MEMORY MANAGEMENT

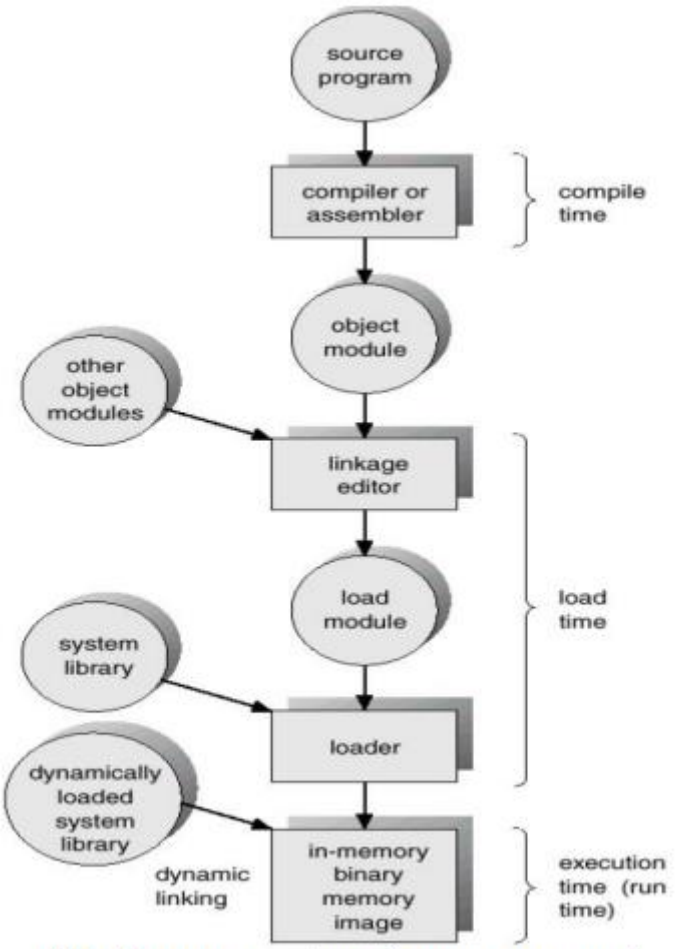
In a uni-programming system, main memory is divided into two parts: one part for the operating system (resident monitor, kernel) and one part for the user program currently being executed.

In a multiprogramming system, the "user" part of memory must be further subdivided to accommodate multiple processes. The task of subdivision is carried out dynamically by the operating system and is known as **memory management**.

Binding of Instructions and Data to Memory

Address binding of instructions and data to memory addresses can happen at three different stages.

1. **Compile time**: The compile time is the time taken to compile the program or source code. During compilation, if memory location known a priori, then it generates absolute codes.
2. **Load time**: It is the time taken to link all related program file and load into the main memory. It must generate relocatable code if memory location is not known at compile time.
3. **Execution time**: It is the time taken to execute the program in main memory by processor. Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., base and limit registers).



(Multistep processing of a user program.)

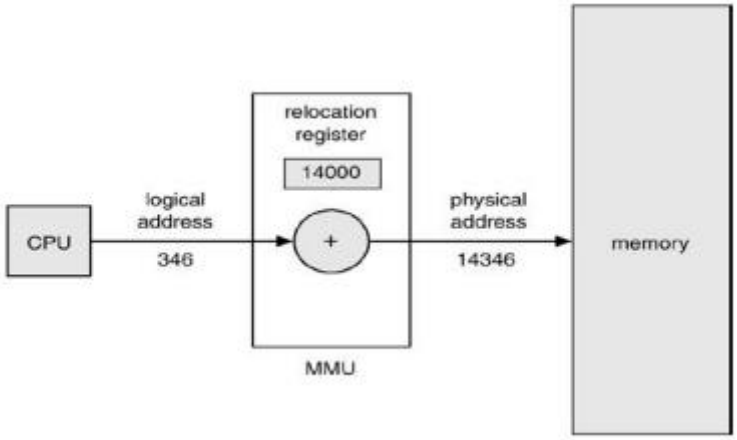
Logical- Versus Physical-Address Space

⇒ An address generated by the CPU is commonly referred to as a **logical address or** a **virtual address** whereas an address seen by the main memory unit is commonly referred to as a **physical address.**

⇒ The set of all logical addresses generated by a program is a **logical-address space whereas** the set of all physical addresses corresponding to these logical addresses is a **physical-address space.**

⇒ 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.

⇒ The Memory Management Unit is a hardware device that maps virtual 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 sent to memory as follows:



(Dynamic relocation using a relocation register)

Dynamic Loading

⇒ It loads the program and data dynamically into physical memory to obtain better memory-space utilization.

⇒ With dynamic loading, a routine is not loaded until it is called.

⇒ The advantage of dynamic loading is that an unused routine is never loaded.

⇒ This method is useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines.

⇒ Dynamic loading does not require special support from the operating system.

Dynamic Linking

⇒ Linking postponed until execution time.

⇒ Small piece of code (stub) used to locate the appropriate memory-resident library routine.

⇒ Stub replaces itself with the address of the routine and executes the routine.

⇒ Operating system needed to check if routine is in processes memory address.

⇒ Dynamic linking is particularly useful for libraries.

Overlays

⇒ Keep in memory only those instructions and data that are needed at any given time.

⇒ Needed when process is larger than amount of memory allocated to it.

⇒ Implemented by user, no special support needed from operating system, programming design of overlay structure is complex.

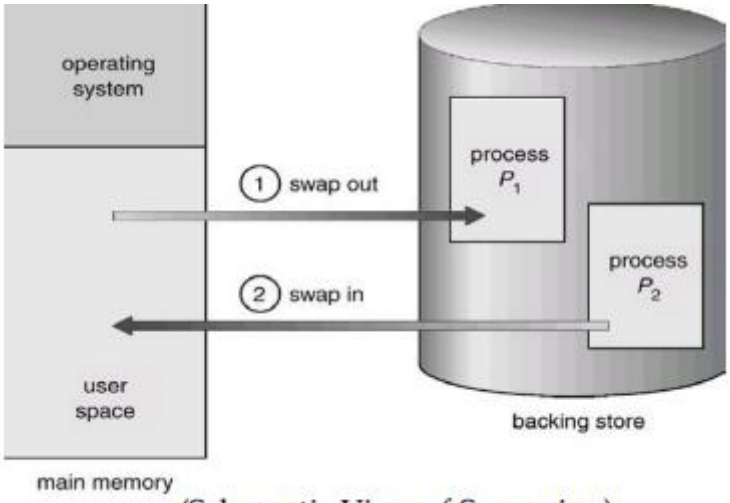
Swapping

⇒ A process can be swapped temporarily out of memory to a backing store (large disc), and then brought back into memory for continued execution.

⇒ **Roll out, roll in**: 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 so that it can 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 called **roll out, roll in.**

⇒ Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped.

⇒ Modified versions of swapping are found on many systems (UNIX, Linux, and Windows).



(Schematic View of Swapping)

MEMORY ALLOCATION

The main memory must accommodate both the operating system and the various user processes. We need to allocate different parts of the main memory in the most efficient way possible.

The main memory is usually divided into two partitions: one for the resident operating system, and one for the user processes. We may place the operating system in either low memory or high memory. The major factor affecting this decision is the location of the interrupt vector. Since the interrupt vector is often in low memory, programmers usually place the operating system in low memory as well.

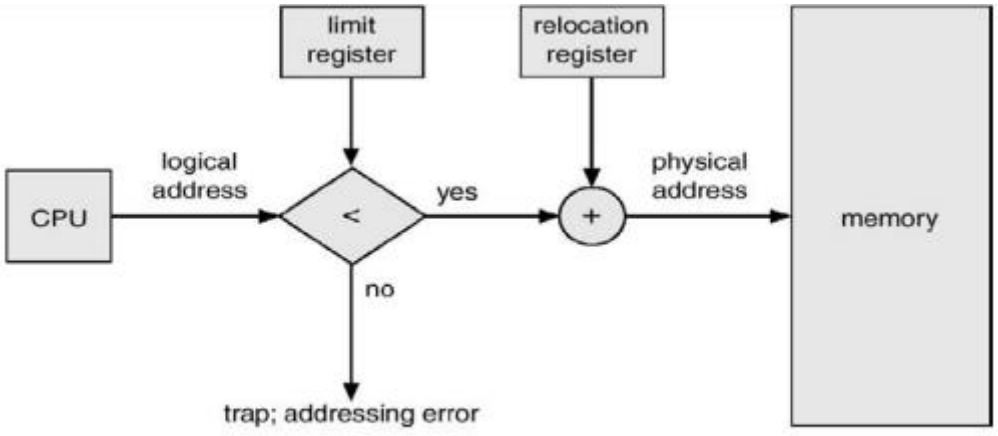
There are following two ways to allocate memory for user processes:

1. Contiguous memory allocation
2. Non contiguous memory allocation

1. Contiguous Memory Allocation

Here, all the processes are stored in contiguous memory locations. To load multiple processes into memory, the Operating System must divide memory into multiple partitions for those processes.

Hardware Support: The 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 of a partition and limit register contains range of that partition. Each logical address must be less than the limit register.



(Hardware support for relocation and limit registers)

According to size of partitions, the multiple partition schemes are divided into two types:

i. Multiple fixed partition/ multiprogramming with fixed task(MFT)
ii. Multiple variable partition/ multiprogramming with variable task(MVT)

i. Multiple fixed partitions: Main memory is divided into a number of static partitions at system generation time. In this case, any process whose size is less than or equal to the partition size can be loaded into any available partition. If all partitions are full and no process is in the Ready or Running state, the operating system can swap a process out of any of the partitions and load in another process, so that there is some work for the processor.

Advantages: Simple to implement and little operating system overhead.

Disadvantage:
* Inefficient use of memory due to internal fragmentation.
* Maximum number of active processes is fixed.

ii. Multiple variable partitions: With this partitioning, the partitions are of variable length and number. When a process is brought into main memory, it is allocated exactly as much memory as it requires and no more.

Advantages: No internal fragmentation and more efficient use of main memory.

Disadvantages: Inefficient use of processor due to the need for compaction to counter external fragmentation.

Partition Selection policy:

When the multiple memory holes (partitions) are large enough to contain a process, the operating system must use an algorithm to select in which hole the process will be loaded. The partition selection algorithm are as follows:

⇒ **First-fit**: The OS looks at all sections of free memory. The process is allocated to the first hole found that is big enough size than the size of process.
⇒ **Next Fit:** The next fit search starts at the last hole allocated and The process is allocated to the next hole found that is big enough size than the size of process.
⇒ **Best-fit**: The Best Fit searches the entire list of holes to find the smallest hole that is big enough size than the size of process.
⇒ **Worst-fit**: The Worst Fit searches the entire list of holes to find the largest hole that is big enough size than the size of process.

Fragmentation: The wasting of memory space is called fragmentation. There are two types of fragmentation as follows:

1. **External Fragmentation:** The total memory space exists to satisfy a request, but it is not contiguous. This wasted space not allocated to any partition is called external fragmentation. The external fragmentation can be reduce by compaction. The goal is to shuffle the memory contents to place all free memory together in one large block. Compaction is possible only if relocation is dynamic, and is done at execution time.
2. **Internal Fragmentation:** The allocated memory may be slightly larger than requested memory. The wasted space within a partition is called internal fragmentation. One method to reduce internal fragmentation is to use partitions of different size.

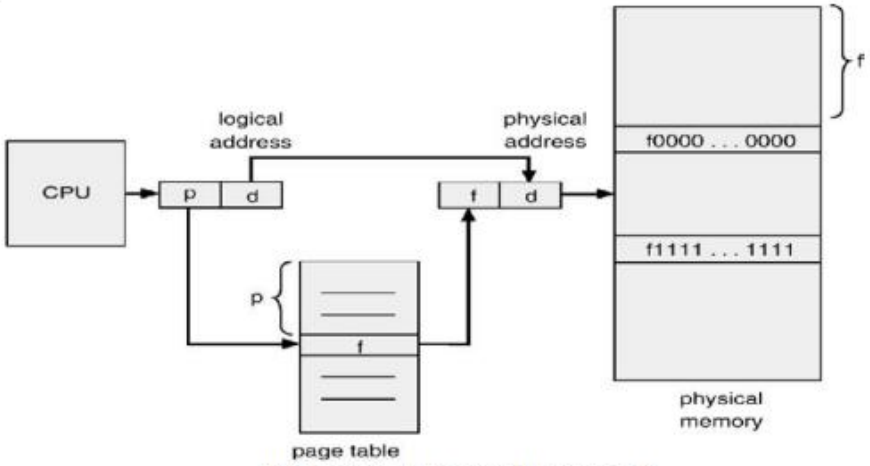
2. Noncontiguous memory allocation

In noncontiguous memory allocation, it is allowed to store the processes in non contiguous memory locations. There are different techniques used to load processes into memory, as follows:

1. Paging
2. Segmentation
3. Virtual memory paging(Demand paging) etc.

PAGING

Main memory is divided into a number of equal-size blocks, are called **frames**. Each process is divided into a number of equal-size block of the same length as frames, are called **Pages**. A process is loaded by loading all of its pages into available frames (may not be contiguous).



(Diagram of Paging hardware)

Process of Translation from logical to physical addresses

⇒ Every address generated by the CPU is divided into two parts: a page number **(p)** and a page offset **(d).** The page number is used as an index into a page table.

⇒ The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit.

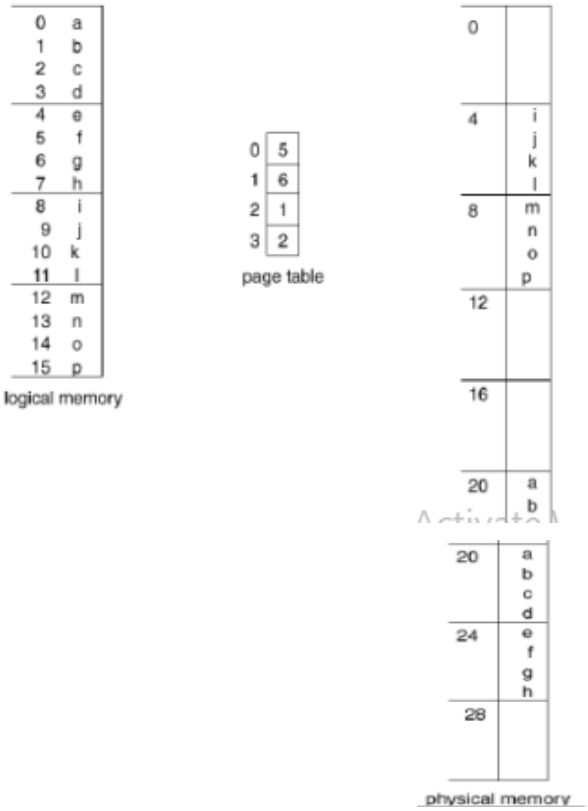
⇒ If the size of logical-address space is 2^m and a page size is 2^n addressing units (bytes or words), then the high-order (m – n) bits of a logical address designate the page number and the n low-order bits designate the page offset. Thus, the logical address is as follows:

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

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

Example:

Consider 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(= (2 x 4)+1).



Hardware Support for Paging:

Each operating system has its own methods for storing page tables. Most operating systems allocate a page table for each process. A pointer to the page table is stored with the other register values (like the instruction counter) in the process control block. When the dispatcher is told to start a process, it must reload the user registers and define the correct hardware page table values from the stored user page table.

Implementation of Page Table

⇒ Generally, Page table is kept in main memory. The Page Table Base Register (PTBR) points to the page table. And Page-table length register (PRLR) indicates size of the page table.

⇒ In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.

⇒ The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers (TLBs)**.

Paging Hardware With TLB

The TLB is an associative and high-speed memory. Each entry in the TLB consists of two parts: a key (or tag) and a value. The TLB is used with page tables in the following way.

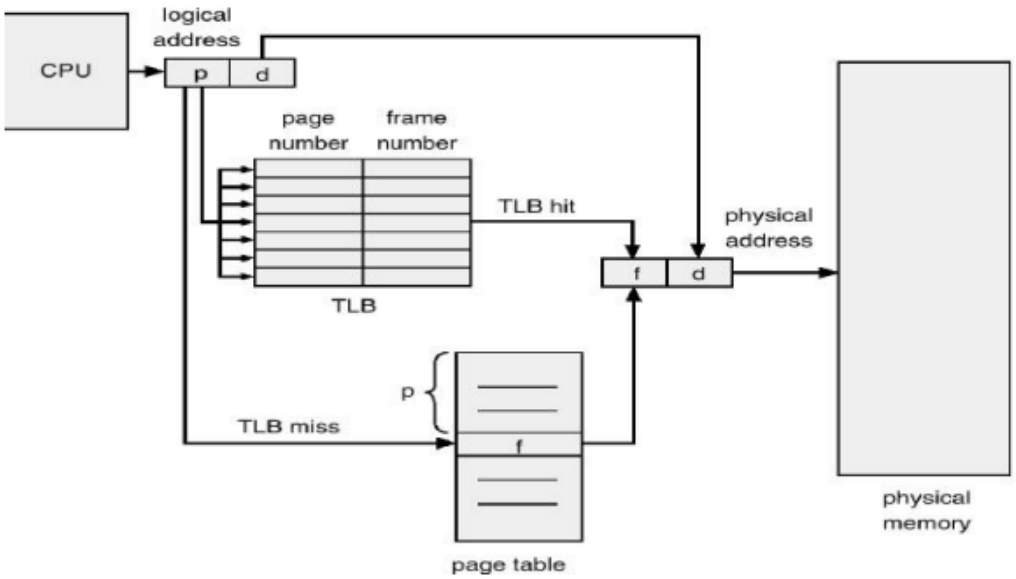
⇒ The TLB contains only a few of the page-table entries. When a logical address is generated by the CPU, its page number is presented to the TLB.

⇒ If the page number is found (known as a **TLB Hit**), its frame number is immediately available and is used to access memory. It takes only one memory access.

⇒ If the page number is not in the TLB (known 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. It takes two memory accesses.

⇒ In addition, it stores the page number and frame number to the TLB, so that they will be found quickly on the next reference.

⇒ If the TLB is already full of entries, the operating system must select one for replacement by using replacement algorithm.



(Paging hardware with TLB)

The percentage of times that a particular page number is found in the TLB is called the **hit ratio.** The effective access time (EAT) is obtained as follows:

EAT= HR x (TLBAT + MAT) + MR x (TLBAT + 2 x MAT)

Where HR: Hit Ratio, TLBAT: TLB access time, MAT: Memory access time, MR: Miss Ratio.

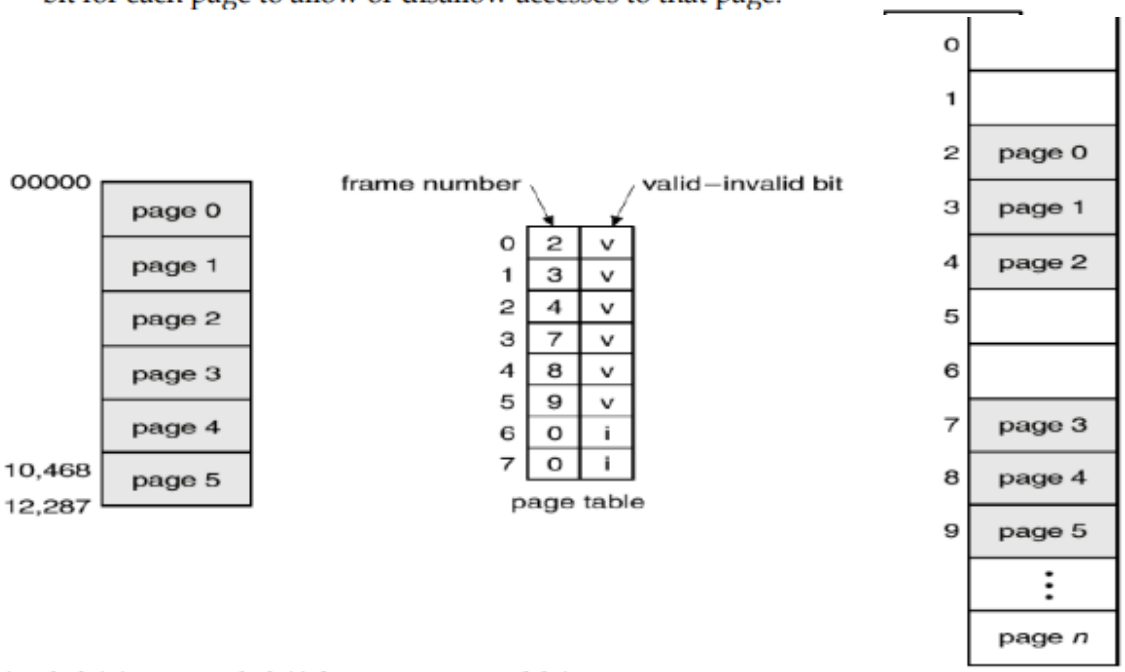
Memory protection in Paged Environment:

⇒ Memory protection in a paged environment is accomplished by protection bits that are associated with each frame. These bits are kept in the page table.

⇒ One bit can define a page to be read-write or read-only. This protection bit can be checked to verify that no writes are being made to a read-only page. An attempt to write to a read-only page causes a hardware trap to the operating system (or memory-protection violation).

⇒ One more bit is attached to each entry in the page table: a **valid-invalid** bit. When this bit is set to "valid," this value indicates that the associated page is in the process' logical-address space, and is a legal (or valid) page. If the bit is set to "invalid," this value indicates that the page is not in the process' logical-address space.

⇒ Illegal addresses are trapped by using the valid-invalid bit. The operating system sets this bit for each page to allow or disallow accesses to that page.

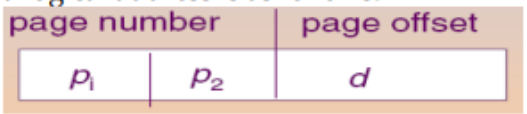


(Valid (v) or invalid (i) bit in a page table)

Structure of the Page Table

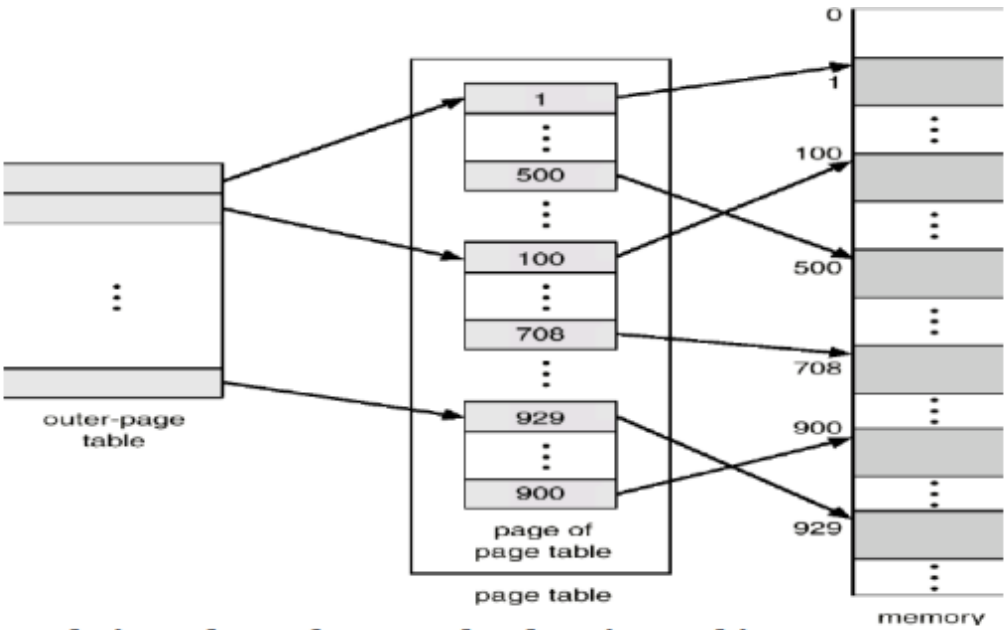
There are different structures of page table described as follows:

1. Hierarchical Page table: When the number of pages is very high, then the page table takes large amount of memory space. In such cases, we use multilevel paging scheme for reducing size of page table. A simple technique is a two-level page table. Since the page table is paged, the page number is further divided into parts: page number and page offset. Thus, a logical address is as follows:



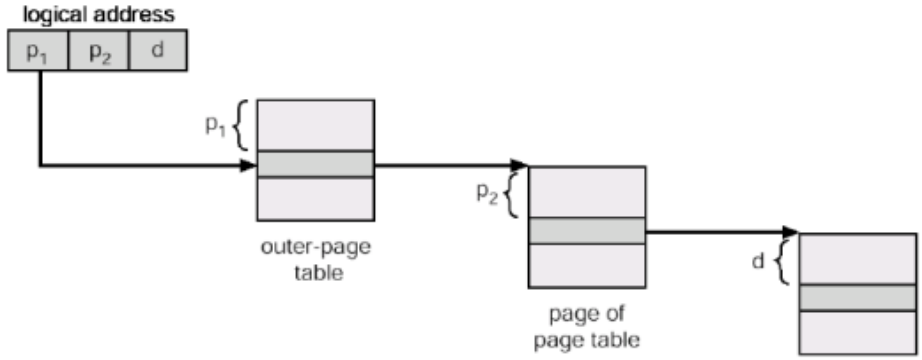
Where p_i is an index into the outer page table, and p2 is the displacement within the page of the outer page table.

Two-Level Page-Table Scheme:

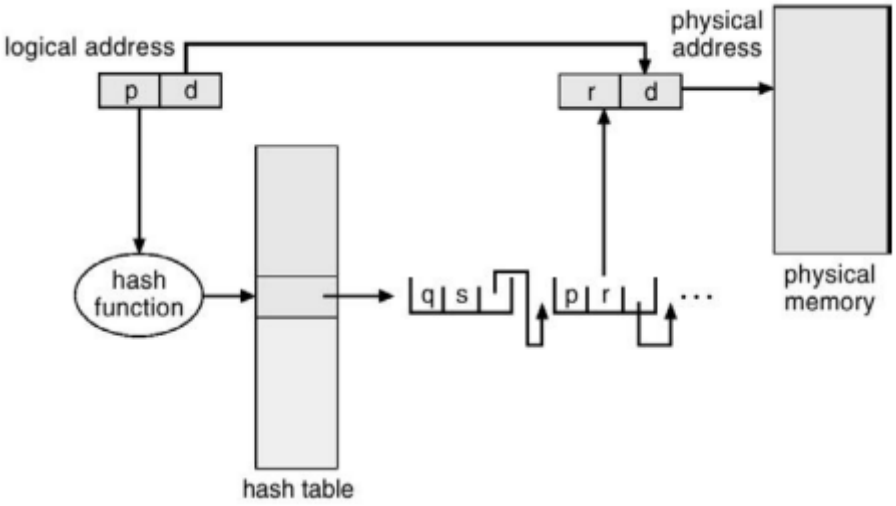


Address translation scheme for a two-level paging architecture:

Address translation scheme for a two-level paging architecture:



2. Hashed Page Tables: This scheme is applicable for address space larger than 32bits. In this scheme, the virtual page number is hashed into a page table. This page table contains a chain of elements hashing to the same location. Virtual page numbers are compared in this chain searching for a match. If a match is found, the corresponding physical frame is extracted.

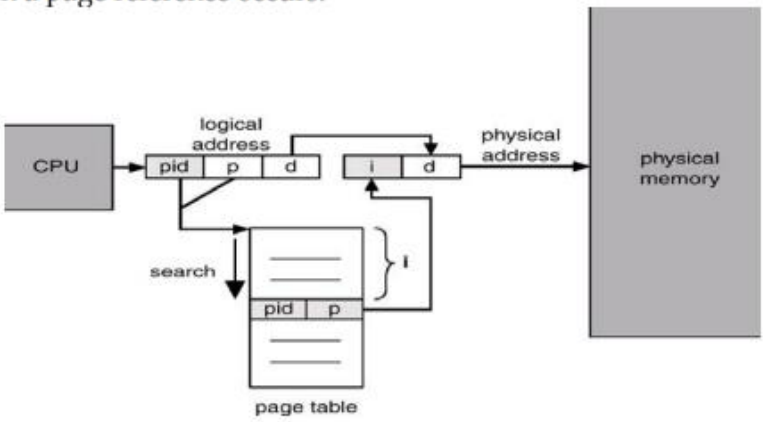


3. Inverted Page Table:

⇒ One entry for each real page of memory.

⇒ Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page.

⇒ Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs.



Shared Pages

Shared code

⇒ One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).

⇒ Shared code must appear in same location in the logical address space of all processes.

Private code and data

⇒ Each process keeps a separate copy of the code and data.

⇒ The pages for the private code and data can appear anywhere in the logical address space.

SEGMENTATION

Segmentation is a memory-management scheme that supports user view of memory. A program is a collection of segments. A segment is a logical unit such as: main program, procedure, function, method, object, local variables, global variables, common block, stack, symbol table, arrays etc.

A logical-address space is a collection of segments. Each segment has a name and a length. The user specifies each address by two quantities: a segment name/number and an offset.

Hence, Logical address consists of a two tuple: <segment-number, offset>

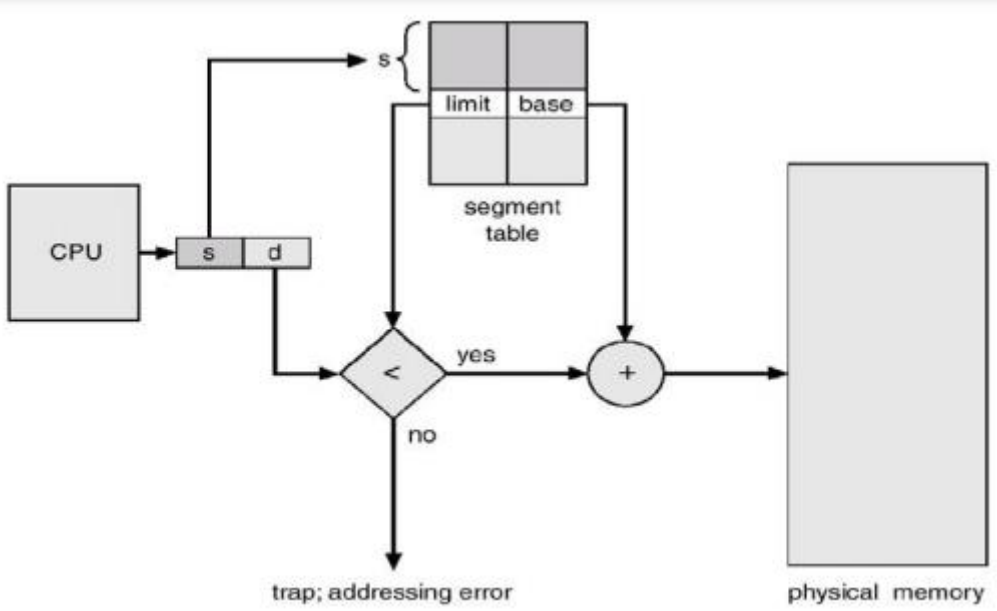
Segment table maps two-dimensional physical addresses and each entry in table has:

base – contains the starting physical address where the segments reside in memory.

limit – specifies the length of the segment.

Segment-table base register (STBR) points to the segment table's location in memory.

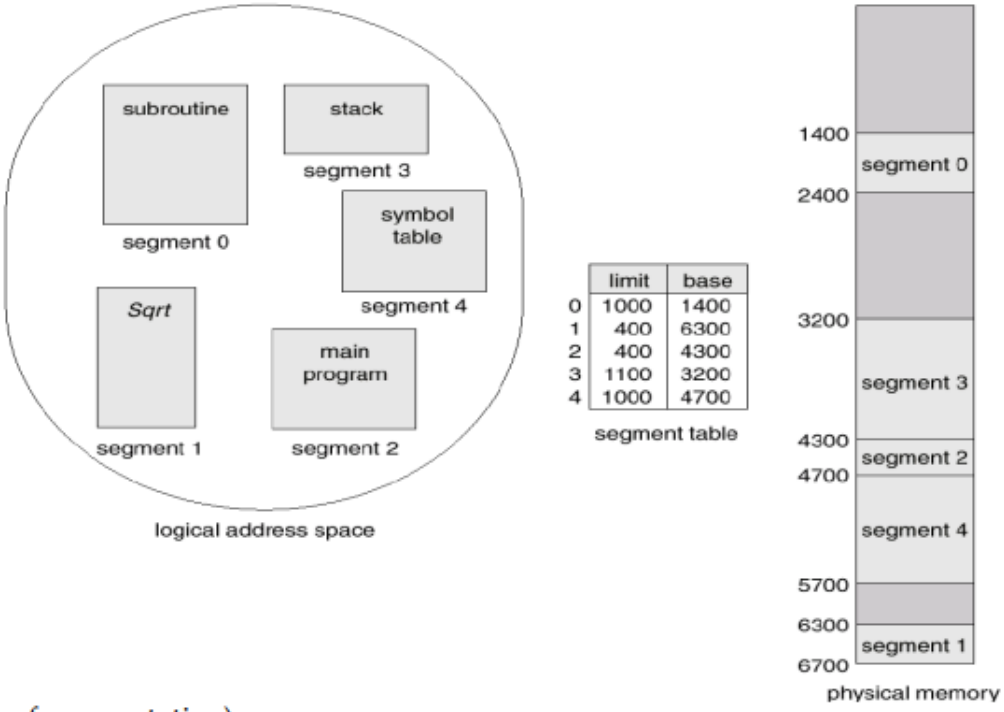
Segment-table length register (STLR) indicates number of segments used by a program.



(Diagram of Segmentation Hardware)

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

Consider we have five segments numbered from 0 through 4. The segments are stored in physical memory as shown in figure. The segment table has a separate entry for each segment, giving start address in physical memory (or base) and the length of that segment (or limit). For example, segment 2 is 400 bytes long and begins at location 4300. Thus, a reference to byte 53 of segment 2 is mapped onto location 4300 + 53 = 4353.

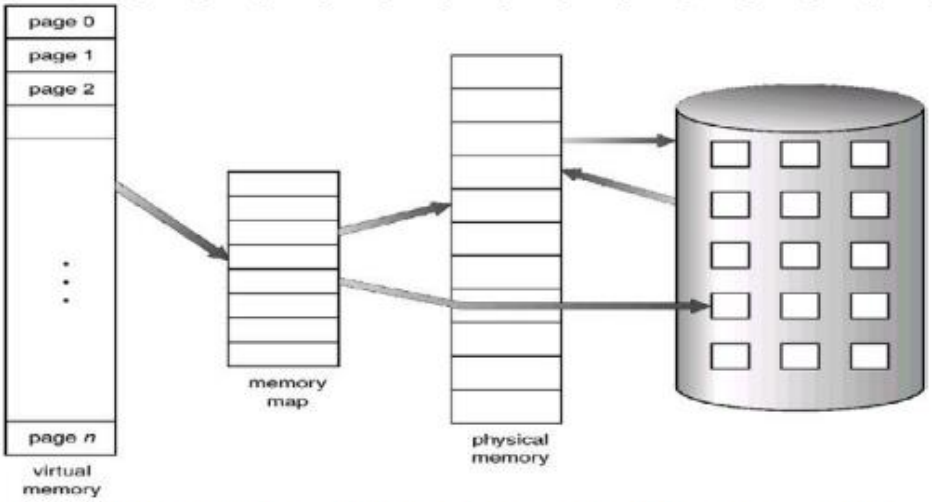


(Example of segmentation)

VIRTUAL MEMORY

Virtual memory is a technique that allows the execution of processes that may not be completely in memory. Only part of the program needs to be in memory for execution. It means that Logical address space can be much larger than physical address space. Virtual memory allows processes to easily share files and address spaces, and it provides an efficient mechanism for process creation.

Virtual memory is the separation of user logical memory from physical memory. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available. Virtual memory makes the task of programming much easier, because the programmer no longer needs to worry about the amount of physical memory available.



(Diagram showing virtual memory that is larger than physical memory)

Virtual memory can be implemented via:

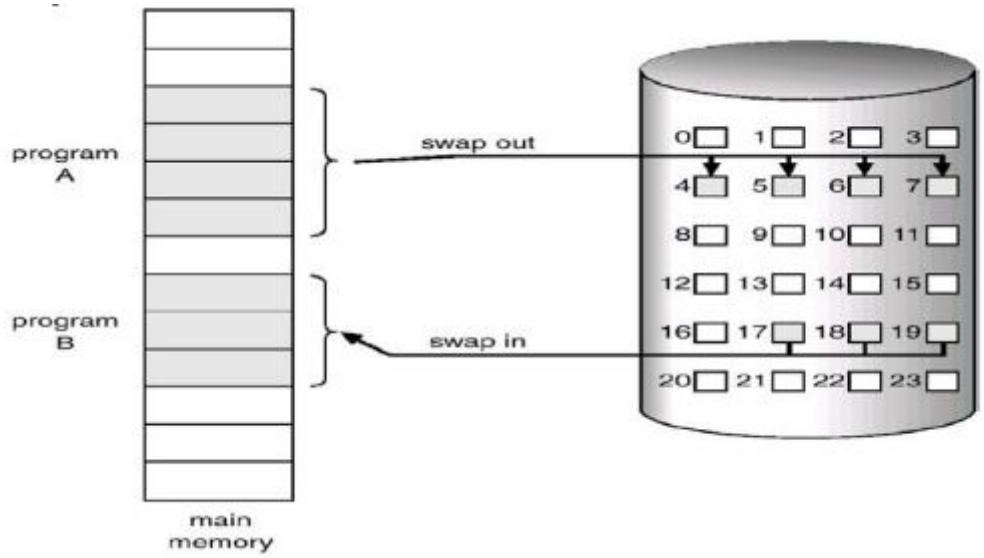
- **Demand paging**
- **Demand segmentation**

DEMAND PAGING

A demand-paging system is similar to a paging system with swapping. Generally, Processes reside on secondary memory (which is usually a disk). When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory, it swaps the required page. This can be done by a **lazy swapper.**

A lazy swapper never swaps a page into memory unless that page will be needed. A swapper manipulates entire processes, whereas a **pager** is concerned with the individual pages of a process.

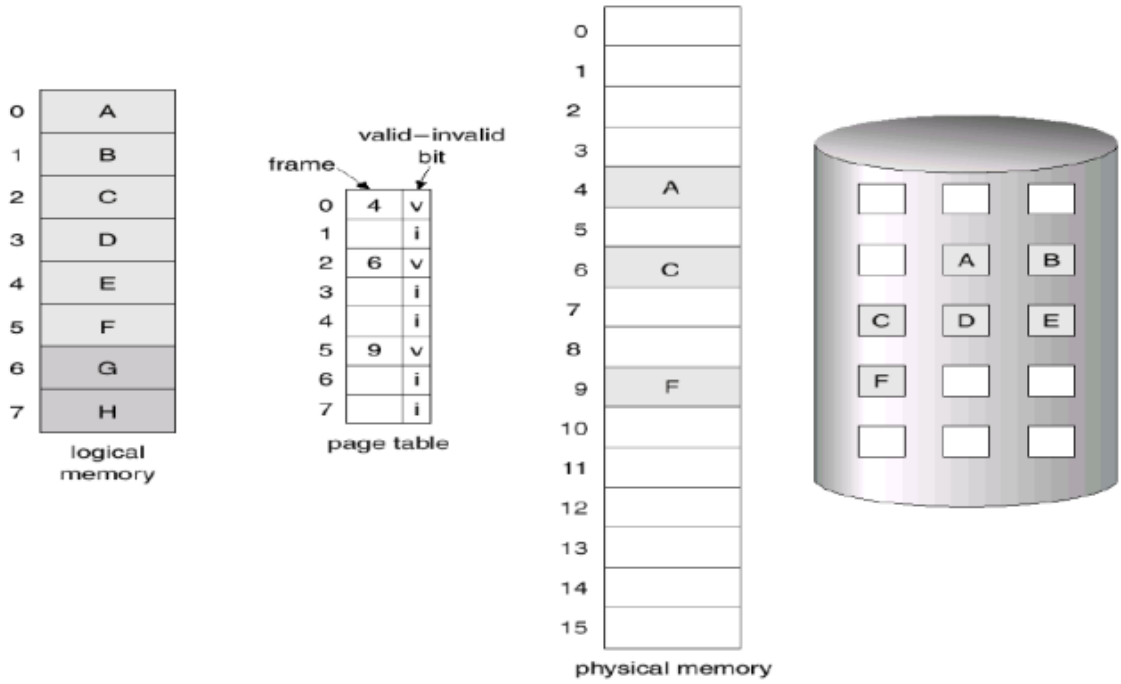
Page transfer Method: When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again. Instead of swapping in a whole process, the pager brings only those necessary pages into memory. Thus, it avoids reading into memory pages that will not be used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



(Transfer of a paged memory to contiguous disk space)

Page Table:

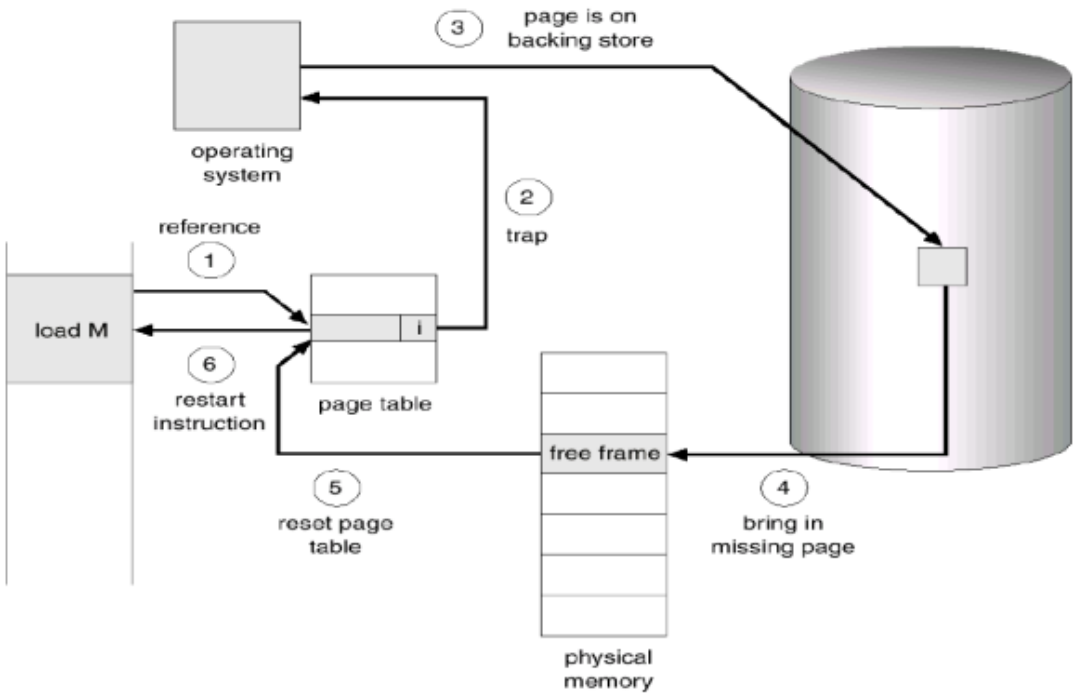
- The valid-invalid bit scheme of Page table can be used for indicating which pages are currently in memory.
- When this bit is set to "valid", this value indicates that the associated page is both legal and in memory. If the bit is set to "invalid", this value indicates that the page either is not valid 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 simply marked invalid, or contains the address of the page on disk.



(Page table when some pages are not in main memory)

When a page references an invalid page, then it is called **Page Fault**. It means that page is not in main memory. The procedure for handling page fault is as follows:

1. We check an internal table for this process, to determine whether the reference was a valid or 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 in to memory.
3. We find a free frame (by taking one from the free-frame list).
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 illegal address trap. The process can now access the page as though it had always been in memory.



(Diagram of Steps in handling a page fault)

Note: The pages are copied into memory, only when they are required. This mechanism is called **Pure Demand Paging**.

Performance of Demand Paging

Let p be the probability of a page fault ($0 \leq p \leq 1$). Then the **effective access time** is

Effective access time = (1 - p) x memory access time + p x page fault time

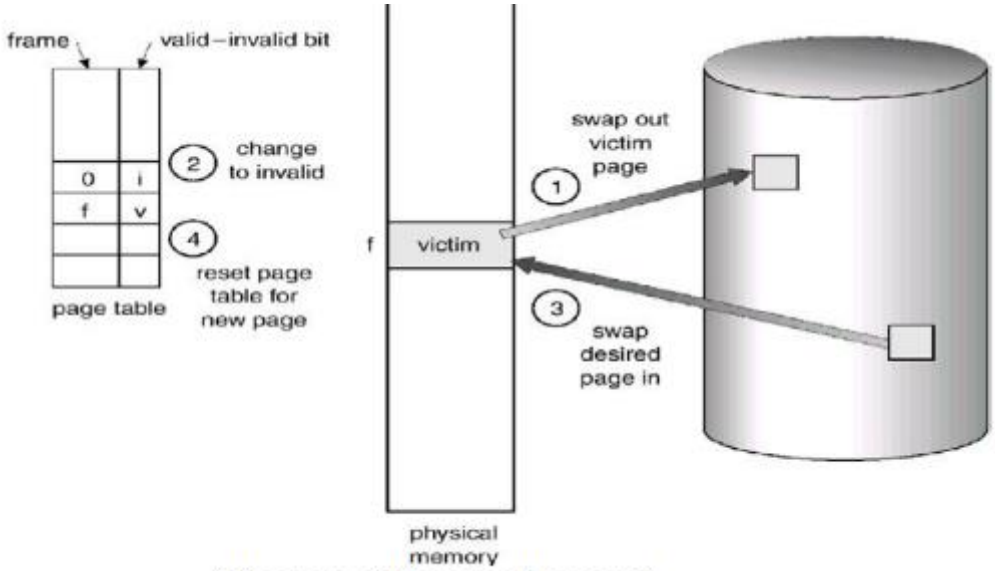
In any case, we are faced with three major components of the page-fault service time:

1. Service the page-fault interrupt.
2. Read in the page.
3. Restart the process.

PAGE REPLACEMENT

The page replacement is a mechanism that loads a page from disc to memory when a page of memory needs to be allocated. Page replacement can be described as follows:

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



(Diagram of Page replacement)

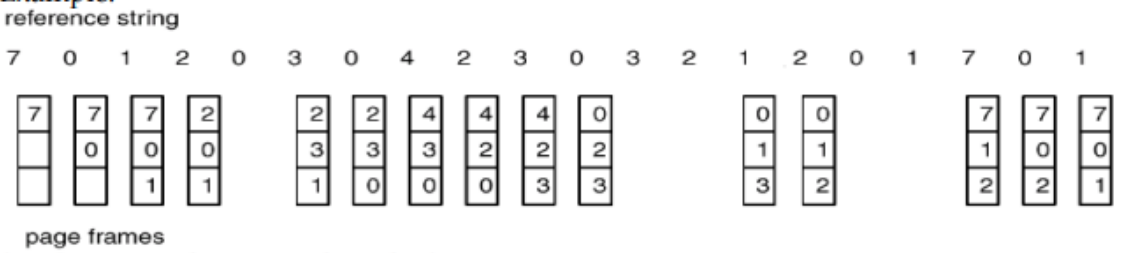
Page Replacement Algorithms: **The page replacement algorithms** decide which memory pages to page out (swap out, write to disk) when a page of memory needs to be allocated. We evaluate an algorithm by running it on a particular string of memory references and computing the number of page faults. The string of memory references is called a **reference string.**

The different page replacement algorithms are described as follows:

1. 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 to swap out. 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:



(FIFO page-replacement algorithm)

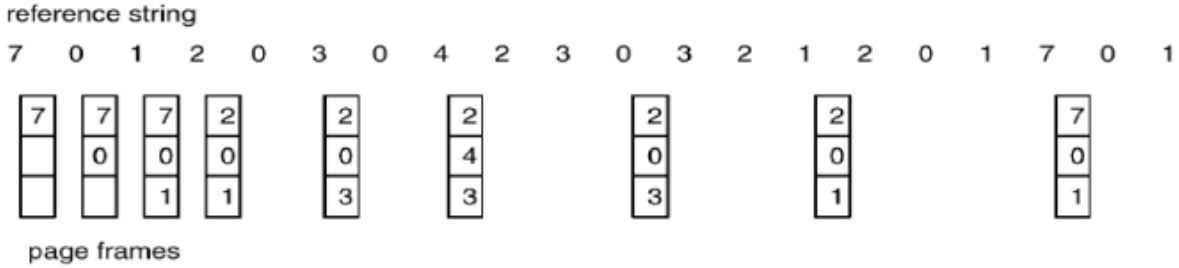
Note: For some page-replacement algorithms, the page fault rate may *increase* as the number of allocated frames increases. This most unexpected result is known as **Belady's anomaly.**

2. Optimal Page Replacement algorithm:

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. Such an algorithm does exist, and has been called OPT or MIN.

It is simply "Replace the page that will not be used for the longest period of time". Use of this page-replacement algorithm guarantees the lowest possible pagefault rate for a fixed number of frames.

Example:



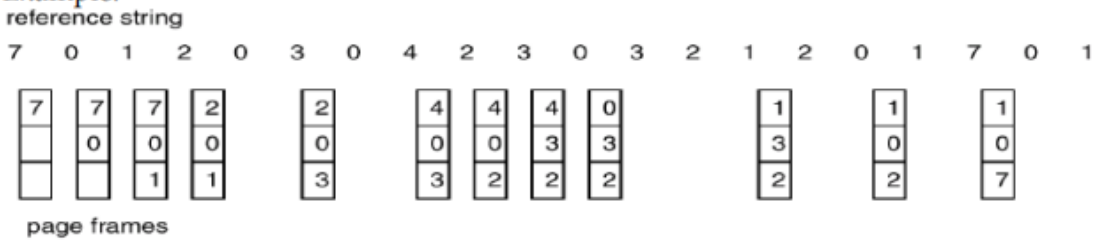
(Optimal page-replacement algorithm)

3. LRU Page Replacement algorithm

If we use the recent past as an approximation of the near future, then we will replace the page that has not been used for the longest period of time. This approach is the **least-recently-used (LRU)** algorithm.

LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses that page that has not been used for the longest period of time.

Example:

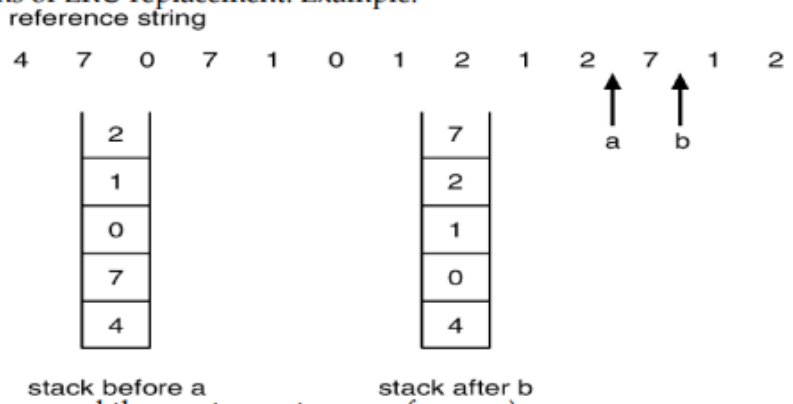


(LRU page-replacement algorithm)

The major problem is ***how*** to implement LRU replacement. An LRU page-replacement algorithm may require substantial hardware assistance. The problem is to determine an order for the frames defined by the time of last use. Two implementations are feasible:

Counters: We associate with each page-table entry a time-of-use field, and add to the CPU a 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 the 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 the LRU page, and a write to memory (to the time-of-use field in the page table) for each memory access. The times must also be maintained when page tables are changed (due to CPU scheduling).

Stack: Another approach to implementing LRU replacement is to keep a stack of page numbers. Whenever a page is referenced, it is removed from the stack and put on the top. In this way, the top of the stack is always the most recently used page and the bottom is the LRU page. Because entries must be removed from the middle of the stack, it is best implemented by a doubly linked list, with a head and tail pointer. Each update is a little more expensive, but there is no search for a replacement; the tail pointer points to the bottom of the stack, which is the LRU page. This approach is particularly appropriate for software or microcode impleme-ntations of LRU replacement. Example:



(Use of a stack to record the most recent page references)

4. LRU Approximation Page Replacement algorithm

In this algorithm, Reference bits are associated with each entry in the page table. Initially, all bits are cleared (to 0) by the operating system. As a user process executes, the bit associated with each page referenced is set (to 1) by the hardware. After some time, we can determine which pages have been used and which have not been used by examining the reference bits.

This algorithm can be classified into different categories as follows:

i. Additional-Reference-Bits Algorithm: It can keep an 8-bit(1 byte) for each page in a page table in memory. At regular intervals, 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, shifting the other bits right over 1 bit position, discarding the low-order bit. These 8 bits shift registers contain the history of page use for the last eight time periods.

If we interpret these 8-bits as unsigned integers, the page with the lowest number is the LRU page, and it can be replaced.

ii. Second-Chance Algorithm: The basic algorithm of second-chance replacement is 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. If the reference bit is set to 1, we give that page a second chance and move on to select the next FIFO page. When a page gets a second chance, its reference bit is cleared and its arrival time is reset to the current time. Thus, a page that is given a second chance will not be replaced until all other pages are replaced.

5. Counting-Based Page Replacement

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

i. LFU page replacement algorithm: 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 a large reference count.

ii. MFU page-replacement algorithm: The **most frequently used** (MFU) **page replacement algorithm** is based on the argument that the page with the largest count be replaced.

ALLOCATION OF FRAMES

When a page fault occurs, there is a free frame available to store new page into a frame. While the page swap is taking place, a replacement can be selected, which is written to the disk as the user process continues to execute. The operating system allocate all its buffer and table space from the free-frame list for new page.

Two major allocation Algorithm/schemes.

1. **equal allocation**
2. **proportional allocation**

1. Equal allocation: The easiest way to split m frames among n processes is to give everyone an equal share, m/n frames. This scheme is called **equal allocation.**

2. proportional allocation: Here, it allocates available memory to each process according to its size. Let the size of the virtual memory for process p_i be s_i, and define $S = \sum Si$

Then, if the total number of available frames is m, we allocate a_i frames to process p_i, where a_i is approximately $a_i = Si / S \times m$.

Global Versus Local Allocation

We can classify page-replacement algorithms into two broad categories: **global replacement** and **local replacement.**

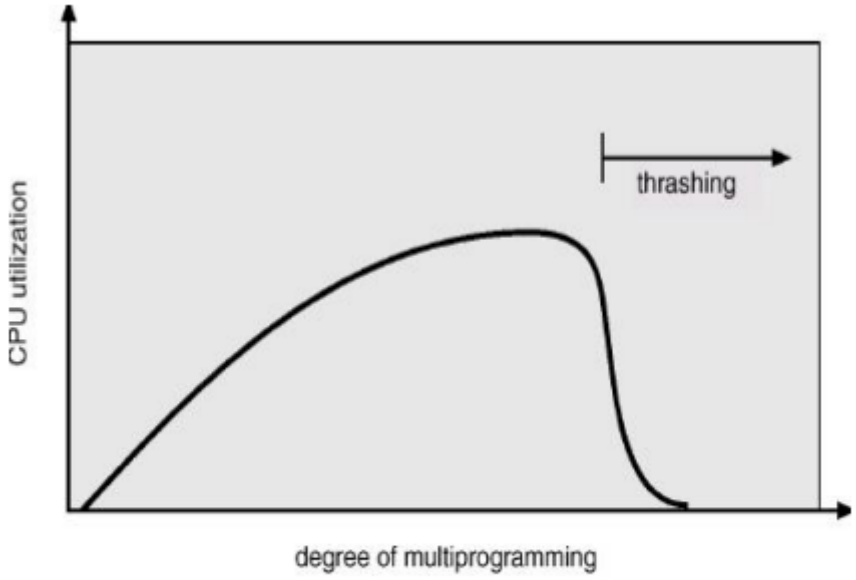
Global replacement allows a process to select a replacement frame from the set of all frames, even if that frame is currently allocated to some other process; one process can take a frame from another.

Local replacement requires that each process select from only its own set of allocated frames.

THRASHING

The system spends most of its time shuttling pages between main memory and secondary memory due to frequent page faults. This behavior is known as thrashing.

A process is thrashing if it is spending more time paging than executing. This leads to: low CPU utilization and the operating system thinks that it needs to increase the degree of multiprogramming.



(Thrashing)