MCA Part-II

Paper-XIII: Operating System

Topic: MEMORY MANAGEMENT

PREPARED BY: DR. KIRAN PANDEY

(School of Computer Science)

Email-id: kiranpandey.nou@gmail.com

INTRODUCTION

Memory management is the functionality of an operating system which handles or manages primary memory and moves processes back and forth between main memory and disk during execution. Main Memory refers to a physical memory that is the internal memory to the computer. The word main is used to distinguish it from external mass storage devices such as disk drives. Main memory is also known as RAM. The computer is able to change only data that is in main memory. Therefore, every program we execute and every file we access must be copied from a storage device into main memory. Interaction is achieved through a sequence of reads/writes of specific memory address. The CPU fetches from the program from the hard disk and stores in memory. If a program is to be executed, it must be mapped to absolute addresses and loaded into memory.

Memory management keeps track of each and every memory location, regardless of either it is allocated to some process or it is free. It checks how much memory is to be allocated to processes. It decides which process will get memory at what time. It tracks whenever some memory gets freed or unallocated and correspondingly it updates the status.

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

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

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

The basic approaches of allocation are of two types:

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

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

OVERLAYS and SWAPPING

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

Compile-time: If memory location is known a priori, absolute code can be generated. Load-time: If it is not known, it must generate relocatable at complete time.

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

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

Overlays

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

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

A program based on overlay scheme mainly consists of following:

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

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

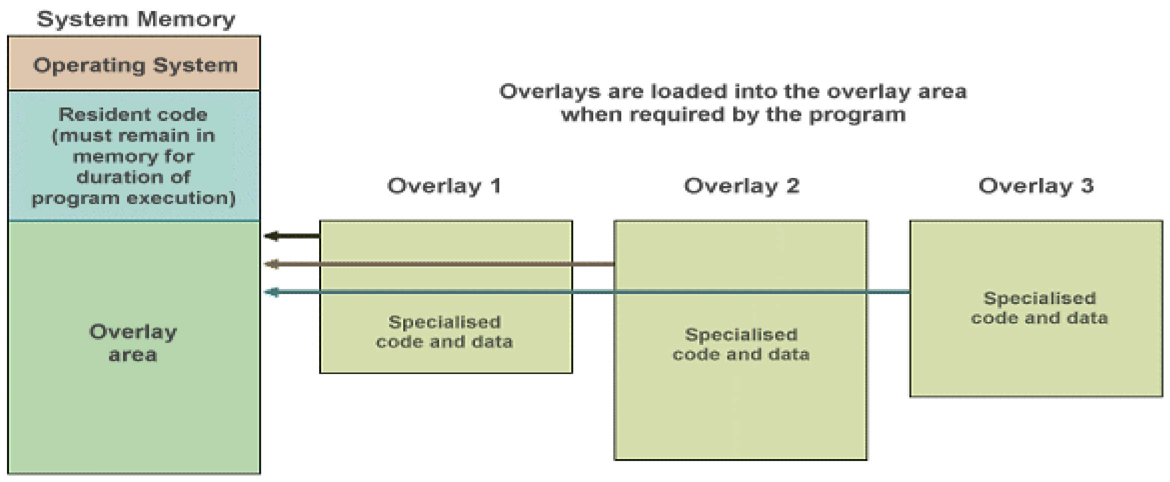

System Memory
Operating System
Resident code (must remain in memory for duration of program execution)
Overlay area
Overlays are loaded into the overlay area when required by the program
Overlay 1
Overlay 2
Overlay 3
Specialised code and data
Specialised code and data
Specialised code and data

Figure 1: Example of overlay

Overlay manager/driver is responsible for loading and unloading on overlay segment as per requirement. But this scheme suffers from following limitations:

- Require careful and time-consuming planning.

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

Swapping

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

Though performance is usually affected by swapping process but it helps in running multiple and big processes in parallel and that's the reason **Swapping is also known as a technique for memory compaction**.

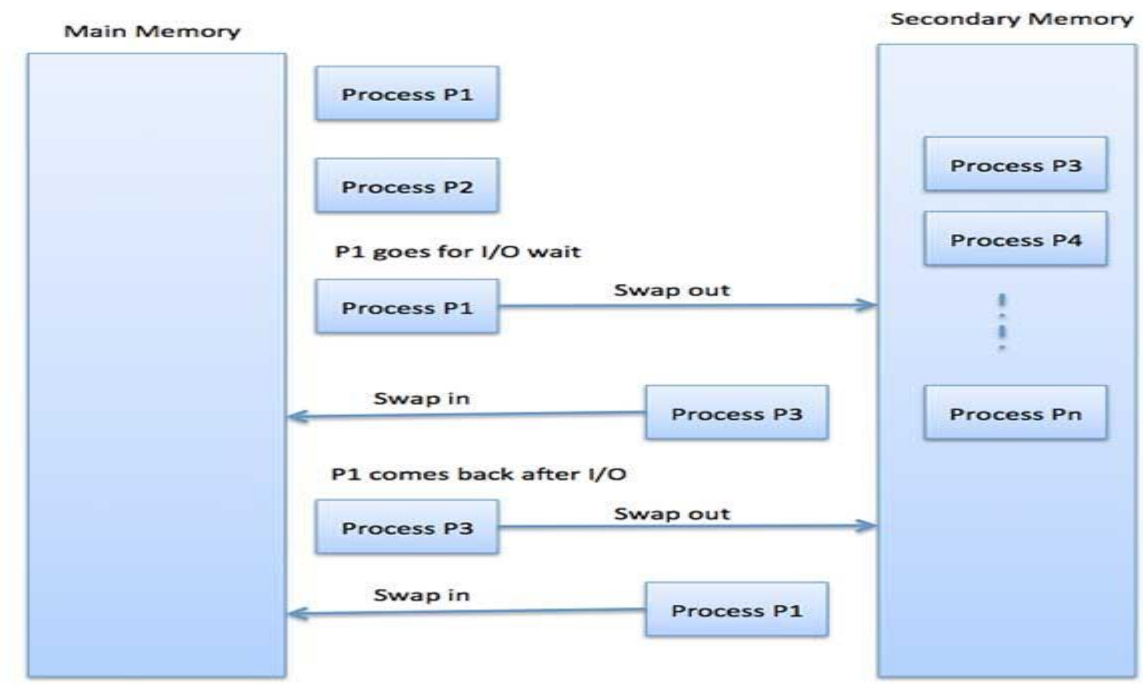


Figure 2 : Swapping

The total time taken by swapping process includes the time it takes to move the entire process to a secondary disk and then to copy the process back to memory, as well as the time the process takes to regain main memory.

Let us assume that the user process is of size 2048KB and on a standard hard disk where swapping will take place has a data transfer rate around 1 MB per second. The actual transfer of the 1000K process to or from memory will take

```
2048KB / 1024KB per second
= 2 seconds
= 2000 milliseconds
```

Now considering in and out time, it will take complete 4000 milliseconds plus other overhead where the process competes to regain main memory.

Process Address Space

The computer interacts via logical and physical addressing to map memory. The operating system takes care of mapping the logical addresses to physical addresses at the time of memory allocation to the program. There are three types of addresses used in a program before and after memory is allocated –

a. Symbolic addresses

   The addresses used in a source code. The variable names, constants, and instruction labels are the basic elements of the symbolic address space.

b. Relative addresses

   At the time of compilation, a compiler converts symbolic addresses into relative addresses.

c. Physical addresses

   The loader generates these addresses at the time when a program is loaded into main memory.

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

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

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

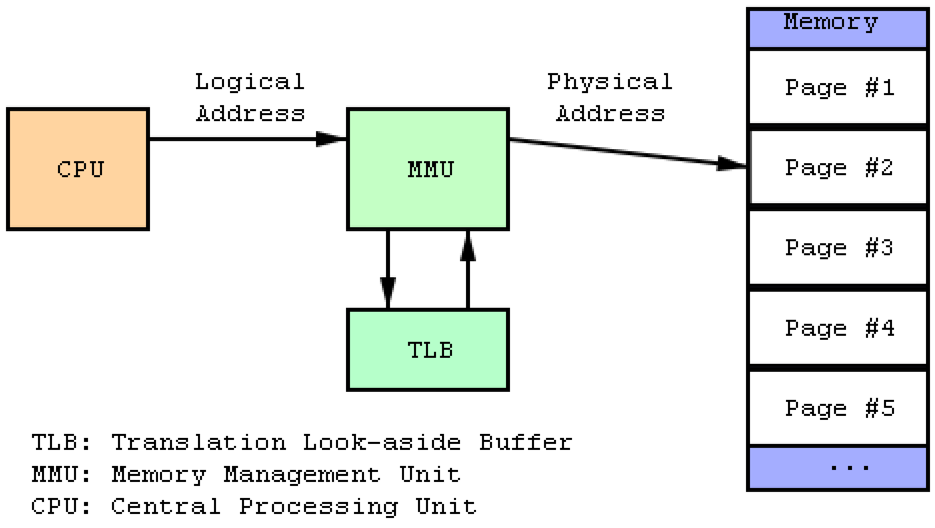

Logical Address
Physical Address
Memory
CPU
MMU
Page #1
Page #2
Page #3
Page #4
Page #5
TLB
...
TLB: Translation Look-aside Buffer
MMU: Memory Management Unit
CPU: Central Processing Unit

Figure 3: Role of MMU

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

Virtual and physical addresses are the same in compile-time and load-time address-binding schemes. Virtual and physical addresses differ in execution-time address-binding scheme.

The set of all logical addresses generated by a program is referred to as a logical address space. The set of all physical addresses corresponding to these logical addresses is referred to as a physical address space.

The runtime mapping from virtual to physical address is done by the memory management unit (MMU) which is a hardware device. MMU uses following mechanism to convert virtual address to physical address.

- The value in the base register is added to every address generated by a user process, which is treated as offset at the time it is sent to memory. For example, if the base register value is 10000, then an attempt by the user to use address location 100 will be dynamically reallocated to location 10100.

- The user program deals with virtual addresses; it never sees the real physical addresses.

CONTIGUOUS ALLOCATION METHODS

The main memory must accommodate both the operating system and the various user processes. We therefore need to allocate different parts of the main memory in the most efficient way possible.

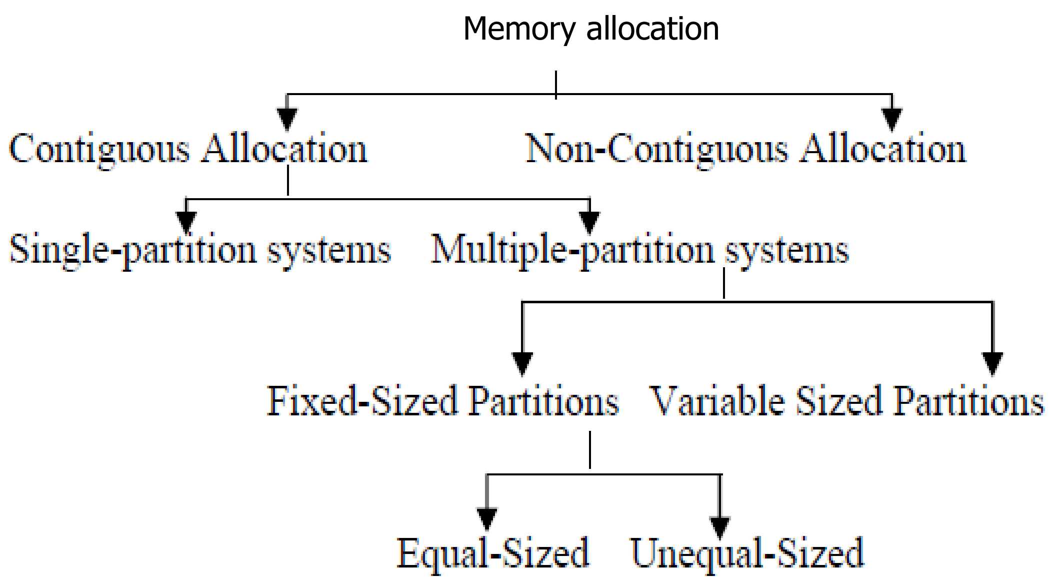
The memory is usually divided into two partitions: one for the resident operating system, and one for the user processes. We may place the operating system in either low memory or high memory. With this approach each process is contained in a single contiguous section of memory.

Memory is divided into several fixed size partitions. Each partition contains exactly one process. When a partition is free, a process is selected from the input queue and loaded into it. The free blocks of memory are known as *holes*. The set of holes is searched to determine which hole is best to allocate.

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

1) Single process system

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



Further we will learn these schemes in next section.

Partitioned Memory allocation:

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

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

Single-Partition System

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

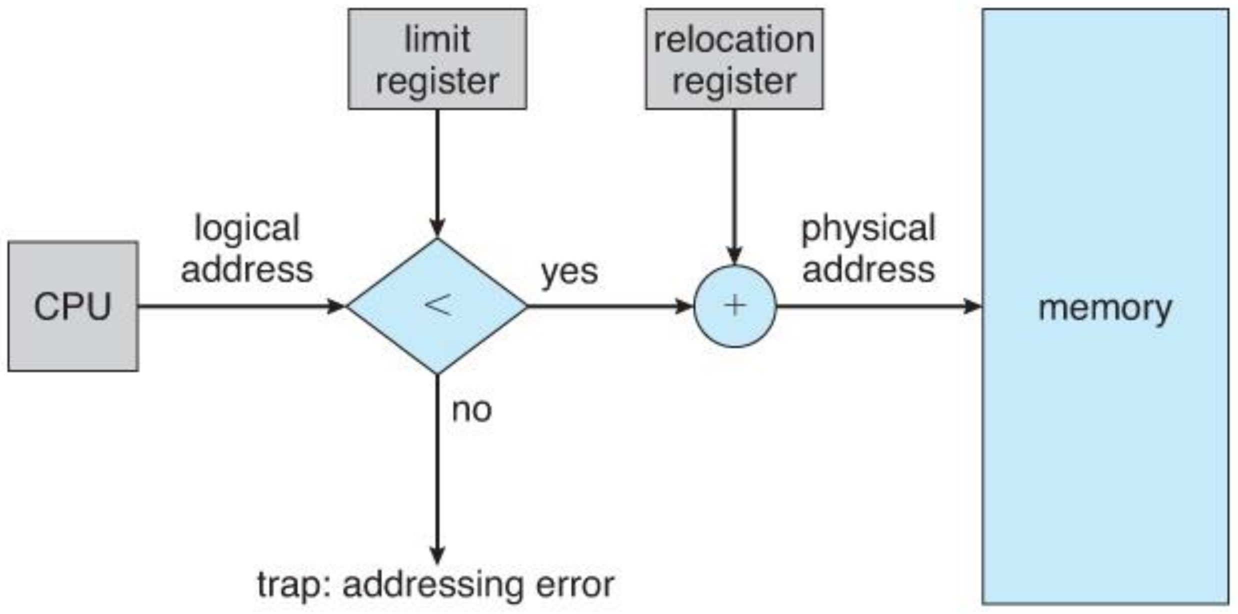


Figure 5: Dynamic Relocation

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

To summarize this, we can say, in dynamic relocation scheme if the logical address space range is 0 to Max then physical address space range is R+0 to R+Max (where R is relocation register contents). Similarly, a limit register is checked by H/W to be sure that logical address generated by CPU is not bigger than size of the program.

Multiple Partition System: Fixed-sized partition

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

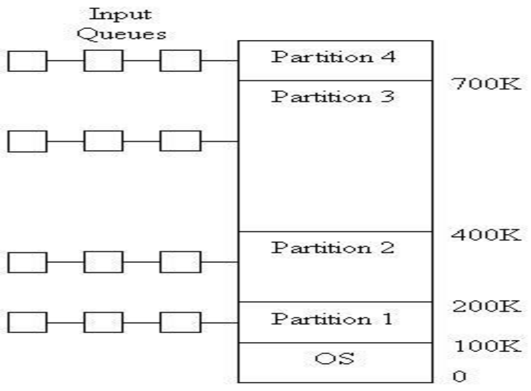


Figure 6: Multiple Partition System

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

Figure 6: Fixed-sized Partition Scheme

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

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

2) Added with set of holes.

Fragmentation

As processes are loaded and removed from memory, the free memory space is broken into little pieces. It happens after sometimes that processes cannot be allocated to memory blocks considering their small size and memory blocks remains unused. This problem is known as Fragmentation.

Fragmentation is of two types –

1 **External fragmentation:** Total memory space is enough to satisfy a request or to reside a process in it, but it is not contiguous, so it cannot be used.

2 **Internal fragmentation:** Memory block assigned to process is bigger. Some portion of memory is left unused, as it cannot be used by another process.

The following diagram shows how fragmentation can cause waste of memory and a compaction technique can be used to create more free memory out of fragmented memory –

External fragmentation can be reduced by compaction or shuffle memory contents to place all free memory together in one large block. To make compaction feasible, relocation should be dynamic.

The internal fragmentation can be reduced by effectively assigning the smallest partition but large enough for the process.

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

Variable-sized Partition:

This scheme is also known as dynamic partitioning. In this scheme, boundaries are not fixed. Processes accommodate memory according to their requirement. There is no wastage as partition size is exactly same as the size of the user process. Initially when processes start this wastage can be avoided but later on when they terminate they leave holes in the main storage. Other processes can accommodate these, but eventually they become too small to accommodate new jobs as shown in Figure 7.

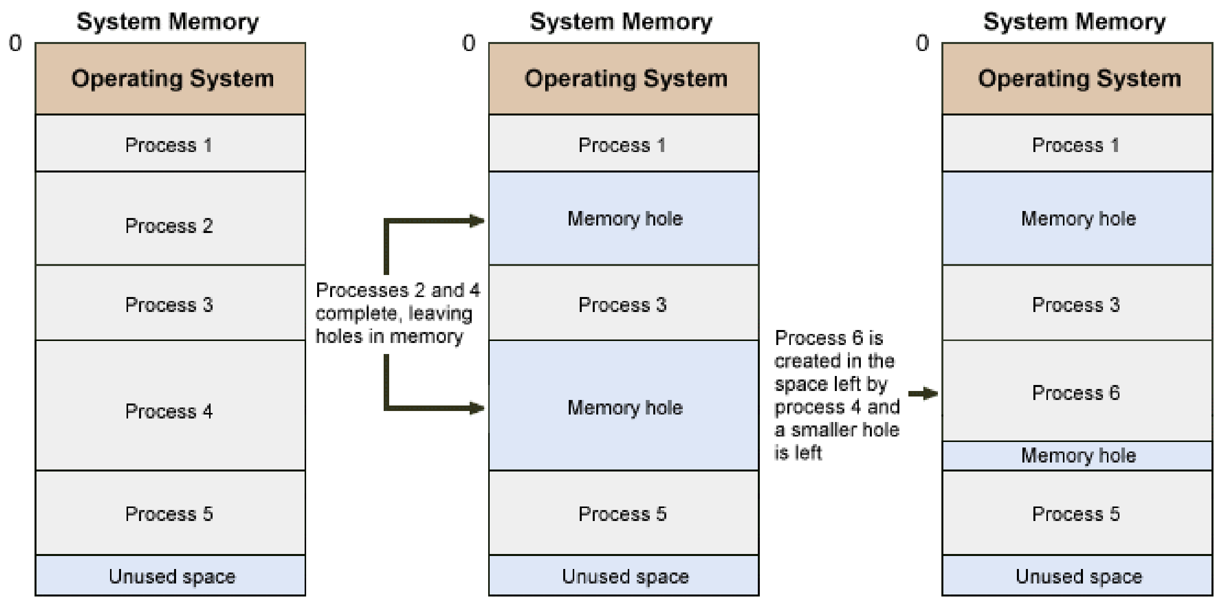


Figure 7: Variable sized partitions

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

External fragmentation problem can be resolved by coalescing holes and storage compaction. Coalescing holes is process of merging existing hole adjacent to a process that will terminate and free its allocated space. Thus, new adjacent holes and existing holes can be viewed as a single large hole and can be efficiently utilised. There is another possibility that holes are distributed throughout the memory. For utilising such scattered holes, shuffle all occupied areas of memory to one end and leave all free memory space as a single large block, which can further be utilised. This mechanism is known as Storage Compaction, as shown in Figure 8.

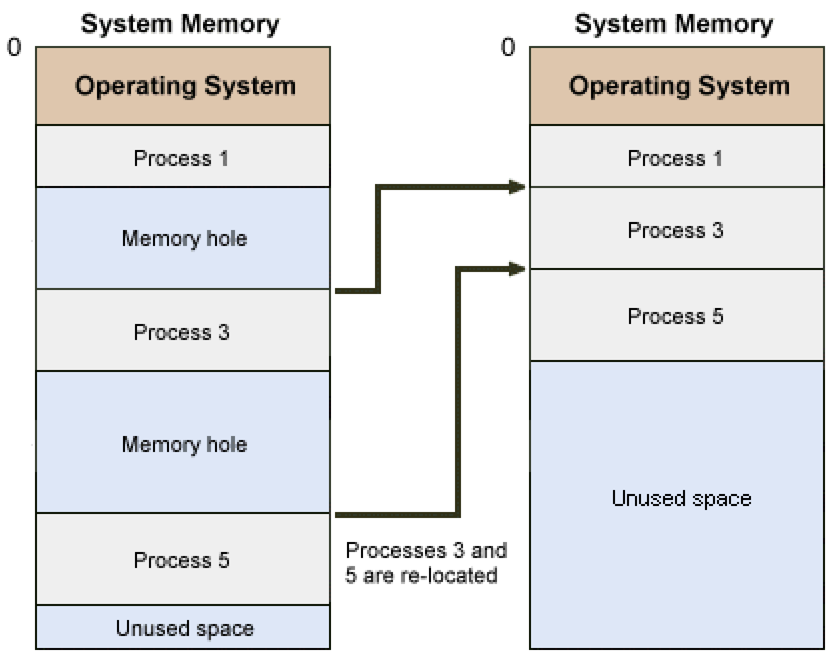


Figure 8: Storage Compaction

But storage compaction also has its limitations as shown below:

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

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

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

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

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

PAGING

A computer can address more memory than the amount physically installed on the system. This extra memory is actually called virtual memory and it is a section of a hard that's set up to emulate the computer's RAM. Paging technique plays an important role in implementing virtual memory.

Paging is a memory management technique in which process address space is broken into blocks of the same size called pages (size is power of 2, between 512 bytes and 8192 bytes). The size of the process is measured in the number of pages.

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

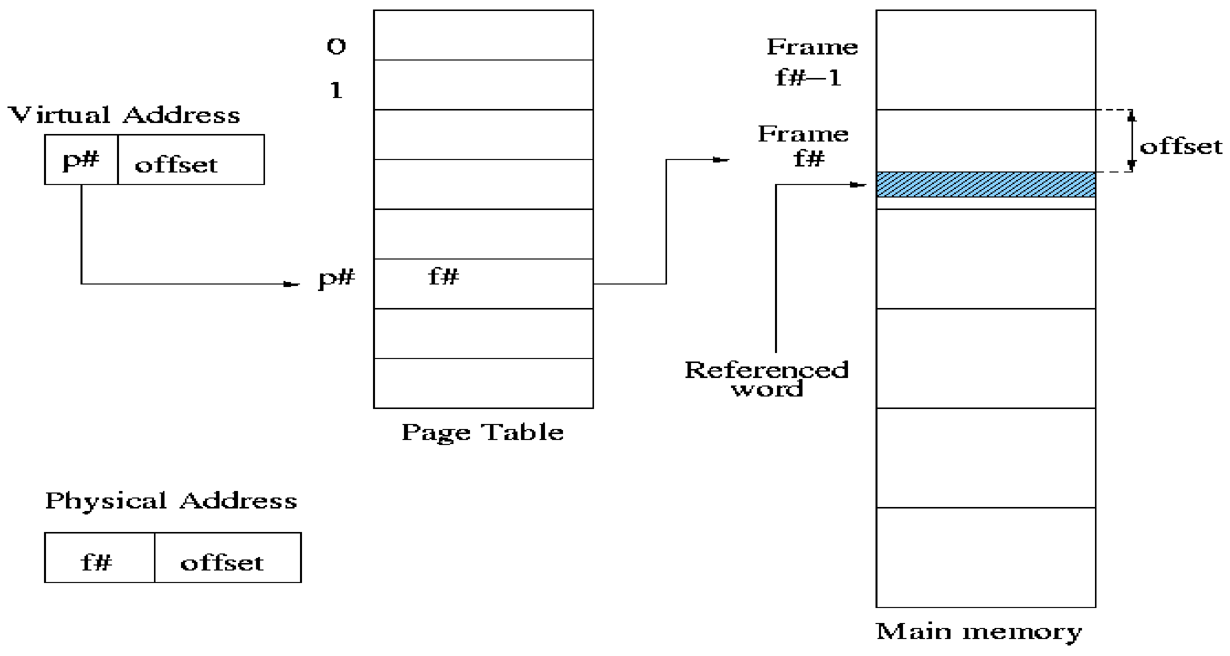


Figure 9: Paging

Hardware Support for Paging

The mapping from virtual to physical address is done by the memory management unit (MMU) which is a hardware device and this mapping is known as paging technique.

- The Physical Address Space is conceptually divided into a number of fixed-size blocks, called frames.
- The Logical address Space is also splitted into fixed-size blocks, called pages.
- Page Size = Frame Size

Let us consider an example:

- Physical Address = 12 bits, then Physical Address Space = 4 K words
- Logical Address = 13 bits, then Logical Address Space = 8 K words
- Page size = frame size = 1 K words (assumption)

Address generated by CPU is divided into

- Page number(p): Number of bits required to represent the pages in Logical Address Space or Page number
- Page offset (d): Number of bits required to represent particular word in a page or page size of Logical Address Space or word number of a page or page offset.

Physical Address is divided into

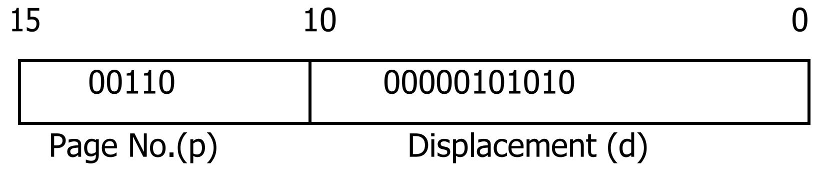
- Frame number (f): Number of bits required to represent the frame of Physical Address Space or Frame number.
- Frame offset (d): Number of bits required to represent particular word in a frame or frame size of Physical Address Space or word number of a frame or frame offset.

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

- A page number (p) in logical address space

- The displacement (or offset) in page p at which item resides (i.e., from start of page).

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



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

The table, which holds virtual address to physical address translations, is called the **page table**. As displacement is constant, so only translation of virtual page number to physical page is required.

Address Translation

Page address is called logical address and represented by page number and the offset.

Logical Address = Page number + page offset

Frame address is called physical address and represented by a frame number and the offset.

Physical Address = Frame number + page offset

A data structure called page map table is used to keep track of the relation between a pages of a process to a frame in physical memory.

There are different types of address translation schemes such as:

(i) Direct mapping address translation
(ii) Associative mapping address translation

Direct Address Translation

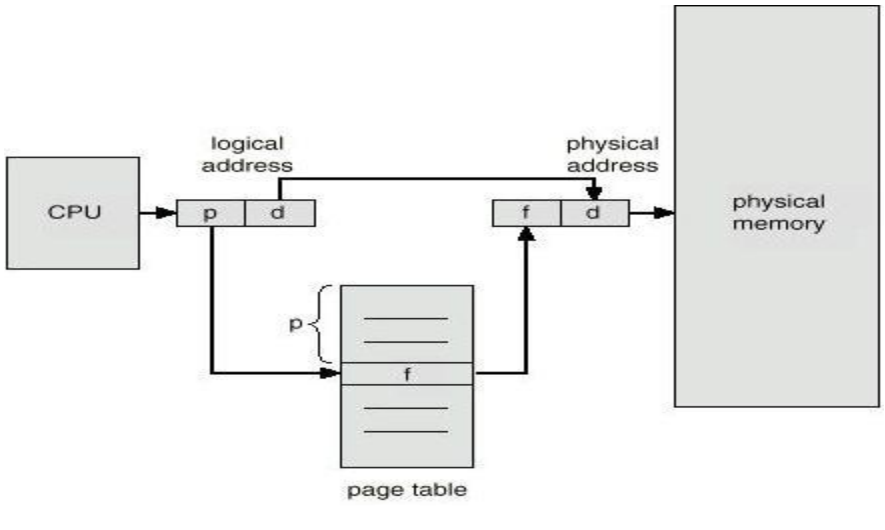


Figure 10: Direct Address Translation Scheme

The above figure shows the direct address translation scheme. When the system allocates a frame to any page, it translates this logical address into a physical address and create entry into the page table to be used throughout execution of the program.

When a process is to be executed, its corresponding pages are loaded into any available memory frames. Suppose you have a program of 8Kb but your memory can accommodate only 5Kb at a given point in time, then the paging concept will come into picture. When a computer runs out of RAM, the operating system (OS) will move idle or unwanted pages of memory to secondary memory to free up RAM for other processes and brings them back when needed by the program. This process continues during the whole execution of the program where the OS keeps removing idle pages from the main memory and write them onto the secondary memory and bring them back when required by the program.

Associative address translation

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

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

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

It is similar to direct mapping scheme but here as TLB's contain only few page table entries, so search is fast. But it is quite expensive due to register support.

So, both direct and associative mapping schemes can also be combined to get more benefits. Here, page number is matched with all associative registers simultaneously. The percentage of the number of times the page is found in TLB's is called hit ratio. If it is not found, it is searched in page table and added into TLB. But if TLB is already full then page replacement policies can be used. Entries in TLB can be limited only. This combined scheme is shown in Figure 11.

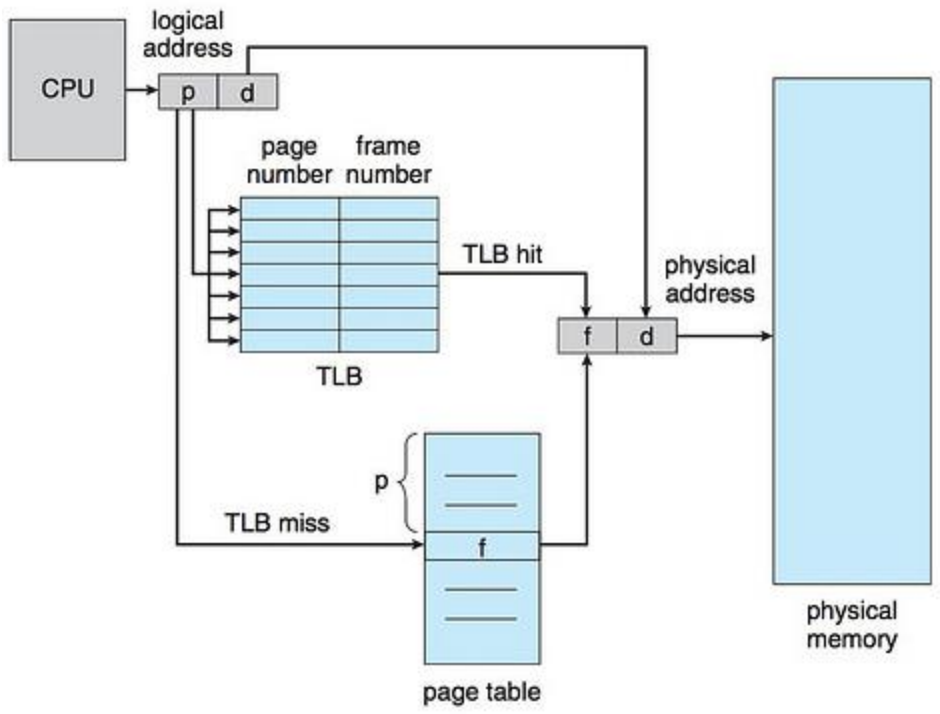


Figure 11: Combined associative/ direct mapping

Page Allocation

In variable sized partitioning of memory every time when a process of size n is to be loaded, it is important to know the best location from the list of available/free holes. This dynamic storage allocation is necessary to increase efficiency and throughput of system. Most commonly used strategies to make such selection are:

a. **Best-fit Policy:** Allocating the hole in which the process fits most "tightly" i.e., the difference between the hole size and the process size is the minimum one.

b. **First-fit Policy:** Allocating the first available hole (according to memory order), which is big enough to accommodate the new process.

c. **Worst-fit Policy:** Allocating the largest hole that will leave maximum amount of unused space i.e., leftover space is maximum after allocation.

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

Protection and Sharing

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

- Virtual address space must be greater than main memory size.i.e., can execute program with large logical address space as compared with physical address space.
- Avoid external fragmentation and hence storage compaction.
- Full utilization of available main storage.

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

Advantages and Disadvantages of Paging

Here is a list of advantages and disadvantages of paging –

- Paging reduces external fragmentation, but still suffer from internal fragmentation.

- Paging is simple to implement and assumed as an efficient memory management technique.
- Due to equal size of the pages and frames, swapping becomes very easy.
- Page table requires extra memory space, so may not be good for a system having small RAM.

SEGMENTATION

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

Segmentation is a memory management technique in which each job is divided into several segments of different sizes, one for each module that contains pieces that perform related functions. Each segment is actually a different logical address space of the program.

When a process is to be executed, its corresponding segmentation are loaded into non-contiguous memory though every segment is loaded into a contiguous block of available memory.

Segmentation memory management works very similar to paging but here segments are of variable-length where as in paging pages are of fixed size.

Address Translation

This mapping between two is done by segment table, which contains segment base and its limit. The segment base has starting physical address of segment, and segment limit provides the length of segment. This scheme is shown below in the figure:

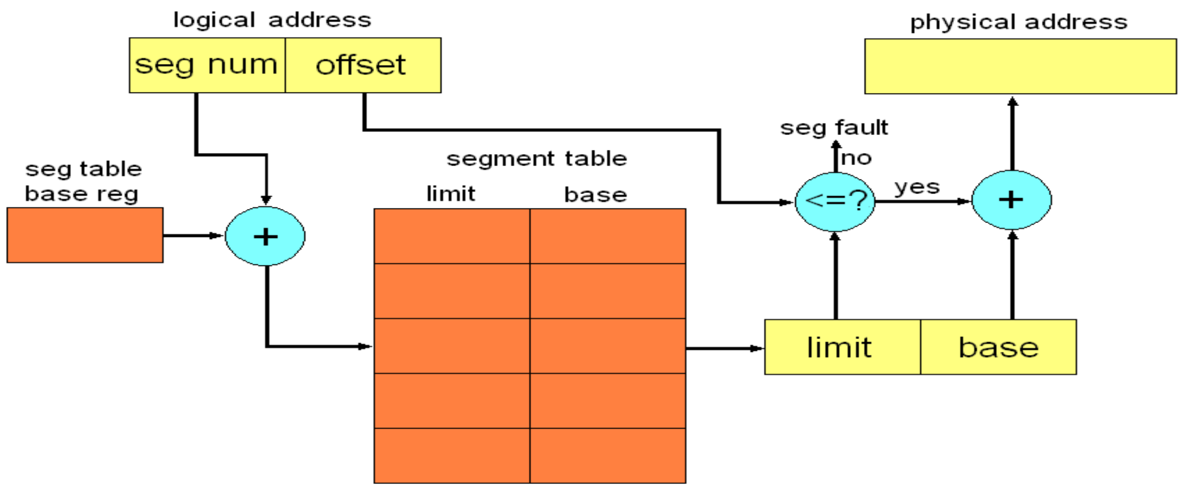
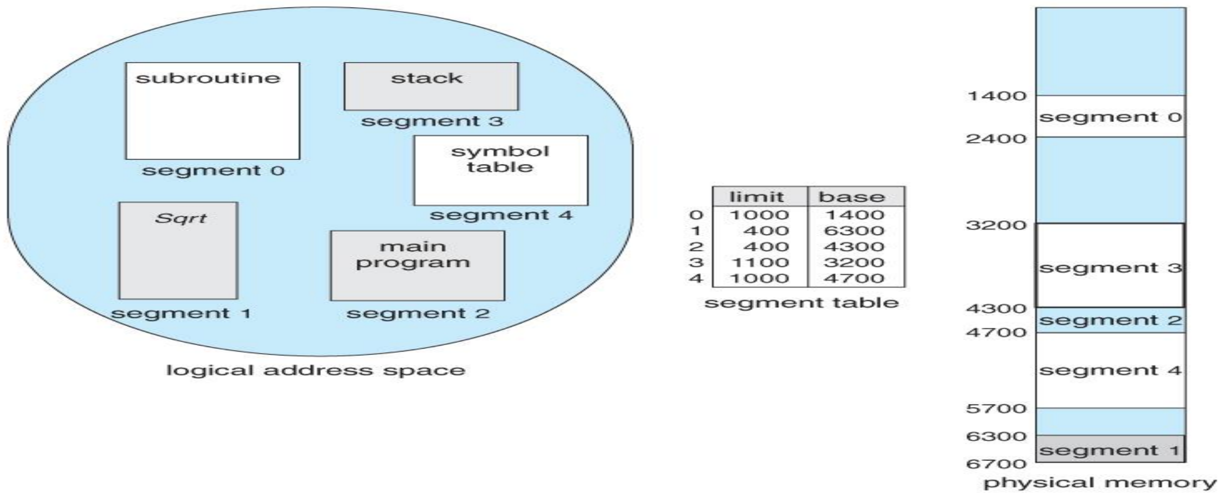


Figure 12: Address Translation

A program segment contains the program's main function, utility functions, data structures, and so on. The operating system maintains a segment map table for every process and a list of free memory blocks along with segment numbers, their size and corresponding memory locations in main memory. For each segment, the table stores the starting address of the segment and the length of the segment. A reference to a memory location includes a value that identifies a segment and an offset.



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

Figure 13: Principle of operation of representation

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

Protection and Sharing

The base limit form of protection is preferred in segmented system. Except for shared segments, separation of distinct address space is enforced by placing different segments in disjoint areas of memory. This method also allows segments that are read-only to be shared, so that two processes can use shared code for better memory efficiency. The implementation is such that no program can read from or write to segments belonging to another program, except the segments that have been set up to be shared. With each segment-table entry protection bit specifying segment as read-only or execute only can be used. Hence illegal attempts to write into a read-only segment can be prevented.

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