MEMORY MANAGEMENT

Syllabus:

Memory management - Different address bindings – compile, link and run time bindings. – Difference between logical address and physical address - Contiguous memory allocation – fixed partition and variable partition – Allocation Strategies - first fit, best fit and worst fit - Define fragmentation – internal and external, and suggest solutions - Paging and paging hardware - Segmentation, and the advantages of segmentation over paging
Concept of virtual memory - Demand paging - Page-faults and how to handle page faults. – Page replacement algorithms: FIFO, optimal, LRU, LRU Approximation, Counting based (LFU and MFU) – Learn the concept of thrashing

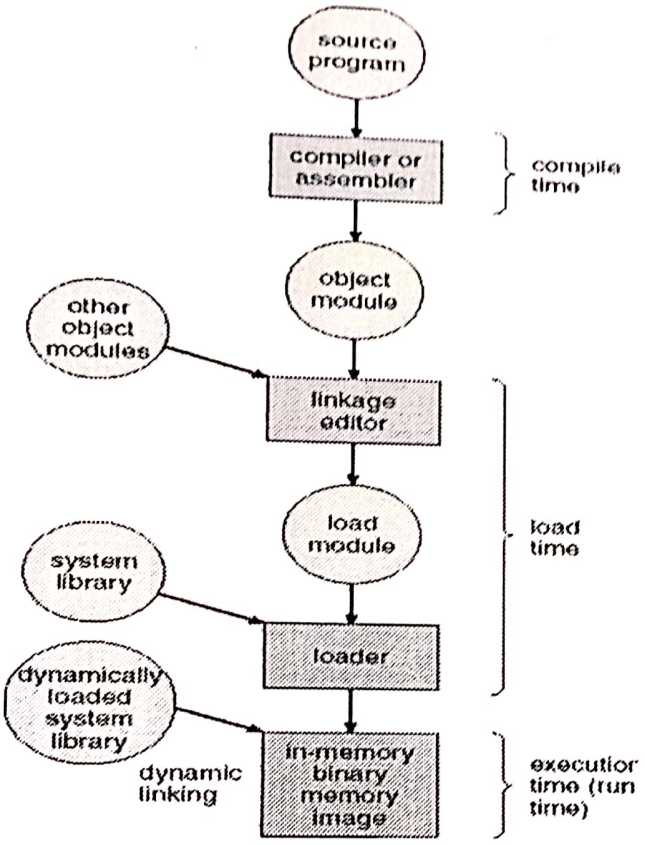
3.1 Address Bindings

- Binding is a mapping from one address space to another.
- A program resides on a disk as a binary executable file
- Program must be brought into memory and placed within a process for it to be executed.
- The process may be moved between disks and memory during execution
- Collection of processes on the disk that are waiting to be brought into memory for execution forms ***Input Queue***
- Normal procedure is to select one of the processes in the input queue and to load that process into memory. As the process is executed, it access instructions and data from memory . When the process terminates its memory space is declared available.

The binding of instructions and data to memory address can be done at any step along the way

- **Compile Time**: If you know at compile time where the process will reside in memory,then **absolute code** can be generated
- **Load Time**:If it is not known at compile time, where the process will reside in memory, then compiler must generate ***relocatable*** **code.** In this case final binding will delayed until load time
- **Execution Time :** If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. Need special hardware support for this.

3.1.1 Multistep Processing of a User Program



3.2 Logical Address and Physical Address

- Despite these problems, LRU replacement with 12 faults is much better than FIFO replacement with 15.

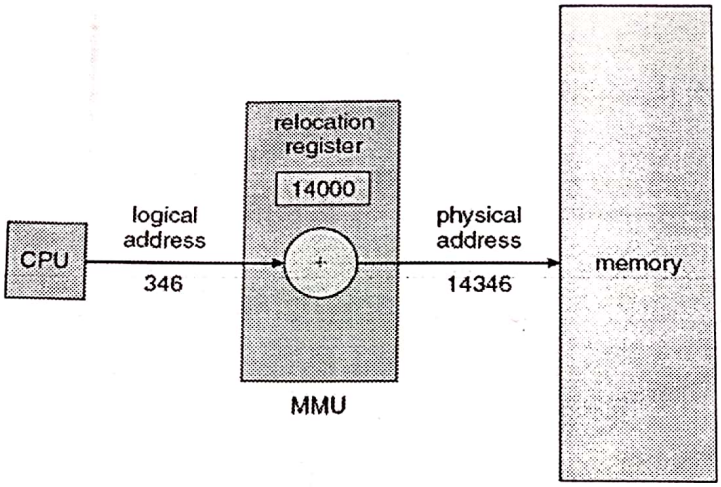
3.12 THRASHING

- ❖ Thrashing is defined as the rate at which page fault occurs during the execution of a program
- ❖ If page fault occurs every time after executing only few instructions, then the system is said to be thrashing
- ❖ Thrashing degrades the system performance

3.12.1 CAUSES OF THRASHING

- suppose that a process enters a new phase in its execution and needs more frames.
- It starts faulting and taking frames away from other processes (global page-replacement algorithm).
- These processes need those pages, however, and so they also fault, taking frames from other processes.
- These faulting processes must use the paging device to swap pages in and out.
- As processes wait for the paging device, CPU utilization decreases.

- The CPU scheduler sees the decreasing CPU utilization and increases the degree of multiprogramming as a result.
  - The new process tries to get started by taking frames from running processes, causing more page faults and a longer queue for the paging device.
  - As a result, CPU utilization drops even further, and the CPU scheduler tries to increase the degree of multiprogramming even more.
- Thrashing has occurred. The page-fault rate increases tremendously. As a result, the effective memory-access time increases.
- No work is getting done, because the processes are spending all their time paging.
- If the degree of multiprogramming is increased even further, thrashing sets in, and CPU utilization drops sharply.
- At this point, to increase CPU utilization and stop thrashing, we must decrease the degree of multiprogramming.

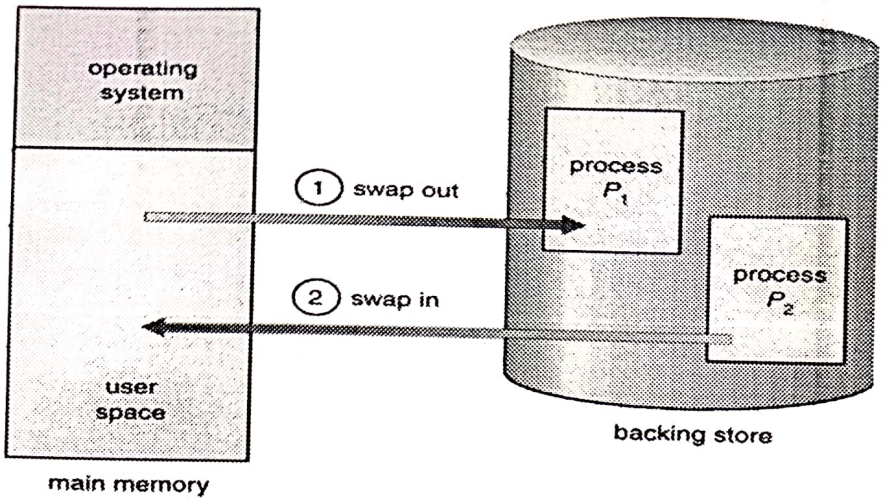


- The set of all logical addresses generated by a program is a ***logical address space***
- Set of all physical address corresponding to these logical addresses, is a ***physical address spaces***
- The run time mapping from virtual to physical address is done by a hardware device called ***Memory Management Unit (MMU)***

3.3 Swapping

- A process can be *swapped* temporarily out of memory to a *backing store*, and then brought back into memory for continued execution.
- Backing store – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images.
- *Roll out, roll in* – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed.
- Major part of swap time is transfer time; total transfer time is directly proportional to the *amount* of memory swapped.
- Modified versions of swapping are found on many systems, i.e., UNIX and Microsoft Windows.

3.3.1 Schematic View of Swapping



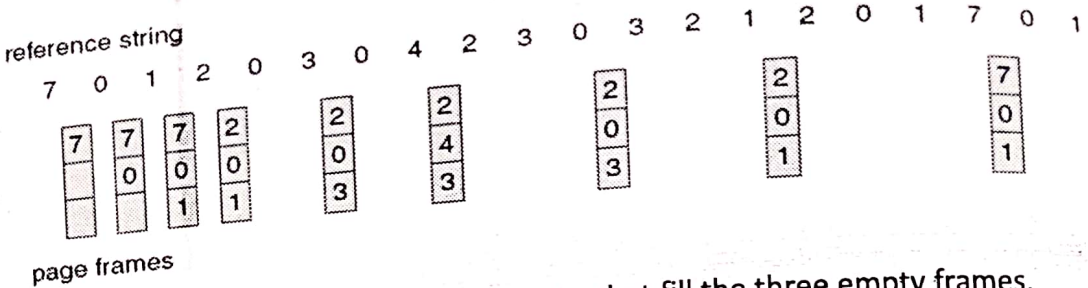
3.4 Contiguous Memory Allocation

- Main memory must accommodate both the operating system and the user process
- Memory is usually divided into two partitions : one for OS and other for user process
- We can place OS in either low memory or in high memory depends upon the location of the interrupt vector
- Since Interrupt vector is in low memory , Usually OS will be in low memory
- We need to consider how to allocate the available memory to the process that are in the input queue waiting to be brought into memory
- In contiguous memory allocation, each process is contained in a single contiguous section of memory
- OS keep a table indicating which part of a memory are available and which are occupied
- Block of available is usually called as ***Hole***
- Different Contiguous allocation methods are ;
  - i) Fixed Partitioning
  - ii) Variable partitioning(Dynamic partitioning)

3.4.1 Fixed Partitioning

- OS occupies some fixed portion of the main memory and rest of the available memory is used for user process
- Simplest scheme for allocating this available memory is to partition it into regions with fixed boundaries
- There are two alternatives for fixed partitioning
  1. Equal Size partition
  2. Un equal size partition

8 MB

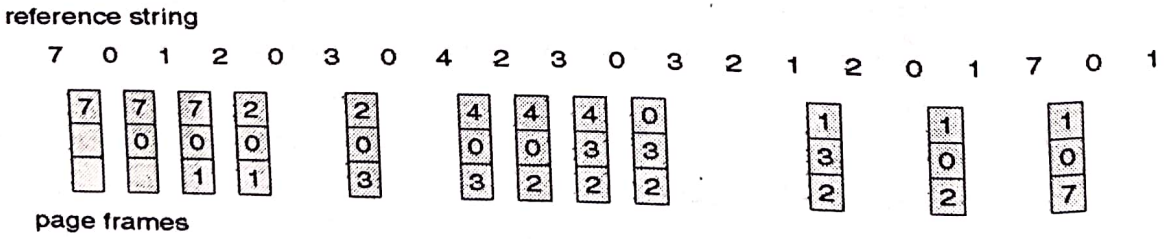


- The first three references cause faults that fill the three empty frames.
- The reference to page 2 replaces page 7, because 7 will not be used until reference 18,whereas page 0 will be used at 5, and page 1 at 14.
- The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again.

- With only nine page faults, optimal replacement is much better than a FIFO algorithm, which resulted in fifteen faults.
- If we ignore the first three, which all algorithms must suffer, then optimal replacement is twice as good as FIFO replacement.
- Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string (similar situation with the SJF CPU-scheduling algorithm).
- As a result, the optimal algorithm is used mainly for comparison studies.

3.11.4 Least Recently Used (LRU) Algorithm

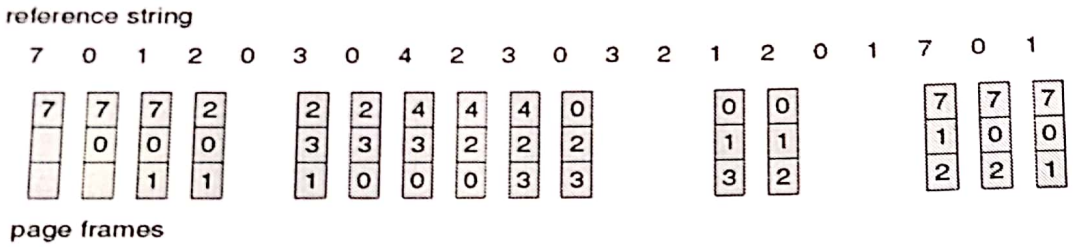
- Use past knowledge rather than future
- Replace page that has not been used in the most amount of time
- Associate time of last use with each page
- LRU algorithm produces 12 faults.
  - Notice that the first 5 faults are the same as those for optimal replacement.
  - When the reference to page 4 occurs, however, LRU replacement sees that, of the three frames in memory, page 2 was used least recently.



  - Thus, the LRU algorithm replaces page 2, not knowing that page 2 is about to be used.
  - When it then faults for page 2, the LRU algorithm replaces page 3, since it is now the least recently used of the three pages in memory.

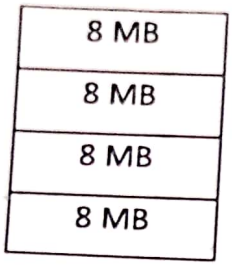
3.11.2 First-In-First-Out (FIFO) Algorithm

- The simplest page-replacement algorithm is a first-in, first-out (FIFO) algorithm.
- A FIFO replacement algorithm associates with each page the time when that page was brought into memory.
- When a page must be replaced, the **oldest page is chosen**.
- For our example reference string, our three frames are initially empty.

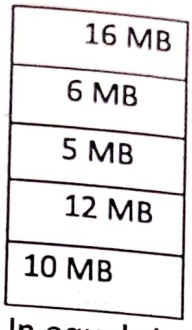


- The first three references (7, 0, 1) cause page faults and are brought into these empty frames.
- The next reference (2) replaces page 7, because page 7 was brought in first.
- Since 0 is the next reference and 0 is already in memory, we have no fault for this reference.
- The first reference to 3 results in replacement of page 0, since it is now first in line.
- Because of this replacement, the next reference, to 0, will fault.
- Page 1 is then replaced by page O. This process continues and there are 15 faults altogether

- The FIFO page-replacement algorithm is easy to understand and program. However, its performance is not always good.



Equal size partition



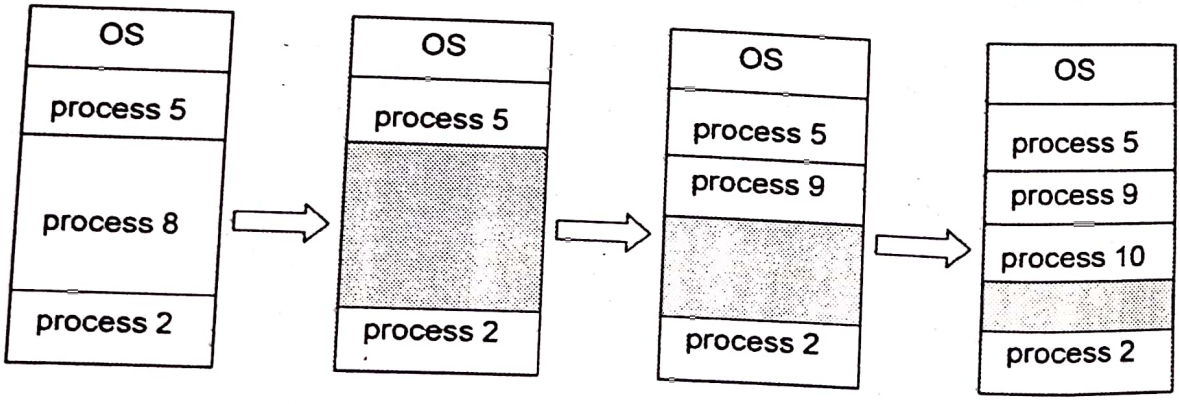
Un equal size partition

In equal size partitions, any process whose size is less than or equal to the partition size can be loaded into the available partition. If all partitions are full and if any process is in the ready or Running state, the OS can swap a process out of any of the partitions and load the new process.

- There are two difficulties with the use of equal size partitions
  - A program may be too big to fit into a partition. In this case the programmer must design the program with the help of using **Overlays**(Keep in memory only those instructions and data that are needed at any given time.)
  - Main memory utilization is extremely inefficient. Any process, no matter how small, occupies an entire partition.

3.4.2 Variable Size Partitioning

- Whenever one of running processes, (p8) is terminated
- Find a ready process whose size is best fit to the hole, (p9)
- Allocate it from the top of hole
- If there is still an available hole, repeat the above (for p10).
- Any size of processes, (up to the physical memory size) can be allocated.



As Example shows, this method will eventually leads to a situation in which there are lot of small holes in memory. As time goes on memory becomes more and more fragmented and utilization declines.

3.5 Allocation Strategies (Dynamic storage Allocation)

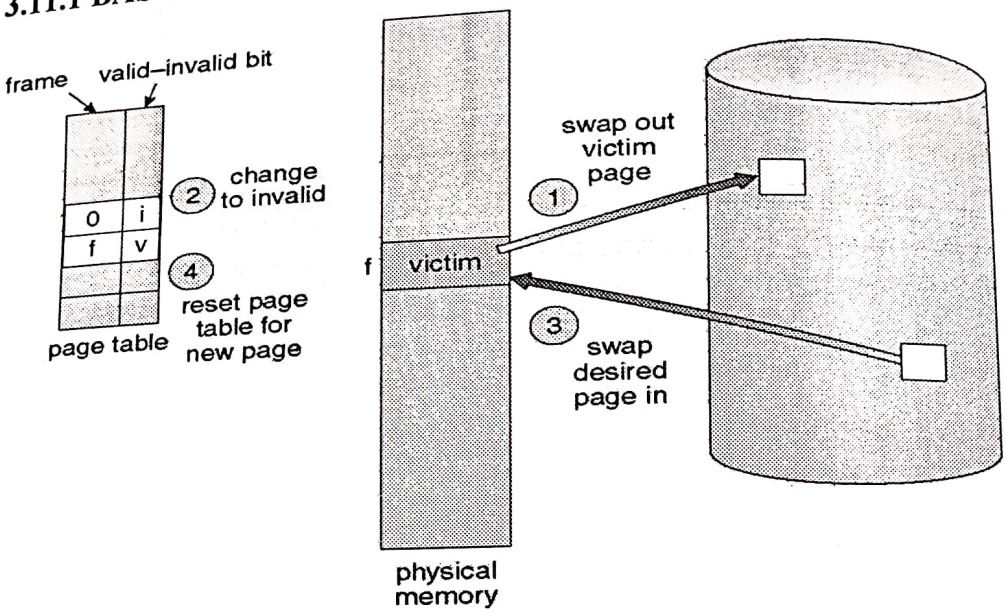
3.5.1 First Fit , Best Fit, and Worst fit Allocation strategies

- In general set of holes, of various sizes is scattered throughout memory at any given time
- When a process arrives and needs memory, the system searches this set for a hole that is large enough for this process. If the hole is too large it will split into two: One part is allocated to the arriving process, the other is returned to the set of holes.
- When a process terminates, it releases its block of memory which is then placed back in the set of holes. If the new hole is adjacent to the other holes, the adjust holes are merged to form one larger hole.
- At this point the system may need to check whether there are processes waiting for memory and whether this newly freed and recombined memory could satisfy the demands of any of these waiting processes
- To satisfy a request of size *n* from a list of free holes we have certain strategies :
    1. **First-fit**: Allocate the *first* hole that is big enough(fastest search).
    2. **Best-fit**: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size. Produces the smallest leftover hole(Best memory usage).
    3. **Worst-fit**: Allocate the *largest* hole; must also search entire list. Produces the largest leftover hole.
- First-fit and best-fit is better than worst-fit in terms of speed and storage utilization.

3.6 Fragmentation

- **External fragmentation** – total memory space exists to satisfy a request, but it is not contiguous.
- **Internal fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.
- The general approach for avoiding this problem is to break the physical memory into fixed-sized blocks and allocate memory in units based on block size.
- One **solution** to the problem of external fragmentation is **compaction**. The goal is to shuffle the memory contents so as to place all free memory together in one large block.
- The simplest compaction algorithm is 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

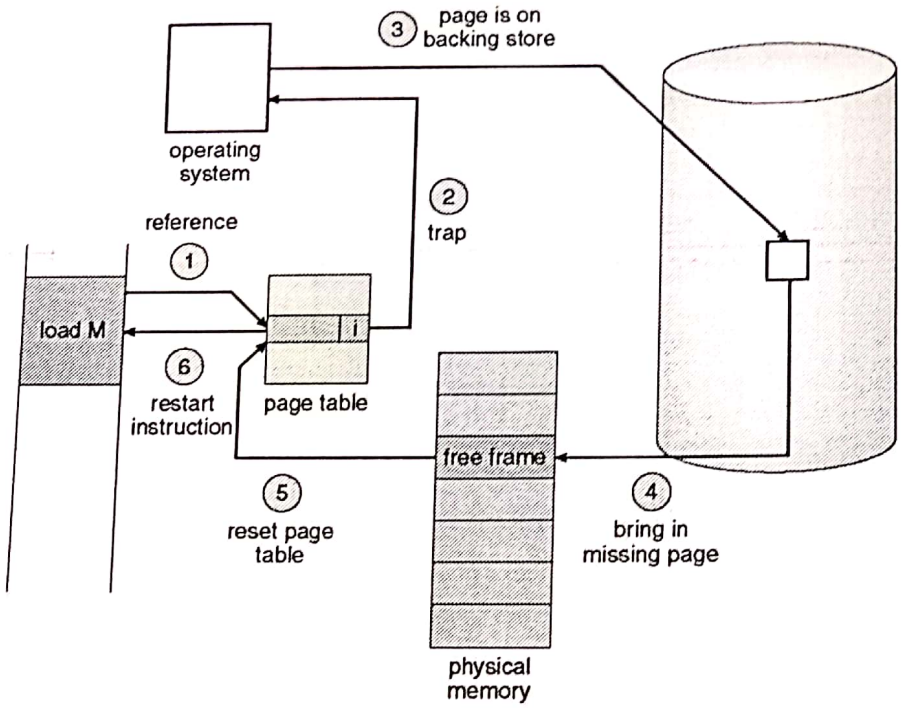
3.11.1 BASIC SCHEME



Page replacement takes the following approach :

1. Find the location of the desired page on the disk
2. Find a free frame
   a. If there is a free frame, use it
   b. If there is no free frame, use a page replacement algorithm to select a ***victim*** frame
   c. Write the victim page on the disk, change the page and frame tables accordingly
3. Read the desired page into the (newly) free frame. Change the page and frame table
4. Restart the user process

- If no frames are free, two page transfers (one out and one in) are required.
- We can reduce this overhead by using a **modify bit** (or **dirty bit**).
- The modify bit for a page is set by the hardware whenever any word or byte in the page is written into, indicating that the page has been modified.
- When we select a page for replacement, we examine its modify bit.
  - If the bit is set, we know that the page has been modified since it was read in from the disk (write that page to the disk).
  - If the modify bit is not set, the page has not been modified since it was read into memory (not write the memory page to the disk: It is already there).
- We evaluate an algorithm by running it on a particular string of memory references and computing the number of page faults.
- The string of memory references is called ***reference string***
- We can generate reference string artificially



2. If the reference was invalid, we terminate the process. If it was valid, but we have not yet brought in that page, we now page it in.
3. We find a free frame.
4. We schedule a disk operation to read the desired page into the newly allocated frame.
5. When the disk read is complete, we modify the internal table kept with the process and the page table.
6. We restart the instruction that was interrupted by the trap.

- At that point, it can execute with no more faults. This scheme is **pure demand paging**: Never bring a page into memory until it is required.
- Programs tend to have **locality of reference** to save the state of interrupted process.

Because we save the state (registers, condition code, instruction counter) of the interrupted process when the page fault occurs, we must be able to restart the process in exactly the same place and state.

- If the page fault occurs on the instruction fetch, we can restart by fetching the instruction again.

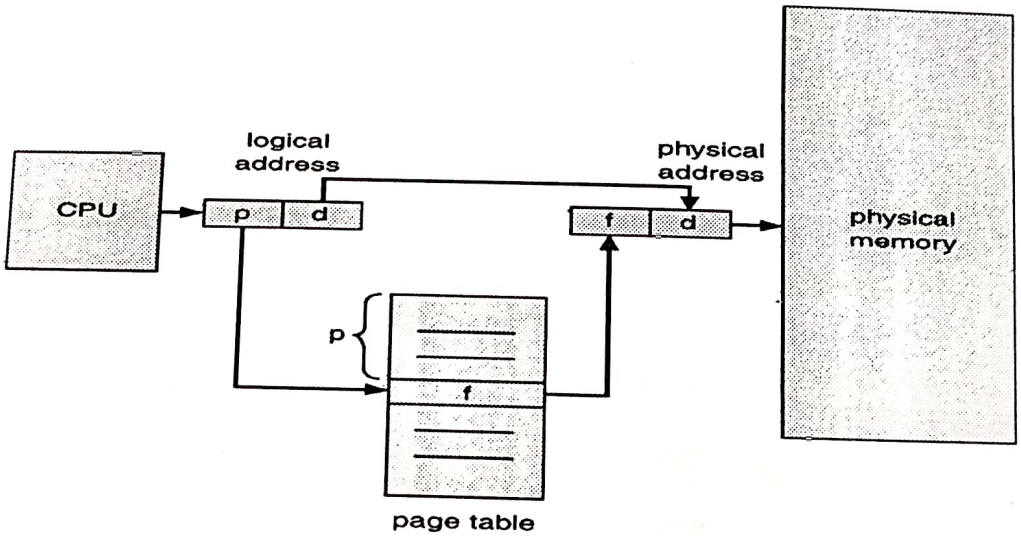
3.11 PAGE REPLACEMENT ALGORITHMS

- Another possible solution to the external-fragmentation problem is to permit the logical address space of the processes to be **non-contiguous,** thus allowing a process to be allocated physical memory wherever the latter is available.
- Two complementary techniques achieve this solution:
  - paging
  - segmentation
- These techniques can also be combined.

3.7 PAGING

- ❖ Paging is a memory management scheme that permits the physical address space of a process to be noncontiguous.
- ❖ Paging can be implemented with the help of hardware

3.7.1 Paging Hardware



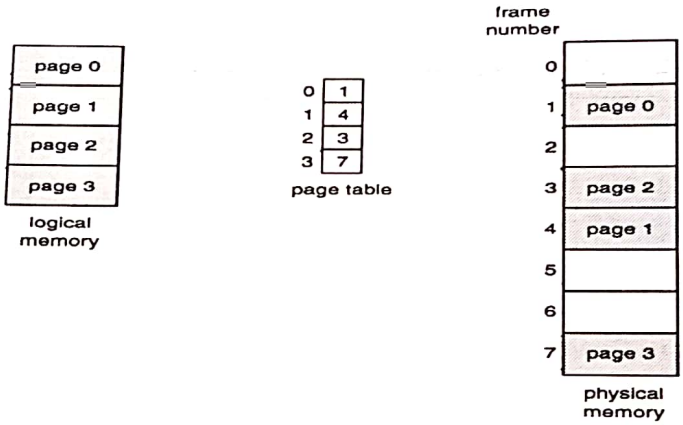
3.7.2 Basic Method

- Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8192 bytes).
- Divide logical memory into blocks of same size called **pages.**
- When a process is to be executed, the pages are loaded into any available memory frames from the backing store.
- Every address generated by CPU is divided into 2 parts :

Page Number(p) : used as an index into a *page table* which contains base address of each page in physical memory.

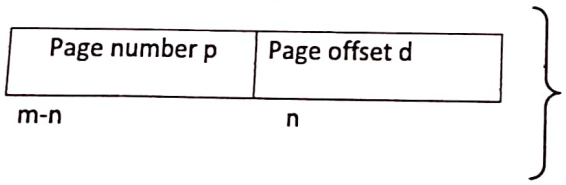
Page offset(d) : Combined with base address to define the physical memory address that is sent to the memory unit.

3.7.3 Paging Model of Logical and Physical Memory



3.7.4 Principle of Operation

- Page size is defined by the hardware
- If the size of the logical address space is 2^m, and page size is 2^n addressing unit, then high- order *m-n* bits of logical address designate the page number, and the *n* low order bits designate the page offset
- It can be represented as follows

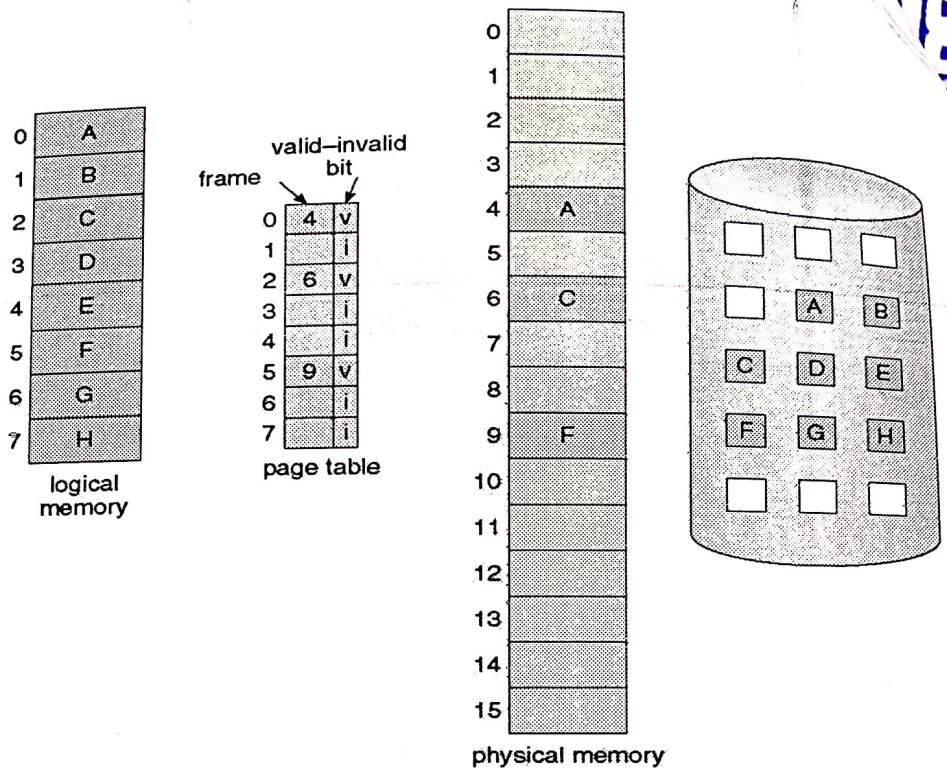


3.7.5 Paging Example

Using a page size of 4 bytes and a physical memory of 32 bytes .
It is shown that how the user's view of memory can be mapped into physical memory.

Physical address = (frame number * size of page) + offset

- Logical address 0 is page O, offset O. Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 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 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 (= (6 x 4) + 0).



- The valid -invalid bit scheme can be used for this purpose.
  - This time however, when this bit is set to ``valid'', the associated page is both legal and in memory.
  - If the bit is set to ``invalid'', the page either is not valid (that is, not in the logical address space of the process) or is valid but is currently on the disk.

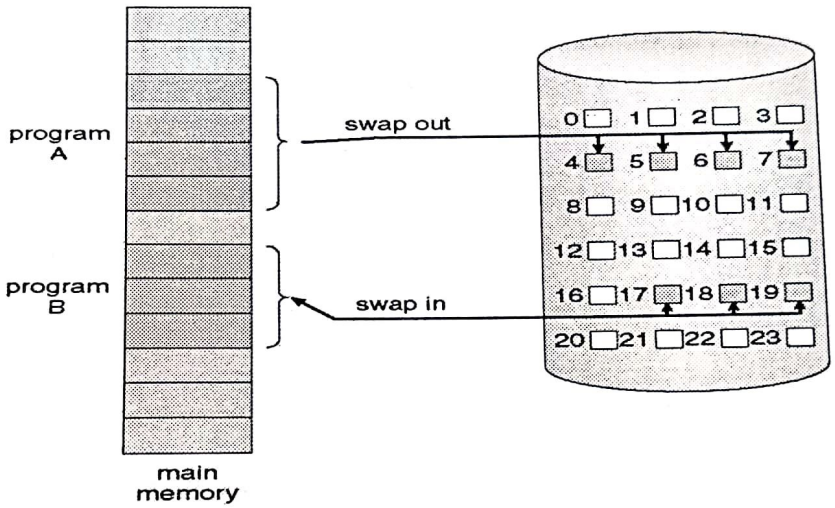
3.10.3 Page table when some pages are not in main memory

- While the process executes and accesses pages that are memory resident, execution proceeds normally.
- if the process tries to access a page that was not brought into memory, Access to a page marked invalid causes a **page-fault trap**.
- The paging hardware, in translating the address through the page table, will notice that the invalid bit is set, causing a trap to the OS. The procedure for handling this page fault is straightforward.

3.10.4 Steps in handling Page Fault

1. We check an internal table (in PCB) for this process to determine whether the reference was a valid or an invalid memory access.

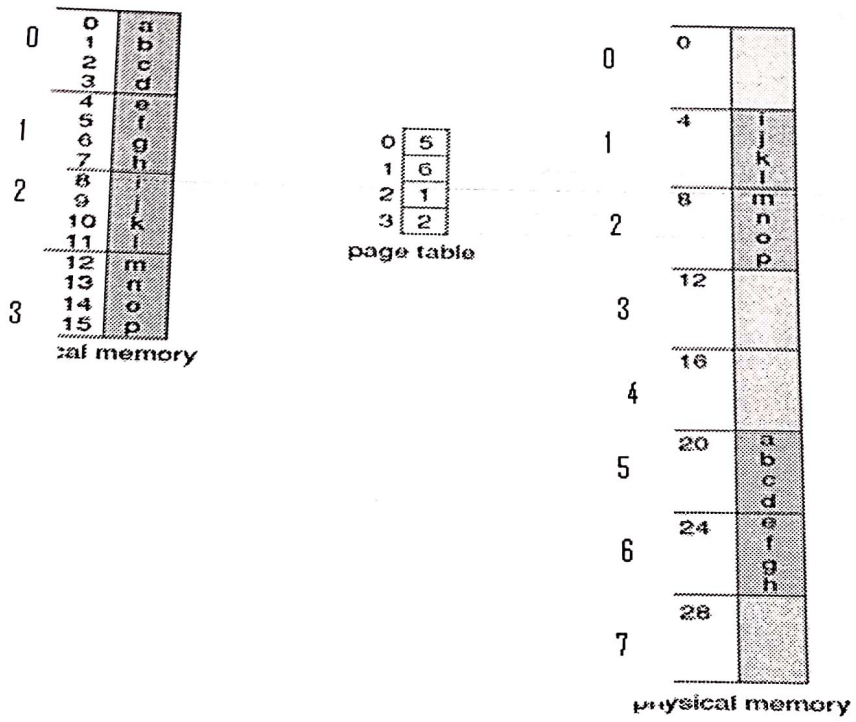
- A swapper manipulates entire processes, whereas a pager is concerned with the individual pages of a process.
- We thus use **pager**, rather than swapper, in connection with demand paging.



3.10.2 BASIC CONCEPTS

- When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again.
- It avoids reading into memory pages that will not be used anyway, decreasing the swap time and the amount of physical memory needed.
- Some form of hardware support is needed to distinguish between the pages that are in memory and the pages that are on the disk.

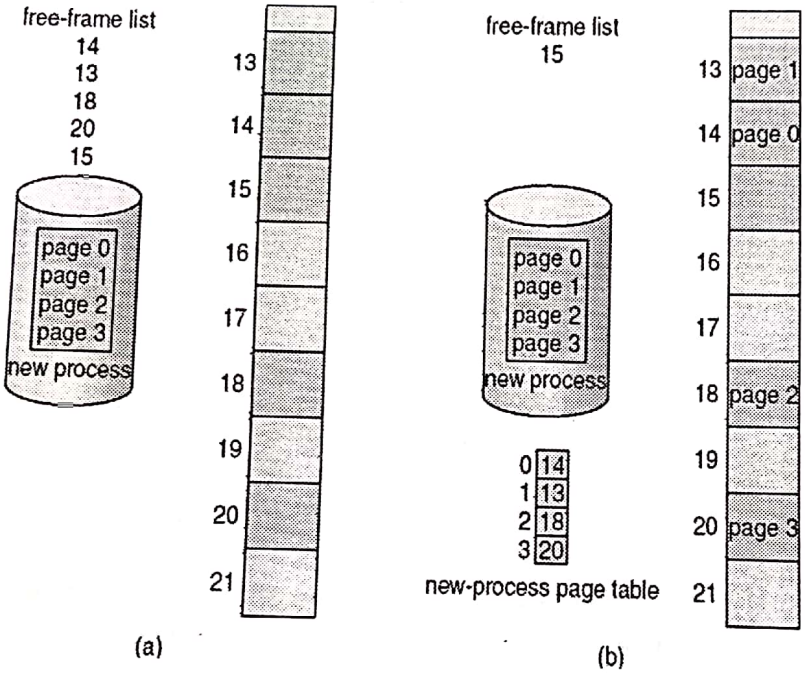
- Logical address 13 maps to physical address 9.



32-byte memory and 4-byte pages

- Every logical address is bound by the paging hardware to some physical address.
- Any free frame can be allocated to a process that needs it.
- Its size, expressed in pages, is examined. Each page of the process needs one frame.
- When a process arrives in the system to be executed,
  - Its size, expressed in pages, is examined. Each page of the process needs one frame.
  - Thus, if the process requires pages, at least frames must be available in memory. If frames are available, they are allocated to this arriving process.
  - The first page of the process is loaded into one of the allocated frames, and the frame number is put in the page table for this process.
  - The next page is loaded into another frame, and its frame number is put into the page table, and so on
- An important aspect of paging is the clear separation between the user's view of memory and the actual physical memory.
- The user program views memory as one single space, containing only this one program. In fact, the user program is scattered throughout physical memory, which also holds other programs.

- The logical addresses are translated into physical addresses by the address-translation hardware. This mapping is hidden from the user and is controlled by the OS.
- Since the OS is managing physical memory, it must be aware of the allocation details of physical memory-which frames are allocated, which frames are available, how many total frames there are, and so on.



- This information is generally kept in a data structure called a **frame table.** The frame table has one entry for each physical page frame, indicating whether the latter is free or allocated and, if it is allocated, to which page of which process or processes.

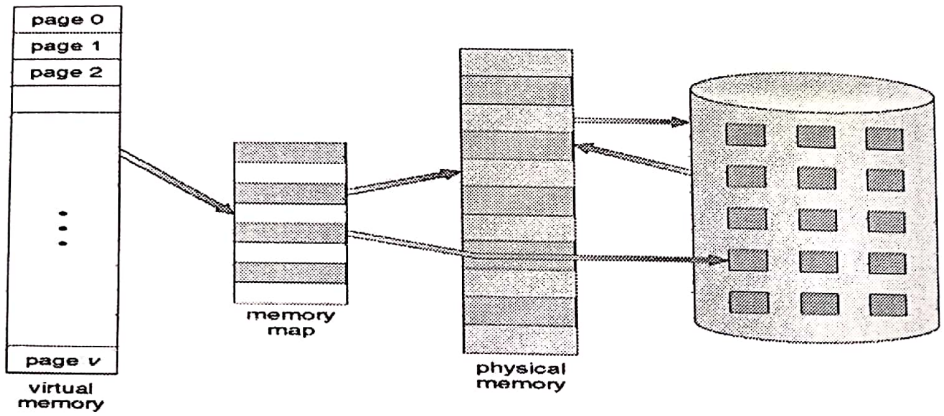
3.7.6 Implementation of Page Table

- Page table is kept in main memory
- **Page-table base register (PTBR)** points to the page table
- **Page-table length register** indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers (TLBs)**
- **TLB** is a special,small, fast, lookup ,hardware, cache associative high speed memory.
- Each Entry in TLB consist of two parts: a Key(tag) and value

3.7.6.1 Paging hardware with TLB

- One advantage of this scheme is that programs can be larger than physical memory
- Virtual memory abstracts main memory i9nto an extremely large, uniform array of storage separating logical memory as viewed by the user from physical memory
- This will frees the programmer from concerns of memory storage limitations
- Virtual memory can be implemented via:
    - Demand paging
    - Demand segmentation

3.9.1 Virtual Memory That is Larger Than Physical Memory

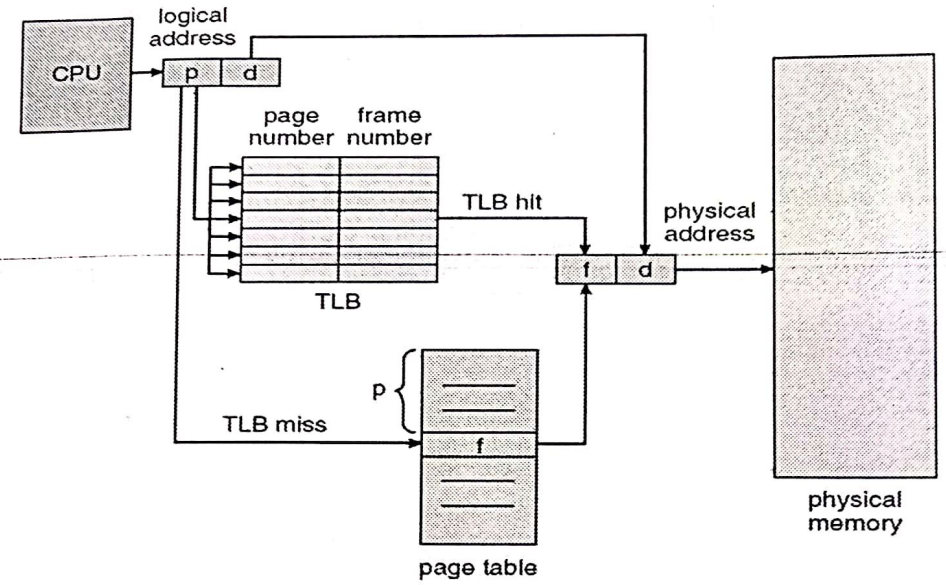


3.10 DEMAND PAGING

- Consider how an executable program might be loaded from disk into memory.
    - One option is to load the entire program in physical memory at program execution time. However, a problem with this approach is that we may not initially need the entire program in memory.
    - An alternative strategy is to initially load pages only as they are needed. This technique is known as **demand paging** and is commonly used in virtual memory systems.
- A demand-paging system is similar to a paging system with swapping , where processes reside in secondary memory (usually a disk).

3.10.1 Transfer of a Paged Memory to Contiguous Disk Space

- When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory, however, we use a **lazy swapper**. A lazy swapper never swaps a page into memory unless that page will be needed.



- TLB works in following way :
  - TLB contains only a few of the page table entries
  - When a logical address is generated by the CPU, its page number is presented to the TLB
  - IF the page number is found(Known as TLB **Hit)**, its frame number is immediately available and is used to access memory
  - If the page number is not in TLB(known as TLB **Miss**), a memory reference to the page table is made.When the frame number is obtained, we can use it to access memory
  - In addition, we add the page number and frame number to the TLB, so that it will easy for next reference
  - If the TLB is full, the contents will be removed using some replacement algorithms
  - Some TLB's allow entries to be **wired down**(cannot be removed from TLB)

- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address-space protection for that process
- When a new page table is selected TLB must be **Flushed**
- The Percentage of times that a particular page number is found in the TLB is called the **Hit Ratio**

3.7.7 Advantages of Paging

Paging solves fragmentation without physical moving the partition. So it is possible to accommodate more jobs

- The *degree of multi programming* increases
- Increases memory and processor utilization

3.7.8 Demerits of Paging

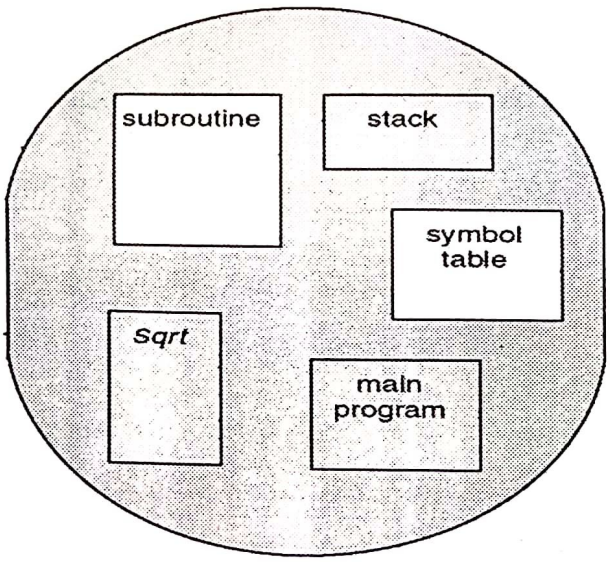
- Page map hardware increases the cost of computer and slows down the speed
- Some memory area will be still unused page break
- Processor time will be wasting in maintaining and updating page table

3.8 SEGMENTATION

- Memory-management scheme that supports user view of memory.
- A program is a collection of segments
- A segment is a logical unit such as:
  - main program — local variables & global variables
  - procedure — Common Block
  - function — Stack
  - method — Symbol table
  - object — Array

3.8.1 User's View of a Program

- A logical address space is a collection of segments.
- Each segment has a name and a length. The addresses specify both the segment name and the offset within the segment.



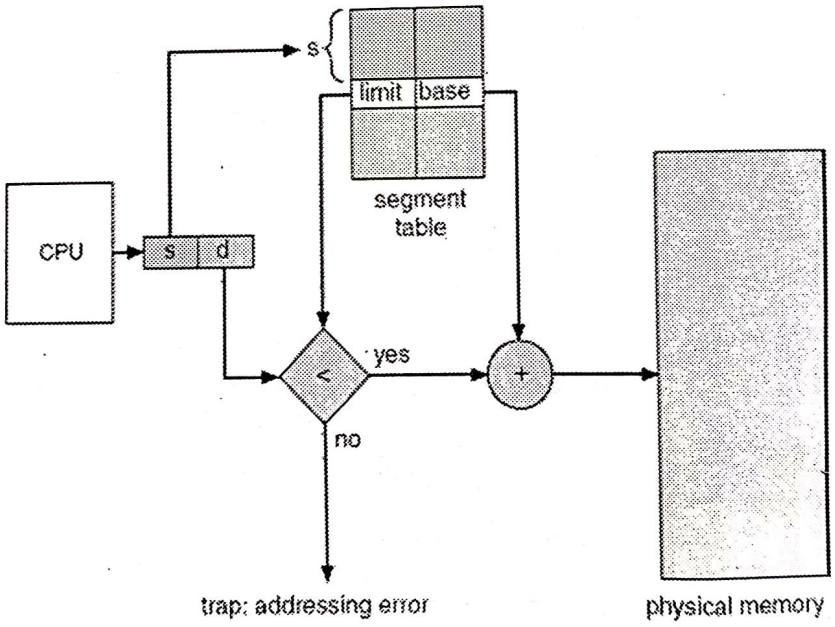
logical address

- The user therefore specifies each address by two quantities:
  1. a segment name
  2. an offset

- Thus, a logical address consists of a two tuple:
  <segment-number, offset>

3.8.2 HARDWARE

- **Segment table** – maps two-dimensional physical addresses; each table entry has:
  - **Segment base** – contains the starting physical address where the segments reside in memory
  - **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 to 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 OS (logical addressing attempt beyond end of segment).



- When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte. The segment table is thus essentially an array of base-limit register pairs

3.8.3 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).