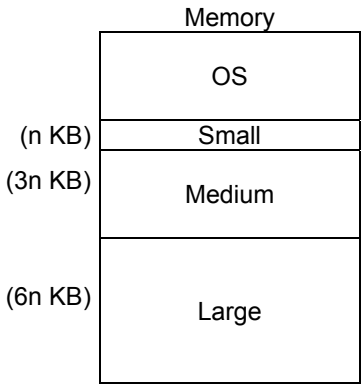
# Chapter 3
# Memory Management

In a multiprogramming system, in order to share the processor, a number of processes must be kept in memory. Memory management is achieved through memory management algorithms. Each memory management algorithm requires its own hardware support. In this chapter, we shall see the partitioning, paging and segmentation methods.

In order to be able to load programs at anywhere in memory, the compiler must generate relocatable object code. Also we must make it sure that a program in memory, addresses only its own area, and no other program's area. Therefore, some protection mechanism is also needed.

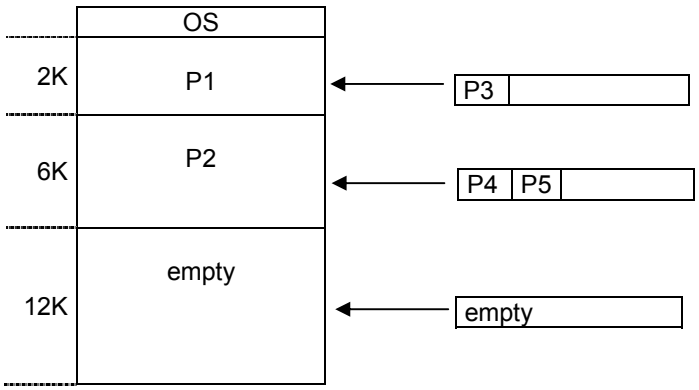
## 3.1 Fixed Partitioning




In this method, memory is divided into partitions whose sizes are fixed. OS is placed into the lowest bytes of memory. Processes are classified on entry to the system according to their memory they requirements. We need one *Process Queue (PQ)* for each class of process. If a process is selected to allocate memory, then it goes into memory and competes for the processor. The number of fixed partition gives the degree of multiprogramming. Since each queue has its own memory region, there is no competition between queues for the memory.

**Fixed Partitioning with Swapping**

This is a version of fixed partitioning that uses RRS with some time quantum. When time quantum for a process expires, it is swapped out of memory to disk and the next process in the corresponding process queue is swapped into the memory.



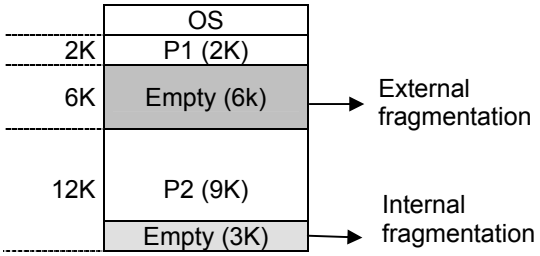
Normally, a process swapped out will eventually be swapped back into the same partition. But this restriction can be relaxed with dynamic relocation.

In some cases, a process executing may request more memory than its partition size. Say we have a 6 KB process running in 6 KB partition and it now requires a more memory of 1 KB. Then, the following policies are possible:

- Return control to the user program. Let the program decide either quit or modify its operation so that it can run (possibly slow) in less space.
- Abort the process. (The user states the maximum amount of memory that the process will need, so it is the user's responsibility to stick to that limit)
- If dynamic relocation is being used, swap the process out to the next largest PQ and locate into that partition when its turn comes.

The main problem with the fixed partitioning method is how to determine the number of partitions, and how to determine their sizes.

If a whole partition is currently not being used, then it is called *an external fragmentation*. And if a partition is being used by a process requiring some memory smaller than the partition size, then it is called an *internal fragmentation*.




In this composition of memory, if a new process, P3, requiring 8 KB of memory comes, although there is enough total space in memory, it can not be loaded because fragmentation.

## 3.2 Variable Partitioning

With fixed partitions we have to deal with the problem of determining the number and sizes of partitions to minimize internal and external fragmentation. If we use variable partitioning instead, then partition sizes may vary dynamically.

In the variable partitioning method, we keep a table (linked list) indicating used/free areas in memory. Initially, the whole memory is free and it is considered as one large block. When a new process arrives, the OS searches for a block of free memory large enough for that process. We keep the rest available (free) for the future processes. If a block becomes free, then the OS tries to merge it with its neighbors if they are also free.

There are three algorithms for searching the list of free blocks for a specific amount of memory.

First Fit : Allocate the first free block that is large enough for the new process. This is a fast algorithm.

Best Fit : Allocate the smallest block among those that are large enough for the new process. In this method, the OS has to search the entire list, or it can keep it sorted and stop when it hits an entry which has a size larger than the size of new process. This algorithm produces the smallest left over block. However, it requires more time for searching all the list or sorting it.

Worst Fit : Allocate the largest block among those that are large enough for the new process. Again a search of the entire list or sorting it is needed. This algorithm produces the largest over block.

**Example 3.1**
Consider the following memory map and assume a new process P4 comes with a memory requirement of 3 KB. Locate this process.

| OS |
|---|
| P1 |
| <free> 10 KB |
| P2 |
| <free> 16 KB |
| P3 |
| <free> 4 KB |

a. First fit algorithm allocates from the 10 KB block.
b. Best fit algorithm allocates from the 4 KB block.
c. Worst fit algorithm allocates from the 16 KB block.

New memory arrangements with respect to each algorithms will be as follows:

| OS |
|---|
| P1 |
| P4 |
| <free> 7 KB |
| P2 |
| <free> 16 KB |
| P3 |
| <free> 4 KB |

First Fit

| OS |
|---|
| P1 |
| <free> 10 KB |
| P2 |
| <free> 16 KB |
| P3 |
| P4 |
| <free> 1 KB |

Best Fit

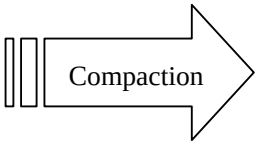
| OS |
|---|
| P1 |
| <free> 10 KB |
| P2 |
| P4 |
| <free> 13 KB |
| P3 |
| <free> 4 KB |

Worst Fit

At this point, if a new process, P5 of 14K arrives, then it would wait if we used worst fit algorithm, whereas it would be located in cases of the others.

**Compaction:** Compaction is a method to overcome the external fragmentation problem. All free blocks are brought together as one large block of free space. Compaction requires dynamic relocation. Certainly, compaction has a cost and selection of an optimal compaction strategy is difficult. One method for compaction is swapping out those processes that are to be moved within the memory, and swapping them into different memory locations.

| OS |
|---|
| P1 |
| <free> 20 KB |
| P2 |
| <free> 7 KB |
| P3 |
| <free> 10 KB |




| OS |
|---|
| P1 |
| P2 |
| P3 |
| <free> 37 KB |

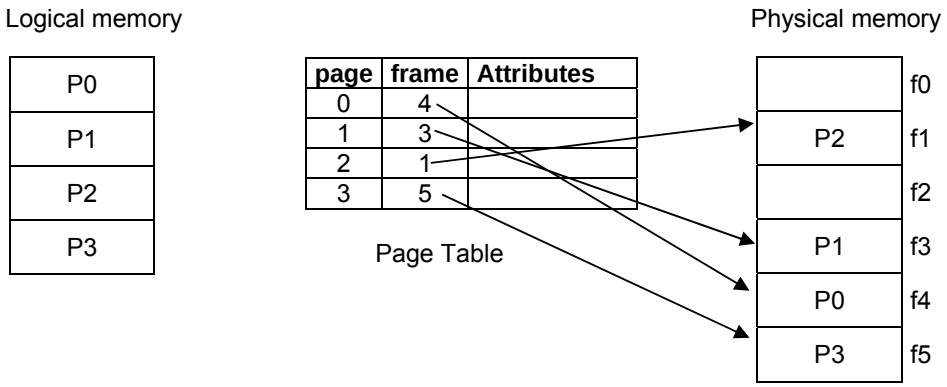
## 3.3 Paging

Paging permits a program to allocate noncontiguous blocks of memory. The OS divide programs into pages which are blocks of small and fixed size. Then, it divides the physical memory into frames which are blocks of size equal to page size. The OS uses a *page table* to map program pages to memory frames. Page size (S) is defined by the hardware. Generally page size is chosen as a power of 2 such as 512 words/page or 4096 words/page etc.

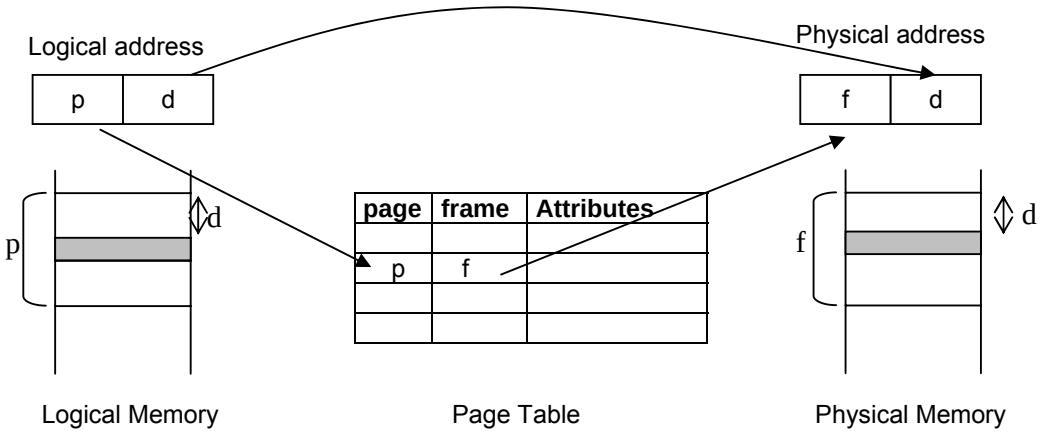
With this arrangement, the words in the program have an address called as *logical address*. Every logical address is formed of

- A page number *p* where p = logical address div S
- An offset *d* where d = logical address mod S

When a logical address <p, d> is generated by the processor, first the frame number f corresponding to page p is determined by using the page table and then the physical address is calculated as (f*S+d) and the memory is accessed.

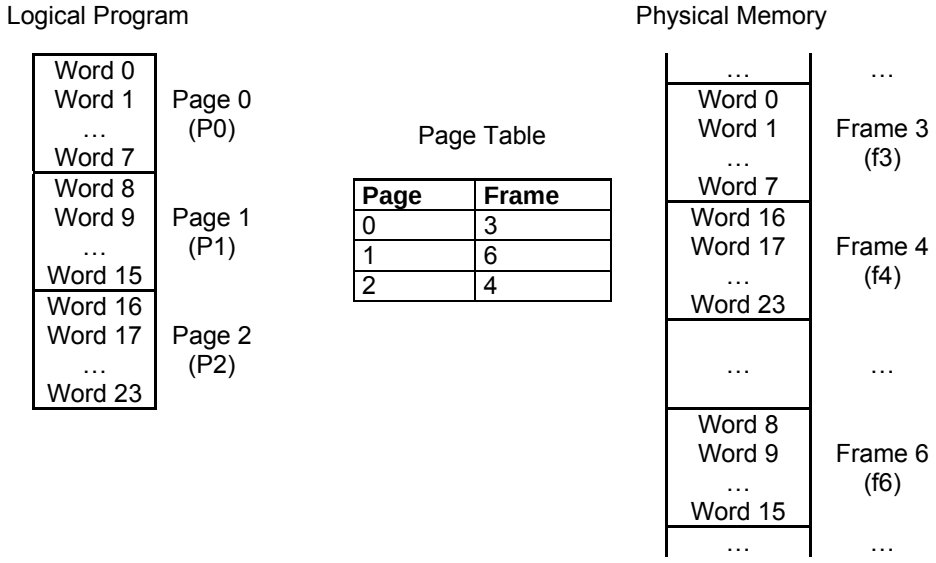



The address translation in paging is shown below



**Example 3.2**

Consider the following information to form a physical memory map.
Page Size = 8 words
Physical Memory Size = 128 words
A program of 3 pages where P0 → f3; P1 → f6; P2 → f4



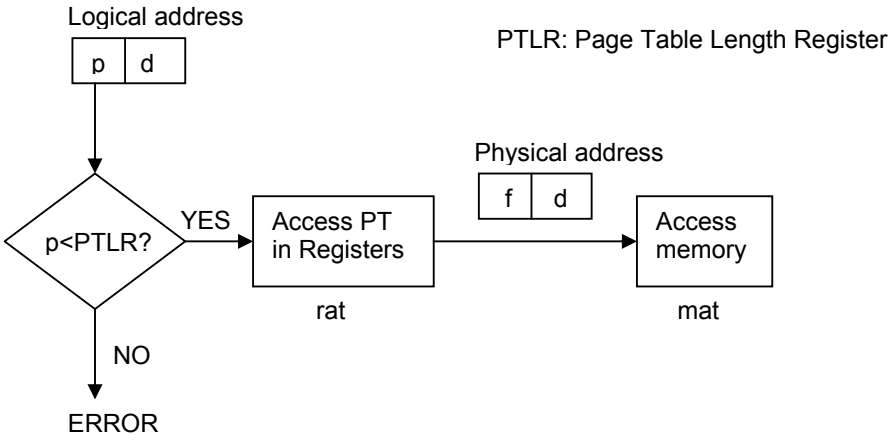

| Program Line | Logical Address | Offset | Page Number | Frame Number | Physical Address |
|---|---|---|---|---|---|
| Word 0 | 00 000 | 000 | 00 | 011 | 011 000 |
| Word 1 | 00 001 | 001 | 00 | 011 | 011 001 |
| … | … | … | … | … | … |
| Word 7 | 00 111 | 111 | 00 | 011 | 011 111 |
| Word 8 | 01 000 | 000 | 01 | 110 | 110 000 |
| Word 9 | 01 001 | 001 | 01 | 110 | 110 001 |
| … | … | … | … | … | … |
| Word 15 | 01 111 | 111 | 01 | 110 | 110 111 |
| Word 16 | 10 000 | 000 | 10 | 100 | 100 000 |
| Word 17 | 10 001 | 001 | 10 | 100 | 100 001 |
| … | | | | | |
| Word 23 | 10 111 | 111 | 10 | 100 | 100 111 |

**How to Implement The Page Table?**

Every access to memory should go through the page table. Therefore, it must be implemented in an efficient way.

a. Using fast dedicated registers

Keep page table in fast dedicated registers. Only the OS is able to modify these registers. However, if the page table is large, this method becomes very expensive since requires too many registers.

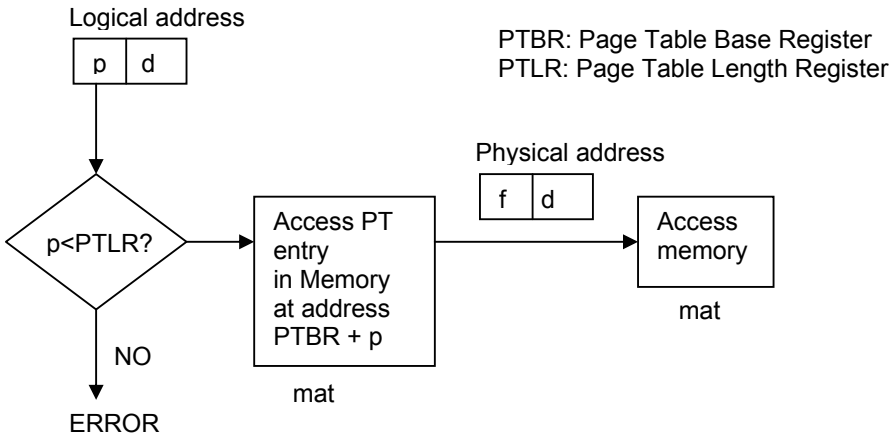



Given a logical address to access the word in physical memory, first access the PT stored in registers, which requires register access time (rat), and then find out the physical address and access the physical memory, which requires memory access time (mat). Therefore effective memory access time (emat) becomes:

emat= rat + mat

b. Keep the page table in main memory

In this method, the OS keeps a page table in the memory. But this is a time consuming method. Because for every logical memory reference, two memory accesses are required:
1. To access the page table in the memory, in order to find the corresponding frame number.
2. To access the memory word in that frame



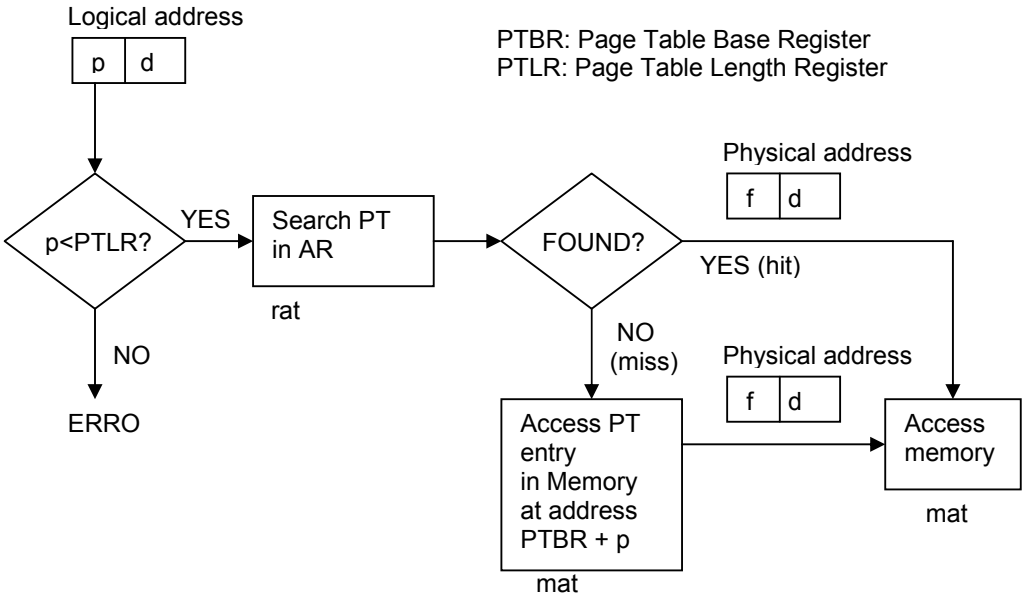

.

In this approach emat is:

emat= 2 * mat

c. Use content-addressable associative registers

These are small, high speed registers built in a special way so that they permit an associative search over their contents. That is, all registers may be searched in one machine cycle simultaneously. However, associative registers are quite expensive. So, a small number of them should be used.




When a logical memory reference is made, first the corresponding page number is searched in associative registers. If that page number is found in one associative register (hit) then the corresponding frame number is get, else (miss) the page table in memory is accessed to find the frame number and that <page number, frame number> pair is stored into associative registers. Once the frame number is obtained, the memory word is accessed.

The *hit ratio* is defined as the percentage of times that a page number is found in associative registers. Hit ratio is important in performance of the system since it affects the effective memory access time. In the case of finding the page number in associative registers, only one memory access time is required whereas if it cannot be found two memory accesses are needed. So, greater the hit ratio, smaller the effective memory access time. Effective memory access time is calculated as fallows:

$$emat = h * emat_{HIT} + (1-h) * emat_{MISS}$$

where

h = The hit ratio
$emat_{HIT}$ = effective memory access time when there is a hit = rat + mat
$emat_{MISS}$ = effective memory access time when there is a miss = rat + mat + mat

**Example 3.3**

Assume we have a paging system which uses associative registers. These associative registers have an access time of 30 ns, and the memory access time is 470 ns. The system has a hit ratio of 90 %.

Now, if the page number is found in one of the associative registers, then the effective access time:

$$emat_{HIT} = 30 + 470 = 500 \text{ ns.}$$

Because one access to associative registers and one access to the main memory is sufficient.

On the other hand, if the page number is not found in associative registers, then the effective access time:

$$emat_{MISS} = 30 + (2 * 470) = 970 \text{ ns.}$$

Since one access to associative registers and two accesses to the main memory iare required.

Then, the emat is calculated as follows:

$$emat = 0.9 * 500 + 0.1 * 970$$
$$= 450 + 97 = 547 \text{ ns}$$

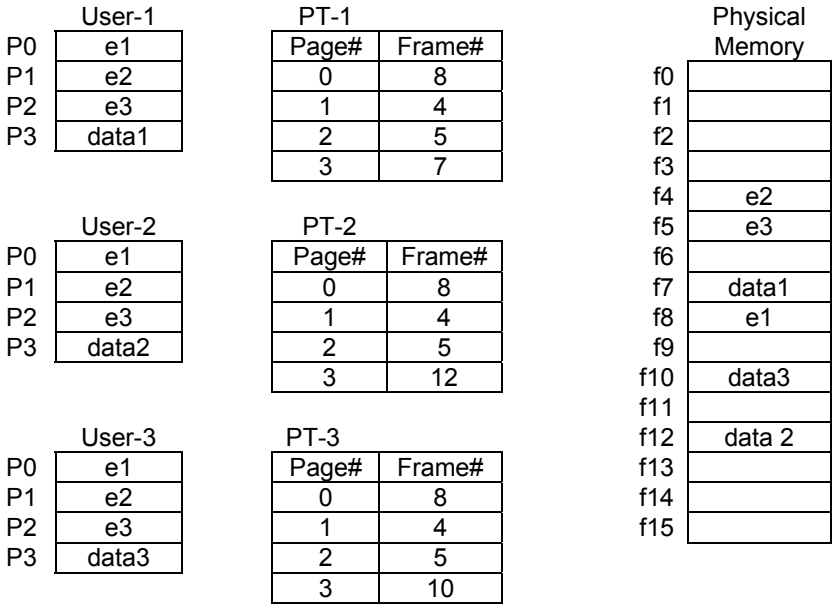
## Sharing Pages

Sharing pages is possible in a paging system, and is an important advantage of paging. It is possible to share system procedures or programs, user procedures or programs, and possibly data area. Sharing pages is especially advantageous in time-sharing systems. A reentrant program (non-self-modifying code = read only) never changes during execution. So, more than one process can execute the same code at the same time. Each process will have its own data storage and its own copy of registers to hold the data for its own execution of the shared program.

**Example 3.4**

Consider a system having page size=30 MB. There are 3 users executing an editor program which is 90 MB (3 pages) in size, with a 30 MB (1 page) data space.

To support these 3 users, the OS must allocate 3 * (90+30) = 360 MB space. However, if the editor program is reentrant (non-self-modifying code = read only), then it can be shared among the users, and only one copy of the editor program is sufficient. Therefore, only 90 + 30 * 3 = 180 MB of memory space is enough for this case.

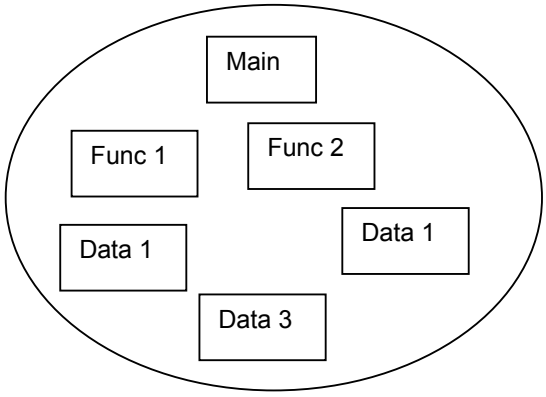



## 3.2 Segmentation

In segmentation, programs are divided into variable size segments, instead of fixed size pages. Every logical address is formed of a segment name and an offset within that segment. In practice, segments are numbered. Programs are segmented automatically by the compiler or assembler.

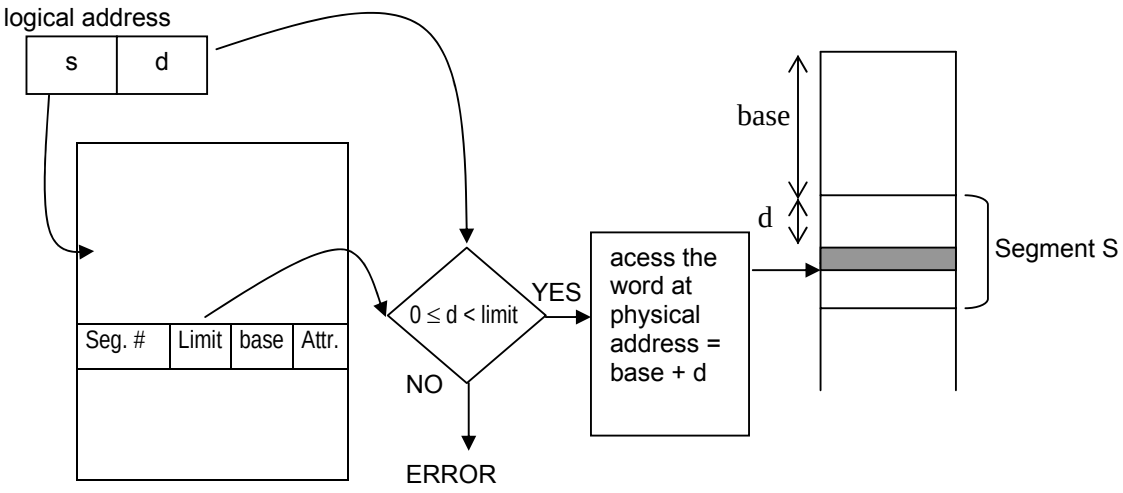
For example, a C compiler will create separate segments for:

1. the code of each function
2. the local variables for each function
3. the global variables.



For logical to physical address mapping, a *segment table* is used. When a logical address <s, d> is generated by the processor:

1. Base and limit values corresponding to segment s are determined using the segment table
2. The OS checks whether d is in the limit. ($0 \leq d < limit$)
3. If so, then the physical address is calculated as (base + d), and the memory is accessed.

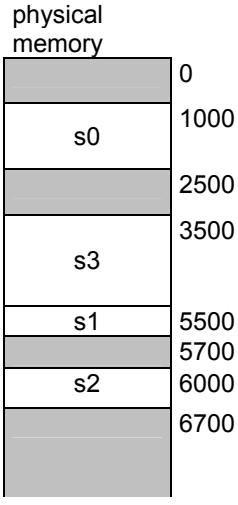



**Example 3.5**

Generate the memory map according to the given segment table. Assume the generated logical address is <1,123>; find the corresponding physical address.

| Segment | Limit | Base |
|---|---|---|
| 0 | 1500 | 1000 |
| 1 | 200 | 5500 |
| 2 | 700 | 6000 |
| 3 | 2000 | 3500 |

Now, check segment table entry for segment 1. The limit for segment 1 is 200. Since 123 < 200, we carry on. The physical address is calculated as 5500 + 123 = 5623, and the memory word 5623 is accessed.



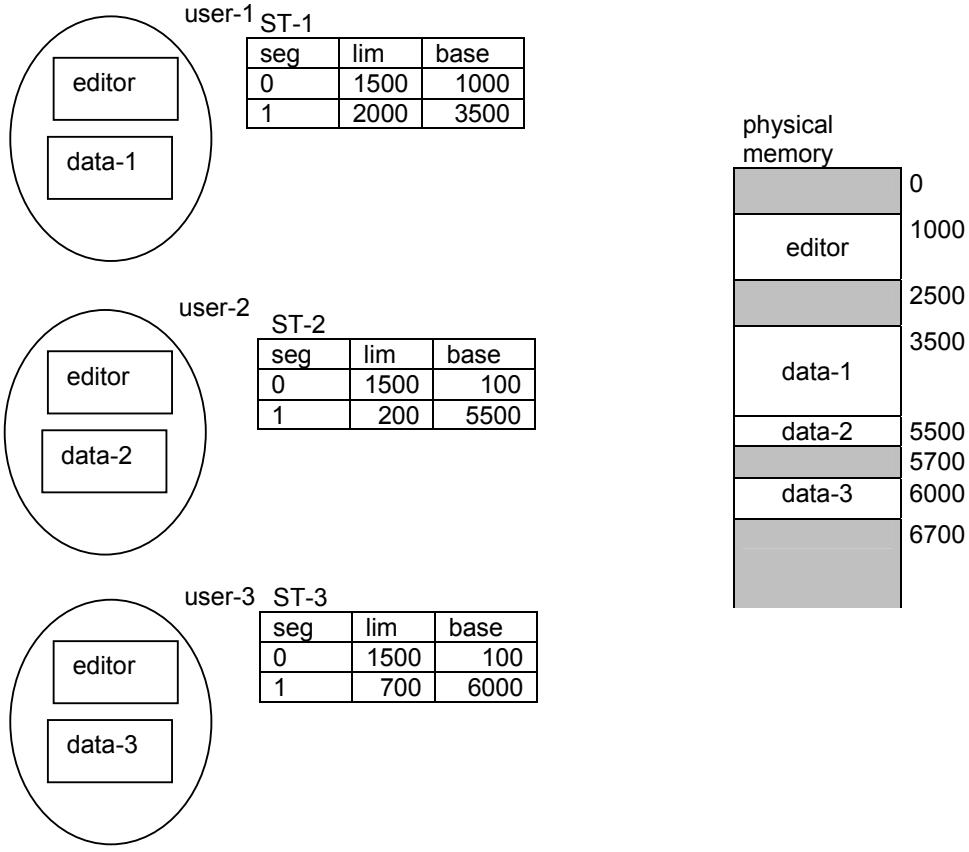

Segment tables are also implemented in the main memory or in associative registers, in the same way it is done for page tables.

## Sharing Segments

Also sharing of segments is applicable as in paging. Shared segments should be read only and should be assigned the same segment number.

**Example 3.6:**

Consider a system in which 3 users executing an editor program which is 1500 KB in size, each having their own data space.



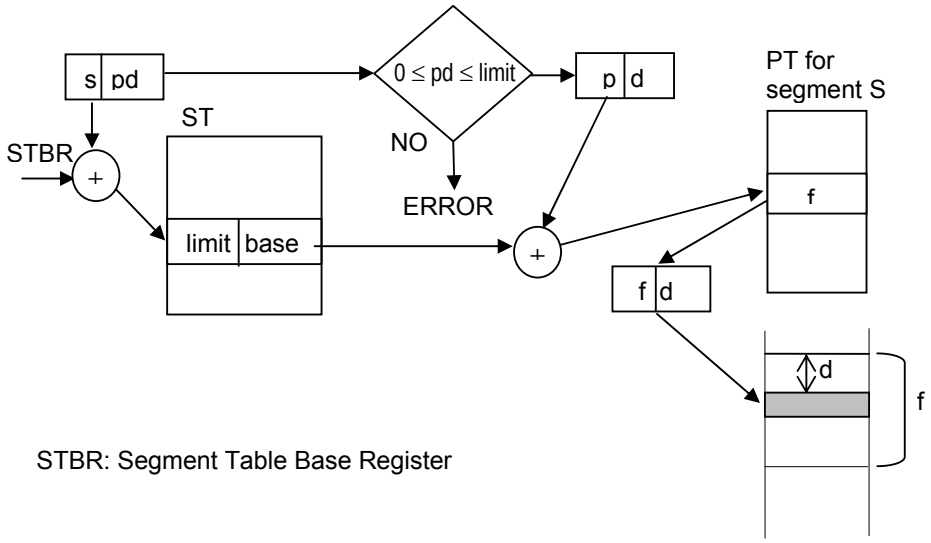
## 3.3. Paged segmentation

The idea is to page the segments and eliminate the external fragmentation problem. In paged segmentation the logical address is made of <s,p,d> triplet. The ST entry for segment S now, contains:

- the length of segment S
- the base address of the PT for segment S.

There is a separate PT for every segment. On the average, now there is half a page of internal fragmentation per segment. However, more table space is needed. In the worst case, again three memory accesses are needed for each memory reference.

The flowchart for accessing a word with logical address <s,p,d> is shown below.




STBR: Segment Table Base Register

## QUESTIONS

**1.** Why do we need memory management and CPU scheduling algorithms for a multiuser system ? Can we do without these algorithms? Explain briefly.

**2. a.** Explain the terms: internal fragmentation and external fragmentation.

**b.** List the memory management methods discussed, and indicate the types of fragmentation caused by each method.

**3.** Consider a multiuser computer system which uses paging. The system has four associative registers. the content of these registers and page table for user_12 are given below:

Page table for user_12

| | |
|---|---|
| 0 | 9 |
| 1 | 6 |
| 2 | 15 |
| 3 | 7 |
| 4 | 42 |

associative registers

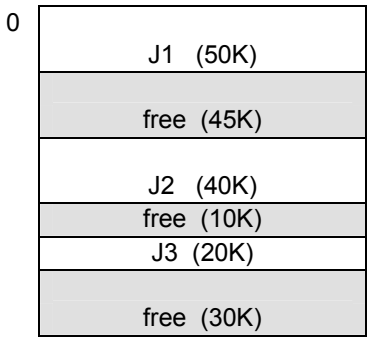
| user # | page # | frame # |
|---|---|---|
| 12 | 3 | 7 |
| 5 | 2 | 18 |
| 12 | 4 | 42 |
| 9 | 0 | 10 |

PTLR[12]:5 PTBR[12]:50000
PAGE SIZE :1024 words

For the following logical addresses generated by user_12's program, calculate the physical addresses, explain how those physical addresses are found, and state the number of physical memory accesses needed to find the corresponding word in memory. If a given logical address is invalid, explain the reason.

**i.** <2,1256> **ii.** <3,290>
**iii.** <4,572> **iv**. <5,290>
**v.** <0,14>

**4.** The following memory map is given for a computer system with variable partitioning memory management.




| Job | Requested memory |
|---|---|
| 1) J4 arrives | 10K |
| 2) J5 arrives | 20K |
| 3) J6 arrives | 15K |
| 4) J7 arrives | 20K |
| 5) J3 leaves | |
| 6) J8 arrives | 50K |

Find and draw the resulting memory maps, after each step of the above job sequence is processed, for :

**a.** first-fit **b.** best-fit **c.** worst-fit

**5.** Consider a computer system which uses paging. Assume that the system also has associative registers.

**a.** Explain how logical addresses are translated to physical addresses.

**b.** Calculate the effective memory access time given:

assoc. register access time = 50 nanosec.
memory access time = 250 nanosec.
hit ratio = 80%

**c.** With the above associative register and memory access times, calculate the minimum hit ratio to give an effective memory access time less than 320 nanoseconds.

**6.** A system which utilizes segmentation gives you the ability to share segments. If the segment numbers are fixed, then a shared segment table is needed, and the ST must be modified. We shall assume that the system is capable of dynamic relocation, but to reduce the overhead, we want to avoid it unless it is absolutely necessary. The following example is given for such a system :

ST-6

| s# | base | size | shares |
|---|---|---|---|
| 0 | - | - | 256 |
| 1 | 0 | 100 | - |
| 2 | 100 | 90 | - |
| 3 | 600 | 15 | - |

ST-9

| s# | base | size | shares |
|---|---|---|---|
| 0 | 190 | 100 | - |
| 1 | - | - | 256 |
| 2 | 290 | 10 | - |

SST

| s# | base | size | no. of sh. |
|---|---|---|---|
| 256 | 400 | 200 | 2 |
| | | | |

Assume maximum number of segments per process is 256, and segments are numbered starting with 0.

**a.** What would be done when a segment previously unshared, becomes a shared segment?

**b.** When do we need dynamic relocation in this system?

**c.** Assume segment-2 of process 6 is being executed. A reference is made to segment-0 of process 6. How is the corresponding physical address going to be found?

**d.** How would self-references within shared segments be handled?

**e.** What is the no. of sharers field in SST used for?

**7.** In the X-2700 computer, logical addresses are 24 bits long. The machine implements paged segmentation with a maximum segment size of 64K words and 4K-word pages:

**a.** Show the logical address structure indicating the segment, page and displacement bits.

**b.** How many segments can a user process contain?

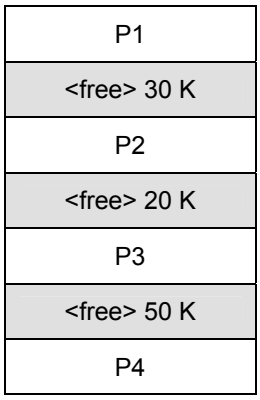
**c.** If a process has to be fully loaded into memory to execute, what is the minimum physical memory capacity?

**d.** If the memory unit contains 65536 frames, show the physical address structure.

**e.** Show the functional block structure of a suitable architecture for implementing paged segmentation in the X-2700. Indicate the sizes of all necessary tables.

**8.** Given the memory map in the figure, where areas not shaded indicate free regions, assume that the following events occur:

| Step | event | required contiguous memory size (K) |
|---|---|---|
| i) | process 5 arrives | 16 |
| ii) | process 6 arrives | 40 |
| iii) | process 7 arrives | 20 |
| iv) | process 8 arrives | 14 |
| v) | process 5 leaves | - |
| vi) | process 9 arrives | 30 |




**a.** Draw the memory maps after step (iv) and (vi) using first fit, best-fit and worst-fit allocation techniques, without compaction

**b.** Draw the same memory maps as in part (**a**) if compaction is performed whenever required. Also show the maps after each compaction.

**9.** In a paging system , the logical address is formed of 20 bits. the most significant 8 bits denote the page number, the least significant 12 bits denote the offset. Memory size is 256K bits.

**a.** What is the page size (in bits)?

**b.** What is the maximum number of pages per process?

**c.** How many frames does this system have?

**d.** Give the structure of the page table of a process executing in this system. Assume 2 bits are reserved for attributes.

**e.** How many bits are required for each page table entry?

**f.** If physical memory is upgraded to 1024 K bits, how will your answer to c and e change?

**10.** Consider a segmentation system with the following data given:

STBR=1000
STLR=5
Associative Registers access time = 50 nsec
Memory Access time = 500 nsec

ST

| s# | base | limit |
|---|---|---|
| 0 | 10000 | 1000 |
| 1 | 12000 | 2000 |
| 2 | 25000 | 4000 |
| 3 | 15000 | 8000 |
| 4 | 38000 | 4000 |

AR

| s# | base | limit |
|---|---|---|
| 0 | 10000 | 1000 |
| 1 | 12000 | 2000 |

Assume data can be written into associative registers in parallel with a memory read or write operation. For replacement of data in associative registers, LRU policy is used.

For each of the following logical addresses, find the corresponding physical address to be accessed, and the total execution time required for that access, including the time spent for address translation operations. Also indicate which memory locations are accessed during address translation. Clearly show all changes made in associative registers.

**a.** <0,150> **b.** <0,3700> **c.** <2,900>
**d.** <2,3780> **e.** <5,200> **f.** <1,200>

**11.**

| P1 |
|---|
| <free> 9K |
| P2 |
| <free> 20 K |
| P3 |
| <free> 14 K |
| P4 |

Consider the memory map given in the figure. If worst fit policy is to be applied, then what will be the memory map after arrival of the processes

P5=3K, P6=5K, P7=7K P8=6K.

Indicate if compaction is needed.

**12.** The following memory map is given for a computer system with variable partitioning memory management.

| P1 9K |
|---|
| <free> 20 K |
| P2 11K |
| <free> 10 K |
| P3 18K |
| <free> 30 K |

| event | required contiguous memory size (K) |
|---|---|
| i) P4 arrives | 16 |
| ii) P5 arrives | 40 |
| iii) P6 arrives | 20 |
| iv) P7 arrives | 14 |

Find and draw the resulting memory maps after the above job sequence is processed completely for

**a.** first fit **b.** best fit **c.** worst fit

indicating whenever a compaction is needed.