Unit 7 Memory Management

Structure:

7.1 Introduction

In the previous unit we have discussed about deadlocks. In this unit we shall discuss about memory management which is very much important in effective CPU utilization. The concept of CPU scheduling allows a set of processes to share the CPU thereby increasing the utilization of the CPU. This set of processes needs to reside in memory. The memory is thus shared and the resource requires to be managed. Various memory management algorithms exist, each having its own advantages and disadvantages. The hardware design of the system plays an important role in the selection of an algorithm for a particular system. That means to say, hardware support is essential to implement the memory management algorithm.

Objectives:

After studying this unit, you should be able to:

- differentiate logical and physical address space
- explain swapping technique
- discuss various memory allocation methodologies
- describe paging and page table implementation
- explain segmentation

7.2 Logical vs. Physical Address Space

An address generated by the CPU is referred to as a logical address. An address seen by the memory unit, that is, the address loaded into the memory address register (MAR) of the memory for a fetch or a store is referred to as a physical address. The logical address is also sometimes referred to as a virtual address. The set of logical addresses generated by the CPU for a program is called the logical address space. The set of all physical addresses corresponding to a set of logical address is called the physical address space. At run time / execution time, the virtual addresses are mapped to physical addresses by the memory management unit (MMU). A simple mapping scheme is shown in Figure 7.1.

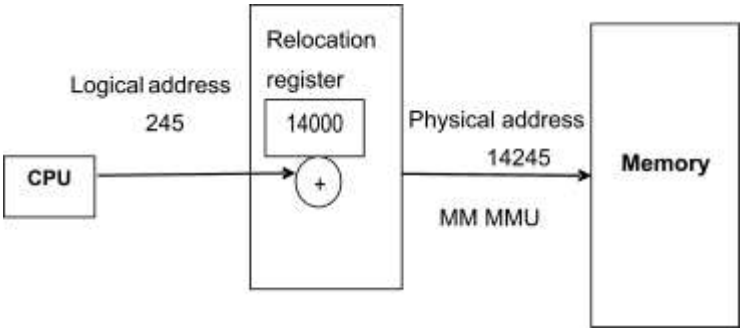


Fig. 7.1: Dynamic Relocation

The relocation register contains a value to be added to every address generated by the CPU for a user process at the time it is sent to the memory for a fetch or a store. For example, if the base is at 14000 then an address 0 is dynamically relocated to location 14000, an access to location 245 is mapped to 14245 (14000 + 245). Thus, every address is relocated relative to the value in the relocation register. The hardware of the MMU maps logical addresses to physical addresses. Logical addresses range from 0 to

a maximum (MAX) and the corresponding physical addresses range from (R + 0) to (R + MAX) for a base value of R. User programs generate only logical addresses that are mapped to physical addresses before use.

7.3 Swapping

A process to be executed needs to be in memory during execution. A process can be swapped out of memory in certain situations to a backing store and then brought into memory later for execution to continue. One such situation could be the expiry of a time slice if the round-robin CPU scheduling algorithm is used. On expiry of a time slice, the current process is swapped out of memory and another process is swapped into the memory space just freed because of swapping out of a process (See figure 7.2). Every time a time slice expires, a process is swapped out and another is swapped in. The memory manager that does the swapping is fast enough to always provide a process in memory for the CPU to execute. The duration of a time slice is carefully chosen so that it is sufficiently large when compared to the time for swapping.

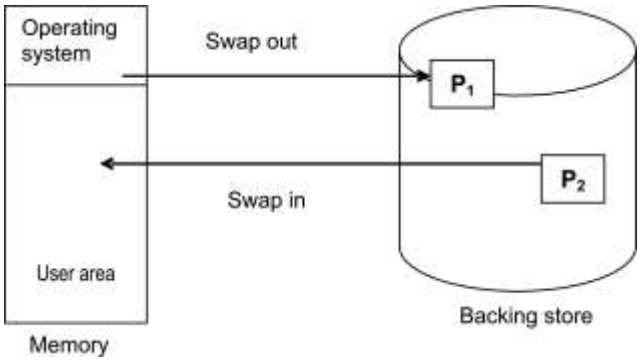


Fig. 7.2: Swapping of processes

Processes are swapped between the main memory and the backing store when priority based CPU scheduling is used. The arrival of a high priority process will be a lower priority process that is executing to be swapped out to make way for a swap in. The swapping in this case is sometimes referred to as roll out / roll in.

A process that is swapped out of memory can be swapped in either into the same memory location or into a different memory location. If binding is done

at load time then swap in has to be at the same location as before. But if binding is done at execution time then swap in can be into a different memory space since the mappings to physical addresses are completed during execution time.

A backing store is required for swapping. It is usually a fast disk. The processes in the ready queue have their images either in the backing store or in the main memory. When the CPU scheduler picks a process for execution, the dispatcher checks to see if the picked process is in memory. If yes, then it is executed. If not the process has to be loaded into main memory. If there is enough space in memory, the process is loaded and execution starts. If not, the dispatcher swaps out a process from memory and swaps in the desired process.

A process to be swapped out must be idle. Problems arise because of pending I/O. Consider the following scenario: Process P_1 is waiting for an I/O. I/O is delayed because the device is busy. If P_1 is swapped out and in its place if P_2 is swapped in, then the result of the I/O uses the memory that now belongs to P_2. There can be two solutions to the above problem:

1) Never swap out a process that is waiting for an I/O.
2) I/O operations to take place only into operating system buffers and not into user area buffers. These buffers can then be transferred into user area when the corresponding process is swapped in.

Self Assessment Questions

1. The logical address is also sometimes referred to as a virtual address. (True / False)
2. An address generated by the CPU is referred to as a _____.
3. MAR stands for _____. (Pick the right option)
   a) Memory Address Register
   b) Memory Allocation Register
   c) Main Address Register
   d) Main Allocation Register

7.4 Contiguous Allocation

The main memory is usually divided into two partitions, one of which has the resident operating system loaded into it. The other partition is used for loading user programs. The operating system is usually present in the lower

memory because of the presence of the interrupt vector in the lower memory (See figure 7.3).

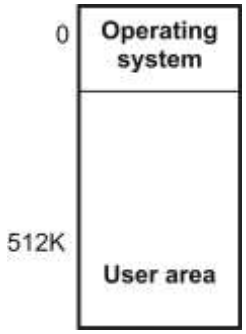


Fig. 7.3: Memory partition

7.4.1 Single partition allocation

The operating system resides in the lower memory. User processes execute in the higher memory. There is always a possibility that user processes may try to access the lower memory either accidentally or intentionally thereby causing loss of operating system code and data. This protection is usually provided by the use of a limit register and relocation register. The relocation register contains the smallest physical address that can be accessed. The limit register contains the range of logical addresses. Each logical address must be less than the content of the limit register. The MMU adds to the logical address the value in the relocation register to generate the corresponding address (See figure 7.4). Since an address generated by the CPU is checked against these two registers, both the operating system and other user programs and data are protected and are not accessible by the running process.

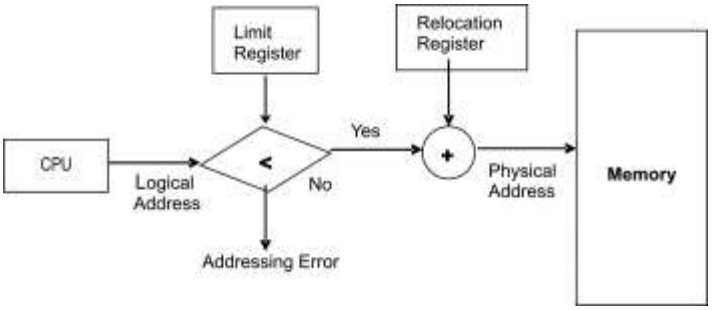


Figure 7.4: Hardware for relocation and limit register

7.4.2 Multiple partition allocation

Multi-programming requires that there are many processes residing in memory so that the CPU can switch between processes. If this has to be so, then, user area of memory has to be divided into partitions. The simplest way is to divide the user area into fixed number of partitions, each one to hold one user process. Thus, the degree of multi-programming is equal to the number of partitions. A process from the ready queue is loaded into one of the partitions for execution. On termination the partition is free for another process to be loaded.

The disadvantage with this scheme where partitions are of fixed sizes is the selection of partition sizes. If the size is too small then large programs cannot be run. Also, if the size of the partition is big then main memory space in each partition goes a waste.

A variation of the above scheme where the partition sizes are not fixed but variable is generally used. A table keeps track of that part of the memory that is used and the part that is free. Initially, the entire memory of the user area is available for user processes. This can be visualized as one big hole for use. When a process is loaded, a hole, big enough to hold this process is searched. If one is found then enough memory for this process is allocated and the rest is available free as illustrated in figure 7.5.

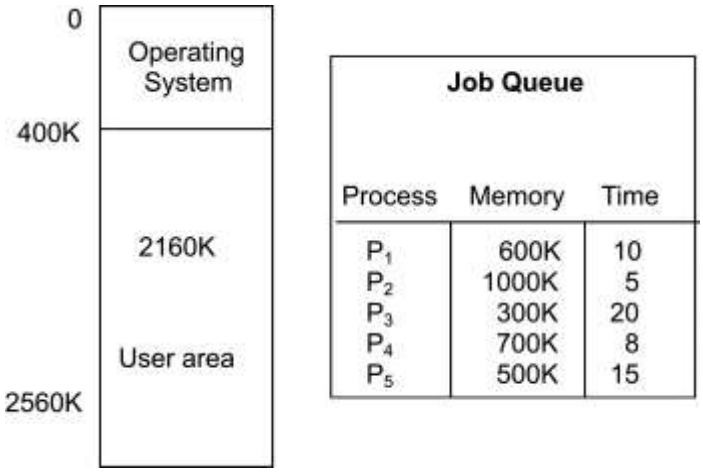


Figure 7.5: Scheduling example

Total memory available: 2560K

Resident operating system: 400K

Memory available for user: 2560 – 400 = 2160K

Job queue: FCFS

CPU scheduling: RR (1 time unit)

Given the memory map in the illustration above, P_1, P_2, P_3 can be allocated memory immediately. A hole of size (2160 – (600 + 1000 + 300)) = 260K is left over which cannot accommodate P_4 (Figure 7.6A). After a while P_2 terminates creating the map of figure 7.6B. P_4 is scheduled in the hole just created resulting in figure 7.6C. Next P_1 terminates resulting in figure 7.6D and P_5 is scheduled as in figure 7.6E.

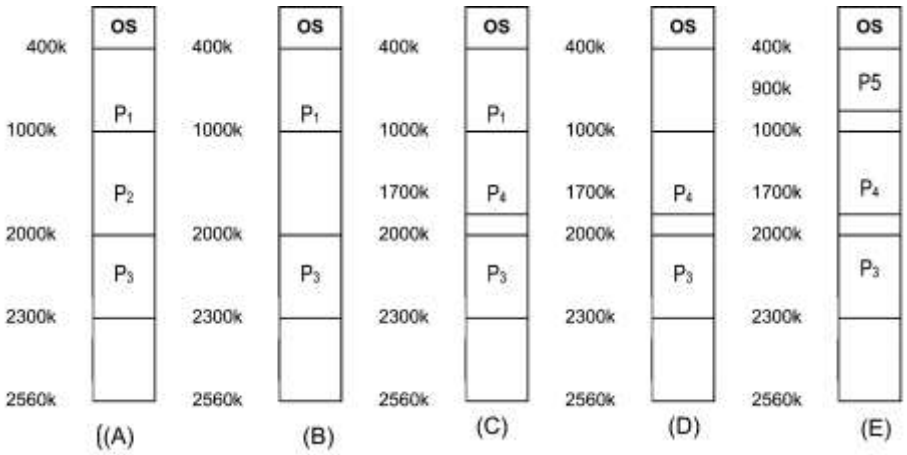


Figure 7.6: Memory allocation and job scheduling

The operating system finds a hole just large enough to hold a process and uses that particular hole to load the process into memory. When the process terminates, it releases the used memory to create a hole, equal to its memory requirement. Processes are allocated memory until the free memory or hole available is not big enough to load another ready process. In such a case the operating system waits for some process to terminate and free memory. To begin with, there is one large big hole equal to the size of the user area. As processes are allocated into this memory, execute and terminate this hole gets divided. At any given instant thereafter there are a set of holes scattered all over the memory. New holes created that are adjacent to existing holes merge to form big holes.

The problem now is to satisfy a memory request of size n from a list of free holes of various sizes. Many solutions exist to determine that hole which is the best to allocate. Most common strategies are:

1) ***First-fit:*** Allocate the first hole that is big enough to hold the process. Search can either start at the beginning of the set of holes or at the point where the last search terminated.
2) ***Best-fit:*** Allocate the smallest hole that is big enough to hold the process. Here the entire list has to be searched or an ordered list of holes by size is to be maintained.
3) ***Worst-fit:*** Allocate the largest hole available. Here also, the entire list has to be searched for the biggest hole or an ordered list of holes by size is to be maintained.

The size of a process is very rarely an exact size of a hole allocated. The best-fit allocation always produces an optimal allocation where the hole left over after allocation is the smallest. The first-fit is the fastest method of allocation when compared to others. The worst-fit allocation is the worst among the three and is seldom used.

7.4.3 Fragmentation

To begin with, there is one large hole for allocation to processes. As processes are loaded into memory and terminate on completion, this large hole is divided into a set of smaller holes that are scattered in between the processes. There may be a situation where the total size of these scattered holes is large enough to hold another process for execution but the process cannot be loaded, as the hole is not contiguous. This is known as external fragmentation. For example, in figure 7.6c, a fragmented hole equal to 560K (300 + 260) is available. P_5 cannot be loaded because 560K is not contiguous.

There are situations where only a few bytes say 1 or 2 would be free if a process were allocated a hole. Then, the cost of keeping track of this hole will be high. In such cases, this extra bit of hole is also allocated to the requesting process. If so then a small portion of memory allocated to a process is not useful. This is internal fragmentation.

One solution to external fragmentation is compaction. **Compaction** is to relocate processes in memory so those fragmented holes create one contiguous hole in memory (See figure 7.7).

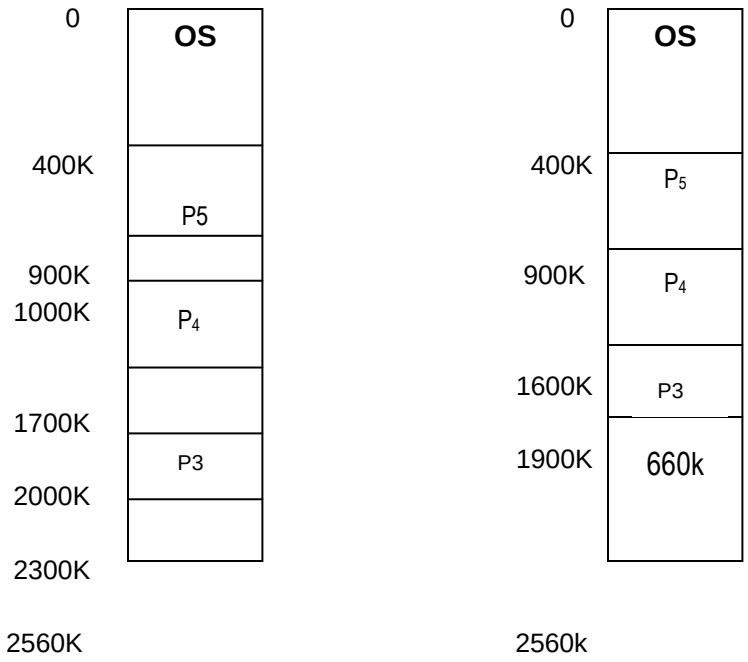


Fig. 7.7: Compaction

Compaction may not always be possible since it involves relocation. If relocation is static at load time, then relocation is not possible and so also compaction. Compaction is possible only if relocation is done at runtime. Even though compaction is possible, the cost involved in relocation is to be considered. Sometimes creating a hole at one end of the user memory may be better whereas in some other cases a contiguous hole may be created in the middle of the memory at lesser cost. The position where the hole is to be created during compaction depends on the cost of relocating the processes involved. Finding an optimal strategy if often difficult.

Processes may also be rolled out and rolled in to affect compaction by making use of a backup store. But this would be at the cost of CPU time.

7.5 Paging

Contiguous allocation scheme requires that a process can be loaded into memory for execution if and only if contiguous memory large enough to hold the process is available. Because of this constraint, external fragmentation is a common problem. Compaction was one solution to tide over external

fragmentation. Another solution to this problem could be to permit non-contiguous logical address space so that a process can be allocated physical memory wherever it is present. This solution is implemented through the use of a paging scheme.

7.5.1 Concept of paging

Physical memory is divided into fixed sized blocks called **frames**. The logical memory is divided into blocks of the same size called **pages**. Allocation of main memory to processes for execution is then just mapping pages to frames. The hardware support for paging is illustrated below (See figure 7.8).

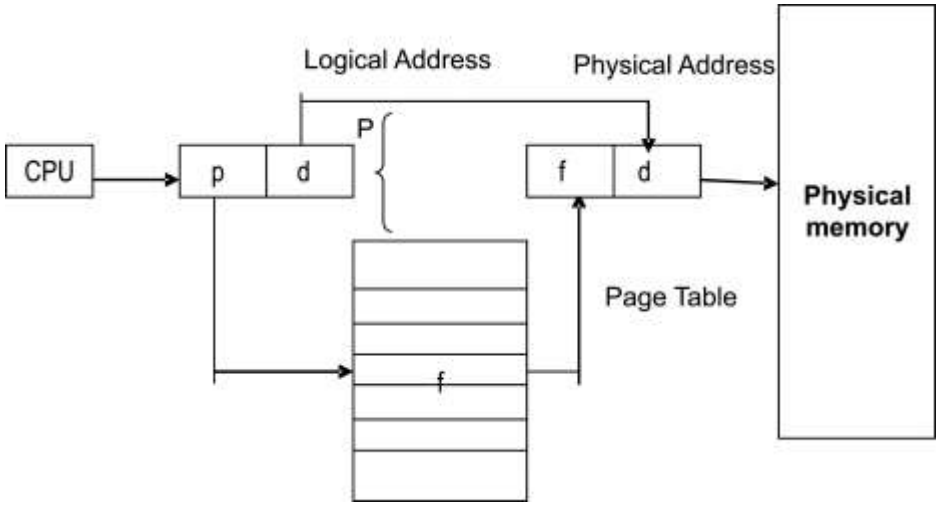


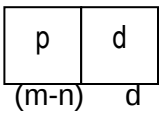
Fig. 7.8: Paging Hardware

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

The size of a page is usually a power of 2. This makes translation of a logical address into page number and offset easy as illustrated below:

Logical address space:	2m
Page size:	2n

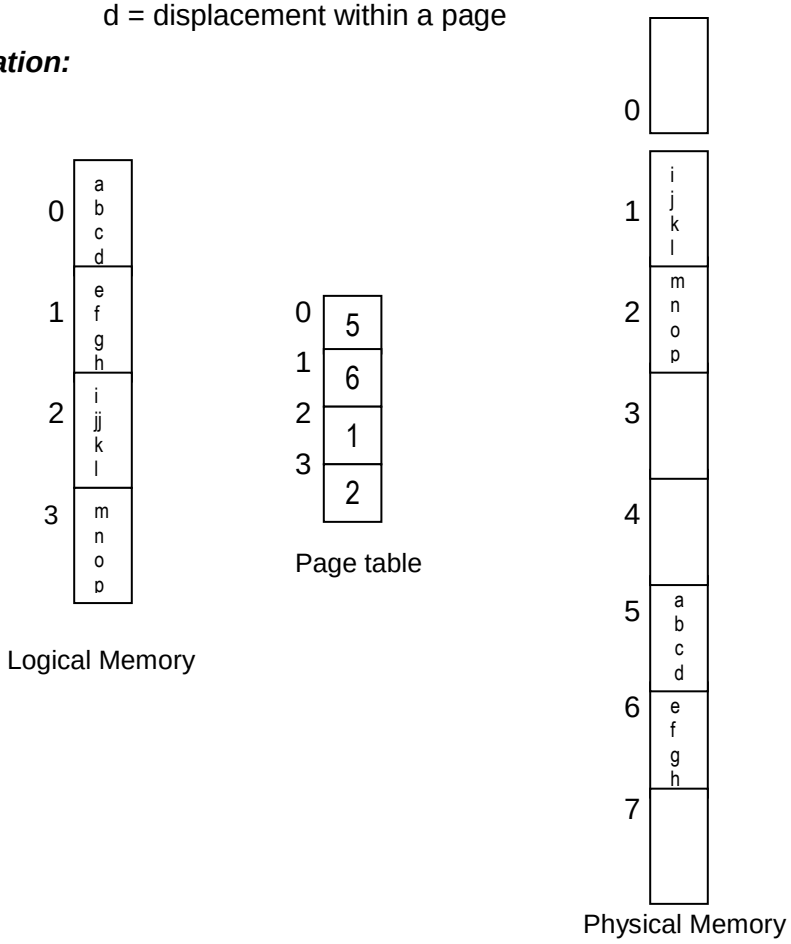
Logical address:



Where p = index into the page table

d = displacement within a page

Illustration:



Page size: 4 bytes

Physical memory: 32 bytes = 8 pages

Logical address 0 ➔ 0 + 0 ➔ (5 x 4) + 0 ➔ physical address 20

3 ➔ 0 + 3 ➔ (5 x 4) + 3 ➔ physical address 23

4 ➔ 1 + 0 ➔ (6 x 4) + 0 ➔ physical address 24

13 ➔ 3 + 1 ➔ (2 x 4) + 1 ➔ physical address 9

Thus, the page table maps every logical address to some physical address. Paging does not suffer from external fragmentation since any page can be loaded into any frame. But internal fragmentation may be prevalent. This is because the total memory required by a process is not always a multiple of the page size. So the last page of a process may not be full. This leads to internal fragmentation and a portion of the frame allocated to this page will be unused. On an average one half of a page per process is wasted due to internal fragmentation. Smaller the size of a page, lesser will be the loss due to internal fragmentation. But the overhead involved is more in terms of number of entries in the page table. Also known is a fact that disk I/O is more efficient if page sizes are big. A trade-off between the above factors is used (See figure 7.9).

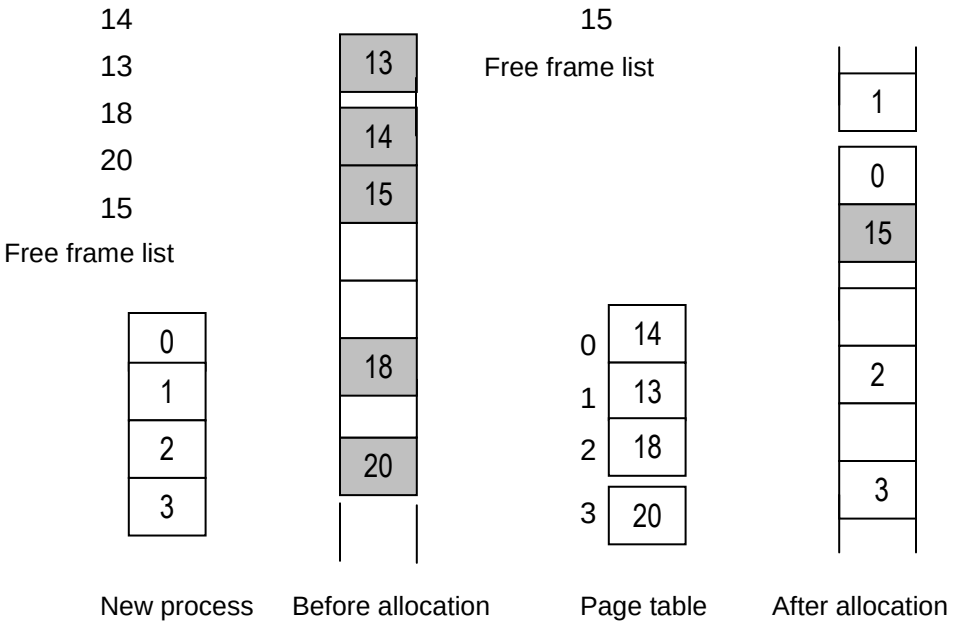


Fig. 7.9: Frame Allocation

A process requires n pages of memory. Then at least n frames must be free in physical memory to allocate n pages. A list of free frames is maintained. When allocation is made, pages are allocated to the free frames sequentially. Allocation details of physical memory are maintained in a frame table. The table has one entry for each frame showing whether it is free or allocated and if allocated, to which page of which process.

7.5.2 Page table implementation

Hardware implementation of a page table is done in a number of ways. In the simplest case, the page table is implemented as a set of dedicated high-speed registers. But this implementation is satisfactory only if the page table is small. Modern computers allow the page table size to be very large. In such cases the page table is kept in main memory and a pointer called the page-table base register (PTBR) helps index the page table. The disadvantage with this method is that it requires two memory accesses for one CPU address generated. For example, to access a CPU generated address, one memory access is required to index into the page table. This access using the value in PTBR fetches the frame number when combined with the page-offset produces the actual address. Using this actual address, the next memory access fetches the contents of the desired memory location.

To overcome this problem a hardware cache called translation look-aside buffers (TLBs) are used. TLBs are associative registers that allow for a parallel search of a key item. Initially the TLBs contain only a few or no entries. When a logical address generated by the CPU is to be mapped to a physical address, the page number is presented as input to the TLBs. If the page number is found in the TLBs, the corresponding frame number is available so that a memory access can be made. If the page number is not found then a memory reference to the page table in main memory is made to fetch the corresponding frame number. This page number is then added to the TLBs so that a mapping for the same address next time will find an entry in the table. Thus, a hit will reduce one memory access and speed up address translation (See figure 7.10).

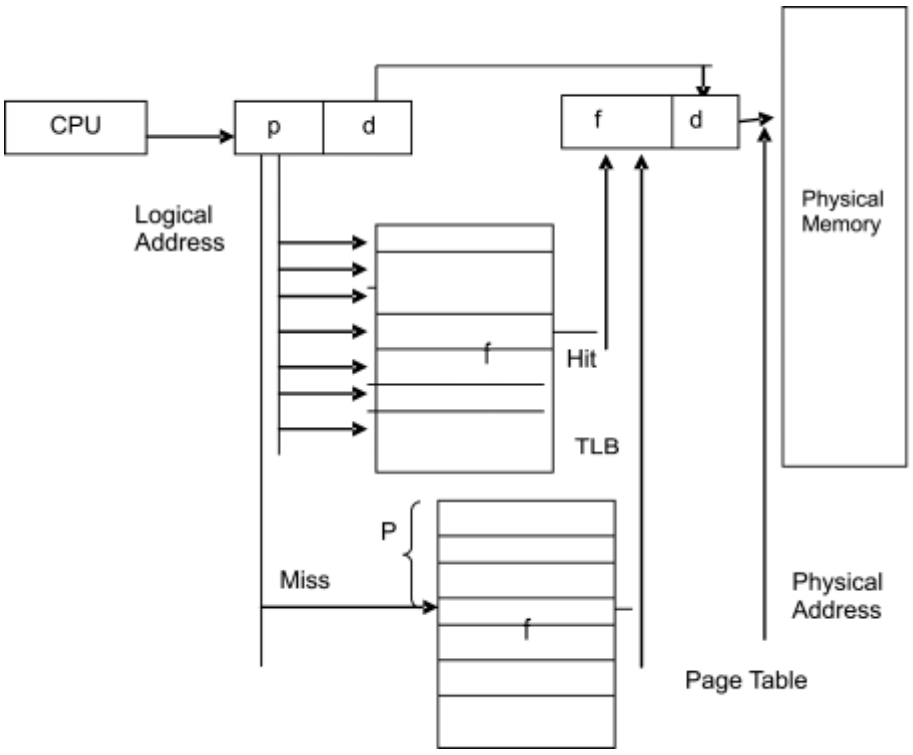


Fig. 7.10: Paging hardware with TLB

Self-Assessment Questions

4. The operating system is usually present in the upper portion of the main memory. (True / False)
5. The method of relocating processes in memory so that the fragmented holes create one contiguous hole in memory is called _____.
6. PTBR stands for _____. (Pick the right option)
   a) Page Table Box Register
   b) Page Table Base Register
   c) Physical Table Base Register
   d) Physical Table Box Register

7.6 Segmentation

Memory management using paging provides two entirely different views of memory – User / logical / virtual view and the actual / physical view. Both

are not the same. In fact, the user's view is mapped on to the physical view. How do users visualize memory? Users prefer to view memory as a collection of variable sized segments (See figure 7.11).

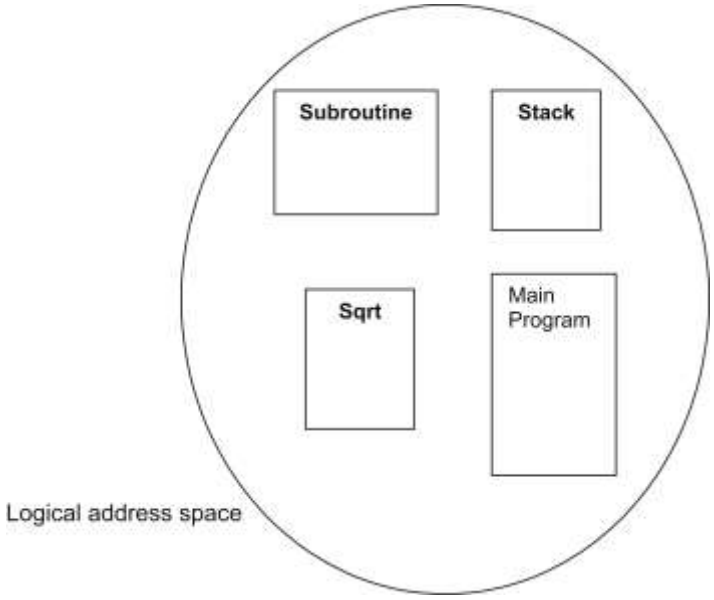


Fig. 7.11: User's view of memory

The user usually writes a modular structured program consisting of a main segment together with a number of functions / procedures. Each one of the above is visualized as a segment with its associated name. Entries in a segment are at an offset from the start of the segment.

7.6.1 Concept of segmentation

Segmentation is a memory management scheme that supports user's view of main memory described above. The logical address is then a collection of segments, each having a name and a length. Since it is easy to work with numbers, segments are numbered. Thus a logical address is represented as <segment number, offset>. User programs when compiled reflect segments present in the input. Loader while loading segments into memory assign them segment numbers.

7.6.2 Segmentation hardware

Even though segments in user view are same as segments in physical view, the two-dimensional visualization in user view has to be mapped on to a one-dimensional sequence of physical memory locations. This mapping is

present in a segment table. An entry in a segment table consists of a base and a limit. The base corresponds to the starting physical address in memory whereas the limit is the length of the segment (See figure 7.12).

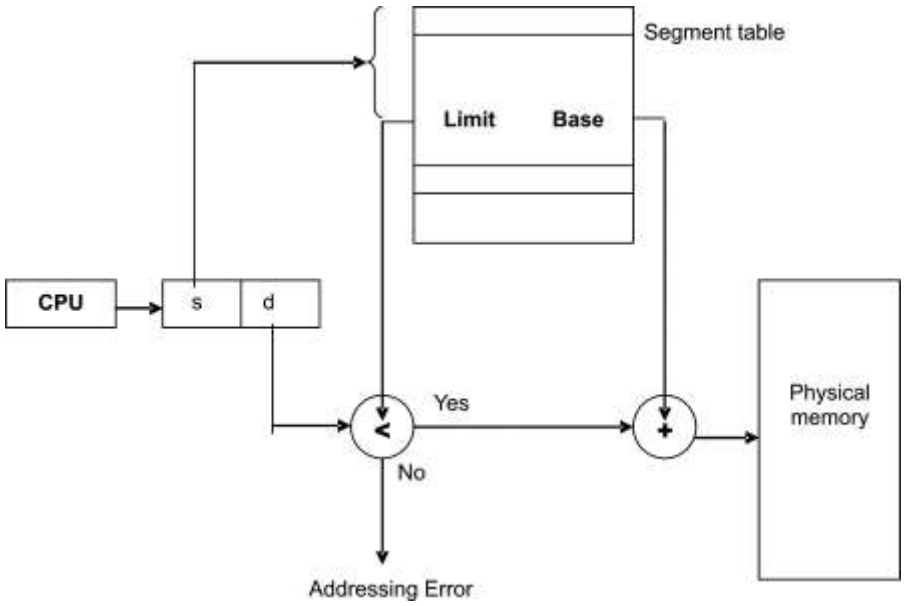


Fig. 7.12: Segmentation Hardware

The logical address generated by the CPU consists of a segment number and an offset within the segment. The segment number is used as an index into a segment table. The offset in the logical address should lie between 0 and a limit specified in the corresponding entry in the segment table. If not the program is trying to access a memory location which does not belong to the segment and hence is trapped as an addressing error. If the offset is correct the segment table gives a base value to be added to the offset to generate the physical address. An illustration is given below (See figure 7.13).

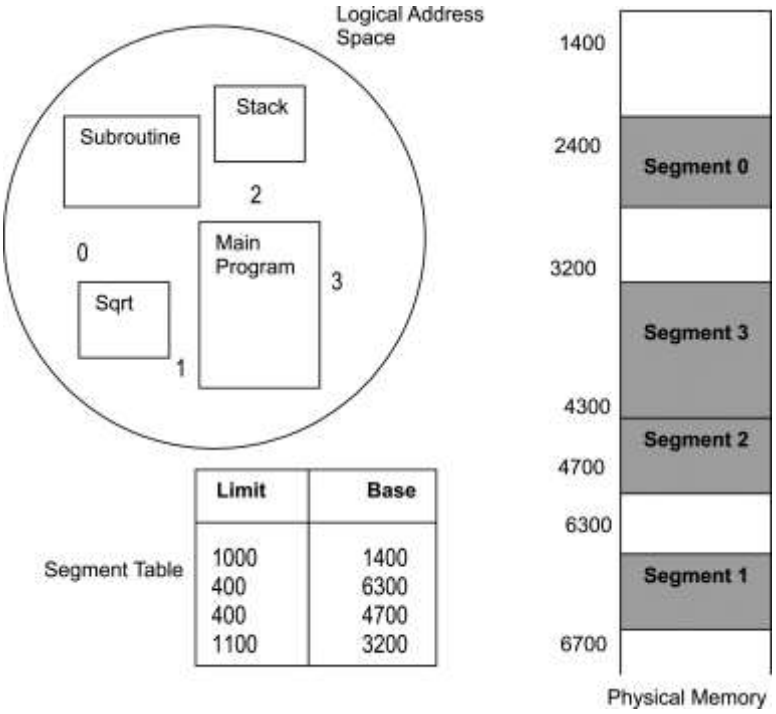


Fig. 7.13: Illustration of Segmentation

A reference to logical location 53 in segment 2 ➔ physical location 4300 + 53 = 4353

852 in segment 3 ➔ physical location 3200 + 852 = 4062

1222 in segment 0 ➔ addressing error because 1222 > 1000

7.6.3 External fragmentation

Segments of user programs are allocated memory by the **bob scheduler**. This is similar to paging where segments could be treated as variable sized pages. So a first-fit or best-fit allocation scheme could be used since this is a dynamic storage allocation problem. But, as with variable sized partition scheme, segmentation too causes external fragmentation. When any free memory segment is too small to be allocated to a segment to be loaded, then external fragmentation is said to have occurred. Memory compaction is a solution to overcome external fragmentation.

The severity of the problem of external fragmentation depends upon the average size of a segment. Each process being a segment is nothing but the variable sized partition scheme. At the other extreme, every byte could be a segment, in which case external fragmentation is totally absent but

relocation through the segment table is a very big overhead. A compromise could be fixed sized small segments, which is the concept of paging. Generally, if segments even though variable are small, external fragmentation is also less.

Self Assessment Questions

7. Memory management using paging provides two entirely different views of memory – logical view and physical view. (True / False)
8. _____is a memory management scheme that supports user's view of main memory.
9. Segments of user programs are allocated memory by the _____. (Pick the right option)
   a) FCFC Scheduler
   b) FIFO Scheduler
   c) Bob Scheduler
   d) LIFO Scheduler

7.7 Summary

Let's summarize the key points discussed in this unit:

- Main memory holds the processes during execution.
- Memory could be contiguously allocated by using any one of the best-fit or first-fit strategies. But one major disadvantage with this method was that of fragmentation.
- To overcome fragmentation problem the memory is divided into pages / frames and the processes are considered to be a set of pages in logical memory.
- These pages are mapped to frames in physical memory through a page table.
- User's view of memory is in the form of segments. Just like a page table, a segment table maps segments to physical memory.

7.8 Terminal Questions

1. What is MMU?
2. Explain how virtual address is mapped on to physical address.
3. Explain single partition allocation.

4. Explain first-fit, Best-fit and worst-fit allocation algorithms with an example.
5. Write a note on External Fragmentation.

7.9 Answers

Self-Assessment Questions

1. True
2. Logical Address
3. a) Memory Address Register
4. False
5. Compaction
6. b) Page Table Base Register
7. True
8. Segmentation
9. c) Bob Scheduler

Terminal Questions

1. MMU stands for Memory Management Unit. At run time / execution time, the virtual addresses are mapped to physical addresses by the MMU. (Refer Section 7.2 for detail)
2. The relocation register contains a value to be added to every address generated by the CPU for a user process at the time it is sent to the memory for a fetch or a store. Thus, every address is relocated relative to the value in the relocation register. The hardware of the MMU maps logical addresses to physical addresses. (Refer Section 7.2).
3. The operating system resides in the lower memory. User processes execute in the higher memory. There is always a possibility that user processes may try to access the lower memory either accidentally or intentionally thereby causing loss of operating system code and data. (Refer Sub-section 7.4.1)
4. The algorithms are as follows:
   A) First-fit: Allocate the first hole that is big enough to hold the process. Search can either start at the beginning of the set of holes or at the point where the last search terminated.

B) Best-fit: Allocate the smallest hole that is big enough to hold the process. Here the entire list has to be searched or an ordered list of holes by size is to be maintained.

C) Worst-fit: Allocate the largest hole available. Here also, the entire list has to be searched for the biggest hole or an ordered list of holes by size is to be maintained. (Refer Sub-section 7.4.2)

5. Segments of user programs are allocated memory by the bob scheduler. This is similar to paging where segments could be treated as variable sized pages. So a first-fit or best-fit allocation scheme could be used since this is a dynamic storage allocation problem. (Refer Sub-section 7.6.3)