BT 0070
Operating Systems

Contents

Prof.V.B.Nanda Gopal
Director & Dean
Directorate of Distance Education
Sikkim Manipal University of Health, Medical & Technological Sciences (SMU DDE)

Edition: Spring 2009

BLOCK INTRODUCTION

Without its software, a computer is basically a lump of metal. With its software, a computer can store, process and retrieve information, find spelling errors in manuscripts, play games, and engage in many valuable activities to earn its keep computer software can be roughly divided into two kinds: the system programs, which manage the operation of the computer itself, and the application programs, which solves problem for their users. The most fundamental of all the system programs is the operating systems. Which controls all the computer resources and provides the base upon which the application programs can be written.

Operating Systems are an essential part of any computer system. Similarly, a course on Operating System is an essential part of any computer-science education. In this book, we do not concentrate on any particular operating system or Hardware. Instead, we discussed fundamental concepts that are applicable to a variety of systems. We present a large number of examples that pertain to UNIX and to other popular operating systems. In particular, we use Windows, Sun Microsystem's Solaris 2 operating system, Windows Nt, Linux and UNIX operating system.

Unit 1: Operating System- An Introduction : This unit speaks about introduction, definition of OS, functions of OS, Evolution of OS and also Structures of Operating System.

Unit 2: Process management: This unit speaks about the process, process state , process control Block, process Scheduling, operations on processes and co-operating processes. In this unit we also study about Threads.

Unit 3: CPU Scheduling Algorithms: This unit speaks about Scheduling concepts, different scheduling algorithms and evaluation of Scheduling algorithms.

Unit 4 : Process Synchronization : This unit speaks about Interprocess communication, Critical Section problem, Semaphores, Monitors and Hardware Assistance.

Unit 5 : Introduction to Deadlocks : This unit speaks about System model, Deadlock characterization, Deadlock prevention, Deadlock avoidance & Deadlock detection.

Unit 6 : Memory Management : This unit speaks about logical v/s physical address space, swapping, contiguous allocation, paging and segmentation.

Unit 7 : Virtual Memory : This unit speaks about need for virtual Memory, Demand Paging, different page Replacement Algorithm and Thrashing.

Unit 8 : File Systems Interface and Implementation : This unit speaks about Concept of file, different file access methods, directory structures, allocation methods, free space management and directory implementation.

Unit 9 : Operating Systems in Distributed Processing : This unit speaks about characteristics of Distributed and Parallel Processing, centralized v/s Distributed processing, Network Operating System architecture, functions of NOS, Global Operating system, RPC and Distributed File Management.

Unit 10 : Security And Protection : This unit speaks about attacks on security, computer Worms, computer Virus, Security Design Principles, Authentication, protection mechanism, encryption and security in distributed Environment

Unit 11: Multiprocessor Systems : This unit speaks about Advantages of Multiprocessors, Multiprocessor classification, Multiprocessor Interconnections, Types of Multi-processor operating Systems, Multiprocessor Operating System functions, requirements and Implementation issues.

Unit 1 Operating System - An Introduction

Structure

1.1 Introduction

Computer software can be divided into two main categories: application software and system software. An application software consists of the programs for performing tasks particular to the machine's utilization. Examples of application software include spreadsheets, database systems, desktop publishing systems, program development software and games. Application software is generally what we think of when someone speaks of computer programs. This software is designed to solve a particular problem for users.

On the other hand, system software is more transparent and less noticed by the typical computer user. This software provides a general programming environment in which programmers can create specific applications to suit their needs. This environment provides new functions that are not available at the hardware level and performs tasks related to executing the application program. System software acts as an interface between the hardware of the computer and the application software that users need to run on the computer.

The most important type of system software is the operating system. An operating system has three main responsibilities:

1. Perform basic tasks, such as recognizing input from the keyboard, sending output to the display screen, keeping track of files and directories on the disk and controlling peripheral devices such as disk drives and printers.
2. Ensure that different programs and users running at the same time do not interfere with each other.
3. Provide a software platform on top of which other programs (i.e., application software) can run.

The first two responsibilities address the need for managing the computer hardware and the application programs that use the hardware. The third responsibility focuses on providing an interface between application software and hardware so that application software can be efficiently developed. Since the operating system is already responsible for managing the hardware, it should provide a programming interface for application developers.

Objectives:

At the end of this unit, you will be able to understand:

- Definition and functions of Operating System
- Evolution of operating systems and
- Operating system structures.

1.2 Definition and functions of Operating System

What is Operating System?

Operating System is a System Software(Set of system programs) which provides an environment to help the user to execute the programs. The Operating System is a resource manager which allocates and manages various resources like processor(s), main memory, input/output devices and information on secondary storage devices.

Functions of Operating System

Operating systems perform the following important functions:

i) **Processor management** : It means assigning processor to different tasks which has to be performed by the computer system.

ii) **Memory management** : It means allocation of main memory and secondary storage areas to the system programmes, as well as user programmes and data.

iii) **Input and output management** : It means co-ordination and assignment of the different output and input devices while one or more programmes are being executed.

iv) **File system management** : Operating system is also responsible for maintenance of a file system, in which the users are allowed to create, delete and move files.

v) **Establishment and enforcement of a priority system** : It means the operating system determines and maintains the order in which jobs are to be executed in the computer system.

vi) Assignment of system resources, both software and hardware to the various users of the system.

1.3 Evolution of Operating systems (Types of OS).

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. A brief description of the evolution of operating systems has been described below .

1.3.1 Simple Batch Operating Systems

In the earliest days digital computers 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 1.1).

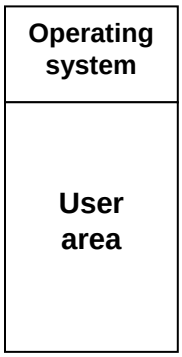


Figure 1.1: Memory layout for simple batch system

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 batch operating systems came into existence .

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. To overcome this problem the concept of SPOOLING came into picture. 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 1.2).

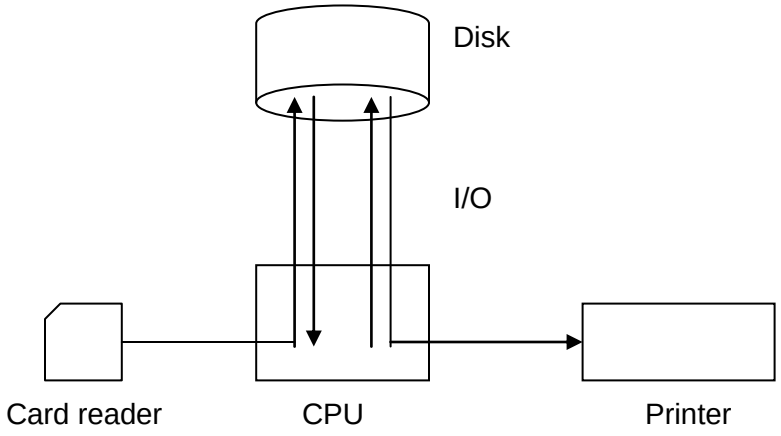


Figure1.2: Spooling

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

1.3.2 Multi-programmed Batched Operating 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 multi-programming. A single user cannot keep both CPU and I/O devices busy. Multiprogramming increases CPU utilization by organizing jobs in such a manner that CPU always has a job to execute.

The idea of multi-programming can be described as follows. A job pool on the disk consists of a number of jobs that are ready to be executed (Figure 1.3). 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.

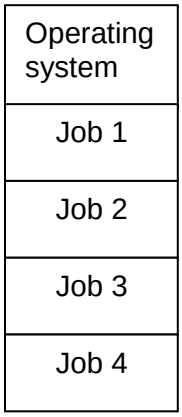


Figure 1.3: Memory layout for multi-programmed system

1.3.3 Time-Sharing Operating Systems

Multi-programmed 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 multi-programming. It provides interactive use of the system.

CPU scheduling and multi-programming 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.

Time-sharing operating systems are more complex than multi-programmed operating systems. This is because in multi-programming several jobs must be kept simultaneously in memory, which requires some form of memory

management and protection. Jobs may have to be swapped in and out of main memory to the secondary memory. To achieve this goal a technique called virtual memory is used.

Virtual memory allows execution of a job that may not be completely in a memory. The main logic behind virtual memory is that programs to be executed can be larger physical memory, it's execution takes place.

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

1.3.4 Personal Computer Operating Systems

The main mode of use of a PC(as its name implies) is by a single user. Thus Os for PCs were designed as a single user single task operating system, that is , it is assumed that only one user uses the machine and runs only one program at a time.

The operating system of PCs consists of two parts. One part is called the BIOS (Basic Input Output system) which is stored in a ROM (Read Only Memory). The other part called the DOS (Disk Operating System) is stored in a floppy disk or a hard disk.

When power is turned on BIOS takes control. It does what is known as power-on self test. The test sees whether memory is OK and all other relevant units function. Having done this, it reads from the disk a small portion of OS known as the boot and loads it into the main memory. This boot program then "pulls" the rest of the OS from the disk and stores it in the main memory. This is known as "booting the system".

BIOS provides basic low level services where as DOS provides many user-level services. The major services provided by DOS are:

- File management which allows user to create , edit, read, write and delete files.

- Directory management which allows creation, change, search and deletion of directories.
- Memory management which allows allocation and deallocation of memory.
- Command interpreter which interprets commands issued by the user and executes DOs functions, utility programs or application programs.
- Executive functions which provide programs to load and execute user programs, retrieve error codes, correct and rerun programs.
- Utility programs that do housekeeping chores such as COPY, ERASE, DIR etc.

1.3.5 Multi-processor Operating Systems (Parallel System)

Most systems to date are single-processor systems, which will be having only one CPU. However there is a need for multi-processor systems. Such systems will be having more than one processor and all these processor will share the computer bus, the clock and sometimes memory and peripheral devices. These systems are also called tightly coupled systems because the processors share the memory or a clock.

There are several reasons for building such systems. One advantage is increased throughput. By increasing the number of processors, we can get more work done in a shorter time.

Another advantage of multiprocessor Operating system is multi-processors can save money compared to multiple single systems because the processors can share peripherals, cabinets and power supplies. If several programs are to operate on the same set of data, it is cheaper to store those data on one disk and to have all the processor share them, rather than to have many computers with local disks and many copies of the data.

Another advantage of multi-processor Operating system is that they increase reliability. If functions can be distributed properly among several

processors then the failure of one processor will not halt the system, but rather will only slow it down. If we have 10 processors and one fails, then each of the remaining nine processors must share the work of the failed processor. Thus the entire system runs only 10 percent slower, rather than failing together.

There are two most common types of multi-processor systems. One is symmetric multiprocessing model, in which each processor runs an identical copy of the operating system, and these copies communicate with one another as needed. Other is asymmetric multiprocessing model, in which each processor is assigned a specific task. Here one processor serves as master which controls the entire system and all the other processors serve as slaves. This model represents a master- slave relationship.

1.3.6 Distributed Systems

In this type of system all the computations are distributed among several processors. Distributed systems are also referred as loosely coupled systems because here the processors do not share memory or a clock. Instead, each processor has its own local memory. The processors communicate with one another through various communication lines, such as high speed buses or telephone lines. The processors in distributed system vary in size, function and are referred as sites, nodes, computers and so on depending on the context in which they are mentioned.

There are variety of reasons for building distributed systems, the major ones are :

Resource sharing. If different user are connected to one another with different resources, then the user at one site may be able to use the resources available at another.

Computation speedup. In distributed system**,** a particular computation will be partitioned into number of sub computations that can run concurrently. In

addition to this, distributed system may allow us to distribute the computations among various sites. It is called load sharing.

Reliability. It means in distributed system, if one site fails, the remaining sites will share the work of failed site.

Communication. When many sites are connected to one another by a communication network, the processes at different sites have the opportunity to exchange information. A User may initiate file transfer or communicate with one another via electronic mail. A user can send mail to another user at the same site or at a different site.

1.3.7 Real –Time Systems

Real-time operating systems are specially designed to respond to events that happen in real time. A real time operating system has well-defined, fixed time constrains. Processing must be done within the defined constraints, or else the system will fail. This feature is very useful in implementing systems such as an airline reservation system. In such a system, the response time should be very short because a customer's reservation is to be done while he/she waits.

There are two flavors of real-time systems. A hard real-time system guarantees that critical tasks complete at a specified time. A less restrictive type of real time system is soft real time system, where a critical real time task gets priority over other tasks, and remains that priority until it completes. The several areas in which this type is useful are multimedia, virtual reality and advance scientific projects such as exploration and planetary rovers. Because of the expanded uses for soft real-time functionality, it is finding its way into most current operating systems, including major versions of Unix and Windows NT Os.

1.4 Operating System Structures

An operating system could be designed as a huge, jumbled collection of processes without any structure. Any process could call any other process to request a service from it. The execution of a user command would usually involve the activation of a series of processes. While an implementation of this kind could be acceptable for small operating systems, it would not be suitable for large operating systems as the lack of a proper structure would make it extremely hard to specify , code, test and debug a large operating system.

A typical operating system that supports a multiprogramming environment can easily be tens of megabytes in length and its design, implementation and testing amounts to the undertaking of a huge software project. In this section we discuss design approaches indeed to handle the complexities of today's large operating systems.

1.4.1 Layered approach

Dijkstra suggested the layered approach to lessen the design and implementation complexities of an operating system. The layered approach divides the operating system into several layers. The functions of operating system are divided among these layers. Each layer has well-defined functionality and input-output interfaces with the two adjacent layers. Typically, the bottom layer is concerned with machine hardware and the top layer is concerned with users(Or operators).

The layered approach has all the advantages of modular design. In modular design, the system is divided into several modules and each module is designed independently. Likewise in layered approach each layer can be designed, coded and tested independently. Consequently the layered approach considerably simplifies the design, specification and implementation of an operating system. However, a drawback of the layered

approach is that operating system functions must be carefully assigned to various layers because a layer can make use only of the functionality provided by the layer beneath it.

A classic example of the layered approach is the THE operating system, which consists of six layers. Figure 1.4 shows these layers with their associated functions.

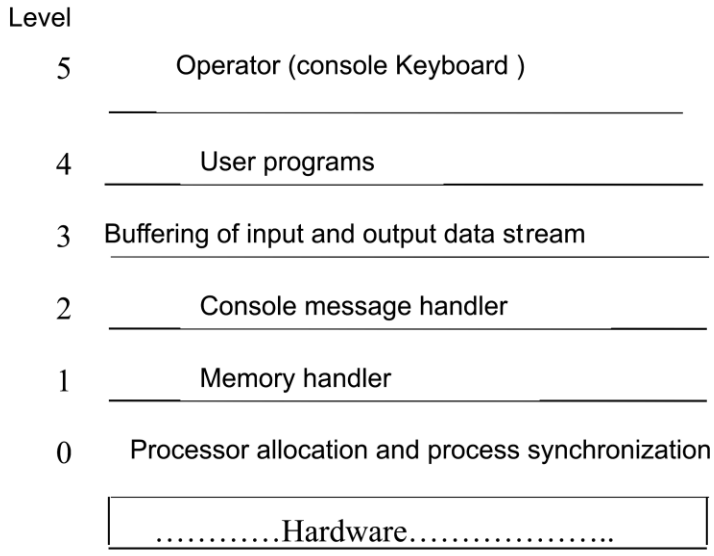


Figure 1.4 : Structure of the THE operating system.

1.4.2 The Kernel Based Approach

The Kernel-based design and structure of the operating system was suggested by Brinch Hansen. The Kernel (more appropriately called nucleus) is a collection of primitive facilities over which the rest of the operating system is built, using the functions provided by the kernel (see fig 1.5). Thus a kernel provides an environment to build operating system in which the designer has considerable flexibility because policy and optimization decisions are not made at the kernel level. It follows that a kernel should support only mechanisms and that all policy decisions should

be left to the outer layer. An operating system is an orderly growth of software over the kernel where all decisions regarding process scheduling, resource allocation, execution environment, file system, and resource protection and so on are made.

A kernel is a fundamental set of primitives that allows the dynamic creation and control of processes, as well as communication among them. Thus kernel only supports the notion of process and does not include the concept of a resource. However, as operating system has matured in functionality and complexity, more functionality has been relegated to the kernel. A kernel should contain a minimal set of functionality that is adequate to build an operating system with a given set of objectives. Including too much functionality in a kernel results in low flexibility at a higher level, whereas including too little functionality in a kernel results in low functional support at a higher level.

An outstanding example of a kernel is Hydra. Hydra is a kernel of an operating system for **c.mmp** a multiprocessor system. Hydra supports the notion of a resource and process, and provides mechanisms for the creation and representation of new types of resources and protected access to resources.

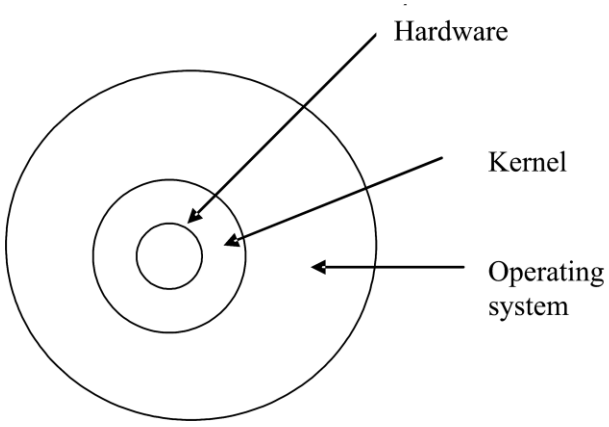


Figure 1.5: Structure of a kernel- based operating system

1.4.3 The virtual machine approach

In the virtual machine approach, a virtual machine software layer on the bare hardware of the machine gives the illusion that all machine hardware (i.e. the processor, main memory, secondary storage, etc) is at the sole disposal of each user. A user can execute the entire instruction set, including the privileged instructions. The virtual machine software creates this illusion by appropriately time multiplexing the system resources among all the users of the machine.

A user can also run a single user operating system on this virtual machine. The design of such a single user operating system can be very simple and efficient because it does not have to deal with the complications that arise due to multi-programming and protection. The virtual machine concept provides higher flexibility in that it allows different operating systems to run on different virtual machines. The efficient implementation of virtual machine software (e.g. VM/370), however , is a very difficult problem because virtual machine is huge and complex.

A classical example of this system is the IBM 370 system wherein the virtual machine software VM/370, provides a virtual machine to each user. When user logs on, VM/370 creates a new virtual machine (i.e a copy of the bare hardware of the IBM 370 system) for the user.

1.5 Summary

An operating system is a collection of programs that is an intermediary between a user and the computer hardware. It performs various functions such as process management, input output management, file management, managing use of main memory, providing security to user jobs and files etc.

In simple batch operating system, to speed up processing, jobs with the same needs were batched together and executed as a group. The primary objective of a multi-programmed operating system is to maximize the

number of programs executed by a computer in a specified period and keep all the units of the computer simultaneously busy. A time shared operating system allows a number of users to simultaneously use a computer. The primary objective of a time shared operating system is to provide fast response to each user of the computer. The primary objective of Parallel System is to increase the throughput by getting done more jobs in less time. A real time system is used in the control of physical systems. The response time of such system should match the needs of the physical system.

Since an operating system is large, a modularity is important. The design of system as sequence of layers is considered first. The virtual machine concept takes the layered approach to heart and treats the kernel of the operating system and hardware as though they were all hardware.

Self Assessment questions

1. Operating system is a----------
2. Spreadsheets and database systems are examples for ----------
3. Spooling is an acronym for ----------
4. A program in execution is referred to as ----------
5. THE operating system is an example for ---------- approach

1.6 Terminal Questions

1. Explain different functions of an operating system.
2. Explain multi-programmed Batched Operating System.
3. Discuss Time-sharing operating system.
4. What are the reasons for building distributed system?
5. What are the advantages of layered architecture? Give an example for layered architecture?
6. Discuss kernel based design approach for an operating system.
7. Explain virtual machine design approach with an example.

1.7 Answers to Self assessment questions and Terminal questions

Answers to Self Assessment Questions

1. System Software
2. Application software
3. Simultaneous Peripheral Operation On Line.
4. Process
5. Layered

Answers to Terminal Questions

1. Refer section 1.2
2. Refer section 1.3.2
3. Refer section 1.3.3
4. Refer section 1.3.6
5. Refer section 1.4.1
6. Refer section 1.4.2
7. Refer section 1.4.3

Unit 2 Process Management

Structure

2.1 Introduction

Current day computer system allows multiple programs to be loaded into memory and to be executed concurrently. This evolution requires more coordination and firmer control of the various programs. These needs resulted in the notion of a process. A Process can be simply defined as a program in execution. A process is created and terminated, and it allows

some or all of the states of process transition; such as New, Ready, Running, Waiting and Exit.

Thread is single sequence stream which allows a program to split itself into two or more simultaneously running tasks. Threads and processes differ from one operating system to another but in general, a thread is contained inside a process and different threads in the same process share some resources while different processes do not. An operating system that has thread facility, the basic unit of CPU utilization, is a thread. A thread has or consists of a program counter (PC), a register set, and a stack space. Threads are not independent of one other like processes as a result threads share with other threads their code section, data section, OS resources also known as task, such as open files and signals.

Objectives:

At the end of this unit, you will be able to understand:

What is process, process state, process control block, process scheduling, operation on processes, co-operating processes and about Threads.

2.2 What is 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. In reality, the CPU switches back and forth among processes.

2.3 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 (Blocked): Process waiting for an event to occur.
- Ready: Process waiting for CPU.
- Terminated: Process has finished execution.

Locally, the 'Running' and 'Ready' states are similar. In both cases the process is willing to run, only in the case of 'Ready' state, there is temporarily no CPU available for it. The 'Blocked' state is different from the 'Running' and 'Ready' states in that the process cannot run, even if the CPU is available.

These above states are arbitrary and very between operating systems. Certain operating systems also distinguish among more finely delineating process states. It is important to realize that only one process can be running on any processor at any instant. Many processes may be ready and waiting. A state diagram (Figure 2.1) is used to diagrammatically represent the states and also the events that trigger the change of state of a process in execution.

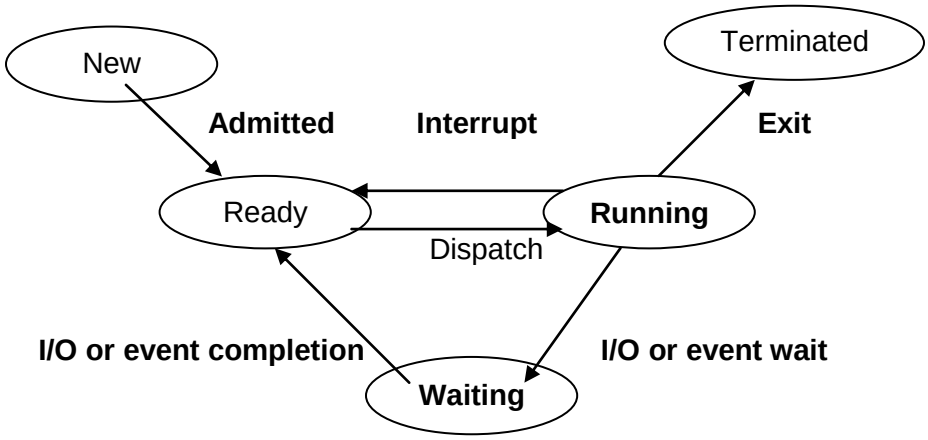


Figure 2.1: Process state diagram

2.4 Process Control Block

Every process has a number and a process control block (PCB) represents a process in an operating system. The PCB serves as a repository of information about a process and varies from process to process. The PCB contains information that makes the process an active entity. A PCB is shown in Figure 2.2. It contains many pieces of information associated with a specific process, including these:

- **Process state:** The state may be new, ready, running, waiting, halted and so on.
- **Program counter:** It is a register and it holds the address of the next instruction to be executed.
- **CPU Registers:** These registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers and general-purpose registers, plus any condition code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterwards.
- **CPU scheduling information:** This information includes a process priority, pointer to scheduling queues and other scheduling parameters.
- **Memory management information**: This information includes the value of the base and limit registers, the page tables, or the segment tables depending on the memory system used by the operating system.
- **Accounting information:** This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers and so on.
- **I/O status information:** The information includes the list of I/O devices allocated to the process, a list of open files, and so on.

The PCB simply serves as the repository of any information that may vary from process to process.

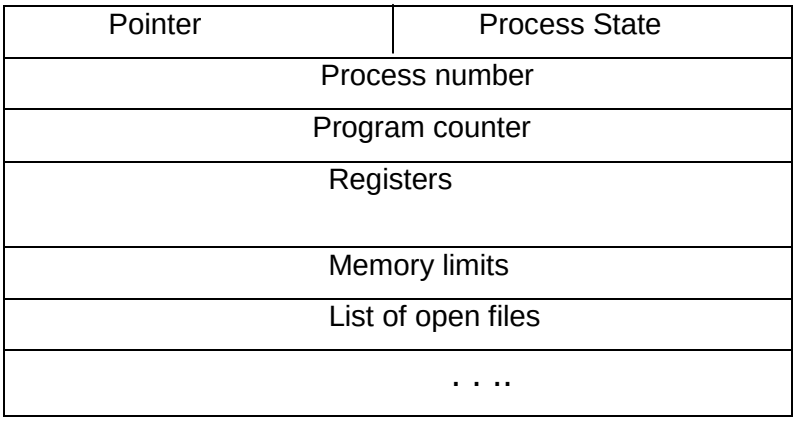


Figure 2.2: The process control Block

2.5 Process Scheduling

The main objective of multiprogramming is to see that some process is always running so as to maximize CPU utilization whereas 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 onto 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 2.3). Once a process terminates, entries for this process in all the queues are deleted. The PCB and resources allocated to it are released.

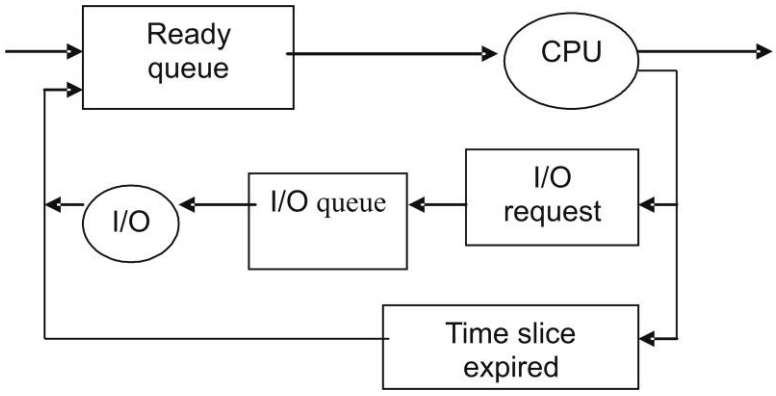


Figure 2.3: Queueing diagram of process scheduling

2.5.1 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 as processes execute for short intervals before waiting for I/O requests. Hence, short-term scheduler must be very fast or 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 multi-programming. So the long-term scheduler determines this degree of multi-programming. 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 multi-programming.

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 whereas 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 as well as the I/O devices will be busy. If this is not to be so, then either the CPU is busy and I/O devices are idle or vice-versa. Time sharing systems usually do not require the services of 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 multi-programming meaning more number of ready processes than the system can handle. The medium-term scheduler(Figure 2.4) 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 multi-programming. Sometime later these swapped processes are reintroduced into memory to join the ready queue.

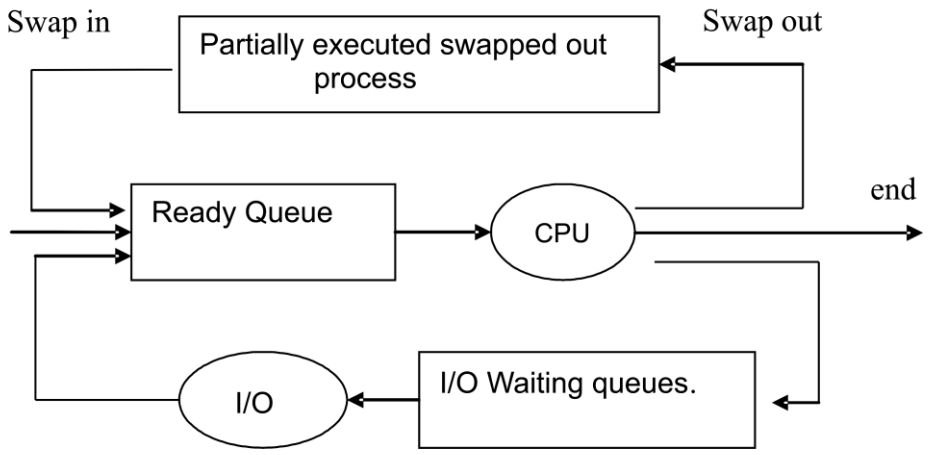


Figure 2.4: Medium term scheduling

2.5.2 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 2.5). 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.

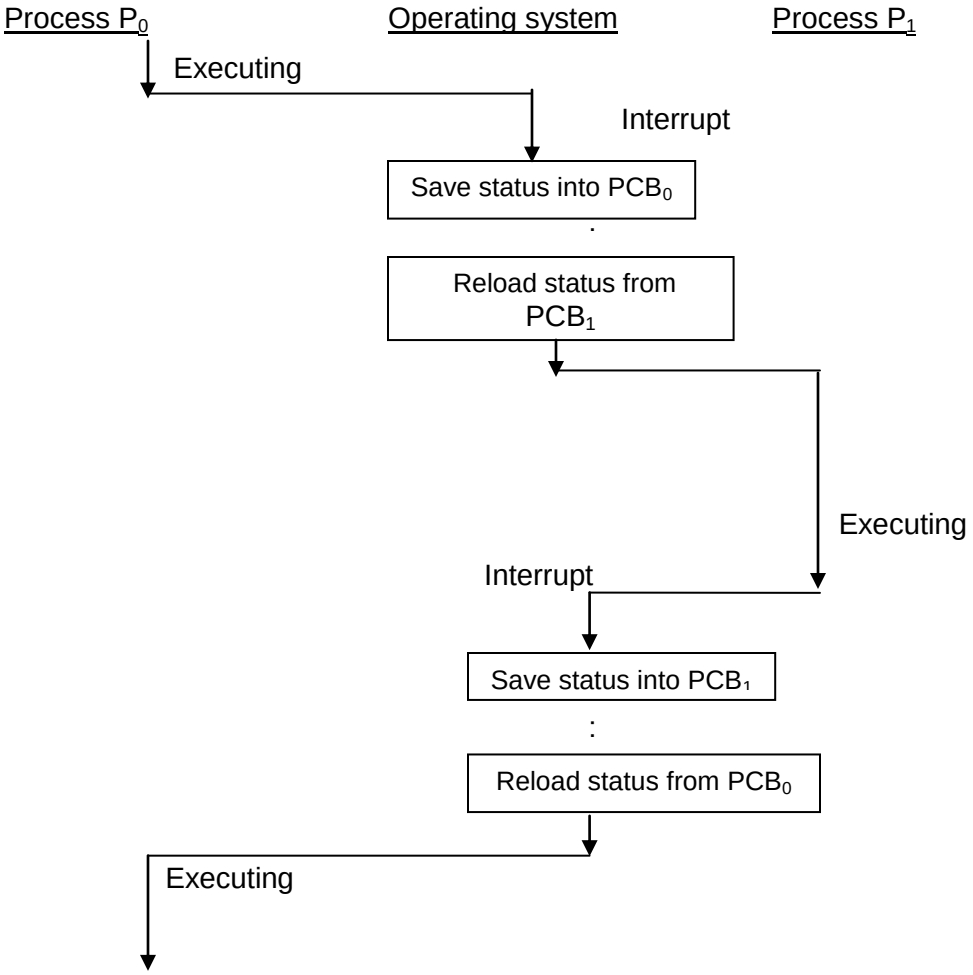


Figure 2.5: CPU switch from process to process

2.6 Operation on Processes

The processes in the system can execute concurrently, and must be created and deleted dynamically. Thus the operating system must provide a mechanism for process creation and termination.

2.6.1 Process Creation

During the course of execution, a process may create several new proceses using a create-process system call. The creating process is called a parent process and the new processes are called the children of that process. Each of these new processes may in turn create other processes, forming a tree of processes.

A process in general need certain resources(CPU time , memory ,I/O devices) to accomplish its task. When a process creates a sub process, the sub process may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process. The parent may have to partition its resources among its children, or it may be able to share some resources(such as memory or files) among several of its children.

In addition to the various physical and logical resources that a proess obtains when it is created, initialization data(input) may be passed along by the parent process to the child process.

When a process creates a new process, two possibilities exit in terms of execution:

- The parent continues to execute concurrently with its children.
- The parent waits until some or all of its children have terminated.

There are also two possibilities in terms of the address space of the new process, the first possibility is,

- The child process is a duplicate of the parent process.
  An example for this is UNIX operating system in which each process is identified by its process identifier, which is a unique integer. A new process is created by the fork system call. The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process.

The second possibility is,

- The child process has a program loaded into it.
  An example for this implementation is the DEC VMS operating system, it creates a new process, loads a specified program into that process, and starts it running. The Microsoft Windows/NT operating system supports both models: the parent's address space may duplicated, or the parent may specify the name of a program for the operating system to load into the address space of the new process.

2.6.2 Process Termination

A process terminates when it finishes executing its last statement and asks the operating system to delete it by using the exit system call. At that time, the process should return the data (output) ti its parent process via the wait system call. All the resources of the process, including physical and virtual memory, open files, and I/O buffers, are deallocated by the operating system.

Only a parent process can cause termination of its children via abort system call. For that a parent needs to know the identities of its children. When one process creates a new process, the identity of the newly created process is passed to the parent.

A parent may terminate the execution of one of its children for a variety of reasons, such as:

- The child has exceeded its usage of some of the resources that it has been allocated.
- The task assigned to the child is no longer required.
- The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

To illustrate process execution and termination, let us consider UNIX system. In UNIX system a process may terminate by using the exit system

call and its parent process may wait for that event by using the wait system call. The wait system call returns the process identifier of a terminated child, so the parent can tell which of the possibly many children has terminated. If the parent terminates, however all the children are terminated by the operating system. Without a parent, UNIX does not know whom to report the activities of a child.

2.7 Co-operating Processes

The processes executing in the operating system may be either independent processes or cooperating processes. A process is said to be independent if it cannot affect or be affected by the other processes executing in the system. Any process that does not share any data with any other process is independent. On the other hand, a process is co-operating if it can affect or be affected by the other processes executing in the system. Clearly any process that shares data with other processes is a co-operating process.

There are several advantages of providing an environment that allows process co-operation:

Information sharing: Since several users may be interested in the same piece of information, we must provide an environment to allow concurrent access to these types of resources.

Computation speedup: If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others.

Modularity: We may want to construct the system in a modular fashion dividing the system functions into separate processes.

Convenience: Even an individual user may have many tasks to work on at one time. For instance a user may be editing, printing and compiling in parallel.

2.8 Threads

What is Thread ?

Thread is a single sequence stream which allows a program to split itself into two or more simultaneously running tasks. It is a basic unit of CPU utilization, and consists of a program counter, a register set and a stack space. Because threads have some of the properties of processes, they are sometimes called lightweight processes. In a process, threads allow multiple executions of streams. Threads are not independent of one other like processes. As a result threads shares with other threads their code section, data section, OS resources also known as task, such as open files and signals.

Some of the similarities between processes and Threads are as follows.

- Like processes threads share CPU and only one thread is running at a time.
- Like processes, threads within processes execute sequentially.
- Like processes, thread can create children.
- And like process if one thread is blocked, another thread can run.

Some of the differences between processes and Threads are as follows.

- Unlike processes, threads are not independent of one another.
- Unlike processes, all threads can access every address in the task.
- Unlike processes, threads are designed to assist one other. (processes might or might not assist one another because processes may originate from different users.)

2.8.1 Why Threads?

Following are some reasons why we use threads in designing operating systems.

1. A process with multiple threads make a great server. For example printer server.

2. Because threads can share common data, they do not need to use interprocess communication.
3. Because of the very nature, threads can take advantage of multiprocessors.

Threads are cheap in the sense that:

1. They only need a stack and storage for registers. Therefore, threads are cheap to create.
2. Threads use very little resources of an operating system in which they are working. That is, threads do not need new address space, global data, program code or operating system resources.
3. Context switching is fast when working with threads. The reason is that we only have to save and/or restore PC, SP and registers.

2.8.2 Advantages of Threads over Multiple Processes

- **Context Switching**: Threads are very inexpensive to create and destroy, and they are inexpensive to represent. For example, they require space to store, the PC, the SP, and the general-purpose registers, but they do not require space to share memory information, Information about open files of I/O devices in use, etc. With so little context, it is much faster to switch between threads. In other words, it is relatively easier for a context switch using threads.
- **Sharing:** Threads allow the sharing of a lot resources that cannot be shared in process. For example, sharing code section, data section, Operating System resources like open file etc.

2.8.3 Disadvantages of Threads over Multiprocesses

- **Blocking:** The major disadvantage is that if the kernel is single threaded, a system call of one thread will block the whole process and CPU may be idle during the blocking period.

- **Security:** Since there is an extensive sharing among threads there is a potential problem of security. It is quite possible that one thread over writes the stack of another thread (or damaged shared data) although it is very unlikely since threads are meant to cooperate on a single task.

2.8.4 Types of Threads

There are two main types of threads:

- User-Level Threads (ULTs)
- Kernel-Level Threads (KLTs)

User-Level Threads (ULTs) : User-level threads implement in user-level libraries, rather than via systems calls. So thread switching does not need to call operating system and to interrupt the kernel. In fact, the kernel knows nothing about user-level threads and manages them as if they were single-threaded processes. In this level, all thread management is done by the application by using a thread library.

Advantages:

The most obvious advantage of this technique is that a user-level threads package can be implemented on an Operating System that does not support threads. Some other advantages are:

- User-level threads does not require modification to operating systems.
- Simple Representation:
  Each thread is represented simply by a PC, registers, stack and a small control block, all stored in the user process address space.
- Simple Management:
  This simply means that creating a thread, switching between threads and synchronization between threads can all be done without intervention of the kernel.
- Fast and Efficient:
  Thread switching is not much more expensive than a procedure call.

Disadvantages:

- There is a lack of coordination between threads and operating system kernel. Therefore, process as whole gets one time slice irrespective of whether process has one thread or 1000 threads within. It is up to each thread to relinquish control to other threads.
- User-level threads requires non-blocking systems call i.e., a multi-threaded kernel. Otherwise, entire process will blocked in the kernel, even if there are runable threads left in the processes. For example, if one thread causes a page fault, the process blocks.

Kernel-Level Threads (KLTs): Kernel Threads (Threads in Windows) are threads supplied by the kernel. All thread management is done by kernel. No thread library exists. The kernel knows about and manages the threads. No runtime system is needed in this case. Instead of thread table in each process, the kernel has a thread table that keeps track of all threads in the system. In addition, the kernel also maintains the traditional process table to keep track of processes. Operating Systems kernel provides system call to create and manage threads.

Advantages:

- Because kernel has full knowledge of all threads, Scheduler may decide to give more time to a process having large number of threads than process having small number of threads.
- Kernel-level threads are especially good for applications that frequently block.
- kernel routines can be multi-threaded

Disadvantages:

- The kernel-level threads are slow and inefficient. For instance, threads operations are hundreds of times slower than that of user-level threads.

- Since kernel must manage and schedule threads as well as processes. It requires a full thread control block (TCB) for each thread to maintain information about threads. As a result, there is significant overhead and increase in kernel complexity.

2.8.5 Combined ULT/KLT Approaches:

Here the idea is to combine the best of both approaches

Solaris is an example of an OS that combines both ULT and KLT

- Thread creation done in the user space
- Bulk of scheduling and synchronization of threads done in the user space
- The programmer may adjust the number of KLTs
- Process includes the user's address space, stack, and process control block
- User-level threads (threads library) invisible to the OS are the interface for application parallelism
- Kernel threads the unit that can be dispatched on a processor
- Lightweight processes (LWP) each LWP supports one or more ULTs and maps to exactly one KLT

2.9 Summary

A process is a program in execution. As process executes, it changes it state. Each process may be in one of the following states: New, Ready, Running, Waiting, or Halted. Each process is represented in the operating system by its own process control block(PCB). The PCB serves as a repository of information about a process and varies from process to process.

The process in the system can execute concurrently. There are several reasons for allowing concurrent execution; information sharing, computation speedup, modularity, and convenience. The processes executing in the

operating system may be either independent processes or co-operating processes. Co-operating processes must have the means to communicate with each other.

Co-operating processes that directly share a logical address space can be implemented as lightweight processes or threads. A thread is a basic unit of CPU utilization, and it shares with peer threads its code section, data section, and operating system resources, collectively known as task.

Self Assessment Questions

1. _____ is single sequence stream which allows a program to split itself into two or more simultaneously running tasks.

2. _____ serves as a repository of information about a process and varies from process to process.

3. A process is _____ if it can affect or be affected by the other processes executing in the system.

4. Because threads can share common data, they do not need to use _____.
5. Co-operating processes that directly share a logical address space can be implemented as _____.

2.10 Terminal Questions

1. Explain with diagram all possible states a process visits during the course of its execution.
2. What is PCB? What are the useful informations available in PCB?
3. Discuss different types of schedulers.
4. Explain different operations on process.
5. What are Co-operating processes? What are the advantages of process co-operation?
6. What are the similarities and differences between Processes and Threads?

7. What are different types of Threads? Explain.
8. Explain the features of any operating system which makes use of the Combined ULT/KLT Approaches.

2.11 Answers to Self assessment questions and Terminal questions

Answers to Self Assessment Questions

1. Thread
2. Process Control block
3. Co-operating
4. Interprocess Communication
5. Lightweight processes or threads

Answers to Terminal Questions.

1. Refer section 2.3
2. Refer section 2.4
3. Refer section 2.5.1
4. Refer section 2.6
5. Refer section 2.7
6. Refer section 2.8
7. Refer section 2.8.4
8. Refer section 2.8.5

Unit 3 CPU Scheduling Algorithms

Structure

3.1 Introduction

The CPU scheduler selects a process from among the ready processes to execute on the CPU. CPU scheduling is the basis for multi-programmed 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 multi-programming 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 multi-programming, 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.

Scheduling is a fundamental operating-system function. Almost all computer resources are scheduled before use. The CPU is, of course, one of the primary computer resources. Thus, its scheduling is central to operating-system design.

Objectives:

At the end of this unit, you will be able to understand:

Basic Scheduling Concepts, Different Scheduling algorithms, and Evolution of these algorithms.

3.2 Basic concepts of Scheduling

3.2.1 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 3.1). 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.

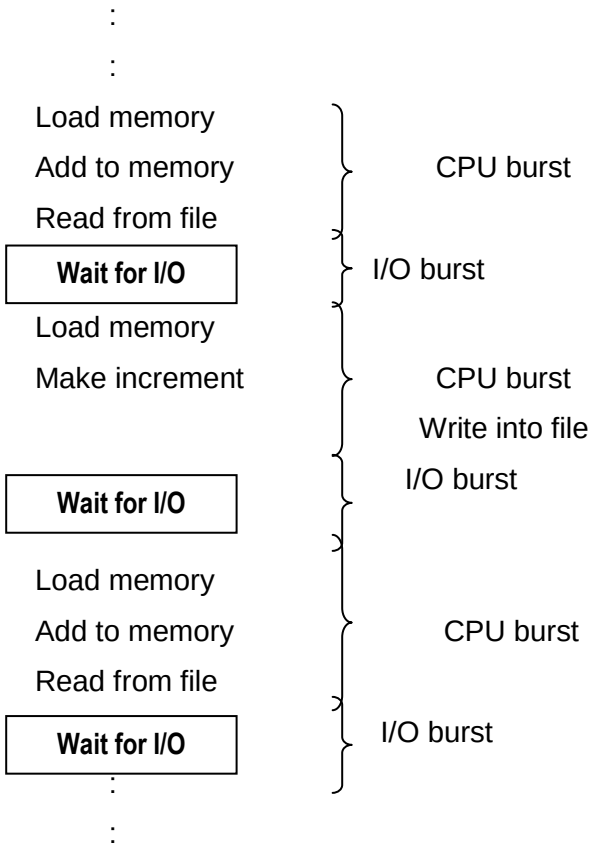


Figure 3.1: CPU and I/O bursts

3.2.2 CPU Scheduler

Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The short-term scheduler (or CPU scheduler) carries out the selection process. The scheduler selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them.

Note that the ready queue is not necessarily a first-in, first-out (FIFO) queue. As we shall see when we consider the various scheduling algorithms, a ready queue may be implemented as a FIFO queue, a priority queue, a tree,

or simply an unordered linked list. Conceptually, however, all the processes in the ready queue are lined up waiting for a chance to run on the CPU. The records in the queue are generally PCBs of the processes.

3.2.3 Preemptive/ Non preemptive scheduling

CPU scheduling decisions may take place under the following four circumstances. 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 non-preemptive. In non-preemptive 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. Windows used non-preemptive scheduling up to Windows 3.x, and started using pre-emptive scheduling with Win95. Note that pre-emptive scheduling is only possible on hardware that supports a timer interrupt. It is to be noted that pre-emptive scheduling can cause problems when two processes share data, because one process may get interrupted in the middle of updating shared data structures.

Preemption also has an effect on the design of the operating-system kernel. During the processing of a system call, the kernel may be busy with an active on behalf *of* a process. Such activities may involve changing important kernel data (for instance, I/O queues). What happens if the process is preempted in t1: middle of these changes, and the kernel (or the device driver) needs to read (modify the same structure). Chaos ensues.

Some operating systems, including most versions of UNIX, deal with this problem by waiting either for a system call to complete, or for an I/O block to take place, before doing a context switch. This scheme ensures that the kernel structure is simple, since the kernel will not preempt a process while the kernel data structures are in an inconsistent state. Unfortunately, this kernel execution model is a poor one for supporting real-time computing and multiprocessing.

3.2.4 Dispatcher

Another component involved in the CPU scheduling function is the *dispatcher.* The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler. This function involves:

- Switching context
- Switching to user mode
- Jumping to the proper location in the user program to restart that program

The dispatcher should be as fast as possible, given that it is invoked during every process switch. The time it takes for the dispatcher to stop one process and start another running is known as the *dispatch latency.*

3.2.5 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:** We want to keep the CPU as busy as possible. CPU utilization 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 whereas 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.

3.3 Scheduling Algorithms

Scheduling algorithms differ in the manner in which the CPU selects a process in the ready queue for execution. In this section, we have described several of these algorithms.

3.3.1 First Come First Served scheduling algorithm

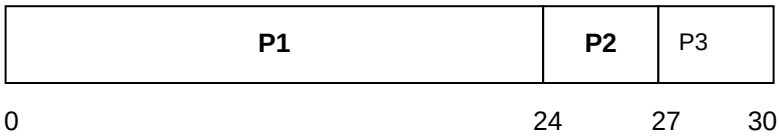
This is one of the very brute force algorithms. A process that requests for the CPU first is allocated the CPU first. Hence, the name first come first serve. The FCFS algorithm is implemented by using a first-in-first-out (FIFO) queue structure for the ready queue. This queue has a head and a tail. When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

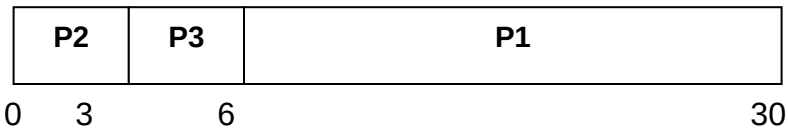
P2 = 24 msecs

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.

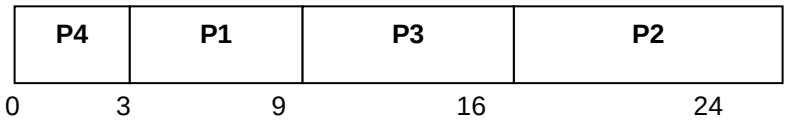
Since the algorithm is non-preemptive in nature, it is not suited for time sharing systems.

3.3.2 Shortest-Job- First Scheduling

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.

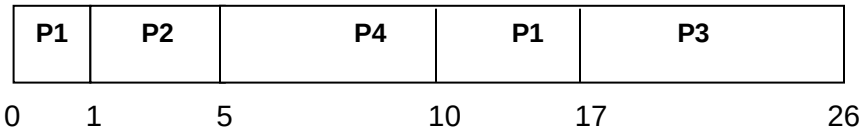
But the SJF scheduling algorithm is provably optimal and thus serves as a benchmark to compare other CPU scheduling algorithms.

SJF algorithm could be either preemptive or non-preemptive. 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 non-preemptive 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.

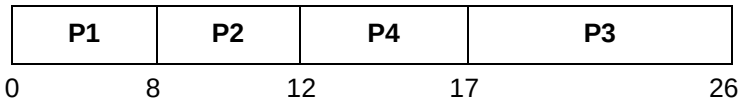
Process	Arrival time (msecs)	Burst time (msecs)
P1	0	8
P2	1	4
P3	2	9
P4	3	5

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



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

If non-preemptive SJF scheduling is used, the result is as follows:



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

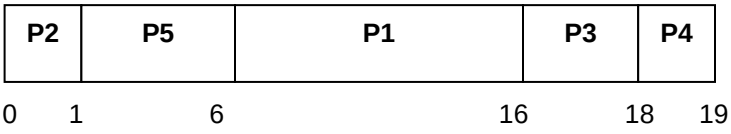
3.3.3 Priority Scheduling

In 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 in the Gantt chart:



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 non-preemptive. 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 non-preemptive 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.

3.3.4 Round-Robin Scheduling

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(Quantum). 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.

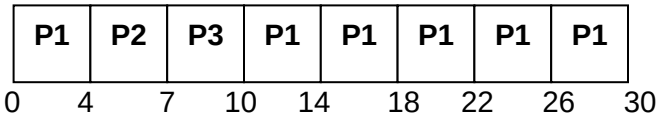
The ready queue in this case is a FIFO queue with new processes joining the tail of the queue. The CPU scheduler picks processes from the head of the queue for allocating the CPU. The first process at the head of the queue gets to execute on the CPU at the start of the current time slice and is deleted 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 thereby 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 thereby making way for the next process.

Consider the same example explained under FCFS algorithm.

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

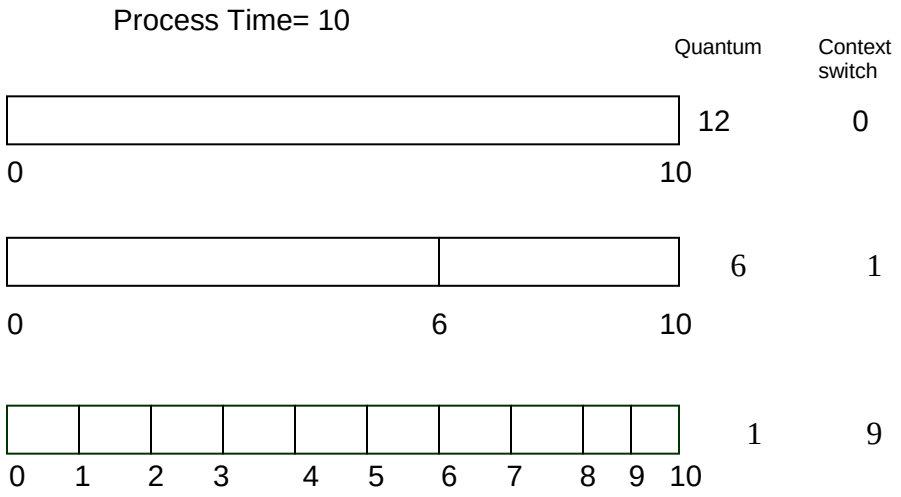
Let the duration of a time slice be 4 msecs, which is to say CPU switches between processes every 4 msecs in a round-robin fashion. The Gantt chart below shows the scheduling of processes.



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. A comparison of time slices of varying duration and the context switches they generate on only one process of 10 time units is shown below..



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.

3.3.5 Multi-level 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 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 3.2).

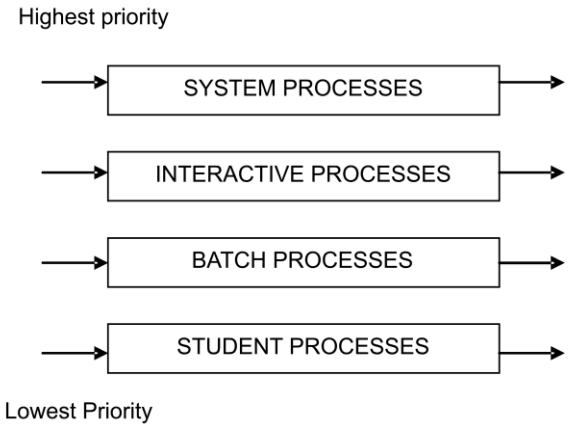


Figure 3.2 : Multilevel Queue scheduling.

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.

3.3.6 Multi-level Feedback Queue Scheduling

In the previous multilevel queue scheduling algorithm, processes once 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 in Figure 3.3.

As in Figure, 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.

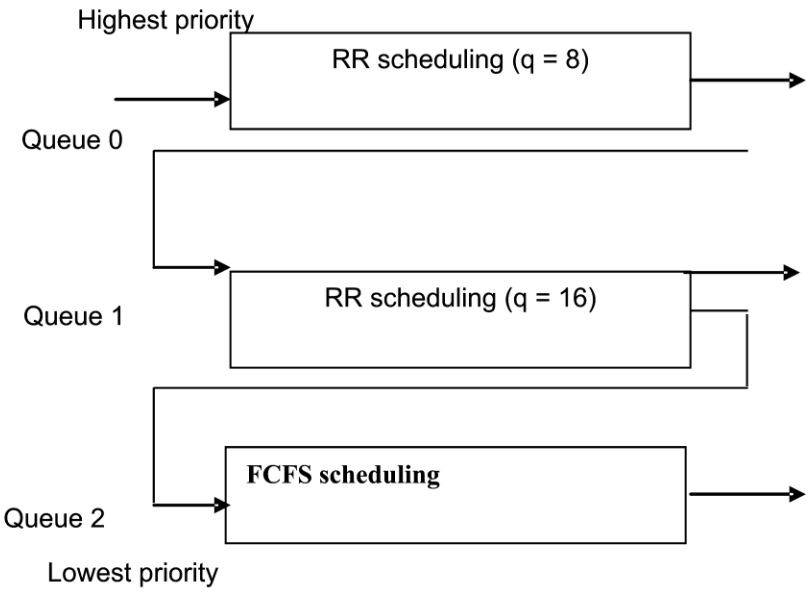


Figure 3.3: Multilevel Feedback Queue Scheduling

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.

3.3.7 Multiple- Processor Scheduling

When multiple processors are available, then the scheduling gets more complicated, because now there is more than one CPU which must be kept busy and in effective use at all times. Multi-processor systems may be **heterogeneous,** (different kinds of CPUs), or **homogenous,** (all the same kind of CPU). This book will restrict its discussion to homogenous systems. In homogenous systems any available processor can then be used to run any processes in the queue. Even within homogenous multiprocessor, there are sometimes limitations on scheduling. Consider a system with an I/O device attached to a private bus of one processor. Processes wishing to use that device must be scheduled to run on that processor, otherwise the device would not be available.

If several identical processors are available, then load sharing can occur. It would be possible to provide a separate queue for each processor. In this case, however one processor could be idle, with an empty queue, while another processor was very busy. To prevent this situation, we use a common ready queue. All processes go into one queue and are scheduled onto any available processor.

In such scheme, one of two scheduling approaches may be used. In one approach, each processor is self –scheduling,. Each processor examines the common ready queue and selects a process to execute. We must ensure that two processors do not choose the same process, and that processes are not lost from the queue. The other approach avoids this

problem by appointing one processor as scheduler for other processors, thus creating a master-slave structure.

Some systems carry this structure one step further, by having all scheduling decisions, I/O processing, and other system activities handled by one single processor- the master server. The other processors only execute user code. This asymmetric multiprocessing is far simpler than symmetric multiprocessing, because only one processor accesses the system data structures, alleviating the need for data sharing.

3.3.8 Real-Time Scheduling

In unit 1 we gave an overview of real-time operating systems. Here we continue the discussion by describing the scheduling facility needed to support real-time computing within a general purpose computer system.

Real-time computing is divided into two types. Hard real-time systems and soft real-time systems. Hard real time systems are required to complete a critical task within a guaranteed amount of time. A process is submitted along with a statement of the amount of time in which it needs to complete or perform I/O. The scheduler then either admits the process, guaranteeing that the process will complete on time, or rejects the request as impossible. This is known as resource reservation. Such a guarantee requires that the scheduler knows exactly how long each type of operating system function takes to perform, and therefore each operation must be guaranteed to take a maximum amount of time. Such a guarantee is impossible in a system with secondary storage or virtual memory, because these subsystems cause unavoidable and unforeseeable variation in the amount of time to execute a particular process. Therefore, hard real-time systems are composed of special purpose software running on hardware dedicated to their critical process, and lack the functionality of modern computers and operating systems.

Soft real-time computing is less restrictive. It requires that critical processes receive priority over less fortunate ones. Implementing soft real-time functionality requires careful design of the scheduler and related aspects of the operating system. First, the system must have priority scheduling, and real-time processes must have the highest priority. The priority of real-time processes must not degrade over time, even though the priority of non-real time processes may. Secondly the dispatch latency must be small. The smaller the latency, the faster a real-time process can start executing once it is runable. To keep dispatch latency low, we need to allow system calls to be preemptible.

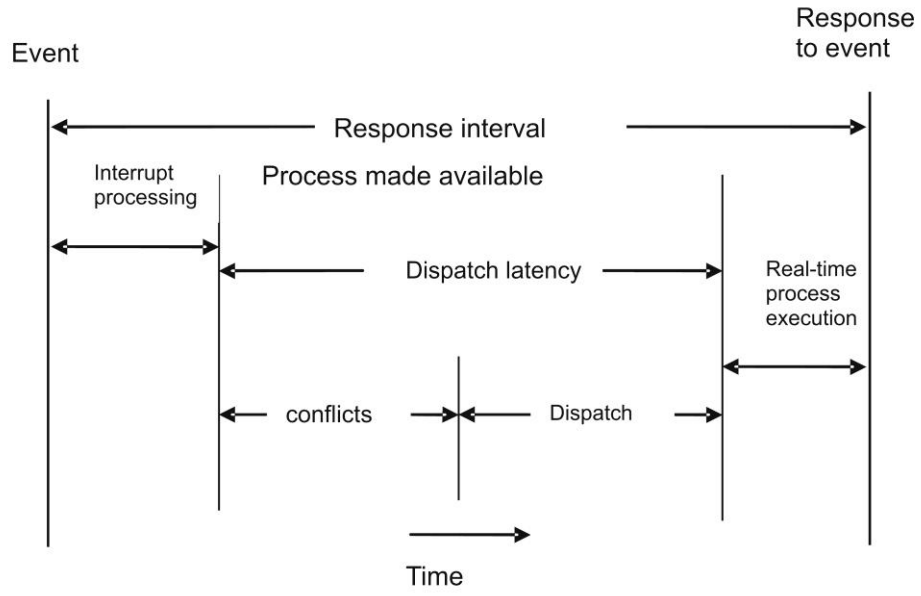


Figure 3.4 : Dispatch latency.

Figure 3.4 shows a make up of dispatch latency. The conflict phase of dispatch latency has three components:

1. preemption of any process running in the kernel.
2. low-priority processes releasing resources needed by the high-priority process

3. context switching from the current process to the high-priority process.

As an example in Solaris 2, the dispatch latency with preemption disabled is over 100 milliseconds. However, the dispatch latency with preemption enabled is usually to 2 milliseconds.

3.4 Evaluation of CPU Scheduling Algorithms

We have many scheduling algorithms, each with its own parameters. As a result, selecting an algorithm can be difficult. To select an algorithm first we must define, the criteria on which we can select the best algorithm. These criteria may include several measures, such as:

- Maximize CPU utilization under the constraint that the maximum response time is 1 second.
- Maximize throughput such that turnaround time is (on average) linearly proportional to total execution time.

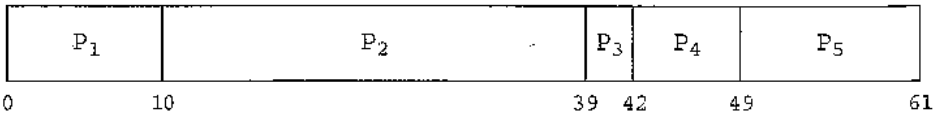
  Once the selection criteria have been defined, we use one of the following different evaluation methods.

3.4.1 Deterministic Modeling

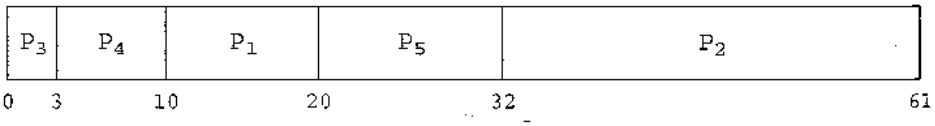
If a specific workload is known, then the exact values for major criteria can be fairly easily calculated, and the "best" determined. For example, consider the following workload (with all processes arriving at time 0), and the resulting schedules determined by three different algorithms:

Process	Burst Time
P1	10
P2	29
P3	3
P4	7
P5	12

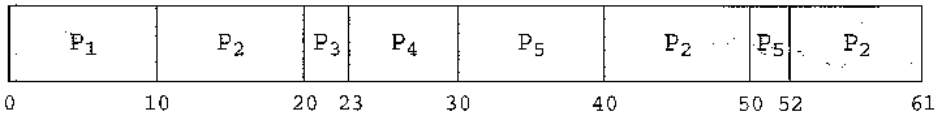
FCFS:



Non-preemptive SJF:



Round Robin:



The average waiting times for FCFS, SJF, and RR are 28ms, 13ms, and 23ms respectively. Deterministic modeling is fast and easy, but it requires specific known input, and the results only apply for that particular set of input. However, by examining multiple similar cases, certain trends can be observed. (Like the fact that for processes arriving at the same time, SJF will always yield the shortest average wait time.)

3.4.2 Queuing Models

Specific process data is often not available, particularly for future times. However, a study of historical performance can often produce statistical descriptions of certain important parameters, such as the rate at which new processes arrive, the ratio of CPU bursts to I/O times, the distribution of CPU burst times and I/O burst times, etc.

Armed with those probability distributions and some mathematical formulas, it is possible to calculate certain performance characteristics of individual waiting queues. For example, **Little's Formula** says that for an average queue length of N, with an average waiting time in the queue of W, and an

average arrival of new jobs in the queue of Lambda, the these three terms can be related by:

N = Lambda * W

Queuing models treat the computer as a network of interconnected queues, each of which is described by its probability distribution statistics and formulas such as Little's formula. Unfortunately real systems and modern scheduling algorithms are so complex as to make the mathematics intractable in many cases with real systems.

3.4.3 Simulations

Another approach is to run computer simulations of the different proposed algorithms (and adjustment parameters) under different load conditions, and to analyze the results to determine the "best" choice of operation for a particular load pattern. Operating conditions for simulations are often randomly generated using distribution functions similar to those described above. A better alternative when possible is to generate ***trace tapes***, by monitoring and logging the performance of a real system under typical expected work loads. These are better because they provide a more accurate picture of system loads, and also because they allow multiple simulations to be run with the identical process load, and not just statistically equivalent loads. A compromise is to randomly determine system loads and then save the results into a file, so that all simulations can be run against identical randomly determined system loads.

Although trace tapes provide more accurate input information, they can be difficult and expensive to collect and store, and their use increases the complexity of the simulations significantly. There are also some questions as to whether the future performance of the new system will really match the past performance of the old system. (If the system runs faster, users may take fewer coffee breaks, and submit more processes per hour than under

the old system. Conversely if the turnaround time for jobs is longer, intelligent users may think more carefully about the jobs they submit rather than randomly submitting jobs and hoping that one of them works out.)

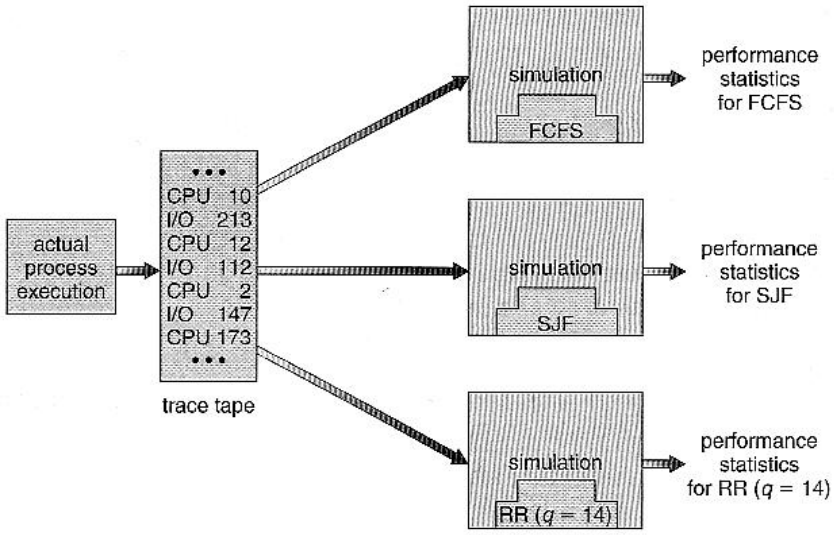


Fig. 3.5: Evaluation of CPU schedulers by simulation

3.4.4 Implementation

Implementation

The only real way to determine how a proposed scheduling algorithm is going to operate is to implement it on a real system. For experimental algorithms and those under development, this can cause difficulties and resistances among users who don't care about developing OS's and are only trying to get their daily work done. Even in this case, the measured results may not be definitive, for at least two major reasons: (1) System workloads are not static, but change over time as new programs are installed, new users are added to the system, new hardware becomes available, new work projects get started, and even societal changes. (For example the explosion of the Internet has drastically changed the amount of network traffic that a system sees and the importance of handling it with

rapid response times.) (2) As mentioned above, changing the scheduling system may have an impact on the workload and the ways in which users use the system.

Most modern systems provide some capability for the system administrator to adjust scheduling parameters, either on the fly or as the result of a reboot or a kernel rebuild.

3.5 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 non-preemptive. 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 non-preemptive 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 have been discussed.

First-come, first-served (FCFS) scheduling is the simplest scheduling algorithm, but it can cause short processes to wait for very long processes. Shortest-Job-First (SJF) scheduling is provably optimal, providing the shortest average waiting time. Implementing SJF scheduling is difficult because predicting the length of the next CPU burst is difficult. The SJF algorithm is a special case of the general priority-scheduling algorithm, which simply allocates the CPU to the highest-priority process. Both priority and SJF scheduling may suffer from starvation. Aging is a technique to prevent starvation. Round robin (RR) scheduling is more appropriate for a time-shared (inter-active) system. RR scheduling allocates the CPU to the first process in the ready queue for q time units, where q is the time

quantum. After *q* time units, if the process has not relinquished the CPU, it is preempted and the process is put at the tail of the ready queue.

Multilevel queue algorithms allow different algorithms to be used for various classes of processes. The most common is a foreground interactive queue, which uses RR scheduling, and a background batch queue, which uses FC scheduling. Multilevel feedback queues allow processes to moves from one queue to another. Finally, we also discussed various algorithm evaluation models.

Self Assessment Questions

1. _____ selects a process from among the ready processes to execute on the CPU.
2. The time taken by the Dispatcher to stop one process and start another running is known as _____.
3. The interval of time between submission and completion of a process is called _____.
4. A solution to starvation is _____.
5. _____ systems are required to complete a critical task within a guaranteed amount of time.

3.6 Terminal Questions

1. Explain Preemptive and Non-preemptive scheduling approaches.
2. Discuss First come First served scheduling algorithm.
3. What are the Drawbacks of Shortest-Job- First Scheduling algorithm?
4. Why are Round-Robin Scheduling algorithm designed for time-sharing systems? Explain.
5. Write a note on Multi-level Queue Scheduling.
6. Write a note on Dispatch Latency.

7. Explain any two evaluation methods used for the evaluation of scheduling algorithms.

3.7 Answers to Self Assessment Questions and Terminal Questions

Answers to Self Assessment Questions

1. CPU Schedular
2. Dispatch Latency
3. Turnaround time
4. Aging

5 . Hard real time.

Answers to Terminal Questions.

1. Refer section 3.2.3
2. Refer section 3.3.1
3. Refer section 3.3.2
4. Refer section 3.3.4
5. Refer section 3.3.5
6. Refer section 3.3.8
7. Refer section 3.4

Unit 4 Process Synchronization

Structure

4.1 Introduction

A co-operating process is one that can affect or be affected by the other processes executing in the system. These processes may either directly share a logical address space (that is, both code and data), or be allowed to share data only through files. Concurrent access to shared data may result in data inconsistency. Maintaining data consistency requires mechanism to

ensure the orderly execution of co-operating processes. In this unit we discuss, how co-operating processes can communicate and various mechanisms to ensure the orderly execution of co-operating processes that share a logical address space, so that data consistency is maintained.

Objectives:

At the end of this unit, you will be able to understand:

About Interprocess Communication, Critical Section Problem , Two process and Multiple Process solution for the Critical Section problem, about Semaphores , monitors and Hardware Assistance for Mutual Exclusion .

4.2 Interprocess Communication

Communication of co-operating processes shared-memory environment requires that these processes share a common buffer pool, and that the code for implementing the buffer be explicitly written by the application programmer. Another way to achieve the same effect is for the operating system to provide the means for co-operating processes to communicate with each other via an inter-process-communication (IPC) facility.

IPC provides a mechanism to allow processes to communicate and to synchronize their actions. Inter-process-communication is best provided by a message system. Message systems can be defined in many different ways. Message-passing systems also have other advantages.

Note that the shared-memory and message system communication schemes are not mutually exclusive, and could be used simultaneously within a single operating system or even a single process.

4.2.1 Basic Structure

The function of a message system is to allow processes to communicate with each other without the need to resort to shared variables. An IPC

facility provides at least the two operations: send (message) and receive (message).

Messages sent by a process can be of either fixed or variable size. If only fixed-sized messages can be sent, the physical implementation is straight forward. This restriction, however, makes the task of programming more difficult. On the other hand, variable-sized messages require a more complex physical implementation, but the programming task becomes simpler.

If processes P and Q want to communicate, they must send messages to and receive messages from each other; a communication link must exist between them. This link can be implemented in a variety of ways. We are concerned here not with the link's physical implementation (such as shared memory, hardware bus, or network), but rather with the issues of its logical implementation, such as its logical properties. Some basic implementation questions are as follows:

- How are links established ?
- Can a link be associated with more than two processes ?
- How many links can there be between every pair of processes ?
- What is the capacity of a link ? That is, does the link have some buffer space ? If it does, how much ?
- What is the size of messages ? Can the link accommodate variable-sized or only fixed-sized messages ?
- Is a link unidirectional or bi-directional ? That is, if a link exists between P and Q, can messages flow in only one direction (such as only from P to Q) or in both directions ?

The definition of unidirectional must be stated more carefully, since a link may be associated with more than two processes. Thus, we say that a link is unidirectional only if each process connected to the link can either send or

receive, but not both, and each link has at least one receiver process connected to it.

In addition, there are several methods for logically implementing a link and the send/receive operations:

- Direct or indirect communication
- Symmetric or asymmetric communication
- Automatic or explicit buffering
- Send by copy or send by reference
- Fixed-sized or variable-sized messages

For the remainder of this section, we elaborate on these types of message systems.

4.2.2 Naming

Processes that want to communicate must have a way to refer to each other. They can use either direct communication or indirect communication, as we shall discuss in the next two subsections.

4.2.2.1 Direct Communication

In the direct-communication discipline, each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, the send and receive primitives are defined as follows:

Send (P, message). Send a message to process P.

Receive (Q, message). Receive a message from process Q.

A communication link in this scheme has the following properties:

- A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.
- A link is associated with exactly two processes.
- Between each pair of processes, there exists exactly one link.
- The link may be unidirectional, but is usually bi-directional.

To illustrate, let us present a solution to the producer-consumer problem. To allow the producer and consumer processes to run concurrently, we allow the producer to produce one item while the consumer is consuming another item. When the producer finishes generating an item, it sends that item to the consumer. The consumer gets that item via the receive operation. If an item has not been produced yet, the consumer process must wait until an item is produced. The producer process is defined as:

```
repeat
.....
        produce an item in nextp
        .....
        send ( consumer, nextp );
until false;
```

The consumer process is defined as

```
repeat
        receive ( producer ,nextc) ;
        ......
        consume the item in nextc
        .....
until  false;
```

This scheme exhibits a symmetry in addressing; that is, both the sender and the receiver processes have to name each other to communicate. A variant of this scheme employs asymmetry in addressing. Only the sender names the recipient; the recipient is not required to name the sender. In this scheme, the send and receive primitives are defined as follows:

- **send** (p, message). Send a message to process P.
- **receive** (id, message). Receive a message from any process; the variable id is set to the name of the process with which communication has taken place.

The disadvantage in both of these schemes (symmetric and asymmetric) is the limited modularity of the resulting process definitions. Changing the name of a process may necessitate examining all other process definitions. All references to the old name must be found, so that they can be modified to the new name. This situation is not desirable from the viewpoint of separate compilation.

4.2.2.2 Indirect Communication

With indirect communication, the messages are sent to and received from mailboxes (also referred to as ports). A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed. Each mailbox has a unique identification. In this scheme, a process can communicate with some other process via a number of different mailboxes. Two processes can communicate only if the processes have a shared mailbox. The send and receive primitives are defined as follows:

send (A, message). Send a message to mailbox A.

receive (A, message). Receive a message from mailbox A.

In this scheme, a communication link has the following properties:

- A link is established between a pair of processes only if they have a shared mailbox.
- A link may be associated with more than two processes.
- Between each pair of communicating processes, there may be a number of different links, each link corresponding to one mailbox.
- A link may be either unidirectional or bi-directional.

Now suppose that processes P_1, P_2, and P_3 all share mailbox A. Process P_1 sends a message to A, while P_2 and P_3 each execute a receive from A. Which process will receive the message sent by P_1 ? This question can be resolved in a variety of ways:

- Allow a link to be associated with at most two processes.
- Allow at most one process at a time to execute a receive operation.

- Allow the system to select arbitrarily which process will receive the message (that is, either P_2 or P_3, but not both, will receive the message). The system may identify the receiver to the sender.

A mailbox may be owned either by a process or by the system. If the mailbox is owned by a process (that is, the mailbox is attached to or defined as part of the process), then we distinguish between the owner (who can only receive messages through this mailbox) and the user of the mailbox (who can only send messages to the mailbox). Since each mailbox has a unique owner, there can be no confusion about who should receive a message sent to this mailbox. When a process that owns a mailbox terminates, the mailbox disappears. Any process that subsequently sends a message to this mailbox must be notified that the mailbox no longer exists (via exception handling, described in Section 4.6.4).

There are various ways to designate the owner and users of a particular mailbox. One possibility is to allow a process to declare variables of type mailbox. The process that declares a mailbox is that mailbox's owner. Any other process that knows the name of this mailbox can use this mailbox.

On the other hand, a mailbox that is owned by the operating system has an existence of its own. It is independent, and is not attached to any particular process. The operating system provides a mechanism that allows a process:

- To create a new mailbox
- To send and receive messages through the mailbox
- To destroy a mailbox

The process that creates a new mailbox is that mailbox's owner by default. Initially, the owner is the only process that can receive messages through this mailbox. However, the ownership and receive privilege may be passed to other processes through appropriate system calls. Of course, this provision could result in multiple receivers for each mailbox. Processes may also share a mailbox through the process-creation facility. For example, if

process P created mailbox A, and then created a new process Q, P and Q may share mailbox A. Since all processes with access rights to a mailbox may ultimately terminate, after some time a mailbox may no longer be accessible by any process. In this case, the operating system should reclaim whatever space was used for the mailbox. This task may require some form of garbage collection (see Section 10.3.5), in which a separate operation occurs to search for and deallocate memory that is no longer in use.

4.2.3 Buffering

A link has some capacity that determines the number of messages that can reside in it temporarily. This property can be viewed as a queue of messages attached to the link. Basically, there are three ways that such a queue can be implemented:

- **Zero capacity:** The queue has maximum length 0; thus, the link cannot have any messages waiting in it. In this case, the sender must wait until the recipient receives the message. The two processes must be synchronized for a message transfer to take place. This synchronization is called a rendezvous.
- **Bounded capacity:** The queue has finite length n; thus, at most n messages can reside in it. If the queue is not full when a new message is sent, the latter is placed in the queue (either the message is copied or a pointer to the message is kept), and the sender can continue execution without waiting. However, the link has a finite capacity. If the link is full, the sender m_1, must be delayed until space is available in the queue.
- **Unbounded capacity:** The queue has potentially infinite length; thus, any number of messages can wait in it. The sender is never delayed.

The zero-capacity case is sometimes referred to as a message system with no buffering; the other cases provide automatic buffering.

We note that, in the nonzero-capacity cases, a process does not know whether a message has arrived at its destination after the send operation is completed. If this information is crucial for the computation, the sender must communicate explicitly with the receiver to find out whether the latter has received the message. For example, suppose process P sends a message to process Q and can continue its execution only after the message is received. Process P executes the sequence.

send (Q, message);
receive (Q, message);

Process Q executes

receive (P, message);
send (P, "acknowledgment");

Such processes are said to communicate asynchronously.

There are special cases that do not fit directly into any of the categories that we have discussed:

- The process sending a message is never delayed. However, if the receiver has not received the message before the sending process sends another message, the first message is lost. The advantage of this scheme is that large messages do not need to be copied more than once. The main disadvantage is that the programming task becomes more difficult. Processes need to synchronize explicitly, to ensure both that messages are not lost and that the sender and receiver do not manipulate the message buffer simultaneously.
- The process of sending a message is delayed until it receives a reply. This scheme was adopted in the Tooth operating system. In this system messages are of fixed size (eight words). A process P that sends a message is blocked until the receiving process has received the message and has sent back an eight-word reply by the reply (P, message) primitive. The reply message overwrites the original message buffer. The only difference between the send and reply primitives is that

a send causes the sending process to be blocked, whereas the reply allows both the sending process and the receiving process to continue with their executions immediately.

This synchronous communication method can be expanded easily into full-featured remote procedure call (RPC) system. An RPC system is based on the realization that a sub-routine or procedure call in a single-process system acts exactly like a message system in which the sender blocks until it receives a reply. The message is then like a subroutine call, and the return message contains the value of the sub-routine computed. The next logic step, therefore, is for concurrent processes to be able to call each other sub-routines using RPC. RPCs can be used between processes running on separate computers to allow multiple computers to work together in a mutually beneficial way.

4.3 The critical-section problem

A **critical-section** is a part of program that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one process of execution. Consider a system consisting of n processes { p0,p1,……pn-1}. Each process has a segment of code called a critical-section. The important feature of the system is that, when one process is executing in its critical-section, no other process is to be allowed to execute in its critical-section. Thus the execution of critical-sections by the processes is mutually exclusive in time. The critical-section problem is to design a protocol that the processes can use to co-operate. Each process must request permission to enter its critical-section. The section of the code implementing this request is the entry section. The critical-section may be followed by an exit section. The remaining code is the remainder section.

A solution to the critical-section problem must satisfy the following three requirements:

1. Mutual Exclusion: If process pi is executing in its critical-section, then no other processes can be executing in their critical-sections.

2. Progress: If no process is executing in its critical-section and there exist some processes that are not executing in their remainder section can participate in the decision of which will enter in its critical-section next, and this selection cannot be postponed indefinitely.

3. Bounded Waiting: There exist a bound on the number of times that other processes are allowed to enter their critical-sections after a process has made a request to enter its critical-section and before that request is granted.

In section 4.3.1 and 4.3.2 we discuss a solution to the critical-section problem that satisfy these requirements. When presenting an algorithm, we define only the variables used for synchronization purposes, and describe only a typical process pi whose general structure is shown in figure 4.1. The entry section and exit section are enclosed in boxes to highlight these important segments of code.

Repeat

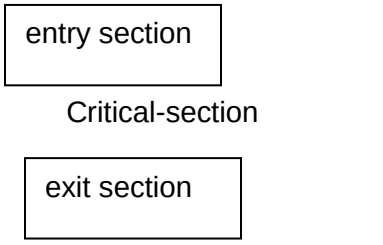


Figure 4.1: General structure of a typical process pi

4.3.1 Two-process solutions

The following two algorithms are applicable to only two processes at a time. The processes are numbered p0 and p1. For convenience, when presenting pi, we use pj to denote the other process; that is j=1-i.

4.3.1.1 Algorithm 1

Let the processes share a common variable turn initialized to 0 (or 1). If turn =i, then process pi is allowed to execute in its critical-section. The structure of process pi is shown in figure 4.2.

This solution ensures that only one process at a time can be in its critical-section. But it does not satisfy the progress requirement, since it requires strict alternation of processes in the execution of the critical-section. For example, if turn=0 and p1 is ready to enter its critical-section, p1 cannot do so, even though p0 may be in its remainder section.

Repeat

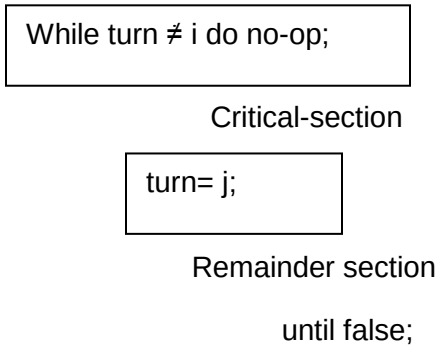


Figure 4.2: The structure of process pi in algorithm 1

4.3.1.2 Algorithm 2

The problem with algorithm 1 is that it does not retain sufficient information about the state of each process; it remembers only which process is allowed to enter that process critical-section. To remedy this problem, we can replace the variable turn with the following array:

var flag: array[0..1] of Boolean;

the elements of the array are initialized to false. If flag[i] is true, this value indicates that pi is ready to enter the critical-section. The structure of process pi is shown in figure 4.3.

In this algorithm process pi first sets flag[i] to be true, signaling that it is ready to enter its critical-section. Then pi checks to verify that process pj is not also ready to enter its critical-section. If pj were ready, then pi would wait until pj had indicated that it no longer needed to be in the critical-section. At this point, pi would enter the critical-section. On exiting the critical section, pi would set its flag to be false, allowing the other process to enter its critical-section.

In this solution, the mutual exclusion requirement is satisfied. Unfortunately, the progress requirement is not met.

Repeat

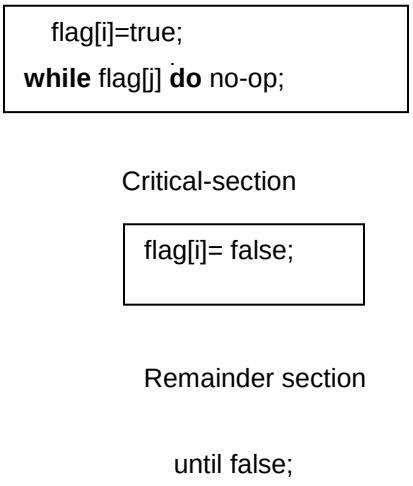


Figure 4.3: The structure of process pi in algorithm 2

4.3.1.3 Algorithm 3

By combining the key ideas of algorithm 1 and algorithm 2, we obtain a correct solution to the critical-section problem, where all three requirements are met. The processes share two variables;

```
var flag: array[0…1] of boolean;
    turn; 0…1;
```

initially flag[0]=flag[1]=false, and the value of turn is immaterial (but it is either 0 or 1). The structure of process pi is shown in figure 4.4. the eventual

value of turn decides which of the two processes is allowed to enter its critical-section first.

Repeat

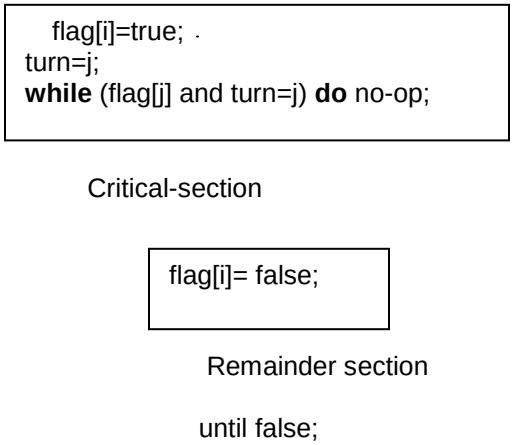


Figure 4.4: The structure of process pi in algorithm 3

This algorithm satisfies the following three properties.

1. Mutual exclusion.
2. The progress requirement
3. The bounded- waiting requirement

To prove property 1, we observe that each pi enters its critical-section only if flag[j]=false or turn=i. Also note that if both processes are executing in their critical-sections at the same time, then flag[0]=flag[1]=true and value of turn can be either 0 or 1, but not both, hence only one of the two can be executing in their critical-section.

To prove property 2 and 3, we note that a process pi can be prevented from entering the critical-section only if it stuck in the while loop with the condition flag[j]=true and turn=j; this loop is the only one. If pj is not ready to enter the critical-section, then flag[j]=false and pi can enter its critical-section. If pj has set flag[j]=true and is also executing in its while statement, then either turn=I or turn=j. if turn=i, then pi will enter the critical-section. If turn=j, then pj will

enter the critical-section. Since pi does not change the value of the variable turn while executing the while statement, pi will enter the critical-section(progress) after at the most one entry by pj (bounded waiting).

4.3.2 Multiple-Process Solutions

An algorithm developed for solving the critical-section problem for n processes is also called Bakery algorithm, because it is based on the scheduling algorithm commonly used in baheries, ice-cream stores, meat markets, motor-vehicle registries and other locations where order must be made out of chaos.

In this algorithm each process, receives a number. Unfortunately, the bakery algorithm cannot guarantee that two processes do not receive the same number. In such a case (if two processes receives same number) the process with the lowest name is served first. Since the process name is unique and totally ordered, the algorithm is completely deterministic.

The common data structures used are

```
Var choosing : array [0..n-1] of Boolean;
    number: array [0..n-1] of integer;
```

initially these data structures are initialized to false and 0 respectively . For convenience, we define the following notation. The structure of process pi, used in the bakery algorithm , is shown in figure 4.5.

Consider a process pi in its critical section and pk trying to enter the pk critical section. When process pk executes the second while statement for j=I, it finds that

- number[i] ≠ 0
- (number[i],i) < (number[k],k)

Thus it continues looping in the while statement until pi leaves the pi critical section.

```
                    repeat
choosing[i]=true;
number[i]=max(number[0],number[1],….number[n-1])+1;
choosing[i]=false;
for j=0 to n-1
do begin
  while choosing[j] do no-op;
  while number[j]   ≠ 0
and (number[j],j) <(number[i],i) do no-op;
end;

          critical section

  number[i]=0;

          remainder section
          until false;
```

Figure 4.5: The structure of process pi in the bakery algorithm.

4.4 Semaphores

Semaphores are the classic method for restricting access to shared resources (e.g. storage) in a multi-processing environment. They were invented by Dijkstra and first used in T.H.E operating system.

A semaphore is a protected variable (or abstract data type) which can only be accessed using the following operations:

```
P(s)
        Semaphore s;
        {
          while (s == 0) ;/* wait until s>0 */
          s = s-1;
        }

V(s)
        Semaphore s;
        {
          s = s+1;
```

```
        Init(s, v)
        Semaphore s;
        Int v;
        {
          s = v;
            }
```

The P operation busy-waits (or maybe sleeps) until a resource is available whereupon it immediately claims one. V is the inverse, it simply makes a resource available again after the process has finished using it. Init is only used to initialize the semaphore before any requests are made. The P and V operations must be indivisible, i.e. no other process can access the semaphore during their execution.

To avoid busy-waiting, a semaphore may have an associated queue of processes (usually a FIFO). If a process does a P on a semaphore which is zero, the process is added to the semaphore's queue. When another process increments the semaphore by doing a V and there are tasks on the queue, one is taken off and resumed.

Semaphores are not provided by hardware. But they have several attractive properties:

- Machine independent.
- Simple.
- Powerful. Embody both exclusion and waiting.
- Correctness is easy to determine.
- Work with many processes.
- Can have many different critical sections with different semaphores.
- Can acquire many resources simultaneously (multiple P's).
- Can permit multiple processes into the critical section at once, if that is desirable.

4.5 Monitors

- Another high-level synchronization construct is the monitor type. A monitor is a collection of procedures, variables and data structures grouped together. Processes can call the monitor procedures but cannot access the internal data structures. Only one process at a time may be be **active** in a monitor.

- The syntax of monitor look like :

  Type monitor-name= monitor
  Variable declarations
  procedure entry P1 (….);
  begin…. end;
  .
  .
  .
  procedure entry P2 (….);
  begin…. end;
  .
  procedure entry Pn (….);
  begin…. end;
  begin
  initialization code
  end

  A procedure defined within a monitor can access only those variables declared locally within the monitor and the formal parameters. Similarly, the local variables of a monitor can be accessed by only the local procedures.

4.6 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.

4.7 Summary

This unit has brought out the information about inter-process communication and process synchronization. Given a collection of co-operating processes that share data, mutual exclusion should be provided.

Mutual exclusion is a way of making sure that if one process is using a shared modifiable data, the other processes will be excluded from doing the same thing. One solution for this is by ensuring that a critical section of code is in use by only one process or thread at a time. There are different algorithms to solve the critical-section problem. But the main disadvantage of these algorithms is that they all need busy waiting. Busy waiting is a technique in which a process repeatedly checks to see if a condition is true. It will delay execution for some amount of time. To overcome this problem we use semaphores, which is a mechanism that prevents two or more processes from accessing a shared resource simultaneously. Another way

to achieve mutual exclusion is to use monitors which are characterized by a set of programmer defined operators.

Self Assessment Questions

1. _____ is one that can affect or be affected by the other processes executing in the system.
2. _____ is a mechanism that allow processes to communicate and to synchronize their actions.
3. To processes can communicate only if the processes have a _____.
4. _____ are the classic method for restricting access to shared resources in a multi-processing environment.
5. _____ is a collection of procedures, variables and data structures grouped together.

4.8 Terminal Questions

1. Discuss Interprocess Communication.
2. Define the following terms.
   i) Mutual exclusion ii) Busy waiting. iii) Critical section.
3. Explain the different requirements for the solution of critical-section problem.
4. Explain in detail the Algorithm 3, a solution to critical-section problem.
5. What are semaphores? Explain.
6. Discuss Monitors.

4.9 Answers to Self Assessment Questions and Terminal Questions

Answers to Self Assessment Questions

1. Co-operating process
2. Inter-Process Communication
3. Shared mailbox
4. Semaphores
5. Monitors

Answers to Terminal Questions

1. Refer section 4.2
2. Refer section 4.3
3. Refer section 4.3
4. Refer section 4.3.1.3
5. Refer section 4.4
6. Refer section 4.5

Unit 5 Introduction to Deadlocks

Structure

5.1 Introduction

Several processes compete for a finite set of resources in a multi-programmed 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. **Deadlock** occurs when we have a set of processes [not necessarily all the processes in the system], each holding some resources, each requesting some resources, and none of them is able to obtain what it needs, i.e. to make progress. We will usually reason in terms of resources R1, R2, .., Rm and processes P1, P2, .., Pn. A process Pi that is waiting for some currently unavailable resource is said to be **blocked**.

Resources can be **preemptable** or **non-preemptable**. A resource is preemptable if it can be taken away from the process that is holding it [we can think that the original holder waits, frozen, until the resource is returned to it]. Memory is an example of a preemptable resource. Of course, one may choose to deal with intrinsically preemptable resources as if they were non-preemptable. In our discussion we only consider non-preemptable resources.

Resources can be **reusable** or **consumable**. They are reusable if they can be used again after a process is done using them. Memory, printers, tape drives are examples of reusable resources. Consumable resources are resources that can be used only once. For example a message or a signal or an event. If two processes are waiting for a message and one receives it, then the other process remains waiting. To reason about deadlocks when dealing with consumable resources is extremely difficult. Thus we will restrict our discussion to reusable resources.

Objectives:

At the end of this unit, you will be able to understand:

System model for Deadlocks, Deadlock Characterization, Deadlock Handling, Deadlock prevention , how to avoid Deadlocks using Banker's Algorithm and Deadlock Detection.

5.2 System Model

The number of resources in a system is always finite. But the number of competing processes are 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 whereas 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.

5.3 Deadlock Characterization

5.3.1 Necessary Conditions for Deadlock

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.

5.3.2 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).

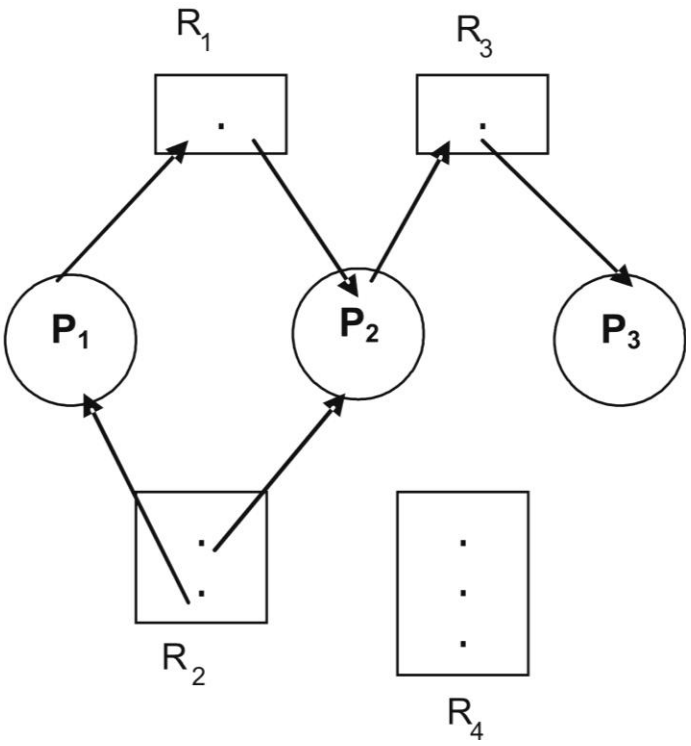


Figure 5.1: Resource allocation graph

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.

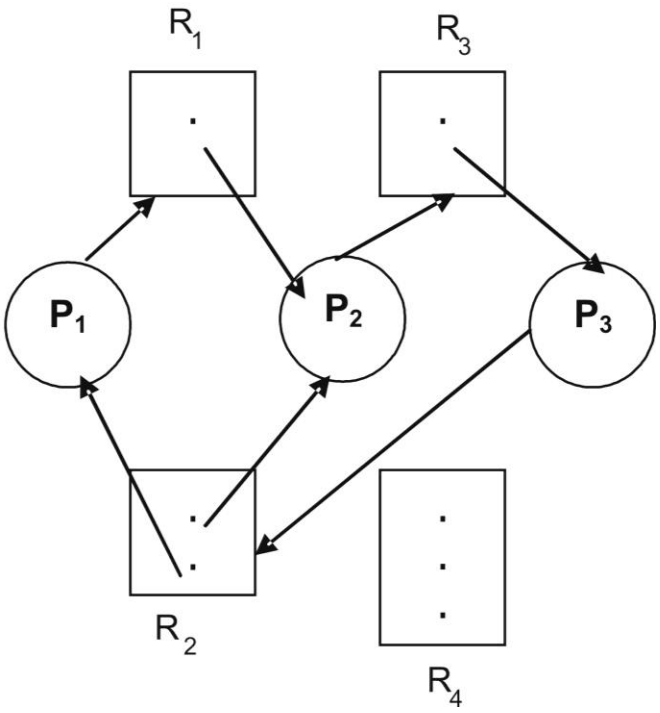


Figure 5.2: Resource allocation graph with deadlock

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.

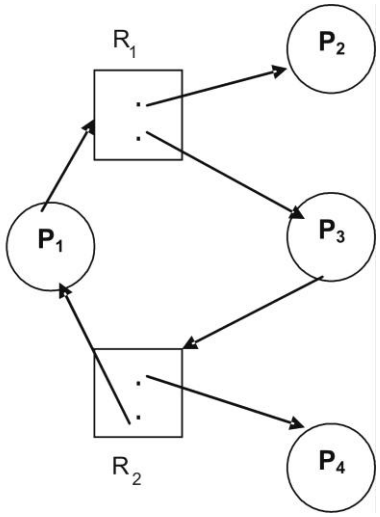


Figure 5.3: Resource allocation graph with a cycle but no deadlock

5.4 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.

5.5 Deadlock Prevention

The four necessary conditions for deadlocks to occur are mutual exclusion, hold 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 simultaneous 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 therefore 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 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 an 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 an increasing order of these numbers. A process, to start with in the very first, instance can request for any resource say R_i. Thereafter 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.

5.6 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.

5.6.1 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 in Figure 5.4.

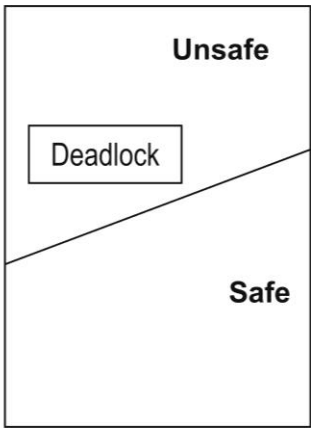


Figure 5.4: Safe, unsafe and deadlock state spaces

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.

5.6.2 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).

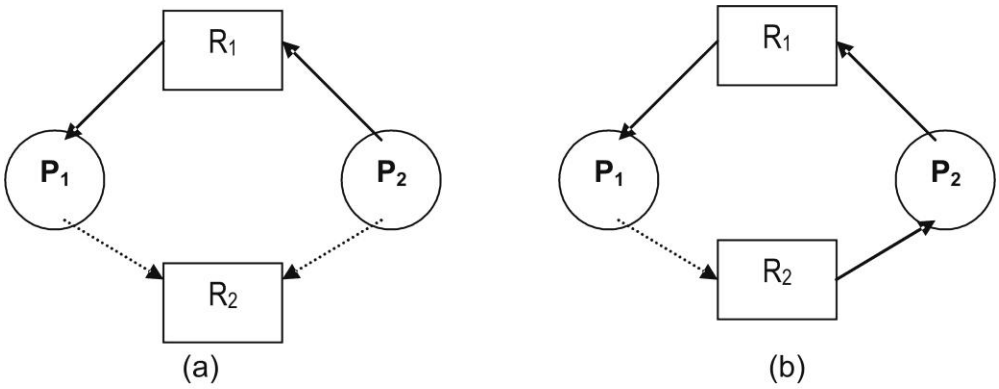


Figure 5.5: Resource allocation graph showing safe and deadlock states

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

5.6.3 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].

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

5.6.3.2 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
Po	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
P0	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
P0	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	< >

5.7 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

5.7.1 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 in Figure 5.6.

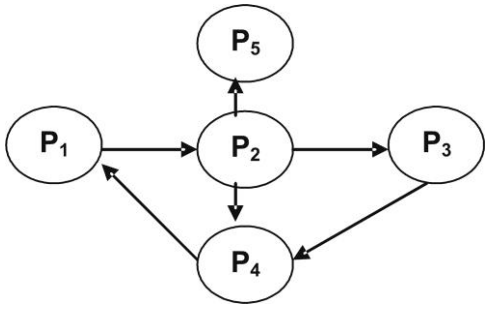


Figure 5.6: Wait-for graph

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.

5.7.2 Multiple Instance of a Resource

A wait-for graph is not applicable for detecting deadlocks where multiple instances of resources exist. 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.

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**
P0	**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.

5.7.3 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.

5.8 Summary

The concept of deadlock has highlighted the problem associated with multi-programming. 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.

Self Assessment Questions

1. _____ can be described by a resource allocation graph.
2. 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 _____.
3. When resources have multiple instances ---------- is used for deadlock Avoidance.
4. A wait-for graph is not applicable for detecting deadlocks where there exist _____.
5. _____ algorithm requires each process to make in advance the maximum number of resources of each type that it may need.

5.9 Terminal Questions

1. Explain how it is possible to prevent a deadlock in a system.
2. Write a note on Resource Allocation Graph.
3. Describe safe, unsafe and deadlock state of a system.
4. What is the use of Resource allocation graph Algorithm?
5. Explain how Banker's algorithm is used to check safe state of a system.
6. Explain the different methods used to detect a deadlock.
7. Compare the relative merits and demerits of using prevention, avoidance and detection as strategies for dealing with deadlocks.

5.10 Answers to Self Assessment Questions and Terminal Questions

Answers to Self Assessment Questions

1. Deadlocks
2. Deadlock
3. Banker's algorithm

4. Multiple instances of resources
5. Deadlock Avoidance.

Answers to Terminal Questions

1. Refer section 5.5
2. Refer section 5.3.2
3. Refer section 5.6
4. Refer section 5.6.2
5. Refer section 5.6.3
6. Refer section 5.7
7. Refer section 5.5 5.6,5.7

Unit 6 Memory Management

Structure

6.1 Introduction

The concept of CPU scheduling allows a set of processes to share the CPU thereby 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.

Objectives:

At the end of this unit, you will be able to understand:

Logical Address Space, Physical Address Space, Swapping, allocation methodologies, paging and page table implementation and about segmentation concepts.

6.2 Logical versus 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 shown in Figure 6.1.

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.

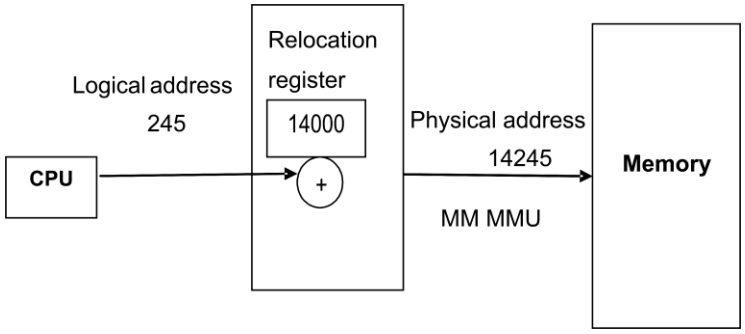


Figure 6.1: Dynamic Relocation

6.3 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.

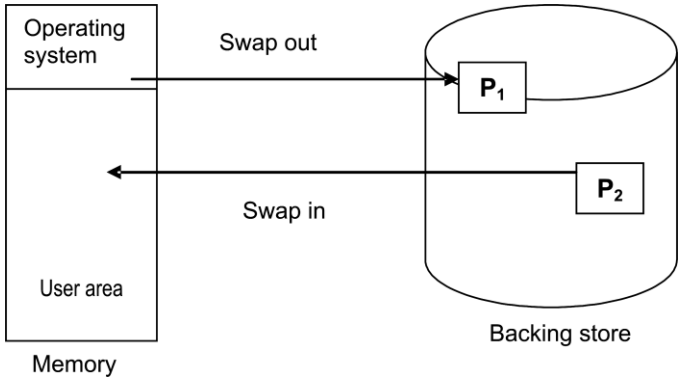


Figure 6.2: Swapping of processes

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 be 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.

6.4 Contiguous Allocation

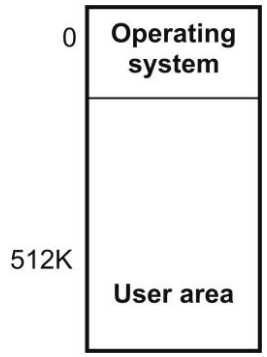


Figure 6.3: Memory partition

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).

6.4.1 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 thereby 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.

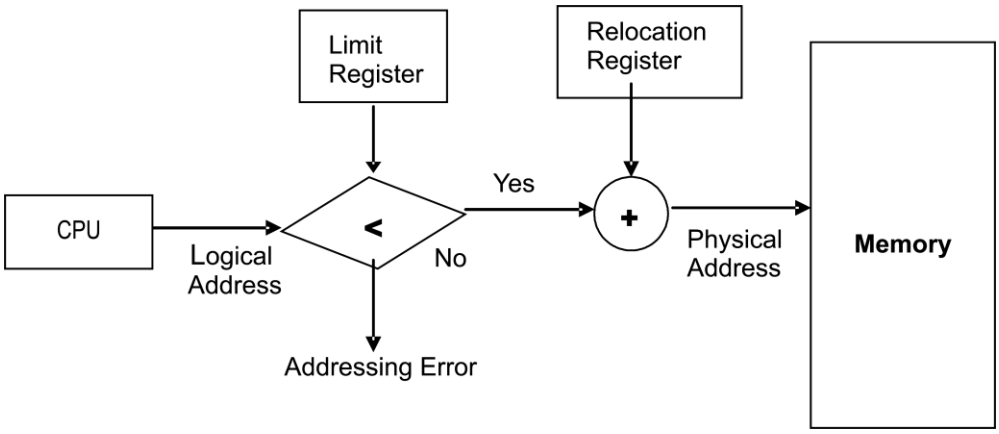


Figure 6.4: Hardware for relocation and limit register

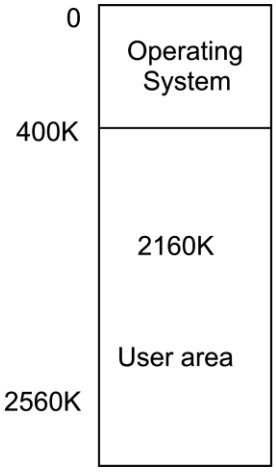
6.4.2 Multiple Partition Allocation

Multi-programming 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 multi-programming 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 in 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.

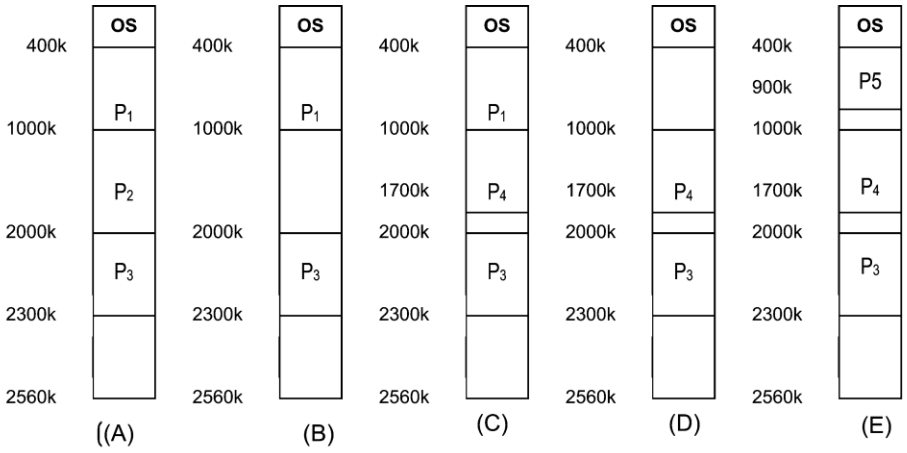


Figure 6.6 : Memory allocation and job scheduling

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 thereafter 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 among the three and is seldom used.

6.4.3 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).

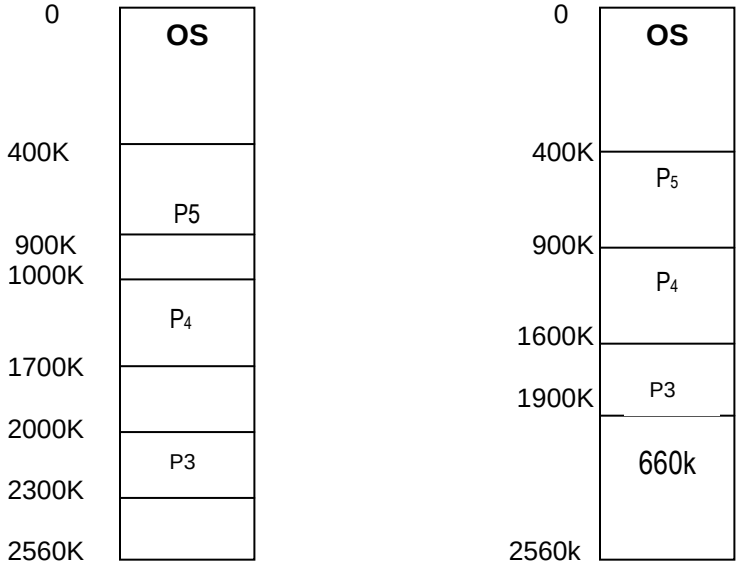


Figure 6.7: Compaction

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 whereas 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.

6.5 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. Another 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.

6.5.1 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).

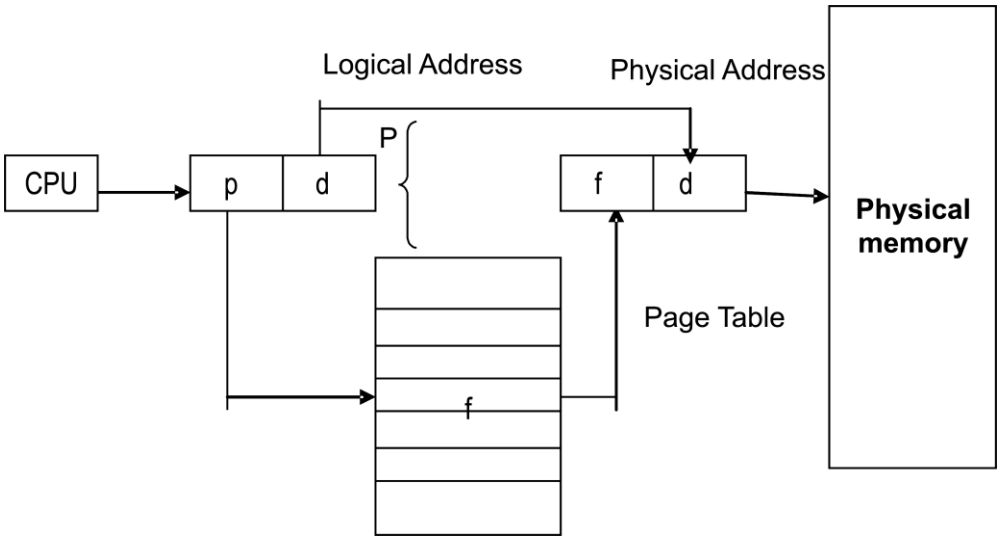


Figure 6.8: Paging Hardware

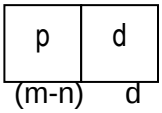
A logical address generated by the CPU consists of two parts: Page number (p) and a page offset (d). The page number is used as an index into a page table. The page table contains the base address of each frame in physical memory. The base address is combined with the page offset to generate the physical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

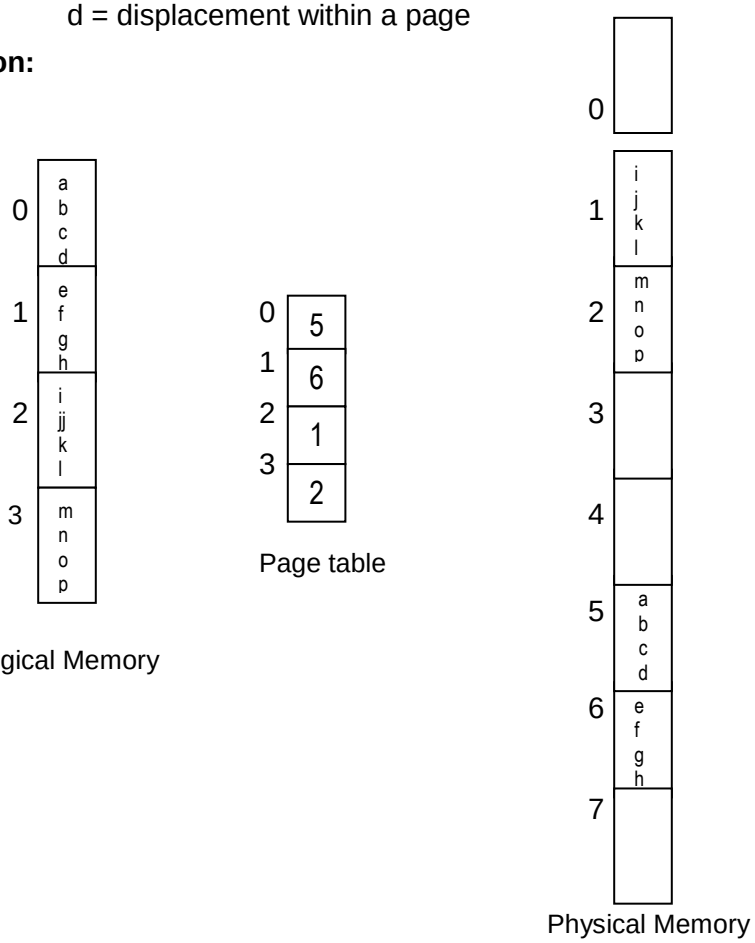
Logical address:



Where p = index into the page table

d = displacement within a page

Illustration:



Page 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 trade-off between the above factors is used.

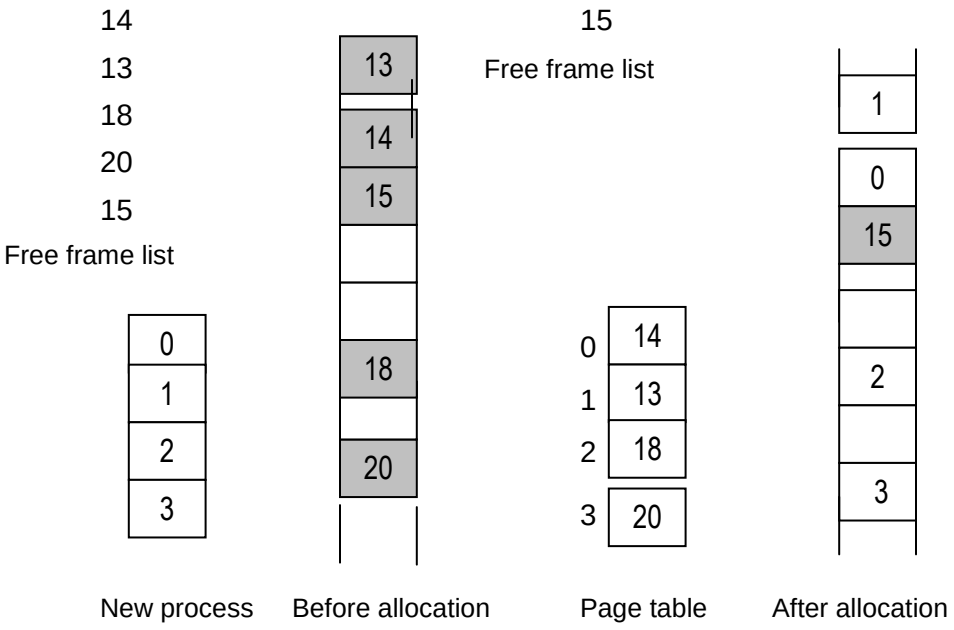


Figure 6.9: Frame Allocation

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.

6.5.2 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).

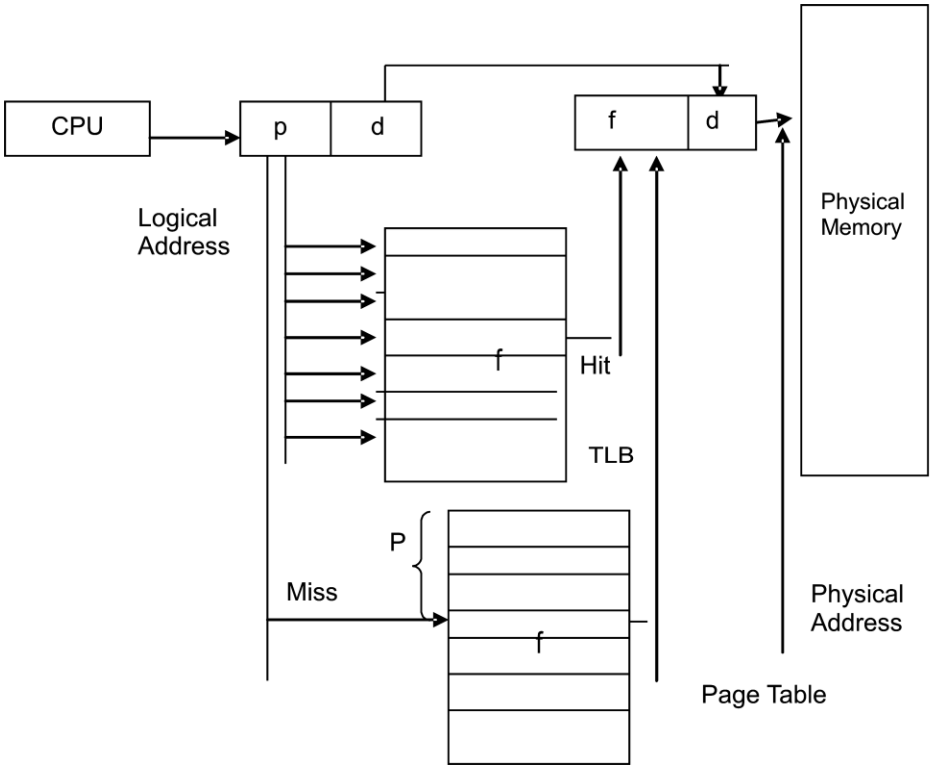


Figure 6.10: Paging hardware with TLB

6.5 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 6.11).

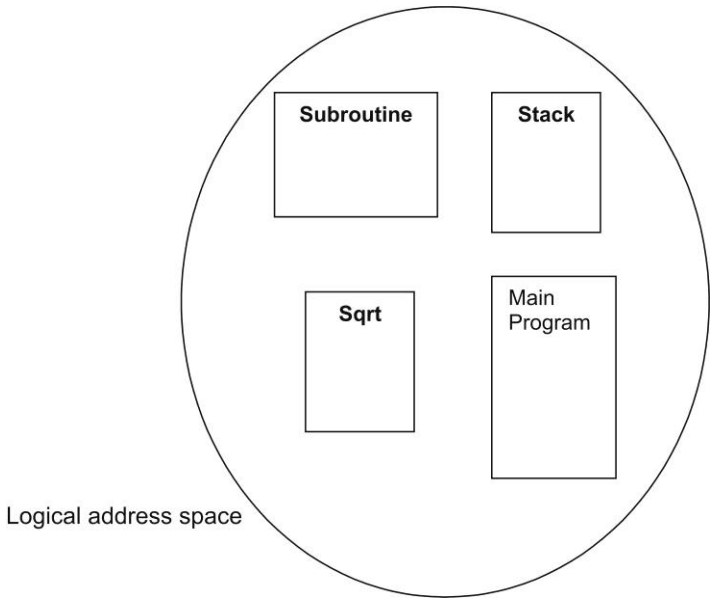


Figure 6.11: User's view of memory

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.

6.6.1 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.

6.6.2 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 whereas the limit is the length of the segment (Figure 6.12).

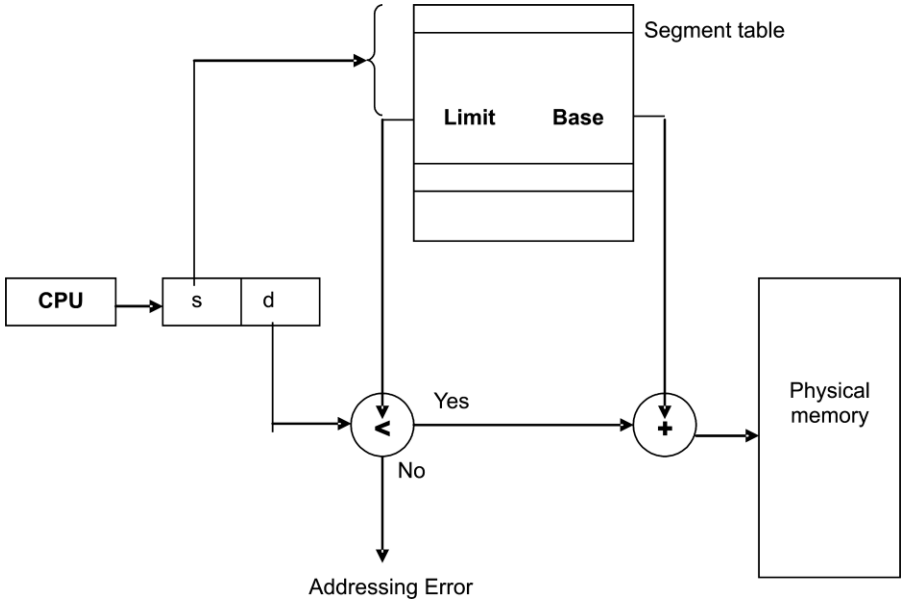


Figure 6.12: Segmentation Hardware

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 6.13).

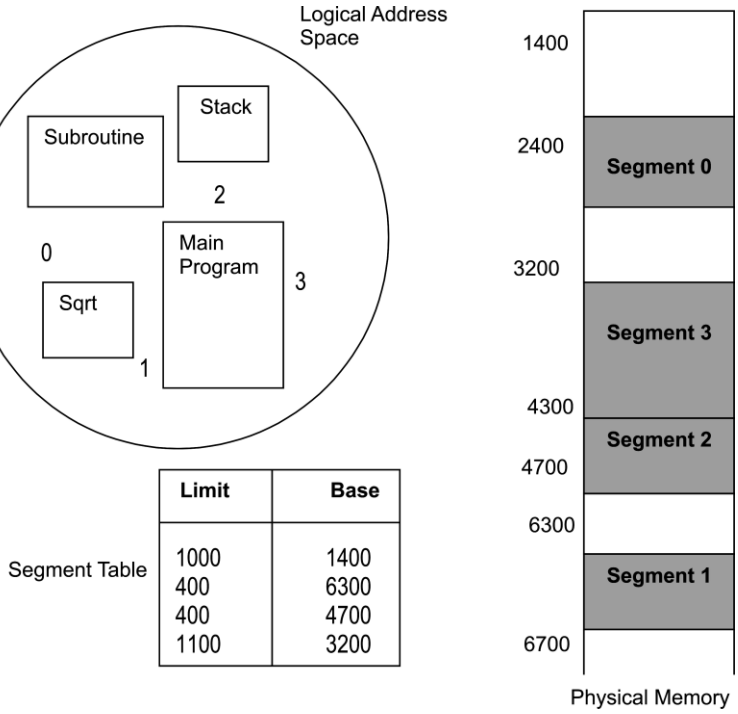


Figure 6.13: Illustration of Segmentation

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

6.6.3 External 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.

6.7 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.

Self Assessment Questions

1. An address generated by the CPU is referred to as _____.
2. At the run time /Execution time, the virtual addresses are mapped to physical addresses by _____.
3. Compaction is possible only if relocation is done at _____.
4. Physical memory is divided into fixed sized blocks called _____.
5. TLBs stands for _____.

6.8 Terminal Questions

1. What is MMU?
2. Explain how virtual address are mapped on to physical address.
3. Explain single partition allocation.
4. Explain first-fit, Best-fit and worst-fit allocation algorithms with an example.
5. Explain the paging concept with the help of a diagram.
6. Why are TLBs required in paging? Explain.
7. With a block diagram explain the hardware required for segmentation.
8. Write a note on External Fragmentation.

6.9 Answers to Self Assessment Questions and Terminal Questions

Answers to Self Assessment Questions

1. Logical address
2. Memory Management Unit
3. Runtime
4. Frames
5. Translation look-aside buffers

Answers to Terminal Questions

1. Refer section 6.2
2. Refer section 6.2
3. Refer section 6.4.1
4. Refer section 6.4.2
5. Refer section 6.5.1
6. Refer section 6.5.2
7. Refer section 6.6.3

Unit 7 Virtual Memory

Structure

7.1 Introduction

Memory management strategies like paging and segmentation helps to implement the concept of multi-programming. 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. In addition, virtual memory allows mapping of a

large virtual address space onto a smaller physical memory. It also raises the degree of multi-programming and increases CPU utilization. Because of the above features, users are freed from worrying about memory requirements and availability.

Objectives:

At the end of this unit, you will be able to understand:

Virtual Memory technique, its need, Demand Paging, different Page replacement Algorithms, Thrashing and its causes.

7.2 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 multi-programming 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).

  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.

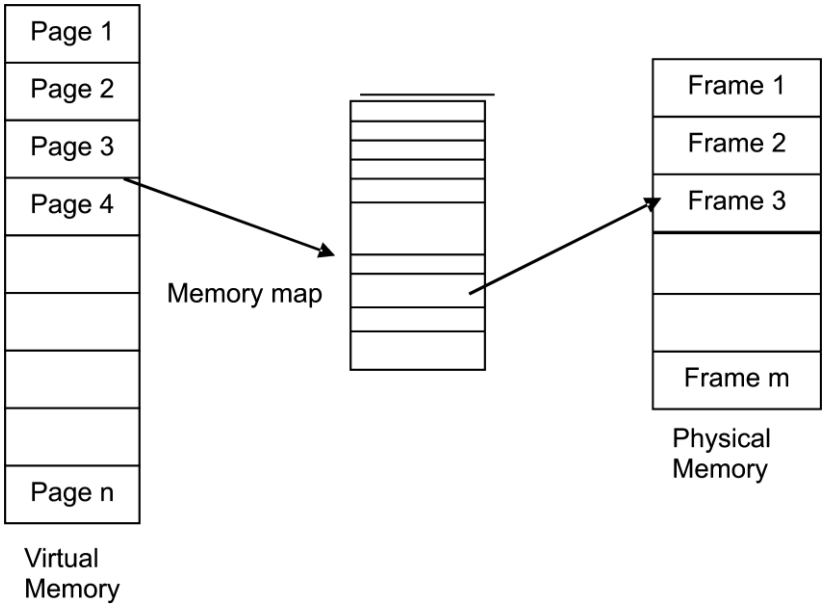


Figure 7.1: Virtual to physical memory mapping (n >> m)

7.3 Demand Paging

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

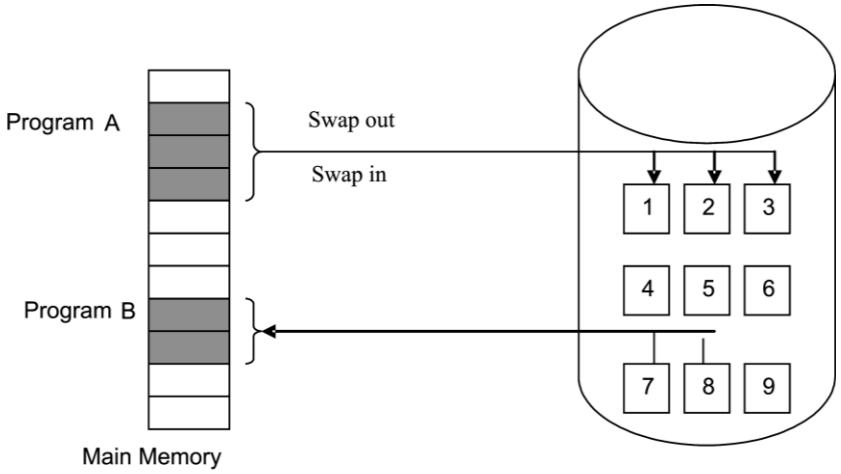


Figure 7.2: Paging with swapping

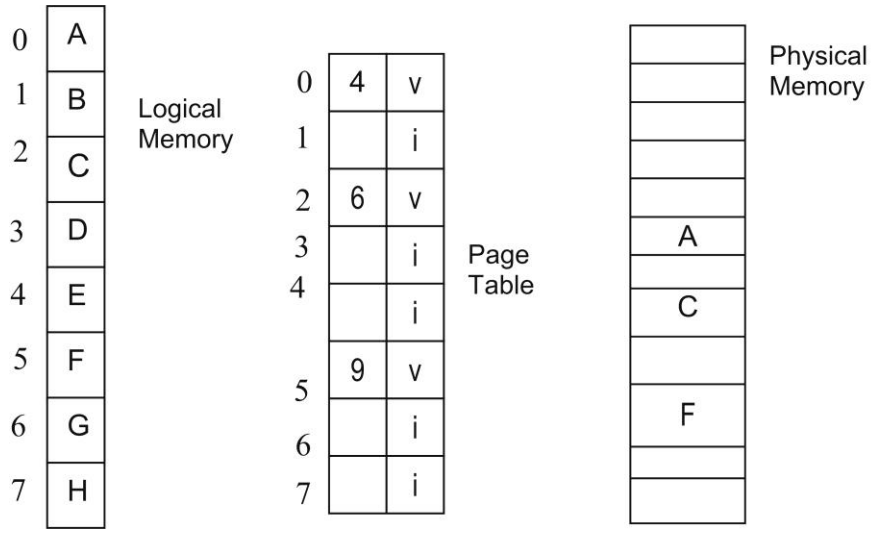


Figure 7.3: Demand paging with protection

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).

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):

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.

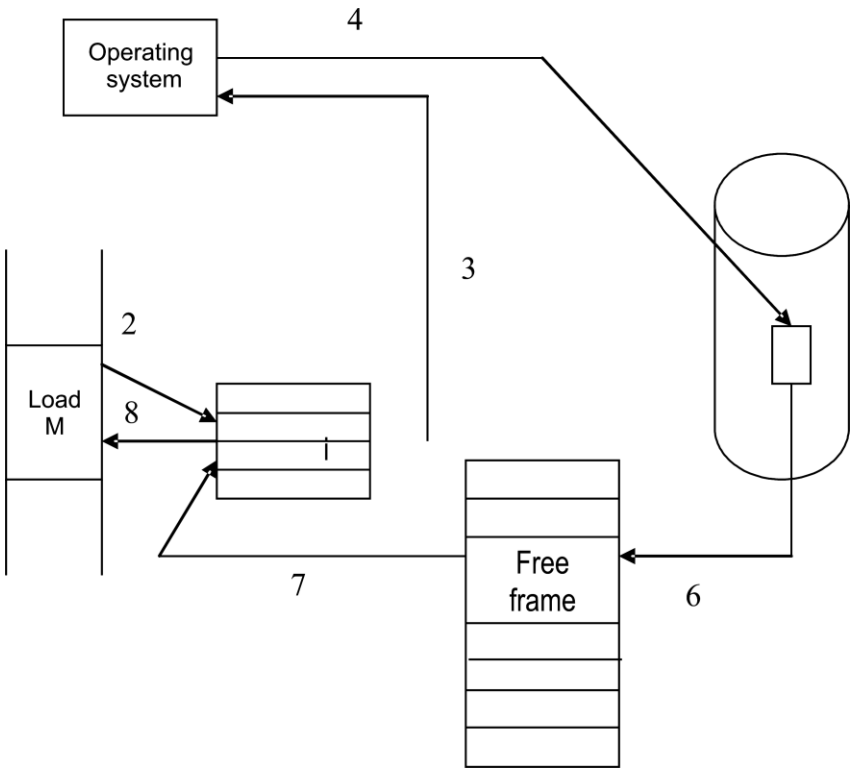


Figure 7.4: Handling a page fault

7.4 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 multi-programming can be increased so that more ready processes can be loaded and executed. Now, we may come across a situation wherein all of sudden, a process hitherto not accessing certain pages starts accessing those pages. The degree of multi-programming 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 multi-programming is lost.
- Swap out a process to free all its frames. This reduces the degree of multi-programming 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. if not, 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. A swap out is not always 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 looks for a victim.
- Frame allocation algorithm: In a multi-programming environment with degree of multi-programming equal to n, the algorithm gives the number of frames to be allocated to a process.

7.5 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.

7.5.1 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.

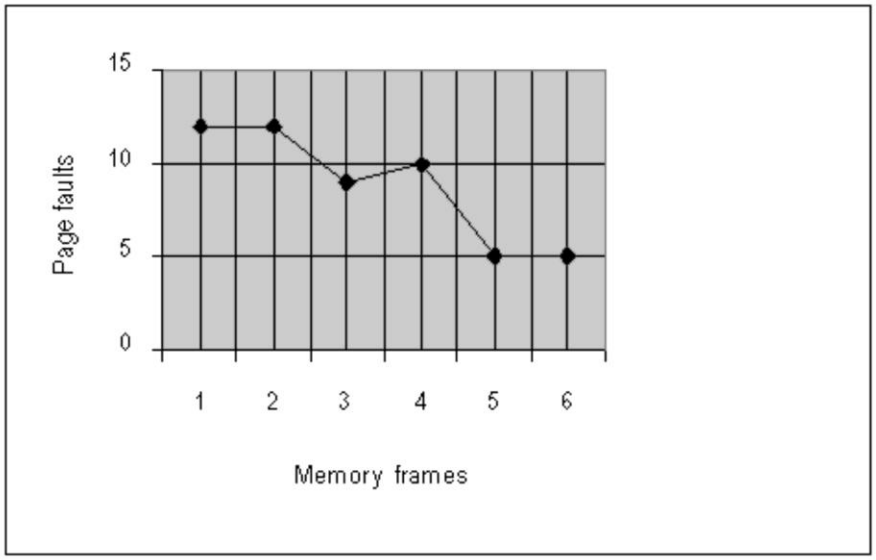
The performance of the FIFO algorithm is not always good. The replaced page may have an initialization module that needs to be executed only once and therefore no longer needed. On the other hand, the page may have a

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.

7.5.2 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.

7.5.3 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) whereas 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.

7.6 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 . In this case a process spends more time in paging than executing. This high paging activity is called thrashing.

7.6.1 Causes 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 throughput drops drastically. This is illustrated in the figure (Figure 7.5):

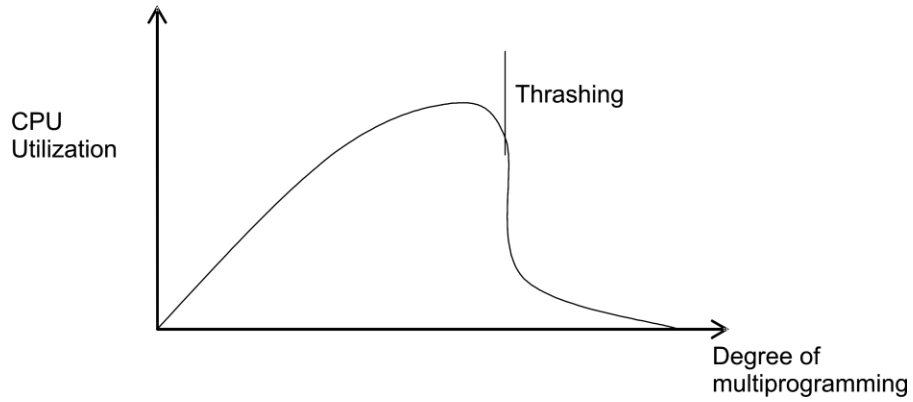


Figure 7.5: Thrashing

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 therein. 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 then thrashing

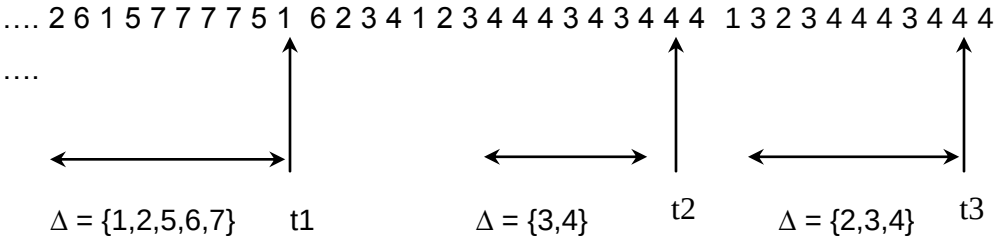
occurs because the process is not able to keep in memory actively used pages.

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:



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 multi-programming can be increased. If at any instant D>m then the operating system swaps out a

process to decrease the degree of multi-programming 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 multi-programming 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 Summary

In this chapter we have studied a technique called virtual memory that creates an illusion for the user that he/she has a large memory at his/her 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.

Self Assessment Questions

1. _____ is a technique that allows execution of processes that may not be entirely in memory.
2. Virtual Memory is implemented using _____ .
3. _____ is basic to demand paging.
4. _____ algorithm produces the lowest page fault rate of all algorithm.
5. _____ raises the degree of multi-programming and increases CPU utilization.

7.8 Terminal Questions

1. What is virtual memory? Distinguish between logical address and physical address.
2. Explain demand paging virtual memory system.
3. Explain Belady's anomaly with the help of FIFO page replacement algorithm.
4. 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.

5. Discuss the Optimal Page replacement algorithm.
6. What is thrashing and what is its cause?
7. Explain different ways of controlling the Thrashing.

7.9 Answers to Self Assessment Questions and Terminal Questions

Answers to Self Assessment Questions

1. Virtual Memory
2. Demand Paging
3. Page Replacement
4. Optimal Page Replacement
5. Virtual Memory

Answers to Terminal Questions

1. Refer Section 7.1 and 7.2
2. Refer Section 7.3
3. Refer Section 7.5.1
4. Refer Section 7.5.1, 7.5.3
5. Refer Section 7.5.2
6. Refer Section 7.6 , 7.6.1
7. Refer Section 7.6.1,7.6.2,7.6.3

Unit 8 File System Interface and Implementation

Structure

8.1 Introduction

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. It resides on the secondary storage because of the two main characteristics of secondary storage, namely, large storage capacity and non-volatile nature.

Objectives:

At the end of this unit, you will be able to understand:

The concepts of Files, Different File access methods. Different directory structures, disk space allocation methods, how to manage free space on the disk and implementation of directory.

8.2 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 whereas 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 pre-defined structure. For example, a file containing text is a collection of characters organized as lines, paragraphs and pages whereas a file containing source code is an organized collection of segments which in turn are organized into declaration and executable statements.

8.2.1 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.

8.2.2 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.

8.2.3 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.

8.2.4 Structure of 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 that 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.

8.3 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

8.3.1 Sequential Access

In a simple access method, information in a file is accessed sequentially one record after another. To process the i^{th} 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.

8.3.2 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.

8.3.3 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.

8.4 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. In 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.

8.4.1 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).

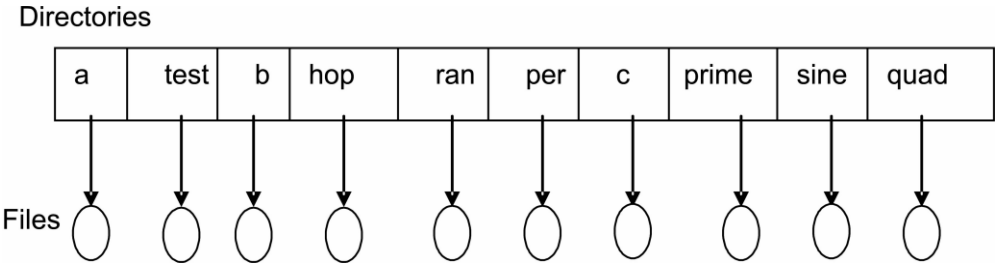


Figure 8.1: Single-level directory structure

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.

8.4.2 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.

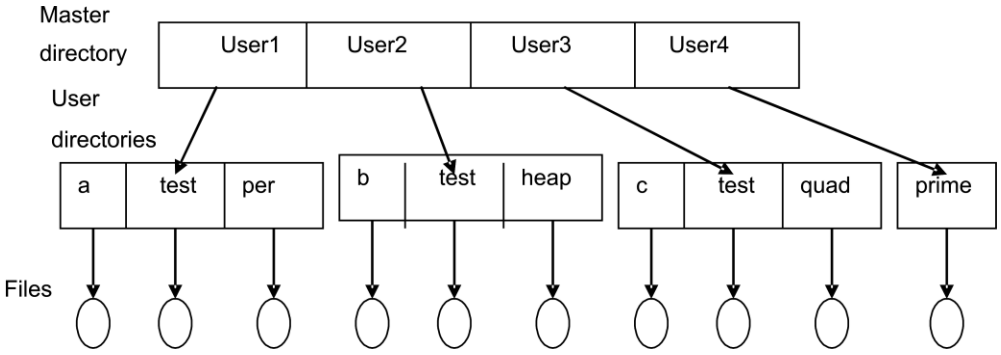


Figure 8.2: Two-level directory structure

8.4.3 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(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.

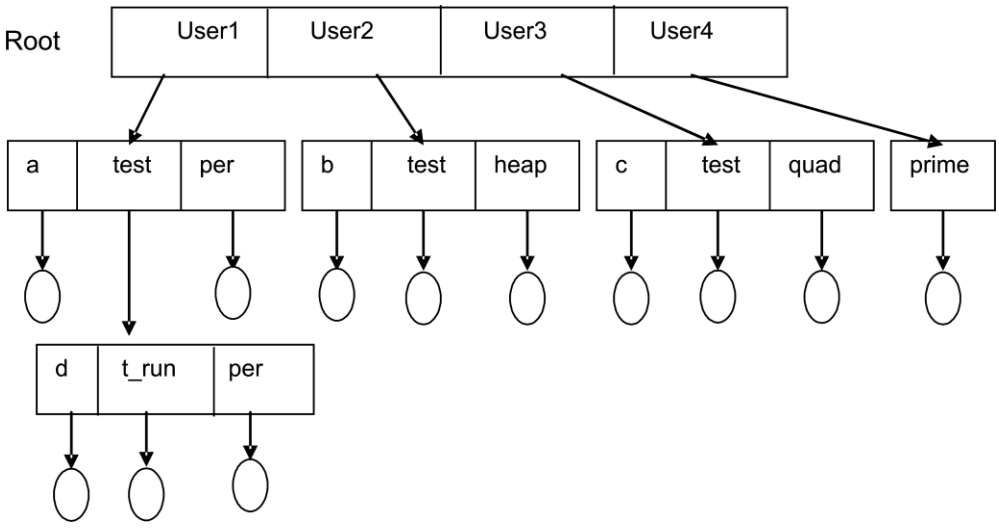


Figure 8.3: Tree-structured directory structure

8.5 Allocation Methods

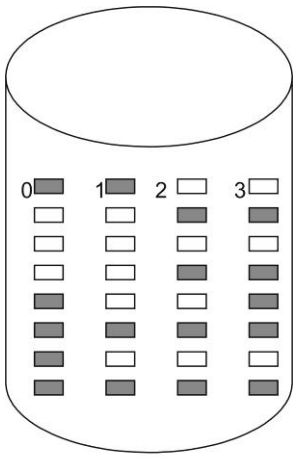
Allocation of disk space to files is a problem that looks at how effectively disk space is utilized and quickly files can be accessed. The three major methods of disk space allocation are:

- Contiguous allocation
- Linked allocation
- Indexed allocation

8.5.1 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 8.4).



Directory:

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

Figure 8.4: 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, whereas 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.

A 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.

Another 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.

8.5.2 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 8.5)

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. moreover a file can grow and shrink without problems of allocation.

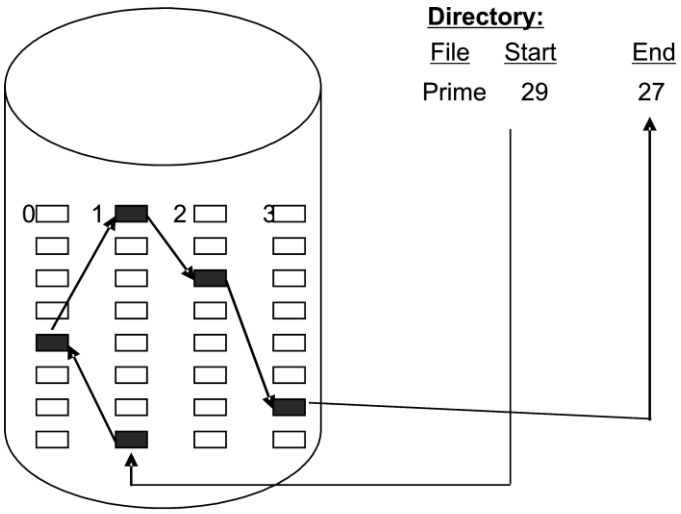


Figure 8.5: Linked allocation

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 8.6). Random access of files is possible because the FAT can be scanned for a direct block address.

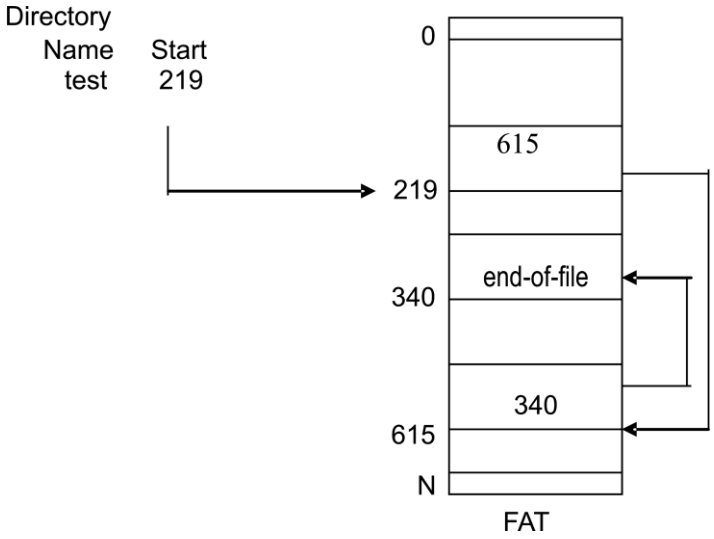


Figure 8.6: File allocation table

8.5.3 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 8.7). Here both sequential and direct access of a file are possible. Also it does not suffer from external fragmentation.

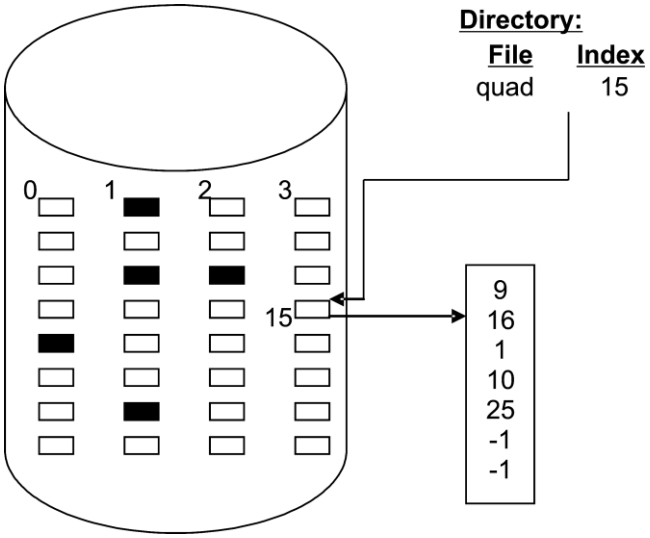


Figure 8.7: Indexed Allocation

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.

8.5.4 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 i^{th} 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.

8.6 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:

8.6.1 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 whereas 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:

001011000110100100101011110001000000……

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.

8.6.2 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.

8.6.3 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.

8.6.4 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.

8.7 Directory Implementation

The two main methods of implementing a directory are:

- **Linear list**
- **Hash table**

8.7.1 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 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.

8.7.2 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.

8.8 Summary

In this chapter the operating system as a secondary resource manager has been 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 methods, File Allocation methods and implementation of a file. We have also learnt the concept of a directory, its various structures for easy and protected access of files and its implementation.

Self Assessment Questions

1. A file is a collection of related information recorded on the _____.
2. _____ is best suited access method where most of the records in a file are to be processed.
3. _____ requires a file to occupy continuous blocks on the disk.
4. In a linked allocation _____ is not possible.
5. Problems of external fragmentation and size declaration present in contiguous allocation are overcome in _____.

8.9 Terminal Questions

1. Explain the concept of File.
2. What is the difference between a File and Directory?
3. Explain Different operations possible on Files.
4. What is the need for a directory? Explain the different directory structures.
5. Explain any two Disk space Allocation Methods.
6. Write a short note on Free space Management.
7. Discuss the different methods of implementing a Directory.

8.10 Answers to Self Assessment Questions and Terminal Questions

Answers to Self Assessment Questions

1. Secondary Storage.
2. Sequential Access
3. Continuous allocation
4. Random access of files.
5. Linked allocation

Answers to Terminal Questions

1. Refer Section 8.2
2. Refer Section 8.2 and 8.4
3. Refer Section 8.2.2
4. Refer Section 8.4
5. Refer Section 8.5
6. Refer Section 8.6
7. Refer Section 8.7

Unit 9 Operating Systems in Distributed Processing

Structure

9.1 Introduction

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. Moreover 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)

Objectives:

At the end of this unit, you will be able to understand:

- Characteristics of Distributed Processing, parallel processing and centralized processing.
- Achitecture of Network Operating System(NOS) and functions of NOS.
- About Global Operating System, Remote Procedure Call and Distributed File Management.

9.2 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.

9.3 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.

9.4 Centralized v/s 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.

9.4.1 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.

9.4.2 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.

9.4.3 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.

9.5 Network Operating System (NOS) Architecture

The architecture of typical NOS is shown below (Figure 9.1). 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.
- 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. This 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.

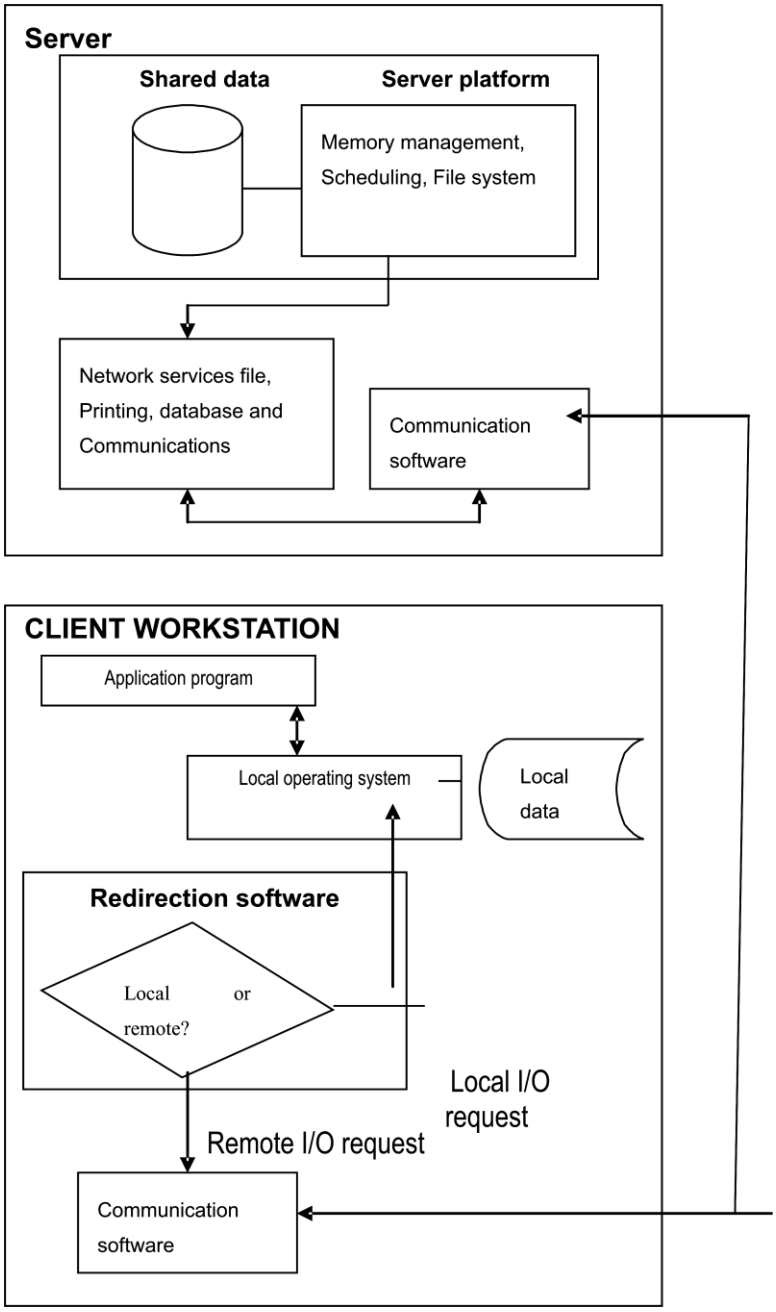


Figure 9.1: NOS architecture

9.6 Functions of NOS

The main functions of NOS can be summarized as follows:

- Redirection
- Communication management
- File / printer services
- Network management

9.6.1 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 may 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.

9.6.2 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.

9.6.3 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.

9.6.4 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 in 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.

9.7 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 9.2). 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.

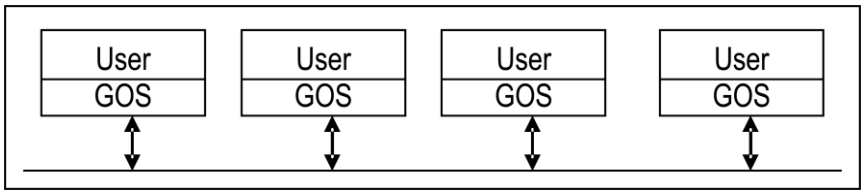


Figure 9.2: GOS environment

9.7.1 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 whereas 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.

9.7.2 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.

9.8 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).

9.8.1 Message Passing Schemes

RPC can be considered as a special case of a generalized remote message-passing scheme as shown in below (Figure 9.3). 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.

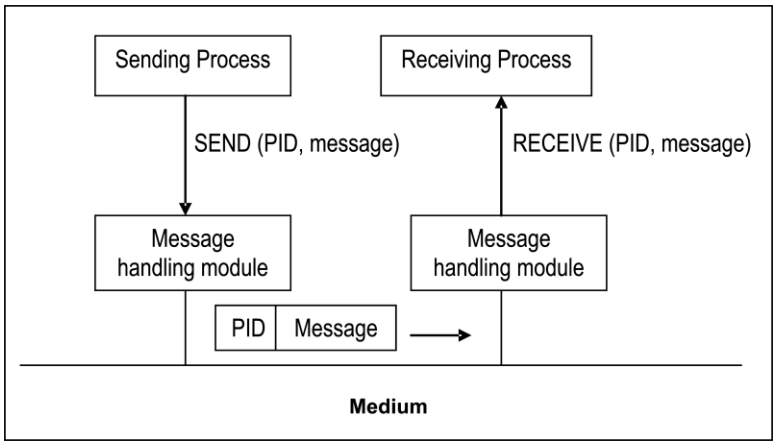


Figure 9.3: Message passing scheme

9.8.2 Types of Services

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 whereas 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.

9.8.3 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.

9.8.4 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 is why only call by value method is used in RPC. A general schematic of RPC is shown below (Figure 9.4). 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.

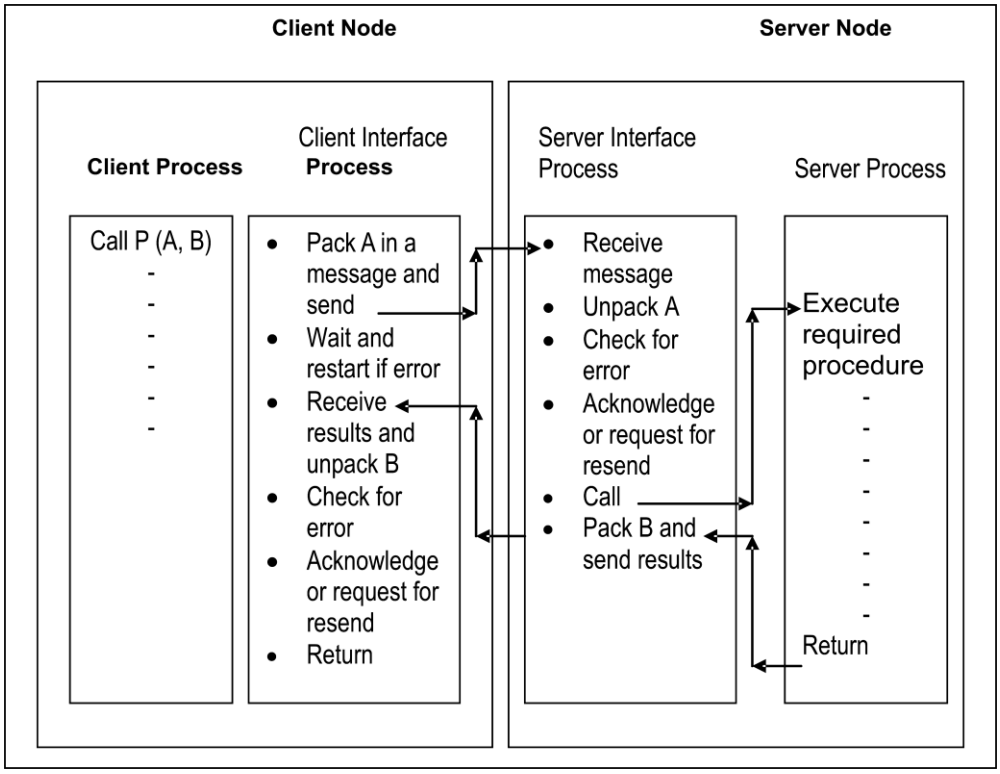


Figure 9.4: A general schematic of RPC

9.8.5 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.

9.8.6 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.

9.9 Distributed File Management

A network has many nodes. Each node has files in its local database. In 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 the number of 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.

9.10 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. In addition to this an overview of Distributed File Management has also been discussed.

Self Assessment Questions

1. Distributed processing and parallel processing have a common goal of _____
2. Distributed processing systems are also called _____.
3. The communication management software runs on _____.
4. In RPC _____ is very difficult because it is difficult to let processors on different machines to share a common address space.
5. A virtual circuit analogous to a telephone service is an example of _____.

9.11 Terminal Questions

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. Write a note on Distributed File Management.

9.12 Answers to Self Assessment Questions and Terminal Questions

Answers to Self Assessment Questions

1. High throughput using more processors.
2. Loosely coupled systems.
3. Both the client and server
4. Call by reference
5. Reliable service

Answers to Terminal Questions

1. Refer section 9.2 and 9.3
2. Refer section 9.4.1 and 9.4.2
3. Refer section 9.5
4. Refer section 9.7.1
5. Refer section 9.8
6. Refer section 9.9

Unit 10 Security and Protection

Structure

10.1 Introduction

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.

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.

Objectives:

At the end of this unit, you will be able to understand:

- Attacks on Security, Meaning of Authentication and Confidentiality.
- Computer Viruses and types of viruses.
- Computer worms.
- Security Design principles.
- Protection Mechanisms and Security in Distributed Environment.

10.2 Attacks on Security

A security system can be attacked in many ways. Some of them are discussed below:

10.2.1 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

10.2.2 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.

10.2.3 Invalid Parameters

Passing of invalid parameters of failure to validate them properly can lead to serious security violations.

10.2.4 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.

10.2.5 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.

10.2.6 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.

10.3 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 or 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.

10.4 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.

10.4.1 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

10.4.2 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.

10.4.3 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.

10.4.4 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.

10.4.5 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.

10.4.6 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.

10.5 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: wherever possible, the system must be designed to depend on more than one condition, for example, two passwords / two keys.

10.6 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 that 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.

10.7 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.

		OBJECTS							
		File 0	**File 1**	**File 2**	**File 3**	**File 4**	**File 5**	**Printer 0**	**Printer 1**
D O M A I N S	0	R W	R					W	
	1			R	R W X				W
	2					W	R W X		W
	3			W		R			

Figure 10.1: Domains in matrix form

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 in Figure 10.1.

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 whereas 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 waste 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.

10.8 Encryption

Encryption is an important tool in protection, security and authentication. The process involves two steps (Figure 10.2):

- Encryption: the original message is changed to some other form
- Decryption: the encrypted message is restored back to the original

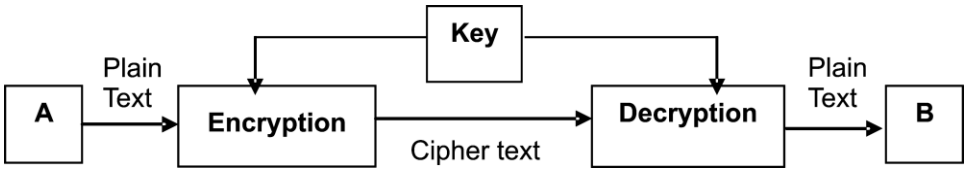


Figure 10.2: Conventional Encryption

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 10.3). 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.

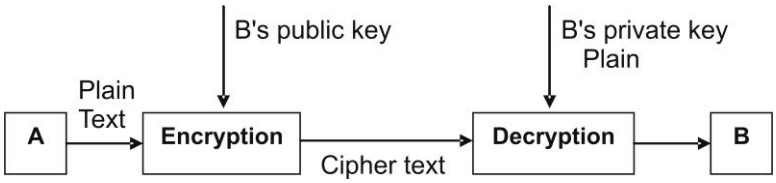


Figure 10.3: Public key Encryption

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.

10.9 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 10.4).

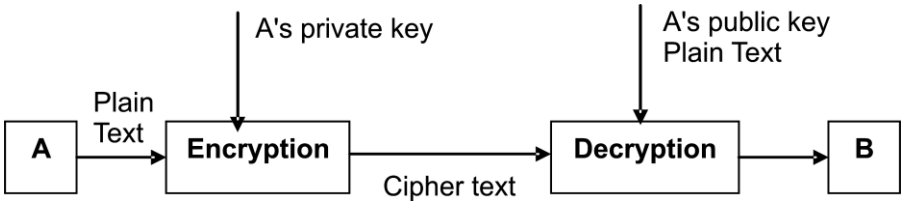


Figure 10.4: Public key Encryption for authentication without protection

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 10.5).

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.

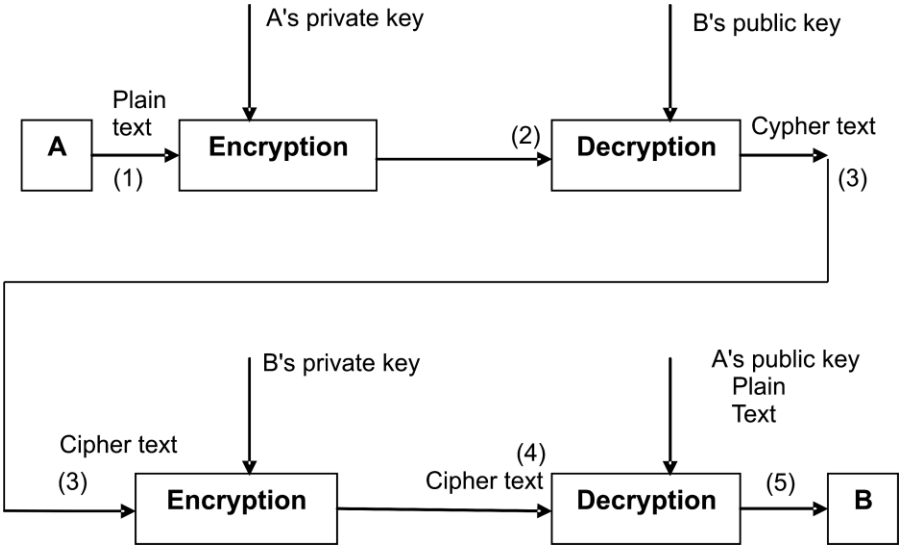


Figure 10.5: Public key Encryption for both authentication and protection

10.10 Summary

This unit 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 has been studied.

Self Assessment Questions

1. Unauthorized use of service (tapping) and unauthorized disclosure of information (disclosure) are _____.
2. _____ is verification of access to system resources.
3. One of the safest ways to prevent virus attacks is to use _____.
4. _____ plays an important role in the effectiveness of the password.
5. In transposition ciphers the contents of the data are not changed but _____.

10.11 Terminal Questions

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.

10.12 Answers to Self Assessment Questions and Terminal Questions

Answers to Self Assessment Questions

1. Passive threats.
2. Authentication.
3. legal copies of software
4. Length of a password
5. the order is changed

Answers to Terminal Questions

1. Refer section 10.2
2. Refer section 10.4
3. Refer section 10.6
4. Refer section 10.7
5. Refer section10.8
6. Refer section 10.9

Unit 11 Multiprocessor Systems

Structure

11.1 Introduction

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.

Objectives:

At the end of this unit, you will be able to understand:

- Advantages of Multiprocessors
- Multiprocessor classification
- Multiprocessor Interconnections
- Types of Multi-processor operating Systems
- Multiprocessor Operating System functions, requirements and Implementation issues.

11.2 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.

11.3 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.

11.4 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

11.4.1 Bus-Oriented systems

A shared bus connects processors and memory in the multiprocessor system as shown below (Figure 11.1).

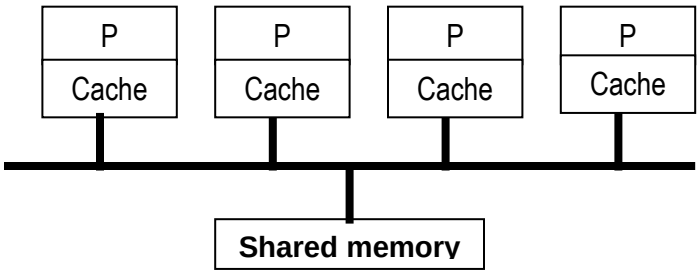


Figure 11.1: Shared-bus multiprocessor organization

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.

11.4.2 Crossbar-Connected Systems

An interconnection of processors and memory in a multiprocessor system using crossbar approach is shown below (Figure 11.2):

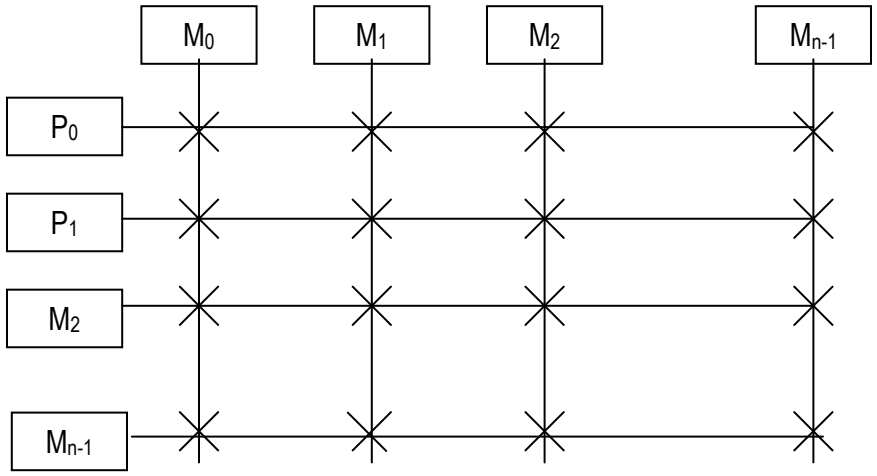


Figure 11.2: Crossbar interconnection

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.

11.4.3 Hyper cubes

A 3-dimensional hypercube can be visualized as shown below (Figure 11.3):

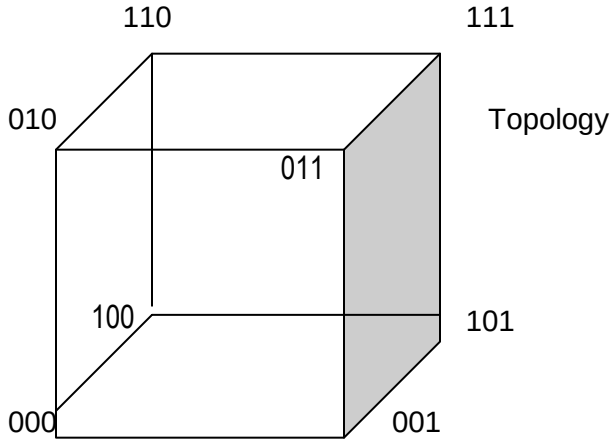


Figure 11.3: 3-dimensional hypercube

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. Hypercube 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. Hypercube 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. Hypercube 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.

11.4.4 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 11.4):

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 $S_i = Di$.

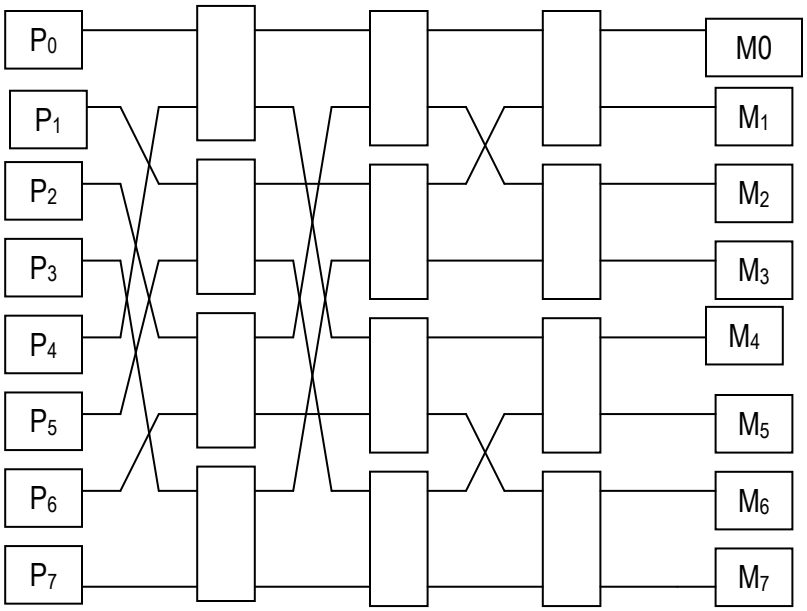


Figure 11.4: Multistage switching network

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.

11.5 Types of Multiprocessor Operating Systems

Three basic types of multiprocessor operating systems are:

- Separate supervisors

- Master / slave
- Symmetric

11.5.1 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.

11.5.2 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.

11.4.5 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.

11.6 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 facts 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.

11.7 Operating System Design and Implementation Issues

Some major issues involved in processor and memory management in multiprocessor operating systems are described below:

11.7.1 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 but 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 multi-programmed. Since multi-programming 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.

11.7.2 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.

11.8 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.

Self Assessment Questions

1. _____ can be improved by executing a number of unrelated user processes on different processors in parallel.
2. MIMD means _____.
3. _____ occurs when more than one processor attempts to access the same memory at the same time.
4. Processors and memory in a multiprocessor system can be interconnected by use of _____.
5. In a uniprocessor system an illusion of multiprocessing is created by multiplexing the processor among _____.

11.9 Terminal Questions

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.

11.10 Answers to Self Assessment Questions and Terminal Questions

Answers to Self Assessment Questions

1. Throughput
2. Multiple Instruction Streams, Multiple Data Stream
3. Contention

4. Multistage switch
5. virtual processes

Answers to Terminal Questions

1. Refer section 11.2
2. Refer section 11.3
3. Refer section 11.4
4. Refer section 11.5
5. Refer section11.7

References:

1. "**Operating System Concepts**" by Abraham Silberschatz & Peter Baer Galvin
2. **"Operating Systems: Design and Implementation" by** Andrew S. Tanenbaum
3. **"Modern Operating Systems" by** Andrew S. Tanenbaum