NIT III
Process Synchronization

Inter process communication

Inter Process Communication (IPC) is a mechanism that involves communication of one process with another process. This usually occurs only in one system.

Communication can be of two types −

- Between related processes initiating from only one process, such as parent and child processes.
- Between unrelated processes, or two or more different processes.

Following are some important terms that we need to know before proceeding further on this topic.

- **Pipes** − Communication between two related processes. The mechanism is half duplex meaning the first process communicates with the second process. To achieve a full duplex i.e., for the second process to communicate with the first processanother pipe is required.
- **FIFO** − Communication between two unrelated processes. FIFO is a full duplex, meaning the first process can communicate with the second process and vice versa at the same time.
- **Message Queues** − Communication between two or more processes with full duplex capacity. The processes will communicate with each other by posting a message and retrieving it out of the queue.
- **Shared Memory** − Communication between two or more processes is achieved through a shared piece of memory among all processes
- **Semaphores** − Semaphores are meant for synchronizing access to multiple processes. When one process wants to access the memory (for reading or writing), it needs to be locked (or protected) and released when the access is removed. This needs to be repeated by all the processes to secure data.
- **Signals** − Signal is a mechanism to communication between multiple processes by way of signaling.

A process can be of two type:

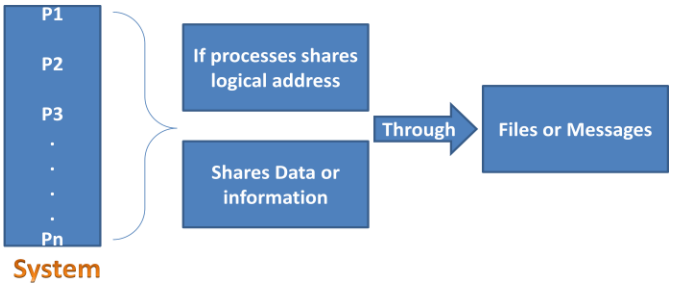
- Independent process.
- Co-operating process.

Independent process

An independent process is not affected by the execution of other processes while a co-operating process can be affected by other executing processes.

Cooperating Processes:

- A Cooperating Process is one, that can effect or to be affected by other processes executing in the system
- Cooperating Processes can either directly share a logical address space or be allowed to share data only through file or messages.
- Example



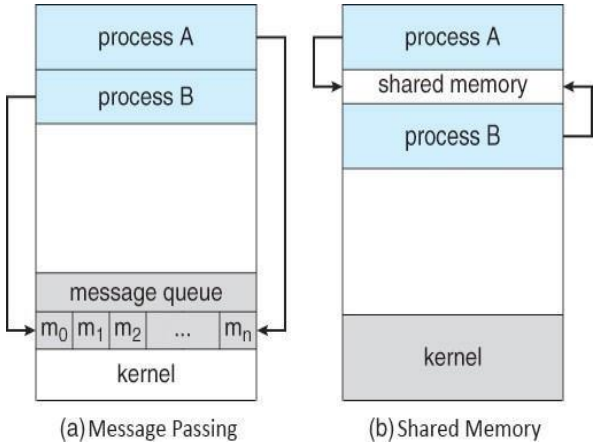
- If one process fails it will effect to all the processes.

Processes can communicate with each other usingthese two ways:

1. Shared Memory
2. Message passing

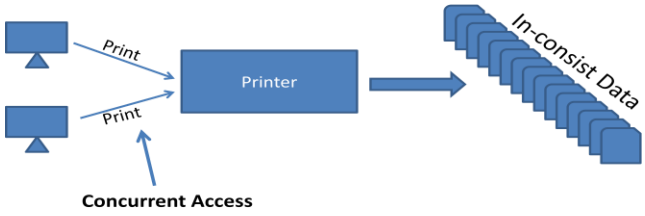
In the shared-memory model, a region of memory which is shared by cooperating processes gets established. Processes can be then able to exchange information by reading and writing all the data to the shared region.

In the message-passing form, communication takes place by way of messages exchanged among the cooperating processes.



Process Synchronization

Problem arises in concurrent access to shared data and it may lead to inconsistency.



Process Synchronization needs between processes, whether, when and how to access the particular shared resource and also for better utilization of process.

Process Synchronization means sharing system resources by processes in such a way that, Concurrent access

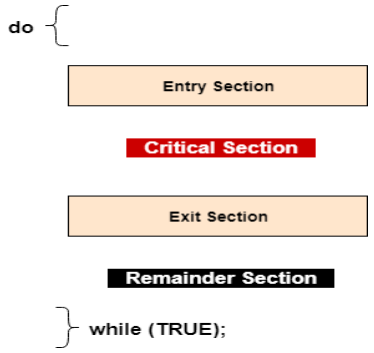
to shared data is handled thereby minimizing the chance of inconsistent data. Maintaining data consistency demands mechanisms to ensure synchronized execution of cooperating processes.

Race Condition

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

Critical Section

- The critical section is a code segment where the shared variables can be accessed.
- An atomic action is required in a critical section i.e. only one process can execute in its critical section ata time.
- All the other processes have to wait to execute in their critical sections.
- A diagram that demonstrates the critical section is as follows:



In the above diagram,

- The entry section handles the entry into the critical section. It acquires the resources needed for execution by the process.
- The exit section handles the exit from the critical section. It releases the resources and also informs the other processes that the critical section is free.

Solution to the Critical Section Problem

The critical section problem needs a solution to synchronize the different processes.

The solution to the critical section problem mustsatisfy the following conditions:

- **Mutual Exclusion**

  Mutual exclusion implies that only one process can be inside the critical section at any time. If any other processes require the critical section, they mustwait until it is free.
- **Progress**

  Progress means that if a process is not using the critical section, then it should not stop any other process from accessing it. In other words, any process can enter a critical section if it is free.
- **Bounded Waiting**

  Bounded waiting means that each process must have a limited waiting time. It should not wait endlessly to access the critical section.

Mutual Exclusion:

Requirements for Mutual Exclusion

- Only one process at a time is allowed in the criticalsection for a resource
- A process that halts in its noncritical section must doso without interfering with other processes
- A process must not be delayed access to a criticalsection when there is no other process using it
- No assumptions are made about relative processspeeds or number of processors
- A process remains inside its critical section for afinite time only

Mutual Exclusion Using Hardware Support

- **Interrupt Disabling**
  - A process runs until it invokes an operatingsystem service or until it is interrupted
  - Disabling interrupts guarantees mutual exclusion
- **Disadvantages:**
  - Processor is limited in its ability to interleave programs
  - Will not work in multiprocessor architecture
- **Compare&Swap Instruction**

```
int compareAndSwap (int *word, int testval, int
newval)
{
    int oldval; oldval =
    *word;
    if (oldval == testval) *word = newval;return
    oldval;
}
```

- **Exchange instruction**

```
void exchange (int *register, int *memory)
{
    int temp;
```

Algorithm for Process Pi

```
do
 {
   flag[i] = TRUE;
   turn = j;
   while (flag[j] && turn == j);

        critical section

        remainder section

   flag[i] = FALSE;

} while (true)
```

Peterson's Solution preserves all three conditions:

- Mutual Exclusion is assured as only one process can access the critical section at any time.
- Progress is also assured, as a process outside the critical section does not block other processes from entering the critical section.
- Bounded Waiting is preserved as every process gets a fair chance.

Disadvantages of Peterson's Solution

- It involves Busy waiting
- It is limited to 2 processes.

Semaphores:

- Considered the simplest of synchronization tools.
- 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) ; // busy wait
   S--;
}
```

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

```
temp = *memory;
*memory = *register;
*register = temp;
}
```

Peterson's Solution

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

- 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.
- In addition, in the case of wait(S), the testing of the integer value of S ($S \leq 0$), as well as its possible modification (S--), must be executed without interruption.
- 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.
- Thus, binary semaphores behave similarly to mutex locks. Binary semaphores can be used instead for providing mutual exclusion.
- Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available.
- Each process that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count). When a process releases a resource, it performs a signal() operation (incrementing the count).

Disadvantage

- **Busy waiting time :** When a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code.

Classical Problems of Synchronization

Below is some of the classical problem of process synchronization in systems where cooperating processes are present.

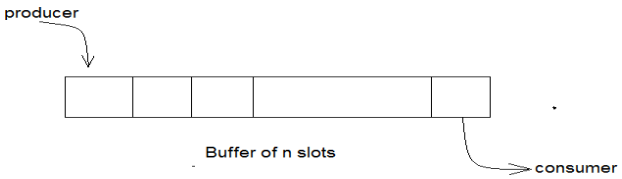
1. Bounded Buffer (Producer-Consumer) Problem
2. Dining Philosophers Problem
3. The Readers Writers Problem

1. Bounded Buffer Problem

Bounded buffer problem, which is also called **producer consumer problem**, is one of the classic problems of synchronization.

Problem Statement

- There is a buffer of n slots and each slot is capable of storing one unit of data.
- There are two processes running, namely, producer and consumer, which are operating on the buffer.



- A producer tries to insert data into an empty slot of the buffer.
- A consumer tries to remove data from a filled slot in the buffer.
- Assume that we have a shared resource counter (Number of elements in the buffer at that time).
- Initially the counter value is 3
- If producer access the counter then producer will increase the Counter value (Counter ++)
- If consumer access the counter then consumer will decrease the Counter value (Counter --)

Solution using semaphore:

- Assume that the pool consists of **n** buffers, each capable of holding one value/item.
- The **"mutex"** semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1.i,e: mutex = 1.
- The **"empty"** and **"full"** semaphores count the number of empty and full buffers.
- The semaphore **empty** is initialized to the value n; The semaphore **full** is initialized to the value 0.

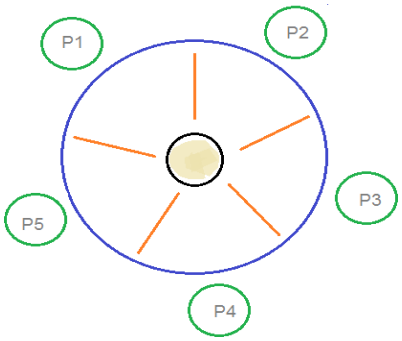
Producer	**Consumer**

do
{
Wait(empty);
Wait(mutex);
// Add item to buffer
Signal(mutex);
Signal(full);
}while(TRUE)
	do
	{
	Wait(full);
	Wait(mutex);
	// Consume item
	Signal(mutex);
	Signal(empty);
	}while(TRUE)

2. Dining Philosophers Problem

Problem Statement

Consider there are five philosophers sitting around a circular dining table. The dining table has five chopsticks and a bowl of rice in the middle as shown in the below figure.



- At any instant, a philosopher is either eating or thinking.
- When a philosopher wants to eat, he uses two chopsticks - one from their left and one from their right.
- When a philosopher wants to think, he keeps down both chopsticks at their original place.

Solution using semaphore:

- A philosopher can think for an indefinite amount of time. But when a philosopher starts eating, he has to stop at some point of time.
- The philosopher is in an endless cycle of thinking and eating.

- An array of five semaphores, stick[5], for each of the five chopsticks.
- The code for each philosopher looks like:

```
do
{
   wait(stick[i]);
   wait(stick[(i+1) % 5]);
            /* eat */
   signal(stick[i]);
   signal(stick[(i+1) % 5]);
            /* think */
} while(TRUE)
```

Consider the situation when all five philosophers are hungry simultaneously, and each of them pickup one chopstick, then a deadlock situation occurs because they will be waiting for another chopstick forever.
The possible solutions for this are:

- A philosopher must be allowed to pick up the chopsticks only if both the left and right chopsticks are available.
- Allow only four philosophers to sit at the table. That way, if all the four philosophers pick up four chopsticks, there will be one chopstick left on the table. So, one philosopher can start eating and eventually, two chopsticks will be available. In this way, deadlocks can be avoided.

3. Readers Writer Problem

Problem Statement

- There is a shared resource which should be accessed by multiple processes.
- There are two types of processes in this context. They are reader and writer.
- Any number of readers can read from the shared resource simultaneously, but only one writer can write to the shared resource.
- When a writer is writing data to the resource, no other process can access the resource.
- A writer cannot write to the resource if there are nonzero numbers of readers accessing the resource at that time.

Solution using semaphore:

- From the above problem statement, If a writer wants to write to the resource, it must wait until there are no readers currently accessing that resource.
- Here, we use one mutex = 1and a semaphore S=1.
- An integer variable read_count is used to maintain the number of readers currently accessing the resource.
- The variable read_count is initialized to 0.

Writer

```
do
{
   wait(S);
 /* perform the write
operation */
   signal(S);
} while(TRUE)
```

Reader

```
do
{
   //acquire lock
   wait(mutex);
   read_count++;
   if(read_count == 1)
      wait(S);
   //release lock
   signal(mutex);
   /* perform the reading
operation */
   // acquire lock
   wait(mutex);
   read_count--;
   if(read_count == 0)
      signal(S);
   // release lock
   signal(mutex);
} while(TRUE)
```

Code explanation

- As seen above in the code for the writer, the writer just waits on the w semaphore until it gets a chance to write to the resource.
- After performing the write operation, it increments w so that the next writer can access the resource.
- On the other hand, in the code for the reader, the lock is acquired whenever the read_count is updated by a process.
- When a reader wants to access the resource, first it increments the read_count value, then accesses the resource and then decrements the read_count value.
- The semaphore w is used by the first reader which enters the critical section and the last reader which exits the critical section.
- The reason for this is, when the first readers enters the critical section, the writer is blocked from the resource. Only new readers can access the resource now.
- Similarly, when the last reader exits the critical section, it signals the writer using the w semaphore because there are zero readers now and a writer can have the chance to access the resource.

Monitor:

- It is characterized as a set of programmer defined operators. Its representation consists of declaring of variables, whose value defines the state of an instance.
- The syntax of monitor is as follows.

```
Monitor monitor_name
{
   Shared variable declarations
   Procedure body P1 (……)
  {
    . . . . . .
  }
  Procedure body P2 (……)
  {
  . . . . . .
  }
   .
   .
   .
  Procedure body Pn (……)
  {
   . . . . . .
   }
  {
    Initialization Code
  }
}
```

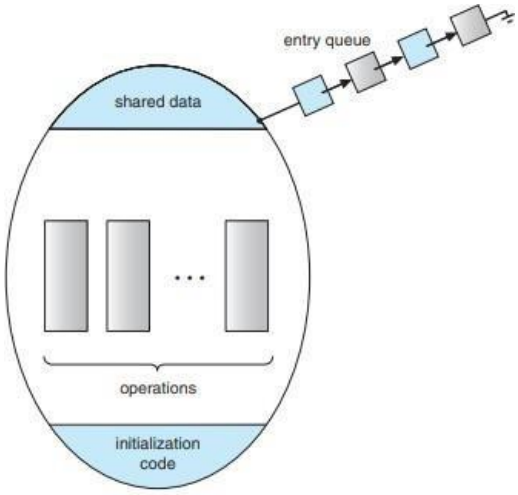


Figure 5.16 Schematic view of a monitor.

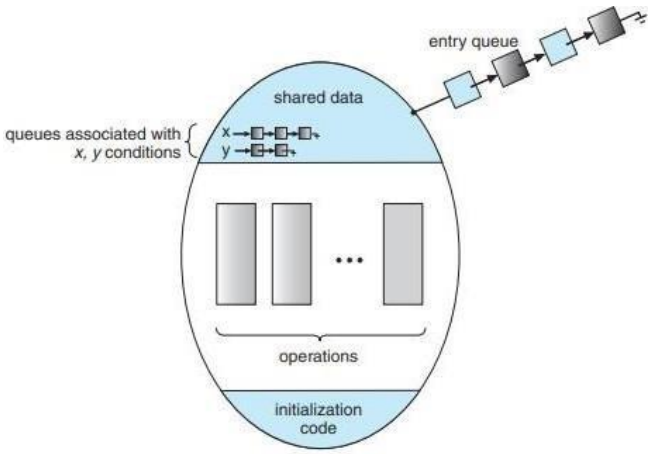


Figure 5.17 Monitor with condition variables.

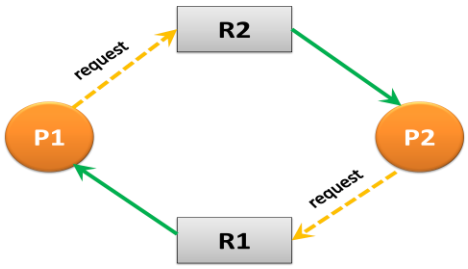
Introduction to Deadlock

Every process needs some resources to complete its execution. However, the resource is granted in a sequential order.

1. The process requests for some resource.
2. OS grant the resource if it is available otherwise let the process waits.
3. The process uses it and release on the completion.

Deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.

Example:



Here,

- Process P1 holds resource R1 and waits for resource R2 which is held by process P2.
- Process P2 holds resource R2 and waits for resource R1 which is held by process P1.
- None of the two processes can complete and release their resource.
- Thus, both the processes keep waiting infinitely.

Necessary conditions for Deadlocks

- **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.

- **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.

- **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.

- **Circular Wait**

  A set {P0, P1, ..., Pn} of waiting processes must exist such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2, ..., Pn−1 is waiting for a resource held by Pn, and Pn is waiting for a resource held by P0.

Deadlock Prevention:

Deadlock prevention is a set of methods for ensuring that at least one of these necessary conditions **cannot** hold.

Mutual Exclusion:

- ✓ The mutual exclusion condition holds for non sharable resources.
- ✓ The example is a printer cannot be simultaneously shared by several processes.
- ✓ Sharable resources do not require mutual exclusive access and thus cannot be involved in a dead lock.
- ✓ The example is read only files which are in sharing condition. If several processes attempt to open the read only file at the same time they can be guaranteed simultaneous access.

Hold and wait:

- ✓ To ensure that the hold and wait condition never occurs in the system, we must guaranty that whenever a process requests a resource it does not hold any other resources.
- ✓ There are two protocols to handle these problems such as
  - One protocol that can be used requires each process to request and be allocated all its resources before it begins execution.
  - The other protocol allows a process to request resources only when the process has no resource.
- ✓ These protocols have two main disadvantages.
  - First, resource utilization may be low, since many of the resources may be allocated but unused for a long period.
  - Second, starvation is possible.
- ✓ A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

No Preemption:

- ✓ To ensure that this condition does not hold, a protocol is used.
- ✓ If a process is holding some resources and request another resource, that cannot be immediately allocated to it.
- ✓ The preempted one added to a list of resources for which the process is waiting. The process will restart only when it can regain its old resources, as well as the new ones that it is requesting.
- ✓ Alternatively if a process requests some resources, we first check whether they are available. If they are, we allocate them. If they are not available, we check whether they are allocated to some other process that is waiting for additional resources. If so, we preempt the desired resources from the waiting process and allocate them to the requesting process.
- ✓ If the resources are not either available or held by a waiting process, the requesting process must wait.

Circular Wait:

- ✓ One way to avoid circular wait is to number all resources, and to require that processes request resources only in strictly increasing (or decreasing) order.
- ✓ In other words, in order to request resource Rj, a process must first release all Ri such that i >= j.
- ✓ One big challenge in this scheme is determining the relative ordering of the different resources
- ✓ For example: Set priorities for R_1=1, R_2=2, R_3=3 and R_4=4. With these priorities, if process P

wants to use R_1 and R_3, it should first request R_1, then R_3. Another protocol is —Whenever a process requests a resource R_j, it must have released all resources R_k with priority(R_k) >= priority(R_j).

Deadlock avoidance

- ✓ Deadlock prevention algorithm may lead to low device utilization and reduces system throughput.
- ✓ Avoiding deadlocks requires additional information about how resources are to be requested.
- ✓ With the knowledge of the complete sequences of requests and releases we can decide for each requests whether or not the process should wait.
- ✓ A deadlock avoidance algorithm dynamically examines the resources allocation state to ensure that a circular wait condition never exists.
- ✓ The resource allocation state is defined by the number of available and allocated resources and the maximum demand of each process.

Safe State:

- ✓ A state is a safe state in which there exists at least one order in which all the process will run completely without resulting in a deadlock.
- ✓ A system is in safe state if there exists a safe sequence.
- ✓ A sequence of processes <P1,P2,… Pn> is a safe sequence for the current allocation state if for each Pi the resources that Pi can request can be satisfied by the currently available resources.
- ✓ If the resources that Pi requests are not currently available then Pi can obtain all of its needed resource to complete its designated task.
- ✓ A safe state is not a deadlock state.
- ✓ Whenever a process request a resource i.e., currently available, the system must decide whether resources can be allocated immediately or whether the process must wait. The request is granted only if the allocation leaves the system in safe state.

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

Resource Allocation Graph Algorithm:

- ✓ This algorithm is used only if we have one instance of a resource type.
- ✓ In addition to the request edge and the assignment edge a new edge called claim edge is used.
- ✓ For eg:-A claim edge Pi,Rj indicates that process Pi may request Rj in future. The claim edge is represented by a dotted line.
  - When a process Pi requests the resource Rj, the claim edge is converted to a request edge. x When resource Rj is released by process Pi, the assignment edge Rj□Pi is replaced by the claim edge Pi**,**Rj.
- ✓ When a process Pi requests resource Rj the request is granted only if converting the request edge Pi Rj to as assignment edge Rj□Pi do not result in a cycle.
- ✓ Cycle detection algorithm is used to detect the cycle. If there are no cycles then the allocation of the resource to process leave the system in safe state

Banker's Algorithm:

- ✓ This algorithm is applicable to the system with multiple instances of each resource types, but this is less efficient then the resource allocation graph algorithm.
- ✓ When a new process enters the system it must declare the maximum number of resources that it may need. This number may not exceed the total number of resources in the system.
- ✓ The system must determine that whether the allocation of the resources will leave the system in a safe state or not. If it is so resources are allocated else it should wait until the process release enough resources.
- ✓ Several data structures are used to implement the banker's algorithm.
- ✓ Let _n' be the number of processes in the system and _m' be the number of resources types. We need the following data structures:
  - **Available:-** A vector of length m indicates the number of available resources. If Available[i]=k, then k instances of resource type Rj is available.

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

Safety Algorithm:

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

Step 1. Let work and finish be two vectors of length M and N respectively. Initialize work = available and Finish[i]=false for i=1,2,3,…….n

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

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

Step 4. If finish[i]=true for all i, then the system is in safe state.

This algorithm may require an order of m*n*n operation to decide whether a state is safe.

Resource Request Algorithm:

- ✓ Let Request(i) be the request vector of process Pi.
- ✓ If Request[i][j]=k,then process Pi wants K instances of the resource type Rj.
- ✓ When a request for resources is made by process Pi the following actions are taken.
  - If Request(i) <= Need(i) go to step 2 otherwise raise an error condition since the process has exceeded its maximum claim. x If Request(i) <= Available go to step 3 otherwise Pi must wait. Since the resources are not available.
  - If the system want to allocate the requested resources to process Pi then modify the state as follows.

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

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

Unit IV Memory Management

Memory Management

- Memory consists of a large array of words or bytes, each with its own address.
- 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.
- Memory unit sees only a stream of memory addresses. It does not know how they are generated.
- Program must be brought into memory and placed within a process for it to be run.
- Input queue – collection of processes on the disk that are waiting to be brought into memory for execution.
- User programs go through several steps before being run.

Base and Limit Register

- Each process has a separate memory space.
- Separate per-process memory space protects the processes from each other and is fundamental to having multiple processes loaded in memory for concurrent execution.
- To separate memory spaces, we need the ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses.
- We can provide this protection by using two registers, usually a base and a limit, as illustrated in Figure 8.1
- The **base register** holds the smallest legal physical memory address
- The **limit register** specifies the size of the range.
- **For example,** if the base register holds 300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420939 (inclusive).

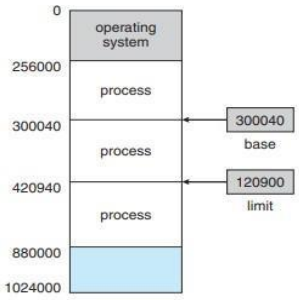
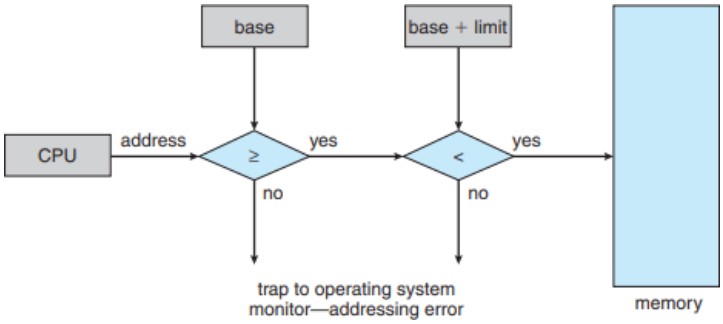


Figure 8.1 A base and a limit register define a logical address space.

- Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode with the registers.
- Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a trap to the operating system, which treats the attempt as a fatal error as shown in below figure.

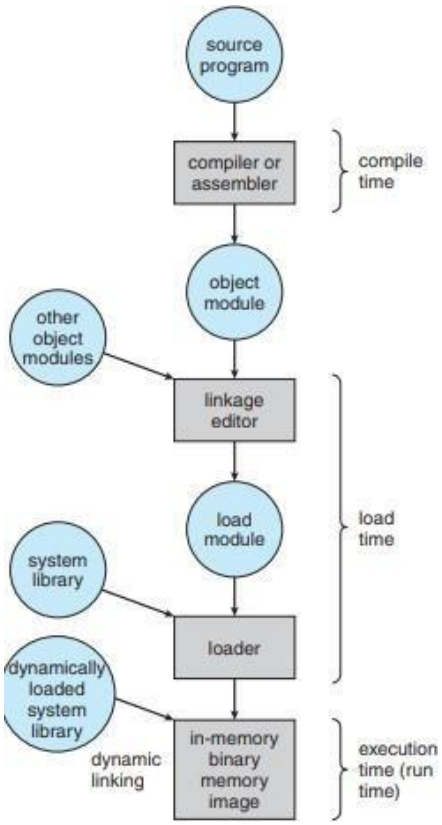


- This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users.

Address Binding

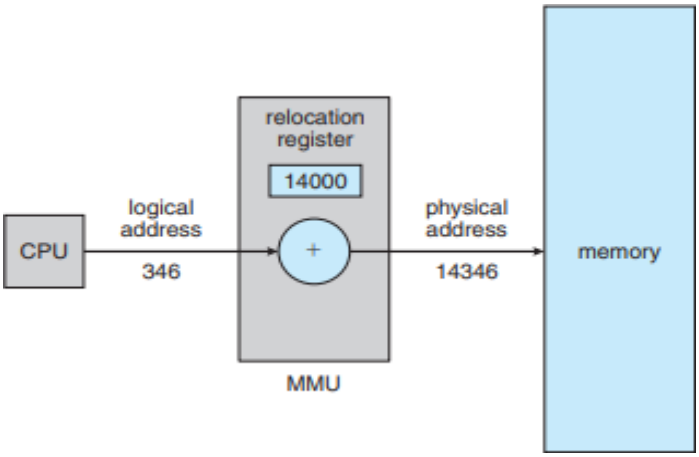
- Usually, a program resides on a disk as a binary executable file. To be executed, the program must be brought into memory and placed within a process.
- Depending on the memory management in use, the process may be moved between disk and memory during its execution.
- **The processes on the disk that are waiting to be brought into memory for execution form the input queue.**
- Addresses in the source program are generally symbolic (such as the variable count).
- A compiler typically binds these symbolic addresses to relocatable addresses.
- The linkage editor or loader in turn binds the relocatable addresses to absolute.
- Each binding is a mapping from one address space to another.
- Address binding of instructions and data to memory addresses can happen at three different stages.
  - ✓ **Compile time:** If memory location known a priori, absolute code can be generated; must recompile code if starting location changes. **Example:** .COM-format programs in MS-DOS.
  - ✓ **Load time:** Must generate relocatable code if memory location is not known at compile time.

- ✓ **Execution time:** Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., relocation registers).



Logical Versus Physical Address Space

- The concept of a logical address space that is bound to a separate physical-address space is central to proper memory management.
  - ✓ **Logical address –** address generated by the CPU; also referred to as virtual address.
  - ✓ **Physical address –** address seen by the memory unit.
- The set of all logical addresses generated by a program is a logical address space; the set of all physical addresses corresponding to these logical addresses are a physical address space.
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme.
- The run-time mapping from virtual to physical addresses is done by a hardware device called the memory management unit (MMU).
- This method requires hardware support slightly different from the hardware configuration. The base register is now called a relocation register. The value in the relocation register is added to every address generated by a user process at the time it is sent to memory.



- The user program never sees the real physical addresses. The program can create a pointer to location 346, store it in memory, manipulate it and compare it to other addresses.
- The user program deals with logical addresses. The memory mapping hardware converts logical addresses into physical addresses.
- The final location of a referenced memory address is not determined until the reference is made.

Dynamic Loading

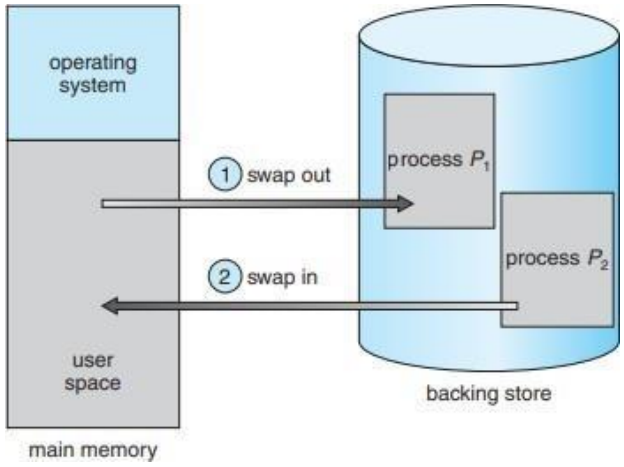
- Routine is not loaded until it is called.
- All routines are kept on disk in a relocatable load format.
- The main program is loaded into memory and is executed. When a routine needs to call another routine, the calling routine first checks to see whether the other the desired routine into memory and to update the program's address tables to reflect this change. Then control is passed to the newly loaded routine.
- **Advantages**
  - ✓ Better memory-space utilization; unused routine is never loaded.
  - ✓ Useful when large amounts of code are needed to handle infrequently occurring cases.
  - ✓ No special support from the operating system is required.
  - ✓ Implemented through program design.

Dynamic Linking

- Linking is postponed until execution time.
- Small piece of code, stub, is used to locate the appropriate memory-resident library routine, or to load the library if the routine is not already present.
- When this stub is executed, it checks to see whether the needed routine is already in memory. If not, the program loads the routine into memory.
- Stub replaces itself with the address of the routine, and executes the routine.
- Thus the next time that code segment is reached, the library routine is executed directly, incurring no cost for dynamic linking.
- Operating system is needed to check if routine is in processes' memory address.
- Dynamic linking is particularly useful for libraries.

Swapping

- A process must be in memory to be executed.
- A process can be swapped temporarily out of memory to a backing store and then brought back into memory for continued execution.
- Swapping makes it possible for the total physical address space of all processes to exceed the real physical memory of the system, thus increasing the degree of multiprogramming in a system.
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms. If a higher priority process arrives and wants service, the memory manager can swap out the lower priority process so that it can load and execute lower priority process can be swapped back in and continued. This variant is sometimes called roll out, roll in.
- **Backing store:** The backing store is commonly a fast disk. It must be large enough to accommodate copies of all memory images for all users, and it must provide direct access to these memory images.
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped.
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows).



- **For example,** assume a multiprogramming environment with a round robin CPU scheduling algorithm. When a quantum expires, the memory manager will start to swap out the process that just finished, and to swap in another process to the memory space that has been freed. In the mean time, the CPU scheduler will allocate a time slice to some other process in memory. When each process finished its quantum, it will be swapped with another process

Memory Allocation

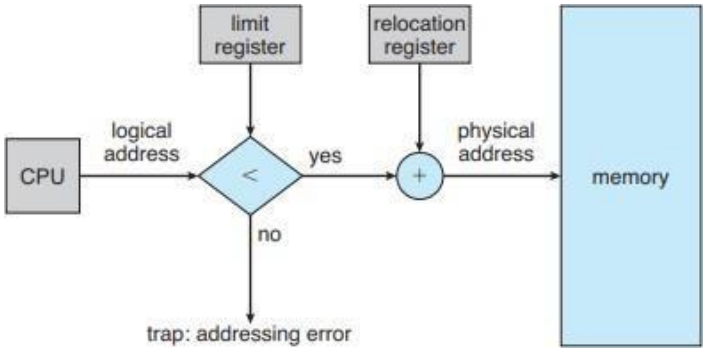
- Memory allocation is a process by which computer programs and services are assigned with physical or virtual memory space.
- Memory allocation has two core types;
  - ✓ Static Memory Allocation: The program is allocated memory at compile time.
  - ✓ Dynamic Memory Allocation: The programs are allocated with memory at run time.

Contiguous Memory Allocation

- Main memory is usually divided into two partitions:
  - ✓ Resident operating system, usually held in low memory with interrupt vector.
  - ✓ User processes, held in high memory.
- In contiguous memory allocation, each process is contained in a single contiguous section of memory

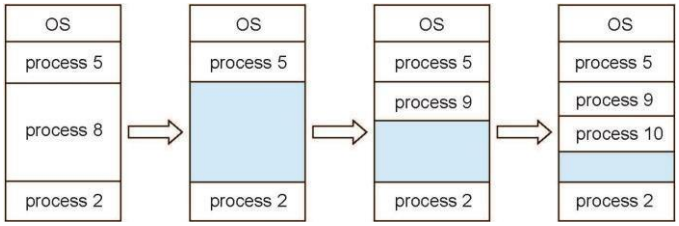
1. Single-partition allocation

- Relocation-register scheme used to protect user processes from each other, and from changing operating-system code and data.
- Relocation register contains value of smallest physical address; limit register contains range of logical addresses – each logical address must be less than the limit register.



2. Multiple-partition allocation

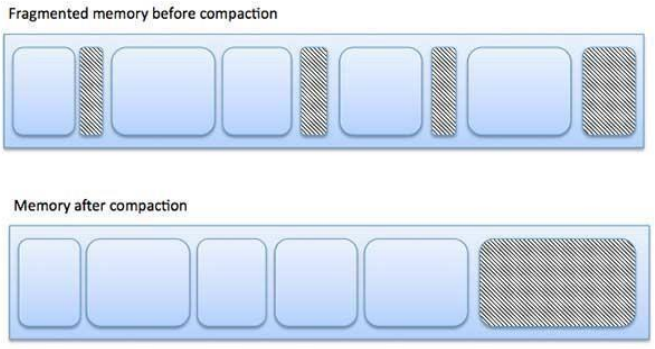
- Hole – block of available memory; holes of various size are scattered throughout memory.
- When a process arrives, it is allocated memory from a hole large enough to accommodate it.
- Operating system maintains information about:
  a) allocated partitions b) free partitions (hole)
- When a process arrives and needs memory, the system searches this set for a hole that is large enough for this process.
- If the hole is too large, it is split into two: one part is allocated to the arriving process; the other is returned to the set of holes.
- When a process terminates, it releases its block of memory, which is then placed back in the set of holes.



- This procedure is a particular instance of the general dynamic storage allocation problem, which is how to satisfy a request of size n from a list of free holes.
- There are many solutions to this problem. The set of holes is searched to determine which hole is best to allocate.
- The first-fit, best-fit and worst-fit strategies are the most common ones used to select a free hole from the set of available holes.
  - ✓ **First fit:** Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.
  - ✓ **Best fit:** Allocate the smallest hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.
  - ✓ **Worst fit:** Allocate the largest hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

Fragmentation

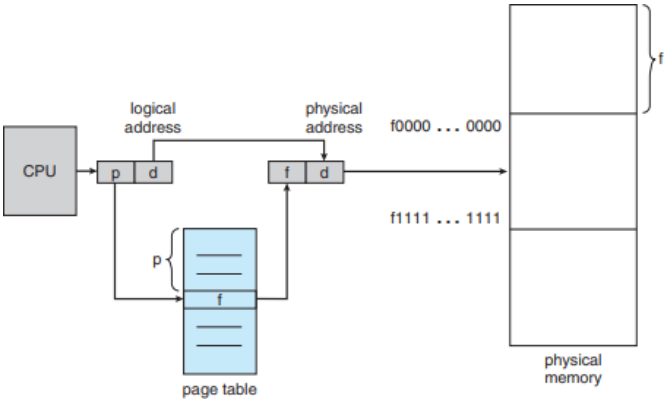
- As processes are loaded and removed from memory, the free memory space is broken into little pieces.
- It happens after sometimes that processes cannot be allocated to memory blocks considering their small size and memory blocks remains unused. This problem is known as Fragmentation.
- There are two types of Fragmentation:-
  1. External Fragmentation
  2. Internal Fragmentation
- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous.
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.
- **Compaction**
  - ✓ Reduce external fragmentation by compaction
  - ✓ Shuffle memory contents to place all free memory together in one large block.
  - ✓ Compaction is possible only if relocation is dynamic, and is done at execution time.



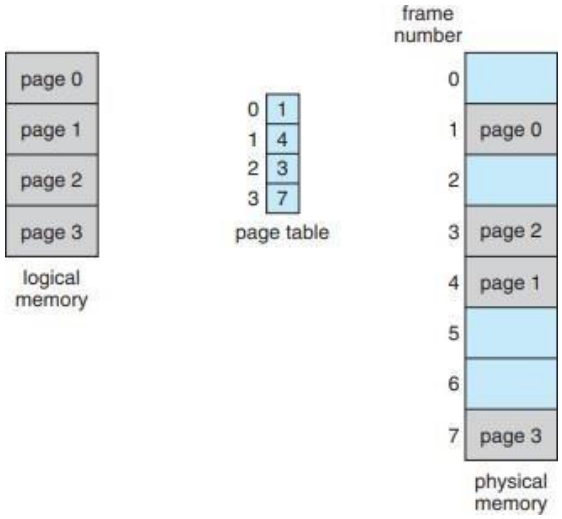
Paging

- Paging is a memory management scheme that permits the physical address space of a process to be non contiguous.
- Divides physical memory into fixed-sized blocks called **frames.**

- Divide logical memory into blocks of same size called **pages.**
- When a process is to be executed, its pages are loaded into any available memory frames from the backing store.
- The backing store is divided into fixed sized blocks that are of the same size as the memory frames.
- The hardware support for paging is illustrated in below figure.

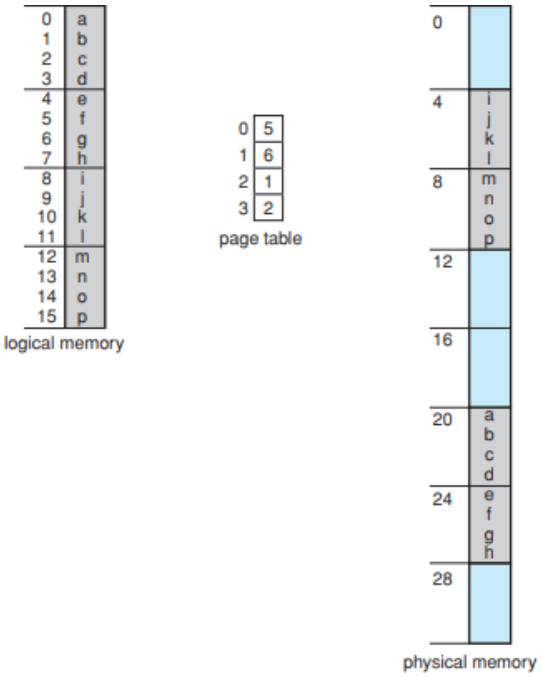


- Every address generated by the CPU is divided into two parts: a page number (p) and a page offset (d).
- The page number is used as an index into a page table. The page table contains the base address of each page in physical memory.
- This base address is combined with the page offset to define the physical memory address that is sent to the memory unit.
- The paging model of memory is shown in Figure



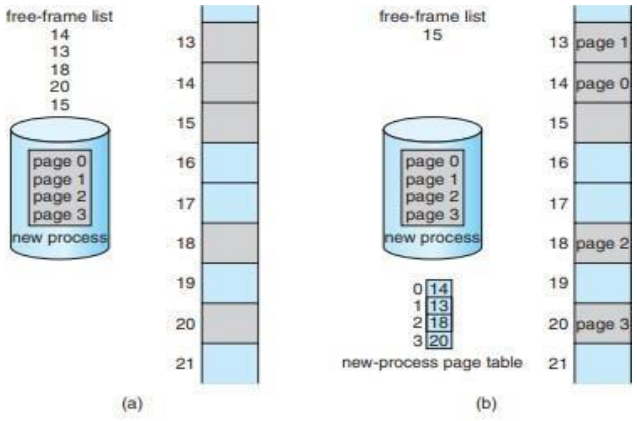
- The page size (like the frame size) is defined by the hardware.
- The size of a page is a power of 2, varying between 512 bytes and 1 GB per page, depending on the computer architecture.
- The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy.
- If the size of the logical address space is 2m, and a page size is 2n bytes, then the high-order m − n bits of a logical address designate the page number, and the n low-order bits designate the page offset.
- Thus, the logical address is as follows:

- For example, consider the memory in below Figure,
  - ✓ Here, in the logical address, n= 2 and m = 4.
  - ✓ Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages).



  - ✓ Logical address 0 is page 0, offset 0.
  - ✓ Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 [= (5 × 4) + 0].
  - ✓ Logical address 3 (page 0, offset 3) maps to physical address 23 [= (5 × 4) + 3].
  - ✓ Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 [= (6 × 4) + 0].
  - ✓ There for , Physical address = [(page size of logical memory X frame number) + offset]
- When a process arrives in the system to be executed, its size expressed in pages is examined.

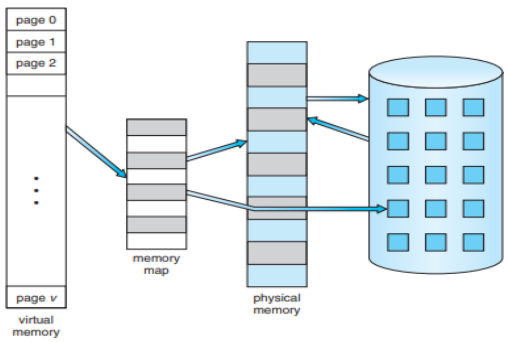
- Each page of the process needs one frame. Thus if the process requires n pages, at least n frames must be available in memory.
- If n frames are available, they are allocated to this arriving process.
- The first page of the process is loaded into one of the allocated frames, and the frame number is put in the page table for this process. The next page is loaded into another frame, and its frame number is put into the page table and so on as in below figure.



Free frames (a) before allocation and (b) after allocation.

Virtual Memory

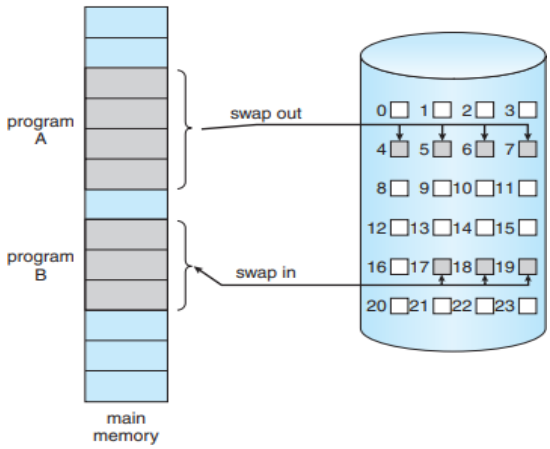
- It is a technique which allows execution of process that may not be compiled within the primary memory.
- It separates the user logical memory from the physical memory. This separation allows an extremely large memory to be provided for program when only a small physical memory is available.
- Virtual memory makes the task of programming much easier because the programmer no longer needs to working about the amount of the physical memory is available or not.
- The virtual memory allows files and memory to be shared by different processes by page sharing.
- It is most commonly implemented by demand paging.



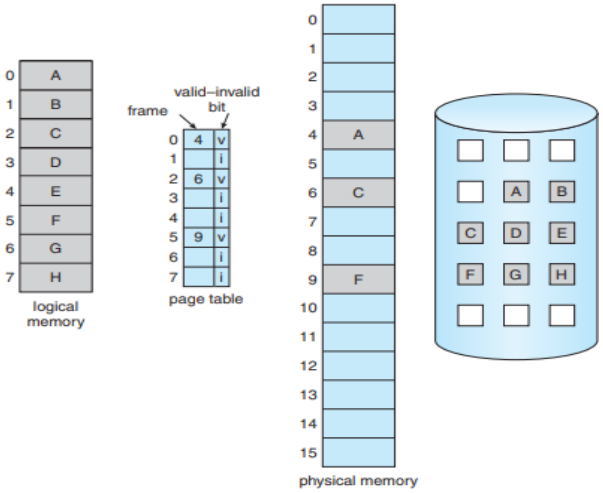
- Advantages of Virtual Memory
  - ✓ The degree of Multiprogramming will be increased.
  - ✓ User can run large application with less real RAM.
  - ✓ There is no need to buy more memory RAMs.
- Disadvantages of Virtual Memory
  - ✓ The system becomes slower since swapping takes time.
  - ✓ It takes more time in switching between applications.
  - ✓ The user will have the lesser hard disk space for its use.

1. Demand Paging

- The basic idea behind **demand paging** is that when a process is swapped in, its pages are not swapped in all at once. Rather they are swapped in only when the process needs them. (on demand.) This is termed a **lazy swapper**, although a **pager** is a more accurate term.
- A demand paging system is similar to the paging system with swapping feature.
- When we want to execute a process we swap it into the memory. A swapper manipulates entire process where as a pager is concerned with the individual pages of a process.
- The demand paging concept is using pager rather than swapper.
- When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again.
- Instead of swapping in a whole process, the pager brings only those necessary pages into memory.
- The transfer of a paged memory to contiguous disk space is shown in below figure.



- Thus it avoids reading into memory pages that will not used any way decreasing the swap time and the amount of physical memory needed.
- In this technique we need some hardware support to distinct between the pages that are in memory and those that are on the disk.
- A valid and invalid bit is used for this purpose.
- When this bit is set to valid it indicates that the associate page is in memory. If the bit is set to invalid it indicates that the page is either not valid or is valid but currently not in the disk.

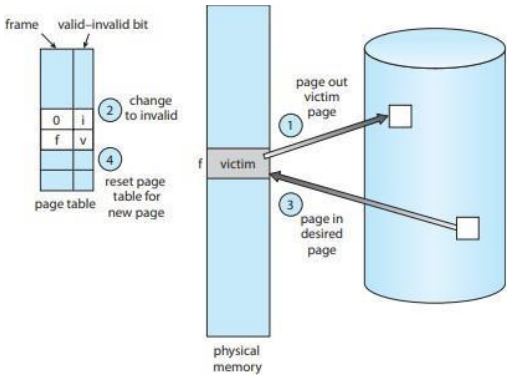


- Marking a page invalid will have no effect if the process never attempts to access that page. So while a process executes and access pages that are memory resident, execution proceeds normally.
- Access to a page marked invalid causes a page fault trap. It is the result of the OS's failure to bring the desired page into memory.
- **Procedure to handle page fault**

  If a process refers to a page that is not in physical memory then
  - ✓ We check an internal table (page table) for this process to determine whether the reference was valid or invalid.
  - ✓ If the reference was invalid, we terminate the process, if it was valid but not yet brought in, we have to bring that from main memory.
  - ✓ Now we find a free frame in memory.
  - ✓ Then we read the desired page into the newly allocated frame.
  - ✓ When the disk read is complete, we modify the internal table to indicate that the page is now in memory.
  - ✓ We restart the instruction that was interrupted by the illegal address trap. Now the process can access the page as if it had always been in memory.

2. Page Replacement

- Each process is allocated frames (memory) which hold the process's pages (data)
- Frames are filled with pages as needed – this is called demand paging
- Over-allocation of memory is prevented by modifying the page-fault service routine to replace pages
- The job of the page replacement algorithm is to decide which page gets victimized to make room for a new page
- Page replacement completes separation of logical and physical memory.



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 (Figure 9.10).

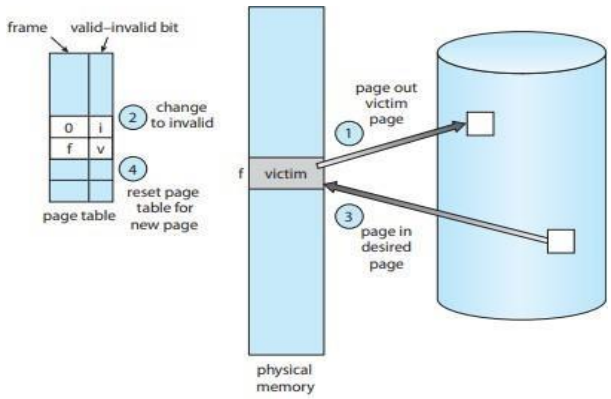


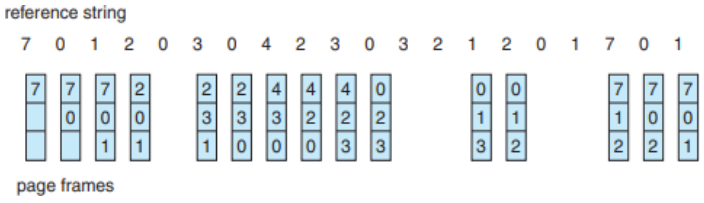
Figure 9.10 Page replacement.

- 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:
  - ✓ Find the location of the desired page on the disk.
  - ✓ Find a free frame:
    - If there is a free frame, use it.
    - If there is no free frame, use a page-replacement algorithm to select a **victim frame.**
    - Write the victim frame to the disk; change the page and frame tables accordingly.
  - ✓ Read the desired page into the newly freed frame; change the page and frame tables.
  - ✓ Continue the user process from where the page fault occurred.

Page Replacement Algorithms

I. First In First Out (FIFO) Algorithm

- Replaces pages based on their order of arrival: oldest page is replaced
- The simplest page-replacement algorithm is a first-in, first-out (FIFO) algorithm.
- When a page must be replaced, the oldest page is chosen.
- We replace the page at the head of the queue.
- When a page is brought into memory, we insert it at the tail of the queue.
- **For example:**
  Reference string : 7,0,1,2,0,3,0,4,2,3,0,7,1.
  Number of Frames : 3
  Solution:
  Three frames are initially empty.



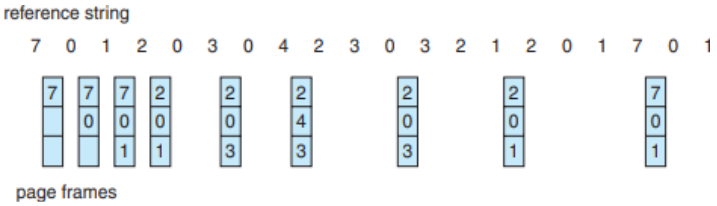
Number of page faults
= (3+1+1+1+1+1+1+1+1+1+1+1) = 15 page faults

Explanation:

- ✓ 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.
- ✓ This process continues as shown in above Figure.
- ✓ Every time a fault occurs, we show which pages are in our three frames.
- ✓ There are fifteen faults altogether.

II. Optimal Page Replacement

- Replace the page that will not be used for the longest period of time.
- Use of this page-replacement algorithm guarantees the lowest possible page fault rate for a fixed number of frames.
- **For example:**
  Reference string : 7,0,1,2,0,3,0,4,2,3,0,7,1.
  Number of Frames : 3
  Solution:
  Three frames are initially empty.



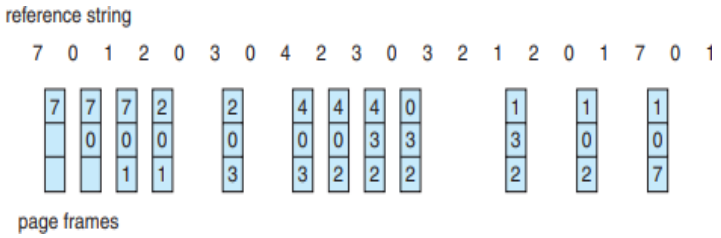
Number of page faults = (3+1+1+1+1+1+1) = 9.

Explanation:

- ✓ 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.
- ✓ In fact, no replacement algorithm can process this reference string in three frames with fewer than nine faults

III. LRU Page Replacement

- 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.
- Replaces pages based on their most recent reference – replace the page with the greatest backward distance in the reference string

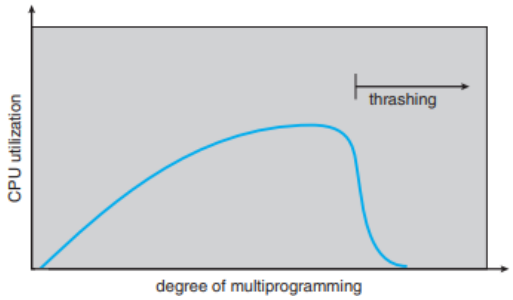


Explanation:

- The LRU algorithm produces twelve faults. 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.

Thrashing

- If a process does not have –enough‖ pages, the page-fault rate is very high
  - ✓ low CPU utilization
  - ✓ OS thinks it needs increased multiprogramming
  - ✓ adds another process to system
- Thrashing is when a process is busy swapping pages in and out
- Thrashing results in severe performance problems. Consider the following scenario, which is based on the actual behavior of early paging systems.



- The operating system monitors CPU utilization. If CPU utilization is too low, we increase the degree of multiprogramming by introducing a new process to the system.
- A global page replacement algorithm is used; it replaces pages with no regard to the process to which they belong.
- Now suppose that a process enters a new phase in its execution and needs more frames.

UNIT V
Input Output (I/O) Management

DISK STRUCTURE

- Disks provide the bulk of secondary storage for modern computer systems.
- Magnetic tape was used as an early secondary-storage medium, but the access time is much slower than for disks.
- Modern disk drives are addressed as large one-dimensional arrays of logical blocks, where the logical block is the smallest unit of transfer.
- The size of a logical block is usually 512 bytes, although some disks can be low-level formatted to choose a different logical block size, such as 1,024 bytes.
- The one-dimensional array of logical blocks is mapped onto the sectors of the disk sequentially. Sector 0 is the first sector of the first track on the outermost cylinder.
- The mapping proceeds in order through that track, then through the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost.

DISK SCHEDULING

- One of the responsibilities of the OS is to use the hardware efficiently. This will help in achieving fast access time and disk bandwidth.
- The access time has two major components:
  - ✓ The seek time is the time for the disk arm to move the heads to the cylinder containing the desired sector.
  - ✓ The rotational latency is the additional time waiting for the disk to rotate the desired sector to the disk head.
- The disk bandwidth is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer.
- We can improve both the access time and the bandwidth by scheduling the servicing of disk I/O requests in a good order.

- Whenever a process needs I/O to or from the disk, it issues a system call to the operating system. The request specifies several pieces of information:
  - ✓ Whether this operation is input or output
  - ✓ What the disk address for the transfer is
  - ✓ What the memory address for the transfer is
  - ✓ What the number of bytes to be transferred is
- If the desired disk drive and controller are available, the request can be serviced immediately. If the drive or controller is busy, any new requests for service will be placed on the queue of pending requests for that drive.
- For a multiprogramming system with many processes, the disk queue may often have several pending requests. Thus, when one request is completed, the operating system chooses which pending request to service next.

Disk Scheduling Algorithms

I. FCFS Scheduling

- The simplest form of disk scheduling is, of course, the first-come, first-served (FCFS) algorithm.
- This algorithm is intrinsically fair, but it generally does not provide the fastest service.

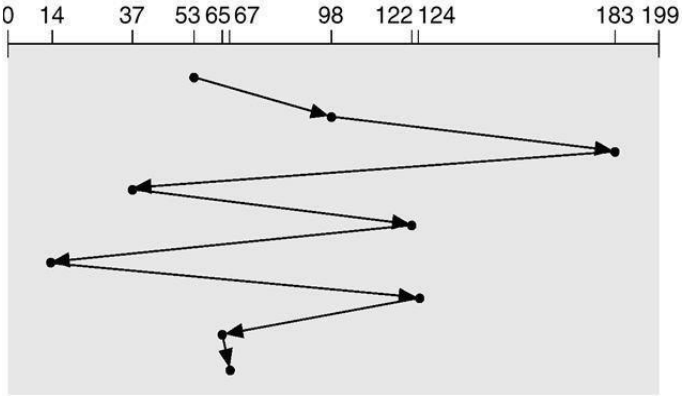
Example:

Consider a disk queue with requests for I/O to blocks on cylinders:98, 183, 37, 122, 14, 124, 65, 67 in that order. There are 200 cylinders numbered from 0 to 199. The disk head is initially at the cylinder 53. Compute total head movements.

Solution:

Starting with cylinder 53, it will first move from 53 to 98, then to 183, 37, 122, 14,124, 65, and finally to 67 as shown in Figure.

Head movement from 53 to 98 = 45
Head movement from 98 to 183 = 85
Head movement from 183 to 37 = 146
Head movement from 37 to 122 =85
Head movement from 122 to 14 =108
Head movement from 14 to 124 =110
Head movement from 124 to 65 =59
Head movement from 65 to 67 = 2
Total head movement = 640



II. SSTF Scheduling

- The SSTF (shortest-seek-time-first) algorithm selects the request with the minimum seek time from the current head position.
- Since seek time increases with the number of cylinders traversed by the head, SSTF chooses the pending request closest to the current head position.

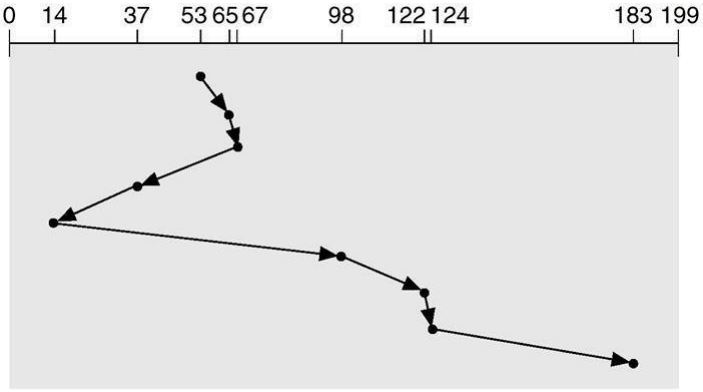
Example:

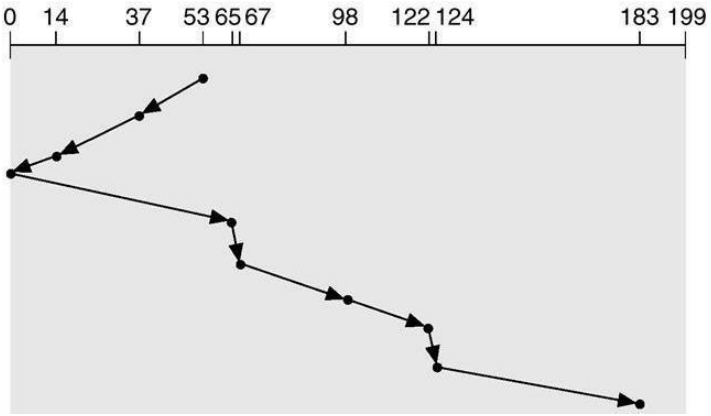
Consider a disk queue with requests for I/O to blocks on cylinders: 98, 183, 37, 122, 14, 124, 65, 67 in that order. There are 200 cylinders numbered from 0 to 199. The disk head is initially at the cylinder 53. Compute total head movements.

Solution:

The closest request to the initial head position 53 is at cylinder 65. Once we are at cylinder 65, the next closest request is at cylinder 67. From there, the request at cylinder 37 is closer than 98, so 37 is served next. Continuing, we service the request at cylinder 14, then 98, 122, 124, and finally 183. It is shown in Figure.

Head movement from 53 to 65 = 12
Head movement from 65 to 67 = 2
Head movement from 67 to 37 = 30
Head movement from 37 to 14 =23
Head movement from 14 to 98 =84
Head movement from 98 to 122 =24
Head movement from 122 to 124 =2
Head movement from 124 to 183 = 59
Total head movement = 236





III. SCAN Scheduling

- In the SCAN algorithm, the disk arm starts at one end of the disk, and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk.
- At the other end, the direction of head movement is reversed, and servicing continues. The head continuously scans back and forth across the disk.

Example:

Consider a disk queue with requests for I/O to blocks on cylinders: 98, 183, 37, 122, 14, 124, 65, 67 in that order. There are 200 cylinders numbered from 0 to 199. The disk head is initially at the cylinder 53.
Compute total head movements.
Solution:

- Before applying SCAN algorithm, we need to know the current direction of head movement. Assume that disk arm is moving toward 0, the head will service 37 and then 14.
- At cylinder 0, the arm will reverse and will move toward the other end of the disk, servicing the requests at 65,67,98, 122, 124, and 183 It is shown in Figure.

Head movement from 53 to 37 = 16
Head movement from 37 to 14 = 23
Head movement from 14 to 0 = 14
Head movement from 0 to 65 =65
Head movement from 65 to 67 =2
Head movement from 67 to 98 =31
Head movement from 98 to 122 =24
Head movement from 122 to 124 = 2
Head movement from 124 to 183 = 59
Total head movement = 236

IV. C- SCAN Scheduling

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

Example:

Consider a disk queue with requests for I/O to blocks on cylinders: 98, 183, 37, 122, 14, 124, 65, 67 in that order. There are 200 cylinders numbered from 0 to 199. The disk head is initially at the cylinder 53.
Compute total head movements.

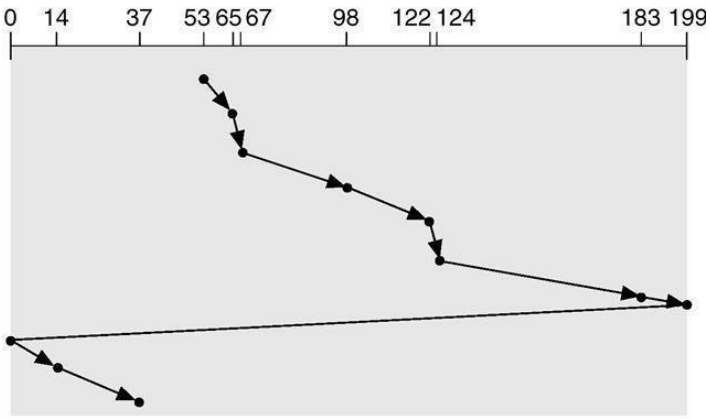
Solution:

Before applying C - SCAN algorithm, we need to know the current direction of head movement.

Assume that disk arm is moving toward 199, the head will service 65, 67, 98, 122, 124, 183.Then it will move to 199 and the arm will reverse and move towards 0. While moving towards 0, it will not serve. But, after reaching 0, it will reverse again and then serve 14 and 37. It is shown in Figure

Head movement from 53 to 65 = 12
Head movement from 65 to 67 = 2
Head movement from 67 to 98 = 31
Head movement from 98 to 122 =24
Head movement from 122 to 124 =2
Head movement from 124 to 183 =59
Head movement from 183 to 199 =16
Head movement from 199 to 0 = 199

Head movement from 0 to 14 = 14
Head movement from 14 to 37 = 23
Total head movement = 382



V. Look Scheduling

- As we described them, both SCAN and C-SCAN move the disk arm across the full width of the disk.
- In practice, neither algorithm is implemented this way. More commonly, the arm goes only as far as the final request in each direction then, it reverses direction immediately, without going all the way to the end of the disk.
- These versions of SCAN and C-SCAN are called LOOK and C-LOOK scheduling, because they look for a request before continuing to move in a given direction.

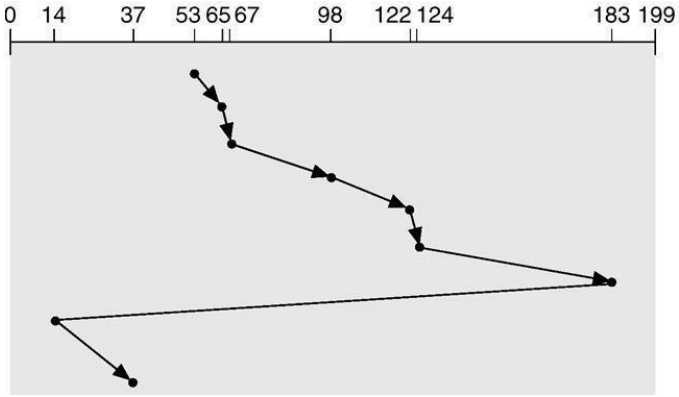
Example:

Consider a disk queue with requests for I/O to blocks on cylinders: 98, 183, 37, 122, 14, 124, 65, 67 in that order. There are 200 cylinders numbered from 0 to 199. The disk head is initially at the cylinder 53. Compute total head movements.

Solution:

Assume that disk arm is moving toward 199, the head will service 65, 67, 98,122, 124, 183. Then the arm will reverse and move towards 14. Then it will serve 37. It is shown in Figure

Head movement from 53 to 65 = 12
Head movement from 65 to 67 = 2
Head movement from 67 to 98 = 31
Head movement from 98 to 122 =24
Head movement from 122 to 124 =2
Head movement from 124 to 183 =59
Head movement from 183 to 14 = 169
Head movement from 14 to 37 = 23
Total head movement = 322



DISK MANAGEMENT

The OS is responsible for several other aspects of disk management as well.

Disk Formatting

- A new magnetic disk is a blank slate: It is just platters of a magnetic recording material.
- Before a disk can store data, it must be divided into sectors that the disk controller can read and write. This process is called low-level formatting (or physical formatting).
- Low-level formatting fills the disk with a special data structure for each sector.
- To use a disk to hold files, the OS needs to record its own data structures on the disk.
- It does so in two steps.
  - ✓ The first step is to partition the disk into one or more groups of cylinders. The OS treats each partition as though it were a separate disk.
  - ✓ After partitioning, the second step is logical formatting (or creation of a file system). In this step, the OS stores the initial file-system data structures onto the disk. These data structures may include maps of free and allocated space and an initial empty directory.

Boot Blocks

- For a computer to start running, it needs to have an initial program to run. This bootstrap program initializes all aspects of the system, from CPU registers to device controllers and the contents of main memory, and then starts the OS.

- To do its job, the bootstrap program finds the OS kernel on disk, loads that kernel into memory, and jumps to an initial address to begin the operating-system execution.

Bad Blocks

- Because disks have moving parts and small tolerances, they are prone to failure.
- Sometimes the failure is complete, and the disk needs to be replaced, and its contents restored from backup media to the new disk.
- More frequently, one or more sectors become defective. Most disks even come from the factory with bad blocks.
- Depending on the disk and controller in use, these blocks are handled in a variety of ways:
  - ✓ On simple disks with IDE, bad blocks are handled manually (like formatting the disk)
  - ✓ The controller maintains a list of bad blocks on the disk. The list is initialized during the low-level format at the factory, and is updated over the life of the disk.

SWAP SPACE MANAGEMENT

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

File concept:

- A file is a collection of related information that is stored on secondary storage.
- Information stored in files must be persistent i.e. not affected by power failures & system reboots. Files may be of free from such as text files or may be formatted rigidly.
- Files represent both programs as well as data.
- Part of the OS dealing with the files is known as file system.
- The important file concepts include:

1. File attributes:

A file has certain attributes which vary from one operating system to another.

- ✓ **Name:** Every file has a name by which it is referred.
- ✓ **Identifier:** It is unique number that identifies the file within the file system.
- ✓ **Type:** This information is needed for those systems that support different types of files.
- ✓ **Location:** It is a pointer to a device & to the location of the file on that device
- ✓ **Size:** It is the current size of a file in bytes, words or blocks.
- ✓ **Protection:** It is the access control information that determines who can read, write & execute a file.
- ✓ **Time, date & user identification:** It gives information about time of creation or last modification & last use.

2. File operations:

The operating system can provide system calls to create, read, write, reposition, delete and truncate files.

- ✓ **Creating files:** Two steps are necessary to create a file. First, space must be found for the file in the file system. Secondly, an entry must be made in the directory for the new file.
- ✓ **Reading a file:** Data & read from the file at the current position. The system must keep a read pointer to know the location in the file from where the next read is to take place. Once the read has been taken place, the read pointer is updated.
- ✓ **Writing a file:** Data are written to the file at the current position. The system must keep a write pointer to know the location in the file where the next write is to take place. The write pointer must be updated whenever a write occurs.
- ✓ **Repositioning within a file (seek):** The directory is searched for the appropriate entry & the current file position is set to a given value. After repositioning data can be read from or written into that position.
- ✓ **Deleting a file:** To delete a file, we search the directory for the required file. After deletion, the space is released so that it can be reused by other files.

- ✓ **Truncating a file**: The user may erase the contents of a file but allows all attributes to remain unchanged expect the file length which is rest to ‗O' & the space is released

3. File types:

- The file name is spilt into 2 parts, Name & extension. Usually these two parts are separated by a period. The user & the OS can know the type of the file from the extension itself.
- Listed below are some file types along with their extension:

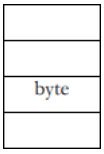
	File Type	Extension
✓	Executable File	exe, bin, com
✓	Object File	obj, o (compiled)
✓	Source Code file	C, C++, Java, pas
✓	Batch File	bat, sh
✓	Text File	txt, doc
✓	Archive File	arc, zip, tar
✓	Multimedia File	mpeg

4. File structure:

Files can be structured in several ways. Three common possible are:

a) Byte sequence:

- The figure shows an unstructured sequence of bytes. The OS doesn't care about the content of file. It only sees the bytes.
- This structure provides maximum flexibility.
- Users can write anything into their files & name them according to their convenience.
- Both UNIX & windows use this approach.



b) Record sequence:

- In this structure, a file is a sequence of fixed length records.
- Here the read operation returns one records & the write operation overwrites or append or record.

c) Tree:

- In this organization, a file consists of a tree of records of varying lengths.
- Each record consists of a key field.
- The tree is stored on the key field to allow first searching for a particular key.

5. Access methods:

Basically, access method is divided into 2 types:

Sequential access:

- It is the simplest access method. Information in the file is processed in order i.e. one record after another.
- A process can read all the data in a file in order starting from beginning but can't skip & read arbitrarily from any location.
- Sequential files can be rewound.
- It is convenient when storage medium was magnetic tape rather than disk.

Direct access:

- A file is made up of fixed length-logical records that allow programs to read & write records rapidly in no particular O order.
- This method can be used when disk are used for storing files.
- This method is used in many applications e.g. database systems.
- If an airline customer wants to reserve a seat on a particular flight, the reservation program must be able to access the record for that flight directly without reading the records before it.
- In a direct access file, there is no restriction in the order of reading or writing.
- **For example,** we can read block 14, then read block 50 & then write block 7 etc. Direct access files are very useful for immediate access to large amount of information.

Directory structure:

- The file system of computers can be extensive. Some systems store thousands of file on disk. To manage all these data, we need to organize them.

The organization is done in 2 steps. The file system is broken into partitions. Each partition contains information about file within it.

Operation on a directory:

- ✓ **Search for a file**: We need to be able to search a directory for a particular file.
- ✓ **Create a file:** New files are created & added to the directory.
- ✓ **Delete a file:** When a file is no longer needed, we may remove it from the directory.
- ✓ **List a directory:** We should be able to list the files of the directory.
- ✓ **Rename a file:** The name of a file is changed when the contents of the file changes.
- ✓ **Traverse the file system:** It is useful to be able to access every directory & every file within a directory.

Structure of a directory:

The most common schemes for defining the structure of the directory are:

1. Single level directory:

- It is the simplest directory structure.
- All files are present in the same directory.
- So it is easy to manage & understand. Limitation: A single level directory is difficult to manage when the no. of files increases or when there is more than one user.
- Since all files are in same directory, they must have unique names. So, there is confusion of file names between different users.

2. Two level directories:

- The solution to the name collision problem in single level directory is to create a separate directory for each user.
- In a two level directory structure, each user has its own user file directory. When a user logs in, then master file directory is searched. It is indexed by user name & each entry points to the UFD of that user.
- **Limitation:** It solves name collision problem. But it isolates one user from another.
- It is an advantage when users are completely independent. But it is a disadvantage when the users need to access each other‘s files & co-operate among themselves on a particular task.

3. Tree structured directories:

- It is the most common directory structure.
- A two level directory is a two level tree. So, the generalization is to extend the directory structure to a tree of arbitrary height.
- It allows users to create their own subdirectories & organize their files.
- Every file in the system has a unique path name.
- It is the path from the root through all the sub-directories to a specified file.
- A directory is simply another file but it is treated in a special way. One bit in each directory entry defines the entry as a file (O) or as sub-directories.