CHAPTER 1

1.1 OPERATING SYSTEMS OVERVIEW

1.1.1 Operating systems functions

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:

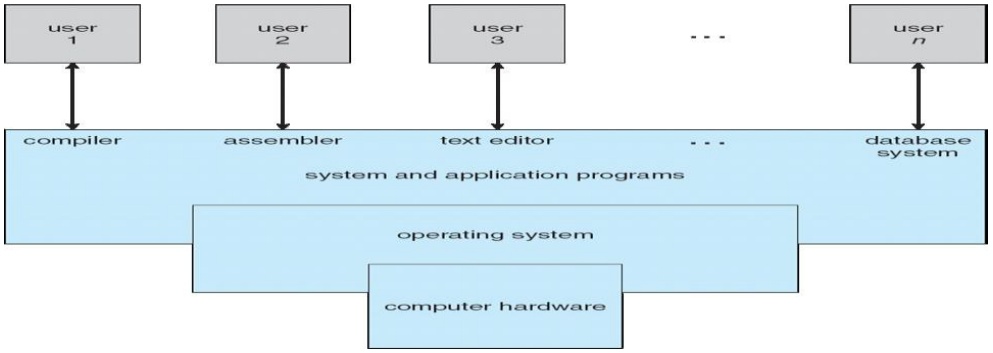
- Execute user programs and make solving user problems easier
- Make the computer system convenient to use
- Use the computer hardware in an efficient manner

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



Process Management

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

Process Management Activities

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

Memory Management

- All data in memory before and after processing
- All instructions in memory in order to execute
- Memory management determines what is in memory when
- Optimizing CPU utilization and computer response to users
- **Memory management activities**
- Keeping track of which parts of memory are currently being used and by whom
- Deciding which processes (or parts thereof) and data to move into and out of memory
- Allocating and deallocating memory space as needed

Storage Management

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

OS activities include

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

Mass-Storage Management

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

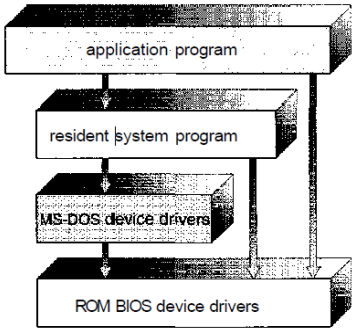
MASS STORAGE activities

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

1.1.2 Operating-System Structure

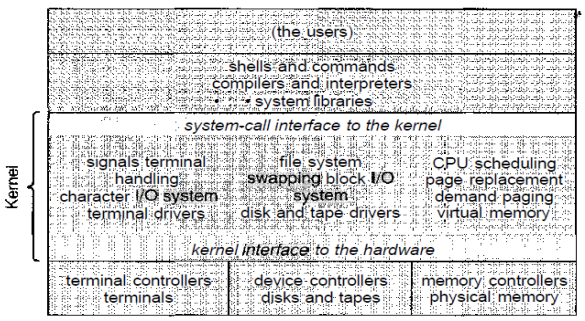
Simple Structure

Many commercial systems do not have well-defined structures. Frequently, such operating systems started as small, simple, and limited systems and then grew beyond their original scope. MS-DOS is an example of such a system.



It was written to provide the most functionality in the least space, so it was not divided into modules carefully. In MS-DOS, the interfaces and levels of functionality are not well separated. For instance, application programs are able to access the basic I/O routines to write directly to the display and disk drives. Such freedom leaves MS-DOS vulnerable to errant (or malicious) programs, causing entire system crashes when user programs fail. Of course, MS-DOS was also limited by the hardware of its era. Another example of limited structuring is the original UNIX operating system. UNIX is another system that initially was limited by hardware functionality.

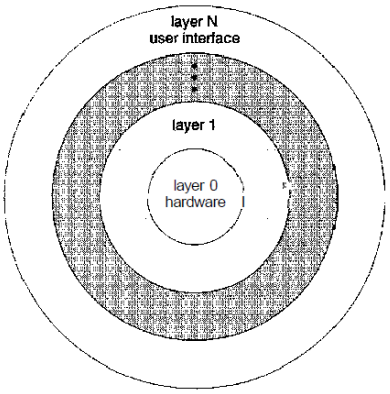
It consists of two separable parts: the kernel and the system programs. The kernel is further separated into a series of interfaces and device drivers, which have been added and expanded over the years as UNIX has evolved.



Layered Approach

The operating system can then retain much greater control over the computer and over the applications that make use of that computer. Implementers have more freedom in changing the inner workings of the system and in creating modular operating systems. Under the top down approach, the overall functionality and features are determined and are separated into components. Information hiding is also important, because it leaves programmers free to implement the low-level routines as they see fit, provided that the external interface of the routine stays unchanged and that the routine itself performs the advertised task.

A system can be made modular in many ways. One method is the **layered approach,** in which the operating system is broken up into a number of layers (levels). The bottom layer (layer 0) is the hardware; the highest (layer N) is the user interface.



An operating-system layer is an implementation of an abstract object made up of data and the operations that can manipulate those data. A typical operating-system layer—say, layer M—consists of data structures and a set of routines that can be invoked by higher-level layers. Layer *M,* in turn, can invoke operations on lower-level layers.

The main advantage of the layered approach is simplicity of construction and debugging. The layers are selected so that each uses functions (operations) and services of only lower-level layers. This approach simplifies debugging and system verification. The first layer can be debugged without any concern for the rest of the system, because, by definition, it uses only the basic hardware (which is assumed correct) to implement its functions. Once the first layer is debugged, its correct functioning can be assumed while the second layer is debugged, and

so on. If an error is found during the debugging of a particular layer, the error must be on that layer, because the layers below it are already debugged. Thus, the design and implementation of the system is simplified.

Each layer is implemented with only those operations provided by lower level layers. A layer does not need to know how these operations are implemented; it needs to know only what these operations do. Hence, each layer hides the existence of certain data structures, operations, and hardware from higher-level layers.

The major difficulty with the layered approach involves appropriately defining the various layers. The backing-store driver would normally be above the CPU scheduler, because the driver may need to wait for I/O and the CPU can be rescheduled during this time. A final problem with layered implementations is that they tend to be less efficient than other types. For instance, when a user program executes an I/O operation, it executes a system call that is trapped to the I/O layer, which calls the memory-management layer, which in turn calls the CPU-scheduling layer, which is then passed to the hardware.

Micro kernels

The kernel became large and difficult to manage. In the mid-1980s, researchers at Carnegie Mellon University 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. microkernels provide minimal process and memory management, in addition to a communication facility.

The main function of the microkernel is to provide a communication facility between the client program and the various services that are also running in user space. One benefit of the microkernel approach is ease of extending the operating system. All new services are added to user space and consequently do not require modification of the kernel. When the kernel does have to be modified, the changes tend to be fewer, because the microkernel is a smaller kernel.

The resulting operating system is easier to port from one hardware design to another. The microkernel also provides more security and reliability, since most services are running as user rather than kernel processes. If a service fails, the rest of the operating system remains untouched.

Modules

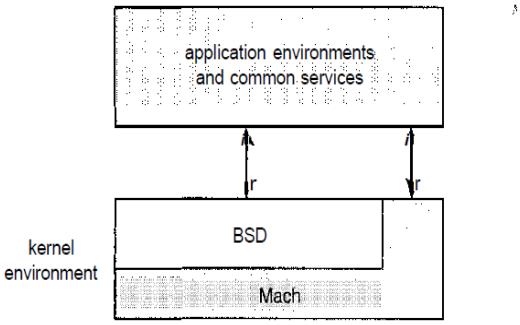
The best current methodology for operating-system design involves using object-oriented programming techniques to create a modular kernel. Here, the kernel has a set of core components and dynamically links in additional services either during boot time or during run time. Such a strategy uses dynamically loadable modules and is common in modern implementations of UNIX, such as Solaris, Linux, and Mac OS X.

A core kernel with seven types of loadable kernel modules:

1. Scheduling classes
2. File systems
3. Loadable system calls
4. Executable formats
5. STREAMS modules
6. Miscellaneous
7. Device and bus drivers

Such a design allows the kernel to provide core services yet also allows certain features to be implemented dynamically. The overall result resembles a layered system in that each kernel section has defined, protected interfaces; but it is more flexible than a layered system in that any module can call any other module. The approach is like the microkernel approach in that the primary module has only core functions and knowledge of how to load and communicate with other modules; but it is more efficient, because modules do not need to invoke message passing in order to communicate.

The Apple Macintosh Mac OS X operating system uses a hybrid structure. Mac OS X (also known as *Danvin)* structures the operating system using a layered technique where one layer consists of the Mach microkernel. The top layers include application environments and a set of services providing a graphical interface to applications. Below these layers is the kernel environment, which consists primarily of the Mach microkernel and the BSD kernel. Mach provides memory management; support for remote procedure calls (RPCs) and inter process communication (IPC) facilities, including message passing; and thread scheduling. The BSD component provides a BSD command line interface, support for networking and file systems, and an implementation of POSIX APIs, including Pthreads.



1.1.3 Operating-System Operations

1. Modern operating systems are **interrupt driven.** If there are no processes to execute, no I/O devices to service, and no users to whom to respond, an operating system will sit quietly, waiting for something to happen. Events are almost always signaled by the occurrence of an interrupt or a trap
2. A **trap (or** an **exception)** is a software-generated interrupt caused either by an error or by a specific request from a user program that an operating-system service be performed.
3. The interrupt-driven nature of an operating system defines that system's general structure. For each type of interrupt, separate segments of code in the operating system determine what action should be taken. An interrupt service routine is provided that is responsible for dealing with the interrupt.
4. The operating system and the users share the hardware and software resources of the computer system, we need to make sure that an error in a user program could cause problems only for the one program that was running. With sharing, many processes could be adversely affected by a bug in one program. For example, if a process gets stuck in an infinite loop, this loop could prevent the correct operation of many other processes.
5. Without protection against these sorts of errors, either the computer must execute only one process at a time or all output must be suspect.

Dual-Mode Operation

Dual-mode operation allows OS to protect itself and other system components

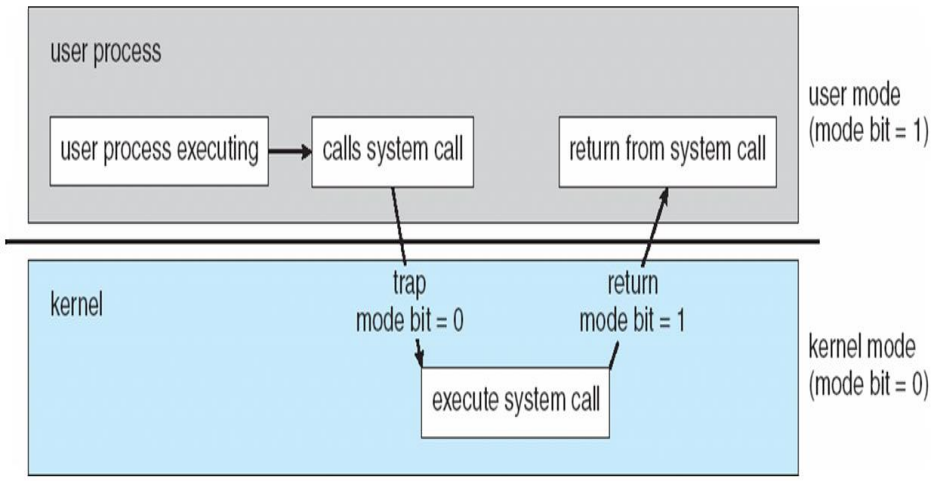
User mode and **kernel mode**

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

Transition from User to Kernel Mode

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

Prot



If a computer system has multiple users and allows the concurrent execution of multiple processes, then access to data must be regulated. For that purpose, mechanisms ensure that files, memory segments, CPU, and other resources can be operated on by only those processes that have gained proper authorization from the operating system.

1.1.4 Protection and security

Protection is any mechanism for controlling the access of processes or users to the resources defined by a computer system. This mechanism must provide means for specification of the controls to be imposed and means for enforcement.

Protection can improve reliability by detecting latent errors at the interfaces between component subsystems. Early detection of interface errors can often prevent contamination of a healthy subsystem by another subsystem that is malfunctioning. An unprotected resource cannot defend against use (or misuse) by an unauthorized or incompetent user. A protection-oriented system

provides a means to distinguish between authorized and unauthorized usage, A system can have adequate protection but still be prone to failure and allow inappropriate access.

It is the job of **security** to defend a system from external and internal attacks. Such attacks spread across a huge range and include viruses and worms, denial-of service attacks Protection and security require the system to be able to distinguish among all its users. Most operating systems maintain a list of user names and associated **user identifiers (user IDs).**

- 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

1.1.5 Kernel Data Structures

The operating system must keep a lot of information about the current state of the system. As things happen within the system these data structures must be changed to reflect the current reality. For example, a new process might be created when a user logs onto the system. The kernel must create a data structure representing the new process and link it with the data structures representing all of the other processes in the system.

Mostly these data structures exist in physical memory and are accessible only by the kernel and its subsystems. Data structures contain data and pointers, addresses of other data structures, or the addresses of routines. Taken all together, the data structures used by the Linux kernel can look very confusing. Every data structure has a purpose and although some are used by several kernel subsystems, they are more simple than they appear at first sight.

Understanding the Linux kernel hinges on understanding its data structures and the use that the various functions within the Linux kernel makes of them. This section bases its description of the Linux kernel on its data structures. It talks about each kernel subsystem in terms of its algorithms, which are its methods of getting things done, and their usage of the kernel's data structures.

1.1.6 Computing Environments

Traditional Computing

As computing matures, the lines separating many of the traditional computing environments are blurring. this environment consisted of PCs connected to a network, with servers providing file and print services. Terminals attached to mainframes were prevalent at many companies as well, with even fewer remote access and portability options.

The current trend is toward providing more ways to access these computing environments. Web technologies are stretching the boundaries of traditional computing. Companies establish **portals,** which provide web accessibility to their internal servers. **Network computers** are essentially terminals that understand web-based computing. Handheld computers can synchronize with PCs to allow very portable use of company information. Handheld PDAs can also connect to **wireless networks** to use the company's web portal.

Batch system processed jobs in bulk, with predetermined input. Interactive systems waited for input from users. To optimize the use of the computing resources, multiple users shared time on these systems. Time-sharing systems used a timer and scheduling algorithms to rapidly cycle processes through the CPU, giving each user a share of the resources.

Client-Server Computing

Designers have shifted away from centralized system architecture. Terminals connected to centralized systems are now being supplanted by PCs. Correspondingly, user interface functionality once handled directly by the centralized systems is increasingly being handled by the PCs. As a result, many of today's systems act as **server systems to** satisfy requests generated by **client systems** Server systems can be broadly categorized as compute servers and file servers:

• The **compute-server system** provides an interface to which a client can send a request to perform an action (for example, read data); in response, the server executes the action and sends back results to the client. A server running a database that responds to client requests for data is an example of such a system.

The **file-server system** provides a file-system interface where clients can create, update, read, and delete files. An example of such a system is a web server that delivers files to clients running web browsers.

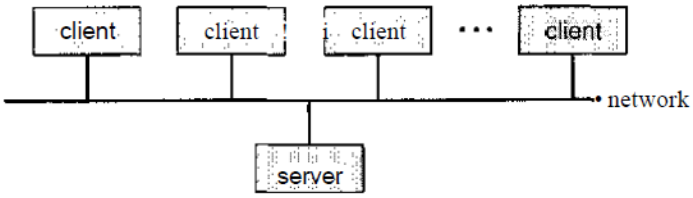


Figure 1.11 General structure of a client-server system.

Peer-to-Peer Computing

In this model, clients and servers are not distinguished from one another; instead, all nodes within the system are considered peers, and each may act as either a client or a server, depending on whether it is requesting or providing a service. Peer-to-peer systems offer an advantage over traditional client-server systems. In a client-server system, the server is a bottleneck; but in a peer-to-peer system, services can be provided by several nodes distributed throughout the network.

To participate in a peer-to-peer system, a node must first join the network of peers. Once a node has joined the network, it can begin providing services to—and requesting services from—other nodes in the network.

Determining what services are available is accomplished in one of two general ways:

- When a node joins a network, it registers its service with a centralized lookup service on the network. Any node desiring a specific service first contacts this centralized lookup service to determine which node provides the service. The remainder of the communication takes place between the client and the service provider.
- A peer acting as a client must first discover what node provides a desired service by broadcasting a request for the service to all other nodes in the network. The node (or nodes) providing that service responds to the peer making the request. To support this approach, a *discovery protocol* must be provided that allows peers to discover services provided by other peers in the network.

Web-Based Computing

The Web has become ubiquitous, leading to more access by a wider variety of devices than was dreamt of a few years ago. Web computing has increased the

emphasis on networking. Devices that were not previously networked now include wired or wireless access. Devices that were networked now have faster network connectivity, provided by either improved networking technology, optimized network implementation code, or both.

The implementation of web-based computing has given rise to new categories of devices, such as **load balancers,** which distribute network connections among a pool of similar servers. Operating systems like Windows 95, which acted as web clients, have evolved into Linux and Windows XP, which can act as web servers as well as clients. Generally, the Web has increased the complexity of devices, because their users require them to be web-enabled.

1.1.7 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

1.2 OPERATING SYSTEM STRUCTURE

1.2.1 Operating System Services

- One set of operating-system services provides functions that are helpful to the user Communications – Processes may exchange information, on the same computer or between computers over a network.
- Communications may be via shared memory or through message passing (packets moved by the OS)
- Error detection – OS needs to be constantly aware of possible errors may occur in the CPU and memory hardware, in I/O devices, in user program
- For each type of error, OS should take the appropriate action to ensure correct and consistent computing.
- Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system.
- Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing
- **Resource allocation -** When multiple users or multiple jobs running concurrently, resources must be allocated to each of them

- Many types of resources - Some (such as CPU cycles, main memory, and file storage) may have special allocation code, others (such as I/O devices) may have general request and release code
- **Accounting -** To keep track of which users use how much and what kinds of computer resources
- **Protection and security -** The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other.
- **Protection** involves ensuring that all access to system resources is controlled.
- **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts.
- If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link.

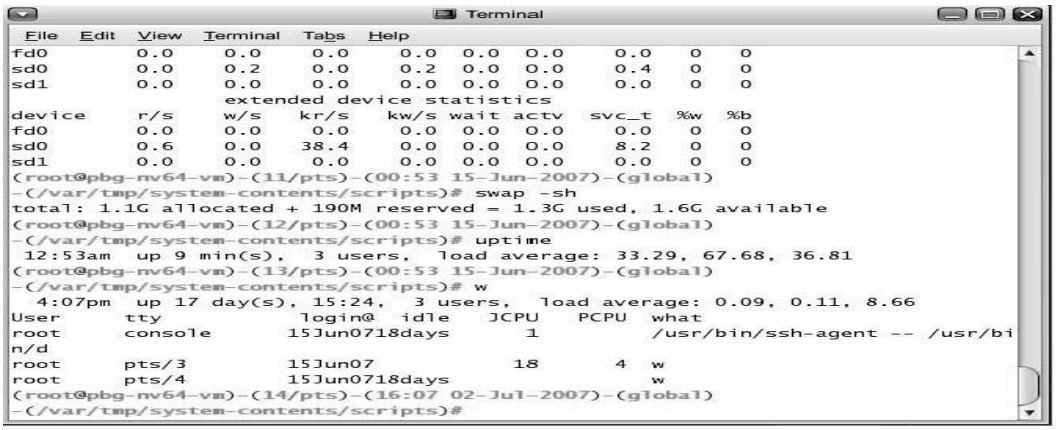
1.2.2 User and Operating System Interface - CLI

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

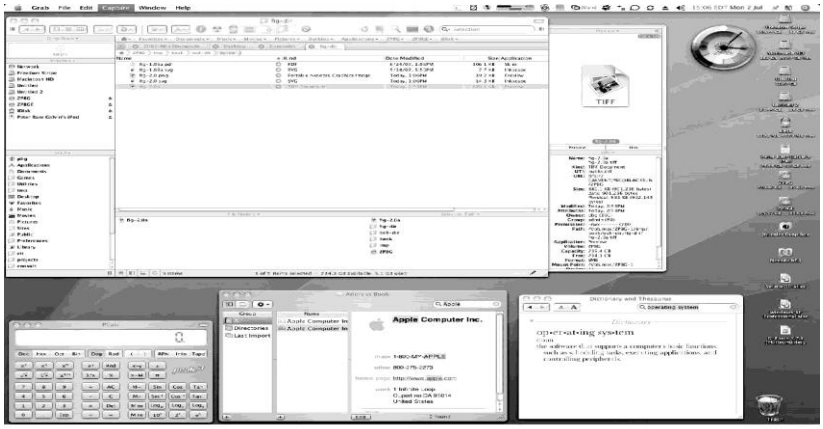
User Operating System Interface - GUI

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

Bourne Shell Command Interpreter



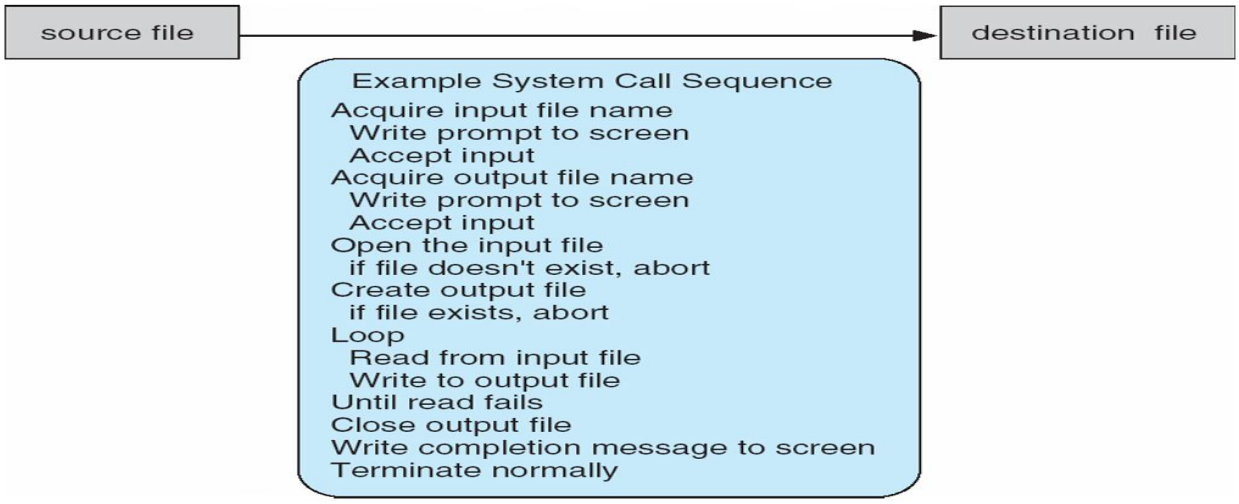
The Mac OS X GUI



1.2.3 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 usenThree most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)
- Why use APIs rather than system calls?(Note that the system-call names used throughout this text are generic)

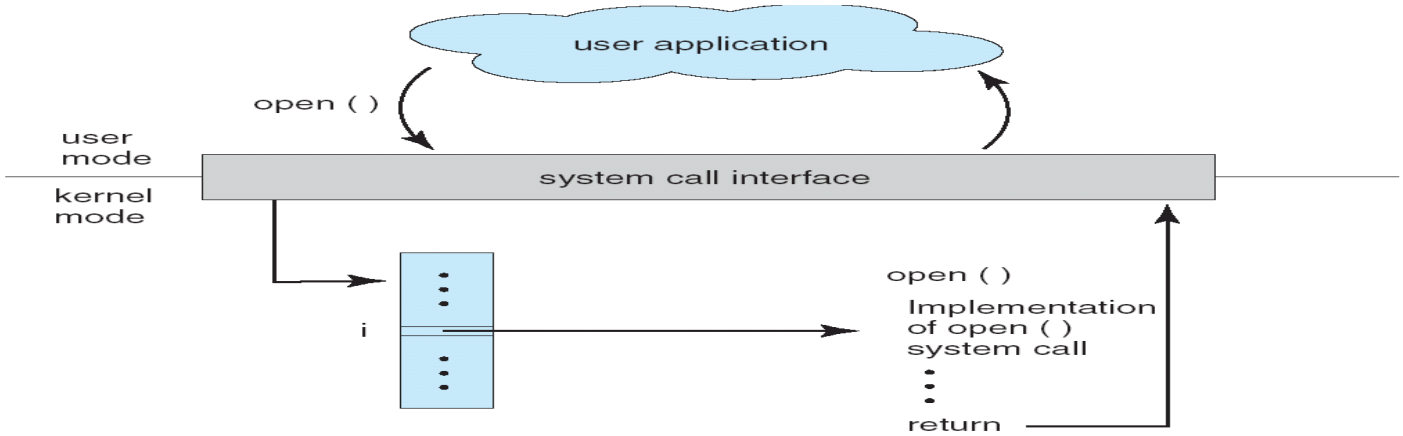
Example of System Calls



System Call Implementation

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

API – System Call – OS Relationship



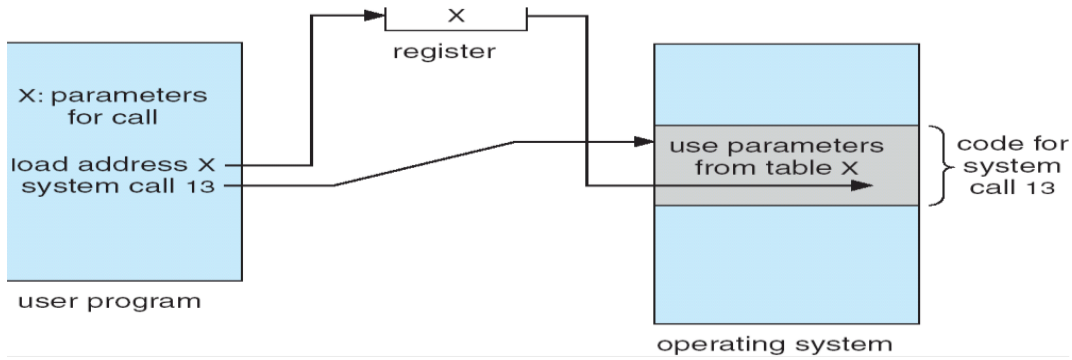
System Call Parameter Passing

- Often, more information is required than simply identity of desired system call
- Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
- Simplest: pass the parameters in *registers*
  - In some cases, may be more parameters than registers
- Parameters stored in a *block,* or table, in memory, and address of block passed as a parameter in a register

This approach taken by Linux and Solaris

- Parameters placed, or *pushed,* onto the *stack* by the program and *popped* off the stack by the operating system
- Block and stack methods do not limit the number or length of parameters being passed

Parameter Passing via Table



1.2.4 Types of System Calls

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

Process Control

A running program needs to be able to halt its execution either normally (end) or abnormally (abort). If a system call is made to terminate the currently running program abnormally, or if the program runs into a problem and causes an error

trap, a dump of memory is sometimes taken and an error message generated. The dump is written to disk and may be examined by a **debugger**—a system program designed to aid the programmer in finding and correcting bugs-—to determine the cause of the problem. Under either normal or abnormal circumstances, the operating system must transfer control to the invoking command interpreter. The command interpreter then reads the next command. In an interactive system, the command interpreter simply continues with the next command; it is assumed that the user will issue an appropriate command to respond to any error.

File Management

We first need to be able to create and delete files. Either system call requires the name of the file and perhaps some of the file's attributes. Once the file is created, we need to open it and to use it. We may also read, write, or reposition (rewinding or skipping to the end of the file, for example). Finally, we need to close the file, indicating that we are no longer using it. We may need these same sets of operations for directories if we have a directory structure for organizing files in the file system. In addition, for either files or directories, we need to be able to determine the values of various attributes and perhaps to reset them if necessary. File attributes include the file name, a file type, protection codes, accounting information, and so on.

At least two system calls, get file attribute and set file attribute, are required for this function. Some operating systems provide many more calls, such as calls for file move and copy.

Device Management

A process may need several resources to execute—main memory, disk drives, access to files, and so on. If the resources are available, they can be granted, and control can be returned to the user process. Otherwise, the process will have to wait until sufficient resources are available. The various resources controlled by the operating sysstem can be thought of as devices. Some of these devices are physical devices (for example, tapes), while others can be thought of as abstract or virtual devices (for example, files). If there are multiple users of the system, the system may require us to first request the device, to ensure exclusive use of it. After we are finished with the device, we release it. These functions are similar to the open and close system calls for files.

Information Maintenance

Many system calls exist simply for the purpose of transferring information between the user program and the operating system. For example, most systems have a system call to return the current t i m e and date . Other system calls may return information about the system, such as the number of current users, the version number of the operating system, the amount of free memory or disk space, and so on.

In addition, the operating system keeps information about all its processes, and system calls are used to access this information. Generally, calls are also used to reset the process information (get process attributes and set process attributes) .

Communication

There are two common models of inter process communication: the message passing model and the shared-memory model. In the message-passing model, the communicating processes exchange messages with one another to transfer information. Messages can be exchanged between the processes either directly or indirectly through a common mailbox. Before communication can take place, a connection must be opened. The name of the other communicator must be known, be it another process on the same system or a process on another computer connected by a communications network. Each computer in a network has a *host name* by which it is commonly known. A host also has a network identifier, such as an IP address. Similarly, each process has a *process name,* and this name is translated into an identifier by which the operating system can refer to the process. The get host id and get processid system calls do this translation. The identifiers are then passed to the general purpose open and close calls provided by the file system or to specific open connection and close connection system calls, depending on the system's model of communication.

In the shared-memory model, processes use shared memory create and shared memory attach system calls to create and gain access to regions of memory owned by other processes. Recall that, normally, the operating system tries to prevent one process from accessing another process's memory. Shared memory requires that two or more processes agree to remove this restriction.

They can then exchange information by reading and writing data in the shared areas. The form of the data and the location are determined by the processes and are not under the operating system's control. The processes are

also responsible for ensuring that they are not writing to the same location simultaneously.

1.2.5 System Programs

At the lowest level is hardware. Next is the operating system, then the system programs, and finally the application programs. System programs provide a convenient environment for program development and execution. Some of them are simply user interfaces to system calls; others are considerably more complex.

They can be divided into these categories:

• **File management.** These programs create, delete, copy, rename, print, dump, list, and generally manipulate files and directories.

• **Status information.** Some programs simply ask the system for the date, time, amount of available memory or disk space, number of users, or similar status information. Others are more complex, providing detailed

performance, logging, and debugging information. Typically, these programs format and print the output to the terminal or other output devices or files or display it in a window of the GUI. Some systems also support a registry, which is used to store and retrieve configuration information.

• **File modification.** Several text editors may be available to create and modify the content of files stored on disk or other storage devices. There may also be special commands to search contents of files or perform transformations of the text.

• **Programming-language support.** Compilers, assemblers, debuggers and interpreters for common programming languages (such as C, C++, Java, Visual Basic, and PERL) are often provided to the user with the operating system.

• **Program loading and execution.** Once a program is assembled or compiled, it must be loaded into memory to be executed. The system may provide absolute loaders, relocatable loaders, linkage editors, and overlay loaders. Debugging systems for either higher-level languages or machine language are needed as well.

• **Communications.** These programs provide the mechanism for creating virtual connections among processes, users, and computer systems. They allow users to send messages to one another's screens, to browse web pages, to send electronic-mail messages, to log in remotely, or to transfer files from one machine to another.

In addition to systems programs, most In addition to systems programs, most operating systems are supplied with programs that are useful in solving common problems or performing common operations. Such programs include web browsers, word processors and text formatters, spreadsheets, database systems, compilers, plotting and statistical-analysis packages, and games. These programs are known as system utilities or application programs.

1.2.6 Operating-System Structure

Refer the page 10

1.2.7 Operating-System Debugging

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

1.2.8 System Boot

The procedure of starting a computer by loading the kernel is known as *booting* the system. On most computer systems, a small piece of code known as the **bootstrap program** or **bootstrap loader** locates the kernel, loads it into main memory, and starts its execution. Some computer systems, such as PCs, use a two-step process in which a simple bootstrap loader fetches a more complex boot program from disk, which in turn loads the kernel.

When a CPU receives a reset event—for instance, when it is powered up or rebooted—the instruction register is loaded with a predefined memory location, and execution starts there. At that location is the initial bootstrap program. This program is in the form of **read-only memory (ROM),** because the RAM is in an

unknown state at system startup. ROM is convenient because it needs no initialization and cannot be infected by a computer virus.

The bootstrap program can perform a variety of tasks. Usually, one task is to run diagnostics to determine the state of the machine. If the diagnostics pass, the program can continue with the booting steps. It can also initialize all aspects of the system, from CPU registers to device controllers and the contents of main memory. Sooner or later, it starts the operating system.

Some systems—such as cellular phones, PDAs, and game consoles—store the entire operating system in ROM. Storing the operating system in ROM is suitable for small operating systems, simple supporting hardware, and rugged operation. A problem with this approach is that changing the bootstrap code requires changing the ROM hardware chips. Some systems resolve this problem by using **erasable programmable read-only memory** (EPROM), which is read only except when explicitly given a command to become writable. All forms of ROM are also known as **firmware,** since their characteristics fall somewhere between those of hardware and those of software. A problem with firmware in general is that executing code there is slower than executing code in RAM.

Some systems store the operating system in firmware and copy it to RAM for fast execution. A final issue with firmware is that it is relatively expensive, so usually only small amounts are available.

For large operating systems (including most general-purpose operating systems like Windows, Mac OS X, and UNIX) or for systems that change frequently, the bootstrap loader is stored in firmware, and the operating system is on disk. In this case, the bootstrap runs diagnostics and has a bit of code that can read a single block at a fixed location (say block zero) from disk into memory and execute the code from that **boot block.** The program stored in the boot block may be sophisticated enough to load the entire operating system into memory and begin its execution. More typically, it is simple code (as it fits in a single disk block) and only knows the address on disk and length of the remainder of the bootstrap program. All of the disk-bound bootstrap, and the operating system itself, can be easily changed by writing new versions to disk.

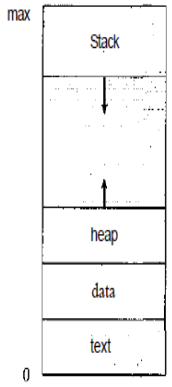
1.3 PROCESSES

1.3.1 Process concepts

Process: A process is a program in execution. A process is more than the program code, which is sometimes known as the **text section.** It also includes the current activity, as represented by the value of the **program counter** and the contents of the processor's registers. A process generally also includes the

process **stack,** which contains temporary data (such as function parameters, return addresses, and local variables), and a **data section,** which contains global variables. A process may also include a **heap,** which is memory that is dynamically allocated during process run time.

Structure of a process



We emphasize that a program by itself is not a process; a program is a *passive* entity, such as a file containing a list of instructions stored on disk (often called an **executable file),** whereas a process is an *active* entity, with a program counter specifying the next instruction to execute and a set of associated resources. A program becomes a process when an executable file is loaded into memory.

Two common techniques for loading executable files are double-clicking an icon representing the executable file and entering the name of the executable file on the command line (as in prog. exe or a. out.)

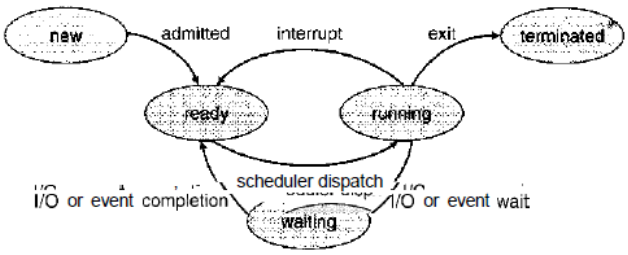


Figure 3.2 Diagram of process state.

Process State

As a process executes, it changes **state.** The state of a process is defined in part by the current activity of that process. Each process may be in one of the following states:

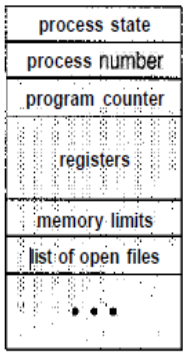
- **New.** The process is being created.
- **Running.** Instructions are being executed.
- **Waiting.** The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
- **Ready.** The process is waiting to be assigned to a processor.
- **Terminated.** The process has finished execution.

These names are arbitrary, and they vary across operating systems. The states that they represent are fotind on all systems, however. Certain operating systems also more finely delineate process states. It is important to realize that only one process can be *running* on any processor at any instant.

Process Control Block

Each process is represented in the operating system by a **process control block (PCB)**—also called a *task control block.*

Process state. The state may be new, ready, running, waiting, halted, and so on.



Program counter-The counter indicates the address of the next instruction to be executed for this process.

- CPU **registers-** The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information.

CPU-scheduling information- This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.

Memory-management information- This information may include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system

Accounting information-This information includes the amount of CPU and real time used, time limits, account members, job or process numbers, and so on.

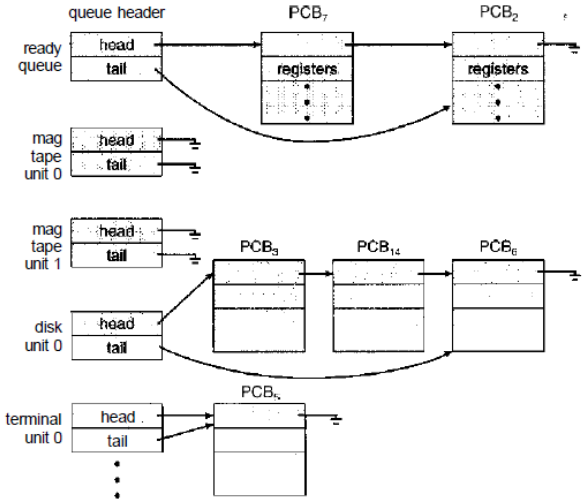
I/O status information-This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

1.3.2 Process Scheduling

The **process scheduler** selects an available process (possibly from a set of several available processes) for program execution on the CPU.

As processes enter the system, they are put into a **job queue,** which consists of all processes in the system. The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue.**

This queue is generally stored as a linked list. A ready-queue header contains pointers to the first and final PCBs in the list. Each PCB includes a pointer field that points to the next PCB in the ready queue.



Each rectangular box represents a queue. Two types of queues are present: the ready queue and a set of device queues. The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.

A new process is initially put in the ready queue. It waits there till it is selected for execution, or is **dispatched.** Once the process is allocated the CPU and is executing, one of several events could occur:

- The process could issue an I/O request and then be placed in an I/O queue.
- The process could create a new sub process and wait for the subprocess's termination.

• The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

Schedulers

A process migrates among the various scheduling queues throughout its lifetime. The operating system must select, for scheduling purposes, processes from these queues in some fashion.

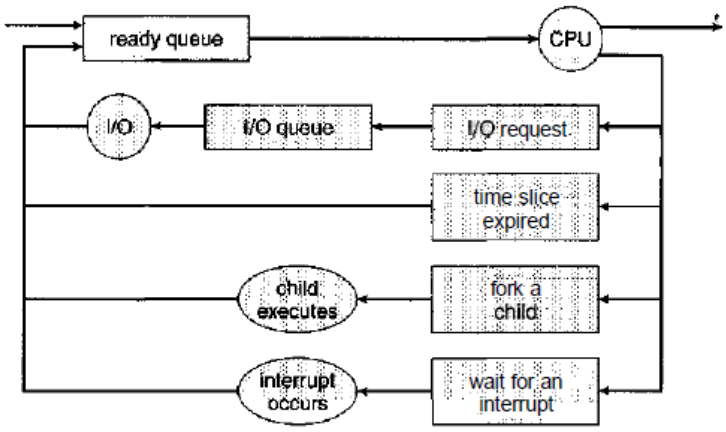


Figure 3.7 Queueing-diagram representation of process scheduling.

The selection process is carried out by the appropriate **scheduler.** The **long-term scheduler,** or **job scheduler,** selects processes from this pool and loads them into memory for execution. The **short-term scheduler, or** CPU **scheduler,** selects from among the processes that are ready to execute and allocates the CPU to one of them.

1.3.3 Operations on Processes

Process Creation

A process may create several new processes, via a create-process system call, during the course of execution. The creating process is called a **parent** process, and the new processes are called the **children** of that process. Each of these new processes may in turn create other processes, forming a **tree** of processes.

Most operating systems identify processes according to a unique **process identifier** (or **pid),** which is typically an integer number. These processes are responsible for managing memory and file systems. The sched process also creates the init process, which serves as the root parent process for all user processes.

When a process creates a new process, two possibilities exist in terms of execution:

1. The parent continues to execute concurrently with its children.
2. The parent waits until some or all of its children have terminated.

There are also two possibilities in terms of the address space of the new process:
1. The child process is a duplicate of the parent process (it has the same program and data as the parent).
2. The child process has a new program loaded into it.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main()
{
pid-t pid;
/* fork a child process */
pid = fork();
if (pid < 0) {/* error occurred */
fprintf(stderr, "Fork Failed");
exit (-1) ;
}
else if (pid == 0} {/* child process */
execlpf"/bin/Is","Is",NULL);
}
else {/* parent process */
/* parent will wait for the child to complete */
wait(NULL);
printf("Child Complete");
exit (0) ;

}

}
```

In UNIX, as we've seen, each process is identified by its process identifier, which is a unique integer. A new process is created by the fork() system call. The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process. Both processes (the parent and the child) continue execution at the instruction after the f ork(), with one difference: The return code for the fork() is

zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent. the exec() system call is used after a fork() system call by one of the two processes to replace the process's memory space with a new program. The exec () system call loads a binary file into memory (destroying the memory image of the program containing the execO system call) and starts its execution.

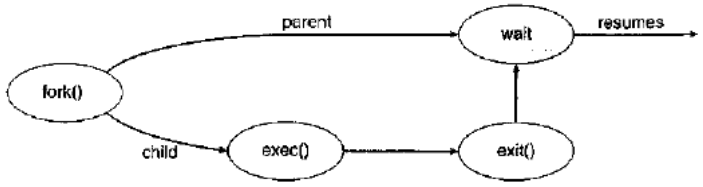


Figure 3.11 Process creation.

Process Termination

A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit () system call. At that point, the process may return a status value (typically an integer) to its parent process (via the wait() system call). All the resources of the process—including physical and virtual memory, open files, and I/O buffers—are deallocated by the operating system.

Termination can occur in other circumstances as well. A process can cause the termination of another process via an appropriate system call (for example, TerminateProcessO in Win32). Usually, such a system call can be invoked only by the parent of the process that is to be terminated.

A parent may terminate the execution of one of its children for a variety of reasons, such as these:

• The child has exceeded its usage of some of the resources that it has been allocated.

• The task assigned to the child is no longer required.

• The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

Consider that, in UNIX, we can terminate a process by using the exit() system call; its parent process may wait for the termination of a child process by using the wait() system call. The wait () system call returns the process identifier

of a terminated child so that the parent can tell which of its possibly many children has terminated.

If the parent terminates, however, all its children have assigned as their new parent the init process.

1.3.4 Interprocess Communication

Processes executing concurrently in the operating system may be either independent processes or cooperating processes. A process is **independent** if it cannot affect or be affected by the other processes executing in the system.
Any process that does not share data with any other process is independent. A process is **cooperating** if **it** can affect or be affected by the other processes executing in the system.

There are several reasons for providing an environment that allows process cooperation:

• **Information sharing.** Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.

• **Computation speedup.** If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing elements (such as CPUs or I/O channels).

• **Modularity.** We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads. • **Convenience.** Even an individual user may work on many tasks at the same time. For instance, a user may be editing, printing, and compiling in parallel.

Cooperating processes require an **interprocess communication (IPC)** mechanism that will allow them to exchange data and information. There are two fundamental models of interprocess communication:

(1) shared memory and (2) **message passing.** In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region. In the message passing model, communication takes place by means of messages exchanged between the cooperating processes.

Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided. Message passing is also easier to implement than is shared memory for intercomputer communication. Shared

memory allows maximum speed and convenience of communication, as it can be done at memory speeds when within a computer.

Shared memory is faster than message passing, as message-passing systems are typically implemented using system calls and thus require the more time consuming task of kernel intervention.

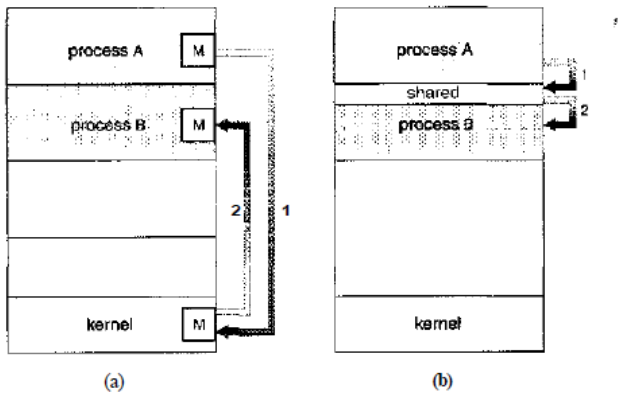


Figure 3.13 Communications models. (a) Message passing. (b) Shared memory.

Shared-Memory Systems Inter process communication using shared memory requires communicating processes to establish a region of shared memory. Typically, a shared-memory region resides in the address space of the process creating the shared-memory segment. Other processes that wish to communicate using this shared-memory segment must attach it to their address space. The operating system tries to prevent one process from accessing another process's memory. Shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas.

Message-Passing Systems The scheme requires that these processes share a region of memory and that the code for accessing and manipulating the shared memory be written explicitly by the application programmer. Another way to achieve the same effect is for the operating system to provide the means for cooperating processes to communicate with each other via a message-passing facility. Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space and is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network.

A message-passing facility provides at least two operations: send(message) and receive(message). Messages sent by a process can be of either fixed or variable

size. If only fixed-sized messages can be sent, the system-level implementation is straightforward. This restriction, however, makes the task of programming more difficult. Conversely, variable-sized messages require a more complex system-level implementation, but the programming task becomes simpler. This is a common kind of tradeoff seen throughout operating system design.

Naming

Processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication.

Under direct communication, each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, the send.0 and receive() primitives are defined as:

• send(P, message)—Send a message to process P.

• receive (Q, message)—Receive a message from process Q.

A communication link in this scheme has the following properties:

• A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.

• A link is associated with exactly two processes.

• Between each pair of processes, there exists exactly one link.

The disadvantage in both of these schemes (symmetric and asymmetric) is the limited modularity of the resulting process definitions. Changing the identifier of a process may necessitate examining all other process definitions.

Synchronization

Communication between processes takes place through calls to send() and receive () primitives. There are different design options for implementing each primitive. Message passing may be either **blocking** or **nonblocking**— also known as **synchronous** and **asynchronous.**

• **Blocking send-** The sending process is blocked until the message is received by the receiving process or by the mailbox.

• **Nonblocking send-** The sending process sends the message and resumes operation.

• **Blocking receive-** The receiver blocks until a message is available.

• **Nonblocking receive-** The receiver retrieves either a valid message or a null.

Buffering

Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, such queues can be implemented in three ways:

• **Zero capacity-** The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.

• **Bounded capacity-** The queue has finite length *n;* thus, at most *n* messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue (either the message is copied or a pointer to the message is kept), and the sender can continue execution without waiting. The links capacity is finite, however. If the link is full, the sender must block until space is available in the queue.

• **Unbounded capacity-** The queues length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.

1.3.5 Examples of IPC Systems

An Example: POSIX Shared Memory

Several IPC mechanisms are available for POSIX systems, including shared memory and message passing. A process must first create a shared memory segment using the shmget () system call (shmget () is derived from SHared Memory GET).

The following example illustrates the use of shmget ():

segment_id = shmget(IPCJPRIVATE, size, SJRUSR | SJVVUSR) ;

This first parameter specifies the key (or identifier) of the shared-memory segment. If this is set to IPC-PRIVATE, a new shared-memory segment is created. The second parameter specifies the size (in bytes) of the shared memory segment. Finally, the third parameter identifies the mode, which indicates how the shared-memory segment is to be used—that is, for reading, writing, or both. By setting the mode to SJRUSR | SJVVUSR, we are indicating that the owner may read or write to the shared memory segment.

Processes that wish to access a shared-memory segment must attach it to their address space using the shmat () (SHared Memory ATtach) system call.

The call to shmat () expects three parameters as well. The first is the integer identifier of the shared-memory segment being attached, and the second is a pointer location in memory indicating where the shared memory will be attached. If we pass a value of NULL, the operating system selects the location

on the user's behalf. The third parameter identifies a flag that allows the shared memory region to be attached in read-only or read-write mode; by passing a parameter of 0, we allow both reads and writes to the shared region.

The third parameter identifies a mode flag. If set, the mode flag allows the shared-memory region to be attached in read-only mode; if set to 0, the flag allows both reads and writes to the shared region. We attach a region of shared memory using shmat () as follows:

shared_memory = (char *) shmat(id, NULL, 0);

If successful, shmat () returns a pointer to the beginning location in memory where the shared-memory region has been attached.

An Example: Windows XP

The Windows XP operating system is an example of modern design that employs modularity to increase functionality and decrease the time needed to implement new features. Windows XP provides support for multiple operating environments, or *subsystems,* with which application programs communicate via a message-passing mechanism. The application programs can be considered clients of the Windows XP subsystem server.

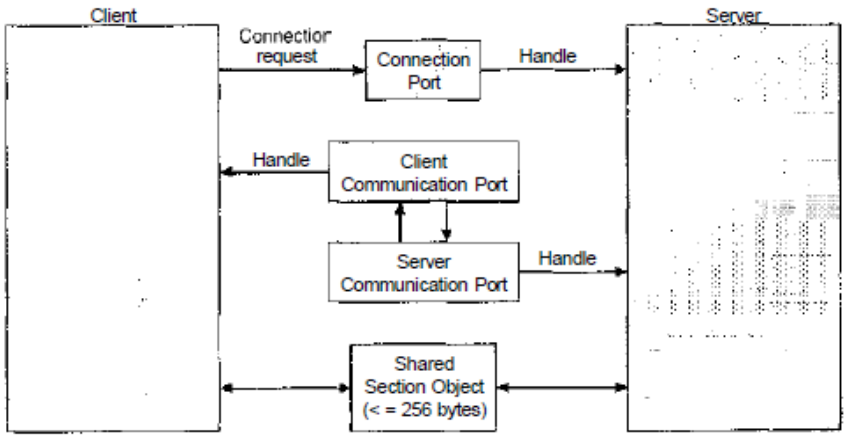
The message-passing facility in Windows XP is called the **local procedure call (LPC)** facility. The LPC in Windows XP communicates between two processes on the same machine. It is similar to the standard RPC mechanism that is widely used, but it is optimized for and specific to Windows XP. Windows XP uses a port object to establish and maintain a connection between two processes. Every client that calls a subsystem needs a communication channel, which is provided by a port object and is never inherited. Windows XP uses two types of ports: connection ports and communication ports. They are really the same but are given different names according to how they are used. Connection ports are named *objects* and are visible to all processes

The communication works as follows:

- The client opens a handle to the subsystem's connection port object.
- The client sends a connection request.
- The server creates two private communication ports and returns the handle to one of them to the client.
- The client and server use the corresponding port handle to send messages or callbacks and to listen for replies.

Windows XP uses two types of message-passing techniques over a port that the client specifies when it establishes the channel. The simplest, which is used for small messages, uses the port's message queue as intermediate storage and copies the message from one process to the other. Under this method, messages of up to 256 bytes can be sent. If a client needs to send a larger message, it passes the message through a section object, which sets up a region of shared memory. The client has to

decide when it sets up the channel whether or not it will need to send a large message. If the client determines that it does want to send large messages, it asks for a section object to be created. Similarly, if the server decides that replies will be large, it creates a section object. So that the section object can be used, a small message is sent that contains a pointer and size information about the section object. This method is more complicated than the first method, but it avoids data copying. In both cases, a callback mechanism can be used when either the client or the server cannot respond immediately to a request.



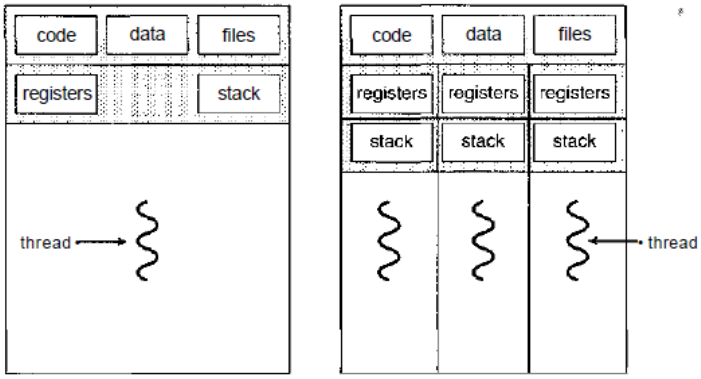
CHAPTER 2

2.1 THREADS

A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack. It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals.

2.1.1 Overview

A word processor may have a thread for displaying graphics, another thread for responding to keystrokes from the user, and a third thread for performing spelling and grammar checking in the background.



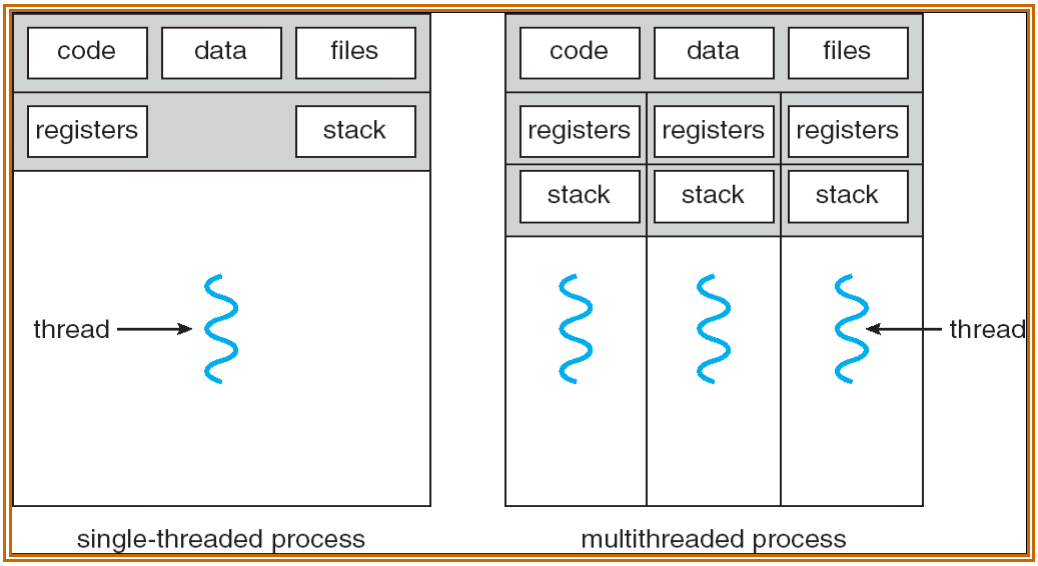
single-threaded process multithreaded process

In certain situations, a single application may be required to perform several similar tasks. For example, a web server accepts client requests for web pages, images, sound, and so forth. A busy web server may have several (perhaps thousands) of clients concurrently accessing it. If the web server ran as a traditional single-threaded process, it would be able to service only one client at a time. The amount of time that a client might have to wait for its request to be serviced could be enormous.

One solution is to have the server run as a single process that accepts requests. When the server receives a request, it creates a separate process to service that request. Threads also play a vital role in remote procedure call (RPC) systems. Finally, many operating system kernels are now multithreaded; several threads operate in the kernel, and each thread performs a specific task, such as managing devices or interrupt handling.

2.1.2 Multithreading Models

Single & Multithreaded Processes



Benefits

- Responsiveness
- Resource Sharing
- Economy
- Utilization of MP Architectures

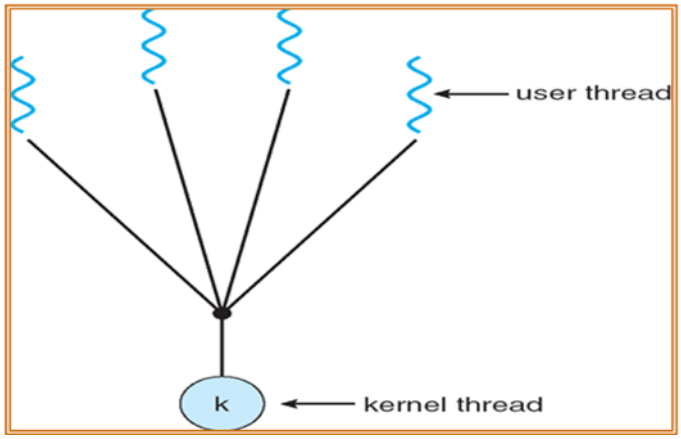
User Threads

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

Multithreading Models

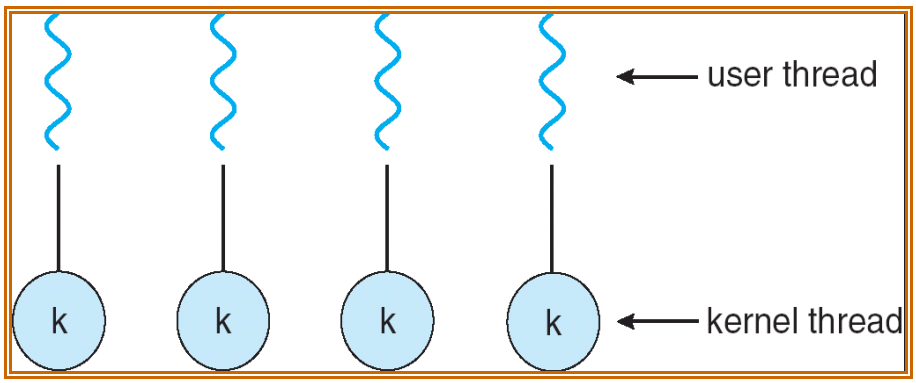
- Many-to-One
- One-to-One
- Many-to-Many

Many-to-One



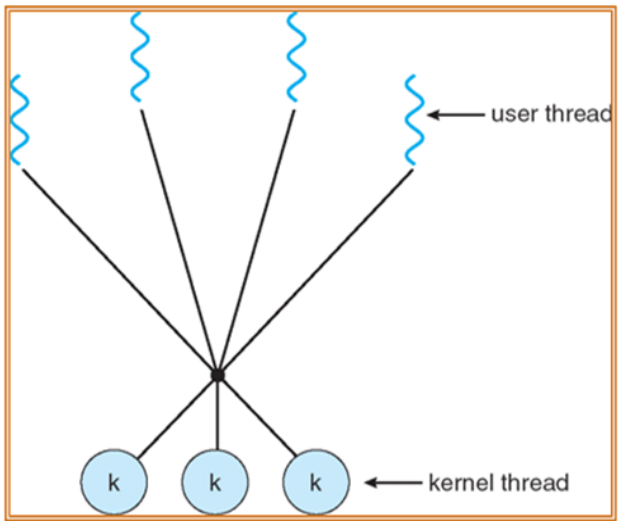
Many-to-One Model

One-to-One



Many-to-Many Model

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



Many-to-Many Model

2.1.6 Threading Issues

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

Thread Cancellation

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

Windows XP Threads

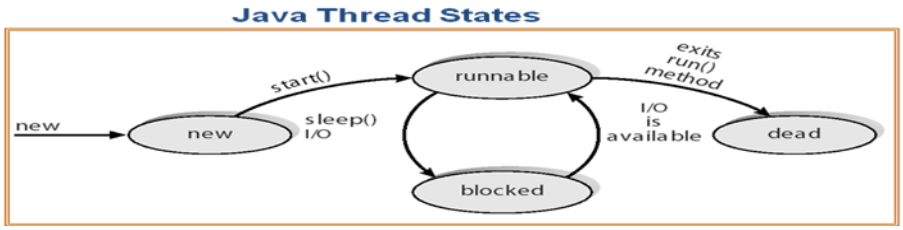
- Implements the one-to-one mapping
- Each thread contains
    - A thread id
    - Register set
    - Separate user and kernel stacks
    - Private data storage area
- The register set, stacks, and private storage area are known as the **context** of the threads
- The primary data structures of a thread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

Linux Threads

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

Java Threads

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



2.2 Process Synchronization

- Concurrent access to shared data may result in data inconsistency (change in behavior)
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Suppose that we wanted to provide a solution to the "producer-consumer" problem that fills all the buffers.
- We can do so by having an integer variable "count" that keeps track of the number of full buffers.
- Initially, count is set to 0.
- It is incremented by the producer after it produces a new buffer.
- It is decremented by the consumer after it consumes a buffer.

Producer

```
while (true) {

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

Consumer

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

2.2.1 Critical section problem:- A section of code which reads or writes shared data.

Race Condition

- The situation where two or more processes try to access and manipulate the same data and output of the process depends on the orderly execution of those processes is called as Race Condition.
- count++ could be implemented as

  register1 = count

  register1 = register1 + 1

  count = register1
- count-- could be implemented as

  register2 = count

  register2 = register2 - 1

  count = register2
- Consider this execution interleaving with "count = 5" initially:
  - S0: producer execute register1 = count {register1 = 5}
  - S1: producer execute register1 = register1 + 1 {register1 = 6}
  - S2: consumer execute register2 = count {register2 = 5}
  - S3: consumer execute register2 = register2 - 1 {register2 = 4}
  - S4: producer execute count = register1 {count = 6 }
  - S5: consumer execute count = register2 {count = 4}

Requirements for the Solution to Critical-Section Problem

1. **Mutual Exclusion**: - If process Pi is executing in its critical section, then no other processes can be executing in their critical sections

2. **Progress:** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.
3. **Bounded Waiting:** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

- To general approaches are used to handle critical sections in operating systems: (1) Preemptive Kernel (2) Non Preemptive Kernel
  - Preemptive Kernel allows a process to be preempted while it is running in kernel mode.
  - Non Preemptive Kernel does not allow a process running in kernel mode to be preempted. (these are free from race conditions)

2.2.2 Peterson's Solution

- It is restricted to two processes that alternates the execution between their critical and remainder sections.
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.
- The two processes share two variables:
  - int turn;
  - Boolean flag[2]
- The variable turn indicates whose turn it is to enter the critical section.
- The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process Pi is ready!

Note:- Peterson's Solution is a software based solution.

Algorithm for Process P_i

```
while (true) {
    flag[i] = TRUE;
    turn = j;
    while ( flag[j] && turn == j);
        CRITICAL SECTION
    flag[i] = FALSE;
        REMAINDER SECTION
}
```

Fig: structure of process P_i in Peterson's solution

Solution to Critical Section Problem using Locks.

```
do{
        acquire lock

            Critical Section
        release lock

            Remainder Section

    }while (True);
```

Note:- Race Conditions are prevented by protecting the critical region by the locks.

2.2.3 Synchronization Hardware

- In general we can provide any solution to critical section problem by using a simple tool called as LOCK where we can prevent the race condition.
- Many systems provide hardware support (hardware instructions available on several systems) for critical section code.
- In UniProcessor hardware environment by disabling interrupts we can solve the critical section problem. So that Currently running code would execute without any preemption .
- But by disabling interrupts on multiprocessor systems is time taking so that it is inefficient compared to UniProcessor system.
- Now a days Modern machines provide special atomic hardware instructions that allow us to either *test memory word and set value* Or *swap contents of two memory words automatically* i.e. done through an uninterruptible unit.

Special Atomic hardware Instructions

- TestAndSet()
- Swap()
- The TestAndSet() Instruction is one kind of special atomic hardware instruction that allow us to test memory and set the value. We can provide Mutual Exclusion by using TestAndSet() instruction.
- Definition:

```
Boolean TestAndSet (Boolean *target)
    {
```

```
    Boolean rv = *target;
    *target = TRUE;
    return rv:
}
```

- Mutual Exclusion Implementation with TestAndSet()
- To implement Mutual Exclusion using TestAndSet() we need to declare Shared Boolean variable called as 'lock' (initialized to false) .
- Solution:

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

- The Swap() Instruction is another kind of special atomic hardware instruction that allow us to swap the contents of two memory words.
- By using Swap() Instruction we can provide Mutual Exclusion.
- Definition:-

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

- Shared Boolean variable called as 'lock' is to be declared to implement Mutual Exclusion in Swap() also, which is initialized to FALSE.

Solution:

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

                //    critical section
```

```
            lock = FALSE;
                    //      remainder section
        }
```

Note:- Each process has a local Boolean variable called as 'key'.

2.2.5 Semaphores

- As it is difficult for the application programmer to use these hardware instructions, to overcome this difficulty we use the synchronization tool called as Semaphore (that does not require busy waiting)
- Semaphore *S* – integer variable, apart from this initialization we can access this only through two standard atomic operations called as wait() and signal().
- Originally the wait() and signal() operations are termed as P() and V() respectively. Which are termed from the Dutch words "proberen" and "verhogen".
- *The definition for wait() is as follows*:

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

- *The definition for signal() is as follows*:

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

- All the modifications to the integer value of the semaphore in the wait() and signal() atomic operations must be executed indivisibly. i.e. when one process changes the semaphore value, no other process will change the same semaphore value simultaneously.
- *Usage of semaphore*:- we have two types of semaphores
  - Counting semaphore
  - Binary Semaphore.
- The value of the Counting Semaphore can ranges over an unrestricted domain.
- The value of the Binary Semaphore can ranges between 0 and 1 only.

- In some systems the Binary Semaphore is called as Mutex locks, because, as they are locks to provide the mutual exclusion.
- We can use the Binary Semaphore to deal with critical section problem for multiple processes.
- Counting Semaphores are used to control the access of given resource each of which consists of some finite no. of instances. This counting semaphore is initialized to number of resources available.
- The process that wish to use a resource must performs the wait() operation (count is decremented)
- The process that releases a resource must performs the signal() operation (count is incremented)
- When the count for the semaphore is 0 means that all the resources are being used by some processes. Otherwise resources are available for the processes to allocate .
- When a process is currently using a resource means that it blocks the resource until the count becomes > 0.
- For example:
  - Let us assume that there are two processes p_0 and p_1 which consists of two statements s_0 & s_1 respectively.
  - Also assume that these two processes are running concurrently such that process p_1 executes the statement s_1 only after process p_0 executes the statement s_0.
  - Let us assume the process p_0 & p_1 share the same semaphore called as "synch" which is initialized to 0 by inserting the statements

    S_0;

    Signal (synch);

  in process p_0 and the statements

    wait (synch);

    S_1;

  in process p_1 .

Implementation:

- The main disadvantage of the semaphore definition is, it requires the busy waiting.
- Because when one process is in critical section and if another process needs to enter in to the critical section must have to loop in the entry code continuously.

Implementation of semaphore with no busy waiting:

- To overcome the need of the busy waiting we have to modify the definition of wait() and signal() operations. i.e. when a process executes wait() operation and finds that it is not positive then it must wait.
- Instead of engaging the busy wait, the process block itself so that there will be a chance to the CPU to select another process for execution. It is done by block() operation.
  - Blocked processes are placed in waiting queue.
- Later the process that has already been blocked by itself is restarted by using wakeup() operation, so that the process will move from waiting state to ready state.
  - Blocked processes that are placed in waiting queue are now placed into ready queue.
- To implement the semaphore with no busy waiting we need to define the semaphore of the wait() and signal() operation by using the 'C' Struct. Which is as follows:

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

  - i.e. each semaphore has an integer value stored in the variable "value" and the list of processes list.
  - When a process perform the wait() operation on the semaphore then it will adds list of processes to the list .
  - When a process perform the signal() operation on the semaphore then it removes the processes from the list.

Semaphore Implementation with no Busy waiting

Implementation of wait: (definition of wait with no busy waiting)

```
wait (S){
        value--;
        if (value < 0) {
            add this process to waiting queue
             block();  }
 }
```

Implementation of signal: (definition of signal with no busy waiting)

```
Signal (S){
          value++;
          if (value <= 0) {
              remove a process P from the waiting queue
```

wakeup(P); }

}

Deadlock and Starvation

- The implementation of semaphore with waiting queue may result in the situation where two or more processes are waiting for an event is called as Deadlocked.
- To illustrate this, let us assume two processes P_0 and P_1 each accessing two semaphores S and Q which are initialized to 1 :-

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

- Now process P_0 executes *wait(S)* and P_1 executes *wait(Q)*, assume that P_0 wants to execute *wait(Q)* and P_1 executes *wait(S)*. But it is possible only after process P_1 executes the *signal(Q)* and P_0 executes *signal(S)*.
- Starvation or indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

2.2.6 Classical Problems of Synchronization

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

Bounded-Buffer Problem

- Let us assume *N* buffers, each can hold only one item.
- Semaphore mutex initialized to the value 1 which is used to provide mutual exclusion.
- Semaphore full initialized to the value 0
- Semaphore empty initialized to the value N.
- Semaphore full and empty are used to count the number of buffers.
- The structure of the producer process

```
while (true) {
```

```
            //   produce an item
        wait (empty);
        wait (mutex);
            //  add the item to the  buffer
        signal (mutex);
        signal (full);
    }
```

The structure of the consumer process

```
    while (true) {
        wait (full);
        wait (mutex);
            //  remove an item from  buffer
        signal (mutex);
        signal (empty);

            //  consume the removed item
    }
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set, do not perform any updates
  - Writers – can both read and write the data set (perform the updates).
- If two readers read the shared data simultaneously, there will be no problem. If both a reader(s) and writer share the same data simultaneously then there will be a problem.
- In the solution of reader-writer problem, the reader process share the following data structures:

```
Semaphore Mutex, wrt;
int readcount;
```

- Where → Semaphore mutex is initialized to 1.

→ Semaphore wrt is initialized to 1.

→ Integer readcount is initialized to 0.

The structure of a writer process

```
    while (true) {
        wait (wrt) ;
```

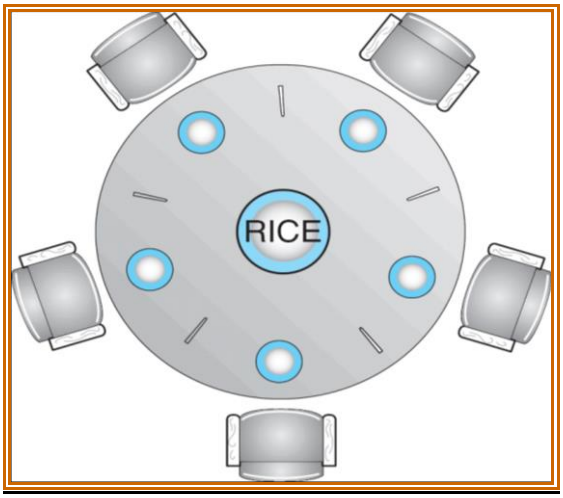
```
            //    writing is performed
        signal (wrt) ;
    }
```

The structure of a reader process

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

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

Dining-Philosophers Problem



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

Dining-Philosophers Problem

- **The structure of Philosopher *i*:**

```
While (true)  {
     wait ( chopstick[i] );
        wait ( chopStick[ (i + 1) % 5] );

              //  eat
        signal ( chopstick[i] );
        signal (chopstick[ (i + 1) % 5] );

          //  think
}
```

Problems with Semaphores

- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex) → Case 1
  - wait (mutex) … wait (mutex) → Case 2
  - Omitting of wait (mutex) or signal (mutex) (or both) → Case 3
- As the semaphores used incorrectly as above may results the timing errors.
- Case 1 → Several processes may execute in critical section by violating the mutual exclusion requirement.
- Case 2 → Dead lock will occur.
- Case 3 → either mutual exclusion is violated or dead lock will occur
- To deal with such type of errors, researchers have developed high-level language constructs.
- One type of high-level language constructs that is to be used to deal with the above type of errors is → the *Monitor* type.

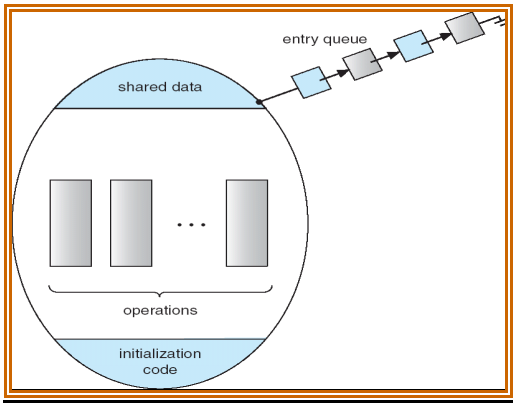
2.2.7 Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization.
- A procedure can access only those variables that are declared in a monitor and formal parameters .
- Only one process may be active within the monitor at a time

Syntax of the monitor :-

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

Schematic view of a Monitor

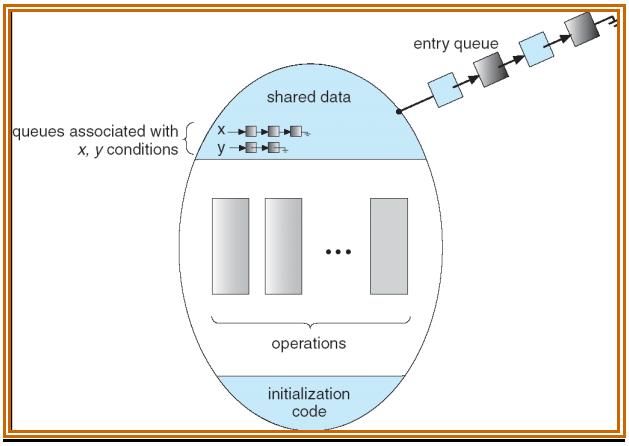


Condition Variables

- Synchronization scheme is not effective with in the monitors.
- A programmer who needs to write 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 operations are
  - x.wait () – a process that request an operation is
  - suspended until another process invokes x.signal ()
  - x.signal () – resumes only one suspended processes (if any) that invoked x.wait ()
- Now suppose that when x.signal() operation is invoked by a process P, there is a suspended process Q associated with condition x. if Q is allowed to resume its execution, the signaling process P must wait. Else both P and Q would be active simultaneously with in a monitor.

- There are two possibilities
  - 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 using Monitors

```
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:

```
dp.pickup (i)
    EAT
dp.putdown (i)
```

Monitor Implementation Using Semaphores Variables

```
semaphore mutex;  // (initially  = 1)
semaphore next;     // (initially  = 0)
int next-count = 0;
```

Each procedure *F* will be replaced by

```
wait(mutex);
    …
                    body of F;
    …
if (next-count > 0)
        signal(next)
else
        signal(mutex);
```

Mutual exclusion within a monitor is ensured.

Monitor Implementation

For each condition variable *x*, we have:

```
semaphore x-sem; // (initially  = 0)
int x-count = 0;
```

The operation x.wait can be implemented as:

```
x-count++;
```

```
if (next-count > 0)
        signal(next);
else
        signal(mutex);
wait(x-sem);
x-count--;
```

The operation x.signal can be implemented as:

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

2.2.8 Synchronization Examples

- **Windows XP**
- **Linux**

Windows XP Synchronization

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

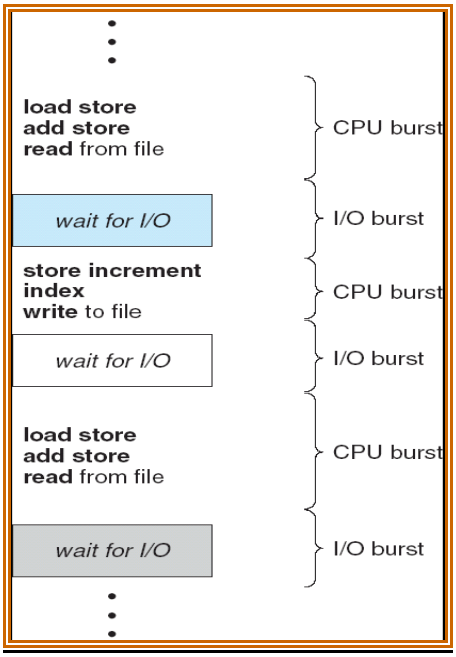
Linux Synchronization

- Linux:
  - disables interrupts to implement short critical sections
- Linux provides:
  - semaphores
  - spin locks

2.3 CPU Scheduling

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a *cycle* of CPU execution and I/O wait
- CPU burst distribution

Alternating Sequence of CPU & I/O Bursts



CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
- CPU scheduling decisions may take place when a process:

1. Switches from running to waiting state
2. Switches from running to ready state
3. Switches from waiting to ready
4. Terminates

- Scheduling under 1 and 4 is *nonpreemptive*
- All other scheduling is *preemptive*

Dispatcher

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

2.3.1 Scheduling Criteria

- CPU utilization – keep the CPU as busy as possible
- Throughput – No. of processes that complete their execution per time unit
- Turnaround time – amount of time to execute a particular process
- Waiting time – amount of time a process has been waiting in the ready queue
- Response time – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

Optimization Criteria

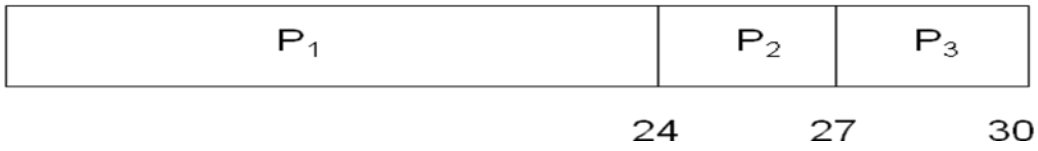
- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

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

The Gantt Chart for the schedule is:



- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

- Suppose that the processes arrive in the order

  P_2 , P_3 , P_1

- The Gantt chart for the schedule is:

- Waiting time for P_1 = 6; P_2 = 0; P_3 = 3

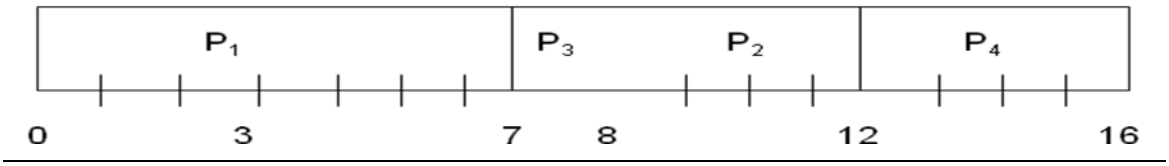
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case
- *Convoy effect* short process behind long process

Shortest-Job-First (SJF) Scheduling

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

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

SJF (non-preemptive)

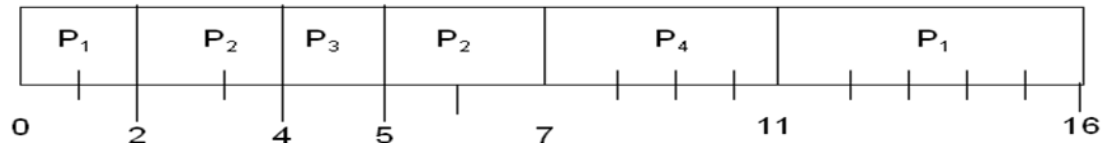


- Average waiting time = (0 + 6 + 3 + 7)/4 = 4

Example of Preemptive SJF

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

- **SJF (preemptive)**



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

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
    - Preemptive
    - Non preemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem ≡ Starvation – low priority processes may never execute
- Solution ≡ Aging - as time progresses increase the priority of the process (means Aging increases the priority of the processes so that to terminate in finite amount of time).

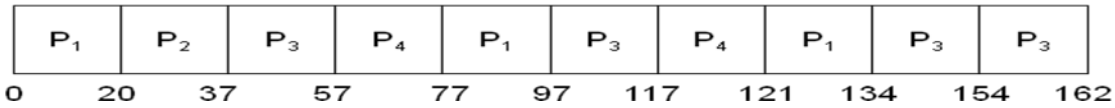
Round Robin (RR)

- Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are n processes in the ready queue and the time quantum is q, then each process gets $1/n$ of the CPU time in chunks of at most q time units at once. No process waits more than $(n-1)q$ time units.
- Performance
    - q large $\Rightarrow$ FIFO
    - q small $\Rightarrow$ q must be large with respect to context switch, otherwise overhead is too high

Example of RR with Time Quantum = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

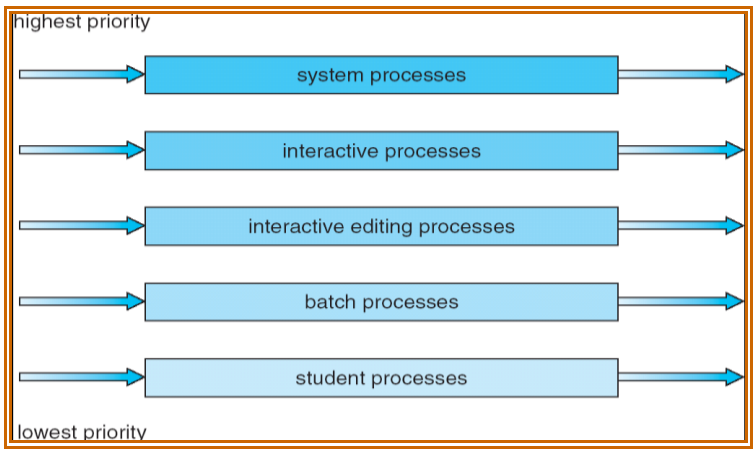
The Gantt chart is:



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

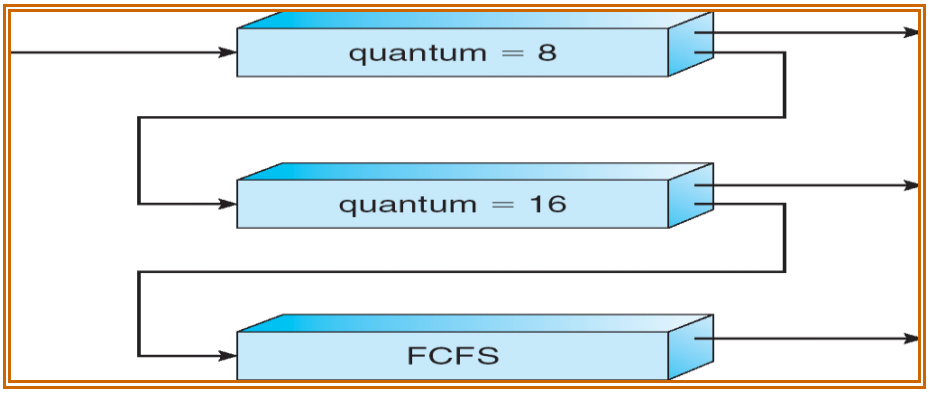
Multilevel Queue Scheduling

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



Multilevel Feedback Queue Scheduling

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



2.3.2 Scheduling Algorithms

2.3.3 Thread Scheduling

- Local Scheduling – How the threads library decides which thread to put onto an available LWP
- Global Scheduling – How the kernel decides which kernel thread to run next

2.3.4 Multiple-Processor Scheduling

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

Operating System Examples

Windows XP Priorities

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

Linux Scheduling

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

Java Thread Scheduling

- JVM Uses a Preemptive, Priority-Based Scheduling Algorithm
- FIFO Queue is Used if There Are Multiple Threads With the Same Priority

JVM Schedules a Thread to Run When:

1. The Currently Running Thread Exits the Runnable State
2. A Higher Priority Thread Enters the Runnable State

* Note – the JVM Does Not Specify Whether Threads are Time-Sliced or Not

CHAPTER 3

3.1 MEMORY MANAGEMENT

Memory management is concerned with managing the primary memory. Memory consists of array of bytes or words each with their own address. The instructions are fetched from the memory by the CPU based on the value program counter.

Functions of memory management:

- Keeping track of status of each memory location.
- Determining the allocation policy.
- Memory allocation technique.
- De-allocation technique.

Address Binding:

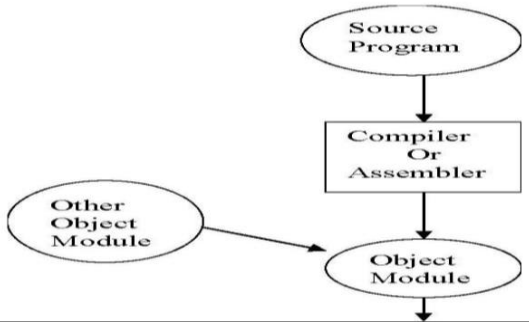
Programs are stored on the secondary storage disks as binary executable files. When the programs are to be executed they are brought in to the main memory and placed within a process. The collection of processes on the disk waiting to enter the main memory forms the input queue. One of the processes which are to be executed is fetched from the queue and placed in the main memory. During the execution it fetches instruction and data from main memory. After the process terminates it returns back the memory space. During execution the process will go through different steps and in each step the address is represented in different ways. In source program the address is symbolic. The compiler converts the symbolic address to re-locatable address. The loader will convert this re-locatable address to absolute address.

Binding of instructions and data can be done at any step along the way:

Compile time:-If we know whether the process resides in memory then absolute code can be generated. If the static address changes then it is necessary to re-compile the code from the beginning.

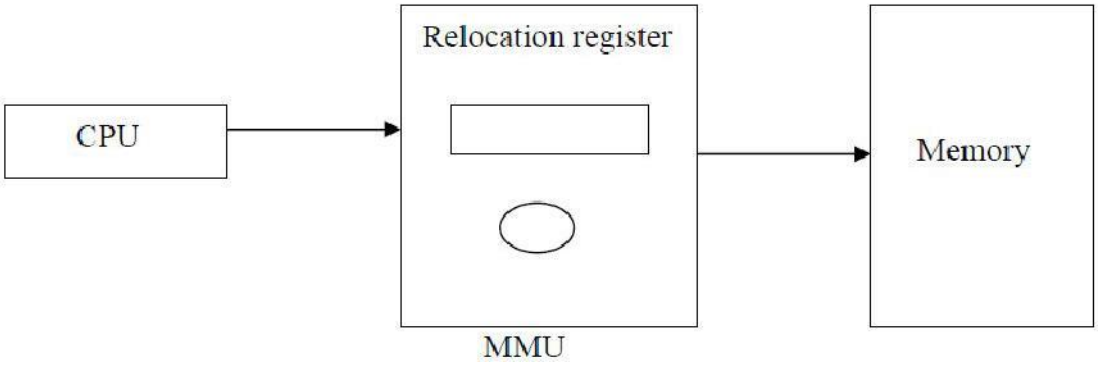
Load time:-If the compiler doesn't know whether the process resides in memory then it generates the re-locatable code. In this the binding is delayed until the load time.

Execution time:-If the process is moved during its execution from one memory segment to another then the binding is delayed until run time. Special hardware is used for this. Most of the general purpose operating system uses this method.



Logical versus physical address:

The address generated by the CPU is called logical address or virtual address. The address seen by the memory unit i.e., the one loaded in to the memory register is called the physical address. Compile time and load time address binding methods generate some logical and physical address. The execution time addressing binding generate different logical and physical address. Set of logical address space generated by the programs is the logical address space. Set of physical address corresponding to these logical addresses is the physical address space. The mapping of virtual address to physical address during run time is done by the hardware device called memory management unit (MMU). The base register is also called re-location register. Value of the re-location register is added to every address generated by the user process at the time it is sent to memory.



Dynamic re-location using a re-location registers

The above figure shows that dynamic re-location which implies mapping from virtual addresses space to physical address space and is performed by the hardware at run time. Re-location is performed by the hardware and is invisible

to the user dynamic relocation makes it possible to move a partially executed process from one area of memory to another without affecting.

Dynamic Loading:

For a process to be executed it should be loaded in to the physical memory. The size of the process is limited to the size of the physical memory. Dynamic loading is used to obtain better memory utilization. In dynamic loading the routine or procedure will not be loaded until it is called. Whenever a routine is called, the calling routine first checks whether the called routine is already loaded or not. If it is not loaded it cause the loader to load the desired program in to the memory and updates the programs address table to indicate the change and control is passed to newly called routine.

Advantage: Gives better memory utilization. Unused routine is never loaded. Do not need special operating system support. This method is useful when large amount of codes are needed to handle in frequently occurring cases.

Dynamic linking and Shared libraries:

Some operating system supports only the static linking. In dynamic linking only the main program is loaded in to the memory. If the main program requests a procedure, the procedure is loaded and the link is established at the time of references. This linking is postponed until the execution time. With dynamic linking a "stub" is used in the image of each library referenced routine. A "stub" is a piece of code which is used to indicate how to locate the appropriate memory resident library routine or how to load library if the routine is not already present. When "stub" is executed it checks whether the routine is present is memory or not. If not it loads the routine in to the memory. This feature can be used to update libraries i.e., library is replaced by a new version and all the programs can make use of this library. More than one version of the library can be loaded in memory at a time and each program uses its version of the library. Only the programs that are compiled with the new version are affected by the changes incorporated in it. Other programs linked before new version is installed will continue using older libraries this type of system is called "shared library".

3.1.1 Swapping

Swapping is a technique of temporarily removing inactive programs from the memory of the system. A process can be swapped temporarily out of the

memory to a backing store and then brought back in to the memory for continuing the execution. This process is called swapping.

Eg:-In a multi-programming environment with a round robin CPU scheduling whenever the time quantum expires then the process that has just finished is swapped out and a new process swaps in to the memory for execution.

A variation of swap is priority based scheduling. When a low priority is executing and if a high priority process arrives then a low priority will be swapped out and high priority is allowed for execution. This process is also called as Roll out and Roll in.

Normally the process which is swapped out will be swapped back to the same memory space that is occupied previously. This depends upon address binding.

If the binding is done at load time, then the process is moved to same memory location. If the binding is done at run time, then the process is moved to different memory location. This is because the physical address is computed during run time.

Swapping requires backing store and it should be large enough to accommodate the copies of all memory images. The system maintains a ready queue consisting of all the processes whose memory images are on the backing store or in memory that are ready to run. Swapping is constant by other factors: To swap a process, it should be completely idle. A process may be waiting for an i/o operation. If the i/o is asynchronously accessing the user memory for i/o buffers, then the process cannot be swapped.

3.1.2 Contiguous memory allocation:

One of the simplest method for memory allocation is to divide memory in to several fixed partition. Each partition contains exactly one process. The degree of multi-programming depends on the number of partitions. In multiple partition method, when a partition is free, process is selected from the input queue and is loaded in to free partition of memory. When process terminates, the memory partition becomes available for another process. Batch OS uses the fixed size partition scheme.

The OS keeps a table indicating which part of the memory is free and is occupied. When the process enters the system it will be loaded in to the

input queue. The OS keeps track of the memory requirement of each process and the amount of memory available and determines which process to allocate the memory. When a process requests, the OS searches for large hole for this process, hole is a large block of free memory available. If the hole is too large it is split in to two. One part is allocated to the requesting process and other is returned to the set of holes. The set of holes are searched to determine which hole is best to allocate. There are three strategies to select a free hole:

o First bit:-Allocates first hole that is big enough. This algorithm scans memory from the beginning and selects the first available block that is large enough to hold the process.

o Best bit:-It chooses the hole i.e., closest in size to the request. It allocates the smallest hole i.e., big enough to hold the process.

o Worst fit:-It allocates the largest hole to the process request. It searches for the largest hole in the entire list.

Operating Systems

First fit and best fit are the most popular algorithms for dynamic memory allocation. First fit is generally faster. Best fit searches for the entire list to find the smallest hole i.e., large enough. Worst fit reduces the rate of production of smallest holes.

All these algorithms suffer from fragmentation.

Memory Protection:

Memory protection means protecting the OS from user process and protecting process from one another. Memory protection is provided by using a re-location register, with a limit register. Re-location register contains the values of smallest physical address and limit register contains range of logical addresses. (Re-location = 100040 and limit = 74600). The logical address must be less than the limit register; the MMU maps the logical address dynamically by adding the value in re-location register. When the CPU scheduler selects a process for execution, the dispatcher loads the re-location and limit register with correct values as a part of context switch. Since every address generated by the CPU is checked against these register we can protect the OS and other users programs and data from being modified.

Fragmentation:

Memory fragmentation can be of two types: Internal Fragmentation External Fragmentation

Internal Fragmentation there is wasted space internal to a portion due to the fact that block of data loaded is smaller than the partition. *Eg*:-If there is a block of 50kb and if the process requests 40kb and if the block is allocated to the process then there will be 10kb of memory left.

External Fragmentation exists when there is enough memory space exists to satisfy the request, but it not contiguous i.e., storage is fragmented in to large number of small holes.

External Fragmentation may be either minor or a major problem.

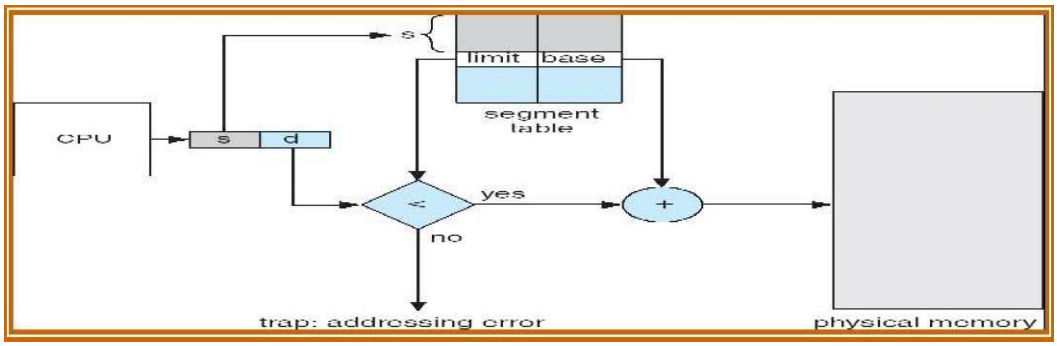
One solution for over-coming external fragmentation is compaction. The goal is to move all the free memory together to form a large block. Compaction is not possible always. If the relocation is static and is done at load time then compaction is not possible. Compaction is possible if the re-location is dynamic and done at execution time.

Another possible solution to the external fragmentation problem is to permit the logical address space of a process to be non-contiguous, thus allowing the process to be allocated physical memory whenever the latter is available.

3.1.3 Segmentation

Most users do not think memory as a linear array of bytes rather the users thinks memory as a collection of variable sized segments which are dedicated to a particular use such as code, data, stack, heap etc. A logical address is a collection of segments. Each segment has a name and length. The address specifies both the segment name and the offset within the segments. The users specify address by using two quantities: a segment name and an offset. For simplicity the segments are numbered and referred by a segment number. So the logical address consists of <segment number, offset>.

Hardware support: We must define an implementation to map 2D user defined address in to 1D physical address. This mapping is affected by a segment table. Each entry in the segment table has a segment base and segment limit. The segment base contains the starting physical address where the segment resides and limit specifies the length of the segment.



The use of segment table is shown in the above figure: Logical address consists of two parts: segment number‘s’ and an offset‘d’ to that segment. The segment number is used as an index to segment table. The offset‘d’ must bi in between 0 and limit, if not an error is reported to OS. If legal the offset is added to the base to generate the actual physical address. The segment table is an array of base limit register pairs.

Protection and Sharing: A particular advantage of segmentation is the association of protection with the segments. The memory mapping hardware will check the protection bits associated with each segment table entry to prevent illegal access to memory like attempts to write in to read-only segment. Another advantage of segmentation involves the sharing of code or data. Each process has a segment table associated with it. Segments are shared when the entries in the segment tables of two different processes points to same physical location. Sharing occurs at the segment table. Any information can be shared at the segment level. Several segments can be shared so a program consisting of several segments can be shared. We can also share parts of a program.

Advantages: Eliminates fragmentation. x Provides virtual growth. Allows dynamic segment growth. Assist dynamic linking. Segmentation is visible.

Differences between segmentation and paging:-

Segmentation:

- Program is divided in to variable sized segments. x User is responsible for dividing the program in to segments.
- Segmentation is slower than paging.
- Visible to user.
- Eliminates internal fragmentation.
- Suffers from external fragmentation.
- Process or user segment number, offset to calculate absolute address.

Paging:

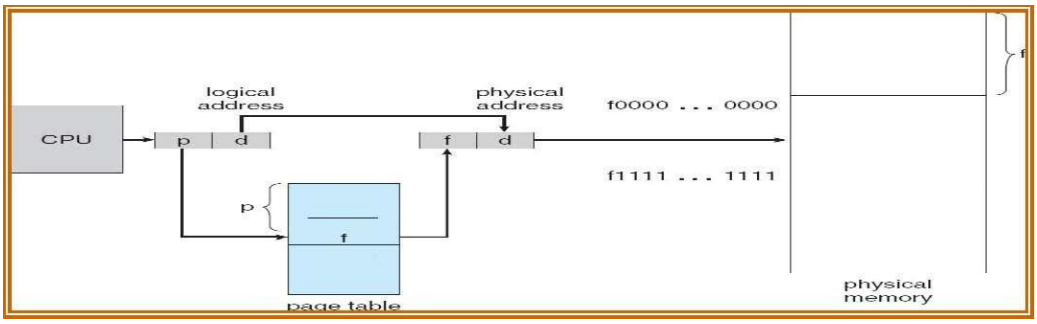
- Programs are divided in to fixed size pages.
- Division is performed by the OS.
- Paging is faster than segmentation.
- Invisible to user.
- Suffers from internal fragmentation.
- No external fragmentation.
- Process or user page number, offset to calculate absolute address.

3.1.4 Paging

Paging is a memory management scheme that permits the physical address space of a process to be non-contiguous. Support for paging is handled by hardware. It is used to avoid external fragmentation. Paging avoids the considerable problem of fitting the varying sized memory chunks on to the backing store. When some code or date residing in main memory need to be swapped out, space must be found on backing store.

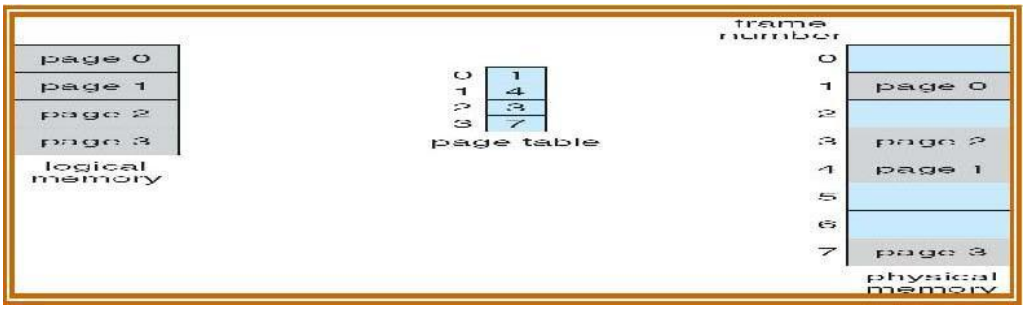
Basic Method:

Physical memory is broken in to fixed sized blocks called frames (f). Logical memory is broken in to blocks of same size called pages (p). When a process is to be executed its pages are loaded in to available frames from backing store. The blocking store is also divided in to fixed-sized blocks of same size as memory frames. The following figure shows paging hardware:



Logical address generated by the CPU is divided in to two parts: page number (p) and page offset (d). The page number (p) is used as index to the page table. The page table contains base address of each page in physical memory. This base address is combined with the page offset to define the physical memory i.e., sent to the memory unit. The page size is defined by the

hardware. The size of a power of 2, varying between 512 bytes and 10Mb per page. If the size of logical address space is 2^m address unit and page size is 2^n, then high order m-n designates the page number and n low order bits represents page offset.



Eg:-To show how to map logical memory in to physical memory consider a page size of 4 bytes and physical memory of 32 bytes (8 pages). Logical address 0 is page 0 and offset 0. Page 0 is in frame 5. The logical address 0 maps to physical address 20. [(5*4) + 0]. ogical address 3 is page 0 and offset 3 maps to physical address 23 [(5*4) + 3]. Logical address 4 is page 1 and offset 0 and page 1 is mapped to frame 6. So logical address 4 maps to physical address 24 [(6*4) + 0]. Logical address 13 is page 3 and offset 1 and page 3 is mapped to frame 2. So logical address 13 maps to physical address 9 [(2*4) + 1].

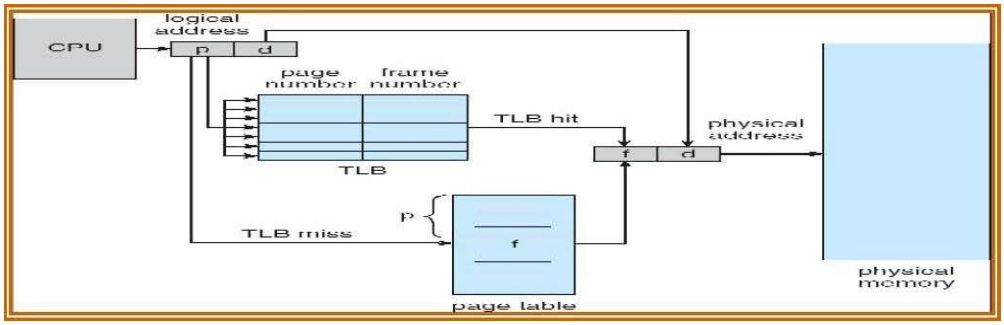
Hardware Support for Paging:

The hardware implementation of the page table can be done in several ways:

The simplest method is that the page table is implemented as a set of dedicated registers. These registers must be built with very high speed logic for making paging address translation. Every accessed memory must go through paging map. The use of registers for page table is satisfactory if the page table is small.

If the page table is large then the use of registers is not visible. So the page table is kept in the main memory and a page table base register [PTBR] points to the page table. Changing the page table requires only one register which reduces the context switching type. The problem with this approach is the time required to access memory location. To access a location [i] first we have to index the page table using PTBR offset. It gives the frame number which is combined with the page offset to produce the actual address. Thus we need two memory accesses for a byte.

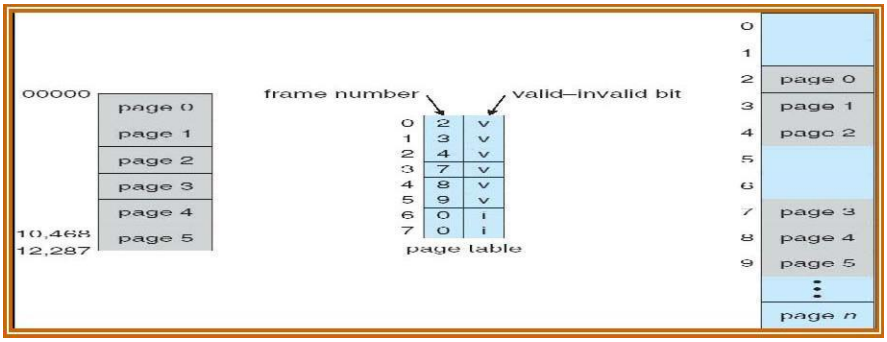
The only solution is to use special, fast, lookup hardware cache called translation look aside buffer [TLB] or associative register. LB is built with associative register with high speed memory. Each register contains two paths a key and a value.



When an associative register is presented with an item, it is compared with all the key values, if found the corresponding value field is return and searching is fast. TLB is used with the page table as follows: TLB contains only few page table entries. When a logical address is generated by the CPU, its page number along with the frame number is added to TLB. If the page number is found its frame memory is used to access the actual memory. If the page number is not in the TLB (TLB miss) the memory reference to the page table is made. When the frame number is obtained use can use it to access the memory. If the TLB is full of entries the OS must select anyone for replacement. Each time a new page table is selected the TLB must be flushed [erased] to ensure that next executing process do not use wrong information. The percentage of time that a page number is found in the TLB is called HIT ratio.

Protection:

Memory protection in paged environment is done by protection bits that are associated with each frame these bits are kept in page table. x One bit can define a page to be read-write or read-only. To find the correct frame number every reference to the memory should go through page table. At the same time physical address is computed. The protection bits can be checked to verify that no writers are made to read-only page. Any attempt to write in to read-only page causes a hardware trap to the OS. This approach can be used to provide protection to read-only, read-write or execute-only pages. One more bit is generally added to each entry in the page table: a valid-invalid bit.



A valid bit indicates that associated page is in the processes logical address space and thus it is a legal or valid page.

If the bit is invalid, it indicates the page is not in the processes logical addressed space and illegal. Illegal addresses are trapped by using the valid-invalid bit.

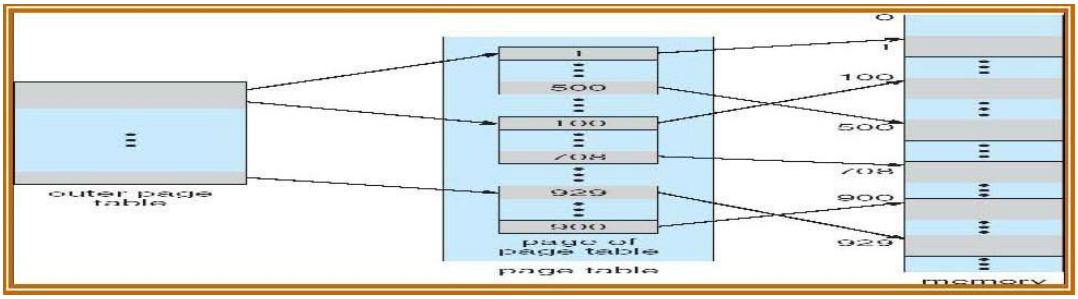
The OS sets this bit for each page to allow or disallow accesses to that page.

3.1.5 Structure Of Page Table

a. Hierarchical paging:

Recent computer system support a large logical address apace from 2^32 to 2^64. In this system the page table becomes large. So it is very difficult to allocate contiguous main memory for page table. One simple solution to this problem is to divide page table in to smaller pieces. There are several ways to accomplish this division.

One way is to use two-level paging algorithm in which the page table itself is also paged. *Eg*:-In a 32 bit machine with page size of 4kb. A logical address is divided in to a page number consisting of 20 bits and a page offset of 12 bit. The page table is further divided since the page table is paged, the page number is further divided in to 10 bit page number and a 10 bit offset.



b. Hashed page table:

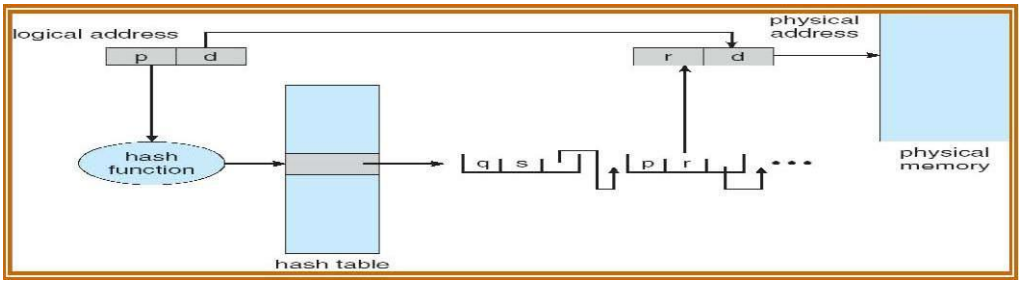
Hashed page table handles the address space larger than 32 bit. The virtual page number is used as hashed value. Linked list is used in the hash table which contains a list of elements that hash to the same location.

Each element in the hash table contains the following three fields: Virtual page number x Mapped page frame value x Pointer to the next element in the linked list

Working:

Virtual page number is taken from virtual address. Virtual page number is hashed in to hash table.

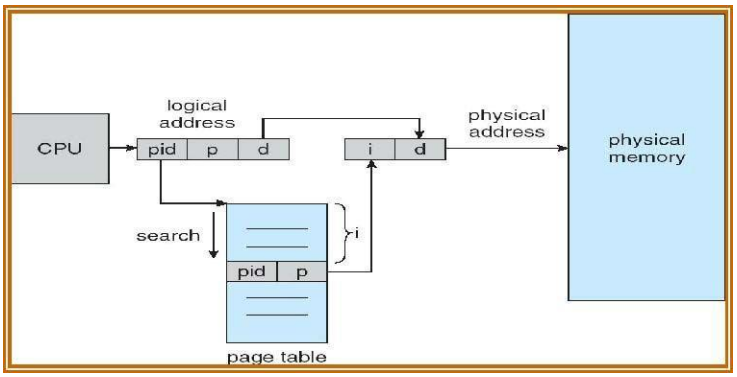
Virtual page number is compared with the first element of linked list. Both the values are matched, that value is (page frame) used for calculating the physical address. If not match then entire linked list is searched for matching virtual page number. Clustered pages are similar to hash table but one difference is that each entity in the hash table refer to several pages.



c. Inverted Page Tables:

Since the address spaces have grown to 64 bits, the traditional page tables become a problem. Even with two level page tables. The table can be too large to handle. An inverted page table has only entry for each page in memory. Each entry consisted of virtual address of the page stored in that read-only location with information about the process that owns that page.

Each virtual address in the Inverted page table consists of triple <process-id , page number , offset >. The inverted page table entry is a pair <process-id , page number>. When a memory reference is made, the part of virtual address i.e., <process-id , page number> is presented in to memory sub-system. The inverted page table is searched for a match. If a match is found at entry I then the physical address <i , offset> is generated. If no match is found then an illegal address access has been attempted. This scheme decreases the amount of memory needed to store each page table, it increases the amount of time needed to search the table when a page reference occurs. If the whole table is to be searched it takes too long.



Advantage:

- Eliminates fragmentation.
- Support high degree of multiprogramming.
- Increases memory and processor utilization.
- Compaction overhead required for the re-locatable partition scheme is also eliminated.

Disadvantage:

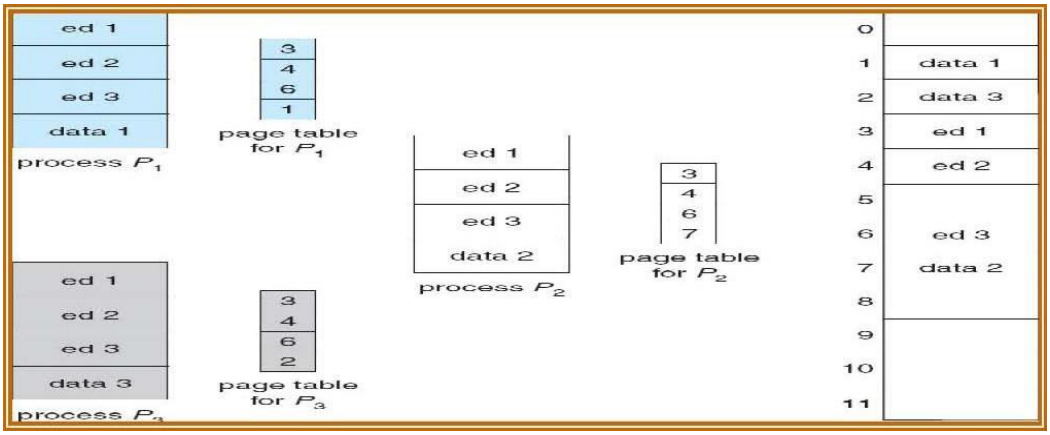
- Page address mapping hardware increases the cost of the computer.
- Memory must be used to store the various tables like page tables, memory map table etc.
- Some memory will still be unused if the number of available block is not sufficient for the address space of the jobs to be run.

d. Shared Pages:

☐ Another advantage of paging is the possibility of sharing common code. This is useful in timesharing environment. *Eg*:-Consider a system with 40 users, each executing a text editor. If the text editor is of 150k and data space is 50k, we need

8000k for 40 users. If the code is reentrant it can be shared. Consider the following figure

If the code is reentrant then it never changes during execution. Thus two or more processes can execute same code at the same time. Each process has its own copy of registers and the data of two processes will vary. Only one copy of the editor is kept in physical memory. Each users page table maps to same physical copy of editor but date pages are mapped to different frames. So to support 40 users we need only one copy of editor (150k) plus 40 copies of 50k of data space i.e., only 2150k instead of 8000k.



3.2 Virtual Memory Management

Virtual memory is a technique that allows for the execution of partially loaded process. There are many advantages of this: A program will not be limited by the amount of physical memory that is available user can able to write in to large virtual space. Since each program takes less amount of physical memory, more than one program could be run at the same time which can increase the throughput and CPU utilization. Less i/o operation is needed to swap or load user program in to memory. So each user program could run faster.

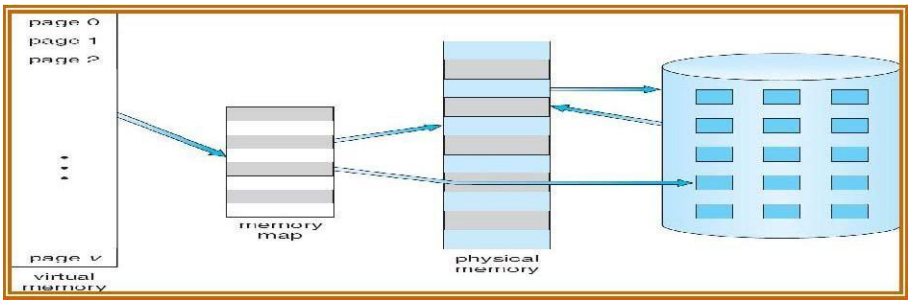
Virtual memory is the separation of users logical memory from physical memory. This separation allows an extremely large virtual memory to be provided when these is less physical memory. Separating logical memory from physical memory also allows files and memory to be shared by several different processes through page sharing.

Virtual memory is implemented using Demand Paging.

3.2.1 Demand Paging

A demand paging is similar to paging system with swapping when we want to execute a process we swap the process the in to memory otherwise it will not be loaded in to memory. A swapper manipulates the entire processes, where as a pager manipulates individual pages of the process.

Basic concept: Instead of swapping the whole process the pager swaps only the necessary pages in to memory. Thus it avoids reading unused pages and decreases the swap time and amount of physical memory needed.



The valid-invalid bit scheme can be used to distinguish between the pages that are on the disk and that are in memory.

o If the bit is valid then the page is both legal and is in memory.

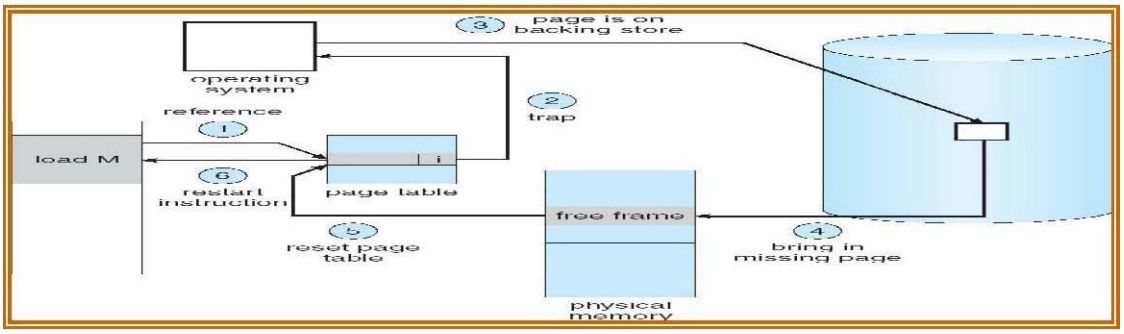
o If the bit is invalid then either page is not valid or is valid but is currently on the disk. Marking a page as invalid will have no effect if the processes never access to that page. Suppose if it access the page which is marked invalid, causes a page fault trap. This may result in failure of OS to bring the desired page in to memory.

The step for handling page fault is straight forward and is given below:

- We check the internal table of the process to determine whether the reference made is valid or invalid.
- If invalid terminate the process,. If valid, then the page is not yet loaded and we now page it in.
- We find a free frame.
- We schedule disk operation to read the desired page in to newly allocated frame.
- When disk reed is complete, we modify the internal table kept with the process to indicate that the page is now in memory.

We restart the instruction which was interrupted by illegal address trap. The process can now access the page. In extreme cases, we start the process without

pages in memory. When the OS points to the instruction of process it generates a page fault. After this page is brought in to memory the process continues to execute, faulting as necessary until every demand paging i.e., it never brings the page in to memory until it is required.



Hardware support:

For demand paging the same hardware is required as paging and swapping.

Page table:-Has the ability to mark an entry invalid through valid-invalid bit.

Secondary memory:-This holds the pages that are not present in main memory.

Performance of demand paging: Demand paging can have significant effect on the performance of the computer system.

Let P be the probability of the page fault (0<=P<=1)

Effective access time = (1-P) * ma + P * page fault. Where P = page fault and ma = memory access time. Effective access time is directly proportional to page fault rate. It is important to keep page fault rate low in demand paging.

A page fault causes the following sequence to occur:

Trap to the OS.

Save the user registers and process state. Determine that the interrupt was a page fault. Checks the page references were legal and determine the location of page on disk. Issue a read from disk to a free frame.

If waiting, allocate the CPU to some other user. Interrupt from the disk. Save the registers and process states of other users. Determine that the interrupt was from the disk. Correct the page table and other table to show that the desired page is now in memory.

Comparison of demand paging with segmentation:-

Segmentation:

- o Segment may of different size.
- o Segment can be shared.
- o Allows for dynamic growth of segments.
- o Segment map table indicate the address of each segment in memory.
- o Segments are allocated to the program while compilation.

Demand Paging:

- o Pages are of same size.
- o Pages can't be shared.
- o Page size is fixed.
- . o Page table keeps track of pages in memory.
- o Pages are allocated in memory on demand.

3.2.2 Copy-On-Write:

Demand paging is used when reading a file from disk in to memory. Fork () is used to create a process and it initially bypass the demand paging using a technique called page sharing. Page sharing provides rapid speed for process creation and reduces the number of pages allocated to the newly created process. Copy-on-write technique initially allows the parent and the child to share the same pages. These pages are marked as copy-on-write pages i.e., if either process writes to a shared page, a copy of shared page is created.

Eg:-If a child process try to modify a page containing portions of the stack; the OS recognizes them as a copy-on-write page and create a copy of this page and maps it on to the address space of the child process. So the child process will modify its copied page and not the page belonging to parent. The new pages are obtained from the pool of free pages.

Memory Mapping: Standard system calls i.e., open (), read () and write () is used for sequential read of a file. Virtual memory is used for this. In memory mapping a

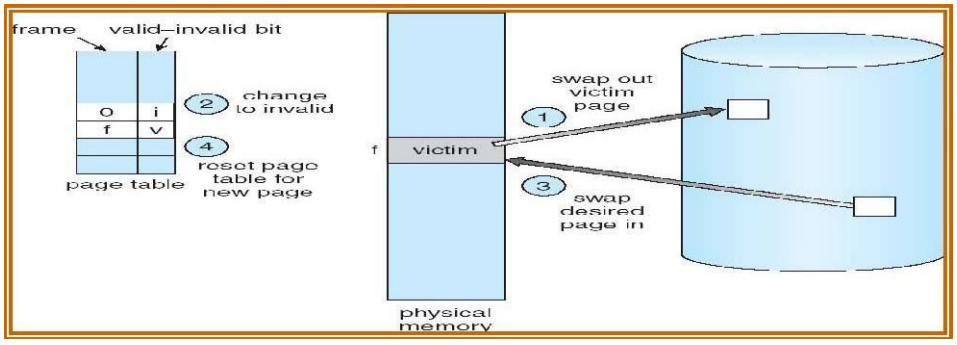
file allows a part of the virtual address space to be logically associated with a file. Memory mapping a file is possible by mapping a disk block to page in memory.

3.2.3 Page Replacement

Demand paging shares the I/O by not loading the pages that are never used.

Demand paging also improves the degree of multiprogramming by allowing more process to run at the sometime. Page replacement policy deals with the solution of pages in memory to be replaced by a new page that must be brought in. When a user process is executing a page fault occurs. The hardware traps to the operating system, which checks the internal table to see that this is a page fault and not an illegal memory access. The operating system determines where the derived page is residing on the disk, and this finds that are no free frames on the list of free frames.

When all the frames are in main memory, it is necessary to bring a new page to satisfy the page fault, replacement policy is concerned with selecting a page currently in memory to be replaced. The page i,e to be removed should be the page i,e least likely to be referenced in future.



Working of Page Replacement Algorithm

1. Find the location of derived page on the disk.

2. Find a free frame x If there is a free frame, use it. x Otherwise, use a replacement algorithm to select a victim. x Write the victim page to the disk; change the page and frame tables accordingly.

3. Read the desired page into the free frame; change the page and frame tables.

Restart the user process.

Victim Page

The page that is supported out of physical memory is called victim page. x If no frames are free, the two page transforms come (out and one in) are read. This will see the effective access time.

- Each page or frame may have a dirty (modify) bit associated with the hardware. The modify bit for a page is set by the hardware whenever any word or byte in the page is written into, indicating that the page has been modified.
- When we select the page for replacement, we check its modify bit. If the bit is set, then the page is modified since it was read from the disk.
- If the bit was not set, the page has not been modified since it was read into memory. Therefore, if the copy of the page has not been modified we can avoid writing the memory page to the disk, if it is already there. Sum pages cannot be modified.

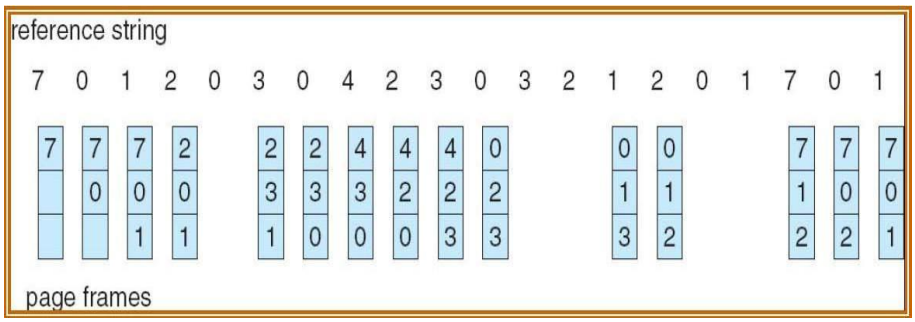
We must solve two major problems to implement demand paging: we must develop a frame allocation algorithm and a page replacement algorithm. If we have multiple processors in memory, we must decide how many frames to allocate and page replacement is needed.

PAGE REPLACEMENT ALGORITHMS:

FIFO Algorithm:

- This is the simplest page replacement algorithm. A FIFO replacement algorithm associates each page the time when that page was brought into memory.
- When a Page is to be replaced the oldest one is selected.
- We replace the queue at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue.

Example: Consider the following references string with frames initially empty.



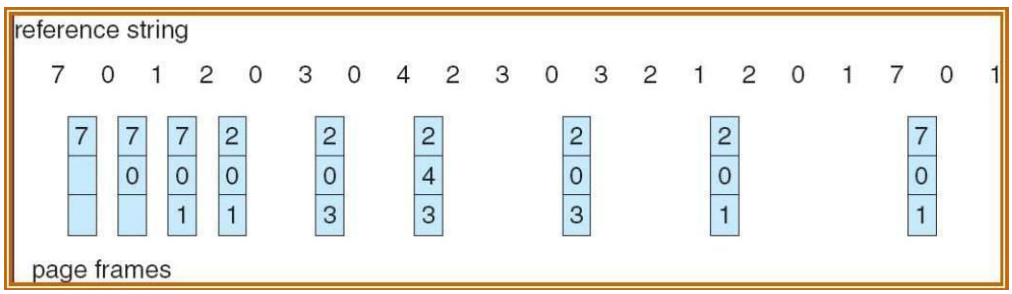
The first three references (7,0,1) cases page faults and are brought into the empty frames. The next references 2 replaces page 7 because the page 7 was brought in first. Since 0 is the next references and 0 is already in memory e has no page faults. The next references 3 results in page 0 being replaced so that the next references to 0 causer page fault. This will continue till the end of string. There are 15 faults all together.

Belady's Anamoly

For some page replacement algorithm, the page fault may increase as the number of allocated frames increases. FIFO replacement algorithm may face this problem.

Optimal Algorithm

Optimal page replacement algorithm is mainly to solve the problem of Belady's Anamoly. Optimal page replacement algorithm has the lowest page fault rate of all algorithms. An optimal page replacement algorithm exists and has been called OPT. The working is simple "Replace the page that will not be used for the longest period of time" Example: consider the following reference string

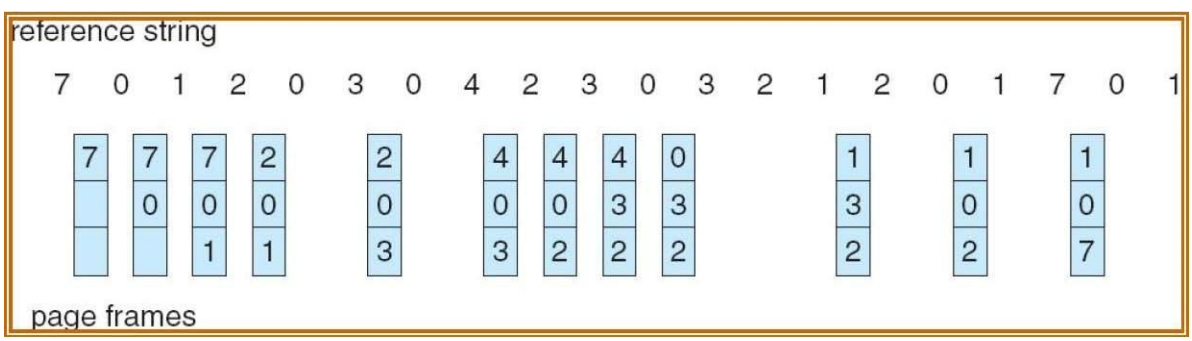


The first three references cause faults that fill the three empty frames. The references to page 2 replaces page 7, because 7 will not be used until reference 18. The page 0 will be used at 5 and page 1 at 14. With only 9 page faults, optimal replacement is much better than a FIFO, which had 15 faults. This algorithm is difficult t implement because it requires future knowledge of reference strings.

Least Recently Used (LRU) Algorithm

If the optimal algorithm is not feasible, an approximation to the optimal algorithm is possible. The main difference b/w OPTS and FIFO is that; FIFO algorithm uses the time when the pages was built in and OPT uses the time when a page is to be used.

The LRU algorithm replaces the pages that have not been used for longest period of time. The LRU associated its pages with the time of that pages last use. x This strategy is the optimal page replacement algorithm looking backward in time rather than forward. Ex: consider the following reference string



The first 5 faults are similar to optimal replacement.

When reference to page 4 occurs, LRU sees that of the three frames, page 2 as used least recently. The most recently used page is page 0 and just before page 3 was used. The LRU policy is often used as a page replacement algorithm and considered to be good. The main problem to how to implement LRU is the LRU requires additional h/w assistance. Two implementation are possible:

Counters: In this we associate each page table entry a time -of -use field, and add to the cpu a logical clock or counter. The clock is incremented for each memory reference. When a reference to a page is made, the contents of the clock register

are copied to the time-of-use field in the page table entry for that page. In this way we have the time of last reference to each page we replace the page with smallest time value. The time must also be maintained when page tables are changed.

Stack: Another approach to implement LRU replacement is to keep a stack of page numbers when a page is referenced it is removed from the stack and put on to the top of stack. In this way the top of stack is always the most recently used page and the bottom in least recently used page. Since the entries are removed from the stack it is best implement by a doubly linked list. With a head and tail pointer. Neither optimal replacement nor LRU replacement suffers from Belady's Anamoly. These are called stack algorithms.

LRU Approximation

An LRU page replacement algorithm should update the page removal status information after every page reference updating is done by software, cost increases.

But hardware LRU mechanism tend to degrade execution performance at the same time, then substantially increases the cost. For this reason, simple and efficient algorithm that approximation the LRU have been developed. With h/w support the reference bit was used. A reference bit associate with each memory block and this bit automatically set to 1 by the h/w whenever the page is referenced. The single reference bit per clock can be used to approximate LRU removal. The page removal s/w periodically resets the reference bit to 0, write the execution of the users job causes some reference bit to be set to 1. If the reference bit is 0 then the page has not been referenced since the last time the reference bit was set to 0.

Count Based Page Replacement

There is many other algorithms that can be used for page replacement, we can keep a counter of the number of references that has made to a page.

a) LFU (least frequently used):

This causes the page with the smallest count to be replaced. The reason for this selection is that actively used page should have a large reference count. This algorithm suffers from the situation in which a page is used heavily during the initial

phase of a process but never used again. Since it was used heavily, it has a large count and remains in memory even though it is no longer needed.

b) Most Frequently Used(MFU):

This is based on the principle that the page with the smallest count was probably just brought in and has yet to be used.

3.2.4 Allocation of Frames

The allocation policy in a virtual memory controls the operating system decision regarding the amount of real memory to be allocated to each active process. In a paging system if more real pages are allocated, it reduces the page fault frequency and improved turnaround throughput. If too few pages are allocated to a process its page fault frequency and turnaround times may deteriorate to unacceptable levels.

The minimum number of frames per process is defined by the architecture, and the maximum number of frames. This scheme is called equal allocation. With multiple processes competing for frames, we can classify page replacement into two broad categories

a) Local Replacement: requires that each process selects frames from only its own sets of allocated frame.

b). Global Replacement: allows a process to select frame from the set of all frames. Even if the frame is currently allocated to some other process, one process can take a frame from another.

In local replacement the number of frames allocated to a process do not change but with global replacement number of frames allocated to a process do not change global replacement results in greater system throughput.

Other consideration

There is much other consideration for the selection of a replacement algorithm and allocation policy.

1) Preparing: This is an attempt to present high level of initial paging. This strategy is to bring into memory all the pages at one time. 2) TLB Reach: The TLB reach refers to the amount of memory accessible from the TLB and is simply the no of entries multiplied by page size.

Page Size: following parameters are considered a) page size us always power of 2 (from 512 to 16k) b) Internal fragmentation is reduced by a small page size. c) A large page size reduces the number of pages needed.

Invented Page table: This will reduces the amount of primary memory i,e. needed to track virtual to physical address translations. 5) Program Structure: Careful selection of data structure can increases the locality and hence lowers the page fault rate and the number of pages in working state.

Real time Processing: Real time system almost never has virtual memory. Virtual memory is the antithesis of real time computing, because it can introduce unexpected long term delay in the execution of a process.

3.2.5 Thrashing

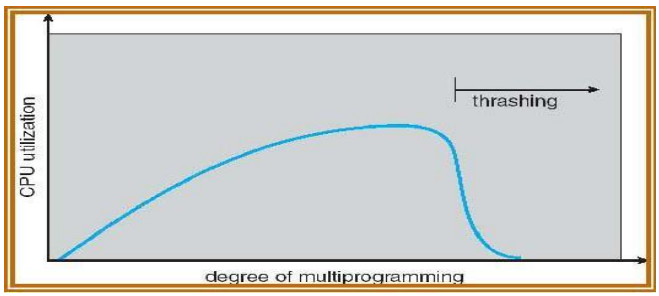
If the number of frames allocated to a low-priority process falls below the minimum number required by the computer architecture then we suspend the process execution. A process is thrashing if it is spending more time in paging than executing.

If the processes do not have enough number of frames, it will quickly page fault. During this it must replace some page that is not currently in use. Consequently it quickly faults again and again. The process continues to fault, replacing pages for which it then faults and brings back. This high paging activity is called thrashing. The phenomenon of excessively moving pages back and forth b/w memory and secondary has been called thrashing.

Cause of Thrashing: Thrashing results in severe performance problem. The operating system monitors the cpu utilization is low. We increase the degree of multi programming by introducing new process to the system. A global page

replacement algorithm replaces pages with no regards to the process to which they belong. The figure shows the thrashing

As the degree of multi programming increases, more slowly until a maximum is reached. If the degree of multi programming is increased further thrashing sets in and the cpu utilization drops sharply.



At this point, to increases CPU utilization and stop thrashing, we must increase degree of multi programming. We can limit the effect of thrashing by using a local replacement algorithm. To prevent thrashing, we must provide a process as many frames as it needs.

Locality of Reference: As the process executes it moves from locality to locality. A locality is a set of pages that are actively used. A program may consist of several different localities, which may overlap. Locality is caused by loops in code that find to reference arrays and other data structures by indices. The ordered list of page number accessed by a program is called reference string. Locality is of two types

1) spatial locality 2) temporal locality

Working set model

Working set model algorithm uses the current memory requirements to determine the number of page frames to allocate to the process, an informal definition is "the collection of pages that a process is working with and which must be resident if the process to avoid thrashing". The idea is to use the recent needs of a process to predict its future reader. The working set is an approximation of programs locality. Ex: given a sequence of memory reference, if the working set window size to memory references, then working set at time t1 is {1,2,5,6,7} and at t2 is changed to {3,4} At any given time, all pages referenced by a process in its last 4 seconds of

execution are considered to compromise its working set. A process will never execute until its working set is resident in main memory. Pages outside the working set can be discarded at any movement. Working sets are not enough and we must also introduce balance set. If the sum of the working sets of all the run able process is greater than the size of memory the refuse some process for a while. Divide the run able process into two groups, active and inactive. The collection of active set is called the balance set. When a process is made active its working set is loaded.

3.3 Deadlocks

When processes request a resource and if the resources are not available at that time the process enters into waiting state. Waiting process may not change its state because the resources they are requested are held by other process. This situation is called deadlock. The situation where the process waiting for the resource i.e., not available is called deadlock.

3.3.1 System Model

A system may consist of finite number of resources and is distributed among number of processes. There resources are partitioned into several instances each with identical instances.

A process must request a resource before using it and it must release the resource after using it. It can request any number of resources to carry out a designated task. The amount of resource requested may not exceed the total number of resources available.

A process may utilize the resources in only the following sequences:

Request:-If the request is not granted immediately then the requesting process must wait it can acquire the resources.

Use:-The process can operate on the resource.

Release:-The process releases the resource after using it.

Deadlock may involve different types of resources. *For eg*:-Consider a system with one printer and one tape drive. If a process Pi currently holds a printer and a process Pj holds the tape drive. If process Pi request a tape drive and process Pj

request a printer then a deadlock occurs. Multithread programs are good candidates for deadlock because they compete for shared resources.

3.3.2 Deadlock Characterization:

Necessary Conditions: A deadlock situation can occur if the following 4 conditions occur simultaneously in a system:-1. Mutual Exclusion: Only one process must hold the resource at a time. If any other process requests for the resource, the requesting process must be delayed until the resource has been released.

Hold and Wait:-A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by the other process.

No Preemption:-Resources can't be preempted i.e., only the process holding the resources must release it after the process has completed its task.

Circular Wait:-A set {P0,P1……..Pn} of waiting process must exist such that P0 is waiting for a resource i.e., held by P1, P1 is waiting for a resource i.e., held by P2. Pn-1 is waiting for resource held by process Pn and Pn is waiting for the resource i.e., held by P1. All the four conditions must hold for a deadlock to occur.

Resource Allocation Graph:

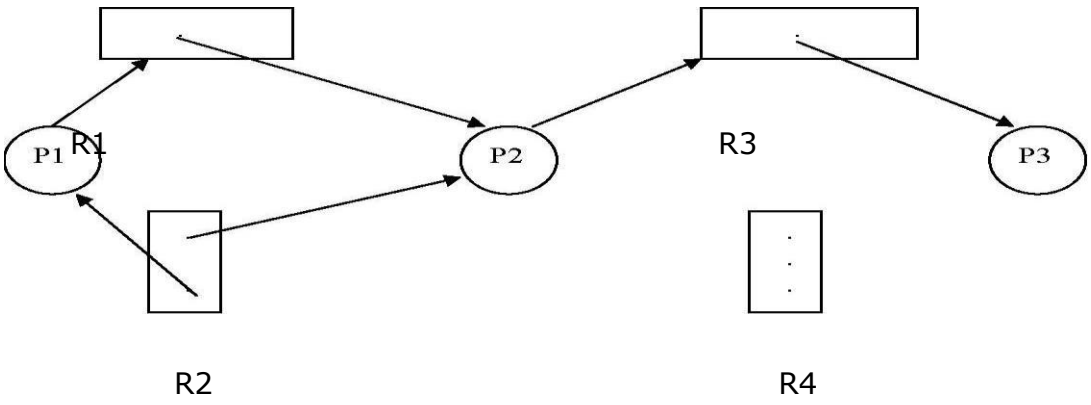
Deadlocks are described by using a directed graph called system resource allocation graph. The graph consists of set of vertices (v) and set of edges (e).

The set of vertices (v) can be described into two different types of nodes P={P1,P2……..Pn} i.e., set consisting of all active processes and R={R1,R2…….Rn}i.e., set consisting of all resource types in the system

A directed edge from process Pi to resource type Rj denoted by Pi->Ri indicates that Pi requested an instance of resource Rj and is waiting. This edge is called Request edge.

8. A directed edge Ri-> Pj signifies that resource Rj is held by process Pi. This is called Assignment edge

Eg:



If the graph contain no cycle, then no process in the system is deadlock. If the graph contains a cycle then a deadlock may exist. If each resource type has exactly one instance than a cycle implies that a deadlock has occurred. If each resource has several instances then a cycle do not necessarily implies that a deadlock has occurred.

3.3.3 Methods for Handling Deadlocks

There are three ways to deal with deadlock problem x We can use a protocol to prevent deadlocks ensuring that the system will never enter into the deadlock state. x We allow a system to enter into deadlock state, detect it and recover from it. x We ignore the problem and pretend that the deadlock never occur in the system. This is used by most OS including UNIX.

To ensure that the deadlock never occur the system can use either deadlock avoidance or a deadlock prevention.

Deadlock prevention is a set of method for ensuring that at least one of the necessary conditions does not occur.

Deadlock avoidance requires the OS is given advance information about which resource a process will request and use during its lifetime.

If a system does not use either deadlock avoidance or deadlock prevention then a deadlock situation may occur. During this it can provide an algorithm that examines the state of the system to determine whether a deadlock has occurred and algorithm to recover from deadlock. Undetected deadlock will result in deterioration of the system performance.

3.3.4 Deadlock Prevention

For a deadlock to occur each of the four necessary conditions must hold. If at least one of the there condition does not hold then we can prevent occurrence of deadlock.

Mutual Exclusion: This holds for non-sharable resources. *Eg*:-A printer can be used by only one process at a time.

Mutual exclusion is not possible in sharable resources and thus they cannot be involved in deadlock. Read-only files are good examples for sharable resources. A process never waits for accessing a sharable resource. So we cannot prevent deadlock by denying the mutual exclusion condition in non-sharable resources.

Hold and Wait: This condition can be eliminated by forcing a process to release all its resources held by it when it request a resource i.e., not available. One protocol can be used is that each process is allocated with all of its resources before its start execution.

Eg:-consider a process that copies the data from a tape drive to the disk, sorts the file and then prints the results to a printer. If all the resources are allocated at the beginning then the tape drive, disk files and printer are assigned to the process. The main problem with this is it leads to low resource utilization because it requires printer at the last and is allocated with it from the beginning so that no other process can use it.

Another protocol that can be used is to allow a process to request a resource when the process has none. i.e., the process is allocated with tape drive and disk file. It performs the required operation and releases both. Then the process once again request for disk file and the printer and the problem and with this is starvation is possible.

No Preemption: To ensure that this condition never occurs the resources must be preempted. The following protocol can be used. If a process is holding some resource and request another resource that cannot be immediately allocated to it, then all the resources currently held by the requesting process are preempted and

added to the list of resources for which other processes may be waiting. The process will be restarted only when it regains the old resources and the new resources that it is requesting.

When a process request resources, we check whether they are available or not. If they are available we allocate them else we check that whether they are allocated to some other waiting process. If so we preempt the resources from the waiting process and allocate them to the requesting process. The requesting process must wait.

Circular Wait:-The fourth and the final condition for deadlock is the circular wait condition. One way to ensure that this condition never, is to impose ordering on all resource types and each process requests resource in an increasing order.

Let R={R1,R2,……Rn} be the set of resource types. We assign each resource type with a unique integer value. This will allows us to compare two resources and determine whether one precedes the other in ordering. *Eg*:-we can define a one to one function

F:R□N as follows :-F(disk drive)=5 F(printer)=12 F(tape drive)=1

Deadlock can be prevented by using the following protocol: Each process can request the resource in increasing order. A process can request any number of instances of resource type say Ri and it can request instances of resource type Rj only F(Rj) > F(Ri).

Alternatively when a process requests an instance of resource type Rj, it has released any resource Ri such that F(Ri) >= F(Rj). If these two protocol are used then the circular wait can't hold.

3.3.5 Deadlock Avoidance

Deadlock prevention algorithm may lead to low device utilization and reduces system throughput. Avoiding deadlocks requires additional information about how resources are to be requested. With the knowledge of the complete sequences of requests and releases we can decide for each requests whether or not the process should wait.

For each requests it requires to check the resources currently available, resources that are currently allocated to each processes future requests and release of each process to decide whether the current requests can be satisfied or must wait to avoid future possible deadlock.

A deadlock avoidance algorithm dynamically examines the resources allocation state to ensure that a circular wait condition never exists. The resource allocation state is defined by the number of available and allocated resources and the maximum demand of each process.

Safe State:

A state is a safe state in which there exists at least one order in which all the process will run completely without resulting in a deadlock.

A system is in safe state if there exists a safe sequence.

A sequence of processes <P1,P2,……..Pn> is a safe sequence for the current allocation state if for each Pi the resources that Pi can request can be satisfied by the currently available resources.

If the resources that Pi requests are not currently available then Pi can obtain all of its needed resource to complete its designated task.

A safe state is not a deadlock state.

Whenever a process request a resource i.e., currently available, the system must decide whether resources can be allocated immediately or whether the process must wait. The request is granted only if the allocation leaves the system in safe state.

In this, if a process requests a resource i.e., currently available it must still have to wait. Thus resource utilization may be lower than it would be without a deadlock avoidance algorithm.

Resource Allocation Graph Algorithm:

☐ This algorithm is used only if we have one instance of a resource type. In addition to the request edge and the assignment edge a new edge called claim edge is used. *For eg*:-A claim edge Pi to Rj indicates that process Pi may request Rj in future. The claim edge is represented by a dotted line. When a process Pi requests the resource Rj, the claim edge is converted to a request edge. When resource Rj is released by process Pi, the assignment edge Rj to Pi is replaced by the claim edge Pi to Rj.

☐ When a process Pi requests resource Rj the request is granted only if converting the request edge Pi to Rj to as assignment edge Rj to Pi do not result in a cycle. Cycle detection algorithm is used to detect the cycle. If there are no cycles then the allocation of the resource to process leave the system in safe state

Banker's Algorithm:

This algorithm is applicable to the system with multiple instances of each resource types, but this is less efficient then the resource allocation graph algorithm.

When a new process enters the system it must declare the maximum number of resources that it may need. This number may not exceed the total number of resources in the system. The system must determine that whether the allocation of the resources will leave the system in a safe state or not. If it is so resources are allocated else it should wait until the process release enough resources.

Several data structures are used to implement the banker's algorithm. Let 'n' be the number of processes in the system and 'm' be the number of resources types. We need the following data structures:

Available:-A vector of length m indicates the number of available resources. If Available[i]=k, then k instances of resource type Rj is available.

Max:-An n*m matrix defines the maximum demand of each process if Max[i,j]=k, then Pi may request at most k instances of resource type Rj.

Allocation:-An n*m matrix defines the number of resources of each type currently allocated to each process. If Allocation[i,j]=k, then Pi is currently k instances of resource type Rj.

Need:-An n*m matrix indicates the remaining resources need of each process. If Need[i,j]=k, then Pi may need k more instances of resource type Rj to compute its task. So Need[i,j]=Max[i,j]-Allocation[i]

Safety Algorithm:

This algorithm is used to find out whether or not a system is in safe state or not.

Step 1. Let work and finish be two vectors of length M and N respectively.

Initialize work = available and Finish[i]=false for i=1,2,3,…….n

Step 2. Find i such that both Finish[i]=false Need i <= work If no such i exist then go 4

Step 3. Work = work + Allocation Finish[i]=true Go to step 2

Step 4. If finish[i]=true for all i, then the system is in safe state. This algorithm may require an order of m*n*n operation to decide whether a state is safe.

Resource Request Algorithm: Let Request(i) be the request vector of process Pi. If Request(i)[j]=k, then process Pi wants K instances of the resource type Rj. When a request for resources is made by process Pi the following actions are taken.

If Request(i) <= Need(i) go to step 2 otherwise raise an error condition since the process has exceeded its maximum claim. If Request(i) <= Available go to step 3 otherwise Pi must wait. Since the resources are not available. If the system want to allocate the requested resources to process Pi then modify the state as follows.

Available = Available – Request(i) Allocation(i) = Allocation(i) + Request(i)

Need(i) = Need(i) – Request(i)

If the resulting resource allocation state is safe, the transaction is complete and Pi is allocated its resources. If the new state is unsafe then Pi must wait for Request(i) and old resource allocation state is restored.

3.3.6 Deadlock Detection

If a system does not employ either deadlock prevention or a deadlock avoidance algorithm then a deadlock situation may occur. In this environment the system may provide x An algorithm that examines the state of the system to determine whether a deadlock has occurred. x An algorithm to recover from the deadlock.

Single Instances of each Resource Type:

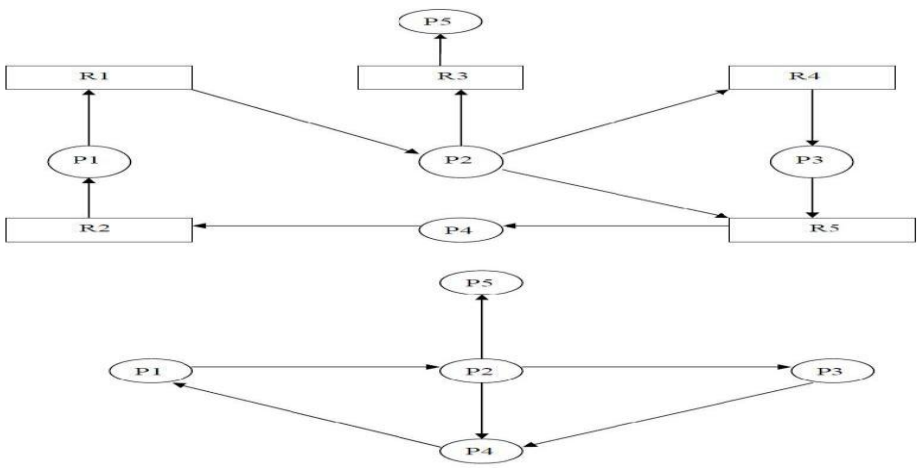
If all the resources have only a single instance then we can define deadlock detection algorithm that uses a variant of resource allocation graph called a wait for

graph. This graph is obtained by removing the nodes of type resources and removing appropriate edges.

An edge from Pi to Pj in wait for graph implies that Pi is waiting for Pj to release a resource that Pi needs.

An edge from Pi to Pj exists in wait for graph if and only if the corresponding resource allocation graph contains the edges Pi□Rq and Rq□Pj.

Deadlock exists within the system if and only if there is a cycle. To detect deadlock the system needs an algorithm that searches for cycle in a graph.



3.3.7 Recovery From Deadlocks

Several Instances of a Resource Types:

The wait for graph is applicable to only a single instance of a resource type. The following algorithm applies if there are several instances of a resource type. The following data structures are used:-

- o Available:-Is a vector of length m indicating the number of available resources of each type .
- o Allocation:-Is an m*n matrix which defines the number of resources of each type currently allocated to each process.
- o Request:-Is an m*n matrix indicating the current request of each process. If

request[i,j]=k then Pi is requesting k more instances of resources type Rj.

Step 1. let work and finish be vectors of length m and n respectively.

Initialize Work = available/expression

For i=0,1,2…….n

if allocation(i)!=0 then Finish[i]=0

else Finish[i]=true

Step 2. Find an index(i) such that both Finish[i] = false Request(i)<=work

If no such I exist go to step 4.

Step 3. Work = work + Allocation(i) Finish[i] = true Go to step 2.

Step 4. If Finish[i] = false for some I where m>=i>=1. When a system is in a deadlock state. This algorithm needs an order of m*n square operations to detect whether the system is in deadlock state or not.

CHAPTER IV

4.1 MASS-STORAGE STRUCTURE

4.1.1 Overview of Mass Storage Structure

Magnetic disks provide bulk of secondary storage of modern computers

– Drives rotate at 60 to 200 times per second

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

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

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

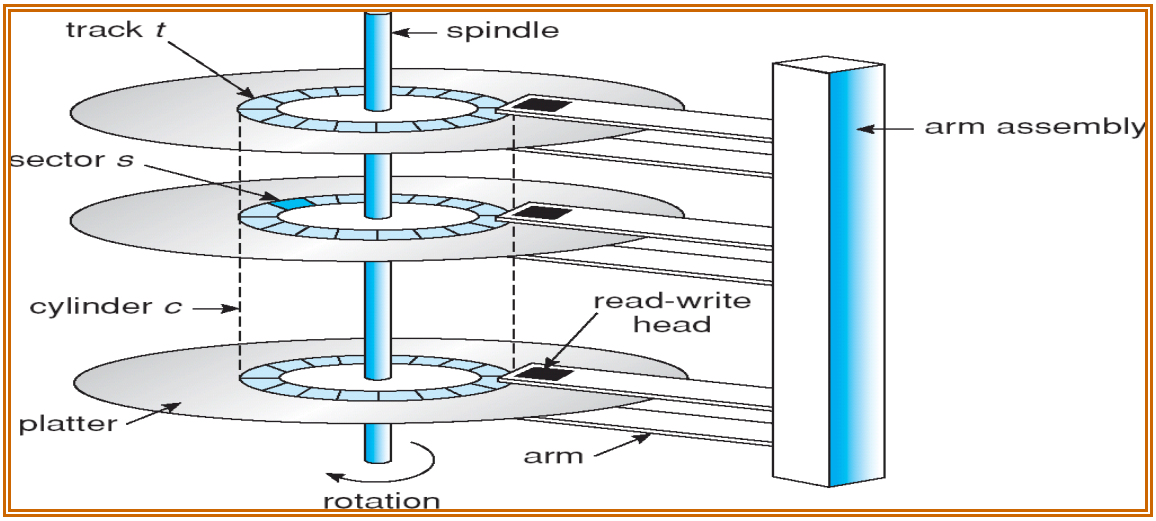
Disks can be removable

Drive attached to computer via I/O bus

– Busses vary, including SATA, USB, Fibre Channel, SCSI

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

Moving-head Disk Mechanism



- **Magnetic tape**

– Was early secondary-storage medium

– Relatively permanent and holds large quantities of data

– Access time slow

– Random access ~1000 times slower than disk

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

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

– Once data under head, transfer rates comparable to disk

– 20-200GB typical storage

4.1.2 Disk Structure

• Disk drives are addressed as large 1-dimensional arrays of *logical blocks*, where the logical block is the smallest unit of transfer.

• The 1-dimensional array of logical blocks is mapped into the sectors of the disk sequentially.

– Sector 0 is the first sector of the first track on the outermost cylinder.

– Mapping proceeds in order through that track, then the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost.

4.1.3 Disk Attachment

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

• SCSI itself is a bus, up to 16 devices on one cable, **SCSI initiator** requests operation and **SCSI targets** perform tasks

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

• FC is high-speed serial architecture

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

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

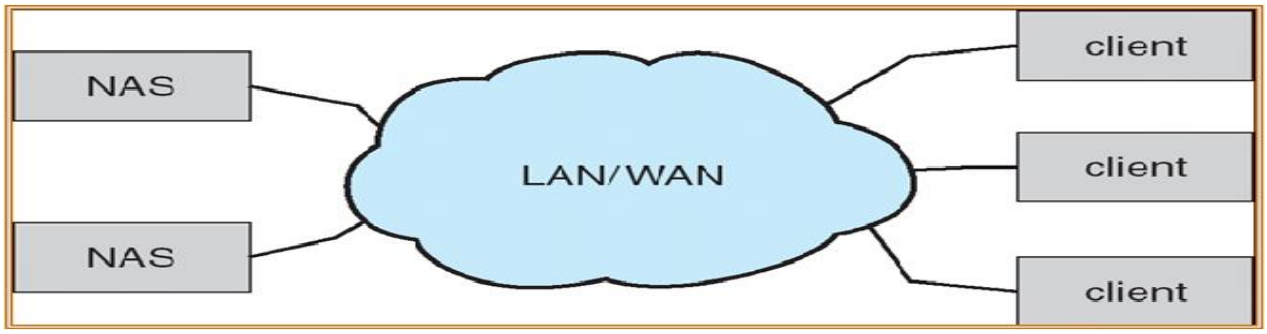
Network-Attached Storage

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

• NFS and CIFS are common protocols

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

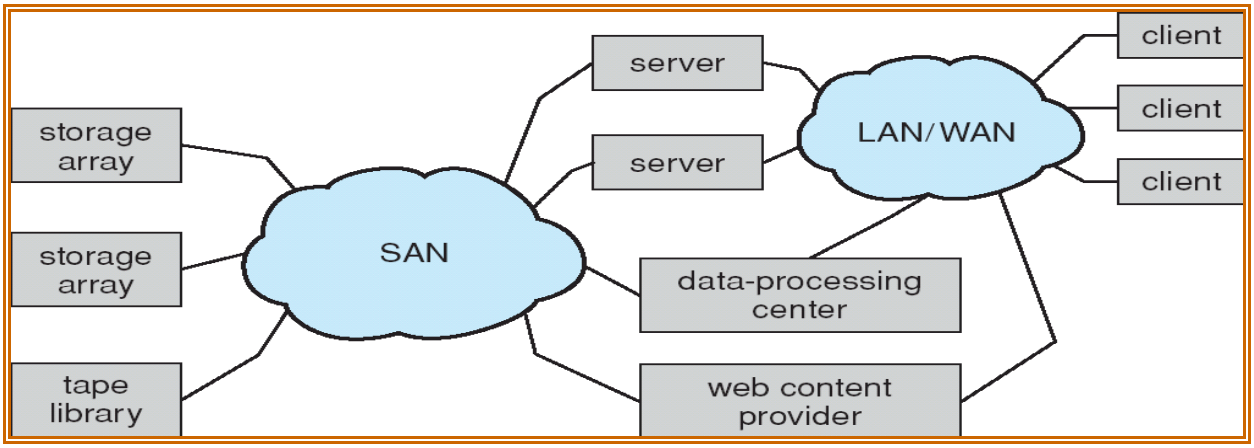
New SCSI protocol uses IP network to carry the SCSI protocol



Storage Area Network

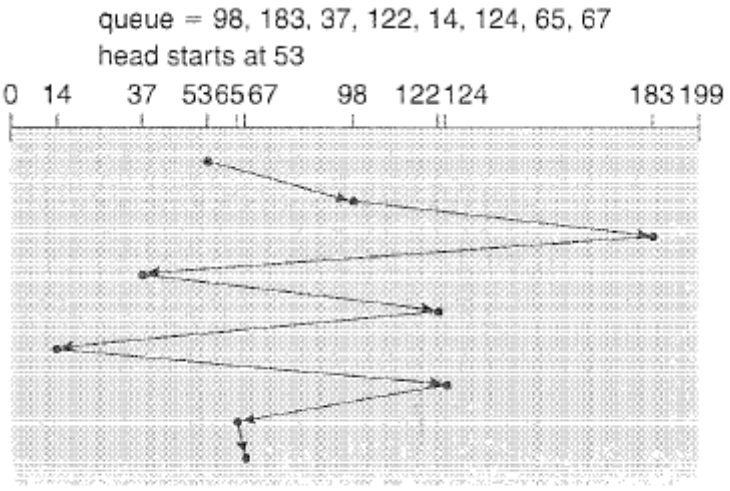
• Common in large storage environments (and becoming more common)

• Multiple hosts attached to multiple storage arrays - flexible



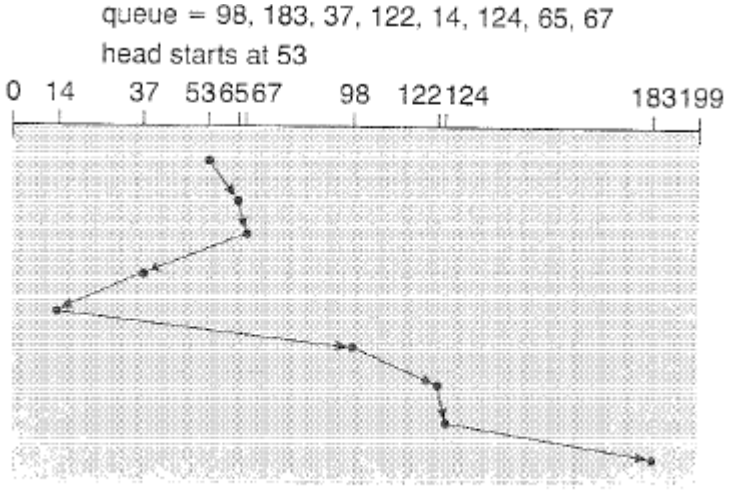
4.1.4 Disk Scheduling

FCFS



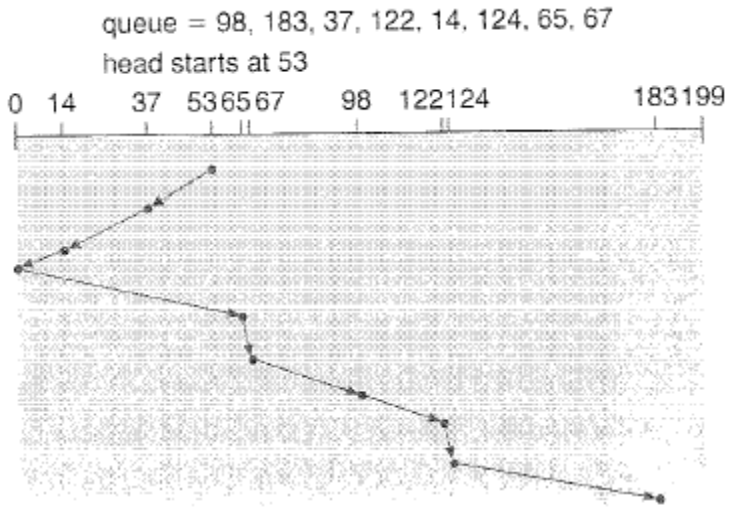
SSTF

- Selects the request with the minimum seek time from the current head position.
- SSTF scheduling is a form of SJF scheduling; may cause starvation of some requests.
- Illustration shows total head movement of 236 cylinders.



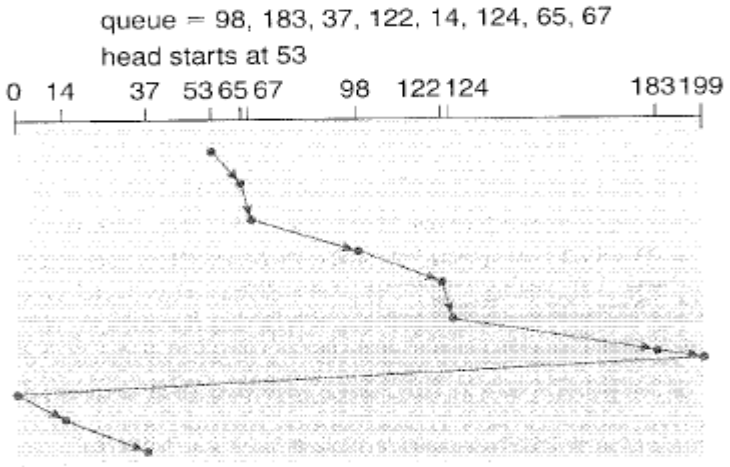
SCAN

- The disk arm starts at one end of the disk, and moves toward the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues.
- Sometimes called the *elevator algorithm*.
- Illustration shows total head movement of 208 cylinders.



C-SCAN

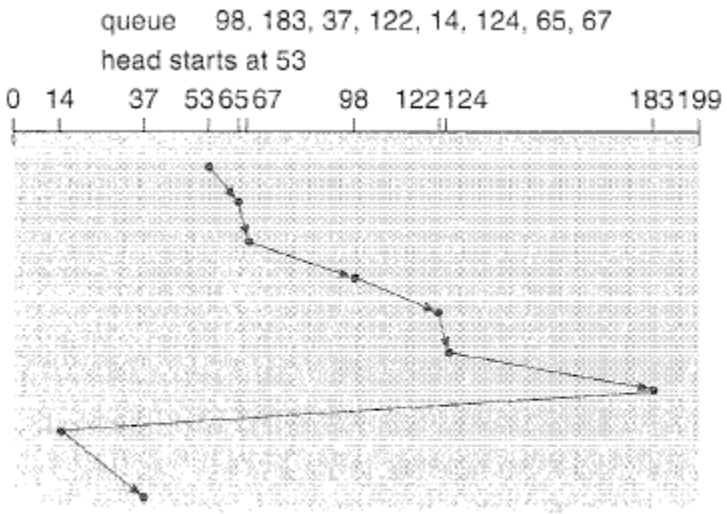
- Provides a more uniform wait time than SCAN.
- The head moves from one end of the disk to the other. servicing requests as it goes. When it reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip.



C-LOOK

- Version of C-SCAN

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



Selecting a Disk-Scheduling Algorithm

- SSTF is common and has a natural appeal
- SCAN and C-SCAN perform better for systems that place a heavy load on the disk.
- Performance depends on the number and types of requests.
- Requests for disk service can be influenced by the file-allocation method.
- The disk-scheduling algorithm should be written as a separate module of the operating system, allowing it to be replaced with a different algorithm if necessary.

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

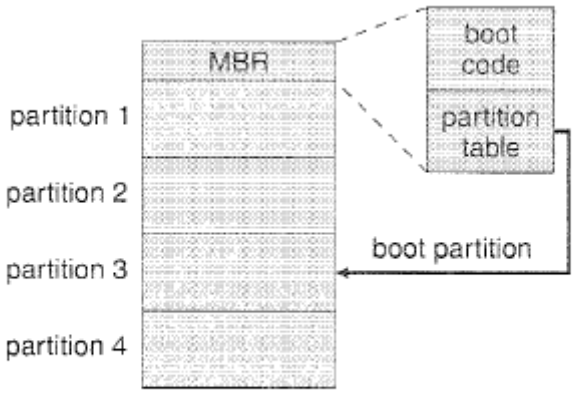
Disk Management

- *Low-level formatting*, or *physical formatting* — Dividing a disk into sectors that the disk controller can read and write.
- To use a disk to hold files, the operating system still needs to record its own data structures on the disk.
  - *Partition* the disk into one or more groups of cylinders.
  - *Logical formatting* or "making a file system".

- Boot block initializes system.
  - The bootstrap is stored in ROM.
  - *Bootstrap loader* program.

Methods such as *sector sparing* used to handle bad blocks

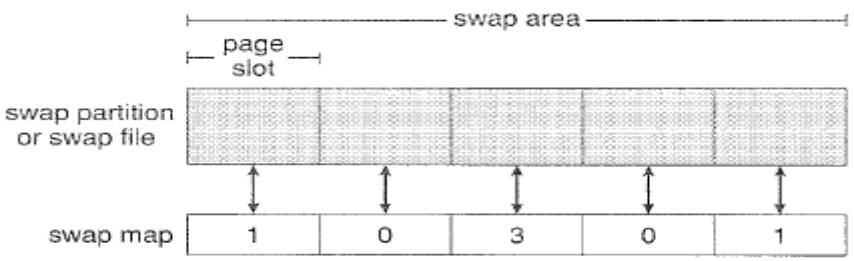
Booting from a Disk in Windows 2000



4.1.5 Swap-Space Management

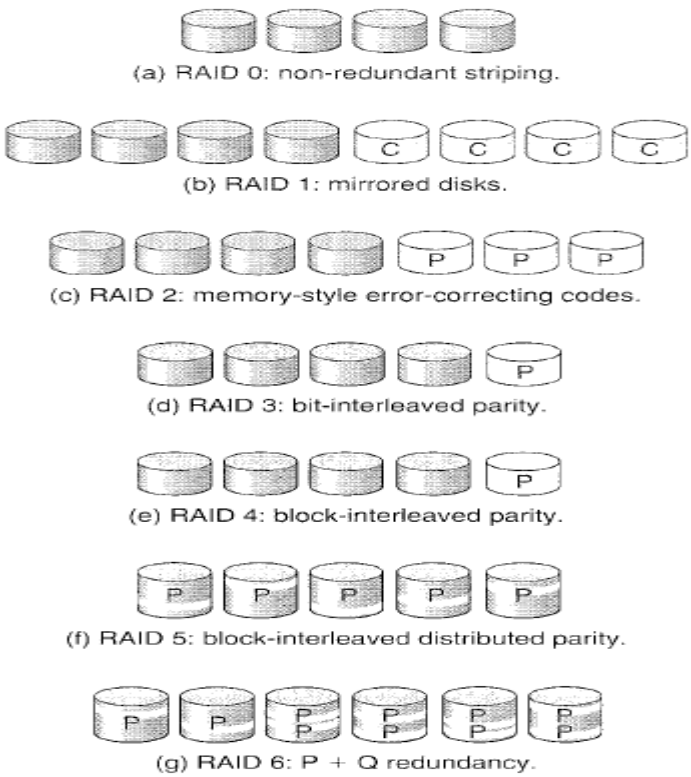
- Swap-space — Virtual memory uses disk space as an extension of main memory.
- Swap-space can be carved out of the normal file system,or, more commonly, it can be in a separate disk partition.
- Swap-space management
  - 4.3BSD allocates swap space when process starts; holds *text segment* (the program) and *data segment.*
  - Kernel uses *swap maps* to track swap-space use.
  - Solaris 2 allocates swap space only when a page is forced out of physical memory, not when the virtual memory page is first created**.**

Data Structures for Swapping on Linux Systems



4.1.6 RAID Structure

- RAID – multiple disk drives provides reliability via redundancy.
- RAID is arranged into six different levels.
- Several improvements in disk-use techniques involve the use of multiple disks working cooperatively.
- Disk striping uses a group of disks as one storage unit.
- RAID schemes improve performance and improve the reliability of the storage system by storing redundant data.
  - *Mirroring* or *shadowing* keeps duplicate of each disk.
  - *Block interleaved parity* uses much less redundancy.



(a) RAID 0: non-redundant striping.

(b) RAID 1: mirrored disks.

(c) RAID 2: memory-style error-correcting codes.

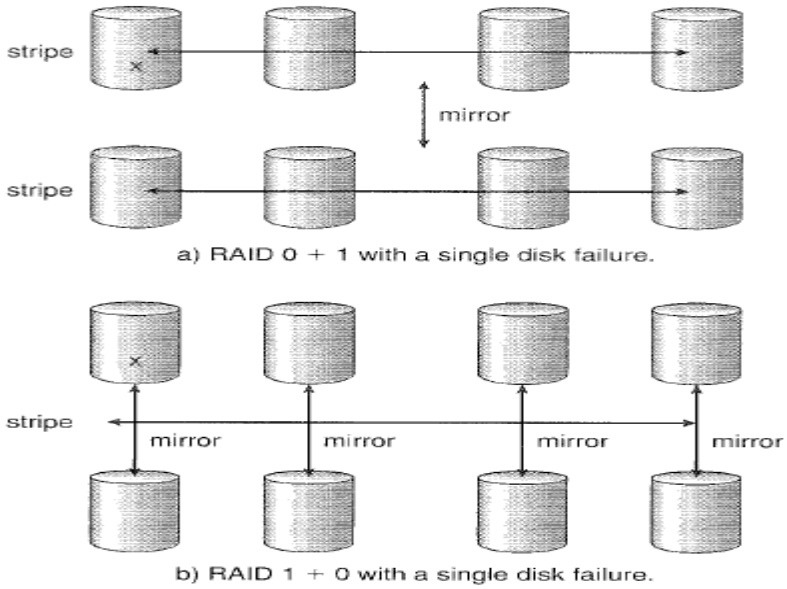
(d) RAID 3: bit-interleaved parity.

(e) RAID 4: block-interleaved parity.

(f) RAID 5: block-interleaved distributed parity.

(g) RAID 6: P + Q redundancy.

RAID (0 + 1) and (1 + 0)



a) RAID 0 + 1 with a single disk failure.

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

4.1.7 Stable-Storage Implementation

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

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

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

Tertiary Storage Devices

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

Removable Disks

- Floppy disk — thin flexible disk coated with magnetic material, enclosed in a protective plastic case.

– Most floppies hold about 1 MB; similar technology is used for removable disks that hold more than 1 GB.

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

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

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

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

– The magneto-optic head flies much farther from the disk surface than a magnetic disk head, and the magnetic material is covered with a protective layer of plastic or glass; resistant to head crashes.

• Optical disks do not use magnetism; they employ special materials that are altered by laser light.

WORM Disks

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

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

• Thin aluminum film sandwiched between two glass or plastic platters.

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

• Very durable and reliable.

Read Only disks, such ad CD-ROM and DVD, com from the factory with the data pre-recorded

Tapes

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

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

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

– stacker – library that holds a few tapes

– silo – library that holds thousands of tapes

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

Application Interface

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

• Tapes are presented as a raw storage medium, i.e., and application does not not

open a file on the tape, it opens the whole tape drive as a raw device.

- Usually the tape drive is reserved for the exclusive use of that application.
- Since the OS does not provide file system services, the application must decide how to use the array of blocks.
- Since every application makes up its own rules for how to organize a tape, a tape full of data can generally only be used by the program that created it.

Tape Drives

- The basic operations for a tape drive differ from those of a disk drive.
- locate positions the tape to a specific logical block, not an entire track (corresponds to seek).
- The read position operation returns the logical block number where the tape head is.
- The space operation enables relative motion.
- Tape drives are "append-only" devices; updating a block in the middle of the tape also effectively erases everything beyond that block.
- An EOT mark is placed after a block that is written**.**

File Naming

- The issue of naming files on removable media is especially difficult when we want to write data on a removable cartridge on one computer, and then use the cartridge in another computer.
- Contemporary OSs generally leave the name space problem unsolved for removable media, and depend on applications and users to figure out how to access and interpret the data.
- Some kinds of removable media (e.g., CDs) are so well standardized that all computers use them the same way.

Hierarchical Storage Management (HSM)

- A hierarchical storage system extends the storage hierarchy beyond primary memory and secondary storage to incorporate tertiary storage — usually implemented as a jukebox of tapes or removable disks.
- Usually incorporate tertiary storage by extending the file system.
  - Small and frequently used files remain on disk.
  - Large, old, inactive files are archived to the jukebox.

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

Speed

Two aspects of speed in tertiary storage are bandwidth and latency. Bandwidth is measured in bytes per second. Sustained bandwidth – average data rate during a large transfer; # of bytes/transfer time. Effective bandwidth – average over the entire I/O time, including seek or locate, and cartridge switching. Access latency – amount of time needed to locate data. Access time for a disk – move the arm to the selected cylinder and wait for the rotational latency; < 35 milliseconds. Access on tape requires winding the tape reels until the selected block reaches the tape head; tens or hundreds of seconds. Generally say that random access within a tape cartridge is about a thousand times slower than random access on disk. The low cost of tertiary storage is a result of having many cheap cartridges shares a few expensive drives. A removable library is best devoted to the storage of infrequently used data, because the library can only satisfy a relatively small number of I/O requests per hour

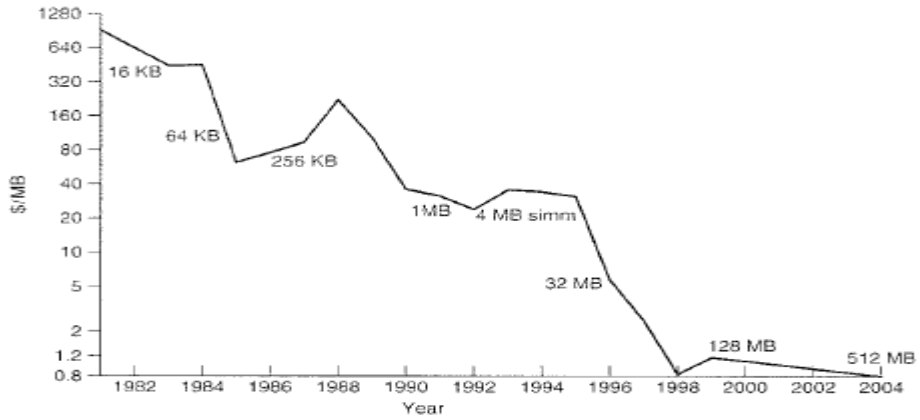
Reliability

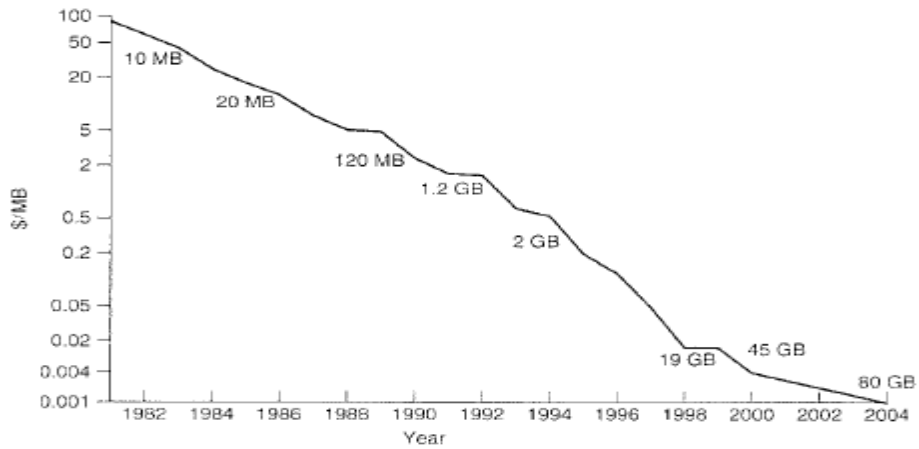
A fixed disk drive is likely to be more reliable than a removable disk or tape drive. An optical cartridge is likely to be more reliable than a magnetic disk or tape. A head crash in a fixed hard disk generally destroys the data, whereas the failure of a tape drive or optical disk drive often leaves the data cartridge unharmed.

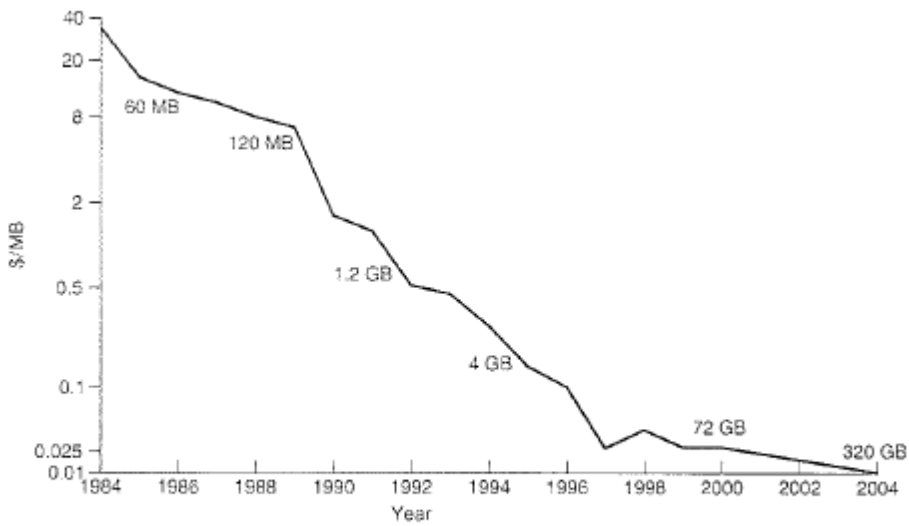
Cost

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

Price per Megabyte of DRAM, From 1981 to 2004







FILE SYSTEM IMPLEMENTATION

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.

4.2.1 The concept of a file

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.

File Attributes

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. A

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.

File Operations

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 file seek.

Deleting a file – To delete a file, search the directory for the named file. When found, release 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 with the named file. To avoid this constant search, many systems require that an open () system 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.

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 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

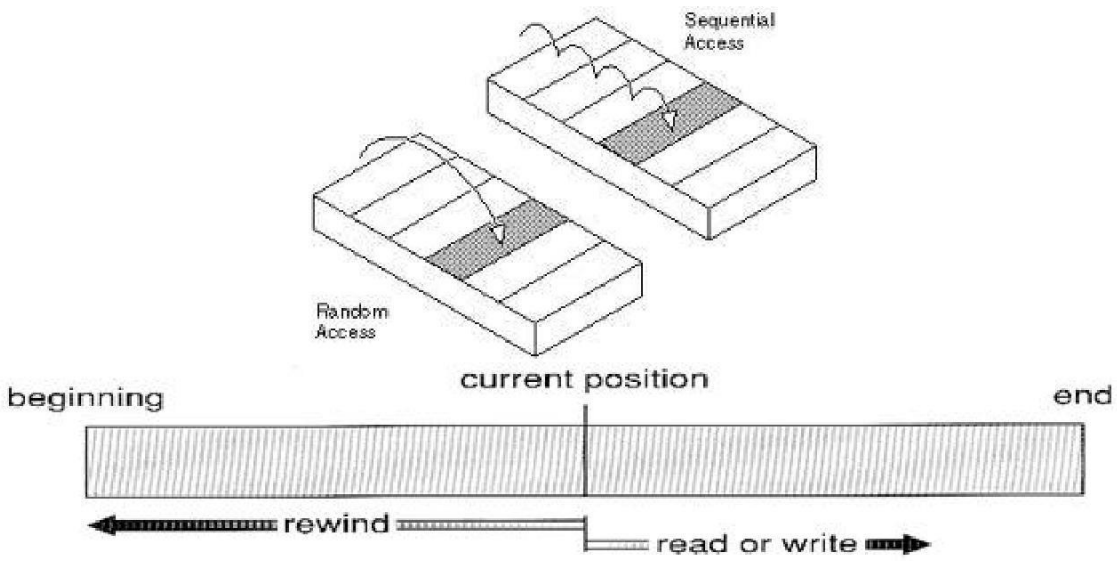
Internally locating an offset within a file can be complicated for the OS. Disk systems have a well defined block size determined by the size of the sector. All disk I/O is performed in units of one block and all blocks are the same size. Since it is unlikely that the physical record size will exactly 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.

4.2.2 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 –

* 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



Sequential-access file.

* 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.

* 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.

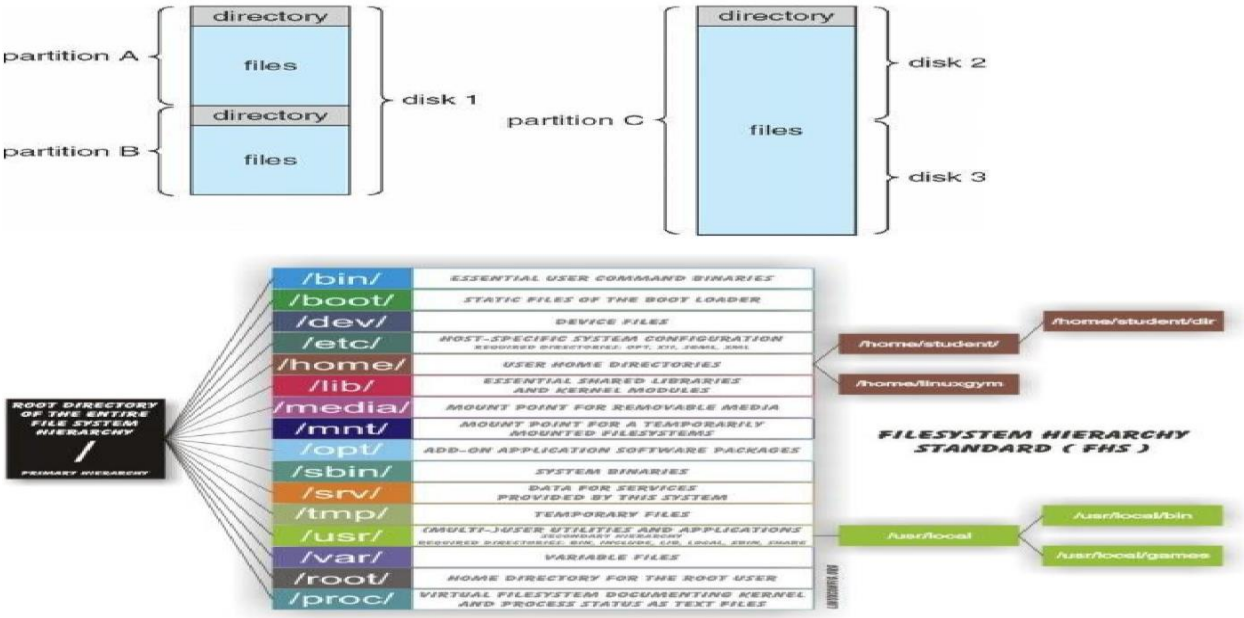
4.2.3 Directory Structure

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.



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

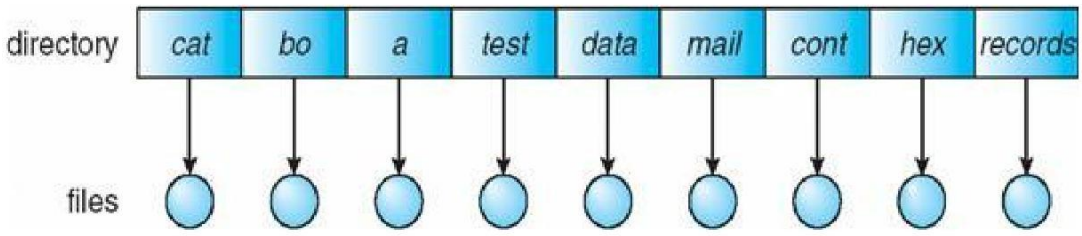
List a directory

Rename a file

Traverse the file system

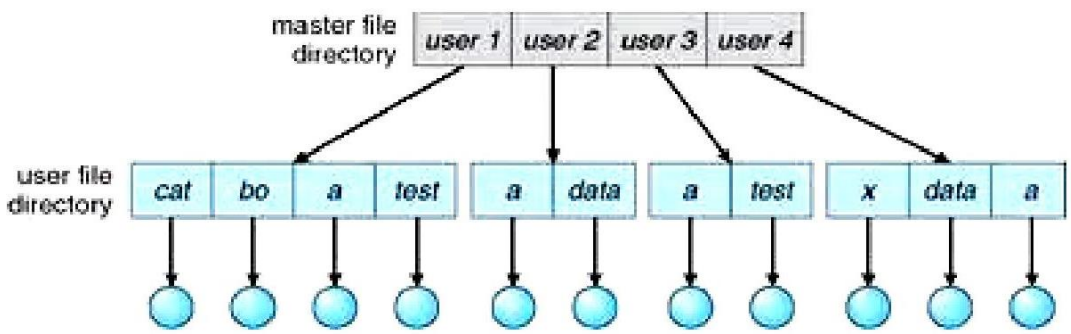
Single level directory

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.

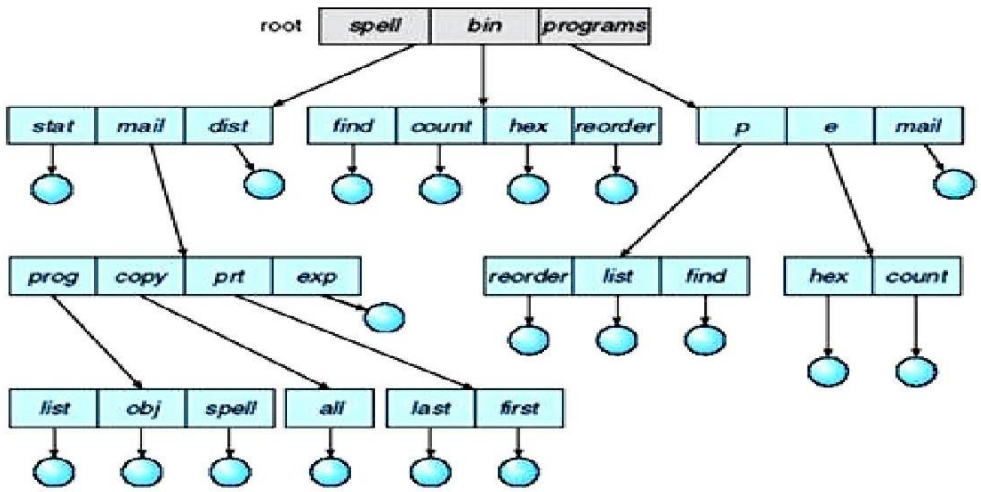


Root of the tree is MFD. Its direct descendants are UFDs. The descendants of the UFDs are the files themselves. The files are the leaves of the tree. The sequence of directories searched when a file is names is called the **search path**.

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.

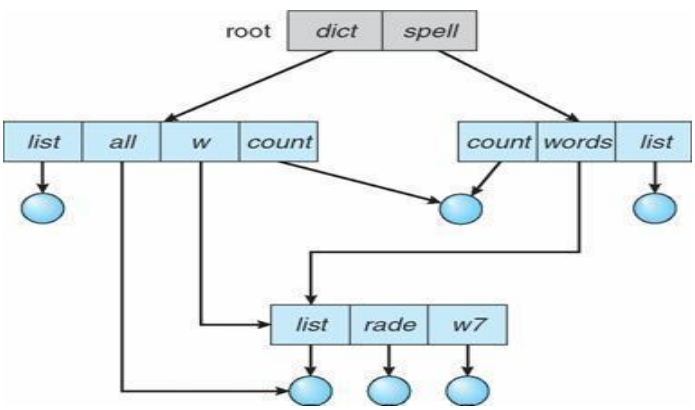
A path to a file in a tree structured directory can be longer than a path in a two

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 be in two different directories.

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.

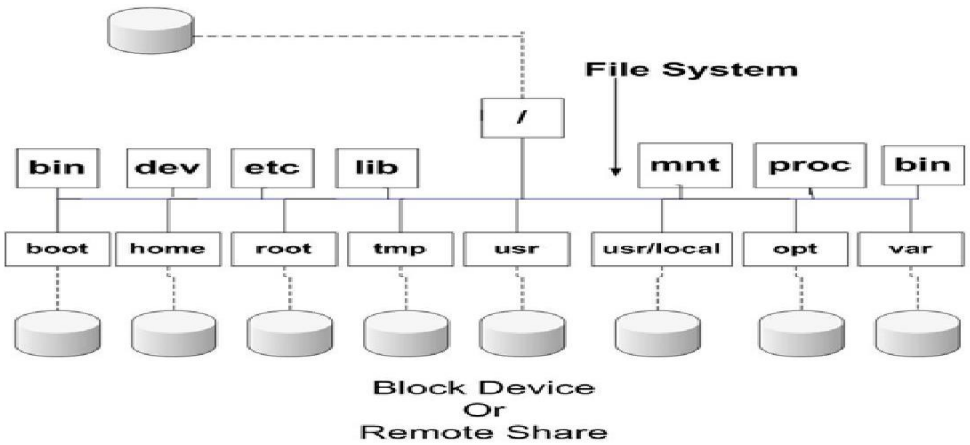
General graph directory

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.

4.2.4 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.



4.2.5 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.

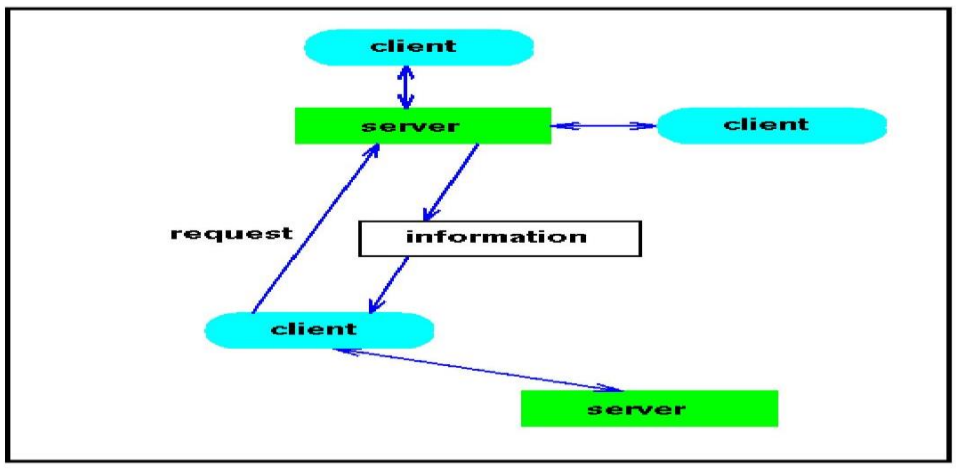
Remote File Systems

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 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

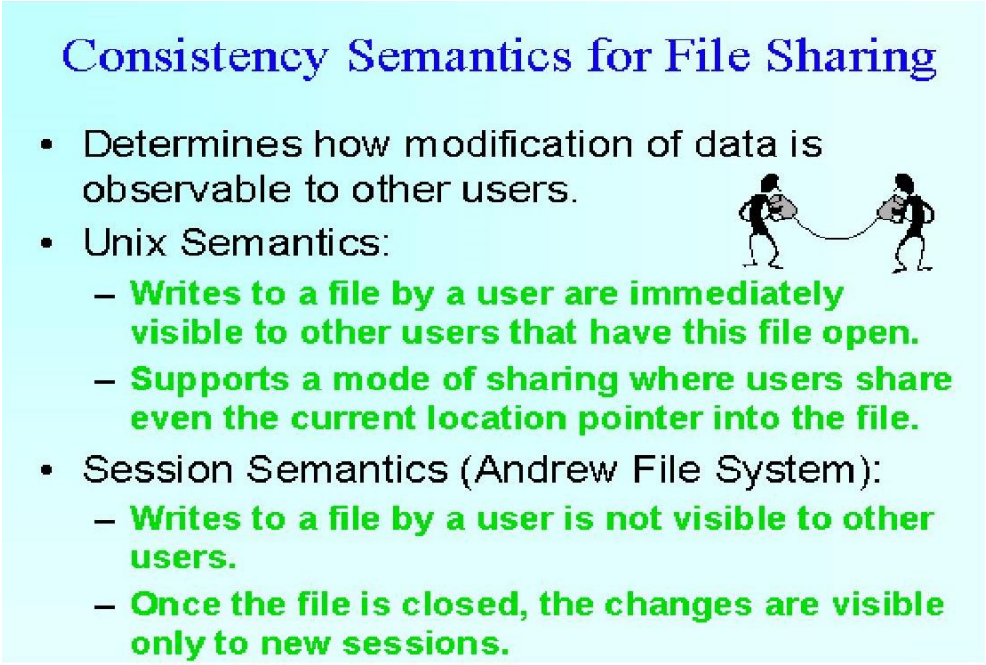
Local file systems can fail for a variety of reasons including failure of the disk containing the file system, corruption of the delivery structure or other disk management information, disk controller failure, cable failure and host adapter failure. User or system administrator failure can also cause 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.



4.2.6 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.

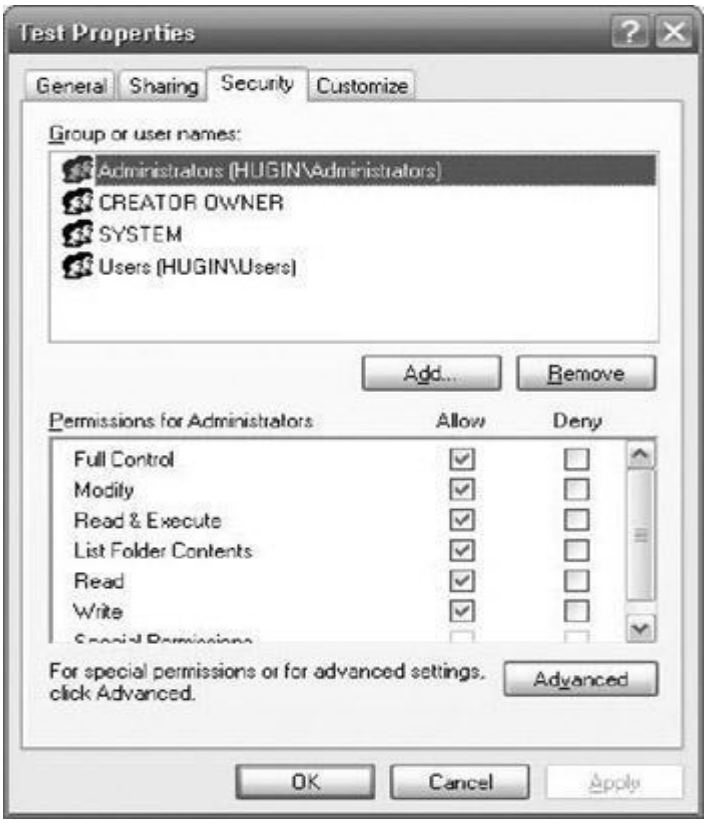
Access Control

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.



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.

4.3 IMPLEMENTING FILE SYSTEMS

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.

4.3.1 File System Structure

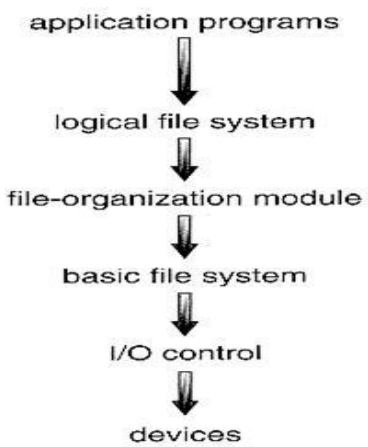
Disks provide the bulk of secondary storage on which a file system is maintained. They have two characteristics that make them a convenient medium for storing multiple files:

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 –



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 interrupt 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.

4.3.2 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.

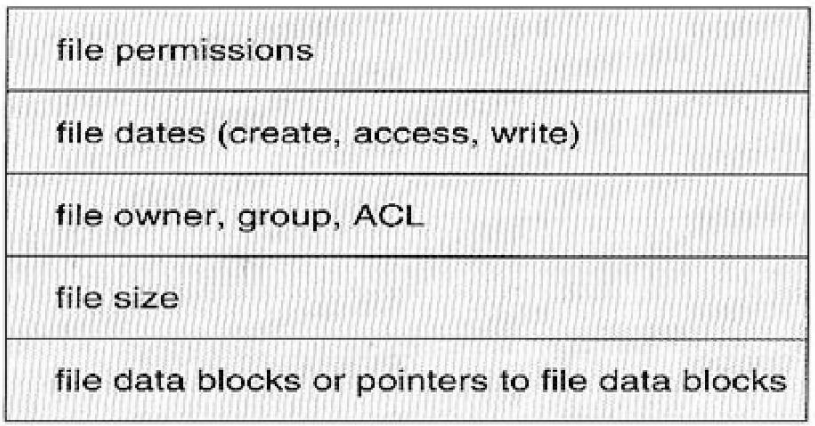
A per fie FCB contains many details about the file, including file permissions, ownership, size and location of data blocks. In UFS, it is called the inode. In NTFS this is stored within the master file table which uses a relational database structure.

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.



A typical file-control block.

Partitions and Mounting

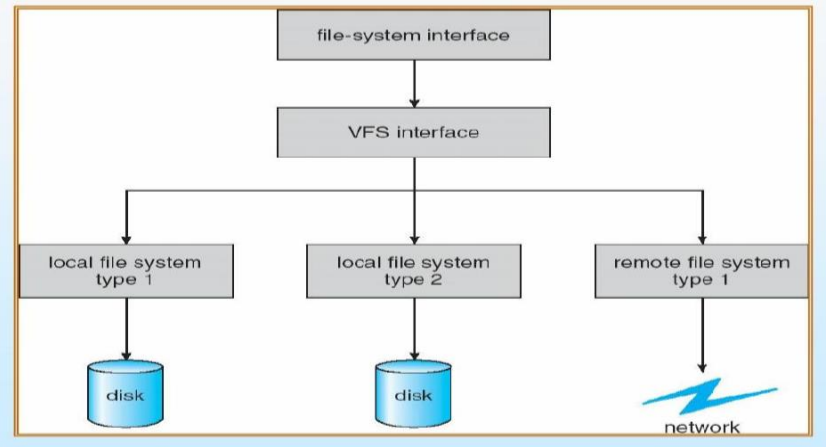
The layout of a disk can have many variations depending on the OS. A disk can be sliced into multiple partitions or a volume can span multiple partitions on multiple disks. Each partition can be either raw containing no file system or may contain a file system. Raw disk is used where no file system is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



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.

4.3.3 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.

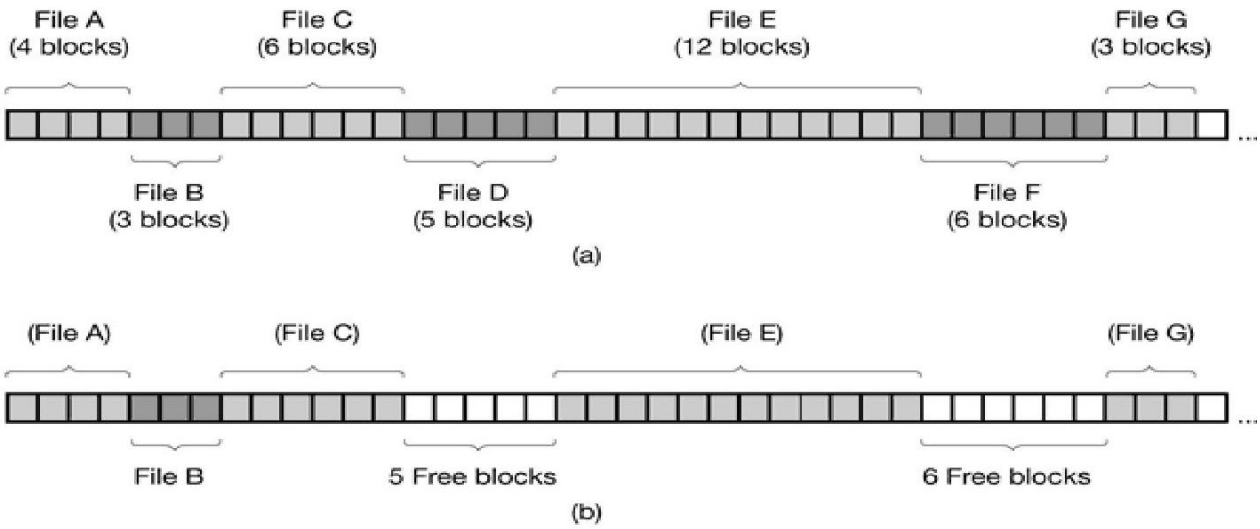
4.3.4 Allocation Methods

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.



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.

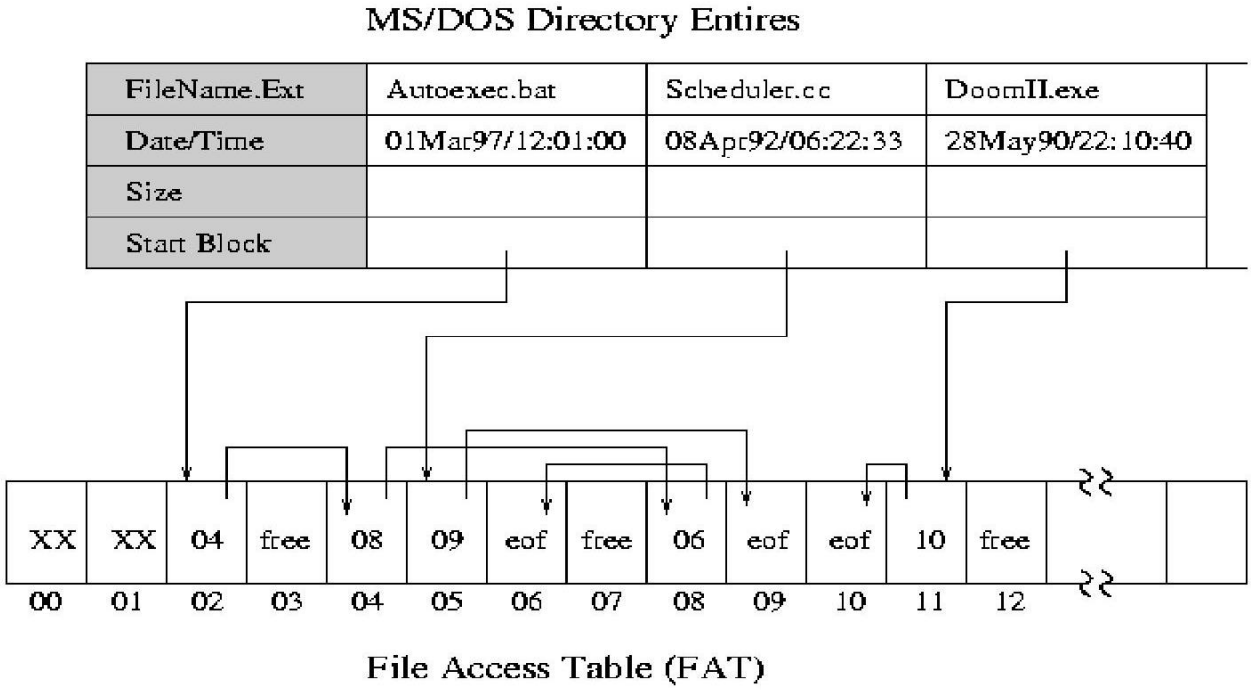
To minimize these drawbacks, some operating systems use a modified contiguous allocation scheme. Here a contiguous chunk of space is allocated initially and if that amount proves not to be large enough another chunk of contiguous space called **extent** is added. Internal fragmentation can still be a 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.
An important variation of linked allocation is the use of a file allocation table (FAT).

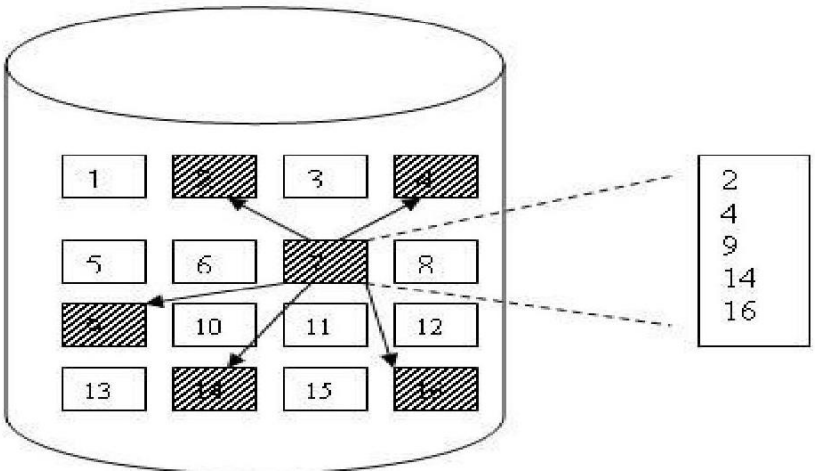


Indexed Allocation

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 –

1. Linked scheme 2. Multilevel index 3. Combined scheme

Indexed allocation scheme suffers from some of the same performance problems as does linked allocation.

Performance

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.

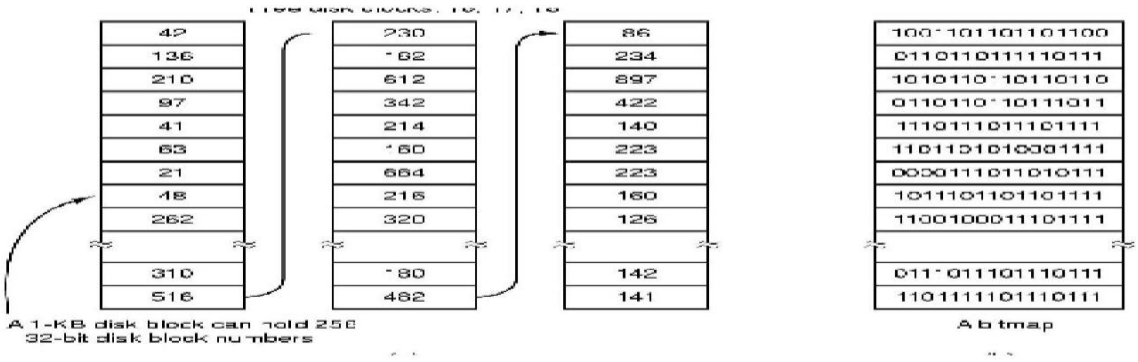
For any type of access, contiguous allocation requires only one access to get a disk block. For linked allocation, we can keep the address of the next block in memory and read it directly. This method is fine for sequential access. Hence some systems support direct access files by using contiguous allocation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

4.3.5 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.



c) Grouping – A modification of the free 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.

Efficiency and Performance

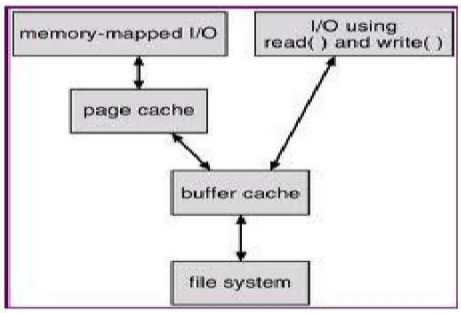
Disks tend to represent a major bottleneck in system performance since they are the slowest main computer component.

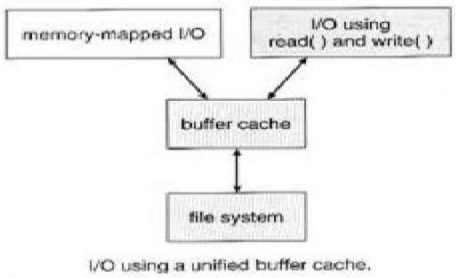
Efficiency

The efficient use of disk space depends heavily on the disk allocation and directory algorithms in use.

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.

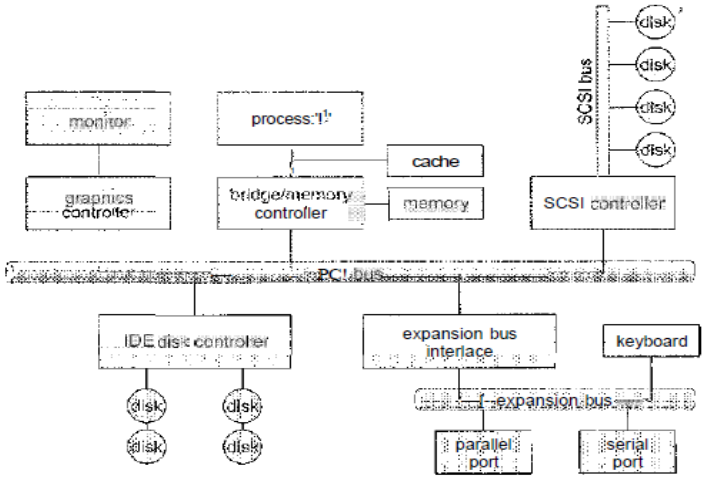
Some systems optimize their page cache by using different replacement algorithms depending on the access type of the file. Sequential access can be optimized by techniques known as free behind and read ahead. Free behind removes a page from buffer as soon as the next page is requested. With read ahead, a requested page and several subsequent pages are read and cached.

CHAPTER -5

5.1.1 I/O Hardware

Computers operate a great many kinds of devices. Most fit into the general categories of storage devices (disks, tapes), transmission devices (network cards, modems), and human-interface devices (screen, keyboard, mouse). Other devices are more specialized, such as the steering of a military fighter jet or a space shuttle. In these aircraft, a human gives input to the flight computer via a joystick and foot pedals, and the computer sends output commands that cause motors to move rudders, flaps, and thrusters.

A device communicates with, a computer system by sending signals over a cable or even through the air. The device communicates with the machine via a connection point (or port)—for example, a serial port. If devices use a common set of wires, the connection is called a *bus.* A **bus** is a set of wires and a rigidly defined protocol that specifies a set of messages that can be sent on the wires. In terms of the electronics, the messages are conveyed by patterns of electrical voltages applied to the wires with defined timings. When 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.



A PCI bus (the common PC system 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 on a SCSI bus plugged into a SCSI controller.

A controller 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 (or portion of a chip) in the computer that controls the signals on the wires of a serial port. By contrast, a SCSI bus controller is not simple. Because the SCSI protocol is complex, the SCSI bus controller is often implemented as a separate circuit board (or a **host** adapter) that plugs into the computer. It typically contains a processor, microcode, and some private memory to enable it to process the SCSI protocol messages.

This board is the disk controller. It implements the disk side of the protocol for some kind of connection—SCSI or ATA, for instance. It has microcode and a processor to do many tasks, such as bad-sector mapping, prefetching, buffering, a caching.

Polling

The complete protocol for interaction between the host and a controller can be intricate, but the basic *handshaking* notion is simple. We assume that 2 bits are used to coordinate the producer-consumer relationship between the controller and the host. The controller indicates its state through the *busy* bit in the *status* register. 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 signals its wishes via the *command-ready* bit in the *command* register. The host sets the *command-ready* bit when a command is available for the controller to execute.

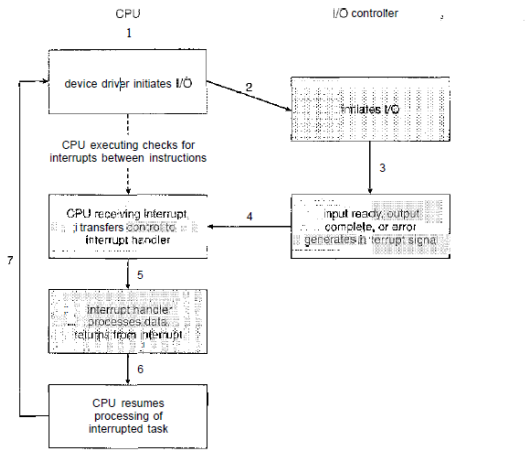
1. The host repeatedly reads the *busy* bit until that bit becomes clear.
2. The host sets the *write,* bit in the *command* register and writes a byte into the *data-out* register.
3. The host sets the *command-ready* bit.
4. When the controller notices that the *command-ready* bit is set, it sets the *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.

Interrupts

The CPU hardware has a wire called the **interrupt-request line** that the CPU senses after executing every instruction. When the CPU detects that a controller has asserted a signal on the interrupt request line, the CPU performs a state save and jumps to the **interrupt handler** routine at a fixed address in memory. The interrupt handler determines the cause of the interrupt, performs the necessary processing, performs a state restore, and executes a return from interrupt instruction to return the CPU to the execution state prior to the interrupt.



This basic interrupt mechanism enables the CPU to respond to an asynchronous event, as when a device controller becomes ready for service. In a modern operating system, however, we need more sophisticated, interrupt handling features.

1. We need the ability to defer interrupt handling during critical processing.

2. We need an efficient way to dispatch to the proper interrupt handler for a device without first polling all the devices to see which one raised the interrupt.

3. We need multilevel interrupts, so that the operating system can distinguish between high- and low-priority interrupts and can respond with the appropriate degree of urgency.

In modern computer hardware, these three features are provided by the CPU and by the **interrupt-controller** hardware.

Most CPUs have two interrupt request lines. One is the nonmaskable interrupt, which is reserved for events such as unrecoverable memory errors. The second interrupt line is maskable: It can be turned off by the CPU before the execution of critical instruction sequences that must not be interrupted. The maskable interrupt is used by device controllers to request service. The interrupt mechanism accepts an address—a number that selects a specific interrupt-handling routine from a small set. In most architectures, this address is an offset in a table called the interrupt **vector.** This vector contains the memory addresses of specialized interrupt handlers. The purpose of a vectored interrupt mechanism is to reduce the need for a single interrupt handler to search all possible sources of interrupts to determine which one needs service.

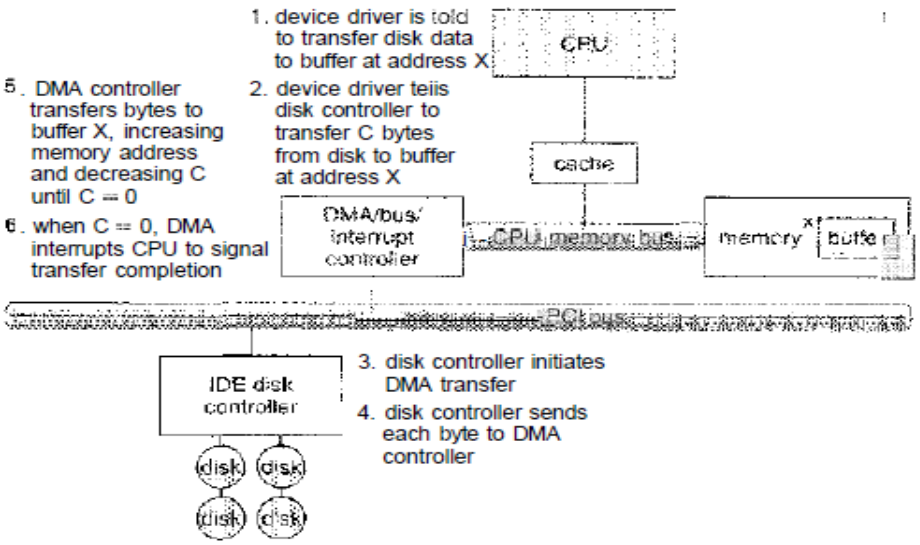
A common way to solve this problem is to use the technique of **interrupt chaining,** in which each element in the interrupt vector points to the head of a list of interrupt handlers. When an interrupt is raised, the handlers on the corresponding list are called one by one, until one is found that can service the request. This structure is a compromise between the overhead of a huge interrupt table and the inefficiency of dispatching to a single interrupt handler.

Direct Memory Access

For a device that does large transfers, such as a disk drive, it seems wasteful to use an expensive general-purpose processor to watch status bits and to feed data into a controller register one byte at a time—a process termed **programmed I/O (PIO).** Many computers avoid burdening the main CPU with PIO by offloading some of this work to a special-purpose processor called a **direct-memory-access (DMA)** controller. To initiate a DMA transfer, the host writes a DMA command block into memory. This block contains a pointer to the source of a transfer, a pointer to the destination of the transfer, and a count of the number of bytes to be transferred. The CPU writes the address of this command block to the DMA controller, then goes on with other work. The DMA controller proceeds to operate the memory bus directly, placing addresses on the bus to perform transfers without the help of the main CPU. A simple DMA controller is a standard component in PCs, and **bus-**

mastering I/O boards for the PC usually contain their own high-speed DMA hardware.

Handshaking between the DMA controller and the device controller is performed via a pair of wires called DMA-request and DMA-acknowledge. The device controller places a signal on the DMA-request wire when a word of data is available for transfer. This signal causes the DMA controller to seize the memory bus, to place the desired, address on the memory-address wires, and to place a signal on the DMA-acknowledge wire. When the device controller receives the DMA-acknowledge signal, it transfers the w7ord of data to memory and removes the DMA-request signal.

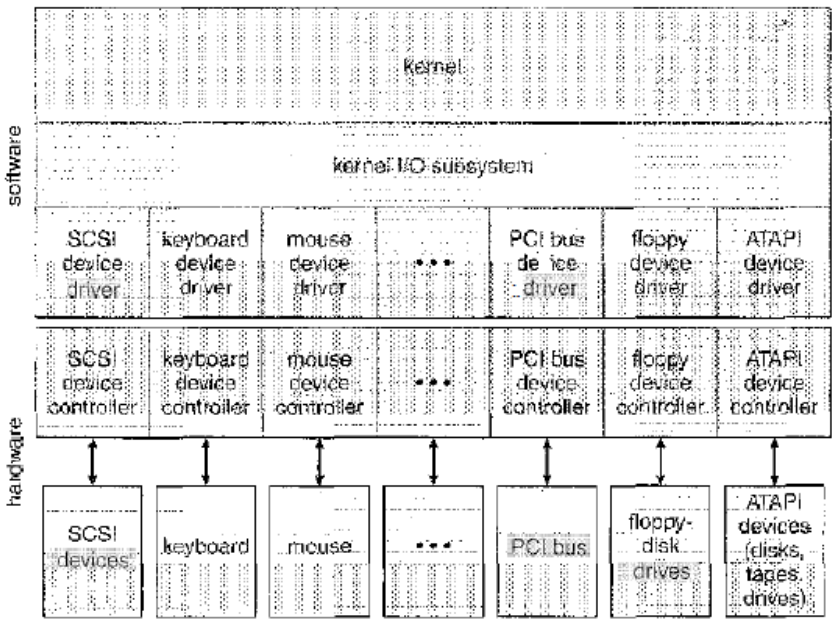


5.1.2 Application I/O interface

Each general kind is accessed through a standardized set of functions—an interface. The differences are encapsulated in kernel modules called device drivers that internally are custom-tailored to each device but that export one of the standard interfaces.

The purpose of the device-driver layer is to hide the differences among device controllers from the I/O subsystem of the kernel, much as the I/O system calls encapsulate the behavior of devices in a few generic classes that hide hardware differences from applications. Making the I/O subsystem independent of the

hardware simplifies the job of the operating-system developer. It also benefits the hardware manufacturers. They either design new devices to be compatible with an existing host controller interface (such as SCSI-2), or they write device drivers to interface the new hardware to popular operating systems.



• **Character-stream or block.** A character-stream, device transfers bytes one by one, whereas a block device transfers a block of bytes as a unit.

• **Sequential or random-access.** A sequential device transfers data in a fixed order determined by the device, whereas the user of a random-access device can instruct the device to seek to any of the available data storage locations.

• **Synchronous or asynchronous.** A synchronous device performs data transfers with predictable response times. An asynchronous device exhibits irregular or unpredictable response times.

• **Sharable or dedicated.** A sharable device can be used concurrently by several processes or threads; a dedicated device cannot.

Speed of operation. Device speeds range from a few bytes per second to a few gigabytes per second.

• Read**-write, read only, or write only.** Some devices perform both input and output, but others support only one data direction.

Block and Character Devices

The block-device interface captures all the aspects necessary for accessing disk drives and other block-oriented devices. The device is expected to understand commands such as read () and wr i t e (); if it is a random-access device, it is also expected to have a seek() command to specify which block to transfer next. Applications normally access such a device through a file-system interface. The operating system itself, as well as 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. If the application performs its own buffering, then using a file system would cause extra, unneeded buffering.

Network Devices

The system calls in the socket interface enable an application to create a socket, to connect a local socket to a remote address (which plugs this application into a socket created by another application), to listen for any remote application to plug into the local socket, and to send and receive packets over the connection. To support the implementation of servers, the socket interface also provides a function called select () that manages a set of sockets. A call to select () returns information about which sockets have a packet waiting to be received and which sockets have room to accept a packet to be sent. The use of select () eliminates the polling and busy waiting that would otherwise be necessary for network I/O. These functions encapsulate the essential behaviors of networks, greatly facilitating the creation of distributed applications that can use any underlying network hardware and protocol stack.

Clocks and Timers

Most computers have hardware clocks and timers that provide three basic functions:

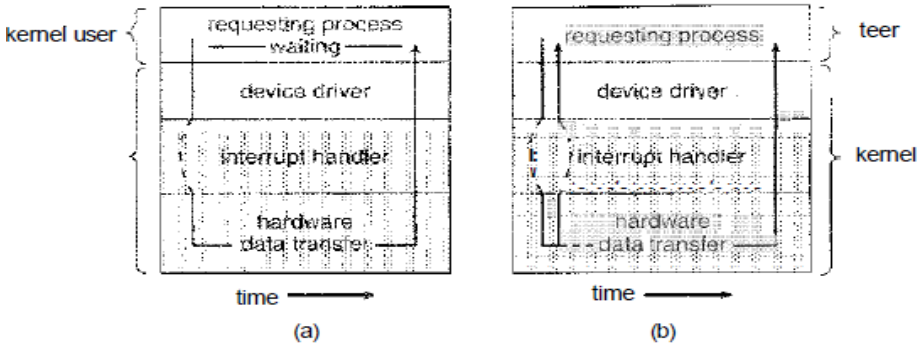
- Give the current time.
- Give the elapsed time.
- Set a timer to trigger operation X at time *T.*

These functions are used heavily by the operating system, as well as by time sensitive applications. Unfortunately, the system calls that implement these functions are not standardized across operating systems.

The hardware to measure elapsed time and to trigger operations is called a **programmable interval timer.** It can be set to wait a certain amount of time and then generate an interrupt, and it can be set to do this once or to repeat the process to generate periodic interrupts. The scheduler uses this mechanism to generate an interrupt that will preempt a process at the end of its time slice. The disk I/O subsystem uses it to invoke the flushing of dirty cache buffers to disk periodically, and the network subsystem uses it to cancel operations that are proceeding too slowly because of network congestion or failures. The operating system may also provide an interface for user processes to use timers. The operating system can support more timer requests than the number of timer hardware channels by simulating virtual clocks.

Blocking and Non blocking IO

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. The physical actions performed by I/O devices are generally asynchronous—they take a varying or unpredictable amount of time. Nevertheless, most operating systems use blocking system calls for the application interface, because blocking application code is easier to understand than non blocking application code.

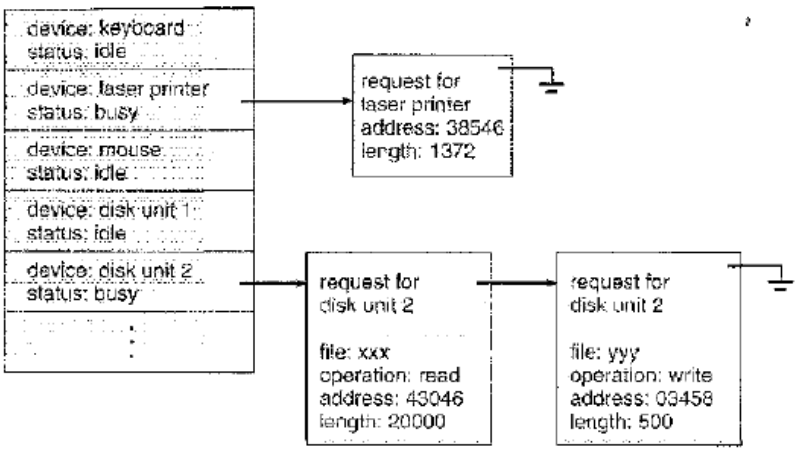


5.1.3 Kernel I/O Subsystem

I/O Scheduling

To schedule a set of I/O requests means to determine a good order in which to execute them. The order in which applications issue system calls rarely is the best choice. Scheduling can improve overall system performance, can share device access fairly among processes, and can reduce the average waiting time for I/O to complete. Here is a simple example to illustrate the opportunity. Suppose that a disk arm is near the beginning of a disk and that three applications issue blocking read calls to that disk. Application 1 requests a block near the end of the disk, application 2 requests one near the beginning, and application 3 requests one in the middle of the disk. The operating system can reduce the distance that the disk arm travels by serving the applications in the order 2, 3,1. Rearranging the order of service in this way is the essence of I/O scheduling.

Operating-system developers implement scheduling by maintaining a wait queue of requests for each device. When an application issues a blocking I/O system call, the request is placed on the queue for that device. The I/O scheduler rearranges the order of the queue to improve the overall system efficiency and the average response time experienced by applications. The operating system may also try to be fair, so that no one application receives especially poor service, or it may give priority service for delay-sensitive requests.



When a kernel supports asynchronous I/O, it must be able to keep track of many I/O requests at the same time. For this purpose, the operating system might attach the wait queue to a **device-status table.**

Buffering

A **buffer** is a memory area that stores data while they are transferred between two devices or between a device and an application. Buffering is done for three reasons. One reason is to cope with a speed mismatch between the producer and consumer of a data stream. Suppose, for example, that a file is being received via modem for storage on the hard disk. The modem then starts to fill the second buffer while the first buffer is written to disk. By the time the modem has filled the second buffer, the disk write from the first one should have completed, so the modem can switch back to the first buffer while the disk writes the second one. This **double buffering** decouples the producer of data from the consumer, thus relaxing timing requirements between them.

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. For instance, the instructions of the currently running process are stored on disk, cached in physical memory, and copied again in the CPU's secondary and primary caches. The difference between a buffer and a cache is that a buffer may hold the only existing copy of a data item, whereas a cache, by definition, just holds a copy on faster storage of an item that resides elsewhere.

Caching and buffering are distinct functions, but sometimes a region of memory can be used for both purposes. For instance, to preserve copy semantics and to enable efficient scheduling of disk I/O, the operating system uses buffers in main memory to hold disk data. These buffers are also used as a cache, to improve the I/O efficiency for files that are shared by applications or that are being written and reread rapidly.

Spooling and Device Reservation

A spool is a buffer that holds output for a device, such as a printer, that cannot accept interleaved data streams. Although 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 operating system solves this problem by intercepting all output to the printer. Each application's output is spooled to a separate disk file. When an application finishes printing, the spooling system queues the corresponding spool file for output to the printer.

The spooling system copies the queued spool files to the printer one at a time. In some operating systems, spooling is managed by a system daemon process. In the, it is handled by an in-kernel thread.

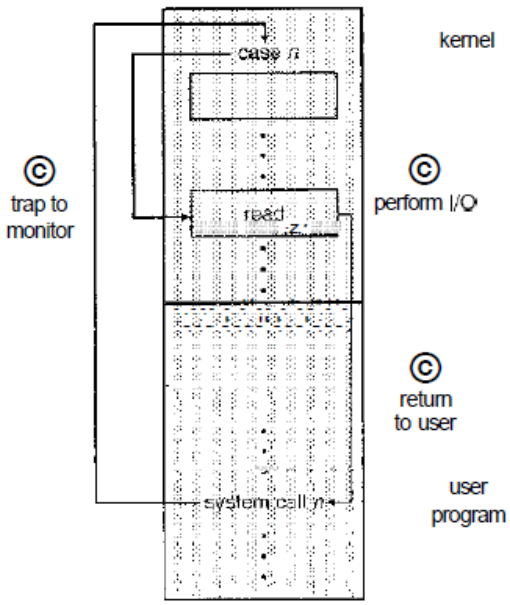
Error Handling

An operating system that uses protected memory can guard against many kinds of hardware and application errors, so that a complete system failure is not the usual result of each minor mechanical glitch. Devices and I/O transfers can fail in many ways, either for transient reasons, as when a network becomes overloaded, or for "permanent" reasons, as when a disk controller becomes defective. Operating systems can often compensate effectively for transient failures. For instance, a disk read() failure results in a readC) retry, and a network send() error results in a resendO, if the protocol so specifies. As a general rule, an I/O system call will return one bit of information about the status of the call, signifying either success or failure. In the UN'IX operating system, an additional integer variable named errno is used to return an error code—one of about a hundred values—indicating the general nature of the failure.

Protection

Errors are closely related to the issue of protection. A user process may accidentally or purposefully attempt to disrupt the normal operation of a system by attempting to issue illegal I/O instructions. We can use various mechanisms to ensure that such disruptions cannot take place in the system. To prevent users from performing illegal I/O, we define all I/O instructions to be privileged instructions. Thus, users cannot issue I/O instructions directly; they must do it through the operating

system. To do I/O, a user program executes a system call to request that the operating system perform I/O on its behalf. The operating system, executing in monitor mode, checks that the request is valid and, if it is, does the I/O requested. The operating system then returns to the user.



5.1.4 Transforming I/O Requests to Hardware Operations

The application refers to the data by a file name. Within a disk, the file system maps from the file name through the file-system directories to obtain the space allocation of the file. For instance, in MS-DOS, the name maps to a number that indicates an entry in the file-access table, and that table entry tells which disk blocks are allocated to the file. In UNIX, the name maps to an inode number, and the corresponding inode contains the space-allocation information.

First, we consider MS-DOS, a relatively simple operating system. The first part of an MS-DOS file name, preceding the colon, is a string that identifies a specific hardware device. For example, *c:* is the first part of every file name on the primary hard disk. The fact that *c:* represents the primary hard disk is built into the operating system; *c:* is mapped to a specific port address through a device table. Because of the colon separator, the device name space is separate from the file-system name space within each device. This separation makes it easy for the operating system to associate extra functionality with each device.

For instance, it is easy to invoke spooling on any files written to the printer.

Modern operating systems obtain significant flexibility from the multiple stages of lookup tables in the path between a request and a physical device controller. The mechanisms that pass requests between applications and drivers are general. Thus, we can introduce new devices and drivers into a computer without recompiling the kernel. In fact, some operating systems have the ability to load device drivers on demand. At boot time, the system first probes the hardware buses to determine what devices are present; it then loads in the necessary drivers, either immediately or when first required by an I/O request.

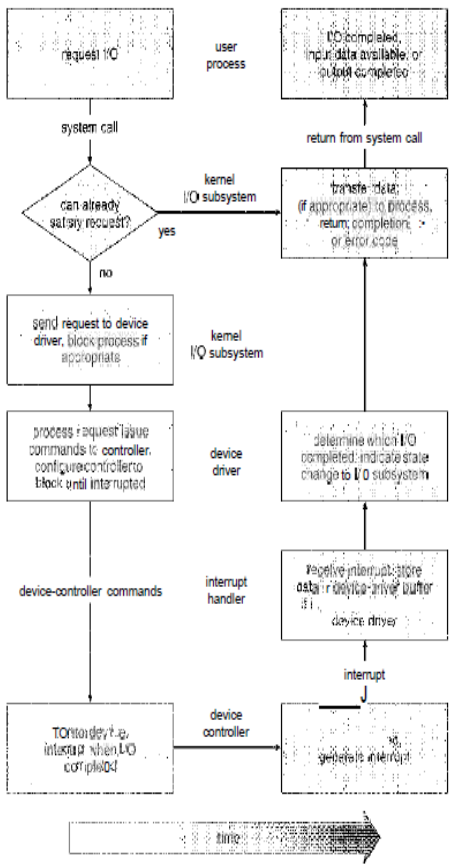
Typical life cycle of a blocking read request

1. A process issues a blocking read () system call to a file descriptor of a file that has been opened previously.
2. The system-call code in the kernel checks the parameters for correctness. In the case of input, if the data are already available in the buffer cache, the data are returned to the process, and the I/O request is completed.
3. Otherwise, a physical I/O must be performed. The process is removed from the run queue and is placed on the wait queue for the device, and the I/O request is scheduled. Eventually, the I/O subsystem sends the request to the device driver. Depending on the operating system, the request is sent via a subroutine call or an in-kernel message.
4. The device driver allocates kernel buffer space to receive the data and schedules the I/O. Eventually, the driver sends commands to the device controller by writing into the device-control registers.
5. The device controller operates the device hardware to perform the data transfer.
6. The driver may poll for status and data, or it may have set up a DMA transfer into kernel memory. We assume that the transfer is managed by a DMA controller, which generates an interrupt when the transfer completes.
7. The correct interrupt handler receives the interrupt via the interrupt vector table, stores any necessary data, signals the device driver, and returns from the interrupt.

8. The device driver receives the signal, determines which I/O request has completed, determines the request's status, and signals the kernel I/O subsystem that the request has been completed.

9. The kernel transfers data or return codes to the address space of the requesting process and moves the process from the wait queue back to the ready queue.

10. Moving the process to the ready queue unblocks the process. When the scheduler assigns the process to the CPU, the process resumes execution at the completion of the system call.



5.2 Protection:

5.2.1 Goals of Protection

Protection was originally conceived as an adjunct to multiprogramming operating systems, so that untrustworthy users might safely share a common logical name space, such as a directory of files, or share a common physical name space, such as memory. Modern protection concepts have evolved to increase the reliability of any complex system that makes use of shared resources.

Protection can improve reliability by detecting latent errors at the interfaces between component subsystems. Early detection of interface errors can often prevent contamination of a healthy subsystem by a malfunctioning subsystem. An unprotected resource cannot defend against use (or misuse) by an unauthorized or incompetent user. A protection-oriented system provides means to distinguish between authorized and unauthorized usage.

The role of protection in a computer system is to provide a mechanism for the enforcement of the policies governing resource use. These policies can be established in a variety of ways. Some are fixed in the design of the system, while others are formulated by the management of a system. Still others are defined by the individual users to protect their own files and programs. A protection system must have the flexibility to enforce a variety of policies.

5.2.2 PRINCIPLES OF PROTECTION:

A key, time-tested guiding principle for protection is the principle of least privilege. It indicates that programs, users, and even systems be given just enough privileges to perform their tasks.

If this key allows the guard into just the public areas that she guards, then misuse of the key will result in minimal damage. An operating system following the principle of least privilege implements its features, programs, system calls, and data structures so that failure or compromise of a component does the minimum damage and allows the minimum damage to be done. Such an operating system also provides system calls and services that allow applications to be written with

fine-grained access controls. It provides mechanisms to enable privileges when they are needed and to disable them when they are not needed.
Managing users with the principle of least privilege entails creating a separate account for each user, with just the privileges that the user needs. An operator who needs to mount tapes and backup files on the system has access to just those commands and files needed to accomplish the job. Some systems implement role-based access control (RBAC) to provide this functionality.

5.2.3 Domain of Protection

A computer system is a collection of processes and objects. By *objects,* we mean both **hardware objects** (such as the CPU, memory segments, printers, disks, and tape drives) and **software objects** (such as files, programs, and semaphores). Each object has a unique name that differentiates it from all other objects in the system, and each can be accessed only through well-defined and meaningful operations. Objects are essentially abstract data types.

The operations that are possible may depend on the object.

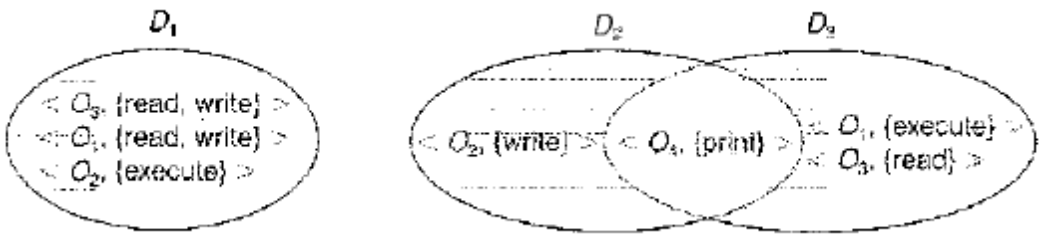
Memory segments can be read and written, whereas a CD-ROM or DVD-ROM can only be read. Tape drives can be read, written, and rewound. Data files can be created, opened, read, written, closed, and deleted; program files can be read, written, executed, and deleted. A process should be allowed to access only those resources for which it has authorization. This second requirement, commonly referred to as the *need-to-knozv* principle, is useful in limiting the amount of damage a faulty process can cause in the system.
when process *p* invokes procedure *A{),* the procedure should be allowed to access only its own variables and the formal parameters passed to it; it should not be able to access all the variables of process *p.* Similarly, consider the case where process *p* invokes a compiler to compile a particular file. The compiler should not be able to access files arbitrarily but should have access only to a well-defined subset of files related to the file to be compiled.

Domain Structure

To facilitate this scheme, a process operates within a protection domain, which specifies the resources that the process may access. Each domain defines a set of objects and the types of operations that may be invoked on each object. The ability to execute an operation on an object is an **access right.** A domain is a collection of access rights, each of which is an ordered pair *<object-iiame, rights-set>.*Domains do not need to be disjoint; they may share access rights.

We have three domains: *Dir D2,* and D3. The access right < *Oi,* (print}> is shared by D? and *D3,* implying that a process executing in either of these two domains can print object O4. Note that a process must be executing in domain *D* to read and write object *O\,* while only processes in domain D3 may execute object *O.*

If the association between processes and domains is fixed, and we want to adhere to the need-to-know principle, then a mechanism must be available to change the content of a domain. The reason stems from the fact that a process may execute in two different phases and may, for example, need read access in one phase and write access in another.



If the association is dynamic, a mechanism is available to allow domain switching, enabling the process to switch from one domain to another. We may also want to allow the content of a domain to be changed. If we cannot change the content of a domain, we can provide the same effect by creating a new domain with the changed content and switching to that new domain when we want to change the domain content.

A domain can be realized in a variety of ways:

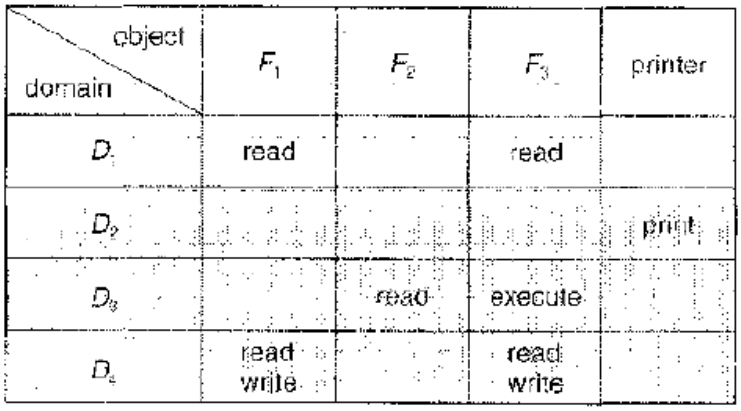
» Each *user* may be a domain. In this case, the set of objects that can be accessed depends on the identity of the user. Domain switching occurs when the user is changed—generally when one user logs out and another user logs in.

• Each *process* may be a domain. In this case, the set of objects that can be accessed depends on the identity of the process. Domain switching occurs when one process sends a message to another process and then waits for a response.

• Each *procedure* may be a domain. In this case, the set of objects that can be accessed corresponds to the local variables defined within the procedure. Domain switching occurs when a procedure call is made.

5.2.4 Access Matrix

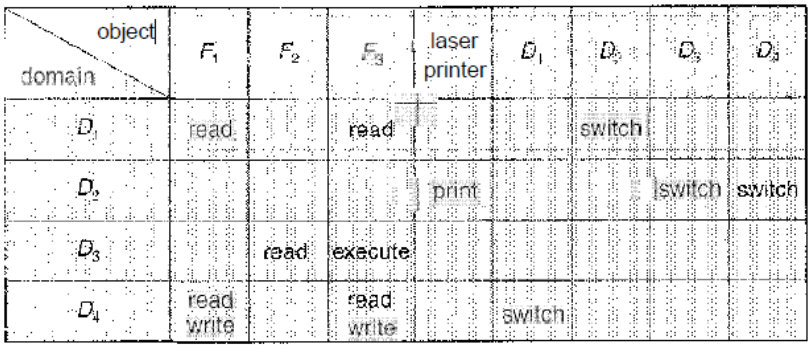
Our model of protection can be viewed abstractly as a matrix, called an **access matrix.** The rows of the access matrix represent domains, and the columns represent objects. Each entry in the matrix consists of a set of access rights. Because the column defines objects explicitly, we can omit the object name from the access right. The entry access(/,/) defines the set of operations that a process executing in domain *Dj* can invoke on object *Or* There are four domains and four objects—three files (F|, F2, F:1) and one laser printer. A process executing in domain *D* can read files Fj and F3. A process executing in domain D4 has the same privileges as one executing in domain *D\;* but in addition, it can also write onto files F| and F3

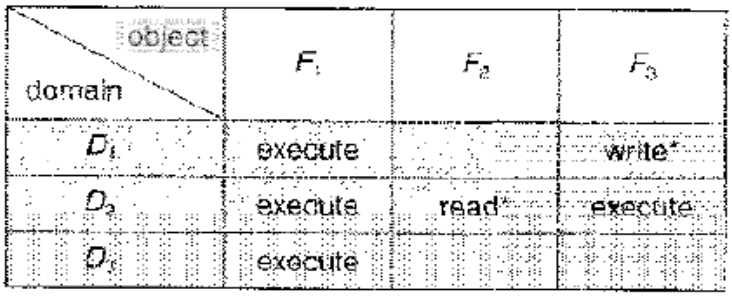


The access-matrix scheme provides us with the mechanism for specifying a variety of policies. The mechanism consists of implementing the access matrix and ensuring that the semantic properties we have outlined indeed, hold. More specifically, we must ensure that a process executing in domain *D,* can access only those objects specified in row \, and then only as allowed by the access-matrix entries.

The access matrix can implement policy decisions concerning protection. The policy decisions involve which rights should be included in the (z',;') the entry. We must also decide the domain in which each process executes. This last policy is usually decided by the operating system.

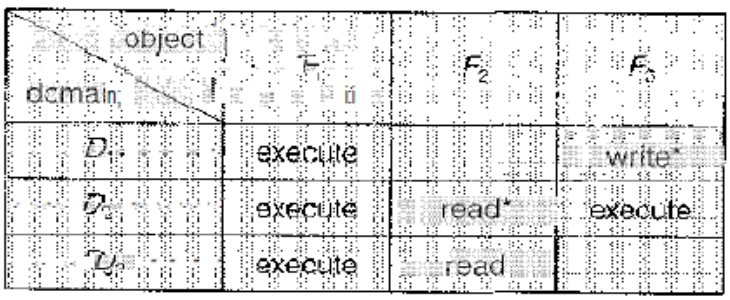
The access matrix provides an appropriate mechanism for defining and implementing strict control for both the static and dynamic association between processes and domains. Then we switch a process from one domain to another, we are executing an operation (switch) on an object (the domain). We can control domain switching by including domains among the objects of the access matrix.





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)

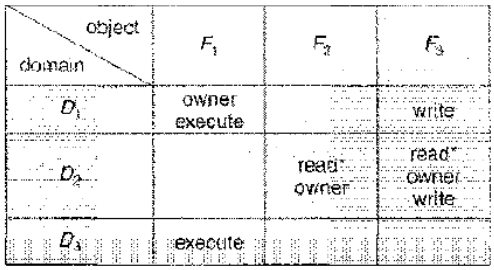
The ability to copy an access right from one domain (or row) of the access matrix to another is denoted by an asterisk (*) appended to the access right. The *copy* right allows the copying of the access right only within the column (that is, for the object) for which the right is defined.

This scheme has two variants:

1. A right is copied from access(/, /) to access(/c,/); it is then removed from access(/,/). This action is a *transfer* of a right, rather than a copy.

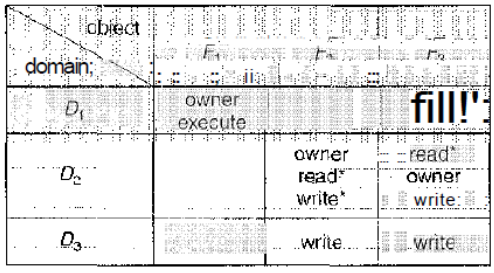
2. Propagation of the *copy* right may be limited. That is, when the right *R** is copied from access(/,y) to access(/t,/), only the right *R* (not *R")* is created. A process executing in domain *Dk* cannot further copy the right *R.*

A system may select only one of these three *copy* rights, or it may provide all three by identifying them as separate rights: *copy, transfer,* and *limited copy.* We also need a mechanism to allow addition of new rights and removal of some rights. The *owner* right controls these operations. If access(/,/) includes the *oivncr* right, then a process executing in domain D, can add and remove any right in any entry in column /'.



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		fill!'
D_2		owner read* write*	read* owner write
D_3		write	write

(b)

The *copy* and *owner* rights allow a process to change the entries in a column. A mechanism is also needed to change the entries in a row. The *control* right is applicable only to domain objects. If access(/,/) includes the *control* right, then a process executing in domain *D.* can remove any access right from row /'.

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			

5.2.5 Implementation of Access Matrix

Global Table

The simplest implementation of the access matrix is a global table consisting of a set of ordered triples *<domain, object, rights-set>.* Whenever an operation *M* is executed on an object O, within domain *D-,,* the global table is searched for a triple

<D,, O;, *Rk>,* with *M* e R/:. If this triple is found, the operation is allowed to continue; otherwise, an exception (or error) condition is raised.
This implementation suffers from several drawbacks. The table is usually large and thus cannot be kept in main memory, so additional I/O is needed. Virtual memory techniques are often used for managing this table.

Access Lists for Objects

Each column in the access matrix can be implemented as an access list for one object, the empty entries can be discarded. The resulting list for each object consists of ordered pairs *<rfomnin, rights-set>,* which define all domains with a nonempty set of access rights for that object.
This approach can be extended easily to define a list plus a *default* set of access rights. When an operation *M* on an object 0/ is attempted in domain *Dj,* we search the access list for object O., looking for an entry *<D,,* R; > with M e Kj. If the entry is found, we allow the operation; if it is not, we check the default set. If M is in the default set, we allow the access.

Capability Lists for Domains

A capability list for a domain is a list of objects together with the operations allowed on those objects. An object is often represented by its physical, name or address, called a capability. To execute operation M on object *0,,* the process executes the operation *M,* specifying the capability (or pointer) for object O/ as a parameter.
Simple possession of the capability means that access is allowed. The capability list is associated with a domain, but it is never directly accessible to a process executing in that domain. Rather, the capability list is itself a protected object, maintained by the operating system and accessed by the user only indirectly. Capability-based protection relies on the fact that the capabilities are never allowed to migrate into any address space directly accessible by a user process (where they could be modified). If all capabilities are secure, the object they protect is also secure against unauthorized access.
Capabilities are usually distinguished from other data in one of two ways:

Each object has a tag to denote its type either as a capability or as accessible data. The tags themselves must not be directly accessible by an application program. Hardware or firmware support may be used to enforce this restriction. Although only 1 bit is necessary to distinguish between capabilities and other objects, more bits are often used. This extension allows all objects to be tagged with their types by the hardware. Thus, the hardware can distinguish integers, floating-point numbers, pointers, Booleans, characters, instructions, capabilities, and uninitialized values by their tags.

• Alternatively, the address space associated with a program can be split into two parts. One part is accessible to the program and contains the program's normal data and instructions. The other part, containing the capability list, is accessible only by the operating system. A segmented memory space (Section 8.6) is useful to support this approach

5.2.6 Access Control:

Each file and directory are assigned an owner, a group, or possibly a list of users, and for each of those entities, access-control information is assigned. A similar function can be added to other aspects of a computer system.

Solaris 10 advances the protection available in the Sun Microsystems operating system by explicitly adding the principle of least privilege via **role-based access control (RBAC).** This facility revolves around privileges. A privilege is the right to execute a system call or to use an option within that system call (such as opening a file with write access). Privileges can be assigned to processes, limiting them to exactly the access they need to perform their work. Privileges and programs can also be assigned to **roles.** Users are assigned roles or can take roles based on passwords to the roles. In this way, a user can take a role that enables a privilege, allowing the user to run a program to accomplish a specific task

5.2.7 Revocation of Access Rights

Rights to objects shared by different users. Various questions about revocation may arise:

• Immediate versus delayed. Does revocation occur immediately/ or is it delayed? If revocation is delayed, can we find out when it will take place?

• **Selective versus** general. When an access right to an object is revoked, does it affect *all* the users who have an access right to that object, or can we specify a select group of users whose access rights should be revoked?

• **Partial versus total.** Can a subset of the rights associated with an object be revoked, or must we revoke all access rights for this object?

• **Temporary versus permanent.** Can access be revoked permanently (that is, the revoked access right will never again be available), or can access be revoked and later be obtained again? Capabilities, however, present a much more difficult revocation problem. Since the capabilities are distributed throughout the system, we must find them before we can revoke them.

Schemes that implement revocation for capabilities include the following:

• **Reacquisition.** Periodically, capabilities are deleted from each domain. If a process wants to use a capability, it may find that that capability has been deleted. The process may then try to reacquire the capability. If access has been revoked, the process will not be able to reacquire the capability.

• **Back-pointers.** A list of pointers is maintained with each object, pointing to all capabilities associated with that object. When revocation is required, we can follow these pointers, changing the capabilities as necessary. This scheme was adopted in the MULTICS system. It is quite general, but its implementation is costly.

• Indirection. The capabilities point indirectly, not directly, to the objects. Each capability points to a unique entry in a global table, which in turn points to the object. We implement revocation by searching the global table for the desired entry and deleting it. Then, when an access is attempted, the capability is found to point to an illegal table entry.

5.2.8 Capability-Based Systems

An Example: Hydra

Hydra is a capability-based protection system that provides considerable flexibility. A fixed set of possible access rights is known to and interpreted by the system. These rights include such basic forms of access as the right to read, write, or

execute a memory segment. In addition, a user (of the protection system) can declare other rights.

Operations on objects are defined procedurally. The procedures that implement such operations are themselves a form of object, and they are accessed indirectly by capabilities. The names of user-defined procedures must be identified to the protection system if it is to deal with objects of the user defined type. When the definition of an object is made known to Hydra, the names of operations on the type become auxiliary rights.

Hydra also provides rights amplification. This scheme allows a procedure to be certified as *trustworthy* to act on a formal parameter of a specified type on behalf of any process that holds a right to execute the procedure.

When a process invokes the operation *P* on an object *A,* however, the capability for access to *A* may be amplified as control passes to the code body of *P.* This amplification may be necessary to allow P the right to access the storage segment representing *A* so as to implement the operation that P defines on the abstract data type. The code body of P may be allowed to read or to write to the segment of *A* directly, even though the calling process cannot. On return from P, the capability for *A* is restored to its original, unamplified state. This case is a typical one in which the rights held by a process for access to a protected segment must change dynamically, depending on the task to be performed. The dynamic adjustment of rights is performed to guarantee consistency of a programmer-defined abstraction. Amplification of rights can be stated explicitly in the declaration of an abstract type to the Hydra operating system.

5.2.9 Language-Based Protection

To the degree that protection is provided in existing computer systems, it is usually achieved through an operating-system kernel, which acts as a security agent to inspect and validate each attempt to access a protected resource.

we must give it hardware support to reduce the cost of each validation or we must accept that the system designer may compromise the goals of protection. Satisfying all these goals is difficult if the flexibility to implement protection policies is restricted by the support mechanisms provided or if protection environments are made larger than necessary to secure greater operational efficiency.

As operating systems have become more complex, and particularly as they have attempted to provide higher-level user interfaces, the goals of protection have become much more refined. The designers of protection systems have drawn heavily on ideas that originated in programming languages and especially on the concepts of abstract data types and objects. Protection systems are now concerned not only with the identity of a resource to which access is attempted but also with the functional nature of that access, in the newest protection systems, concern for the function to be invoked extends beyond a set of system-defined functions, such as standard file-access methods, to include functions that may be user-defined as well.

Compiler-Based Enforcement

Specifying the desired control of access to a shared resource in a system is making a declarative statement about the resource. This kind of statement can be integrated into a language by an extension of its typing facility. When protection is declared along with data typing, the designer of each subsystem can specify its requirements for protection, as well as its need for use of other resources in a system. Such a specification should be given directly as a program is composed, and in the language in which the program itself is stated.

This approach has several significant advantages:

Protection needs are simply declared, rather than programmed as a sequence of calls on procedures of an, operating system.

2. Protection requirements can be stated independently of the facilities provided by a particular operating system.

3. The means for enforcement need not be provided by the designer of a subsystem.

4. A declarative notation is natural because access privileges are closely related to the linguistic concept of data type. A variety of techniques can be provided by a programming-language implementation to enforce protection, but any of these must depend on some degree of support from an underlying machine and its operating system.

5.3 Security

5.3.1 The Security Problem

Security violations (or misuse) of the system can be categorized as intentional (malicious) or accidental. It is easier to protect against accidental misuse than against malicious misuse. For the most part, protection mechanisms are the core of protection from accidents. The following list includes forms of accidental and malicious security violations. We should note that in our discussion of security, we vise the terms *intruder* and *cracker* for those attempting to breach security. In addition, a **threat** is the potential for a security violation, stich as the discovery of a vulnerability, whereas an **attack** is the attempt to break secvirity.

• **Breach of confidentiality.** This type of violation involves unauthorized reading of data (or theft of information). Typically, a breach of confidentiality is the goal of an intruder. Capturing secret data from a system or a data stream, such as credit-card information or identity information for identity theft, can result directly in money for the intruder.

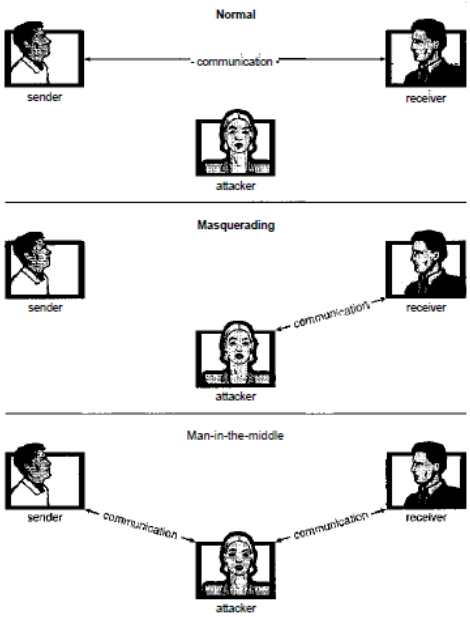
• **Breach of integrity.** This violation involves unauthorized modification of data. Such attacks can, for example, result in passing of liability to an innocent party or modification of the source code of an important commercial application.

• **Breach of availability.** This violation involves unauthorized destruction of data. Some crackers would rather wreak havoc and gain status or bragging rights than gain financially. Web-site defacement is a common example of this type of security breach.

• **Theft of service.** This violation involves unauthorized use of resources. For example, an intruder (or intrusion program) may install a daemon on a system that acts as a file server.

• **Denial of service.** This violation involves preventing legitimate use of the system. **Denial-of-service,** or **DOS,** attacks are sometimes accidental. The original Internet worm turned into a DOS attack when a bug failed to delay its rapid spread. As we have already suggested, absolute protection of the system from malicious abuse is not possible, but the cost to the perpetrator can be made sufficiently high to deter most intruders. In some cases, such as a denial-of service attack, it is

preferable to prevent the attack but sufficient to detect the attack so that countermeasures can be taken.



To protect a system, we must take security measures at four levels: ”

1. Physical. The site or sites containing the computer systems must be physically secured against armed or surreptitious entry by intruders. Both the machine rooms and the terminals or workstations that have access to the machines must be secured.

2. Human. Authorizing users must be done carefully to assure that only appropriate users have access to the system. Even authorized users, however, may be "encouraged" to let others use their access (in exchange for a bribe, for example). They may also be tricked into allowing access via **social engineering.** One type of social-engineering attack is **phishing.** Here, a legitimate-looking e-mail or web page misleads a user into entering confidential information. Another technique is **dumpster diving,** a general term for attempting to gather information in order to gain unauthorized access to the computer (by looking through trash, finding phone books, or finding notes containing passwords, for example). These security problems are management and personnel issues, not problems pertaining to operating systems.

3. **Operating system.** The system must protect itself from accidental or purposeful security breaches. A runaway process could constitute an accidental denial-of-

service attack. A query to a service could reveal passwords. A stack overflow could allow the launching of an unauthorized process. The list of possible breaches is almost endless.

4. Network. Much computer data in modern systems travels over private leased lines, shared lines like the Internet, wireless connections, or dial-up lines. Intercepting these data could be just as harmful as breaking into a computer; and interruption of communications could constitute a remote denial-of-service attack, diminishing users' use of and trust in the system.

5.3.2 Program Threats

Processes, along with the kernel, are the only means of accomplishing work on a computer. Therefore, writing a program that creates a breach of security, or causing a normal process to change its behavior and create a breach, is a common goal of crackers.

Trojan Horse

Many systems have mechanisms for allowing programs written by users to be executed by other users. If these programs are executed in a domain that provides the access rights of the executing user, the other users may misuse these rights. A text-editor program, for example, may include code to search the file to be edited for certain keywords. If any are found, the entire file may be copied to a special area accessible to the creator of the text editor. A code segment that misuses its environment is called a **Trojan horse.** For instance, consider the use of the "." character in a search path. The "." tells the shell to include the current directory in the search. Thus, if a user has "." in her search path, has set her current directory to a friend's directory, and enters the name of a normal system command, the command may be executed from the friend's directory instead. The program would run within the user's domain, allowing the program to do anything that the user is allowed to do, including deleting the user's files, for instance.

A variation of the Trojan horse is a program that emulates a login program. An unsuspecting user starts to log in at a terminal and notices that he has apparently mistyped his password. He tries again and is successful. The emulator stored away the password, printed out a login error message, and exited; the user was then

provided with a genuine login prompt. This type of attack can be defeated by having the operating system print a usage message at the end of an interactive session or by a non-trappable key sequence, such as the control - alt - delete combination used by all modern Windows operating systems.

Trap Door

The designer of a program or system might leave a hole in the software that only she is capable of using. This type of security breach (or **trap door)** was shown in the movie *War Games.* For instance, the code might check for a specific user ID or password, and it might circumvent normal security procedures. Programmers have been arrested for embezzling from banks by including rounding errors in their code and having the occasional half-cent credited to their accounts.

This account crediting can add up to a large amount of money, considering the number of transactions that a large bank executes. A clever trap door could be included in a compiler. The compiler could generate standard object code as well as a trap door, regardless of the source code being compiled.

Logic Bomb

Consider a program that initiates a security incident only under certain circumstances. It would be hard to detect because under normal operations, there would be no security hole. However, when a predefined set of parameters were met, the security hole would be created. This scenario is known as a logic bomb. A programmer, for example, might write code to detect if she is still employed; if that check failed, a daemon could be spawned to allow remote access, or code could be launched to cause damage to the site.

Viruses

Another form of program threat is a **virus.** Viruses are self-replicating and are designed to "infect" other programs. They can wreak havoc in a system by modifying or destroying files and causing system crashes and program malfunctions. A virus is a fragment of code embedded in a legitimate program. As with most penetration attacks, viruses are very specific to architectures, operating systems, and applications. Viruses are a particular problem for users of PCs. UNIX and other multiuser operating systems generally are not susceptible to viruses because the executable programs are protected from writing by the operating

system. Even if a virus does infect such a program, its powers usually are limited because other aspects of the system are protected.

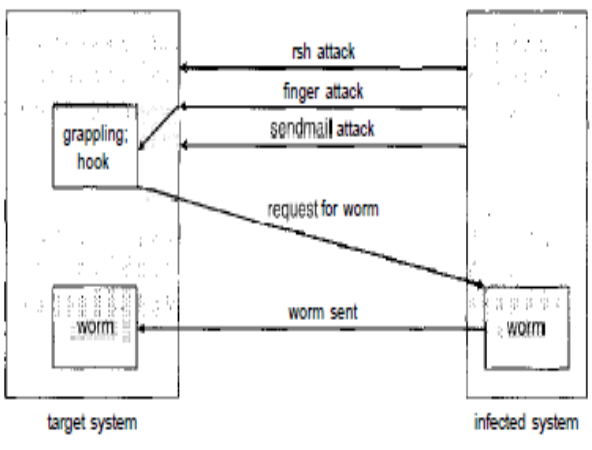
5.3.3 System and Network Threats

Program threats typically use a breakdown in the protection mechanisms of a system to attack programs. In contrast, system and network threats involve the abuse of services and network connections. Sometimes a system and network attack is used to launch a program attack, and vice versa.

System and network threats create a situation in which operating-system resources and user files are misused. Here, we discuss some examples of these threats, including worms, port scanning, and denial-of-service attacks.

Worms

A **worm** is a process that uses the **spawn** mechanism to ravage system performance. The worm spawns copies of itself, using up system resources and perhaps locking out all other processes. On computer networks, worms are particularly potent, since they may reproduce themselves among systems and thus shut down an entire network. Such an event occurred in 1988 to UNIX systems on the Internet, causing millions of dollars of lost system and system administrator time.



The worm was made up of two programs, a grappling hook (also called a bootstrap or vector) program and the main program. Named *11.c,* the grappling hook consisted of 99 lines of C code compiled and run on each machine it accessed. Once established on the computer system under attack, the grappling hook connected to

the machine where it originated and uploaded a copy of the main worm onto the *hooked* system

Port Scanning

Port scanning is not an attack but rather is a means for a cracker to detect a system's vulnerabilities to attack. Port scanning typically is automated, involving a tool that attempts to create a TCP/IP connection to a specific port or a range of ports.

Denial of Service

DOS attacks are aimed not at gaining information or stealing resources but rather at disrupting legitimate use of a system or facility. Most denial-of-service attacks involve systems that the attacker has not penetrated. Indeed, launching an attack that prevents legitimate use is frequently easier than breaking into a machine or facility. Denial-of-service attacks are generally network based. They fall into two categories. The first case is an attack that uses so many facility resources that, in essence, no useful work can be done. For example, a web-site click could download a Java applet that proceeds to vise all available CPU time or to infinitely pop up windows. The second case involves disrupting the network of the facility. There have been several successful denial-of-service attacks of this kind against major web sites. They result from abuse of some of the fundamental functionality of TCP/IP.

5.3.4 Cryptography as a Security Tool

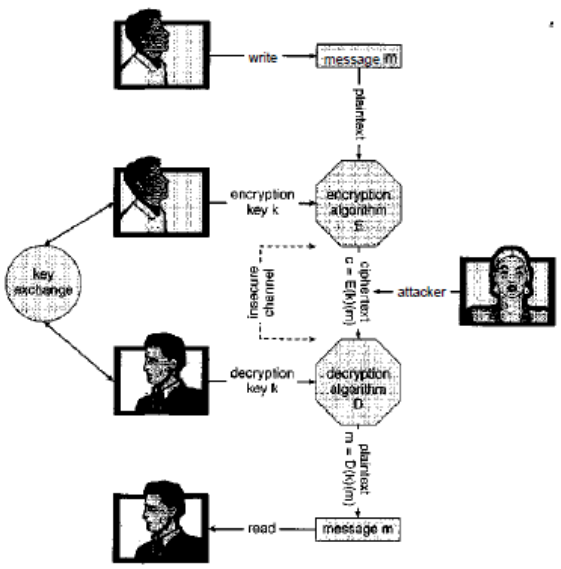
There are many defenses against computer attacks, running the gamut from methodology to technology. The broadest tool available to system designers and users is cryptography networked computer receives bits *fivm the wire* with no immediate and reliable way of determining what machine or application sent those bits. Similarly, the computer sends bits onto the network with no way of knowing who might eventually receive them.

Commonly, network addresses are used to infer the potential senders and receivers of network messages. Network packets arrive with a source address, such as an IP address. And when a computer sends a message, it names the intended receiver by

specifying a destination address. However, for applications where security matters, we are asking for trouble if we assume that the source or destination address of a packet reliably determines who sent or received that packet.

Encryption

Encryption is a means for constraining the possible receivers of a message. An encryption algorithm enables the sender of a message to ensure that only a computer possessing a certain key can read the message. Encryption of messages is an ancient practice, of course, and there have been many encryption algorithms, dating back to before Caesar.



An encryption algorithm consists of the following components:

• A set *K* of keys.

• A set M of messages.

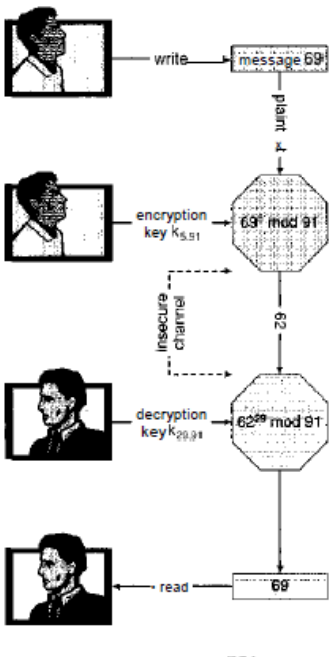
• A set C of ciphertexts.

• A function *E : K ->• (M - •* C). That is, for each *k e K, E (k)* is a function for generating ciphertexts from messages. Both *E* and *E(k)* for any *k* should be efficiently computable functions.

• A function D : *K* -> (C -> *M).* That is, for each *k e K, D{k)* is a function for generating messages from cipher texts. Both D and *D(k)* for any *k* should be efficiently computable functions.

Authentication

We have seen that encryption offers a way of constraining the set of possible receivers of a message. Constraining the set of potential senders of a message is called **authentication.** Authentication is thus complementary to encryption. In fact, sometimes their functions overlap. Consider that an encrypted message can also prove the identity of the sender. For example, if *D{kd, N)(E(ke, N){m))* produces a valid message, then we know that the creator of the message must hold *kc.* Authentication is also useful for proving that a message has not been modified.



An authentication algorithm consists of the following components:

• A set *K* of keys.

• A set *M* of messages.

• A set A of authenticators.

• A function *S : K* —*• (M —>• A).* That is, for each *k* e *K, S(k)* is a function for generating authenticators from messages. Both *S* and *S(k)* for any *k* should be efficiently computable functions.

A function *V* : X - > (M x y l - > {true, false}). That is, for each *k e K, V(k)* is a function for verifying authenticators on messages. Both *V* and *V(k)* for any *k* should be efficiently computable functions.

5.3.5 User Authentication

A major security problem for operating systems is **user authentication.** The protection system depends on the ability to identify the programs and processes currently executing, which in turn depends on the ability to identify each user of the system.

Passwords

The most common approach to authenticating a user identity is the use of **passwords.** When the user identifies herself by user ID or account name, she is asked for a password. If the user-supplied password matches the password stored in the system, the system assumes that the account is being accessed by the owner of that account.

Passwords are often used to protect objects in the computer system, in the absence of more complete protection schemes. They can be considered a special case of either keys or capabilities.

Password Vulnerabilities

Passwords are extremely common because they are easy to understand and use. Unfortunately, passwords can often be guessed, accidentally exposed, sniffed, or illegally transferred from an authorized user to an unauthorized one, as we show next.

There are two common ways to guess a password. One way is for the intruder (either human or program) to know the user or to have information about the user.

The other way is to use brute force, trying enumeration—or all possible combinations of valid password characters (letters, numbers, and punctuation on some systems)—until the password is found.

Encrypted Passwords

One problem with all these approaches is the difficulty of keeping the password secret within the computer. How can the system store a password securely yet allow its use for authentication when the user presents her password? The UNIX system uses encryption to avoid the necessity of keeping its password list secret. Each user has a password. The system contains a function that is extremely difficult—the designers hope impossible—to invert but is simple to compute. That is, given a value *x,* it is easy to compute the function value *f(x).* Given a function value *f(x),* however, it is impossible to compute *x** This function is used to encode all passwords. Only encoded passwords are stored. When a user presents a password, it is encoded and compared against the stored encoded password. Even if the stored encoded password is seen, it cannot be decoded, so the password cannot be determined. Thus, the password file does not need to be kept secret. The function/^) is typically an encryption algorithm that has been designed and tested rigorously.

One-Time Passwords

To avoid the problems of password sniffing and shoulder surfing, a system could use a set of **paired passwords.** When a session begins, the system randomly selects and presents one part of a password pair; the user must supply the other part. In this system, the user is **challenged** and must **respond** with the correct answer to that challenge.

This approach can be generalized to the use of an algorithm as a password. The algorithm might be an integer function, for example. The system selects a random integer and presents it to the user. The user applies the function and replies with the correct result. The system also applies the function. If the two results match, access is allowed.

In this **one-time password** system, the password is different in each instance. Anyone capturing the password from one session and trying to reuse it in another session will fail. One-time passwords are among the only ways to prevent improper authentication due to password exposure.

One-time password systems are implemented in various ways. Commercial implementations, such as SecurlD, use hardware calculators. Most of these calculators are shaped like a credit card, a key-chain dangle, or a USB device; they include a display and may or may not also have a keypad. Some use the current time as the random seed. Others require that the user enters the shared secret, also known as a **personal identification number** or **PIN,** on the keypad. The display then shows the one-time password. The use of both a one-time password generator and a PIN is one form of **two-factor authentication.**

Two different types of components are needed in this case. Two-factor authentication offers far better authentication protection than single-factor authentication.

5.3.6 Computer-Security Classifications

The U.S. Department of Defense Trusted Computer System Evaluation Criteria specify four security classifications in systems: A, B, C, and D. This specification is widely used to determine the security of a facility and to model security solutions, so we explore it here. The lowest-level classification is division D, or minimal protection. Division D includes only one class and is used for systems that have failed to meet the requirements of any of the other security classes.

For instance, MS-DOS and Windows 3.1 are in division D. Division C, the next level of security, provides discretionary protection and accountability of users and their actions through the use of audit capabilities. Division C has two levels: Cl and C2. A Cl-class system incorporates some form of controls that allow users to protect private information and to keep other users from accidentally reading or destroying their data. A Cl environment is one in which cooperating users access data at the same levels of sensitivity. Most versions of UNIX are Cl class.

The sum total of all protection systems within a computer system (hardware, software, firmware) that correctly enforce a security policy is known as a **trusted computer base (TCB).** The TCB of a Cl system controls access between users and files by allowing the user to specify and control sharing of objects by named individuals or defined groups. In addition, the TCB requires that the users identify themselves before they start any activities that the TCB is expected to mediate. This identification is accomplished via a protected mechanism or password; the TCB

protects the authentication data so that they are inaccessible to unauthorized users.

A C2-class system adds an individual-level access control to the requirements of a Cl system. For example, access rights of a file can be specified to the level of a single individual. In addition, the system administrator can selectively audit the actions of any one or more users based on individual identity. The TCB also protects itself from modification of its code or data structures.