OPERATING SYSTEM

CHAPTER 1: INTRODUCTION

Overview of operating system, Evolution of operating system, Difference services of operating system.

Operating system for Main frame computer systems: Batch processing system, Micro programmed system, Time-sharing system. Understanding multiprogramming, Multiprocessing and Multitasking.

Operating system for Multiprocessor systems and Distributed systems, Operating system for client server & Peer-to-peer system, Clustered system, Real time operating system.

➢ Operating System:

- An Operating system is the program that manages the computer & hardware. It also provides a basic of application program are as an intermediate between user of a computer and computer hardware.
- The purpose of an Operating system is to provide an environment in which a user can execute program.

➢ What is an operating system?

- An Operating system is an important part of every computer system.
- Operating system is a collection of program that acts as an interface between user and the hardware of the computer system.
- The system can be divided into four components & they are as follows:

1. Hardware
2. Operating System
3. Application Program
4. User

➢ Hardware:

- It provides basic computing resources CPU, Memory and input/output devices.

➢ Operating System:

- It controlled and co-ordinates use of hardware among various application & users.

➢ Application Programs:

- It defined the way in which the system resource are use to solve the computing problems of the users.
- E.g.: Word processer, Web Browser, Database System.

➢ Users:

- (People, machines, other computers).

➢ Operating System Services:

Operating System provides an environment for the execution of program. It provides certain services to program & to the user of that program. To make that task of programming easier. They are as follow:

1. Program Execution:

In Program Execution system must be load into memory & run that programs. The program must be able to end its execution either normally or abnormally.

2. Input/output Operation:

Since the running program may require input/output which may involve a file or an input/output device cannot be directly control by the user. The operating system must provide the means to the input/output operations.

3. File System manipulation:

The program needs to create, delete & access the file. Also it is used to create & delete them by name, searching for a given file and list file information.

4. Communication:

In many cases one process needs exchange information with other process. On the same or different machine. The operation system should support some mechanism for interposes communication.

5. Error Detection:

The operating system takes appropriate action for various types of error. To ensure correct & consisting computing errors occurs in CPU & memory hardware, in input/output devices & user program.

6. Resource Allocation:

When multiple user or jobs are running at the same time resources must be allocated to each of them. There are many types of resources which are managed by the operating system.

7. Accounting:

This service is used to keep track on how and what kind of resources are been used by the user. The statistic calculated is useful for the researchers for reconfiguring or improving the system.

8. Protection and Security:

This service is particularly useful to owner who wanted to keep control to access of the information on the multi-user system.

➢ Evolution of operating system:

1. Mainframe system:

A) Batch System:

Early computers are physically used machines. The common input devices work card readers. The common output devices like printers & card punches. The use did not interact directly with the computer system. The user prepared job in the form of punch card & submitted to the computer operator. After sometime the output appeared which consisted of the result of the program. As well as the dump of final memory &register contain of all defaulting job consisted of program, data & some control information. The operating system was to transfer control automatically from one job to the next. The operating system was always resident in the memory. To speed up the processing job with the similar needs are batch together and work has run to the group of the computer. Resorting of program in to batches having similar requirement done by computer each batch was done have the computer available. In such system the CPU was often idle. Because the speed different between the mechanical input & output devices & electrical devices.

2. Multiprogramming System:

In Multiprogramming system the operating system gives service job in the memory simultaneously. The set of job is the subset of job kept in the job to. The operating system speaks & begins to execute one of the jobs in the memory. One of the jobs may have to wait for some task such as input/output operation. In such a situation the CPU is switched to another job by the operating system.

As long as at least one job need to execute the CPU is never idle. Multiprogramming increases CPU utilization. In such system the operating system must make decisions for the user. Operating system must choose jobs to be loaded in memory form the job to the successful

required job having overall job in the memory require some form of memory management. CPU scheduling is required means several jobs are ready to use to the CPU. Multiple job running concurrently require the ability to affect the one another be restricted all phase of as including process scheduling big storage & memory management.

3. Time sharing / multitasking system:

The time sharing or the multitasking is the logical extension of multiprogramming. The time share operating system allows many users to share computer simultaneously since each action or command given by the users through the input device tends to the short only a little CPU time which needed for each user. As the system switches rapidly from one user to next each user is given the impression that entire system dedicated to use.

The time share operating system uses CPU scheduling & multiprogramming provides each user with the small portion of the time shared computer. Each use has at least one separate program in memory. A program loaded in memory & executing is commonly referred as process. Times sharing operating system are even complex than multiprogramming operating system since several jobs must be kept simultaneously in the memory. So the system must have memory management & protection. The program may need to swap in & out of the main memory to the disk.

Hence to achieve this goal virtual memory technology was needs. The time sharing system must also provide file system. The file system decides on the disk hence disk management must is required. The time shared operating system required supporting CPU scheduling skill. The operating system must provide for job synchronization & communication & deadlock avoid. Once for the orderly executed of multiple programming.

➢ Multiprocessor system:

Multiprocessor system have more than one processor in a close communication for sharing memory & peripheral devices, potential benefits of multiprocessing include increased performance & computing power, increased fault tolerance flexibility & modular growth. It allows the system to two more work in short period of time.

1. Separate supervisor:

In the separate supervisor system each road contents separate operating system that manages local processor, memory input/output resources. Only few additional services & data structured may be added to support multiprocessor accepts of hardware. The operating systems provide services such as local process & memory management & implement the message passing primitives.

2. Master/slave:

In Master/slave approach one processor is dedicated to execute the operating system. The remaining processors are usually identical and form a tool of computational processor.

3. Symmetric:

In symmetric multiprocessoring model each processor runs and an identical of operating system & these copies communicate with each other as needed.

➢ Distributed system:

The processors communicate with one another through various communication lines, such as high-speed buses or telephone lines. These systems are usually referred to as distributed system may vary in size & function. They may include small microprocessors workstations, minicomputers & large general purpose computer system. These processors are referred to by a number of difference names, such as sites, nodes, computers, & soon. Depending on the context in which they are mentioned.

There is a variety of reasons for building distributed system, the major ones being these.

1. Resource sharing:

If a number of different sites (with different capabilities) are connected to one another, then a user at one site may be able to use the resources available at another.

E.g. A user at site A may be using a laser printer available only at site B mean while, a user at B may access a file that resides at A. In general resource sharing in a distributed system provides mechanism for sharing files at remote sites, processing information in a distributed database, printing files at remote sites, using remote specialized hardware devices & performing other operations.

2. Computation speed up:

If a particular computation can be partitioned into a number of sub computations that can run concurrently then a distributed system may allow us to distribute the computation among the various sites to run that computation concurrently. In addition, if a particular site is currently overloaded with jobs, some of them may be moved to other lightly loaded sites. This movement of jobs is called load sharing.

3. Reliability:

If one site fails in a distributed system, the remaining sites can potentially continue operating. the system is composed of number of large autonomous installations (i.e. general purpose

computers). The failure of one of them should not affect the rest. If on other hand, the system is composed of a number of small machines, each of which is responsible for some crucial system functions (such as terminal character input/output or the file system). Then a single failure may effectively half the operation of the whole system. In general if sufficient redundancy exists in the system, the system can continue with its operations, even if some of its sites have failed.

4. Communication:

There are many instances in which programs need to exchange data with one another or one system window system are one example, since they frequently share data or transfer data between display. When many sites are connected to one another by a computation network, the processes at different sites have the opportunity to exchange information. Users may initiate file transfer or communicate with one another via electronic mail. A user can send mail to another user at the same site or at a different site.

➢ Clustered system:

1. Clustered system gathered multiple CPU's to accomplish computational work. Clustered systems shared storage and are closely linked via LAN, MAN, WAN.
2. Clustered is usually performed to provide high availability a larger of cluster software each node can monitor one or more nodes of the other node of the LAN.
3. If the monitor machine fails, the monitoring machine can take ownership of its storage & restart the applications that were running on the failed machine.
4. The failed machine can remain down but the users. And the clients of the applications could see a brief interruption of service.
5. In asymmetric clustering, one machine is in standby mode while other is running the applications. The standby machine does nothing but monitor the active server if the server fails, the standby post become the active server.
6. In symmetric mode two or more modes a running application & they are monitoring each other. Other form of clusters includes parallel clusters.
7. A parallel cluster allows multiple hosts to access the same data on the shared storage.

➢ Real-Time System:

1. A Real time system is used when rigid time requirement have been placed on the operation of a processor or the flow of data.
2. Such systems are often used in control devices in dedicated application such as those that control scientific experiment, medical imaging system, home applying controller, nuclear system & weapon system.
3. Real time has well defined and fixed time constraints processing must be done within the define constraints, or the system will fail.

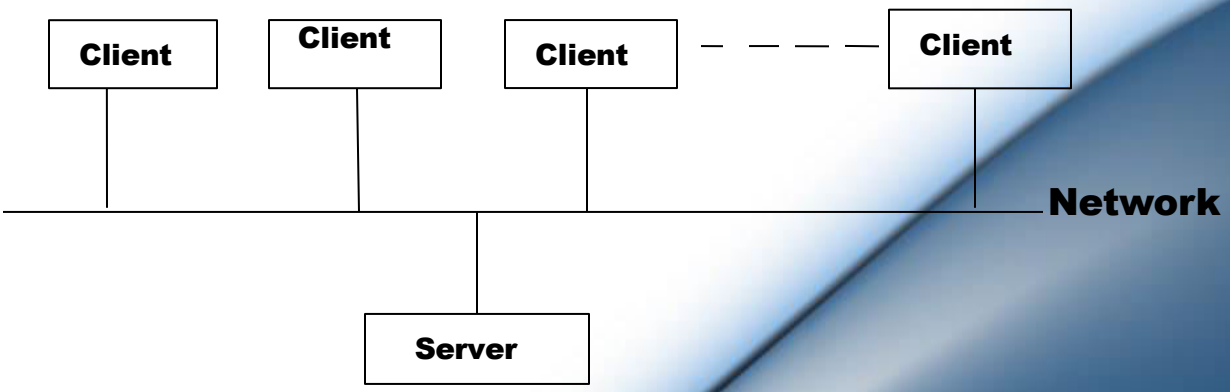
4. The hard real time system guaranties that critical task be completed in time. This goal requires that all delays in the system be bounded from the retrial of the stored data to the time that it takes the operating system to finish any request made to it.
5. A soft real time system is the one in which a critical real time task gets priority over other task & it retains that priority until it completes. The operating system does not have many of the advanced features.

➢ Client –server computing:

As PCs have become faster, more powerful, and cheaper, designers have shifted away from centralized system architecture. Terminals connected to centralized system are now being supplanted by PCs. Correspondingly; user- interface functionality once handled directly by the centralized system is increasingly being handled by the PCs. As a result, many of today's systems act as server system to satisfy requests generated by client systems. This form of specialized distributed system, called client-server system, has the general structure depicted in figure.

Server system can be broadly categorized as computer servers and file servers:

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



- **The file server system:** It provides a file system interface where clients can create, updates, read and delete files. E.g. web server that delivers file to client running web browsers.

➢ Peer-to-peer computing:

1. Peer to peer system the computer networks that consist of a collection of processors that do not share memory or a clock, instead, each processor has its own local memory.
2. The processors communicate with one another through various communication lines, such as high-speed buses or telephone lines.
3. These system are usually referred to as loosely coupled system
4. The different operating system communicates closely enough to provide the illusion that only single operating system controls the network.
5. In this network, there is no centralized computer that manages and processes shared data instead they are equipped with a operating system that enables them to store and share
   resources with each other.

CHAPTER 2: COMPONENTS OF OPERATING SYSTEM

Process management, Main memory management, Secondary storage management, File management, I/O management, Operating system services, Command interpreter, Interface between user and Operating system. Introduction to System calls: Types of System calls.

➢ Operating System Component:

Operating systems provide mechanism for supporting and abstract for computation for the process and form managing resources used by the group of processes. It also addresses the various practical considerations relating to performance protection, correctness, and maintainability and commercial factors. The operating system functions can be classified into categories.

➢ Process Management:

- A process is a program in execution. A program is a passive entity such as programs stored on secondary memory (hard disk, pain drive), whereas a process is an active entity.
- A process needs resources line CPU time memory, files & input/output devices to perform its task.
- These resources are given to the process either at its time of creation or while it is running.
- When a process terminates the OS are reclaim any reusable resources.

- The execution of such a process must be sequential. The CPU executes one instruction of the process after another until the process completes.
- A multithreaded process has multiple program counters, each pointing to the next instruction to execute for a given thread.
- The operating system is responsible for the following activities in connection with process management:

  ○Creating & deleting user & system processes.
  ○ Suspending & resuming process.
  ○Providing mechanism for process synchronization. ○
  Providing mechanism for process communication. ○
  Providing mechanism for deadlock handling.

➢ Main Memory Management:

- For any program to be executed, it must be mapped to absolute address and loaded into memory.
- AS a program executes, it access instructions and data from memory.
- Main memory is large array of words or bits, ranging in size from hundreds of thousands to billions. Each words or bit having its own address.
- Main memory is a repository of quickly accessible data shared by the CPU and I/O devices.
- The main memory is generally the only large storage device that the CPU is able to address and access directly.
- When the execution of process is completed, then its memory space is declared available and next program can be loaded and executed.
- So to improve more utilization of CPU and speed memory management is require.
- The operating system is responsible for the following activities in connection with process management:

  ○ Keeping track of which parts of memory currently being used and by whom.
  ○ Deciding which processes and data to move into and out of memory.
  ○ Allocating and de-allocating memory space as needed.

➢ File Management:

- A file is collection of related information defined by its creation. A file represents programs and data.
- Computers can store information on several different types of physical media. Magnetic disk, optical disk and magnetic tape are the most common.
- Each of these media has its own characteristics and physical organization. Each medium is controlled by a device such as a disk drive or tape drive that also has its properties include access speed, capacity, data transfer rate, and access method.
- Files are normally organized into directories to make easier to use.

- When multiple users access to files, it may be desirable to control by whom and what ways files may be accessed.
- The operating system is responsible for the following activities in connection with process management:

    - Creating & deleting files & directories.
    - Supporting primitives for manipulating files & directories.
    - Mapping files on to the secondary storage.
    - Backing up files on stable (non volatile) storage media.

➢ Input/output Management:

- One of the purposes of an operating system is to hide the peculiarity of specific hardware devices from the user.
- The peculiarities of I/O devices are hidden from the bulk of OS itself by the I/O subsystem.
- The following are the responsible of operating system with regards of the input/output system management.

    - Allocated efficiency & fairly the devices to the various processes.
    - Manage allocation, isolation & sharing of devices according to the desire policy.

➢ Secondary storage system management:

- Main purpose of the computer system is to execute program.
- These program with the data the access, must be in the main memory during the execution.
- Main memory is too small to accommodate all data & programs, and its data are lost when power the lost, the computer system must provide secondary storage to0 back up main memory.
- Most computer systems use disks as storage mediums for both data and programs.
- The operating system is responsible for following activities in connection to the disk management:
    - Free space management.
    - Storage allocation.
    - Disk scheduling.

➢ Command interpreter:

- It is one of the most important system programs for an operating system.
- User can interface with OS with the technique of command interpreter. It is a command line interface, which allows users to directly enter commands that are to be performed by OS.
- Some OS include the command interpreter in the kernel; others like windows XP and UNIX treat the commands as a special program that is running when a job is initiated.
- Many commands are given to the operating system by control statements. When the new job is started in a batch system, or when a user logs on to a time-shared system, a program that reads and interprets control statements is executed automatically.
- This program is sometimes called the control-card interpreter or the command-line interpreter, and is often known as shell.
- Main function of the command interpreter is to get and execute the next user specified command like delete, print, copy, execute and so on.
- There are two ways to in which these commands can be implemented and are as follows:
    - The command interpreter itself contains the code to execute the command. For example, a command to delete a file may cause the command interpreter to jump to a section of its code that sets up the parameters and makes appropriate system call.
    - Another approach used by UNIX is, it implements most of commands through system programs.

➢ System Calls:

- System calls provides the interface between a running program and operating system.
- System calls are generally available as in assembly language instruction.
- System calls occur in different ways, depending upon the computer in use. The system call identifies itself by some number.
- Languages defined to replace assembly language for systems programming allow system calls to be made directly (e.g., C, Bliss, PL/360).
- Three general methods are used to pass parameters between a running program and the operating system:
    - Pass parameters in *registers*.
    - Store the parameters in a table in memory and the table address is passed as a parameter in a register.
    - *Push* (store) the parameters onto the *stack* by the program, and *pop* off the stack by the operating system.

➢ **System calls are of five types:**

a) Process control:

- Process control system calls provide interface for following purpose:

- End, Abort
- To terminate the process normally (end) or abnormally (abort)
- Load, Execute
- To load and run the program in memory
- Create process, Delete process
- Get process attribute, Set process attribute
- For initializing or retrieving process attributes
- Allocate time and free memory
- Wait for time

b) File Management:

- Several common system calls deals with files. These system calls requires some file attributes like name, type, size of file and so on….

- Create file, delete file.
- Open, close.
- Read, Write, reposition.
- Get file attribute, Set file attribute.

c) Device Management:

- Many program read resources for its execution

- Request device, Release device
- Read, Write, reposition
- Get device attribute, Set device attribute.
- Logically attach or detach devices.

d) Information Maintenance:

- system calls exit simply for the purpose of transferring information between user program and operating system.

- This information may be like:

○Get time or date, Set time or date time
○Get system date, Set system date. ○
Get process, file, device attribute ○
Set process, file, device attribute.

e) Communication:

- To establish communication between two processes operating system providing many system calls:

- Create, delete communication connection
- Send, Receive messages
- Transfer status information
- Attach or detach remote devices

CHAPTER 3: SYSTEM PROGRAM AND OPERATING SYSTEM STRUCTURE

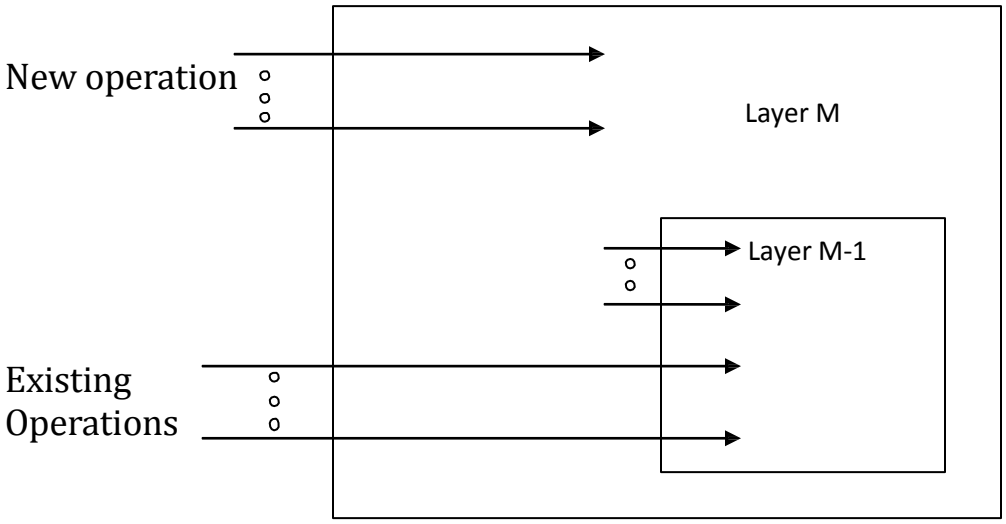
Layered approach, advantages &disadvantages of layered approach, microkernel.

➢ Layered Approach:

- The modularization of a system can be done in many ways. One method is the layered approach, in which the operating system is broken up into a number of layered (or levels), each built on top of lower layers.
- The bottom layer (layer 0) is the hardware; the highest (layer N) is the user interface.
- An operating system layer is an implementation of an abstract object that is the encapsulation of data, and of the operation that can manipulate those data.
- It consists of data structures and a set of routines that can be invoked by higher-level layers.

- **Advantages of layered approach:**
- **Modularity:**

- This is the main advantage of the layered approach.
  - The layers are selected such that each uses functions (or operations) and services of only lower-level layers.



An operating-system layer.

Simplifies debugging and system verification:

- This approach simplifies debugging and system verification. The first layer can be debugged without any concern for the rest of the system, because, by definition, it uses only the basic hardware to implement its functions.
- Thus, the design and implementation of the system are simplified when the system is broken down into layers.
- Each layer is implemented with only those operations provided by lower-level layers.
- A layer does not need to know how these operations are implemented; it needs to know only what these operations do.
- Hence, each layer hides the existence of certain data structures, operations, and hardware from higher-layers.

➢ Disadvantages:

- The main three disadvantages of layered approach are as follows:
- The major difficulty with the layered approach involves the careful definition of the layers, because a layer can use only those layered below it. For example, the device driver for the disk space used by virtual-memory algorithms must be at a level lower than that of the memory-management routines, because memory-management requires the ability to use the disk space.
- Other requirements may not be so obvious. The backing-store driver would. Normally be above the CPU scheduler, because the driver may need to wait for 1/0 and the CPE can be rescheduled during this time. However, on a large system, the CPU scheduler may have more information about all the active processes than can fit in memory. Therefore, this information may need to be swapped in and out of –memory, requiring the backing-store driver routine to be below the CPU scheduler.
- A final problem with layered implementations is that they tend to be less efficient than other types. For instance, when a user program executes an 1/0 operation, it executes a system call that is trapped, to the 1/0 layer which calls the memory-management layer, which in turn calls the CPU-scheduling layer, which is then passed to the hardware. At each layer, the parameters may be modified; data may need to be passed, and so on. Each layer adds overhead to the system call; the net result is a system calls that take longer than does one on a no layered system…

➢ Microkernel:

- Microkernels typically provide minimal process and memory management, in addition to a communications facility.
- The main function of the microkernel is to provide a communication facility between the client program and the various services that are also running in user space.
- Communication is provided by message passing, which was described.
- For example, if the client program wishes to access a file, it must interact with the file server. The client program and the service never interact directly. Rather, they communicate indirectly by exchanging messages with the microkernel
- The benefits of the microkernel approach include the ease of extending the operating system.
- All new services are added to user space and consequently do not require modification of the kernel. When the kernel does have to be modified, the changes tend to be fewer, because the microkernel is a smaller kernel.
- The resulting operating system is easier to port from one hardware design to another.
- The microkernel also provides more security and reliability, since most services are running more user-rather than kernel- processes.
- If a service fails, the rest of the operating system remains.

CHAPTER 4: PROCESS MANAGEMENT

Introduction to process: process states: two states and five state models, processes & resources, concurrent processes, process description, process control block and its role, operation on process, cooperating processes.

➢ Five state process model:

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

  1. **New:** The process is being created.
  2. **Running:** Instructions are being executed.
  3. **Waiting:** The process is waiting for some Event to occur.
  4. **Ready:** The process is waiting to be assigned to a processor.
  5. **Terminated:** The process has finished execution.

- These state names are arbitrary, and they vary across operating systems. The states that they represent are found on all systems, however.
- Certain operating systems more finely delineate process states. Only one, process can be running on any processor at any instant, although many processes may be ready and waiting. The state diagram corresponding to these states is

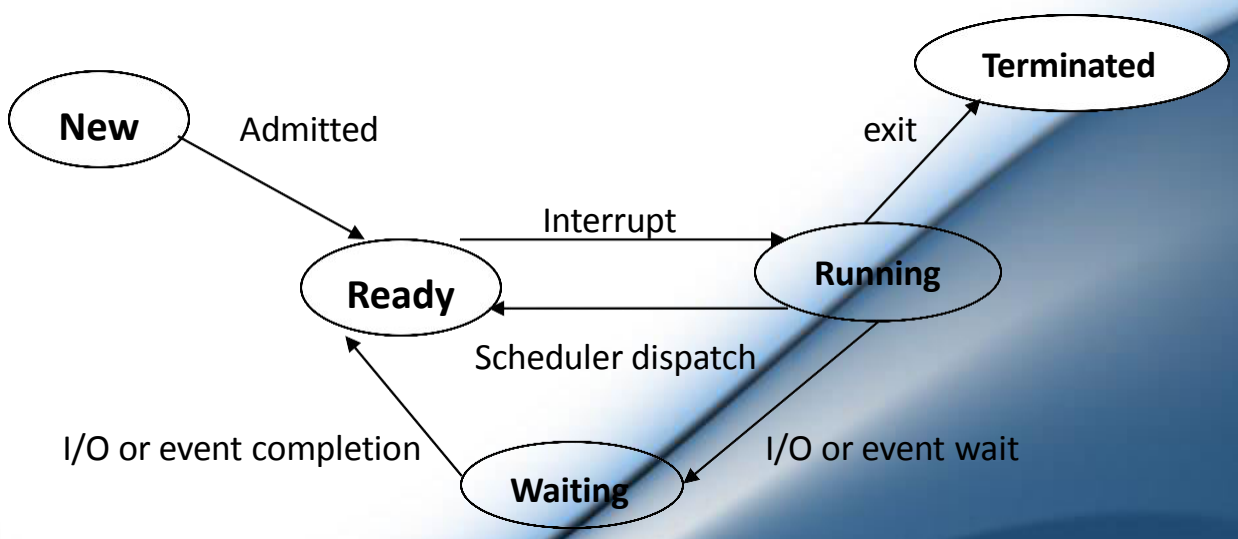
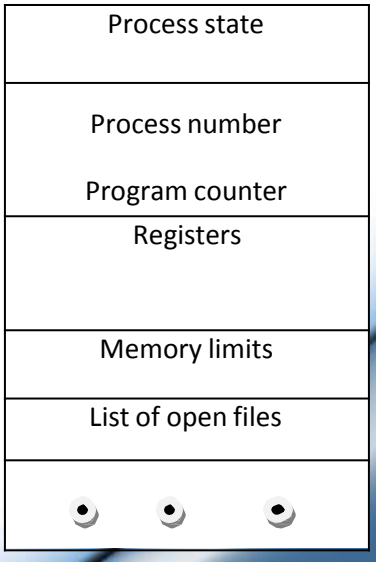


Diagram of process state

➢ Process Control Block:

Each process is represented in the operating system by a process control block (PCB) also called a task control block. A PCB is shown in fig. it contains many pieces of information associated with a specific process, including these:

- **Process state:** The state may be new, ready, running, and 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. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.
- **CPU-scheduling information:** This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- **Memory-management information:** This information may include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system.
- **Accounting information:** This information includes the amount of CPU and real time used, time limits, account 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.
- In brief, the PCB simply serves as the repository for any information that may vary from process to process.



Process control block (PCB)

➢ Operations on process:

The processes in the system can execute concurrently, and they must be created and deleted dynamically. Thus, the operating system must provide a mechanism for process creation and termination.

1. Process Creation:

- A process may create several new processes, via a create-process system call, during the course of execution.
- The creating process is called a ―parent process‖, whereas the new processes are called the children of that process. Or ―child process‖
- Each of these new processes may in turn create other processes, forming a tree of processes.
- In general, a process will need certain resources (such as CPU time, memory, files, I/O devices) to accomplish its task. When a process creates a sub process, that 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 restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many sub processes.
- When a process is created it obtains, in addition to the various physical and logical resources, initialization data (or input) that may be passed along from the parent process to the child process.
- When a process creates a new process, two possibilities exist 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 child process is a duplicate of the parent process
  - The child process has a program loaded into it.

2. Process Termination:

- A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit system call. At that point, the process may return data (output) to its parent process (via the wait system call).
- All the resources of the process-including physical and virtual memory, open files, and I/O buffers-are deallocated by the operating system.
- Termination occurs under additional circumstances. A process can cause the termination of another process via an appropriate system call (for example, abort).usually, only the parent of the process that is to be terminated can invoke such a system call.

- Otherwise, users could arbitrarily kill each other's jobs. A parent therefore needs to know the identities of its children.
- Thus, 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 these:

  - The child has exceeded its usage of some of the resources that it has been allocated. This requires the parent to have a mechanism to inspect the state of its children.
  - 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. On such system, if a process terminates, then all its children must also be terminated. This phenomenon, referred to as cascading termination, is normally initiated by the operating system.

➢ Co-operating process or Inter process communication:

- Processes executing concurrently in the operating system may be either

A) Independent processes:

A process is independent if it cannot affect or be affected by the other processes executing in the system.

B) Co-operating processes:

A process is co-operating if it can affect or be affected by other processes executing in system.

- There are several reasons for providing an environment that allows process co-operation.

1) Information Sharing:

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

2) Computation Speedup:

If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with others. Speedup can be achieved only if the computer had multiple processing elements.

3) Modularity:

We want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.

4) Convenience:

Even an individual user may work of many tasks at the same time. For instance, a user may be editing, printing, and compiling in parallel.

➢ Message passing system:

The function of a message system is to allow processes to communicate with each other without the need to resort to shared data. It provides to operations:

1. **Send** (message)
2. **Receive** (message)

- Message send by the process can be of fixed size or variable size.
- Depending upon it system implementation is done as for the first one is concerned it quiet straightforward but it's slightly complex for the variable one.
- If two processes want to communicate then, communication link must be exit between them.
- Process that want to communicate must have a way to refer to each other. They can use Direct or Indirect communication.

➢ Direct communication:

- In this, each process that wants to communicate must explicitly name the recipient or sender of the communication.
- In this technique they use two primitives,

I. Send (P, Message)-Send a message to process P
II. Receive (Q, Message)- Receive a message from process Q

- A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.
- Here, link is associated exactly with two processes and between each pair of process, there exits exactly one link.
- This scheme exhibits symmetry in addressing.

Disadvantage associated with this:

a. Limited modularity of the resulting process definitions.
b. Changing the name of a process may necessitate examining all other process definitions.
c. All reference to the old name must be found, so that they can be modified to the new name. This situation is not desirable from the view point of separate compilation.

➢ Indirect communications:

- With indirect communication the messages are sent to and received from mailboxes or ports.
- A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed.
- Each mailbox will have a unique identification number. In this scheme, a process can communicate with some other process via a number of different mailboxes.
- Two processes can communicate only if they have shared mail-box.
- While primitives are defined as follows:

a. Send (A, message)
b. Send a message to mailbox A.
c. Receive (A, message)
d. Receive a message from mailbox A.

➢ Points:

- A link is established between a pair of processes only if both members of the pair have a shared mailbox.
- A link may be associated with more than two processes.
- Between each pair of communicating processes, there may be number of different links, with each link corresponding to one mailbox.

➢ Process synchronization:

- Communication between processes take place by calls to SEND () & RECEIVE () primitives.
- Message passing can be Blocking & Unblocking also known as synchronous & asynchronous.

- Blocking Send: Sending process is blocked until message is received by the receiving process or by the mailbox.
- Non-Blocking Send: Sending process is never blocked/The sending process sends the message & resumes operations.
- Blocking Receiver: receiver process is blocked until message is available.

- Non-Blocking Receiver: receiver process is never blocked. /The receiver retrieves either a valid message or a null.
- Different combinations are possible here.

- Two of them are as follows:

➢ Blocking send & blocking receive:

- Here, both sender and receiver are blocked until the message is delivered; this is sometimes referred to as rendezvous.

➢ Non-Blocking send & blocking receive:

- Here, although sender is allowed to continue to send messages, the receiver is blocked until the requested message arrives. This is most useful combination.

➢ Buffering –

- Buffer is a memory area that stores data while they are transferred between two devices or between a device and an application.
- Buffering is done for three reasons:

1. To cope with a speed mismatch between the producer and consumer of a data stream.
2. To adapt between devices that have different data transfer sizes. Such disparities are common in computer networking where buffers are widely used for fragmentation and reassembly of messages.
3. To support copy semantics for application input/output.

- Whether the communication is direct or indirect message exchanged by communicating processes reside in a temporary queue.
- There are three ways to implement these queues:

➢ **Zero capacity -**the queue has maximum length of **zero**; therefore link cannot have any messages. Hence, sender block technique must be used for synchronization.

➢ **Bounded capacity –** here, queue has finite length. If queue is not full then messages send by the sender or a pointer associated with it is saved. However, if queue is full then receiver cannot accept the message hence sender must be blocked until free space is available in the queue.

➢ **Unbounded capacity –** the queue has potentially infinite length; thus any number of messages can wait in it. The sender never blocks.

CHAPTER 5: THREADS

Threads, single &multithreaded process, user and kernel threads, multithreaded models.

➢ Thread:

- A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack.
- Threads are a small part of process. Thread must execute in process.
- In process, a thread allows multiple executions of streams.
- Thread is shares with other threads belonging to the same process its code section, data section, & other operating system resources, such as open files & signals.
- Types of Threads:

1. **User Thread**
2. **Kernel Thread**

➢ User thread:

- User threads are managed by a User thread Library above kernel in the user space.
- User level threads are generally fast to create & manage.
- The User thread Library manages and schedules user threads. No kernel interference.
- The disadvantage is that if kernel is single threaded, even if user is multithreaded, a single thread block will remaining user threads.

➢ Kernel Thread:

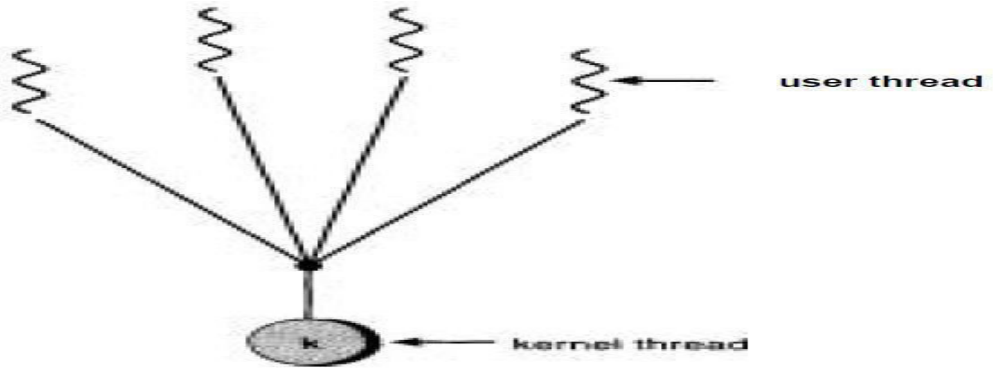
- Kernel threads are managed and scheduled in the kernel space.
- The kernel threads are slowly scheduled & managed in the kernel space. No user interference.
- Kernel Threads are slower to create & manage.
- Even if kernel is single threaded, if one thread is blocked, others can be scheduled by kernel to run.

➢ Multithreaded Models:

- **Many-to-One Model**
- **One-to-One Model**
- **Many-to-Many Model**

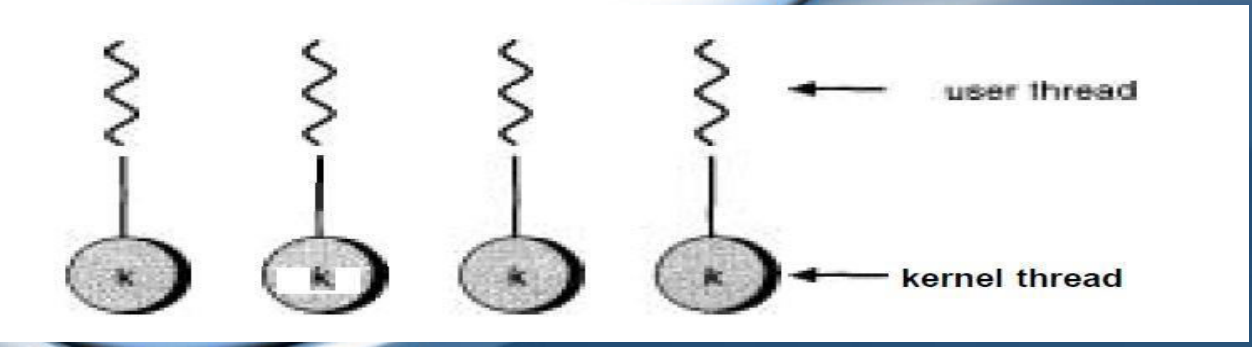
➢ Many-to-One Model:

- The many-to-one model maps many user level threads to one kernel thread.
- Thread management is done by Thread Library in user space.
- The entire process will block if a thread makes a blocking system call.



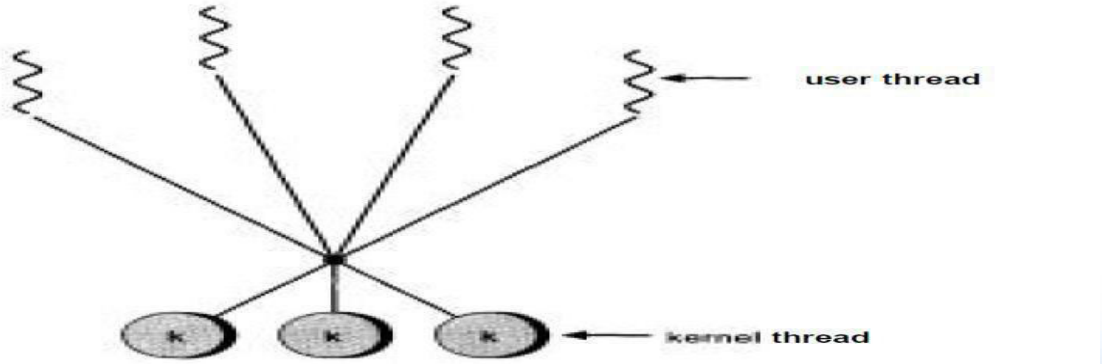
➢ One-to-One Model:

- One-to-one model maps each user thread to a kernel thread.
- If provides move concurrency than the many-to-one model. By allowing another thread to run. When a thread makes a blocking system call it also allows multiple threads to run in parallel on multiprocessors.
- The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread.
- Because the overhead of creating kernel threads can burden the performance of an applications.

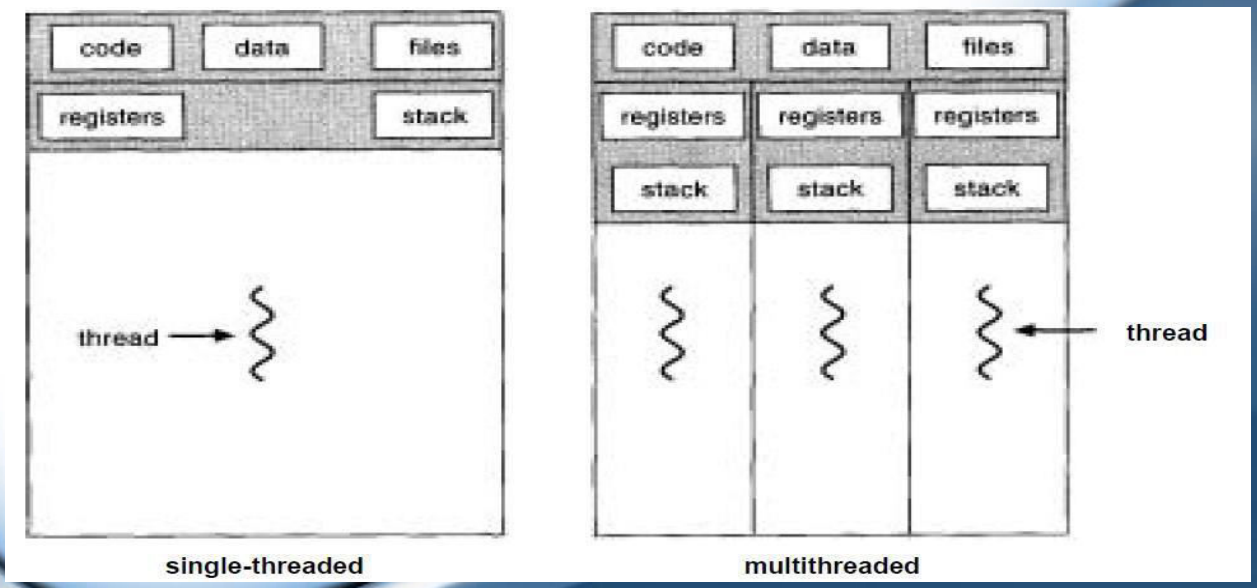


➢ Many-to-Many Model:

- Many-to -many model maps many user level threads to smaller or equal number of kernel threads.
- The number of kernel threads may be specific to either a particular application or particular machines.
- The many-to-many model suffers from neither of these short comings: developers can create as many user thread as memory, and the corresponding kernel threads can run in parallel on a multiprocessor.
- Also, when a thread performs a blocking system call the kernel can schedule another thread for execution.
- One of the variations on the many-to-many model still multiplexes many user level threads to a smaller or equal number of kernel threads but also allows a user level thread to be bound to a kernel threads.



➢ Single & multithreaded process:



- In certain situations, a single application may be required to perform several. Similar tasks for E.g.: A web server accepts client requests for web pages, images, sound and so forth.
- A busy web server may have several clients concurrently accessing it. If a web browser may have several single- threaded processes, it would be able to service only one client at a time, and client may have to wait a very long time for its request to be serviced.

- One solution is to have server run a single process that accepts requests.
- When a server receives a request, it creates a separate process to receive that request. Process creation is time consuming and resources intensive.
- It is generally move efficient to use one process that contains multiple threads. If a web browser process is multithreaded, the server will create a separate thread that listens for client's requests, when a request is made, rather than creating another process, the server will create a new thread to service that request and resume listening for additional requests.

CHAPTER 6: CPU SCHEDULING

Need for process scheduling, queuing diagram, scheduler and its types, Scheduling is the basis of multiprogramming operating system, CPU scheduling takes place under circumstances, Scheduling criteria, context switching, CPU Scheduling algorithms.

➢ Need of Scheduling:

- CPU scheduling is the basic of multiprogrammed OS.
- The objective of multiprogrammed OS is to have some processes running all the times, to maximize CPU utilization.
- For unprocesser system there will never be more than one running process. The idea of multiprogrammed is simple.
- A process is executed until it must wait, typically for the completion of some I/O request.
- In simple computer system, the CPU would then just sit idle. All this waiting time is wasted; no useful work is accomplished.
- With multiprogrammed we try to use this time productively. Several processes are kept in memory at one time.
- When one process has to wait, the OS takes the CPU away from that process and gives CPU to another process.
- Scheduling is fundamental OS function. Almost all computer system resource is scheduled before use.
- Whenever the CPU becomes idle, the OS must select one of the processes in ready queue to be executed.
- The selection process is carried out by short term scheduler. The scheduler selects from among the processes in memory that are ready to execute, and allocate the CPU to one of them.

➢ Process scheduling with neat queuing diagram:

- As the processes enter the system, they are put into a job queue.
- The processes that are residing in main memory and are ready and waiting to execute are kept on a list called ready queue. Generally queue is stored as linked list.
- Each PCB has a pointer field that points to the next process in the ready queue.
- There are other queues also in the system. When a process is allocated CPU, it executes for a while and eventually quits, or is interrupted or waits for I/O completion.
- There are many processes in the system, and disk might be busy with I/O request of some other processes.
- The processes therefore have to wait for the disk. The list of processes waiting for a particular I/O device is called a device queue.
- A common representation of process scheduling is queuing diagram as follows:

- There are two types of queues present ready queue and a set of device queues. The circles represent the resources that serve the queues, and arrows indicate the flow of processes in the system.
- A new process is initially put the ready queue. It waits in the queue until it is selected for execution and is given the CPU. Once the process is allocated the CPU and is executing, one of several events could occur:
-

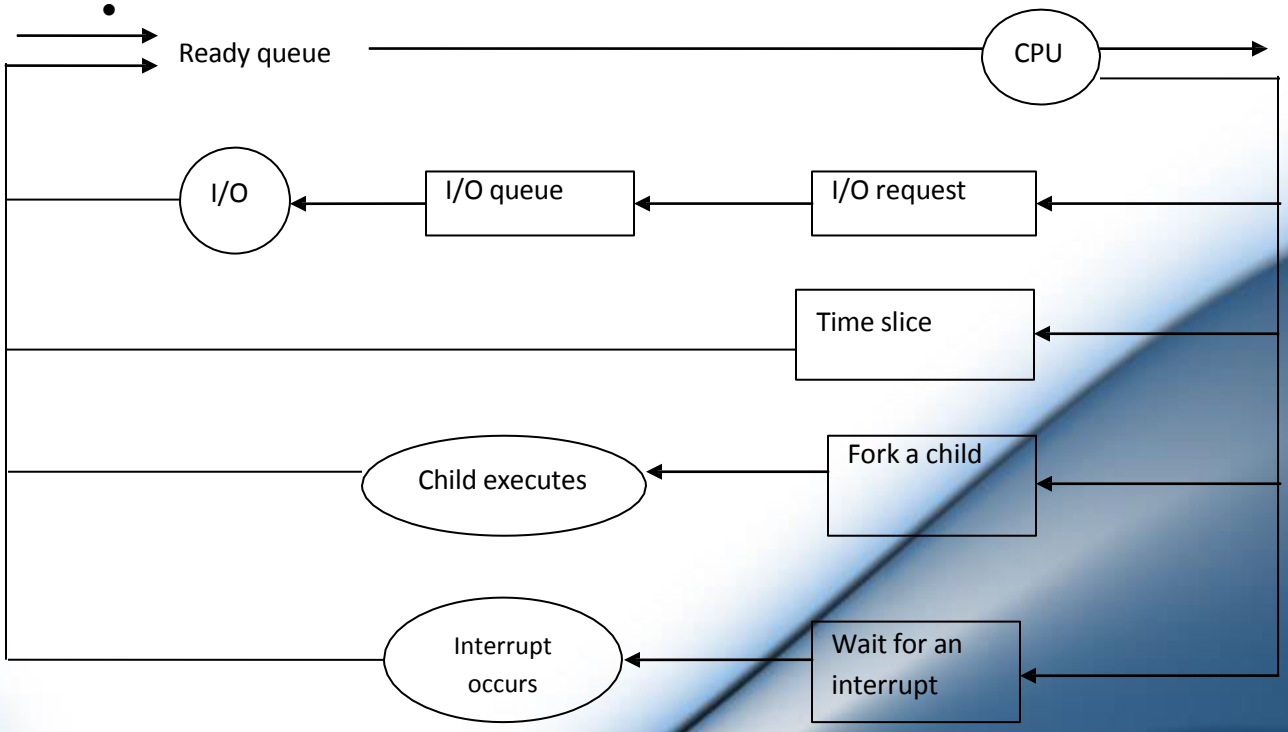


Fig: Queuing diagram representation of process scheduling.

- The process could issue an I/O request, and then be placed in an I/O queue.
- The process could create new sub process and wait for its termination.
- The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

➢ Scheduling is the basis of multiprogramming operating system:

- In a uniprocessor system only one process can run at a time.
- In Multiprocessing system multiple processes are kept in a memory.
- In multiprocessing when process or become idle operating system selects one of the process in the ready queue.

➢ CPU scheduling takes place under following circumstances:

1. When a process switches from the running state to waiting state.
2. When a process switches from running state to ready state.
3. When a process switches from waiting state to ready state.
4. When a process terminates.

- Scheduling has to be done for condition one (1) & four (4) .It are called as non- preemptive OR co-operative scheduling.

➢ Scheduling criteria:

- ***CPU utilization:*** We have to keep CPU as busy as possible .It can range from 0to 100%.
- ***Through put:*** It is work in terms of number of processes completed per unit time.
- ***Turn-around time:*** It is a time required for switching between two processes.
- ***Waiting time:*** Amount of time process spends in a ready process.
- ***Response time:*** It's time between submission of the request & response from processor .It's desired to maximize CPU utilization & throughput & to minimize turn-around, waiting & response time.

➢ Function of different types of schedulers:

- A process migrates between 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.
- There are three types of scheduler there are as follows:

➢ Short term scheduler:

- Short term scheduler or CPU scheduler, select from among these processes that are ready to execute, and allocate the CPU to one of them.

- The primary distinction between these two schedulers is the frequency of their execution. The short term scheduler must select a new process for the CPU frequently.
- A process may execute for only a new milliseconds before waiting for an I/O request. Often, the short term scheduler executes at least once every 100 millisecond. Because of the brief time between executions; the short term scheduler must be fast.

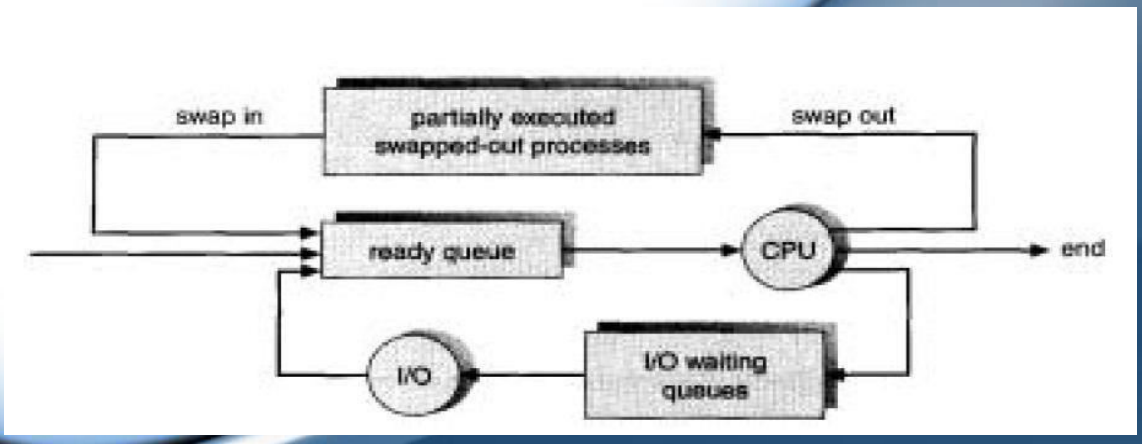
➢ Long term scheduler:

- It execute much less frequently. There may be minutes between the creations of new processes in the system.
- The long term scheduler controls the degree of multiprogramming the number of process in the memory.
- If the degree of multiprogramming is stable, then the average rate of process creation is equal to average rate of the process leaving the system.
- The long term scheduler must make a careful selection. In general, must processes can be described as either I/O bound or CPU bound. I/O bound process spends more time in I/O operation and CPU bound process spends more time in doing computation.
- The long term scheduler should select a good process mix of I/O bound and CPU bound processes.

- This must be there because a system with the best performance will have a combination of CPU bound and I/O bound processes.

➢ Medium term scheduler:

- This is an intermediate level of scheduling. This removes processes from memory and thus reduces the degree of multiprogramming.
- At some later time, the process can be continuing where it left off. Thus scheme is called swapping in, by medium term scheduler.
- Swapping may be necessary to improve the process mix, or because a change in memory requirements has overcommitted available memory, requiring memory to be freed up.



➢ Context switching:

- The CPU to another process requires saving the state of the old process and loading the saved state for the new process. This task is known as a context switch.
- The context of a process is represented in the PCB of a process; it includes the value of the CPU registers, the process state, and memory-management information.
- 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.
- Context-switch time is pure overhead, because the system does no useful work while switching. Its speed varies from machine to machine,
- A context switch simply includes changing the pointer to the current register set. Of course, if active processes exceed register sets, the system resorts to copying register data to and from memory, as before.
- Also the more complex the operating system, the more work must be done during a context switch.
- How the address space is preserved, and what amount of work is needed to preserve it, depend on the memory-management method of the operating system.
- But context switching has become such a performance bottleneck that programmers are using new structures to avoid it whenever possible.

➢ CPU SCHEDULING ALGORITHMS:

1. First-come, First-served scheduling
2. Shortest-Job First scheduling
3. Priority scheduling
4. Round-Robin scheduling

1. First-come, First-served scheduling:

- A process that request for the CPU first is processed first/allocated the CPU first .Hence the name first come, first serve.
- It is similar to FIFO queue; this queue has a head & a tail.
- Whenever process enters a ready queue .It enters at the end of the queue .When CPU is processes the first process in the queue. /It is allocated to the process at the head of the queue.
- Running process is then removed from the queue.

2. Shortest-Job First scheduling:

- This algorithm assigns the length of process time required to each process.
- When CPU is available it is assigned to the process that has shortest length.
- If two processes have same processing time then FCFS approach is used.

- Problem with this algorithm knows the length of next process to execute.
- It cannot be applied to short term processing.

3. Priority scheduling:

- This is the general case in which SJF comes.
- In this a priority is associated with each process and CPU is allocated to process with highest priority.
- Equal priority processes are processed with FCFS approach.
- Major problem of priority is indefinite blocking.
- A solution to the above problem is aging. Here, priority is increased with respect to time.

4. Round-Robin scheduling:

- It is specially designed for time sharing systems.
- A small unit of time called time quantum ranging between 10to 100 milliseconds is added.

- The CPU scheduler allocates each process in the ready queue one time quantum at a time in round-robin scheduling
- .If processing time required for the process is less than the time quantum then after completion of the process it is released & scheduler moves to the next process.
- If processing their required for the process is more than the time quantum then timers goes off, interrupt is caused context switch gets executed & process is putted at the end of the queue.

CHAPTER 7: PROCESS SYNCHRONIZATION

Critical section problem and its solution, Semaphores, binary semaphores, monitor, dining philosopher problem, Dekker's algorithm, Peterson's algorithm and their final correct solution for two processes, The Bakery Algorithm or Multiple Process Solution.

➢ Critical-section problem & its solution:

- A system consisting of n processes {p0, p1,.pn-1}.
- Each process has a segment of code, called critical section, in which process may be changing common variables, updating a table, writing a file, and so on.
- The important feature of the system is that, when one process is executing in its critical section, no other process is allowed to execute in its critical section.

- Thus, the execution of the critical sections by the processes is mutually exclusive in time. The critical section problem is to design a protocol that the processes can use to cooperate.
- Each process must request permission to enter its critical section.

```
do
{
Entry section
   Critical section
Exit section
    Remainder section
} while (1);
```

General structure of typical process p1

- Solution to critical section problem must satisfy the following three requirements.

1. Mutual Exclusion:

- If process p1 is executing in its critical section, then no other processes can be executing in their critical section.

2. Progress:

- If no other process is executing in its critical section and there exit some processes that wish to enter their critical sections, then only those processes that are not executing in their remainder section can be participated in the decision of which will enter its critical section next, and this selections cannot be postponed indefinitely.

3. Bounding Waiting:

- There exists a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

➢ Short Note on SEMAPHORE

- A semaphore is an integer variable that apart from initialization, two standard atomic operations: Wait & Signal.

```
Wait(S): while S <=0 do no-op;
S: =S-1;
Signal(S): S: =S+1;
```

- Modification to the integer value of the semaphore in the Wait & Signal operation must be executed indivisibly.
- That is one process modifies the semaphore value no other process can simultaneously modify that same semaphore value.
- We can use semaphore to deal with the n- process critical section problem. The n processes share a semaphore, mutex (standing for mutual exclusion), initialized to 1.
- We can use semaphore to solve various synchronization problems. For example, consider two concurrent running processes: p1 with a statement s1 and p2 with a statement s2.
- Suppose that we require that s2 be executed only after s1 has completed.

```
do
{
  Wait (mutex);
        Critical section
      Signal (mutex);
Remainder section
    } while (1);
```

➢ The semaphore as a variable that has an inter value upon which the following three operations are defined:

- A semaphore may be initialized to a nonnegative value.
- The Wait operations decrement the semaphore value. If the value becomes negative, then the process executing the Wait is blocked.
- The Signal operation increments the semaphore value. If the value is not negative, then a process blocked by a Wait operation is unblocked.

➢ Definitions:

```
   Struct Semaphores
{
      Int count;
      Queue_type queue;
};
Void wait (semaphore s)
{
      If (s.count>0)
      {
            S.count--;
      }
      If (s.count<0)
```

```
        {
                Place the process in s.queue;
                Block this process;
        }
}
Void signal (semaphore s)
{
        If (s.queue.Is-empty ())
                S.count++;
        Else
        {
                Remove process P from queue;
                Place process P on ready list;
        }       }
```

➢ Binary Semaphore

- A binary semaphore may only take on the values 0 & 1. In principle, it should be easier to implement the binary semaphore.
- The process that has been blocked the longest (FIFO) is released from the queue first; semaphore where definition includes this policy is called a Strong semaphore.
- A semaphore that does not specify the order in which processes are removed from the queue is called a Weak semaphore.

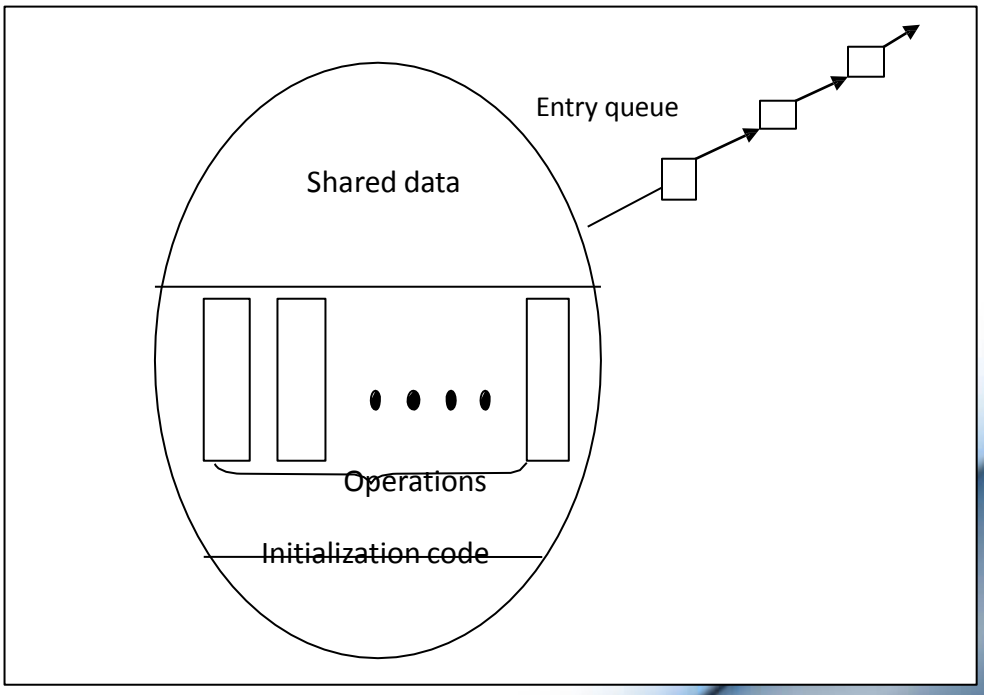
```
Struct binary-semaphore
{
        Enum (zero, one) value;
        Queue_type queue;
};
Void wait_b (binary-semaphore s)
{
        if (s.value==1)
                s.value=0;
        Else
        {
                Place the process in s.queue;
                Block this process;
        }
}
Void signal b (semaphore s)
{
        if(s.queue.Is-empty())
        {
                s.value=1;
        }
        Else
        {
```

```
            Remove the process P from s.queue;
            Place process P on ready list;
        }
}
```

➢ Short Note on Monitors

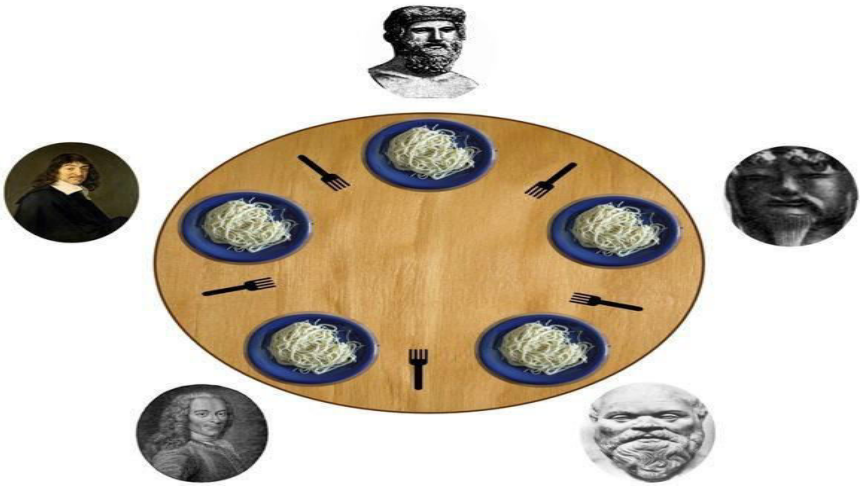
- A monitor is characterized by a set of programmer defined operators.
- The representation of a monitor type consists of declarations of variable whose values define or function that implement operation
- Two variables operate on condition variables.

- **C Wait(c):** suspend execution of calling process on condition c. The monitor is now available to be used by another process.
- **C Signal(c):** resume execution of some process suspend after C Wait on the same condition. If there are several such processes then choose one of them; if there is no such process do nothing.



Schematic view of a monitor.

➢ A monitor solution to the dining philosopher problem.



- Five philosophers spend their lives in thinking and eating.
- In the center of the table there is a bowl of rice and table is laid with five single chopsticks.
- Times to time philosopher's gets hungry and try to pick up two chopsticks that are closer to them.
- Problem is philosopher can only pick up only one chopstick at a time and cannot pick up the chopstick that is already is in the hand of neighbor.
- After eating philosopher releases both the chopsticks.

➢ To prevent deadlock have to follow one of the way.

- At the most only four philosophers are allowed.
- They must be allowed to pick up the chopsticks only if both are available.
- Asymmetric solution odd philosopher must first pick up the first left chopstick and then the right. While even ones first pick up the first right and then left.

➢ Solution to the problem

- enum { thinking, hungry, eating } state [5];
- Philosopher 1 can set the variable state[i]= eating only if her two neighbors are not eating:(state[(i+4)%5]=eating) and (state[(i+1)%5]!=eating).

- Where Philosopher 1 can be delay herself when she is hungry, but is unable to obtain the chopsticks she needs.

```
Monitors dining philosophers
{
enum { thinking, hungry, eating } state[5];
Condition self[5];
Void pickup(int i)
{
State [i]=hungry;
Test(i)=hungry;
If ( state[i] =eating)
Self[i].wait();
}
Void putdown (int i)
{
State[i]=thinking;
test (i+4)%5);
Test(i+1)%5);
}
Void test(int i)
      {

if((state[(i+4)%5]=eating) && (state[i]==hungry ) &&
(state[(i+1)%5)!=eating))
{
State[i]=eating;
Self[i].signal();
}
}
Void init( )
{
for( int i=0; i<5; i++)
State[i]= thinking;
}
}
```

➢ The Bakery Algorithm or Multiple Process Solution

- Bakery algorithm is used to solve the problem of critical section for n processes. On entering the store, each customer receives a number.
- The customer with the lowest number is served next. Unfortunately, the bakery algorithm cannot guarantee that the two processes do not receive the same number.
- In the case of tie, the process with lowest name is served first. That is ,if pi and pj receives the same number and if i<j, then pi is served first.

- Since process names are unique and totally ordered, this algorithm is completely deterministic.
- The common data structures are:

```
        Boolean chosing[n];
            Int number[n];
Initially these data structure are initialized to false and 0, respectively.
(a, b)<(c, d) if a<c or if a==c and b<d.
Max ⌈a0,…..., an-1⌉ is a number k such that k>=ai for i=0,……, n-1.
Do{
Choosing [i] =true;
Number [i] = max ⌈number [0], number [1],……, number [n-1]) +1;
Choosing [i] = false;
For (j=0; j<n; j++)
{
      While (choosing [j]);
      While ((number[j]! =0) && (number [j, i] <number [i, j]));
}
Critical section
Number [i] =0;
Remainder section
      } while (1);
```

➢ Dekker's algorithm:

- The first known correct software solution to the critical-section problem for two processes was developed by Dekker.
- The two processes, p0 and p1, share the following variable:

```
      Boolean flag [2];     /*initially false*/
            Int turn;
      Dekker's algorithm:
      Do {
                        Flag[i] =true;
                        While (flag[i])
      If (turn==j)
{
      Flag[i] =false;
      While (turn==j);        //do nothing
}
      Critical section
      Flag[i] =false;
      Turn=j;  Remainder
      section
} while (1);
```

➢ Peterson has given more elegant algorithm

```
Do{
                Flag[i] =true;
      Turn=j;
      While (flag[j] and turn=j);
      Critical section
      Flag[i] =false;
      Remainder section
} while (1);
```

Meets all three requirements; solves the critical-section problem for two processes.The processes share two variables:

```
Boolean flag[2];
Int turn;
Initially flag [0] =flag [1] =false, and the value of turn is immaterial (but
is either 0 or 1).
```

To enter the critical section, process pi first sets flag [i], to be true and then sets turn to the value j, thereby asserting that if the other process wishes to enter the critical section it can do so. If both processes try to enter at the same time, turn will be set to both I and j at roughly the same time. Only one of these assignments will last; the other will occur, but will be overwritten immediately. The eventual value of turn decides which of the two processes is allowed to enter its critical section first.

We now prove that this solution is correct. We to show that:

1. Mutual exclusion is preserved.
2. The progress requirement is satisfied.
3. The bound-waiting requirement is met.

1. To prove property 1, we note that each pi enters its critical section only if either flag[j]==false or turn==j. also note that, if both processes can be executing in their critical sections at the same time, then flag[0]==flag[1]=true.
2. These two observations imply that p0 and p1 could not have successfully executed their while statements at about the same time, since the values of turn can be either 0 or 1, but cannot be both. Hence, one the of the processes any pj must have successfully executed the while statement, whereas pi had to execute at least one additional statement (‖turn==j‖). however, since, at that time, flag[j]==true, and turn==j, and this condition will persist as long as pj is in its critical section, the result follows: mutual exclusion is preserved.
3. To prove properties 2 and 3, we note that process pi can prevented from entering the critical section only if it is struck in the while loop with the condition flag[j]==true and
   turn==j this loop is the only one.

If pj is not ready to enter the critical section, then flag[j]==false and pi can enter itsa critical section.

If pj has set flag[j] to true and is also executing in its while statement, then either turn==I or turn==j. if turn==I, then p1 will enter the critical section.

If turns==j then pj will enter the critical section.

However, once pj exist its critical section, it will reset flag[j] to false, allowing pi to enter its critical section.

If pj , resets flag[J] to true, it must also set turn to i. thus, since pi does not change the value of the variable turn while statement, pi will enter the critical section(progress) after at most one entry by pj, (bound waiting).

CHAPTER 8: DEADLOCK

➢ CONCURRENCY AND DEADLOCK

In a multiprogramming environment, several processes may compete for a finite number of resources. **A** process requests resources; if the resources are not available at that time, the process enters a wait state. Waiting processes may never again change state, because the resources they have requested are held by other waiting processes. This situation is called a **deadlock.**

➢ Deadlock Characterization

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

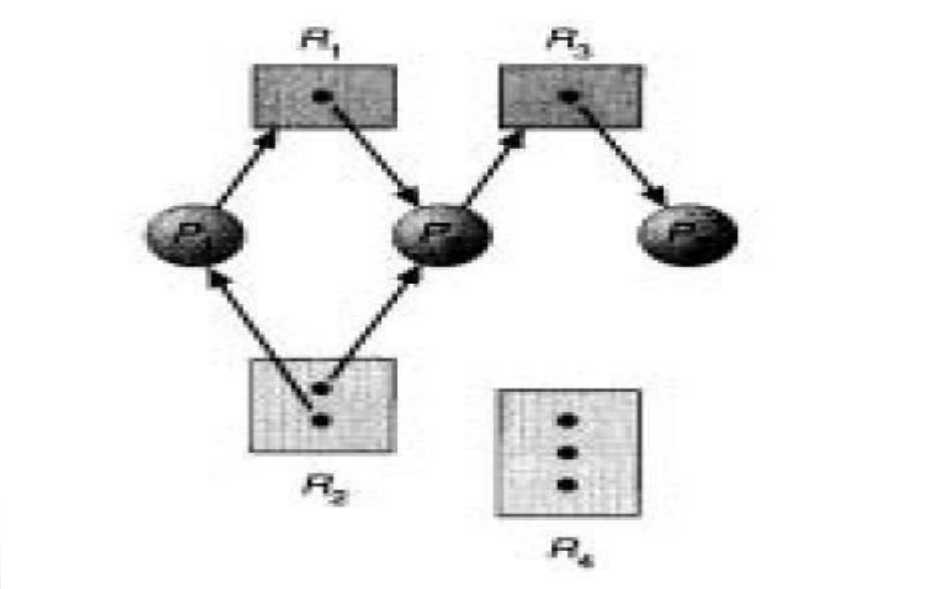
~~Necessary Conditions~~ A deadlock situation can arise if the following four conditions hold simultaneously in a system:

1. **Mutual exclusion**: At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.
2. **Hold and wait**: A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
3. **No preemption**: Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

4. **Circular wait**: A set {Po, PI, ..., P,) of waiting processes must exist such that Po is waiting for a resource that is held by PI, PI is waiting for a resource that is held by P2, ..., Pn .Pl is waiting for a resource that is held by P,, and P, is waiting for a resource that is held by Po.

1. Resource-Allocation Graph

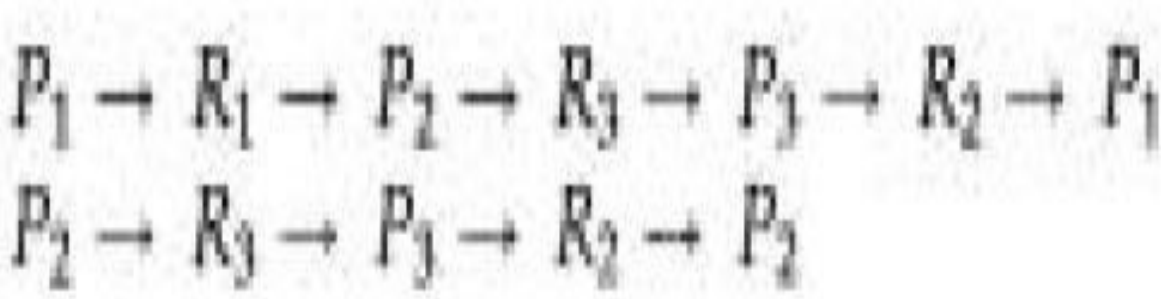
- Deadlocks can be described more precisely in terms of a directed graph called a system resource-allocation graph.
- This graph consists of a set of vertices V and a set of edges E. The set of vertices V is partitioned into two different types of nodes P = {PI, P2, ..., Pn}, the set consisting of all the active processes in the system, and R = {R1, R2, ..., Rm), the set consisting of all resource types in the system.
- A directed edge from process Pi to resource type *Rj* is denoted by Pi -+ Rj; it signifies that process ***Pi*** requested an instance of resource type Ri and is currently waiting for that resource.
- **A** directed edge from resource type Ri to process Pi is denoted by Rj -+ Pi; it signifies that an instance of resource type
- Ri has been allocated to process Pi. **A** directed edge Pi -+ Rj is called a request edge; a directed edge Rj + Pi is called an assignment edge. Pictorially, we represent each process Pi as a circle, and each resource type Ri as a square.
- Since resource type Ri may have more than one instance, we represent each such instance as a dot within the square.
- Note that a request edge points to only the square Xi, whereas an assignment edge must also designate one of the dots in the square.
- When process Pi requests an instance of resource type Rj, a request edge is inserted in the resource-allocation graph.
- When this request can be fulfilled, the request edge is *instantaneously* transformed to an assignment edge.
- When the process no longer needs access to the resource it releases the resource, and as a result the assignment edge is deleted.
- The resource-allocation graph shown in Figure 8.1 depicts the following situation.

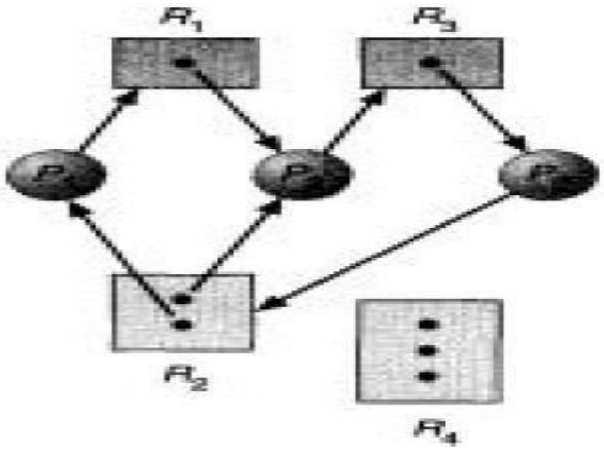


Process states:

1. Process PI is holding an instance of resource type R2, and is waiting for an instance of resource type R1.
2. Process P2 is holding an instance of R1 and R2, and is waiting for an instance of resource type R3.
3. Process P3 is holding an instance of R3.

- Given the definition of a resource-allocation graph, it can be shown that, if the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.
- If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred.
- If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred. Each process involved in the cycle is deadlocked.
- In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock.
- If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred.
- In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.
- To illustrate this concept, let us return to the resource-allocation graph depicted in Figure 8.1. Suppose that process *P3* requests an instance of resource type *R2.* Since no resource instance is currently available, a request edge *P3* -+ ***R2*** is added to the graph (Figure 8.2). At this point, two minimal cycles exist in the system:





Resource-allocation graph with a deadlock.

- Processes PI, P2, and P3 are deadlocked. Process P2 is waiting for the resource *R3,* which is held by process P3.
- Process P3, on the other hand, is waiting for either process PI or process P2 to release resource R2.
- In addition, process PI is waiting for process P2 to release resource R1. Now consider the resource-allocation graph in Figure 8.3. In this example, we also have a cycle

$$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$

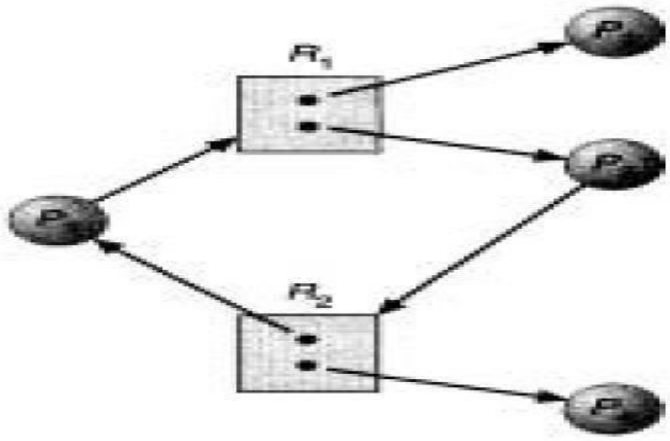
However, there is no deadlock.

- Observe that process **P4** may release its instance of resource type R2. That resource can then be allocated to P3, breaking the cycle.
- In summary, if a resource-allocation graph does not have a cycle, then the system is not in a deadlock state. On the other hand, if there is a cycle, then the system may or may not be in a deadlock state. This observation is important when we deal with the deadlock problem.

➢ Methods for Handling Deadlocks

Principally, we can deal with the deadlock problem in one of three ways:

1. We can use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlock state.
2. We can allow the system to enter a deadlock state, detect it, and recover.



3. We can ignore the problem altogether, and pretend that deadlocks never occur in the system. This solution is used by most operating systems, including UNIX.

➢ Deadlock Prevention

For a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock.

1. Mutual Exclusion

- ✓ The mutual-exclusion condition must hold for non sharable resources. For example, a printer cannot be simultaneously shared by several processes. Sharable resources, on the other hand, do not require mutually exclusive access, and thus cannot be involved in a deadlock.
- ✓ Read-only files are a good example of a sharable resource.
- ✓ If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file.
- ✓ A process never needs to wait for a sharable resource. In general, however, we cannot prevent deadlocks by denying the mutual-exclusion condition: Some resources
- ✓ are intrinsically non sharable.

2. Hold and Wait

- ✓ To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources.

- ✓ One protocol that can be used requires each process to request and be allocated all its resources before it begins execution.

- ✓ We can implement this provision by requiring that system calls requesting resources for a process precede all other system calls.
- ✓ An alternative protocol allows a process to request resources only when the process has none.
- ✓ A process may request some resources and use them. Before it can request any additional resources, however, it must release all the resources that it is currently allocated.
- ✓ These protocols have two main disadvantages. First, resource utilization may be low, since many of the resources may be allocated but unused for a long period.
- ✓ Second, starvation is possible. A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

3. No Preemption

- ✓ The third necessary condition is that there be no preemption of resources that have already been allocated. To ensure that this condition does not hold, we can use the following protocol.
- ✓ If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources currently being held are preempted.
- ✓ In other words, these resources are implicitly released. The preempted resources are added to the list of resources for which the process is waiting.
- ✓ The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
- ✓ Alternatively, if a process requests some resources, we first check whether they are available. If they are, we allocate them.
- ✓ If they are not available, we check whether they are allocated to some other process that is waiting for additional resources.
- ✓ If so, we preempt the desired resources from the waiting process and allocate them to the requesting process. If the resources are not either available or held by a waiting process, the requesting process must wait.
- ✓ While it is waiting, some of its resources may be preempted, but only if another process requests them.
- ✓ A process can be restarted only when it is allocated the new resources it is requesting and recovers any resources that were pre-empted while it was waiting.

NOTE-

This protocol is often applied to resources whose state can be easily saved and restored later, such as CPU registers and memory space. It cannot generally be applied to such resources as printers and tape drives.

4. Circular Wait

- ✓ The fourth and final condition for deadlocks is the circular-wait condition. One way to ensure that this condition never holds is to impose a total ordering of all resource types, and to require that each process requests resources in an increasing order of enumeration.
- ✓ Let R = {XI, R2, ..., R,) be the set of resource types.
- ✓ We assign n to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering. Formally , we define a one-to-one function F: R + N, where N is the set of natural numbers.

➢ Deadlock Avoidance

- Deadlock-prevention algorithms, as discussed in Section 8.4, prevent deadlocks by restraining how requests can be made. The restraints ensure that at least one of the necessary conditions for deadlock cannot occur, and, hence, that deadlocks cannot hold. Possible side effects of preventing deadlocks by this method, however, are low device utilization and reduced system throughput.
- An alternative method for avoiding deadlocks is to require additional information about how resources are to be requested.
- The simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need.
- Given a prior information about the maximum number of resources of each type that may be requested for each process, it is possible to construct an algorithm that ensures that the system will never enter a deadlock state. This algorithm defines the **deadlock-avoidance** approach.
- A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular wait condition can never exist.
- The resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes

1. Safe State

- *A* state is *safe* if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a **safe sequence.**
- A sequence of processes <PI, P2, ..., P,> is a safe sequence for the current allocation state if, for each Pi, the resources that Pi can still request can be satisfied by the currently available resources plus the resources held by all the Pi, with j < *i.* In this situation, if the resources that process Pi needs are not immediately available, then Pi can wait until all Pi have finished.
- When they have finished, Pi can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate. When Pi terminates, Pi+l can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be unsafe.

- A safe state is not a deadlock state. Conversely, a deadlock state is an unsafe state. Not all unsafe states are deadlocks.
- An unsafe state may lead to a deadlock. As long as the state is safe, the operating system can avoid unsafe (and deadlock) states. In an unsafe state, the operating system cannot prevent processes from requesting resources such that a deadlock occurs: The behaviour of the processes controls unsafe states.
- We consider a system with 12 magnetic tape drives and **3** processes: Po, PI, and P2. Process Po requires 10 tape drives, process PI may need as many as 4, and process P2 may need up to 9 tape drives. Suppose that, at time to, process Po is holding 5 tape drives, process PI is holding 2, and process PZ is holding 2 tape drives. (Thus, there are **3** free tape drives.)

	Maximum Needs	Current Needs
P_0	10	5
P_1	4	2
P_2	9	2

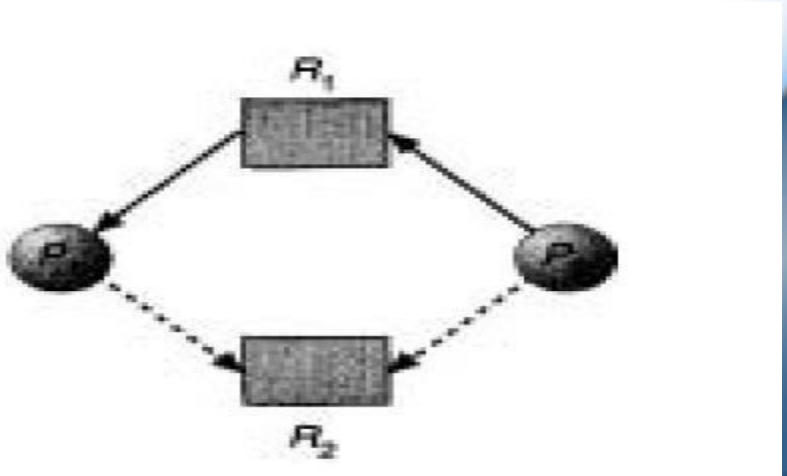
At time to, the system is in a safe state.

- The sequence <PI, PO, P2> satisfies the safety condition, since process PI can immediately be allocated all its tape drives and then return them (the system will then have 5 available tape drives), then process Po can get all its tape drives and return them (the system will then have 10 available tape drives),
- and finally process P2 could get all its tape drives and return them (the system will then have all 12 tape drives available).
- A system may go from a safe state to an unsafe state. Suppose that, at time tl, process Pp requests and is allocated 1 more tape drive. The system is no longer in a safe state. At this point, only process PI can be allocated all its tape drives.
- When it returns them, the system will have only 4 available tape drives. Since process Po is allocated 5 tape drives, but has a maximum of 10, it may then request 5 more tape drives. Since they are unavailable, process Po must wait.
- Similarly, process P2 may request an additional 6 tape drives and have to wait, resulting in a deadlock.
- Our mistake was in granting the request from process P2 for 1 more tape drive. If we had made P2 wait until either of the other processes had finished and released its resources, then we could have avoided the deadlock.
- Given the concept of a safe state, we can define avoidance algorithms that ensure that the system will never deadlock. The idea is simply to ensure that the system will always remain in a safe state.
- Initially, the system is in a safe state. Whenever a process requests a resource that is currently available, the system must decide whether the resource can be allocated immediately or whether the process must wait. The request is granted only if the allocation leaves the system in a safe state.

- In this scheme, if a process requests a resource that is currently available, it may still have to wait. Thus, resource utilization may be lower than it would be without a deadlock- avoidance algorithm.

➢ Resource-Allocation Graph Algorithm

- In addition to the request and assignment edges, we introduce a new type of edge, called a **claim edge.**
- A claim edge Pi -+ Rj indicates that process Pi may request resource Rj at some time in the future.
- This edge resembles a request edge in direction, but is represented by a dashed line. When process Pi requests resource Rj, the claim edge Pi -+ Rj is converted to a request edge.
- Similarly, when a resource Rj is released by ***Pi,***th e assignment edge Rj -+ Pi is reconverted to a claim edge Pi -+ Xi. We note that the resources must be claimed a prior in the system. That is, before process Pi starts executing, all its claim edges must already appear in the resource-allocation graph.
- We can relax this condition by allowing a claim edge Pi + Rj to be added to the graph only if all the edges associated with process Pi are claim edges.
- Suppose that process Pi requests resource Rj. The request can be granted only if converting the request edge Pi -+ Rj to an assignment edge *Rj* + Pi does not result in the formation of a cycle in the resource-allocation graph.
- Note that we check for safety by using a cycle-detection algorithm. An algorithm for detecting a cycle in this graph requires an order of n2 operations, where n is the number of processes in the system.

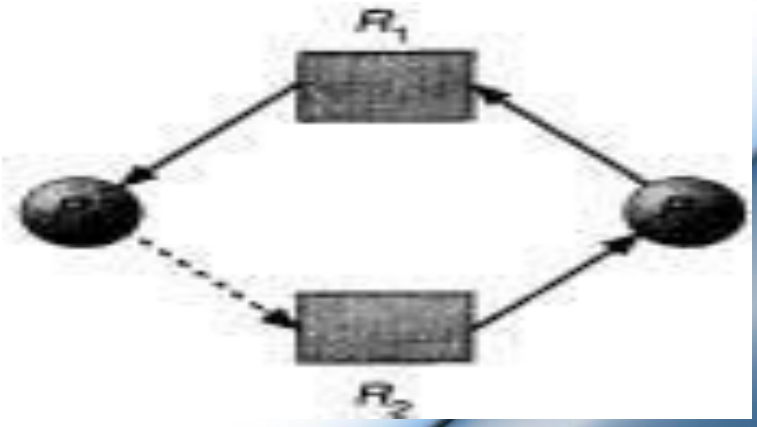


- If no cycle exists, then the allocation of the resource will leave the system in a safe state. If a cycle is found, then the allocation will put the system in an unsafe state. Therefore, process *Pi* will have to wait for its requests to be satisfied.

- To illustrate this algorithm, we consider the resource-allocation graph. Suppose that P2 requests R2. Although R2 is currently free, we cannot allocate it to P2, since this action will create a cycle in the graph . A cycle indicates that the system is in an unsafe state. If PI requests R2, and P2 requests R1, then a deadlock will occur.

➢ Banker's Algorithm

- ✓ The resource-allocation graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type.
- ✓ The deadlock-avoidance algorithm that we describe next is applicable to such a system, but is less efficient than the resource-allocation graph scheme. This algorithm is commonly known as the banker's algorithm.
- ✓ The name was chosen because this algorithm could be used in a banking system to ensure that the bank never allocates its available cash such that it can no longer satisfy the needs of all its customers.
- ✓ When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system.
- ✓ When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state.
- ✓ If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.
- ✓ Several data structures must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system.
- ✓ Let n be the number of processes in the system and m be the number of resource types.



We need the following data structures:

1. **Available: A** vector of length ***m*** indicates the number of available resources of each type. If *Available[j]* = k, there are k instances of resource type *Rj* available.

2. **Max:** *An n x* ***rn*** matrix defines the maximum demand of each process. If *Max[i,j]* = k, then process *Pi* may request at most k instances of resource type ***Ri***.
3. **Allocation:** An *n* x ***rn*** matrix defines the number of resources of each type currently allocated to each process. If *Allocation[i,j]* = k, then process ***Pi*** is currently allocated k instances of resource type *Rj.*

Need: *An n* x ***rn*** matrix indicates the remaining resource need of each process. **`If Need[i,j] = k`**, then process ***Pi*** may need k more instances of resource type *Ri* to complete its task. Note that **`Need[i,j] = Max[i,j] - Allocation[i,j].`**

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

```
Work := Available and Finish[i] :=false for i = 1,2, ..., n.
```

2. Find an *i* such that both

```
a. Finish [i] =false
b. Need i[5]= Work.
```

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

```
1. Work := Work + Allocation[i]
   Finish[i] := true
   go to step 2.

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

This algorithm may require an order of ***m*** x *n2* operations to decide whether a state is safe.

➢ Deadlock Detection

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur. In this environment, the system must provide:

1. An algorithm that examines the state of the system to determine whether a deadlock has occurred.
2. An algorithm to recover from the deadlock.

In the following discussion, we elaborate on these two requirements as they pertain to systems with only a single instance of each resource type, as well as to systems with several instances of each resource type. At this point, however, let us note that a detection-and-recovery scheme requires overhead that includes not only the run-time costs of maintaining the necessary information and executing the detection algorithm, but also the potential losses inherent in recovering from a deadlock.

1. Single Instance of Each Resource Type

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

2. Several Instances of a Resource Type

The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type.

3. Detection-Algorithm Usage

When should we invoke the detection algorithm? The answer depends on two factors:

1. How often is a deadlock likely to occur?
2. How many processes will be affected by deadlock when it happens?

If deadlocks occur frequently, then the detection algorithm should be invoked frequently. Resources allocated to deadlocked processes will be idle until the deadlock can be broken. In addition, the number of processes involved in the deadlock cycle may grow.

Deadlocks occur only when some process makes a request that cannot be granted immediately. This request may be the final request that completes a chain of waiting processes. In the extreme, we could invoke the deadlock detection algorithm every time a request for allocation cannot be granted immediately. In this case, we can identify not only the set of processes that is deadlocked, but also the specific process that "caused" the deadlock.

If there are many different resource types, one request may cause many cycles in the resource graph, each cycle completed by the most recent request and "caused" by the one identifiable process.

➢ Recovery from Deadlock

- When a detection algorithm determines that a deadlock exists, several alternatives exist. One possibility is to inform the operator that a deadlock has occurred, and to let the operator deal with the deadlock manually.
- The other possibility is to let the system recover from the deadlock automatically. There are two options for breaking a deadlock. One solution is simply to abort one or more processes to break the circular wait. The second option is to pre-empt some resources from one or more of the deadlocked processes

1. Process Termination

To eliminate deadlocks by aborting a process, we use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes.

a. **Abort all deadlocked processes:** This method clearly will break the deadlock cycle, but at a great expense; these processes may have computed for a long time, and the results of these partial computations must be discarded and probably recomputed later.
b. **Abort one process at a time until the deadlock cycle is eliminated:** This method incurs considerable overhead, since, after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

- Aborting a process may not be easy. If the process was in the midst of updating a file, terminating it will leave that file in an incorrect state. Similarly, if the process was in the midst of printing data on the printer, the system must reset the printer to a correct state before printing the next job.
- If the partial termination method is used, then, given a set of deadlocked processes, we must determine which process (or processes) should be terminated in an attempt to break the deadlock. This determination is a policy decision, similar to CPU-scheduling problems.

2. Resource Preemption

To eliminate deadlocks using resource preemption, we successively pre-empt some resources from processes and give these resources to other processes until the deadlock cycle is broken. If preemption is required to deal with deadlocks, then three issues need to be addressed:

a. **Selecting a victim:** Which resources and which processes are to be preempted? As in process termination, we must determine the order of pre-emption to minimize cost. Cost factors may include such parameters as the number of resources a deadlock process is holding, and the amount of time a deadlocked process has thus far consumed during its execution.
b. **Rollback:** We must roll back the process to some safe state, and restart it from that state. Since, in general, it is difficult to determine what a safe state is, the simplest solution is a total rollback: Abort the process and then restart it. However, it is more effective to roll back the process only as far as necessary to break the deadlock. On the other hand, this method requires the system to keep more information about the state of all the running processes.
c. **Starvation:** In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim. As a result, this process never completes its designated task, a starvation situation that needs to be dealt with in any practical system. Clearly, we must ensure that a process can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.

CHAPTER 9: MEMORY MANAGEMENT

➢ Memory Management:

Address Binding:

Usually, a program resides on a disk as a binary executable file. The program must be brought into memory and placed within a process for it to be executed. Depending on the memory management in use, the process may be moved between disk and memory during its execution. The collection of processes on the disk that is waiting to be brought into memory for execution forms the input queue.

Different address binding are as follows

- **Compile time:** If you know at compile time where the process will reside in memory, then absolute code can be generated. For example, if you know a priori that a user process resides starting at location R, then the generated compiler code will start at that location and extend up from there.

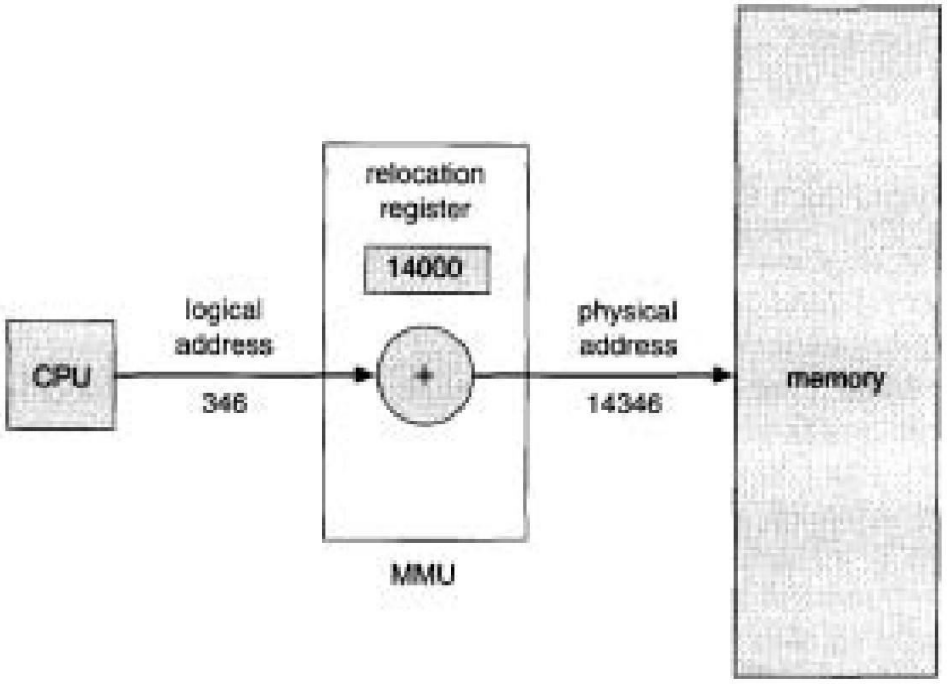
- **Load time:** If it is not known at compile time where the process will reside in memory, then the compiler must generate relocatable code. In this case, final binding is delayed until load time. If the starting address changes, we need only to reload the user code to incorporate this changed value.

- **Execution time:** If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time.

➢ Logical- Versus Physical-Address Space:

- An address generated by the CPU is commonly referred to as a **logical address,** whereas an address seen by the memory unit—that is, the one loaded into the **memory-address register** of the memory—is commonly referred to as a **physical address.**
- The compile-time and load-time address-binding methods generate identical logical and physical addresses. However, the execution-time address- binding scheme results in differing logical and physical addresses. In this case, we usually refer to the logical address as a **virtual address.**
- We use logical address and virtual address interchangeably in this text. The set of all logical addresses generated by a program is a **logical-addressspace;** the set of all physical addresses corresponding to these logical addresses is a **physical-address space.**
- Thus, in the execution-time address-binding scheme, the logical- and physical-address spaces differ.

- The run-time mapping from virtual to physical addresses is done by a hardware device called the **memory-management unit (MMU).**
- The value in the relocation register is added to every address generated by a user process at the time it is sent to memory.
- For example, if the base is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location 14000; an access to location 346 is mapped to location 14346..
- The user program deals with logical addresses. The memory-mapping hardware converts logical addresses into physical addresses.



➢ Dynamic Loading:

- To obtain better memory-space utilization, we can use dynamic loading. With dynamic loading, a routine is not loaded until it is called. All routines are kept on disk in a relocatable load format.
- The main program is loaded into memory and is executed. When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded.
- If not, the relocatable linking loader is called to load the desired routine into memory and to update the program's address tables to reflect this change. Then, control is passed to the newly loaded routine.
- The advantage of dynamic loading is that an unused routine is never loaded. This method is particularly useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines.

- Dynamic loading does not require special support from the operating system. It is the responsibility of the users to design their programs to take advantage of such a method.

Dynamic Linking and Shared Libraries:

- The concept of dynamic linking is similar to that of dynamic loading. Rather than loading being postponed until execution time, linking is postponed. This feature is usually used with system libraries, such as language subroutine libraries.
- With dynamic linking, a stub is included in the image for each library-routine reference. This stub is a small piece of code that indicates how to locate the appropriate memory- resident library routine, or how to load the library if the routine is not already present.
- When this stub is executed, it checks to see whether the needed routine is already in memory. If not, the program loads the routine into memory.
- Unlike dynamic loading, dynamic linking generally requires help from the operating system.
- If the processes in memory are protected from one another (Section 9.3), then the operating system is the only entity that can check to see whether the needed routine is in another process' memory space, or that can allow multiple processes to access the same memory addresses.

➢ Overlays:

- The size of the process is limited to the size of memory allocated to it, a technique called Overlays is sometimes use.The idea of overlays is to keep in memory only those instructions and data that are needed at any given time.
- When other instructions are needed, they are loaded into space occupied previously by instructions that are no longer needed.
- As an example, consider a two-pass assembler. During pass 1, it constructs a symbol table; then, during pass 2, it generates machine-language code.
- We may be able to partition such an assembler into pass 1 code, pass 2 code, the symbol table, and common support routines used by both pass 1 and pass 2.

For E.g.

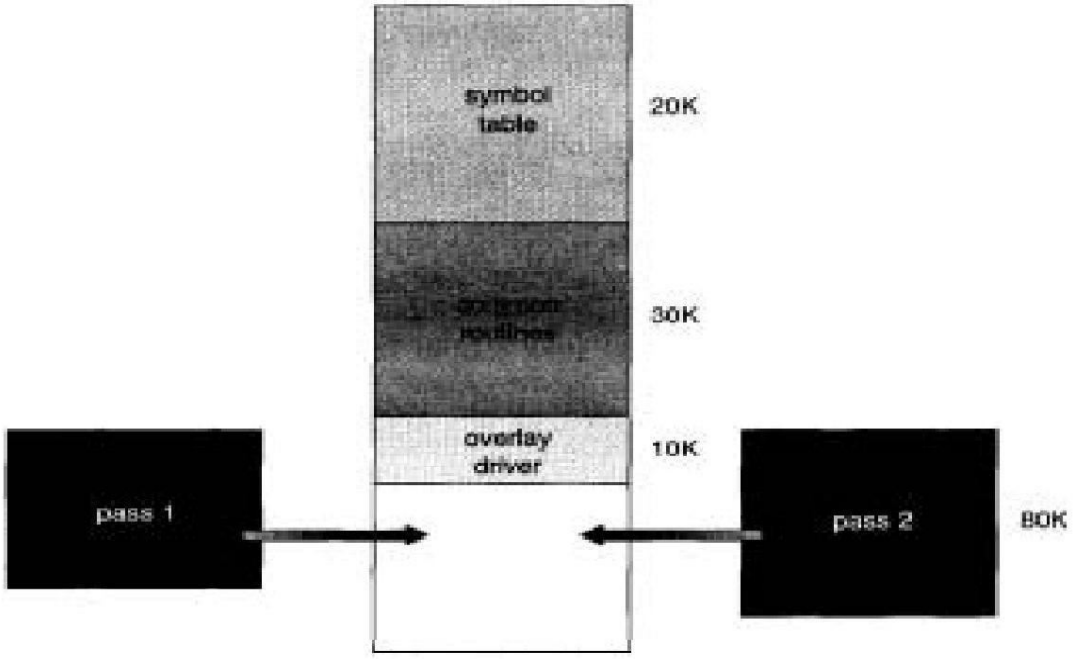
- Pass1 70k
- Pass2 80k
- Symbol table 20k
- Common routine 30k

- To load everything at once, we would require 200 KB of memory. If only 150 KB is available, we cannot run our process.

- However, notice that pass 1 and pass 2 do not need to be in memory at the same time. We thus define two overlays:

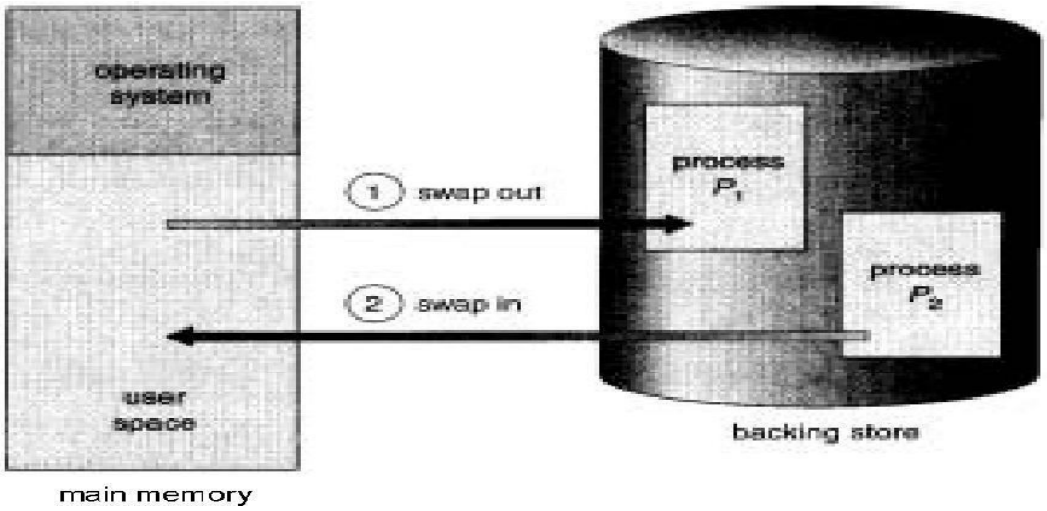
- Overlay A is the symbol table, common routines, and pass 1, and
- overlay B is the symbol table, common routines, and pass 2.

- We add an overlay driver (10 KB) and start with overlay *A* in memory. When we finish pass 1, we jump to the overlay driver, which reads overlay B into memory, overwriting overlay A, and then transfers control to pass
- As in dynamic loading, overlays do not require any special support from the operating system. They can be implemented by the user completely.



➢ Swapping

- A process needs to be in memory to be executed. A process, however, can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution.
- If a higher-priority process arrives and wants service, the memory manager can swap out the lower-priority process so that it can load and execute the higher-priority process.
- When the higher-priority process finishes, the lower-priority process can be swapped back in and continued. This variant cf swapping is sometimes called **roll out, roll in.**.Swapping requires a backing store.
- The backing store is commonly a fast disk. It must be large enough to accommodate copies of all memory images for all users, and it must provide direct access to these memory images. The system maintains a **ready queue** consisting of all processes whose memory images are on the backing store or in memory and are ready to run.

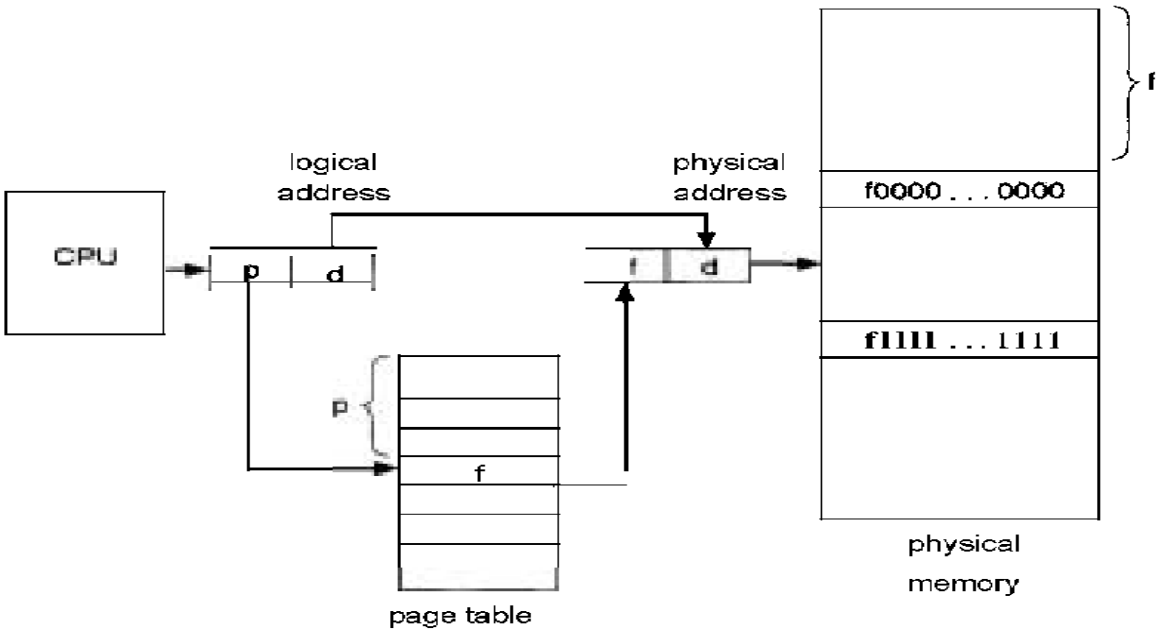


➢ Paging

- Another possible solution of the external fragmentation problem is to permit the logical address space of a process to be noncontiguous.
- One avoids the considerable problem of fitting the varying-sized memory chunks onto the backing store, from which most of the previous memory-management schemes suffered.

➢ Basic Method

- Physical memory is broken into fixed-sized blocks called frames. Logical memory is also broken into blocks cf the same size called pages.
- When a process is to be executed, its pages are loaded into any available memory frames.
- The backing store is divided into fixed-sized blocks that are of the same size as the memory frames.
- Every address generated by the CPU is divided into two parts: a 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 page in physical memory.
- This base address is combined with the page offset to define the physical memory address that is sent to the memory unit.



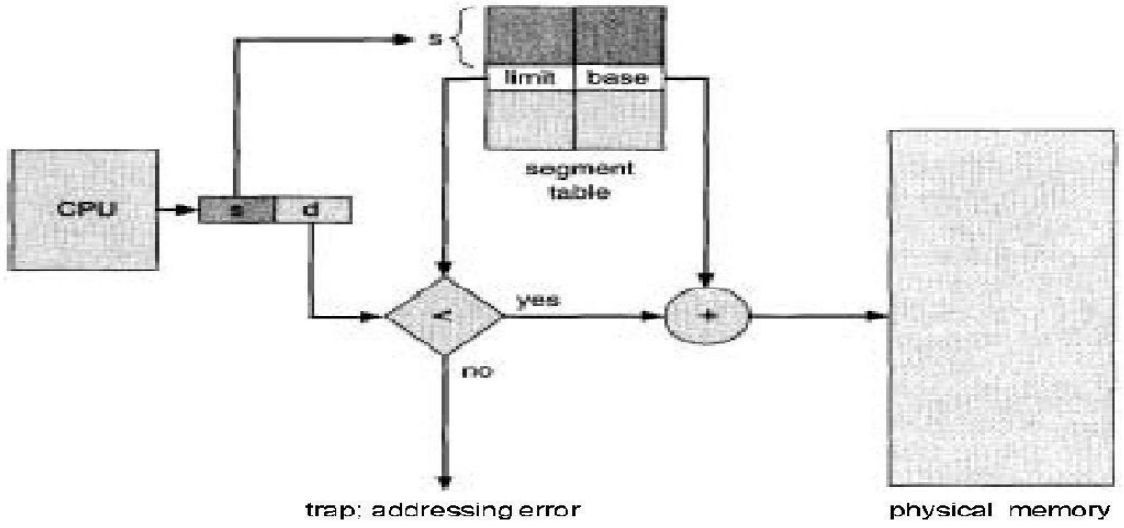
➢ Segmentation

- An important aspect of memory-management scheme that became unavoidable with paging is the separation of the user view of memory and the actual physical memory.
- The user view of memory is not the same as the actual physical memory. The user's view is mapped on to physical memory.
- The mapping allows differentiation between logical memory and physical memory.

➢ Basic Method

- Segmentation is a memory-management scheme that supports this user view of memory.
- A logical-address space is a collection of segments. Each segment has a name and a length. The addresses specify both the segment name and the offset within the segment.
- The user therefore specifies each address by two quantities: a segment name and an offset. For simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by a segment name. Thus, a logical address consists of a *two* tuple:

  <segment-number, offset>



➢ **Hardware Support**

- Although the user can now refer to objects in the program by a two-dimensional address, the actual physical memory is still, of course, a one-dimensional sequence of bytes.
- Thus, we must define an implementation to map two-dimensional user-defined addresses into one-dimensional physical addresses.
- This mapping is affected by a segment table. Each entry of the segment table has a segment base and a segment limit.
- The segment base contains the starting physical address where the segment resides in memory, whereas the segment limit specifies the length of the segment.
- A logical address consists cf two parts: a segment number, *s,* and an offset into that segment, d. The segment number is used as an index into the segment table.
- The offset d of the logical address must be between **o** and the segment limit. If it is not, we trap to the operating system. If this offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte.

CHAPTER 10: VIRTUAL MEMORY

➢ Virtual memory

- Virtual memory is a technique that allows the execution of processes that may not be completely in memory.
- One major advantage of this scheme is that programs can be larger than physical memory.
- Further, virtual memory abstracts main memory into an extremely large, uniform array of storage, separating logical memory as viewed by the user from physical memory.

- This technique frees programmers from the concerns of memory-storage limitations. Virtual memory is the separation of user logical memory from physical memory. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available.

- Virtual memory makes the task of programming much easier, because the programmer no longer needs to worry about the amount of physical memory available. Virtual memory is commonly implemented by demand paging.

➢ Demand Paging

- A demand-paging system is similar to a paging system with swapping. Processes reside on secondary memory. When we want to execute a process, we swap it into memory.
- Rather than swapping the entire process into memory, however, we use a **lazy swapper.** A lazy swapper never swaps a page into memory unless that page will be needed.
- The use of word swapper is technically incorrect. A swapper manipulates entire processes, whereas a **pager** is concerned with the individual pages of a process.
- We will then use the term pager, rather than swapper, in connection with demand paging. When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again.
- Instead of swapping in a whole process, the pager brings only those necessary pages into memory. Thus, it avoids reading into memory pages that will not be used anyway, decreasing the swap time and the amount of physical memory needed.
- With this scheme, we need some form of hardware support to distinguish between those pages that are in memory and those pages that are on the disk. The valid-invalid bit scheme can be used for this purpose.
- when this bit is set to "valid," this value indicates that the associated page is both legal and in memory. If the bit is set to "invalid," this value indicates that the page either is not valid or is valid but is currently on the disk.

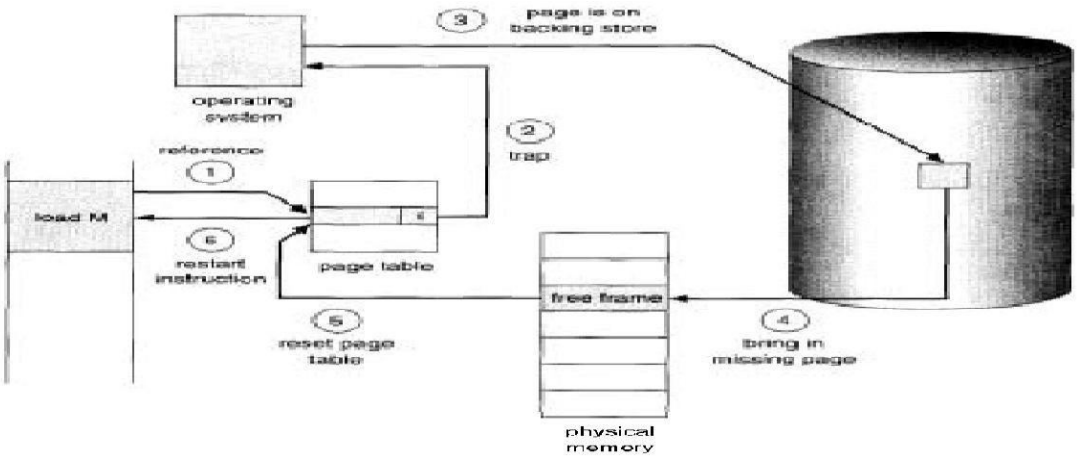
- **Page table:** This table has the ability to mark an entry invalid through a valid-invalidbit or special value of protection bits.

- **Secondary memory:** This memory holds those pages that are not present in main memory. The secondary memory is usually a high-speed disk. It is known as the swap device, and the section of disk used for this purpose is known as **swap space or backing store.**

➢ Steps in handling a page fault:

1. We check an internal table for this process, to determine whether the reference was a valid or invalid memory access.
2. If the reference was invalid, we terminate the process. If it was valid, but we have not yet brought in that page, we now page it in.

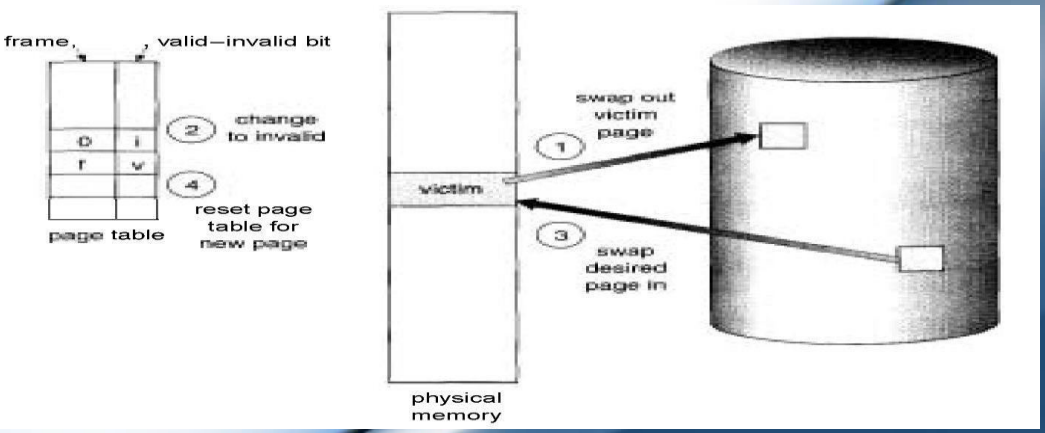
3. We find a free frame.
4. We schedule a disk operation to read the desired page into the newly allocated frame.
5. Then the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.
6. We restart the instruction that was interrupted by the illegal address trap. The process can now access the page as though it had always been in memory.



➢ Pure Demand paging

In the extreme case, we could start executing a process with no pages in memory. When the operating system sets the instruction pointer to the first instruction of the process, which is on a non-memory-resident page, the process immediately faults for the page. After this page is brought into memory, the process continues to execute, faulting as necessary until every page that it needs is in memory. At that point, it could execute in to memory until it is required.

➢ Page Replacement



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

2. Find a free frame:

    a. If there is a free frame, use it.
    b. If there is no free frame, use a page-replacement algorithm to select a victim frame.
    c. Write the victim page to the disk; change the page and frame tables accordingly.

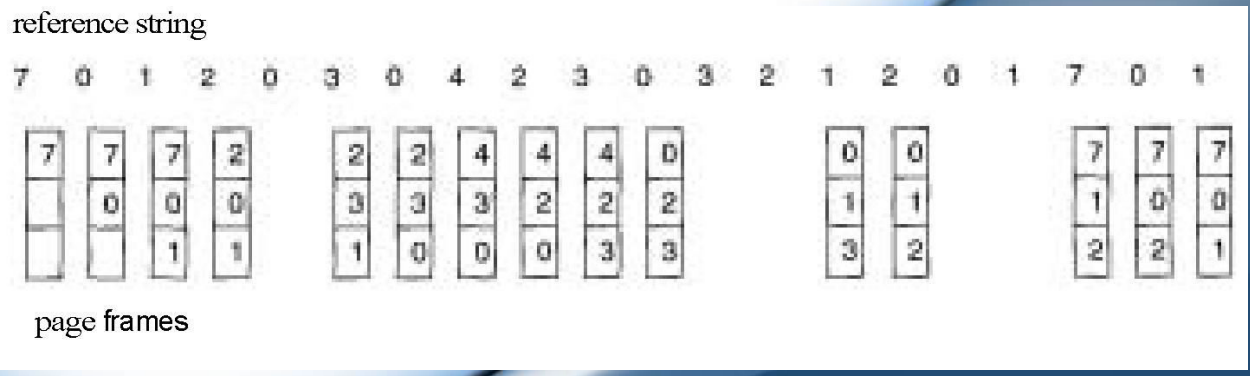
3. Read the desired page into the (newly) free frame; change the page and frame tables.
4. Restart the user process.

➢ Page Replacement Algorithms

There are many different page-replacement algorithms. Every operating system probably has its own replacement scheme. How do we select a particular replacement algorithm? In general, we want the one with the lowest page-fault rate. We evaluate an algorithm by running it on a particular string of memory references and computing the number of page faults. The string of memory references is called a reference string.

➢ FIFO Page Replacement

- The simplest page-replacement algorithm is a FIFO algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory.
- When a page must be replaced, the oldest page is chosen. Notice that it is not strictly necessary to record the time when a page is brought in. We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue.
- For our example reference string, our three frames are initially empty. The first three references (7, 0, 1) cause page faults, and are brought into these empty frames. The next reference (2) replaces page 7, because page 7 was brought in first. Since 0 is the next reference and 0 is already in memory, we have no fault for this reference.
- The first reference to 3 results in page 0 being replaced, since it was the first of the three pages in memory (**0**, **1**. and **2**) to be brought in. Because of this replacement, the next reference, to 0, will fault.

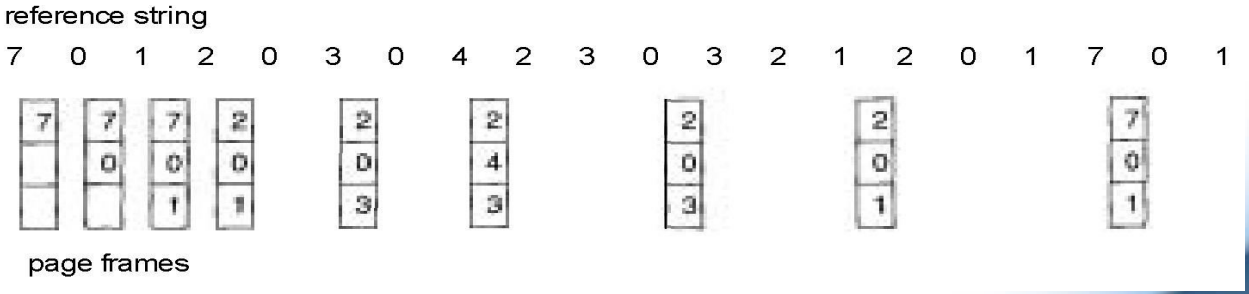


- Page 1 is then replaced by page 0. There are 15 faults altogether. The FIFO page-replacement algorithm is easy to understand and program.
- However, its performance is not always good. For the reference string 1,2,3,4,1,2,5,1,2,3,4,5 we notice that the number of faults for frame(10)is greater than the number of faults for three frames(9).
- This result is most unexpected and is known as Belady's anomaly.Belady's anomaly reflects the fact that, for some page-replacement algorithm.

➢ Optimal Page Replacement

- One result of the discovery of Belay's anomaly was the search for an **optimal page-replacement algorithm.** An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms.
- An optimal algorithm will never suffer from Belady's anomaly. An optimal page replacement algorithm exist, and has been called OPT or MIN.

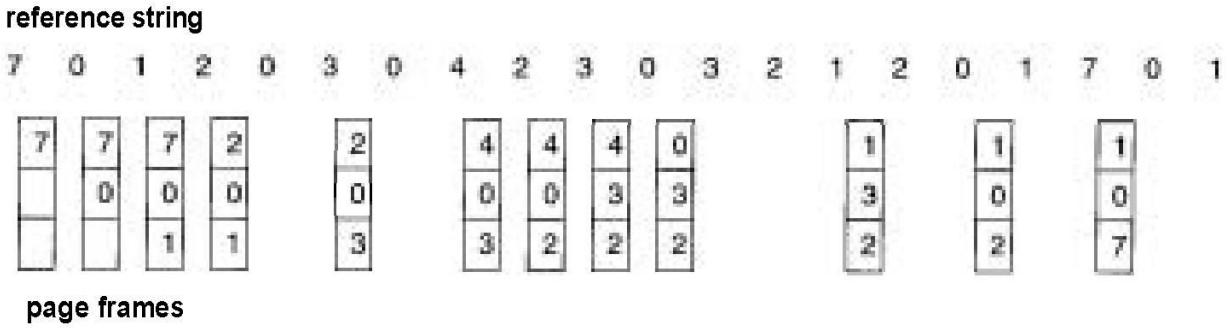
- It is simply replace the page that will not be used for the longest period of time. Use of this page-replacement algorithm guarantees the lowest possible page- fault rate for a fixed number of frames.



- For example, on our sample reference string, the optimal page-replacement algorithm would yield nine page faults. The first three references cause faults that fill the three empty frames.
- The reference to page 2 replaces page 7, because 7 will not be used until reference 18, whereas page will be used at 5, and page 1 at 14.
- The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again. With only nine page faults, optimal replacement is much better than a FIFO algorithm.
- Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string.

➢ LRU Page Replacement

- LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses that page that has not been used for the longest period of time.
- This strategy is the optimal page-replacement algorithm looking backward in time, rather than forward.



- The first five faults are the same as the optimal replacement. When the reference to page 4 occurs, however, LRU replacement sees that, of the three frames in memory, page 2 was used least recently.
- The most recently used page is page 0, and just before that page 3 was used. Thus, the LRU algorithm replaces page not knowing that page 2 is about to be used.

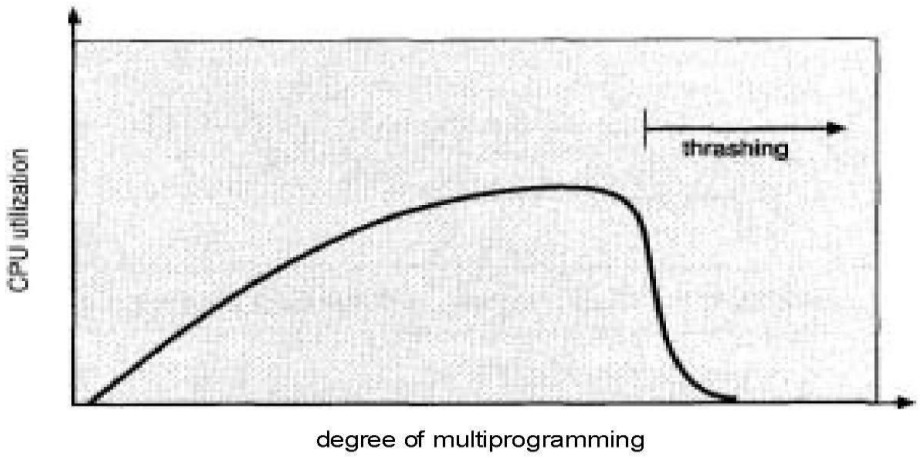
- When it then faults for page 2, the LRU algorithm replaces page 3 since, of the three pages in memory {0, 3, 4}, page 3 is the least recently used. Despite these problems, LRU replacement with 12 faults is still much better than FIFO replacement with 15.
- LRU page-replacement algorithm may require substantial hardware assistance.

➢ Thrashing

- In fact, look at any process that does not have "enough" frames. Although it is technically possible to reduce the number of allocated frames to the minimum, there is some (larger) number of pages in active use.
- If the process does not have this number of frames, it will quickly page fault. At this point, it must replace some page.
- However, since all its pages are in active use, it must replace a page that will be needed again right away.
- Consequently, it quickly faults again, and again, and again. The process continues to fault, replacing pages for which it then faults and brings back in right away.
- This high paging activity is called thrashing. A process is thrashing if it is spending more time paging than executing.

➢ Cause of Thrashing

- The operating system monitors CPU utilization. If CPU utilization is too low, we increase the degree of multiprogramming by introducing a new process to the system.
- A global page-replacement algorithm is used; it replaces pages with no regard to the process to which they belong. Now suppose that a process enters a new phase in its execution and needs more frames.
- It starts faulting and taking frames away from other processes. These processes need those pages, however, and so they also fault, taking frames from other processes.
- These faulting processes must use the paging device to swap pages in and out. As they queue up for the paging device, the ready queue empties. As processes wait for the paging device, CPU utilization decreases.



- The CPU scheduler sees the decreasing CPU utilization, and *increases* the degree of multiprogramming as a result. The new process tries to get started by taking frames from running processes, causing more page faults, and a longer queue for the paging device.
- As a result, CPU utilization drops even further, and the CPU scheduler tries to increase the degree of multiprogramming even more. Thrashing has occurred and system throughput plunges.
- The page- fault rate increases tremendously. As a result, the effective memory access time increases. No work is getting done, because the processes are spending all their time paging.

CHAPTER 11: FILE SYSTEM

➢ File-System Interface

The file system consists of two distinct parts: a collection of files, each storing related data, and a directory structure, which organizes and provides information about all the files in the system.

➢ File Concept

- The operating system abstracts from the physical properties of its storage devices to define a logical storage unit (the file).Files are mapped, by the operating system, onto physical devices. These storage devices are usually nonvolatile, so the contents are persistent through power failures and system reboots.
- From a user's perspective, a file is the smallest allotment of logical secondary storage; that is, data cannot be written to secondary storage unless they are within a file.
- Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic, alphanumeric, or binary.
- Files may be free form, such as text files, or may be formatted rigidly. In general, a file is a sequence of bits, bytes, lines, or records, the meaning of which is defined by the file's creator and user. The concept of a file is thus extremely general.
- A file has a certain defined structure according to its type. A *text* file is a sequence of characters organized into lines (and possibly pages). A *source* file is a sequence of subroutines and functions.

➢ File Attributes

- Name: The symbolic file name is the only information kept in human- readable form.
- Identifier: This unique tag, usually a number, identifies the file within the file system; it is the non-human-readable name for the file.
- Type: This information is needed for those systems that support different types.
- Location: This information is a pointer to a device and to the location of the file on that device.

- Size: The current size of the file (in bytes, words, or blocks), and possibly the maximum allowed size are included in this attribute.
- Protection: Access-control information determines who can do reading, writing, executing, and so on.
- Time, date, and user identification: This information may be kept for creation, last modification, and last use. These data can be useful for protection, security, and usage monitoring.

➢ File Operations

- **Creating a file:** Two steps are necessary to create a file. First, space in the file system must be found for the file. We shall discuss how to allocate space for the file in Chapter 12. Second, an entry for the new file must be made in the directory. The directory entry records the name of the file and the location in the file system.
- **Writing a file:** To write a file, we make a system call specifying both the name of the file and the information to be written to the file. Given the name of the file, the system searches the directory to find the location of the file. The system must keep a write

pointer to the location in the file where the next write is to take place. The write pointer must be updated whenever a write occurs.

- **Reading a file:** To read from a file, we use a system call that specifies the name of the file and where (in memory) the next block of the file should be put. Again, the directory is searched for the associated directory entry, and the system needs to keep a read pointer to the location in the file where the next read is to take place.
- **Repositioning within a file:** The directory is searched for the appropriate entry, and the current-file-position is set to a given value. Repositioning within a file does not need to involve any actual I/O.
- **Deleting a file:** To delete a file, we search the directory for the named file. Having found the associated directory entry, we release all file space, so that it can be reused by other files, and erase the directory entry.
- **Truncating a file:** The user may want to erase the contents of a file but keep its attributes. Rather than forcing the user to delete the file and then recreate it, this function allows all attributes to length zero.

➢ **File System Access Methods**

Files store information. When it is used, this information must be accessed and read into computer memory. The information in the file can be accessed. There are two to access a file.

➢ Sequential Access

- The simplest access method is sequential access. Information in the file is processed in order, one record after the other. This mode of access is by far the most common; for example, editors and compilers usually access files in this fashion.
- The bulk of the operations on a file is reads and writes. A read operation reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location. Similarly, a write appends to the end of the file and advances to the end of the newly written material (the new end of file).

➢ Direct Access

- Another method is direct access (or relative access). A file is made up of fixed- length logical records that allow programs to read and write records rapidly in no particular order. Direct—access file allows arbitrary blocks to be read or written.
- Thus, we may read block 14, then read block 53, and then write block 7. There are no restrictions on the order of reading or writing for a direct—access file.
- Direct—access files are of great use for immediate access to large amounts of information. Databases are often of this type.

➢ Other Access Methods

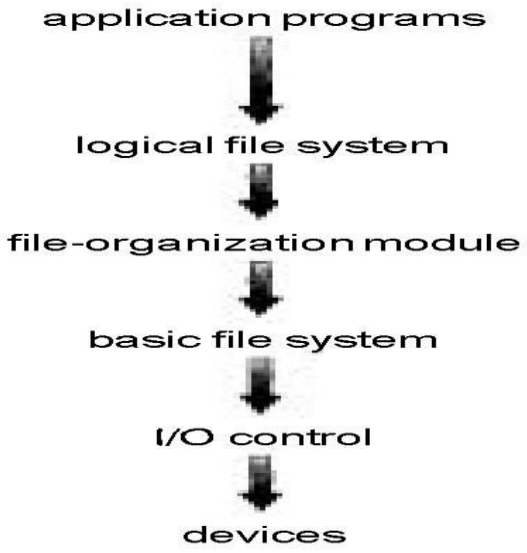
- Other access methods can be built on top cf a direct-access method. These methods generally involve the construction of an index for the file.
- The index, like an index in the back of a book, contains pointers to the various blocks. To find a record in the file, we first search the index, and then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

➢ File Type

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

CHAPTER 12: FILE- SYSTEM IMPLEMENTATION

- To provide an efficient and convenient access to the disk, the operating system imposes one or more **file systems** to allow the data to be stored, located, and retrieved easily.
- **A** file system poses two quite different design problems. Each level in the design uses the features of lower levels to create new features for use by higher levels.
- The lowest level, the *I/O* control, consists of **device drivers** and interrupts handlers to transfer information between the main memory and the disk system. A device driver can be thought of as a translator. Its input consists of high-level commands such as "retrieve block 123.‖ Its output consists of low- level, hardware-specific instructions.



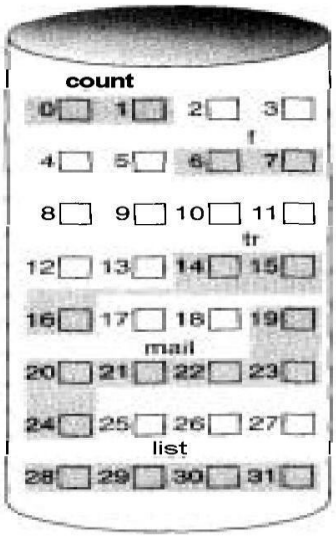
- The basic file system needs only to issue generic commands to the appropriate device driver to read and write physical blocks on the disk
- The file-organization module can translate logical block addresses to physical block addresses for the basic file system to transfer.
- Finally, the logical file system manages the directory structure to provide the file-organization module with the information. The logical file system is also responsible for protection and security.

➢ Allocation Methods

The main problem is how to allocate space to these files so that disk space is utilized effectively and files can be accessed quickly. Three major methods of allocating disk space are in wide use: contiguous, linked, and indexed. Each method has advantages and disadvantages.

➢ Contiguous Allocation

- The contiguous-allocation method requires each file to occupy a set of contiguous blocks on the disk. Disk addresses define a linear ordering on the disk. Thus, the number of disk seeks required for accessing contiguously allocated files is minimal.
- In Contiguous allocation consider if file is n blocks long, and start at location b, then it occupies blocks b,b+1,b+2,…..,b+n-1. The directory entry for each file indicates the address of the starting block and the length of the area allocated for this file.
- Accessing a file that has been allocated contiguously is easy. For sequential access, the file system remembers the disk address of the last block referenced and, when necessary, reads the next block.
- For direct access to block i of a file that starts at block b we can immediately access block b + i. Thus, both sequential and direct access can be supported by contiguous allocation. One difficulty with contiguous allocation is finding space for a new file.

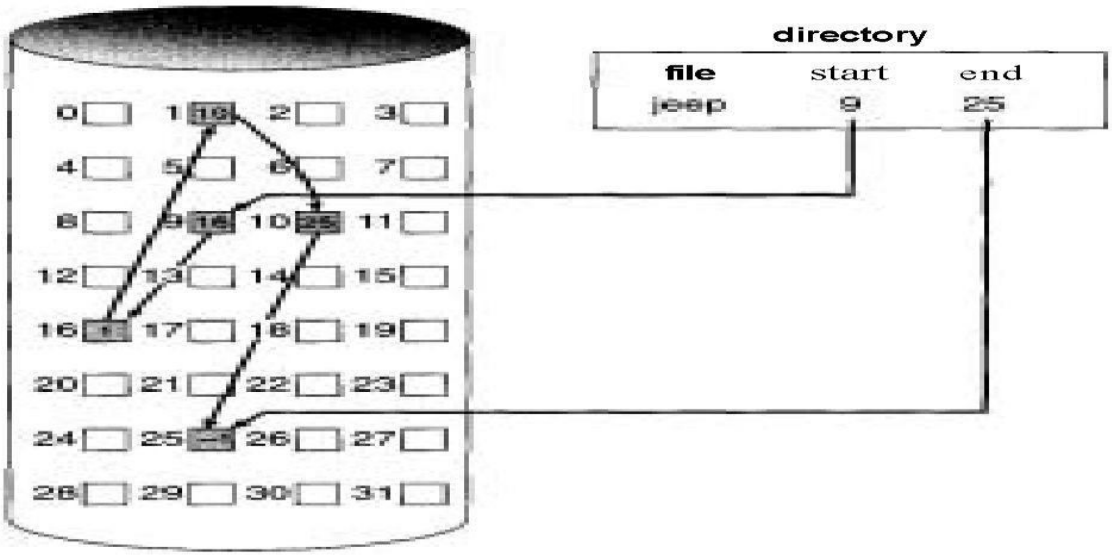


directory

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

➢ Linked Allocation

- **Linked allocation** solves all problems cf contiguous allocation. With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk.
- The directory contains a pointer to the first and last blocks of the file. For example, a file of five blocks might start at block continue at block 16, then block 1, block 10, and finally block 25. Each block contains a pointer to the next block.



Advantage: There is no external fragmentation with linked allocation, and any free block on the free-space list can be used to satisfy a request.

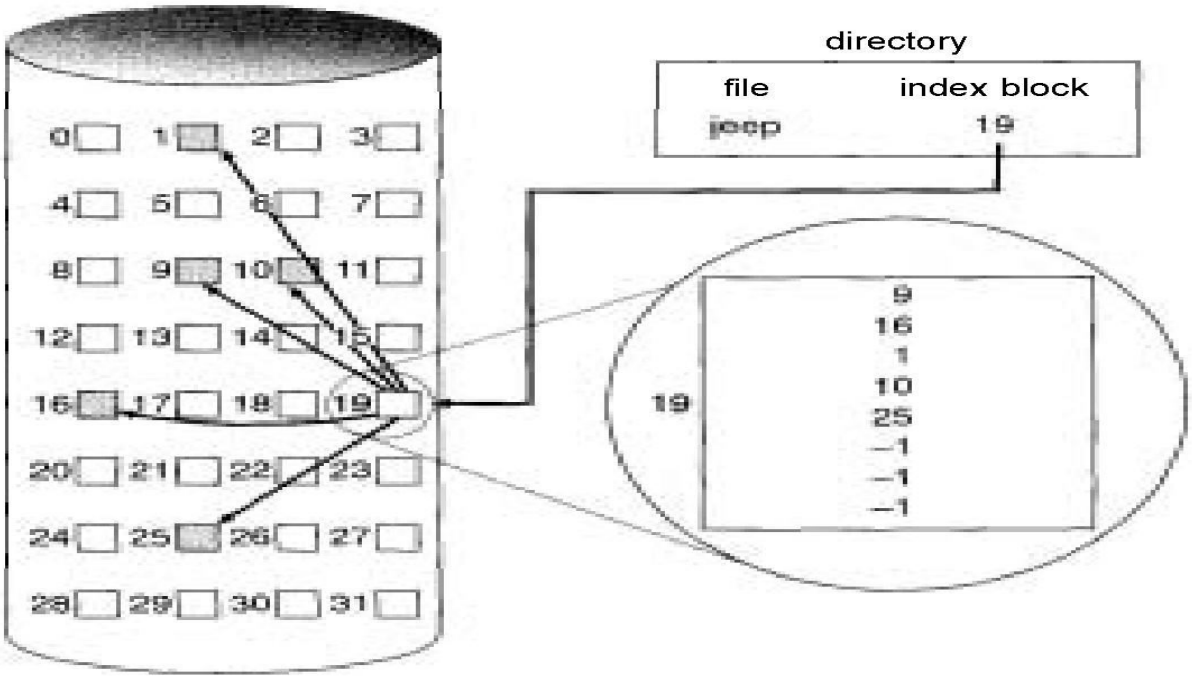
Disadvantage:

1. Linked allocation does have disadvantages, however. The major problem is that it can be used effectively only for sequential-access files. To find the ith block of a file, we must start at the beginning of that file, and follow the pointers until we get to the ith block. Each access to a pointer requires a disk read, and sometimes a disk seek. Consequently, it is inefficient to support a direct-access capability for linked files.
2. Another disadvantage to linked allocation is the space required for the pointers.
3. Yet another problem of linked allocation is reliability. Since the files are linked together by pointers scattered all over the disk, consider what would happen if a pointer were lost or damaged.

➢ Indexed Allocation

- Linked allocation solves the external-fragmentation and size-declaration problems of contiguous allocation. In linked allocation it all pointer together in to one location: the index block.
- Each file has its own index block, which is an array of disk-block addresses. The ith entry in the index block points to the ith block of the file. The directory contains the address of the index block.
- To read the ith block, use the pointer in the ith index-block entry to find and read the desired block. When the file is created, all pointers in the index block are set to nil. When the ith block is first written, a block is obtained from the free-space manager, and its address is put in the ith index-block entry.

- Indexed allocation does not suffer from wasted space. The pointer overhead of the index block is generally greater then the pointer overhead of linked allocation.



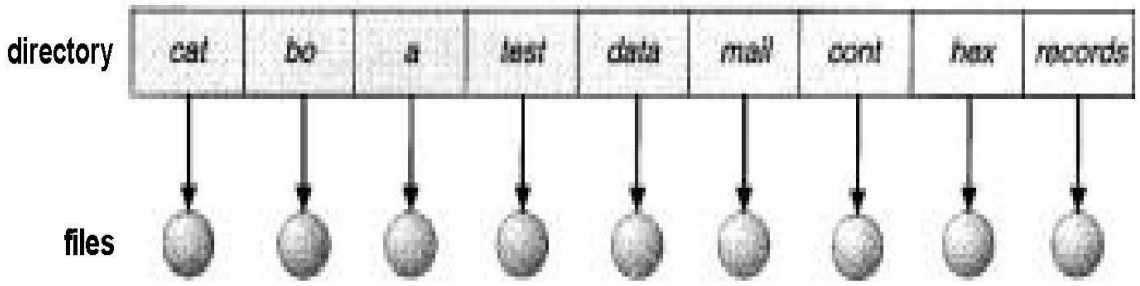
➢ Explain Directory Operation and Structure

➢ Directory Operation:

- **Search for a file:** We need to be able to search a directory structure to find the entry for a particular file. Since files have symbolic names and similar names may indicate a relationship between files, we may want to be able to find all files whose names match a particular pattern.
- **Create a file:** New files need to be created and added to the directory.
- **Delete a file:** When a file is no longer needed, we want to remove it from the directory.
- **List a directory:** We need to be able to list the files in a directory, and the contents of the directory entry for each file in the list.
- **Rename a file:** Because the name of a file represents its contents to its users, the name must be changeable when the contents or use of the file changes. Renaming a file may also allow its position within the directory structure to be changed.
- **Traverse the file system:** We may wish to access every directory, and every file within a directory structure.

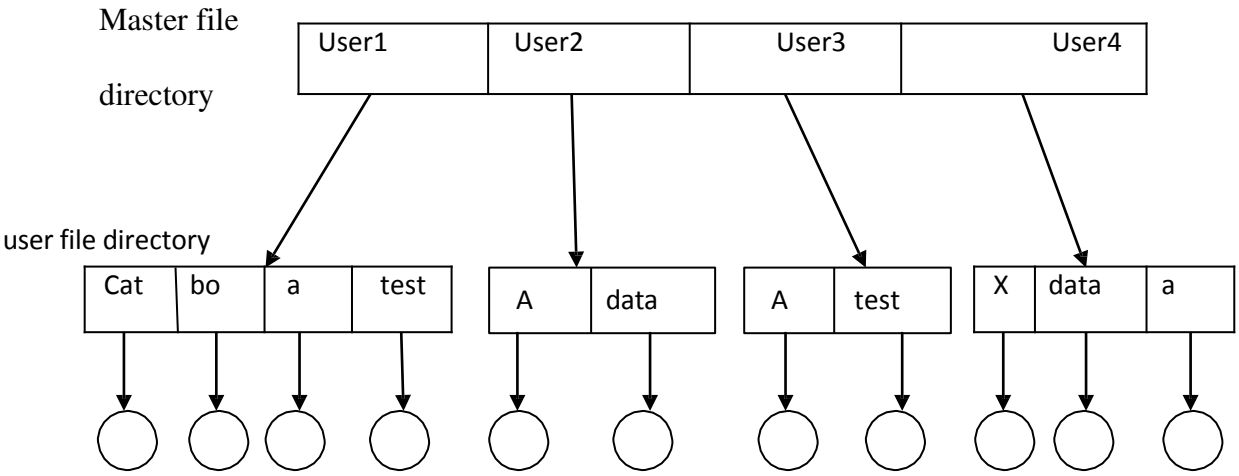
➢ Single-Level Directory

- The simplest directory structure is the single-level directory. All files are contained in the same directory, which is easy to support and understand. A single-level directory has significant limitations, however, when the number of files increases or when the system has more than one user.
- Since all files are in the same directory, they must have unique names. If two users call their data file test, then the unique-name rule is violated.
- Even a single user, as the number of files increases. It becomes difficult to remember the names of all the so to create only files with unique names.



➢ Two-Level Directory

- The major disadvantage to A single-level directory often leads to confusion of file names between different users. The standard solution is to create a *separate* directory for each user.
- In the two-level directory structure, each user has her own **user file directory (UFD).** Each UFD has a similar structure, but lists only the files of a single user.
- When a user job starts or a user logs in, the system's **master file directory (MFD)** is searched.
- The MFD is indexed by user name or account number, and each entry points to the UFD for that user. When a user refers to a particular file, only his own UFD is searched.
- Thus, different users may have files with the same name, as long as all the file names within each UFD are unique.
- To create a file for a user, the operating system searches only that user's UFD to ascertain whether another file of that name exists. To delete a file, the operating system confines its search to the local UFD, thus, it cannot accidentally delete another user's file that has the same name.
- Although the two-level directory structure solves the name-collision problem, it still has problems. This isolation is an advantage when the users are completely independent, but is a disadvantage when the users want to cooperate on some task and to access one another's files. Some system simply does not allow local files to be accessed by other users.



➢ Tree-structured directory

- this structure allows users to create their own sub directory and to organize their files accordingly. The MS-DOS system, for instance is structured as a tree. In fact a tree is the most common directory structure.
- The tree has a root directory. Every file in the system has a unique path name is the path from the root, through all the sub-directories, to a specified file. A directory contains a set of files or sub directories.
- A directory is simply another file, but it is treated in a special way. All directories have the same internal format. In normal each user has a current directory.
- The current directory should contain most of the files that are of current interest to the user. When reference is made to a file, the current directory is searched.
- If a file is needed that is not in the current directory, then the user must either specify a path name or change the current directory to be the directory that file.

➢ Directory Implementation

➢ Linear List

- The simplest method of implementing a directory is to use a linear list of file names with pointers to the data blocks. A linear list of directory entries requires a linear search to find a particular entry.
- This method is simple to program but time-consuming to execute. To create a new file, we must first search the directory to be sure that no existing file has the same name. Then, we add a new entry at the end of the directory. To delete a file, we search the directory for the named file, then release the space allocated to it. To reuse the directory entry, we can do one of several things. We can mark the entry as unused or we can attach it to a list of free directory entries.
- A third alternative is to copy the last entry in the directory into the freed location, and to decrease the length of the directory. A linked list can also be used to decrease the time to delete a file.

- The real disadvantage of a linear list of directory entries is the linear search to find a file. Directory information is used frequently, and users would notice a slow implementation of access to it would be noticed by the users.

➢ Hash Table

- Another data structure that has been used for a file directory is a hash table. In this method, a linear list stores the directory entries, but a hash data structure is also used.
- The hash table takes a value computed from the file name and returns a pointer to the file name in the linear list.
- Therefore, it can greatly decrease the directory search time.
- The major difficulties with a hash table are its generally fixed size and the dependence of the hash function on that size.

CHAPTER 13: I/O SYSTEM

➢ Polling

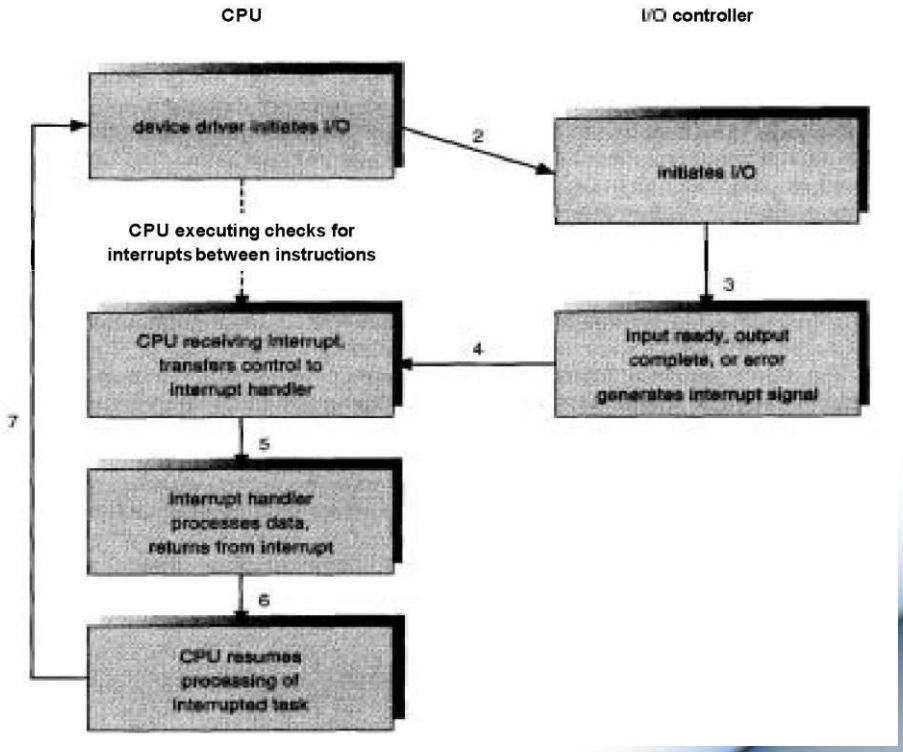
- The complete protocol for interaction between the host and a controller can be intricate, but the basic handshaking notion is simple.
- The controller indicates its state through the busy bit in the status register. The controller sets the busy bit when it is busy working, and clears the bit when it is ready to accept the next command.

1. The host signals its wishes via the command-ready bit in the command register. The host sets the *command-* ready bit when a command is available for the controller to execute.
2. The host repeatedly reads the busy bit until that bit becomes clear.
3. The host sets the write bit in the command register and writes a byte in to the data-out register.
4. The host sets the command-ready bit.
5. When the controller notices that the command-ready bit is set, it sets the busy bit.
6. The controller reads the command register and sees the `write` command. It reads the data-out register to get the byte, and does the I/0 to the device.
7. The controller clears the *command-ready* bit, clears the *error* bit in the status register to indicate that the device I/O succeeded, and clears the *busy* bit to indicate that it is finished.

➢ Interrupts

- The basic interrupt mechanism works as follows. The CPU hardware has a wire called the interrupt-request line that the CPU senses after executing every instruction.
- When the CPU detects that a controller has asserted a signal on the interrupt request line, the CPU saves a small amount of state, such as the current value of the instruction
pointer, and jumps to the interrupt-handler routine at a fixed address in memory.

- The interrupt handler determines the cause of the interrupt, performs the necessary processing, and executes a `return from interrupt` instruction to return the CPU to the execution state prior to the interrupt.
- We say that the device controller *raises* an interrupt by asserting a signal on the interrupt request line, the CPU *catches* the interrupt and *dispatches* to the interrupt handler, and the handler *clears* the interrupt by servicing the device.
- This basic interrupt mechanism enables the CPU to respond to an asynchronous event, such as a device controller becoming ready for service. In a modern operating system, we need more sophisticated interrupt-handling features.
- First, we need the ability to defer interrupt handling during critical processing.
- Second, we need an efficient way to dispatch to the proper interrupt handler for a device, without first polling all the devices to see which one raised the interrupt.
- Third, we need multilevel interrupts, so that the operating system can distinguish between high and low urgency.

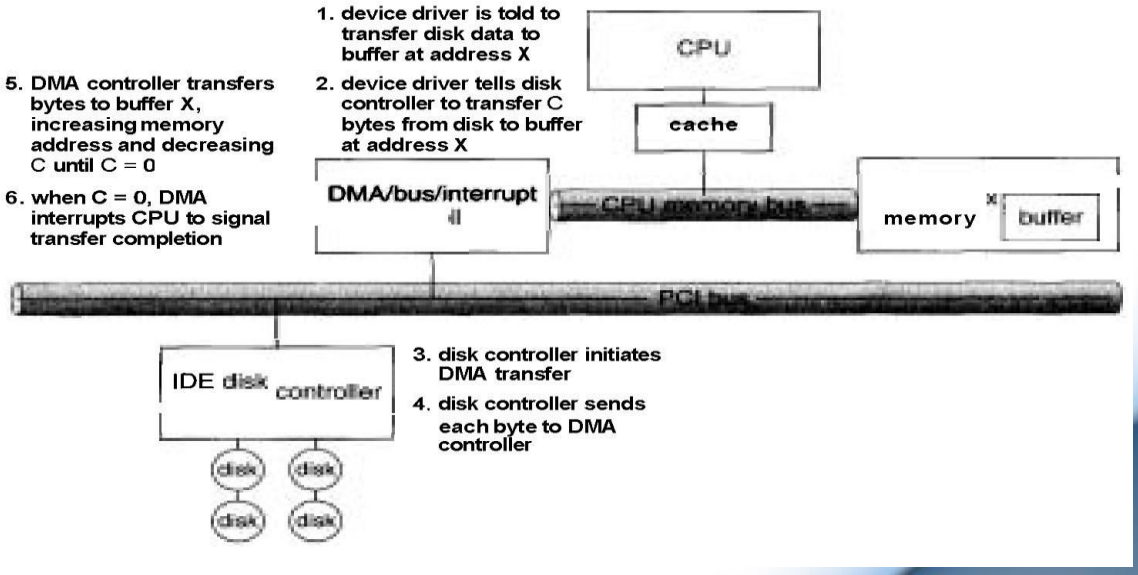


Interrupt-Driven I/O Cycle

➢ Direct Memory Access

- For a device that does large transfers, such as a disk drive, it seems wasteful to use an expensive general-purpose processor.
- Many computers avoid burdening the main CPU with PIO by offloading some of this work to a special-purpose processor called a direct- memory-access (DMA) controller.
- To initiate a DMA transfer, the host writes a DMA command block into memory.

- This block contains a pointer to the source of a transfer, a pointer to the destination of the transfer, and a count of the number of bytes to be transferred.
- The CPU writes the address of this command block to the DMA controller, then goes on with other work. The DMA controller proceeds to operate the memory bus directly, placing addresses on the bus to perform transfers without the help of the main CPU.
- Handshaking between the DMA controller and the device controller is performed via a pair of wires called DMA- request and DMA- acknowledge.
- The device controller places a signal on the DMA- request wire when a word of data is available for transfer.
- This signal causes the DMA controller to seize the memory bus, to place the desired address on the memory-address wires, and to place a signal on the DMA- acknowledge wire.
- When the device controller receives the DMA- acknowledge signal, it transfers the word of data to memory, and removes the DMA- request signal. When the entire transfer is finished, the DMA controller interrupts the CPU.



Steps in DMA-Transfer

CHAPTER 14: DISK SCHEDULING

➢ Secondary Storage device

- Disk scheduling one of the responsibilities of the operating system is to take the hardware efficiently. For the disk drives, this means having a fast access time and the disk bandwidth.

- The access time has two major components. The seek time is the time for the disk arm to move the heads to the cylinder containing the desired sector.
- The rotational latency is the additional time waiting for the disk to rotate the desired sector to the disk head.
- The 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.
- We can improve both the access time and the bandwidth by scheduling the servicing of disk I/O request in a good order.
- One of the responsibilities of the operating system is to take the hardware efficiently. For the disk drives, this means having a fast access time and the disk bandwidth.
- The access time has two major components. The seek time is the time for the disk arm to move the heads to the cylinder containing the desired sector.
- The rotational latency is the additional time waiting for the disk to rotate the desired sector to the disk head.
- The 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.
- We can improve both the access time and the bandwidth by scheduling the servicing of disk I/O request in good order.

➢ FCFS Scheduling

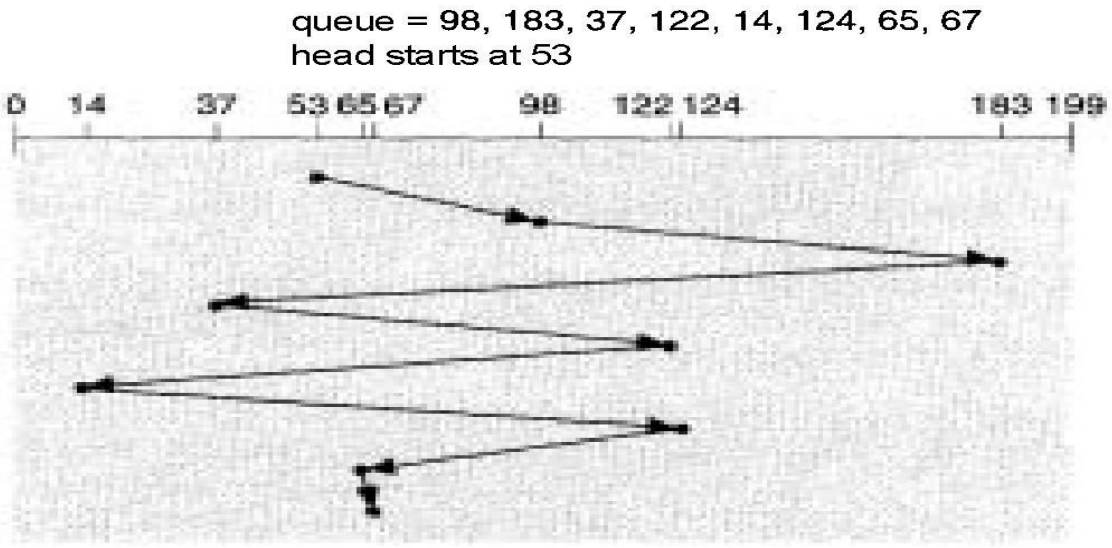
The simplest form of disk scheduling is, of course, the first-come, first-served (FCFS) algorithm. This algorithm is intrinsically fair, but it generally does not provide the fastest service. Consider, for example, a disk queue with requests for 1/0 to blocks on cylinders in that order

1&5, 37, 122, 14, *124.* 65, 67

If the disk head is initially at cylinder 53, it will first move from 53 to 98, then to 183, 37, 122, 14, 124, 65, and finally to 67, for a total head movement of 640 cylinders.

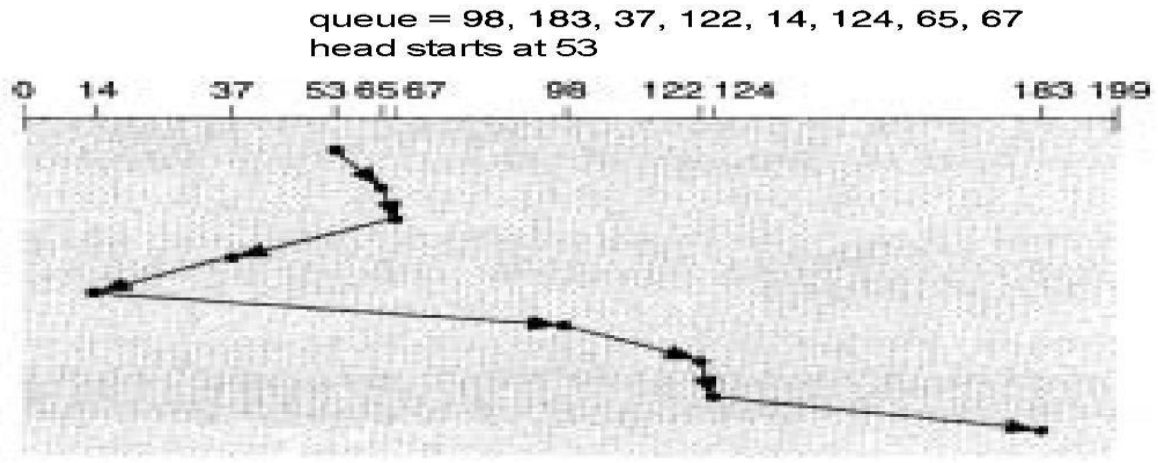
The problem of this scheduling is illustrated by the wild swing from 122 to 14 and then back to 124 illustrates the problem with this schedule.

If the requests for cylinders 37 and 14 could be serviced together, before or after the requests at 122 and 124, the total head movement could be decreased substantially, and performance could be thereby improved.



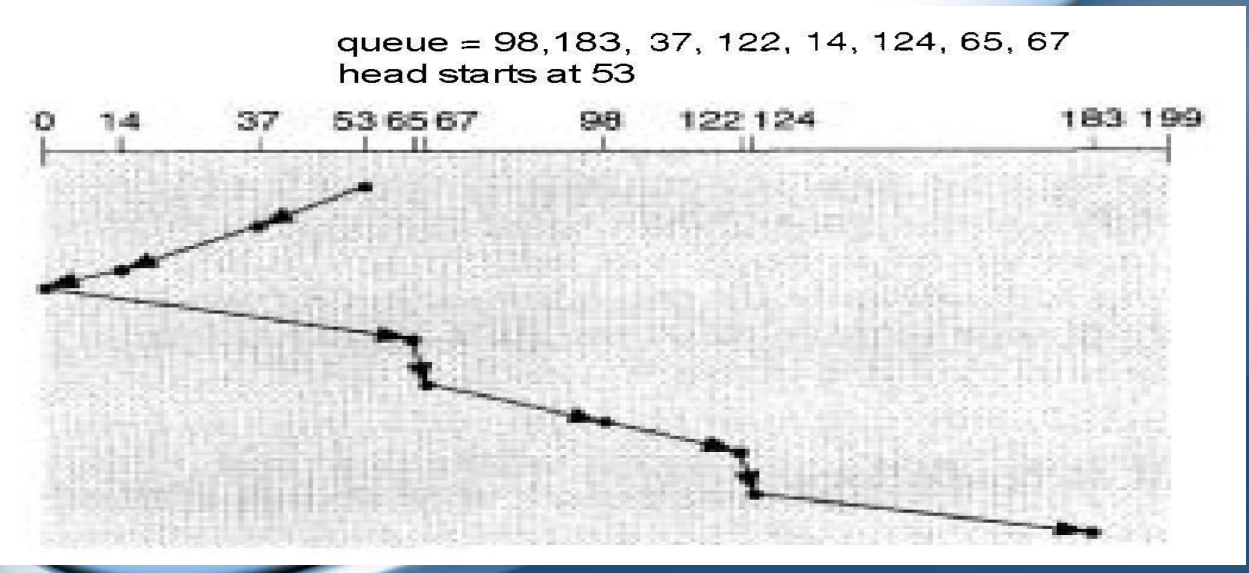
➢ SSTF Scheduling

- In this scheduling it service the entire request that are near to the current head position, before moving the head far away to service other requests.
- This assumption is the basis for the shortest-seek-time-first (SSTF) algorithm. The SSTF algorithm selects the request with the minimum seek time from the current head position.
- Since seek time increases with the number of cylinders traversed by the head, SSTF chooses the pending request closest to the current head position.
- For our example request queue, the closest request to the initial head position *(53)* is at cylinder 65. Once we are at cylinder 65, the next closest request is at cylinder 67.
- From there, the request at cylinder 37 is closer than 98, so 37 is served next. Continuing, we service the request at cylinder 14, then 98, 122, 124, and finally 183.
- This scheduling method results in a total head movement of only 236 cylinders — little more than one-third of the distance needed for FCFS scheduling of this request queue. This algorithm gives a substantial improvement in performance.
- SSTF scheduling is essentially a form of shortest—job—first (SJF) scheduling, and, like SJF scheduling, it may cause starvation of some requests. Remember that requests may arrive at any time. Suppose that we have two requests in the queue, for cylinders 14 and 186, and while servicing the request from 14, a new request near 14 arrives. This new request will be serviced next, making the request at 186 wait. While this request is being serviced, another request closes to could arrive. In theory, a continual stream of requests near one another could arrive, causing the request for cylinder 186 to wait indefinitely.



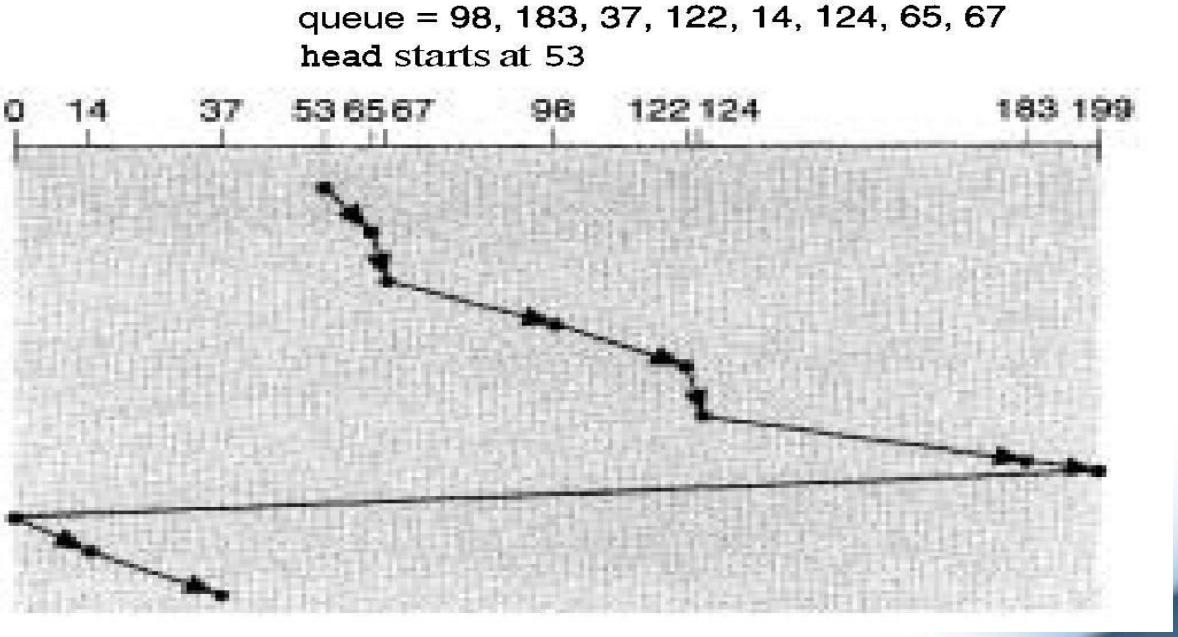
SCAN Scheduling

- In this scheduling, the disk arm starts at one end of the disk, and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk.
- At the other end, the direction of head movement is reversed, and servicing continues. The head continuously scans back and forth across the disk.
- We again use our example. Before applying SCAN to schedule the requests on cylinders 98, 183, 37, 122, 14, 124, 65, and 67; we need to know the direction of head movement, in addition to the head's current position (53).
- If the disk arm is moving toward the head will service 37 and then 14. At cylinder 0, the arm will reverse and will move toward the other end of the disk, servicing the requests at 65, 67, 98, 122, 124, and 183.
- If a request arrives in the queue just in front of the head, it will be serviced almost immediately; a request arriving just behind the head will have to wait until the arm moves to the end of the disk, reverses direction, and comes back.



➢ C-SCAN Scheduling

- Circular SCAN (C-SCAN) scheduling is a variant of SCAN designed to provide a more uniform wait time.
- Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing requests along the way.
- When the head reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip (Figure 14.4). The C-SCAN scheduling algorithm essentially treats the cylinders as a circular list that wraps around from the final cylinder to the first one.



Thank You