CHAPTER 1

INTRODUCTION

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.*

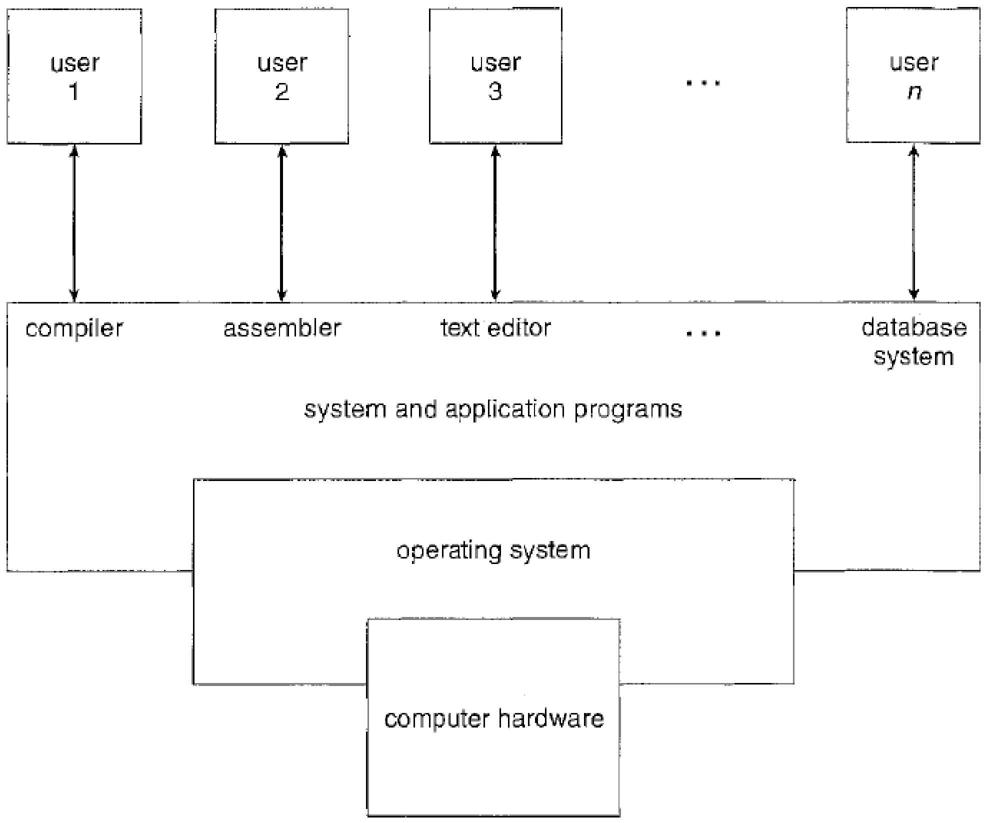


Fig: Abstract view of the components of a computer system.

The hardware – CPU (Central Processing Unit), memory, 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. The operating system controls the hardware and coordinates its use among the various application programs for the various users.

User View

The user's view of the computer varies according to the interface being used. Most computer users sit in front of a PC, consisting of a monitor/keyboard/ mouse, and system unit. Such a system is designed for one user to monopolize its resources. The goal is to maximize the work (or play) that the user is performing. In this case/ the operating system is designed mostly for ease of use, with some attention paid to performance and none paid to resource utilization – how various hardware and software resources are shared. Performance is, of course, important to the user; but such systems are optimized for the single-user experience rather than the requirements of multiple users.

In other cases, a user sits at a terminal connected to a mainframe or a mini computer. Other users are accessing the same computer through other terminals. These users share resources and may exchange information. The operating system in such cases is designed to maximize resource utilization to assure that all available CPU time, memory, and I/0 are used efficiently and that no individual user takes more than her fair share.
In still other cases, users sit at workstations connected to networks of other workstations and servers. These users have dedicated resources at their disposal, but they also share resources such as networking and servers-file, compute, and print servers. Therefore, their operating system is designed to compromise between individual usability and resource utilization.
Recently, many varieties of handheld computers have come into fashion. Most of these devices are standalone units for individual users. Some are connected to networks, either directly by wire or (more often) through wireless modems and networking. Because of power, speed, and interface limitations, they perform relatively few remote operations. Their operating systems are designed mostly for individual usability, but performance per unit of battery life is important as well.
Some computers have little or no user view. For example, embedded computers in home devices and automobiles may have numeric keypads and may turn indicator lights on or off to show status, but they and their operating systems are designed primarily to run without user intervention.

System View
From the computer's point of view, the operating system is the program most intimately involved with the hardware. In this context, we can view an operating system as a resource allocator. A computer system has many resources that may be required to solve a problem: CPU time, memory space, file-storage space, I/0 devices, and so on. The operating system acts as the manager of these resources. Facing numerous and possibly conflicting requests for resources, the operating system must decide how to allocate them to specific
programs and users so that it can operate the computer system efficiently and fairly. As we have seen, resource allocation is especially important where many users access the same mainframe or minicomputer. An operating system is a control program. A control program manages the execution of user programs to prevent errors and improper use of the computer. It is especially concerned with the operation and control of I/O devices.

Defining Operating Systems
In general, we have no completely adequate definition of an operating system. Operating systems exist because they offer a reasonable way to solve the problem of creating a usable computing system. The fundamental goal of computer systems is to execute user programs and to make solving user problems easier. Toward this goal, computer hardware is constructed. Since bare hardware alone is not particularly easy to use, application programs are developed. These programs require certain common operations, such as those controlling the I/0 devices. The common functions of controlling and allocating resources are then brought together into one piece of software: the operating system.

Computer System Organization
Before we can explore the details of how computer systems operate, we need general knowledge of the structure of a computer system. In this section, we look at several parts of this structure. The section is mostly concerned with computer-system organization.

Computer-System Operation
A modern general-purpose computer system consists of one or more CPUs and a number of device controllers connected through a common bus that provides access to shared memory. Each device controller is in charge of a specific type of device (for example, disk drives, audio devices, and video displays). The CPU and the device controllers can execute concurrently, competing for memory cycles. To ensure orderly access to the shared memory, a memory controller is provided whose function is to synchronize access to the memory.
For a computer to start running-for instance, when it is powered up or rebooted-it needs to have an initial program to run. This initial program, or bootstrap program, tends to be simple. Typically, it is stored in read-only memory or electrically erasable programmable read-only memory, known by the general term firmware.

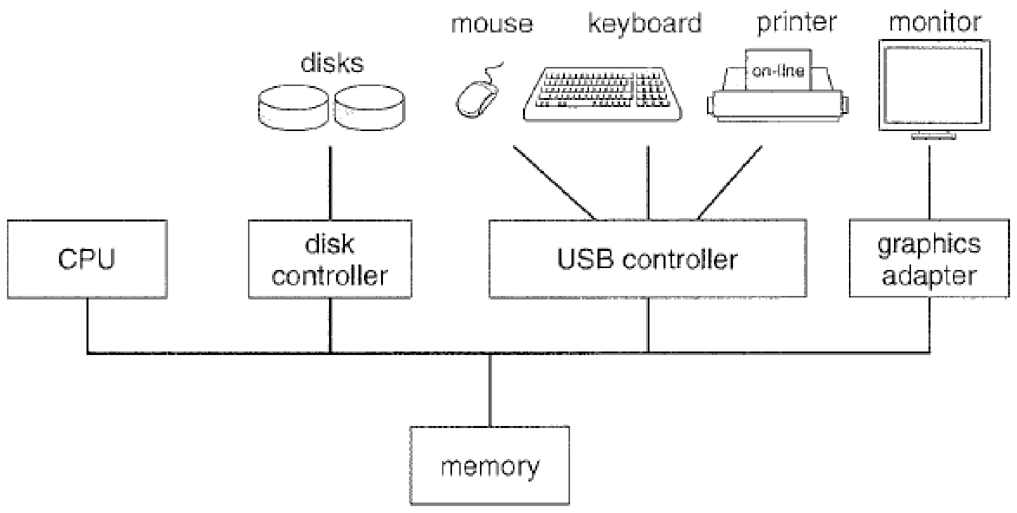


Fig: A modern computer system

The operating system then starts executing the first process, such as "init," and waits for some event to occur. The occurrence of an event is usually signaled by an interrupt from either the hardware or the software. Hardware may trigger an interrupt at any time by sending a signal to the CPU, usually by way of the system bus. Software may trigger an interrupt by executing a special operation called a system call.
When the CPU is interrupted, it stops what it is doing and immediately transfers execution to a fixed location. The fixed location usually contains the starting address where the service routine for the interrupt is located. The interrupt service routine executes; on completion, the CPU resumes the interrupted computation.

Will be updated soon..

CHAPTER 3

Synchronization and deadlocks

Background

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

To guard against the race condition, we need to ensure that only one process at a time can be manipulating the shared variable. To make such a guarantee, we require that the processes be synchronized in some way.

Critical Section Problem

Consider a system consisting of *n* processes $\{P_o, P_1 \dots P_{n-1}\}$. Each process has a segment of code, called a Critical Section, in which the process may be changing common variables, updating a table, writing a file, and so on. The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time. The *critical-section problem* is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this request is the entry section. The critical section may be followed by an exit section. The remaining code is the Remainder section. The general structure of a typical process P_i is shown in following figure:

```
do {
              entry section

              critical section

              exit section

              remainder section
  } while (TRUE);
```

Figure. General structure of a typical process P_i

The entry section and exit section are enclosed in boxes to highlight these important segments of code.
A solution to the critical-section problem must satisfy the following three requirements:
1. **Mutual exclusion.** If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress.** If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.
3. **Bounded waiting.** There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.
We assume that each process is executing at a nonzero speed. However, we can make no assumption concerning the relative speed of the *n* processes. At a given point in time, many kernel-mode processes may be active in the operating system. As a result, the code implementing an operating system is subject to several possible race conditions. Two general approaches are used to handle critical sections in operating systems:
(1) **preemptive kernels** and (2) **non-preemptive kernels.**
A preemptive kernel allows a process to be preempted while it is running in kernel mode.

A non-preemptive kernel does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU.

Semaphores

The hardware-based solutions to the critical-section problem are complicated for application programmers to use. To overcome this difficulty, we can use a synchronization tool called a semaphores. A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait () and signal ().
The wait () operation was originally termed P (from the Dutch *proberen,* "to test"); signal() was originally called V (from *verhogen,* "to increment"). The definition of wait () is as follows:

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

The definition of signal() is as follows:

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

All modifications to the integer value of the semaphore in the wait () and signal() operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.
Usage: Operating systems often distinguish between counting and binary semaphores. The value of a counting semaphore can range over an unrestricted domain. The value of a binary semaphore can range only between 0 and 1. On some systems, binary semaphores are known as mutex locks, as they are locks that provide mutual exclusion.
We can use binary semaphores to deal with the critical-section problem for multiple processes. Counting semaphores can be used to control access to a given resource consisting of a finite number o£ instances.
Implementation: The main disadvantage of the semaphore definition given here is that it requires busy waiting. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code. This continual looping is clearly a problem in a real multiprogramming system, where a single CPU is shared among many processes. Busy waiting wastes CPU cycles that some other process might be able to use productively. This type of semaphore is also called a spinlock because the process "spins" while waiting for the lock.

```
do {
    wait (mutex);
      II critical section
    signal(mutex);
  II remainder section
} while (TRUE);
```

Figure. Mutual-exclusion implementation with semaphores

Deadlocks

System Model

A system consists of a finite number of resources to be distributed among a number of competing processes. The resources are partitioned into several types, each consisting of some number of identical instances. Memory space, CPU cycles, files, and I/0 devices (such as printers and DVD drives) are examples of resource types. If a system has two CPUs, then the resource type *CPU* has two instances. Similarly, the resource type *printer* may have five instances. If a process requests an instance of a resource type, the allocation of *any* instance of the type will satisfy the request. A process must request a resource before using it and must release the resource after using it. A process may request as many resources as it requires to carry out its designated task. Obviously, the number of resources requested may not exceed the total number of resources available in the system. In other words, a process cannot request three printers if the system has only two.

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

1. Request: The process requests the resource. If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.
2. Use: The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).
3. Release: The process releases the resource.

The request and release of resources are system calls.

Deadlock Characterization

Necessary Conditions:

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

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

2.Hold and wait. A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.

3.No preemption. Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

4.Circular wait. A set $\{P_0, P_1, ..., P_n\}$ of waiting processes must exist such that *Po* is waiting for a resource held by P_1, P_1 is waiting for a resource held by P2, ... , Pn-1 is waiting for a resource held by P_n and P_n is waiting for a resource held by *Po.*

All four conditions must hold for a deadlock to occur.

Resource-Allocation Graph

Deadlocks can be described more precisely in terms of a directed graph called a system resource allocation graph. This graph consists of a set of vertices V and a set of edges E. The set of vertices V is partitioned into two different types of nodes: P = { P1, P2, ... , Pn}, the set consisting of all the active processes in the system, and R = {R1, R2, ... , Rm} the set consisting of all resource types in the system. A directed edge from process Pi to resource type Rj is denoted by Pi → Rj; it signifies that process Pi has requested an instance of resource type Rj and is currently waiting for that resource. A directed edge from resource type Rj to process Pi is denoted by Rj → Pi ; it signifies that an instance of resource type Rj has been allocated to process Pi. A directed edge Pi → Rj is called a request edge; a directed edge Rj → Pi is called an assignment edge.

Pictorially we represent each process Pi as a circle and each resource type Rj as a rectangle. Since resource type Rj may have more than one instance, we represent each such instance as a dot within the rectangle. Note that a request edge points to only the rectangle Rj, whereas an assignment edge must also designate one of the dots in the rectangle.

The resource-allocation graph shown in Figure below:

- The sets P, R and E:

P = {P1, P2, P3}
R = {R1, R2, R3, R4}
E = {P1→R1, P2→ R3, R1→ P2, R2→ P2, R2→ P1, R3→ P3}

- Resource instances:
  One instance of resource type R1
  Two instances of resource type R2
  One instance of resource type R3
  Three instances of resource type R4

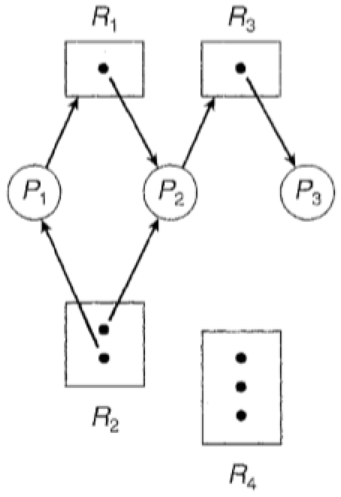


Fig. Resource-allocation graph

- 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.

Given the definition of a resource-allocation graph, it can be shown that, if the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.
If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred. If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred. Each process involved in the cycle is deadlocked. In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock.
If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. In this case, a cycle in. the graph is a necessary but not a sufficient condition for the existence of deadlock.

P1 → R 1 → P2 → R3 → P3 → R2 → P1
P2 → R3 → P3 → R2 → P2

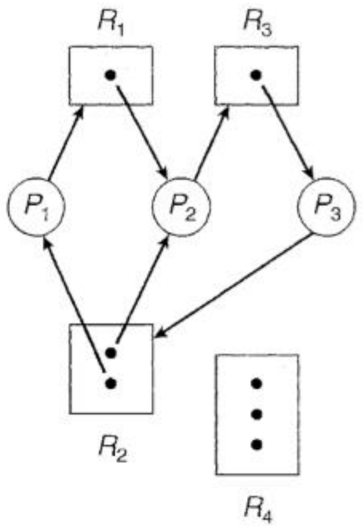


Fig. Resource-allocation graph with a deadlock

P1→R1→P3→R2→P1

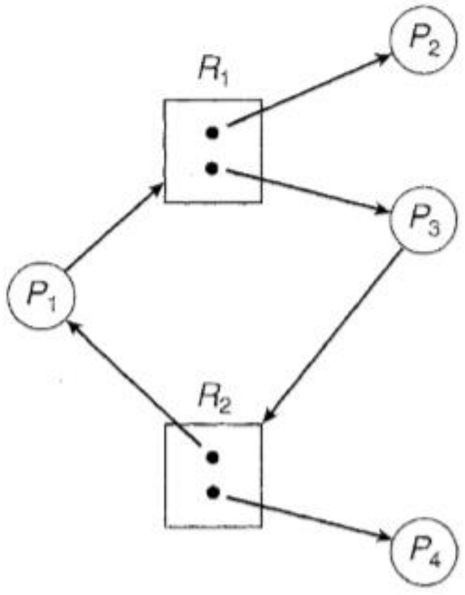


Fig. Resource-allocation graph with a cycle but no deadlock

Methods for handling deadlocks

Generally speaking, 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.

To ensure that deadlocks never occur, the system can use either a deadlock prevention or a deadlock-avoidance scheme. Deadlock prevention provides a set of methods for ensuring that at least one of the necessary conditions cannot hold.

Deadlock avoidance requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this additional knowledge, it can decide for each request whether or not the process should wait.
To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.
If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may arise. In this environment, the system can provide an algorithm that examines the state of the system to determine whether a deadlock has occurred and an algorithm to recover from the deadlock.

Deadlock Prevention

For a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can *prevent* the occurrence of a deadlock. We elaborate on this approach by examining each of the four necessary conditions separately.

Mutual Exclusion

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

Hold and Wait

To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources. One protocol that can be used requires each process to request and be allocated all its resources before it begins execution. We can implement this provision by requiring that system calls requesting resources for a process precede all other system calls.

No Preemption

The third necessary condition for deadlocks is that there be no preemption of resources that have already been allocated. To ensure that this condition does not hold, we can use the following protocol. If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources the process is currently holding are preempted. In other words, these resources are implicitly released. The preempted resources are added to the list of resources for which the process is waiting. The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

Circular Wait

The fourth and final condition for deadlocks is the circular-wait condition. One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration.

Deadlock Avoidance

Possible side effects of preventing deadlocks are low device utilization and reduced system throughput. An alternative method for avoiding deadlocks is to require additional information about how resources are to be requested. For example, in a system with one tape drive and one printer, the system might need to know that
process *P* will request first the tape drive and then the printer before releasing both resources, whereas process *Q* will request first the printer and then the tape drive. With this knowledge of the complete sequence of requests and
releases for each process, the system can decide for each request whether or not the process should wait in order to avoid a possible future deadlock. Each request requires that in making this decision the system consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process. The various algorithms that use this approach differ in the amount and type
of information required. The simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need. Given this a priori information, it is possible to construct an algorithm that ensures that the system will never enter a deadlocked state. Such an algorithm defines the deadlock-avoidance approach. A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular wait condition can never exist.

Safe State

A state is *safe* if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a safe sequence. A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state. Not all unsafe states are deadlocks however An unsafe state *may* lead to a deadlock.

We consider a system with twelve magnetic tape drives and three processes: *Po,* P1, and P2. Process *Po* requires ten tape drives, process P1 may need as many as four tape drives, and process P2 may need up to nine tape drives. Suppose that, at time *to,* process *Po* is holding five tape drives, process P1 is holding two tape drives, and process P2 is holding two tape drives. (Thus, there are three free tape drives.)

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

At time *to,* the system is in a safe state. The sequence <P1, *Po,* P2> satisfies the safety condition. Process P1 can immediately be allocated all its tape drives and then return them (the system will then have five available tape drives); then process *Po* can get all its tape drives and return them (the system will then have ten available tape drives); and finally process *P2* can get all its tape drives and return them (the system will then have all twelve tape drives available).

Resource-Allocation-Graph Algorithm

If we have a resource-allocation system with only one instance of each resource type, we can use a variant of the resource-allocation graph for deadlock avoidance. In addition to the request and assignment edges already described, we introduce a new type of edge, called a claim edge. A claim edge *Pi*→*Pj* indicates that process *Pi* may request resource *Rj* at some time in the future. This edge resembles a request edge in direction but is represented in the graph by a dashed line. When process *Pi* requests resource *Rj,* the claim edge *Pi*→*Rj* is converted to a request edge. Similarly, when a resource *Rj* is released by *Pi,* the assignment edge *Rj* → *Pi* is reconverted to a claim edge *Pi*→*Rj.* indicates that process *Pi* may request resource *Rj* at some time in the future. This edge resembles a request edge in direction but is represented in the graph by a dashed line. When process *Pi* requests resource *Rj,* the claim edge *Pi*→*Rj* is converted to a request edge. Similarly, when a resource *Rj* is released by *Pi,* the assignment edge *Rj* →*Pi* is reconverted to a claim edge *Pi*→*Rj.*

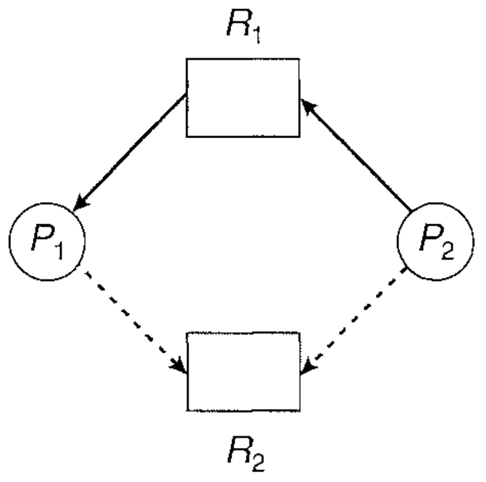


Figure 7.6 Resource-allocation graph for deadlock avoidance.

We note that the resources must be claimed a priori in the system. That is, before process *Pi* starts executing, all its claim edges must already appear in the resource-allocation graph. We can relax this condition by allowing a claim edge *Pi*→*Rj* to be added to the graph only if all the edges associated with process *Pi* are claim edges.
Now suppose that process *Pi* requests resource *Rj*. The request can be granted only if converting the request edge *Pi*→*Rj* to an assignment edge *Rj*→*Pi* does not result in the formation of a cycle in the resource-allocation graph. We check for safety by using a cycle-detection algorithm. An algorithm for detecting a cycle in this graph requires an order of n^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. In that case, process *Pi* will have to wait for its requests to be satisfied.

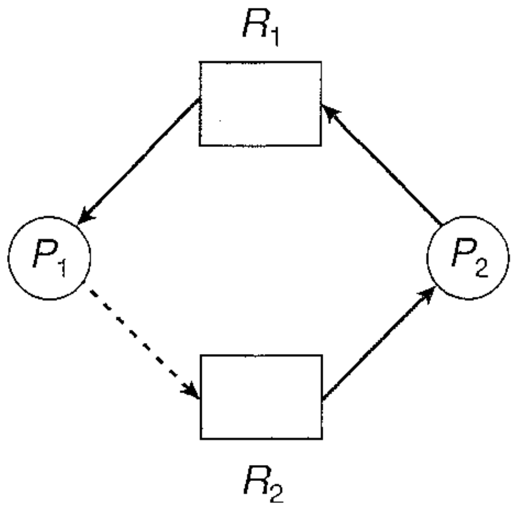


Figure 7.7 An unsafe state in a resource-allocation graph.

Banker's Algorithm

The resource-allocation-graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type. The deadlock avoidance algorithm that we describe next is applicable to such a system but is less efficient than the resource-allocation graph scheme. This algorithm is commonly known as the *banker's algorithm.* The name was chosen because 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.
When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This nun1.ber may not exceed the total number of resources in the system. When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.
Several data structures must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system. We need the following data structures, where *n* is the number of processes in the system and *m* is the number of resource types:
***Available.** A vector of length *m* indicates the number of available resources of each type. If *Available[j]* equals *k,* then *k* instances of resource type *Rj* are available.

***Max.** An *n* x *m* matrix defines the maximum demand of each process. If *Max[i] [j]* equals *k*, then process *Pi* may request at most *k* instances of resource type Rj.
***Allocation.** An *n* x *m* matrix defines the number of resources of each type currently allocated to each process. If *Allocation[i][j]* equals k, then process *Pi* is currently allocated k instances of resource type *Rj*.
***Need.** An *n* x *m* matrix indicates the remaining resource need of each process. If *Need[i][j]* equals *k*, then process *Pi* may need *k* more instances of resource type *Rj* to complete its task. Note that *Need[i][j]* equals *Max[i][j] - Allocation [i][j]*.

These data structures vary over time in both size and value. To simplify the presentation of the banker's algorithm, we next establish some notation. Let X and Y be vectors of length *n*. We say that X<=*Y* if and only if *X[i]* <= Y[i] for all *i* = 1, 2, ... , *n*. For example, if X = (1,7,3,2) and *Y* = (0,3,2,1), then *Y* <=X. In addition, *Y* < X if *Y* <=X and Y!= X.
We can treat each row in the matrices *Allocation* and *Need* as vectors and refer to them as $Allocation_i$ and $Need_i$. The vector $Allocation_i$ specifies the resources currently allocated to process *Pi;* the vector $Need_i$ specifies the additional resources that process *Pi* may still request to complete its task.

Safety Algorithm
We can now present the algorithm for finding out whether or not a system is in a safe state. This algorithm can be described as follows:

```
1. Let Work and Finish be vectors of length m and n, respectively. Initialize
   Work= Available and Finish[i] =false for i = 0, 1, ... , n - 1.
2. Find an index i such that both
   a. Finish[i] ==false
   b. Needi <=  Work
   If no such i exists, go to step 4.
3. Work = Work + Allocation;
   Finish[i] = true
   Go to step 2.
4. If Finish[i] ==true for all i, then the system is in a safe state.
   This algorithm may require an order of $m$ x $n^2$ operations to determine whether a state is safe.
```

Resource-Request Algorithm
Next, we describe the algorithm for determining whether requests can be safely granted.
Let $Request_i$ be the request vector for process *Pi*. If *Requesti [j]* == *k*, then process *Pi* wants *k* instances of resource type *Rj*. When a request for resources is made by process *Pi*, the following actions are taken:

1. If *Requesti<=Needi*, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If *Requesti<= Available*, go to step 3. Otherwise, *Pi* must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process *Pi* by modifying the state as follows:
   Available= Available - Request i;
   Allocation i =Allocation i +Request i;
   Need i =Need i - Request i;

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

An Illustrative Example
To illustrate the use of the banker's algorithm, consider a system with five processes *Po* through *P4* and three resource types *A*, *B*, and C. Resource type *A* has ten instances, resource type *B* has five instances, and resource type C has seven instances. Suppose that, at time T_0, the following snapshot of the system has been taken:

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

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

We claim that the system is currently in a safe state. Indeed, the sequence < P_1, P_3, P_4, P_2, Po> satisfies the safety criteria. Suppose now that process P_1 requests one additional instance of resource type *A* and two instances of resource type C, so $Request_1$ = (1,0,2). To decide whether this request can be immediately granted, we first check that $Request_1$ <= *Available - that* is, that (1,0,2) <= (3,3,2), which is true. We then pretend that this request has been fulfilled, and we arrive at the following new state:

	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	
P_4	0 0 2	4 3 1	

We must determine whether this new system state is safe. To do so, we execute our safety algorithm and find that the sequence <P_1, P_3, P4, Po, P_2> satisfies the safety requirement. Hence, we can immediately grant the request of process P1. You should be able to see, however, that when the system is in this state, a request for (3,3,0) by P_4 cannot be granted, since the resources are not available. Furthermore, a request for (0,2,0) by *Po* cannot be granted, even though the resources are available, since the resulting state is unsafe.

Deadlock Detection

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

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

Single Instance of Each Resource Type

If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a *wait-for* graph. We obtain this graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges. 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*. For example, in below figure, we present a resource-allocation graph and the corresponding wait-for graph. As before, 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. An algorithm to detect a cycle in a graph requires an order of n^2 operations, where *n* is the number of vertices in the graph.

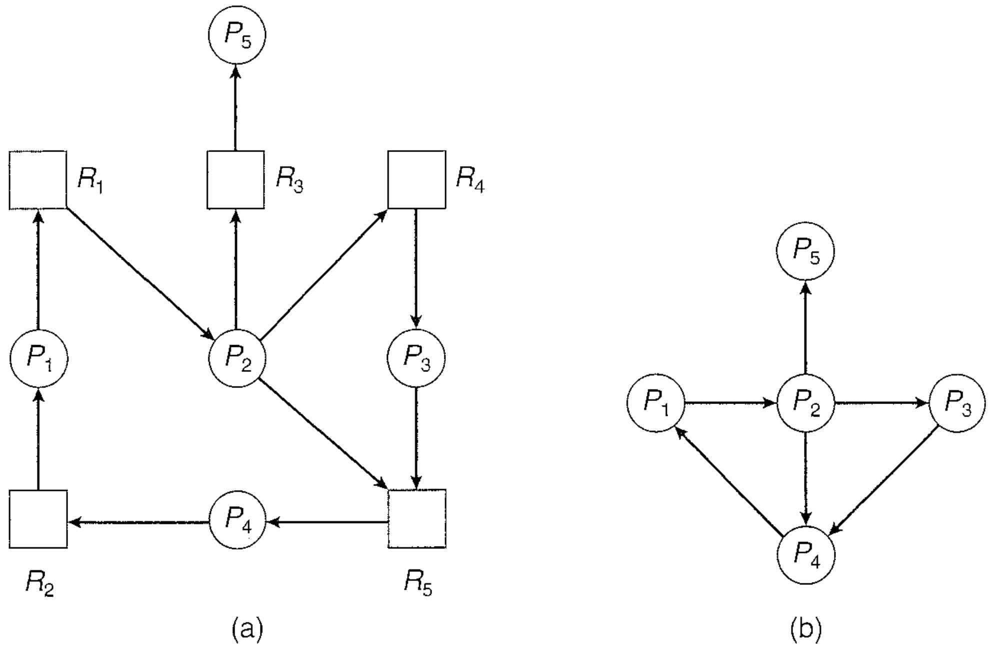
Several Instances of a Resource Type

The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type. We turn now to a deadlock detection algorithm that is applicable to such a system. The algorithm employs several time-varying data structures that are similar to those used in the banker's algorithm:

Available. A vector of length *m* indicates the number of available resources of each type.

Allocation. An n x *m* matrix defines the number of resources of each type currently allocated to each process.

Request. An *n* x *m* matrix indicates the current request of each process. If *Request[i][j]* equals *k,* then process *Pi* is requesting *k* more instances of resource type *Rj*.



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

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

Work= Available. For i= 0, 1, ... , *n-1,* if *Allocationi !=* 0, then *Finish[i] =false;*
otherwise, *Finish[i] = true.*
2. Find an index *i* such that both
a. *Finish[i] ==false*
b. *Request i <= Work*
If no such *i* exists, go to step 4.
3.Work= Work+ Allocation i
Finish[i] = true
Go to step 2.
4.If *Finish[i] ==false* for some *i,* 0 <= *i* < *n,* then the system is in a deadlocked state. Moreover, if *Finish[i] ==false,* then process *Pi* is deadlocked.

This algorithm requires an order o£ m x n^2 operations to detect whether the system is in a deadlocked state. You may wonder why we reclaim the resources of process *Pi* (in step 3) as soon as we determine that *Request i<= Work* (in step 2b). We know that *Pi* is currently *not* involved in a deadlock (since *Request i<= Work).* Thus, we take an optimistic attitude and assume that *P;* will require no more resources to complete its task; it will thus soon return all currently allocated resources to the system. If our assumption is incorrect, a deadlock may occur later. That deadlock will be detected the next tince the deadlock-detection algorithm is invoked.

Detection-Algorithm Usage

When should we invoke the detection algorithm? The answer depends on two factors:
1.How *often* is a deadlock likely to occur?
2.How *many* processes will be affected by deadlock when it happens?

Recovery from Deadlock

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

Process Termination

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

- 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, and the results of these partial computations must be discarded and probably will have to be recomputed later.
- 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. Many factors may affect which process is chosen, including:
1.What the priority of the process is
2. How long the process has computed and how much longer the process will compute before completing its designated task
3.How many and what types of resources the process has used (for example, whether the resources are simple to preempt)
4.How many more resources the process needs in order to complete
5. How many processes will need to be terminated
6.Whether the process is interactive or batch

Resource Preemption

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:

Selecting a victim. Which resources and which processes are to be preempted? As in process termination, we must determine the order of preemption to minimize cost. Cost factors may include such parameters as the number of resources a deadlocked process is holding and the amount of time the process has thus far consumed during its execution.
Rollback. If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state and restart it from that state. Since, in general, it is difficult to determine what a safe state is, the simplest solution is a total rollback: abort the process and then restart it. Although it is more effective to roll back the process only as far as necessary to break the deadlock, this method requires the system to keep more information about the state of all running processes.
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?

CHAPTER 4

Memory management Strategies

Background

The CPU fetches instructions from memory according to the value of the program counter. These instructions may cause additional loading from and storing to specific memory addresses. A typical instruction-execution cycle, for example, first fetches an instruction from memory. The instruction is then decoded and may cause operands to be fetched from memory. After the instruction has been executed on the operands, results may be stored back in memory. The memory unit sees only a stream of memory addresses; it does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, and so on) or what they are for (instructions or data). Accordingly, we can ignore *how* a program generates a memory address. We are interested only in the sequence of memory addresses generated by the running program. We begin our discussion by covering several issues that are pertinent to the various techniques for managing memory. This coverage includes an overview of basic hardware issues, the binding of symbolic memory addresses to actual physical addresses, and the distinction between logical and physical addresses. We conclude the section with a discussion of dynamically loading and linking code and shared libraries.

Basic Hardware

Main memory and the registers built into the processor itself are the only storage that the CPU can access directly. There are machine instructions that take memory addresses as arguments, but none that take disk addresses. Therefore, any instructions in execution, and any data being used by the instructions, must be in one of these direct-access storage devices. If the data are not in memory, they must be moved there before the CPU can operate on them.
Registers that are built into the CPU are generally accessible within one cycle of the CPU clock. Most CPUs can decode instructions and perform simple operations on register contents at the rate of one or more operations per clock tick. The same cannot be said of main memory, which is accessed via a transaction on the memory bus. Completing a memory access may take many cycles of the CPU clock. In such cases, the processor normally needs to stall, since it does not have the data required to complete the instruction that it is executing. This situation is intolerable because of the frequency of memory accesses. The remedy is to add fast memory between the CPU and main memory. A memory buffer used to accommodate a speed differential, called a cache.

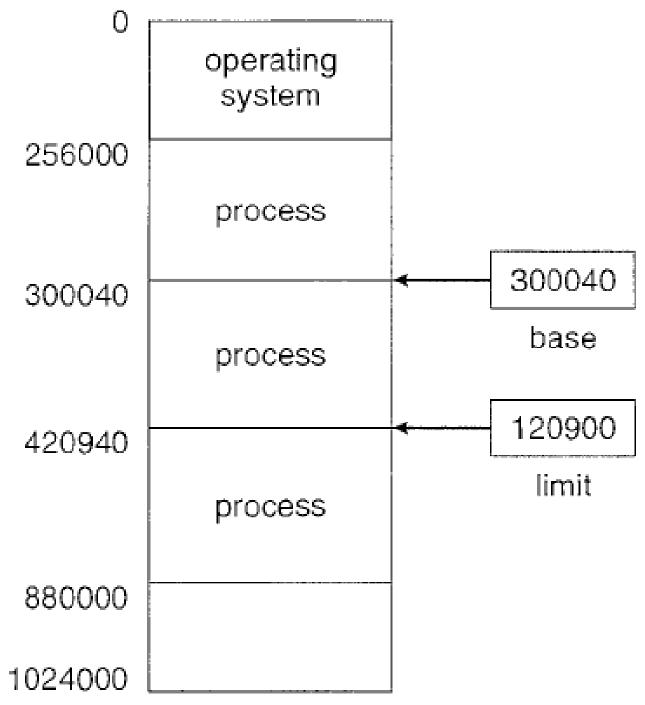


Fig: A base and a limit register define a logical address space

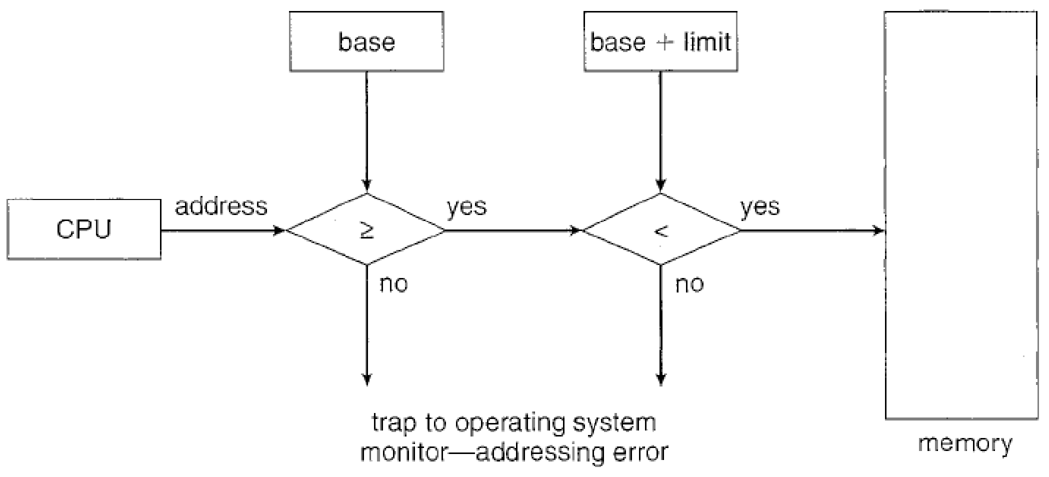


Fig: Hardware address protection with base and limit registers

Address Binding

Usually, a program resides on a disk as a binary executable file. To be executed, the program must be brought into memory and placed within a process. Depending on the memory management in use, the process may be moved between disk and memory during its execution. The processes on the disk that are waiting to be brought into memory for execution form the input queue. The normal procedure is to select one of the processes in the input queue and to load that process into memory. As the process is executed, it accesses instructions and data from memory. Eventually,

the process terminates, and its memory space is declared available. Most systems allow a user process to reside in any part of the physical memory. Thus, although the address space of the computer starts at 00000, the first address of the user process need not be 00000. This approach affects the addresses that the user program can use. In most cases, a user program will go through several steps-some of which may be optional-before being executed. Addresses may be represented in different ways during these steps. Addresses in the source program are generally symbolic (such as *count).* A compiler will typically bind these symbolic addresses to relocatable addresses (such as "14 bytes from the beginning of this module"). The linkage editor or loader will in turn bind the relocatable addresses to absolute addresses (such as 74014). Each binding is a mapping from one address space to another. Classically, the binding of instructions and data to memory addresses can be done at any step along the way:
Compile time. If you know at compile time where the process will reside in memory, then absolute code can be generated. For example, if you know that a user process will reside starting at location *R*, then the generated compiler code will start at that location and extend up from there. If, at some later time, the starting location changes, then it will be necessary to recompile this code. The 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. If the starting address changes, we need only reload the user code to incorporate this changed value. **Execution time**. If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. Special hardware must be available for this scheme to work.

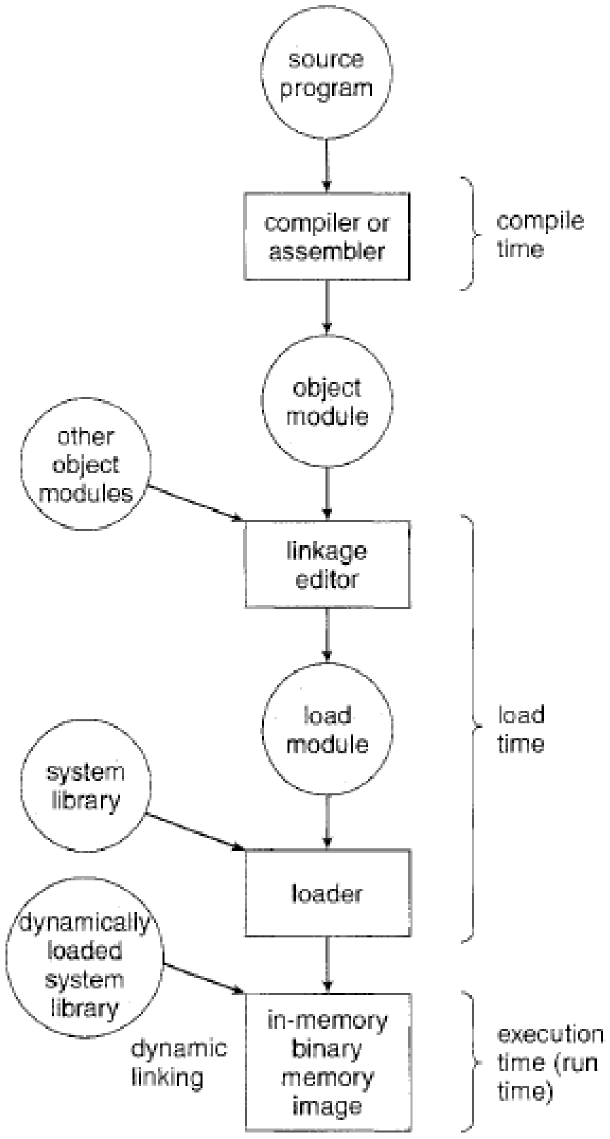


Fig: Multistep processing of a user program

WEEK-1

OS Basic Concepts.

- Computer system or its computing environment has a collection of resources that are very much essential for computing task.
- Resources of a computer include the processor, memory, files, peripheral devices, disk storage units.
- OS is a software that manages the basic resources of a computer it is a master to drive the hardware. OS is referred to as system software
- When a computer is switched on the OS is the only program that runs all the time.
- OS allocates main memory for the running programs on CPU, schedules and manage the data transfer to and from I/O devices to main memory and secondary storage units
- Loading and executing the programmes efficiently & monitor the input, output, storage and retrieval of data during the course of execution.
- OS provides easy to use and easy to access user friendly computing environment.
- It provides command line user interface (CUI) with ms dos or a GUI (Graphical user interface)(GUI) with all windows.

  Eg: windows, Mac, Linux, Ubuntu, etc.
  Android (MOS), IOS (MOS), Unix

Need of an OS

- It acts as a platform for application program these can be run and helps the users to

perform specific task easily. It is designed in such
a way that it operates, controls and executes
various applications on computer.

- Managing IO units :

OS allows the computer to manage its own resources such as monitor, keyboard, mouse etc. OS controls the various syste
i/o resources and allocates them to the users ~~or~~ programs asper their requireme

- Multitasking :

OS manages memory and allows multiple programs to run in their own space and even communicate with each other through shared memory.

- Consistent UI :

It provides easy to work user interfac
so the user doesn't have to learn a different
UI everytime and can focus on the content and
be productive as quick as possible.

- Functions of an OS

1. Process management :

It deals with management of CPU that is allotment of CPU time to different processes.

2) Context switching:
Multiple running process on the system may need a change of state in execution

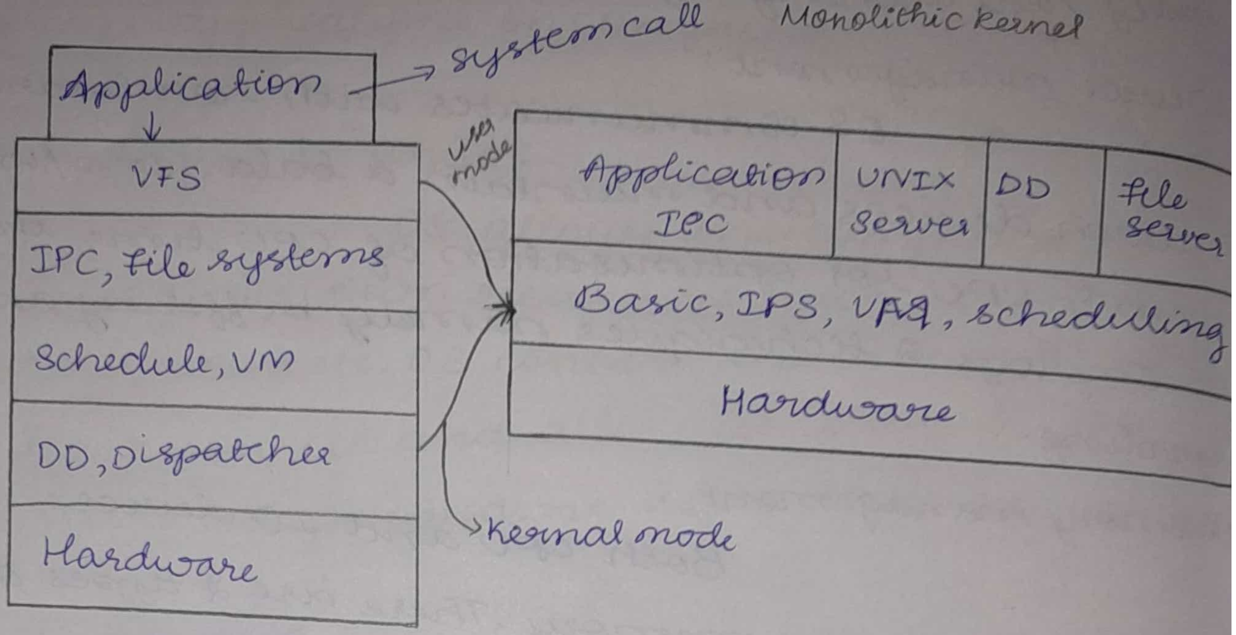
3) Device management:
OS communicates with hardware & attach devices and maintain a balance b/w them & CPU for optimisation of CPU time the OS employs 2 techniques namely buffering and spooling.

4) Memory management:
Both CPU and IO devices interact with the memory. There are 2 types of memory namely partitioning and virtual memory.

5) File management:
OS manages file, folder and directory systems on a computer. OS keeps all the info using FAT (File allocation table) this stores general info about files like file name, types, size, the starting address, and access mode

<u>Kernal</u>:
It is a central component of OS that manages Operations of computer and hardware. It manages operations of memory and CPU time. Kernel loads first into memory when OS kernal is loaded and remains into the memory until OS is shutdown again It acts as an interface b/w user application & hardware, functions of a kernel includes scheduling processes, resource allocation, device management, interrupt handling, (memory management) MM, PM (Process management).
Micro kernel is a s/w which contains the required

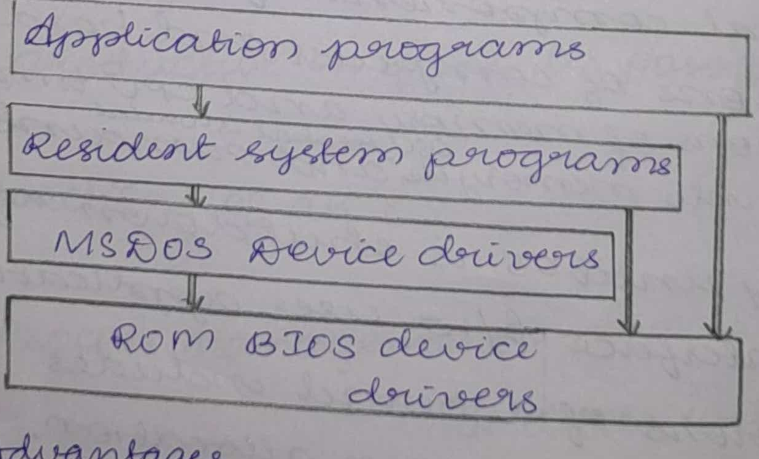
minimum amount of functions, data and features to implement an OS.



Structure of OS.

It depends on how the various components are connected and melded into kernel.

Simple structure
- No well defined structure
- It is small simple and limited systems.
- Interfaces, levels of functionality are not well separated
- If one user program fails the entire system will crash.



Advantages
- Better application performance because of few interfaces between application program & hardware
- Easy for developers to develop such OS.

Disadvantage

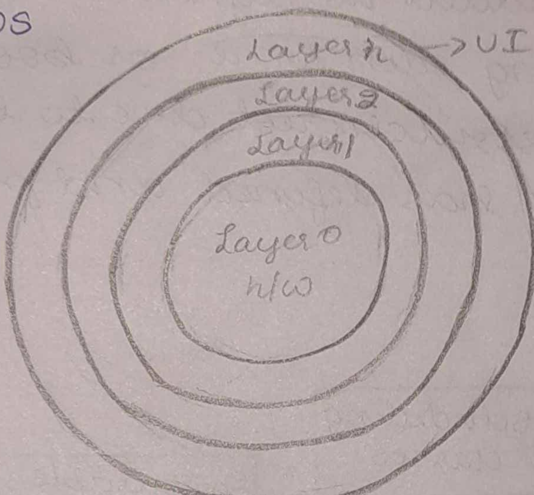
• ~~The do~~ It doesn't enforce data hiding in OS.

2) Layered structure:

In this, OS is broken into number of layers (Levels) The bottom layer (layer 0) is the h/w and the topmost layer (Layer N) is the user interface. It is designed in such a way that each layer uses the function of lower level layers only

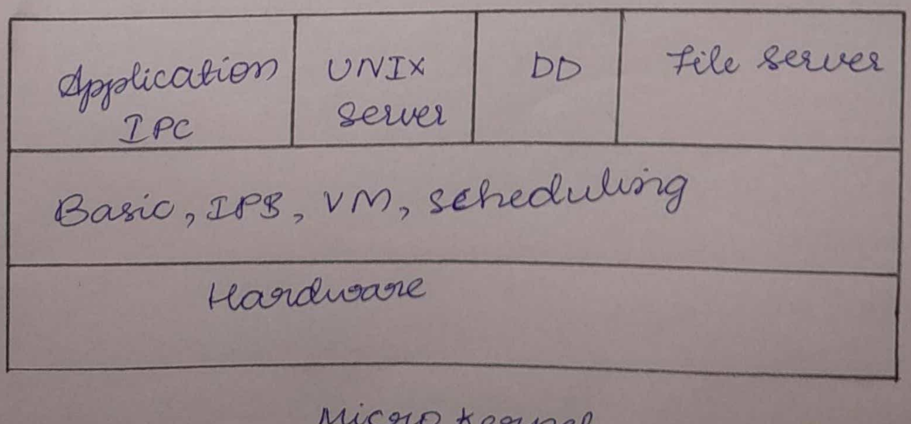
Advantage
The debugging is possible layer wise and the system verification is easier.

UNIX OS



Micro kernel:

Designs OS by removing all non essential components from the kernel and implement them as system and user programs. All new services can be added to the users space and doesn't require kernel to be modified.
It is secure and reliable



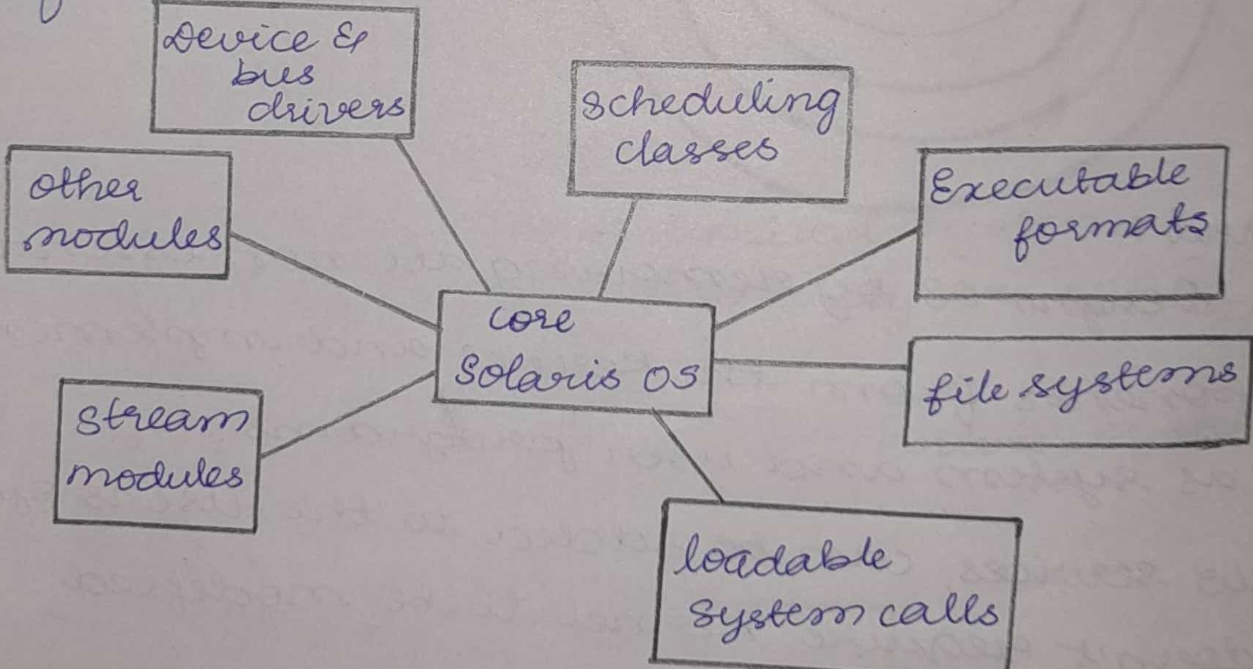
Micro kernel.

Advantages

- It is portable to various platforms.
- As they are smaller it can be tested easily.
- As intermodule communication increases it degrades the system performance.

3) Modular structure:

It involves designing of a modular kernel. Kernel has only set of four components and other services can be added as dynamically loaded modules. either during run time or boot time. It resembles layered structured due to the fact that each kernel has defined and protected interfaces



CHAPTER 5

VIRTUAL MEMORY MANAGEMENT

Background

The instructions being executed must be in physical memory. The first approach to meeting this requirement is to place the entire logical address space in physical memory. Dynamic loading can help to ease this restriction, but it generally requires special precautions and extra work by the programmer.
The requirement that instructions must be in physical memory to be executed seems both necessary and reasonable; but it is also unfortunate, since it limits the size of a program to the size of physical memory. In fact, an examination of real programs shows us that, in many cases, the entire program is not needed. For instance, consider the following:

- Programs often have code to handle unusual error conditions. Since these errors seldom, if ever, occur in practice, this code is almost never executed.
- Arrays, lists, and tables are often allocated more memory than they actually need. An array may be declared 100 by 100 elements, even though it is seldom larger than 10 by 10 elements. An assembler symbol table may have room for 3,000 symbols, although the average program has less than 200 symbols.
- Certain options and features of a program may be used rarely. For instance, the routines on U.S. government computers that balance the budget have not been used in many years.

Even in those cases where the entire program is needed, it may not all be needed at the same time.
The ability to execute a program that is only partially in memory would confer many benefits:

- A program would no longer be constrained by the amount of physical memory that is available. Users would be able to write programs for an extremely large *virtual* address space, simplifying the programming task.
- Because each user program could take less physical memory, more programs could be run at the same time, with a corresponding increase in CPU utilization and throughput but with no increase in response time or turnaround time.
- Less I/O would be needed to load or swap user programs into memory, so each user program would run faster.

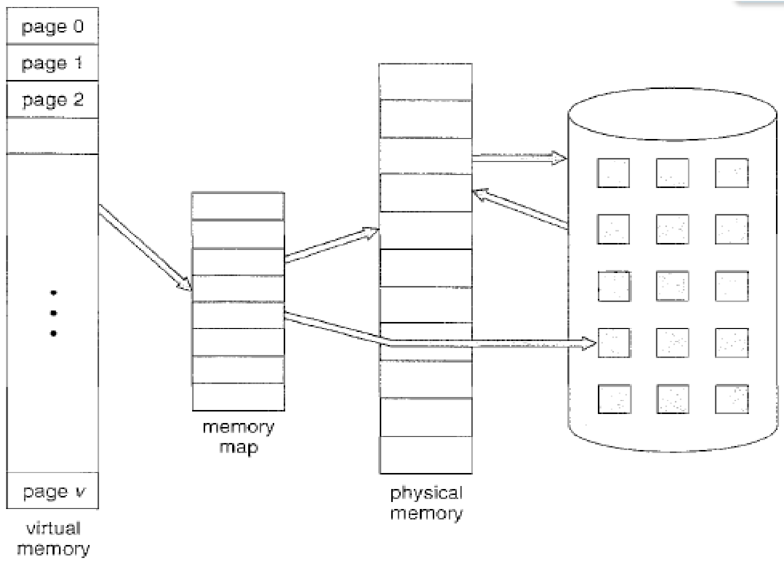


Fig: Diagram showing virtual memory that is larger than physical memory.

Thus, running a program that is not entirely in memory would benefit both the system and the user.

Virtual Memory involves the separation of logical memory as perceived by users 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). Virtual memory makes the task of programming much easier, because the programmer no longer needs to worry about the amount of physical memory available; she can concentrate instead on the problem to be programmed.

The **virtual address space** of a process refers to the logical (or virtual) view of how a process is stored in memory. Typically, this view is that a process begins at a certain logical address-say, address 0-and exists in contiguous memory, as shown in next Figure. Though, that in fact physical memory may be organized in page frames and that the physical page frames assigned to a process may not be contiguous. It is up to the memory management unit (MMU) to map logical pages to physical page frames in memory.

Note in Figure that we allow for the heap to grow upward in memory as it is used for dynamic memory allocation. Similarly, we allow for the stack to grow downward in memory through successive function calls. The large blank space (or hole) between the heap and the stack is part of the virtual address space but will require actual physical pages only if the heap or stack grows.

Virtual address spaces that include holes are known as sparse address spaces. Using a sparse address space is beneficial because the holes can be filled as the stack or heap segments grow or if we wish to dynamically link libraries (or possibly other shared objects) during program execution.

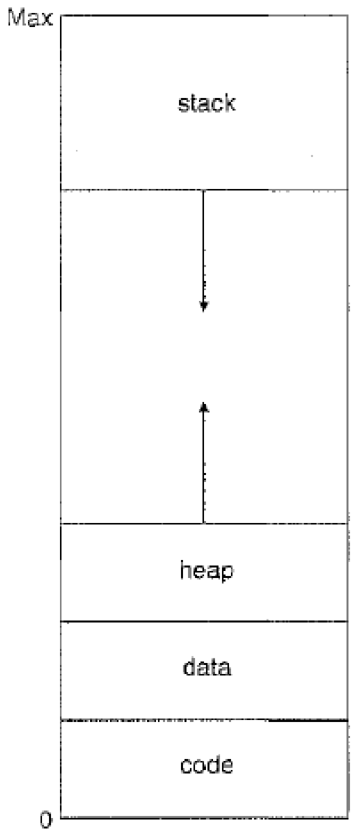


Figure Virtual address space

In addition to separating logical memory from physical memory, virtual memory allows files and memory to be shared by two or more processes through page sharing. This leads to the following benefits:

- System libraries can be shared by several processes through mapping of the shared object into a virtual address space. Although each process considers the shared libraries to be part of its virtual address space, the actual pages where the libraries reside in physical memory are shared by all the processes. Typically, a library is mapped read-only into the space of each process that is linked with it.

- Similarly, virtual memory enables processes to share memory
- Virtual memory can allow pages to be shared during process creation with the fork() system call thus speeding up process creation.

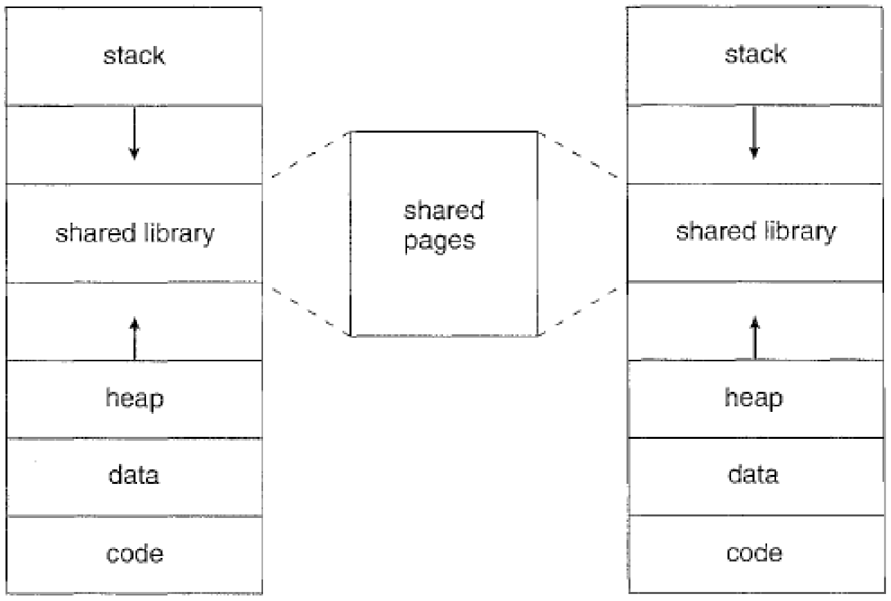


Fig: Shared library using virtual memory

DEMAND PAGING

Consider how an executable program might be loaded from disk into memory. One option is to load the entire program in physical memory at program execution time. However, a problem with this approach is that we may not initially *need* the entire program in memory. Suppose a program starts with a list of available options from which the user is to select. Loading the entire program into memory results in loading the executable code for *all* options, regardless of whether an option is ultimately selected by the user or not. An alternative strategy is to load pages only as they are needed. This technique is known as Demand Paging and is commonly used in virtual memory systems. With demand-paged virtual memory, pages are only loaded when they are demanded during program execution; pages that are never accessed are thus never loaded into physical memory.

A demand-paging system is similar to a paging system with swapping where 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. Since we are now viewing a process as a sequence of pages, rather than as one large contiguous address space, use of the term *swapper* is technically incorrect. A swapper manipulates entire processes, whereas a pager is concerned with the individual pages of a process. We thus use *pager,* rather than *swapper*, in connection with demand paging.

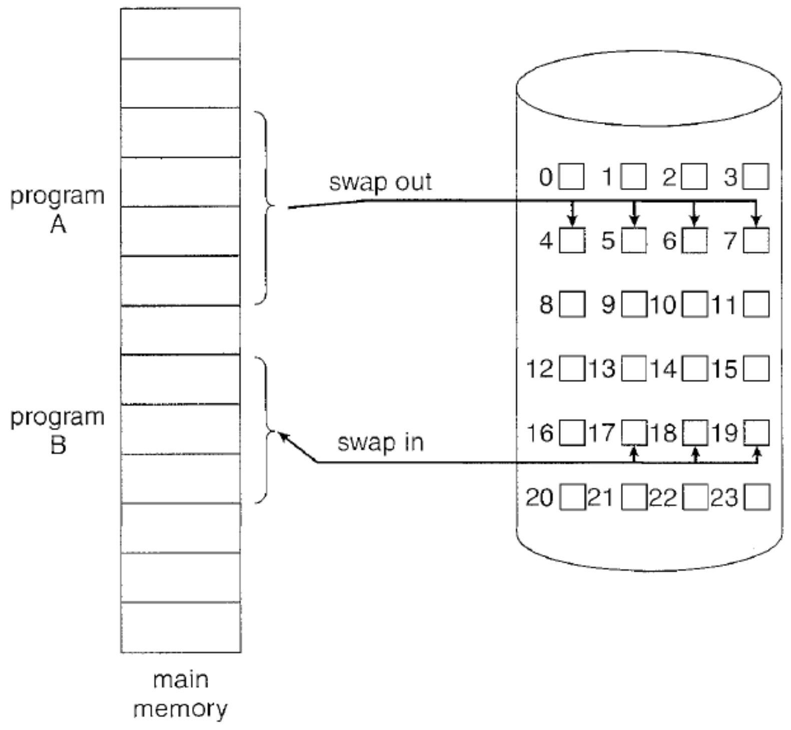


FIG: Transfer of a paged memory to contiguous disk space

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. Thus, it avoids reading into memory pages that will not be used anyway, decreasing the swap time and the amount of physical memory needed. With this scheme, we need some form of hardware support to distinguish between the pages that are in memory and the pages that are on the disk. The valid -invalid bit scheme can be used for this purpose. This time, however, when this bit is set to "valid/' the associated page is both legal and in memory. If the bit is set to "invalid/' the page either is not valid (that is, not in the logical address space of the process) or is valid but is currently on the disk. The page-table entry for a page that is brought into memory is set as usual but the page-table entry for a page that is not currently in memory is either simply marked invalid or contains the address of the page on disk. Notice that marking a page invalid will have no effect if the process never attempts to access that page. Hence, if we guess right and page in all and only those pages that are actually needed, the process will run exactly as though we had brought in all pages. While the process executes and accesses pages that are memory resident, execution proceeds normally.

But what happens if the process tries to access a page that was not brought into memory? Access to a page marked invalid causes a page fault. The paging hardware, in translating the address through the page table, will notice that the invalid bit is set, causing a trap to the operating system. This trap is the result of the operating system's failure to bring the desired page into memory. The procedure for handling this page fault is straightforward.

1) We check an internal table (usually kept with the process control block) for this process to determine whether the reference was a valid or an invalid memory access.
2) If the reference was invalid, we terminate the process. If it was valid, but we have not yet brought in that page, we now page it in.
3) We find a free frame (by taking one from the free-frame list, for example).
4) We schedule a disk operation to read the desired page into the newly allocated frame.

5) When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.
6) We restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory.

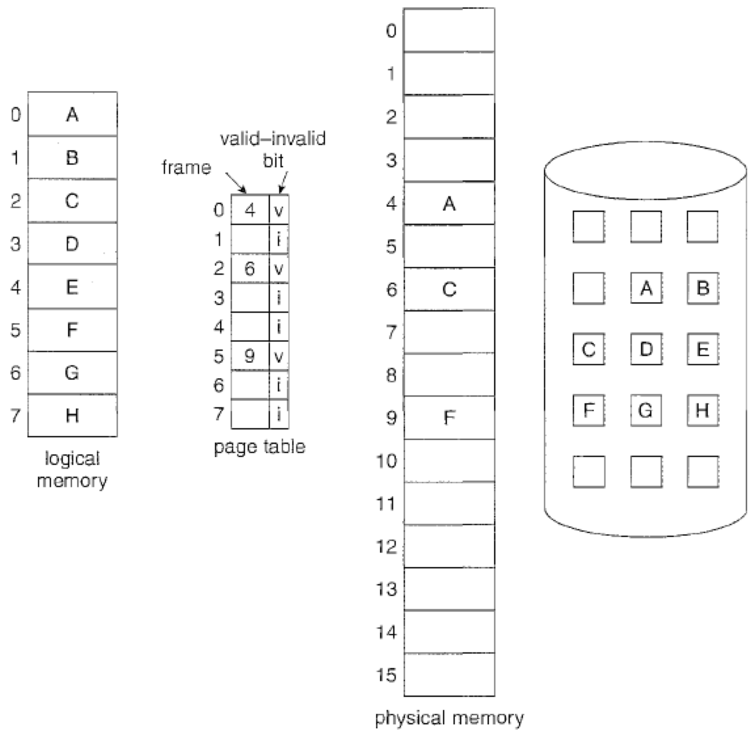


Fig: Page table when some pages are not in main memory.

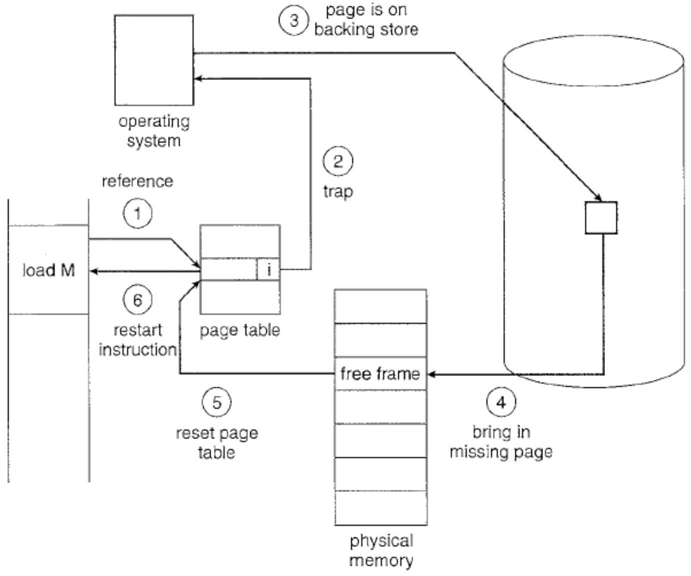


Fig: Steps in handling a page fault

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.

The hardware to support demand paging is the same as the hardware for paging and swapping:

- Page table. This table has the ability to mark an entry invalid through a valid -invalid bit or a special value of protection bits.
- Secondary memory. This memory holds those pages that are not present in main memory. The secondary memory is usually a high-speed disk. It is known as the swap device, and the section of disk used for this purpose is known as swap space.

A crucial requirement for demand paging is the ability to restart any instruction after a page fault. Because we save the state (registers, condition code, instruction counter) of the interrupted process when the page fault occurs, we must be able to restart the process in *exactly* the same place and state, except that the desired page is now in memory and is accessible.

Performance of Demand Paging

Demand paging can significantly affect the performance of a computer system. To see why, let's compute the effective access time for a demand-paged memory. For most computer systems, the memory-access time, denoted ma, ranges from 10 to 200 nanoseconds. As long as we have no page faults, the effective access time is equal to the memory access time. If, however a page fault occurs, we must first read the relevant page from disk and then access the desired word.

Let p be the probability of a page fault ($0 <= p <= 1$). We would expect p to be close to zero-that is, we would expect to have only a few page faults. The effective access time is then

effective access time= $(1 - p)$ x $ma + p$ x page fault time.

Copy-on-write

process creation using the fork() system call may initially bypass the need for demand paging by using a technique similar to page sharing This technique provides for rapid process creation and minimizes the number of new pages that must be allocated to the newly created process.

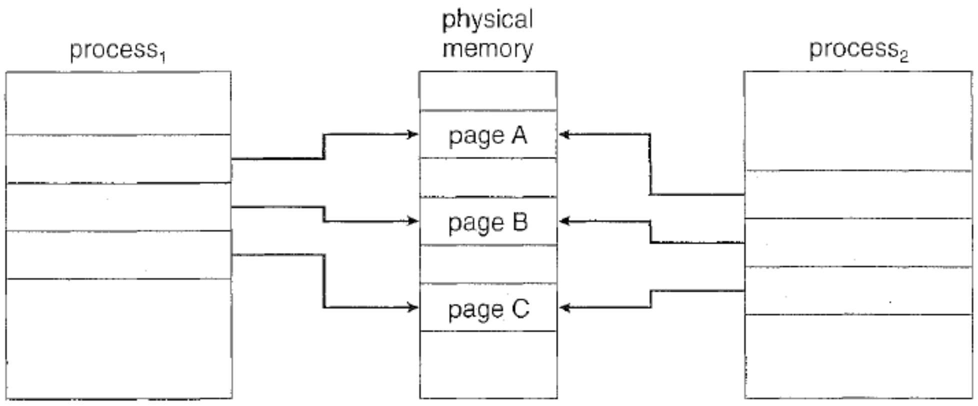


Fig: Before process 1 modifies page C.

The fork() system call creates a child process that is a duplicate of its parent. Traditionally, fork() worked by creating a copy of the parent's address space for the child, duplicating the pages belonging to the parent. However, considering that many child processes invoke the exec() system call immediately after creation, the copying of the parent's address space may be unnecessary. Instead, we can use a technique known as COPY-ON-WRITE, which works by allowing the parent and child processes initially to share the same pages. These shared pages are marked as copy-on-write pages, meaning that if either process writes to a shared page, a copy of the shared page is created.

For example, assume that the child process attempts to modify a page containing portions of the stack, with the pages set to be copy-on-write. The operating system will create a copy of this page, mapping it to the address space of the child process. The child process will then modify its copied page and not the page belonging to the parent process. Obviously, when the copy-on-write technique is used, only the pages that are modified by either process are copied; all unmodified pages can be shared by the parent and child processes. Note, too, that only pages that can be modified need be marked as copy-on-write. Pages that cannot be modified (pages containing executable code) can be shared by the parent and child.

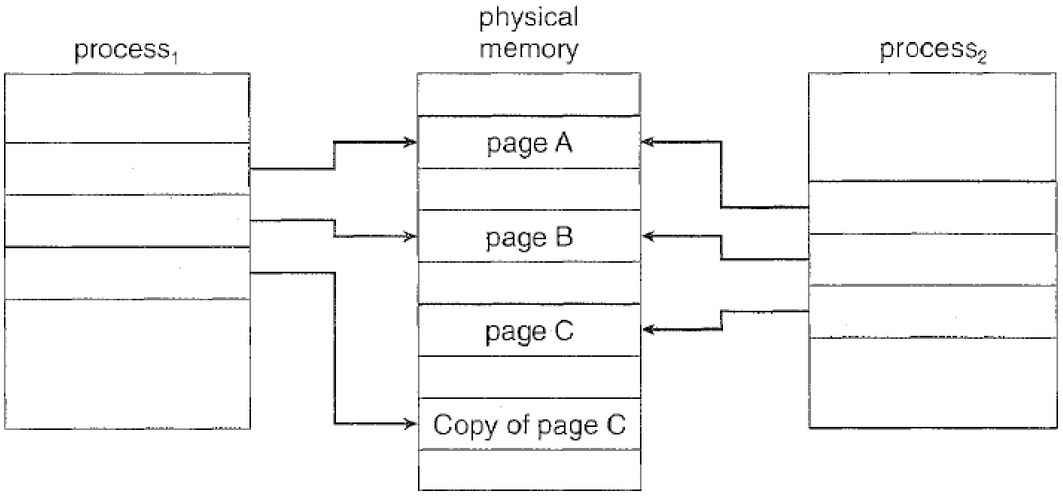


Fig: After process 1 modifies page C.

PAGE REPLACEMENT

If we increase our degree of multiprogramming, we are memory. If we run six processes, each of which is ten pages in size but uses only five pages, we have higher CPU utilization and throughput, ten frames to spare. It is possible, however, that each of these processes, for a particular data set, may suddenly try to use all ten of its pages, resulting in a need for sixty frames when only forty are available. Further, consider that system memory is not used only for holding program pages. Buffers for I/ 0 also consume a considerable amount of memory. This use can increase the strain on memory-placement algorithms. Deciding how much memory to allocate to I/0 and how much to program pages is a significant challenge. Some systems allocate a fixed percentage of memory for I/0 buffers, whereas others allow both user processes and the I/0 subsystem to compete for all system memory.

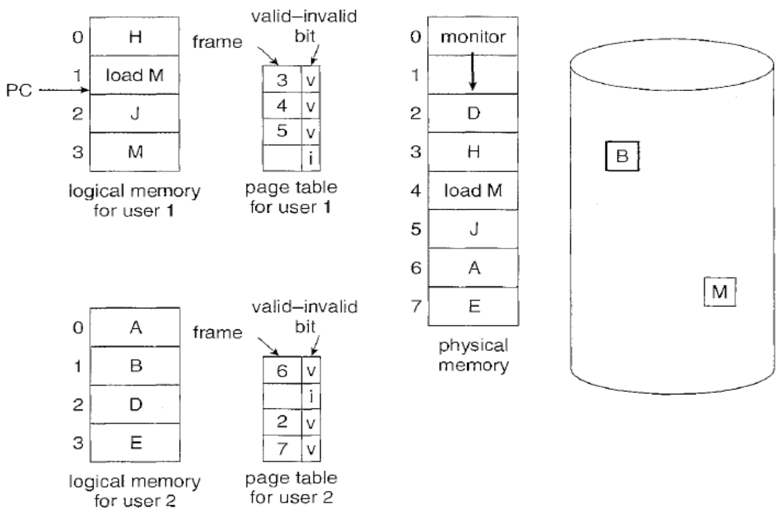


Fig: Need for page replacement

Over-allocation of memory manifests itself as follows. While a user process is executing, a page fault occurs. 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.
The operating system has several options at this point. It could terminate the user process. However, demand paging is the operating system's attempt to improve the computer system's utilization and throughput. Users should not be aware that their processes are running on a paged system-paging should be logically transparent to the user. So, this option is not the best choice.
The operating system could instead swap out a process, freeing all its frames and reducing the level of multiprogramming. This option is a good one in certain circumstances.

Basic Page Replacement
Page replacement takes the following approach. If no frame is free, we find one that is not currently being used and free it. We can free a frame by writing its contents to swap space and changing the page table (and all other tables) to indicate that the page is no longer in memory. We can now use the freed frame to hold the page for which the process faulted. We modify the page-fault service routine to include page replacement:

1) Find the location of the desired page on the disk.
2) Find a free frame:
    a. If there is a free frame, use it.
    b. If there is no free frame, use a page-replacement algorithm to select a victim frame.
    c. Write the victim 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.

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.
We can reduce this overhead by using a modify bit (or dirty bit). When this scheme is used, each page or frame has a modify bit associated with it in the hardware. The modify bit for a page is set by the hardware whenever any word or byte in the page is written into, indicating that the page has been modified. When we select a page for replacement, we examine its modify bit. If the bit is set, we know that the page has been modified since it was read in from the disk. In this case, we must write the page to the disk. If the modify bit is not set, however, the page has *not* been modified since it was read into memory. In this case, we need not write the memory page to the disk: it is already there. This technique also applies to read-only pages (for example, pages of binary code). Such pages cannot be modified; thus, they may be discarded when desired. This scheme can significantly reduce the time required to service a page fault, since it reduces I/O time by one-half *if* the page has not been modified.

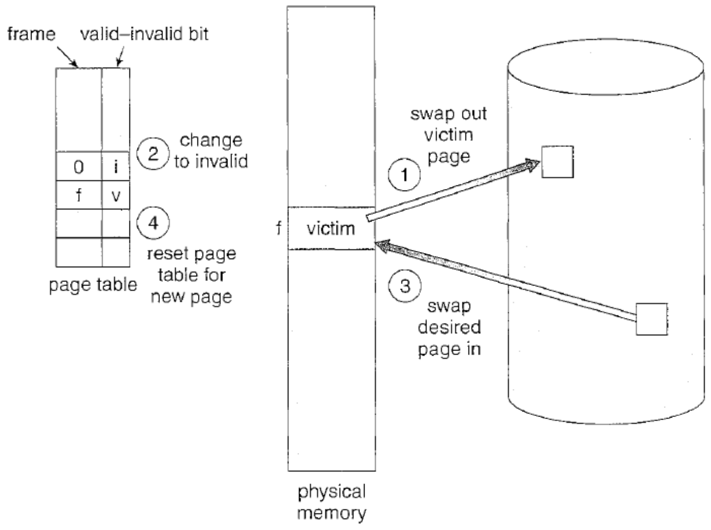


Fig: Page replacement

Page replacement is basic to demand paging. It completes the separation between logical memory and physical memory. With this mechanism, an enormous virtual memory can be provided for programmers on a smaller physical memory. With no demand paging, user addresses are mapped into physical addresses, so the two sets of addresses can be different. All the pages of a process still must be in physical memory, however. With demand paging, the size of the logical address space is no longer constrained by physical memory.

FIFO Page Replacement

The simplest page-replacement algorithm is a first-in, first-out (FIFO) algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. Notice that it is not strictly necessary to record the time when a page is brought in. We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue.

For our example reference string, our three frames are initially empty. The first three references (7, 0, 1) cause page faults and are brought into these empty frames. The next reference (2) replaces page 7, because page 7 was brought in first. Since 0 is the next reference and 0 is already in memory, we have no fault for this reference. The first reference to 3 results in replacement of page 0, since it is now first in line. Because of this replacement, the next reference, to 0, will fault. Page 1 is then replaced by page 0. Every time a fault occurs, we show which pages are in our three frames. There are fifteen faults altogether.

The FIFO page-replacement algorithm is easy to understand and program. However, its performance is not always good. On the one hand, the page replaced may be an initialization module that was used a long time ago and is no longer needed. On the other hand, it could contain a heavily used variable that was initialized early and is in constant use.

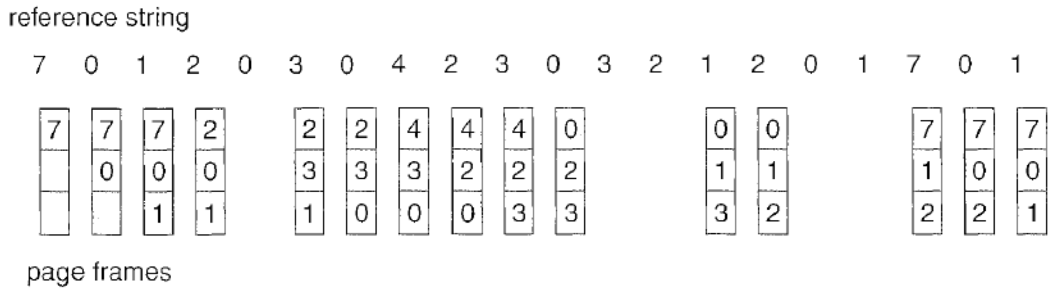


Fig: FIFO page-replacement algorithm.

Notice that, even if we select for replacement a page that is in active use, everything still works correctly. After we replace an active page with a new one, a fault occurs almost immediately to retrieve the active page. Some other page must be replaced to bring the active page back into memory. Thus, a bad replacement choice increases the page-fault rate and slows process execution. It does not, however, cause incorrect execution.

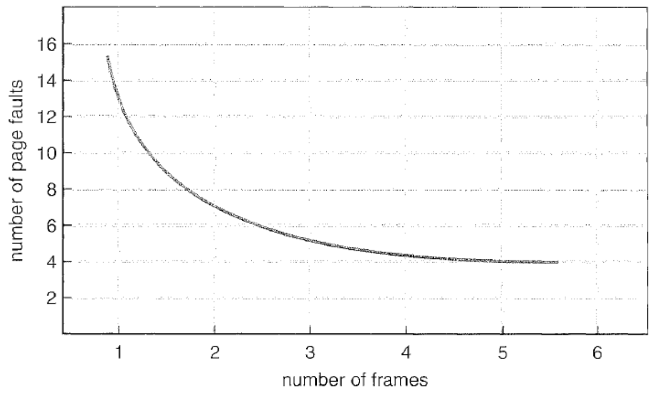


Fig: Graph of page faults versus number of frames.

To illustrate the problems that are possible with a FIFO page-replacement algorithm, we consider the following reference string:
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
Notice that the number of faults for four frames (ten) is *greater* than the number of faults for three frames (nine)! This most unexpected result is known as Belady's Anomaly: for some page-replacement algorithms, the page-fault rate may *increase* as the number of allocated frames increases. We would expect that giving more memory to a process would improve its performance. In some early research, investigators noticed that this assumption was not always true. Belady's anomaly was discovered as a result.

Optimal Page Replacement

One result of the discovery of Belady's anomaly was the search for an optimal page replacement algorithm, which has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly. Such an algorithm does exist and has been called OPT or MIN. It is simply this:

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

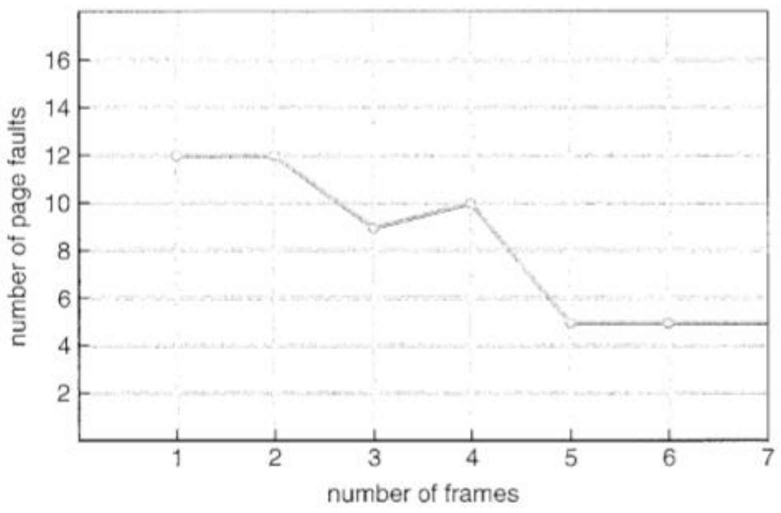


Fig: Page-fault curve for FIFO replacement on a reference string.

Use of this page-replacement algorithm guarantees the lowest possible page fault rate for a fixed number of frames. For example, on our sample reference string, the optimal page-replacement algorithm would yield nine page faults, as shown in Figure 9.14. 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, whereas page 0 will be used at 5, and page 1 at 14. The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again. With only nine page faults, optimal replacement is much better than a FIFO algorithm, which results in fifteen faults. (If we ignore the first three, which all algorithms must suffer, then optimal replacement is twice as good as FIFO replacement.) In fact, no replacement algorithm can process this reference string in three frames with fewer than nine faults. Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string.

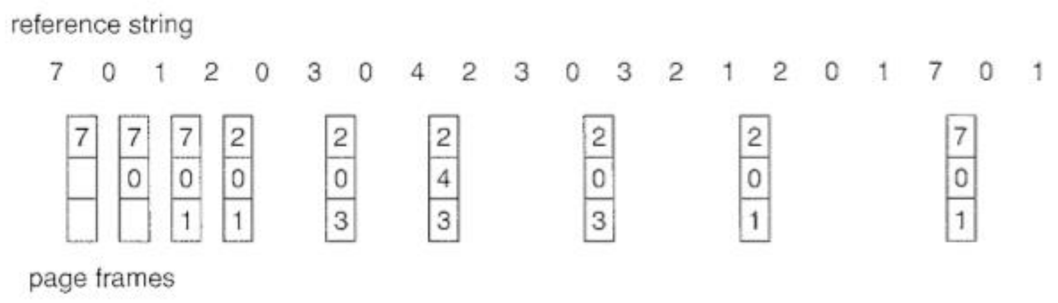


Fig: Optimal page-replacement algorithm

LRU Page Replacement

lf the optimal algorithm is not feasible, perhaps an approximation of the optimal algorithm is possible. The key distinction between the FIFO and OPT algorithms (other than looking backward versus forward in time) is that the FIFO algorithm uses the time when a page was brought into memory, whereas the OPT algorithm uses the time when a page is to be used. If we use the recent past as an approximation of the near future, then 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. We can think of this strategy as the optimal page-replacement algorithm looking backward in time, rather than forward. The result of applying LRU replacement to our example reference string is shown in Figure.

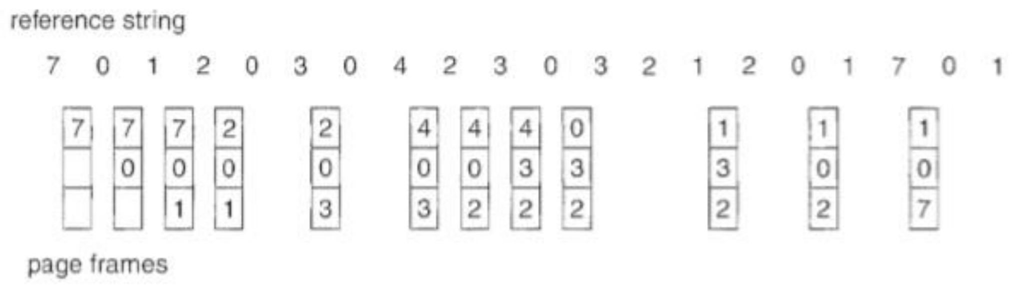


Fig: LRU page-replacement algorithm

The LRU algorithm produces twelve faults. Notice that 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. Despite these problems, LRU replacement with twelve faults is much better than FIFO replacement with fifteen. The LRU policy is often used as a page-replacement algorithm and is considered to be good. The major problem is how to implement LRU replacement. An LRU page-replacement algorithm may require substantial hardware assistance. The problem is to determine an order for the frames defined by the time of last use. Two implementations are feasible:

Counters. In the simplest case, we associate with each page-table entry a time-of-use field and add to the CPU a logical clock or counter. The clock is incremented for every memory reference. Whenever a reference to a page is made, the contents of the clock register are copied to the ti1ne-of-use field in the page-table entry for that page. In this way, we always have the "time" of the last reference to each page. We replace the page with the smallest time value. This scheme requires a search of the page table to find the LRU page and a write to memory (to the time-of-use field in the page table) for each memory access. The times must also be maintained when page tables are changed (due to CPU scheduling). Overflow of the clock must be considered.

Stack. Another approach to implementing LRU replacement is to keep a stack of page numbers. Whenever a page is referenced, it is removed from the stack and put on the top. In this way, the most recently used page is always at the top of the stack and the least recently used page is always at the bottom. Because entries must be removed from the middle of the stack, it is best to implement this approach by using a doubly linked list with a head pointer and a tail pointer. Removing a page and putting it on the top of the stack then requires changing six pointers at worst. Each update is a little more expensive, but there is no search for a replacement; the tail pointer points to the bottom of the stack, which is the LRU page. This approach is particularly appropriate for software or microcode implementations of LRU replacement.

Like optimal replacement, LRU replacement does not suffer from Belady's anomaly. Both belong to a class of page-replacement algorithms, called stack algorithm, that can never exhibit Belady's anomaly. A stack algorithm is an algorithm for which it can be shown that the set of pages in memory for n frames is always a subset of the set of pages that would be in memory with n + 1 frames. For LRU replacement, the set of pages in memory would be the n most recently referenced pages. If the number of frames is increased, these n pages will still be the most recently referenced and so will still be in memory. Note that neither implementation of LRU would be conceivable without hardware assistance beyond the standard TLB registers. The updating of the clock fields or stack must be done for every memory reference. If we were to use an interrupt for every reference to allow software to update such data structures, it would slow every memory reference by a factor of at least ten, hence slowing every user process by a factor of ten. Few systems could tolerate that level of overhead for memory management.

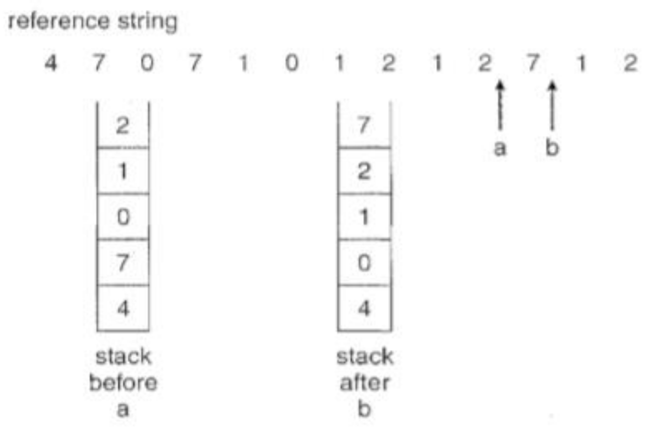


Fig: Use of a stack to record the most recent page references

LRU-Approximation Page Replacement

Few computer systems provide sufficient hardware support for true LRU page replacement. Some systems provide no hardware support, and other page replacement algorithms (such as a FIFO algorithm) must be used. Many systems provide some help, however, in the form of a reference bit. The reference bit for a page is set by the hardware whenever that page is referenced (either a read or a write to any byte in the page). Reference bits are associated with each entry in the page table. Initially, all bits are cleared (to 0) by the operating system. As a user process executes, the bit associated with each page referenced is set (to 1) by the hardware. After some time, we can determine which pages have been used and which have not been used by examining the reference bits, although we do not know the order of use. This information is the basis for many page-replacement algorithms that approximate LRU replacement.

Additional-Reference-Bits Algorithm

We can gain additional ordering information by recording the reference bits at regular intervals. We can keep an 8-bit byte for each page in a table in memory. At regular intervals (say, every 100 milliseconds), a timer interrupt transfers control to the operating system. The operating system shifts the reference bit for each page into the high-order bit of its 8-bit byte, shifting the other bits right by 1 bit and discarding the low-order bit. These 8-bit shift registers contain the history of page use for the last eight time periods. If the shift register contains 00000000, for example, then the page has not been used for eight time periods; a page that is used at least once in each period has a shift register value of 11111111. A page with a history register value of 11000100 has been used more recently than one with a value of 01110111. If we interpret these 8-bit bytes as unsigned integers, the page with the lowest number is the LRU page, and it can be replaced. Notice that the numbers are not guaranteed to be unique, however. We can either replace (swap out) all pages with the smallest value or use the FIFO method to choose among them. The number of bits of history included in the shift register can be varied, of course, and is selected (depending on the hardware available) to make the updating as fast as possible. In the extreme case, the number can be reduced to zero, leaving only the reference bit itself. This algorithm is called the second-chance page-replacement algorithm.

Second-Chance Algorithm

The basic algorithm of second-chance replacement is a FIFO replacement algorithm. When a page has been selected, however, we inspect its reference bit. If the value is 0, we proceed to replace this page; but if the reference bit is set to 1, we give the page a second chance and move on to select the next FIFO page. When a page gets a second chance, its reference bit is cleared, and its arrival time is reset to the current time. Thus, a page that is given a second chance will not be replaced until all other pages have been replaced (or given second chances). In addition, if a page is used often enough to keep its reference bit set, it will never be replaced.

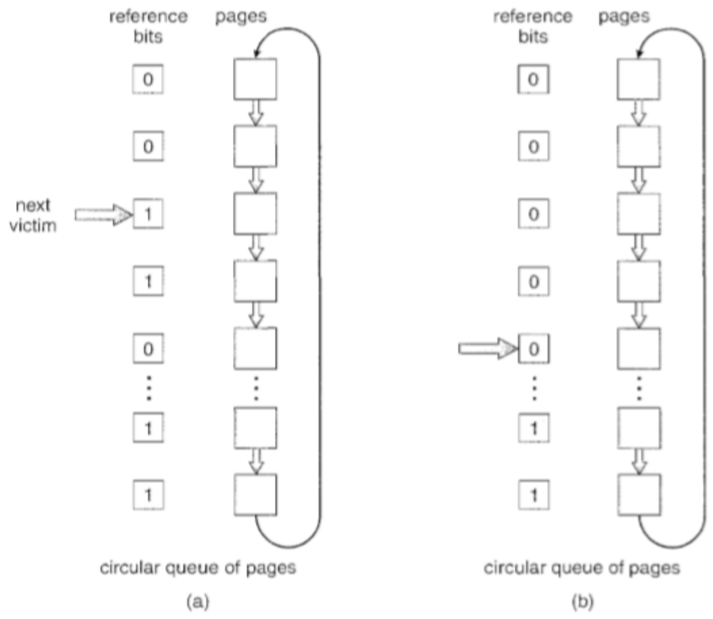


Fig: Second-chance (clock) page-replacement algorithm

One way to implement the second-chance algorithm (sometimes referred to as the clock algorithm) is as a circular queue. A poi11ter (that is, a hand on the clock) indicates which page is to be replaced next. When a frame is needed, the pointer advances until it finds a page with a 0 reference bit. As it advances, it clears the reference bits (Figure 9.17). Once a victim page is found, the page is replaced, and the new page is inserted in the circular queue in that position. Notice that, in the worst case, when all bits are set, the pointer cycles through the whole queue, giving each page a second chance. It clears all the reference bits before selecting the next page for replacement. Second-chance replacement degenerates to FIFO replacement if all bits are set.

Enhanced Second-Chance Algorithm

We can enhance the second-chance algorithm by considering the reference bit and the modify bit as an ordered pair. With these two bits, we have the following four possible classes:

1. (0, 0) neither recently used nor modified -best page to replace
2. (0, 1) not recently used hut modified-not quite as good, because the page will need to be written out before replacement
3. (1, 0) recently used but clean-probably will be used again soon
4. (1, 1) recently used and modified -probably will be used again soon, and the page will be need to be written out to disk before it can be replaced

Each page is in one of these four classes. When page replacement is called for, we use the same scheme as in the clock algorithm; but instead of examining whether the page to which we are pointing has the reference bit set to 1, we examine the class to which that page belongs. We replace the first page encountered in the lowest nonempty class. Notice that we may have to scan the circular queue several times before we find a page to be replaced. The major difference between this algorithm and the simpler clock algorithm is that here we give preference to those pages that have been modified to reduce the number of I/O s required.

Counting-Based Page Replacement

There are many other algorithms that can be used for page replacement. For example, we can keep a counter of the number of references that have been made to each page and develop the following two schemes.

- The least frequently used (LFU) page-replacement algorithm requires that the page with the smallest count be replaced. The reason for this selection is that an actively used page should have a large reference count. A problem arises, however, when a page is used heavily during the initial phase of a process but then is never used again. Since it was used heavily, it has a large count and remains in memory even though it is no longer needed. One solution is to shift the counts right by 1 bit at regular intervals, forming an exponentially decaying average usage count.
- The most frequently used (MFU) page-replacement algorithm is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

As you might expect, neither MFU nor LFU replacement is common. The implementation of these algorithms is expensive, and they do not approximate OPT replacement well.

Page-Buffering Algorithms

Other procedures are often used in addition to a specific page-replacement algorithm. For example, systems commonly keep a pool of free frames. When a page fault occurs, a victim frame is chosen as before. However, the desired page is read into a free frame from the pool before the victim is written out. This procedure allows the process to restart as soon as possible, without waiting for the victim page to be written out. When the victim is later written out, its frame is added to the free-frame pool. An expansion of this idea is to maintain a list of modified pages. Whenever the paging device is idle, a modified page is selected and is written to the disk. Its modify bit is then reset. This scheme increases the probability that a page will be clean when it is selected for replacement and will not need to be written out. Another modification is to keep a pool of free frames but to remember which page was in each frame. Since the frame contents are not modified when a frame is written to the disk, the old page can be reused directly fronc the free-frame pool if it is needed before that frame is reused. No I/O is needed in this case. When a page fault occurs, we first check whether the desired page is in the free-frame pool. If it is not, we must select a free frame and read into it. This technique is used in the VAX/VMS system along with a FIFO replacement algorithm. When the FIFO replacement algorithm mistakenly replaces a page that is still in active use, that page is quickly retrieved from the free-frame pool, and no I/O is necessary. The free-frame buffer provides protection against the relatively poor, but simple, FIFO replacement algorithm. This method is necessary because the early versions of VAX did not implement the reference bit correctly. Some versions of the UNIX system use this method in conjunction with the second-chance algorithm. It can be a useful augmentation to any page replacement algorithm, to reduce the penalty incurred if the wrong victim page is selected.

Allocation of Frames

How do we allocate the fixed amount of free memory among the various processes? If we have 93 free frames and two processes, how many frames does each process get? 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 list was exhausted, a page-replacement algorithm would 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. There are many variations on this simple strategy. We can require that the operating system allocate all its buffer and table space from the free-frame list. When this space is not in use by the operating system, it can be used to support user paging. We can try to keep three free frames reserved on the free-frame list at all times. Thus, when a page fault occurs, there is a free frame available to page into. While the page swap is taking place, a replacement can be selected, which is then written to the disk as the user process continues to execute. Other variants are also possible, but the basic strategy is clear: the user process is allocated any free frame.

Minimum Number of Frames

Our strategies for the allocation of frames are constrained in various ways. We cannot, for example, allocate more than the total number of available frames (unless there is page sharing). We must also allocate at least a minimum number of frames. Here, we look more closely at the latter requirement. One reason for allocating at least a minimum number of frames involves performance. Obviously, as the number of frames allocated to each process decreases, the page-fault rate increases, slowing process execution. In addition, remember that when a page fault occurs before an executing instruction is complete, the instruction must be restarted. Consequently.

we must have enough frames to hold all the different pages that any single instruction can reference. For example, consider a machine in which all memory-reference instructions may reference only one memory address. In this case, we need at least one frame for the instruction and one frame for the memory reference. In addition, if one-level indirect addressing is allowed (for example, a load instruction on page 16 can refer to an address on page 0, which is an indirect reference to page 23), then paging requires at least three frames per process. Think about what might happen if a process had only two frames. The minimum number of frames is defined by the computer architecture. For example, the move instruction for the PDP-11 includes more than one word for some addressing modes, and thus the instruction itself may straddle two pages. In addition, each of its two operands may be indirect references, for a total of six frames. Another example is the IBM 370 MVC instruction. Since the instruction is from storage location to storage location, it takes 6 bytes and can straddle two pages. The block of characters to move and the area to which it is to be moved can each also straddle two pages. This situation would require six frames. The worst case occurs when the MVC instruction is the operand of an EXECUTE instruction that straddles a page boundary; in this case, we need eight frames. The worst-case scenario occurs in computer architectures that allow multiple levels of indirection (for example, each 16-bit word could contain a 15-bit address plus a 1-bit indirect indicator). Theoretically, a simple load instruction could reference an indirect address that could reference an indirect address (on another page) that could also reference an indirect address (on yet another page), and so on, until every page in virtual memory had been touched. Thus, in the worst case, the entire virtual memory must be in physical memory. To overcome this difficulty, we must place a limit on the levels of indirection (for example, limit an instruction to at most 16levels of indirection). When the first indirection occurs, a counter is set to 16; the counter is then decremented for each successive indirection for this instruction. If the counter is decremented to 0, a trap occurs (excessive indirection). This limitation reduces the maximum number of memory references per instruction to 17, requiring the same number of frames. Whereas the minimum number of frames per process is defined by the architecture, the maximum number is defined by the amount of available physical memory. In between, we are still left with significant choice in frame allocation.

Allocation Algorithms

The easiest way to split m frames among n processes is to give everyone an equal share, m/n frames. For instance, if there are 93 frames and five processes, each process will get 18 frames. The three leftover frames can be used as a free-frame buffer pool. This scheme is called equal allocation. An alternative is to recognize that various processes will need differing amounts of memory. Consider a system with a 1-KB frame size. If a small student process of 10 KB and an interactive database of 127 KB are the only two processes running in a system with 62 free frames, it does not make much sense to give each process 31 frames. The student process does not need more than 10 frames, so the other 21 are, strictly speaking, wasted. To solve this problem, we can use proportional allocation, in which we allocate available memory to each process according to its size.

CHAPTER 6

FILE SYSTEM

FILE CONCEPT

Computers can store information on various storage media, such as magnetic disks, magnetic tapes, and optical disks. So that the computer system will be convenient to use, the operating system provides a uniform logical view of information storage. The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, the *file.* Files are mapped by the operating system onto physical devices. These storage devices are usually nonvolatile, so the contents are persistent through power failures and system reboots.
A file is a named collection of related information that is recorded on secondary storage. From a user's perspective, a file is the smallest allotment of logical secondary storage; that is, data cannot be written to secondary storage unless they are within a file. Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic, alphanumeric, or binary. Files may be free form, such as text files, or may be formatted rigidly. In general, a file is a sequence of bits, bytes, lines, or records, the meaning of which is defined by the file's creator and user. The concept of a file is thus extremely general.
The information in a file is defined by its creator. Many different types of information may be stored in a file-source programs, object programs, executable programs, numeric data, text, payroll records, graphic images, sound recordings, and so on. *A* file has a certain defined structure which depends on its type. *A text* file is a sequence of characters organized into lines (and possibly pages). *A source* file is a sequence of subroutines and functions, each of which is further organized as declarations followed by executable statements. An *object* file is a sequence of bytes organized into blocks understandable by the system's linker. An *executable* file is a series of code sections that the loader can bring into memory and execute.

File Attributes

A file is named, for the convenience of its human users, and is referred to by its name. *A* name is usually a string of characters, such as *example.c.* Some systems differentiate between uppercase and lowercase characters in names, whereas other systems do not. When a file is named, it becomes independent of the process, the user, and even the system that created it. For instance, one user might create the file *example.c*, and another user might edit that file by specifying its name. The file's owner might write the file to a floppy disk, send it in an e-mail, or copy it across a network, and it could still be called *example.c* on the destination system.
A file's attributes vary from one operating system to another but typically consist of these:

- **Name**. The symbolic file name is the only information kept in human readable form.
- **Identifier**. This unique tag, usually a number, identifies the file within the file system; it is the non-human-readable name for the file.
- **Type**. This information is needed for 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. These data can be useful for protection, security, and usage monitoring.

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

File Operations

A file is an abstract datatype. To define a file properly, we need to consider the operations that can be performed on files. The operating system can provide system calls to create, write, read, reposition, delete, and truncate files.

Creating a file. Two steps are necessary to create a file. First, space in the file system must be found for the file. We discuss how to allocate space for the file in Chapter 11. Second, 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. Given the name of the file, the system searches the directory to find the file's location.

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

Reading a file. To read from a file, we use a system call that specifies the name of the file and where (in memory) the next block of the file should be put. Again, the directory is searched for the associated entry, and the system needs to keep a *read* pointer to the location in the file where the next read is to take place. Once the read has taken place, the read pointer is updated. Because a process is usually either reading from or writing to a file, the current operation location can be kept as a per-process current-file-position-pointer. Both the read and write operations use this same pointer, saving space and reducing system complexity.

Repositioning within a file. The directory is searched for the appropriate entry, and the current-file-position pointer is repositioned to a given value. Repositioning within a file need not involve any actual I/0. This file operation is also known as a file *seek.*

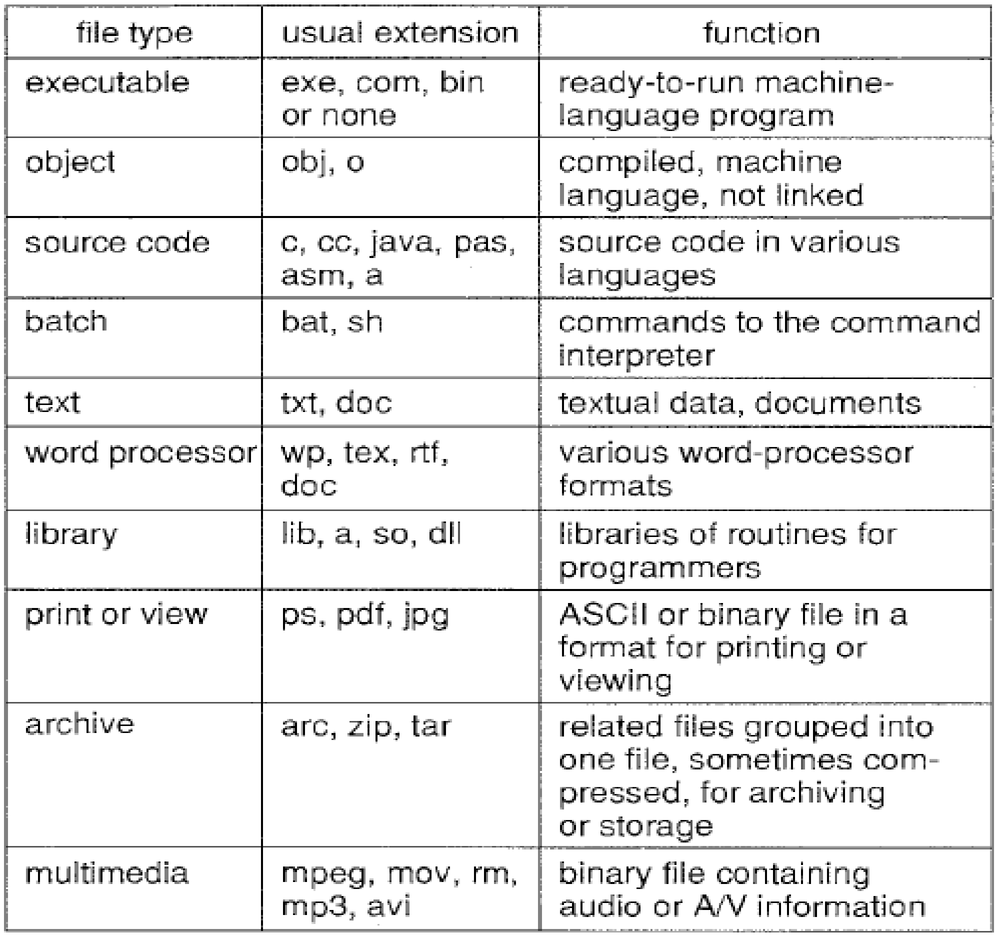
Deleting a file. To delete a file, we search the directory for the named file. Having found the associated directory entry, we release all file space, so that it can be reused by other files and erase the directory entry.

Truncating a file. The user may want to erase the contents of a file but keep its attributes. Rather than forcing the user to delete the file and then recreate it, this function allows all attributes to remain unchanged -except for file length-but lets the file be reset to length zero and its file space released.

File Types

The system uses the extension to indicate the type of the file and the type of operations that can be done on that file. Only a file with a *.com, .exe,* or *.bat* extension can be *executed,* for instance.

The *.com* and *.exe* files are two forms of binary executable files, whereas a *.bat* file is a batch file containing, in ASCII format, commands to the operating system. MS-DOS recognizes only a few extensions, but application programs also use extensions to indicate file types in which they are interested. For example, assemblers expect source files to have an *.asm* extension, and the Microsoft Word word processor expects its files to end with a *.doc* extension.



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

Fig: Common file types

File Structure

File types also can be used to indicate the internal structure of the file. Certain files must conform to a required structure that is understood by the operating system. For example, the operating system requires that an executable file have a specific structure so that it can determine where in memory to load the file and what the location of the first instruction is.

Internal File Structure

Internally, locating an offset within a file can be complicated for the operating system. Disk systems typically have a well-defined block size determined by the size of a sector. All disk I/0 is performed in units of one block (physical record), and all blocks are the same size. It is unlikely that the physical record size will exactly match the length of the desired logical record. Logical records may even vary in length.

The UNIX operating system defines all files to be simply streams of bytes. Each byte is individually addressable by its offset from the begi1ming (or end) of the file. In this case, the logical record size is 1 byte. The file system automatically packs and unpacks bytes into physical disk blocks say, 512 bytes per block-as necessary. The waste incurred to keep everything in units of blocks (instead of bytes) is internal fragmentation.

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. Some systems provide only one access method for files. Other systems, such as those of IBM, support many access methods, and choosing the right one for a particular application is a major design problem.

Sequential Access

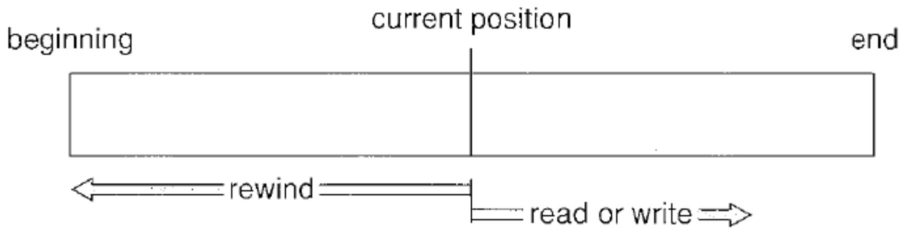
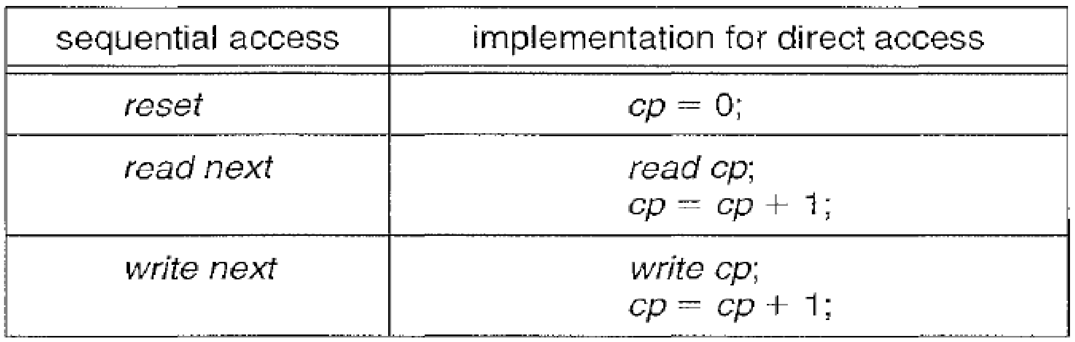


Fig: Sequential-access file.

The simplest access method is sequential access. Information in the file is processed in order, one record after the other. This mode of access is by far the most common; for example, editors and compilers usually access files in this fashion. 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; and on some systems, a program may be able to skip forward or backward *n* records for some integer n-perhaps only for *n* = 1.

Direct Access

Another method is direct access (or relative access). A file is made up of fixed length logical records that allow programs to read and write records rapidly in no particular order. The direct-access method is based on a disk model of a file, since disks allow random access to any file block. 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.
Direct-access files are of great use for immediate access to large amounts of information. 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.



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

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

Other Access Methods

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

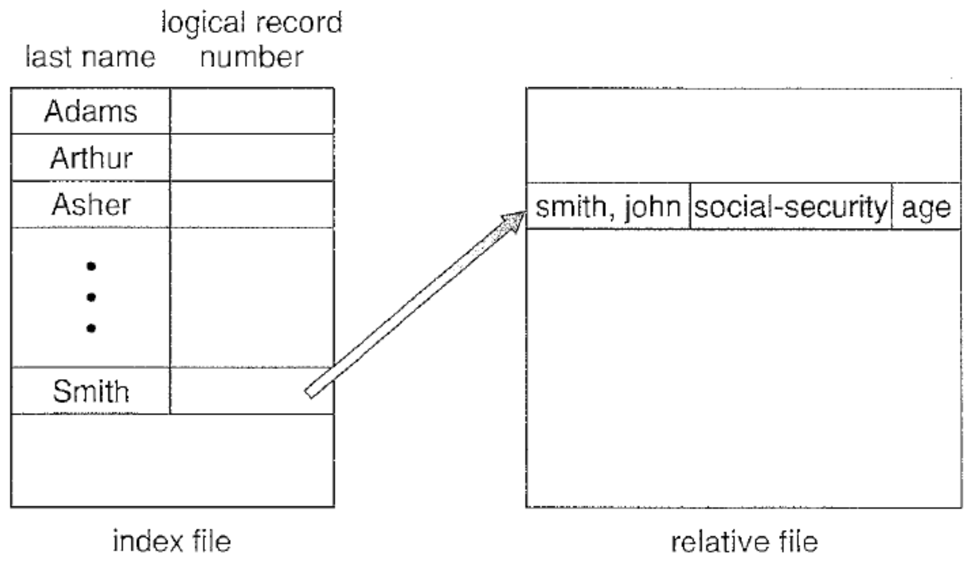


Fig : Example of index and relative files

DIRECTORY AND DISK STRUCTURE

Next, we consider how to store files. Certainly, no general-purpose computer stores just one file. There are typically thousand, millions, and even billions of files within a computer. Files are stored on random-access storage devices, including hard disks, optical disks, and solid state (memory-based) disks.

Partitioning is useful for limiting the sizes of individual file systems, putting multiple file-system types on the same device, or leaving part of the device available for other uses, such as swap space or unformatted (RAW) disk space. Partitions are also known as Slices or (in the IBM world) minidisks. A file system can be created on each of these parts of the disk. Each volume that contains a file system must also contain information about the files in the system. This information is kept in entries in a device directory or volume table of contents. The device directory (more commonly known simply as that directory) records information -such as name, location, size, and type-for all files on that volume.

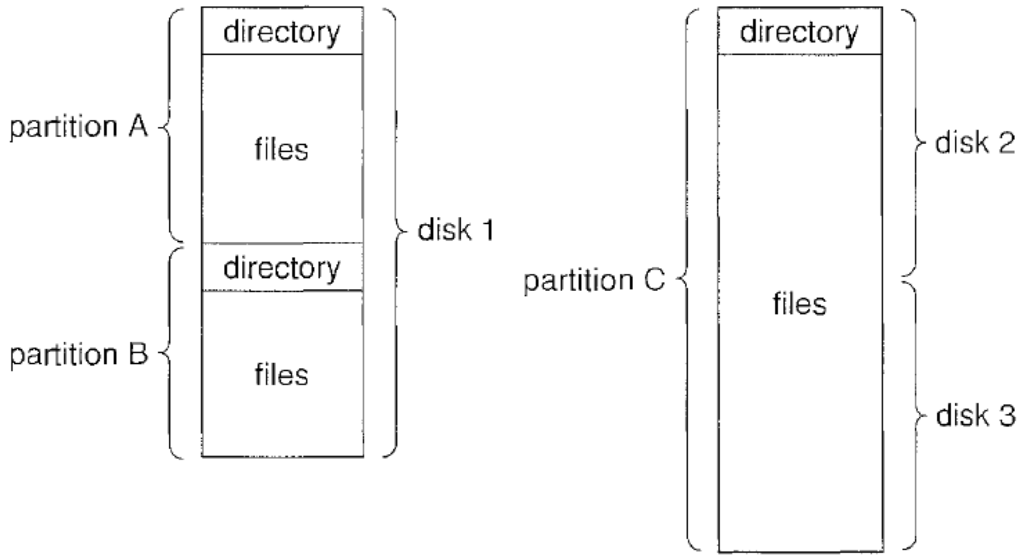


Fig : A typical file-system organization

Storage Structure

A general-purpose computer system has multiple storage devices, and those devices can be sliced up into volumes that hold file systems. Computer systems may have zero or more file systems, and the file systems may be of varying types. The file systems of computers, then, can be extensive. Even within a file system, it is useful to segregate files into groups and manage and act on those groups. This organization involves the use of directories.

Directory Overview

The directory can be viewed as a symbol table that translates file names into their directory entries. If we take such a view, we see that the directory itself can be organized in many ways. We want to be able to insert entries, to delete entries, to search for a named entry, and to list all the entries in the directory.
When considering a particular directory structure, we need to keep in mind the operations that are to be performed on a directory:

- **Search for a file**. We need to be able to search a directory structure to find the entry for a particular file. Since files have symbolic names, and similar names may indicate a relationship between files, we may want to be able to find all files whose names match a particular pattern.
- **Create a file**. New files need to be created and added to the directory.
- **Delete a file**. When a file is no longer needed, we want to be able to remove it from the directory.
- **List a directory**. We need to be able to list the files in a directory and the contents of the directory entry for each file in the list.
- **Rename a file**. Because the name of a file represents its contents to its users, we must be able to change the name when the contents or use of the file changes. Renaming a file may also allow its position within the directory structure to be changed.
- **Traverse the file system**. We may wish to access every directory and every file within a directory structure. For reliability, it is a good idea to save the contents and structure of the entire file system at regular intervals. Often, we do this by copying all files *to* magnetic tape. This technique provides a backup copy in case of system failure. In addition, if a file is no longer in use, the file can be copied *to* tape and the disk space of that file released for reuse by another file.

Single-level Directory

The simplest directory structure is the single-level directory. All files are contained in the same directory, which is easy to support and understand.

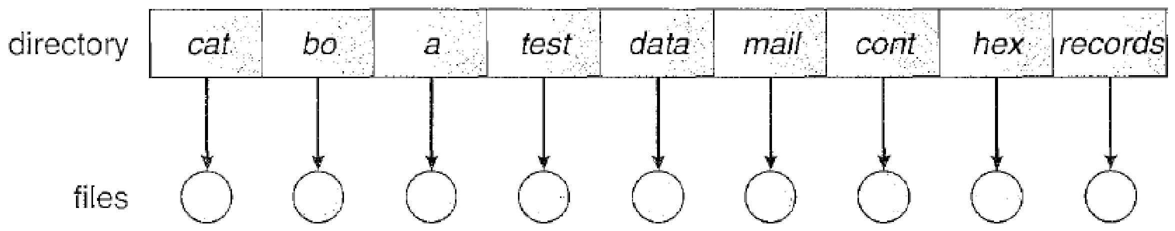


Fig : Single-level directory.

Two-Level Directory
As we have seen, a single-level directory often leads to confusion of file names among different users. The standard solution is to create a *separate* directory for each user.
In the two-level directory structure, each user has his own user file directory (UFD). The UFDs have similar structures, but each lists only the files of a single user. W11en a user job starts or a user logs in, the system's master file directory (MFD) is searched. The MFD is indexed by user name or account number, and each entry points to the UFD for that user.

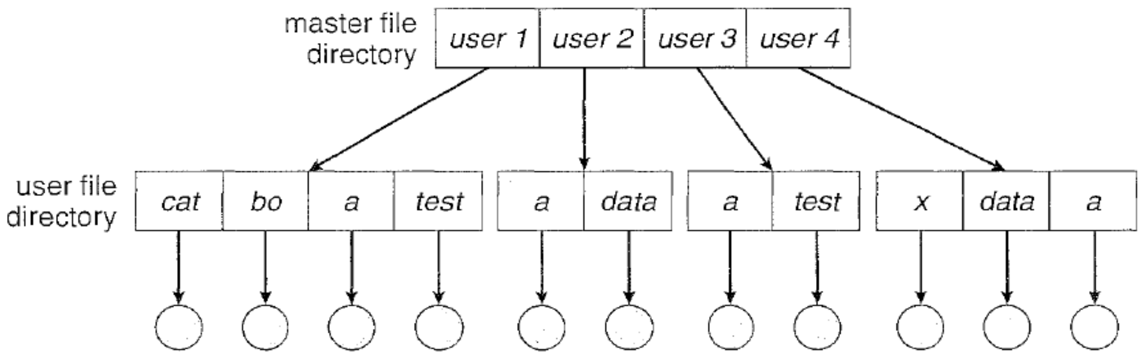


Fig : Two level directory structure.

Tree-Structured Directories
Once we have seen how to view a two-level directory as a two-level tree, the natural generalization is to extend the directory structure to a tree of arbitrary height. This generalization allows users to create their own subdirectories and to organize their files accordingly. A tree is the most common directory structure. The tree has a root directory, and every file in the system has a unique path name.

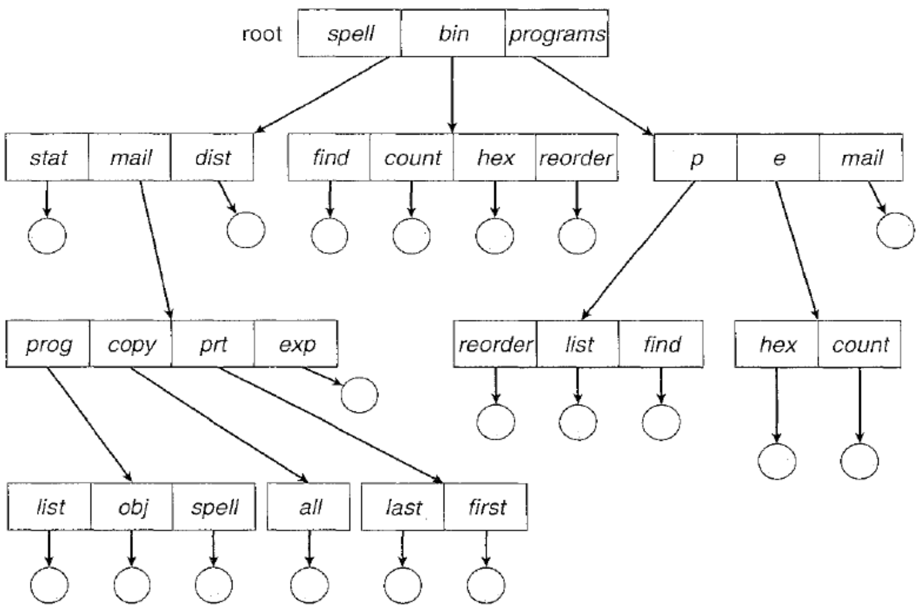


fig: Tree-structured directory structure

A directory (or subdirectory) contains a set of files or subdirectories. A directory is simply another file, but it is treated in a special way. 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). Special system calls are used to create and delete directories. In normal use, each process has a current directory. The current directory should contain most of the files that are of current interest to the process.

When reference is made to a file, the current directory is searched. If a file is needed that is not in the current directory, then the user usually must either specify a path name or change the current directory to be the directory holding that file. To change directories, a system call is provided that takes a directory name as a parameter and uses it to redefine the current directory. Thus, the user can change his current directory whenever he desires. From one change directory system call to the next, all open system calls search the current directory for the specified file.

Acyclic-Graph Directories

Consider two programmers who are working on a joint project. The files associated with that project can be stored in a subdirectory, separating them from other projects and files of the two programmers. But since both programmers are equally responsible for the project, both want the subdirectory to be in their own directories. The common subdirectory should be *shared.* A shared directory or file will exist in the file system in two (or more) places at once.

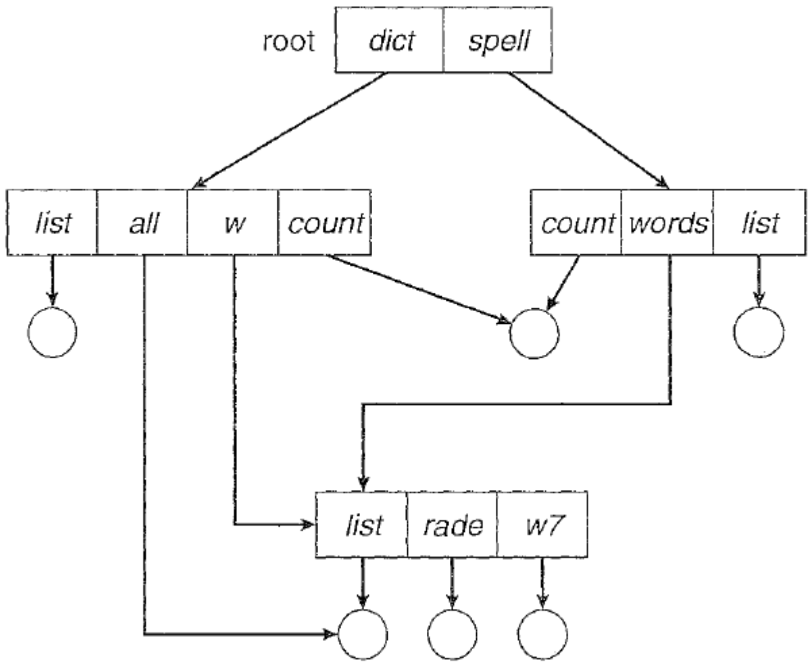


Fig: Acyclic-graph directory structure

General Graph Directory

A serious problem with using an acyclic-graph structure is ensuring that there are no cycles. If we start with a two-level directory and allow users to create subdirectories, a tree-structured directory results. It should be fairly easy to see that simply adding new files and subdirectories to an existing tree-structured directory preserves the tree-structured nature. However, when we add links, the tree structure is destroyed, resulting in a simple graph structure. The primary advantage of an acyclic graph is the relative simplicity of the algorithms to traverse the graph and to determine when there are no more references to a file. We want to avoid traversing shared sections of an acyclic graph twice, mainly for performance reasons. If we have just searched a major shared subdirectory for a particular file without finding it, we want to avoid searching that subdirectory again; the second search would be a waste of time. If cycles are allowed to exist in the directory, we likewise want to avoid searching any component twice, for reasons of correctness as well as performance. A poorly designed algorithm might result in an infinite loop continually searching through the cycle and never terminating. One solution is to limit arbitrarily the number of directories that will be accessed during a search.

A similar problem exists when we are trying to determine when a file can be deleted. With acyclic-graph directory structures, a value of 0 in the reference count means that there are no more references to the file or directory, and the file can be deleted. However, when cycles exist, the reference count may not be 0 even when it is no longer possible to refer to a directory or file.

This anomaly results from the possibility of self-referencing (or a cycle) in the directory structure. In this case, we generally need to use a garbage-collection scheme to determine when the last reference has been deleted and the disk space can be reallocated. Garbage collection involves traversing the entire file system, marking everything that can be accessed. Then, a second pass collects everything that is not marked onto a list of free space. (A similar marking procedure can be used to ensure that a traversal or search will cover everything in the file system once and only once.) Garbage collection for a disk-based file system, however, is extremely time consuming and is thus seldom attempted.

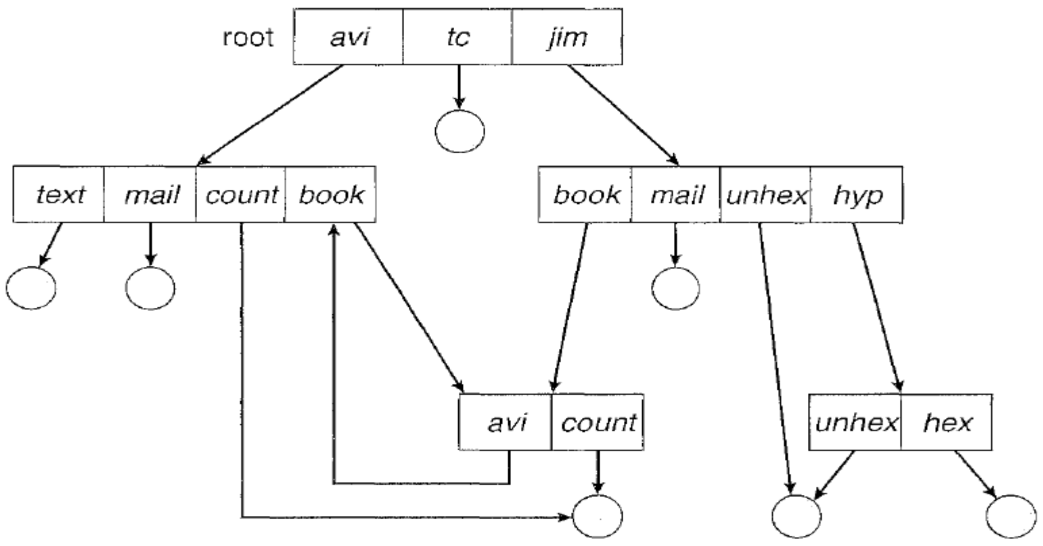


Fig : General graph directory

File System Mounting

Just as a file must be *opened* before it is used, a file system must be *mounted* before it can be available to processes on the system. More specifically, the directory structure may be built out of multiple volumes, which must be mounted to make them available within the file-system name space.

The mount procedure is straightforward. The operating system is given the name of the device and the mount point – the location within the file structure where the file system is to be attached. Some operating systems require that a file system type be provided, while others inspect the structures of the device and determine the type of file system. Typically, a mount point is an empty directory. For instance, on a UNIX system, a file system containing a user's home directories might be mounted as */home;* then, to access the directory structure within that file system, we could precede the directory names with */home,* as in */home/jane.* Mounting that file system under */users* would result in the path name */users/jane,* which we could use *to* reach the same directory. Next, the operating system verifies that the device contains a valid file system. It does so by asking the device driver to read the device directory and verifying that the directory has the expected format. Finally, the operating system notes in its directory structure that a file system is mounted at the specified mount point. This scheme enables the operating system to traverse its directory structure, switching among file systems, and even file systems of varying types, as appropriate.

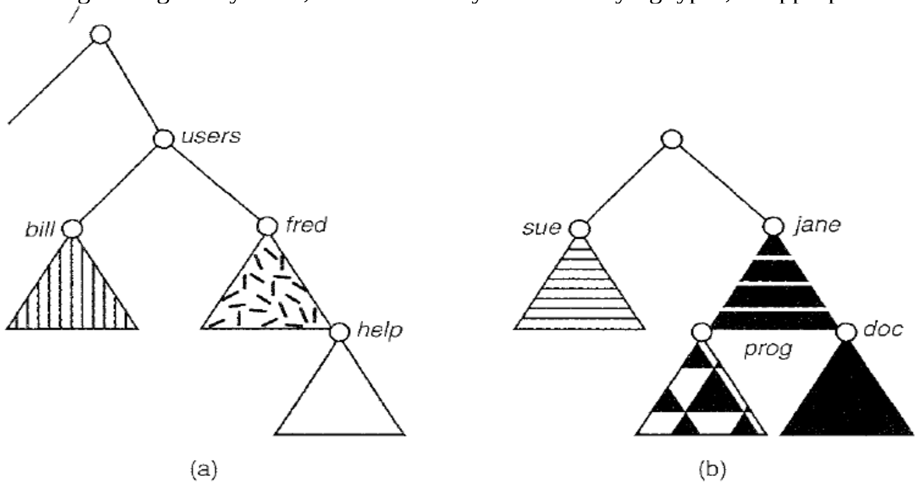


Fig: File system. (a) Existing system. (b) Unmounted volume

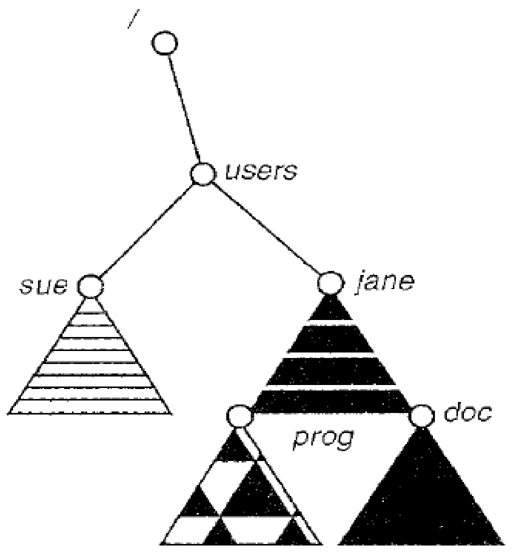


Fig: Mount point

File Sharing

Multiple Users: When an operating system accommodates multiple users, the issues of file sharing, file naming, and file protection become preeminent. Given a directory structure that allows files to be shared by users, the system must mediate the file sharing. The system can either allow a user to access the files of other users by default or require that a user specifically grant access to the files. These are the issues of access control and protection. The owner and group IDs of a given file (or directory) are stored with the other file attributes. When a user requests an operation on a file, the user ID can be compared with the owner attribute to determine if the requesting user is the owner of the file. Likewise, the group IDs can be compared. The result indicates which permissions are applicable. The system then applies those permissions to the requested operation and allows or denies it. Many systems have multiple local file systems, including volumes of a single disk or multiple volumes on multiple attached disks. In these cases, the ID checking and permission matching are straightforward, once the file systems are mounted.

Remote File Systems: With the advent of networks, communication among remote computers became possible. Networking allows the sharing of resources spread across a campus or even around the world. One obvious resource to share is data in the form of files.

Through the evolution of network and file technology, remote file-sharing methods have changed. The first implemented method involves manually transferring files between machines via programs like ftp. The second major method uses a distributed file system (DFS) in which remote directories are visible from a local machine. In some ways, the third method, the World Wide Web, is a reversion to the first. A browser is needed to gain access to the remote files, and separate operations (essentially a wrapper for ftp) are used to transfer files.

The Client-Server Model: Remote file systems allow a computer to mom1.t one or more file systems from one or more remote machines. In this case, the machine containing the files is the *server,* and the machine seeking access to the files is the *client.* The client-server relationship is common with networked machines. Generally, the server declares that a resource is available to clients and specifies exactly which resource (in this case, which files) and exactly which clients. A server can serve multiple clients, and a client can use multiple servers, depending on the implementation details of a given client-server facility.

The server usually specifies the available files on a volume or directory level. Client identification is more difficult. A client can be specified by a network name or other identifier, such as an *IP address,* but these can be spoofed, or imitated.

Distributed Information Systems

To make client-server systems easier to manage, distributed information system, also known as distributed naming services, provide unified access to the information needed for remote computing. The Domain Name System (DNS) provides host-name-to-network-address translations for the entire Internet (including the World Wide Web). Before DNS became widespread, files containing the same information were sent via e-mail or ftp between all networked hosts. This methodology was not scalable.
Other distributed information systems provide *user name/password/user ID/group ID* space for a distributed facility. UNIX systems have employed a wide variety of distributed-information methods.

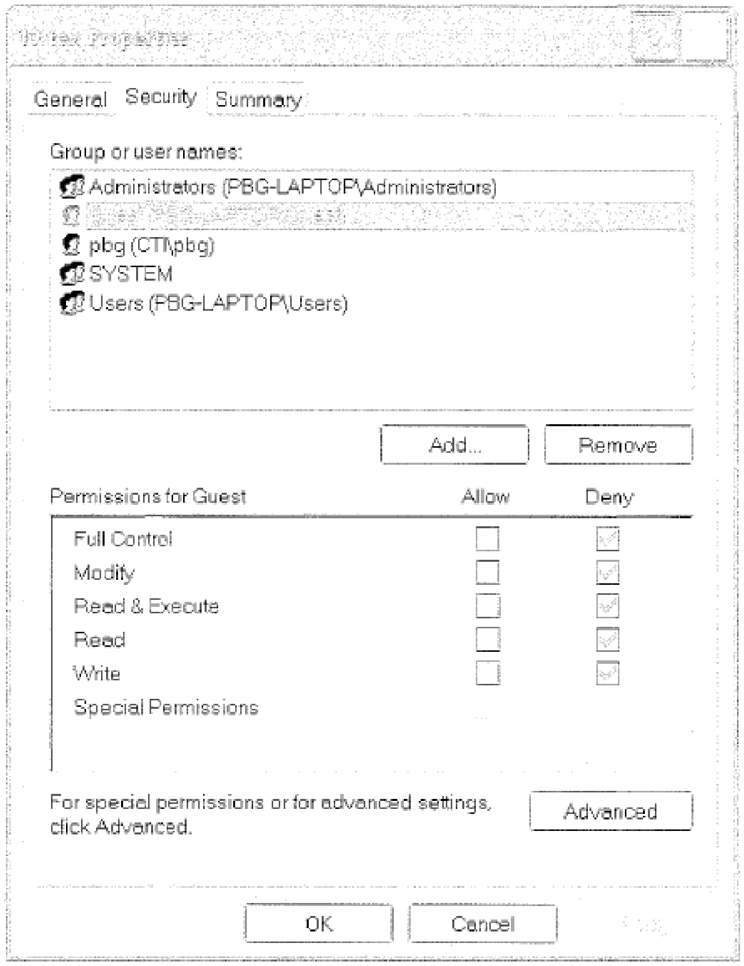
Failure Modes
Local file systems can fail for a variety of reasons, including failure of the disk containing the file system, corruption of the directory structure or other disk-management information (collectively called disk-controller failure, cable failure, and host-adapter failure. User or system-administrator failure can also cause files to be lost or entire directories or volumes to be deleted. Many of these failures will cause a host to crash and an error condition to be displayed, and human intervention will be required to repair the damage.
Remote file systems have even more failure modes. Because of the complexity of network systems and the required interactions between remote machines, many more problems can interfere with the proper operation of remote file systems. In the case of networks, the network can be interrupted between two hosts. Such interruptions can result from hardware failure, poor hardware configuration, or networking implementation issues. Although some networks have built-in resiliency, including multiple paths between hosts, many do not. Any single failure can thus interrupt the flow of DFS commands.

PROTECTION
When information is stored in a computer system, we want to keep it safe from physical damage (the issue of *reliability)* and improper access (the issue of *protection).*
Reliability is generally provided by duplicate copies of files. Many computers have systems programs that automatically (or through computer-operator intervention) copy disk files to tape at regular intervals (once per day or week or month) to maintain a copy should a file system be accidentally destroyed.
File systems can be damaged by hardware problems (such as errors in reading or writing), power surges or failures, head crashes, dirt, temperature extremes, and vandalism. Files may be deleted accidentally. Bugs in the file-system software can also cause file contents to be lost.

Types of Access
- **Read**. Read from the file.
- **Write**. Write or rewrite the file.
- **Execute**. Load the file into memory and execute it.
- **Append**. Write new information at the end of the file.
- **Delete**. Delete the file and free its space for possible reuse.
- **List**. List the name and attributes of the file.

Access Control
The most common approach to the protection problem is to make access dependent on the identity of the user. Different users may need different types of access to a file or directory. The most general scheme to implement dependent access is to associate with each file and directory an access control list (ACL) specifying user names and the types of access allowed for each user.
When a user requests access to a particular file, the operating system checks the access list associated with that file. If that user is listed for the requested access, the access is allowed. Otherwise, a protection violation occurs, and the user job is denied access to the file.
To condense the length of the access-control list, many systems recognize three classifications of users in connection with each file:
- **Owner**. The user who created the file is the owner.
- **Group**. A set of users who are sharing the file and need similar access is a group, or work group.
- **Universe**. All other users in the system constitute the universe

  .



General Security Summary

Group or user names:

- Administrators (PBG-LAPTOP\Administrators)
- [illegible]
- pbg (CTI\pbg)
- SYSTEM
- Users (PBG-LAPTOP\Users)

Add... Remove

Permissions for Guest	Allow	Deny
Full Control	☐	✓
Modify	☐	✓
Read & Execute	☐	✓
Read	☐	✓
Write	☐	✓
Special Permissions		

For special permissions or for advanced settings, click Advanced. Advanced

OK Cancel

Fig: Windows XP access-control list management

```
-rw-rw-r--    1 pbg   staff      31200   Sep 3 08:30    intro.ps
drwx------    5 pbg   staff        512   Jul 8 09.33    private/
drwxrwxr-x    2 pbg   staff        512   Jul 8 09:35    doc/
drwxrwx---    2 pbg   student      512   Aug 3 14:13    student-proj/
-rw-r--r--    1 pbg   staff       9423   Feb 24 2003    program.c
-rwxr-xr-x    1 pbg   staff      20471   Feb 24 2003    program
drwx--x--x    4 pbg   faculty      512   Jul 31 10:31   lib/
drwx------    3 pbg   staff       1024   Aug 29 06:52   mail/
drwxrwxrwx    3 pbg   staff        512   Jul 8 09:35    test/
```

Fig: A sample directory listing