UNIT 1 MEMORY MANAGEMENT

Structure **Page Nos.**

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 utilisation 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 utilisation (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 organisation;
- memory protection against unauthorised access and sharing;
- to manage swapping between main memory and disk in case main storage is small to hold all processes;
- to summarise 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 utilisation 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*:

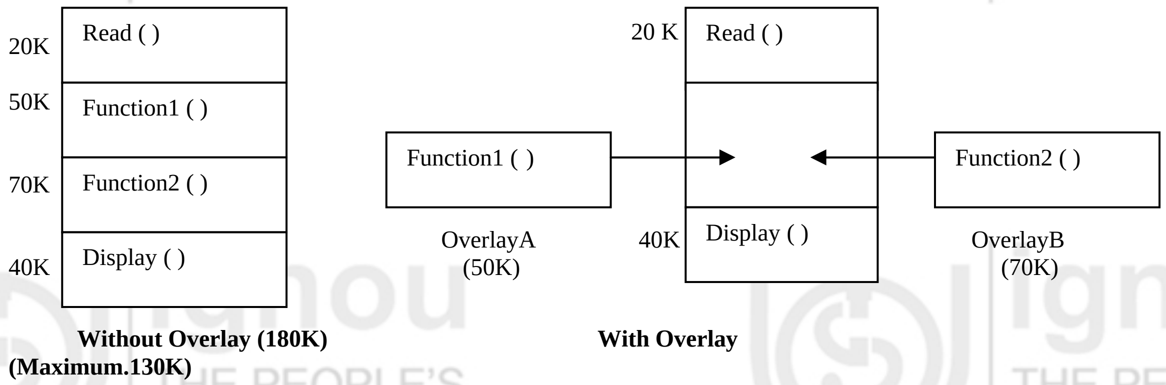


Figure 1: Example of overlay

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:

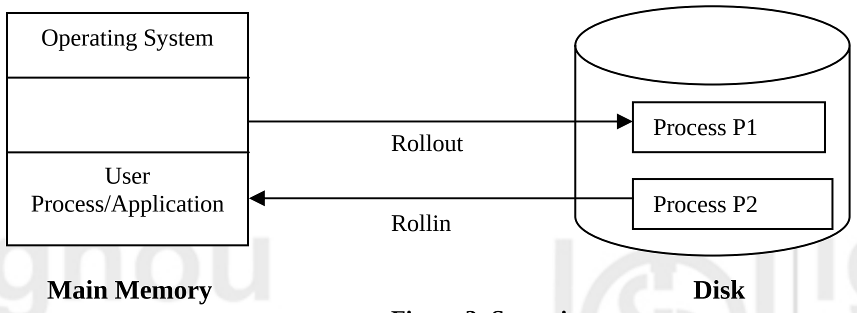


Figure 2: Swapping

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 utilisation.
- 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.

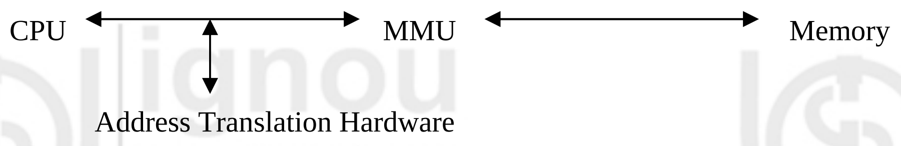


Figure 3: Role of MMU

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.

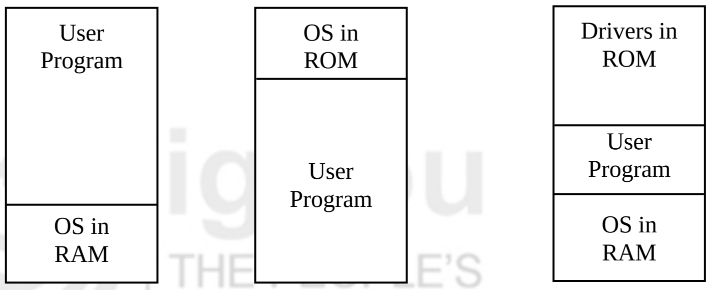


Figure 4: Single Partition System

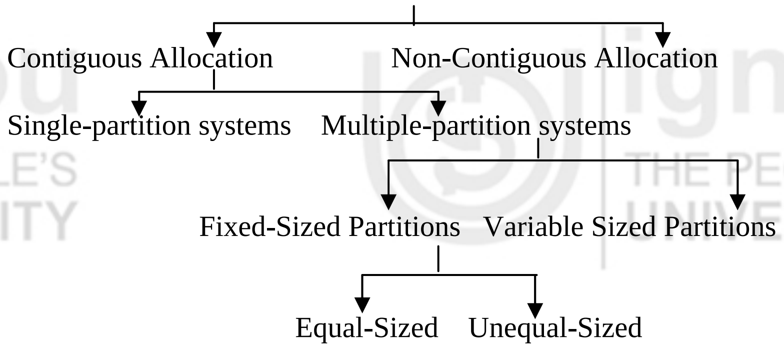
1.5 CONTIGUOUS ALLOCATION METHODS

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.

Memory Allocation



Further we will learn these schemes in next section.

Partitioned Memory allocation:

The concept of multiprogramming emphasizes on maximizing CPU utilisation 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.*

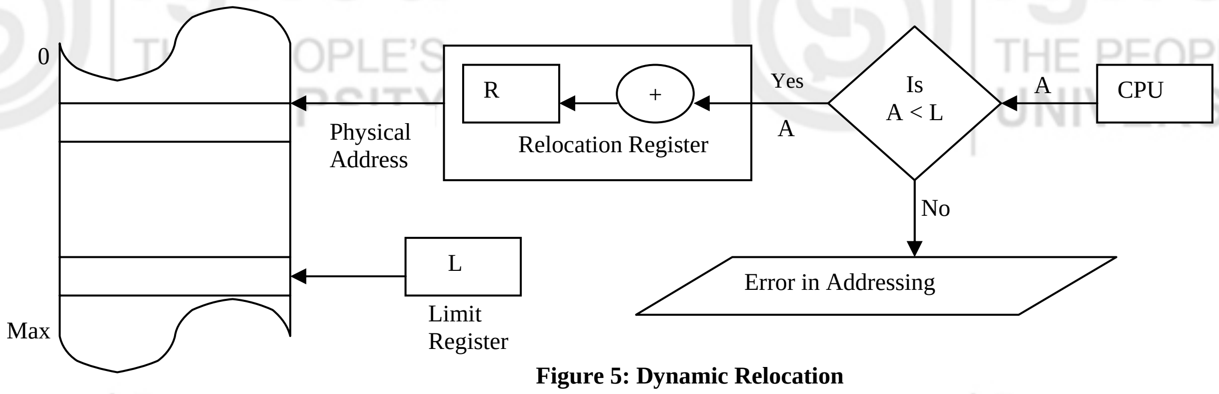


Figure 5: Dynamic Relocation

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.

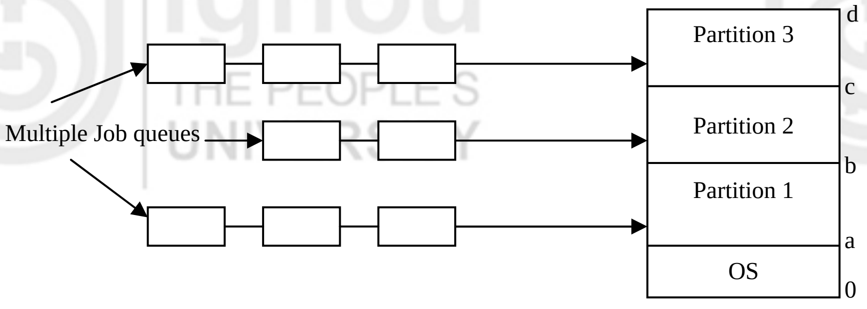


Figure 6: Multiple Partition System

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:*

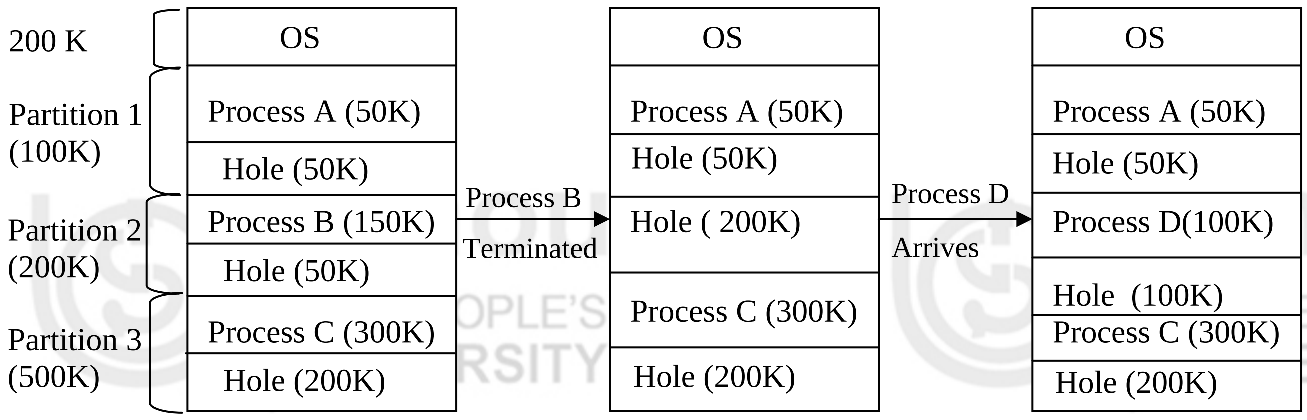


Figure 7: Fixed-sized Partition Scheme

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 utilisation 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.

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*.

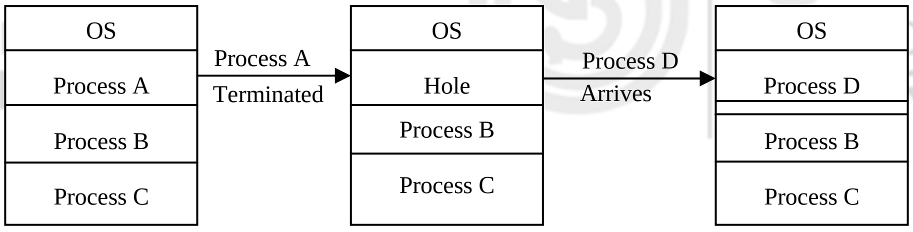


Figure 8: Variable sized partitions

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 utilisation 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 utilised.

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 utilised. 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 utilised. This mechanism is known as *Storage Compaction*, as shown in *Figure 9*.

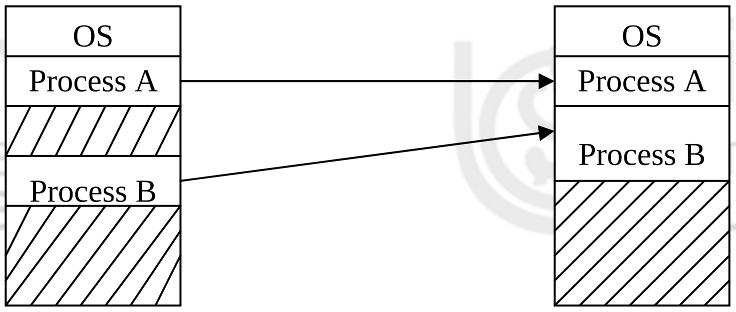


Figure 9: Storage Compaction

But storage compaction also has its limitations as shown below:

1) It requires extra overheads in terms of resource utilisation 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:

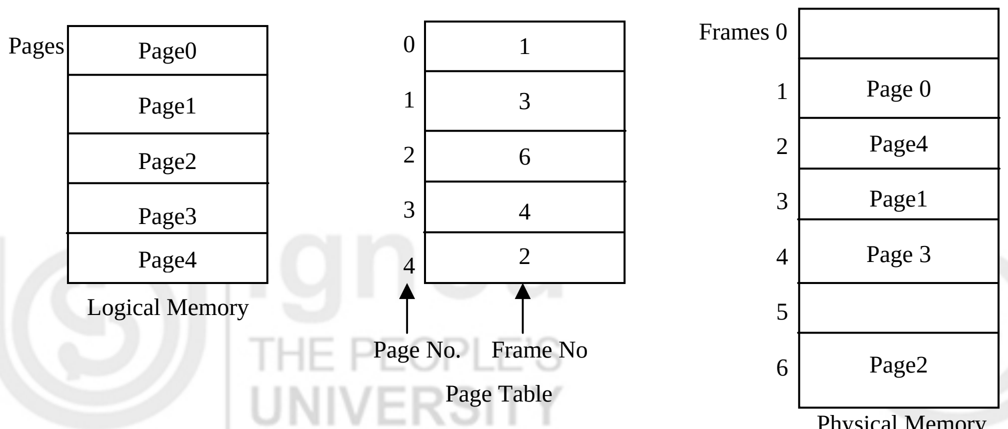


Figure 10: Principle of operation of paging

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.

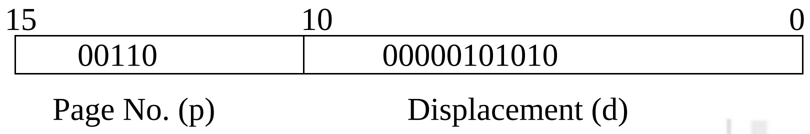
1.6.3 Hardware Support for Paging

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*.

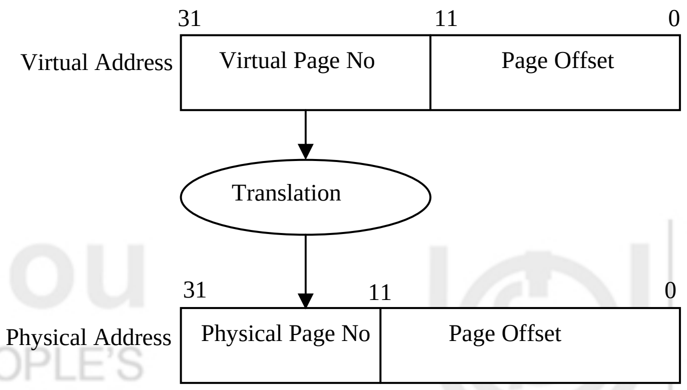


Figure 11: Address Translation scheme

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*:

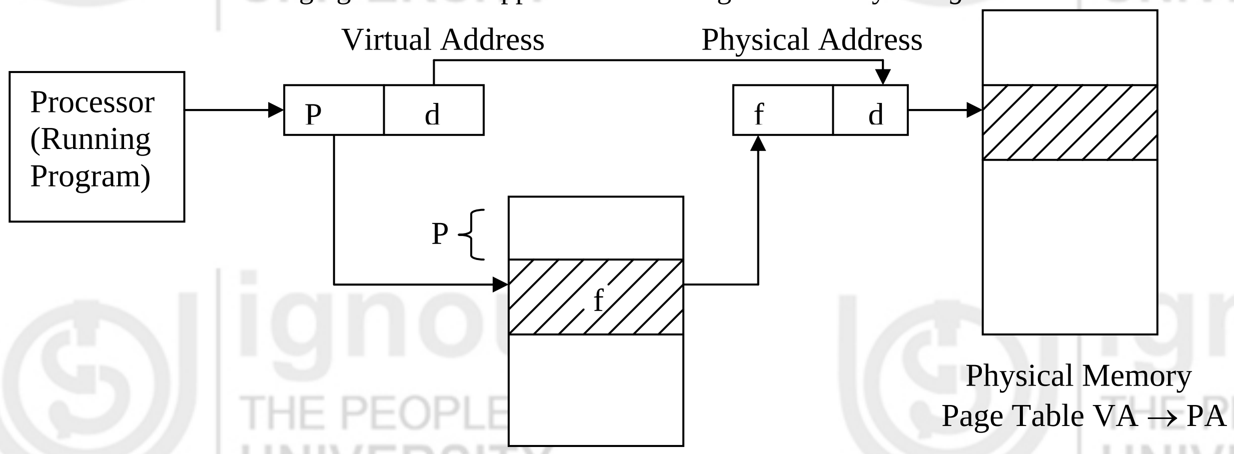


Figure 12: Direct Mapping

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*.

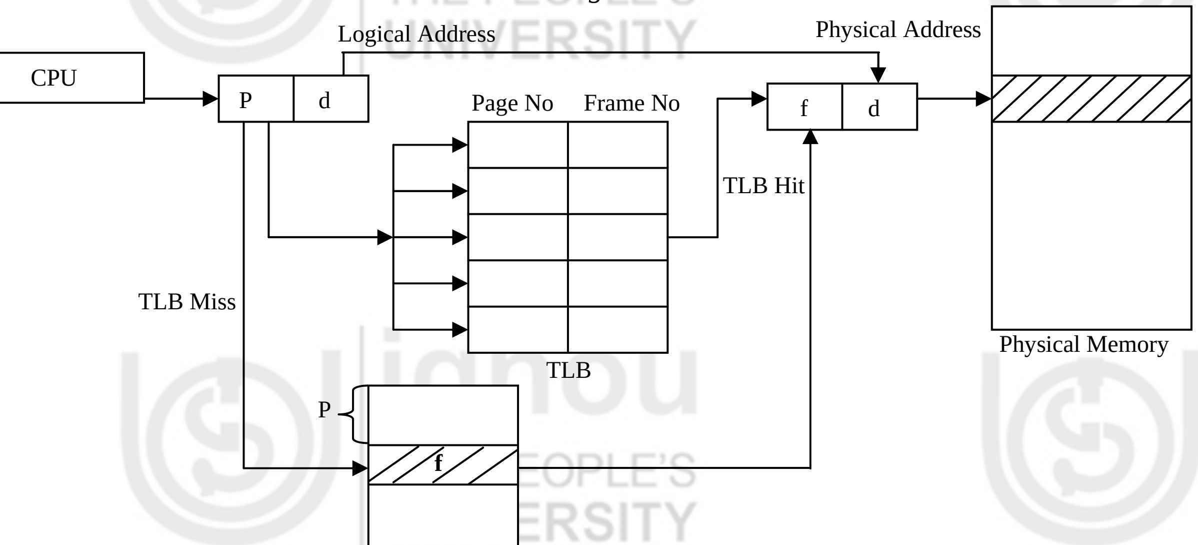


Figure 13: Combined Associative/ Direct Mapping

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 utilisation. 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 utilisation 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*.

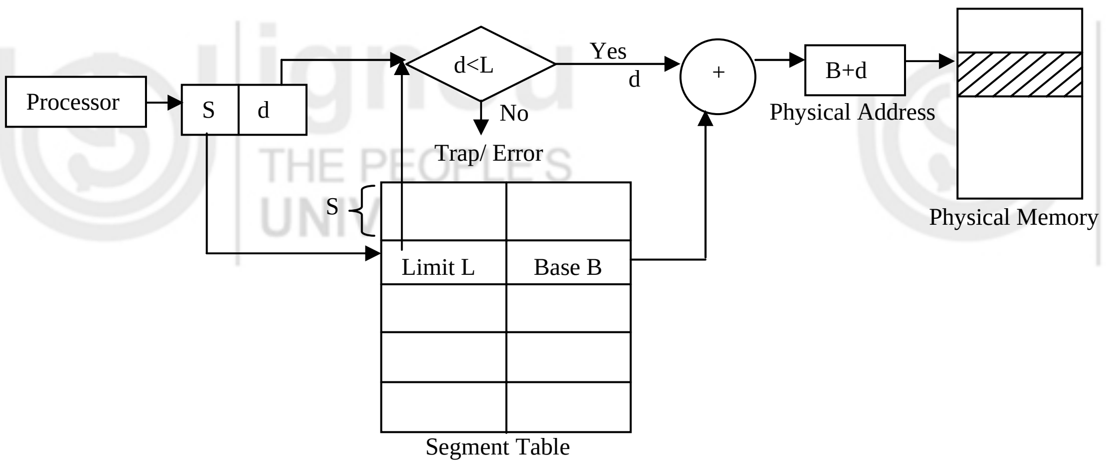
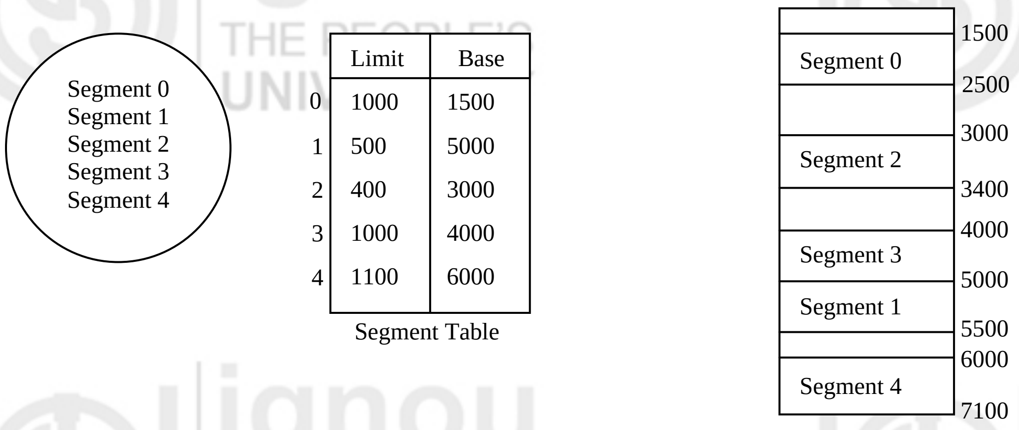


Figure 14: Address Translation

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*.



	Limit	Base
0	1000	1500
1	500	5000
2	400	3000
3	1000	4000
4	1100	6000

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 utilisation 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.