CHAPTER 7

OPERATING SYSTEM

Syllabus: Operating system: Processes, Threads, Inter-process communication, Concurrency, Synchronization, Deadlock, CPU scheduling, Memory management and virtual memory, File systems, I/O systems, Protection and security

7.1 INTRODUCTION

An operating sysem (OS) can be defined as an intermediate program between the user and the computer hardware. On the user end, it handles application programs and at the other end it makes use of system call(s) to instruct the hardware to perform certain task as instructed by the user. There are different types of operating systems to accomplish various tasks. Mainframe operating systems are designed to optimize hardware utilization. The real-time operating systems are designed for timeliness, the time-sharing systems are designed for efficient usage of the system and other resources. Some of the examples of OS are: UNIX, Linux (different flavors, such as, Redhat, Suse, Fedora, Debian etc.), MacOS, Microsoft Windows, etc. Nowadays, the mobiles that we use are also equipped with operating system. Some of the well-known OS for mobiles are — Symbian and Android.

The function of the operating systems is to provide an environment in which a user can execute his/her commands in a **convenient** and **efficient** manner. Operating system is also called **resource allocator**. The placement of operating systems in a computing environment is shown in Fig. 7.1.

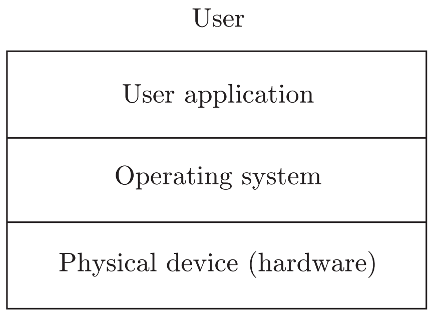


Figure 7.1 Location of an operating system in a computing environment.

System Call is a request made by a user program to get the service of an operating system.

To consider the “inside view” of the system, the OS is written as a collection of procedures, each of which can call any other, whenever required. Each parameter has a well-defined interface with input parameters and output results. A simplified layered structure of Dijkstra’s “THE” operating system is shown in Fig. 7.2:

LAYER	FUNCTION
5	The operator (user)
4	User programs
3	Input/output management
2	Operator-process communication
1	Memory management
0	Process allocation and multiprogramming

Figure 7.2 Simplified structure of Dijkstra’s THE operating system.

7.2 TYPES OF OPERATING SYSTEM

7.2.1 Batch Operating System

In a batch operating system, tasks are submitted to a pool, and only after the completion of one job, the next job is executed (Fig. 7.3). In this system, the CPU idle time (free time) is high and throughput of the system is low. Throughput is the number of jobs executed per unit time.

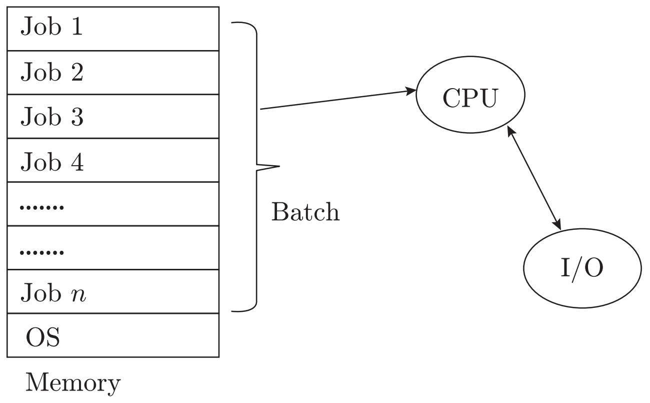


Figure 7.3 A batch operating system.

7.2.2 Multiprogramming Operating System

A single user cannot keep either the CPU or the I/O devices busy at all times. Multiprogramming increases CPU utilization by organizing jobs so that the CPU is not idle at any time (Fig. 7.4). Jobs to be executed are maintained in the memory simultaneously and the OS switches among these jobs for their execution. When one job waits for some input (WAIT state), the CPU switches to another job. This process is followed for all jobs in memory. When the wait for the first job is over, the CPU is back to serve it. The CPU will be busy till all the jobs have been executed. Thus, increasing the CPU utilization and throughput.

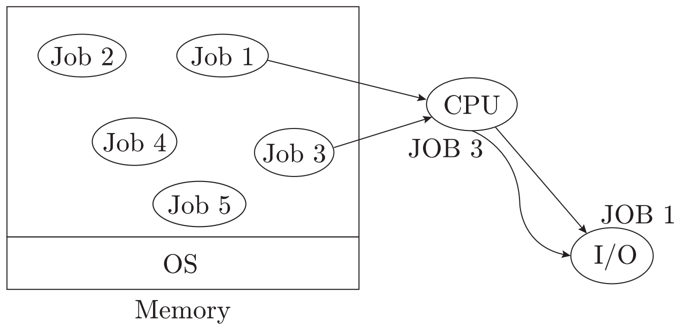


Figure 7.4 A multiprogramming operating system.

7.2.3 Multitasking Operating System

Multitasking system is the extension of multiprogramming system. In this system, jobs will be executed in the time-sharing mode. It uses CPU scheduling and multiprogramming techniques to provide a small portion of a timeslot to each user. Each user has at least one separate program in memory.

7.2.4 Multiprocessor Operating System

Multiprocessor operating system consists of more than one processor for execution (Fig. 7.5). It improves the throughput of the system and it is also more reliable than the single processor system because if one processor breaks down then the other can share the load of that processor (Table 7.1).

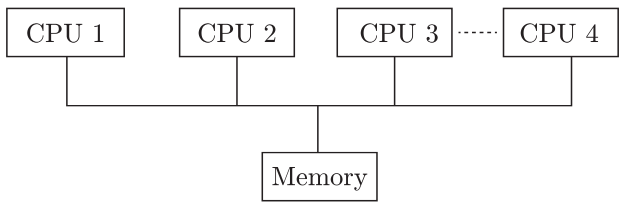


Figure 7.5 A multiprocessor operating system.

Table 7.1 Comparison between Batch Processing and Timesharing OS

	Batch Processing	Timesharing
AIM	Maximize processor utilization	Minimize response time
COMMANDS	Job Control Language (JCL) provided along with the job	As per the commands provided at the terminal

7.2.5 Real-Time Operating System

A real-time operating system (RTOS) is the one which deals with real-time application requests (Fig. 7.6). The input data must be processed without any delays. The processing time is shorter and the waiting time is minimum. If an RTOS meets a deadline approximately, it is known as a soft RTOS, but if deadline has to be met deterministically (hard deadline), it is a hard RTOS.

The events that an RTOS may have to respond to can be either periodic (occurring at regular intervals) or aperiodic (occurring at irregular intervals). Further, if there are n periodic events and an event i occurs with period P_i and requires T_i seconds of CPU time, then the load can be handled if

$$\sum_{i=1}^{n}(T_i / P_i) \leq 1$$

An RTOS that meets this criteria is said to be *schedulable.*

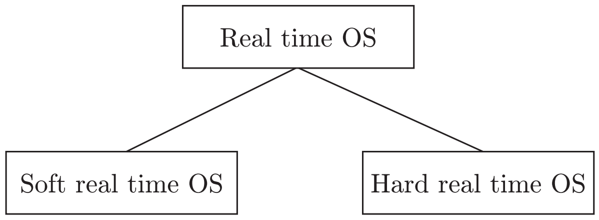


Figure 7.6 Classification of a real-time operating system.

An operating system is concerned with the management functions of the following resources: (a) process management, (b) memory management, (c) file management and (d) device management.

7.3 PROCESS MANAGEMENT

7.3.1 Process

A program under execution is called a process. In other words, a program is in the main memory and the occupied processor is called a process. A process has various attributes, states and operation.

7.3.1.1 Attributes of a Process

A process is distinct from another based upon the following attributes:

1. **Process id:** It is a unique identification number which is assigned by the OS at the time of process creation.
2. **Process state:** It contains the state of the process where the process is currently residing. The state may be new, ready, running, waiting, halted and so on.
3. **Program counter:** It is having the address of the next instruction to be executed.
4. **Priority:** It is also a parameter assigned by the OS at the time of process creation.
5. **General purpose register**
6. **List of open files**
7. **List of closed files**

All the attributes of a process are stored in the process control block (PCB) (Fig. 7.7).

Process Id	*Process State*
Process counter	Priority
List of open files	List of close files
General purpose register information	Accounting information
I/O status information	CPU scheduling information
Memory management information	Context data memory pointers

Figure 7.7 A process control block.

7.3.1.2 Process States

Any process is defined by a unique characteristic of its state, which informs about the current activity of the process. A process may be in one of the following states during its execution:

1. **New:** Whenever a new process is created.
2. **Running:** Whenever the process is able to get attention from the busy CPU, that is, the CPU is actually processing the instruction.
3. **Waiting:** The process waits for its turn as it requires some resource(s) which is/are held by another process(es).
4. **Ready:** A process has been allocated all the resources required and is waiting for the attention

from CPU because the CPU is busy in executing some other task.

5. **Terminated:** A process is in terminated state when the processor has finished the execution of a particular task.
6. **Suspended ready:** If processes are more than the capacity of the memory, then they will be suspended and go to suspended ready state, which is in the secondary memory.
7. **Suspended wait:** If processes are more than the capacity of memory, then they will be suspended and go to suspended waiting state, which is in the secondary memory.

7.3.2 Schedulers

An operating system has three types of scheduler (Fig. 7.8):

1. **Long-term scheduler:** It is responsible for bringing a newly created process into the system.
2. **Short-term scheduler:** It is responsible for selecting one of the processes from the ready state and scheduling that process into the running state.
3. **Mid-term scheduler:** It is responsible for suspending and resuming the processor.

7.3.3.1 Dispatcher

It is responsible for saving and loading the context of a process. The context switching is done by a dispatcher.

7.3.3.2 Degree of Multiprogramming

The number of jobs present in the memory is called as the degree of multiprogramming. Long-term scheduler controls the degree of multiprogramming.

Note: The best data structure to implement ready queue is double linked list.

7.3.3.3 Time Periods

A process has various time periods as follows:

1. **Arrival time (AT):** When a process arrives into ready state, it is called as arrival time.

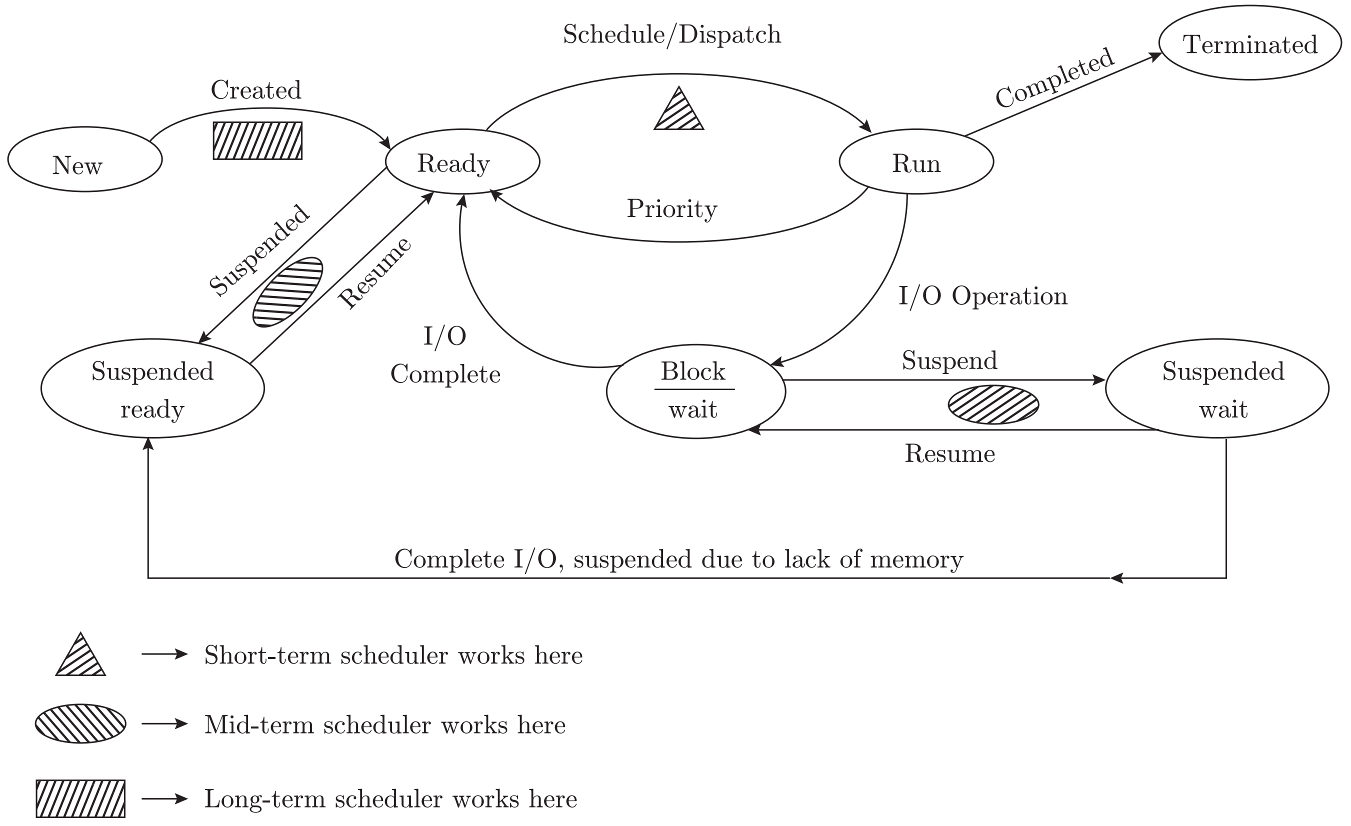


Figure 7.8 Lifecycle of a scheduler.

2. **Burst time (BT):** The time required by a process for its execution is called as burst time.
3. **Completion time (CT):** The time when a process completes its execution is called as completion time.
4. **Turn-around time (TAT):** The time difference between completion time and arrival time is called as turnaround time.

$$\text{TAT} = \text{CT} - \text{AT}$$

5. **Waiting time (WT):** Waiting time for a process is the time duration which is spent in waiting queue by that process.

$$\text{WT} = \text{TAT} - \text{BT}$$

6. **Response time:** The first scheduled time for a process is called response time.

7.4 CPU SCHEDULING

CPU scheduling is the technique to put a process from ready queue to running state. The objective of CPU scheduling is to minimize the turnaround time and average waiting time of a process. The short-term scheduler is used to apply the CPU scheduling in the ready state to select the process to schedule on to the processor (running state).

7.4.1 Scheduling Algorithms

There are many CPU scheduling algorithms as follows:

1. First come, first serve (FCFS)
2. Shortest job first (SJF)
3. Shortest remaining time first (SRTF)
4. Round robin
5. Priority based
6. Highest response ratio next (HRRN)
7. Multilevel queue
8. Multilevel feedback queue

7.4.1.1 First-Come, First-Serve Scheduling

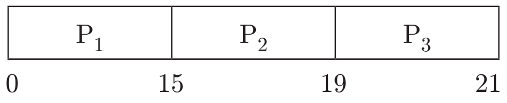
In this simple scheme, the allocation of CPU is on the basis of first come, first serve (FCFS). A queue data structure is maintained for FCFS scheduling. When the wait of the process is over and it enters into the ready state, its PCB is linked onto the tail of the queue. The process at the head of the queue is allocated to the processor and the running process is removed from the queue. The FCFS scheduling is non-preemptive.

Example 7.1

Consider the following workload table of three processes:

Process	Burst Time (ms)
P_1	15
P_2	4
P_3	2

Let us suppose that the process arrives at time = 0 in the order P_1, P_2, P_3. By applying the FCFS scheduling algorithm, we get the result shown in the following Gantt chart:



Waiting Time:

Process	Waiting Time (ms) WT = (TAT − BT)
P_1	0
P_2	15
P_3	19

Average waiting time = (Total waiting time)/3
= (0 + 15 + 19)/3 = 34/3 = 11.33 ms

Turnaround Time:

Process	Turnaround Time (ms) TAT = (CT − AT)
P_1	15
P_2	19
P_3	21

Average turnaround time = (15 + 19 + 21)/3
= 55/3 = 18.33 ms.

7.4.1.2 Shortest-Job-First Scheduling

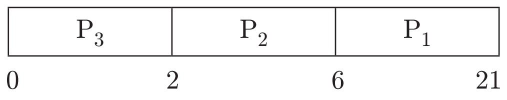
In SJF scheduling, the job that requires CPU attention for lesser time is chosen first for CPU attention. The data structure used is a queue in which the jobs are arranged on the basis of CPU time required by the process (in ascending order). If there are two processes that have the same requirement of CPU time, then the order in which they arrived (FCFS scheduling) is used to break the tie. This algorithm can be either pre-emptive or non-preemptive. Pre-emptive SJF scheduling is sometimes called shortest-time-first scheduling (STF).

Example 7.2

Consider the same Example 7.1:

Process	Burst Time (ms)
P_1	15
P_2	4
P_3	2

Assuming that all these processes arrive at the same time, using SJF scheduling, we would schedule these processes according to the following Gantt chart:



Waiting Time:

Process	Waiting Time (ms) WT = (TAT − BT)
P_1	6
P_2	2
P_3	0

Average waiting time = (6 + 2 + 0)/3 = 8/3 = 2.66 ms

Turnaround Time:

Process	Turnaround Time (ms) TAT = (CT − AT)
P_1	21
P_2	6
P_3	2

Average turnaround time = (21 + 6 + 2)/3 = 29/3
= 9.66 ms

7.4.1.3 Shortest Remaining Time First

SRTF is a pre-emptive version of the SJF. It permits processes that enter the ready list to pre-empt the running process if the time for the new process (or for its next burst) is less than the remaining time for the running process (or for its current burst).

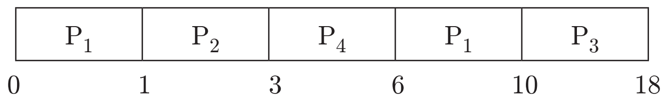
Example 7.3

Consider the following example. We have the workload table of four processes.

Process	Arrival Time (ms)	Burst Time (ms)
P_1	0	5
P_2	1	2
P_3	2	8
P_4	3	3

1. Now, P_1 is executed first, because it is the only process available at that time ($t = 0$).
2. At $t = 1$, we have a new process P_2 which requires only 2 ms whereas the already executing process P_1 still requires 4 ms to complete. Therefore, the CPU will start executing P_2 because it has the shortest running time out of the two available processes P_1 and P_2.
3. At $t = 2$, we have three processes, namely P_1, P_2 and P_3, P_2 has the shortest remaining time, therefore it will continue to execute.
4. At $t = 3$, process P_2 terminates and a new process P_4 enters the ready queue. Now, we have three processes P_1, P_3 and P_4 waiting for CPU attention. P_4 has the shortest remaining time (3 ms) as compared to P_1 (4 ms) and P_3 (8 ms).

The Gantt chart is shown below:



Turnaround Time:

Process	Turnaround Time (ms) TAT = (CT − AT)
P_1	10 − 0 = 10
P_2	3 − 1 = 2
P_3	18 − 2 = 16
P_4	6 − 3 = 3

Average turnaround time = (10 + 2 + 16 + 3)/4
= 7.75 ms

Waiting Time:

Process	Waiting Time (ms) WT = (TAT − BT)
P_1	10 − 5 = 5
P_2	2 − 2 = 0
P_3	18 − 8 = 10
P_4	3 − 3 = 0

Average waiting time = (5 + 0 + 8 + 10)/4 = 5.75 ms

7.4.1.4 Round-Robin Scheduling

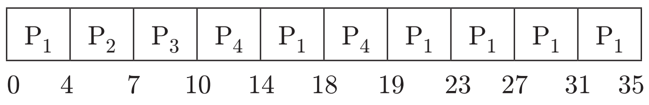
Round-robin scheduling is basically similar to FCFS scheduling, but pre-emption is added to switch between processes. A small unit of time, generally from 10 to 100 ms, called a time quantum (or time slice), is defined. The ready queue is treated as a circular queue. The round-robin (RR) scheduling algorithm is designed especially for time-sharing systems.

Example 7.4

Consider the following example. We have the workload table of four processes, and time quantum of 4 ms is used.

Process	Burst Time (ms)
P_1	21
P_2	3
P_3	3
P_4	5

Using RR scheduling, we would schedule these processes according to the following Gantt chart:



Waiting Time:

Process	Waiting Time (ms) WT = (TAT − BT)
P_1	35 − 24 = 11
P_2	7 − 3 = 4
P_3	10 − 3 = 7
P_4	19 − 5 = 14

Average waiting time = (11 + 4 + 7 + 14)/4 = 9 ms

Turnaround Time:

Process	Turnaround Time (ms) TAT = (CT − AT)
P_1	35
P_2	7
P_3	10
P_4	19

Average turnaround time = (35 + 7 + 10 + 19)/4
= 17.75 ms

7.4.1.5 Priority Scheduling

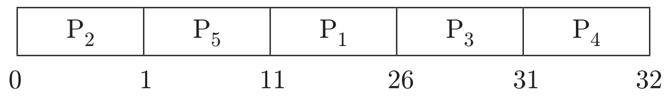
In priority scheduling, each process is assigned a priority and the highest priority process is allocated to the CPU first. The data structure used is a priority queue, that is, the process with the highest priority is in the front of the queue. If there are two processes that have the same priority, then their order of arrival (FCFS scheduling) is used to break the tie. The major drawback of this algorithm is starvation, that is, a low-priority process have to wait for an infinite time because the high priority processes gets executed first and after some time another high priority process comes and the low priority process is again left out.

Example 7.5

Consider the following example. We have the workload table of five processes.

Process	Burst Time (ms)	Priority
P_1	15	3
P_2	1	1
P_3	5	4
P_4	1	5
P_5	10	2

Using priority scheduling, we would schedule these processes according to the following Gantt chart:



Waiting Time:

Process	Waiting Time (ms)
P_1	11
P_2	0
P_3	26
P_4	31
P_5	1

Average waiting time = (11 + 0 + 26 + 31 + 1)/5
= 69/5 = 13.8 ms

Turnaround Time:

Process	Turnaround Time (ms)
P_1	26
P_2	1
P_3	31

(*Continued*)

Continued

Process	Turnaround Time (ms)
P_4	32
P_5	11

Average turnaround time = (26 + 1 + 31 + 32 + 11)/5
= 20.2 ms

7.4.1.6 Highest Response Ratio Next (HRRN)

In HRRN algorithm, a processor is assigned to a process on the basis of response ratio. Response ratio is calculated by the following formula:

$$\text{Response ratio} = \frac{w+s}{s}$$

where w = waiting time and s = service time or burst time.

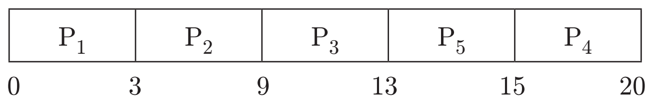
The HRRN algorithm favours the shorter jobs and limits the waiting time of larger jobs. It is non-preemptive scheduling algorithm.

Example 7.6

Consider the following example. We have the workload table of five processes.

Process	Burst Time (ms)	Arrival Time
P_1	3	0
P_2	6	2
P_3	4	4
P_4	5	6
P_5	2	8

Using HRRN scheduling, we would schedule these processes according to the following Gantt chart:



1. Now P_1 is executed first, because it is the only process available at that time ($t = 0$).
2. At $t = 2$, we have a new process P_2 which requires only 6 ms whereas the already executing process P_1 still requires 1 ms to complete. Therefore, the CPU will continue to execute P_1 because it is non-preemptive scheduling.
3. At $t = 3$, we have only one process P_2, so P_2 will get executed.
4. At $t = 9$, we have three processes P_3, P_4 and P_5. To select which process will be executed next, we have to calculate the response ratio for all of the three processes. The process having the highest response ratio will be executed first.
   At $t = 9$,
   Response ratio of $P_3 = (w + s)/s = (5 + 4)/4 = 2.25$
   Response ratio of $P_4 = (w + s)/s = (3 + 5)/5 = 1.60$
   Response ratio of $P_5 = (w + s)/s = (1 + 2)/2 = 1.5$
   Process response ration of P_3 is high, so it will be executed next.
5. Repeat the above steps for all the processes.

Waiting Time:

Process	Waiting Time (ms) WT = TAT − BT
P_1	0
P_2	1
P_3	5
P_4	9
P_5	5

Average waiting time = (0 + 1 + 5 + 9 + 5)/5 = 20/5
= 4 ms

Turnaround Time:

Process	Turnaround Time (ms)
P_1	3
P_2	7
P_3	9
P_4	14
P_5	7

Average turnaround time = (3 + 7 + 9 + 14 + 7)/5
= 8 ms

7.4.1.7 Multilevel Queue Scheduling

In multilevel queue scheduling, the ready queue is partitioned into several separate queues such that each process belongs to some queue. Various properties for allocating jobs to different queues may depend upon memory size, process priority or process type, or any other parameter(s). Each queue may follow different scheduling algorithms. Figure 7.9 depicts the multilevel queue scheduling.

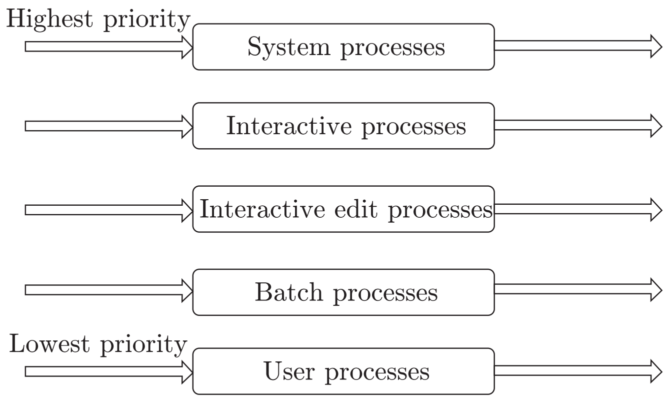


Figure 7.9 | Multilevel queue scheduling.

7.4.1.8 Multilevel Feedback Queue Scheduling

In multilevel feedback queue scheduling, the multilevel queue scheduling approach has been enhanced. Here, the process is not permanently assigned to a queue but a process is allowed to move between queues. The major advantage of this approach is that it prevents starvation, because a process which is a low-priority process can be moved to a higher-priority queue after some time.

Figure 7.10 depicts a three-level feedback queue scheduler. The scheduler first executes all processes in the first queue. When this queue is empty, it will execute processes in the next queue (having a higher quantum or time-slice). Similarly, processes in subsequent queue will be executed only if the previous queue(s) are empty. Further, a process that arrives for lower queue (having lower quantum) will pre-empt a process in queue having a higher quantum.

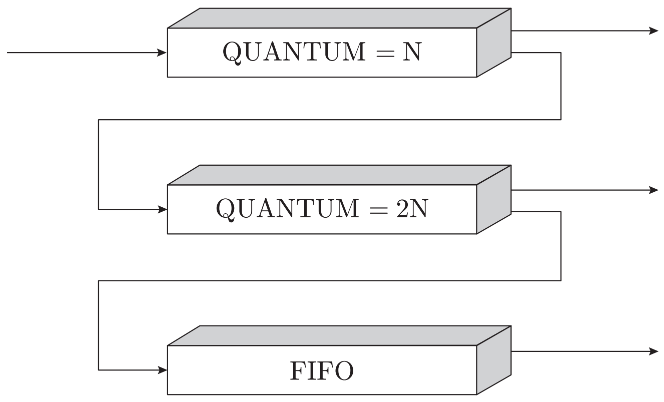


Figure 7.10 | Multilevel feedback queue scheduling.

Note: Apart from the above well-known scheduling algorithms, there are two other approaches also.

1. **Guaranteed scheduling:** This scheduling algorithm guarantees the user that particular amount of attention will be given to user. For example, if there are n users, then we can say that each user will get about $1/n$ of the CPU time. This seems to be a good scheduling algorithm theoretically but is harder to implement due to overheads involved in switching of processes.
2. **Lottery scheduling:** This is similar to lottery. Whenever a scheduling decision has to be made, any process is selected at random and is given more amount of CPU time as compared to other processes similar to the lottery winner.

There are two different categories of schedulers–Dynamic and Static. A dynamic scheduling can be defined as a scheduling defined at run-time; whereas in static the scheduling decision is taken before the system starts running. There are some popular dynamic scheduling algorithms such as:

1. **Rate monotonic algorithm:** A priority value is assigned to each process depending upon the frequency of the occurrence. A process with high frequency occurrence gets a high priority. So, at run-time the highest priority process gets the CPU.
2. **Earliest deadline first:** The processes are sorted according to the deadline (say, in RTOS) and the CPU time is allocated to the first process in the list, that is, which has to meet the deadline urgently.
3. **Least laxity algorithm:** A process that has the least spare-time (laxity) is given to the CPU.

7.5 PROCESS SYNCHRONIZATION

7.5.1 Types of Processes

A process consists of two major elements namely, the program code and a data set. The processes can be classified as follows:

1. **Independent process:** This is a process which cannot affect or get affected by execution of another process.
2. **Cooperating process:** This is a process that affects and gets affected by execution of other processes.
3. **Concurrent processes:** These processes are independent, interacting processes which are executed simultaneously over a period of time.

When two processes want to use or access a single global variable, then the problem of inconsistency occurs. The problem can be solved using the process synchronization.

7.5.2 Interprocess Communication

Inter-process communication is a technique to exchange data among multiple processes or threads. IPC allows programmers to schedule activities to different processes. One process controls the whole communication. There are different IPC techniques available:

- Message passing
- Shared memory
- Remote procedure calls
- Synchronization

7.5.3 Classical Synchronization Problems

Classical synchronization problems are used to implement synchronization mechanisms and critical sections. Some of the interesting problems are: **The Producer-Consumer Problem, The Readers-Writers Problem, The Dining Philosophers Problem, Sleeping Barber, Cigarette Smoker Problem**, etc. The following sections discuss only the first three of them.

7.5.3.1 The Producer-Consumer Problem

To further understand the concept, let us take an example of producer—consumer problem. A producer produces data that is consumed by a consumer (e.g., spooler and printer). A buffer holds the produced data, which is not yet consumed. There exist several producers and consumers, and initially, we assume the buffer is unbounded.

Let count be a global variable. Count = 5

Producer: Register1 = Count [5]

Producer: Register1 + = 1 [6]

Consumer: Register2 = Count [5]

Consumer: Register2 = Register2 -1 [4]

Producer: Count = Register 1 [6]

Consumer: Count = Register 2 [4]

As we can see, output will be either 4 or 6. But it should be 5.

Here, output depends on the order of execution of statements. Synchronization is required, otherwise output will be inconsistent.

7.5.3.2 The Readers-Writers Problem

In this problem, there are many concurrent processes that try to access the same file. Some processes try to read data from file or some want to insert or change the contents in the file. According to Bernstein constraint, many readers may access the file at the same time, but when the writer is accessing the file, no one should be allowed to access it. This can be ensured by following methods:

1. Assign readers more priority than writers. If a writer has no permission to access the file, any reader can get access to it. But this may cause infinite waiting for the writer.
2. Another method can be used to assign writers large priority over readers. When writer requests for file, all the current readers finish their access and new readers have to wait until the writer finishes its task. This may cause infinite waiting for reader.

7.5.3.3 The Dining Philosopher's Problem

Another classic problem of synchronization is the dining philosopher's problem. Here five philosophers sit around a circular table with five chairs. There are five plates and forks placed in a circular manner (Fig. 7.11). When the philosopher wants to eat, s/he picks up forks from left and right of his plate. When finishes eating, s/he keeps both the forks back. This ensures that no philosopher starves. Two problems are faced in this:

1. Each philosopher picks up one fork and none of them gets a second fork.
2. No philosopher should starve due to another.

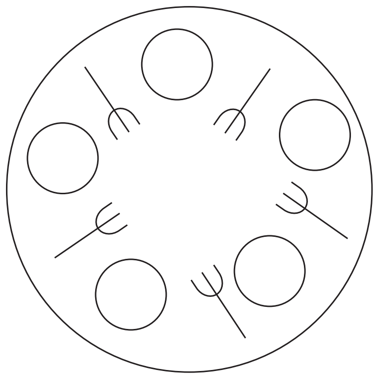


Figure 7.11 | The Dining Philosopher's Table.

7.5.4 Race Condition

When several processes have access to shared data, then the final value of this shared data will eventually depend upon the action performed by the last process. This is a situation in which the result depends upon which process is executed first. Such a situation or condition is known as “race condition”. The producer—consumer example discussed in the previous section is one such example. The part of the program where the shared memory is accessed is known as critical section. The following are the four conditions that are used to provide a good solution to this problem:

1. No assumption has to be made about the number of CPUs and their speed.
2. No two processes may be executing their critical section at the same time.
3. No process has to wait indefinitely to enter its critical region.
4. No process executing outside its critical section can block other processes.

7.5.5 Critical Section

Critical section is a part of program global variables which are manipulated by the processes.

7.5.5.1 Critical Section Problem

Critical section problem occurs when n number of processes are competing for critical section. The problem is to ensure that if one process is in the critical section, no other process is allowed to enter into critical section at the same time.

```
do {
     ┌───────────────┐
     │ Entry section │
     └───────────────┘
              Critical Section
     ┌───────────────────┐
     │Remainder section  │
     └───────────────────┘
              Remainder Section
   } while(1);
```

Busy Waiting

Spinning or busy waiting is a mechanism in which the process keeps on checking for the condition to be true. For example the process is checking repeatedly whether a lock is available or not. This technique can be used to generate the random time delays. This consumes so many CPU cycles.

```
do
{
    wait(s);
// critical section
    signal(s);
// remainder section
} while(1);
```

Processes keep on checking for semaphore value to be zero. This continuous looping problem is busy waiting. When semaphore does busy waiting, it is called ‘spinlock’.

7.5.5.2 Solutions to Critical Section Problem

1. **Mutual exclusion:** If process P_i is executing in its critical section, then no other process can execute their critical sections.
2. **Progress:** If no process is executing in its critical section and there exist some process that wishes to enter into critical section, then the selection of process that will enter into critical section cannot be postponed indefinitely.
3. **Bounded waiting:** A bound must exist on the number of times that other processes are allowed to enter into critical section after a process has made the request for critical section and it is not granted.

7.5.6 Peterson’s Algorithm for Two Processes

Peterson’s algorithm is considered the most correct algorithm for process synchronization.

P_i	P_j
`do{`	`do{`
`flag[i] = true;`	`flag[j] = true;`
`turn = j;`	`turn = i;`
`while(flag[j] and turn = j);`	`while(flag[i] and turn = i);`
`Critical Section`	`Critical Section`
`flag[i] = false`	`flag[j] = false`
`Remainder Section`	`Remainder Section`
`}while(1);`	`}while(1);`

The above code ensures mutual exclusion, progress and bounded wait.

7.5.7 Semaphores

Semaphores are synchronization tools. These are the shared variables, which can be implemented as integer variables. These can be accessed via two indivisible atomic operations that are wait and signal.

7.5.7.1 Integer Implementation of Semaphores

The operations (wait and signal) are realized by making use of integer values.

1. **Wait:** It reduces the value of semaphore by 1 unit. It can be denoted by $P(S)$.
   Wait(S):
```
while S ≤ 0
      Do no_operation;
S--;
```
2. **Signal:** It increases the value of semaphore by 1. This can be denoted by $V(S)$.
   Signal(S):
```
S++;
```
3. **Disadvantages:**
   - We must keep the value of $S = 1$
   - Busy waiting which consumes many CPU cycles

7.5.7.2 Binary Semaphores

The value of binary semaphores can range between 0 and 1 only. Binary semaphores are also known as mutex locks as they ensure mutual exclusion.

Problem 7.1: Each process $P_i = 1, 2, \ldots 11$ is coded as follows:

```
P (mutex)
   Critical Section
V (mutex)
```

The code for P_{10} and P_{11} is identical except it uses V in the place of P and P in the place of V.

How many processes will be in critical section?

Solution:

Let $S = 1$

P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}	P_{11}
P	P	P	P	P	P	P	P	P	P	V
V	V	V	V	V	V	V	V	V	V	P

Say P_1 enters in the critical section, $S_1 \rightarrow 0$.
Now P_{11} process can increment $S \rightarrow 1$ and will have access to critical section.
Now any process can enter into critical section, because S is incremented by P_{11}.
Three processes will enter into critical section.

Note: 1. With change in one sequence, three processes will enter in the critical section.

2. With change in two sequences, five processes will enter into the critical section and so on.

7.5.7.3 Counting Semaphores

This semaphore makes use of a ‘count’ variable. This variable is initialized by the numeric value of the finite number of instances of a resource. The count value is decremented whenever the instance of that resource is allocated to a process. So, if the count value of a resource is zero, it means that all instances of that resource have been allocated and no process can enter into critical section. Further, when a process releases a resource this count value is incremented.

Implementation of Counting Semaphore

1. **Wait operation**

   Wait (S):
```
{
        C--;
        if(C < 0)
        {
               Block the process and put
                 in blocked queue
        }
}
```
2. **Signal operation**

   Signal (S):
```
{
        C++;
        if(C ≤ 0)
        {
            Remove process from block queue
              and keep in ready queue
        }
}
```

Example 7.7

Counting semaphore = 7

$P = 20$ [decrement semaphore]

$V = 15$ [increment semaphore]

Value after implementing P and V: $C - P + V$

$$7 - 20 + 15 = 2$$

Table 7.2 summarizes the common concurrency mechanisms

Table 7.2 | Common concurrency mechanisms

1.	Semaphore	An integer value is used for signalling between different processes. Only three operations are performed – initialize, increment and decrement. Also known as counting semaphore.
2.	Binary Semaphore	Only two values 0 and 1 are used.
3.	Mutex	Similar to binary semaphore (lock and unlock values are used) but the major difference is that the process which has locked the mutex is the only one that can unlock it.
4.	Monitor	It is a line of code that contains access procedures, initialization code and variables. The access procedures are critical sections.
5.	Event Flag	It is a memory word (such as accumulator flags), which is used for synchronization.
6.	Condition Variable	A particular data type is used to block a given process until some specified condition is not met.
7.	Mailboxes	The two processes can exchange information through mailbox and also can take its help for synchronizing their activities.
8.	Spinlocks	A process executes in an infinite loop until the locked variable is available.

7.5.8 Deadlock and Starvation

P_0	P_1
wait(S);	wait(Q);
wait(Q);	wait(S);
.	.
.	.
.	.
.	.
signal(S);	signal(Q);
signal(Q);	signal(S);

In the above code, let $S = 1,\ Q = 1$

Process P_0 will make $S \rightarrow 0$

Process P_1 will make $Q \rightarrow 0$

Both will not be able to move further. This is the deadlock state.

7.6 DEADLOCKS

A deadlock is a situation where computer programs sharing the same resources are permanently blocked. As a result of which, each program prevents the other from acquiring the resources required for its completion. Let us consider an example, there are two friends (processes) and they intend to write a letter. To accomplish this task they need two pens and two pages (resources). Now, one of them has pages (papers) and the other a pen. A possible solution would have been that one person can give his resource to another so that the other person completes the job (write the letter) and after his job is finished this person can also complete the job. But none of them is willing to part the resources s/he has. This is a simple case of deadlock.

7.6.1 Resource Types

There are two types of resources, namely, pre-emptable and non-preemptable resources.

1. **Pre-emptable:** These resources can be taken away from the process with no ill effects, for example, CPU, memory.
2. **Non-preemptable:** These resources can be taken away from the process (without causing any ill effect). For example, CD, USB drive, plotter, printer, etc. We cannot give such resources to other processes in the middle of their jobs.

7.6.2 Characteristics of a Deadlock

The following four conditions must exist simultaneously for a deadlock to occur:

1. **Mutual exclusion condition:** The resources involved are non-shareable. Each resource must be assigned to only one single process at a time.
2. **Hold and wait condition:** The process currently holds a resource and is waiting to acquire another requested resource held by other process(es).
3. **Non-preemptive condition:** Resource(s) that have already been acquired by a process cannot be taken back forcibly unless and until they are released by the holding process.
4. **Circular wait condition:** The processes form a circular list or chain, where each process in the list is waiting for a resource held by another process.

To understand these four concepts clearly, let us take an example of the road as a resource, shown in Fig. 7.12.

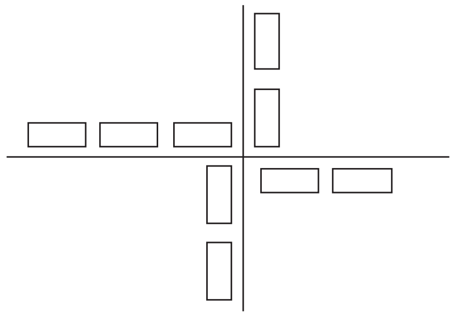


Figure 7.12 A traffic deadlock.

1. **Mutual exclusion condition:** Only one vehicle can be moved.
2. **Hold and wait condition:** Each vehicle occupies some part of the road and is waiting to move to the other side of road.
3. **Non-preemptive condition:** Road space occupied by a vehicle cannot be taken away.
4. **Circular wait condition:** Each vehicle on the road is waiting for the next vehicle to move.

7.6.3 Resource Allocation Graph

A resource allocation graph is a graph which has two different types of nodes, the process nodes (represented by circle) and resource nodes (represented by rectangles). For different instances of a resource, there is a dot in the resources nodes (rectangle). For example, if there are two identical drives, we will depict it with two dots.

The directed edges among these nodes represent resource allocation and release. An edge from the resource to the process node denotes that it has been acquired by the process, whereas an edge from the process node to the resource node denotes that the process is requesting for the resource. For example, consider the resource allocation graph in Fig. 7.13 and we can easily conclude that there is a deadlock.

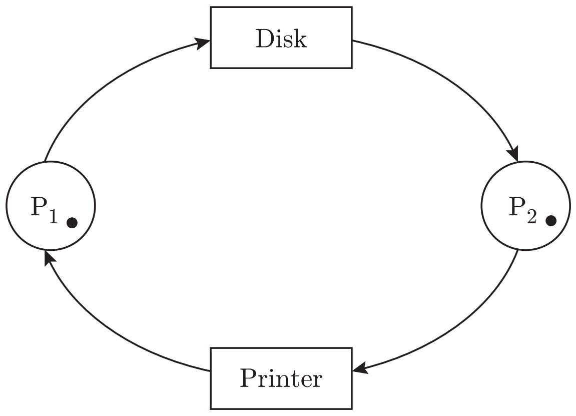


Figure 7.13 Resource allocation graph.

By making use of resource allocation graph we can check for deadlock (i.e., the graph has a circle) and if there is no circle, it indicates the lack of circular-wait condition, hence deadlock will not occur. This is depicted in Fig. 7.14.

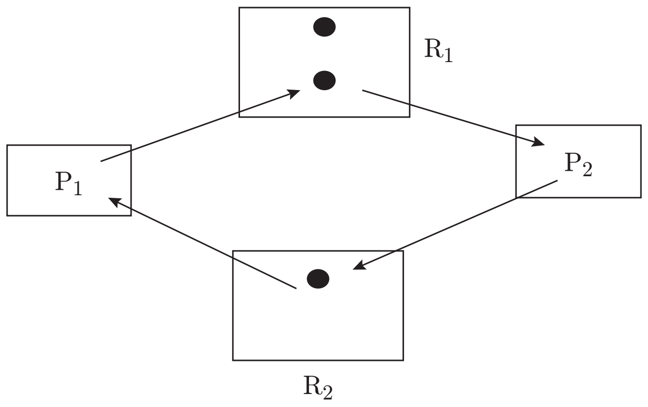


Figure 7.14 A resource allocation graph having a cycle and no deadlock. (The figure also depicts an alternative way of representing resource allocation graphs.)

The following conditions are depicted in the resource allocation graph shown above:

1. **There are three sets of process (P), resource (R) and edges (E):**
   $P = \{P_1, P_2\}$
   $R = \{R_1, R_2\}$
   $E = \{R_1 \rightarrow P_1, P_1 \rightarrow R_2, R_2 \rightarrow P_2, P_2 \rightarrow R_1\}$
2. **Resource instances:**
   Two instances of R_1
   One instance of R_2
3. **Process states:** Process P_1 is holding an instance of resource type R_1 and is waiting for an instance of resource type R_2. Further, process P_2 is holding an instance of resource type R_2 and is waiting for an instance of resource type R_1.

The following tips help us to check the resource allocation graph easily and predict the presence of cycles.

1. If there is no cycle in the resource allocation graph, there is no deadlock.
2. If there is a cycle in the graph and each resource has only one instance, then there is a deadlock. In such a case, the presence of a cycle is the necessary condition in order for deadlock to occur (as there is only one instance of the resource).
3. If there is a cycle in the graph, and each resource has more than one instance, there may or may not be a deadlock. Therefore, we conclude that a cycle in the resource allocation graph is a necessary but not a sufficient condition for a deadlock in the case of multiple instances of a resource.

The following table summarizes the above discussion

Possibility of Deadlock (03 conditions)	**Existence of Deadlock (04 conditions)**
Mutual Exclusion, No Preemption & Hold and wait	Mutual Exclusion, No Preemption, Hold and wait & Circular Wait

A deadlock can be dealt by following any of the three strategies:

1. Deadlock prevention
2. Deadlock avoidance
3. Deadlock detection and recovery

7.6.4 Deadlock Prevention

To prevent a deadlock, we must ensure that any one of the following four conditions listed below (Section 7.5.2) should not hold. Let us consider them one by one:

1. **Mutual exclusion condition:** Let us take the example of tossing a coin, getting a head or tail is mutually exclusive, which means that we cannot have both head and tail. The mutual exclusion condition means that there are some resources (such as CD Drive, Printer, etc.) which once allocated to a process cannot be taken back unless and until the process releases it (i.e., non-preemptable resources). So, such resources should be allocated to only one process at a time. They cannot be shared. Although, this does not mean that none of the resources can be shared. Certain pre-emptable resources (such as memory, CPU) can be shared, because they will never result in a deadlock. So, those resources which cannot be shared are mutually exclusive and this condition needs to be taken care by OS.
2. **Hold and wait condition:** In order to ensure this condition never occurs in the system, we have to devise a strategy **such** that a process requests all its required resources at one time and block**s** the process until all requests can be granted. This approach has its drawback that a process cannot know what resources would be required before it actually starts execution, and sometimes it may have to wait for a long time in order to get all its required resources; whereas it could have used some of the already allocated resources and released them back to the pool to be used by other processes. For example, a program may require five CDs, there should be five free drives before the execution begins, while the program might be using only one drive to begin with. Thus, this approach leads to serious waste of resources.
3. **Non-Preemption condition:** The non-preemption condition can be overcome by the following protocol. If a process is holding some resources and requests another resource that cannot be immediately allocated to it (which means, the process must wait), then all resources currently being held are pre-empted (released). These pre-empted resources are added to the list of resources. For example, when a process releases resources, the process may lose all its work to that point. The major disadvantage of this strategy is the possibility of indefinite postponement (starvation). However, this protocol is often applied to resource whose state can be easily saved and restored later, such as CPU registers and memory space. It cannot generally be applied to printers and drives.
4. **Circular wait condition:** A linear ordering of different resources can be used to allocate the resources to a process, which means that if a process has been allocated a resource, then it can only be allocated a resource in the linear order.

Another way to implement it could be use some kind of indexing or a subscript. Such a strategy is helpful in avoiding cycle(s) in resource allocation graph and results in overcoming the circular wait condition.

7.6.5 Deadlock Avoidance

The side effects of preventing deadlocks are similar to the side effects of walking carefully so that you donot fall as compared to walking freely. These are as follows:

1. Low device utilization
2. Reduced system throughput

Another approach is to anticipate a deadlock before it actually occurs. This method differs from deadlock prevention, which guarantees that deadlock cannot occur by denying any of the four necessary conditions of the deadlock, whereas a 'Deadlock Avoidance' algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular wait condition. There are three basic pillars to attain this objective:

1. Safe state algorithm
2. Resource allocation graph algorithm
3. Banker's algorithm

7.6.5.1 Safe State

A safe state can be defined as a state in which there is no deadlock. This can be achieved if

1. All the requested resources are allocated to the process.
2. If a process needs an unavailable resource, it may wait until the same has been released by a process to which it has already been allocated. If such a sequence does not exist, it is known as unsafe state.

Taking care of these two rules will never result in a deadlock.

7.6.5.2 Resource Allocation Graph Algorithm

The resource allocation graphs can also be used for deadlock avoidance. The major limitation of this strategy is that it is not applicable to a resource allocation system with multiple instance of each resource type.

7.6.5.3 Banker's Algorithm (Proposed by Dijkstra's)

Banker's algorithm is used to avoid deadlocks, is discussed in the following text:

1. **Basic principle:** It is similar to a banking system, that is, to ensure that the bank never allocates its available cash such that it can no longer satisfy the needs of all its customers.
2. **Working:** When a new process enters the system, it must declare the maximum number of instances of each resources type it may need. This number should not exceed the total number of resources in the system.

Now, 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 or in unsafe state. If in safe state, then the resources requested are allocated, otherwise the process must wait until some other process releases the requested resource.

We need the following data structures:

1. **Available:** A vector of length m indicates the number of available resources of each type. If Available[j] = k, there are k instances available of resource type R_j.
2. **Max:** An $n \times m$ matrix defines the maximum demand of each process, if Max[i, j] = k, then process P_i may request at most 'k' instances of resource type R_j.
3. **Allocation:** An $n \times m$ matrix defines the number of resources of each type currently allocated to each process.
   - If Allocation[i, j] = k
   - Then, process P_i is currently allocated 'k' instances of resource type R_j.
4. **Need:** An $n \times m$ matrix indicates the remaining resources needed for each process.
   - If Need[i,j] = k
   - Then, process P_i may need 'k' more instances of resource type R_j to complete its task.
   - Further, Need[i, j] = Max[i, j] − Allocation[i, j]

The Banker's algorithm consists of the following two parts:

1. **Safety algorithm:** The algorithm for finding out whether a system is in a safe state can be defined as follows:
   - Let 'Work' and 'Finish' be vectors of lengths m and n, respectively.
     Work: = Available
     Finish [i]: = False, for i = 1,2,3, …, n
   - Find an 'i', such that
     Finish[i]: = False
     Need $i \leq$ work, if no such i exists, go to step 4.
   - Work: = Work + Allocation
     Finish[i]: = true
     Go to step 2.
   - If Finish[i]: = true for all i, then the system is in a safe state.

2. Resource request algorithm

If Request$_i$[j] = k, then process P$_i$ wants k instances of resources type R$_j$. When a request for resources is made by process P$_i$, the following actions are taken:

- If Request$_i$ ≤ Need$_i$, go to step 2.
  Otherwise, raise an error condition because the process has exceeded its maximum claim.
- If Request$_i$ ≤ Available, go to step 3.
  Otherwise, P$_i$ must wait, as the resources are not available.
- Let the system pretend to have allocated the requested resources to process P$_i$ by modifying the state as follows:
  Available: = Available − Request$_i$;
  Allocation$_i$: = Allocation$_i$ + Request$_i$;
  Need$_i$ = Need$_i$ − Request$_i$

Now, if the resulting transactions are safe, process P$_i$ is allocated its resources.

Time Complexity

Banker's algorithm considers each request as it occurs, and checks whether it leads to a safe state or not. If it does, the request is granted; otherwise, it is postponed for some time later. This algorithm has a time complexity of $O(n^2)$, where n is the number of processes.

Limitations of the Banker's Algorithm

The following are some limitations of the banker's algorithms:

1. A fixed number of processes and resources are required. Further, all the processes should know their requirement of resources before starting.
2. No other process can start until the algorithm has been completely executed.
3. It relies on the input information, the resources may get underutilized if inaccurate information is provided.
4. This algorithm itself takes good amount of time to execute, more information about processes and resources will result in more time to execute.
5. The requests granted should be in a finite time.
6. If a resource becomes unavailable (e.g., a CD drive stops functioning), it can result in an unsafe state.
7. It is difficult to have a priori information about the need of resources.

Problem 7.2: Consider a system having five processes and three resources types A, B and C. Resource type A has nine instances, B has four instances and C has seven instances. Initially, the snapshot of the given system is as follows:

Process	Allocated	Maximum	Available
	ABC	ABC	ABC
P$_0$	0,1,0	7,5,3	2,2,2
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	

Using Banker's algorithm, how can we avoid the deadlock?

Solution: We will start with the first part of the Banker's algorithm, as follows:

Step 1.1: We will first calculate the Need matrix

Need[i] = Max[i] − Allocation [i]

Process	Need
	ABC
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

Step 1.2: Now, we will decide which process should execute first, that is, what are the processes that need resources which are less than available.

Process	Need	Available
	ABC	ABC
P_0	7,4,3	2,2,2
P_1	1,2,2	

So, we will start with P_1 (because its resource need is less than the available resource) and after its completion all the resources will be free; hence, after completion of P_1 the available matrix contents will be

$$\begin{array}{rl} & \text{Available} \\ & 2\ 2\ 2 \\ + & \underline{2\ 0\ 0}\ (\text{resources held by } P_1) \\ & \underline{4\ 2\ 2} \end{array}$$

Step 1.3: Again let us see the status

Process	Need	Available
	ABC	ABC
P_0	7,4,3	4,2,2
P_1	Complete	
P_2	6,0,0	
P_3	0,1,1	
P_4	4,3,1	

Now, we observe that P_3 can be executed, and is considered to be executed.

Step 1.4: Again, let us see the status

Process	Need	Available
	ABC	ABC
P_0	7,4,3	6,3,3
P_1	Complete	
P_2	6,0,0	
P_3	Complete	
P_4	4,3,1	

So, P_4 executes and similarly we can say that P_2 and then P_0 will be executed, and as a result, the available matrix after execution of each process would be:

$$\begin{array}{ll} \text{Available} & 6,3,3 \\ P_4 \text{ executes} & \underline{0,0,2}\ (+) \\ & \underline{6,3,5} \end{array}$$

After the execution of P_2

$$\begin{array}{ll} \text{Available} & 6,3,5 \\ P_2 \text{ executes} & \underline{3,0,2} \\ & \underline{9,3,7} \end{array}$$

After the execution of P_0	
Available	9,3,7
P_0 executes	0,1,0
	9,4,7

Thus, the final status of the system under consideration will be:

Process	**Need**	**Available**
	ABC	**ABC**
P_0	Complete	9,4,7
P_1	Complete	
P_2	Complete	
P_3	Complete	
P_4	Complete	

Therefore, the proposed sequence of execution is [P_1, P_3, P_4, P_2, P_0]. Now, we will find whether this sequence of execution is a 'safe sequence'. We will now consider the resource request algorithm (the second portion) of the Banker's algorithm. In the beginning, we have:

Step 2.1:

Work = [2,2,2] (i.e., avail at the start)

Finish = [F, F, F, F, F] (the finish value of all processes is F meaning False)

So, P_1 is executed first, and we observe that

$Need_i < Work$: [1,2,2] < [2,2,2] holds good, and as a result we have

Work = [4,2,2]

Finish = [F,T,F,F,F] (the finish value of P_1 is T meaning True)

Step 2.2:

Next, we have P_3 executed and in this case the condition $Need_i < Work$ holds good, and as a result we have,

Work = [6,3,3]

Finish = [F,T,F,T,F]

After the execution of P_4, we have

Step 2.3:

Work = [6,3,5]

Finish = [F,T,F,T,T]

After the execution of P_2, we have

Step 2.4:

Work = [9,3,7]

Finish = [F,T,T,T,T]

After the execution of P_0, we have

Step 2.5:

Work = [9,4,7]

Finish = [T,T,T,T,T]

In the whole process, we have never encountered a situation where the condition $Need_i < Work$ does not hold good. Therefore, we can execute our processes in the order [P_1, P_3, P_4, P_2, P_0] without encountering the deadlock.

7.6.6 Deadlock Detection and Recovery

Deadlock detection is a process of detecting whether a deadlock has occurred. Deadlock occurs if neither avoidance nor prevention has been employed. To detect a deadlock, we simulate, in a recursive manner, the most favoured execution of each unblocked process.

1. Any unblocked process can acquire the resources needed for its execution.
2. All the acquired resources are released.
3. These released resources can be used by the previously blocked processes, which are in need.

Repeat the steps 1 to 3 above. If any blocked process remains, they are in a deadlock state.

7.6.6.1 Deadlock Detection

To deal with this concept further, we can discuss the different types of resources available, namely, reusable resources and consumable resources (Table 7.3).

Table 7.3 Reusable vs. consumable resources

Reusable Resources	Consumable Resources
Number of units are constant	Number of units may vary
Unit is either free or allocated; no sharing	Process may create new units
Process requests, acquires and releases the units	Process may consume the units
Example: memory, devices, files, tables	Example: messages, signals

We have already discussed about resource allocation graphs (Section 7.6.3) and now we will discuss about deadlock detection in two different scenarios:

1. **Single instance of each resource type:** If all the resources have only a single instance, then we use a variant of resource allocation graph known as **wait-for graph** (Fig. 7.15). A deadlock exists in the system if and only if the wait-for graph contains a cycle. To detect deadlocks, the system maintains the wait-for graph and an algorithm is invoked, periodically in order to check for cycle in the graph.
2. **Single instance of each resource type:** The wait-for graph scheme is not applicable to a resource system with multiple instances. The following algorithm is used for deadlock detection in such type of systems (somewhat similar to Banker's algorithm).

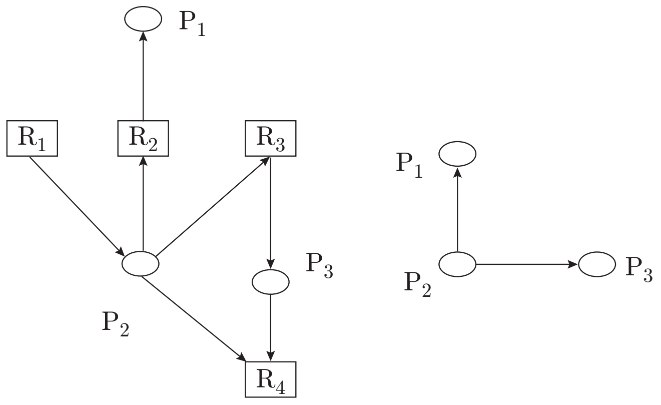


Figure 7.15 A resource allocation graph and the corresponding wait-for graph.

- **Available:** A vector of length m indicates the number of available resources of each type.
- **Allocation:** An $n \times m$ matrix defines the number of resources of each type currently allocated to each process.
- **Request:** An $n \times m$ matrix defines the current request of each process.

The algorithm is as follows:

- Let Work and Finish be vectors of length m and n, respectively. Initially,

  Work = Available.
- Find an index i such that both the following conditions are false:

  Finish[i] = False

  $\text{Request}_i \leq$ Work. If no such i exists, go to step 4.
- Work = Work + Allocation_i

  Finish [i] = True.

  Go to step 2.
- If Finish[i] = False, for some value of i, $1 \leq i \leq n$, then the system is in deadlock. Moreover, if Finish[i] = False, then it states that the process P_i is deadlocked.

7.6.6.2 Recovery from Deadlock

If at all a deadlock has been detected, any one of the following techniques are used by the OS to recover from the deadlock.

1. **Recovery by Process Termination:** Once the deadlock has occurred, our approach is to overcome the deadlock. This can be achieved if the processes that have been allocated the resources submit them back to the resource pool. This submission is done in a one-by-one fashion until the deadlock

is removed. Now, the question is that which process should submit first or the order of submission. Some of the possible parameters could be:

- ***The priority of a process*:** A process having a low priority could be the first one to give up the resources.
- ***Number of resources held*:** A process holding most of the resources could be the victim for termination or a process having the least number of allocated resources be the first.
- ***Remaining time for completion*:** In this approach, a process which needs more time for completion is terminated.
- ***Process type*:** Whether the processes are interactive (which means that if we terminate one process, all interactive processes will be terminated because it might be possible that one process in execution needs some input from the process that was terminated, hence this will not serve our purpose) or Batch processes (in this case entire batch has to be taken care of).

2. **Recovery by checkpointing and rollback (resource pre-emption):** Another way to recover from deadlock is making use of checkpoints and rollback. These two strategies involve saving the states of a running process at certain intervals, so that they can be terminated and start their execution from any of the saved states. The major overhead is extra memory and complexity to store the intermediate states.

Deadlock recovery is a high cost process, so is used when it is very rare and their might be a need for process migration. Such a concept is more popular in databases.

7.7 THREADS

A process can have more than one thread. A light-weight process is called thread. A process is generally known as a single-threaded process. Multithreaded process has more than one thread.

A process has code, data and files. A thread shares code, data and files with all other threads. A thread has its own registers and stacks which are non-sharable (Fig. 7.16).

7.7.1 Benefits of Threads

1. Improved responsiveness
2. Faster context switching (context-switching time for thread < context-switching time for process)
3. Resource sharing
4. Enhanced throughput of system
5. Effective utilization of multiprocessor systems
6. Economical to implement

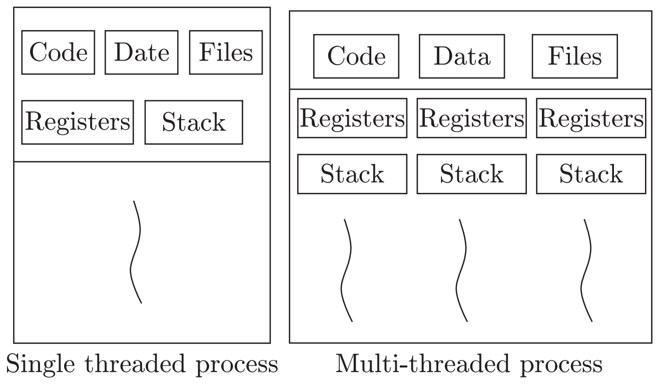


Figure 7.16 Threads and their types.

7.7.2 Types of Threads

A process can be composed of a single thread or multithreads. A thread (also known as lightweight process) includes the program counter, stack pointer and its own data area. A thread executes sequentially and is also interruptable. Whereas a process is composed of thread(s) and other system resources (data files, devices, RAM etc). The concept of multithreading is used to perform a number of independent tasks. There are two types of threads, namely,

1. User-level threads
2. Kernel-level threads

User-Level Threads	Kernel-Level Threads
These are created by user or programmer.	Kernel threads are created by operating system.
OS cannot recognize user threads.	These are recognized by OS.
If thread is performing I/O system call, then the entire process will be blocked.	If thread is performing I/O system call, then other threads will continue to execute.
User threads are dependent threads.	Kernel threads are independent threads.
It has less context-switching time.	It has more context-switching time.
User threads do not require hardware support.	Kernel threads require hardware support.

7.7.2.1 fork() Function

The fork is a system call which is used to create the child process. The fork() function returns a value of 0 (zero) to the newly created child process. The fork() returns a positive integer (process id of child process) to the parent process. The fork() returns a negative value if a child process creation is unsuccessful.

Note: If a program contains n fork() calls, then the number of created child processes are $(2^n - 1)$.

Problem 7.3: Consider the following code and find the total number of processes.

```
main()
{
fork ();
fork ();
fork ();
printf ("hello");
}
```

Solution:

Number of child process $= (2^n - 1) = 2^3 - 1 = 7 = 7$

Total number of process = Parent process + Child process = $(1 + 7) = 8$

7.8 MEMORY MANAGEMENT

The OS manages both the primary memory (RAM) and the secondary memory (hard disk). In the following section, we discuss various primary management algorithms followed by secondary memory management algorithms (hard disk scheduling algorithms). The OS manages the memory in an efficient and orderly manner. The OS has the following main responsibilities related to the memory—Isolation of a process (non-interference), Allocation and management (based upon an algorithm, the user does not decide but knows where the content is), Protection and access control (security) and reliable long-term storage.

7.8.1 Partitioning

It is a technique to divide memory into several parts for process use. Basically, two partitioning methods are used:

1. Fixed **p**artition **s**cheme: Memory is divided into several fixed-sized partitions (also known as static partitioning) (Fig. 7.17). The partition can contain only one process, therefore, the number of partitions is directly equal to the number of processes (concept of multiprogramming). So, whenever a process needs this resource any free partition is allocated and when the process is complete it releases the particular partition to be re-used by another process.

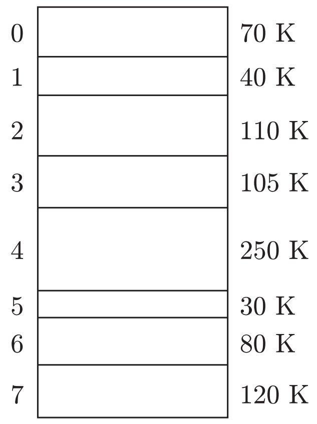


Figure 7.17 A fixed memory partition scheme.

2. Variable partition scheme: Variable-sized partition is also known as dynamic partitioning (Fig. 7.18). In this scheme, initially the memory will be full contiguous free block. Whenever it gets a request from a process, accordingly partition is made. The processes are accommodated in the memory according to their requirements and thus internal fragmentation is overcome. But external fragmentation exists.

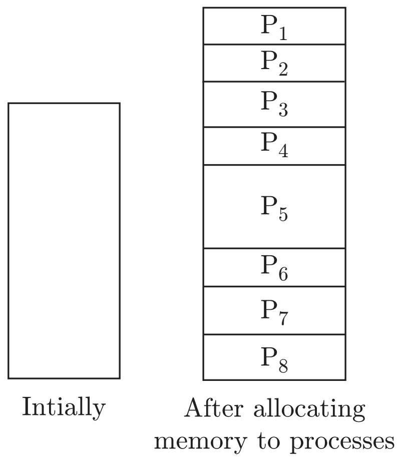


Figure 7.18 A variable memory partition scheme.

7.8.2 Partition Allocation Policies

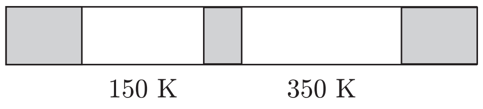
Whenever a process requests this resource, the allocation of free memory (partitions or holes) is done based upon different strategies. There are four different strategies:

1. **First Fit:** Allocation is made to the first free partition that is capable of meeting the requirements

of the process. The search can start from either end (up or down).

2. **Next Fit:** A variation of first fit, the allocation is started from the previous partition that was allocated.
3. **Best Fit:** The entire list of free partitions is scanned and the partition that fits best, that is, which results in a lesser free space after allocation is chosen.
4. **Worst Fit:** It is the exact opposite of best fit. The resource is allocated a partition that results in higher free space. The idea behind this strategy is that, a small process when allocated with a bigger partition will be able to make it free in a lesser time, this free partition may be used at a later stage by a process which might require a large partition; so that larger process does not have to wait for resource allocation.

Problem 7.4: Request from the processes is 300K, 25K, 125K and 50K, respectively.



The above request could be satisfied with? (Assume variable partition scheme)

(a) Best fit but not first fit
(b) First fit but not best fit
(c) Both best and first fit
(d) Neither best nor first fit

Solution:

We have to check for only two algorithms best fit and first fit (as given in question). Let us discuss by considering one algorithm at a time, as follows:

(a) In Best Fit:

1. First of all, the 300K jobs will use 350K slot. Free space in the slot will be = 350 − 300 = 50K.
2. Next process of 25K will use remaining 50K slot (available as free from the operation above), as per the best fit algorithm. So, the free space left = 50 − 25 = 25K.
3. Third process of 125K will be accommodated by 150K space, so we are left with 150 − 125 = 25K.
4. Fourth process requires 50K. Now, we have two slots of 25K left but our last process is of 50K size.

Therefore, the need of all the processes is not satisfied.

(b) In First Fit:

1. First process of 300K will use 350K slot. Remaining free space will be = 350 − 300 = 50K.
2. Next process of 25K will use remaining 150K slot (as per first fit algorithm). So, free space left = 150 − 25 = 125K slot.
3. Third process of 125K will be accommodating by 125K slot which is left free after the allocation of the memory to second process.
4. Now, we have only one slot of 50K left and our process requires 50K memory.

Observing, the execution of both the algorithms, we can say that by using the first fit algorithm, all processes can be executed. Therefore, option (b) is the correct answer.

7.8.3 Loading and Linking

A program may consist of different modules which are to be 'linked' and then 'loaded' in the Main Memory for execution. An analogy to understand it could be that a program in C is first linked (as it makes use of different header files) and then loaded in memory for execution. It is hereby worth-mentioning that the image of the program gets loaded into memory.

7.8.3.1 Loading

Loading is copying a program from secondary storage into main memory so it is ready to run. In some cases, loading just involves copying the data from disk to memory, in others it involves allocating storage, setting protection bits or arranging for virtual memory to map virtual addresses to disk pages. Loading can be done through the following two ways:

1. **Static Loading:** It is the technique in which the entire program loads into memory before the start of execution of the program. Some points of static loading are as follows:
   - Inefficient utilization of memory
   - Faster program execution
   - Use of static linking if static loading is used
2. **Dynamic Loading:** It is the technique in which the program is loaded into memory on demand. Some points of dynamic loading are as follows:
   - Efficient utilization of memory
   - Slower program execution
   - Use of dynamic linking if dynamic loading is used

The majority of OS is used dynamic loading and dynamic linking.

7.8.3.2 Linking (or Address Binding)

Association of program instruction and data to the actual physical memory location is called as address binding.

Instruction	Address
I_1	20
I_2	30
I_3	40
I_4	50

Address binding track to increment program counter. There are three types of address binding:

1. **Compiler Time:** If the compiler takes responsibility of associating a program and data to the actual physical memory location, it is called compile-time address binding. The compiler needs to interact with the operating system memory manager to perform compile-time address binding. This type of address binding is done before loading the program into memory.
2. **Load-time Address Binding:** This type of address binding is performed after loading the program into memory. The load-time address binding is performed by a loader.
3. **Run-Time (dynamic) Address Binding:** In the dynamic address binding, address binding is postponed till the time of execution. Dynamic binding is performed by the processor. The majority of OS uses dynamic binding.

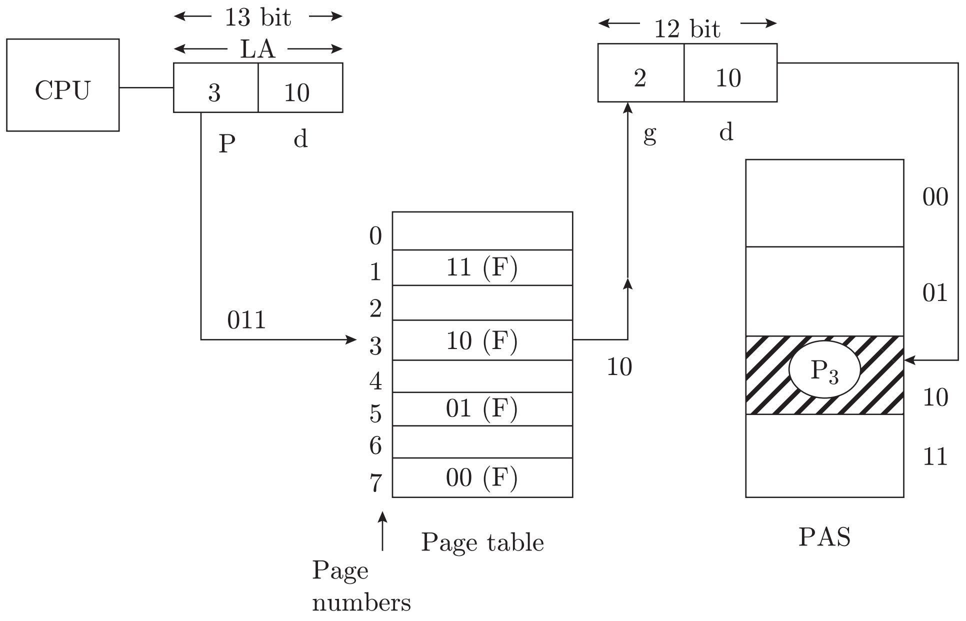
7.8.4 Paging

Paging is one of the memory-management schemes where a computer can store and retrieve data from secondary storage for use in the main memory (Fig. 7.19). This technique maps processor-generated logical address to physical address. As we know,

Logical address > Physical address

There are few terms used in paging:

1. **Logical Address Space (LAS):** Defines the size of logical memory.
2. **Physical Address Space (PAS):** Defines the size of physical memory.



P = Page number bits
f = Frame number bits
d = Page size or frame size

Figure 7.19 A memory paging scheme.

3. **Logical address (LA):** Number of bits to represent logical address space.
4. **Physical address (PA):** Number of bits to represent physical address space.
5. **Page:** Logical memory is divided into equal parts called pages.
6. **Frame:** Physical memory is divided into equal parts called frames.

Assume that processor-generated logical address (LA) is 13-bit long and physical address (PA) is 12-bit long, then

$$\text{Logical address space} = 2^{13} \text{ words} = 2^3 \times 2^{10} \text{ words} = 8\text{K words}$$

$$\text{Physical address space} = 2^{12} \text{ words} = 2^2 \times 2^{10} \text{ words} = 4\text{K words}$$

The logical address is divided into equal size pages. In our case, page size is given as 1K words. So,

$$\text{Number of pages} = \frac{\text{LAS}}{\text{Page size}}$$

$$\text{Number of pages} = \frac{8\text{K}}{1\text{K}} = 8$$

To represent eight pages, we required 3 bits. So, logical address is divided into two parts P (page number or number of bits required to represent the pages of LAS) and d (page size or number of bits required to represent the page size of LAS). This means LA is divided into $P = 3$ and $d = 10$ bits.

The physical address is divided into equal-size frames. Generally, frame size = page size, given as 1K words. So,

$$\text{Number of frames} = \frac{\text{PAS}}{\text{Frame size}}$$

$$\text{Number of pages} = \frac{4\text{K}}{1\text{K}} = 4$$

To represent four frames, we required 2 bits. So, physical address is divided into two parts f (frame number or number of bits required to represent the frames of PAS) and d (frame size or number of bits required to represent the frame size of PAS). This means PA is divided into $f = 2$ and $d = 10$ bits.

Figure Description:

1. Whenever we apply paging, we have to maintain a page table.
2. Number of entries in page table is same as the number of pages in LAS.
3. The page table entries contain frame number (binary format).
4. P is mapped to page table and corresponding entries go to PAS.

Problem 7.5: Consider the following system:

Number of pages = 2K

Page size = 4K words

Physical address = 18 bits

Calculate logical address space and number of frames.

Solution:

1. We know that

$$\text{Number of pages} = \frac{\text{LAS}}{\text{Page size}}$$

So, LAS = Number of pages × Page size

$$\text{LAS} = (2\text{K} \times 4\text{K}) \text{ words} = 2^{11} \times 2^{12} \text{ words} = 2^{23} \text{ words} = 8\text{M words}$$

2. $\text{Number of frames} = \dfrac{\text{PAS}}{\text{Frame size}} = \dfrac{2^{18}}{2^{12}}$

Because frame size = page size, the number of frames is 2^6 frames = 64 frames.

Problem 7.6: Consider a system with LA = 32 bits, PAS = 64 MB and page size is 4 KB. The memory is byte addressable. Page table entry size is 2 bytes. What is the approximate page table size?

Solution:

LA = 32 bits. So, LAS = 2^{32} bytes

$$\text{Number of pages} = \frac{\text{LAS}}{\text{Page size}} = \frac{2^{32}}{2^{12}} = 2^{20}$$

Page table size = Number of entries × Page table entry size ⇒ $2^{20} \times 2$ bytes = 2MB (because number of page table entries = number of pages)

7.8.4.1 Paging Performance with TLB

Translation lookaside buffer is a cache memory used by memory management unit to improve the translation speed of virtual address. The use of virtual memory becomes time consuming without TLB. This buffer keeps the record of all the recently accessed pages.TLB works as shown in Fig. 7.20.

CPU first searches the page in TLB, if the page is found in TLB then physical address is calculated. But if there is TLB miss, then CPU goes to the page table. It checks the page in page table, if found then it calculates the effective address and make the entry in TLB too. But if page is not found in page table, it generates the page fault.

The following points show how the performance is improved by using TLB:

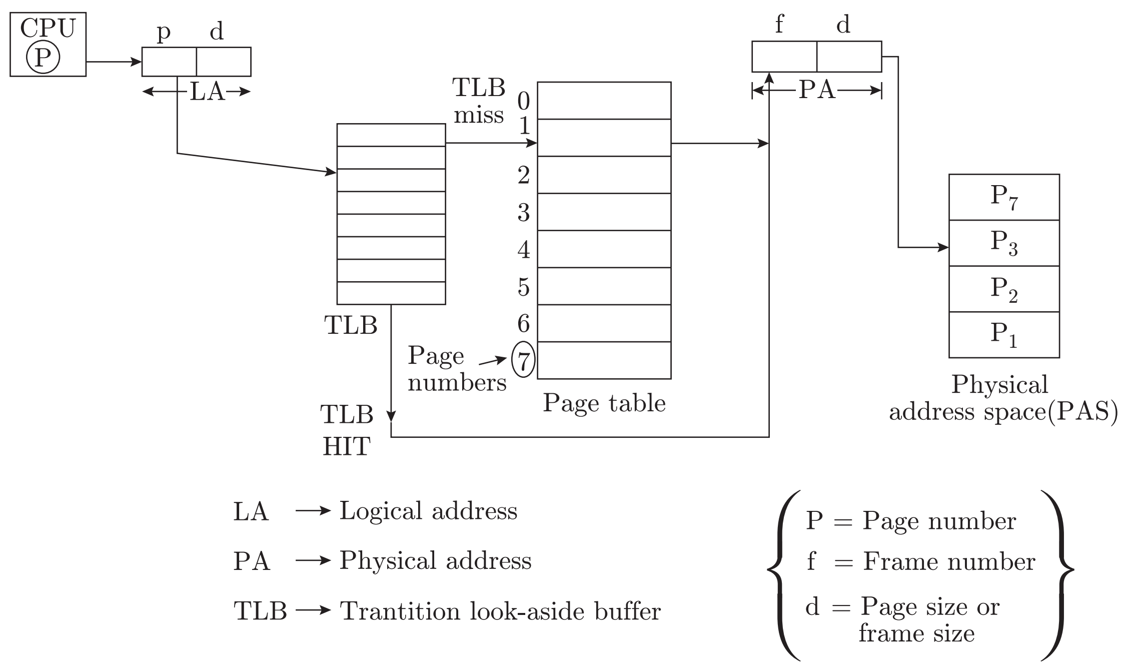


Figure 7.20 | TLB.

1. If page table is kept in the main memory and the main memory access time is M, then formula for effective memory access time for page table is

$$E_{\text{MAT}} = 2M$$

2. Translation look-aside buffer (TLB) is added to improve the performance of page table. TLB contains the frequently referenced page number and corresponding frame number. If the TLB access time is C and TLB hit ratio is X, then effective memory access time for page table is

$$E_{\text{MAT}} = X(C+M)+(1-X)(C+2M)$$

Problem 7.7: Consider a system with TLB access time 20 ns and main memory access time 100 ns. Calculate the effective memory access time if TLB hit ratio is 95%.

Solution:

Given that $C = 20$ ns, $M = 90$ ns, $X = 95\%$. Without TLB effective memory access time

$$E_{\text{MAT}} = 2M = 2\times 100 = 200 \text{ ns}$$

With TLB, effective memory access time

$$\begin{aligned} E_{\text{MAT}} &= X(C+M)+(1-X)(C+2M) \\ &= 0.95(20+100)+(1-0.95)(20+2\times 100) \\ &= 114+11 = 125 \text{ ns} \end{aligned}$$

7.8.5 Multi-Level Paging

Paging technique faces problem of large memory requirement due to the large logical address space. Multi-level paging technique provides the solution to the problem that occurred in paging technique (Fig. 7.21).

Example:

Assume that processor-generated logical address (LA) is 35-bit long and physical address (PA) is 26-bit long, then

$$\begin{aligned} \text{Logical address space} &= 2^{35} \text{ words} \\ &= 2^{5} \times 2^{30} \text{ words} \\ &= 32\text{G words} \end{aligned}$$

$$\begin{aligned} \text{Physical address space} &= 2^{26} \text{ words} \\ &= 2^{6} \times 2^{20} \text{ words} \\ &= 64\text{M words} \end{aligned}$$

The logical address is divided into equal size pages. In our case, page size is given as 4K words. So,

$$\text{Number of pages} = \frac{\text{LAS}}{\text{Page size}} = \frac{32\text{G words}}{4\text{K words}} = 8\text{M pages}$$

To represent 8M pages, we require 23 bits. So, the logical address is divided into two parts P (page number or number of bits required to represent the pages of LAS) and d (page size or number of bits required to represent the page size of LAS). This means LA is divided into P = 23 and d = 12 bits. And P is further divided into two parts P_1 and P_2.

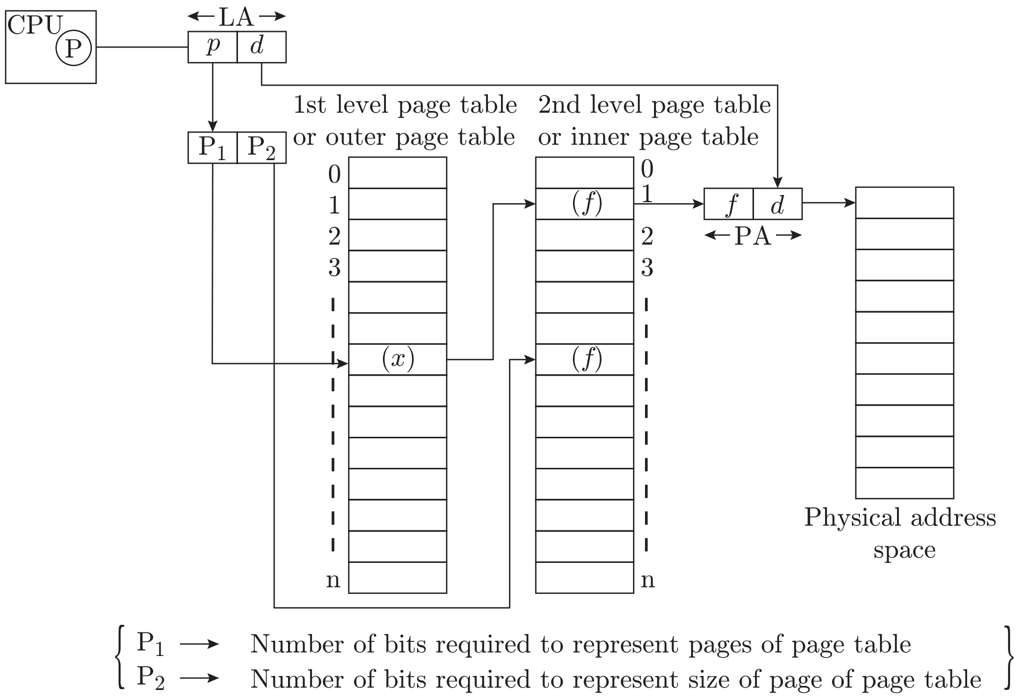

CPU
P
LA
p
d
P1
P2
1st level page table or outer page table
2nd level page table or inner page table
0
1
2
3
n
(x)
(f)
f
d
PA
Physical address space
P1 → Number of bits required to represent pages of page table
P2 → Number of bits required to represent size of page of page table

Figure 7.21 Multi-level paging scheme.

1. P_1 is the page number or number of bits required to represent the pages of page table.
2. P_2 is the page size or number of bits required to represent the page size of page table.

The physical address is divided into equal-size frames. Generally, frame size = page size, given as 4K words. So,

$$\text{Number of frames} = \frac{\text{PAS}}{\text{Frame size}} = \frac{64\text{M words}}{4\text{K words}} = 2^{14}$$

$$= 16\text{K frames}$$

To represent 16K frames, we required 14 bits. So, physical address is divided into two parts f (frame number or number of bits required to represent the frames of PAS) and d (frame size or number of bits required to represent the frame size of PAS). Means PA is divided into $f = 14$ and $d = 12$ bits.

Problems associated with multilevel paging: Memory requirements are low for multilevel paging but address translation time is more due to more number of levels. This can be solved by using Translation lookaside buffer.

7.8.5.1 Inverted Paging

In paging technique, every process has its own page table and because of this there is a problem of huge page table size. To overcome overhead of maintaining the page table for every page, inverted paging is used (Fig. 7.22). In inverted paging, only one table is maintained for all

Problem 7.8: Consider a system with logical address = 34 bits, physical address = 29 bits and page size = 16 KB. The memory is byte addressable and the page table entry size is 8 bytes.

(a) What is the conventional page table size?
(b) What is the inverted page table size?

Solution:

(a) Conventional page table size

$$\text{Number of pages} = \frac{\text{LAS}}{\text{Page size}} = \frac{2^{34}}{2^{14}} = 2^{20}$$

Page table size = Number of pages × Page table entry size = $2^{20} \times 8$ bytes × 8 B = 8 MB

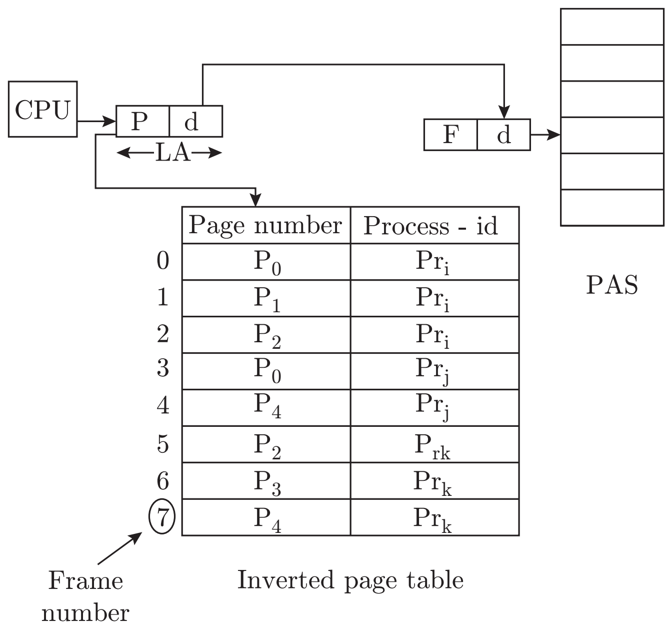
(b) Inverted page table size

$$\text{Number of frames} = \frac{\text{PAS}}{\text{Frame size}} = \frac{2^{29}}{2^{14}} = 2^{15}$$

Page table size = Number of frames × Page table entry size

$$= 2^{15} \times 8 \text{ bytes} \times 8 \text{ B}$$

$$= 256 \text{ KB}$$



	Page number	Process - id
0	P_0	Pr_i
1	P_1	Pr_i
2	P_2	Pr_i
3	P_0	Pr_j
4	P_4	Pr_j
5	P_2	P_{rk}
6	P_3	Pr_k
7	P_4	Pr_k

Figure 7.22 | An inverted paging scheme.

the processes. The number of entries in the inverted page table is the same as number of frames in the PAS.

7.8.6 Segmentation

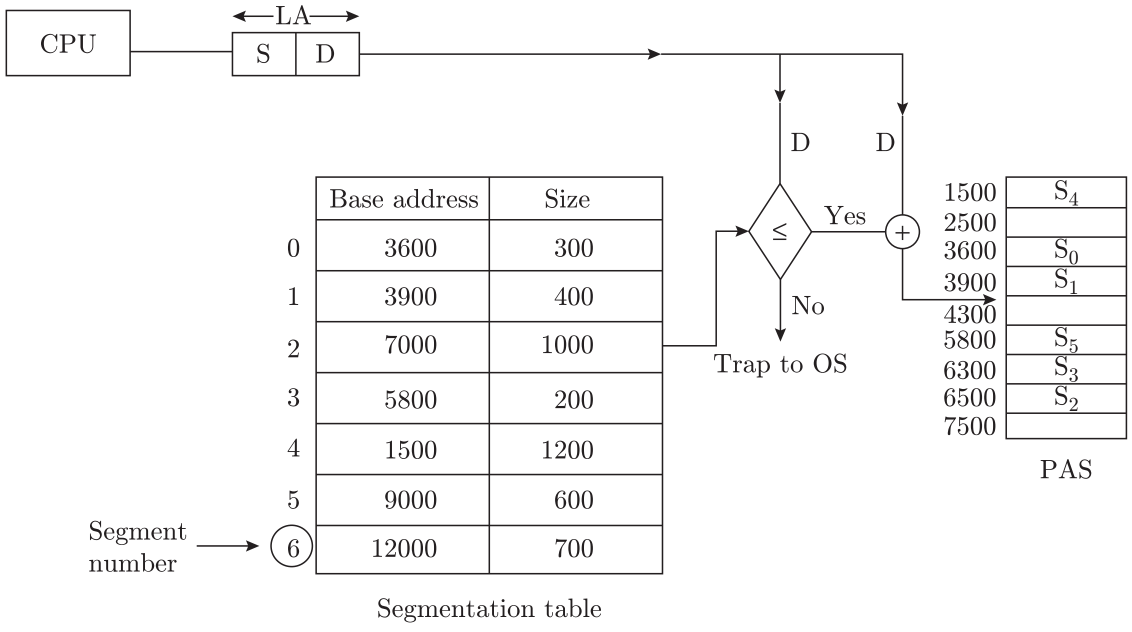
Paging technique does not follow user's view of memory allocation. To achieve user's view of memory allocation, segmentation technique is used (Fig. 7.23). In this technique, logical address space is divided into several segments. Segments may vary in size.

S is the segment number; it is equal to the number of bits required to represent the segment of LAS.

D is the number of bits required to represent the segment size. It is also known as segment offset. So,

Number of segments in segment table
= Number of segments in LAS

Segmentation also suffers from external fragmentation.



	Base address	Size
0	3600	300
1	3900	400
2	7000	1000
3	5800	200
4	1500	1200
5	9000	600
6	12000	700

1500	S_4
2500	
3600	S_0
3900	S_1
4300	
5800	S_5
6300	S_3
6500	S_2
7500	

Figure 7.23 | A memory segmentation scheme.

7.8.7 Virtual Memory

Virtual memory is a memory management technique that is implemented using both hardware and software. It maps memory addresses used by a program, called virtual addresses, into physical addresses in computer memory. Virtual memory gives an illusion to the programmer that programs which are larger in size than actual memory can be executed. Virtual memory can be implemented with demand paging.

7.8.7.1 Demand Paging

Demand paging is a method of virtual memory management (Fig. 7.24). In a system that uses demand paging, the operating system copies a disk page into physical memory only if an attempt is made to access it and that page is not already in memory (i.e., if a page fault occurs).

Figure Description:

Step 1: Processor is looking for page number 4. But page 4 is not available in the memory, that is, page fault. Without page 4 currently executing, program will stop.

Step 2: The signal is send to OS that processor is looking for page 4, which is currently not available in the main memory.

Step 3: OS will search for required page 4 in LAS.

Step 4: OS will replace the page from LAS to PAS by using the page replacement algorithm.

Step 5: OS updates the page table accordingly.

Step 6: The signal is sent to the processor, which says the required page is brought into the memory and then the processor continues the program execution.

7.8.7.2 Effective Memory Access Time

The time taken to service a page fault is called as page fault service time. If the main memory access time is M, page fault service time is S, which is much larger than M ($S \gg M$) and page fault rate is P, then

Effective memory access time,

$$E_{\text{MAT}} = P \times S + (1 - P) \times M$$

Problem 7.9: Consider a system with page fault service time (S) = 100 ns, main memory access time (M) = 20 ns, and page fault rate (P) = 65%. Calculate effective memory access time.

Solution:

$$\begin{aligned} E_{\text{MAT}} &= P \times S + (1 - P) \times M \\ &= 0.65 \times 100 + (1 - 0.65) \times 20 \\ &= 0.65 \times 100 + 0.35 \times 20 \\ &= 65 + 7 = 72 \text{ ns} \end{aligned}$$

Table 7.4 summarizes the different memory management techniques.

7.8.8 Page Replacement Algorithms

Whenever there is a demand of any page which is not in memory then that must be loaded. But if there is no free frame in memory, one page is selected for replacement. Page replacement algorithm decides which page is to be replaced at the time of page fault.

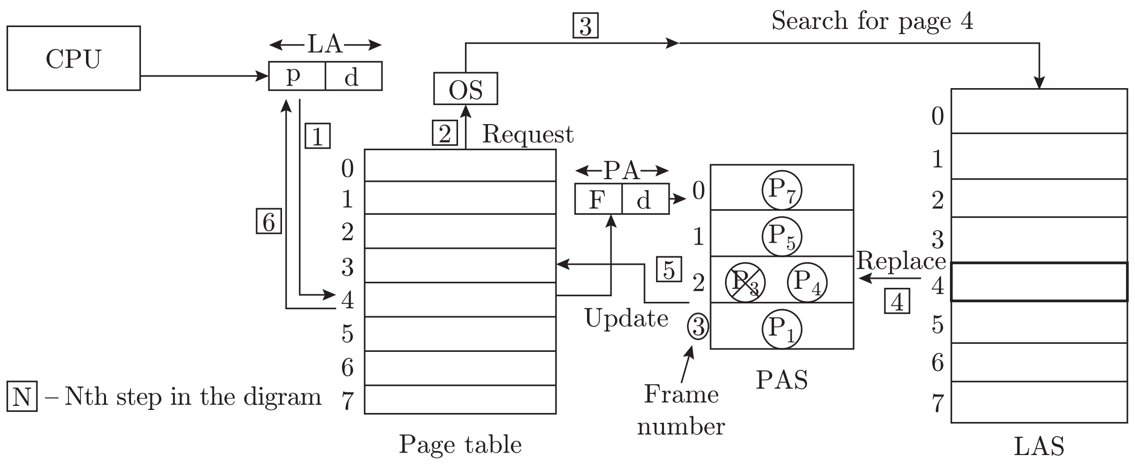


Figure 7.24 A demand paging scheme.

Table 7.4 | Comparison of different memory management techniques

Technique	Pros	Cons
Fixed Partitioning	Simple, No overhead on OS	Internal fragmentation
Dynamic Partitioning	No internal fragmentation, Efficient use of RAM	External fragmentation, Inefficient use of CPU
Paging	No external fragmentation	Very little internal fragmentation
Segmentation	No internal fragmentation, Less overhead as compared to dynamic partitioning	External fragmentation
Virtual Memory Paging	No external fragmentation, High degree of multiprogramming	Overhead on OS
Virtual Memory Segmentation	No internal fragmentation, Higher degree of multiprogramming, Support for protection and sharing	Extra overhead on OS

7.8.8.1 First In, First Out

In this technique, the new page will replace the page which is entered first in the memory frame. This means that page is selected which has been in the main memory for the longest time. Queue data structure is used.

Example 7.8

We have a reference string: 1, 2, 2, 1, 0, 3, 1, 2, 4, 1, 0

Reference String

1	2	2	1	0	3	1	2	4	1	0

1	1	1	1	1	3	3	3	4	4	4
	2	2	2	2	2	1	1	1	1	0
				0	0	0	2	2	2	2
Miss	Miss	Hit	Hit	Miss	Miss	Miss	Miss	Miss	Hit	Miss

Step 1: Suppose we have three empty frames. We insert values in frames 1, 2. Then there is a hit for 2 and 1. Because they are already present in frames, so the next 0 will be inserted in frame.

1
2
0

Step 2: When page 3 is referenced, there is no free frame in memory; 3 will be replaced with the page which is in memory for the longest time. That page is 1.

$\cancel{1}$ 3
2
0

Step 3: Next page referred is 1, which will be replaced with 2 and so on.

$\cancel{3}$ 4
$\cancel{2}\,\cancel{1}$ 0
$\cancel{0}$ 2

Number of hits: 2, 1, 1 = 3

Number of page faults = 1,2,0,3,1,2,4,0 = 8

$$\text{Hit ratio} = \frac{\text{Number of hits}}{\text{Total page references}} = \frac{3}{11}$$

Problems with FIFO technique: FIFO suffers from Belady's anomaly. According to **Belady's anomaly**, sometimes with the increase in number of frames, page fault rate also increases. The performance of the OS drops instead going up (Fig. 7.25).

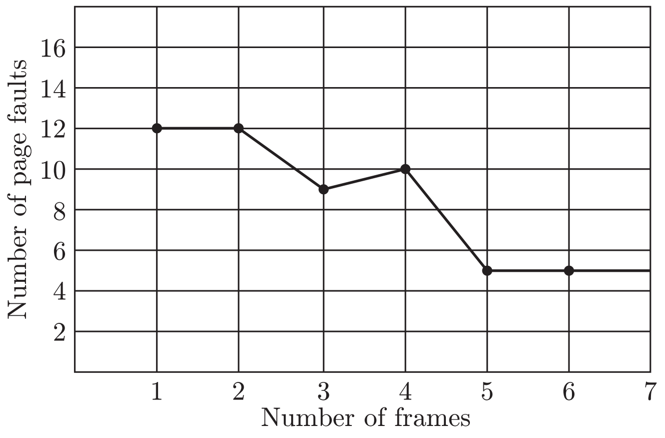


Figure 7.25 | Belady's anomaly.

7.8.8.2 Optimal Replacement Algorithm

In the case of optimal page replacement algorithm, page will be replaced that will not be referenced in future.

Reference String

1	2	2	1	0	3	1	2	4	1	0

1	1	1	1	1	1	1	1	1	1	0
	2	2	2	2	2	2	2	4	4	4
				0	3	3	3	3	3	3
Miss	Miss	Hit	Hit	Miss	Miss	Hit	Hit	Miss	Hit	Miss

Step 1: We have three empty frames. We insert values in frames 1, 2. Then there is a hit for 2 and 1. Because they are already present in frames, so the next 0 will be inserted in the frame.

1
2
0

Step 2: When page 3 is referenced, there is no free frame in memory; 3 will be replaced with the page which will not be used for long. That page is 0.

1
2
$\cancel{0}$ 3

Step 3: Next page referred is 1, 2 will be hit. Now page 4 is demanded. No free frame is there in memory. So it will be replaced by either 2 or 3. Both are not required in future. To decide among the two, FIFO is used. 2 will be replaced with 4.

1
$\cancel{2}$ 4
3

Step 4: Next in reference string is 1. This is already present, so it is a hit. Next is 0, 0 will also use FIFO to break the tie among 1, 4, 3. And 0 will be replaced with 1.

$\cancel{1}$ 0
4
3

- When there is a tie for replacement in optimal page replacement (OPR), FIFO is used to break the tie.

  Number of page faults = 6

  Number of hits = 5

- OPR is the most efficient algorithm. Lowest number of page faults occurs for this algorithm.
- Realization of this algorithm is difficult, because OS needs to know that how long it will take to refer that page again.

7.8.8.3 Least Recently Used

Least recently used (LRU) page replacement algorithm replaces the page which has not been used recently.

Reference String

1	2	2	1	0	3	1	2	4	1	0

1	1	1	1	1	1	1	1	1	1	1
	2	2	2	2	3	3	3	4	4	4
				0	0	0	2	2	2	0
Miss	Miss	Hit	Hit	Miss	Miss	Hit	Miss	Miss	Hit	Miss

Step 1: We have three empty frames. We insert values in frames 1, 2. Then there is a hit for 2 and 1. They are already present in frames. So, next 0 will be inserted in frame.

1
2
0

Step 2: When page 3 is referenced, there is no free frame in memory. So, 3 will be replaced with the page which is not recently used. That page is 2.

1
$\cancel{2}$ 3
0

Step 3: Next page referred is 1, which will be hit. Now page 2 is demanded. No free frame is there in memory. So it will be replaced by 2, due to LRU page.

1
3
$\cancel{0}$ 2

Step 4: Next in reference string is 4. This will be replaced with 1.

1
$\cancel{3}$ 4
2

Step 5: Next in reference string is 1. This will be hit. Next, page 0 will be replaced with 2.

1
4
$\cancel{2}$ 0

Number of hits = 4

Number of page faults = 7

- Counter is maintained after each page reference with the last access time.
- Expensive.

Frame allocation: Each process needs minimum number of pages. The allocation can be done in the following ways.

Fixed allocation:

- **Equal allocation:** Each process gets equal number of frames. For example, there are 80 frames and 4 processes. So, each process will get 20 frames.
- **Proportional allocation:** This is a dynamic way of frame allocation. Allocation is done according to the size of process.

  Allocation for $P_{\text{i}} = \frac{s_i}{S} \times m$

  Here P_{i}: *i*th process

  s_{i}: size of *i*th process

  S: total size

  m: total number of frames

Priority allocation: This method uses priority instead of size to allocate pages.

7.8.8.4 Global versus Local Replacement

As discussed earlier, a page replacement algorithm decides which page is to be replaced at the time of page fault. Now, the questions that arise are:

1. Which page (that we have seen through different page replacement algorithms, above), and
2. From where (to be discussed in the following section).

Global Replacement: The process can replace any page available in the main memory when page fault occurs. All the pages in the system are viewed as common pool. Further, fixed allocation method cannot be implemented using this technique. One of the major problems with this scheme is that each process will spend more time in recovering lost pages, which is called 'thrashing'.

Local Replacement: When the page fault occurs, local replacement algorithm can replace the pages from the same process. Other processes will not suffer by this scheme.

1. Local page replacement does not remove thrashing, but it reduces it to some extent.
2. Working set model is solution of thrashing.

7.8.8.5 Thrashing

A program that causes page faults more frequently is said to be thrashing, which means that the system more busy in dealing with page faults instead of doing some constructive work. This leads to the decrease in throughput of the system.

Further, it has been observed that there is a degree of multiprogramming, which is optimal for any system's performance, because if we increase the degree of multiprogramming it might result in thrashing (Fig. 7.26).

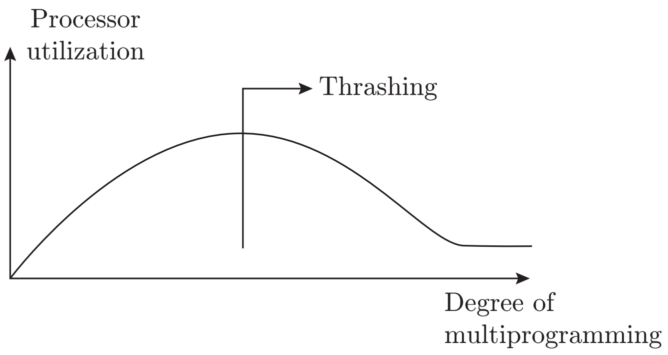


Figure 7.26 | Thrashing in multiprogramming

There are different remedies to overcome thrashing, one of them is discussed below:

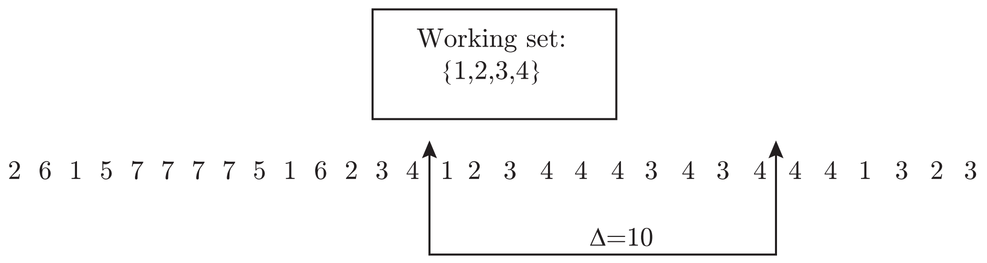
Working set model: According to Peter Denning, working set is a collection of pages that are being used by some process during a finite time interval. This gives an idea about approximate set of pages the process will require in future.

$$\Delta = \text{Working set window}$$

Example:

Window size = 10

As shown in the figure below, active pages are in set.



7.9 FILE SYSTEM

7.9.1 Directory Structures

The directory structures are used to manage and organize data. This can be done in the following ways:

1. The disks can be split into one or more partitions. These partitions provide separate areas within one disk and are treated as a separate storage devices.
2. Each partition contains information about files. The device directory contains information such as name, location, size and type.

7.9.1.1 Single-Level Directory

It is the simplest directory structure (Fig. 7.27), in which there is one directory which contains all the files. Major limitation lies in the number of files (memory size) a directory can hold and there is more than one user that needs to access the files.

7.9.1.2 Two-Level Directory

To overcome the above problems, a separate directory can be created for each user, known as – User File Directory (UFD). The major limitation is the sharing of files among the users. This scheme is represented in Fig. 7.28.

7.9.1.3 Tree-Structured Directories

Directory structure is extended to a tree of arbitrary height. In this, users can create their own subdirectories and manage their own files. There is a root directory too. Such a type of file system is more popular in present day OS. A path name is the path from the root, through all the subdirectories, to a specified file. There are two types of path names:

1. **Absolute path name:** It begins at the root and follows a path down to the specified file, giving directory names on the path. For example, root/spell/mail/sys/units.
2. **Relative path name:** It defines a path from the current directory. Considering the same example as above, let us suppose that we are working in a current directory named root/spell/mail, then the relative path can be sys/units, instead of specifying the full path name.

7.9.1.4 Acyclic Graph Directories

An acyclic graph is a graph with no cycles and is a natural generalization of the tree-structured directory scheme (Fig. 7.29). A tree structure prohibits the sharing of files or directories, whereas an acyclic graph allows directories to have shared subdirectories and file, which means the same file or subdirectory may be in two different directories. With a shared file, only one actual file exists, so any changes made by one person are immediately visible to the other.

7.9.2 Allocation Methods

The direct access nature of disks allows us flexibility in the implementation of files and usually many files are stored on the same disk. The main problem is how to allocate space to these files so that disk space is utilized effectively and files can be accessed quickly. Three

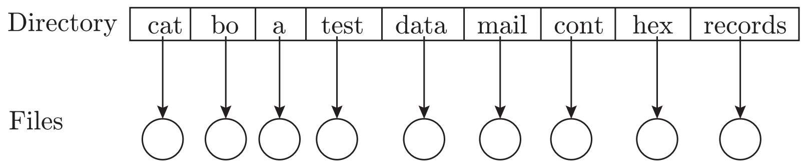


Figure 7.27 A single-level directory.

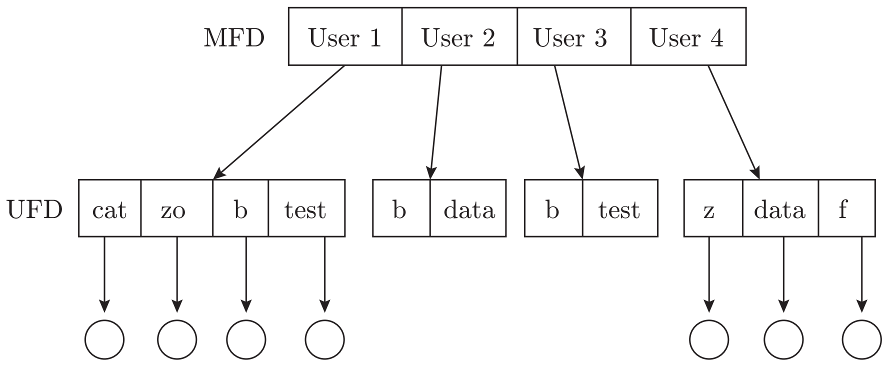


Figure 7.28 Two-level directory structures.

Note: User 2 and User 4 have a file named data, thus name-collision problem is solved.

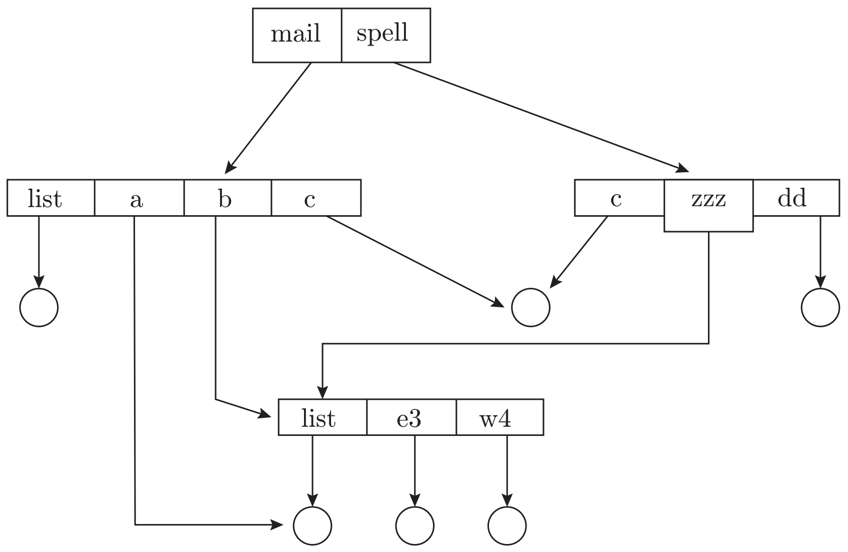


Figure 7.29 Acyclic graph directory structures (the *sub* directory list is shared by *zzz* and *b*).

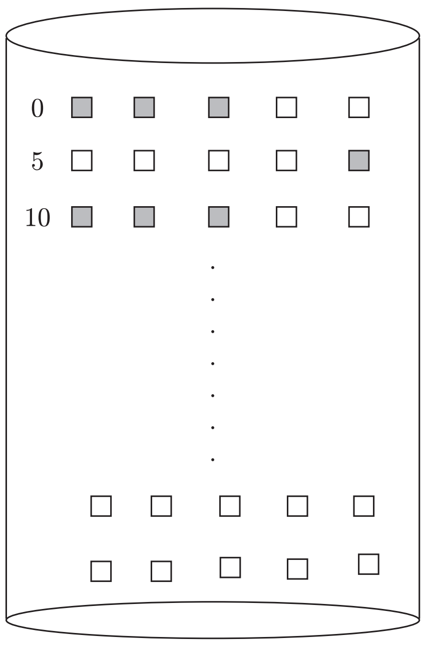
widely used methods of allocating disk space are discussed in the following text.

7.9.2.1 Contiguous Allocation

In contiguous allocation method, each file occupies a set of contiguous blocks on the disk. All the blocks are continuously kept together. There is an index that specifies the starting address and the length (size) indicating the blocks (Fig. 7.30). Different allocation strategies are first-fit, best-fit or worst-fit (already discussed earlier). This technique suffers from internal and external fragmentation. It supports sequential and random access.

7.9.2.2 Linked Allocation

In linked allocation, each data block contains the address of the next block in the file (Fig. 7.31). In contrast to contiguous allocation, the disk blocks are scattered on disk. The directory entry contains an index, which points to the starting block as well as the ending block. Each block in turn contains an address of the next block, similarly, all the blocks are linked together. The last block does not have a



File	Start	Length
Cnt	0	3
List	9	4

Figure 7.30 Contiguous allocation of disk space.

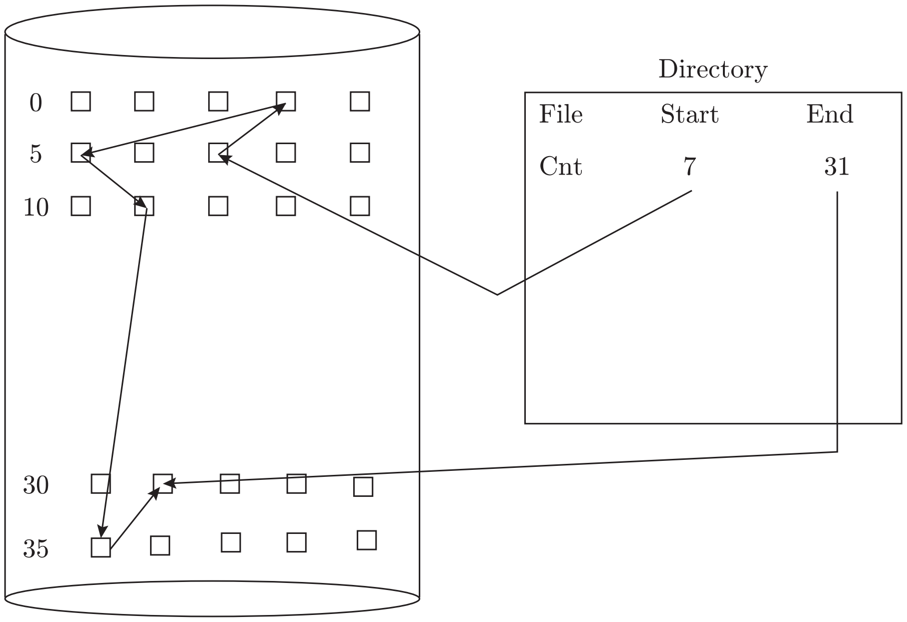


Figure 7.31 | Linked allocation of disk space.

next block address (just as linked list). The major advantage being, no external fragmentation with a limitation of slow access (because we have to traverse all the blocks in the list before the required block). Another major drawback is the breaking of the link (similar to dangling references).

7.9.2.3 Indexed Allocation

There is an index block that contains the address of every block that contains the information associated with that file (Fig. 7.32). This approach is better as provides dynamic access without any external fragmentation and there is very less loss of information in case of breaking of links. Let us take an example of a loss of pointer, in this case only information about that block will be lost whereas in linked allocation information stored in the rest of blocks will also be lost.

It is hotly debated how large the index block should be. Every file must have an index block, so we want the index block to be as small as possible. If the index block is too small, however, it will not be able to hold enough pointers for a large file, and a mechanism will have to be

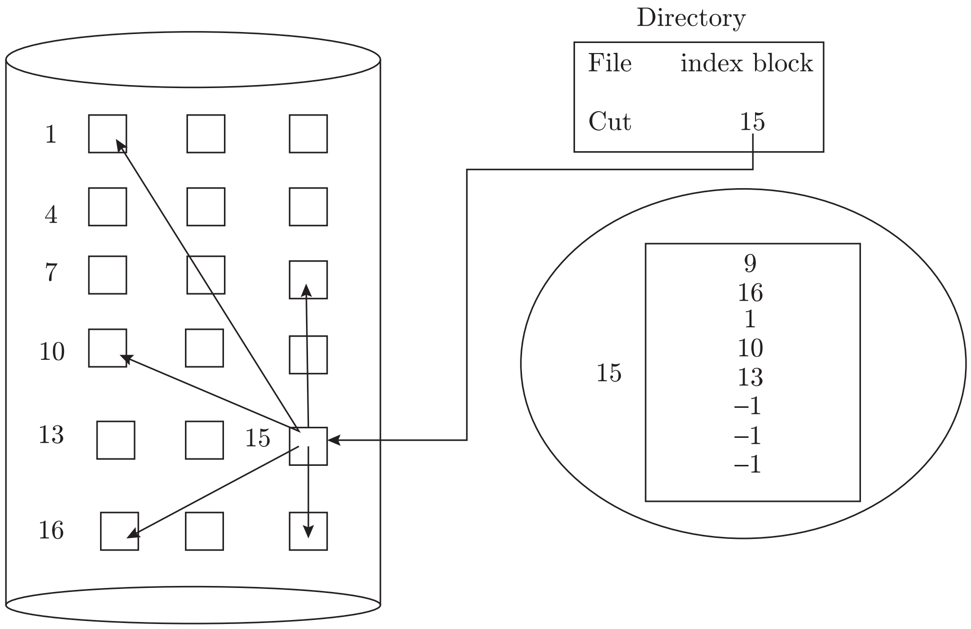


Figure 7.32 | Indexed allocation of disk space (-1 represents not used).

available to deal with this issue. The following are the possible solutions:

1. **Linked scheme:** An index block is normally of one disk space. Thus, it can be read and written directly by itself.
2. **Multilevel index:** The first index block contains the set of pointers to secondary index blocks, which in turn contain pointers to the actual data blocks.
3. **Combined scheme:** It is used in the UNIX File System (UFS) i-nodes (see Fig. 7.33, which represents the UFS i-node scheme), where first few data block pointers of the index block are stored directly in the i-node and then single, double and triple indirect pointers provide access to more data blocks as needed.

   Total size of file system

$$= \left\{\text{Number of direct DBA} + \left(\frac{\text{DBA size}}{\text{DBA}}\right) + \left(\frac{\text{DBA size}}{\text{DBA}}\right)^2 + \left(\frac{\text{DBA size}}{\text{DBA}}\right)^3 + \cdots\right\} \times \text{DB size}$$

   where DBA is disk block address.

- Number of disk block possible in one disk block = $\left(\dfrac{\text{DBA size}}{\text{DBA}}\right)$
- If asked for maximum file size, then calculate for highest power only such as Problem 7.10.

Problem 7.10: Consider the UNIX i-node which uses 12 direct DBAs, 1 single indirect, 1 double indirect, 1 triple indirect. The disk block address requires 32 bits and the disk block size is 1 KB. What is the maximum file size?

(a) 8 GB (b) 16 GB
(c) 32 GB (d) 64 GB

Solution:

$$\text{Maximum file size} = \left(\frac{\text{DBA size}}{\text{DBA}}\right)^3 \times \text{DB size}$$

$$= \left(\frac{1\text{ KB}}{4\text{ bytes}}\right)^3 \times 1\text{ KB}$$

$$= \left(\frac{2^{10}}{2^2}\right)^3 \times 2^{10}\ \text{bytes}$$

$$= 16\text{ GB}$$

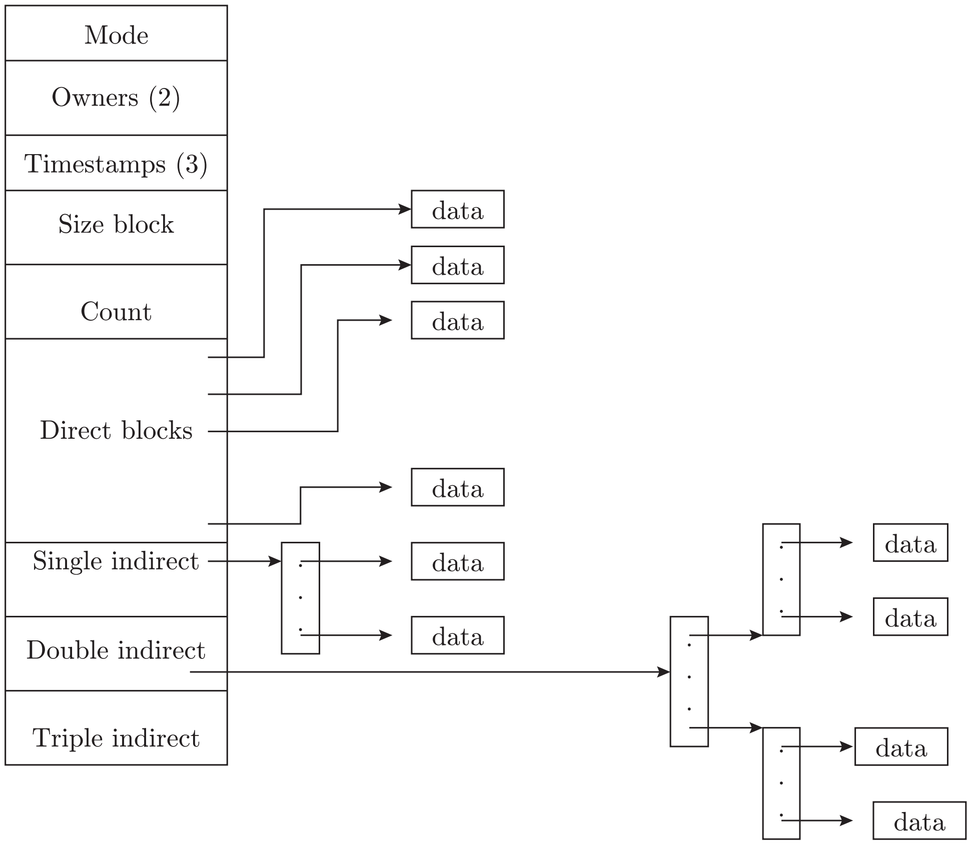


Figure 7.33 The UNIX i-node combined scheme.

7.10 I/O SYSTEMS

There are different devices that are connected with a computer. These devices are broadly classified as:

1. **Machine readable:** Most of the electronic equipments such as hard disk, floppy disk, CD, USB, bar code reader, actuators, etc.
2. **Human readable:** Devices such as printer, keyboard and Video Display Unit (VDU), usually called monitor screen (but is different from monitor, as used in OS). Such devices transfer the data at a slower rate as compared to the data transferred between the memory and CPU. To overcome such differences in the rate of transfer, the concept of SPOOLing (Simultaneous Peripheral Operations OnLine) is used.
3. **Communication:** Network Interface Cards (NIC) are used for network communication or modems, etc.

There are different techniques used for I/O, such as Programmed I/O, Interrupt driven I/O and DMA I/O. These are covered in detail in Chapter 2.

7.10.1 Disk Structure

Due to advent of technology, the gap in the access speeds of a processor (as well as primary memory, i.e., RAM) and disks (the secondary memory, i.e., hard disks) have increased. In order to provide a faster access to user, it is equally important for the OS to manage the disks efficiently by making use of suitable algorithms. A simple disk structure is shown in Fig. 7.34.

Note: Number of cylinders per track = Number of tracks per surface

Problem 7.11: Consider a disk which has 16 plotters, each plotter is divided into two surfaces. Each surface has 1K tracks and each track has 512 sectors. Each sector can store 2 KB data.

(a) Find the capacity of the disk.
(b) How many bits are required to identify a sector?

Solution:

(a) Capacity of disk = (Plotters × Surface × Tracks × Sector × Sector Size) bytes

$= (16 \times 2 \times 1024 \times 512 \times 2048)$ bytes

$= (32 \times 2^{10} \times 2^{9} \times 2^{11})$ bytes

$= 32 \times 2^{3}$ bytes

$= 32$ GB

(b) Number of bits required to identify a sector

= (Bits for plotter + bits for surface + bits for tracks + bits for sectors)

$= (4 + 1 + 10 + 9)$ bits

$= 24$ bits

7.10.1.1 Time Related to Disk

1. **Seek time:** The amount of time required to move the read/write head from its current position to desired track.
2. **Rotational latency:** The amount of time to rotate the track when the read/write head comes to desired sector position. In simple disk, rotational latency is the time to rotate 1/2 disk to the access.
3. **Transfer time:** The amount of time taken to transfer the required data is called as transfer time.

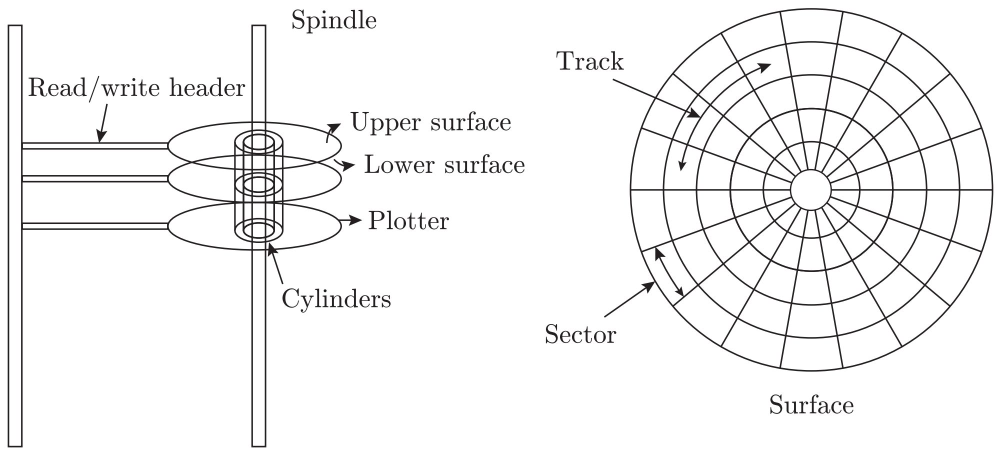


Figure 7.34 The disk structure.

Transfer time depends on the total size of the track and the rotational rate of the disk.

4. **Transfer Rate:** The number of bytes transferred in one unit time is called as the transfer rate of the disk.

7.10.1.2 Disk Interleaving

When the disk rotation speed is higher, then read/write head skips sector 2 after reading sector 1 because it does not get enough time to read the next sector. This problem is solved by the disk interleaving technique (Fig. 7.35):

1. **Single interleaving:** There is a gap between two consecutive sectors.
2. **Double interleaving:** There are two gaps between two consecutive sectors.

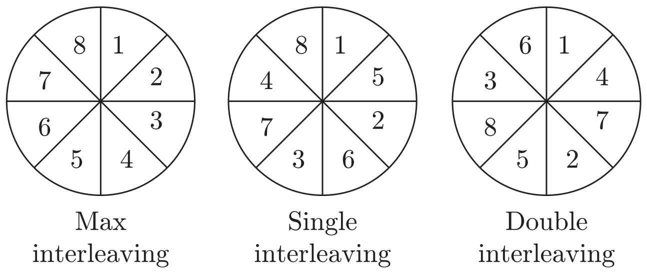


Figure 7.35 Disk interleaving.

Note:

1. In single interleaving technique, to read a track, two rotations are required.
2. In double interleaving technique, to read a track, 2.75 rotations are required.

Problem 7.12: Consider a disk which has average seek time of 32 ns and rotational rate of 360 rpm (round per minute). Each track of the disk has 512 sectors, each of size 512 bytes.

(a) What is the time taken to read four continuous sectors?
(b) What is the data transfer rate?

Solution:

(a) Time to read four continuous sectors = (Seek time + rotational latency + transfer time)

I. Seek time = 32 ns = 32×10^{-9} s

II. Rotational latency
$\Rightarrow$360 rotations took 60 s
1 rotation $\rightarrow$ (60/360) s
1/2 rotation $\rightarrow$ (60/360) $\times$ (1/2) $\Rightarrow$ 0.083 s

III. Transfer time
$\Rightarrow$1 Track can be read in one rotation, and 1 track has 512 sectors each having 512 bytes.

So, (512 $\times$ 512) bytes data can be read in one rotation time (1/6) s.

256 KB data read time = (1/6) s

$$1 \text{ KB data read time} = \left(\frac{1}{6\times 256}\right)\text{s}$$

$$2 \text{ KB data read time} = 2\times\left(\frac{1}{6\times 256}\right)\text{s}$$

(1 sector size= 512 byte, so four sector = 2KB) = 0.0013 s

Time to read four continuous sectors = $(32 \times 10^{-9} + 0.083 + 0.0013)$ s = 0.0843 s

(b) Transfer rate = Number of bytes transferred in one unit time (1 s)

1/6 s $\rightarrow$ 256 KB

1 s $\rightarrow$ (256 $\times$ 6) KB = 1536 Kbps

Problem 7.13: Consider an interleaving disk where the track is divided into eight sectors, each of size 2K. Seek time is 30 ms and rotation rate of disk is 3600 rpm. If half rotation is required to place the read/write head at the start of the sector, then

(a) How much time is required reading all eight sectors of the track using double-interleaving disk?
(b) What is the time difference to read all eight sectors if it is a non-interleaving disk?
(c) What is the data transfer rate using double-interleaving disk?

Solution:

(a) Time to read eight continuous sectors = (Seek time + rotational latency + transfer time)

I. Seek time = 30 ms
II. Rotational latency = (1/2) rotation
3600 rotations took 60 s

1 rotation → (60/3600) = 16.66 ms

1/2 rotation → (16.66) × (1/2) = 0.0083 = 8.33 ms

III. Transfer time
We have to read all of the eight sectors mean one whole track. 1 Track can be read in 2.75 rotations (double interleaving).

So, transfer time of eight sectors = Transfer time of 1 track = 2.75 Rotation time

Time to read eight continuous sectors = (Seek time + rotational latency + transfer time)
= (Seek time + (1/2) rotation time + 2.75 rotation time)
= (Seek time + 3.25 rotation time)
=–(30 + 3.25 × (16.66)} = 84.14 ms

(b) Time to read eight continuous sectors = (Seek time + rotational latency + transfer time)
I. Seek time = 30 ms
II. Rotational latency ⇒ 8.33 ms
III. Transfer time
We have to read all of the eight sectors means one whole track.

1 Track can be read in 1 rotation (non-interleaving).

So, transfer time of eight sectors = Transfer time of 1 track = 1 Rotation time

Time to read eight continuous sectors = (Seek time + rotational latency + transfer time)
= (Seek time + (1/2) rotation time + 1 rotation time)
= (Seek time + 1.5 rotation time)
= {(30 + 1.5 × (16.66)} = 55 ms

Time difference between double interleaving and non-interleaving = (84.14 − 55)
= 29.14 ms

(c) Transfer rate = Number of bytes transferred in one unit time (1 s)

2.75 Rotation time → 16 KB

60 s = 3600 rotations

1 s = 3600/60 = 60 rotations

2.75 rotation = 16 KB data

1 rotation = (16/2.75) KB

60 rotations = {(16/2.75) × 60} KB ≈ 349 KB

So, transfer rate ≈ 349 Kbps

7.10.2 Disk Scheduling Algorithms

One of the responsibilities of the operating system is to use the hardware efficiently, which means that we should have fast disk access time and a broader disk bandwidth. 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, such as:

1. Whether this operation is input or output?
2. What the disk address for the transfer is?
3. What the memory address for the transfer is?
4. 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 driver or controller is busy, any new requests for service will be placed on the queue of pending requests for that device. We require disk-scheduling algorithms to service these pending requests.

Problem 7.14: Consider, for example, a disk queue with requests for I/O to blocks on cylinders

88, 83, 7, 13, 90, 79, 11, 63

The disk head is initially at cylinder 53.

Solution:

The head will first move from 53 to 88 (being the first request), then to 83, 7, 13, 90, 79, 11, and finally to 63, for a total head movement of 339 cylinders. This schedule is shown in Fig. 7.36.

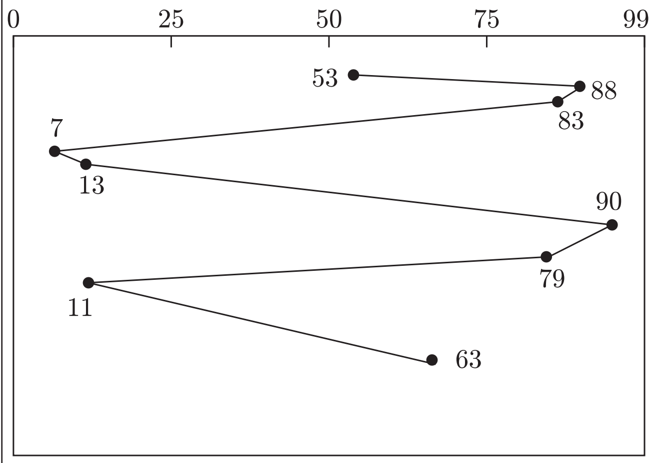


Figure 7.36 FCFS disk-scheduling algorithm.

Total head movement = (53~88) + (88~83) + (83~7) + (13~7) + (99~13) + (79~90) + (11~79) + (11~63)
= 35 + 5 + 76 + 6 + 86 + 11 + 68 + 52
= 339 cylinders

7.10.2.1 FCFS Scheduling

The simplest form of the 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.

7.10.2.2 SSTF Scheduling

A variation of shortest job first (SJF), applied in CPU scheduling. It serves the request that has a minimum seek time from the current head position. To calculate the minimum seek time, we look for the minimum number of cylinders to be traversed. The major limitation is — starvation, as encountered in SJF algorithm.

Problem 7.15: Consider, the same problem, a disk queue with requests for I/O to blocks on cylinders

88, 83, 7, 13, 90, 79, 11, 63

The disk head is initially at cylinder 53.

Solution:

The head is positioned at cylinder 53, the minimum distance it has to move is 10 when it moves to cylinder 63, which is minimum, then from 63 the next access will be 79, then 83, to 88, then to 90, to 13, 11 and finally 7. This schedule is shown in Fig. 7.37.

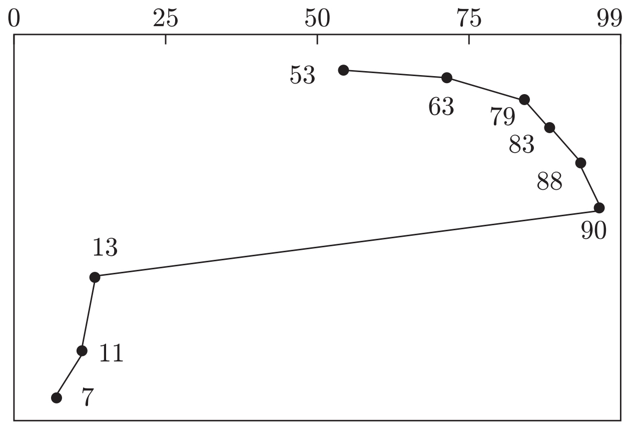


Figure 7.37 SSTF disk-scheduling algorithm.

Total head movement = (53~63) + (63~79) + (79~83) + (83~88) + (88~90) + (90~13) + (13~11) + (11~7)

= 10 + 4 + 4 + 5 + 2 + 77 + 2 + 4 = 108 cylinders

7.10.2.3 SCAN Scheduling

Also, called the elevator algorithm, here the disk arm begins serving the requests at one end and moves to the other end (it does not come back, as in earlier algorithms) and then returns back serving the requests, similar to an elevator. This technique eventually results in reduced variance as compared to SSTF.

Problem 7.16: Consider, the same previous problem, a disk queue with requests for I/O to block on cylinders

88, 83, 7, 13, 90, 79, 11, 63

The disk head is initially at cylinder 53 and the disk arm is moving towards 0.

Solution:

The head is positioned at cylinder 53 and the disk arm is moving towards 0. So, the cylinders accessed will be 12, 11 and 7, the disk arm will touch 0 and then will read 63, then 79, 83, 88 and finally 90. This schedule is shown in Fig. 7.38.

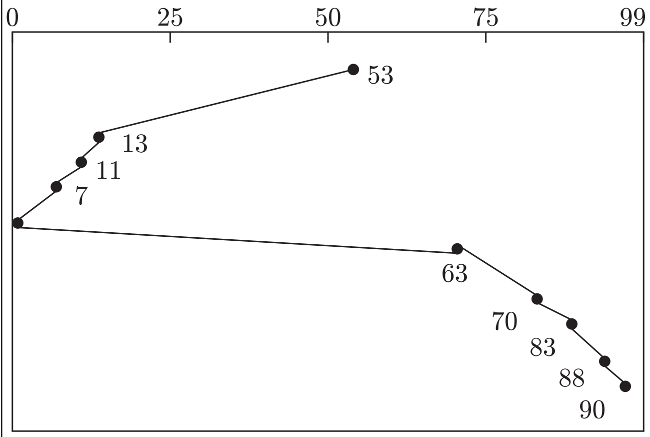


Figure 7.38 SCANdisk-scheduling algorithm.

Total head movement = (53~13) + (13~11) + (11~7) + (7~0) + (0~63) + (63~79) + (79~83) + (83~88) + (88~90)

= 40 + 2 + 4 + 7 + 63 + 16 + 4 + 5 + 2
= 143 cylinders

It should take into account the head movement from cylinder 7 to 0 and 0 to 63 also.

7.10.2.4 C-SCAN Scheduling

This algorithm treats the cylinders as a circular list, hence provides a uniform wait time as compared to the SCAN algorithm discussed above. The major difference is that the requests are served in any one direction of the motion of the head, not in both the directions as in SCAN. This can be compared by looking at the following numerical problem.

Problem 7.17: Consider, the same problem, a disk queue with requests for I/O to blocks on cylinders

88, 83, 7, 13, 90, 79, 11, 63

The disk head is initially at cylinder 53 and the disk arm is moving towards 0.

Solution:

The head is positioned at cylinder 53 and the disk arm is moving towards 0. So, the cylinders accessed will be 13,11 and 7, the disk arm will touch 0 and then will return to the end (in this case) and start reading 90, 88, 83, 79, and finally read 63. This schedule is shown in Fig. 7.39.

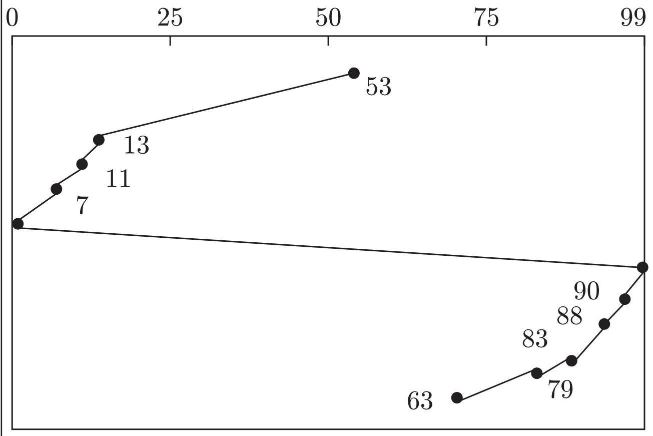


Figure 7.39 C-SCAN disk-scheduling algorithm.

Total head movement = (53~13) + (13~11) + (11~7) + (7~0) + (0~99) + (99~90) + (90~88) + (88~83) + (83~79) + (79~63)

= 40 + 2 + 4 + 7 + 99 + 9 + 2 + 5 + 4 + 16
= 188 cylinders

7.10.2.5 LOOK and C-LOOK Scheduling

LOOK and C-LOOK are the variants of SCAN and C-SCAN algorithm, respectively. Here the head does not move till the end of the cylinder, it only moves to the farthest cylinder which requires to be served. After serving the farthest request, it changes the direction (in case of LOOK) or jumps to another end (in case of C-LOOK). The same is illustrated below.

Problem 7.18: Consider, the same problem, a disk queue with requests for I/O to blocks on cylinders

88, 83, 7, 13, 90, 79, 11, 63

The disk head is initially at cylinder 53 and the disk arm is moving towards 0.

Solution:

The head is positioned at cylinder 53 and the disk arm is moving towards 0. So, the cylinders accessed will be 13, 11 and 7, now the disk arm will not touch 0 (as in SCAN algorithm) but will read 63, then 79, 83, 88 and finally 90. This schedule is shown in Fig. 7.40.

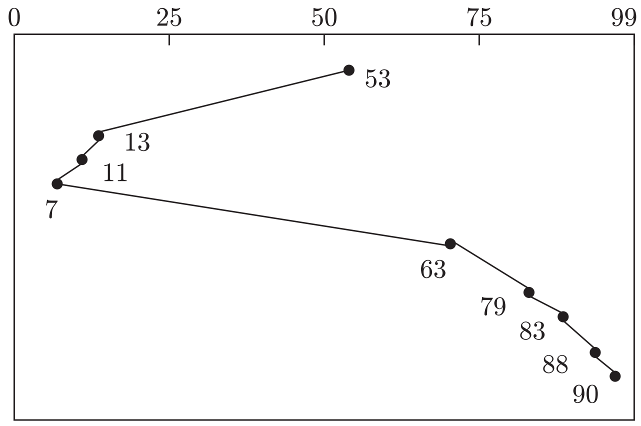


Figure 7.40 LOOK disk-scheduling algorithm.

Total head movement = (53∼13) + (13∼11) + (11∼7) + (7∼63) + (63∼79) + (79∼83) + (83∼88) + (88∼90)

= 40 + 2 + 4 + 56 + 16 + 4 + 5 + 2 = 129 cylinder

Problem 7.19: Consider, the same example, a disk queue with requests for I/O to blocks on cylinders

88, 83, 7, 13, 90, 79, 11, 63

The disk head is initially at cylinder 53 and the disk arm is moving towards 0. Simulate the C-LOOK disk-scheduling algorithm.

Solution:

The head is positioned at cylinder 53 and the disk arm is moving towards 0. So, the cylinders accessed will be 13,11 and 7, the disk arm will not touch 0 but will return to 90, then read 88, 83, 79 and finally read 63. This schedule is shown in Fig. 7.41.

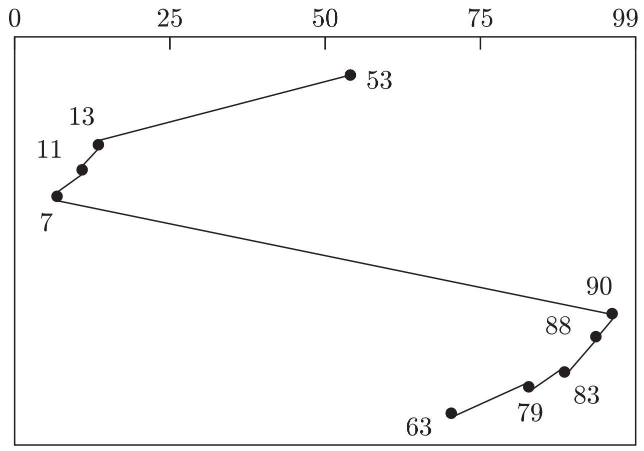


Figure 7.41 C-LOOK disk-scheduling algorithm.

Total head movement = (53∼13) + (13∼11) + (11∼7) + (7∼90) + (90∼88) + (88∼83) + (83∼79) + (79∼63)

= 40 + 2 + 4 + 83 + 2 + 5 + 4 + 16 = 156 cylinder

7.10.2.6 Selection of a Disk-Scheduling Algorithm

Given so many disk-scheduling algorithms, how do we choose the best one? SSTF is common and has a natural appeal because it increases performance over FCFS. SCAN and C-SCAN perform better for system that place a heavy load on the disk, because they are less likely to have starvation problem. Therefore, it is dependent on the type of application we want to use.

7.11 PROTECTION AND SECURITY

As huge information is stored in computer systems, the need to protect it is equally important. Usually, security and protection are used interchangeably.

7.11.1 Security

There are three important aspects, known as CIA triad, which are basic to providing security. These are:

1. **Confidentiality:** It involves data confidentiality as well as user privacy.
2. **Integrity:** It involves two terms—data integrity (which means that data is handled in an authorized manner) and system integrity (which means

that the system is free from unauthorised access or manipulation).

3. **Availability:** It means the timely access to user (for which he has the rights).

7.11.2 Security Environment

Security has many facets. Three of the most important ones are the nature of the threats, the nature of intruders and accidental data loss.

There are two different kinds of attack. A *'Passive Attack'* does not affect the system resources but observes the various acts of the system (such as – trying to know the password by looking over the shoulders of a user), whereas in an *'Active Attack'* the system resources are affected (such as – an unauthorised person deleting a file).

7.11.2.1 Threats

From a security perspective, computer systems have three general goals, with corresponding threats to them (see Table 7.5):

Table 7.5 Goal and threats.

Goal	**Threat**
Data confidentiality	Exposure of data
Data integrity	Tempering with data
System availability	Denial of Service (DOS)

7.11.2.2 Intruders

In the securityperspective, intruders are defined as unauthorized people, who are entering into the system they do not have business with. Intruders are also known as '*adversaries*', and are capable of deploying active attacks as well as passive attacks.

7.11.2.3 Accidental Data Loss

Malicious intruders cause threats to the system. Hence, valuable data can be lost by accident. The common causes of accidental data loss are as follows:

1. **Hardware or software errors:** CPU malfunctions, unreadable disks or tapes, program bugs, etc.
2. **Human errors:** Incorrect data entry, wrong tape or disk mounted, wrong program run, lost disk or tape, etc.
3. **Natural disaster:** Fires, floods, earthquakes, wars, riots, or rats gnawing tapes or floppy disks, etc.

7.11.3 Cryptography

First we will discuss about cryptography, in brief, to understand more about security.

The aim of cryptography is to encrypt plaintext (message) into cipher text in such a way that only the authorized person or user can decrypt it back to plaintext.

In symmetric cryptography (or private key cryptography), one makes use of a single key. This key is used for encryption at the sender side and the same key is used at the receiver side to decrypt the message.

Whereas, in asymmetric cryptography (or public key cryptography), two keys are used in which one is private and the other one is public. The public key is used to encrypt the message, whereas the private key, which is only known to the receiver, is used for decryption.

The encryption process is illustrated in the Fig. 7.42 below:

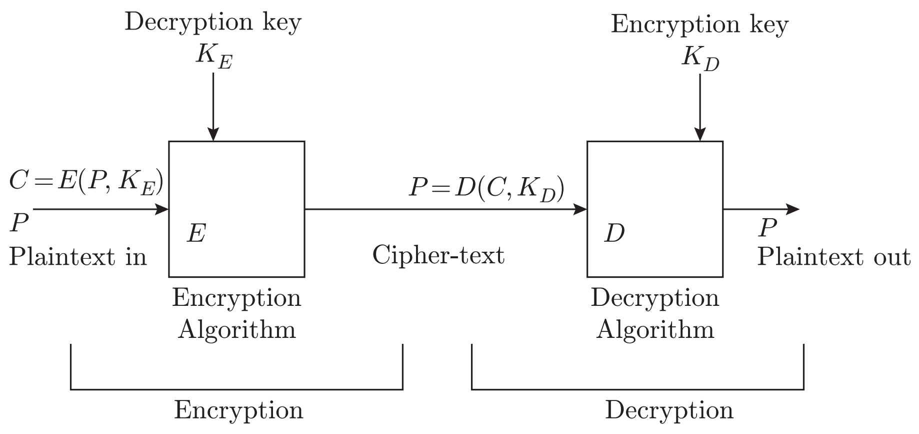


Figure 7.42 Mechanism of cryptography.

7.11.4 Attacks from within the System

When an adversary enters into a system, he or she can start doing damage. The damage is of the following types:

1. **Trojan horses:** Trojan horse is a program, which contains a code hidden within a block of code, used to modify/delete/encrypt the user's data (functioning is similar to the Trojan horse, we read in history). The Trojan horse cannot be differentiated from a normal code of program by reading the source code of the program housing the Trojan horse.
2. **Login spoofing:** Related to Trojan horse, it is a technique to obtain a user's password. The user is provided with the same lookalike page, as his login and he is unable to differentiate between the actual page and this page; so he enters his login password and the vital information does not remain private.
3. **Logic bombs:** Similar to a bomb, this code explodes (causes destruction) when certain condition(s) are met like – a particular day of time or a particular date or when the user is running a particular file. Such a code is embedded in a legitimate program, as in a Trojan horse. These codes can modify or delete a file or can even cause a system to halt or do any other damage.
4. **Trap doors:** These are also small programs embedded in the programs to quickly gain access at a later time. These are also known as '*backdoors'*. It is very difficult to block these backdoors by an OS.
5. **Buffer overflow:** It is a program written to block memory or buffer. It allows adversary to modify portions of the target process address space.

7.11.5 Attacks from Outside the System

In the previous sections, we have discussed about the threats largely caused by users already (logged) in the system; however, for machines connected to the network, there are external threats too.

7.11.5.1 Virus Damage Scenarios

A virus is a program that can do anything a program can do. It replicates itself, infects user's data or files, stops the operation of the system or functioning of the system. It gets its name because of the similarities in its functionality to a biological virus. The common acronym for virus is Vital Information and Resources Under Siege. Nowadays, there are certain programs known as *'Kit'*, capable of generating new viruses automatically. Some of the well-known anti-virus softwares are: Symantec, McAfee, AVG, Kaspersky, etc.

Viruses can be divided into different classes based upon the following characteristics:

1. Environment
2. Operating system (OS)
3. Destructive capabilities

(Not to forget: there also exist other 'harmful' programs or so called '**malware**', such as Trojan horses).

1. **Environment:** Viruses can be described as:
   - **File viruses:** Most common virus that infects the executable (*.exe) files.
   - **Boot viruses:** The virus infects the boot sector program, the contents are replaced by infected code. Nowadays, modern OS are capable of thwarting the attack by such kind of virus.
   - **Macro viruses:** Macros are defined as a group of lines, technically speaking a subroutine, which is capable of infecting documents like – MSOffice, that make use of OLE (Object Linking and Embedding) formats.
   - **Network viruses:** These viruses make use of protocols and commands of computer network or email to spread themselves in the system.

   There is a large number of combinations possible, for example, file-boot viruses infecting both files and boot sectors on the disks. Another example of the combo – network macrovirus, not only infect the documents which are being edited, but also send copies of itself by email.

2. **Mode of Operation:** Viruses can be described as follows:
   - **TSR:** A terminate-and-stay-resident (TSR) virus infects a computer and remains in RAM and then performs an action for which it was planted. The virus becomes a part of the OS and remains active always. In contrast a non-resident virus is active for a limited time only.
   - **Stealth algorithm:** These viruses do not reveal their identity.
   - **Self-encrypting and polymorphic:** Such viruses are not detected by antivirus as they do not have a fixed signature. They are able to change themselves with every infection, and therefore, are very hard to detect.

3. **Destructive capabilities:** Viruses can be described as follows:
   - **Harmless:** Do not affect the computing resources.
   - **Not dangerous:** Have a limited effect on computing resources. For example, a certain sound can be produced after the computer is ON for say 2 hours.

- **Dangerous:** They disrupt the normal working of computing resources, such as you might not be able to read from the USB port of the computer.
- **Very dangerous:** Certain viruses result in loss of data, such as erasing system files which required when a system is to be booted, so as a result, the system does not work (boot).

7.11.6 Anti Virus Approaches

The best approach is the protection against the virus. If at all the system has been infected by virus, the following steps are commonly used:

1. **Detect:** Locate it (by checking the file size of vital files).
2. **Identify:** Identify the virus (may be your antivirus scanner might tell you).
3. **Remove:** All the traces of the virus are removed (disinfected by the antivirus scanner).

The antivirus scanners use different approaches to protect the system, some of these approaches are:

1. **Signature scanning:** Each virus has a certain signature (a code), which is located and deleted.
2. **Heuristic scanning:** Makes use of heuristics (rules of thumb or a guess) based upon certain analysis.
3. **Generic decryption:** The virus decrypts itself after the execution of code.
4. **Behaviour blocking:** Its function is similar to a firewall, it blocks the execution of a suspicious piece of code.

IMPORTANT FORMULAS

1. Turnaround Time: TAT = CT − AT [(CT: completion time, AT: arrival time)]
2. Waiting Time: WT = TAT − BT [(BT: burst time)]
3. Waiting Time for Round Robin: $(n - 1)q + nc$
4. Complexity of Safe Sequence using Banker's Algorithm: $m \times n^2$(n: processes, m: resources)
5. To Avoid Deadlock:
   - Total need $< m + n$
   - Max need of each process should be between 1 and m resources
6. Response Ratio $= \dfrac{w+s}{s}$(w = waiting time, s = service time or burst time)
7. Thread maintains different registers and stack but shares code, data and files.
8. Need[I] = Max[I] − Allocation[I]
9. If program contains n fork() call, then the number of created child process are $(2^n - 1)$.
10. Number of pages $= \dfrac{\text{LAS}}{\text{Page size}}$
11. Number of frames $= \dfrac{\text{PAS}}{\text{Frame size}}$
12. $E_{\text{MAT}} = X(C+M)+(1-X)(C+2M)$
13. Effective memory access time (E_{MAT}) $= P \times S + (1-P) \times M$
14. Total size of file system

$$= \left\{\text{Number of direct DBA} + \left(\frac{\text{DBA size}}{\text{DBA}}\right) + \left(\frac{\text{DBA size}}{\text{DBA}}\right)^2 + \left(\frac{\text{DBA size}}{\text{DBA}}\right)^3 + \cdots\right\} \times \text{DB size}$$

15. Number of disk block possible in one disk block = $\left(\dfrac{\text{DBA size}}{\text{DBA}}\right)$
16. Number of cylinder per track = Number of tracks per surface
17. Capacity in disk = (Plotters × Surface × Tracks × Sector × Sector size) bytes
18. Number of bits required to identify a sector = (Bits for plotter + bits for surface + bits for tracks + bits for sectors)
19. Time to read four continuous sectors = (Seek time + rotational latency + transfer time)

SOLVED EXAMPLES

1. The term thrashing is used to define

(a) A reduce page I/O
(b) A decreased degree of multiprogramming
(c) An excessive page I/O
(d) Improvement(s) in the system performance

Solution: Thrashing means that the system more busy in dealing with page faults instead of doing some constructive work. Thus, it leads to excessive page I/O.

Ans. (c)

2. Page fault occurs when

(a) The page is not in cache memory.
(b) The page is in main memory.
(c) The page is not in main memory.
(d) The page has an address, which cannot be loaded.

Solution: If page is not found in page table, it generates the page fault.

Ans. (c)

3. If $m =$ number of resources, $n =$ number of processes in the system, then Banker's algorithm although a general algorithm may require ________ operations.

(a) $n \times n \times m$
(b) $(m \times m) \times n$
(c) $(m \times n) \times n$
(d) $(n \times m) \times m$

Solution: Complexity of Safe Sequence using Banker's Algorithm is $m \times n^2$
where n is the number of processes, and m is the number of resources.

Ans. (a, c)

4. Mutual exclusion problem occurs

(a) Between two disjoint processes that do not interact.
(b) Among processes that share resources.
(c) Among processes that do not use the same resource.
(d) Between two processes that uses different resources of different machines.

Solution: In this problem, if process P_i is executing in its critical section, then no other process can execute their critical sections.

Ans. (b)

5. Linux operating system uses

(a) i-Node scheduling.
(b) fair pre-emptive scheduling.
(c) RR scheduling.
(d) highest penalty ratio next.

Solution: It uses fair pre-emptive scheduling.

Ans. (b)

6. In modern operating systems such as Windows 2000, all the processor-dependent code is isolated in a dynamic link library called

(a) NTFS file system
(b) Hardware abstraction layer
(c) Microkernel
(d) CORBA Process Manager

Solution: The hardware abstraction layer (HAL) provides logical link between NT-based operating systems and physical hardware of the computer.

Ans. (b)

7. Translation look-aside buffer (TLB) is

(a) A cache-memory in which item to be searched is compared one by one with the keys.
(b) A cache-memory in which item to be searched is compared with all the keys simultaneously.
(c) A main memory in which item to be searched is compared one by one with the keys.
(d) An associative memory in which item to be searched is compared with all the keys simultaneously.

Solution: TLB is a cache memory used by memory management unit to improve the translation speed of virtual address.

Ans. (d)

8. Windows 2000 operating system include the following process states:

(a) Ready, running and waiting
(b) Ready, standby, running, waiting, transition and terminated
(c) Ready, running, waiting, transition, sleep and terminated
(d) Standby, running, transition, sleep and terminated

Solution: It includes ready, standby, running, waiting, transition and terminated states.

Ans. (b)

9. A memory management algorithm, realizing virtual memory, partially swaps out a process. This is similar to which kind of CPU scheduling.

(a) Short-term scheduling
(b) Long-term scheduling

(c) Medium-term scheduling
(d) Mutual exclusion

Solution: Medium-term scheduling is one in which decision is taken to add the number of processes that are present either partially or fully in the main memory. This is similar to CPU scheduling. In short-term scheduling the available process is executed by the processor. In long-term scheduling, the process is added to the pool of processes to be executed.

Ans. (c)

10. The working set model is used in memory management to implement the concept of

(a) Segmentation (b) Principle of locality
(c) Paging (d) Thrashing

Solution: Fact

Ans. (b)

11. Which scheduling algorithm gives a minimum average waiting time?

(a) RR (b) SJF
(c) FCFS (d) Priority

Solution: Shortest job first algorithm.

Ans. (b)

12. If the disk is rotating at 3600 rpm, determine the effective data transfer rate which is defined as the number of bytes transferred per second between disk and memory. (Given size of track = 512 bytes)

(a) 20Kbps (b) 30 Kbps
(c) 40Kbps (d) 50Kbps

Solution: 3600 rotation $\rightarrow$ 60 s

1 rotation $\rightarrow 60/3600 = 1/60$ s

Transfer rate = $(60 \times \text{track size})$ per second
= 60×512 bytes/s
= 30 Kbps

Ans. (b)

13. Which of the following features will characterize an OS as multiprogrammed OS?

1. More than one program may be loaded into main memory at the same time.
2. If a program waits for certain event, another program is immediately scheduled.
3. If the execution of a program terminates, another program is immediately scheduled.

(a) Only
(b) (1) and (2) only
(c) (1) and (3) only
(d) (1), (2) and (3) only

Solution: Multiprogrammed OS performs all the given three tasks.

Ans. (d)

14. The P and V operations on counting semaphores, where s is a counting semaphore are defined as follows:

P(s):$s = s - 1$
if $s < 0$ then wait;
$V(s)$:$s = s + 1$
if $s = 0$ then wake up a process waiting on s.
Assume that Pb and Vb are the wait and signal operation on binary semaphore. Two binary semaphores Xb and Yb are used to implement the semaphore operation $P(s)$ and $V(s)$ as follows:

```
P(s): Pb(Xb);
s=s-1
if(s<0){
Vb(Xb);
Pb(Yb);
}
else Vb(Xb);
V(s): Pb(Yb);
s= s +1
if(s<=0)
Vb(Yb);
```

The initial value of Xb and Yb are, respectively,

(a) 0 and 0 (b) 0 and 1
(c) 1 and 0 (d) 1 and 1

Solution: From the give code, we get $P(s)$ and $V(s)$ decrements and increment the value of semaphore, respectively. So, from the given conditions, we conclude that to avoid mutual exclusion the value of Xb should be 1 and that of Yb should be 0.

Ans. (c)

15. Consider the following table of arrival time and burst time for three processes P_0, P_1 and P_2.

Process	Arrival Time	Burst Time
P_0	0	9
P_1	1	4
P_2	2	9

The pre-emptive shortest job first scheduling algorithm is used. Scheduling is carried out only at arrival or completion of process. The average waiting time for the three processes is ___.

Solution: Waiting time for $P_0 = 5 - 1 = 4$ s
Waiting time for $P_1 = 1 - 1 = 0$ s
Waiting time for $P_2 = 13 - 2 = 4$ s
Average waiting time = $4 + 0 + 11/3 = 5$ s

Ans. (5)