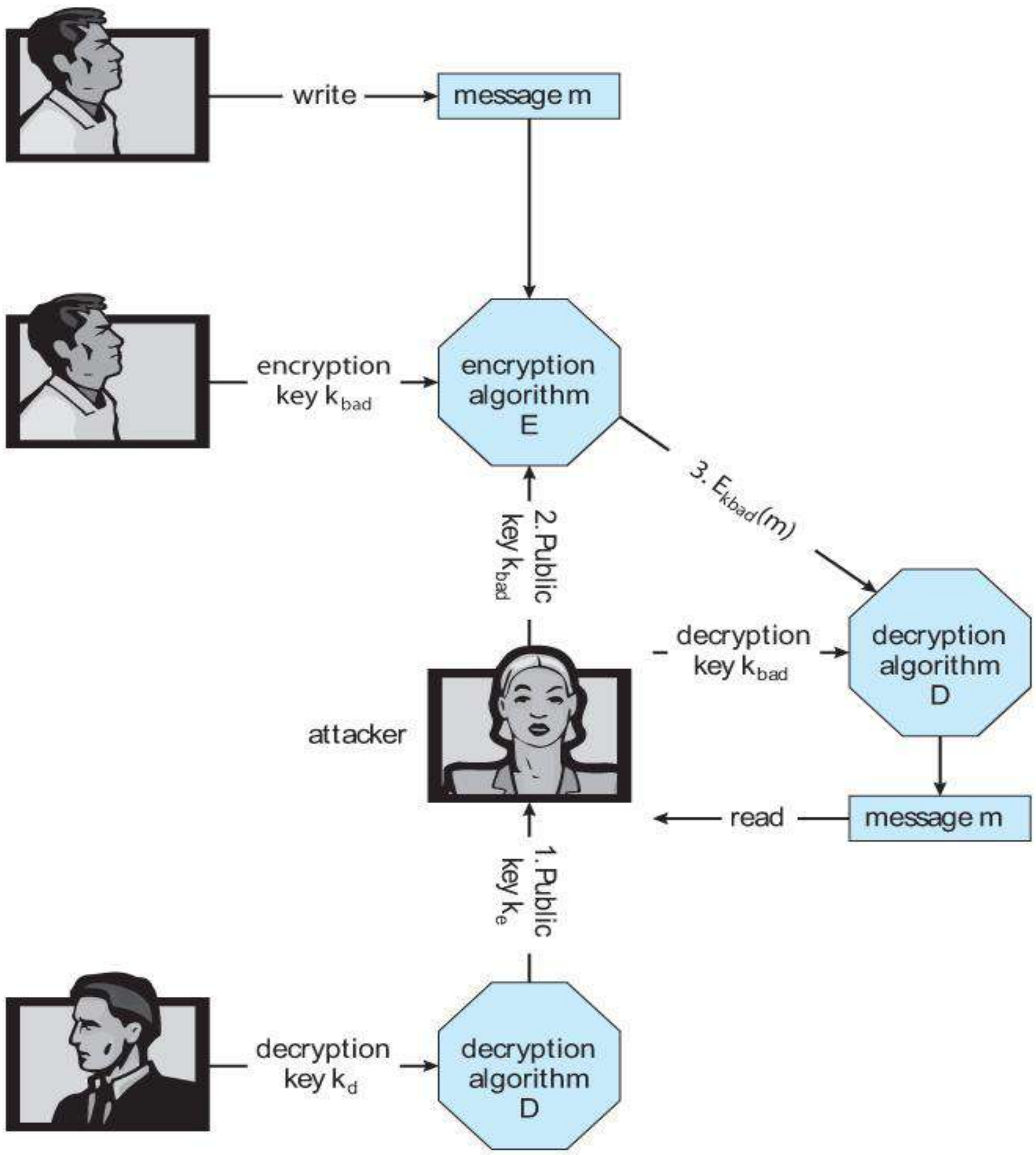


Figure - A man-in-the-middle attack on asymmetric cryptography.

- One solution to the above problem involves ***digital certificates,*** which are public keys that have been digitally signed by a trusted third party. But wait a minute - How do we trust that third party, and how do we know ***they*** are really who they say they are? Certain ***certificate authorities*** have their public keys included within web browsers and other certificate consumers before they are distributed. These certificate authorities can then vouch for other trusted entities and so on in a web of trust.

- Network communications are implemented in multiple layers - Physical, Data Link, Network, Transport, and Application being the most common breakdown.
- Encryption and security can be implemented at any layer in the stack, with pros and cons to each choice:
  - Because packets at lower levels contain the contents of higher layers, encryption at lower layers automatically encrypts higher layer information at the same time.
  - However security and authorization may be important to higher levels independent of the underlying transport mechanism or route taken.
- At the network layer the most common standard is **IPSec,** a secure form of the IP layer, which is used to set up **Virtual Private Networks, VPNs.**
- At the transport layer the most common implementation is SSL, described below.

An Example: SSL

- SSL (Secure Sockets Layer) 3.0 was first developed by Netscape, and has now evolved into the industry-standard TLS protocol. It is used by web browsers to communicate securely with web servers, making it perhaps the most widely used security protocol on the Internet today.
- SSL is quite complex with many variations, only a simple case of which is shown here.
- The heart of SSL is ***session keys***, which are used once for symmetric encryption and then discarded, requiring the generation of new keys for each new session. The big challenge is how to safely create such keys while avoiding man-in-the-middle and replay attacks.
- Prior to commencing the transaction, the server obtains a ***certificate*** from a ***certification authority, CA,*** containing:
  - Server attributes such as unique and common names.
  - Identity of the public encryption algorithm, E( ), for the server.
  - The public key, k_e for the server.
  - The validity interval within which the certificate is valid.
  - A digital signature on the above issued by the CA:
    - a = S(K_CA)((attrs, E(k_e), interval)
- In addition, the client will have obtained a public ***verification algorithm,*** V(K_CA), for the certifying authority. Today's modern browsers include these built-in by the browser vendor for a number of trusted certificate authorities.
- The procedure for establishing secure communications is as follows:
  1. The client, c, connects to the server, s, and sends a random 28-byte number, n_c.
  2. The server replies with its own random value, n_s, along with its certificate of authority.
  3. The client uses its verification algorithm to confirm the identity of the sender, and if all checks out, then the client generates a 46 byte random **premaster secret, pms,** and sends an encrypted version of it as cpms = E(k_s)(pms)

4. The server recovers pms as D(k_s)(cpms).
5. Now both the client and the server can compute a shared 48-byte ***master secret, ms,*** = f(pms, n_s, n_c)
6. Next, both client and server generate the following from ms:
    - Symmetric encryption keys k_sc_crypt and k_cs_crypt for encrypting messages from the server to the client and vice-versa respectively.
    - MAC generation keys k_sc_mac and k_cs_mac for generating authenticators on messages from server to client and client to server respectively.
7. To send a message to the server, the client sends:
    - c = E(k_cs_crypt)(m, S(k_cs_mac))(m)))
8. Upon receiving c, the server recovers:
    - (m,a) = D(k_cs_crypt)(c)
    - and accepts it if V(k_sc_mac)(m,a) is true.

- This approach enables both the server and client to verify the authenticity of every incoming message, and to ensure that outgoing messages are only readable by the process that originally participated in the key generation.
- SSL is the basis of many secure protocols,including ***Virtual Private Networks, VPNs,*** in which private data is distributed over the insecure public internet structure in an encrypted fashion that emulates a privately owned network.

User Authentication

- A lot of chapter 14, Protection, dealt with making sure that only certain users were allowed to perform certain tasks, i.e. that a users privileges were dependent on his or her identity. But how does one verify that identity to begin with?

Passwords

- Passwords are the most common form of user authentication. If the user is in possession of the correct password, then they are considered to have identified themselves.
- In theory separate passwords could be implemented for separate activities, such as reading this file, writing that file, etc. In practice most systems use one password to confirm user identity, and then authorization is based upon that identification. This is a result of the classic trade-off between security and convenience.

Password Vulnerabilities

- Passwords can be guessed.
    - Intelligent guessing requires knowing something about the intended target in specific, or about people and commonly used passwords in general.
    - Brute-force guessing involves trying every word in the dictionary, or every valid combination of characters. For this reason good passwords should not be in any dictionary (in any language), should be reasonably lengthy, and should use the

full range of allowable characters by including upper and lower case characters, numbers, and special symbols.

- "**Shoulder surfing**" involves looking over people's shoulders while they are typing in their password.
    - Even if the lurker does not get the entire password, they may get enough clues to narrow it down, especially if they watch on repeated occasions.
    - Common courtesy dictates that you look away from the keyboard while someone is typing their password.
    - Passwords echoed as stars or dots still give clues, because an observer can determine how many characters are in the password.
- "**Packet sniffing**" involves putting a monitor on a network connection and reading data contained in those packets.
    - SSH encrypts all packets, reducing the effectiveness of packet sniffing.
    - However you should still never e-mail a password, particularly not with the word "password" in the same message or worse yet the subject header.
    - Beware of any system that transmits passwords in clear text. ("Thank you for signing up for XYZ. Your new account and password information are shown below".) You probably want to have a spare throw-away password to give these entities, instead of using the same high-security password that you use for banking or other confidential uses.
- Long hard to remember passwords are often written down, particularly if they are used seldomly or must be changed frequently. Hence a security trade-off of passwords that are easily divined versus those that get written down. :-(
- Passwords can be given away to friends or co-workers, destroying the integrity of the entire user-identification system.
- Most systems have configurable parameters controlling password generation and what constitutes acceptable passwords.
    - They may be user chosen or machine generated.
    - They may have minimum and/or maximum length requirements.
    - They may need to be changed with a given frequency. (In extreme cases for every session.)
    - A variable length history can prevent repeating passwords.
    - More or less stringent checks can be made against password dictionaries.

Encrypted Passwords

- Modern systems do not store passwords in clear-text form, and hence there is no mechanism to look up an existing password.
- Rather they are encrypted and stored in that form. When a user enters their password, that too is encrypted, and if the encrypted version match, then user authentication passes.
- The encryption scheme was once considered safe enough that the encrypted versions were stored in the publicly readable file "/etc/passwd".
    - They always encrypted to a 13 character string, so an account could be disabled by putting a string of any other length into the password field.

    - Modern computers can try every possible password combination in a reasonably short time, so now the encrypted passwords are stored in files that are only readable by the super user. Any password-related programs run as setuid root to get access to these files. (/etc/shadow)
    - A random seed is included as part of the password generation process, and stored as part of the encrypted password. This ensures that if two accounts have the same plain-text password that they will not have the same encrypted password. However cutting and pasting encrypted passwords from one account to another will give them the same plain-text passwords.

One-Time Passwords

- One-time passwords resist shoulder surfing and other attacks where an observer is able to capture a password typed in by a user.
    - These are often based on a **challenge** and a **response.** Because the challenge is different each time, the old response will not be valid for future challenges.
        - For example, The user may be in possession of a secret function f(x). The system challenges with some given value for x, and the user responds with f(x), which the system can then verify. Since the challenger gives a different (random) x each time, the answer is constantly changing.
        - A variation uses a map (e.g. a road map) as the key. Today's question might be "On what corner is SEO located?", and tomorrow's question might be "How far is it from Navy Pier to Wrigley Field?" Obviously "Taylor and Morgan" would not be accepted as a valid answer for the second question!
    - Another option is to have some sort of electronic card with a series of constantly changing numbers, based on the current time. The user enters the current number on the card, which will only be valid for a few seconds. A ***two-factor authorization*** also requires a traditional password in addition to the number on the card, so others may not use it if it were ever lost or stolen.
    - A third variation is a ***code book,*** or ***one-time pad.*** In this scheme a long list of passwords is generated and each one is crossed off and cancelled as it is used. Obviously it is important to keep the pad secure.

Biometrics

- Biometrics involve a physical characteristic of the user that is not easily forged or duplicated and not likely to be identical between multiple users.
    - Fingerprint scanners are getting faster, more accurate, and more economical.
    - Palm readers can check thermal properties, finger length, etc.
    - Retinal scanners examine the back of the users' eyes.
    - Voiceprint analyzers distinguish particular voices.
    - Difficulties may arise in the event of colds, injuries, or other physiological changes.

Implementing Security Defenses

Security Policy

- A security policy should be well thought-out, agreed upon, and contained in a living document that everyone adheres to and is updated as needed.
- Examples of contents include how often port scans are run, password requirements, virus detectors, etc.

Vulnerability Assessment

- Periodically examine the system to detect vulnerabilities.
  - Port scanning.
  - Check for bad passwords.
  - Look for suid programs.
  - Unauthorized programs in system directories.
  - Incorrect permission bits set.
  - Program checksums / digital signatures which have changed.
  - Unexpected or hidden network daemons.
  - New entries in startup scripts, shutdown scripts, cron tables, or other system scripts or configuration files.
  - New unauthorized accounts.
- The government considers a system to be only as secure as its most far-reaching component. Any system connected to the Internet is inherently less secure than one that is in a sealed room with no external communications.
- Some administrators advocate "security through obscurity", aiming to keep as much information about their systems hidden as possible, and not announcing any security concerns they come across. Others announce security concerns from the rooftops, under the theory that the hackers are going to find out anyway, and the only one kept in the dark by obscurity are honest administrators who need to get the word.

Intrusion Detection

- Intrusion detection attempts to detect attacks, both successful and unsuccessful attempts. Different techniques vary along several axes:
  - The time that detection occurs, either during the attack or after the fact.
  - The types of information examined to detect the attack(s). Some attacks can only be detected by analyzing multiple sources of information.
  - The response to the attack, which may range from alerting an administrator to automatically stopping the attack (e.g. killing an offending process), to tracing back the attack in order to identify the attacker.
    - Another approach is to divert the attacker to a ***honeypot***, on a ***honeynet.*** The idea behind a honeypot is a computer running normal services, but which no one uses to do any real work. Such a system should not see any network traffic under normal conditions, so any traffic going

to or from such a system is by definition suspicious. Honeypots are normally kept on a honeynet protected by a ***reverse firewall,*** which will let potential attackers in to the honeypot, but will not allow any outgoing traffic. (So that if the honeypot is compromised, the attacker cannot use it as a base of operations for attacking other systems.) Honeypots are closely watched, and any suspicious activity carefully logged and investigated.

- Intrusion Detection Systems, IDSs, raise the alarm when they detect an intrusion. Intrusion Detection and Prevention Systems, IDPs, act as filtering routers, shutting down suspicious traffic when it is detected.
- There are two major approaches to detecting problems:
    - ***Signature-Based Detection*** scans network packets, system files, etc. looking for recognizable characteristics of known attacks, such as text strings for messages or the binary code for "exec /bin/sh". The problem with this is that it can only detect previously encountered problems for which the signature is known, requiring the frequent update of signature lists.
    - ***Anomaly Detection*** looks for "unusual" patterns of traffic or operation, such as unusually heavy load or an unusual number of logins late at night.
        - The benefit of this approach is that it can detect previously unknown attacks, so called ***zero-day attacks.***
        - One problem with this method is characterizing what is "normal" for a given system. One approach is to benchmark the system, but if the attacker is already present when the benchmarks are made, then the "unusual" activity is recorded as "the norm."
        - Another problem is that not all changes in system performance are the result of security attacks. If the system is bogged down and really slow late on a Thursday night, does that mean that a hacker has gotten in and is using the system to send out SPAM, or does it simply mean that a CS 385 assignment is due on Friday? :-)
        - To be effective, anomaly detectors must have a very low ***false alarm (false positive)*** rate, lest the warnings get ignored, as well as a low ***false negative*** rate in which attacks are missed.

Virus Protection

- Modern anti-virus programs are basically signature-based detection systems, which also have the ability (in some cases) of ***disinfecting*** the affected files and returning them back to their original condition.
- Both viruses and anti-virus programs are rapidly evolving. For example viruses now commonly mutate every time they propagate, and so anti-virus programs look for families of related signatures rather than specific ones.
- Some antivirus programs look for anomalies, such as an executable program being opened for writing (other than by a compiler.)

- Avoiding bootleg, free, and shared software can help reduce the chance of catching a virus, but even shrink-wrapped official software has on occasion been infected by disgruntled factory workers.
- Some virus detectors will run suspicious programs in a ***sandbox,*** an isolated and secure area of the system which mimics the real system.
- ***Rich Text Format, RTF,*** files cannot carry macros, and hence cannot carry Word macro viruses.
- Known safe programs (e.g. right after a fresh install or after a thorough examination) can be digitally signed, and periodically the files can be re-verified against the stored digital signatures. (Which should be kept secure, such as on off-line write-only medium.)

Auditing, Accounting, and Logging

- Auditing, accounting, and logging records can also be used to detect anomalous behavior.
- Some of the kinds of things that can be logged include authentication failures and successes, logins, running of suid or sgid programs, network accesses, system calls, etc. In extreme cases almost every keystroke and electron that moves can be logged for future analysis. (Note that on the flip side, all this detailed logging can also be used to analyze system performance. The down side is that the logging also *affects* system performance (negatively!), and so a Heisenberg effect applies.)
- Cliff Stoll detected one of the early UNIX break-ins when he noticed anomalies in the accounting records on a computer system being used by physics researchers.

Firewalling to Protect Systems and Networks

- Firewalls are devices (or sometimes software) that sit on the border between two security domains and monitor/log activity between them, sometimes restricting the traffic that can pass between them based on certain criteria.
- For example a firewall router may allow HTTP: requests to pass through to a web server inside a company domain while not allowing telnet, ssh, or other traffic to pass through.
- A common architecture is to establish a de-militarized zone, DMZ, which sort of sits "between" the company domain and the outside world, as shown below. Company computers can reach either the DMZ or the outside world, but outside computers can only reach the DMZ. Perhaps most importantly, the DMZ cannot reach any of the other company computers, so even if the DMZ is breached, the attacker cannot get to the rest of the company network. (In some cases the DMZ may have limited access to company computers, such as a web server on the DMZ that needs to query a database on one of the other company computers.)

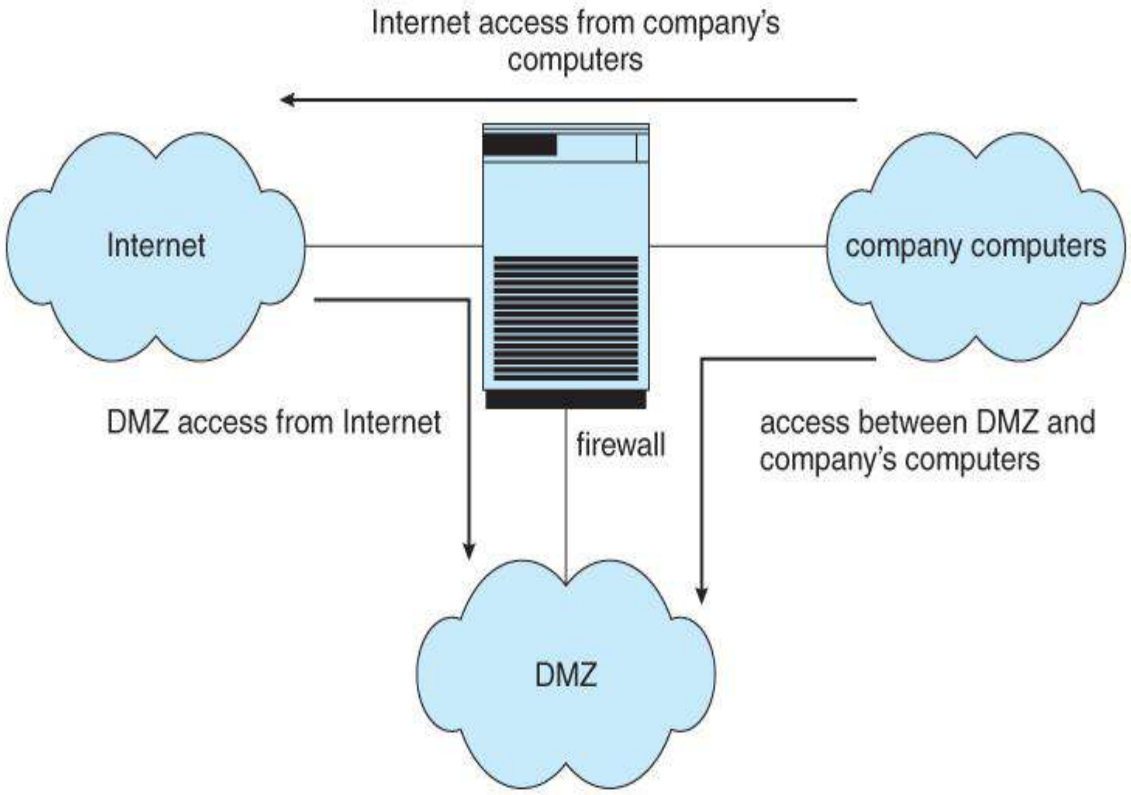

Internet access from company's computers
Internet
company computers
DMZ access from Internet
firewall
access between DMZ and company's computers
DMZ

Figure - Domain separation via firewall.

- Firewalls themselves need to be resistant to attacks, and unfortunately have several vulnerabilities:
    - ***Tunneling,*** which involves encapsulating forbidden traffic inside of packets that are allowed.
    - Denial of service attacks addressed at the firewall itself.
    - Spoofing, in which an unauthorized host sends packets to the firewall with the return address of an authorized host.
- In addition to the common firewalls protecting a company internal network from the outside world, there are also some specialized forms of firewalls that have been recently developed:
    - A ***personal firewall*** is a software layer that protects an individual computer. It may be a part of the operating system or a separate software package.
    - An ***application proxy firewall*** understands the protocols of a particular service and acts as a stand-in (and relay) for the particular service. For example, and SMTP proxy firewall would accept SMTP requests from the outside world, examine them for security concerns, and forward only the "safe" ones on to the real SMTP server behind the firewall.
    - ***XML firewalls*** examine XML packets only, and reject ill-formed packets. Similar firewalls exist for other specific protocols.
    - ***System call firewalls*** guard the boundary between user mode and system mode, and reject any system calls that violate security policies.

Computer-Security Classifications (Optional)

- No computer system can be 100% secure, and attempts to make it so can quickly make it unusable.
- However one can establish a level of trust to which one feels "safe" using a given computer system for particular security needs.
- The U.S. Department of Defense's "Trusted Computer System Evaluation Criteria" defines four broad levels of trust, and sub-levels in some cases:
    - Level D is the least trustworthy, and encompasses all systems that do not meet any of the more stringent criteria. DOS and Windows 3.1 fall into level D, which has no user identification or authorization, and anyone who sits down has full access and control over the machine.
    - Level C1 includes user identification and authorization, and some means of controlling what users are allowed to access what files. It is designed for use by a group of mostly cooperating users, and describes most common UNIX systems.
    - Level C2 adds individual-level control and monitoring. For example file access control can be allowed or denied on a per-individual basis, and the system administrator can monitor and log the activities of specific individuals. Another restriction is that when one user uses a system resource and then returns it back to the system, another user who uses the same resource later cannot read any of the information that the first user stored there. (I.e. buffers, etc. are wiped out between users, and are not left full of old contents.) Some special secure versions of UNIX have been certified for C2 security levels, such as SCO.
    - Level B adds sensitivity labels on each object in the system, such as "secret", "top secret", and "confidential". Individual users have different clearance levels, which controls which objects they are able to access. All human-readable documents are labeled at both the top and bottom with the sensitivity level of the file.
    - Level B2 extends sensitivity labels to all system resources, including devices. B2 also supports covert channels and the auditing of events that could exploit covert channels.
    - B3 allows creation of access-control lists that denote users NOT given access to specific objects.
    - Class A is the highest level of security. Architecturally it is the same as B3, but it is developed using formal methods which can be used to ***prove*** that the system meets all requirements and cannot have any possible bugs or other vulnerabilities. Systems in class A and higher may be developed by trusted personnel in secure facilities.
    - These classifications determine what a system *can* implement, but it is up to security policy to determine *how* they are implemented in practice. These systems and policies can be reviewed and certified by trusted organizations, such as the National Computer Security Center. Other standards may dictate physical protections and other issues.