OPERATING SYSTEMS

Dr.Sushma Jaiswal
Assistant Professor,CSIT
Guru Ghasidas Central University
Bilaspur(C.G.)

Tarun Jaiswal
AME, MCA(Persuing)

Reg.No.U74120 MH2013 PTC 251205
Harshwardhan Publication Pvt.Ltd.
At.Post.Limbaganesh,Tq.Dist.Beed
Pin-431126 (Maharashtra) Cell:07588057695,09850203295
harshwardhanpubli@gmail.com, vaidyawarta@gmail.com
All Types Educational & Reference Book Publisher & Distributors

❖ **Publisher :(Online)**
Harshwardhan Publication Pvt.Ltd.
Limbaganesh, Dist. Beed (Maharashtra)
Pin-431126, vidyawarta@gmail.com

❖ **Page design**
Harshwardhan Publication Pvt.Ltd.
Limbaganesh, Dist. Beed, Pin-431126

❖ **Online Edition: February 2017**

ISBN 978-93-85882-27-2

❖ **Price : 300/ -**

Dedicated to My Parents
&
Loving Brother 'Santosh'

OPERATING SYSTEM

PREFACE

Operating systems are an essential part of any computer system. Similarly, a course on operating systems is an essential part of any computer-science education. This field is undergoing change at a breathtakingly rapid rate, as computers are now prevalent in virtually every application, from games for children through the most sophisticated planning tools for governments and multinational firms. Yet the fundamental concepts remain fairly clear, and it is on these that we base this book. We wrote this book as a text for an introductory course in operating systems at the junior or senior undergraduate level, or at the first-year graduate level. It provides a clear description of the concepts that underlie operating systems. As prerequisites, we assume that the reader is familiar with basic data structures, computer organization, and a high-level language, such as C. The hardware topics required for an understanding of operating systems are included in Chapter. The fundamental concepts and algorithms covered in the book are often based on those used in existing commercial operating systems. Our aim is to present these concepts and algorithms in a general setting that is not tied to one particular operating system. Concepts are presented using intuitive descriptions. Important theoretical results are covered, but formal proofs are omitted. The bibliographical notes contain pointers to research papers in which results were first presented and proved, as well as references to material for further reading. In place of proofs, figures and examples are used to suggest why we should expect the result in question to be true.

CONTENTS

13. MULTIPROCESSOR SYSTEMS: Advantages of multiprocessors, Multiprocessor classification, Multiprocessor interconnections, Types of multiprocessor operating systems, Multiprocessor operating functions and requirements, Operating system design and implementation issues

14. OPERATING SYSTEMS IN DISTRIBUTED PROCESSING: Characteristics of distributed processing, Characteristics of parallel processing, Centralized Vs distributed processing, Network operating system architecture, Functions of NOS, Global operating system, Remote procedure call, Distributed file management, Cache management, Printer server, Client-based computing, Client-server computing

15. SECURITY AND PROTECTION: Security threats, Attacks on security, Computer worms, Computer virus, Security design principles, Authentication, Protection mechanism, Encryption, Security in distributed environment

CHAPTER 1

1. What is an operating system? What are its functions?
2. What is the need for secondary storage and a storage hierarchy?

INTRODUCTION

An operating system is a program that is an intermediary between a user and the computer hardware. Different levels in a computer system are shown below (Figure 1.1).

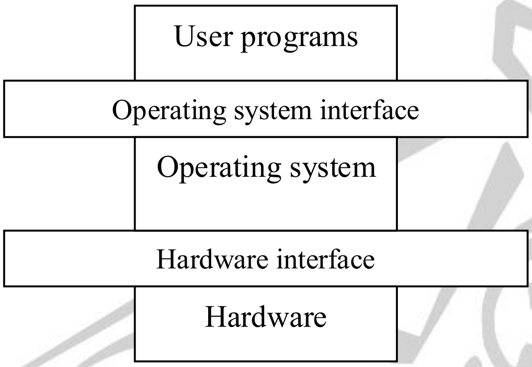


Figure 1.1: Levels in a computer system

An operating system provides an environment in which a user can execute programs. The purpose of an operating system is two fold:

1. To make the computer system convenient to use.
2. To use the computer hardware in an efficient manner.

1.1 OPERATING SYSTEM AS A RESOURCE MANAGER

A computer system consists of hardware, the operating system, application programs and users (Figure 1.2).

Hardware includes the central processing unit (CPU), memory and input / output (I/O) devices. These provide the basic computing resources. Application programs like compilers, database management systems, editors and many more allow users to use the hardware to solve their problems. The operating system controls and co-ordinates the use of the hardware among the various application programs of the various users. It provides an environment where software and data make use of the hardware in the computer system.

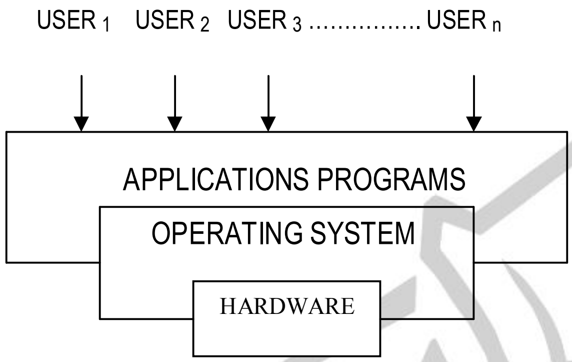


Figure 1.2: Components of a computer system

The operating system is a resource allocator. Many hardware and software resources are needed to solve a problem. These include CPU time, memory space, file storage, I/O devices and so on. The operating system as a manager of these resources, allocates resources to programs of users as required by the tasks. Since there is always a possibility of conflicts for resources because of more than one user requesting for resources, the operating system resolves the conflicts for efficient operation of the computer system.

The operating system as a control program controls the execution of user programs to avoid errors and improve use of the computer system.

A computer system has been developed to execute user programs there by making work easier for the user. But the bare hardware alone is not easy to use. Hence application programs are developed. These require certain common operations such as I/O device handling. The common functions of controlling and allocating resources are present in one piece of software – the operating system.

The primary goal of an operating system, that is convenience for the user is seen in small computers where as the secondary goal of efficient operation is important in large shared multi-user systems.

1.2 STORAGE STRUCTURE

Programs to be executed need to reside in the main memory. This is the memory that the CPU directly accesses. The main memory can be visualized as an array of bytes / words, each having its own address. Load / Store instructions allow movement of a word from the main memory to a CPU register and vice versa.

The Von-Neumann architecture brought in the concept of a stored program. The following (Figure 1.3) is the sequence of steps in one instruction execute cycle:

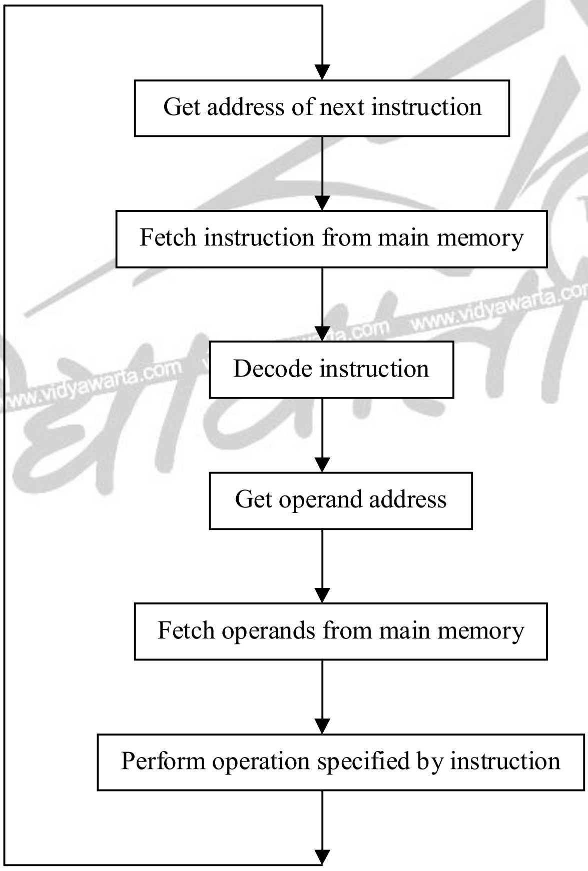

Get address of next instruction
Fetch instruction from main memory
Decode instruction
Get operand address
Fetch operands from main memory
Perform operation specified by instruction

Figure 1.3: Instruction execute cycle

Program and data are to be in the main memory for execution. Ideally they should reside there permanently, which is not possible because main memory

- is not all that big to hold all needed programs and data permanently.
- is volatile (contents are lost when power is turned off).
- cost is very high.

Secondary storage is an answer to the above listed problems. The main characteristics of secondary storage include

- large storage space
- Non-volatile or permanent storage
- Low cost

Thus program and data need to be loaded from secondary storage to main memory during execution. Common secondary storage devices include disks and tapes.

1.3 STORAGE HIERARCHY

The various storage devices differ in speed, cost, size and volatility (permanence of storage). They can be organized in a hierarchy. Shown below are some different levels of hierarchy (Figure 1.4).

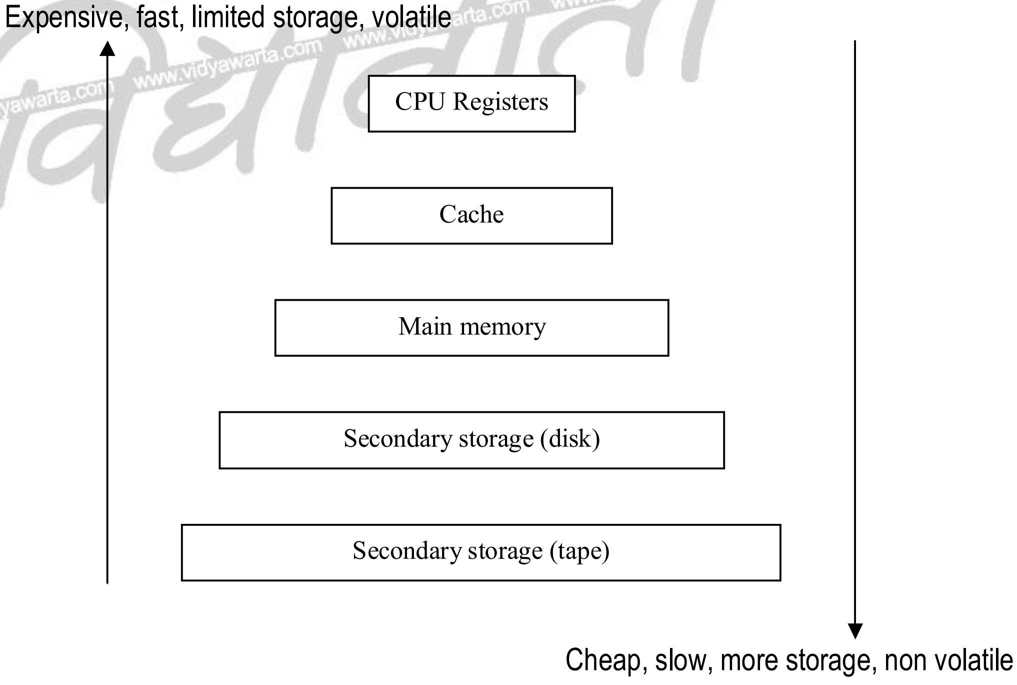

Expensive, fast, limited storage, volatile
CPU Registers
Cache
Main memory
Secondary storage (disk)
Secondary storage (tape)
Cheap, slow, more storage, non volatile

Figure 1.4: Storage device hierarchy

As one moves up the hierarchy, the devices are expensive but fast and as we move down the hierarchy, the cost per bit of storage decreases but access times increases (devices are slow). There usually exists a tradeoff in the amount of storage required at each level in the hierarchy. Of the various storage devices in the hierarchy, those above the disks are volatile and the rest are non-volatile.

Thus the design of a computer system balances or makes a tradeoff of all the above factors namely speed, cost and volatility. It provides as much expensive volatile memory as necessary and as much inexpensive non-volatile memory as possible.

1.4 SUMMARY

This chapter has introduced the term 'operating system' and its primary goals. They are

- Efficient use of computer resources
- To provide a good user-friendly interface

We have also discussed the Von-Neumann concept of a stored program and the need for secondary storage. Since different secondary storage devices differ with respect to speed, cost and permanence of storage, a hierarchy of storage exists and the design of a computer system makes the right balance of the above factors.

1.5 EXERCISE

3. What is an operating system? What are its functions?
4. What is the need for secondary storage and a storage hierarchy?

1.6 ACTIVITY

Make a note of some popular operating systems.

CHAPTER 2

HISTORY OF OPERATING SYSTEMS

The present day operating systems have not been developed overnight. Just like any other system, operating systems also have evolved over a period of time, starting from the very primitive systems to the present day complex and versatile ones. Described below is a brief description of the evolution of operating systems.

2.1 SIMPLE BATCH SYSTEMS

Computers in earlier days of their inception were very bulky, large machines usually run from a console. I/O devices consisted of card readers, tape drives and line printers. Direct user interaction with the system did not exist. Users made a job consisting of programs, data and control information. The job was submitted to an operator who would execute the job on the computer system. The output appeared after minutes, hours or sometimes days. The user collected the output from the operator, which also included a memory dump. The operating system was very simple and its major task was to transfer control from one job to another. The operating system was resident in memory (Figure 2.1).

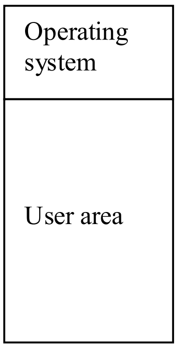


Figure 2.1: Memory layout for simple batch system

To speed up processing, jobs with the same needs were batched together and executed as a group. For example, all FORTRAN jobs were batched together for execution; all COBOL jobs were batched together for execution and so on. Thus came into existence batch operating systems.

Even though processing speed increased to a large extent because of batch processing, the CPU was often idle. This is because of the disparity between operating speeds of electronic devices like the CPU and the mechanical I/O devices. CPU operates in the microsecond / nanosecond ranges whereas I/O devices work in the second / minute range. With improvements in technology, I/O devices became faster but CPU speeds became even faster. So the problem of disparity in operating speeds only widened.

2.2 SPOOLING

The introduction of disks brought in the concept of spooling. Instead of reading from slow input devices like card readers into the computer memory and then processing the job, the input is first read into the disk. When the job is processed or executed, the input is read directly from the disk. Similarly when a job is executed for printing, the output is written into a buffer on the disk and actually printed later. This form of processing is known as spooling an acronym for Simultaneous Peripheral Operation On Line. Spooling uses the disk as a large buffer to read ahead as possible on input devices and for storing output until output devices are available to accept them.

Spooling overlaps I/O of one job with the computation of other jobs. For example, spooler may be reading the input of one job while printing the output of another and executing a third job (Figure 2.2).

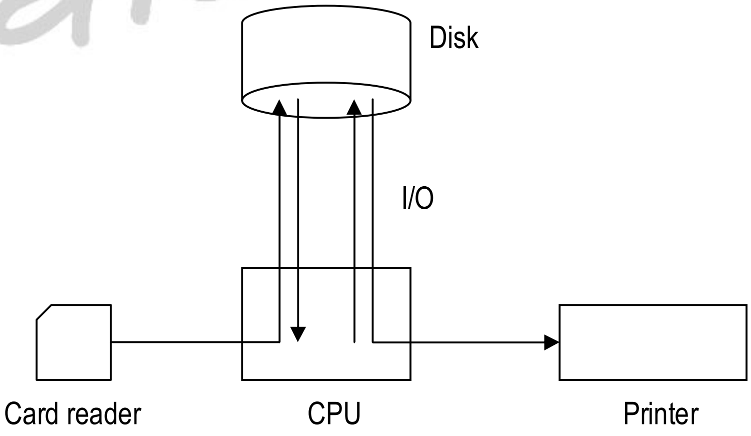


Figure2.2: Spooling

Spooling increases the performance of the system by allowing both a faster CPU and slower I/O devices to work at higher operating rates.

2.3 MULTIPROGRAMMED BATCH SYSTEMS

The concept of spooling introduced an important data structure called the job pool. Spooling creates a number of jobs on the disk which are ready to be executed / processed. The operating system now has to choose from the job pool, a job that is to be executed next. This increases CPU utilization. Since the disk is a direct access device, jobs in the job pool may be scheduled for execution in any order, not necessarily in sequential order.

Job scheduling brings in the ability of multiprogramming. A single user cannot keep both CPU and I/O devices busy. Multiprogramming increases CPU utilization by organizing jobs in a manner such that CPU always has a job to execute.

The idea of multiprogramming can be described as follows. A job pool on the disk consists of a number of jobs that are ready to be executed (Figure 2.3). Subsets of these jobs reside in memory during execution. The operating system picks and executes one of the jobs in memory. When this job in execution needs an I/O operation to complete, the CPU is idle. Instead of waiting for the job to complete the I/O, the CPU switches to another job in memory. When the previous job has completed the I/O, it joins the subset of jobs waiting for the CPU. As long as there are jobs in memory waiting for the CPU, the CPU is never idle. Choosing one out of several ready jobs in memory for execution by the CPU is called CPU scheduling.

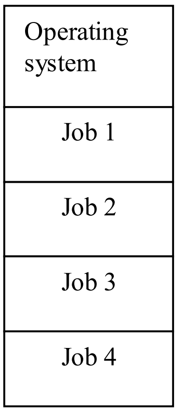


Figure 2.3: Memory layout for multiprogrammed system

2.4 TIME SHARING SYSTEMS

Multiprogrammed batch systems are suited for executing large jobs that need very little or no user interaction. On the other hand interactive jobs need on-line communication between the user and the system. Time-sharing / multi tasking is a logical extension of multiprogramming. It provides interactive use of the system.

CPU scheduling and multiprogramming provide each user one time slice (slot) in a time-shared system. A program in execution is referred to as a process. A process executes for one time slice at a time. A process may need more than one time slice to complete. During a time slice a process may finish execution or go for an I/O. The I/O in this case is usually interactive like a user response from the keyboard or a display on the monitor. The CPU switches between processes at the end of a time slice. The switching between processes is so fast that the user gets the illusion that the CPU is executing only one user’s process.

Multiprogramming and Time-sharing are the main concepts in all modern operating systems.

2.5 SUMMARY

This chapter has traced the evolution of operating systems. Operating systems have evolved from the simple batch systems to the latest multiprogrammed, time-sharing systems. We have also studied the concept of spooling where the disk acts as a buffer and queues up processes for CPU to execute. This increases system performance and allows a faster device like the CPU to work with slower I/O devices. Multiprogramming allows many ready jobs to be present in memory simultaneously for execution. This helps increases CPU utilization.

2.6 EXERCISE

1. Discuss the evolution of the present day computer systems.
2. Explain the following concepts:
   Spooling, Multiprogramming, Time-sharing.

2.7 ACTIVITY

For the operating systems noted, study their evolution find out which of the above concepts are implemented.

CHAPTER 3

PROCESS MANAGEMENT

The operating system is a CPU manager since it manages one of the system resources called CPU computing time. The CPU is a very high speed resource to be used efficiently and to the maximum. Users submit jobs to the system for execution. These jobs are run on the system as processes and need to be managed thus are scheduled.

3.1 WHAT IS A PROCESS?

A program in execution is a process. A process is executed sequentially, one instruction at a time. A program is a passive entity. For example, a file on the disk. A process on the other hand is an active entity. In addition to program code, it includes the values of the program counter, the contents of the CPU registers, the global variables in the data section and the contents of the stack that is used for subroutine calls.

3.2 PROCESS STATE

A process being an active entity changes state as execution proceeds. A process can be any one of the following states:

- New: Process being created.
- Running: Instructions being executed.
- Waiting: Process waiting for an event to occur.
- Ready: Process waiting for CPU.
- Terminated: Process has finished execution.

A state diagram (Figure 3.1) is used to diagrammatically represent the states and also the events that trigger the change of state of a process in execution.

3.3 PROCESS CONTROL BLOCK

Every process has a number and a process control block (PCB) represents a process in an operating system. The PCB contains information that makes the process an active entity. The PCB consists of the number of the process, its state, value of the program counter, contents of the CPU registers and much more as shown below (Figure 3.2). The PCB serves as a repository of information about a process and varies from process to process.

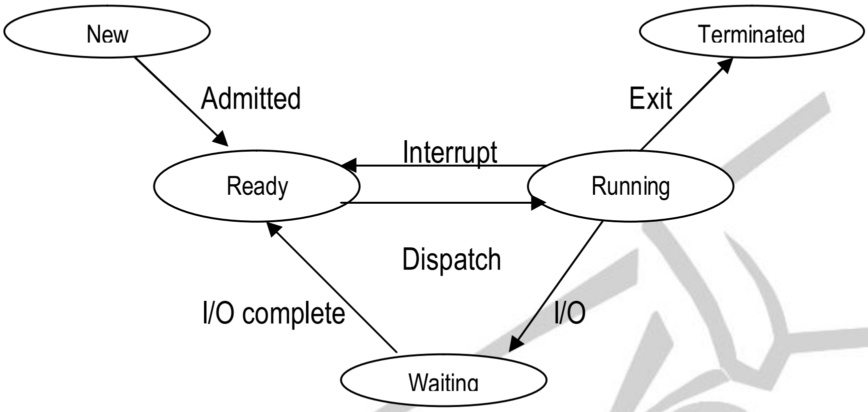


Figure 3.1: Process state diagram

Pointer	Process state
Process number	
Program counter	
CPU registers	
Memory management information	
I/O information	
. .	

Figure 3.2: Process control block

The process state gives the current state of the process is. It can be either new, ready, running, waiting or terminated.

The program counter contains the address of the next instruction to be executed.

The contents of the CPU registers which include the accumulator, general purpose registers, index register, stack pointer and others are required when the CPU switches from one process to another for a successful continuation.

CPU scheduling information consists of parameters for scheduling various processes by the CPU.

Memory management information is dependent on the memory system used by the operating. It includes information pertaining to base registers, limit registers, page tables, segment tables and other related details.

Process numbers and amount of CPU time used by the process are stored for accounting purposes in case users are to be billed for the amount of CPU time used.

I/O information gives details about the I/O devices allotted to the process, the number of files that are open and in use by the process and so on.

3.4 PROCESS SCHEDULING

The main objective of multiprogramming is to see that some process is always running so as to maximize CPU utilization where as in the case of time sharing, the CPU is to be switched between processes frequently so that users interact with the system while their programs are executing. In a uniprocessor system, there is always a single process running while the other processes need to wait till they get the CPU for execution on being scheduled.

As a process enters the system, it joins a job queue that is a list of all processes in the system. Some of these processes are in the ready state and are waiting for the CPU for execution. These processes are present in a ready queue. The ready queue is nothing but a list of PCB's implemented as a linked list with each PCB pointing to the next.

There are also some processes that are waiting for some I/O operation like reading from a file on the disk or writing on to a printer. Such processes are present in device queues. Each device has its own queue.

A new process first joins the ready queue. When it gets scheduled, the CPU executes the process until

1. an I/O occurs and the process gives up the CPU to join a device queue only to rejoin the ready queue after being serviced for the I/O.
2. it gives up the CPU on expiry of its time slice and rejoins the ready queue.

Every process is in this cycle until it terminates (Figure 3.3). Once a process terminates, entries for this process in all the queues are deleted. The PCB and resources allocated to it are released.

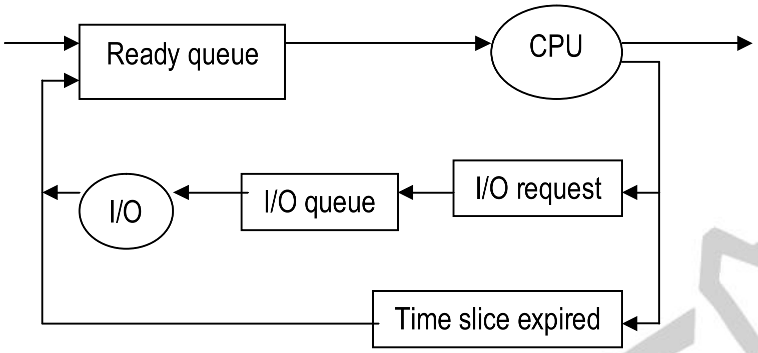


Figure 3.3: Queueing diagram of process scheduling

3.5 SCHEDULERS

At any given instant of time, a process is in any one of the new, ready, running, waiting or terminated state. Also a process moves from one state to another as long as it is active. The operating system scheduler schedules processes from the ready queue for execution by the CPU. The scheduler selects one of the many processes from the ready queue based on certain criteria.

Schedulers could be any one of the following:

- Long-term scheduler
- Short-term scheduler
- Medium-term scheduler

Many jobs could be ready for execution at the same time. Out of these more than one could be spooled onto the disk. The long-term scheduler or job scheduler as it is called picks and loads processes into memory from among the set of ready jobs. The short-term scheduler or CPU scheduler selects a process from among the ready processes to execute on the CPU.

The long-term scheduler and short-term scheduler differ in the frequency of their execution. A short-term scheduler has to select a new process for CPU execution quite often since processes execute for short intervals before waiting for I/O requests. Hence short-term scheduler must be very fast else the CPU will be doing only scheduling work.

A long-term scheduler executes less frequently since new processes are not created at the same pace at which processes need to be executed. The number of processes present in the ready queue determines the degree of multiprogramming. So the long-term scheduler determines this degree of multiprogramming. If a good selection is made by the long-term scheduler in choosing new jobs to join the ready queue, then the average rate of process creation must almost equal the average rate of processes leaving the system. The long-term scheduler is thus involved only when processes leave the system to bring in a new process so as to maintain the degree of multiprogramming.

Choosing one job from a set of ready jobs to join the ready queue needs careful selection. Most of the processes can be classified as either I/O bound or CPU bound depending on the time spent for I/O and time spent for CPU execution. I/O bound processes are those which spend more time executing I/O where as CPU bound processes are those which require more CPU time for execution. A good selection of jobs by the long-term scheduler will give a good mix of both CPU bound and I/O bound processes. In this case, the CPU will also be busy and so also the I/O devices. If this is not to be so, then either the CPU is busy and I/O devices are idle or I/O devices are busy and CPU is idle. Time sharing systems usually do not require the services the a long-term scheduler since every ready process gets one time slice of CPU time at a time in rotation.

Sometimes processes keep switching between ready, running and waiting states with termination taking a long time. One of the reasons for this could be an increased degree of multiprogramming meaning more number of ready processes than the system can handle. The medium-term scheduler handles such a situation. When system throughput falls below a threshold, some of the ready processes are swapped out of memory to reduce the degree of multiprogramming. Sometime later these swapped processes are reintroduced into memory to join the ready queue.

3.6 CONTEXT SWITCH

CPU switching from one process to another requires saving the state of the current process and loading the latest state of the next process. This is known as a context switch (Figure 3.4). Time that is required for a context switch is a clear overhead since the system at that instant of time is not doing any useful work. Context switch time varies from machine to machine depending on speed, the amount of saving and loading to be done, hardware support and so on.

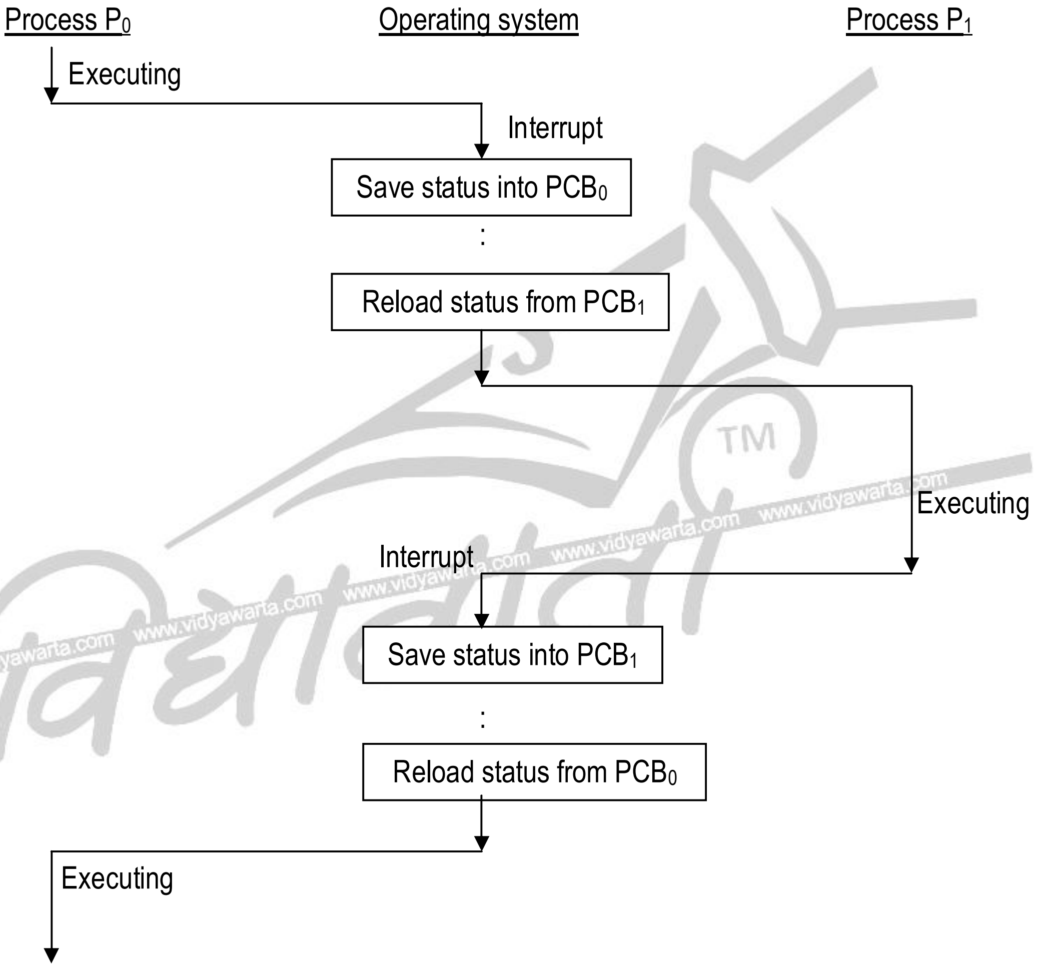


Figure 3.4: CPU switch from process to process

3.7 ADVANCED CONCEPTS

Multitasking and multithreading are two advanced concepts in process management. A short description of each one of them is given below.

3.7.1 MULTITASKING

A multi-user operating system supports multiple processes at the same time. Similarly a multitasking operating system supports processes each of which can consist of multiple tasks. Multiple tasks run concurrently within a process. Thus a task can be defined as an asynchronous path of code within a process. Let us analyze the following illustration.

Illustration: A program reads records from a tape, processes them and writes each one of them onto the disk. A simple algorithm for this statement could be as follows:

```
Begin
    While not end-of-file
    Begin
        Read a record
        Process the record
        Write the record
    End
End
```

Let round robin scheduling with a time slice of ‘n’ units be used. When the above process is scheduled, ‘n’ units of CPU time will be allocated to the process. But due to the Read, a system call is generated thereby utilizing only a fraction of its time slice. When the process is ready after the I/O and gets its next time slice, it again almost immediately goes for an I/O because of the Write. This cycle continues till all the records on the tape have been read, processed and written on to the disk. This being an I/O bound process it spends more time executing I/O.

Multitasking provides a solution for the problem described above. The entire process of reading, processing and writing is divided into two tasks. The advantage here is that the two tasks can be run concurrently when synchronized. The executable code for the two tasks is maintained separately as shown below:

```
Begin
    While not end-of-file
    Begin
        Task-0-begin
            Read a record
            Process the record
        Task-0-end
        Task-1-begin
            Write a record
        Task-1-end
    End
End
```

A process in execution has a process control block (PCB) associated with it which is created by the operating system. Similarly each task created needs a task control block (TCB). The TCB saves register information at the time of switching between tasks within a process.

Similar to processes, tasks are present in different states like ready, blocked or running. They could also have priorities. TCBs are linked in queues and scheduled for execution.

When a process with multiple tasks is scheduled by an operating system and the process gets its time slice for execution,

- A task of the process with highest priority is scheduled.
- When a time slice expires the entire process along with all its tasks goes into the ready state.
- If an allotted time slice is not over, but the current task is either over or blocked then the operating system searches and executes the next task of the process. Thus a process does not get blocked if there exists at least one task in it which can still execute within the allocated time slice.
- Different tasks need to communicate with each other. Hence inter task communication and synchronization is necessary.

An operating system with multitasking is complex when compared to one that does not support multitasking. CPU utilization improves with multitasking. For example, when a task is blocked another task of the same process can be scheduled instead of blocking the entire process as is the case with no multitasking. This also reduces the overhead of a context switch at process level. A context switch at task level does occur but has fewer overheads as compared to a context switch as process level. More the I/O and more the processing involved within a process, greater will be the advantage if the process is divided into tasks.

3.7.2 MULTITHREADING

A thread is similar to a task and multithreading is a concept similar to multitasking. A thread like a process has a program counter, stack and an area for saving register contents. It shares the same address space with others, that is, different threads read and write the same memory location. Hence inter thread communication is easy. Threads can be created dynamically their priorities changed and so on.

Another concept is that of a session. A session like a process gives rise to child sessions. Thus there exists a hierarchy consisting of sessions, each session having multiple processes and each process having multiple threads as shown below (Figure 11.1).

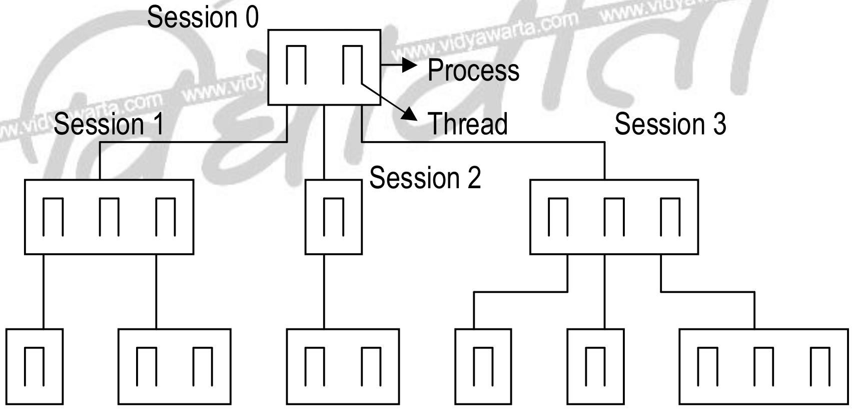


Figure 11.1: Process hierarchy

The scheduling algorithm in such a case is more complex. Multiple sessions are used to simulate multi window environments on dumb terminals. Each session has its own virtual terminal with a keyboard, screen, etc. Each session can be used to run different processes like one for a fore ground process (interactive editing), one for a background process (sorting), one for another background process (compilation) and so on. Use of a hot key (combination of keys) allows users to cycle through the various sessions.

3.8 SUMMARY

This chapter has introduced the concept of a process. A process is a program in execution. It goes through various states during its life cycle. Information required for the running of a process is maintained in a process control block. As explained in the previous chapter, multiprogramming keeps more than one ready processes in memory. This calls for scheduling these processes for efficient use of CPU. A proper mix of CPU and I/O bound jobs (long term scheduler) and proper scheduling of these processes (short term scheduler) increases CPU utilization. We have also learnt two new concepts – multitasking and multithreading.

3.9 EXERCISE

1. What is a process? With the help of a process state diagram, explain the different transitions of a process.
2. Discuss long-term, short-term and medium-term schedulers.
3. What is a context switch?
4. Differentiate between Multiprocessing, Multitasking and Multithreading.

3.10 ACTIVITY

For the operating systems noted, find out the schedulers involved in process scheduling. Also study the details that go into the process control block or its variant in each of the systems.

CHAPTER 4

CPU SCHEDULING

The CPU scheduler selects a process from among the ready processes to execute on the CPU. CPU scheduling is the basis for multiprogrammed operating systems. CPU utilization increases by switching the CPU among ready processes instead of waiting for each process to terminate before executing the next.

The idea of multiprogramming could be described as follows: A process is executed by the CPU until it completes or goes for an I/O. In simple systems with no multiprogramming the CPU is idle till the process completes the I/O and restarts execution. With multiprogramming, many ready processes are maintained in memory. So when CPU becomes idle as in the case above, the operating system switches to execute another process each time a current process goes into a wait for I/O.

4.1 CPU – I/O BURST CYCLE

Process execution consists of alternate CPU execution and I/O wait. A cycle of these two events repeats till the process completes execution (Figure 4.1). Process execution begins with a CPU burst followed by an I/O burst and then another CPU burst and so on. Eventually a CPU burst will terminate the execution. An I/O bound job will have short CPU bursts and a CPU bound job will have long CPU bursts.

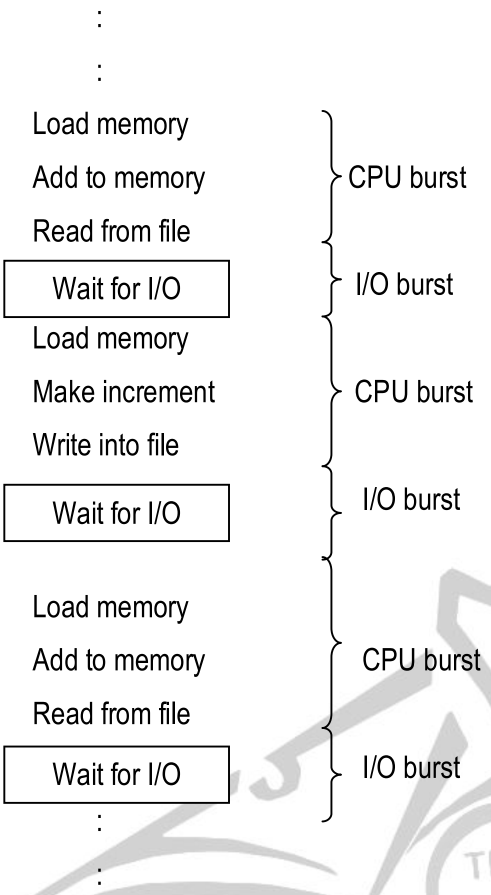


Figure 4.1: CPU and I/O bursts

4.2 PREEMPTIVE / NONPREEMPTIVE SCHEDULING

CPU scheduler has to take a decision when a process:

1. switches from running state to waiting (an I/O request).
2. switches from running state to ready state (expiry of a time slice).
3. switches from waiting to ready state (completion of an I/O).
4. terminates.

Scheduling under condition (1) or (4) is said to be nonpreemptive. In nonpreemptive scheduling, a process once allotted the CPU keeps executing until the CPU is released either by a switch to a waiting state or by termination. Preemptive scheduling occurs under condition (2) or (3). In preemptive scheduling, an executing process is stopped executing and returned to the ready queue to make the CPU available for another ready process.

4.3 SCHEDULING CRITERIA

Many algorithms exist for CPU scheduling. Various criteria have been suggested for comparing these CPU scheduling algorithms. Common criteria include:

1. CPU utilization: This may range from 0% to 100% ideally. In real systems it ranges from 40% for a lightly loaded systems to 90% for heavily loaded systems.
2. Throughput: Number of processes completed per time unit is throughput. For long processes may be of the order of one process per hour where as in case of short processes, throughput may be 10 or 12 processes per second.
3. Turnaround time: The interval of time between submission and completion of a process is called turnaround time. It includes execution time and waiting time.
4. Waiting time: Sum of all the times spent by a process at different instances waiting in the ready queue is called waiting time.
5. Response time: In an interactive process the user is using some output generated while the process continues to generate new results. Instead of using the turnaround time that gives the difference between time of submission and time of completion, response time is sometimes used. Response time is thus the difference between time of submission and the time the first response occurs.

Desirable features include maximum CPU utilization, throughput and minimum turnaround time, waiting time and response time.

4.4 SCHEDULING ALGORITHMS

Scheduling algorithms differ in the manner in which the CPU selects a process in the ready queue for execution.

4.4.1 FIRST COME FIRST SERVE (FCFS) SCHEDULING ALGORITHM

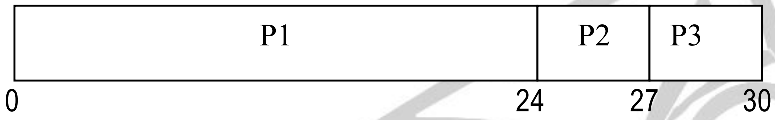
This is one of the very brute force algorithms. A process that requests for the CPU first is allocated the CPU first. Hence the name first come first serve. The FCFS algorithm is implemented by using a first-in-first-out (FIFO) queue structure for the ready queue. This queue has a head and a tail. When a process joins the ready queue its PCB is linked to the tail of the FIFO queue. When the CPU is idle, the process at the head of the FIFO queue is allocated the CPU and deleted from the queue.

Even though the algorithm is simple, the average waiting is often quite long and varies substantially if the CPU burst times vary greatly as seen in the following example.

Consider a set of three processes P1, P2 and P3 arriving at time instant 0 and having CPU burst times as shown below:

Process	Burst time (msecs)
P1	24
P2	3
P3	3

The Gantt chart below shows the result.



Average waiting time and average turnaround time are calculated as follows:

The waiting time for process P1 = 0 msecs

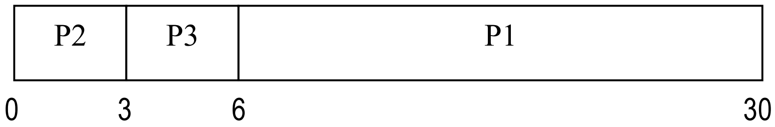
P2 = 24 msecs

P3 = 27 msecs

Average waiting time = (0 + 24 + 27) / 3 = 51 / 3 = 17 msecs.

P1 completes at the end of 24 msecs, P2 at the end of 27 msecs and P3 at the end of 30 msecs. Average turnaround time = (24 + 27 + 30) / 3 = 81 / 3 = 27 msecs.

If the processes arrive in the order P2, P3 and P3, then the result will be as follows:



Average waiting time = (0 + 3 + 6) / 3 = 9 / 3 = 3 msecs.

Average turnaround time = (3 + 6 + 30) / 3 = 39 / 3 = 13 msecs.

Thus if processes with smaller CPU burst times arrive earlier, then average waiting and average turnaround times are lesser.

The algorithm also suffers from what is known as a convoy effect. Consider the following scenario. Let there be a mix of one CPU bound process and many I/O bound processes in the ready queue.

The CPU bound process gets the CPU and executes (long I/O burst).

In the meanwhile, I/O bound processes finish I/O and wait for CPU thus leaving the I/O devices idle.

The CPU bound process releases the CPU as it goes for an I/O.

I/O bound processes have short CPU bursts and they execute and go for I/O quickly. The CPU is idle till the CPU bound process finishes the I/O and gets hold of the CPU.

The above cycle repeats. This is called the convoy effect. Here small processes wait for one big process to release the CPU.

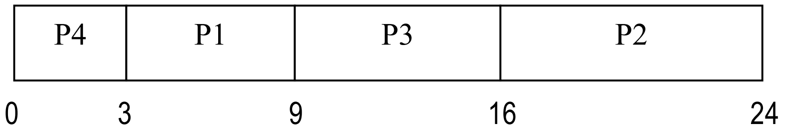
Since the algorithm is nonpreemptive in nature, it is not suited for time sharing systems.

4.4.2 SHORTEST JOB FIRST (SJF) SCHEDULING ALGORITHM

Another approach to CPU scheduling is the shortest job first algorithm. In this algorithm, the length of the CPU burst is considered. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. Hence the name shortest job first. In case there is a tie, FCFS scheduling is used to break the tie. As an example, consider the following set of processes P1, P2, P3, P4 and their CPU burst times:

Process	Burst time (msecs)
P1	6
P2	8
P3	7
P4	3

Using SJF algorithm, the processes would be scheduled as shown below.



Average waiting time = (0 + 3 + 9 + 16) / 4 = 28 / 4 = 7 msecs.

Average turnaround time = (3 + 9 + 16 + 24) / 4 = 52 / 4 = 13 msecs.

If the above processes were scheduled using FCFS algorithm, then

Average waiting time = (0 + 6 + 14 + 21) / 4 = 41 / 4 = 10.25 msecs.

Average turnaround time = (6 + 14 + 21 + 24) / 4 = 65 / 4 = 16.25 msecs.

The SJF algorithm produces the most optimal scheduling scheme. For a given set of processes, the algorithm gives the minimum average waiting and turnaround times. This is because, shorter processes are scheduled earlier than longer ones and hence waiting time for shorter processes decreases more than it increases the waiting time of long processes.

The main disadvantage with the SJF algorithm lies in knowing the length of the next CPU burst. In case of long-term or job scheduling in a batch system, the time required to complete a job as given by the user can be used to schedule. SJF algorithm is therefore applicable in long-term scheduling.

The algorithm cannot be implemented for CPU scheduling as there is no way to accurately know in advance the length of the next CPU burst. Only an approximation of the length can be used to implement the algorithm.

But the SJF scheduling algorithm is provably optimal and thus serves as a benchmark to compare other CPU scheduling algorithms.

SJF algorithm could be either preemptive or nonpreemptive. If a new process joins the ready queue with a shorter next CPU burst then what is remaining of the current executing process, then the CPU is allocated to the new process. In case of nonpreemptive scheduling, the current executing process is not preempted and the new process gets the next chance, it being the process with the shortest next CPU burst.

Given below are the arrival and burst times of four processes P1, P2, P3 and P4.

Process	Arrival time (msecs)	Burst time (msecs)
P1	0	8
P2	1	4
P3	2	9
P4	3	5

If SJF preemptive scheduling is used, the following Gantt chart shows the result.

P1	P2	P4	P1	P3

0 1 5 10 17 26

Average waiting time = ((10 – 1) + 0 + (17 – 2) + (15 – 3)) / 4 = 26 / 4 = 6.5 msecs.

If nonpreemptive SJF scheduling is used, the result is as follows:

P1	P2	P4	P3

0 8 12 17 26

Average waiting time = ((0 + (8 – 1) + (12 – 3) + (17 – 2)) / 4 = 31 / 4 = 7.75 msecs.

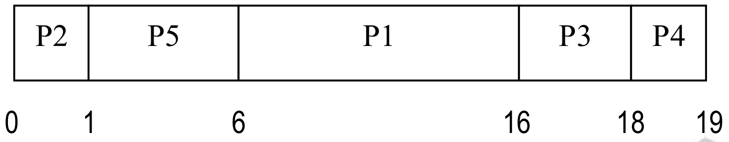
4.4.3 PRIORITY SCHEDULING

Each process can be associated with a priority. CPU is allocated to the process having the highest priority. Hence the name priority. Equal priority processes are scheduled according to FCFS algorithm.

The SJF algorithm is a particular case of the general priority algorithm. In this case priority is the inverse of the next CPU burst time. Larger the next CPU burst, lower is the priority and vice versa. In the following example, we will assume lower numbers to represent higher priority.

Process	Priority	Burst time (msecs)
P1	3	10
P2	1	1
P3	3	2
P4	4	1
P5	2	5

Using priority scheduling, the processes are scheduled as shown below:



Average waiting time = (6 + 0 + 16 + 18 + 1) / 5 = 41 / 5 = 8.2 msecs.

Priorities can be defined either internally or externally. Internal definition of priority is based on some measurable factors like memory requirements, number of open files, and so on. External priorities are defined by criteria such as importance of the user depending on the user's department and other influencing factors.

Priority based algorithms can be either preemptive or nonpreemptive. In case of preemptive scheduling, if a new process joins the ready queue with a priority higher than the process that is executing, then the current process is preempted and CPU allocated to the new process. But in case of nonpreemptive algorithm, the new process having highest priority from among the ready processes, is allocated the CPU only after the current process gives up the CPU.

Starvation or indefinite blocking is one of the major disadvantages of priority scheduling. Every process is associated with a priority. In a heavily loaded system, low priority processes in the ready queue are starved or never get a chance to execute. This is because there is always a higher priority process ahead of them in the ready queue.

A solution to starvation is aging. Aging is a concept where the priority of a process waiting in the ready queue is increased gradually. Eventually even the lowest priority process ages to attain the highest priority at which time it gets a chance to execute on the CPU.

4.4.4 ROUND-ROBIN (RR) SCHEDULING ALGORITHM

The round-robin CPU scheduling algorithm is basically a preemptive scheduling algorithm designed for time-sharing systems. One unit of time is called a time slice. Duration of a time slice may range between 10 msecs. and about 100 msecs. The CPU scheduler allocates to each process in the ready queue one time slice at a time in a round-robin fashion. Hence the name round-robin.

The ready queue in this case is a FIFO queue with new processes joining the tail of the queue. The CPU scheduler picks processes from the head of the queue for allocating the CPU. The first process at the head of the queue gets to execute on the CPU at the start of the current time slice and is deleted from the ready queue. The process allocated the CPU may have the current CPU burst either equal to the time slice or smaller than the time slice or greater than the time slice. In the first two cases, the current process will release the CPU on its own and there by the next process in the ready queue will be allocated the CPU for the next time slice. In the third case, the current process is preempted, stops executing, goes back and joins the ready queue at the tail there by making way for the next process.

Consider the same example explained under FCFS algorithm.

Process	Burst time (msecs)
P1	24
P2	3
P3	3

Let the duration of a time slice be 4 msecs, which is to say CPU switches between processes every 4 msecs in a round-robin fashion. The Gantt chart below shows the scheduling of processes.

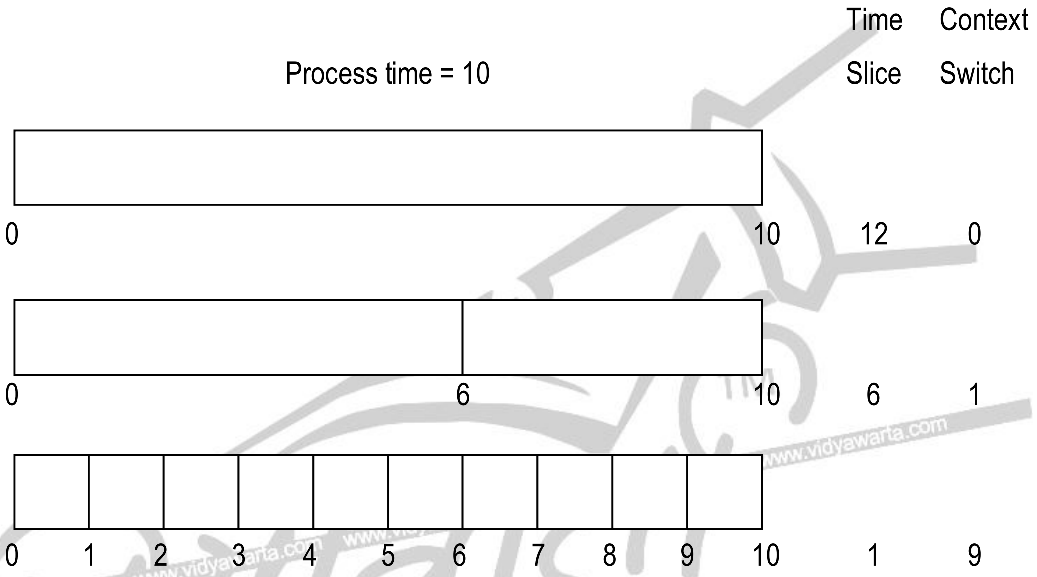
P1	P2	P3	P1	P1	P1	P1	P1

0 4 7 10 14 18 22 26 30

Average waiting time = (4 + 7 + (10 – 4)) / 3 = 17/ 3 = 5.66 msecs.

If there are 5 processes in the ready queue that is n = 5, and one time slice is defined to be 20 msecs that is q = 20, then each process will get 20 msecs or one time slice every 100 msecs. Each process will never wait for more than (n – 1) x q time units.

The performance of the RR algorithm is very much dependent on the length of the time slice. If the duration of the time slice is indefinitely large then the RR algorithm is the same as FCFS algorithm. If the time slice is too small, then the performance of the algorithm deteriorates because of the effect of frequent context switching. Below is shown a comparison of time slices of varying duration and the context switches they generate on only one process of 10 time units.

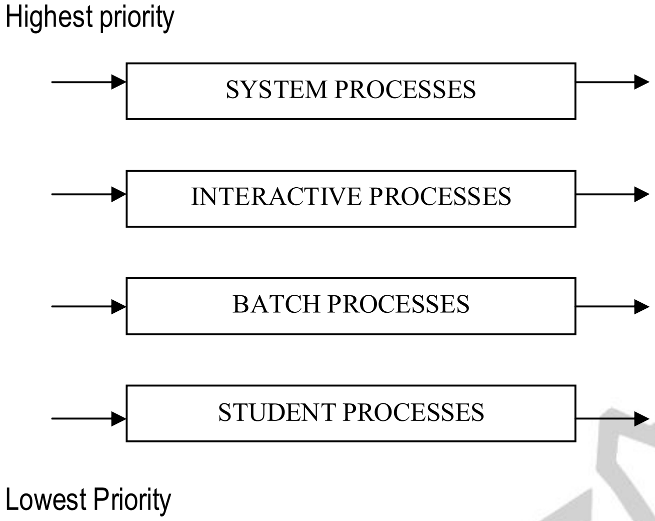


The above example shows that the time slice should be large with respect to the context switch time else if RR scheduling is used the CPU will spend more time in context switching.

4.4.5 MULTILEVEL QUEUE SCHEDULING

Processes can be classified into groups. For example, interactive processes, system processes, batch processes, student processes and so on. Processes belonging to a group have a specified priority. This algorithm partitions the ready queue into as many separate queues as there are groups. Hence the name multilevel queue. Based on certain properties is process is assigned to one of the ready queues. Each queue can have its own scheduling algorithm like FCFS or RR.

For example, Queue for interactive processes could be scheduled using RR algorithm where queue for batch processes may use FCFS algorithm. An illustration of multilevel queues is shown below (Figure 4.2).

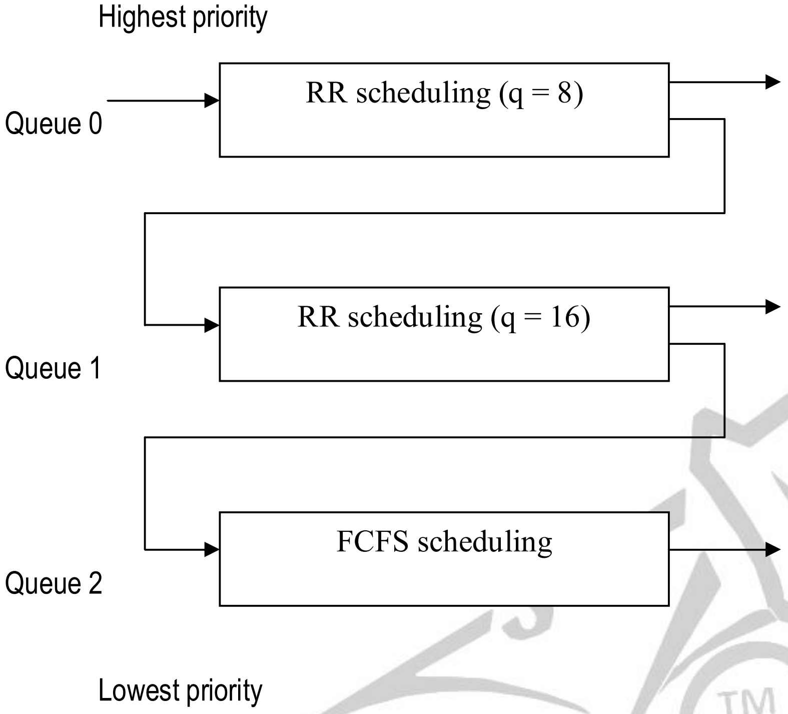


Queues themselves have priorities. Each queue has absolute priority over low priority queues, that is a process in a queue with lower priority will not be executed until all processes in a queue with higher priority have finished executing. If a process in a lower priority queue is executing (higher priority queues are empty) and a process joins a higher priority queue, then the executing process is preempted to make way for a process in the higher priority queue.

This priority on the queues themselves may lead to starvation. To overcome this problem, time slices may be assigned to queues when each queue gets some amount of CPU time. The duration of the time slices may be different for queues depending on the priority of the queues.

4.4.6 MULTILEVEL FEEDBACK QUEUE SCHEDULING

In the previous multilevel queue scheduling algorithm, processes one assigned to queues do not move or change queues. Multilevel feedback queues allow a process to move between queues. The idea is to separate out processes with different CPU burst lengths. All processes could initially join the highest priority queue. Processes requiring longer CPU bursts are pushed to lower priority queues. I/O bound and interactive processes remain in higher priority queues. Aging could be considered to move processes from lower priority queues to higher priority to avoid starvation. An illustration of multilevel feedback queues is shown below (Figure 4.3).



A process entering the ready queue joins queue 0. RR scheduling algorithm with q = 8 is used to schedule processes in queue 0. If the CPU burst of a process exceeds 8 msecs., then the process preempted, deleted from queue 0 and joins the tail of queue 1. When queue 0 becomes empty, then processes in queue 1 will be scheduled. Here also RR scheduling algorithm is used to schedule processes but q = 16. This will give processes a longer time with the CPU. If a process has a CPU burst still longer, then it joins queue 3 on being preempted. Hence highest priority processes (processes having small CPU bursts, that is I/O bound processes) remain in queue 1 and lowest priority processes (those having long CPU bursts) will eventually sink down. The lowest priority queue could be scheduled using FCFS algorithm to allow processes to complete execution.

Multilevel feedback scheduler will have to consider parameters such as number of queues, scheduling algorithm for each queue, criteria for upgrading a process to a higher priority queue, criteria for downgrading a process to a lower priority queue and also the queue to which a process initially enters.

4.5 SUMMARY

In this chapter we have discussed CPU scheduling. The long-term scheduler provides a proper mix of CPU-I/O bound jobs for execution. The short-term scheduler has to schedule these processes for execution. Scheduling can either be preemptive or nonpreemptive. If preemptive, then an executing process can be stopped and returned to ready state to make the CPU available for another ready process. But if nonpreemptive scheduling is used then a process once allotted the CPU keeps executing until either the process goes into wait state because of an I/O or it has completed execution. Different scheduling algorithms were studied.

4.6 EXERCISE

1. What do you understand by CPU-I/O burst?
2. Differentiate between preemptive and nonpreemptive scheduling.
3. Consider the work load shown below:

Process	Burst time (msecs)
P1	10
P2	29
P3	03
P4	07
P5	12

All the processors arrive at time 0 in the order given. Compute average turn around time and average waiting time using FCFS, SJF and RR (Q = 5msecs) scheduling algorithms. Illustrate using Gantt charts.

4.7 ACTIVITY

Find out the scheduling algorithms implemented for the operating systems noted.

CHAPTER 5

DEADLOCKS

Several processes compete for a finite set of resources in a multiprogrammed environment. A process requests for resources that may not be readily available at the time of the request. In such a case the process goes into a wait state. It may so happen that this process may never change state because the requested resources are held by other processes which themselves are waiting for additional resources and hence in a wait state. This situation is called a deadlock.

5.1 SYSTEM MODEL

The number of resources in a system is always finite. But the number of competing processes is many. Resources are of several types, each type having identical instances of the resource. Examples for resources could be memory space, CPU time, files, I/O devices and so on. If a system has 2 CPUs that are equivalent, then the resource type CPU time has 2 instances. If they are not equivalent, then each CPU is of a different resource type. Similarly the system may have 2 dot matrix printers and 1 line printer. Here the resource type of dot matrix printer has 2 instances where as there is a single instance of type line printer.

A process requests for resources, uses them if granted and then releases the resources for others to use. It goes without saying that the number of resources requested shall not exceed the total of each type available in the system. If a request for a resource cannot be granted immediately then the process requesting the resource goes into a wait state and joins the wait queue for the resource.

A set of processes is in a state of deadlock if every process in the set is in some wait queue of a resource and is waiting for an event (release resource) to occur that can be caused by another process in the set.

For example, there are 2 resources, 1 printer and 1 tape drive. Process P1 is allocated tape drive and P2 is allocated printer. Now if P1 requests for printer and P2 for tape drive, a deadlock occurs.

5.2 DEADLOCK CHARACTERIZATION

5.2.1 NECESSARY CONDITIONS FOR DEADLOCKS

A deadlock occurs in a system if the following four conditions hold simultaneously:

1. Mutual exclusion: At least one of the resources is non-sharable that is only one process at a time can use the resource.
2. Hold and wait: A process exists that is holding on to at least one resource and waiting for an additional resource held by another process.
3. No preemption: Resources cannot be preempted that is a resource is released only by the process that is holding it.
4. Circular wait: There exist a set of processes P_0, P_1, ….., P_n of waiting processes such that P_0 is waiting for a resource held by P_1, P_1 is waiting for a resource held by P_2, ….., P_{n-1} is waiting for a resource held P_n and P_n is in turn waiting for a resource held by P_0.

5.2.2 RESOURCE ALLOCATION GRAPH

Deadlocks can be described by a resource allocation graph. The resource allocation graph is a directed graph consisting of vertices and directed edges. The vertex set is partitioned into two types, a subset representing processes and another subset representing resources. Pictorially, the resources are represented by rectangles with dots within, each dot representing an instance of the resource and circles represent processes.

A directed edge from a process to a resource ($P_i \rightarrow R_j$) signifies a request from a process P_i for an instance of the resource R_j and P_i is waiting for R_j. A directed edge from a resource to a process ($R_j \rightarrow P_i$) indicates that an instance of the resource R_j has been allotted to process P_i. Thus a resource allocation graph consists of vertices which include resources and processes and directed edges which consist of request edges and assignment edges. A request edge is introduced into the graph when a process requests for a resource. This edge is converted into an assignment edge when the resource is granted. When the process releases the resource, the assignment edge is deleted. Consider the following system:

There are 3 processes P_1, P_2 and P_3.

Resources R_1, R_2, R_3 and R_4 have instances 1, 2, 1, and 3 respectively.

P_1 is holding R_2 and waiting for R_1.

P_2 is holding R_1, R_2 and is waiting for R_3.

P_3 is holding R_3.

The resource allocation gragh for a system in the above situation is as shown below (Figure 5.1).

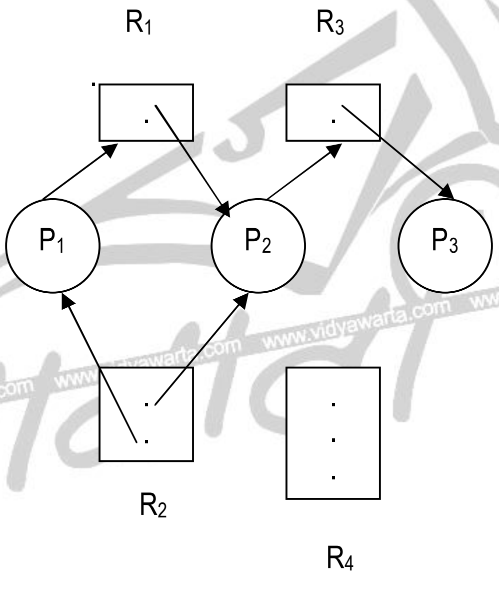


Figure 5.1: Resource allocation graph

If a resource allocation graph has no cycles (a closed loop in the direction of the edges), then the system is not in a state of deadlock. If on the other hand, there are cycles, then a deadlock may exist. If there are only single instances of each resource type, then a cycle in a resource allocation graph is a necessary and sufficient condition for existence of a deadlock (Figure 5.2). Here two cycles exist:

$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

Processes P_0, P_1 and P_3 are deadlocked and are in a circular wait. P_2 is waiting for R_3 held by P_3. P_3 is waiting for P_1 or P_2 to release R_2. So also P_1 is waiting for P_2 to release R_1.

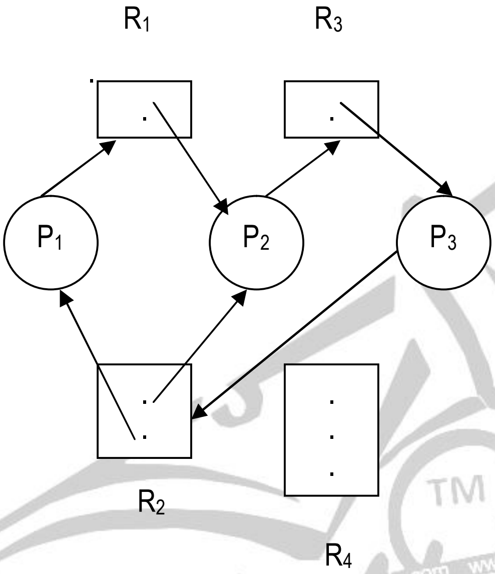


Figure 5.2: Resource allocation graph with deadlock

If there are multiple instances of resources types, then a cycle does not necessarily imply a deadlock. Here a cycle is a necessary condition but not a sufficient condition for the existence of a deadlock (Figure 5.3). Here also there is a cycle:

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

The cycle above does not imply a deadlock because an instance of R_1 released by P_2 could be assigned to P_1 or an instance of R_2 released by P_4 could be assigned to P_3 there by breaking the cycle.

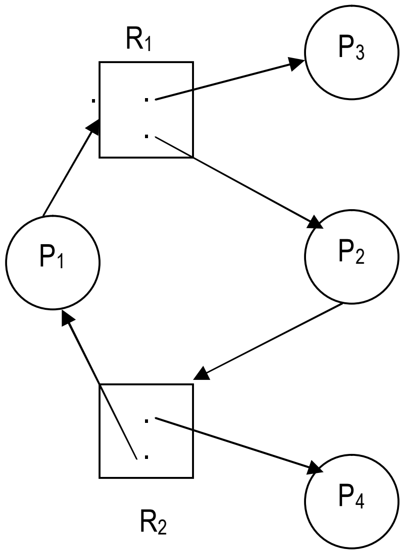


Figure 5.3: Resource allocation graph with a cycle but no deadlock

5.3 DEADLOCK HANDLING

Different methods to deal with deadlocks include methods to ensure that the system will never enter into a state of deadlock, methods that allow the system to enter into a deadlock and then recover or to just ignore the problem of deadlocks.

To ensure that deadlocks never occur, deadlock prevention / avoidance schemes are used. The four necessary conditions for deadlocks to occur are mutual exclusion, hod and wait, no preemption and circular wait. Deadlock prevention ensures that at least one of the four necessary conditions for deadlocks do not hold. To do this the scheme enforces constraints on requests for resources. Dead lock avoidance scheme requires the operating system to know in advance, the resources needed by a process for its entire lifetime. Based on this a priori information, the process making a request is either made to wait or not to wait in case the requested resource is not readily available.

If none of the above two schemes are used, then deadlocks may occur. In such a case, an algorithm to recover from the state of deadlock is used.

If the problem of deadlocks is ignored totally that is to say the system does not ensure that a deadlock does not occur and also does not provide for recovery from deadlock and such a situation arises, then there is no way out of the deadlock. Eventually the system may crash because more and more processes request for resources and enter into deadlock.

5.4 DEADLOCK PREVENTION

The four necessary conditions for deadlocks to occur are mutual exclusion, hod and wait, no preemption and circular wait. If any one of the above four conditions does not hold, then deadlocks will not occur. Thus prevention of deadlock is possible by ensuring that at least one of the four conditions cannot hold.

Mutual exclusion: Resources that can be shared are never involved in a deadlock because such resources can always be granted simultaneously access by processes. Hence processes requesting for such a sharable resource will never have to wait. Examples of such resources include read-only files. Mutual exclusion must therefor hold for non-sharable resources. But it is not always possible to prevent deadlocks by denying mutual exclusion condition because some resources are by nature non-sharable, for example printers.

Hold and wait: To avoid hold and wait, the system must ensure that a process that requests for a resource does not hold on to another. There can be two approaches to this scheme:

1. a process requests for and gets allocated all the resources it uses before execution begins.
2. a process can request for a resource only when it does not hold on to any other.

Algorithms based on these approaches have poor resource utilization. This is because resources get locked with processes much earlier than they are actually used and hence not available for others to use as in the first approach. The second approach seems to applicable only when there is assurance about reusability of data and code on the released resources. The algorithms also suffer from starvation since popular resources may never be freely available.

No preemption: This condition states that resources allocated to processes cannot be preempted. To ensure that this condition does not hold, resources could be preempted. When a process requests for a resource, it is allocated the resource if it is available. If it is not, than a check is made to see if the process holding the wanted resource is also waiting for additional resources. If so the wanted resource is preempted from the waiting process and allotted to the requesting process. If both the above is not true that is the resource is neither available nor held by a waiting process, then the requesting process waits. During its waiting period, some of its resources could also be preempted in which case the process will be restarted only when all the new and the preempted resources are allocated to it.

Another alternative approach could be as follows: If a process requests for a resource which is not available immediately, then all other resources it currently holds are preempted. The process restarts only when the new and the preempted resources are allocated to it as in the previous case.

Resources can be preempted only if their current status can be saved so that processes could be restarted later by restoring the previous states. Example CPU memory and main memory. But resources such as printers cannot be preempted, as their states cannot be saved for restoration later.

Circular wait: Resource types need to be ordered and processes requesting for resources will do so in increasing order of enumeration. Each resource type is mapped to a unique integer that allows resources to be compared and to find out the precedence order for the resources. Thus F: R → N is a 1:1 function that maps resources to numbers. For example:

F (tape drive) = 1, F (disk drive) = 5, F (printer) = 10.

To ensure that deadlocks do not occur, each process can request for resources only in increasing order of these numbers. A process to start with in the very first instance can request for any resource say R_i. There after it can request for a resource R_j if and only if $F(R_j)$ is greater than $F(R_i)$. Alternately, if $F(R_j)$ is less than $F(R_i)$, then R_j can be allocated to the process if and only if the process releases R_i.

The mapping function F should be so defined that resources get numbers in the usual order of usage.

5.5 DEADLOCK AVOIDANCE

Deadlock prevention algorithms ensure that at least one of the four necessary conditions for deadlocks namely mutual exclusion, hold and wait, no preemption and circular wait do not hold. The disadvantage with prevention algorithms is poor resource utilization and thus reduced system throughput.

An alternate method is to avoid deadlocks. In this case additional a priori information about the usage of resources by processes is required. This information helps to decide on whether a process should wait for a resource or not. Decision about a request is based on all the resources available, resources allocated to processes, future requests and releases by processes.

A deadlock avoidance algorithm requires each process to make known in advance the maximum number of resources of each type that it may need. Also known is the maximum number of resources of each type available. Using both the above a priori knowledge, deadlock avoidance algorithm ensures that a circular wait condition never occurs.

5.5.1 SAFE STATE

A system is said to be in a safe state if it can allocate resources upto the maximum available and is not in a state of deadlock. A safe sequence of processes always ensures a safe state. A sequence of processes < P_1, P_2, ..., P_n > is safe for the current allocation of resources to processes if resource requests from each P_i can be satisfied from the currently available resources and the resources held by all P_j where j < i.If the state is safe then P_i requesting for resources can wait till P_j's have completed. If such a safe sequence does not exist, then the system is in an unsafe state.

A safe state is not a deadlock state. Conversely a deadlock state is an unsafe state. But all unsafe states are not deadlock states as shown below (Figure 5.4).

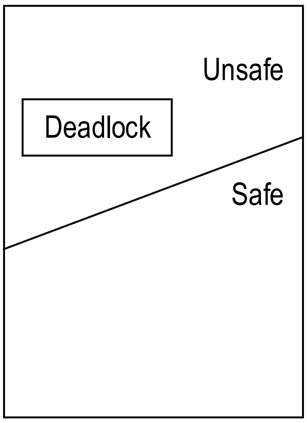


Figure 5.4: Safe, unsafe and deadlock state spaces.

If a system is in a safe state it can stay away from an unsafe state and thus avoid deadlock. On the other hand, if a system is in an unsafe state, deadlocks cannot be avoided.

Illustration: A system has 12 instances of a resource type and 3 processes using these resources. The maximum requirements for the resource by the processes and their current allocation at an instance say t0 is as shown below:

Process	Maximum	Current
P0	10	5
P1	4	2
P3	9	2

At the instant t_0, the system is in a safe state and one safe sequence is < P_1, P_0, P_2 >. This is true because of the following facts:

Out of 12 instances of the resource, 9 are currently allocated and 3 are free.

P_1 needs only 2 more, its maximum being 4, can be allotted 2.

Now only 1 instance of the resource is free.

When P_1 terminates, 5 instances of the resource will be free.

P_0 needs only 5 more, its maximum being 10, can be allotted 5.

Now resource is not free.

Once P_0 terminates, 10 instances of the resource will be free.

P_3 needs only 7 more, its maximum being 9, can be allotted 7.

Now 3 instances of the resource are free.

When P_3 terminates, all 12 instances of the resource will be free.

Thus the sequence < P_1, P_0, P_3 > is a safe sequence and the system is in a safe state. Let us now consider the following scenario at an instant t_1. In addition to the allocation shown in the table above, P_2 requests for 1 more instance of the resource and the allocation is made. At the instance t_1, a safe sequence cannot be found as shown below:

Out of 12 instances of the resource, 10 are currently allocated and 2 are free.

P_1 needs only 2 more, its maximum being 4, can be allotted 2.

Now resource is not free.

Once P_1 terminates, 4 instances of the resource will be free.

P_0 needs 5 more while P_2 needs 6 more.

Since both P_0 and P_2 cannot be granted resources, they wait.

The result is a deadlock.

Thus the system has gone from a safe state at time instant t_0 into an unsafe state at an instant t_1. The extra resource that was granted to P_2 at the instant t_1 was a mistake. P_2 should have waited till other processes finished and released their resources.

Since resources available should not be allocated right away as the system may enter an unsafe state, resource utilization is low if deadlock avoidance algorithms are used.

5.5.2 RESOURCE ALLOCATION GRAPH ALGORITHM

A resource allocation graph could be used to avoid deadlocks. If a resource allocation graph does not have a cycle, then the system is not in deadlock. But if there is a cycle then the system may be in a deadlock. If the resource allocation graph shows only resources that have only a single instance, then a cycle does imply a deadlock. An algorithm for avoiding deadlocks where resources have single instances in a resource allocation graph is as described below.

The resource allocation graph has request edges and assignment edges. Let there be another kind of edge called a claim edge. A directed edge $P_i \rightarrow R_j$ indicates that P_i may request for the resource R_j some time later. In a resource allocation graph a dashed line represents a claim edge. Later when a process makes an actual request for a resource, the corresponding claim edge is converted to a request edge $P_i \rightarrow R_j$. Similarly when a process releases a resource after use, the assignment edge $R_j \rightarrow P_i$ is reconverted to a claim edge $P_i \rightarrow R_j$. Thus a process must be associated with all its claim edges before it starts executing.

If a process Pi requests for a resource R_j, then the claim edge $P_i \rightarrow R_j$ is first converted to a request edge $P_i \rightarrow R_j$. The request of P_i can be granted only if the request edge when converted to an assignment edge does not result in a cycle.

If no cycle exists, the system is in a safe state and requests can be granted. If not the system is in an unsafe state and hence in a deadlock. In such a case, requests should not be granted. This is illustrated below (Figure 5.5a, 5.5b).

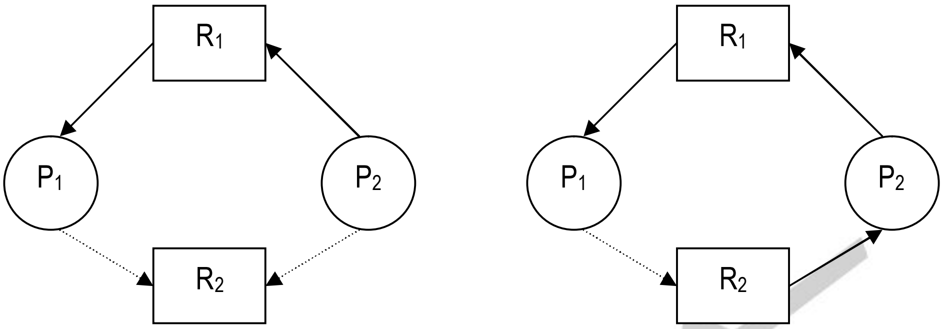


Figure 5.5: Resource allocation graph showing safe and deadlock states.

Consider the resource allocation graph shown on the left above. Resource R_2 is currently free. Allocation of R_2 to P_2 on request will result in a cycle as shown on the right. Therefore the system will be in an unsafe state. In this situation if P_1 requests for R_2, then a deadlock occurs.

5.5.3 BANKER'S ALGORITHM

The resource allocation graph algorithm is not applicable where resources have multiple instances. In such a case Banker's algorithm is used.

A new process entering the system must make known a priori the maximum instances of each resource that it needs subject to the maximum available for each type. As execution proceeds and requests are made, the system checks to see if the allocation of the requested resources ensures a safe state. If so only are the allocations made, else processes must wait for resources.

The following are the data structures maintained to implement the Banker's algorithm:

1. n: Number of processes in the system.
2. m: Number of resource types in the system.
3. Available: is a vector of length m. Each entry in this vector gives maximum instances of a resource type that are available at the instant. Available[j] = k means to say there are k instances of the jth resource type R_j.

4. Max: is a demand vector of size n x m. It defines the maximum needs of each resource by the process. Max[i][j] = k says the ith process P_i can request for atmost k instances of the jth resource type R_j.

5. Allocation: is a n x m vector which at any instant defines the number of resources of each type currently allocated to each of the m processes. If Allocation[i][j] = k then ith process P_i is currently holding k instances of the jth resource type R_j.

6. Need: is also a n x m vector which gives the remaining needs of the processes. Need[i][j] = k means the ith process P_i still needs k more instances of the jth resource type R_j. Thus Need[i][j] = Max[i][j] – Allocation[i][j].

5.5.3.1 SAFETY ALGORITHM

Using the above defined data structures, the Banker's algorithm to find out if a system is in a safe state or not is described below:

```
1.    Define a vector Work of length m and a vector Finish of length n.
2.    Initialize Work = Available and Finish[i] = false for i = 1, 2, ....., n.
3.    Find an i such that
          a.    Finish[i] = false and
          b.    $Need_i$ <= Work ($Need_i$ represents the ith row of the vector Need).
      If such an i does not exist , go to step 5.
4.    Work = Work + $Allocation_i$
      Finish[i] = true
      Go to step 3.
5.    If finish[i] = true for all i, then the system is in a safe state.
```

5.5.3.2 RESOURCE-REQUEST ALGORITHM

Let $Request_i$ be the vector representing the requests from a process P_i. Requesti[j] = k shows that process P_i wants k instances of the resource type R_j. The following is the algorithm to find out if a request by a process can immediately be granted:

1. If $Request_i$ <= $Need_i$, go to step 2.
   else Error "request of P_i exceeds Max_i".
2. If $Request_i$ <= $Available_i$, go to step 3.
   else P_i must wait for resources to be released.
3. An assumed allocation is made as follows:
   Available = Available – $Request_i$
   $Allocation_i$ = $Allocation_i$ + $Request_i$
   $Need_i$ = $Need_i$ – $Request_i$

If the resulting state is safe, then process P_i is allocated the resources and the above changes are made permanent. If the new state is unsafe, then P_i must wait and the old status of the data structures is restored.

Illustration: n = 5 < P_0, P_1, P_2, P_3, P_4 >
M = 3 < A, B, C >
Initially Available = < 10, 5, 7 >
At an instant t0, the data structures have the following values:

	Allocation	Max	Available	Need
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

To find a safe sequence and to prove that the system is in a safe state, use the safety algorithm as follows:

Step	Work	Finish	Safe sequence
0	3 3 2	F F F F F	< >
1	5 3 2	F T F F F	< P_1 >
2	7 4 3	F T F T F	< P_1, P_3 >
3	7 4 5	F T F T T	< P_1, P_3, P_4 >
4	7 5 5	T T F T T	< P_1, P_3, P_4, P_0 >
5	10 5 7	T T T T T	< P_1, P_3, P_4, P_0, P_2 >

Now at an instant t_1, Request$_1$ = < 1, 0, 2 >. To actually allocate the requested resources, use the request-resource algorithm as follows:

Request$_1$ < Need$_1$ and Request$_1$ < Available so the request can be considered. If the request is fulfilled, then the new the values in the data structures are as follows:

	Allocation	Max	Available	Need
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	2 3 0	7 4 3
P_1	3 0 2	3 2 2		0 2 0
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

Use the safety algorithm to see if the resulting state is safe:

Step	Work	Finish	Safe sequence
0	2 3 0	F F F F F	< >
1	5 3 2	F T F F F	< P_1 >
2	7 4 3	F T F T F	< P_1, P_3 >
3	7 4 5	F T F T T	< P_1, P_3, P_4 >
4	7 5 5	T T F T T	< P_1, P_3, P_4, P_0 >
5	10 5 7	T T T T T	< P_1, P_3, P_4, P_0, P_2 >

Since the resulting state is safe, request by P_1 can be granted.

Now at an instant t_2 Request$_4$ = < 3, 3, 0 >. But since Request$_4$ > Available, the request cannot be granted. Also Request$_0$ = < 0, 2, 0> at t_2 cannot be granted since the resulting state is unsafe as shown below:

	Allocation	Max	Available	Need
	A B C	A B C	A B C	A B C
P_0	0 3 0	7 5 3	2 1 0	7 2 3
P_1	3 0 2	3 2 2		0 2 0
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

Using the safety algorithm, the resulting state is unsafe since Finish is false for all values of i and we cannot find a safe sequence.

Step	Work	Finish	Safe sequence
0	2 1 0	F F F F F	< >

5.6 DEADLOCK DETECTION

If the system does not ensure that a deadlock cannot be prevented or a deadlock cannot be avoided, then a deadlock may occur. In case a deadlock occurs the system must

1. detect the deadlock
2. recover from the deadlock

5.6.1 SINGLE INSTANCE OF A RESOURCE

If the system has resources, all of which have only single instances, then a deadlock detection algorithm, which uses a variant of the resource allocation graph, can be used. The graph used in this case is called a wait-for graph.

The wait-for graph is a directed graph having vertices and edges. The vertices represent processes and directed edges are present between two processes one of which is waiting for a resource held by the other. Two edges $P_i \rightarrow R_q$ and $R_q \rightarrow P_j$ in the resource allocation graph are replaced by one edge $P_i \rightarrow P_j$ in the wait-for graph. Thus the wait-for graph is obtained by removing vertices representing resources and then collapsing the corresponding edges in a resource allocation graph. An Illustration is shown below (Figure 5.6).

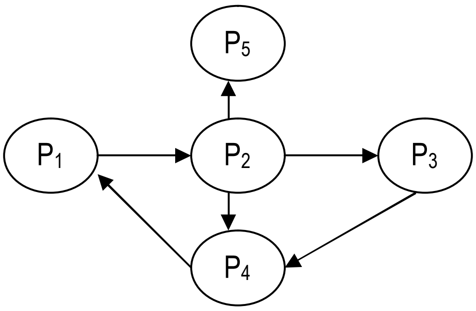


Figure 5.6: Wait-for graph

As in the previous case, a cycle in a wait-for graph indicates a deadlock. Therefore the system maintains a wait-for graph and periodically invokes an algorithm to check for a cycle in the wait-for graph.

5.6.2 MULTIPLE INSTANCES OF A RESOURCE

A wait-for graph is not applicable for detecting deadlocks where there exist multiple instances of resources. This is because there is a situation where a cycle may or may not indicate a deadlock. If this is so then a decision cannot be made. In situations where there are multiple instances of resources, an algorithm similar to Banker's algorithm for deadlock avoidance is used.

Data structures used are similar to those used in Banker's algorithm and are given below:

1. n: Number of processes in the system.
2. m: Number of resource types in the system.
3. Available: is a vector of length m. Each entry in this vector gives maximum instances of a resource type that are available at the instant.
4. Allocation: is a n x m vector which at any instant defines the number of resources of each type currently allocated to each of the m processes.
5. Request: is also a n x m vector defining the current requests of each process. Request[i][j] = k means the ith process P_i is requesting for k instances of the jth resource type R_j.

5.6.2.1 ALGORITHM

```
1. Define a vector Work of length m and a vector Finish of length n.
2. Initialize      Work = Available and
                   For I = 1, 2, ….., n
                           If $Allocation_i$ != 0
                                   Finish[i] = false
                           Else
                                   Finish[i] = true
3. Find an i such that
```

a. Finish[i] = false and

b. $Request_i$ <= Work

If such an i does not exist , go to step 5.

4. Work = Work + $Allocation_i$

   Finish[i] = true

   Go to step 3.

5. If finish[i] = true for all i, then the system is not in deadlock.

   Else the system is in deadlock with all processes corresponding to Finish[i] = false being deadlocked.

Illustration: n = 5 < P_0, P_1, P_2, P_3, P_4 >

M = 3 < A, B, C >

Initially Available = < 7, 2, 6 >

At an instant t0, the data structures have the following values:

	Allocation	Request	Available
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

To prove that the system is not deadlocked, use the above algorithm as follows:

Step	Work	Finish	Safe sequence
0	0 0 0	F F F F F	< >
1	0 1 0	T F F F F	< P_0 >
2	3 1 3	T F T F F	< P_0, P_2 >
3	5 2 4	T F T T F	< P_0, P_2, P_3 >
4	5 2 6	T F T T T	< P_0, P_2, P_3, P_4 >
5	7 2 6	T T T T T	< P_0, P_2, P_3, P_4, P_1 >

Now at an instant t_1, $Request_2$ = < 0, 0, 1 > and the new values in the data structures are as follows:

	Allocation	Request	Available
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 1	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

To prove that the system is deadlocked, use the above algorithm as follows:

Step	Work	Finish	Safe sequence
0	0 0 0	F F F F F	< >
1	0 1 0	T F F F F	< P_0 >

The system is in deadlock with processes P_1, P_2, P_3, and P_4 deadlocked.

5.6.2.2 WHEN TO INVOKE?

The deadlock detection algorithm takes m x n^2 operations to detect whether a system is in deadlock. How often should the algorithm be invoked? This depends on the following two main factors:

1. Frequency of occurrence of deadlocks
2. Number of processes involved when it occurs

If deadlocks are known to occur frequently, then the detection algorithm has to be invoked frequently because during the period of deadlock, resources are idle and more and more processes wait for idle resources.

Deadlocks occur only when requests from processes cannot be immediately granted. Based on this reasoning, the detection algorithm can be invoked only when a request cannot be immediately granted. If this is so, then the process causing the deadlock and also all the deadlocked processes can also be identified.

But invoking the algorithm an every request is a clear overhead as it consumes CPU time. Better would be to invoke the algorithm periodically at regular less frequent intervals. One criterion could be when the CPU utilization drops below a threshold. The drawbacks in this case are that deadlocks may go unnoticed for some time and the process that caused the deadlock will not be known.

5.6.2.3 RECOVERY FROM DEADLOCK

Once a deadlock has been detected, it must be broken. Breaking a deadlock may be manual by the operator when informed of the deadlock or automatically by the system. There exist two options for breaking deadlocks:

1. abort one or more processes to break the circular-wait condition causing deadlock
2. preempting resources from one or more processes which are deadlocked.

In the first option, one or more processes involved in deadlock could be terminated to break the deadlock. Then either abort all processes or abort one process at a time till deadlock is broken. The first case guarantees that the deadlock will be broken. But processes that have executed for a long time will have to restart all over again. The second case is better but has considerable overhead as detection algorithm has to be invoked after terminating every process. Also choosing a process that becomes a victim for termination is based on many factors like

- priority of the processes
- length of time each process has executed and how much more it needs for completion
- type of resources and the instances of each that the processes use
- need for additional resources to complete
- nature of the processes, whether iterative or batch

Based on these and many more factors, a process that incurs minimum cost on termination becomes a victim.

In the second option some resources are preempted from some processes and given to other processes until the deadlock cycle is broken. Selecting the victim whose resources can be preempted is again based on the minimum cost criteria. Parameters such as number of resources a process is holding and the amount of these resources used thus far by the process are used to select a victim. When resources are preempted, the process holding the resource cannot continue. A simple solution is to abort the process also. Better still is to rollback the process to a safe state to restart later. To determine this safe state, more information about running processes is required which is again an overhead. Also starvation may occur when a victim is selected for preemption, the reason being resources from the same process may again and again be preempted. As a result the process starves for want of resources. Ensuring that a process can be a victim only a finite number of times by having this information as one of the parameters for victim selection could prevent starvation.

Prevention, avoidance and detection are the three basic approaches to handle deadlocks. But they do not encompass all the problems encountered. Thus a combined approach of all the three basic approaches is used.

5.7 SUMMARY

This problem has highlighted the problem associated with multiprogramming. Since many processes compete for a finite set of resources, there is always a possibility that requested resources are not readily available. This makes processes wait. When there is a set of processes where each process in the set waits on another from the same set for release of a wanted resource, then a deadlock has occurred. We have the analyzed the four necessary conditions for deadlocks. Deadlock handling schemes that either prevent, avoid or detect deadlocks were discussed.

5.8 EXERCISE

1. Explain the condition under which deadlock occurs.
2. Explain banker's algorithm to avoid deadlock in the allocation of system resources.
3. Compare the relative merits and demerits of using prevention, avoidance and detection as strategies for dealing with deadlocks.

5.9 ACTIVITY

Find out the strategies for handling deadlocks in the operating systems noted.

CHAPTER 5

DEADLOCKS

Several processes compete for a finite set of resources in a multiprogrammed environment. A process requests for resources that may not be readily available at the time of the request. In such a case the process goes into a wait state. It may so happen that this process may never change state because the requested resources are held by other processes which themselves are waiting for additional resources and hence in a wait state. This situation is called a deadlock.

5.1 SYSTEM MODEL

The number of resources in a system is always finite. But the number of competing processes is many. Resources are of several types, each type having identical instances of the resource. Examples for resources could be memory space, CPU time, files, I/O devices and so on. If a system has 2 CPUs that are equivalent, then the resource type CPU time has 2 instances. If they are not equivalent, then each CPU is of a different resource type. Similarly the system may have 2 dot matrix printers and 1 line printer. Here the resource type of dot matrix printer has 2 instances where as there is a single instance of type line printer.

A process requests for resources, uses them if granted and then releases the resources for others to use. It goes without saying that the number of resources requested shall not exceed the total of each type available in the system. If a request for a resource cannot be granted immediately then the process requesting the resource goes into a wait state and joins the wait queue for the resource.

A set of processes is in a state of deadlock if every process in the set is in some wait queue of a resource and is waiting for an event (release resource) to occur that can be caused by another process in the set.

For example, there are 2 resources, 1 printer and 1 tape drive. Process P1 is allocated tape drive and P2 is allocated printer. Now if P1 requests for printer and P2 for tape drive, a deadlock occurs.

5.2 DEADLOCK CHARACTERIZATION

5.2.1 NECESSARY CONDITIONS FOR DEADLOCKS

A deadlock occurs in a system if the following four conditions hold simultaneously:

1. Mutual exclusion: At least one of the resources is non-sharable that is only one process at a time can use the resource.
2. Hold and wait: A process exists that is holding on to at least one resource and waiting for an additional resource held by another process.
3. No preemption: Resources cannot be preempted that is a resource is released only by the process that is holding it.
4. Circular wait: There exist a set of processes P_0, P_1, ….., P_n of waiting processes such that P_0 is waiting for a resource held by P_1, P_1 is waiting for a resource held by P_2, ….., P_{n-1} is waiting for a resource held P_n and P_n is in turn waiting for a resource held by P_0.

5.2.2 RESOURCE ALLOCATION GRAPH

Deadlocks can be described by a resource allocation graph. The resource allocation graph is a directed graph consisting of vertices and directed edges. The vertex set is partitioned into two types, a subset representing processes and another subset representing resources. Pictorially, the resources are represented by rectangles with dots within, each dot representing an instance of the resource and circles represent processes.

A directed edge from a process to a resource ($P_i \rightarrow R_j$) signifies a request from a process P_i for an instance of the resource R_j and P_i is waiting for R_j. A directed edge from a resource to a process ($R_j \rightarrow P_i$) indicates that an instance of the resource R_j has been allotted to process P_i. Thus a resource allocation graph consists of vertices which include resources and processes and directed edges which consist of request edges and assignment edges. A request edge is introduced into the graph when a process requests for a resource. This edge is converted into an assignment edge when the resource is granted. When the process releases the resource, the assignment edge is deleted. Consider the following system:

There are 3 processes P_1, P_2 and P_3.

Resources R_1, R_2, R_3 and R_4 have instances 1, 2, 1, and 3 respectively.

P_1 is holding R_2 and waiting for R_1.

P_2 is holding R_1, R_2 and is waiting for R_3.

P_3 is holding R_3.

The resource allocation gragh for a system in the above situation is as shown below (Figure 5.1).

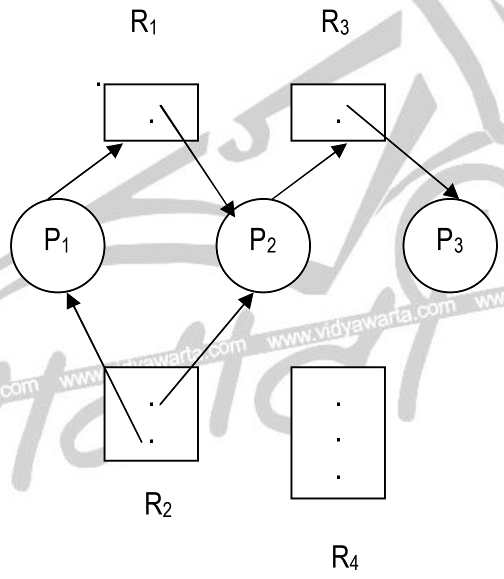


Figure 5.1: Resource allocation graph

If a resource allocation graph has no cycles (a closed loop in the direction of the edges), then the system is not in a state of deadlock. If on the other hand, there are cycles, then a deadlock may exist. If there are only single instances of each resource type, then a cycle in a resource allocation graph is a necessary and sufficient condition for existence of a deadlock (Figure 5.2). Here two cycles exist:

$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

Processes P_0, P_1 and P_3 are deadlocked and are in a circular wait. P_2 is waiting for R_3 held by P_3. P_3 is waiting for P_1 or P_2 to release R_2. So also P_1 is waiting for P_2 to release R_1.

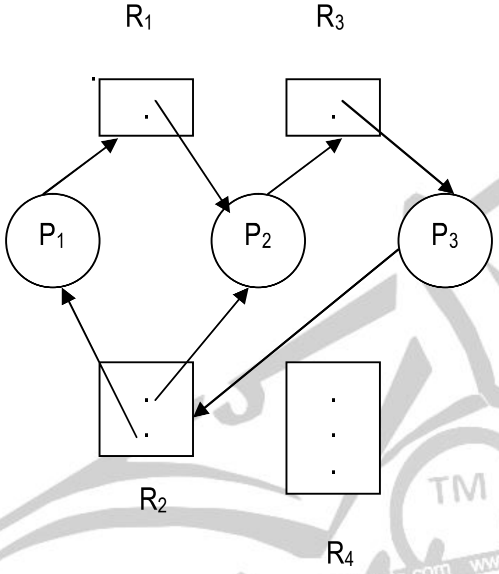


Figure 5.2: Resource allocation graph with deadlock

If there are multiple instances of resources types, then a cycle does not necessarily imply a deadlock. Here a cycle is a necessary condition but not a sufficient condition for the existence of a deadlock (Figure 5.3). Here also there is a cycle:

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

The cycle above does not imply a deadlock because an instance of R_1 released by P_2 could be assigned to P_1 or an instance of R_2 released by P_4 could be assigned to P_3 there by breaking the cycle.

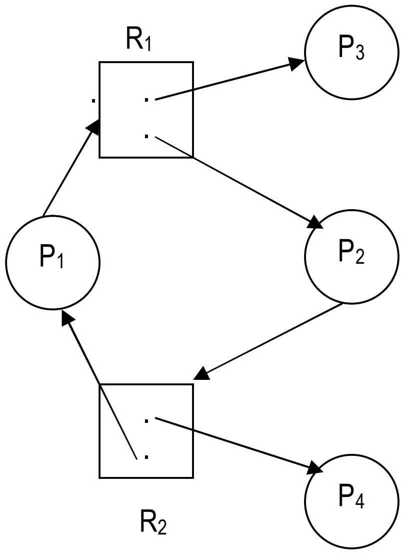


Figure 5.3: Resource allocation graph with a cycle but no deadlock

5.3 DEADLOCK HANDLING

Different methods to deal with deadlocks include methods to ensure that the system will never enter into a state of deadlock, methods that allow the system to enter into a deadlock and then recover or to just ignore the problem of deadlocks.

To ensure that deadlocks never occur, deadlock prevention / avoidance schemes are used. The four necessary conditions for deadlocks to occur are mutual exclusion, hod and wait, no preemption and circular wait. Deadlock prevention ensures that at least one of the four necessary conditions for deadlocks do not hold. To do this the scheme enforces constraints on requests for resources. Dead lock avoidance scheme requires the operating system to know in advance, the resources needed by a process for its entire lifetime. Based on this a priori information, the process making a request is either made to wait or not to wait in case the requested resource is not readily available.

If none of the above two schemes are used, then deadlocks may occur. In such a case, an algorithm to recover from the state of deadlock is used.

If the problem of deadlocks is ignored totally that is to say the system does not ensure that a deadlock does not occur and also does not provide for recovery from deadlock and such a situation arises, then there is no way out of the deadlock. Eventually the system may crash because more and more processes request for resources and enter into deadlock.

5.4 DEADLOCK PREVENTION

The four necessary conditions for deadlocks to occur are mutual exclusion, hod and wait, no preemption and circular wait. If any one of the above four conditions does not hold, then deadlocks will not occur. Thus prevention of deadlock is possible by ensuring that at least one of the four conditions cannot hold.

Mutual exclusion: Resources that can be shared are never involved in a deadlock because such resources can always be granted simultaneously access by processes. Hence processes requesting for such a sharable resource will never have to wait. Examples of such resources include read-only files. Mutual exclusion must therefor hold for non-sharable resources. But it is not always possible to prevent deadlocks by denying mutual exclusion condition because some resources are by nature non-sharable, for example printers.

Hold and wait: To avoid hold and wait, the system must ensure that a process that requests for a resource does not hold on to another. There can be two approaches to this scheme:

1. a process requests for and gets allocated all the resources it uses before execution begins.
2. a process can request for a resource only when it does not hold on to any other.

Algorithms based on these approaches have poor resource utilization. This is because resources get locked with processes much earlier than they are actually used and hence not available for others to use as in the first approach. The second approach seems to applicable only when there is assurance about reusability of data and code on the released resources. The algorithms also suffer from starvation since popular resources may never be freely available.

No preemption: This condition states that resources allocated to processes cannot be preempted. To ensure that this condition does not hold, resources could be preempted. When a process requests for a resource, it is allocated the resource if it is available. If it is not, than a check is made to see if the process holding the wanted resource is also waiting for additional resources. If so the wanted resource is preempted from the waiting process and allotted to the requesting process. If both the above is not true that is the resource is neither available nor held by a waiting process, then the requesting process waits. During its waiting period, some of its resources could also be preempted in which case the process will be restarted only when all the new and the preempted resources are allocated to it.

Another alternative approach could be as follows: If a process requests for a resource which is not available immediately, then all other resources it currently holds are preempted. The process restarts only when the new and the preempted resources are allocated to it as in the previous case.

Resources can be preempted only if their current status can be saved so that processes could be restarted later by restoring the previous states. Example CPU memory and main memory. But resources such as printers cannot be preempted, as their states cannot be saved for restoration later.

Circular wait: Resource types need to be ordered and processes requesting for resources will do so in increasing order of enumeration. Each resource type is mapped to a unique integer that allows resources to be compared and to find out the precedence order for the resources. Thus F: R → N is a 1:1 function that maps resources to numbers. For example:

F (tape drive) = 1, F (disk drive) = 5, F (printer) = 10.

To ensure that deadlocks do not occur, each process can request for resources only in increasing order of these numbers. A process to start with in the very first instance can request for any resource say R_i. There after it can request for a resource R_j if and only if $F(R_j)$ is greater than $F(R_i)$. Alternately, if $F(R_j)$ is less than $F(R_i)$, then R_j can be allocated to the process if and only if the process releases R_i.

The mapping function F should be so defined that resources get numbers in the usual order of usage.

5.5 DEADLOCK AVOIDANCE

Deadlock prevention algorithms ensure that at least one of the four necessary conditions for deadlocks namely mutual exclusion, hold and wait, no preemption and circular wait do not hold. The disadvantage with prevention algorithms is poor resource utilization and thus reduced system throughput.

An alternate method is to avoid deadlocks. In this case additional a priori information about the usage of resources by processes is required. This information helps to decide on whether a process should wait for a resource or not. Decision about a request is based on all the resources available, resources allocated to processes, future requests and releases by processes.

A deadlock avoidance algorithm requires each process to make known in advance the maximum number of resources of each type that it may need. Also known is the maximum number of resources of each type available. Using both the above a priori knowledge, deadlock avoidance algorithm ensures that a circular wait condition never occurs.

5.5.1 SAFE STATE

A system is said to be in a safe state if it can allocate resources upto the maximum available and is not in a state of deadlock. A safe sequence of processes always ensures a safe state. A sequence of processes < P_1, P_2, ….., P_n > is safe for the current allocation of resources to processes if resource requests from each P_i can be satisfied from the currently available resources and the resources held by all P_j where $j < i$.If the state is safe then P_i requesting for resources can wait till P_j's have completed. If such a safe sequence does not exist, then the system is in an unsafe state.

A safe state is not a deadlock state. Conversely a deadlock state is an unsafe state. But all unsafe states are not deadlock states as shown below (Figure 5.4).

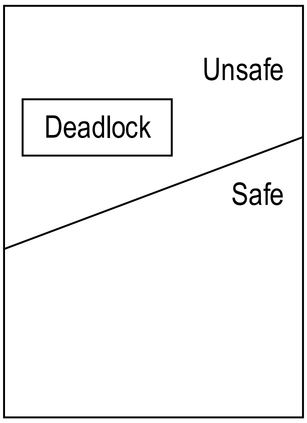


Figure 5.4: Safe, unsafe and deadlock state spaces.

If a system is in a safe state it can stay away from an unsafe state and thus avoid deadlock. On the other hand, if a system is in an unsafe state, deadlocks cannot be avoided.

Illustration: A system has 12 instances of a resource type and 3 processes using these resources. The maximum requirements for the resource by the processes and their current allocation at an instance say t0 is as shown below:

Process	Maximum	Current
P0	10	5
P1	4	2
P3	9	2

At the instant t_0, the system is in a safe state and one safe sequence is < P_1, P_0, P_2 >. This is true because of the following facts:

Out of 12 instances of the resource, 9 are currently allocated and 3 are free.

P_1 needs only 2 more, its maximum being 4, can be allotted 2.

Now only 1 instance of the resource is free.

When P_1 terminates, 5 instances of the resource will be free.

P_0 needs only 5 more, its maximum being 10, can be allotted 5.

Now resource is not free.

Once P_0 terminates, 10 instances of the resource will be free.

P_3 needs only 7 more, its maximum being 9, can be allotted 7.

Now 3 instances of the resource are free.

When P_3 terminates, all 12 instances of the resource will be free.

Thus the sequence < P_1, P_0, P_3 > is a safe sequence and the system is in a safe state. Let us now consider the following scenario at an instant t_1. In addition to the allocation shown in the table above, P_2 requests for 1 more instance of the resource and the allocation is made. At the instance t_1, a safe sequence cannot be found as shown below:

Out of 12 instances of the resource, 10 are currently allocated and 2 are free.

P_1 needs only 2 more, its maximum being 4, can be allotted 2.

Now resource is not free.

Once P_1 terminates, 4 instances of the resource will be free.

P_0 needs 5 more while P_2 needs 6 more.

Since both P_0 and P_2 cannot be granted resources, they wait.

The result is a deadlock.

Thus the system has gone from a safe state at time instant t_0 into an unsafe state at an instant t_1. The extra resource that was granted to P_2 at the instant t_1 was a mistake. P_2 should have waited till other processes finished and released their resources.

Since resources available should not be allocated right away as the system may enter an unsafe state, resource utilization is low if deadlock avoidance algorithms are used.

5.5.2 RESOURCE ALLOCATION GRAPH ALGORITHM

A resource allocation graph could be used to avoid deadlocks. If a resource allocation graph does not have a cycle, then the system is not in deadlock. But if there is a cycle then the system may be in a deadlock. If the resource allocation graph shows only resources that have only a single instance, then a cycle does imply a deadlock. An algorithm for avoiding deadlocks where resources have single instances in a resource allocation graph is as described below.

The resource allocation graph has request edges and assignment edges. Let there be another kind of edge called a claim edge. A directed edge $P_i \rightarrow R_j$ indicates that P_i may request for the resource R_j some time later. In a resource allocation graph a dashed line represents a claim edge. Later when a process makes an actual request for a resource, the corresponding claim edge is converted to a request edge $P_i \rightarrow R_j$. Similarly when a process releases a resource after use, the assignment edge $R_j \rightarrow P_i$ is reconverted to a claim edge $P_i \rightarrow R_j$. Thus a process must be associated with all its claim edges before it starts executing.

If a process Pi requests for a resource R_j, then the claim edge $P_i \rightarrow R_j$ is first converted to a request edge $P_i \rightarrow R_j$. The request of P_i can be granted only if the request edge when converted to an assignment edge does not result in a cycle.

If no cycle exists, the system is in a safe state and requests can be granted. If not the system is in an unsafe state and hence in a deadlock. In such a case, requests should not be granted. This is illustrated below (Figure 5.5a, 5.5b).

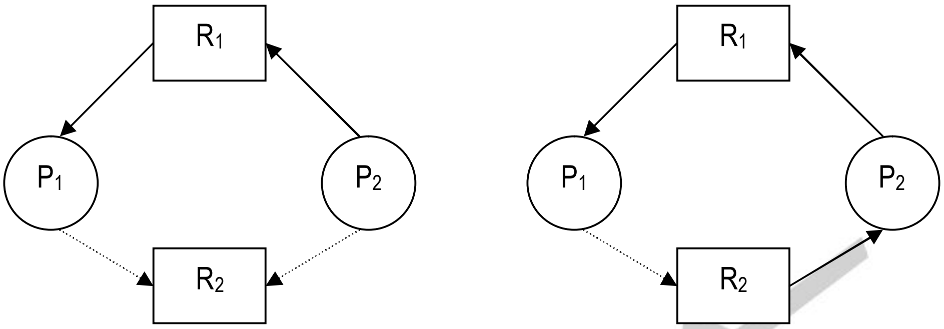


Figure 5.5: Resource allocation graph showing safe and deadlock states.

Consider the resource allocation graph shown on the left above. Resource R_2 is currently free. Allocation of R_2 to P_2 on request will result in a cycle as shown on the right. Therefore the system will be in an unsafe state. In this situation if P_1 requests for R_2, then a deadlock occurs.

5.5.3 BANKER'S ALGORITHM

The resource allocation graph algorithm is not applicable where resources have multiple instances. In such a case Banker's algorithm is used.

A new process entering the system must make known a priori the maximum instances of each resource that it needs subject to the maximum available for each type. As execution proceeds and requests are made, the system checks to see if the allocation of the requested resources ensures a safe state. If so only are the allocations made, else processes must wait for resources.

The following are the data structures maintained to implement the Banker's algorithm:

1. n: Number of processes in the system.
2. m: Number of resource types in the system.
3. Available: is a vector of length m. Each entry in this vector gives maximum instances of a resource type that are available at the instant. Available[j] = k means to say there are k instances of the jth resource type R_j.

4. Max: is a demand vector of size n x m. It defines the maximum needs of each resource by the process. Max[i][j] = k says the ith process P_i can request for atmost k instances of the jth resource type R_j.
5. Allocation: is a n x m vector which at any instant defines the number of resources of each type currently allocated to each of the m processes. If Allocation[i][j] = k then ith process P_i is currently holding k instances of the jth resource type R_j.
6. Need: is also a n x m vector which gives the remaining needs of the processes. Need[i][j] = k means the ith process P_i still needs k more instances of the jth resource type R_j. Thus Need[i][j] = Max[i][j] – Allocation[i][j].

5.5.3.1 SAFETY ALGORITHM

Using the above defined data structures, the Banker's algorithm to find out if a system is in a safe state or not is described below:

1. Define a vector Work of length m and a vector Finish of length n.
2. Initialize Work = Available and Finish[i] = false for i = 1, 2, ..., n.
3. Find an i such that
   a. Finish[i] = false and
   b. $Need_i$ <= Work ($Need_i$ represents the ith row of the vector Need).

   If such an i does not exist , go to step 5.
4. Work = Work + $Allocation_i$

   Finish[i] = true

   Go to step 3.
5. If finish[i] = true for all i, then the system is in a safe state.

5.5.3.2 RESOURCE-REQUEST ALGORITHM

Let $Request_i$ be the vector representing the requests from a process P_i. Requesti[j] = k shows that process P_i wants k instances of the resource type R_j. The following is the algorithm to find out if a request by a process can immediately be granted:

1. If Request_i <= Need_i, go to step 2.
   else Error "request of P_i exceeds Max_i".
2. If Request_i <= Available_i, go to step 3.
   else P_i must wait for resources to be released.
3. An assumed allocation is made as follows:
   Available = Available – Request_i
   Allocation_i = Allocation_i + Request_i
   Need_i = Need_i – Request_i

If the resulting state is safe, then process P_i is allocated the resources and the above changes are made permanent. If the new state is unsafe, then P_i must wait and the old status of the data structures is restored.

Illustration: n = 5 < P_0, P_1, P_2, P_3, P_4 >
M = 3 < A, B, C >
Initially Available = < 10, 5, 7 >
At an instant t0, the data structures have the following values:

	Allocation	Max	Available	Need
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

To find a safe sequence and to prove that the system is in a safe state, use the safety algorithm as follows:

Step	Work	Finish	Safe sequence
0	3 3 2	F F F F F	< >
1	5 3 2	F T F F F	< P_1 >
2	7 4 3	F T F T F	< P_1, P_3 >
3	7 4 5	F T F T T	< P_1, P_3, P_4 >
4	7 5 5	T T F T T	< P_1, P_3, P_4, P_0 >
5	10 5 7	T T T T T	< P_1, P_3, P_4, P_0, P_2 >

Now at an instant t_1, Request$_1$ = < 1, 0, 2 >. To actually allocate the requested resources, use the request-resource algorithm as follows:

Request$_1$ < Need$_1$ and Request$_1$ < Available so the request can be considered. If the request is fulfilled, then the new the values in the data structures are as follows:

	Allocation	Max	Available	Need
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	2 3 0	7 4 3
P_1	3 0 2	3 2 2		0 2 0
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

Use the safety algorithm to see if the resulting state is safe:

Step	Work	Finish	Safe sequence
0	2 3 0	F F F F F	< >
1	5 3 2	F T F F F	< P_1 >
2	7 4 3	F T F T F	< P_1, P_3 >
3	7 4 5	F T F T T	< P_1, P_3, P_4 >
4	7 5 5	T T F T T	< P_1, P_3, P_4, P_0 >
5	10 5 7	T T T T T	< P_1, P_3, P_4, P_0, P_2 >

Since the resulting state is safe, request by P_1 can be granted.

Now at an instant t_2 Request$_4$ = < 3, 3, 0 >. But since Request$_4$ > Available, the request cannot be granted. Also Request$_0$ = < 0, 2, 0> at t_2 cannot be granted since the resulting state is unsafe as shown below:

	Allocation	Max	Available	Need
	A B C	A B C	A B C	A B C
P_0	0 3 0	7 5 3	2 1 0	7 2 3
P_1	3 0 2	3 2 2		0 2 0
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

Using the safety algorithm, the resulting state is unsafe since Finish is false for all values of i and we cannot find a safe sequence.

Step	Work	Finish	Safe sequence
0	2 1 0	F F F F F	< >

5.6 DEADLOCK DETECTION

If the system does not ensure that a deadlock cannot be prevented or a deadlock cannot be avoided, then a deadlock may occur. In case a deadlock occurs the system must

1. detect the deadlock
2. recover from the deadlock

5.6.1 SINGLE INSTANCE OF A RESOURCE

If the system has resources, all of which have only single instances, then a deadlock detection algorithm, which uses a variant of the resource allocation graph, can be used. The graph used in this case is called a wait-for graph.

The wait-for graph is a directed graph having vertices and edges. The vertices represent processes and directed edges are present between two processes one of which is waiting for a resource held by the other. Two edges $P_i \rightarrow R_q$ and $R_q \rightarrow P_j$ in the resource allocation graph are replaced by one edge $P_i \rightarrow P_j$ in the wait-for graph. Thus the wait-for graph is obtained by removing vertices representing resources and then collapsing the corresponding edges in a resource allocation graph. An Illustration is shown below (Figure 5.6).

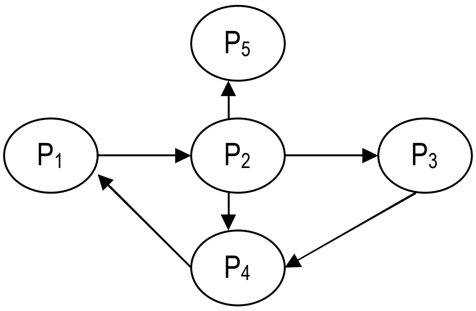


Figure 5.6: Wait-for graph

As in the previous case, a cycle in a wait-for graph indicates a deadlock. Therefore the system maintains a wait-for graph and periodically invokes an algorithm to check for a cycle in the wait-for graph.

5.6.2 MULTIPLE INSTANCES OF A RESOURCE

A wait-for graph is not applicable for detecting deadlocks where there exist multiple instances of resources. This is because there is a situation where a cycle may or may not indicate a deadlock. If this is so then a decision cannot be made. In situations where there are multiple instances of resources, an algorithm similar to Banker's algorithm for deadlock avoidance is used.

Data structures used are similar to those used in Banker's algorithm and are given below:

1. n: Number of processes in the system.
2. m: Number of resource types in the system.
3. Available: is a vector of length m. Each entry in this vector gives maximum instances of a resource type that are available at the instant.
4. Allocation: is a n x m vector which at any instant defines the number of resources of each type currently allocated to each of the m processes.
5. Request: is also a n x m vector defining the current requests of each process. Request[i][j] = k means the ith process P_i is requesting for k instances of the jth resource type R_j.

5.6.2.1 ALGORITHM

```
1. Define a vector Work of length m and a vector Finish of length n.
2. Initialize    Work = Available and
                 For I = 1, 2, ….., n
                         If Allocation_i != 0
                                 Finish[i] = false
                         Else
                                 Finish[i] = true
3. Find an i such that
```

a. Finish[i] = false and

b. $Request_i$ <= Work

If such an i does not exist , go to step 5.

4. Work = Work + $Allocation_i$

   Finish[i] = true

   Go to step 3.

5. If finish[i] = true for all i, then the system is not in deadlock.

   Else the system is in deadlock with all processes corresponding to Finish[i] = false being deadlocked.

Illustration: n = 5 < P_0, P_1, P_2, P_3, P_4 >

M = 3 < A, B, C >

Initially Available = < 7, 2, 6 >

At an instant t0, the data structures have the following values:

	Allocation	Request	Available
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

To prove that the system is not deadlocked, use the above algorithm as follows:

Step	Work	Finish	Safe sequence
0	0 0 0	F F F F F	< >
1	0 1 0	T F F F F	< P_0 >
2	3 1 3	T F T F F	< P_0, P_2 >
3	5 2 4	T F T T F	< P_0, P_2, P_3 >
4	5 2 6	T F T T T	< P_0, P_2, P_3, P_4 >
5	7 2 6	T T T T T	< P_0, P_2, P_3, P_4, P_1 >

Now at an instant t_1, $Request_2$ = < 0, 0, 1 > and the new values in the data structures are as follows:

	Allocation	Request	Available
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 1	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

To prove that the system is deadlocked, use the above algorithm as follows:

Step	Work	Finish	Safe sequence
0	0 0 0	F F F F F	< >
1	0 1 0	T F F F F	< P_0 >

The system is in deadlock with processes P_1, P_2, P_3, and P_4 deadlocked.

5.6.2.2 WHEN TO INVOKE?

The deadlock detection algorithm takes m x n^2 operations to detect whether a system is in deadlock. How often should the algorithm be invoked? This depends on the following two main factors:

1. Frequency of occurrence of deadlocks
2. Number of processes involved when it occurs

If deadlocks are known to occur frequently, then the detection algorithm has to be invoked frequently because during the period of deadlock, resources are idle and more and more processes wait for idle resources.

Deadlocks occur only when requests from processes cannot be immediately granted. Based on this reasoning, the detection algorithm can be invoked only when a request cannot be immediately granted. If this is so, then the process causing the deadlock and also all the deadlocked processes can also be identified.

But invoking the algorithm an every request is a clear overhead as it consumes CPU time. Better would be to invoke the algorithm periodically at regular less frequent intervals. One criterion could be when the CPU utilization drops below a threshold. The drawbacks in this case are that deadlocks may go unnoticed for some time and the process that caused the deadlock will not be known.

5.6.2.3 RECOVERY FROM DEADLOCK

Once a deadlock has been detected, it must be broken. Breaking a deadlock may be manual by the operator when informed of the deadlock or automatically by the system. There exist two options for breaking deadlocks:

1. abort one or more processes to break the circular-wait condition causing deadlock
2. preempting resources from one or more processes which are deadlocked.

In the first option, one or more processes involved in deadlock could be terminated to break the deadlock. Then either abort all processes or abort one process at a time till deadlock is broken. The first case guarantees that the deadlock will be broken. But processes that have executed for a long time will have to restart all over again. The second case is better but has considerable overhead as detection algorithm has to be invoked after terminating every process. Also choosing a process that becomes a victim for termination is based on many factors like

- priority of the processes
- length of time each process has executed and how much more it needs for completion
- type of resources and the instances of each that the processes use
- need for additional resources to complete
- nature of the processes, whether iterative or batch

Based on these and many more factors, a process that incurs minimum cost on termination becomes a victim.

In the second option some resources are preempted from some processes and given to other processes until the deadlock cycle is broken. Selecting the victim whose resources can be preempted is again based on the minimum cost criteria. Parameters such as number of resources a process is holding and the amount of these resources used thus far by the process are used to select a victim. When resources are preempted, the process holding the resource cannot continue. A simple solution is to abort the process also. Better still is to rollback the process to a safe state to restart later. To determine this safe state, more information about running processes is required which is again an overhead. Also starvation may occur when a victim is selected for preemption, the reason being resources from the same process may again and again be preempted. As a result the process starves for want of resources. Ensuring that a process can be a victim only a finite number of times by having this information as one of the parameters for victim selection could prevent starvation.

Prevention, avoidance and detection are the three basic approaches to handle deadlocks. But they do not encompass all the problems encountered. Thus a combined approach of all the three basic approaches is used.

5.7 SUMMARY

This problem has highlighted the problem associated with multiprogramming. Since many processes compete for a finite set of resources, there is always a possibility that requested resources are not readily available. This makes processes wait. When there is a set of processes where each process in the set waits on another from the same set for release of a wanted resource, then a deadlock has occurred. We have the analyzed the four necessary conditions for deadlocks. Deadlock handling schemes that either prevent, avoid or detect deadlocks were discussed.

5.8 EXERCISE

1. Explain the condition under which deadlock occurs.
2. Explain banker’s algorithm to avoid deadlock in the allocation of system resources.
3. Compare the relative merits and demerits of using prevention, avoidance and detection as strategies for dealing with deadlocks.

5.9 ACTIVITY

Find out the strategies for handling deadlocks in the operating systems noted.

CHAPTER 6

MEMORY MANAGEMENT

The concept of CPU scheduling allows a set of processes to share the CPU there by increasing the utilization of the CPU. This set of processes needs to reside in memory. The memory is thus shared and the resource requires to be managed. Various memory management algorithms exist, each having its own advantages and disadvantages. The hardware design of the system plays an important role in the selection of an algorithm for a particular system. That means to say hardware support is essential to implement the memory management algorithm.

6.1 LOGICAL Vs PHYSICAL ADDRESS SPACE

An address generated by the CPU is referred to as a logical address. An address seen by the memory unit that is the address loaded into the memory address register (MAR) of the memory for a fetch or a store is referred to as a physical address. The logical address is also sometimes referred to as a virtual address. The set of logical addresses generated by the CPU for a program is called the logical address space. The set of all physical addresses corresponding to a set of logical address is called the physical address space. At run time / execution time, the virtual addresses are mapped to physical addresses by the memory management unit (MMU). A simple mapping scheme is illustrated below (Figure 6.1).

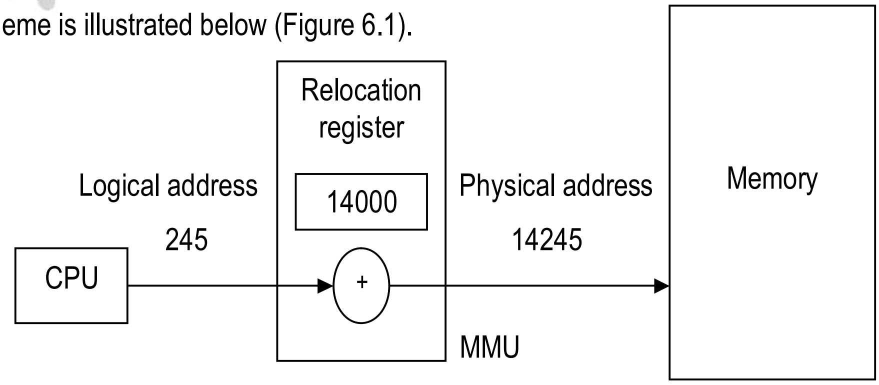


Figure 6.1: Dynamic relocation

The relocation register contains a value to be added to every address generated by the CPU for a user process at the time it is sent to the memory for a fetch or a store. For example, if the base is at 14000 then an address 0 is dynamically relocated to location 14000, an access to location 245 is mapped to 14245 (14000 + 245). Thus every address is relocated relative to the value in the relocation register. The hardware of the MMU maps logical addresses to physical addresses. Logical addresses range from 0 to a maximum (MAX) and the corresponding physical addresses range from (R + 0) to (R + MAX) for a base value of R. User programs generate only logical addresses that are mapped to physical addresses before use.

6.2 SWAPPING

A process to be executed needs to be in memory during execution. A process can be swapped out of memory in certain situations to a backing store and then brought into memory later for execution to continue. One such situation could be the expiry of a time slice if the round-robin CPU scheduling algorithm is used. On expiry of a time slice, the current process is swapped out of memory and another process is swapped into the memory space just freed because of swapping out of a process (Figure 6.2). Every time a time slice expires, a process is swapped out and another is swapped in. The memory manager that does the swapping is fast enough to always provide a process in memory for the CPU to execute. The duration of a time slice is carefully chosen so that it is sufficiently large when compared to the time for swapping.

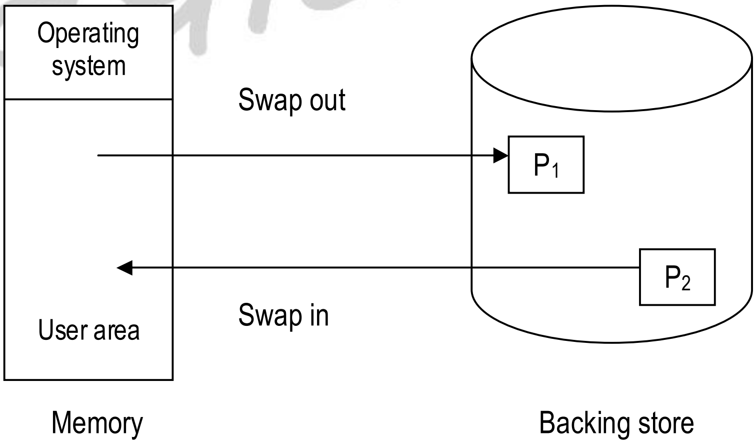


Figure 6.2: Swapping of processes

Processes are swapped between the main memory and the backing store when priority based CPU scheduling is used. The arrival of a high priority process will can a lower priority process that is executing to be swapped out to make way for a swap in. The swapping in this case is sometimes referred to as roll out / roll in.

A process that is swapped out of memory can be swapped in either into the same memory location or into a different memory location. If binding is done at load time then swap in has to be at the same location as before. But if binding is done at execution time then swap in can be into a different memory space since the mapping to physical addresses are completed during execution time.

A backing store is required for swapping. It is usually a fast disk. The processes in the ready queue have their images either in the backing store or in the main memory. When the CPU scheduler picks a process for execution, the dispatcher checks to see if the picked process is in memory. If yes then it is executed. IF not the process has to be loaded into main memory. If there is enough space in memory the process is loaded and execution starts. If not, the dispatcher swaps out a process from memory and swaps in the desired process.

A process to be swapped out must be idle. Problems arise because of pending I/O. Consider the following scenario: Process P_1 is waiting for an I/O. I/O is delayed because the device is busy. If P_1 is swapped out and its place P_2 is swapped in, then the result of the I/O uses the memory that now belongs to P_2. There can be two solutions to the above problem:

1. Never swap out a process that is waiting for an I/O
2. I/O operations to take place only into operating system buffers and not into user area buffers. These buffers can then be transferred into user area when the corresponding process is swapped in.

6.3 CONTIGUOUS ALLOCATION

The main memory is usually divided into two partitions, one of which has the resident operating system loaded into it. The other partition is used for loading user programs. The operating system is usually present in the lower memory because of the presence of the interrupt vector in the lower memory (Figure 6.3).

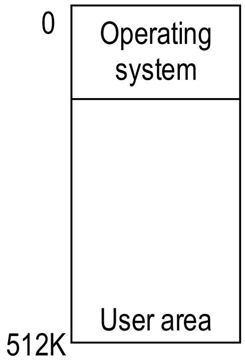


Figure 6.3: Memory partition

6.3.1 SINGLE PARTITION ALLOCATION

The operating system resides in the lower memory. User processes execute in the higher memory. There is always a possibility that user processes may try to access the lower memory either accidentally or intentionally there by causing loss of operating system code and data. This protection is usually provided by the use of a limit register and relocation register. The relocation register contains the smallest physical address that can be accessed. The limit register contains the range of logical addresses. Each logical address must be less than the content of the limit register. The MMU adds to the logical address the value in the relocation register to generate the corresponding address (Figure 6.4). Since an address generated by the CPU is checked against these two registers, both the operating system and other user programs and data are protected and are not accessible by the running process.

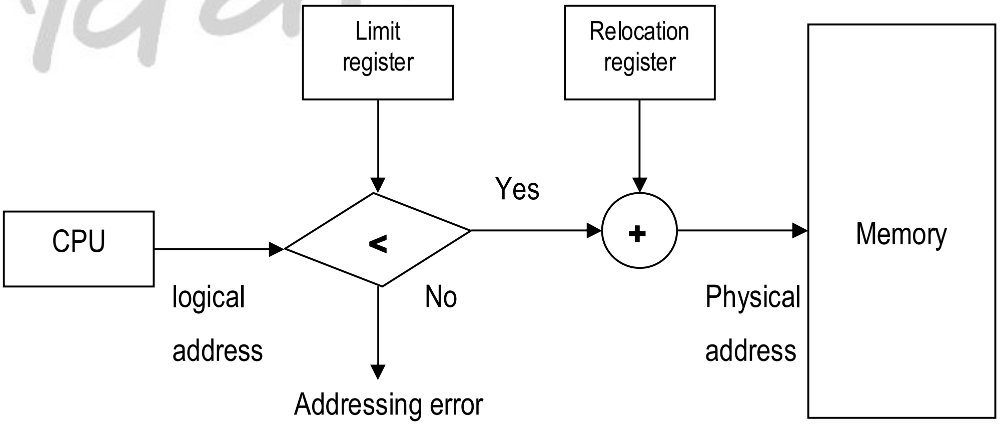


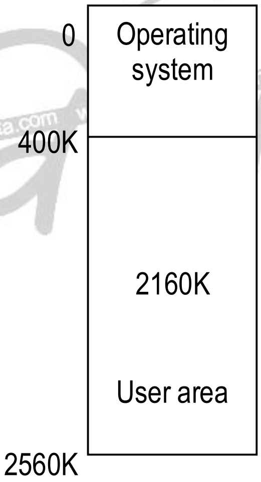
Figure 6.4: Hardware for relocation and limit register

6.3.2 MULTIPLE PARTITION ALLOCATION

Multiprogramming requires that there are many processes residing in memory so that the CPU can switch between processes. If this has to be so then user area of memory has to be divided into partitions. The simplest way is to divide the user area into fixed number of partitions, each one to hold one user process. Thus the degree of multiprogramming is equal to the number of partitions. A process from the ready queue is loaded into one of the partitions for execution. On termination the partition is free for another process to be loaded.

The disadvantage with this scheme where partitions are of fixed sizes is the selection of partition sizes. If the size is too small then large programs cannot be run. Also if the size of the partition is big then main memory space in each partition goes a waste.

A variation of the above scheme where the partition sizes are not fixed but variable is generally used. A table keeps track of that part of the memory that is used and the part that is free. Initially the entire memory of the user area is available for user processes. This can be visualized as one big hole for use. When a process is loaded a hole big enough to hold this process is searched. If one is found then memory enough for this process is allocated and the rest is available free as illustrated below (Figure 6.5).



Job queue		
Process	Memory	Time
P_1	600K	10
P_2	1000K	5
P_3	300K	20
P_4	700K	8
P_5	500K	15

Figure 6.5: Scheduling example

Total memory available:	2560K
Resident operating system:	400K

Memory available for user:	2560 – 400 = 2160K
Job queue:	FCFS
CPU scheduling:	RR (1 time unit)

Given the memory map in the illustration above, P_1, P_2, P_3 can be allocated memory immediately. A hole of size (2160 – (600 + 1000 + 300)) = 260K is left over which cannot accommodate P_4 (Figure 6.6a). After a while P_2 terminates creating the map of Figure 6.6b. P_4 is scheduled in the hole just created resulting in Figure 6.6c. Next P_1 terminates resulting in Figure 6.6d and P_5 is scheduled as in Figure 6.6e.

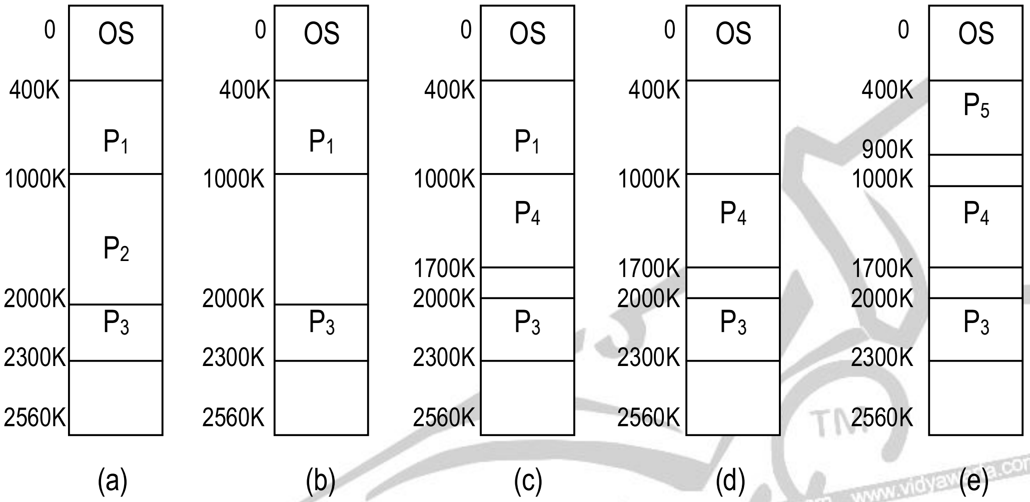


Figure 6.6: Memory allocation and job scheduling

The operating system finds a hole just large enough to hold a process and uses that particular hole to load the process into memory. When the process terminates it releases the used memory to create a hole equal to its memory requirement. Processes are allocated memory until the free memory or hole available is not big enough to load another ready process. In such a case the operating system waits for some process to terminate and free memory. To begin with there is one large big hole equal to the size of the user area. As processes are allocated into this memory, execute and terminate this hole gets divided. At any given instant there after there are a set of holes scattered all over the memory. New holes created that are adjacent to existing holes merge to form big holes.

The problem now is to satisfy a memory request of size n from a list of free holes of various sizes. Many solutions exist to determine that hole which is the best to allocate. Most common strategies are:

1. First-fit: Allocate the first hole that is big enough to hold the process. Search can either start at the beginning of the set of holes or at the point where the last search terminated.
2. Best-fit: Allocate the smallest hole that is big enough to hold the process. Here the entire list has to be searched or an ordered list of holes by size is to be maintained.
3. Worst-fit: Allocate the largest hole available. Here also the entire list has to be searched for the biggest hole or an ordered list of holes by size is to be maintained.

The size of a process is very rarely an exact size of a hole allocated. The best-fit allocation always produces an optimal allocation where the hole left over after allocation is the smallest. The first-fit is the fastest method of allocation when compared to others. The worst-fit allocation is the worst amongst the three and is seldom used.

6.3.3 FRAGMENTATION

To begin with there is one large hole for allocation to processes. As processes are loaded into memory and terminate on completion, this large hole is divided into a set of smaller holes that are scattered in between the processes. There may be a situation where the total size of these scattered holes is large enough to hold another process for execution but the process cannot be loaded, as the hole is not contiguous. This is known as external fragmentation. For example, in Figure 6.6c, a fragmented hole equal to 560K (300 + 260) is available. P_5 cannot be loaded because 560K is not contiguous.

There are situations where only a few bytes say 1 or 2 would be free if a process were allocated a hole. Then the cost of keeping track of this hole will be high. In such cases this extra bit of hole is also allocated to the requesting process. If so then a small portion of memory allocated to a process is not useful. This is internal fragmentation.

One solution to external fragmentation is compaction. Compaction is to relocate processes in memory so those fragmented holes create one contiguous hole in memory (Figure 6.7).

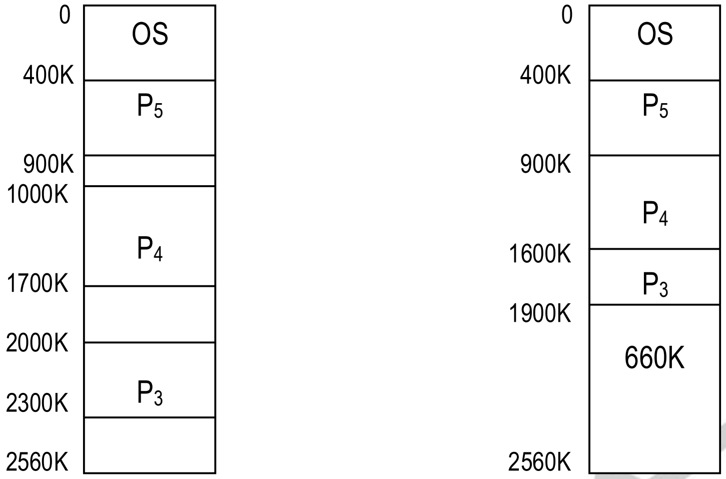


Figure 6.7: Compaction

Compaction may not always be possible since it involves relocation. If relocation is static at load time then relocation is not possible and so also compaction. Compaction is possible only if relocation is done at runtime. Even though compaction is possible the cost involved in relocation is to be considered. Sometimes creating a hole at one end of the user memory may be better where as in some other cases a contiguous hole may be created in the middle of the memory at lesser cost. The position where the hole is to be created during compaction depends on the cost of relocating the processes involved. An optimal strategy if often difficult.

Processes may also be rolled out and rolled in to affect compaction by making use of a back up store. But this would be at the cost of CPU time.

6.4 PAGING

Contiguous allocation scheme requires that a process can be loaded into memory for execution if and only if contiguous memory large enough to hold the process is available. Because of this constraint, external fragmentation is a common problem. Compaction was one solution to tide over external fragmentation. One other solution to this problem could be to permit non-contiguous logical address space so that a process can be allocated physical memory wherever it is present. This solution is implemented through the use of a paging scheme.

6.4.1 CONCEPT OF PAGING

Physical memory is divided into fixed sized blocks called frames. So also logical memory is divided into blocks of the same size called pages. Allocation of main memory to processes for execution is then just mapping pages to frames. The hardware support for paging is illustrated below (Figure 6.8).

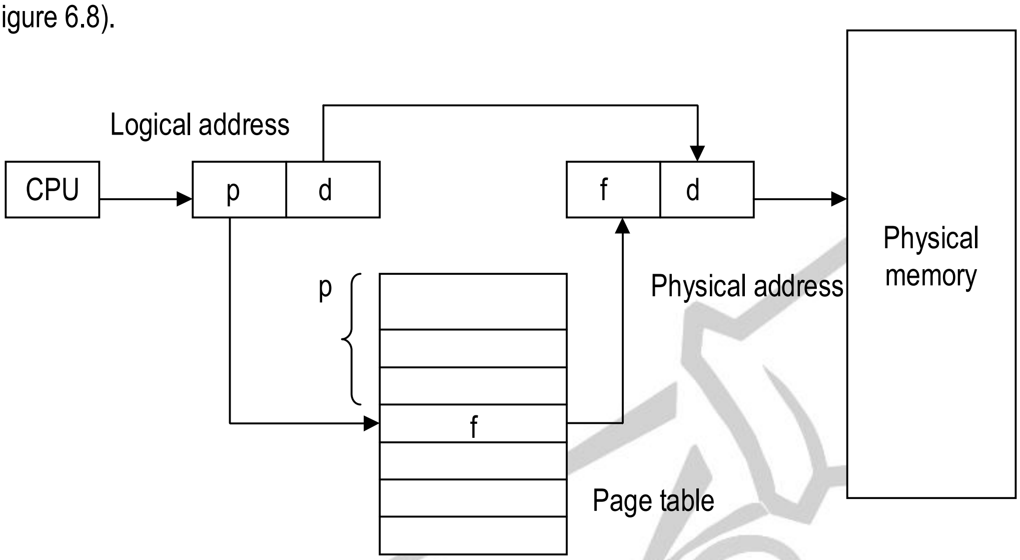


Figure 6.8: Paging hardware

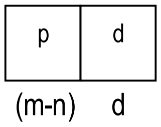
A logical address generated by the CPU consists of two parts: Page number (p) and a page offset (d). The page number is used as an index into a page table. The page table contains the base address of each frame in physical memory. The base address is combined with the page offset to generate the physical address required to access the memory unit.

The size of a page is usually a power of 2. This makes translation of a logical address into page number and offset easy as illustrated below:

Logical address space: 2m

Page size: 2n

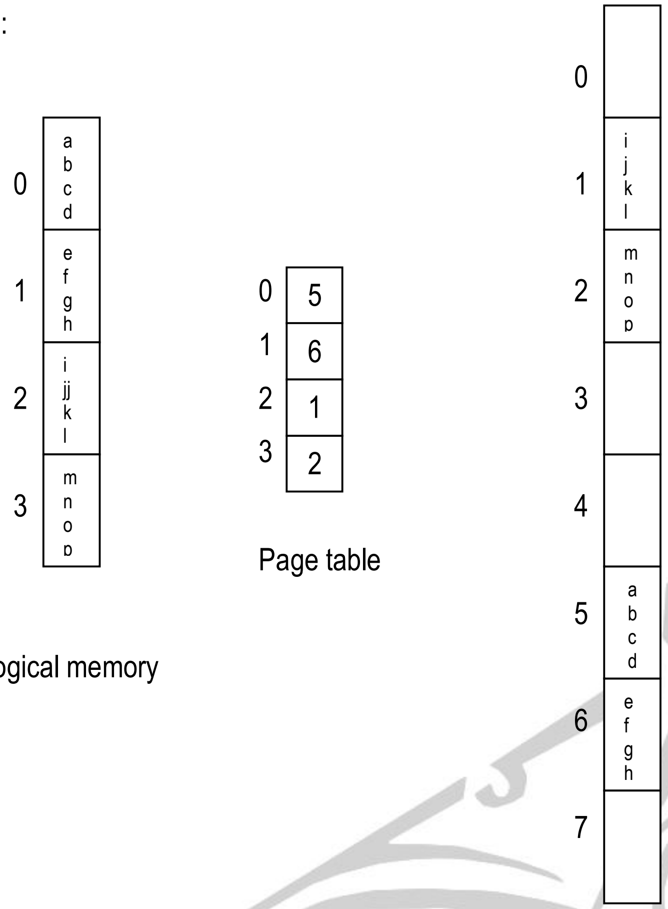
Logical address:



Where p = index into the page table

d = displacement within a page

Illustration:



Page table

Logical memory

Physical memory

Page size: 4 bytes

Physical memory: 32 bytes = 8 pages

Logical address 0 ➔ 0 + 0 ➔ (5 x 4) + 0 ➔ physical address 20

3 ➔ 0 + 3 ➔ (5 x 4) + 3 ➔ physical address 23

4 ➔ 1 + 0 ➔ (6 x 4) + 0 ➔ physical address 24

13 ➔ 3 + 1 ➔ (2 x 4) + 1 ➔ physical address 9

Thus the page table maps every logical address to some physical address. Paging does not suffer from external fragmentation since any page can be loaded into any frame. But internal fragmentation may be prevalent. This is because the total memory required by a process is not always a multiple of the page size. So the last page of a process may not be full. This leads to internal fragmentation and a portion of the frame allocated to this page will be unused. On an average one half of a page per process is wasted due to internal fragmentation. Smaller the size of a page, lesser will be the loss due to internal fragmentation. But the overhead involved is more in terms of number of entries in the page table. Also known is a fact that disk I/O is more efficient if page sizes are big. A tradeoff between the above factors is used.

A process requires n pages of memory. Then at least n frames must be free in physical memory to allocate n pages. A list of free frames is maintained. When allocation is made, pages are allocated the free frames sequentially (Figure 6.9). Allocation details of physical memory are maintained in a frame table. The table has one entry for each frame showing whether it is free or allocated and if allocated to which page of which process.

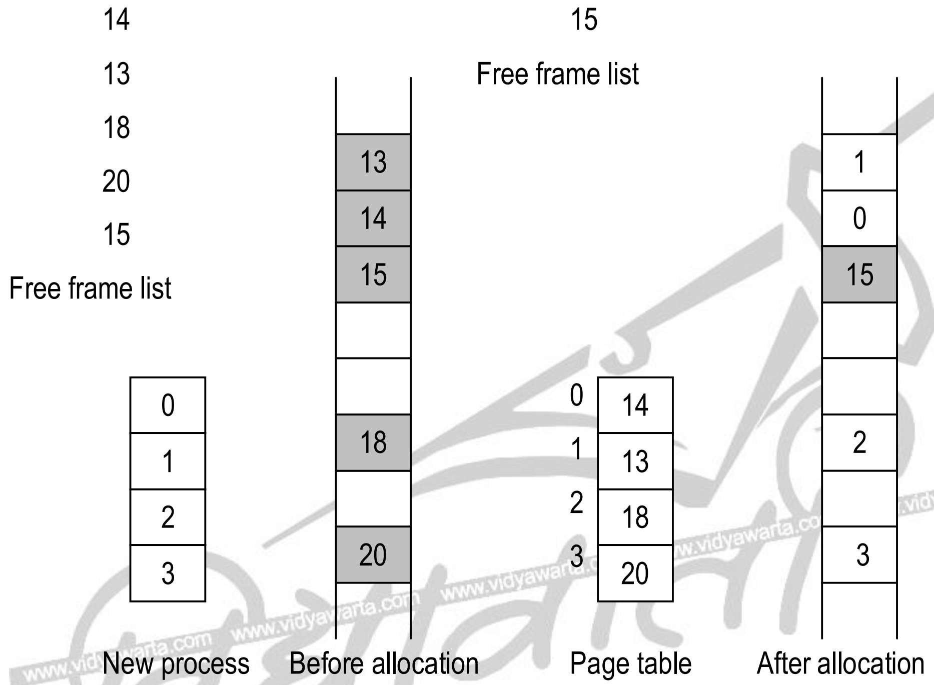


Figure 6.9: Frame allocation

6.4.2 PAGE TABLE IMPLEMENTATION

Hardware implementation of a page table is done in a number of ways. In the simplest case the page table is implemented as a set of dedicated high-speed registers. But this implementation is satisfactory only if the page table is small. Modern computers allow the page table size to be very large. In such cases the page table is kept in main memory and a pointer called the page-table base register (PTBR) helps index the page table. The disadvantage with this method is that it requires two memory accesses for one CPU address generated. For example, to access a CPU generated address, one memory access is required to index into the page table. This access using the value in PTBR fetches the frame number when combined with the page-offset produces the actual address. Using this actual address, the next memory access fetches the contents of the desired memory location.

To overcome this problem, a hardware cache called translation look-aside buffers (TLBs) are used. TLBs are associative registers that allow for a parallel search of a key item. Initially the TLBs contain only a few or no entries. When a logical address generated by the CPU is to be mapped to a physical address, the page number is presented as input to the TLBs. If the page number is found in the TLBs the corresponding frame number is available so that a memory access can be made. If the page number is not found then a memory reference to the page table in main memory is made to fetch the corresponding frame number. This page number is then added to the TLBs so that a mapping for the same address next time will find an entry in the table. Thus a hit will reduce one memory access and speed up address translation (Figure 6.10).

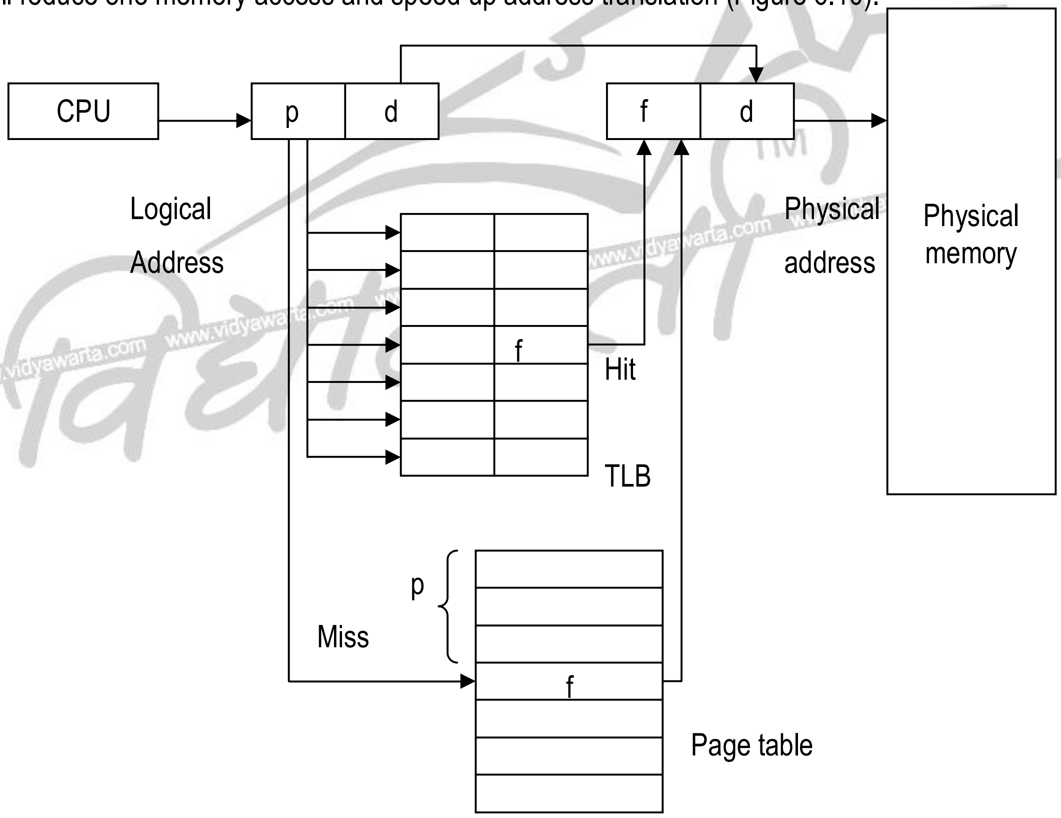


Figure 6.10: Paging hardware with TLB

6.4.3 PROTECTION

Each frame in physical memory is protected. Protection bits are used for this purpose. A page can have read only / execute only / read-write access. Hardware provides these protection accesses to frames. Illegal accesses are trapped by the operating system.

Also a process needs to access frames corresponding to its pages only. A valid-invalid bit is thus associated with each entry in a page table. When the bit is set, the page belongs to the logical address space and thus has a corresponding frame. If the bit is not set, then the page is not in the processes logical space and hence access to the frame is denied (Figure 6.11).

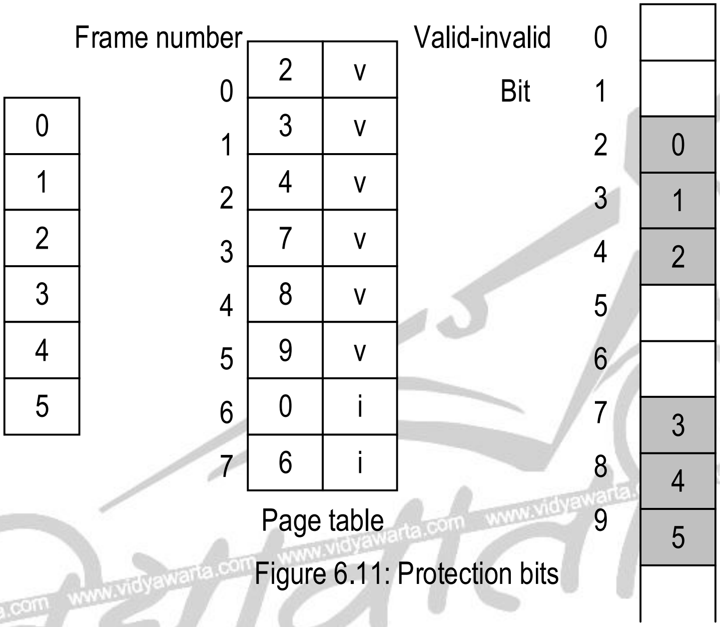


Figure 6.11: Protection bits

6.4.4 INVERTED PAGE TABLE

Since each process requires a mapping of its logical address space to a physical address space, it has a page table associated with it. This page table has one entry for each page the process is using. The corresponding entry for each page gives the mapping to physical address space. The entries in the page table are stored by page number and hence a search is easy. Main disadvantages with this scheme are that each page table may have very many entries and the page tables themselves consume a lot of physical memory, all this exercise to keep track of physical memory.

A solution to this is the use of an inverted table. This table has an entry for each frame of physical memory. Each entry has information about the process using the frame that is the process-id along with the virtual address. Thus the entire system has only one page table (Figure 6.12).

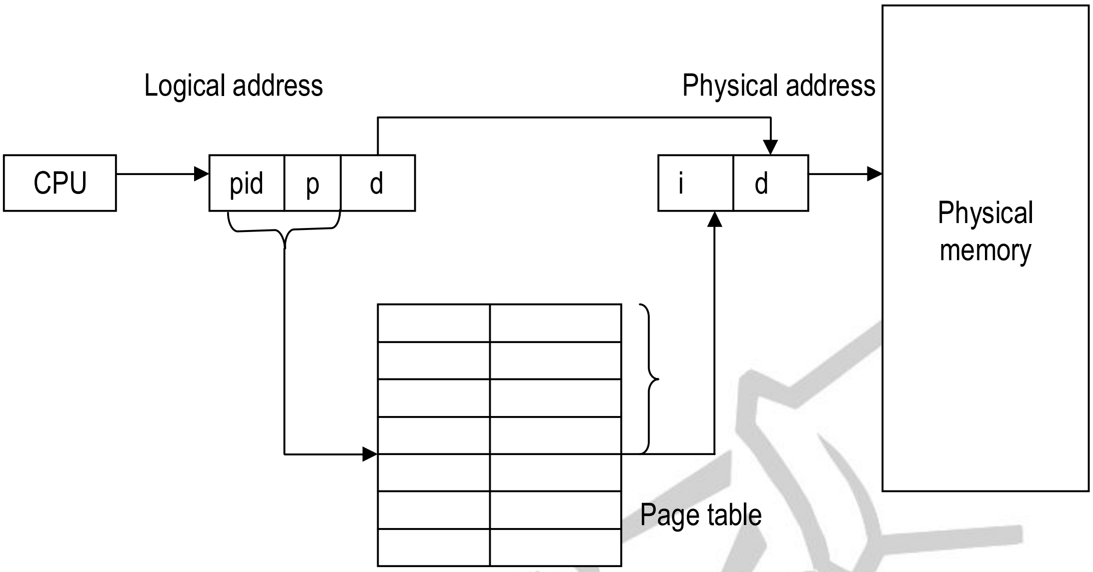


Figure 6.12: Inverted page table

Each virtual address is a triple < process-id, page number, offset >. Each entry in the inverted page table is a tuple consisting of < process-id, page number >. When CPU generates a logical address a search is made in the inverted page table. If a match is found at the ith position, then a physical address < I, offset > is generated. If no match occurs then it is an illegal memory access.

Memory required in this scheme is less. But time to search the inverted page table is more because the inverted page table is sorted by physical address and the search is being made by logical address. Hash tables or associative registers are used to overcome this problem.

6.4.5 SHARED PAGES

Programs such as editors and compilers can be shared among a number of processes. That is to say many processes can use only one copy of the above code. This is because these codes are examples of reentrant code or pure code or non-self-modifying code. Code in such cases never changes during execution. Thus many processes can execute the same code at the same time.

Paging provides the possibility of sharing common code. Only constraint is that the shared code has to be reentrant (Figure 6.13).

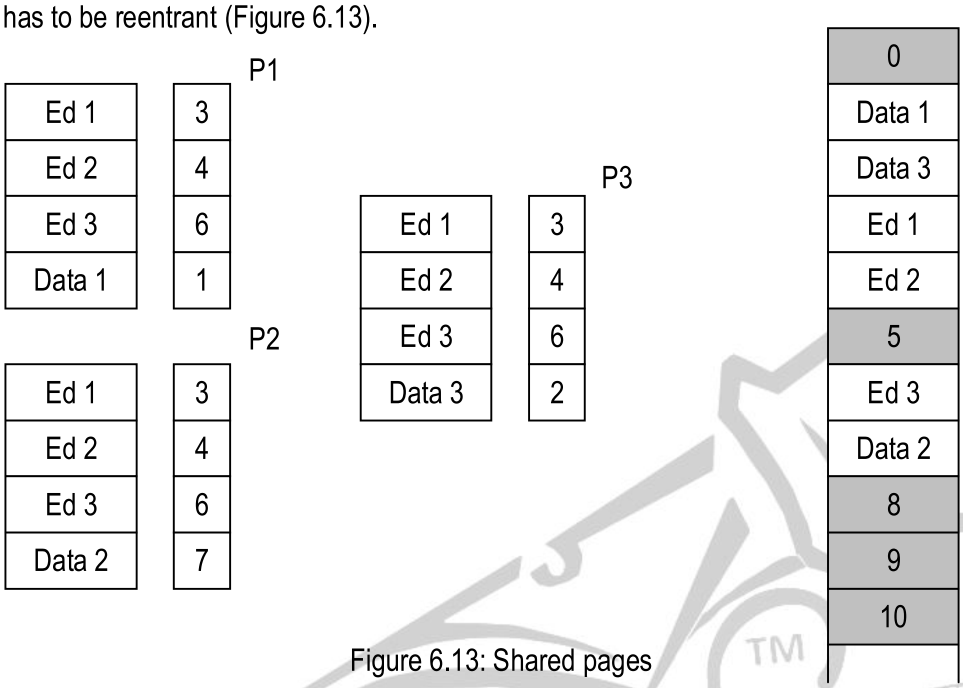


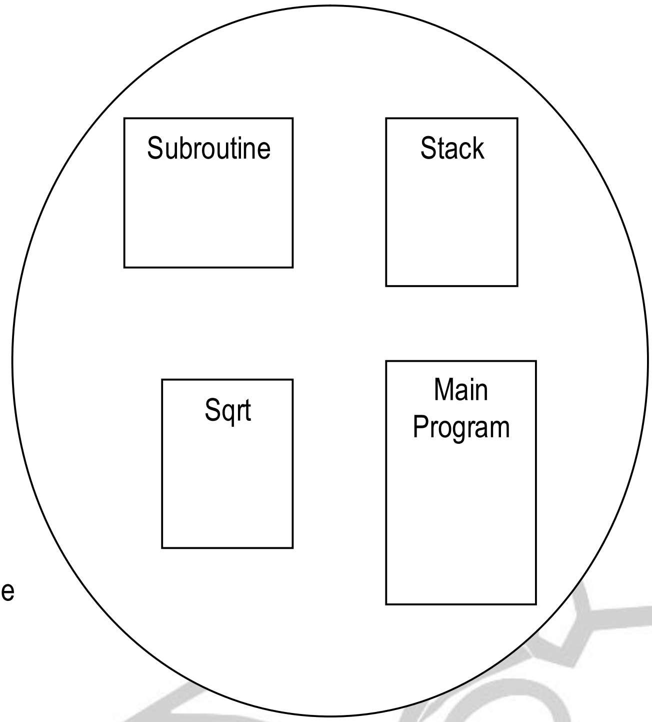
Figure 6.13: Shared pages

Each process has its own copy of registers and data storage to hold data during execution. Only one copy of the editor is kept in physical memory. The individual page tables of the different processes show the same mapping for pages corresponding to editor and different mapping for the data pages. The concept of shared pages provides a significant saving in total space required. System using inverted page table cannot implement this shared concept because more than one virtual address maps on to the same physical address which is not possible to implement in an inverted page table.

6.5 SEGMENTATION

Memory management using paging provides two entirely different views of memory – User / logical / virtual view and the actual / physical view. Both are not the same. In fact the user's view is mapped on to the physical view.

How do users visualize memory? Users prefer to view memory as a collection of variable sized segments (Figure 6.14).



Logical address space

Figure 6.14: User's view of memory

The user usually writes a modular structured program consisting of a main segment together with a number of functions / procedures. Each one of the above is visualized as a segment with its associated name. Entries in a segment are at an offset from the start of the segment.

6.5.1 CONCEPT OF SEGMENTATION

Segmentation is a memory management scheme that supports users view of main memory described above. The logical address is then a collection of segments, each having a name and a length. Since it is easy to work with numbers, segments are numbered. Thus a logical address is < segment number, offset >. User programs when compiled reflect segments present in the input. Loader while loading segments into memory assign them segment numbers.

6.5.2 SEGMENTATION HARDWARE

Even though segments in user view are same as segments in physical view, the two-dimensional visualization in user view has to be mapped on to a one-dimensional sequence of physical memory locations. This mapping is present in a segment table. An entry in a segment table consists of a base and a limit. The base corresponds to the starting physical address in memory where as the limit is the length of the segment (Figure 6.15).

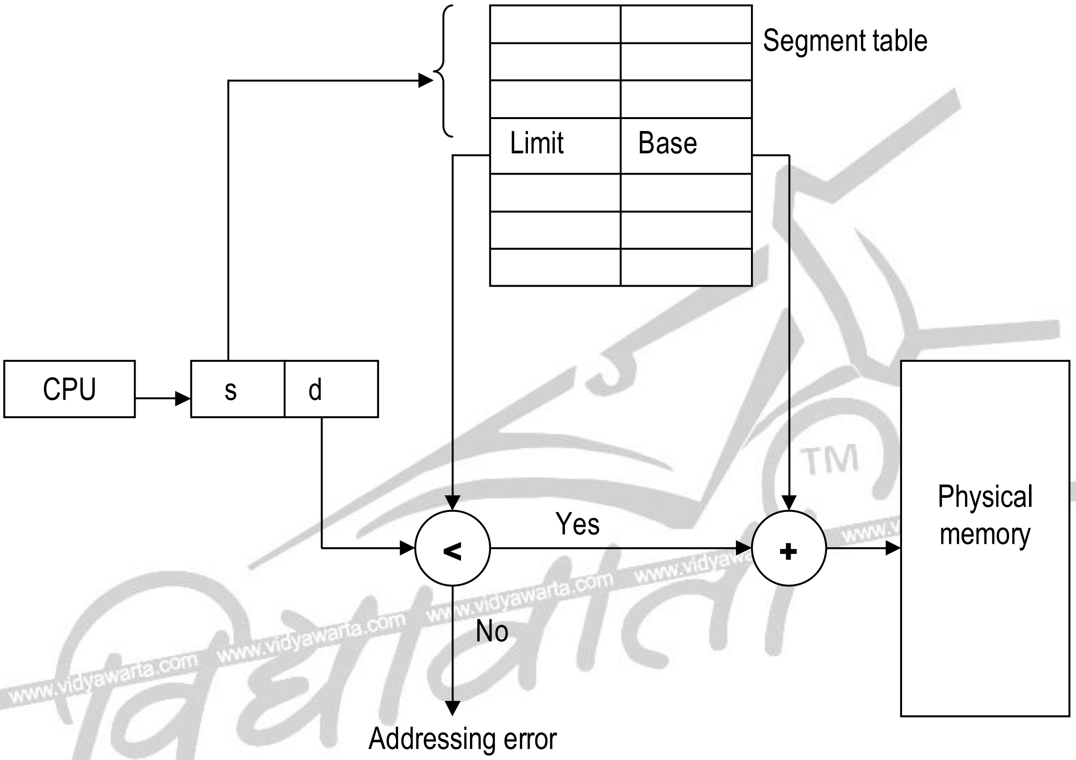
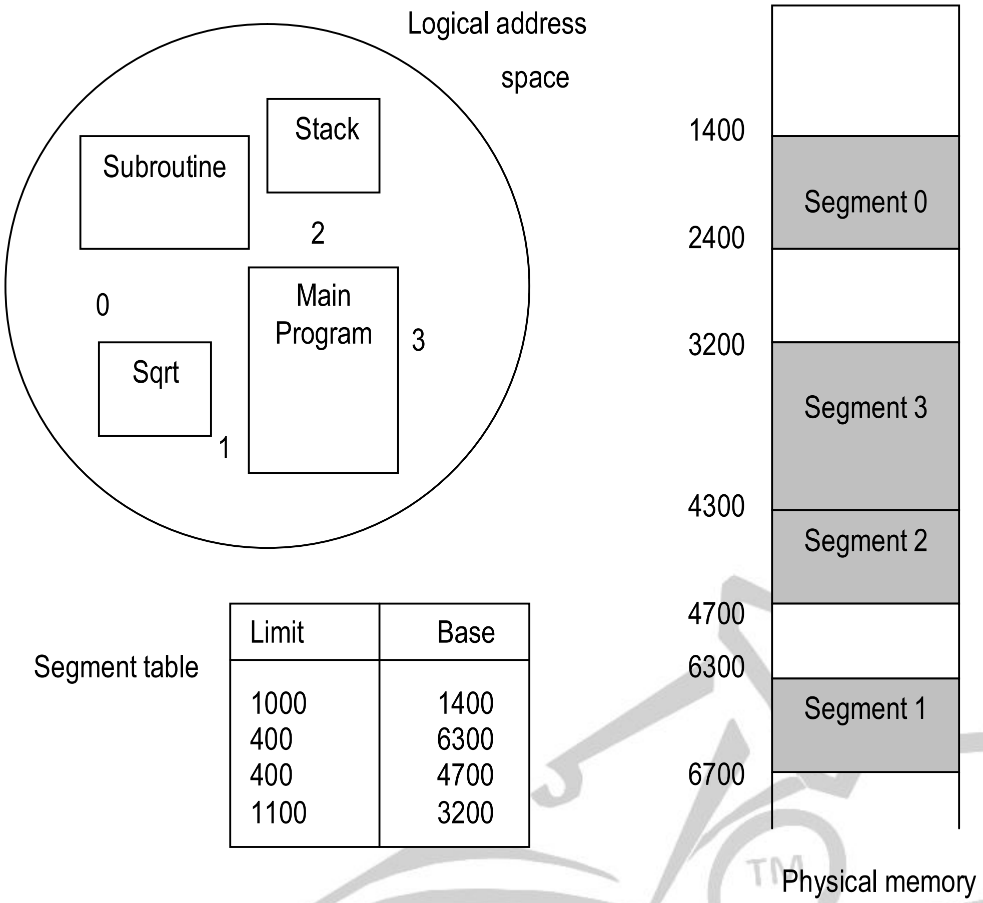


Figure 6.15: Segmentation hardware

The logical address generated by the CPU consists of a segment number and an offset within the segment. The segment number is used as an index into a segment table. The offset in the logical address should lie between 0 and a limit specified in the corresponding entry in the segment table. If not the program is trying to access a memory location which does nor belong to the segment and hence is trapped as an addressing error. If the offset is correct the segment table gives a base value to be added to the offset to generate the physical address. An illustration is given below (figure 6.16).



Limit	Base
1000	1400
400	6300
400	4700
1100	3200

Figure 6.16: Illustration of segmentation

A reference to logical location 53 in segment 2 ➔ physical location 4300 + 53 = 4353
852 in segment 3 ➔ physical location 3200 + 852 = 4062
1222 in segment 0 ➔ addressing error because 1222 > 1000

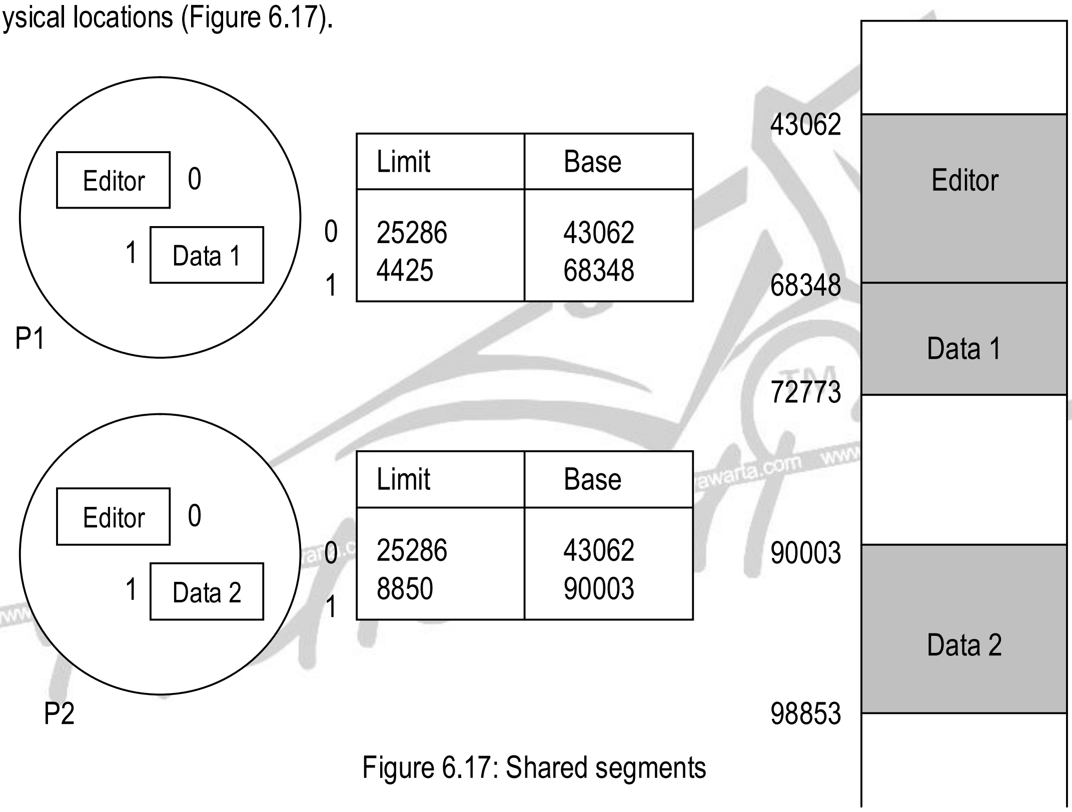
6.5.3 SEGMENTATION TABLE IMPLEMENTATION

A segment table kept in registers can be referenced quickly. Parallel execution of addition to base and comparison with limit can save considerable time. But when segment tables are big containing entries for many segments then segment tables will have to be maintained in memory. The procedure is similar as in paging and requires two memory accesses. The solution to this is also the use of associative registers.

6.5.4 PROTECTION AND SHARING

Providing protection to segments is easier than in paging. This is because segments represent semantically defined portions of code and all portions of code in a segment are likely to be used in the same way. Hence some segments contain instructions and some data. Those that contain instructions can be read only or execute only where as data segments can have write access.

Segments can be shared similar to sharing of pages in a paging environment. Segments can be shared when entries in segment tables of two different processes point to the same physical locations (Figure 6.17).



	Limit	Base
0	25286	43062
1	4425	68348

	Limit	Base
0	25286	43062
1	8850	90003

Figure 6.17: Shared segments

Sharing occurs at segment level. Several segments of a program may be shared. But each process still needs separate unique segments for data. Some problems exist while sharing parts / segments of programs. If the shared segment consists of a jump instruction then a logical address is generated for this transfer. Now its corresponding mapping into the segment table creates problems. In such cases direct reference to current segment numbers must be avoided.

6.5.5 FRAGMENTATION

Segments of user programs are allocated memory by the bob scheduler. This is similar to paging where segments could be treated as variable sized pages. So a first-fit or best-fit allocation scheme could be used since this is a dynamic storage allocation problem. But as with variable sized partition scheme, segmentation too causes external fragmentation. When any free memory segment is too small to be allocated to a segment to be loaded, then external fragmentation is said to have occurred. Memory compaction is a solution to overcome external fragmentation.

The severity of the problem of external fragmentation depends upon the average size of a segment. Each process being a segment is nothing but the variable sized partition scheme. At the other extreme, every byte could be a segment in which case external fragmentation is totally absent but relocation through the segment table is a very big overhead. A compromise could be fixed sized small segments, which is the concept of paging. Generally if segments even though variable are small, external fragmentation is also less.

6.6 SUMMARY

In this chapter we have learnt how a resource called memory is managed by the operating system. The memory in question is the main memory that holds processes during execution. Memory could be contiguously allocated by using any one of the best-fit or first-fit strategies. But one major disadvantage with this method was that of fragmentation. To overcome this problem we have seen how memory is divided into pages / frames and the processes are considered to be a set of pages in logical memory. These pages are mapped to frames in physical memory through a page table. We have also seen user's view of memory in the form of segments. Just like a page table, a segment table maps segments to physical memory.

6.7 EXERCISE

1. Differentiate between internal and external fragmentation. Explain how it is overcome.
2. Explain the paging concept with the help of a diagram.

3. Explain first-fit, Best-fit and worst-fit allocation algorithms with an example.
4. Why are TLBs required in paging? Explain.
5. With a block diagram explain the hardware required for segmentation.
6. What is an inverted page table? What are its advantages over the conventional page table? Does it have any disadvantages?
7. Write a note on shared pages.

6.8 ACTIVITY

Study the memory management schemes implemented in the operating systems noted. How is protection provided? Does it allow sharing?

CHAPTER 7

VIRTUAL MEMORY

Memory management strategies like paging and segmentation help implement the concept of multiprogramming. But they have a few disadvantages. One problem with the above strategies is that they require the entire process to be in main memory before execution can begin. Another disadvantage is the limitation on the size of the process. Processes whose memory requirement is larger than the maximum size of the memory available will never be able to be run that is users are desirous of executing processes whose logical address space is larger than the available physical address space.

Virtual memory is a technique that allows execution of processes that may not be entirely in memory. Also virtual memory allows mapping of a large virtual address space onto a smaller physical memory. It also raises the degree of multiprogramming and increases CPU utilization. Because of the above features, users are freed from worrying about memory requirements and availability.

7.1 NEED FOR VIRTUAL MEMORY TECHNIQUE

Every process needs to be loaded into physical memory for execution. One brute force approach to this is to map the entire logical space of the process to physical memory as in the case of paging and segmentation.

Many a time the entire process need not be in memory during execution. The following are some of the instances to substantiate the above statement:

- Code used to handle error and exceptional cases is executed only in case errors and exceptional conditions occur which is usually a rare occurrence may be one or no occurrences in an execution.
- Static declarations of arrays lists and tables declared with a large upper bound but used with no greater than 10% of the limit.

- Certain features and options provided in the program as a future enhancement, never used, as enhancements are never implemented.
- Even though entire program is needed, all its parts may not be needed at the same time because of overlays.

All the examples show that a program can be executed even though it is partially in memory. This scheme also has the following benefits:

- Physical memory is no longer a constraint for programs and therefore users can write large programs and execute them
- Physical memory required for a program is less. Hence degree of multiprogramming can be increased because of which utilization and throughput increase.
- I/O time needed for load / swap is less.

Virtual memory is the separation of logical memory from physical memory. This separation provides a large logical / virtual memory to be mapped on to a small physical memory (Figure 7.1).

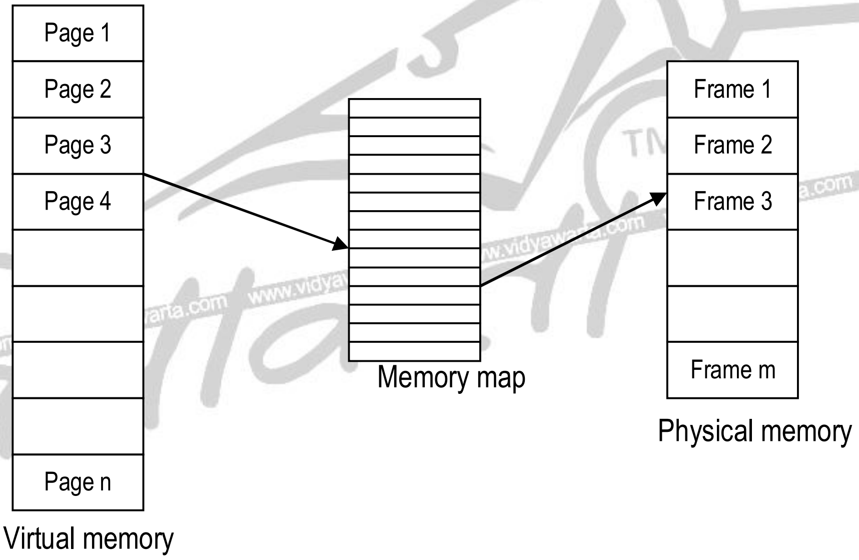


Figure 7.1: Virtual to physical memory mapping (n >> m)

Virtual memory is implemented using demand paging. Also demand segmentation could be used. A combined approach using a paged segmentation scheme is also available. Here user view is segmentation but the operating system implements this view with demand paging.

7.2 DEMAND PAGING

Demand paging is similar to paging with swapping (Figure 7.2).

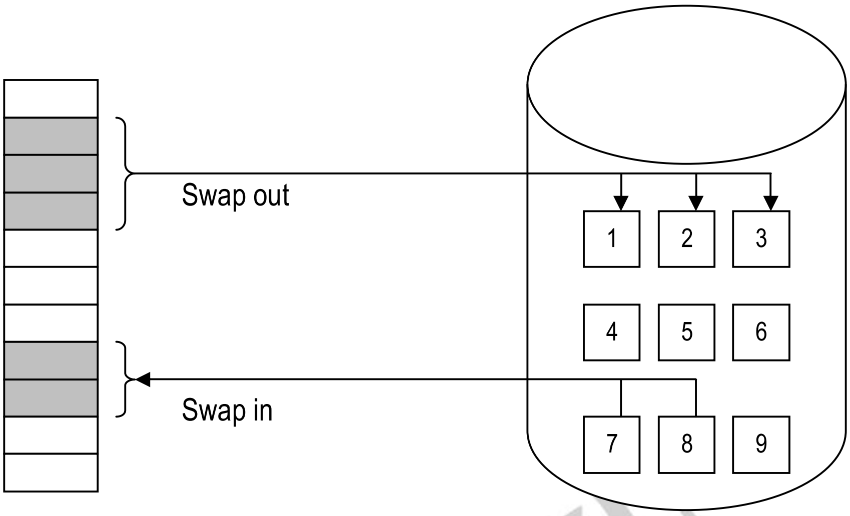


Figure 7.2: Paging with swapping

When a process is to be executed then only that page of the process, which needs to be currently executed, is swapped into memory. Thus only necessary pages of the process are swapped into memory thereby decreasing swap time and physical memory requirement. The protection valid-invalid bit which is used in paging to determine valid / invalid pages corresponding to a process is used here also (Figure 7.3).

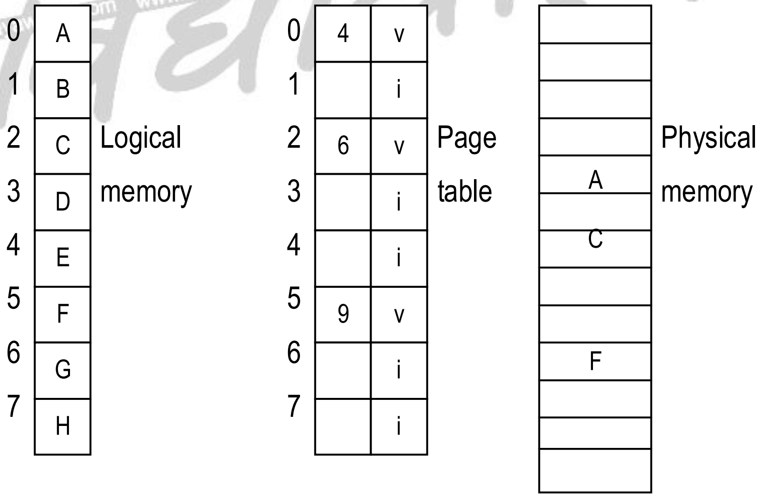


Figure 7.3: Demand paging with protection

If the valid-invalid bit is set then the corresponding page is valid and also in physical memory. If the bit is not set then any of the following can occur:

- Process is accessing a page not belonging to it that is an illegal memory access
- Process is accessing a legal page but the page is currently not in memory.

If the same protection scheme as in paging is used, then in both the above cases a page fault error occurs. The error is valid in the first case but not in the second because in the latter a legal memory access failed due to non-availability of the page in memory which is an operating system fault. Page faults can thus be handled as follows (Figure 7.4):

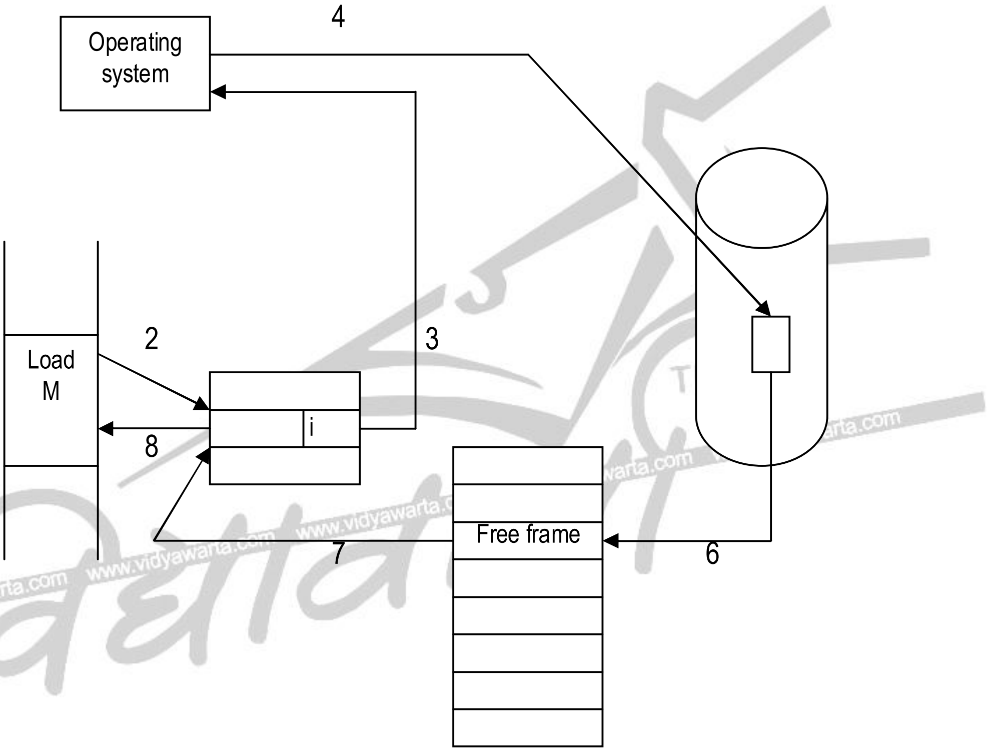


Figure 7.4: Handling a page fault

1. Check the valid-invalid bit for validity.
2. If valid then the referenced page is in memory and the corresponding physical address is generated.
3. If not valid then an addressing fault occurs.

4. The operating system checks to see if the page is in the backing store. If present then the addressing error was only due to non-availability of page in main memory and is a valid page reference.
5. Search for a free frame.
6. Bring in the page into the free frame.
7. Update the page table to reflect the change.
8. Restart the execution of the instruction stalled by an addressing fault.

In the initial case, a process starts executing with no pages in memory. The very first instruction generates a page fault and a page is brought into memory. After a while all pages required by the process are in memory with a reference to each page generating a page fault and getting a page into memory. This is known as pure demand paging. The concept 'never bring in a page into memory until it is required'.

Hardware required to implement demand paging is the same as that for paging and swapping.

- Page table with valid-invalid bit
- Secondary memory to hold pages not currently in memory, usually a high speed disk known as a swap space or backing store.

A page fault at any point in the fetch-execute cycle of an instruction causes the cycle to be repeated.

7.3 PAGE REPLACEMENT

Initially execution of a process starts with none of its pages in memory. Each of its pages page fault at least once when it is first referenced. But it may so happen that some of its pages are never used. In such a case those pages which are not referenced even once will never be brought into memory. This saves load time and memory space. If this is so the degree of multiprogramming can be increased so that more ready processes can be loaded and executed. Now we may come across a situation where in all of a sudden a process hither to not accessing certain pages starts accessing those pages. The degree of multiprogramming has been raised without looking into this aspect and the memory is over allocated. Over allocation of memory shows up when there is a page fault for want of page in memory and the operating system finds the required page in the backing store but cannot bring in the page into memory for want of free frames. More than one option exists at this stage:

- Terminate the process. Not a good option because the very purposes of demand paging to increase CPU utilization and throughput by increasing the degree of multiprogramming is lost.
- Swap out a process to free all its frames. This reduces the degree of multiprogramming that again may not be a good option but better than the first.
- Page replacement seems to be the best option in many cases.

The page fault service routine can be modified to include page replacement as follows:

1. Find for the required page in the backing store.
2. Find for a free frame
   a. if there exists one use it
   b. else find for a victim using a page replacement algorithm
   c. write the victim into the backing store
   d. modify the page table to reflect a free frame
3. Bring in the required page into the free frame.
4. Update the page table to reflect the change.
5. Restart the process.

When a page fault occurs and no free frame is present, then a swap out and a swap in occurs. Not always is a swap out necessary. Only a victim that has been modified needs to be swapped out. If not the frame can be over written by the incoming page. This will save time required to service a page fault and is implemented by the use of a dirty bit. Each frame in memory is associated with a dirty bit that is reset when the page is brought into memory. The bit is set whenever the frame is modified. Therefore the first choice for a victim is naturally that frame with its dirty bit which is not set.

Page replacement is basic to demand paging. The size of the logical address space is no longer dependent on the physical memory. Demand paging uses two important algorithms:

- Page replacement algorithm: When page replacement is necessitated due to non-availability of frames, the algorithm finds for a victim.

- Frame allocation algorithm: In a multiprogramming environment with degree of multiprogramming equal to n, the algorithm gives the number of frames to be allocated to a process.

7.4 PAGE REPLACEMENT ALGORITHMS

A good page replacement algorithm generates as low a number of page faults as possible. To evaluate an algorithm, the algorithm is run on a string of memory references and a count of the number of page faults is recorded. The string is called a reference string and is generated using either a random number generator or a trace of memory references in a given system.

Illustration:

Address sequence: 0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103, 0104,
0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105

Page size: 100 bytes

Reference string: 1 4 1 6 1 6 1 6 1 6 1

The reference in the reference string is obtained by dividing (integer division) each address reference by the page size. Consecutive occurrences of the same reference are replaced by a single reference.

To determine the number of page faults for a particular reference string and a page replacement algorithm, the number of frames available to the process need to be known. As the number of frames available increases the number of page faults decreases. In the above illustration, if frames available were 3 then there would be only 3 page faults, one for each page reference. On the other hand if there were only 1 frame available then there would be 11 page faults, one for every page reference.

7.4.1 FIFO PAGE REPLACEMENT ALGORITHM

The first-in-first-out page replacement algorithm is the simplest page replacement algorithm. When a page replacement is required the oldest page in memory is the victim.

Illustration:

Reference string: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

Memory frames: 3

Page faults:	7	7	7	2	2	2	4	4	4	0	0	0	7	7	7
		0	0	0	3	3	3	2	2	2	1	1	1	0	0
			1	1	1	0	0	0	3	3	3	2	2	2	1

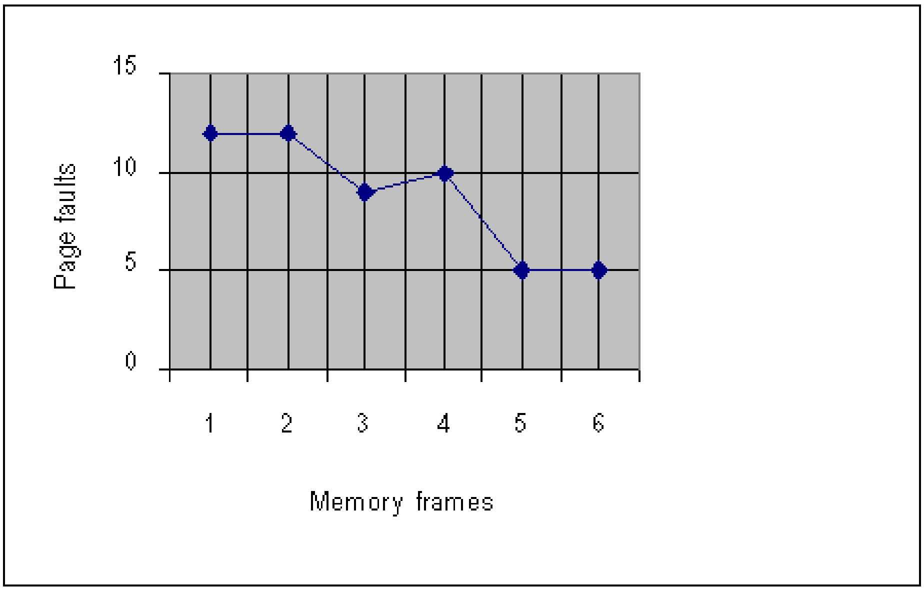
Number of page faults = 15.

The performance of the FIFO algorithm is not always good. The replaced page may have an initialization module that needs to be executed only once and therefore no longer needed. On the other hand the page may have a heavily used variable in constant use. Such a page swapped out will cause a page fault almost immediately to be brought in. Thus the number of page faults increases and results in slower process execution. Consider the following reference string:

Reference string: 1 2 3 4 1 2 5 1 2 3 4 5

Memory frames: 1, 2, 3, 4, 5

The chart below gives the number of page faults generated for each of the 1, 2, 3, 4 and 5 memory frames available.



As the number of frames available increases, the number of page faults must decrease. But the chart above shows 9 page faults when memory frames available are 3 and 10 when memory frames available are 4. This unexpected result is known as Belady's anomaly.

Implementation of FIFO algorithm is simple. A FIFO queue can hold pages in memory with a page at the head of the queue becoming the victim and the page swapped in joining the queue at the tail.

7.4.2 OPTIMAL ALGORITHM

An optimal page replacement algorithm produces the lowest page fault rate of all algorithms. The algorithm is to replace the page that will not be used for the longest period of time to come. Given a fixed number of memory frame by allocation, the algorithm always guarantees the lowest possible page fault rate and also does not suffer from Belady’s anomaly.

Illustration:

Reference string:	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
Memory frames:	3																			
Page faults:	7	7	7	2		2		2			2			2				7		
		0	0	0		0		4			0			0				0		
			1	1		3		3			3			1				1		

Number of page faults = 9.

Ignoring the first three page faults that do occur in all algorithms, the optimal algorithm is twice as better than the FIFO algorithm for the given string.

But implementation of the optimal page replacement algorithm is difficult since it requires future a priori knowledge of the reference string. Hence the optimal page replacement algorithm is more a benchmark algorithm for comparison.

7.4.3 LRU PAGE REPLACEMENT ALGORITHM

The main distinction between FIFO and optimal algorithm is that the FIFO algorithm uses the time when a page was brought into memory (looks back) where as the optimal algorithm uses the time when a page is to be used in future (looks ahead). If the recent past is used as an approximation of the near future, then replace the page that has not been used for the longest period of time. This is the least recently used (LRU) algorithm.

Illustration:

Reference string: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

Memory frames: 3

Page faults:	7	7	7	2		2		4	4	4	0			1		1		1		
		0	0	0		0		0	0	3	3			3		0		0		
			1	1		3		3	2	2	2			2		2		7		

Number of page faults = 12.

The LRU page replacement algorithm with 12 page faults is better than the FIFO algorithm with 15 faults. The problem is to implement the LRU algorithm. An order for the frames by time of last use is required. Two options are feasible:

- By use of counters
- By use of stack

In the first option using counters, each page table entry is associated with a variable to store the time when the page was used. When a reference to the page is made, the contents of the clock are copied to the variable in the page table for that page. Every page now has the time of last reference to it. According to the LRU page replacement algorithm the least recently used page is the page with the smallest value in the variable associated with the clock. Overheads here include a search for the LRU page and an update of the variable to hold clock contents each time a memory reference is made.

In the second option a stack is used to keep track of the page numbers. A page referenced is always put on top of the stack. Therefore the top of the stack is the most recently used page and the bottom of the stack is the LRU page. Since stack contents in between need to be changed, the stack is best implemented using a doubly linked list. Update is a bit expensive because of the number of pointers to be changed, but there is no necessity to search for a LRU page.

LRU page replacement algorithm does not suffer from Belady’s anomaly. But both of the above implementations require hardware support since either the clock variable or the stack must be updated for every memory reference.

7.4.4 LRU APPROXIMATION ALGORITHM'S

Hardware support to implement LRU algorithm is very essential. But when hardware support provided is partial then an approximation of the LRU algorithm is used. Many systems provide hardware support in the form of a reference bit. A reference bit is associated with every page and is set each time a reference to the page is made. The reference bit gives information about reference to pages. Initially the reference bits for all pages are reset. As execution proceeds these bits get set as and when references to pages are made. At an instant there after some of the bits are set and some are not depending on reference to pages.

7.4.4.1 ADDITIONAL REFERENCE BITS ALGORITHM

Each page can have an additional 8-bit byte to store ordering information. At regular intervals the operating system shifts the reference bit into the high order bit of the 8-bit byte, shifting the other bits right by 1 bit, discarding the low order bit. History of the page reference is recorded in this byte. If the byte contains all 0's then the page has not been used for the last 8 time periods. If it has all 1's, then it is used in each of the previous time periods. A page with the lowest value in the b-bit byte is the LRU page. Number of bits of history required depends on the hardware available. In the extreme case it can be 0 bits which is nothing but the use of only a reference bit.

7.4.4.2 SECOND CHANCE ALGORITHM

In this algorithm the reference bit is the only bit to store history. If the reference bit of a page is not set then that page is a victim for replacement whenever a frame is required. But if the reference bit is set then that page is given a second chance and next FIFO page is selected. This page that is given a second chance will not be replaced until all the other pages have been replaced or given a second chance.

The second chance algorithm is best implemented using a circular queue. A pointer is maintained to indicate the next page to be replaced. Whenever a page replacement is necessary the pointer is moved in FIFO order to find a page with its reference bit not set. As the pointer moves the reference bits which are set are reset thus giving a second chance to the pages that have their reference bits set. A page used often keeps its reference bit always set and will never be replaced. If all the reference bits are set, then the algorithm is the same as FIFO algorithm.

In some cases the dirty bit is combined with the reference bit to form an ordered pair. The enhanced second chance algorithm uses this pair of values. Given two bits we have the following four combinations of values:

Reference	Dirty	
0	0	Neither used recently nor modified, best page as a victim.
0	1	Not used recently but modified, If replaced a swap out needed.
1	0	Recently used but not modified, can be a better victim than the previous.
1	1	Recently used and also modified, if replaced a swap out necessary

Replacement is done by examining pages belonging to lower classes. In this algorithm pages that have been modified are given preference sine a swap out of these pages needs an I/O.

7.5 ALLOCATION OF FRAMES

The number of frames a process can use has to be determined. In a single user system the entire set of frames is available to the user. Only when all the available frames are exhausted, a page replacement algorithm will be used to bring in a page. The free frame list has the list of free frames from which allocation can be made.

The problem is difficult in a multiprogramming environment with demand paging because there are more than one user processes in memory at the same time. The maximum number of frames that can be allocated is equal to the maximum number of frames available. The minimum number of frames that can be allocated is less than this maximum. The number of frames allocated to a process determines the number of page faults. Lesser the number of allocated frames, more the number of page faults and slower is the execution.

The minimum number of frames that can be allocated to a process is dictated by the system architecture, more precisely by the instruction set architecture. Whenever a page fault occurs during instruction execution, the instruction has to be re-executed. Therefore the minimum number of frames allocated must be enough to hold all pages that any single instruction can reference.

For example, instructions having only one memory address will require at least 2 frames- one for the instruction and one for the memory reference. In addition if one level of indirection is allowed then at least 3 frames will be required.

The minimum number of frames is defined by the system architecture where as the maximum is dependent on the total number of frames available in physical memory. An appropriate choice is some where between these limits.

In a multiprogramming environment n processes are in memory. Thus m frames are to allocated among n processes where m is the maximum number of free frames available in physical memory. The easiest way is to allocate m frames among n processes is an equal allocation scheme where each process gets m / n frames for use. A better allocation will make use of the need for frames by processes while making allocation. A proportional allocation scheme allocates frames to processes as a proportion of some property say memory requirement or priority. If memory required be processes is considered, then number of frames allocated to processes is proportional to the size of the process. If S is the total memory requirement of the n processes, then (s_i / S) x m rounded to the next integer will be number of frames allocated to a process P_i.

In either of the two schemes, increase in the degree of multiprogramming will decrease the number of frames allocated to a process. When the degree of multiprogramming decreases with processes terminating after execution the free frames could be spread out over the remaining processes.

Page replacement algorithms can be classified into global and local replacement algorithms. Global replacement allows a process to select a frame for replacement from the set all frames even though victim frame is allocated to another process where as local replacement requires processes to select a frame for replacement only from its own set of allocated frames. With global replacement number of frames allocated to a process change with process stealing frames from other processes thus trying to increase the number of frames allocated to it. But with local replacement number of frames allocated to a process is static and does not change.

With global replacement algorithm page fault rates associated with processes are not dependent on the process alone but also on other processes. This is not the case with local replacement where the process alone is responsible for its paging activity. It has been seen that global replacement results in better system through put by making available to processes that need more frames from those that need less.

7.6 THRASHING

When a process does not have enough frames or when a process is executing with a minimum set of frames allocated to it which are in active use, there is always a possibility that the process will page fault quickly. The page in active use becomes a victim and hence page faults will occur again and again and again. In this case a process spends more time in paging than executing. This high paging activity is called thrashing.

7.6.1 CAUSE FOR THRASHING

The operating system closely monitors CPU utilization. When CPU utilization drops below a certain threshold, the operating system increases the degree of multiprogramming by bringing in a new process to increase CPU utilization. Let a global page replacement policy be followed. A process requiring more frames for execution page faults and steals frames from other processes which are using those frames. This causes the other processes also to page fault. Paging activity increases with longer queues at the paging device but CPU utilization drops. Since CPU utilization drops, the job scheduler increases the degree of multiprogramming by bringing in a new process. This only increases paging activity to further decrease CPU utilization. This cycle continues. Thrashing has set in and through put drastically drops. This is illustrated in the figure below (Figure 7.5):

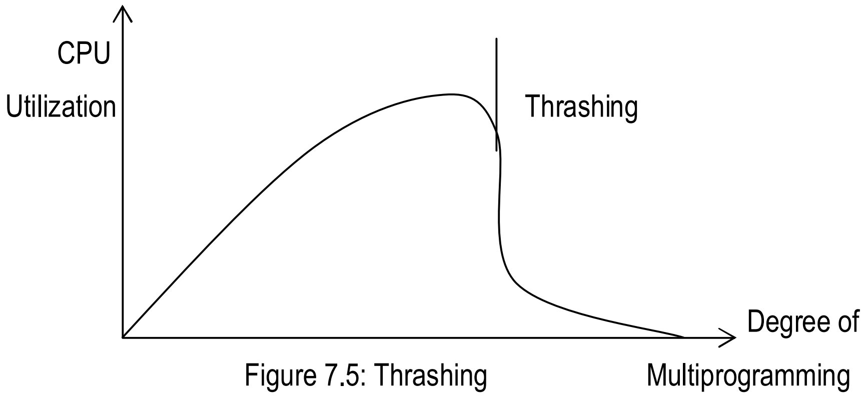


Figure 7.5: Thrashing

When a local page replacement policy is used instead of a global policy, thrashing is limited to a process only.

To prevent thrashing, a process must be provided as many frames as it needs. A working-set strategy determines how many frames a process is actually using by defining what is known as a locality model of process execution.

The locality model states that as a process executes it moves from one locality to another where a locality is a set of active pages used together. These localities are strictly not distinct and overlap. For example, a subroutine call defines a locality by itself where memory references are made to instructions and variables in the subroutine. A return from the subroutine shifts the locality with instructions and variables of the subroutine no longer in active use. So localities in a process are defined by the structure of the process and the data structures used there in. The locality model states that all programs exhibit this basic memory reference structure.

Allocation of frames enough to hold pages in the current locality will cause faults for pages in this locality until all the required pages are in memory. The process will then not page fault until it changes locality. If allocated frames are less than those required in the current locality than thrashing occurs because the process is not able to keep in memory actively used pages.

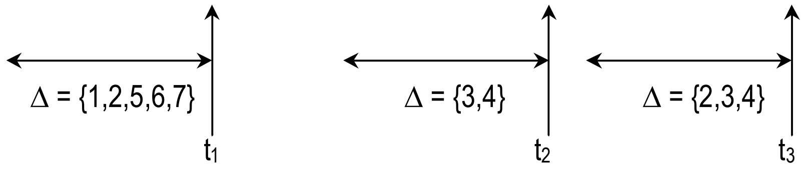
7.6.2 WORKING SET MODEL

The working set model is based on the locality model. A working set window is defined. It is a parameter Δ that maintains the most recent Δ page references. This set of most recent Δ page references is called the working set. An active page always finds itself in the working set. Similarly a page not used will drop off the working set Δ time units after its last reference. Thus the working set is an approximation of the program's locality.

Illustration:

Pages referenced:

…. 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 ….



If Δ = 10 memory references then a working set {1,2,5,6,7} at t_1 has changed to {3,4} at t_2 and {2,3,4} at t_3. The size of the parameter Δ defines the working set. If too small it does not consist of the entire locality. If it is too big then it will consist of overlapping localities.

Let WSS_i be the working set for a process P_i. Then $D = \sum WSS_i$ will be the total demand for frames from all processes in memory. If total demand is greater than total available that is D>m then thrashing has set in.

The operating system thus monitors the working set of each process and allocates to that process enough frames equal to its working set size. If free frames are still available then degree of multiprogramming can be increased. If at any instant D>m then the operating system swaps out a process to decrease the degree of multiprogramming so that released frames could be allocated to other processes. The suspended process is brought in later and restarted.

The working set window is a moving window. Each memory reference appears at one end of the window while an older reference drops off at the other end.

The working set model prevents thrashing while the degree of multiprogramming is kept as high as possible there by increasing CPU utilization.

7.6.3 PAGE FAULT FREQUENCY

One other way of controlling thrashing is by making use of the frequency of page faults. This page fault frequency (PPF) strategy is a more direct approach.

When thrashing has set in page fault is high. This means to say that a process needs more frames. If page fault rate is low, the process may have more than necessary frames to execute. So upper and lower bounds on page faults can be defined. If the page fault rate exceeds the upper bound, then another frame is allocated to the process. If it falls below the lower bound then a frame already allocated can be removed. Thus monitoring the page fault rate helps prevent thrashing.

As in the working set strategy, some process may have to be suspended only to be restarted later if page fault rate is high and no free frames are available so that the released frames can be distributed among existing processes requiring more frames.

7.7 SOME OTHER CONSIDERATIONS FOR SELECTION OF REPLACEMENT ALGORITHM AND ALLOCATION POLICY

A good paging system depends on the page replacement algorithm used and the frame allocation policy followed. Some other considerations for selection of replacement algorithm and allocation policy are pre-paging, page size, program structure, I/O inter locks and many more. Discussed below are the two important policies of pre-paging and page size.

7.7.1 PRE-PAGING

In a pure demand paging system, a number of page faults occur when a process starts execution. This is because the process tries to bring into memory all pages referenced in the initial locality. The same occurs when a process swapped out has to be restarted. Pre-paging is an attempt to prevent this large number of page faults that occur initially to bring in all pages that are required in a given locality. The working set is a good approximation of the program's locality. So pages required in the working set need to be in memory when the process initially starts or when it resumes after an I/O thus preventing initial page faults.

The cost of pre-paging has to be weighed against the cost of servicing page faults. If s is the number of pages pre-paged and α is a fraction of s that will actually be used, then pre paging is advantageous only if α is very close to one.

7.7.2 PAGE SIZE

Many factors decide the page size. The size of a page is typically a power of 2. The selection of a power of 2 as the page size makes address translation from a logical address to a physical address easy.

Page size is defined by the hardware and is dependent on the system architecture. Page sizes generally range from 2^9 to 2^{14} bytes.

For a given logical memory space decreasing page size increases the number of pages and hence the size of the page table. If an inverted page table is not used then each process must have its own page table. In such a case it is desirable to have large page size so that size of the page table is small.

Memory is better utilized when the page size is small. The total memory required by a process is not an exact multiple of page size. Because of this there is always some amount of internal fragmentation present in the last page of the process. On an average half of the last page is lost due to internal fragmentation. If this is the case, smaller the page size lesser is the amount of internal fragmentation.

The time required for reading or writing a page is also a consideration for page size. I/O time is composed of seek time, latency time and transfer time. Out of these the transfer time is proportional to the amount of information transfer or the page size. Smaller the page size lesser is the transfer time. Therefore page size need be small. But seek and latency times are very large when compared to transfer time. Infact transfer time forms only about 1% of the total I/O time. Then if the page size is doubled the increase in I/O time is very marginal and therefore minimizes I/O time. Thus page sizes should be large.

Smaller the page size, locality will be improved. Also pre-paging based on working set will be advantageous, as a greater fraction of pages pre-paged will be active. So smaller page size will result is less I/O and less allocated memory.

Smaller page size generates more page faults. To minimize the number of page faults the page size should be large. But this will result in more internal fragmentation.

Choosing the best page size is not straightforward. Factors such as internal fragmentation and locality of reference argue for a small page size where as table size, I/O times and page faults argue for a large page size. It is a compromise among so many factors. Some systems therefore allow for the use of different page sizes.

7.8 SUMMARY

In this chapter we have studied a technique called virtual memory that creates an illusion for the user that he has a large memory at his disposal. But in reality, only a limited amount of main memory is available and that too is shared amongst several users. We have also studied demand paging which is the main concept needed to implement virtual memory. Demand paging brought in the need for page replacements if required pages are not in memory for execution. Different page replacement algorithms were studied. Thrashing, its cause and ways of controlling it were also addressed.

7.9 EXERCISE

1. What is virtual memory? Distinguish between logical address and physical address.
2. Explain demand paging virtual memory system.
3. Describe how a page fault occurs. What action does the operating system take when a page fault occurs?
4. Explain Belady’s anomaly with the help of FIFO page replacement algorithm.
5. Consider the following sequence of memory references from a 460 word program:
   10, 11, 104, 170, 73, 309, 185, 245, 246, 434, 458, 364.
   Give the reference string assuming a page size of 100 words.
   Find the total number of page faults using LRU and FIFO page replacement algorithms. Assume a page size of 100 words and main memory divided into 3 page frames.
6. What is thrashing and what is its cause?
7. What do we understand by locality of reference? How does the working set model approximate it?
8. What is the role of page size in a paging environment?

7.10 ACTIVITY

Find out if paging is implemented in the operating systems noted. If so is demand paging used and what are the algorithms for page replacement? What are the mechanisms used to control thrashing.

CHAPTER 8

FILE SYSTEM INTERFACE

The operating system is a resource manager. Secondary resources like the disk are also to be managed. Information is stored in secondary storage because it costs less, is non-volatile and provides large storage space. Processes access data / information present on secondary storage while in execution. Thus the operating system has to properly organize data / information in secondary storage for efficient access.

The file system is the most visible part of an operating system. It is a way for on-line storage and access of both data and code of the operating system and the users. The file system has two distinct parts:

1. Collection of files, each storing related information.
2. Directory structure that organizes and provides information about the files in the system.

8.1 CONCEPT OF A FILE

Users use different storage media such as magnetic disks, tapes, optical disks and so on. All these different storage media have their own way of storing information. The operating system provides a uniform logical view of information stored in these different media. The operating system abstracts from the physical properties of its storage devices to define a logical storage unit called a file. These files are then mapped on to physical devices by the operating system during use. The storage devices are usually non-volatile meaning the contents stored in these devices persist through power failures and system reboots.

The concept of a file is extremely general. A file is a collection of related information recorded on the secondary storage. For example, a file containing student information, a file containing employee information, files containing C source code and so on. A file is thus the smallest allotment of logical secondary storage, that is any information to be stored on the secondary storage need to be written on to a file and the file is to be stored. Information in files could be program code or data in numeric, alphanumeric, alphabetic or binary form either formatted or in free form. A file is therefore a collection of records if it is a data file or a collection of bits / bytes / lines if it is code. Program code stored in files could be source code, object code or executable code where as data stored in files may consist of plain text, records pertaining to an application, images, sound and so on. Depending on the contents of a file, each file has a predefined structure. For example, a file containing text is a collection of characters organized as lines, paragraphs and pages where as a file containing source code is an organized collection of segments which in turn are organized into declaration and executable statements.

8.1.1 ATTRIBUTES OF A FILE

A file has a name. The file name is a string of characters. For example, test.c, pay.cob, master.dat, os.doc. In addition to a name, a file has certain other attributes. Important attributes among them are:

- Type: information on the type of file.
- Location: information is a pointer to a device and the location of the file on that device.
- Size: The current size of the file in bytes.
- Protection: Control information for user access.
- Time, date and user id: information regarding when the file was created last modified and last used. This information is useful for protection, security and usage monitoring.

All these attributes of files are stored in a centralized place called the directory. The directory is big if the numbers of files are many and also requires permanent storage. It is therefore stored on secondary storage.

8.1.2 OPERATIONS ON FILES

A file is an abstract data type. Six basic operations are possible on files. They are:

1. Creating a file: two steps in file creation include space allocation for the file and an entry to be made in the directory to record the name and location of the file.

2. Writing a file: parameters required to write into a file are the name of the file and the contents to be written into it. Given the name of the file the operating system makes a search in the directory to find the location of the file. An updated write pointer enables to write the contents at a proper location in the file.
3. Reading a file: to read information stored in a file the name of the file specified as a parameter is searched by the operating system in the directory to locate the file. An updated read pointer helps read information from a particular location in the file.
4. Repositioning within a file: a file is searched in the directory and a given new value replaces the current file position. No I/O takes place. It is also known as file seek.
5. Deleting a file: The directory is searched for the particular file, If it is found, file space and other resources associated with that file are released and the corresponding directory entry is erased.
6. Truncating a file: file attributes remain the same, but the file has a reduced size because the user deletes information in the file. The end of file pointer is reset.

Other common operations are combinations of these basic operations. They include append, rename and copy. A file on the system is very similar to a manual file. An operation on a file is possible only if the file is open. After performing the operation the file is closed. All the above basic operations together with the open and close are provided by the operating system as system calls.

8.1.3 TYPES OF FILES

The operating system recognizes and supports different file types. The most common way of implementing file types is to include the type of the file as part of the file name. The attribute ‘name’ of the file consists of two parts: a name and an extension separated by a period. The extension is the part of a file name that identifies the type of the file. For example, in MS-DOS a file name can be up to eight characters long followed by a period and then a three-character extension. Executable files have a .com / .exe / .bat extension, C source code files have a .c extension, COBOL source code files have a .cob extension and so on.

If an operating system can recognize the type of a file then it can operate on the file quite well. For example, an attempt to print an executable file should be aborted since it will produce only garbage. Another use of file types is the capability of the operating system to automatically recompile the latest version of source code to execute the latest modified program. This is observed in the Turbo / Borland integrated program development environment.

8.1.4 STRUCTURE OF A FILE

File types are an indication of the internal structure of a file. Some files even need to have a structure that need to be understood by the operating system. For example, the structure of executable files need to be known to the operating system so the it can be loaded in memory and control transferred to the first instruction for execution to begin. Some operating systems also support multiple file structures.

Operating system support for multiple file structures makes the operating system more complex. Hence some operating systems support only a minimal number of files structures. A very good example of this type of operating system is the UNIX operating system. UNIX treats each file as a sequence of bytes. It is up to the application program to interpret a file. Here maximum flexibility is present but support from operating system point of view is minimal. Irrespective of any file structure support, every operating system must support at least an executable file structure to load and execute programs.

Disk I/O is always in terms of blocks. A block is a physical unit of storage. Usually all blocks are of same size. For example, each block = 512 bytes. Logical records have their own structure that is very rarely an exact multiple of the physical block size. Therefore a number of logical records are packed into one physical block. This helps the operating system to easily locate an offset within a file. For example, as discussed above, UNIX treats files as a sequence of bytes. If each physical block is say 512 bytes, then the operating system packs and unpacks 512 bytes of logical records into physical blocks.

File access is always in terms of blocks. The logical size, physical size and packing technique determine the number of logical records that can be packed into one physical block. The mapping is usually done by the operating system. But since the total file size is not always an exact multiple of the block size, the last physical block containing logical records is not full. Some part of this last block is always wasted. On an average half a block is wasted. This is termed internal fragmentation. Larger the physical block size, greater is the internal fragmentation. All file systems do suffer from internal fragmentation. This is the penalty paid for easy file access by the operating system in terms of blocks instead of bits or bytes.

8.2 FILE ACCESS METHODS

Information is stored in files. Files reside on secondary storage. When this information is to be used, it has to be accessed and brought into primary main memory. Information in files could be accessed in many ways. It is usually dependent on an application. Access methods could be

- Sequential access
- Direct access
- Indexed sequential access

8.2.1 SEQUENTIAL ACCESS

A simple access method, information in a file is accessed sequentially one record after another. To process the ith record all the i-1 records previous to I must be accessed. Sequential access is based on the tape model that is inherently a sequential access device. Sequential access is best suited where most of the records in a file are to be processed. For example, transaction files.

8.2.2 DIRECT ACCESS

Sometimes it is not necessary to process every record in a file. It may not be necessary to process records in the order in which they are present. Information present in a record of a file is to be accessed only if some key value in that record is known. In all such cases, direct access is used. Direct access is based on the disk that is a direct access device and allows random access of any file block. Since a file is a collection of physical blocks, any block and hence the records in that block are accessed. For example, master files. Databases are often of this type since they allow query processing that involves immediate access to large amounts of information. All reservation systems fall into this category. Not all operating systems support direct access files.

Usually files are to be defined as sequential or direct at the time of creation and accessed accordingly later. Sequential access of a direct access file is possible but direct access of a sequential file is not.

8.2.3 INDEXED SEQUENTIAL ACCESS

This access method is a slight modification of the direct access method. It is in fact a combination of both the sequential access as well as direct access. The main concept is to access a file direct first and then sequentially from that point onwards. This access method involves maintaining an index. The index is a pointer to a block. To access a record in a file, a direct access of the index is made. The information obtained from this access is used to access the file. For example, the direct access to a file will give the block address and within the block the record is accessed sequentially. Sometimes indexes may be big. So a hierarchy of indexes are built in which one direct access of an index leads to info to access another index directly and so on till the actual file is accessed sequentially for the particular record. The main advantage in this type of access is that both direct and sequential access of files is possible.

8.3 DIRECTORY STRUCTURE

Files systems are very large. Files have to be organized. Usually a two level organization is done:

- The file system is divided into partitions. Default there is at least one partition. Partitions are nothing but virtual disks with each partition considered as a separate storage device.
- Each partition has information about the files in it. This information is nothing but a table of contents. It is known as a directory.

The directory maintains information about the name, location, size and type of all files in the partition. A directory has a logical structure. This is dependent on many factors including operations that are to be performed on the directory like search for file/s, create a file, delete a file, list a directory, rename a file and traverse a file system. For example, the dir, del, ren commands in MS-DOS.

8.3.1 SINGLE-LEVEL DIRECTORY

This is a simple directory structure that is very easy to support. All files reside in one and the same directory (Figure 8.1).

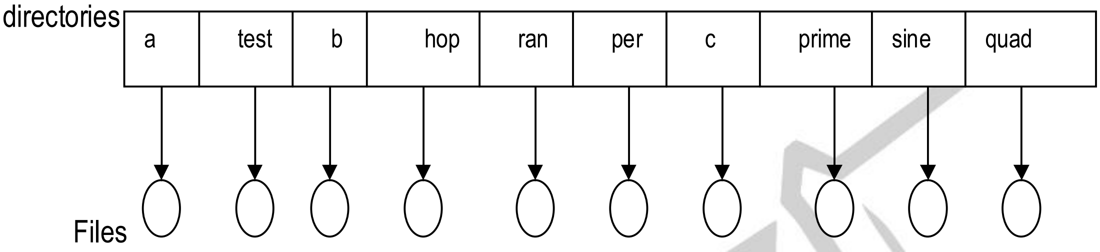


Figure 8.1: Single-level directory structure

A single-level directory has limitations as the number of files and users increase. Since there is only one directory to list all the files, no two files can have the same name, that is, file names must be unique in order to identify one file from another. Even with one user, it is difficult to maintain files with unique names when the number of files becomes large.

8.3.2 TWO-LEVEL DIRECTORY

The main limitation of single-level directory is to have unique file names by different users. One solution to the problem could be to create separate directories for each user.

A two-level directory structure has one directory exclusively for each user. The directory structure of each user is similar in structure and maintains file information about files present in that directory only. The operating system has one master directory for a partition. This directory has entries for each of the user directories (Figure 8.2).

Files with same names exist across user directories but not in the same user directory. File maintenance is easy. Users are isolated from one another. But when users work in a group and each wants to access files in another users directory, it may not be possible.

Access to a file is through user name and file name. This is known as a path. Thus a path uniquely defines a file. For example, in MS-DOS if 'C' is the partition then C:\USER1\TEST, C:\USER2\TEST, C:\USER3\C are all files in user directories. Files could be created, deleted, searched and renamed in the user directories only.

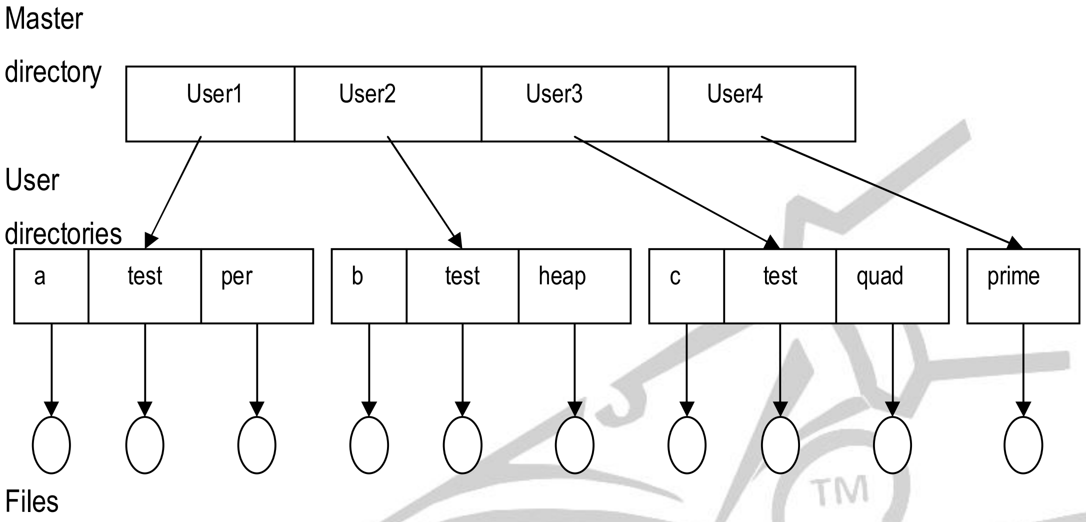


Figure 8.2: Two-level directory structure

8.3.3 TREE-STRUCTURED DIRECTORIES

A two-level directory is a tree of height two with the master file directory at the root having user directories as descendants that in turn have the files themselves as descendants. Tree-structured directories area generalization of this two-level directory structure to have a directory structure of arbitrary height (Figure 8.3). This generalization allows users to organize files within user directories into sub directories. Every file has a unique path. Here the path is from the root through all the sub directories to the specific file.

Usually the user has a current directory. User created sub directories could be traversed. Files are usually accessed by giving their path names. Path names could be either absolute or relative. Absolute path names begin with the root and give the complete path down to the file. Relative path names begin with the current directory. Allowing users to define sub directories allows for organizing user files based on topics. A directory is treated as yet another file in the directory, higher up in the hierarchy. To delete a directory it must be empty. Two options exist: delete all files and then delete the directory or delete all entries in the directory when the directory is deleted. Deletion may be a recursive process since directory to be deleted may contain sub directories.

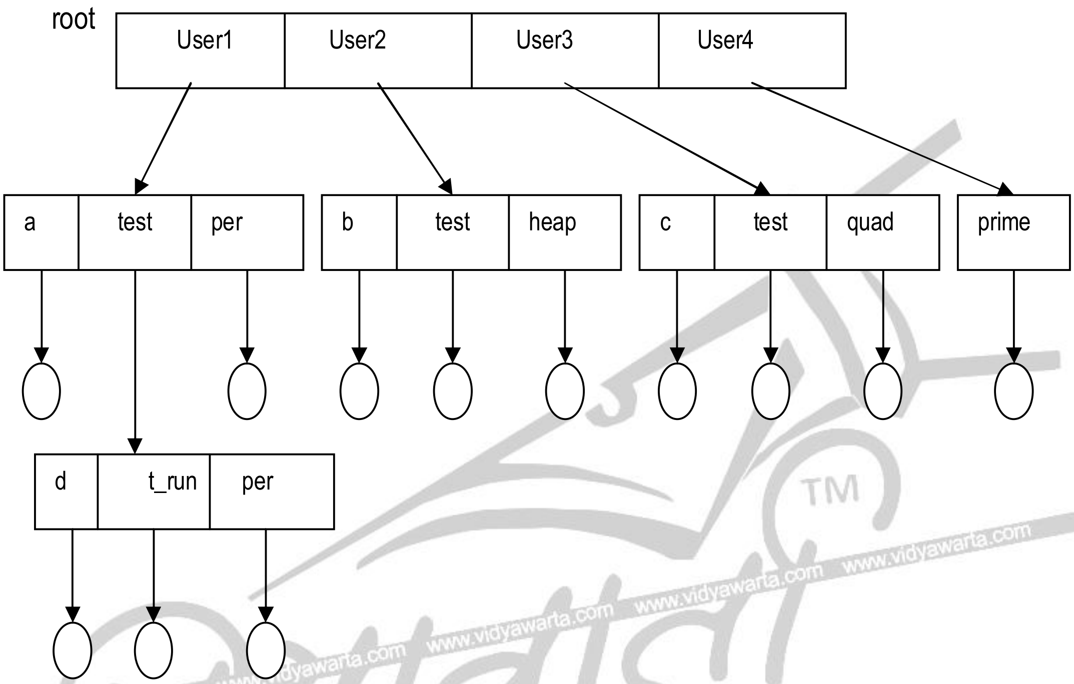


Figure 8.3: Tree-structured directory structure

8.4 SUMMARY

In this chapter the operating system as a secondary resource manager was studied. Data / information stored in secondary storage has to be managed and efficiently accessed by executing processes. To do this the operating system uses the concept of a file. A file is the smallest allotment of secondary storage. Any information to be stored needs to be written on to a file. We have studied file attributes, operations on files, types and structure of files. File access could be sequential or direct or a combination of both. We have also learnt the concept of a directory and its various structures for easy and protected access of files.

8.5 EXERCISE

1. Explain the concept of a file.
2. Discuss the different file access methods.
3. Explain how a file is structured.
4. What is the need for a directory? Explain the different directory structures.

8.6 ACTIVITY

Study the file system interface implemented in the operating systems noted.

CHAPTER 9

FILE SYSTEM IMPLEMENTATION

The file system is a mechanism for storing data and programs on-line. It resides on the secondary storage because of the two main characteristics of secondary storage namely large storage capacity and non-volatile nature.

9.1 FILE SYSTEM STRUCTURE

The disk is a very common secondary storage device. A file system has therefore to be stored on the disk. Memory transfer or I/O between the disk and the main memory is always in terms of physical blocks where each block is of a known size say ‘n’ bytes. Disks have two important characteristics that make them popular secondary devices. They are:

1. Random access or sequential access of stored files.
2. Rewriting of updated blocks in place.

9.1.1 FILE SYSTEM ORGANIZATION

The file system on the disk of an operating system allows data to be stored, searched and retrieved. One part of the file system is the user interface. This interface defines the definition of a file, file attributes, operations allowed on files and a directory structure for organizing files. The other part of the file system has algorithms and data structures to map the logical file system onto the physical storage devices.

A file system has a layered design (Figure 9.1). Each layer uses the features provided by the layer below it to provide features to the layer above it.

The lowest layer is the I/O control. It consists of device drivers and interrupt handlers for information transfer between the memory and the disk. A device driver is a translator between the file system and the actual disk hardware. The device driver takes high level instructions as input and produces low level instructions which are hardware specific as output. These low level instructions from the device driver spell out the actions to be taken on the hardware device at specified locations.

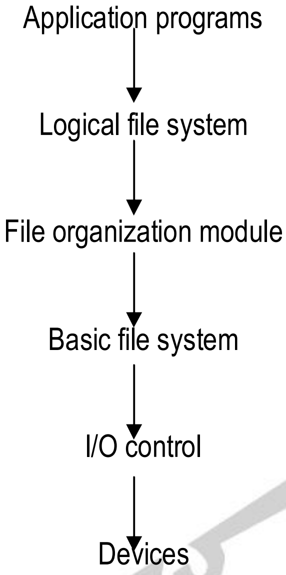


Figure 9.1: Layered file system

The next layer, the basic file system issues generic commands to the appropriate device driver to access (read / write) physical blocks on the disk.

The layer above the basic file system is the file organization module. This module has information about files, the logical records and the physical blocks. The file organization module translates / maps logical addresses to physical block addresses for the basic file system. This module also keeps track of free blocks.

The logical file system makes use of the directory structure of the file system and provides information to the file organization module given a file name. Protection and security are also part of the logical file system.

9.2 ALLOCATION METHODS

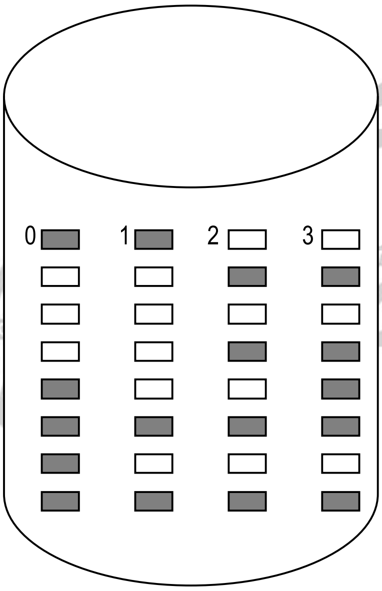
Allocation of disk space to files is a problem that looks at how effectively disk space is utilized and quickly files can be accessed. The three major methods of disk space allocation are:

- Contiguous allocation
- Linked allocation
- Indexed allocation

9.2.1 CONTIGUOUS ALLOCATION

Contiguous allocation requires a file to occupy contiguous blocks on the disk. Because of this constraint disk access time is reduced, as disk head movement is usually restricted to only one track. Number of seeks for accessing contiguously allocated files is minimal and so also seek times.

A file that is ‘n’ blocks long starting at a location ‘b’ on the disk occupies blocks b, b+1, b+2, ….., b+(n-1). The directory entry for each contiguously allocated file gives the address of the starting block and the length of the file in blocks as illustrated below (Figure 9.2).



Directory:

File	Start	Length
a.c	0	2
per	14	3
hop	19	6
b	28	4
f	6	2

Figure 9.2: Contiguous allocation

Accessing a contiguously allocated file is easy. Both sequential and random access of a file is possible. If a sequential access of a file is made then the next block after the current is accessed where as if a direct access is made then a direct block address to the i^{th} block is calculated as b+i where b is the starting block address.

One major disadvantage with contiguous allocation is to find contiguous space enough for the file. From a set of free blocks, a first-fit or best-fit strategy is adopted to find ‘n’ contiguous holes for a file of size ‘n’. But these algorithms suffer from external fragmentation. As disk space is allocated and released, a single large hole of disk space is fragmented into smaller holes. Sometimes the total size of all the holes put together is larger than the size of the file size that is to be allocated space. But the file cannot be allocated space because there is no contiguous hole of size equal to that of the file. This is when external fragmentation has occurred. Compaction of disk space is a solution to external fragmentation. But it has a very large overhead.

One other problem with contiguous allocation is to determine the space needed for a file. The file is a dynamic entity that grows and shrinks. If allocated space is just enough (a best-fit allocation strategy is adopted) and if the file grows, there may not be space on either side of the file to expand. The solution to this problem is to again reallocate the file into a bigger space and release the existing space. Another solution that could be possible if the file size is known in advance is to make an allocation for the known file size. But in this case there is always a possibility of a large amount of internal fragmentation because initially the file may not occupy the entire space and also grow very slowly.

9.2.2 LINKED ALLOCATION

Linked allocation overcomes all problems of contiguous allocation. A file is allocated blocks of physical storage in any order. A file is thus a list of blocks that are linked together. The directory contains the address of the starting block and the ending block of the file. The first block contains a pointer to the second, the second a pointer to the third and so on till the last block (Figure 9.3).

Initially a block is allocated to a file with the directory having this block as the start and end. As the file grows, additional blocks are allocated with the current block containing a pointer to the next and the end block being updated in the directory.

This allocation method does not suffer from external fragmentation because any free block can satisfy a request. Hence there is no need for compaction. Also a file can grow and shrink without problems of allocation.

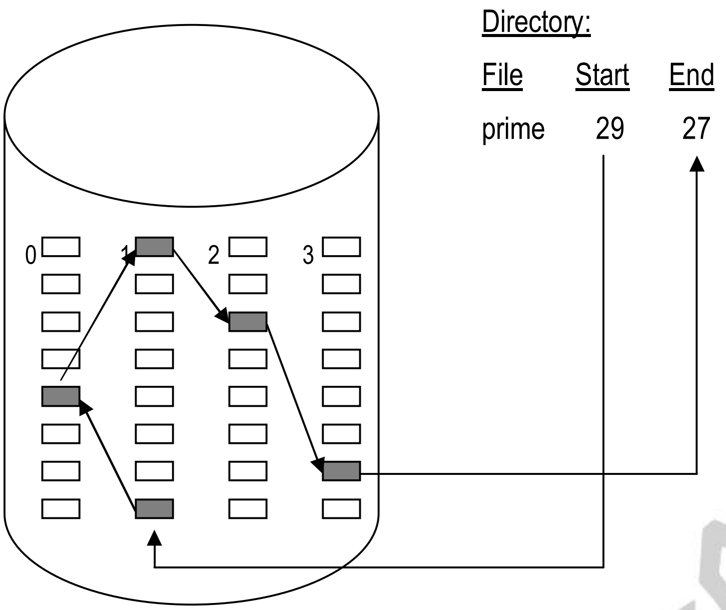


Figure 9.3: Linked allocation

Linked allocation has some disadvantages. Random access of files is not possible. To access the i^{th} block access begins at the beginning of the file and follows the pointers in all the blocks till the i^{th} block is accessed. Therefore access is always sequential. Also some space in all the allocated blocks is used for storing pointers. This is clearly an overhead as a fixed percentage from every block is wasted. This problem is overcome by allocating blocks in clusters that are nothing but groups of blocks. But this tends to increase internal fragmentation. Another problem in this allocation scheme is that of scattered pointers. If for any reason a pointer is lost, then the file after that block is inaccessible. A doubly linked block structure may solve the problem at the cost of additional pointers to be maintained.

MS-DOS uses a variation of the linked allocation called a file allocation table (FAT). The FAT resides on the disk and contains entry for each disk block and is indexed by block number. The directory contains the starting block address of the file. This block in the FAT has a pointer to the next block and so on till the last block (Figure 9.4). Random access of files is possible because the FAT can be scanned for a direct block address.

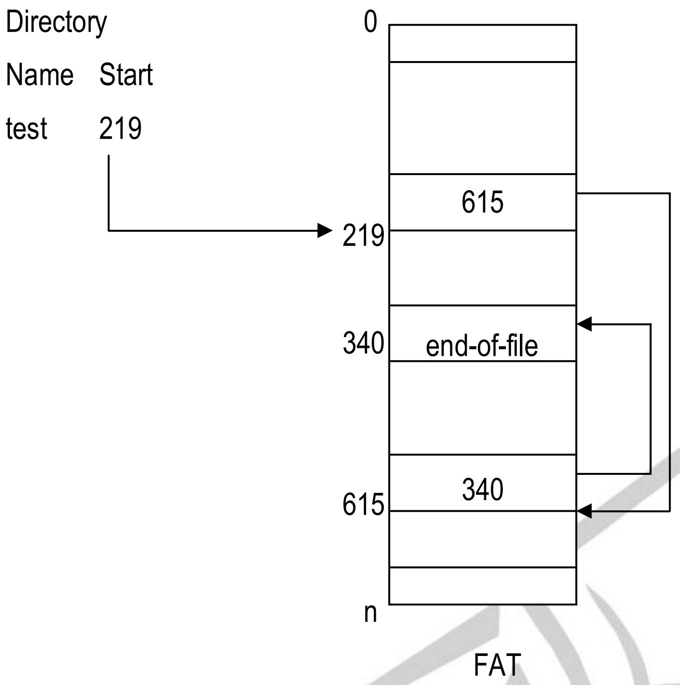


Figure 9.4: File allocation table

9.2.3 INDEXED ALLOCATION

Problems of external fragmentation and size declaration present in contiguous allocation are overcome in linked allocation. But in the absence of FAT, linked allocation does not support random access of files since pointers hidden in blocks need to be accessed sequentially. Indexed allocation solves this problem by bringing all pointers together into an index block. This also solves the problem of scattered pointers in linked allocation.

Each file has an index block. The address of this index block finds an entry in the directory and contains only block addresses in the order in which they are allocated to the file. The i^{th} address in the index block is the i^{th} block of the file (Figure 9.5). Here both sequential and direct access of a file are possible. Also it does not suffer from external fragmentation.

Indexed allocation does suffer from wasted block space. Pointer overhead is more in indexed allocation than in linked allocation. Every file needs an index block. Then what should be the size of the index block? If it is too big, space is wasted. If it is too small, large files cannot be stored. More than one index blocks are linked so that large files can be stored. Multilevel index blocks are also used. A combined scheme having direct index blocks as well as linked index blocks has been implemented in the UNIX operating system.

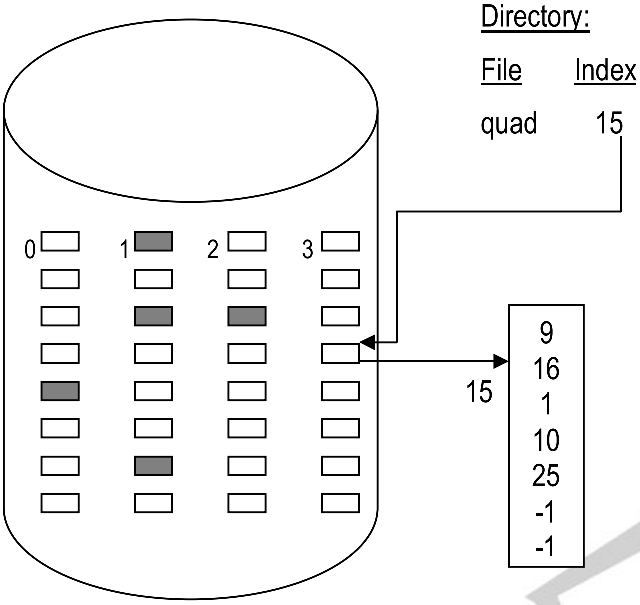


Figure 9.5: Indexed allocation

9.2.4 PERFORMANCE COMPARISON

All the three allocation methods differ in storage efficiency and block access time. Contiguous allocation requires only one disk access to get a block whether it be the next block (sequential) or the ith block (direct). In the case of linked allocation, the address of the next block is available in the current block being accessed and so is very much suited for sequential access. Hence direct access files could use contiguous allocation and sequential access files could use linked allocation. But if this is fixed then the type of access on a file needs to be declared at the time of file creation. Thus a sequential access file will be linked and cannot support direct access. On the other hand a direct access file will have contiguous allocation and can also support sequential access, the constraint in this case is making known the file length at the time of file creation. The operating system will then have to support algorithms and data structures for both allocation methods. Conversion of one file type to another needs a copy operation to the desired file type.

Some systems support both contiguous and linked allocation. Initially all files have contiguous allocation. As they grow a switch to indexed allocation takes place. If on an average files are small, than contiguous file allocation is advantageous and provides good performance.

9.3 FREE SPACE MANAGEMENT

The disk is a scarce resource. Also disk space can be reused. Free space present on the disk is maintained by the operating system. Physical blocks that are free are listed in a free-space list. When a file is created or a file grows, requests for blocks of disk space are checked in the free-space list and then allocated. The list is updated accordingly. Similarly freed blocks are added to the free-space list. The free-space list could be implemented in many ways as follows:

9.3.4 BIT VECTOR

A bit map or a bit vector is a very common way of implementing a free-space list. This vector 'n' number of bits where 'n' is the total number of available disk blocks. A free block has its corresponding bit set (1) in the bit vector where as an allocated block has its bit reset (0).

Illustration: If blocks 2, 4, 5, 9, 10, 12, 15, 18, 20, 22, 23, 24, 25, 29 are free and the rest are allocated, then a free-space list implemented as a bit vector would look as shown below:

00101100011010010010101111000100000………

The advantage of this approach is that it is very simple to implement and efficient to access. If only one free block is needed then a search for the first '1' in the vector is necessary. If a contiguous allocation for 'b' blocks is required, then a contiguous run of 'b' number of 1's is searched. And if the first-fit scheme is used then the first such run is chosen and the best of such runs is chosen if best-fit scheme is used.

Bit vectors are inefficient if they are not in memory. Also the size of the vector has to be updated if the size of the disk changes.

9.3.5 LINKED LIST

All free blocks are linked together. The free-space list head contains the address of the first free block. This block in turn contains the address of the next free block and so on. But this scheme works well for linked allocation. If contiguous allocation is used then to search for ‘b’ contiguous free blocks calls for traversal of the free-space list which is not efficient. The FAT in MS-DOS builds in free block accounting into the allocation data structure itself where free blocks have an entry say –1 in the FAT.

9.3.6 GROUPING

Another approach is to store ‘n’ free block addresses in the first free block. Here (n-1) blocks are actually free. The last nth address is the address of a block that contains the next set of free block addresses. This method has the advantage that a large number of free block addresses are available at a single place unlike in the previous linked approach where free block addresses are scattered.

9.3.7 COUNTING

If contiguous allocation is used and a file has freed its disk space then a contiguous set of ‘n’ blocks is free. Instead of storing the addresses of all these ‘n’ blocks in the free-space list, only the starting free block address and a count of the number of blocks free from that address can be stored. This is exactly what is done in this scheme where each entry in the free-space list is a disk address followed by a count.

9.4 DIRECTORY IMPLEMENTATION

The two main methods of implementing a directory are:

- Linear list
- Hash table

9.4.4 LINEAR LIST

A linear list of file names with pointers to the data blocks is one way to implement a directory. A linear search is necessary to find for a particular file. The method is simple but the search is time consuming. To create a file a linear search is made to look for the existence of a file with the same file name and if no such file is found the new file created is added to the directory at the end. To delete a file a linear search for the file name is made and if found allocated space is released. Every time making a linear search consumes time and increases access time that is not desirable since a directory information is frequently used. A sorted list allows for a binary search that is time efficient compared to the linear search. But maintaining a sorted list is an overhead especially because of file creations and deletions.

9.4.5 HASH TABLE

Another data structure for directory implementation is the hash table. A linear list is used to store directory entries. A hash table takes a value computed from the file name and returns a pointer to the file name in the linear list. Thus search time is greatly reduced. Insertions are prone to collisions that are resolved. The main problem is the hash function that is dependent on the hash table size. A solution to the problem is to allow for chained overflow with each hash entry being a linked list. Directory lookups in a hash table are faster than in a linear list.

9.5 SUMMARY

This chapter deals with actual implementation of a file. Space allocation for files on secondary storage is important for secondary resources to be efficiently managed as files grow and shrink over a period of time which calls for additional file space or utilization of released file space. Different allocation methods were studied and their performance evaluated. Usually a mix of more than one method is practical. Knowledge of free space is to be made available for any allocation scheme. Different ways of making it known were learnt.

9.6 EXERCISE

1. Describe the various file allocation methods in disk-based systems.
2. List and explain the different methods for free-space management.

9.6.4 ACTIVITY

Study the different file allocation methods implemented and also methods for free-space management in the operating systems noted.

CHAPTER 10

SECONDARY STORAGE STRUCTURE

Disks provide the bulk of secondary storage. All disks can be visualized as a large one-dimensional array of logical blocks. The block is the basic unit of data transfer. The size of all blocks is the same usually 512 bytes but an option exists for choosing a different block size.

A disk consists of sectors, tracks and cylinders. The one-dimensional array of logical blocks is mapped on through all sectors in a track, then through all sectors through a cylinder and through all cylinders sequentially. This mapping converts a logical block address or number into a physical block address consisting of a cylinder number, a track number within that cylinder and a sector number within the track.

10.1 DISK SCHEDULING

Disk access time has two main parts:

- Seek time
- Latency time

Seek time is the time needed for the disk arm to move the heads to the cylinder containing the desired sector. The latency time is the time required for the disk to rotate the desired sector under the disk head. The disk bandwidth is the ratio of the total number of bytes transferred to the total time taken between the first request for service and the completion of the last transfer. Execution of I/O requests in proper order improves disk performance both in terms of access time as well as bandwidth.

An I/O to or from the disk initiates a system call to the operating system. It consists of several pieces of information like:

- Whether the operation is input or output
- The disk address for transfer
- The memory address for transfer

- The number of bytes to be transferred

Since the operating system is working in a multiprogramming environment there may be many requests for disk I/O. The disk therefore maintains a disk queue and services pending requests in some order as soon as the controller is available. The following are a few disk scheduling algorithms:

10.1.1 FCFS SCHEDULING

The first-cum-first-served (FCFS) disk scheduling algorithm is the simplest of all the algorithms. It is fair to all requests but does not guarantee the fastest service.

Illustration: Consider a disk queue with the following requirements for I/O to blocks on cylinders:

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

Let the disk head initially be at cylinder 53. Then according to the FCFS scheduling algorithm the disk head will move from 53 to 98 and then to 183, 37, 122, 14, 124, 65 and finally to 67 in that order for a total head movement of 640 cylinders (Figure 10.1).

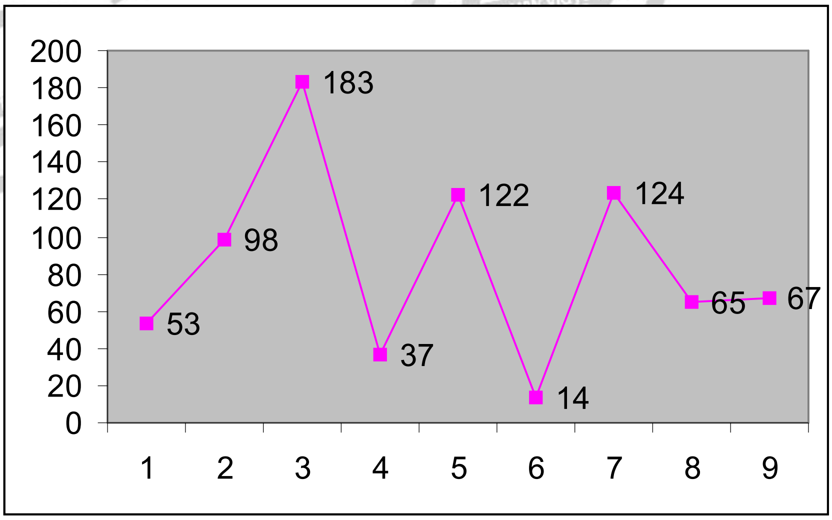


Figure 10.1: FCFS disk scheduling

The disadvantage with FCFS disk scheduling algorithm is the large wild swing of the disk head that is noticed, for example while servicing requests for 122 followed by 14 and then 124. If cylinders close together could be serviced together then access time could be improved.

10.1.2 SSTF SCHEDULING

To overcome the disadvantage in FCFS algorithm, I/O requests close to the disk head need to be serviced together before moving the head far away for other requests. This assumption is the basis for the shortest-seek-time-first (SSTF) algorithm. The algorithm selects the next request for service as that request which will involve the minimum seek time from the current disk position.

Illustration: Consider a disk queue with the following requirements for I/O to blocks on cylinders:

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

Let the disk head initially be at cylinder 53. Then according to the SSTF scheduling algorithm the disk head will move from 53 to 65 and then to 67, 37, 14, 98, 122, 124 and finally to 183 in that order for a total head movement of 236 cylinders (Figure 10.2).

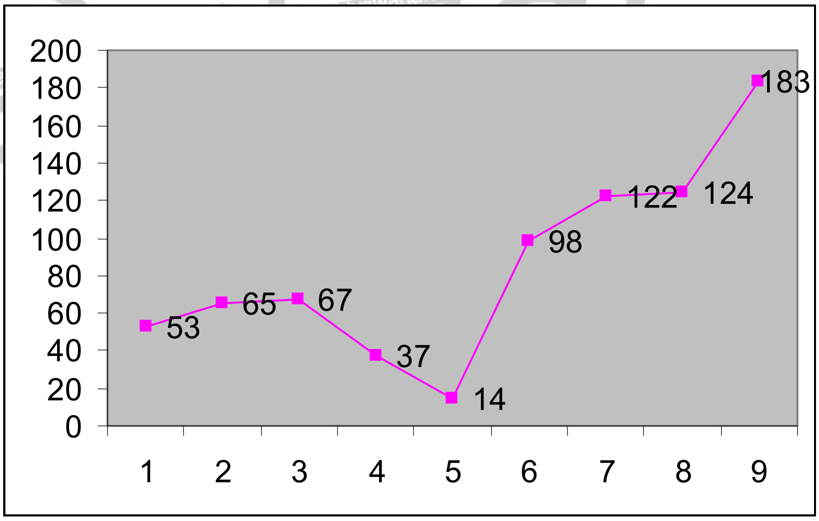


Figure 10.2: SSTF disk scheduling

The SSTF disk scheduling algorithm results in total head movement of only 236 cylinders for the same set of I/O requests thus giving a substantial improvement in performance.

The main disadvantage of SSTF algorithm is starvation. The disk queue is a dynamic structure that grows and shrinks as requests wait and get serviced. In this process if requests for I/O start coming in that are close to the current head position then those requests waiting in the queue that have come in earlier keep waiting for service because they are far away from the disk head. This is the problem of starvation.

10.1.3 SCAN SCHEDULING

The problem of starvation in the SSTF disk scheduling algorithm is overcome in the SCAN algorithm to a certain extent. In the SCAN algorithm the disk arm moves from one end of the disk towards the other and then again in the reverse direction. As it moves along it services requests. The head thus scans the disk back and forth for requests and hence the name SCAN algorithm.

The SCAN algorithm is based no the principle of the elevator and sometimes called the elevator algorithm. The disk arm scans the disk just like the movement of an elevator in a building.

Illustration: Consider a disk queue with the following requirements for I/O to blocks on cylinders:

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

Let the disk head initially be at cylinder 53 and move towards 0. Then according to the SCAN scheduling algorithm the disk head will move from 53 to 37 and then to 14, 0, 65, 67, 98, 122, 124, and finally to 183 in that order (Figure 10.3).

A fresh request just in front of the head will get serviced almost immediately while those at the back will have to wait for the head to start a scan in the reverse direction. Considering a uniform distribution of requests, the density of requests in front of the head when the head reaches one end of the disk and reverses direction is low since the head has just then serviced those cylinders. On the other hand the density is more at the other end with those requests having waited for a longer period of time. The wait time is thus not uniform. The next algorithm is based on density of requests and overcomes this drawback of the present algorithm.

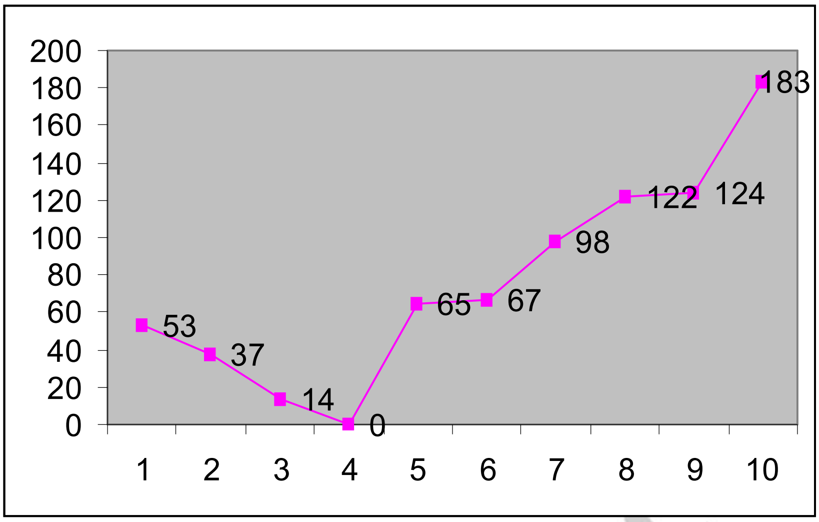


Figure 10.3: SCAN disk scheduling

10.1.4 C-SCAN SCHEDULING

Circular SCAN or C-SCAN scheduling algorithm is a variant of the SCAN algorithm and provides a more uniform wait time. In this algorithm the disk head scans / traverses the disk as in the previous case servicing requests as it moves along but the direction of head movement is always in one direction. The cylinders can thus be visualized as a circular list with the last one wrapped on to the first.

Illustration: Consider a disk queue with the following requirements for I/O to blocks on cylinders:

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

Let the disk head initially be at cylinder 53 and move towards the end (200 cylinders from 0 to 199). Then according to the C-SCAN scheduling algorithm the disk head will move from 53 to 65 and then to 67, 98, 122, 124, 183, 199, 0, 14 and finally to 37 in that order (Figure 10.4).

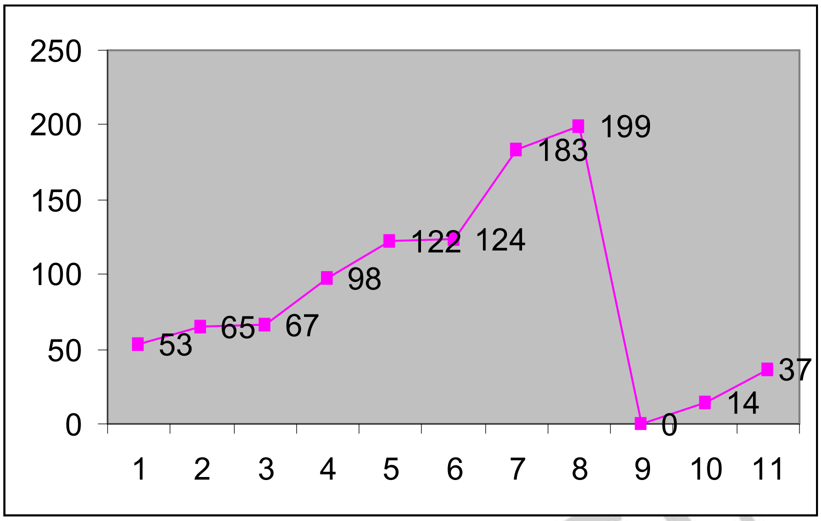


Figure 10.4: C-SCAN disk scheduling

10.1.5 LOOK SCHEDULING

In the SCAN algorithm the disk head moves from one end of the disk to another. But there may not be any requests till the very end in a given direction. A change is thus brought about in the SCAN algorithm resulting in the LOOK algorithm. The disk head scans the disk for requests only as far as the final request in each direction at which point it reverses direction.

Illustration: Consider a disk queue with the following requirements for I/O to blocks on cylinders:

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

Let the disk head initially be at cylinder 53 and move towards the end. Then according to the LOOK scheduling algorithm the disk head will move from 53 to 65 and then to 67, 98, 122, 124, 183, 37 and finally to 14 in that order (Figure 10.5).

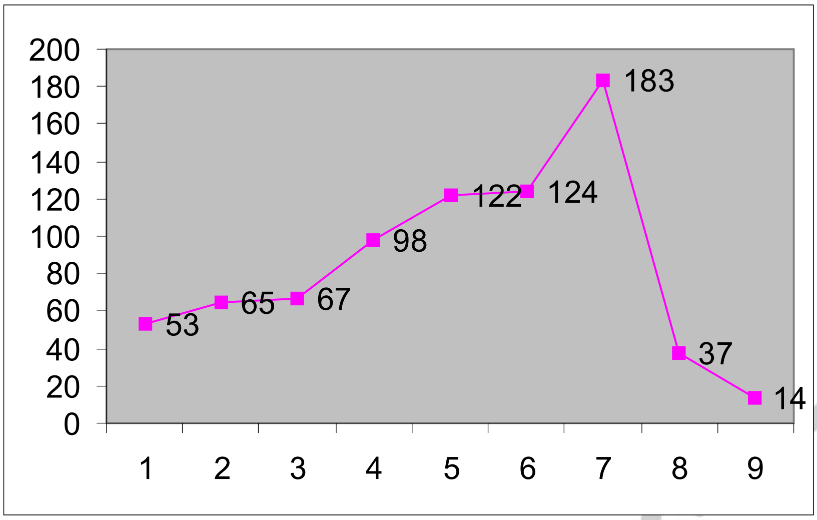


Figure 10.5: LOOK disk scheduling

10.1.6 C-LOOK SCHEDULING

Similar to the C-SCAN algorithm the C-LOOK scheduling has the change of the LOOK algorithm incorporated. Hence the disk head scans for requests only in one direction that is bounded by the final pending request at one end and the smallest pending request at the other.

Illustration: Consider a disk queue with the following requirements for I/O to blocks on cylinders:

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

Let the disk head initially be at cylinder 53 and move towards the end. Then according to the C-LOOK scheduling algorithm the disk head will move from 53 to 65 and then to 67, 98, 122, 124, 183, 14, and finally to 37 in that order (Figure 10.6).

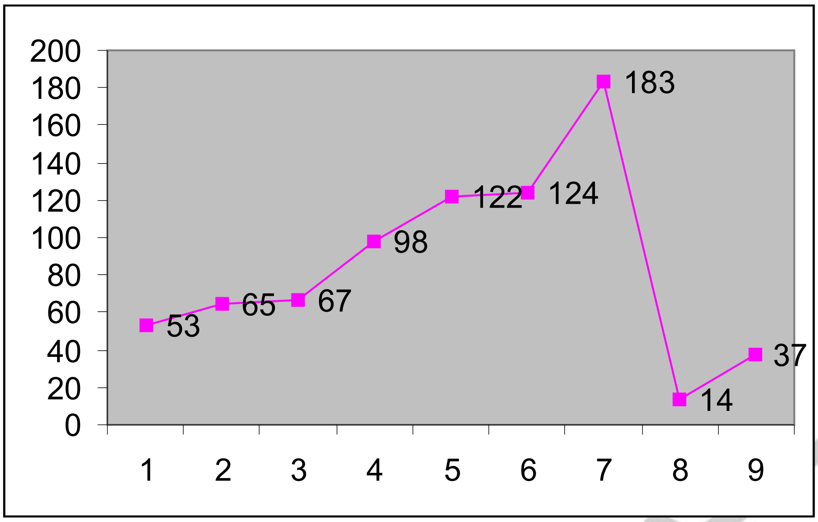


Figure 10.6: C-LOOK disk scheduling

10.2 SELECTION OF A DISK SCHEDULING ALGORITHM

Having a choice of many disk scheduling algorithms, choosing the best one can be difficult. A natural choice is the SSTF algorithm. LOOK and C-LOOK which are practical versions of the SCAN and C-SCAN respectively do better when load is heavy since they do not suffer from starvation problem. Performance also depends on the number and types of requests. For example if there is only one request or requests come in at a very slow pace, then all algorithms behave like the FCFS algorithm.

Requests for disk service can closely be related to file allocation methods. Program reading contiguously allocated file will generate requests that are close to one another thus resulting in limited head movement. On the other hand a file with linked or indexed allocation may request for disk blocks that are widely scattered resulting in greater head movement.

Locations of directory structure and index blocks also play an important role. All files need to be opened before use and to open a file, the directory structure needs to be searched for an entry of that file. If the directory is on one cylinder and the actual file data on another cylinder far away, then the disk head movement is considerably more. In the worst case the head moves the entire width of the disk if the directory is in the first and the data on the last cylinder. If the directory is held in the middle of the disk then on an average the head moves one half width of the disk. Bringing the directory and index blocks into cache can help reduce head movement.

Because of these complexities and many more disk scheduling algorithms form a separate module of the operating system so as to enable the existing disk scheduling algorithm to be replaced by a better one when available. Either the SSTF or LOOK is a choice for default.

10.3 SUMMARY

In this chapter we have studied the components that make up total access time while data / information is being accessed from a disk. To reduce access time, the disk needs to be scheduled. Different disk scheduling algorithms were analyzed.

10.4 EXERCISE

1. Describe the various disk scheduling algorithms.
2. List the available disk scheduling algorithms. Compare them based on their merits and demerits.

10.5 ACTIVITY

Study the disk scheduling algorithms implemented for the operating systems noted.

CHAPTER 11

GRAPHICAL USER INTERFACE AND THE OPERATING SYSTEM

Poor human-computer interface has always been a problem area in using computers. However good a user interface is, there is always something that is wanting according to users who use it and thus a scope for improvement.

In the earlier days when computers made a beginning, they were not very powerful. They were also intended for use by users who were computer literates. Thus only cryptic commands were provided as a user interface to the operating system. The problem with these commands was that the users had to remember them and know exactly what each one stood for. The advent of personal computers brought about a new paradigm. Computer users were no longer only computer literates and so users no longer wanted the cryptic human-computer interface. The interface ought to become more user-friendly. As the saying goes ‘A picture is worth a thousand words’, cryptic commands gave way to graphical representation. Such an interface is called a graphical user interface (GUI). For example, MS-Windows and X-Windows.

11.1 WINDOWING TECHNOLOGY

Computers of today are very powerful. They can do more than one job at a time. With any windows product, the screen can be split into many partitions of different sizes. Each of these partitions is called a window. Hence the name windowing technology. Each window can run an application and is independent of others. Characteristics of a window are title, borders, work areas and command area. Users running an application in a window can configure it to have specific settings. Windows can overlap. The windowing software reserves separate areas in memory for different windows and allow execution on them as if each were executed on a separate terminal. But screen display shows all windows simultaneously. A user can use only one input device at a time. Then where does the user input go? Or in which window will user interaction show up? The user has a choice of selecting a window and bringing it on the top of the screen. The window on top is said to be in-focus and hence will receive user interaction.

11.2 GUI

Graphical user interface came into existence as a substitute to the less user-friendly cryptic commands. Also cryptic commands were difficult to remember especially when commands had several options. In a GUI, commands are grouped into various levels of hierarchy and when the user selects a group commands belonging to that group are displayed. This makes a GUI user-friendly and the user runs an application without having to know about the computer and its working.

An important feature supported by all GUI-based applications is the 'HELP'. HELP assists the user in getting to know everything about the application. A fundamental concept of windowing environment is what is called 'event driven programming'. An application running in a windows environment responds to each command from the users and waits for the next. Events could be a mouse click, mouse movement, keyboard input and so on. The mouse is a very handy input device used in a windowing environment since it gives the user ease in navigation.

11.3 COMPONENTS OF A GUI

Some common components of a GUI environment are:

- Menu bars with pull down menus
- Scroll bars
- Controls
- Dialogue boxes
- Feedback pointers
- Icons

Menu bars appear at the top of a window just under the title bar. Common menu bar options are File, Edit, View and Help. When the user clicks on any one of the options on the menu bar, a pull down menu appears. This is a box where specific actions are listed. It lists the actions and associated combination keys that will perform the same action from the keyboard. Any one action can be selected from this menu. Each option on the menu bar displays only grouped actions pertaining to the selected menu.

Scroll bars help look at information not currently visible in the window. There are horizontal and vertical scroll bars. The scroll bar has scroll area with a slider box and arrows at each end. The slider box gives information about the position of the visible part of the object in a window in relation to the entire object.

Many different controls are used in a GUI that enable users to select presented information in a window. Controls may be in the form of buttons (push buttons, radio buttons, option buttons) or boxes (list boxes, combo boxes, entry boxes).

A dialogue box is a window for application-user interaction. It can also be used only to put out a message for the user. Dialogue boxes usually combine some controls to interact with a user.

Feedback for an action performed is provided in two ways:

- Hourglass pointer
- Progress indicator

These are visible when user request is not immediately completed but takes a finite amount of time that is noticeable by the user. The hourglass pointer shows up when the system is performing simple operations. A progress indicator is visible when the request is expected to take much longer time and indicates the percentage of progress made like for example downloading a file.

A very important GUI object is an icon. It is a graphical representation of an application or a utility. A good icon should be able to help identify an application. For example, mailbox, calculator, MS-Word, etc.

11.4 REQUIREMENTS OF A WINDOWS-BASED GUI

Some basic requirements of a windows-based GUI are as follows:

- Consistency
- Direct manipulation
- Flexibility
- Explicit destruction

All applications within a windowing environment must be consistent. This implies that similar controls work similarly and are used similarly. Placement of control buttons and boxes should be consistent across all applications. An action needs objects to act on. So selection of action and objects are necessary. For example, delete action needs an object say a file. The sequence of action and selection must be consistent. That is, in one application selection of objects to be done before action and in another action to be selected before objects should not happen. Menus have to be presented in an orderly manner. Actions and objects that cannot be selected should not be presented or should be de-emphasized.

Direct manipulation allows user to control actions. User also gets a feedback on actions performed.

The interface should be flexible in many ways. User specified settings are better. They should be able to change configurations later. A very good example for this could be mouse configuration by a left-handed or a right-handed user. Also each user has his / her own liking for colors and borders and a change to suit user requirement encourages them to work and learn more. Another level of flexibility would be to allow multiple ways of performing the same function. For example, an experienced user may want to use function keys and keyboard keys rather than go through a GUI select, some may prefer to enter file names rather hunt for them and so on.

An action that is irreversible and has negative consequences such as file deletion has to provide for explicit confirmation. This would minimize user apprehension and give him confidence to use the application for his work.

11.5 SUMMARY

One of the primary goals of an operating system is to provide a good user interface. This chapter has stressed the need for a good user-friendly graphical interface. We have also discussed the various components that make a good GUI and its relevance in applications.

11.6 EXERCISE

1. What is the need for a good user-friendly GUI?
2. Explain in brief the windowing technology.

3. List and explain the different components of a GUI.
4. What are the requirements for a good windows-based GUI?

11.7 ACTIVITY

Study the GUIs available on the operating systems noted.

CHAPTER 12

INTER-PROCESS COMMUNICATION

Processes need to communicate with each other. Proper synchronization and use of data residing in shared memory is absolutely necessary for inter-process communication (IPC). The producer-consumer problem is considered for illustration.

12.1 THE PRODUCER-CONSUMER PROBLEM

Illustration 1: Let there be multiple users at different terminals each one running the same program but different processes. The user is supposed to enter a number against a prompt and on entry the program writes the entered number into a shared memory location. All these user processes are producer processes. Let there be a consumer process that prints the number present in the shared memory location. Here IPC between producer processes and the consumer process is through the shared memory location / variable.

Illustration 2: UNIX provides a facility called pipe. A file (usually 4096 bytes long) which resides in memory replaces the single shared memory variable used in the previous case. A producer process that wants to communicate with a consumer process writes bytes into this shared file called a pipe. The consumer process for which it is intended reads this pipe in the sequence in which it is written. The output of one process thus becomes the input of the next and so on. For example, to execute a query on a particular selection of a database, a process say P_1 selects the required records from the database and gives them to a process say P_2 through a pipe to execute the required query. UNIX internally manages the pipe and the user is not aware of it. The pipe here forms a vehicle for IPC.

Illustration 3: A spooler with an operating system is yet another example of a producer-consumer problem and IPC. The operating system maintains a shared list of files to be printed from which the spooler process picks and prints one file at a time. This shared list of files is the medium for IPC. A process wanting to print a file just adds the file name to the end of the shared list.

It is usually necessary to synchronize two or more processes involved in IPC. The common shared memory forms the basis for synchronization. For example, A consumer process P_2 can use data / information in the common memory only after a producer process P_1 has written into it. All examples illustrated above are examples of synchronization and IPC where in the consumer processes can and should execute only after the producer processes have written into the shared memory. Also synchronization should take care to see that one producer process does not overwrite the contents written by another producer. For example, in illustration 1, another user should not overwrite a number written by one user for printing before it is printed. Thus synchronization is an important aspect of IPC.

Use of flags to synchronize processes could be one solution. For example, a producer process can write data into a shared location only when the flag is reset (flag = 0) and a consumer process can use data in the shared location only when flag is set (flag = 1). When a process has written data into the shared location then it sets the flag so that no other process will be able to overwrite it as shown below:

```
Begin                          Begin
  While flag = 1 wait            While flag = 0 wait
  Output the number              Print the number
  Set flag = 1                   Set flag = 0
End                            End

Producer                       Consumer
```

All looks fine, but a certain sequence of events brings about a problem. Consider the following scenario:

Initially flag = 0. A producer process P_1 outputs a number into the shared location. At this moment P_1's time slice expires and P_1 moves into the ready state while the flag has still not been set, that is flag = 0. Process P2 is scheduled and since flag = 0, outputs a number into the shared location thus overwriting the number written by P_1. Hence the problem.

A delay in setting the flag has brought about the problem. The above sequence of instructions could thus be modified as follows:

```
Begin                          Begin
  While flag = 1 wait            While flag = 0 wait
  Set flag = 1                   Set flag = 0
  Output the number              Print the number
End                            End

Producer                       Consumer
```

Now look at the following scenario:

Initially flag = 0. Process P_1 sets flag = 1 and its time slice expires. Process P_2 is scheduled. But since flag = 1, it does not output anything. Also at this time the shared buffer is considered empty (in a practical situation it is not empty). Next if a consumer process P_3 is scheduled, it will reset the flag (flag = 0) and print what ever is there in the shared location. But what is printed is wrong and is not of any producer process. The consumer process P_3 has printed the contents of the shared location before it is written.

Thus just using flags and setting their values will not bring about proper synchronization. It may work in a few instances but is not a general solution.

To analyze the problem, let us look at what is a critical section. That portion of any program code that accesses a shared resource (a shared memory location or a shared file, etc) is called a critical section or a critical region. For example, Instructions to set the flag and output the number in the producer and instructions to set the flag and print the number in the consumer from the critical section code because all of them access shared variables. The problem in the earlier two solutions where flags were being set is called the race condition. When two or more processes are accessing (read / write) some shared data and the output is dependent upon which process runs precisely when, then the situation is called race condition. Race condition is undesirable as results are unpredictable.

What are the causes for race condition? The answer to this question is hidden in our previous solutions. If more than one process is in its critical section at the same time then race condition occurs. For example, in our first solution with flags, both P_1 and P_2 are in their respective critical sections of the same program at the same time and in the second solution both P_1 and P_3 are in their respective critical sections in different programs at the same time.

If only one process is allowed to enter its critical section at a given time, then the problem of race condition will not occur. For example, when P_1 has entered its critical section and timed out without getting out of its critical section, no other process must be allowed to enter into its critical section. If this is taken care of then race condition is avoided. In technical terms, what is wanted is mutual exclusion. But achieving mutual exclusion could be expensive. An operating system that implements mutual exclusion at the cost of speed is not a good solution. A good solution to mutual exclusion should satisfy the following conditions:

- No two processes to be inside their critical sections at the same time (mutual exclusion).
- Solution is more preferable in software as it aids portability.
- No process to wait for a long time before it enters its critical section (indefinite postponement).
- Solutions not to be based on any assumptions like number of CPUs, processor speeds, etc.
- Any process outside its critical section should not be able to prevent another from entering its critical section.

12.2 SOLUTIONS

12.2.1 INTERRUPT DISABLING / ENABLING

Interrupts can be enabled / disabled. A process switch happens due to an interrupt generated by the timer hardware when a time slice allotted to a process expires. A simple solution is to disable all interrupts before any process that enters its critical section. If this is done the time slice of the process in critical section will not expire till it has completed execution of the critical section after which the interrupts can be enabled. Therefore no process will be able to enter into its critical section simultaneously. This is illustrated below:

```
- - - - - - - - - - -
- - - - - - - - - - -
Disable interrupts
Critical section instructions
Enable interrupts
- - - - - - - - - - -
- - - - - - - - - - -
```

But playing around with interrupts is extremely dangerous such as a situation where a process executing in its critical section goes into an infinite loop because of a bug.

12.2.2 LOCK-FLAG

Use of a lock-flag could be one alternative. If a process is its critical section then the lock-flag is set to ‘N’ and if no process is in its critical section then the lock-flag is set to ‘F’. A producer-consumer problem can use this flag as below:

```
Begin
  While lock-flag = N wait
  Set lock-flag = N
  Critical section instructions
  Set lock-flag = F
End
```

Use of this flag is to ensure that no two processes be in their critical sections simultaneously. The lock-flag thus indicated whether any critical section is being executed or not. But since the lock-flag is a shared variable, race condition is not solved altogether. The following scenario will illustrate the problem:

Process P_1 finds lock-flag = F wants to set it to ‘N’ but before executing it times out at which time P_2 is scheduled. P_2 also finds lock-flag = F and so set lock-flag = n. P_2 enters its critical section and times out. P_1 is scheduled again and starts off from where it left by setting lock-flag = N and then enters into its critical section. A race condition has occurred.

12.2.3 PRIMITIVES FOR MUTUAL EXCLUSION

Mutual exclusion primitives / instructions are needed to guarantee mutual exclusion. Two primitives Begin-critical-section and End-critical-section can be used. They define the boundary of a critical section of a process as illustrated below:

```
Begin                               Begin
  While flag = 1 wait                 While flag = 0 wait
  Begin-critical-section              Begin-critical-section
        Output number                       Print number
        Set flag = 1                        Set flag = 0
  End-critical-section                End-critical-section
End                                 End

Producer                            Consumer
```

Any process that encounters Begin-critical-section will not enter into its critical section as there is already one process in critical section. Thus mutual exclusion is guaranteed. Even though a process has timed out leaving the flag = 0, a second process that is scheduled will not enter into its critical section because of a Begin-critical-section encountered before entering its critical section.

A consumer process keeps waiting and is in a loop to check a flag until it is set. This is called busy waiting. This consumer is also like any other process and contends for CPU computing resources when it is ready. The process is not blocked for an I/O but for a flag to change state. A process busy waiting thus wastes the allotted computing resources. If this flag operation can also be treated as an I/O then a consumer process busy waiting can be blocked thus making available resources for other processes.

12.2.4 ALTERNATING POLICY

An alternating policy is an attempt to implement mutual exclusion primitives discussed earlier. This is applicable to only two processes where the CPU alternates between them. A shared variable contains the process-id of the process to be executed as illustrated below:

```
Begin
  While process-id = $P_2$ wait
        Critical section instructions of $P_1$
        Set process-id = $P_2$
End
```

Process P_1

```
Begin
  While process-id = $P_1$ wait
        Critical section instructions of $P_2$
        Set process-id = $P_1$
End
```

Process P_2

Mutual exclusion is guaranteed and the instructions just before and after the critical section set of instructions in any process form the mutual exclusion primitives. Even though mutual exclusion is guaranteed the scheme fails if there are more than two processes.

12.2.5 PETERSON'S ALGORITHM

This algorithm is also applicable only on two processes. It uses three variables to ensure mutual exclusion. They are P_1-to-enter, P_2-to-enter and chosen-process. The variable chosen-process takes values P_1 or P_2 depending on which process is chosen. P_1-to-enter and P_2-to-enter are two flags which take values yes or no depending on whether P_1 / P_2 want to enter its respective critical section or not. The value in these flags enables each process to know about whether the other intends to execute its critical section as shown below:

```
Begin
        $P_1$-to-enter = Y
        Chosen-process = $P_2$
        While ($P_2$-to-enter = Y and chosen-process = $P_2$) wait
                Critical section instructions of $P_1$
                $P_1$-to-enter = N
End
```

Process P_1

```
Begin
          $P_2$-to-enter = Y
          Chosen-process = $P_1$
          While ($P_1$-to-enter = Y and chosen-process = $P_1$) wait
                    Critical section instructions of $P_2$
                    $P_2$-to-enter = N
End
```

Process P_2

The algorithm guarantees mutual exclusion and is simple. But it is applicable only on two processes. Also treating busy waiting processes as blocked is not implemented and hence resources are wasted when such processes are scheduled.

12.2.6 HARDWARE ASSISTANCE

Checking for mutual exclusion is also possible through hardware. Special instructions called Test and Set Lock (TSL) is used for the purpose. An important feature is that the set of instructions used for this purpose is indivisible, that is, they cannot be interrupted during execution. The instruction has the format 'TSL ACC, IND' where ACC is an accumulator register and IND is a memory location used as a flag. Whenever the instruction is executed, contents of IND are copied to ACC and IND is set to 'N' which implies that a process is in its critical section. If the process comes out of its critical section IND is set to 'F'. Hence the TSL instruction is executed only if IND has a value 'F' as shown below:

```
Begin
          - - - - - - -
          Call Enter-critical-section
          Critical section instructions
          Call Exit-critical-section
          - - - - - - -
```

Call Enter-critical-section executes the TSL instruction if IND = F else waits. Call Exit-critical-section sets IND = F. any process producer / consumer can be handled using the above algorithm. Demerits of the algorithm include the use of special hardware that restricts portability.

12.2.7 SEMAPHORES

Semaphores represent an abstraction of many ideas in mutual exclusion. A semaphore is a protected variable ‘S’ which can be accessed by operations such as DOWN(S) or UP(S). Semaphore can be a counting semaphore or a general semaphore in which case it can take any value. Semaphore can also be a binary semaphore taking values 0 / 1. Semaphore can be implemented both in hardware and software. DOWN(S) and UP(S) form the mutual exclusion primitives for a process. A process in critical section is bounded by the instructions DOWN(S) and UP(S) as shown below:

```
Begin
        - - - - - - -
        DOWN(S)
        Critical section instructions
        UP(S)
        - - - - - - -
End
```

The mutual exclusion primitives DOWN(S) and UP(S) guarantee that only one process is in its critical section. All other processes wanting to execute their respective critical sections have to wait and are held in a semaphore queue. The operating system scans this semaphore queue where processes are waiting on a semaphore and releases a process from queue only when the current process has come out of its critical section (Figure 12.1).

It is evident from the figure that a process is in its critical section only if it has successfully executed S = S-1 without getting into the semaphore queue. If S is a binary semaphore then S takes values 0 / 1. If this is the case, a process can enter its critical section only if S = 1 at which time s becomes 0 because of S = S-1. The other way round, S = 0 implies a process is in its critical section. Any new process wanting to execute its critical section will find S = 0 and get added to the semaphore queue. S can again be 1 only in the UP routine where a process sets S = 1 when it has come out of its critical section. Just as in the previous case, DOWN(S) and UP(S) must be indivisible instructions. The Lock and Unlock operations shown in the figure take care of this condition by disabling / enabling interrupts.

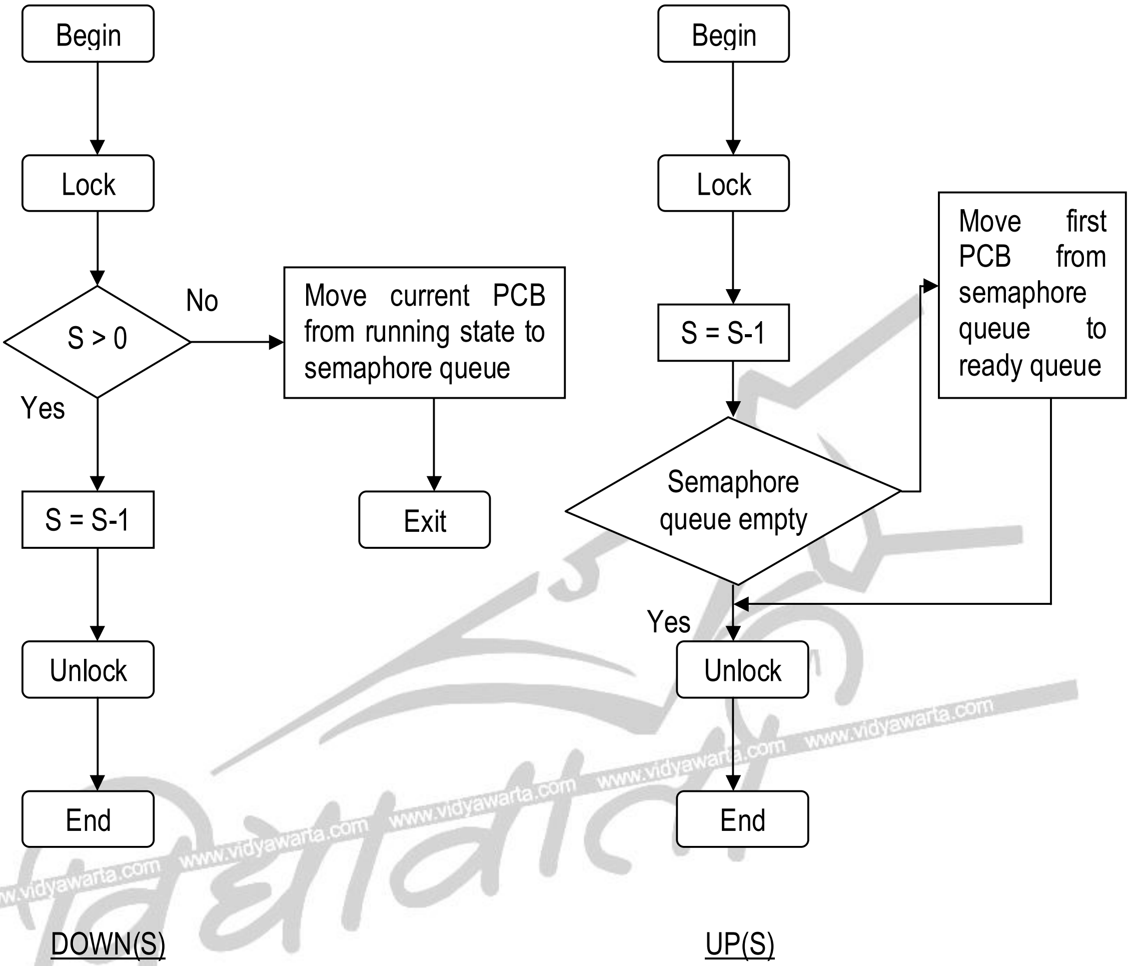

Begin
Lock
S > 0
No
Move current PCB from running state to semaphore queue
Yes
S = S-1
Exit
Unlock
End
DOWN(S)
Begin
Lock
S = S-1
Move first PCB from semaphore queue to ready queue
Semaphore queue empty
Yes
Unlock
End
UP(S)

Figure 12.1: Semaphore instructions

Algorithms for DOWN(S) and UP(S) could be as follows:

```
Begin                          Begin
  Disable interrupts             Disable interrupts
  If (S > 0) S = S-1             S = S+1
  Else Wait on S                 If semaphore queue not empty release a process
  Enable interrupts              Enable interrupts
End                            End
```

Semaphores have very wide use and applications where shared variables are used. Blocking / waking up of processes when they wait for completion of an I/O can be implemented using semaphores. Semaphores are used for synchronization.

12.3 SUMMARY

This chapter has brought out the need for inter-process communication and synchronization. These were illustrated using the producer-consumer problem. Various solutions to the problem were discussed. An important concept called the semaphore was introduced.

12.4 EXERCISE

1. Explain with examples the producer-consumer problem.
2. What is race condition? What are its causes?
3. What are semaphores? Explain how semaphores are used to implement mutual exclusion and critical section problem.

12.5 ACTIVITY

Study inter-process communication in the operating systems noted.

CHAPTER 13

MULTIPROCESSOR SYSTEMS

There are basically two ways of increasing the speed of computer hardware. They are by using:

- High speed components
- New architectural structures

In the first category, high-speed components rely on exotic technology and fabrication processes and such technologies and processes tend to be expensive and non-standardized. Multiprocessor systems fall into the second category and provide an alternative for improving performance of computer systems by coupling a number of low cost standard processors. Multiprocessing can be applied to provide:

- Increased throughput: by executing a number of different user processes on different processors in parallel
- Application speedup: by executing some portion of the application in parallel

Throughput can be improved by executing a number of unrelated user processes on different processors in parallel. System throughput is improved as a large number of tasks are completed in unit time.

Application speedup may be obtained by exploiting the hidden parallelism in the application by creating multiple threads / processes / tasks for execution on different processors.

Inter-processor communication and synchronization are an overhead in multiprocessor systems. Design goals aim to minimize inter-processor interaction and provide an efficient mechanism for carrying them out when necessary.

13.1 ADVANTAGES OF MULTIPROCESSORS

- Performance and computing power: Use of multiprocessor systems speeds up an application. Problems with high inter-processor interaction can be solved quickly.

- Fault tolerance: The inherent redundancy in multiprocessor systems can be used to increase availability and eliminate single point failures.
- Flexibility: A multiprocessor system can be made to dynamically reconfigure itself so as to optimize different objectives for different applications such as throughput, application speedup or fault tolerance.
- Modular growth: Use of a modular system design overcomes certain problems and can be accomplished say by adding exactly tailor made components such as processors, memories, I/O devices and the like.
- Functional specialization: specialized processors can be added to improve performance in particular applications.
- Cost / performance: cost performance ratio in case of multiprocessor systems is far below that of large computers of the same capacity and hence multiprocessor systems are cost effective. They may be structured similar to large computers for a wide range of applications.

13.2 MULTIPROCESSOR CLASSIFICATION

Flynn classified computer systems based on how the machine relates its instructions to the data being processed. Instructions may form either a single instruction stream or a multiple instruction stream. Similarly the data which the instructions use could be either single or multiple. Based on such instructions and data, Flynn classified computer systems as follows:

- SISD: single instruction stream, single data stream. Usually found in conventional serial computer systems.
- SIMD: single instruction stream, multiple data stream. These are vector processors / array processors where a single instruction operates on different data in different execution units at the same time.
- MISD: multiple instruction streams, single data stream. Multiple instructions operate on a single data stream. Not a practical viability.
- MIMD: multiple instruction streams, multiple data stream. This is the most general classification where multiple instructions operate on multiple data stream simultaneously. This is the class that contains multiprocessors of different types.

Based on the relationships between processes and memory, multiprocessor systems can be classified as:

- Tightly coupled: individual processors within the multiprocessor system share global shared memory.
- Loosely coupled: individual processors within the multiprocessor system access their own private / local memory.

This classification may not be very rigid and multiprocessor systems have both shared memory as well as local memory.

Inter-processor communication (IPC) and synchronization in tightly coupled multiprocessor systems is through shared memory. High bandwidth and low delay in interconnection paths are the main characteristics of tightly coupled multiprocessor systems.

In loosely coupled multiprocessor systems, message passing is the primary mechanism for IPC. Distributed systems fit into this class of loosely coupled systems. Lower bandwidth and high delay in the interconnection paths of the past have reduced drastically with the use of optical fiber links and high speed LANs.

Hybrid systems have both local and global memories. Some loosely coupled systems also allow access of global memory in addition to local memory.

Based on memory and access delays, multiprocessor systems are classified as:

- Uniform memory access (UMA): multiple processors can access all the available memory with the same speed.
- Non uniform memory access (NUMA): Different areas of memory have different access times. This is based on the nearness of the memory to a given processor and also on the complexity of the switching logic between the processor and the memory.
- No remote memory access (NORMA): systems have no shared memory.

13.3 MULTIPROCESSOR INTERCONNECTIONS

The nature of multiprocessor interconnections has an affect on the bandwidth for communication. Complexity, cost, IPC and scalability are some features considered in interconnections. Basic architectures for multiprocessor interconnections are as follows:

- Bus-oriented systems

- Crossbar-connected systems
- Hyper cubes
- Multistage switch-based systems

13.3.1 BUS-ORIENTED SYSTEMS

A shared bus connects processors and memory in the multiprocessor system as shown below (Figure 13.1).

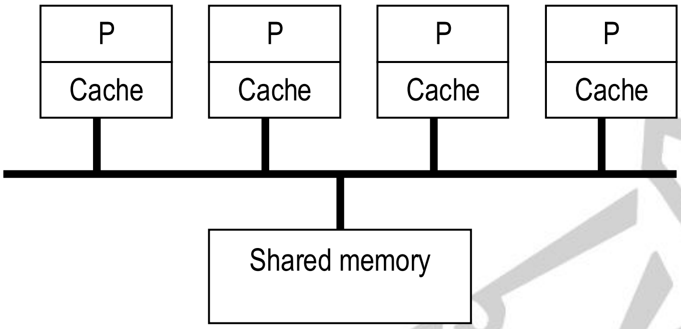


Figure 13.1: Shared-bus multiprocessor organization

Processors communicate with each other and the shared memory through the shared bus. Variations of this basic scheme are possible where processors may or may not have local memory, I/O devices may be attached to individual processors or the shared bus and the shared memory itself can have multiple banks of memory.

The bus and the memory being shared resources there is always a possibility of contention. Cache memory is often used to release contention. Cache associated with individual processors provides a better performance. A 90% cache hit ratio improves the speed of the multiprocessor systems nearly 10 times as compared to systems without cache.

Existence of multiple cache in individual processors creates problems. Cache coherence is a problem to be addressed. Multiple physical copies of the same data must be consistent in case of an update. Maintaining cache coherence increases bus traffic and reduces the achieved speedup by some amount. Use of a parallel bus increases bandwidth.

The tightly coupled, shared bus organization usually supports 10 processors. Because of its simple implementation many commercial designs of multiprocessor systems are based on shared-bus concept.

13.3.2 CROSSBAR-CONNECTED SYSTEMS

An interconnection of processors and memory in a multiprocessor system using crossbar approach is shown below (Figure 13.2):

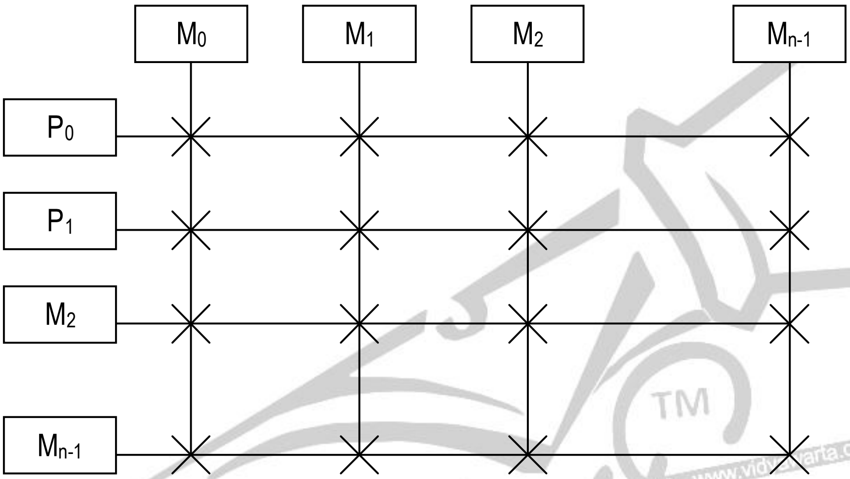


Figure 13.2: Crossbar interconnection

Simultaneous access of ‘n’ processors and ‘n’ memories is possible if each of the processors accesses a different memory. The crossbar switch is the only cause of delay between processor and memory. If no local memory is available in the processors then the system is a UMA multiprocessor system.

Contention occurs when more than one processor attempts to access the same memory at the same time. Careful distribution of data among the different memory locations can reduce or eliminate contention.

High degree of parallelism exists between unrelated tasks but contention is possible if inter-process and inter-processor communication and synchronization are based on shared memory, for example, semaphore.

Since ‘n’ processors and ‘n’ memory locations are fully connected, n^2 cross points exist. The quadratic growth of the system makes the system expensive and limits scalability.

13.3.3 HYPERCUBES

A 3-dimensional hypercube can be visualized as shown below (Figure 13.3):

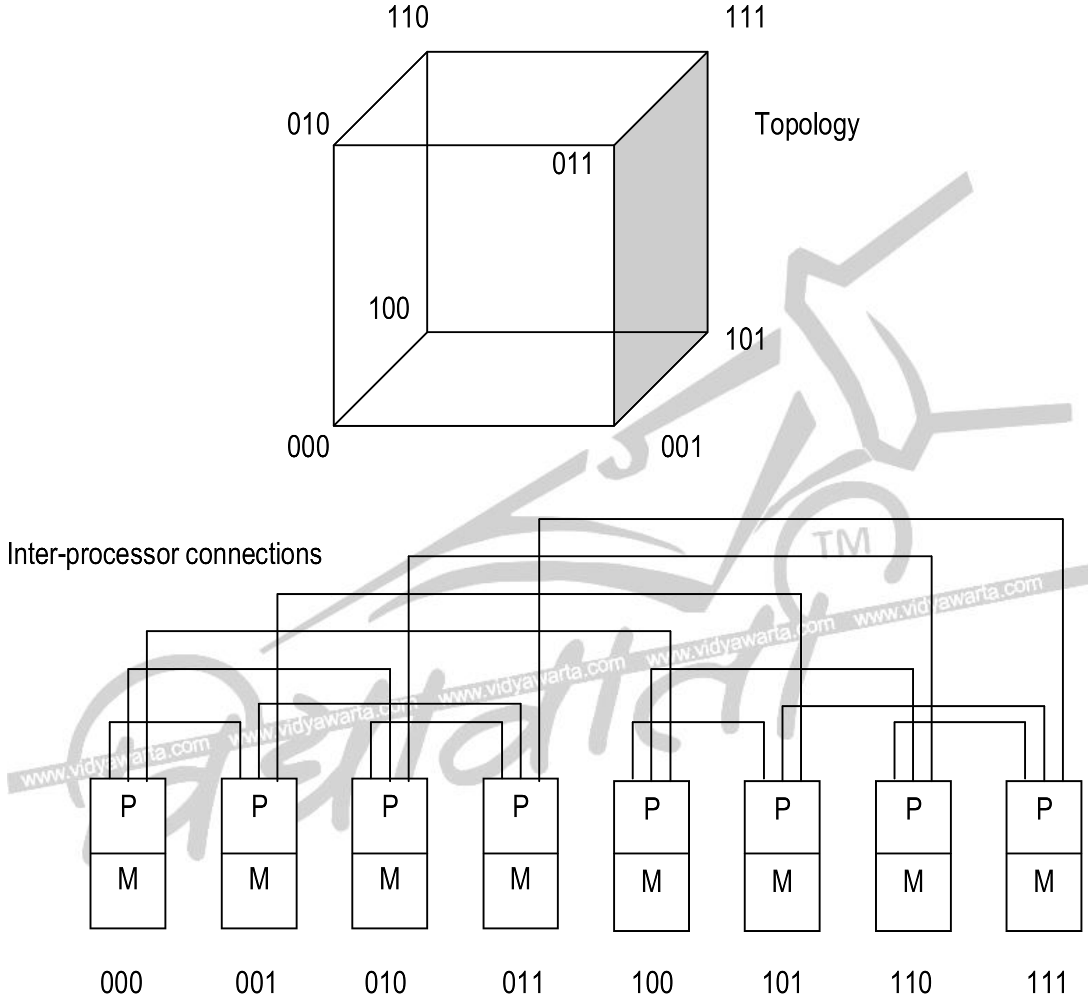


Figure 13.3: Hypercube interconnection

Cube topology has one processor at each node / vertex. Given a 3-dimentional cube (a higher dimensional cube cannot be visualized), 2^3 = 8 processors are interconnected. The result is a NORMA type multiprocessor and is a common hypercube implementation.

Each processor at a node has a direct link to $\log_2 N$ nodes where N is the total number of nodes in the hypercube. For example, in a 3-dimensional hypercube, N = 8 and each node is connected to $\log_2 8 = 3$ nodes. Hypercubes can be recursive structures with high dimension cubes containing low dimension cubes as proper subsets. For example, a 3-dimensional cube has two 2-dimensional cubes as subsets. Hypercubes have a good basis for scalability since complexity grows logarithmically where as it is quadratic in the previous case. They are best suited for problems that map on to a cube structure, those that rely on recursion or exhibit locality of reference in the form of frequent communication with adjacent nodes. Hypercubes form a promising basis for large-scale multiprocessors.

Message passing is used for inter-processor communication and synchronization. Increased bandwidth is sometimes provided through dedicated nodes that act as sources / repositories of data for clusters of nodes.

13.3.4 MULTISTAGE SWITCH-BASED SYSTEMS

Processors and memory in a multiprocessor system can be interconnected by use of a multistage switch. A generalized type of interconnection links N inputs and N outputs through $\log_2 N$ stages, each stage having N links to N / 2 interchange boxes. The structure of a multistage switch network is shown below (Figure 13.4):

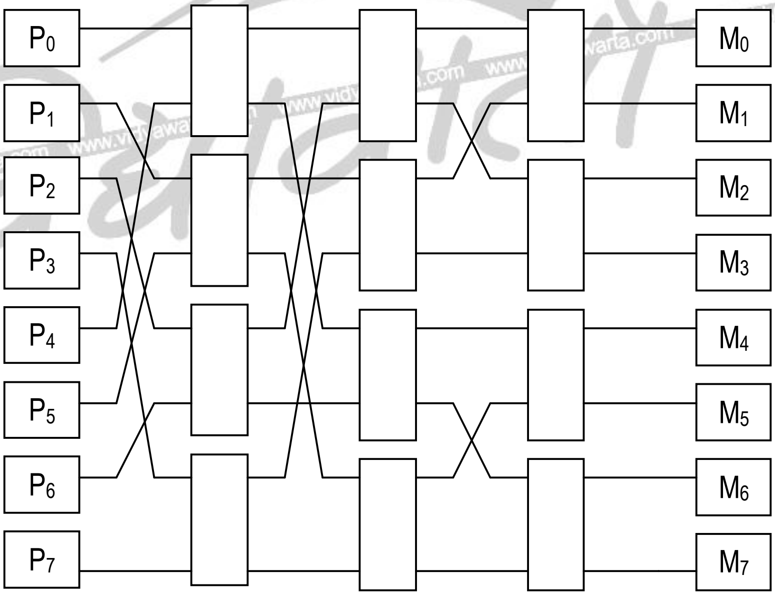


Figure 13.4: Multistage switching network

The network has $\log_2 N = \log_2 2^3 = 3$ stages of N / 2 = 8 / 2 = 4 switches each. Each switch is a 2x2 crossbar that can do anyone of the following:

- Copy input to output
- Swap input and output
- Copy input to both output

Routing is fixed and is based on the destination address and the source address. In general to go from source S to destination D the i^th^ stage switch box in the path from S to D should be set to swap if $S_i \neq D_i$ and set to straight if S_i = Di.

Illustration: If S = 000 and D = 000 then Si = Di for all bits. Therefore all switches are straight.

If S = 010 and D = 100 then $S_1 \neq D_1$, $S_2 \neq D_2$ and $S_3 = D_3$. Therefore switches in the first two stages should be set to swap and the third to straight.

Multistage switching network provides a form of circuit switching where traffic can flow freely using full bandwidth when blocks of memory are requested at a time. All inputs can be connected to all outputs provided each processor is accessing a different memory. Contention at the memory module or within the interconnection network may occur. Buffering can relieve contention to some extent.

13.4 TYPES OF MULTIPROCESSOR OPERATING SYSTEMS

Three basic types of multiprocessor operating systems are:

- Separate supervisors
- Master / slave
- Symmetric

13.4.1 SEPARATE SUPERVISORS

In separate supervisor systems, each node is a processor having a separate operating system with a memory and I/O resources. Addition of a few additional services and data structures will help to support aspects of multiprocessors.

A common example is the hypercube. Due to their regular repeating structure constructed of identical building blocks, they tend to replicate identical copies of a kernel in each node. Kernel provides services such as local process, memory management and message passing primitives. Parallelism is achieved by dividing an application into subtasks that execute on different nodes.

13.4.2 MASTER / SLAVE

In this approach, one processor – the master is dedicated to execute the operating system. The remaining processors are slaves and form a pool of computational processors. The master schedules and controls the slaves. This arrangement allows parallelism in an application by allocating to it many slaves.

Master / slave systems are easy to develop. A uniprocessor operating system can be adapted for master / slave multiprocessor operations with the addition of slave scheduling. Such systems have limited scalability. Major disadvantages are that computational power of a whole processor is dedicated for control activity only and if the master fails, the entire system is down.

13.4.3 SYMMETRIC

All processors are functionally identical. They form a pool of resources. Other resources such as memory and I/O devices are available to all processors. If they are available to only a few then the system becomes asymmetric.

The operating system is also symmetric. Any processor can execute it. In response to workload requirements and processor availability, different processors execute the operating system at different times. That processor which executes the operating system temporarily is the master (also called floating master).

An existing uniprocessor operating system such as UNIX can easily be ported to a shared memory UMA multiprocessor. Shared memory contains the resident operating system code and data structures. Any processor can execute the operating system. Parallel execution of applications is possible using a ready queue of processes in shared memory. The next ready process to the next available processor until either all processors are busy / queue is empty could be a possible allocation scheme. Concurrent access of shared data structures provides parallelism.

13.5 MULTIPROCESSOR OPERATING SYSTEM FUNCTIONS AND REQUIREMENTS

Multiprocessor operating systems manage available resources to facilitate program execution and interaction with users. Resources to be managed include:

- Processors
- Memory
- I/O devices

Processor scheduling is crucial for efficient use of multiple processors. The scheduler has to:

- Allocate processors among applications
- Ensure efficient use of processors allocated to an application.

A tradeoff exists between the two. The former affects throughput while the latter affects speedup. Depending on an application if speedup is given priority then a large portion of the processors is dedicated to the application. On the other hand if throughput is to be increased then several applications are scheduled each availing fewer processors. The two main facets of operating system support for multiprocessor are:

- Flexible and efficient inter-process and inter-processor synchronization mechanisms.
- Efficient creation and management of a large number of threads / processes.

Memory management in multiprocessor systems is dependent on the architecture and interconnection scheme. In loosely coupled systems, memory management is usually independent. In shared memory system, operating system should provide access to shared data structures and synchronization variables in a safe and efficient way. A hardware independent unified model of a shared memory for easy portability is expected of a multiprocessor operating system. A unified memory model consisting of messages and shared memory provides a flexible tool.

Device management is of little importance in a multiprocessor operating system. This may be due to the importance attached to speedup in the so called compute-intensive applications with minimal I/O. As more general purpose applications are run on multiprocessor systems, I/O requirements will also be a matter of concern along with throughput and speed.

13.6 OPERATING SYSTEM DESIGN AND IMPLEMENTATION ISSUES

Some major issues involved in processor and memory management in multiprocessor operating systems are described below:

13.6.1 PROCESSOR MANAGEMENT AND SCHEDULING

The main issues in processor management include:

- Support for multiprocessing
- Allocation of processing resources
- Scheduling

The operating system can support multiprocessors by providing a mechanism for creating and maintaining a number of processes / threads. Each process has allocated resources, state and accesses I/O devices. An application having several co-operating processes can be viewed as a virtual multiprocessor. In a multiprocessor environment true multiprocessing is possible by allocating a physical processor to each virtual processor where as in a uniprocessor system an illusion of multiprocessing is created by multiplexing the processor among virtual processes.

When threads are used, each application is implemented as a process. Its concurrent portions are coded as separate threads within the enclosing process. Threads of a single process share memory and resources acquired by the process. Threads not only facilitate multiprocessors bot also help the scheduling process. Related threads could be co-scheduled to reduce slowdown associated with the out-of-phase scheduling.

Processor allocation creates problems in massively parallel systems. One way of keeping track of processor resources is to organize them into a hierarchy. A processor at the highest level in the hierarchy notes state and activity of a group of processors. When an application is to be allocated them then idle machines at the bottom level get allocated. The hierarchy can grow upwards. This is called wave scheduling.

If wanted number of resources is not available then a request to a higher level in the hierarchy is made. Fault tolerance is possible as a process higher up in the hierarchy could reallocate activities of a failed processor to another. But implementation has practical difficulties such as a note of wrong availability.

Processors are allocated to applications. These have to be scheduled. One main objective is to co-schedule processes that interact so that they run at the same time. Processes that can be co-scheduled include several threads of a single process, sender and receiver of a message, processes at the end of a pipe, etc. Individual processes may be uniprogrammed or multiprogrammed. Since multiprogramming of individual processes creates problems for communication, co-scheduling process groups that communicate with each other are preferred.

In loosely coupled systems the scheduler should note affinity of some processes for certain processors that may be due to process state stored in local memory. Also placing interacting processes in the same processor or cluster with direct interprocessor links can reduce communication costs.

13.6.2 MEMORY MANAGEMENT

In tightly coupled multiprocessor systems the operating system must provide access to shared memory through primitives for allocation and reallocation of shared memory segments. If shared virtual memory is supported then translation look aside buffers (TLBs) contain mappings to shared segments. Similarly open files could also be shared. Use of shared memory improves performance of message passing.

13.7 SUMMARY

We have studied the advantages of using multiprocessor systems and their classification. We have also studied the different standard interconnection patterns of multiprocessor systems and types of multiprocessor environments like supervisors and master / slave. The chapter brings out the functions, requirements, and design and implementation issues of multiprocessor systems.

13.8 EXERCISE

1. List the advantages of multiprocessor systems.
2. How can multiprocessors be classified?
3. Explain the various multiprocessor interconnections.
4. Describe the basic types of multiprocessor operating systems.
5. Discuss the various issues for multiprocessor operating system design.

13.9 ACTIVITY

Find out if the operating systems noted are multiprocessor systems. If so study their types and interconnections.

CHAPTER 14

OPERATING SYSTEMS IN DISTRIBUTED PROCESSING

Earlier were the days of centralized computing. With the advent of micro and mini computers, distributed processing is becoming more and more popular. Merely having a large central computer with a number of remote terminals connected to it or with a number of computers at different locations with no connection among them do not constitute a distributed processing because neither processing nor data is distributed in any sense.

Operating systems have moved from single process systems to single processor, multi-user, and multitasking systems. Today the trend is towards multiprocessor, multitasking systems. Distributed processing and parallel processing are two technologies used to harness the power of a multiprocessor system. A proper mix of the technologies may yield better results.

Distributed processing and parallel processing have a common goal – high throughput using more processors. Then why not use a faster processor? It is difficult to achieve higher throughput out of hardware just by increasing speed of the processor. Also faster processors mean high costs. Higher throughput was envisaged by using the available microprocessors and interconnecting them. This is called distributed processing or loosely coupled system. In parallel processing or tightly coupled systems there is only one computer with multiple CPUs. The operating system here is responsible for load distribution, communication and co-ordination.

In distributed processing computers have to be connected to one another by links enabling electronic data transfer and data sharing among the various connected computers. In a distributed client-server computing environment, the server is huge and handles large databases / computational requests. Clients have smaller processing capability and are spread across different locations. The operating system in such a case has to be restructured to cater to this form of distributed processing. Two approaches to the problem are:

- Network operating system (NOS)
- Global operating system (GOS)

14.1 CHARACTERISTICS OF DISTRIBUTED PROCESSING

- Processing may be distributed by location
- Processing is divided among different processors depending on the type of processing done. For example, I/O handled by one processor, user interaction by another and so on.
- Processes can be executing on dissimilar processors.
- Operating system running on each processor may be different.

14.2 CHARACTERISTICS OF PARALLEL PROCESSING

- All processors are tightly coupled, use shared memory for communication and are present in one case.
- Any processor can execute any job. All processors are similar.
- All processors run a common operating system.

14.3 CENTRALIZED Vs DISTRIBUTED PROCESSING

Distributed processing implies a number of computers connected together to form a network. This connection enables distributed applications, data, control or a combination of all of them as against centralized applications, data and control in centralized systems.

14.3.1 DISTRIBUTED APPLICATIONS

Distributed applications mean different programs on different computers. This scheme allows the possibility of data capture at the place of its origin. Connections between these computers then allow this data to be shared. Programs / applications could be distributed in two ways. They are:

- Horizontal distribution
- Vertical / hierarchical distribution

In horizontal distribution all computers are at the same level implying that all the computers are capable of handling any functionality. Examples include office automation and reservation systems where many computers in a network are able to reserve, cancel or enquire. Application with all its programs is duplicated at almost all the computers.

In vertical or hierarchical distribution, functionality is distributed among various levels. These levels usually reflect some hierarchical levels in the organization. Computers at each of these levels perform specialized functions. For example, computers at branch level carry out branch level functions and those at zonal level are used for zonal level functions in a banking organization. Computers at each level can be networked together to avail shared data. There are possibilities of connections between levels to enable exchange of data and information. Here applications running on different computers may be the same but for an application program different capabilities may be present at different levels. For example, sales analysis at branch level and sales analysis at zonal level may generate summaries in different formats.

14.3.2 DISTRIBUTION OF DATA

In a distributed environment, data can also be distributed similar to distribution of programs. Data for applications could be maintained as:

- Centralized data
- Replicated data
- Partitioned data

In centralized data, data resides only at one central computer that can be accessed or shared by all other computers in the network. For example, master database. This central computer must run an operating system that implements functions of information management. It must keep track of users and their files and handle data sharing, protection, disk space allocation and other related issues. It must also run a front-end software for receiving requests / queries from other computers for data. These requests are then serviced one by one. It is because of this software that this central computer is called a server. Computers connected to the server can have their own local data but shared data has to necessarily reside in the server. In a distributed environment part of the master database could be centralized and the rest distributed among the connecting computers.

Sometimes a particular database is required very often at each computer in the network. If it is stored only in a central computer, as above, transmitting it from the server to local computers when required is time consuming and an unwanted exercise because the current state of the database may not have changed from a previous state. In such cases the specific database can be replicated or duplicated in the computer where it is needed often. But to maintain data coherence when part of the database has been updated, the modifications have to be reflected in all the places where it has been duplicated. For example, information about train timings and fares would need replication because this information is needed at all terminals which cater to train bookings / reservations / enquires the reason being frequency of changes to this particular database is very low.

Data could be distributed in a partitioned way. The entire database is sliced into many parts. Each part of the database then resides on a computer. Processing depends upon the kind of data distribution. Any other computer wanting to access information / data present not locally but at a remote site must send a query and receive the contents needed. If such is the case then each computer will run front-end software to receive queries and act a server for the data stored in it.

14.3.3 DISTRIBUTION OF CONTROL

Control in a distributed environment refers to deciding which program should be scheduled to run next, at which node / computer, what is its data requirement, is there a necessity for data at remote site to be transferred to the node and so on. Network management routines continuously monitor lines and nodes. They help in fault detection and suggest and implement necessary actions to be taken.

14.4 NETWORK OPERATING SYSTEM (NOS) ARCHITECTURE

The architecture of typical NOS is shown below (Figure 14.1). The basic features in any NOS are explained by tracing the steps involved in a remote read. It is assumed that shared data resides on the server and clients are those computers in the network (other than the server) that want to access the shared data.

- Software called redirection software exists in the client.

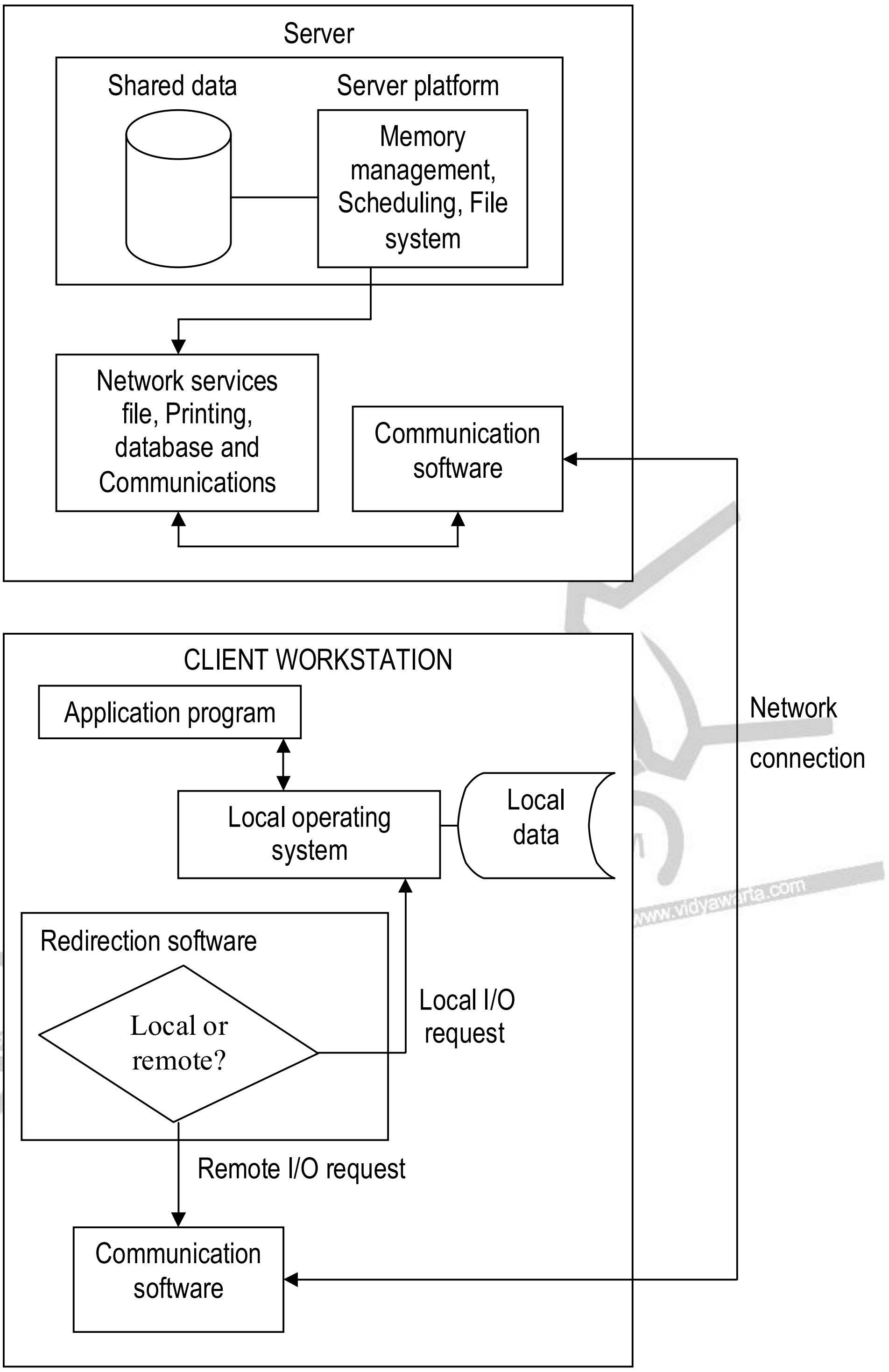


Figure 14.1: NOS architecture

- A system call generated by an application program not related to any I/O function is handled by the local operating system (LOS) running on the client.
- In a non-NOS environment all I/O by an application program is to the LOS only. However in the case of NOS environment this cannot be assumed. I/O may be to a local database or a remote database. In such a case a call is made to the redirection software of the NOS. The application program making this I/O call has knowledge about the location of the data (local / remote) and hence requests either the LOS for local data or the NOS for shared data. The NOS differentiates between a LOS I/O call and a NOS I/O call.
- If the request is for remote data then the call has to be processed as a remote procedure call (RPC) from the client to the server. In response to this request data traverses back to the client from the server. Communication management software handles the request for data and the actual data. This software resides both on the server as well as the client and ensures that a message is communicated between client and the server without any error and implements network functions such as packetizing, routing, error and flow control.
- For a remote request the redirection software on the client sends a request to the communication management software on the client.
- The communication management software on the client generates a RPC and sends it across the network.
- The communication management software on the server receives the request and in turn requests the network services software on the server itself for the clients request. This software is responsible for sharable resources such as files, disks, databases and printers. The software receives many such requests from different clients, generates a task for each one of them and schedules them for service. Thus NOS implements some kind of multitasking to service multiple tasks. Since network services software accesses shared resources, access control and protection are implemented.
- The network services software on the server communicates with the information management module of the operating system running on the server to get the requested data. Two approaches are possible. In one approach capabilities of information management are built into the NOS such as in NetWare. In the other

approach a separate operating system such as UNIX runs on the server and the network services software module of the NOS generates calls to the operating system in this case UNIX running on the server for required data.

- The network services software on the server sends the required data to the communication management software on the server to be sent to the client.
- The communication management software on the server also implements network functions such as packetizing, routing, sequence control, error and flow control to ensure error free data transfer to the client.
- The communication management software on the client now sends the received data to the application program so that it proceeds.

NOSs are available on LANs. LAN is an interconnection of a number of workstations to form a network. The network also has a large and more powerful computer attached to it. His computer called the server has a large disk and a printer attached to it. The server stores data that can be accessed by clients connected to the network. The clients in the form of workstations have small local memories that can be used for storing frequently accessed data once accessed from the server. Workstations can also be diskless in which case they have no local memory. The LOS is also downloaded into main memory during power up. All data in this case is requested and got from the server.

14.5 FUNCTIONS OF NOS

The main functions of NOS can be summarized as follows:

- Redirection
- Communication management
- File / printer services
- Network management

14.5.1 REDIRECTION

Redirection software normally resides on the client and also on the server. On the server also because if it is not a dedicated one then user of the server machine may want access to other computers. When does the redirection software actually work? An interrupt is executed by a system call generated say for an I/O. It is at the time of execution of the interrupt that redirection software intercepts to check if the I/O is local / remote. If it is local processing continues. If it is remote the redirection software has to generate a request to the server. But generating a request to the server has problems. The operating system running on the server ma y be different from that on the local machine generating the request. Also system architecture of the server may be different from the client. Therefore some conversion is necessary.

14.5.2 COMMUNICATION MANAGEMENT

The communication management software runs on both the client and the server. It is responsible for communication management. It is concerned with error-free transmission of messages (requests and data) to the destination. The ordinary operating system depends on separate software for this purpose. But in a NOS environment communication management software is built into the NOS as a part of it. Thus it resides on all clients and the server. It consists of a number of modules corresponding to the OSI layers.

14.5.3 FILE / PRINTER SERVICES

File / printer resources are controlled by these services. This software runs only on the server. Requests for shared resources are queued up, scheduled and then run as separate tasks thus making the NOS a multitasking operating system.

14.5.4 NETWORK MANAGEMENT SOFTWARE

Network management software is responsible for monitoring the network and its components such as computers, modems, repeaters, lines, adapters, multiplexers and many more.

Special software enables online testing of these equipment from time to time, checks their status and hence monitors the entire network. The network management software is responsible for all this. It maintains a list of hardware equipment along with its location and status. The list is updated when additional equipment is added or when equipment is down for repair. It generates reports based on which action can be taken ion terms of repair / replacements. It helps routing algorithms to route data on appropriate paths. The network management software resides on top of the existing operating system in ordinary operating systems. But in a NOS environment it is part of the NOS.

14.6 GLOBAL OPERATING SYSTEM (GOS)

The NOS is responsible for activities such as memory and process management on the server. The NOS converts a request into a task schedules and executes it. Memory and processing power in all other computers in the network is not tapped to the maximum by a NOS. This is exactly what the GOS attempts to do. It has a list of processes executing on different machines and the resources needed by each one of them. Relatively free processors can be scheduled with tasks for execution. Memory is managed at a global level. The various functions of the GOS are:

- User interface
- Information management
- Process / object management
- Memory management
- Communication management
- Network management

A typical GOS environment is depicted in the figure below (Figure 14.2). Part of the kernel of a GOS is duplicated at all sites. This kernel contains software to control hardware. Resources like information, memory, etc are managed by software that need not be replicated.

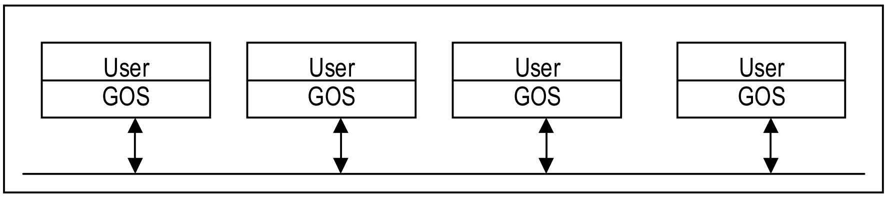


Figure 14.2: GOS environment

14.6.1 MIGRATION

The GOS has a pool of resources that it allocates to various processes / tasks at a global level. Migrations are necessary for optimal use of available resources. Migrations include:

- Data migration
- Computation migration
- Process migration

Data migration involves movement of data. A program running at a site X wants access to a file at site Y. Two options exist:

- Send the full file from Y to X
- Send only required portion of the file from Y to X

The first option is similar to the approach of a file server where as the second is similar to a database server. Software for sending the full file is simple. But the network will be loaded and in case the file is updated at site X, the entire file has to be again sent back to Y. If only required portions of a file are sent then network load is less but software to handle this is complex. Depending on requests for remote data, The GOS may migrate portion of data from one node to another or may replicate data to improve performance. This also brings with it the problems of data integrity.

The GOS may sometimes resort to computation migration. If nodes are distributed in a hierarchical fashion then data migration will need to transfer all files between levels. Alternatively if computation migration is followed then a process on one node can request for execution of another process at a remote site through a RPC. The results of this computation at remote site are then sent back for use. Here data file transfer is avoided.

Sometimes a process may be scheduled on a node that does not have the necessary requirements for the process because of which the process does not complete execution but is waiting in a blocked state for a long time. Since it was the only processor at the time of allocation it runs the process. Now that another processor with higher capacity is free, the GOS should be able to migrate the process to the new processor. There exists a tradeoff between the gain in performance of the migrated process and the overheads involved.

GOS may resort to process migration to enforce:

- Load balancing: to have a uniform utilization of available resources
- Special facilities: to use hardware / software facilities available at a particular node
- Reducing network load: process execution at a proper node reduces data migration and hence the load on the network.

14.6.2 RESOURCE ALLOCATION / DEALLOCATION

The GOS maintains a global list of all resources and allocates them to processes. This also includes migrated processes. . The resource allocation may lead to deadlocks. Deadlock handling in distributed systems is complex due to difficulties in maintaining an updated list of global resources. There is also a possibility of a false deadlock alarm. This may be caused because of incorrect information about resources that in turn may be due to delay in resource status reaching the global list. Deadlock detection can be centralized or a distributed function. Deadlocks can also occur in the communication system due to buffers getting full.

14.7 REMOTE PROCEDURE CALL (RPC)

A distributed environment consists of servers and clients. Server is a computer that offers services of shared resources. Client is a computer that requests for a shared resource present on the server through a request. A procedure is present on the server to locate and retrieve data present on a shared device attached to it. This procedure is part of the operating system running on the server. When a client requests for some data on the server this procedure on the server operating system is called remotely from the client. Hence it is called a remote procedure call (RPC).

14.7.1 MESSAGE PASSING SCHEMES

RPC can be considered as a special case of a generalized remote message-passing scheme as shown in below (Figure 14.3). The message handling module forms the interface that runs on all the nodes connected in the network. It interfaces with processes running on the nodes using primitives like SEND and RECEIVE. These modules handle communication across the network. Communication management functions are executed to ensure error-free communication.

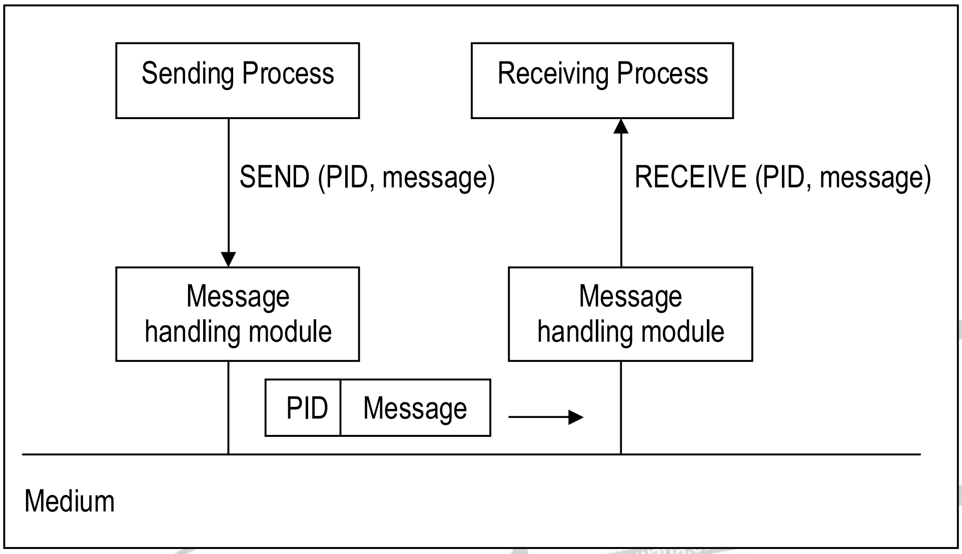


Figure 14.3: Message passing scheme

14.7.2 TYPES OF SERVICE

Message passing can be of two types. They are:

- Reliable service
- Unreliable service

A virtual circuit analogous to a telephone service is an example of a reliable service where as a datagram analogous to the postal service is an example for unreliable services. A reliable service ensures that the receiver receives the message sent by a sender correctly and properly in sequence. The overhead in this service includes an increased load on the network. An unreliable service only guarantees a high probability that a sent message is correctly received in proper order.

Message passing schemes could also be categorized as:

- Blocking
- Non-blocking

In the blocking scheme the process on the client that has requested for service from the server gets blocked until it receives back the data whereas in the non-blocking scheme the process requesting for service continues without waiting.

14.7.3 RPC

RPC can be viewed as an enhancement of a reliable blocking message-passing scheme to execute a remote procedure on another node. The message in this case is not a general one but specifies the procedure to be executed on the remote node along with required parameters.

14.7.4 CALLING PROCEDURE

A general format for an RPC could be as follows:

CALL P (A, B)

where P is the called procedure
A are the passed parameters
B are the returned parameters

Parameters can be passed either by value or by reference. When parameters are passed by value, the actual parameters are passed. Thus A and B will be actual parameters. If parameters are passed by reference then the addresses of the actual parameters are passed.

In RPC call by reference is very difficult because it is difficult to let processors on different machines to share a common address space. Hence call by reference does not make sense in RPC. It becomes tedious and time consuming. It also increases the load on the network. That us why only call by value method is used in RPC. A general schematic of RPC is shown below (Figure 14.4). The client process issues an RPC and gets blocked. The interface process completes the call and returns the results after which the client process becomes ready again.

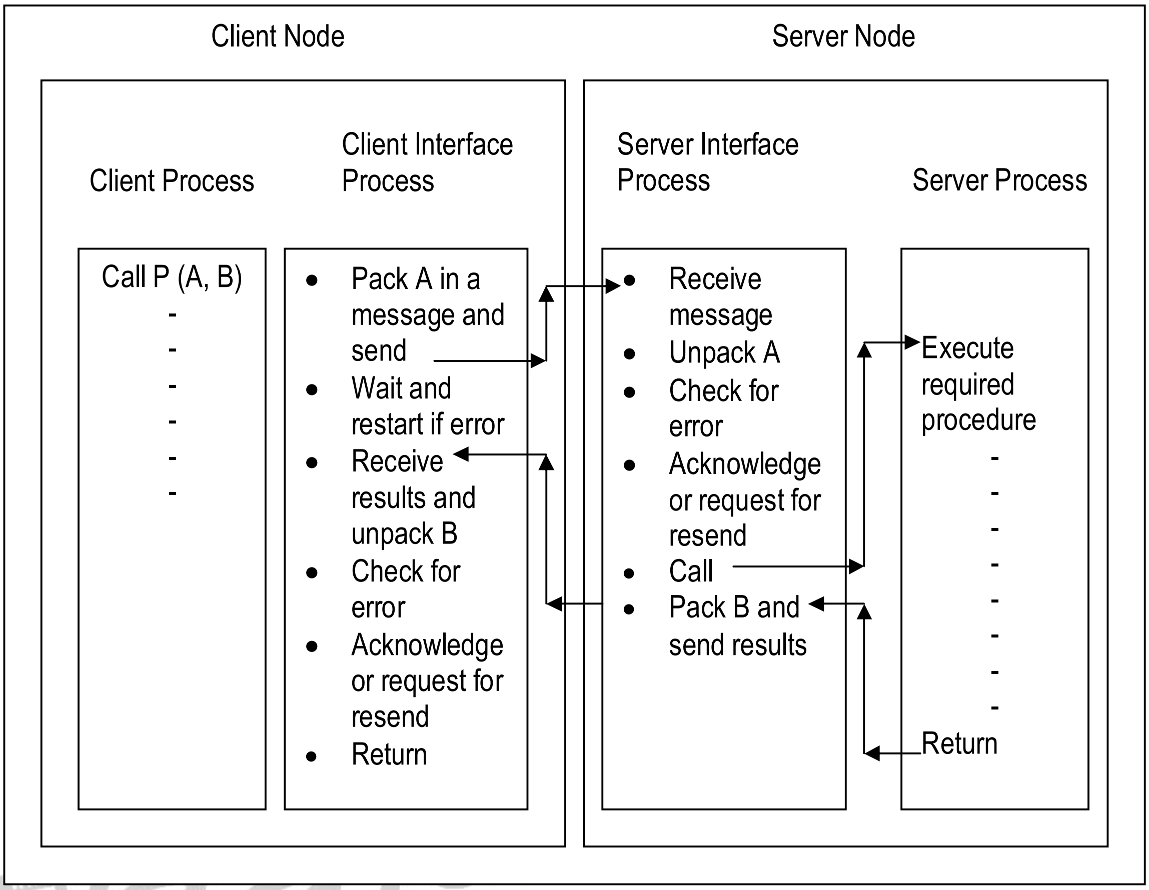


Figure 14.4: A general schematic of RPC

14.7.5 PARAMETER REPRESENTATION

If an RPC is issued between processes running on identical machines with same operating systems then parameters passed will be identical for a given language. But this is not the case if the machine architecture or the operating system or the programming language differs. One approach to this problem could be to have a common standard format. Then each interface module will have routines to convert from / to its own formats to / from the standard format. These routines will have to be present in all nodes as well as the server.

14.7.6 PORTS

If a server provides multiple services then normally a port number is associated with each service. For example, port number 1154 for listing current users, port number 2193 for opening a file and so on. RPC makes use of these port numbers. This simplifies communication. Hence a message sent as a RPC to a remote node contains among other information the port number and parameters for the service. The interface module on the remote node reads the port number and then executes the appropriate service.

14.8 DISTRIBUTED FILE MANAGEMENT

A network has many nodes. Each node has files in its local database. If NOS a user has to specify the exact location of a file to get it transferred to his / her node. But this is not required in GOS.

Sometimes in a NOS environment it is advantageous to keep multiple copies of the same file at different nodes. This reduces transfer time and also traffic on the network. The nearest node having the file can then satisfy a user request. To implement this, the node requesting the file, the remote node where the file is present and the frequency of requests need to be known. This is a dynamic situation since the pattern for file requests change with time. Hence in how many nodes to replicate a file is a dynamic issue. Maintaining data integrity is a problem as will have to be made at multiple locations.

Each node in the network runs its own local operating system and thus has its own file system. This local file system (LFS) is responsible for allocating space to a file, maintaining buffers, tables like FAT and so on. Services for file creation, deletion, read and write are provided by it. It maintains the directory structure and associated files. The functions of the LFS on a remote file are carried out by the distributed file system (DFS). It allows the users to see an entire structure of files and directories present in all the nodes put together as a hierarchy. An important implementation consideration in the design of DFS is the policy to be used to implement file operations especially write and update operations. DFS has to have software to interface with the operating system running on different nodes. This software should be present on all the nodes. If all nodes run the same operating system then complexity of DFS is greatly reduced.

UNIX has a feature called RFS that is a DFS for UNIX. SUN has its NFS that is again a DFS and is part of the SunOS operating system. NetWare-386 can support multiple machines and multiple networks / distributed file systems at the same time.

14.9 CACHE MANAGEMENT IN DISTRIBUTED PROCESSING

In a distributed environment load on the network determines the performance of the system. Cache is added to local nodes and also the server to reduce network traffic.

Cache is used very much like buffers in ordinary systems. Buffering is used in non-distributed systems and is implemented by the operating system. This helps save time for I/O because more data than what is absolutely necessary at the moment is read into the buffers. But this is advantageous only is concept of locality of reference prevails.

Cache management in distributed systems provides multiple buffers at different locations in the network. Desired data may be present at one of the following places:

- Client’s cache
- Client’s disk
- Server’s cache
- Server’s disk

Least time for I/O is when data is present in client’s cache and the maximum time is required when it is present in the server’s disk. The operating system in the distributed environment responsible for cache management searches for required data first in the client’s cache. If not present in the client’s disk, then in the server’s cache and at last in the server’s disk in that order. Performance depends upon the presence of data at any one of the four locations. After sometime a file in more than one block gets divided among nay of the four locations. The operating system has to move data depending on the past history of references. It keeps most recently used blocks in the client’s cache and the least recently used blocks on the server’s disk with blocks referenced in-between these two extremes kept on the client’s disk and the server’s cache. To implement this the operating system will have to maintain a list of blocks referenced along with the reference pattern and move data between locations using an algorithm like least recently used (LRU) approximation at each location. This is certainly very complex. Cache consistency problems arise when local copies of a file get updated. A solution to this problem is to use file locks.

14.10 PRINTER SERVER

A printer is an expensive resource and as such each node / client in a network need not have an attached printer. A printer server is used in a network environment to provide an illusion of a separate printer for each client. The printer server is a computer to which a printer is attached and is controlled by a program which is part of the NOS / GOS. It is very similar to the spooler program. A local disk attached to the printer server stores data to be printed.

A client wanting to print something, issues commands to print and data to be printed is redirected to the disk attached to the printer server. The spooler queues up all these requests and services them as and when the printer is free.

14.11 CLIENT-BASED (FILE SERVER) COMPUTING

In a client-based environment, sharable data is stored on the file server. A client can request for any data. Then the server sends down the entire file to the client. Files on the server are stored as raw files or as a stream of bits. Since the server has no knowledge about pointers / indexes forming relationships such as records in the file, the server will not be in a position to send only specific part (record) of a file. Processing (retrieval of a particular record) is done at the client. Network load is heavy. If the client does not have enough memory, then the file is broken down into chunks and sent. Since entire processing is done at the client, it is called client-based computing. It is useful to store a single copy of commonly used programs such as word processors.

To implement a file server, a multitasking operating system is necessary. Requests for files from different clients can arrive at the server. Each one of these requests becomes a task on the server, gets scheduled and executed.

Client-based computing is based on the concept of file servers. Limitations of this approach are:

- Multiple users cannot update the same file at the same time.
- There is a tremendous increase in network traffic.

14.12 CLIENT-SERVER (DATABASE SERVER) COMPUTING

In client-server computing, a program such as a database management system is split into two parts – a client part and a server part. That is why this environment is called client-server computing. The two pieces of the program reside on different machines. The data communication part runs on the client where a powerful workstation processes a user-friendly graphical user interface using languages like Visual Basic, Power Builder or Visual C++. The data base part runs on the server. As of today this can be one of the many products such as Oracle, Sybase or SQL server. For any application program, the user interface is provided for on the client machine and the database service on the server machine.

The server maintains indexes and uses them to search, locate and access desired records on the server in response to a client's request. It handles multiple concurrent requests and uses the concept of locking. That is why the server software is called database server. To maintain database integrity, it implements the concept of a transaction with facilities for roll back / roll forward. In contrast to this the file server does not maintain indexes, sends the entire file in response to a client's request and does not allow multiple users.

The database server is an intelligent server. It sends back only the records asked for by the client that uses them for further computing. This reduces traffic on the network. Since it handles multiple concurrent requests from clients it has to be a multitasking server with scheduling capability. There are two approaches for doing this:

- By scheduling multiple processes
- By scheduling multiple threads

If a multi-user operating system is running on the server, multiple requests by clients become processes on the server and get scheduled as any other processes. After a database record is fetched, the front-end on the server passes it on to the appropriate client. The communication capability of the NOS interfaces with the database server to bring about communication between the server and client and vice versa. Disadvantages with this approach is that the multi process architecture is inefficient in terms of use of system resources and also time consuming because of the context switch between processes.

Another approach is by having a single process on the server with multi-threading facility. The database server takes over some of the functionality of the operating system and schedules the requests from clients as tasks. Usually the database server runs on a powerful machine running an operating system such as UNIX.

Today the client-server environment is very popular among users. Powerful tools like Power Builder, Visual Basic and Visual C++ help users develop user-friendly graphical user interfaces for user interaction in application programs. These tools make database calls in standard format as expected by popular server products like Oracle and Sybase. These database products are sometimes referred to as database engines.

14.13 SUMMARY

We have studied what distributed processing is all about. We have seen how applications / data / control can be distributed. We have also seen the architecture of typical NOS and its functions. A GOS is necessary for optimal use of memory and processing power in all computers in a network. We have learnt what a RPC is and how it is executed. Distributed file management and cache management with associated problems were looked into. An earlier concept of client-based computing and the now very popularly used client-server computing are analyzed.

14.14 EXERCISE

1. Distinguish between distributed processing and parallel processing.
2. Explain how applications and data can be distributed.
3. Describe the procedure of performing a remote read in a NOS.
4. What is the need for migration? Explain the different types of migration.
5. Explain the execution of a RPC.
6. Discuss the importance of cache in distributed processing.
7. Compare client-based computing with client-server computing.

14.15 ACTIVITY

Find out if a distributed processing environment exists in the noted operating systems. If so study the execution of a RPC.

CHAPTER 15

SECURITY AND PROTECTION

Security is an important aspect of any operating system. Open Systems Interconnection (OSI) defines the elements of security in the following terms:

- Confidentiality: Information is not accessed in an unauthorized manner (controlled read)
- Integrity: Information is not modified or deleted in an unauthorized manner (controlled write)
- Availability: Information is available to authorized users when needed (controlled read / write / fault recovery)

Security is concerned with the ability of the operating system to enforce control over storage and movement of data in and between the objects that the operating system supports.

15.1 SECURITY THREATS

Personal computers were designed and intended for individual use. Hence security and protection features were minimal. No two users could simultaneously use the same machine. Locking the room physically which housed the computer and its accessories could easily protect data and stored information. But today hardware costs have reduced and people have access to a wide variety of computing equipment. With a trend towards networking, users have access to data and code present locally as well as at remote locations. The main advantages of networking like data sharing and remote data access have increased the requirements of security and protection. Security and protection are the two main features that motivated development of a network operating system (example Novell NetWare).

Major threats to security can be categorized as

- Tapping
- Disclosure

- Amendment
- Fabrication
- Denial

Unauthorized use of service (tapping) and unauthorized disclosure of information (disclosure) are passive threats whereas unauthorized alteration or deletion of information (amendment), unauthorized generation of information (fabrication) and denial of service to authorized users (denial) are active threats. In either tapping or disclosure information goes to a third party. In the former information is accessed by the third party without the knowledge of the other two parties and in the latter the source willingly / knowingly discloses it to the third party.

15.2 ATTACKS ON SECURITY

A security system can be attacked in many ways. Some of them are discussed below:

15.2.1 AUTHENTICATION

Authentication is verification of access to system resources. Penetration is by an intruder who may

- Guess / steal somebody's password and use it
- Use vendor supplied password usually used by system administrator for purposes of system maintenance
- Find a password by trial and error
- Use a terminal to access information that has been logged on by another user and just left like that.
- Use a dummy login program to fool a user

15.2.2 BROWSING

Browsing through system files could get an intruder information necessary to access files with access controls which are very permissive thus giving the intruder access to unprotected files / databases.

15.2.3 INVALID PARAMETERS

Passing of invalid parameters of failure to validate them properly can lead to serious security violations.

15.2.4 LINE TAPPING

A communication line is tapped and confidential data is accessed or even modified. Threat could be in the form of tapping, amendment or fabrication.

15.2.5 IMPROPER ACCESS CONTROLS

If the system administrator has not planned access controls properly, then some users may have too many privileges and others very few. This amounts to unauthorized disclosure of information or denial of service.

15.2.6 ROGUE SOFTWARE

A variety of software programs exist under this title. Computer virus is very well known among others. This is a deliberately written program or part of it intended to create mischief. Such programs vary in terms of complexity or damage they cause. Creators of this software have a deep knowledge of the operating system and the underlying hardware. Other rogue software includes Trojan horse, Chameleon, Software bomb, Worm, etc.

The above mentioned were some common ways in which a security system could be attacked. Other ways in which a security system can be attacked may be through Trap doors, Electronic data capture, Lost line, Waste recovery and Covert channels.

15.3 COMPUTER WORMS

A computer worm is a full program by itself. It spreads to other computers over a network and while doing so consumes network resources to a very large extent. It can potentially bring the entire network to a halt.

The invention of computer worms was for a good purpose. Research scientists at XEROX PARC research center wanted to carry out large computations. They designed small programs (worms) containing some identified piece of computations that could be carried out independently and which could spread to other computers. The worm would then execute on a machine if idle resources were available else it would hunt the network for machines with idle resources.

A computer worm does not harm any other program or data but spreads thereby consuming large resources like disk storage, transmission capacity, etc. thus denying them to legal users. A worm usually operates on a network. A node in a network maintains a list of all other nodes on the network and also a list of machine addresses on the network. A worm program accesses this list and using it copies itself to all those address and spreads. This large continuous transfer across the network eats up network resources like line capacity, disk space, network buffers, tables, etc.

Two major safeguards against worms are:

- Prevent its creation: through strong security and protection policies
- Prevent its spreading: by introducing checkpoints in the communication system and disallowing transfer of executable files over a network unless until they are permitted by some authorized person.

15.4 COMPUTER VIRUS

A computer virus is written with an intention of infecting other programs. It is a part of a program that piggybacks on to a valid program. It differs from the worm in the following ways:

- Worm is a complete program by itself and can execute independently whereas virus does not operate independently.
- Worm consumes only system resources but virus causes direct harm to the system by corrupting code as well as data.

15.4.1 TYPES OF VIRUSES

There are several types of computer viruses. New types get added every now and then. Some of the common varieties are:

- Boot sector infectors
- Memory resident infectors
- File specific infectors
- Command processor infectors
- General purpose infectors

15.4.2 INFECTION METHODS

Viruses infect other programs in the following ways:

- Append: virus code appends itself to a valid unaffected program
- Replace: virus code replaces the original executable program either completely or partially
- Insert: virus code gets inserted into the body of the executable code to carry out some undesirable actions
- Delete: Virus code deletes some part of the executable program
- Redirect: The normal flow of a program is changed to execute a virus code that could exist as an appended portion of an otherwise normal program.

15.4.3 MODE OF OPERATION

A virus works in a number of ways. The developer of a virus (a very intelligent person) writes an interesting program such as a game or a utility knowing well the operating system details on which it is supposed to execute. This program has some embedded virus code in it. The program is then distributed to users for use through enticing advertisements and at a low price. Having bought the program at a throwaway price, the user copies it into his / her machine not aware of the devil which will show up soon. The virus is now said to be in a nascent state. Curious about the output of the program bought, the user executes it. Because the virus is embedded in the host program being run, it also executes and spreads thus causing havoc.

15.4.4 VIRUS DETECTION

Virus detection programs check for the integrity of binary files by maintaining a checksum and recalculating it at regular intervals. A mismatch indicates a change in the executable file, which may be caused due to tampering. Some programs are also available that are resident in memory and continuously monitor memory and I/O operations.

15.4.5 VIRUS REMOVAL

A generalized virus removal program is very difficult. Anti-virus codes for removal of viruses are available. Bit patterns in some virus code are predictable. The anti-virus programs scan the disk files for such patterns of the known virus and remove them. But with a number of viruses cropping up every now and then, development and availability of anti-virus for a particular type is delayed and harm done.

15.4.6 VIRUS PREVENTION

'Prevention is better than cure'. As the saying goes, there is no good cure available after infection. One of the safest ways to prevent virus attacks is to use legal copies of software. Also system needs to be protected against use of unauthorized / unchecked floppy disks. Frequent backups and running of monitoring programs help detection and subsequent prevention.

15.5 SECURITY DESIGN PRINCIPLES

General design principles for protection put forward by Saltzer and Schroeder can be outlined as under:

- Public design: a security system should not be a secret, an assumption that the penetrator will know about it is a better assumption.

- Least privileges: every process must be given the least possible privileges necessary for execution. This assures that domains to be protected are normally small. But an associated overhead is frequent switching between domains when privileges are updated.
- Explicit demand: access rights to processes should not be granted as default. Access rights should be explicitly demanded. But this may result in denial of access on some ground to a legal user.
- Continuous verification: access rights should be verified frequently. Checking only at the beginning may not be sufficient because the intruder may change access rights after initial check.
- Simple design: a simple uniform security system built in layers, as an integral part of the system is preferred.
- User acceptance: Users should not have to spend a lot of effort to learn how to protect their files.
- Multiple conditions: where ever possible the system must be designed to depend on more than one condition, for example, two passwords / two keys.

15.6 AUTHENTICATION

Authentication is a process of verifying whether a person is a legal user or not. This can be by either verification of users logging into a centralized system or authentication of computers that are to work in a network or a distributed environment.

Password is the most commonly used scheme. It is easy to implement. User name is associated with a password. This is stored in encrypted form by the system. When the user logs onto the system, The user has to enter his user name and password against a prompt. The entered password is then encrypted and matched with the one that is stored in the file system. A tally will allow the user to login. No external hardware is needed. But limited protection is provided.

The password is generally not echoed on the screen while being keyed in. Also it is stored in encrypted form. It cannot be deciphered easily because knowing the algorithm for deciphering will not suffice as the key is ought to be known for deciphering it.

Choosing a password can be done by the system or by the system administrator or by the users themselves. A system-selected password is not a good choice as it is difficult to remember. If the system administrator gives a user a password then more than one person knows about it. User chosen passwords is practical and popular. Users should choose passwords are not easy to guess. Choosing user names, family names, names of cities, etc are easy to guess.

Length of a password plays an important role in the effectiveness of the password. If it is short it is easy to remember and use but easy to decipher too. Longer the password it is difficult to break and also to remember and key in. A trade off results in a password of length 6-8 characters.

Salting is a technique to make it difficult to break a password. Salting technique appends a random number ‘n’ to the password before encryption is done. Just knowing the password is not enough. The system itself calculates stores and compares these random numbers each time a password is used.

Multiple passwords at different levels could provide additional security. Change of password at regular intervals is a good practice. Many operating systems allow a user to try only a few guesses for a login after which the user is logged off the system.

15.7 PROTECTION MECHANISM

System resources need to be protected. Resources include both hardware and software. Different mechanisms for protection are as follows:

Files need to be protected from unauthorized users. The problem of protecting files is more acute in multi-user systems. Some files may have only read access for some users, read / write access for some others, and so on. Also a directory of files may not be accessible to a group of users, for example, student users do not access to any other files except their own. Like files devices, databases, processes also need protection. All such items are grouped together as objects. Thus objects are to be protected from subjects who need access to these objects.

The operating system allows different access rights for different objects. For example, UNIX has read, write and execute (rwx) rights for owners, groups and others. Possible access rights are listed below:

- No access
- Execute only

- Read only
- Append only
- Update
- Modify protection rights
- Delete

A hierarchy of access rights is identified. For example, If update right is granted then it is implied that all rights above update in the hierarchy are granted. This scheme is simple but creation of a hierarchy of access rights is not easy. It is easy for a process to inherit access rights from the user who has created it. The system then need maintain a matrix of access rights for different files for different users.

The operating system defines the concept of a domain. A domain consists of objects and access rights of these objects. A subject then gets associated with the domains and access to objects in the domains. A domain is a set of access rights for associated objects and a system consists of many such domains. A user process always executes in any one of the domains. Domain switching is also possible. Domains in the form of a matrix is shown below (Figure 15.1):

		OBJECTS							
		File 0	File 1	File 2	File 3	File 4	File 5	Printer 0	Printer 1
DOMAINS	0	R W	R					W	
	1			R	R W X				W
	2					W	R W X		W
	3			W		R			

Figure 15.1: Domains in matrix form

A variation of the above scheme is to organize domains in a hierarchy. Here also a domain is a set of access rights for associated objects. But the protection space is divided into ‘n’ domains from 0 to (n-1) in such a way that domain 0 has maximum access rights and domain (n-1) has the least. Domain switching is also possible. A domain switch to an outer domain is easy because it is less privileged where as a domain switch to an inner domain requires permissions.

Domain is an abstract concept. In reality domain is a user with a specific id having different access rights for different objects such as files, directories and devices. Processes created by the user inherit all access rights for that user. An access control matrix showing users and objects (files) needs to be stored by the operating system in order to decide granting of access rights to users for files.

Since the matrix has many holes, storing the entire matrix is wasteful of space. Access control list is one way of storing the matrix. Only information in the columns is stored and that too only where information is present that is each file has information about users and their access rights. The best place to maintain this information is the directory entry for that file.

Capability list is another way of storing the access control matrix. Here information is stored row wise. The operating system maintains a list of files / devices (objects) that a user can access along with access rights.

A combination of both access control list and capability list is also possible.

15.8 ENCRYPTION

Encryption is an important tool in protection, security and authentication. The process involves two steps (Figure 15.2):

- Encryption: the original message is changed to some other form
- Decryption: the encrypted message is restored back to the original

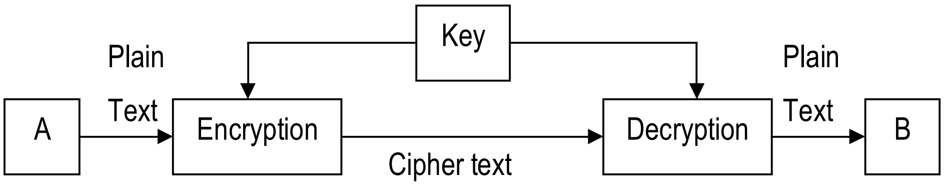


Figure 15.2: Conventional Encryption

Data before encryption is called plain text and after encryption is called cipher text. Usually the above operations are performed by hardware.

Encryption could be by one of the following two basic methods:

- Transposition ciphers
- Substitution ciphers

In transposition ciphers the contents of the data are not changed but the order is changed. For example, a message could be sent in reverse order like:

I am fine → enif ma I

Railfence cipher is a method that belongs to this class. The method is slow because the entire message is to be stored and then encrypted. It also requires more storage space when messages are long.

Substitution ciphers work by sending a set of characters different from the original like:

I am fine → r zn ormv

Ceasar cipher is a popular method of this type. This method is fast and requires less memory because characters can be changed as they are read and no storage is required.

Variations of this scheme are used for bit streams. Encryption in this case involves adding a key to every bit stream and decryption is removing the key from the cipher text.

Thus every algorithm has a key. It must ensure restoration. Normally a single piece of hardware is responsible for both encryption and decryption.

In the conventional encryption scheme two parties A and B agree upon a key. Someone say A or B or a third party has to decide upon this common key get concurrence from concerned parties and initiate communication. This is called key distribution. Each pair of nodes needs a unique key. If there are ‘n’ nodes then there will be nx(n-1)/2 keys. If ‘n’ is large then the number of keys will also be large. Deciding, conveying and storing these keys is a mammoth job. Tapping can take place. This is the key distribution problem.

An alternate is the public key encryption. Keys used for encryption and decryption are not the same. Key K1 is used for encryption and another key K2 is used for decryption. A message encrypted using K1 can be decrypted only using K2 and not K1. One of the keys is publicly known. Hence the name public key encryption. Decryption is done using a private key and hence information cannot leak out. Interchange of keys K1 and K2 is possible that is, K2 to encrypt and K1 to decrypt.

Each user has two keys, one public and one private (Figure 15.3). The private key is a secret but the user publishes the public key to a central key database. The database maintains public keys of different users.

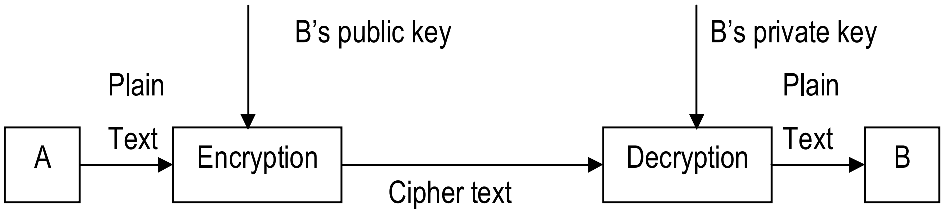


Figure 15.3: Public key Encryption

Encryption and decryption are as follows:

- A wants to send a message to B.
- A searches the database of public keys for the public key of B.
- A encrypts the data using B's public key.
- The cipher text is sent to B.
- B receives this cipher text.
- B decrypts the received cipher text using its private key and reads the message.

The problem here is that of authentication. B does not know who has sent the message to it because everybody knows B's public key. In the conventional encryption method a single key is used between two parties and hence the receiver knows the sender. But it suffers from the problem of key distribution. In public key encryption method, for 'n' nodes in the network only 2xn keys (1 public and 1 private for each of the nodes) are required. There need be no agreement. Private key is chosen and a public key is made known. Key distribution is really not necessary. Key leakage and tapping are minimal. Protection is ensured but authentication is not provided.

15.9 SECURITY IN DISTRIBUTED ENVIRONMENT

Security problems in a distributed environment are complex. Messages through a network can be tapped at multiple locations. For an active attack the intruder gets control over a link so that data modification / deletion is possible. For a passive attack the intruder just listens to a link and uses the passing information.

Encryption in a distributed environment can be of two forms:

- End-to-end encryption
- Link encryption

If end-to-end encryption is used the encryption / decryption devices are needed only at the ends. Data from source to destination moves on the network in encrypted form. In packet switched networks data is sent in the form of packets. Each packet has control information (source address, destination address, checksum, routing information, etc.) and data. Since routing address is needed for the packet to hop from the source till it reaches the destination, the control information cannot be encrypted as there is no facility to decrypt it anywhere in between. Only the data part in a packet can be encrypted. The system thus becomes vulnerable for tapping.

Link encryption needs more encryption / decryption devices, usually two for each link. This allows total encryption of a packet and prevents tapping. The method is expensive and slow.

A combination of both is possible.

Message authentication allows users to verify that data received is authentic. Usually the following attributes of a user need to be authenticated:

- Actual message
- Time at which sent
- Sequence in which sent
- Source from which it has arrived

Common methods for message authentication are:

- Authentication code
- Encryption
- Digital signatures

In authentication code, a secret key is used to generate a check sum, which is sent along with the data. The receiver performs the same operation using the same secret key on the received data and regenerates the check sum. If both of them are same then the receiver knows the sender since the secret key is known to only both of them.

Encryption is as discussed above where conventional encryption provides authentication but suffers from key distribution problems and public key encryption provides good protection but no authentication.

Digital signature is like a human signature on paper. If a signed letter is sent by A to B, A cannot deny having sent it to B (B has the signed copy) and B cannot refuse having got it (A has an acknowledgement for B having received it). This is what happens in a manual system and should happen in electronic messages as well.

As discussed earlier, public key encryption provides protection but not authentication. If we want to authentication without protection, reversal of the keys applied is a solution as shown below (Figure 15.4).

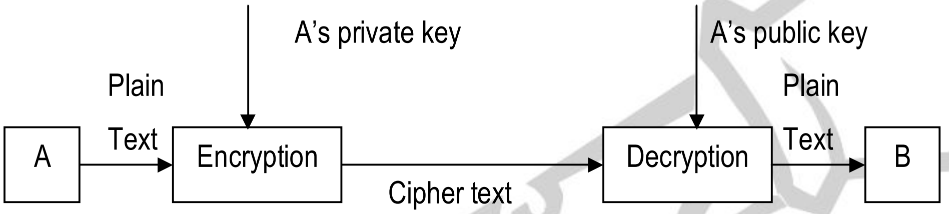


Figure 15.4: Public key Encryption for authentication without protection

This is based on the concept that public key encryption algorithm works by using either of the keys to encrypt and the other for decryption. A encrypts the message to be sent to B using its private key. At the other end B decrypts the received message using A’s public key which is known to everybody. Thus B knows that A has sent the message. Protection is not provided as anyone can decrypt the message sent by A.

If both authentication and protection are needed then a specific sequence of public and private keys is used as show below (Figure 15.5).

The two keys are used as shown. At points 2 and 4 the cipher text is the same. Similarly at points 1 and 5 the text is the same. Authentication is possible because between 4 and 5 decryption is done by A’s public key and is possible only because A has encrypted it with its private key. Protection is also guaranteed because from point 3 onwards only B can decrypt with its private key. This is how digital signatures work.

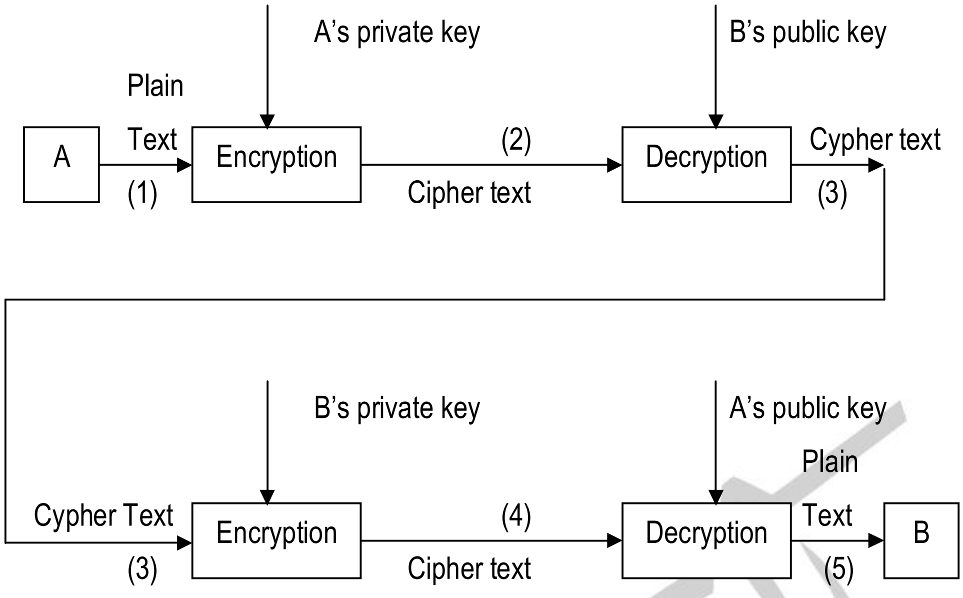


Figure 15.5: Public key Encryption for both authentication and protection

15.10 SUMMARY

This chapter looks into an important part of any operating system – security and protection. These were trivial matters in earlier systems since computers were centralized systems accessed only by knowledgeable users. With advances and use of networking, security and protection requirements have increased. Different ways in which system could be attacked are understood. Authentication using passwords is studied. Protection by looking at objects and users in domains accessing objects with different access rights is analyzed. Encryption as an important tool in protection, security and authentication is studied.

15.11 EXERCISE

1. Discuss the need for security and protection in computer systems.
2. Write a note on computer virus.
3. Describe authentication by using passwords.

4. How is protection implemented using the concept of domains?
5. What is encryption? What are the different ways in which a message can be encrypted?
6. Write a note on digital signatures.

15.12 ACTIVITY

Study the security and protection mechanisms implemented in some computer systems.

REFERENCES

1. Silberschatz and Galvin, Operating system concepts, Addison-Wesley publication, 5th edition.
2. Achyut S Godbole, Operating systems, Tata McGraw-Hill publication.
3. Milan Milankovic, Operating systems – concepts and design, Tata McGraw-Hill publication, 2nd edition.