New Edition

Operating Systems

B.Sc. CSIT

Bhupendra Singh Saud

Deepak Bhatt

Contents

1 Chapter OPERATING SYSTEM OVERVIEW

1. Introduction 1
2. What is operating system? 1
3. Aspects of Operating System 2
4. Evolution of operating System 4
5. System Calls 8
6. System Programs 10
7. Structure of Operating System 11
8. The Shell 14
9. Open Source Operating System 15

- ❑ Laboratory Works 17
- ❑ **Exercise** **19**

2 Chapter PROCESS MANAGEMENT

1. Introduction to Process 21
2. Program 22
3. Process Model 23
4. Process Life Cycle 24
5. Process Control Block (PCB) 25
6. Multitasking 27
7. Multiprogramming 28
8. Multiprocessing 29
9. Threads 29
10. The Thread Model 31
11. nter Process Communication 35
12. Race condition 35

13. Critical section 36
14. Synchronization Hardware 37
15. Disabling Interrupts 38
16. Lock variables 38
17. Turn Variable or Strict alternation 38
18. Peterson's solution 39
19. Test and set lock 40
20. Sleep and Wakeup 41
21. Semaphore 41
22. Monitors 42
23. Message passing 43
24. The Producer-Consumer Problem with Message Passing 45
25. Classical IPC Problems 46
26. Process Scheduling 51
27. Process Scheduling Queues 51
28. Schedulers 52
29. Dispatcher 53
30. CPU - I/O Burst Cycle 54
31. Context Switch 54
32. Non-preemptive scheduling 55
33. Preemptive Scheduling 56
34. Scheduling Criteria 57
35. Batch System Scheduling 57
36. Interactive System Scheduling 61
37. Overview of real time system scheduling 63
- Laboratory Works 64
- Exercise 71

3 Chapter PROCESS DEADLOCKS

1. Introduction to Deadlock 73
2. Deadlock Characterization 74
3. Preemptable and Nonpreemptable Resources 75
4. Resource Allocation Graph (RAG) 75
5. Checking Deadlock (Safe or not) 78
6. Dealing with Deadlock Problem 80
7. Deadlock Recovery 90
- Laboratory Works 91
- Exercises 93

4 Chapter MEMORY MANAGEMENT

1. Introduction 95
2. Mono-programming 95
3. Multi-programming 96
4. Modeling Multiprogramming 99
5. Multiprogramming with fixed and variable partitions 100
6. Relocation and Protection 102
7. Memory Management (Bitmaps & Linked-list) 104
8. Memory Allocation Strategies 104
9. Virtual Memory 107
10. Swapping 109
11. Paging 109
12. Handling Page Faults 111
13. TLB's 112
14. Demand Paging 113
15. Page Replacement Algorithms 114
16. Concept of Locality of Reference 121
17. Thrashing 121
18. Working Set 122
19. WS- Clock Page Replacement Algorithms 124
20. Segmentation 125
21. Segmentation with Paging (MULTICS) 127
22. Fragmentation 128
- Laboratory Works 131
- Exercise 138

5 Chapter FILE MANAGEMENT

1. File Overview 141
2. File Naming 141
3. Basic File Operations 143
4. File Structure 143

5. File Types 144
6. File Access 144
7. File Attributes 146
8. File Operations 146
9. Directory Structure 147
10. Implementing Files 151
11. Linked List Allocation using Table in Memory 154
12. I-nodes 155
13. Directory Operations 157
14. Path Names 157
15. Directory Implementation 158
16. Shared Files 159
17. Free Space Management 160
❑ Laboratory Works 163
❑ Exercise 163

6 Chapter DEVICE MANAGEMENT

1. Introduction 165
2. Classification of I/O devices 165
3. Device Controllers 167
4. Communication of CPU to I/O Devices 168
5. Interrupts 171
6. Goals of I/O Software 172
7. Handling I/O 174
8. I/O Software Layers 177
9. Disk Structure 179
10. Disk Scheduling 183
11. Disk Formatting 187
12. Cylinder Skew 188
13. Interleaving 189
14. Error Handling 189
15. Redundant Array of Independent Disks (RAID) 190
❑ Laboratory Works 198
❑ Exercise 204

7 Chapter LINUX CASE STUDY

1. History 207
2. Features of LINUX Operating System 208
3. Components of LINUX System 208
4. LINUX System Architecture 209
5. Kernel Modules 210
6. Process Management 212
7. Creation of a Processes in Linux 212
8. Scheduler 213
9. Inter-process Communication 214
10. Memory Management 215
❑ Exercise 217
❑ **Reference** **219**

Chapter 1

OPERATING SYSTEM OVERVIEW

1. Introduction

Computer has a great importance in today's world. Most of the things done today are because of computers. The computer is not only a bare hardware; it consists of proper combination of hardware and software. The one most important software required to run computer is operating system.

One or more processor, some main memory, disks, printers, keyboards, display, etc. are the major components of today's computer. As a whole today's computer is a complex system. These complex systems require running smoothly where as a record of every component should be kept every time a task is done, for these all an operating system is required.

2. What is operating system?

It is difficult for one to define operating system in a precise manner. We can say what an operating system does and what for operating system is? Though operating system can be defined as "an interface between users and hardware, which hides the complexities of hardware from users i.e. operating system performs the logical operations in machine language and displays it to user in a non-complex manner". In general Operating system is system software designed to run a computer's hardware and application programs.

For large systems, the operating system has even greater responsibilities and powers. It is like a traffic police, which makes sure that different program and users running at the same time do not interfere with each other. The operating system is also responsible for security, ensuring that unauthorized users do not access the system.

In other words, an operating system is a program that acts as an intermediary between a user of a computer and the computer hardware, to provide an environment in which a user can execute programs. The primary goal of an operating system is thus to make the computer system convenient to use. And to use the computer hardware in an efficient manner. In brief, an operating system is the set of programs that controls a computer. An operating system is an important part of almost every computer system. This can be divided roughly into four components: the hardware, the operating system, the application programs and the users.

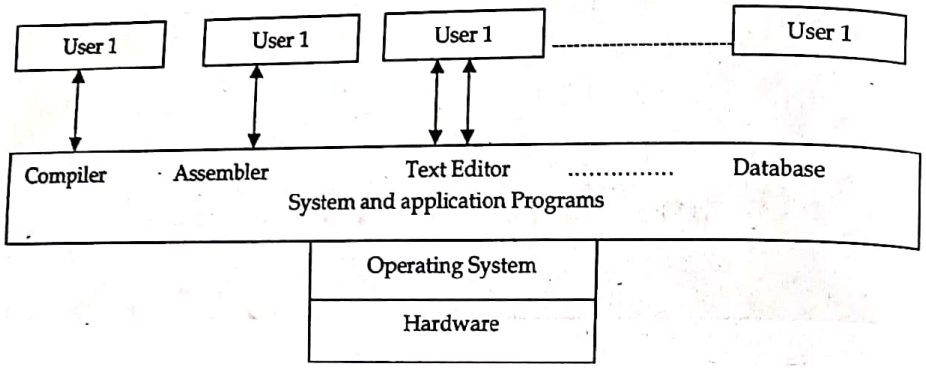


Fig 1.1: An overview of Complex System.

Hardware, provides basic computing resources (CPU, Memory, I/O devices etc.) whereas Operating System controls and co-ordinates the use of the hardware among the various users. It is usually that portion of software that runs in kernel mode or supervisor mode. The application Programs define the way in which the system resources are used to solve the computing problems and users are the People, machine, other computers or processes. Some common operating systems are desktop operating system; network operating system; mobile operating system; embedded system operating system etc.

3. Aspects of Operating System

The aspect of operating system also called views or functions of operating system are generally covered in two parts.

- Operating system as an extended machine
- Operating system as a resource manager

3.1 Operating system as an extended machine

As we have discussed earlier today's computers comes with multiple components, we (users) are not interested on how these components work together to perform our task, all we need is our task needs to be completed with no complexity and overhead. We are no more interested about the read write commands performed during computing nor about the error and status list maintained by system.

Thus all we need is that the program should hide the truth about the hardware from the programmer and presents a nice, simple view of named files that can be read and written. This is the very primitive task of operating system. User is neither interested weather the file is written in CD or hard drive or some other device, nor about how it is stored? Contiguously or is broken into blocks. In this view, the function of the operating system is to present the user with the equivalence of an extended machine or virtual machine that is easier to program than the underlying hardware. To achieve it all, the operating system provides a variety of services that programs can obtain using special instructions called system calls. (Will be covered later)

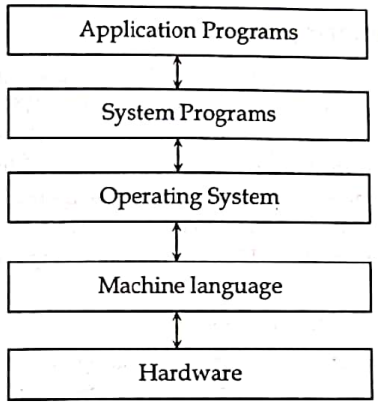


Fig 1.2: OS act as extended machine

3.2 The Operating System as a Resource Manager

No matter what kind of machine you have constructed, its major function is how resources are managed. For example, we have constructed a system which provides all necessary banking operations, one need to deposit amount in one account meanwhile another user requires to print statement regarding to the account details i.e. first user requires to access the database while the second one too requires to access database as well as printer. As we have discussed earlier users are interested in getting their job done. But the problem occurs with the system how to manage these requests. The process of managing all these (resources) is resource management, and this is done by resource manager. Now let's think about complex system (computer).

- What happens if three programs try to print their output on the same printer at the same time?
- What happens if two network users try to update a shared document at same time?

The primary function is to manage all pieces of a complex system i.e. provide all necessary resources to the process asking for it.

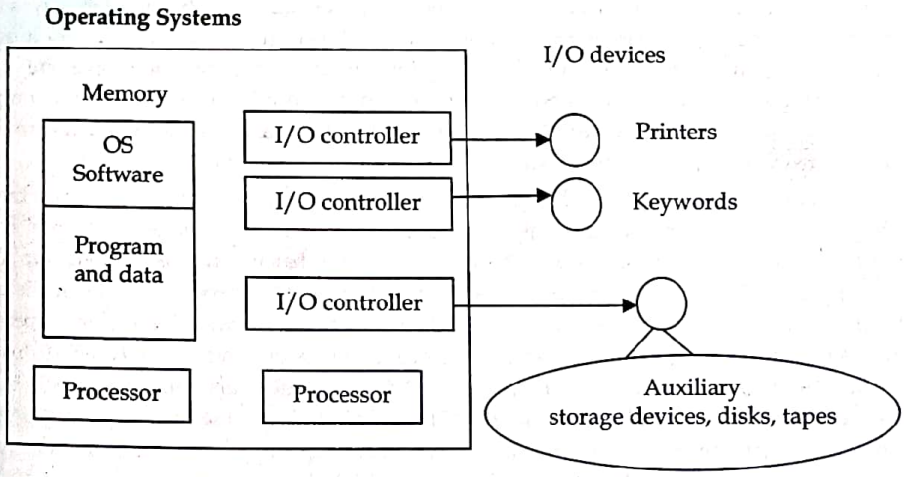


Fig 1.3: OS act as resource manager

4. Evolution of operating System

Operating systems have been evolving through the years. The first true digital computer was designed by the English mathematician Charles Babbage (1792-1871). Babbage spent most of the life trying to build his "analytical engine." He was unsuccessful because of the technology of those days could not produce the necessary parts required like wheels, gears, and cogs to the high precision for his machine. Needless to say, the analytical engine did not have an operating system. Babbage realized that he would need software for his analytical engine, so he hired a young woman named Ada Lovelace. The programming language **Ada** is named after her.

4.1 The First Generation (1945-55) Vacuum Tubes

The early computers of the late 1940s had no operating system. Every program that ran on these early systems had to include all of the code necessary to run the computer, communicate with connected hardware, and perform the computation the program was actually intended to perform. This situation meant that even simple programs were complex. As computer systems diversified and became more complex and powerful, it became increasingly impractical to write programs that functioned as both an operating system and a useful application.

Human operators scheduled jobs for execution and supervised the use of the computer's resources. Because these early computers were very expensive, the main purpose of an operating system in these early days was to make the hardware as efficient as possible. Now, computer hardware is relatively cheap by comparison with the cost of the personnel required to operate it, so the purpose of the operating system has evolved to encompass the task of making the user as efficient as possible.

4.2 The Second Generation (1955-65) Transistors and Batch Systems

In 1950s transistor were introduced and for the first time, there was a clear separation between designers, builders, operators, programmers, and maintenance personnel. These machines (now called mainframes) were locked in air conditioned rooms, with professional operators to run them. To run a job, a programmer would first write the program on paper (in FORTRAN or assembler), then punch it on cards. When the computer finished whatever job it was currently running, an operator would go over to the printer and tear off the output and carry it over to the output room, so that the programmer could collect it later. Then he would take one of the card decks that had been brought from the input room and read it in. If the FORTRAN compiler was needed, the operator would have to get it from a file cabinet and read it in. Much computer time was wasted while operators were walking around the machine room.

To speed up processing, jobs with similar needs were batched together and were run through the computer as a group, which was introduced in the same generation. These computers used batch operating systems, in which the computer ran batches of jobs without stop. Programs were punched into cards that were usually copied to tape for processing. When the computer finished one job, it would immediately start the next one on the tape. Professional operators, not the users, interacted with the machine. Users dropped jobs off, and then returned to pick up the results after their jobs had run. This was inconvenient for the users, but the expensive computer was kept busy with a steady stream of jobs. The definitive feature of a batch system is the lack of interaction between the user and the job while that job is executing. In this execution environment, the CPU is often idle. The problems with Batch Systems are as follows:

- Lack of interaction between the user and the job.
- CPU is often idle, because the speed of the mechanical I/O devices is slower than the CPU.
- Difficult to provide the desired priority.

4.3 The Third Generation (1965-1980) ICs and Multiprogramming

The period of third generation was from 1965-1980. The computers of third generation used Integrated Circuits (ICs) in place of transistors. A single IC has many transistors, resistors, and capacitors along with the associated circuitry. This development made computers smaller in size, reliable, and efficient. In this generation remote processing, time-sharing, multiprogramming operating system was used. High-level languages (FORTRAN-II TO IV, COBOL, PASCAL PL/1, BASIC, ALGOL-68 etc.) were used during this generation. During this generation several advancements were done in computer operating system like multiprogramming, time sharing, Real time operating systems were introduced.

Multi programming System

To overcome the problem of underutilization of CPU and main memory, the multiprogramming was introduced. The multiprogramming is interleaved execution of multiple jobs by the same computer. In multiprogramming system, when one program is waiting for I/O transfer; there is another program ready to utilize the CPU. So it is possible for several jobs to share the time of the CPU. But it is important to note that multiprogramming is not defined to be the execution of jobs at the same instance of time. Rather it does mean that there are a number of jobs available to the CPU (placed in main memory) and a portion of one is executed then a segment of another and so on. A simple process of multiprogramming is shown in figure

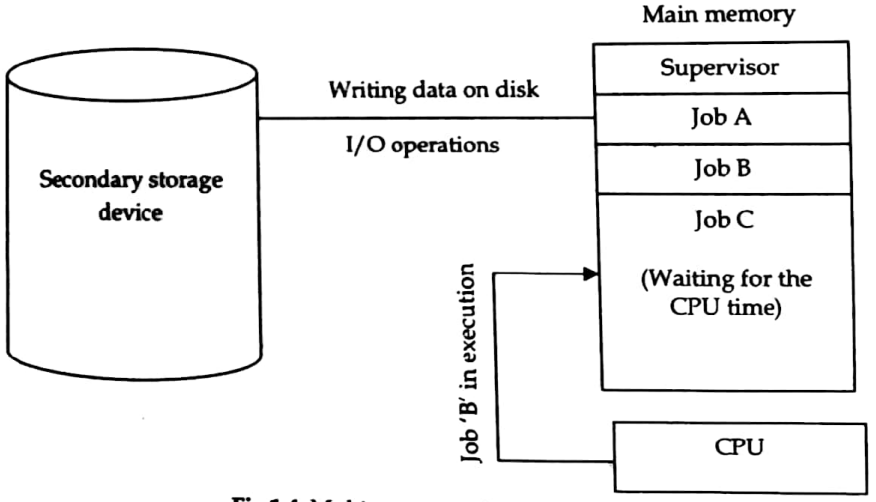


Fig 1.4: Multi programming system

As shown in fig, at the particular situation, job' A' is not utilizing the CPU time because it is busy in I/0 operations. Hence the CPU becomes busy to execute the job 'B'. Another job C is waiting for the CPU for getting its execution time. So in this state the CPU will never be idle and utilizes maximum of its time. A program in execution is called a "Process", "Job" or a "Task". The concurrent execution of programs improves the utilization of system resources and enhances the

system throughput as compared to batch and serial processing. In this system, when a process requests some I/O to allocate; meanwhile the CPU time is assigned to another ready process. So, here when a process is switched to an I/O operation, the CPU is not set idle.

Time Sharing System

Time-sharing is a technique which enables many people, located at various terminals, to use a particular computer system at the same time. Time-sharing or multitasking is a logical extension of multiprogramming. Processor's time which is shared among multiple users simultaneously is termed as time-sharing. The main difference between Multi-Programmed Batch Systems and Time-Sharing Systems is that in case of Multi-Programmed batch systems, the objective is to maximize processor use, whereas in Time-Sharing Systems, the objective is to minimize response time. Multiple jobs are executed by the CPU by switching between them, but the switches occur so frequently. Thus, the user can receive an immediate response. For example, in a transaction processing, the processor executes each user program in a short burst or quantum of computation. That is, if n users are present, then each user can get a time quantum. When the user submits the command, the response time is in few seconds at most.

The operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time. Computer systems that were designed primarily as batch systems have been modified to time-sharing systems.

Advantages of Timesharing operating systems are as follows

- Provides the advantage of quick response
- Avoids duplication of software.
- Reduces CPU idle time.

Disadvantages of Time-sharing operating systems are as follows

- Problem of reliability.
- Question of security and integrity of user programs and data.
- Problem of data communication.

Real Time Operating System

A real-time system is defined as a data processing system in which the time interval required to process and respond to inputs is so small that it controls the environment. The time taken by the system to respond to an input and display of required updated information is termed as the response time. So in this method, the response time is very less as compared to online processing.

Real-time systems are used when there are rigid time requirements on the operation of a processor or the flow of data and real-time systems can be used as a control device in a dedicated application. A real-time operating system must have well-defined, fixed time constraints, otherwise the system will fail. For example, scientific experiments, medical image systems, industrial control systems, weapon systems, robots, air traffic control systems, etc.

There are two types of real-time operating systems

1. **Hard real-time systems:** Hard real-time systems guarantee that critical tasks complete on time. In hard real-time systems, secondary storage is limited or missing and the data is stored in ROM. In these systems, virtual memory is almost never found.
2. **Soft real-time systems:** Soft real-time systems are less restrictive. A critical real-time task gets priority over other tasks and retains the priority until it completes. Soft real-time systems have limited utility than hard real-time systems. For example, multimedia, virtual reality, Advanced Scientific Projects likes undersea exploration and planetary rovers, etc.

4.4 The Fourth Generation (1980-Present) Personal Computers

With the development of LSI (Large Scale Integration) circuits, chips containing thousands of transistors on a square centimeter of silicon, the age of the personal computer dawned, personal Computers were easy to. These were chips containing thousands of transistors on a square centimeter of silicon. Because of these, microcomputers were much cheaper than minicomputers and that made it possible for a single individual to own one of them.

The advent of personal computers also led to the growth of networks. This created network operating systems and distributed operating systems. The users were aware of a network while using a network operating system and could log in to remote machines and copy files from one machine to another.

Distributed System

Distributed systems use multiple central processors to serve multiple real-time applications and multiple users. Data processing jobs are distributed among the processors accordingly. The processors communicate with one another through various communication lines (such as high-speed buses or telephone lines). These are referred as loosely coupled systems or distributed systems. Processors in a distributed system may vary in size and function. These processors are referred as sites, nodes, computers, and so on.

The advantages of distributed systems are as follows

- With resource sharing facility, a user at one site may be able to use the resources available at another.
- Speedup the exchange of data with one another via electronic mail.
- If one site fails in a distributed system, the remaining sites can potentially continue operating.
- Better service to the customers.
- Reduction of the load on the host computer.
- Reduction of delays in data processing.

Network Operating System

Network Operating System runs on a server and provides the server the capability to manage data, users, groups, security, applications, and other networking functions. The primary purpose of the network operating system is to allow shared file and printer access among multiple computers in a network, typically a local area network (LAN), a private network or to other networks. Examples of network operating systems include Microsoft Windows Server 2003, Microsoft Windows Server 2008, UNIX, Linux, Mac OS X, Novell NetWare, and BSD.

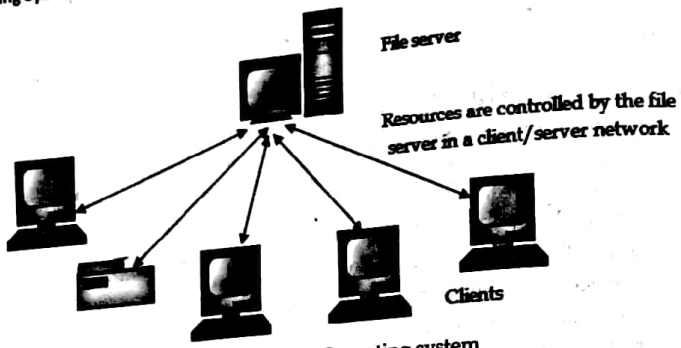


Fig 1.5: Network Operating system

The advantages of network operating systems are as follows

- Centralized servers are highly stable.
- Security is server managed.
- Upgrades to new technologies and hardware can be easily integrated into the system.
- Remote access to servers is possible from different locations and types of systems.

The disadvantages of network operating systems are as follows

- High cost of buying and running a server.
- Dependency on a central location for most operations.
- Regular maintenance and updates are required.

5. System Calls

The interface between a process and an operating system is provided by system calls. In general, system calls are available as assembly language instructions. They are also included in the manuals used by the assembly level programmers. System calls are usually made when a process in user mode requires access to a resource. Then it requests the kernel to provide the resource via a system call. A figure representing the execution of the system call is given as follows:

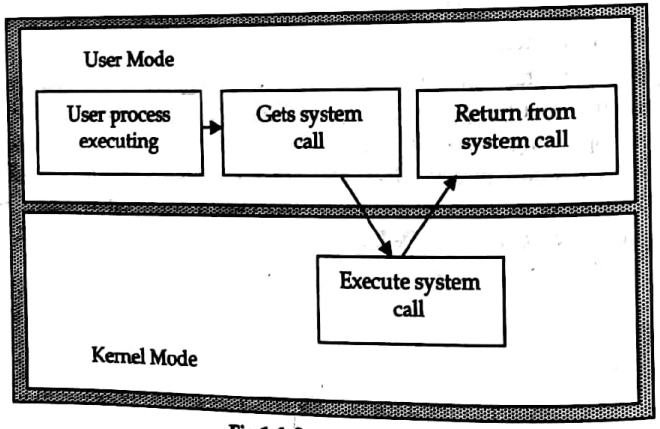


Fig 1.6: System calls

As can be seen from this diagram, the processes execute normally in the user mode until a system call interrupts this. Then the system call is executed on a priority basis in the kernel mode. After the execution of the system call, the control returns to the user mode and execution of user processes can be resumed. In general, system calls are required in the following situations:

- If a file system requires the creation or deletion of files. Reading and writing from files also require a system call.
- Creation and management of new processes.
- Network connections also require system calls. This includes sending and receiving packets.
- Access to a hardware device such as a printer, scanner etc. requires a system call.

Types of System Calls

There are mainly five types of system calls. These are explained in detail as follows:

- **Process Control**: These system calls deal with processes such as process creation, process termination etc.
- **File Management:** These system calls are responsible for file manipulation such as creating a file, reading a file, writing into a file etc.
- **Device Management:** These system calls are responsible for device manipulation such as reading from device buffers, writing into device buffers etc.
- **Information Maintenance:** These system calls handle information and its transfer between the operating system and the user program.
- **Communication:** These system calls are useful for inter process communication. They also deal with creating and deleting a communication connection.

Examples of Windows and UNIX System Calls

	Windows	Unix
Process Control	CreateProcess(),ExitProcess() WaitForSingleObject()	fork() exit(),wait()
File Manipulation	CreateFile(),ReadFile() WriteFile(),CloseHandle()	open(),read() write(),close()
Device Manipulation	SetConsoleMode(),ReadConsole() WriteConsole()	ioctl() read(),write()
Information Maintenance	GetCurrentProcessID(),SetTimer() Sleep()	getpid() alarm(),sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

6. System Programs

System programs provide an environment where programs can be developed and executed. In the simplest sense, system programs also provide a bridge between the user interface and system calls. In reality, they are much more complex. For example, a compiler is a complex system program. The system programs are used to program the operating system software. While application programs provide software that is used directly by the user, system programs provide software that are used by other systems such as SaaS applications, computational science applications etc. Most system programs are created to have a low runtime overhead. These programs may have small runtime library. Some parts of the system programs may be directly written in assembly language by the programmers. A debugger cannot be used on system programs mostly. This problem can be solved by running the programs in a simulated environment. Some examples of system programs are operating system, networking system, web site server, data backup server etc.

System Programs Purpose

The system program serves as a part of the operating system. It traditionally lies between the user interface and the system calls. The user view of the system is actually defined by system programs and not system calls because that is what they interact with and system programs are closer to the user interface.

An image that describes system programs in the operating system hierarchy is as follows:

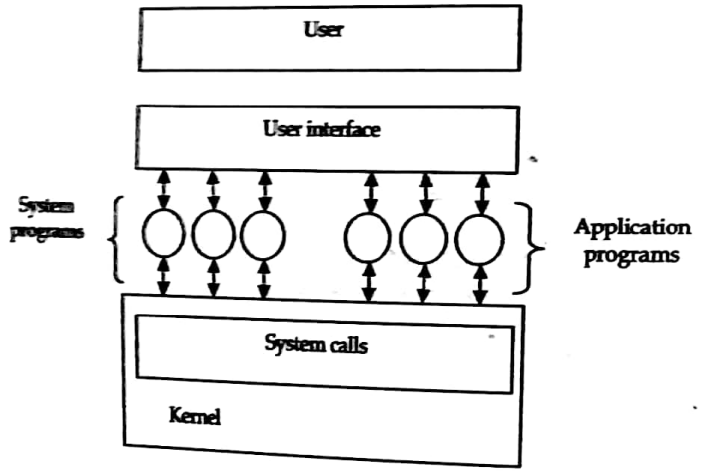


Fig 1.7: System program structure

In the above image, system programs as well as application programs form a bridge between the user interface and the system calls. So, from the user view the operating system observed is actually the system programs and not the system calls.

Types of System Programs

System programs can be divided into seven parts. These are given as follows:

- **Status Information:** The status information system programs provide required data on the current or past status of the system. This may include the system date, system time, and available memory in system, disk space, logged in users etc.
- **Communications:** These system programs are needed for system communications such as web browsers. Web browsers allow systems to communicate and access information from the network as required.
- **File Manipulation:** These system programs are used to manipulate system files. This can be done using various commands like create, delete, copy, rename, print etc. These commands can create files, delete files, copy the contents of one file into another, rename files, print them etc.
- **Program Loading and Execution:** The system programs that deal with program loading and execution make sure that programs can be loaded into memory and executed correctly. Loaders and Linkers are a prime example of this type of system programs.
- **File Modification:** System programs that are used for file modification basically change the data in the file or modify it in some other way. Text editors are a big example of file modification system programs.
- **Application Programs:** Application programs can perform a wide range of services as per the needs of the users. These include programs for database systems, word processors, plotting tools, spreadsheets, games, scientific applications etc.
- **Programming Language Support:** These system programs provide additional support features for different programming languages. Some examples of these are compilers, debuggers etc. These compile a program and make sure it is error free respectively.

7. Structure of Operating System

Many commercial operating systems do not have well defined structures. Such systems started as small, simple, and limited systems and then grew beyond their original scope. MS-DOS is an example of such system. In MS-DOS, the interface and the levels of functionality are not well separated. Application programs are able to access the basic I/O routines to write directly to the display and disk drives. Such program leaves MS-DOS vulnerable to malicious programs, causing entire system crashes when user program fails.

7.1 Simple Structure

There are many operating systems that have a rather simple structure. These started as small systems and rapidly expanded much further than their scope. A common example of this is MS-DOS. It was designed simply for a niche amount for people. There was no indication that it would become so popular. An image to illustrate the structure of MS-DOS is as follows:

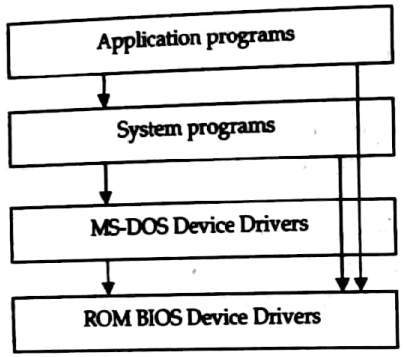


Fig. 1.8: MS-DOS Structure

It is better that operating systems have a modular structure, unlike MS-DOS. That would lead to greater control over the computer system and its various applications. The modular structure would also allow the programmers to hide information as required and implement internal routines as they see fit without changing the outer specifications.

7.2 Layered Structure

One way to achieve modularity in the operating system is the layered approach. This approach breaks up the operating system into different layers. In this, the bottom layer is the hardware and the topmost layer is the user interface.

This allows implementers to change the inner workings, and increases modularity. As long as the external interface of the routines doesn't change, developers have more freedom to change the inner workings of the routines. With the layered approach, the bottom layer is the hardware, while the highest layer is the user interface. The main advantage with layered structure is simplicity of construction and debugging. And the main difficulty is defining the various layers. Also this OS tends to be less efficient than other implementations. An image demonstrating the layered approach is as follows:

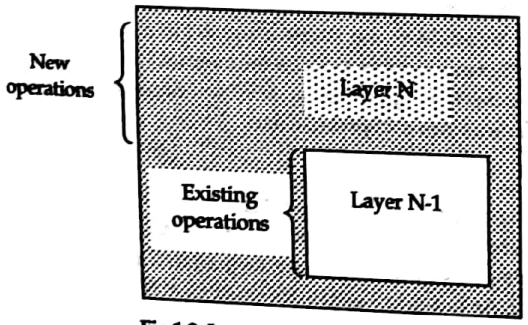


Fig 1.9: Layered Structure of Operating system

As seen from the image, each upper layer is built on the bottom layer. All the layers hide some structures, operations etc. from their upper layers. One problem with the layered structure is that each layer needs to be carefully defined. This is necessary because the upper layers can only use the functionalities of the layers below them.

7.3 Monolithic Systems

These systems are written as a collection of procedures, each of which can call any of the other ones whenever it needs to. When this technique is used, each procedure in the system has a well-defined interface in terms of parameters and results, and each one is free to call any other one, if the latter provides some useful computation that the former needs. Even in monolithic systems, however, it is possible to have *at* least a little structure. The services (system calls) provided by the operating system are requested by putting the parameters in a well-defined place (e.g., on the stack) and then executing a trap instruction. This instruction switches the machine from user mode to kernel mode and transfers control to the operating system. The operating system then fetches the parameters and determines which system call is to be carried out. After that, it indexes into a table that contains in slot *k* a pointer to the procedure that carries out system call *k*. This organization suggests a basic structure for the operating system:

- A main program that invokes the requested service procedure.
- A set of service procedures that carry out the system calls.
- A set of utility procedures that help the service procedures.

In this model, for each system call there is one service procedure that takes care of it. The utility procedures do things that are needed by several service procedures, such as fetching data from user programs.

This division of the procedures into three layers is shown in Fig. below:

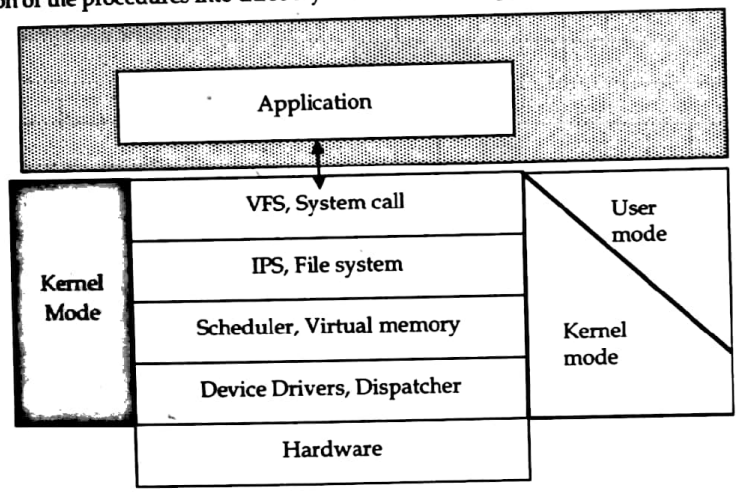


Fig 1.10: Monolithic system structure

7.4 Microkernel

This structures the operating system by removing all nonessential portions of the kernel and implementing them as system and user level programs. Generally, they provide minimal process and memory management, and a communications facility and communication between components of the OS is provided by message passing. With microkernel, Extending the operating system becomes much easier as well as any changes to the kernel tend to be fewer, since the kernel is smaller and the microkernel also provides more security and reliability. The

ıain disadvantage of microkernel is the poor performance due to increased system overhead rom message passing.

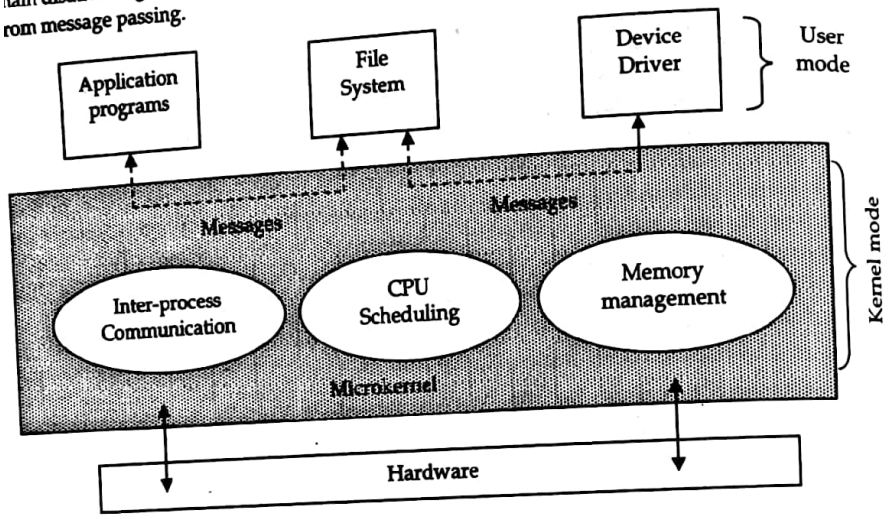


Fig 1.11: Microkernel structure

8. The Shell

The operating system carries out the system calls. Editors, compilers, assemblers, linkers, and command interpreters definitely are not part of the operating system, even though they are important and useful, called the shell. Shell is not part of the operating system, but makes heavy use of many operating system features and thus serves as a good example of how the system calls can be used. It is also the primary interface between a user and the operating system, unless the user is using a graphical user interface. Examples of some shells are *sh, csh, ksh,* and *bash.*

When any user logs in, a shell is started up. The shell has the terminal as standard input and standard output. It starts out by typing the prompt, a character such as a dollar sign, which tells the user that the shell is waiting to accept a command. If the user types

Date then the shell creates a child process and runs the date program as the child. When the child finishes, the shell types the prompt again and tries to read the next input line. The user can specify that standard output be redirected to a file, for example,

```
date >file
```

Similarly, standard input can be redirected, as in

```
sort <file1 >file2
```

This invokes the sort program with input taken from file1 and output sent to file2.

Trap: A trap is a software generated interrupt caused either by an error (for example, division by zero or invalid memory access) or by a specific request from a user that an operating system service be provided.

9. Open Source Operating System

In general, open source refers to any program whose source code is made available for use or modification as users or other developers see fit. Open source software is usually developed as a public collaboration and made freely available. Open Source operating systems are released under a license where the copyright holder allows others to study, change as well as distribute the software to other people. This can be done for any reason. The different open source operating systems available in the market are:

- **Cosmos:** This is an open source operating system written mostly in programming language C#. Its full form is C# Open Source Managed Operating System. Till 2016, Cosmos did not intend to be a fully-fledged operating system but a system that allowed other developers to easily build their own operating systems. It also hid the inner workings of the hardware from the developers' thus providing data abstraction.
- **FreeDOS:** This was a free operating system developed for systems compatible with IBM PC computers. FreeDOS provides a complete environment to run legacy software and other embedded systems. It can boot from a floppy disk or USB flash drive as required. FreeDOS is licensed under the GNU General Public license and contains free and open source software. So there are no license fees required for its distribution and changes to the system are permitted.
- **Genode:** Genode is free as well as open source. It contains a microkernel layer and different user components. It is one of the few open source operating systems not derived from a licensed operating system such as UNIX. Genode can be used as an operating system for computers, tablets etc. as required. It is also used as a base for virtualization, inter-process communication, software development etc. as it has a small code system.
- **Ghost OS:** This is a free, open source operating system developed for personal computers. It started as a research project and developed to contain various advanced features like graphical user interface, C library etc. The Ghost operating system features multiprocessing and multitasking and is based on the Ghost Kernel. Most of the programming in Ghost OS is done in C++.
- **ITS:** The incompatible time-sharing system was developed by the MIT Artificial Intelligence Library. It is principally a time sharing system. There is a remote login facility which allowed guest users to informally try out the operating system and its features using Arpanet. ITS also gave out many new features that were unique at that time such as device independent graphics terminal, virtual devices, inter machine file system access etc.
- **OSv:** This was an operating system released in 2013. It was mainly focused on cloud computing and was built to run on top of a virtual machine as a guest. This is the reason it doesn't include drivers for bare hardware. In the OSv operating system, everything runs in the kernel address space and there is no concept of a multi-user system.
- **Phantom OS:** This is an operating system that is based on the concepts on persistent virtual memory and is code oriented. It was mostly developed by Russian developers. Phantom OS is not based on concepts of famous operating systems such as UNIX. Its main goal is simplicity and effectiveness in process management.

Since the development of computer, they are used for different purposes. Similarly, for different purposes, different operating system has been developed. Depending upon the usages, following are some operating systems currently existing in the market.

- **Mainframe Operating Systems:** These computers distinguish themselves from personal computers in terms of their I/O capacity. A mainframe with 1000 disks and thousands of gigabytes of data is not unusual: a personal computer with these specifications would be odd indeed. Mainframes are also making something of a comeback as high-end Web servers, servers for large-scale electronic commerce sites, and servers for business-to-business transactions.

  The operating systems for mainframes are heavily oriented toward processing many jobs at once, most of which need prodigious amounts of I/O. They typically offer three kinds of services: batch, transaction processing, and timesharing. A batch system is one that processes routine jobs without any interactive user present. Claims processing in an insurance company or sales reporting for a chain of stores are typically done in batch mode. Transaction processing systems handle large numbers of small requests; for example, check processing at a bank or airline reservations. Each unit of work is small, but the system must handle hundreds or thousands per second. Timesharing systems allow multiple remote users to run jobs on the computer at once, such as querying a big database. These functions are closely related: mainframe operating systems often perform all of them. An example mainframe operating system is OS/390, a descendant of OS/360.

- **Server Operating Systems:** They run on servers, which are very large personal computers, workstations, or even mainframes. They serve multiple users at once over a network and allow the users to share hardware and software resources. Servers can provide print service, file service, or Web service. Typical server operating systems are UNIX and Windows 2000. Linux is also gaining ground for servers.

- **Multiprocessor Operating Systems:** An increasingly common way to get major-league computing power is to connect multiple CPUs into a single system. Depending on precisely how they are connected and what is shared, these systems are called parallel computers, multi-computers, or multiprocessors. They need special operating systems, but often these are variations on the server operating systems, with special features for communication and connectivity.

- **Personal Computer Operating Systems:** Their job is to provide a good interface to a single user. They are widely used for word processing, spreadsheets, and Internet access. Common examples are Windows 2000, Windows vista, windows 7, the Macintosh operating system, and Linux. Personal computer operating systems are so widely known that probably little introduction is needed.

- **Real-Time Operating Systems:** These systems are characterized by having time as a key parameter. For example, in industrial process control systems, real-time computers have to collect data about the production process and use it to control machines in the factory. For example, if a car is moving down an assembly line, certain actions must take place at certain instants of time, if a welding robot welds too early or too late, the car will be ruined, a hard **real-time system.** Another kind of real-time system is a **soft real-time** system, in which missing an occasional deadline is acceptable. Digital audio or multimedia systems fall in this category. VxWorks and QNX are well-known real-time operating systems.

- **Embedded Operating Systems:** A palmtop computer or **PDA (Personal Digital Assistant)** is a small computer that fits in a shirt pocket and performs a small number of functions such as an electronic address book and memo pad. Embedded systems run on the computers that control devices that are not generally thought of as computers, such as TV sets, microwave ovens, and mobile telephones. These often have some characteristics of real-time systems but also have size, memory, and power restrictions that make them special. Examples of such operating systems are Palm OS and Windows CE (Consumer Electronics).

- **Smart Card Operating Systems:** The smallest operating systems run on smart cards, which are credit card-sized devices containing a CPU chip. They have very severe processing power and memory constraints. Some of them can handle only a single function, such as electronic payments, but others can handle multiple functions on the same smart card. Some smart cards are Java oriented. What this means is that the ROM on the smart card holds an interpreter for the Java Virtual Machine (JVM). Java applets (small programs) are downloaded to the card and are interpreted by the JVM interpreter. Some of these cards can handle multiple Java applets at the same time, leading to multiprogramming and the need to schedule them. Resource management and protection also become an issue when two or more applets are present at the same time. These issues must be handled by the (usually extremely primitive) operating system present on the card.

Laboratory Works

Learn basic LINUX Commands

Navigation

Linux file systems are based on a directory tree. This means that you can create directories (or "folders") inside other directories, and files can exist in any directory.

- To see what directory you are currently active in:
  Pwd
  This stands for "print working directory", and will print the path to your current directory.
- To see other files and directories that exist in your current working directory:
  ls
  This will give you a list of names of files and directories.
- To navigate into a directory, use its name:
  cd <name of directory>
  This will change your new current working directory to the directory you specified.
- We can also create new directories in our current working directory. For example, to create a new directory called bar:
  mkdir bar
- We can also delete bar if we no longer find it useful:
  rm -d bar
- Will only delete empty directories.
  rm -d

File Manipulation

- We can view files. Say we have a file test in our current directory:
  cat test
  This will print out the entire contents of **test** to the terminal.
- With long files, this is impractical and unreadable. To paginate the output:
  less test

This will also print the contents of test, but one terminal page at a time, beginning at the start of the file. Use the spacebar to advance a page, or the arrow keys to go up and down one line at a time. Press q to quit out of less.

- To create a new file called csit:

  touch csit

  This creates an empty file with the name csit in your current working directory. The contents of this file are empty.
- It is also possible to copy a file to a new location. If we want to bring back csit, but keep bca too:

  cp bca csit
- To edit text into test:

  nano test

  This will open up a space where you can immediately start typing to edit test. To save the written text, press "Ctrl-X"
- To delete the empty cdcsit:

  rm cdcsit
- **sudo**, that's an essential yet potentially dangerous command. Whenever you're getting a Permission denied, Authorization failed or something like that use sudo.

  sudo touch csit
- **shoutdown –h now**: to power off immediately
- **shoutdown –h +10**: to shout down after 10 minutes
- **reboot**: to reboot the machine immediately

Using the Visual Editor

An editor is the program that is used to edit source code. There are many text editors available for Linux, but the two most widely used are Visual Editor Improved (VIM) and Emacs. Here we discuss VIM editor.

Creating and opening file

- **$ vi <filename>** - opens file if it is already exist, otherwise creates new file.
- **$ vi** - opens VIM editors, without filename.

Save Changes and Exit the Visual Editor (normal mode operation.)

- :w - write (save) the file.
- :q - quit if saved
- :wq - save and quit. Or :x or ZZ
- :wq filename - save file with filename.
- :q! - Quit without saving any changes.
- :e! - Abandon the changes and reload the last saved file

Compiling with GCC

A compiler turns human-readable code into machine readable object code that can actually run. The compilers of choice on Linux systems are all part of the GNU compiler collection, usually known as gcc. It supports the ANSI standard syntax for C. Compile, link and run the program.

- **$ gcc -o <exefile> <sourcefile>**
- **$ ./<exefile>**

Example

- $ gcc -o hello hello.c
- $ ./hello

Exercise

1. Explain why all modern systems have cache memory in addition to main memory.
2. Explain what an interrupt line is, and why it is used. Suggest which functional unit of the processor it connects to. Explain what happens to the processor's flow of execution when an interrupt line is asserted.
3. Explain the term "multiprogramming". Explain why a system which is not multi-programmed does not need secure multiplexing of resources. Suggest what functions an operating system might still be used for on such a system.
4. How does a microkernel differ from a conventional kernel? Briefly list the motivations and difficulties behind this.
5. What is a software interrupt? Explain why these might not necessarily be found in a monolithic operating system, but are always found in kernel-based systems?
6. Briefly explain the terms "process" and "context switch"
7. What is an Operating System? Explain their features.
8. Explain Brief History of Operating Systems.
9. What is system program? How it is differ from application programs? Explain
10. What is system call? Describe their importance.
11. Describe structure of OS with suitable example.
12. What is micro kernel? How it is differ from kernel? Explain
13. What is the use of kernel in OS? Explain
14. Describe two aspects of OS with suitable example.
15. How multi programmings differ from multi tasking? Explain
16. What is Distributed OS? How it is differ from real time OS? Explain.
17. What is batch OS? Describe their advantages and disadvantages.
18. What is multi processing system? Explain with suitable example.
19. What is program? How it is differ from software? Explain.
20. Describe basic commands used in LINUX.

❑❑❑

Chapter 2
PROCESS MANAGEMENT

1. Introduction to Process

A process is an instance of a program running in a computer. It is close in meaning to task, a term used in some operating systems. In UNIX and some other operating systems, a process is started when a program is initiated (either by a user entering a shell command or by another program). Like a task, a process is a running program with which a particular set of data is associated so that the process can be kept track of. An application that is being shared by multiple users will generally have one process at some stage of execution for each user. A process is basically a program in execution. In computing, a process is the instance of a computer program that is being executed. It contains the program code and its activity. Further A process is defined as an entity which represents the basic unit of work to be implemented in the system. A process can initiate a sub-process, which is a called a child process (and the initiating process is sometimes referred to as its parent). A child process is a replica of the parent process and shares some of its resources, but cannot exist if the parent is terminated. Processes can exchange information or synchronize their operation through several methods of inter process communication (IPC will be covered later).

To put it in simple terms, we write our computer programs in a text file and when we execute this program, it becomes a process which performs all the tasks mentioned in the program. When a program is loaded into the memory and it becomes a process, it can be divided into four sections stack, heap, text and data. The following image shows a simplified layout of a process inside main memory.

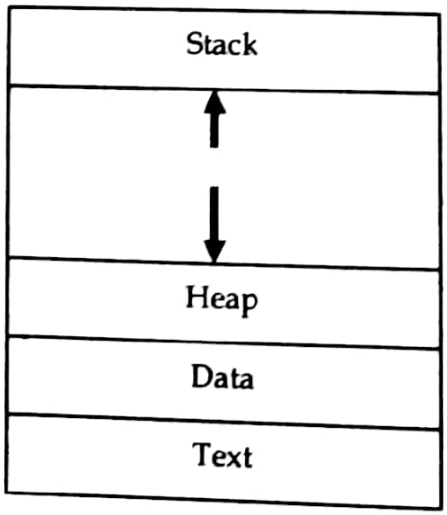


Fig 2.1: Layout of a process inside main memory

- **Stack:** The process Stack contains the temporary data such as method/function parameters return address and local variables.
- **Heap:** This is dynamically allocated memory to a process during its run time.
- **Text:** This includes the current activity represented by the value of Program Counter and the contents of the processor's registers.
- **Data:** This section contains the global and static variables.

2. Program

Program is an executable file containing the set of instructions written to perform a specific job on your computer. For example, chrome.exe is an executable file containing the set of instructions written so that we can view web pages. **notepad.exe** is an executable file containing the set of instructions which help us to edit and print the text files. Programs are not stored on the primary memory in your computer. They are stored on a disk or a secondary memory on your computer. They are read into the primary memory and executed by the kernel. A program is sometimes referred as passive entity as it resides on a secondary memory.

A program is a piece of code which may be a single line or millions of lines. A computer program is usually written by a computer programmer in a programming language. For example, here is a simple program written in C programming language

```
#include <stdio.h>
int main()
{
  printf("Hello, World! \n");
  return 0;
}
```

A computer program is a collection of instructions that performs a specific task when executed by a computer. When we compare a program with a process, we can conclude that a process is a dynamic instance of a computer program. A part of a computer program that performs a well-defined task is known as an algorithm. A collection of computer programs, libraries and related data are referred to as a software.

Some examples of computer programs:

- Operating system
- A web browser like Mozilla Firefox and Apple Safari can be used to view web pages on the Internet.
- An office suite can be used to write documents or spreadsheets.
- Video games are computer programs.

A computer program is written by a programmer. It is very difficult to write in the ones and zeroes of machine code, which is what the computer can read, so computer programmers write in a programming language, such as BASIC, C, or Java. Once it is written, the programmer uses a compiler to turn it into a language that the computer can understand.

There are also bad programs, called malware, written by people who want to do bad things to a computer. Some are spyware, trying to steal information from the computer. Some try to damage the data stored on the hard drive. Some others send users to web sites that offer to sell them things. Some are computer viruses or ransomware.

Differences between program and process

1. A program is a definite group of ordered operations that are to be performed. On the other hand, an instance of a program being executed is a process.
2. The nature of the program is passive as it does nothing until it gets executed whereas a process is dynamic or active in nature as it is an instance of executing program and performs the specific action.
3. A program has a longer lifespan because it is stored in the memory until it is not manually deleted while a process has a shorter and limited lifespan because it gets terminated after the completion of the task.
4. The resource requirement is much higher in case of a process; it could need processing, memory, I/O resources for the successful execution. In contrast, a program just requires memory for storage.

3. Process Model

In process model, all the runnable software on the computer is organized into a number of sequential processes. Each process has its own virtual Central Processing Unit (CPU).The real Central Processing Unit (CPU) switches back and forth from process to process. This work of switching back and forth is called multi-programming. A process is basically an activity. It has a program, input, output, and a state. The operating system must need a way to make sure that all the essential processes exist. There are the following four principal events that cause the processes to be created.

- System initialization
- Execution of a process creation system call by a running process
- A user request to create a new process
- Initiation of a batch work

Generally, there are some processes that are created whenever an operating system is booted. Some of those are foreground processes and others are background processes.

- Foreground process is the process that interacts with the computer users or computer programmers.
- Background processes have some specific functions.

In UNIX system, the ps program can be used to list all the running processes and in windows, the task manager is used to see what programs are currently running into the system. In addition to the processes that are created at the boot time, new processes can also be created. Sometime a running process will issue the system calls just to create one or more than one new processes to help it to do its work.

User can start a program just by typing the command of the program on the command prompt (CMD) or just by doing the double click on the icon of that program. When a process has been created, it starts running and does its work. The new process will terminate generally due to one of the following conditions, described in the table given below.

Condition	Description
Normal Exit	In Normal exit, process terminates because they have done their work successfully
Error Exit	In error exit, the termination of a process is done because of an error caused by the process, sometimes due to the program bug
Fatal Exit	In fatal exit, process terminates because it discovers a fatal error
Killed by other process	In this reason or condition, a process might also terminate due to that it executes a system call that tells the operating system (OS) just to kill some other process

In some computer systems when a process creates another process, then the parent process and child process continue to be associated in certain ways. The child process can itself creates more processes that form a process hierarchy.

4. Process Life Cycle

When a process executes, it passes through different states. These stages may differ in different operating systems, and the names of these states are also not standardized. The operating system maintains management information about a process in a process control block (PCB). Modern operating systems allow a process to be divided into multiple threads of execution, which share all process management information except for information directly related to execution. This information is held in a thread control block (TCB). Threads in a process can execute different parts of the program code at the same time. They can also execute the same parts of the code at the same time, but with different execution state:

- They have independent current instructions; that is, they have (or appear to have) independent program counters.
- They are working with different data; that is, they are (or appear to be) working with independent registers.

In general, a process can have one of the following five states at a time.

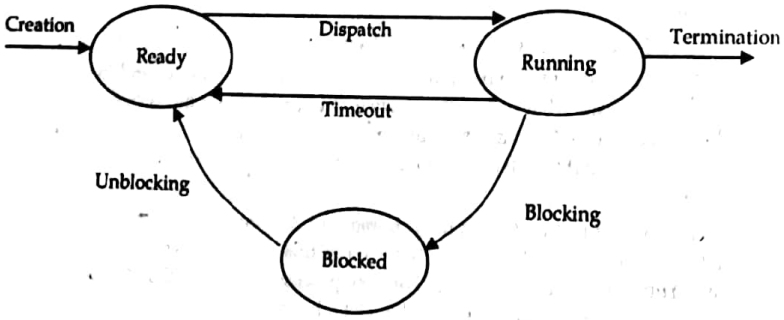


Fig 2.2: Process life cycle

1. **Start:** This is the initial state when a process is first started/created
2. **Ready:** The process is waiting to be assigned to a processor. Ready processes are waiting to have the processor allocated to them by the operating system so that they can run. Process may come into this state after Start state or while running it by but interrupted by the scheduler to assign CPU to some other process.
3. **Running:** Once the process has been assigned to a processor by the OS scheduler, the process state is set to running and the processor executes its instructions.
4. **Waiting:** Process moves into the waiting state if it needs to wait for a resource, such as waiting for user input, or waiting for a file to become available.
5. **Terminated or Exit:** Once the process finishes its execution, or it is terminated by the operating system, it is moved to the terminated state where it waits to be removed from main memory.

Suspended Processes

Characteristics of suspend process

- Suspended process is not immediately available for execution.
- The process may or may not be waiting on an event.
- For preventing the execution, process is suspended by OS, parent process, process itself and an agent.
- Process may not be removed from the suspended state until the agent orders the removal.

Note: Swapping is used to move all of a process from main memory to disk. When all the process by putting it in the suspended state and transferring it to disk.

Reasons for process suspension

- Swapping
- Timing
- Interactive user request
- Parent process request

Swapping

OS needs to release required main memory to bring in a process that is ready to execute.

Timing

Process may be suspended while waiting for the next time interval.

Interactive user request

Process may be suspended for debugging purpose by user.

Parent process request

To modify the suspended process or to coordinate the activity of various descendants

5. Process Control Block (PCB)

While creating a process the operating system performs several operations. To identify these processes, it must identify each process; hence it assigns a process identification number (PID) to each process. As the operating system supports multi-programming, it needs to keep track of all the processes. For this task, the process control block (PCB) is used to track the process's execution status. Each block of memory contains information about the process state, program counter, stack pointer, status of opened files, scheduling algorithms, etc. All these information is required and must be saved when the process is switched from one state to another. When the

process made transitions from one state to another, the operating system must update information in the process's PCB.

Role of Process Control Block

The role or work of process control block (PCB) in process management is that it can access or modified by most OS utilities including those are involved with memory, scheduling, and input / output resource access. It can be said that the set of the process control blocks give the information of the current state of the operating system. Data structuring for processes is often done in terms of process control blocks. For example, pointers to other process control blocks inside any process control block allows the creation of those queues of processes in various scheduling states. The various information that is contained by process control block is listed below:

- Naming the process
- State of the process
- Resources allocated to the process
- Memory allocated to the process
- Scheduling information
- Input / output devices associated with process

Components of PCB

A Process Control Block is a data structure maintained by the Operating System for every process. The PCB is identified by an integer process ID (PID). A PCB keeps all the information needed to keep track of a process as listed below:

1. **Process State:** As we know that the process state of any process can be New, running, waiting, executing, blocked, suspended, terminated. For more details regarding process states you can refer process management of an Operating System. Process control block is used to define the process state of any process. In other words, process control block refers the states of the processes.
2. **Process privileges:** This is required to allow/disallow access to system resources.
3. **Process ID:** In computer system there are various processes running simultaneously and each process has its unique ID. This Id helps system in scheduling the processes. This Id is provided by the process control block. In other words, it is an identification number that uniquely identifies the processes of computer system.
4. **Pointer:** A pointer to parent process.
5. **Program Counter:** Program Counter is a pointer to the address of the next instruction to be executed for this process.
6. **CPU registers:** This information is comprising with the various registers, such as index and stack that are associated with the process. This information is also managed by the process control block.
7. **CPU Scheduling Information:** Scheduling information is used to set the priority of different processes. This is very useful information which is set by the process control block. In computer system there were many processes running simultaneously and each process have its priority. The priority of primary feature of RAM is higher than other secondary features. Scheduling information is very useful in managing any computer system.

8. **Memory management information:** This includes the information of page table, memory limits, Segment table depending on memory used by the operating system.
9. **Accounting information:** This includes the amount of CPU used for process execution, time limits, execution ID etc.
10. **IO status information:** This includes a list of I/O devices allocated to the process.

Since PCB contains the critical information for the process, it must be kept in an area of memory protected from normal user access. In some operating systems the PCB is placed in the beginning of the kernel stack of the process as it is a convenient protected location. The architecture of a PCB is completely dependent on Operating System and may contain different information in different operating systems. The PCB is maintained for a process throughout its lifetime, and is deleted once the process terminates. Here is a simplified diagram of a PCB;

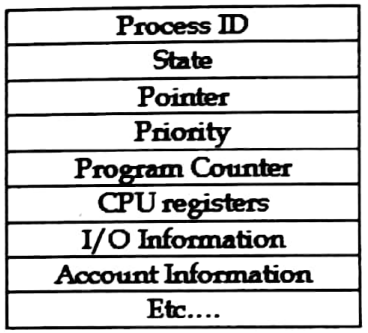


Fig 2.3: PCB diagram

6. Multitasking

Multitasking is the ability of an operating system to execute more than one task simultaneously on a single processor machine. Though we say so but in reality no two tasks on a single processor machine can be executed at the same time. Actually CPU switches from one task to the next task so quickly that appears as if all the tasks are executing at the same time. More than one task/program/job/process can reside into the same CPU at one point of time.

Multitasking is a method to allow multiple processes to share processors (CPUs) and other system resources. Each CPU executes a single task at a time. However, multitasking allows each processor to switch between tasks that are being executed without having to wait for each task to finish. Depending on the operating system implementation, switches could be performed when tasks perform input/output operations, when a task indicates that it can be switched, or on hardware interrupts.

A common form of multitasking is time-sharing. Time-sharing is a method to allow high responsiveness for interactive user applications. In time-sharing systems, context switches are performed rapidly, which makes it seem like multiple processes are being executed simultaneously on the same processor. This seeming execution of multiple processes simultaneously is called concurrency. For security and reliability, most modern operating systems prevent direct communication between independent processes, providing strictly mediated and controlled inter-process communication functionality.

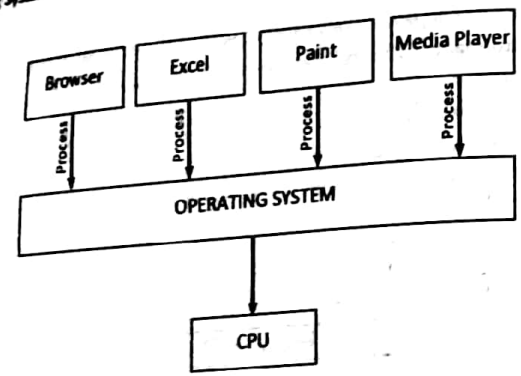


Fig 2.4: Multitasking OS

7. Multiprogramming

Multiprogramming is also the ability of an operating system to execute more than one program on a single processor machine. More than one task/program/job/process can reside into the main memory at one point of time. Computer runnings excel and Firefox browser simultaneously is an example of multiprogramming. In a multiprogramming system there are one or more programs loaded in main memory which are ready to execute. Only one program at a time is able to get the CPU for executing its instructions (i.e., there is at most one process running on the system) while all the others are waiting their turn.

The main idea of multiprogramming is to maximize the use of CPU time. Indeed, suppose the currently running process is performing an I/O task (which, by definition, does not need the CPU to be accomplished). Then, the OS may interrupt that process and give the control to one of the other in-main-memory programs that are ready to execute (i.e. process context switching). In this way, no CPU time is wasted by the system waiting for the I/O task to be completed, and a running process keeps executing until either it voluntarily releases the CPU or when it blocks for an I/O operation. Therefore, the ultimate goal of multiprogramming is to keep the CPU busy as long as there are processes ready to execute.

Note that in order for such a system to function properly, the OS must be able to load multiple programs into separate areas of the main memory and provide the required protection to avoid the chance of one process being modified by another one. Other problems that need to be addressed when having multiple programs in memory is fragmentation as programs enter or leave the main memory. Another issue that needs to be handled as well is that large programs may not fit at once in memory which can be solved by using pagination and virtual memory. Finally, note that if there are N ready processes and all of those are highly CPU-bound (i.e., they mostly execute CPU tasks and none or very few I/O operations), in the very worst case one program might wait all the other N-1 ones to complete before executing.

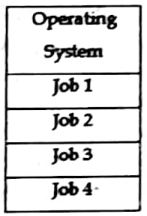


Fig 2.5: Batch of multiple jobs

8. Multiprocessing

Multiprocessing is the ability of an operating system to execute more than one process simultaneously on a multi processor machine. In this, a computer uses more than one CPU at a time.

Multiprocessing sometimes refers to executing multiple processes (programs) at the same time. This might be misleading because we have already introduced the term "multiprogramming" to describe that before. In fact, multiprocessing refers to the hardware (i.e., the CPU units) rather than the software (i.e., running processes). If the underlying hardware provides more than one processor, then that is multiprocessing. Several variations on the basic scheme exist, e.g., multiple cores on one die or multiple dies in one package or multiple packages in one system.

Anyway, a system can be both multi-programmed by having multiple programs running at the same time and multiprocessing by having more than one physical processor.

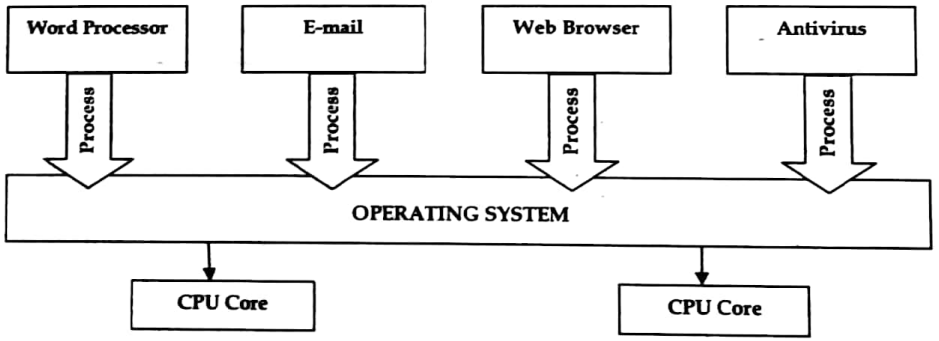


Fig 2.6: Multi-processing system

9. Threads

Despite of the fact that a thread must execute in process, the process and its associated threads are different concept. Processes are used to group resources together and threads are the entities scheduled for execution on the CPU.

A thread is a single sequence stream within in a process. Because threads have some of the properties of processes, they are sometimes called lightweight processes. In a process, threads allow multiple executions of streams. In many respect, threads are popular way to improve application through parallelism. The CPU switches rapidly back and forth among the threads giving illusion that the threads are running in parallel. Like a traditional process i.e., process with one thread, a thread can be in any of several states (Running, Blocked, Ready or terminated).

Each thread has its own stack. Since thread will generally call different procedures and thus a different execution history. This is why thread needs its own stack. 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 shares with other threads their code section, data section, OS resources also known as task, such as open files and signals.

Traditional (heavyweight) processes have a single thread of control - There is one program counter, and one sequence of instructions that can be carried out at any given time, as shown in figure below multi-threaded applications have multiple threads within a single process, each having their own program counter, stack and set of registers, but sharing common code, data and certain structures such as open files.

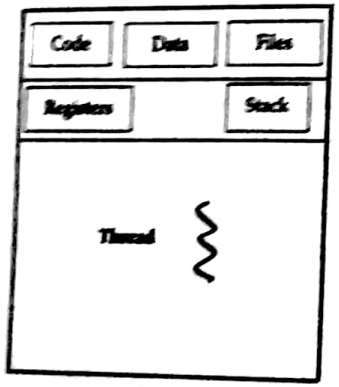


Fig 2.7: Single threaded process

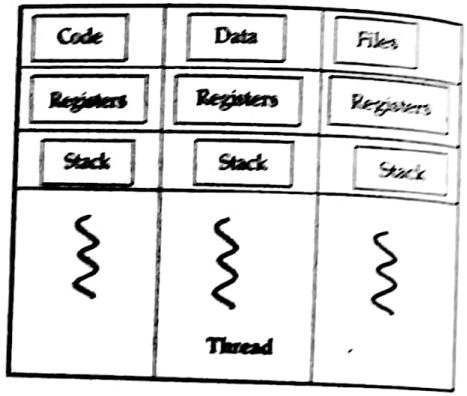


Fig 2.8: Multi threaded process

Threads are very useful in modern programming whenever a process has multiple tasks to perform independently of the others. This is particularly true when one of the tasks may block, and it is desired to allow the other tasks to proceed without blocking. For example, in a word processor, a background thread may check spelling and grammar while a foreground thread processes user input (keystrokes), while yet a third thread loads images from the hard drive, and a fourth does periodic automatic backups of the file being edited.

Another example is a web server - Multiple threads allow for multiple requests to be satisfied simultaneously, without having to service requests sequentially or to fork off separate processes for every incoming request. (The latter is how this sort of thing was done before the concept of threads was developed. A daemon would listen at a port, fork off a child for every incoming request to be processed, and then go back to listening to the port.)

There are four major benefits to multi-threading:

- **Responsiveness** - One thread may provide rapid response while other threads are blocked or slowed down doing intensive calculations.
- **Resources haring** - By default threads share common code, data, and other resources, which allows multiple tasks to be performed simultaneously in a single address space.
- **Economy** - Creating and managing threads (and context switches between them) is much faster than performing the same tasks for processes.

- **Scalability, i.e. Utilization of multiprocessor architectures** - A single threaded process can only run on one CPU, no matter how many may be available, whereas the execution of a multi-threaded application may be split amongst available processors. (Note that single threaded processes can still benefit from multi-processor architectures when there are multiple processes contending for the CPU, i.e. when the load average is above some certain threshold.)

Properties of a Thread

- Only one system call can create more than one thread (Lightweight process).
- Threads share data and information.
- Threads shares instruction, global and heap regions but have its own individual stack and registers.
- Thread management consumes no or fewer system calls as the communication between threads can be achieved using shared memory.
- The isolation property of the process increases its overhead in terms of resource consumption.

10. The Thread Model

There are two types of threads to be managed in a modern system: User threads and kernel threads. User threads are supported above the kernel, without kernel support. These are the threads that application programmers would put into their programs. Kernel threads are supported within the kernel of the OS itself. All modern operating systems support kernel level threads, allowing the kernel to perform multiple simultaneous tasks and/or to service multiple kernel system calls simultaneously. In a specific implementation, the user threads must be mapped to kernel threads, using one of the following strategies.

- **Many-To-One Model:** This model maps many user level threads to one kernel level thread. Thread management is done by thread library in user space, so it is efficient. However, if a blocking system call is made, then the entire process blocks, even if the other user threads would otherwise be able to continue. Because a single kernel thread can operate only on a single CPU, the many-to-one model does not allow individual processes to be split across multiple CPUs. Green threads for Solaris and GNU Portable Threads implement the many-to-one model in the past, but few systems continue to do so today.

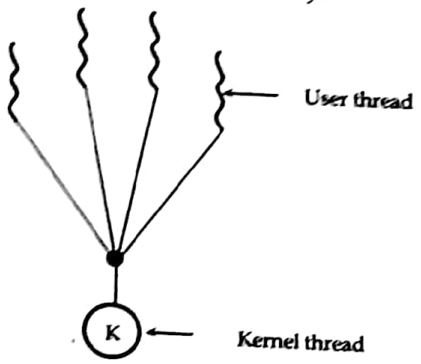


Fig 2.9: many to one thread model

- **One-To-One Model:** One user thread is mapped to one kernel thread. It provides more concurrency than previous model by allowing another thread to run during blocking. It also allows parallelism. The only overhead is for each threads corresponding kernel thread should be created. Most implementations of this model place a limit on how many threads can be created. Linux and Windows from 95 to XP implement the one-to-one model for threads.

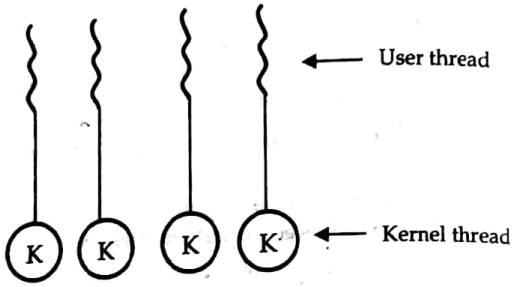


Fig 2.10: one to one thread model

- **Many-To-Many Model:** In this model, many user level threads multiplex to the Kernel thread of smaller or equal numbers. The number of Kernel threads may be specific to either a particular application or a particular machine. Users have no restrictions on the number of threads created. Blocking kernel system calls do not block the entire process. Processes can be split across multiple processors. Individual processes may be allocated variable numbers of kernel threads, depending on the number of CPUs present and other factors.

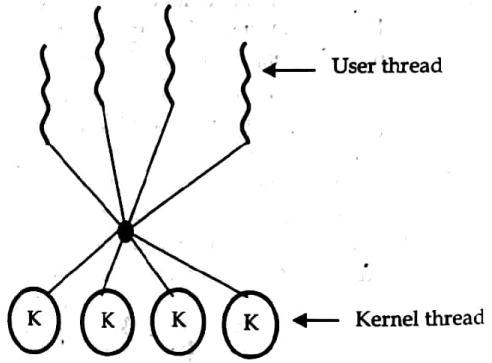


Fig 2.11: Many-to-many thread model

Implementing Threads in User Space

The first method is to put the threads package entirely in user space. The kernel knows nothing about them. As far as the kernel is concerned, it is managing ordinary, single-threaded processes. The first, and most obvious, advantage is that a user-level threads package can be implemented on an operating system that does not support threads. All operating systems used to fall into this category, and even now some still do.

The threads run on top of a run-time system, which is a collection of procedures that manage threads. When threads are managed in user space, each process needs its own private thread table. It is analogous to the kernel's process table, except that it keeps track only of the per-thread

properties such each thread's program counter stack pointer, registers, state, etc. The thread table is managed by the runtime system. When a thread is moved to ready state or blocked state, the information needed to restart it is stored in the thread table, exactly the same way as the kernel stores information about processes in the process table.

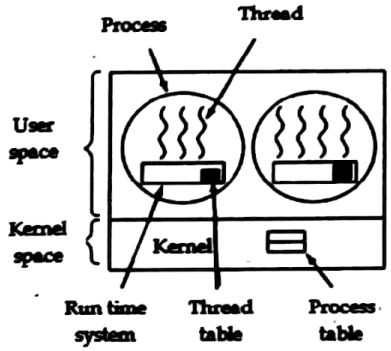


Fig 2.12: A user-level threads package.

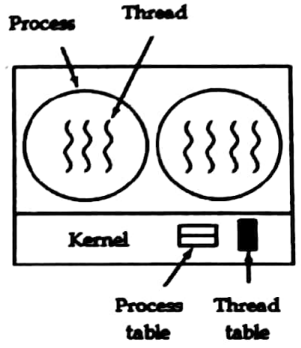


Fig 2.13: A threads package managed by the kernel.

Thread vs. process

As we mentioned earlier that in many respect threads operate in the same way as that of processes. Some of the similarities and differences are:

Similarities

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

Differences

- All threads of a program are logically contained within a process.
- A process is heavy weighted, but a thread is light weighted.
- A program is an isolated execution unit whereas thread is not isolated and shares memory.
- A thread cannot have an individual existence; it is attached to a process. On the other hand, a process can exist individually.
- At the time of expiration of a thread, its associated stack could be recovered as every thread has its own stack. In contrast, if a process dies, all threads die including the process.

Kernel-Level Threads

To make concurrency cheaper, the execution aspect of process is separated out into threads. As such, the OS now manages threads and processes. All thread operations are implemented in the kernel and the OS schedules all threads in the system. OS managed threads are called kernel-level threads or light weight processes.

In this method, 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 ma threads.

Advantages

- Because kernel has full knowledge of all threads, Scheduler may decide to give more tim 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.

Disadvantages

- The kernel-level threads are slow and inefficient. For instance, threads operations a 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 thre control block (TCB) for each thread to maintain information about threads. As a result, th is significant overhead and increased in kernel complexity.

User-Level Threads

Kernel-Level threads make concurrency much cheaper than process because, much less state allocate and initialize. However, for fine-grained concurrency, kernel-level threads still suff from too much overhead. Thread operations still require system calls. Ideally, we require thre operations to be as fast as a procedure call. Kernel-Level threads have to be general to support th needs of all programmers, languages, runtimes, etc. For such fine grained concurrency, we nee still "cheaper" threads.

To make threads cheap and fast, they need to be implemented at user level. User-Level threa are managed entirely by the run-time system (user-level library). The kernel knows nothing abo user-level threads and manages them as if they were single-threaded processes. User-Leve threads are small and fast, each thread is represented by a PC, register, stack, and small threa control block. Creating a new thread, switching between threads, and synchronizing threads ar done via procedure call i.e. no kernel involvement. User-Level threads are hundred times faste than Kernel-Level threads.

Advantages

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

- User-level threads do 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 require non-blocking systems call i.e., a multithreaded kernel. Otherwise entire process will have blocked in the kernel, even if there are runnable threads left in the processes. For example, if one thread causes a page fault, the process blocks.

11. Inter Process Communication

Inter-process communication (IPC) is a mechanism that allows the exchange of data between processes. By providing a user with a set of programming interfaces, IPC helps a programmer organize the activities among different processes. IPC allows one application to control another application, thereby enabling data sharing without interference. IPC enables data communication by allowing processes to use segments, semaphores, and other methods to share memory and information. IPC facilitates efficient message transfer between processes. The idea of IPC is based on Task Control Architecture (TCA). It is a flexible technique that can send and receive variable length arrays, data structures, and lists. It has the capability of using publish/subscribe and client/server data-transfer paradigms while supporting a wide range of operating systems and languages.

The working principle of FIFO is very similar to that of pipes. The data flow in FIFO is unidirectional and is identified by access points. The difference between the two is that FIFO is identified by an access point, which is a file within the file system, whereas pipes are identified by an access point.

For example, in a shell pipeline, the output of the first process must be passed to the second process, and so on down the line. Thus there is a need for communication between processes, preferably in a well-structured way not using interrupts. Very briefly, there are three issues here.

- How one process can pass information to another?
- Making sure two or more processes do not get into each other's way when engaging in critical activities.
- Proper sequencing when dependencies are present.

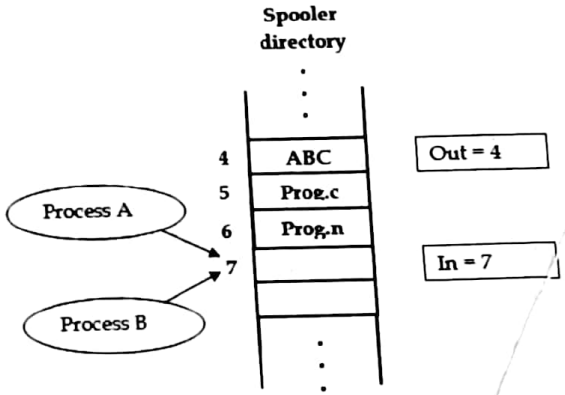


Fig 2.14: IPC with race condition.

12. Race condition

A race condition occurs when two or more threads can access shared data and they try to change it at the same time. Because the thread scheduling algorithm can swap between threads at any time, you don't know the order in which the threads will attempt to access the shared data. Therefore, the result of the change in data is dependent on the thread scheduling algorithm, i.e. both threads are "racing" to access/change the data.

Problems often occur when one thread does a "check-then-act" (e.g. "check" if the value is X, then "act" to do something that depends on the value being X) and another thread does something to the value in between the "check" and the "act". E.g.:

```
if (x == 5) // The "Check"
{
  y = x * 2; // the "Act"
  // If another thread changed x in between "if (x == 5)" and "y = x * 2" above,
  // y will not be equal to 10.
}
```

The point being, y could be 10, or it could be anything, depending on whether another thread changed x in between the check and act. We have no real way of knowing. In order to prevent race conditions from occurring, you would typically put a lock around the shared data to ensure only one thread can access the data at a time. This would mean something like this:

```
// obtain lock for x
if (x == 5)
{
  y = x * 2; // Now, nothing can change x until the lock is released.
       // Therefore y = 10
}
// release lock for x
```

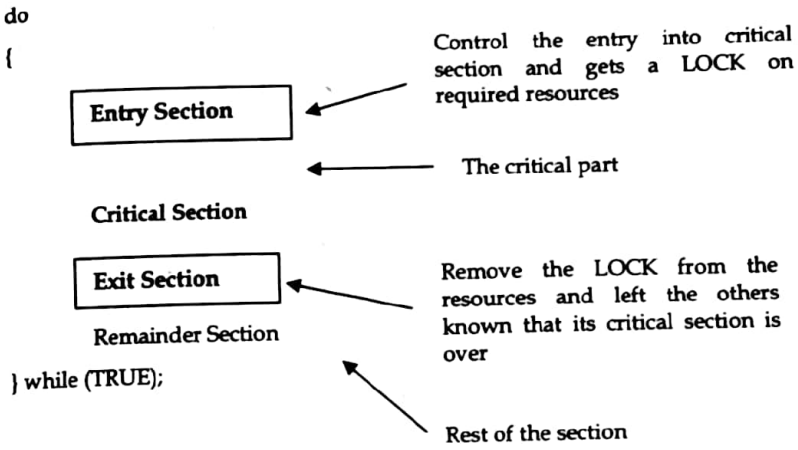
Therefore, Operating System Concerns of following things

1. The operating system must be able to keep track of the various processes
2. The operating system must allocate and de-allocate various resources for each active process.
3. The operating system must protect the data and physical resources of each process against unintended interference by other processes.
4. The functioning of a process, and the output it produces, must be independent of the speed at which its execution is carried out relative to the speed of other concurrent processes.

Two or more processes need to access a resource during the course of their execution results in conflict. Each process is unaware of the existence of the other processes. There is no exchange of information between the competing processes.

13. Critical section

How do we avoid race conditions? One way to prevent two or more process, using the shared data is mutual exclusion i.e. some way of making sure that if one process is using a shared variable or file, the process will be excluded from doing same thing. Sometimes a process has to access shared memory or files that can lead to races. That part of the program where the shared memory is accessed is called the critical region or critical section. If we could arrange matters such that no two processes were ever in their critical regions at the same time, we could avoid races. The critical section is given as follows:



In the above diagram, the entry sections handle the entry into the critical section. It acquires the resources needed for execution by the process. The exit section handles the exit from the critical section. It releases the resources and also informs the other processes that critical section is free.

The critical section problem needs a solution to synchronize the different processes. The solution to the critical section problem must satisfy the following conditions:

1. **Mutual Exclusion:** Mutual exclusion implies that only one process can be inside the critical section at any time. If any other processes require the critical section, they must wait until it is free.
2. **Progress:** Progress means that if a process is not using the critical section, then it should not stop any other process from accessing it. In other words, any process can enter a critical section if it is free.
3. **Bounded Waiting:** Bounded waiting means that each process must have a limited waiting time. It should not wait endlessly to access the critical section.

14. Synchronization Hardware

Many systems provide hardware support for critical section code. The critical section problem could be solved easily in a single-processor environment if we could disallow interrupts to occur while a shared variable or resource is being modified. In this manner, we could be sure that the current sequence of instructions would be allowed to execute in order without pre-emption. Unfortunately, this solution is not feasible in a multiprocessor environment. Disabling interrupt on a multiprocessor environment can be time consuming as the message is passed to all the processors. This message transmission lag, delays entry of threads into critical section and the system efficiency decreases.

Implementing mutual exclusion: A mutual exclusion (mutex) is a program object that prevents simultaneous access to a shared resource. This concept is used in concurrent programming with a critical section, a piece of code in which processes or threads access a shared resource. Only one thread owns the mutex at a time, thus a mutex with a unique name is created when a program

starts. When a thread holds a resource, it has to lock the mutex from other threads to prev
concurrent access of the resource. Upon releasing the resource, the thread unlocks the mutex.
Mutex comes into the picture when two threads work on the same data at the same time. It acts
a lock and is the most basic synchronization tool. When a thread tries to acquire a mutex, it gai
the mutex if it is available; otherwise the thread is set to sleep condition. Mutual exclusio
reduces latency and busy-waits using queuing and context switches. Mutex can be enforced
both the hardware and software levels.

15. Disabling Interrupts

The simplest solution is to have each process disable all interrupts just after entering its critica
region and re-enable them just before leaving it. With interrupts disabled, no clock interrupts ca
occur. The CPU is only switched from process to process as a result of clock or other interrupt
after all, and with interrupts turned off the CPU will not be switched to another process.
This approach is generally unattractive because it is unwise to give user processes the power t
turn off interrupts. Suppose that one of them did it and never turned them on again? That coul
be the end of the system.

16 Lock variables

A lock variable provides the simplest synchronization mechanism for processes. Some
noteworthy points regarding Lock Variables are-

1. It's a software mechanism implemented in user mode, i.e. no support required from the
   Operating System.
2. It's a busy waiting solution (keeps the CPU busy even when its technically waiting).
3. It can be used for more than two processes.

The pseudo code looks something like this:

```
Entry section – while (lock != 0);
                Lock = 1;
//critical section
Exit section - Lock = 0;
```

Consider having a single, shared (lock) variable, initially 0. When a process wants to enter its
critical region, it first tests the lock.

1. If lock = 0, the process sets it to 1 and enters the critical region.
2. If the lock is already 1, the process just waits until it becomes 0. Thus, a 0 means that no
   process is in its critical region.

Problem: Suppose that one process reads the lock and sees that it is 0. Before it can set the lock to
1, another process is scheduled, runs, and sets the lock to 1. When the first process runs again, it
will also set the lock to 1, and two processes will be in their critical regions at the same time.

17. Turn Variable or Strict alternation

Turn Variable or Strict Alternation Approach is the software mechanism implemented at user
mode. It is a busy waiting solution which can be implemented only for two processes. In this
approach, A turn variable is used which is actually a lock.

This approach can only be used for only two processes. In general, let the two processes be Pi and
Pj. They share a variable called turn variable. The pseudo code of the program can be given as
following.

For Process Pi

```
No - CS
while (turn ! = i);
Critical Section
turn = j;
No - CS
```

For Process Pj

```
No- CS
while (turn ! = j);
Critical Section
turn = i ;
No- CS
```

The actual problem of the lock variable approach was the fact that the process was entering in the
critical section only when the lock variable is 1. More than one process could see the lock variable
as 1 at the same time hence the mutual exclusion was not guaranteed there.

- This problem is addressed in the turn variable approach. Now, a process can enter in the
  critical section only in the case when the value of the turn variable equal to the PID of the
  process.
- There are only two values possible for turn variable, i or j. if its value is not i then it will
  definitely be j or vice versa.
- In the entry section, in general, the process Pi will not enter in the critical section until its
  value is j or the process Pj will not enter in the critical section until its value is i.

Continuously testing a variable until some value appears is called busy waiting. It should usually
be avoided, since it wastes CPU time. Only when there is a reasonable expectation that the wait
will be short is busy waiting used. A lock that uses busy waiting is called a spin lock.

18. Peterson's solution

By combining the idea of taking turns with the idea of lock variables and warning variables, a
Dutch mathematician, T. Dekker, was the first one to devise a software solution to the mutual
exclusion problem that does not require strict alternation.

```
#define FALSE 0
#define TRUE 1
#define N 2 /* number of processes */
int turn; /* whose turn is it? */
int interested[N]; /* all values initially 0 (FALSE) */
voidenter_region(int process) /* process is 0 or 1 */
{
        int other; /* number of the other process */
        other = 1 – process; /* the opposite of process */
```

```
        interested[process] = TRUE; /* show that you are interested */
        turn = process; /* set flag */
        while (turn == process && interested[other] == TRUE) /* null statement */;
}
voidleave_region (int process) /* process, who is leaving */
{
        interested[process] = FALSE; // indicate departure from critical region
}
```

Let us see how this solution works. Initially neither process is in its critical region. Now process
calls enter_region. It indicates its interest by setting its array element and sets turn to 0. Sinc
process 1 is not interested, enter_regionreturns immediately. If process 1 now calls enter_regio
it will hang there until interested[0] goes to FALSE, an event that only happens when process
calls leave_regionto exit the critical region.

Problem: Difficult to program for n processes, may lead to starvation.

19. Test and set lock

Now let us look at a proposal that requires a little help from the hardware. Many computers
especially those designed with multiple processors in mind, have an instruction

TSL RX, LOCK

It reads the contents of the memory word lock into register RX and then stores a nonzero value a
the memory address lock. The operations of reading the word and storing into it are guaranteed
to be indivisible—no other processor can access the memory word until the instruction is
finished. The CPU executing the TSL instruction locks the memory bus to prohibit other CPUs
from accessing memory until it is done.

To use the TSL instruction, we will use a shared variable, lock, to coordinate access to shared
memory. When lock is 0, any process may set it to 1 using the TSL instruction and then read or
write the shared memory. When it is done, the process sets lock back to 0 using an ordinary move
instruction.

enter_region:

```
TSL REGISTER,LOCK | copy lock to register and set lock to 1
CMP REGISTER,#0 | was lock zero?
JNE enter_region | if it was non zero, lock was set, so loop
RET | return to caller; critical region entered
leave_region:
MOVE LOCK,#0 | store a 0 in lock
RET | return to caller
```

One solution to the critical region problem is now straightforward. Before entering its critical
region, a process calls enter_region, which does busy waiting until the lock is free; then it
acquires the lock and returns. After the critical region the process calls leave_region, which stores
a 0 in lock. As with all solutions based on critical regions, the processes must call enter_regionand
leave_regionat the correct times for the method to work. If a process cheats, the mutual exclusion
will fail.

Alternative to busy waiting

Busy wait causes wastage of CPU time.

20. Sleep and Wakeup

Both Peterson's solution and the solution using TSL are correct, but both have the defect of requiring busy waiting. Sleep is the system call that causes the caller to block, that is suspended until another process wakes it up. The wakeup call has one parameter, the process to be awakened.

Consider a computer with two processes, H, with high priority and L, with low priority. The scheduling rules are such that H is run whenever it is in ready state. At a certain moment, with L in its critical region, H becomes ready to run (e.g., an I/O operation completes). H now begins busy waiting, but since L is never scheduled while H is running, L never gets the chance to leave its critical region, so H loops forever. This situation is sometimes referred to as the priority inversion problem.

21. Semaphore

In 1965, Dijkstra proposed a new and very significant technique for managing concurrent processes by using the value of a simple integer variable to synchronize the progress of interacting processes. This integer variable is called semaphore. So it is basically a synchronizing tool and is accessed only through two low standard atomic operations, wait and signal designated by P(S) and V(S) respectively.

In very simple words, semaphore is a variable which can hold only a non-negative Integer value, shared between all the threads, with operations wait and signal, which work as follow:

P(S): if S ≥ 1 then S = S - 1
    Else <block and enqueue the process>
V(S): if <some process is blocked on the queue>
    Then <unblock a process>
    Else S = S + 1;

The classical definitions of wait and signal are:

- **Wait:** Decrements the value of its argument S, as soon as it would become non-negative (greater than or equal to 1)
- **Signal:** Increments the value of its argument S, as there is no more process blocked on the queue

Properties of Semaphores

- It's simple and always has a non-negative Integer value.
- Semaphores are machine independent.
- Semaphores are simple to implement.
- Works with many processes.
- Can have many different critical sections with different semaphores.
- Each critical section has unique access semaphores.
- Can permit multiple processes into the critical section at once, if desirable.

Use of Semaphores

- For achieving mutual exclusion
- To solve the synchronization problems

Types of Semaphores

Semaphores are mainly of two types:

1. **Binary Semaphore:** It is a special form of semaphore used for implementing mutual exclusion; hence it is often called a Mutex. A binary semaphore is initialized to 1 and only takes the values 0 and 1 during execution of a program.
2. **Counting Semaphores:** These are used to implement bounded concurrency.

Drawback of Semaphore

- They are essentially shared global variables
- Access to semaphores can come from anywhere in a program
- There is no control or guarantee of proper usage
- There is no linguistic connection between the semaphore and the data to which the semaphore controls access
- They serve two purposes, mutual exclusion and scheduling constraints

22. Monitors

Monitors are a synchronization construct that were created to overcome the problems caused by semaphores such as timing errors. Monitors are abstract data types and contain shared data variables and procedures. The shared data variables cannot be directly accessed by a process and procedures are required to allow a single process to access the shared data variables at a time. This is demonstrated as follows:

```
Monitor monitorName
{
        data variables;
        Condition variables;
        Procedure P1 {.....}
        Procedure P2.{.....}
        Procedure Pn {.....}
}
```

Condition Variables

Two different operations are performed on the condition variables of the monitor.

Wait.

Signal.

Let say we have 2 condition variables

```
Condition x, y; //Declaring variable
```

Wait operation

x.wait() : Process performing wait operation on any condition variable are suspended. The suspended processes are placed in block queue of that condition variable.

Note: Each condition variable has its unique block queue.

Signal operation

x.signal(): When a process performs signal operation on condition variable, one of the blocked processes is given chance.

```
If (x block queue empty)
        // ignore signal
else
        // Resume a process from block queue.
```

Only one process can be active in a monitor at a time. Other processes that need to access the shared variables in a monitor have to line up in a queue and are only provided access when the previous processes release the shared variables.

23. Message passing

It is the mechanism for processes to communicate and to synchronize their actions. Message system – processes communicate with each other without resorting to shared variables. This method of inter-process communication uses two primitives, send and receive.

Send (destination, &message);

And

Receive (source, &message);

If P and Q wish to communicate, they need to:

- establish a communication link between them
- exchange messages via send/receive
- Implementation of communication link
  - ✓ physical (e.g., shared memory, hardware bus)
  - ✓ logical (e.g., logical properties)

Mode of communication between two processes can take place through two methods

- Direct Addressing
- Indirect Addressing

Direct Communication

With direct communication, 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:

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

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

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. Exactly one link exists between each pair of processes. This scheme exhibits symmetry in addressing; that is, both the sender and the receiver processes must name the 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 symmetric and asymmetric schemes 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.

Indirect Communication

With indirect communication, the messages are sent to and received from mailboxes, or 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 they share a mailbox. The send and receive primitives are defined as follows:

- Send (A, message) - Send a message to mailbox A.
- Receive (A, message) - Receive 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 both members of the pair have a shared mailbox. A link may be associated with more than two processes. A number of different links may exist between each pair of communicating processes, with each link corresponding to one mailbox.

Now suppose that processes PI, P2, and P3 all share mailbox A. Process P1 sends a message to A, while P2 and P3 each execute a receive from A. Which process will receive the message sent by P1? The answer depends on the scheme that we choose: 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 P2 or P3, 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 operating system. If the mailbox is owned by a process (that is, the mailbox is part of the address space of the process), then we distinguish between the owner (who can only receive messages through this mailbox) and the user (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. On the other hand, a mailbox owned by the operating system is independent and is not attached to any particular process. The operating system then must provide a mechanism that allows a process to do the following: Create a new mailbox. Send and receive messages through the mailbox. Delete 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.

24. The Producer-Consumer Problem with Message Passing

Now let us see how the producer-consumer problem can be solved with message passing. Assume that all messages are the same size and that messages sent but not yet received are buffered automatically by the operating system. In this solution, a total of N messages is used, analogous to the N slots in a shared memory buffer. The consumer starts out by sending N empty messages to the producer. Whenever the producer has an item to give to the consumer, it takes an empty message and sends back a full one. In this way, the total number of messages in the system remains constant in time, so they can be stored in a given amount of memory known in advance.

If the producer works faster than the consumer, all the messages will end up full, waiting for the consumer: the producer will be blocked, waiting for an empty to come back. If the consumer works faster, then the reverse happens: all the messages will be empties waiting for the producer to fill them up: the consumer will be blocked, waiting for a full message.

```
#define N 100 /* number of slots in the buffer */
void producer(void)
{
        int item;
        message m; /* message buffer */
        while (TRUE)
        {
                item = produce_item( ); /* generate something to put in buffer */
                receive(consumer, &m); /* wait for an empty to arrive */
                build_message (&m, item); /* construct a message to send */
                send(consumer, &m); /* send item to consumer */
        }
}
void consumer(void)
{
        int item, i;
        message m;
        for (i = 0; i < N; i++) send(producer, &m); /* send N empties */
        while (TRUE)
        {
                receive(producer, &m); /* get message containing item */
                item = extract_item(&m); /* extract item from message */
                send(producer, &m); /* send back empty reply */
                consume_item(tem); /* do something with the item */
        }
}
```

The producer-consumer problem with N messages

25. Classical IPC Problems

The operating systems literature is full of interesting problems that have been widely discussed and analyzed using a variety of synchronization methods. In the following sections we will examine three of the better-known problems.

25.1 Producer consumer

In computing, the producer–consumer problem (also known as the bounded-buffer problem) is a classic example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue. The producer's job is to generate data, put it into the buffer, and start again. At the same time, the consumer is consuming the data (i.e., removing it from the buffer), one piece at a time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

The solution for the producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same way, the consumer can go to sleep if it finds the buffer empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer. The solution can be reached by means of inter-process communication, typically using semaphores. An inadequate solution could result in a deadlock where both processes are waiting to be awakened. The problem can also be generalized to have multiple producers and consumers.

In the solution two library routines are used, sleep and wakeup. When sleep is called, the caller is blocked until another process wakes it up by using the wakeup routine. The global variable itemCount holds the number of items in the buffer.

```
intitemCount=0;
procedureproducer()
{
        while(true)
        {
                item=produceItem();
                if(itemCount==BUFFER_SIZE)
                {
                        sleep();
                }
                putItemIntoBuffer(item);
                itemCount=itemCount+1;

                if(itemCount==1)
                {
                        wakeup(consumer);
                }
        }
}
procedureconsumer()
{
```



```
        while(true)
        {
                if(itemCount==0)
                {
                        sleep();
                }
                item=removeItemFromBuffer();
                itemCount=itemCount-1;
                if(itemCount==BUFFER_SIZE-1)
                {
                        wakeup(producer);
                }
                consumeItem(item);
        }
}
```

The problem with this solution is that it contains a race condition that can lead to a deadlock. Consider the following scenario:

1. The consumer has just read the variable item Count, noticed its zero and is just about to move inside the if block.
2. Just before calling sleep, the consumer is interrupted and the producer is resumed.
3. The producer creates an item, puts it into the buffer, and increases item Count.
4. Because the buffer was empty prior to the last addition, the producer tries to wake up the consumer.
5. Unfortunately, the consumer wasn't yet sleeping, and the wakeup call is lost. When the consumer resumes, it goes to sleep and will never be awakened again. This is because the consumer is only awakened by the producer when item Count is equal to 1.
6. The producer will loop until the buffer is full, after which it will also go to sleep.
7. Since both processes will sleep forever, we have run into a deadlock. This solution therefore is unsatisfactory.

25.2 Sleeping barber

Customers arrive to a barber, if there are no customers the barber sleeps in his chair. If the barber is a sleep then the customers must wake him up. The analogy is based upon a hypothetical barber shop with one barber. The barber has one barber chair and a waiting room with a number of chairs in it. When the barber finishes cutting a customer's hair, he dismisses the customer and then goes to the waiting room to see if there are other customers waiting. If there are, he brings one of them back to the chair and cuts his hair. If there are no other customers waiting, he returns to his chair and sleeps in it.

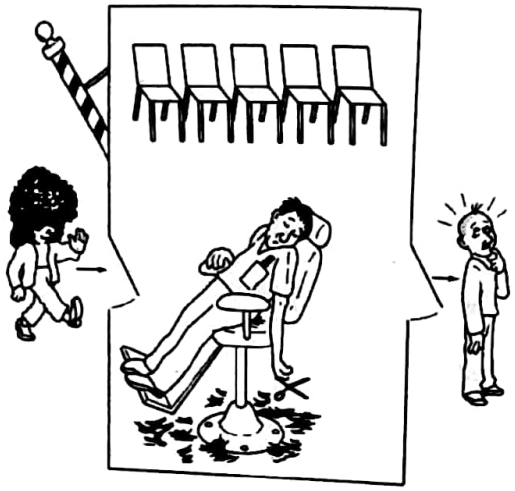


Fig 2.15: Sleeping barber problem

Each customer, when he arrives, looks to see what the barber is doing. If the barber is sleeping, then the customer wakes him up and sits in the chair. If the barber is cutting hair, then the customer goes to the waiting room. If there is a free chair in the waiting room, the customer sits in it and waits his turn. If there is no free chair, then the customer leaves. Based on a naive analysis, the above description should ensure that the shop functions correctly, with the barber cutting the hair of anyone who arrives until there are no more customers, and then sleeping until the next customer arrives. In practice, there are a number of problems that can occur that are illustrative of general scheduling problems.

The problems are all related to the fact that the actions by both the barber and the customer (checking the waiting room, entering the shop, taking a waiting room chair, etc.) all take an unknown amount of time. For example, a customer may arrive and observe that the barber is cutting hair, so he goes to the waiting room. While he is on his way, the barber finishes the haircut he is doing and goes to check the waiting room. Since there is no one there (the customer not having arrived yet), he goes back to his chair and sleeps. The barber is now waiting for a customer and the customer is waiting for the barber.

In another example, two customers may arrive at the same time when there happens to be a single seat in the waiting room. They observe that the barber is cutting hair, go to the waiting room, and both attempt to occupy the single chair.

Solution: Many possible solutions are available. The key element of each is a mutex, which ensures that only one of the participants can change state at once. The barber must acquire this mutex exclusion before checking for customers and release it when he begins either to sleep or cut hair. A customer must acquire it before entering the shop and release it once he is sitting in either a waiting room chair or the barber chair. This eliminates both of the problems mentioned in the previous section. A number of semaphores are also required to indicate the state of the system. For example, one might store the number of people in the waiting room. A multiple sleeping barber's problem has the additional complexity of coordinating several barbers among the waiting customers.

25.3 Dining philosopher problem

In 1965, Dijkstra posed and solved a synchronization problem he called the dining philosophers problem. Since that time, everyone inventing yet another synchronization primitive has felt obligated to demonstrate how wonderful the new primitive is by showing how elegantly it solves the dining philosophers' problem.

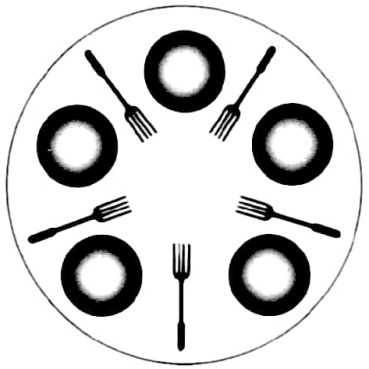


Fig: 2.16: Dining philosopher problem

There are N philosophers sitting around a circular table eating spaghetti and discussing philosophy. The problem is that each philosopher needs 2 forks to eat, and there are only N forks, one between each 2 philosophers. Design an algorithm that the philosophers can follow that insures that none starves as long as each philosopher eventually stops eating, and such that the maximum numbers of philosopher scan eat at once.

- Philosophers eat/think
- Eating needs 2 forks
- Pick one fork at a time
- How to prevent deadlock

The problem was designed to illustrate the problem of avoiding deadlock, a system state in which no progress is possible. One idea is to instruct each philosopher to behave as follows:

- think until the left fork is available; when it is, pick it up
- think until the right fork is available; when it is, pick it up
- eat
- put the left fork down
- put the right fork down
- repeat from the start

This solution is incorrect: it allows the system to reach deadlock. Suppose that all five philosophers take their left forks simultaneously. None will be able to take their right forks, and there will be a deadlock. We could modify the program so that after taking the left fork, the program checks to see if the right fork is available. If it is not, the philosopher puts down the left one, waits for some time, and then repeats the whole process. This proposal too, fails, although for a different reason. With a little bit of bad luck, all the philosophers could start the algorithm simultaneously, picking up their left forks, seeing that their right forks were not available, putting down their left forks, waiting, and picking up their left forks again simultaneously, and so on,

forever. A situation like this, in which all the programs continue to run indefinitely but fail to make any progress is called starvation.

The solution presented below is deadlock-free and allows the maximum parallelism for an arbitrary number of philosophers. It uses an array, state, to keep track of whether a philosopher is eating, thinking, or hungry (trying to acquire forks). A philosopher may move into eating state only if neither neighbor is eating. Philosopher is neighbors are defined by the macros LEFT and RIGHT. In other words, if i is 2, LEFT is 1 and RIGHT is 3.

```
#define N 5                     /* number of philosophers */
#define LEFT (i+N-1)%N          /* number of i's left neighbor */
#define RIGHT (i+1)%N           /* number of i's right neighbor */
#define THINKING 0              /* philosopher is thinking */
#define HUNGRY 1                /* philosopher is trying to get forks */
#define EATING 2                /* philosopher is eating */
Typedef int semaphore;          /* semaphores are a special kind of int */
int state[N];                   /* array to keep track of everyone's state */
semaphore mutex = 1;            /* mutual exclusion for critical regions */
semaphore s[N];                 /* one semaphore per philosopher */
void philosopher (int i)        /* i: philosopher number, from 0 to N-1 */
{
        while (TRUE)
        {
                                /* repeat forever */
                think();        /* philosopher is thinking */
                take_forks(i);  /* acquire two forks or block */
                eat();          /* yum-yum, spaghetti */
                put_forks(i);   /* put both forks back on table */
        }
}
voidtake_forks(int i)           /* i: philosopher number, from 0 to N-1 */
{
        down(&mutex);           /* enter critical region */
        state[i] = HUNGRY;      /* record fact that philosopher i is hungry */
        test(i);                /* try to acquire 2 forks */
        up(&mutex);             /* exit critical region */
        down(&s[i]);            /* block if forks were not acquired */
}
voidput_forks(i)                /* i: philosopher number, from 0 to N-1 */
{
        down(&mutex);           /* enter critical region */
        state[i] = THINKING;    /* philosopher has finished eating */
        test(LEFT);             /* see if left neighbor can now eat */
        test(RIGHT);            /* see if right neighbor can now eat */
        up(&mutex);             /* exit critical region */
}
void test(i)                    /* i: philosopher number, from 0 to N-1 */
{
        if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING)
        {
                state[i] = EATING;
                up(&s[i]);
        }
}
```

A solution to the dining philosophers' problem

26. Process Scheduling

The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy. Process scheduling is an essential part of multiprogramming operating systems. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.

27. Process Scheduling Queues

The OS maintains all PCBs in Process Scheduling Queues. The OS maintains a separate queue for each of the process states and PCBs of all processes in the same execution state are placed in the same queue. When the state of a process is changed, its PCB is unlinked from its current queue and moved to its new state queue.

The Operating System maintains the following important process scheduling queues –

- **Job queue** – This queue keeps all the processes in the system.
- **Ready queue** – This queue keeps a set of all processes residing in main memory, ready and waiting to execute. A new process is always put in this queue.
- **Device queues** – The processes which are blocked due to unavailability of an I/O device constitute this queue.

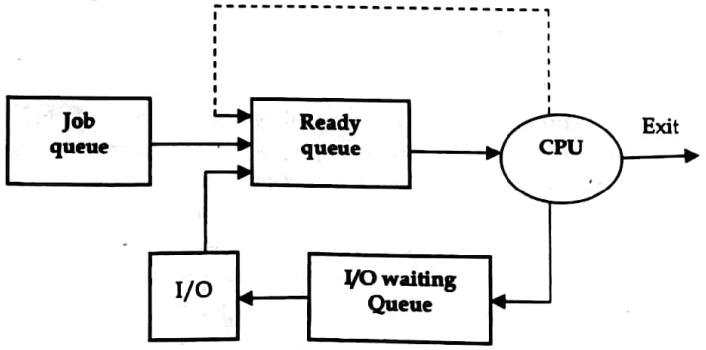


Fig 2.17: Process scheduling queue

The OS can use different policies to manage each queue (FIFO, Round Robin, Priority, etc.). The OS scheduler determines how to move processes between the ready and run queues which can only have one entry per processor core on the system; in the above diagram, it has been merged with the CPU.

Two-State Process Model

Two-state process model refers to running (When a new process is created, it enters into the system as in the running state.) and non-running (Processes that are not running are kept in queue, waiting for their turn to execute. Each entry in the queue is a pointer to a particular process. Queue is implemented by using linked list. Use of dispatcher is as follows. When a process is interrupted, that process is transferred in the waiting queue. If the process has completed or aborted, the process is discarded. In either case, the dispatcher then selects a process from the queue to execute) states.

28. Schedulers

Schedulers are special system software which handles process scheduling in various ways. Their main task is to select the jobs to be submitted into the system and to decide which process to run. Schedulers are of three types

1. **Long Term Scheduler:** It is also called a job scheduler. A long-term scheduler determines which programs are admitted to the system for processing. It selects processes from the queue and loads them into memory for execution. Process loads into the memory for CPU scheduling.

   The primary objective of the job scheduler is to provide a balanced mix of jobs, such as I/O bound and processor bound. It also controls the degree of multiprogramming. If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system. On some systems, the long-term scheduler may not be available or minimal. Time-sharing operating systems have no long term scheduler. When a process changes the state from new to ready, then there is use of long-term scheduler.

2. **Short Term Scheduler:** It is also called as CPU scheduler. Its main objective is to increase system performance in accordance with the chosen set of criteria. It is the change of ready state to running state of the process. CPU scheduler selects a process among the processes that are ready to execute and allocates CPU to one of them. Short-term schedulers, also known as dispatchers, make the decision of which process to execute next. Short-term schedulers are faster than long-term schedulers.

3. **Medium Term Scheduler:** Medium-term scheduling is a part of swapping. It removes the processes from the memory. It reduces the degree of multiprogramming. The medium-term scheduler is in-charge of handling the swapped out-processes.

   A running process may become suspended if it makes an I/O request. A suspended process cannot make any progress towards completion. In this condition, to remove the process from memory and make space for other processes, the suspended process is moved to the secondary storage. This process is called swapping, and the process is said to be swapped out or rolled out. Swapping may be necessary to improve the process mix.

29. Dispatcher

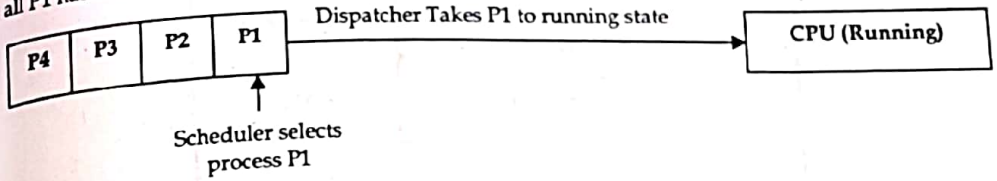
Dispatcher is a special program which comes into play after scheduler. When scheduler completed its job of selecting a process, then after it is the dispatcher which takes that process to the desired state/queue. The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler. This function involves the following:

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

Difference between the Scheduler and Dispatcher

Consider a situation, where various process residing in ready queue and waiting for execution. But CPU can't execute all the process of ready queue simultaneously, operating system have to choose a particular process on the basis of scheduling algorithm used. So, this procedure of selecting a process among various processes is done by scheduler. Now here the task of scheduler completed. Now dispatcher comes into picture as scheduler have decide a process for execution, it is dispatcher who takes that process from ready queue to the running status, or you can say that providing CPU to that process is the task of dispatcher.

Example: There are 4 processes in ready queue, i.e., P1, P2, P3, P4; they all are arrived at t0, t1, t2, t3 respectively. First in First out scheduling algorithm is used. So, scheduler decided that first of all P1 has came, so this is to be executed first. Now dispatcher takes P1 to the running state.

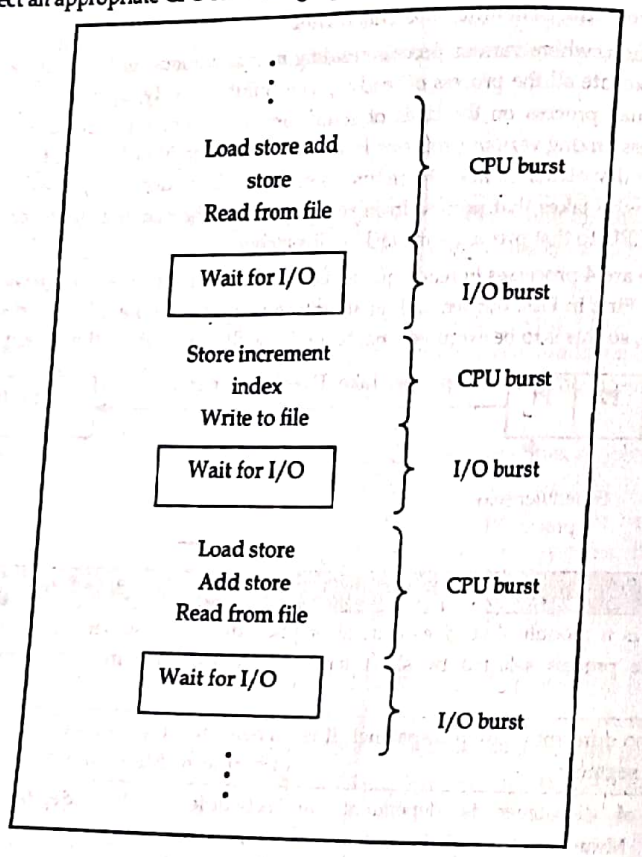


Dispatcher	Scheduler
Dispatcher is a module that gives control of CPU to the process selected by short term scheduler.	Scheduler is something which selects a process among various processes.
There are no different types in dispatcher. It is just a code segment.	There are 3 types of scheduler i.e. Long-term, Short-term, Medium-term.
Working of dispatcher is dependent on scheduler. Means dispatchers have to wait until scheduler selects a process.	Scheduler works independently. It works immediately when needed.
Dispatcher has no specific algorithm for its implementation	Scheduler works on various algorithm such as FCFS, SJF, RR etc.
The time taken by dispatcher is called dispatch latency.	Time taken by scheduler is usually negligible. Hence we neglect it.
Dispatcher is also responsible for: Context Switching, Switch to user mode, Jumping to proper location when process again restarted	The only work of scheduler is selection of processes.

30. CPU - I/O Burst Cycle

The success of CPU scheduling depends on the following observed property of processes: Proces execution consists of a cycle of CPU execution and I/O wait. Processes alternate back and forth between these two states. The execution begins with CPU burst, followed by I/O burst, the another CPU burst and so on. The last CPU burst will end with a system request to terminat execution rather than with another I/O burst.

An I/O bound program would typically have many short CPU bursts; a CPU bound program might have a few very long CPU bursts. The duration of these CPU bursts are measured, which help to select an appropriate CPU scheduling algorithm.



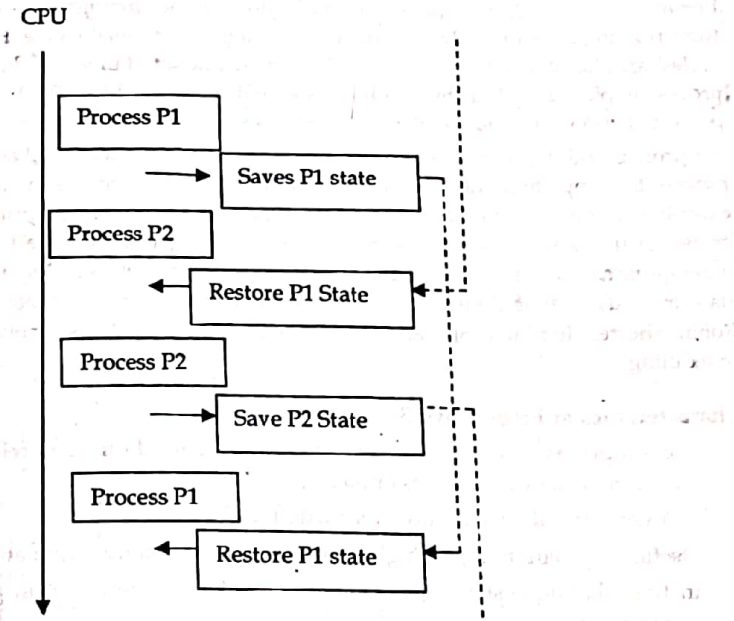
31. Context Switch

A context switch is the mechanism to store and restore the state or context of a CPU in Process Control block so that a process execution can be resumed from the same point at a later time. Using this technique, a context switcher enables multiple processes to share a single CPU. Context switching is an essential part of a multitasking operating system features.

When the scheduler switches the CPU from executing one process to execute another, the state from the current running process is stored into the process control block. After this, the state for

the process to run next is loaded from its own PCB and used to set the PC, registers, etc. At that point, the second process can start executing. Context switches are computationally intensive since register and memory state must be saved and restored. To avoid the amount of context switching time, some hardware systems employ two or more sets of processor registers. When the process is switched, the following information is stored for later use.

- Program Counter
- Scheduling information
- Base and limit register value
- Currently used register
- Changed State
- I/O State information
- Accounting information



32. Non-preemptive scheduling

Non-preemptive Scheduling is one which can be applied in the circumstances when a process terminates, or a process switches from running to waiting state. In Non-Preemptive Scheduling, once the resources (CPU) are allocated to a process, the process holds the CPU till it gets terminated or it reaches a waiting state.

Unlike preemptive scheduling, non-preemptive scheduling does not interrupt a process running CPU in middle of the execution. Instead, it waits for the process to complete its CPU burst time and then it can allocate the CPU to another process. In Non-preemptive scheduling, if a process with long CPU burst time is executing then the other process will have to wait for a long time which increases the average waiting time of the processes in the ready queue. However, the non-

preemptive scheduling does not have any overhead of switching the processes from ready queue to CPU but it makes the scheduling rigid as the process in execution is not even preempted for a process with higher priority.

Characteristics of Non-Preemptive Scheduling

- Picks a process to run until it releases the CPU
- Once a process has been given the CPU, it runs until blocks for I/O or termination
- Treatment of all processes is fair
- Response times are more predictable
- Useful in real-time system
- Shorts jobs are made to wait by longer jobs -no priority

33. Preemptive Scheduling

Preemptive scheduling is one which can be done in the circumstances when a process switches from running state to ready state or from waiting state to ready state. Here, the resources (CPU cycles) are allocated to the process for the limited amount of time and then is taken away, and the process is placed back in the ready queue again if it still has CPU burst time remaining. The process stays in ready queue till it gets next chance to execute.

If a process with high priority arrives in the ready queue, it does not have to wait for the current process to complete its burst time. Instead, the current process is interrupted in the middle of execution and is placed in the ready queue till the process with high priority is utilizing the CPU cycles. In this way, each process in the ready queue gets some time to run CPU. It makes the preemptive scheduling flexible but, increases the overhead of switching the process from running state to ready state and vise-verse. Algorithms that work on preemptive scheduling are Round Robin. Shortest Job First (SJF) and Priority scheduling may or may not come under preemptive scheduling.

Characteristics of Preemptive Scheduling

- Picks a process and let it run for a maximum of fixed time and releases the CPU after that quantum, whether it finishes or not.
- Processes are allowed to run for a maximum of some fixed time.
- Useful in systems in which high-priority processes requires rapid attention.
- In time sharing systems, preemptive scheduling is important in guaranteeing acceptable response times.
- High overhead.

CPU scheduling decisions may take place under the following four circumstances:

- When a process switches from the running state to the waiting state (for. example, I/O request, or invocation of wait for the termination of one of the child processes).
- When a process switches from the running state to the ready state (for example, when an interrupt occurs).
- When a process switches from the waiting state to the ready state (for example, completion of I/O).
- When a process terminates.

Comparisons of preemptive and non-preemptive scheduling

Preemptive scheduling	Non preemptive scheduling
The resources are allocated to a process for a limited time.	Once resources are allocated to a process, the process holds it till it completes its burst time or switches to waiting state.
Process can be interrupted in between.	Process cannot be interrupted till it terminates or switches to waiting state.
If a high priority process frequently arrives in the ready queue, low priority process may starve.	If a process with long burst time is running CPU, then another process with less CPU burst time may starve.
Preemptive scheduling has overheads of scheduling the processes.	Non-preemptive scheduling does not have overheads.
Preemptive scheduling is flexible.	Non-preemptive scheduling is rigid.
Preemptive scheduling is cost associated.	Non-preemptive scheduling is not cost associative.

34. Scheduling Criteria

Different CPU scheduling algorithms have different properties and may favor one class of processes over another. In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms. Many criteria have been suggested for comparing CPU scheduling algorithms. Criteria that are used include the following:

1. **CPU utilization** – keep the CPU as busy as possible
2. **Throughput** – number of processes that complete their execution per time unit
3. **Turnaround time** – amount of time to execute a particular process
4. **Waiting time** – amount of time a process has been waiting in the ready queue
5. **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

The goals of scheduling are as follows:

- **Fairness:** Each process gets fair share of the CPU.
- **Efficiency:** When CPU is 100% busy then efficiency is increased.
- **Response Time:** Minimize the response time for interactive user.
- **Throughput:** Maximizes jobs per given time period.
- **Waiting Time:** Minimizes total time spent waiting in the ready queue.
- **Turnaround Time:** Minimizes the time between submission and termination.

35. Batch System Scheduling

35.1 First come first served

FCFS provides an efficient, simple and error-free process scheduling algorithm that saves valuable CPU resources. It uses non-preemptive scheduling in which a process is automatically

queued and processing occurs according to an incoming request or process order. FCFS derives its concept from real-life customer service.

Let's take a look at how FCFS process scheduling works. Suppose there are three processes in a queue: P1, P2 and P3. P1 is placed in the processing register with a waiting time of zero seconds and 10 seconds for complete processing. The next process, P2, must wait 10 seconds and is placed in the processing cycle until P1 is processed. Assuming that P2 will take 15 seconds to complete, the final process, P3, must wait 25 seconds to be processed. FCFS may not be the fastest process scheduling algorithm, as it does not check for priorities associated with processes. These priorities may depend on the processes' individual execution times.

Characteristics

- Processes are scheduled in the order they are received.
- Once the process has the CPU, it runs to completion
- Easily implemented, by managing a simple queue or by storing time the process was received.
- Fair to all processes.

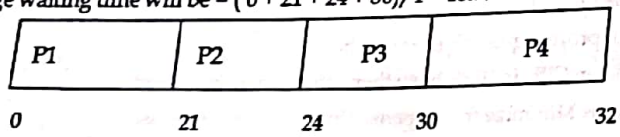
Problems

- No guarantee of good response time.
- Large average waiting time.

Example: Consider the processes P1, P2, P3, P4 given in the below table, arrives for execution in the same order, with Arrival Time 0, and given Burst Time, let's find the average waiting time using the FCFS scheduling algorithm.

Process	Burst Time
P1	21
P2	3
P3	6
P4	2

The average waiting time will be = (0 + 21 + 24 + 30)/4 = 18.75 ms



The average waiting time will be 18.75 ms

35.2 Shortest job first

Shortest job first (SJF) or shortest job next, is a scheduling policy that selects the waiting process with the smallest execution time to execute next. SJN is a non-preemptive algorithm. Shortest Job first has the advantage of having minimum average waiting time among all scheduling algorithms.

It is a Greedy Algorithm, which may cause starvation if shorter processes keep coming. It is practically infeasible as Operating System may not know burst time and therefore may not sort

them. While it is not possible to predict execution time, several methods can be used to estimate the execution time for a job, such as a weighted average of previous execution times. SJF can be used in specialized environments where accurate estimates of running time are available.

Characteristics

- The processing times are known in advanced.
- SJF selects the process with shortest expected processing time. In case of the tie FCFS scheduling is used.
- The decision policies are based on the CPU burst time. Advantages:
- Reduces the average waiting time over FCFS.
- Favors shorts jobs at the cost of long jobs.

Problems

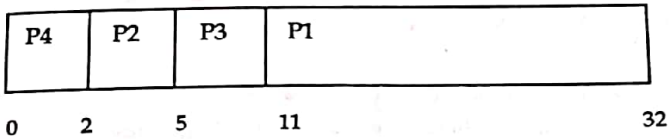
- Estimation of run time to completion.
- Accuracy
- Not applicable in timesharing system.

Example: Consider the processes P1, P2, P3, P4 given in the below table, arrives for execution in the same order, with Arrival Time 0, and given Burst Time, let's find the average waiting time using the SJF scheduling algorithm.

Process	Burst Time
P1	21
P2	3
P3	6
P4	2

In shortest Job First Scheduling, the shortest Process is executed first.

Hence the Gantt chart will be as follow



Now, the average waiting time will be = (0 + 2 + 5 + 11) /4 =4.5 ms

As in the Gantt chart above, the process P4 will be picked up first as it has the shortest burst time, then P2, followed by P3 and at last P1.

We scheduled the same set of processes using the First come first serve algorithm, and got average waiting time to be 18.75 ms, whereas with SJF, the average waiting time comes out 4.5 ms.

35.3 Shortest remaining time next

It is a preemptive version of shortest job next scheduling. In this scheduling algorithm, the process with the smallest amount of time remaining until completion is selected to execute. Since the currently executing process is the one with the shortest amount of time remaining by definition, and since that time should only reduce as execution progresses, processes will always run until they complete or a new process is added that requires a smaller amount of time.

Shortest remaining time is advantageous because short processes are handled very quickly. The system also requires very little overhead since it only makes a decision when a process completes or a new process is added, and when a new process is added the algorithm only needs to compare the currently executing process with the new process, ignoring all other processes currently waiting to execute.

Characteristics

- Low average waiting time than SJF
- Useful in timesharing

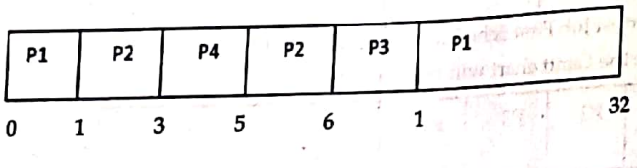
Demerits

- Very high overhead than SJF
- Requires additional computation.
- Favors short jobs, longs jobs can be victims of starvation.

Example: Consider the processes P1, P2, P3, P4 given in the below table, arrives for execution in the same order, with Arrival Time 0, and given Burst Time, let's find the average waiting time using the **Shortest remaining time next** scheduling algorithm.

Process	Burst Time	Arrival Time
P1	21	0
P2	3	1
P3	6	2
P4	2	3

The Gantt chart for Preemptive Shortest remaining time next Scheduling will be



The average waiting time will be ((5-3) + (6-2) + (12-1))/4 =4.25 ms

The average waiting time for preemptive shortest job first scheduling is less than both, non-preemptive SJF scheduling and FCFS scheduling.

As it is seen in the GANTT chart above, as P1 arrives first, hence it's execution starts immediately, but just after 1 ms, process P2 arrives with a burst time of 3 ms which is less than the burst time of P1, hence the process P1(1 ms done, 20 ms left) is preempted and process P2 is executed.

As P2 is getting executed, after 1 ms, P3 arrives, but it has a burst time greater than that of P2, hence execution of P2 continues. But after another millisecond, P4 arrives with a burst time of 2 ms, as a result P2(2 ms done, 1 ms left) is preempted and P4 is executed.

After the completion of P4, process P2 is picked up and finishes, then P2 will get executed and at last P1.

36. Interactive System Scheduling

36.1 Round Robin scheduling

In this algorithm the process is allocated the CPU for the specific time period called time slice called quantum, which is normally of 10 to 100 milliseconds. If the process completes its execution within this time slice, then it is removed from the queue otherwise it has to wait for another time slice. Preempted process is placed at the back of the ready list.

Advantages

- Fair allocation of CPU across the process.
- Used in timesharing system.
- Low average waiting time when process lengths vary widely.
- Poor average waiting time when process lengths are identical.

Quantum size: If the quantum is very large, each process is given as much time as needs for completion; RR degenerate to FCFS policy. If quantum is very small, system busy at just switching from one process to another process, the overhead of context-switching causes the system efficiency degrading.

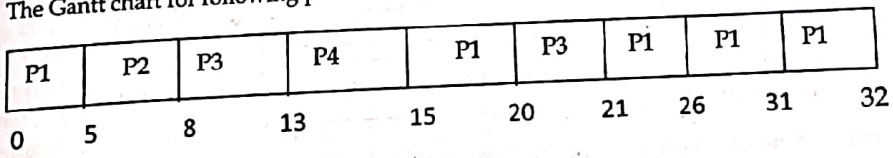
Optimal quantum size

Key idea: 80% of the CPU bursts should be shorter than the quantum. 20-50 msec reasonable for many general processes.

Example: Consider the processes P1, P2, P3, P4 given in the below table, arrives for execution in the same order, with Arrival Time 0, and given Burst Time, let's find the average waiting time using the **Round Robin scheduling** algorithm.

Process	Burst Time
P1	21
P2	3
P3	6
P4	2

The Gantt chart for following processes based on Round robin scheduling will be



The average waiting time will be 11 ms

36.2 Priority scheduling

In this scheduling algorithm the priority is assigned to all the processes and the process with highest priority executed first. Priority assignment of processes is done on the basis of internal factor such as CPU and memory requirements or external factor such as user's choice. The priority scheduling algorithm supports preemptive and non - preemptive scheduling policy. The CPU is allocated to the process with the highest priority (smallest integer ° highest priority). SJF is a priority scheduling where priority is the predicted next CPU burst time

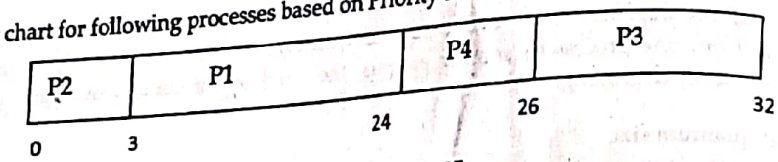
Problem: Starvation – low priority processes may never execute

Solution: Aging – as time progresses increase the priority of the process

Example: Consider the processes P1, P2, P3, P4 given in the below table, arrives for execution in the same order, with Arrival Time 0, and given Burst Time, let's find the average waiting time using the Priority scheduling algorithm.

Process	Burst Time	Priority
P1	21	2
P2	3	1
P3	6	4
P4	2	3

The Gantt chart for following processes based on Priority scheduling will be

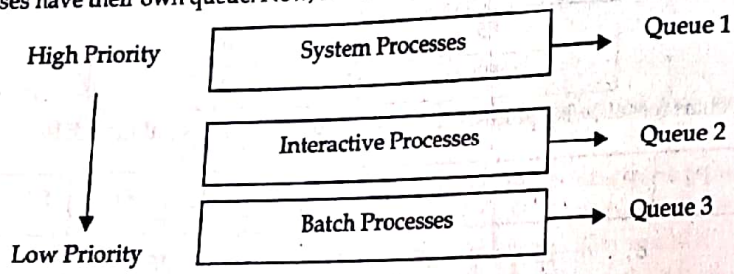


The average waiting time will be (0 + 3 + 24 + 26) / 4 = 13.25 ms

36.3 Multiple queues

It may happen that processes in the ready queue can be divided into different classes where each class has its own scheduling needs. For example, a common division is a foreground (interactive) process and background (batch) processes. These two classes have different scheduling needs. For this kind of situation Multilevel Queue Scheduling is used. Now, let us see how it works.

Ready Queue is divided into separate queues for each class of processes. For example, let us take three different types of process System processes, Interactive processes and Batch Processes. All three processes have their own queue. Now, look at the below figure.



All three different type of processes have their own queue. Each queue has its own Scheduling algorithm. For example, queue 1 and queue 2 uses Round Robin while queue 3 can use FCFS to schedule their processes.

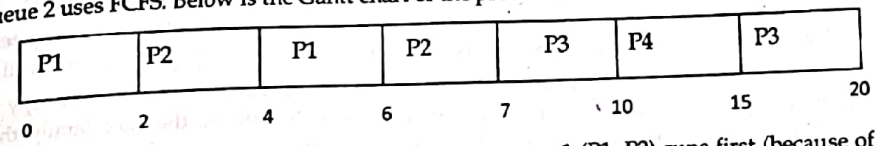
Scheduling among the queues: What will happen if all the queues have some processes? Which process should get the CPU? To determine this Scheduling among the queues is necessary. There are two ways to do so

1. **Fixed priority preemptive scheduling method** – Each queue has absolute priority over lower priority queue. Let us consider following priority order queue 1 > queue 2 > queue 3. According to this algorithm no process in the batch queue (queue 3) can run unless queue 1 and 2 are empty. If any batch process (queue 3) is running and any system (queue 1) or Interactive process (queue 2) entered the ready queue the batch process is preempted.
2. **Time slicing** – In this method each queue gets certain portion of CPU time and can use it to schedule its own processes. For instance, queue 1 takes 50 percent of CPU time queue 2 takes 30 percent and queue 3 gets 20 percent of CPU time.

Example: Consider below table of four processes under multilevel queue scheduling. Queue number denotes the queue of the process.

Process	Arrival Time	CPU Burst Time	Queue Number
P1	0	4	1
P2	0	3	1
P3	0	8	2
P4	10	5	1

Priority of queue 1 is greater than queue 2. Queue 1 uses Round Robin (Time Quantum = 2) and queue 2 uses FCFS. Below is the Gantt chart of the problem:



At starting both queues have process so process in queue 1 (P1, P2) runs first (because of higher priority) in the round robin fashion and completes after 7 units then process in queue 2 (P3) starts running (as there is no process in queue 1) but while it is running P4 comes in queue 1 and interrupts P3 and start running for 5 second and after its completion P3 takes the CPU and completes its execution.

37. Overview of real time system scheduling

A real-time scheduling System is composed of the scheduler, clock and the processing hardware elements. In a real-time system, a process or task has schedulability; tasks are accepted by a real-time system and completed as specified by the task deadline depending on the characteristic of the scheduling algorithm. Modeling and evaluation of a real-time scheduling system concern is on the analysis of the algorithm capability to meet a process deadline. A deadline is defined as the time required for a task to be processed. For example, in a real-time scheduling algorithm a deadline could be set to 5 nano-seconds. In a critical operation the task must be processed in the time specified by the deadline (i.e. five nano-seconds). A task in a real-time system must be completed neither too early nor too late. A system is said to be unschedulable when tasks cannot meet the specified deadlines. A task can be classified as either a periodic or a periodic process.

The criteria of a real-time can be classified as hard, firm or soft. The scheduler set the algorithms for executing tasks according to a specified order. There are multiple mathematical models to represent a scheduling System; most implementations of real-time scheduling algorithm are modeled for the implementation of uniprocessors or multiprocessors configurations. In the algorithm for a real-time scheduling system, each task is assigned a description, deadline and an identifier (indicating priority). The selected scheduling algorithm determines how priorities are assigned to a particular task. A real-time scheduling algorithm can be classified as static or dynamic. For a static scheduler, task priorities are determined before the system runs. A dynamic

scheduler determines task priorities as it runs. Tasks are accepted by the hardware elements in real-time scheduling system from the computing environment and processed in real-time. A output signal indicates the processing status. A task deadline indicates the time set to comple for each task.

Laboratory Works

Create process, threads and implement IPC techniques. Simulate process Scheduling algorithm

Create process

- Each process in a Linux system is identified by its unique process ID, referred to as pid.
- Process IDs are 1- 32768 numbers that are assigned sequentially by Linux as new proces are created.
- Every process also has a parent process (except init). Thus, we can think the processes in the Linux are arranged in a tree, with the init process at its root.
- The parent process ID, or ppid, is simply the process ID of the process's parent.
- Most of the process manipulation functions are declared in the header file <unistd.h>.
- A program can obtain the process ID of the process it's running with the getpid() system call, and it can obtain the process ID of its parent process with the getppid() system call.
- Linux provides one function, **fork()** that makes the child process that is an exact copy of its parent process. Linux also provides another set of functions, the exec family that replaces the current process image with a new process image.

Program 1: Creation of a single process and display their id.

```
#include <stdio.h>
#include <unistd.h> /* contains fork prototype */
int main(void)
{
        printf("Hello World!\n");
        fork( );
        printf("I am after forking\n");
        printf("\t I am process %d.\n", getpid( ));
}
```

When this program is executed, it first prints Hello World! . When the fork is executed, an identical process called the child is created. Then both the parent and the child process begin execution at the next statement.

Program 2: Program getting an id of a process

```
#include <stdio.h>
#include <unistd.h> /* contains fork prototype */
int main(void)
{
        int pid;
        printf("Hello World!\n");
        printf("I am the parent process and pid is : %d .\n", getpid());
        printf("Here i am before use of forking\n");
        pid = fork();
        printf("Here I am just after forking\n");
        if (pid == 0)
                printf("I am the child process and pid is :%d.\n", getpid());
        else
                printf("I am the parent process and pid is: %d .\n", getpid());
}
```


Program 3: Use of multiple fork()

```
#include <stdio.h>
#include <unistd.h> /* contains fork prototype */
void main(void)
{
        printf("Here I am just before first forking statement\n");
        fork();
        printf("Here I am just after first forking statement\n");
        fork();
        printf("Here I am just after second forking statement\n");
        printf("\t\t Hello World from process %d!\n", getpid());
}
```

Thread Creation

Each thread has their own thread ID as process, thread ID referred by type **pthread_t.**

The pthread_create function creates new threads. It has following format.

```
int pthread_create (pthread_t *thread, pthread_attr_t *attr, void
*(*start_routine)(void*), void *arg);
```

The pthread_exit function terminates the thread.

```
thread_exit(void *return_val);
```

The pthread_join function waits other process for termination – equivalent of wait.

```
int pthread_join(pthread_t th, void **thread_return);
```

Program 4: Thread Creation (threadc.c)

```
#include<stdio.h>
#include<string.h>
#include<pthread.h>
#include<stdlib.h>
#include<unistd.h>
pthread_t tid[2];
void* doSomeThing(void *arg)
{
        unsigned long i = 0;
        pthread_t id = pthread_self();
        if(pthread_equal(id,tid[0]))
        {
          printf("\n First thread processing\n");
```

```
        }
        else
        {
          printf("\n Second thread processing\n");
        }
        for(i=0; i<(0xFFFFFFFF);i++);
        return NULL;
}
int main(void)
{
        int i = 0;
        int err;
        while(i < 2)
        {
          err = pthread_create(&(tid[i]), NULL, &doSomeThing, NULL);
          if (err != 0)
              printf("\ncan't create thread :[%s]", strerror(err));
          else
              printf("\n Thread created successfully\n");
          i++;
        }
        sleep(5);
        return 0;
}
```

So what this code does is :

- It uses the **pthread_create()** function to create two threads
- The starting function for both the threads is kept same.
- Inside the function 'doSomeThing()', the thread uses pthread_self() and pthread_equal() functions to identify whether the executing thread is the first one or the second one as created.
- Also, Inside the same function 'doSomeThing()' a for loop is run so as to simulate some time consuming work.

Now, when the above code is run, we get following output:

- $ ./threads
- Thread created successfully
- First thread processing
- Thread created successfully
- Second thread processing

Program 5: Implement IPC techniques

This program demonstrate the solution (strict alternation) for critical region problem.

Strictalt.c

```
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>
```

```
#include<stdio.h>
void *thread1f (void * arg);
void *thread2f (void * arg);
int turn = 1;
int main()
{
        pthread_t thid1;
        pthread_t thid2;
        pthread_create (&thid1, NULL, &thread1f, NULL);
        pthread_create (&thid2, NULL, &thread2f, NULL);
        pthread_join(thid1, NULL);
        pthread_join(thid2, NULL);
        return 0;
}
void *thread1f(void *arg)
{
        int a = 0;
        while(a++<20)
        {
                while(turn!= 1);
                        fputc('b',stderr);
                turn = 0;
        }
}
void *thread2f (void * arg)
{
        int b = 0;
        while(b++<20)
        {
                while(turn != 0);
                fputc('a', stderr);
                turn = 1;
        }
}
```

Simulate process Scheduling algorithms

Program 6: Simulating First Come First Served (FCFS) FIFO scheduling algorithm in C

```
#include<stdio.h>
int main()
{
  int n, bt[20],wt[20],tat[20], avwt=0, avtat=0, i, j;
  printf("Enter total number of processes(maximum 20):");
  scanf("%d", &n);
  printf("\n Enter Process Burst Time\n");
  for(i=0; i<n; i++)
  {
    printf("P[%d]:", i+1);
```

```
        scanf("%d", &bt[i]);
      }
      wt[0]=0;   //waiting time for first process is 0
      for(i=1; i<n; i++)   //calculating waiting time
      {
        wt[i]=0;
        for(j=0; j<i; j++)
          wt[i]+=bt[j];
      }
      printf("\n Process\t\t Burst Time\t Waiting Time\t Turnaround Time");
      for(i=0; i<n; i++)  //calculating turnaround time
      {
        tat[i]=bt[i]+wt[i];
        avwt+=wt[i];
        avtat+=tat[i];
        printf("\nP[%d]\t\t%d\t\t%d\t\t%d",i+1,bt[i],wt[i],tat[i]);
      }
      avwt/=i;
      avtat/=i;
      printf("\n\n Average Waiting Time:%d", avwt);
      printf("\n Average Turnaround Time:%d", avtat);
      return 0;
}
```

Program 7: Simulating Shortest Job first (SJF) scheduling algorithm in C

```
#include<stdio.h>
void main()
{
      int bt[20], p[20], wt[20],tat[20], i, j, n, total=0, pos, temp;
      float avg_wt, avg_tat;
      printf("Enter number of process:");
      scanf("%d", &n);
      printf("\n Enter Burst Time:\n");
      for(i=0; i<n; i++)
      {
            printf("p%d:", i+1);
            scanf("%d", &bt[i]);
            p[i]=i+1;        //contains process number
      }
      //sorting burst time in ascending order using selection sort
      for(i=0; i<n; i++)
      {
        pos=i;
        for(j=i+1;j<n;j++)
        {
          if(bt[j]<bt[pos])
            pos=j;
```

```
        }
        temp=bt[i];
        bt[i]=bt[pos];
        bt[pos]=temp;
        temp=p[i];
        p[i]=p[pos];
        p[pos]=temp;
      }
      wt[0]=0;          //waiting time for first process will be zero
      //calculate waiting time
      for(i=1;i<n;i++)
      {
            wt[i]=0;
            for(j=0;j<i;j++)
            wt[i]+=bt[j];
            total+=wt[i];
      }
      avg_wt = (float)total/n;     //average waiting time
      total=0;
      printf("\n Process\t Burst Time \t Waiting Time\t Turnaround Time");
      for(i=0; i<n; i++)
      {
        tat[i]=bt[i]+wt[i];    //calculate turnaround time
        total+=tat[i];
        printf("\n p %d\t\t  %d\t\t    %d\t\t\t %d", p[i], bt[i], wt[i], tat[i]);
      }
      avg_tat = (float) total/n;    //average turnaround time
      printf("\n\n Average Waiting Time=%f", avg_wt);
      printf("\n Average Turnaround Time=%f\n", avg_tat);

}
```

Program 8: Simulating Round Robin scheduling algorithm in C

```
#include<stdio.h>
int main()
{
 int count, j, n, time, remain, flag=0, time_quantum;
 int wait_time=0, turnaround_time=0, at[10], bt[10], rt[10];
 printf("Enter Total Process:\t ");
 scanf("%d", &n);
 remain=n;
for(count=0; count<n; count++)
{
 printf("Enter Arrival Time and Burst Time for Process Process Number %d :",count+1);
 scanf("%d", &at[count]);
scanf("%d", &bt[count]);
rt[count]=bt[count];
}
```

```
printf("Enter Time Quantum:\t");
scanf("%d", &time_quantum);
printf("\n\n Process\t|Turnaround Time|Waiting Time\n\n");
for(time=0, count=0; remain!=0;)
{
 if(rt[count]<=time_quantum && rt[count]>0)
 {
  time+=rt[count];
  rt[count]=0;
  flag=1;
 }
 else if(rt[count]>0)
 {
  rt[count]-=time_quantum;
  time+=time_quantum;
 }
 if(rt[count]==0 && flag==1)
 {
  remain--;
  printf("P[%d]\t|\t%d\t|\t%d\n",count+1,time-at[count],time-at[count]-bt[count]);
  wait_time+=time-at[count]-bt[count];
  turnaround_time+=time-at[count];
  flag=0;
 }
 if(count==n-1)
        count=0;
 else if(at[count+1]<=time)
        count++;
 else
        count=0;
}
printf("\n Average Waiting Time= %f\n", wait_time*1.0/n);
printf("AVG Turnaround Time = %f", turnaround_time*1.0/n);
return 0;
}
```

Exerci

1. Wh
2. Wh
3. W
4. Fo
   tr

3.
4.
5.
6.
7.
8

Exercise

1. What are disadvantages of too much multiprogramming?
2. What is process? How it is differ from program? Explain.
3. What is thread? How it is differ from process? Explain.
4. For each of the following transitions between the processes states, indicate whether the transition is possible. If it is possible, give an example of one thing that would cause it.
   a. Running -> Ready
   b. Running -> Blocked
   c. Blocked -> Running
3. Describe how multithreading improve performance over a singled-threaded solution.
4. What are the two differences between the kernel level threads and user level threads? Which one has a better performance?
5. What is multi threading? Explain
6. Describe differentiate between preemptive and non-preemptive scheduling with suitable example.
7. How processes and threads are created in compiler like gcc.
8. What resources are used when a thread is created? How do they differ from those used when a process is created?
9. What is the meaning of busy waiting? What others kinds of waiting are in OS? Compare each type on their applicability and relative merits.
10. Compare the use of monitor and semaphore operations.
11. What is producer consumer problem? explain
12. When a computer is being developed, it is usually first simulated by a program that runs one instruction at a time. Even multiprocessors are simulated strictly sequentially like this. Is it possible for a race condition to occur when there are no simultaneous events like this?
13. Does the busy waiting solution using the turn variable work when the two processes are running on a shared-memory multiprocessor, that is, two CPUs sharing a common memory?
14. What is process scheduling? Why it is used? Explain any two scheduling algorithms with suitable example.
15. What is semaphore? Explain their functions.
16. Does Peterson's solution to the mutual exclusion problem work when process scheduling is preemptive? How about when it is non-preemptive?
17. Round-robin schedulers normally maintain a list of all runnable processes, with each process occurring exactly once in the list. What would happen if a process occurred twice in the list? Can you think of any reason for allowing this?
18. Five jobs are waiting to be run. Their expected run times are 9, 6, 3, 5, and X. In what order should they be run to minimize average response time? (Your answer will depend on X.)
19. Five batch jobs A through E, arrive at a computer center at almost the same time. They have estimated running times of 10, 6, 2, 4, and 8 minutes. Their (externally determined) priorities are 3, 5, 2, 1, and 4, respectively, with 5 being the highest priority. For each of the following

scheduling algorithms, determine the mean process turnaround time. Ignore proces switching overhead.

(a) Round robin.

(b) Priority scheduling.

(c) First-come, first-served (run in order 10, 6, 2, 4, and 8).

(d) Shortest job first.

For (a), assume that the system is multi-programmed, and that each job gets its fair share the CPU. For (b) through (d) assume that only one job at a time runs, until it finishes. A jobs are completely CPU bound.

20. For the processes listed in following table, draw a Gantt chart illustrating their executi using:

(a) First-Come-First-Serve

(b) Short-Job-First

(c) Shortest-Remaining-Time-Next

(d) Round-Robin (quantum = 2)

(e) Round-Robin (quantum = 1)

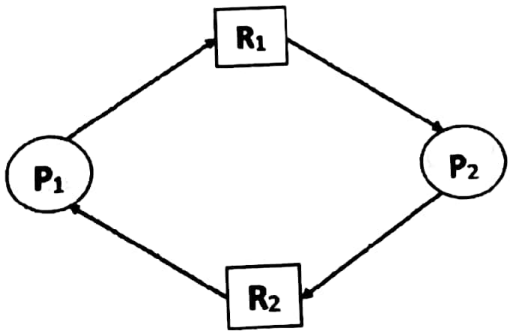
Processes	Arrival Time	Burst Time
A	0.00	4
B	2.01	7
C	3.01	2
D	3.02	2

Chapter 3
PROCESS DEADLOCKS

1. Introduction to Deadlock

The earliest computer operating systems ran only one program at a time. All of the resources of the system were available to this one program. Later, operating systems ran multiple programs at once, interleaving them. Programs were required to specify in advance what resources they needed so that they could avoid conflicts with other programs running at the same time. Eventually some operating systems offered dynamic allocation of resources. Programs could request further allocations of resources after they had begun running.

A set of process is in a deadlock state if each process in the set is waiting for an event that can be caused by only another process in the set. In other words, deadlock is a situation in which two computer programs sharing the same resource are effectively preventing each other from accessing the resource, resulting in none of the processes can run, none of them can release any resources, and none of them can be awakened. It is important to note that the number of processes and the number and kind of resources possessed and requested are unimportant. The resources may be either physical or logical. Examples of physical resources are Printers, Tape Drivers, Memory Space, and CPU Cycles. Examples of logical resources are Files, Semaphores, and Monitors. This leads to the problem of the deadlock. Here is the simplest example:



- Program 1 requests resource R1 and receives it.
- Program 2 requests resource R2 and receives it.
- Program 1 requests resource R2 and is queued up, pending the release of R2.
- Program 2 requests resource R1 and is queued up, pending the release of R1.

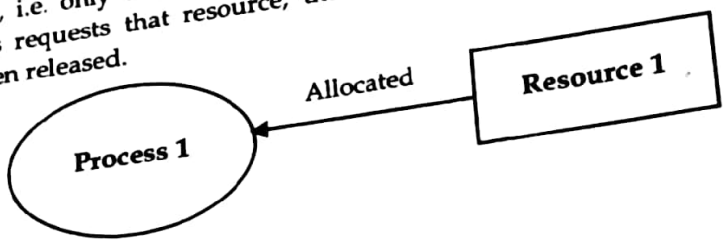
Now neither program can proceed until the other program releases a resource. The operating system cannot know what action to take. At this point the only alternative is to abort (stop) one of the programs.

2. Deadlock Characterization

In a deadlock, processes never finish executing, and system resources are tied up, preventing other jobs from starting. Various features that characterize deadlock are listed below:

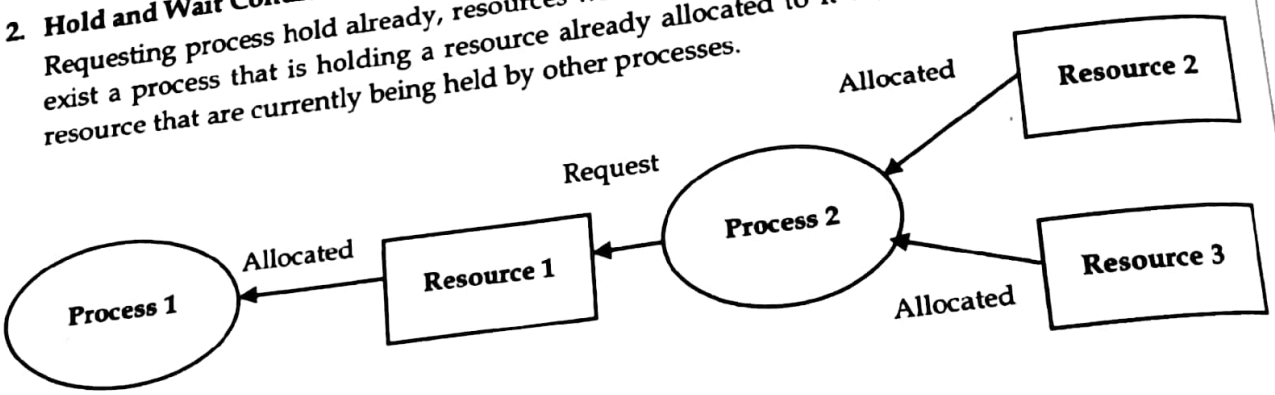
1. **Mutual Exclusion Condition**

The resources involved are non-shareable, At least one resource must be held in a non-shareable mode, i.e. only one process at a time claims exclusive control of the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.



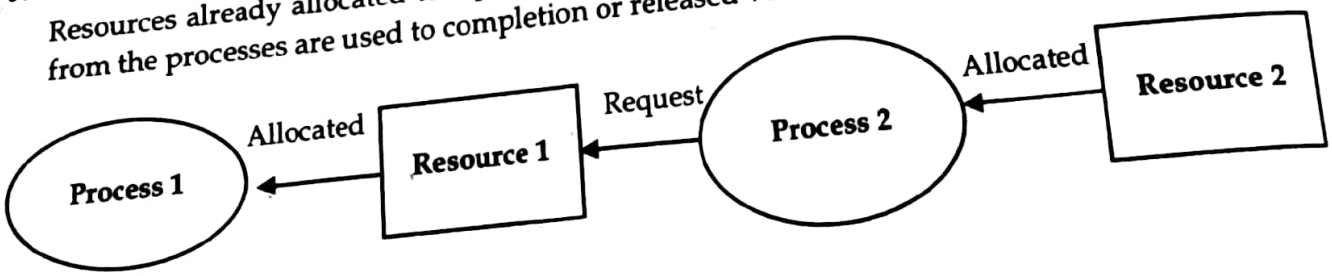
2. **Hold and Wait Condition**

Requesting process hold already, resources while waiting for requested resources, there must exist a process that is holding a resource already allocated to it while waiting for additional resource that are currently being held by other processes.



3. **No-Preemptive Condition**

Resources already allocated to a process cannot be preempted, Resources cannot be removed from the processes are used to completion or released voluntarily by the process holding it.



4. **Circular Wait Condition**

A set {P0 P1, P2... Pn} of waiting processes must exist such that P1 is waiting for a resource that is held by P1, P1 is waiting for a resource that is held by P2, ..., Pn-1 is waiting for a resource that is held by Pn and Pn is waiting for a resource that is held by P0. The processes in the system form a circular list or chain where each process in the list is waiting for a resource held by the next process in the list. As an example, consider the traffic deadlock in the following figure

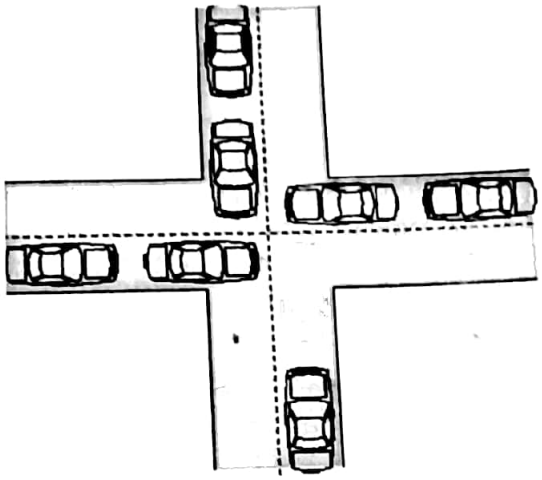


Fig: 3.1: Condition of deadlock

Consider each section of the street as a resource.

- Mutual exclusion condition applies, since only one vehicle can be on a section of the street at a time.
- Hold-and-wait condition applies, since each vehicle is occupying a section of the street, and waiting to move on to the next section of the street.
- No-preemptive condition applies, since a section of the street that is a section of the street that is occupied by a vehicle cannot be taken away from it.
- Circular wait condition applies, since each vehicle is waiting on the next vehicle to move. That is, each vehicle in the traffic is waiting for a section of street held by the next vehicle in the traffic.

The simple rule to avoid traffic deadlock is that a vehicle should only enter an intersection if it is assured that it will not have to stop inside the intersection. It is not possible to have a deadlock involving only one single process. The deadlock involves a circular hold-and-wait condition between two or more processes, so one process cannot hold a resource, yet be waiting for another resource that it is holding. In addition, deadlock is not possible between two threads in a process, because it is the process that holds resources, not the thread that is, each thread has access to the resources held by the process.

3. Preemptable and Nonpreemptable Resources

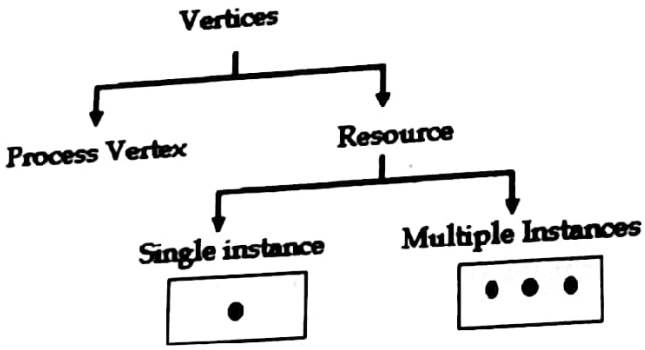
Resources come in two flavors: preemptable and nonpreemptable. A preemptable resource is one that can be taken away from the process with no ill effects. Memory is an example of a preemptable resource. On the other hand, a nonpreemptable resource is one that cannot be taken away from process (without causing ill effect). For example, CD resources are not preemptable at an arbitrary moment. Reallocating resources can resolve deadlocks that involve preemptable resources. Deadlocks that involve nonpreemptable resources are difficult to deal with.

4. Resource Allocation Graph (RAG)

Resource allocation graph is explained as what is the state of the system in terms of processes and resources. Like how many resources are available, how many are allocated and what is the

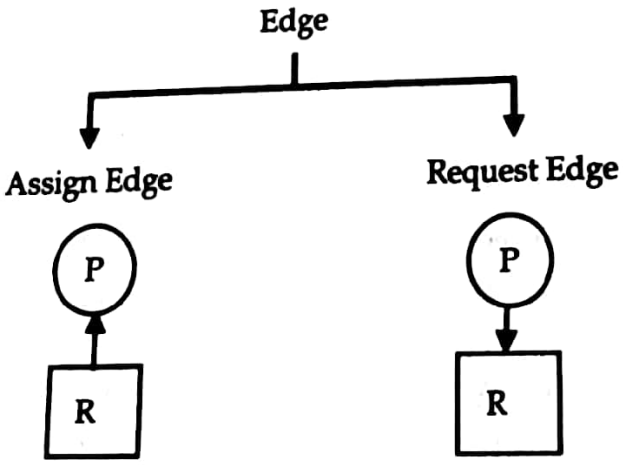
request of each process. Everything can be represented in terms of the diagram. One of the advantages of having a diagram is, sometimes it is possible to see a deadlock directly by using RAG, but then it might not be able to know that by looking at the table. But the tables are better if the system contains lots of process and resource and Graph is better if the system contains less number of process and resource. We know that any graph contains vertices and edges. So RAG also contains vertices and edges. In RAG vertices are two types:

1. **Process vertex** – Every process will be represented as a process vertex. Generally, the process will be represented with a circle.
2. **Resource vertex** – Every resource will be represented as a resource vertex. It is also two type
   - **Single instance type resource** – It represents as a box, inside the box, there will be one dot. So the number of dots indicates how many instances are present of each resource type.
   - **Multi-resource instance type resource** – It also represents as a box, inside the box, there will be many dots present.



There are two types of edges in RAG

1. **Assign Edge:** If you already assign a resource to a process then it is called Assign edge.
2. **Request Edge:** It means in future the process might want some resource to complete the execution that is called request edge.



So, if a process is using a resource, an arrow is drawn from the resource node to the process node. If a process is requesting a resource, an arrow is drawn from the process node to the resource node.

Example 1 (Single instances RAG)

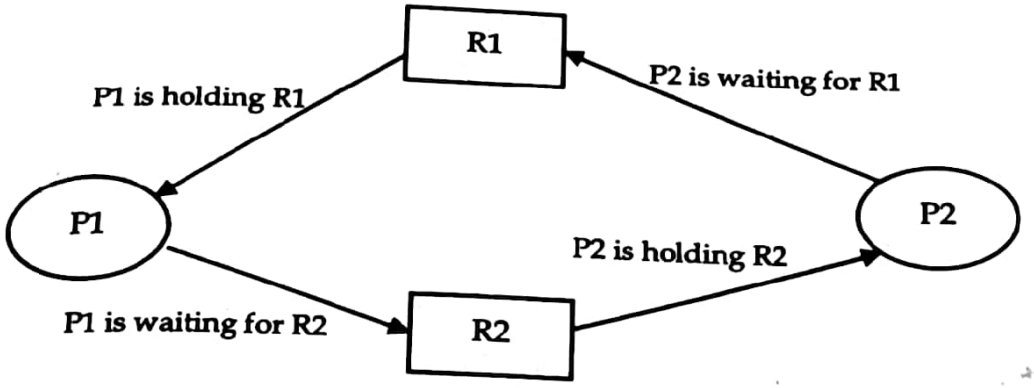


Fig 3.2: Single Instance Resource Type with Deadlock

If there is a cycle in the Resource Allocation Graph and each resource in the cycle provides only one instance, then the processes will be in deadlock. For example, if process P1 holds resource R1, process P2 holds resource R2 and process P1 is waiting for R2 and process P2 is waiting for R1, then process P1 and process P2 will be in deadlock.

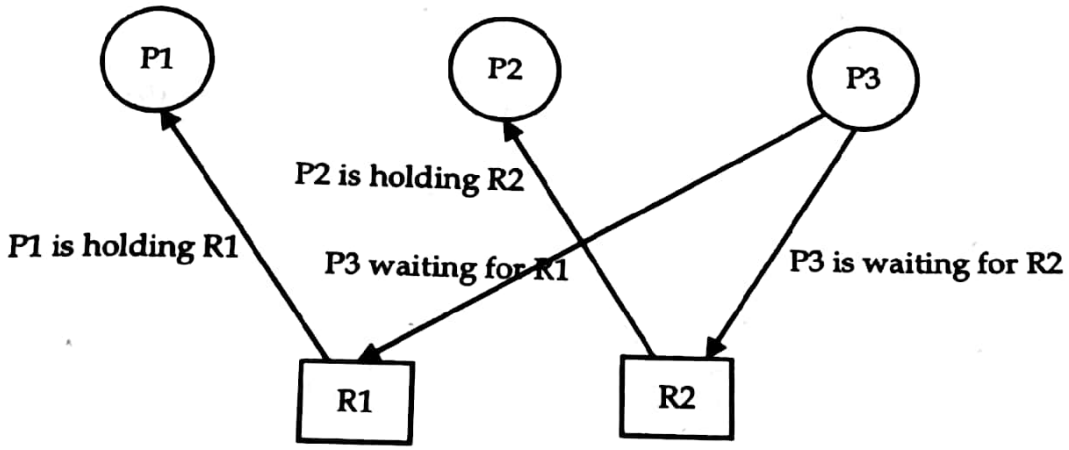


Fig 3.3: Single instance resource type without deadlock

Here is another example that shows Processes P1 and P2 acquiring resources R1 and R2 while process P3 is waiting to acquire both resources. In this example, there is no deadlock because there is no circular dependency. So cycle in single-instance resource type is the sufficient condition for deadlock.

Example 2 (Multi-instances RAG)

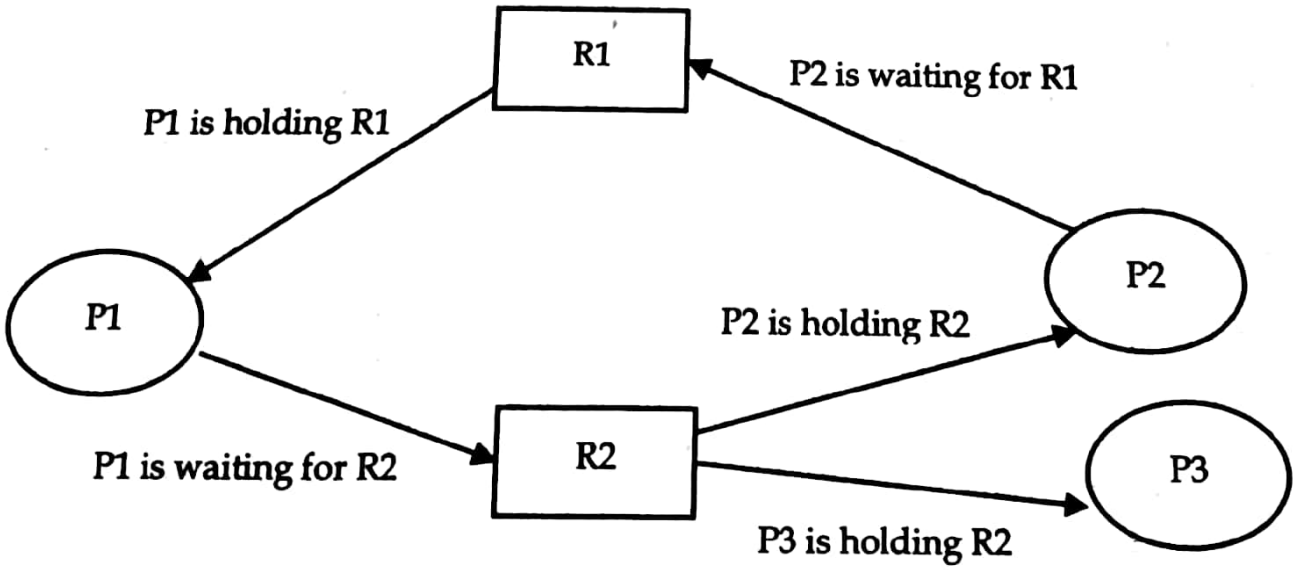


Fig 3.4: Multi Instances without Deadlock

From the above example, it is not possible to say the RAG is in a safe state or in an unsafe state. So to see the state of this RAG, let's construct the allocation matrix and request matrix.

Process	Allocation Resources		Request Resources	
	R1	R2	R1	R2
P1	1	0	0	1
P2	0	1	1	0
P3	0	1	0	0

- The total number of processes are three; P1, P2 & P3 and the total number of resources are two; R1 & R2.

Allocation matrix

- For constructing the allocation matrix, just go to the resources and see to which process it is allocated.
- R1 is allocated to P1, therefore write 1 in allocation matrix and similarly, R2 is allocated to P2 as well as P3 and for the remaining element just write 0.

Request matrix

- In order to find out the request matrix, you have to go to the process and see the outgoing edges.
- P1 is requesting resource R2, so write 1 in the matrix and similarly, P2 requesting R1 and for the remaining element write 0.So now available resource is = (0, 0).

5. Checking Deadlock (Safe or not)

Available = [0 0] (As P3 does not require any extra resources to complete the execution and after completion P3 [0 1] P3 release its own resources)

New Available = [0 0] (As using new available resource we can satisfy the requirement of process P1 and P1 also P1 [1 0] release its previous resource)

New Available = [1 1] (Now easily we can satisfy the requirement of Process P2)

P2 [0 1]

New Available = [1 2]

So, there is no deadlock in this RAG. Even though there is a cycle, still there is no deadlock. Therefore, in multi-instance resource cycle is not sufficient condition for deadlock.

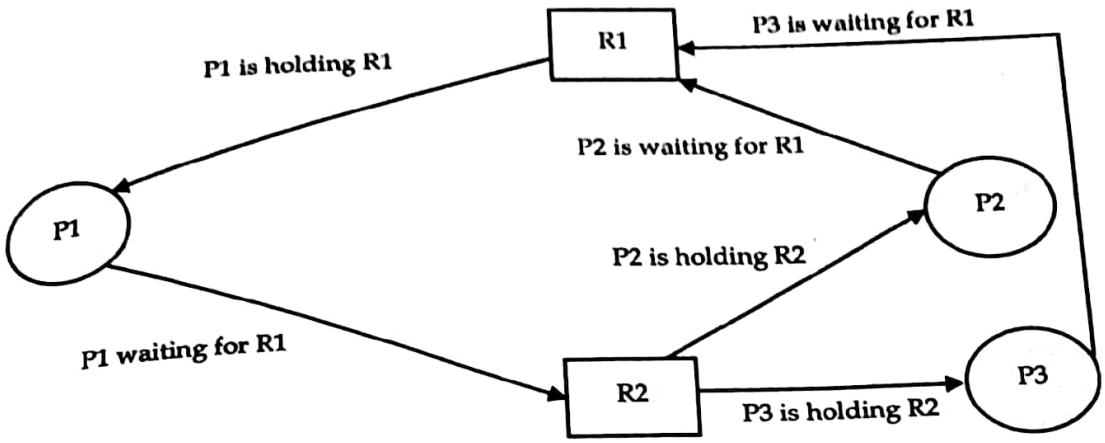


Fig 3.5: Multi Instances with deadlock

Above example is the same as the previous example except that, the process P3 requesting for resource R1.So the table becomes as shown in below.

Process	Allocation		Request	
	Resources		Resources	
	R1	R2	R1	R2
P1	1	0	0	1
P2	0	1	1	0
P3	0	1	1	0

So, the Available resource is = (0, 0), but requirement are (0, 1), (1, 0) and (1, 0).So you can't fulfill any one requirement. Therefore, it is in deadlock. Therefore, every cycle in a multi-instance resource type graph is not a deadlock, if there has to be a deadlock, there has to be a cycle. So, in case of RAG with multi-instance resource type, the cycle is a necessary condition for deadlock, but not sufficient.

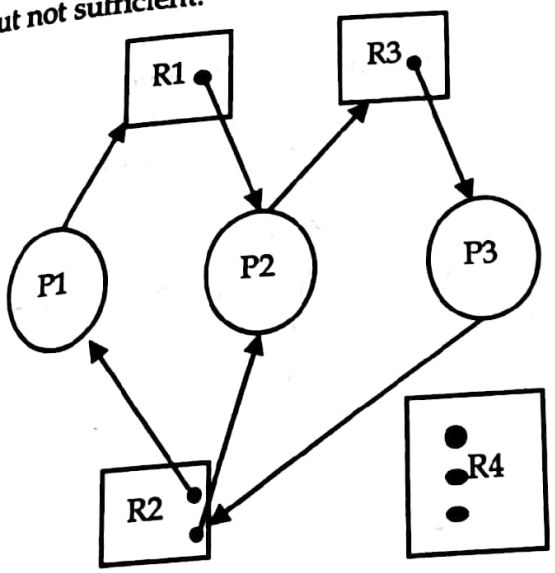


Fig 3.6: Resource allocation Graph with a deadlock

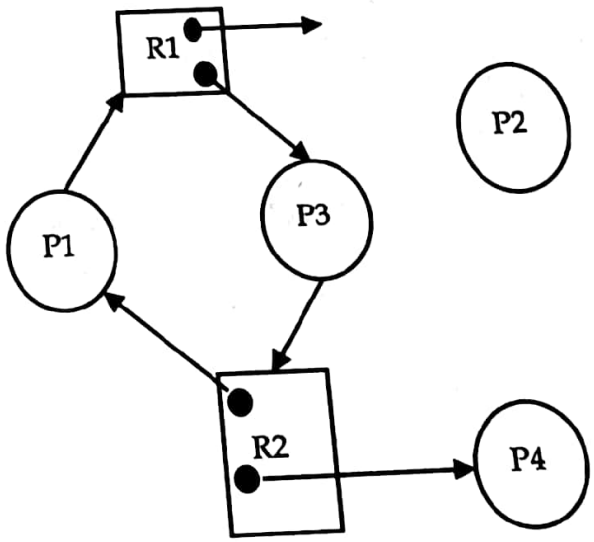


Fig 3.7: Resource allocation Graph with a cycle but no deadlock

Note: If each resource has only one instance, the cycle always causes the deadlock, if each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. In the graph represented on right P4 may release its instance of resource type R2 and P3 is allowed breaking the cycle.

6. Dealing with Deadlock Problem

In general, there are four strategies of dealing with deadlock problem:

6.1 The Ostrich Approach

The ostrich algorithm is a strategy of ignoring potential problems on the basis that they may be exceedingly rare. It is named for the ostrich effect which is defined as "to stick one's head in the sand and pretend there is no problem". It is used when it is more cost-effective to allow the problem to occur than to attempt its prevention. This approach may be used in dealing with deadlocks in concurrent programming if they are believed to be very rare and the cost of detection or prevention is high. For example, if each PC deadlocks once per 10 years, the one reboot may be less painful than the restrictions needed to prevent it. The ostrich algorithm pretends there is no problem and is reasonable to use if deadlocks occur very rarely and the cost of their prevention would be high. The UNIX and Windows operating systems take this approach.

6.2 Deadlock Detection

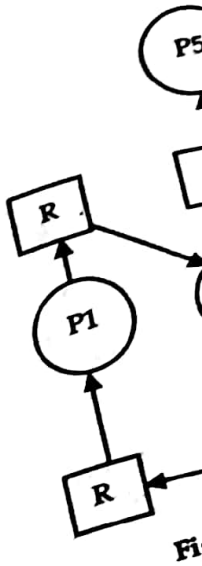
Deadlock detection is the process of actually determining that a deadlock exists and identifying the processes and resources involved in the deadlock. The basic idea is to check allocation against resource availability for all possible allocation sequences to determine if the system is in deadlocked state a. Of course, the deadlock detection algorithm is only half of this strategy. Once a deadlock is detected, there needs to be a way to recover several alternatives exists:

- Temporarily prevent resources from deadlocked processes.
- Back off a process to some check point allowing preemption of a needed resource and restarting the process at the checkpoint later.
- Successively kill processes until the system is deadlock free.

These methods are expensive in the sense that each iteration calls the detection algorithm until the system proves to be deadlock free. The complexity of algorithm is $O(N^2)$ where N is the number of proceeds. Another potential problem is starvation; same process killed repeatedly.

Single Instance of each resource type

If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a wait-for graph. We obtain this graph from the resource-allocation graph by removing the nodes of type resource and collapsing the appropriate edges.



Several Instan
Wait for grap
resource type
• Eac
• As
th
• W

6.3 Dead
We can

Elimin
The m
canno
some
shar
be i

El
T
F

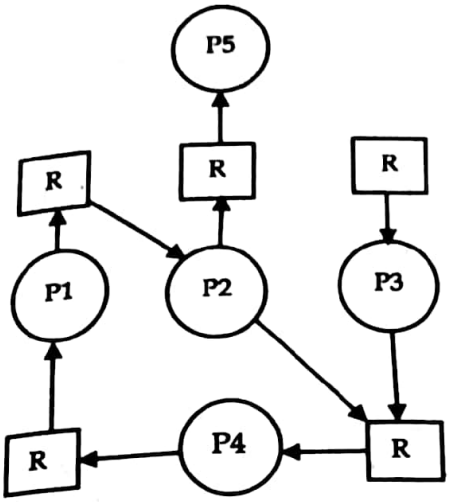


Fig 3.8: Wait for graph

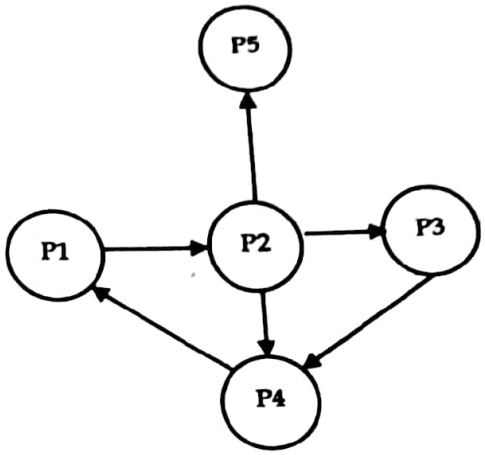


Fig 3.9: Resource allocation Graph

Several Instances of Resource Type

Wait for graph is not applicable to a resource allocation system, when multiple instances of each resource type are available.

- Each process is initially said to be unmarked.
- As an algorithm process will be marked, indicating that they are able to complete and thus not deadlocked.
- When algorithm terminates, any unmarked process is known to be deadlocked.

6.3 Deadlock Prevention

We can prevent Deadlock by eliminating any of the above four condition.

Elimination of "Mutual Exclusion" Condition

The mutual exclusion condition must hold for non-sharable resources. That is, several processes cannot simultaneously share a single resource. This condition is difficult to eliminate because some resources, such as the tap drive and printer, are inherently non-shareable. Note that shareable resources like read-only-file do not require mutually exclusive access and thus cannot be involved in deadlock.

Elimination of "Hold and Wait" Condition

There are two possibilities for elimination of the second condition. The first alternative is that a process request be granted all of the resources it needs at once, prior to execution. The second alternative is to disallow a process from requesting resources whenever it has previously allocated resources. This strategy requires that all of the resources a process will need must be requested at once. The system must grant resources on "all or none" basis. If the complete set of resources needed by a process is not currently available, then the process must wait until the complete set is available. While the process waits, however, it may not hold any resources. Thus the "wait for" condition is denied and deadlocks simply cannot occur. This strategy can lead to serious waste of resources. For example, a program requiring ten tap drives must request and receive all ten derives before it begins executing. If the program needs only one tap drive to begin

execution and then does not need the remaining tap drives for several hours. Then substantial computer resources (9 tape drives) will sit idle for several hours. This strategy can cause indefinite postponement (starvation). Since not all the required resources may become available at once.

Elimination of "No-preemption" Condition

The non-preemption condition can be alleviated by forcing a process waiting for a resource that cannot immediately be allocated to relinquish all of its currently held resources, so that other processes may use them to finish. Suppose a system does allow processes to hold resources while requesting additional resources. Consider what happens when a request cannot be satisfied. A process holds resources a second process may need in order to proceed while second process may hold the resources needed by the first process. This is a deadlock. This strategy requires that when a process that is holding some resources is denied a request for additional resources. The process must release its held resources and, if necessary, request them again together with additional resources. Implementation of this strategy denies the "no-preemptive" condition effectively **high Cost** when a process release resources the process may lose all its work to that point. One serious consequence of this strategy is the possibility of indefinite postponement (starvation). A process might be held off indefinitely as it repeatedly requests and releases the same resources.

Elimination of "Circular Wait" Condition

The last condition, the circular wait, can be denied by imposing a total ordering on all of the resource types and then forcing, all processes to request the resources in order (increasing or decreasing). This strategy imposes a total ordering of all resources types, and to require that each process requests resources in a numerical order (increasing or decreasing) of enumeration. With this rule, the resource allocation graph can never have a cycle. For example, provide a global numbering of all the resources, as shown

1 ≡ Card reader

2 ≡ Printer

3 ≡ Plotter

4 ≡ Tape drive

5 ≡ Card punch

Now the rule is this: processes can request resources whenever they want to, but all requests must be made in numerical order. A process may request first printer and then a tape drive (order: 2, 4), but it may not request first a plotter and then a printer (order: 3, 2). The problem with this strategy is that it may be impossible to find an ordering that satisfies everyone.

6.4 Deadlock Avoidance

This approach to the deadlock problem anticipates deadlock before it actually occurs. This approach employs an algorithm to access the possibility that deadlock could occur and acting accordingly. This method differs from deadlock prevention, which guarantees that deadlock cannot occur by denying one of the necessary conditions of deadlock. If the necessary conditions for a deadlock are in place, it is still possible to avoid deadlock by being careful when resources are allocated. So named because the process is analogous to that used by a banker in deciding if a loan can be safely made.

6.5 Banker's Algorithm

The Banker algorithm, sometimes referred to as the detection algorithm, is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation of predetermined maximum possible amounts of all resources, and then makes an "s-state" check to test for possible deadlock conditions for all other pending activities, before deciding whether allocation should be allowed to continue. Following **data structures** are used to implement the Banker's Algorithm:

Let 'n' be the number of processes in the system and 'm' be the number of resources types.

Available

- It is a one dimensional array of size 'm' indicating the number of available resources of each type.
- Available[j] = k means there are 'k' instances of resource type R_j

Max

- It is a 2-dimensional array of size 'n*m' that defines the maximum demand of each process in a system.
- Max[i, j] = k means process P_i may request at most 'k' instances of resource type R_j

Allocation

- It is a 2-dimensional array of size 'n*m' that defines the number of resources of each type currently allocated to each process.
- Allocation[i, j] = k means process Pi is currently allocated 'k' instances of resource type Rj

Need

- It is a 2-dimensional array of size 'n*m' that indicates the remaining resource need of each process.
- Need[i, j] = k means process Pi currently need 'k' instances of resource type Rj for its execution.
- Need[i, j] = Max[i, j] – Allocation[i, j]

For the Banker's algorithm to work, it to three things

- How much of each resource each process could possibly request [MAX]
- How much of each resource each process is currently holding [ALLOCATED]
- How much of each resource the system currently has available [AVAILABLE]

Resources may be allocated to a process only if the amount of resources requested is less than or equal to the amount available; otherwise, the process waits until resources are available.

In this analogy

Customers ≡ Processes

Units ≡ resources, say, tape drive

Banker ≡ Operating System

Customers	Used	Max	
A	0	6	Available Units = 10
B	0	5	
C	0	4	
D	0	7	

Table 1

In the above table, we see four customers each of whom has been granted a number of credit units. The banker reserved only 10 units rather than 22 units to service them. At certain moment, the situation becomes

Customers	Used	Max	
A	1	6	Available Units = 2
B	1	5	
C	2	4	
D	4	7	

Table 2

Safe State

The key to a state being safe is that there is at least one way for all users to finish. In other analogy, the state of Table 2 is safe because with 2 units left, the banker can delay any request except *C's*, thus letting *C* finish and release all four resources. With four units in hand, the banker can let either *D* or *B* have the necessary units and so on.

Unsafe State

Consider what would happen if a request from B for one more unit were granted in above Table 2. We would have following situation

Customers	Used	Max	
A	1	6	Available Units = 1
B	2	5	
C	2	4	
D	4	7	

Table 3

This is an unsafe state.

If all the customers namely *A, B, C,* and *D* asked for their maximum loans, then banker could not satisfy any of them and we would have a deadlock.

Numerical problem 1: Assume that there are 5 processes, P0 through P4, and 4 types of resources. At T0 we have the following system state:

	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P0	0	1	1	0	0	2	1	0	1	5	2	0
P1	1	2	3	1	1	6	5	2				
P2	1	3	6	5	2	3	6	6				
P3	0	6	3	2	0	6	5	2				
P4	0	0	1	4	0	6	5	6				

1. Create the need matrix (max-allocation)

	Need matrix(max-allocation)			
	A	B	C	D
P0	0	1	0	0
P1	0	4	2	1
P2	1	0	0	1
P3	0	0	2	0
P4	0	6	4	2

2. Use the safety algorithm to test if the system is in a safe state.
We will first define work and finish:

Work vector	Finish matrix	
1	P0	FALSE
5	P1	FALSE
2	P2	FALSE
0	P3	FALSE
	P4	FALSE

Check to see if need0 for process P0 is (0, 1, 0, 0) is less than or equal to work. It is, so let's set finish to true for that process and also update work by adding the allocated resources (0, 1, 1, 0) for that process to work.

Work vector	Finish matrix	
1	P0	TRUE
6	P1	FALSE
3	P2	FALSE
0	P3	FALSE
	P4	FALSE

Now, let's check to see if need1 (0, 4, 2, 1) <= work. Remember that we have to check each element of the vector need1 against the corresponding element in work. Because 1 is not less than 0 (the fourth element), we need to move on to P2.

Need2 (1, 0, 0, 1) is not less than work, so must move on to P3.

Need3 (0, 0, 2, 0) is less than work, so we can update work and finish.

Work vector	Finish matrix	
1	P0	TRUE
12	P1	FALSE
6	P2	FALSE
2	P3	TRUE
	P4	FALSE

Next, let's look at P4. Need4 (0, 6, 4, 2) is less than work, so we can update work and finish as follows:

Work vector	Finish matrix	
1	P0	TRUE
12	P1	FALSE
7	P2	FALSE
6	P3	TRUE
	P4	TRUE

Now we can go back up to P1. Need1 (0, 4, 2, 1) is less than work, so let's update work and finish.

Work vector	Finish matrix	
2	P0	TRUE
14	P1	TRUE
10	P2	FALSE
7	P3	TRUE
	P4	TRUE

Finally, let's look at P2. Need2 (1, 0, 0, 1) is less than work, so we can then say that the system is in a safe state and the processes will be executed in the following order:

{P0, P3, P4, P1, P2}

3. If the system is in a safe state, can the following requests be granted, why or why not? Please also run the safety algorithm on each request as necessary.

   a) P1 requests (2, 1, 1, 0)

   We cannot grant this request, because we do not have enough available instances of resource A.

   b) P1 requests (0, 2, 1, 0)

   There are enough available instances of the requested resources, so first let's pretend to accommodate the request and see what the system looks like:

	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P0	0	1	1	0	0	2	1	0	1	3	1	0
P1	1	4	4	1	1	6	5	2				
P2	1	3	6	5	2	3	6	6				
P3	0	6	3	2	0	6	5	2				
P4	0	0	1	4	0	6	5	6				

Similarly calculate need matrix as

	Need matrix(max-allocation)			
	A	B	C	D
P0	0	1	0	0
P1	0	2	1	1
P2	1	0	0	1
P3	0	0	2	0
P4	0	6	4	2

Now we need to run the safety algorithm:

Work vector	Finish matrix	
1	P0	FALSE
3	P1	FALSE
1	P2	FALSE
0	P3	FALSE
	P4	FALSE

Let's first look at P0. Need0 (0, 1, 0, 0) is less than work, so we change the work vector and finish matrix as follows:

Work vector	Finish matrix	
1	P0	TRUE
4	P1	FALSE
2	P2	FALSE
0	P3	FALSE
	P4	FALSE

Need1 (0, 2, 1, 1) is not less than work, so we need to move on to P2.

Need2 (1, 0, 0, 1) is not less than work, so we need to move on to P3.

Need3 (0, 0, 2, 0) is less than or equal to work. Let's update work and finish:

Work vector	Finish matrix	
1	P0	TRUE
10	P1	FALSE
5	P2	FALSE
2	P3	TRUE
	P4	FALSE

Let's take a look at Need4 (0, 6, 4, 2). This is less than work, so we can update work and finish:

Work vector	Finish matrix	
1	P0	TRUE
10	P1	FALSE
6	P2	FALSE
6	P3	TRUE
	P4	TRUE

We can now go back to P1. Need1 (0, 2, 1, 1) is less than work, so work and finish can be updated:

Work vector	Finish matrix	
1	P0	TRUE
14	P1	FALSE
10	P2	FALSE
7	P3	TRUE
	P4	TRUE

Finally, Need2 (1, 0, 0, 1) is less than work, so we can also accommodate this. Thus, the system is in a safe state when the processes are run in the following order: {P0, P3, P4, P1, P2}. We therefore can grant the resource request.

Numerical problem 2: Assume that there are three resources, A, B, and C. There are 4 processes P0 to P3. At T0 we have the following snapshot of the system:

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P0	1	0	1	2	1	1	2	1	1
P1	2	1	2	5	4	4			
P2	3	0	0	3	1	1			
P3	1	0	1	1	1	1			

1. Create the need matrix (max-allocation)

	Need matrix(max-allocation)		
	A	B	C
P0	1	1	0
P1	3	3	2
P2	0	1	1
P3	0	1	0

2. Is the system in a safe state? Why or why not?

In order to check this, we should run the safety algorithm. Let's create the work vector and finish matrix:

Work vector	Finish matrix	
2	P0	FALSE
1	P1	FALSE
1	P2	FALSE
	P3	FALSE

Need0 (1, 1, 0) is less than work, so let's go ahead and update work and finish:

Work vector	Finish matrix	
3	P0	TRUE
1	P1	FALSE
2	P2	FALSE
	P3	FALSE

Need1 (3, 3, 2) is not less than work, so we have to move on to P2.
Need2 (0, 1, 1) is less than work, let's update work and finish:

Work vector	Finish matrix	
6	P0	TRUE
1	P1	FALSE
2	P2	TRUE
	P3	FALSE

Need3 (0, 1, 0) is less than work, we can update work and finish:

Work vector	Finish matrix	
7	P0	TRUE
1	P1	FALSE
3	P2	TRUE
	P3	TRUE

We now need to go back to P1. Need1 (3, 3, 2) is not less than work, so we cannot continue. Thus, the system is not in a safe state.

Problem with Banker's Algorithm

- An algorithm requires fixed number of resources; some processes dynamically change the number of resources.
- An algorithm requires the number of resources in advance; it is very difficult to predict the resources in advanced.
- Algorithms predict all process returns within finite time, but the system does not guarantee it.

It is important to note that an unsafe state does not imply the existence or even the eventual existence a deadlock. What an unsafe state does imply is simply that some unfortunate sequence

of events might lead to a deadlock. The Banker's algorithm is thus to consider each request as it occurs, and see if granting it leads to a safe state. If it does, the request is granted, otherwise, it postponed until later.

7. Deadlock Recovery

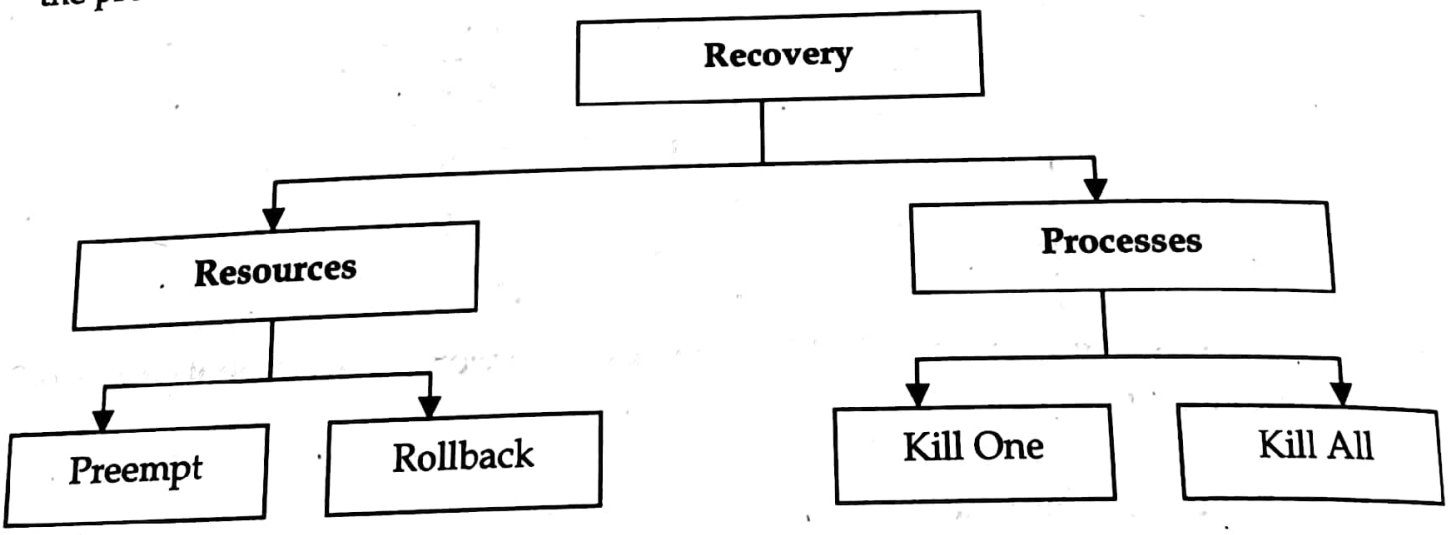
Traditional operating system such as Windows doesn't deal with deadlock recovery as it is time and space consuming process. Real time operating systems use Deadlock recovery. Some of the common recovery methods are

For Resource

- **Preempt the resource:** We can snatch one of the resources from the owner of the resource (process) and give it to the other process with the expectation that it will complete the execution and will release this resource sooner. Well, choosing a resource which will be snatched is going to be a bit difficult.
- **Rollback to a safe state:** System passes through various states to get into the deadlock state. The operating system can roll back the system to the previous safe state. For this purpose, OS needs to implement check pointing at every state. The moment, we get into deadlock, we will rollback all the allocations to get into the previous safe state.

For Process

- **Kill a process:** Killing a process can solve our problem but the bigger concern is to decide which process to kill. Generally, operating system kills a process which has done least amount of work until now.
- **Kill all process:** This is not a suggestible approach but can be implemented if the problem becomes very serious. Killing all process will lead to inefficiency in the system because all the processes will execute again from starting.



Laboratory Works

Program 1: Implementing deadlock detection algorithm

```
#include<stdio.h>
static int mark[20];
int i, j, np, nr;
int main()
{
        int alloc[10][10], request[10][10], avail[10], r[10], w[10];
        printf("\n Enter the no of process: ");
        scanf("%d", &np);
        printf("\n Enter the no of resources: ");
        scanf("%d", &nr);
        for(i=0; i<nr; i++)
        {
                printf("\n Total Amount of the Resource R %d: ",i+1);
                scanf("%d", &r[i]);
        }
        printf("\n Enter the request matrix:");
        for(i=0; i<np; i++)
        for(j=0; j<nr; j++)
        scanf("%d", &request[i][j]);
        printf("\n Enter the allocation matrix:");
        for(i=0; i<np; i++)
        for(j=0; j<nr; j++)
        scanf("%d", &alloc[i][j]);
        for(j=0; j<nr; j++)  /*Available Resource calculation*/
        {
                avail[j]=r[j];
                for(i=0; i<np; i++)
                {
                        avail[j] - = alloc[i][j];
                }
        }
        for(i=0; i<np; i++) //marking processes with zero allocation
        {
                int count=0;
                for(j=0; j<nr; j++)
                {
                        if(alloc[i][j]==0)
                                count++;
                        else
                                break;
                }
```

```
            if(count==nr)
                mark[i]=1;
        }
        // initialize w with avail
        for(j=0; j<nr; j++)
            w[j]=avail[j];
        //mark processes with request less than or equal to W
        for(i=0; i<np; i++)
        {
            int canbeprocessed=0;
            if(mark[i]!=1)
            {
                for(j=0; j<nr; j++)
                {
                    if(request[i][j]<=w[j])
                        canbeprocessed=1;
                    else
                    {
                        canbeprocessed=0;
                        break;
                    }
                }
                if(canbeprocessed)
                {
                    mark[i]=1;
                    for(j=0; j<nr; j++)
                        w[j]+=alloc[i][j];
                }
            }
        }
    //checking for unmarked processes
    int deadlock=0;
    for(i=0; i<np; i++)
    if(mark[i]!=1)
    deadlock=1;
    if(deadlock)
        printf("\n Deadlock detected");
    else
        printf("\n No Deadlock possible");
}
```

Output

```
Enter the no of process: 4
Enter the no of resources: 5
Total Amount of the Resource R1: 2
Total Amount of the Resource R2: 1
Total Amount of the Resource R3: 1
Total Amount of the Resource R4: 2
Total Amount of the Resource R5: 1
Enter the request matrix:0 1 0 0 1
0 0 1 0 1
0 0 0 0 1
1 0 1 0 1
Enter the allocation matrix:1 0 1 1 0
1 1 0 0 0
0 0 0 1 0
0 0 0 0 0
Deadlock detected.
```

Exercises

1. What is deadlock? State the conditions necessary for deadlock to exist. Give reason, why all conditions are necessary.
2. Define the term indefinite postponement. How does it differ from deadlock?
3. A system has two processes and three identical resources. Each process needs a maximum of two resources. Is deadlock possible? Explain your answer.
4. Write a short note on deadlock?
5. Explain the characteristic of deadlock?
6. Describe various methods for deadlock prevention?
7. Explain the resource allocation graph?
8. Write a note on safe state?
9. What are the necessary conditions for deadlocks?
10. Explain how deadlocks are detected and corrected?
11. What are the difference between a deadlock prevention and deadlock Avoidance?
12. Explain the Resource allocation graph in detail.
13. Students working at individual PCs in a computer laboratory send their files to be printed by a server which spools the files on its hard disk. Under what conditions may a deadlock occur if the disk space for the print spool is limited? How may the deadlock be avoided?
14. Explain the concept of Deadlock prevention in detail
15. In the preceding question which resources are preemptable and which are nonpreemptable?
16. The discussion of the ostrich algorithm mentions the possibility of process table slots or other system tables filling up. Can you suggest a way to enable a system administrator to recover from such a situation?

17. In theory, resource trajectory graphs could be used to avoid deadlocks. By clever scheduling, the operating system could avoid unsafe regions. Suggest a practical problem with actually doing this. Can a system be in a state that is neither deadlocked nor safe? If so, give an example. If not, prove that all states are either deadlocked or safe.

18. A system has two processes and three identical resources. Each process needs a maximum of two resources. Is deadlock possible? Explain your answer.

19. A system has four processes and five allocatable resources. The current allocation and maximum needs are as follows:

	Allocated	Maximum	Available
Process A	10211	11213	00x11
Process B	20110	22210	
Process C	11010	21310	
Process D	11110	11221	

What is the smallest value of x for which this is a safe state?

20. Two processes, A and B, each need three records, 1, 2 and 3, in a database. If A asks for them in the order 1, 2, 3, and B asks for them in the same order, deadlock is not possible. However, if B asks for them in the order 3, 2, 1, then deadlock is possible. With three resources, there are 3! Or 6 possible combinations each process can request the resources. What fraction of all the combinations is guaranteed to be deadlock free?

❑❑❑

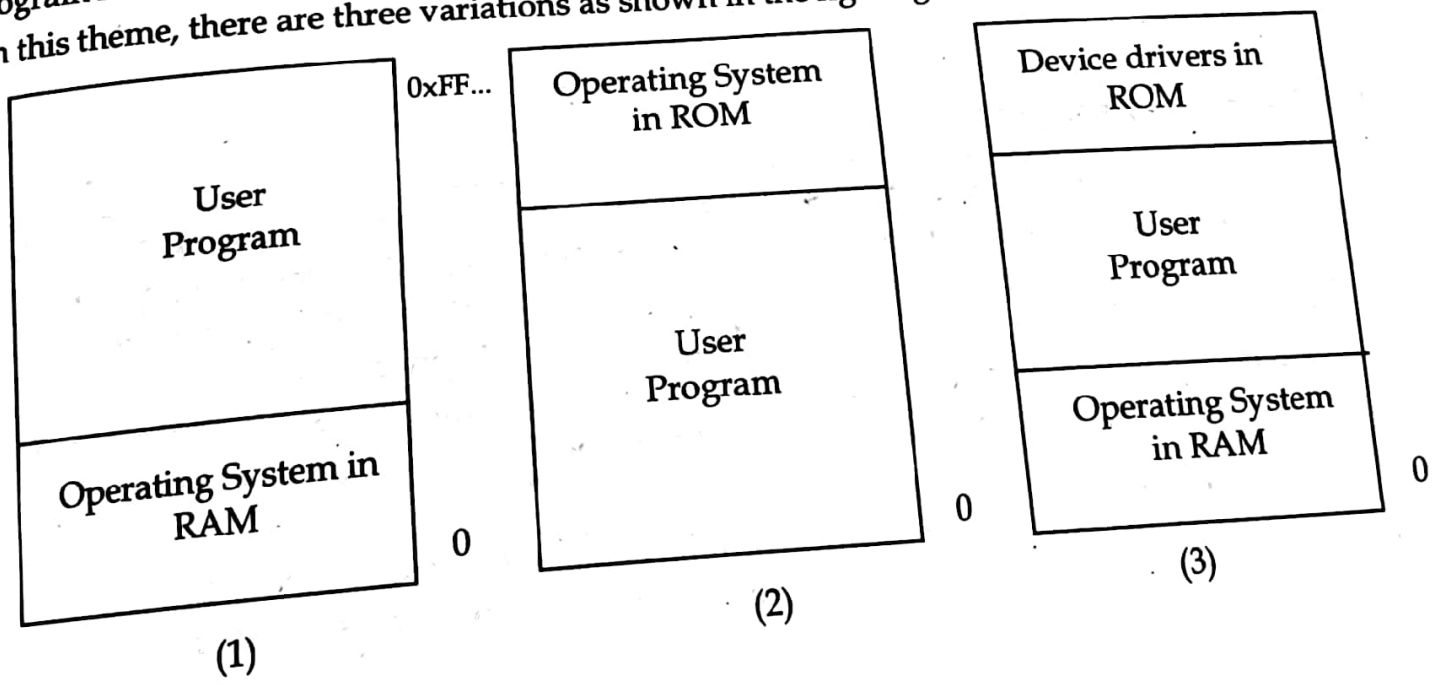
Chapter 4
MEMORY MANAGEMENT

1. Introduction

Memory is central to the operation of a modern computer system. Memory consists of a large array of words or bytes, each with its own address. The CPU fetches instructions from memory according to the value of the program counter. These instructions may cause additional loading from and storing to specific memory addresses. A typical instruction-execution cycle, for example, first fetches an instruction from memory. The instruction is then decoded and may cause operands to be fetched from memory. After the instruction has been executed on the operands, results may be stored back in memory. The memory unit sees only a stream of memory addresses; it does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, and so on) or what they are for (instructions or data). Accordingly, we can ignore how a memory address is generated by a program. We are interested in only the sequence of memory addresses generated by the running program.

2. Mono-programming

In mono-programming, memory contains only one program at any point of time. In case of mono-programming, when CPU is executing the program and I/O operation is encountered then the program goes to I/O devices, during that time CPU sits idle. Thus, in mono-programming CPU is not effectively used i.e. CPU utilization is poor. In mono-programming, most of the memory capacity is dedicated to a single program; only a small part is needed to hold the operating system. In this configuration, the whole program is in memory for execution. When the program finishes running, the program area is occupied by another program.
On this theme, there are three variations as shown in the figure given below:



As shown in the first figure, the OS may be at the bottom of the memory in RAM. Here RAM stands for Random Access Memory. And as shown in the second figure given above, the OS may be at the top of the memory in ROM. Here ROM stands for Read Only Memory. And as shown in the last or third figure given above, the device drivers may be at the top of the memory in a Read Only Memory (ROM), and rest of the system in RAM lies below the device drivers.

The first model is rarely used now-a-day, but was formally used on mainframes and minicomputers. The second model is used on few palmtop computers and embedded systems. And the third model was used by early PC, for example MS-DOS. Here PC stands for Personal Computer and MS-DOS stands for Microsoft-Disk Operating System. In that model, BIOS was the portion of the system in Read Only Memory (ROM). Here BIOS stands for Basic I/O System. Only one process can be running at a time when the system is organized in this way. As soon as the user types a command, the OS copies the requested program from disk to memory and then executes it. The OS displays a prompt character and waits for a new command when the process finishes. When it receives a command, then it loads a new program into memory, overwriting the first one.

Example of mono-programming

- Batch processing in old computers and mobiles
- The old operating system of computers
- Old mobile operating system

The characteristics of Uniprogramming are as follows:

- Uniprogramming allows only one program to be present in memory at a time.
- The resources are provided to the single program that is present in the memory at that time.
- Since only one program is loaded the size is small as well.

Question 1: *A computer has a mono-programming operating system. If the size of memory is 64 MB and the memory reserved part for the operating system is 4 MB, what is the maximum size of program that can be run by this computer?*

Answer: *64 – 4 = 60 MB*

Question 2: *Mono-programming operating system runs programs that on average need 10 microseconds access to the CPU and 70 microseconds access to the I/O devices. What percentage of time is the CPU idle?*

Answer: *70 / (70 + 10) × 100 = 87.5%*

3. Multi-programming

In multiprogramming, multiple programs reside in main memory (RAM) at a time. OS which handles multiple programs at a time is known as multiprogramming operating system. One processor or CPU can only run one process at a time. OS use context switching in main memory for running multiple programs. Context switching is to switch programs so all programs are given a suitable amount of time. OS can handle only a limited number of programs. If we run many programs on the computer or mobile then the computer becomes very slow or unresponsive.

Today, almost all computer systems allow multiple processes to run at the same time. When multiple processes running at the same time, means that when one process is blocked waiting for

input/output to finish, another one can use the CPU. Therefore, multiprogramming increases the central processing unit utilization. Now-a-day, both network servers and client machines have the ability to run multiple processes at the same time. To achieve multiprogramming, the simplest way is just to divide memory up into n partitions (normally unequal partitions).It can be implemented in following two ways:

- Dedicate partitions for each process (Absolute translation)
- Maintaining a single queue (Re-locatable translation)

3.1 Dedicate partitions for each process (Absolute translation)

Here separate input queue is maintained for each partition. Processes are translated with absolute assemblers and compilers to run only in specific partition. The main problems of this process is that if a process is ready to run and its partition is occupied then that process has to wait even if other partitions are available. Wastage of storage.

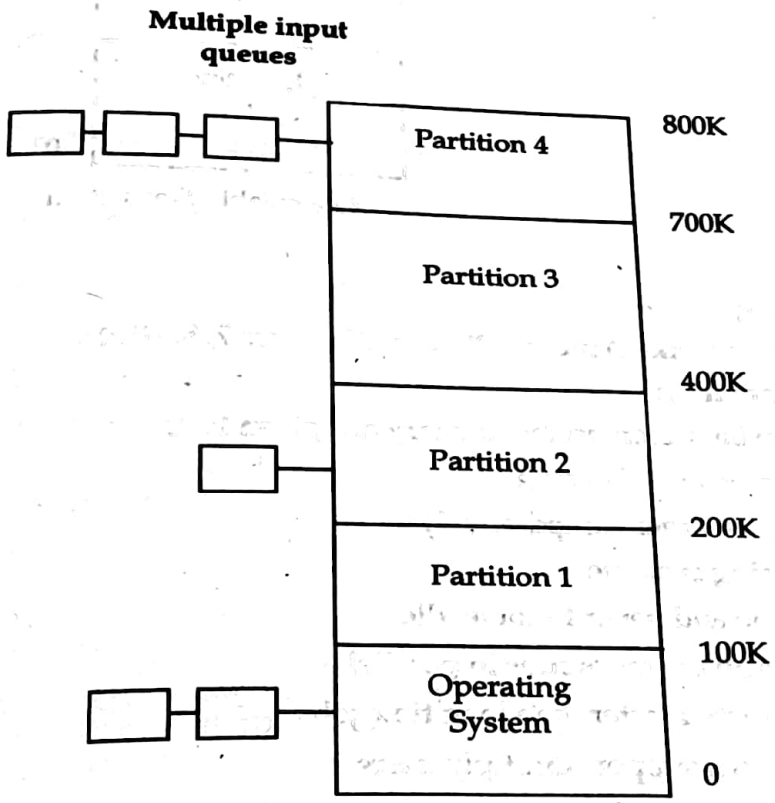


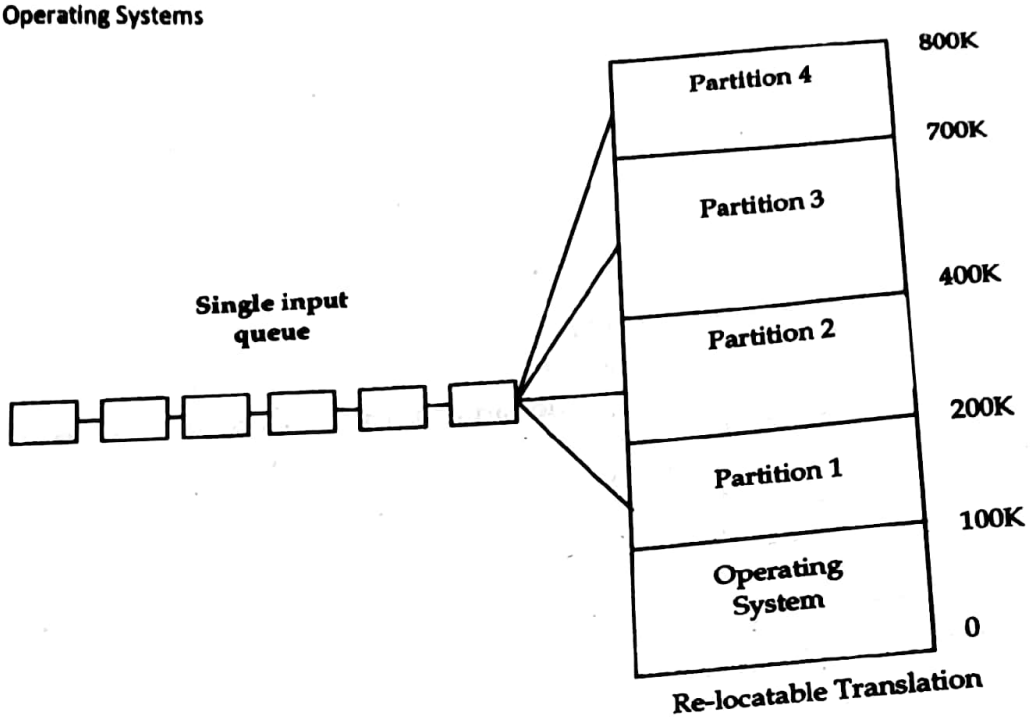
Fig 4.1: Absolute Translation

3.2 Maintaining a single queue (Re-locatable translation)

Here we maintain a single queue for all processes. When a partition becomes free, the processes closest to the front of queue that fits in it could be loaded into the empty partition and run. Not to waste the large partition for small process, another strategy to search the whole input queue whenever a partition becomes free and pick the largest process that fits.

Problems

- Eliminate the absolute problems but implementation is complex.
- Wastage of storage when many processes are small.



Example of multiprogramming

- Modern operating systems like Windows XP and Windows 7, 8, 10 etc.

Characteristics of multiprogramming

- Multiple programs can be present in the memory at a given time.
- The resources are dynamically allocated
- The size of the memory is larger comparatively

Advantages of multiprogramming systems

- CPU is used most of time and never become idle
- The system looks fast as all the tasks runs in parallel
- Short time jobs are completed faster than long time jobs
- Multiprogramming systems support multiply users
- Resources are used nicely
- Total read time taken to execute program/job decreases
- Response time is shorter
- In some applications multiple tasks are running and multiprogramming systems better handle these type of applications

Disadvantages of multiprogramming systems

- It is difficult to program a system because of complicated schedule handling
- Tracking all tasks/processes is sometimes difficult to handle
- Due to high load of tasks, long time jobs have to wait long

Mono-programming Vs. Multi-programming

In Uniprogramming only one program sits in main memory so it has a small size. But in the case of multiprogramming main memory needs more space. Uniprogramming system runs smoothly as only one task is run at a time. The slow processor can also work well in Uniprogramming but in multiprogramming processor needs to be fast. In multiprogramming large space of RAM is needed. Fixed size partition is used in Uniprogramming. Both fixed and variable size partition can be used in multiprogramming systems.

Example of Uniprogramming

- Batch processing in old computers and mobiles
- The old operating system of computers
- Old mobile operating system

Example of multiprogramming

- Modern operating systems like Windows XP and Windows 7,8,10

Question 1: A multiprogramming operating system uses paging. The available memory is60 MB divided into 15 frames, each of 4 MB. The first program needs 13 MB, the second program needs 12 MB, and the third program needs 27 MB.

a. How many frames are used by the first / second / third program?

P1 = 13/4= 3.4 = 4 frames

P2 = 12 / 4 = 3 frames

P3 = 27 / 4 = 6.75 = 7 frames

b. How many frames are un-used?

The used frames are 4 + 3 + 7 = 14 = 15- 14= 1

So the un-used frame is only one

c. What is the total memory used?

13+12+27= 52MB

d. What percentage of memory is used?

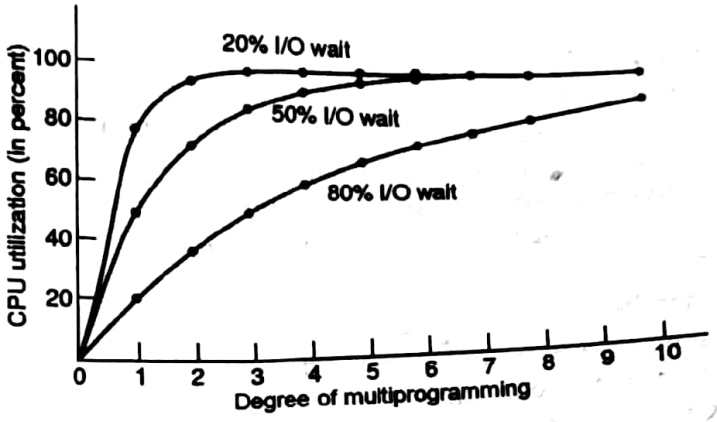
(52/ 60) × 100 = 86.67%

4. Modeling Multiprogramming

If we have five processes that use the processor twenty percent (20%) of the time (spending eighty percent doing I/O) then we should be able to achieve 100% CPU utilization. Of course, in reality, this will not happen as there may be times when all five processes are waiting for I/O. However, it seems reasonable that we will achieve better than 20% utilization that we would achieve with mono-programming.

We can build a model from a probabilistic viewpoint. Assume that a process spend p% of its time waiting for I/O. With n processes in memory the probability that all n processes are waiting for I/O (meaning the CPU is idle) is p^n. The CPU utilization is then given by;

$$\text{CPU utilization}=1-p^n$$



We can see that with an I/O wait time of 20%, almost 100% CPU utilization can be achieved with four processes. If the I/O waits time is 90% then with 10 processes, we only achieve just above 60% utilization. The important point is that, as we introduce more processes the CPU utilization raises.

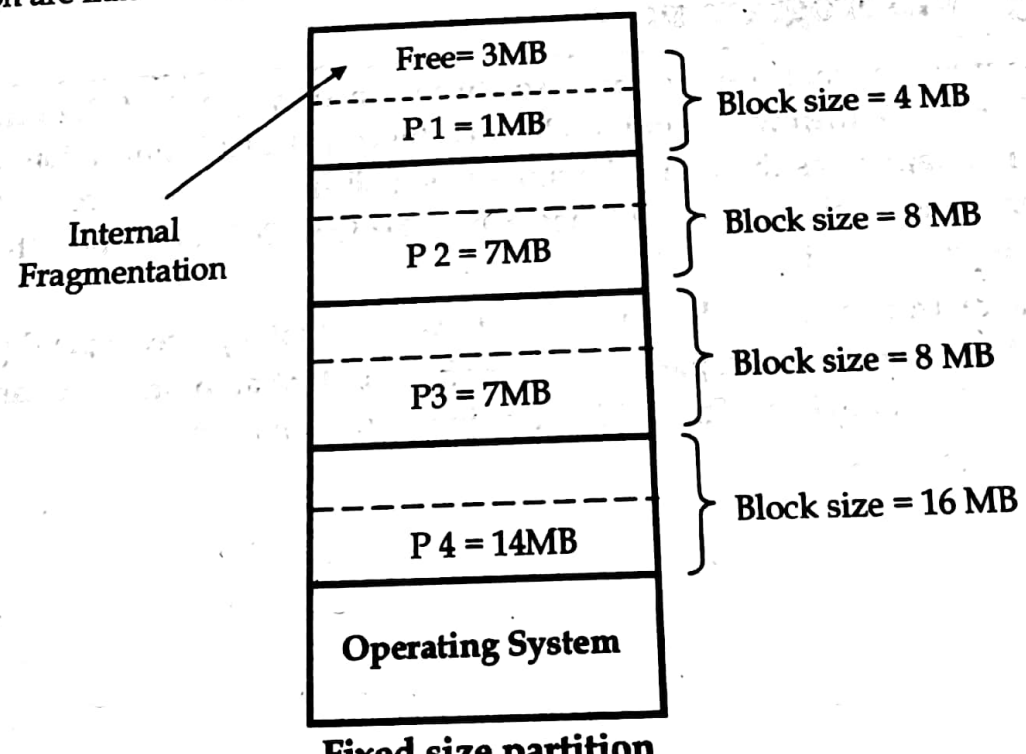
5. Multiprogramming with fixed and variable partitions

There are two Memory Management Techniques: Contiguous, and Non-Contiguous. In Contiguous Technique, executing process must be loaded entirely in main-memory. Contiguous Technique can be divided into:

- Fixed (or static) partitioning
- Variable (or dynamic) partitioning

5.1 Fixed (or static) partitioning

Dividing the main memory into a set of non overlapping blocks is known as fixed partition. This is the oldest and simplest technique used to put more than one processes in the main memory. In this partitioning, number of partitions (non-overlapping) in RAM is fixed but size of each partition may or may not be same. As it is contiguous allocation, hence no spanning is allowed. Here partition are made before execution or during system configure.



Fixed size partition

As illustrated in above figure, first process is only consuming 1MB out of 4MB in the main memory. Hence, Internal Fragmentation in first block is (4-1) = 3MB. Sum of Internal Fragmentation in every block = (4-1)+(8-7)+(8-7)+(16-14)= 3+1+1+2 = 7MB.

Advantages of Fixed Partitioning

- **Easy to implement:** Algorithms needed to implement Fixed Partitioning are easy to implement. It simply requires putting a process into certain partition without focusing on the emergence of Internal and External Fragmentation.
- **Little OS overhead:** Processing of Fixed Partitioning require lesser excess and indirect computational power.

Disadvantages of Fixed Partitioning

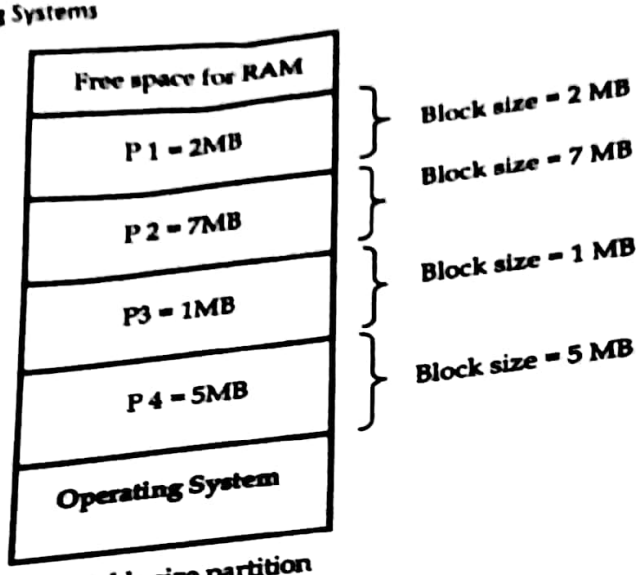
- **Internal Fragmentation:** Main memory use is inefficient. Any program, no matter how small, occupies an entire partition. This can cause internal fragmentation.
- **External Fragmentation:** The total unused space (as stated above) of various partitions cannot be used to load the processes even though there is space available but not in the contiguous form (as spanning is not allowed).
- **Limit process size:** Process of size greater than size of partition in Main Memory cannot be accommodated. Partition size cannot be varied according to the size of incoming process's size. Hence, process size of 32MB in above stated example is invalid.
- **Limitation on Degree of Multiprogramming:** Partition in Main Memory is made before execution or during system configure. Main Memory is divided into fixed number of partition. Suppose if there are n_1 partitions in RAM and n_2 are the number of processes, then n2 <= n1 condition must be fulfilled. Number of processes greater than number of partitions in RAM is invalid in Fixed Partitioning.

5.2 Variable (or dynamic) partitioning

In this when an execution request of a process has to be made, then the memory is partition according to the size which is needed by the processes so that there will not be the internal and external fragmentation. In this when a process requested for some memory then the needed memory will be allocated by the process so that there will not be the case when some memory spaces will be the left. The memory spaces will be provided up to that time, when the ability of performing the number of programs doesn't comes to end or up to that time when the space of the hard disk never comes to an end. The dynamic memory allocation also provides us the compaction. In the compaction the memory areas those are free and those are not allocated by the process, will be combined and make a single large memory part.

Characteristics of dynamic partitions

- Initially RAM is empty and partitions are made during the run-time according to process's need instead of partitioning during system configure.
- The size of partition will be equal to incoming process.
- The partition size varies according to the need of the process so that the internal fragmentation can be avoided to ensure efficient utilization of RAM.
- Number of partitions in RAM is not fixed and depends on the number of incoming process and Main Memory's size.



Variable size partition

Advantages of Variable Partitioning

- **No Internal Fragmentation:** In variable Partitioning, space in main memory is allocated strictly according to the need of process, hence there is any case of internal fragmentation. There will be no unused space left in the partition.
- **No restriction on degree of multiprogramming:** More number of processes can be accommodated due to absence of internal fragmentation. A process can be loaded until the memory is not empty.
- **No limitation on the size of the process:** In Fixed partitioning, the process with the size greater than the size of the largest partition could not be loaded and process cannot be divided as it is invalid in contiguous allocation technique. Here, in variable partitioning, the process size can't be restricted since the partition size is decided according to the process size.

Disadvantages of Variable Partitioning

- **Difficult implementation:** Implementing variable Partitioning is difficult as compared to Fixed Partitioning as it involves allocation of memory during run-time rather than during system configure.
- **External fragmentation:** There will be external fragmentation in spite of absence of internal fragmentation.

6. Relocation and Protection

Relocation

Relocation is the process of assigning load addresses for position-dependent code and data of a program and adjusting the code and data to reflect the assigned addresses. Programmers typically do not know in advance which other programs will be resident in main memory at the time of execution of their program. Active processes need to be able to be swapped in and out of main memory in order to maximize processor utilization. Suppose 1st instruction of a program

jumps at absolute address 200 within the exe file and it is loaded into partition 1 started at 100K. Jump should be performed at 100K+200K. This problem is called relocation problem.

It can be minimized by using relocation as the program is loaded into memory, modify the instructions accordingly. Address locations are added to the base location of the partition to map to the physical address. There are two types of relocations:

- **Static Relocation:** Program must be relocated before or during loading of process into memory. Program must always be loaded into same address space in memory, or relocator must be run again.
- **Dynamic Relocation:** Process can be freely moved around in memory. Virtual-to-physical address space mapping is done at run-time.

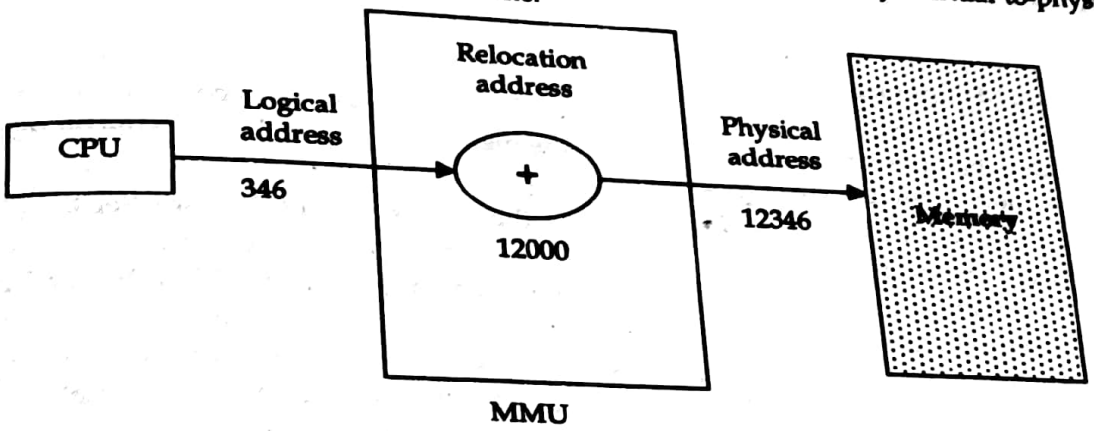


Fig 4.2: Memory relocation.

Protection

Memory protection is a way to control memory access rights on a computer, and is a part of most modern instruction set architectures and operating systems. The main purpose of memory protection is to prevent a process from accessing memory that has not been allocated to it. This prevents a bug or malware within a process from affecting other processes, or the operating system itself. An attempt to access unwonted memory results in a hardware fault, called a segmentation fault or storage violation exception, generally causing abnormal termination of the offending process. Memory protection for computer security includes additional technique such as address space layout randomization and executable space protection.

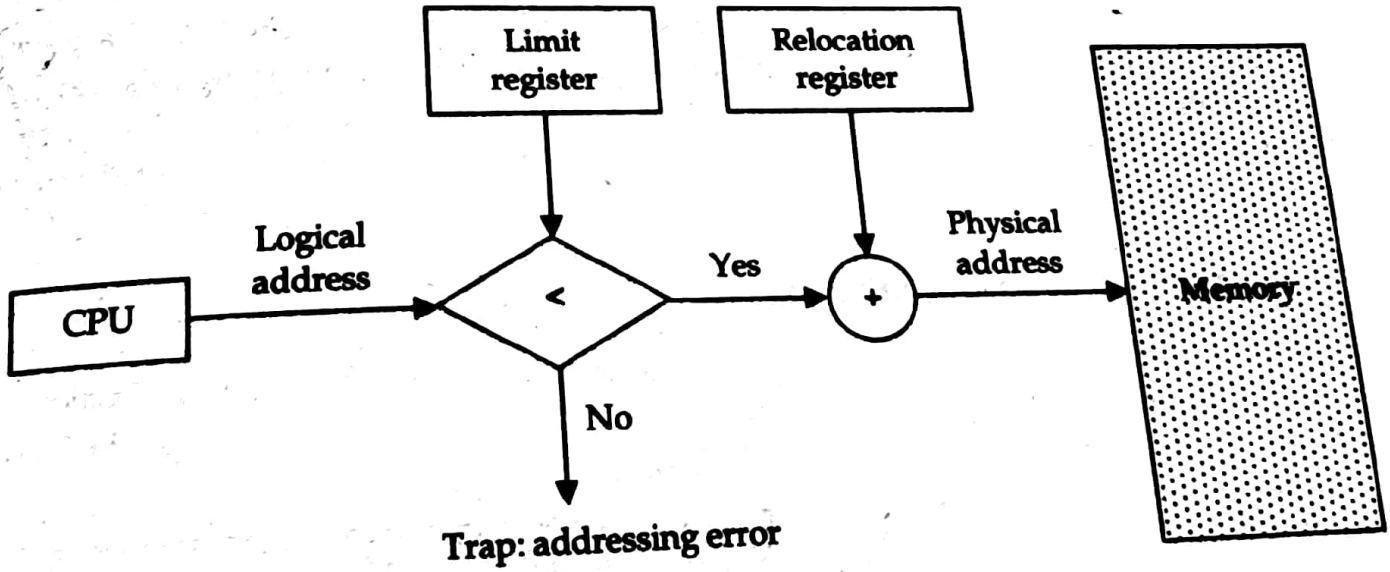


Fig 4.3: Memory protection and relocation

7. Memory Management (Bitmaps & Linked-list)

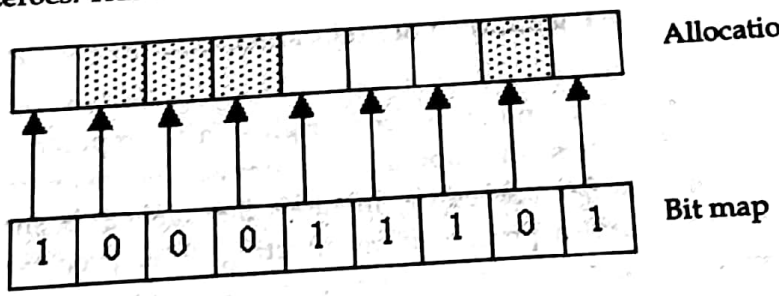
Memory management is the functionality of an operating system which handles or manages primary memory and moves processes back and forth between main memory and disk during execution. Memory management keeps track of each and every memory location, regardless of either it is allocated to some process or it is free. It checks how much memory is to be allocated to processes. It decides which process will get memory at what time. It tracks whenever some memory gets freed or unallocated and correspondingly it updates the status. There are two ways of doing memory management:

- Using Bitmaps and
- Using Linked list

7.1 Memory Management by Bitmaps

Under this scheme the memory is divided into allocation units and each allocation unit has a corresponding bit in a bit map. If the bit is zero, the memory is free. If the bit in the bit map is one, then the memory is currently being used.

The main decision with this scheme is the size of the allocation unit. The smaller the allocation unit, the larger the bit map has to be. But, if we choose a larger allocation unit, we could waste memory as we may not use all the space allocated in each allocation unit. The other problem with a bit map memory scheme is when we need to allocate memory to a process. Assume the allocation size is 4 bytes. If a process requests 256 bytes of memory, we must search the bit map for 64 consecutive zeroes. This is a slow operation and for this reason bit maps are not often used.



7.2 Memory Management with Linked Lists

Another way of keeping track of memory is to maintain a linked list of allocated and free memory segments, where a segment is either a process or a hole between two processes. Each entry in the list specifies a hole (H) or process (P), the address at which it starts, the length, and a pointer to the next entry. In this example, the segment list is kept sorted by address. Sorting this way has the advantage that when a process terminates or is swapped out, updating the list is straight forward. A terminating process normally has two neighbors (except when it is at the very top or very bottom of memory).

8. Memory Allocation Strategies

Memory allocation is the process of assigning blocks of memory on request. Typically the allocator receives memory from the operating system in a small number of large blocks that it must divide up to satisfy the requests for smaller blocks.

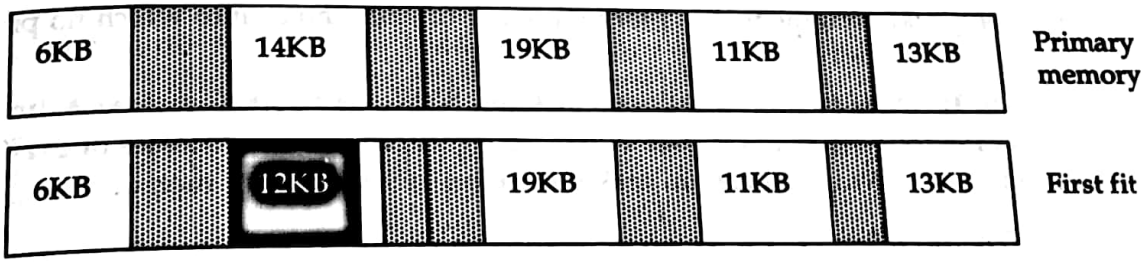
When the processes and holes are kept on a list sorted by address, several algorithms can be used to allocate memory for a newly created process (or an existing process being swapped in from disk). We assume that the memory manager knows how much memory to allocate. Following popular methods are used to allocate memory for a process in the memory management with linked list:

- First Fit
- Next Fit
- Best Fit
- Worst Fit
- Quick Fit

First Fit

The simplest algorithm is first fit. The process manager scans along the list of segments until it finds a hole that is big enough. The hole is then broken up into two pieces, one for the process and one for the unused memory, except in the statistically unlikely case of an exact fit. First fit is a fast algorithm because it searches as little as possible.

There may be many holes in the memory, so the operating system, to reduce the amount of time it spends analyzing the available spaces, begins at the start of primary memory and allocates memory from the first hole it encounters large enough to satisfy the request. For example, suppose a process requests 12KB of memory and the memory manager currently has a list of unallocated blocks of 6KB, 14KB, 19KB, 11KB, and 13KB blocks. The first-fit strategy will allocate 12KB of the 14KB block to the process.

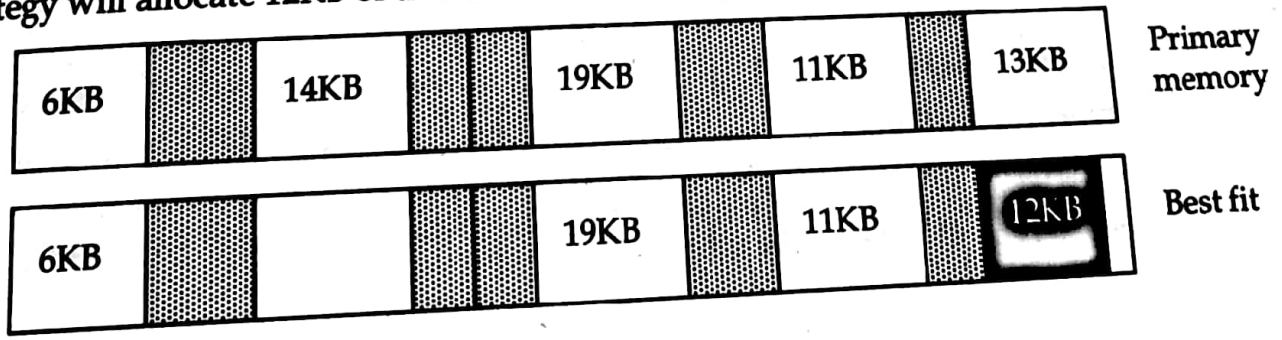


Next Fit

It works the same way as first fit, except that it keeps track of where it is whenever it finds a suitable hole. The next time it is called to find a hole, it starts searching the list from the place where it left off last time, instead of always at the beginning, as first fit does.

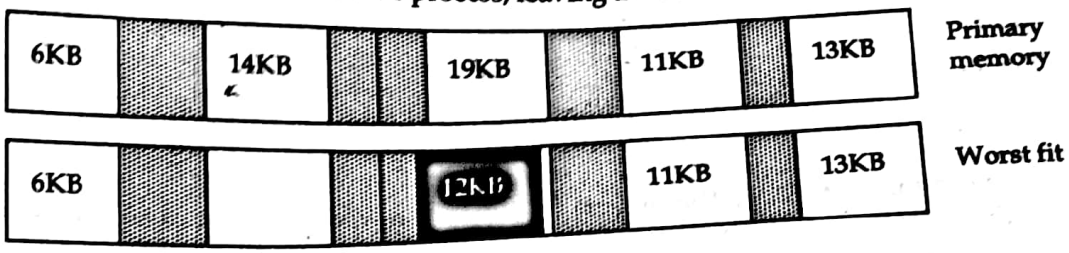
Best Fit

Best fit searches the entire list and takes the smallest hole that is adequate. Rather than breaking up a big hole that might be needed later, best fit tries to find a hole that is close to the actual size needed. The allocator places a process in the smallest block of unallocated memory in which it will fit. For example, suppose a process requests 12KB of memory and the memory manager currently has a list of unallocated blocks of 6KB, 14KB, 19KB, 11KB, and 13KB blocks. The best-fit strategy will allocate 12KB of the 13KB block to the process.



Worst Fit

Always take the largest available hole, so that the hole broken off will be big enough to be useful. Simulation has shown that worst fit is not a very good idea either. The memory manager places a process in the largest block of unallocated memory available. The idea is that this placement will create the largest hold after the allocations, thus increasing the possibility that compared to best fit; another process can use the remaining space. Using the same example as above, worst fit will allocate 12KB of the 19KB block to the process, leaving a 7KB block for future use.



Quick Fit

It maintains separate lists for some of the more common sizes requested. For example, it might have a table with n entries, in which the first entry is a pointer to the head of a list of 4-KB holes; the second entry is a pointer to a list of 8-KB holes, the third entry a pointer to 12-KB holes, and so on. Holes of say, 21 KB, could either be put on the 20-KB list or on a special list of odd-sized holes. With quick fit, finding a hole of the required size is extremely fast, but it has the same disadvantage as all schemes that sort by hole size, namely, when a process terminates or is swapped out, finding its neighbors to see if a merge is possible is expensive. If merging is not done, memory will quickly fragment into a large number of small holes into which no processes fit.

Numerical Problem 1: Given memory partitions of 100K, 500K, 200K, 300K, and 600K (in order), how would each of the First-Fit, Best-Fit, and Worst-Fit algorithms place processes of 212K, 417K, 112K, and 426K (in order)? Which algorithm makes the most efficient use of memory?

Answer: For First-Fit algorithm

1. 212K is put in 500K partition
2. 417K is put in 600K partition
3. 112K is put in 288K partition (new partition 288K = 500K - 212K)
4. 426K must wait

For Best-Fit algorithm

1. 212K is put in 300K partition
2. 417K is put in 500K partition
3. 112K is put in 200K partition
4. 426K is put in 600K partition

For Worst-Fit algorithm

1. 212K is put in 600K partition
2. 417K is put in 500K partition
3. 112K is put in 388K partition
4. 426K must wait

In this example, Best-Fit turns out to be the best.

9. Virtual Memory

Virtual memory is a feature of an operating system that enables a computer to be able to compensate shortages of physical memory by transferring pages of data from random access memory to disk storage.

A computer can address more memory than the amount physically installed on the system. This extra memory is actually called virtual memory and it is a section of a hard disk that's set up to emulate the computer's RAM. The main visible advantage of this scheme is that programs can be larger than physical memory. Virtual memory serves two purposes. First, it allows us to extend the use of physical memory by using disk. Second, it allows us to have memory protection, because each virtual address is translated to a physical address.

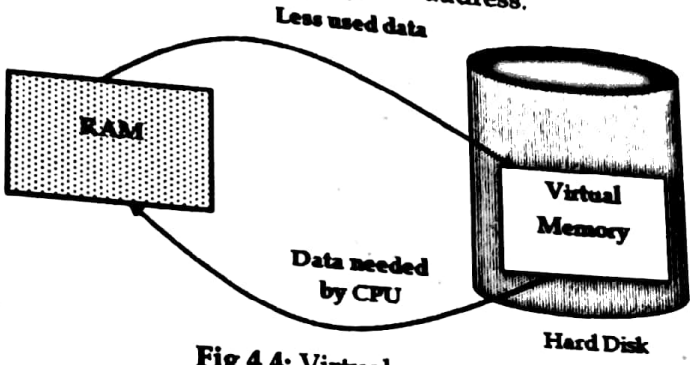


Fig 4.4: Virtual memory

Limitations of virtual memory

The use of virtual memory has its tradeoffs, particularly with speed. It's generally better to have as much physical memory as possible so programs work directly from RAM or physical memory. The use of virtual memory slows a computer because data must be mapped between virtual and physical memory, which requires extra hardware support for address translations.

Virtual address space vs. Physical address space

Address generated by CPU while a program is running is referred as logical address. The logical address is virtual as it does not exist physically. Hence, it is also called as virtual address. This address is used as a reference to access the physical memory location. The set of all logical addresses generated by a programs perspective is called Logical Address Space. The logical address is mapped to its corresponding physical address by a hardware device called Memory-Management Unit. The address-binding methods used by MMU generate identical logical and physical address during compile time and load time. However, while run-time the address-binding methods generate different logical and physical address.

Physical address identifies a physical location in a memory. MMU (Memory-Management Unit) computes the physical address for the corresponding logical address. MMU also uses logical address computing physical address. The user never deals with the physical address. Instead, the physical address is accessed by its corresponding logical address by the user. The user program generates the logical address and thinks that the program is running in this logical address. But the program needs physical memory for its execution. Hence, the logical address must be mapped to the physical address before they are used. The logical address is mapped to the physical address using a hardware called Memory-Management Unit. The set of all physical

addresses corresponding to the logical addresses in a logical address space is called physical address space.

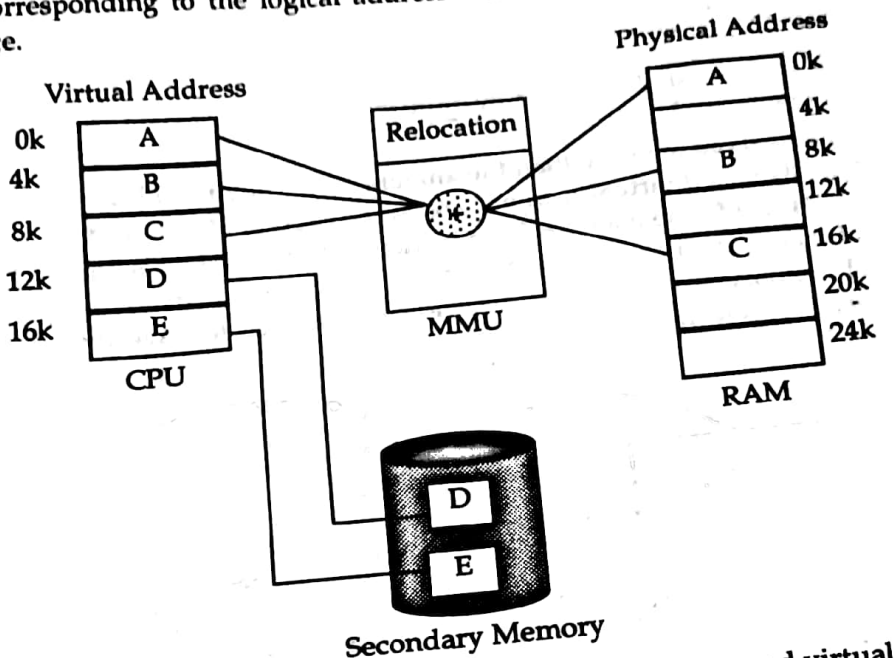


Fig 4.5: Block diagram showing physical address and virtual address

Key differences between logical and physical address in OS

- The basic difference between logical and physical address is that logical address is generated by CPU in perspective of a program. On the other hand, the physical address is a location that exists in the memory unit.
- The set of all logical addresses generated by CPU for a program is called logical address space. However, the set of all physical address mapped to corresponding logical addresses is referred as physical address space.
- The logical address is also called virtual address as the logical address does not exist physically in the memory unit. The physical address is a location in the memory unit that can be accessed physically.
- Identical logical address and physical address are generated by compile-time and load time address binding methods.
- The logical and physical address generated while run-time address binding method differs from each other.
- The logical address is generated by the CPU while program is running whereas; the physical address is computed by the MMU (Memory Management Unit).

Numerical Problem 1: Consider a logical address space of 8 pages of 1024 words mapped into memory of 32 frames.

a. How many bits are there in the logical address?

b. How many bits are there in physical address?

Answer

a. Logical address will have

3 bits to specify the page number (for 8 pages).

10 bits to specify the offset into each page (2^{10} =1024 words) = 13 bits.

b. For (2^5) 11 32 frames of 1024 words each (Page size = Frame size)
We have 5 + 10 = 15 bits.

10. Swapping

Swapping is mechanisms in which a process can be swapped temporarily out of main memory (or move) to secondary storage (disk) and make that memory available to other processes. At some later time, the system swaps back the process from the secondary storage to main memory.

For improving the performance of the system we use the concept of swapping. In the swapping the processes those are on waiting state and those are on suspend or temporary suspend will be stored from outside the memory locations so that the speed of process will be high. In this the process those are waiting for some input and output are transferred to the physical memory from they are running and the processes those are ready for the execution will be execute by the CPU. When there is a situation to perform swapping, then we use the sweeper. The sweeper is used for:

- Selecting which process to be out
- Selecting which process to be in
- Providing the memory space to the processes those are newly entered.

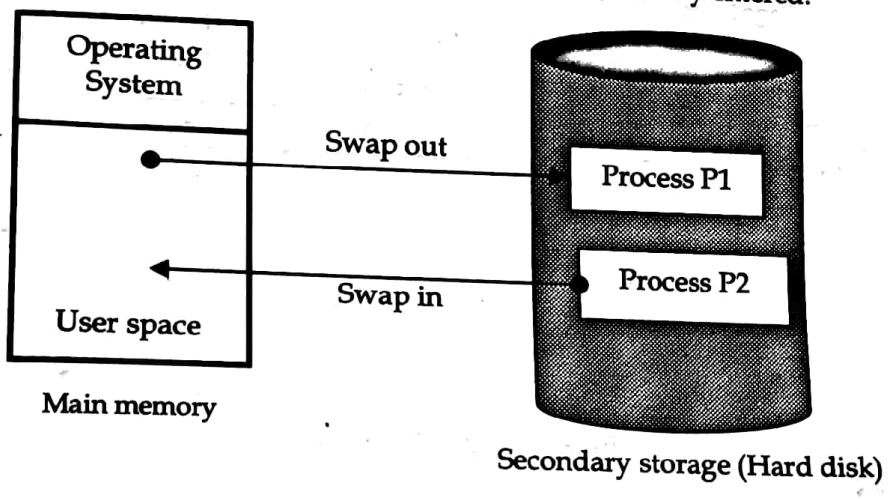


Fig 4.6: Swapping of two processes using a disk as a backing store

11. Paging

In computer operating systems, paging is a memory management scheme by which a computer stores and retrieves data from secondary storage for use in main memory. In this scheme, the operating system retrieves data from secondary storage in same-size blocks called **pages.**

A computer can address more memory than the amount physically installed on the system. This extra memory is actually called virtual memory and it is a section of a hard disk that's set up to emulate the computer's RAM. Paging technique plays an important role in implementing virtual memory. Paging is a memory management technique in which process address space is broken into blocks of the same size called pages (size is power of 2, between 512 bytes and 8192 bytes). The size of the process is measured in the number of pages. Similarly, main memory is divided

into small fixed-sized blocks of (physical) memory called frames and the size of a frame is kept the same as that of a page to have optimum utilization of the main memory and to avoid external fragmentation.

A computer can address more memory than the amount physically installed on the system. This extra memory is actually called virtual memory and it is a section of a hard disk that's set up to emulate the computer's RAM. Paging technique plays an important role in implementing virtual memory. Paging is a memory management technique in which process address space is broken into blocks of the same size called pages (size is power of 2, between 512 bytes and 8192 bytes). The size of the process is measured in the number of pages. Similarly, main memory is divided into small fixed-sized blocks of (physical) memory called frames and the size of a frame is kept the same as that of a page to have optimum utilization of the main memory and to avoid external fragmentation.

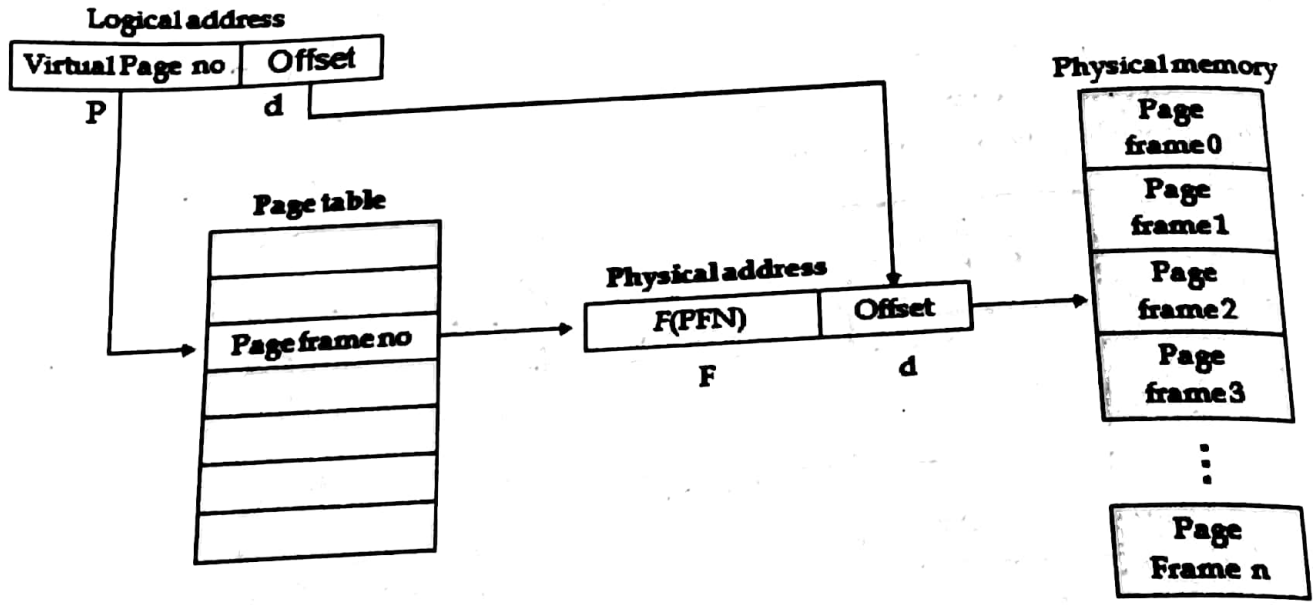


Fig 4.7: Paging Hardware

Drawbacks of Paging

- Size of Page table can be very big and therefore it wastes main memory.
- CPU will take more time to read a single word from the main memory.

Page Table

A page table is the data structure used by a virtual memory system in a computer operating system to store the mapping between virtual addresses and physical addresses.

It is a data structure that holds information (such as page number, offset, present/absent bit, modified bit etc.) about virtual pages. Since it acts as bridge between virtual pages and page frames thus mathematically we can say that the page table is a function, with the virtual page number as argument and the physical frame number as result. Page table entry has the following information:

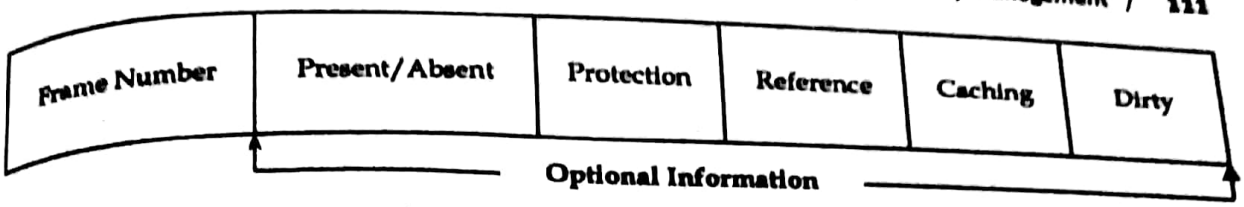


Fig 4.8: Page table entry structure

- **Frame Number:** It gives the frame number in which the current page you are looking for is present. The number of bits required depends on the number of frames.

$$\text{Number of bits for frame} = \frac{\text{Size of physical memory}}{\text{Frame size}}$$

- **Present/Absent bit:** Present or absent bit says whether a particular page you are looking for is present or absent. In case if it is not present, that is called Page Fault. It is set to 0 if the corresponding page is not in memory. Used to control page fault by the operating system to support virtual memory. Sometimes this bit is also known as valid/invalid bits.
- **Protection bit:** Protection bit says that what kind of protection you want on that page. So, these bit for the protection of the page frame (read, write etc).
- **Referenced bit:** Referenced bit will say whether this page has been referred in the last clock cycle or not. It is set to 1 by hardware when the page is accessed.
- **Caching enabled/disabled:** Sometimes we need the fresh data. Let us say the user is typing some information from the keyboard and your program should run according to the input given by the user. In that case, the information will come into the main memory. Therefore main memory contains the latest information which is typed by the user. Now if you try to put that page in the cache, that cache will show the old information. So whenever freshness is required, we don't want to go for caching or many levels of the memory. The information present in the closest level to the CPU and the information present in the closest level to the user might be different. So we want the information has to be consistency, which means whatever information user has given, CPU should be able to see it as first as possible. That is the reason we want to disable caching. So, this bit enables or disables caching of the page.
- **Modified bit:** Modified bit says whether the page has been modified or not. Modified means sometimes you might try to write something on to the page. If a page is modified, then whenever you should replace that page with some other page, then the modified information should be kept on the hard disk or it has to be written back or it has to be saved back. It is set to 1 by hardware on write-access to page which is used to avoid writing when swapped out. Sometimes this modified bit is also called as the **dirty bit.**

12. Handling Page Faults

A page fault occurs when a program attempts to access data or code that is in its address space, but is not currently located in the system RAM. So when page fault occurs then following sequence of events happens:

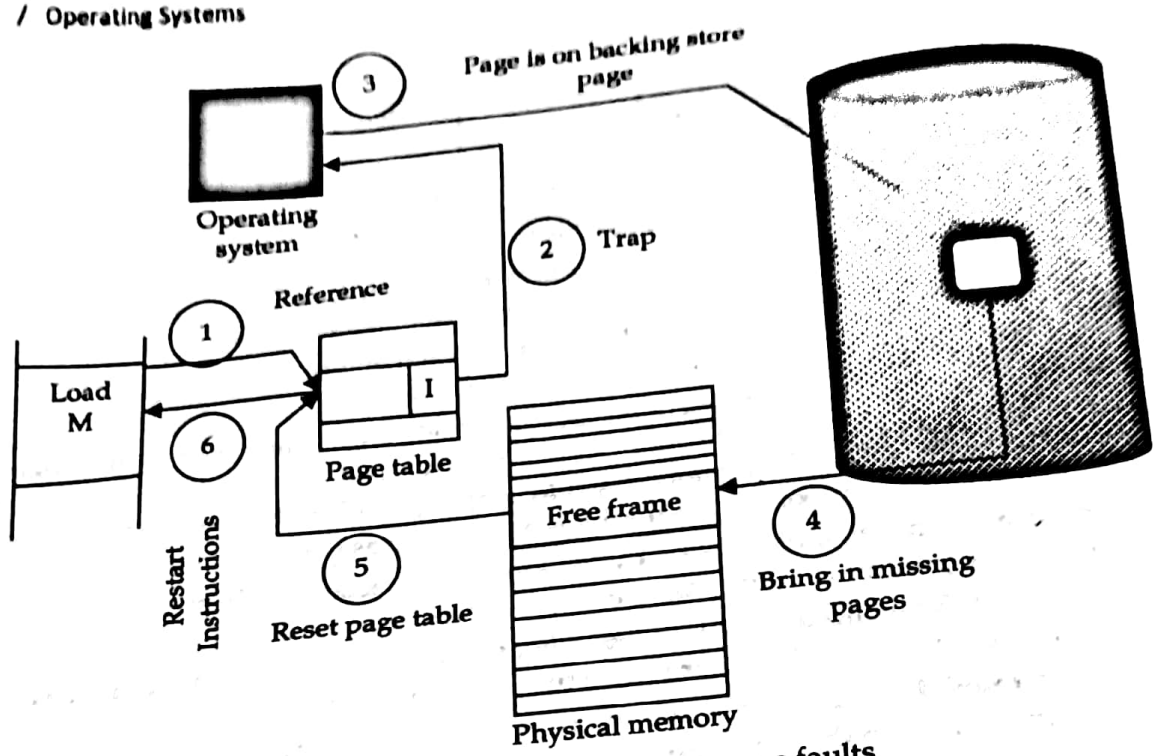


Fig 4.9: block diagram handling page faults

If a page is needed that was not originally loaded up, then a page fault trap is generated, which must be handled in a series of steps:

- The memory address requested is first checked, to make sure it was a valid memory request.
- If the reference was invalid, the process is terminated. Otherwise, the page must be paged in.
- A free frame is located, possibly from a free-frame list.
- A disk operation is scheduled to bring in the necessary page from disk. (This will usually block the process on an I/O wait, allowing some other process to use the CPU in the meantime)
- When the I/O operation is complete, the process's page table is updated with the new frame number, and the invalid bit is changed to indicate that this is now a valid page reference.
- The instruction that caused the page fault must now be restarted from the beginning, (as soon as this process gets another turn on the CPU)

13. TLB's

Since the page tables are stored in the main memory, each memory access of a program requires at least one memory accesses to translate virtual into physical address and to try to satisfy it from the cache. On the cache miss, there will be two memory accesses. The key to improving access performance is to rely on locality of references to page table. When a translation for a virtual page is used, it will probably be needed again in the near future because the references to the words on that page have both temporal and spatial locality. Each virtual memory reference can cause two physical memory accesses:

- One to fetch the page table
- One to fetch the data

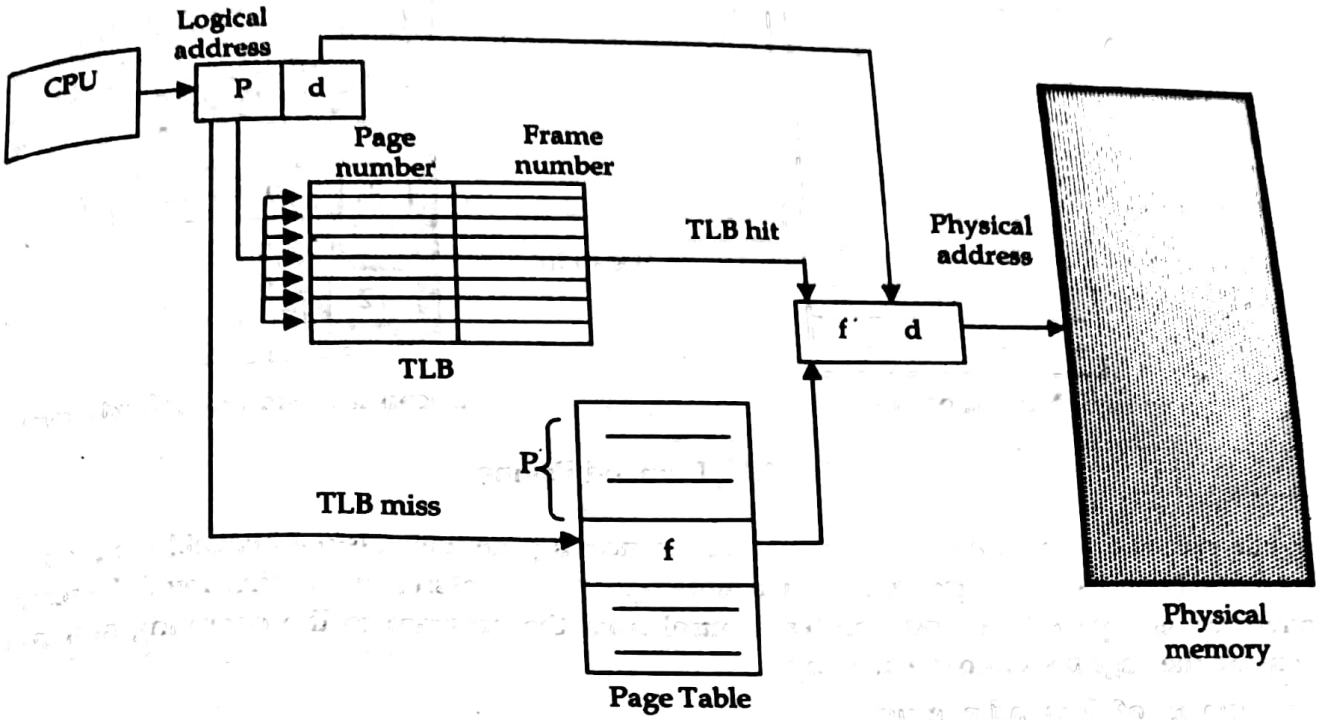


Fig 4.10: Paging hardware with TLB

To overcome this problem a high-speed cache is set up for page table entries called a Translation Look aside Buffer (TLB). Translation Look aside Buffer (TLB) is nothing but a special cache used to keep track of recently used transactions. TLB contains page table entries that have been most recently used. Given a virtual address, processor examines the TLB If page table entry is present (TLB hit), the frame number is retrieved and the real address is formed. If page table entry is not found in the TLB (TLB miss), the page number is used to index the process page table. TLB first checks if page is already in main memory, if not in main memory a page fault is issued then the TLB is updated to include the new page entry.

14. Demand Paging

A demand paging system is quite similar to a paging system with swapping where processes reside in secondary memory and pages are loaded only on demand, not in advance. When a context switch occurs, the operating system does not copy any of the old program's pages out to the disk or any of the new program's pages into the main memory instead, it just begins executing the new program after loading the first page and fetches that program's pages as they are referenced.

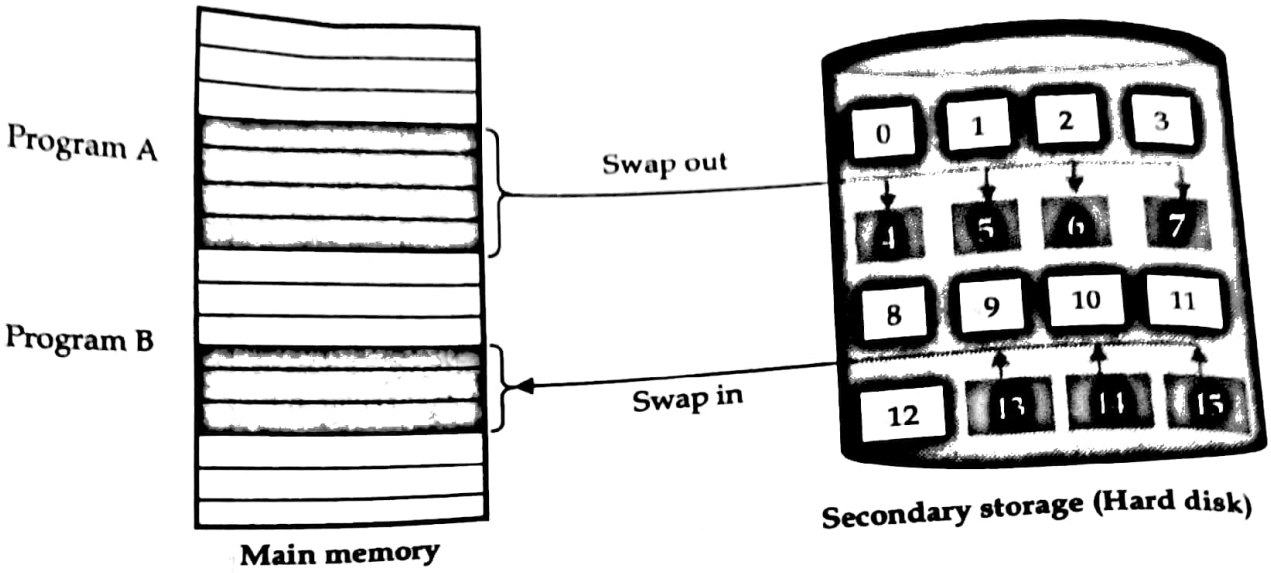


Fig 4.11: Demand Paging

While executing a program, if the program references a page which is not available in the main memory because it was swapped out a little ago, the processor treats this invalid memory reference as a page fault and transfers control from the program to the operating system to demand the page back into the memory.

Advantages of demand paging

- Large virtual memory
- More efficient use of memory
- There is no limit on degree of multiprogramming

Disadvantages of demand paging

- Number of tables and the amount of processor overhead for handling page interrupts are greater than in the case of the simple paged management techniques.

15. Page Replacement Algorithms

In an operating system that uses paging for memory management, a page replacement algorithm is needed to decide which page needs to be replaced when new page comes in. Since actual physical memory is much smaller than virtual memory, page faults happen. In case of page fault, Operating System might have to replace one of the existing pages with newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce number of page faults.

Page Fault: A page fault happens when a running program accesses a memory page that is mapped into the virtual address space, but not loaded in physical memory.

There are a lot of page replacement algorithms some of major page replacement algorithms are listed below:

- FIFO Page Replacement Algorithms
- Optimal Page Replacement Algorithms
- LRU Page Replacement Algorithms

- Second Chance Page Replacement Algorithms
- LFU Page Replacement Algorithms
- Clock Page Replacement Algorithms
- WS- Clock Page Replacement Algorithms

FIFO Page Replacement Algorithms

This is the simplest page replacement algorithm. In this algorithm, operating system keeps track of all pages in the memory in a queue; oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

For example: 3 page frames which initially empty and 20 pages system the FIFO page replacement is as below,

Reference string

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2	2	2	2	4	4	4	0	0	0	0	0	0	0	7	7	7
	0	0	0	0	3	3	3	2	2	2	2	2	1	1	1	1	1	0	0
		1	1	1	1	0	0	0	3	3	3	3	3	2	2	2	2	2	1
Pf	Pf	Pf	Pf	X	Pf	Pf	Pf	Pf	Pf	Pf	X	X	Pf	Pf	X	X	Pf	Pf	Pf

Page fault =15

Advantages FIFO page replacement algorithm

- Easy to understand and program
- Distribute fair chance to all

Disadvantages FIFO page replacement algorithm

- FIFO is likely to replace heavily (or constantly) used pages and they are still needed for further processing
- It is not very effective
- System needs to keep track of each frame
- Bad replacement choice increases the page fault rate and slow process execution
- Sometimes it behaves abnormally. This behavior is called Belady's anomaly.

Belady's anomaly

In computer storage, Belady's anomaly is the phenomenon in which increasing the number of page frames results in an increase in the number of page faults for certain memory access patterns. This phenomenon is commonly experienced when using the first-in first-out (FIFO) page replacement algorithm.

For example, if we consider reference string 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 and number of frames allocated=3, we get 9 total page faults, but if we increase slots to 4, we get 10 page faults.

Reference string

1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	4	4	4	5	5	5	5	5	5
	2	2	2	1	1	1	1	1	3	3	3
		3	3	3	2	2	2	2	2	4	4
Pf	Pf	Pf	Pf	Pf	Pf	Pf	X	X	Pf	Pf	X

Total page faults = 9

But if we use page frame size 4 for the same reference string we get 10 page faults as below;

1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	1	1	1	5	5	5	5	4	4
	2	2	2	2	2	2	1	1	1	1	5
		3	3	3	3	3	3	2	2	2	2
			4	4	4	4	4	4	3	3	3
Pf	Pf	Pf	Pf	X	X	Pf	Pf	Pf	Pf	Pf	Pf

Total number of page faults = 10

Second Chance Page Replacement Algorithms

It is the modified form of the FIFO page replacement algorithm. One way to implement is to have a circular queue. If the value is equal to 0, then we proceed to replace the page. But if the reference bit is equal to 1, then we give the page second chance. When the page get second chance, its reference bit is cleared. Second-chance page replacement algorithm gives every page a second-chance.

The basic algorithm of second-chance replacement is a FIFO replacement algorithm. When a page has been selected, however, we inspect its reference bit. If the value is 0, we proceed to replace this page. If the reference bit is set to 1, however, we give that page a second chance and move on to select the next FIFO page. When a page gets a second chance, its reference bit is cleared and its arrival time is reset to the current time. Thus, a page that is given a second chance will not be replaced until all other pages are replaced (or given second chances). In addition, if a page is used often enough to keep its reference bit set, it will never be replaced. One way to implement the second-chance (sometimes referred to as the clock) algorithm is as a circular queue. A pointer indicates which page is to be replaced next. When a frame is needed, the pointer advances until it finds a page with a 0 reference bit. As it advances, it clears the reference bits. Once a victim page is found, the page is replaced, and the new page is inserted in the circular queue in that position. Notice that, in the worst case, when all bits are set, the pointer cycles through the whole queue, giving each page a second chance. It clears all the reference bits before selecting the next page for replacement. Second-chance replacement degenerates to FIFO replacement if all bits are set.

Example: if we consider reference string 2, 3, 2, 1, 5, 2, 4, 5, 3, 2, 5, 2 and number of frames allocated=3, we get 7 total page faults by using this page replacement algorithm.

Reference strings

2	3	2	1	5	2	4	5	3	2	5	2
2(0)	2(0)	2(1)	2(1)	2(0)	2(1)	2(0)	2(0)	3(0)	3(0)	3(0)	3(0)
	3(0)	3(0)	3(0)	5(0)	5(0)	5(0)	5(1)	5(0)	5(0)	5(1)	5(1)
			1(0)	1(0)	1(0)	4(0)	4(0)	4(0)	2(0)	2(0)	2(1)
Pf	Pf	X	Pf	Pf	X	Pf	X	Pf	Pf	X	X

Total page faults = 7

Advantages

- Obvious improvement over FIFO
- Allows commonly used pages to stay in queue

Disadvantages

- Still suffers from Belady's anomaly

Optimal Page replacement

In this algorithm, pages are replaced which would not be used for the longest duration of time in the future. An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms, and will never suffer from Belady's anomaly. Such an algorithm does exist, and has been called OPT or MIN. It is simply replace the page that will not be used for the longest period of time. Use of this page replacement algorithm guarantees the lowest possible page fault rate for a fixed number of frames. For example, on our sample reference string, the optimal page-replacement algorithm would yield 9 page faults. The first three references cause faults that fill the three empty frames. The reference to page reference string 2 replaces page 7, because 7 will not be used until reference 18, whereas page 0 will be used at 5, and page 1 at 14. The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again. With only 9 page faults, optimal replacement is much better than a FIFO algorithm, which had 15 faults.

Example: if we consider reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 and number of frames allocated=3, we get 9 total page faults by using this page replacement algorithm.

Reference string

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2	2	2	2	2	2	2	2	2	2	2	2	2	2	7	7	7
	0	0	0	0	0	0	4	4	4	0	0	0	0	0	0	0	0	0	0
		1	1	1	3	3	3	3	3	3	3	3	1	1	1	1	1	1	1
Pf	Pf	Pf	Pf	X	Pf	X	Pf	X	X	Pf	X	X	Pf	X	X	X	Pf	X	X

Total page faults = 9

Advantages of optimal page replacement algorithm

- It is less complex and easy to implement.
- A page is replaced with minimum fuss.
- Simple data structures are used for this purpose.
- Lowest page fault rate.
- Never suffers from Belady's anomaly.
- Twice as good as FIFO Page Replacement Algorithm.

Disadvantages of optimal replacement algorithm

- Not all operating systems can implement this algorithm.
- Error detection is harder.
- Least recently used page will be replaced which may sometimes take a lot of time.

LRU Page Replacement Algorithms

In this algorithm, the page that has not been used for the longest period of time has to be replaced.

If the optimal algorithm is not feasible, perhaps an approximation to the optimal algorithm is possible. The key distinction between the FIFO and OPT algorithms (other than looking backward or forward in time) is that the FIFO algorithm uses the time when a page was brought into memory; the OPT algorithm uses the time when a page is to be used. If we use the recent

past as an approximation of the near future, then we will replace the page that has not been used for the longest period of time. This approach is the least-recently-used (LRU) algorithm. LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses that page that has not been used for the longest period of time. This strategy is the optimal page replacement algorithm looking backward in time, rather than forward.

Example: if we consider reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 and number of frames allocated=3, we get 12 total page faults by using this page replacement algorithm.

Reference string

	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
	7	7	7	2	2	2	2	4	4	4	0	0	0	1	1	1	1	1	1	1
		0	0	0	0	0	0	0	0	3	3	3	3	3	3	0	0	0	0	0
			1	1	1	3	3	3	2	2	2	2	2	2	2	2	2	7	7	7
	Pf	Pf	Pf	Pf	X	Pf	X	Pf	Pf	Pf	Pf	X	X	Pf	X	Pf	X	Pf	X	X

Total page faults = 12

Advantages of LRU page replacement algorithm

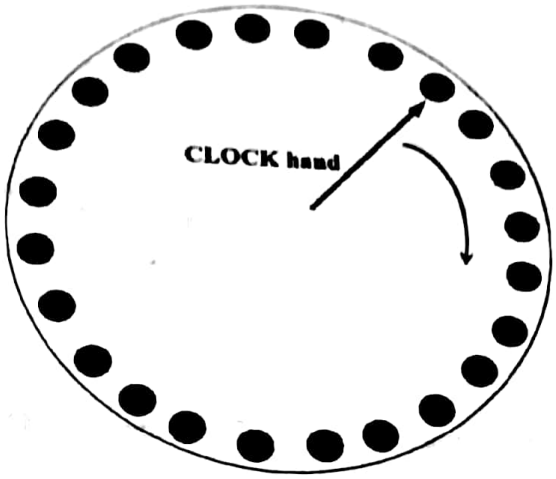
- It is amenable to full statistical analysis
- Never suffers from Belady's anomaly

Clock Page Replacement Algorithms

Clock is a more efficient version of **FIFO** than Second-chance because pages don't have to be constantly pushed to the back of the list, but it performs the same general function as Second-Chance. The clock algorithm keeps a circular list of pages in memory, with the "**hand**" pointing to the last examined page frame in the list. When a page fault occurs and no empty frames exist, then the R (referenced) bit is inspected at the hand's location. If R is 0, the new page is put in place of the page the "**hand**" points to, and the **hand** is advanced one position. Otherwise, the R bit is cleared, then the clock **hand** is incremented and the process is repeated until a page is replaced.

CLOCK Algorithm

1. Begin
2. Read new page, say P
3. If P is available in CLOCK (Circular Linked List).
   - 3.1. Page hit occurs. then
   - 3.2. Turn reference bit to 1 and do nothing else.
4. else
   - 4.1 Page miss occurs then
   - 4.2 If its *R* bit is 0, the page is evicted, the new page is inserted into the clock in its place, and the hand is advanced one position.
   - 4.3 If *R* is 1, it is cleared and the hand is advanced to the next page. This process is repeated until a page is found with $R = 0$.
5. Stop



Example: Input References: {1, 2, 3, 4, 3, 1, 2, 1, 5, 4}
Size of Memory: 3
Total Number of References: 10
Number of Distinct References: 5

Upon accessing 1

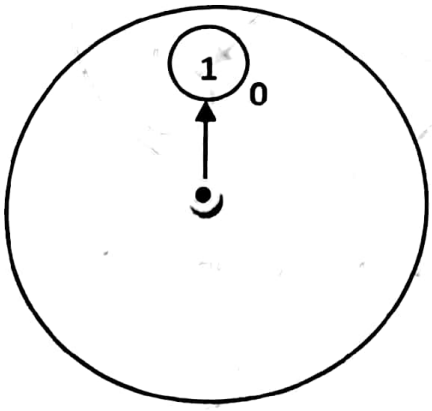
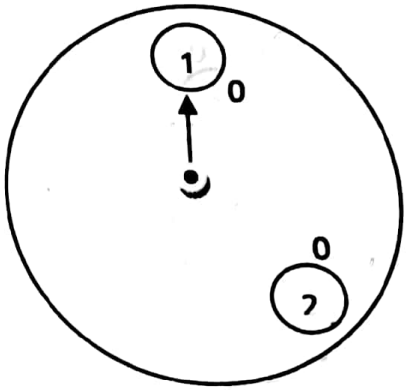


Fig: pages in CLOCK at virtual time 1

Page fault

Upon accessing 2



Page fault

Fig: pages in CLOCK at virtual time 2

Upon accessing 3

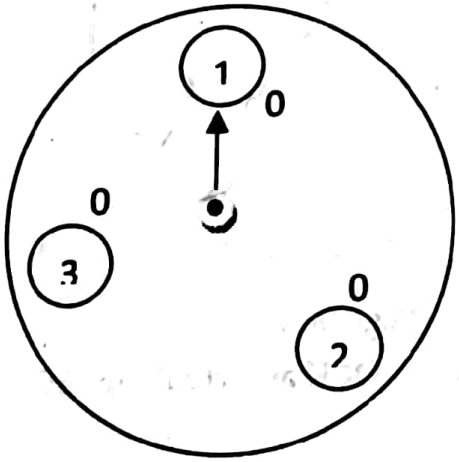


Fig: pages in CLOCK at virtual time 3

Upon accessing 4

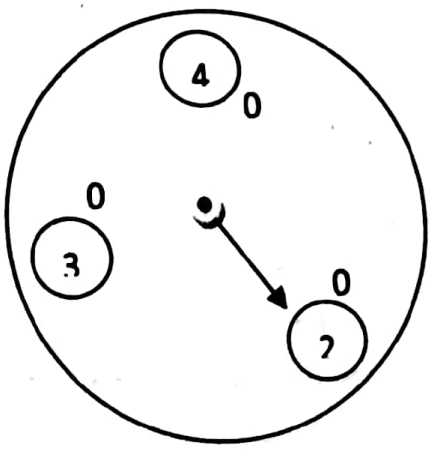


Fig: pages in CLOCK at virtual time 4

Upon accessing 3

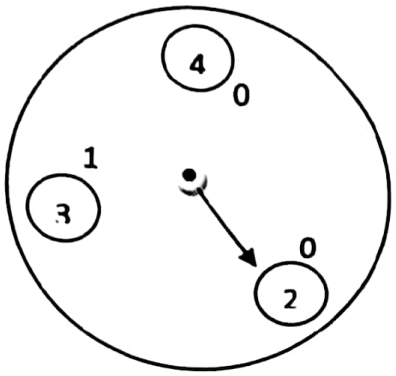


Fig: pages in CLOCK at virtual time 5

Upon accessing 1

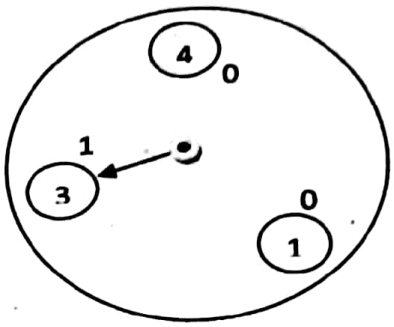
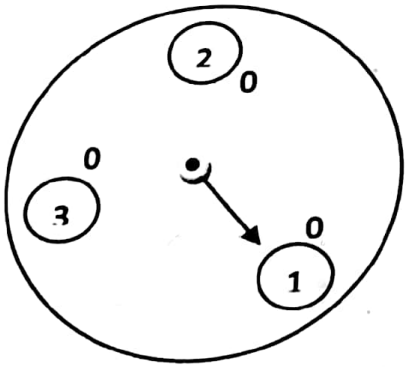


Fig: pages in CLOCK at virtual time 6

Upon accessing 2



Page fault

Fig: pages in CLOCK at virtual time 7

Upon accessing 1

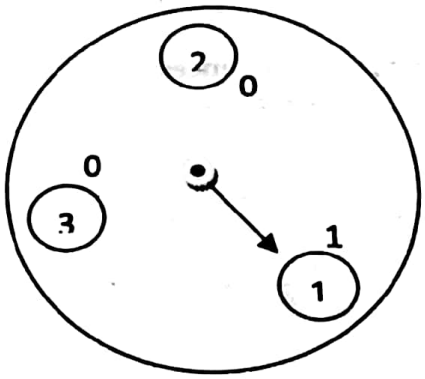
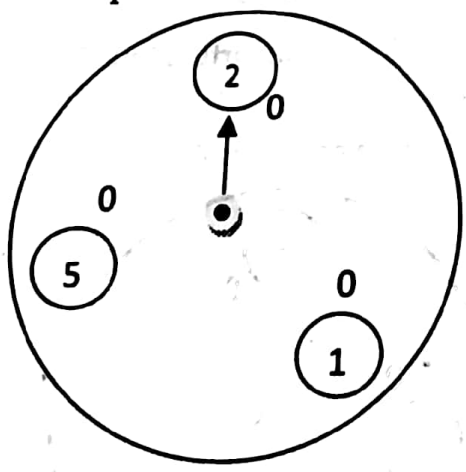


Fig: pages in CLOCK at virtual time 8

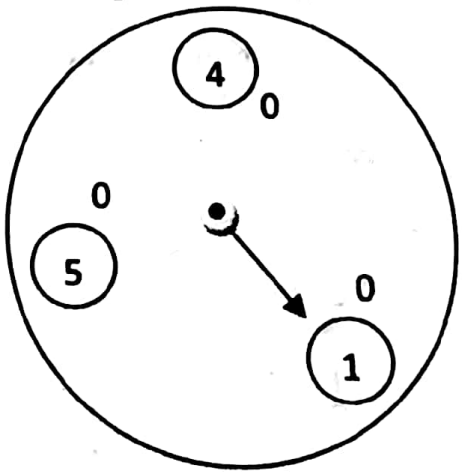
Upon accessing 5



Page fault

Fig: pages in CLOCK at virtual time 9

Upon accessing 4



Page fault

Fig: pages in CLOCK at virtual time 10

Total number of page faults: 8

16. Concept of Locality of Reference

Locality of reference, also known as the principle of locality, is a term for the phenomenon in which the same values, or related storage locations, are frequently accessed, depending on the memory access pattern. There are two basic types of reference locality:

- Temporal locality of reference and
- Spatial locality of reference

16.1 Temporal locality of reference

Temporal locality means current data or instruction that is being fetched may be needed soon. So we should store that data or instruction in the cache memory so that we can avoid again searching in main memory for the same data and thus saving time.

16.2 Spatial locality of reference

Spatial locality means instruction or data near to the current memory location that is being fetched, may be needed soon in near future.

Temporal locality refers to the reuse of specific data, and resources, within relatively small time duration whereas spatial locality refers to the use of data elements within relatively close storage locations.

Example: Let's take a code segment

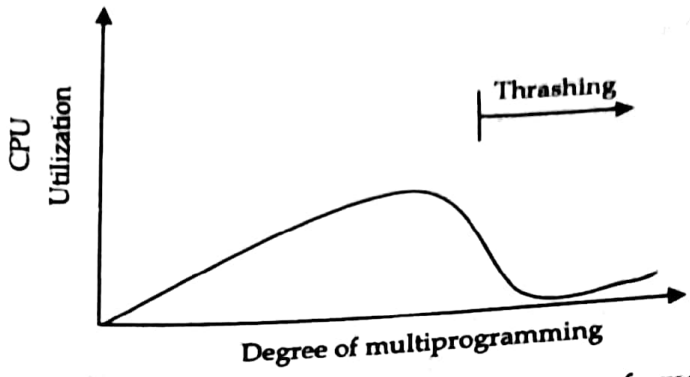
```
sum = 0;
for (i = 0; i < arr.length; i++)
        sum += arr[i];
return sum;
```

Now looking at this example, here variable **sum** is being used again and again which shows temporal locality and then the values of array **arr** is being accessed in order i.e. arr[0], arr[1], arr[2] ,... and so on and which shows spatial locality as arrays are contiguous memory blocks so data near to current memory location is being fetched.

17. Thrashing

Thrashing is a condition or a situation when the system is spending a major portion of its time in servicing the page faults, but the actual processing done is very negligible.

A process that is spending more time paging than executing is said to be thrashing. In other words it means that the process doesn't have enough frames to hold all the pages for its execution, so it is swapping pages in and out very frequently to keep executing. Sometimes, the pages which will be required in the near future have to be swapped out. Initially when the CPU utilization is low, the process scheduling mechanism to increase the level of multiprogramming loads multiple processes into the memory at the same time, allocating a limited amount of frames to each process. As the memory fills up, process starts to spend a lot of time for the required pages to be swapped in, again leading to low CPU utilization because most of the processes are waiting for pages. Hence the scheduler loads more processes to increase CPU utilization, as this continues at a point of time the complete system comes to a stop.

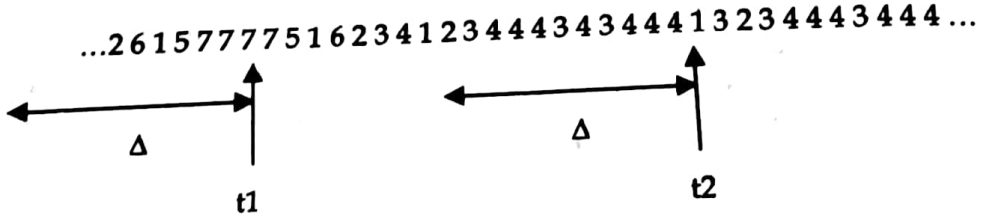


To prevent thrashing we must provide processes with as many frames as they really need right now and we also can use working set described as below.

18. Working Set

Working set is a concept in computer science which defines the amount of memory that a process requires in a given time interval. The set of pages that a process is currently using is called its working set.

The working set model is based on the assumption of locality. This model uses a parameter Δ to define the working set window. The idea is to examine the most recent Δ page references. The set of pages in the most recent Δ page references is called the working set. If a page is in active use it will be in the working set. If it no longer being used it will drop from the working set Δ time units after its last reference. Thus the working set is an approximation of the program's locality. In the following example, we use a value of Δ to be 10 and identify two localities of reference, one having five pages and the other having two pages.



WS (t1) = {1, 2, 5, 6, 7}

WS (t2) = {3, 4}

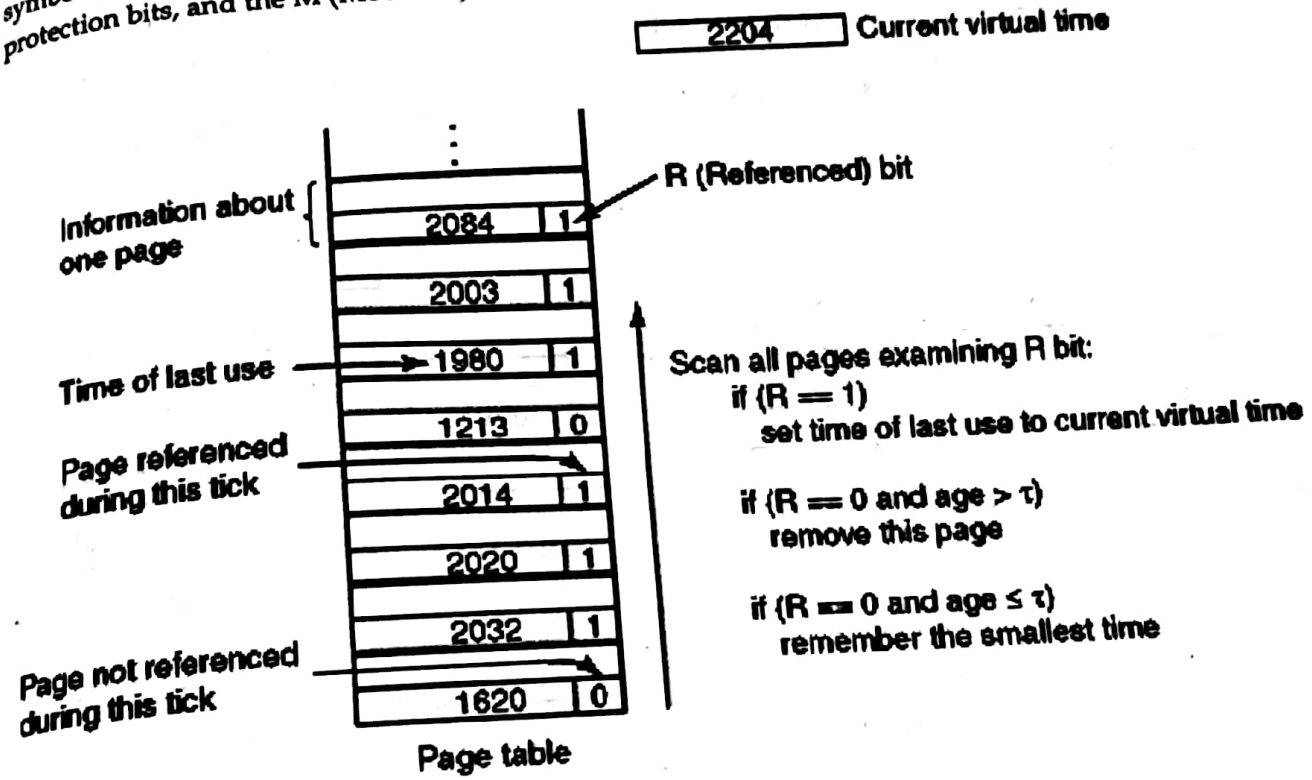
We now ientify various localities in the process execution trace given in the previous section. Here are the first two and last localities are: L1=(18-26, 31-34}, L2={18-23, 29-31, 34}, and last={18-20, 24-34}. Note that in that last locality, pages 18-20 are referenced right in the beginning only and are effectively out of the locality.

Working set page replacement algorithm

The set of pages that a process is currently using is called its **working set**. If the entire working set is in memory, the process will run without causing many faults until it moves into another execution phase (e.g. the next pass of the compiler). If the available memory is too small to hold the entire working set, the process will cause many page faults and run slowly since executing an instruction takes a few nanoseconds and reading in a page from the disk typically takes 10 milliseconds. At a rate of one or two instructions per 10 milliseconds, it will take ages to finish. A program causing page faults every few instructions is said to be thrashing. Therefore, many

paging systems try to keep track of each process' working set and make sure that it is in memory before letting the process run. This approach is called the working set model.

The basic idea behind this page replacement algorithm is to find a page that is not in the working set and evict it. In Fig. below we see a portion of a page table for some machine. Because only pages that are in memory are considered as candidates for eviction, pages that are absent from memory are ignored by this algorithm. Each entry contains (at least) two items of information: the approximate time the page was last used and the R (Referenced) bit. The empty white rectangle symbolizes the other fields not needed for this algorithm, such as the page frame number, the protection bits, and the M (Modified) bit.



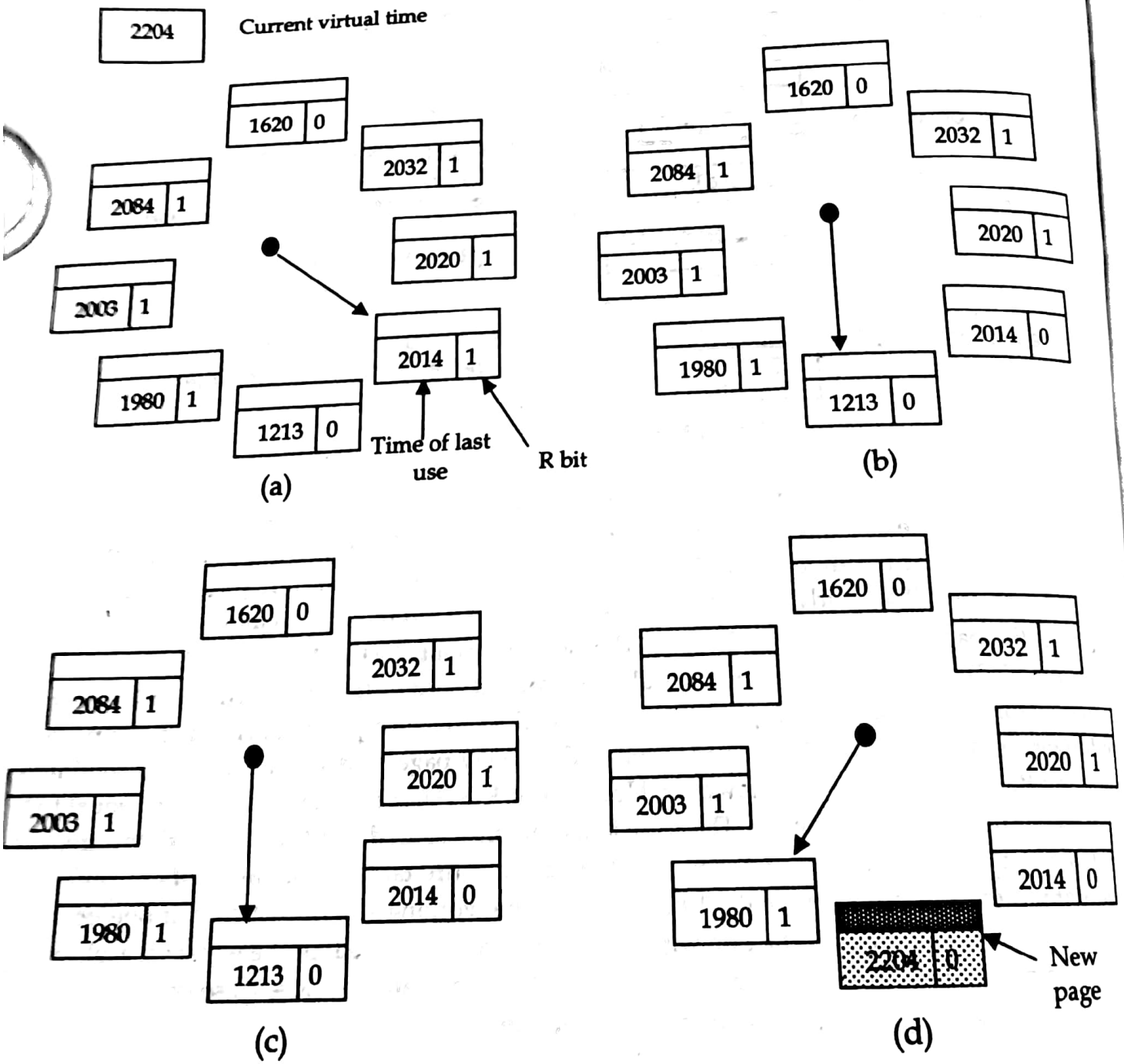
The algorithm works as follows. As each entry is processed, the R bit is examined. If it is 1, the current virtual time is written into the time of last use field in the page table, indicating that the page was in use at the time the fault occurred. Since the page has been referenced during the current clock tick, it is clearly in the working set and is not a candidate for removal.

If R is 0, the page has not been referenced during the current clock tick and may be a candidate for removal. To see whether or not it should be removed, its age, that is, the current virtual time minus its time of last use is computed and compared to t. If the age is greater than t, the page is no longer in the working set. It is reclaimed and the new page loaded here. The scan continues updating the remaining entries, however. However, if R is 0 but the age is less than or equal to t, the page is still in the working set. The page is temporarily spared, but the page with the greatest age (smallest value of time of last use) is noted. If the entire table is scanned without finding a candidate to evict, that means that all pages are in the working set. In that case, if one or more pages with R = 0 were found, the one with the greatest age is evicted. In the worst case, all pages have been referenced during the current clock tick (and thus all have R = 1), so one is chosen at random for removal, preferably a clean page, if one exists.

19. WS- Clock Page Replacement Algorithms

The basic working set algorithm is cumbersome since the entire page table has to be scanned at each page fault until a suitable candidate is located. An improved algorithm that is based on the clock algorithm but also uses the working set information is called WS-Clock. Due to its simplicity of implementation and good performance, it is widely used in practice.

The data structure needed is a circular list of page frames, as in the clock algorithm. Initially, this list is empty. When the first page is loaded, it is added to the list. As more pages are added, they go into the list to form a ring. Each entry contains the time of last use field from the basic working set algorithm, as well as the R bit (shown) and the M bit (not shown). As with the clock algorithm, at each page fault the page pointed to by the hand is examined first. If the R bit is set to 1, the page has been used during the current tick so it is not an ideal candidate to remove. The R bit is then set to 0, the hand advanced to the next page, and the algorithm repeated for that page.

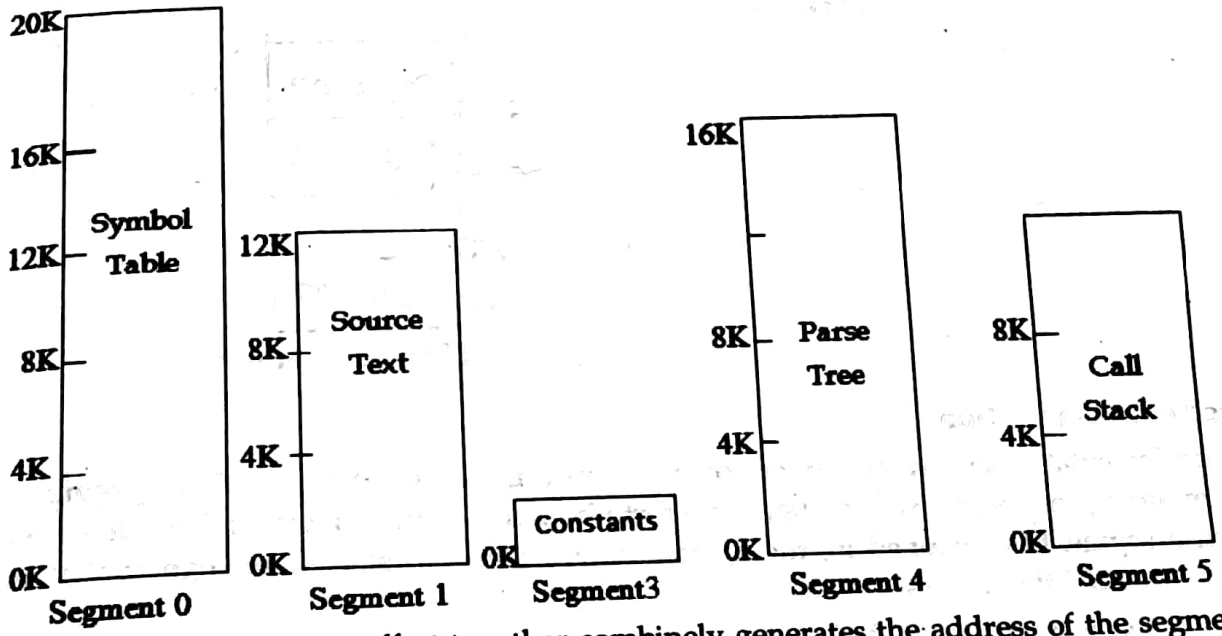


20. Segmentation

In Operating Systems, Segmentation is a memory management technique in which, the memory is divided into the variable size parts. Each part is known as segment which can be allocated to a process. The details about each segment are stored in a table called as segment table. Segment table is stored in one (or many) of the segments. Segment table contains mainly following information about segment:

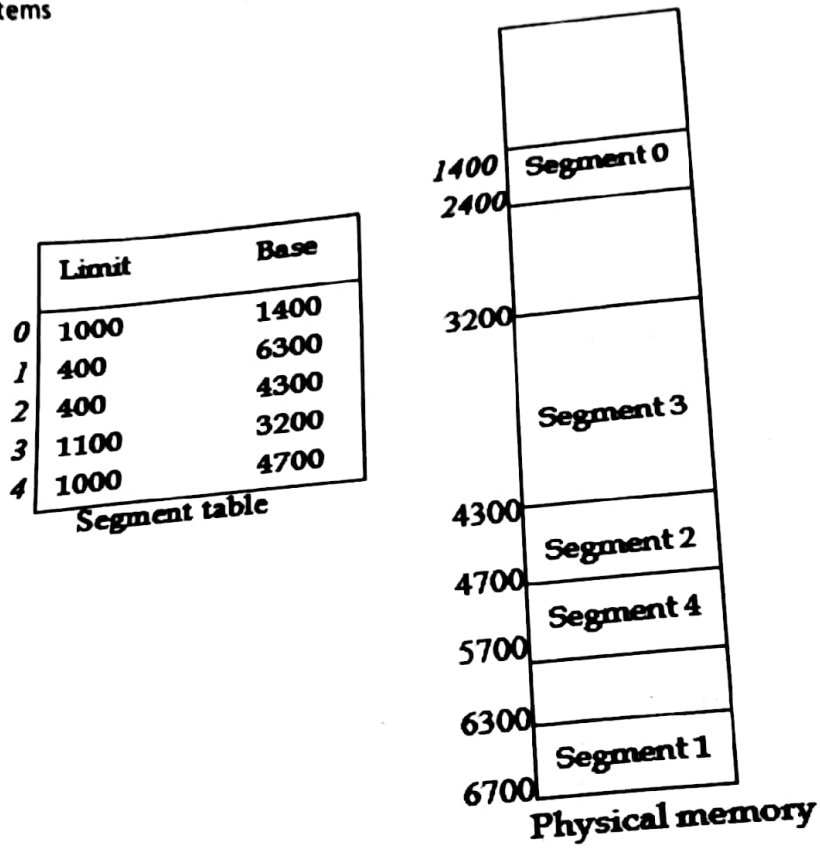
- **Base**: It is the base address of the segment
- **Limit**: It is the length of the segment

Each segment has a name and length. The addresses specify both the segment number and offset within the segment. The user therefore specifies each address by two quantities: **segment number** and an **offset**. Let us consider following example.



The segment number and the offset together combinely generates the address of the segment in the physical memory space.

Example: As an example, consider the situation shown in figure below. We have five segments numbered through 0 to 4. The segments are stored in physical memory as shown. The segment table has a separate entry for each segment, giving the beginning address of the segment in the physical memory (or base) and length of that segment (or limit). For example segment 2 is 400 byte long and begins at location 4300. Thus a reference to byte 53 of segment 2 is mapped onto location 4300+53=4353. A reference to segment 3, byte 852 is mapped to 3200 (the base of segment 3) + 852=4052. A reference to byte 1222 of segment 0 would result in a rap to operating system, as this segment is only 1000 bytes long.



Need of Segmentation

An implementation of virtual memory on a system using segmentation without paging requires that entire segments be swapped back and forth between main memory and secondary storage. When a segment is swapped in, the operating system has to allocate enough contiguous free memory to hold the entire segment.

Memory segmentation is the division of a computer's primary memory into segments or sections. In a computer system using segmentation, a reference to a memory location includes a value that identifies a segment and an offset (memory location) within that segment. Segments or sections are also used in object files of compiled programs when they are linked together into a program image and when the image is loaded into memory. Segments usually correspond to natural divisions of a program such as individual routines or data tables so segmentation is generally more visible to the programmer than paging alone. Different segments may be created for different program modules, or for different classes of memory usage such as code and data segments. Certain segments may be shared between programs.

Advantages of Segmentation

- No internal fragmentation
- Average Segment Size is larger than the actual page size
- Less overhead
- It is easier to relocate segments than entire address space
- The segment table is of lesser size as compare to the page table in paging

Disadvantages of Segmentation

- It can have external fragmentation
- It is difficult to allocate contiguous memory to variable sized partition
- Costly memory management algorithms

Differences between paging and segmentation

1. The basic difference between paging and segmentation is that a page is always of fixed block size whereas; a segment is of variable size.
2. Paging may lead to internal fragmentation as the page is of fixed block size, but it may happen that the process does not acquire the entire block size which will generate the internal fragment in memory. The segmentation may lead to external fragmentation as the memory is filled with the variable sized blocks.
3. In paging the user only provides a single integer as the address which is divided by the hardware into a page number and Offset. On the other hands, in segmentation the user specifies the address in two quantities i.e. segment number and offset.
4. The size of the page is decided or specified by the hardware. On the other hands, the size of the segment is specified by the user.
5. In paging, the page table maps the logical address to the physical address, and it contains base address of each page stored in the frames of physical memory space. However, in segmentation, the segment table maps the logical address to the physical address, and it contains segment number and offset (segment limit).

21. Segmentation with Paging (MULTICS)

Instead of an actual memory location the segment information includes the address of a page table for the segment. When a program references a memory location the offset is translated to a memory address using the page table. A segment can be extended simply by allocating another memory page and adding it to the segment's page table.

An implementation of virtual memory on a system using segmentation with paging usually only moves individual pages back and forth between main memory and secondary storage, similar to a paged non-segmented system. Pages of the segment can be located anywhere in main memory and need not be contiguous. This usually results in a reduced amount of input/output between primary and secondary storage and reduced memory fragmentation.

The MULTICS operating system was one of the most influential operating systems ever, Having had a major influence on topics as disparate as UNIX, the x86 memory architecture, TLBs, and cloud computing. It was started as a research project at M.I.T. and went live in 1969. The MULTICS system solved problems of external fragmentation and lengthy search times by paging the segments.

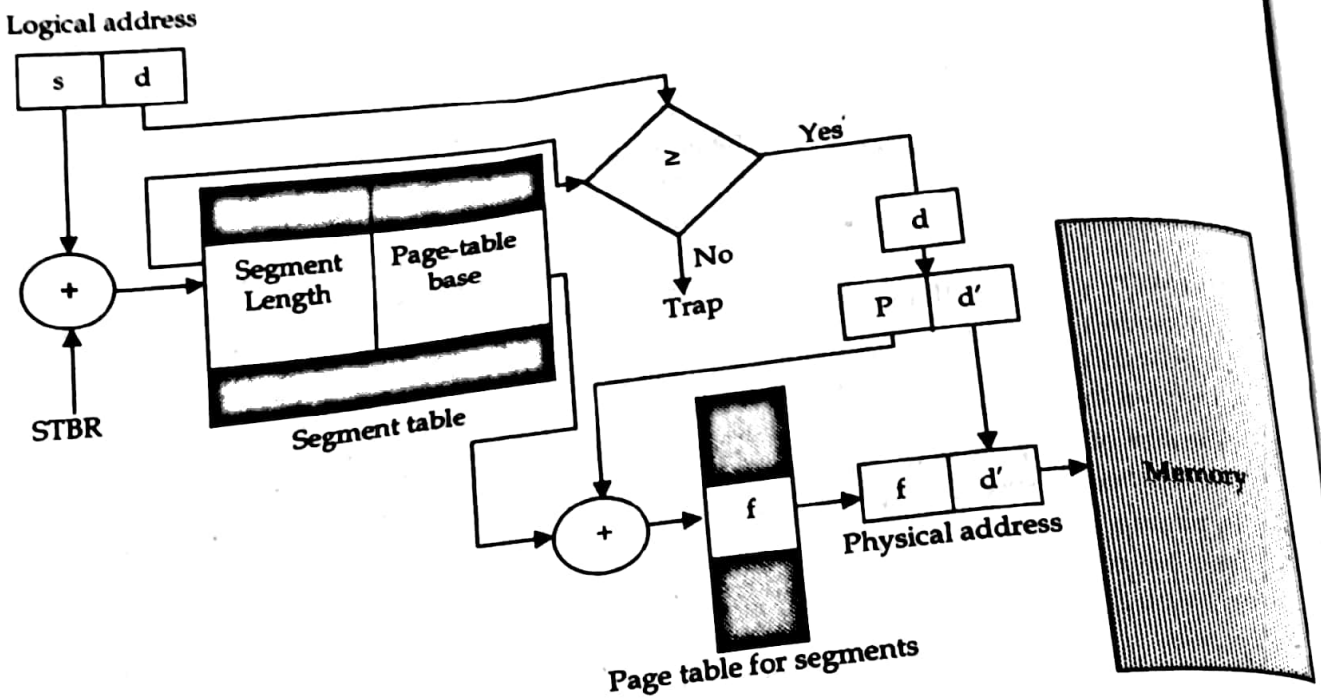


Fig 4.12: Segmentation with paging

When a memory reference occurred, the following algorithm was carried out.

- The segment number was used to find the segment descriptor.
- A check was made to see if the segment's page table was in memory. If it was, it was located. If it was not, a segment fault occurred. If there was a protection violation, a fault (trap) occurred.
- The page table entry for the requested virtual page was examined. If the page itself was not in memory, a page fault was triggered. If it was in memory, the main-memory address of the start of the page was extracted from the page table entry.
- The offset was added to the page origin to give the main memory address where the word was located.
- The read or store finally took place.

22. Fragmentation

As processes are loaded and removed from memory, the free memory space is broken into little pieces. It happens after sometimes that processes cannot be allocated to memory blocks considering their small size and memory blocks remains unused. This problem is known as Fragmentation. Fragmentation is a condition that occurs when we dynamically allocate the RAM to the processes, then many free memory blocks are available but they are not enough to load the process on RAM. There are two types of fragmentations:

- Internal fragmentation and
- External fragmentation

22.1 Internal fragmentation

Internal fragmentation occurs when fixed sized memory blocks are available and a process gets a block that is too much larger than the storage requirement of process. Internal fragmentation occurs when the memory is divided into fixed sized blocks. Whenever a process request for the memory, the fixed sized block is allocated to the process. In case the memory assigned to the process is somewhat larger than the memory requested, then the difference between assigned and requested memory is the internal fragmentation.

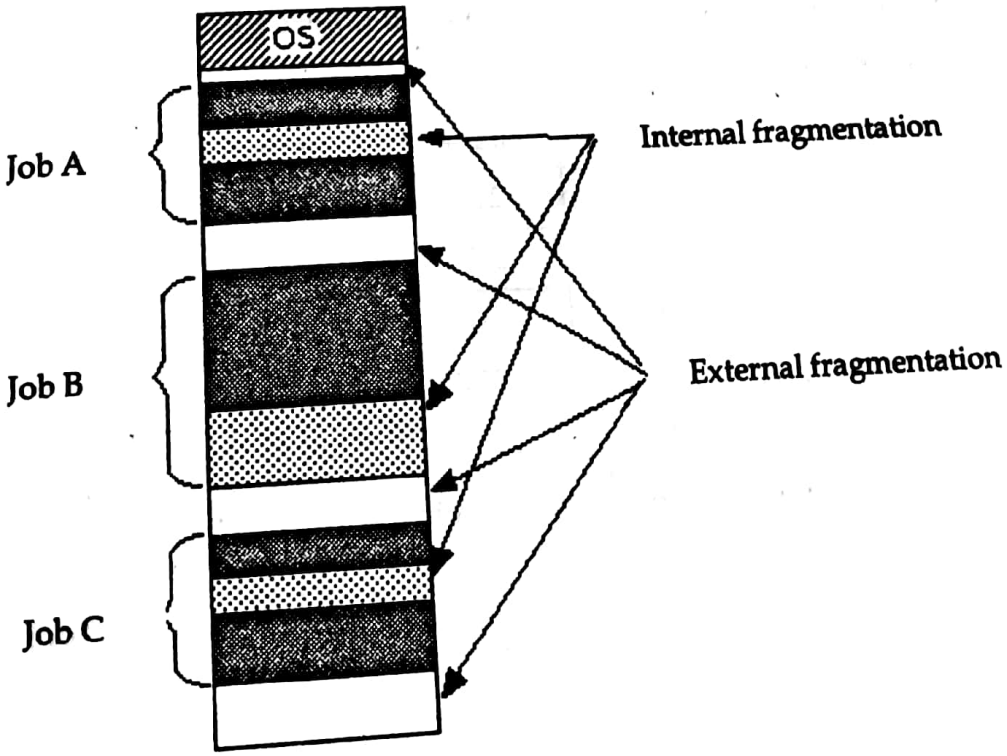
This leftover space inside the fixed sized block cannot be allocated to any process as it would not be sufficient to satisfy the request of memory by the process. Let us understand internal fragmentation with the help of an example. The memory space is partitioned into the fixed-sized blocks of 18,464 bytes. Let us say a process request for 18,460 bytes and partitioned fixed-sized block of 18,464 bytes is allocated to the process. The result is 4 bytes of 18,464 bytes remained empty which is the internal fragmentation.

22.2 External fragmentation

Total free RAM space is enough to load a process but the process still can't load because free blocks of RAM are not contiguous. In other words, we can say that all free blocks are not located together.

Characteristics of external fragmentation

- It exists when there is enough total memory space available to satisfy a request, but available memory space is not contiguous.
- Storage space is fragmented into large number of small holes.
- Both first fit and best fit strategies suffer from this.
- First fit is better in some systems, whereas best fit is better for other.
- Depending on the total amount of memory storage, size, external fragmentation may be minor or major problem.



Comparison between paging and segmentation

Basis for comparison	Internal fragmentation	External fragmentation
Basic	It occurs when fixed sized memory blocks are allocated to the processes.	It occurs when variable size memory spaces are allocated to the processes dynamically.
Occurrence	When the memory assigned to the process is slightly larger than the memory requested by the process this creates free space in the allocated block causing internal fragmentation.	When the process is removed from the memory, it creates the free space in the memory causing external fragmentation.
Solution	The memory must be partitioned into variable sized blocks and assign the best fit block to the process.	Compaction, paging and segmentation.

Differences between paging and segmentation

- The basic reason behind the occurrences of internal and external fragmentation is that internal fragmentation occurs when memory is partitioned into fixed-sized blocks whereas external fragmentation occurs when memory is partitioned into variable size blocks.
- When the memory block allotted to the process comes out to be slightly larger than requested memory, and then the free space left in the allotted memory block causes internal fragmentation. On the other hands, when the process is removed from the memory it creates free space causing a hole in the memory which is called external fragmentation.
- The problem of internal fragmentation can be solved by partitioning the memory into variable sized blocks and assign the best fit block to the requesting process. However, the solution for external fragmentation is compaction, but it is expensive to implement, so the processes must be allowed to acquire physical memory in a non-contiguous manner, to achieve this technique of paging and segmentation is introduced.

Numerical Problem 1: Consider the following segment table:

Segment	Base	Length
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

What are the physical addresses for the following logical addresses?

a. 0, 430
b. 1, 10
c. 2,500
d. 3,400
e. 4,112

Answer

a. 219 + 430 = 649
b. 2300 + 10 = 2310
c. illegal reference, trap to operating system
d. 1327 + 400 = 1727
e. illegal reference, trap to operating system

Laboratory Works

Simulate page replacement algorithms

Program 1: C-Program for simulating FIFO page replacement program

Algorithm

1. Start the process
2. Declare the size with respect to page length
3. Check the need of replacement from the page to memory
4. Check the need of replacement from old page to new page in memory
5. Forma queue to hold all pages
6. Insert the page require memory into the queue
7. Check for bad replacement and page fault
8. Get the number of processes to be inserted
9. Display the values
10. Stop the process

C-Source code

```
#include<stdio.h>
int main()
{
    int i, j, n, a[50], frame[10], no, k, avail, count=0;
    printf("\n Enter the number of pages:\n");
    scanf("%d", &n);
    printf("\n Enter the page number :\n");
    for(i=1; i<=n; i++)
    scanf("%d", &a[i]);
    printf("\n Enter the number of frames :");
    scanf("%d", &no);
    for(i=0; i<no; i++)
    frame[i]= -1;
    j=0;
    printf("\t ref string\t page frames\n");
    for(i=1; i<=n; i++)
    {
        printf("%d\t\t", a[i]);
        avail=0;
        for(k=0; k<no; k++)
            if(frame[k]==a[i])
                avail=1;
        if (avail==0)
        {
```

```
                        frame[j]=a[i];
                        j=(j+1)%no;
                        count++;
                        for(k=0; k<no; k++)
                        printf("%d\t", frame[k]);
                    }
                printf("\n");
            }
        printf("Page Fault Is %d", count);
        return 0;
}
```

Program 2: C-Program for simulating second chance page replacement program

```
#include<stdio.h>
#define SIZE 3
int full=0; //To check whether all frames are filled
int a[21]; //To take the input
int ref[SIZE]; //This is for reference bits for each frame
int frame[SIZE];
int repptr = 0; //Initialized to first frame
int count = 0;
int display()
{
        int i;
        printf("\n The elements in the frame are\n");
        for(i=0; i<full; i++)
        printf("%d\n", frame[i]);
}
int Pagerep(int ele)
{
        int temp;
        while(ref[repptr]!=0)
        {
                ref[repptr++]=0;
                if(repptr==SIZE)
                        repptr=0;
        }
        temp = frame[repptr];
        frame[repptr]=ele;
        ref[repptr]=1; //The latest page reference, hence it is set to 1
        return temp;
}
int Pagefault(int ele)
{
        if(full!=SIZE)
        {
                ref[full]=1;//All the ref bits are set to 1 as each frame is being filled firstly
```

```
        frame[full++]=ele;
        }
        else
                printf("The page replaced is %d", Pagerep(ele));

}
int Search(int ele)
{       int i, flag;
        flag=0;
        if(full!=0)
        {
                for(i=0; i<full; i++)
                        if(ele==frame[i])
                        {
                                flag=1;
                                ref[i]=1;
                                break;
                        }
        }
        return flag;

}
int main()
{
        int n, i;
        printf("The number of elements in the reference string are :");
        scanf("%d", &n);
        printf("%d", n);
        printf("Enter elements");
        for(i=0; i<n; i++)
                scanf(%d", &a[i]);
        printf("\n The elements present in the string are\n");
        for(i=0; i<n; i++)
                printf("%d  ", a[i]);
        printf("\n\n");
        for(i=0; i<n; i++)
        {
          if(Search(a[i])!=1)
          {
                        Pagefault(a[i]);
                        display();
                        count++;
          }
    }
    printf("\n The number of page faults are %d\n", count);
    getche();
     return 0;
}
```

Program 3: C-Program for simulating optimal page replacement program

```
#include<stdio.h>
int main()
{
  int no_of_frames, no_of_pages, frames[10], pages[30], temp[10], flag1, flag2;
  int flag3, i, j, k, pos, max, faults = 0;
  printf("Enter number of frames: ");
  scanf("%d", &no_of_frames);
  printf("Enter number of pages: ");
  scanf("%d", &no_of_pages);
  printf("Enter page reference string: ");
  for(i = 0; i < no_of_pages; ++i)
  {
    scanf("%d", &pages[i]);
  }
  for(i = 0; i < no_of_frames; ++i)
  {
    frames[i] = -1;
  }
  for(i = 0; i < no_of_pages; ++i)
  {
    flag1 = flag2 = 0;
    for(j = 0; j < no_of_frames; ++j)
    {
      if(frames[j] == pages[i])
      {
        flag1 = flag2 = 1;
        break;
      }
    }
    if(flag1 == 0)
    {
      for(j = 0; j < no_of_frames; ++j)
      {
        if(frames[j] == -1)
        {
          faults++;
          frames[j] = pages[i];
          flag2 = 1;
          break;
        }
      }
    }
    if(flag2 == 0)
    {
      flag3 =0;
      for(j = 0; j < no_of_frames; ++j)
      {
        temp[j] = -1;
        for(k = i + 1; k < no_of_pages; ++k)
```

```
                {
                if(frames[j] == pages[k])
                        {
                        temp[j] = k;
                        break;
                }
        }
        }
        for(j = 0; j < no_of_frames; ++j)
        {
        if(temp[j] == -1)
                {
                pos = j;
                flag3 = 1;
                break;
        }
        }
        if(flag3 ==0)
        {
        max = temp[0];
        pos = 0;
        for(j = 1; j < no_of_frames; ++j)
                {
                if(temp[j] > max)
                        {
                        max = temp[j];
                        pos = j;
                }
        }
        }
                        frames[pos] = pages[i];
                        faults++;

    }
    printf("\n");
    for(j = 0; j < no_of_frames; ++j)
    {
        printf("%d\t", frames[j]);
    }
  }
  printf("\n\nTotal Page Faults = %d", faults);
  return 0;
}
```

Program 4: C-Program for simulating LRU page replacement program

```
#include<stdio.h>
int main()
{
    int frames[10], temp[10], pages[10];
    int total_pages, m, n, position, k, l, total_frames;
    int a = 0, b = 0, page_fault = 0;
```

```
printf("\n Enter Total Number of Frames:\t");
scanf("%d", &total_frames);
for(m = 0; m < total_frames; m++)
{
    frames[m] = -1;
}
printf("Enter Total Number of Pages:\t");
scanf("%d", &total_pages);
printf("Enter Values for Reference String:\n");
for(m = 0; m < total_pages; m++)
{
    printf("Value No.[%d]:\t", m + 1);
    scanf("%d", &pages[m]);
}
for(n = 0; n < total_pages; n++)
{
    a = 0, b = 0;
    for(m = 0; m < total_frames; m++)
    {
        if(frames[m] == pages[n])
        {
            a = 1;
            b = 1;
            break;
        }
    }
    if(a == 0)
    {
        for(m = 0; m < total_frames; m++)
        {
            if(frames[m] == -1)
            {
                frames[m] = pages[n];
                b = 1;
                break;
            }
        }
    }
}
if(b == 0)
{
    for(m = 0; m < total_frames; m++)
    {
        temp[m] = 0;
    }
    for(k = n - 1, l = 1; l <= total_frames - 1; l++, k--)
    {
        for(m = 0; m < total_frames; m++)
```

```
            {
                if(frames[m] == pages[k])
                {
                    temp[m] = 1;
                }
            }
        }
        for(m = 0; m < total_frames; m++)
        {
            if(temp[m] == 0)
            position = m;
        }
        frames[position] = pages[n];
        page_fault++;
    }
    printf("\n");
    for(m = 0; m < total_frames; m++)
    {
        printf("%d\t", frames[m]);
    }
}
printf("\n Total Number of Page Faults:\t %d\n", page_fault);
return 0;
}
```

Program 5: C-Program for simulating CLOCK page replacement program

```
#include<stdio.h>
int main()
{
    int n, p[100], f[10], ava, hit=0,usebit[10], i, j;
    printf("enter the length of the Reference string: ");
    scanf("%d", &n);
    printf("enter the reference string: \n");
    for(i=0; i<n; i++)
    scanf("%d", &p[i]);
    for(i=0; i<n; i++)
    {
        ava=0;
        for(j=0;j<3;j++)
        {
            if(p[i]==f[j])
            {
                ava=1;
                hit++;
                usebit[j]=1;
                break;
            }
```

```
        }
    //search for use bit 0
        if(ava==0)
        {
            for(j=0;j<3;j++)
            {
                if(usebit[j]==0)
                {
                    f[j]=p[i];
                    usebit[j]=1;
                    ava=1;
                    break;
                }
            }
        }
        if(ava==0)
        {
    for(j=0;j<3;j++)
    usebit[j]=0;
        }
        f[0]=p[i];
        usebit[0]=1;
    }
    printf("The number of Hits: %d", hit);
    return 0;
}
```

Exercise

1. Given five memory partitions of 100 KB, 500 KB, 200 KB, 300 KB, and 600 KB (in order), how would each of the first-fit, best-fit, and worst-fit algorithms place processes of 212 KB, 417 KB, 112 KB, and 426 KB (in order)?Which algorithm makes the most efficient use of memory?
2. Most systems allow programs to allocate more memory to its address space during execution. Data allocated in the heap segments of programs is an example of such allocated memory. What is required to support dynamic memory allocation in the following schemes?
   a) contiguous-memory allocation
   b) pure segmentation
   c) pure paging
3. On a system with paging, a process cannot access memory that it does not own; why? How could the operating system allow access to other memory? Why should it or should it not?
4. Compare paging with segmentation with respect to the amount of memory required by the address translation structures in order to convert virtual addresses to physical addresses.
5. Why are segmentation and paging sometimes combined into one scheme?

6. Explain why it is easier to share a reentrant module using segmentation than it is to do so when pure paging is used.
7. Consider the following segment table:

Segment	Base	Length
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

What are the physical addresses for the following logical addresses?

8. What is the purpose of paging the page tables?
9. Consider the hierarchical paging scheme used by the VAX architecture. How many memory operations are performed when a user program executes a memory load operation?
10. Compare the segmented paging scheme with the hashed page table's scheme for handling large address spaces. Under what circumstances is one scheme preferable over the other?
11. Discuss the hardware support required to support demand paging.
12. What is the copy-on-write feature and under what circumstances is it beneficial to use this feature? What is the hardware support required to implement this feature?
13. What is the cause of thrashing? How does the system detect thrashing? Once it detects thrashing, what can the system do to eliminate this problem?
14. Is it possible for a process to have two working sets? One representing data and another representing code? Explain.
15. Suppose that your replacement policy (in a paged system) is to examine each page regularly and to discarding that page if it has not been used since the last examination. What would you gain and what would you lose by using this policy rather than LRU or second-chance replacement?
16. Discuss situations under which the least frequently used page-replacement algorithm generates fewer page faults than the least recently used page replacement algorithm. Also discuss under what circumstance does the opposite holds.
17. Given five memory partitions of 100 KB, 500 KB, 200 KB, 300 KB, and 600 KB (in order), how would each of the first-fit, best-fit, and worst-fit algorithms place processes of 212 KB, 417 KB, 112 KB, and 426 KB (in order)?Which algorithm makes the most efficient use of memory?
18. Explain the difference between logical and physical addresses.
19. Explain the difference between internal and external fragmentation.
20. Why is it that, on a system with paging, a process cannot access memory it does not own? How could the operating system allow access to other memory? Why should it or should it not?

❑❑❑

Chapter 5
FILE MANAGEMENT

1. File Overview

A file is a named collection of related information that is recorded on secondary storage such as magnetic disks, magnetic tapes and optical disks. In general, a file is a sequence of bits, bytes, lines or records whose meaning is defined by the files creator and user.

In the Microsoft Windows family of operating systems, users are presented with several different choices of file systems when formatting such media. These choices depend on the type of media involved and the situations in which the media is being formatted. The two most common file systems in Windows are as follows:

- NTFS
- FAT
- exFAT
- HFS Plus
- EXT

File systems can differ between operating systems (OS), such as Microsoft Windows, MacOS and Linux-based systems. Some file systems are designed for specific applications. Major types of file systems include distributed file systems, disk-based file systems and special purpose file systems.

A file system stores and organizes data and can be thought of as a type of index for all the data contained in a storage device. These devices can include hard drives, optical drives and flash drives. File systems specify conventions for naming files, including the maximum number of characters in a name, which characters can be used and, in some systems, how long the file name suffix can be. In many file systems, file names are not case sensitive.

2. File Naming

When a process creates a file, it gives the file name; while process terminates, the file continue to exist and can be accessed by other processes. A file is named, for the convenience of its human users, and it is referred to by its name. A name is string of characters. The string may be of digits or special characters (e.g. 2, !,% etc.). Some system differentiates between the upper and lower case character, whereas other system considers the equivalent (likeUNIX and MS-DOS). Normally the string of max 8 characters are legal file name (e.g., in DOS), but many recent system support as long as 255 characters (e.g. Windows 2000).

Many OS support two-part file names; separated by period; the part following the period is called the file extension and usually indicates something about the file (e.g., file.c Cprogramming source file). But in some system it may have two or more extension such as in UNIXproc.c.Z ((C Programming source file compressed using Ziv-Lampel algorithm. In some system e.g. UNIX, file extension are just conventions; in other system it requires (e.g., C compiler must requires .c source file).

The name of each file must be unique within the directory where it is stored. This ensures that the file also has a unique path name in the file system. File naming guidelines are:

- A file name can be up to 255 characters long and can contain letters, numbers, and underscores.
- The operating system is case-sensitive, which means it distinguishes between uppercase and lowercase letters in file names. Therefore, FILEA, FiLea, and filea are three distinct file names, even if they reside in the same directory.
- File names should be as descriptive and meaningful as possible.
- Directories follow the same naming conventions as files.
- Certain characters have special meaning to the operating system. Avoid using these characters when you are naming files. These characters include the following:
  / \ " ' * ; - ? [ ] ( ) ~ ! $ { } <> # @ & | space tab newline
- A file name is hidden from a normal directory listing if it begins with a dot (.). When the **ls** command is entered with the **-a** flag, the hidden files are listed along with regular files and directories.

Here, the table given below lists the most common file extensions with their meaning:

File Extension	File Meaning
myfile.bak	This indicates backup file
myfile.c	This indicates C programming language source file
myfile.gif	This indicates gif format image file
myfile.hlp	This indicates help file
myfile.html	This indicates HyperText Markup Language (HTML) file
myfile.jpg	This indicates jpg format image file
myfile.mp3	This indicates mp3 music or audio file in which music encoded in MPEG layer 3 audio format
myfile.mpg	This indicates mpg video file in which movie encoded with the MPEG standard
myfile.o	This indicates an object file
myfile.pdf	This indicates Portable Document Format (PDF) file
myfile.ps	This indicates PostScript file
myfile.tex	This indicates input for the TEX formatting program
myfile.txt	This indicates normal textual file
myfile.zip	This indicates compressed archive

3. Basic File Operations

There are many file operations that can be perform by the computer system some of them are listed below:

- **Create**: find space for the file and make an entry in the directory.
- **Open**: find file and determine if it has already been opened. If not open search directory, cache information, add entry in per-process open-file table. If open check lock and cache information if lock can be acquired. Increment the open count.
- **Close**: decrement the open count and remove the file's entry from the open-file table if count reaches zero.
- **Read**: read data from the file.
- **Write**: write data to the file.
- **Delete**: search directory, release file space and erase directory entry.
- **Reposition**: reposition the file position pointer. This is more commonly known as seek.
- **Truncate**: delete content of a file, but keep file properties.
- **Lock**: file locks provide concurrency control. A shared lock allows multiple readers to acquire a lock concurrently, while exclusive lock ensures only one writer can modify a file. With mandatory locking the operating system ensures locking integrity, while with advisory locking the application process ensures that the correct locking strategy is followed. The Windows operating system uses the mandatory locking strategy.

4. File Structure

A File Structure should be according to a required format that the operating system can understand.

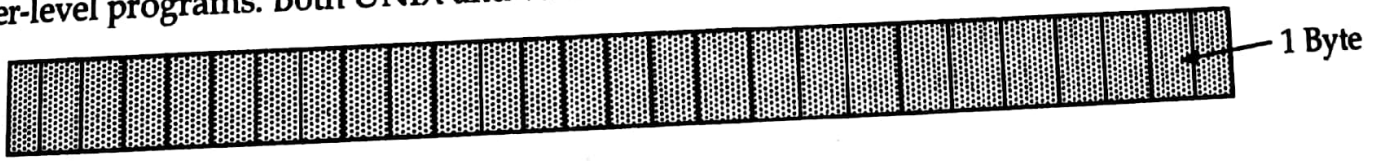
- A file has a certain defined structure according to its type.
- A text file is a sequence of characters organized into lines.
- A source file is a sequence of procedures and functions.
- An object file is a sequence of bytes organized into blocks that are understandable by the machine.
- When operating system defines different file structures, it also contains the code to support these file structure. UNIX, MS-DOS support minimum number of file structure.

Files can be structured in several ways in which three common structures are given below:

- Unstructured sequence of bytes
- Sequence of fixed-length records
- Tree of records

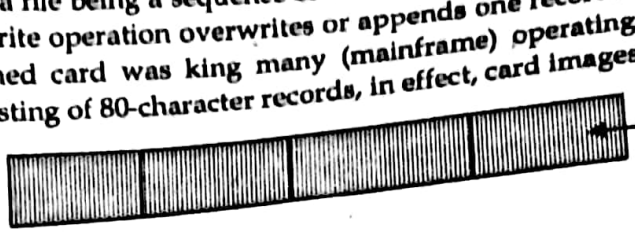
4.1 Byte Sequence

The file in Fig below is just an unstructured sequence of bytes. In effect, the operating system does not know or care what is in the file. All it sees are bytes. Any meaning must be imposed by user-level programs. Both UNIX and Windows 98 use this approach.



4.2 Record Sequence

In this model, a file is a sequence of fixed-length records, each with some internal structure. Central to the idea of a file being a sequence of records is the idea that the read operation returns one record and the write operation overwrites or appends one record. As a historical note, when the 80-column punched card was king many (mainframe) operating systems based their file systems on files consisting of 80-character records, in effect, card images.



4.3 Tree

In this organization, a file consists of a tree of records, not necessarily all the same length, each containing a key field in a fixed position in the record. The tree is sorted on the key field, to allow rapid searching for a particular key.

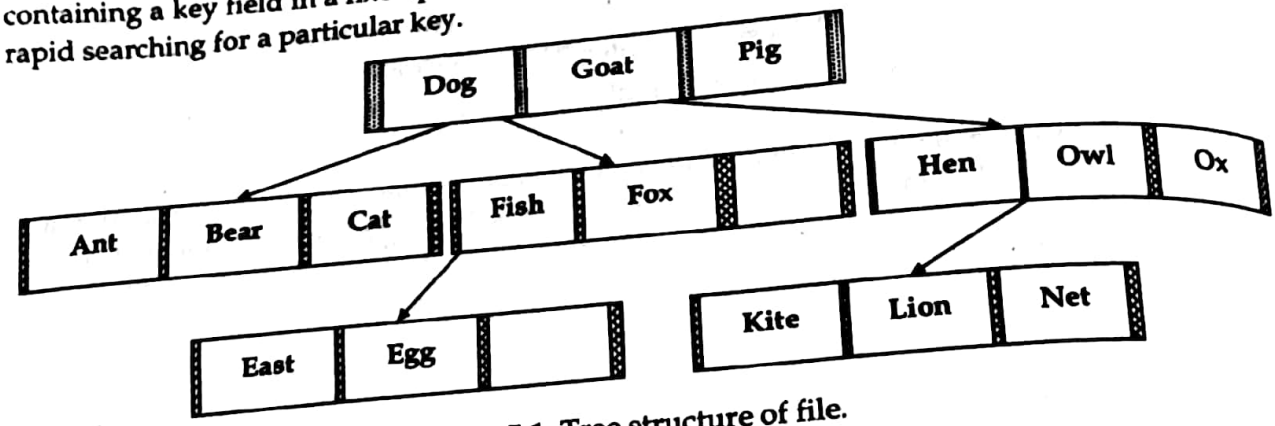


Fig 5.1: Tree structure of file.

5. File Types

Many OS supports several types of files

- **Regular files**: contains user information, are generally ASCII or binary.
- **Directories**: system files for maintaining the structure of file system.
- **Character Special files**: related to I/O and used to model serial I/O devices such as terminals, printers, and networks.
- **Block special files**: used to model disks
- **ASCII files:** Consists of line of text. Each line is terminated either by carriage return character or by line feed character or both. They can be displayed and printed as is and can be edited with ordinary text editor.
- **Binary files:** Consists of sequence of byte only. They have some internal structure known to programs that use them (e.g., executable or archive files). Many OS use extension to identify the file types; but UNIX like OS use a magic number to identify file types.

6. File Access

Files store information. When it is used, this information must be accessed and read into computer memory. The information in the file can be accessed in several ways. Some systems

provide only one access method for files. Other system, such as those of IBM, supports many access methods, and choosing the right one for a particular application is a major design problem. The access methods are Sequential and Direct access

6.1 Sequential Access

The simplest access method; Information in the file is processed in order, one record after the other starting at the beginning, but could not skip around and read them out of order. The files could be rewound. It is convenient when the storage medium is magnetic tape. It is used in many early systems. Process files from beginning to end, in order, one record after the other. For example, if we want to read a document from beginning to end, we typically start at the beginning and read page, by page until we reach the end.

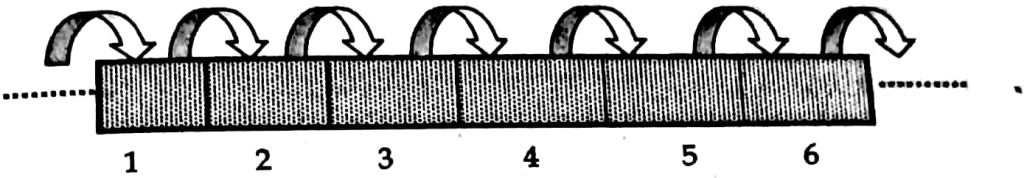


Fig 5.2: Sequential File access

Advantages of sequential access of file

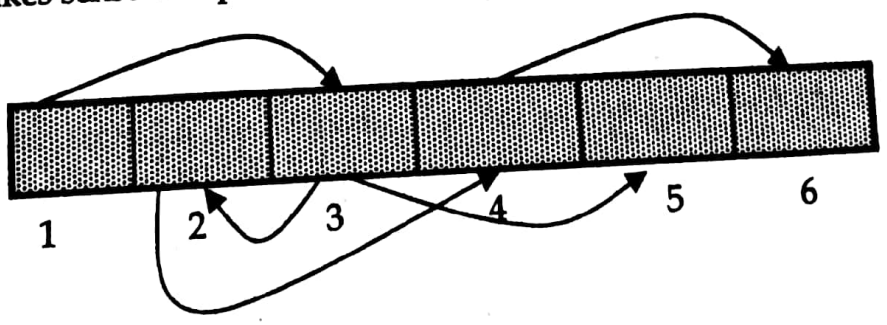
- The method is simple & easy to understand.
- Sequential files are easy to organize and maintain.
- Loading or reading a record requires only the Record Key.
- It is efficient & economical if the number of file records to be processed is high.
- Relatively inexpensive Input/output media and devices may be used.
- Errors in the files remain localized.

Disadvantages of sequential file organization

- The sorting does not remove the need to access other records as the search looks for particular records.
- Sequential records cannot support modern technologies that require fast access to stored records.
- The requirement that all records be of the same size is sometimes difficult to enforce.

6.2 Direct Access

Files whose bytes or records can be read in any order are called direct access. It is based on disk model of file, since disks allow random access to any block. It is used for immediate access to large amounts of information. When a query concerning a particular subject arrives, we compute which block contain the answer, and then read that block directly to provide desired information. Here process file by accessing the content in no specific order. For example, if we only want to read page 1013 it makes sense to reposition (seek) to page 1013 and read the page.



Advantages of Random File Access

- Quick retrieval of records.
- The records can be of different sizes.

Disadvantages of Random File Organization

- Data may be accidentally erased or overwritten unless special precautions are taken.
- Random files are less efficient in the use of storage space compared to sequentially organized files.
- Expensive hardware and software resources are required.
- Relatively complex when programming
- System design based on random file organization is complex and costly

7. File Attributes

File attributes are settings associated with computer files that grant or deny certain rights to how a user or the operating system can access that file. For example, IBM compatible computers running MS-DOS or Microsoft Windows have capabilities of having read, archive, system, and hidden attributes. Following are some of the attributes of a file:

- **Name:** It is the only information which is in human-readable form.
- **Identifier.** The file is identified by a unique tag(number) within file system.
- **Type.** It is needed for systems that support different types of files.
- **Location.** Pointer to file location on device.
- **Size.** The current size of the file.
- **Protection.** This controls and assigns the power of reading, writing, executing.
- **Time, date, and user identification.** This is the data for protection, security, and usage monitoring.
- **Read-only:** Allows a file to be read, but nothing can be written to the file or changed.
- **Archive:** Tells Windows Backup to backup the file.
- **System:** System file.
- **Hidden:** File will not be shown when doing a regular dir from DOS.

8. File Operations

OS provides system calls to perform operations on files. Some common calls are:

- **Create:** Creation of the file is the most important operation on the file. Different types of files are created by different methods for example text editors are used to create a text file, word processors are used to create a word file and Image editors are used to create the image files.
- **Open:** Before using a file, a process must open it.
- **Write:** Writing the file is different from creating the file. The OS maintains a write pointer for every file which points to the position in the file from which, the data needs to be written.
- **Read:** Every file is opened in three different modes: Read, Write and append. A Read pointer is maintained by the OS, pointing to the position up to which, the data has been read.

- **Re-position:** Re-positioning is simply moving the file pointers forward or backward depending upon the user's requirement. It is also called as seeking.
- **Delete:** Deleting the file will not only delete all the data stored inside the file, It also deletes all the attributes of the file. The space which is allocated to the file will now become available and can be allocated to the other files.
- **Truncate:** Truncating is simply deleting the file except deleting attributes. The file is not completely deleted although the information stored inside the file gets replaced.
- **Close:** When all access is finished, the file should be closed to free up the internal table space.
- **Append:** Add data at the end of the file.
- **Seek:** Repositions the file pointer to a specific place in the file.
- **Get attributes:** Returns file attributes for processing.
- **Set attributes:** To set the user settable attributes when file changed.
- **Rename:** Rename file.

9. Directory Structure

A directory is a node containing information about files. It is also called folder. Directory can be defined as the listing of the related files on the disk. The directory may store some or the entire file attributes.

To get the benefit of different file systems on the different operating systems, a hard disk can be divided into the number of partitions of different sizes. The partitions are also called volumes or mini disks. Each partition must have at least one directory in which, all the files of the partition can be listed. A directory entry is maintained for each file in the directory which stores all the information related to that file.

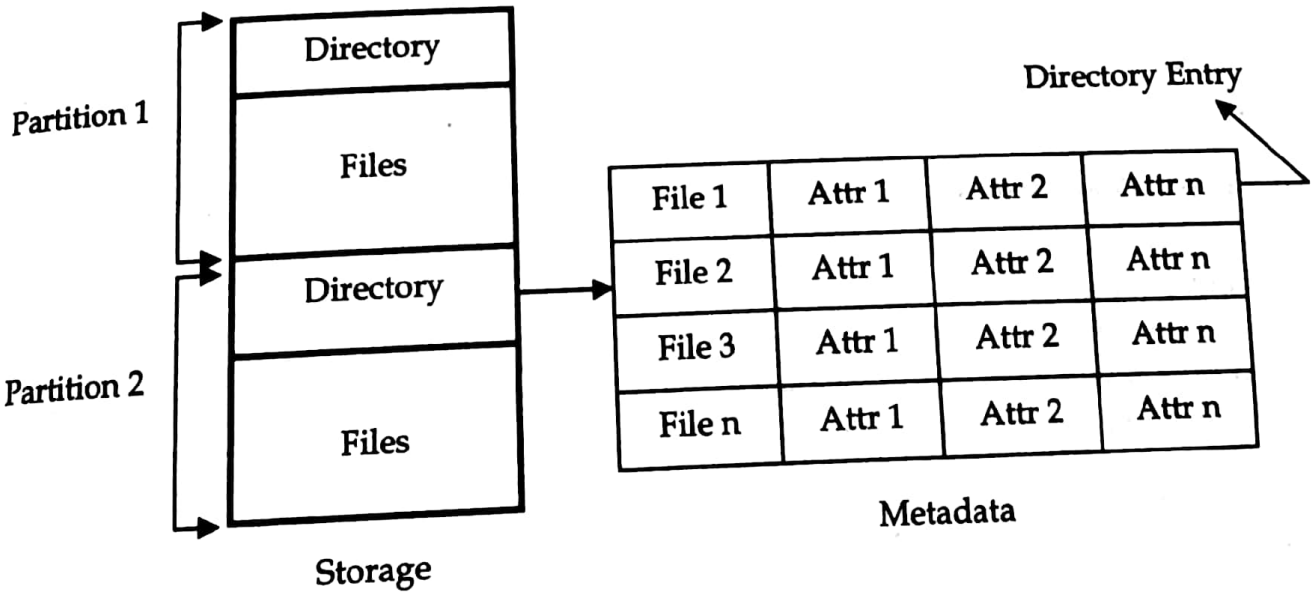


Fig 5.3: Directory structure.

Directories can have different structures.

9.1 Single-Level-Directory

The simplest form of directory system is having one directory containing all the files.It is also called root directory. The entire system will contain only one directory which is supposed to mention all the files present in the file system. The directory contains one entry per each file present on the file system. It is easy to support and understand; but difficult to manage large amount of files and to manage different users. An example of a system with one directory is given below.

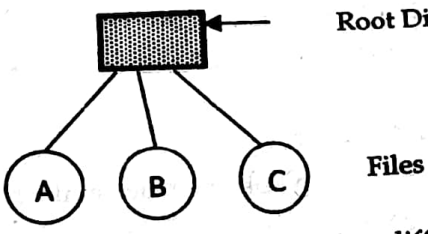


Fig 5.4: A single level directory containing different files.

The problem with having only one directory in a system with multiple users is that different users may accidentally use the same names for their files. For examples, if A creates a file name called Ram, and then later user B also creates a file name called Ram, B's file will overwrite A's file. Consequently, this scheme is not used on multiuser system.

Advantages

- Implementation is very simple.
- If the sizes of the files are very small then the searching becomes faster.
- File creation, searching, deletion is very simple since we have only one directory.

Disadvantages

- We cannot have two files with the same name.
- The directory may be very big therefore searching for a file may take so much time.
- Protection cannot be implemented for multiple users.
- There are no ways to group same kind of files.
- Choosing the unique name for every file is a bit complex and limits the number of files in the system because most of the Operating System limits the number of characters used to construct the file name.

9.2 Two-Level-Directory

To avoid conflicts caused by different users choosing the same file name for their own files, the two level directory systems, we can create a separate directory for each user. There is one master directory which contains separate directories dedicated to each user. For each user, there is a different directory present at the second level, containing group of user's file. The system doesn't let a user to enter in the other user's directory without permission.

The example of this system is shown below.

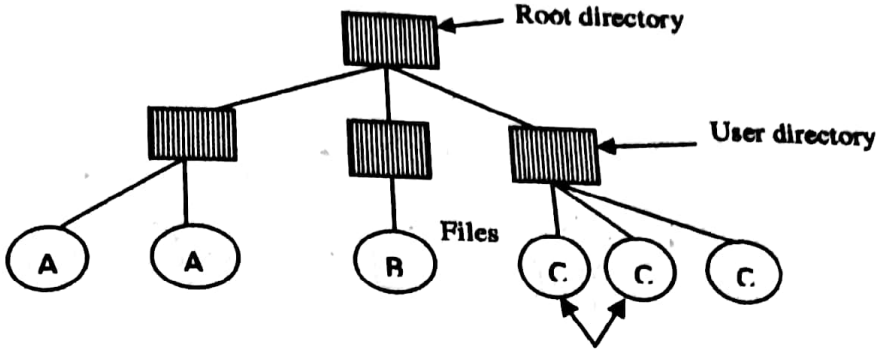


Fig 5.5: A two-level directory system

Characteristics of two level directory systems

- Each files has a path name as /User-name/directory-name/
- Different users can have the same file name.
- Searching becomes more efficient as only one user's list needs to be traversed.
- The same kind of files cannot be grouped into a single directory for a particular user.

9.3 Hierarchical-Directory

In Tree structured directory system, any directory entry can either be a file or sub directory. Tree structured directory system overcomes the drawbacks of two level directory system. The similar kind of files can now be grouped in one directory.

Each user has its own directory and it cannot enter in the other user's directory. However, the user has the permission to read the root's data but he cannot write or modify this. Only administrator of the system has the complete access of root directory.

Searching is more efficient in this directory structure. The concept of current working directory is used. A file can be accessed by two types of path, either relative or absolute.

Absolute path is the path of the file with respect to the root directory of the system while relative path is the path with respect to the current working directory of the system. In tree structured directory systems, the user is given the privilege to create the files as well as directories. This approach is shown below.

Here, the directory A, B, C contained in the root directory each belong to different user, two of whom have created subdirectories for projects they are working on.

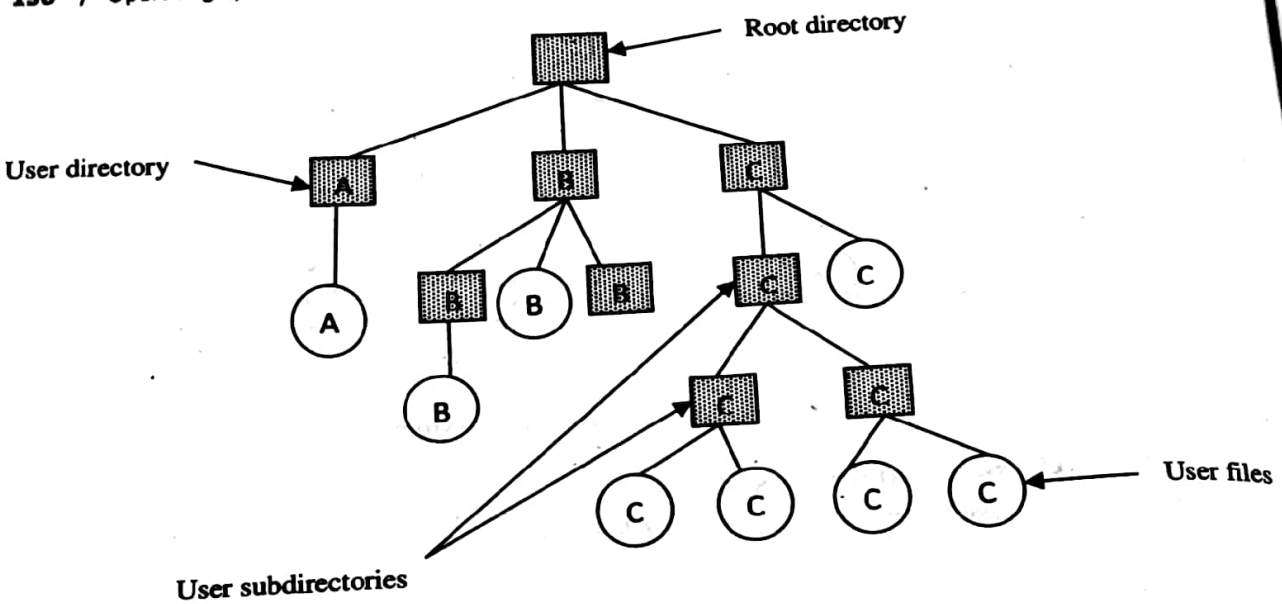


Fig 5.6: A hierarchical directory system.

9.4 File System Layout

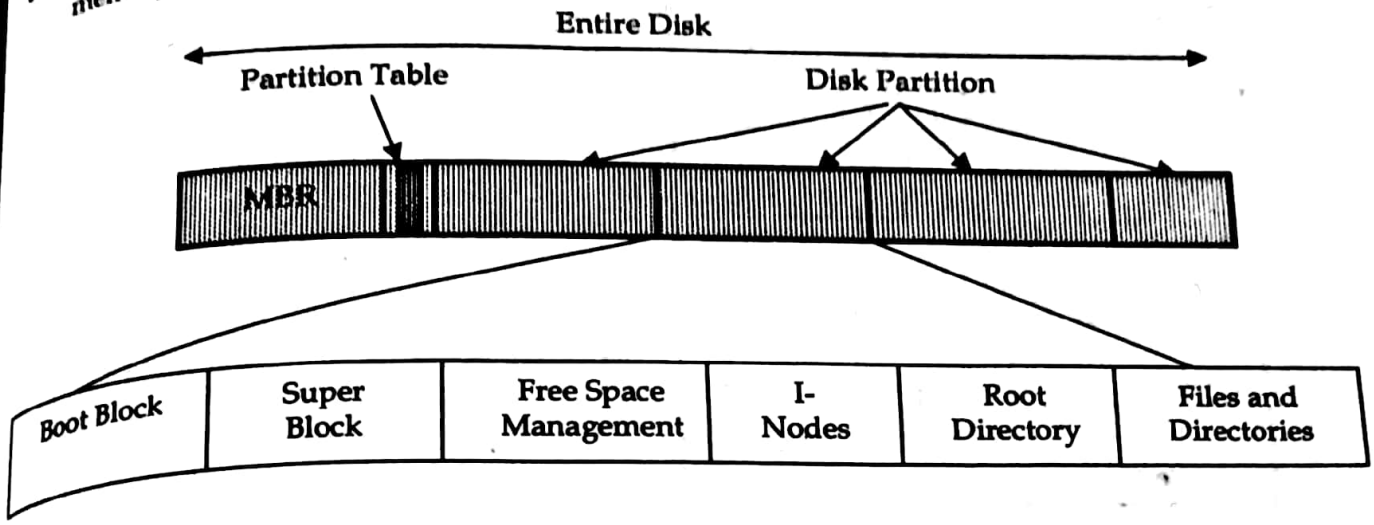
A file system is a set of files, directories, and other structures. File systems maintain information and identify where a file or directory's data is located on the disk. In addition to files and directories, file systems contain a boot block, a superblock, bitmaps, and one or more allocation groups. An allocation group contains disk i-nodes and fragments. Each file system occupies one logical volume.

Basically, file systems are stored on the disks. Almost all disks can be divided up into multiple partitions with independent file systems on each partition. Here, in the partition of the disk, Sector 0 is called as Master Boot Record (MBR), is used to boot the computer system. The Master Boot Record's end contains the partition table. That partition table gives the starting and ending addresses of each partition of the disk. From those partitions in the table, one is marked as active. So that, whenever the computer system is booted up, the BIOS read in and execute the Master Boot Record.

The very first thing that the master boot record program does is, locate the active partition, read in its first block that is called as the boot block and execute it. Now the program present inside the boot block loads the OS that contained in that partition. For the purpose of uniformity, each and every partition starts with a boot block, even if it doesn't contain a bootable OS. File Systems are stored on disks. The figure below depicts a possible File-System Layout.

- **MBR:** Master Boot Record is used to boot the computer
- **Partition Table:** Partition table is present at the end of MBR. This table gives the starting and ending addresses of each partition.
- **Boot Block:** When the computer is booted, the BIOS read in and execute the MBR. The first thing the MBR program does is locate the active partition, read in its first block, which is called the boot block, and execute it. The program in the boot block loads the operating system contained in that partition. Every partition contains a boot block at the beginning though it does not contain a bootable operating system.

- **Super Block:** It contains all the key parameters about the file system and is read into memory when the computer is booted or the file system is first touched.



10. Implementing Files

The allocation methods define how the files are stored in the disk blocks. There are three main disk space or file allocation methods.

- Contiguous Allocation
- Linked Allocation
- Indexed Allocation

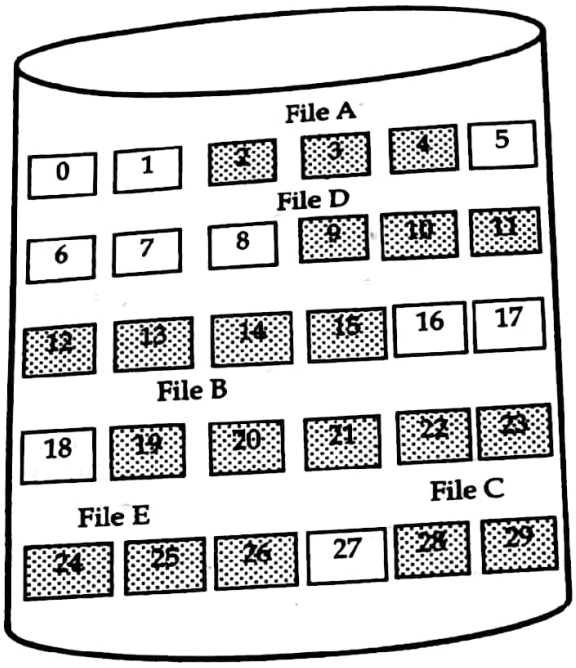
10.1 Implementing Files by contiguous allocation

A single continuous set of blocks is allocated to a file at the time of file creation. Thus, this is a pre-allocation strategy, using variable size portions. The file allocation table needs just a single entry for each file, showing the starting block and the length of the file. This method is best from the point of view of the individual sequential file. Multiple blocks can be read in at a time to improve I/O performance for sequential processing.

In this scheme, each file occupies a contiguous set of blocks on the disk. For example, if a file requires n blocks and is given a block b as the starting location, then the blocks assigned to the file will be: b, b+1, b+2,……b+n–1. This means that given the starting block address and the length of the file (in terms of blocks required), we can determine the blocks occupied by the file. The directory entry for a file with contiguous allocation contains

- Address of starting block
- Length of the allocated portion.

The file B in the following figure starts from the block 19 with length = 5 blocks. Therefore, it occupies 19, 20, 21, 22, 23 blocks.



File Allocation Table

File Name	Start Block	Length
File A	2	3
File B	19	5
File C	28	2
File D	9	7
File E	24	3

Fig 5.7: Contiguous allocation of disk space

Advantages

- Both the Sequential and Direct Accesses are supported by this. For direct access, the address of the k^{th} block of the file which starts at block b can easily be obtained as (b+k).
- This is extremely fast since the number of seeks are minimal because of contiguous allocation of file blocks.

Disadvantages

- This method suffers from both internal and external fragmentation. This makes it inefficient in terms of memory utilization.
- Compaction algorithm will be necessary to free up additional space on disk.
- Increasing file size is difficult because it depends on the availability of contiguous memory at a particular instance.
- Also, with pre-allocation, it is necessary to declare the size of the file at the time of creation.

10.2 Implementing Files by Linked List Allocation

Allocation is on an individual block basis. Each block contains a pointer to the next block in the chain. Again the file table needs just a single entry for each file, showing the starting block and the length of the file. Although pre-allocation is possible, it is more common simply to allocate blocks as needed. Any free block can be added to the chain. The blocks need not be continuous. Increase in file size is always possible if free disk block is available. There is no external fragmentation because only one block at a time is needed but there can be internal fragmentation but it exists only in the last disk block of file.

In this scheme, each file is a linked list of disk blocks which need not be contiguous. The disk blocks can be scattered anywhere on the disk. The directory entry contains a pointer to the starting and the ending file block. Each block contains a pointer to the next block occupied by the

file. The file-A in following image shows how the blocks are randomly distributed. The last block (25) contains -1 indicating a null pointer and does not point to any other block.

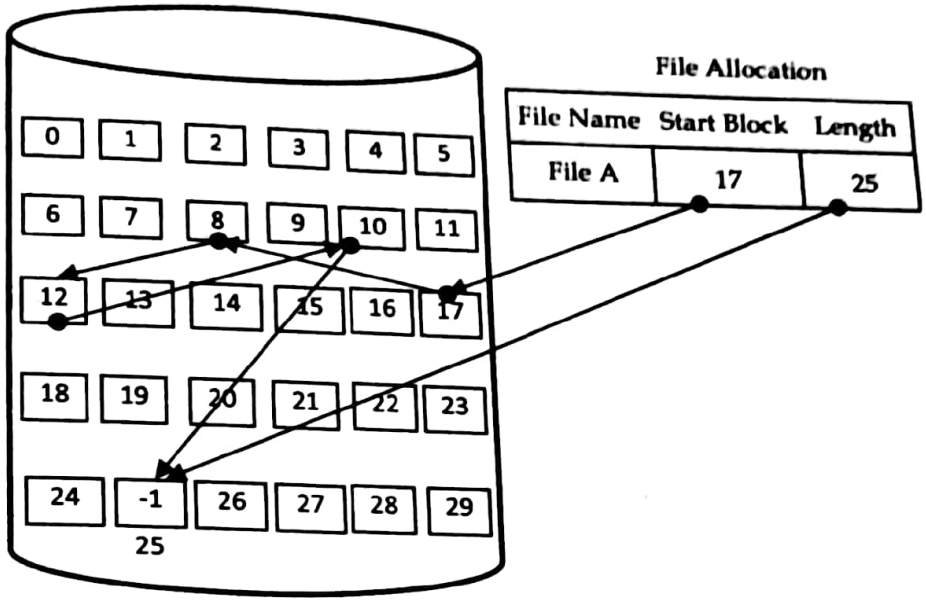


Fig 5.8: Linked list allocation of disk space

Advantages

- This is very flexible in terms of file size. File size can be increased easily since the system does not have to look for a contiguous chunk of memory.
- This method does not suffer from external fragmentation. This makes it relatively better in terms of memory utilization.
- Internal fragmentation exists in last disk block of file only.

Disadvantages

- Because the file blocks are distributed randomly on the disk, a large number of seeks are needed to access every block individually. This makes linked allocation slower.
- It does not support random or direct access. We cannot directly access the blocks of a file. A block k of a file can be accessed by traversing k blocks sequentially (sequential access) from the starting block of the file via block pointers.
- Pointers required in the linked allocation incur some extra overhead.

10.3 Index Allocation

It addresses many of the problems of contiguous and chained allocation. In this case, the file allocation table contains a separate one-level index for each file: The index has one entry for each block allocated to the file. Allocation may be on the basis of fixed-size blocks or variable-sized blocks. Allocation by blocks eliminates external fragmentation, whereas allocation by variable-size blocks improves locality. This allocation technique supports both sequential and direct access to the file and thus is the most popular form of file allocation.

In this scheme, a special block known as the Index block contains the pointers to all the blocks occupied by a file. Each file has its own index block. The i^{th} entry in the index block contains the disk address of the i^{th} file block.

The directory entry contains the address of the index block as shown in the image:

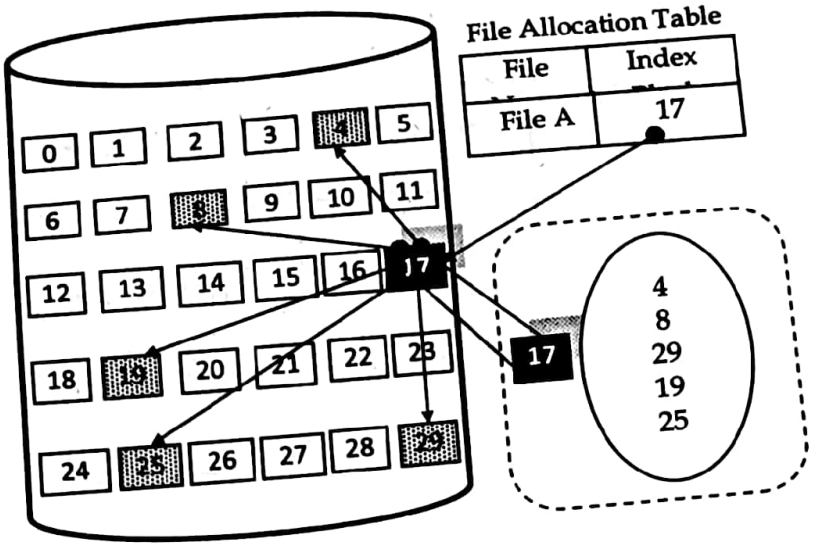


Fig 5.9: Indexed allocation of disk space

Advantages

- This supports direct access to the blocks occupied by the file and therefore provides fast access to the file blocks.
- It overcomes the problem of external fragmentation.

Disadvantages

- The pointer overhead for indexed allocation is greater than linked allocation.
- For very small files, say files that expand only 2-3 blocks, the indexed allocation would keep one entire block (index block) for the pointers which is inefficient in terms of memory utilization. However, in linked allocation we lose the space of only 1 pointer per block.

11. Linked List Allocation using Table in Memory

The main disadvantage of linked list allocation is that the Random access to a particular block is not provided. In order to access a block, we need to access all its previous blocks. File Allocation Table overcomes this drawback of linked list allocation. In this scheme, a file allocation table is maintained, which gathers all the disk block links. The table has one entry for each disk block and is indexed by block number. File allocation table needs to be cached in order to reduce the number of head seeks. Now the head doesn't need to traverse all the disk blocks in order to access one successive block.

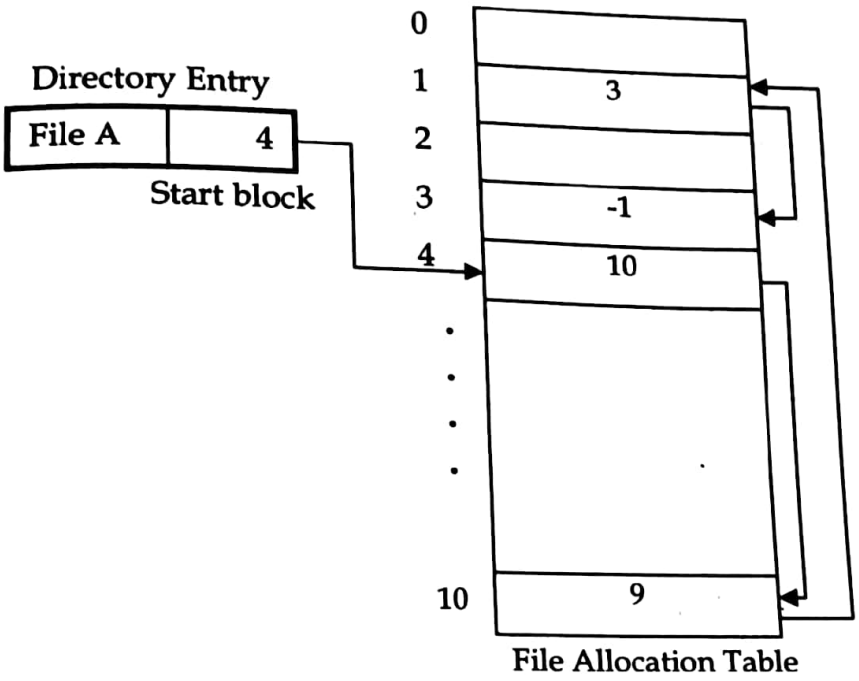
It simply accesses the file allocation table, read the desired block entry from there and access that block. This is the way by which the random access is accomplished by using FAT. It is used by MS-DOS and pre-NT Windows versions.

Advantages

- Uses the whole disk block for data.
- A bad disk block doesn't cause all successive blocks lost.
- Random access is provided although it's not too fast.
- Only FAT needs to be traversed in each file operation.

Disadvantages

- Each Disk block needs a FAT entry.
- FAT size may be very big depending upon the number of FAT entries.
- Number of FAT entries can be reduced by increasing the block size but it will also increase Internal Fragmentation.



Example: Consider a disk of 100 KB. If block size is 2 KB, calculate the size of FAT assuming that each entry in FAT takes 4 bytes.

Solution

Size of disk=100 GB =100 x 2^{20} KB = 104857600 KB

Size of block=2 KB

Thus, number of blocks=104857600/2 = 52428800

⇨ Number of Entries in FAT = 52428800

Since size of an entry= 4 byte

⇨ Size of FAT (File Allocation Table)= 52428800 x 4 byte = 200 MB

12. I-nodes

I-node (Index node) is a data structure which is used to identify which block belongs to which file. It contains the attributes and disk addresses of the file's blocks. Unlike the in-memory table the i-node need to be in memory only when the corresponding file is open.

List the attributes and disk address of the block. Which block belongs to which file, associate each file with a data structure called i-node. It is possible with i-node to find all the blocks of the file. The big advantage with this schema is that only i-node is in memory when its corresponding file is open.

All the attributes for the file are stored in an I-node entry, which is loaded into memory when the file is opened. The I-node also contains a number of direct pointers to disc blocks. I-node need

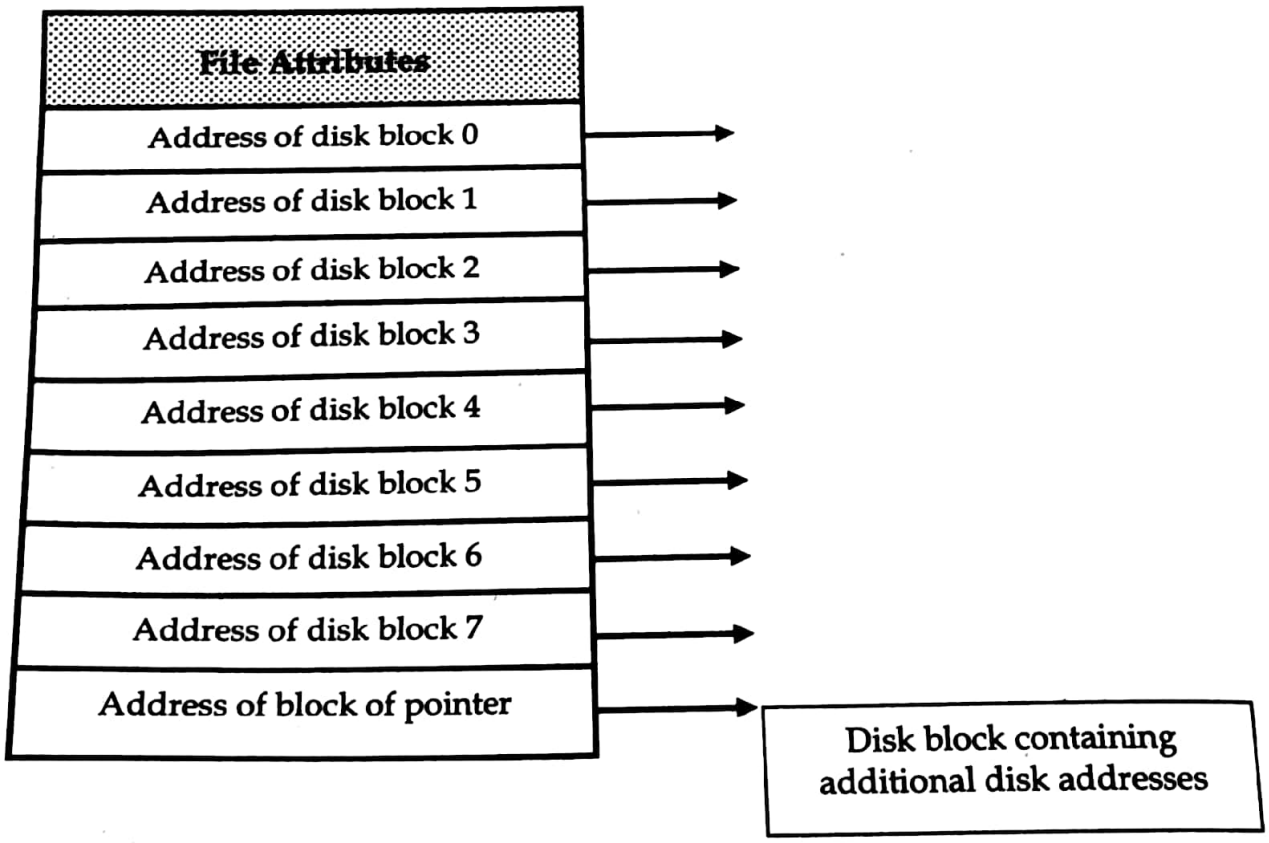
only be in memory when the corresponding file is open unlike file allocation table which grows linearly with the disk.

Advantage

- Space needed bin memory is directly proportional to number of files opened not the size of disk. If each i-node occupies n bytes and a maximum of k files may be open at once, the total memory occupied by the i-nodes for the open files is only kn bytes.

Disadvantage

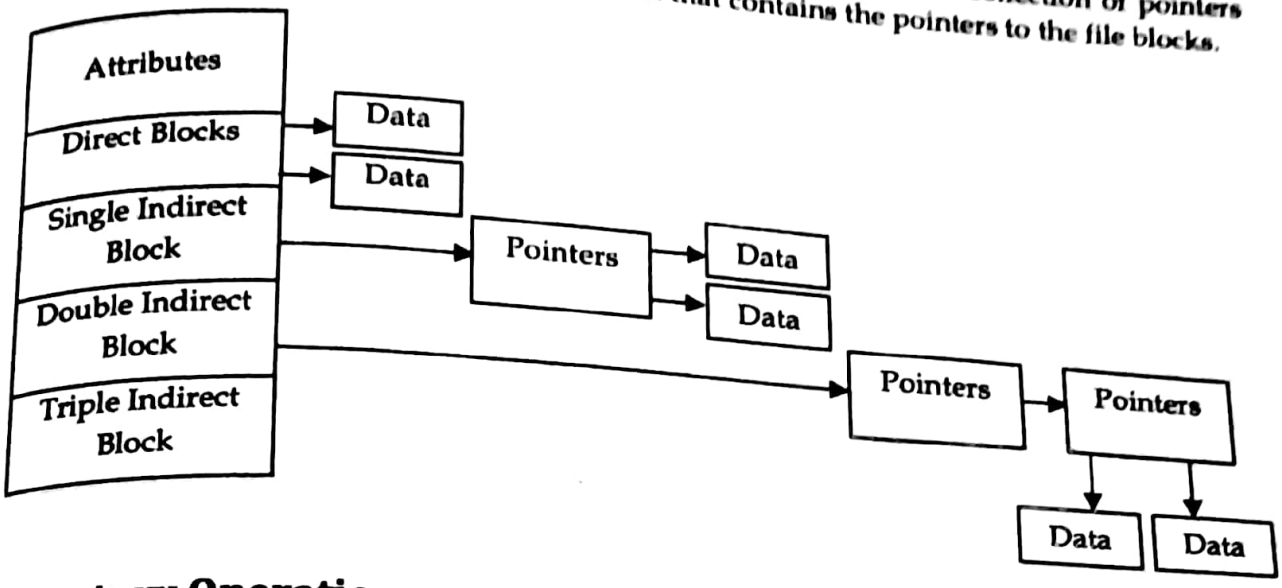
- I-nodes have room for a fixed number of disk addresses, what happens when a file grows beyond this limit? One solution is to reserve the last disk address not for a data block, but instead for the address of a address of a block containing more disks block addresses.



In UNIX based operating systems, each file is indexed by an I-node. I-node is the special disk block which is created with the creation of the file system. The number of files or directories in a file system depends on the number of I-nodes in the file system. An I-node includes the following information

- Attributes (permissions, time stamp, ownership details, etc.) of the file
- A single indirect pointer which points to an index block. If the file cannot be indexed entirely by the direct blocks then the single indirect pointer is used.
- A double indirect pointer which points to a disk block that is a collection of the pointers to the disk blocks which are index blocks. Double index pointer is used if the file is too big to be indexed entirely by the direct blocks as well as the single indirect pointer.

- A triple index pointer that points to a disk block that is a collection of pointers. Each of the pointers is separately pointing to a disk block which also contains a collection of pointers which are separately pointing to an index block that contains the pointers to the file blocks.



13. Directory Operations

The allowed system calls for managing directories exhibit more variation from system to system then system calls for files. The following are the some of the key directory operations:

1. **Create:** A directory is created. It is empty except for dot and dot dot, which are put there automatically by the system
2. **Delete:** A directory is deleted. Only an empty directory can be deleted.
3. **Opendir:** Directory can be read. For example, to list all files in a directory, a listing program opens the directory to read out the names of all the files it contains.
4. **Closedir:** When a directory has been read, it should be closed to free up internal table space.
5. **Readdir:** This call returns the next entry in an open directory. Formally, it was possible to read directories using the usable read system call, but that approach has the disadvantage for forcing the programmer to know and deal with the internal structure of the directories.
6. **Rename:** In many respects, directories are just like files and can be renamed the same way file can be.
7. **Link:** Linking is the technique that allows a file to appear in more than one directory.
8. **Unlink:** A directory entry is removed. If the file being unlinked is only present in one directory, it is removed from the file system.

14. Path Names

When the file system is organized as a directory tree, some way is needed for specifying file names. Two different methods are commonly used. These are **absolute** and **relative** path names.

1. **Absolute Path Name**

   The path name starting from root directory to the file. E.g. In UNIX: /usr/user1/bin/lab2.Path separated by / in UNIX and \ in windows.

2. **Relative Path Name**

Concept of working directory (also called working directory).

A user can designate one directory as the current working directory, in which case all path names not beginning at the root directory are taken relative to the working directory. E.g., bin/lab2 is enough to locate same file if current working directory is /usr/user1.

15. Directory Implementation

The selection of directory-allocation and directory-management algorithms significantly affects the efficiency, performance, and reliability of the file system. We implement the directory by using different two methods.

15.1 Linear list

The simplest method of implementing a directory is to use a linear list of file names with pointers to the data blocks. This method is simple to program but time-consuming to execute. To create a new file, we must first search the directory to be sure that no existing file has the same name. Then, we add a new entry at the end of the directory. To delete a file, we search the directory for the named file, then release the space allocated to it. The real disadvantage of a linear list of directory entries is that finding a file requires a linear search. Directory information is used frequently, and users will notice if access to it is slow. In fact, many operating systems implement a software cache to store the most recently used directory information. A cache hit avoids the need to constantly reread the information from disk. A sorted list allows a binary search and decreases the average search time.

Characteristics

- When a new file is created, then the entire list is checked whether the new file name is matching to an existing file name or not. In case, it doesn't exist, the file can be created at the beginning or at the end. Therefore, searching for a unique name is a big concern because traversing the whole list takes time.
- The list needs to be traversed in case of every operation (creation, deletion, updating, etc) on the files therefore the systems become inefficient.

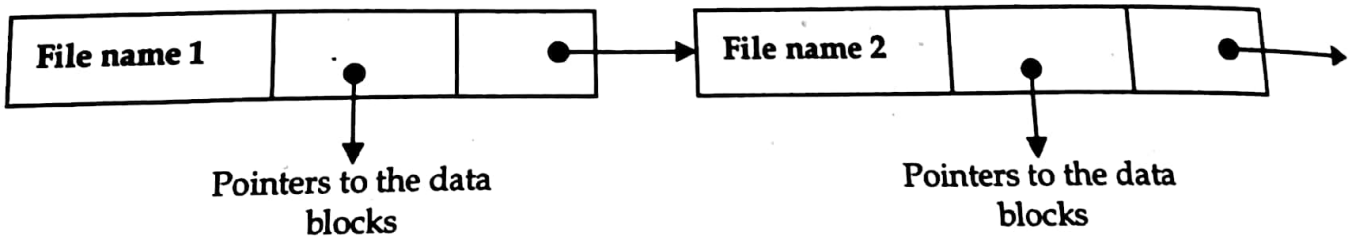
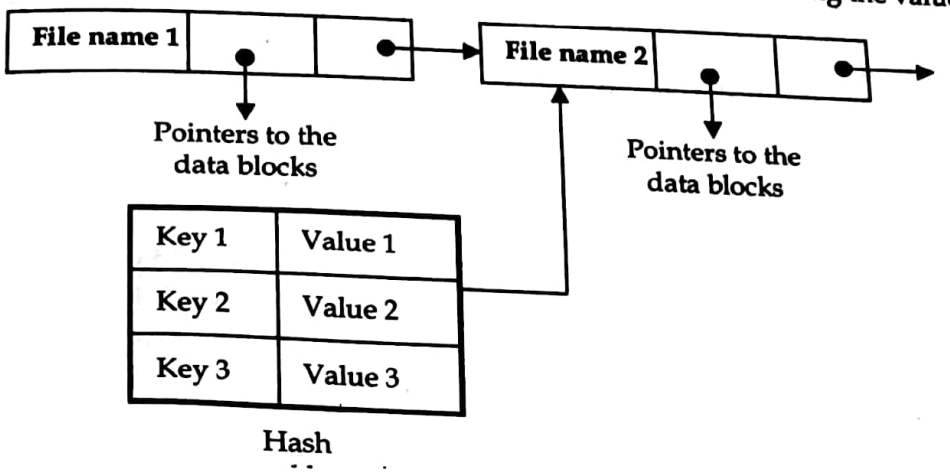


Fig 5.6: Linear List

15.2 Hash Table

Another data structure that has been used for a file directory is a hash table. It consist linear list with hash table. The hash table takes a value computed from the file name and returns a pointer to the file name in the linear list. Therefore, it can greatly decrease the directory search time. Insertion and deletion are also fairly straightforward, although some provision must be made for collisions-situations where two file names hash to the same location. If that key is already in use, a linked list is constructed.

A key-value pair for each file in the directory gets generated and stored in the hash table. The key can be determined by applying the hash function on the file name while the key points to the corresponding file stored in the directory. Now, searching becomes efficient due to the fact that now, entire list will not be searched on every operating. Only hash table entries are checked using the key and if an entry found then the corresponding file will be fetched using the value.



Advantages: greatly decrease the file search time.

Problem: It greatly fixed size and dependence of the hash function on that size.

16. Shared Files

In many situations we need to share files between users. For example, when several users are working together on a project, they often need to share files. It is often convenient for a shared file to appear simultaneously in different directories belonging to different users. Thus, it is better to represent file system by using directed acyclic graph (DAG) rather than tree structure as shown in figure below;

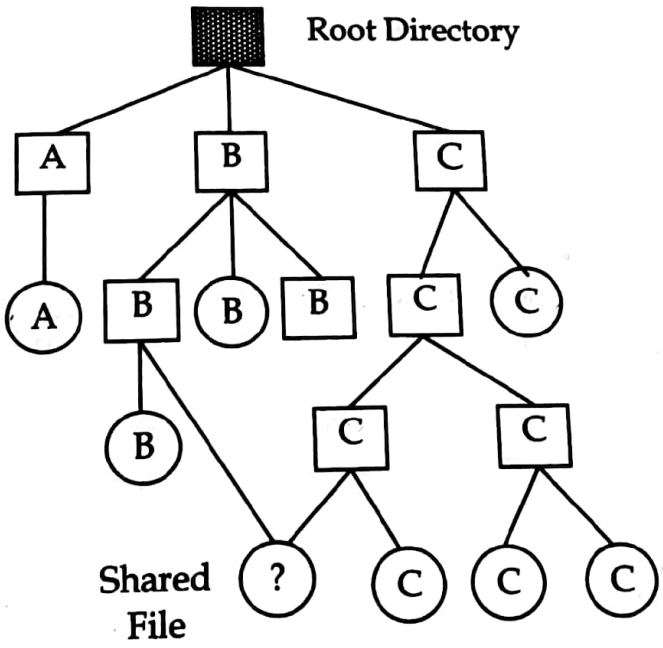


Fig 5.10: File system representation using DAG

It is not a good idea to make a copy if a file is being shared. If directories contain disk addresses problem may occur in synchronizing the change made to the file by multiple users. If one user

appends new block in the file new block will be listed only in the directory of the user doing the append. Following two approaches can be used to solve this problem.

- Directory entry that only points to i-nodes
- Directory entry that points to link file

In the first approach, disk blocks are not listed in directories rather directory entry point to the little data structure (i-node in UNIX) associated with the file itself. Creating a link does not change the ownership, but it does increase the link count in the i-node. Main problem with this approach is that if owner of the shared file tries to remove the file, the system is faced with a problem. If it removes the file and clears the i-node, another user may have a directory entry pointing to an invalid i-node. One solution to this is to only remove owner's directory entry. But again if the system has quotas, owner will continue to be billed for the file until another user decides a remove it.

In second approach, when another user B want to share the file owned by used C then system creates a link file and then maker entry of the link file in B's directory. Link file contains just the path name of the file to which it is linked. When B reads from the linked file, the operating system sees that the file being read from is of type link, looks up the name of the file, and reads that file. This approach is called symbolic linking. Extra overhead is the main problem associated with this approach. The file containing the path must be read, and then the path must be parsed and followed, component by component, until the I-node is reached. All of this activity may require a considerable number of extra disk accesses.

17. Free Space Management

Since disk space is limited, we need to reuse the space from deleted files for new files, if possible. To keep track of free disk space, the system maintains a free space list. The free space list records all free disk blocks those not allocated to some file or directory. A file system is responsible to allocate the free blocks to the file therefore it has to keep track of all the free blocks present in the disk. Just as the space that is allocated to files must be managed, so the space that is not currently allocated to any file must be managed. To perform any of the file allocation techniques, it is necessary to know what blocks on the disk are available. Thus we need a disk allocation table in addition to a file allocation table. There are mainly two approaches by using which, the free blocks in the disk are managed.

- Bit Vector and
- Linked List

17.1 Bitmaps Free Space Management

In this approach, the free space list is implemented as a bit map vector. It contains the number of bits where each bit represents each block. If the block is empty then the bit is 1 otherwise it is 0. Initially all the blocks are empty therefore each bit in the bit map vector contains 1. A disk with n blocks requires a bitmap with n bits.

Example: Consider a disk where blocks 2,3,4,5,8,9,10,11,12,13,17,18,..... are free, and rest are allocated. The free space bit map would be

0011110011111100010.....

To find the first free block, the Macintosh operating system checks sequentially each word in the bit map to see whether that value is not 0, since a 0 valued word has all 0 bits and represents a set of allocated blocks. The first non 0 word is scanned for the first 1 bit, which is the location of the first free block. The calculation of the block number is

(Number of bits per word) × (number of 0 value words) + offset of first 1 bit.

Example: Let's take block list {2, 3, 4, 5, 9, 10, 13}

Solution

Blocks →	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Bits →	0	0	1	1	1	1	0	0	0	1	1	0	0	1

Advantages: Simple and efficient in finding first free block, or n consecutive free blocks.

Problems: Inefficient unless the entire bitmap is kept in main memory. Keeping in main memory is possible only for small disk, when disk is large the bitmap would be large. Many system use bitmap method

17.2 Linked List Free Space Management

It is another approach for free space management. This approach suggests linking together all the free blocks and keeping a pointer in the cache which points to the first free block.Therefore, all the free blocks on the disks will be linked together with a pointer. Whenever a block gets allocated, its previous free block will be linked to its next free block. We would keep a pointer to block 2, as the first free block. Block 2 would contain a pointer to block. 3, which would point to block 4, which would point to block 5, which would point to block 8, However, this scheme is not efficient; to traverse the list, we must read each block, which requires substantial 1/0 time. Fortunately, traversing the free list is not a frequent action. Usually, the operating system simply needs a free block so that it can allocate that block to a file, so the first block in the free list is used. The FAT method incorporates free-block accounting into the allocation data structure. No separate method is needed.

Advantages: Only one block is kept in memory.

Problems: Not efficient; to traverse list, it must read each block. Usually, the operating system simply needs a free block so that it can allocate that block to a file, so the first block in the free list is used. The FAT method incorporates free block accounting into the allocation data structure. No separate method is needed.

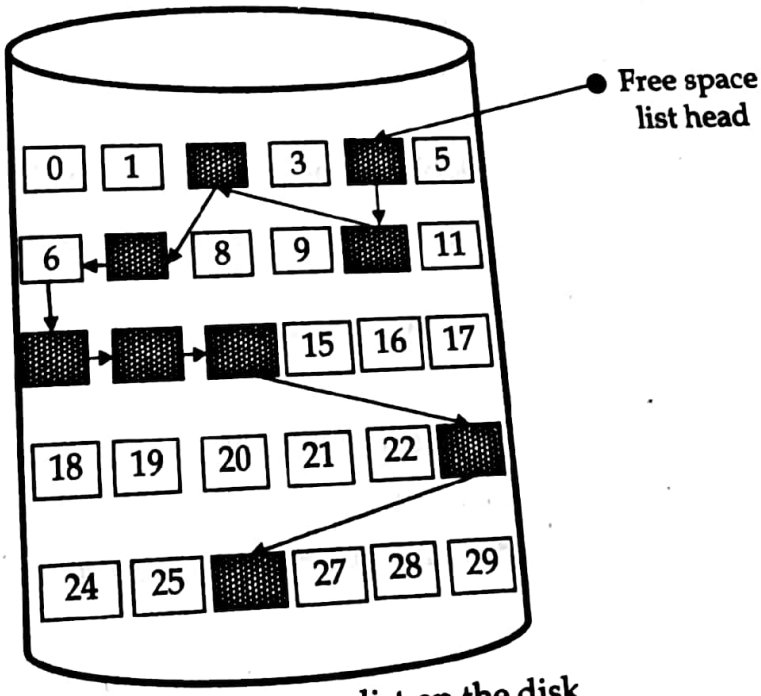


Fig 5.11: Linked free space list on the disk

Problem 1: Consider a 40 MB disc with 2 K blocks. Calculate the number of blocks needed to hold the disc bitmap. If we require 16-bit to hold a disc block number (i.e. disk block number range from 0-65535). What will be number of blocks needed by linked list of free blocks?

Answer: Case for Bitmap

Disk size = 40 MB = 40 x 1024 KB

⇨ Number of blocks= (40 x 1024)/2= 20 x 1024

Since every block needs 1- bit in bitmap

⇨ Size of Bitmap=20 x 1024 bit = 2560 byte = 2.5 KB

Since block size is 2 KB

⇨ 2 blocks are used to hold the bitmap

For Case of Free list

Block size = 2K= 2 x 1024 byte

Number of bits needed to store block number = 16 bit = 2 byte

Thus, number of blocks that can be stored in a block = (2 x 1024)/2= 1024

Since one of the addresses is used to store address of the next block in the free list

⇨ Number of blocks that can be stored in a block = 1024-1= 1023

We know that,

Size of disk=40 MB = 40 x 1024 KB

Since, size of block= 2 K

⇨ Number of blocks in disc= (40 x 1024)/2 = 20 x 1024

Thus, number of blocks needed to store free list= (20 x 1024)/1023 = 20.019 = 20

Problem 2: Assume you have an i-node-based file system. The file system has 512 byte blocks. Each i-node has 10 direct, 1 single indirect, 1 double indirect and 1 triple indirect block pointers. Block pointers are 4 bytes each. Assume the i-node and any block free list is always in memory. Blocks are not cached.

(i) What is the maximum file size that can be stored before...
   a) The single indirect pointer is needed?
   b) The double indirect pointer is needed?
   c) The triple indirect pointer is needed?

(ii) What is the maximum file size supported?

(iii) What is the number of disk block reads required to read 1 byte from a file
   a) In the best case?
   b) In the worst case?

Answer:

i) a) 10×512 bytes = 5 KB

   b) $10 \times 512 + (512 / 4) \times 512$ bytes = 69 KB

   c) $10 \times 512 + (512 / 4) \times 512 + (512 / 4)^2 \times 512$ bytes ≈ 8 MB

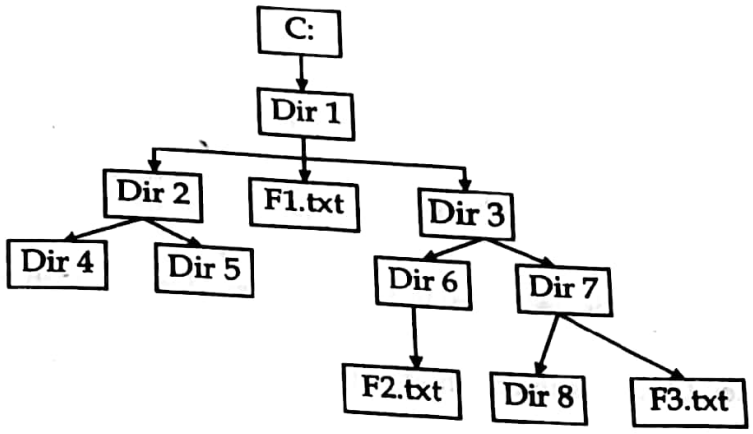
ii) $10 \times 512 + (512 / 4) \times 512 + (512 / 4)\ 2 \times 512 + (512 / 4)\ 3 \times 512$ bytes ≈ 1 GB

iii) a) 1

   b) 4

Laboratory Works

1. Create the files and directories as following structure:



Copy the F3.txt to dir 9 and F1.txt to dir 8

2. Rename the file c:\dir1\dir3\dir6\F2.txt to F.txt and c:\dir1\dir3\dir7\F3.txt to F3.txt
3. Delete the file F1.txt
4. Delete the directory dir 9

Exercise

1. What is a file? Write down their importance in OS.
2. What are the typical operations performed on files?
3. What are File Control Blocks? Explain their structure with suitable example.
4. What are file types? Explain types of file used in OS.
5. How is free space managed? Explain free space management techniques in detail.
6. Consider a file system where a file can be deleted and its disk space Reclaimed while links tothat file still exist. What problems may occur if a new file is created in the same storage area orwith the same absolute path name? How can these problems be avoided?
7. What are the advantages and disadvantages of a system providing mandatory locks instead ofproviding advisory locks whose usage is left to the users' discretion?
8. What are the advantages and disadvantages of recording the name of the creating programwith the file's attributes (as is done in the Macintosh Operating System)?
9. If the operating system were to know that a certain application is going to access the file data ina sequential manner, how could it exploit this information to improve performance?
10. Give an example of an application that could benefit from operating system support for randomaccess to indexed files.
11. Discuss the merits and demerits of supporting links to files that cross mount points (that is thefile link refers to a file that is stored in a different volume).
12. Discuss the advantages and disadvantages of associating with remote file systems (stored onfile servers) a different set of failure semantics from that associated with local file systems
13. Give an example of a Directory implementation that could benefit from operating system.
14. What is Hierarchical Directory Systems? Explain with suitable example.

15. What does Contiguous Allocation requires? Write down their advantages and disadvantages.
16. Suppose tree-directory structure and acyclic-graph directory structure is implemented in two different file systems to organize files and directories. Which one would be better and why?
17. What is virtual file system (VFS)? How multiple file systems are handled by virtual file system?
18. What is indexed-allocation method? Is multilevel indexed allocation is a better solution for applications that need files with very large size? Explain your answer.
19. Why access protection is necessary in file systems? How protection can be implemented by the identity of the users?
20. Compare linked file allocation method with indexed allocation method

❑❑❑

Chapter 6
DEVICE MANAGEMENT

1. Introduction

All computers have physical devices for acquiring input and producing output. OS is responsible to manage and control all the I/O operations and I/O devices. Device management generally performs the following:

- Installing device and component-level drivers and related software
- Configuring a device so it performs as expected using the bundled operating system, business/workflow software and with other hardware devices.
- Implementing security measures and processes.

Devices usually refer to physical devices such as computers, laptops, servers, mobile phones and more. They could also be virtual, however such as virtual machines or virtual switches. In Windows, device management is also an administrative module that is used for managing or configuring the physical devices, ports and interfaces of a computer or server.

One of the important jobs of an operating system is to manage various I/O devices including mouse, keyboards, touch pad, disk drives, display adapters, USB devices, bit-mapped screen, LED, analog-to-digital converter, on/off switch, network connections, audio I/O, printers etc. An I/O system is required to take an application I/O request and send it to the physical device, then take whatever response comes back from the device and send it to the application.

2. Classification of I/O devices

I/O devices can be divided into following three categories

- Machine readable or Block devices
- User readable or Character devices
- Communications devices

2.1 Block devices

A block device is one with which the driver communicates by sending entire blocks of data. Here information in fixed blocks with each block has its own address. Each block can be read from/written to independently. For example, Hard disks, USB cameras, Disk-On-Key, tapes, sensors etc.

A block device generally requires fewer pins and can thus be placed in a smaller package than a word-addressed device. In addition, a block device interface to slower forms of memory, such as rotating media and NAND flash, is much easier to build than a word-addressed interface, and these slower forms of memory generally have a much larger capacity dollar-for-dollar than NOR flash.

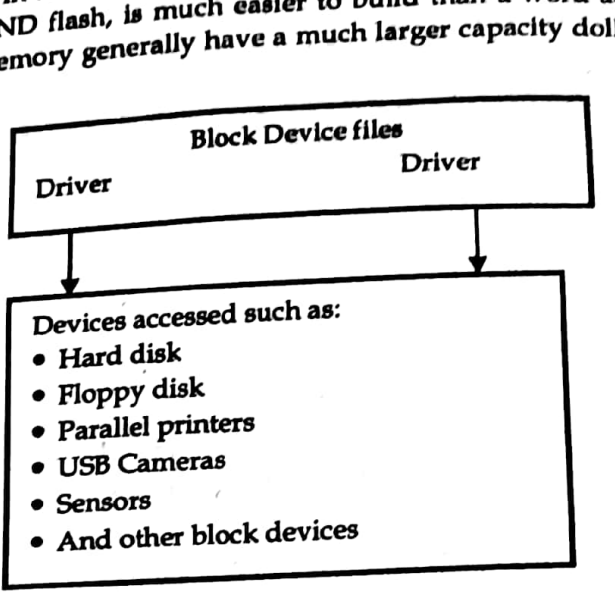


Fig 6.1: Block Device

2.2 Character devices

A character device is one with which the driver communicates by sending and receiving single characters (bytes, octets). It accepts a stream of characters with no attention paid to block structure. It is not addressable and has no seek ability. For example, printers, graphical terminals, screen, keyboard, mouse, serial ports, parallel ports, sounds cards etc. Character devices are devices that do not have physically addressable storage media, such as tape drives or serial ports, where I/O is normally performed in a byte stream.

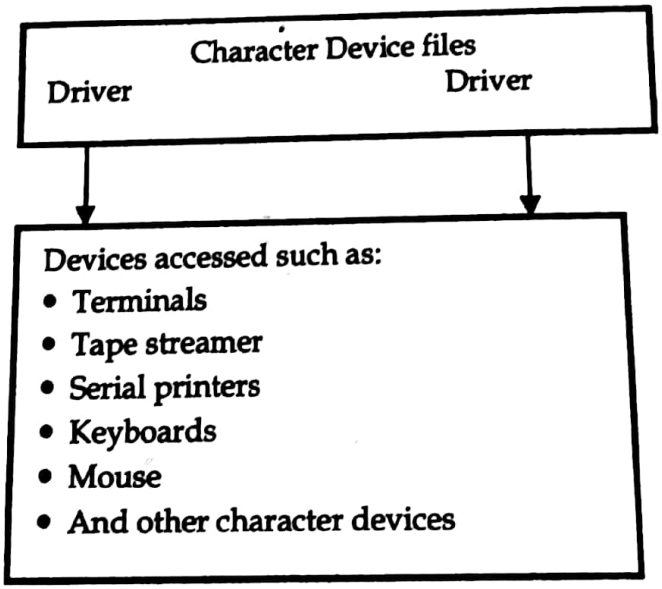


Fig 6.2: Character Device

2.3 Communication devices

A communication device is a hardware device capable of transmitting an analog or digital signal over the telephone, other communication wire, or wirelessly. The best example of a

communication device is a computer Modem, which is capable of sending and receiving a signal to allow computers to talk to other computers over the telephone. Other examples of communication devices include a NIC (network interface card), Wi-Fi devices, and access points. For a computer to communicate with other computers they need a communication device. For example, for your computer to connect to the Internet to view this web page it needed a communication device. Without a communication device you'd have to use a net to transfer or share data between computers.

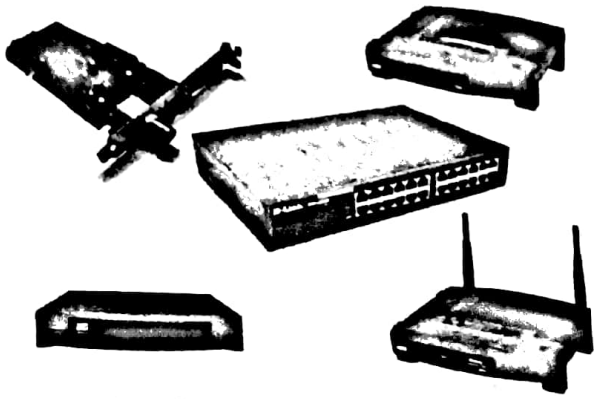


Fig 6.3: Communication devices

3. Device Controllers

A controller is a collection of electronics that can operate a bus or a device. On PC, it often takes the form of printed circuit card that can be inserted into an expansion slot. A single controller can handle multiple devices; some devices have their own built-in controller. The controller has one or more registers for data and signals. The processor communicates with the controller by reading and writing bit patterns in these registers.

When transferring a disk block of size 512 bytes, the block first assembled bit by bit in a buffer inside the controller. After its checksum has been verified and the block declared to be error free, it can then be copied to main memory.

Device drivers are software modules that can be plugged into an OS to handle a particular device. Operating System takes help from device drivers to handle all I/O devices. The Device Controller works like an interface between a device and a device driver. I/O units typically consist of a mechanical component and an electronic component where electronic component is called the device controller. There is always a device controller and a device driver for each device to communicate with the Operating Systems. A device controller may be able to handle multiple devices. As an interface its main task is to convert serial bit stream to block of bytes, perform error correction as necessary. Following is a model for connecting the CPU, memory, controllers, and I/O devices where CPU and device controllers all use a common bus for communication.

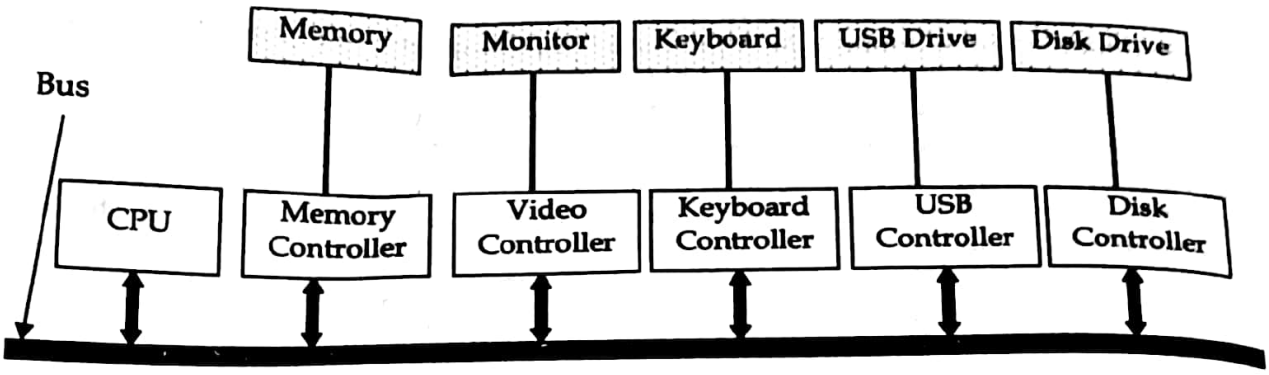


Fig 6.4: Device Controllers System

Difference between device driver and device controller

The main difference between device driver and device controller is that the device driver is software that works as the interface for the device controller to communicate with the operating system or an application program. Whereas the device controller is a hardware component that works as a bridge between the hardware device and the operating system or an application program. A device driver is specific to an operating system and it is hardware dependent. It provides interrupt handling required for necessary asynchronous time-dependent hardware interface. On the other hand, device controller is a circuit board between the device and the operating system.

4. Communication of CPU to I/O Devices

The CPU must have a way to pass information to and from an I/O device. There are three approaches available to communicate with the CPU and Device.

1. Special Instruction I/O
2. Memory-mapped I/O
3. Direct memory access (DMA)

4.1 Special Instruction I/O

This uses CPU instructions that are specifically made for controlling I/O devices. These instructions typically allow data to be sent to an I/O device or read from an I/O device.

4.2 Memory Mapped I/O

Device controller has their own register and buffer for communicating with the CPU, by writing and reading these register OS perform the I/O operation. The device control registers are mapped into memory space, called memory-mapped I/O. Usually, the assigned addresses are at the top of the address space.

While using memory mapped IO, OS allocates buffer in memory and informs I/O device to use that buffer to send data to the CPU. I/O device operates asynchronously with CPU, interrupts CPU when finished. The advantage to this method is that every instruction which can access memory can be used to manipulate an I/O device. Memory mapped IO is used for most high-speed I/O devices like disks, communication interfaces.CPU communicates with the control registers and the device data buffers in following three ways:

- By using I/O Port system
- By using memory mapped I/O system
- By using hybrid system

4.2.1 Using I/O Port

It is also called isolated I/O system. In this approach each control register is assigned an I/O port number of an 8 or 16-bit integer. CPU reads and writes control registers by using special I/O instructions IN and OUT as shown in below. Access to them requires the use of assembly code since there is no way to execute an IN and OUT instruction in programming languages like C, C++ etc.

```
IN REG, PORT
OUT PORT, REG
```

The first instruction is used to read data from specified port or control registers and second instruction is used to write data to specified port or control register.

The address space for memory and I/O are different therefore separate instructions are needed to read/write memory and I/O control registers.

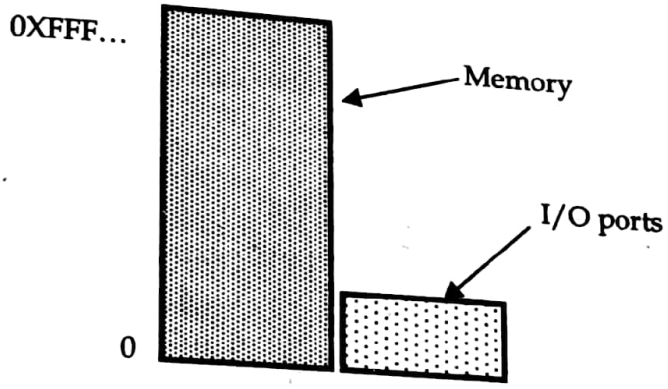


Fig 6.5: Two separate memory for isolated I/O system.

4.2.2 Memory mapped I/O

In this approach all control registers are mapped into the memory space. Each control register is assigned a unique address to which no memory is assigned. No special protection mechanism is needed to keep user processes from performing I/O. Here the address space for memory and I/O is same therefore instructions needed to read/write memory can be used to read/write I/O control registers.

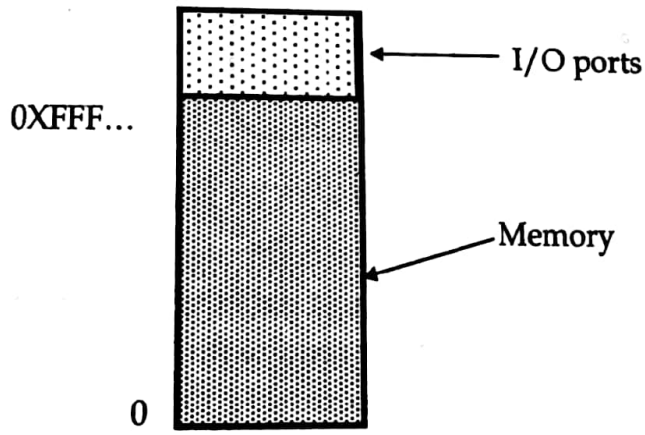


Fig 6.6: Memory mapped I/O system

4.2.3 By using Hybrid System

It is the combined form of both memory mapped I/O and I/O port system. It also uses two types of memory separately.

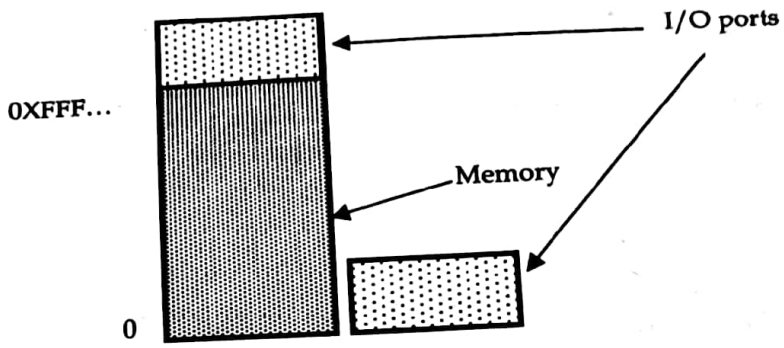


Fig 6.7: Hybrid I/O system

Advantages of memory mapped I/O

- Can be implemented in high-level languages such as C.
- No separate protection mechanism is needed.
- Every instruction that can reference the memory can also reference the control registers (in hybrid).

Disadvantages memory mapped I/O

- Adds extra complexity to both hardware and OS.
- All memory modules and all I/O devices must examine all memory references to see which one to respond to.

4.3 Direct memory access (DMA)

Slow devices like keyboards will generate an interrupt to the main CPU after each byte is transferred. If a fast device such as a disk generated an interrupt for each byte, the operating system would spend most of its time handling these interrupts. So a typical computer uses direct memory access (DMA) hardware to reduce this overhead.

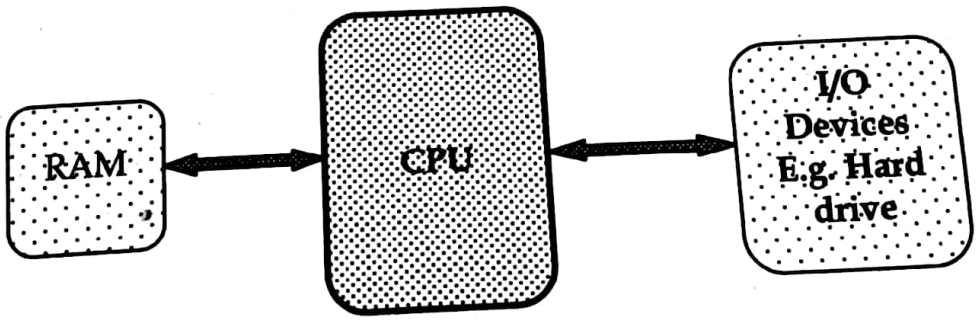


Fig 6.8: Communicate CPU with I/O devices without DMA.

DMA is a method of transferring data from the computer's RAM to another part of the computer without processing it using the CPU. While most data that is input or output from your computer is processed by the CPU, some data does not require processing, or can be processed by another device. In these situations, DMA can save processing time and is a more efficient way to move data from the computer's memory to other devices. In order for devices to use direct memory access, they must be assigned to a DMA channel. Each type of port on a computer has a set of

DMA channels that can be assigned to each connected device. For example, a PCI controller and a hard drive controller each have their own set of DMA channels.

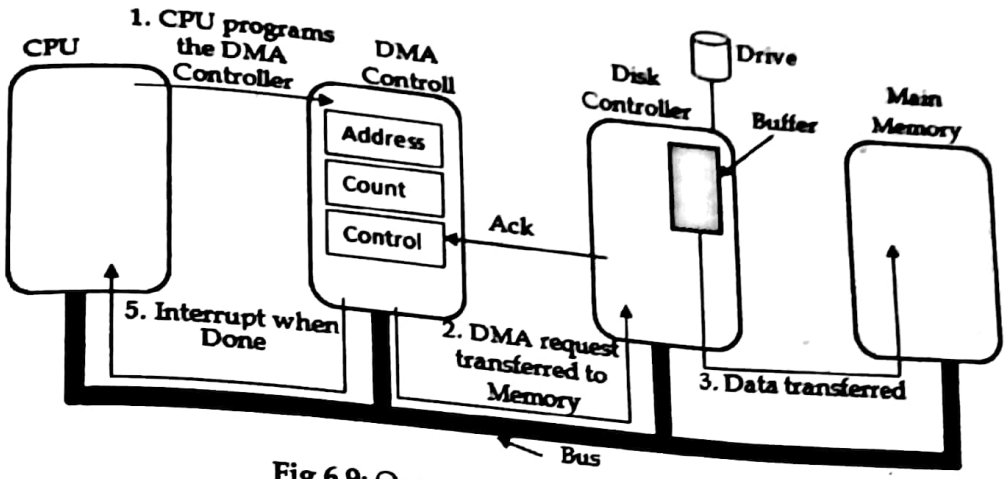


Fig 6.9: Operations of a DMA transfer

Direct Memory Access (DMA) means CPU grants I/O module authority to read from or write to memory without involvement. DMA module itself controls exchange of data between main memory and the I/O device. CPU is only involved at the beginning and end of the transfer and interrupted only after entire block has been transferred. Direct Memory Access needs a special hardware called DMA controller (DMAC) that manages the data transfers and arbitrates access to the system bus. The controllers are programmed with source and destination pointers (where to read/write the data), counters to track the number of transferred bytes, and settings, which includes I/O and memory types, interrupts and states for the CPU cycles.

Working procedure of DMA

- The CPU programs the DMA controller by its registers so it knows what to transfer where. It also issues a command to disk controller telling it to read data from disk to its internal buffer and verify the checksum. When valid data are in disk's controller buffer, DMA can begin.
- The DMA controller initiates the transfer by issuing a read request over the bus to disk controller.
- Data transferred from disk controller to memory.
- When transferred completed, the disk controller sends an acknowledgment signal to DMA controller. The DMA controller then increments the memory address to use and decrement the byte count. This continues until the byte count greater than 0.
- When transfer completed the DMA controller interrupt the CPU.

5. Interrupts

The hardware mechanism that enables a device to notify the CPU is called an interrupt. Interrupt forced to stop CPU what it is doing and start doing something else. Interrupts are signals sent to the CPU by external devices, normally I/O devices. They tell the CPU to stop its current activities and execute the appropriate part of the operating system.

There are three types of interrupts

- **Hardware Interrupts** are generated by hardware devices to signal that they need some attention from the OS. They may have just received some data or they have just completed a

task which the operating system previous requested, such as transferring data between the hard drive and memory.

- **Software Interrupts** are generated by programs when they want to request a system call to be performed by the operating system.
- **Traps** are generated by the CPU itself to indicate that some error or condition occurred for which assistance from the operating system is needed.

An interrupt is a signal to the microprocessor from a device that requires attention. A device controller puts an interrupt signal on the bus when it needs CPU's attention when CPU receives an interrupt; it saves its current state and invokes the appropriate interrupt handler using the interrupt vector. When the interrupting device has been dealt with, the CPU continues with its original task as if it had never been interrupted.

Interrupts are important because they give the user better control over the computer. Without interrupts, a user may have to wait for a given application to have a higher priority over the CPU to be run. This ensures that the CPU will deal with the process immediately.

Working Mechanism of Interrupts

When I/O device has finished the work given to it, it causes an interrupt. It does this by asserting a signal on a bus, that it has been assigned. The signal detected by the interrupt controller then decides what to do? If no other interrupts are pending, the interrupt controller processes the interrupts immediately, if another is in progress, then it is ignored for a moment.

To handle the interrupt, the controller puts a number of address lines specifying which device wants attention and asserts a signal that interrupt the CPU. The number of address lines is used as index to a table called interrupt vector to fetch a new program counter, this program counter points to the start of corresponding service procedure.

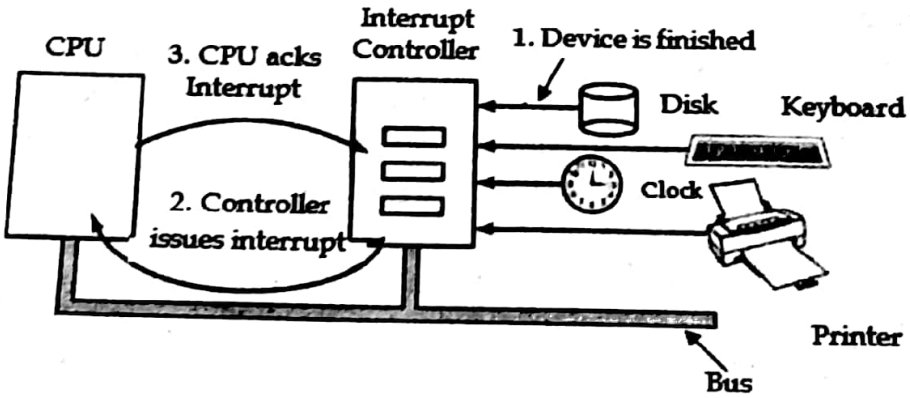


Fig 6.10: Interrupt handler

6. Goals of I/O Software

There are many goals of I/O software. Some of major principles of I/O software are listed below:

a. Device independence
b. Uniform naming
c. Error Handling
d. Synchronous vs. asynchronous transfers
e. Buffering
f. Sharable and Dedicated devices

a. **Device independence**

A key concept in the design of I/O software is known as device independence. It means that I/O devices should be accessible to programs without specifying the device in advance. For example, a program that reads a file as input should be able to read a file on a file on a floppy disk, on a hard disk, or on a CD-ROM, without having to modify the program for each different device.

b. **Uniform Naming**

Uniform Naming, simply be a string or an integer and not depend on the device in any way. In UNIX, all disks can be integrated in the file-system hierarchy in arbitrary ways so the user need not be aware of which name corresponds to which device. Here the name of file or device should be some specific string or number. It must not depend upon device in any way. All files and devices are addressed the same way: by a path name.

c. **Error handling**

Generally, errors should be handled as close as possible to the computer hardware. It should try to correct the error itself if it can in case if the controller discovers a read error. And in case if it can't then the device driver should handle it. If the controller discovers a read error, it should try to correct the error itself if it can. If it cannot, then the device driver should handle it, perhaps by just trying to read the block again. In many cases, error recovery can be done transparently at a low level without the upper levels even knowing about the error.

d. **Synchronous (blocking) and Asynchronous (interrupt-driven) transfers**

Most physical input/output is asynchronous; however, some very high performance applications need to control all the details of the I/O, so some operating systems make asynchronous I/O available to them. The central processing unit starts the transfer and goes off to do something other until the interrupt arrives. In case if input/output operations are blocking the user programs are much easier to write. After a read system call the program is automatically suspended until the data are available in buffer. Basically, it is up to the OS to make the operation that are really asynchronous look blocking to the user programs.

e. **Buffering**

Sometime data that come off a device can't be stored directly in its final destination. Buffering sometime has a major impact on the systems input/output performance because it involves considerable copying.

Data comes in main memory cannot be stored directly. For example data packets come from the network cannot be directly stored in physical memory. Packet to be put into output buffer for examining them. Some devices have several real-time constraints, so data must be put into output buffer in advance to decouple the rate at which buffer is filled and the rate at which it is emptied, in order to avoid buffer under runs.

f. **Sharable and Dedicated devices**

Some I/O devices, such as disks, can be used by many users at the same time. No problems are caused by multiple users having open files on the same disk at the same time. Other devices, such as printers, have to be dedicated to a single user until that user is finished.

Then another user can have the printer. Introducing dedicated (unshared) devices also introduces a variety of problems, such as deadlocks. Again, the operating system must be able to handle both shared and dedicated devices in a way that avoids problems.

7. Handling I/O

There are three fundamentally different ways of performing I/O operations which are listed below:

a. Programmed I/O
b. Interrupt Driven I/O and
c. I/O using DMA

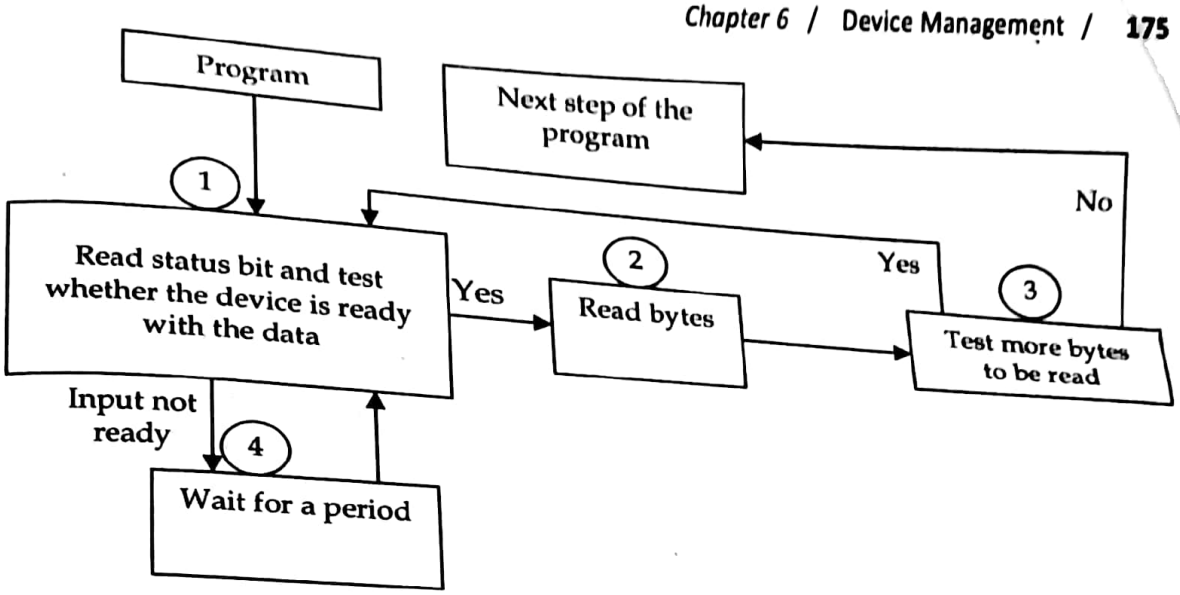
a. Programmed I/O

This is one of the three fundamentally different ways that I/O can be performed. The programmed I/O was the simplest type of I/O technique for the exchanges of data or any types of communication between the processor and the external devices. With programmed I/O, data are exchanged between the processor and the I/O module. The processor executes a program that gives it direct control of the I/O operation, including sensing device status, sending a read or write command, and transferring the data. When the processor issues a command to the I/O module, it must wait until the I/O operation is complete. If the processor is faster than the I/O module, this is wasteful of processor time. The overall operation of the programmed I/O can be summaries as follow:

- The processor is executing a program and encounters an instruction relating to I/O operation.
- The processor then executes that instruction by issuing a command to the appropriate I/O module.
- The I/O module will perform the requested action based on the I/O command issued by the processor and set the appropriate bits in the I/O status register.
- The processor will periodically check the status of the I/O module until it finds that the operation is complete.

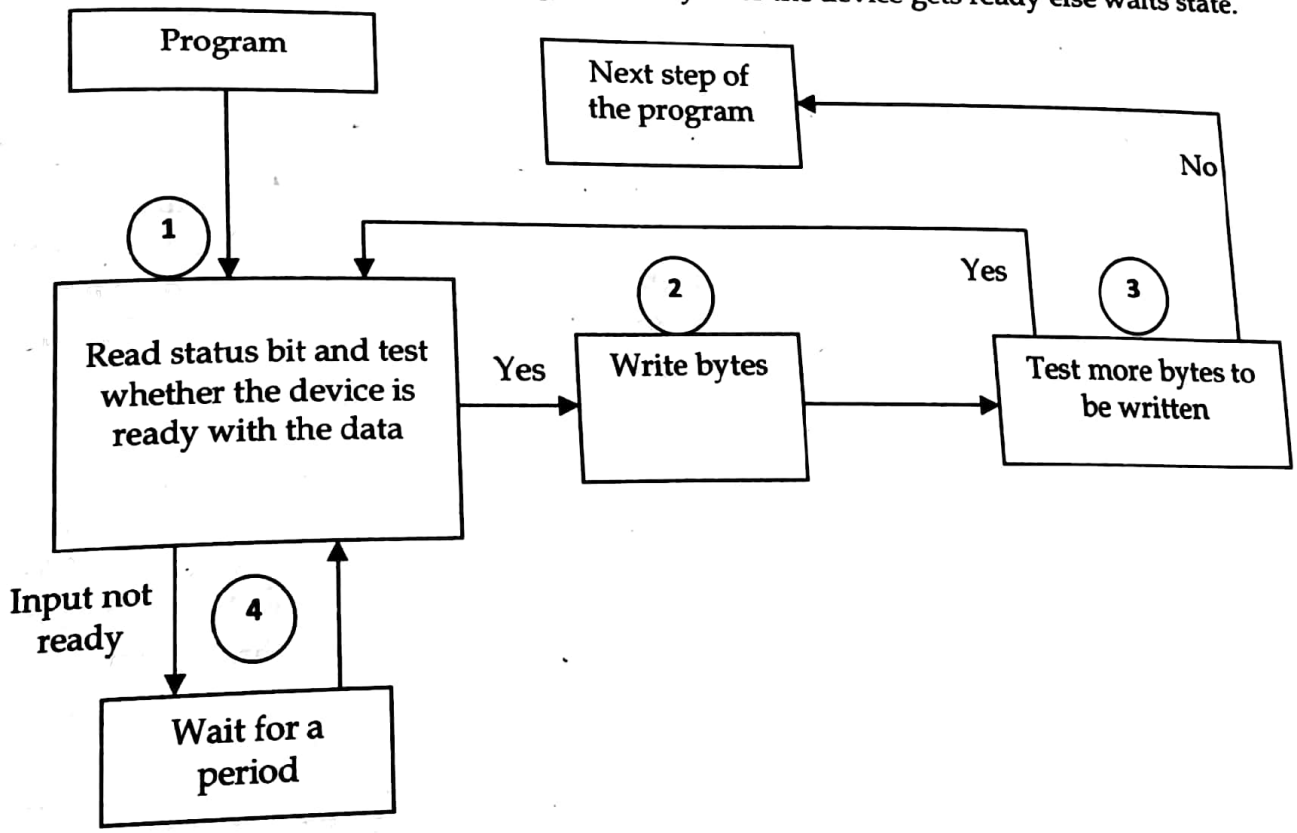
Programmed I/O Mode: Input Data Transfer

- Each input is read after first testing whether the device is ready with the input (a state reflected by a bit in a status register).
- The program waits for the ready status by repeatedly testing the status bit and till all targeted bytes is read from the input device.
- The program is in busy (non-waiting) state only after the device gets ready else in wait state.



Programmed I/O Mode: Output Data Transfer

- Each output written after first testing whether the device is ready to accept the byte at its output register or output buffer is empty.
- The program waits for the ready status by repeatedly testing the status bit(s) and till all the targeted bytes are written to the device.
- The program in busy (non-waiting) state only after the device gets ready else waits state.



Advantages of Programmed I/O

- Simple to implement
- Very little hardware support

Disadvantages of Programmed I/O

- Busy waiting
- Ties up CPU for long period with no useful work

b. Interrupt-Driven I/O

Interrupt driven I/O is an alternative scheme dealing with I/O. Interrupt I/O is a way of controlling input/output activity whereby a peripheral or terminal that needs to make or receive a data transfer sends a signal. This will cause a program interrupt to be set. At a time appropriate to the priority level of the I/O interrupts. Relative to the total interrupt system, the processors enter an interrupt service routine.

This strategy allows the CPU to carry on with its other operations until the module is ready to transfer data. When the CPU wants to communicate with a device, it issues an instruction to the appropriate I/O module, and then continues with other operations. When the device is ready, it will interrupt the CPU. The CPU can then carry out the data transfer. This also removes the need for the CPU to continually poll input devices to see if it must read any data. When an input device has data, then the appropriate I/O module can interrupt the CPU to request a data transfer.

Advantages of Interrupt-Driven I/O

- It is faster than Programmed I/O
- Efficient too

Disadvantages of Interrupt-Driven I/O

- It can be tricky to write if using a low level language.
- It can be tough to get various pieces to work well together.

Differentiate between programmed I/O and Interrupt Driven I/O

Programmed I/O	Interrupt Driven I/O
In programmed I/O, processor has to check each I/O devices in sequence and in effect ask each one if it needs communication with the processor. This checking is achieved by continuous polling cycle and hence processor cannot execute other instructions in sequence.	External asynchronous input is used to tell the processor that I/O device needs its service and hence processor does not have to check whether I/O device needs it service or not.
During polling processors is bushy and therefore have serious and decrement effect on system throughput.	In interrupt driven I/O, the processor is allowed to execute its instructions in sequence and only stop to service I/O device when it is told to do so by the device itself. This increases system throughput.
It is implemented without interrupt hardware support.	It is implemented using interrupt hardware support.
It does not depend on interrupt status.	Interrupt must be enabled to process interrupt driven I/O

It does not need initialization of stack.	It needs initialization of stack.
System throughput decreases as number of I/O devices connected in the system increases.	System throughput does not depend on the number of I/O devices connected in the system.

c. I/O using DMA

Although interrupt driven I/O is much more efficient than program controlled I/O, all data is still transferred through the CPU. This will be inefficient if large quantities of data are being transferred between the peripheral and memory. The transfer will be slower than necessary, and the CPU will be unable to perform any other actions while it is taking place. Many systems therefore use an additional strategy, known as direct memory access (DMA). DMA uses an additional piece of hardware- a DMA controller. The DMA controller can take over the system bus and transfer data between an I/O module and main memory without the intervention of the CPU. Whenever the CPU wants to transfer data, it tells the DMA controller the direction of the transfer, the I/O module involved, the location of the data in memory, and the size of the block of data to be transferred. It can then continue with other instructions and the DMA controller will interrupt it when the transfer is complete.

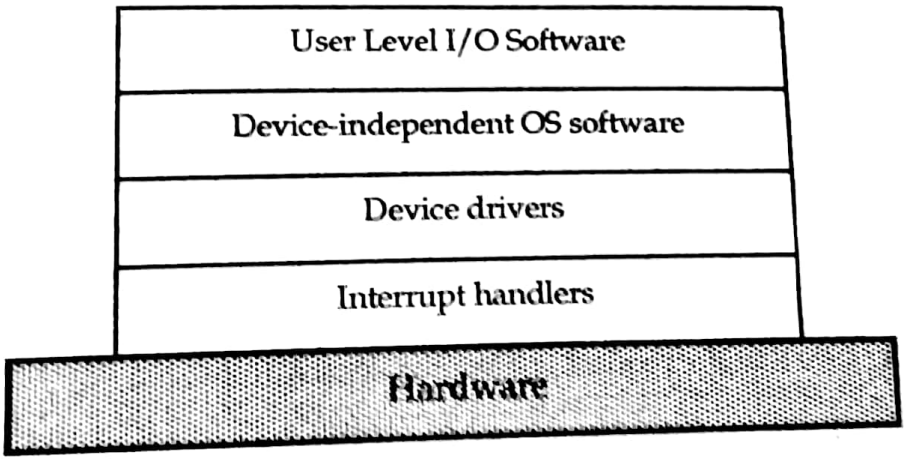
In brief, direct memory access (DMA) is a method that allows an input/output (I/O) device to send or receive data directly to or from the main memory, bypassing the CPU to speed up memory operations. The process is managed by a chip known as a DMA controller (DMAC).

8. I/O Software Layers

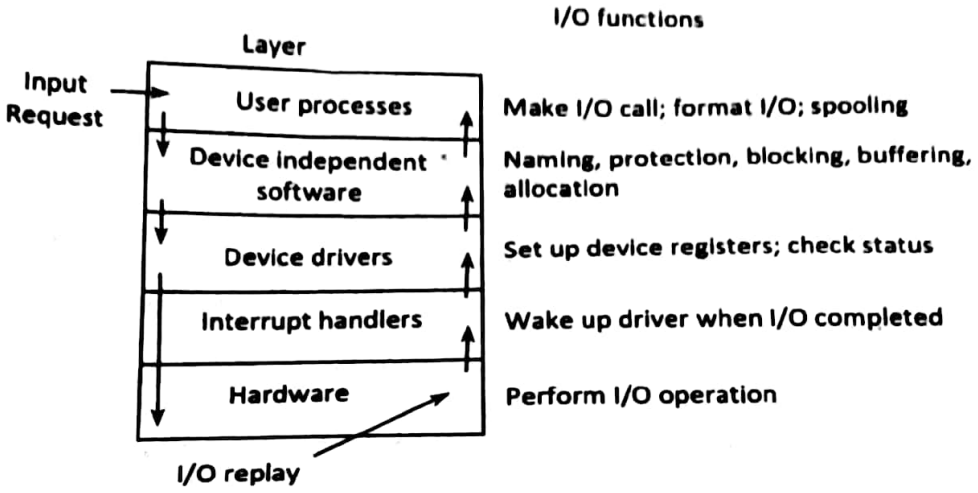
Basically, input/output software organized in the following four layers:

- Interrupt handlers
- Device drivers
- Device-independent input/output software
- User-space input/output software

In every input/output software, each of the above given four layer has a well-defined function to perform and a well-defined interface to the adjacent layers. The figure given below shows all the layers along with hardware of the input/output software system.



Here is another figure shows all the layers of the input/output software system along with their principal functions.



Now let's describe briefly, all the four input/output software layers that are listed above.

a. Interrupt Handlers

An interrupt handler, also known as an interrupt service routine or ISR, is a piece of software or more specifically a callback function in an operating system or more specifically in a device driver, whose execution is triggered by the reception of an interrupt.

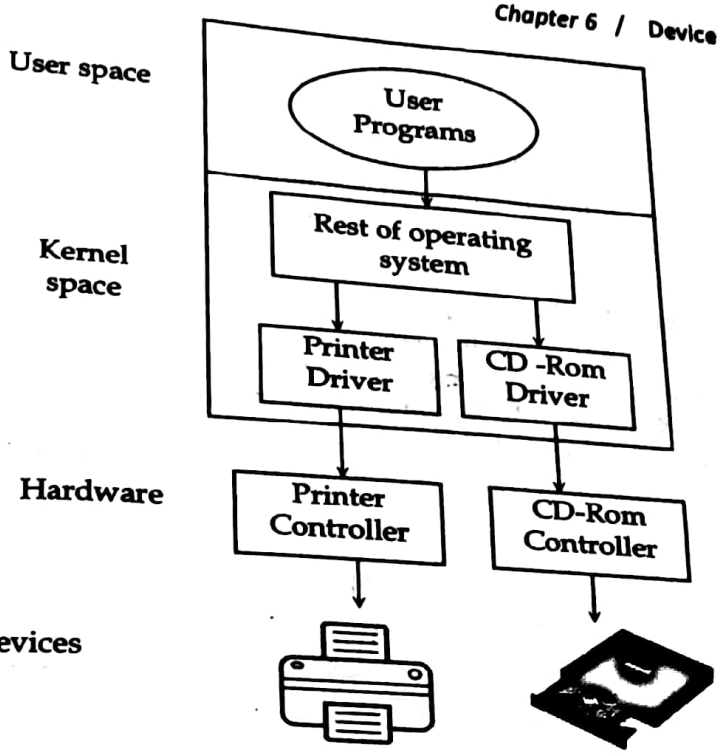
When the interrupt happens, the interrupt procedure does whatever it has to in order to handle the interrupt, updates data structures and wakes up process that was waiting for an interrupt to happen. The interrupt mechanism accepts an address - a number that selects a specific interrupt handling function from a small set. In most architecture, this address is an offset stored in a table called the interrupt vector table. This vector contains the memory addresses of specialized interrupt handlers.

b. Device Drivers

Basically, device drivers are a device-specific code just for controlling the input/output device that is attached to the computer system.

Device drivers are software modules that can be plugged into an OS to handle a particular device. Operating System takes help from device drivers to handle all I/O devices. Device drivers encapsulate device-dependent code and implement a standard interface in such a way that code contains device-specific register reads/writes. Device driver is generally written by the device's manufacturer and delivered along with the device on a CD-ROM.A device driver performs the following jobs:

- To accept request from the device independent software above to it.
- Interact with the device controller to take and give I/O and perform required error handling
- Making sure that the request is executed successfully



c. **Device-Independent Input/output Software**

In some of the input/output software is device specific and other parts of that input/output software are device-independent. The exact boundary between the device-independent software and drivers is device dependent, just because of that some functions that could be done in a device-independent way sometime be done in the drivers, for efficiency or any other reasons. Here are the lists of some functions that are done in the device-independent software:

- Uniform interfacing for device drivers
- Buffering
- Error reporting
- Allocating and releasing dedicated devices
- Providing a device-independent block size

d. **User-Space Input/output Software**

Generally most of the input/output software is within the operating system (OS), and some small part of that input/output software consists of libraries that are linked with the user programs and even whole programs running outside the kernel.

9. Disk Structure

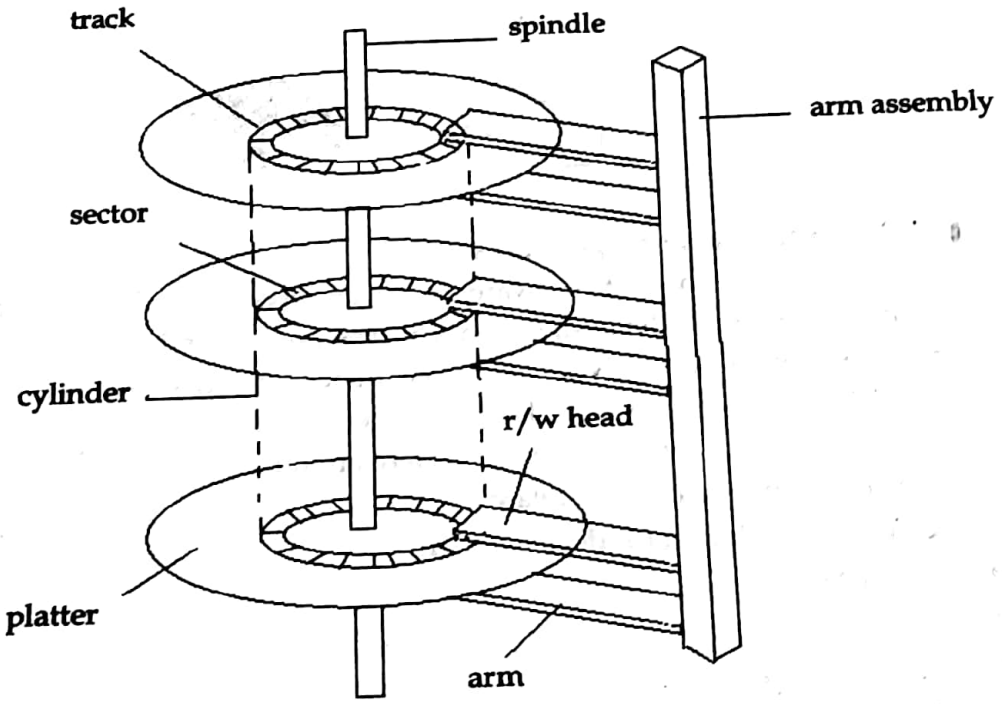
In modern computers, most of the secondary storage is in the form of magnetic disks. Hence, knowing the structure of a magnetic disk is necessary to understand how the data in the disk is accessed by the computer. Each modern disk contains concentric tracks and each track is divided into multiple sectors. The disks are usually arranged as a one dimensional array of blocks, where blocks are the smallest storage unit. Blocks can also be called as sectors. For each surface of the

disk, there is a read/write desk available. The same tracks on all the surfaces are known as a cylinder. Sector 0 is the first sector of the first track on the outermost cylinder. Mapping proceeds in order through that track, then the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost.

Tracks of the same distance from centre form a cylinder. A read-write head is used to read data from a sector of the magnetic disk. The speed of the disk is measured as two parts:

- **Transfer rate:** This is the rate at which the data moves from disk to the computer.
- **Random access time:** It is the sum of the seek time and rotational latency.

Seek time is the time taken by the arm to move to the required track. Rotational latency is defined as the time taken by the arm to reach the required sector in the track. Even though the disk is arranged as sectors and tracks physically, the data is logically arranged and addressed as an array of blocks of fixed size. The size of a block can be 512 or 1024 bytes. Each logical block is mapped with a sector on the disk, sequentially. In this way, each sector in the disk will have a logical address.



Disk structure key terms

- **Seek Time**

  Seek time is the time taken in locating the disk arm to a specified track where the read/write request will be satisfied.

- **Rotational Latency**

  Rotational Latency is the time taken by the desired sector of disk to rotate into a position so that it can access the read/write heads. So the disk scheduling algorithm that gives minimum rotational latency is better.

- **Transfer Time**

  Transfer time is the time to transfer the data. It depends on the rotating speed of the disk and number of bytes to be transferred.

- **Disk Access Time**

  Disk access time is given as,

  Disk Access Time = Rotational Latency + Seek Time + Transfer Time

- **Disk Response Time**

  Response Time is the average of time spent by a request waiting to perform its I/O operation. Average Response time is the response time of the all requests. Variance Response Time is measure of how individual request are serviced with respect to average response time. So the disk scheduling algorithm that gives minimum variance response time is better.

- **Transmission Time**

  To access a particular record, first the arm assembly must be moved to the appropriate cylinder, and then rotate the disk until it is immediately under the read write head. The time taken to access the whole record is called transmission time.

Numerical Problem 1: Consider a hard disk with:

4 surfaces

64 tracks/surface

128 sectors/track

256 bytes/sector

a. What is the capacity of the hard disk?

Disk capacity = surfaces * tracks/surface * sectors/track * bytes/sector

Disk capacity = 4 * 64 * 128 * 256

Thus, Disk capacity = 8 MB

b. The disk is rotating at 3600 RPM, what is the data transfer rate?

60 sec → 3600 rotations

1 sec → 60 rotations

Data transfer rate = number of rotations per second * track capacity * number of surfaces (since 1 R-W head is used for each surface)

Data transfer rate = 60 * 128 * 256 * 4

Data transfer rate = 7.5 MB/sec

c. The disk is rotating at 3600 RPM, what is the average access time?

Since, seek time, controller time and the amount of data to be transferred is not given, we consider all the three terms as 0.

Therefore, Average Access time = Average rotational delay

Rotational latency = 60 sec =3600 rotations

1 sec →60 rotations

Rotational latency = (1/60) sec = 16.67 msec.

Average Rotational latency = (16.67)/2

= 8.33 msec.

Average Access time = 8.33 msec.

Numerical problem 2: disk has an average seek time of 5ms, a rotational speed of 15,000 rpm, and 500 sectors per track. What is the average access time to read a single sector? What is the expected time to read 500 contiguous sectors on the same track? What is the expected time to read 500 sectors scattered over the disk?

Answer: Given,

Avg. seek time, Ts = 5ms = 5×10^{-3} s

Rotational speed, r = 15000 rpm = $\frac{15000}{60}$ rps = 250 rps

Sectors per track = 500

Hence, average time to read a single sector = $Ts + \frac{1}{2r} + \frac{1}{r*500}$

$= 5x\ 10^{-3} + \frac{1}{2x250} + \frac{1}{250x500}$

$= 7.008 \times 10^{-3}$ s

= 7.008 ms

Hence expected time read 500 contiguous sectors on the same track =

$7.008 \text{ ms} + \frac{1}{250x500} \times 499$ s

= 7.008 ms + 3.992 ms

= 11 ms

Hence, time to read 500 sectors scattered over the disk = 7.008 ms x 500

= 3.504 s

Problem 3: Consider a disk with a mean seek time of 8ms, a rotational rate of 15,000 rpm, and 262,144 bytes per track. What are the access times for block sizes of 1 KB, 2 KB, and 4 KB, respectively?

Answer: Given,

Average seek time, Ts = 8ms = $8x10^{-3}$ s

Rotational rate, r = 15000 rpm = $\frac{15000}{60}$ rps = 250 rps

Number of bytes per track, N = 262144

Hence, access time for 1 KB block = $Ts + \frac{1}{2r} + \frac{1024}{r \times N}$

$= 8 \times 10^{-3} + \frac{1}{2 \times 250} + \frac{1024}{250 \times 262144}$

= 10.0156 ms

Hence, access time for 2 KB block = $Ts + \frac{1}{2r} + \frac{1024}{r \times N}$

$= 8 \times 10^{-3} + \frac{1}{2 \times 250} + \frac{1024 \times 2}{250 \times 262144}$

= 10.03125 ms

Hence, access time for 4 KB block = $Ts + \frac{1}{2r} + \frac{1024}{r \times N}$

$= 8 \times 10^{-3} + \frac{1}{2 \times 250} + \frac{1024 \times 4}{250 \times 262144}$

= 10.0625 ms

10. Disk Scheduling

A hard disk drive is a collection of plates called platters. The surface of each platter is divided into circular tracks. Furthermore, each track is divided into smaller pieces called sectors. Disk I/O is done sector by sector. A group of tracks that are positioned on top of each other form a cylinder. There is a head connected to an arm for each surface, which handles all I/O operations. For each I/O request, first head is selected. It is then moved over the destination track. The disk is then rotated to position the desired sector under the head and finally, the read/write operation is performed. There are two objectives for any disk scheduling algorithm:

- **Maximize the throughput:** the average number of requests satisfied per time unit.
- **Minimize the response time:** the average time that a request must wait before it is satisfied.

Although there are other algorithms that reduce the seek time of all requests, we will only concentrate on the following major disk scheduling algorithms:

a. First Come-First Serve (FCFS)
b. Shortest Seek Time First (SSTF)
c. Elevator (SCAN)
d. Circular SCAN (C-SCAN)
e. LOOK
f. C-LOOK

a. First Come-First Serve (FCFS)

FCFS is the simplest of all the Disk Scheduling Algorithms. In FCFS, the requests are addressed in the order they arrive in the disk queue. Since no reordering of request takes place the head may move almost randomly across the surface of the disk. This policy aims to minimize response time with little regard for throughput.

Features of FCFS

- perform operations in order requested
- no reordering of work queue
- **no starvation:** every request is serviced
- poor performance

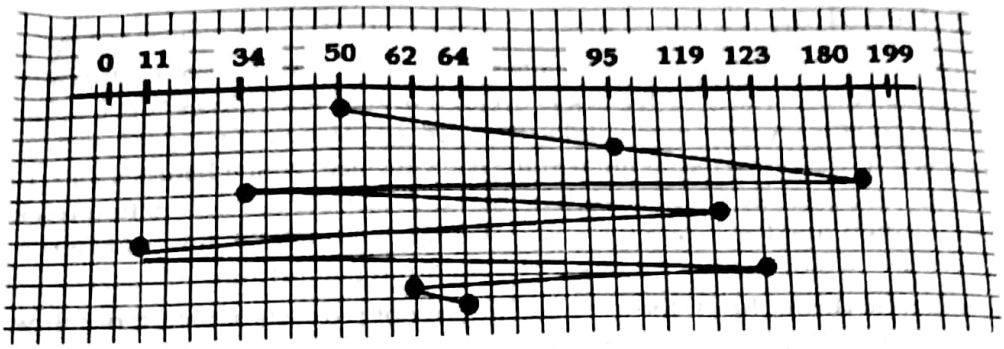
Advantages

- Every request gets a fair chance
- No indefinite postponement

Disadvantages

- Does not try to optimize seek time
- May not provide the best possible service

Example: Given the following queue {95, 180, 34, 119, 11, 123, 62, 64} with the Read-write head initially at the track 50 and the tail track being at 199. Then find total head movement by using FCFS disk scheduling algorithm.



Total disk movements =

(95-50)+(180-95)+(180-34)+(119-34)+(119-11)+(123-11)+(123-62)+(64-62)

= 45+85+146+85+108+112+61+2

= 644 tracks

b. Shortest Seek Time First (SSTF)

In SSTF requests having shortest seek time are executed first. So, the seek time of every request is calculated in advance in queue and then they are scheduled according to their calculated seek time. As a result, the request near the disk arm will get executed first. SSTF is certainly an improvement over FCFS as it decreases the average response time and increases the throughput of system. This policy will have better throughput than FCFS but a request may be delayed for a long period if many closely located requests arrive just after it.

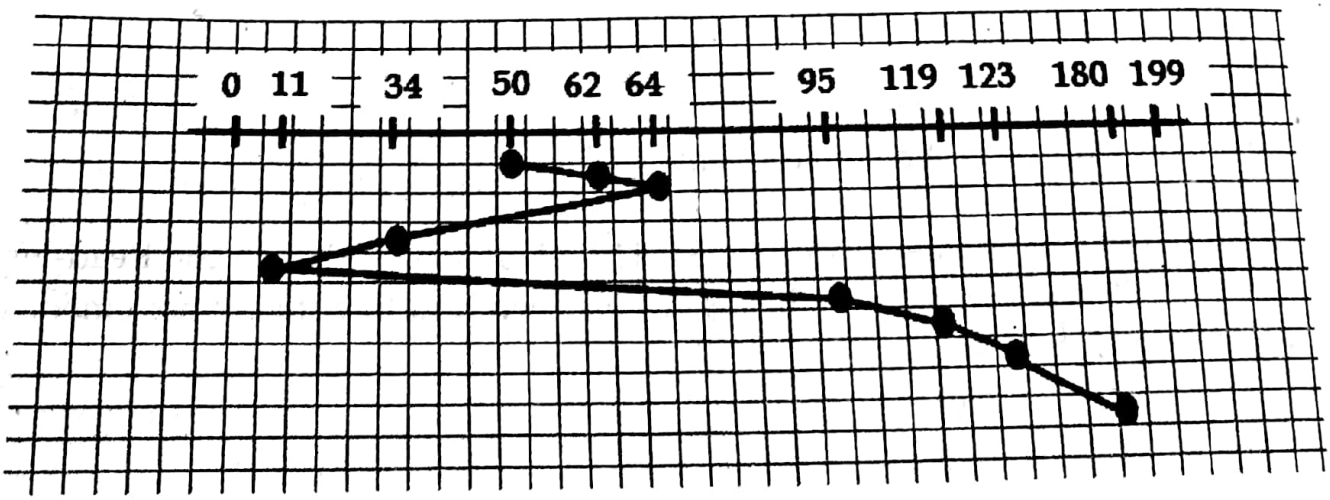
Advantages

- Average Response Time decreases
- Throughput increases

Disadvantages

- Overhead to calculate seek time in advance
- Can cause Starvation for a request if it has higher seek time as compared to incoming requests
- High variance of response time as SSTF favors only some requests

Example: Given the following queue {95, 180, 34, 119, 11, 123, 62, 64} with the Read-write head initially at the track 50 and the tail track being at 199. Then find total head movement by using SSTF disk scheduling algorithm.



Total disk movements =
(62-50)+(64-62)+(64-34)+(34-11)+(95-11)+(119-95)+(123-119)+(180-123)
= 12+2+30+23+84+24+4+57
= 236 tracks

c. Elevator (SCAN)

In SCAN algorithm the disk arm moves into a particular direction and services the requests coming in its path and after reaching the end of disk, it reverses its direction and again services the request arriving in its path. So, this algorithm works like an elevator and hence also known as elevator algorithm. As a result, the requests at the midrange are serviced more and those arriving behind the disk arm will have to wait.

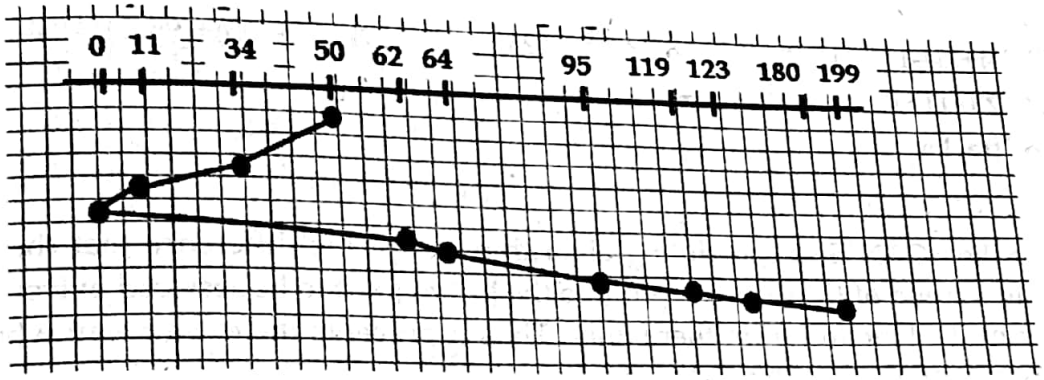
Advantages

- High throughput
- Low variance of response time
- Average response time

Disadvantages

- Long waiting time for requests for locations just visited by disk arm

Example: Given the following queue {95, 180, 34, 119, 11, 123, 62, 64} with the Read-write head initially at the track 50 and the tail track being at 199. Then find total head movement by using SCAN disk scheduling algorithm.



Total disk movements =
(50-34)+(34-11)+(11-0)+(62-0)+(64-62)+(95-64)+(119-95)+(123-119)+(180-123)
= 16+23+11+62+2+31+24+4+57
= 230 tracks

d. Circular SCAN (C-SCAN)

In SCAN algorithm, the disk arm again scans the path that has been scanned, after reversing its direction. So, it may be possible that too many requests are waiting at the other end or there may be zero or few requests pending at the scanned area. These situations are avoided in CSAN algorithm in which the disk arm instead of reversing its direction goes to the other end of the disk and starts servicing the requests from there. So, the disk arm moves in a circular fashion and this algorithm is also similar to SCAN algorithm and hence it is known as Circular SCAN.

Advantages

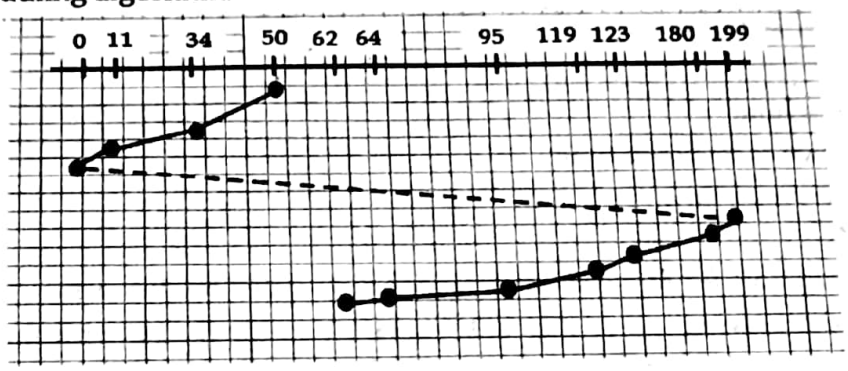
- The waiting time for the cylinders just visited by the head is reduced as compared to the SCAN Algorithm.

- It provides uniform waiting time.
- It provides better response time.

Disadvantages

- It causes more seek movements as compared to SCAN Algorithm.
- It causes the head to move till the end of the disk even if there are no requests to be serviced.

Example: Given the following queue {95, 180, 34, 119, 11, 123, 62, 64} with the Read-write head initially at the track 50 and the tail track being at 199. Then find total head movement by using CSCAN disk scheduling algorithm.



Total disk movements =

(50-34)+(34-11)+(11-0)+(199-180)+(180-123)+(123-119)+(119-95)+(95-64)+(64-62)

= 16+23+11+19+57+4+24+31+2

= 187 tracks

e. LOOK

It is similar to the SCAN disk scheduling algorithm except the difference that the disk arm in spite of going to the end of the disk goes only to the last request to be serviced in front of the head and then reverses its direction from there only. Thus it prevents the extra delay which occurred due to unnecessary traversal to the end of the disk.

The main difference between SCAN Algorithm and LOOK Algorithm is:

- SCAN Algorithm scans all the cylinders of the disk starting from one end to the other end even if there are no requests at the ends.
- LOOK Algorithm scans all the cylinders of the disk starting from the first request at one end to the last request at the other end.

Advantages

- It does not cause the head to move till the ends of the disk when there are no requests to be serviced.
- It provides better performance as compared to SCAN Algorithm.
- It does not lead to starvation.
- It provides low variance in response time and waiting time.

Disadvantages

- There is an overhead of finding the end requests.
- It causes long waiting time for the cylinders just visited by the head.

f. C-LOOK

Circular-LOOK Algorithm is an improved version of the LOOK Algorithm. Head starts from the first request at one end of the disk and moves towards the last request at the other end servicing all the requests in between. After reaching the last request at the other end, head reverses its direction. It then returns to the first request at the starting end without servicing any request in between. The same process repeats.

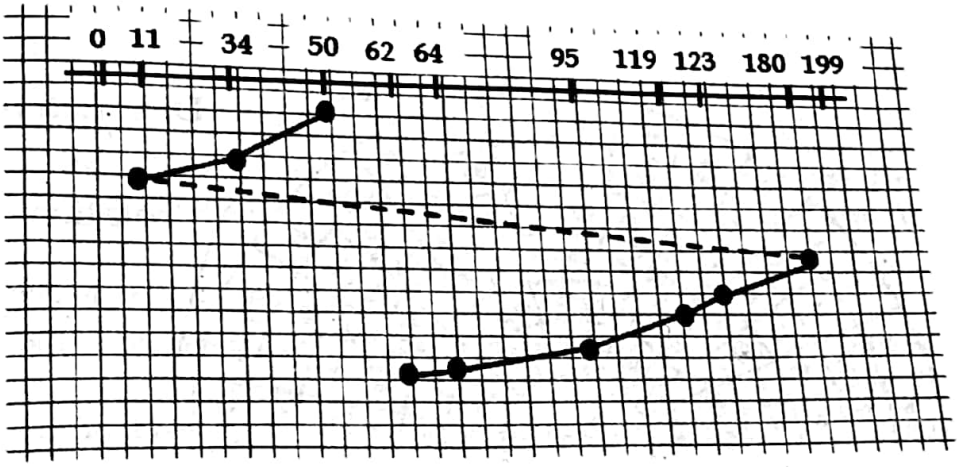
Advantages

- It does not cause the head to move till the ends of the disk when there are no requests to be serviced.
- It reduces the waiting time for the cylinders just visited by the head.
- It provides better performance as compared to LOOK Algorithm.
- It does not lead to starvation.
- It provides low variance in response time and waiting time.

Disadvantages

- There is an overhead of finding the end requests.

Example: Given the following queue {95, 180, 34, 119, 11, 123, 62, 64} with the Read-write head initially at the track 50 and the tail track being at 199. Then find total head movement by using C-LOOK disk scheduling algorithm.



Total disk movements =

(50-34)+(34-11)+(180-123)+(123-119)+(119-95)+(95-64)+(64-62)

= 16+23+57+4+24+31+2

= 157 tracks

11. Disk Formatting

Disk formatting is the configuring process of a data storage media such as a hard disk drive, floppy disk or flash drive for initial usage. Any existing files on the drive would be erased with disk formatting. Disk formatting is usually done before initial installation or before installation of a new operating system. Disk formatting is also done if there is a requirement for additional storage in the computer.

Before a disk can store a data, it must be divided into sectors that the disk controller can read and write, called low level formatting. The sector typically consists of preamble, data and ECC (error

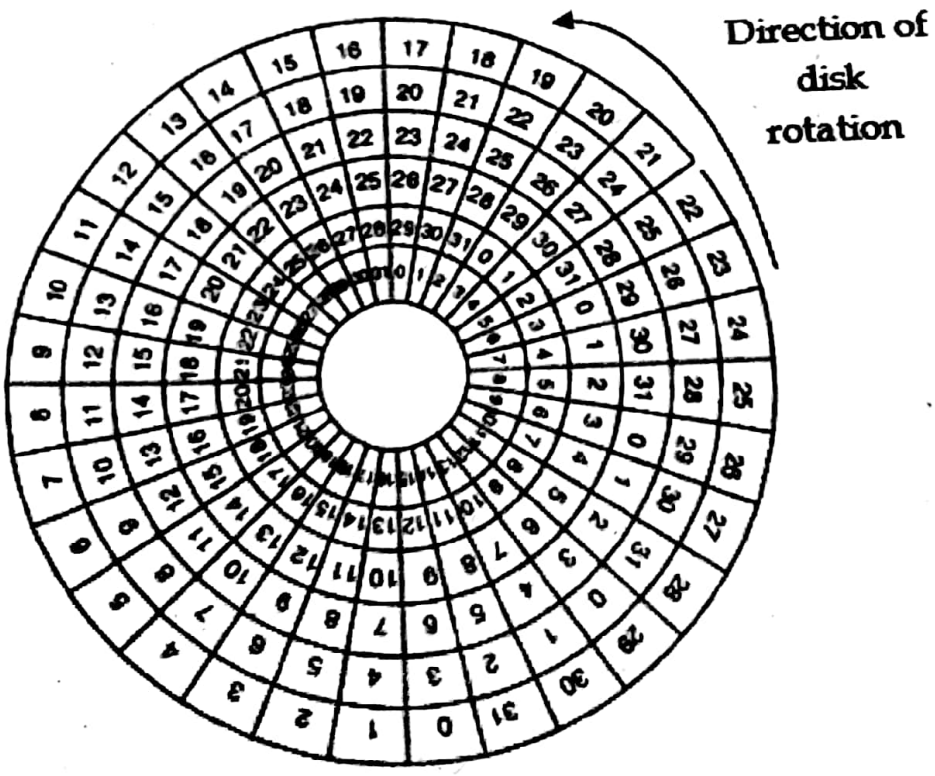
correcting code). The preamble contains the cylinder and the sector numbers and the ECC contains redundant information that can be used to recover from read error. The size depends upon the manufacturer, depending on reliability.

Preamble	Data	ECC

If disk I/O operations are limited to transferring the single sector at a time, it reads the first sector from the disk and doing the ECC calculation, and transfers to main memory, during this time the next sector will fly by the head.

A new magnetic disk is a blank state. Before a disk can store data it must be divided into sectors that disk controller can read and write. This process is called low-level or physical formatting.

- Disk must be formatted before storing data.
- Disk must be divided into sectors that the disk controllers can read/write.
- Low level formatting files the disk with a special data structure for each sector.
- Data structure consists of three fields: header, data area and trailer.
- Header and trailer contain information used by the disk controller.
- Sector number and Error Correcting Codes (ECC) contained in the header and trailer.
- For writing data to the sector –ECC is updated.
- For reading data from the sector– ECC is recalculated.
- Low level formatting is done at factory.



12. Cylinder Skew

If the position of sector s_i on each track is offset from the previous track then such an offset is called cylinder skew. It allows the disk to read multiple tracks in one continuous operation without losing data.

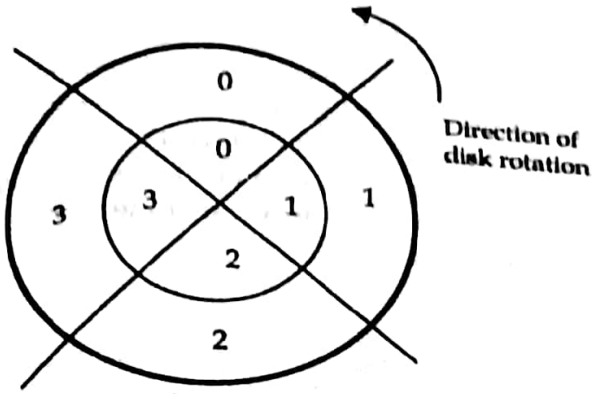


Fig: No Skew

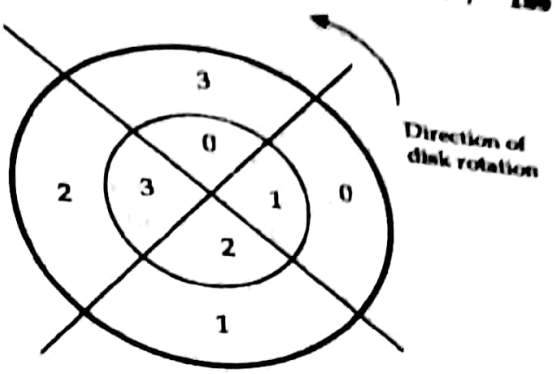


Fig: 1- Sector Skew

13. Interleaving

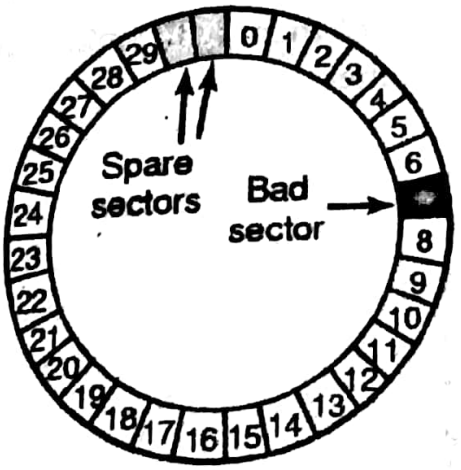
Interleave is the process through which gaps are placed between two sectors on the platter of a disk. This was done in earlier days when the computers were not quick enough to read continuous streams of data. Without interleaving there would be no gaps between the sectors and data would arrive immediately before the reading unit is ready. Due to this to read the same data a complete rotation of the disk would be required again. The interleaving ratio was not fixed and can be set by the end user depending on their system specs. Nowadays the ratio of interleaving is 1:1 i.e. no interleaving is used.

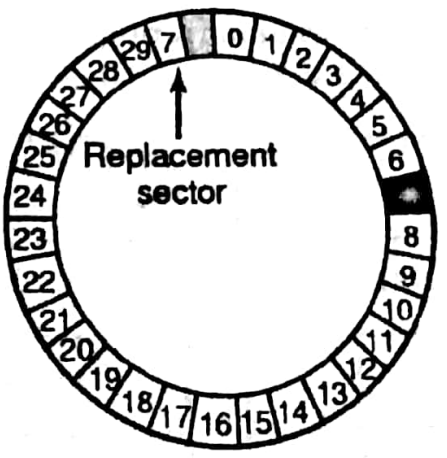
14. Error Handling

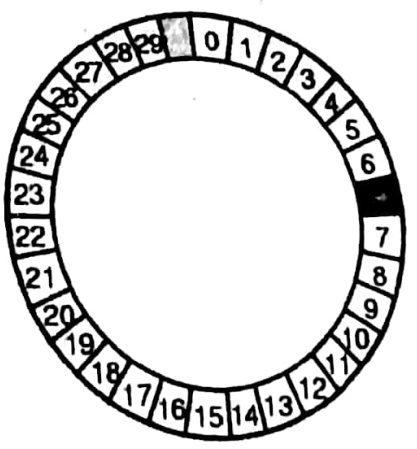
Manufacturing defects introduce bad sectors; if defect is small (say few bits). Then the hard drive is shipped and ECC corrects the error every time the sector is accessed. If error is bigger it cannot be masked.

There are two ways for error correction

a. When one of the sectors is bad the controller remaps between the bad and spare sectors as in figure.
b. If controller cannot remap, the operating system does the same thing in software.





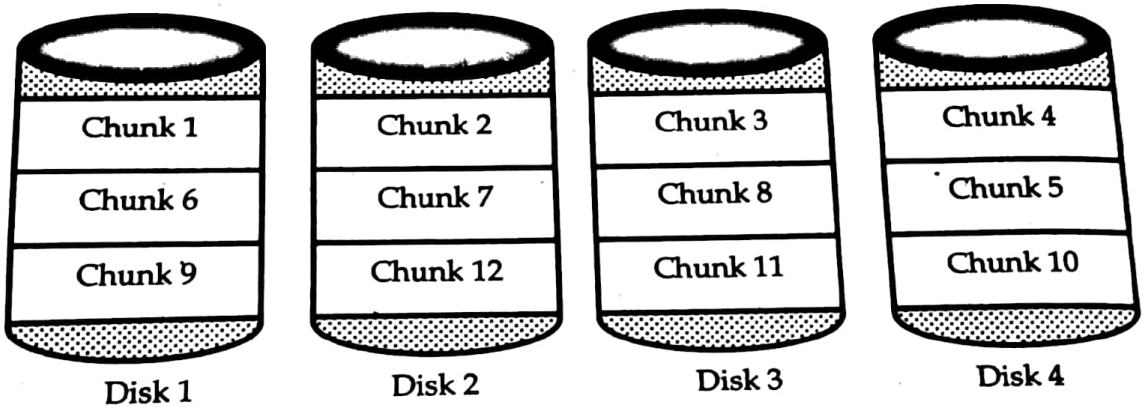


15. Redundant Array of Independent Disks (RAID)

RAID (redundant array of independent disks; originally redundant array of inexpensive disks) is a way of storing the same data in different places on multiple hard disks to protect data in the case of a drive failure. However, not all RAID levels provide redundancy. RAID works by placing data on multiple disks and allowing input/output (I/O) operations to overlap in a balanced way, improving performance. Because the use of multiple disks increases the mean time between failures (MTBF), storing data redundantly also increases fault tolerance.

What is Stripping in RAID?

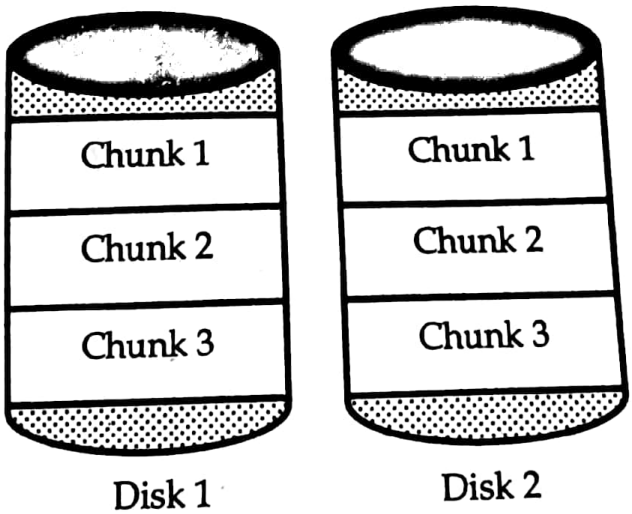
Writing data on a single disk is slower, but writing data by spreading it on multiple disks is faster (because data is written in small chunks to different disks, and also fetched in small chunks by different disks). When data is fetched from different disks, the CPU does not have to wait, because the throughput will be a combined one of all the disks. Each and every disk drive is partitioned in small chunks. Thus the process of dividing large data into multiple disks for faster access is called stripping in RAID.



In the above shown example diagram you can see that all four disks contain different data (data that needs to be stored will be striped in chunks and will be spread across different disks).There is no redundancy in this method, but is better known for high performance.

What is mirroring in Raid?

Mirroring is a mechanism in which the same data is written to another disk drive. The main advantage of mirroring (multiple sets of same data on two disks), is that it provides 100 percent redundancy. Suppose there are two drives in mirroring mode, then both of them will contain an exact same copy of data. So even if one disk fails, the data is safe on the other.



What is parity in RAID?

Parity is an interesting method used to rebuild data in case of failure of one of the disks. Parity makes use of a very famous mathematical binary operation called as "XOR".XOR is a mathematical operation that's done to produce one output from two inputs. Some examples of XOR operations are as below.

1'st operator	2'nd operator	XOR Output
1	1	0
1	0	1
0	1	1
0	0	0

We can simply make a rule while performing XOR binary operation, that if there is a difference in the operator then the XOR output is 1.In the above shown example table consider the columns **1'st operator** and **2nd operator** as hard disks in a RAID array, and the third column **XOR Output** as a parity disk. And now if one of the disks fails, we can easily construct the data on the failed disk with the help of the parity disk and the other disk which is not failed.

What are hot spares in RAID?

Hot Spare is an extra drive added to the disk array, to increase the fault tolerance. If we have a hot spare in our RAID disk array, the RAID controller will automatically start rebuilding data on to that hot spare drive, if one of the disks from the array fails. Which means the hot spare will automatically take the role of the failed drive once data rebuilding is complete. We can later on replace our failed drive. RAID management software's will provide you with a mechanism to specify the hot spare drive for your array.

Different Levels of RAID

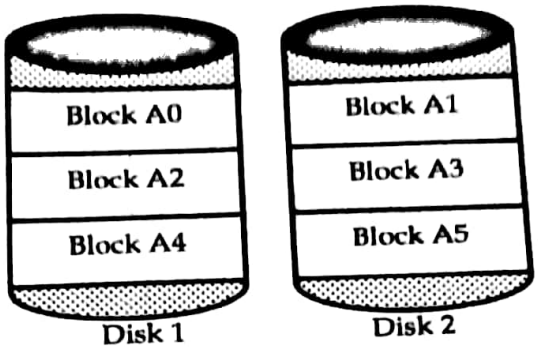
RAID (redundant array of independent disks), can be classified to different levels based on its operation and level of redundancy provided. There is no one size fits all solution as far as raid levels are concerned. Selecting the suitable raid level for your application depends on the following things.

- You can select a raid level based on the performance that it provides
- RAID level based on the level of redundancy it provides
- RAID level based on read and writes operations.

Let's discuss some of the widely used RAID levels.

RAID 0 or No RAID

If your main priority is performance, then raid 0 fits right. An important fact that should be kept in mind is that, RAID 0 does not provide any kind of redundancy. This means even if one drive fails, your data is at risk. It is simply striping done on your disk array. Data is broken into smaller chunks and are spread across the number of disks you have. It has no mirroring, no parity (which means no redundancy).



In fact RAID level 0 is not RAID, because RAID was primarily build for redundancy, and RAID 0 does not provide any kind of redundancy, although it provides high performance.

Calculation

No. of Disk: 5
Size of each disk: 100GB
Usable Disk size: 500GB

Advantages

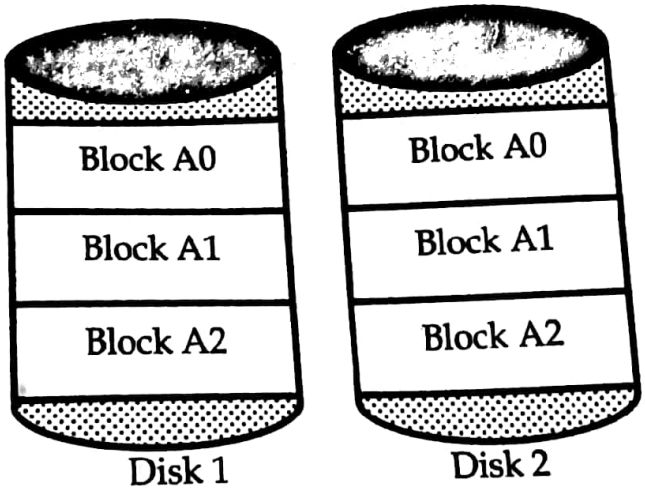
- Data is stripped into multiple drives
- Disk space is fully utilized
- High performance
- The technology is easy to implement.
- Minimum 2 drives required

Disadvantages

- No support for Data Redundancy
- No support for Fault Tolerance
- No error detection mechanism
- Failure of either disk results in complete data loss in respective array

RAID level 1(RAID 1)

RAID 1 implements heavy use of mirroring. All data in the drive is duplicated to another drive. It can be used in a situation where fault tolerance is of primary importance. This level performs mirroring of data in drive 1 to drive 2. It offers 100% redundancy as array will continue to work even if either disk fails. So organization looking for better redundancy can opt for this solution but again cost can become a factor. You can refer the diagram shown in mirroring of raid section in this article, for RAID 1.



Calculation

No. of Disk: 2

Size of each disk: 100GB

Usable Disk size: 100GB

Advantages

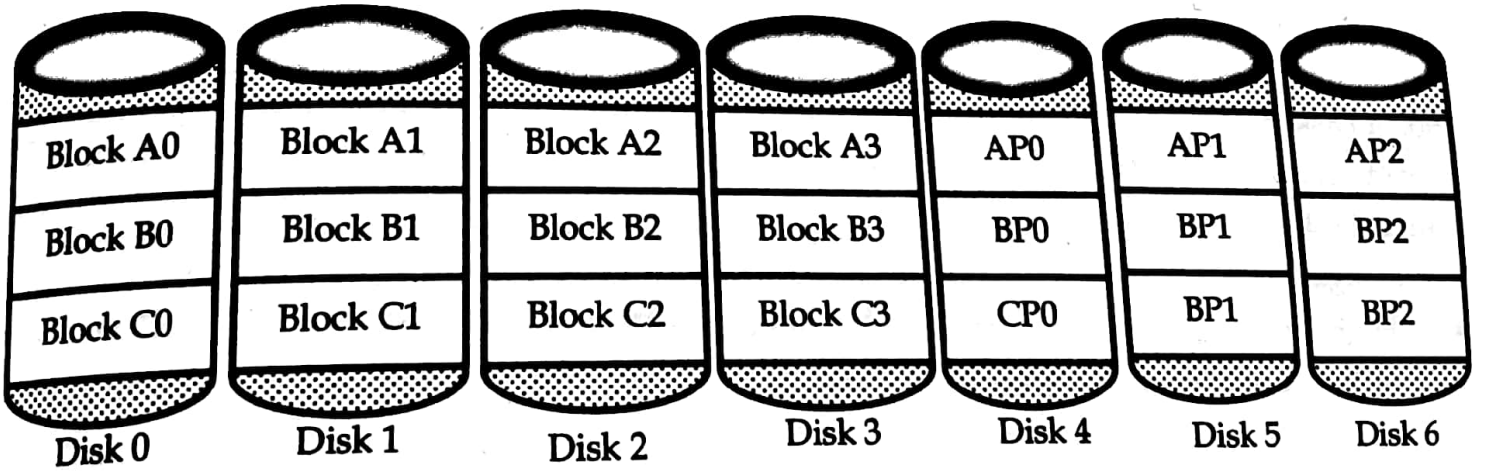
- Performs mirroring of data i.e. identical data from one drive is written to another drive for redundancy.
- High read speed as either disk can be used if one disk is busy
- Array will function even if any one of the drive fails
- Minimum 2 drives required
- RAID 1 offers excellent read speed and a write-speed that is comparable to that of a single drive.
- In case a drive fails, data do not have to be rebuilding, they just have to be copied to the replacement drive.
- RAID 1 is a very simple technology.

Disadvantages

- Slow write performance as all drives has to be updated
- The main disadvantage is that the effective storage capacity is only half of the total drive capacity because all data get written twice.
- Software RAID 1 solutions do not always allow a hot swap of a failed drive. That means the failed drive can only be replaced after powering down the computer it is attached to. For servers that are used simultaneously by many people, this may not be acceptable. Such systems typically use hardware controllers that do support hot swapping.

RAID 2

This level uses bit-level data stripping rather than block level. To be able to use RAID 2 make sure the disk selected has no self disk error checking mechanism as this level uses external Hamming code for error detection. This is one of the reasons RAID is not in the existence in real IT world as most of the disks used these days come with self error detection. It uses an extra disk for storing all the parity information.



Calculation

Formula: n-1 where n is the no. of disk

No. of Disk: 7

Size of each disk: 100GB
Usable Disk size: 600GB

Advantages

- BIT level stripping with parity
- One designated drive is used to store parity
- Uses Hamming code for error detection

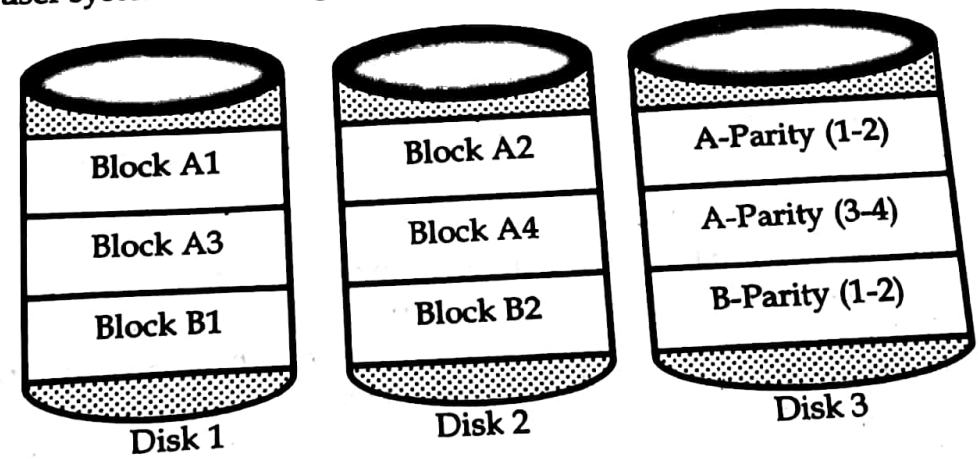
Disadvantages

- It is used with drives with no built in error detection mechanism
- Additional drives required for error detection

RAID 3

This level uses byte level stripping along with parity. One dedicated drive is used to store the parity information and in case of any drive failure the parity is restored using this extra drive. But in case the parity drive crashes then the redundancy gets affected again so not much considered in organizations.

This technique uses striping and dedicates one drive to storing parity information. The embedded ECC information is used to detect errors. Data recovery is accomplished by calculating the exclusive OR (XOR) of the information recorded on the other drives. Since an I/O operation addresses all the drives at the same time, RAID 3 cannot overlap I/O. For this reason, RAID 3 is best for single-user systems with long record applications.



Calculation

Formula: n-1 where n is the no. of disk

No. of Disk: 3

Size of each disk: 100GB

Usable Disk size: 200GB

Advantages

- High throughput for transferring large amounts of data
- BYTE level stripping with parity
- High data transfer rates (for large sized files)
- Data is accessed parallel

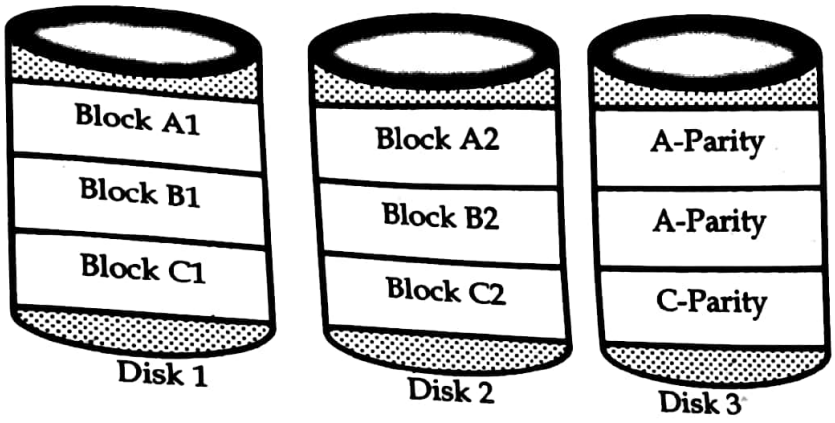
Disadvantages

- Additional drives required for parity

- No redundancy in case parity drive crashes
- Slow performance for operating on small sized files

RAID 4

Level 4 provides block-level striping (like Level 0) with a parity disk. If a data disk fails, the parity data is used to create a replacement disk. A disadvantage to Level 4 is that the parity disk can create write bottlenecks. This level uses large stripes, which means you can read records from any single drive. This allows you to use overlapped I/O for read operations. Since all write operations have to update the parity drive, no I/O overlapping is possible.



Calculation

Formula: n-1 where n is the no. of disk

No. of Disk: 3

Size of each disk: 100GB

Usable Disk size: 200GB

Advantages

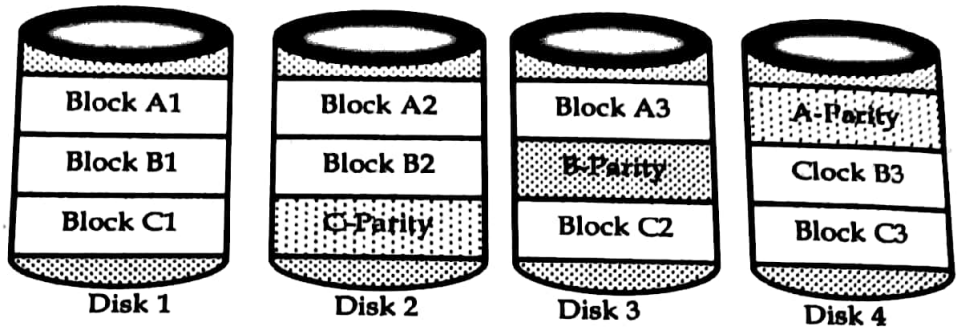
- BLOCK level stripping along with dedicated parity
- Data is accessed independently
- High read performance since data is accessed independently.

Disadvantages

- Since only 1 block is accessed at a time so performance degrades
- Write operation becomes slow as every time parity has to be entered
- Additional drives required for parity

RAID 5

RAID 5 is the most common secure RAID level. It requires at least 3 drives but can work with up to 16. Data blocks are striped across the drives and on one drive a parity checksum of all the block data is written. The parity data are not written to a fixed drive, they are spread across all drives, as the drawing below shows. Using the parity data, the computer can recalculate the data of one of the other data blocks, should those data no longer be available. That means a RAID 5 array can withstand a single drive failure without losing data or access to data. Although RAID 5 can be achieved in software, a hardware controller is recommended. Often extra cache memory is used on these controllers to improve the write performance.



Calculation

Formula: n-1 where n is the no. of disk

No. of Disk: 4

Size of each disk: 100GB

Usable Disk size: 300GB

Advantages

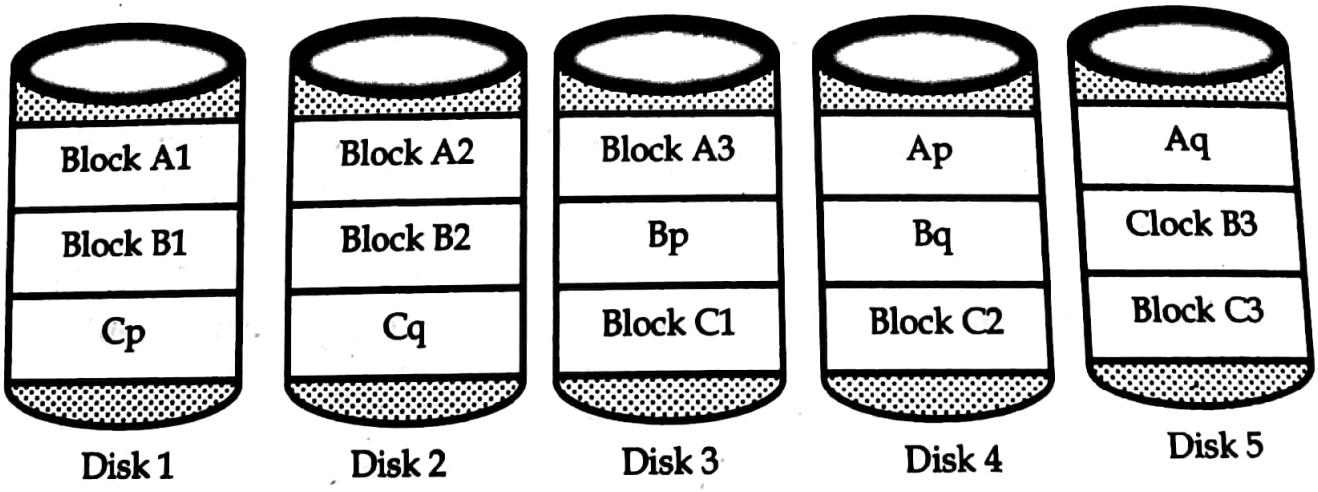
- Read data transactions are fast as compared to write data transactions that are somewhat slow due to the calculation of parity.
- Data remains accessible even after the drive failure and during replacement of the failed hard drive because the storage controller rebuilds the data on the new drive.

Disadvantages

- Drive failures have an effect on throughput, although this is still acceptable.
- This is complex technology. If one of the disks in an array using 4TB disks fails and is replaced, restoring the data (the rebuild time) may take a day or longer, depending on the load on the array and the speed of the controller. If another disk goes bad during that time, data are lost forever.

RAID 6 – Striping with Double Parity

RAID Level 6 is like RAID 5 with block-level striping with two distributed parity. This means it requires a minimum 4 drives and provides fault tolerance up to 2 failed drives. It takes hours or days to rebuild raid array. During the rebuilding of RAID 5 if any other drive also fails then all data will be lost. But in RAID 6 the raid array will survive if the second drives also failed. So RAID 6 is more secure than RAID 5



Advantages

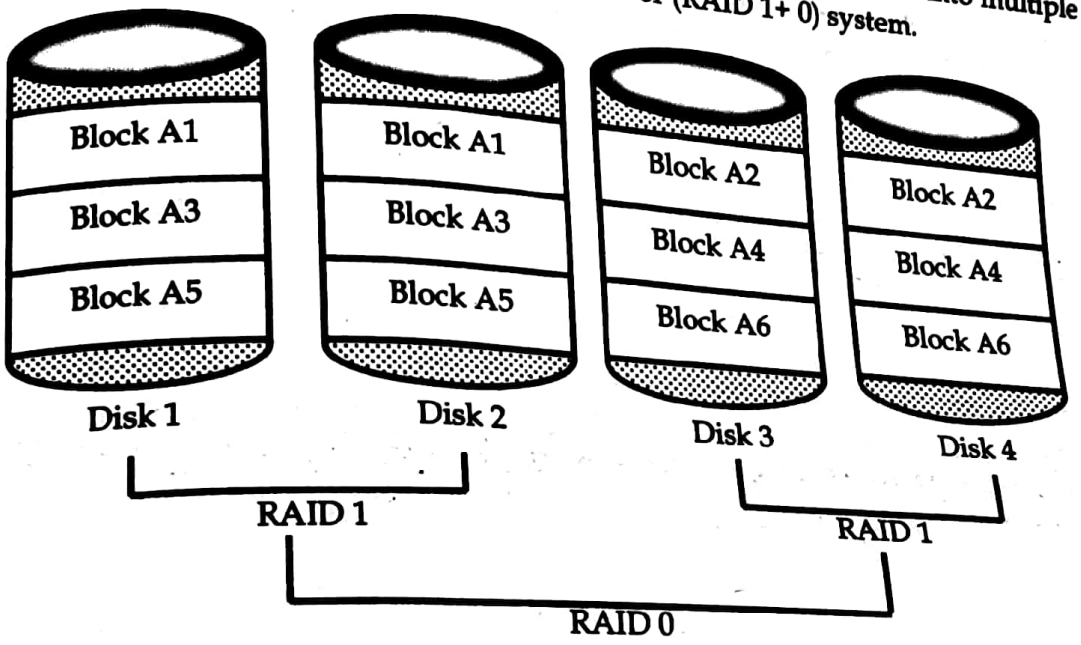
- Like with RAID 5, read data transactions are very fast.
- If two drives fail, you still have access to all data, even while the failed drives are being replaced. So RAID 6 is more secure than RAID 5.

Disadvantages

- Write data transactions are slower than RAID 5 due to the additional parity data that have to be calculated.
- Drive failures have an effect on throughput, although this is still acceptable.
- This is complex technology. Rebuilding an array in which one drive failed can take a long time.

RAID 10 (Combination of RAID 0 and RAID 1)

RAID 10 is a good solution that will give you both the performance advantage of RAID 0 and also the redundancy of RAID 1 mirroring. RAID 10 was made by a combination of RAID 0 and RAID 1. It provides both security and performance by mirroring and striping data into multiple drives. It requires minimum 4 drives to build RAID Level 10 or (RAID 1+ 0) system.



Advantages

- If something goes wrong with one of the disks in a RAID 10 configuration, the rebuild time is very fast since all that is needed is copying all the data from the surviving mirror to a new drive. This can take as little as 30 minutes for drives of 1 TB.
- Combination of two levels makes it fast and resilient at same time.
- Mirroring makes RAID 10 secure

Disadvantages

- Half of the storage capacity goes to mirroring, so compared to large RAID 5 or RAID 6 arrays; this is an expensive way to have redundancy.
- Not used for mission critical systems

Laboratory Works

Simulate free space management techniques and disk scheduling algorithms.

Program 1: Best Fit Algorithm

- Get no. of Processes and no. of blocks.
- After that get the size of each block and process requests.
- Then select the best memory block that can be allocated using the above definition.
- Display the processes with the blocks that are allocated to a respective process.
- Value of Fragmentation is optional to display to keep track of wasted memory.

Source code

```
#include<stdio.h>
void main()
{
        int fragment[20], b[20], p[20], i, j, nb, np, temp, lowest=9999;
        static int barray[20], parray[20];
        printf("\n\t\t\t Memory Management Scheme - Best Fit");
        printf("\n Enter the number of blocks:");
        scanf("%d", &nb);
        printf("Enter the number of processes:");
        scanf("%d", &np);

        printf("\n Enter the size of the blocks:-\n");
        for(i=1; i<=nb; i++)
      {
              printf("Block no.%d:",i);
              scanf("%d",&b[i]);
       }
       printf("\n Enter the size of the processes :-\n");
       for(i=1;i<=np;i++)
       {
              printf("Process no.%d:",i);
              scanf("%d", &p[i]);
       }
       for(i=1; i<=np; i++)
       {
              for(j=1; j<=nb; j++)
              {
                     if(barray[j]!=1)
                     {
                            temp=b[j]-p[i];
                            if(temp>=0)
                                   if(lowest>temp)
                                   {
                                          parray[i]=j;
                                          lowest=temp;
                                   }
```

```
                }
                fragment[i]=lowest;
                barray[parray[i]]=1;
                lowest=10000;
        }
        printf("\nProcess_no\tProcess_size\tBlock_no\tBlock_size\tFragment");
        for(i=1;i<=np && parray[i]!=0;i++)
                printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,p[i],parray[i],b[parray[i]],
                fragment[i]);
}
```

Program 2: First Fit Algorithm

1. Get no. of Processes and no. of blocks.
2. After that get the size of each block and process requests.
3. Now allocate processes
4. if(block size >= process size)
   //allocate the process
5. else
   //move on to next block
6. Display the processes with the blocks that are allocated to a respective process.
7. Stop.

Source code

```
#include<stdio.h>
void main()
{
        int bsize[10], psize[10], bno, pno, flags[10], allocation[10], i, j;
        for(i = 0; i < 10; i++)
        {
                flags[i] = 0;
                allocation[i] = -1;
        }
        printf("Enter no. of blocks: ");
        scanf("%d", &bno);
        printf("\nEnter size of each block: ");
        for(i = 0; i < bno; i++)
                scanf("%d", &bsize[i]);
        printf("\nEnter no. of processes: ");
        scanf("%d", &pno);

        printf("\nEnter size of each process: ");
        for(i = 0; i < pno; i++)
                scanf("%d", &psize[i]);
        for(i = 0; i < pno; i++)        //allocation as per first fit
                for(j = 0; j < bno; j++)
                        if(flags[j] == 0 && bsize[j] >= psize[i])
```

```
                {
                        allocation[j] = i;
                        flags[j] = 1;
                        break;
                }
        //display allocation details
        printf("\n Block no.\tsize\t\tprocess no.\t\tsize");
        for(i = 0; i < bno; i++)
        {
                printf("\n%d\t\t%d\t\t", i+1, bsize[i]);
                if(flags[i] == 1)
                        printf("%d\t\t\t%d",allocation[i]+1,psize[allocation[i]]);
                else
                        printf("Not allocated");
        }
}
```

Program 3: C Program For Shortest Seek Time First Disk Scheduling Algorithm

```
#include<stdio.h>
struct head
{
    int num;
    int flag;
};
int main()
{
    struct head h[33];
    int array_1[33], array_2[33];
    int count = 0, j, x, limit, minimum, location, disk_head, sum = 0;
    printf("\nEnter total number of locations:\t");
    scanf("%d", &limit);
    printf("\n Enter position of disk head:\t");
    scanf("%d", &disk_head);
    printf("\nEnter elements of disk head queue\n");
    while(count < limit)
    {
        scanf("%d", &h[count].num);
        h[count].flag = 0;
        count++;
    }
    for(count = 0; count < limit; count++)
    {
        x = 0;
        minimum = 0;
        location = 0;
        for(j = 0; j < limit; j++)
        {
```

```
            if(h[j].flag == 0)
            {
                if(x == 0)
                {
                    array_1[j] = disk_head - h[j].num;
                    if(array_1[j] < 0)
                    {
                        array_1[j] = h[j].num - disk_head;
                    }
                    minimum = array_1[j];
                    location = j;
                    x++;
                }
                else
                {
                    array_1[j] = disk_head - h[j].num;
                    if(array_1[j] < 0)
                    {
                        array_1[j] = h[j].num - disk_head;
                    }
                }
                if(minimum > array_1[j])
                {
                    minimum = array_1[j];
                    location = j;
                }
            }
        }
        h[location].flag = 1;
        array_2[count] = h[location].num - disk_head;
        if(array_2[count] < 0)
        {
            array_2[count] = disk_head - h[location].num;
        }
        disk_head = h[location].num;
    }
    count = 0;
    while(count < limit)
    {
        sum = sum + array_2[count];
        count++;
    }
    printf("\nTotal movements of the cylinders:\t%d", sum);
    return 0;
}
```

Program 4: C Program for SCAN Disk Scheduling Algorithm

```
#include<stdio.h>
#include<conio.h>
void scan_algorithm(int left[], int right[], int count, int limit)
{
    int arr[20];
    int x = count - 1, y = count + 1, c = 0, d = 0, j;
    while(x > -1)
    {
        printf("\nX:\t%d", x);
        printf("\nLeft[X]:\t%d", left[x]);
        arr[d] = left[x];
        x--;
        d++;
    }
    arr[d] = 0;
    while(y < limit + 1)
    {
        arr[y] = right[c];
        c++;
        y++;
    }
    printf("\nScanning Order:\n");
    for(j = 0; j < limit + 1; j++)
    {
        printf("\n%d", arr[j]);
    }
}
void division(int elements[], int limit, int disk_head)
{
    int count = 0, p, q, m, x;
    int left[20], right[20];
    for(count = 0; count < limit; count++)
    {
        if(elements[count] > disk_head)
        {
            printf("\nBreak Position:\t%d\n", elements[count]);
            break;
        }
    }
    printf("\nValue:\t%d\n", count);
    q = 1;
    p = 0;
    m = limit;
    left[0] = elements[0];
    printf("\n Left:\t%d", left[0]);
    while(q < count)
```

```
        {
            printf("\n Element[l] value:\t%d" , elements[q]);
            left[q] = elements[q];
            printf("\n Left:\t%d", left[q]);
            q++;
            printf("\nl:\t%d", q);
        }
        x = count;
        while(x < m)
        {
            right[p] = elements[x];
            printf("\n Right:\t%d", right[p]);
            printf("\n Element:\t%d", elements[x]);
            p++;
            x++;
        }
        scan_algorithm(left, right, count, limit);
    }
    void sorting(int elements[], int limit)
    {
        int location, count, j, temp, small;
        for(count = 0; count < limit - 1; count++)
        {
            small = elements[count];
            location = count;
            for(j = count + 1; j < limit; j++)
            {
                if(small > elements[j])
                {
                    small = elements[j];
                    location = j;
                }
            }
            temp = elements[location];
            elements[location] = elements[count];
            elements[count] = temp;
        }
    }
    int main()
    {
        int count, disk_head, elements[20], limit;
        printf("Enter total number of locations:\t");
        scanf("%d", &limit);
        printf("\n Enter position of disk head:\t");
        scanf("%d", &disk_head);
        printf("\n Enter elements of disk head queue\n");
        for(count = 0; count < limit; count++)
```

```
    {
        printf("Element[%d]:\t", count + 1);
        scanf("%d", &elements[count]);
    }
    sorting(elements, limit);
    division(elements, limit, disk_head);
    getch();
    return 0;
}
```

Exercise

1. Which of the following disk scheduling techniques has a drawback of starvation?
2. Consider a file system on a disk that has both logical and physical block sizes of 512 bytes. Assume that the information about each file is already in memory. For each of the three allocation strategies (contiguous, linked, and indexed), answer these questions:
   a. How is the logical-to-physical address mapping accomplished in this system? (For the indexed allocation, assume that a file is always less than 512 blocks long)
   b. If we are currently at logical block 10 (the last block accessed was block 10) and want to access logical block 4, how many physical blocks must be read from the disk?
3. In what situations would use memory as a RAM disk be more useful than using it as a disk cache?
4. None of the disk-scheduling disciplines, except FCFS, are truly fair (starvation may occur).
   a. Explain why this assertion is true.
   b. Describe a way to modify algorithms such as SCAN to ensure fairness.
   c. Explain why fairness is an important goal in a time-sharing system.
   d. Give three or more examples of circumstances in which it is important that the operating system be unfair in serving I/O requests.
5. Suppose that a disk drive has 5,000 cylinders, numbered 0 to 4999. The drive is currently serving a request at cylinder 143, and the previous request was at cylinder 125. The queue of pending requests, in FIFO order, is

   {86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130}

   Starting from the current head position, what is the total distance (in cylinders) that the disk arm moves to satisfy all the pending requests for each of the following disk-scheduling algorithms?

   a. FCFS b. SSTF c. SCAN d. LOOK
   e. C-SCAN f. C-LOOK
6. Is disk scheduling, other than FCFS scheduling, useful in a single-user environment? Explain your answer.
7. Compare the performance of C-SCAN and SCAN scheduling, assuming a uniform distribution of requests. Consider the average response time (the time between the arrival of a request and the completion of that request's service), the variation in response time, and the effective band width. How does performance depend on the relative sizes of seek time and rotational latency?

8. Why rotational latency is usually not considered in disk scheduling? How would you modify SSTF, SCAN, and C-SCAN to include latency optimization?
9. Why is it important to balance file system I/O among the disks and controllers on a system in a multitasking environment?
10. Is there any way to implement truly stable storage? Explain your answer.
11. Discuss the relative advantages and disadvantages of sector sparing and sector slipping.
12. What would be the effects on cost and performance if tape storage had the same areal density as disk storage?
13. If magnetic hard disks eventually have the same cost per gigabyte as do tapes, will tapes become obsolete, or will they still be needed? Explain your answer.
14. What is a bad sector? Explain the purpose of interleave?
15. What you understand by access time. Explain different types of timings in a hard disk drive?
16. What is RAID? What are the significant properties of a stable storage system? Discuss.
17. What are the impact of buffering in data transfer between I/O devices and operating system kernel?
18. Suppose that a disk drive has 5000 cylinders numbered 0 to 4999. The drive is currently serving a request at cylinder 143, and the previous request was at cylinder 125. The queue of pending requests, in FIFO order is:

    85, 1465, 920, 1784, 948, 1510, 1025, 1745, 128

    Starting from the current head position, what is the total distance (in cylinders) that the disk arm moves to satisfy all the pending requests for each of the following disk scheduling algorithms?

    a. SSTF b. SCAN c. C-SCAN d. C-LOOK
    e. LOOK f. FIFO
19. Determine the total head movement for algorithms SSTF and C-LOOK for the following requests (in track number) where the current position of head is on track number 0.

    98, 181, 23, 65, 122, 14, 72, 36, 26, 67
20. A disk has an average seek time of 5ms, a rotational speed of 15,000 rpm, and 500 sectors per track. What is the average access time to read a single sector? What is the expected time to read 500 contiguous sectors on the same track? What is the expected time to read 500 sectors scattered over the disk?

❑❑❑

Chapter 7
LINUX CASE STUDY

1. History

LINUX is one of popular version of UNIX operating System. It is open source as its source code is freely available. It is free to use. LINUX was designed considering UNIX compatibility. Its functionality list is quite similar to that of UNIX. LINUX operating system is one of the popular versions of the UNIX operating system, which is designed to offer a free or low cost operating system for personal computer users. It gained the reputation as a fast performing and very efficient system. This is a remarkably complete operating system, including a GUI (graphical user interface), TCP/IP, the Emacs editor, can X Window System, etc.

The History of LINUX began in the 1991 with the beginning of a personal project by a Finland student Linus Torvalds to create a new free operating system kernel. Since then, the resulting LINUX Kernel has been marked by constant growth throughout the history.

- In the year 1991, LINUX was introduced by a Finland student Linus Torvalds.
- Hewlett Packard UNIX(HP-UX) 8.0 was released.
- In the year 1992, Hewlett Packard 9.0 was released.
- In the year 1993, NetBSD 0.8 and FreeBSD 1.0 released.
- In the year 1994, Red Hat LINUX was introduced; Caldera was founded by Bryan Sparks and Ransom Love and NetBSD1.0 Released.
- In the year 1995, FreeBSD 2.0 and HP UX 10.0 were released.
- In the year 1996, K Desktop Environment was developed by Matthias Ettrich.
- In the year 1997, HP-UX 11.0 was released.
- In the year 1998, the fifth generation of SGI Unix i.e. IRIX 6.5, Sun Solaris 7 operating system and Free BSD 3.0 was released.
- In the year 2000, the agreement of Caldera Systems with SCO server software division and the professional services division was announced.
- In the year 2001, Linus Torvalds released the LINUX2.4 version source code.
- In the year 2001, Microsoft filed a trademark suit against Lindows.com
- In the year 2004, Lindows name was changed to Linspire.
- In the year 2004, the first release of Ubuntu was released.
- In the year 2005, the project, openSUSE began a free distribution from Novell's community.
- In the year 2006, Oracle released its own distribution of Red Hat.
- In the year 2007, Dell started distributing laptops with Ubuntu pre installed in it.
- In the year 2011, LINUX kernel 3.0 versions were released.

- In the year 2013, Google LINUX based Android claimed 75% of the Smartphone market share, in terms of the number of phones shipped.
- In the year 2014, Ubuntu claimed 22,000,000 users.

2. Features of LINUX Operating System

The main features of LINUX operating system are listed below:

- **Portable:** LINUX operating system can work on different types of hardware as well as LINUX kernel supports the installation of any kind of hardware platform.
- **Open Source:** Source code of LINUX operating system is freely available and, to enhance the ability of the LINUX operating system, many teams work in collaboration.
- **Multiuser:** LINUX operating system is a multiuser system, which means; multiple users can access the system resources like RAM, Memory or Application programs at the same time.
- **Multiprogramming:** LINUX operating system is a multiprogramming system, which means multiple applications can run at the same time.
- **Hierarchical File System:** LINUX operating system affords a standard file structure in which system files or user files are arranged.
- **Shell:** LINUX operating system offers a special interpreter program that can be used to execute commands of the OS. It can be used to do several types of operations like call application programs, and so on.
- **Security:** LINUX operating system offers user security systems using authentication features like encryption of data or password protection or controlled access to particular files.

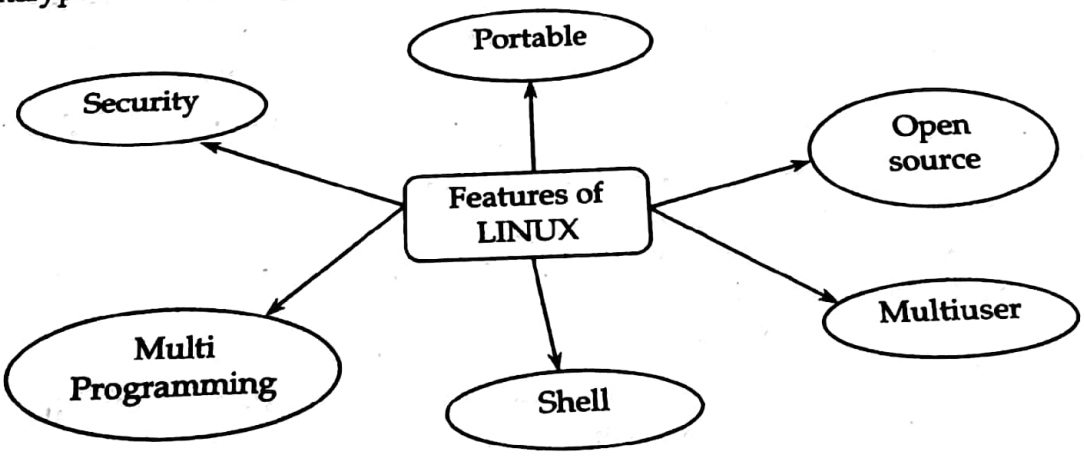


Fig 7.1: Features of LINUX.

3. Components of LINUX System

LINUX Operating System has primarily three components

a. **Kernel:** Kernel is the core part of Linux. It is responsible for all major activities of this operating system. It consists of various modules and it interacts directly with the underlying hardware. Kernel provides the required abstraction to hide low level hardware details to system or application programs.

b. **System Library:** System libraries are special functions or programs using which application programs or system utilities accesses Kernel's features. These libraries implement most of

the functionalities of the operating system and do not require kernel module's code access rights.

c. **System Utility:** System Utility programs are responsible to do specialized, individual level tasks.

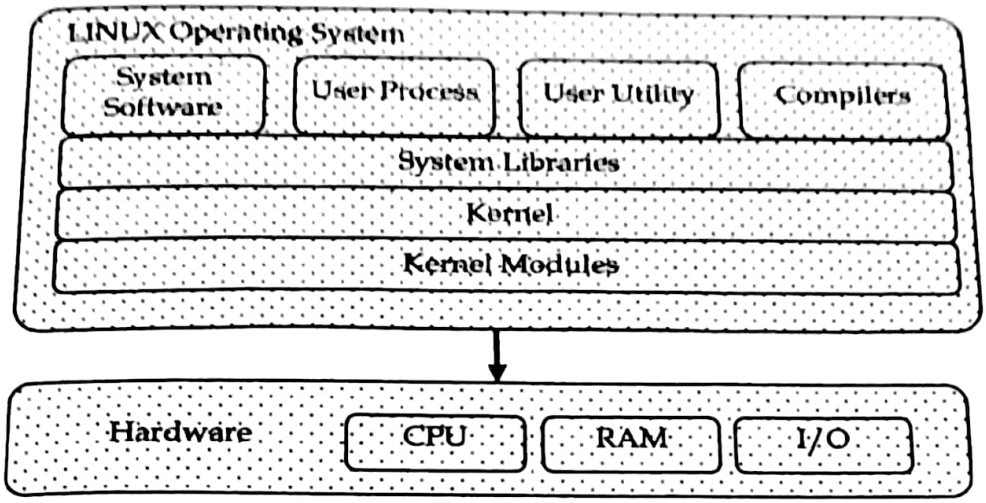


Fig. 7.2: Linux operating system

4. LINUX System Architecture

The LINUX Operating System's architecture primarily has these components: the Kernel, Hardware layer, System library, Shell and System utility.

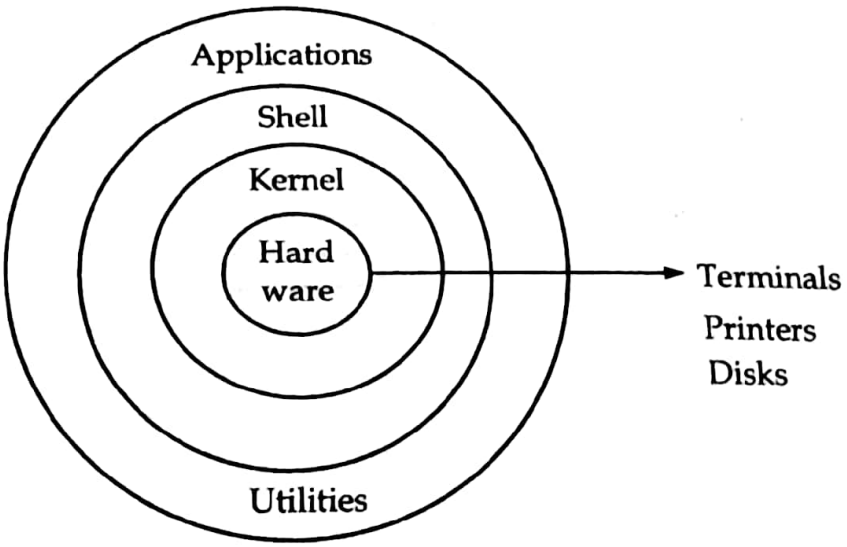


Fig 7.3: LINUX system architecture.

1. The kernel is the core part of the operating system, which is responsible for all the major activities of the LINUX operating system. This operating system consists of different modules and interacts directly with the underlying hardware. The kernel offers the required abstraction to hide application programs or low-level hardware details to the system. The types of Kernels are as follows:
   - Monolithic Kernel
   - Micro kernels

- Exo kernels
- Hybrid kernels

2. System libraries are special functions, that are used to implement the functionality of the operating system and do not require code access rights of kernel modules
3. System Utility programs are liable to do individual and specialized level tasks
4. Hardware layer of the LINUX operating system consists of peripheral devices such as RAM, HDD, CPU
5. The shell is an interface between the user and the kernel, and it affords services of the kernel It takes commands from the user and executes kernel's functions. The Shell is present in different types of operating systems, which are classified into two types command line shells and graphical shells.The command line shells provide a command line interface, while the graphical line shells provide a graphical user interface. Though both shells perform operations, but the graphical user interface shells perform slower than the command line interface shells. Types of shells are classified into four:

- Korn shell
- Bourne shell
- C shell
- POSIX shell

5. Kernel Modules

The LINUX kernel is a monolithic kernel i.e. it is one single large program where all the functional components of the kernel have access to all of its internal data structures and routines. The alternative to this is the micro kernel structure where the functional pieces of the kernel are broken out into units with strict communication mechanism between them. This makes adding new components into the kernel, via the configuration process, rather time consuming. The best and the robust alternative is the ability to dynamically load and unload the components of the operating system using LINUX kernel Modules.

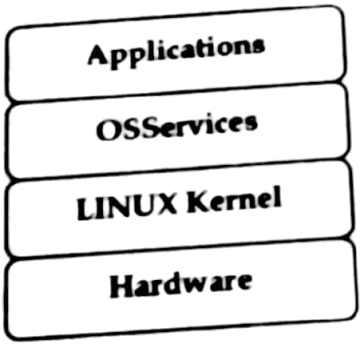


Fig. 7.4: Kernel Modules

Applications and OS services

These are the user application running on the LINUX system. These applications are not fixed but typically include applications like email clients, text processors etc. OS services include utilities and services that are traditionally considered part of an OS like the windowing system, shells, programming interface to the kernel, the libraries and compilers etc.

Kernel abstracts the hardware to the upper layers. The kernel presents the same view of the hardware even if the underlying hardware is different. It mediates and controls access to system resources.

Hardware

This layer consists of the physical resources of the system that finally do the actual work. This includes the CPU, the hard disk, the parallel port controllers, the system RAM etc.

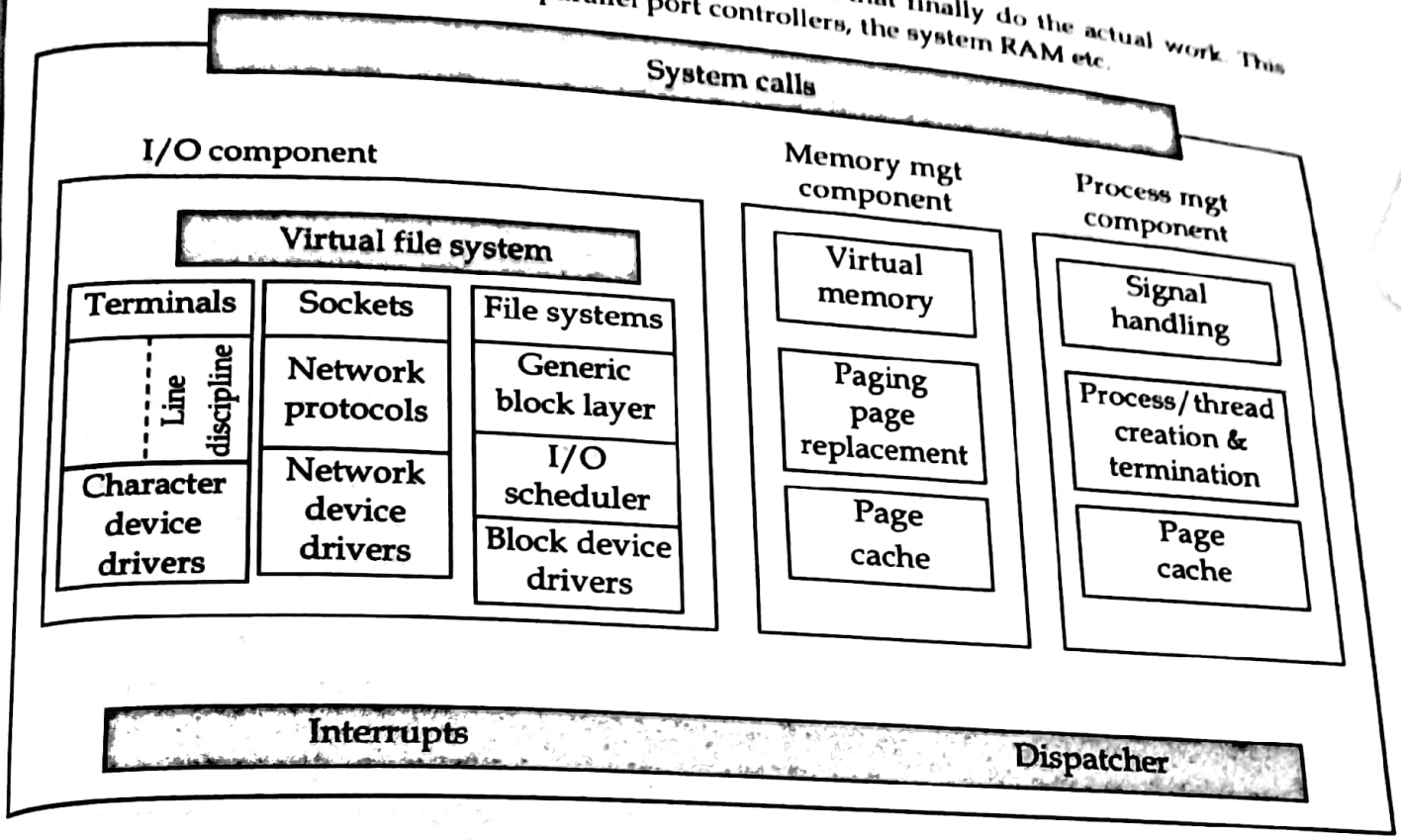


Fig 7.5: Structure of the LINUX kernel.

Purpose of the Kernel

The LINUX kernel presents a virtual machine interface to user processes. Processes are written without needing any knowledge of the type of the physical hardware that constitutes the computer. The LINUX kernel abstracts all hardware into a consistent interface. In addition, LINUX Kernel supports multi-tasking in a manner that is transparent to user processes: each process can act as though it is the only process on the computer, with exclusive use of main memory and other hardware resources. The kernel actually runs several processes concurrently, and mediates access to hardware resources so that each process has fair access while inter-process security is maintained.

The kernel code executes in privileged mode called kernel mode. Any code that does not need to run in privileged mode is put in the system library. The interesting thing about LINUX kernel is that it has a modular architecture – even with binary codes: LINUX kernel can load and unload modules dynamically just as it can load or unload the system library modules.

Here we shall explore the conceptual view of the kernel without really bothering about the implementation issues. Kernel code provides for arbitrations and for protected access to HW

resources. Kernel supports services for the applications through the system libraries. System calls within applications may also use system library. For instance, the buffered file handling is operated and managed by LINUX kernel through system libraries. Programs like utilities that are needed to initialize the system and configure network devices are classed as user mode programs and do not run with kernel privileges. Programs like those that handle login requests are run as system utilities and also do not require kernel privileges.

6. Process Management

A process refers to a program in execution; it's a running instance of a program. It is made up of the program instruction, data read from files, other programs or input from a system user.
User process as also the kernel processes seek the CPU and otherservices. Usually a fork system call results in creating a new process. System call execve results in execution of a newly **forked** process. Processes have an id (PID) and also havea user id (UID) like in UNIX. LINUX additionally has a personality associated with aprocess. Personality of a process is used by emulation libraries to be able to cater to arange of implementations. Usually a forked process inherits parent's environment.In LINUX two vectors define a process: these are argument vector and environmentvector. The environment vector essentially has a (name, value) value list whereindifferent environment variable values are specified. The argument vector has thecommand line arguments used by the process. Usually the environment is inherited however, upon execution of execve the process body may be redefined with a new set of environment variables. This helps in the customization of a process's operationalenvironment. Usually a process also has some indication on its scheduling context.Typically a process context includes information on scheduling, accounting, file tables,capability on signal handling and virtual memory context.
In Linux, internally, both processes and threads have the same kind of representation. LINUX processes and threads are POSIX compliant and are supported by a threads librarypackage which provides for two kinds of threads: user and kernel. User-controlledscheduling can be used for user threads. The kernel threads are scheduled by the kernel.While in a single processor environment there can be only one kernel thread scheduled.In a multiprocessor environment one can use the kernel supported library and clonesystem call to have multiple kernel threads created and scheduled

Types of Processes

There are fundamentally two types of processes in Linux:

- **Foreground processes (also referred to as interactive processes)** – these are initialized and controlled through a terminal session. In other words, there has to be a user connected to the system to start such processes; they haven't started automatically as part of the system functions/services.
- **Background processes (also referred to as non-interactive/automatic processes)** theseare processes not connected to a terminal; they don't expect any user input.

7. Creation of a Processes in Linux

A new process is normally created when an existing process makes an exact copy of itself in memory. The child process will have the same environment as its parent, but only the process ID number is different. There are two conventional ways used for creating a new process in Linux:

- **Using The System () Function** – this method is relatively simple, however, it's inefficient and has significantly certain security risks.
- **Using fork () and exec () Function** – this technique is a little advanced but offers greater flexibility, speed, together with security.

States of a Process in Linux

During execution, a process changes from one state to another depending on its environment/circumstances. In Linux, a process has the following possible states:

- **Running** – here it's either running (it is the current process in the system) or it's ready to run (it's waiting to be assigned to one of the CPUs).
- **Waiting** – in this state, a process is waiting for an event to occur or for a system resource. Additionally, the kernel also differentiates between two types of waiting processes; interruptible waiting processes – can be interrupted by signals and uninterruptible waiting processes – are waiting directly on hardware conditions and cannot be interrupted by any event/signal.
- **Stopped** – in this state, a process has been stopped, usually by receiving a signal. For instance, a process that is being debugged.
- **Zombie** – here, a process is dead, it has been halted but it's still has an entry in the process table.

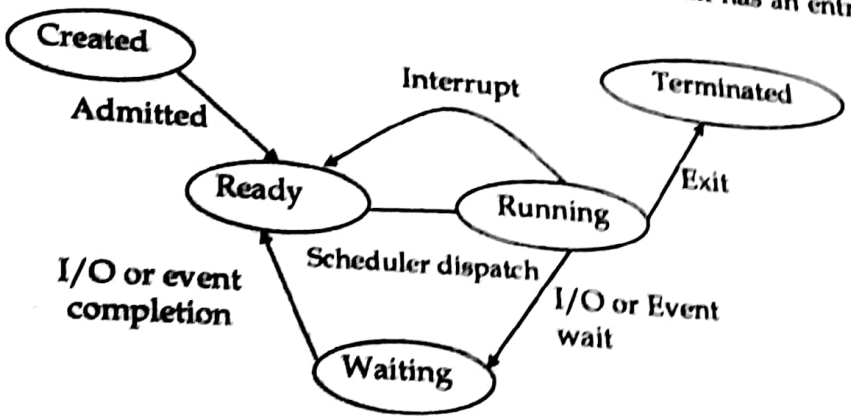


Fig. 7.6 States of a process in Linux

8. Scheduler

Schedulers control the access to CPU by implementing some policy such that the CPU is shared in a way that is fair and also the system stability is maintained. In LINUX scheduling is required for the user processes and the kernel tasks. Kernel tasks may be internal tasks on behalf of the drivers or initiated by user processes requiring specific OS services. Examples are: a page fault or because some device driver raises an interrupt. In Linux, normally, the kernel mode of operation cannot be pre-empted. Kernel code runs to completion - unless it results in a page fault, or an interrupt of some kind or kernel code itself calls the scheduler. LINUX is a time sharing system. So a timer interrupt happens and rescheduling may be initiated at that time. LINUX uses a credit based scheduling algorithm. The process with the highest credits gets scheduled. The credits are revised after every run. If all run-able processes exhaust all the credits a priority based fresh credit allocation takes place. The crediting system usually gives higher credits to interactive or IO bound processes – as these require immediate responses from a user. LINUX also implements UNIX like nice process characterization.

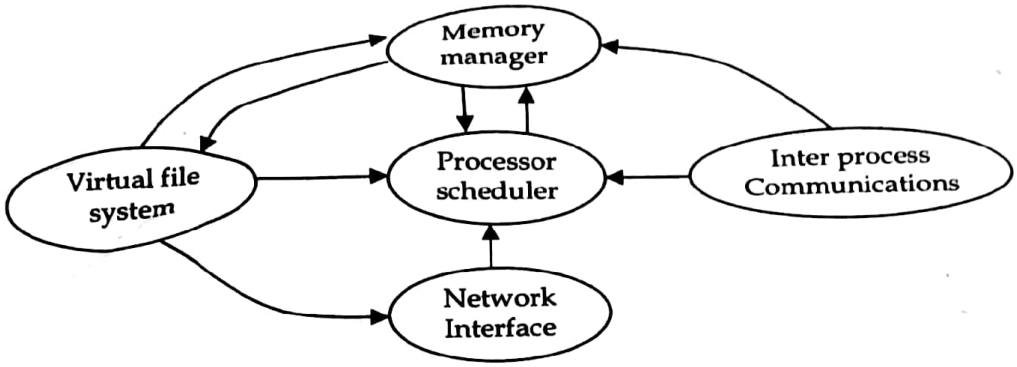


Fig. 7.7: Scheduler

9. Inter-process Communication

Inter Process Communication (IPC) refers to a mechanism, where the operating systems allow various processes to communicate with each other. This involves synchronizing their actions and managing shared data.The IPC primitives for processes also reside on the same system. With the explanationabove we should think of the typical loadable kernel module in LINUX to have three maincomponents:

- Module management,
- Driver registration and
- Conflict resolution mechanism.

9.1 Module Management

For new modules this is done at two levels – the management of kernel referencedsymbols and the management of the code in kernel memory. The LINUX kernelmaintains a symbol table and symbols defined here can be exported explicitly. The new module must seek thesymbols. In fact this is like having an external definition in C and then getting thedefinition at the kernel compile time. The module management system also definesall the required communications interfaces for this newly inserted module. With thisdone, processes can request the services from this module.

9.2 Driver registration

The kernel maintains a dynamic table which gets modified once a new module is added some times one may wish to delete also. In writing these modules care is taken to ensure that initializations and cleaning up operations are defined for the driver. A module may register one or more drivers of one or more types of drivers. Usually the registration of drivers is maintained in a registration table of the module. The registration of drives entails the following:

- Driver context identification: as a character or bulk device or a network driver
- File system context: essentially the routines employed to store files in LINUX virtual file system or network file system like NFS
- Network protocols and packet filtering rules
- File formats for executable and other files

The PC hardware configuration is supported by a large number of chip set configurations and with a large range of drivers for SCSI devices, video display devices and adapters, network cards. This results in the situation where we have module device drivers which vary over a very wide range of capabilities and options. This necessitates a conflict resolution mechanism to resolve accesses in a variety of conflicting concurrent accesses. The conflict resolution mechanisms help in preventing modules from having an access conflict to the HW for example an access to a printer. Modules usually identify the HW resources it needs at the time of loading and the kernel makes these available by using a reservation table. The kernel usually maintains information on the address to be used for accessing HW be it DMA channel or an interrupt line. The drivers avail kernel services to access HW resources.

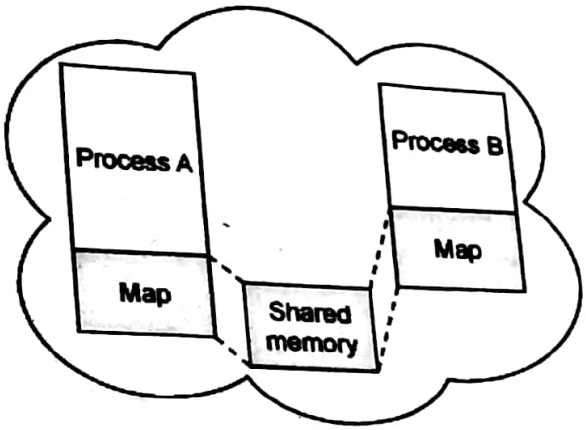


Fig. 7.8: Conflict resolution

10. Memory Management

The Memory Management Issues

The two major components in LINUX memory management are:

- The page management
- The virtual memory management

The page management: Pages are usually of a size which is a power of 2. Given the main memory LINUX allocates a group of pages using a buddy system. The allocation is the responsibility of software called "page allocator". Page allocator software is responsible for both allocation, as well as, freeing the memory. The basic memory allocator uses a buddy heap which allocates contiguous area of size 2^n> the required memory with minimum n obtained by successive generation of "buddies" of equal size. We explain the buddy allocation using an example.

An Example: Suppose we need memory of size 1556 words. Starting with a memory size 16K we would precede as follows:

- First create 2 buddies of size 8k from the given memory size i.e. 16K
- From one of the 8K buddy create two buddies of size 4K each
- From one of the 4k buddy create two buddies of size 2K each.
- Use one of the most recently generated buddies to accommodate the 1556 size memory requirement.

Note that for a requirement of 1556 words, memory chunk of size 2K words satisfies the property of being the smallest chunk larger than the required size.

Virtual memory Management

The basic idea of a virtual memory system is to expose address space to a process. A process should have the entire address space exposed to it to make an allocation or de-allocation. LINUX makes a conscious effort to allocate logically, "page aligned" contiguous address space. Such page aligned logical spaces are called regions in the memory. LINUX organizes these regions to form a binary tree structure for fast access. In addition to the above logical view the LINUX kernel maintains the physical view i.e. maps the hardware page table entries that determine the location of the logical page in the exact location on a disk. The process address space may have private or shared pages. Changes made to a page require that locality is preserved for a process by maintaining a copy-on-write when the pages are private to the process where as these have to be visible when they are shared.

A process, when first created following a fork system call, finds its allocation with a new entry in the page table – with inherited entries from the parent. For any page which is shared amongst the processes (like parent and child), a reference count is maintained. LINUX has a far more efficient page swapping algorithm than Unix – it uses a second chance algorithm dependent on the usage pattern. The manner it manifests itself is that a page gets a few chances of survival before it is considered to be no longer useful. Frequently used pages get a higher age value and a reduction in usage brings the age closer to zero – finally leading to its exit.

File System Management Approaches

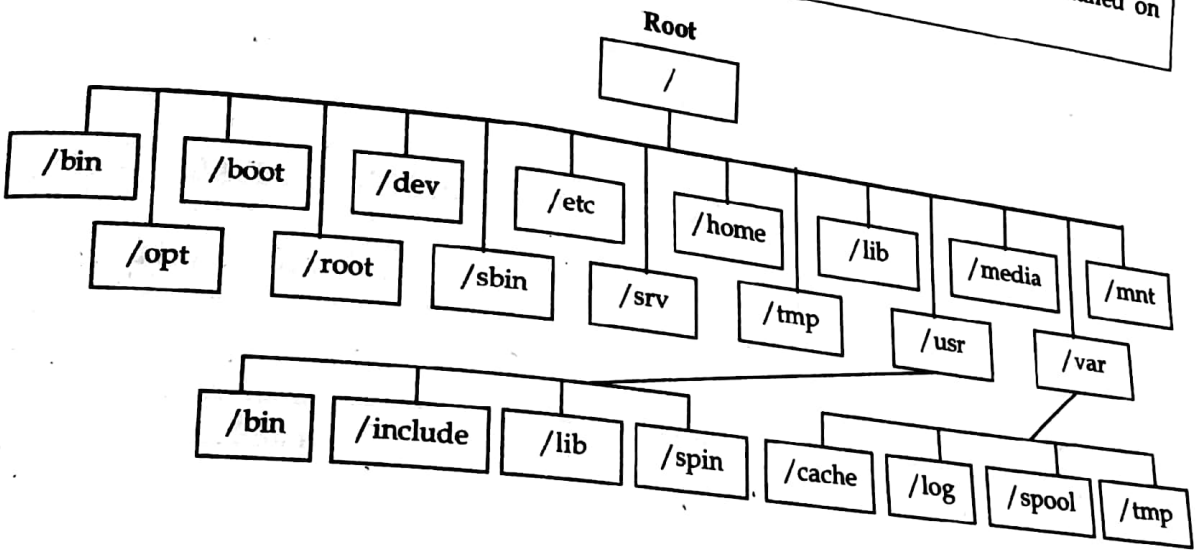
Linux filesystems refer to how Linux-based computers organize, store and track system files. The filesystem is basically a combination of directories or folders that serve as a placeholder for addresses of other files. In other words, there is no distinction between a file and a directory in Linux filessystem, because a directory is considered to be a file containing names of other files. Hence, software programs, services, texts, images, and so forth, are all considered files. In the same way, input and output devices are considered to be files according to the filesystem. Consequently, the filesystem provides the namespace which comprises the naming convention as to how a file can be named in terms of the length and character combinations. The namespace also includes the organizational framework which is the logical structure of the files on disk, often arranged in a hierarchical form using directories.

In Linux, all files and directories are located in a tree-like structure. The topmost directory is referred to as the file system root or just / (not to be confused with the root user). The counterpart of / in a Windows system would probably be C:\. All other directories in Linux can be accessed from the root directory and are arranged in a hierarchical structure.

The following table provides a short overview of the most important higher-level directories on a Linux system.

Directory	Contents
/	Root directory—the starting point of the directory tree.
/bin	Essential binary files, such as commands that are needed by both the system administrator and normal users. Usually also contains the shells, such as Bash.
/boot	Static files of the boot loader.
/dev	Files needed to access host-specific devices.
/etc	Host-specific system configuration files.

/lib	Essential shared libraries and kernel modules.
/media	Mount points for removable media.
/mnt	Mount point for temporarily mounting a file system.
/opt	Add-on application software packages.
/root	Home directory for the superuser root.
/sbin	Essential system binaries.
/srv	Data for services provided by the system.
/tmp	Temporary files.
/usr	Secondary hierarchy with read-only data.
/var	Variable data such as log files
/windows	Only available if you have both Microsoft Windows* and Linux installed on your system. Contains the Windows data.



Exercise

1. What is the name of the method that kernel uses to minimize the frequency of disk access by maintaining a pool of internal data buffer to increase the response time and throughput?
2. What does the following command <mknodmyfifo b 4 16> do?
3. What is the command to count the number of characters in a file?
4. What is Linux?
5. What is the difference between UNIX and LINUX?
6. What is Linux Kernel?
7. What is the advantage of open source?
8. What are the basic components of Linux?
9. Does it help for a Linux system to have multiple desktop environments installed?
10. What is the basic difference between BASH and DOS?

11. Describe the root account.
12. How can you find out how much memory Linux is using?
13. What is a typical size for a swap partition under a Linux system?
14. How do you refer to the parallel port where devices such as printers are connected?
15. How do you change permissions under Linux?
16. What is the maximum length for a filename under Linux?
17. How do you share a program across different virtual desktops under Linux?
18. What are the different modes when using vi editor?
19. Is it possible to use shortcuts for a long pathname?
20. What could be the problem when a command that was issued gave a different result frc the last time it was used?

❑❑❑

KEC's B.Sc. CSIT Text Book Series

First Semester
- Introduction to Informatioin Technology
- C Programming
- Digital Logic
- Mathematics I
- Physics

Second Semester
- Discrete Structure
- Object Oriented Progamming
- Microprocessor
- Mathematics II
- Statistics I

Third Semester
- Data Structure and Algorithms
- Numerical Method
- Computer Architecture
- Computer Graphics
- Statistics II

Fourth Semester
- Theory of Computation
- Computer Networks
- Operating System
- Database Management System
- Artificial Intelligence

Fifth Semester
- Design and Analysis of Algorithms
- System Analysis and Design
- Cryptography
- Simulation and Modeling
- Web Technology
- Elective I

Sixth Semester
- Software Engineering
- Complier Design and Construction
- E-Governance
- NET Centric Computing
- Technical Writing
- Elective II

Seventh Semester
- Advanced Jave Programming
- Data Warehousing and Data Mining
- Principles of Management
- Project Work
- Elective III

Eighth Semester
- Advanced Database
- Internship
- Elective IV
- Elective V

KEC
Publication and Distribution (P.) Ltd.
Kathmandu, Nepal, Tel.: 01-4168301, 01-4241777
E-mail: kecpublication14@gmail.com