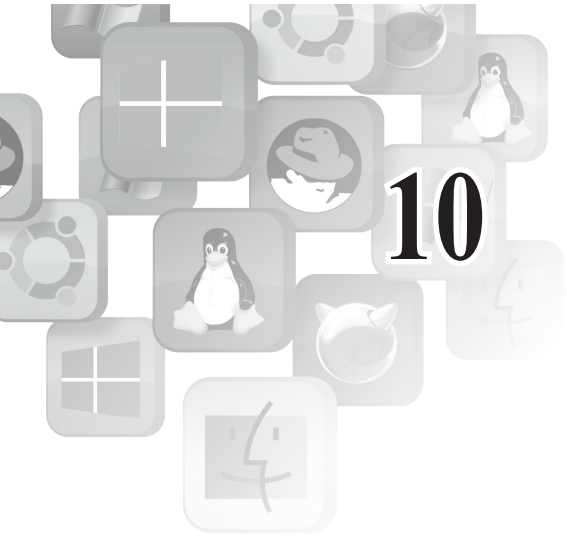


10 Basic Memory Management

10.1 INTRODUCTION

The multi-programming concept of an OS gives rise to another issue known as memory management. Process management needs the support of memory management. Memory, as a resource, needs to be partitioned and allocated to the ready processes, such that both processor and memory can be utilized efficiently. It is partitioned into two parts; one for the OS, and the other for the user area. Besides this, the user area needs to be divided into multiple parts for various user processes. This division of memory for processes needs proper management, including its efficient allocation and protection. Memory management needs hardware support also, therefore, the management technique also depends on the hardware available. The memory management issue extends from basic memory management techniques to virtual memory management, in order to meet the challenge of huge memory requirements of processes, since the size of the main memory is not sufficient. Therefore, there are two types of memory management: real memory (main memory) and virtual memory. This chapter discusses all the issues related to real memory management.

Learning Objectives

After reading this chapter, you should be able to understand:

- Static and dynamic memory allocation
- Basic concepts for memory management: logical and physical addresses, swapping, relocation, protection and sharing, fragmentation
- Fixed and variable memory partitioning
- Contiguous memory allocation
- Memory partition selection techniques: First fit, best fit, worst fit
- Non-contiguous memory allocation
- Paging concept
- Paging implementation with associative cache memory
- Page table and its structures
- Segmentation concept
- Hardware requirements for segmentation

10.2 BASIC CONCEPTS

To understand memory allocation and other memory management schemes, some basic concepts are discussed in this section. These will be used throughout this chapter.

10.2.1 Static and Dynamic Allocation

Memory allocation is generally performed through two methods: *static allocation* and *dynamic allocation*. In static allocation, the allocation is done before the execution of a process. There are two instances when this type of allocation is performed:

1. When the location of the process in the memory is known at compile time, the compiler generates an absolute code for the process. If the location of the process needs to be changed on the memory, the code must be recompiled.
2. When the location of the process in the memory is not known at compile time, the compiler does not produce an actual memory

address but generates a relocatable code, that is, the addresses that are relative to some known point. With this relocatable code, it is easy to load the process to a changed location and there is no need to recompile the code. In both cases of static allocation, size should be known before start of the execution of the process.

If memory allocation is deferred till the process starts executing, it is known as dynamic allocation. It means the process is loaded in memory initially with all the memory references in relative form. The absolute addresses in memory are calculated as an instruction in the process executed. In this way, memory allocation is done during execution of a program. Dynamic allocation also has the flexibility to allocate memory in any region.

10.2.2 Logical and Physical Addresses

To accommodate the multi-programming concept of modern OSs, dynamic memory allocation method is adopted. In this method, two types of addresses are generated. Since in dynamic allocation, the place of allocation of the process is not known at the compile and load time, the processor, at compile time, generates some addresses, known as *logical addresses*. The set of all logical addresses generated by the compilation of the process is known as *logical address space*. These logical addresses need to be converted into absolute addresses at the time of execution of the process. The absolute addresses are known as *physical addresses*. The set of physical addresses generated, corresponding to the logical addresses during process execution, is known as *physical address space*. There are now two types of memory: logical memory and physical memory. The logical memory views the memory from 0 to its maximum limit, say *m*. The user process generates only logical addresses in the range 0 to *m*, and a user thinks that the process runs in this logical address space. But the user process, in the form of logical address space, is converted into physical address space with a reference or base in memory. The memory management component of the OS performs this conversion of logical addresses into physical addresses. Thus, when a process is compiled, the CPU generates a logical address, which is then converted into a physical address by the memory management component, to map it to the physical memory.

10.2.3 Swapping

Swapping was introduced in Chapter 5 in concern with suspended processes. Swapping plays an important role in memory management. There are some instances in multi-programming when there is no memory for executing a new process. In this case, if a process is taken out of memory, there will be space for a new process. This is a good solution, but the following factors matter during the implementation.

- Where will this process reside?
- Which process will be taken out?
- Where in the memory will the process be brought back?

For the first question, the help of any secondary storage (generally, hard disk) known as backing store, is taken, and the process is stored there. The action of taking out a process from memory is called *swap-out*, and the process is known as a *swapped-out process* (see Fig.10.1). The action of bringing back the swapped-out processes into memory is called *swap-in*. A separate space in the hard disk, known as *swap space*, is reserved for swapped-out processes. The swap space should be large enough such that a swapped out process can be accommodated. The swap space stores the images of all swapped out processes. Thus, whenever a process is selected

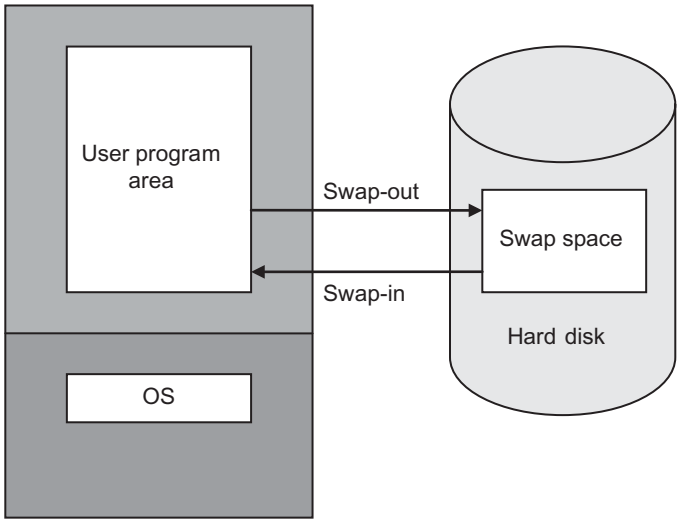


Fig. 10.1 Swapping

by the scheduler to execute, the dispatcher checks whether or not the desired process is in the ready queue. If not, a process is swapped out of memory and the desired process is swapped in.

For the second question, some of the processes that can be swapped-out are:

- In round robin process-scheduling, the processes are executed, according to their time quantum. If the time quantum expires and a process has not finished its execution, it can be swapped-out.
- In priority-driven scheduling, if a higher-priority process wishes to execute, a lower-priority process in memory will be swapped out.
- The blocked processes, which are waiting for an I/O, can be swapped out.

For the third question, there are two options to swap in a process. The first method is to swap-in the process at the same location, if there is compile time or load time binding. However, this may not be possible every time and it is inconvenient. Therefore, another method is to place the swapped-in process anywhere in the memory where there is space. But this requires the relocation, which is discussed in the next section.

Swapping incurs the cost of implementation. As discussed above, the swap space should be reserved in the secondary storage. It should be large enough to store images of the processes which are swapped out. Another cost factor is *swap time*, which is time taken to access the hard disk. Here, the transfer time from the hard disk matters. There is already some latency in data transfer from the hard disk, as compared to memory. So the swap time increases the transfer time. The transfer time is also affected by the size of the process to be swapped out from the hard disk. Hence, larger the size of the process, larger the transfer time.

Example 10.1

A process of size 200 MB needs to be swapped into the hard disk. But there is no space in memory. A process of size 250 MB is lying idle in memory and therefore, it can be swapped out. How much swap time is required to swap-in and swap-out the processes if:
Average latency time of hard disk = 10 ms
Transfer rate of hard disk = 60 MB/s

Solution

The transfer time of the process to be swapped into hard disk = 200/60 = 3.34 s. = 3340 ms
Therefore, the swap time of 200 MB process = 3340 +10 = 3350 ms
The transfer time of the process to be swapped-out from memory = 250/60 = 4.17 s. = 4170 ms
Therefore, the swap time of 250 MB process = 4170 +10 = 4180 ms
Total swap time = 3350 + 4180 = 7530 ms

10.2.4 Relocation

In a multi-programming environment, the memory is shared among multiple processes. Due to this, it may not be possible to know in advance which processes will reside in the memory and their locations. Therefore, static allocation may be difficult. Moreover, the processes may be swapped out and swapped in many times. It may not be possible to swap in the processes in the same location in the memory from where it was swapped out. It forces us to manage the memory in such a way that a static allocation cannot be given to a process. It should be possible to relocate the process to a different memory area where it gets space. The relocation process should be able to provide a new location to the process and translate all the memory references (either data reference or branch instructions) found in the process code into physical addresses, according to the current location in the memory. To translate all the memory references of the process, there must be an origin or base address in the memory. All relative addresses generated are added in this base address to get the new location in the memory.

To implement relocation, some hardware support is required. There should be a processor register which holds the base address. This register is known as the *base register* or *relocation register*. Base register holds the starting address in the main memory from where the process needs to be relocated. Along with the base register, there should be a register that stores the ending location of the process, that is, the knowledge of the limit of the process in the memory. The limit is stored in a *limit* or *bounce register* (see Fig. 10.2). Thus, with the help of base and limit registers, the process may be dynamically loaded or relocated. As soon as a process is loaded or swapped into the memory, these registers must be set by the OS.

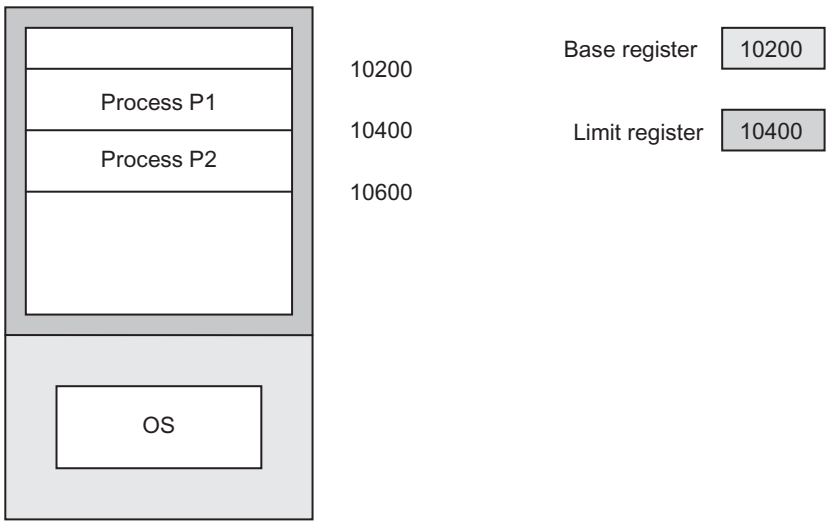


Fig. 10.2 Relocation with base and limit register

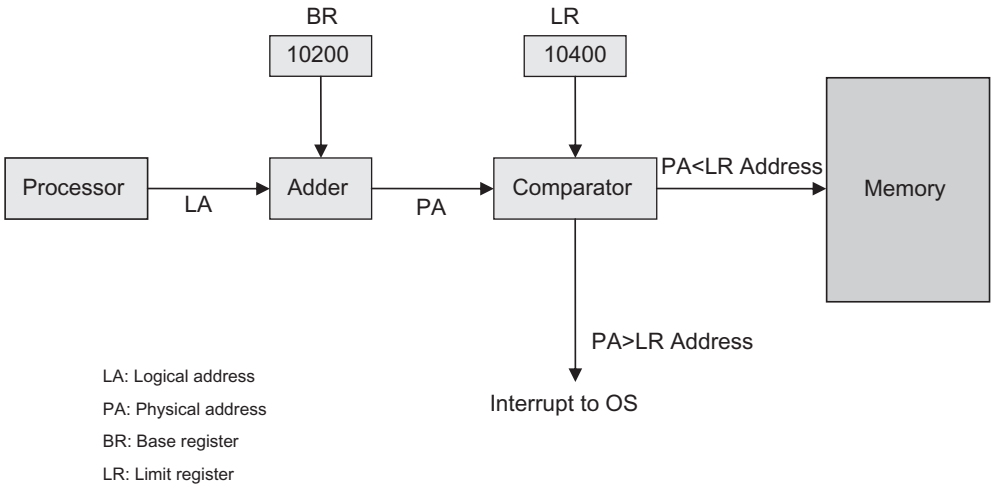


Fig. 10.3 Memory mapping using base and limit registers

When the process starts executing, relative or logical addresses are generated by the CPU. These relative addresses are first added in the base register to produce an absolute address or physical address (see Fig. 10.3). Next, the address is compared with the address stored in the limit register. If the address is less than the limit register address, the instruction execution continues. It is clear that adder (to add the base and relative address) and comparator (to compare the generated physical address with the limit register address) are also required in the hardware of a Memory Management Unit (MMU). In this way, the MMU, with the help of base and limit registers, performs the memory mapping, by converting logical addresses into physical addresses.

Example 10.2

A process is to be swapped-in at the location 20100 in memory. If logical addresses generated by the process are 200, 345, 440, and 550, what are the corresponding physical addresses?

Solution

The relocation register will be loaded with the address 20100. So adding the logical addresses to the relocation register, the corresponding physical addresses are as follows:

20100 + 200 = 20300
20100 + 345 = 20445
20100 + 440 = 20540
20100 + 550 = 20650

10.2.5 Protection and Sharing

In a multi-programming environment, there is always an issue of protection of user processes, such that they do not interfere with others and even the OS. Therefore, it is required that the process should not be able to enter the memory area of other processes or the OS area. Each process should execute within its allocated memory. It means whenever a process executes, all the address generated must be checked, so that it does not try to access the memory of other processes. The base and limit registers serve this purpose. In protection terminology, some-

times base and limit registers are also known as *lower-bound* and *upper-bound registers*, respectively. Each relative address, generated by the process during its execution, is added to the address in the base register, and its physical address is calculated. This physical address generated must be less than the address in the limit register. This provides protection to other processes, because no process can cross its boundaries set by the base and limit registers. If, somehow the physical address is more than the address in the limit register, a memory protection violation interrupt is generated. When this interrupt is processed, the kernel may terminate the process. Furthermore, protection is provided to base and limit registers also, so that they cannot be loaded or updated by any user process. The loading of these registers with addresses is a privileged operation, and is performed by the kernel only.

From the protection viewpoint, it is clear that each process has its own boundaries in the memory for its execution. However, to have a better utilization of the memory, sometimes sharing of memory spaces is also required. Although sharing is contradictory to protection, protection should not be compromised at all. Sharing of memory is predefined among the processes, but why is sharing of memory important? For instance, if some processes are using the same utility in their execution, it would be wastage of memory, if same kind of utility is given space in the memory for each process. Therefore, the better idea is to have a single copy of that utility in the memory and processes will share it, when required. Thus, memory management techniques must consider memory-sharing, but with protection.

Example 10.3

A process has relocatable code of size of 900 K. The relocation register is loaded with 40020 K and the limit register contains the address 41000 K. If the processor generates a logical address 990, where will it be located in the physical memory?

Solution

The physical address corresponding to logical address 990
= relocation register address + logical address
= 40020 + 990 = 41010
But the process will not be allocated, because it is greater than the address in the limit register, hence, violating the criterion for protection. Consequently, an interrupt will be generated.

10.2.6 Fixed and Variable Memory Partitioning

The memory space is divided into fixed or variable partitions, which will be described in detail in the subsequent sections. Fixed partitioning is a method of partitioning the memory at the time of system generation. In fixed partitioning, the partition size can be of fixed size as well as variable, but once fixed, it cannot be changed. Variable partitioning is not performed at the system generation time. In this partitioning, the number and size of the memory partition vary and are created at run time, by the OS.

10.2.7 Fragmentation

When a process is allocated to a partition, it may be possible that its size is less than the size of partition, leaving a space after allocation, which is unusable by any other process. This wastage of memory, internal to a partition, is known as *internal fragmentation*.

While allocating and de-allocating memory to the processes in partitions through various methods, it may be possible that there are small spaces left in various partitions throughout the memory, such that if these spaces are combined, they may satisfy some other process' request. But these spaces cannot be combined. This total memory space fragmented, external to all the partitions, is known as *external fragmentation*.

10.3 CONTIGUOUS MEMORY ALLOCATION

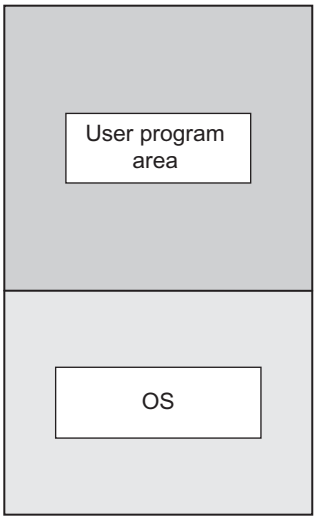


Fig. 10.4 Single partition in memory

In older systems, memory allocation is done by allocating a single contiguous area in memory to the processes. When there was multi-programming or a multi-user system, memory was divided into two partitions, - one for the OS, and the other for the user process (see Fig. 10.4). The user process in its partition has a single contiguous region of memory.

After the invention of multi-user and multi-programming systems, more processes need to be accommodated in the memory (see Fig. 10.5). Multiple processes are accommodated by having multiple partitions in the memory. However, multiple partitioning has an effect on the OS partition. It is still present in the memory. The OS area is generally placed in the lower memory addresses. It can be in high memory also, but interrupt vector is often in lower memory addresses.

The contiguous allocation method was implemented by partitioning the memory into various regions. The memory can be partitioned, using either fixed memory or variable memory partitioning.

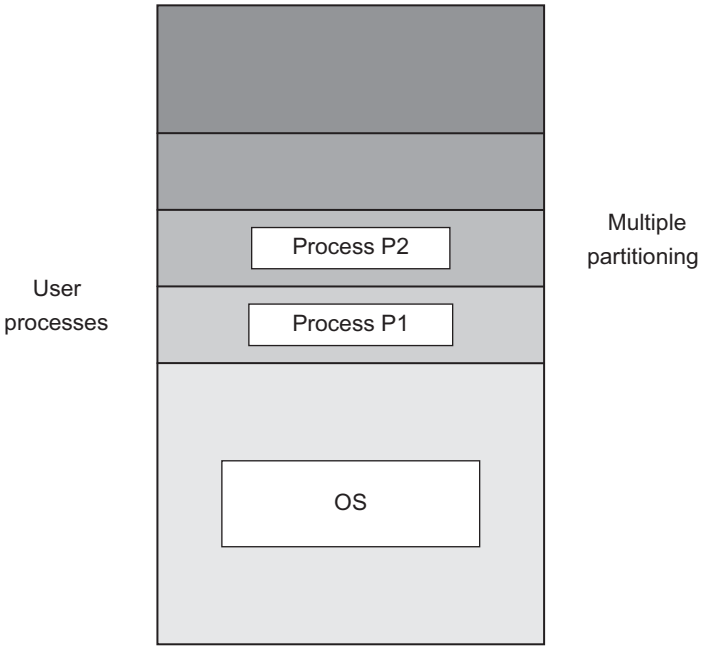


Fig. 10.5 Multiple partitioning of memory

In this method, a process is allocated a contiguous memory in a single partition. Thus, the memory partition, which fits the process, is searched and allocated. The memory partition, which is free to allocate, is known as a *hole*. Thus, an appropriate hole is searched for allocating it to a process. When the process terminates, the occupied memory becomes free and the hole is available again. Initially, all the holes are available. When all the holes are allocated to the processes, the remaining processes enter wait state. As soon as a process terminates, a hole becomes free, and is allocated to a waiting process.

10.3.1 Contiguous Allocation with Fixed Partitioning

Fixed partitioning is a method of memory partitioning at the time of system generation. The partition can be of fixed as well as variable size, but once fixed, it cannot be changed. If there is a need to change the partition size, the OS must be generated again with modified partitions. It means, the partitions, once fixed, are static in nature. To allocate memory to the processes in partitions, the OS creates a table to store the information, regarding the partitions in the memory. This table is called the *partition description table* (PDT). The structure of the table is shown in Table 10.1.

The contiguous allocation method with fixed partitioning was adopted by earlier systems like IBM mainframe OS/360, which was called OS/MFT (Multi-programming with Fixed number of Tasks). In this method, the long-term scheduler performs job scheduling, and decides which process is to be brought into the memory. It then finds out the size of the process to be loaded and requests the memory manager to allocate a hole in the memory. The memory manager uses one of the allocation techniques (discussed later) to find a best match for the process. After getting a hole, the scheduler places the process in the allocated partition. Next, it enters the partition ID in the PCB of the process, and then PCB is linked to the chain of the ready queue. Memory manager marks the status of the partition as 'Allocated'. Initially, the status of all memory partitions is 'Free'. As soon as a process terminates, the OS updates the allocation status (in the PDT) of the partition, where the terminated process resided. Thus, with the use of PDT, a list of available holes is obtained, and holes are allocated to the processes, based on their memory requirements.

What should be the size of the memory partitions? One approach is memory partitions, which are of equal size, as shown in Fig 10.6. But there are disadvantages to this approach. One is that a process may be too big to fit into the partition. Another is that a small process, occupying the entire partition, leads to the problem of internal fragmentation.

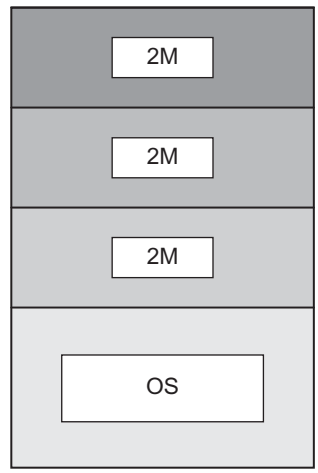


Fig. 10.6 Fixed equal sized partitioning based contiguous memory allocation

Table 10.1 Partition description table

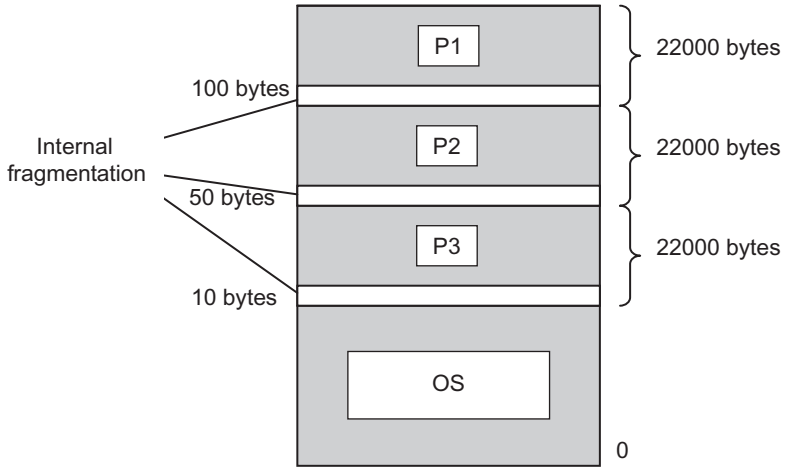
Partition ID	Starting address	Size	Allocation status

Example 10.4

Three processes P1, P2, and P3 of size 21900, 21950, and 21990 bytes, respectively, need space in the memory. If equal-sized partitions of 22000 bytes are allocated to P1, P2, and P3, will there be any fragmentation in this allocation?

Solution

After allocating the partitions to the processes, the leftover space in each partition is estimated by the difference between partition size and process size. Hence,
The leftover space in the first partition = 22000 – 21900 = 100 bytes
The leftover space in the second partition = 22000 – 21950 = 50 bytes
The leftover space in the third partition = 22000 – 21990 = 10 bytes.
This leftover space in each partition is nothing but internal fragmentation, as shown in the following figure:



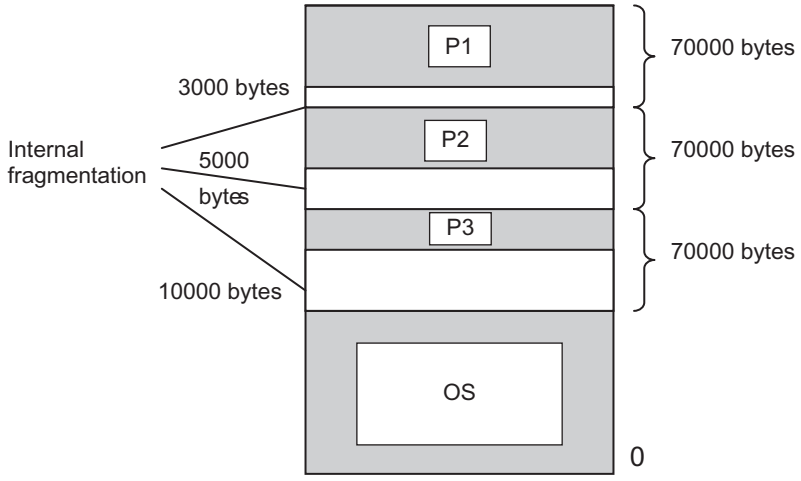
Example 10.5

Three processes P1, P2, and P3 of size 67000, 65000, and 60000 bytes, respectively, need space in the memory. If partitions of equal size, that is, 70000 bytes, are allocated to P1, P2, and P3, will there be any fragmentation in this allocation?

Solution

After allocating the partitions to the processes, the first, second, and third partitions are left with 3000 bytes, 5000 bytes, and 10000 bytes, respectively. This leftover space in each partition is internal fragmentation, as shown in the figure.

In the above two examples, fixed memory partitioning, with equal partitioning size, leads to wastage of a large space in the memory. The second approach is to use unequal-sized partitions in the memory. Unequal-sized partitions can be chosen, such that smaller to bigger size processes can be accommodated, thereby, wasting less memory, as shown in Fig. 10.7. Keeping in view the average size of the processes, the size of partitions in the memory can be fixed.



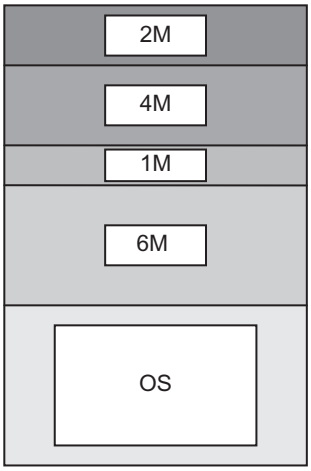


Fig. 10.7 Fixed unequal-sized partitioning-based contiguous memory allocation

Although the second approach has improved the performance of memory allocation, overall fixed partitioning suffers from disadvantages, due to which this method has become obsolete today. The following are some drawbacks of this method:

- The degree of multi-programming is limited by the number of partitions fixed.
- There may be some processes whose memory requirement is not known in advance. In this case, unequal-size partitioning may also cause internal fragmentation. Moreover, due to internal fragmentation, there may be memory space scattered in various partitions. If these memory spaces are combined, it can be allocated to a process. But it is not possible in fixed partitioning method. This results in external fragmentation also. Thus, fixed partitioning method suffers from both types of fragmentation.

Example 10.6

Three processes P1, P2, and P3 of size 19900, 19990, and 19888 bytes, respectively, need space in memory. If partitions of equal size, that is, 20000 bytes, are allocated to P1, P2, and P3, will there be any fragmentation in this allocation? Can a process of 200 bytes be accommodated?

Solution

After allocating the partitions to the processes, the first, second, and third partitions are left with 100 bytes, 10 bytes, and 112 bytes, respectively. This leftover space in each partition is internal fragmentation.

The total space left = 100 + 10 + 112 = 222 bytes

Process of 200 bytes cannot be accommodated, even if the total space left is more than 200 bytes. This is because the space left is not contiguous. Hence, this partitioning also leads to external fragmentation.

10.3.2 Contiguous Allocation with Dynamic/Variable Partitioning

Contiguous allocation with fixed partitioning suffers from drawbacks, as discussed above. To overcome these drawbacks, contiguous allocation with variable partitioning was devised. It is also known as dynamic partitioning. Instead of having static partitions, the memory partition will be allocated to a process dynamically. In other words, the number and size of partitions are not fixed at the time of system generation. They are variable and are created at run time by the OS. The procedure for memory allocation in this method is the same as fixed partitioning. The only difference is that partitions for processes are created at run time, when they are brought to the ready queue. PDT is maintained in variable partitioning as well. In the same way as discussed in fixed partitioning, a hole is searched for in the process. Initially, there is only a single large hole, that is, a partition allocated to user processes. The first process is allocated the required memory, out of this large hole, and the rest of the memory is returned. It means there are two partitions of the original hole: one allocated to the process and the other partition, which is available. In this way, the processes are allocated the required space in the hole and variable-sized partitions are produced. Two contiguous free holes can be combined into a single partition.

The IBM mainframe was developed as Multi-programming with Variable number of Tasks (OS/MVT). The advantage in variable partitioning is that the process is given exactly as much space as it requires, reducing the internal fragmentation faced in fixed partitioning. Although variable partitioning reduces memory wastage, it can still cause fragmentation. Eventually, there are small holes in the memory partitions. These holes cannot be allocated to any process, because they are not contiguous, and hence, cause external fragmentation.

Example 10.7

Let us understand dynamic partitioning with an example. Consider three processes P1, P2, and P3, as shown in Fig. 10.8. Initially, 120M hole is available in the memory. P1 of 30M consumes memory from the first hole and leaves 90M space. Similarly, P2 and P3 are allocated 40M and 48M of memory, respectively, leaving a hole of 2M. In Fig. 10.8(e), P2 releases the memory, leaving a hole of 30M. In Fig. 10.8(f), P4 arrives and acquires 28M, leaving a hole of 2M. In Fig. 10.8(g), P3 releases the memory, leaving a hole of 48M. In Fig. 10.8(h), P1 arrives again, and now requires 30M space. Since P4 resides in the space, which was earlier allocated to P1, it will consume 30M from the hole left by P3.

10.3.3 Compaction

External fragmentation in dynamic partitioning can be reduced, if all the small holes formed during partitioning and allocation, are compacted together. Compaction helps to control memory wastage, occurring in dynamic partitioning. The OS observes the number of holes in the memory and compacts them after a period, so that a contiguous memory can be allocated for a new process. The compaction is done by shuffling the memory contents, such that all occupied memory region is moved in one direction, and all unoccupied memory region in the other direction. This results in contiguous free holes, that is, a single large hole, which is then allocated to a desired process.

Compaction, however, is not always possible. One limitation is that it can be applied, only if the memory is relocated dynamically at the execution time, so that it is easy to move one process from one region to another. Another limitation is that compaction incurs cost, because it is time consuming and wastes CPU time.

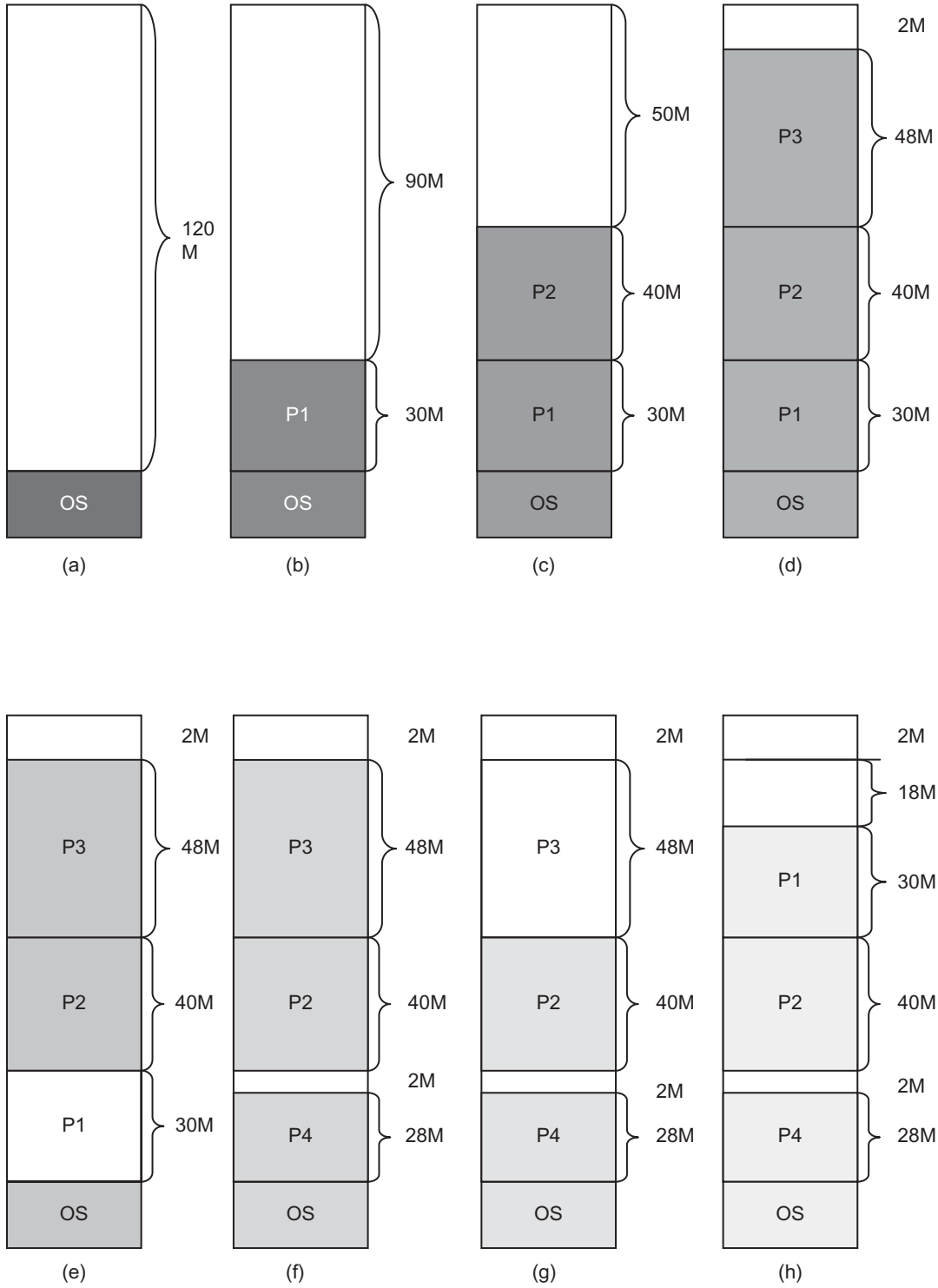


Fig. 10.8 Variable partitioning

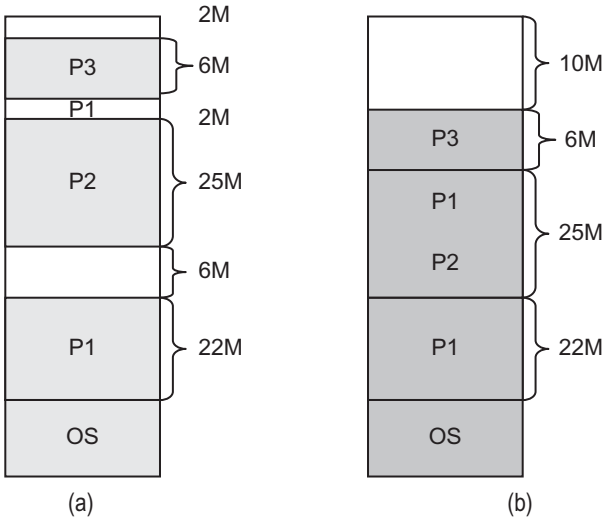


Fig. 10.9 Compaction

Example 10.8

Fig. 10.9(a) shows the state of three processes in memory. Here, the wastage of memory is 6M + 2M + 2M = 10M. No process can be allocated to this space. However, through compaction, that is, by merging these memory spaces, 10M space can be obtained contiguously, and can be used for the allocation of a process, which is less than or equal to 10M (see Fig. 10.9 (b)). For compaction, the processes should be relocated to different addresses. This may consume time, thereby making this method costly.

The contiguous allocation method, with fixed as well as variable partitioning method, suffers from fragmentation and wastes memory, as discussed above. The alternate method is non-contiguous partitioning, which will be discussed later.

10.3.4 Memory Allocation Techniques

Memory allocation techniques are algorithms that satisfy the memory needs of a process: they decide which hole from the list of free holes must be allocated to the process. Thus, it is also known as partition selection algorithms. In fixed partitioning with equal-sized partitions, these algorithms are not applicable, because all the partitions are of the same size and therefore, it does not matter which partition is selected. These algorithms, however, play a great role in fixed-partitioning with unequal-sized partitions and in dynamic partitioning, in terms of system performance and memory wastage. There are primarily three techniques for memory allocation.

First-fit Allocation

This algorithm searches the list of free holes and allocates the first hole in the list that is big enough to accommodate the desired process. Searching is stopped when it finds the first-fit hole. The next time, searching is resumed from that location. The first hole is counted from this last location. In this case, it becomes the *next-fit* allocation. The first-fit algorithm does not take care of the memory wastage. It may be possible that the first-fit hole is very large, compared to the memory required by the process, resulting in wastage of memory.

Best-fit Allocation

This algorithm takes care of memory storage and searches the list, by comparing memory size of the process to be allocated with that of free holes in the list. The smallest hole that is big enough to accommodate the process is allocated. The best-fit algorithm may be better in terms of wastage of memory space, but it incurs cost of searching all the entries in the list. This is an additional overhead. Moreover, it also leaves small memory holes, causing internal fragmentation.

Worst-fit Allocation

This algorithm is just reverse of the best-fit algorithm. It searches the list for the largest hole. This algorithm in its first perception seems that it is not a good algorithm, in terms of memory. But it may be helpful in dynamic partitioning, because the large holes, leftover due to this algorithm, may be allocated to the processes that fit. However, this algorithm incurs overhead of searching the list for the largest hole. It may also leave small holes, causing internal fragmentation.

Example 10.9

Consider the memory allocation scenario as shown in Fig. 10.10. Allocate memory for additional requests of 4K and 10K (in this order). Compare the memory allocation, using first-fit, best-fit, and worst-fit allocation methods, in terms of internal fragmentation.

Solution

First-fit allocation: It allocates the first hole in the list that is big enough. Hole of 10K is allocated to the process of 4K, leaving a hole of 6K in memory. The next request is of 10K. So, the next hole in the list is of 5K and it cannot accommodate the process. Hence, the next available hole of 15K is allocated, leaving a hole of 5K (see Fig. 10.11). It leaves fragmentation of 6K +5K = 11K.

Best-fit allocation: It allocates the smallest hole that is big enough. Hole of 5K is allocated to the process of 4K, leaving a hole of just 1K. The next request is of 10K. After comparing the size of all the holes, hole of 10K is allocated. It leaves fragmentation of 0K +1K = 1K (see Fig. 10.12).

Worst-fit allocation: It allocates the largest hole in the list. For the process of 4K, the largest hole of 22K is allocated. This leaves a hole of 18K in the memory. The next request is of 10K.

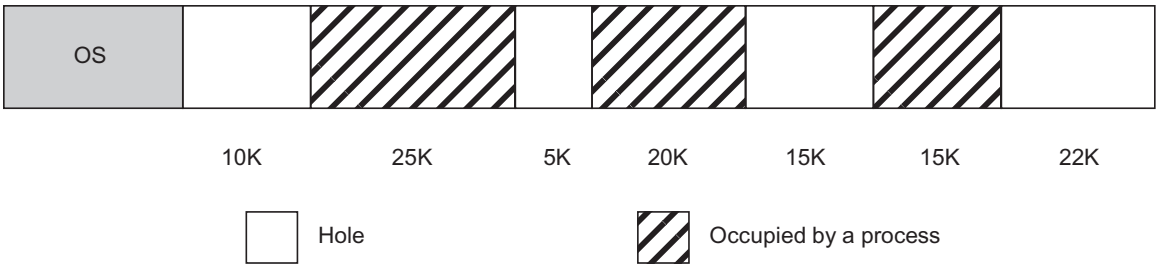


Fig. 10.10 Example memory allocation scenario

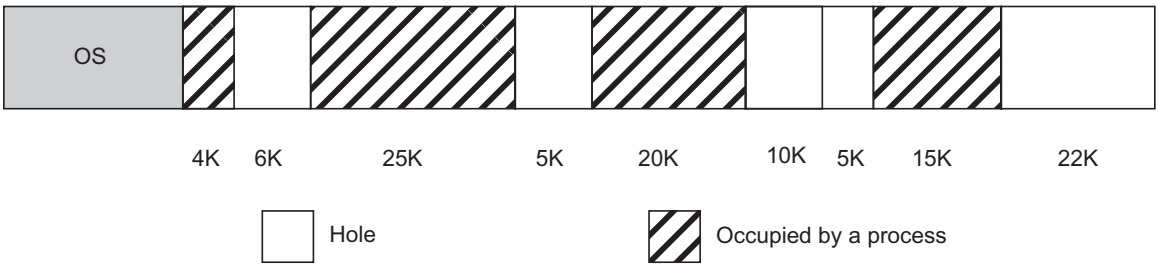


Fig. 10.11 First-fit allocation for Example 10.9

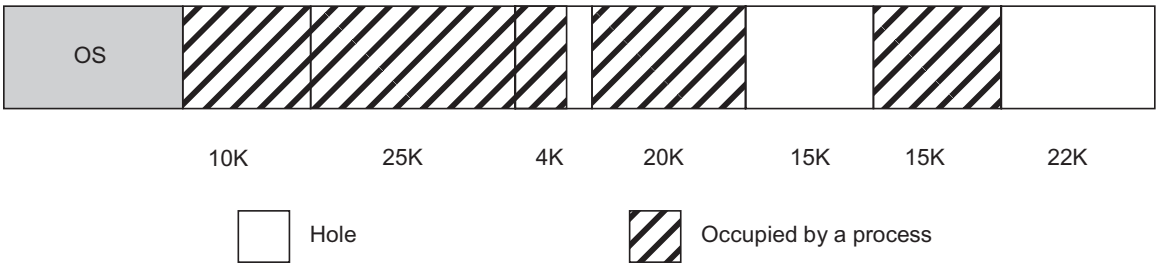


Fig. 10.12 Best-fit allocation for Example 10.9

Comparing the size of all holes, the 18K hole is the largest hole in the list. Hence, it is allocated to the process. It leaves fragmentation of 22K − 4K − 10K = 8K (see fig. 10.13).

By comparing all the algorithms on the basis of internal fragmentation, the best-fit allocation is the best method, as it wastes least memory. The worst-fit is better than the first-fit allocation, as it provides more holes of appropriate sizes for future requests. Also, memory wastage is comparatively less.

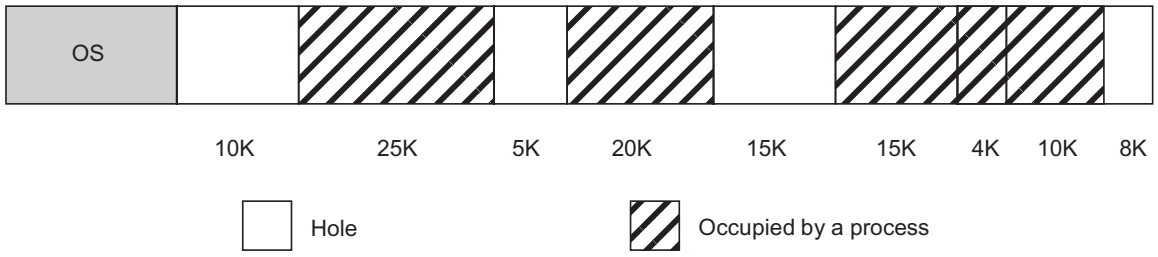


Fig. 10.13 Worst-fit allocation for Example 10.9

10.4 BUDDY SYSTEM

Both types of memory partitioning have drawbacks. Buddy system is a technique that is developed as a compromise. A buddy system allocates memory from a fixed-size block of memory. The allocated block will be of power-of-two size. Therefore, if there is a request of size *s* for memory allocation, the smallest block, greater than or equal to *s*, is allocated. Initially, the segment is considered as a single block, and may be allocated, if the request can occupy the whole segment. The block will be allocated, if the wastage of space is very less. If the request is of small size, then the segment is first divided into two parts, known as buddies. One of the buddies is then considered for allocation. If the size of the buddy is large, compared to the request, then this buddy is again divided into two buddies. This process goes on, until the smallest buddy or block, enough to satisfy the request, is generated. In this way, the request for a block is satisfied, by dividing the segment into buddies of power-of-two sizes. When the memory is released by a process, a buddy can be combined with another buddy, if it is free. This is known as *coalescing*, that is, merging the buddies, which are unallocated, forming a larger block or buddy. For example, in Fig. 10.14, initially, there is a fixed-size segment of 512K. A request for 100K block is received. Therefore, 512K segment is split into two buddies, A and B, of 256K each. Then, one of the buddies, say *A*, is further divided into two more buddies, *C* and *D*, of 128K each. In this way, block *C* is allocated to the request. After this, suppose a request of 240K block is received. *B* is allocated for this request, as its size is 256K. After some time, *A* releases the memory. Another request of 240K block is fulfilled by coalescing *A* (128K) and *B* (128K), making a larger block to accommodate the new request.

The buddy system is efficient, compared to fixed and variable partitioning. The good feature of this method is coalescing of blocks. UNIX kernel memory allocation is done using the buddy system. However, this is not used in any contemporary OS, as this may also suffer from fragmentation, if the system rounds up the size of a block

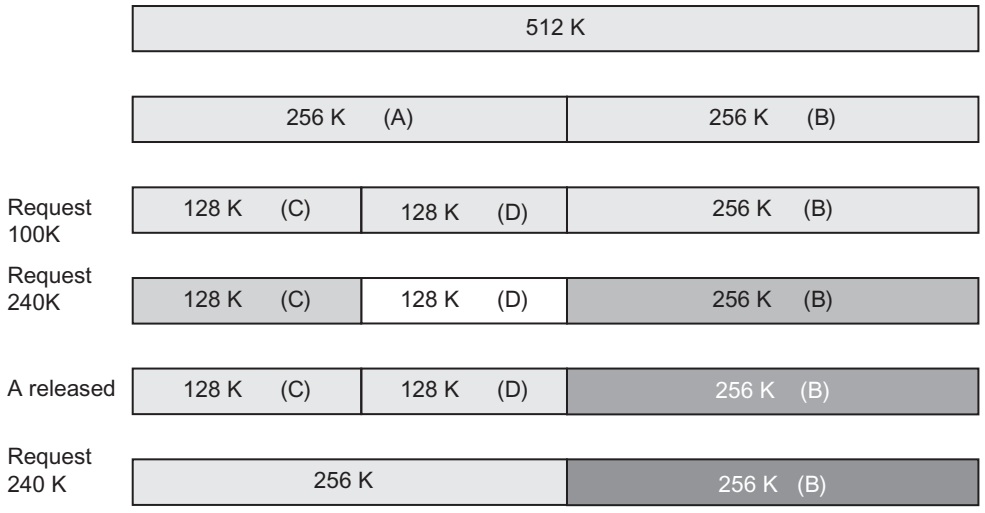


Fig. 10.14 Buddy System

to the next higher power of 2. For example, a request for 65K, which cannot be satisfied with a block of size 2^5, will be satisfied with a block of 2^6, but there will be a large wastage of memory. Therefore, for any modern OS, paging and segmentation are used for memory allocation. These methods will be discussed later in this chapter and the next chapter.

10.5 NON-CONTIGUOUS MEMORY ALLOCATION

Contiguous memory allocation method suffers from many drawbacks. Both types of this method cause internal as well as external fragmentation. Moreover, the fragmentation is also in the secondary storage, where the swapped-out processes of variable sizes need space. Thus, the contiguous allocation method wastes a lot of memory space. These drawbacks lead us to the non-contiguous allocation method. In this method, the holes do not need to be contiguous. They may be scattered in the memory, and can be allocated to a process. The non-contiguous memory allocation is also classified as fixed partitioning and variable partitioning. The former is known as paging, and the latter is known as segmentation. These are discussed in the subsequent sections.

10.6 PAGING CONCEPT

The first non-contiguous allocation method is paging, where memory is divided into equal-size partitions. The partitions are relatively smaller, compared to the contiguous method. They are known as *frames*. Since the idea is to reduce external fragmentation, the logical memory of a process is also divided into small chunks or blocks of the same size as frames. These chunks are called *pages* of a process. In this way, whenever a frame in memory is allocated to a page, it fits well in the memory, thereby eliminating the problem of external fragmentation. Moreover, the hard disk is also divided into *blocks*, which are of same size as frames. Thus, paging is a logical concept that divides the logical address space of a process into fixed-size pages, and is implemented in physical memory through frames. All the pages of the process to be executed are loaded into any available frame in the memory.

Let us understand the paging concept in detail and the benefits of a non-contiguous method with an example.

Example 10.10

In Fig. 10.15(a), it is given that there are 10 free frames in the memory. There are four processes P1, P2, P3, and P4, consisting of three, four, two, and five pages, respectively. All three pages of P1 have been allocated to frames (see Fig.10.15 (b)). Similarly, all the pages of P2 and P3 have been allocated, as shown in Fig. 10.15 (c) and 10.15 (d). Now only one frame is free in the memory, whereas P4 requires 5 frames. After some time, P2 finishes its execution and therefore, releases memory. Therefore, frames 3 to 6 are free now, making the total number of free frames as five. These five frames, though non-contiguous, are allocated to P4. Four pages are allocated frames 3, 4, 5, and 6 and one page gets frame 9 (see Fig 10.15 (f)). Suppose after some time, P1 releases page 1, P4 releases page 2, and P3 releases page 1, as shown in (see Fig. 10.15(g)), as they are swapped out into the disk. At this time, another process, say P5, is introduced in the system with five pages, but needs only three pages to be accommodated in the memory. As shown in the figure, three non-contiguous frames are available. These will be then allocated to P5 (see 10.15 (h)), through paging.

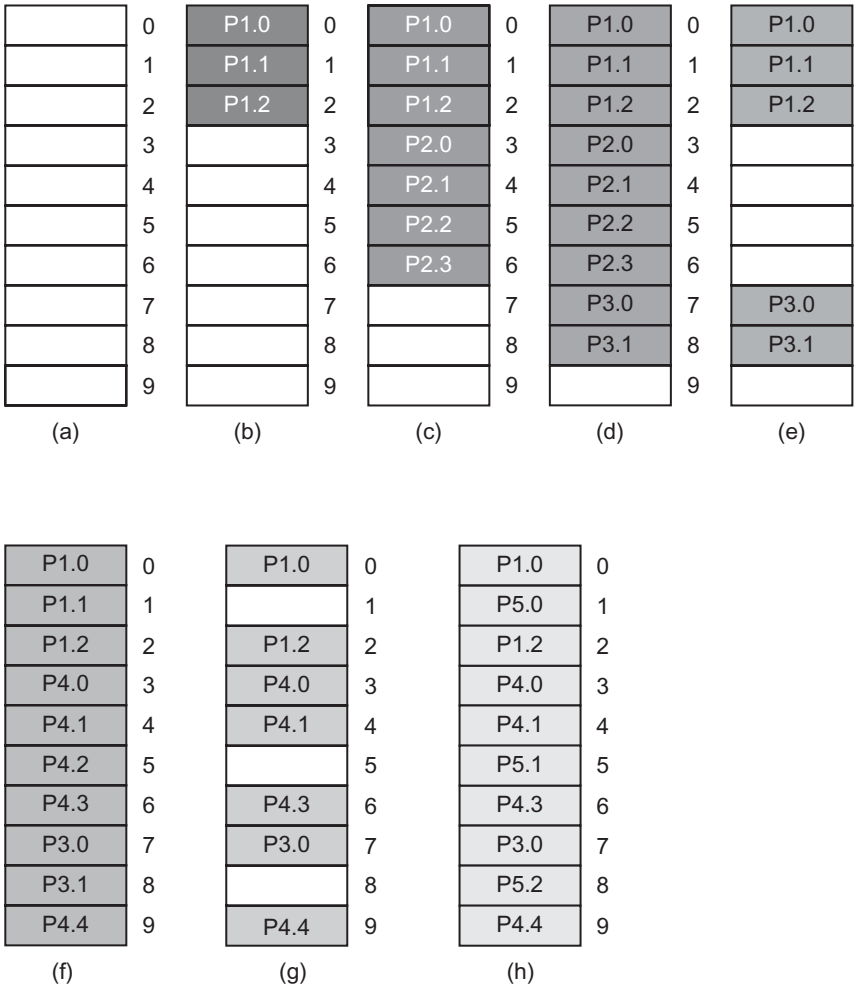


Fig. 10.15 Paging concept example

One can realize that a logical address in the paging concept has two parts: a page number and its displacement or offset in the page. Consequently, this logical address is converted into a physical address. For that, the start address of the page in the memory, that is, address in the base register, must be known. In this way, every page will have one start address in the memory. But a single base register will not suffice for this purpose, unlike contiguous allocation. Instead of a base register for every page, the start addresses of pages are stored in the form of a table, known as a *page table*. A page table is a data structure used to store the base addresses of each page in the process, that is, the entry in the page table indicates the frame location of pages in the memory. Thus, the paging concept involves the logical memory division into pages, page table to keep the base addresses of pages, and physical memory divided into frames (see Fig. 10.16).

Example 10.11

Design page tables for all the processes at the time instant of Fig. 10.15(h).

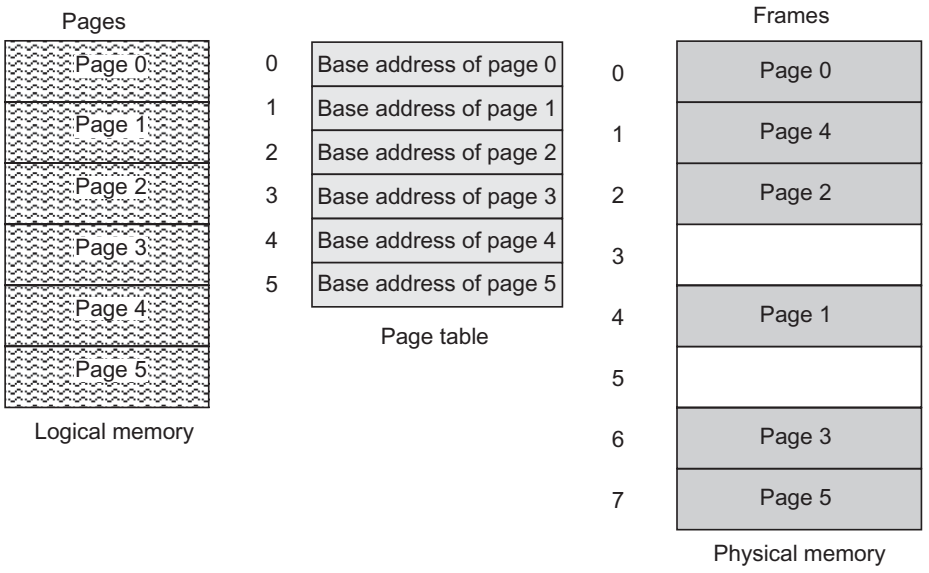
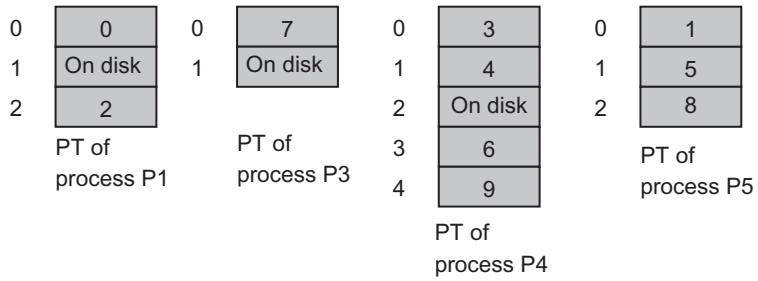


Fig. 10.16 Paging concept



Solution

After reaching the start location of a page, its displacement or offset is added to map the complete logical address into physical address and then, the particular instruction in a page can be executed. Thus, the processor, in case of paging, generates the logical address in the form:

(Page number p, Offset d)

The logical address is converted into a physical address by the MMU. The hardware must be updated such that it must know how to access the page table for conversion. The steps for logical to physical address conversion (see Fig. 10.17) are as under:

1. The processor generates a two dimensional logical address, (p, d).
2. The page number p is extracted from the logical address, and is used as an index in the page table.
3. The base address b, corresponding to the page number, is retrieved.
4. b is added to the offset d to get the corresponding physical address.

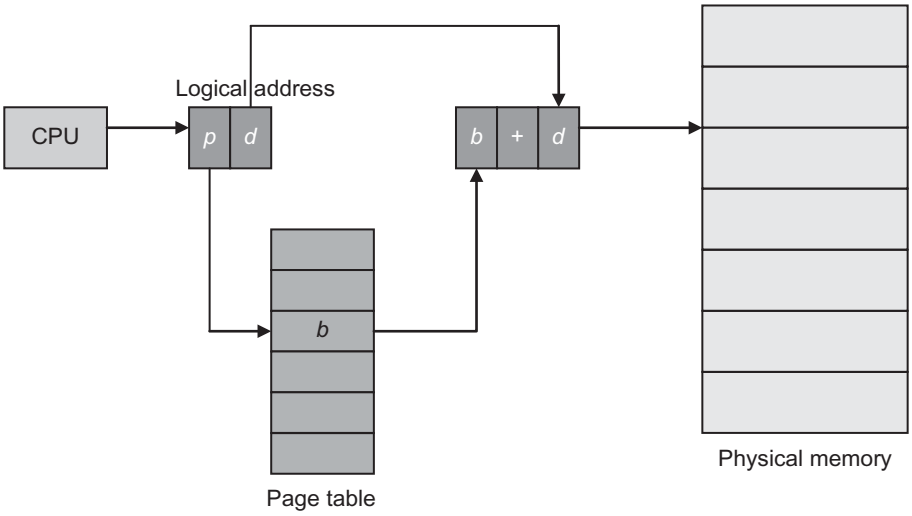
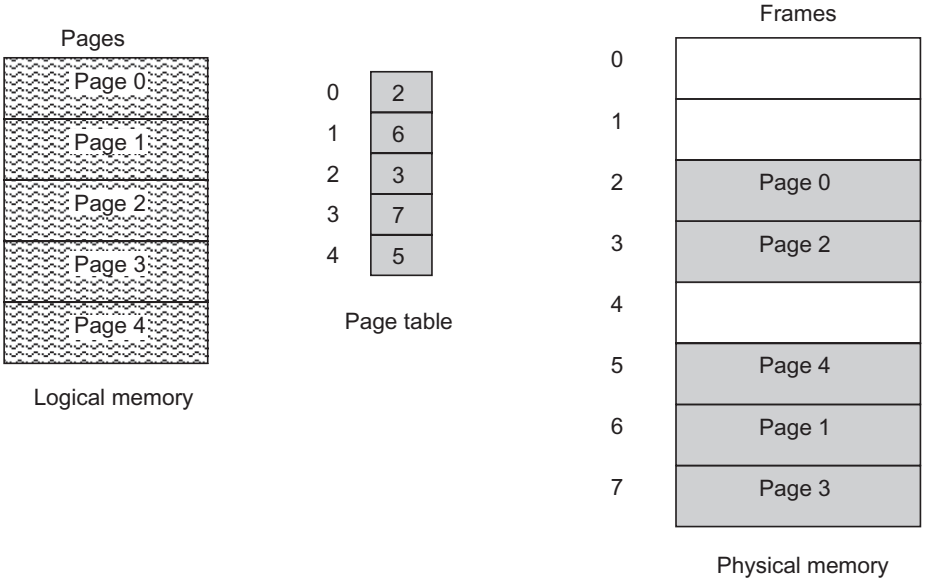


Fig. 10.17 Address translation in paging

Example 10.12

A program's logical memory has been divided into five pages and these pages are allocated frames, 2, 6, 3, 7, and 5. Show the mapping of logical memory to physical memory.

Solution

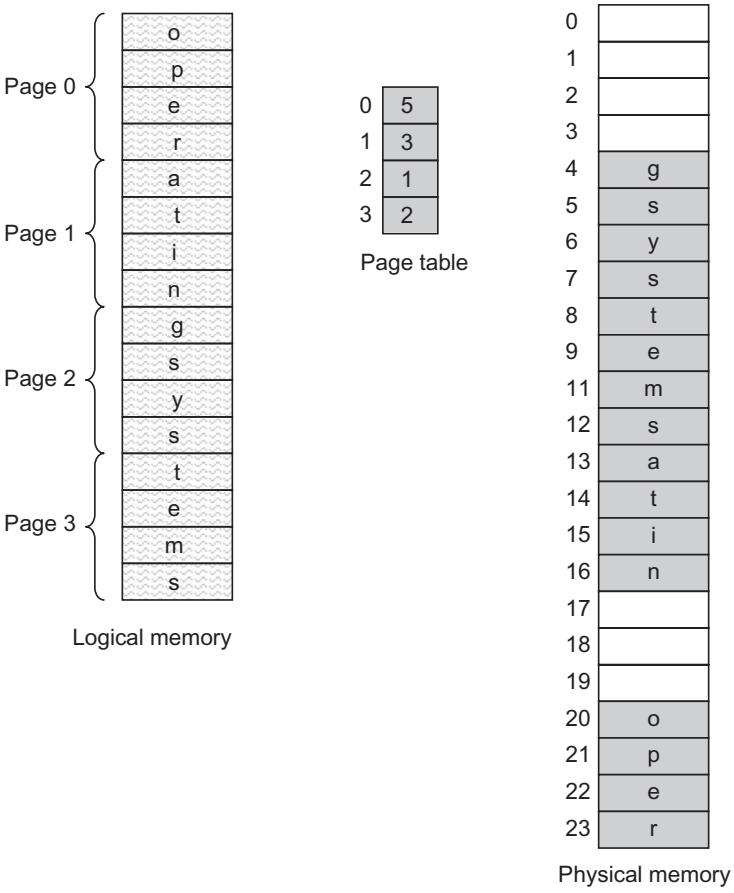


Example 10.13

A program's logical memory has been divided into four pages. The pages are allocated frames 5, 3, 1, and 2. Assume that each page size is 4 bytes. Show the mapping of logical memory to physical memory in bytes.

Solution

Since the page size is of 4 bytes, the frame size will also be the same. Therefore, a logical address needs to be converted accordingly. Suppose the logical address is (2, 2) (page number = 2 and offset = 2). After the page number is used to index into the page table, page 2 is in frame 1. The actual location of frame 1 is at $1 \times 4 = 4^{th}$ byte in physical memory. So the logical address will map to $4 + 2 = 6^{th}$ byte in physical memory. Similarly the logical address page number = 0 and offset = 3, will map to $(5 \times 4) + 3 = 23^{rd}$ byte in the memory.



The logical address will not be generated as two-dimensional address by the processor. The processor generates only a simple single-dimensional binary address, but the page number and offset is separated out from the bits of the logical address. Some lower bits (rightmost) are for offset and higher bits (leftmost) are for page number. The question is how many bits in the address should be fixed for the page number and offset. This is decided by the page size. If a process is of size 65536 bytes (2^{16}) and the page size is taken as 1024 bytes (2^{10}), then the number of pages will be 64 (2^6). It means the page number has 6 bits (leftmost) and the remaining 10 bits (rightmost) are for offset. In this way, the page size decides the division of the logical address into a page number and offset. Thus, if the size of a logical address space is 2^a, and the page size is 2^b, then the high-order bits (a-b) indicates the page number and the low-order bits b gives the page offset (see Fig. 10.18).

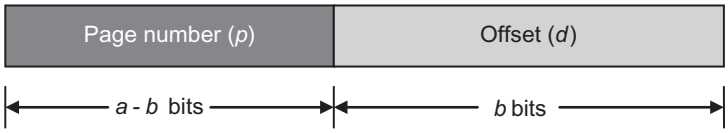


Fig. 10.18 Logical address format in paging

But this scheme is only possible if page size is taken as power of 2. Otherwise, it is not possible to divide the logical address in this fashion. The page size, as power of 2, also conventionally facilitates the conversion of the logical address into a physical address.. After extracting the page number from the logical address, it is used as an index in the page table, and the base address of that page is retrieved. This base address is then appended to the offset and there is no need to calculate their sum. Thus, the computation is also reduced with this method.

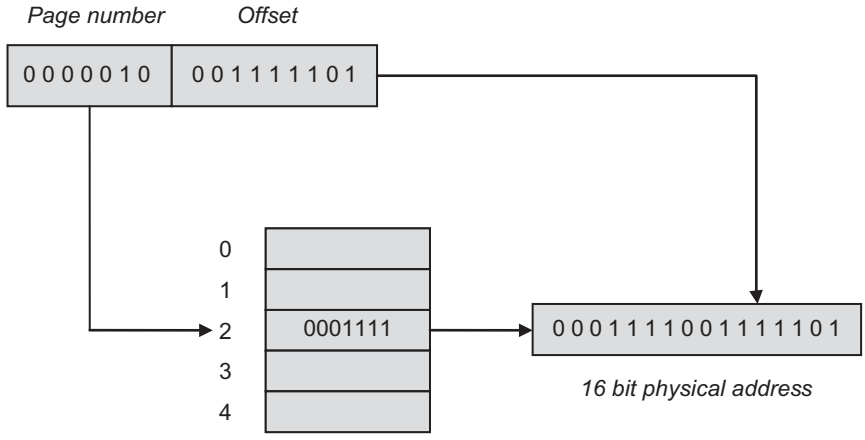
Let us illustrate this method using an example

Example 10.14

In a paging scheme, 16-bit addresses are used with a page size of 512 bytes. If the logical address is 0000010001111101, how many bits are used for the page number and offset? Compute the page number and offset as well. What will be the physical address, if the frame address corresponding to the computed page number is 15.

Solution

Logical address space = 2^{16}
Page size = 512 = 2^9
Number of bits required for the page number = 16 - 9 = 7
Number of bits required for offset, b = 9
Page number is obtained by considering the leftmost 7 bits of the logical address
i.e., Page number = $(0000010)_2 = 2$
Offset is obtained by considering the rightmost 9 bits of the logical address
i.e., Offset = $(001111101)_2 = 125$
Frame address corresponding to the second page in bits = $(15)_{10} = 0001111$
Appending the frame address to the offset, we get
the physical address = **0001111**001111101



Example 10.15

There are 128 pages in a logical address space, with a page size of 1024 bytes. How many bits will be there in the logical address?

Solution

The logical address space contains 128 pages i.e., 2^7 pages.
That means the number of bits required for the page number, $p = 7$
Page size, $2^b = 1024$ bytes $= 2^{10}$
That means the number of bits required for the offset, $b = 10$

$$p = a - b$$
$$7 = a - 10$$
$$a = 17$$

Therefore, 17 bits are required in the logical address.

Example 10.16

There is a system with 64 pages of 512 bytes page size and a physical memory of 32 frames. How many bits are required in the logical and physical address?

Solution

Given, page size = 512 bytes = 2_9 i.e., $b = 9$
No. of pages = 64 = 2_6 i.e., $p = 6$
Since $p = a - b$, $a = 15$
Therefore, 15 bits are required in the logical address.
Since the number of bits required for offset, $b = 9$ and
The number of bits required to address 32 (2^5) frames = 5
Therefore, after appending the number of bits required for the frame size to the offset, 14 bits are required in the physical address.

10.6.1 Paging Implementation and Hardware Requirements

To implement the paging concept, the memory management component of the OS maintains a list of status of all the frames in the memory. Initially, all the frames are empty in the user area of the memory. As soon as a process loads its pages, the empty frames are allocated, and their status is marked as allocated. As the pages are loaded to the frames, the page table of that process is also updated correspondingly. The frame address of a page, where it has been allocated, is entered in the page table. In this way, a page table must be updated as soon as the frame address of a page changes. Furthermore, the address of a page table, where it is stored in the memory, is also stored in the PCB of the process. This information is useful at the time of execution of a process. As soon as a process is scheduled for execution, its appropriate page table is referred for execution, if there is a page table address entry in its PCB.

It is clear from the above discussion that paging will incur cost and increase the access time, because the page table will also be stored in the memory. To reduce the access time, some systems (like DEC PDP-11) were developed, with the help of fast access registers at the hardware level. These registers were used to store the page table entries. These registers were

developed with very high-speed logic, so that address translation in paging is efficient and does not affect the access time. However, this register-based page table entries method could not be successful, because such entries increased (more than million entries) with the later architecture of the system. The large number of page table entries was not feasible to implement with fast access registers. Therefore, the page table is stored in the memory only in any contemporary OS. The hardware support needed in this case is to have a register just like the base register, so that the page table address can be stored. This base register is known as *Page Table Base Register* (PTBR). So, whenever a process is scheduled to be executed, the page table address from its PCB is loaded into PTBR, and the corresponding page table is accessed in the memory. Thus, a page table per process, along with one PTBR in hardware, is sufficient to implement the paging concept. When the current process is suspended or terminated, and another process is scheduled to execute, then the PTBR entry is replaced with the page table address of a new process.

Another problem in paging is an increased number of memory accesses. To execute an instruction in a page of a process, first its page table is accessed in the memory to find the frame address of the desired page. This frame address is then combined with the offset, and then the actual frame is accessed in the memory for the operation. Therefore, there are two memory accesses that slow down the execution.

Total memory access time = Time to access page table + Time to access memory location

This problem can be reduced if some of the page table entries are cached in a high-speed cache memory. High speed associative cache memory known as *Translation Look-aside Buffer* (TLB) is used for this purpose. The TLB consists of some recently-used page table entries with page number (p) and its corresponding frame address (b). Whenever the CPU generates a logical address, TLB is first searched for the page frame address. If the page number is found in the TLB, it is known as a *TLB hit*, otherwise it is a *TLB miss*. This type of memory mapping through TLB is known as *associative mapping*. In case of a TLB hit, the frame address of the desired page number is retrieved from the TLB, and there is no need to access the page table, thereby reducing to one memory access. However, in case of a TLB miss, the page table must be searched for the frame address, and then the physical address is mapped. Furthermore, this page table entry must be entered in the TLB as well, so that the next reference to this page number can be found in the TLB. In this way, the solution to reduce the two-memory accesses in paging with TLB may be costlier, if the TLB miss ratio is high. To make it successful, the TLB hit ratio must be high, otherwise Effective-memory Access Time (EAT) will increase further (see Fig. 10.19). The total access time, in this case of a system with TLB, will be increased as compared to that without TLB as in the following:

Total memory access time = Time to access TLB + Time to access page table + Time to access memory location

The effective memory access time can be calculated, if TLB is used for paging address translation. It is calculated, based on the probability of TLB hit or miss, as in the following:

Effective memory access time (EAT) = $P(H) \times$ (*Time to access TLB + Time to access the memory location*) + $[(1 - P(H)) \times$ (*Time to access TLB + 2* (*Time to access the memory location*))]

where P(H) is TLB hit ratio.

If the hit ratio is high, the EAT is reduced. So, there should be a mechanism, such that the hit ratio is high, in order to have a low EAT. The performance of the system with TLB is highly dependent on the high value of hit ratio, otherwise its performance will be worse

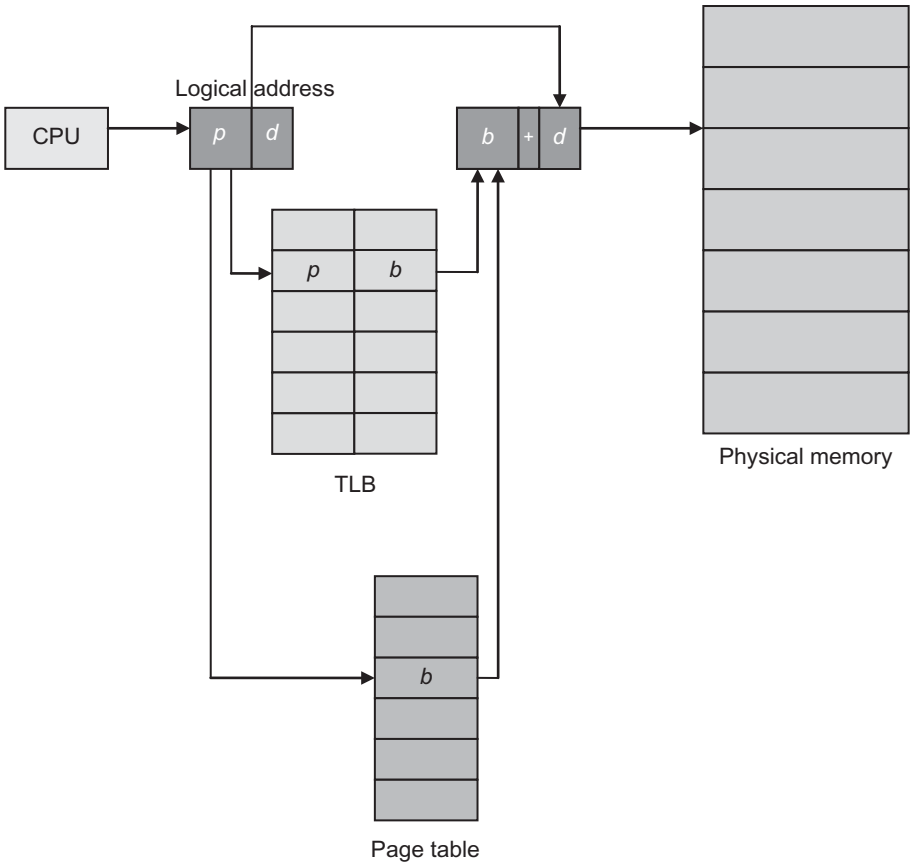


Fig. 10.19 Paging implementation with TLB

than the system without TLB. Therefore, the TLB entries should be designed to have the most frequent page table entries. Associative registers can also be added to have more number of frequent page table entries.

Example 10.17

In a paging system with TLB, it takes 30 ns to search the TLB and 90 ns to access the memory. If the TLB hit ratio is 70%, find the effective memory access time. What should be the hit ratio to achieve the effective memory access time of 130 ns?

Solution

Time taken in case of TLB hit = Time to access TLB + Time to access memory
= 30 + 90 = 120 ns
Time taken in case of TLB miss = Time to access TLB + Time to access page table + Time to access memory
= Time to access TLB + 2(Time to access the memory location)
= 30 + 2 (90) = 210 ns

Effective memory access time (EAT) = *P(H) × (Time to access TLB + Time to access the memory location) + [(1−P(H)) × (Time to access TLB + 2(Time to access the memory location))]*

= 0.70 x 120 + 0.30 x 210

= 147 ns

Next, let us calculate the hit ratio when EAT is 130 ns.

130 = P(H) x 120 + ((1-P(H)) x 210)
130 = P(H) x 120 + 210 – 210 x P(H)
90 P(H) = 80
P(H) = 0.89
P(H) = 89%

Therefore, to reduce EAT to 130 ns, the hit ratio should be increased to 89%.

10.6.2 Protection in Pages

In a paging environment, every page has separate access rights. A page may be read-only, write-only, or read-write. This kind of protection, associated with a page, can be implemented with the help of a page table. The entries of a page in the page table may also contain its protection bits. These protection bits are known as *access protection bits*. The read, write, and execute bits can be set or reset as desired for a page, so that when that page is referred to, its access rights will be there. If an execution tries to violate the access rights in access protection bits, an interrupt is generated by the OS (see Fig. 10.20). However, access protection bits are only useful when a programmer knows in advance, which page is to be protected, and that is difficult. For this, the module should be mapped to the pages, which need to be protected.

It may be possible that a process in its lifetime does not use all its pages. Some of the pages, at a given time, may not be valid. However, the page table entries for these invalid pages are still there, and therefore, may be accessed. To make these pages invalid, another protection bit known as *valid-invalid* bit can be added to the page table. This bit is used in the page table to mark a page as valid or invalid. Valid means the page is being referred by the process, and invalid means the page is not in use by the process and it is illegal (see Fig. 10.21). Another method to have protection against these invalid pages is to have a *Page Table Length Register* (PTLR), just like the PTBR. Any access beyond the PTLR causes an interrupt in the OS. However, it would be a waste to have page entries of those pages, which are not in use. As page tables consume space in the memory, it is not recommended to have these invalid pages in the page table entries.

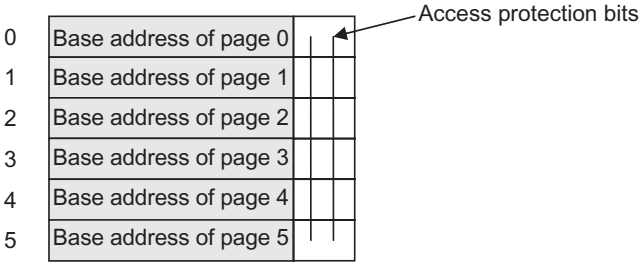


Fig. 10.20 Page table with access protection bits

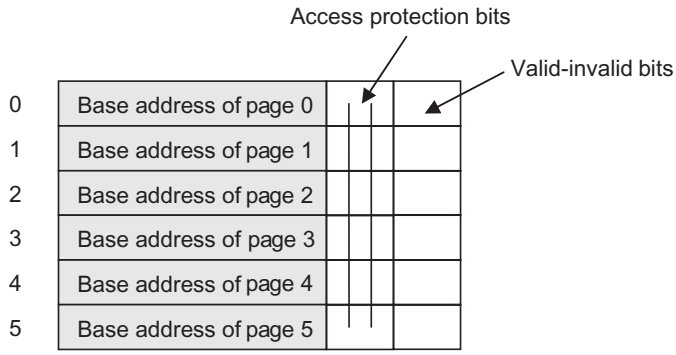


Fig. 10.21 Page table with valid-invalid bits

Example 10.18

In Example 10.12, assume that page 4 is invalid, page 0 is read-only, page 2 is read-write, and other pages have read, write, and execute permissions. Draw the page table, incorporating the protection bits for the pages.

Solution

The page table is given by:

Base Address	Read	Write	Execute	Valid/Invalid bit
2	1	0	0	1
6	1	1	1	1
3	1	1	0	1
7	1	1	1	1
5	1	1	1	0

10.6.3 Shared Pages

In a multi-user time-sharing environment, different users may need to use the same software. However, this does not mean that each user has copies of the desired software. To save memory space, there should be a single copy of the software in the memory, instead of multiple copies. Paging can be used here by means of shared pages. Suppose, if a compiler consuming 1500 KB memory is shared between two users, then all the pages, related to the compiler, can be shared among all users. In the memory, there will be only one copy of the compiler. The page table corresponding to all user processes will map the compiler to the same location in the memory, except the data page of the process (see Fig. 10.22). If each user process uses a separate copy of the compiler, 3000 KB memory will be consumed, whereas with shared pages concept, only 1500 KB memory will be consumed, thereby, saving the memory space.

However, not all the pages can be shared. The data areas of processes cannot be shared, as the processes will need separate data areas to run the same software. Moreover, the sharable code page should be of re-entrant type. A re-entrant code never changes during execution, and thus can be shared easily by multiple users. Thus, all the processes, sharing the same software, can execute the same code in the memory in the form of shared pages, and each process has its own copy of registers and data storage in the memory as non-sharable pages. The page table of each process maps to the same location in the memory for the shared pages.

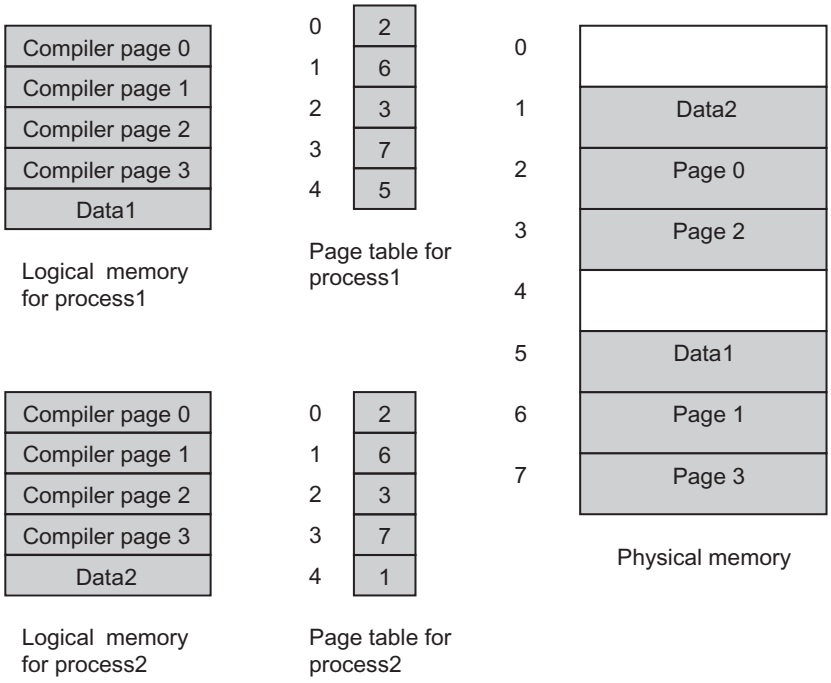


Fig. 10.22 Shared pages

10.6.4 Fragmentation

The paging concept was developed to reduce external fragmentation. However, it suffers from internal fragmentation, as it may leave some bytes of memory, if the process does not fit in a page size. If the size of a process is an exact multiple of the page size chosen, there will not be any internal fragmentation, because each frame will be utilized completely. In practice, it may not be possible that process size is an exact multiple of page size. Therefore, there will be some internal fragmentation. Paging eliminates external fragmentation, because even if there is sufficient space that is non-contiguous in the memory, it will be allocated to a process utilizing the memory space.

10.7 PAGE TABLE STRUCTURES

The page tables in the paging concept may have various page table structures, depending on different requirements.

10.7.1 Hierarchical/Multi-level Page Table Structure

Modern computer architecture supports a larger address space, such that a page table consisting of page entries, consumes a large memory, even in megabytes. Most of the systems are of 32-bit logical address space. If the page size is 4 KB (2^{12}), and a page table entry

consumes 4 bytes, then the page table will consume 4 MB of memory. Thus, if a single page table, corresponding to a single process, consumes memory in megabytes, a large space is required to accommodate page tables of all the processes in the system. It is obvious that this much space cannot be accommodated in the memory contiguously. The solution is to allocate the memory to page tables as non-contiguous. The page table of a process may also be scattered in the memory, if it does not find contiguous space. Another table must be maintained to keep the record of the page table, where it occupies the space in the memory. It means the page table is also paged and is called two-level paging. Furthermore, if this second level page table also cannot be accommodated in a contiguous space, it may again be divided and memory is allocated non-contiguously. In this way, there may be several levels to manage the page tables in the memory. This is known as *multi-level* or *hierarchical page table structures*.

Let us discuss in detail how the multi-level page structures are implemented. In a two-level page table structure, the page table is paged, such that its entries are scattered in the memory and allocated non-contiguous memory. But another table is maintained that will contain the entries of the page table, where these are stored in the memory. This table is known as *outer page table* or *directory* of page table (see Fig.10.23).

To implement a two-level page table structure, the logical address needs to be modified, in order to have the outer page table entries. To perform an operation in the process' page, first the outer page table will be searched for the address of a page table, and then the page table is searched for the address of the page. So, the page number field in the logical address is divided into two parts: one for the outer page table and another for the page table. The modified logical address is shown in Fig. 10.24 for a 32-bit logical address space. Using this modified logical address, the address translation starts from the outer page table bits p1, and then continues using p2 and offset d, as shown in Fig. 10.25.

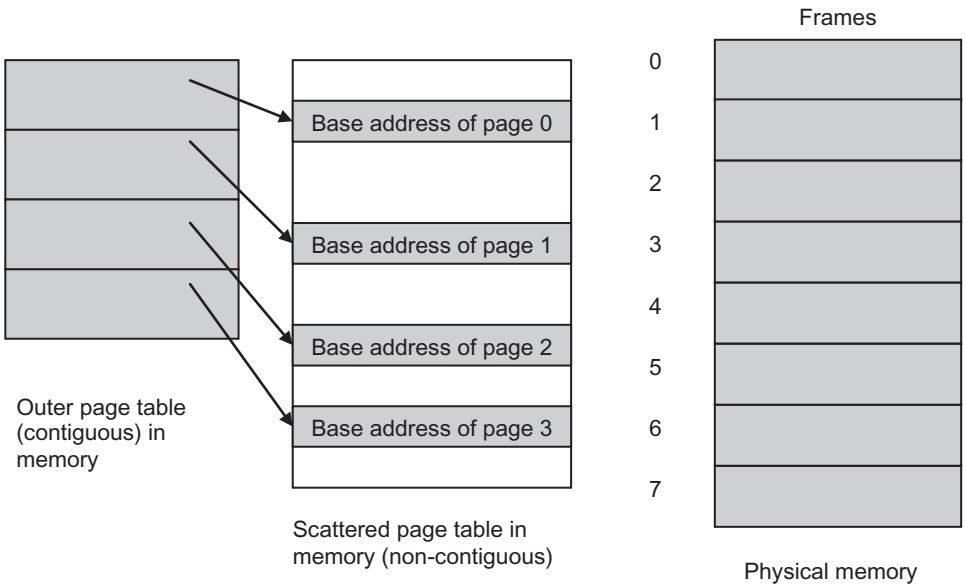


Fig. 10.23 Two-level page table structure

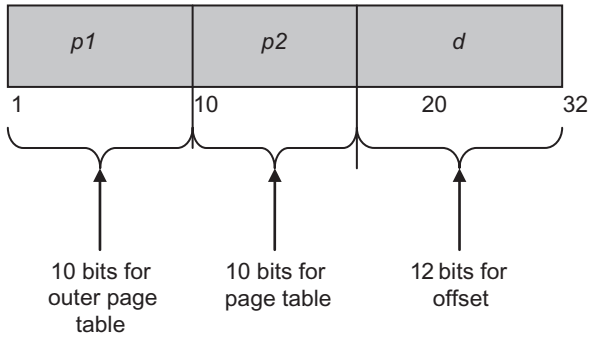


Fig.10.24 32-bit Logical address corresponding to two-level page table structure

Example 10.19

A system with 32-bit logical address uses a two-level page table structure. It uses page size of 2^{10}. The outer page table or directory is accessed with 8 bits of the address.

i) How many bits are required to access the page table?
ii) How many entries are there in the directory?
iii) How many entries are there in the page table?

Solution

i) Since the page size is 2_{10}, therefore, 10 bits are required for the offset. It is given that 8 bits are required to access the outer page table, so bits required to access the page table = 32 – (10 + 8) = 14.

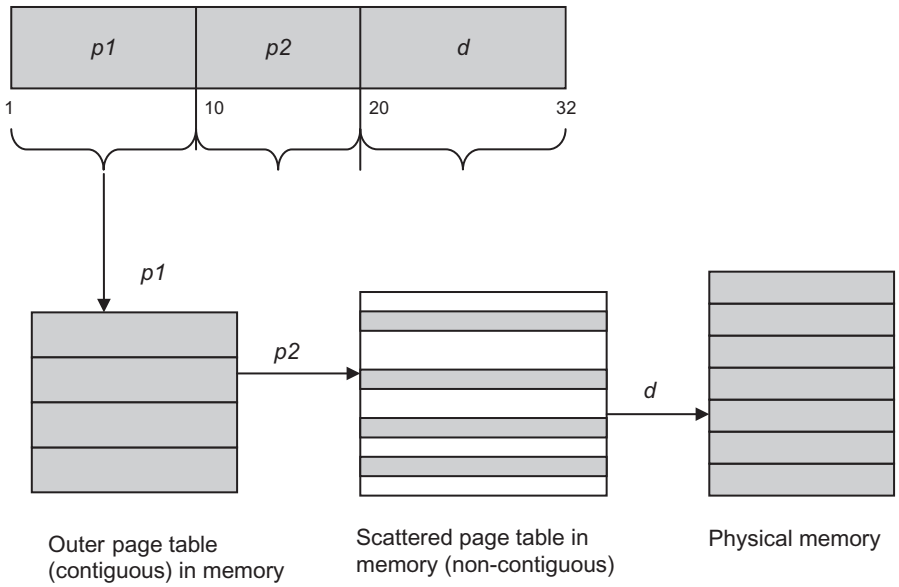


Fig. 10.25 Address translation using two-level page table structure

ii) Since directory is accessed with 8 bits,
It can have $2^8 = 256$ entries
iii) Since page table is accessed with 14 bits.
It can have $2^{14} = 16,384$ entries

Example 10.20

A system with 32-bit logical address space uses 512 bytes page size. A page table entry takes 4 bytes. If a multi-level scheme is to be used for the page table structure, how many levels are required?

Solution

Page size = $512 = 2^9$
Therefore, page offset requires = 9 bits
The page number required = 32 – 9 = 23 bits

A single-level page table can handle 9 of 23 bits. That means three levels are required: Dividing 23 bits into 9 for the first level, 9 for the second level, and 5 for the third level, page table at the first level and second level will have 2^9 entries each, and the third level will have 2^5 entries.

The two-level paging may not be valid for 64-bit systems. The directory or outer page table entries may be too large, such that these entries cannot be allocated contiguously. Therefore, the directory should also be divided so that these entries are allocated non-contiguous space as available. This results in three-level paging. To implement three-level paging, logical address needs to be modified to accommodate another outer page table. In this paging, there will be two outer page tables and one page table (see Fig.10.26). In this way, multiple levels can be formed, if the size of page table entries is high, resulting in hierarchical or multi-level page table structures.

The disadvantage of hierarchical or multi-level paging is that it increases the memory accesses. As the number of levels increases, the number of memory accesses also increases. There are two memory accesses in a simple page table implementation, as discussed earlier. If there is a two-level paging structure, there are three memory accesses. Similarly, if there is a three-level paging structure, there are four memory accesses, and so on. Multi-level paging, though, manages the memory non-contiguously; and the number of memory accesses is large. However, this number of accesses can be reduced with the help of associative mapping, by having TLB as associative memory, and the performance of execution can be increased.

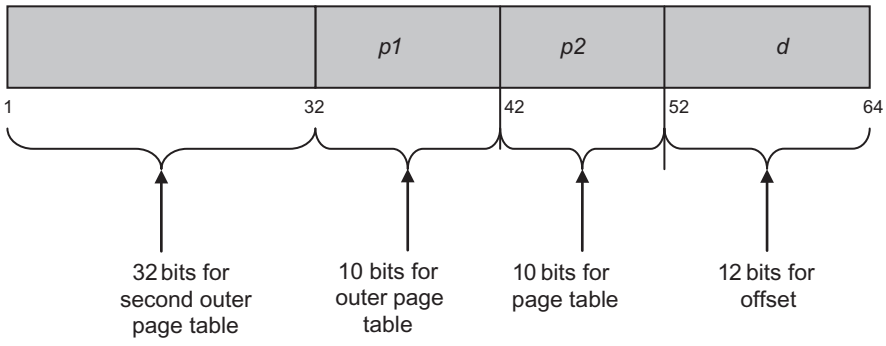


Fig. 10.26 64-bit Logical address corresponding to three-level page table structure

10.7.2 Inverted Page Table Structure

The page table size is a serious issue for an OS designer. The page table size is directly proportional to the virtual address space. There is another way to design it: using an inverted table structure. In this design, instead of a virtual page, a real page frame is taken as the page table entry. In other words, an inverted page table has one entry for each real page frame of memory. Each entry consists of the virtual address of the page stored in the real memory location. It consists of information about the process that owns the page. Since the process and page information is stored together in one entry, there is no need to prepare a separate process table for each process. There is a standard page table for all the processes that contains only one entry for each physical page frame of the memory. This saves memory space, but inverted page table structures are not appropriate for shared pages, as there is more than one virtual address for one physical page frame.

The inverted page table is indexed by the page frame number of the physical memory, rather than the virtual page number. The logical address in this page table structure consists of process ID (*pid*), page number (*p*), and the displacement (*d*): (*pid*, *p*, *d*) (see Fig. 10.27). Once the logical address is generated, the inverted page table is then matched for the required *pid* and *p*. As soon as a match is found, the frame address is obtained, which is then merged with the offset to get the physical mapping address.

10.7.3 Hashed Page Table Structure

Inverted page table is efficient in memory saving, but searching the table for a match of process ID and page number is a disadvantage. Searching the inverted page table may adversely affect the performance of paging in the system. Therefore, hashing is used to speed up the page table lookup. An appropriate hash function is used and applied on the page number of the virtual address to locate the page table entry. The hash function on the page number results in a value

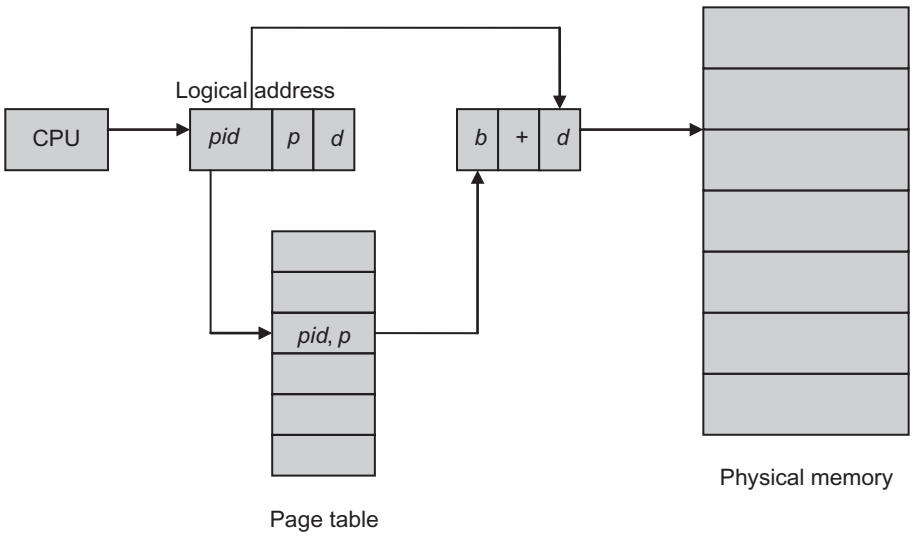


Fig. 10.27 Address translation in paging using inverted page table

that is used as an entry number for the page table to be searched and mapped. If this entry contains the page table, its frame address is used to map the virtual address. Otherwise, the system checks the value of the chaining pointer, as the chaining mechanism is used with hashing. If the chaining pointer is null, it means the page is not in memory, and therefore, it is a page fault. Otherwise, there is a collision at that page table entry.

10.8 SEGMENTATION

A programmer writes programs not in terms of pages, but modules, to reduce the problem complexity. There may be many modules: main program, procedures, stacks, data, and so on. So, it would be better if memory management is also implemented in terms of these modules. Segmentation is a memory management technique that supports the concept of modules. The modules in this technique are called segments. Now the memory management is implemented in the form of segments, instead of pages (see Fig.10.28). The segments are logical divisions of a program, and they may be of different sizes, whereas pages in the paging concept are physical divisions of program, and are of equal size.

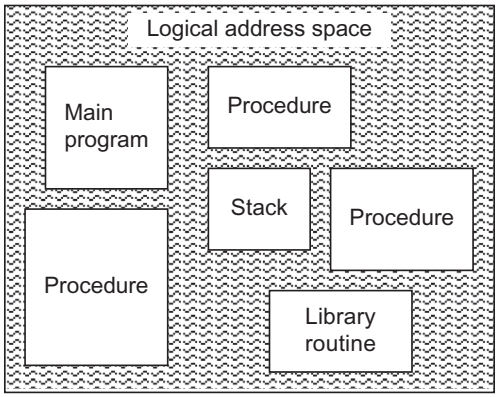


Fig. 10.28 Logical address space divided into segments

Managing memory in the form of segments has two obvious advantages: one is that, segments as logical memory are closer to a programmer's way of thinking, and the other is that the segments need not be of the same size, compared to pages. All the modules or segments in the programs are of different sizes. Therefore, it is advantageous to divide the logical address space into blocks of different sizes as required. This division of logical address space into variable-sized segments eliminates the problem of internal fragmentation that occurred in the paging concept. Thus, segmentation can be defined as a memory management technique, where the logical address space is divided into variable-sized segments, as required. Each segment is identified by a name and its length. The logical address is in two parts: the segment name and its offset, in order to know the location within a segment. There are three major segments in an executable program: code segment, data segment, and stack segment. Each of these segments might use another segment. For example, a program may use segments for precompiled library routines and for any other sub-routines. The segmentation does not require a programmer to specify the segments and their sizes.

The logical address in segmentation has two parts: segment number (segment name is replaced by segment number for the convenience of implementation), and offset in the segment. Consequently, this logical address is converted into physical address. To convert it into physical address, the starting address of segment in the memory, that is, address in the base register, must be known. The starting addresses of all the segments are stored in a table, known as *segment table*, as shown in Fig.10.29. The compiler/linker creates the segments at the time of compilation/linkage, numbers them, builds a segment table, and finally an executable image is produced by assigning two-dimensional addresses. In segmentation, there is a need to know