Operating System and System Software

UNIT 1

Overview of Operating System: Batch Processing, Multiprogrammed, Time-Sharing, Multiprocessor, Real-Time Systems. Operating System Structures: System Components, Operating System Services, System Calls, File Systems Interface: File Concept, Access Methods, Directory Structure.

UNIT 2

CPU Scheduling: Basic Concepts, Scheduling Criteria, Scheduling Algorithms. Deadlocks: Deadlock Characterization, Methods for Handling Deadlocks, Deadlock Prevention, Avoidance and Detection, Recovery from Deadlock.

UNIT 3.

Memory Management: Background, Swapping, Contiguous Memory Allocation, Paging, Segmentation, Segmentation with Paging, Virtual Memory: Background, Demand Paging, Page Replacement, Allocation of Frames.

UNIT 4

Mass Storage Structure: Disk Structure, Disk Scheduling- FCFS, SSTF, SCAN Scheduling, Disk Management, Swap-Space Management.

UNIT 5

System software and application software, layered organisation of system software. Assemblers, Macros, Compilers, Cross compilers, Linking and loading, Relocation.

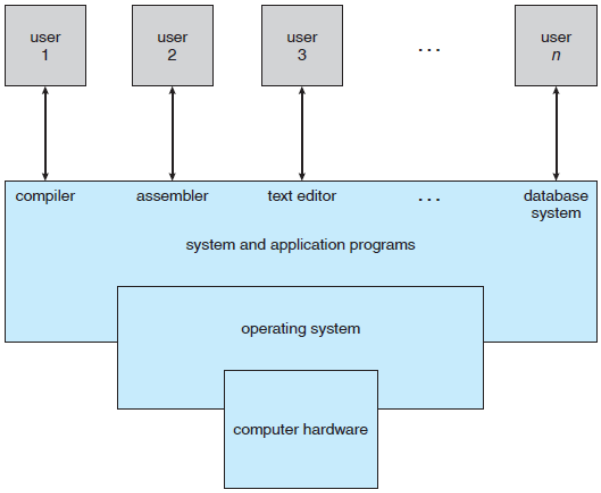
UNIT-I

Overview of Operating System

Operating System can be define in various terms like-
- An *operating system* acts as an intermediary between the user of a computer and the computer hardware.
- The purpose of an operating system is to provide an environment in which a user can execute programs in a *convenient* and *efficient* manner.
- An operating system is software that manages the computer hardware.
- an operating system is large and complex, it must be created piece by piece. Each of these pieces should be a well delineated portion of the system, with carefully defined inputs, outputs, and functions.

What Operating Systems Do:

A computer system can be divided roughly into four components: the *hardware,* the *operating system,* the *application programs,* and the *users.*



The **hardware**—the **central processing unit (CPU)**, the **memory**, and the **input/output (I/O) devices**—provides the basic computing resources for the system.

The **application programs**—such as word processors, spreadsheets, compilers, and Web browsers—define the ways in which these resources are used to solve users' computing problems.

Thus we can say the operating system controls the hardware and coordinates its use among the various application programs for the various users. An operating system is similar to a *government*. Like a government, it performs no useful function by itself. It simply provides an *environment* within which other programs can do useful work.

Types of Operating System

Simple Batch System:

Earlier system executes one job at a time and were very expensive, therefore it was important to maximize processor utilization. To improve utilization, the concept of a batch operating system was developed.

It appears that the first batch operating system (and the first OS of any kind) was developed in the mid-1950s by General Motors for use on an IBM 701 [WEIZ81]. It was called Resident monitor (small OS).

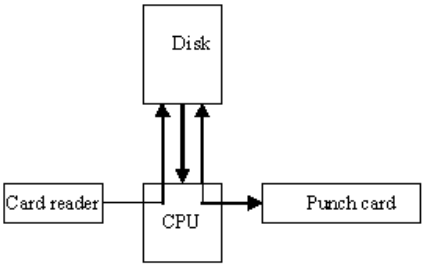
The central idea behind the simple batch-processing scheme is -
the user no longer has direct access to the processor. Instead, the user submits the job on cards or tape to a computer operator, who batches the jobs together sequentially and places the entire batch on an input device.

Thus a number of jobs with similar requirements were grouped together and are executed sequentially. Such system is called batched system.

- Ex: One batch of all Fortran program, one batch of all COBOL program.

Spooling: First a number of jobs are collected from users by an operator and copied from cards to magnetic tape on a small, cheap computer. This tape is carried by the operator to the main computer, which executes the batch of jobs one by one, delivering their output to another tape. Finally, this output tape is carried to the small computer and listed on the printer. Notice that although jobs are executed in their order of arrival inside a batch, the printed output of the first job is not available until the entire batch has been executed.

In this way CPU never sat idle until output not printed or jobs not submitted. And this process is called spooling.



During the execution of a batch on the main computer, the operator uses the small computer to print the output of an earlier batch and input a new batch on tape. In this way the main computer, as well as the card reader and printer, is kept busy all the time. Input/output delays on the main computer are negligible in this system, but another source of idle time has

appeared: the mounting and dismounting of tapes. This can only be reduced by batching many jobs together on a single tape. But in doing so we also increase the waiting time of users for the results of their jobs.

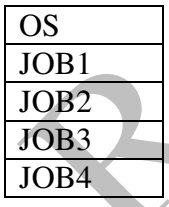
Multiprogramming:

It is a technique that allows more than one program to be ready for execution and provides the ability to switch execution from one program to another, even any program is not yet completed.

Need of multiprogramming: A single program cannot, in general, keep either the CPU or the I/O devices busy at all times. Single users frequently have multiple programs running. m**ultiprogramming** increases CPU utilization by organizing jobs (code and data) so that the CPU always has one to execute.

Mechanism:

The idea is as follows: The operating system keeps several jobs in memory simultaneously. Since, in general, main memory is too small to accommodate all jobs, the jobs are kept initially on the disk in the **job pool**. This pool consists of all processes residing on disk awaiting allocation of main memory.



The set of jobs in memory can be a subset of the jobs kept in the job pool. The operating system picks and begins to execute one of the jobs in memory. Eventually, the job may have to wait for some task, such as an I/O operation, to complete. In a non-multiprogrammed system, the CPU would sit idle. In a multiprogrammed system, the operating system simply switches to, and
executes, another job. When *that* job needs to wait, the CPU is switched to *another* job, and so on. Eventually, the first job finishes waiting and gets the CPU back. As long as at least one job needs to execute, the CPU is never idle.

Benefit: Multiprogrammed systems provide an environment in which the various system resources (for example, CPU, memory, and peripheral devices) are utilized effectively.

Limitation: user cant interact and debug with the job when it is executing.

Time-Sharing Systems (Multitasking):

Time sharing (or **multitasking**) is a logical extension of multiprogramming. In time-sharing systems, the CPU executes multiple jobs by switching among them, but the switches occur so frequently that the users can interact with each program while it is running.

Time sharing requires an **interactive** (or **hands-on**) **computer system**, which provides direct communication between the user and the system. The user gives instructions to the operating system or to a program directly, using a input device such as a keyboard or a mouse, and waits for immediate results on an output device.

A time-shared operating system allows many users to share the computer simultaneously. Since each action or command in a time-shared system tends to be short, only a little CPU time is needed for each user. As the system switches rapidly from one user to the next, each user is given the impression that the entire computer system is dedicated to his use, even though it is being shared
among many users.

A time-shared operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time-shared computer. Each user has at least one separate program in memory.
When a process executes, it typically executes for only a short time before it either finishes or needs to perform I/O.

Time sharing needs following task-

job scheduling:

If several jobs are ready to be brought into memory, and if there is not enough room for all of them, then the system must choose among them. Making this decision is **job scheduling.**

memory management:

When the operating system selects a job from the job pool, it loads that job into memory for execution. Having several programs in memory at the same time requires some form of memory management.

CPU scheduling :

If several jobs are ready to run at the same time, the system must choose among them. Making this decision is **CPU scheduling.**

Multiprocessor Systems

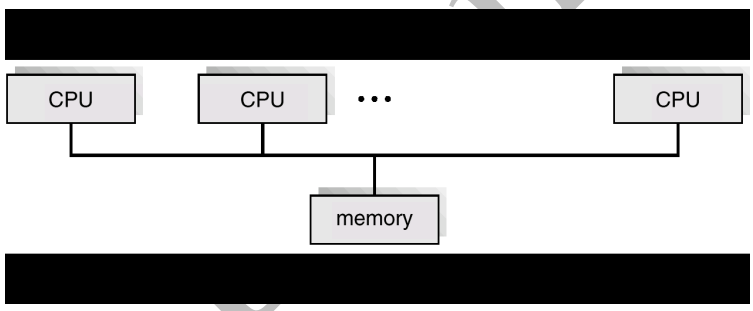
Multiprocessor systems have two or more processors in close communication, sharing the computer bus and sometimes the clock, memory, and peripheral devices.

Type of Multiprocessing:

1) Asymmetric multiprocessing: in which each processor is assigned a specific task. A master processor controls the system; the other processors

either look to the master for instruction or have predefined tasks. This scheme defines a master–slave relationship. The master processor schedules and allocates work to the slave processors.
2) **symmetric multiprocessing (SMP):** in which each processor performs all tasks within the operating system. SMP means that all processors are peers; no master–slave relationship exists between processors.
3) Tightly coupled system – processors share memory and a clock; communication usually takes place through the shared memory.

Multiprocessor systems have three main advantages:
1. Increased throughput. By increasing the number of processors, we expect to get more work done in less time.
2. Economy of scale. Multiprocessor systems can cost less than equivalent multiple single-processor systems, because they can share peripherals, mass storage, and power supplies.
3. Increased reliability. If functions can be distributed properly among several processors, then the failure of one processor will not halt the system, only slow it down.



Real Time System

Definition:

A real-time system has well-defined, fixed time constraints. Processing *must* be done within the defined constraints, or the system will fail. A real-time system functions correctly only if it returns the correct result within its time constraints.

A real-time system is used when rigid time requirements have been placed on the operation of a processor or the flow of data; thus, it is often used as a control device in a dedicated application.

Sensors bring data to the computer. The computer must analyze the data and possibly adjust controls to modify the sensor inputs.

Ex: For instance, it would not do for a robot arm to be instructed to halt *after* it had smashed into the car it was building.

Some real time systems: Systems that control scientific experiments, medical imaging systems, industrial control systems, and certain display systems are realtime systems. Some

automobile-engine fuel-injection systems, home-appliance controllers, and weapon systems are also real-time systems.

==========

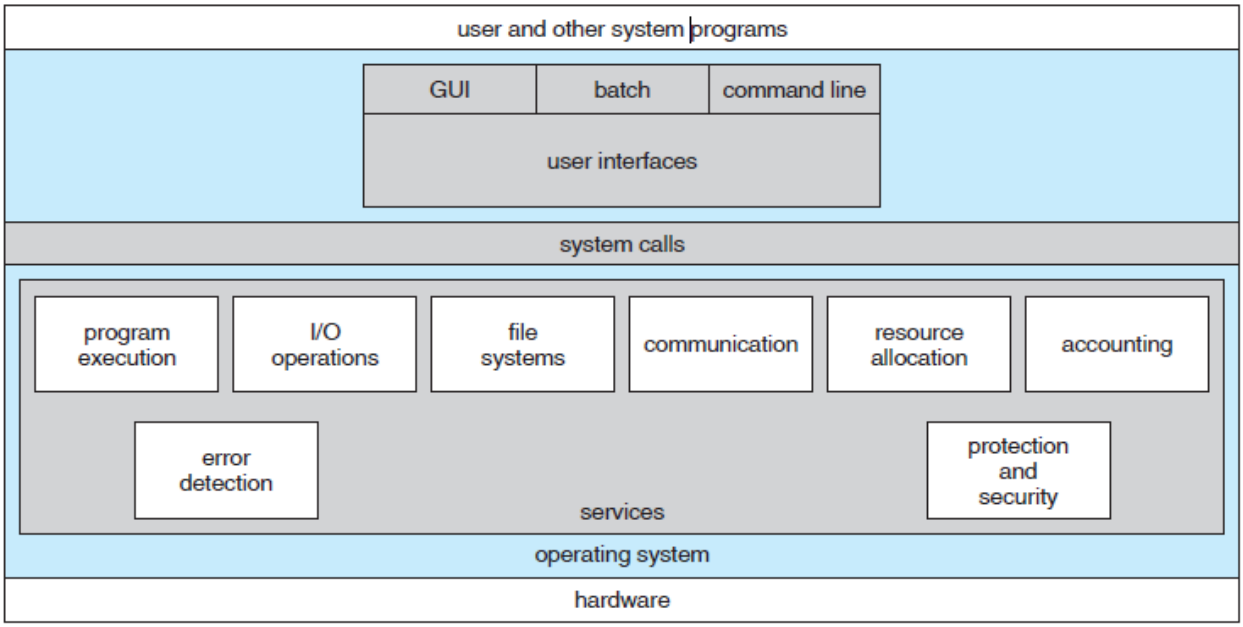
Operating -System Structures

An operating system provides the environment within which programs are executed. The design of a new operating system is a major task. It is important that the goals of the system be well defined before the design begins. We can view an operating system from several points.

1) services that the system provides
2) the interface that it makes available to users and programmers
3) components and their interconnections.

Operating-System Services

An operating system provides an environment for the execution of programs. It provides certain services to programs and to the users of those programs. These operating-system services are provided for the convenience of the programmer, to make the programming task easier.



• **Program execution**. The system must be able to load a program into memory and to run that program. The program must be able to end its execution, either normally or abnormally (indicating error).

• **I/O operations**. A running program may require I/O, which may involve a file or an I/O device. For specific devices, special functions may be desired . users usually cannot control I/O devices directly. Therefore, the operating system must provide a means to do I/O.

• **File-system manipulation**. Programs need to read and write files and directories. They also need to create and delete them by name, search for a given file, and list file information.

Operating systems provide a variety of file systems, to provide specific features or performance characteristics.

• **Communications**. There are many circumstances in which one process needs to exchange information with another process. Such communication may be implemented via *shared memory* or through *message passing,* in which packets of information are moved between processes by the operating system.

• **Error detection**. The operating system needs to be constantly aware of possible errors like- memory, cpu failure, a connection failure on a network, arithmetic overflow etc. For each type of error, the operating system should take the appropriate action to ensure correct and consistent computing.

• **Resource allocation**. When there are multiple users or multiple jobs running at the same time, resources (such as CPU cycles, main memory, and file storage) must be allocated to each of them. These resources are managed by the operating system.

• **Accounting**. We want to keep track of which users use how much and what kinds of computer resources. This record keeping may be used for accounting.

• **Protection and security**. The owners of information stored in a multi user or networked computer system may want to control use of that information. Protection involves ensuring that all access to system resources is controlled. Security of the system from outsiders is usually by means of a password.

System Calls

System calls provide an interface to the services made available by an operating system. These calls are generally available as routines .

Use of system calls:

For example-writing a simple program to read data from one file and copy them to another file. This operation needs a number of system calls for prompting message on screen, read input and output filenames from the keyboard, to open the input file and create the output file, read content from input file and copy to output file. Finally, close both files.

Thus we can see, simple programs may make heavy use of the operating system and systems execute thousands of system calls per second.

Types of System Calls

System calls can be grouped roughly into six major categories: Process control, file manipulation, device manipulation, information maintenance, communications, and protection.

1) Process Control :A running program needs to be able to halt its execution either normally (end) or abnormally (abort),
A process or job executing one program may want to load and execute another program etc. In that case process control system calls are applicable.
Some of process control system calls are following-
◦ end, abort

◦ load, execute
◦ create process, terminate process
◦ get process attributes, set process attributes
◦ wait for time
◦ wait event, signal event
◦ allocate and free memory
EXAMPLES OF some UNIX SYSTEM CALLS: fork(), exit(), wait() etc.

2) file manipulation:Such system call requires the name of the file and file's attributes. Once the file is created, we need to open it and to use it for read, write, or reposition, Finally close the file, that we are no longer using it. In that case following system calls are applicable.
◦ create file, delete file
◦ open, close
◦ read, write, reposition
◦ get file attributes, set file attributes.
EXAMPLES OF some UNIX SYSTEM CALLS: open(), read(),write(), close() etc

3) Device management: A process may need several resources to execute—main memory, disk drives, access to files, and so on. If the resources are available, they can be granted, and control can be returned to the user process. Otherwise, the process will have to wait until sufficient resources are available. In that case following system calls are applicable.
◦ request device, release device
◦ read, write, reposition
◦ get device attributes, set device attributes
◦ logically attach or detach devices

EXAMPLES OF some UNIX SYSTEM CALLS: ioctl(), read(), write()

4) Information maintenance: Such system calls exist simply for the purpose of transferring information between the user program and the operating system. They are-
◦ get time or date, set time or date
◦ get system data, set system data
◦ get process, file, or device attributes
◦ set process, file, or device attributes
EXAMPLES OF some UNIX SYSTEM CALLS: getpid(), alarm(), sleep()

5) Communications:
Messages can be exchanged between the processes either directly or indirectly through a common mailbox. Before communication can take place, a connection must be opened. Following system calls used-
◦ create, delete communication connection
◦ send, receive messages
◦ transfer status information
◦ attach or detach remote devices
EXAMPLES OF some UNIX SYSTEM CALLS: pipe(),shmget(),mmap()
6) Protection:
It provides a mechanism for controlling access to the resources provided by a computer system. Following system calls fall to this one-
- File Security

- Initialize Security Descriptor
Ex: chmod(), umask()

System Components

1) **process management:** A process can be thought of as a program in execution. A system consist a collection of processes, some of which are operating system process and rest of which are user processes. All these process are executed concurrently, by multiplexing cpu among them.
   The operating system is responsible for the following activities in connection with process management.
- The creation and deletion of both user and system processes
- The suspension and resumption of processes
- The provision of mechanisms for process synchronization
- The provision of mechanisms for process communication
- The provision of mechanisms for deadlock handling

2) **Main memory management:** Main memory is a large array of words or bytes. Data from main memory are quickly accessed by CPU. To speed up of program execution, we keep all of them in to memory. Selection of memory management depends on its hardware design.
   The operating system is responsible for the following activities in connection with memory management.
- Keep track of which parts of memory are currently being used and by whom.
- Decide which processes are to be loaded into memory when memory space becomes available.
- Allocate and deallocate memory space as needed.

3) **Secondary-storage management:**
   secondary storage provides the backup of the devices. The operating system is responsible for the following activities in connection with disk management.
- Free-space management.
- Storage allocation.
- Disk scheduling.

4) **File management:** It is one of the most visible components of O.S. The OS implements the abstract concept of a file managing mass storage media such as tap and disks, and devices which control them.
   The operating system is responsible for the following activities in connection with file management.
- The creation and deletion of files
- The creation and deletion of directories
- The support of primitives for manipulating files and directories
- The mapping of files onto secondary storage
- The backup of files on stable (nonvolatile) storage media

5) **Command interpreter System:** Command Interpreter is the user interface between user and operating system. It is treated as a special program that is running when a job is initiated or when a user first logs on. It reads commands from the user or from a file of commands and executes them, usually by turning them into one or more system calls.

Sometimes it is also called control card interpreter or *shell*. Its function is simple- gets commands from the user and execute it.
It is usually not part of the kernel since the command interpreter is subject to changes.

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

Files are mapped by the operating system onto physical devices such as disk, tape etc. These storage devices are usually nonvolatile, so the contents are persistent through power failures and system reboots.

A file is a named collection of related information that is recorded on secondary storage. 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.

Many different types of information may be stored in a file—source programs, object programs, executable programs, numeric data, text, payroll records, graphic images, sound recordings, and so on.

File Attributes: A file's attributes typically consist of theses -

- Name: The symbolic file name is the only information kept in human readable form.
- Type: This information is needed for systems that support different types of files.
- Location: This information is a pointer to a device and to the location of the file on that device.
- Size: The current size of the file (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.

The information about all files is kept in the directory structure, which also resides on secondary storage.

File Operations

Following operation can be performed with any file:

» **Creating a file.** When space in the file system is found for the file then an entry for the new file must be made in the directory.

• **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.

• **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.

» **Repositioning within** a **file. T**he current-file-position pointer is repositioned to a given value.
This file operation is also known as a file *seek.*

• **Deleting a file.** To delete a file, we search the directory for the named file.

• **Truncating a file.** forcing the user to delete the file and then recreate it, this function allows all attributes to remain unchanged—except for file lengt.

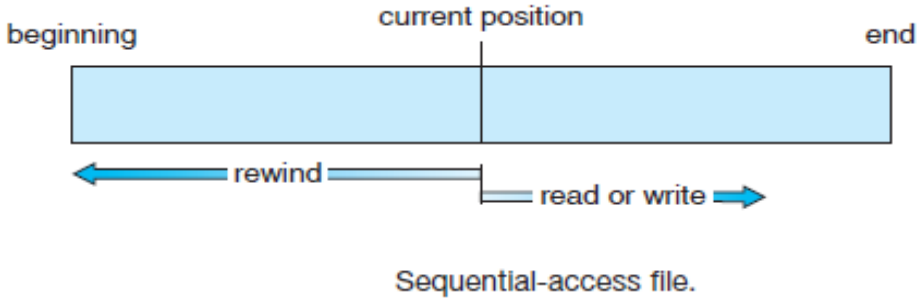
File Type: File types is a part of the file name. The name is split into two parts—a name and an *extension,*
usually separated by a period character . In this way, the user and the operating system can tell from the name alone what the type of a file is.
For example: *resume.doc, Server.java,* and *ReaderThread.c.*
Common type of file system:

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

Access Methods

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

1 Sequential Access:
The simplest access method is **sequential access**. Information in the file is processed in order, one record after the other. for example, editors and compilers usually access files in this fashion.

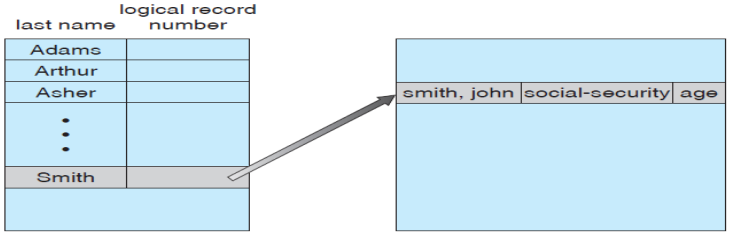


Sequential-access file.

Reads and writes make up the bulk of the operations on a file. A read operation—*read next*—reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location. Similarly, the write
operation—*write next*—appends to the end of the file and advances to the end of the newly written material (the new end of file). Such a file can be reset to the beginning. Sequential access, is based on a tape model of a file and works as well on sequential-access devices as it does on random-access ones.

2 Direct Access: (**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. This method is based on a disk model of a file, since disks allow random access to any file block. For direct access, the file is viewed as a numbered sequence of blocks or records. 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.
Ex: Databases are often of this type. When a query concerning a particular subject arrives, we compute which block contains the answer and then read that block directly to provide the desired information.
Thus the block number provided by the user to the operating system is normally a **relative block number**.

3 Other Access Methods:



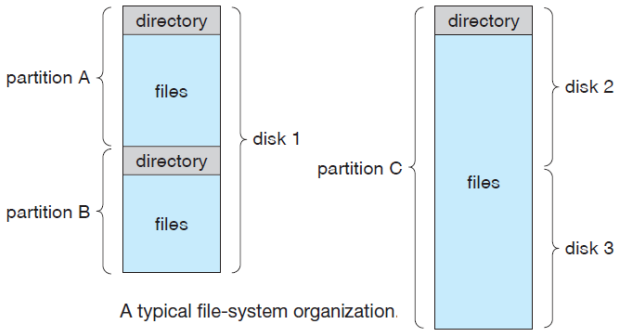
Example of index and relative files.

Other access methods can be built on top of a direct-access method. These methods generally involve the construction of an index for the file. 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 record.

Directory Structure:

File systems store millions of files on disk. To manage all these files, we need to organize them. This organization involves the use of directories.

Each disk on a system contains at least one partition. Each partition contain information about the files within it. This information is kept in entries in a **device directory** or simply directories. **Directory** records information—such as name, location, size, and type—for all files on that partition.



A typical file-system organization.

Following operations that are to be performed on a directory:
Search for a file. To find the entry for a particular file.
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 be able to remove it from the directory.
List a directory. We need to be able to list the files in a directory
Rename a file. We must be able to change the name of the file.

Traverse the file system. It is able to access every directory and every file within a directory structure.

Most common schemes for defining the logical structure of a directory:-

Single-Level Directory: All files are contained in the same directory, which is easy to support and understand.

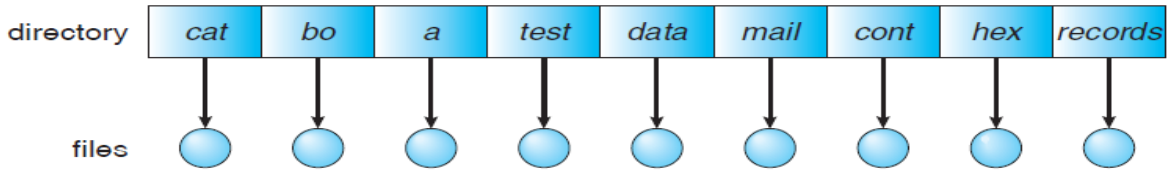


Figure Single-level directory.

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 and so it difficult to remember the names of all the files.

Two-Level Directory: In the two-level directory structure, each user has his own **user file directory (UFD)**. When a user logs in, the system's **master file directory (MFD)** is searched. The MFD is indexed by user name.

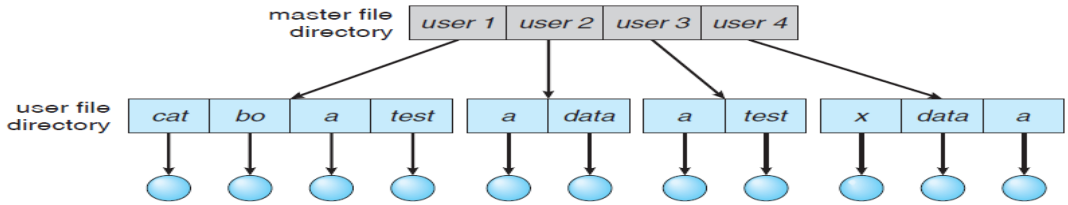


Figure Two-level directory structure.

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.

Tree-Structured Directories: A tree is the most common directory structure. The tree has a root directory, and every file in the system has a unique path name.

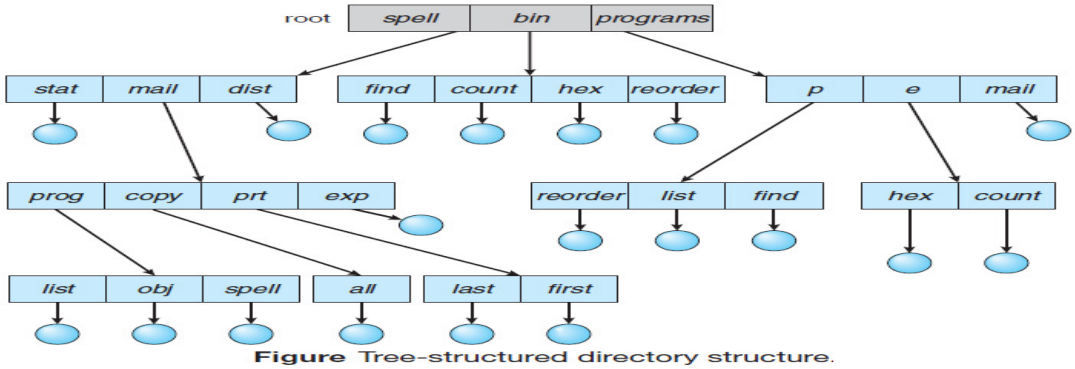


Figure Tree-structured directory structure.

A directory (or subdirectory) contains a set of files or subdirectories. All directories have the same internal format. One bit in each directory entry defines the entry as a file (0) or as a subdirectory (1).

Acyclic-Graph Directories: An **acyclic graph** (a graph with no cycles) —allows directories to share subdirectories and files.

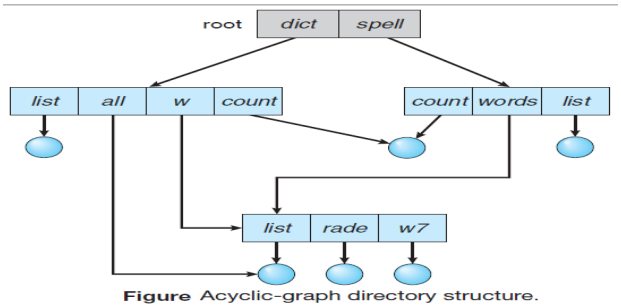


Figure Acyclic-graph directory structure.

The *same* file or subdirectory may be in two different directories.

UNIT-II

CPU Scheduling:

Switching the CPU among processes in different manner so that computer becomes more productive, it is called CPU scheduling.

Basic Concepts:

1) The objective of multiprogramming is to have some process running at all times, thus to maximize CPU utilization.
2) Several processes are kept in memory at one time. When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process. This pattern continues.
3) Scheduling of this kind is a fundamental operating-system function.

Following components are involved in the CPU scheduling -

CPU-I/O Burst Cycle:

Process execution consists of a **cycle** of CPU execution and I/O wait. Processes alternate between these two states. Process execution begins with a **CPU burst**.That is followed by an **I/O burst**, which is followed by another CPU burst, then another I/O burst, and so on. Eventually, the final CPU burst ends with a system request to terminate execution.

CPU Scheduler:

Whenever the CPU becomes idle, the scheduler selects a process from the processes in memory that are ready to execute and allocates the CPU to that process.

Preemptive Scheduling:

CPU-scheduling decisions may take place under the following four circumstances:
1. When a process switches from the running state to the waiting state. (an I/O request or wait ends child processes)
2. When a process switches from the running state to the ready state (interrupt)
3. When a process switches from the waiting state to the ready state (at completion of I/O)
4. When a process terminates.

When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is **nonpreemptive** or **cooperative**; otherwise, it is **preemptive**.
Under nonpreemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state.

Dispatcher:

The dispatcher is the module that gives control of the CPU to the process selected by the short-term .

Scheduling Criteria:

Many criteria have been suggested for comparing CPU-scheduling algorithms. The criteria include the
following:

• **CPU utilization**. We want to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).
• **Throughput**. If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed per unit time, called *throughput.*
• **Turnaround time**. How long CPU takes to execute any particular process. The interval from the time of submission of a process to the time of completion is the *turnaround time.*
Ie Turnaround time= time(waiting into memory+waiting in the ready queue+ executing on the CPU+ I/O).
• **Waiting time**. *Waiting time* is the sum of the periods spent waiting in the ready queue.
• **Response time**. The time from the submission of a request until the first response is produced. This measure, called *response time.*

It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time, and response time.

Scheduling Algorithms:

It deciding which of the processes in the ready queue is to be allocated the CPU. There are many different CPU-scheduling algorithms.

1) First Come, First Serve (FCFS)
2) Shortest Job First (SJF)
3) Priority
4) Round Robin
5) Multilevel Queue
6) Multilevel Feedback Queue

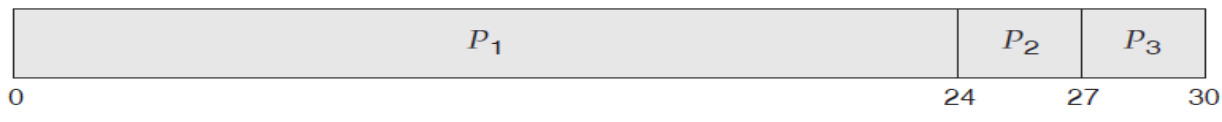
1) First-Come, First-Served Scheduling: With this scheme, the process that requests the CPU first is allocated the CPU first. It is easily managed with a FIFO queue.

When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue.
The average waiting time under the FCFS policy is often quite long.

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

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

Gantt chart:
If the processes arrive in the order *P*1, *P*2, *P*3, and are served in FCFS order, we get the result like this –



The waiting time for process:
P1= 0 milliseconds
P2=24 milliseconds
P3= 27 milliseconds
Thus, the average waiting time=(0+ 24 + 27)/3 = 17 milliseconds.
FCFS scheduling algorithm is nonpreemptive. Once the

CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O.

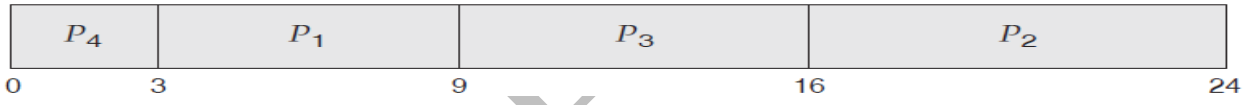
---

2) Shortest-Job-First Scheduling: (*shortest-next-CPU-burst)*
When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie.

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

Process	Burst Time
*P*1	6
*P*2	8
*P*3	7
*P*4	3

Gantt Chart: we would schedule these processes according to this algorithms like this-



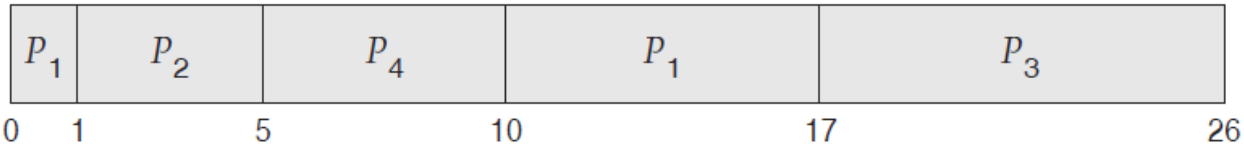
The waiting time for the processes-
P1= 3 milliseconds
P2= 16 milliseconds
P3= 9 milliseconds
P4= 0 milliseconds
Average waiting time = (3 + 16 + 9 + 0)/4 = 7 milliseconds.

The SJF algorithm can be either preemptive or nonpreemptive. When a new process arrives at the ready queue while a previous process is still executing. Let the CPU burst of the new arrived process is shorter than currently executing process.

Process	Arrival Time	Burst Time
*P*1	0	8
*P*2	1	4
*P*3	2	9
*P*4	3	5

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

Gantt chart for Preemptive SJF scheduling:



Waiting Time for the process:
P1=0+(10-1)=9
P2=(1-1)=0
P3=(17-2)=15
P4=(5-3)=2
The average waiting time= [9 +0 + 15 + 2]/4 = 26/4 = 6.5 milliseconds.

---

3) Priority Scheduling: A priority is associated with each process, and the CPU is allocated to the process
with the highest priority. Equal-priority processes are scheduled in FCFS order.

we assume that low numbers represent high priority.
As an example, consider the following set of processes, assumed to have arrived at time 0 in the order *P*1, *P*2, ・ ・ ・, *P*5, with the length of the CPU burst given in milliseconds are-

Process	**Burst Time**	**Priority**
*P*1	10	3
*P*2	1	1
*P*3	2	4
*P*4	1	5
*P*5	5	2

Gantt Chart:

Waiting time for the process:
P1=6
P2=0
P3=16
P4=18
P5=1
The average waiting time=(6+0+16+18+1)/5=41/5= 8.2 milliseconds.
A major problem with priority scheduling algorithms is **indefinite blocking**, or **starvation**. A process that is ready to run but waiting for the CPU can be considered blocked. A priority scheduling algorithm can leave
some low priority processes waiting indefinitely.

---

4) Round-Robin Scheduling:

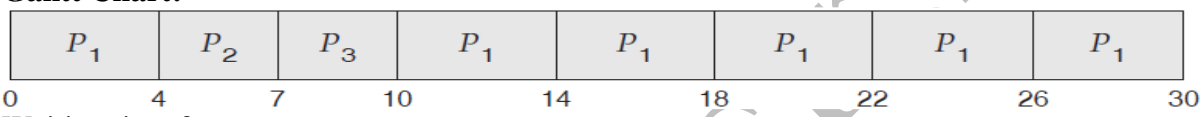
It is designed especially for timesharing systems. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum. To implement RR scheduling, we keep the ready queue as a FIFO queue of processes. The CPU scheduler picks the first process from the ready queue, sets a timer to1 time quantum. If the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and the process will be put at the **tail** of the ready queue. The CPU scheduler will then select the next process in the ready queue. This process continued to last process of ready queue.

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

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

If we use a time quantum of 4 milliseconds.

Gantt Chart:



Waiting time for process:
P1=0+(10-4)=6
P2=4
P3=7

Average waiting time= (6+ 4 +7)/3=17/3 = 5.66 milliseconds.

5) Multilevel Queue Scheduling:

A **multilevel queue scheduling algorithm** partitions the ready queue into several separate queues.

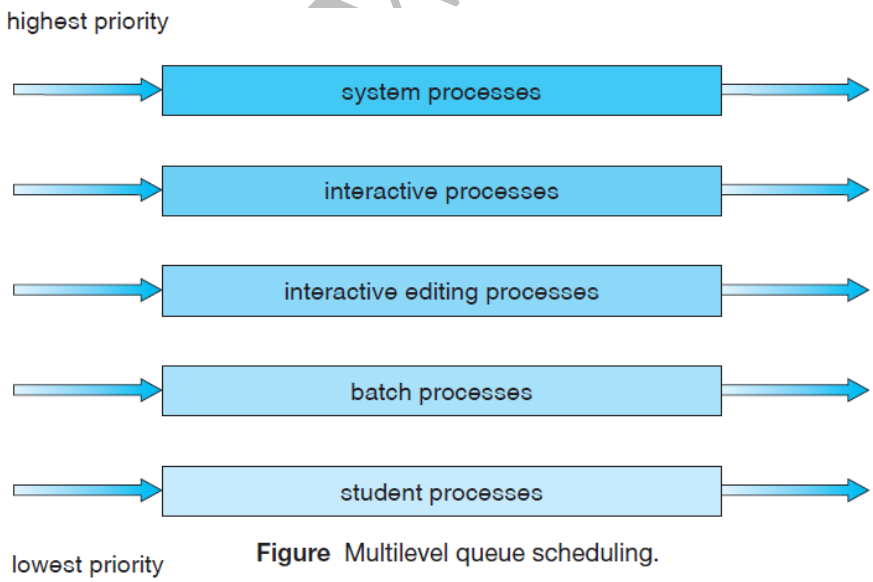


Figure Multilevel queue scheduling.

Each queue has its own scheduling algorithm. For example each queue has absolute priority over lower-priority queues that means no process in the batch queue, could run unless the queues for system processes,
interactive processes, and interactive editing processes were all empty.

6) Multilevel Feedback Queue Scheduling:

It allows a process to move between queues. If a process uses too much CPU time, it will be moved to a lower-priority queue.

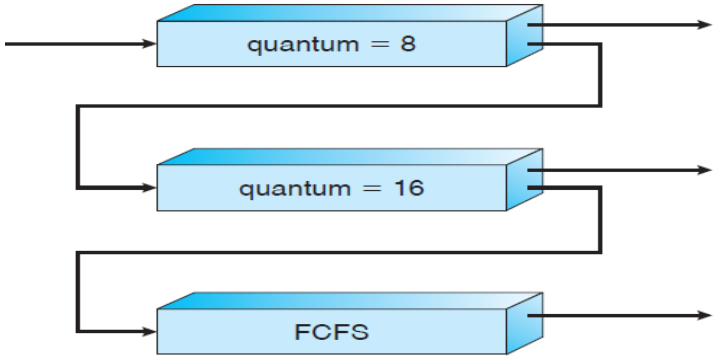


Figure Multilevel feedback queues.

Deadlock

In a multiprogramming environment, when a process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a **deadlock**. Memory space, CPU cycles, files, and I/O devices (such as printers andDVDdrives) are examples of resource types.

Deadlock Characterization:

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

1. Mutual exclusion. 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. A resource can be released only by the process holding it, after that process has completed its task.

4. Circular wait. A set *{P*0, *P*1, ..., *Pn}* of waiting processes must exist such that *P*0 is waiting for a resource held by *P*1, *P*1 is waiting for a resource held by *P*2, ..., and *Pn* is waiting for a resource held by *P*0.

Resource-Allocation Graph: This graph consists of a set of vertices *V* and a set of edges *E*.

Let there are two set of vertices-one for no of processes and other for no of resources. Edges are directed line between process and resources.

- The sets *P, R,* and *E*:
  - *P = {P*1, *P*2, *P*3*}*
  - *R = {R*1, *R*2, *R*3, *R*4*}*
  - *E = {P*1 → *R*1, *P*2 → *R*3, *R*1 → *P*2, *R*2 → *P*2, *R*2 → *P*1, *R*3 → *P*3*}*

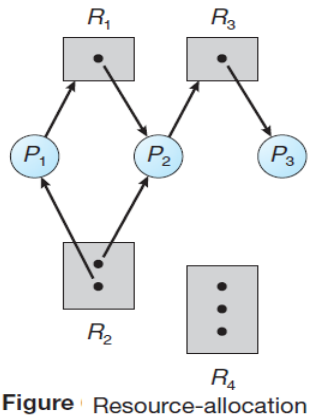


Figure Resource-allocation graph.

- Resource instances:
  - One instance of resource type *R*1
  - Two instances of resource type *R*2
  - One instance of resource type *R*3
  - Three instances of resource type *R*4
- Process states:

◦ Process $P1$ is holding an instance of resource type $R2$ and is waiting for an instance of resource type $R1$.
◦ Process $P2$ is holding an instance of $R1$ and an instance of $R2$ and is waiting for an instance of $R3$.
◦ Process $P3$ is holding an instance of $R3$.
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.

Suppose that process $P3$ requests an instance of resource type $R2$. Since no resource instance is currently available, a request edge $P3 \rightarrow R2$ is added to the graph.

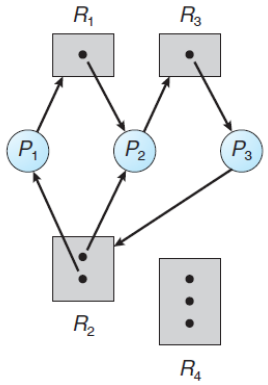


Figure Resource-allocation graph with a deadlock.

At this point, two minimal cycles exist in the system:
1) $P1 \rightarrow R1 \rightarrow P2 \rightarrow R3 \rightarrow P3 \rightarrow R2 \rightarrow P1$
2) $P2 \rightarrow R3 \rightarrow P3 \rightarrow R2 \rightarrow P2$

Processes $P1$, $P2$, and $P3$ are deadlocked.

Methods for Handling Deadlocks:

We can deal with the deadlock problem in one of three ways:
• We can use a protocol to prevent or avoid deadlocks, ensuring that the system will *never* enter a deadlocked state.
• We can allow the system to enter a deadlocked state, detect it, and recover.
• We can ignore the problem altogether and pretend that deadlocks never occur in the system.
Following four methods for handling deadlocks:-
1) Deadlock Prevention
2) Deadlock Avoidance
3) Deadlock Detection
4) Recovery from Deadlock
To ensure that deadlocks never occur, the system can use either a deadlock prevention or a deadlock-avoidance scheme. If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may arise.

If a system does not ensure that a deadlock will never occur and also does not provides a mechanism for deadlock detection and recovery, in this case, more and more processes, make requests for resources, eventually, the system will stop functioning and will need to be restarted manually.

Deadlock Prevention

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.

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.

3 No Preemption

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.

4 Circular Wait

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.

Deadlock Avoidance

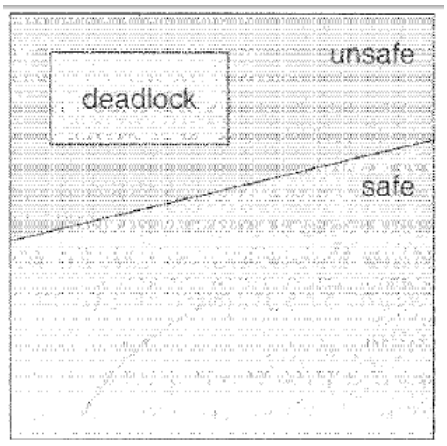
To avoid deadlock each process declare the *maximum number of* resources of each type that it may need. It is possible to construct an algorithm that ensures that the system will never enter a deadlocked state. It examines the resource-allocation state to ensure that a circular wait condition can never exist.

Safe State:

A state is *safe* if the system can allocate resources to each process in some order and still avoid a deadlock.

A sequence of processes is a safe sequence. It means if the resources that Pi needs are not immediately available, then Pi can wait until all *Pj(previous)* have finished. When they have finished, *Pi* can obtain all of its needed resources, complete its designated task, return its allocated resources to Pi+1 and so on.

If no such sequence exists, then the system state is said to be unsafe.



A deadlocked state is an unsafe state. Not all unsafe states are deadlocks, however. In an unsafe state, the operating system cannot prevent processes from requesting resources such that a deadlock occurs.

Ex: A system with 12 magnetic tape drives then at time t0 3 tapes are free.

Process	Maximum Needs	Current Helds at time t0
P0	10	5
P1	4	2
P2	9	2

The sequence < P1, Po, *P2* > satisfies the safety condition. A system can go to an unsafe state if sequence < P1, P2, *P1* >. P2 wait until process P1 finished and released its resources, It is an deadlock.

Avoidance algorithms

It ensure that the system "will never deadlock. The idea is that the system will always remain 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.

1) Resource-Allocation-Graph Algorithm

A claim edge Pi → Rj; Means Pi may request *Rj* at some time in the future, and represented by a dashed line.

A requested edge Pi→Rj; Means Pi requests *Rj* , at that time.

assignment edge *Rj* → Pi; means Rj hold by Pi.

When *Rj→Pi* is released then this edge is reconverted to a claim edge Pi → *Rj* • *B*efore process Pi starts executing, all its claim edges must already appear in the resource-allocation graph.

Suppose Pi→Rj. This 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.

we check for safety by using a cycle-detection algorithm. An algorithm for detecting a cycle in this graph requires an order of(N power 2) 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.

Ex: *P2* requests *R2 .shown in diagram.*

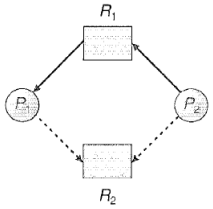


Figure Resource-allocation graph for deadlock avoidance.

Although R2 is currently free, we cannot allocate it to *P2,* since this action will create a cycle in the graph.

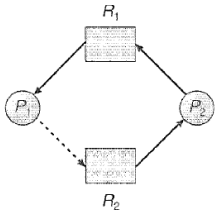


Figure An unsafe state in a resource-allocation graph.

A cycle indicates that the system is in an unsafe state. If *p1* requests *R2* , and *P2* requests *R1* then a deadlock will occur.

Limitation: The resource-allocation-graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type.

Banker's Algorithm to avoid deadlock

This algorithm is applicable to a resource allocation system with multiple instances of each resource type.

The name was chosen because the algorithm could be used in a banking system to ensure that the bank never

allocated its available cash in such a way that it could no longer satisfy the needs of all its customers.

Data structures used in the banker's algorithm.

- **Available. I**ndicates the number of available resources of each type.
- **Max**. It defines the maximum demand of each process.
- **Allocation**. It defines the number of resources of each type currently' allocated to each process.
- **Need. I**ndicates the remaining resource need of each process.

Thus according to this algorithm, when a new process enters the system, it must declare the maximum number of instances of each resource type that it may need and it not exceed the total number of resources available in the system. When a user requests a set of resources, the system must determine that system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.

Banker's has two algorithm-1) Safety 2) Resource Request

Safety Algorithm:

This algorithm finds whether or not a system is in a safe state. It is described as follows:

1. Initialize *Work = Available* and *Finish= false* for Current process.
2. Check *Finish ==false* and *Need <= Work.* If current process not exists, go to step 1.
3. *Work = Work + Allocation*
   Finish = true
   Go to step 2 for next process.
4. If *Finish =true* for all process then the system is in a safe state.

Resource-Request Algorithm :

This algorithm determines if requests can be safely granted.

When a request for resources is made by process *P,* the following actions are taken:

1. If *Request<= Need:,* go to step 2. Otherwise, raise an error condition since exceeded maximum claim.
2. If *Request<= Available,* go to step 3. Otherwise, process must wait, since the resources are not available.
3. In this step system is able to allocated the requested resources to process P by modifying following states:

Available = Available - Request
Allocation-, = Allocation + Request
Need = Need – Request

If the resulting resource-allocation state is safe, the transaction is completed,and process P; is allocated its resources. However, if the new state is unsafe, then *P,* must wait for *Request*, and the old resource-allocation state is restored.

Example: consider a system with five processes and three resource types *A, B,* and C. Resource type A has 10, *B* has 5 and *C* has 7 instances.
Suppose that, at time To, the following snapshot of the system has been taken:

	Allocation	*Max*	*Available*
	ABC	*A B C*	*ABC*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 02	
P_3	21 1	2 2 2	
P_4	0 0 2	4 3 3	

Get Need using Max - Allocation and is as follows:

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

1) Safty state:

Now we get safe state sequence of process by putting data structure of each process in bankers algorithm.

Initialize Finish =0 for all process P0, P1, P2, P3, P4 and Work =332

For P0

1. *Work = Available=332* and *Finish= false.*
2. Check *Finish ==false* and *743(need) <= 332(work).*

False . Move to next process.

For P1

2. Check *Finish ==false* and 112 <= *332. True*
3. *Work = Work + Allocation=332+200=532*

Finish = true

For P2

1. *Work = 532* and *Finish= false.*
2. Check *Finish ==false* and 600 <= *532.*

False . Move to next process

For P3

1. *Work = 532* and *Finish= false.*
2. Check *Finish ==false* and 011 <= *532. True*
3. *Work = 532+211=743*

Finish = true

For P4

1. *Work = 743* and *Finish= false.*
2. Check *Finish ==false* and 431 <= *743. True*
3. *Work = 743+002=745*

Finish = true

For P0

1. *Work = 745* and *Finish= false.*
2. Check *Finish ==false* and 743 <= *745.*
3. *Work = 745+010=755*

Finish = true

For P2

1. *Work = 755* and *Finish= false.*
2. Check *Finish ==false* and 600 <= *755. True*
3. *Work = 755+302=10,5,7*

Finish = true

Finally we observed that all process set finish value by true that means sequence <*P1,* P3, *P4, P0, P2*> satisfies the safety criteria. In this sequence deadlock never occurred.

2) Resource request allocation:

Suppose now that process *P1 made* requests (1,0,2). To decide this request can be immediately granted, we first check that *Request*(1,0,2) <*Need(112) and Request*(1,0,2) < *Available*(3,3,2)—

Both are true. It means this request has been fulfilled, and now we find safe stste for new request.

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

According to above manner we claim that the system is currently in a safe state. Indeed, the sequence

<*P1,* P3, *P4, P0, P2*> satisfies the safety criteria.

Deadlock Detection

If a system does not employ either a deadlock-prevention or a deadlock avoidance then a deadlock situation may occur. In this environment, the system must provide an algorithm that examines the state of the system to determine whether a deadlock has occurred called deadlock detection.

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 w*ait –for* graph. We obtain this

graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.

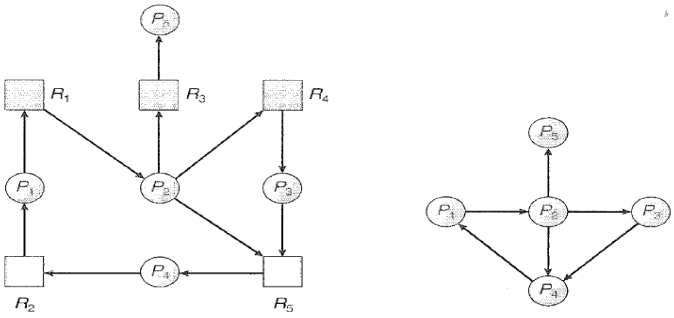


Figure (a) Resource-allocation graph. (b) Corresponding wait-for graph.

More precisely, an edge from Pi to *Pj* in a wait-for graph implies that process Pi is waiting for process *Pj* to release a resource that *Pi* needs. An edge *Pi* → *Pj* exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges Pi→ *Rq* and Rq→ *Pj* for some resource *Rq* .

As we know, a deadlock exists in the system if and only if the wait-for graph contains a cycle.

To detect deadlocks, the system needs to *maintain* the wait-for graph and periodically *invoke an algorithm* that searches for a cycle in the graph.

Recovery From Deadlock:

There are two ways for recovery from deadlock.
1 Process Termination
2 Resource Preemption

1 Process Termination: It is Inform to operator that a deadlock has occurred and he deal with deadlock manually by abort one or more processes to break the circular wait. To eliminate deadlocks by aborting a process, we use one of two methods.

a) **Abort all deadlocked processes:** This method clearly will break the deadlock cycle, but at great expense; the deadlocked processes may have computed for a long time.
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. Ex: If the process was in the midst of updating a filet terminating it will leave that file in an incorrect state. Therefore We should abort those processes whose termination will incur the minimum cost. Many factors may affect which process is chosen, including:
1. What the priority of the process is.
2. How long the process has computed.
3. How many and what type of resources the process has used.
4. How many more resources the process needs in order to complete
5. How many processes will need to be terminated

2 Resource Preemption:

Another possibility is that system *recover* from the deadlock automatically by preempt some resources from one or more of the deadlocked processes. To eliminate deadlocks using resource preemption, we successively preempt 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:

1. Selecting a victim:

Which resources and which processes are to be preempted?

2. Rollback.

If we preempt a resource from a process, we must roll back the process to some safe state and restart it from that state.

3. Starvation.

How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process?

UNIT-III
Main Memory

Memory consists of a large array of words or bytes, each with its own address. The CPU fetches instructions from memory. There are several issues for managing memory. They are-

1 Basic Hardware: Main memory is the storage that the CPU can access directly. If the data are not in memory, they must be moved there before the CPU can operate on them.

Each process has a separate memory space. We can provide this protection by using two registers, usually a base(holds the smallest legal physical memory address) and a limit(specifies the size of the range).

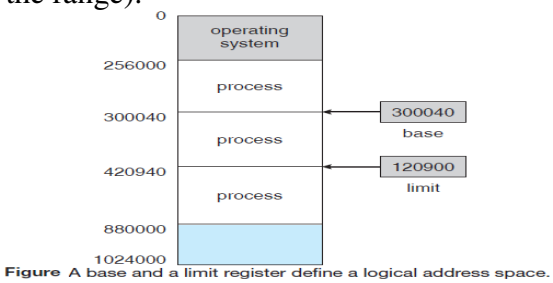


Figure A base and a limit register define a logical address space.

Protection of memory space is accomplished by having the CPU hardware compare *every* address generated in user mode with the registers.

2 Address Binding: The binding of instructions and data to memory addresses can be done at any step along the way:

• **Compile time**. If we know at compile time where the process will reside in memory, then **absolute code** can be generated. For example MS-DOS .COM-format programs are bound at compile time.

• **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.

• **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.

3 Logical(virtual) V/S Physical Address Space: An address generated by the CPU(At compile/load/execute time) 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 set of all logical addresses generated by a program is a **logical address space**; the set of all physical addresses corresponding to these logical addresses is a **physical address space**.

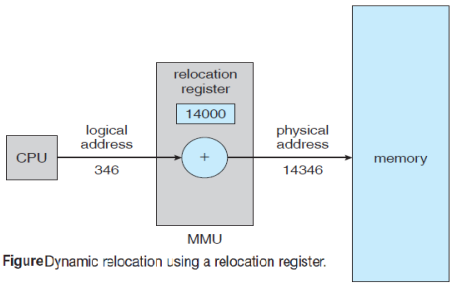


Figure Dynamic relocation using a relocation register.

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.

The user program deals with *logical* addresses. The memory-mapping hardware converts logical addresses into physical addresses. The concept of a *logical address space* that is bound to a separate *physical address space* is central to proper memory management.
Range: logical addresses (in the range 0 to *max*) and physical addresses (in the range $R + 0$ to $R + max$ for a base value R).

4 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. When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded. If it has not, the relocatable linking loader is called to load the desired routine into memory and to update the program's address table.

5 Dynamic Linking and Shared Libraries: Here linking of one sub routines to another is postponed until execution time.This feature is usually used with system libraries, such as language subroutine libraries. More than one program can share same library at a time.

Swapping

A process must 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.

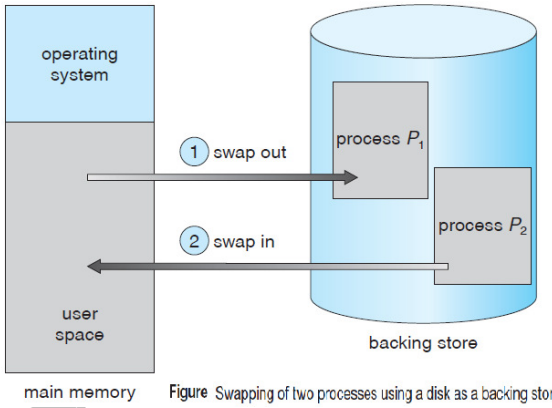


Figure Swapping of two processes using a disk as a backing store.

For example:

1) In a round-robin CPU-scheduling algorithm, when a quantum expires of any process then , the memory manager will start to swap out it and swap in another process to the memory space that has been next ready to execute.
2) In priority-based scheduling algorithms, if a higher-priority process arrives to execute then, the memory manager can swap out the lower-priority process and then 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 of swapping is sometimes called **roll out**, **roll in**.

Properties:

Normally, a process that is swapped out will be swapped back into the same memory space it occupied previously. In case of execution-time binding a process can be swapped into a different memory space, because the physical addresses are computed during execution time.

Disk features:

Swapping requires a backing store. It must be fast and large enough to accommodate copies of all memory images for all users, and it must provide direct access to these memory images.

Mechanism:

A **ready queue** consisting of all processes whose memory images are on the backing store or in memory and are ready to run. Whenever the CPU scheduler decides to execute a process, it calls the dispatcher. The dispatcher checks to see whether the next process in the queue is in memory. If it is not, and if there is no free memory region, the dispatcher swaps out a process currently in memory and swaps in the desired process.

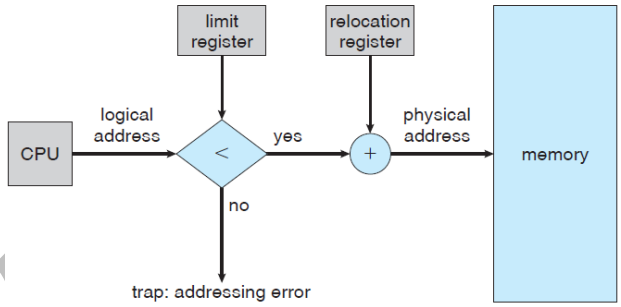
Contiguous Memory Allocation

The memory is usually divided into two partitions: one for the resident operating system and one for the user processes. Several user processes reside in memory at the same time. In **contiguous memory allocation**, each process is contained in a single contiguous section of memory.

It needs following things.

1 Memory Mapping and Protection

The relocation register contains the value of the smallest physical address; the limit register contains the range of logical addresses (for example, relocation = 100040 and limit = 74600). MMU maps the logical address *dynamically* by adding the value in the relocation register. This mapped address is sent to memory.



When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers. Every address generated by a CPU is checked against these registers. In this way we can protect both the operating system and other users' programs and data from being modified by this running process.

2 Memory Allocation

There are two way for memory allocation to any process.

1) Fixed-sized **partitions:**

   Divide memory into several fixed-sized **partitions.** Each partition may contain exactly one process. When a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process.

2) Variable-partition:

In this scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied. Initially, memory contains a set of holes (one large block of available memory) of various sizes. When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process. If the hole is too large, it is split into two parts. One part is allocated to the arriving process; the other is returned to the set of holes. When a process terminates, it releases its block of memory, which is then placed back in the set of holes.

There are three strategies most commonly used to select a free hole from the set of available holes.

• **First fit**. Allocate the *first* hole that is big enough. Searching can start either at the beginning or previous first-fit search ended .We can stop searching as soon as we find a free hole that is large enough.

• **Best fit**. Allocate the *smallest* hole that is big enough. We must search the entire list.

• **Worst fit**. Allocate the *largest* hole. Again, we must search the entire list.

It is observed that both first fit and best fit are better than worst fit in terms of decreasing time and storage utilization. Neither first fit nor best fit is clearly better than the other in terms of storage utilization, but first fit is generally faster.

3) Fragmentation:

There are two types- External and Internal fragmentation.

External fragmentation

exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous; storage is fragmented into a large number of small holes. If all such holes merge into in one big free block , we might be able to run more processes. No matter which algorithm is used, however, external fragmentation will be a problem. External fragmentation may be a minor or a major problem. Its solution is **compaction**. Place all free memory together in one large block.

Internal fragmentation:

Unused memory that is internal to a partition. Ex: a hole of 1000 bytes. One process requests 998 bytes. If we allocate exactly the requested block, then the overhead to keep track of 2 Byte hole will be larger than the hole itself. To avoiding this problem, allocate memory on block size. In this way memory allocated to a process may be slightly larger than the requested memory. Here 2 Bytes is **internal fragmentation**—unused memory that is internal to a partition.

Paging

Paging is a memory-management scheme that permits the physical address space of a process to be non contiguous. Paging avoids external fragmentation and the need for compaction.

Basic Concept

It involves breaking physical memory into fixed-sized blocks called **frames** and breaking logical memory into blocks of the same size called **pages**. When a process is to be executed, its pages are loaded into any available memory frames from their source (a file system or the backing store). The backing store is divided into fixed-sized blocks that are of the same size as the memory frames.

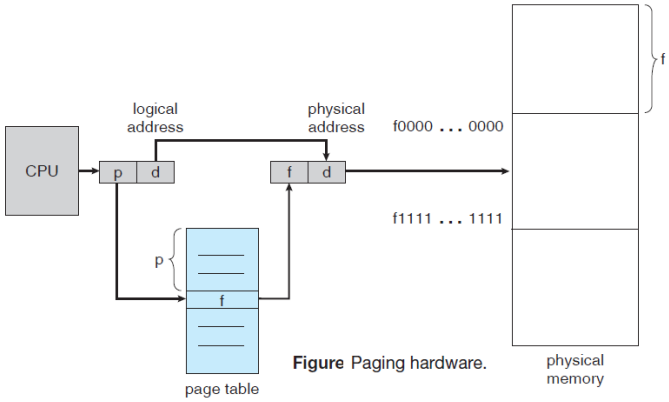


Figure Paging hardware.

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**(contains the base address of each page in physical memory). This base address(frame number) is combined with the page offset(logical address) to define the physical memory address.

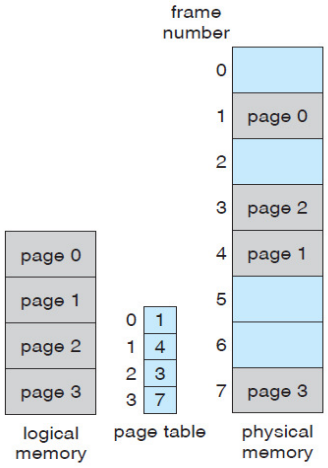


Figure Paging model of logical and physical memory.

Segmentation

Breaking program up into its logical segments, and allocating space for these segments into memory separately. The segments may be of variable length, and need not be allocated contiguously. This method called segmentation.

In this method memory is a collection of variable-sized segments, with no necessary ordering among segments.

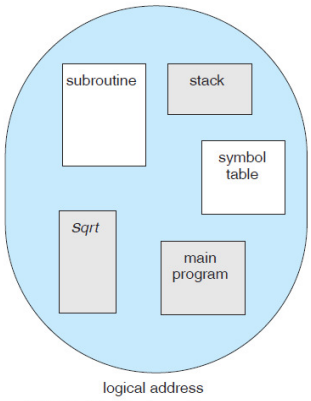


Figure User's view of a program.

Ex: a main program with a set of methods, procedures, or functions. It may also include various data structures : objects, arrays, stacks, variables, and so on. Each of these segments is of variable length.

Elements within a segment are identified by their offset from the beginning of the segment. Thus 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.

Thus each address specifies by user in two quantities: a segment name and an offset. (In paging scheme, address partitioned by the hardware into a page number and an offset). For implementation, segments are numbered rather than by a segment name.
Thus, a logical address consists of a *two tuple:* **<segment-number, offset>**

Normally, the user program is compiled, and the compiler automatically constructs segments.A C compiler might create separate segments for the following:
1. The code
2. Global variables
3. The heap, from which memory is allocated
4. The stacks used by each thread
5. The standard C library

Segmentation uses segment table. Each entry in 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, and the segment limit specifies the length of the segment.

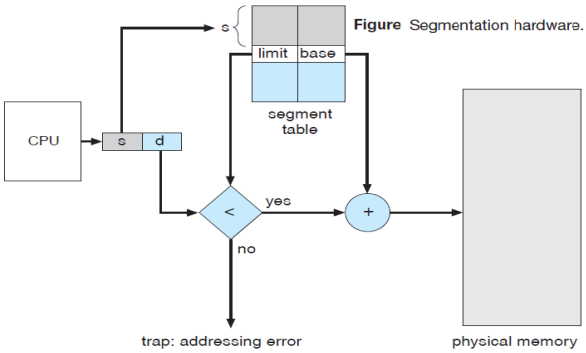


Figure Segmentation hardware.

A logical address consists of two parts: a segment number, *s,* and an offset into that segment, *d.*
The segment number is used as an index to the segment table. The offset *d* of the logical address must be between 0 and the segment limit. When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte.

Virtual Memory

Virtual memory is a technique that allows the execution of processes that are not completely in memory. One major advantage of this scheme is that programs can be larger than physical memory.

Background

Virtual memory is the separation of 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.

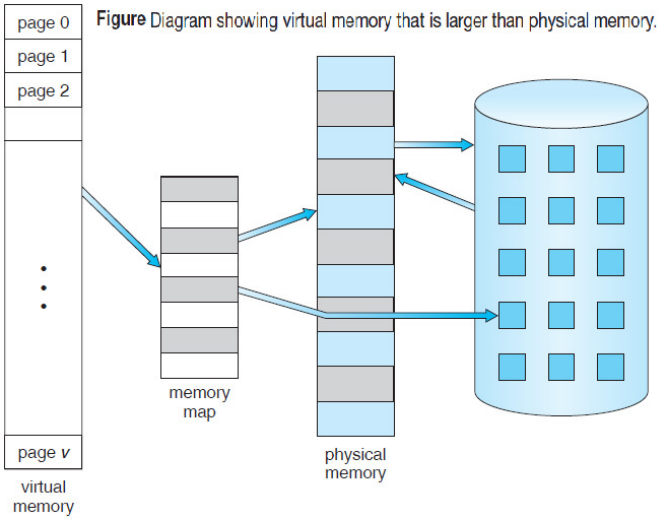


Figure Diagram showing virtual memory that is larger than physical memory.

Virtual memory makes the task of programming much easier, because the programmer no longer needs to worry about the amount of physical memory available; but can concentrate instead on the problem to be programmed.

Demand Paging

A demand-paging system is similar to a paging system with swapping.

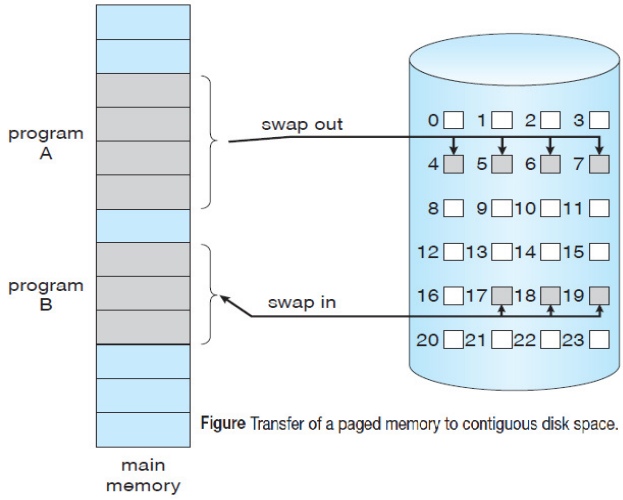


Figure Transfer of a paged memory to contiguous disk space.

Processes reside in secondary memory (usually a disk). 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 and we use term *pager,* rather than *swapper,* in connection with demand paging.

Basic Concepts: 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 pages into memory.

Page Fault: If the process tries to access a page that was not brought into memory then it is called a **page fault**.

pure demand paging: In the extreme case, we can 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 can execute with no more faults. This scheme is **pure demand paging**: never bring a page into memory until it is required.

Page Replacement

While a user process is executing, a page fault occurs. Operating system checks its internal tables to see that this is a page fault and not an illegal memory access. The operating system determines where the desired page is residing on the disk but then finds that there are *no* free frames on the free-frame list; all memory is in use.

Page replacement takes following approach.
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 frame to the disk; change the page and frame tables accordingly.
3. Read the desired page into the newly freed frame; change the page and frame tables.
4. Restart the user process.

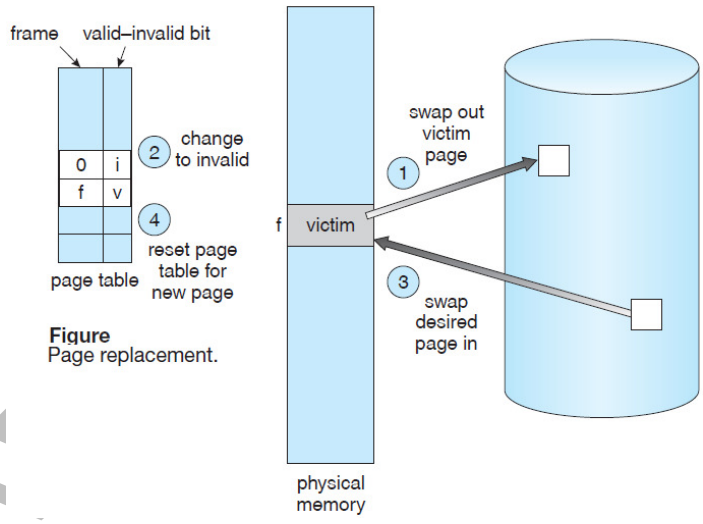


Figure
Page replacement.

Notice that, if no frames are free, *two* page transfers (one out and one in) are required. This situation effectively doubles the page-fault service time and increases the effective access time accordingly.

Page-replacement algorithms: There are many different page-replacement algorithms. Select a particular replacement algorithm which has 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.
For example, if we trace a particular process, we might record the following address sequence:
0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103, 0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105

At 100 bytes per page, this sequence is reduced to the following reference string: 1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1

To determine the number of page faults for a particular reference string and page-replacement algorithm, we also need to know the number of page frames available. Obviously, as the number of frames available increases, the number of page faults decreases.

Different algorithm are following:- 1)FIFO 2) Optimal 3) LRU etc.
In doing so, we use the reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 with 3 frames.

1)FIFO Page Replacement: 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.

We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue.

For 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 and so on.

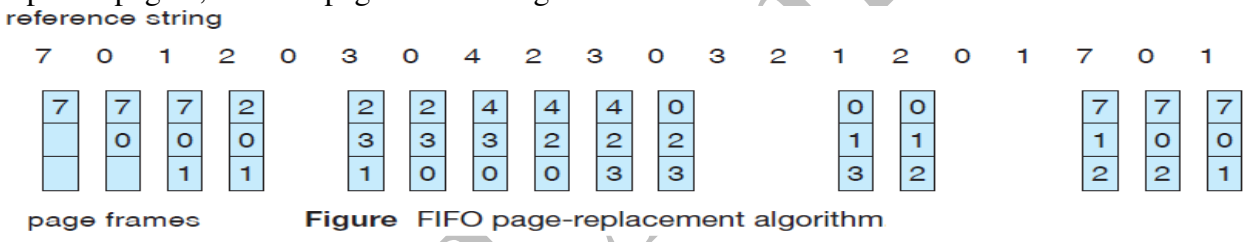


Figure FIFO page-replacement algorithm

Every time a fault occurs, we show which pages are in our three frames. There are 15 faults altogether.

2) Optimal Page Replacement:

An Optimal page-replacement algorithm, which has the lowest page-fault rate of all algorithms. It is simply this: "Replace the page that will not be used for the longest period of time.".

For example, on our sample reference string, the first three references cause faults that fill the three empty frames. The reference to page 2 replaces page 7, because page 7 will not be used until reference 18, and so on.

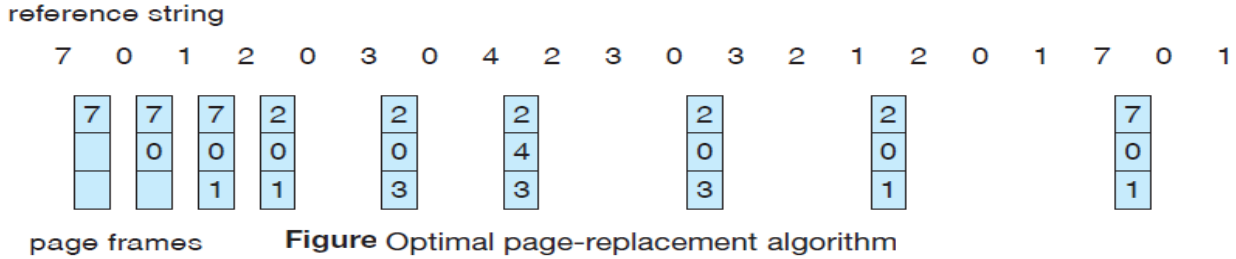


Figure Optimal page-replacement algorithm

Every time a fault occurs, we show which pages are in our three frames. There are 9 page faults altogether.

3) LRU Page Replacement:

we can replace the page that *has not been used* for the longest period of time. This approach is the **least-recently-used (LRU) algorithm**. LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses the page that has not been used for the longest period of time.

For example : for given reference string , the first five faults are the same as those for 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. Thus, the LRU algorithm replaces page 2, not knowing that page 2 is about to be used. When it then faults for page 2, the LRU algorithm replaces page 3, since it is now the least recently used of the three pages in memory.

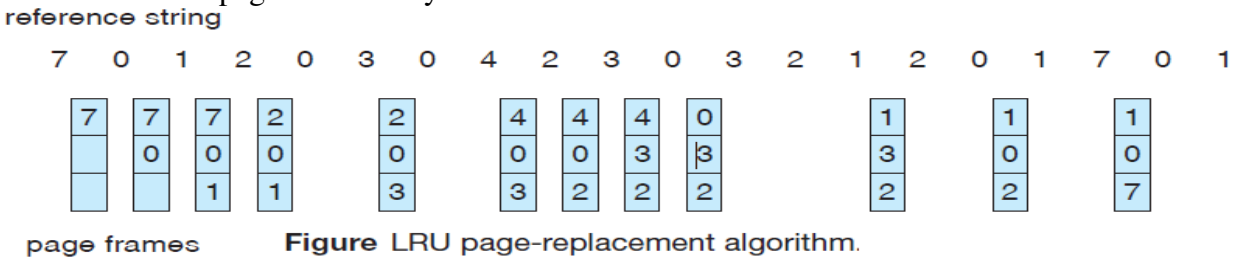


Figure LRU page-replacement algorithm.

Every time a fault occurs, we show which pages are in our three frames. There are 12 page faults altogether.

Allocation of Frames

The simplest case is the single-user system. Consider a single-user system with 128 KB of memory composed of pages 1 KB in size. This system has 128 frames. The operating system may take 35 KB, leaving 93 frames for the user process. Under pure demand paging, all 93 frames would initially be put on the free-frame list. When a user process started execution, it would generate a sequence of page faults. The first 93 page faults would all get free frames from the free-frame list.When the free-frame listwas exhausted, a page-replacement algorithmwould be used to select one of the 93 in-memory pages to be replaced with the 94th, and so on. When the process terminated, the 93 frames would once again be placed on the free-frame list.

1 Minimum Number of Frames

We must also allocate at least a minimum number of frames. As the number of frames allocated to each process decreases, the page-fault rate increases, slowing process execution.

2 Global Versus Local Allocation

Another important factor in the way frames are allocated to the various processes is page replacement. With multiple processes competing for frames, we can classify page-replacement algorithms into two broad categories: **global replacement** and **local replacement**.

Global replacement allows a process to select a replacement frame from the set of all frames, even if that frame is currently allocated to some other process; that is, one process can take a frame from another.

Local replacement requires that each process select from only its own set of allocated frames.

UNIT-IV

Mass-Storage Devices: Disk Structure

Magnetic disks and magnetic tapes are mass storage devices. A general overview of the physical structure of these mass-storage devices are following.

1 Magnetic Disks

Magnetic disks provide the bulk of mass storage for modern computer systems. Each disk **platter** has a flat circular shape, diameters range from1.8 to5.25 inches. The two surfaces of a platter are covered with a magnetic material. We store information by recording it magnetically on the platters.

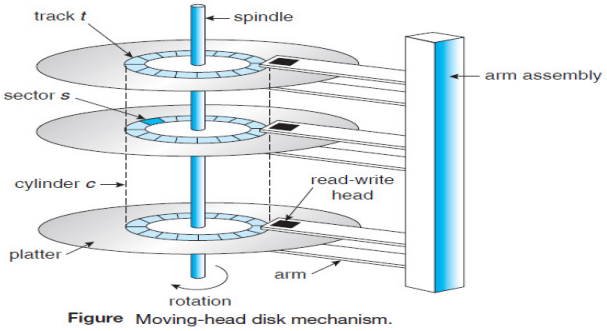


Figure Moving-head disk mechanism.

A read–write head "flies" just above each surface of every platter. The heads are attached to a **disk arm** that moves all the heads as a unit. The surface of a platter is logically divided into circular **tracks**, which are subdivided into **sectors**. The set of tracks that are at one arm position makes up a **cylinder**. There may be thousands of concentric cylinders in a disk drive, and each track may contain hundreds of sectors. The storage capacity of common disk drives is measured in gigabytes.

When the disk is in use, a drive motor spins it at high speed. Most drives rotate 60 to 200 times per second. Disk speed has two parts.

The **transfer rate** is the rate at which data flow between the drive and the computer.

The **positioning time**, sometimes called the **random-access time**, consists of the time necessary to move the disk arm to the desired cylinder, called the **seek time**, and the time necessary for the desired sector to rotate to the disk head, called the **rotational latency**.

Floppy disks is one example of inexpensive, removable magnetic disks that have a soft plastic case containing a flexible platter. The storage capacity of a floppy disk is typically only 1.44 MB or so.

Disk drives are addressed as large one-dimensional arrays of **logical blocks.** The size of a logical block is usually 512 bytes. Logical blocks is mapped onto the sectors. Sector 0 is the first sector of the first track on the outermost cylinder. The mapping proceeds from outermost to innermost cylinder. As we move, the number of sectors per track decreases and increases its rotation speed.

To keep the same rate of data moving the disk rotation speed can stay constant; in this case, the density of bits decreases from inner tracks to outer tracks to keep the data rate constant. This method is used in hard disks and is known as **constant angular velocity (CAV)**.

Disk Scheduling

Whenever a process needs I/O to or from the disk and if the desired disk drive available, the request can be serviced immediately. If the drive is busy, any new requests for service will be placed in the queue of pending requests for that drive.In this case the disk queue may often have several pending requests. Thus, when one request is completed, the operating system chooses which pending request to service next, it uses any one of several disk-scheduling algorithms.

1 FCFS Scheduling: The simplest form of disk scheduling is, the first-come, first-served (FCFS) algorithm. In this algorithm head move from current cylinder to first requested services and then next and so on.

Generally it does not provide the fastest service.

for example: Consider, a disk queue with requests for I/O to blocks on cylinders in following order

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

Let disk head is initially at cylinder 53. According to this algorithm it will first move from 53 to 98, then to 183, 37, 122, 14, 124, 65, and finally to 67. It can be express as

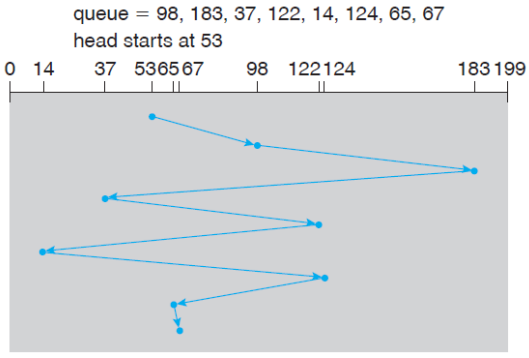


Figure FCFS disk scheduling.

total head movement = (98-53)+(183-98)+(37-183)+(122-37)+(14-122)+(124-14)+(65-124)+(67-65)

= 45+85+146+85+108+110+59+2

= 640 cylinders.

Problem with scheduling: In this example, If the requests for cylinders 37 and 14 could be serviced together, before or after the requests for 122 and 124, the total head movement could be decreased substantially, and performance could be thereby improved.

2 SSTF(shortest-seek-time-first) Scheduling algorithm.

The SSTF algorithm selects the request with the least seek time from the current head position. Since seek time increases with the number of cylinders traversed by the head, so SSTF chooses the pending request closest to the current head position.

for example: Consider, a disk queue with requests for I/O to blocks on cylinders in following order

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

Let disk head is initially at cylinder 53.

The closest request to the initial head position (53) is at cylinder 65, the next closest request is at cylinder 67. From there, the request at cylinder 37 is closer, Continuing, we service the request at cylinder 14, then 98, 122, 124, and finally 183.

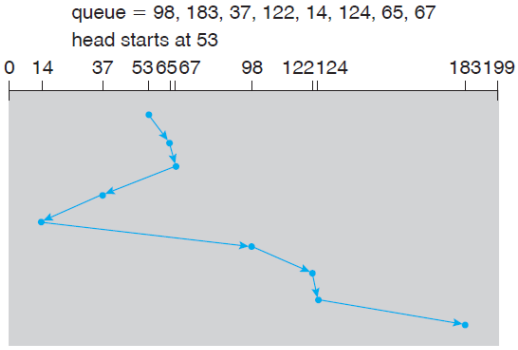


Figure SSTF disk scheduling.

Total head movement= (53-65)+(65-67)+(67-37)+(37-14)+(14-98)+(98-122)+(122-124)+(124-183)

= 12+2+30+23+84+24+2+59

= 236 cylinders—

It is less than FCFS scheduling. Clearly, this algorithm gives a substantial improvement in performance.

Problem: SSTF is a form of SJF) scheduling; and like SJF, it may cause starvation of some requests.

---

3 SCAN Scheduling(elevator Algorithm)

In the **SCAN algorithm**, 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.

for example: Consider, a disk queue with requests for I/O to blocks on cylinders in following order

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

Assuming that the disk arm is moving toward 0 and that the initial head position is 53, the head will next 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 .

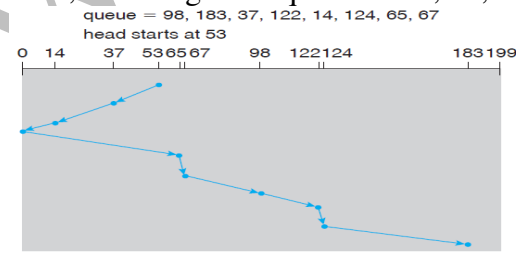


Figure SCAN disk scheduling.

Total head movement= (53-37)+(37-14)+(14-0)+(0-65)+(65-67)+(67-98)+(98-122)+(122-124)+(124-183)

= 16+23+14+65+2+31+24+2+59

= 236 cylinders—

Problem: 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.

---

4 Circular SCAN (C-SCAN) scheduling:

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. It treats the cylinders as a circular list that wraps around from the final cylinder to the first one.

for example: Consider, a disk queue with requests for I/O to blocks on cylinders in following order

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

Assuming initial head position is 53

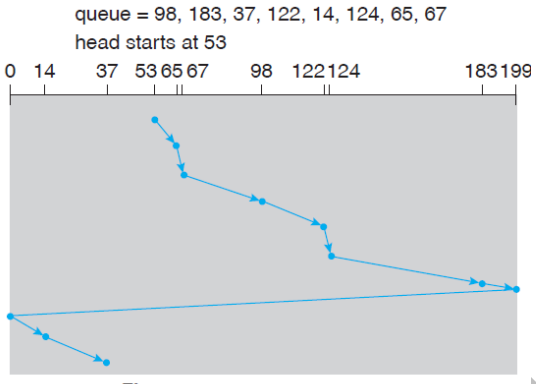


Figure C-SCAN disk scheduling.

5 LOOK Scheduling

The arm goes only as far as the final request in each direction. Then, it reverses direction immediately,without going all the way to the end of the disk. Versions of SCAN and C-SCAN that follow this pattern are called **LOOK** and **C-LOOK scheduling**, because they *look* for a request before continuing to move in a given direction.

for example: Consider, a disk queue with requests for I/O to blocks on cylinders in following order

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

Assuming initial head position is 53

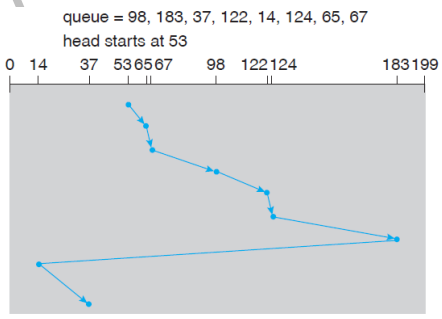


Figure C-LOOK disk scheduling

Selection of a Disk-Scheduling Algorithm

SSTF is common and has a natural appeal because it increases performance over FCFS. SCAN and C-SCAN perform better for systems that place a heavy load on the disk, because they are less likely to cause a starvation problem.

Disk Management

The operating system is responsible for several other aspects of disk management. They are-

1) Disk Formatting 2) Boot Block 3) Bad Block

1 Disk Formatting:

A new magnetic disk is a blank slate: it is just a platter of a magnetic recording material. Before a disk can store data, it must be divided into sectors that the disk controller can read and write. This process is called l**ow-level formatting**, or **physical formatting**.

Low-level formatting fills the disk with a special data structure for each sector. The data structure for a sector typically consists of a header, a data area (usually 512 bytes in size), and a trailer. The header and trailer contain information used by the disk controller, such as a sector number and an **error-correcting code (ECC)**.

When the controller writes a sector of data during normal I/O, the ECC is updatedwith a value calculated from all the bytes in the data area. When the sector is read, the ECC is recalculated and compared

with the stored value. If the stored and calculated numbers are different, this mismatch indicates that the data area of the sector has become corrupted and that the disk sector may be bad.

2 Boot Block

For a computer to start running(powered up or rebooted) it must have an initial *bootstrap* program to run. Bootstrap program finds the operating system kernel on disk, loads that kernel into memory, and jumps to an initial address to begin the operating-system execution.

A tiny Bootstrap is stored in **read-only memory (ROM)** because processor can start executing when powered up or reset. And, since ROM is read only, it cannot be infected by a computer virus. The Full bootstrap program is stored in the "boot blocks" at a fixed location on the disk. A disk that has a boot partition is called a **boot disk** or **system disk**. The code in the boot ROM instructs the disk controller to read the boot blocks into memory and then starts executing that code. The full bootstrap program is more sophisticated than the bootstrap loader in the boot ROM; it is able to load the entire operating system from a non-fixed location on disk and to start the operating system running.

3 Bad Blocks

It means one or more sectors become defective. Most disks even come from the factory with **bad blocks**. Depending on the disk and controller in use, these blocks are handled in a variety of ways.

A typical bad-sector transaction might be as follows:

• The operating system tries to read logical block 87.

• The controller calculates the ECC and finds that the sector is bad. It reports this finding to the operating system.

• The next time the system is rebooted, a special command is run to tell the SCSI controller to replace the bad sector with a spare.

• After that, whenever the system requests logical block 87, the request is translated into the replacement sector's address by the controller.

Swap-Space Management

Swapiing means moving entire processes between disk and main memory. Swapping occurs when physical memory reaches a critically low point and processes are moved from memory to swap space to free available memory. The terms *swapping* and *paging uses* interchangeably.

Swap-space management is a low-level task of the operating system. Virtual memory uses disk space as an extension of main memory. Since disk access is much slower than memory access, using swap space
significantly decreases system performance. It needs following things.

1) how swap space is used
2) where swap space is located on disk, and
3) how swap space is managed.

1 How Swap-Space Used

Systems that implement swapping may use swap space to hold an entire process image, including the code and data segments and system needs varius memory-management algorithms to use swap-space. Paging systems may simply store pages that have been pushed out of main memory. The amount of swap space needed on a system can therefore vary from a few mega bytes of disk space to gigabytes, depending on the amount of physical memory.

2 Swap-Space Location

A swap space can reside in one of two places:

1) it can be carved out of the normal file system or
2) it can be in a separate disk partition.

If the swap space is simply a large file within the file system, normal file-system routines can be used to create it, name it, and allocate its space. This approach, though easy to implement, but is inefficient. Navigating the directory structure and External fragmentation can greatly increase swapping times.

In case of separate disk partition, a separate swap-space storage manager is used to allocate and deallocate the blocks from the disk partition.

3 Swap-Space Management: An Example

The traditional UNIX kernel started with an implementation of swapping that copied entire processes between contiguous disk regions and memory. UNIX later evolved to a combination of swapping and paging as paging hardware became available.

Ending unit-4

UNIT-5

System software

System software is computer software designed to operate the computer hardware and to provide a platform for running application software.

The most basic types of system software are:

- The computer BIOS and device firmware, provides basic functionality to operate and control the hardware connected to or built into the computer.
- The operating system (prominent examples being Microsoft Windows, Mac OS X and Linux), which allows the parts of a computer to work together by performing tasks like transferring data between memory and disks or rendering output onto a display device. It also provides a platform to run high-level system software and application software.
- Utility software, which helps to analyze, configure, optimize and maintain the computer.
- compiler, linker or debugger used to designate software development tools

Application software

In contrast to system software, software that allows users to do things like create text documents, play games, listen to music, or surf the web is called application software. Thus **Application software**, is computer software designed to help the user to perform specific tasks.

Examples include enterprise software, accounting software, office suites, graphics software and media players.

The system software serves the application, which in turn serves the user.

==========

ASSEMBLERS

Assembler is system software which is used to convert an assembly language program to its equivalent object code. The input to the assembler is a source code written in assembly language (using mnemonics) and the output is the object code. The design of an assembler depends upon the machine architecture as the language used is mnemonic language.

BASIC ASSEMBLER FUNCTIONS: The basic assembler functions are:

1) Translating mnemonic language code to its equivalent object code.
2) Assigning machine addresses to symbolic labels.

The design of assembler:

The design of assembler can be to perform the following:

1) Scanning (tokenizing)
2) Parsing (validating the instructions)
3) Creating the symbol table
4) Resolving the forward references
5) Converting into the machine language

It also includes-

□□□Convert mnemonic operation codes to their machine language equivalents

- Convert symbolic operands to their equivalent machine addresses
- Decide the proper instruction format Convert the data constants to internal machine representations
- Write the object program and the assembly listing

assembler directives

these do not generate the object code but directs the assembler to perform certain operation. These directives are:

START: Specify name & starting address.
END: End of the program, specify the first execution instruction.
BYTE, WORD, RESB, RESW
End of record: a null char(00) End of file: a zero length record

The assembler design can be done in two ways:

1. Single pass assembler
2. Multi-pass assembler

Single-pass Assembler: In this case the whole process of scanning, parsing, and object code conversion is done in single pass. The only problem with this method is resolving forward reference.

Two-Pass assembler: (multi-pass assembler) It resolves the forward references. The process of the multi-pass assembler can be as follows:

Pass-1

a) Assign addresses to all the statements
b) Save the addresses assigned to all labels to be used in *Pass-2*
c) Perform some processing of assembler directives such as RESW, RESB to find the length of data areas for assigning the address values.
d) Defines the symbols in the symbol table(generate the symbol table)

Pass-2

a) Assemble the instructions (translating operation codes and looking up addresses).
b) Generate data values defined by BYTE, WORD etc.
c) Perform the processing of the assembler directives not done during *pass-1*.
d) Write the object program and assembler listing.

LOADERS AND LINKERS

The Source Program written in assembly language or high level language will be converted to object program, which is in the machine language form for execution. This conversion either from assembler or from compiler, contains translated instructions and data values from the source program, or specifies addresses in primary memory where these items are to be loaded for execution. This contains the following three processes, and they are,

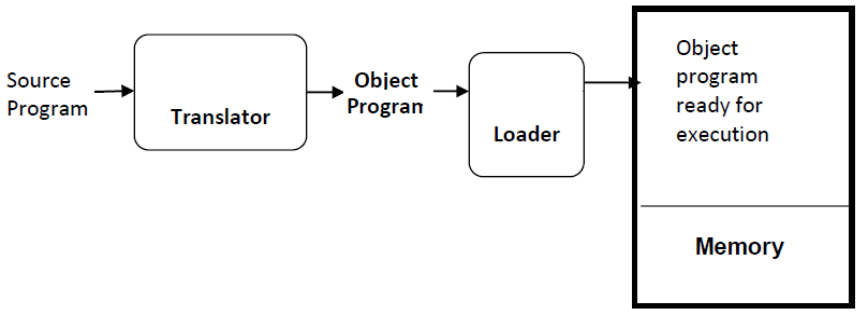
Loading - which allocates memory location and brings the object program into memory for execution - (Loader)

Linking- which combines two or more separate object programs and supplies the information needed to allow references between them - (Linker)

Relocation - which modifies the object program so that it can be loaded at an address different from the location originally specified - (Linking Loader)

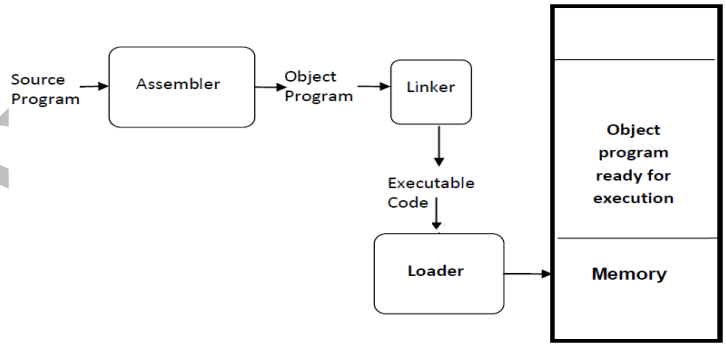
Basic Loader Functions

A loader is a system program that performs the loading function. It brings object program into memory and starts its execution. The role of loader is as shown in the follwing figure-



In figure translator may be assembler/complier, which generates the object program and later loaded to the memory by the loader for execution.

Role of both loader and linker:



Type of Loaders

The different types of loaders are, absolute loader, bootstrap loader, relocating loader (relative loader), and, direct linking loader.

1) **Absolute Loader** The object code is loaded to specified locations in the memory. At the end the loader jumps to the specified address to begin execution of the loaded program.

2) **A Bootstrap Loader** When a computer is first turned on or restarted, a special type of absolute loader, called bootstrap loader is executed. This bootstrap loads the first program to be run by the computer -- usually an operating system.

3) **Relocating loader** The object program is loaded into memory wherever there is room for it. The actual starting address of the object program is not known until load time.Loaders that allow for program relocation are called relocating loaders or relative loaders.

4) **Direct Linking Loader** Linking Loader uses two-passes logic.

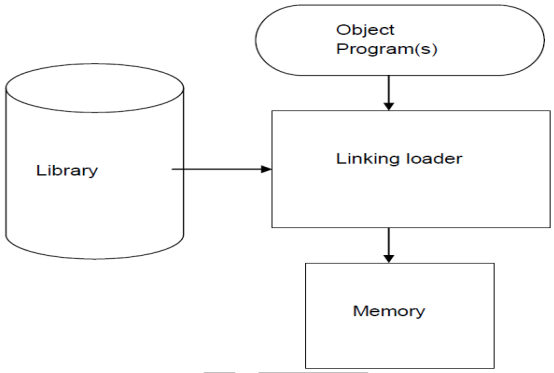
Pass 1: Assign addresses to all external symbols

Pass 2: Perform the actual loading, relocation, and linking.

Linking Options

There are some common alternatives for organizing the loading functions, including relocation and linking are-

Linking Loaders – Perform all linking and relocation at load time.



The above diagram shows the processing of an object program using Linking Loader. The source program is first assembled or compiled, producing an object program. A linking loader performs all linking and loading operations, and loads the program into memory for execution.

Linkage editors - A linkage editor produces a load module or an executable image – which is written to a file or library for later execution. Some useful functions of Linkage editor are, an absolute object program can be created, if starting address is already known.

Dynamic linking, in which linking function is performed at execution time. A subroutine is loaded and linked to the rest of the program when it is first called – usually called dynamic linking, dynamic loading or load on call. The advantages of dynamic linking are, it allow several executing programs to share one copy of a subroutine or library.

Relocation

It modifies the object program so that it can be loaded at an address different from the location originally specified .**Two different techniques used for relocation.**

1) Modification record method
2) Relocation bit method.

Relocation bit method. If the relocation bit corresponding to a word of object code is set to 1, the program"s starting address is to be added to this word when the program is relocated. Bit value 0 indicates no modification is required.

bit mask. The relocation bits are gathered together following the length indicator in each text record and which is called as bit mask. For e.g. the bit mask FFC (111111111100) specifies that the first 10 words of object code are to be modified during relocation.

MACRO PROCESSORS

A *Macro* represents a group of statements in the source programming language that allows the programmer to write shorthand version of a program. The macro processor replaces each macro instruction with the corresponding group of source language statements (*expanding*). The design of a macro processor generally is *machine independent.*

Macro processor is system software that replaces each macroinstruction with the corresponding group of source language statements. This is also called as expanding of macros.

Macro expansion statements give the name of the macro instruction being invoked and the arguments to be used in expanding the macros. These statements are also known as macro call.

Directives used in macro definition is MACRO - it identifies the beginning of the macro definition and MEND - it marks the end of the macro definition

Data structures used in macro processor:
DEFTAB –it contains a macro prototype and the statements that make up the macro body.
NAMTAB – it is used to store the macro names and it contains two pointers for each macro instruction which indicate the starting and end location of macro definition
ARGTAB – it is used to store the arguments during the expansion of macro invocations.

Conditional macro expansion If the macro is expanded depends upon some conditions in macro definition then it is called as conditional macro expansion. Sfollowing statements used for conditional macro expansion.
IF-ELSE-ENDIF statement
WHILE-ENDW statement

For example: Consider the macro definition
#define DISPLAY (EXPR) Printf ("EXPR = %d\n", EXPR)
Expand the macro instruction DISPLAY (ANS) by Printf ("EXPR = %d\n", ANS)

Nested macro call: The statement, in which a macro calls on another macro, is called nested macro all. In the nested macro call, the call is done by outer macro and the macro called is the inner macro.
Two passes of macro: Pass1: processing of definitions. Pass 2:actual-macro expansion.

Line by line processor: This macro processor reads the source program statements, process the statements and then the output lines are passed to the language translators as they are generated, instead of being written in an expanded file. Its **advantages are-** It avoids the extra pass over the source program during assembling
and it may use some of the utility that can be used by language translators so that can be loaded once.

General-purpose macro processors: The macro processors that are not dependent on any particular programming language, but can be used with a variety of different languages are

known as general purpose macro processors. Eg. The ELENA macro processor. Its **advantages are -** programmer does not need to learn about a macro facility for each compiler and overall saving in software development cost and maintenance cost.

Execution of nested macro: The execution of nested macro call follows the LIFO rule. In case of nested macro calls the expansion of the latest macro call is completed first.

The tasks involved in macro expansion:

- identify the macro calls in the program
- the values of formal parameters are identified
- maintain the values of expansion time variables declared in a macro
- expansion time control flow is organized
- determining the values of sequencing symbols
- expansion of a model statement is performed

Compiler

A **compiler** is a computer program (or set of programs) that transforms source code written in a programming language (the *source language*) into another computer language (the *target language*, often having a binary form known as *object code*).

The name "compiler" is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly language or machine code).

If the compiled program can run on a computer whose CPU or operating system is different from the one on which the compiler runs, the compiler is known as a cross-compiler.

A program that translates from a low level language to a higher level one is a *decompiler*.

A compiler operations: lexical analysis, preprocessing, parsing, semantic analysis (Syntax-directed translation), code generation, and code optimization.

machine-independent Before the development of FORTRAN (FORmula TRANslator), the first higher-level language, in the 1950s, machine-dependent assembly language was widely used. With the advance of high-level programming languages soon followed after FORTRAN, such as COBOL, C, BASIC, programmers can write machine-independent source programs.

The structure of a compiler

A compiler requires-

1) determining the correctness of the syntax of programs,
2) generating correct and efficient object code,
3) run-time organization, and
4) formatting output according to assembler and/or linker conventions.

A compiler consists of three main parts: the frontend, the middle-end, and the backend.
The **front end** checks whether the program is correctly written in terms of the programming language syntax and semantics.
The **middle end** is where optimization takes place.
The **back end** is responsible for translating the IR from the middle-end into assembly code.

Compiler output
One classification of compilers is by the platform on which their generated code executes. This is known as the *target platform.*
A *native* or *hosted* compiler is one which output is intended to directly run on the same type of computer and operating system that the compiler itself runs on.
The output of a cross compiler is designed to run on a different platform. Cross compilers are often used when developing software for embedded systems that are not intended to support a software development environment.
The output of a compiler that produces code for a virtual machine (VM) may or may not be executed on the same platform as the compiler that produced it. For this reason such compilers are not usually classified as native or cross compilers.

One-pass versus multi-pass compilers a single pass it simplifies the job of writing a compiler and one-pass compilers generally perform compilations faster than multi-pass compilers. Thus, partly driven by the resource limitations of early systems, many early languages were specifically designed so that they could be compiled in a single pass (e.g., Pascal).

The disadvantage of compiling in a single pass is that it is not possible to perform many of the sophisticated optimizations needed to generate high quality code.

While the typical multi-pass compiler outputs machine code from its final pass, there are several other types:

- A "source-to-source compiler"
- Stage compiler
- Just-in-time compiler,
- Common Intermediate Language (CIL)

Cross compiler

A **cross compiler** is a compiler capable of creating executable code for a platform other than the one on which the compiler is run. Cross compiler tools are used to generate executables for embedded system or multiple platforms. It is used to compile for a platform upon which it is not feasible to do the compiling, like microcontrollers that don't support an operating system. It has become more common to use this tool for paravirtualization where a system may have one or more platforms in use.

Uses of cross compilers

The fundamental use of a cross compiler is to separate the build environment from target environment. This is useful in a number of situations:

- Embedded computers where a device has extremely limited resources. For example, a microwave oven.
- Compiling for multiple machines. to support several different versions of an operating system or to support several different operating systems.
- Compiling on a server farm. involves many compile operations can be executed across any machine
- Bootstrapping to a new platform.
- Compiling native code for emulators

Canadian Cross: The Canadian Cross is a technique for building cross compilers for other machines. Given three machines A, B, and C, one uses machine A to build a cross compiler that runs on machine B to create executables for machine C.

GCC and cross compilation: GCC, a free software collection of compilers, can be set up to cross compile. It supports many platforms and languages.Cross compiling GCC requires that a portion of the *target platform*'s C standard library be available on the *host platform*.

Manx Aztec C cross compilers: Manx's Aztec C86, their native mode 8086 MS DOS compiler, was also a cross compiler. It created binary executables for then-legacy operating systems for the 16 bit 8086 family of processors.

Microsoft C cross compilers: The first Microsoft C Compilers were made by the same company who made Lattice C and were rebranded by Microsoft as their own, until MSC 4 was released, which was the first version that Microsoft produced themselves.

.NET and beyond: The .NET Framework runtime and CLR provide a mapping layer to the core routines for the processor and the devices on the target computer. The command-line C compiler in Visual Studio will compile native code for a variety of processors and can be used to build the core routines themselves.

==========