UNIT-IV

Memory Management and Virtual Memory - Logical versus Physical Address Space, Swapping, Contiguous Allocation, Paging, Segmentation, Segmentation with Paging, Demand Paging, Page Replacement, Page Replacement Algorithms.

****

Logical & physical Address Space:

- An address generated by the CPU is commonly referred to as l a logical address. The set of all logical addresses generated by a program is known as **logical address space**.
- Whereas, an address seen by the memory unit- that is, the one loaded into the memory-address register of the memory- is commonly referred to as physical address. The set of all physical addresses corresponding to the logical addresses is known as **physical Address Space**.
- The compile-time and load-time address-binding methods generate identical logical and physical addresses. However, in the execution-time address-binding scheme, the logical and physical-address spaces differ.
- The run-time mapping from virtual to physical addresses is done by a hardware device called the **memory management unit (MMU**). We can choose from many different methods to accomplish such mapping.

For example,

- If the base is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location 14000; an access to location 346 is mapped to location 14346.
- The MS-DOS operating system running on the Intel 80 x 86 families of processors used four relocation registers when loading and running processes.

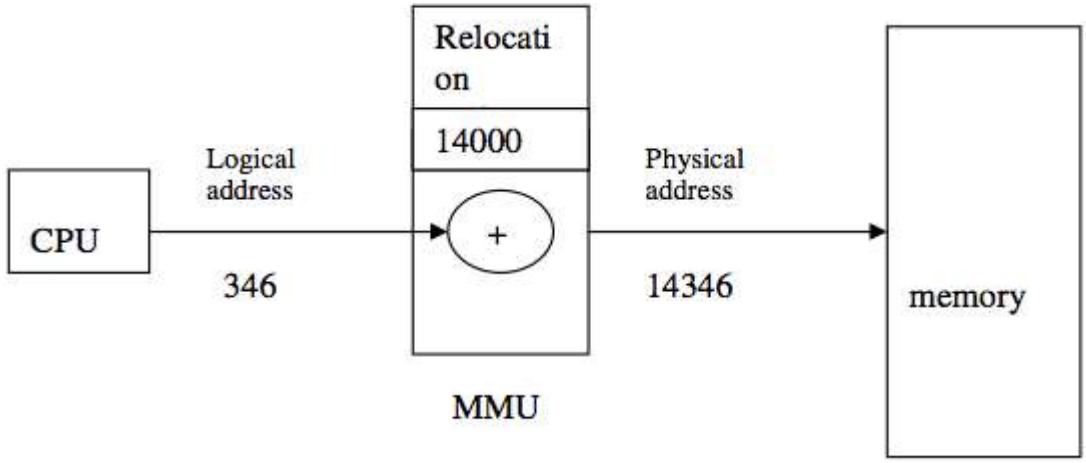


Figure A: Dynamic relocation using a relocation register

- The user program never sees the physical addresses. The program creates a pointer to a logical address, says 346, stores it in memory, manipulates it, and compares it to other logical addresses- all as the number 346.
- Only when a logical address is used as memory address, it is relocated relative to the base/relocation register. The memory-mapping hardware device called the memory- management unit (MMU) converts logical addresses into physical addresses.
- We now have two different types of addresses: logical addresses (in the range 0 to max) and physical addresses (in the range R+ 0 to R+ max for a base value R).
- The user generates only logical addresses and thinks that the process runs in locations 0 to max. However, these logical addresses must be mapped to physical addresses before they are used.

Swapping

- A process must be in memory to be executed. A process, however, can be swapped temporarily out of memory to a backing store and then brought into memory for continued execution.
- **For example,** assume a multiprogramming environment with a round-robin CPU-scheduling algorithm. When a quantum expires, the memory manager will start to swap out the process that just finished and to swap another process into the memory space that has been freed (Figure B).
- In the meantime, the CPU scheduler will allocate a time slice to some other process in memory. When each process finishes its quantum, it will be swapped with another process.

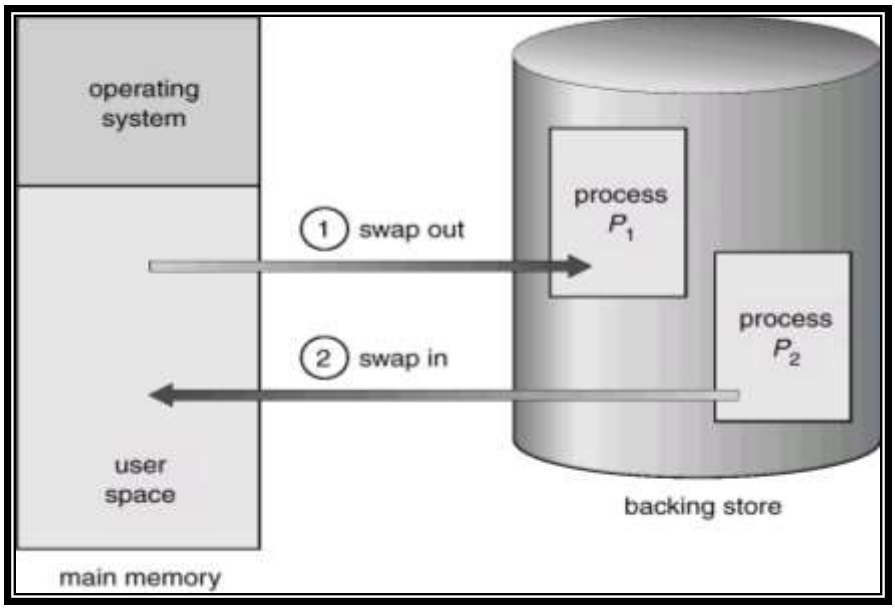


Figure B: Swapping of two processes using a disk as a backing store.

- A variant of this swapping policy is used for priority-based scheduling algorithms. If a higher-priority process arrives and wants service, the memory manager can swap out the 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 requires **a backing store**. The backing store is commonly a fast disk. It must be large enough to accommodate copies of all memory images for all users, and it must provide direct access to these memory images. The system maintains a consisting of all processes whose memory images are on the backing store or in memory and are ready to run.

Memory Allocation

- **Contiguous Allocation**
  - Each process allocated a single contiguous chunk of memory
- **Non-contiguous Allocation**
  - Parts of a process can be allocated noncontiguous chunks of memory

Contiguous Memory Allocation

- The memory is usually divided into two partitions:
  - Resident operating system Resident operating system, usually held in low memory with interrupt vector.
  - User processes then held in high memory.
- Since the interrupt vector is often in low memory, programmers usually place the operating system in low memory as well.
- We usually want several user processes to reside in memory at the same time. We therefore need to consider how to allocate available memory to the processes . contiguous memory allocation, each process is contained in a single contiguous section of memory
- Relocation-register scheme used to protect user processes from each other, and from changing operating-system code and data.
- Relocation register contains value of smallest physical address; limit register contains range of logical addresses – each logical address must be less than the limit register.

Fragmentation

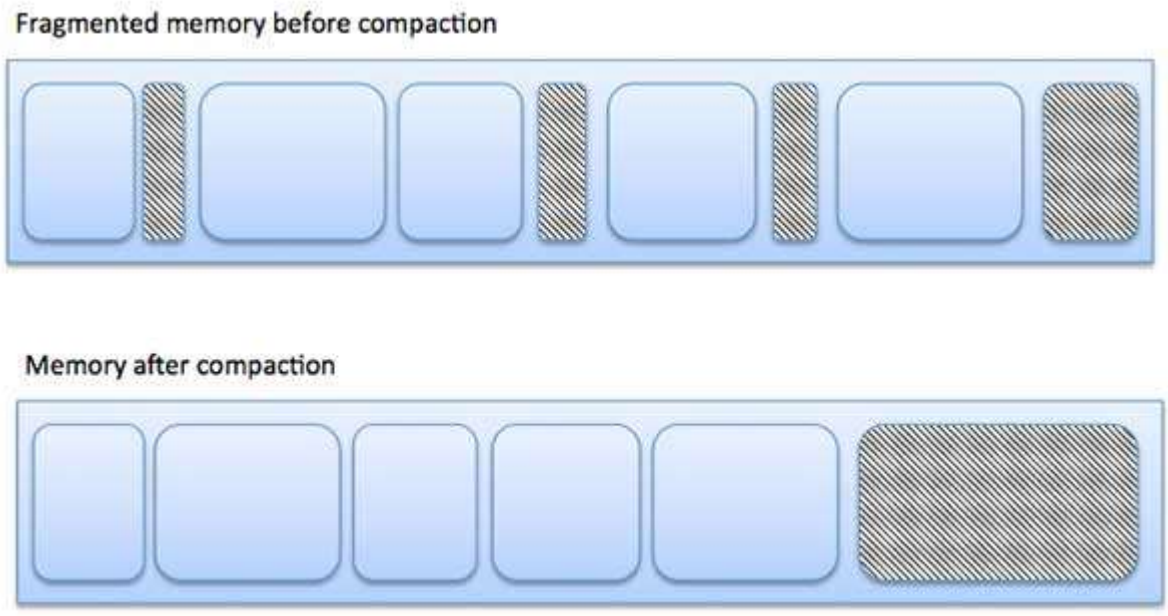
- As processes are loaded and removed from memory, the free memory space is broken into little pieces. It happens after sometimes that processes cannot be allocated to memory blocks considering their small size and memory blocks remains unused. This problem is known as **Fragmentation**.

Fragmentation is of two types –

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

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

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



- External fragmentation can be reduced by compaction or shuffle memory contents to place all free memory together in one large block. To make compaction feasible, relocation should be dynamic.
- The internal fragmentation can be reduced by effectively assigning the smallest partition but large enough for the process.

Paging:

Paging is a memory management mechanism that allows the physical address space of a process to be non-contagious. A solution to fragmentation problem is paging.

- ❖ Divide physical memory into fixed-sized (power of 2) blocks called **frames** or **blocks**
- ❖ Divide logical memory into blocks of same size called **pages**
- ❖ Keep track of all free frames.
- ❖ To run a program of size n pages, need to find n free frames and load program.
- ❖ **Page table**: used to translate logical to physical addresses

- The basic idea behind paging is to divide physical memory into a number of equal sized blocks called **frames**, and to divide programs logical memory space into blocks of the same size called **pages.**
- Any page (from any process) can be placed into any available frame.
- The **page table** is used to look up what frame a particular page is stored in at the moment. In the **following example**, for instance, page 2 of the program's logical memory is currently stored in frame 3 of physical memory:

NOTE: The line or offset address in address space and memory space is the same; the only mapping required is from a page number to a block/frame number

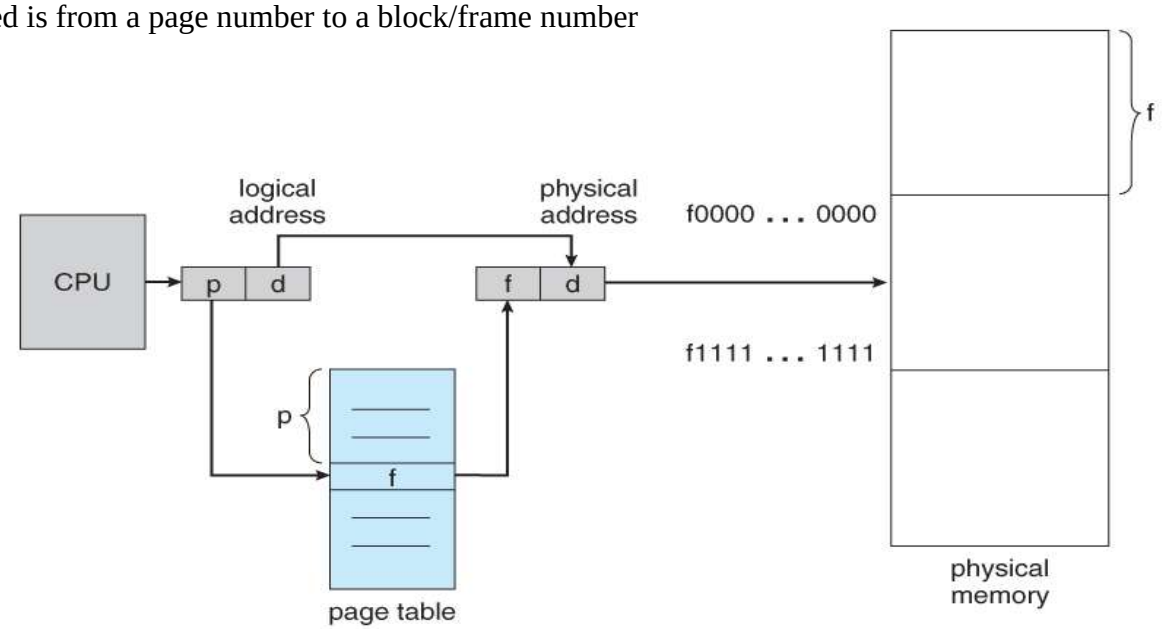


Figure C: Paging hardware

- A logical address consists of two parts: A **page number** in which the address resides, and an **offse**t from the beginning of that page. (The number of bits in the page number limits how many pages a single process can address. The number of bits in the offset determines the maximum size of each page, and should correspond to the system frame size.)

- The **page table maps** the page number to a frame number, to yield a physical address which also has two parts: The frame number and the offset within that frame. The number of bits in the frame number determines how many frames the system can address, and the number of bits in the offset determines the size of each frame.

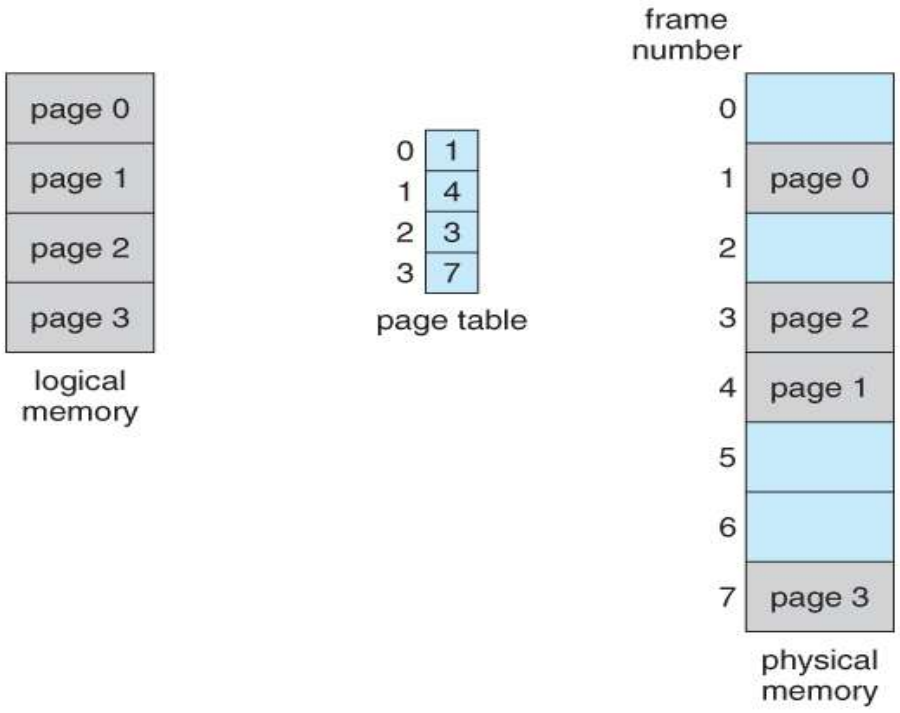


Figure D: Paging model of logical and physical memory

- Page numbers, frame numbers, and frame sizes are determined by the architecture, but are typically powers of two, allowing addresses to be split at a certain number of bits.

- For example, if the logical address size is 2^m and the page size is 2^n, then the high-order m-n bits of a logical address designate the page number and the remaining n bits represent the offset.

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

- **Consider an example**, consider the memory in Figure E. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages), we show how the user's view of memory can be mapped into physical memory.
- Logical address 0 is page 0, offset 0. 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]. Logical address 13 maps to physical address 9.

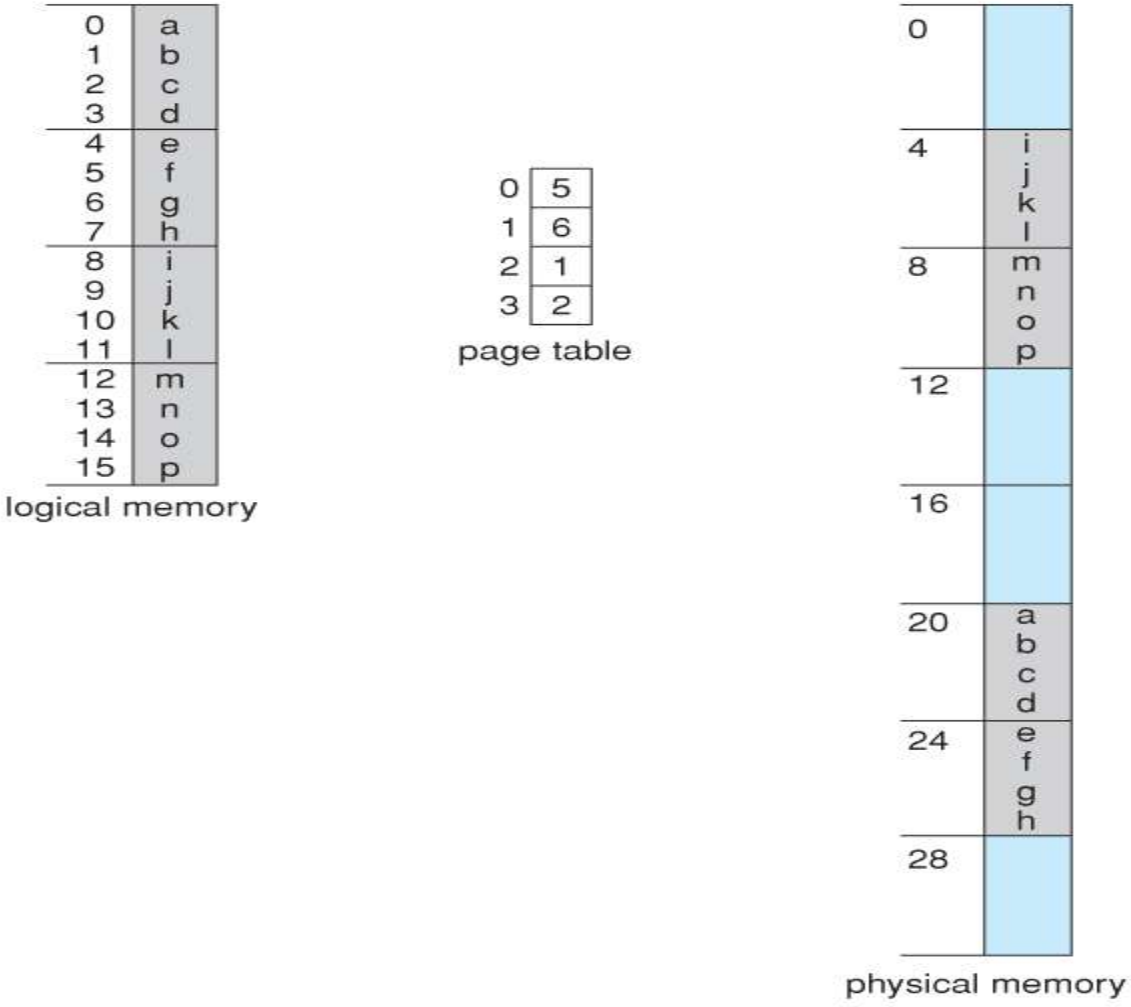


Figure E: Paging example for a 32-byte memory with 4-byte pages

- 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 **n** pages, at least **n** frames must be available in memory. If n 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, its frame number is put into the page table, and so on (Figure F).

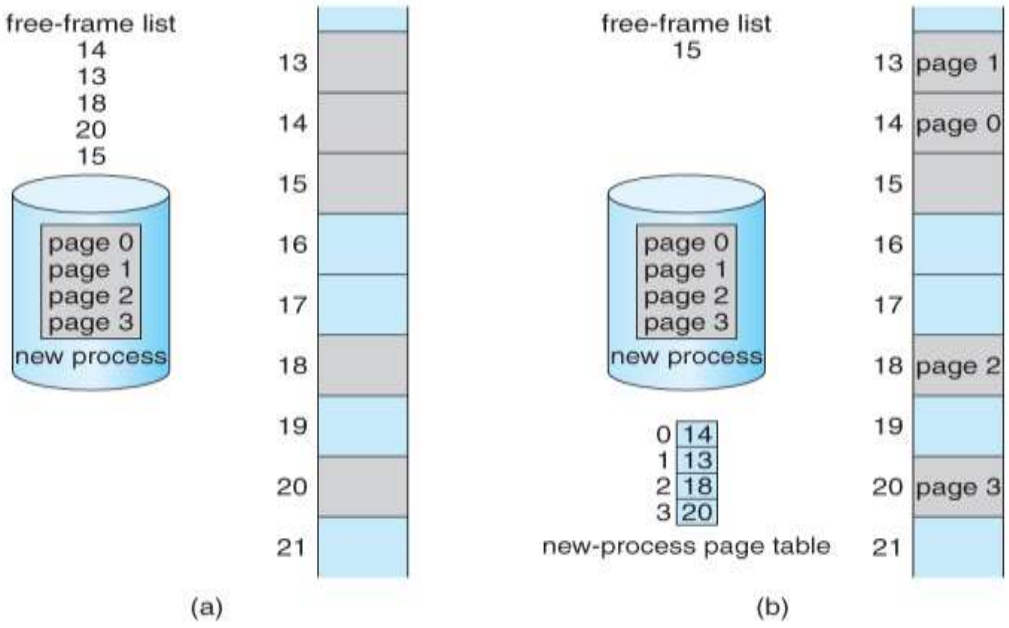


Figure F: Free frames (a) before allocation and (b) after allocation

- 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. This mapping is hidden from the user and is controlled by the operating system.

Protection:

- The page table can also help to protect processes from accessing memory.
- 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 (or valid) page.
- When the bit is set to "**in-valid,**" the page is not in the process's logical address space. Illegal addresses are trapped by use of the valid -invalid bit.
- The operating system sets this bit for each page to allow or disallow access to the page.

For example, that in a system with a 14-bit address space (0 to 16383), we have a program that should use only addresses 0 to 10468. Given a page size of 2 KB, we have the situation shown in Figure G.

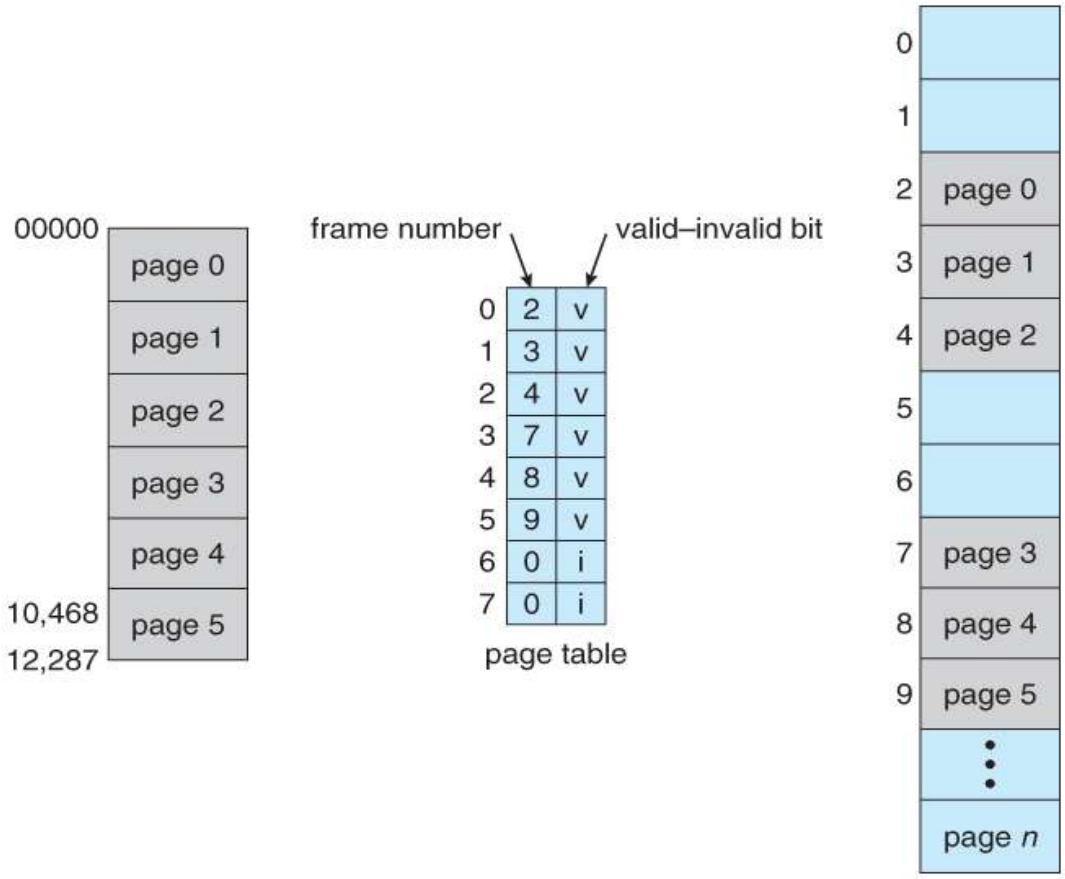


Figure G: Valid (v) or invalid (i) bit in a page table.

- Addresses in pages 0, 1, 2, 3, 4, and 5 are mapped normally through the page table. Any attempt to generate an address in pages 6 or 7, however, will find that the valid -invalid bit is set to invalid, and the computer will trap to flee operating system (invalid page reference).
- Notice that this scheme has created a problem. Because the program extends only to address 10468, any reference beyond that address is illegal. However, references to page 5 are classified as valid, so accesses to addresses up to 12287 are valid.
- Only the addresses from 12288 to 16383 are invalid. This problem is a result of the 2-KB page size and reflects the internal fragmentation of paging.
- Many processes do not use entire page table available to them, particularly in modern systems with very large potential page tables. Rather than waste memory by creating a full-size page table for every process, some systems use a page-table length register, PTLR, to specify the length of the 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 150 KB of code and 50 KB of data space, we need 8,000 KB to support the 40 users.
- If the code is **reentrant or pure code** (it does not write to or change the code in any way however, it can be shared, as shown in Figure H.
- Here we see a three-page editor-each page 50 KB in size being shared among three processes. Each process has its own data page.
- Reentrant code is non-self-modifying code: it never changes during execution. Thus, two or more processes can execute the same code at the same time.
- Each process has its own copy of registers and data storage to hold the data for the process's execution. The data for two different processes will, of course, be different.

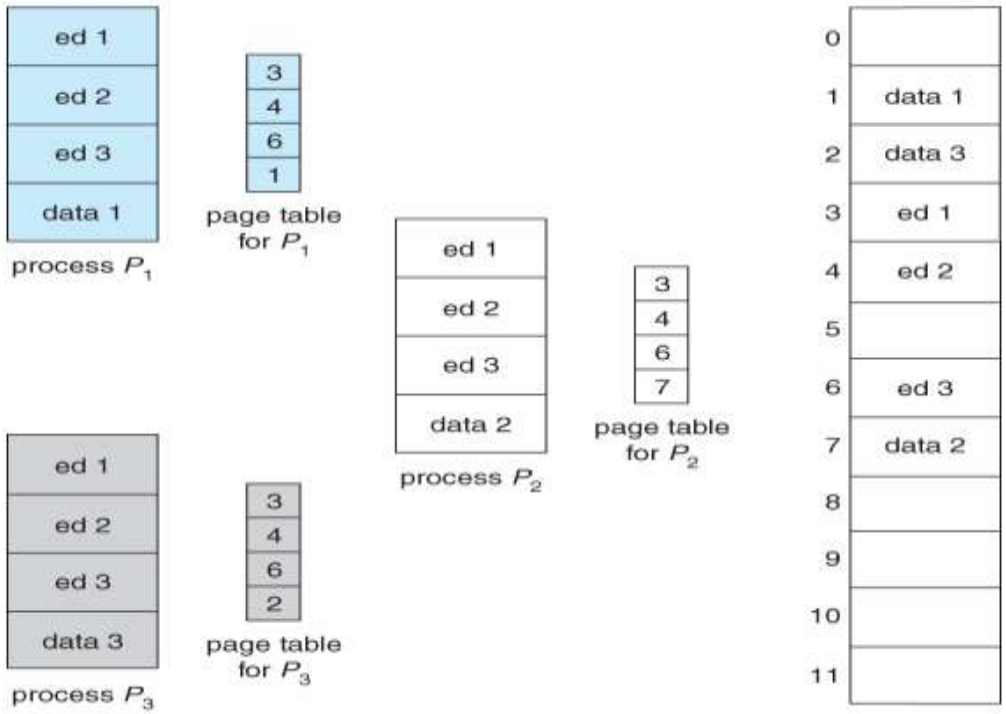


Figure H: Sharing of code in a paging environment.

- Only one copy of the editor need be kept in physical memory. Each user's page table maps onto the same physical copy of the editor, but data pages are mapped onto different frames.

- Thus, to support 40 users, we need only one copy of the editor (150 KB), plus 40 copies of the 50 KB of data space per user. The total space required is now 2,150 KB instead of 8,000 KB-a significant savings.
- Other heavily used programs can also be shared -compilers, window systems, run-time libraries, database systems, and so on. To be sharable, the code must be reentrant.

Structure of the Page Table

1. Hierarchical Paging:

- Most modern computer systems support a large logical address space (2^{32} to 2^{64}).
- For example, consider a system with a 32-bit logical address space. If the page size in such a system is 4 KB (2^{12}), then a page table may consist of up to 1 million entries (2^{32} /2^{12}).
- Assuming that each entry consists of 4 bytes, each process may need up to 4MB of physical address space for the page table alone.
- One simple solution to this problem is to divide the page table into smaller pieces. We can accomplish this division in several ways.
- **One way is to use a two-level paging algorithm**, in which the page table itself is also paged (Figure I).

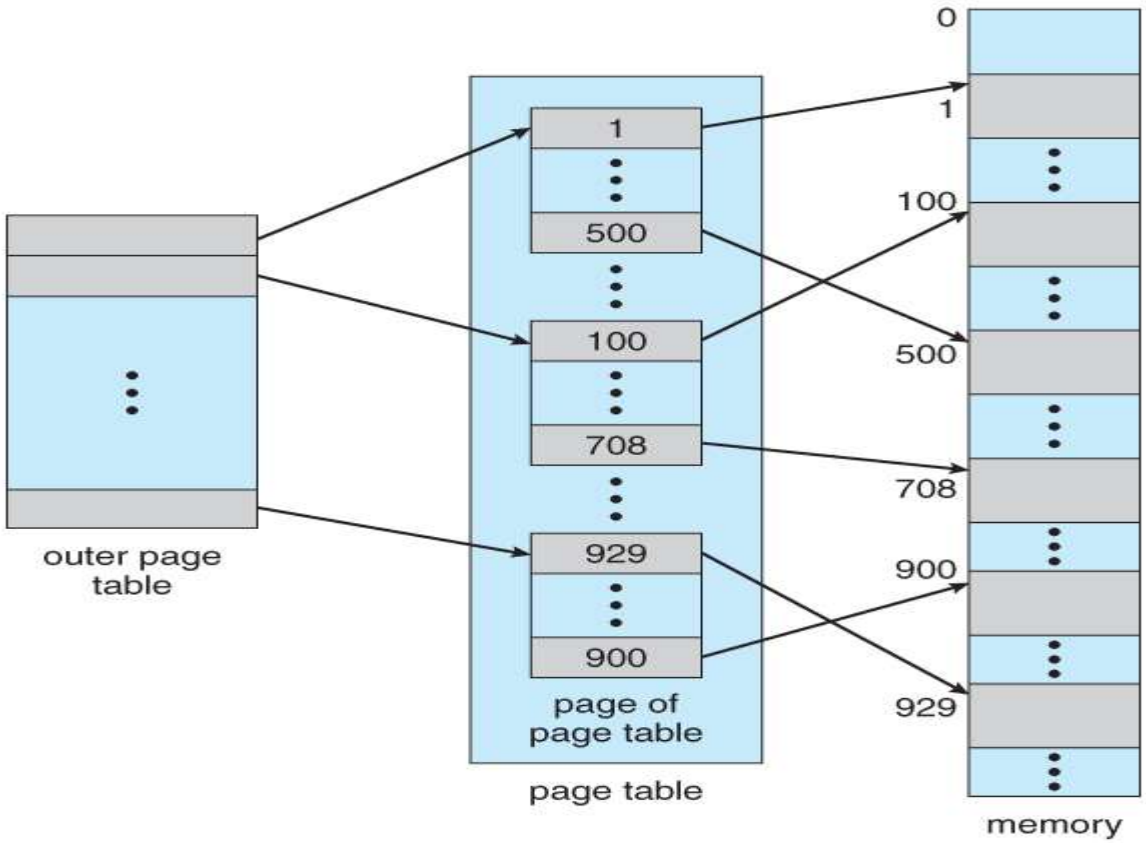
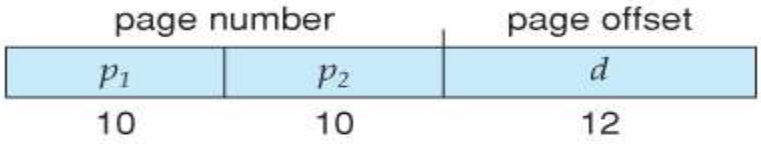


Figure I: A two-level page-table scheme.

- **For example,** consider again the system with a 32-bit logical address space and a page size of 4 KB. A logical address is divided into a page number consisting of 20 bits and a page offset consisting of 12 bits. Because we page the page table, 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:



- Where **P1** is an index into the outer page table and **P2** is the displacement within the page of the outer page table. The **address-translation method for this architecture** is shown in Figure J.

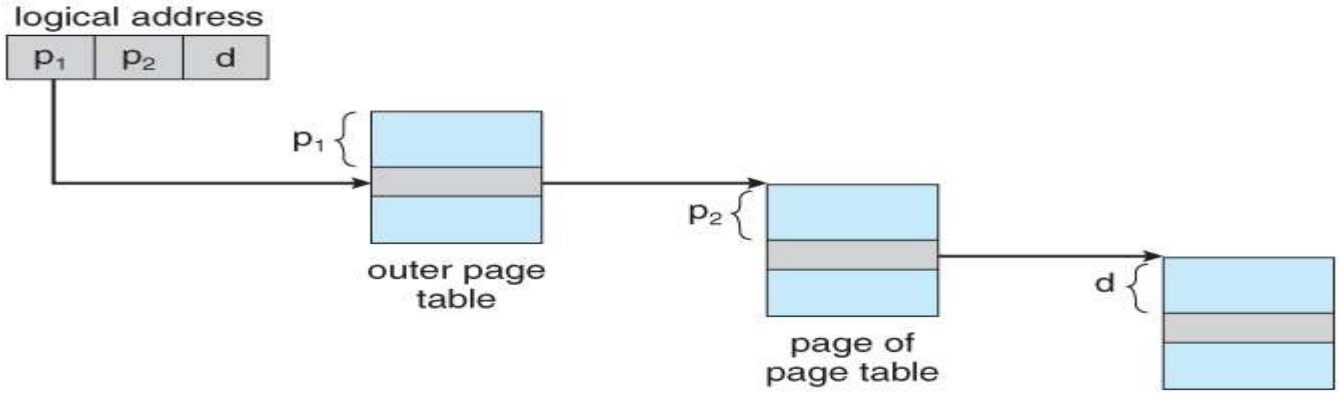
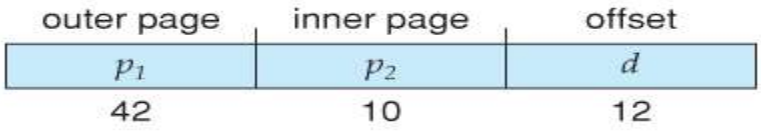


Figure J: Address translation for a two-level 32-bit paging architecture

- Because address translation works from the outer page table inward, this scheme is also known as a **forward-mapped page table.**

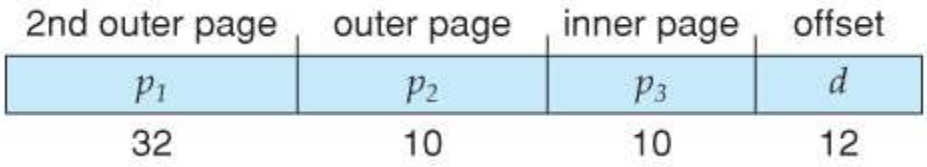
For a system with a 64-bit logical address space, a two-level paging scheme is no longer appropriate. To illustrate this point,------

- Let us suppose that the page size in such a system is 4 KB (2^{12}). In this case, the page table consists of up to 2^{52} entries. If we use a two-level paging scheme, then the inner page tables can conveniently be one page long, or contain 2^{10} 4-byte entries. The addresses look like this:



- The outer page table consists of 2^{42} entries, or 2^{44} bytes. The obvious way to avoid such a large table is to divide the outer page table into smaller pieces.

- We can divide the outer page table in various ways. We can page the outer page table, giving us a **three-level paging scheme**. Suppose that the outer page table is made up of standard-size pages (2^{10} entries, or 2^{12} bytes). In this case, a 64-bit address space is still daunting:



  - ❖ The outer page table is still 2^{32} bytes in size.

- The next step would be a four-level paging scheme, where the second-level outer page table itself is also paged, and so forth

2. Hashed Page Tables:

- ❖ A common approach for handling address spaces larger than 32 bits is to use a hashed page tables with the hash value being the virtual page number.
- ❖ Each entry in the hash table contains a linked list of elements that hash to the same location (to handle collisions). Each element consists of three fields:

  Field 1. The virtual page number

  Filed 2. The value of the mapped page frame

  Filed 3. A pointer to the next element in the linked list.

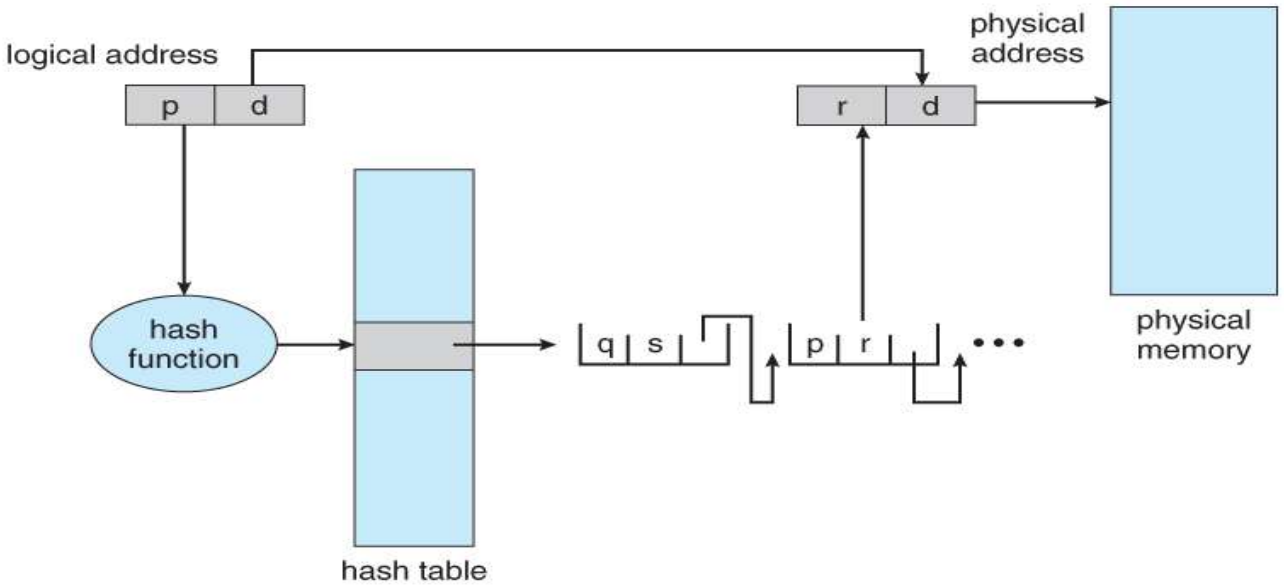


Figure K: Hashed page table

- The algorithm works as follows: The virtual page number in the virtual address is hashed into the hash table. The virtual page number is compared with field 1 in the first element in the linked list.
- If there is a match, the corresponding page frame (field 2) is used to form the desired physical address.
- If there is no match, subsequent entries in the linked list are searched for a matching virtual page number. This scheme is shown in Figure K.
- One of the drawbacks of this method is that each page table may consist of millions of entries. These tables may consume large amounts of physical memory just to keep track of how other physical memory is being used. To solve this problem, we can use an **Inverted Page Tables.**

3. Inverted Page Tables:

- An inverted page table has one entry for each real page (or frame) of memory.
- Each entry consists of the virtual address of the page stored in that real memory location; with information about the process that owns the page.
- Thus, only one page table is in the system, and it has only one entry for each page of physical memory. Figure L- shows the operation of an inverted page table.
- Access to an inverted page table can be slow, as it may be necessary to search the entire table in order to find the desired page (or to discover that it is not there.) Hashing the table can help speed up the search process.

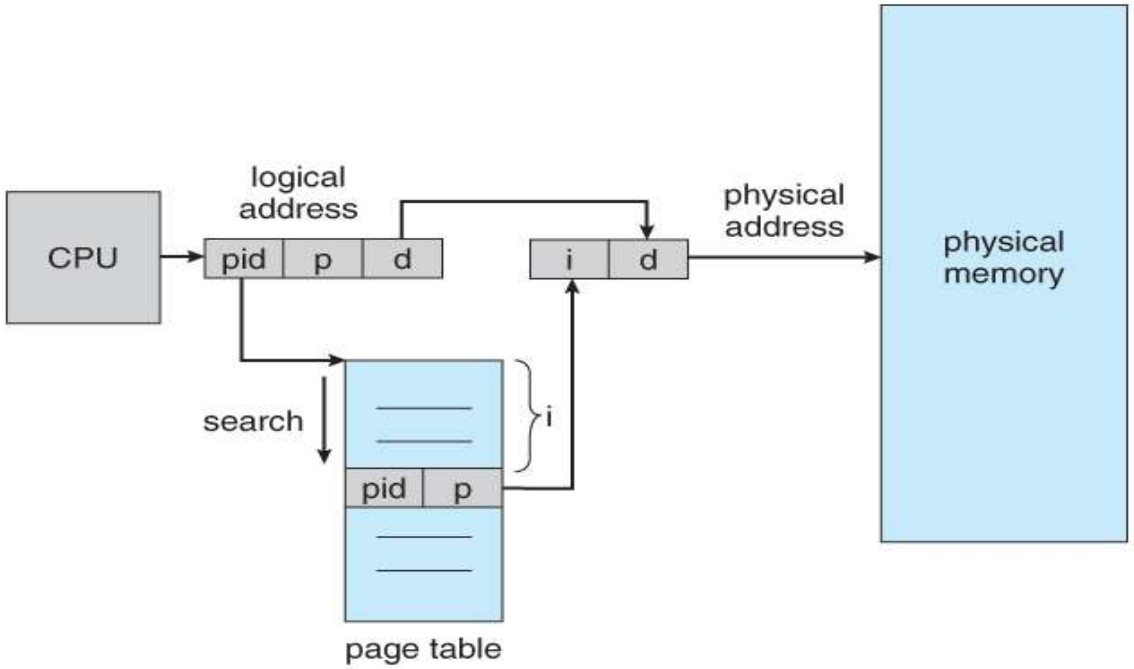


Figure L: Inverted page table

- Inverted page tables prohibit the normal method of implementing shared memory, which is to map multiple logical pages to a common physical frame. (Because each frame is now mapped to one and only one process.)

Advantages and Disadvantages of Paging:

Here is a list of advantages and disadvantages of paging −

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

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 a name and a length.
- The addresses specify both the segment name and the offset within the segment.
- The user therefore specifies each address by two quantities: a segment name and an offset.

For simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by a segment name. Thus, a logical address consists of a two tuple:

<Segment-number, offset>.

Segmentation hardware:

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

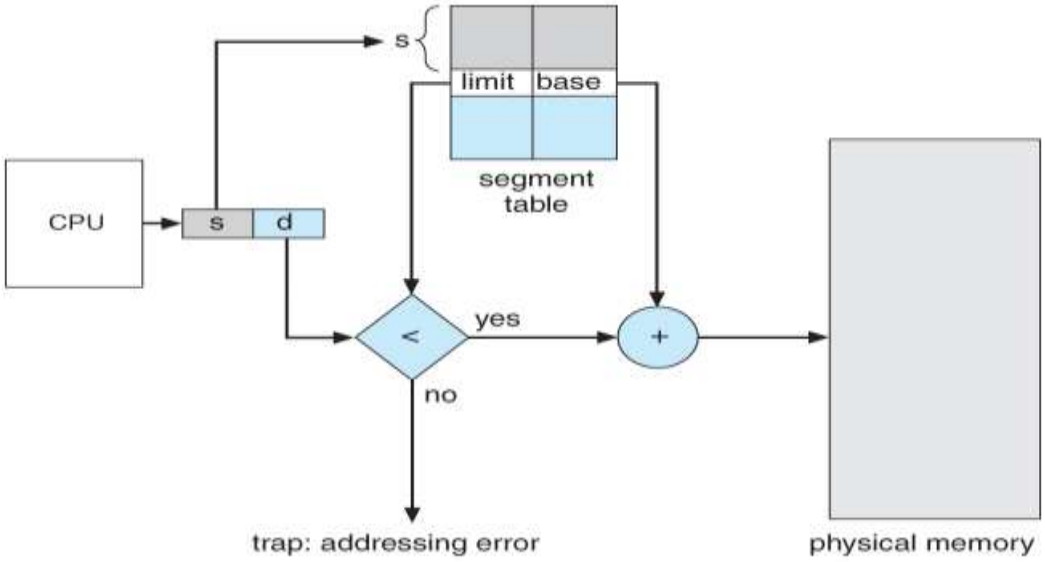


Figure M: Segmentation hardware

- The use of a segment table is illustrated in Figure -M.
  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 operating system (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.

Example:

Consider the situation shown in Figure -N. 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).

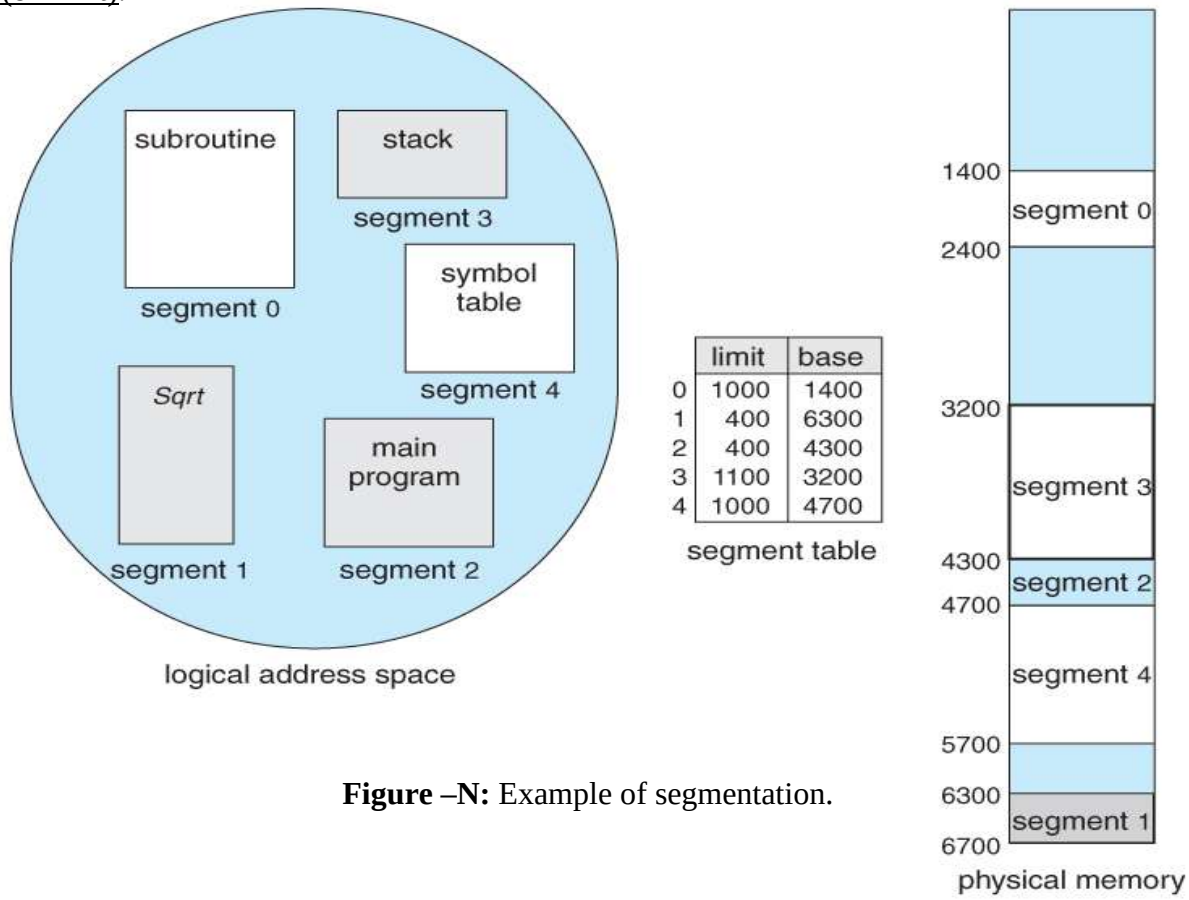


Figure –N: Example of segmentation.

- For example, segment 2 is 400 bytes long and begins at location 4300. Thus, a reference to byte 53 of segment 2 is mapped onto location 4300 +53= 4353.
- A reference to segment 3, byte 852, is mapped to 3200 (the base of segment 3) + 852 = 4052.
- A reference to byte 1222 of segment 0 would result in a trap to the operating system, as this segment is only 1,000 bytes long.

Intel Pentium Architecture:

- The Intel Pentium architecture, which supports both pure segmentation and segmentation with paging.
- In Pentium systems, the CPU generates logical addresses, which are given to the segmentation unit.
- The segmentation unit produces a linear address for each logical address. The linear address is then given to the paging unit.
- The paging unit generates the physical address in main memory.
- Thus, the segmentation and paging units form the equivalent of the memory-management unit (MMU).

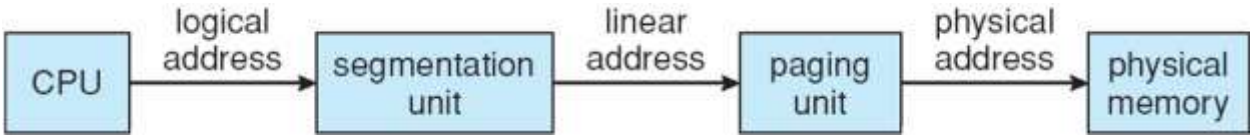


Figure –O: Logical to physical address translation in the Pentium.

Pentium Segmentation:

- The Pentium architecture allows a segment to be as large as 4 GB (2^{32}), and the maximum number of segments per process is 16K.
- The logical-address space of a process is divided into two partitions.
  - The first partition consists of up to 8 K segments those are private to that process.
  - The second partition consists of up to 8 K segments that are shared among all the processes.
- Information about the first partition is kept in the local description table (LDT).
- Information about the second partition is kept in the global description table (GDT).
- Each entry in the LDT and GDT consists of an 8-byte segment descriptor with detailed information about a particular segment, including the base location and limit of that segment.
- The logical address is a pair (selector, offset), where the selector is a 16-bit number:

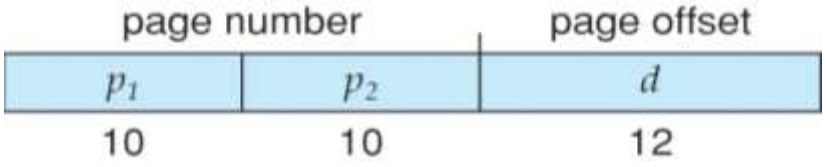
-- In which **s** designates the segment number, **g** indicates whether the segment is in the GDT or LDT, and **p** deals with protection.

Segmentation with paging:

Pentium with Paging:

In the following section, we discuss how the paging unit turns this linear address into a physical address.

- The Pentium architecture allows a page size of either 4 KB or 4 MB. For 4-KB pages, the Pentium uses a two-level paging scheme in which the division of the 32-bit linear address is as follows:



The address-translation scheme for this architecture is similar to the scheme shown in Figure - J.

- The Intel Pentium address translation is shown in more detail in Figure -Q.
- The 10 high-order bits reference an entry in the outer most page table, which the Pentium terms the **page directory.** (The CR3 register points to the page directory for the current process.)
- The page directory entry points to an inner page table that is indexed by the contents of the innermost 10 bits in the linear address.
- Finally, the low-order bits 0-11 refer to the offset in the 4-KB page pointed to in the page table.

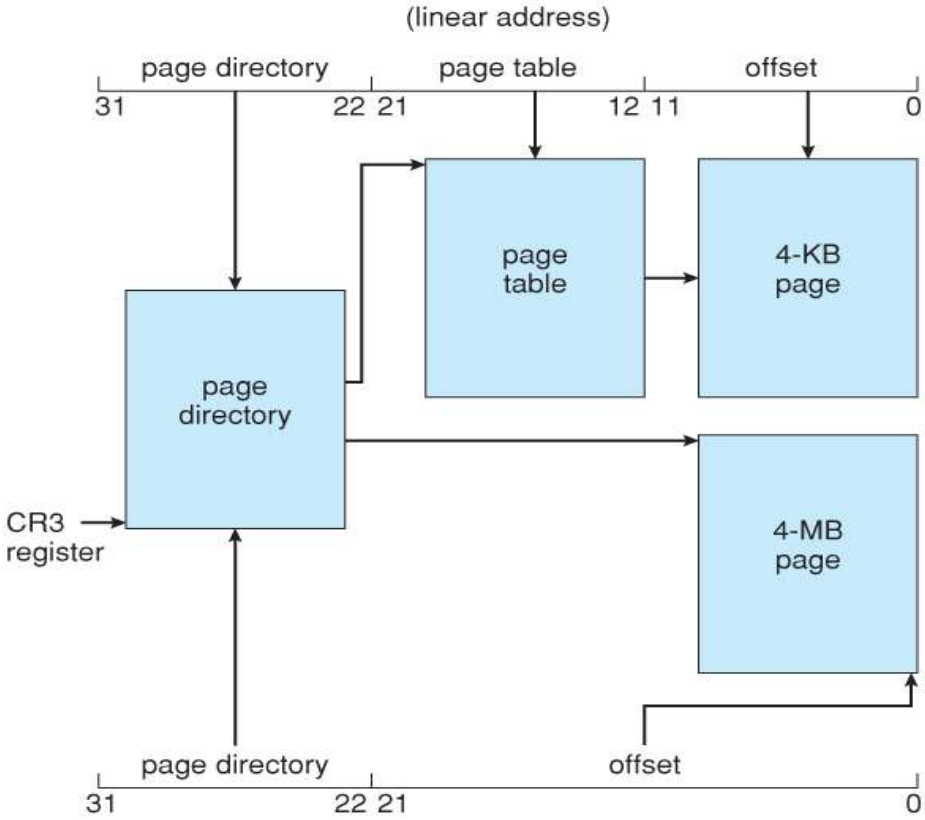


Figure –Q: Paging in the Pentium architecture.

- One entry in the page directory is the Page Size flag; which-if set indicates that the size of the page frame is 4 MB and not the standard 4 KB.
- If this flag is set, the page directory points directly to the 4-MB page frame, bypassing the inner page table; and the 22 low-order bits in the linear address refer to the offset in the 4-MB page frame.

To improve the efficiency of physical memory use, Intel Pentium page tables can be swapped to disk. In this case, an invalid bit is used in the page directory entry to indicate whether the table to which the entry is pointing is in memory or on disk. If the table is on disk, the operating system can use the other 31 bits to specify the disk location of the table; the table then can be brought into memory on demand.

Virtual Memory:

- Virtual memory is a technique that allows the execution of processes that are not completely in memory. One major advantage of this scheme is that programs can be larger than physical memory.
- Virtual memory serves two purposes. First, it allows us to extend the use of physical memory by using disk. Second, it allows us to have memory protection, because each virtual address is translated to a physical address.

Following are the situations, when entire program is not required to be loaded fully in main memory.

- User written error handling routines are used only when an error occurred in the data or computation.
- Certain options and features of a program may be used rarely.
- Many tables are assigned a fixed amount of address space even though only a small amount of the table is actually used.
- The ability to execute a program that is only partially in memory would counter many benefits.
- Less number of I/O would be needed to load or swap each user program into memory.
- A program would no longer be constrained by the amount of physical memory that is available.
- Each user program could take less physical memory, more programs could be run the same time, with a corresponding increase in CPU utilization and throughput.

- Virtual memory involves the separation of logical memory by users from physical memory. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available (Figure R).
- Virtual memory makes the task of programming much easier, because the programmer no longer needs to worry about the amount of physical memory available; she/he can concentrate instead on the problem to be programmed.
- The virtual address space of a process refers to the logical (or virtual) view of how a process is stored in memory. Typically, this view is that a process begins at a certain logical address-say, addresses 0- and exists in contiguous memory.
- Physical memory may be organized in page frames and that the physical page frames assigned to a process may not be contiguous. It is up to the memory management unit (MMU) to map logical pages to physical page frames in memory.

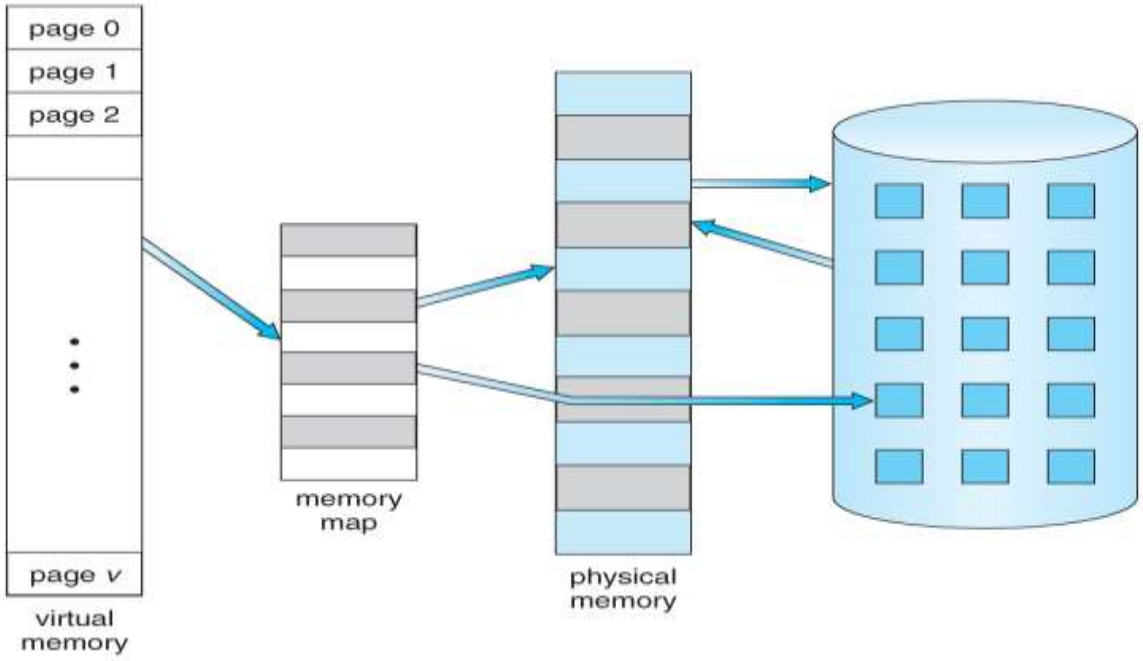


Figure –R: Diagram showing virtual memory that is larger than physical memory.

- Note in Figure - S that we allow for the heap to grow upward in memory as it is used for dynamic memory allocation. Similarly, we allow for the stack to grow downward in memory through successive function calls.
- The large blank space (or hole) between the heap and the stack is part of the virtual address space but will require actual physical pages only if the heap or stack grows. Virtual address spaces that include holes are known as sparse address spaces.
- Using a sparse address space is beneficial because the holes can be filled as the stack or heap segments grow or if we wish to dynamically link libraries (or possibly other shared objects) during program execution.

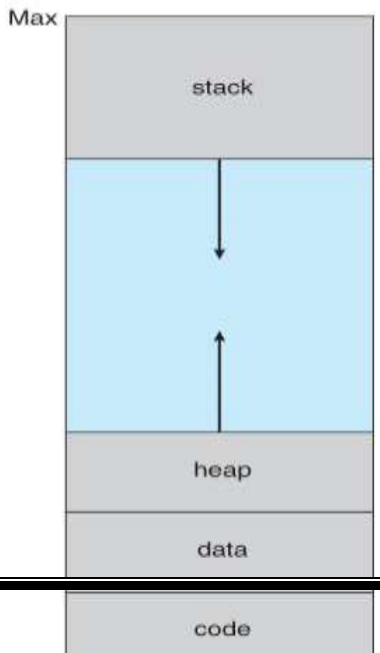


Figure -S: Virtual address space

Demand Paging:

The basic idea behind demand paging is that when a process is swapped in, its pages are not swapped in all at once. Rather they are swapped in only when the process needs them (On demand). This is termed as lazy swapper, although a pager is a more accurate term..

- Demand paging is commonly used in virtual memory systems.
- With demand-paged virtual memory, pages are **only loaded** when they are demanded during program execution
- Pages that are never accessed are thus never loaded into physical memory.

- A demand-paging system is similar to a paging system with swapping (Figure T) where processes reside in secondary memory (usually a disk).
- 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.
- Since we are now viewing a process as a sequence of pages, rather than as one large contiguous address space, use of the term swapper is technically incorrect.
- 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**.

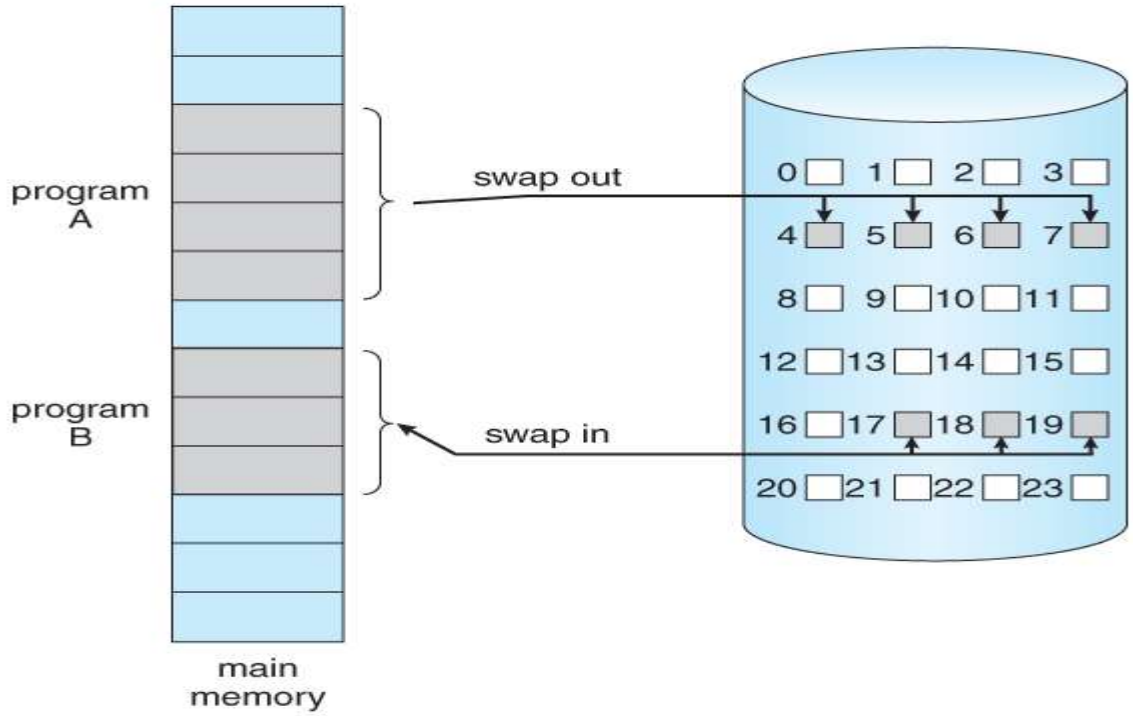


Figure T: Transfer of a paged memory to contiguous disk space

The Basic Concept:

When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again. Instead of swapping in a whole process, the pager brings only those pages into memory. Thus, it avoids reading into memory pages that will not be used anyway, decreasing the swap time and the amount of physical memory needed.

- With this scheme, we need some form of hardware support to distinguish between the pages that are in memory and the pages that are on the disk.
- The valid -invalid bit scheme shown in Figure-G 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.
- The page-table entry for a page that is brought into memory is set as usual but the page-table entry for a page that is not currently in memory is either simply marked invalid or contains the address of the page on disk. This situation is depicted in Figure U

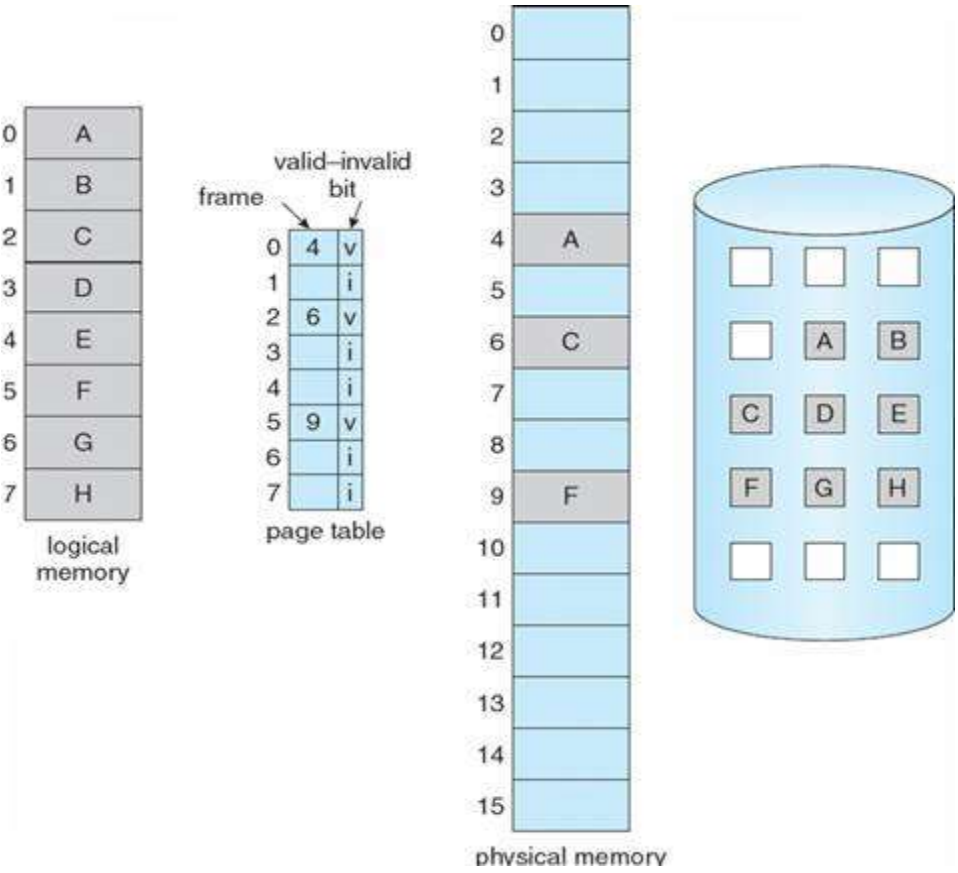


Figure U: Page table when some pages are not in main memory.

- Notice that marking a page invalid will have no effect if the process never attempts to access that page. Hence, if we guess right and page in all and only those pages that are actually needed, the process will run exactly as though we had brought in all pages.
- While the process executes and accesses pages that are memory resident, execution proceeds normally.

But what happens if the process tries to access a page that was not brought into memory?

- Access to a page marked invalid causes a page fault. The paging hardware, in translating the address through the page table, will notice that the invalid bit is set, causing a trap to the operating system.
- This trap is the result of the operating system's failure to bring the desired page into memory.
- The procedure for handling this page fault is straightforward (Figure V):

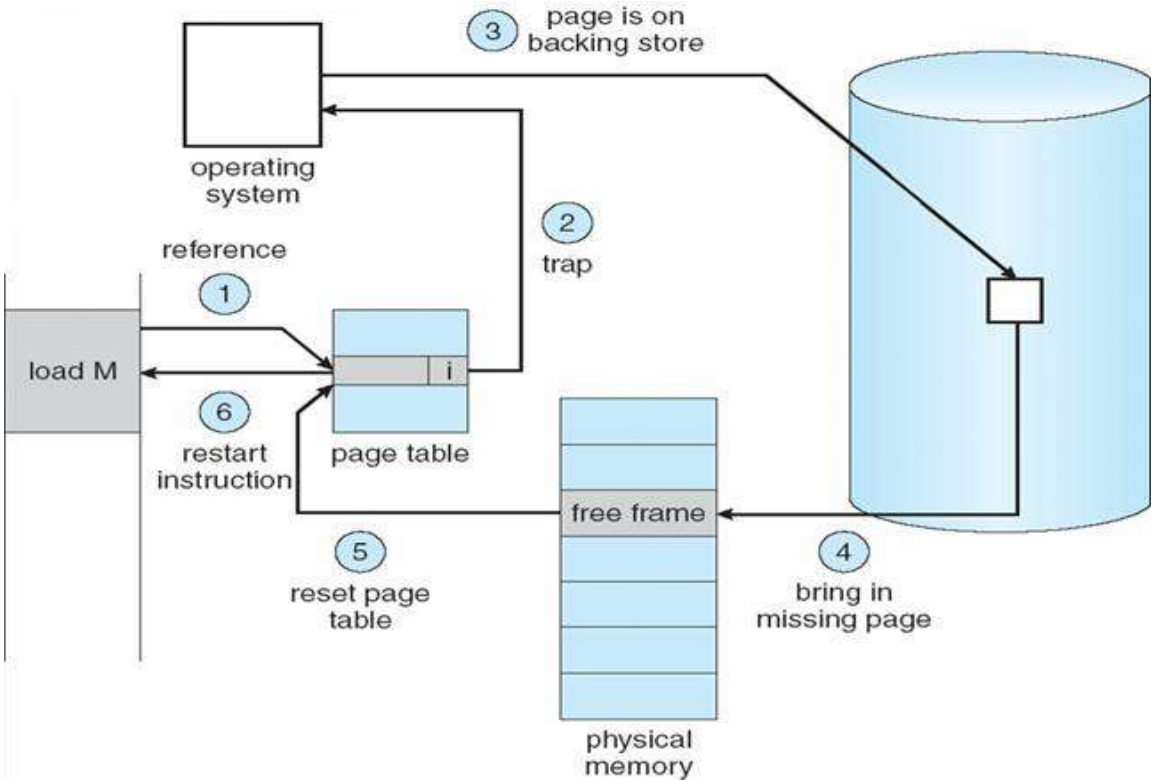


Figure V: Steps in handling a page fault.

1. We check an internal table (usually kept with the process control block) for this process to determine whether the reference was a valid or an invalid memory access.
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 (by taking one from the free-frame list, for example).
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 to indicate that the page is now in memory.
6. We restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory.

In the extreme case, we can start executing a process with **no** pages in memory. After the page is brought into memory, the process continues to execute. At that it can execute with no more faults. This scheme is **pure demand paging**: never bring a page into memory until it is required.

Page replacement:

- In order to make the most use of virtual memory, we load several processes into memory at the same time. Since we only load the pages that are actually needed by each process at any given time, there is room to load many more processes than if we had to load in the entire process.
- However memory is also needed for other purposes (such as I/O buffering), and what happens if some process suddenly decides it needs more pages and there aren't any free frames available? There are several possible solutions to consider:

1. Adjust the memory used by I/O buffering, etc., to free up some frames for user processes. The decision of how to allocate memory for I/O versus user processes is a complex one, yielding different policies on different systems. (Some allocate a fixed amount for I/O, and others let the I/O system contend for memory along with everything else.)
2. Put the process requesting more pages into a wait queue until some free frames become available.
3. Swap some process out of memory completely, freeing up its page frames.
4. Find some page in memory that isn't being used right now, and swap that page only out to disk, freeing up a frame that can be allocated to the process requesting it. **This is known as page replacement**, and is the most common solution..

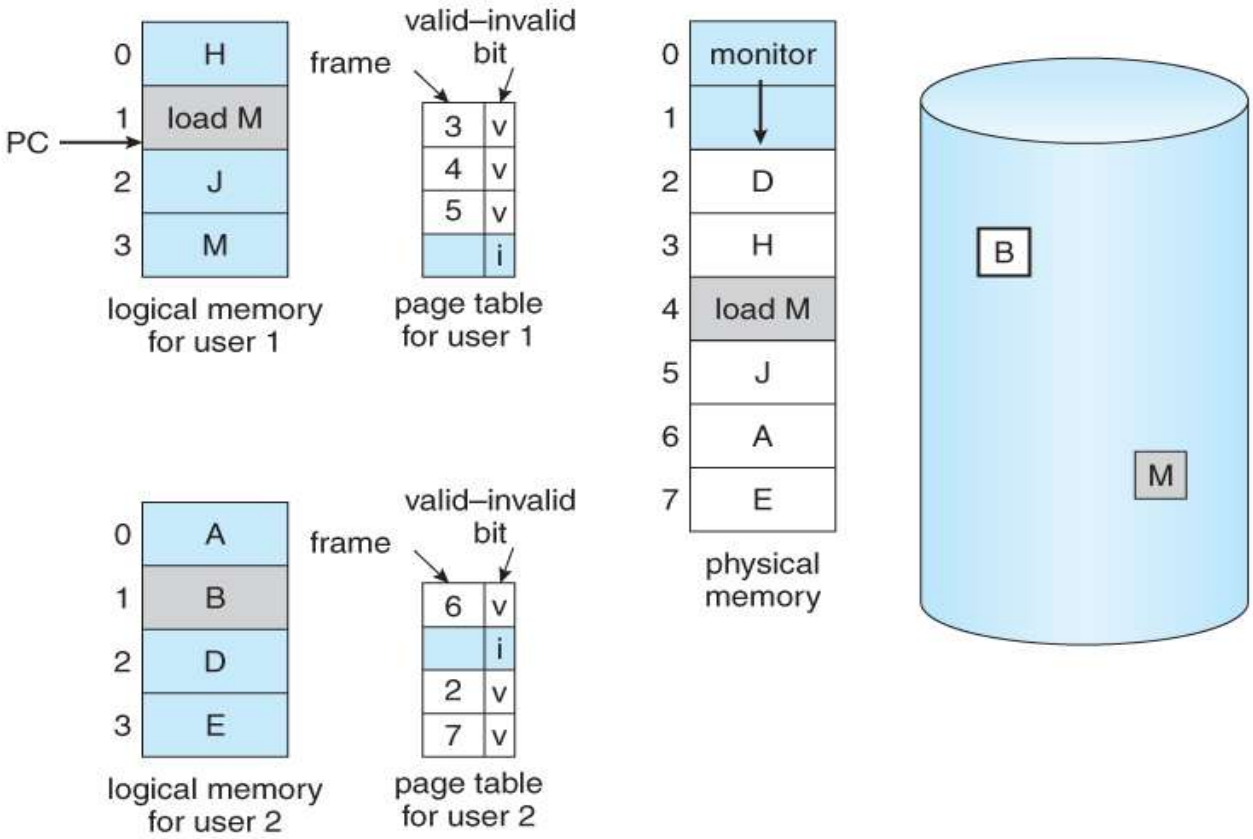


Figure W: Need for page replacement

Basic Page Replacement:

- Page replacement takes the following approach. If no frame is free, we find one that is not currently being used and free it. We can free a frame by writing its contents to swap space and changing the page table to indicate that the page is no longer in memory (Figure X).
- Now the page-fault handling must be modified to free up a frame if necessary, as follows:

1. Find the location of the desired page on the disk, either in swap space or in the file system.
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 an existing frame to be replaced, known as the **victim frame**.
    c. Write the victim frame to disk. Change all related page tables to indicate that this page is no longer in memory.
3. Read in the desired page and store it in the frame. Adjust all related page and frame tables to indicate the change.
4. Restart the process that was waiting for this page.

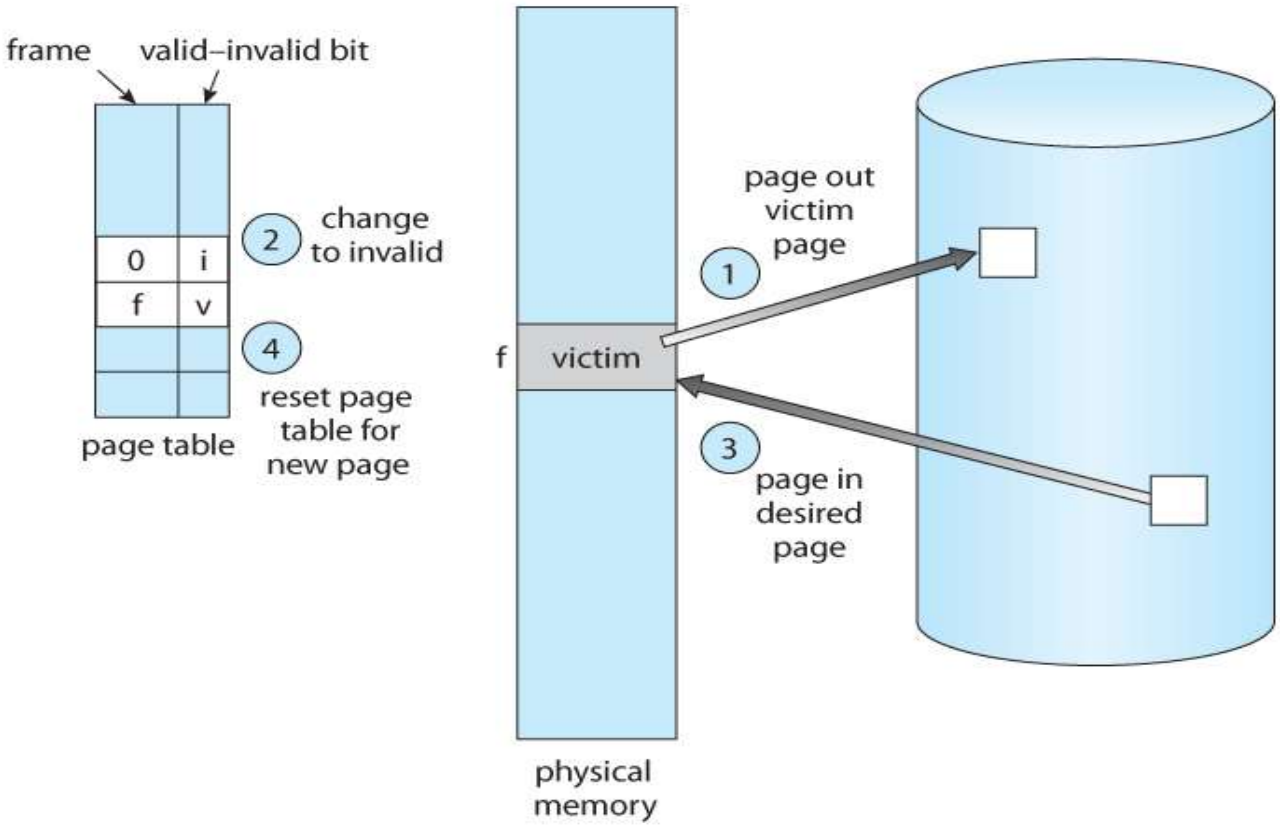


Figure X: Page replacement

- Notice that, if no frames are free, two page transfers (one out and one in) are required. It effectively doubles the page-fault service time and increases the effective access time accordingly.
- We can reduce this overhead by using a **modify bit (or dirty bit**). When this scheme is used, each page or frame has a modify bit associated with it in the hardware.
- The modify bit for a page is set, indicating that the page has been modified.
- When we select a page for replacement, we examine it's modify bit. If the **bit is set,** we know that the page has been modified since it was read in from the disk. In this case, we must write the page to the disk.
- If the modify **bit is not set**, however, the page has not been modified since it was read into memory. In this case, we need not write the memory page to the disk: it is already there.
- This technique also applies to read-only pages (for example, pages of binary code).
- Such pages cannot be modified; thus, they may be discarded when desired.
- This scheme can significantly reduce the time required to service a page fault, since it reduces I/O time by one-half if the page has not been modified.

❖ Page replacement is basic to demand paging. It completes the separation between logical memory and physical memory. With this mechanism, an enormous virtual memory can be provided for programmers on a smaller physical memory.

❖ We must solve two major problems to implement demand paging: we must develop **a frame –allocation algorithm** and a **page –replacement algorithm.**

❖ That is, if we have multiple processes in memory, we must decide how many frames to allocate to each process; and when page replacement is required,

❖ We must select the frames that are to be replaced. Designing appropriate algorithms to solve these problems is an important task, because disk I/0 is so expensive.

How do we select a particular replacement algorithm ?

➢ 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 a **reference string.**

➢ We can generate reference strings artificially or we can trace a given system and record the address of each memory reference.

➢ **First**, for a given page size (and the page size is generally fixed by the hardware or system), we need to consider only the page number, rather than the entire address.

➢ **Second,** if we have a reference to a page **p**, then any references to page **p** that immediately follow will never cause a page fault.

- Page **p** will be in memory after the first reference, so the immediately following references will not fault.

To determine the number of page faults for a particular reference string and page-replacement algorithm, we also need to know the number of page frames available. Obviously, as the **number of frames available increases, the number of page faults decreases.**

Page Replacement algorithms:

We illustrate several page-replacement algorithms. In doing so, we use the reference string

7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1

For a memory with **3** frames

1. FIFO Page Replacement:(First In First Out)

- 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.
- 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.

Example: Reference string 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1 with 3 frames

- 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 0. This process continues as shown in Figure Y. Every time a fault occurs, we show which pages are in our three frames. There are fifteen faults altogether.

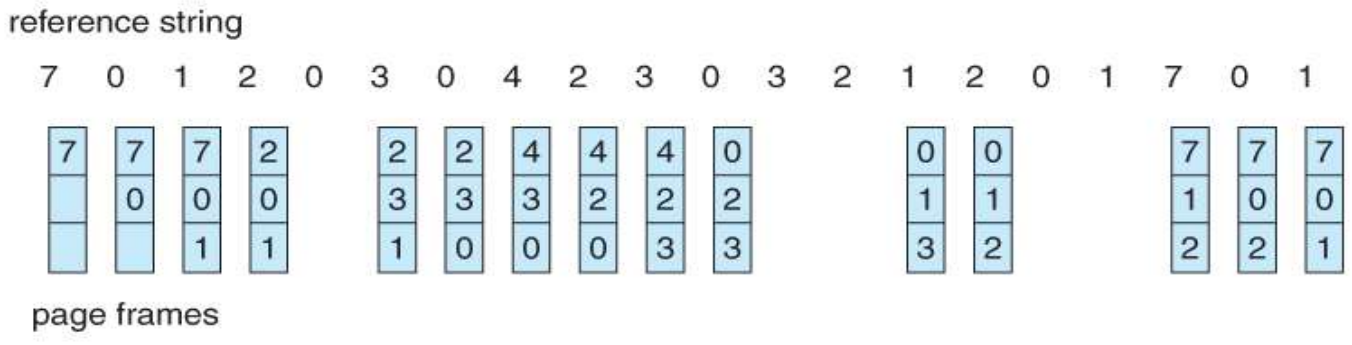


Figure Y: FIFO page-replacement algorithm.

- FIFO is simple and easy, it is not always optimal, or even efficient.
- We consider the following reference string: **1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**

  Figure -Z shows the curve of page faults for this reference string versus the number of available frames. Notice that the number of faults for four frames (10) is greater than the number of faults for three frames (9)! This most unexpected result is known as **Belady's anomaly:** for some page-replacement algorithms, the page-fault rate may increase as the number of allocated frames increases.

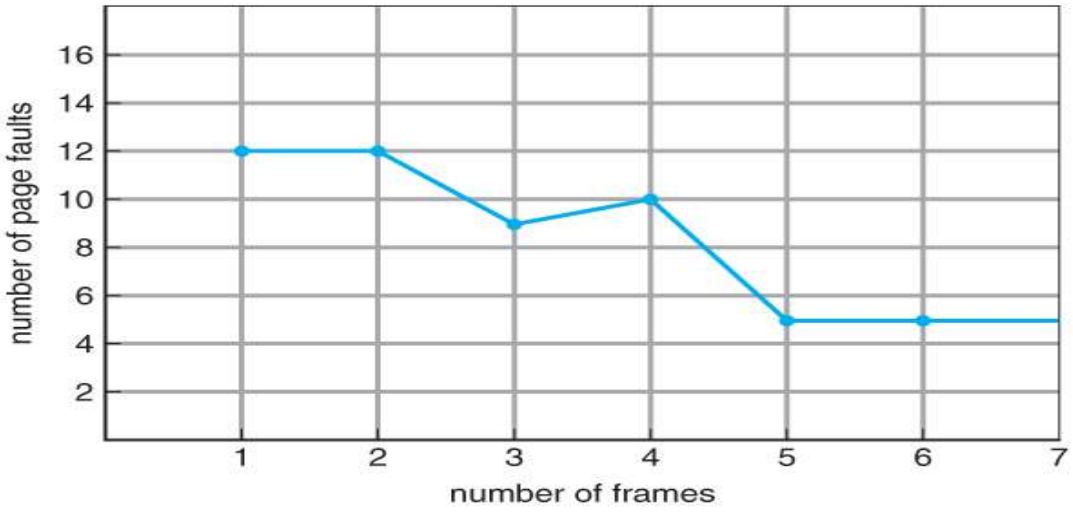


Figure Z: Page-fault curve for FIFO replacement on a reference string.

2. Optimal Page Replacement:

- The discovery of Belady's anomaly lead to the search for an **optimal page-replacement algorithm,** which has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly.
- Such an algorithm does exist and has been called OPT or MIN.
- It is simply this: Replace the page that will not be used for the longest period of time.

"Replace the page that will not be used for the longest time in the future."

- Use of this page-replacement algorithm guarantees the lowest possible page fault rate for a fixed number of frames.

Example: Reference string 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1 with 3 frames

- The optimal page-replacement algorithm would yield **nine page faults**, as shown in Figure AA.
- The first three references cause faults that fill the three empty frames.
- The reference to page 2 replaces page 7, because page 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 results in fifteen faults.

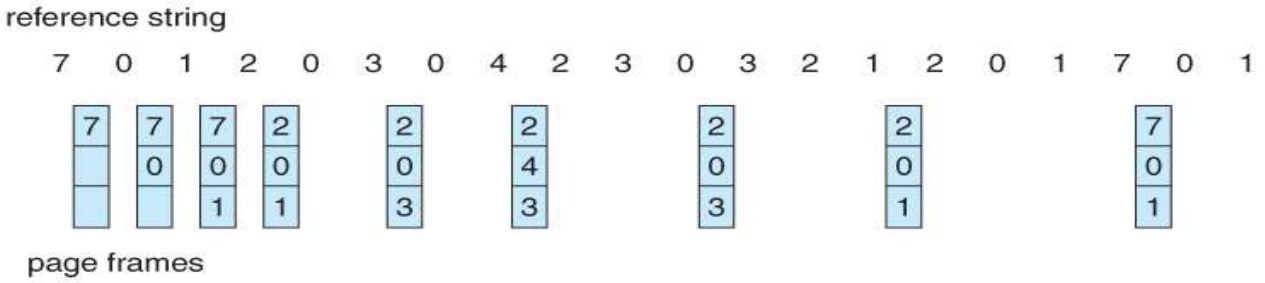


Figure AA: Optimal page-replacement algorithm.

- Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string.

3. LRU Page Replacement: (Least Recently Used)

- The prediction behind **LRU (**Least Recently Used**)** algorithm is **page that has not been used for the longest period of time.** This approach is the Least Recently used algorithm.

Example: Reference string 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1 with 3 frames

- LRU replacement to our example reference string is shown in Figure BB. The LRU algorithm produces **twelve page faults.**
- Notice that the first five 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.
- Despite these problems, LRU replacement with twelve faults is much better than FIFO replacement with fifteen.

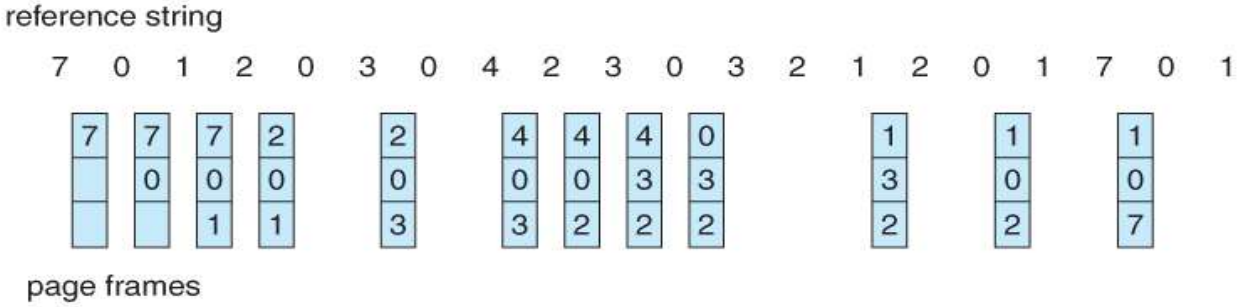


Figure BB : LRU page-replacement algorithm.

The LRU policy is often used as a page-replacement algorithm and is considered to be good.

The major problem is how to implement LRU replacement. Two implementations are feasible:

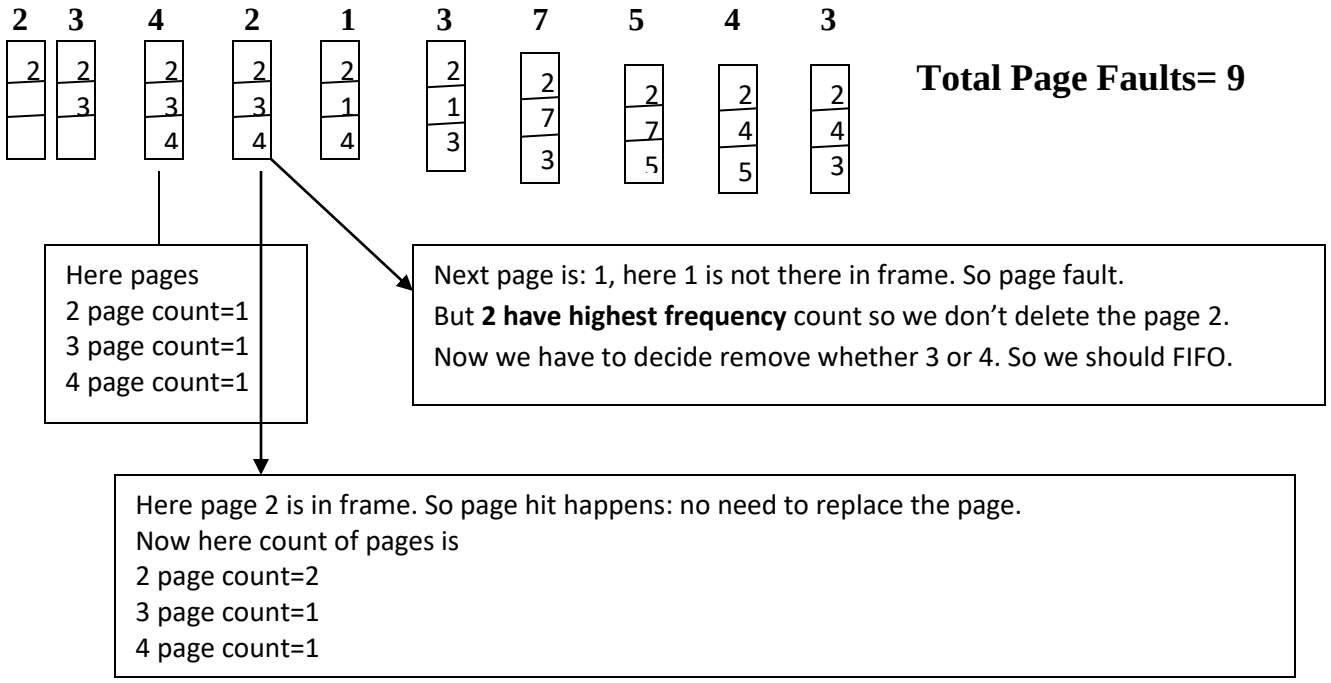
1. **Counters.** Every memory access increments a counter and the current value of this counter is stored in the page table entry for that page. Then finding the LRU page involves simple searching the table for the page with the smallest counter value. Note that overflowing of the counter must be considered.

2. **Stack.** Another approach is to use a stack and whenever a page is accessed, pull that page from the middle of the stack and place it on the top. The LRU page will always be at the bottom of the stack. Because this requires removing objects from the middle of the stack, a doubly linked list is the recommended data structure.

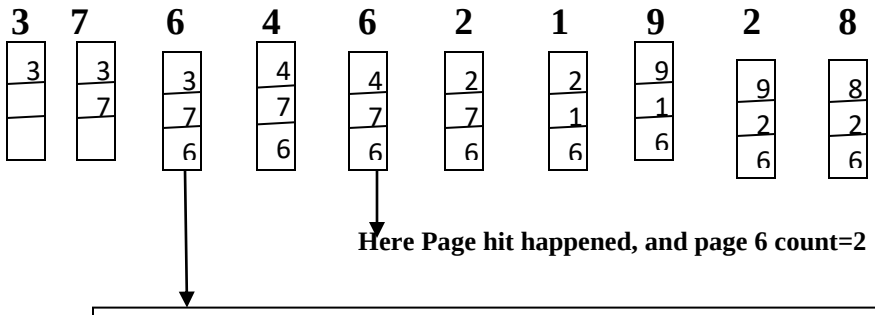
4. LFU Page Replacement :(Least Frequently Used)

- Replace the page with the smallest reference count.
- Any frequency-based policy requires a reference counting mechanism,
- **E.g.,** MMU increments a counter each time an in-memory page are referenced.
- Pure frequency-based policies have several potential drawbacks: – Old references are never forgotten. This can be addressed by periodically reducing the reference count of every in-memory page. – Freshly loaded pages have small reference counts and are likely victims -ignores temporal locality.

Example 1: Reference string 2, 3, 4, 2, 1, 3, 7, 5, 4, 3 **with 3 frames**



Example 2: Reference string 3, 7, 6, 4, 6, 2, 1, 9, 2, 8 **with 3 frames**

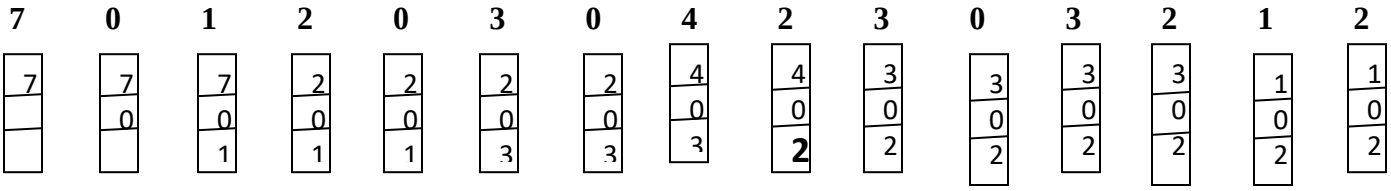


Total Page Faults= 9

Here pages in Frames 3,7,6
Pages count are 3 page=1
7 page=1
6 page=1

So, the next page is 4 is not there in Frame. We have to decide which page to be removed. Here all pages count is 1. So we should follow FIFO.

Example 3: Reference string 7, 0,1,2,0,3,0,4,2,3,0,3,2,1,2 **with 3 frames**



here page faults are: 9

If we observe last two columns that are: (1,0,2) and (1,0,2) but actually if you observe last third column (3,0,2). Here page count of 0 is higher compared to 2 and 3. And 2 and 3 page counts same. So we should follow FIFO to replace next page i.e. 1

So we have to delete 3 or 2. If we observe last 7th column (4,0,2), from 7th column 2 is remaining in frame only . Next 3 will come into the frame. If we follow FIFO 2 can be deleted in place of 3.

So last two columns would be (3, 0, 1) and (3, 0, 2)

Now in this case page faults are: 10