Introduction to Operating System

DEFINITION

Operating system acts as interface between the user and the computer hardware. They sit between the user and the hardware of the computer providing an operational environment to the users and application programs. For a user, therefore, a computer is nothing but the operating system running on it. It is an extended machine.
Operating System (or shortly OS) primarily provides services for running applications on a computer system.

User does not interact with the hardware of a computer directly but through the services offered by OS. This is because the language that users employ is different from that of the hardware where as users prefer to use natural language or near natural language for interaction, the hardware uses machine language. Os takes instruction in the form of commands from the user and translates into machine understandable instructions, gets these instructions executed by CPU and translates the result back into user understandable form.

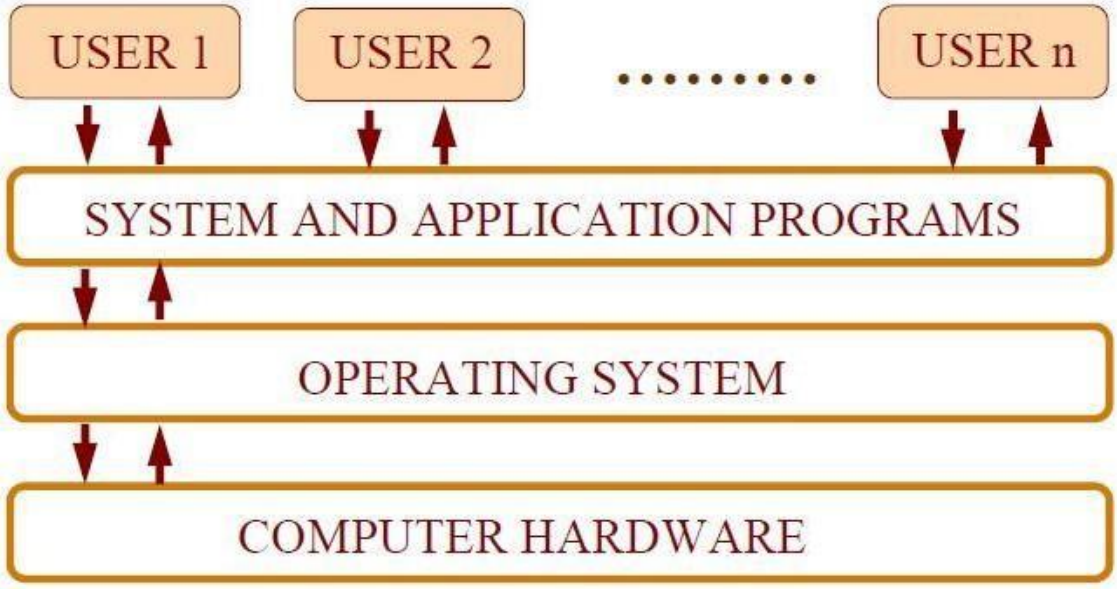
OS is a resource allocate
Manages all resources
Decides between conflicting requests for efficient and fair resource use OS is a control program
Controls execution of programs to prevent errors and improper use of the computer

Need for an OS:

The primary need for the OS arises from the fact that user needs to be provided with services and OS ought to facilitate the provisioning of these services. The central part of a computer system is a processing engine called CPU. A system should make it possible for a user's application to use the processing unit. A user application would need to store information. The OS makes memory available to an application when required. Similarly, user applications need use of input facility to communicate with the application. This is
often in the form of a key board, or a mouse or even a joy stick (if the application is a game for instance).

User and System View:

From the user point of view the primary consideration is always the convenience. It should be easy to use an application. In launching an application, it helps to have an icon which gives a clue which application it is. We have seen some helpful clues for launching a browser, e-mail or even a document preparation application. In other words, the human computer interface which helps to identify an application and its launch is very useful. This hides a lot of details of the more elementary instructions that help in selecting the application. Similarly, if we examine the programs that help us in using input devices like a key board – all the complex details of character reading program are hidden from the user. The same is true when we write a program. For instance, when we use a programming language like C, a printf command helps to generate the desired form of output. The following figure essentially depicts the basic schema of the use of OS from a user stand point. However, when it comes to the view point of a system, the OS needs to ensure that all the system users and applications get to use the facilities that they need.



Also, OS needs to ensure that system resources are utilized efficiently. For instance, there may be many service requests on a Web server. Each user request need to be serviced. Similarly, there may be many programs residing in the main memory. The system need to determine which programs are active and which need to await some form of input or output. Those that need to wait can be suspended temporarily from engaging the processor. This strategy alone enhances the processor throughput. In other words, it is important for an operating system to have a control policy and algorithm to allocate the system resources.

The Evolution of OS:

It would be worthwhile to trace some developments that have happened in the last four to five decades. In the 1960s, the common form of computing facility was a mainframe computer system. The mainframe computer system would be normally housed in a computer center with a controlled environment which was usually an air conditioned area with a clean room like facility. The users used to bring in a deck of punched cards which encoded the list of program instructions.

The mode of operation was as follows:

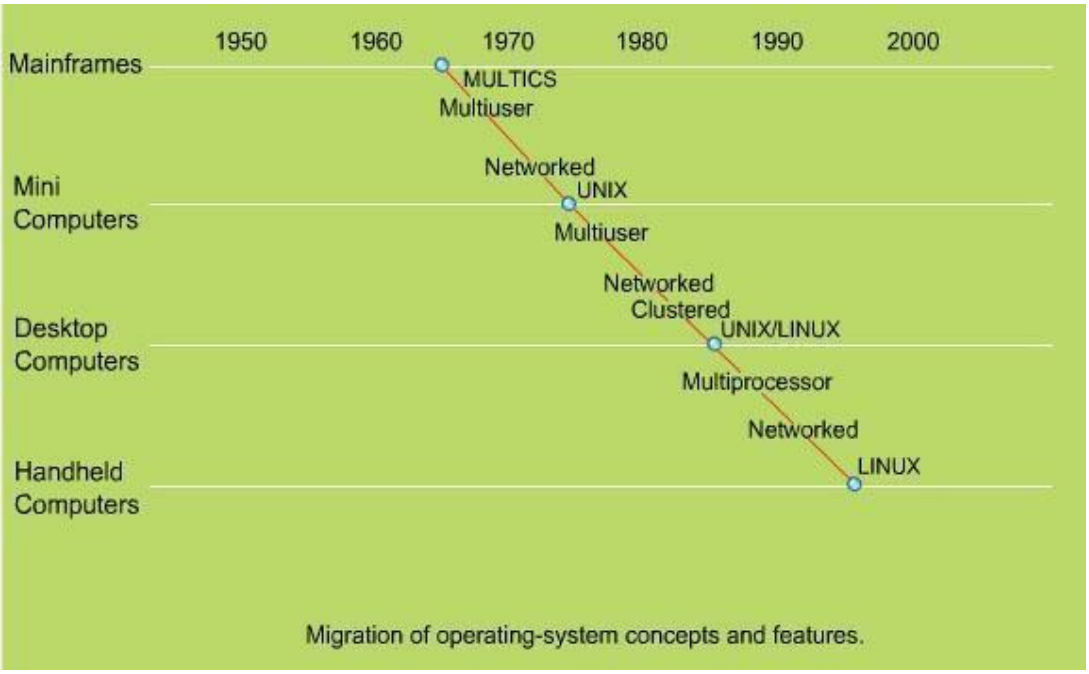
User would prepare a program as a deck of punched cards.

The header cards in the deck were the "job control" cards which would indicate which compiler was to be used (like Fortran / Cobol compilers).

The deck of cards would be handed in to an operator who would collect such jobs from various users.

The operators would invariably group the submitted jobs as Fortran jobs, Cobol jobs etc. In addition, these were classified as "long jobs" that required considerable processing time or short jobs which required a short and limited computational time.

Each set of jobs was considered as a batch and the processing would be done for a batch. Like for instance there may be a batch of short Fortran jobs. The output for each job would be separated and turned over to users in a collection area. This scenario clearly shows that there was no interactivity. Users had no direct control. Also, at any one time only one program would engage the processor. This meant that if there was any input or output in between processing then the processor would wait idling till such time that the I/O is completed. This meant that processor would idling most of the time as processor speeds were orders of magnitude higher than the input or output or even memory units. Clearly, this led to poor utilization of the processor. The systems that utilized the CPU and memory better and with multiple users connected to the systems evolved over a period of time as shown in Table1.1.



Migration of operating-system concepts and features.

At this time we would like to invoke Von - Neumann principle of stored program operation. For a program to be executed it ought to be stored in the memory. In the scheme of things discussed in the previous paragraph, we notice that at any time only one program was kept in the memory and executed. In the decade of 70s this basic mode of operation was altered and system designers contemplated having more than one program resident in the memory. This clearly meant that when one program is awaiting completion of an input or output, another program could, in fact, engage the CPU.

Late 60's and early 70's

Storing multiple executables (at the same time) in the main memory is called multiprogramming. With multiple executables residing in the main memory, the immediate consideration is: we now need a policy to allocate memory and processor time to the resident programs. It is obvious that by utilizing the processor for another process when a process is engaged in input or output the processor utilization and, therefore, its output are higher. Overall, the multiprogramming leads to higher throughput for this reason.

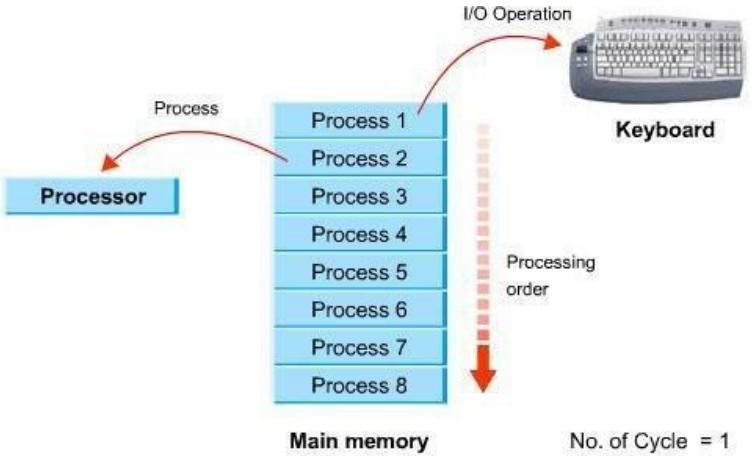


Fig: Multiprogramming

While multiprogramming did lead to enhanced throughput of a system, the systems still essentially operated in batch processing mode.

1980's

In late 70s and early part of the decade of 80s the system designers offered some interactivity with each user having a capability to access system. This is the period when the timeshared systems came on the scene.

Basically, the idea is to give every user an illusion that all the system resources were available to him as his program executed. To strengthen this illusion a clever way was devised by which each user was allocated a slice of time to engage the processor. During the allocated time slice a users' program would be executed. Now imagine if the next turn for the same program comes quickly enough, the user would have an illusion that the system was continuously available to his task. This is what precisely time sharing systems attempted – giving each user a small time slice and returning back quickly enough so that he never feels lack of continuity. In fact, he carries an impression that the system is entirely available to him alone.

Timeshared systems clearly require several design considerations.

These include the following:

- How many programs may reside in the main memory to allow, and also sustain
- timesharing? What should be the time slice allocated to process each program?
- How would one protect a users' program and data from being overwritten by another users' program?

Basically, the design trends that were clearly evident during the decade of 1970-80 were: Achieve as much overlapping as may be feasible between I/O and processing. Bulk storage on disks clearly witnessed a phenomenal growth. This also helped to implement the concept to offer an illusion of extended storage. The concept of "virtual storage" came into the vogue. The virtual storage essentially utilizes these disks to offer enhanced addressable space. The fact that only that part of a program that is currently active need be in the main memory also meant that multi-programming could support many more programs.

In fact this could be further enhanced as follows:

1. Only required active parts of the programs could be swapped in from disks.
2. Suspended programs could be swapped out.

This means that a large number of users can access the system. This was to satisfy the notion that "computing" facility be brought to a user as opposed to the notion that the "user go to compute". The fact that a facility is brought to a user gives the notion of a utility or a service in its true sense. In fact, the PC truly reflects the notion of "computing utility" - it is regarded now as a personal productivity tool.

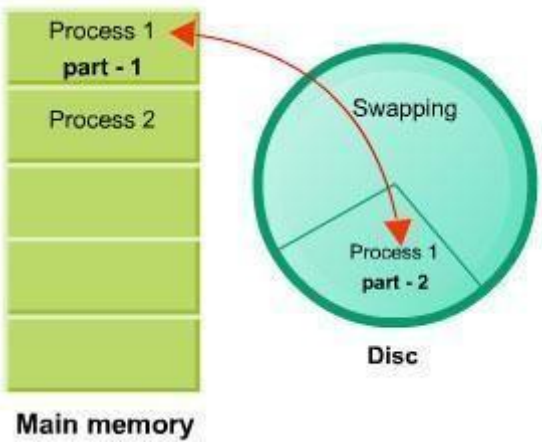


Fig: Swapping of program parts main memory - disc, vice-versa

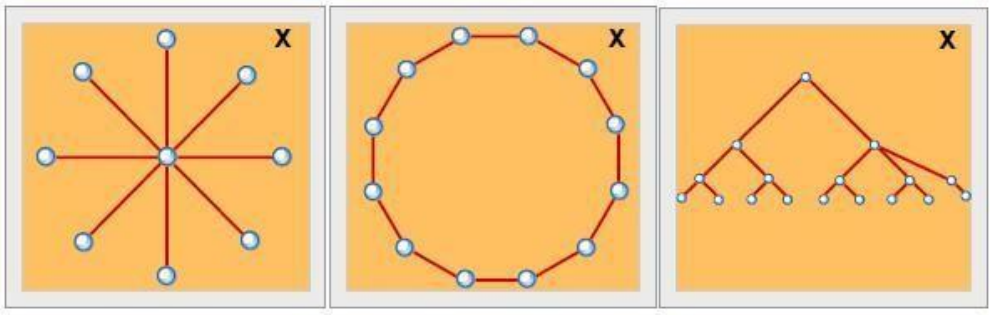
It was in early 1970s Bell Laboratory scientists came up with the now well known OS: Unix. Also, as the microcomputers came on scene in 1980s a forerunner to current DOS was a system called CP/M. The decade of 1980s saw many advances with the promise of networked systems. One notable project

 amongst these was the project Athena at MIT in USA. The project forms the basis to several modern developments. The client-server paradigm was indeed a major fall out. The users could have a common server to the so called X-terminals.

The X windows also provided many widgets to support convenient human computer interfaces. Using X windows it is possible to create multiple windows. In fact each of the windows offers a virtual terminal. In other words it is possible to think about each of these windows as a front-end terminal connection. So it is possible to launch different applications from each of the windows. This is what you experience on modern day PC which also supports such an operating environment.

CP/M based computer

Networking topologies like star, ring and general graphs, as shown in the figure, were being experimented with protocols for communication amongst computers evolved. On the micro-computer front the development was aimed at relieving the processor from handling input output responsibilities. The I/O processing was primarily handled by two mechanisms: one was BIOS and the other was the graphics cards to drive the display. The processor now was relieved from regulating the I/O. This made it possible to utilize the processor more effectively for other processing tasks With the advent of 1990s the computer communication was pretty much the order of the day. With the advent of 1990s the computer networking Technology like Star, Ring and General Graphs, communication was pretty much the order of the day.particular, the TCP/IP suite of network protocols were implemented.

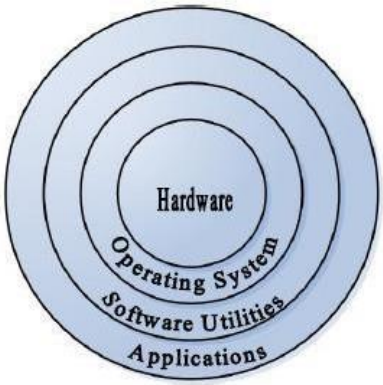


(a) Star (b) Ring (c) Tree

The growth in the networking area also resulted in giving users a capability to establish communication between computers. It was now possible to connect to a remote computer using a telnet protocol. It was also possible to get a file stored in a remote location using a file transfer (FTP) protocol. All such services are broadly called network services

Layered Operating System Structure:

Let's now briefly explore where the OS appears in the context of the software and application.



COMPUTER SYSTEM SATRT UP (BOOTING)

- Power On Self Test (POST) is Done
- Bootstrap program is loaded at power-up or reboot.
- Typically stored in ROM or EPROM, generally known as firmware.
- Initialized all aspects of system
- Loads operating system kernel and starts execution

COMPUTER SYSTEM ORGANIZATION

One or more CPUs, device controllers connect through common bus providing access to shared memory.

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

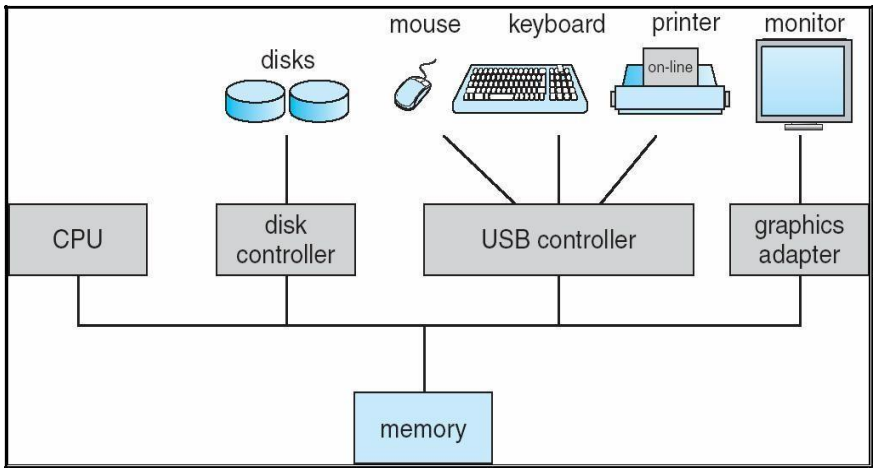


Fig Computer System Organization

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

Operating system goals

- Execute user programs and make solving user problems
- easier. Make the computer system convenient to use.
- Application programs :
    - It 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

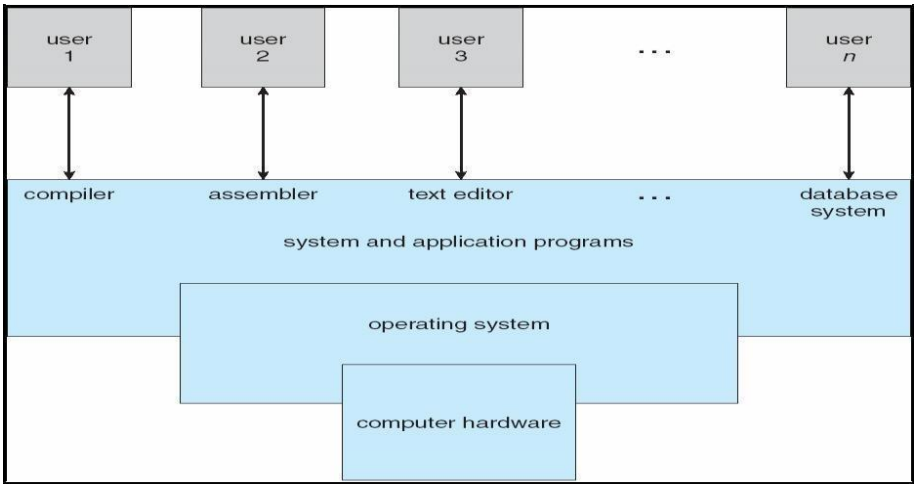


Fig Extended machine view of operating system

All operating system contain the same components whose functionalities are almost the same. For instance, all the operating systems perform the functions of storage management, process management, protection of users from one-another, etc.Operating system in general, performs similar functions but may have distinguishing features. Therefore, they can be classified into different categories on different bases.

Different types of operating system

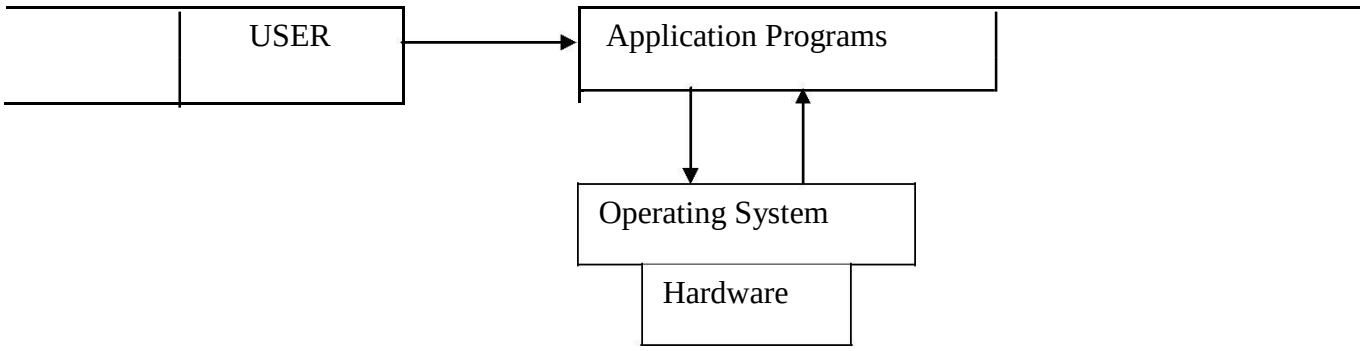
- ***Single user- Single Processing System:-***
  - It has a single processor, runs a single program and interacts with a single user at a time. The OS for this system is very simple to design and implement. Example: - MS-DOS
  - Only one program resides in computer memory and it remains that till it is executed. It is also called uni-program OS. In this OS, the whole, memory space is allocated to one program to memory management's not very difficult task to do. The CPU has to execute only 1 program at a time, so that CPU management also does not have any problem.
  - In a single user OS, a single user can access the computer at a particular time. The computer which are based on this OS have a single processor and able to execute only a single program at a particular time. This system provides all the resources to the users all the time. The single user OS into following categories: -

Single user, single tasking:

- In a single user, single tasking OS, There is a single user to execute a program at a particular system.
- Example – MS-DOS

Single user, multitasking OS:

- In single user, multitasking OS a single user can execute multiple programs.
- Example – A user can program different programs such as making calculations in excel sheet, printing a word document & downloading into the file from internet at the same time.



- *Advantages of single user OS:-*
  - The CPU has to handle only one application program at a time so that process management is easy in this environment.
  - Due to the limited number of programs allocation of memory to the process & allocation of resources to the process becomes any easy task to handle.

- *Disadvantages of single user OS:-*
  - As the OS is handling one application at a time most of the CPU time is wasted, because it has to sit idle most of the time while executing a single program.
  - Resources like memory, CPU are not utilized at the maximum.

e.g: Resident Monitors

- Monitors are the simplest operating systems.
- Single user systems
- Allow user interaction

Batch operating system

The users of batch operating system do not interact with the computer directly. Each user prepares his job on an off-line device like punch cards and submits it to the computer operator. To speed up processing, jobs with similar needs are batched together and run as a group. Thus, the programmers left their programs with the operator. The operator then sorts programs into batches with similar requirements.

The problems with Batch Systems are following.

Lack of interaction between the user and job.

CPU is often idle, because the speeds of the mechanical I/O devices is slower than CPU.

Difficult to provide the desired priority.

Time-sharing operating systems

Time sharing is a technique which enables many people, located at various terminals, to use a particular computer system at the same time. Time-sharing or multitasking is a logical extension of multiprogramming. Processor's time which is shared among multiple users simultaneously is termed as time-sharing. The main difference between Multiprogrammed Batch Systems and Time-Sharing Systems is that in case of Multiprogrammed batch systems, objective is to maximize processor use, whereas in Time-Sharing Systems objective is to minimize response time.

Multiple jobs are executed by the CPU by switching between them, but the switches occur so frequently. Thus, the user can receives an immediate response. For example, in a transaction processing, processor execute each user program in a short burst or quantum of computation. That is if n users are present, each user can get time quantum. When the user submits the command, the response time is in few seconds at most.

Operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time. Computer systems that were designed primarily as batch systems have been modified to time-sharing systems.

Advantages of Timesharing operating systems are following

1. Provide advantage of quick response.
2. Avoids duplication of software.
3. Reduces CPU idle time.

Disadvantages of Timesharing operating systems are following.

Problem of reliability.

Question of security and integrity of user programs and data.

Problem of data communication.

Multi-Programming

As we know that in the Batch Processing System there are multiple jobs Execute by the System. The System first prepare a batch and after that he will Execute all the jobs those are Stored into the Batch. But the Main Problem is that if a process or job requires an Input and Output Operation, then it is not possible and second there will be the wastage of the Time **when we are preparing the batch and the CPU will remain idle at that Time.** But With the help of **Multi programming we can Execute Multiple Programs on the System at a Time** and in the Multi-programming the CPU will never get idle, because with the help of Multi-Programming we can Execute Many Programs on the System and When we are Working with the Program then we can also Submit the Second or Another Program for Running and the CPU will then Execute the Second Program after the completion of the First Program. And in this we can also specify our Input means a user can also interact with the System.

The Multi-programming Operating Systems never use any cards because the Process is entered on the Spot by the user. But the **Operating System also uses the Process of Allocation and De-allocation of the Memory** Means he will provide the Memory Space to all the Running and all the Waiting Processes. There must be the Proper Management of all the Running Jobs.

Distributed operating System

Distributed systems use multiple central processors to serve multiple real time application and multiple users. Data processing jobs are distributed among the processors accordingly to which one can perform each job most efficiently.

The processors communicate with one another through various communication lines (such as high-speed buses or telephone lines). These are referred as loosely coupled systems or distributed systems.

Processors in a distributed system may vary in size and function. These processors are referred as sites, nodes, computers and so on.

The advantages of distributed systems are following.

With resource sharing facility user at one site may be able to use the resources available at another.

- Speedup the exchange of data with one another via electronic mail.
- If one site fails in a distributed system, the remaining sites can potentially continue operating.

Better service to the customers.

Reduction of the load on the host computer.

Reduction of delays in data processing.

Network operating System

Network Operating System runs on a server and and provides server the capability to manage data, users, groups, security, applications, and other networking functions. The primary purpose of the network operating system is to allow shared file and printer access among multiple computers in a network, typically a local area network (LAN), a private network or to other networks. Examples of network operating systems are Microsoft Windows Server 2003, Microsoft Windows Server 2008, UNIX, Linux, Mac OS X, Novell NetWare, and BSD.

The advantages of network operating systems are following.

Centralized servers are highly stable.
Security is server managed.
Upgrades to new technologies and hardwares can be easily integrated into the system.
Remote access to servers is possible from different locations and types of
systems. The disadvantages of network operating systems are following.
High cost of buying and running a server.
Dependency on a central location for most operations.
Regular maintenance and updates are required.

Real Time operating System

Real time system is defines as a data processing system in which the time interval required to process and respond to inputs is so small that it controls the environment. Real time processing is always on line whereas on line system need not be real time. The time taken by the system to respond to an input and display of required updated information is termed as response time. So in this method response time is very less as compared to the online processing.

Real-time systems are used when there are rigid time requirements on the operation of a processor or the flow of data and real-time systems can be used as a control device in a dedicated application. Real-time operating system has well-defined, fixed time constraints otherwise system will fail.For example Scientific experiments, medical imaging systems, industrial control systems, weapon systems, robots, and home-applicance controllers, Air traffic control system etc.

There are two types of real-time operating systems.

HARD REAL-TIME SYSTEMS

Hard real-time systems guarantee that critical tasks complete on time. In hard real-time systems secondary storage is limited or missing with data stored in ROM. In these systems virtual memory is almost never found.

SOFT REAL-TIME SYSTEMS

Soft real time systems are less restrictive. Critical real-time task gets priority over other tasks and retains the priority until it completes. Soft real-time systems have limited utility than hard real-time systems.For example, Multimedia, virtual reality, Advanced Scientific Projects like undersea exploration and planetary rovers etc.

Functions of Operating Systems

Operating systems perform the following important functions:

1. **Processor Management:** It means assigning processor to different tasks which has to be performed by the computer system.
2. **Memory Management:** It means allocation of main memory and secondary storage areas to the system programs, as well as user programs and data.
3. **Input and Output Device Management:** It means co-ordination and assignment of the different output and input devices while one or more programs are being executed.
4. **File System Management:** Operating system is also responsible for maintenance of a file system, in which the users are allowed to create, delete and move files.
5. **Process Management Functions:**
   - ***Traffic Controller:*** It constantly checks processor and status of processes.
   - ***Job Scheduler:*** Selects jobs from job queue submitted to the system for execution.
   - ***Process Scheduler:*** decides when process is to be executed in case of multiprogramming.

     Dispatcher: Allocates the processor for particular process which is chosen by process scheduler.

     I/O Traffic Controller: It constantly keeps the track of I/O devices, channels and control processors.
   - ***I/O scheduler:*** Decides which device is to be allocated to which process.
   - ***File System:*** It is a collection of functions which are used to find status, usage and location of a file. From this all informations can be easily find which are stored on a disk in form of files.

Responsibilities of Operating Systems:

1. Perform basic tasks, such as recognizing input from the keyboard, sending output to the display screen, keeping track of files and directories on the disk and controlling peripheral devices such as disk drives and printers.
2. Ensure that different programs and users running at the same time do not interfere with each other.
3. Provide a software platform on top of which other programs (i.e., application software) can run.

The first two responsibilities address the need for managing the computer hardware and the application programs that use the hardware. The third responsibility focuses on providing an interface between application software and hardware so that application software can be efficiently developed. Since the operating system is already responsible for managing the hardware, it should provide a programming interface for application developers.

Services Provided by Operating Systems:

An Operating System provides services to both the users and to the programs.

1. **It provides programs, an environment to execute.**

2. **It provides users, services to execute the programs in a convenient manner.**
3. **Following are few common services provided by operating systems.**
4. **Program execution**
5. **I/O operations**
6. **File System manipulation**
7. **Communication**
8. **Error Detection**
9. **Resource Allocation**
10. **Protection**

1. Program execution

Operating system handles many kinds of activities from user programs to system programs like printer spooler, name servers, file server etc. Each of these activities is encapsulated as a process.

A process includes the complete execution context (code to execute, data to manipulate, registers, OS resources in use). Following are the major activities of an operating system with respect to program management.

- Loads a program into memory.
- Executes the program.
- Handles program's execution.
- Provides a mechanism for process synchronization.
  Provides a mechanism for process communication.
  Provides a mechanism for deadlock handling.

2. I/O Operation

I/O subsystem comprised of I/O devices and their corresponding driver software. Drivers hides the peculiarities of specific hardware devices from the user as the device driver knows the peculiarities of the specific device.

Operating System manages the communication between user and device drivers. Following are the major activities of an operating system with respect to I/O Operation.

I/O operation means read or write operation with any file or any specific I/O device.

- Program may require any I/O device while running.
- Operating system provides the access to the required I/O device when required.

3. File system manipulation

A file represents a collection of related information. Computer can store files on the disk (secondary storage), for long term storage purpose. Few examples of storage media are magnetic tape, magnetic disk and optical disk drives like CD, DVD. Each of these media has its own properties like speed, capacity, data transfer rate and data access methods.

A file system is normally organized into directories for easy navigation and usage. These directories may contain files and other directions. Following are the major activities of an operating system with respect to file management.

Program needs to read a file or write a file.
The operating system gives the permission to the program for operation on file.
Permission varies from read-only, read-write, denied and so on.

- Operating System provides an interface to the user to create/delete files.
- Operating System provides an interface to the user to create/delete directories.
- Operating System provides an interface to create the backup of file system.

4. Communication

In case of distributed systems which are a collection of processors that do not share memory, peripheral devices, or a clock, operating system manages communications between processes. Multiple processes with one another through communication lines in the network.

OS handles routing and connection strategies, and the problems of contention and security. Following are the major activities of an operating system with respect to communication.

Two processes often require data to be transferred between them.
The both processes can be on the one computer or on different computer but are connected through computer network.
Communication may be implemented by two methods either by Shared Memory or by Message Passing.

5. Error handling

Error can occur anytime and anywhere. Error may occur in CPU, in I/O devices or in the memory hardware. Following are the major activities of an operating system with respect to error handling.

- OS constantly remains aware of possible errors.
- OS takes the appropriate action to ensure correct and consistent computing.

6. Resource Management

In case of multi-user or multi-tasking environment, resources such as main memory, CPU cycles and files storage are to be allocated to each user or job. Following are the major activities of an operating system with respect to resource management.

OS manages all kind of resources using schedulers.
CPU scheduling algorithms are used for better utilization of CPU.

7. Protection

Considering a computer system having multiple users the concurrent execution of multiple processes, then the various processes must be protected from each another's activities.

Protection refers to mechanism or a way to control the access of programs, processes, or users to the resources defined by computer systems. Following are the major activities of an operating system with respect to protection.

OS ensures that all access to system resources is controlled.
OS ensures that external I/O devices are protected from invalid access attempts.
OS provides authentication feature for each user by means of a password.

The owners of information stored in a multi-user 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.

OPERATING SYSTEM – TASKS:

Following are few of very important tasks that Operating System handles

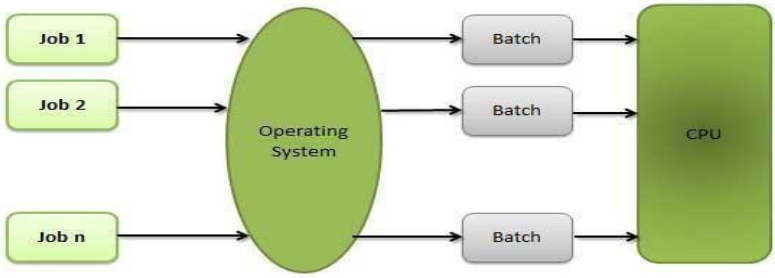
1. Batch processing

Batch processing is a technique in which Operating System collects one programs and data together in a batch before processing starts. Operating system does the following activities related to batch processing.

OS defines a job which has predefined sequence of commands, programs and data as a single unit.

- OS keeps a number a jobs in memory and executes them without any manual information.
- Jobs are processed in the order of submission i.e first come first served fashion.

When job completes its execution, its memory is released and the output for the job gets copied into an output spool for later printing or processing.



Advantages

Batch processing takes much of the work of the operator to the computer.

Increased performance as a new job gets started as soon as the previous job finished without any manual intervention.

Disadvantages

Difficult to debug program.

A job could enter an infinite loop.

Due to lack of protection scheme, one batch job can affect pending jobs.

2. Multitasking

Multitasking refers to term where multiple jobs are executed by the CPU simultaneously by switching between them.Switches occur so frequently that the users may interact with each program while it is running. Operating system does the following activities related to multitasking.

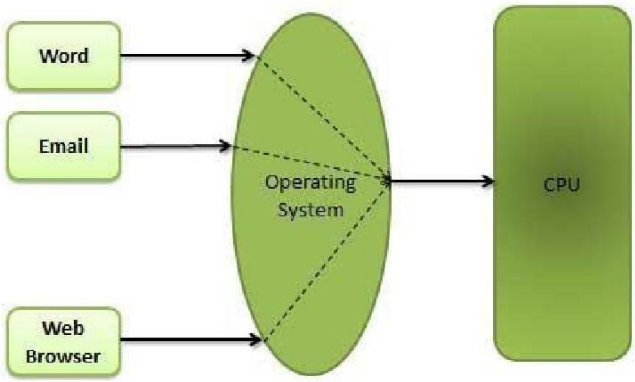
The user gives instructions to the operating system or to a program directly, and receives an immediate response.

Operating System handles multitasking in the way that it can handle multiple operations / executes multiple programs at a time.

Multitasking Operating Systems are also known as Time-sharing systems.

These Operating Systems were developed to provide interactive use of a computer system at a reasonable cost.

A time-shared operating system uses concept of CPU scheduling and multiprogramming to provide each user with a small portion of a time-shared CPU. Each user has at least one separate program in memory.



- A program that is loaded into memory and is executing is commonly referred to as a process.

  When a process executes, it typically executes for only a very short time before it either finishes or needs to perform I/O.

  Since interactive I/O typically runs at people speeds, it may take a long time to completed. During this time a CPU can be utilized by another process.

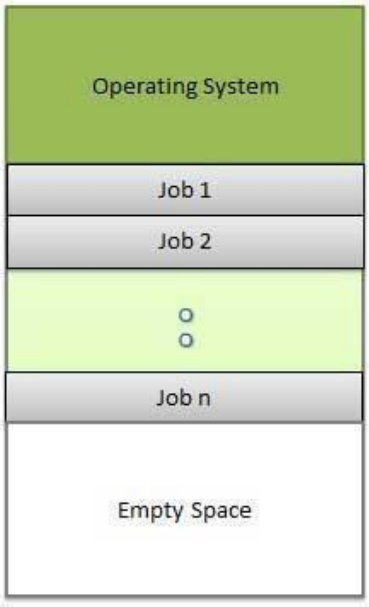
  Operating system allows the users to share the computer simultaneously. Since each action or command in a time-shared system tends to be short, only a little CPU time is needed for each user.

  As the system switches CPU rapidly from one user/program to the next, each user is given the impression that he/she has his/her own CPU, whereas actually one CPU is being shared among many users.

3. Multiprogramming

When two or more programs are residing in memory at the same time, then sharing the processor is referred to the multiprogramming. Multiprogramming assumes a single shared processor. Multiprogramming increases CPU utilization by organizing jobs so that the CPU always has one to execute.

Following figure shows the memory layout for a multiprogramming system.



Operating system does the following activities related to multiprogramming. The

- operating system keeps several jobs in memory at a time. This
- set of jobs is a subset of the jobs kept in the job pool.
- The operating system picks and begins to execute one of the job in the memory.

Multiprogramming operating system monitors the state of all active programs and system resources using memory management programs to ensures that the CPU is never idle unless there are no jobs

ADVANTAGES

High and efficient CPU utilization.

User feels that many programs are allotted CPU almost simultaneously.

- DISADVANTAGES
- CPU scheduling is required.
- To accommodate many jobs in memory, memory management is required.
- *Interactivity*
- Interactivity refers that a User is capable to interact with computer system. Operating system does the following activities related to interactivity.

  OS provides user an interface to interact with system.
- OS managers input devices to take inputs from the user. For example, keyboard. OS
- manages output devices to show outputs to the user. For example, Monitor. OS
- Response time needs to be short since the user submits and waits for the result.
-

4. **Real Time System**

Real time systems represents are usually dedicated, embedded systems. Operating system does the following activities related to real time system activity.

In such systems, Operating Systems typically read from and react to sensor data. The Operating system must guarantee response to events within fixed periods of time to ensure correct performance.

5. **Distributed Environment**

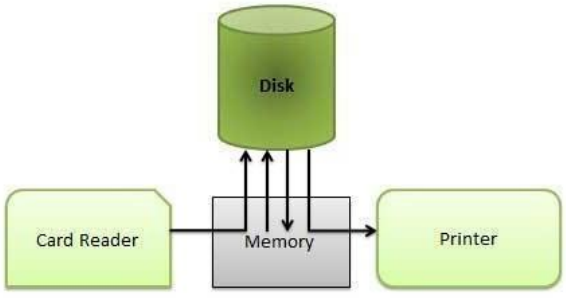
Distributed environment refers to multiple independent CPUs or processors in a computer system. Operating system does the following activities related to distributed environment.

- OS Distributes computation logics among several physical processors.
- The processors do not share memory or a clock.
- Instead, each processor has its own local memory.
- OS manages the communications between the processors. They communicate with each other through various communication lines.
- *Spooling*
- Spooling is an acronym for simultaneous peripheral operations on line. Spooling refers to putting data of various I/O jobs in a buffer. This buffer is a special area in memory or hard disk which is accessible to I/O devices. Operating system does the following activites related to distributed environment.

  OS handles I/O device data spooling as devices have different data access rates.

  OS maintains the spooling buffer which provides a waiting station where data can rest while the slower device catches up.

  OS maintains parallel computation because of spooling process as a computer can perform I/O in parallel fashin. It becomes possible to have the computer read data from a tape, write data to disk and to write out to a tape printer while it is doing its computing task.



Advantages

The spooling operation uses a disk as a very large buffer.
Spooling is capable of overlapping I/O operation for one job with processor operations for another job.

OPERATING SYSTEM STRUCTURE:

Kernels

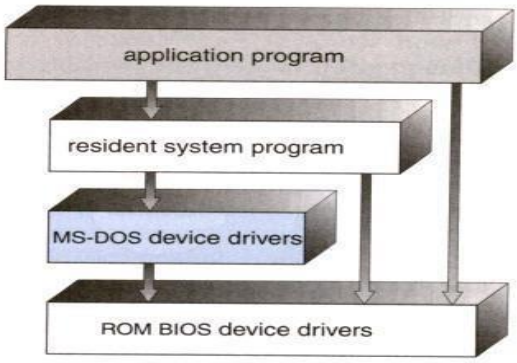
The **kernel** is the core code of an operating system. It includes the lowest-level code, and it provides the basic abstractions that all other code requires.

Modern processors can execute in various privilege modes. At the very least, a processor must allow for **user mode** and **supervisor mode** (another term used is **kernel mode**). The modes differ in the privileges allowed. Essentially, supervisor mode can do anything the processor is capable of, while user mode, or other lower-privilege modes, are limited to certain address spaces and operations.

As one might expect, the basic idea is that an OS kernel executes in kernel mode, and other code, including application code, executes in user mode. Different OS designs differ in where this user-kernel boundary is drawn: what are the responsibilities of the kernel, and what code lies in it.

The portion of an OS that runs in user mode is often called its **userland**. (This term can also be used to refer to all code that runs in user mode.)

Monolithic Architecture:



MS-DOS Structure

The first, and, historically, most common, kernel design is that of a **monolithic kernel**. In this design, essentially all non-application code resides in the kernel and executes at high privilege: interrupt handlers, system call implementation, memory management, process scheduling, device drivers. The kernel is a single large executable, and all of this code is linked together.

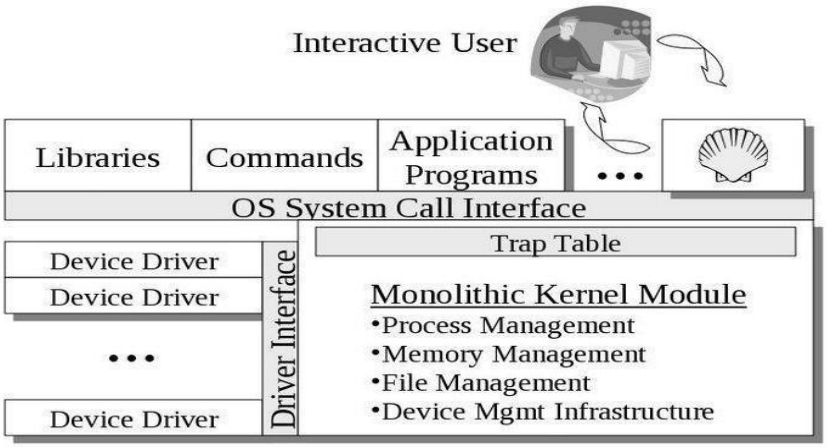
Today, the primary example of a monolithic design is the Linux kernel. This kernel is the core of the various OSs known as either "GNU/Linux" or (somewhat incorrectly) just "Linux". It is also used in the Android OS.

Because monolithic kernels have a large amount of code linked together, and executing at high privilege, they can suffer in terms of maintainability. Another problem is that, since so much code executes at high privilege, bugs can have severe effects. Thus, security and robustness can also be issues.

- Operating systems such as MS-DOS and the original UNIX did not have well-defined structures.
- There is no *CPU Execution Mode* (user and kernel), and so errors in applications can cause the whole system to crash

  Functionality of the OS is invoked with simple function calls within the kernel, which is one large program.

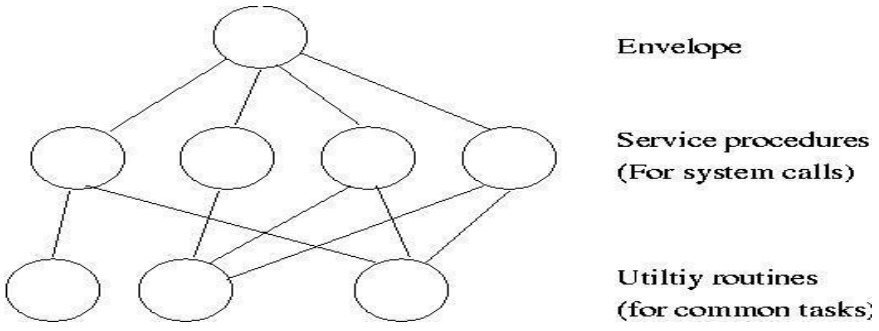
  Device drivers are loaded into the running kernel and become part of the kernel.



A monolithic kernel, such as Linux and other Unix systems.

Layered Approach:

The disadvantages of a monolithic kernel can be mitigated somewhat using a **layered design**. Such a kernel is divided into logical layers, each of which provides abstractions for the layers above it. For example, the lowest layer might handle memory management and process scheduling; it would provide the abstractions of *process* and*address space* to all the layers above it.This approach breaks up the operating system into different layers.



Some systems have more layers and are more strictly structured. An early layered system was "THE" operating system by Dijkstra. The layers were.

1. The operator
2. User programs
3. I/O mgt
4. Operator-process communication
5. Memory and drum management

The layering was done by convention, i.e. there was no enforcement by hardware and the entire OS is linked together as one program. This is true of many modern OS systems as well (e.g., linux).

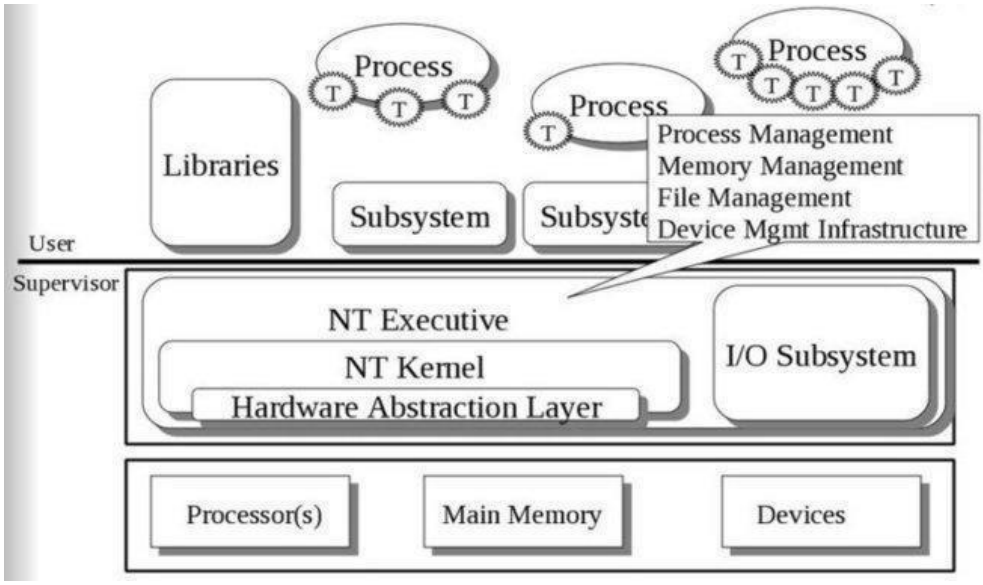
The multics system was layered in a more formal manner. The hardware provided several protection layers and the OS used them. That is, arbitrary code could not jump to or access data in a more protected layer.

This allows implementers to change the inner workings, and increases modularity.

As long as the external interface of the routines don't change, developers have more freedom to change the inner workings of the routines.

With the layered approach, the bottom layer is the hardware, while the highest layer is the user interface.

- The main *advantage* is simplicity of construction and debugging.
- The main *difficulty* is defining the various layers.
- The main *disadvantage* is that the OS tends to be less efficient than other implementations.



Microkernel Architecture:

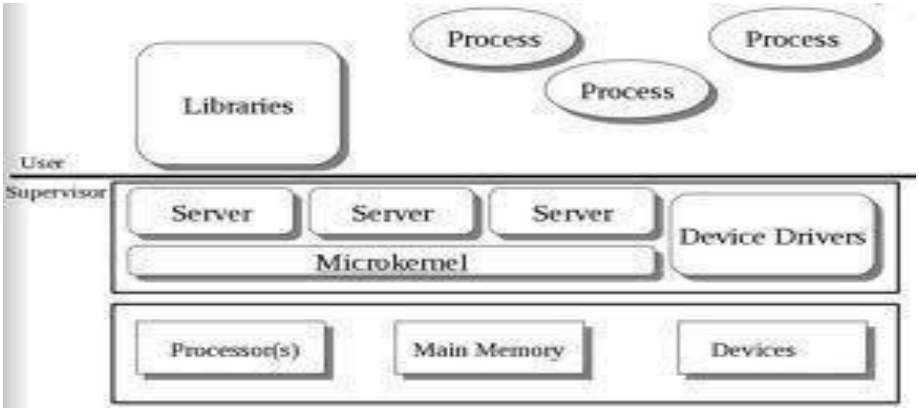
"Lean kernel" or expanded kernel architecture moves as much service functionality as possible from the kernel into "user" space.

Kernel uses minimal process and memory management;

It uses the Inter Process Communication (IPC) i.e. message passing mechanism for communication between client programs and other system processes running under user space.

The basic structure is modularized in which all non essential components are removed from the kernel and are implemented in system or user level programs. As a result the kernel becomes smaller and hence the name " micro kernel".

Example: Mach kernel, used e.g. in Tru64 Unix or Mac OS-X, Sysbian OS.



Advantages:

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

Disadvantages:

Performance overhead of user space to kernel space communication

Virtual machines:

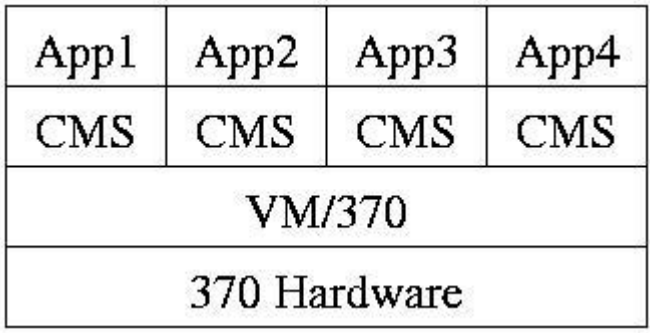
Usually in this approach, system programs are treated as an application programs and are at higher level than H/W routines, the application program may view all these layers under them in the hierarchy though the H/w routines are the part of machine itself. This layered approach is a concept of Virtual Machine.

e.g. VMOS for IBM 370 & IBM mainframe

Components of Virtual Machine:

- Control Program (Controls physical machine)
- Conversational Monitor System (Controls Virtual Machine, Single user Interactive OS, runs at the top of control program and interacts with useruser and any application program running at that virtual machine)

The Virtual Machine is an exact copy of H/W which provides support for the kernel or user mode, I/O instructions, Interrupts etc.



Use a ``hypervisor" (beyond supervisor, i.e. beyond a normal OS) to switch between multiple *Operating Systems*

Each App/CMS runs on a *virtual*
370 CMS is a *single user* OS
A system call in an App traps to the corresponding CMS
CMS believes it is running on the machine so issues I/O instructions but ...
... I/O instructions in CMS trap to VM/370

Advantages:

- Each VM is completely isolated from other, so no security problem.
- Allows system development to be done w/o disrupting normal system operations.

Disadvantages:

Difficult to implements
Much work is required to provide an exact duplicate of underlying machine.

Operating System and System Calls

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.

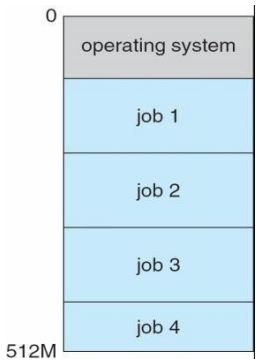
Concurrency by multiplexing the CPUs among the processes / threads.

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 deal locating memory space as needed

Memory Layout for Multiprogrammed System



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, and 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 directories.
- Mapping files onto secondary storage.

Backup files onto stable (non-volatile) storage media.

OBJECTIVES of OS

- To describe the services an operating system provides to users, processes, and other systems
- To discuss the various ways of structuring an operating system
- To explain how operating systems are installed and customized and how they boot

SYSTEM CALL

System calls are the way a user (i.e. a program) directly interfaces with the OS.

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

TYPE OF SYSTEM CALL

- management Device
- management
- Information maintenance
- Communications

SYSTEM BOOT

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

FILE MANAGEMENT

Introduction

Motivation for files:

Only a limited amount of information could be kept in the virtual address space of a process. There is a need for non-volatile storage and storage that will survive the life of a process using it. Need a convenient way of sharing large amounts of data among processes concurrently.

The solution: store information on disks (or other media) in ***units*** called files … essentially a named set/collection of data stored on some media.

Basic requirement for a file:

Processes must be able to create, update, and destroy files, and the file must not require the process to exist.

A file should disappear only when its ***owner*** explicitly removes it.

Files should be managed by the operating system: how they are structured, named, accessed, used, protected, and implemented. This is the File System component of an operating system.

A File from the User Point of view (user interface)

Files are abstract objects – they provide a way to store information and retrieve it later. They should shield the user from the details on how they are stored on a disk, and how a disk works.

File naming:

There are no standard conventions for all operating systems. All systems minimally allow up to 8 character strings for a name. Names may or may not be case sensitive.

Most systems support some form of file extensions:

"filename.extension" Examples:

rat.txt a text file, OS and application supported

rat.exe a binary executable file –DOS/Windows - OS supported
rat.c a C source file – application supported (compiler)
rat.doc a MS Word file – application supported

Extensions sometimes are just conventions and are not enforced, and other times are required, for example a C compiler requires a —.c| extension.

Whatever the objectives of the applications, it involves the generation and use of information. As you know the input of the application is by means of a file, and in virtually all Applications, output is saved in a file for long-term storage. You should be aware of the objectives such as accessing of files, saving the information and maintaining the integrity of the contents, virtually all computer systems provide file management services.

Hence a file management system needs special services from the operating system.

Files

- The following are the commonly discussed with respect to files:

Field: Basic element of data. Its length and data type characterizes it. They can be of fixed or variable length.

Record: Collection of related fields. Depending on the design, records may be of fixed or variable length. Ex: In sequential file organization the records are of fixed length where as in Line sequential file organization the records are of variable length.

File: Collection of similar records and is referenced by name.

- They have unique file names. Restrictions on access control usually apply at the file level. But in some systems, such controls are enforced at the record or even at the field level also.

Database: Collection of related data. The essential aspects of a database are that the relationships that exists among elements of data. The database itself consists of one or more types of files.

Files are managed by the operating system. How they are structured, named, accessed, used, protected and implemented are the major issues in operating system design. As a whole, the part of the operating system deals with files is known as the file system. The linked lists and bitmaps are used to keep track of free storage and how many sectors there are in a logical block are important to the designers of the file system.

File System components

- Device Drivers:
- Communicates directly with peripherals devices (disks, tapes, etc)
- Responsible for starting physical I/O operations on the device
- Processes the completion of an I/O request

Schedule access to the device in order to optimize performance Basic File System:

Uses the specific device driver

Deals with blocks of data that are exchanged with the physical device Concerned with the placement of blocks on the disk

Concerned with buffering blocks in main memory

- Logical File System
- Responsible for providing the previously discussed interface to the user
- including: File access
- Directory operations
- Security and protection.

File Types : Many operating systems support several types of files. Unix and Windows, have regular files and directories. Regular files are the ones that contain user information generally in ASCII form. Directories are system files for maintaining the Structure of the file system. Character special files are related to input/output and used to model serial I/O devices such as terminals, printers and networks. Block special files are used to Model disks.

File Access: Early operating systems provided only one kind of file access: sequential access. In these system, a process could read all the bytes or records in a file in order, starting at the beginning, but could not skip around and read them our of order. Sequential files were convenient when the storage medium was magnetic tape, rather than disk. Files whose bytes or records can be read in any order are called random access files. Two methods are used form specifying where to start reading. In the first one, every read operation gives the position in the file to start reading at. In the second one, a special operation, seek, is provided to set the current position. This allows the system to use different storage techniques for the two classes. Where as in modern operating systems all the files are automatically random access.

File Attributes:

Every file has a name and its data. In addition all operating systems associate other information with each file such as the date and time the file was created and the file‘s size. The list of attributes varies considerably from system to system. Attributes such as protection, password, creator and owner tell who may access it and who may not. The flags are bits or short fields that control or enable some specific property. The record length, key, position and key length fields are only present in files whose records can be looked up using a key. The various times keep track of when the file was created, most recently accessed and most recently modified. These are useful for a variety of purpose. The current size tells how big the file is at present.

1. **File name**
2. **File Type**
3. **Location**
4. **Size**
5. **Current position:** This is a pointer to current ‗raed or write‘ position in the file.
6. **Protection:** Access-control information controls that can do reading, writing, executing, and so on.
7. **Usage Count:** This value indicates the number of processes that are currently using (have opened) this file.
8. **Time, date and process identification:** This information may be kept for creation, last modification and last use.

File Extensions :

Files are an abstraction mechanism. The main characteristic feature of abstraction mechanism is the way the objects being managed are name. The exact rules for the file naming vary from system to system, but all current operating system allows strings of one to eight letters as legal file names.

Many file systems support names as long as 255 characters with a distinguish in upper and lower case. Many operating systems support two-part file names, with the two parts separated by a period. The first part is called primary file name and the second part is called secondary or extension file name.

Documents	MS Office	Media	Pictures	System
.doc .docx .txt .rtf .wpd .wks	.xls .xlsx .ppt .pptx	.mp3 .mpeg .wma .dvi .fla .swf	.gif .jpg .jpeg .bmp .png	.exe .dll .ini

File Structure:

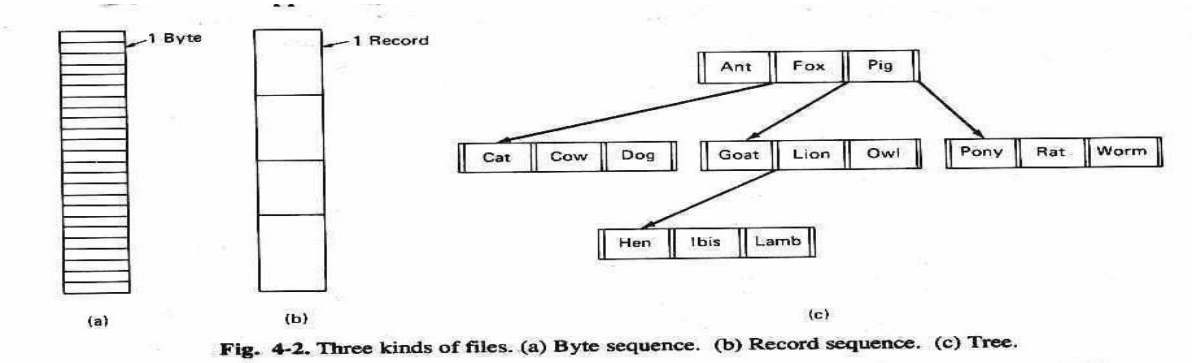


Fig. 4-2. Three kinds of files. (a) Byte sequence. (b) Record sequence. (c) Tree.

Three common structures are depicted in fig. above.

Most commonly used in modern operating systems is part (a): **an unstructured sequence of bytes.** The operating system does not know nor care what is in the file. It only sees bytes. This is used by UNIX and DOS/Windows. The user can put anything it wants in the file and the OS does not help or get in the way of the user.

In part b of fig 1, the file is made up of a **series of fixed length records**. The unit of access is a record which is may be many bytes. This roots back to early days when a file represented a sequence of punch cards of 80 columns wide, or for output a 132 byte record was sent to the printer. Read and write operations affected a whole record. This organization is not used much anymore.

The 3rd type of structure depicted in part (c) is a tree of variable size records, each having a key field in a fixed position in the record for sorting. Can used randomly by requesting a record having a certain key. Used in large mainframes for commercial processing.

File Access:

Early operating systems provided only one type of access: ―**sequential a**ccess‖. The file had to be read from the beginning to get at some byte or record inside.

These models of a media such as a tape.\

Random access is now more commonly used. A seek call may provide the programmer the capability of accessing anywhere in the file.

File Access – for the programmer and user

Create, delete, open, close, read, write, append, seek, get attributes, set attributes, and rename. These may be system calls, or some may be commands for the user.

Files may be access as memory mapped – see text on this.

Directorie structures:

A directory contains an entry per file – associated with the directory.

Essential information in a directory (directly or indirectly represented): File name
File attributes
Disk address

When a file is ―opened‖, the OS searches its directory for the name, and extracts the attributes and disk address of the file and puts these into a table in memory. All subsequent references to the file uses this in-memory table. The pointer to this table is sometimes known as a ―handle‖ or file descriptor.

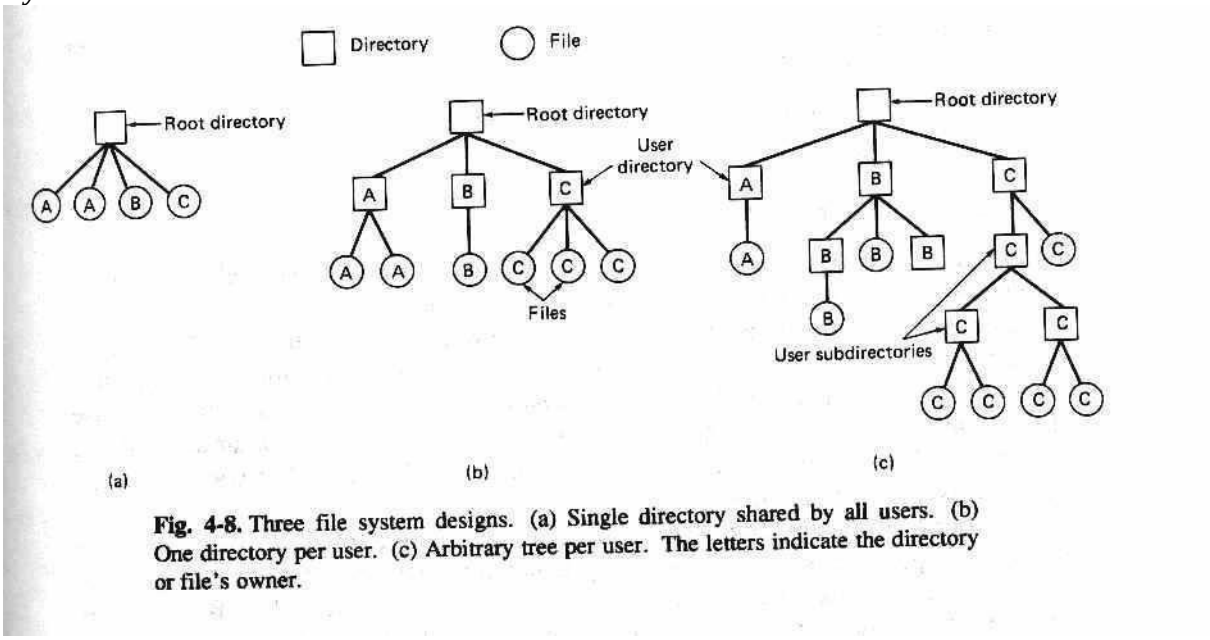


Fig. 4-8. Three file system designs. (a) Single directory shared by all users. (b) One directory per user. (c) Arbitrary tree per user. The letters indicate the directory or file's owner.

The simplest directory is to have a single directory containing all files of all users. (fig. 3 part (a) above). This was used in only the most primitive OS's.

An improvement is to have one directory per user (fig. 3 part (b) above). This eliminates name conflicts among users – example IBM/VM.

Fig. 3 part (c) above allows users to group their files together into categories of their choice. This is a tree directory – most commonly see in modern systems.

For part (c), a ―path will have to be used to access a file in a tree directory.

Single-Level Directory

This is a simple directory structure that is very easy to support. All files reside in one and the same directory .

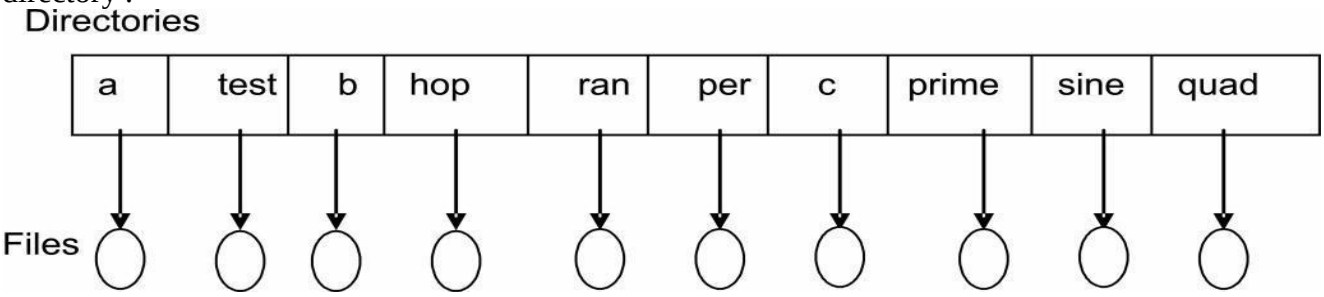


Figure: Single-level directory structure

A single-level directory has limitations as the number of files and users increase. Since there is only one directory to list all the files, no two files can have the same name that is, file names must be unique in order to identify one file from another. Even with one user, it is difficult to maintain files with unique names when the number of files becomes large.

Two-Level Directory

The main limitation of single-level directory is to have unique file names by different users. One solution to the problem could be to create separate directories for each user. A two-level directory structure has one directory exclusively for each user. The directory structure of each user is similar in structure and maintains file information about files present in that directory only. The operating system has one master directory for a partition. This directory has entries for each of the user directories.

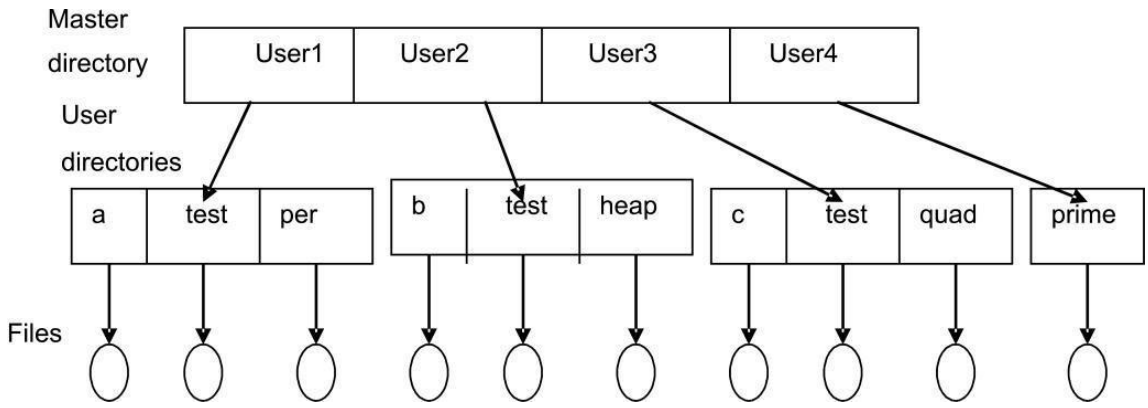


Figure: Two-level directory structure

Files with same names exist across user directories but not in the same user directory. File maintenance is easy. Users are isolated from one another. But when users work in a group and each wants to access files in another users directory, it may not be possible. Access to a file is through user name and file name. This is known as a path. Thus a path uniquely defines a file. For example, in MS-DOS if _C' is the

partition then C:\USER1\TEST, C:\USER2\TEST, C:\USER3\C are all files in user directories. Files could be created, deleted, searched and renamed in the user directories only.

Tree-Structured Directories

A two-level directory is a tree of height two with the master file directory at the root having user directories as descendants that in turn have the files themselves as descendants. This generalization allows users to organize files within user directories into sub directories. Every file has a unique path. Here the path is from the root through all the sub directories to the specific file. Usually the user has a current directory. User created sub directories could be traversed. Files are usually accessed by giving their path names. Path names could be either absolute or relative. Absolute path names begin with the root and give the complete path down to the file. Relative path names begin with the current directory. Allowing users to define sub directories allows for organizing user files based on topics. A directory is treated as yet another file in the directory, higher up in the hierarchy. To delete a directory it must be empty. Two options exist: delete all files and then delete the directory or delete all entries in the directory when the directory is deleted. Deletion may be a recursive process since directory to be deleted may contain sub directories.

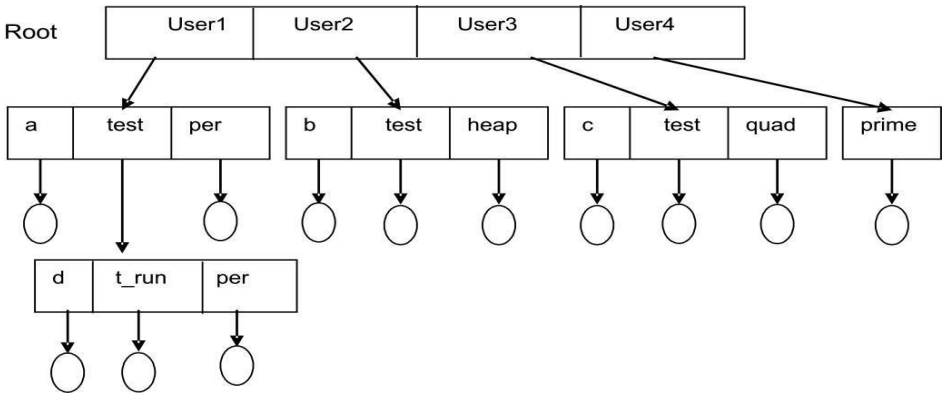


Figure : Tree-structured directory structure

File operations: .

Files exist to store information and allow it to be retrieved later. Different systems provide different Operations to allow storage and retrieval. The few of them of the most common system calls relating to files are:

***Create*:** The file is created with no data. The purpose of the call is to announce that the file is coming and to set some of the attributes.

- ***Delete*:** When the file is no longer needed, it has to be deleted to free up disk space.
- ***Open*:** Before using a file, a process must open, the purpose of the open call is to allow the system to fetch the attributes and list of disk addresses into main memory for rapid access on later calls.

***Close*:** When all the accesses are finished, the attributes and disk addresses are no longer needed, so the file should be closed to free up internal table space.

***Read*:** Data re read from file. Usually, the bytes a come from the current position. The caller must specify how much data are needed and must also provide a buffer to put them in.

***Write*:** Data are written to the file, again, usually at the current position. If the current position is end of the file then the file size gets increased.

Append: This call is a restricted from of write. It can only add data to the end of the file.

- ***Seek*:** For random access files, a method is needed to specify from where to take the data.
- ***Get attributes***: Processes often need to read file attributes to do their work.
- ***Set attributes*:** Some of the attributes are user settable and can be changed after the file has been created. This system call makes that possible. The protection mode information is an obvious example.

***Rename*:** It frequently happens that a user needs to change the name of an existing file.

File management systems:

The *file management system, FMS* is the subsystem of an operating system that manages the data storage organization on disk, and provides services to processes related to file access. In this sense, it interfaces the application programs with the low-level media-I/O (e.g. disk I/O) subsystem, freeing on the application programmers from having to deal with low-level intricacies and allowing them to implement I/O using convenient data-organizational abstractions like files and records. On the other end, the FMS services often is the only way thorough which applications can access the data stored in the files, thus achieving an encapsulation of the data themselves which can be usefully exploited for the purposes of data protection, maintenance and control.

We can summarize the aims of a FMS as follows:

Data Management. An FMS should provide data management services to the applications through convenient abstractions, simplifying and making device-independant the common operations involved in data access and modification.

Generality with respect to storage devices. The FMS data abstractions and access methods should remain unchanged irrespective of the devices involved in data storage.

Validity. An FMS should guarantee that at any given moment the stored data reflect the operations performed on them, regardless of the time delays involved in actually performing those operations. Appropriate access synchronization mechanism should be used to enforce validity when multiple accesses from independant processes are possible.

Protection. Illegal or potentially dangerous operations on the data should be controlled by the FMS, by enforcing a well defined data protection policy.

Concurrency. In multiprogramming systems, concurrent access to the data should be allowed with minimal differences with respect to single-process access, save for access synchronization enforcement.

Performance. The above functionalities should be offered achieving at the same a good compromise in terms of data access speed and data transfering rate.

From the point of view of an end user (or application) an FMS typically provides the following functionalities:

File creation, modification and deletion.

User's (or user groups') ownership of files, and access control on the basis of ownership permissions.

Facilities to structure data within files (predefined record formats, etc).

Facilities for maintaining data redundance against technical failure (back-ups, disk mirroring, etc.).

Logical identification and structuring of the data, via file names and hyerarchical directory structures.

FMS Architecture

A general scheme of the architecture of an FMS is depicted in fig.

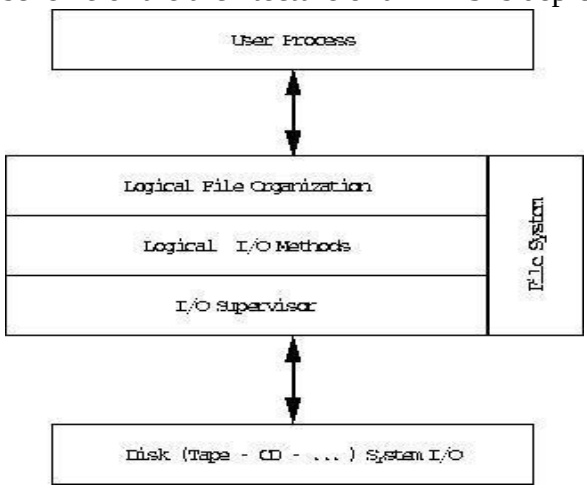


Figure : A scheme of a generic File Management System architecture

The file system provides a unified data structure containing the data along with the necessary management information for the use of the FMS.

The user application ``sees" and/or internally organizes the files according to a certain logical scheme (pile, sequential, ...).

A logical I/O module provides the necessary translation from this logical data organization into a physical one, suited for the low-level storage management subsystem. Record management is performed at this level, in the sense that it maps a physical (vlock-oriented) data structure into and from a record-oriented one, which is organized into some logical scheme at the upper level.

The I/O supervisor is responsible for I/O initialization, termination and data validity through multiple accesses. It also provides device independance, managing the communication between so to offer to the upper layers a uniform service structure across the various possible storage I/O devices managed by the disk (tape, cdrom, ...) I/O subsystem.

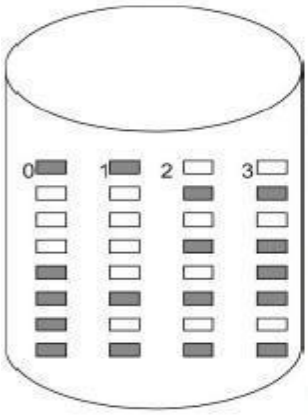
File allocation Methods:

Allocation of disk space to files is a problem that looks at how effectively disk space is utilized and quickly files can be accessed. The three major methods of disk space allocation are:

Contiguous allocation
Linked allocation
Indexed allocation

Contiguous Allocation:

Contiguous allocation requires a file to occupy contiguous blocks on the disk. Because of this constraint disk access time is reduced, as disk head movement is usually restricted to only one track. Number of seeks for accessing contiguously allocated files is minimal and so also seek times. A file that is _n' blocks long starting at a location _b' on the disk occupies blocks b, b+1, b+2, ….., b+(n-1). The directory entry for each contiguously allocated file gives the address of the starting block and the length of the file in blocks as illustrated below:



Directory:

File	Start	Length
a.c	0	2
per	14	3
hop	19	6
b	28	4
f	6	2

Accessing a contiguously allocated file is easy. Both sequential and random access of a file is possible. If a sequential access of a file is made then the next block after the current is accessed, whereas if a direct access is made then a direct block address to the ith block is calculated as b+i where b is the starting block address.

A major disadvantage with contiguous allocation is to find contiguous space enough for the file. From a set of free blocks, a first-fit or best-fit strategy is adopted to find _n' contiguous holes for a file of size _n'. But these algorithms suffer from external fragmentation. As disk space is allocated and released, a single large hole of disk space is fragmented into smaller holes. Sometimes the total size of all the holes put together is larger than the size of the file size that is to be allocated space. But the file cannot be allocated space because there is no contiguous hole of size equal to that of the file. This is when external fragmentation has occurred. Compaction of disk space is a solution to external fragmentation. But it has a very large overhead. Another problem with contiguous allocation is to determine the space needed for a file. The file is a dynamic entity that grows and shrinks. If allocated space is just enough (a best-fit allocation strategy is adopted) and if the file grows, there may not be space on either side of the file to expand. The solution to this problem is to again reallocate the file into a bigger space and release the existing space. Another solution that could be possible if the file size is known in advance is to make an allocation for the known file size. But in this case there is always a possibility of a large amount of internal fragmentation because initially the file may not occupy the entire space and also grow very slowly.

Linked Allocation

Linked allocation overcomes all problems of contiguous allocation. A file is allocated blocks of physical storage in any order. A file is thus a list of blocks that are linked together. The directory contains the address of the starting block and the ending block of the file. The first block contains a pointer to the second, the second a pointer to the third and so on till the last block Initially a

block is allocated to a file, with the directory having this block as the start and end. As the file grows, additional blocks are allocated with the current block containing a pointer to the next and the end block being updated in the directory.

This allocation method does not suffer from external fragmentation because any free block can satisfy a request. Hence there is no need for compaction. moreover a file can grow and shrink without problems of allocation.

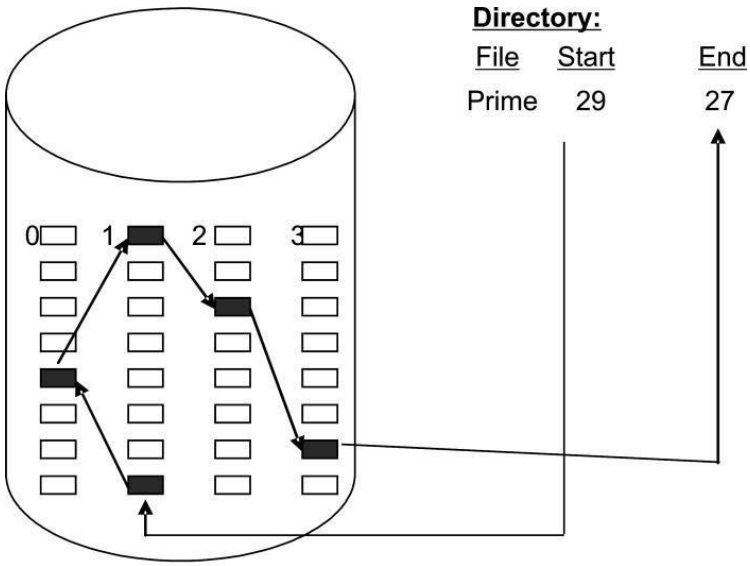


Figure: Linked allocation

Linked allocation has some disadvantages. Random access of files is not possible. To access the ith block access begins at the beginning of the file and follows the pointers in all the blocks till the ith block is accessed. Therefore access is always sequential. Also some space in all the allocated blocks is used for storing pointers. This is clearly an overhead as a fixed percentage from every block is wasted. This problem is overcome by allocating blocks in clusters that are nothing but groups of blocks. But this tends to increase internal fragmentation. Another problem in this allocation scheme is that of scattered pointers. If for any reason a pointer is lost, then the file after that block is inaccessible. A doubly linked block structure may solve the problem at the cost of additional pointers to be maintained. MS-DOS uses a variation of the linked allocation called a file allocation table (FAT). The FAT resides on the disk and contains entry for each disk block and is indexed by block number. The directory contains the starting block address of the file. This block in the FAT has a pointer to the next block and so on till the last block (Figure) Random access of files is possible because the FAT can be scanned for a direct block address.

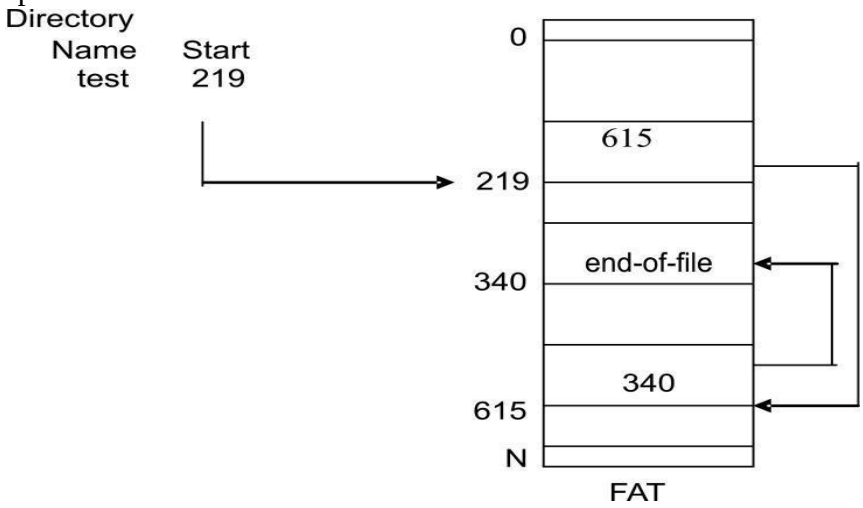


Figure : File allocation table

Indexed Allocation

Problems of external fragmentation and size declaration present in contiguous allocation are overcome in linked allocation. But in the absence of FAT, linked allocation does not support random access of files since pointers hidden in blocks need to be accessed sequentially. Indexed allocation solves this problem by bringing all pointers together into an index block. This also solves the problem of scattered pointers in linked allocation. Each file has an index block. The address of this index block finds an entry in the directory and contains only block addresses in the order in which they are allocated to the file. The ith address in the index block is the ith block of the file. Here both sequential and direct access of a file are possible. Also it does not suffer from external fragmentation.

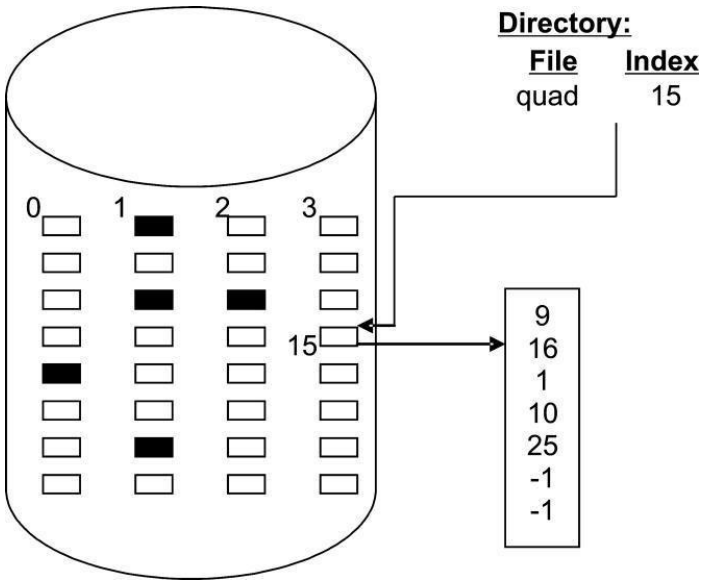


Figure: Indexed Allocation

Indexed allocation does suffer from wasted block space. Pointer overhead is more in indexed allocation than in linked allocation. Every file needs an index block. Then what should be the size of the index block? If it is too big, space is wasted. If it is too small, large files cannot be stored. More than one index blocks are linked so that large files can be stored. Multilevel index blocks are also used. A combined scheme having direct index blocks as well as linked index blocks has been implemented in the UNIX operating system.

Performance Comparison:

All the three allocation methods differ in storage efficiency and block access time. Contiguous allocation requires only one disk access to get a block, whether it be the next block (sequential) or the ith block (direct). In the case of linked allocation, the address of the next block is available in the current block being accessed and so is very much suited for sequential access. Hence direct access files could use contiguous allocation and sequential access files could use linked allocation. But if this is fixed then the type of access on a file needs to be declared at the time of file creation. Thus a sequential access file will be linked and cannot support direct access. On the other hand a direct access file will have contiguous allocation and can also support sequential access; the constraint in this case is making known the file length at the time of file creation. The operating system will then have to support algorithms and data structures for both allocation methods. Conversion of one file type to another needs a copy operation to the desired file type. Some systems support both contiguous and linked allocation. Initially all files have contiguous allocation. As they grow a switch to indexed allocation takes place. If on an average files are small, than contiguous file allocation is advantageous and provides good performance.

Free Space Management

The disk is a scarce resource. Also disk space can be reused. Free space present on the disk is maintained by the operating system. Physical blocks that are free are listed in a free-space list. When a file is created or a file grows, requests for blocks of disk space are checked in the free-space list and then allocated. The list is updated accordingly. Similarly, freed blocks are added to the free-space list. The free-space list could be implemented in many ways as follows:

Bit Vector

A bit map or a bit vector is a very common way of implementing a freespace list. This vector ‗n' number of bits where ‗n' is the total number of available disk blocks. A free block has its corresponding bit set (1) in the bit vector whereas an allocated block has its bit reset (0).

Illustration:

If blocks 2, 4, 5, 9, 10, 12, 15, 18, 20, 22, 23, 24, 25, 29 are free and the rest are allocated, then a free-space list implemented as a bit vector would look as shown below:

00101100011010010010101111000100000……

The advantage of this approach is that it is very simple to implement and efficient to access. If only one free block is needed then a search for the first ‗1' in the vector is necessary. If a contiguous allocation for ‗b' blocks is required, then a contiguous run of ‗b' number of 1's is searched. And if the first-fit scheme is used then the first such run is chosen and the best of such runs is chosen if best-fit scheme is used.

Bit vectors are inefficient if they are not in memory. Also the size of the vector has to be updated if the size of the disk changes.

Linked List

All free blocks are linked together. The free-space list head contains the address of the first free block. This block in turn contains the address of the next free block and so on. But this scheme works well for linked allocation. If contiguous allocation is used then to search for ‗b' contiguous free blocks calls for traversal of the free-space list which is not efficient. The FAT in MSDOS builds in free block accounting into the allocation data structure itself where free blocks have an entry say –1 in the FAT.

Grouping

Another approach is to store ‗n' free block addresses in the first free block. Here (n-1) blocks are actually free. The last nth address is the address of a block that contains the next set of free block addresses. This method has the advantage that a large number of free block addresses are available at a single place unlike in the previous linked approach where free block addresses are scattered.

Counting

If contiguous allocation is used and a file has freed its disk space then a contiguous set of ‗n' blocks is free. Instead of storing the addresses of all these ‗n' blocks in the free-space list, only the starting free block address and a count of the number of blocks free from that address can be stored. This is exactly what is done in this scheme where each entry in the freespace list is a disk address followed by a count.

Directory Implementation

The two main methods of implementing a directory are:

- **Linear list**
- **Hash table**

Linear List

A linear list of file names with pointers to the data blocks is one way to implement a directory. A linear search is necessary to find a particular file. The method is simple but the search is time consuming. To create a file, a linear search is made to look for the existence of a file with the same file name and if no such file is found the new file created is added to the directory at the end. To delete a file, a linear search for the file name is made and if found allocated space is released. Every time making a linear search consumes time and increases access time that is not desirable since directory information is frequently used. A sorted list allows for a binary search that is time efficient compared to the linear search. But maintaining a sorted list is an overhead especially because of file creations and deletions.

Hash table

Another data structure for directory implementation is the hash table. A linear list is used to store directory entries. A hash table takes a value computed from the file name and returns a pointer to the file name in the linear list. Thus search time is greatly reduced. Insertions are prone to collisions that are resolved. The main problem is the hash function that is dependent on the hash table size. A solution to the problem is to allow for chained overflow with each hash entry being a linked list. Directory lookups in a hash table are faster than in a linear list.

Disk Scheduling

Disk can do only one request at a time; what order do you choose to do queued requests?!
– Request denoted by (track, sector)!

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.

Disk Scheduling

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

Access time has two major components

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

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

Minimize seek time

Seek time seek distance

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

Disk Scheduling Algorithms:

We can minimize seek time by minimizing the distance the read/write head has to move in order to service the incoming requests.

Given a sequence of cylinders that must be visited to service a set of pending disk read/write requests, the system can order the requests to minimize seek time.

This may be done by the disk, the hardware controller, or by the operating system.

We will compare algorithms by examining their performance on a given **request queue**.

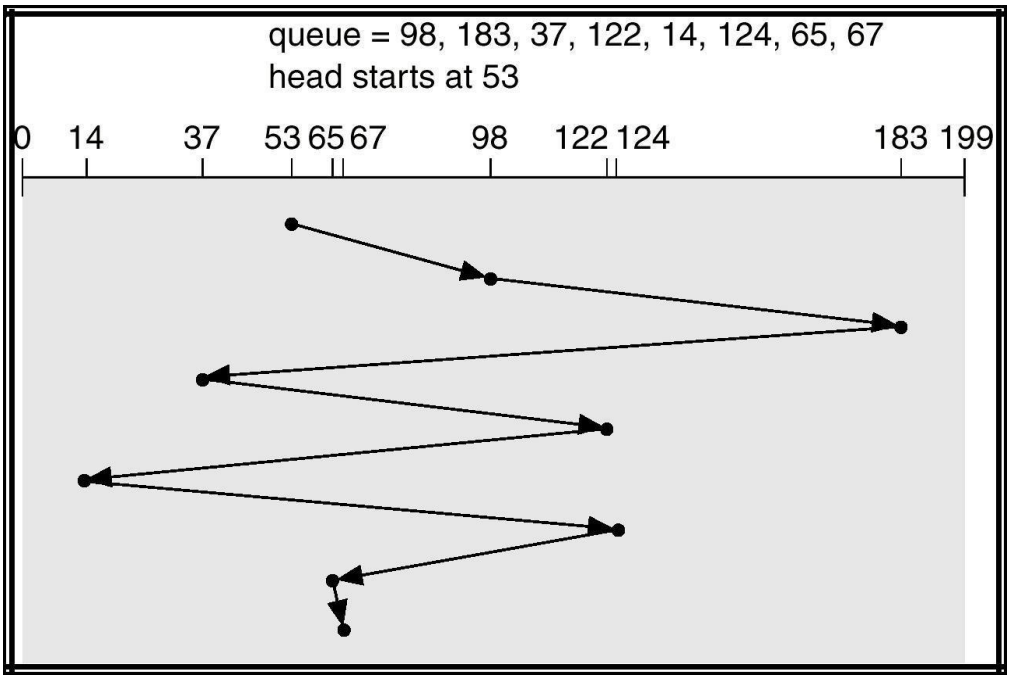
Given a disk with 200 cylinders (0-199), suppose we have 8 pending requests:

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

67 and that the read/write read is currently at cylinder 53.

First-Come First-Served (FCFS):

We include this analog of FCFS CPU scheduling or FIFO page replacement mainly for comparison purposes.



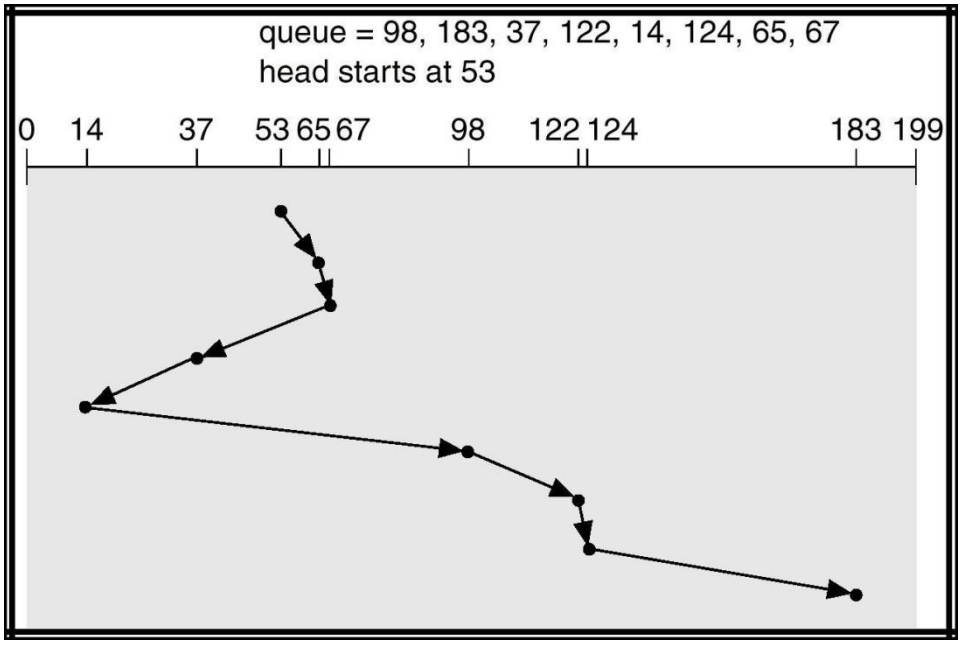
Requests are serviced in queue order, for a total of **640** cylinders of movement.

Shortest Seek Time First (SSTF)/Closest Cylinder Next :

Service the request next that has the shortest movement from the current position.

This is the analog of SJF CPU scheduling and OPT page replacement, but unlike those, it's actually possible, since we do have our actual request queue available to us.

In our example:



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

The total seek distance is **236** cylinders.

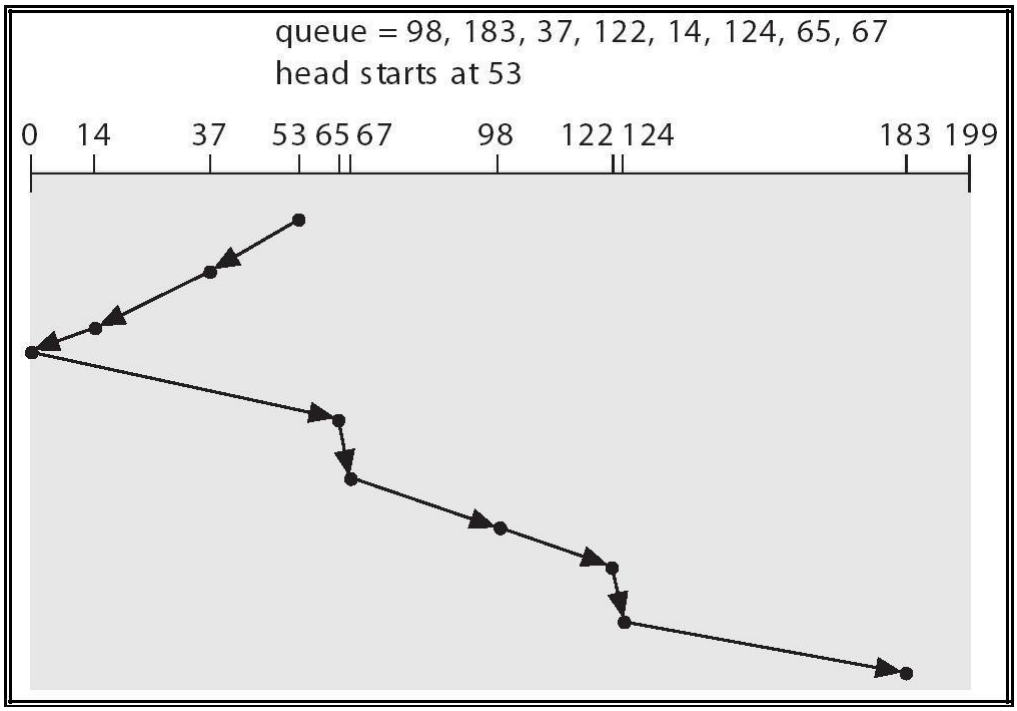
Potential problem: if many requests keep arriving near where the disk head is positioned, distant requests may be starved.

SCAN or Elevator Algorithm:

When an elevator is going in one direction, it stops at all the floors where there is a pending request. Then it reverses direction and does the same thing.

With this algorithm, the disk arm does just this. Service requests in one direction, then reverse direction.

In our example, assuming we are "going down" at the start:



37, 14, (0), 65, 67, 98, 122, 124, 183

236 cylinders again. It is a coincidence that this is the same as SSTF.

Note that the disk arm went all the way to 0, even though there were no requests below 14. This is because this particular algorithm doesn't look ahead, it just moves back and forth from one end to the other.

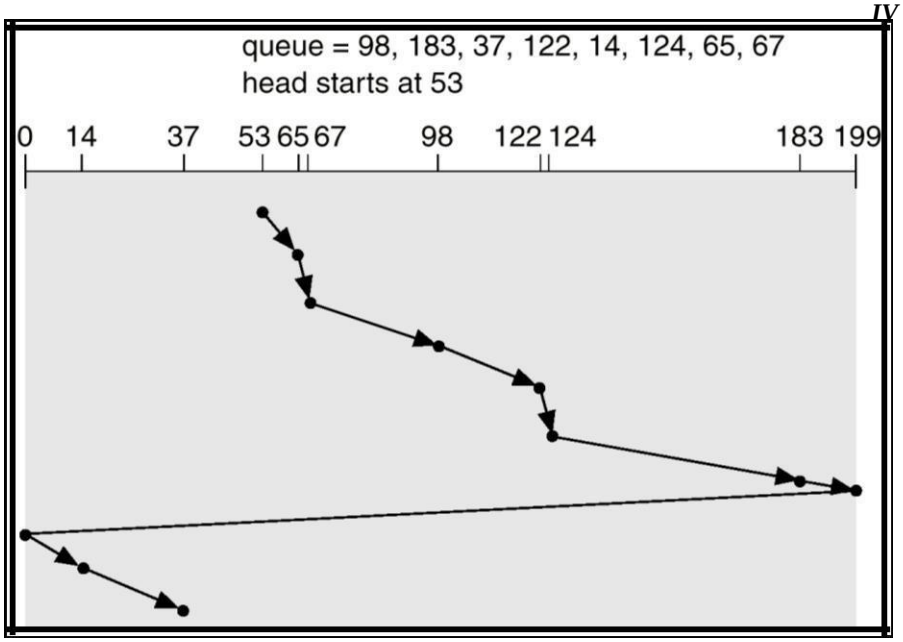
We can take care of that extra movement down to 0 with ...

C-SCAN Algorithm:

Provides a more uniform wait time than SCAN.

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

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



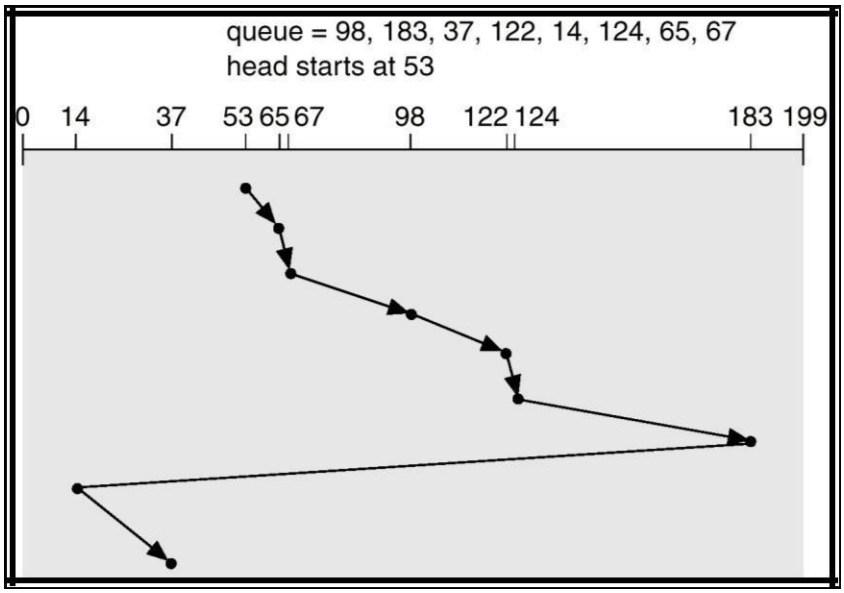
LOOK Algorithm:

It's the same as SCAN, but the head reverses direction as soon as there are no pending requests in the current direction.

The movement is the same as SCAN, just without that move from 14 to 0 and then up to 65. This reduces the movement to 208 cylinders.

Both SCAN and LOOK can lead to non-uniform waiting times. A request near one end of the disk sometimes needs to wait for two sweeps across the disk, while other times it will be serviced very quickly. Requests near the middle have a more uniform average waiting time.

ookAlgorithm:



Circular Algorithms(Details)

This problem can be addressed using circular versions of SCAN (C-SCAN) and LOOK (C-LOOK), where when the disk arm gets to the end of the disk, it jumps immediately back to the other end.

Assuming the disk services requests only when "going up", our example using these algorithms are served in order:

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

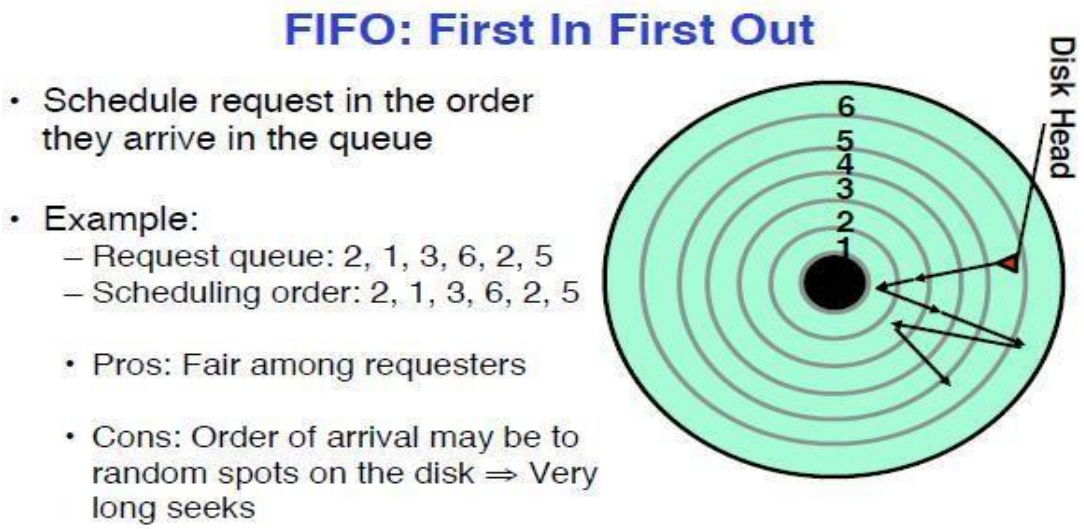
With C-SCAN, the head goes all the way to 199 and all the way back to 0, giving total movement of 382. With C-LOOK, we do not need to go up past 183 or down past 14, making the movement total 322.

The penalty of the movement all the way back in the other direction may not be as large as it seems. Think of the mechanics of the situation - starting and stopping the disk arm takes more time than simply sweeping all the way across with just one acceleration and deceleration.

Comparing Disk Scheduling Algorithms

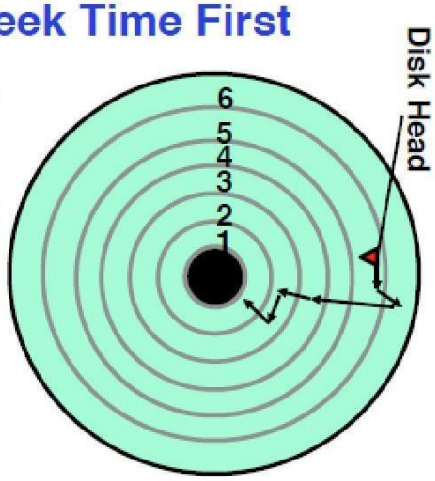
SSTF or LOOK are often reasonable for a default algorithm
SCAN and C-SCAN are better for heavily loaded systems where LOOK is unlikely to save much and SSTF runs the risk of starvation
performance depends on the frequency and types of requests
we may want to consider some of this when we look next week at how to organize file systems

One More Example:



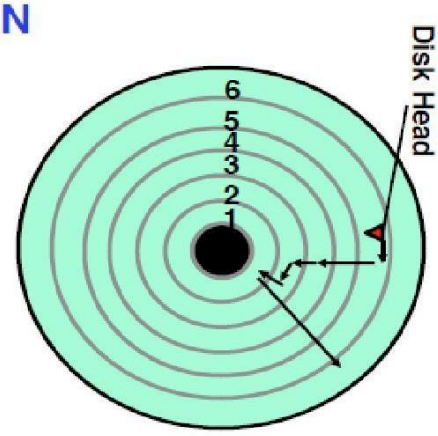
SSTF: Shortest Seek Time First

- Pick the request that's closest to the head on the disk
  - Although called SSTF, include rotational delay in calculation, as rotation can be as long as seek
- Example:
  - Request queue: 2, 1, 3, 6, 2, 5
  - Scheduling order: 5, 6, 3, 2, 2, 1
- Pros: reduce seeks
- Cons: may lead to starvation



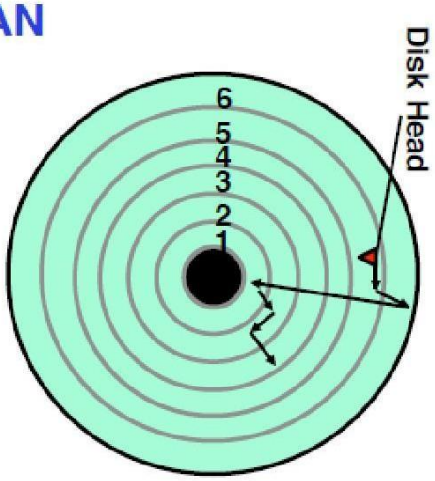
SCAN

- Implements an Elevator Algorithm: take the closest request in the direction of travel
- Example:
  - Request queue: 2, 1, 3, 6, 2, 5
  - Head is moving towards center
  - Scheduling order: 5, 6, 3, 2, 2, 1
- Pros:
  - No starvation
  - Low seek
- Cons: favor middle tracks



C-SCAN

- Like SCAN but only serves request in only one direction
- Example:
  - Request queue: 2, 1, 3, 6, 2, 5
  - Head only servers request on its way from center towards edge
  - Scheduling order: 5, 6, 1, 2, 2, 3
- Pros:
  - Fairer than SCAN
- Cons: longer seeks on the way back



n
n

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.

Sector queuing:

With a fixed-head disk, there is no head movement, so rotational latency becomes the dominant component of access time.

A. For such disks, we can maintain a (FIFO) queue of requests for each of the sectors. As the disk rotates, we service one request from each queue in turn as each sector goes by. (Note: we may have to skip over a sector each time to allow for delay in starting/finishing each access.)

   Example: a fixed-head disk with 8 sectors per track (0..7 are track 0, 8..15 are track, 1 etc.)

   Using the same example requests as before, but assuming they are **sector numbers**, not track numbers, we would get:

Queue	0	1	2	3	4	5	6	7
Sector Number		65	98 122	67	124	37	14	183

   With the head initially at sector 53 (which is part of queue 5), and assuming we can access one sector immediately after we finish its predecessor (a big assumption) our service order would be

   14 183 [skip] 65 98 67 124 37 [skip] [skip]

   [skip] [skip] 122 So we finish in two

   rotations (assuming no new arrivals).

B. Sector queuing can be combined with track ordering on a movable head disk if there are multiple requests for the same track, but this is seldom worthwhile.

UNIT III

Process Management & Scheduling

Prof. Vishal M. Tiwari
Department of CSE,TGPCET

A key concept in all OS is the **process**. A program does nothing unless its instructions are executed by a CPU. A program in execution, as mentioned, is a process.

A process needs certain resources-including CPU time, memory, files, and I/O devices-to accomplish its task. These resources are either given to the process when it is created or allocated to it while it is running.

When the process terminates, the OS will reclaim any reusable resources.

Associated with each process is its **address space**, a list of memory locations from 0 to some maximum, which the process can read and write. The address space contains the executable program, the program's data and its stack. Also associated with each process is a set of resources, commonly including registers (program counter, stack pointer, ..), a list of open files, outstanding alarms, lists of related processes, and all the other information needed to run the program.

In many OSs, all the information about each process, other than the contents of its own address space, is stored in a table called the **process table**, which is an array (or linked list) of structures, one for each process currently in existence.

If a process can create one or more other processes (referred to as **child processes**) and these processes in turn can create child processes, we quickly arrive at the process tree structure of Fig. 3.1

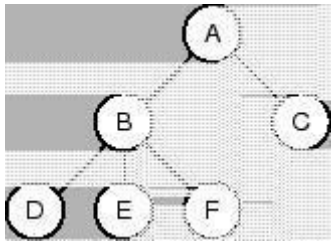


Figure 3.1: A process tree. Process A created two child processes, B and C. Process B created three child processes, D, E, and F.

A process is the unit of work in a system. Such a system consists of a collection of processes, some of which are operating-system processes (those that execute system code) and the rest of which are user processes (those that execute user code). All these processes can potentially execute concurrently by multiplexing the CPU among them on a single CPU.

The OS 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.

Process Concept

A batch system executes jobs, whereas a time-shared system has user programs, or tasks. In many respects, all these activities are similar, so we call all of them **processes**. The terms job and process are used almost interchangeably.

The 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.

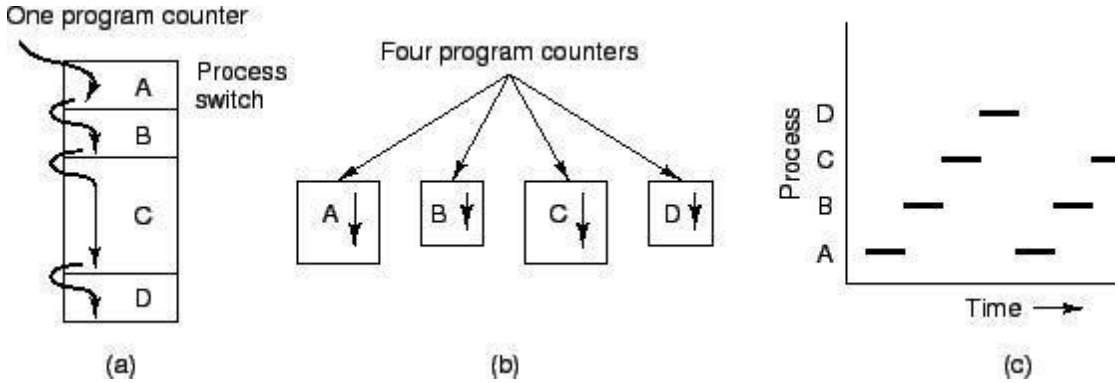


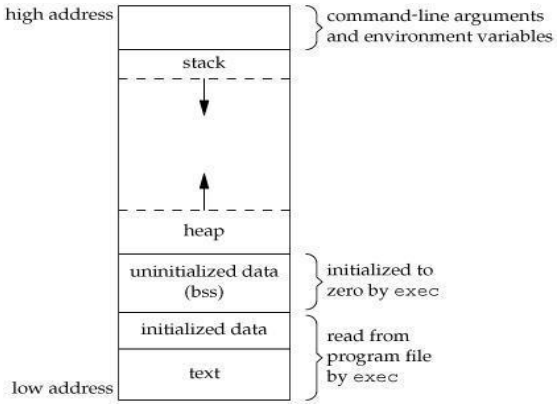
Figure 3.2: (a) Multiprogramming of four programs. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at once.

- In Fig. 3.2 a we see a computer multiprogramming four programs in memory.
- In Fig. 3.2 b we see four processes, each with its own flow of control (i.e., its own logical program counter), and each one running independently of the other ones. Of course, there is only one physical program counter, so when each process runs, its logical program counter is loaded into the real program counter. When it is finished for the time being, the physical program counter is saved in the process' logical program counter in memory.
- In Fig. 3.2 c we see that viewed over a long enough time interval, all the processes have made progress, but at any given instant only one process is actually running.

A process generally also includes

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

The structure of a process in memory is shown in Fig. 3.3



A program becomes a process when an executable file is loaded into memory.

Although two processes may be associated with the same program, they are nevertheless considered two separate execution sequences.

- For instance, several users may be running different copies of the mail program,
- or the same user may invoke many copies of the web browser program.

Each of these is a separate process; and although the text sections are equivalent, the data, heap, and stack sections vary.

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.

The state diagram corresponding to these states is presented in Fig. 3.4

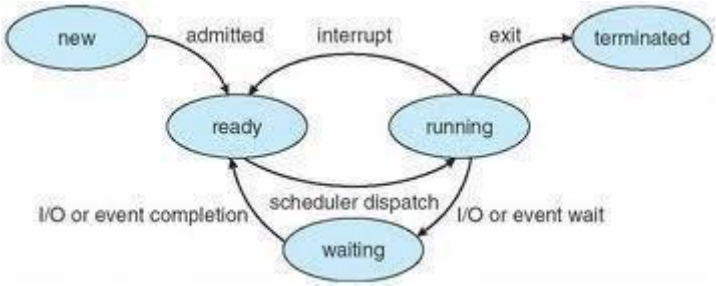


Figure 3.4: Diagram of process state.

Using the process model, it becomes much easier to think about what is going on inside the system.

- Some of the processes run programs that carry out commands typed in by a user.
- Other processes are part of the system and handle tasks such as carrying out requests for file services or managing the details of running a disk or a tape drive.
- When a disk interrupt occurs, the system makes a decision to stop running the current process and run the disk process, which was blocked waiting for that interrupt.

Instead of thinking about interrupts, we can think about user processes, disk processes, terminal processes, and so on, which block when they are waiting for something to happen. When the disk has been read or the character typed, the process waiting for it is unblocked and is eligible to run again. This view gives rise to the model shown in Fig. 3.5.

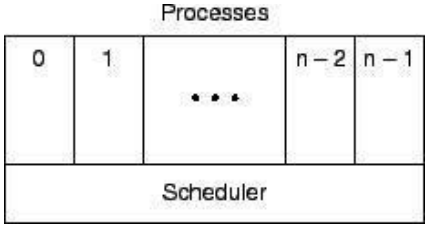


Figure 3.5: *The lowest layer of a process-structured OS handles interrupts and scheduling. Above that layer are sequential processes.*

Process Control Block

The OS must know specific information about processes in order to manage, control them and also to implement the process model, the OS maintains a table (an array of structures), called the **process table**, with one entry per process.

These entries are called **process control blocks** (PCB) - also called a task control block. This entry contains information about the process'state, its program counter, stack pointer, memory allocation, the status of its open files, its accounting and scheduling information, and everything else about the process that must be saved when the process is switched from running to ready or blocked state so that it can be restarted later as if it had never been stopped. A PCB is shown in Fig. 3.6.

Figure 3.6: Process control block (PCB).

Such information is usually grouped into two categories:
Process State Information and Process Control Information.
Including these:

- **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 OS.
- **Accounting information**. This information includes the amount of CPU and real time used, time limits, account numbers, 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.

Figure 3.7 shows some of the more important fields in a typical system. The fields in the first column relate to process management. The other two columns relate to memory management and file management, respectively.

Process management	Memory management	File management
Registers	Pointer to text segment info	Root directory
Program counter	Pointer to data segment info	Working directory
Program status word	Pointer to stack segment info	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Figure 3.7: Some of the fields of a typical process table entry.

Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward (see Fig. 3.8).

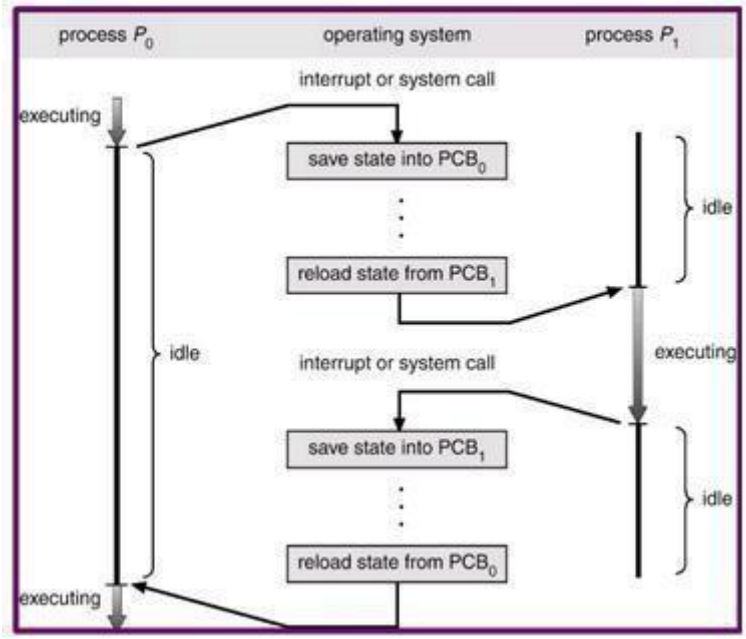


Figure 3.8: *Diagram showing CPU switch from process to process (Context Switching).*

Process Scheduling

The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.

With the CPU switching back and forth among the processes, the rate at which a process performs its computation will not be uniform and probably not even

reproducible if the same processes are run again.

The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running.

To meet these objectives, the process scheduler selects an available process (possibly from a set of several available processes) for program execution on the CPU.

For a single-processor system, there will never be more than one running process. If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled.

Scheduling Queues

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.

Suppose the process makes an I/O request to a shared device, such as a disk. Since there are many processes in the system, the disk may be busy with the I/O request of some other process.

The process therefore may have to wait for the disk. The list of processes waiting for a particular I/O device is called a device queue. Each device has its own device queue (see Fig. 3.9).

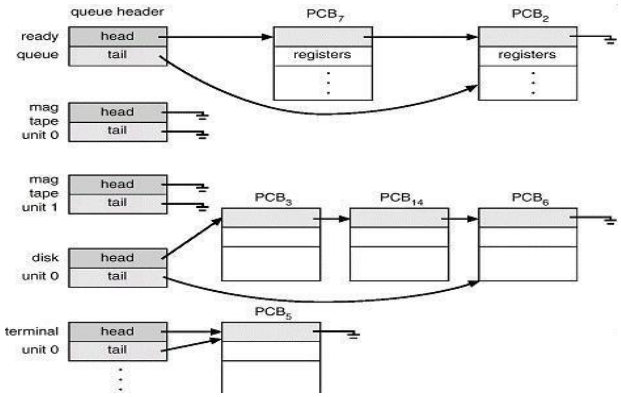


Figure 3.9: The ready queue and various I/O device queues.

A common representation for a discussion of process scheduling is a queuing diagram, such as that in Fig. 3.10.

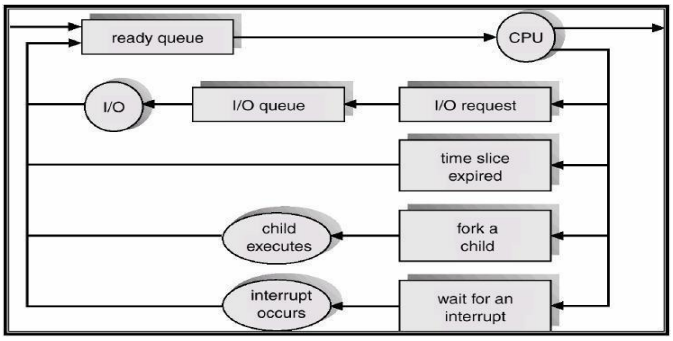


Figure 3.10: Queuing-diagram representation of process scheduling.

- 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 until 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 subprocess 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.

A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.

Process Scheduling Queues

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

Schedulers

A process migrates among the various scheduling queues throughout its lifetime.
The OS must select, for scheduling purposes, processes from these queues in some fashion. The selection process is carried out by the appropriate **scheduler**.
Often, in a batch system, more processes are submitted than can be executed immediately. These processes are spooled to a mass-storage device (typically a disk), where they are kept for later execution.

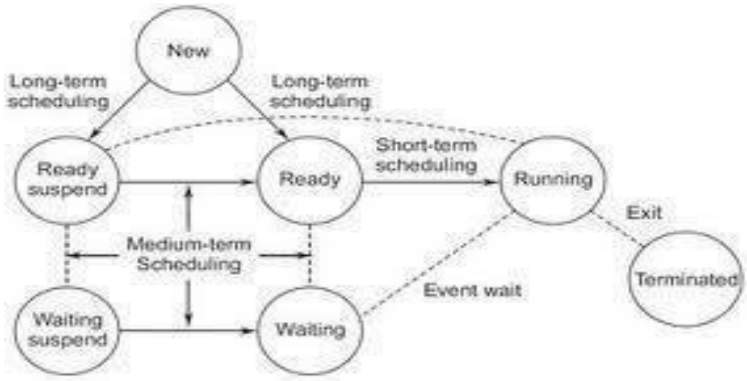


Fig: 3.11 Types of Schedulers/Scheduling

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.

The long-term scheduler controls the **degree of multiprogramming** (the number of processes in memory). If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system.

Thus, the long-term scheduler may need to be invoked only when a process leaves the system. Because of the longer interval between executions, the long-term scheduler can afford to take more time to decide which process should be selected for execution.

It is important that the long-term scheduler make a careful selection. In general, most processes can be described as either I/O bound or CPU bound.

- An I/O-bound process is one that spends more of its time doing I/O than it spends doing computations.
- A CPU-bound process, in contrast, generates I/O requests infrequently, using more of its time doing computations.

It is important that the long-term scheduler select a good **process mix** of I/O-bound and CPU- bound processes.

On some systems, the long-term scheduler may be absent or minimal. For example, time- sharing systems such as UNIX and Microsoft Windows systems often have no long-term scheduler but simply put every new process in memory for the short-term scheduler.

The stability of these systems depends either on a physical limitation (such as the number of available terminals) or on the self-adjusting nature of human users.

Some OSs, such as time-sharing systems, may introduce an additional, intermediate level of scheduling. This **medium-term scheduler** is diagrammed in Fig. 3.12.

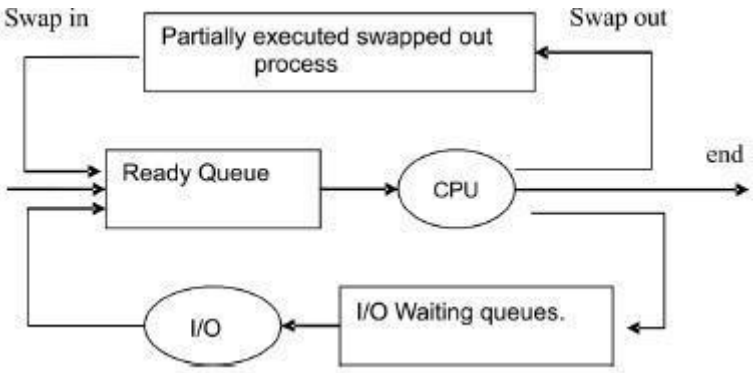


Figure 3.12: *Addition of medium-term scheduling to the queuing diagram.*

The key idea behind a medium-term scheduler is that sometimes it can be advantageous to remove processes from memory (and from active contention for the CPU) and thus reduce the degree of multiprogramming.

Later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is called **swapping**.

Context Switch

When an interrupt occurs, the system needs to save the current **context** of the process currently running on the CPU so that it can restore that context when its processing is done, essentially suspending the process and then resuming it.

The context is represented in the PCB of the process; it includes

- the value of the CPU registers,
- the process state,

- and memory-management information.

Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process.

This task is known as a **context switch**. When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run.

- process table keeps track of processes,
- context information stored in PCB,
- process suspended: register contents stored in PCB,
- process resumed: PCB contents loaded into registers

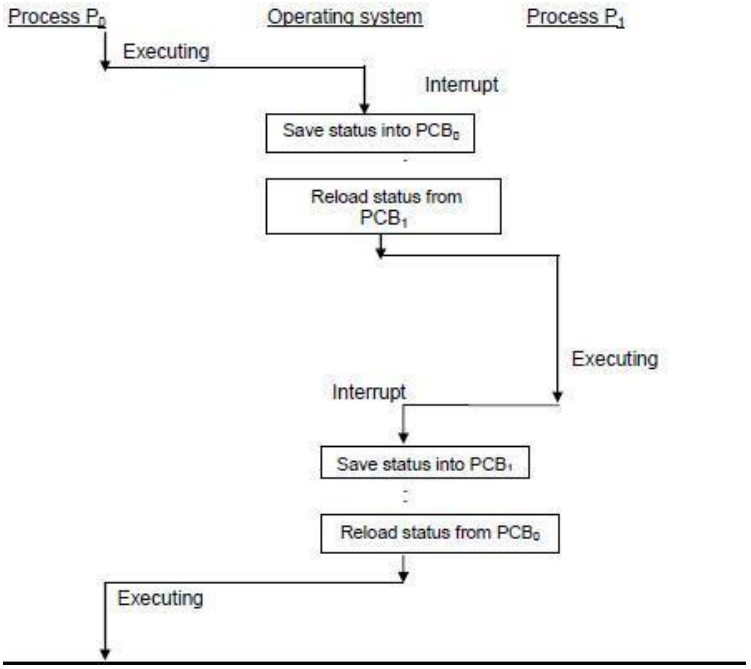


Fig: 3.13 Context Switch

Context-switch time is pure overhead, because the system does no useful work while switching. Context switching can be critical to performance.

Modelling Multiprogramming

When multiprogramming is used, the CPU utilization can be improved. Crudely put, if the average process computes only 20% of the time it is sitting in memory at once, the CPU should be busy all the time.

Unrealistically optimistic, assumes that all five processes will never be waiting for I/O at the same time.

Suppose that a process spend a fraction p of its time waiting for I/O to complete. With n processes in memory at once, the probability that all n processes are waiting for I/O is p^n. The CPU utilization is then given by the formula:

$$CPU\ utilization = 1 - p^n$$

Figure 3.11 shows the CPU utilization as a function of n, which is called the **degree**

of multiprogramming.

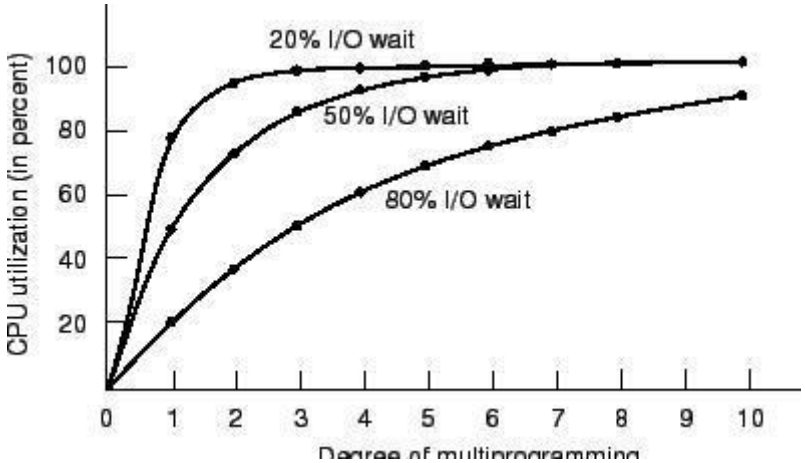


Figure 3.11: CPU utilization as a function of the number of processes in memory.

For the sake of complete accuracy, it should be pointed out that the probabilistic model is only an approximation. Context switching overhead is ignored.

Types of system calls

- *Process control:*
  end, abort; load, execute; create process, terminate process; get process attributes, set process attributes; wait for time; wait event, signal event; allocate and free memory
- *File management*
  Create file, delete file; open, close; read, write, reposition; get file attributes, set file attributes
- *Device management*
  Request device, release device; read, write, reposition; get device attributes, set device attributes;
  Logically attach or detach devices
- *Information maintenance*
  Get time or date, set time or date; get system data, set system data; get process, file, or device attributes; set process, file, or device attributes
- *Communications*
  Create, delete communication connection; send, receive messages; transfer status information; Attach or detach remote devices

Operations on Processes (OS Services for Process Management)

The processes in most systems can execute concurrently, and they may be created and deleted dynamically.
Thus, these systems must provide a mechanism for process creation and termination.

The processes in the system can execute concurrently, and must be created and

deleted dynamically. Thus the operating system must provide a mechanism for process creation and termination.

Process Creation

There are four principal events that cause processes to be created:

1. System initialization. When an OS is booted, typically several processes are created.
2. Execution of a process creation system call by a running process. Often a running process will issue system calls to create one or more new processes to help it do its job. Creating new processes is particularly useful when the work to be done can easily be formulated in terms of several related, but otherwise independent interacting processes.
3. A user request to create a new process. In interactive systems, users can start a program by typing a command or (double) clicking an icon.
4. Initiation of a batch job. Here users can submit batch jobs to the system (possibly remotely). When the OS decides that it has the resources to run another job, it creates a new process and runs the next job from the input queue in it.

During the course of execution, a process may create several new proceses using a create-process system call. 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.

A process in general need certain resources(CPU time , memory ,I/O devices) to accomplish its task. When a process creates a sub process, the sub process may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process. The parent may have to partition its resources among its children, or it may be able to share some resources (such as memory or files) among several of its children. In addition to the various physical and logical resources that a process obtains when it is created, initialization data (input) may be passed along by the parent process to the child process.

When a process creates a new process, two possibilities exit in terms of

execution: The parent continues to execute concurrently with its children.

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, the first possibility is,

The child process is a duplicate of the parent process.

An example for this is UNIX operating system in which 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.

The second possibility is,

The child process has a program loaded into it.

An example for this implementation is the DEC VMS operating system, it creates a new process, loads a specified program into that process, and starts it running. The Microsoft Windows/NT operating system supports both models: the parent's address space may
duplicated, or the parent may specify the name of a program for the operating system to load into the address space of the new process.

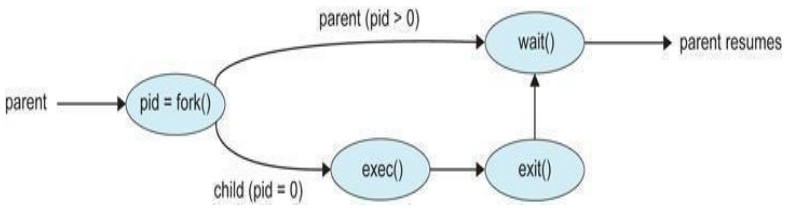


Figure 3.12: Process creation using Fork().

Process Termination

A process terminates when it finishes executing its last statement and asks the operating system to delete it by using the exit system call. At that time, the process should return the data (output) ti 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.

Only a parent process can cause termination of its children via abort system call. For that a parent needs to know the identities of its children. When one process creates a new process, the identity of the newly created process is passed to the parent. A parent may terminate the execution of one of its children for a variety of reasons, such as:

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.

To illustrate process execution and termination, let us consider UNIX system. In UNIX system a process may terminate by using the exit system call and its parent process may wait for that event by using the wait system call. The wait system call returns the process identifier of a terminated child, so the parent can tell which of the possibly many children has terminated. If the parent terminates, however all the children are terminated by the operating system. Without a parent, UNIX does not know whom to report the activities of a child.

Normal exit (voluntary):
Abnormal termination: programming errors, run time errors, I/O, user intervention.

- Error exit (voluntary): An error caused by the process, often due to a program bug (executing an illegal instruction, referencing non-existent memory, or dividing by zero). In some systems (e.g. UNIX), a process can tell the OS that it wishes to handle certain errors itself, in which case the process is signaled (interrupted) instead of terminated when one of the errors occurs.
- Fatal error (involuntary): i.e.; no such file exists during the compilation.
- Killed by another process (involuntary): A process can cause the termination of another process via an appropriate system call .Such a system call can be invoked only by the parent of the process that is to be terminated.

Threads

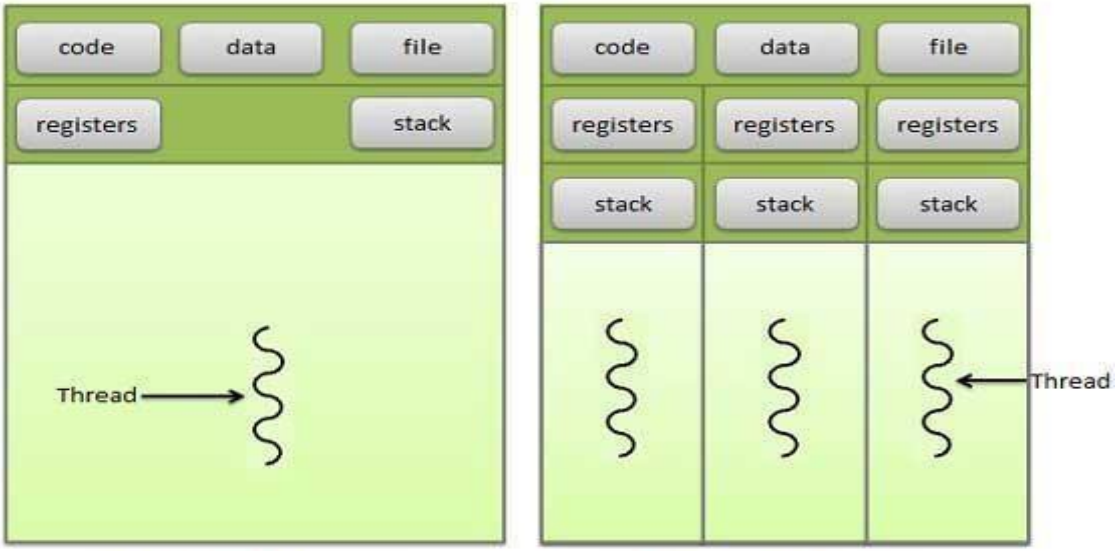
What is Thread?

A thread is a flow of execution through the process code, with its own program counter, system registers and stack. A thread is also called a light weight process. Threads provide a way to improve application performance through parallelism. Threads represent a software approach to improving performance of operating system by reducing the overhead thread is equivalent to a classical process.

Motivation for Threads

The use of threads can improve the performance of applications by allowing concurrent (parallel) execution.
The treads of a process typically cooperate to solve a problem.
The threads of a process can cooperate more efficiently than could separate processes.

Each thread belongs to exactly one process and no thread can exist outside a process. Each thread represents separate flows of control.Threads have been successfully used in implementing network servers and web server. They also provide a suitable foundation for parallel execution of applications on shared memory multiprocessors. Following figure shows the working of the single and multithreaded processes.



Difference between Process and Thread

S.N.	Process	Thread
1	Process is heavy weight or resource intensive.	Thread is light weight taking lesser resources than a process.
2	Process switching needs interaction with	Thread switching does not need to interact with

	operating system.	operating system.
3	In multiple processing environments each process executes the same code but has its	All threads can share same set of open files, child

	own memory and file resources.	processes.
4	If one process is blocked then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, second thread in the same task can run.
5	Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
6	In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

Advantages of Thread

Thread minimize context switching time.
Use of threads provides concurrency within a process. Efficient communication.
Economy- It is more economical to create and context switch threads. Utilization of multiprocessor architectures to a greater scale and efficiency.

Example 1: File Server on a LAN

Needs to handle many file requests over a short period
Threads can be created (and later destroyed) for each request
If multiple processors: different threads could execute simultaneously on different processors

Example 2: Spreadsheet on a single processor machine:

One thread displays menu and reads user input while the other executes the commands and updates display

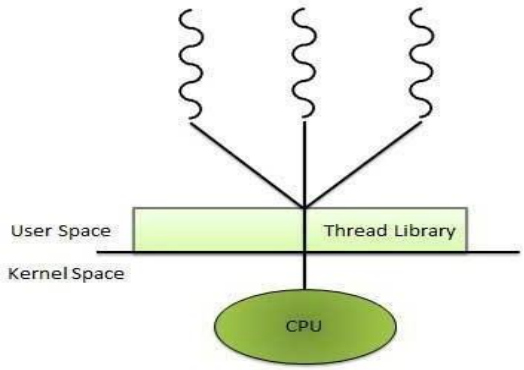
Types of Thread

Threads are implemented in following two ways

User Level Threads -- User managed threads
Kernel Level Threads -- Operating System managed threads acting on kernel, an operating system core.

User Level Threads

In this case, application manages thread management kernel is not aware of the existence of threads. The thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts. The application begins with a single thread and begins running in that thread.



Advantages

Thread switching does not require Kernel mode privileges. User level thread can run on any operating system.
Scheduling can be application specific in the user level thread. User level threads are fast to create and manage.

Disadvantages

In a typical operating system, most system calls are blocking.
Multithreaded application cannot take advantage of multiprocessing.

Kernel Level Threads

In this case, thread management done by the Kernel. There is no thread management code in the application area. Kernel threads are supported directly by the operating system. Any application can be programmed to be multithreaded. All of the threads within an application are supported within a single process.

The Kernel maintains context information for the process as a whole and for individuals threads within the process. Scheduling by the Kernel is done on a thread basis. The Kernel performs thread creation, scheduling and management in Kernel space. Kernel threads are generally slower to create and manage than the user threads.

Advantages

Kernel can simultaneously schedule multiple threads from the same process on multiple processes.
If one thread in a process is blocked, the Kernel can schedule another thread of the same process.
Kernel routines themselves can multithreaded.

Disadvantages

Kernel threads are generally slower to create and manage than the user threads.
Transfer of control from one thread to another within same process requires a mode switch to the Kernel.

Multithreading Models

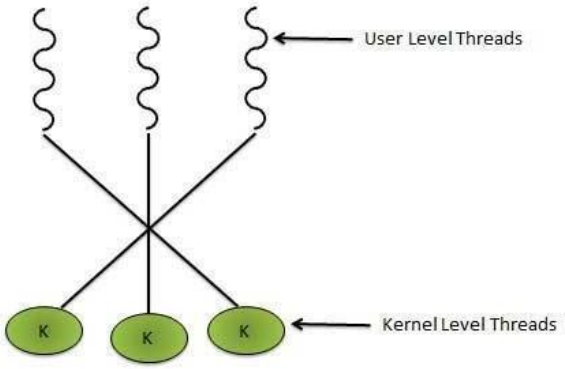
Some operating system provides a combined user level thread and Kernel level thread facility. Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the entire process. Multithreading models are three types

- *Many to many relationship.*
- **Many to one relationship.**
- **One to one relationship.**

Many to Many Model:

In this model, many user level threads multiplexes to the Kernel thread of smaller or equal numbers. The number of Kernel threads may be specific to either a particular application or a particular machine.

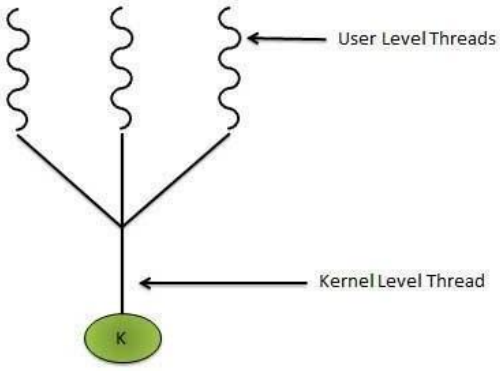
Following diagram shows the many to many model. In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallels on a multiprocessor.



Many to One Model

Many to one model maps many user level threads to one Kernel level thread. Thread management is done in user space. When thread makes a blocking system call, the entire process will be blocks. Only one thread can access the Kernel at a time,so multiple threads are unable to run in parallel on multiprocessors.

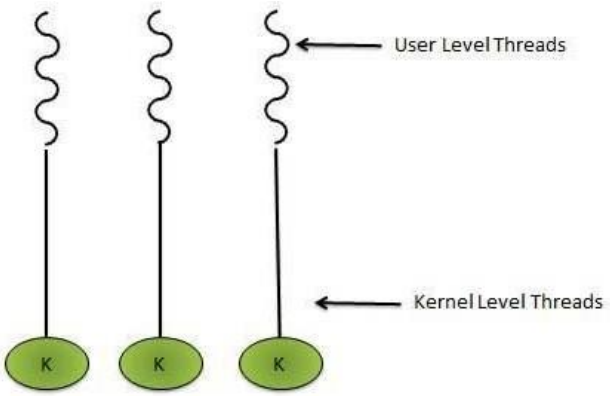
If the user level thread libraries are implemented in the operating system in such a way that system does not support them then Kernel threads use the many to one relationship modes.



One to One Model

There is one to one relationship of user level thread to the kernel level thread. This model provides more concurrency than the many to one model. It also has another thread to run when a thread makes a blocking system call. It support multiple thread to execute in parallel on microprocessors.

Disadvantage of this model is that creating user thread requires the corresponding Kernel thread. OS/2, windows NT and windows 2000 use one to one relationship model.



Benefits of Multi-Threading

The benefits of multithreaded programming can be broken down into four major categories:

1. **Responsiveness**. Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user. For instance, a multithreaded web browser could still allow user interaction in one thread while an image was being loaded in another thread.
2. **Resource sharing**. By default, threads share the memory and the resources of the process to which they belong. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.
3. **Economy of Overheads**. Allocating memory and resources for process creation is costly. Because threads share resources of the process to which they belong, it is more economical to create and context-switch threads. In Solaris, for example, creating a process is about thirty times slower than is creating a thread, and context switching is about five times slower.
4. **Utilization of multiprocessor architectures**. The benefits of multithreading can be greatly increased in a multiprocessor architecture, where threads may be running in parallel on different processors (real parallelism).

Difference between User Level & Kernel Level Thread

S.N.	User Level Threads	Kernel Level Thread
1	User level threads are faster to create and manage.	Kernel level threads are slower to create and manage.
2	Implementation is by a thread library at the user level.	Operating system supports creation of Kernel threads.
3	User level thread is generic and can run on any operating system.	Kernel level thread is specific to the operating system.
4	Multi-threaded application cannot take advantage of multiprocessing.	Kernel routines themselves can be multithreaded.

Thread Libraries

A thread library provides the programmer an API for creating and managing threads. There are two primary ways of implementing a thread library.

1. The first approach is to provide a library entirely in user space with no kernel support. All code and data structures for the library exist in user space. This means that invoking a function in the library results in a local function call in user space and not a system call.
2. The second approach is to implement a kernel-level library supported directly by the OS. In this case, code and data structures for the library exist in kernel space. Invoking a function in the API for the library typically results in a system call to the kernel.

Three main thread libraries are in use today:

1. **POSIX Pthreads**. Pthreads, the threads extension of the POSIX standard, may be provided as either a user- or kernel-level library.
2. **Win32**. The Win32 thread library is a kernel-level library available on Windows systems.
3. **Java**. The Java thread API allows thread creation and management directly in Java programs. However, because in most instances the JVM is running on top of a host OS, the Java thread API is typically implemented using a thread library available on the host system.

Pthreads

Pthreads refers to the POSIX standard (IEEE 1003.1c) defining an API for thread creation and synchronization.

This is a specification for thread behavior, not an implementation. Operating system designers may implement the specification in any way they wish.

Numerous systems implement the Pthreads specification, including Solaris, Linux, Mac OS X, and Tru64 UNIX. Shareware implementations are available in the public domain for the various Windows OSs as well.

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

Fig: Some of the Pthreads function calls.

Implementation of Threads

A process is **sequential** if a single thread of control regulates its address space.

Allowing **multiple threads of control** in a process introduces concurrency. Each thread shares and operates within the common process address space but each has its own local processor state maintained in a thread control block(TCB) associated with the process. Thread management is very lightweight as thread creation and context switching involves little overhead. Threads communicate and synchronize with each other using fast shared

memory mechanisms.

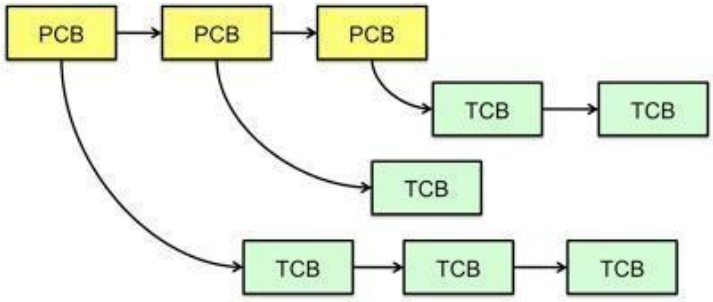
Thread Control Block

A thread control block (TCB) contains a thread identifier, a reference to the PCB for the process to which it belongs, a reference to the TCB for the thread that created it, and an execution snapshot.

A thread has indirect access to non-execution related resources through the PCB of the process to which it belongs. Thus threads within a process share memory, open files, and I/O streams.

How OS handles Threads

The operating system saves information about each process in a process control block (PCB). These are organized in a process table or list. Thread-specific information is stored in a data structure called a **TCB**. Since a process can have one or more threads (it has to have at least one; otherwise there's nothing to run!), each PCB will point to a list of TCBs.



User Level:

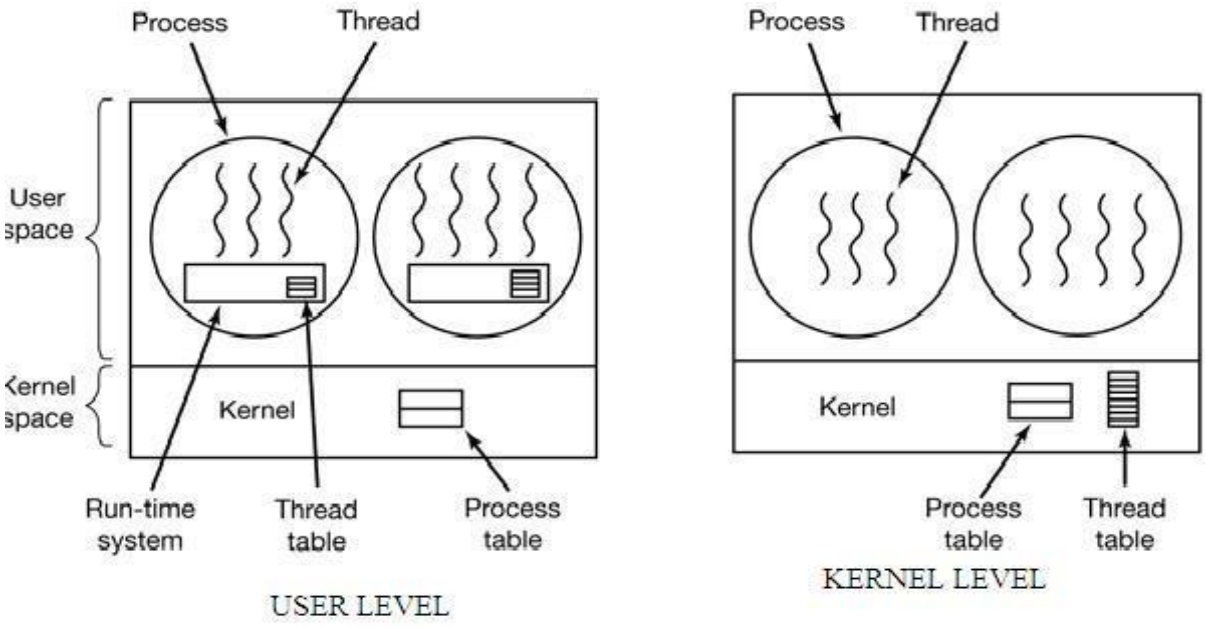
There are two main ways to implement a threads package: in user space and in the kernel. The first method is to put the threads package entirely in user space. The kernel knows nothing about them. As far as the kernel is concerned, it is managing ordinary, single-threaded processes. The first, and most obvious, advantage is that a user-level threads package can be implemented on an operating system that does not support threads. All operating systems used to fall into this category, and even now some still do.

All of these implementations have the same general structure, which is illustrated in Fig. 2-13(a). The threads run on top of a run-time system, which is a collection of procedures that manage threads.

When threads are managed in user space, each process needs its own private **thread table** to keep track of the threads in that process. This table is analogous to the kernel's process table, except that it keeps track only of the per-thread properties such the each thread's program counter stack pointer, registers, state, etc. The thread table is managed by the runtime system. When a thread is moved to ready state or blocked state, the information needed to restart it is stored in the thread table, exactly the same way as the kernel stores information about processes in the process table.

When a thread does something that may cause it to become blocked locally, for example, waiting for another thread in its process to complete some work, it calls a run-time system procedure. This procedure checks to see if the thread must be put into blocked state. If so, it

stores the thread's registers (i.e., its own) in the thread table, looks in the table for a ready thread to run and reloads the machine registers with the new thread's saved values. As soon as the stack pointer and program counter have been switched, the new thread comes to life again automatically. If the machine has an instruction to store all the registers, and another one to load them all, the entire thread switch can be done in a handful of instructions. Doing thread switching like this is at least an order of magnitude faster than trapping to the kernel and is a strong argument in favor of user-level threads packages.



Kernel Level:

Kernel knows about and manages the threads. No run-time system is needed in each, as shown in Fig. Also, there is no thread table in each process. Instead, the kernel has a thread table that keeps track of all the threads in the system. When a thread wants to create a new thread or destroy an existing thread, it makes a kernel call, which then does the creation or destruction by updating the kernel thread table.

The kernel's thread table holds each thread's registers, state, and other information. The information is the same as with user-level threads, but it is now in the kernel instead of in user space (inside the run-time system). This information is a subset of the information that traditional kernels maintain about each of their single-threaded processes, that is, the process state. In addition, the kernel also maintains the traditional process table to keep track of processes.

All calls that might block a thread are implemented as system calls, at considerably greater cost than a call to a run-time system procedure. When a thread blocks, the kernel, at its option, can run either another thread from the same process (if one is ready), or a thread from a different process. With user- level threads, the run-time system keeps running threads from its own process until the kernel takes the CPU away from it (or there are no ready threads left to run).

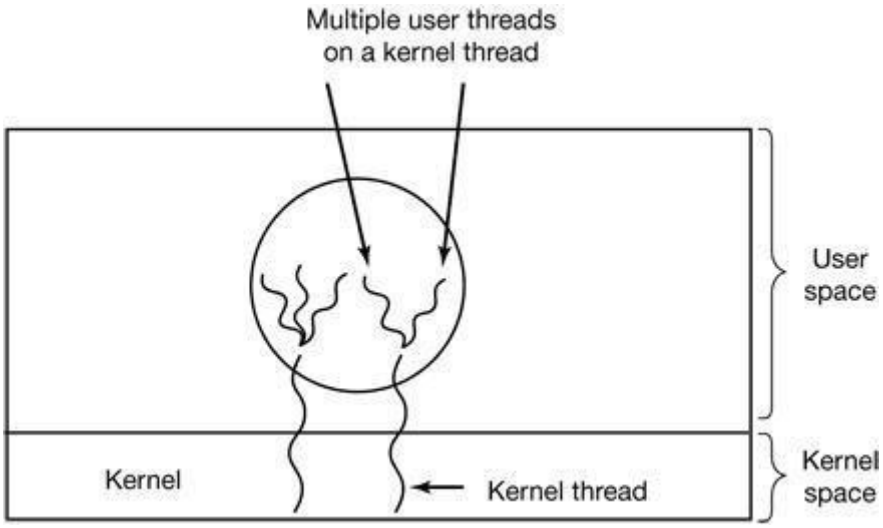
Due to the relatively greater cost of creating and destroying threads in the kernel, some systems take an environmentally correct approach and recycle their threads. When a thread is destroyed, it is marked as not runnable, but its kernel data structures are not otherwise affected. Later, when a new thread must be created, an old thread is reactivated, saving some overhead. Thread recycling is also possible for user-level threads, but since the thread management overhead is much smaller, there is less incentive to do this.

Kernel threads do not require any new, nonblocking system calls. In addition, if one thread in a process causes a page fault, the kernel can easily check to see if the process has any other runnable threads, and if so, run one of them while waiting for the required page to be brought in from the disk. Their main disadvantage is that the cost of a system call is substantial, so if

thread operations (creation, termination, etc.) are common, much more overhead will be incurred.

Hybrid Level

Various ways have been investigated to try to combine the advantages of user-level threads with kernel-level threads. One way is use kernel-level threads and then multiplex user-level threads onto some or all of the kernel threads, as shown in Fig. below.



In this design, the kernel is aware of only the kernel-level threads and schedules those. Some of those threads may have multiple user-level threads multiplexed on top of them. These user-level threads are created, destroyed, and scheduled just like user-level threads in a process that runs on an operating system without multithreading capability. In this model, each kernel-level thread has some set of user- level threads that take turns using it.

CPU Scheduling

Basic Concepts

Maximum CPU utilization obtained with multiprogramming

CPU–I/O Burst Cycle – Process execution consists of a *cycle* of CPU execution and I/O wait CPU burst distribution

CPU Scheduler

Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them

CPU scheduling decisions may take place when a process:

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

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

Dispatcher

Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:

- switching context
- switching to user mode
- jumping to the proper location in the user program to restart that program

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

Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **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

The CPU scheduler selects a process from among the ready processes to execute on the CPU. CPU scheduling is the basis for multi-programmed operating systems. CPU utilization increases by switching the CPU among ready processes instead of waiting for each process to terminate before executing the next. The idea of multi-programming could be described as follows: A process is executed by the CPU until it completes or goes for an I/O. In simple systems with no multi- programming, the CPU is idle till the process completes the I/O and restarts execution. With multiprogramming, many ready processes are maintained in memory. So when CPU becomes idle as in the case above, the operating system switches to execute another process each time a current process goes into a wait for I/O. Scheduling is a fundamental operating-system function. Almost all computer resources are scheduled before use. The CPU is, of course, one of the primary computer resources. Thus, its scheduling is central to operating system design.

Basic concepts of Scheduling

CPU- I/O Burst Cycle

Process execution consists of alternate CPU execution and I/O wait. A cycle of these two events repeats till the process completes execution (Figure 1). Process execution begins with a CPU burst followed by an I/O burst and then another CPU burst and so on. Eventually, a CPU burst will terminate the execution. An I/O bound job will have short CPU bursts and a CPU bound job will have long CPU bursts.

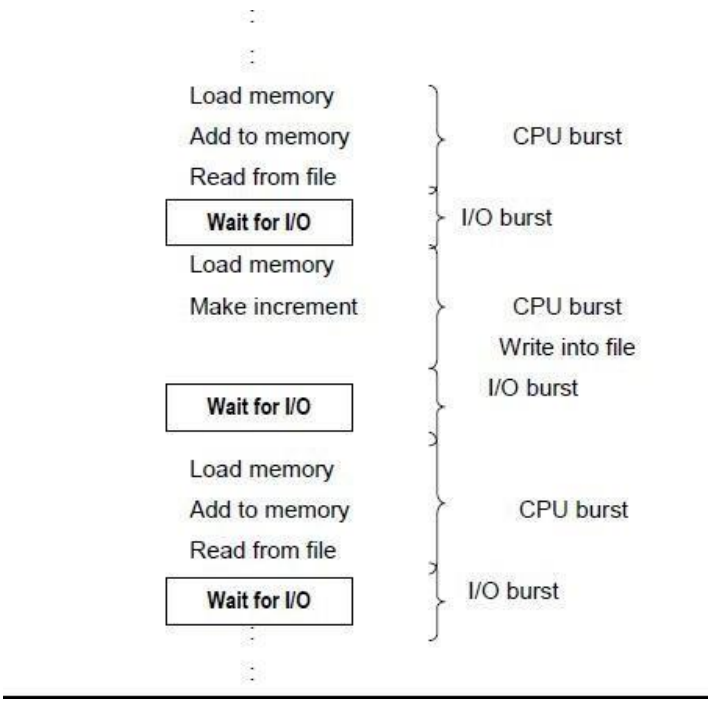


Figure 1: CPU and I/O bursts

CPU Scheduler

Whenever the CPU becomes idle, the operating system must select one of the processes in the

ready queue to be executed. The short-term scheduler (or CPU scheduler) carries out the selection process. The scheduler selects from among the processes in memory that are ready to execute, and allocates

the CPU to one of them. Note that the ready queue is not necessarily a first-in, first-out (FIFO) queue. As we shall see when we consider the various scheduling algorithms, a ready queue may be implemented as a FIFO queue, a priority queue, a tree, or simply an unordered linked list. Conceptually, however, all the processes in the ready queue are lined up waiting for a chance to run on the CPU. The records in the queue are generally PCBs of the processes.

Preemptive/ Non preemptive scheduling

CPU scheduling decisions may take place under the following four circumstances. When a process:

1. switches from running state to waiting (an I/O request).
2. switches from running state to ready state (expiry of a time slice).
3. switches from waiting to ready state (completion of an I/O).
4. terminates.

Scheduling under condition (1) or (4) is said to be non-preemptive. In non-preemptive scheduling, a process once allotted the CPU keeps executing until the CPU is released either by a switch to a waiting state or by termination. Preemptive scheduling occurs under condition (2) or (3). In preemptive scheduling, an executing process is stopped executing and returned to the ready queue to make the CPU available for another ready process. Windows used non-preemptive scheduling up to Windows 3.x, and started using pre-emptive scheduling with Win95. Note that pre-emptive scheduling is only possible on hardware that supports a timer interrupt. It is to be noted that pre- emptive scheduling can cause problems when two processes share data, because one process may get interrupted in the middle of updating shared data structures. Preemption also has an effect on the design of the operating-system kernel. During the processing of a system call, the kernel may be busy with an active on behalf *of* a process. Such activities may involve changing important kernel data (for instance, I/O queues). What happens if the process is preempted in t1: middle of these changes, and the kernel (or the device driver) needs to read (modify the same structure). Chaos ensues. Some operating systems, including most versions of UNIX, deal with this problem by waiting either for a system call to complete, or for an I/O block to take place, before doing a context switch. This scheme ensures that the kernel structure is simple, since the kernel will not preempt a process while the kernel data structures are in an inconsistent state. Unfortunately, this kernel execution model is a poor one for supporting real-time computing and multiprocessing.

Dispatcher

Another component involved in the CPU scheduling function is the *dispatcher*. The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler. This function involves:

Switching context

Switching to user

mode

Jumping to the proper location in the user program to restart that program

The dispatcher should be as fast as possible, given that it is invoked during every process switch. The time it takes for the dispatcher to stop one process and start another running is known as the ***dispatch latency.***

Scheduling Criteria

Many algorithms exist for CPU scheduling. Various criteria have been suggested for comparing these CPU scheduling algorithms. Common criteria include:

1. **CPU utilization:** We want to keep the CPU as busy as possible. CPU utilization may range from 0% to 100% ideally. In real systems it ranges, from 40% for lightly loaded systems to 90% for heavily loaded systems.
2. **Throughput:** Number of processes completed per time unit is throughput. For long processes may be of the order of one process per hour whereas in case of short processes, throughput may be 10 or 12 processes per second.
3. **Turnaround time:** The interval of time between submission and completion of a process is called turnaround time. It includes execution time and waiting time.
4. **Waiting time:** Sum of all the times spent by a process at different instances waiting in the ready queue is called waiting time.
5. **Response time:** In an interactive process the user is using some output generated while the process continues to generate new results. Instead of using the turnaround time that gives the difference between time of submission and time of completion, response time is sometimes used. Response time is thus the difference between time of submission and the time the first response occurs Desirable features include maximum CPU utilization, throughput and minimum turnaround time, waiting time and response time.

Scheduling Algorithms

Scheduling algorithms differ in the manner in which the CPU selects a process in the ready queue for execution.

First Come First Served scheduling algorithm:

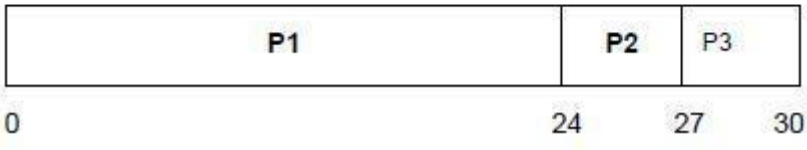
This is one of the very brute force algorithms. A process that requests for the CPU first is allocated the CPU first. Hence, the name first come first serve. The FCFS algorithm is implemented by using a first-in-first-out (FIFO) queue structure for the ready queue. This queue has a head and a tail. When a process joins the ready queue its PCB is linked to the tail of the FIFO queue. When the CPU is idle, the process at the head of the FIFO queue is allocated the CPU and deleted from the queue.

Even though the algorithm is simple, the average waiting is often quite long and varies substantially if the CPU burst times vary greatly, as seen in the following example.

Consider a set of three processes P1, P2 and P3 arriving at time instant 0 and having CPU burst times as shown below:

Process	Burst time (msecs)
P1	24
P2	3
P3	3

The Gantt chart below shows the result.



Average waiting time and average turnaround time are calculated as

follows: The waiting time for process

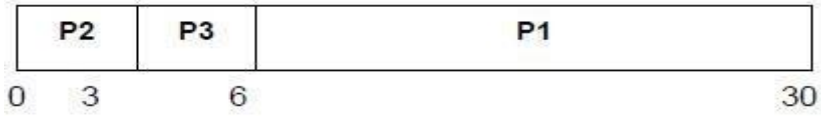
- P1 = 0 msecs
- P2 = 24
- msecs P3 =
- 27 msecs

Average waiting time = (0 + 24 + 27) / 3 = 51 / 3 = 17 msecs.

P1 completes at the end of 24 msecs, P2 at the end of 27 msecs and P3 at the end of 30 msecs.

Average turnaround time = (24 + 27 + 30) / 3 = 81 / 3 = 27 msecs.

If the processes arrive in the order P2, P3 and P3, then the result will be as follows:



Average waiting time = (0 + 3 + 6) / 3 = 9 / 3 = 3 msecs.

Average turnaround time = (3 + 6 + 30) / 3 = 39 / 3 = 13 msecs.

Thus, if processes with smaller CPU burst times arrive earlier, then average waiting and average turnaround times are lesser. The algorithm also suffers from what is known as a convoy effect. Consider the following scenario. Let there be a mix of one CPU bound process and many I/O bound processes in the ready queue. The CPU bound process gets the CPU and executes (long I/O burst). In the meanwhile, I/O bound processes finish I/O and wait for CPU, thus leaving the I/O devices idle.

The CPU bound process releases the CPU as it goes for an I/O. I/O bound processes have short CPU bursts and they execute and go for I/O quickly. The CPU is idle till the CPU bound process finishes the I/O and gets hold of the CPU. The above cycle repeats. This is called the convoy effect. Here small processes wait for one big process to release the CPU. Since the algorithm is non-preemptive in nature, it is not suited for time sharing systems.

Shortest-Job- First 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:

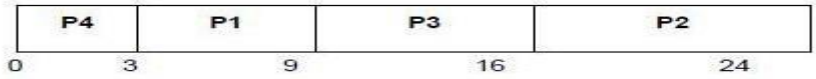
- Non preemptive – once CPU given to the process it cannot be preempted until completes its CPU burst
- Preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is know as the Shortest-Remaining-Time-First (SRTF)

In this algorithm, the length of the CPU burst is considered. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. Hence the name shortest job first. In case there is a tie, FCFS scheduling is used to break the tie. As an example, consider

the following set of processes P1, P2, P3, P4 and their CPU burst times:

Process	Burst time (msecs)
P1	6
P2	8
P3	7
P4	3

Using SJF algorithm, the processes would be scheduled as shown below.



Average waiting time = (0 + 3 + 9 + 16) / 4 = 28 / 4 = 7 msecs.

Average turnaround time = (3 + 9 + 16 + 24) / 4 = 52 / 4 = 13 msecs.

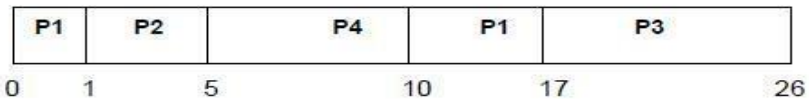
If the above processes were scheduled using FCFS algorithm, then Average waiting time = (0 + 6 + 14 + 21) / 4 = 41 / 4 = 10.25 msecs. Average turnaround time = (6 + 14 + 21 + 24) / 4 = 65 / 4 = 16.25 msecs.

The SJF algorithm produces the most optimal scheduling scheme. For a given set of processes, the algorithm gives the minimum average waiting and turnaround times. This is because, shorter processes are scheduled earlier than longer ones and hence waiting time for shorter processes decreases more than it increases the waiting time of long processes. The main disadvantage with the SJF algorithm lies in knowing the length of the next CPU burst. In case of long-term or job scheduling in a batch system, the time required to complete a job as given by the user can be used to schedule. SJF algorithm is therefore applicable in long-term scheduling. The algorithm cannot be implemented for CPU scheduling as there is no way to accurately know in advance the length of the next CPU burst. Only an approximation of the length can be used to implement the algorithm.

But the SJF scheduling algorithm is provably optimal and thus serves as a benchmark to compare other CPU scheduling algorithms. SJF algorithm could be either preemptive or non-preemptive. If a new process joins the ready queue with a shorter next CPU burst then what is remaining of the current executing process, then the CPU is allocated to the new process. In case of non-preemptive scheduling, the current executing process is not preempted and the new process gets the next chance, it being the process with the shortest next CPU burst. Given below are the arrival and burst times of four processes P1, P2, P3 and P4.

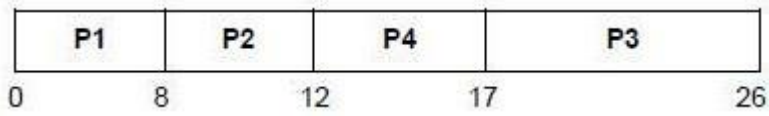
Process	Arrival time (msecs)	Burst time (msecs)
P1	0	8
P2	1	4
P3	2	9
P4	3	5

If SJF preemptive scheduling is used, the following Gantt chart shows the result.



Average waiting time = ((10 – 1) + 0 + (17 – 2) + (15 – 3)) / 4 = 26 / 4 = 6.5 msecs.

If non-preemptive SJF scheduling is used, the result is as follows:



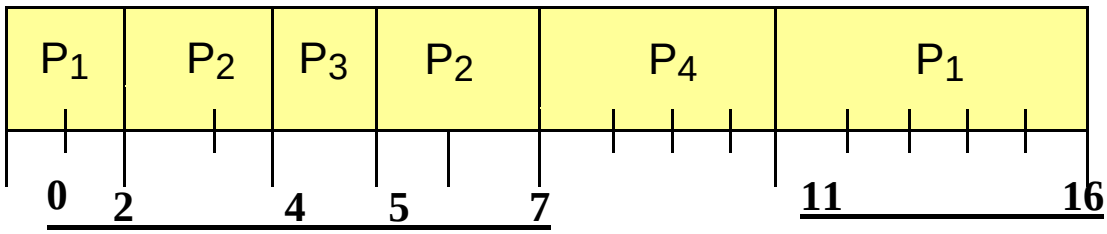
Average waiting time = ((0 + (8 – 1) + (12 – 3) + (17 – 2)) / 4 = 31 / 4 = 7.75 msecs.

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

✓ **Example of Non-Preemptive SJF (Shortest Remaining Time First /SRTF)**

Process	Arrival Time	Burst Time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	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
no 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

Round Robin (RR)

Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.

If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once. No process waits more than (*n*- 1)*q* time units.

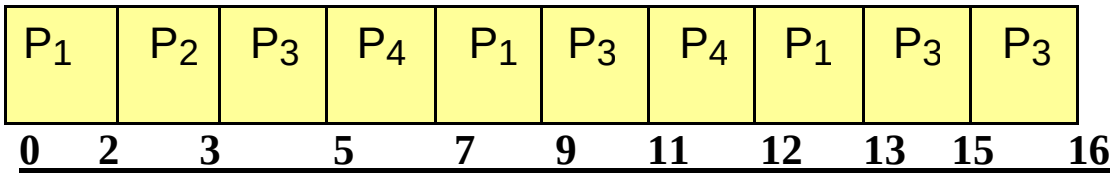
Performance

- *q* large FIFO
- *q* small *q* must be large with respect to context switch, otherwise overhead is too high

Example of RR with Time Quantum = 20

Process	Burst Time
P1	53
P2	17
P3	68
P4	24

The Gantt chart is



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

Multilevel Queue

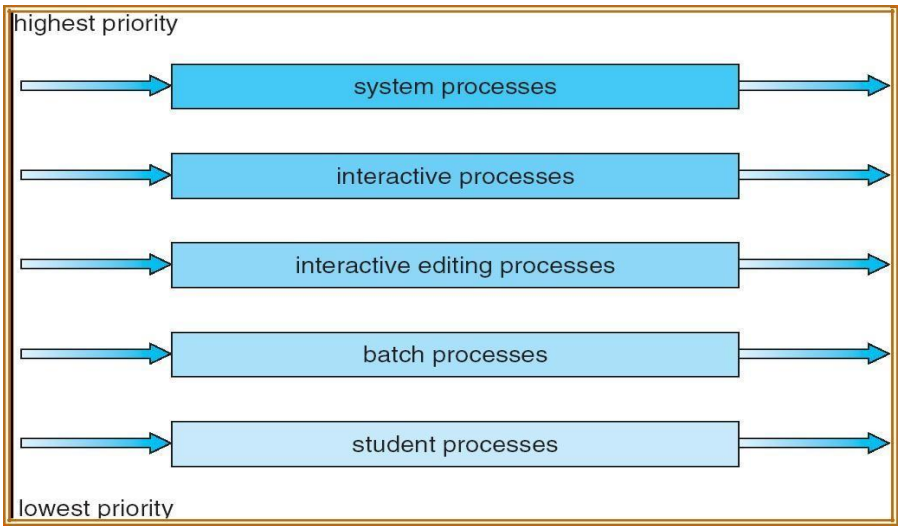
Ready queue is partitioned into separate queues: foreground(interactive)background (batch).

Each queue has its own scheduling algorithm

- foreground – RR
- background – FCFS

Scheduling must be done between the queues

- Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
- Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
- 20% to background in FCFS



Multilevel Queue Scheduling

Multilevel Feedback Queue

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

Example of Multilevel Feedback Queue

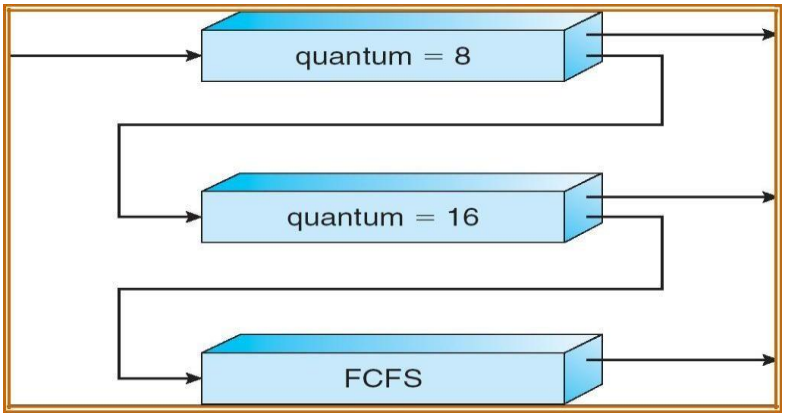
Three queues:

- *Q*0 – RR with time quantum 8 milliseconds
- *Q*1 – RR time quantum 16 milliseconds
- *Q*2 – FCFS

Scheduling

- A new job enters queue *Q0* which is served FCFS. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue *Q*1.
- At *Q*1 job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue *Q*2.

Multilevel Feedback Queues



Multiple-Processor Scheduling

PU scheduling more complex when multiple CPUs are available.
omogeneous processors within a multiprocessor.
oad sharing.
Asymmetric multiprocessing – only one processor accesses the system data structures, alleviating the need for data sharing.

Questions:

1) What is process?
2) What are different states of process? Sketch process state transition diagram?
3) Explain process control block?
4) What is context –switching?
5) What are CPU schedulers? What are different types of schedulers?
6) Explain mid-term schedulers, short -term schedulers and long-term schedulers in detail.
7) Explain performance criteria for the selection of schedulers.
8) Explain in brief evolution method for CPU scheduling algorithm.
9) Explain FCFS CPU scheduling algorithm.
10) Explain SJF CPU scheduling algorithm.
11) Explain SRTM scheduling algorithm.
12) Explain priority based non preemptive scheduling algorithm.
13) Explain priority based preemptive scheduling algorithm.
14) Explain round robin scheduling

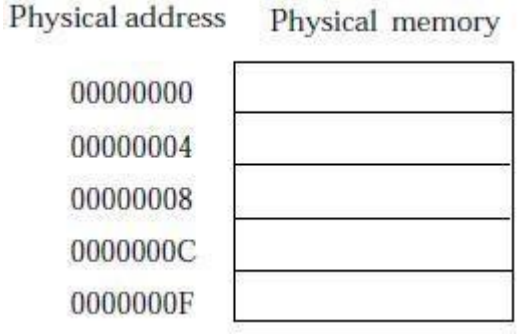
Memory Management

Memory:

1. Registers (all computations take place here)
2. Cache (all frequently accessed data may reside here)
3. Primary Memory
4. Mass Memory (All archiving is done here).

Physical Characteristics of Primary Memory:

1. A linear list of words (Size is defined by the Hardware).
2. Each word is associated with a unique address (absolute).



3. Random access possible.
4. Reading time - fixed (1 cycle: 80 ns)
5. Writing time - fixed (1 and a half cycle: 120 ns)
6. Address type - physical (Absolute address).
7. Memory size can be expanded dynamically.

Data movement between

Disk Cache. Primary memory disk. Registers cache. Registers primary memory

Introduction:

The concept of CPU scheduling allows a set of processes to share the CPU thereby increasing the utilization of the CPU. This set of processes needs to reside in memory. The memory is thus shared and the resource requires to be managed. Various memory management algorithms exist, each having its own advantages and disadvantages. The hardware design of the system plays an important role in the selection of an algorithm for a particular system.
That means to say, hardware support is essential to implement the memory management algorithm.

Machine with no resident monitor

1. The entire memory belongs to one program.
2. The operator loads the program and provides commands to execute the program via some keys.
4. Outputs are distributed manually.
5. Such system has limitations and cannot satisfy today's processing requirements.

Machine with resident monitor:

The operator's functions are stored in the system as resident monitor. To do this the entire memory is divided into monitor part and user part. The monitor is loaded permanently into its portion of the memory. A program is loaded into the primary memory for execution.

Requirement: The monitor must be protected from the user program.

Solution: The only way this can be done is by precisely defining the higher boundary (end address) of the monitor portion of memory. A fixed memory location (fence word) is used to hold this address. Any memory access is checked against the contents of the fence word and access is granted only if the desired address is larger.

Limitations

a. Monitor cannot grow and shrink.
b. A fence word must be chosen carefully.
c. User program must be loaded at a fixed location.
d. If program size > memory then program cannot be executed.
e. Program must be recompiled whenever any change is made.

Limitation

1. A program must be loaded at the compiled addresses.
2. If the monitor grows and shrinks then program must be recompiled before execution.

Swapping:

A process to be executed needs to be in memory during execution. A process can be swapped out of memory in certain situations to a backing store and then brought into memory later for execution to continue. One such situation could be the expiry of a time slice if the round-robin CPU scheduling algorithm is used. On expiry of a time slice, the current process is swapped out of memory and another process is swapped into the memory space just freed because of swapping out of a process (Figure 2). Every time a time slice expires, a process is swapped out and another is swapped in. The memory manager that does the swapping is fast enough to always provide a process in memory for the CPU to execute. The duration of a time slice is carefully chosen so that it is sufficiently large when compared to the time for swapping.

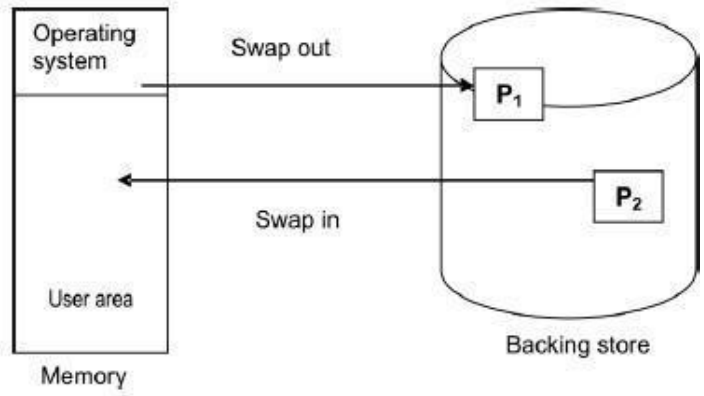


Figure 1: Swapping

Processes are swapped between the main memory and the backing store when priority based CPU scheduling is used. The arrival of a high priority process will be a lower priority process that is executing to be swapped out to make way for a swap in. The swapping in this case is sometimes referred to as roll out / roll in.

A process that is swapped out of memory can be swapped in either into the same memory location or into a different memory location. If binding is done at load time then swap in has to be at the same location as before. But if binding is done at execution time then swap in can be into a different memory space since the mappings to physical addresses are completed during execution time.

A backing store is required for swapping. It is usually a fast disk. The processes in the ready queue have their images either in the backing store or in the main memory. When the CPU scheduler picks a process for execution, the dispatcher checks to see if the picked process is in memory? If yes, then it is executed. If not the process has to be loaded into main memory. If there is enough space in memory, the process is loaded and execution starts. If not, the dispatcher swaps out a process from memory and swaps in the desired process.

A process to be swapped out must be idle. Problems arise because of pending I/O. Consider the following scenario: Process P1 is waiting for an I/O. I/O is delayed because the device is busy. If P1 is swapped out and its place P2 is swapped in, then the result of the I/O uses the memory that now belongs to P2. There can be two solutions to the above problem:

1. Never swap out a process that is waiting for an I/O

2. I/O operations to take place only into operating system buffers and not into user area buffers. These buffers can then be transferred into user area when the corresponding process is swapped in.

Relocation:

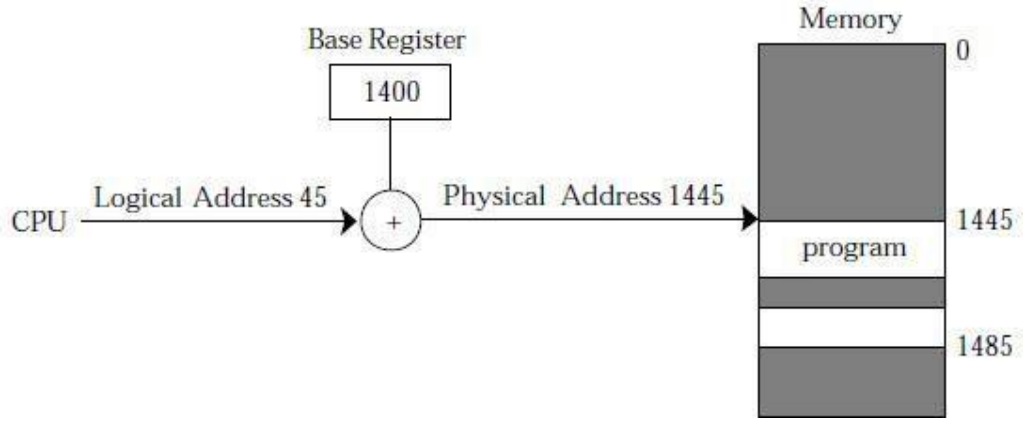
Facility to relocate the program anywhere in the memory.

Requirements of relocation:

1. The addresses assigned to the program by the compiler should be independent to physical addresses. It should compile every program address starting from **0**. This means the compiler should generate logical addresses.
2. Program addresses should be relative to a fixed location.
3. The compiler should not load the program in the memory.

Implementation

1. Define a hardware register (relocation/base register). This will achieve the aim no. 2 above.
2. Convert the logical addresses generated by the CPU, using relocation register, to get the physical address of the required location. The CPU then accesses the contents of this location and continues with the program.



Observation

Logical address: program instruction address

Physical address: Real address of the program in memory.

Mapping from logical to physical:
by the base/relocation register

Swapping: Consider the following situation -

- Job 1 is very large (50K).
- Job 2 is a small job (5K)
- Suppose Job 1 is in the memory and is under execution.
- Job 2 has to wait for Job 1 to finish.

What if Job 2 is a very high priority job? Result: job 2 must wait.

Improvement

- Swap the Job 1 out of the memory briefly.
- Schedule job 2 and let it finish then schedule Job 1.

Implementation:

This scheme, under the resident monitor can be implemented by saving the current state of Job 1 on the disk and Job 2 can be loaded for execution. Job 1 can be reloaded once Job 2 is finished. There is

some improvement, but we still could not run more than one job at a time.

Analysis: The main activity which affects the response time of a job is swapping

activity. Suppose Latency time - 8 milliseconds. Program size - 20K.

Transfer rate - 250,000 words/second = 250 words/ms. Transfer time (20K) - 20,000/250

= 80 ms Total transfer time - 8 (latency time) + 80 = 88 ms = 0.088 secs.

Number of Swaps: Minimum number of movement between the memory and the disk is 2. In its lifetime a job may be swapped more than 2 and may have a large swapping time. Some efficiency can be gained by making the CPU time for a process longer than the swap time. In this way the CPU will not remain idle for a longer period of time. Under this scenario the CPU time should be larger than 0.176 seconds (0.088 x 2).

How much to swap: Another improvement can be achieved by computing the exact swap amount, i.e., the size of the program in the memory. It is not necessary to swap the entire user area, since the active program may not be as large as 20/30 K (the size of user memory). Also the swap time can be reduced by improving the disk hardware.

Logical versus Physical Address Space:

An address generated by the CPU is referred to as a logical address. An address seen by the memory unit, that is, the address loaded into the memory address register (MAR) of the memory for a fetch or a store is referred to as a physical address. The logical address is also sometimes referred to as a virtual address. The set of logical addresses generated by the CPU for a program is called the logical address space. The set of all physical addresses corresponding to a set of logical address is called the physical address space. At run time / execution time, the virtual addresses are mapped to physical addresses by the memory management unit (MMU). A simple mapping scheme is shown in Figure 1. The relocation register contains a value to be added to every address generated by the CPU for a user process at the time it is sent to the memory for a fetch or a store. For example, if the base is at 14000 then an address 0 is dynamically relocated to location 14000, an access to location 245 is mapped to 14245 (14000 + 245). Thus, every address is relocated relative to the value in the relocation register. The hardware of the MMU maps logical addresses to physical addresses. Logical addresses range from 0 to a maximum (MAX) and the corresponding physical addresses range from (R + 0) to (R + MAX) for a base value of R. User programs generate only logical addresses that are mapped to physical addresses before use.

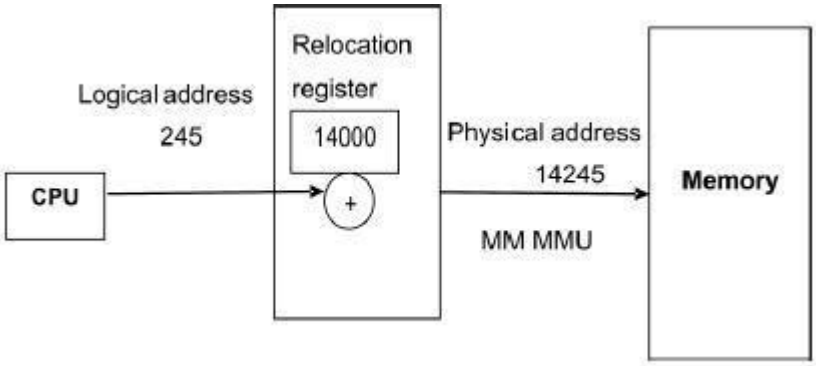


Figure 2: Dynamic Relocation

Contiguous Allocation:

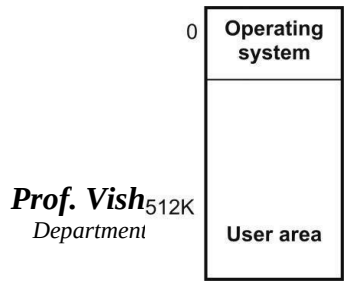


Figure 3: Contiguous Allocation

The main memory is usually divided into two partitions, one of which has the resident operating system loaded into it. The other partition is used for loading user programs. The operating system is usually present in the lower memory because of the presence of the interrupt vector in the lower memory (Figure 3).

Single Partition Allocation:

The operating system resides in the lower memory. User processes execute in the higher memory. There is always a possibility that user processes may try to access the lower memory either accidentally or intentionally thereby causing loss of operating system code and data. This protection is usually provided by the use of a limit register and relocation register. The relocation register contains the smallest physical address that can be accessed. The limit register contains the range of logical addresses. Each logical address must be less than the content of the limit register. The MMU adds to the logical address the value in the relocation register to generate the corresponding address (Figure 4). Since an address generated by the CPU is checked against these two registers, both the operating system and other user programs and data are protected and are not accessible by the running process.

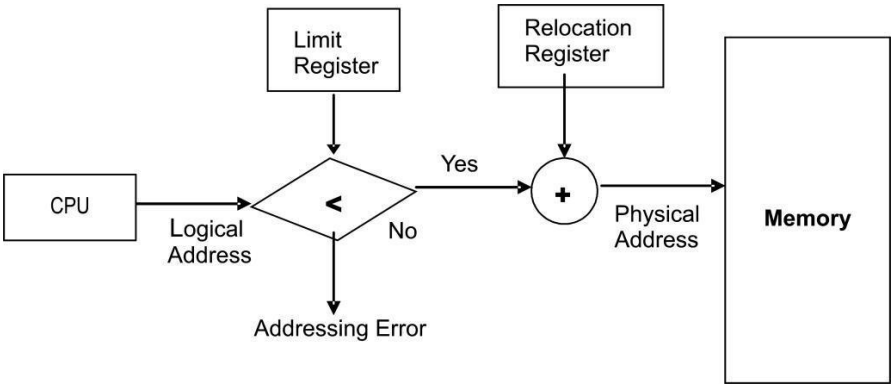


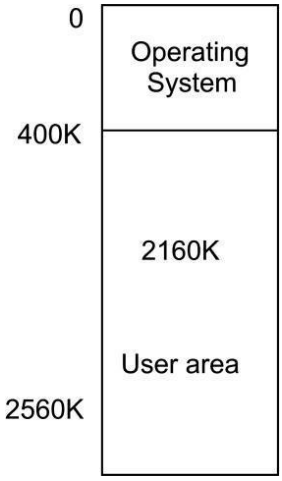
Figure 4: Hardware for relocation and limit register

Multiple Partition Allocation:

Multi-programming requires that there are many processes residing in memory so that the CPU can switch between processes. If this has to be so, then, user area of memory has to be divided into partitions. The simplest way is to divide the user area into fixed number of partitions, each one to hold one user process. Thus, the degree of multi- programming is equal to the number of partitions. A process from the ready queue is loaded into one of the partitions for execution. On termination the partition is free for another process to be loaded.

The disadvantage with this scheme where partitions are of fixed sizes is the selection of partition sizes. If the size is too small then large programs cannot be run. Also, if the size of the partition is big then main memory space in each partition goes a waste.

A variation of the above scheme where the partition sizes are not fixed but variable is generally used. A table keeps track of that part of the memory that is used and the part that is free. Initially, the entire memory of the user area is available for user processes. This can be visualized as one big hole for use. When a process is loaded a hole big enough to hold this process is searched. If one is found then memory enough for this process is allocated and the rest is available free as illustrated in Figure 5.



Job Queue

Process	Memory	Time
P_1	600K	10
P_2	1000K	5
P_3	300K	20
P_4	700K	8
P_5	500K	15

Figure 5: Scheduling example

Total memory available:
2560K Resident operating
system: 400K
Memory available for user: 2560 – 400 =
2160K Job queue: FCFS
CPU scheduling: RR (1 time unit)

Given the memory map in the illustration above, P_1, P_2, P_3 can be allocated memory immediately. A hole of size (2160 – (600 + 1000 + 300)) = 260K is left over which cannot accommodate P_4 (Figure 6 A). After a while P_2 terminates creating the map of Figure 6 B. P_4 is scheduled in the hole just created resulting in Figure 6 C. Next P_1 terminates resulting in Figure 6 D and P_5 is scheduled as in Figure 6 E.

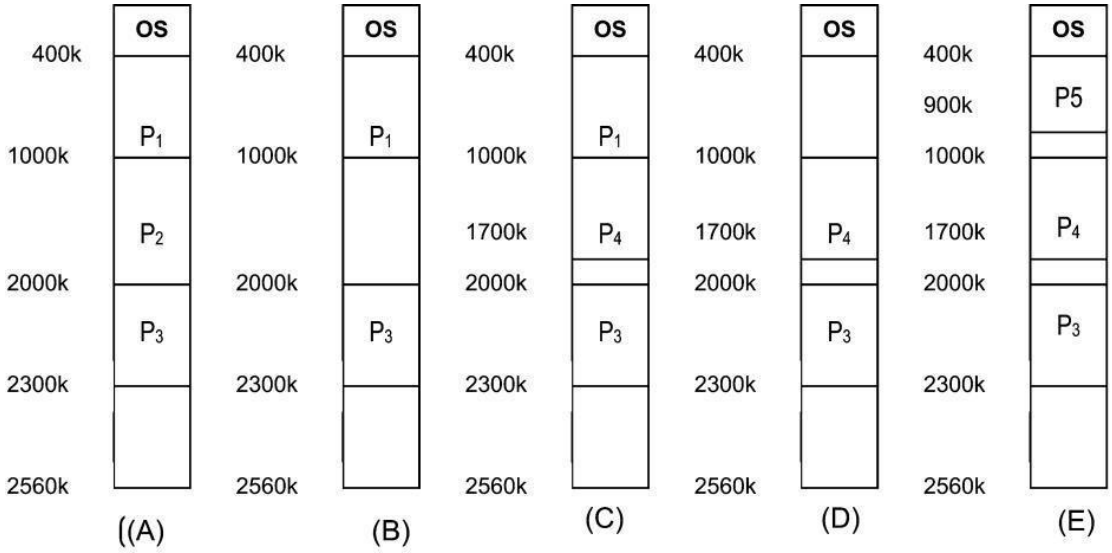


Figure 6 : Memory allocation and job scheduling

The operating system finds a hole just large enough to hold a process and uses that particular hole to load the process into memory. When the process terminates, it releases the used memory to create a hole equal to its memory requirement. Processes are allocated memory until the free memory or hole available is not big enough to load another ready process. In such a case the operating system waits for some process to terminate and free memory. To begin with, there is one large big hole equal to the size of the user area. As processes are allocated into this memory, execute and terminate this hole gets divided. At any given instant thereafter there are a set of holes scattered all over the memory. New holes created that are adjacent to existing holes merge to form big holes.

The problem now is to satisfy a memory request of size n from a list of free holes of various sizes. Many solutions exist to determine that hole which is the best to allocate.

Most common strategies are:

1. **First-fit:** Allocate the first hole that is big enough to hold the process. Search can either start at the beginning of the set of holes or at the point where the last search terminated.

2. **Best-fit:** Allocate the smallest hole that is big enough to hold the process. Here the entire list has to be searched or an ordered list of holes by size is to be maintained.

3. **Worst-fit:** Allocate the largest hole available. Here also, the entire list has to be searched for the biggest hole or an ordered list of holes by size is to be maintained.

4. **Next fit: T**he search pointer does not start at the top (beginning), instead it begins from where it ended during the previous search. it locates the next first-fit hole that can be used. Note that unlike the first-fit policy the next-fit policy can be expected to distribute small holes uniformly in the main memory.

Once processes have been created, the OS organizes their execution. This requires interaction between process management and main memory management. To understand this interaction better, we shall create a scenario requiring memory allocations. For the operating environment we assume the following:

A uni-processor, multi-programming operation.

A Unix like operating system environment.

With a Unix like OS, we can assume that main memory is partitioned in two parts. One part is for user processes and the other is for OS. We will assume that we have a main memory of 20 units (for instance it could be 2 or 20 or 200 MB). We show the requirements and time of arrival and processing requirements for 6 processes in Table below:

	P1	P2	P3	P4	P5	P6
Time of arrival	0	0	0	0	10	15
Processing time required	8	5	20	12	10	5
Memory required	3 units	7 units	2 units	4 units	2 units	2 units

The given data:

We shall assume that OS requires 6 units of space. To be able to compare various policies, we shall repeatedly use the data in Table for every policy option. We shall assume round robin allocation of processor time slots with no context switching over-heads. We shall trace the events as they occur giving reference to the corresponding part in Table 4.2. This table also shows a memory map as the processes move in and out of the main memory.

Time units	Programs in Main memory	Programs on disk	Holes with sizes	Figure 4.3	Comments
0	P1, P2, P3	P4	H1=2	(a)	P4 requires more space than H1
5	P1, P4, P3		H1=2; H2=3	(b)	P2 is finished P4 is loaded Hole H2 is created
8	P4, P3		H1=2; H2=3; H3=3	(c)	New hole created
10	P4, P3	P5			P5 arrives
10+	P5, P4, P3		H1=2; H2=3 H3=1	(d)	P5 is allocated P1's space
15	P5, P4, P3	P6	H1=2; H2=3; H3=1		P6 has arrived
15+	P5, P4, P6, P3		H1=2; H2=1; H3=1	(e)	P6 is allocated

FCFS Memory Allocation

The First Fit Policy: Memory Allocation:

In this example we make use of a policy called first fit memory allocation policy. The first fit policy suggests that we use the first available hole, which is large enough to accommodate an incoming process. In Figure below, it is important to note that we are following first-come first-served (process management) and first fit (memory allocation) policies. The process index denotes its place in the queue. As per first-come first-served policies the queue order determines the order in which the processes are allocated areas.

In addition, as per first-fit policy allocation we scan the memory always from one end and find the first block of free space which is large enough to accommodate the incoming process.

In our example, initially, processes P1, P2, P3 and P4 are in the queue. The allocations for processes P1, P2, P3 are shown in 7(a). At time 5, process P2 terminates. So, process P4 is allocated in the hole created by process P2. This is shown at 7(b) in the figure. It still leaves a hole of size 3. Now on advancing time further we see that at time 8, process P1 terminates. This creates a hole of size 3 as shown at 7(c) in the figure. This hole too is now available for allocation. We have 3 holes at this stage. Two of these 3 holes

are of size 3 and one is of size 2. When process P5 arrives at time 10, we look for the first hole which can accommodate it. This is the one created by the departure of process P1. Using the first-fit argument this is the hole allocated to process P5 as shown in Figure 7(d). The final allocation status is shown in Figure 7. The first-fit allocation policy is very easy to implement and is fast in execution.
Figure 7:

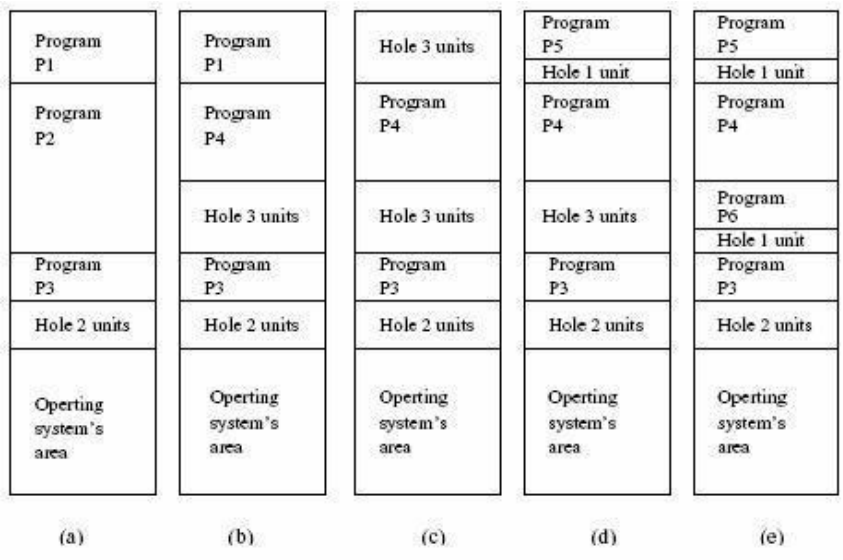


Figure 7: First-fit policy allocation.

The Best Fit Policy: Memory Allocation:

The main criticism of first-fit policy is that it may leave many smaller holes. For instance, let us trace the allocation for process P5. It needs 2 units of space. At the time it moves into the main memory there is a hole with 2 units of space. But this is the last hole when we scan the main memory from the top (beginning). The first hole is 3 units. Using the first-fit policy process P5 is allocated this hole. So when we used this hole we also created a still smaller hole. Note that smaller holes are less useful for future allocations. In the best-fit policy we scan the main memory for all the available holes. Once we have information about all the holes in the memory then we choose the one which is closest to the size of the requirement of the process. In our example we allocate the hole with size 2 as there is one available. Figure 8 follows best-fit policy for the current example.

Also, as we did for the previous example, we shall again assume round-robin allocation of the processor time slots. With these considerations we can now trace the possible emerging scenario. In Figure 8, we are following first-come first-served (process management) and best fit (memory allocation) policies. The process index denotes its place in the queue. Initially, processes P1, P2, P3 and P4 are in the queue. Processes P1, P2 and P3 are allocated as

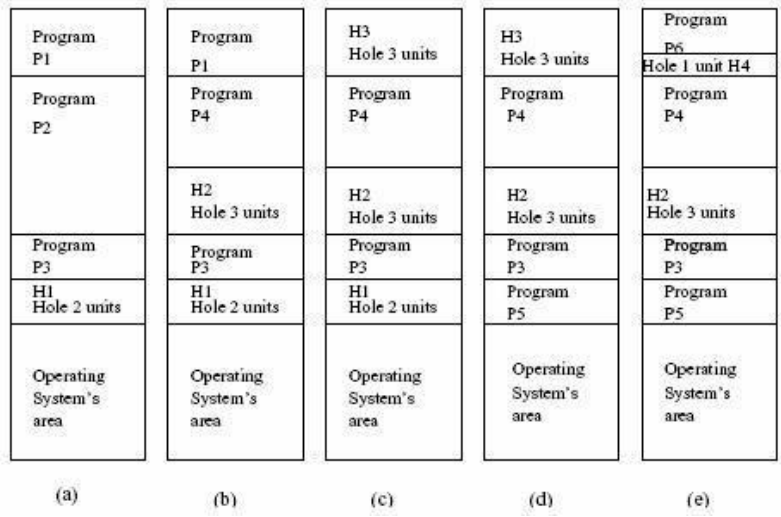


Figure 8: Best-fit policy allocation

shown in Figure 8(a). At time 5, P2 terminates and process P4 is allocated in the hole so created. This is shown in Figure 8(b). This is the best fit. It leaves a space of size 3 creating a new hole. At time 8, process P1 terminates. We now have 3 holes. Two of these holes are of size 3 and one is of size 2. When process P5 arrives at time 10, we look for a hole whose size is nearest to 2 and can accommodate P5. This is the last hole.

Clearly, the best-fit (and also the worst-fit) policy should be expected to be slow in execution. This is so because the implementation requires a time consuming scan of all of main memory. There is another method called the next-fit policy. In the next-fit method the search pointer does not start at the top (beginning), instead it begins from where it ended during the previous search. Like the first-fit policy it locates the next first-fit hole that can be used. Note that unlike the first-fit policy the next-fit policy can be expected to distribute small holes uniformly in the main memory. The first-fit policy would have a tendency to create small holes towards the beginning of the main memory scan. Both first-fit and next-fit methods are very fast and easy to implement. In conclusion, first-fit and next-fit are the fastest and seem to be the preferred methods. One of the important considerations in main memory management is: how should an OS allocate a chunk of main memory required by a process. One simple approach would be to somehow create partitions and then different processes could reside in different partitions. We shall next discuss how the main memory partitions may be created.

Fixed and Variable Partitions:

In a fixed size partitioning of the main memory all partitions are of the same size. The memory resident processes can be assigned to any of these partitions. Fixed sized partitions are relatively simple to implement. However, there are two problems. This scheme is not easy to use when a program requires more space than the partition size. In this situation the programmer has to resort to overlays. Overlays involve moving data and program segments in and out of memory essentially reusing the area in main memory.

The second problem has to do with internal fragmentation. No matter what the size of the process is, a fixed size of memory block is allocated as shown in Figure 9(a). So there will always be some space which will remain unutilized within the partition. In a variable-sized partition, the memory is partitioned into partitions with different sizes. Processes are loaded into the size nearest to its requirements. It is easy to always ensure the best-fit. One may organize a queue for each size of the partition as shown in the Figure 9(b). With best-fit policy, variable partitions minimize internal fragmentation.

However, such an allocation may be quite slow in execution. This is so because a process may end up waiting (queued up) in the best-fit queue even while there is space available elsewhere. For example, we may have several jobs queued up in a queue meant for jobs that require 1 unit of memory, even while no jobs are queued up for jobs that require say 4 units of memory.

Both fixed and dynamic partitions suffer from external fragmentation whenever there are partitions that have no process in it. One of techniques that have been used to keep both internal and external fragmentations low is dynamic partitioning. It is basically a variable partitioning with a variable number of partitions determined dynamically (i.e. at run time).

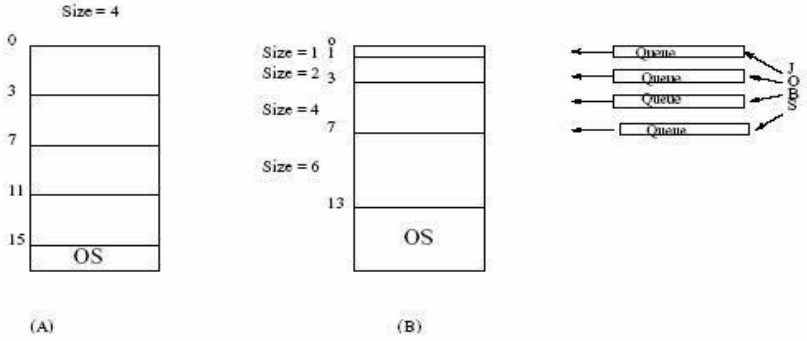


Figure 9: Fixed and variable sized partitions.

The size of a process is very rarely an exact size of a hole allocated. The best-fit allocation always produces an optimal allocation where the hole left over after allocation is the smallest. The first-fit is the fastest method of allocation when compared to others. The worst-fit allocation is the worst among the three and is seldom used.

Fragmentation:

To begin with, there is one large hole for allocation to processes. As processes are loaded into memory and terminate on completion, this large hole is divided into a set of smaller holes that are scattered in between the processes. There may be a situation where the total size of these scattered holes is large enough to hold another process for execution but the process cannot be loaded, as the hole is not contiguous. **This is known as external fragmentation.** For example, in Figure 6C, a fragmented hole equal to 560K (300 + 260) is available. P_5 cannot be loaded because 560K is not contiguous.

There are situations where only a few bytes say 1 or 2 would be free if a process were allocated a hole. Then, the cost of keeping track of this hole will be high. In such cases, this extra bit of hole is also allocated to the requesting process. If so then a small portion of memory allocated to a process is not useful. **This is internal fragmentation**.

One solution to external fragmentation is compaction. Compaction is to relocate processes in memory so those fragmented holes create one contiguous hole in memory (Figure 10).

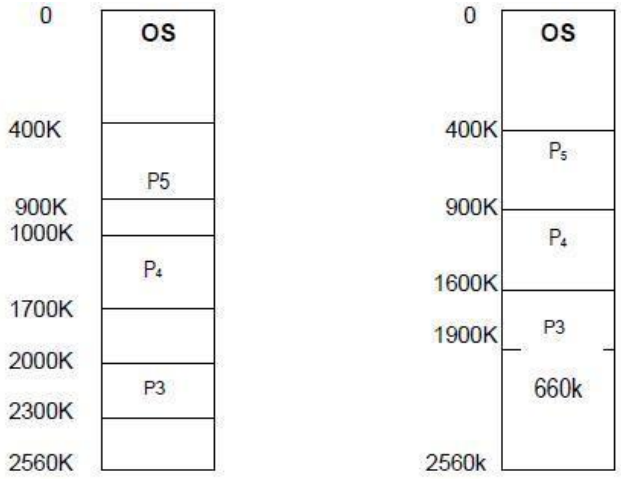


Figure 10: Compaction

Compaction may not always be possible since it involves relocation. If relocation is static at load time, then relocation is not possible and so also compaction. Compaction is possible only if relocation is done at runtime. Even though compaction is possible, the cost involved in relocation is to be considered. Sometimes creating a hole at one end of the user memory may be better whereas in some other cases a contiguous hole may be created in the middle of the memory at lesser cost. The position where the hole is to be created during compaction depends on the cost of relocating the processes involved. An optimal strategy if often difficult. Processes may also be rolled out and rolled in to affect compaction by making use of a back up store. But this would be at the cost of CPU time.

Paging:

The mechanism is based on the idea that at a time only one instruction of a program is required by the CPU. Furthermore, physical contiguity of program instructions is not necessary but they must have the logical contiguity. For example a JUMP or GOTO instruction breaks the physical contiguity of these instructions for correct execution of a program. Since a program is concerned only about the logical contiguity, the memory management is free to break the physical contiguity. A logical contiguity, usually, is provided by pointers. Linking instructions of a program via pointers to maintain logical contiguity is

impractical. Paging mechanism establishes the logical contiguity by restructuring the physical address space and the logical address space.

Physical address space: The entire physical memory.
Logical address space: All program referenced addresses.

Contiguous allocation scheme requires that a process can be loaded into memory for execution if and only if contiguous memory large enough to hold the process is available. Because of this constraint, external fragmentation is a common problem. Compaction was one solution to tide over external fragmentation. Another solution to this problem could be to permit noncontiguous logical address space so that a process can be allocated physical memory wherever it is present. This solution is implemented through the use of a paging scheme.

Concept of paging:

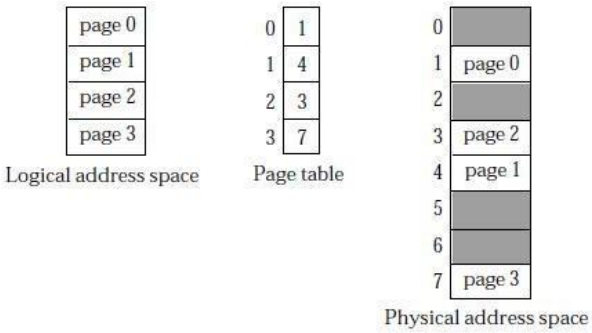


Figure 11: Concept of Paging

Physical memory is divided into fixed sized blocks called frames. Also logical memory is divided into blocks of the same size called pages. Allocation of main memory to processes for execution is then just mapping pages to frames. The hardware support for paging is illustrated below: (Figure 12).

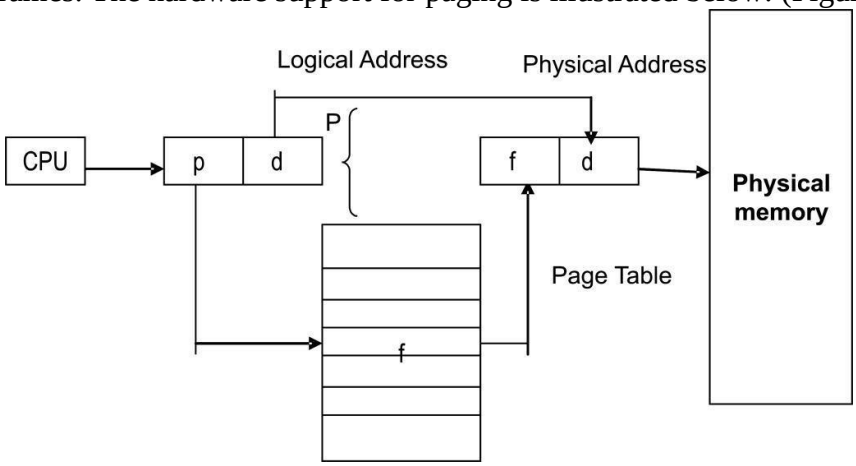


Figure 12: Paging Hardware

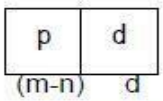
A logical address generated by the CPU consists of two parts: Page number (p) and a page offset (d). The page number is used as an index into a page table. The page table contains the base address of each frame in physical memory. The base address is combined with the page offset to generate the physical address required to access the memory unit.

Why size of page is usually in the power of 2:

The size of a page is usually a power of 2. This makes translation of a logical address into page number and offset easy as illustrated below:

Logical address space:
2m Page size: 2n

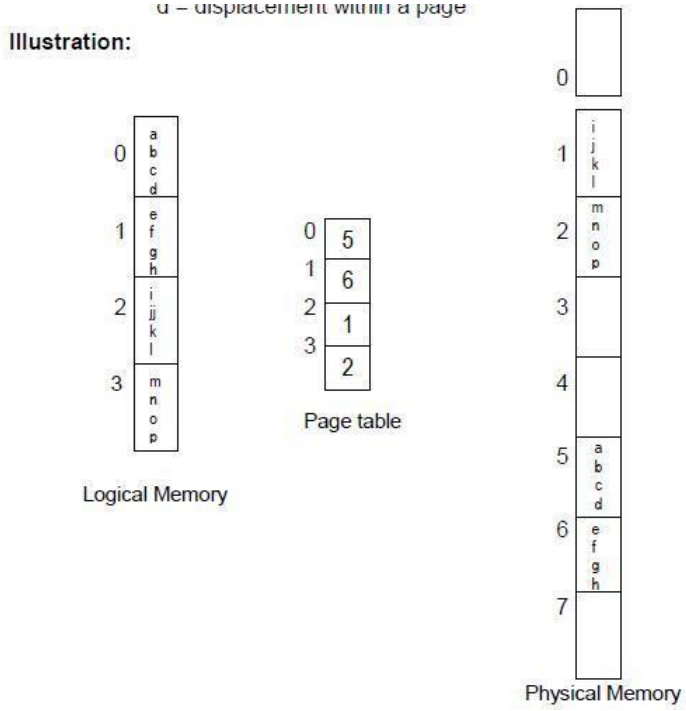
Logical address:



Where p = index into the page table
d = displacement within a page

d – displacement within a page

Illustration:



Page size: 4 bytes
Physical memory: 32 bytes = 8 pages

Logical address 0 ➔ 0 + 0 ➔ (5 x 4) + 0 ➔ physical address 20

3 ➔ 0 + 3 ➔ (5 x 4) + 3 ➔ physical address 23

4 ➔ 1 + 0 ➔ (6 x 4) + 0 ➔ physical address 24

13 ➔ 3 + 1 ➔ (2 x 4) + 1 ➔ physical address 9

Thus, the page table maps every logical address to some physical address. Paging does not suffer from external fragmentation since any page can be loaded into any frame. But internal fragmentation may be prevalent. This is because the total memory required by a process is not always a multiple of the page size. So the last page of a process may not be full. This leads to internal fragmentation and a portion of the frame allocated to this page will be unused. On an average one half of a page per process is wasted due to internal fragmentation. Smaller the size of a page, lesser will be the loss due to internal fragmentation. But the overhead involved is more in terms of number of entries in the page table. Also known is a fact that disk I/O is more efficient if page sizes are big. A trade-off between the above factors is used

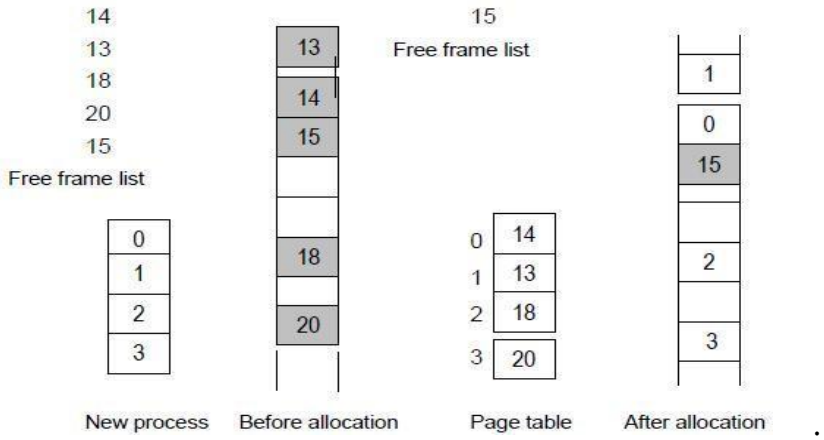


Figure 13: Frame Allocation

A process requires n pages of memory. Then at least n frames must be free in physical memory to allocate n pages. A list of free frames is maintained.

When allocation is made, pages are allocated the free frames sequentially (Figure 13). Allocation details

of physical memory are maintained in a frame table. The table has one entry for each frame showing whether it is free or allocated and if allocated, to which page of which process.

Page Table Implementation:

Hardware implementation of a page table is done in a number of ways. In the simplest case, the page table is implemented as a set of dedicated high-speed registers. But this implementation is satisfactory only if the page table is small. Modern computers allow the page table size to be very large. In such cases the page table is kept in main memory and a pointer called the page-table base register (PTBR) helps index the page table. The disadvantage with this method is that it requires two memory accesses for one CPU address generated. For example, to access a CPU generated address, one memory access is required to index into the page table. This access using the value in PTBR fetches the frame number when combined with the page-offset produces the actual address. Using this actual address, the next memory access fetches the contents of the desired memory location.

To overcome this problem, hardware caches called translation look-aside buffers (TLBs) are used. TLBs are associative registers that allow for a parallel search of a key item. Initially the TLBs contain only a few or no entries. When a logical address generated by the CPU is to be mapped to a physical address, the page number is presented as input to the TLBs. If the page number is found in the TLBs, the corresponding frame number is available so that a memory access can be made. If the page number is not found then a memory reference to the page table in main memory is made to fetch the corresponding frame number. This page number is then added

to the TLBs so that a mapping for the same address next time will find an entry in the table. Thus, a hit will reduce one memory access and speed up address translation (Figure 14).

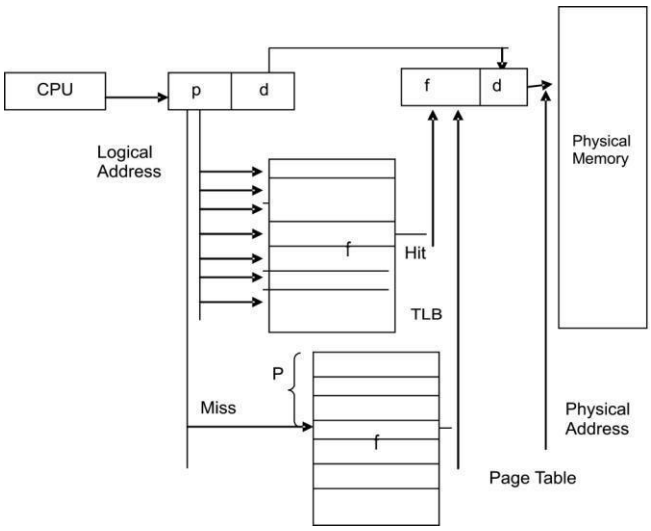


Figure 14: Paging hardware with TLB

Segmentation:

Memory management using paging provides two entirely different views of memory – User / logical / virtual view and the actual / physical view. Both are not the same. In fact, the user's view is mapped on to the physical view. How do users visualize memory? Users prefer to view memory as a collection of variable sized segments (Figure 15). The memory where his program is loaded is modularized as his program, i.e. segmented. The memory management technique that implements this view is called segmentation.

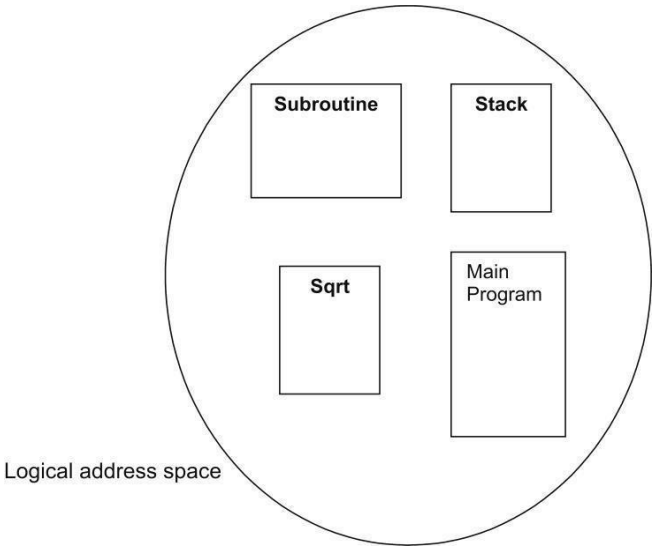


Figure 15: User's view of memory

Implementation:

- Segment is referred to by a segment number.
- During compilation the compiler constructs the segments.

The user usually writes a modular structured program consisting of a main segment together with a number of functions / procedures. Each one of the above is visualized as a segment with its associated name. Entries in a segment are at an offset from the start of the segment.

Concept of Segmentation:

Segmentation is a memory management scheme that supports users' view of main memory described above. The logical address is then a collection of segments, each having a name and a length. Since it is easy to work with numbers, segments are numbered. Thus a logical address is < segment number, offset >. User programs when compiled reflect segments present in the input. Loader while loading segments into memory assign them segment numbers.

Segmentation Hardware:

Even though segments in user view are same as segments in physical view, the two-dimensional visualization in user view has to be mapped on to a one-dimensional sequence of physical memory locations. This mapping is present in a segment table. An entry in a segment table consists of a base and a limit. The base corresponds to the starting physical address in memory whereas the limit is the length of the segment (Figure 16).

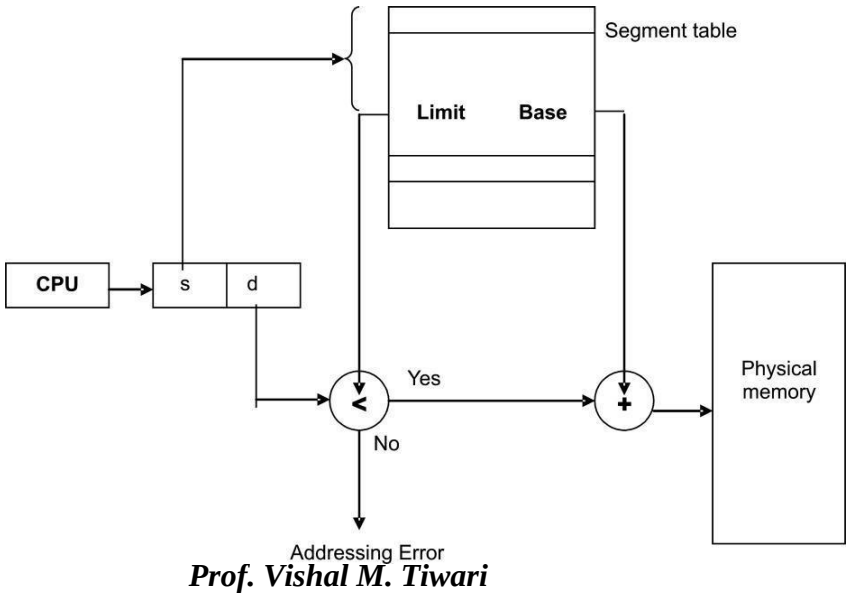


Figure 16: Segmentation Hardware

The logical address generated by the CPU consists of a segment number and an offset within the segment. The segment number is used as an index into a segment table. The offset in the logical address should lie between 0 and a limit specified in the corresponding entry in the segment table. If not the program is trying to access a memory location which does not belong to the segment and hence is trapped as an addressing error. If the offset is correct the segment table gives a base value to be added to the offset to generate the physical address. An illustration is given below (figure 17).

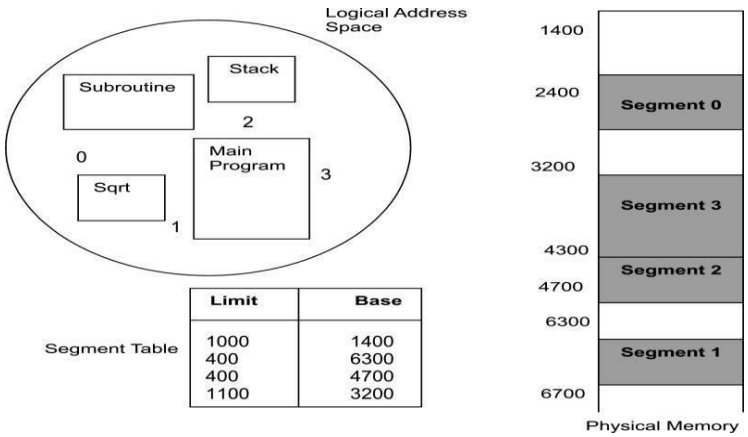


Figure 17: Illustration of Segmentation

A reference to logical location 53 in segment 2 ➔ physical location 4300 + 53 = 4353

852 in segment 3 ➔ physical location 3200 + 852 = 4062

1222 in segment 0 ➔ addressing error because 1222 > 1000

A Pascal compiler might create separate segments for the global variables, local variables of each procedure, procedure calls, stack to store parameters and link address, etc. The loader assigns segment numbers to these segments.

External Fragmentation:

Segments of user programs are allocated memory by the bob scheduler. This is similar to paging where segments could be treated as variable sized pages. So a first-fit or best-fit allocation scheme could be used since this is a dynamic storage allocation problem. But, as with variable sized partition scheme, segmentation too causes external fragmentation. When any free memory segment is too small to be allocated to a segment to be loaded, then external fragmentation is said to have occurred. Memory compaction is a solution to overcome external fragmentation.

The severity of the problem of external fragmentation depends upon the average size of a segment. Each process being a segment is nothing but the variable sized partition scheme. At the other extreme, every byte could be a segment, in which case external fragmentation is totally absent but relocation through the segment table is a very big overhead. A compromise could be fixed sized small segments, which is the concept of paging. Generally, if segments even though variable are small, external fragmentation is also less.

Sharing and Protection:

Program and data sharing is possible under paging. Sharing avoids providing a personal copy of a page to each program. This is required when re-entrant (pure procedure) codes are used by several programs. Reentrant codes are those codes which are not modified during their execution, i.e., there is no store operation in the code (no self address modification). Editor, compilers, loaders etc., are some of such codes.

No sharing example:

No. of users: 40. Text editor size: 30K. Data space: 5K.

Each user must have a copy of 30K of editor since they cannot share other user's copy of the editor. Total memory required to support these users: = 40 X 35 = 1400K .

Sharing Example:

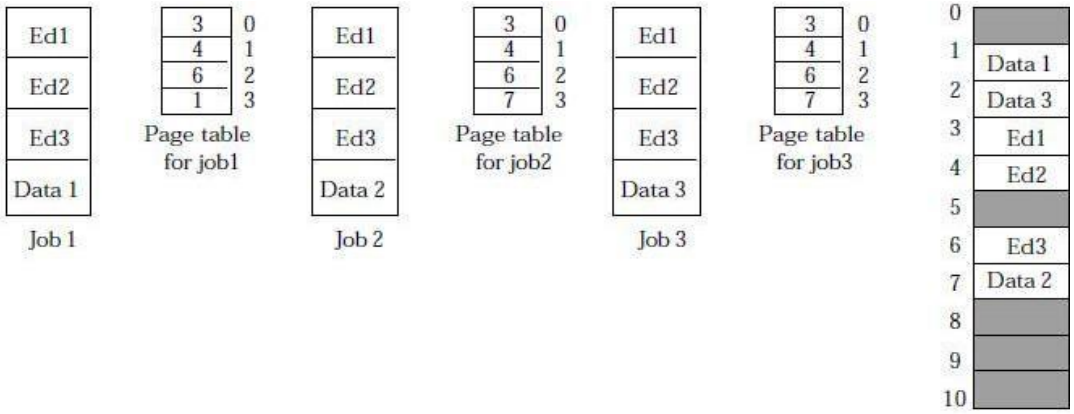


Figure 18: Page Sharing

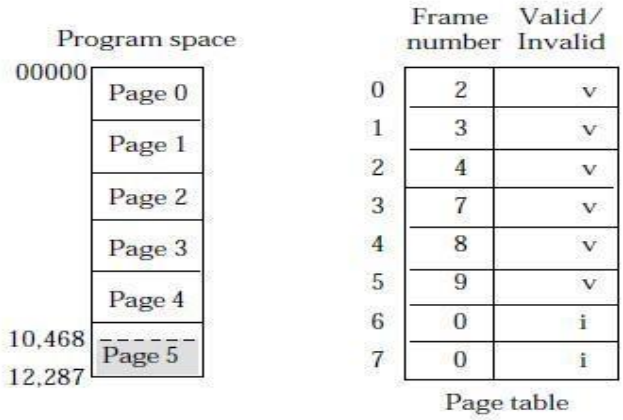
Protection: To protect illegal access by a program.

Consider the following situation:

Address Space: 14 bits. (address from 0 to 16,383). Page size: 2K (2048 words) Program size: 0 to 10,468 (pages from 0 to 5).

Total number of pages that can be addressed: 16,383/2048 = 8.

This means pages 0, 1, 2, 3, 4 and 5 are legal and pages 6 and 7 are illegal for this program. So if the program tries to access pages 6 or 7 the system must trap it. To recognize the illegal page request invalid bits are provided in the page table. The page table for the above system then looks like:



Virtual Memory

Introduction:

Memory management strategies like paging and segmentation helps to implement the concept of multi-programming. But they have a few disadvantages. One problem with the above strategies is that they require the entire process to be in main memory before execution can begin. Another disadvantage is the limitation on the size of the process. Processes whose memory requirement is larger than the maximum size of the memory available, will never be able to be run, that is, users are desirous of executing processes whose logical address space is larger than the available physical address space.

Virtual memory is a technique that allows execution of processes that may not be entirely in memory. In addition, virtual memory allows mapping of a large virtual address space onto a smaller physical memory. It also raises the degree of multi-programming and increases CPU utilization. Because of the above features, users are freed from worrying about memory requirements and availability.

Need for Virtual Memory Technique

Every process needs to be loaded into physical memory for execution. One brute force approach to this is to map the entire logical space of the process to physical memory, as in the case of paging and segmentation. Many a time, the entire process need not be in memory during execution. The following are some of the instances to substantiate the above statement:

- Code used to handle error and exceptional cases is executed only in case errors and exceptional conditions occur, which is usually a rare occurrence, may be one or no occurrences in an execution.
- Static declarations of arrays lists and tables declared with a large upper bound but used with no greater than 10% of the limit.
- Certain features and options provided in the program as a future enhancement, never used, as enhancements are never implemented.
- Even though entire program is needed, all its parts may not be needed at the same time because of overlays.

All the examples show that a program can be executed even though it is partially in

memory. This scheme also has the following benefits:

- Physical memory is no longer a constraint for programs and therefore users can write large programs and execute them.
- Physical memory required for a program is less. Hence degree of multiprogramming can be increased because of which utilization and throughput increase.
- I/O time needed for load / swap is less.

Virtual memory is the separation of logical memory from physical memory. This separation provides a large logical / virtual memory to be mapped on to a small physical memory (Figure 1).

Virtual memory is implemented using demand paging. Also demand segmentation could be used. A combined approach using a paged segmentation scheme is also available. Here user view is segmentation but the operating system implements this view with demand paging.

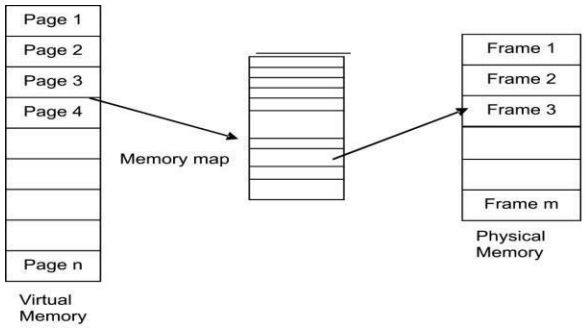


Figure 1: Virtual to physical memory mapping (n >> m)

Demand Paging

Demand paging is similar to paging with swapping (Figure 2). When a process is to be executed then only that page of the process, which needs to be currently executed, is swapped into memory. Thus, only necessary pages of the process are swapped into memory thereby decreasing swap time and physical memory requirement.

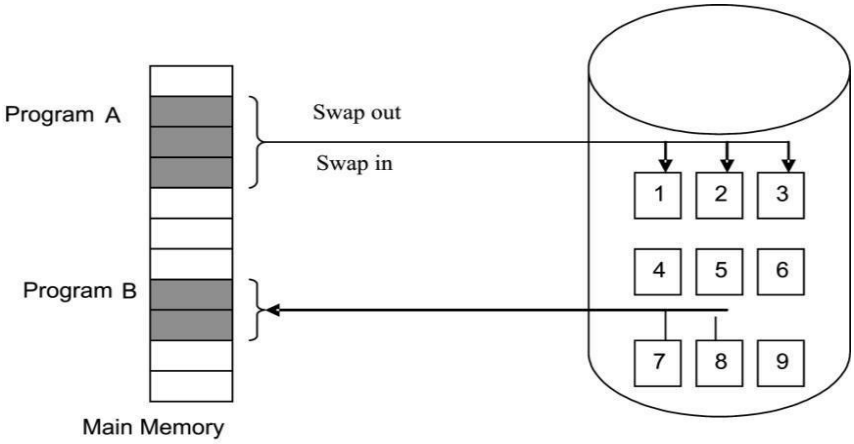


Figure 2: Paging with swapping

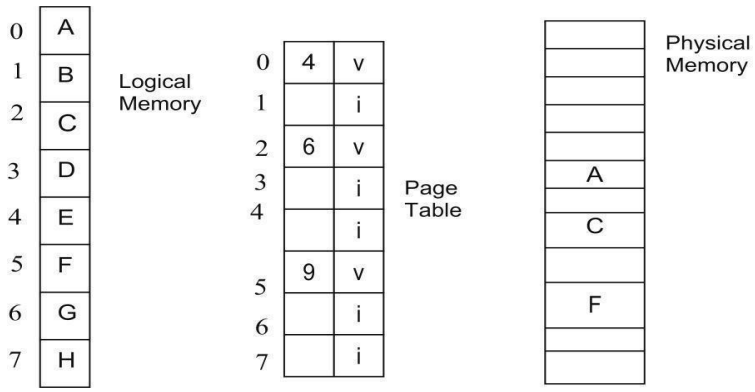


Figure 3: Demand paging with protection

The protection valid-invalid bit which is used in paging to determine valid / invalid pages corresponding to a process is used here also (Figure 3). If the valid-invalid bit is set, then the corresponding page is valid and also in physical memory. If the bit is not set, then any of the following can occur:

> Process is accessing a page not belonging to it, that is, an illegal memory access. Process is accessing a legal page but the page is currently not in memory.

If the same protection scheme as in paging is used, then in both the above cases a page fault error occurs. The error is valid in the first case but not in the second because in the latter a legal memory access failed due to non- availability of the page in memory which is an operating system fault. Page faults can thus be handled as follows (Figure 4):

1. Check the valid-invalid bit for validity.
2. If valid, then the referenced page is in memory and the corresponding physical address is generated.
3. If not valid then, an addressing fault occurs.
4. The operating system checks to see if the page is in the backing store. If present, then the addressing error was only due to non-availability of page in main memory and is a valid page reference.
5. Search for a free frame.
6. Bring in the page into the free frame.
7. Update the page table to reflect the change.
8. Restart the execution of the instruction stalled by an addressing fault.

In the initial case, a process starts executing with no pages in memory. The very first instruction generates a page fault and a page is brought into memory. After a while all pages required by the process are in memory with a

reference to each page generating a page fault and getting a page into memory. This is known as pure demand paging. The concept, never bring in a page into memory until it is required'. Hardware required to implement demand paging is the same as that for paging and swapping.

Page table with valid-invalid bit

Secondary memory to hold pages not currently in memory, usually a high speed disk known as a swap space or backing store.

A page fault at any point in the fetch-execute cycle of an instruction causes the cycle to be repeated.

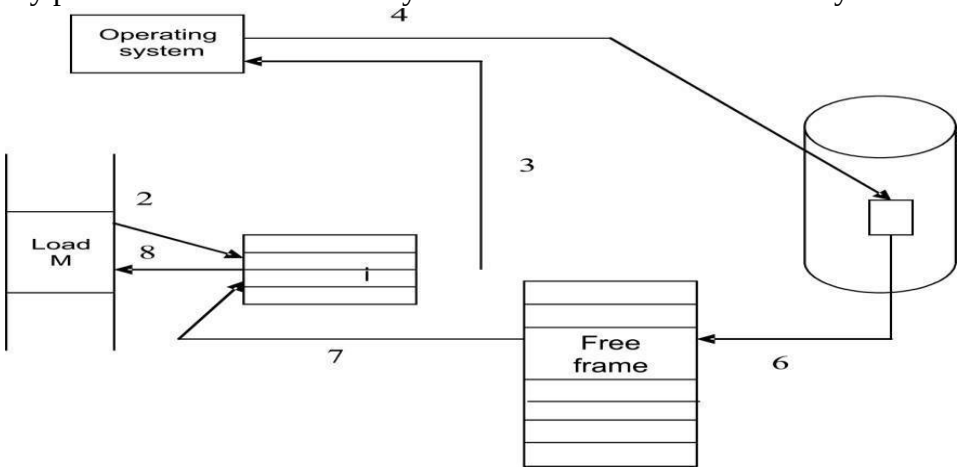


Figure 4: Handling a page fault

Page Replacement:

Initially, execution of a process starts with none of its pages in memory. Each of its pages page fault at least once when it is first referenced. But it may so happen that some of its pages are never used. In such a case those pages which are not referenced even once will never be brought into memory. This saves load time and memory space. If this is so, the degree of multi-programming can be increased so that more ready processes can be loaded and executed. Now, we may come across a situation wherein all of sudden, a process hitherto not accessing certain pages starts accessing those pages. The degree of multi-programming has been raised without looking into this aspect and the memory is over allocated. Over allocation of memory shows up when there is a page fault for want of page in memory and the operating syst n finds the required page in the backing store but cannot bring in the page into memory for want of free trames. More than one option exists at this stage:

Terminate the process. Not a good option because the very purposes of demand paging to increase CPU utilization and throughput by increasing the degree of multi-programming is lost.

Swap out a process to free all its frames. This reduces the degree of multi-programming that again may not be a good option but better than the first.

Page replacement seems to be the best option in many cases. The page fault service routine can be modified to include page replacement as follows:

1. Find for the required page in the backing store.
2. Find for a free frame
    a. if there exists one use it
    b. if not, find for a victim using a page replacement algorithm
    c. write the victim into the backing store.
    d. modify the page table to reflect a free frame
3. Bring in the required page into the free frame.
4. Update the page table to reflect the change.
5. Restart the process.

When a page fault occurs and no free frame is present, then a swap out and a swap in occur. A swap out is not always necessary. Only a victim that has been modified needs to be swapped out. If not, the frame can be over written by the incoming page. This will save time required to service a page fault and is implemented by the use of a dirty bit. Each frame in memory is associated with a dirty bit that is reset when the page is brought into memory. The bit is set whenever the frame is modified. Therefore, the first choice for a victim is naturally that frame with its dirty bit which is not set.

Page replacement is basic to demand paging. The size of the logical address space is no longer dependent on the physical memory. Demand paging uses two important algorithms:

Page replacement algorithm: When page replacement is necessitated due to non-availability of frames, the algorithm looks for a victim.

Frame allocation algorithm: In a multi-programming environment with degree of multi-programming equal to n, the algorithm gives the number of frames to be allocated to a process.

Page Replacement Algorithms:

A good page replacement algorithm generates as low a number of page faults as possible. To evaluate an algorithm, the algorithm is run on a string of memory references and a count of the number of page faults is recorded. The string is called a reference string and is generated using either a random number generator or a trace of memory references in a given system.

Illustration:

Address sequence: 0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103, 0104, 0101, 0610, 0102,
0103, 0104, 0101, 0609, 0102, and 0105

Page size: 100 bytes

Reference string: 1 4 1 6 1 6 1 6 1 6 1

The reference in the reference string is obtained by dividing (integer division) each address reference by the page size. Consecutive occurrences of the same reference are replaced by a single reference. To determine the number of page faults for a particular reference string and a page replacement algorithm, the number of frames available to the process needs to be known. As the number of frames available increases the number of page faults decreases. In the above illustration, if frames available were 3 then there would be only 3 page faults, one for each page reference. On the other hand, if there were only 1 frame available then there would be 11 page faults, one for every page reference.

FIFO (First Come First Served) Page Replacement Algorithm:

The first-in-first-out page replacement algorithm is the simplest page replacement algorithm. When a page replacement is required the oldest page in memory is the victim.

Illustration:

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

Memory frames: 3

Page faults:	7	7	7	2		2	2	4	4	4	0		0	0			7	7	7
		0	0	0		3	3	3	2	2	2		1	1			1	0	0
			1	1		1	0	0	0	3	3		3	2			2	2	1

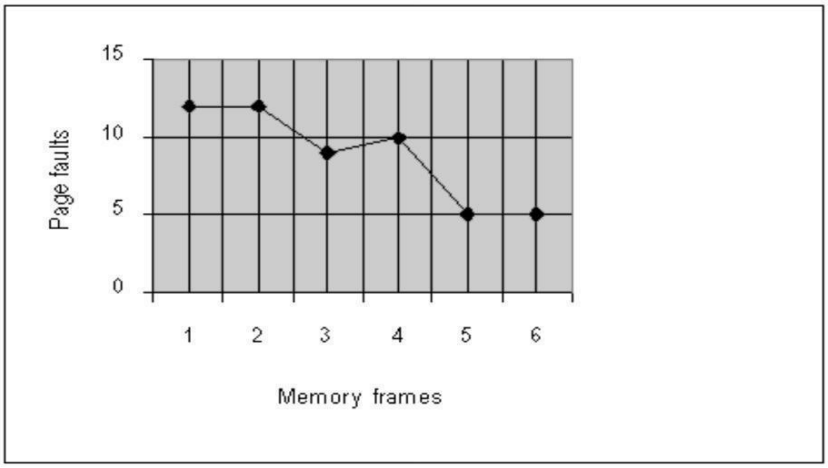
Number of page faults = 15.

The performance of the FIFO algorithm is not always good. The replaced page may have an initialization module that needs to be executed only once and therefore no longer needed. On the other hand, the page may have easily used variable in constant use. Such a page swapped out will cause a page fault almost immediately to be brought in. Thus, the number of page faults increases and results in slower process execution. Consider the following reference string:

Reference string: 1 2 3 4 1 2 5 1 2 3 4 5
Memory frames: 1, 2, 3, 4, 5

The chart below gives the number of page faults generated for each of the 1, 2, 3, 4 and 5 memory frames

available.



As the number of frames available increases, the number of page faults must decrease. But the chart above shows 9 page faults when memory frames available are 3 and 10 when memory frames available are 4. This unexpected result is known as ***Belady's anomaly***.

Implementation of FIFO algorithm is simple. A FIFO queue can hold pages in memory with a page at the head of the queue becoming the victim and the page swapped in joining the queue at the tail.

Optimal Algorithm:

An optimal page replacement algorithm produces the lowest page fault rate of all algorithms. The algorithm is to replace the page that will not be used for the longest period of time to come. Given a fixed number of memory frame by allocation, the algorithm always guarantees the lowest possible page fault rate and also does not suffer from Belady's anomaly.

Illustration:

Reference string:	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
Memory frames:	3																			
Page faults:	7	7	7	2		2		2			2			2				7		
		0	0	0		0		4			0			0				0		
			1	1		3		3			3			1				1		

Number of page faults = 9.

Ignoring the first three page faults that do occur in all algorithms, the optimal algorithm is twice as better than the FIFO algorithm for the given string. But implementation of the optimal page replacement algorithm is difficult since it requires future a priori knowledge of the reference string. Hence the optimal page replacement algorithm is more a benchmark algorithm for comparison.

LRU (Least Recently Used) Page Replacement Algorithm:

The main distinction between FIFO and optimal algorithm is that the FIFO algorithm uses the time when a page was brought into memory (looks back) whereas the optimal algorithm uses the time when a page is to be used in future (looks ahead). If the recent past is used as an approximation of the near future, then replace the page that has not been used for the longest period of time. This is the least recently used (LRU) algorithm.

Illustration:

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

Memory frames: 3

Page faults:	7	7	7	2		2		4	4	4	0		1		1		1
		0	0	0		0		0	0	3	3		3		0		0
			1	1		3		3	2	2	2		2		2		7

Number of page faults = 12.

The LRU page replacement algorithm with 12 page faults is better than the FIFO algorithm with 15 faults. The problem is to implement the LRU algorithm. An order for the frames by time of last use is required. Two options are feasible:

By use of counters By use of stack

In the first option using counters, each page table entry is associated with a variable to store the time when the page was used. When a reference to the page is made, the contents of the clock are copied to the variable in the page table for that page. Every page now has the time of last reference to it. According to the LRU page replacement algorithm the least recently used page is the page with the smallest value in the variable associated with the clock. Overheads here include a search for the LRU page and an update of the variable to hold clock contents each time a memory reference is made. In the second option a stack is used to keep track of the page numbers. A page referenced is always put on top of the stack. Therefore the top of the stack is the most recently used page and the bottom of the stack is the LRU page. Since stack contents in between need to be changed, the stack is best implemented using a doubly linked list. Update is a bit expensive because of the number of pointers to be changed, but there is no necessity to search for a LRU page. LRU page replacement algorithm does not suffer from Belady's anomaly. But both of the above implementations require hardware support since either the clock variable or the stack must be updated for every memory reference.

Thrashing:

When a process does not have enough frames or when a process is executing with a minimum set of frames allocated to it which are in active use, there is always a possibility that the process will page fault quickly. The page in active use becomes a victim and hence page faults will occur again and again. In this case a process spends more time in paging than executing. This high paging activity is called thrashing.

Causes for Thrashing:

The operating system closely monitors CPU utilization. When CPU utilization drops below a certain threshold, the operating system increases the degree of multiprogramming by bringing in a new process to increase CPU utilization. Let a global page replacement policy be followed. A process requiring more frames for execution page faults and steals frames from other processes which are using those frames. This causes the other processes also to page fault. Paging activity increases with longer queues at the paging device but CPU utilization drops. Since CPU utilization drops, the job scheduler increases the degree of multiprogramming by bringing in a new process. This only increases paging activity to further decrease CPU utilization. This cycle continues. Thrashing has set in and throughput drops drastically. This is illustrated in the figure (Figure 5):

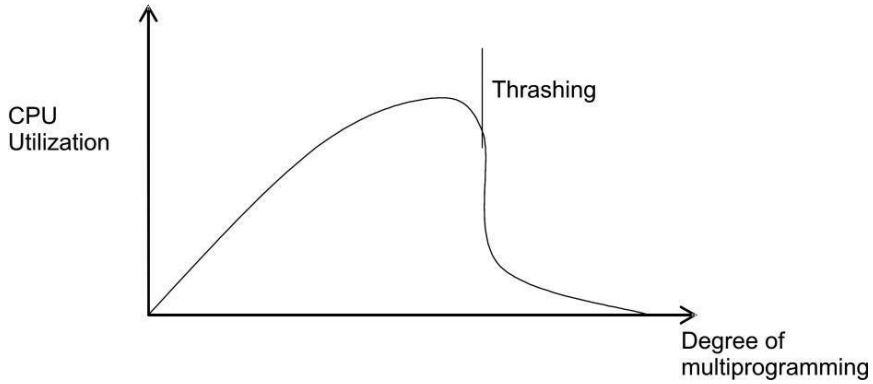


Figure 5: Thrashing

When a local page replacement policy is used instead of a global policy, thrashing is limited to a process

only.

To prevent thrashing, a process must be provided as many frames as it needs. A working-set strategy determines how many frames a process is actually using by defining what is known as a locality model of process execution. The locality model states that as a process executes, it moves from one locality to another, where a locality is a set of active pages used together. These localities are strictly not distinct and overlap.

For example, a subroutine call defines a locality by itself where memory references are made to instructions and variables in the subroutine. A return from the subroutine shifts the locality with instructions and variables of the

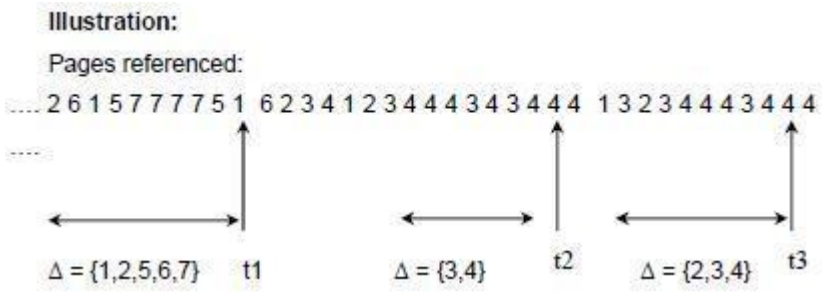
subroutine no longer in active use. So localities in a process are defined by the structure of the process and the data structures used therein. The locality model states that all programs exhibit this basic memory reference structure. Allocation of frames enough to hold pages in the current locality will cause faults for pages in this locality until all the required pages are in memory. The process will then not page fault until it changes locality. If allocated frames are less than those required in the current locality then thrashing occurs because the process is not able to keep in memory actively used pages

.

Working Set Model:

The working set model is based on the locality model. A working set window is defined. It is a parameter that maintains the most recent page references. This set of most recent page references is called the working set. An active page always finds itself in the working set. Similarly a page not used will drop off the working set time units after its last reference.

Thus the working set is an approximation of the program's locality.



If = 10 memory references then a working set {1,2,5,6,7} at t1 has changed to {3,4} at t2 and {2,3,4} at t3. The size of the parameter defines the working set. If too small it does not consist of the entire locality. If it is too big then it will consist of overlapping localities.

Let WSSi be the working set for a process Pi. Then D = Σ WSSi will be the total demand for frames from all processes in memory. If total demand is greater than total available, that is, D>m then thrashing has set in.

The operating system thus monitors the working set of each process and allocates to that process enough frames equal to its working set size. If free frames are still available then degree of multi-programming can be increased. If at any instant D>m then the operating system swaps out process to decrease the degree of multi-programming so that released frames could be allocated to other processes. The suspended process is brought in later and restarted.

The working set window is a moving window. Each memory reference appears at one end of the window while an older reference drops off at the other end. The working set model prevents thrashing while the degree of multiprogramming is kept as high as possible thereby increasing CPU utilization.

Page Fault Frequency:

One other way of controlling thrashing is by making use of the frequency of page faults. This page fault frequency (PPF) strategy is a more direct approach. When thrashing has set in page fault is high. This means to say that a process needs more frames. If page fault rate is low, the process may have more than necessary frames to execute. So upper and lower bounds on page faults can be defined. If the page fault rate exceeds the upper bound, then another frame is allocated to the process. If it falls below the lower bound then a frame already allocated can be removed. Thus monitoring the page fault rate helps prevent thrashing. As in the

working set strategy, some process may have to be suspended only to be restarted later if page fault rate is high and no free frames are available so that the released frames can be distributed among existing processes requiring more frames.

Unit 5

Cooperating Processes

An *Independent* process is not affected by other running processes.

Cooperating processes may affect each other, hopefully in some controlled way.

Why cooperating processes?

- information sharing
- computational speedup
- modularity or convenience

It's hard to find a computer system where processes do not coop erate. Consider the commands you type at the Unix command line. Your shell process and the process that executes your command must cooperate. If you use a pipe to hook up two commands, you have even more process cooperation (See the shell lab later this semester).

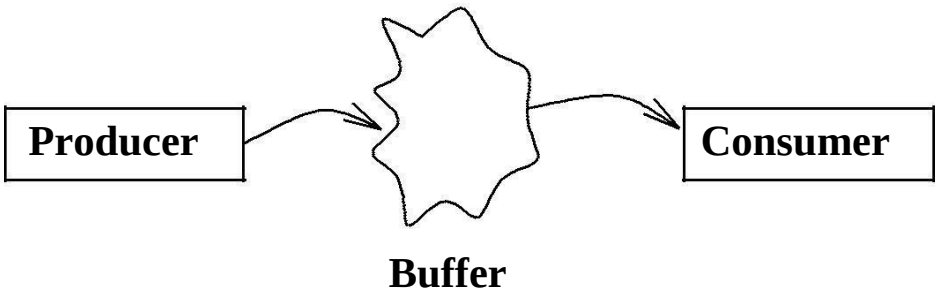
For the processes to cooperate, they must have a way to communicate with each other. Two common methods:

- shared variables – some segment of memory accessible to both processes
- message passing – a process sends an explicit message that is received by another

For now, we will consider shared-memory communication. We saw that threads, for example, share their global context, so that is one way to get two processes (threads) to share a variable.

Producer-Consumer Problem

The classic example for studying cooperating processes is the Producer-Consumer problem.



One or more produces processes is “producing” data. This dat a is stored in a buffer to be “consumed” by one or more consumer processes.

The buffer may be:

- *unbounded* – We assume that the producer can continue producing items an d storing them in the buffer at all times. However, the consumer must wait for an item to be inserted into the buffer before it can take one out for consumption.
- *bounded* – The producer must also check to make sure there is space avai lable in the buffer.

Bounded Buffer, buffer size n

For simplicity, we will assume the objects being produced and consumed are int values.

This solution leaves one buffer entry empty at all times:

- Shared data

```
int    buffer[n];    int
in=0;
int out=0;
```

- Producer process

```
while (1) {
    ...
    produce item;
    ...
    while (((in+1)%n) == out); /* busy wait */ buffer[in]=item;
    in=(in+1)%n;
}
```

- Consumer process

```
while (1) {
    while (in==out); /* busy wait */
    item=buffer[out]; out=(out+1)%n;
    ...
    consume item;
    ...
```

See Example:
/cluster/examples/prodcons-shmem

See Example:
/cluster/examples/prodcons-pthreads

Is there any danger with this solution in terms of concurrency? Remember that these processes can be interleaved in any order – the system could preempt the producer at any time and run the consumer.. Things to be careful about are shared references to variables.

Note that only one of the processes can *modify* the variables in and out. Both use the values, but only the producer modifies in and only the consumer modifies out. Try to come up with a situation that causes incorrect behavior – hopefully you ca nnot.

Perhaps we want to use the entire buffer...let's add a variable to keep track of how many items are in the buffer, so we can tell the difference between an empty and a full buffer:

- Shared data

```
int buffer[n]; int
in=0;
int out=0; int
counter=0;
```

- Producer process

```
while (1) {
    ...
    produce item;
    ...
    while (counter==n); /* busy wait */ buffer[in]=item;
    in=(in+1)%n;
    counter=counter+1;
}
```

- Consumer process

```
while (1) {
    while (counter==0); /* busy wait */ item=buffer[out];
    out=(out+1)%n;
    counter=counter-1;
    ...
    consume item;
    ...
}
```

We can now use the entire buffer. However, there is a potential danger here. We modify counter in both the producer and the consumer.

See Example:
/cluster/examples/prodcons-shmem

See Example:
/cluster/examples/prodcons-pthreads

Everything looks fine, but let's think about how a computer ac tually executes those statements to increment or decrement counter.

counter++ really requires three machine instructions: (i) load a register with the value of counter's memory location, (ii) increment the register, and (iii) store the register value back in counter's memory location. There's no reason that the operating system can't switch the process out in the middle of this.

Consider the two statements that modify counter:

Producer		Consumer	
P_1	R0 = counter;	C_1	R1 = counter;
P_2	R0 = R0 + 1;	C_2	R1 = R1 - 1;
P_3	counter = R0;	C_3	counter = R1;

Consider one possible ordering: P_1 P_2 C_1 P_3 C_2 C_3 , where counter=17 before starting. Uh oh.

What we have here is a *race condition*.

You may be thinking, “well, what are the chances, one in a mill ion that the scheduler will choose to preempt the process at exactly the wrong time?”

Doing something millions or billions of times isn't really that unusual for a computer, so it would come up..

Some of the most difficult bugs to find in software (often in ope rating systems) arise from race conditions.

This sort of interference comes up in painful ways when “real ” processes are interacting.

Consider two processes modifying a linked list, one inserting and one removing. A context switch at the wrong time can lead to a corrupted structure:

```
struct node {
    ...
    struct node *next;
}

struct node *head, *tail;

void insert(val) { struct node
    *newnode;
```

```
    newnode = getnode(); newnode-
    >next = NULL; if (head ==
    NULL){
          head = tail = newnode;
      } else { // <==== THE WRONG PLACE tail->next =
          newnode;
          tail = newnode;
      }
}

void remove() {
    // ... code to remove value ...
    head = head->next;
    if (head == NULL) tail = NULL; return
    (value);
}
```

If the process executing insert is interrupted at "the wrong place" and then another process calls remove until the list is empty, when the insert process resumes, it will be operating on invalid assumptions and the list will be corrupted.

In the bounded buffer, we need to make sure that when one process starts modifying counter, that it finishes before the other can try to modify it. This req uires *synchronization* of the processes.

Process synchronization is one of the major topics of this course, and one of the biggest reasons I think every undergraduate CS major should take an OS course.

If there were mutliple producers or consumers, we would have the same issue with the modification of in and out, so we can't rely on the "empty slot" approach in the more gene ral case.

We need to make those statements that increment and decrement counter *atomic*.

We say that the modification of counter is a *critical section*.

Critical Sections

The Critical-Section problem:

- n processes, all competing to use some shared data
- each process has a code segment (the critical section) in which shared data is accessed

```
while (1) { <CS
      Entry>
      critical section
```

<CS Exit> non-critical section
}

- Need to ensure that when one process is executing in its critical section, no other process is allowed to do so

Example: Intersection/traffic light analogy Example: one-

lane bridges during construction

Any solution to the critical section problem must satisfy three conditions:

1. *Mutual exclusion*: If process P_i is executing in its critical section, then no other processes can be executing in their critical sections. “One at a time.”
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. “no unnecessary waiting.”
3. *Bounded waiting*: A bound must exist on the number of times that other processes are al-lowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted. “no starvation. ” (We must assume that each pro-cess executes at non-zero speed, but make no assumptions about relative speeds of processes)

One possible way to deal with this is to make sure the problematic context switch doesn't happen.

If we disable interrupts so a context switch isn't possible while we're executing the critical section, we will prevent the interference.

However, this is a bit extreme, since it doesn't just restrict another process that will be modify-ing the same shared variable from being switched in, it prevents ANY other process from being switched in.

This approach would also not work in a multiprocessor environment when the interference could be from two processes running truly concurrently.

Algorithmic Approaches for 2 Processes

We first attempt to solve this for two processes, P_0 and P_1. They share some variables to syn-chronize. We fill in <CS Entry> and <CS Exit> from above with code that should satisfy the three conditions.

Critical Section Algorithm 1

- Share

```
int turn=0;
```

- Process P_i (define j = 1 − i, the other process)

```
while (1) {
    while (turn!=i); /* busy wait */

        /* critical section */

    turn=j;

        /* non-critical section */

}
```

Note the semicolon at the end of the while statement's condition at the line labeled “busy wait” above. This means that P_i just keeps comparing turn to i over and over until it succeeds. This is sometimes called a *spin lock*. For now, this is our only method of making one process wait for something to happen. More on this later.

This does satisfy mutual exclusion, but not progress (alternation is forced).

Critical Section Algorithm 2

We'll avoid this alternation problem by having a process wait only when the other has “indicated interest” in the critical section.

- Shared data

```
boolean flag[2];
flag[0]=false;
flag[1]=false;
```

- Process P_i

```
while (1) { flag[i]=true;
    while (flag[j]);

        /* critical section */

    flag[i]=false;

        /* non-critical section */

}
```

flag[i] set to true means that P_i is requesting access to the critical section.

This one also satisties mutual exclusion, but not progress.

Both can set their flags, then both start waiting for the other t o set flag[j] back to false. Not going to happen...

If we swap the order of the flag[i]=true; and while (flag[j]); statements, we no longer satisfy mutual exclusion.

Critical Section Algorithm 3

We combine the two previous approaches:

- Shared data

```
int turn=0; boolean
flag[2]; flag[0]=false;
flag[1]=false;
```

- Process P_i

```
while (1) {
    flag[i]=true; turn=j;
    while (flag[j] && turn==j);

        /* critical section */

    flag[i]=false;

        /* non-critical section */

}
```

So, we first indicate interest. Then we set turn=j;, effectively saying “no, you first” to the other process. Even if both processes are interested and both get to the while loop at the “same” time, only one can proceed. Whoever set turn first gets to go first.

This one satisfies all three of our conditions. This is known a s *Peterson’s Algorithm*.

Peterson's Algorithm in action:

See Example:
/cluster/examples/prodcons-pthreads

Algorithmic Approach for n Processes: Bakery algorithm

Can we generalize this for n processes? The Bakery Algorithm (think deli/bakery "now ser ving customer X " systems) does this.

The idea is that each process, when it wants to enter the critical section, takes a number. Whoever has the smallest number gets to go in. This is more complex than the bakery ticket-spitters because two processes may grab the same number (to guarantee that they wouldn't would require mutual exclusion – exactly the thing we're trying to implement), an d because there is no attendant to call out the next number – the processes all must come to agreement on who should proceed next into the critical section.

We break ties by allowing the process with the we call it i. This assumes PIDs from 0 to n − 1 lower process identifier (PID) to proceed. For P_i, for n processes, but this can be generalized.

Although two processes that try to pick a number at about the same time may get the same number, we do guarantee that once a process with number k is in, all processes choosing numbers will get a number > k.

Notation used below: an ordered pair (number, pid) fully identifies a process' number. We define a *lexicographic order* of these:

- (a, b) < (c, d) is a < c or if a = c and b < d

The algorithm:

- Shared data, initialized to 0's and false

```
boolean choosing[n]; int
number[n];
```

- Process P_i

```
while (1) { choosing[i]=true;
    number[i]=max(number[0],number[i],...,number[n-1])+1; choosing[i]=false;
    for (j=0; j<n; j++) { while
        (choosing[j]);
        while ((number[j]!=0) &&
                ((number[j],j) < (number[i],i)));
    }

        /* critical section */

    number[i]=0;
```

/* non-critical section */

}

Before choosing a number, a process indicates that it is doing so. Then it looks at everyone else's number and picks a number one larger. Then it says it's done choosing.

Then look at every other process. First, wait for that process not to be choosing. Then make sure we are allowed to go before that process. Once we have successfully decided that it's safe to go before every other process, then go!

To leave the CS, just reset the number back to 0.

So great, we have a solution. But...problems:

That's a lot of code. Lots of while loops and for loops. Could be expensive if we're going to do this a lot

1. If this is a highly popular critical section, the numbers might never reset, and we could overflow our integers. Unlikely, but think what could happen if we did.
2. It's kind of inconvenient and in some circumstances, unreasonable, to have these arrays of n values. There may not always be n processes, as some may come and go.

Synchronization hardware

Hardware support can make some of this a little easier. Problems can arise when a process is preempted within a single high-level language line. But we can't preempt in the middle of a machine instruction.

If we have a single machine instruction that checks the value of a variable and sets it **atomically**, we can use that to our advantage.

This is often called a **Test-and-Set** or Test and Set Lock instruction, and does this, atomi-cally:

```
boolean TestAndSet(boolean *target) { boolean orig_val =

    *target; *target = TRUE;
    return orig_val;
}
```

So it sets the variable passed in to true, and tells us if it was true or false *before* we called it. So if two processes do this operation, both will set the value of target to true, but only one will get a return value of false.

This is the functionality we want, but how does the instruction actually work?

Really, this would be an instruction that takes two operands:

TAS R, X

Where R is a CPU register and X is a memory location. After the instruction completes (atomically), the value that was in memory at X is copied into R, and the value of X is set to 1. R contains a 0 if X previously contained a 0. If two processes do this operation concurrently, only one will actually get the 0.

The Intel x86 BTS instruction sets a single bit in memory and sets the carry bit of the status word to the previous value. This is equivalent.

Think about how you might implement an atomic test and set on, for example, a microprogrammed architecture.

Any of these can be used to implement a “test and set” function as described above.

Armed with this atomic test-and-set, we can make a simple mutual exclution solution for any number of processes, with just a single shared variable:

- Shared data

  boolean lock = false;

- Process P_i

```
while (1) {
    while (TestAndSet(&lock)); /* busy wait */

        /* critical section */

    lock = false;

        /* non-critical section */
}
```

This satisfies mutual exclusion and progress, but not bounde d waiting (a process can leave the CS and come back around and grab the lock again before others who may be waiting ever get a chance to look).

A solution that does satisfy bounded waiting is still fairly complicated:

- Shared data

```
boolean lock=false;
boolean waiting[n]; /* all initialized to false */
```

- Process P_i and its local data

```
int j; boolean key;

while (1) { waiting[i]=true;
    key=true;
    while (waiting[i] && key) key =
        TestAndSet(&lock);
    waiting[i]=false;

        /* critical section */

    j=(i+1)%n;
    while ((j!=i) && !waiting[j]) j=(j+1)%n;
    if (j==i) lock=false; else
    waiting[j]=false;

        /* non-critical section */

}
```

Another hardware instruction that might be available is the atomic *swap* operation:

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

Different architectures have variations on this one, including the x86 XCHG instruction.

An algorithm to use this, minus the bounded wait again, is straightforward:

- Shared data

```
boolean lock = false;
```

- Process P_i

```
boolean key = true;
while (1) {
```

```
        while (key == true) swap(&key,&lock); /* busy wait */


                /
                 * critical        section */
              lock = false;

                /* non-critical section */

    }
```

It's pretty similar to what we saw before with TestAndSet().

Semaphores

All that busy waiting in all of our algorithms for mutual exclusion is pretty annoying. It's just wasted time on the CPU. If we have just one CPU, it doesn't make sense for that process to take up its whole quantum spinning away waiting for a shared variable to change that can't change until the current process relinquishes the CPU!

This inspired the development of the *semaphore*. The name comes from old-style railroad traffic control signals (see http://www.semaphores.com), where mechanical arms swing down to block a train from a section of track that another train is currently using. When the track was free, the arm would swing up, and the waiting train could now proceed.

A semaphore S is basically an integer variable, with two atomic operations:

```
wait(S):
   while (S <= 0); /* wait */ S--;

signal(S):
   S++;
```

wait and signal are also often called down and up (from the railroad semaphore analogy) and occasionally are called P and V (because Dijkstra, who invented them, was Dutch, and these are the first letters of the Dutch words *proberen* (to test) and *verhogen* (to increment)).

Important!!! Processes using a semaphore in its most pure form are not allowed to set or examine its value. They can use the semaphore *only* through the wait and signal operations.

Note, however, that we don't want to do a busy-wait. A process that has to wait should be put to sleep, and should wake up only when a corresponding signal occurs, as that is the only time the process has any chance to proceed.

Semaphores are built using hardware support, or using software techniques such as the ones we

discussed for critical section management.

Since the best approach is just to take the process out of the ready queue, some operating systems provide semaphores through system calls. We will examine their implementation in this context soon.

Given semaphores, we can create a much simpler solution to the critical section problem for n processes:

- Shared data

  semaphore mutex=1;

- Process P_i

```
while (1) {
    wait(mutex);

        /* critical section */

    signal(mutex);

      /* non-critical section */
}
```

The semaphore provides the mutual exclusion for sure, and should satify progress, but depending on the implementation of semaphores, may or may not provide bounded waiting.

A semaphore implementation might look like this:

```
struct semaphore { int
   value; proclist L;
};
```

- *block* operation suspends the calling process, removes it from consideration by the scheduler
- *wakeup(P)* resumes execution of suspended process P , puts it back into consideration

- wait(S):

```
S.value--;
if (S.value < 0) {
    add this process to S.L; block;
}
```

- signal(S):

```
S.value++;
if (S.value <= 0) {
    remove a process P from S.L; wakeup(P);
}
```

There is a fairly standard implementation of semaphores on many Unix systems: POSIX semaphores

- create a shared variable of type sem_t
- initialize it with sem_init(3)
- wait operation is sem_wait(3)
- signal operation is sem_post(3)
- deallocate with sem_destroy(3)

Examples using POSIX semaphores and a semaphore-like construct called a *mutex* provided by pthreads to provide mutual exclusion in the bounded buffer problem:

See Example:
/cluster/examples/prodcons-pthreads-counter-sem

The pthreads library *mutex* is essentially a binary semaphore (only 0 and 1 are allowed). See pthread_mutex_init(3).

Note that what we have been looking at are *counting semaphores*. This means that is the semaphore's value is 0 and there are two signal operations, its value will be 2. This means that the next two wait operations will not block.

This means that semaphores can be used in more general situations than simple mutual exclusion. Perhaps we have a section that we want at most 3 processes in at the same time. We can start with a semaphore initialized to 3.

Semaphores can also be used as a more general-purpose synchronization tool. Suppose statement B in process P_j can be executed only after statement A in P_i. We use a semaphore called flag, initialized to 0:

P_i	P_j
...	...
A;	wait(flag);
signal(flag);	B;
...	...

Here, P_j will be forced to wait only if it arrives at the wait call before P_i has executed the signal.

Of course, we can introduce *deadlocks* (two or more processes waiting indefinitely for an event that can only be caused by one of the waiting processes).

Consider semaphores Q and R, initialized to 1, and two processes that need to wait on both. A careless programmer could write:

```
        P P
          0   1 wait(Q); wait(R);
 wait(R); wait(Q);
        ...                 ...
signal(R); signal(Q); signal(Q);
signal(R);
        ...                 ...
```

Things might be fine, but they might not be.

There's also the possibility that a process might just forget a signal and leave one or more other processes (maybe even itself) waiting indefinitely.

We will look into the implementation of semaphores and mutexes later, but for now we will look more at how to use them as a synchronization tool.

Classical Problems of Synchronization

We will use semaphores to consider some synchronization problems. While some actual implementations provide the ability to “try to wait”, or to examin e the value of a semaphore's counter, we restict ourselves to initialization, wait, and signal.

Bounded buffer using semaphores

First, we revisit our friend the bounded buffer.

- Shared data:

  semaphore fullslots, emptyslots, mutex; full=0; empty=n;
  mutex=1;

- Producer process:

```
while (1) { produce item;
    wait(emptyslots);
```

```
    wait(mutex);
    add item to buffer;
    signal(mutex);
    signal(fullslots);
}
```

- Consumer process:

```
while (1) { wait(fullslots);
    wait(mutex);
    remove item from buffer;
    signal(mutex); signal(emptyslots);
    consume item;
}
```

mutex provides mutual exclusion for the modification of the buffer (not shown in detail). The others make sure that the consumer doesn't try to remove from an empty buffer (fullslots is > 0) or that the producer doesn't try to add to a full buffer (emptyslots is > 0).

Dining Philsophers

- One way to tell a computer scientist from other people is to ask about the dining philosophers.
- Five philosophers alternate thinking and eating
- Need 2 forks to do so (eating spaghetti), one from each side
- keep both forks until done eating, then replace both

Since fork is the name of a C function, we'll use a different (and possibly more appropriate) analogy of chopsticks. The philosophers needs two chopsticks to eat rice.

- Shared data:

```
semaphore          chopstick[5];
chopstick[0..4]=1;
```

- Philosopher i:

```
while (1) { wait(chopstick[i]);
    wait(chopstick[(i+1)%5]);

      /* eat */

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

      /* think */

}
```

This solution may deadlock.

One way to reduce the chances of deadlock might be to think firs t, since each might think for a different amount of time.

Another possibility:

Each philosopher

1. Picks up their left chopstick
2. Checks to see if the right chopstick is in use
3. If so, the philosopher puts down their left chopstick, and starts over at 1.
4. Otherwise, the philosopher eats.

Does this work?

No! It livelocks. Consider this: all could pick up their left chopstick, look right, put down the left, and repeat indefinitely.

How to solve this? Must either

1. introduce an asymmetry, or
2. limit the number of concurrently hungry philosophers to n − 1.

Here's one that includes an asymmetry, by having odd numbered philosophers pick up to the right first. The code for philosopher i and problem size n.

```
void philosopher() { think;
    if (odd(i)) {
```

```
        wait(chopstick[(i+1) % n]);
        wait(chopstick[i]);
    }
    else { wait(chopstick[i]);
        wait(chopstick[(i+1) % n]);
    }
    eat;
    if (odd(i)) { signal(chopstick[(i+1) % n]);
        signal(chopstick[i]);
    }
    else { signal(chopstick[i]);
        signal(chopstick[(i+1) % n]);
    }
}
```

Readers-Writers

We have a database and a number of processes that need access to it. We would like to maximize concurrency.

- There are multiple “reader processes” and multiple “writer processes”.
- Readers see what's there, but don't change anything. Like a person on a travel web site seeing what flights have seats available.
- Writers change the database. The act of making the actual reservation.
- It's bad to have a writer in with any other writers or readers – may sell the same seat to a number of people (airline, sporting event, etc). Remember counter++ and counter--!
- Multiple readers are safe, and in fact we want to allow as much concurrent access to readers as we can. Don't want to keep potential customers waiting.

A possible solution:

- Shared data:

  semaphore wrt, mutex; int readcount;
  wrt=1; mutex=1; readcount=0;

- Writer process:

```
while (1) {
    wait(wrt);

        /* perform writing */

    signal(wrt);
}
```

- Reader process:

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

        /* perform reading */

    wait(mutex);
    readcount--;
    if (readcount == 0) signal(wrt); signal(mutex);
}
```

Note that the semaphore mutex protects readcount and is shared among readers only.

Semaphore wrt is indicates whether it is safe for a writer, or the first reade r, to enter.

Danger: a reader may wait(wrt) while inside mutual exclusion of mutex. Is this OK?

This is a reader-preference solution. Writers can starve! This might not be good if the readers are "browsing customers" but the writers are "paying customers !"

Sleeping Barber

Problem: A certain barber shop has a single barber, a barber's chair, and n chairs in a waiting area. The barber spends his life in the shop. If there are no customers in the shop, the barber sleeps in the chair until a customer enters and wakes him. When customers are in the shop, the barber is giving one a haircut, and will call for the next customer when he finishes with each. Customers arrive periodically. If the shop is empty and the barber is asleep in the barber's chair, he wakes the barber and gets a haircut. If the barber is busy, but there are chairs available in the waiting room, he sleeps in one of those chairs until called. Finally, if there are no available chairs in the waiting room, the customer leaves and comes back another time.

A possible solution:

- Shared Data
  constant CHAIRS = maximum number of chairs (including barber chair) semaphore mutex=1,next_cust=0,barber_ready=0;
  int cust_count=0;

- Customer process

```
while (1) {
  /* live your non barber-shop life until you decide you need a haircut */
  wait(mutex);
    if
      (cust_count>=CHAIR
      S) { signal(mutex);
      exit; /* leave the shop if full, try tomorrow */
    }
    cust_count++; /* increment customer count */
    signal(mutex);
    signal(next_cust); /* wake the barber if he's sleeping */ wait(barber_ready); /* wait in
    the waiting room */

      /* get haircut here */

    wait(mutex);

    cust_count--; /* leave the shop, freeing a chair */ signal(mutex);
}
```

- Barber process

```
while (1) {
  wait(next_cust); /* sleep until a customer shows up */ signal(barber_ready); /* tell the next customer
  you are ready */

  /* give the haircut here */

}
```

Monitors

Semaphores are error-prone (oops, did I say wait? I meant signal!). You might never release a mutex, might run into unexpected orderings that lead to deadlock.

Monitors are a high-level language constuct intended to avoid some of these problems. A monitor is

an abstract data type with shared variables and methods, similar to a C++ or Java class. monitor example_mon { shared variables;

```
    procedure P1(...) {
        ...
    }

    procedure P2(...) {
        ...
    }

    ...

    initialization/constructor;
}
```

A monitor has a special property that at most one process can be actively executing in its methods at any time. Mutual exclusion everywhere inside the monitor!

But if only one process can be in, and a process needs to wait, no other processes can get in. So an additional feature of monitors is the *condition variable*. This is a shared variable that has semaphore-like operations wait() and signal(). When a process in a monitor has to wait on a condition variable, other processes are allowed in.

But when happens on a signal? If we just wake up the waiting process and let the signalling process continue, we have violated our monitor's rules and have two active processes in the monitor. Two possibilities:

- Force the signaler to leave immediately
- Force the signaler to wait until the awakened waiter leaves

There are waiting queues associated with each condition variable, and with the entire monitor for processes outside waiting for initial entry.

onstruct. They can be implemented using OS-provided functionality such as semaphores.

Also note the similarity between these monitors and Java classes and methods that use the synchronized keyword.

Introduction:

Note that monitors are a language c

Unit 6

Deadlock

Several processes compete for a finite set of resources in a multiprogrammed environment. A process requests for resources that may not be readily available at the time of the request. In such a case the process goes into a wait state. It may so happen that this process may never change state because the requested resources are held by other processes which themselves are waiting for additional resources and hence in a wait state. ***This situation is called a deadlock.***

Deadlock occurs when we have a set of processes [not necessarily all the processes in the system], each holding some resources, each requesting some resources, and none of them is able to obtain what it needs, i.e. to make progress. We will usually reason in terms of resources R1, R2, .., Rm and processes P1, P2, .., Pn. A process Pi that is waiting for some currently unavailable resource is said to be **blocked**. Resources can be **preemptable** or **non- preemptable**.

A resource is preemptable if it can be taken away from the process that is holding it [we can think that the original holder waits, frozen, until the resource is returned to it]. Memory is an example of a preemptable resource. Of course, one may choose to deal with intrinsically preemptable resources as if they were nonpreemptable. In our discussion we only consider non-preemptable resources. Resources can be **reusable** or **consumable**. They are reusable if they can be used again after a process is done using them. Memory, printers, tape drives are examples of reusable resources. Consumable resources are resources that can be used only once.

For example a message or a signal or an event. If two processes are waiting for a message and one receives it, then the other process remains waiting. To reason about deadlocks when dealing with consumable resources is extremely difficult. Thus we will restrict our discussion to reusable resources.

System Model:

The number of resources in a system is always finite. But the numbers of competing processes are many. Resources are of several types, each type having identical instances of the resource. Examples for resources could be memory space, CPU time, files, I/O devices and so on. If a system has 2 CPUs that are equivalent, then the resource type CPU time has 2 instances. If they are not equivalent, then each CPU is of a different resource type. Similarly the system may have 2 dot matrix printers and 1 line printer. Here the resource type of dot matrix printer has 2 instances whereas there is a single instance of type line printer. A process requests for resources, uses them if granted and then releases the resources for others to use. It goes without saying that the number of resources requested shall not exceed the total of each type available in the system. If a request for a resource cannot be granted immediately then the process requesting the resource goes into a wait state and joins the wait queue for the resource. A set of processes is in a state of deadlock if every process in the set is in some wait queue of a resource and is waiting for an event (release resource) to occur that can be caused by another process in the set.

For example, there are 2 resources, 1 printer and 1 tape drive. Process P1 is allocated tape drive and P2 is allocated printer. Now if P1 requests for printer and P2 for tape drive, a deadlock occurs.

Deadlock Characterization:

Necessary Conditions for Deadlock:

A deadlock occurs in a system if the following four conditions hold simultaneously:

1. **Mutual exclusion:** At least one of the resources is non-sharable, that is, only one process at a time can use the resource.

2. **Hold and wait:** A process exists that is holding on to at least one resource and waiting for an additional resource held by another process.

3. **No preemption:** Resources cannot be preempted, that is, a resource is released only by the process that is holding it.

4. **Circular wait:** There exist a set of processes P0, P1, ….., Pn of waiting processes such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2, ….., Pn-1 is waiting for a resource held Pn and Pn is in turn waiting for a resource held by P0.

Resource-Allocation Graph:

Deadlocks can be described by a resource allocation graph. The resource allocation graph is a directed graph consisting of vertices and directed edges. The vertex set is partitioned into two types, a subset representing processes and another subset representing resources. Pictorially, the resources are represented by rectangles with dots within, each dot representing an instance of the resource and circles represent processes.

A directed edge from a process to a resource (Pi → Rj) signifies a request from a process Pi for an instance of the resource Rj and Pi is waiting for Rj. A directed edge from a resource to a process (Rj → Pi) indicates that an instance of the resource Rj has been allotted to process Pi. Thus a resource allocation graph consists of vertices which include resources and processes and directed edges which consist of request edges and assignment edges.

A request edge is introduced into the graph when a process requests for a resource. This edge is converted into an assignment edge when the resource is granted. When the process releases the resource, the assignment edge is deleted. Consider the following system:

There are 3 processes P1, P2 and P3.

Resources R1, R2, R3 and R4 have instances 1, 2, 1, and 3

respectively. P1 is holding R2 and waiting for R1.

P2 is holding R1, R2 and is waiting for R3. P3 is holding R3.

The resource allocation graph for a system in the above situation is as shown below: (Figure 1)

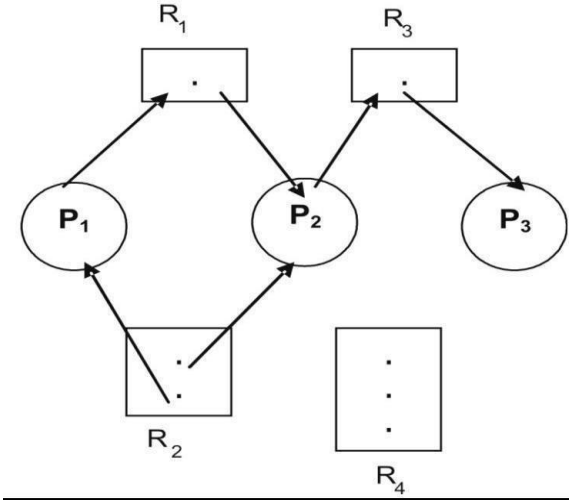


Figure 1: Resource allocation graph

If a resource allocation graph has no cycles (a closed loop in the direction of the edges), then the system is not in a state of deadlock. If on the other hand, there are cycles, then a deadlock may exist. If there are only single instances of each resource type, then a cycle in a resource allocation graph is a necessary and sufficient condition for existence of a deadlock (Figure 2). Here two cycles exist:

P1 → R1 → P2 →R3 → P3 →
R2→P1 P2 → R3 → P3 → R2 →
P2

Processes P0, P1 and P3 are deadlocked and are in a circular wait. P2 is waiting for R3 held by P3. P3 is waiting for P1 or P2 to release R2. So also P1 is waiting for P2 to release R1.

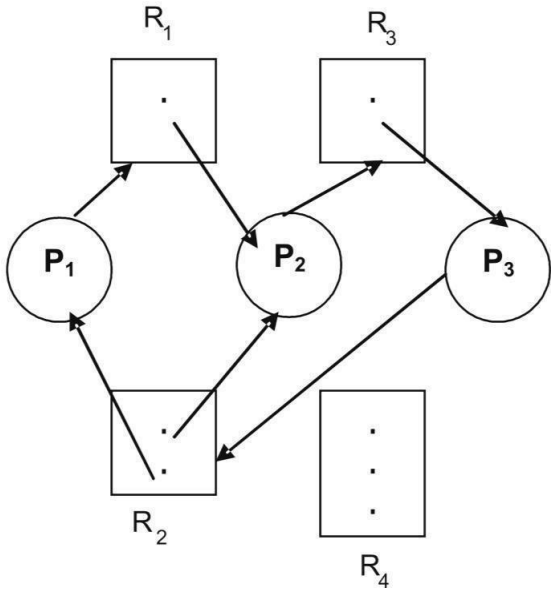


Figure 2: Resource allocation graph with deadlock

If there are multiple instances of resources types, then a cycle does not necessarily imply a deadlock. Here a cycle is a necessary condition but not a sufficient condition for the existence of a deadlock (Figure 3). Here also there is a cycle:

$$P1 \rightarrow R1 \rightarrow P3 \rightarrow R2 \rightarrow P1$$

The cycle above does not imply a deadlock because an instance of R1 released by P2 could be assigned to P1 or an instance of R2 released by P4 could be assigned to P3 there by breaking the cycle.

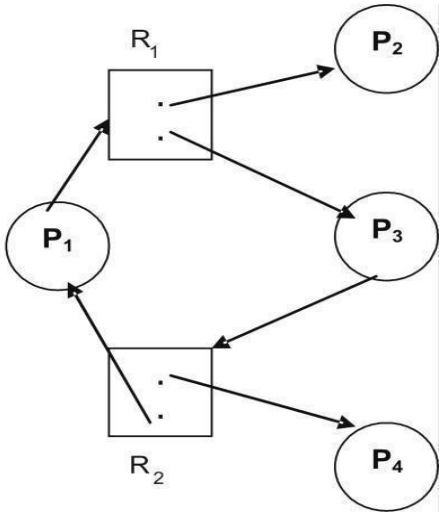


Figure 3: Resource allocation graph with a cycle but no deadlock

Deadlock Handling

Deadlock Prevention
Deadlock Avoidance
Deadlock Detection & Recovery

Different methods to deal with deadlocks include methods to ensure that the system will never enter into a state of deadlock, methods that allow the system to enter into a deadlock and then recover or to just ignore the problem of deadlocks. To ensure that deadlocks never occur, deadlock prevention / avoidance schemes are used. The four necessary conditions for deadlocks to occur are mutual exclusion, hold and wait, no preemption and circular wait.

Deadlock prevention ensures that at least one of the four necessary conditions for deadlocks does not hold. To do this the scheme enforces constraints on requests for resources. Dead-lock avoidance scheme

requires the operating system to know in advance, the resources needed by a process for its entire lifetime. Based on this a priori information, the process making a request is either made to wait or not to wait in case the requested resource is not readily available. If none of the above two schemes are used, then deadlocks may occur. In such a case, an

algorithm to recover from the state of deadlock is used. If the problem of deadlocks is ignored totally, that is, to say the system does not ensure that a deadlock does not occur and also does not provide for recovery from deadlock and such a situation arises, then there is no way out of the deadlock. Eventually, the system may crash because ore and more processes request for resources and enter into deadlock.

Deadlock Prevention

The four necessary conditions for deadlocks to occur are mutual exclusion, hold and wait, no preemption and circular wait. If any one of the above four conditions does not hold, then deadlocks will not occur. Thus prevention of deadlock is possible by ensuring that at least one of the four conditions cannot hold.

Mutual exclusion: Resources that can be shared are never involved in a deadlock because such resources can always be granted simultaneous access by processes. Hence processes requesting for such a sharable resource will never have to wait.

Examples of such resources include read-only files. Mutual exclusion must therefore hold for non-sharable resources. But it is not always possible to prevent deadlocks by denying mutual exclusion condition because some resources are by nature non-sharable, for example printers.

Hold and wait: To avoid hold and wait, the system must ensure that a process that requests for a resource does not hold on to another. There can be two approaches to this scheme:

1. A process requests for and gets allocated all the resources it uses before execution begins.
2. A process can request for a resource only when it does not hold on to any other.

Algorithms based on these approaches have poor resource utilization. This is because resources get locked with processes much earlier than they are actually used and hence not available for others to use as in the first approach. The second approach seems applicable only when there is assurance about reusability of data and code on the released resources. The algorithms also suffer from starvation since popular resources may never be freely available.

No preemption: This condition states that resources allocated to processes cannot be preempted. To ensure that this condition does not hold, resources could be preempted. When a process requests for a resource, it is allocated the resource if it is available. If it is not, than a check is made to see if the process holding the wanted resource is also waiting for additional resources. If so the wanted resource is preempted from the waiting process and allotted to the requesting process. If both the above is not true, that is, the resource is neither available nor held by a waiting process, then the requesting process waits. During its waiting period, some of its resources could also be preempted in which case the process will be restarted only when all the new and the preempted resources are allocated to it. Another alternative approach could be as follows: If a process requests for a resource which is not available immediately, then all other resources it currently holds are preempted. The process restarts only when the new and the preempted resources are allocated to it as in the previous case. Resources can be preempted only if their current status can be saved so that processes could be restarted later by restoring the previous states.

Example: CPU memory and main memory. But resources such as printers cannot be preempted, as their states cannot be saved for restoration later.

Circular wait: Resource types need to be ordered and processes requesting for resources will do so in an increasing order of enumeration. Each resource type is mapped to a unique integer that allows resources to be compared and to find out the precedence order for the resources. Thus F: R $\rightarrow$ N is a 1:1 function that maps resources to numbers.
For example:
F (tape drive) = 1, F (disk drive) = 5, F (printer) = 10.

To ensure that deadlocks do not occur, each process can request for resources only in an increasing order of these numbers. A process, to start with in the very first, instance can request for any resource say Ri. Thereafter it can request for a resource Rj if and only if F(Rj) is greater than F(Ri). Alternately, if F(Rj) is less than F(Ri), then Rj can be allocated to the process if and only if the process releases Ri. The mapping function F should be so defined that resources get numbers in the usual order of usage.

Deadlock Avoidance

Deadlock prevention algorithms ensure that at least one of the four necessary conditions for deadlocks namely mutual exclusion, hold and wait, no preemption and circular wait do not hold. The disadvantage with prevention algorithms is poor resource utilization and thus reduced system throughput. An alternate method is to avoid deadlocks. In this case additional a priori information about the usage of resources by processes is required. This

information helps to decide on whether a process should wait for a resource or not. Decision about a request is based on all the resources available, resources allocated to processes, future requests and releases by processes.

A deadlock avoidance algorithm requires each process to make known in advance the maximum number of resources of each type that it may need. Also known is the maximum number of resources of each type available. Using both the above a priori knowledge, deadlock avoidance algorithm ensures that a circular wait condition never occurs.

Safe State

A system is said to be in a safe state if it can allocate resources upto the maximum available and is not in a state of deadlock. A safe sequence of processes always ensures a safe state. A sequence of processes < P1, P2, ..., Pn > is safe for the current allocation of resources to processes if resource requests from each Pi can be satisfied from the currently available resources and the resources held by all Pj where j < i. If the state is safe then Pi requesting for resources can wait till Pj‟s have completed. If such a safe sequence does not exist, then the system is in an unsafe state. A safe state is not a deadlock state. Conversely a deadlock state is an unsafe state. But all unsafe states are not deadlock states as in Figure 4

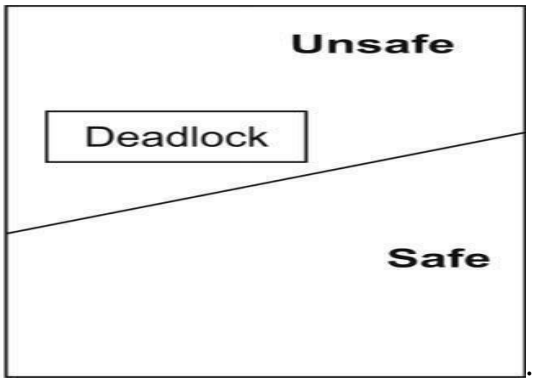


Figure 4: Safe, unsafe and deadlock state spaces

If a system is in a safe state it can stay away from an unsafe state and thus avoid deadlock. On the other hand, if a system is in an unsafe state, deadlocks cannot be avoided.

Illustration: A system has 12 instances of a resource type and 3 processes using these resources. The maximum requirements for the resource by the processes and their current allocation at an instance say t0 is as shown below:

Process	Maximum	Current
P0	10	5
P1	4	2
P3	9	2

At the instant t0, the system is in a safe state and one safe sequence is < P1, P0, P2 >. This is true because of the following facts: Out of 12 instances of the resource, 9 are currently allocated and 3 are free. P1 needs only 2 more, its maximum being 4, can be allotted 2. Now only 1 instance of the resource is free. When P1 terminates, 5 instances of the resource will be free. P0 needs only 5 more, its maximum being 10, can be allotted 5.

Now resource is not free. Once P0 terminates, 10 instances of the resource will be free. P3 needs only 7 more, its maximum being 9, can be allotted 7. Now 3 instances of the resource are free. When P3 terminates, all 12 instances of the resource will be free. Thus the sequence < P1, P0, P3 > is a safe sequence and the system is in a safe state. Let us now consider the following scenario at an instant t1. In addition to the allocation shown in the table above, P2 requests for 1 more instance of the resource and the allocation is made. At the instance t1, a safe sequence cannot be found as shown below:

Out of 12 instances of the resource, 10 are currently allocated and 2 are free.

P1 needs only 2 more, its maximum being 4, can be allotted 2. Now resource is not free. Once P1

terminates, 4 instances of the resource will be free. P0 needs 5 more while P2 needs 6 more. Since both P0 and P2 cannot be granted resources, they wait. **The result is a deadlock.**

Thus the system has gone from a safe state at time instant t0 into an unsafe state at an instant t1. The extra resource that was granted to P2 at the instant t1 was a mistake. P2 should have waited till other processes finished and

released their resources. Since resources available should not be allocated right away as the system may enter an unsafe state, resource utilization is low if deadlock avoidance algorithms are used.

Resource Allocation Graph Algorithm:

A resource allocation graph could be used to avoid deadlocks. If a resource allocation graph does not have a cycle, then the system is not in deadlock. But if there is a cycle then the system may be in a deadlock. If the resource allocation graph shows only resources that have only a single instance, then a cycle does imply a deadlock. An algorithm for avoiding deadlocks where resources have single instances in a resource allocation graph is as described below.

The resource allocation graph has request edges and assignment edges. Let there be another kind of edge called a claim edge. A directed edge Pi →Rj indicates that Pi may request for the resource Rj some time later. In a resource allocation graph a dashed line represents a claim edge. Later when a process makes an actual request for a resource, the corresponding claim edge is converted to a request edge Pi → Rj. Similarly when a process releases a resource after use, the assignment edge Rj → Pi is reconverted to a claim edge Pi Rj. Thus a process must be associated with its entire claim edges before it starts executing.

If a process Pi requests for a resource Rj, then the claim edge Pi → Rj is first converted to a request edge Pi → Rj. The request of Pi can be granted only if the request edge when converted to an assignment edge does not result in a cycle. If no cycle exists, the system is in a safe state and requests can be granted. If not the system is in an unsafe state and hence in a deadlock. In such a case, requests should not be granted. This is illustrated below (Figure 5a, 5b).

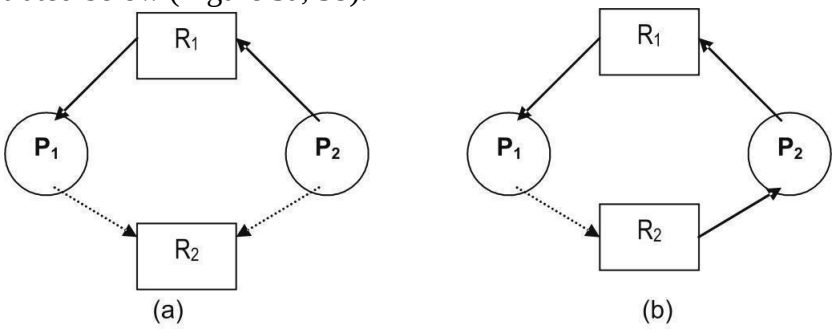


Figure 5: Resource allocation graph showing safe and deadlock states

Consider the resource allocation graph shown on the left above. Resource R2 is currently free. Allocation of R2 to P2 on request will result in a cycle as shown on the right. Therefore the system will be in an unsafe state. In this situation if P1 requests for R2, then a deadlock occurs

Banker's Algorithm:

The resource allocation graph algorithm is not applicable where resources have multiple instances. In such a case Banker‟s algorithm is used. A new process entering the system must make known a priori the maximum instances of each resource that it needs subject to the maximum available for each type. As execution proceeds and requests are made, the system checks to see if the allocation of the requested resources ensures a safe Consider the resource allocation graph shown on the left above. Resource R2 is currently free. Allocation of R2 to P2 on request will result in a cycle as shown on the right. Therefore the system will be in an unsafe state. In this situation if P1 requests for R2, then a deadlock occurs state. If so only are the allocations made, else processes must wait for resources.

The following are the data structures maintained to implement the Banker‟s algorithm:

1. **n:** Number of processes in the system.
2. **m:** Number of resource types in the system.
3. **Available:** is a vector of length m. Each entry in this vector gives maximum instances of a resource type that are available at the instant. Available[j] = k means to say there are k instances of the jth resource type Rj.
4. **Max:** is a demand vector of size n x m. It defines the maximum needs of each resource by the process. Max[i][j] = k says the ith process Pi can request for atmost k instances of the jth resource type Rj.

5. **Allocation:** is a n x m vector which at any instant defines the number of resources of each type currently allocated to each of the m processes. If Allocation[i][j] = k then ith process Pi is currently holding k instances of the jth resource type Rj.
6. **Need:** is also a n x m vector which gives the remaining needs of the processes. Need[i][j] = k means the i^{th} process Pi still needs k more instances of the jth resource type Rj.
Thus Need[i][j] = Max[i][j] – Allocation[i][j].

1. Safety Algorithm:

Using the above defined data structures, the Banker‟s algorithm to find out if a system is in a safe state or not is described below:

1. Define a vector Work of length m and a vector Finish of length n.
2. Initialize Work = Available and Finish[i] = false for i = 1, 2, ., n.
3. Find an i such that
   a. Finish[i] = false and
   b. Needi <= Work (Needi represents the ith row of the vector Need). If such an i does not exist , go to step 5.
4. Work = Work + Allocation i
   Go to step 3.
5. If finish[i] = true for all i, then the system is in a safe state.

2. Resource-Request Algorithm:

Let Requesti be the vector representing the requests from a process Pi. Requesti[j] = k shows that process Pi wants k instances of the resource type Rj. The following is the algorithm to find out if a request by a process can immediately be granted:

1. If Requesti <= Needi, go to step 2. Else Error “request of Pi exceeds Maxi”.
2. If Requesti <= Availablei, go to step 3. Else Pi must wait for resources to be released.
3. An assumed allocation is made as follows:

Available = Available – Requesti
Allocationi = Allocationi +
Requesti Needi = Needi –
Requesti

If the resulting state is safe, then process Pi is allocated the resources and the above changes are made permanent. If the new state is unsafe, then Pi must wait and the old status of the data structures is restored.

Illustration: n = 5 < P0, P1, P2, P3, P4 >
M = 3 < A, B, C >
Initially Available = < 10, 5, 7 >

At an instant t0, the data structures have the following values:

	Allocation	Max	Available	Need
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

To find a safe sequence and to prove that the system is in a safe state, use the safety algorithm as follows:

Step	Work	Finish	Safe sequence
0	3 3 2	F F F F F	< >
1	5 3 2	F T F F F	< P_1 >
2	7 4 3	F T F T F	< P_1, P_3 >
3	7 4 5	F T F T T	< P_1, P_3, P_4 >
4	7 5 5	T T F T T	< P_1, P_3, P_4, P_0 >
5	10 5 7	T T T T T	< P_1, P_3, P_4, P_0, P_2 >

Now at an instant t_1, $Request_1$ = < 1, 0, 2 >. To actually allocate the requested resources, use the request-resource algorithm as follows:

Request1 < Need1 and Request1 < Available so the request can be considered. If the request is fulfilled, then the new the values in the data structures are as follows:

	Allocation	Max	Available	Need
	A B C	A B C	A B C	A B C
P0	0 1 0	7 5 3	2 3 0	7 4 3
P_1	3 0 2	3 2 2		0 2 0
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

Use the safety algorithm to see if the resulting state is safe:

Step	Work	Finish	Safe sequence
0	2 3 0	F F F F F	< >
1	5 3 2	F T F F F	< P_1 >
2	7 4 3	F T F T F	< P_1, P_3 >
3	7 4 5	F T F T T	< P_1, P_3, P_4 >
4	7 5 5	T T F T T	< P_1, P_3, P_4, P_0 >
5	10 5 7	T T T T T	< P_1, P_3, P_4, P_0, P_2 >

Since the resulting state is safe, request by P1 can be granted. Now at an instant t2 Request4 = < 3, 3, 0 >. But since Request4 > Available, the request cannot be granted. Also Request0 = < 0, 2, 0> at t2 cannot be granted since the resulting state is unsafe as shown below:

	Allocation	Max	Available	Need
	A B C	A B C	A B C	A B C
P0	0 3 0	7 5 3	2 1 0	7 2 3
P_1	3 0 2	3 2 2		0 2 0
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

Using the safety algorithm, the resulting state is unsafe since Finish is false for all values of i and we cannot find a safe sequence.

Step	Work	Finish	Safe sequence
0	2 1 0	F F F F F	< >

Deadlock Detection

If the system does not ensure that a deadlock cannot be prevented or a deadlock cannot be avoided, then a deadlock may occur. In case a deadlock occurs, the system must-

1. Detect the deadlock
2. Recover from the deadlock

Single Instance of a Resource:

If the system has resources, all of which have only single instances, then a deadlock detection algorithm, which uses a variant of the resource allocation graph, can be used. The graph used in this case is called a wait-for graph. The wait-for graph is a directed graph having vertices and edges. The vertices represent processes and directed edges are present between two processes, one of which is waiting for a resource held by the other.

Two edges Pi → Rq and Rq → Pj in the resource allocation graph are replaced by one edge Pi → Pj in the wait-for graph. Thus, the wait-for graph is obtained by removing vertices representing resources and then collapsing the corresponding edges in a resource allocation graph. An Illustration is shown in Figure 6.

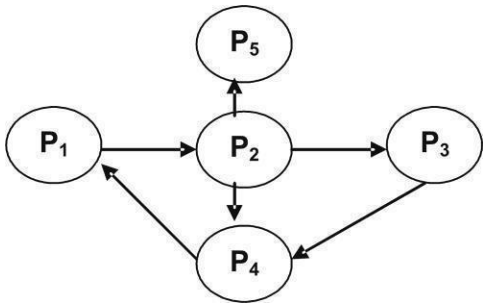


Figure 6: Wait-for graph

As in the previous case, a cycle in a wait-for graph indicates a deadlock. Therefore, the system maintains a wait-for graph and periodically invokes an algorithm to check for a cycle in the wait-for graph.

Multiple Instance of a Resource:

A wait-for graph is not applicable for detecting deadlocks where multiple instances of resources exist. This is because there is a situation where a cycle may or may not indicate a deadlock. If this is so then a decision cannot be made. In situations where there are multiple instances of resources, an algorithm similar to Banker‟s algorithm for deadlock avoidance is used.

Data structures used are similar to those used in Banker‟s algorithm and are given below:

1. **n:** Number of processes in the system.
2. **m:** Number of resource types in the system.
3. **Available:** is a vector of length m. Each entry in this vector gives maximum instances of a resource type that are available at the instant.
4. **Allocation:** is a n x m vector which at any instant defines the number of resources of each type currently allocated to each of the m processes.
5. **Request:** is also a n x m vector defining the current requests of each process. Request[i][j] = k means the i^{th} process Pi is requesting for k instances of the j^{th} resource type Rj.

ALGORITHM

1. Define a vector Work of length m and a vector Finish of length n.
2. Initialize Work = Available and
   For I = 1, 2, ….., n
   If $Allocation_i$!= 0
   Finish[i] = false
   Else
   Finish[i] = true
3. Find an i such that
   a. Finish[i] = false and
   b. $Request_i$ <= Work
   If such an i does not exist , go to step 5.
4. Work = Work + $Allocation_i$
   Finish[i] = true
   Go to step 3.
5. If finish[i] = true for all i, then the system is not in deadlock.

Else the system is in deadlock with all processes corresponding to Finish[i] = false being deadlocked.

Illustration: n = 5 < P_0, P_1, P_2, P_3, P_4 >

M = 3 < A, B, C >

Initially Available = < 7, 2, 6 >

At an instant t0, the data structures have the following values:

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

To prove that the system is not deadlocked, use the above algorithm as follows:

Step	Work	Finish	Safe sequence
0	0 0 0	F F F F F	< >
1	0 1 0	T F F F F	< P_0 >
2	3 1 3	T F T F F	< P_0, P_2 >
3	5 2 4	T F T T F	< P_0, P_2, P_3 >
4	5 2 6	T F T T T	< P_0, P_2, P_3, P_4 >
5	7 2 6	T T T T T	< P_0, P_2, P_3, P_4, P_1 >

Now at an instant t_1, $Request_2$ = < 0, 0, 1 > and the new values in the data structures are as follows:

	Allocation	Request	Available
	A B C	A B C	A B C
P0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 1	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

To prove that the system is deadlocked, use the above algorithm as follows:

Step	Work	Finish	Safe sequence
0	0 0 0	F F F F F	< >
1	0 1 0	T F F F F	< P_0 >

The system is in deadlock with processes P_1, P_2, P_3, and P_4 deadlocked.

Recovery from Deadlock

Once a deadlock has been detected, it must be broken. Breaking a deadlock may be manual by the operator when informed of the deadlock or automatically by the system. There exist two options for breaking deadlocks:

1. Abort one or more processes to break the circular-wait condition causing deadlock
2. Preempting resources from one or more processes which are deadlocked.

In the first option, one or more processes involved in deadlock could be terminated to break the deadlock. Then either aborts all processes or abort one process at a time till deadlock is broken. The first case guarantees that the deadlock will be broken. But processes that have executed for a long time will have to restart all over again. The second case is better but has considerable overhead as detection algorithm has to be invoked after terminating every process. Also choosing a process that becomes a victim for termination is based on many factors like

Priority of the processes

Length of time each process has executed and how much more it needs for completion Type of resources and the instances of each that the processes use

Need for additional resources to complete

Nature of the processes, whether iterative or batch

Based on these and many more factors, a process that incurs minimum cost on termination becomes a victim. In the second option some resources are preempted from some processes and given to other processes until the deadlock cycle is broken. Selecting the victim whose resources can be preempted is again based on the minimum cost criteria. Parameters such as number of resources a process is holding and the amount of these resources used thus far by the process are used to select a victim. When resources are preempted, the process holding the resource cannot continue. A simple solution is to abort the process also. Better still is to rollback the process to a safe state to restart later. To determine this safe state, more information about running processes is required which is again an overhead. Also starvation may occur when a victim is selected for preemption, the reason being resources from the same process may again and again be preempted. As a result the process starves for want of resources. Ensuring that a process can be a victim only a finite number of times by having this information as one of the parameters for victim selection could prevent starvation. Prevention, avoidance and detection are the three basic approaches to handle deadlocks. But they do not encompass all the problems encountered. Thus a combined approach of all the three basic approaches is used.

Questions:

1. Explain what is deadlock? What are the necessary conditions for deadlocks? Explain how it is possible to prevent a deadlock in a system.
2. Write a note on Resource Allocation Graph.
3. Describe safe, unsafe and deadlock state of a system.
4. What is the use of Resource allocation graph Algorithm?
5. Explain how Banker‟s algorithm is used to check safe state of a system.
6. Explain the different methods used to detect a deadlock.
7. Compare the relative merits and demerits of using prevention, avoidance and detection as strategies for dealing with deadlocks.
8. Numericals‟ based on Bankers and Safety Algorithms.

Protection and Security

Introduction

Personal computers were designed and intended for individual use. Hence security and protection features were minimal. No two users could simultaneously use the same machine. Locking the room physically which housed the computer and its accessories could easily protect data and stored information. But today hardware costs have reduced and people have access to a wide variety of computing equipment. With a trend towards networking, users have access to data and code present locally as well as at remote locations. The main advantages of networking like data sharing and remote data access have increased the requirements of security and protection. Security and protection are the two main features that motivated development of a network operating system (example Novell NetWare).

Major th eats to security can be categorized as

Tapping
Disclosure
Amendme
nt
Fabrication
Denial

Unauthorized use of service (tapping) and unauthorized disclosure of information (disclosure) are passive threats whereas unauthorized alteration or deletion of information (amendment), unauthorized generation of information (fabrication) and denial of service to authorized users (denial) are active threats. In either tapping or disclosure, information goes to a third party. In the former, information is accessed by the third party without the knowledge of the other two parties and in the latter the source willingly / knowingly discloses it to the third party.

Security is an important aspect of any operating system. Open Systems Interconnection (OSI) defines the elements of security in the following terms:

Confidentiality: Information is not accessed in an unauthorized manner (controlled read)
Integrity: Information is not modified or deleted in an unauthorized manner (controlled write)
Availability: Information is available to authorized users when needed (controlled read / write / fault recovery)

Security is concerned with the ability of the operating system to enforce control over storage and movement of data in and between the objects that the operating system supports.

1. Potential Information Security Violations
   - Unauthorized information release
   - Unauthorized information modification
   - Unauthorized denial of service
2. Aspects of Security
   - external (physical) = access to computer system
   - internal = allowed actions within the computer system
     - policies - what should be done
     - mechanisms - how it should be done
       including *protection mechanisms*
   - authentication = verification of user's identity
   - implementations:
     - Writing a new OS from scratch costs too much
     - Getting people to use a new OS is difficult
     - New OS technology is adopted if it can be added onto an existing OS
     - Otherwise, it is not adopted
     - How to implement security?
     - Security kernel
   - Access lists vs. capability lists

- access list ~ reservation (e.g. party in Chuckecheese)
- capability list ~ ticket (e.g. theater ticket)

3. Protection *versus* Security

- protection -- mechanisms
  e.g., putting locks on doors

- security -- policies, which specify how protection mechanisms are to be used *e.g.*, who gets keys? when are doors locked?
- protection domain of a process
  - resources that the process can access
  - operations it can perform on those resources
  - good policy is to restrict domain to what is needed

4. Design Principles

- economy - overhead should be tolerable
- complete mediation - all requests are checked
- open design - does not depend on attacker's ignorance
- separation of privileges -- requiring two keys is more robust
- least privilege -- process given bare minimum right to finish
- least common mechanism -- mechanisms common to two users should be minimized acceptability -- simple to use fail-safe defaults -- default should mean lack of access

Objectives:

Attacks on Security, Meaning of Authentication and Confidentiality. Computer Viruses and types of viruses.

Computer worms.
Security Design principles.

Protection Mechanisms and Security in Distributed Environment. **a> Attacks on Security**

A security system can be attacked in many ways. Some of them are discussed below:

1 Authentication

Authentication is verification of access to system resources. Penetration is by an intruder who may : Guess / steal somebody‟s password and use it

Use vendor supplied password usually used by system administrator for purposes of system maintenance

Find a password by trial and error

Use a terminal to access information that has been logged on by another user and just left like that.

Use a dummy login program to fool a user

2 Browsing

Browsing through system files could get intruder information necessary to access files with access controls which are very permissive thus giving the intruder access to unprotected files / databases. **3 Invalid Parameters**

Passing of invalid parameters of failure to validate them properly can lead to serious security violations.

4 Line Tapping

A communication line is tapped and confidential data is accessed or even modified. Threat could be in the form of tapping, amendment or fabrication.

5 Improper Access Controls

If the system administrator has not planned access controls properly, then some users may have too many privileges and others very few. This amounts to unauthorized disclosure of information or denial of service. **6 Rogue Software**

A variety of software programs exist under this title. Computer virus is very well known among others. This is a deliberately written program or part of it intended to create mischief. Such programs vary in terms of complexity or damage they cause. Creators of this software have a deep knowledge of the operating system and the underlying hardware. Other rogue software includes Trojan horse, Chameleon, Software bomb, Worm, etc.

The above mentioned were some common ways in which a security system could be attacked. Other ways in which a security system can be attacked may be through Trap doors, Electronic data capture, Lost line, Waste recovery and Covert channels.

b> Computer Worms

A computer worm is a full program by itself. It spreads to other computers over a network and while doing so consumes network resources to a very large extent. It can potentially bring the entire network to a halt. The invention of computer worms was for a good purpose. Research scientists at XEROX PARC research center wanted to carry out large computations. They designed small programs (worms) containing some identified piece of computations that could be carried out independently and which could spread to other computers. The worm would then execute on a machine if idle resources were available or else it would hunt the network for machines with idle resources. A computer worm does not harm any other program or data but spreads, thereby consuming large resources like disk storage, transmission capacity, etc. thus denying them to legal users. A worm usually operates on a network. A node in a network maintains a list of all other nodes on the network and also a list of machine addresses on the network. A worm program accesses this list and using it copies itself to all those address and spreads. This large continuous transfer across the network eats up network resources like line capacity, disk space, network buffers, tables, etc.

Two major safeguards against worms are:

Prevent its creation: through strong security and protection policies

Prevent its spreading: by introducing checkpoints in the communication system and disallowing transfer of executable files over a network unless until they are permitted by some authorized person.

c> Computer Virus

A computer virus is written with an intention of infecting other programs. It is a part of a program that piggybacks on to a valid program. It differs from the worm in the following way

Worm is a complete program by itself and can execute independently whereas virus does not operate independently.

Worm consumes only system resources but virus causes direct harm to the system by corrupting code as well as data.

1. Types of Viruses

There are several types of computer viruses. New types get added every now and then. Some of the common varieties are:

- Boot sector infectors
- Memory resident infectors
- File specific infectors
- Command processor
- infectors General purpose infectors

2. Infection Methods

Viruses infect other programs in the following ways:

- Append: virus code appends itself to a valid unaffected program
- Replace: virus code replaces the original executable program either completely
- or partially Insert: virus code gets inserted into the body of the executable code to carry out some undesirable actions

Delete: Virus code deletes some part of the executable program

Redirect: The normal flow of a program is changed to execute a virus code that could exist as an appended portion of an otherwise normal program.

3. Mode of Operation

A virus works in a number of ways. The developer of a virus (a very intelligent person) writes an interesting program such as a game or a utility knowing well the operating system details on which it is supposed to execute. This program has some embedded virus code in it. The program is then distributed to users for use through enticing advertisements and at a low price. Having bought the program at a throwaway price, the user copies it into his / her machine not aware of the devil which will show up soon. The virus is now said to be in a nascent state. Curious about the output of the program bought, the user executes it. Because the virus is embedded in the host program being run, it also executes and spreads thus causing havoc.

4. Virus Detection

Virus detection programs check for the integrity of binary files by maintaining a checksum and recalculating it at regular intervals. A mismatch indicates a change in the executable file, which may be caused due to tampering. Some programs are also available that are resident in memory and continuously monitor memory and I/O operations.

5. Virus Removal

A generalized virus removal program is very difficult. Anti-virus codes for removal of viruses are available. Bit patterns in some virus code are predictable. The anti-virus programs scan the disk files for such patterns of the known virus and remove them. But with a number of viruses cropping up every now and then, development and availability of anti-virus for a particular type is delayed and harm done.

6. Virus Prevention

„Prevention is better than cure". As the saying goes, there is no good cure available after infection. One of the safest ways to prevent virus attacks is to use legal copies of software. Also system needs to be protected against use of unauthorized / unchecked floppy disks. Frequent backups and running of monitoring programs help detection and subsequent prevention.

d> Security Design Principles

General design principles for protection put forward by Saltzer and Schroeder can be outlined as und :

Public design: a security system should not be a secret, an assumption that the penetrator will know about it is a better assumption.

Least privileges: every process must be given the least possible privileges necessary for execution. This assures that domains to be protected are normally small. But an associated overhead is frequent switching between domains when privileges are updated.

Explicit demand: access rights to processes should not be granted as default. Access rights should be explicitly demanded. But this may result in denial of access on some ground to a legal user.

Continuous verification: access rights should be verified frequently. Checking only at the beginning may not be sufficient because the intruder may change access rights after initial check.

Simple design: a simple uniform security system built in layers, as an integral part of the system is preferred.

- User acceptance: Users should not have to spend a lot of effort to learn how to protect their
- files.

Multiple conditions: wherever possible, the system must be designed to depend on more than one condition, for example, two passwords / two keys.

e> Authentication

Authentication is a process of verifying whether a person is a legal user or not. This can be by either verification of users logging into a centralized system or authentication of computers that are to work in a network or a distributed environment. Password is the most commonly used scheme. It is easy to implement. User name is associated with a password. This is stored in encrypted form by the system. When the user logs onto the system, the user has to enter his user name and password against a prompt. The entered password is then encrypted and matched with the one that is stored in the file system. A tally will allow the user to login. No external hardware is needed. But limited protection is provided. The password is generally not echoed on the screen while being keyed in. Also it is stored in encrypted form. It cannot be deciphered easily because knowing the algorithm for deciphering will not suffice as the key is ought to be known for deciphering it. Choosing a password can be done by the system or by the system administrator or by the users themselves. A system-selected password is not a good choice as it is difficult to remember. If the system administrator gives a user a password then more than one person knows about it. User chosen passwords is practical and popular. Users should choose passwords that are

not easy to guess. Choosing user names, family names, names of cities, etc are easy to guess. Length of a password plays an important role in the effectiveness of the password. If it is short it is easy to remember and use but easy to decipher too. Longer the password it is difficult to break and also to remember and key- in. The trade off results in a password of length 6-8 characters. Salting is a technique to make it difficult to break a password. Salting technique appends a random number „n" to the password before encryption is done. Just knowing the password is not enough. The system itself calculates stores and compares these random numbers each time a password is used. Multiple passwords at different levels could provide additional security. Change of password at regular intervals is a good practice. Many operating systems allow a user to try only a few guesses for a login after which the user is logged off the system.

f> Protection Mechanism

System resources need to be protected. Resources include both hardware and software. Different mechanisms for protection are as follows:

Files need to be protected from unauthorized users. The problem of protecting files is more acute in multi- user systems. Some files may have only read access for some users, read / write access for some others, and so on. Also a directory of files may not be accessible to a group of users.

For example, student users do not access to any other files except their own. Like files devices, databases, processes also need protection. All such items are grouped together as objects. Thus objects are to be protected from subjects who need access to these objects.

Access Matrix:

The operating system allows different access rights for different objects.

For example, UNIX has read, write and execute (rwx) rights for owners, groups and others. Possible access rights are listed below:

No access
Execute
only Read
only
Append
only
Update
Modify
protection rights
Delete

A hierarchy of access rights is identified. For example, if update right is granted then it is implied that all rights above update in the hierarchy are granted. This scheme is simple but creation of a hierarchy of access rights is not easy. It is easy for a process to inherit access rights from the user who has created it. The system then need maintain a matrix of access rights for different files for different users.

		OBJECTS							
		File 0	File 1	File 2	File 3	File 4	File 5	Printer 0	Printer 1
DOMAINS	0	R W	R					W	
	1			R	R W X				W
	2					W	R W X		W
	3			W		R			

The operating system defines the concept of a domain. A domain consists of objects and access rights of these objects. A subject then gets associated with the domains and access to objects in the domains. A domain is a set of access rights for associated objects and a system consists of many such domains. A user process always executes in any one of the domains. Domain switching is also possible. Domains in the form of a matrix are shown in Figure above.

A variation of the above scheme is to organize domains in a hierarchy. Here also a domain is a set of access rights for associated objects. But the protection space is divided into „n" domains from 0 to (n-1) in such a way that domain 0 has maximum access rights and domain (n-1) has the least. Domain switching is also possible. A domain switch to an outer domain is easy because it is less privileged whereas a domain switch to an inner domain requires permissions.

Domain is an abstract concept. In reality domain is a user with a specific id having different

access rights for different objects such as files, directories and devices. Processes created by the user inherit all access rights for that user. An access control matrix showing users and objects (files) needs to be stored by the operating system in order to decide granting of access rights to users for files.

Since the matrix has many holes, storing the entire matrix is waste of space. Access control list is one way of storing the matrix. Only information in the columns is stored and that too only where information is present that is each file has information about users and their access rights. The best place to maintain this information is the directory entry for that file.

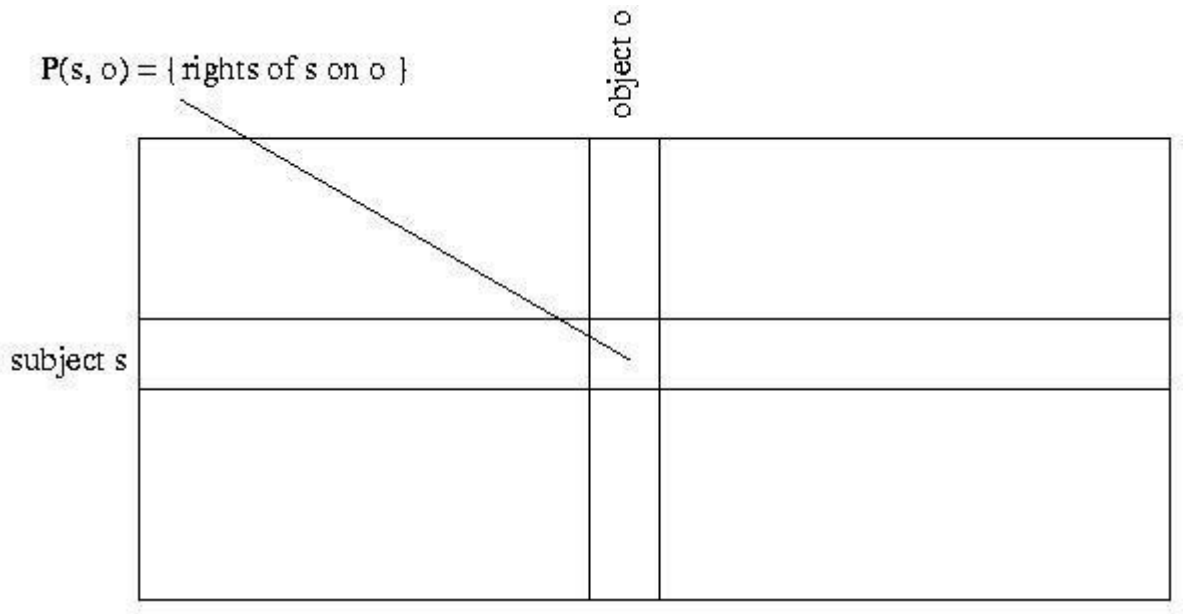
An Another Example

A model of protection abstracts the essential features of a protection system so that various properties of it can be proven.

O = current objects -- finite set of entities to which access is to be controlled, e.g., a file, a process *S* = current subjects -- finite set of entities that access current objects, e.g., a process

note: S ⊂ O

R = generic rights, R = { r_1, r_2, ..., r_m } e.g., read, write, execute, delete
P = access matrix, indexed by (*subject*, *object*)
Protection state of a system = (S,
O, P)



P(s, o) ⊂ R
Enforcement of Model
Every object has a monitor that validate all accesses to that object in the following manner:

1. A subject s requests an access α to object o.
2. The protection system presents triplet (s, α, o) to the monitor of o.
3. The monitor looks into the access rights of s to o. If α ∈ P(s, o), then the access is permitted, otherwise it is denied

Example of Access Matrix

	o_1	o_2	s_1	s_2	s_3
s_1	read, write	own, delete	own	sendmail	recmail
s_2	execute	copy	recmail	own	block, wakeup
s_3	own	read, write	sendmail	block, wakeup	own

Implementation Methods

- capability list = collapsed row of access matrix
- **access control list = collapsed column of access matrix**
- **lock-key = combination**

Capability Lists

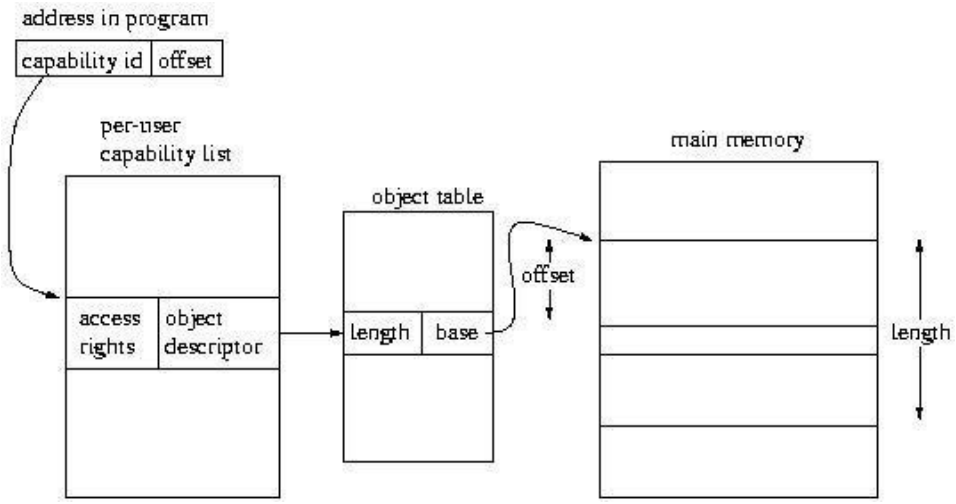
Capability list is another way of storing the access control matrix. Here information is stored row wise. The operating system maintains a list of files / devices (objects) that a user can access along with access rights.

- capability = tuple (o, $P(s,o)$)
- each subject has a set of capabilities
- possession of capability confers access rights
- Capability Based Addressing:

A schematic view of a capability Object descriptor Access rights

read, write, execute, etc.

- The object descriptor can be the address of the corresponding objects and therefore, aside from providing protection, capabilities can also be used as an addressing mechanism by the system.
- A user program issues a request to access a word within an object. The address of the request contains the capability ID of the object and an offset within the object.



- How does this relate to segmented/paged virtual memory?
- Implementing Capabilities
  - tagged memory -- one or more bits are attached to memory location or register to indicate if it contains a capability
  - partitioned memory -- capabilities and ordinary data are saved separately
- Capability Advantages and Disadvantages
  - efficiency - no need to explicit check permissions for each access
  - simplicity - a single mechanism, based on adressability
  - flexibility - can control access by granting capabilities
  - propagation control - can pass on rights
  - review of access - who currently has rights to an object? difficult to determine as one has to search through all programs and data structures for copies of the capabilities
  - revocation - very difficult to revoke rights
  - garbage collection - when can object be reclaimed?

A combination of both access control list and capability list is also possible.

Access Control List

CSE Prof. Vishal M. Tiwari
Department of CSE,TGPCET

1. list contains pairs (s, $P(s,o)$)
2. when subject s requests access a of object o
   - search ACL of o for entry corresponding to s
   - check whether this entry contains a
3. ACL Example for a file:

Subjects	Access Rights
Arnold	read, write, execute
George	read
Michael	write
Tong	execute
Smith	read, write

4. ACL Advantages and Disadvantages
   - slow access time
   - difficult review of accessibility, by subject
   - easy revocation
   - easy review of acessibility, by object
   - amenable to protection groups, which saves storage
   - right to change protections easily controlled
5. Changing Protections
   - self control -- by owner, the control is centralized to one process
   - hierarchical control -- with a tree of processes; the owner specifies a set of other processes which have the right to modify the access control list of the new object

The Lock-Key Method

1. each subject has capability list of tuples (o,k), where k is a key
2. each object has ACL of tuples (l,A), where A is a set of access modes, l is a 'lock'
3. when s wants access a of object o
   i. find a tuple (o,k) in s's capability list (if not found, access is declined)
   ii. find a matching tuple (l,A) in o's ACL, such that $k = l$, $a \in A$.
4. revocation is easy

g> Encryption

It is used to provide data security. Store and transmit information in an encoded form that does not make any sense.

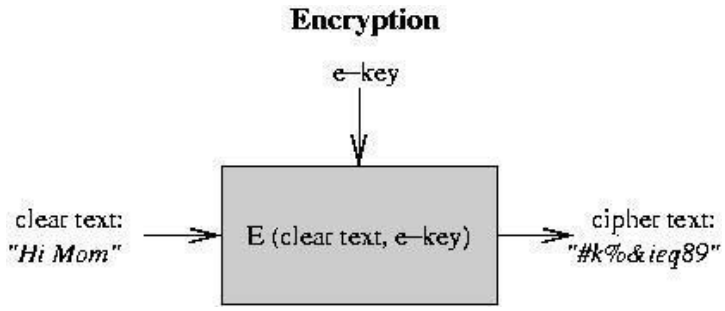
The process involves two steps

Encryption: the original message is changed to some other form Decryption: the encrypted message is restored back to the original

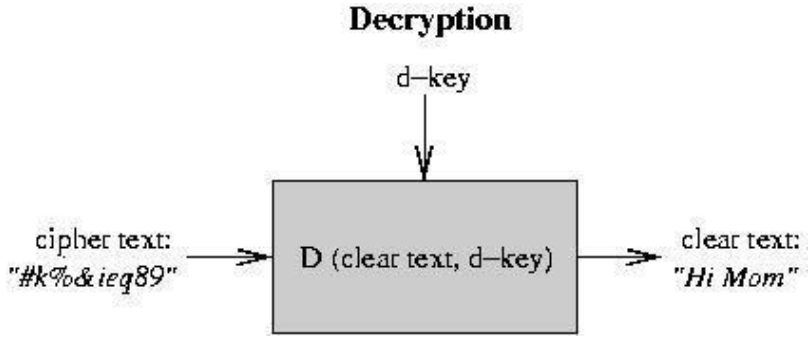
The basic mechanism:

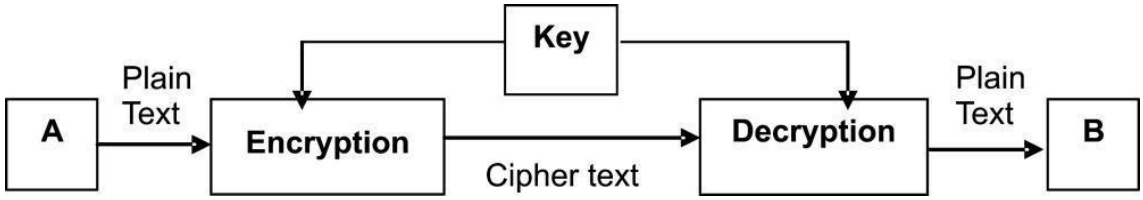
- Start with text to be protected. Initial readable text is called *clear text* (or plain text).
- Encrypt the clear text so that it does not make any sense at all. The nonsense text is called *cipher text*. The encryption is controlled by a secret password or number; this is called the *encryption key*.

Prof. Vishal M. Tiwari
Department of CSE,TGPCET



- The encrypted text can be stored in a readable file, or transmitted over unprotected channels.
- To make sense of the cipher text, it must be *decrypted* back into clear text. This is done with some other algorithm that uses another secret password or number, called the *decryption key*.





All of this only works under three conditions:

- The encryption function cannot easily be inverted (cannot get back to clear text unless you know the decryption key).
- The encryption and decryption must be done in some safe place so the clear text cannot be stolen.
- The keys must be protected. In most systems, can compute one key from the other (sometimes the encryption and decryption keys are identical), so cannot afford to let either key leak out.

Public key encryption: new mechanism for encryption where knowing the encryption key does not help you to find decryption key, or vice versa.

- Consider an example:
  - Question 1: What is the product of 31415926538979 x 31415926538979 ?
  - Question 2: What is square root of 391257150641938709059482850824 1 ?
- User provides a single password, system uses it to generate two keys (use a one-way function, so cannot derive password from either key).
- In these systems, keys are inverses of each other: could just as easily encrypt with decryption key and then use encryption key to recover clear text.
- Each user keeps one key secret, publicizes the other. Cannot derive private key from public. Public keys are made available to everyone, in a phone book for example.

Encryption procedure E and decryption procedure D must satisfy the following properties:

1. for every message M, D(E(M)) = M
2. E and D can be efficiently applied to any message M
3. it is extremely hard to derive D from E

Safe mail:

Use public key of destination user to encrypt mail.

Anybody can encrypt mail for this user and be certain that only the user will be able to decipher it.

It is a nice scheme because the user only has to remember one key, and all senders can use the same key. However, how does receiver know for sure who it is getting mail from?

Does such a scheme exist?

The **RSA** (Rivest-Shamir-Adleman) scheme:

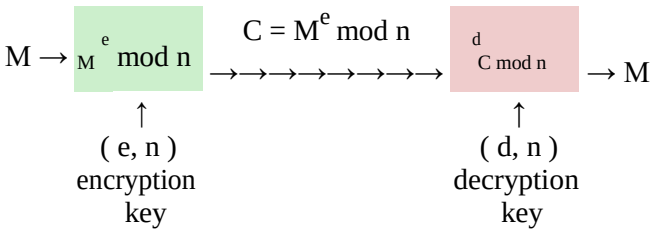
- encryption key is a pair (e, n) where e is a positive integer
- a message block M is expressed as a value between 0 and n-1 inclusive
- M is encrypted to give cipher text C by :

$$C = M^e \bmod n$$

(note that C also lies between 0 and n - 1)

- the decryption key is a pair (d, n) where d is a positive integer
- C is decrypted to M by:

$$M = C^d \bmod n$$



- how to find the encryption and corresponding decryption keys?
  1. choose two large prime numbers p, q and calculate n by

$$n = p \times q$$

  2. choose any large integer as d so that the chosen d is relatively prime to (p - 1) x (q - 1). i.e.

$$GCD(d, m) = 1, \text{ where } m = (p - 1) \times (q - 1$$

) 3. Compute e as the multiplicative inverse

of d: e x d = 1 mod m

- Example:
  - Assume p = 5 and q = 11. Thus,

$$n = 55,$$
$$m = (p - 1) \times (q - 1) = 4 \times 10 = 40$$

- We choose d = 23 because 23 is relatively prime to 40

 ○ We need to choose e satisfying

$$23 \times e \bmod 40 = 1$$

- e = 7 satisfies the above equation
- Some sample computations:

M	M^7	encrypt: C = $M^7 \bmod 55$	C^{23}	decrypt: M = $C^{23} \bmod 55$
8	209152	2	8388608	8
9	4782969	4	70368744177664	9
51	897410677851	6	789730223053602816	51

h> Digital Signatures

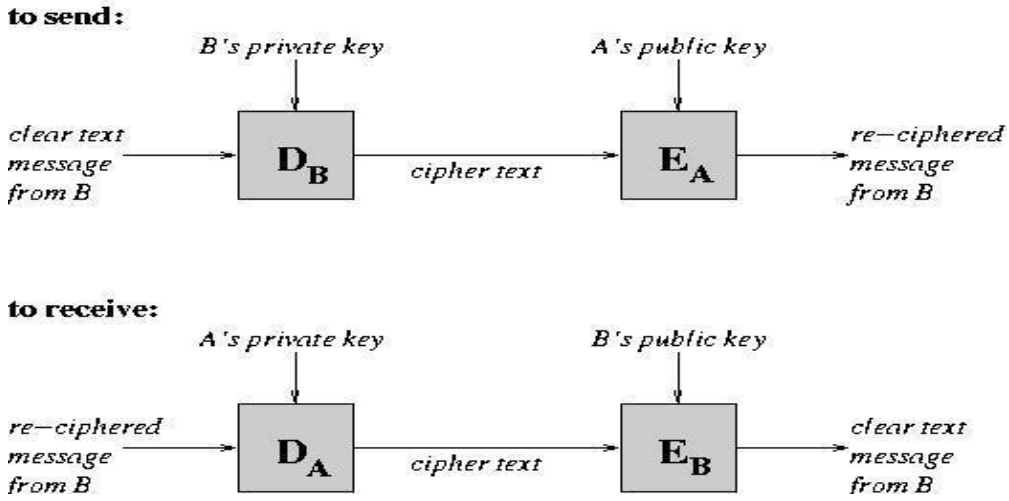
Positive identification: can also use public keys to certify identity:

a. To certify your identity, use your private key to encrypt a text message, e.g. "I agree to pay Mary Wallace $100 per year for the duration of life."
b. You can give the encrypted message to anybody, and they can certify that it came from you by seeing if it decrypts with your public key. Anything that decrypts into readable text with your public key *must* have come from you! This can be made legally binding as a form of electronic signature.

These two forms of encryption can be combined together. To identify sender in secure mail, encrypt first with your private key, then with receiver's public key. The encryption/decryption functions to send from B to A are:

$$\text{encrypted text} = E(D(P, \text{d-key}_B), \text{e-key}_A)$$
$$\text{decrypted text} = E(D(P, \text{d-key}_A), \text{e-key}_B)$$

Digital Signatures



Encryption appears to be a great way to thwart listeners. It does not help with Trojan Horses, though.

Old Data Encryption Standard (DES) is not public-key based, but as implemented efficiently and appeared to be *relatively* safe.

New Advanced Encryption Standard (AES), called Ryndal (pronounced "rine doll").

General problem: how do we know that an encryption mechanism is safe? It is extremely hard to prove. This is a hot topic for research: theorists are trying to find provably hard problems, and use them for proving safety of encryption.

Summary of Protection: very hard, but is increasingly important as things like electronic funds transfer become more and more prevalent.

i> Security in Distributed Environment

Security problems in a distributed environment are complex. Messages through a network can be tapped at multiple locations. For an active attack the intruder gets control over a link so that data modification / deletion is possible. For a passive attack the intruder just listens to a link and uses the passing information. Encryption in a distributed environment can be of two forms:

End-to-end
encryption Link
encryption

If **end-to-end encryption** is used, the encryption / decryption devices are needed only at the ends. Data from source to destination moves on the network in encrypted form. In packet switched networks, data is sent in the form of packets. Each packet has control information (source address, destination address, checksum, routing information, etc.) and data. Since routing address is needed for the packet to hop from the source till it reaches the destination, the control information cannot be encrypted as there is no facility to decrypt it anywhere in between. Only the data part in a packet can be encrypted. The system thus becomes vulnerable for tapping.

Link encryption needs more encryption / decryption devices, usually two for each link. This allows total encryption of a packet and prevents tapping. The method is expensive and slow.

A combination of both is possible.

Message authentication allows users to verify that data received is authentic. Usually the following attributes of a user need to be authenticated:

- Actual message
- Time at which sent
- Sequence in which sent
- Source from which it has arrived

Common methods for message authentication are:

- Authentication code
- Encryption
- Digital signatures

In authentication code, a secret key is used to generate a check sum, which is sent along with the data. The receiver performs the same operation using the same secret key on the received data and regenerates the check sum. If both of them are same then the receiver knows the sender since the secret key is known to only both of them.

Encryption is as discussed above where conventional encryption provides authentication but suffers from key distribution problems and public key encryption provides good protection but no authentication.

Digital signature is like a human signature on paper. If a signed letter is sent by A to B, A cannot deny having sent it to B (B has the signed copy) and B cannot refuse having got it (A has an acknowledgement for B having received it). This is what happens in a manual system and should happen in electronic messages as well.

As discussed earlier, public key encryption provides protection but not authentication. If we want to authentication without protection, reversal of the keys applied is a solution as shown below.

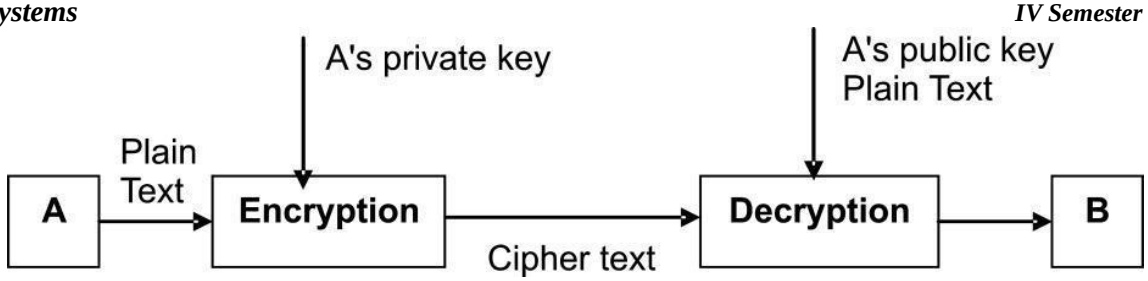


Figure : Public key Encryption for authentication without protection

This is based on the concept that public key encryption algorithm works by using either of the keys to encrypt and the other for decryption. A encrypts the message to be sent to B using its private key. At the other end B decrypts the received message using A‟s public key which is known to everybody. Thus B knows that A has sent the message. Protection is not provided as anyone can decrypt the message sent by

A. If both authentication and protection are needed then a specific sequence of public and private keys is used as show below. The two keys are used as shown. At points 2 and 4 the cipher text is the same. Similarly at points 1 and 5 the text is the same. Authentication is possible because between 4 and 5 decryption is done by A‟s public key and is possible only because A has encrypted it with its private key. Protection is also guaranteed because from point 3 onwards only B can decrypt with its private key. This is how digital signatures work.

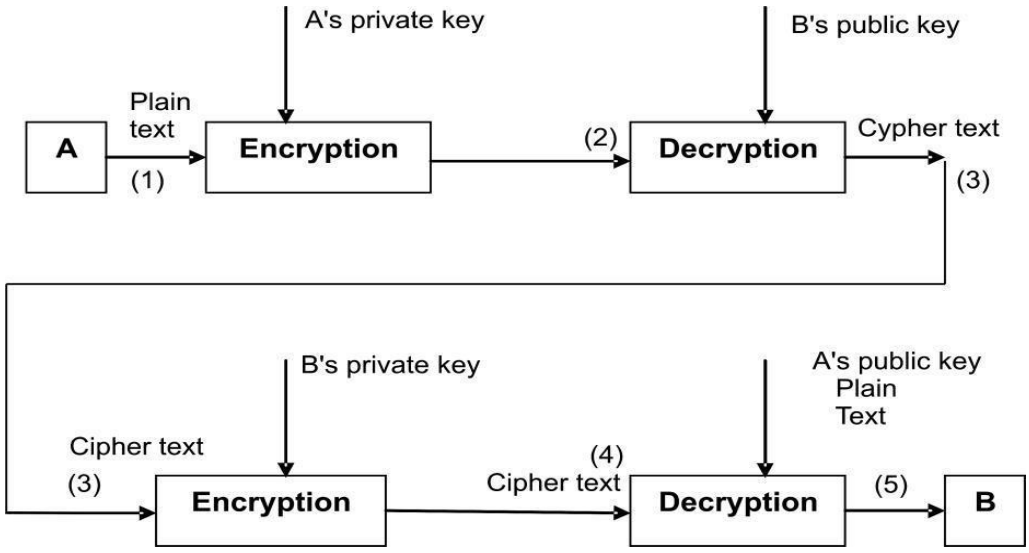


Figure: Public key Encryption for both authentication and protection

j> **Wi Fi Security**

1. **What is PKI**?
   - Public Key Infrastructure
   - Collection of digital certificates:
     1. objects that bind identity information to keys using distinguished names
     2. used to verify identities of servers/services or individuals/clients
   - Used to provide authentication, non-repudiation
2. PKI Components
   - Digital certificates
   - Digital signatures
   - Certificate Authority
   - Key management protocols
   - Public key -- distributed, preferably through a centralized directory; used to encrypt data
   - Private key -- used to decrypt or electronically sign data; preferably protected with passphrase
3. Digital Certificates
   - Issued to or generated by an owning entity (client or server)

     --Often issued by trusted authority for authentication systems

   - Contains identifying information

     --Owner name, owner public key, key validity timeframe, issuer identity

   - Can contain additional information for specific applications
4. Certificate Authority (CA)

- Responsible for issuing certificates
- Trust aggrega

Process control File

- Performs identity verification for certificate requests
- Signs public keys of entities that prove their identity
- Public and private CA's

5. CA - Certificate Management
   - CA's must accommodate key revocation (CRL)

     -- Entities need a method to recover from a compromised key
     -- Verifying parties should check CRL before authenticating identity

   - May provide key recovery services

     -- Lost private keys, forgotten passwords

6. Key Management Protocols
   - X.509 used for most PKI implementations
   - Key contains two sections:
     1. Data section includes identity, use information, public key, CRL location
     2. Signature section includes algorithm, encrypted hash of identity section data
   - Signature section signed by CA's private key
7. Key File Formats
   - DER (Distinguished Encoding Rules) -- Certificate in ASN.1 file format

     --Includes .der, .cer file extensions

   - PEM (Privacy Enhanced Mail) -- Base64 encoded DER file
   - PKCS#12 (Public Key Crytography Standard #12) -- Storage of private and associated public keys, password
   - PKCS#7 -- Format to disseminate certificates (such as a CA certificate)
8. Trust and Key Distribution
   - Trust is a critical component of PKI
   - In large PKI deployments, impossible to trust everyone directly

     --Trust is extended through relationships with other trusted entities

   - Trust can be centrally managed, or distributed
9. Wireless Network Authentication Architecture
   - Deploying 802.1x assumes user database exists

     --Microsoft AD, LDAP, Cisco Secure ACS, Sun iPlanet, etc.

   - Integrity of user database influences security of WLAN
10. User Database Recommendations
    - Enforce strong password selection
    - Audit regularly for weak passwords

      --Expire weak passords, force reset

    - Enable failed login account lockout
    - Monitor accounts for signs of abuse
    - Consider time-based authorization

      --Do users require to access WLAN 24-hours a day?

    - Limit number of simultaneous logins ○

Grant access to limited user population

Process control File