Basic Bare Machine

The memory model that can be used for building up any dedicated (application specific) system is called a bare machine. In this bare machine it is assumed that the system does not have any preloaded operating system. The memory model that will be used is a blank memory i.e. no operating system preloaded in the system.

In this model the entire memory is available to the designer. The designer has to decide that where to put the monitor program, which part will be used for storage of data, etc. all those things is decided by the designer himself and there is no limitation put on the designer.

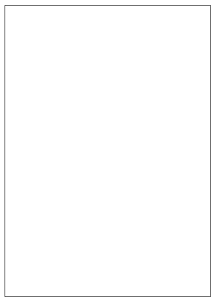


Fig. Bare Machine Memory Model

Resident Monitor

The computer system that we use, the memory model is called resident monitor system. In this memory model a part of the operating system (called monitor) will always reside into the main memory.

In resident monitors the memory is divided into two parts:

- One part is always used by the monitor part of the operating system. This portion of memory is never available to the user program.
- The remaining part of the memory is given to the user for loading user's program and user's data, this part of the memory is known as user area.

In this case if the user program may try to access the operating system area then it will lead to machine crash. To avoid such problem we must have some protection scheme.

To ensure that the user program is not permitted to access the operating system area we need to have some kind of fencing is called a fence address.

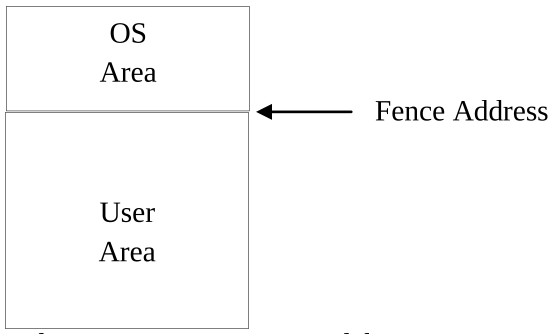


Fig. Resident Monitor Memory Model

Whenever a user program try to access the memory the address which is generated that is to be check with the fence address if the address is less than the fence address then user process is terminated otherwise the generated address is valid address and user program can continue.

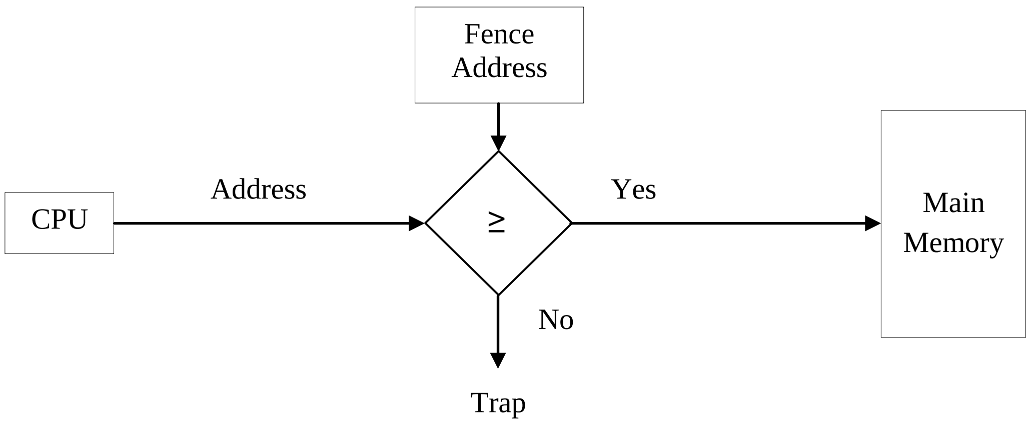


Fig. Address protection with fence address

Swapping

The above memory model (Resident Monitor) is mostly suitable for single user system; this model can be converted to a multi user system by using the concept of 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.

Swap out operation is required when

- A process changes its state from running to waiting or ready.
- A process terminates itself.
- An abnormal termination of process.

Before swap out operation we need to check whether the copy of job is modified into main memory or not. If it is not modified then swap out operation is not required as we are already having the same copy in secondary storage. If it is modified then swap out operation is required.

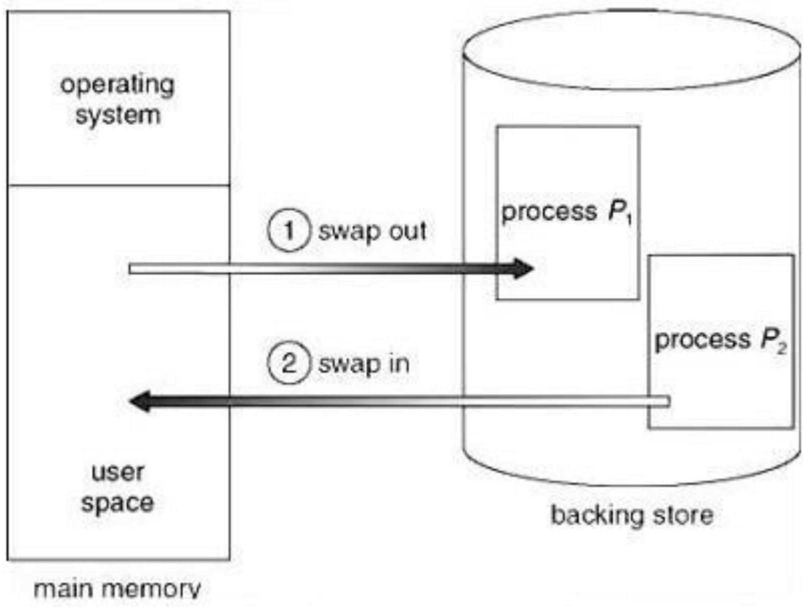


Fig. Swapping of two processes using a disk as a backing store

Multiprogramming with Fixed Partitions

In this model the main memory is divided into a number of partitions. One partition is always allocated to the monitor part of the operating system. User area is divided into a number of partitions. These partitions can be of equal size or different partitions can be of different size. Each of these partitions can contain one user program. Degree of multiprogramming is decided by the number of partitions. In this case switching over from one job to another is simply changing the instruction pointer. To protect every job against other job we need two bound registers: Lower Bound Register & Upper Bound Register.

The protection scheme of this model will be as follows: Whenever CPU generates an address we need to have two comparisons:

- Address should be greater than or equal to lower bound register.
- Address should be less than the upper bound register.

This protection scheme has two comparison operations; it can be modified to be more efficient by reducing the number of comparison operations.

Since the size of every partition is fixed, we have the starting address of the partition and also the length of the each partition. Instead of having ‘Lower Bound Register’ and ‘Upper Bound Registers’ we have ‘Base Register’ and ‘Limit Register’. The value of base register will be same as the lower bound register. CPU assumes that the logical address starts from zero.

Multiprogramming with Fixed Partitions suffers from both internal and external fragmentation.

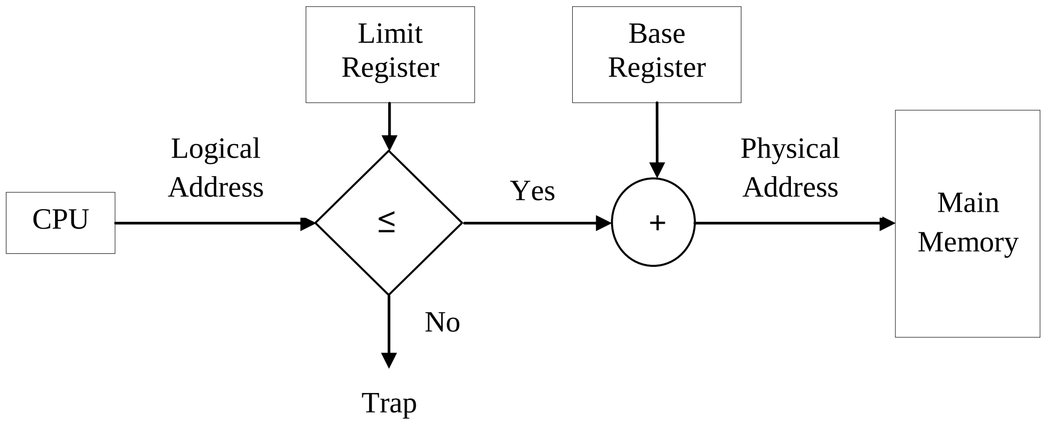


Fig. Protection scheme with Base and Limit Registers

Memory Allocation Problem: When a new job is put into a ready queue, it should be allocated to which partition?

There are many solutions to this problem. The set of holes (available memory) is searched to determine which hole is best to allocate. The first-Fit, Best-Fit, and Worst-Fit strategies are the most common ones used to select a free hole from the set of available holes.

- **First-Fit:** Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.
- **Best-Fit:** Allocate the smallest hole that is big enough. We must search the entire list, unless the list is kept ordered by size. This strategy produces the smallest leftover hole.
- **Worst-Fit:** Allocate the largest hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

Both first-fit and best-fit are better than worst-fit in terms of decreasing both time and storage utilization but first fit is generally faster.

Question: Given memory partitions of 100K, 500K, 200K, 300K, and 600K (in order). How would each of the first-fit, Best-fit and worst-fit algorithms place processes of 212K, 417K, 112K and 426K (in order)? Which algorithm makes the most efficient use of memory?

Solution: *Refer you class notebook (Discussed during the lecture)*
Answer:
First Fit
212K is put in 500K partition
417K is put in 600K partition
112K is put in 288K partition (new partition 288K = 500K - 212K)
426K must wait
Best Fit
212K is put in 300K partition
417K is put in 500K partition
112K is put in 200K partition

426K is put in 600K partition

Worst Fit
212K is put in 600K partition
417K is put in 500K partition
112K is put in 388K partition
426K must wait
In this example, Best-fit turns out to be the best.

To implement Multiprogramming with Fixed Partitions some extra information needs to be maintained. This information is used by the memory management module for the allocation of memory. This information includes:

- What are the memory partitions in the memory?
- What is the size of every partition?
- What is the starting location of every partition?
- What is the status of every partition (i.e. whether the partition is already allocated or free)?

While allocating the partition, the system will only search those partitions whose status is free.
The above information is maintained in a table called **Partition Allocation Table.**

Fragmentation

Fragmentation is a phenomenon in which storage space is used inefficiently, reducing capacity or performance and often both. It leads to storage space being "wasted".

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.

Types of Fragmentation

Fragmentation is of two types −

1. **External Fragmentation**: External Fragmentation happens when a dynamic memory allocation algorithm allocates some memory and a small piece is left over that cannot be effectively used. If too much external fragmentation occurs, the amount of usable memory is drastically reduced. Total memory space exists to satisfy a request, but it is not contiguous.
2. **Internal Fragmentation**: Internal fragmentation is the space wasted inside of allocated memory blocks because of restriction on the allowed sizes of allocated blocks. Allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used

Depending on the total amount of memory storage and the average process size, external fragmentation may be a minor or a major problem. Statistical analysis of first fit, for instance, reveals that, even with some optimization, given N allocated blocks, another *0.5N* blocks will be lost due to fragmentation. That is, one-third of memory may be unusable! This property is known as the ***50-percent rule***.

Solution to Fragmentation

1. **Compaction:** The goal is to shuffle the memory contents to place all free memory together in one large block. Compaction is not always possible. The simplest compaction algorithm is simply to move all processes toward one end of memory; all holes move in the other direction, producing one large hole of available memory. This scheme can be expensive.
2. **Paging**
3. **Segmentation**

Difference between External and Internal Fragmentation

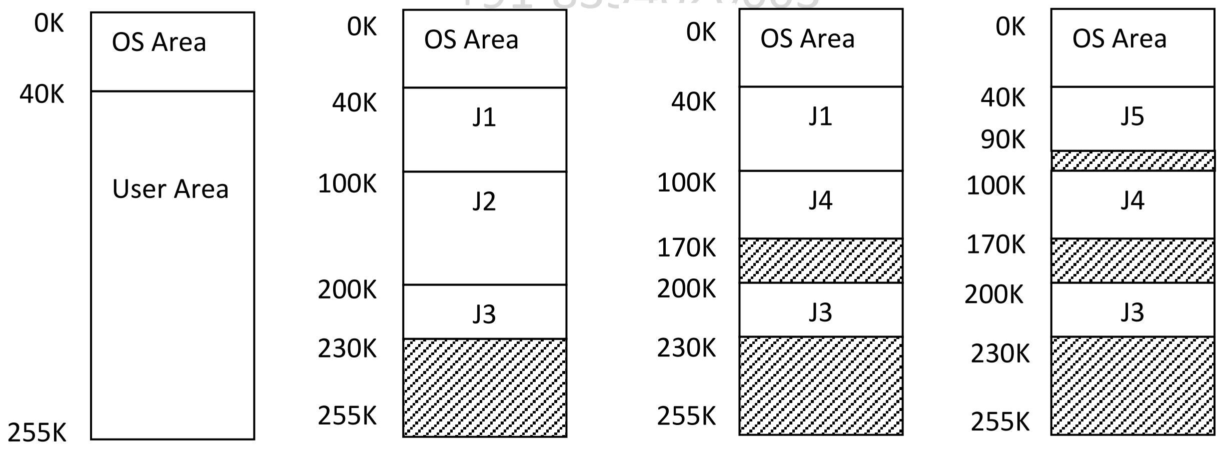
External Fragmentation	Internal Fragmentation
The difference between the memory allocated and the required memory is called internal fragmentation.	The unused spaces formed between non-contiguous memory fragments are too small to serve a new process request. This is called external fragmentation.
It occurs when memory is divided into fixed size blocks regardless of the size of the process.	It occurs when memory is allocated to processes dynamically based on process requests.
It refers to the unused space in the partition which resides with an allocated region.	It refers to the unused memory blocks that are too small to handle a request.
It can be eliminated by: • Allocating memory to processes dynamically.	It can be eliminated by : • Compaction • Paging • segmentation

Multiprogramming with variable partitions

In case of multiprogramming with variable partition we don’t have any pre-decided partition rather the partitions are created of appropriate size when they are needed. There is no fixed number of partitions. The number of partition will vary upon the number of jobs. In this memory model the degree of multiprogramming is also variable.

Initially memory is divided into two partitions. One operating system area and other is user area. Let us assume we have a memory of 256K out of which 40K is used by operating system area. We have a number of jobs. The CPU scheduling used is FCFS.

Job Number	Memory Requirement	Burst Time
J1	60K	10
J2	100K	5
J3	30K	20
J4	70K	8
J5	50K	15
J6	60K	9



Initially After 5 time unit After 10 time unit

Paging/ Paged Memory Management

Paging is a memory-management scheme that permits the physical-address space of a process to be non contiguous. Paging avoids the considerable problem of fitting the varying-sized memory chunks onto the backing store, from which most of the previous memory-management schemes suffered.

Physical memory is broken into fixed-sized blocks called frames. Logical memory is also broken into blocks of the same size called ***pages***. When a process is to be executed, its pages are loaded into any available memory frames from the backing store. The backing store is divided into fixed-sized blocks that are of the same size as the memory frames. The hardware support for paging is illustrated in Fig.

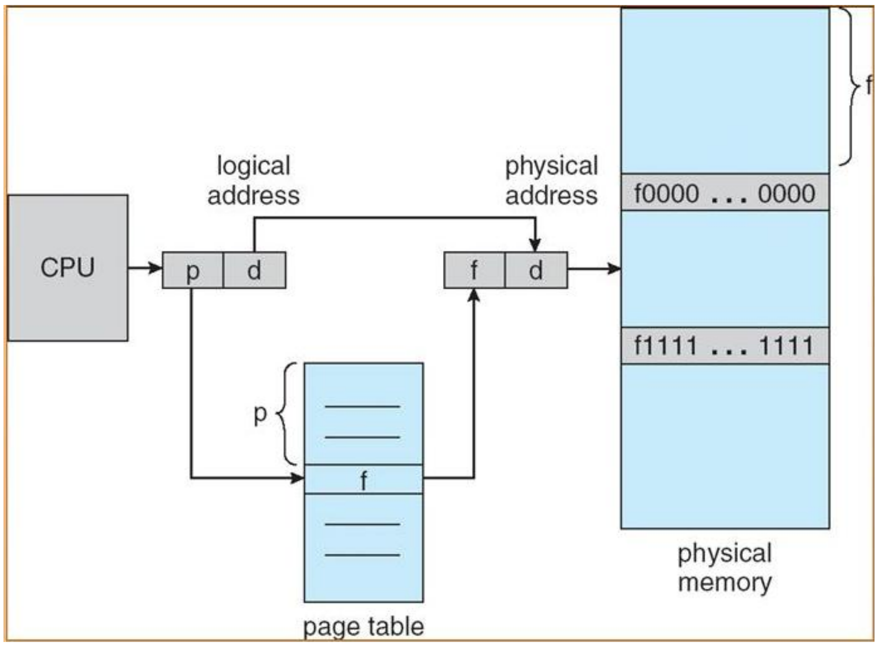


Fig. Paging Hardware

Every address generated by the CPU is divided into two parts: a 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 page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit. The paging model of memory is shown in Fig.

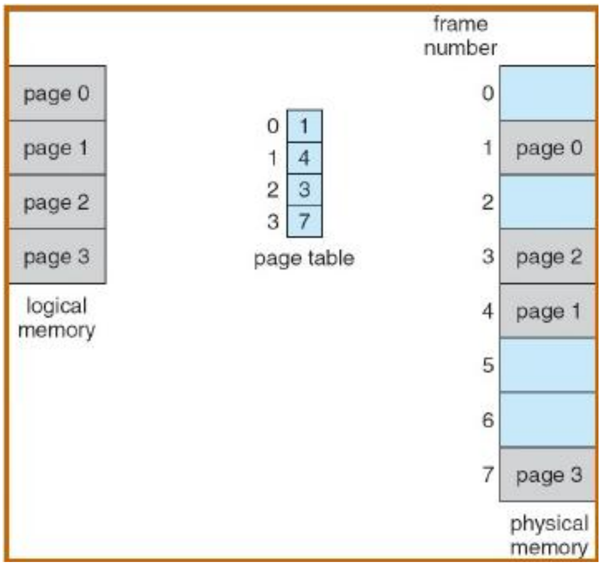


Fig. Paging model of logical and physical memory

The page size (like the frame size) is defined by the hardware. The size of a page is typically a power of 2, varying between 512 bytes and 16 MB per page, depending on the computer architecture. The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy. If the size of logical-address space is 2^m, and a page size is 2^n addressing units (bytes or words), then the high-order $m- n$ bits of a logical address designate the page number, and the n low-order bits designate the page offset. Thus, the logical address is as follows:

page number	page offset
p	d
$m - n$	n

Where p is an index into the page table and d is the displacement within the page.

Example: Consider the memory in Fig. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages)

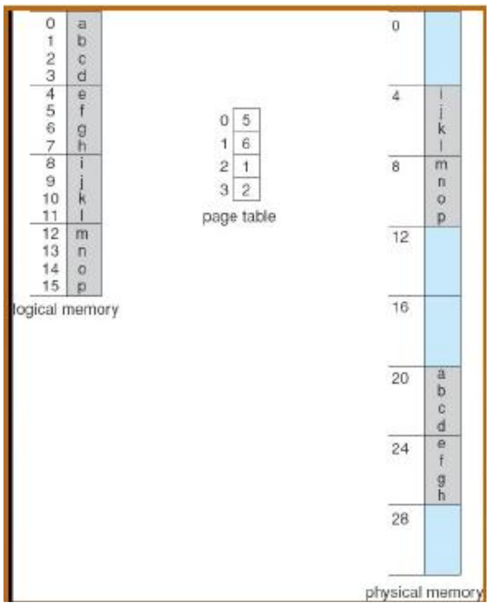


Fig. Paging example

Logical address 0 (page 0, offset 0) maps to physical address 20 (= (5 x 4) + 0).
Logical address 3 (page 0, offset 3) maps to physical address 23 (= (5 x 4) + 3).
Logical address 4 (page 1, offset 0) maps to physical address 24 (= (6 x 4) + 0).
Logical address 13 (page 4, offset 1) maps to physical address 9 (= (2 x 4) + 1).

Advantages and Disadvantages of Paging

- Paging reduces external fragmentation, but still suffers 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.

Protection

Memory protection in a paged environment is accomplished by protection bits associated with each frame. Normally, these bits are kept in the page table. One bit can define a page to be read-write or read-only. Every reference to memory goes through the page table to find the correct frame number. At the same time that the physical address is being computed, the protection bits can be checked to verify that no writes are being made to a read-only page. An attempt to write to a read-only page causes a hardware trap to the operating system (or memory-protection violation).

One more bit is generally attached to each entry in the page table: a **valid invalid** bit. When this bit is set to "valid," this value indicates that the associated page is in the process' logical-address space, and is thus a legal (or valid) page. If the bit is set to "invalid," this value indicates that the page is not in the process' logical-address space. Illegal addresses are trapped by using the valid invalid bit. The operating system sets this bit for each page to allow or disallow accesses to that page.

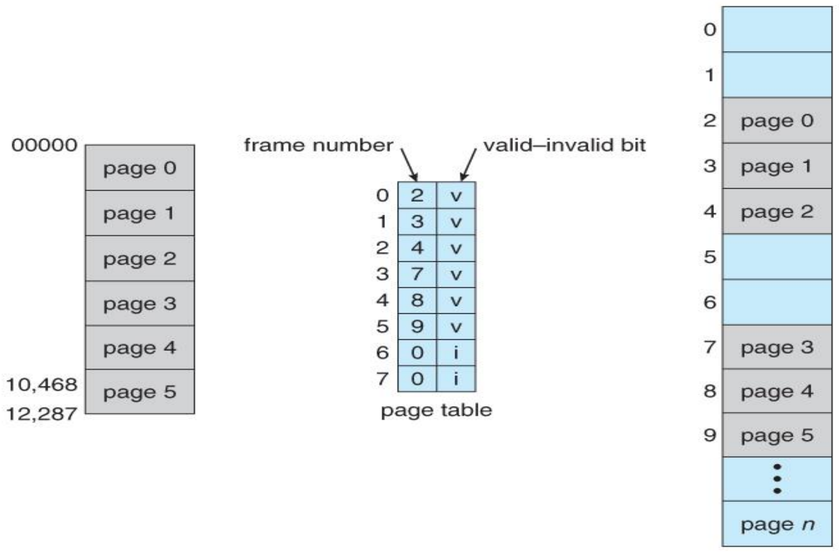


Fig. Valid or Invalid bit in a page table

Segmentation/ Segmented Memory Management

Segmentation is a memory-management scheme that supports modular programming view of memory. A logical-address space is a collection of segments. Each segment has a segment number and a length. The addresses specify both the segment number and the offset within the segment. The user therefore specifies each address by two quantities: a segment number and an offset. Thus, a logical address consists of a two tuple:

<segment-number, offset>

Each entry of the segment table has a segment base and a segment limit. The segment base contains the starting physical address where the segment resides in memory, whereas the segment limit specifies the length of the segment.

A logical address consists of two parts: a segment number, s, and an offset into that segment, d. The segment number is used as an index into the segment table. The offset d of the logical address must be between 0 and the segment limit. If it is not, we trap to the operating system (logical addressing attempt beyond end of segment). If this offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte.

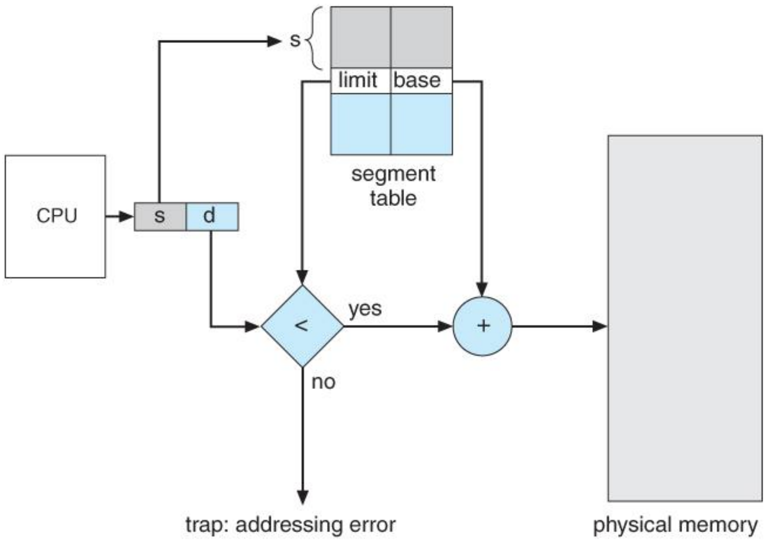
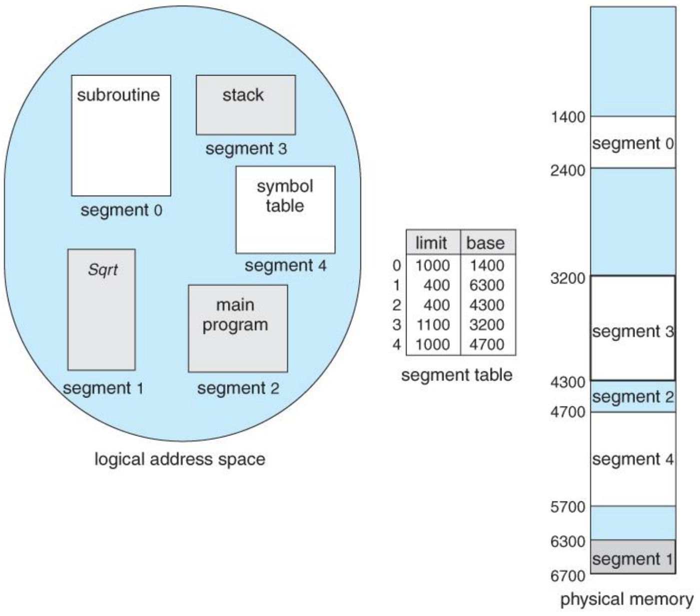


Fig. Segmentation Hardware

Example: Consider the situation shown in Fig .



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

Fig. Example of segmentation

We have five segments numbered from 0 through 4. The segments are stored in physical memory as shown. The segment table has a separate entry for each segment giving the beginning address of the segment in physical memory (or base) and the length of that segment (or limit). For example, segment 2 is 400 bytes long and begins at location 4300. Thus, a reference to byte 53 of segment 2 is mapped onto location 4300 + 53 = 4353.
A reference to byte 852 of segment 3 is mapped onto location 3200 + 852 = 4052.
A reference to byte 1222 of segment 0 would result in a trap to the operating system, as this segment is only 1,000 bytes long.