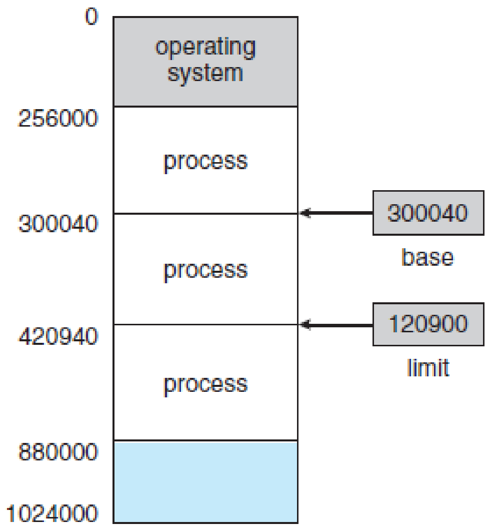
Memory Management

Logical address – is reference to memory location independent of the current assignment of data to memory; a translation must be made to physical address before the memory access can be achieved. It is generated by the CPU; also referred to as ***virtual address.***

Physical address or Absolute Address is actual memory location in memory. It is the address seen by the memory unit (HW).

Logical and physical addresses are the **same in compile-time** and load-time address-binding schemes; logical (virtual) and physical addresses **differ in execution-time** address-binding scheme.

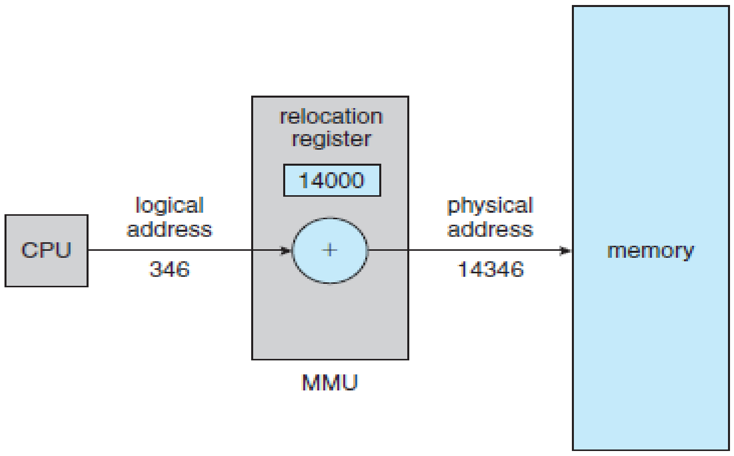
A relative address is a particular example of logical address, in which the address is expressed as a location relative to some known point, usually a value in a processor register.



A base and a limit register define a logical address space.

1. Memory-Management Unit (MMU)

Hardware device **maps** virtual address to physical address. In MMU scheme, the value in the **relocation register** is added to every address generated by a user process at the time it is referenced. The user program deals with ***logical*** addresses; it never sees the ***real*** physical addresses.



Dynamic relocation using a relocation register.

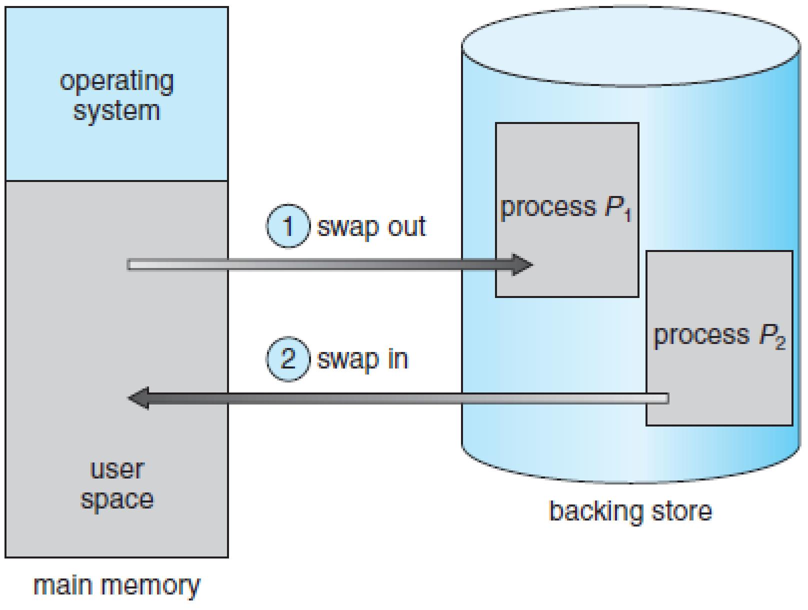
Fig 1:Dynamic relocation using relocation register

2. Swapping

A process must be in memory to be executed. A process can be ***swapped*** temporarily out of memory to a ***backing store***, and then brought back into memory for continued execution. Sometimes called **"swap space," "swap partition," "swap disk".**

For example, assume a multiprogramming environment with a round-robin CPU-Scheduling algorithm. When a time slice expires, the memory manager will start to swap out the process that just finished and to swap another process into memory space that has been freed. In the meantime, CPU scheduler will allocate the time slice to another process in the main memory.

If a higher-priority process arrives and wants service, the memory manager can swap-out lower priority process and then load and execute the higher-priority process. When the higher-priority process finishes, the lower priority process can be swapped back in and continued. This variant of swapping is sometimes called **roll out**, **roll in**.



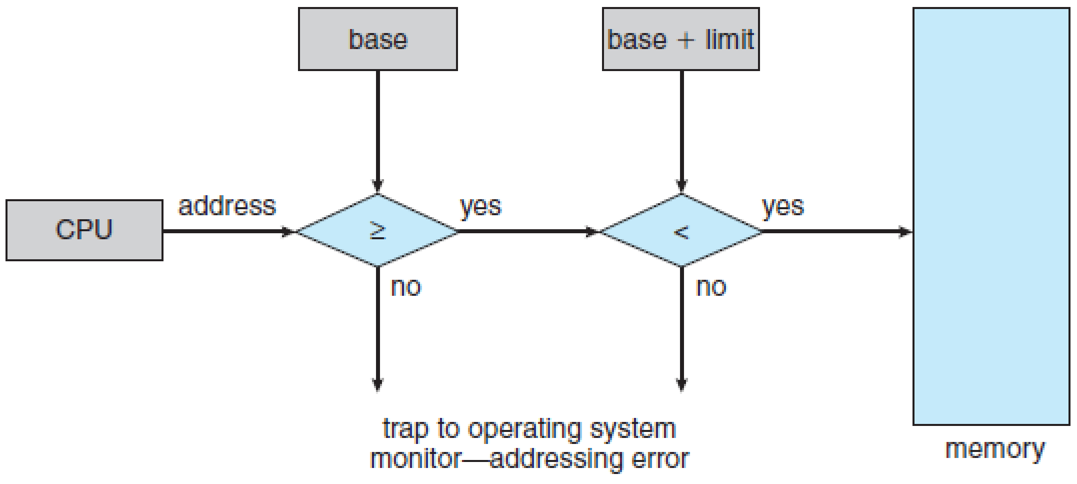
Swapping of two processes using a disk as a backing store.

Fig 2: Schematic View in Swapping

Swapping requires a backing store. The backing store is commonly a fast disk. A job can be swapped out and still be on the Ready Queue. So now, we can have more jobs on Ready Queue than will fit in main memory. If a dispatcher selects a process which is not in main memory, that process must be brought into memory. If binding is done at assembly or load time, then the process cannot be loaded to a different memory location. If execution-time binding is being used then the process can be loaded anywhere in the memory.

3. Contiguous Memory Allocation:

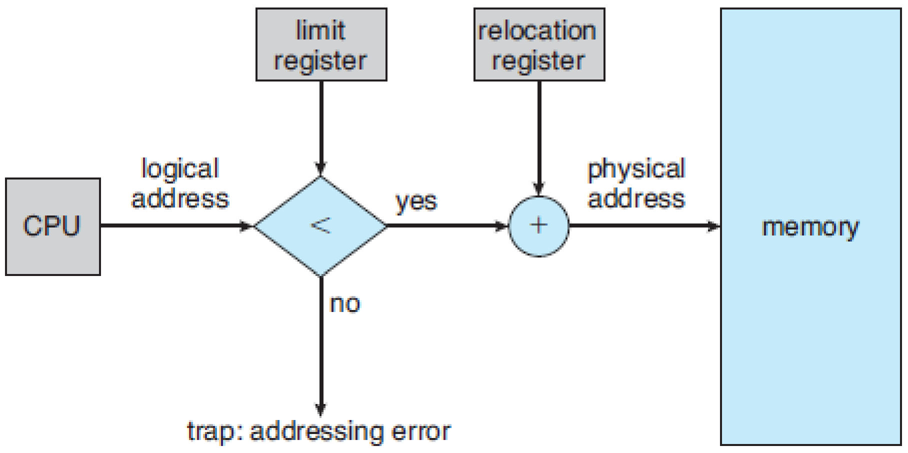
The main memory is usually divided into two partitions: one for the resident operating system and one for the user process. The operating system can be either in the low memory or high memory. We want usually want several user processes to reside in memory at the same time. In **contiguous memory allocation**, each process is contained in a single contiguous section of memory.



Hardware address protection with base and limit registers.

Fig 3: Check base first, then validate that within limit

Memory Mapping and Protection: The relocation register contains the value of the smallest physical address; the limit register contains the range of logical addresses (for example, each relocation= 100040 and limit = 74600). With relocation and limit registers, each logical address must be less than the limit register; the MMU maps the logical address dynamically by adding the value in relocation register. The mapped address is sent to memory. When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers as a part of context switching.



Hardware support for relocation and limit registers.

Fig 4: Validate limit first, then add relocation value

Memory Allocation:

Fixed sized partitions (Static Partitions): One of the simplest methods for allocating memory is divide memory into several **fixed**-sized **Partitions.** Each partition may contain exactly one process. Thus, the degree of multiprogramming is bound by number of partitions. In this, **multi partition method**. When a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process. This method is used in batch processing operating system. In the fixed-partition scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied.

A. ***Equal-size partitions:*** Main memory is divided into a number of static partitions at a system generation time. Any process whose size is less than or equal to the partition size can be loaded into an available partition. The operating system can swap a process out of a partition, if none are in a ready or running state. The maximum size of program that can be executed is 8M.

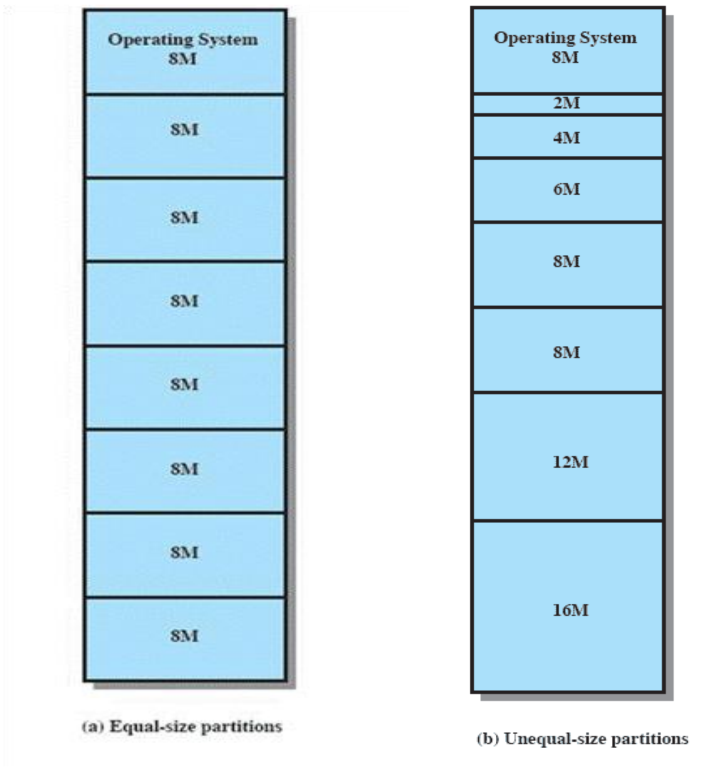


Fig 5: Equal Size partitions **Fig 6: Unequal Size partitions**

B. ***Unequal Size Partitions:*** The maximum size of program that can be loaded into memory is 16M. Smaller programs can be placed in smaller partitions, reducing internal fragmentation.

Placement Algorithm: If process size is equal to partition size process is allocated to same partition. If process size is not equal to partition size, the process is assigned to the smallest partition within which it will fit.

Fixed Partitioning Problems: If a program does not fit in a partition it cannot be executed. Any program, no matter how small, occupies an entire partition. This results in ***internal fragmentation***. The number of active processes is limited by the system. A large number of very small processes will not use the space efficiently.

Fixed Partitioning Strengths: It is very simple to implement; little operating system overhead.

Dynamic Storage Allocation Method:

Initially, all user memory is assumed as a single partition. With dynamic partitioning, the partitions are of variable length and number. Initially, 64MB main memory is empty, except for the operating system 8(a). Assume that the size of four processes are P1-20M,P2-14M,P318M,P4-8M. The first three processes are loaded in as shown if fig 8(b,c,d). This leaves a “hole” at the end of memory that is too small for the fourth process. At some point, none of processes in memory is ready. The OS swaps out process 2 (fig 8 e), which is sufficient to load new process, process 4 (fig 8 f). Because the process 4 is smaller than process, another small hole is created. Later, a point is reached at which none of processes in main memory is ready, but process 2, in ready-suspended state, is available. Because there is insufficient room in memory for process 2, the OS swaps process 1 out (fig 8 g) and swaps process 2 back in (fig 8 h). As time goes on, memory becomes more and more fragmented, and memory utilization declines. This Phenomenon referred as **External Fragmentation**.

Compaction: One technique for overcoming external fragmentation is **COMPACTATION**. From time to time, the operating system shifts the processes so that they are contiguous and so that all of the free memory is together in one block. For example, in fig (8.h), compaction will result in a block of free memory of length 16M. The difficulty with compaction is that it is time consuming procedure and wasteful of processor time. The compaction implies the need for a dynamic relocation capability.

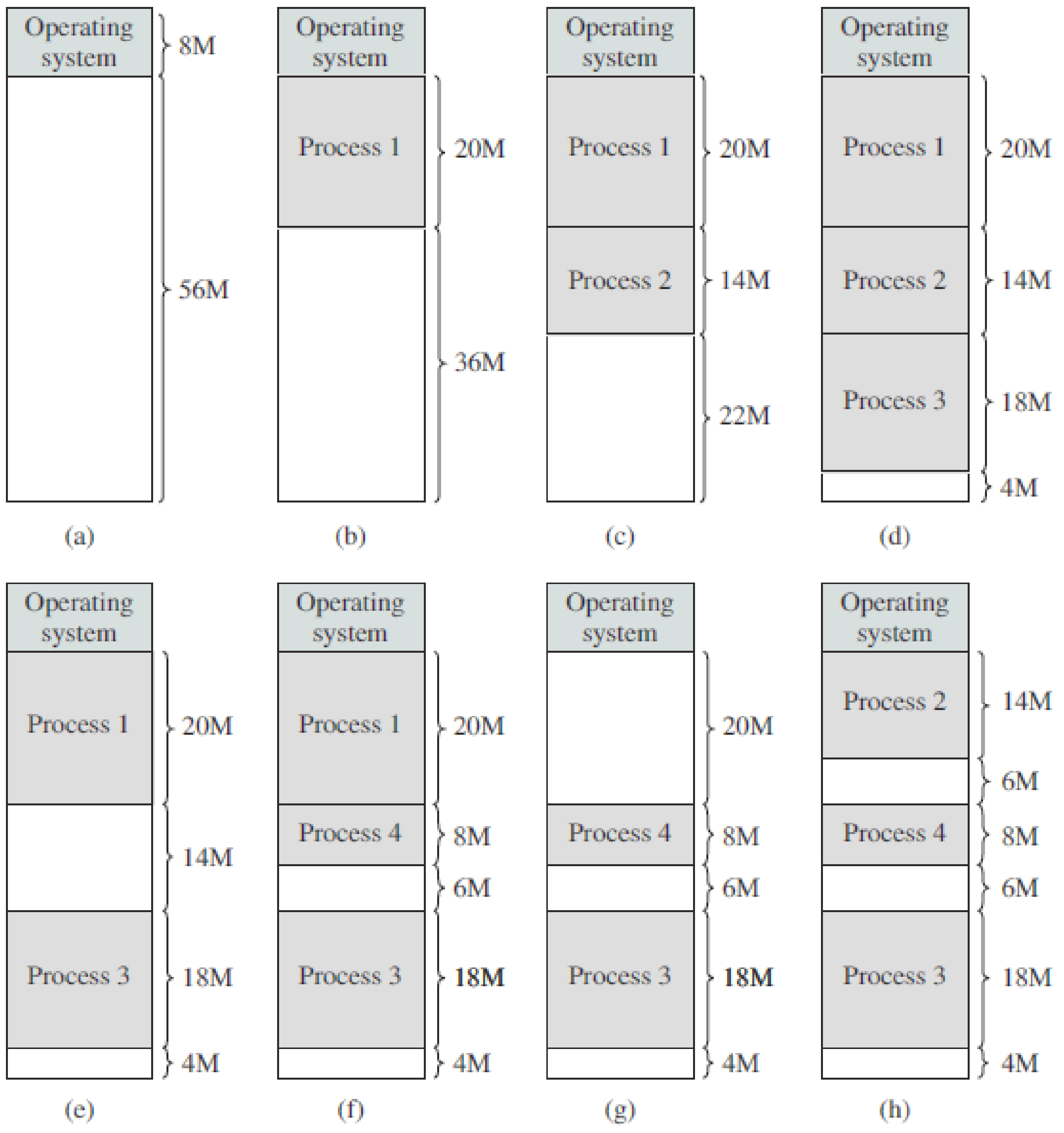


Fig 8: The effect of Dynamic Partitioning

In general, at any given time we have a set of holes of various sizes scattered throughout memory. When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process. If the hole is too large, it is split into two parts. One part is allocated to the arriving process; the other is returned to set of holes. If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole.

At any given time, we have a list of available block sizes and the input queue. The operating system can order the input queue according to scheduling algorithm. Three placement algorithms that might be considered are best-fit, first-fit and next-fit.

First-fit: allocates 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. The search stops as soon as it finds a free hole that is large enough.
Best fit: Allocate a smallest hole that is big enough. This strategy produces the smallest leftover hole.
Worst fit: Allocate the largest hole. This strategy produces the largest leftover hole, which may be more useful **than the smaller leftover hole from a best-fit approach.**

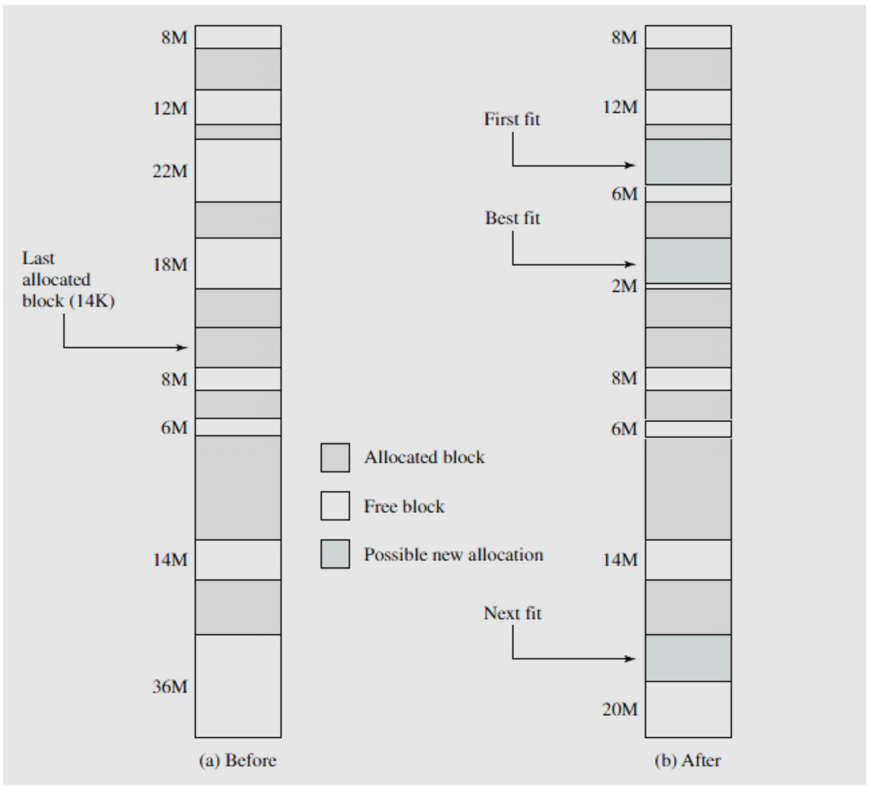


Fig 9: Example Memory configuration before and after Allocation of 16M Byte block

Paging:

Paging is a memory-management scheme that permits the physical address of a process to be noncontiguous. Divide logical memory (programs) into blocks of same size called pages. Divide physical memory into equal sized partitions called frames (size is power of 2, between ½K and 8K). The size of the page and frame must be equal.
When a process is to be executed, its pages are loaded into any available memory frames from the disk. Page table is used to map the page number to corresponding frame number. The page table contains the base address of the page (frame) in memory. Page number will be used as index for the page table. Whenever program is loaded into memory the corresponding frame number is updated to page table. In the fig 11, page 0 is loaded into frame 1, page 1 is loaded into frame 4, page 2 is loaded into frame 3 and page 3 is loaded into frame 7. The frame numbers are updates to page table. If a frame is allocated to one process it cannot be allocated to another process.

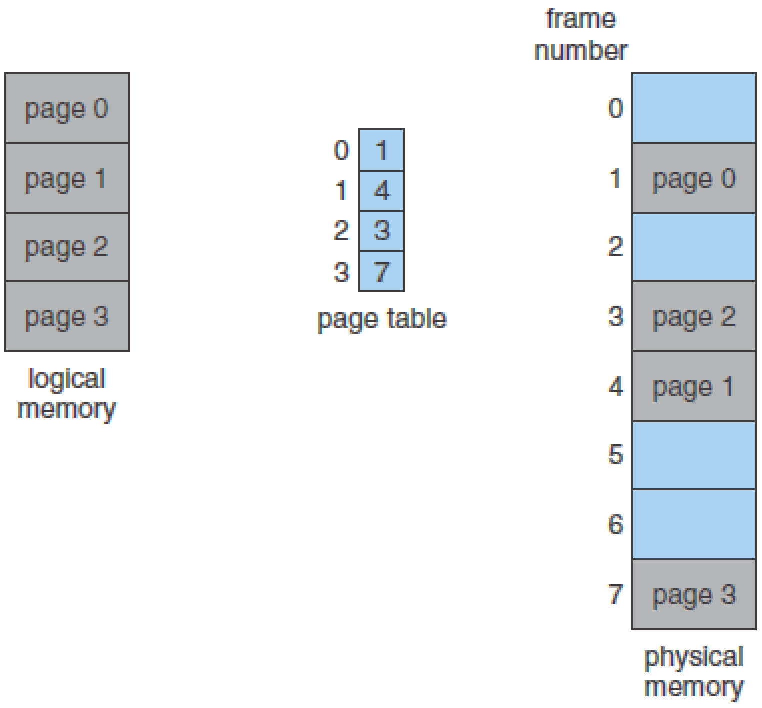


Figure 11 Paging model of logical and physical memory.

CPU generates a logical address; 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 index for the page table. The page number is replaced with corresponding frame number as shown in fig 10. The MMU(memory management unit) converts logical address into physical address. The base address + offset is sent to MMU which is mapped to the corresponding physical page.

Assume that, the size of process P1 is 10KB and frame size of memory is 4 KB. Now, process P1 requires 3 frames. The process P1 occupies 2KB of third frame. The remaining 2KB cannot be allocated to any other process. This *hole* is called internal fragmentation.

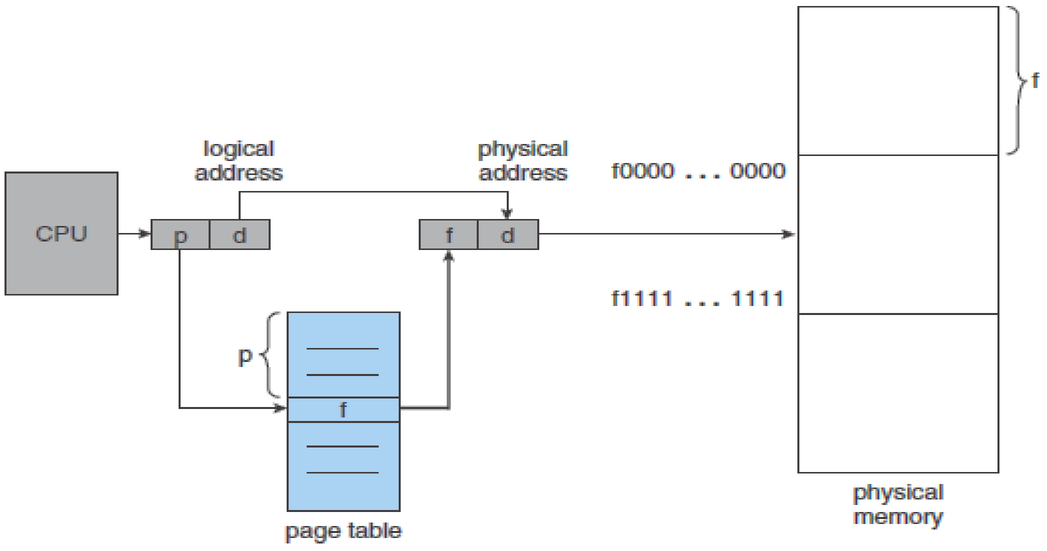


Figure 10 Paging hardware.

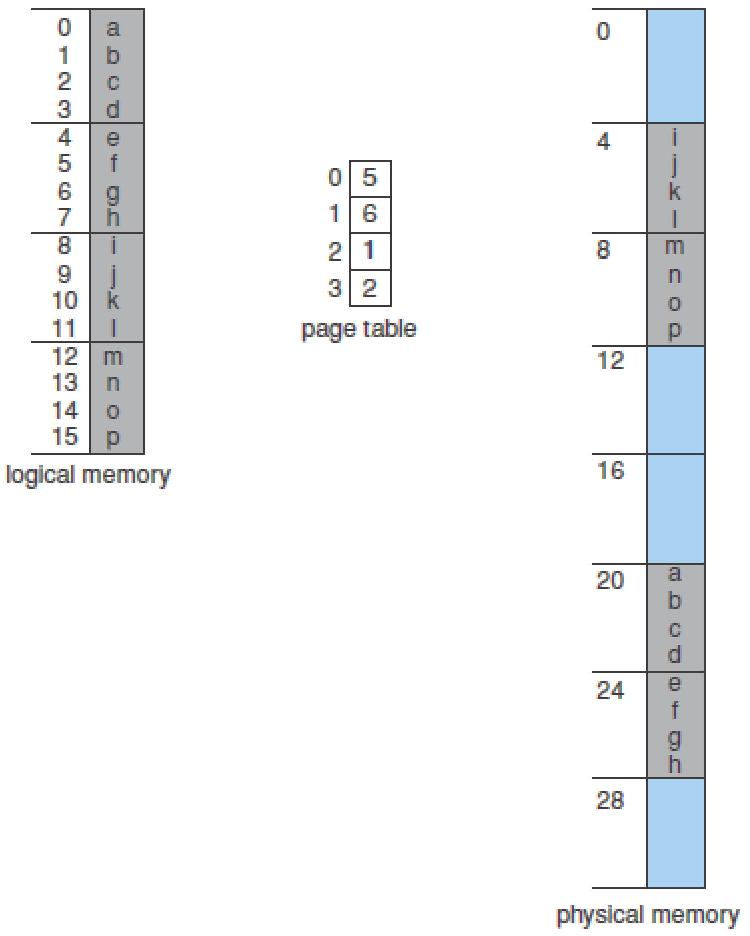


Figure 12 Paging example for a 32-byte memory with 4-byte pages.

In fig 12, the physical memory size is 28 bytes. So, it requires five bits to represent each bit(2^5). The size of page is 4 bytes. So, it requires 2 bits to represent 4 bytes (2^2). So, d= 2 bits. 2^5 - 2^2 = 3 bits required page number.

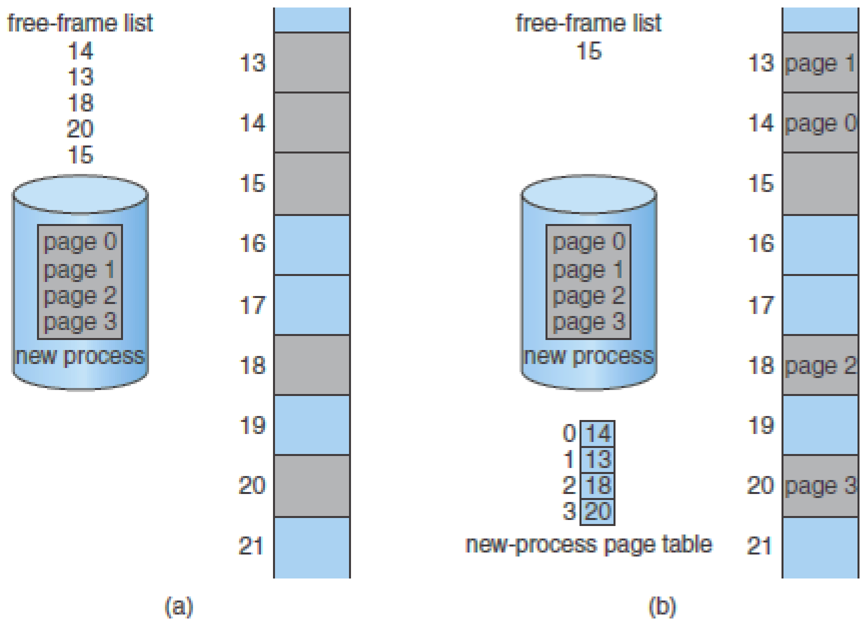


Figure 13 Free frames (a) before allocation and (b) after allocation.

When a process arrives in the system to be executed, its size is expressed in terms of pages. Each page of the process needs one frame. Thus, if the size of the process is 'n' pages, 'n' frames must be allocated to execute the program. As shown in fig 12, page 0 is loaded into frame 14 and it is updated to page table. Before loading the program there are five free frames, after loading the process of 4 pages there is only one free frame.

Hardware Support:

The hard ware implementation of page table can be done in several ways. In the simplest case, the page table is implemented as a set of dedicated **registers**. These registers should be built with very high speed logic. The use of registers for page table is satisfactory if the page table is reasonably small(256 entries). Most contemporary computers, however, allow the page table to be very large(1 million entries). The use of fast registers for such a large page table is not feasible. Rather, page table is kept in main memory, and a **page-table base register** (PTBR) points to the page table.

The problem with this approach is the time required to access a user memory location. In this scheme two memory references are required to access a byte. Thus, memory access is slowed by a factor of 2. This delay would be intolerable under most circumstances.

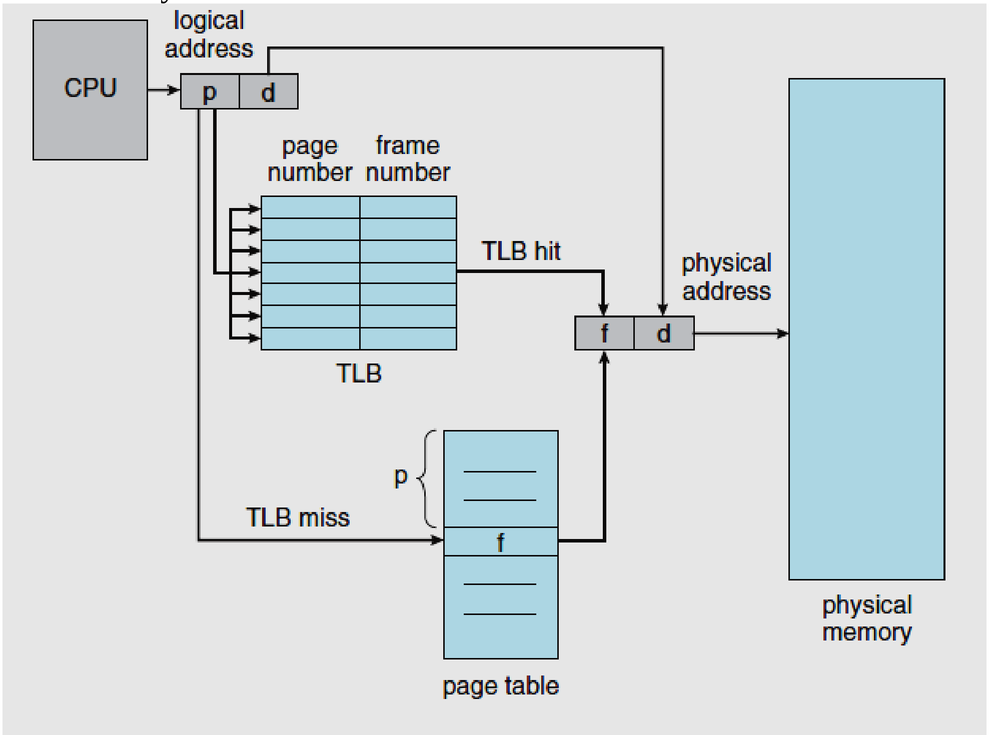


Figure 14 Paging hardware with TLB.

The solution to this problem is to use a special small fast-lookup hardware cache, called a Translation look-aside buffer(TLB). The TLB is associative high-speed memory. Each entry in the TLB consists of two parts: a key(or tag) and a value. When the associative memory is presented with an item, the item is compared with all keys simultaneously. If the item is found, the corresponding value filled is returned. The search is fast; the hardware however, is expensive. Typically, the number of entries in a TLB is small, often numbering between 64 and 1,024.

The TLB is used with page tables in the following way. The 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, its frame number is immediately available and is used to access memory. The whole task may take less than 10 percent longer than it would if an unmapped memory reference were used.

If the page number is not in the **TLB** (know as a **TLB miss**), a memory reference to the page table must be made. When the frame number is obtained, we can use it to access memory (figure 13). If the TLB is already full of entries, the operating system must select one of replacement. Replacement policies range from least recently used (LRU) to random.
The percentage of times that a particular page number is found in the TLB is called **hit ratio**.
Hit ratio = 80% i.e hit success = 80% and hit miss = 20%.
Time take to access TLB = 20 nanoseconds.
Time taken to access memory = 100 nanoseconds.
Hit success = TLB access + memory access = 20 + 100 = 120 nanoseconds.
Hit Miss = TLB access + memory access for page table + memory access for byte
20 + 100 + 100 = 220
Effective access time for 80% hit ratio = 0.80 X 120 + 0.20 X 220 = 140 nanoseconds.
Effective access time for 98% hit ratio = 0.980 X 120 + 0.02 X 220 = 122 nanoseconds.

Protection:
One bit can be defined a page to be read-write or read-only. Every reference to memory goes through the page table to find the frame number. At the same time physical address 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.
One additional bit is generally attached to each entry in the page table: a **Valid-invalid** bit. When this bit is set to "valid", the associated page is in the process's logical address space and is thus a legal page. When the bit is set to "invalid ", the page is not in the process's logical address space.
As shown fig 15, program has 0-5 pages. An attempt to generate an address in page 6 or 7, however, will find that the valid-invalid bits is set to invalid, and the computer tarp to OS.

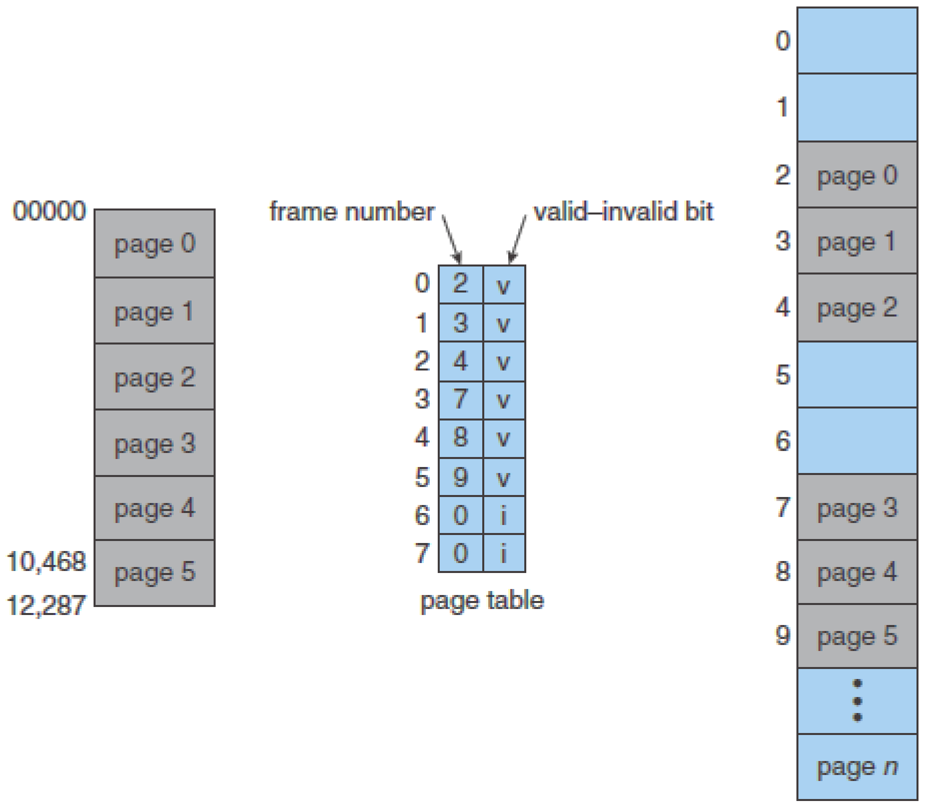


Figure .15 Valid (v) or invalid (i) bit in a page table.

Shared Pages:

An advantage of paging is the possibility of *sharing* common code. This consideration is particularly important in a time-sharing environment. Consider a system that supports 40 users, each of whom executes a text editor. If the text editor consists of 150KB of code and 50 KB of data space, we need 8,00KB to support 40 users. Assume each page size is 50KB. If the code is reentrant code, pages can be shared. As shown in fig 16, three page editors share the three code pages and each process has separate data page.

Reentrant code is non-self-modifying code; it never changes during execution. Thus two or processes can execute the same code at the same time.

Only, one copy of editor needs to be kept in physical memory. Each user's page table maps onto same physical copy of the editor, but the data pages are mapped onto different frames. Thus, to support 40 users, we need only one copy of the editor (150KB), plus 40 copies of the 50KB of data space per user. The total space required is now 2,150KB instead of 8,00KB.

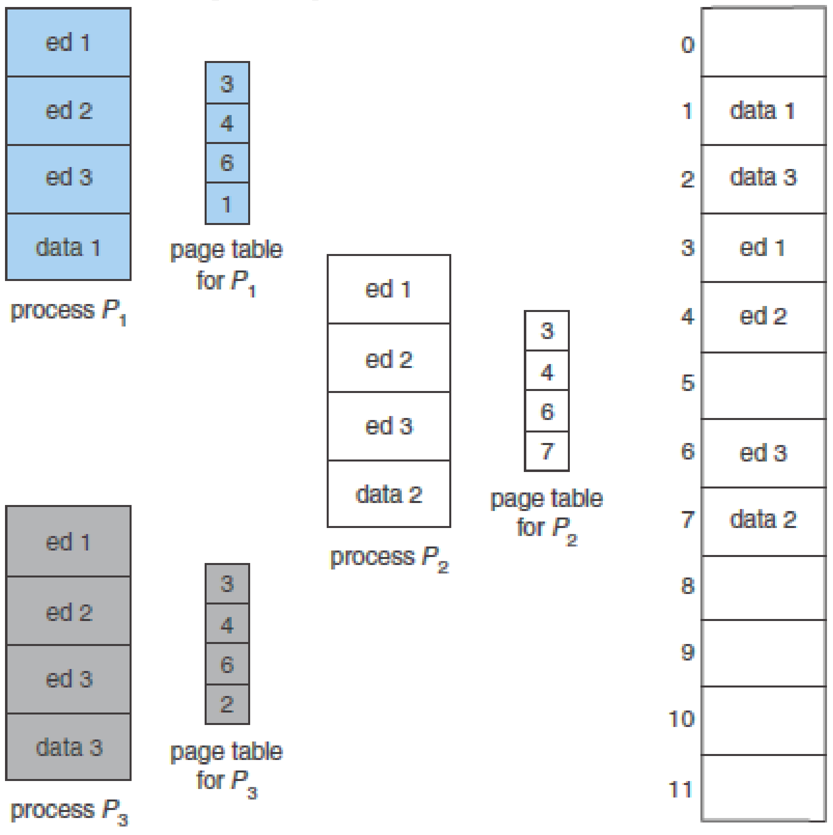


Figure 16 Sharing of code in a paging environment.

Structure of Page Tables:

Hierarchical Paging:

Most modern computer systems support a large logical address space (2^{32} or 2^{64}). In such case page table itself becomes excessively large. For example, consider a system with a 32-bit logical address space; if page size in such a system is 4KB (2^{12}), then the page table consists of one million entries. Assume each entry consists of 4 bytes; each process may need up to 4MB of physical address space for the page table alone. Clearly, we would not want to allocate the page table contiguously in main memory. One simple solution to this problem is to divide the page table into smaller pieces.

Two level Paging algorithm:

For a 32-bit machine with a page size of 4KB, a logical address is divided into page number consisting of 20 bits and a page offset consisting of 12 bits. The page number is further divided into a 10-bit page number and a 10-bit page offset. Thus, a logical address is as follows:

Page Number		Offset
P1	P2	d
10	10	12

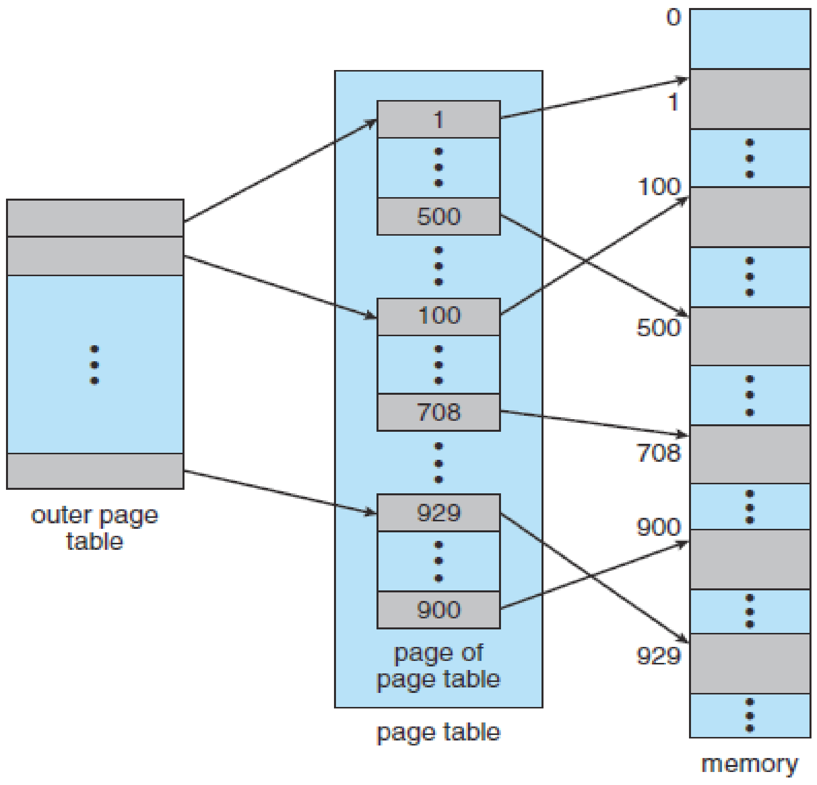


Figure 8.17 A two-level page-table scheme.

Where p1 is an index into the outer page table and p2 is the displacement within the page of outer page table. This scheme is also known as **forward-mapped page table**. The address translation scheme is mentioned in fig 18.

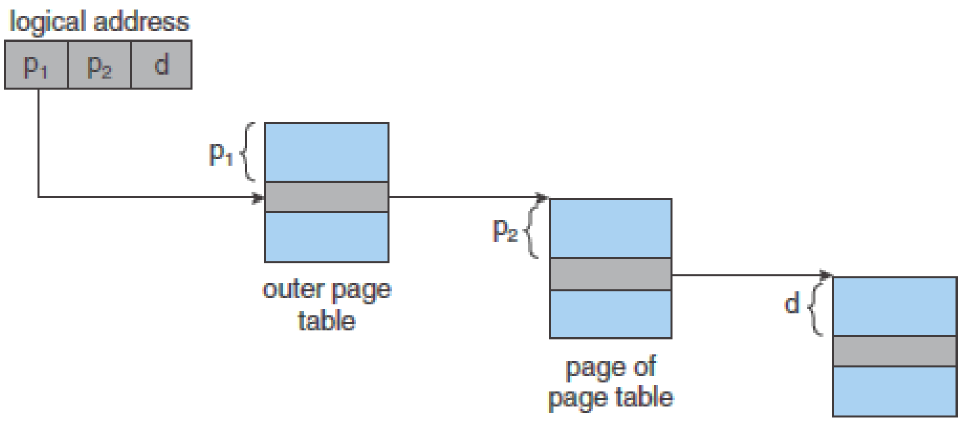


Figure ..18 Address translation for a two-level 32-bit paging architecture.

For a system with a 64-bit logical address, a two-level paging scheme is no longer appropriate. Assume page size is 4KB. If we adopt two level paging scheme

Outer Page	inner Page	Offset
P1	P2	d
42	10	12

The outer page table consists of 2^{42} entries. To avoid such a large page table is divide outer page table.

2nd outer page table	outer page	Inner Page	offset
P1	P2	P3	d
32	10	10	12

Hashed Page table:

A common approach for handling address spaces larger than 32 bits is to use a **Hashed Page Table** with a hash value being the virtual page number. Each entry in the hash table consists a linked list of elements that hash to the same location(Collisions). Each element consists of three fields (1)The virtual page number, (2) the value of mapped page frame, (3) A pointer to the next element in the linked list.

The algorithm works as follows: The virtual page numbering the virtual page address is hashed into the hash table. The virtual page number is compared with the field 1 in the first element in the linked list. If there is a match, the corresponding page frame(field2) is used to form desired physical address. If there is no match, subsequent entries in the linked list are searched for a matching value page number.

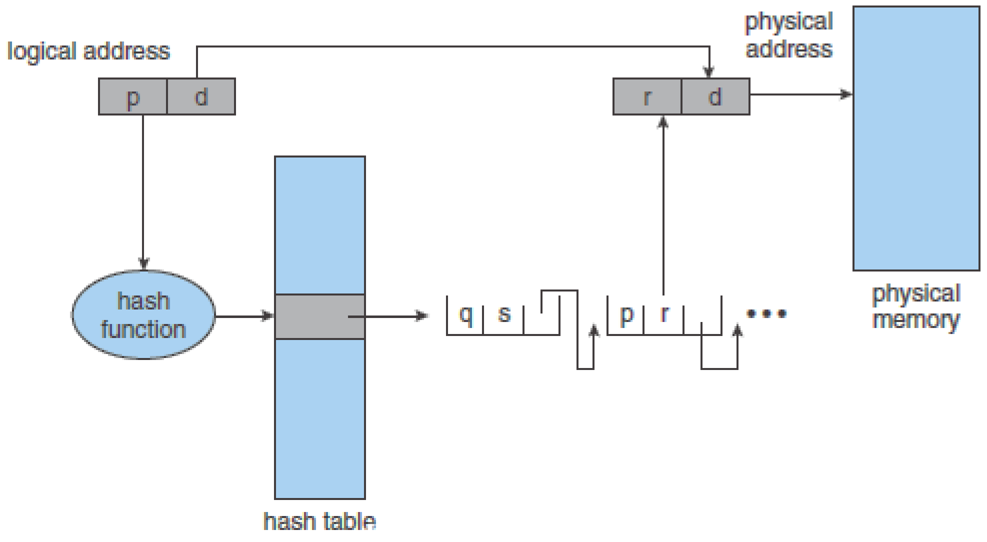


Figure 19 Hashed page table.

Segmentation:

Segmentation is a memory-management scheme that supports this user view of memory. A logical address space is a collection of segments. Each segment has name and offset within the segment. The segments are numbered and are referred to by a segment number, rather by a segment name. Thus, a logical address consists of a <segment-number, offset>. Whenever user compiles a program, it might create the following segments:

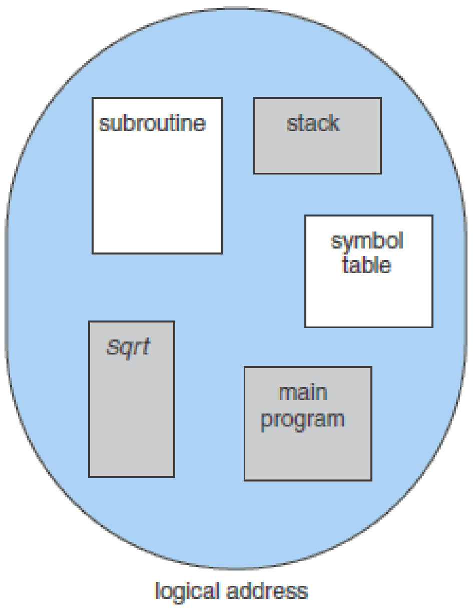


Fig 20: User's view of program

1. The code
2. Global Variables
3. The heap, from which memory is allocated
4. The stacks used by each thread
5. The standard C library

Hardware:

The segment table is used to map the two-dimensional logical address into one dimensional physical address. Each entry in the segment table has a segment base and 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.

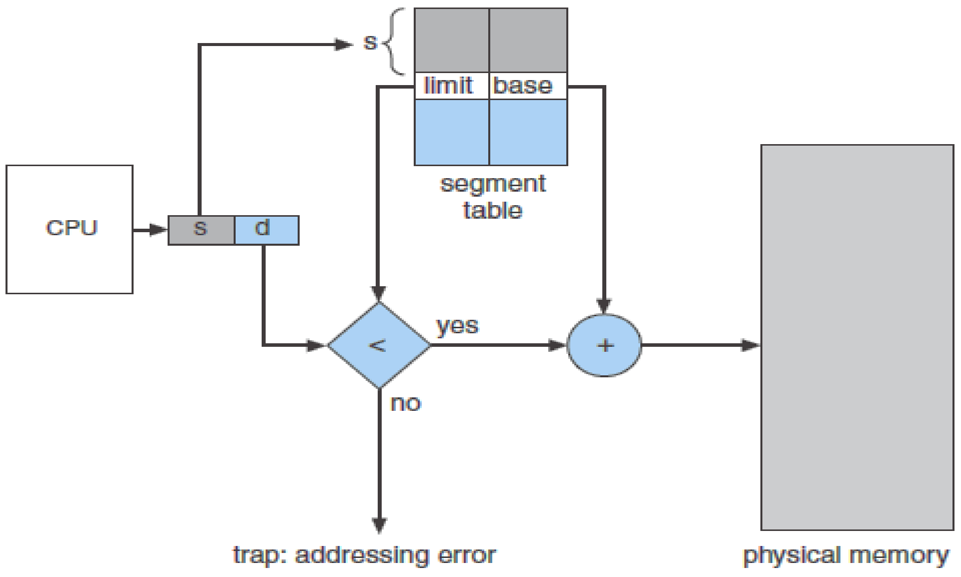


Fig 21: Segmentation Hardware

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 logical address must be between 0 and the segment limit. If it is not, we trap to the operating system. When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte.

Fig 22 have five segments numbered through from 0 to through 4. The segments are stored in physical memory as shown. The segment table has separate entry for each segment giving the beginning address of the segment in physical memory (or base) and the length of the segment (or limit). For example (fig: 22), segment 2 is 400 bytes long and beginning at location 4300. Thus, a reference to byte 53 of segment 2 is mapped onto location 4300 + 53 = 4353. A reference to segment 3, byte 852, is mapped to 3200 + 852= 4052. A reference to byte 1222 of segment 0 would result in a trap to OS, as this segment is only 1000 bytes long.

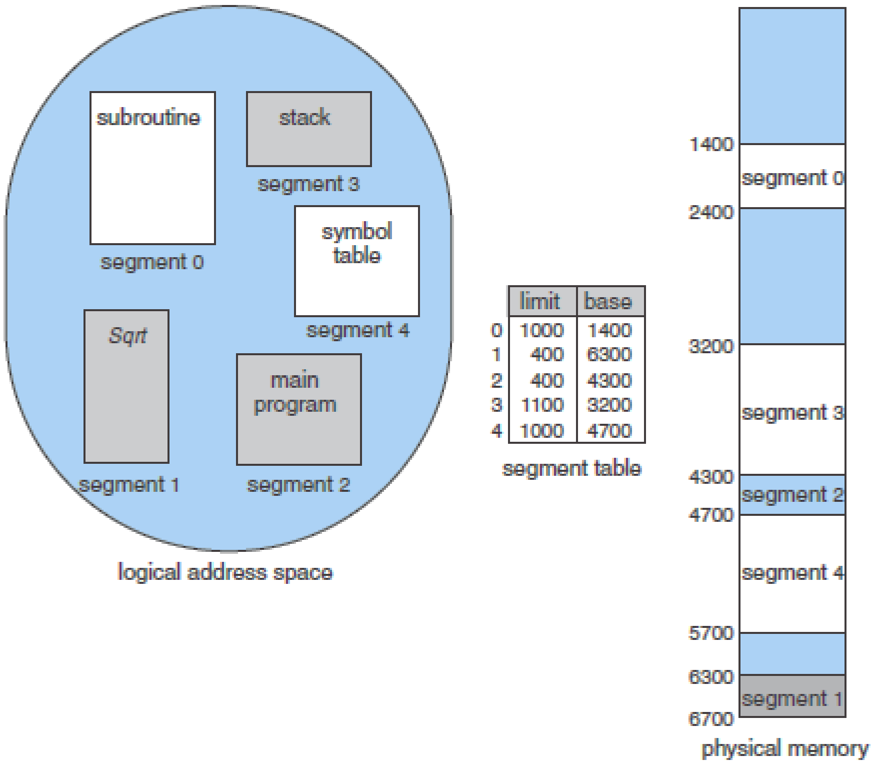


Fig 22: Example of Segmentation

Virtual Memory Management

The instructions being executed must be in physical memory. The requirement that instructions must be in physical memory to be executed seems both necessary and reasonable; but it limits the size of a program to the size of a program to the size of physical memory. In fact, an examination of real programs shows us that the entire program is rarely used. For instance, 1) Error handling routines are rarely used. 2) Arrays, lists and tables are often allocated more memory than they actually need.

Virtual Memory:

Virtual memory involves separation of logical memory from physical memory. This separation allows an extremely large virtual memory to be provided for programmers when only smaller memory is available (Fig 1). The programmer need not concentrate on the amount of physical memory.

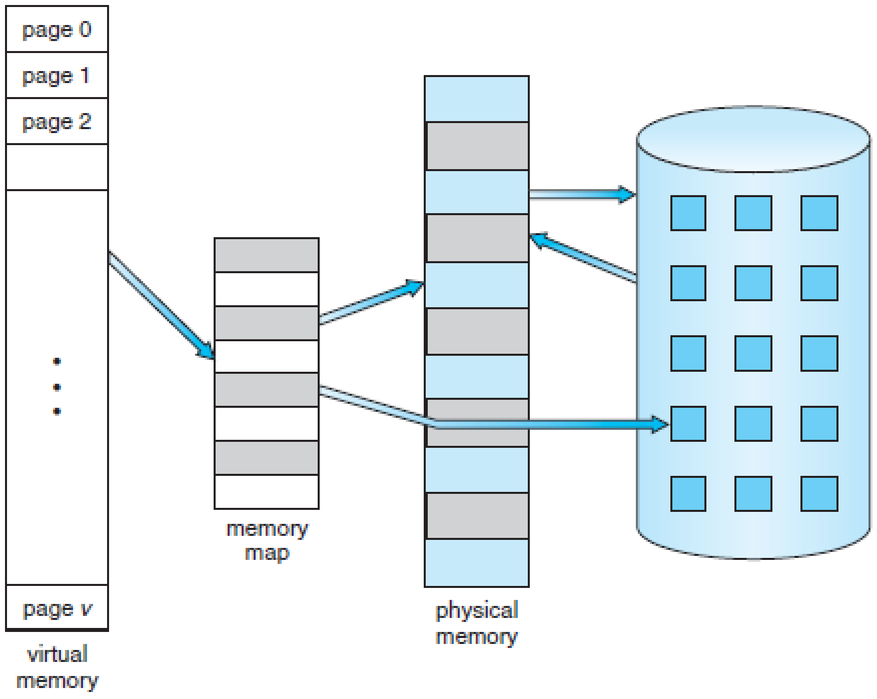


Fig 1: Diagram showing virtual memory that is larger than physical memory

The part of the program is in main memory and remaining part of memory will be in secondary memory as shown in Fig 1. The virtual memory allows files and memory to be shared by two or more processes through page sharing.

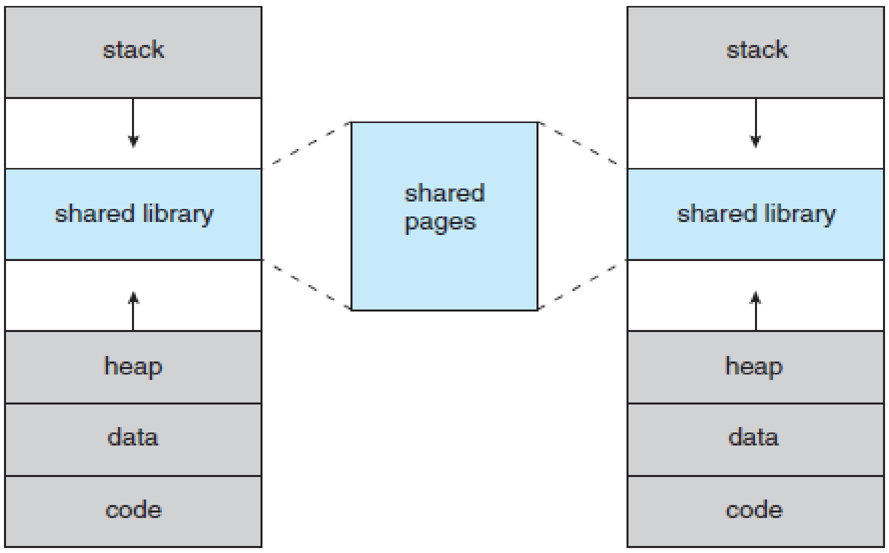


Fig 2: shared library using virtual memory

Demand Paging:

One option to execute a program is load the entire program into memory. Loading the entire program into memory results in loading the executable code for all options regardless of whether an option is ultimately selected by the user or not. 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 management. Pages that are

never accessed are never loaded into memory. Demand paging uses Lazy swapper. A Lazy swapper never swaps a page into memory unless that page will be needed.
Divide logical memory (programs) into blocks of same size called pages. Divide physical memory into equal sized partitions called frames (size is power of 2, between ½K and 8K). The size of the page and frame must be equal. When a process is to be executed, **some** of its pages are loaded into any available memory frames from the disk. Page table is used to map the page number to corresponding frame number. The page table contains the base address of the page (frame) in memory. Page number will be used as index for the page table. Whenever program is loaded into memory the corresponding frame number is updated to page table.

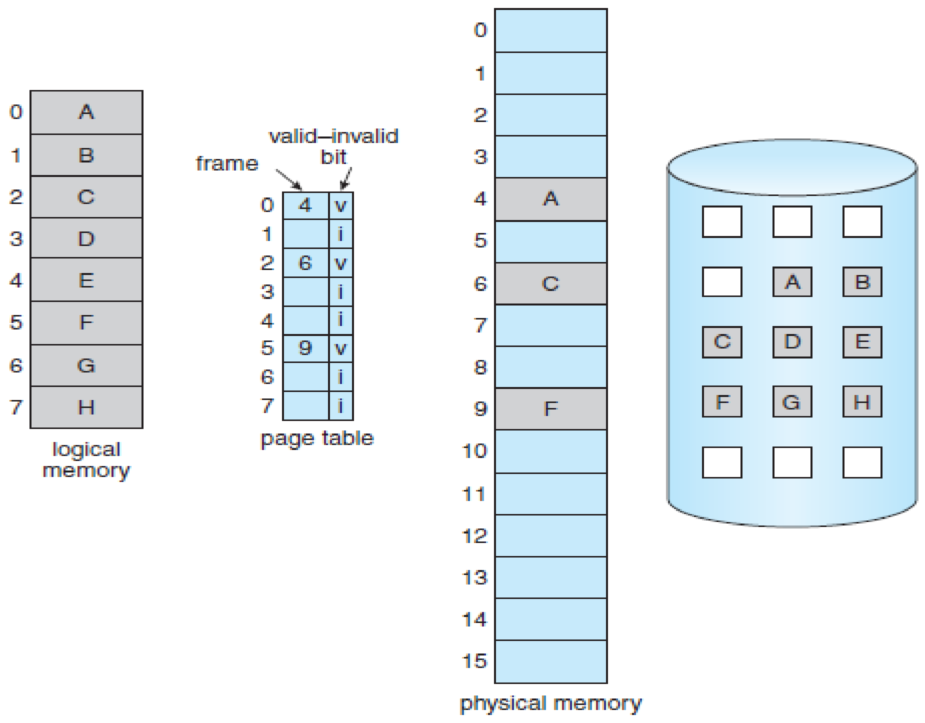


Fig : Page table when some pages are not in main memory.
With each page table entry a valid–invalid bit is associated. If valid-invalid bit is '**v**', the associated page is both legal and in-memory. If the valid-invalid bit is '**i**', the page is not valid or is valid but currently on the disk

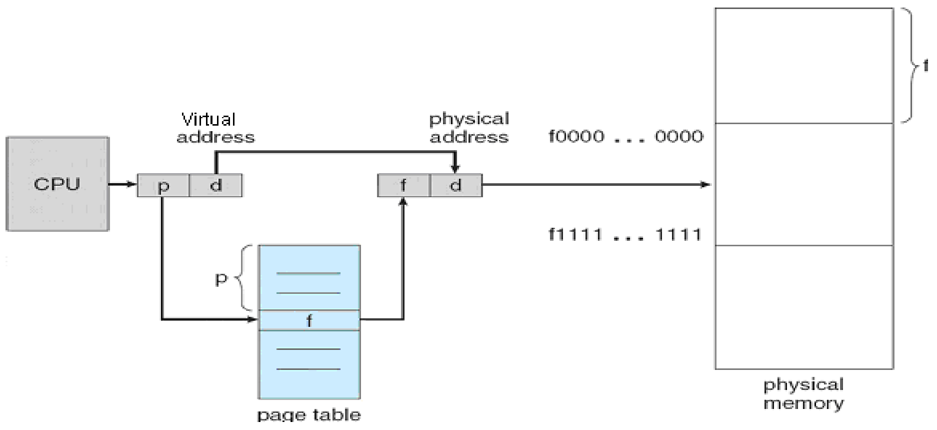


Fig 3: Address Translation in a paging system

CPU generates a virtual address and it contains page number and offset. The page number is used as index of the page table. If its corresponding valid-invalid bit is 'v', then physical address is generated by replacing page number in virtual address with corresponding frame number. If the corresponding valid-invalid bit is 'i' then **page fault** is generated.
The procedure for handling **page fault:**

1) Check memory access is valid or invalid.
2) If the reference is invalid terminate the process. If it is valid and valid-invalid bit is 'i', page fault is generated and trap to Operating system.
3) Find the free frame in the main memory and requested page on the disk.
4) Load the requested page into free frame.
5) Update page table with the corresponding frame number.
6) Restart the defaulted instruction.

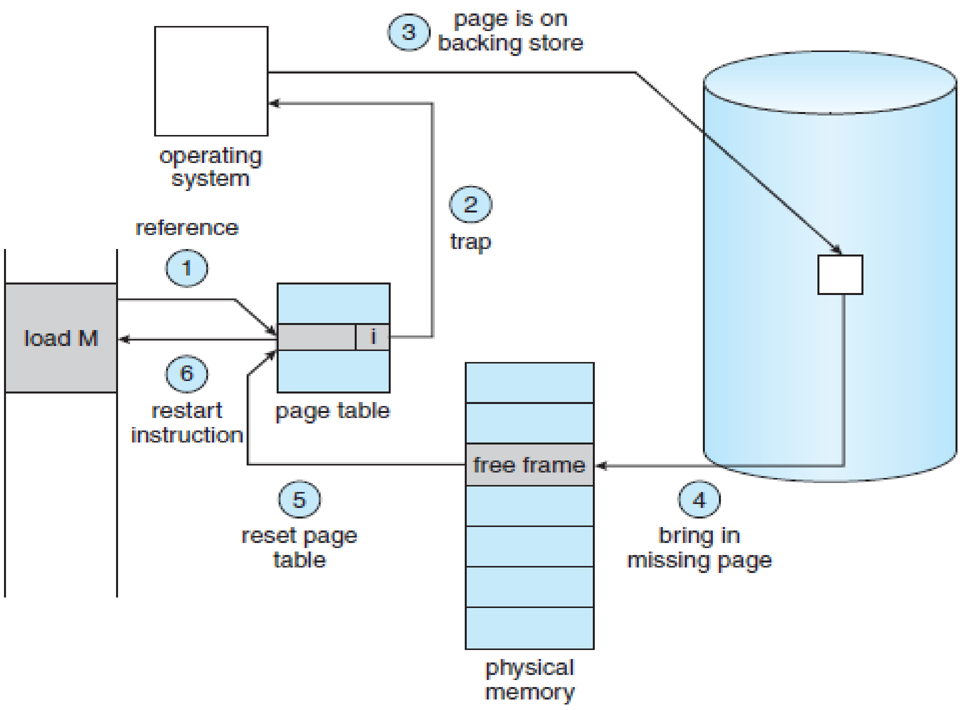


Fig 4: Steps in handling page faults

Performance of Demand Paging:

For most computer systems, the memory-access time, denoted *ma*, ranges from 10 to 200 nanoseconds. Let *P* be the probability of a page fault (0 <= P <=). We would expect *P* could be close to zero. The *effective access time* is then
Effective access time = (1 – *p*) X *ma* + *p* X page fault time.
To compute the effective access time, we must know how much time is needed to service a page fault. A page fault causes the following sequence to occur:
1. Trap to the operating system.
2. Save the user registers and process state.
3. Determine that the interrupt was a page fault.
4. Check that the page reference was legal and determine the location of the page on the disk.
5. Issue a read from the disk to a free frame:
   a. Wait in a queue for this device until the read request is serviced.
   b. Wait for the device seek and/or latency time.
   c. Begin the transfer of the page to a free frame.
6. While waiting, allocate the CPU to some other user (CPU scheduling, optional).

7. Receive an interrupt from the disk I/O subsystem (I/O completed).
8. Save the registers and process state for the other user (if step 6 is executed).
9. Determine that the interrupt was from the disk.
10. Correct the page table and other tables to show that the desired page is now in memory.
11. Wait for the CPU to be allocated to this process again.
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction.

Page fault service time = 8 milli seconds (8,000,000 nano seconds) and memory access time is 200 nano seconds.

Effective access time = $(1 – p)$ X (200) + p X 8 milliseconds.

$= (1 – p)$ X 200 + p X 8,000,000

$= 200 + 7{,}999{,}800$ X p

The effective access time is directly proportional to the ***page-fault rate.***

Copy-on-write:

Fork is a system call creates a child process as a duplicate of its parent. fork() worked by creating a copy of the parent's address space for the child, duplicating the pages belonging to the parent. Instead of that, a copy-on-write technique can be used to share same pages by parent and child. These shared pages are marked as copy-on-write pages, meaning that if either process writes to a shared page, a copy of shared page created. Copy-on-write allows more efficient process creation as only modified pages are copied.

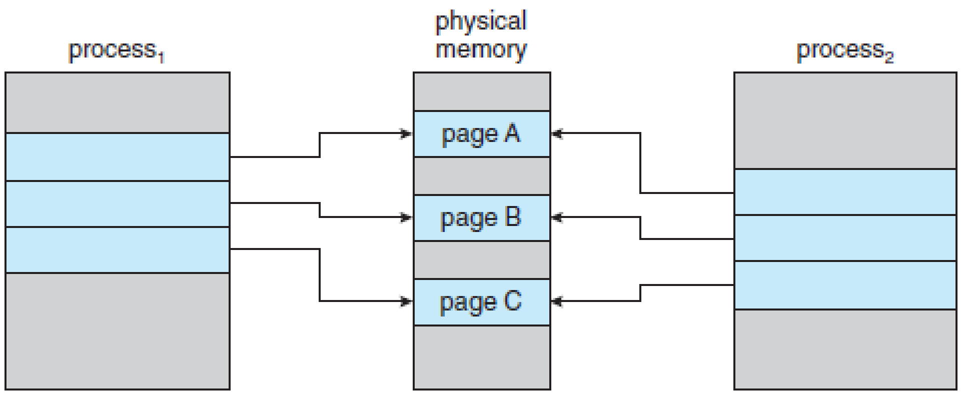


Fig 5: Before process 1 modifies page C

For example, assume that the process1 attempts to modify page C, with the page set to be Copy-on-write. The operating system will then create a copy of this page, mapping it to the address space of process1. The process1 will then modify its copied page and not belonging to shared page with process2. All unmodified pages can be shared by parent and child. Copy-on-write is a common technique used by windows-xp, linux, and solaris.

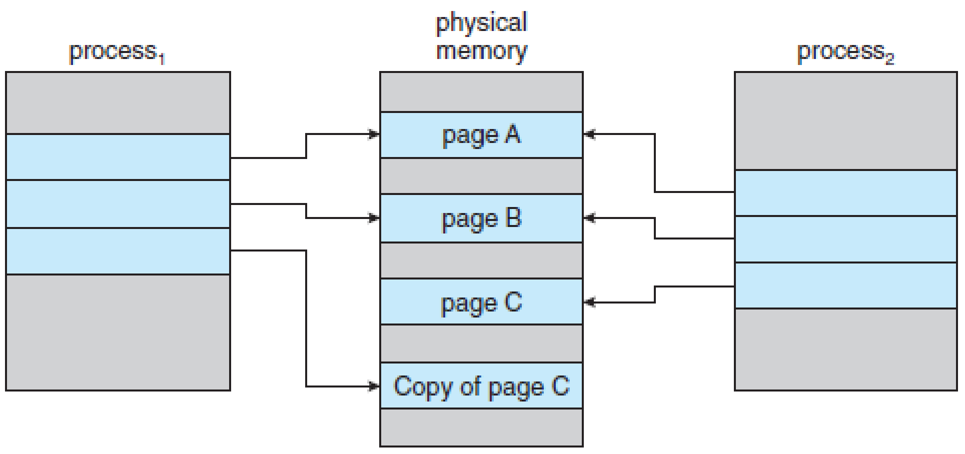


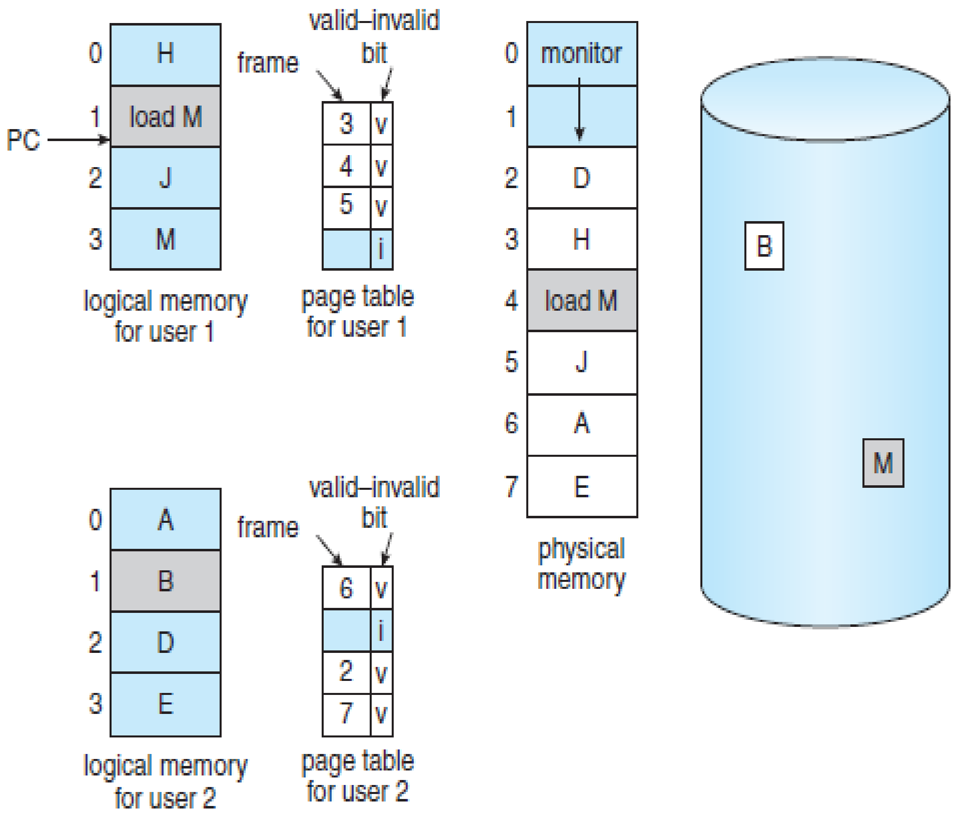
Fig 6: After process 1 modifies page C

Page Replacement Algorithms:

Need for page replacement:

While a user process is executing, a page fault occurs. The operating system determines where the desired page is residing on the disk but then finds that there are no free frames on the free-frame list; all memory is in use (Figure below). The operating system has several options at this point. It could terminate the user process. However, demand paging is the operating system's attempt to improve the computer system's utilization and throughput. Users should not be aware that their processes are running on a paged system—paging should be logically transparent to the user. So this option is not the best choice. The operating system could instead swap out a process, freeing all its frames and reducing the level of multiprogramming.

Here, we discuss the most common solution: page replacement.



Whenever page fault occurs:

1. Find the requested page on the disk.
2. Find the free frame in the memory.
    a) If there is a free frame, use it.
    b) If there is no free frame use page replacement algorithm to find a victim frame.
    c) Write the victim frame to the disk; change the page and frame tables accordingly.
3. Load the requested page into free frame.
4. Update page table with the corresponding frame number and set valid-invalid bit = 'V'.
5. Restart the instruction that caused the page fault.

The page replacement algorithm finds out the victim page. 1) If there are any modifications on that page, the victim page will be written on to the disk. 2) The corresponding page entry values in the page table are modified (i.e frame number =0, and valid-invalid bit = 'i'). 3) The requested page will be loaded into vacated frame in the memory 4) Reset the page table for new page.

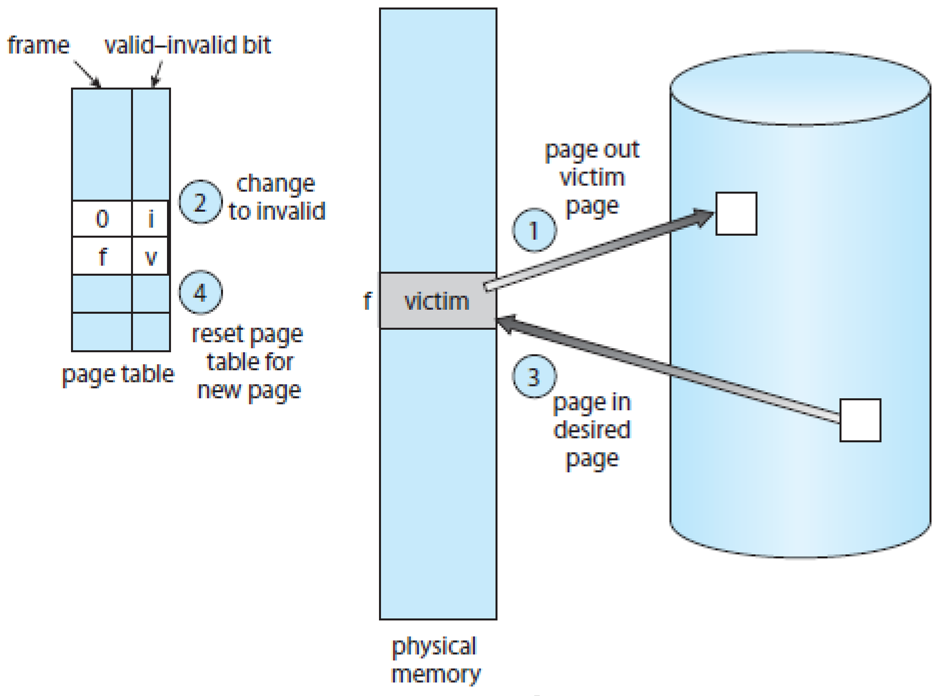


Fig 7: Page Replacement

FIFO Page replacement:

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. (OR) We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue.

Assume that memory contains only three frames. Initially all three frames are empty. The page reference string is 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1. The first three references (7,0,1) cause page faults and are brought into these empty frames.

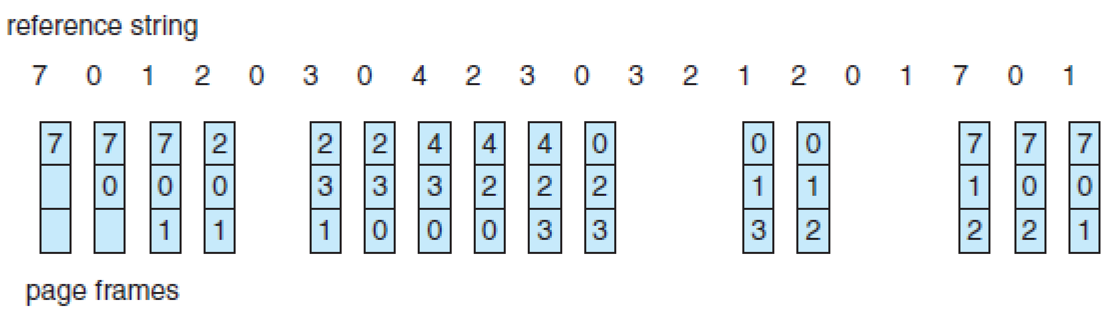


Fig 8: FIFO page replacement algorithm

The next reference (2) replaces page 7, because the page 7 was brought in first. Since 0 is the next reference and 0 is already in main 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. This process will continue as shown in fig 8.

Notice that, even if we select for replacement a page that is in active use, every thing works correctly. After we replace an active page with a new one, a fault occurs almost immediately to retrieve the active page. Some other pages will need to be replaced to bring the active page back into memory. Thus, a bad replacement choice increases the page-fault rate and slow process execution.

Belady's anomaly:

Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

3 frames (3 pages can be in memory at a time per process)

1	1	4	5	
2	2	1	3	9 Page faults
3	3	2	4	

Fig 9: FIFO page replacement algorithm with 3 frames

1	1	5	4	
2	2	1	5	10 page faults
3	3	2		
4	4	3		

Fig 10: FIFO page replacement algorithm with 4 frames

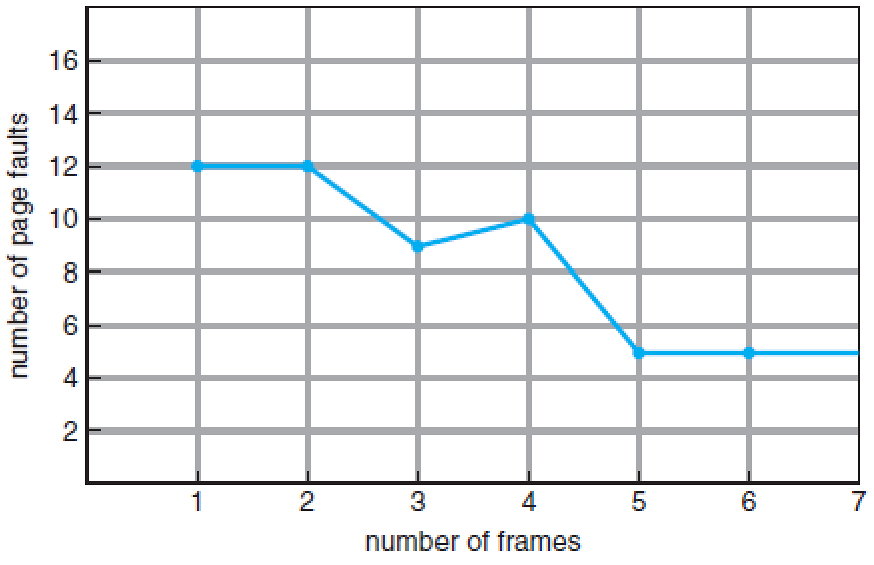


Fig 11: FIFO Illustrating Belady's Anomaly

Notice that number of faults for four frames (ten) is greater than the number of faults for three frames (nine). This most unexpected result is known as **Belady's anomaly**: for some page-replacement algorithms, the page fault rate may increase as number of allocated frames increases.

Optimal Page Replacement:

One result of the discovery of **Belady's anomaly** was the search for an *Optimal page-replacement algorithm*. An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms and will never suffer from **Belady's anomaly**. Optimal algorithm simply replaces the page that will not be used for the longest period of time. Use of optimal page-replacement algorithm guarantees the lowest possible page fault rate for a fixed number of frames.

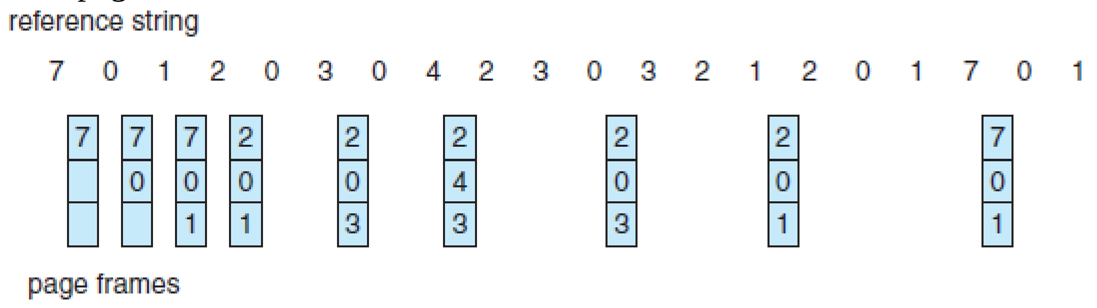


Fig 12: Optimal Page Replacement algorithm

For example, on our sample reference string, the optimal-page replacement algorithm would yield nine page faults as shown in fig 12. With only nine page faults, optimal replacement algorithm is much better than a FIFO algorithm. The optimal algorithm replaces the pages based on the future references of pages. But prediction of future references of pages is very difficult.

LRU (least-recently-used) page replacement algorithm:

The LRU algorithm replaces the page that has not been used for longest period of time. This approach is the least-recently-used (LRU) algorithm. LRU is associated with each page the time of that page's last use. When a page must be replaced, LRU chooses the page that has not been used for the longest period of time.

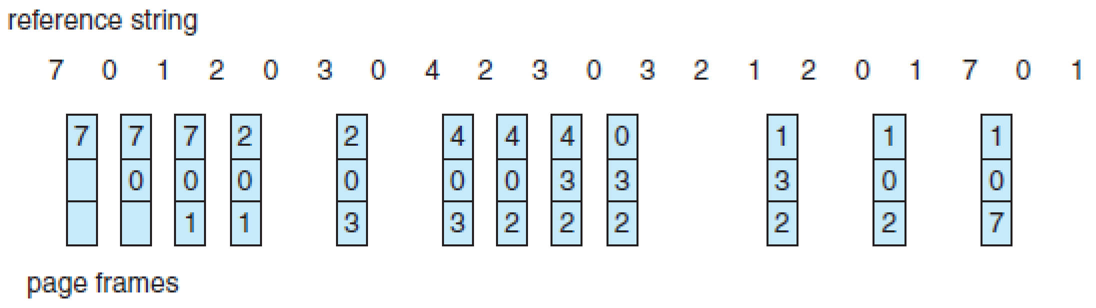


Fig 13: LRU Page Replacement algorithm

Implementation of LRU page replacement Algorithm:

Counters: We associate with each page-table entry a time-of-use field and add to the CPU logical clock or counter. The clock is incremented for every memory reference. Whenever a reference to a page is made, the contents of the clock register are copied to time-of-use field in the page-table entry for that page. We replace the page with the smallest time value. This scheme requires a search of the page table to find LRU page.

STACK: Another approach to implement LRU replacement algorithm is to keep a stack of page numbers. Whenever a page is referenced, it is remove from the stack and put it on the top. In this way, the most recently used page is always at the top of the stack and the least recently used page is always at the bottom. It is the best to implement the stack by using double linked list.

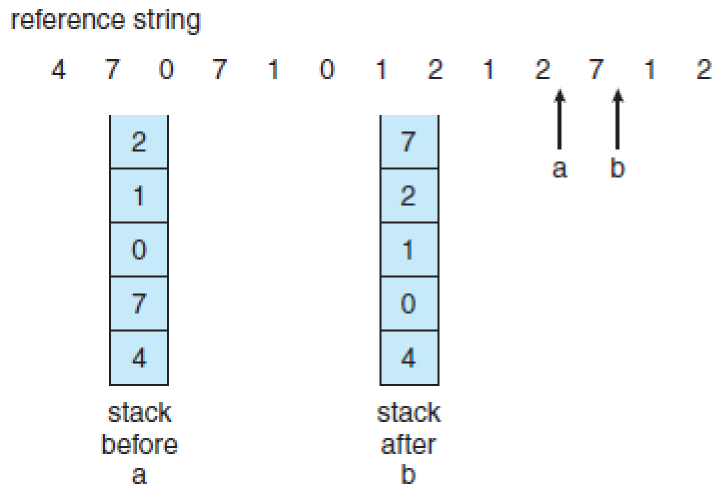


Fig 14: Use of a stack to record the most recent page references.

LRU-Approximation Page Replacement Algorithm:

Some systems provide few computer systems provide sufficient hardware support for true LRU page replacement. Some systems provide no hardware support, and other page replacement algorithm must be used. Many systems provide reference bit for each page entry in the page table. Initially all bits are cleared (set to 0) by operating system. As a user process executes, the bit associated with each page reference is set (to 1) by the hardware.

Additional-Reference-Bits Algorithm:

We can keep an 8-bit byte for each page in a table in memory. At regular intervals (every 100 milliseconds), a timer interrupt transfers control to the operating system. The operating system shifts the reference bit for each page into the high-order bit of its 8-bit byte, shifting the other bits right by 1 bit discarding low-order bit. If the shift register contains 00000000, for example, then the page has no been used for eight time periods; a page that is used at least once in each period has a shift register value of 11111111. A page with a history register value of 10000000 has been used more recently than one with the value of 01111111. The page with the lowest 8-bit byte number is the LRU page, and it can be replaced.

Second-Chance Algorithm or Clock algorithm:

It is like a FIFO replacement algorithm. When a page has been selected, we inspect its reference bit. If the value is 0, we proceed to replace this page; but if the reference bit is set to 1, we give the page a second chance and move on to select the next FIFO page. If a page is used often enough to keep its reference bit set, it will never be replaced.

One way to implement the second-chance algorithm is as circular queue. A pointer indicates that which page to be replaced next as shown in fig 13. When a frame is needed the pointer advances until it finds a page with a 0 reference bit. As it advances it clears reference bits. Once victim page is found, the page is replaced, and the new page is inserted in the circular queue in that position. Second-chance replacement degenerates to FIFO replacement if all bits are set.

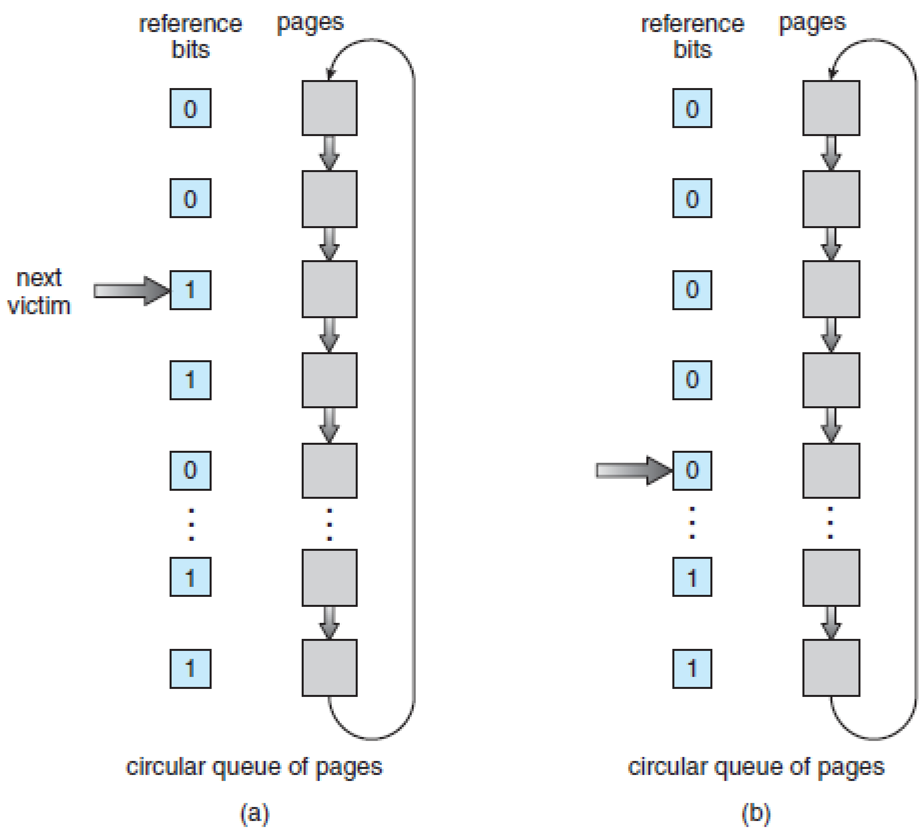


Fig 15: Second-chance (clock) page-replacement algorithm

Enhanced second-chance algorithm:

We can enhance the second-chance algorithm by considering the reference bit and modify bit as an ordered pair. With these two bits, we have the following four possible classes:

1. (0,0) neither recently used nor modified – best page to replace.
2. (0,1) not recently used but modified – the page will need to be written out before replacement.
3. (1,0) recently used but clean – probably will be used again soon.
4. (1,1) recently used and modified – probably will be used again soon, and the page will be need to be written out to disk before it can be replaced.

Each page is in one of these four classes. When page replacement is called for, we examine the class to which that page belongs. We replace the first page encountered in the lowest nonempty class.

Counting-based page replacement algorithm:

We can keep a counter of the number of references that have been made to each page and develop the following two schemes:

The least frequently used (LFU) page-replacement algorithm requires that the page with the smallest count be replaced. The reason for this selection is that an actively used page should have large reference count. A problem arises, however, when a page is used heavily during the initial phase of a process but that is never used again. Since it was used heavily, it has a large count and remains in memory even though it is no longer needed. One solution is to shift the counts right by 1 bit regular intervals forming an exponential decaying average usage count.

The most frequently used (MFU) page-replacement algorithm is based on the argument that the page with smallest count was probably just brought in and has yet to be used.

Thrashing

Assume any process does not have “enough” frames and memory is full. When the operating system brings one page in, it must throw another out. If it throws out a piece just before it is used, then it will just have to go get that piece again almost immediately. Too much of this leads to a condition known as thrashing.

Causes of thrashing:

Thrashing results in severe performance problems. The OS monitors CPU utilization. If CPU utilization is too low, we increase degree of multiprogramming by introducing new process to the system. A global page-replacement algorithm is used; it replaces pages without regard to the process to which they belong. When a process starts faulting, it takes frames away from other process. These processes need those pages, and so they also fault, taking frames from other processes. As they queue up for the paging device, the ready queue empties.

The CPU scheduler sees the decreasing CPU utilization and increases the degree of multiprogramming as a result. The new processes tries to get started by taking frames from running processes, causing more page faults and longer queue for the paging device. As a result CPU utilization drops even further and CPU scheduler tries to increase the degree of multiprogramming even more. Thrashing has occurred, and system throughput plunges. The page fault rate increases tremendously. No work is getting done, because the processes are spending all their time in paging.

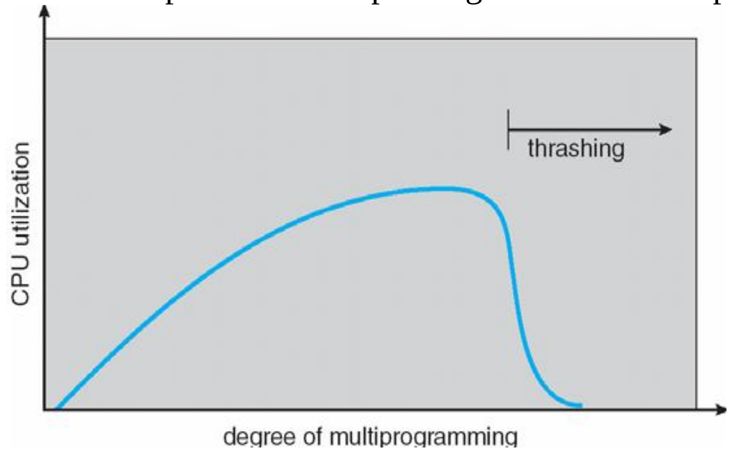


Fig 16: Thrashing

In fig 16, CPU utilization is plotted against the degree of multiprogramming. As the degree of multiprogramming increases, CPU utilization also increases, until maximum reached. 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.

We can limit the effects of thrashing by local replacement algorithm. With local replacement, if one process starts thrashing, it cannot steal frames from other process.

Working-set model

To prevent thrashing, we must provide a process with as many frames as it needs. The working-set strategy starts by looking at how many frames a process is actually using. This approach defines the locality model of process execution. A locality is a set of pages that are actively used together. A program is generally composed of several different localities. A process moves from locality to another locality as process execution continues.

This model uses a parameter, Δ, to define working set window. The idea is to examine the most recent Δ page references. The set of pages in the most recent Δ page references is the **working set**.

If the page is in active use, it will be in the working set. If it is no longer being used, it will drop from the working set Δ time after its last reference. Thus, the working set is approximation of the program's locality.

For example, given the sequence of memory references shown in fig 15, if Δ = 10 memory references, then the working set at time t_1 is { 1,2,5,6,7}. By time t_2, the working set has changed to {3,4}. The accuracy of working set depends on the section of Δ. If Δ is too small, it will not encompasses the entire locality; if Δ is too large, it may overlap several localities.

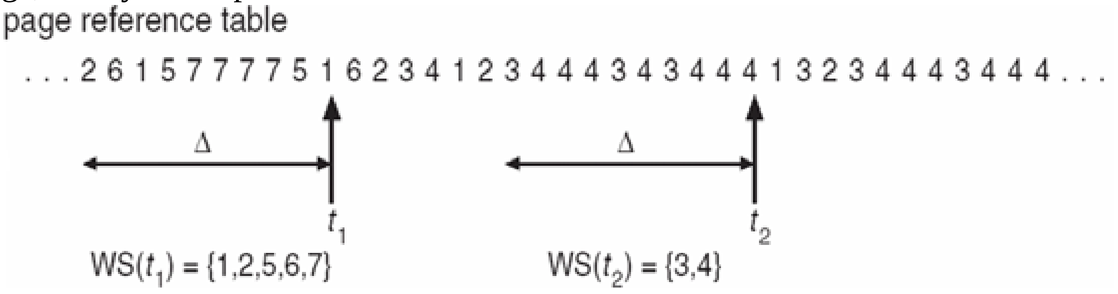


Fig 17: Working-set model

Allocating Frames:

Consider a single-user system with 128KB of memory composed of pages 1 KB in size. This system has 128 frames. The operating system may take 35KB, leaving 93 frames for user process. Under pure demand paging, all 93 frames would initially be put on the free-frame list. When a user process started execution, it would generate a page faults. The first 93 page faults would all get free frames from the free-frame list. When the free-frame list was exhausted, a page-replacement algorithm would be used to select one of 93 in-memory pages to be replaced with the 94th, and so on. When the process terminated, the 93 frames would once again be placed on the free-frame list.

Allocation Algorithm:

The easiest way to split ***m*** frames among ***n*** processes is to give everyone an equal size, ***m/n*** frames. For instance, if there are 93 frames and five processes, each process will get 18 frames. The leftover three frames can be used as a free-frame buffer pool. This scheme is called **Equal Allocation**.

Now consider a system with a 1-KB frame size. If a small student process of 10KB and an interactive database process of 127KB are only two processes are only two processes running in a system with 62 free frames, it does not make much sense to give each process 31 frames. The student process does not need more than 10 frames, so the other 21 are, strictly speaking wasted.

Proportional allocation:

In this method, frames are allocated based on the size of the process. Let the size of the virtual memory for process p_i be defined as s_i,

$$S = \Sigma\ s_i$$

Then, if the total number of available frames is ***m***, we allocate a_i frames to process p_i, where a_i is approximately

$$a_i = s_i / S \times m$$

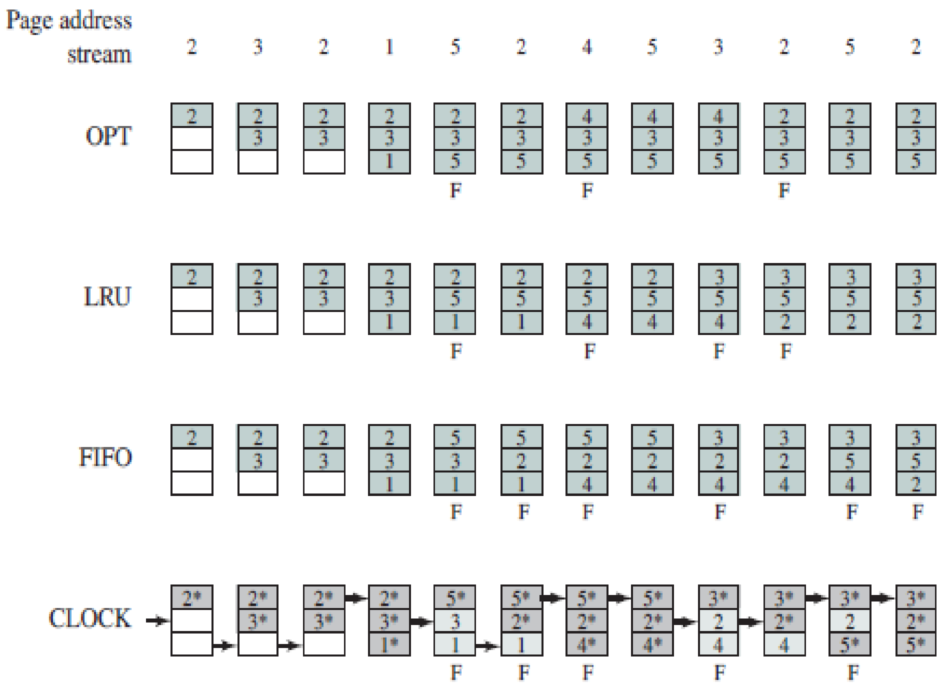
Now, if we split 62 pages among student and data base processes:

Student process will get 10/137 X 62 is approximately 4 frames

Database Process will get 127/137 X 62 is approximately 57 frames
If multi program level is increased, each process will lose some frames to provided the memory need.

Global Versus Local Allocation:

We can classify the page-replacement algorithms into two broad categories: **Global replacement and local replacement**. Global replacement allows a process to select a replacement frame from the set of all frames, even if that frame is currently allocated to some other process; i.e one process can take frames from another. Local replacement requires that each process select from only its own set of allocated frames. With a local replacement strategy, the number of frames allocated to a process does not change. With global replacement, a process may happen to select only frames allocated other process, thus increasing the number of frames allocated to it. The problem with the global page replacement algorithm is that a process canot control its own page-fault rate.



F = page fault occurring after the frame allocation is initially filled

Figure **Behavior of Four Page Replacement Algorithms**