Indira Gandhi
National Open University
School of Computer and
Information Sciences

MCS-203
Operating Systems

Block

2

MEMORY MANAGEMENT, FILE MANAGEMENT AND SECURITY

PROGRAMME/COURSE DESIGN COMMITTEE

Prof. Sanjeev K. Aggarwal, IIT, Kanpur
Shri Navneet Aggarwal
Trinity BPM, New Delhi
Prof. M. Balakrishnan, IIT, Delhi
Prof. Pandurangan, C., IIT, Madras
Ms. Bhoomi Gupta
Sirifort College of Computer and Technology Management, New Delhi
Shri Manoj Kumar Gupta
Keane India Ltd., New Delhi
Shri Sachin Gupta
Delhi Institute of Advanced Studies, New Delhi
Prof. Harish Karnick, IIT, Kanpur
Shri Anil Kumar
Amity School of Engineering and Technology New Delhi
Dr. Kapil Kumar, IIMT, Meerut
Dr. Sachin Kumar,
CCS University,
Meerut Ms. Manjulata
Amity School of Engineering and Technology New Delhi
Shri Ajay Rana
Amity School of Computer Sciences, Noida
Dr. Divya Sharma
Bharati College, Delhi
Shri Neeraj Sharma
Havard Institute of Management Technology Noida
Shri Sanjeev Thakur
Amity School of Computer Sciences, Noida
Shri Amrit Nath Thulal
Amity School of Engineering and Technology New Delhi
Dr. Om Vikas(Retd), Ex-Sr. Director, Ministry of ICT, Delhi
Shri Vishwakarma
Amity School of Engineering and Technology New Delhi
Prof (Retd) S. K. Gupta, IIT Delhi
Prof. T.V. Vijaya Kumar, Dean, SC&SS, JNU, New Delhi
Prof. Ela Kumar, Dean, CSE, IGDTUW, Delhi
Prof. Gayatri Dhingra, GVMITM, Sonipat
Sh. Milind Mahajani Vice President, Impressico Business Solutions, Noida, UP
Prof. V. V. Subrahmanyam, Director SOCIS, New Delhi
Prof. P. V. Suresh,
SOCIS, IGNOU, New Delhi
Dr. Shashi Bhushan
SOCIS, IGNOU, New Delhi
Shri Akshay Kumar, Associate Prof.
SOCIS, IGNOU, New Delhi
Shri M. P. Mishra, Associate Prof.
SOCIS, IGNOU, New Delhi
Dr. Sudhansh Sharma, Asst. Prof
SOCIS, IGNOU, New Delhi

BLOCK PREPARATION TEAM

Prof. D.P. Vidyarthi *(Content Editor)*
Jawaharlal Nehru University, New Delhi

Ms. Bhoomi Gupta
Sirifort College of Computer and Technology Management, New Delhi

Shri Sachin Kumar
CCS University, Meerut

Ms. Manjulata
Amity School of Engineering and Technology New Delhi

Prof. V.V. Subrahmanyam
SOCIS, IGNOU

Prof. Nandini Sahu
&
Ms. Malathy. A
Editorial Unit, SOH, IGNOU

Prof. A.K. Verma
Professor & Head (Retired)
Indian Institute of Mass Communication, Delhi

(*Language Editors*)

Course Coordinator: Prof. V. V. Subrahmanyam

(The Units of Block-2, Unit 1 to Unit-4 were adopted from MCS-041 Operating Systems)

PRINT PRODUCTION

Mr. Tilak Raj
Assistant Registrar,
MPDD, IGNOU, New Delhi

July, 2021

ISBN : 978-93-91229-15-3

Further information on the Indira Gandhi National Open University courses may be obtained from the University's office at Maidan Garhi, New Delhi-110068.

Printed and published on behalf of the Indira Gandhi National Open University, New Delhi by the Registrar, MPDD, IGNOU, New Delhi

Laser Typesetting : Akashdeep Printers, 20-Ansari Road, Daryaganj, New Delhi-110002

Printed at : Akashdeep Printers, 20-Ansari Road, Daryaganj, New Delhi-110002

BLOCK INTRODUCTION

The objective of this block is to familiarize you with the issues involved in the memory management, I/O management, File management and Security by the Operating System.

The block is organized into 4 units.

Unit 1 covers the single process monitor, overlays, memory allocation methods, paging and segmentation schemes.

Unit 2 covers the concept of virtual memory, demand paging, demand segmentation and combined systems like segmented paging and paged segmentation.

Unit 3 covers the I/O management and issues related to file management.

Unit 4 covers the very important and critical aspect namely security and protection.

ignou
THE PEOPLE'S
UNIVERSITY

UNIT 1 MEMORY MANAGEMENT

Structure

1.0 INTRODUCTION

In Block 1 we have studied about introductory concepts of the OS, process management and deadlocks. In this unit, we will go through another important function of the Operating System – the memory management.

Memory is central to the operation of a modern computer system. Memory is a large array of words or bytes, each location with its own address. Interaction is achieved through a sequence of reads/writes of specific memory address. The CPU fetches from the program from the hard disk and stores in memory. If a program is to be executed, it must be mapped to absolute addresses and loaded into memory.

In a multiprogramming environment, in order to improve both the CPU utilization and the speed of the computer's response, several processes must be kept in memory. There are many different algorithms depending on the particular situation to manage the memory. Selection of a memory management scheme for a specific system depends upon many factors, but especially upon the hardware design of the system. Each algorithm requires its own hardware support.

The Operating System is responsible for the following activities in connection with memory management:

- Keep track of which parts of memory are currently being used and by whom.
- Decide which processes are to be loaded into memory when memory space becomes available.
- Allocate and deallocate memory space as needed.

In the multiprogramming environment operating system dynamically allocates memory to multiple processes. Thus memory plays a significant role in the important aspects of computer system like performance, S/W support, reliability and stability.

Memory can be broadly classified into two categories–the primary memory (like cache and RAM) and the secondary memory (like magnetic tape, disk etc.). The memory is a resource that needs effective and efficient management. The part of OS that perform this vital task of memory management is known **as memory manager**. In multiprogramming system, as available memory is shared among number of processes, so the allocation speed and the efficient memory utilization (in terms of minimal overheads and reuse/relocation of released memory block) are of prime concern. Protection is difficult to achieve with relocation requirement, as location of process and absolute address in memory is unpredictable. But at run-time, it can be done. Fortunately, we have mechanisms supporting protection like processor (hardware) support that is able to abort the instructions violating protection and trying to interrupt other processes.

This unit collectively depicts such memory management related responsibilities in detail by the OS. Further we will discuss, the basic approaches of allocation are of two types:

Contiguous Memory Allocation: Each programs data and instructions are allocated a single contiguous space in memory.

Non-Contiguous Memory Allocation: Each programs data and instructions are allocated memory space that is not continuous. This unit focuses on contiguous memory allocation scheme.

1.1 OBJECTIVES

After going through this unit, you should be able to:

- describe the various activities handled by the operating system while performing the memory management function;
- to allocate memory to the processes when they need it;
- reallocation when processes are terminated;
- logical and physical memory organization;
- memory protection against unauthorized access and sharing;
- to manage swapping between main memory and disk in case main storage is small to hold all processes;
- to summarize the principles of memory management as applied to paging and segmentation;

- compare and contrast paging and segmentation techniques, and
- analyse the various memory portioning/partitioning techniques including overlays, swapping, and placement and replacement policies.

1.2 OVERLAYS AND SWAPPING

Usually, programs reside on a disk in the form of executable files and for their execution they must be brought into memory and must be placed within a process. Such programs form the ready queue. In general scenario, processes are fetched from ready queue, loaded into memory and then executed. During these stages, addresses may be represented in different ways like in source code addresses or in symbolic form (ex. LABEL). Compiler will bind this symbolic address to relocatable addresses (for example, 16 bytes from base address or start of module). The linkage editor will bind these relocatable addresses to absolute addresses. Before we learn a program in memory we must bind the memory addresses that the program is going to use. Binding is basically assigning which address the code and data are going to occupy. You can bind at compile-time, load-time or execution time.

Compile-time: If memory location is known a priori, absolute code can be generated.

Load-time: If it is not known, it must generate relocatable at complete time.

Execution-time: Binding is delayed until run-time; process can be moved during its execution. We need H/W support for address maps (base and limit registers).

For better memory utilization all modules can be kept on disk in a relocatable format and only main program is loaded into memory and executed. Only on need the other routines are called, loaded and address is updated. Such scheme is called *dynamic loading*, which is user's responsibility rather than OS.But Operating System provides library routines to implement dynamic loading.

In the above discussion we have seen that entire program and its related data is loaded in physical memory for execution. But what if process is larger than the amount of memory allocated to it? We can overcome this problem by adopting a technique called as *Overlays*. Like dynamic loading, overlays can also be implemented by users without OS support. The entire program or application is divided into instructions and data sets such that when one instruction set is needed it is loaded in memory and after its execution is over, the space is released. As and when requirement for other instruction arises it is loaded into space that was released previously by the instructions that are no longer needed. Such instructions can be called as overlays, which are loaded and unloaded by the program.

Definition: An overlay is a part of an application, which has been loaded at same origin where previously some other part(s) of the program was residing.

A program based on overlay scheme mainly consists of following:

- A "root" piece which is always memory resident
- Set of overlays.

Overlay gives the program a way to extend limited main storage. An important aspect related to overlays identification in program is concept of mutual exclusion i.e., routines which do not invoke each other and are not loaded in memory simultaneously.

For example, suppose total available memory is 140K. Consider a program with four subroutines named as: ***Read ( ), Function1( ), Function2( )*** and ***Display( ).*** First, *Read* is invoked that reads a set of data. Based on this data set values, conditionally either one of routine *Function1* or *Function2* is called. And then *Display* is called to output results. Here, *Function1* and *Function2* are mutually exclusive and are not required simultaneously in memory. The memory requirement can be shown as in *Figure 1*:

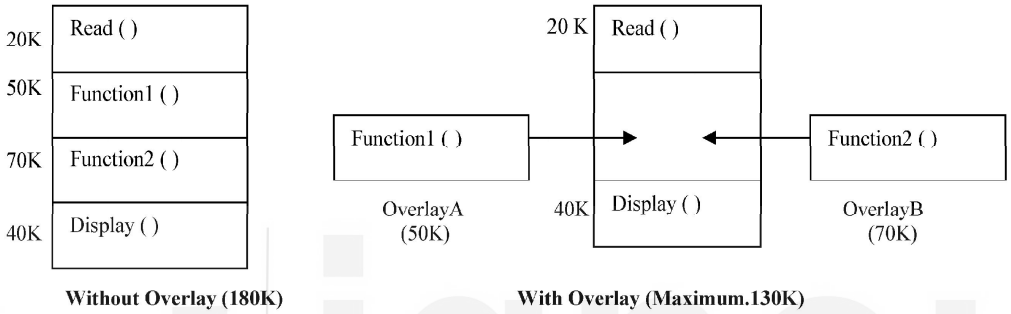


Figure 1: Example of overlay

Without the overlay it requires 180 K of memory and with the overlay support memory requirement is 130K. Overlay manager/driver is responsible for loading and unloading on overlay segment as per requirement. But this scheme suffers from following limitations:

- Require careful and time-consuming planning.
- Programmer is responsible for organising overlay structure of program with the help of file structures etc. and also to ensure that piece of code is already loaded when it is called.
- Operating System provides the facility to load files into overlay region.

Swapping

Swapping is an approach for memory management by bringing each process in entirety, running it and then putting it back on the disk, so that another program may be loaded into that space. Swapping is a technique that lets you use a disk file as an extension of memory. Lower priority user processes are swapped to backing store (disk) when they are waiting for I/O or some other event like arrival of higher priority processes. This is *Rollout Swapping*. Swapping the process back into store when some event occurs or when needed (may be in a different partition) is known as *Roll-in swapping*. *Figure 2* depicts technique of swapping:

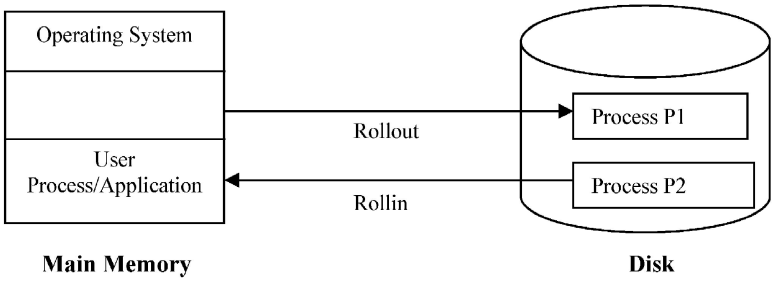


Figure 2: Swapping

Major benefits of using swapping are:

- Allows higher degree of multiprogramming.
- Allows dynamic relocation, i.e., if address binding at execution time is being used we can swap in different location else in case of compile and load time bindings processes have to be moved to same location only.
- Better memory utilization.
- Less wastage of CPU time on compaction, and
- Can easily be applied on priority-based scheduling algorithms to improve performance.

Though swapping has these benefits but it has few limitations also like entire program must be resident in store when it is executing. Also processes with changing memory requirements will need to issue system calls for requesting and releasing memory. It is necessary to know exactly how much memory a user process is using and also that it is blocked or waiting for I/O.

If we assume a data transfer rate of 1 megabyte/sec and access time of 10 milliseconds, then to actually transfer a 100Kbyte process we require:

Transfer Time = 100K / 1,000 = 1/10 seconds

= 100 milliseconds

Access time = 10 milliseconds

Total time = 110 milliseconds

As both the swap out and swap in should take place, the total swap time is then about 220 milliseconds (above time is doubled). A round robin CPU scheduling should have a time slot size much larger relative to swap time of 220 milliseconds. Also if process is not utilising memory space and just waiting for I/O operation or blocked, it should be swapped.

1.3 LOGICAL AND PHYSICAL ADDRESS SPACE

The computer interacts via logical and physical addressing to map memory. Logical address is the one that is generated by CPU, also referred to as virtual address. The program perceives this address space. Physical address is the actual address understood by computer hardware i.e., memory unit. Logical to physical address translation is taken care by the Operating System. The term *virtual memory* refers to the abstraction of separating LOGICAL memory (i.e., memory as seen by the process) from PHYSICAL memory (i.e., memory as seen by the processor). Because of this separation, the programmer needs to be aware of only the logical memory space while the operating system maintains two or more levels of physical memory space.

In compile-time and load-time address binding schemes these two tend to be the same. These differ in execution-time address binding scheme and the MMU (Memory Management Unit) handles translation of these addresses.

Definition: MMU (as shown in the *Figure 3*) is a hardware device that maps logical address to the physical address. It maps the virtual address to the real store location. The simple MMU scheme adds the relocation register contents to the base address of the program that is generated at the time it is sent to memory.

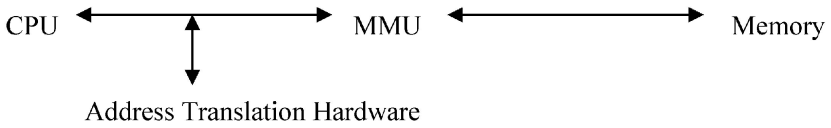


Figure 3: Role of MMU

The entire set of logical addresses forms logical address space and set of all corresponding physical addresses makes physical address space.

1.4 SINGLE PROCESS MONITOR (MONOPROGRAMMING)

In the simplest case of single-user system everything was easy as at a time there was just one process in memory and no address translation was done by the operating system dynamically during execution. Protection of OS (or part of it) can be achieved by keeping it in ROM.We can also have a separate OS address space only accessible in supervisor mode as shown in *Figure 4*.

The user can employ overlays if memory requirement by a program exceeds the size of physical memory. In this approach only one process at a time can be in running state in the system. Example of such system is MS-DOS which is a single tasking system having a command interpreter. Such an arrangement is limited in capability and performance.

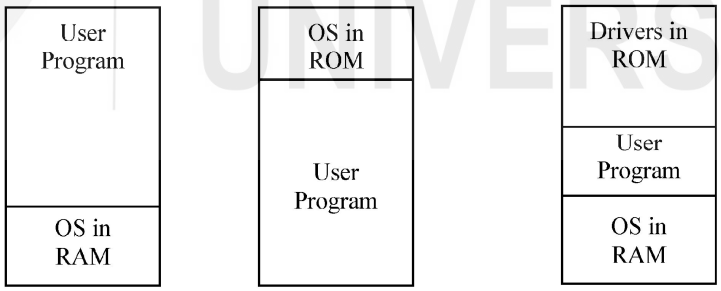


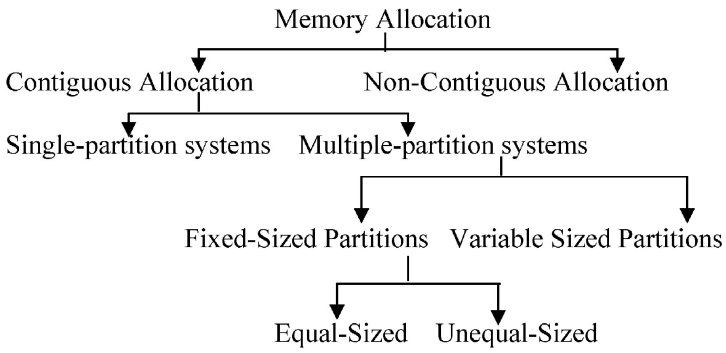
Figure 4: Single Partition System

1.5 CONTIGUOUS ALLOCATION METHODS

In a practical scenario Operating System could be divided into several categories as shown in the hierarchical chart given below:

1) Single process system

2) Multiple process system with two types: Fixed partition memory and variable partition memory.



Further we will learn these schemes in next section.

Partitioned Memory allocation:

The concept of multiprogramming emphasizes on maximizing CPU utilization by overlapping CPU and I/O. Memory may be allocated as:

- Single large partition for processes to use or
- Multiple partitions with a single process using a single partition.

1.5.1 Single-Partition System

This approach keeps the Operating System in the lower part of the memory and other user processes in the upper part. With this scheme, Operating System can be protected from updating in user processes. Relocation-register scheme known as *dynamic relocation* is useful for this purpose. It not only protects user processes from each other but also from changing OS code and data. Two registers are used: relocation register, contains value of the smallest physical address and limit register, contains logical addresses range. Both these are set by Operating System when the job starts. At load time of program (i.e., when it has to be relocated) we must establish "addressability" by adjusting the relocation register contents to the new starting address for the program. The scheme is shown in *Figure 5*.

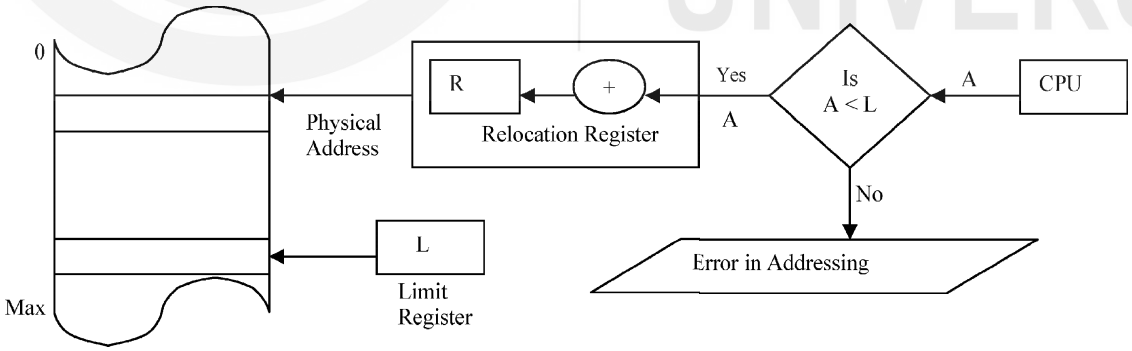


Figure 5: Dynamic Relocation

The contents of a relocation register are implicitly added to any address references generated by the program. Some systems use base registers as relocation register for easy addressability as these are within programmer's control. Also, in some systems, relocation is managed and accessed by Operating System only.

To summarize this, we can say, in dynamic relocation scheme if the logical address space range is 0 to *Max* then physical address space range is R+0 to R+Max (where R is relocation register contents). Similarly, a limit register is checked by

H/W to be sure that logical address generated by CPU is not bigger than size of the program.

1.5.2 Multiple Partition System: Fixed-sized partition

This is also known as *static partitioning* scheme as shown in *Figure 6*. Simple memory management scheme is to divide memory into *n* (possibly unequal) fixed-sized partitions, each of which can hold exactly one process. The degree of multiprogramming is dependent on the number of partitions. IBM used this scheme for systems 360 OS/MFT (Multiprogramming with a fixed number of tasks). The partition boundaries are not movable (must reboot to move a job). We can have one queue per partition or just a single queue for all the partitions.

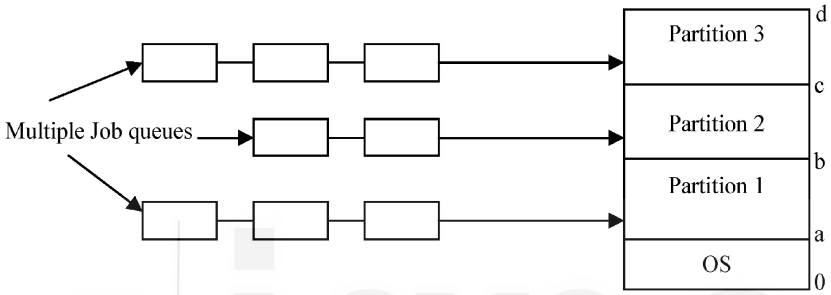


Figure 6: Multiple Partition System

Initially, whole memory is available for user processes and is like a large block of available memory. Operating System keeps details of available memory blocks and occupied blocks in tabular form. OS also keeps track on memory requirements of each process. As processes enter into the input queue and when sufficient space for it is available, process is allocated space and loaded. After its execution is over it releases its occupied space and OS fills this space with other processes in input queue. The block of available memory is known as a *Hole*. Holes of various sizes are scattered throughout the memory. When any process arrives, it is allocated memory from a hole that is large enough to accommodate it. This example is shown in *Figure 7:*

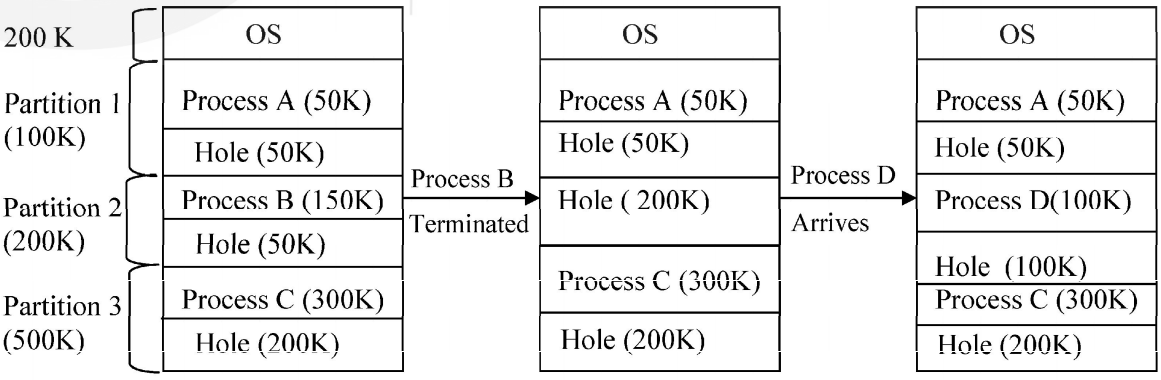


Figure 7: Fixed-sized Partition Scheme

If a hole is too large, it is divided into two parts:

1) One that is allocated to next process of input queue

2) Added with set of holes.

Within a partition if two holes are adjacent then they can be merged to make a single large hole. But this scheme suffers from fragmentation problem. Storage

fragmentation occurs either because the user processes do not completely accommodate the allotted partition or partition remains unused, if it is too small to hold any process from input queue. Main memory utilization is extremely inefficient. Any program, no matter how small, occupies entire partition. In our example, process B takes 150K of partition2 (200K size). We are left with 50K sized hole. This phenomenon, in which there is wasted space internal to a partition, is known as *internal fragmentation*. It occurs because initially process is loaded in partition that is large enough to hold it (i.e., allocated memory may be slightly larger than requested memory). "Internal" here means memory that is internal to a partition, but is not in use.

1.5.3 Variable-sized Partition

This scheme is also known as *dynamic partitioning*. In this scheme, boundaries are not fixed. Processes accommodate memory according to their requirement. There is no wastage as partition size is exactly same as the size of the user process. Initially when processes start this wastage can be avoided but later on when they terminate they leave holes in the main storage. Other processes can accommodate these, but eventually they become too small to accommodate new jobs as shown in *Figure 8*.

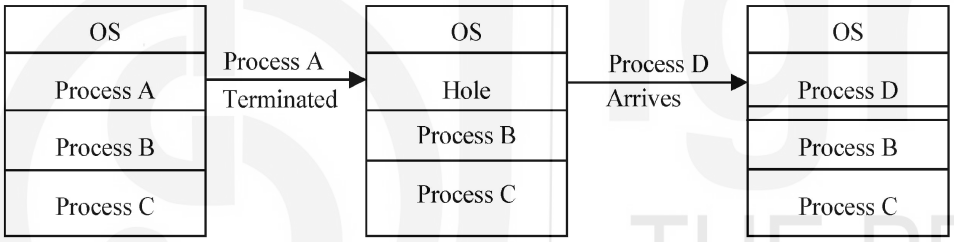


Figure 8: Variable sized partitions

IBM used this technique for OS/MVT (Multiprogramming with a Variable number of Tasks) as the partitions are of variable length and number. But still fragmentation anomaly exists in this scheme. As time goes on and processes are loaded and removed from memory, fragmentation increases and memory utilization declines. This wastage of memory, which is external to partition, is known as *external fragmentation*. In this, though there is enough total memory to satisfy a request but as it is not contiguous and it is fragmented into small holes, that can't be utilized.

External fragmentation problem can be resolved by **coalescing holes** and **storage compaction**. **Coalescing** holes is process of merging existing hole adjacent to a process that will terminate and free its allocated space. Thus, new adjacent holes and existing holes can be viewed as a single large hole and can be efficiently utilized.

There is another possibility that holes are distributed throughout the memory. For utilising such scattered holes, shuffle all occupied areas of memory to one end and leave all free memory space as a single large block, which can further be utilized. This mechanism is known as *Storage Compaction*, as shown in *Figure 9*.

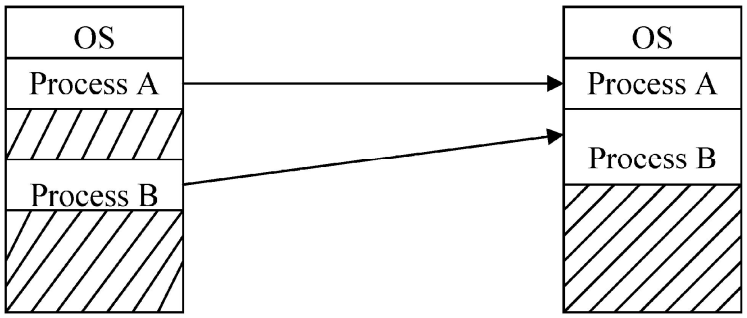


Figure 9: Storage Compaction

But storage compaction also has its limitations as shown below:

1) It requires extra overheads in terms of resource utilization and large response time.

2) Compaction is required frequently because jobs terminate rapidly. This enhances system resource consumption and makes compaction expensive.

3) Compaction is possible only if dynamic relocation is being used (at run-time). This is because the memory contents that are shuffled (i.e., relocated) and executed in new location require all internal addresses to be relocated.

In a multiprogramming system memory is divided into a number of fixed size or variable sized partitions or regions, which are allocated to running processes. For example: a process needs *m* words of memory may run in a partition of *n* words where *n* is greater than or equal to *m*. The variable size partition scheme may result in a situation where available memory is not contiguous, but fragmentation and external fragmentation. The difference (*n*-*m*) is called internal fragmentation, memory which is internal to a partition but is not being use. If a partition is unused and available, but too small to be used by any waiting process, then it is accounted for external fragmentation. These memory fragments cannot be used.

In order to solve this problem, we can either compact the memory making large free memory blocks, or implement paging scheme which allows a program's memory to be noncontiguous, thus permitting a program to be allocated physical memory wherever it is available.

☞ **Check Your Progress 1**

1) What are the four important tasks of a memory manager?

.....

.....

.....

.....

.....

2) What are the three tricks used to resolve absolute addresses?

.....

.....

.....

.....

.....

3) What are the problems that arise with absolute addresses in terms of swapping?

.....

.....

.....

.....

.....

1.6 PAGING

We will see the principles of operation of the paging in the next section.

Paging scheme solves the problem faced in variable sized partitions like external fragmentation.

1.6.1 Principles of Operation

In a paged system, logical memory is divided into a number of fixed sizes 'chunks' called *pages*. The physical memory is also predivided into same fixed sized blocks (as is the size of pages) called *page frames*. The page sizes (also the frame sizes) are always powers of 2, and vary between 512 bytes to 8192 bytes per page. The reason behind this is implementation of paging mechanism using page number and page offset. This is discussed in detail in the following sections:

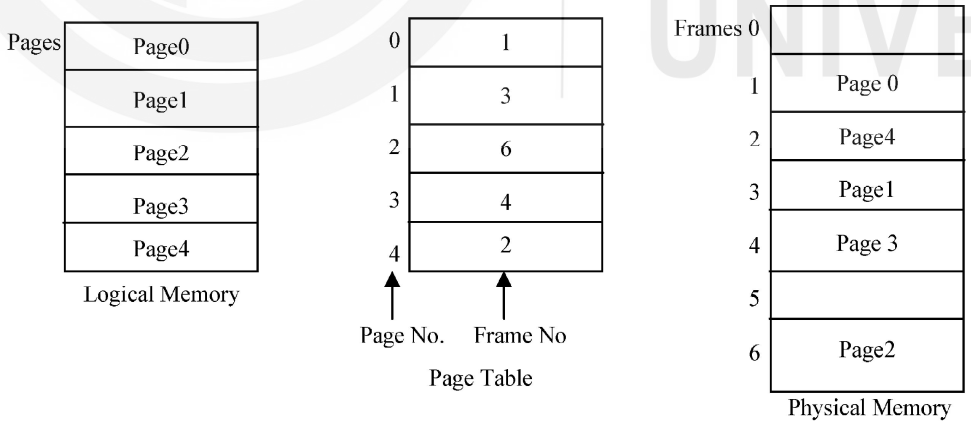


Figure 10: Principle of operation of paging

Each process page is loaded to some memory frame. These pages can be loaded into contiguous frames in memory or into noncontiguous frames also as shown in *Figure 10*. The external fragmentation is alleviated since processes are loaded into separate holes.

1.6.2 Page Allocation

In variable sized partitioning of memory every time when a process of size *n* is to be loaded, it is important to know the best location from the list of available/free

holes. This dynamic storage allocation is necessary to increase efficiency and throughput of system. Most commonly used strategies to make such selection are:

1) **Best-fit Policy:** Allocating the hole in which the process fits most "tightly" i.e., the difference between the hole size and the process size is the minimum one.
2) **First-fit Policy:** Allocating the first available hole (according to memory order), which is big enough to accommodate the new process.
3) **Worst-fit Policy:** Allocating the largest hole that will leave maximum amount of unused space i.e., leftover space is maximum after allocation.

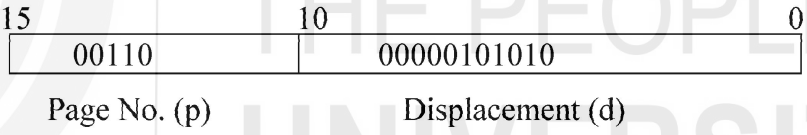
Now, question arises which strategy is likely to be used? In practice, best-fit and first-fit are better than worst-fit. Both these are efficient in terms of time and storage requirement. Best-fit minimize the leftover space, create smallest hole that could be rarely used. First-fit on the other hand requires least overheads in its implementation because of its simplicity. Possibly worst-fit also sometimes leaves large holes that could further be used to accommodate other processes. Thus all these policies have their own merits and demerits.

1.6.3 Hardware Support for Paging

Every logical page in paging scheme is divided into two parts:

1) A page number (p) in logical address space
2) The displacement (or offset) in page p at which item resides (i.e., from start of page).

This is known as Address Translation scheme. For example, a 16-bit address can be divided as given in *Figure* below:



Here, as page number takes 5bits, so range of values is 0 to 31(i.e. 2^5-1). Similarly, offset value uses 11-bits, so range is 0 to 2023(i.e., 2^{11}–1). Summarizing this we can say paging scheme uses 32 pages, each with 2024 locations.

The table, which holds virtual address to physical address translations, is called the **page table**. As displacement is constant, so only translation of virtual page number to physical page is required. This can be seen diagrammatically in *Figure 11.*

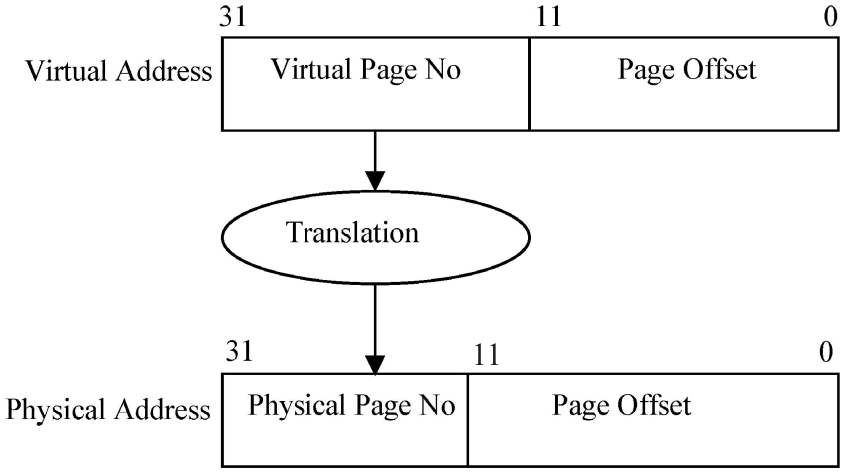


Figure 11: Address Translation scheme

Page number is used as an index into a page table and the latter contains base address of each corresponding physical memory page number (Frame). This reduces dynamic relocation efforts. The Paging hardware support is shown diagrammatically in *Figure 12*:

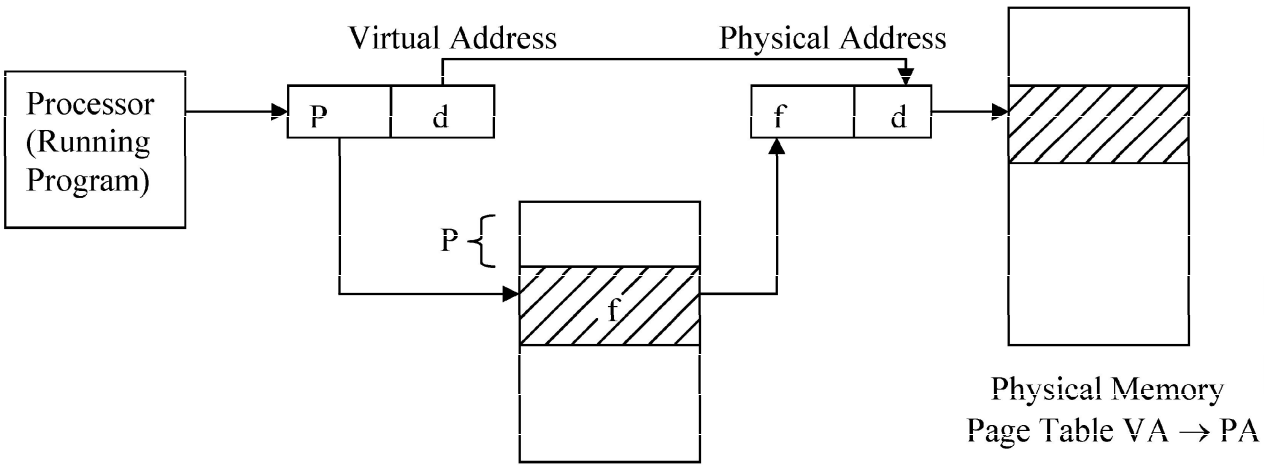


Figure 12: Direct Mapping

Paging address Translation by direct mapping

This is the case of direct mapping as page table sends directly to physical memory page. This is shown in *Figure 12*. But disadvantage of this scheme is its speed of translation. This is because page table is kept in primary storage and its size can be considerably large which increases instruction execution time (also access time) and hence decreases system speed. To overcome this additional hardware support of registers and buffers can be used. This is explained in next section.

Paging Address Translation with Associative Mapping

This scheme is based on the use of dedicated registers with high speed and efficiency. These small, fast-lookup cache help to place the entire page table into a content-addresses associative storage, hence speed-up the lookup problem with a cache. These are known as associative registers or Translation Look-aside Buffers (TLB's). Each register consists of two entries:

1) Key, which is matched with logical page p.

2) Value which returns page frame number corresponding to p.

It is similar to direct mapping scheme but here as TLB's contain only few page table entries, so search is fast. But it is quite expensive due to register support. So, both direct and associative mapping schemes can also be combined to get more benefits. Here, page number is matched with all associative registers simultaneously. The percentage of the number of times the page is found in TLB's is called hit ratio. If it is not found, it is searched in page table and added into TLB. But if TLB is already full then page replacement policies can be used. Entries in TLB can be limited only. This combined scheme is shown in *Figure 13.*

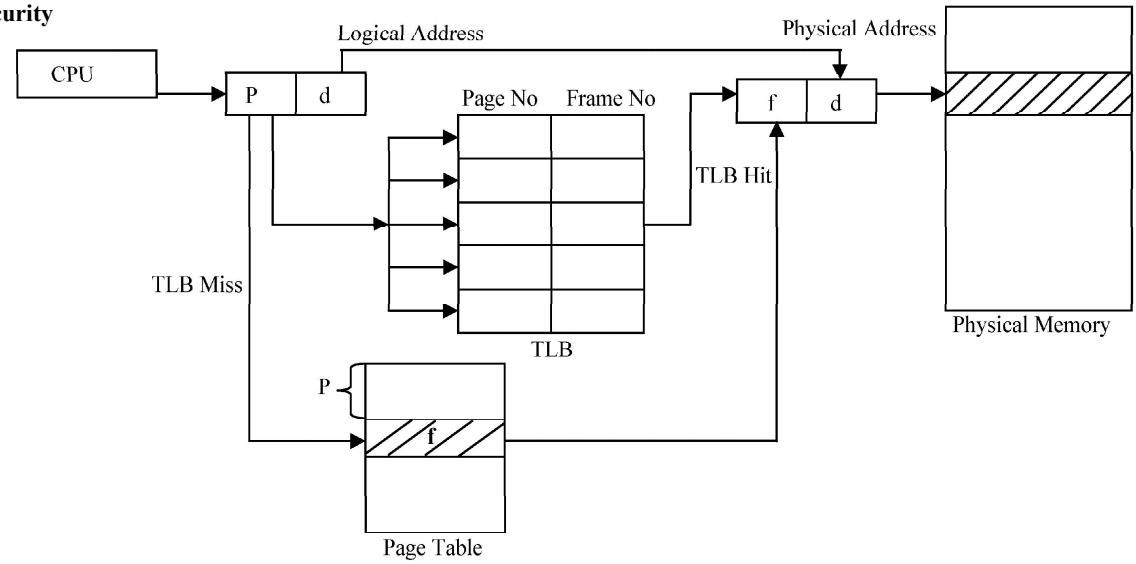


Figure 13: Combined Associative/ Direct Mapping

1.6.4 Protection and Sharing

Paging hardware typically also contains some protection mechanism. In page table corresponding to each frame a protection bit is associated. This bit can tell if page is read-only or read-write. Sharing code and data takes place if two page table entries in different processes point to same physical page, the processes share the memory. If one process writes the data, other process will see the changes. It is a very efficient way to communicate. Sharing must also be controlled to protect modification and accessing data in one process by another process. For this programs are kept separately as procedures and data, where procedures and data that are non-modifiable (pure/reentrant code) can be shared. Reentrant code cannot modify itself and must make sure that it has a separate copy of per-process global variables. Modifiable data and procedures cannot be shared without concurrency controls. Non-modifiable procedures are also known as pure procedures or reentrant codes (can't change during execution). For example, only one copy of editor or compiler code can be kept in memory, and all editor or compiler processes can execute that single copy of the code. This helps memory utilization. Major advantages of paging scheme are:

1) Virtual address space must be greater than main memory size.i.e., can execute program with large logical address space as compared with physical address space.

2) Avoid external fragmentation and hence storage compaction.

3) Full utilization of available main storage.

Disadvantages of paging include internal fragmentation problem i.e., wastage within allocated page when process is smaller than page boundary. Also, extra resource consumption and overheads for paging hardware and virtual address to physical address translation takes place.

1.7 SEGMENTATION

In the earlier section we have seen the memory management scheme called as paging.

In general, a user or a programmer prefers to view system memory as a collection of variable-sized segments rather than as a linear array of words. Segmentation is a memory management scheme that supports this view of memory.

1.7.1 Principles of Operation

Segmentation presents an alternative scheme for memory management. This scheme divides the logical address space into variable length chunks, called segments, with no proper ordering among them. Each segment has a name and a length. For simplicity, segments are referred by a segment number, rather than by a name. Thus, the logical addresses are expressed as a pair of segment number and offset within segment. It allows a program to be broken down into logical parts according to the user view of the memory, which is then mapped into physical memory. Though logical addresses are two-dimensional but actual physical addresses are still one-dimensional array of bytes only.

1.7.2 Address Translation

This mapping between two is done by segment table, which contains segment base and its limit. The segment base has starting physical address of segment, and segment limit provides the length of segment. This scheme is depicted in *Figure 14*.

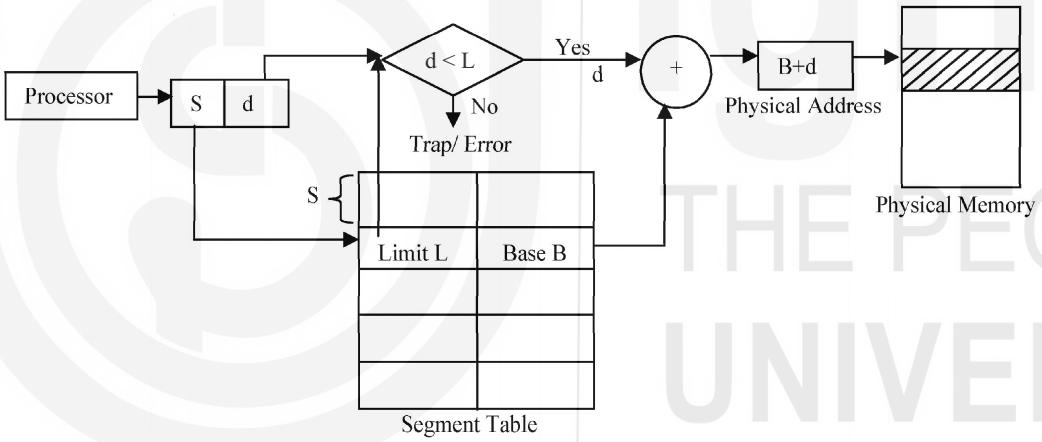


Figure 14: Address Translation

The offset d must range between 0 and segment limit/length, otherwise it will generate address error. For example, consider situation shown in *Figure 15*.

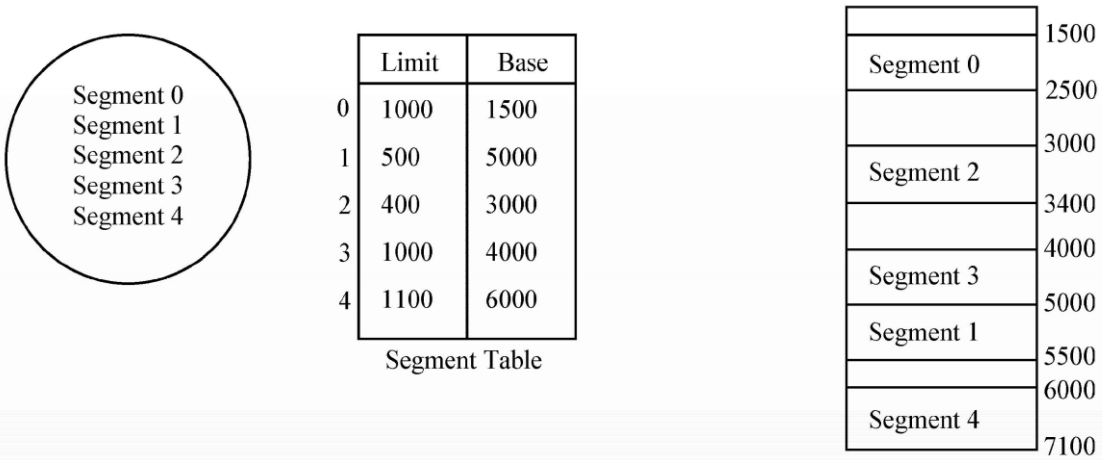


Figure 15: Principle pf operation of representation

This scheme is similar to variable partition allocation method with improvement that the process is divided into parts. For fast retrieval we can use registers as in paged scheme. This is known as a segment-table length register (STLR). The segments in a segmentation scheme correspond to logical divisions of the process and are defined by program names. Extract the segment number and offset from logical address first. Then use segment number as index into segment table to obtain segment base address and its limit /length. Also, check that the offset is not greater than given limit in segment table. Now, general physical address is obtained by adding the offset to the base address.

1.7.3 Protection and Sharing

This method also allows segments that are read-only to be shared, so that two processes can use shared code for better memory efficiency. The implementation is such that no program can read from or write to segments belonging to another program, except the segments that have been set up to be shared. With each segment-table entry protection bit specifying segment as read-only or execute only can be used. Hence illegal attempts to write into a read-only segment can be prevented.

Sharing of segments can be done by making common /same entries in segment tables of two different processes which point to same physical location. Segmentation may suffer from external fragmentation i.e., when blocks of free memory are not enough to accommodate a segment. Storage compaction and coalescing can minimize this drawback.

☞ **Check Your Progress 2**

1) What is the advantage of using Base and Limit registers?

.....

.....

.....

.....

.....

2) How does lookup work with TLB's?

.....

.....

.....

.....

.....

.....

3) Why is page size always powers of 2?

.....

.....

.....

.....

.....

.....

4) A system with 18-bit address uses 6 bits for page number and next 12 bits for offset. Compute the total number of pages and express the following address according to paging scheme 001011(page number) and 000000111000 (offset)?

.....

.....

.....

.....

.....

.....

1.8 SUMMARY

In this unit, we have learnt how memory resource is managed and how processes are protected from each other. The previous two sections covered memory allocation techniques like swapping and overlays, which tackle the utilization of memory. Paging and segmentation was presented as memory management schemes. Both have their own merits and demerits. We have also seen how paging is based on physical form of process and is independent of the programming structures, while segmentation is dependent on logical structure of process as viewed by user. We have also considered fragmentation (internal and external) problems and ways to tackle them to increase level of multiprogramming and system efficiency. Concept of relocation and compaction helps to overcome external fragmentation.

1.9 SOLUTIONS / ANSWERS

Check Your Progress 1

1) i) Keep track of which parts of memory are in use.

   ii) Allocate memory to processes when they require it.

   iii) Protect memory against unauthorised accesses.

   iv) Simulate the appearance of a bigger main memory by moving data automatically between main memory and disk.

2) i) Position Independent Code [PIC]

   ii) Dynamic Relocation

   iii) Base and Limit Registers.

3) When a program is loaded into memory at different locations, the absolute addresses will not work without software or hardware tricks.

Check Your Progress 2

1) These help in dynamic relocation. They make a job easy to move in memory.

2) With TLB support steps determine page number and offset first. Then look up page number in TLB. If it is there, add offset to physical page number and access memory location. Otherwise, trap to OS. OS performs check, looks up physical page number, and loads translation into TLB.Then restart the instruction.

3) Paging is implemented by breaking up an address into a page and offset number. It is efficient to break address into X page bits and Y offset bits, rather than calculating address based on page number and offset. Because each bit position represents a power of 2, splitting an address between bits results in a page size that is a power of 2.

4) Page Number (6 Bits)=001011 = 11(decimal)

   Page Number range= $(2^6–1) = 63$

   Displacement (12 Bits)=000000111000 = 56(decimal)

1.10 FURTHER READINGS

1) H.M.Deitel, *Operating Systems*, published by Pearson Education Asia, New Delhi.

2) Abraham Silberschatz and Peter Baer Galvin, *Operating System Concepts, Wiley and Sons* (Asia) Publications, New Delhi.

3) Andrew S.Tanenbaum, *Operating Systems- Design and Implementation*, published by Prentice Hall of India Pvt. Ltd., New Delhi.

4) Colin Ritchie, *Operating Systems incorporating UNIX and Windows*, BPB Publications, New Delhi.

UNIT 2 VIRTUAL MEMORY

Structure

2.0 INTRODUCTION

In the earlier unit, we have studied Memory Management covering topics like the overlays, contiguous memory allocation, static and dynamic partitioned memory allocation, paging and segmentation techniques. In this unit, we will study an important aspect of memory management known as Virtual memory.

Storage allocation has always been an important consideration in computer programming due to the high cost of the main memory and the relative abundance and lower cost of secondary storage. Program code and data required for execution of a process must reside in the main memory but the main memory may not be large enough to accommodate the needs of an entire process. Early computer programmers divided programs into the sections that were transferred into the main memory for the period of processing time. As the program proceeded, new sections moved into the main memory and replaced sections that were not needed at that time. In this early era of computing, the programmer was responsible for devising this overlay system.

As higher-level languages became popular for writing more complex programs and the programmer became less familiar with the machine, the efficiency of complex programs suffered from poor overlay systems. The problem of storage allocation became more complex.

Two theories for solving the problem of inefficient memory management emerged — static and dynamic allocation. ***Static*** allocation assumes that the availability of memory resources and the memory reference string of a program can be predicted. ***Dynamic*** allocation relies on memory usage increasing and decreasing with actual program needs, not on predicting memory needs.

Program objectives and machine advancements in the 1960s made the predictions required for static allocation difficult, if not impossible. Therefore, the dynamic allocation solution was generally accepted, but opinions about implementation were still divided. One group believed the programmer should continue to be responsible for storage allocation, which would be accomplished by system calls to allocate or deal locate memory. The second group supported **automatic storage allocation** performed by the operating system, because of increasing complexity of storage allocation and emerging importance of multiprogramming. In 1961, two groups proposed a one-level memory store. One proposal called for a very large main memory to alleviate any need for storage allocation. This solution was not possible due to its very high cost. The second proposal is known as virtual memory.

In this unit, we will go through virtual memory and related topics.

2.1 OBJECTIVES

After going through this unit, you should be able to:

- discuss why virtual memory is needed;
- define virtual memory and its underlying concepts;
- describe the page replacement policies like Optimal, FIFO and LRU;
- discuss the concept of thrashing, and
- explain the need of the combined systems like Segmented paging and Paged segmentation.

2.2 VIRTUAL MEMORY

It is common for modern processors to be running multiple processes at one time. Each process has an address space associated with it. To create a whole complete address space for each process would be much too expensive, considering that processes may be created and killed often, and also considering that many processes use only a tiny bit of their possible address space. Last but not the least, even with modern improvements in hardware technology, machine resources are still finite. Thus, it is necessary to share a smaller amount of physical memory among many processes, with each process being given the appearance of having its own exclusive address space.

The most common way of doing this is a technique called virtual memory, which has been known since the 1960s but has become common on computer systems since the late 1980s. The virtual memory scheme divides physical memory into blocks and allocates blocks to different processes. Of course, in order to do this sensibly it is highly desirable to have a protection scheme that restricts a process to be able to access only those blocks that are assigned to it. Such a protection scheme is thus a necessary, and somewhat involved, aspect of any virtual memory implementation.

One other advantage of using virtual memory that may not be immediately apparent is that it often reduces the time taken to launch a program, since not all the program code and data need to be in physical memory before the program execution can be started.

Although sharing the physical address space is a desirable end, it was not the sole reason that virtual memory became common on contemporary systems. Until the late 1980s, if a program became too large to fit in one piece in physical memory, it was the programmer's job to see that it fit. Programmers typically did this by breaking programs into pieces, each of which was mutually exclusive in its logic. When a program was launched, a main piece that initiated the execution would first be loaded into physical memory, and then the other parts, called overlays, would be loaded as needed.

It was the programmer's task to ensure that the program never tried to access more physical memory than was available on the machine, and also to ensure that the proper overlay was loaded into physical memory whenever required. These responsibilities made for complex challenges for programmers, who had to be able to divide their programs into logically separate fragments, and specify a proper scheme to load the right fragment at the right time. Virtual memory came about as a means to relieve programmers creating large pieces of software of the wearisome burden of designing overlays.

Virtual memory automatically manages two levels of the memory hierarchy, representing the main memory and the secondary storage, in a manner that is invisible to the program that is running. The program itself never has to bother with the physical location of any fragment of the virtual address space. A mechanism called relocation allows for the same program to run in any location in physical memory, as well. Prior to the use of virtual memory, it was common for machines to include a relocation register just for that purpose. An expensive and messy solution to the hardware solution of a virtual memory would be software that changed all addresses in a program each time it was run. Such a solution would increase the running times of programs significantly, among other things.

Virtual memory enables a program to ignore the physical location of any desired block of its address space; a process can simply seek to access any block of its address space without concern for where that block might be located. If the block happens to be located in the main memory, access is carried out smoothly and quickly; else, the virtual memory has to bring the block in from secondary storage and allow it to be accessed by the program.

The technique of virtual memory is similar to a degree with the use of processor caches. However, the differences lie in the block size of virtual memory being

typically much larger (64 kilobytes and up) as compared with the typical processor cache (128 bytes and up). The hit time, the miss penalty (the time taken to retrieve an item that is not in the cache or primary storage), and the transfer time are all larger in case of virtual memory. However, the miss rate is typically much smaller. (This is no accident−since a secondary storage device, typically a magnetic storage device with much lower access speeds, has to be read in case of a miss, designers of virtual memory make every effort to reduce the miss rate to a level even much lower than that allowed in processor caches).

Virtual memory systems are of two basic kinds—those using fixed-size blocks called pages, and those that use variable-sized blocks called segments.

Suppose, for example, that a main memory of 64 megabytes is required but only 32 megabytes is actually available. To create the illusion of the larger memory space, the memory manager would divide the required space into units called pages and store the contents of these pages in mass storage. A typical page size is no more than four kilobytes. As different pages are actually required in main memory, the memory manager would exchange them for pages that are no longer required, and thus the other software units could execute as though there were actually 64 megabytes of main memory in the machine.

In brief we can say that virtual memory is a technique that allows the execution of processes that may not be completely in memory. One major advantage of this scheme is that the program can be larger than physical memory. Virtual memory can be implemented via demand paging and demand segmentation.

2.2.1 Principles of Operation

The purpose of virtual memory is to enlarge the address space, the set of addresses a program can utilise. For example, virtual memory might contain twice as many addresses as main memory. A program using all of virtual memory, therefore, would not be able to fit in main memory all at once. Nevertheless, the computer could execute such a program by copying into the main memory those portions of the program needed at any given point during execution.

To facilitate copying virtual memory into real memory, the operating system divides virtual memory into pages, each of which contains a fixed number of addresses. Each page is stored on a disk until it is needed. When the page is needed, the operating system copies it from disk to main memory, translating the virtual addresses into real addresses.

Addresses generated by programs are virtual addresses. The actual memory cells have physical addresses. A piece of hardware called a memory management unit (MMU) translates virtual addresses to physical addresses at run-time. The process of translating virtual addresses into real addresses is called ***mapping***. The copying of virtual pages from disk to main memory is known as ***paging*** or ***swapping***.

Some physical memory is used to keep a list of references to the most recently accessed information on an I/O (input/output) device, such as the hard disk. The optimisation it provides is that it is faster to read the information from physical memory than use the relevant I/O channel to get that information. This is called ***caching***. It is implemented inside the OS.

2.2.2 Virtual Memory Management

This section provides the description of how the virtual memory manager provides virtual memory. It explains how the logical and physical address spaces are mapped to one another and when it is required to use the services provided by the Virtual Memory Manager.

Before going into the details of the management of the virtual memory, let us see the functions of the virtual memory manager. It is responsible to:

- Make portions of the logical address space resident in physical RAM
- Make portions of the logical address space immovable in physical RAM
- Map logical to physical addresses
- Defer execution of application-defined interrupt code until a safe time.

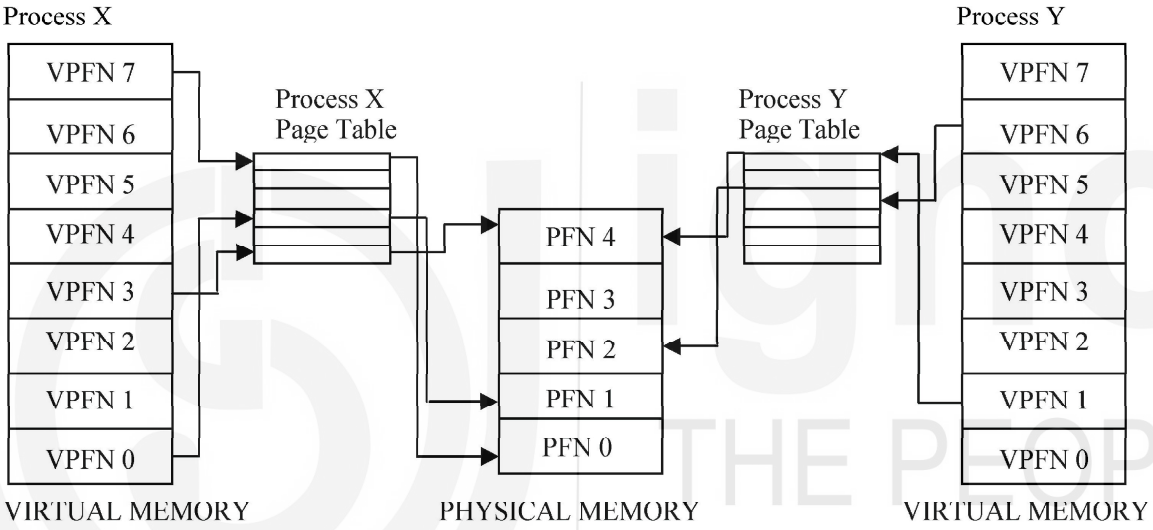


Figure 1: Abstract model of Virtual to Physical address mapping

Before considering the methods that various operating systems use to support virtual memory, it is useful to consider an abstract model that is not cluttered by too much detail.

As the processor executes a program it reads an instruction from memory and decodes it. In decoding the instruction it may need to fetch or store the contents of a location of operands in the memory. The processor then executes the instruction and moves onto the next instruction in the program. In this way the processor is always accessing memory either to fetch instructions or to fetch and store data.

In a virtual memory system all of these addresses are virtual addresses and not physical addresses. These virtual addresses are converted into physical addresses by the processor based on information held in a set of tables maintained by the operating system.

To make this translation easier, virtual and physical memory are divided into small blocks called ***pages***. These pages are all of the same size. (It is not necessary that all the pages should be of same size but if they were not, the system would be very hard to administer). Linux on Alpha AXP systems uses 8 Kbytes pages

and on Intel x86 systems it uses 4 Kbytes pages. Each of these pages is given a unique number; the page frame number (PFN) as shown in the *Figure 1*.

In this paged model, a virtual address is composed of two parts, an offset and a virtual page frame number. If the page size is 4 Kbytes, bits 11:0 of the virtual address contain the offset and bits 12 and above are the virtual page frame number. Each time the processor encounters a virtual address it must extract the offset and the virtual page frame number. The processor must translate the virtual page frame number into a physical one and then access the location at the correct offset into that physical page. To do this the processor uses ***page tables***.

The *Figure1* shows the virtual address spaces of two processes, process *X* and process *Y*, each with their own page tables. These page tables map each processes virtual pages into physical pages in memory. This shows that process *X's* virtual page frame number 0 is mapped into memory in physical page frame number 1 and that process *Y's* virtual page frame number 1 is mapped into physical page frame number 4. Each entry in the theoretical page table contains the following information:

- *Valid flag* : This indicates if this page table entry is valid.
- *PFN* : The physical page frame number that this entry is describing.
- *Access control information* : This describes how the page may be used. Can it be written to? Does it contain executable code?

The page table is accessed using the virtual page frame number as an offset. Virtual page frame 5 would be the 6th element of the table (0 is the first element).

To translate a virtual address into a physical one, the processor must first work out the virtual addresses page frame number and the offset within that virtual page. By making the page size a power of 2 this can be easily done by masking and shifting. Looking again at the *Figure1* and assuming a page size of *0x2000* bytes (which is decimal 8192) and an address of *0x2194* in process *Y's* virtual address space then the processor would translate that address into offset *0x194* into virtual page frame number 1.

The processor uses the virtual page frame number as an index into the processes page table to retrieve its page table entry. If the page table entry at that offset is valid, the processor takes the physical page frame number from this entry. If the entry is invalid, the process has accessed a non-existent area of its virtual memory. In this case, the processor cannot resolve the address and must pass control to the operating system so that it can fix things up.

Just how the processor notifies the operating system that the correct process has attempted to access a virtual address for which there is no valid translation is specific to the processor. However, the processor delivers it, this is known as a ***page fault*** and the operating system is notified of the faulting virtual address and the reason for the page fault. A page fault is serviced in a number of steps:

i) Trap to the OS.

ii) Save registers and process state for the current process.

iii) Check if the trap was caused because of a page fault and whether the page reference is legal.

iv) If yes, determine the location of the required page on the backing store.

v) Find a free frame.

vi) Read the required page from the backing store into the free frame. (During this I/O, the processor may be scheduled to some other process).

vii) When I/O is completed, restore registers and process state for the process which caused the page fault and save state of the currently executing process.

viii) Modify the corresponding PT entry to show that the recently copied page is now in memory.

ix) Resume execution with the instruction that caused the page fault.

Assuming that this is a valid page table entry, the processor takes that physical page frame number and multiplies it by the page size to get the address of the base of the page in physical memory. Finally, the processor adds in the offset to the instruction or data that it needs.

Using the above example again, process *Y's* virtual page frame number 1 is mapped to physical page frame number 4 which starts at *0x8000* (4 x *0x2000*). Adding in the *0x194* byte offset gives us a final physical address of *0x8194*. By mapping virtual to physical addresses this way, the virtual memory can be mapped into the system's physical pages in any order.

2.2.3 Protection and Sharing

Memory protection in a paged environment is realised by protection bits associated with each page, which are normally stored in the page table. Each bit determines a page to be read-only or read/write. At the same time the physical address is calculated with the help of the page table, the protection bits are checked to verify whether the memory access type is correct or not. For example, an attempt to write to a read-only memory page generates a trap to the operating system indicating a memory protection violation.

We can define one more protection bit added to each entry in the page table to determine an invalid program-generated address. This bit is called valid/invalid bit, and is used to generate a trap to the operating system. Valid/invalid bits are also used to allow and disallow access to the corresponding page. This is shown in *Figure 2*.

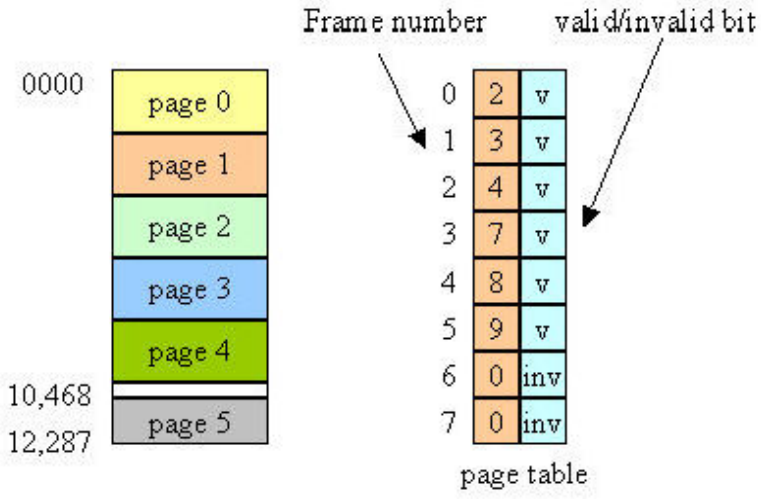


Figure 2: Protection bit in the page table Shared pages

The paging scheme supports the possibility of sharing common program code. For example, a system that supports 40 users, each of them executes a text editor. If the text editor consists of 30 KB of code and 5 KB of data, we need 1400 KB. If the code is reentrant, i.e., it never changes by any write operation during execution (non-self-modifying code) it could be shared as presented in *Figure 3*.

Only one copy of the editor needs to be stored in the physical memory. In each page table, the included editor page is mapped onto the same physical copy of the editor, but the data pages are mapped onto different frames. So, to support 40 users, we only need one copy of the editor, i.e., 30 KB, plus 40 copies of the 5 KB of data pages per user; the total required space is now 230 KB instead of 1400 KB.

Other heavily used programs such as assembler, compiler, database systems etc. can also be shared among different users. The only condition for it is that the code must be reentrant. It is crucial to correct the functionality of shared paging scheme so that the pages are unchanged. If one user wants to change a location, it would be changed for all other users.

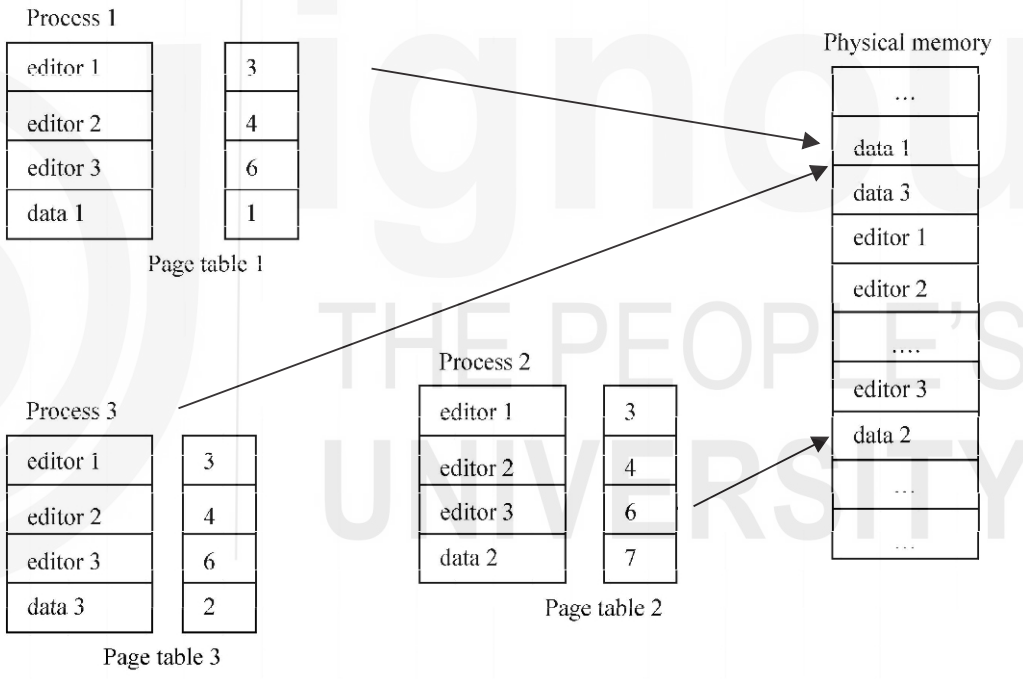


Figure 3: Paging scheme supporting the sharing of program code

2.3 DEMAND PAGING

In a multiprogramming system memory is divided into a number of fixed-size or variable-sized partitions or regions that are allocated to running processes. For example: a process needs *m* words of memory may run in a partition of *n* words where $n \geq m$. The variable size partition scheme may result in a situation where available memory is not contiguous, but fragmented into many scattered blocks. We distinguish between *internal fragmentation* and *external fragmentation*. The difference *(n – m)* is called internal fragmentation, memory that is internal to a partition but is not being used. If a partition is unused and available, but too small to be used by any waiting process, then it is accounted for external fragmentation. These memory fragments cannot be used.

In order to solve this problem, we can either ***compact*** the memory making large free memory blocks, or implement ***paging*** scheme which allows a program's memory to be non-contiguous, thus permitting a program to be allocated to physical memory.

Physical memory is divided into fixed size blocks called ***frames***. Logical memory is also divided into blocks of the same, fixed size called ***pages***. When a program is to be executed, its pages are loaded into any available memory frames from the disk. The disk is also divided into fixed sized blocks that are the same size as the memory frames.

A very important aspect of paging is the clear separation between the user's view of memory and the actual physical memory. Normally, a user believes that memory is one contiguous space containing only his/her program. In fact, the logical memory is scattered through the physical memory that also contains other programs. Thus, the user can work correctly with his/her own view of memory because of the address translation or address mapping. The address mapping, which is controlled by the operating system and transparent to users, translates logical memory addresses into physical addresses.

Because the operating system is managing the memory, it must be sure about the nature of physical memory, for example: which frames are available, which are allocated; how many total frames there are, and so on. All these parameters are kept in a data structure called ***frame table*** that has one entry for each physical frame of memory indicating whether it is free or allocated, and if allocated, to which page of which process.

As there is much less physical memory than virtual memory the operating system must be careful that it does not use the physical memory inefficiently. One way to save physical memory is to load only virtual pages that are currently being used by the executing program. For example, a database program may be run to query a database. In this case not the entire database needs to be loaded into memory, just those data records that are being examined. Also, if the database query is a search query then it is not necessary to load the code from the database that deals with adding new records. This technique of only loading virtual pages into memory as they are accessed is known as ***demand paging***.

When a process attempts to access a virtual address that is not currently in memory the CPU cannot find a page table entry for the virtual page referenced. For example, in *Figure 1*, there is no entry in *Process X's* page table for virtual PFN 2 and so if *Process X* attempts to read from an address within virtual PFN 2 the CPU cannot translate the address into a physical one. At this point the CPU cannot cope and needs the operating system to fix things up. It notifies the operating system that a ***page fault*** has occurred and the operating system makes the process wait whilst it fixes things up. The CPU must bring the appropriate page into memory from the image on disk. Disk access takes a long time, and so the process must wait quite a while until the page has been fetched. If there are other processes that could run then the operating system will select one of them to run. The fetched page is written into a free physical page frame and an entry for the virtual PFN is added to the processes page table. The process is then restarted at the point where the memory fault occurred. This time the virtual memory access is made, the CPU can make the address translation and so the process continues to

run. This is known as demand paging and occurs when the system is busy but also when an image is first loaded into memory. This mechanism means that a process can execute an image that only partially resides in physical memory at any one time.

The valid/invalid bit of the page table entry for a page, which is swapped in, is set as valid. Otherwise it is set as invalid, which will have no effect as long as the program never attempts to access this page. If all and only those pages actually needed are swapped in, the process will execute exactly as if all pages were brought in.

If the process tries to access a page, which was not swapped in, i.e., the valid/ invalid bit of this page table, entry is set to invalid, then a page fault trap will occur. Instead of showing the "invalid address error" as usually, it indicates the operating system's failure to bring a valid part of the program into memory at the right time in order to minimize swapping overhead.

In order to continue the execution of process, the operating system schedules a disk read operation to bring the desired page into a newly allocated frame. After that, the corresponding page table entry will be modified to indicate that the page is now in memory. Because the state (program counter, registers etc.) of the interrupted process was saved when the page fault trap occurred, the interrupted process can be restarted at the same place and state. As shown, it is possible to execute programs even though parts of it are not (yet) in memory.

In the extreme case, a process without pages in memory could be executed. Page fault trap would occur with the first instruction. After this page was brought into memory, the process would continue to execute. In this way, page fault trap would occur further until every page that is needed was in memory. This kind of paging is called ***pure demand paging***. Pure demand paging says that "never bring a page into memory until it is required".

2.4 PAGE REPLACEMENT POLICIES

Basic to the implementation of virtual memory is the concept of ***demand paging***. This means that the operating system, and not the programmer, controls the swapping of pages in and out of main memory, as the active processes require them. When a process needs a non-resident page, the operating system must decide which resident page is to be replaced by the requested page. The part of the virtual memory which makes this decision is called the ***replacement policy***.

There are many approaches to the problem of deciding which page is to replace but the object is the same for all-the policy that selects the page that will not be referenced again for the longest time. A few page replacement policies are described below.

2.4.1 First In First Out (FIFO)

The First In First Out (FIFO) replacement policy chooses the page that has been in the memory the longest to be the one replaced.

Belady's Anomaly

Normally, as the number of page frames increases, the number of page faults should decrease. However, for FIFO there are cases where this generalisation will fail! This is called Belady's Anomaly. Notice that OPT's never suffers from Belady's anomaly.

2.4.2 Second Chance (SC)

The Second Chance (SC) policy is a slight modification of FIFO in order to avoid the problem of replacing a heavily used page. In this policy, a reference bit *R* is used to keep track of pages that have been recently referenced. This bit is set to 1 each time the page is referenced. Periodically, all the reference bits are set to 0 by the operating system to distinguish pages that have not been referenced recently from those that have been. Using this bit, the operating system can determine whether old pages are still being used (i.e., $R = 1$). If so, the page is moved to the end of the list of pages, and its load time is updated as though it had just arrived in memory. Then the search continues. Thus heavily accessed pages are given a "second chance."

2.4.3 Least Recently Used (LRU)

The Least Recently Used (LRU) replacement policy chooses to replace the page which has not been referenced for the longest time. This policy assumes the recent past will approximate the immediate future. The operating system keeps track of when each page was referenced by recording the time of reference or by maintaining a stack of references.

2.4.4 Optimal Algorithm (OPT)

Optimal algorithm is defined as replace the page that will not be used for the longest period of time. It is optimal in the performance but not feasible to implement because we cannot predict future time.

Example:

Let us consider a 4 frames example

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

1	1	1	1	1	4
	2	2	2	2	2
		3	3	3	3
			4	5	5

In the above *Figure*, we have 6 page faults.

2.4.5 Least Frequently Used (LFU)

The Least Frequently Used (LFU) replacement policy selects a page for replacement if the page had not been used often in the past. This policy keeps count of the number of times that a page is accessed. Pages with the

lowest counts are replaced while pages with higher counts remain in primary memory.

2.5 THRASHING

Thrashing occurs when a system spends more time processing page faults than executing transactions. While processing page faults it is necessary to be in order to appreciate the benefits of virtual memory, thrashing has a negative effect on the system.

As the page fault rate increases, more transactions need processing from the paging device. The queue at the paging device increases, resulting in increased service time for a page fault. While the transactions in the system are waiting for the paging device, CPU utilization, system throughput and system response time decrease, resulting in below optimal performance of a system.

Thrashing becomes a greater threat as the degree of multiprogramming of the system increases.

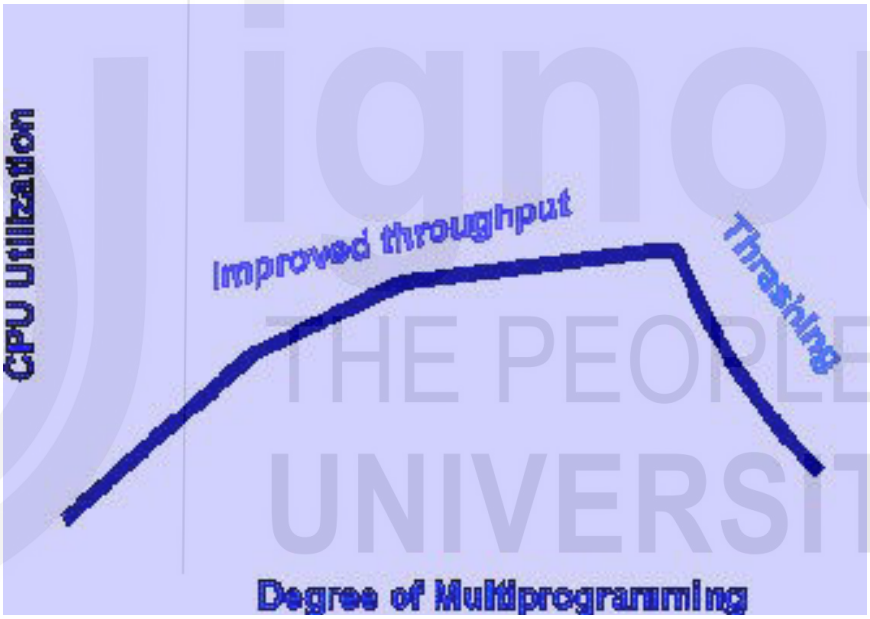


Figure 4: Degree of Multiprogramming

The graph in *Figure 4* shows that there is a degree of multiprogramming that is optimal for system performance. CPU utilization reaches a maximum before a swift decline as the degree of multiprogramming increases and thrashing occurs in the over-extended system. This indicates that controlling the load on the system is important to avoid thrashing. In the system represented by the graph, it is important to maintain the multiprogramming degree that corresponds to the peak of the graph.

The selection of a replacement policy to implement virtual memory plays an important part in the elimination of the potential for thrashing. A policy based on the local mode will tend to limit the effect of thrashing. In local mode, a transaction will replace pages from its assigned partition. Its need to access memory will not affect transactions using other partitions. If other transactions have enough page frames in the partitions they occupy, they will continue to be processed efficiently.

A replacement policy based on the global mode is more likely to cause thrashing. Since all pages of memory are available to all transactions, a memory-intensive transaction may occupy a large portion of memory, making other transactions susceptible to page faults and resulting in a system that thrashes. To prevent thrashing, we must provide a processes as many frames as it needs. There are two techniques for this – *Working-Set Model and Page-Fault Rate.*

2.5.1 Working-Set Model

Principle of Locality

Pages are not accessed randomly. At each instant of execution a program tends to use only a small set of pages. As the pages in the set change, the program is said to move from one phase to another. The principle of locality states that most references will be to the current small set of pages in use. The examples are shown below:

Examples:

1) Instructions are fetched sequentially (except for branches) from the same page.

2) Array processing usually proceeds sequentially through the array functions repeatedly, access variables in the top stack frame.

Ramification

If we have locality, we are unlikely to continually suffer page-faults. If a page consists of 1000 instructions in self-contained loop, we will only fault once (at most) to fetch all 1000 instructions.

Working Set Definition

The working set model is based on the assumption of locality. The idea is to examine the most recent page references in the working set. If a page is in active use, it will be in the Working-set. If it is no longer being used, it will drop from the working set.

The set of pages currently needed by a process is its working set.

WS(k) for a process P is the number of pages needed to satisfy the last k page references for process P.

WS(t) is the number of pages needed to satisfy a process's page references for the last *t* units of time.

Either can be used to capture the notion of locality.

Working Set Policy

Restrict the number of processes on the ready queue so that physical memory can accommodate the working sets of all ready processes. Monitor the working sets of ready processes and, when necessary, reduce multiprogramming (i.e. swap) to avoid thrashing.

Note: Exact computation of the working set of each process is difficult, but it can be estimated, by using the reference bits maintained by the hardware to implement an aging algorithm for pages.

When loading a process for execution, pre-load certain pages. This prevents a process from having to "fault into" its working set. May be only a rough guess at start-up, but can be quite accurate on swap-in.

2.5.2 Page-Fault Rate

The ***working-set model*** is successful, and knowledge of the working set can be useful for pre-paging, but it is a scattered way to control thrashing. A page-fault frequency (page-fault rate) takes a more direct approach. In this we establish upper and lower bound on the desired page- fault rate. If the actual page fault rate exceeds the upper limit, we allocate the process another frame. If the page fault rate falls below the lower limit, we remove a Frame from the process. Thus, we can directly measure and control the page fault rate to prevent thrashing.

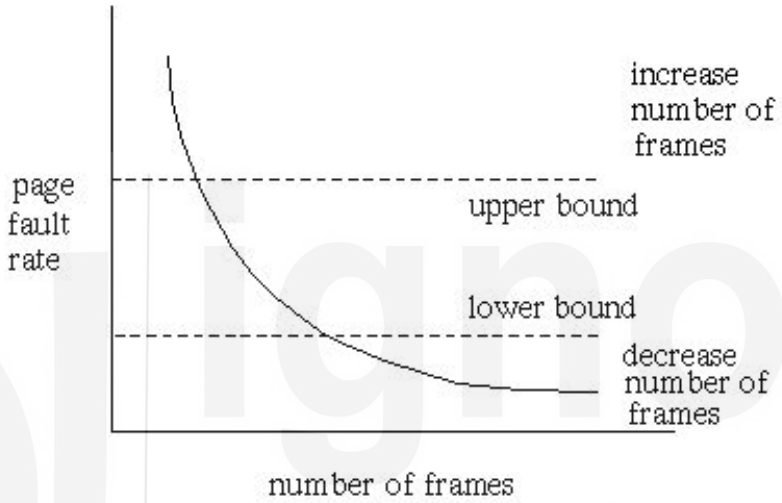


Figure 5: Page-fault frequency

Establish "acceptable" page-fault rate.

- If actual rate too low, process loses frame.
- If actual rate too high, process gains frame.

2.6 DEMAND SEGMENTATION

Segmentation

Programs generally divide up their memory usage by function. Some memory holds instructions, some static data, some dynamically allocated data, some execution frames. All of these memory types have different protection, growth, and sharing requirements. In the monolithic memory allocation of classic Virtual Memory systems, this model isn't well supported.

Segmentation addresses this by providing multiple sharable, protectable, growable address spaces that processes can access.

In pure segmentation architecture, segments are allocated like variable partitions, although the memory management hardware is involved in decoding addresses. Pure segmentation addresses replace the page identifier in the virtual address with a segment identifier, and find the proper segment (not page) to which to apply the offset.

The segment table is managed like the page table, except that segments explicitly allow sharing. Protections reside in the segment descriptors, and can use keying or explicit access control lists to apply them.

Of course, the segment name space must be carefully managed, and thus OS must provide a method of doing this. The file system can come to the rescue here−a process can ask for a file to be mapped into a segment and have the OS return the segment register to use. This is known as memory mapping files. It is slightly different from memory mapping devices, because one file system abstraction (a segment) is providing an interface to another (a file). Memory mapped files may be reflected into the file system or not and may be shared or not at the process's discretion.

The biggest problem with segmentation is the same as with variable sized real memory allocation: managing variable sized partitions can be very inefficient, especially when the segments are large compared to physical memory. External fragmentation can easily result in expensive compaction when a large segment is loaded, and swapping large segments (even when compaction is not required) can be costly.

Demand Segmentation

Same idea as demand paging applied to segments.

If a segment is loaded, base and limit are stored in the Segment Table Entry (STE) and the valid bit is set in the Page Table Entry (PTE). The PTE is accessed for each memory reference. If the segment is not loaded, the valid bit is unset. The base and limit as well as the disk address of the segment is stored in the OS table. Reference to a non-loaded segment generates a segment fault (analogous to page fault). To load a segment, we must solve both the placement question and the replacement question (for demand paging, there is no placement question).

2.7 COMBINED SYSTEMS

The combined systems are of two types. They are:

- Segmented Paging
- Paged Segmentation

2.7.1 Segmented Paging

In a pure paging scheme, the user thinks in terms of a contiguous linear address space, and internal fragmentation is a problem when portions of that address space include conservatively large estimates of the size of dynamic structures. Segmented paging is a scheme in which logically distinct (e.g., dynamically-sized) portions of the address space are deliberately given virtual addresses a LONG way apart, so we never have to worry about things bumping into each other, and the page table is implemented in such a way that the big unused sections don't cost us much. Basically the only page table organization that doesn't work well with segmented paging is a single linear array. Trees, inverted (hash) tables,

and linked lists work fine. If we always start segments at multiples of some large power of two, we can think of the high-order bits of the virtual address as specifying a segment, and the low-order bits as specifying an offset within the segment. If we have tree-structured page tables in which we use k bits to select a child of the root, and we always start segments at some multiple of 2^(word size-k), then the top level of the tree looks very much like a segment table. The only difference is that its entries are selected by the high-order address bits, rather than by some explicit architecturally visible mechanism like segment registers. Basically all modern operating systems on page-based machines use segmented paging.

2.7.2 Paged Segmentation

In a pure segmentation scheme, we still have to do dynamic space management to allocate physical memory to segments. This leads to external fragmentation and forces us to think about things like compaction. It also doesn't lend itself to virtual memory. To address these problems, we can page the segments of a segmented machine. This is paged segmentation. Instead of containing base/bound pairs, the segment table entries of a machine with paged segmentation indicate how to find the page table for the segment. In MULTICS, there was a separate page table for each segment. The segment offset was interpreted as consisting of a page number and a page offset. The base address in the segment table entry is added to the segment offset to produce a "linear address" that is then partitioned into a page number and page offset, and looked up in the page table in the normal way. Note that in a machine with pure segmentation, given a fast way to find base/bound pairs (e.g., segment registers), there is no need for a TLB. Once we go to paged segmentation, we need a TLB. The difference between segmented paging and paged segmentation lies in the user's programming model, and in the addressing modes of the CPU. On a segmented architecture, the user generally specifies addresses using an effective address that includes a segment register specification. On a paged architecture, there are no segment registers. In practical terms, managing segment registers (loading them with appropriate values at appropriate times) is a bit of a nuisance to the assembly language programmer or compiler writer. On the other hand, since it only takes a few bits to indicate a segment register, while the base address in the segment table entry can have many bits, segments provide a means of expanding the virtual address space beyond 2^(word size). We can't do segmented paging on a machine with 16-bit addresses. It's beginning to get problematical on machines with 32-bit addresses. We certainly can't build a MULTICS-style single level store, in which every file is a segment that can be accessed with ordinary loads and stores, on a 32-bit machine. Segmented architectures provide a way to get the effect we want (lots of logically separate segments that can grow without practical bound) without requiring that we buy into very large addresses. As 64-bit architectures become more common, it is possible that segmentation will become less popular. One might think that paged segmentation has an additional advantage over segmented paging: protection information is logically associated with a segment, and could perhaps be specified in the segment table and then left out of the page table. Unfortunately, protection bits are used for lots of purposes other than simply making sure we cannot write your code or execute your data.

☞ **Check Your Progress 1**

1) What are the steps that are followed by the Operating system in order to handle the page fault?

.....

.....

2) What is demand paging?

.....

.....

3) How can you implement the virtual memory?

.....

.....

4) When do the following occurs?

   i) A Page Fault

   ii) Thrashing

.....

.....

5) What should be the features of the page swap algorithm?

.....

.....

6) What is a working set and what happens when the working set is very different from the set of pages that are physically resident in memory?

.....

.....

2.8 SUMMARY

With previous schemes, all the code and data of a program have to be in main memory when the program is running. With virtual memory, only some of the code and data have to be in main memory: the parts needed by the program now. The other parts are loaded into memory when the program needs them without the program having to be aware of this. The size of a program (including its data) can thus exceed the amount of available main memory.

There are two main approaches to virtual memory: paging and segmentation. Both approaches rely on the separation of the concepts virtual address and physical address. Addresses generated by programs are virtual addresses. The actual memory cells have physical addresses. A piece of hardware called a memory

management unit (MMU) translates virtual addresses to physical addresses at run-time.

In this unit we have discussed the concept of Virtual memory, its advantages, demand paging, demand segmentation, Page replacement algorithms and combined systems.

2.9 SOLUTIONS / ANSWERS

Check Your Progress 1

1) The list of steps that are followed by the operating system in handling a page fault:

   a) If a process refers to a page which is not in the physical memory then an internal table kept with a process control block is checked to verify whether a memory reference to a page was valid or invalid.

   b) If the memory reference to a page was valid, but the page is missing, the process of bringing a page into the physical memory starts.

   c) Free memory location is identified to bring a missing page.

   d) By reading a disk, the desired page is brought back into the free memory location.

   e) Once the page is in the physical memory, the internal table kept with the process and page map table is updated to indicate that the page is now in memory.

   f) Restart the instruction that was interrupted due to the missing page.

2) In demand paging pages are loaded only on demand, not in advance. It is similar to paging system with swapping feature. Rather than swapping the entire program in memory, only those pages are swapped which are required currently by the system.

3) Virtual memory can be implemented as an extension of paged or segmented memory management scheme, also called **demand** paging or **demand segmentation** respectively.

4) A Page Fault occurs when a process accesses an address in a page which is not currently resident in memory. Thrashing occurs when the incidence of page faults becomes so high that the system spends all its time swapping pages rather than doing useful work. This happens if the number of page frames allocated is insufficient to contain the working set for the process either because there is inadequate physical memory or because the program is very badly organised.

5) A page swap algorithm should

   a) Impose the minimum overhead in operation

   b) Not assume any particular hardware assistance

   c) Try to prevent swapping out pages which are currently referenced (since they may still be in the Working Set)

d) Try to avoid swapping modified pages (to avoid the overhead of rewriting pages from memory to swap area).

6) The working Set is the set of pages that a process accesses over any given (short) space of time. If the Working Set is very different from the set of pages that are physically resident in memory, there will be an excessive number of page faults and thrashing will occur.

2.10 FURTHER READINGS

1) Abraham Silberschatz and Peter Baer Galvin, *Operating System Concepts*, Wiley and Sons (Asia) Publications, New Delhi.

2) H.M.Deitel, *Operating Systems*, Pearson Education Asia, New Delhi.

3) Andrew S.Tanenbaum, *Operating Systems- Design and Implementation*, Prentice Hall of India Pvt. Ltd., New Delhi.

4) Achyut S. Godbole, *Operating Systems*, Second Edition, TMGH, 2005, New Delhi.

5) Milan Milenkovic, *Operating Systems*, TMGH, 2000, New Delhi.

6) D. M. Dhamdhere, *Operating Systems – A Concept Based Approach*, TMGH, 2002, New Delhi.

7) William Stallings, *Operating Systems*, Pearson Education, 2003, New Delhi.

UNIT 3 I/O AND FILE MANAGEMENT

Structure

3.0 INTRODUCTION

Input and output devices are components that form part of the computer system. These devices are controlled by the operating system. Input devices such as keyboard, mouse, and sensors provide input signals such as commands to the operating system. These commands received from input devices instruct the operating system to perform some task or control its behaviour. Output devices such as monitors, printers and speakers are the devices that receive commands or information from the operating system.

In the earlier unit, we had studied the memory management of primary memory. The physical memory, as we have already seen, is not large enough to accommodate all of the needs of a computer system. Also, it is not permanent.

Secondary storage consists of disk units and tape drives onto which data can be moved for permanent storage. Though there are actual physical differences between tapes and disks, the principles involved in controlling them are the same, so we shall only consider disk management here in this unit.

The operating system implements the abstract concept of the file by managing mass storage devices, such as types and disks. For convenient use of the computer system, the operating system provides a uniform logical view of information storage. The operating system abstracts from the physical properties of its storage devices to define a logical storage Unit, the **file**. Files are mapped by the operating system, onto physical devices.

Definition: A file is a collection of related information defined by its creator. Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic or alphanumeric. Files may be free-form, such as text files, or may be rigidly formatted. In general, a file is a sequence of bits, bytes, lines or records whose meaning is defined by its creator and user.

File management is one of the most visible services of an operating system. Computers can store information in several different physical forms among which magnetic tape, disk, and drum are the most common forms. Each of these devices has their own characteristics and physical organization.

Normally files are organized into directories to ease their use. When multiple users have access to files, it may be desirable to control by whom and in what ways files may be accessed. The operating system is responsible for the following activities in connection with file management:

- The creation and deletion of files.
- The creation and deletion of directory.
- The support of primitives for manipulating files and directories.
- The mapping of files onto disk storage.
- Backup of files on stable (non volatile) storage.

The most significant problem in I/O system is the speed mismatch between I/O devices and the memory and also with the processor. This is because I/O system involves both H/W and S/W support and there is large variation in the nature of I/O devices, so they cannot compete with the speed of the processor and memory.

A well-designed file management structure makes the file access quick and easily movable to a new machine. Also it facilitates sharing of files and protection of non-public files. For security and privacy, file system may also provide encryption and decryption capabilities. This makes information accessible to the intended user only.

In this unit we will study the I/O and the file management techniques used by the operating system in order to manage them efficiently.

3.1 OBJECTIVES

After going through this unit, you should be able to:

- describe the management of the I/O activities independently and simultaneously with processor activities;
- summarize the full range of views that support file systems, especially the operating system view;
- compare and contrast different approaches to file organisations;
- discuss the disk scheduling techniques, and
- know how to implement the file system and its protection against unauthorized usage.

3.2 ORGANISATION OF THE I/O FUNCTION

The range of I/O devices and the large variation in their nature, speed, design, functioning, usage etc. makes it difficult for the operating system to handle them with any generality. The key concept in I/O software designing is device independence achieved by using uniform naming.

The name of the file or device should simply be a string or an integer and not dependent on the device in any way. Another key issue is sharable versus dedicated devices. Many users can share some I/O devices such as disks, at any instance of time. The devices like printers have to be dedicated to a single user until that user has finished an operation.

The basic idea is to organise the I/O software as a series of layers with the lower ones hiding the physical H/W and other complexities from the upper ones that present simple, regular interface interaction with users. Based on this I/O software can be structured in the following four layers given below with brief descriptions:

(i) Interrupt handlers: The CPU starts the transfer and goes off to do something else until the interrupt arrives. The I/O device performs its activity independently and simultaneously with CPU activity. This enables the I/O devices to run asynchronously with the processor. The device sends an interrupt to the processor when it has completed the task, enabling CPU to initiate a further task. These interrupts can be hidden with the help of device drivers discussed below as the next I/O software layer.

(ii) Device Drivers: Each device driver handles one device type or group of closely related devices and contains a device dependent code. It accepts individual requests from device independent software and checks that the request is carried out. A device driver manages communication with a specific I/O device by converting a logical request from a user into specific commands directed to the device.

(iii) Device-independent Operating System Software: Its basic responsibility is to perform the I/O functions common to all devices and to provide interface with user-level software. It also takes care of mapping symbolic device names onto the proper driver. This layer supports device naming, device protection, error reporting, allocating and releasing dedicated devices etc.

(iv) User level software: It consists of library procedures linked together with user programs. These libraries make system calls. It makes I/O call, format I/

O and also support spooling. Spooling is a way of dealing with dedicated I/O devices like printers in a multiprogramming environment.

3.3 I/O BUFFERING

A buffer is an intermediate memory area under operating system control that stores data in transit between two devices or between user's work area and device. It allows computation to proceed in parallel with I/O.

In a typical unbuffered transfer situation the processor is idle for most of the time, waiting for data transfer to complete and total read-processing time is the sum of all the transfer/read time and processor time as shown in *Figure 1*.

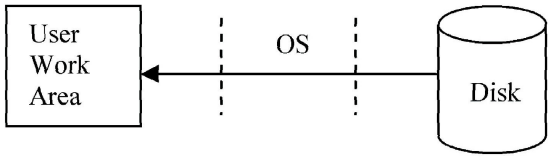


Figure 1: Unbuffered Transfers

In case of single-buffered transfer, blocks are first read into a buffer and then moved to the user's work area. When the move is complete, the next block is read into the buffer and processed in parallel with the first block. This helps in minimizing speed mismatch between devices and the processor. Also, this allows process computation in parallel with input/output as shown in *Figure 2*.

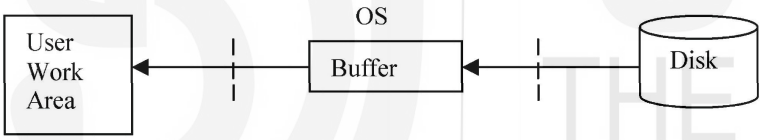


Figure 2: Single Buffering

Double buffering is an improvement over this. A pair of buffers is used; blocks/records generated by a running process are initially stored in the first buffer until it is full. Then from this buffer it is transferred to the secondary storage. During this transfer the other blocks generated are deposited in the second buffer and when this second buffer is also full and first buffer transfer is complete, then transfer from the second buffer is initiated. This process of alternation between buffers continues which allows I/O to occur in parallel with a process's computation. This scheme increases the complexity but yields improved performance as shown in *Figure 3*.

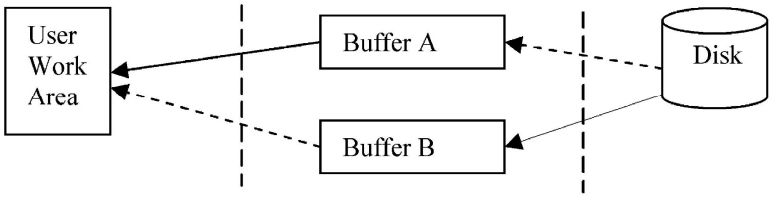


Figure 3: Double Buffering

3.4 DISK ORGANISATION

Disks come in different shapes and sizes. The most obvious distinction between floppy disks, diskettes and hard disks is: floppy disks and diskettes consist, of a

single disk of magnetic material, while hard-disks normally consist of several stacked on top of one another. Hard disks are totally enclosed devices which are much more finely engineered and therefore require protection from dust. A hard disk spins at a constant speed, while the rotation of floppy drives is switched on and off. On the Macintosh machine, floppy drives have a variable speed operation, whereas most floppy drives have only a single speed of rotation. As hard drives and tape units become more efficient and cheaper to produce, the role of the floppy disk is diminishing. We look therefore mainly at hard drives.

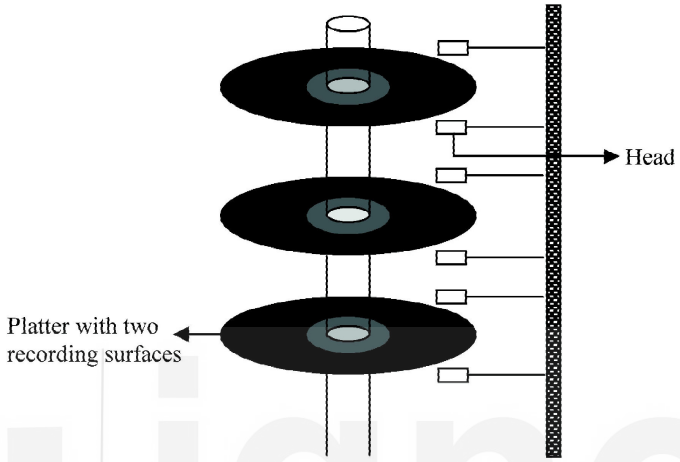


Figure 4: Hard Disk with 3 platters

Looking at the *Figure 4*, we see that a hard disk is composed of several physical disks stacked on top of each other. The disk shown in the *Figure 4* has 3 platters and 6 recording surfaces (two on each platter). A separate read *head* is provided for each *surface*. Although the disks are made of continuous magnetic material, there is a limit to the *density* of information which can be stored on the disk. The heads are controlled by a *stepper motor* which moves them in fixed-distance intervals across each surface. i.e., there is a fixed number of *tracks* on each surface. The tracks on all the surfaces are aligned, and the sum of all the tracks at a fixed distance from the edge of the disk is called a *cylinder*. To make the disk access quicker, tracks are usually divided up into *sectors* – or fixed size regions which lie along tracks. When writing to a disk, data are written in units of a whole number of sectors. (In this respect, they are similar to pages or frames in physical memory). On some disks, the sizes of sectors are decided by the manufacturers in hardware. On other systems (often microcomputers) it might be chosen in software when the disk is prepared for use (*formatting*). Because the heads of the disk move together on all surfaces, we can increase read-write efficiency by allocating blocks in parallel across all surfaces. Thus, if a file is stored in consecutive blocks, on a disk with *n* surfaces and *n* heads, it could read *n* sectors *per-track* without any head movement. When a disk is supplied by a manufacturer, the physical properties of the disk (number of tracks, number of heads, sectors per track, speed of revolution) are provided with the disk. An operating system must be able to adjust to different types of disk. Clearly *sectors per track* is not a constant, nor is the number of tracks. The numbers given are just a convention used to work out a consistent set of addresses on a disk and may not have anything to do with the hard and fast physical limits of the disk. To address any portion of a disk, we need a three component address consisting of (*surface, track and sector*).

The *seek time* is the time required for the disk arm to move the head to the cylinder with the desired sector. The *rotational latency* is the time required for the disk to rotate the desired sector until it is under the read-write head.

3.4.1 Device drivers and IDs

A hard-disk is a *device*, and as such, an operating system must use a *device controller* to talk to it. Some device controllers are simple microprocessors which translate numerical addresses into head motor movements, while others contain small decision making computers of their own. The most popular type of drive for larger personal computers and workstations is the SCSI drive. SCSI (pronounced scuzzy) (Small Computer System Interface) is a protocol and now exists in four variants SCSI 1, SCSI 2, fast SCSI 2, and SCSI 3. SCSI disks live on a *data bus* which is a fast parallel data link to the CPU and memory, rather like a very short network. Each drive coupled to the bus identifies itself by a SCSI address and each SCSI controller can address up to seven units. If more disks are required, a second controller must be added. SCSI is more efficient at multiple accesses sharing than other disk types for microcomputers. In order to talk to a SCSI disk, an operating system must have a SCSI device driver. This is a layer of software which translates disk requests from the operating system's abstract command-layer into the language of signals which the SCSI controller understands.

3.4.2 Checking Data Consistency and Formatting

Hard drives are not perfect: they develop defects due to magnetic dropout and imperfect manufacturing. On more primitive disks, this is checked when the disk is *formatted* and these damaged sectors are avoided. If the sector becomes damaged under operation, the structure of the disk must be patched up by some repair program. Usually the data are lost.

On more intelligent drives, like the SCSI drives, the disk itself keeps a *defect list* which contains a list of all bad sectors. Anew disk from the manufacturer contains a starting list and this is updated as time goes by, if more defects occur. Formatting is a process by which the sectors of the disk are:

- (If necessary) created by setting out 'signposts' along the tracks,
- Labelled with an address, so that the disk controller knows when it has found the correct sector.

On simple disks used by microcomputers, formatting is done manually. On other types, like SCSI drives, there is a low-level formatting already on the disk when it comes from the manufacturer. This is part of the SCSI protocol, in a sense. High level formatting on top of this is not necessary, since an advanced enough *filesystem* will be able to manage the hardware sectors. *Data consistency* is checked by writing to disk and reading back the result. If there is disagreement, an error occurs. This procedure can best be implemented inside the hardware of the disk−modern disk drives are small computers in their own right. Another cheaper way of checking data consistency is to calculate a number for each sector, based on what data are in the sector and store it in the sector. When the data are read back, the number is recalculated and if there is disagreement then an error is signalled. This is called a *cyclic redundancy check* (CRC) or *error correcting*

code. Some device controllers are intelligent enough to be able to detect bad sectors and move data to a spare 'good' sector if there is an error. Disk design is still a subject of considerable research and disks are improving both in speed and reliability by leaps and bounds.

3.5 DISK SCHEDULING

The disk is a resource which has to be shared. It is therefore has to be scheduled for use, according to some kind of scheduling system. The secondary storage media structure is one of the vital parts of the file system. Disks are the one, providing lot of the secondary storage. As compared to magnetic tapes, disks have very fast access time and disk bandwidth. The access time has two major constituents: seek time and the rotational latency.

The *seek time* is the time required for the disk arm to move the head to the cylinder with the desired sector. The *rotational latency* is the time required for the disk to rotate the desired sector until it is under the read-write head. The *disk bandwidth* is the total number of bytes transferred per unit time.

Both the access time and the bandwidth can be improved by efficient disk I/O requests scheduling. Disk drivers are large single dimensional arrays of logical blocks to be transferred. Because of large usage of disks, proper scheduling algorithms are required.

A scheduling policy should attempt to maximize throughput (defined as the number of requests serviced per unit time) and also to minimize mean response time (i.e., average waiting time plus service time). These scheduling algorithms are discussed below:

3.5.1 FCFS Scheduling

First-Come, First-Served (FCFS) is the basis of this simplest disk scheduling technique. There is no reordering of the queue. Suppose the requests for inputting/ outputting to blocks on the cylinders have arrived, forming the following disk queue:

50, 91, 150, 42, 130, 18, 140, 70, 60

Also assume that the disk head is initially at cylinder 50 then it moves to 91, then to 150 and so on. The total head movement in this scheme is 610 cylinders, which makes the system slow because of wild swings. Proper scheduling while moving towards a particular direction could decrease this. This will further improve performance. FCFS scheme is clearly depicted in *Figure 5*.

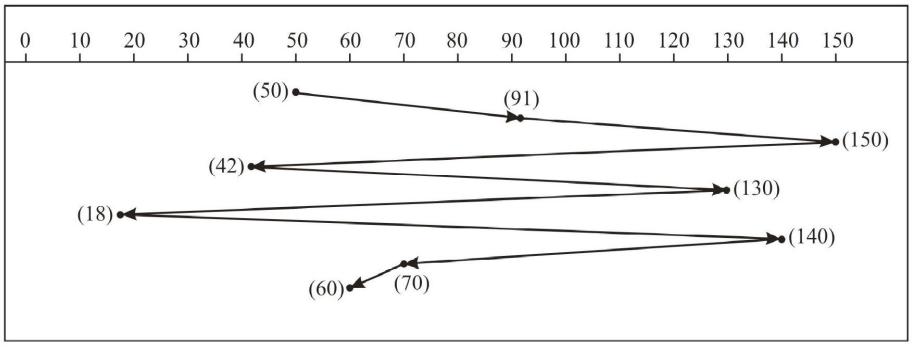


Figure 5: FCFS Scheduling

3.5.2 SSTF Scheduling

The basis for this algorithm is Shortest-Seek-Time-First (SSTF) i.e., service all the requests close to the current head position and with minimum seeks time from current head position.

In the previous disk queue sample the cylinder close to critical head position i.e., 50, is 42 cylinder, next closest request is at 60. From there, the closest one is 70, then 91,130,140,150 and then finally 18-cylinder. This scheme has reduced the total head movements to 248 cylinders and hence improved the performance. Like SJF (Shortest Job First) for CPU scheduling SSTF also suffers from starvation problem. This is because requests may arrive at any time. Suppose we have the requests in disk queue for cylinders 18 and 150, and while servicing the 18-cylinder request, some other request closest to it arrives and it will be serviced next. This can continue further also making the request at 150-cylinder wait for long. Thus a continual stream of requests near one another could arrive and keep the far away request waiting indefinitely. The SSTF is not the optimal scheduling due to the starvation problem. This whole scheduling is shown in *Figure 6*.

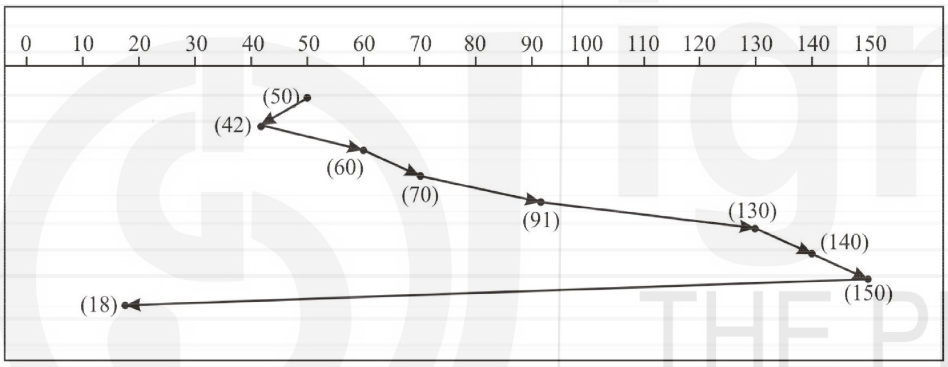


Figure 6: SSTF Scheduling

3.5.3 SCAN Scheduling

The disk arm starts at one end of the disk and service all the requests in its way towards the other end, i.e., until it reaches the other end of the disk where the head movement is reversed and continue servicing in this reverse direction. This scanning is done back and forth by the head continuously.

In the example problem two things must be known before starting the scanning process. Firstly, the initial head position i.e., 50 and then the head movement direction (let it towards 0, starting cylinder). Consider the disk queue again:

91, 150, 42, 130, 18, 140, 70, 60

Starting from 50 it will move towards 0, servicing requests 42 and 18 in between. At cylinder 0 the direction is reversed and the arm will move towards the other end of the disk servicing the requests at 60, 70, 91,130,140 and then finally 150.

As the arm acts like an elevator in a building, the SCAN algorithm is also known as elevator algorithm sometimes. The limitation of this scheme is that few requests need to wait for a long time because of reversal of head direction. This scheduling algorithm results in a total head movement of only 200 cylinders. *Figure 7* shows this scheme:

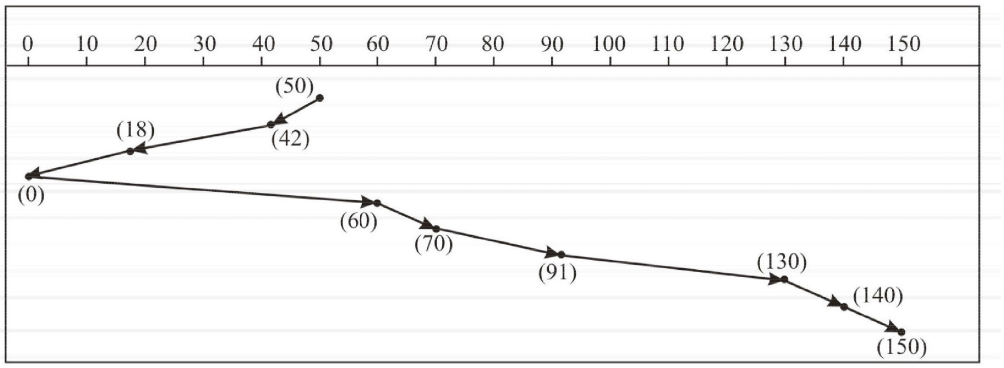


Figure 7: SCAN Scheduling

3.5.4 C-SCAN Scheduling

Similar to SCAN algorithm, C-SCAN also moves head from one end to the other servicing all the requests in its way. The difference here is that after the head reaches the end it immediately returns to beginning, skipping all the requests on the return trip. The servicing of the requests is done only along one path. Thus comparatively this scheme gives uniform wait time because cylinders are like circular lists that wrap around from the last cylinder to the first one.

3.5.5 LOOK and C-LOOK Scheduling

These are just improvements of SCAN and C-SCAN but difficult to implement. Here the head moves only till final request in each direction (first and last ones), and immediately reverses direction without going to end of the disk. Before moving towards any direction the requests are looked, avoiding the full width disk movement by the arm.

The performance and choice of all these scheduling algorithms depend heavily on the number and type of requests and on the nature of disk usage. The file allocation methods like contiguous, linked or indexed, also affect the requests. For example, a contiguously allocated file will generate nearby requests and hence reduce head movements whereas linked or indexed files may generate requests from blocks that are scattered throughout the disk and hence increase the head movements. While searching for files the directories will be frequently accessed, hence location of directories and also blocks of data in them are also important criteria. All these peculiarities force the disk scheduling algorithms to be written as a separate module of the operating system, so that these can easily be replaced. For heavily used disks the SCAN / LOOK algorithms are well suited because they take care of the hardware and access requests in a reasonable order. There is no real danger of starvation, especially in the C-SCAN case. The arrangement of data on a disk plays an important role in deciding the efficiency of data-retrieval.

3.6 RAID

Disks have high failure rates and hence there is the risk of loss of data and lots of downtime for restoring and disk replacement. To improve disk usage many techniques have been implemented. One such technology is RAID (Redundant Array of Inexpensive Disks). Its organization is based on disk striping (or interleaving), which uses a group of disks as one storage unit. Disk striping is a way of increasing the disk transfer rate up to a factor of *N*, by splitting files

across *N* different disks. Instead of saving all the data from a given file on one disk, it is split across many. Since the *N* heads can now search independently, the speed of transfer is, in principle, increased manifold. Logical disk data/blocks can be written on two or more separate physical disks which can further transfer their sub-blocks in parallel. The total transfer rate system is directly proportional to the number of disks. The larger the number of physical disks striped together, the larger the total transfer rate of the system. Hence, the overall performance and disk accessing speed is also enhanced. The enhanced version of this scheme is mirroring or shadowing. In this RAID organisation a duplicate copy of each disk is kept. It is costly but a much faster and more reliable approach. The disadvantage with disk striping is that, if one of the *N* disks becomes damaged, then the data on all *N* disks is lost. Thus striping needs to be combined with a reliable form of backup in order to be successful.

Another RAID scheme uses some disk space for holding parity blocks. Suppose, three or more disks are used, then one of the disks will act as a parity block, which contains corresponding bit positions in all blocks. In case some error occurs or the disk develops a problems all its data bits can be reconstructed. This technique is known as disk striping with parity or block interleaved parity, which increases speed. But writing or updating any data on a disk requires corresponding recalculations and changes in parity block. To overcome this the parity blocks can be distributed over all disks.

3.7 DISK CACHE

Disk caching is an extension of buffering. Cache is derived from the French word cacher, meaning to hide. In this context, a ***cache*** is a collection of blocks that logically belong on the disk, but are kept in memory for performance reasons. It is used in multiprogramming environment or in disk file servers, which maintain a separate section of main memory called disk cache. These are sets of buffers (cache) that contain the blocks that are recently used. The cached buffers in memory are copies of the disk blocks and if any data here is modified only its local copy is updated. So, to maintain integrity, updated blocks must be transferred back to the disk. Caching is based on the assumption that most shortly accessed blocks are likely to be accessed again soon. In case some new block is required in the cache buffer, one block already there can be selected for "flushing" to the disk. Also to avoid loss of updated information in case of failures or loss of power, the system can periodically flush cache blocks to the disk. The key to disk caching is keeping frequently accessed records in the disk cache buffer in primary storage.

☞ **Check Your Progress 1**

1) Indicate the major characteristics which differentiate I/O devices.

.....

.....

.....

2) Explain the term device independence. What is the role of device drivers in this context?

.....

.....

.....

3) Describe the ways in which a device driver can be implemented?

.....

.....

.....

4) What is the advantage of the double buffering scheme over single buffering?

.....

.....

.....

5) What are the key objectives of the I/O system?

.....

.....

.....

.....

3.8 COMMAND LANGUAGE USER'S VIEW OF THE FILE SYSTEM

The most important aspect of a file system is its appearance from the user's point of view. The user prefers to view the naming scheme of files, the constituents of a file, what the directory tree looks like, protection specifications, file operations allowed on them and many other interface issues. The internal details like, the data structure used for free storage management, number of sectors in a logical block etc. is of less interest to the users. From a user's perspective, a file is the smallest allotment of logical secondary storage. Users should be able to refer to their files by symbolic names rather than having to use physical device names.

The operating system allows users to define named objects called files which can hold interrelated data, programs or any other thing that the user wants to store/save.

3.9 THE SYSTEM PROGRAMMER'S VIEW OF THE FILE SYSTEM

As discussed earlier, the system programmer's and designer's view of the file system is mainly concerned with the details/issues like whether linked lists or simple arrays are used for keeping track of free storage and also the number of sectors useful in any logical block. But it is rare that physical record size will

exactly match the length of desired logical record. The designers are mainly interested in seeing how disk space is managed, how files are stored and how to make everything work efficiently and reliably.

3.10 THE OPERATING SYSTEM'S VIEW OF FILE MANAGEMENT

As discussed earlier, the operating system abstracts (maps) from the physical properties of its storage devices to define a logical storage unit i.e., the file. The operating system provides various system calls for file management like creating, deleting files, read and write, truncate operations etc. All operating systems focus on achieving device-independence by making the access easy regardless of the place of storage (file or device). The files are mapped by the operating system onto physical devices. Many factors are considered for file management by the operating system like directory structure, disk scheduling and management, file related system services, input/output etc. Most operating systems take a different approach to storing information. Three common file organisations are byte sequence, record sequence and tree of disk blocks. UNIX files are structured in simple byte sequence form. In record sequence, arbitrary records can be read or written, but a record cannot be inserted or deleted in the middle of a file. CP/M works according to this scheme. In tree organisation each block hold ***n*** keyed records and a new record can be inserted anywhere in the tree. The mainframes use this approach. The OS is responsible for the following activities in regard to the file system:

- The creation and deletion of files
- The creation and deletion of directory
- The support of system calls for files and directories manipulation
- The mapping of files onto disk
- Backup of files on stable storage media (non-volatile).

The coming sub-sections cover these details as viewed by the operating system.

3.10.1 Directories

A file directory is a group of files organised together. An entry within a directory refers to the file or another directory. Hence, a tree structure/hierarchy can be formed. The directories are used to group files belonging to different applications/ users. Large-scale time sharing systems and distributed systems store thousands of files and bulk of data. For this type of environment a file system must be organised properly. A File system can be broken into partitions or volumes. They provide separate areas within one disk, each treated as separate storage devices in which files and directories reside. Thus directories enable files to be separated on the basis of user and user applications, thus simplifying system management issues like backups, recovery, security, integrity, name-collision problem (file name clashes), housekeeping of files etc.

The device directory records information like name, location, size, and type for all the files on partition. A root refers to the part of the disk from where the root directory begins, which points to the user directories. The root directory is distinct

from sub-directories in that it is in a fixed position and of fixed size. So, the directory is like a symbol table that converts file names into their corresponding directory entries. The operations performed on a directory or file system are:

1) Create, delete and modify files.
2) Search for a file.
3) Mechanisms for sharing files should provide controlled access like read, write, execute or various combinations.
4) List the files in a directory and also contents of the directory entry.
5) Renaming a file when its contents or uses change or file position needs to be changed.
6) Backup and recovery capabilities must be provided to prevent accidental loss or malicious destruction of information.
7) Traverse the file system.

The most common schemes for describing logical directory structure are:

(i) **Single-level directory**

All the files are inside the same directory, which is simple and easy to understand; but the limitation is that all files must have unique names. Also, even with a single user as the number of files increases, it is difficult to remember and to track the names of all the files. This hierarchy is depicted in *Figure 8*.

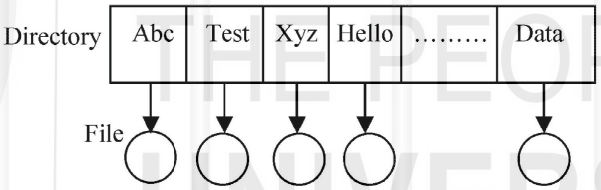


Figure 8: Single-level directory

(ii) **Two-level directory**

We can overcome the limitations of the previous scheme by creating a separate directory for each user, called User File Directory (UFD). Initially when the user logs in, the system's Master File Directory (MFD) is searched which is indexed with respect to username/account and UFD reference for that user. Thus different users may have same file names but within each UFD they should be unique. This resolves name-collision problem up to some extent but this directory structure isolates one user from another, which is not desired sometimes when users need to share or cooperate on some task. *Figure 9* shows this scheme clearly.

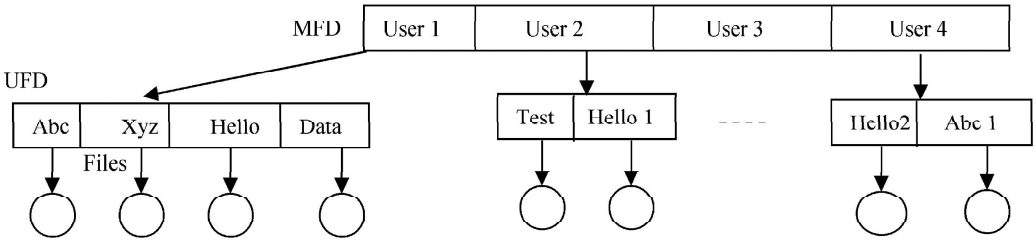


Figure 9: Two-level directory

(iii) **Tree-structured directory**

The two-level directory structure is like a 2-level tree. Thus to generalise, we can extend the directory structure to a tree of arbitrary height. Thus the user can create his/her own directory and subdirectories and can also organise files. One bit in each directory entry defines entry as a file (0) or as a subdirectory (1).

The tree has a root directory and every file in it has a unique path name (path from root, through all subdirectories, to a specified file). The pathname prefixes the filename, helps to reach the required file traversed from a base directory. The pathnames can be of 2 types: absolute path names or relative path names, depending on the base directory. An absolute path name begins at the root and follows a path to a particular file. It is a full pathname and uses the root directory. Relative defines the path from the current directory. For example, if we assume that the current directory is */Hello2* then the file *F4.doc* has the absolute pathname */Hello/Hello2/Test2/F4.doc* and the relative pathname is */Test2/F4.doc*. The pathname is used to simplify the searching of a file in a tree-structured directory hierarchy. *Figure 10* shows the layout:

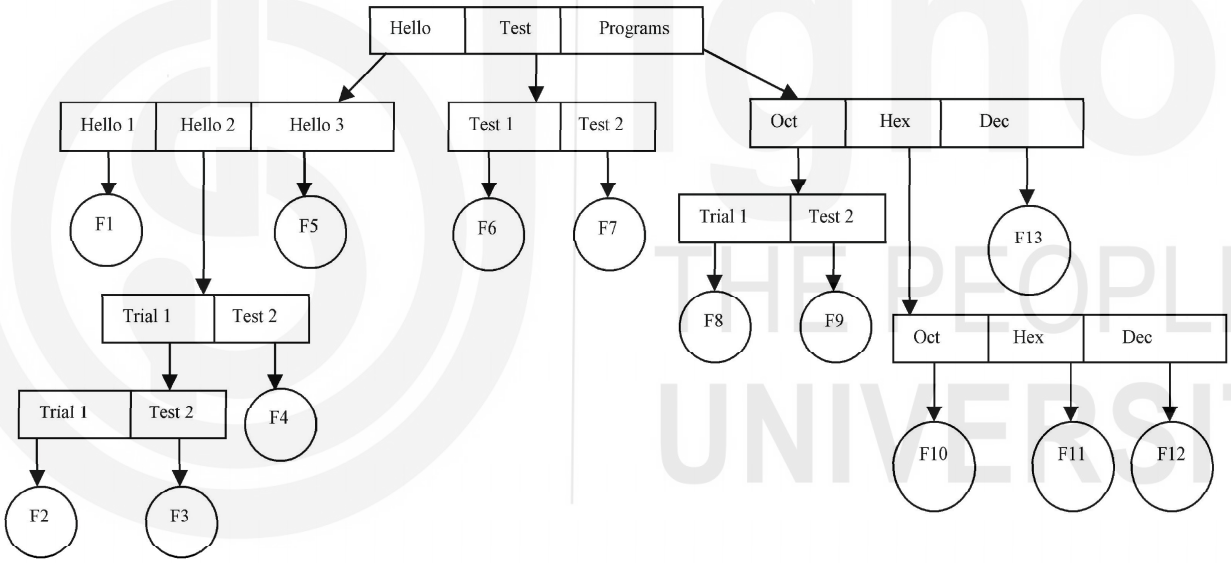


Figure 10: Tree-structured directory

(iv) **Acyclic-graph directory**:

As the name suggests, this scheme has a graph with no cycles allowed in it. This scheme added the concept of shared common subdirectory / file which exists in file system in two (or more) places at once. Having two copies of a file does not reflect changes in one copy corresponding to changes made in the other copy.

But in a shared file, only one actual file is used and hence changes are visible. Thus an acyclic graph is a generalisation of a tree-structured directory scheme. This is useful in a situation where several people are working as a team, and need access to shared files kept together in one directory. This scheme can be implemented by creating a new directory entry known as a link which points to another file or subdirectory. *Figure 11* depicts this structure for directories.

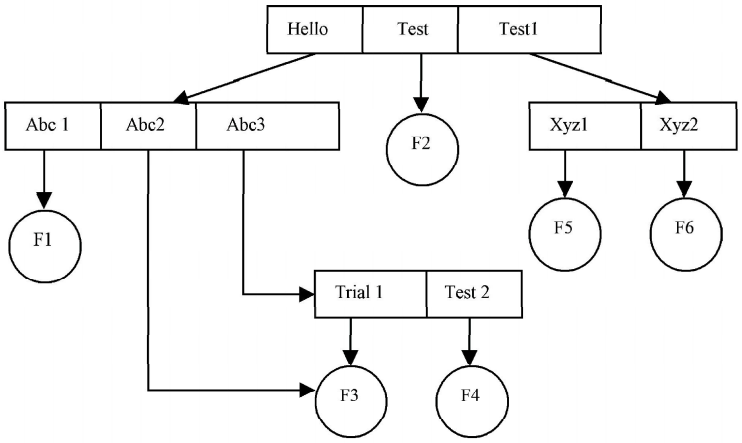


Figure 11: Acyclic-graph directory

The limitations of this approach are the difficulties in traversing an entire file system because of multiple absolute path names. Another issue is the presence of dangling pointers to the files that are already deleted, though we can overcome this by preserving the file until all references to it are deleted. For this, every time a link or a copy of directory is established, a new entry is added to the file-reference list. But in reality as the list is too lengthy, only a count of the number of references is kept. This count is then incremented or decremented when the reference to the file is added or it is deleted respectively.

(v) **General graph Directory**:

Acyclic-graph does not allow cycles in it. However, when cycles exist, the reference count may be non-zero, even when the directory or file is not referenced anymore. In such situation garbage collection is useful. This scheme requires the traversal of the whole file system and marking accessible entries only. The second pass then collects everything that is unmarked on a free-space list. This is depicted in *Figure 12*.

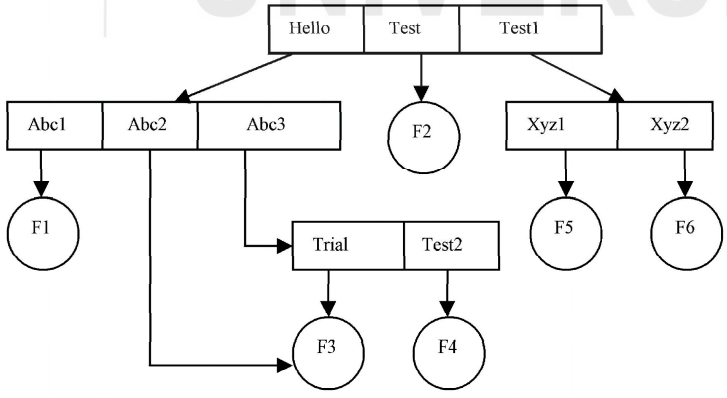


Figure 12: General-graph directory

3.10.2 Disk Space Management

The direct-access of disks and keeping files in adjacent areas of the disk is highly desirable**.** But the problem is how to allocate space to files for effective disk space utilization and quick access. Also, as files are allocated and freed, the space on a disk become fragmented. The major methods of allocating disk space are:

i) Continuous

ii) Non-continuous (Indexing and Chaining)

i) Continuous

This is also known as contiguous allocation as each file in this scheme occupies a set of contiguous blocks on the disk. A linear ordering of disk addresses is seen on the disk. It is used in VM/CMS– an old interactive system. The advantage of this approach is that successive logical records are physically adjacent and require no head movement. So disk seek time is minimal and speeds up access of records. Also, this scheme is relatively simple to implement. The technique in which the operating system provides units of file space on demand by user running processes, is known as dynamic allocation of disk space. Generally space is allocated in units of a fixed size, called an allocation unit or a 'cluster' in MS-DOS. Cluster is a simple multiple of the disk physical sector size, usually 512 bytes. Disk space can be considered as a one-dimensional array of data stores, each store being a cluster. A larger cluster size reduces the potential for fragmentation, but increases the likelihood that clusters will have unused space. Using clusters larger than one sector reduces fragmentation, and reduces the amount of disk space needed to store the information about the used and unused areas on the disk.

Contiguous allocation merely retains the disk address (start of file) and length (in block units) of the first block. If a file is n blocks long and it begins with location b (blocks), then it occupies b, b+1, b+2,…, b+n-1 blocks. First-fit and best-fit strategies can be used to select a free hole from available ones. But the major problem here is searching for sufficient space for a new file. *Figure 13* depicts this scheme:

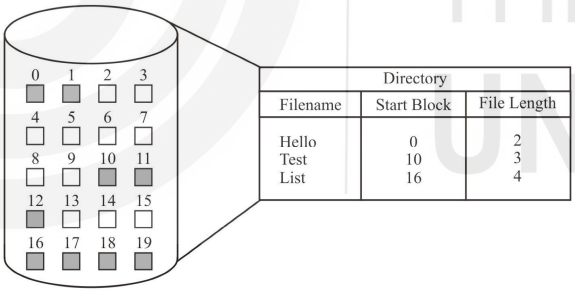


Figure 13: Contiguous Allocation on the Disk

This scheme exhibits similar fragmentation problems as in variable memory partitioning. This is because allocation and deal location could result in regions of free disk space broken into chunks (pieces) within active space, which is called external fragmentation. A repacking routine called compaction can solve this problem. In this routine, an entire file system is copied on to tape or another disk and the original disk is then freed completely. Then from the copied disk, files are again stored back using contiguous space on the original disk. But this approach can be very expensive in terms of time. Also, size-declaration in advance is a problem because each time, the size of file is not predictable. But it supports both sequential and direct accessing. For sequential access, almost no seeks are required. Even direct access with seek and read is fast. Also, calculation of blocks holding data is quick and easy as we need just offset from the start of the file.

ii) Non-Continuous (Chaining and Indexing)

This scheme has replaced the previous ones. The popular non-contiguous storages allocation schemes are:

- Linked/Chained allocation
- Indexed Allocation.

Linked/Chained allocation: All files are stored in fixed size blocks and adjacent blocks are linked together like a linked list. The disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last block of the file. Also each block contains pointers to the next block, which are not made available to the user. There is no external fragmentation in this as any free block can be utilised for storage. So, compaction and relocation is not required. But the disadvantage here is that it is potentially inefficient for direct-accessible files since blocks are scattered over the disk and have to follow pointers from one disk block to the next. It can be used effectively for sequential access only but there also it may generate long seeks between blocks. Another issue is the extra storage space required for pointers. Yet the reliability problem is also there due to loss/damage of any pointer. The use of doubly linked lists could be a solution to this problem but it would add more overheads for each file. A doubly linked list also facilitates searching as blocks are threaded both forward and backward. The *Figure 14* depicts linked /chained allocation where each block contains the information about the next block (i.e., pointer to next block).

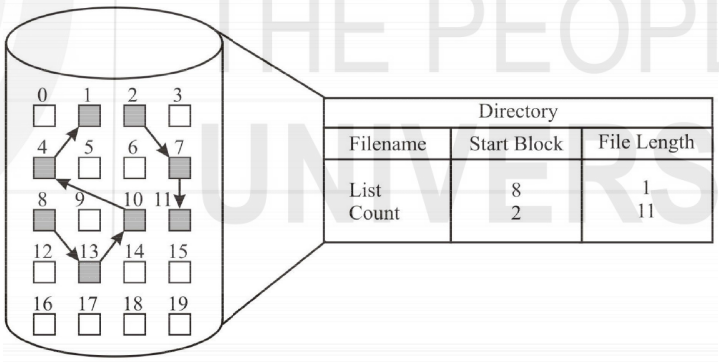


Figure 14: Linked Allocation on the Disk

MS-DOS and OS/2 use another variation on linked list called FAT (File Allocation Table). The beginning of each partition contains a table having one entry for each disk block and is indexed by the block number. The directory entry contains the block number of the first block of file. The table entry indexed by block number contains the block number of the next block in the file. The Table pointer of the last block in the file has EOF pointer value. This chain continues until EOF (end of file) table entry is encountered. We still have to linearly traverse next pointers, but at least we don't have to go to the disk for each of them. 0(Zero) table value indicates an unused block. So, allocation of free blocks with FAT scheme is straightforward, just search for the first block with 0 table pointer. MS-DOS and OS/2 use this scheme. This scheme is depicted in *Figure 15.*

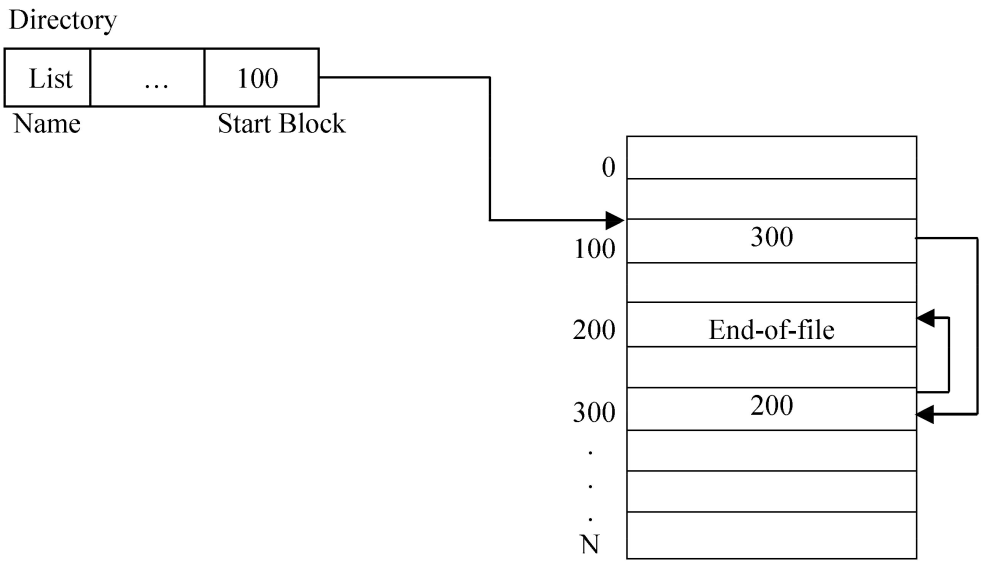


Figure 15: File-Allocation Table (FAT)

Indexed Allocation: In this each file has its own index block. Each entry of the index points to the disk blocks containing actual file data i.e., the index keeps an array of block pointers for each file. So, index block is an array of disk block addresses. The i^{th} entry in the index block points to the i^{th} block of the file. Also, the main directory contains the address where the index block is on the disk. Initially, all the pointers in index block are set to ***NIL***. The advantage of this scheme is that it supports both sequential and random access. The searching may take place in index blocks themselves. The index blocks may be kept close together in secondary storage to minimize seek time. Also space is wasted only on the index which is not very large and there's no external fragmentation. But a few limitations of the previous scheme still exists in this, like, we still need to set maximum file length and we can have overflow scheme of the file larger than the predicted value. Insertions can require complete reconstruction of index blocks also. The indexed allocation scheme is diagrammatically shown in *Figure 16*.

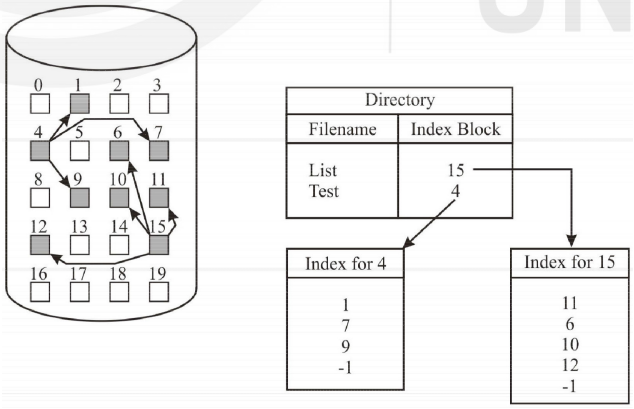


Figure 16: Indexed Allocation on the Disk

3.10.3 Disk Address Translation

We have seen in Unit-1 memory management that the virtual addresses generated by a program is different from the physical. The translation of virtual addresses to physical addresses is performed by MMU. Disk address translation considers the aspects of data storage on the disk. Hard disks are totally enclosed devices, which are more finely engineered and therefore require protection from dust. A

hard disk spins at a constant speed. Briefly, hard disks consist of one or more rotating platters. A read-write head is positioned above the rotating surface and is able to read or write the data underneath the current head position. The hard drives are able to present the "geometry" or "addressing scheme" of the drive. Consider the disk internals first. Each track of the disk is divided into sections called sectors. A sector is the smallest physical storage unit on the disk. The size of a sector is always a power of two, and is almost always 512 bytes. A sector is the part of a slice that lies within a track. The position of the head determines which track is being read. A cylinder is almost the same as a track, except that it means all tracks lining up under each other on all the surfaces. The head is equivalent to side(s). It simply means one of the rotating platters or one of the sides on one of the platters. If a hard disk has three rotating platters, it usually has 5 or 6 readable sides, each with its own read-write head.

The MS-DOS file systems allocate storage in clusters, where a cluster is one or more contiguous sectors. MS-DOS bases the cluster size on the size of the partition. As a file is written on the disk, the file system allocates the appropriate number of clusters to store the file's data. For the purposes of isolating special areas of the disk, most operating systems allow the disk surface to be divided into *partitions*. A partition (also called a *cylinder group*) is just that: a group of cylinders, which lie next to each other. By defining partitions we divide up the storage of data to special areas, for convenience. Each partition is assigned a separate logical device and each device can only write to the cylinders, which are defined as being its own. To access the disk the computer needs to convert physical disk geometry (the number of cylinders on the disk, number of heads per cylinder, and sectors per track) to a logical configuration that is compatible with the operating system. This conversion is called translation. Since sector translation works between the disk itself and the system BIOS or firmware, the operating system is unaware of the actual characteristics of the disk, if the number of cylinders, heads, and sectors per track the computer needs is within the range supported by the disk. MS-DOS presents disk devices as logical volumes that are associated with a drive code (A, B, C, and so on) and have a volume name (optional), a root directory, and from zero to many additional directories and files.

3.10.4 File Related System Services

A file system enables applications to store and retrieve files on storage devices. Files are placed in a hierarchical structure. The file system specifies naming conventions for files and the format for specifying the path to a file in the tree structure. OS provides the ability to perform input and output (I/O) operations on storage components located on local and remote computers. In this section we briefly describe the system services, which relate to file management. We can broadly classify these under categories:

i) Online services

ii) Programming services.

i) **Online-services**: Most operating systems provide interactive facilities to enable the on-line users to work with files. Few of these facilities are built-in commands of the system while others are provided by separate utility

programs. But basic operating systems like MS-DOS with limited security provisions can be potentially risky because of these user owned powers. So, these must be used by technical support staff or experienced users only. For example: DEL *. * Command can erase all the files in the current directory. Also, FORMAT c: can erase the entire contents of the mentioned drive/disk. Many such services provided by the operating system related to directory operations are listed below:

- Create a file
- Delete a file
- Copy a file
- Rename a file
- Display a file
- Create a directory
- Remove an empty directory
- List the contents of a directory
- Search for a file
- Traverse the file system.

ii) **Programming services**: The complexity of the file services offered by the operating system vary from one operating system to another but the basic set of operations like: open (make the file ready for processing), close (make a file unavailable for processing), read (input data from the file), write (output data to the file), seek (select a position in file for data transfer).

All these operations are used in the form of language syntax procedures or built-in library routines, of high-level language used like C, Pascal, and Basic etc. More complicated file operations supported by the operating system provide wider range of facilities/services. These include facilities like reading and writing records, locating a record with respect to a primary key value etc. The software interrupts can also be used for activating operating system functions. For example, Interrupt 21(hex) function call request in MS-DOS helps in opening, reading and writing operations on a file.

In addition to file functions described above the operating system must provide directory operation support also like:

- Create or remove directory
- Change directory
- Read a directory entry
- Change a directory entry etc.

These are not always implemented in a high level language but language can be supplied with these procedure libraries. For example, UNIX uses C language as system programming language, so that all system calls requests are implemented as C procedures.

3.10.5 Asynchronous Input/Output

Synchronous I/O is based on blocking concept while asynchronous is interrupt-driven transfer. If an user program is doing several things simultaneously and request for I/O operation, two possibilities arise. The simplest one is that the I/O is started, then after its completion, control is transferred back to the user process. This is known as synchronous I/O where you make an I/O request and you have to wait for it to finish. This could be a problem where you would like to do some background processing and wait for a key press. Asynchronous I/O solves this problem, which is the second possibility. In this, control is returned back to the user program without waiting for the I/O completion. The I/O then continues while other system operations occur. The CPU starts the transfer and goes off to do something else until the interrupt arrives.

Asynchronous I/O operations run in the background and do not block user applications. This improves performance, because I/O operations and applications processing can run simultaneously. Many applications, such as databases and file servers, take advantage of the ability to overlap processing and I/O. To manage asynchronous I/O, each asynchronous I/O request has a corresponding control block in the application's address space. This control block contains the control and status information for the request. It can be used again when the I/O operation is completed. Most physical I/O is asynchronous.

After issuing an asynchronous I/O request, the user application can determine when and how the I/O operation is completed. This information is provided in any of three ways:

- The application can poll the status of the I/O operation.
- The system can asynchronously notify the application when the I/O operation is done.
- The application can block until the I/O operation is complete.

Each I/O is handled by using a device-status table. This table holds entry for each I/O device indicating device's type, address, and its current status like bust or idle. When any I/O device needs service, it interrupts the operating system. After determining the device, the Operating System checks its status and modifies table entry reflecting the interrupt occurrence. Control is then returned back to the user process.

☞ **Check Your Progress 2**

1) What is the meaning of the term 'virtual device'? Give an Example.

.....

.....

2) What are the advantages of using directories?

.....

.....

3) Explain the advantages of organising file directory structure into a tree structure?

.....

.....

4) List few file attributes?

.....

.....

5) Why is SCAN scheduling also called Elevator Algorithm?

.....

.....

6) In an MS-DOS disk system, calculate the number of entries (i.e., No. of clusters) required in the FAT table. Assume the following parameters:

Disk Capacity - 40 Mbytes

Block Size - 512 Bytes

Blocks/Cluster- 4

.....

.....

7) Assuming a cluster size of 512 bytes calculate the percentage wastage in file space due to incomplete filling of last cluster, for the file sizes below:

(i) 1000 bytes (ii) 20,000 bytes

.....

.....

8) What is meant by an 'alias filename' and explain its UNIX implementation.

.....

.....

3.11 SUMMARY

This unit briefly describes the aspects of I/O and File Management. We started by looking at I/O controllers, organisation and I/O buffering. We briefly described various buffering approaches and how buffering is effective in smoothing out the speed mismatch between I/O rates and processor speed. We also looked at the four levels of I/O software: the interrupt handlers, device drivers, the device independent I/O software, and the I/O libraries and user-level software.

A well-designed file system should provide a user-friendly interface. The file system generally includes access methods, file management issues like file integrity, storage, sharing, security and protection etc. We have discussed the services provided by the operating system to the user to enable fast access and processing of files.

The important concepts related to the file system are also introduced in this unit like file concepts, attributes, directories, tree structure, root directory, pathnames, file services etc. Also a number of techniques applied to improve disk system performance have been discussed and in summary these are: disk caching, disk scheduling algorithms (FIFO, SSTF, SCAN, CSCAN, LOOK etc.), types of disk space management (contiguous and non-contiguous-linked and indexed), disk address translation, RAID based on interleaving concept etc. Auxiliary storage management is also considered as it is mainly concerned with allocation of space for files:

3.12 SOLUTIONS/ANSWERS

Check Your Progress 1

1) The major characteristics are:

Data Rate	Disk-2Mbytes/sec Keyboard-10-15 bytes/sec
Units of transfer Operation	Disk-read, write, seek Printer-write, move, select font
Error conditions	Disk-read errors Printer-paper out etc.

2) Device independence refers to making the operating system software and user application independent of the devices attached. This enables the devices to be changed at different executions. Output to the printer can be sent to a disk file. Device drivers act as software interface between these I/O devices and user-level software.

3) In some operating systems like UNIX, the driver code has to be compiled and linked with the kernel object code while in some others, like MS-DOS, device drivers are installed and loaded dynamically. The advantage of the former way is its simplicity and run-time efficiency but its limitation is that addition of a new device requires regeneration of kernel, which is eliminated in the latter technique.

4) In double - buffering the transfers occur at maximum rate hence it sustains device activity at maximum speed, while single buffering is slowed down by buffer to transfer times.

5) The key objectives are to maximize the utilization of the processor, to operate the devices at their maximum speed and to achieve device independence.

Check Your Progress 2

1) A virtual device is a simulation of an actual device by the operating system. It responds to the system calls and helps in achieving device independence. Example: Print Spooler.

2) Directories enable files to be separated on the basis of users and their applications. Also, they simplify the security and system management problems.

3) Major advantages are:

   - This helps to avoid possible file name clashes
   - Simplifies system management and installations
   - Facilitates file management and security of files
   - Simplifies running of different versions of same application.

4) File attributes are: Name, Type (in UNIX), Location at which file is stored on disk, size, protection bits, date and time, user identification etc.

5) Since an elevator continues in one direction until there are no more requests pending and then it reverses direction just like SCAN scheduling.

6) Cluster size= 4*512=2048 bytes

   Number of clusters = (40*1,000,000)/2048=19531 approximately

7) (i) File size = 1000 bytes

   No. of clusters = 1000/512 = 2(approximately)

   Total cluster capacity = 2*512 = 1024 bytes

   Wasted space = 1024-1000 = 24 bytes

   Percentage Wasted space = (24 / 1024)*100

   = 2.3%

   (ii) File size = 20,000 bytes

   No. of clusters = 20,000/512 = 40(approximately)

   Total cluster capacity = 40 * 512 = 20480 bytes

   Wasted space = 20480-20,000 = 480 bytes

   Percentage Wasted space = (480 / 20480) * 100

   = 2.3%

8) An alias is an alternative name for a file, possibly in a different directory. In UNIX, a single inode is associated with each physical file. Alias names are stored as separate items in directories but point to the same inode. It allows one physical file to be referenced by two or more different names or same name in different directory.

3.13 FURTHER READINGS

1) Abraham Silberschatz and Peter Baer Galvin, *Operating System Concepts*, Wiley and Sons (Asia) Publications, New Delhi.

2) H.M.Deitel, *Operating Systems*, Pearson Education Asia Place, New Delhi.

3) Andrew S.Tanenbaum, *Operating Systems- Design and Implementation*, Prentice Hall of India Pvt. Ltd., New Delhi.

4) Colin Ritchie, *Operating Systems incorporating UNIX and Windows*, BPB Publications, New Delhi.

5) J. Archer Harris, *Operating Systems*, Schaum's Outlines, TMGH, 2002, New Delhi.

6) Achyut S Godbole, *Operating Systems*, Second Edition, TMGH, 2005, New Delhi.

7) Milan Milenkovic, *Operating Systems*, TMGH, 2000, New Delhi.

8) D. M. Dhamdhere, *Operating Systems – A Concept Based Approach*, TMGH, 2002, New Delhi.

9) William Stallings, *Operating Systems*, Pearson Education, 2003, New Delhi.

10) Gary Nutt, *Operating Systems – A Modern Perspective*, Pearson Education, 2003, New Delhi.

UNIT 4 SECURITY AND PROTECTION

Structure

4.0 INTRODUCTION

Modern organisations depend heavily on their information systems and large investments are made on them annually. These systems are now computerised, and networking has been the common trend during the last decade. The availability of information and computer resources within an organisation as well as between cooperating organisations is often critical for the production of goods and services. In addition, data stored in or derived from the system must be correct, i.e., data integrity must be ensured. In some situations, it is also of great importance to keep information confidential.

Computer security is traditionally defined by the three attributes of confidentiality, integrity, and availability. *Confidentiality* is the prevention of unauthorised disclosure of information. *Integrity* is the prevention of unauthorised modification of information, and *availability* is the prevention of unauthorised withholding of information or resources. Protection refers to a mechanism for controlling the access of programs, processes, or users to the resources defined by a computer controls to be imposed, together with some means of enforcement. Protection can improve reliability by detecting latent errors at the interfaces between component subsystems. Early detection of interface errors can often prevent contamination of a healthy subsystem by a subsystem that is malfunctioning. An

unprotected resource cannot defend against use (or misuse) by an unauthorised or incompetent user.

Even if a perfectly secure operating system was created, human error (or purposeful human malice) can make it insecure. More importantly, there is no such thing as a completely secure system. No matter how secure the experts might think a particular system is, there exists someone who is clever enough to bypass the security.

It is important to understand that a trade-off always exists between security and the ease of use. It is possible to be *too* secure but security always extracts a price, typically making it more difficult to use your systems for their intended purposes. Users of a secure system need to continually type in, continually change and memorise complex passwords for every computer operation, or use biometric systems with retina and fingerprint and voice scans. All these measures along with confirmations are likely to bring any useful work to a snail's pace. If the consequences are great enough, then you might have to accept extreme security measures. Security is a relative concept, and gains in security often come only with penalties in performance. To date, most systems designed to include security in the operating system structure have exhibited either slow response times or awkward user interfaces-or both.

This unit discusses about the various types of security threats and the commonly deployed detective and preventive methods.

4.1 OBJECTIVES

After going through this unit, you should be able to:

- identify the security threats and goals;
- mention the role of security policies and mechanisms;
- discuss the significance of authentication and also describe various types of authentication procedures, and
- describe the various models of protection.

4.2 SECURITY THREATS

System security can mean several things. To have system security we need to protect the system from corruption and we need to protect the data on the system. There are many reasons why these need not be secure.

- Malicious users may try to hack into the system to destroy it.
- Power failure might bring the system down.
- A badly designed system may allow a user to accidentally destroy important data.
- A system may not be able to function any longer because one user fills up the entire disk with garbage.

Although discussions of security usually concentrate on the first of these possibilities, the latter two can be equally damaging the system in practice. One can protect against power failure by using un-interruptible power supplies (UPS). These are units which detect quickly when the power falls below a certain threshold and switch to a battery. Although the battery does not last forever−the UPS gives a system administrator a chance to halt the system by the proper route.

The problem of malicious users has been heightened in recent years by the growth of international networks. Anyone connected to a network can try to log on to almost any machine. If a machine is very insecure, they may succeed. In other words, we are not only looking at our local environment anymore, we must consider potential threats to system security to come from any source. The final point can be controlled by enforcing quotas on how much disk each user is allowed to use.

We can classify the security attacks into two types as mentioned below:

1) **Direct:** This is any direct attack on your specific systems, whether from outside hackers or from disgruntled insiders.

2) **Indirect:** This is general random attack, most commonly computer viruses, computer worms, or computer Trojan horses.

These security attacks make the study of security measures very essential for the following reasons:

- **To prevent loss of data:** You don't want someone hacking into your system and destroying the work done by you or your team members. Even if you have good back-ups, you still have to identify that the data has been damaged (which can occur at a critical moment when a developer has an immediate need for the damaged data), and then restore the data as best you can from your backup systems.

- **To prevent corruption of data:** Sometimes, the data may not completely be lost, but just be partially corrupted. This can be harder to discover, because unlike complete destruction, there is still data. If the data seems reasonable, you could go a long time before catching the problem, and cascade failure could result in serious problems spreading far and wide through your systems before discovery.

- **To prevent compromise of data:** Sometimes it can be just as bad (or even worse) to have data revealed than to have data destroyed. Imagine the consequences of important trade secrets, corporate plans, financial data, etc. falling in the hands of your competitors.

- **To prevent theft of data:** Some kinds of data are subject to theft. An obvious example is the list of credit card numbers belonging to your customers. Just about anything associated with money can be stolen.

- **To prevent sabotage:** A disgruntled employee, a dishonest competitor, or even a stranger could use any combination of the above activities to maliciously harm your business. Because of the thought and intent, this is the most dangerous kind of attack, the kind that has the potential for the greatest harm to your business.

Let's now discuss some security-related issues that the OS must deal to maintain confidentiality, integrity, and availability of system resources.

4.3 SECURITY POLICIES AND MECHANISMS

Computer systems and especially their protection mechanisms must be penetration resistant. However, most, or perhaps all, current systems have security holes that make them vulnerable. As long as vulnerabilities remain in a system, the security may be circumvented. It is thus not difficult to understand why system owners would like to know how secure their particular system actually is.

Security evaluations have been performed for quite some time by mainly two different methods. The first is to classify systems into a predefined set of categories, each with different security requirements. The other method is referred to as penetration testing, which is a form of stress testing exposing weaknesses in a system. Here, the idea is to let a group of people, normally very skilled in the computer security area, attack a target system.

A **security policy** establishes accountability for information protection by defining a set of rules, conditions, and practices that regulate how an organisation manages, protects, and distributes sensitive information. While substantial effort may be put in by the vendor in implementing the mechanisms to enforce the policy and developing assurance that the mechanisms perform properly, all efforts fail if the policy itself is flawed or poorly understood. For this reason, the standards require that "there must be an explicit and well-defined security policy enforced by the system". A security policy may address confidentiality, integrity, and/or availability.

4.4 AUTHENTICATION

It is the process of verifying a user's identity (who you are) through the use of a shared secret (such as a password), a physical token or an artifact (such as a key or a smart card), or a biometric measure (such as a fingerprint).

These three types of authentication are commonly referred to as something you have (physical token), something you know (shared secret), and something you are (biometric measure).

The types and rigor of authentication methods and technologies vary according to the security requirements or policies associated with specific situations and implementations.

The goal of authentication is to provide "reasonable assurance" that anyone who attempts to access a system or network is a legitimate user. In other words, authentication is designed to limit the possibility that an unauthorised user can gain access by impersonating as an authorised user. Here, again, the organisation's security policy should guide how difficult it is for one user to impersonate another. Highly sensitive or valuable information demands stronger authentication technologies than less sensitive or valuable information.

4.4.1 Passwords

The most common and least stringent form of authentication technology demands that users provide only a valid account name and a password to obtain access to a system or network. The password-based authentication is one-way and normally stores the user-id and password combination in a file that may be stored on the server in an encrypted or plaintext file. Most people using the public e-mail systems use this form of authentication.

Protection of Passwords

Some systems store the passwords of all users in a protected file. However, this scheme is vulnerable to accidental disclosure and to abuse by system administrators.

UNIX stores only an encrypted form of every password; a string is the password of user X if its encrypted form matches the encrypted password stored for X.

The encrypted password is accessible to everyone, but one cannot find the clear text password without either guessing it or breaking the encryption algorithm.

Data Encryption Standard (DES)

Encryption is based on one-way functions: functions that are cheap to compute but whose inverse is very expensive to compute. A still widely used, though older encryption algorithm is the Data Encryption Standard (DES), which uses a 56bit key.

UNIX does not encrypt passwords with a secret key, instead, it uses the password as the key to encrypt a standard string. The latter method is not as vulnerable to attacks based on special properties of the "secret" key, which must nevertheless be known to many people.

Limitations of Passwords

Users tend to choose passwords that are easy to guess; persistent guessing attacks can find 80-90% of all passwords. Good guesses include the login name, first names, treat names, words in the on-line dictionary, and any of these reversed or doubled.

The way to defeat these attacks is to choose a password that does not fall into any of those categories. Preferably passwords should contain some uppercase letters, numbers and/or special characters; their presence increases the search space by a large factor.

4.4.2 Alternative Forms of Authentication

Alternative forms of authentication include the following technologies:

- **Biometrics:** These systems read some physical characteristic of the user, such as their fingerprint, facial features, retinal pattern, voiceprints, signature analysis, signature motion analysis, typing rhythm analysis, finger length analysis, DNA analysis. These readings are compared to a database of authorised users to determine identity. The main problems of these schemes are high equipment cost and low acceptance rates.

- **Security Devices or Artifacts:** These systems require use of a special-purpose hardware device that functions like a customised key to gain system access. The device may be inserted into the system like a key or used to generate a code that is then entered into the system. The best example is the use of an ATM card, which is inserted in the machine and also requires password to be entered simultaneously. It holds the user information on either a magnetic strip or a smart chip capable of processing information.
- **Concentric-ring Authentication:** These systems require users to clear additional authentication hurdles as they access increasingly sensitive information. This approach minimizes the authentication burden as users access less sensitive data while requiring stronger proof of identity for more sensitive resources.

Any authentication method may be broken into, given sufficient time, expense, and knowledge. The goal of authentication technologies is to make entry into system expensive and difficult enough that malicious individuals can be deterred from trying to break the security.

4.5 PROTECTION IN COMPUTER SYSTEMS

The use of computers to store and modify information can simplify the composition, editing, distribution, and reading of messages and documents. These benefits are not free, however, part of the cost is the aggravation of some of the security problems discussed above and the introduction of some new problems as well. Most of the difficulties arise because a computer and its programs are shared amongst several users.

For example, consider a computer program that displays portions of a document on a terminal. The user of such a program is very likely not its developer. It is, in general, possible for the developer to have written the program so that it makes a copy of the displayed information accessible to himself (or a third party) without the permission or knowledge of the user who requested the execution of the program. If the developer is not authorised to view this information, security has been violated.

In compensation for the added complexities automation brings to security, an automated system, if properly constructed, can bestow a number of benefits as well. For example, a computer system can place stricter limits on user discretion. In the paper system, the possessor of a document has complete discretion over its further distribution. An automated system that enforces distribution constraints strictly can prevent the recipient of a message or document from passing it to others. Of course, the recipient can always copy the information by hand or repeat it verbally, but the inability to pass it on directly is a significant barrier.

An automated system can also offer new kinds of access control. Permission to execute certain programs can be granted or denied so that specific operations can be restricted to designated users. Controls can be designed so that some users can execute a program but cannot read or modify it directly. Programs protected in this way might be allowed to access information not directly available to the user, filter it, and pass the results back to the user.

Information contained in an automated system must be protected from three kinds of threats: (1) the *unauthorised disclosure* of information, (2) the *unauthorised modification* of information, and (3) the *unauthorised withholding* of information (usually called *denial of service).*

To protect the computer systems, we often need to apply some security models. Let us see in the next section about the various security models available.

4.6 SECURITY MODELS

Security models are more precise and detailed expressions of security policies discussed as above and are used as guidelines to build and evaluate systems. Models can be discretionary or mandatory. In a *discretionary* model, holders of rights can be allowed to transfer them at their discretion. In a *mandatory* model only designated roles are allowed to grant rights and users cannot transfer them. These models can be classified into those based on the access matrix, and multilevel models. The first model controls access while the second one attempts to control information flow.

Security Policy vs. Security Model

The Security Policy outlines several high level points: how the data is accessed, the amount of security required, and what the steps are when these requirements are not met. The security model is more in depth and supports the security policy. Security models are an important concept in the design of any security system. They all have different security policies applying to the systems.

4.6.1 Access Matrix Model

The access matrix model for computer protection is based on abstraction of operating system structures. Because of its simplicity and generality, it allows a variety of implementation techniques, as has been widely used.

There are three principal components in the access matrix model:

- A set of passive *objects,*
- A set of active *subjects,* which may manipulate the objects,
- A set of *rules* governing the manipulation of objects by subjects.

Objects are typically files, terminals, devices, and other entities implemented by an operating system. A subject is a process and a *domain* (a set of constraints within which the process may access certain objects). It is important to note that every subject is also an object; thus it may be read or otherwise manipulated by another subject. The ***access matrix*** is a rectangular array with one row per subject and one column per object. The entry for a particular row and column reflects the mode of access between the corresponding subject and object. The mode of access allowed depends on the type of the object and on the functionality of the system; typical modes are read, write, append, and execute.

Objects / File 3		File 1	File 2
Subject			
User 1	*r, w*	*R*	*r, w, x*
User 2	*r*	*R*	*r, w, x*
User 3	*r, w, x*	*r, w*	*r, w, x*

Figure 1: An access matrix

All accesses to objects by subjects are subject to some conditions laid down by an enforcement mechanism that refers to the data in the access matrix. This mechanism, called a *reference monitor,* rejects any accesses (including improper attempts to alter the access matrix data) that are not allowed by the current protection state and rules. References to objects of a given type must be validated by the *monitor* for that type.

While implementing the access matrix, it has been observed that the access matrix tends to be very sparse if it is implemented as a two-dimensional array. Consequently, implementations that maintain protection of data tend to store them either row wise, keeping with each subject a list of the objects and access modes allowed on it; or column wise, storing with each object a list of those subjects that may access it and the access modes allowed on each. The former approach is called the ***capability list*** approach and the latter is called the ***access control list*** approach.

In general, access control governs each user's ability to read, execute, change, or delete information associated with a particular computer resource. In effect, access control works at two levels: first, to grant or deny the ability to interact with a resource, and second, to control what kinds of operations or activities may be performed on that resource. Such controls are managed by an access control system. Today, there are numerous methods of access controls implemented or practiced in real-world settings. These include the methods described in the next four sections.

4.6.2 Mandatory Access Control

In a ***Mandatory Access Control (MAC)*** environment, all requests for access to resources are automatically subject to access controls. In such environments, all users and resources are classified and receive one or more security labels (such as "Unclassified," "Secret," and "Top Secret"). When a user requests a resource, the associated security labels are examined and access is permitted only if the user's label is greater than or equal to that of the resource.

4.6.3 Discretionary Access Control

In a ***Discretionary Access Control (DAC)*** environment, resource owners and administrators jointly control access to resources. This model allows for much greater flexibility and drastically reduces the administrative burdens of security implementation.

4.6.4 Rule-Based Access Control

In general, ***rule-based access control*** systems associate explicit access controls with specific system resources, such as files or printers. In such environments, administrators typically establish access rules on a per-resource basis, and the underlying operating system or directory services employ those rules to grant or deny access to users who request access to such resources. Rule-based access controls may use a MAC or DAC scheme, depending on the management role of resource owners.

4.6.5 Role-Based Access Control

Role-based access control (RBAC) enforces access controls depending upon a user's role(s). Roles represent specific organisational duties and are commonly mapped to job titles such as "Administrator," "Developer," or "System Analyst." Obviously, these roles require vastly different network access privileges.

Role definitions and associated access rights must be based upon a thorough understanding of an organisation's ***security policy***. In fact, roles and the access rights that go with them should be directly related to elements of the security policy.

4.6.6 Take-Grant Model

The access matrix model, properly interpreted, corresponds very well to a wide variety of actual computer system implementations. The simplicity of the model, its definition of subjects, objects, and access control mechanisms, is very appealing. Consequently, it has served as the basis for a number of other models and development efforts. We now discuss a model based on access matrix.

Take-grant models use graphs to model access control. They have been well researched. Although explained in the terms of graph theory, these models are fundamentally access matrix models. The graph structure can be represented as an adjacency matrix, and labels on the arcs can be coded as different values in the matrix.

In a take-grant model, the protection state of a system is described by a directed graph that represents the same information found in an access matrix. Nodes in the graph are of two types, one corresponding to subjects and the other to objects. An arc directed from a node A to another node B indicates that the subject (or object) A has some access right(s) to subject (or object) B. The arc is labeled with the set of A's rights to B. The possible access rights are read (r), write (w), take (t), and grant (g). Read and write have the obvious meanings. "*Take*" access implies that node A can take node B's access rights to any other node.

For example, if there is an arc labeled (r, g) from node B to node C, and if the arc from A to B includes a "t" in its label, then an arc from A to C labeled (r, g) could be added to the graph (see *Figure 2*). Conversely, if the arc from A to B is marked with a "g", B can be granted any access right A possesses. Thus, if A has (w) access to a node D and (g) access to B, an arc from B to D marked (w) can be added to the graph (see *Figure 3*).

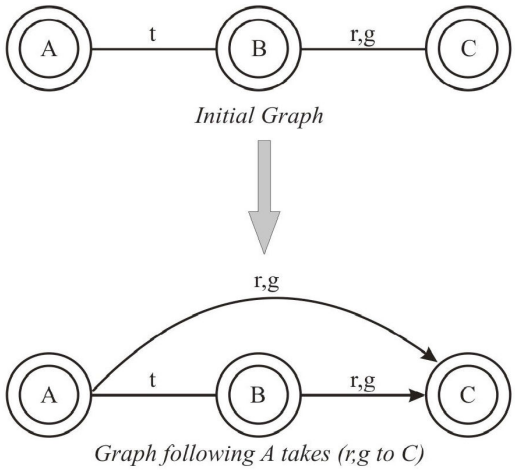


Figure 2: Example of *Take*

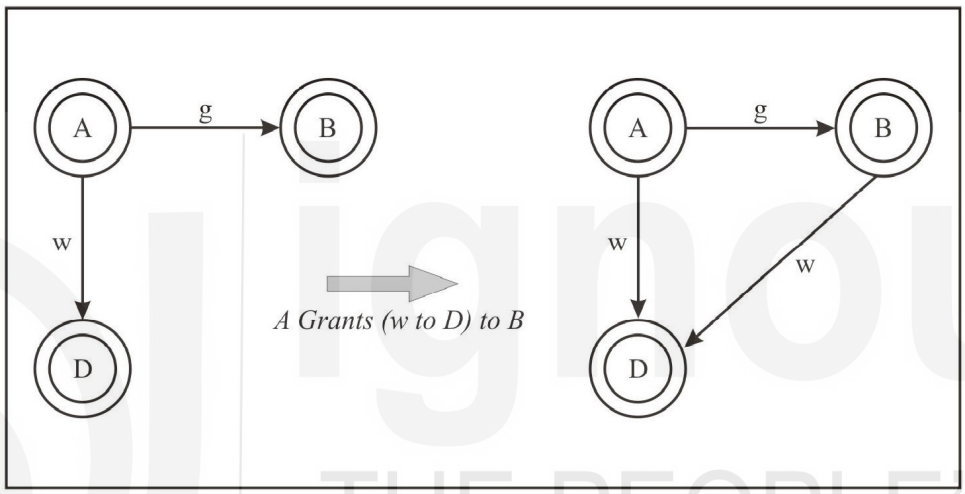


Figure 3: Example of Grant

Because the graph needs only the inclusion of arcs corresponding to non-null entries in the access matrix, it provides a compact way to present the same information given in a relatively sparse access matrix. Capability systems are thus prime candidates for this modeling technique; each arc would then represent a particular capability. Together with the protection graph, the model includes a set of rules for adding and deleting both nodes and arcs to the graph.

Two of these, corresponding to the exercise of "take" and "grant" access rights, have already been described. A "create" rule allows a new node to be added to the graph. If subject A creates a new node Y, both the node Y and an arc AY are added to the graph. The label on AY includes any subset of the possible access rights. A "remove" rule allows an access right to be removed from an arc; if all rights are removed from an arc, the arc is removed as well.

4.6.7 Multilevel Models

This type of models corresponds to the multilevel policies where data is classified into sensitivity levels and users have access according to their clearances. Because of the way they control security they have also been called *data flow* models, they control the allowed flow of data between levels.

- The Bell-La Padula model, intended to control leakage of information between levels.

- The Biba model, which controls the integrity.
- The Lattice model, which generalises the partially ordered levels of the previous models using the concept of mathematical lattices.

Bell and La Padula Model

Bell and La Padula use finite-state machines to formalise their model. They define the various components of the finite state machine, define what it means (formally) for a given system state to be secure, and then consider the transitions that can be allowed so that a secure state can never lead to an insecure state.

In addition to the subjects and objects of the access matrix model, it includes the security levels of the military security system: each subject has a authority and each object has a classification. Each subject also has a *current security level,* which may not exceed the subject's authority. The access matrix is defined as above, and four modes of access are named and specified as follows:

- **Read-only**: subject can read the object but not modify it;
- **Append**: subject can write the object but cannot read it;
- **Execute**: subject can execute the object but cannot read or write it directly; and
- **Read-write**: subject can both read and write the object.

A control attribute, which is like an ownership flag, is also defined. It allows a subject to pass to other subjects some or all of the access modes it possesses for the controlled object. The control attribute itself cannot be passed to other subjects; it is granted to the subject that created the object.

For a system state to be secure, it should hold the following two properties:

(1) The *simple security property:* No subject has read access to any object that has a classification greater than the clearance of the subject; and

(2) The **property* (pronounced "star-property"): No subject has append-access to an object whose security level is not at least the current security level of the subject; no subject has read-write access to an object whose security level is not equal to the current security level of the subject; and no subject has read access to an object whose security level is not at most the current security level of the subject.

An example restatement of the model is discussed below:

- In this case, we use security levels as (*unclassified* < *confidential* < *secret* < *top-secret*).

The security levels are used to determine appropriate access rights.

The essence of the model can be defined as follows:

- A higher-level subject (e.g., secret level) can always "*read-down"* to the objects with level which is either equal (e.g., secret level) or lower (e.g., confidential / unclassified level). So the system high (top security level in the system) has the read-only access right to all the objects in the entire system.

- A lower-level subject can never "*read-up"* to the higher-level objects, as the model suggests that these objects do not have enough authority to read the high security level information.
- The subject will have the Read-Write access right when the objects are in the same level as the subject.
- A lower-level subject (e.g., confidential level) can always "*write-up"* (**Append** access right) to the objects with level, which is either equal (e.g., confidential level) or higher (e.g., secret / top-secret level). This happens as a result of all the subjects in the higher security levels (e.g., secret / top-secret level) having the authority to read the object from lower levels.
- A higher-level subject can never "*write-down"* to the lower level objects, as the model suggests that these objects are not secure enough to handle the high security level information.

There are also a few problems associated with this model:

- For a subject of a higher-level, all of its information is considered as the same level. So there is no passing of information from this subject to any object in the lower level.
- For an environment where security levels are not hierarchical related, the model does not account for the transfer of information between these domains.
- The model does not allow changes in access permission.
- The model deals only with confidentiality but not on integrity.

Biba Integrity model

This model is a modification of the Bell-La Padula model, with emphasis on the integrity. There are two properties in the Biba model:

- **SI-property** (*simple integrity property*)**:** a subject may have write access to an object only if the security level of the subject is either higher or equals to the level of the object.
- **Integrity property:** a subject has the read-only access to an object *o*, then it can also have the write access to another object *p* only if the security level of *p* is either lower or equals to the level of *o*.

The essence of the model can be defined as follows:

- The read access policy is the same as the Bell-La Padula model.
- No information from a subject can be passed onto an object in a higher security level. This prevents the contamination of the data of higher integrity from the data of lower integrity.

The major problem associated with this model is that there isn't a practical model that can fulfil the confidentiality of the Bell-La Padula model and the integrity of the Biba model.

Information-Flow Models

The significance of the concept of information flow is that it focuses on the actual operations that transfer information between objects. Access control models

(such as the original Bell and La Padula model) represent instead the transfer or exercise by subjects of access rights to objects. Information-flow models can be applied to the variables in a program directly, while the access matrix models generally apply to larger objects such as files and processes.

The flow model, compared with the Bell and La Padula model, is relatively uncomplicated. Instead of a series of conditions and properties to be maintained, there is the single requirement that information flow obey the lattice structure as described below. An information-flow model has five components:

- A set of objects, representing information receptacles (e.g., files, program variables, bits),
- A set of processes, representing the active agents responsible for information flow,
- A set of security classes, corresponding to disjoint classes of information,
- An associative and commutative class-combining operator that specifies the class of the information generated by any binary operation on information from two classes, and
- A flow relation that, for any pair of security classes, determines whether information is allowed to flow from one to the other.

Maintaining secure information flow in the modeled system corresponds to ensuring that the actual information that flows between objects do not violate the specified flow relation. This problem is addressed primarily in the context of programming languages. Information flows from an object x to an object y whenever information stored in x is transferred directly to y or used to derive information transferred to y. Two kinds of information flow, *explicit* and *implicit,* are identified. A flow from x to y is explicit if the operations causing it are independent of the value of x (e.g., in a statement directly assigning the value of x to y). It is implicit if the statement specifies a flow from some other object z to y, but execution of the statement depends on the value of x (e.g., in the conditional assignment *if x then y: = z;* information about the value of x can flow into y whether or not the assignment is executed).

The ***lattice model*** for security levels is widely used to describe the structure of military security levels. A lattice is a finite set together with a partial ordering on its elements such that for every pair of elements there is a least upper bound and a greatest lower bound. The simple linear ordering of sensitivity levels has already been defined. Compartment sets can be partially ordered by the subset relation: one compartment set is greater than or equal to another if the latter set is a subset of the former. Classifications, which include a sensitivity level and a (perhaps empty) compartment set, can then be partially ordered as follows:

For any sensitivity levels a, b and any compartment sets c, d; the relation $(a, c) \geq (b, d)$ exists if and only if $a \geq b$ and $c \supseteq d$. That each pair of classifications has a greatest lower bound and a least upper bound follows from these definitions and the facts that the classification "unclassified, no compartments" is a global lower bound and that we can postulate a classification "top secret, all compartments" as a global upper bound. Because the lattice model matches the military classification structure so closely, it is widely used.

☞ **Check Your Progress 1**

1) What is computer security and what are the several forms of damages that can be done by the intruders to the computer systems?

.....

.....

2) Explain the role of Access Lists.

.....

.....

4.7 SUMMARY

Security is an important aspect of an OS. The distinction between security and protection is:

Security: Broad sense to refer all concerns about controlled access to facilities.

Protection: Mechanism to support security.

In this unit, we have discussed the concept of security and various threats to it. Also we have discussed various protection and security mechanisms to enforce security to the system.

4.8 SOLUTIONS / ANSWERS

Check Your Progress 1

1) Computer security is required because most organisations can be damaged by hostile software or intruders. There may be several forms of damage which are obviously interrelated. These include:

- Damage or destruction of computer systems.
- Damage or destruction of internal data.
- Loss of sensitive information to hostile parties.
- Use of sensitive information to steal items of monetary value.
- Use of sensitive information against the organisation's customers which may result in legal action by customers against the organisation and loss of customers.
- Damage to the reputation of an organisation.
- Monetary damage due to loss of sensitive information, destruction of data, hostile use of sensitive data, or damage to the organisation's reputation.

2) The logical place to store authorisation information about a file is with the file itself or with its inode.

Each element of an Access Control List (ACL) contains a set of user names and a set of operation names. To check whether a given user is allowed to perform a given operation, the kernel searches the list until it finds an entry that applies to the user, and allows the access if the desired operation is named in that entry. The ACL usually ends with a default entry. Systems that use ACLs stop searching when they find an ACL entry that applies to the subject. If this entry denies access to the subject, it does not matter if later entries would also match and grant access; the subject will be denied access. This behaviour is useful for setting up exceptions, such as everyone can access this file except people in this group.

4.9 FURTHER READINGS

1) Abraham Silberschatz and Peter Baer Galvin, *Operating System Concepts*, Wiley and Sons (Asia) Publications, New Delhi.

2) H.M.Deitel, *Operating Systems*, Pearson Education Asia, New Delhi.

3) Andrew S. Tanenbaum, *Operating Systems- Design and implementation*, Prentice Hall of India Pvt. Ltd., New Delhi.

4) Achyut S. Godbole, *Operating Systems*, Second Edition, TMGH, 2005, New Delhi.

5) Milan Milenkovic, *Operating Systems*, TMGH, 2000, New Delhi.

6) D. M. Dhamdhere, *Operating Systems – A Concept Based Approach*, TMGH, 2002, New Delhi.

7) William Stallings, *Operating Systems*, Pearson Education, 2003, New Delhi.

ignou
THE PEOPLE'S
UNIVERSITY