Operating System (5th semester)

Prepared by SANJIT KUMAR BARIK (ASST PROF, CSE)

MODULE-III

TEXT BOOK:

1. Operating System Concepts – Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, 8th edition, Wiley-India, 2009.
2. Modern Operating Systems – Andrew S. Tanenbaum, 3rd Edition, PHI
3. Operating Systems: A Spiral Approach – Elmasri, Carrick, Levine, TMH Edition.

DISCLAIMER:

"THIS DOCUMENT DOES NOT CLAIM ANY ORIGINALITY AND CANNOT BE USED AS A SUBSTITUTE FOR PRESCRIBED TEXTBOOKS. THE INFORMATION PRESENTED HERE IS MERELY A COLLECTION FROM DIFFERENT REFERENCE BOOKS AND INTERNET CONTENTS. THE OWNERSHIP OF THE INFORMATION LIES WITH THE RESPECTIVE AUTHORS OR INSTITUTIONS."

Memory Management

Main Memory refers to a physical memory that is the internal memory to the computer. The word main is used to distinguish it from external mass storage devices such as disk drives. Main memory is also known as RAM. The computer is able to change only data that is in main memory. Therefore, every program we execute and every file we access must be copied from a storage device into main memory.

A program resides on a disk as binary execution file. The program must be brought into memory and placed within a process area (user space) to be executed. Depending on the memory management in use the process may be moved between disk and memory during its execution.

The collection of process on the disk that are waiting to be brought into memory for execution forms input queue i.e selected one of the process in the *input queue* and to load that process into memory.

All the programs are loaded in the main memory for execution. Sometimes complete program is loaded into the memory, but sometimes a certain part or routine of the program is loaded into the main memory only when it is called by the program, this mechanism is called **Dynamic Loading**, this enhance the performance.

Also, at times one program is dependent on some other program. In such a case, rather than loading all the dependent programs, CPU links the dependent programs to the main executing program when it is required. This mechanism is known as **Dynamic Linking**.

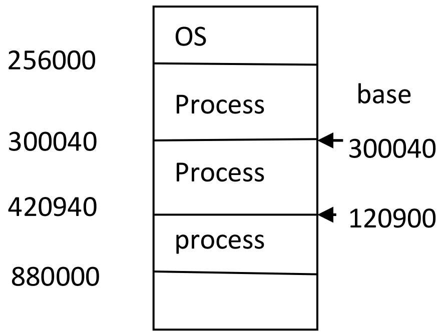
Memory protection:

We can provide protection by using two registers:

Base register holds the smallest legal physical memory address.

Limit register specifies the size of range of the process.

Ex: if the base register holds 300040 and the limit register is 120900,then the program can legally access all the address from 300040 through 420939(inclusive)



Memory Management Basic Concepts:

Address Binding:

In a system, the user program have symbolic name for address like ptr,addr,and so on. When the program gets compiled, the compiler bind symbolic addresses to relocate-able addresses and then the linkage editor or loader will bind relocateble address to absolute address. At each step binding is mapping from one address space to another space ,the binding of user program(instructions and data) to memory address can be done at any time- compile time , load time or execution time.

A.Compile-time binding:

If starting location of user program in memory is known at compile time, then compiler generates absolute code. Absolute code is executable binary code that must always be loaded at a specific location in memory. If location of user program changes, then program need to be recompiled.

Ex: Ms-DOS.COM programs in MS-DOS OS are absolute code generates by compile time binding.

B. Load time Binding:

If the location of user program in memory is not known at compile time, the compiler generates relocateble code .If location of user program changes, then program need not to be recompiled , only user code need to be reloaded to integrates changes.

C.Run time binding:

Programs (or process) may need to be relocated during runtime from one memory segment to another. Run-time (execution time) binding is most popular and flexible scheme, providing we have the required H/W support available in the system.

Logical and Physical address:

The address defined and referenced by user (programmer) in their program is called logical address. CPU generates logical address.

Physical address is the address generated by OS and is the actual physical address in the system memory. The logical address is also known as *virtual* address and physical address is known as *real* address.

At compile time and load time address binding the logical and physical address are generated are same but at execution time address binding of logical and physical addresses are different.

The space set aside for set of all logical address defined and referenced by user in their program are called logical address space and the space set aside for set of all physical address corresponding to these logical address is known as physical address space.

Mapping from virtual address to Physical address:

It is handled by H/W device memory management unit(MMU) that generates physical memory corresponding to each logical address (memory).There are various methods to perform mapping from virtual to physical address. Some of these are paging, segmentation and so on.

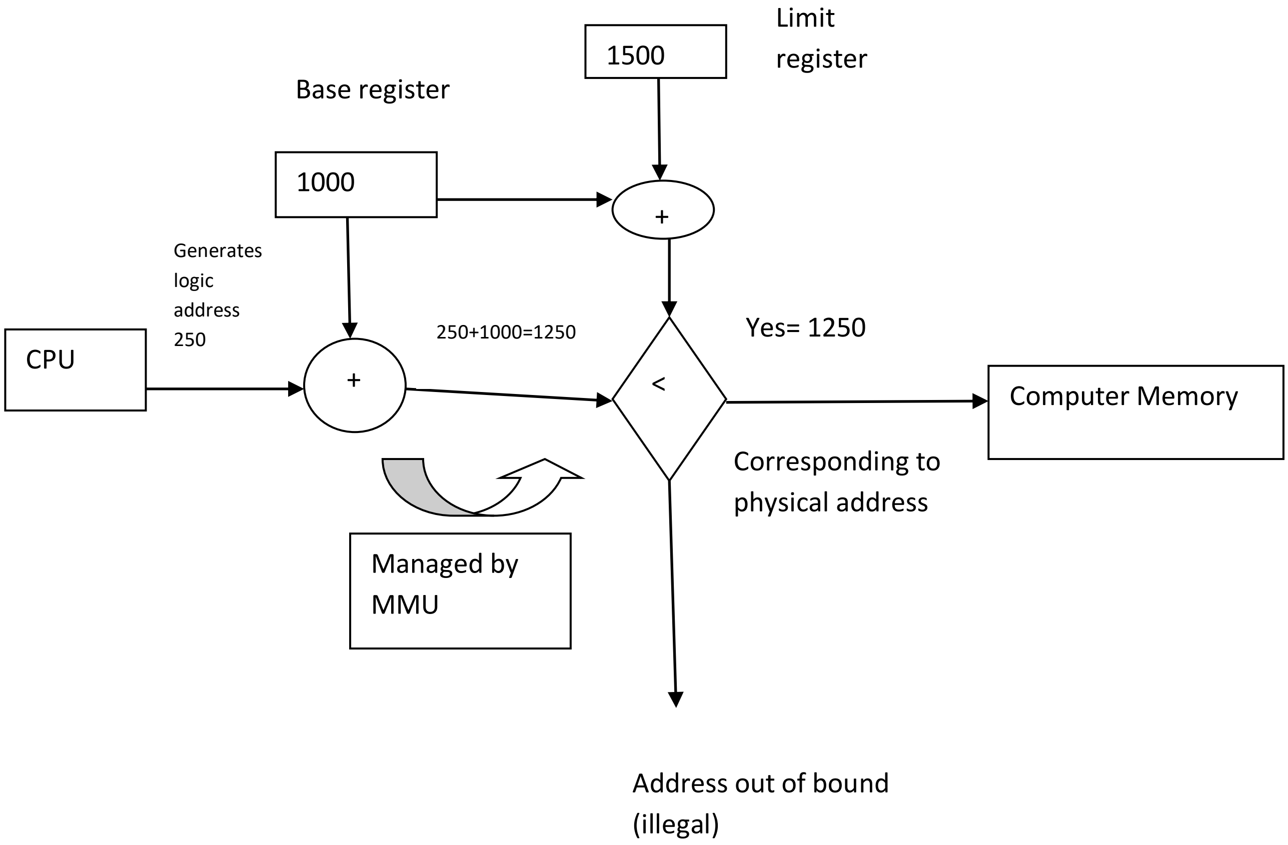
Simple MMU scheme for mapping virtual to physical addresses:-

In this scheme there are two registers base registers and limit registers. The base register (also called relocation registers) contain the starting (base) address from where the user programs start in memory.

EX: If user programs starts at 1000 in memory, then base register contains the value 1000.The limit register contains the value that specifies the range(or boundary) so that user program can access address within that range only. In other-word limit register specify the range of logical address to be used by the user programs.

Ex: if limit register contains the value 1500,and base register contains the value 1000 then, user program can access address from range 1000 to 2499

Limit register solves the protection problem by bounding each user programs to works in its defined range only and does not illegally or accidentally access other users program area.

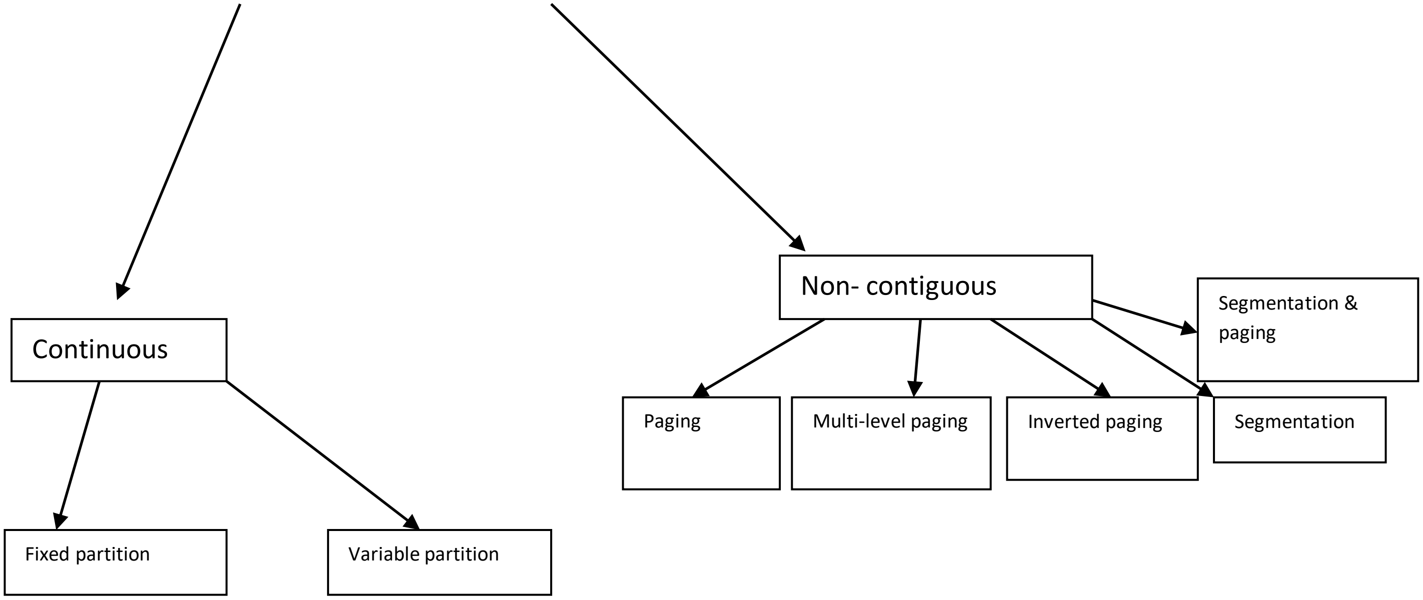


(MMU scheme for mapping virtual address to physical address)

Basically there are four function of memory management:

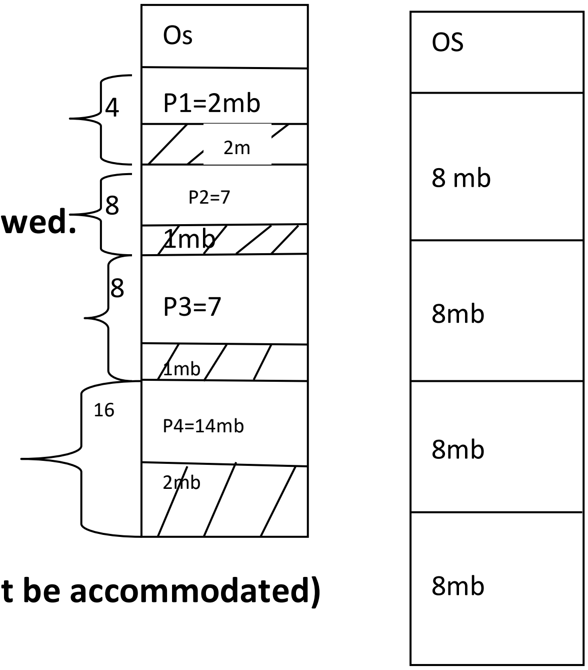
1. Keep track by which part of the memory will be currently used.
2. Deciding which process is to be loaded into the memory.
3. Allocation of memory is an efficient manner.
4. Deallocation of memory.

Memory management Techniques:



Fixed partition (Static partition):-

- No of partitions are fixed.
- Size of each partition may or may not same.
- Contiguous allocation so spanning is not allowed.



Ex: p1=2MB,P2:7 MB,P3=7 MB,P4=14 MB

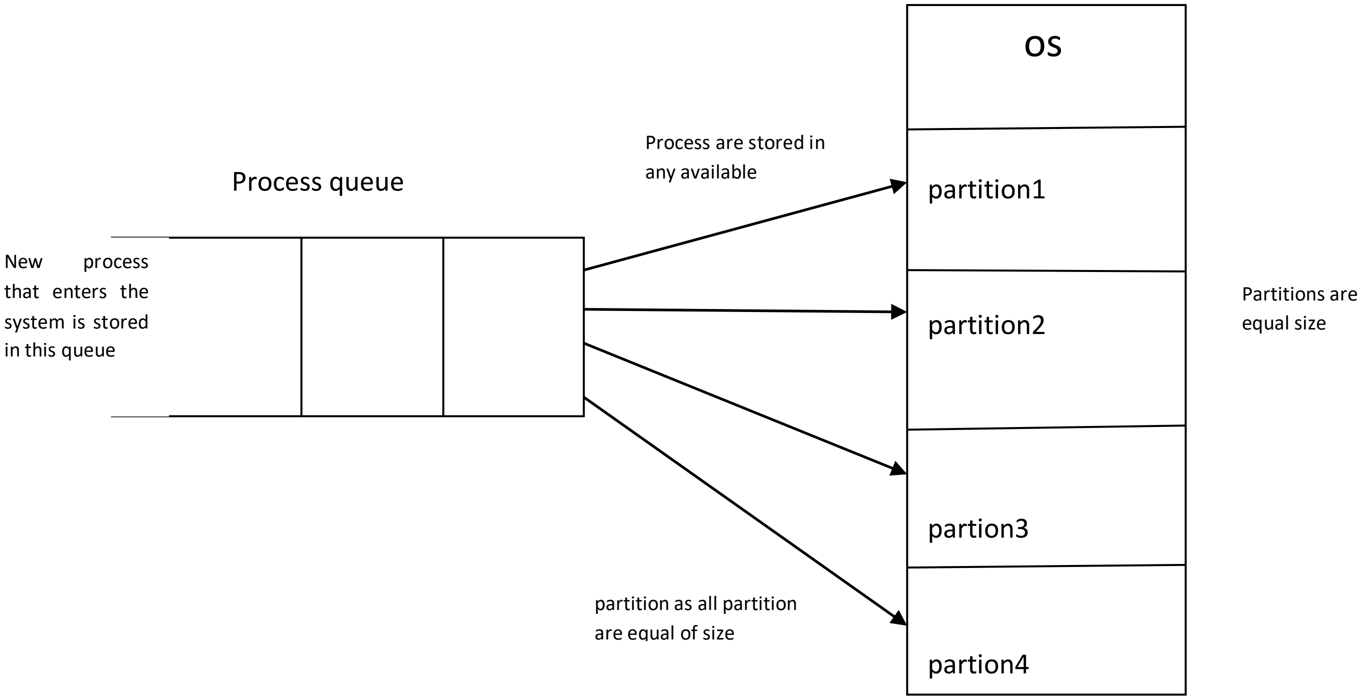
Drawback:

1. **Internal fragmentation**
2. **Limit in process size(32 MB of process cannot be accommodated)**
3. **Limit on degree of multiprogramming**
   (P5 cannot be brought into RAM)
4. **External fragmentation occur(5 mb)**

Multiprogramming with Fixed partition:

Earlier in multiprogramming system the main memory was divided into fixed partitions where each partition was capable of holding one process. Fixed partitions can be of equal sizes or unequal sizes .but their size once fixed cannot be changed means staring and an end address of each partition is fixed in advance.

In case of equal size fixed partition, there is a single process queue in which all the processes waiting to come in main memory for execution are maintained. If there is partition available, the process is loaded into that partition. Since all partitions are equal size, it does not matter which partition is used.


OS
Process are stored in any available
Process queue
partition1
New process that enters the system is stored in this queue
partition2
Partitions are equal size
partion3
partition as all partition are equal of size
partion4

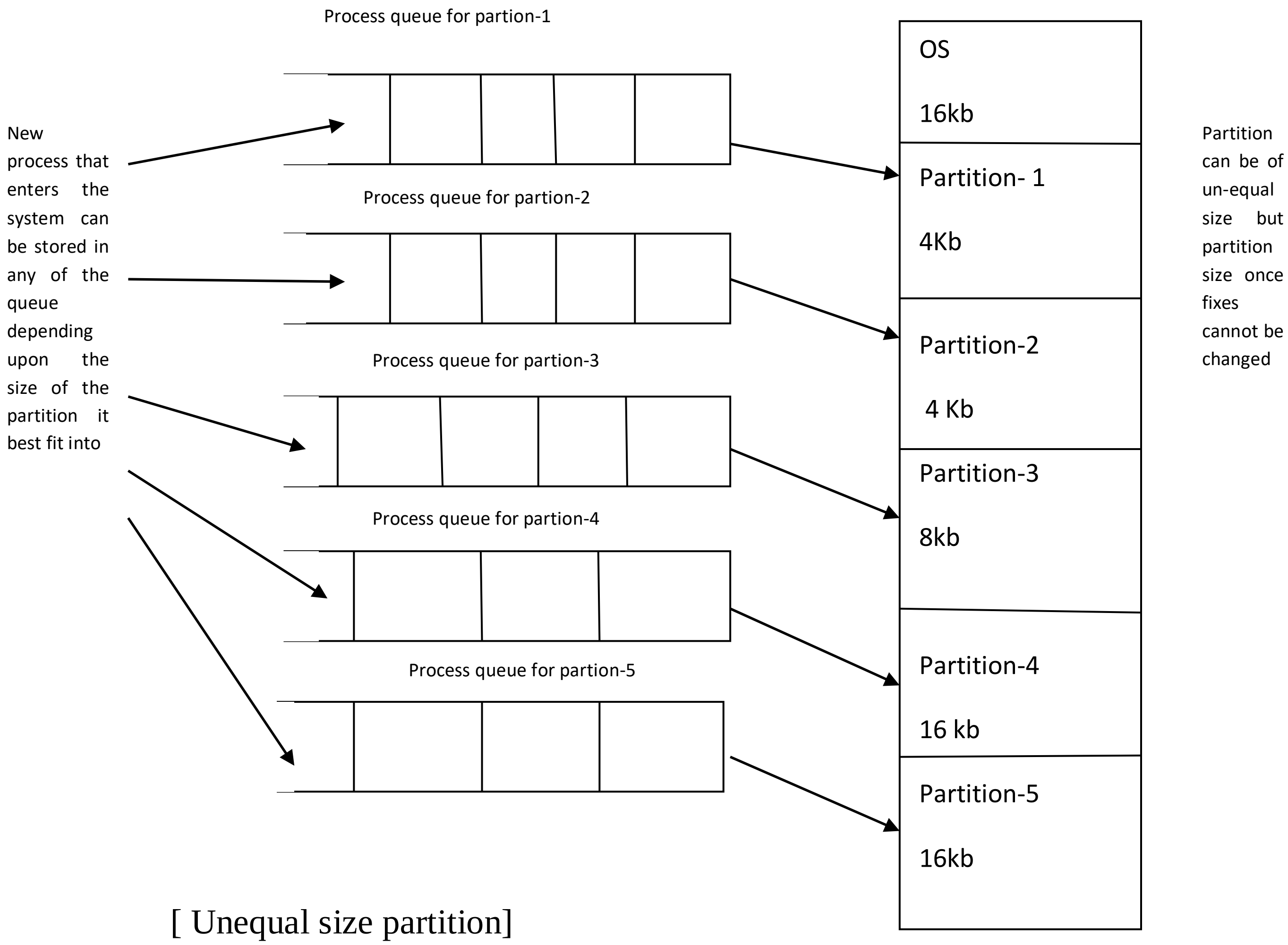
[Equal size partition]

Unequal size fixed partition:

In case of unequal size fixed partition, the size of each partition is fixed once in starting and that size cannot be changed, it is fixed.

Ex: S**uppose** system has memory of size 64 kb. Out of this 16kb is occupied by OS and rest 48 kb is there for user process. This is divided into five partitions as:

Two partitions of size 4kb
One partition of size 8 kb
Two partition of size 16kb


Process queue for partion-1
Process queue for partion-2
Process queue for partion-3
Process queue for partion-4
Process queue for partion-5
New process that enters the system can be stored in any of the queue depending upon the size of the partition it best fit into
OS
16kb
Partition- 1
4Kb
Partition-2
4 Kb
Partition-3
8kb
Partition-4
16 kb
Partition-5
16kb
Partition can be of un-equal size but partition size once fixes cannot be changed

[Unequal size partition]

A separate process queue is maintained corresponding to each partition size. Here strategy is that when a new process enters the system it is stored in the queue of that partition whose size is best fitted to accommodate that program. The problem is here that some queues might be empty while some might be loaded.

In both cases [equal or unequal size partition] when a program completes its execution and terminates, its allocated memory partition is free for another program waiting in a queue. But this multiprogramming with restriction of fixed size causes wastage of memory and lead to internal fragmentation.

Internal fragmentation: It is problem that occurs when size of the program is smaller than the size of partition allocated to it as this extract allocated memory is not used by program and got wasted.

Advantage:

- Easy to use (implement)
- Allocation and de-allocation of process is easy

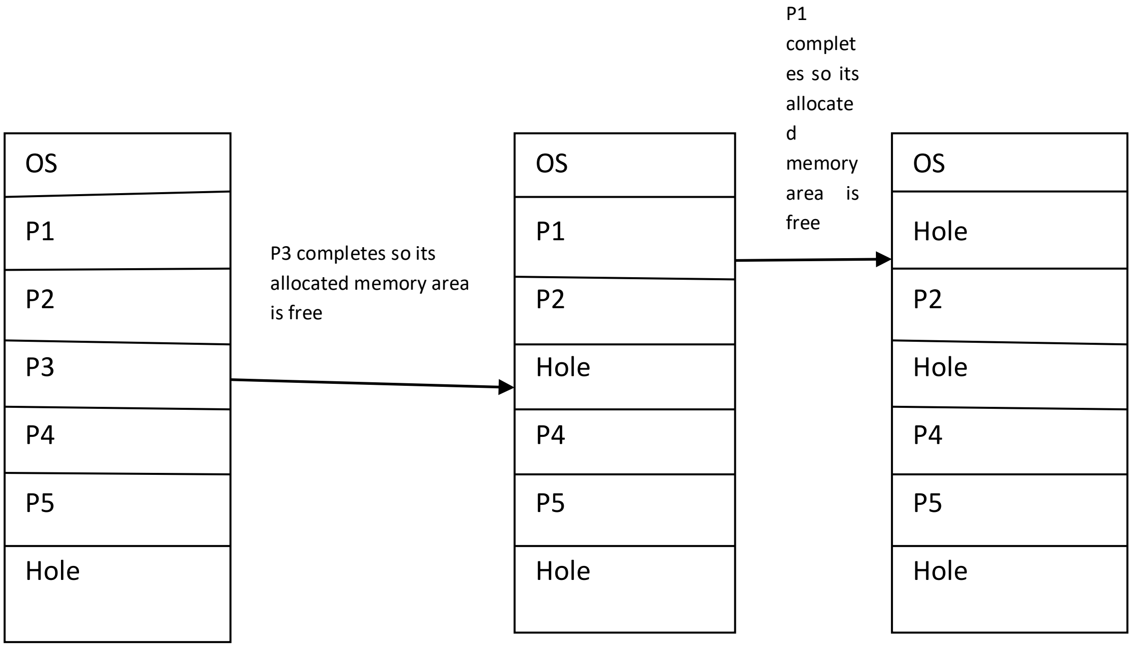
Multiprogramming with Variables Partitions:

In multiprogramming system with variable partitions the main memory is divided into partitions that are variable (not fixed). Here in this system:

- Each system is capable of holding one process.
- Partition can change size as the need arises.
- The adjacent free partitions can combine together if needed so as to form bigger space to accommodate large process.
- There can be different number of partitions depending upon the size of memory.

There is a process queue in which all the processes waiting to come in main memory for execution are maintained .The OS uses scheduling algorithm to dispatch process one by one from the process queue. If there is partition available large enough to accommodate the process, OS loads the process into that partition. When a program completes its execution and terminates, its allocated memory area is free. The various available memory areas are called

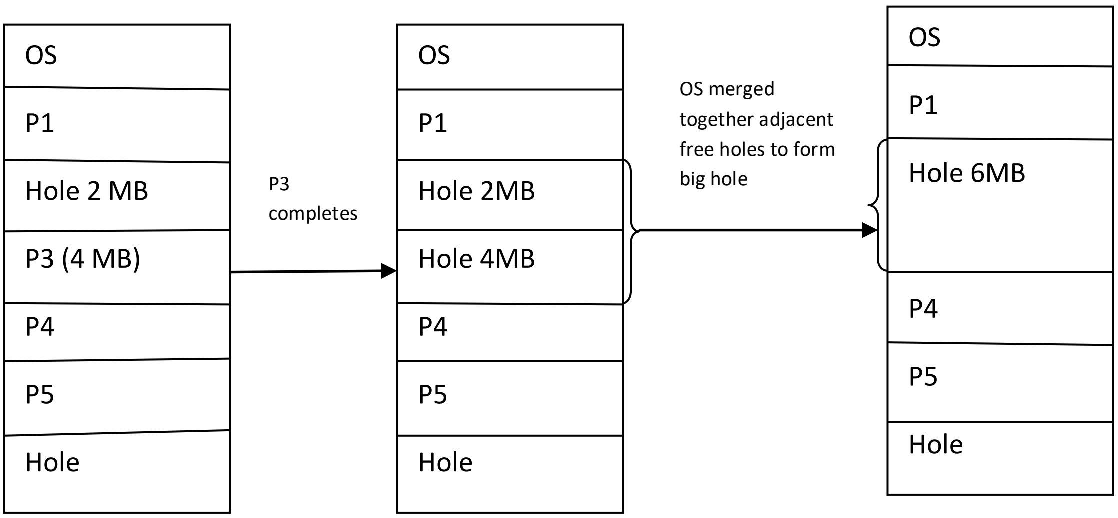
holes. As the memory is allocated and de-allocated , ***holes*** will appear in the memory .***Holes*** of various sizes are spread throughout the memory.


OS
P1
P2
P3
P4
P5
Hole
P3 completes so its allocated memory area is free
OS
P1
P2
Hole
P4
P5
Hole
P1 completes so its allocated memory area is free
OS
Hole
P2
Hole
P4
P5
Hole

[Holes in variable partitions multiprogramming system]

When a process is dispatched from process queue to store in memory , OS first searches for the *Hole* large enough to accommodate that process .If none of the available holes are large enough to accommodate that process then available adjacent free partitions can be merged together to form a big hole (bigger space) to accommodate that process. This process of merging adjacent free partitions to form big hole is called ***coalescing.***

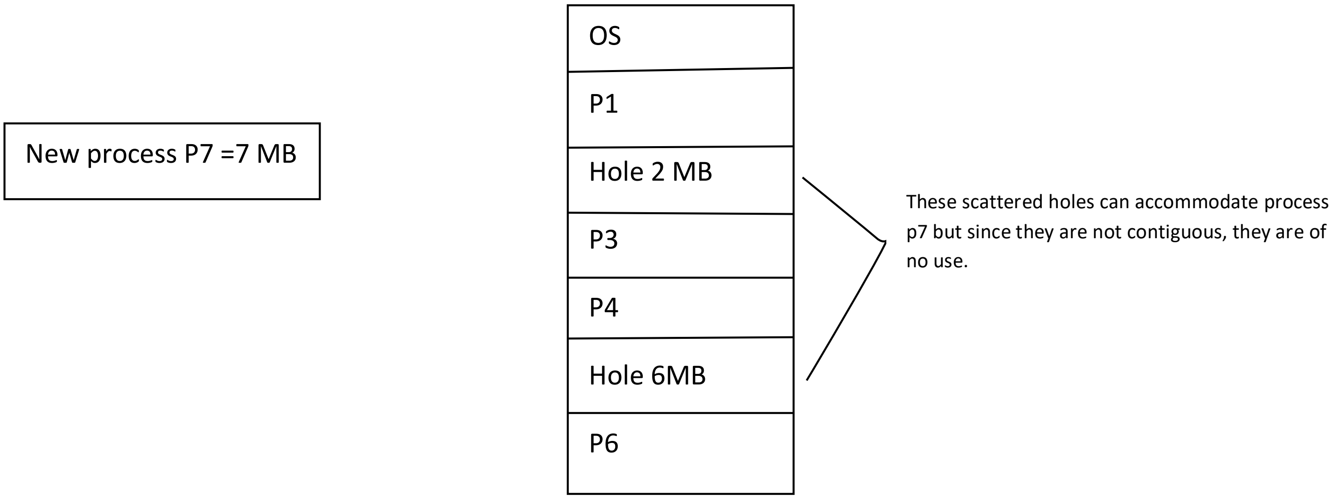
Coalescing Technique



There are three main approaches to assign available ***Hole*** to the process. The OS can Use any of these approaches depending upon its strategy. These approaches are:

1. **<u>First fit:</u>** There may be many holes available in the memory. In this approach the OS did not waste time in searching the free hole large enough to accommodate the given process , instead it allocates the first hole it find large enough to satisfy the request.
2. **<u>Best fit:</u>** In this approach OS maintains ordered list holes. When a process arrives ,the OS allocates the smallest hole from that available holes (arranged order) that is large enough to accommodate the given process .Using this method , the memory has the smallest leftover.(allocate the smallest hole that is big enough).
3. **<u>Worst fit:</u>** When a process arrives, the OS allocate the largest hole (from the list of available holes) that is large enough to accommodate the given process. Using this method , the method has the largest leftover hole, this largest leftover hole may be more useful than smallest leftover hole because that largest left over hole may be too large to accommodate any process(allocate the largest hole).

Multiprogramming system with variable partitions suffers from problem of external fragmentation. External fragmentation occurs when a system has enough holes(free spaces) satisfy the process space requirements but hole are non-contiguous , means spread through memory they are of no use.(they are too small to satisfy any process request).These scattered holes if combined together to form one contiguous hole may form a large percentage of disk free space.

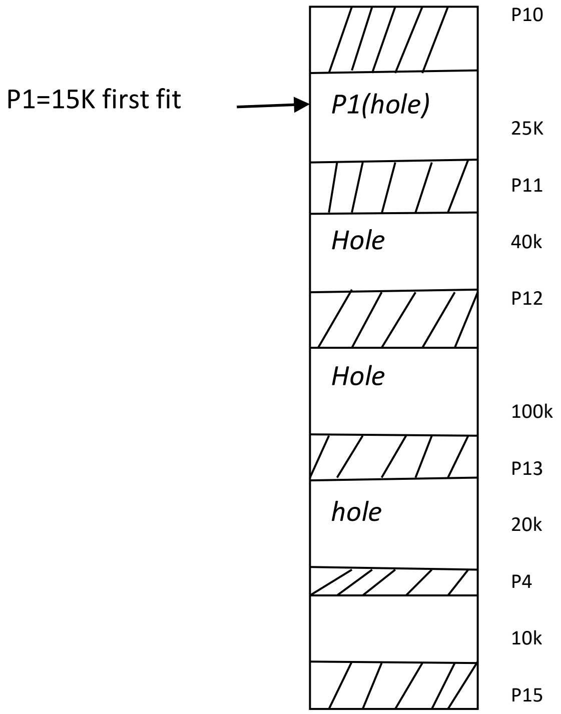


These scattered holes can accommodate process p7 but since they are not contiguous, they are of no use.

[External fragmentation]

EX: first-Fit, Best-fit, worst –fit

p1=15k



First fit=simple/fast

Best fit=slow/internal fragmentation will be less

Worst fit= slow

Coalescing is one solution to external fragmentation .Another solution to reduce external fragmentation is compaction. In compaction all the allocated blocks are moved to one end of memory, hence all the allocated blocks are together and all the holes are together in memory. So now holes are continuous instead of scattered throughout the memory and thus can be used more efficiently. Compaction is sometimes called *garbage collection.*

Disadvantage of memory compaction:

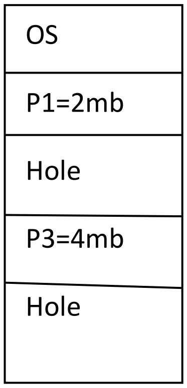
1. Much of CPU time and other system resources got wasted in compaction procedure.
2. During compaction some compaction tasks need to be halted which can create problematic situation especially in interactive and real time systems.
3. Compaction requires relocation of process i.e moving process and its associated data to new address. This will changes base address of process and requires changing the base register value to hold new changed value. Hence system need to maintain all this relocation information which itself is a big overhead. Also if relocation is static means done at load time

compaction is not possible. Compaction is possible when relocation is dynamic means done at run time.

Solution this problem of fragmentation are *paging and segmentation*

Advantage of Variable Partition:

1. No internal fragmentation
2. No limitation of degree of multiprogramming
3. No limit of process size



Draw back:

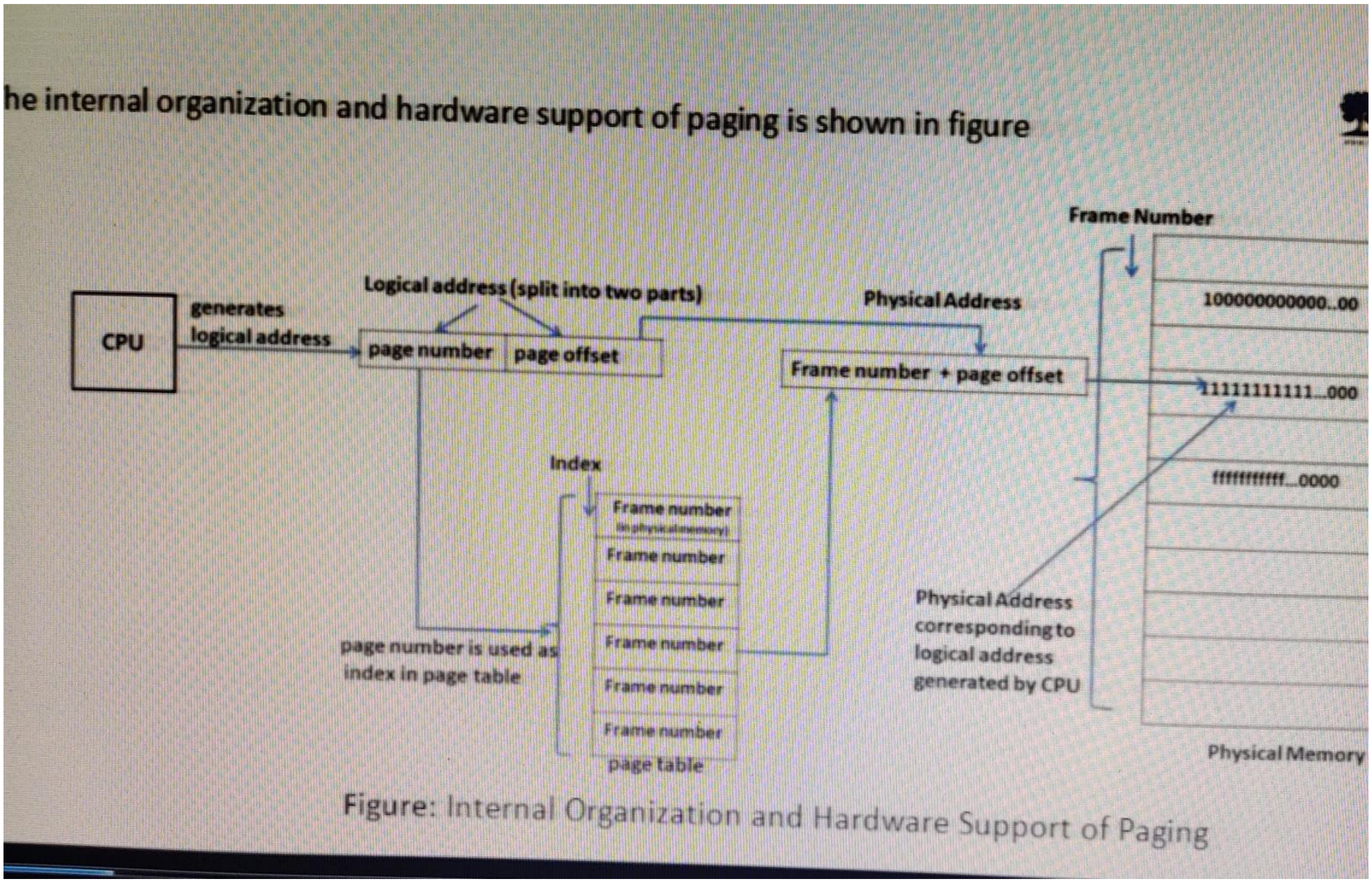
1. External fragmentation
2. Allocation and Deallocation is complex

PAGING:

1. Basic paging Method
2. Paging H/W-TLB(Translation Look –aside buffer)
3. Different types of page table
   a. Hierarchical paging
   b. Hashed paged Table
   c. Inverted Page table

Basic paging Method:

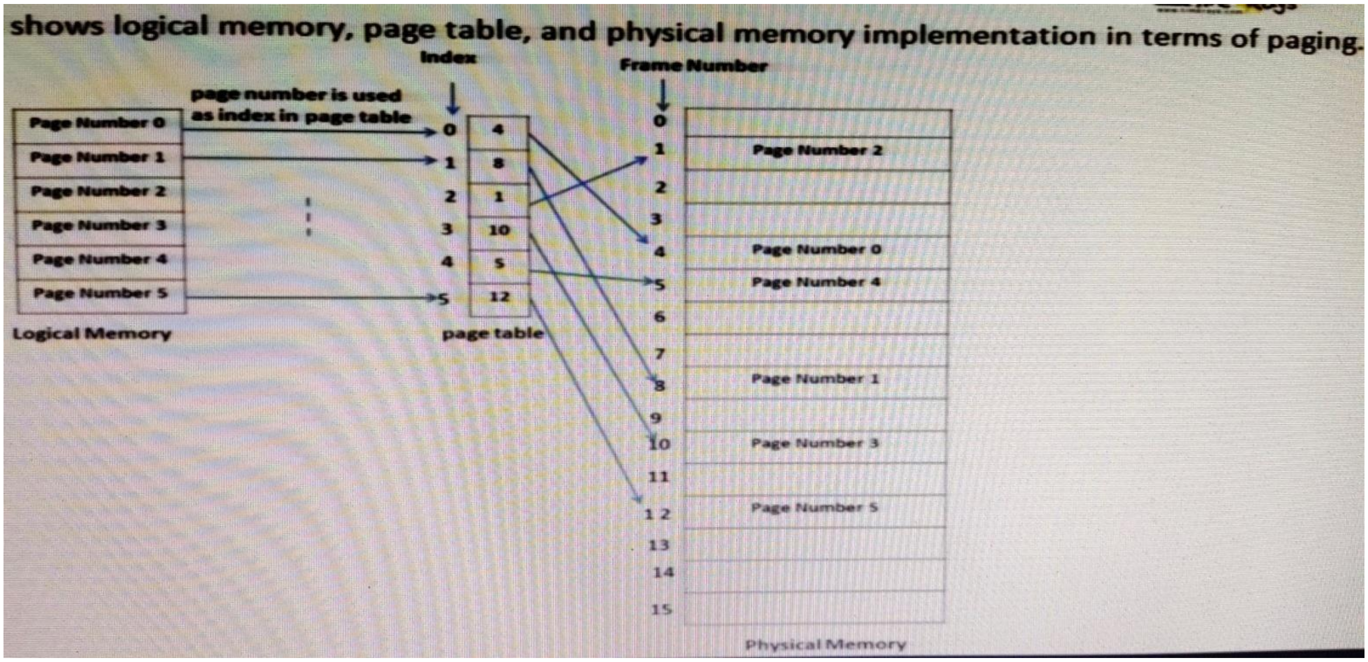
Paging is one of the memory management scheme used generally in most of the OS. Paging is good solution to problem of fragmentation. In paging both physical and virtual memory are divided into fixed sized blocks. These fixed sized blocks are called frames in physical memory and pages in logical memory. Both frame and pages are of same size.



he internal organization and hardware support of paging is shown in figure

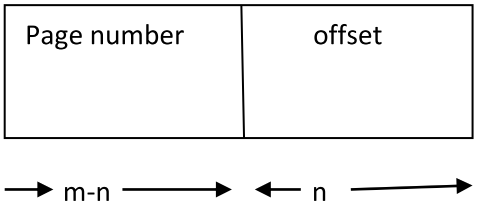
CPU
generates logical address
Logical address (split into two parts)
page number
page offset
Physical Address
Frame Number
Frame number + page offset
100000000000..00
11111111111...000
ffffffffff...0000
Index
Frame number (in physical memory)
Frame number
Frame number
Frame number
Frame number
Frame number
page number is used as index in page table
page table
Physical Address corresponding to logical address generated by CPU
Physical Memory

Figure: Internal Organization and Hardware Support of Paging

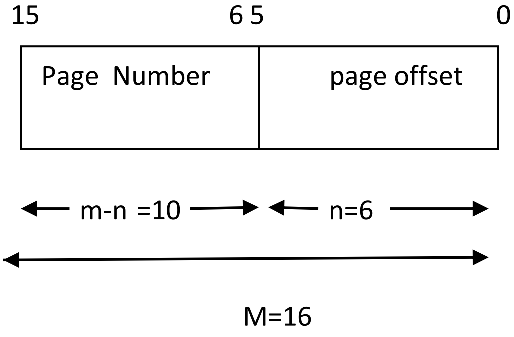

shows logical memory, page table, and physical memory implementation in terms of paging.

Page size is decided by the paging H/W.Page size is of the form 2^x (power of 2 i.e 2^4 =16 and so on). .If suppose logical address is of size 2^m (unit can be bits , bytes or word) in which page size is 2^n , then higher order (m-n) bits of logical address denotes page number and 'n' lower bits denotes page offset.

Hence logical address division is:



Ex: logical address of size of 2^{16} and page size is 2^6



Consider an Example:

Physical memory is of size 16 bytes. Logical memory has pages of size 2 bytes each. So total number of pages can be stored in physical memory = Physical memory size/logical memory size (page size) =16/2=8 pages.

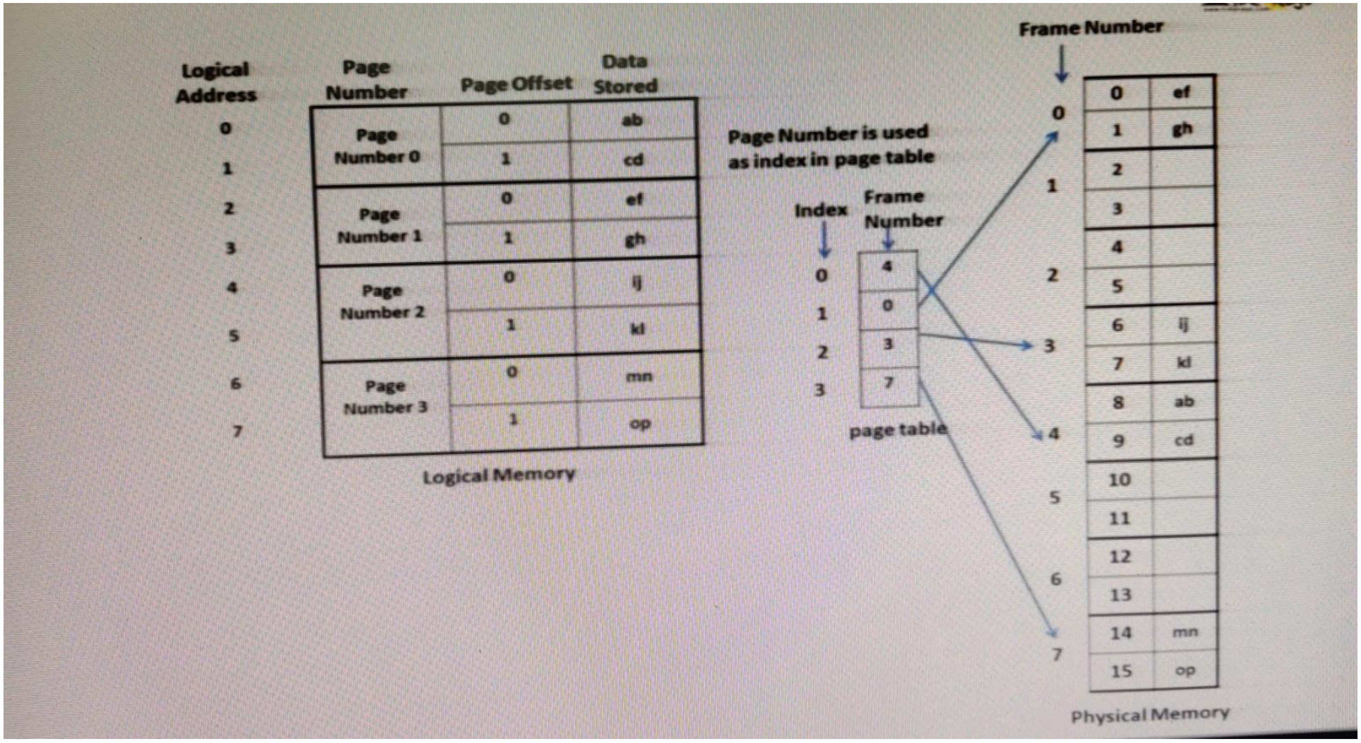
3	1
Frame no.	frame offset/size

← Physical address 4 bits →

The formula to find out physical address corresponding to logical address is:

(Frame number X page size)+page offset

Ex: logical address 0 is page no. 0 , page offset 0 and frame no. 4
Physical address =(4*2)+0=8
Logical address 0 maps to physical address 8 and the data here is “ab”



Logical Address	Page Number	Page Offset	Data Stored
0	Page Number 0	0	ab
1		1	cd
2	Page Number 1	0	ef
3		1	gh
4	Page Number 2	0	ij
5		1	kl
6	Page Number 3	0	mn
7		1	op

Logical Memory

Page Number is used as index in page table

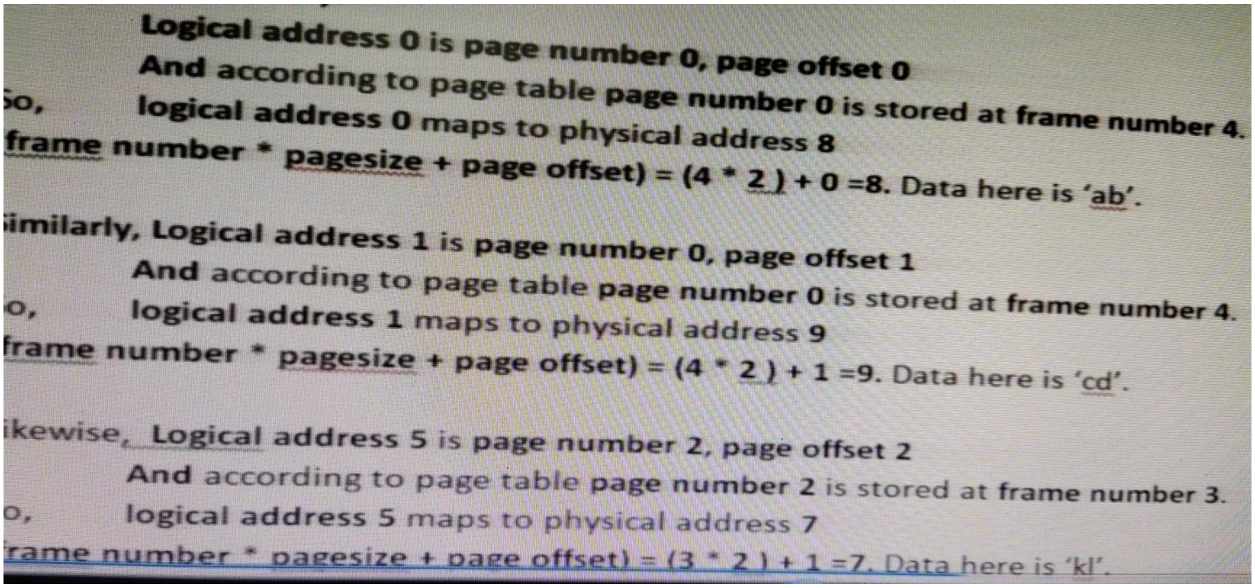
Index	Frame Number
0	4
1	0
2	3
3	7

page table

Frame Number		
0	0	ef
	1	gh
1	2	
	3	
2	4	
	5	
3	6	ij
	7	kl
4	8	ab
	9	cd
5	10	
	11	
6	12	
	13	
7	14	mn
	15	op

Physical Memory

EXAMPLE:



Logical address 0 is page number 0, page offset 0

And according to page table page number 0 is stored at frame number 4.

So, logical address 0 maps to physical address 8

frame number * pagesize + page offset) = (4 * 2) + 0 =8. Data here is 'ab'.

imilarly, Logical address 1 is page number 0, page offset 1

And according to page table page number 0 is stored at frame number 4.

o, logical address 1 maps to physical address 9

frame number * pagesize + page offset) = (4 * 2) + 1 =9. Data here is 'cd'.

ikewise, Logical address 5 is page number 2, page offset 2

And according to page table page number 2 is stored at frame number 3.

o, logical address 5 maps to physical address 7

rame number * pagesize + page offset) = (3 * 2) + 1 =7. Data here is 'kl'.

<u>Assignment:</u>

Question1: Consider a system which has logical address (LA=7bits), Physical address =6bits, page size =8 word, and then calculate the no. of pages and no. of frames.

Question2: Logical address space(LAS)=4GB,Physical Address space (PAS)=64GB, and the page size=4KB, then determine the following:

No. of pages=?

No. of frames=?

No. of entries in page Table=?

Size of page table =?

Paging H/W –TLB(Translation Look –aside Buffer)

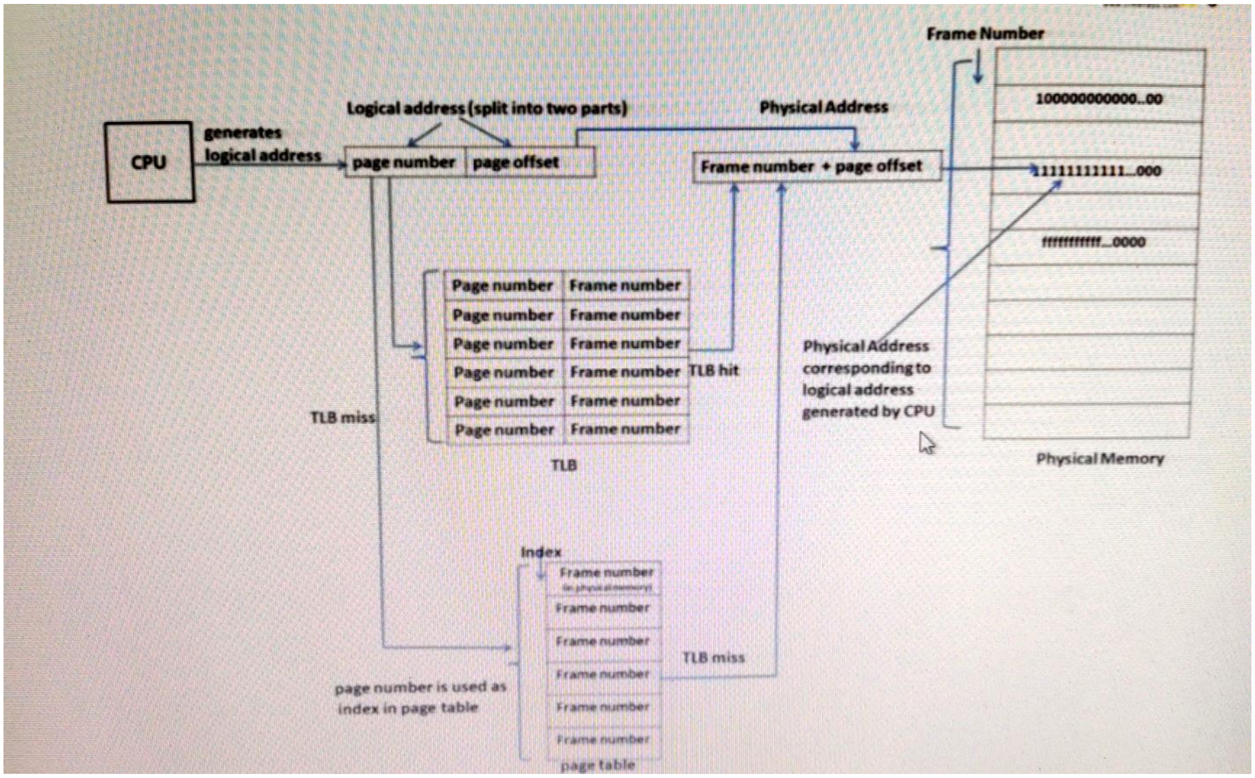
Paging requires a large amount of mapping information and this mapping information in form of page table is generally stored in physical memory so paging logically requires to access memory very often .Each time a virtual address generated by the program, memory access to read the page table is required so as to map virtual address to physical address. Accessing the page table after each virtual address generated by CPU (once per instruction) makes the system considerably slow and affects the system performance .Many times not one but several memory access are required per instruction and this make the situation more worse by degrading the system performance terribly low .Memory page table can be faster on context switching but very time consuming in actual accessing.

So to overcome this problem, the solution is to use *small and fast to* access hardware cache known as TLB(Translation Look –aside Buffer).TLB is the part of MMU and is used for virtual to physical address translation.

TLB construction:

TLB is in the form of table where each row of TLB consists of a page number and its associated frame number. We know that the logical address generated by CPU has two parts-*page number* and *page offset*. So whenever CPU generates logical address, page number part of this address is searched in the TLB. If the required page number is found in the TLB, this is known as ***TLB hit*** and its associated frame number is used to map the logical address to the physical address.

If the page number is not found in the TLB, this is known as ***TLB miss*** and then the page table is referenced. From the page table frame number is obtained which is for mapping and also this page number and frame number entry is done in the TLB so that they can be found when TLB is referenced next time. If the TLB is already full then OS uses page replacement algorithm (policies) like LRU,FIFO and so on to transfer one old entry to disk storage so as to make for new entry that shows in figure paging using TLB.



There is one important thing that needs to be taken care with TLBs .The TLB contains entry for virtual address to physical translations that are only valid for the currently executing process and these translations entries in TLB are of no use for other processes. Therefore when context switching takes place from one process to another, the paging H/W and OS should make sure that processes that gives to be executed next and using the TLB doesn't accidentally use TLB entry of some previously executed process. To solve this problem there are two approaches:

1. **Flush(Erase):**
   Flush the TLB each time there is a context switch means clearing off the TLB entries before processes that is going to be executed and using the TLB starts its execution.
2. **Address space identifiers:(ASID)**
   Add one more field in the TLB called Address space identifiers(ASID) along with each page number and frame number entry in the TLB.ASID uniquely identifies a process and is used to differentiates one process from the other. ASID is somewhat like a process identifier (PID), but usually

ASID has fewer bits as compared to PID. Hence, with ASID the TLB can have translation entries of different processes at the same without any problem, when page number entry is found in TLB, then ASID associated it with is also matched with currently executing process ASID
If TLB ASID entry matches with currently executing process ASID, it is a TLB hit other TLB miss.

<u>TLB problem:</u>

Assume that the TLB search time is 9s and memory access time is 100ns.If TLB is not used , then memory access time= 2 * 100 ns=200ns.

If the TLB is used then:

If the TLB hit= 9s+100ns=109ns

TLB miss= (9s+100s+100ns=209ns)

Let TLB hit ratio =80%i.e means for 80% of time the page number is found in the TLB and 20% of time the page number not found in the TLB i.e 20% of TLB miss ratio.

$$\left.\begin{array}{l}\textit{For 80\%, the time taken 109ns (TLB hit)}\\ \textit{And 20\%, the time taken 209ns (TLB miss)}\end{array}\right\}\text{Effective access time}$$

So, effective access time: 0.8 *109ns+0.2*209ns

General formula=

ETA=H * (T+t)+(1-H)*(2T+t)
Where H=probability that an intended frame number would be found in TLB itself (H proportional to size of TLB)
T=RAM access time
t=TLB access time
While effective memory access time (without TLB) =2T.

Q.A paging scheme using TLB. TLB access time 10ns and main memory access time takes 50 ns. What is the effective memory access time(in ns)if TLB hit ratio is 90% and there is no page fault.

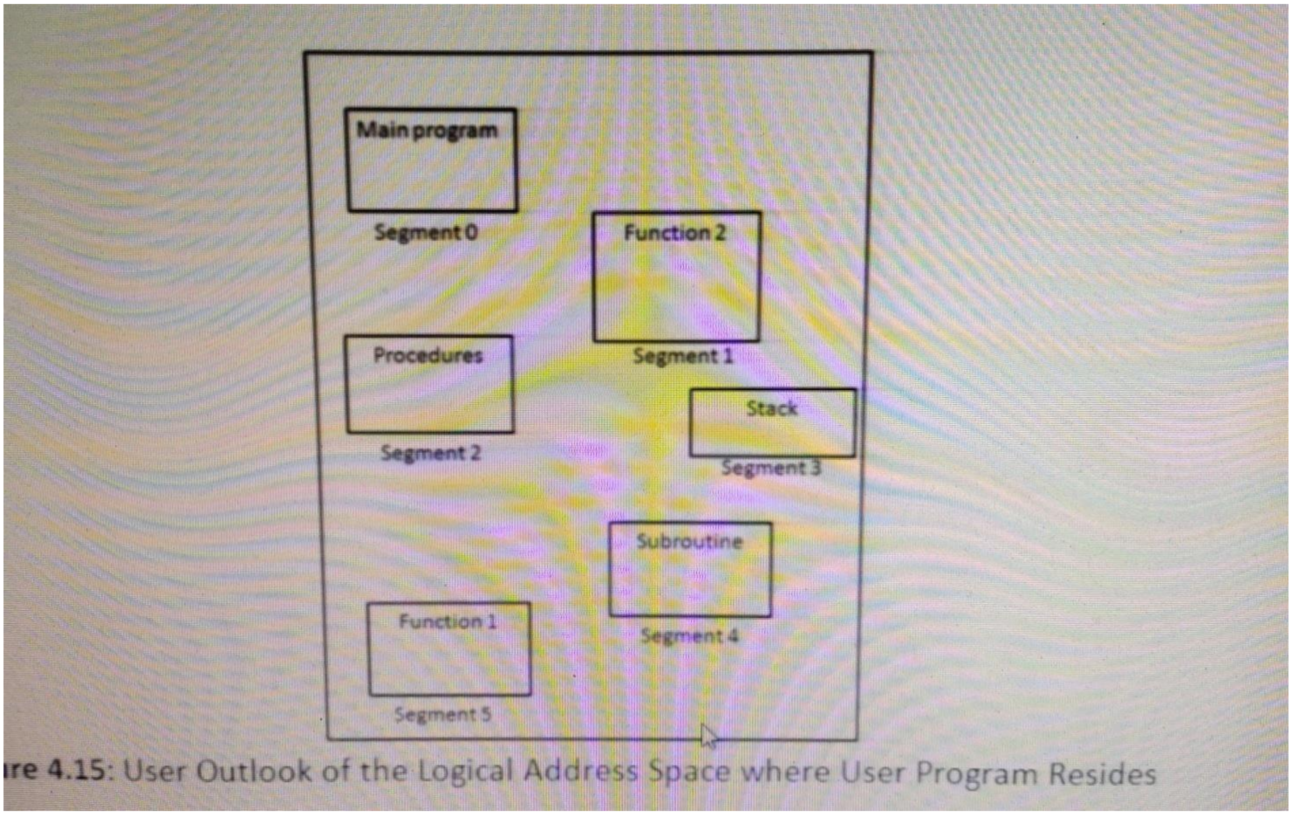
*ETA= 90%(50+10)+10%(2*50+10)=65ns*

Q. H=0.9 t=20ns T=100ns calculate the reduction in memory access time.

Solution: Effective memory acess time with TLB:
*ETA=H * (T+t)+(1-H)*(2T+t)*
*0.9*120+0.1*220*
=130ns
*Effective memory access time without TLB: =2T=2*100=200ns*
Reduction in memory access time
*= ((200-130)/200)*100=35%.*

Segmentation:

Segmentation is the scheme used by memory-management unit for virtual to physical address translation. Segmentation provides a view that a user can more easily relate and understand. This is because a user program has different segments such as main program, procedure, functions, local variables, global variables, common block, stack, symbol table, arrays, and other data structures and so on. All the segments define in a user program have specific purpose and that is why these segments are of variable length. Also user is not concerned where and how these elements are stored in memory. Using segmentation the logical address space (where user program resides), is divided into segments where each segments is of variable length and different segments can be stored anywhere in memory. Thus user defined segmentation more easy to relate with their program without going into details of how these segmentation are managed. The logical address space where user program resides from user outlook is shown in figure.



re 4.15: User Outlook of the Logical Address Space where User Program Resides

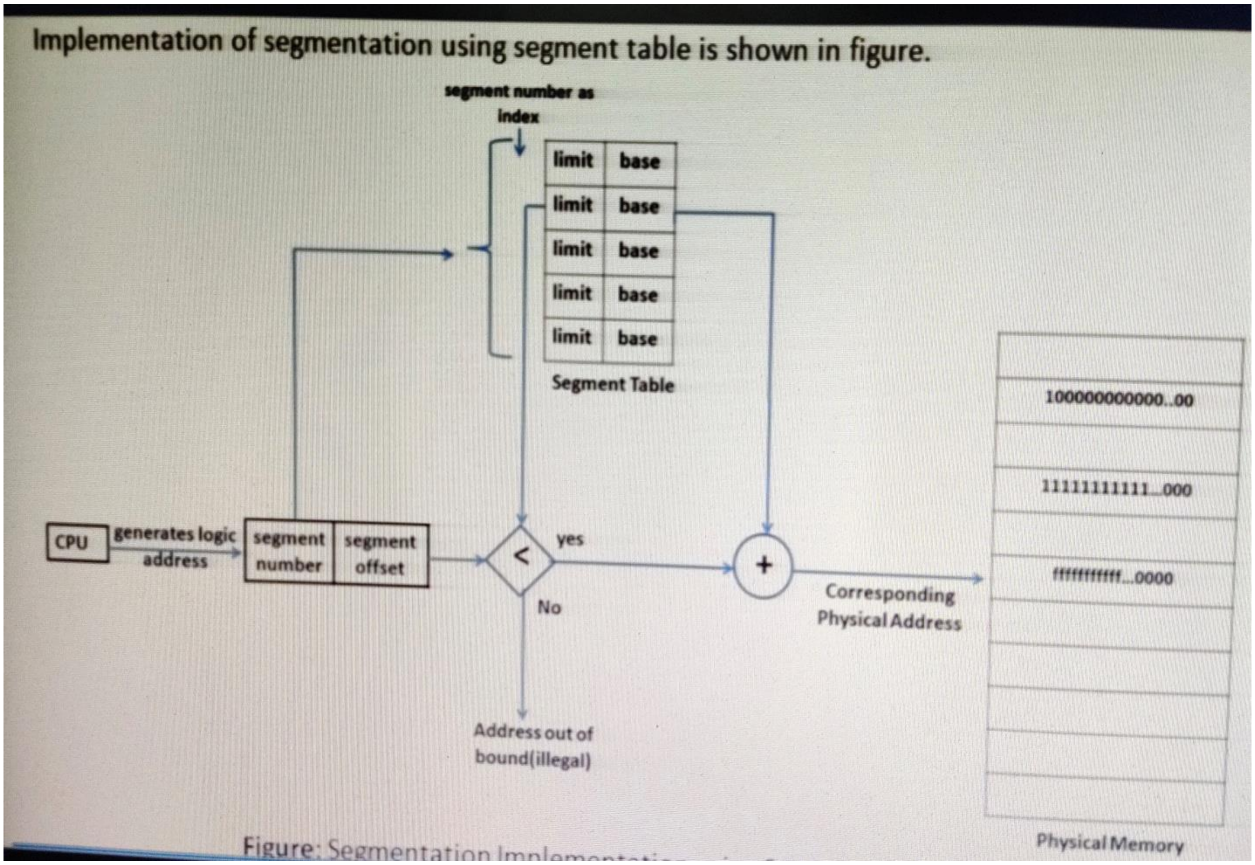
Segmentation Implementation:

In segmentation, each segment in the logical address space has a specific number and the length associated with it. Each segment has starting address (base) and limit that seats the range that seats the range of that segment. In paging, user specifies the logical address which was divides into page number and page offset by paging H/W, but in segmentation the user specifies the logical address using two dimensions:

<Segment number, segment offset>

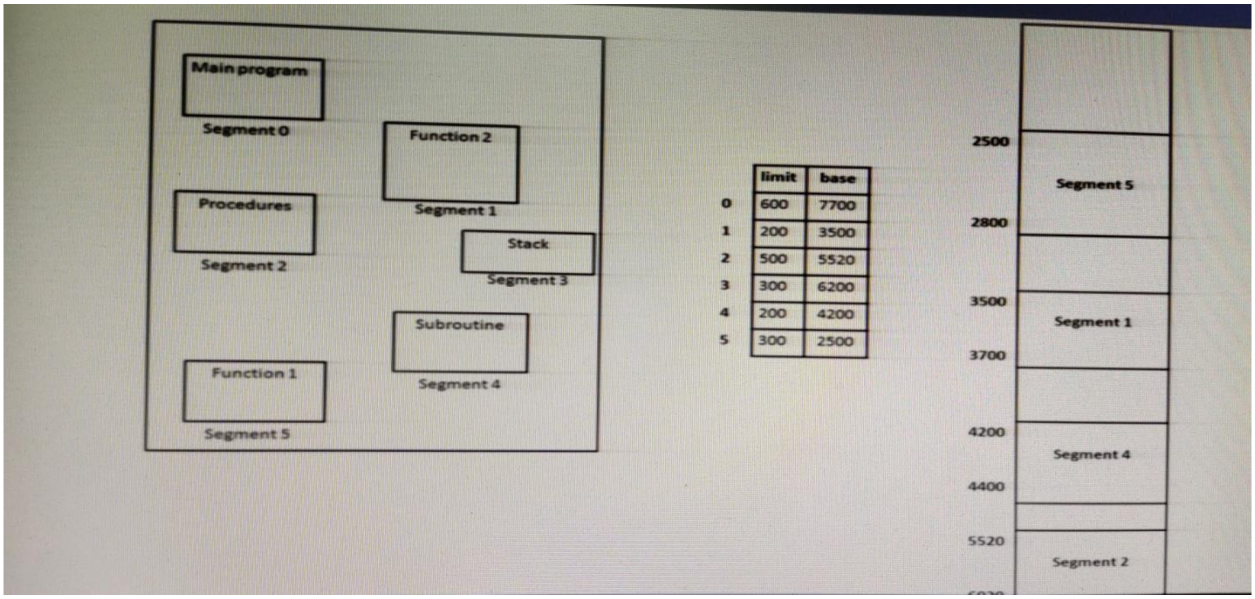
Thus in segmentation two –dimensional user defined logical is to be mapped into one –dimensional physical address. For this segment table is used .Each entry in the segment table has:

1. Base that points the starting address of the segment in physical memory.
2. Limit that specifies the length of the segment.

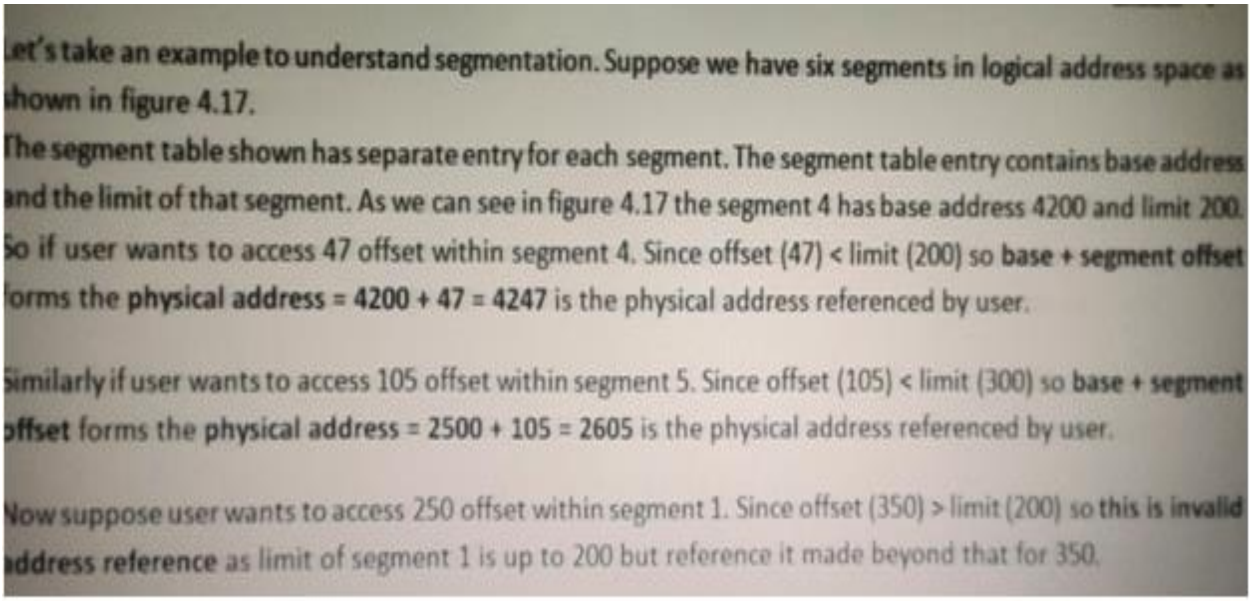


Implementation of segmentation using segment table is shown in figure.

Figure: Segmentation Implementati



	limit	base
0	600	7700
1	200	3500
2	500	5520
3	300	6200
4	200	4200
5	300	2500



Let's take an example to understand segmentation. Suppose we have six segments in logical address space as shown in figure 4.17.

The segment table shown has separate entry for each segment. The segment table entry contains base address and the limit of that segment. As we can see in figure 4.17 the segment 4 has base address 4200 and limit 200. So if user wants to access 47 offset within segment 4. Since offset (47) < limit (200) so **base + segment offset** forms the **physical address = 4200 + 47 = 4247** is the physical address referenced by user.

Similarly if user wants to access 105 offset within segment 5. Since offset (105) < limit (300) so **base + segment offset** forms the **physical address = 2500 + 105 = 2605** is the physical address referenced by user.

Now suppose user wants to access 250 offset within segment 1. Since offset (350) > limit (200) so **this is invalid address reference** as limit of segment 1 is up to 200 but reference it made beyond that for 350.

Segmentation Disadvantage:

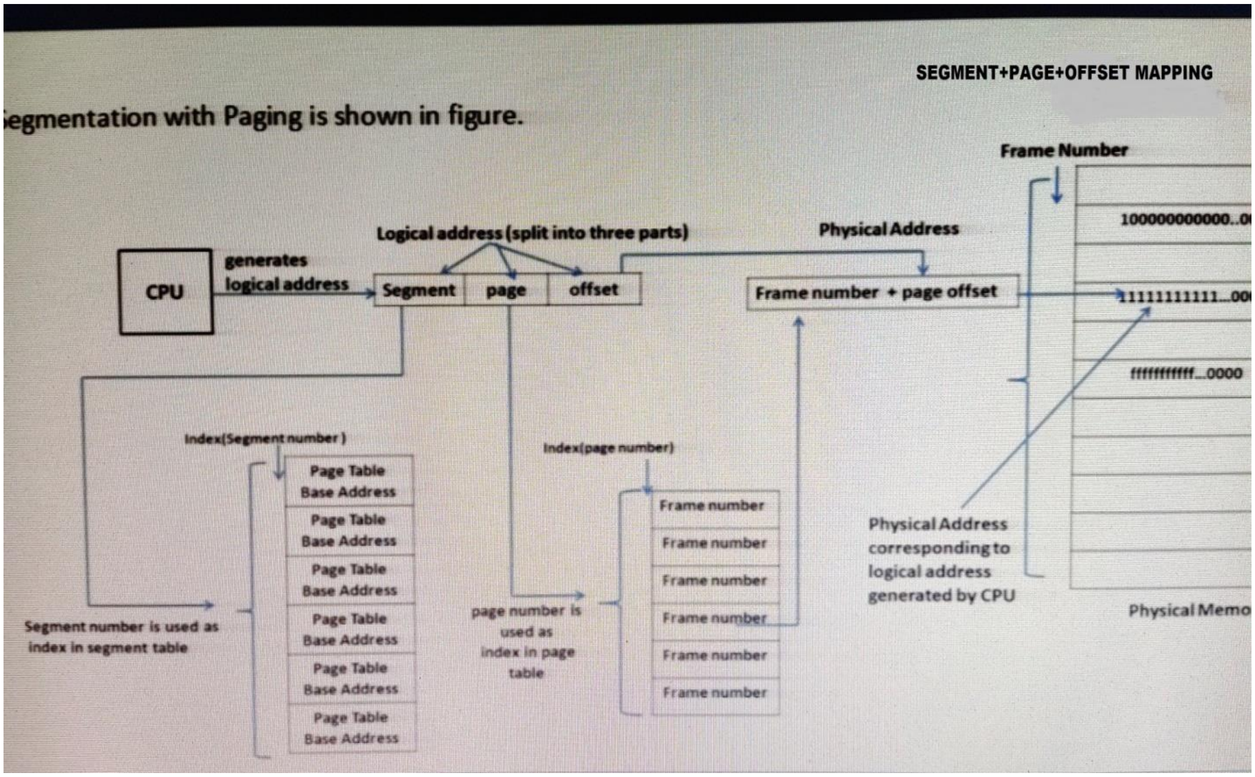
1. Segmentation requires more complicated Hardware for address translation than paging.
2. Segmentation suffers from external fragmentation. Paging only yields a small internal fragmentation.

Paged segmentation (or segmentation with paging):

To take advantage both paging and segmentation some system combines both of these approaches. This approach is called paged segmentation or we can say segmentation with paging. In this technique, *segment is viewed as a collection of pages*. Logical address generated by CPU is divided into three parts- the segment, the page and the offset, this is shown in figure.

Segment	Page	Offset

Logical address division in paged segmentation



- The segment is used as an index is segment table. Entry in the segment table contains the base address of the page table.

- Page number is used as an index in a page table and selects an entry within page table. Page table is used to stored frame number of each page in physical memory. This ***frame number is actually the base address of the page***. This frame number + offset part of logical address forms the physical address. The physical address is the actual address in computer physical memory corresponding to the logical address generated by CPU.

Virtual memory

(Introduction)

Virtual memory is a concept of an OS that virtually increases the apparent size of main memory and gives liberty of user (programmer) to write programs without worrying about the size of physical memory. The user has an illusion of an extremely large main memory but in actual only a limited memory is available. The user actually uses address and space of virtually memory which is then translated (mapped) into corresponding main memory space. The address generated and referenced by user program is called virtual address and the collection of virtual address forms virtual address space. Similarly, the address of main memory is called physical address and collection of physical address is called physical address space.

The OS implement virtual memory concept by loading only that portion of the user program at a time from disk storage(2^{ndry} memory) into main memory that is currently need to be executed instead of loading full program in main memory. Also that portion of user program that is not currently required is temporarily full program in main memory. Also that portion of user program that is not currently required is temporarily transferred back from main memory to disk storage (2^{ndry} memory) to make space or other program. For example: following are situations when entire program is not required to be loaded fully in main memory.

- Parts of the program called ***error handing routines*** are used when an error occurs.
- Some functions and procedures of a program may be used seldom.

- Many data structures like arrays, structures, tables etc are assigned a fixed size of memory space in user programs but actually only a small amount of memory assigned to these data structures is used.

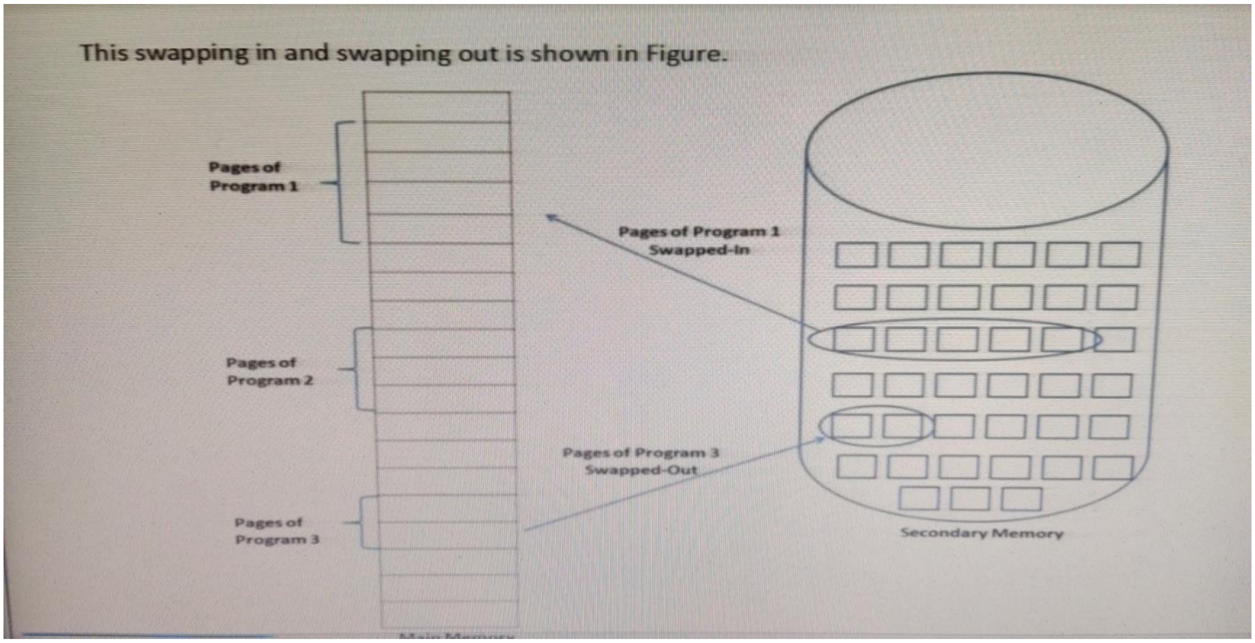
Thus virtual memory gives the facility to execute a program by loading only portion of the program in main memory instead of entire program. This concept provides many benefits:

- The user (programmer) is no longer be bounded by the amount of main memory that is available and can write programs without worrying about the size of memory.
- Since each user program is loaded in portion into main memory thus taking less memory space so more programs could reside in main memory and execute simultaneously. This will lead to efficient CPU utilization and throughput and will give overall good system performance.

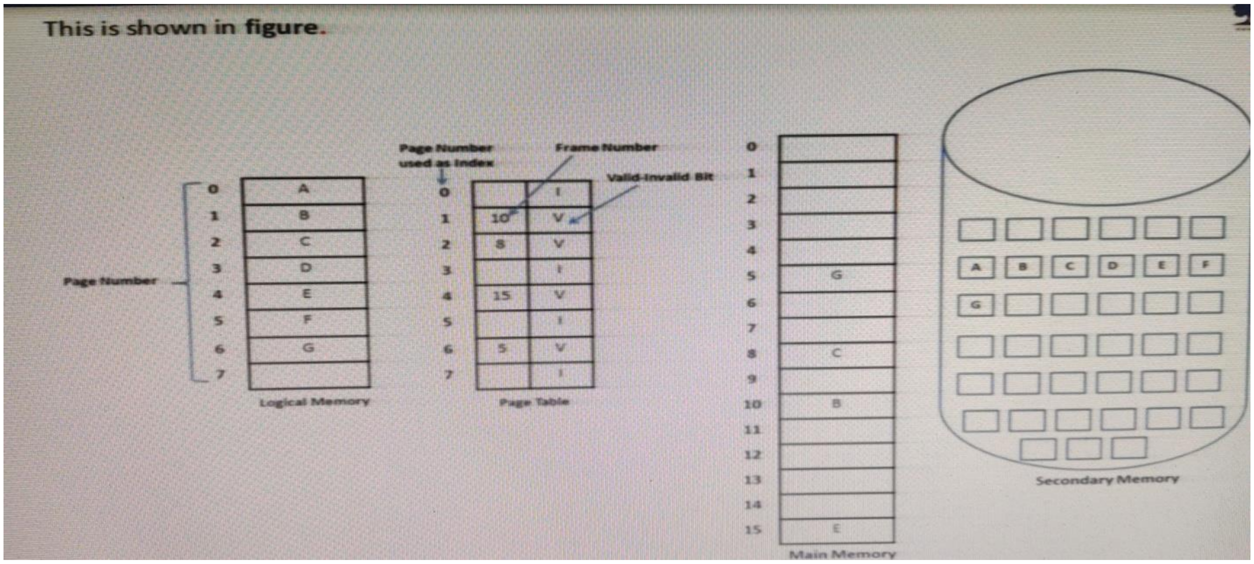
Virtually memory is generally implemented by demand paging. It can also implement in a segmentation system. Demand segmentation can also be used to provide virtual memory.

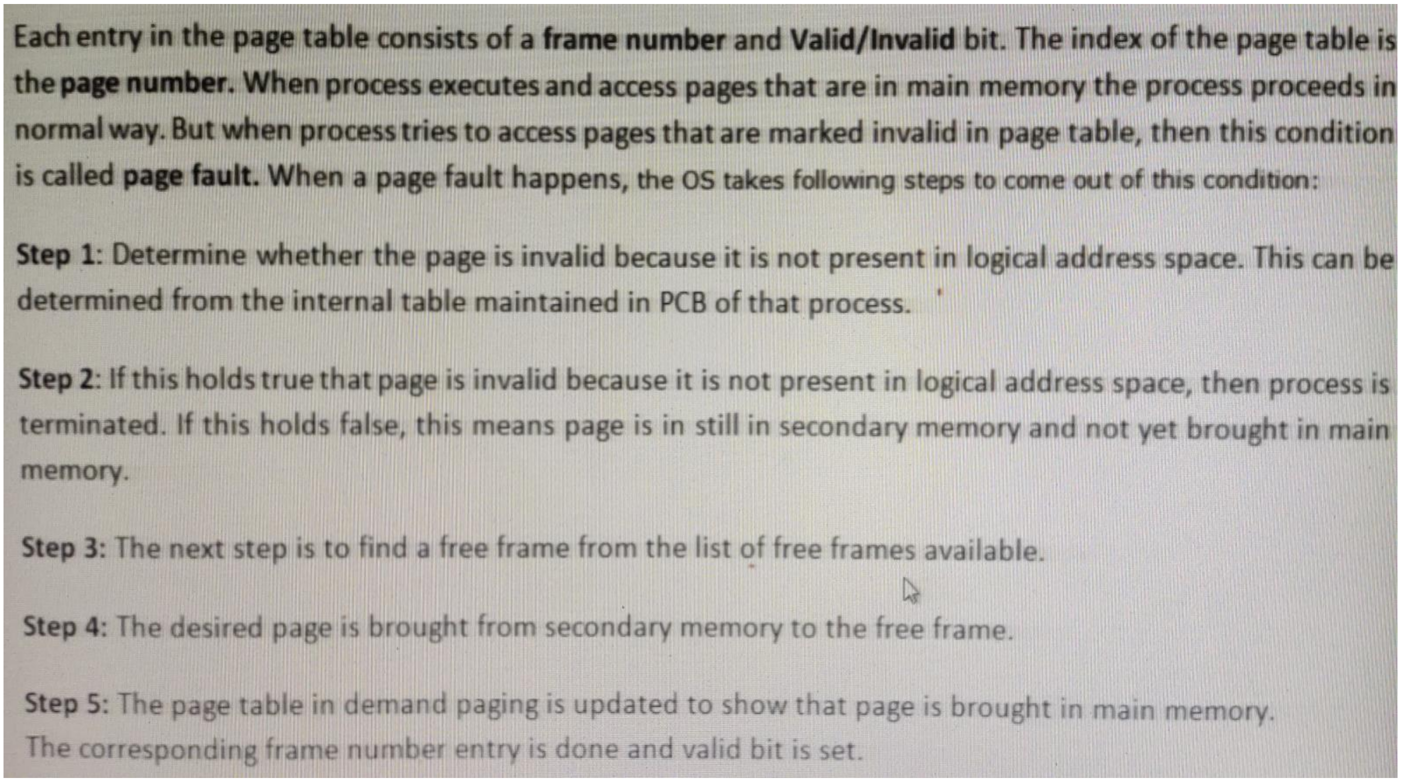
Demand Paging:

The concept of virtual memory is generally implemented by demand paging .A demand paging system is pretty like a paging system but with additional feature of swapping. The user process resides in 2^{ndry} memory and is a considered as a set of pages. The basic concept of demand paging is when we want to execute a process, than instead of bringing (swapping in) the entire process from 2^{ndry} memory into main memory, we use a *lazy swapper* called pager which swaps only those pages into memory that are needed currently. Thus, pages that are not needed in process execution are not brought into main memory. This considerably reduces the time required for swapping and the amount of physical memory needed by a process.


This swapping in and swapping out is shown in Figure.

To implement demand paging Hardware support is required to identify which pages are in memory and which pages is on the disk. To do this page identification, method of ***valid –invalid*** bit is used. Each page in main memory has associated bit with it. The pages that are currently in main memory and are legally valid, their corresponding bit is set to "**valid**" (V). The pages that are currently not in main memory (it is in 2^{ndry} memory) or legally not valid, their corresponding bit is set to "**Invalid**"(I).A page is invalid if it is not present in logical address space. The 2ndary memory holds the pages that are not in main memory and swapping in and out of pages are done between main and 2ndary memory.


This is shown in **figure.**



Each entry in the page table consists of a **frame number** and **Valid/Invalid** bit. The index of the page table is the **page number.** When process executes and access pages that are in main memory the process proceeds in normal way. But when process tries to access pages that are marked invalid in page table, then this condition is called **page fault.** When a page fault happens, the OS takes following steps to come out of this condition:

Step 1: Determine whether the page is invalid because it is not present in logical address space. This can be determined from the internal table maintained in PCB of that process.

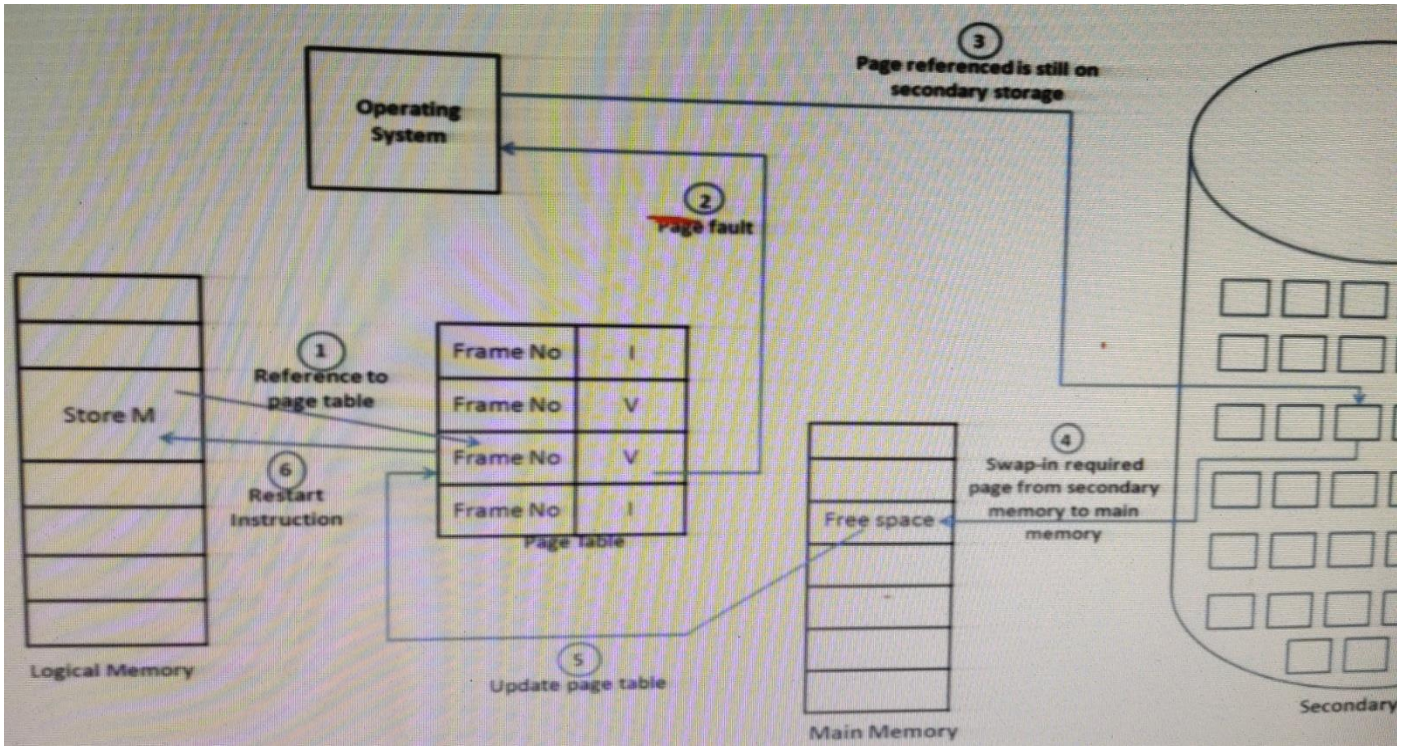
Step 2: If this holds true that page is invalid because it is not present in logical address space, then process is terminated. If this holds false, this means page is in still in secondary memory and not yet brought in main memory.

Step 3: The next step is to find a free frame from the list of free frames available.

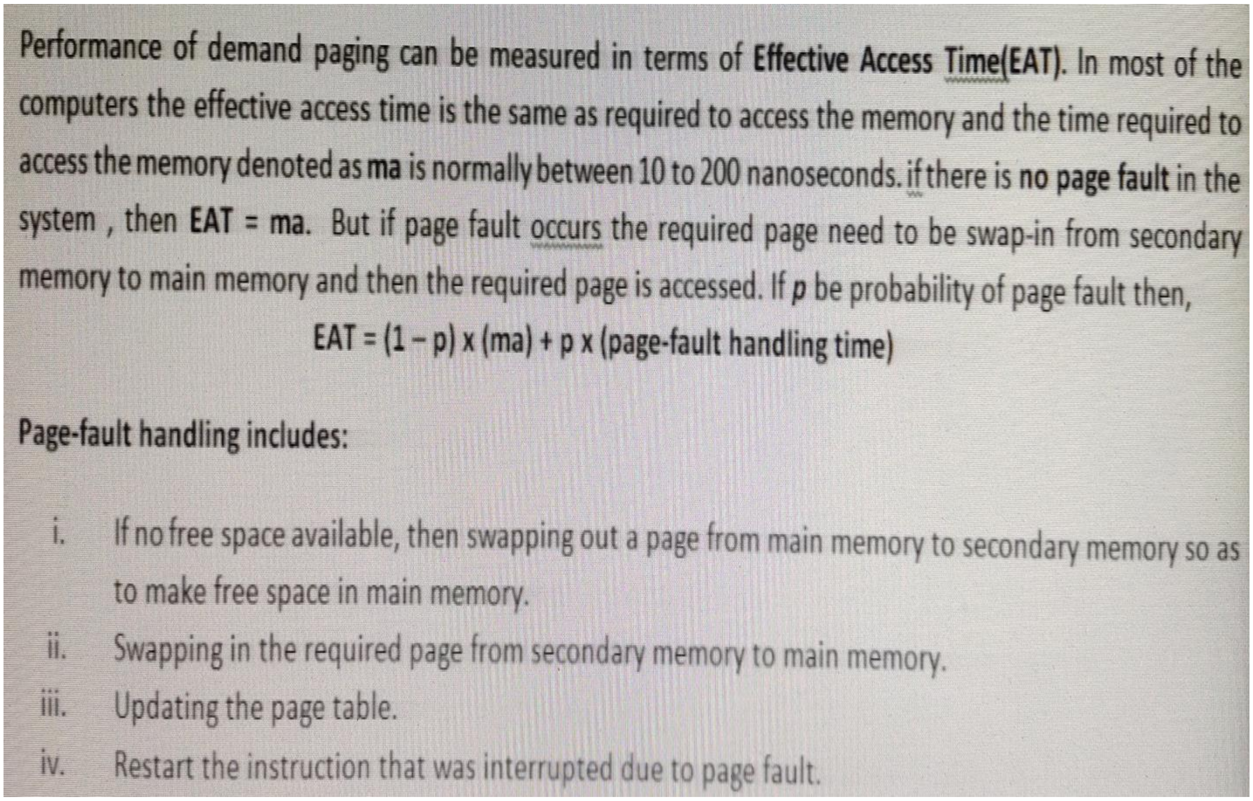
Step 4: The desired page is brought from secondary memory to the free frame.

Step 5: The page table in demand paging is updated to show that page is brought in main memory. The corresponding frame number entry is done and valid bit is set.

Step6. The instruction that was interrupted due to page fault is now restarted as the required page is now available in the main memory.



Performance of Demand Paging:



Performance of demand paging can be measured in terms of **Effective Access Time(EAT)**. In most of the computers the effective access time is the same as required to access the memory and the time required to access the memory denoted as **ma** is normally between 10 to 200 nanoseconds. if there is **no page fault** in the system , then **EAT = ma.** But if page fault occurs the required page need to be swap-in from secondary memory to main memory and then the required page is accessed. If ***p*** be probability of page fault then,

EAT = (1 – p) x (ma) + p x (page-fault handling time)

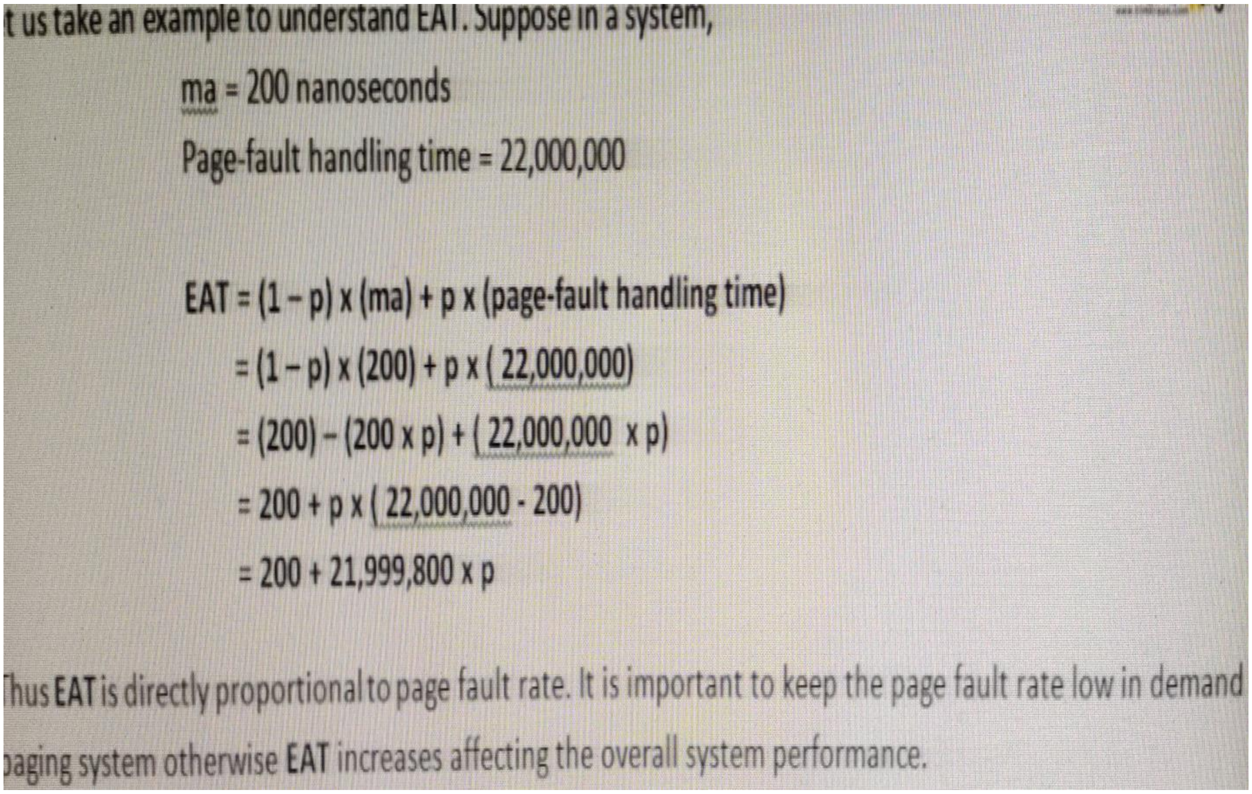
Page-fault handling includes:

i. If no free space available, then swapping out a page from main memory to secondary memory so as to make free space in main memory.

ii. Swapping in the required page from secondary memory to main memory.

iii. Updating the page table.

iv. Restart the instruction that was interrupted due to page fault.



t us take an example to understand EAT. Suppose in a system,

ma = 200 nanoseconds

Page-fault handling time = 22,000,000

EAT = (1 – p) x (ma) + p x (page-fault handling time)

= (1 – p) x (200) + p x (22,000,000)

= (200) – (200 x p) + (22,000,000 x p)

= 200 + p x (22,000,000 - 200)

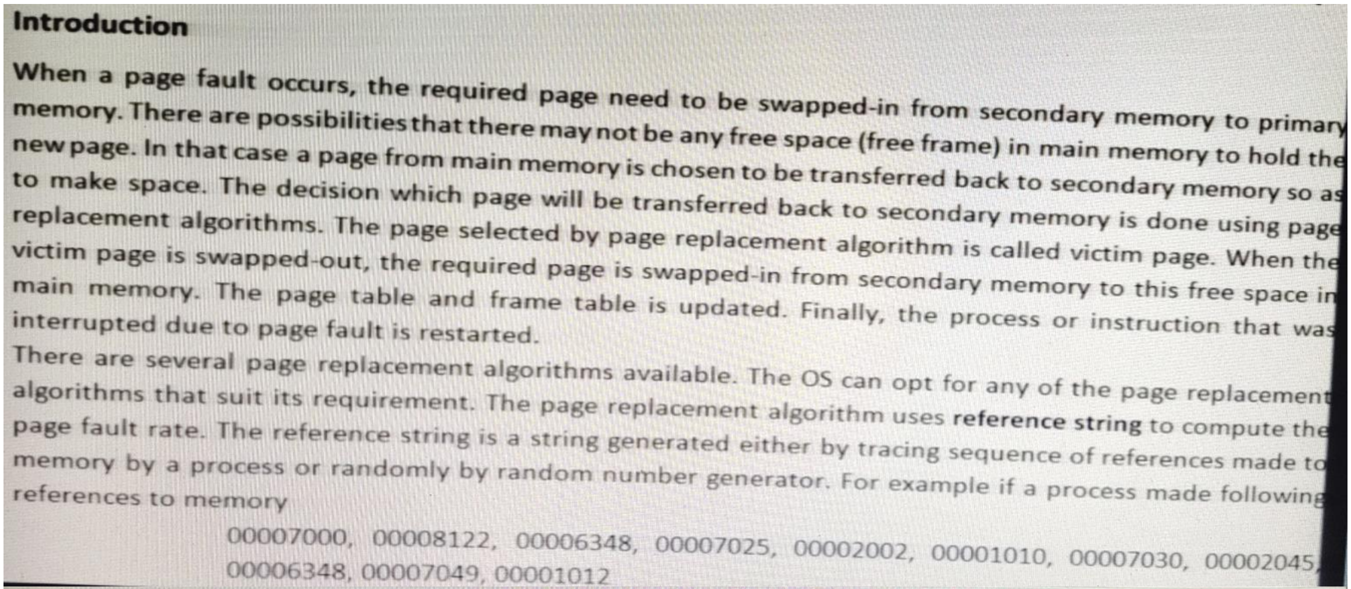
= 200 + 21,999,800 x p

hus **EAT** is directly proportional to page fault rate. It is important to keep the page fault rate low in demand aging system otherwise **EAT** increases affecting the overall system performance.

Page replacement Algorithms

- Introduction
- FIFO Page replacement Algorithm
- Optimal page replacement algorithm
- LRU page replacement algorithm
- LRU –Approximation Page replacement
- Counting –Based page replacement
- Page-buffering Algorithms.

Introduction:

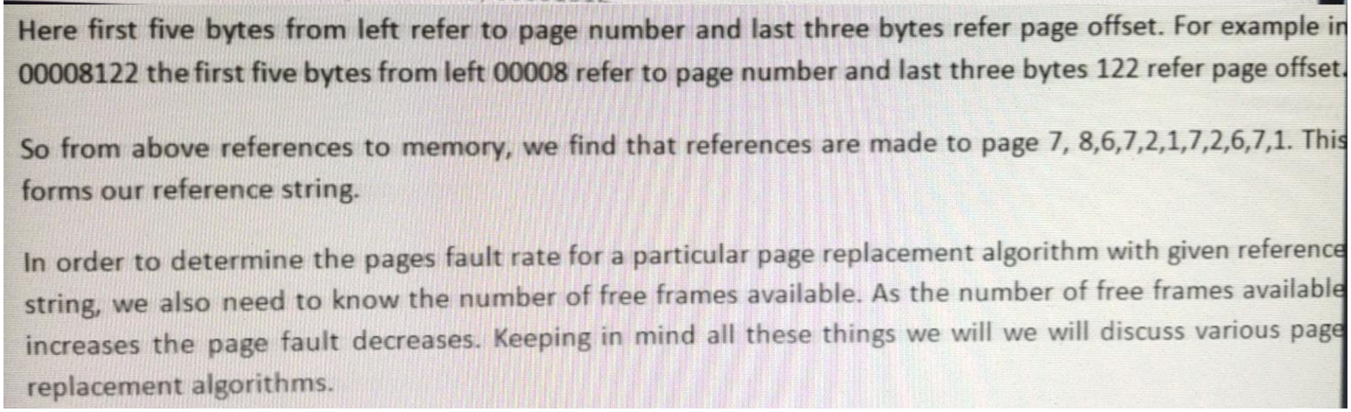


Introduction

When a page fault occurs, the required page need to be swapped-in from secondary memory to primary memory. There are possibilities that there may not be any free space (free frame) in main memory to hold the new page. In that case a page from main memory is chosen to be transferred back to secondary memory so as to make space. The decision which page will be transferred back to secondary memory is done using page replacement algorithms. The page selected by page replacement algorithm is called victim page. When the victim page is swapped-out, the required page is swapped-in from secondary memory to this free space in main memory. The page table and frame table is updated. Finally, the process or instruction that was interrupted due to page fault is restarted.

There are several page replacement algorithms available. The OS can opt for any of the page replacement algorithms that suit its requirement. The page replacement algorithm uses **reference string** to compute the page fault rate. The reference string is a string generated either by tracing sequence of references made to memory by a process or randomly by random number generator. For example if a process made following references to memory

00007000, 00008122, 00006348, 00007025, 00002002, 00001010, 00007030, 00002045, 00006348, 00007049, 00001012



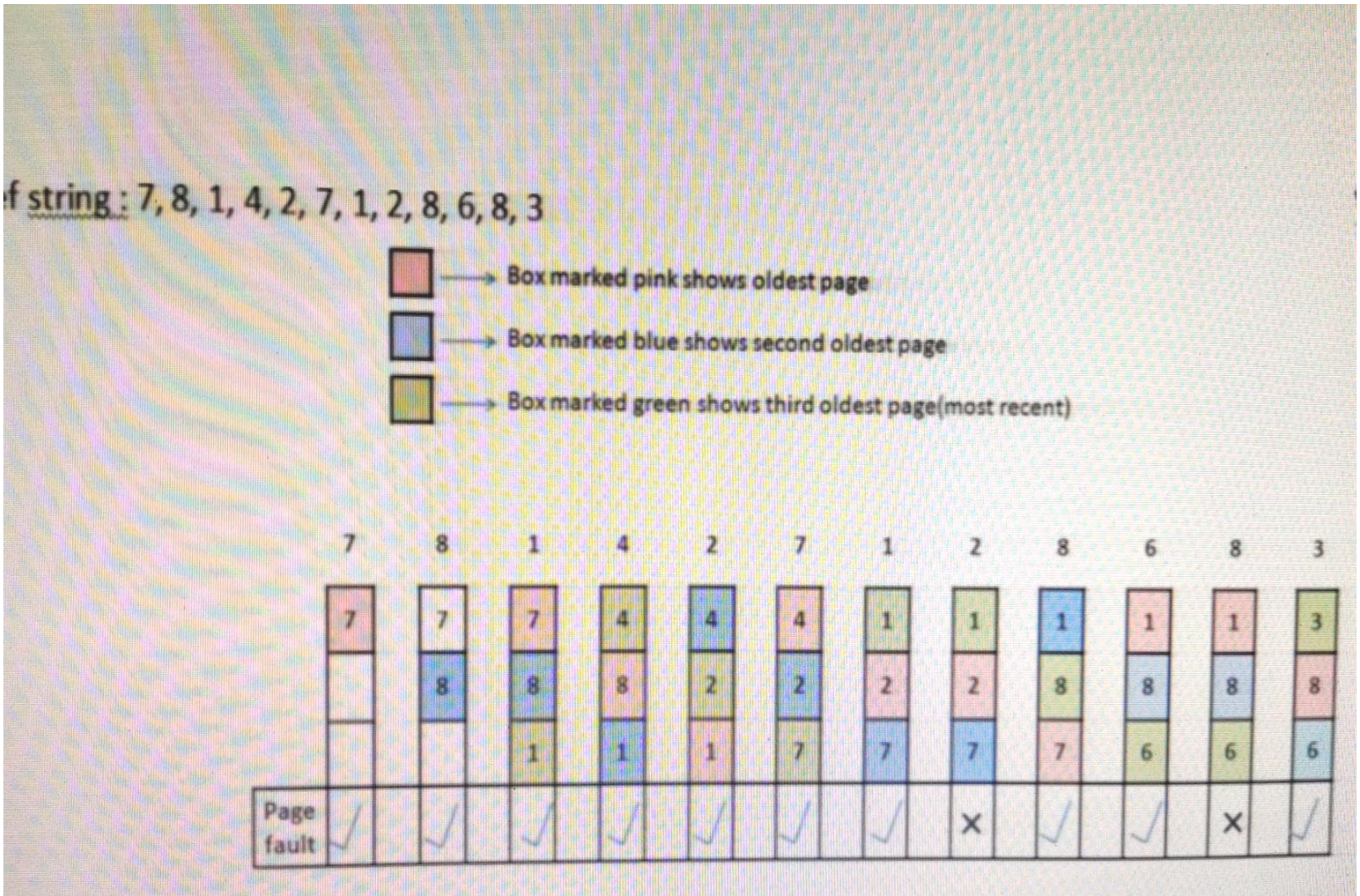
Here first five bytes from left refer to page number and last three bytes refer page offset. For example in 00008122 the first five bytes from left 00008 refer to page number and last three bytes 122 refer page offset.

So from above references to memory, we find that references are made to page 7, 8,6,7,2,1,7,2,6,7,1. This forms our reference string.

In order to determine the pages fault rate for a particular page replacement algorithm with given reference string, we also need to know the number of free frames available. As the number of free frames available increases the page fault decreases. Keeping in mind all these things we will we will discuss various page replacement algorithms.

FIFO PAGE REPLACEMNT ALGORITHM:

The First-in First –Out (FIFO) page replacement algorithm is one of the easy to implement algorithm. As the name suggest the operating system maintains a FIFO queue to store pages in memory. The pages are arranged in FIFO queue according to their arrival. As the new page is brought from 2^{ndary} memory, it is placed at the tail (end) of the queue. When a page needs to be swapped out, the page at the head (front) of the queue (the oldest page) is selected.



f string : 7, 8, 1, 4, 2, 7, 1, 2, 8, 6, 8, 3

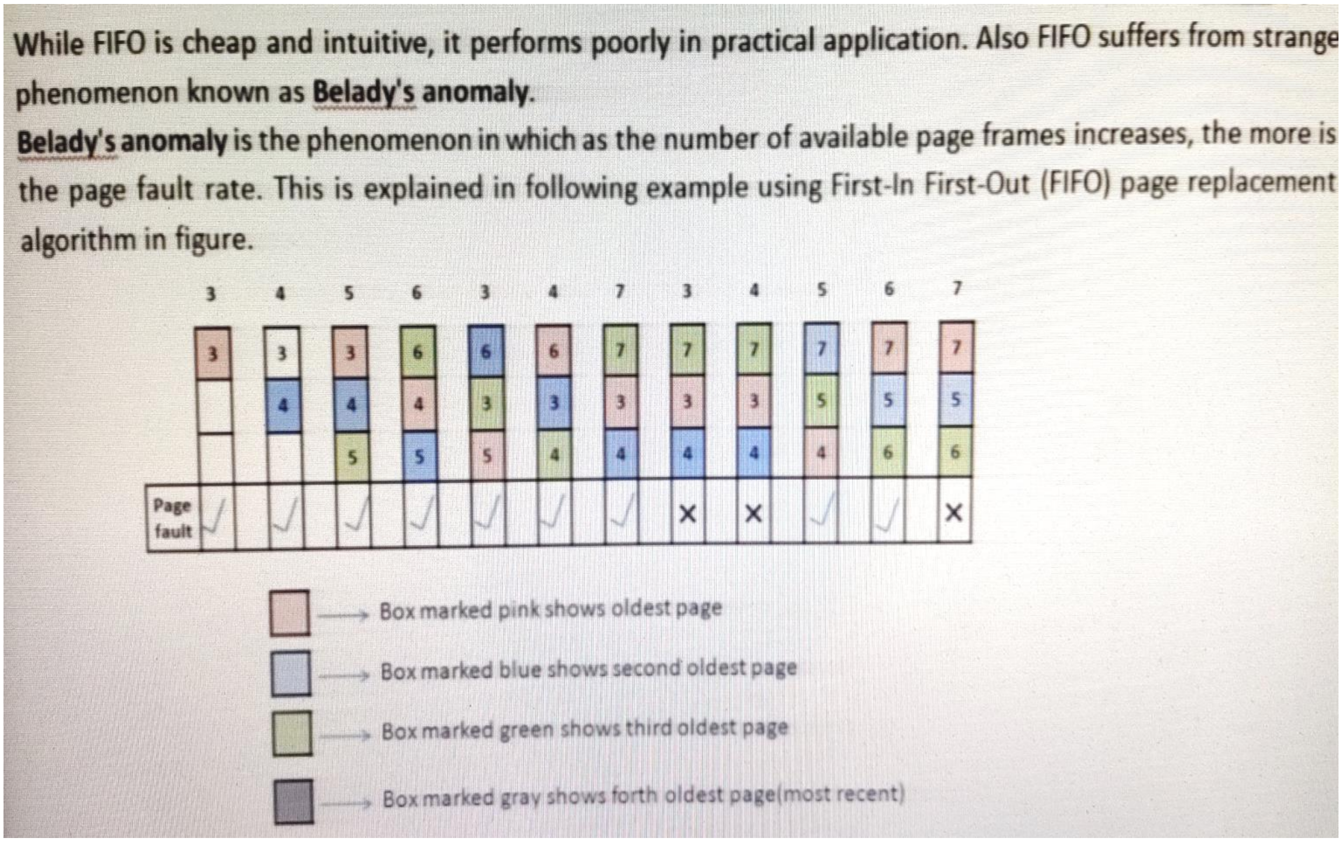
🟥 → Box marked pink shows oldest page

🟦 → Box marked blue shows second oldest page

🟩 → Box marked green shows third oldest page(most recent)

	7	8	1	4	2	7	1	2	8	6	8	3
	7	7	7	4	4	4	1	1	1	1	1	3
		8	8	8	2	2	2	2	8	8	8	8
			1	1	1	7	7	7	7	6	6	6
Page fault	✓	✓	✓	✓	✓	✓	✓	×	✓	✓	×	✓

Figure: Page Replacement Algorithm Example



While FIFO is cheap and intuitive, it performs poorly in practical application. Also FIFO suffers from strange phenomenon known as **Belady's anomaly.**

Belady's anomaly is the phenomenon in which as the number of available page frames increases, the more is the page fault rate. This is explained in following example using First-In First-Out (FIFO) page replacement algorithm in figure.

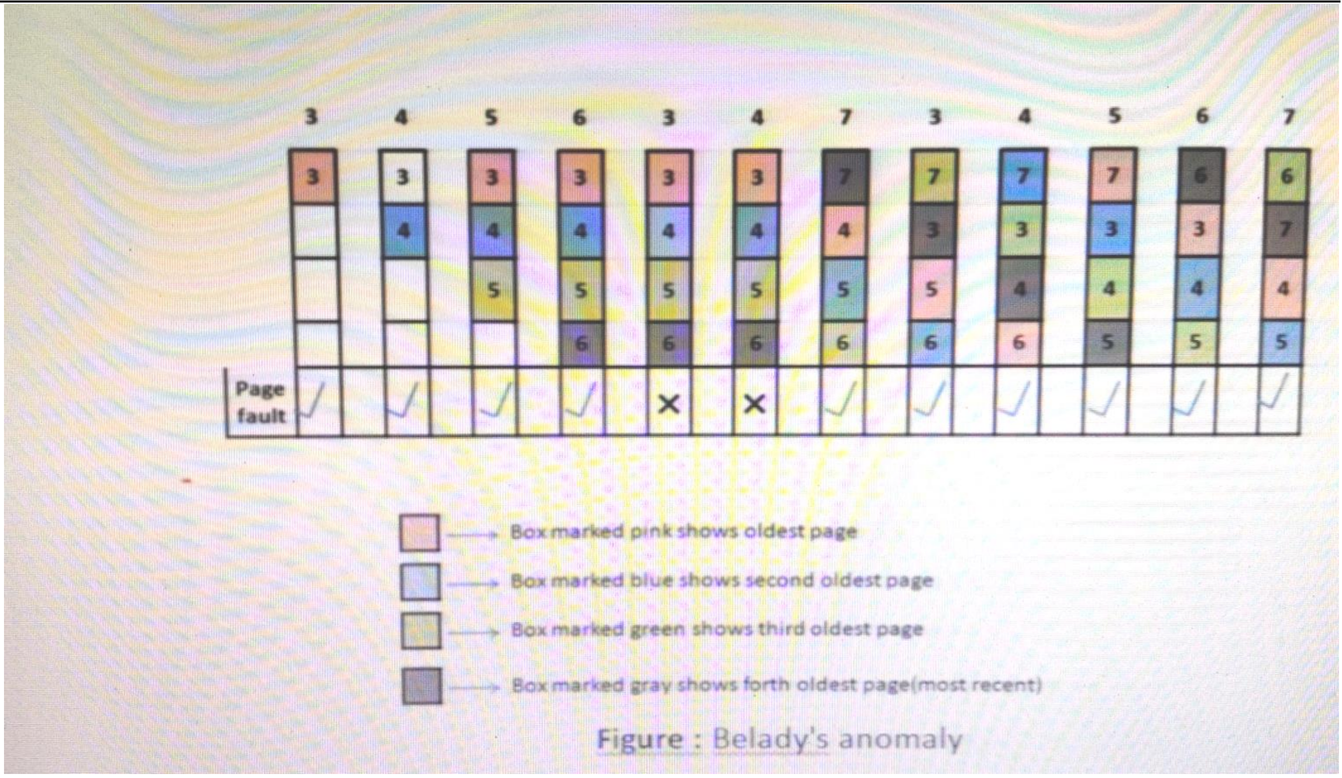
	3	4	5	6	3	4	7	3	4	5	6	7
	3	3	3	6	6	6	7	7	7	7	7	7
		4	4	4	3	3	3	3	3	5	5	5
			5	5	5	4	4	4	4	4	6	6
Page fault	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	×

□ → Box marked pink shows oldest page

□ → Box marked blue shows second oldest page

□ → Box marked green shows third oldest page

■ → Box marked gray shows forth oldest page(most recent)



	3	4	5	6	3	4	7	3	4	5	6	7
	3	3	3	3	3	3	7	7	7	7	6	6
		4	4	4	4	4	4	3	3	3	3	7
			5	5	5	5	5	5	4	4	4	4
				6	6	6	6	6	6	5	5	5
Page fault	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓

□ → Box marked pink shows oldest page

□ → Box marked blue shows second oldest page

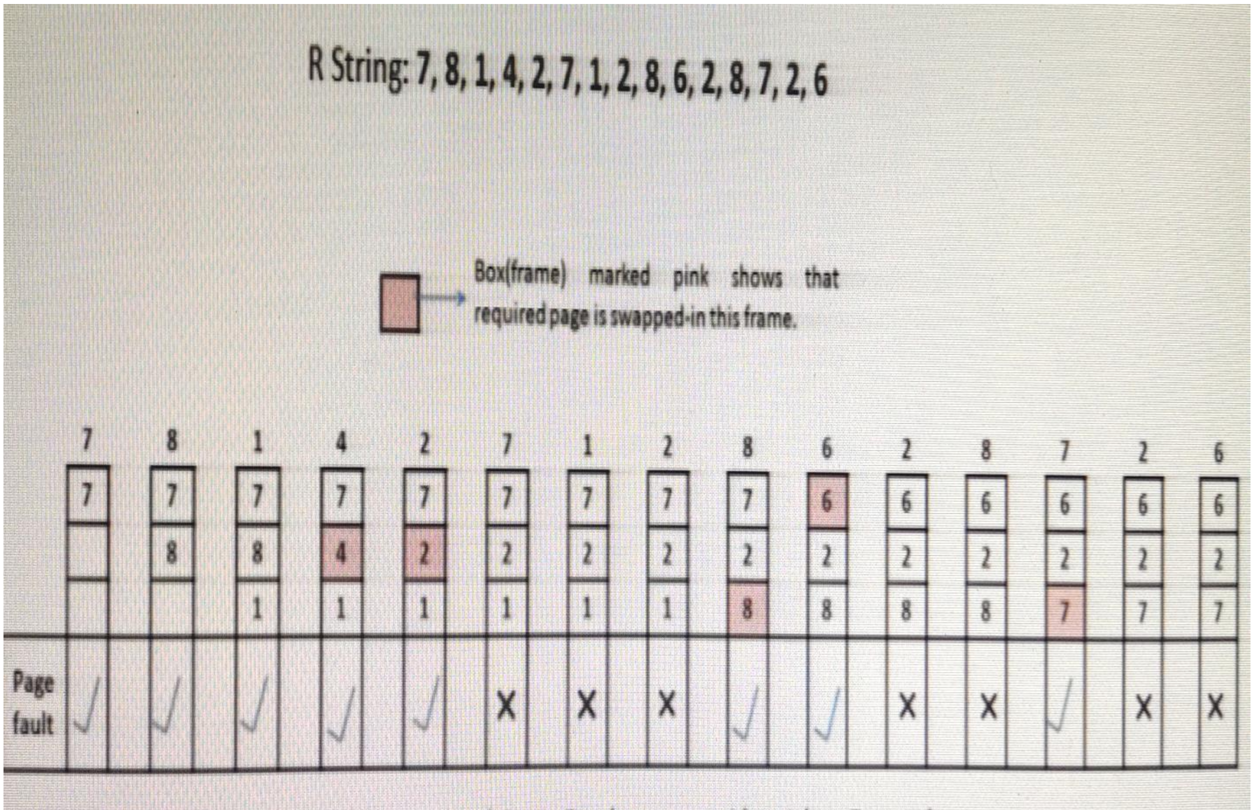
□ → Box marked green shows third oldest page

■ → Box marked gray shows forth oldest page(most recent)

Figure : Belady's anomaly

Optimal page replacement algorithm:

An optimal page replacement algorithm (also known as MIN) has the least possible page fault rate among all the page replacement algorithms. When a page needs to be swapped-out so as to swap in required page from 2^{ndary} memory, the OS choose that page for swapping out, whose next use is predicted late in future. The problem with this algorithm is that it is not practicable to implement in real situations. This is because OS need to know in advance after how much time the page will be referenced again, means OS should know reference string beforehand. Optimal page replacement algorithm does not suffer from ***Belady's anomaly*** phenomenon.



R String: 7, 8, 1, 4, 2, 7, 1, 2, 8, 6, 2, 8, 7, 2, 6

▨ → Box(frame) marked pink shows that required page is swapped-in this frame.

	7	8	1	4	2	7	1	2	8	6	2	8	7	2	6
	7	7	7	7	7	7	7	7	7	6	6	6	6	6	6
		8	8	4	2	2	2	2	2	2	2	2	2	2	2
			1	1	1	1	1	1	8	8	8	8	7	7	7
Page fault	✓	✓	✓	✓	✓	X	X	X	✓	✓	X	X	✓	X	X

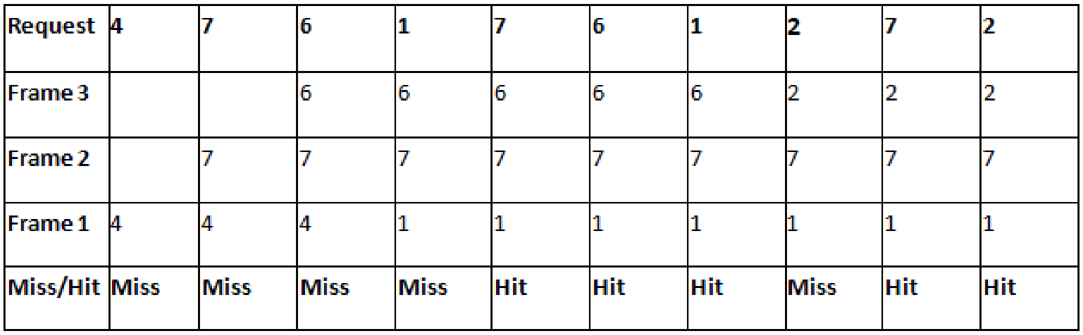
Figure: Optimal Page Replacement Algorithm Example

Numerical on Optimal and FIFO:

Q. Consider a reference string: 4, 7, 6, 1, 7, 6, 1, 2, 7, 2. the number of frames in the memory is 3. Find out the number of page faults respective to:

1. Optimal Page Replacement Algorithm
2. FIFO Page Replacement Algorithm

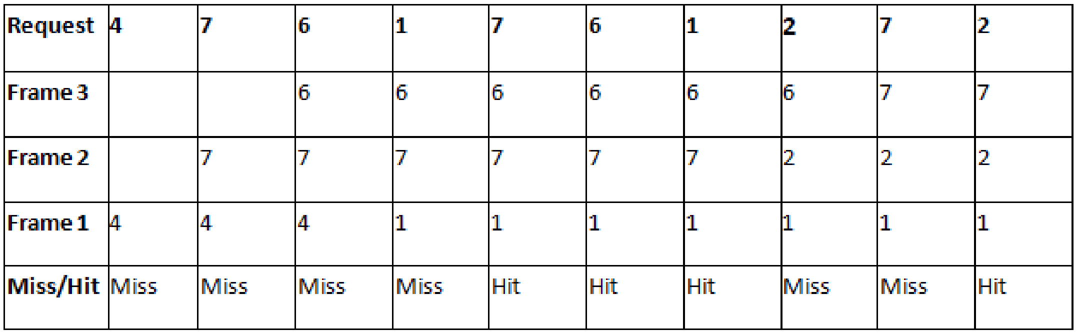
Optimal Page Replacement Algorithm



Request	4	7	6	1	7	6	1	2	7	2
Frame 3			6	6	6	6	6	2	2	2
Frame 2		7	7	7	7	7	7	7	7	7
Frame 1	4	4	4	1	1	1	1	1	1	1
Miss/Hit	Miss	Miss	Miss	Miss	Hit	Hit	Hit	Miss	Hit	Hit

Number of Page Faults in Optimal Page Replacement Algorithm = 5

FIFO Page Replacement Algorithm



Request	4	7	6	1	7	6	1	2	7	2
Frame 3			6	6	6	6	6	6	7	7
Frame 2		7	7	7	7	7	7	2	2	2
Frame 1	4	4	4	1	1	1	1	1	1	1
Miss/Hit	Miss	Miss	Miss	Miss	Hit	Hit	Hit	Miss	Miss	Hit

Number of Page Faults in FIFO = 6

Q. Given reference String is as 0 ,1 ,5,3 ,0 ,1, 4 ,0, 1, 5, 3, 4. Let's analyze the behavior of FIFO algorithm in two cases (take no. of frame 3 and 4).

Case 1: Number of frames = 3

Request	0	1	5	3	0	1	4	0	1	5	3	4
Frame 3			5	5	5	1	1	1	1	1	3	3
Frame 2		1	1	1	0	0	0	0	0	5	5	5
Frame 1	0	0	0	3	3	3	4	4	4	4	4	4
Miss/Hit	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Hit	Hit	Miss	Miss	Hit

Number of Page Faults = 9

Case 2: Number of frames = 4

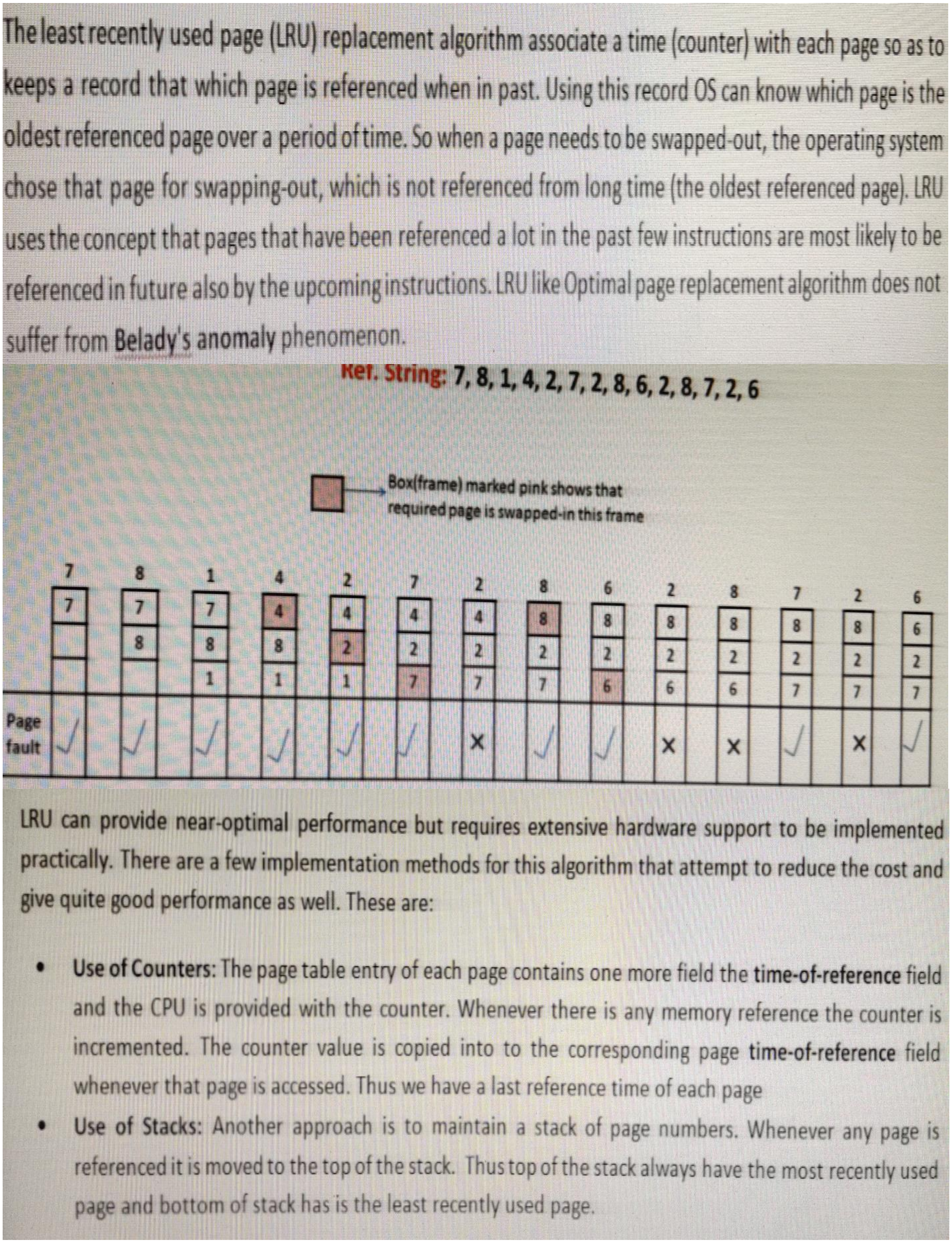
Request	0	1	5	3	0	1	4	0	1	5	3	4
Frame 4				3	3	3	3	3	3	5	5	5
Frame 3			5	5	5	5	5	5	1	1	1	1
Frame 2		1	1	1	1	1	1	0	0	0	0	4
Frame 1	0	0	0	0	0	0	4	4	4	4	3	3
Miss/Hit	Miss	Miss	Miss	Miss	Hit	Hit	Miss	Miss	Miss	Miss	Miss	Miss

Number of Page Faults = 10

Therefore, in this example, the number of page faults is increasing by increasing the number of frames hence this suffers from Belady'sAnomaly.

Q. Given reference string is as 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1. The number of frames in the memory is 4. Find out the number of page faults in case Optimal Page Replacement Algorithm. Also find out hit ratio and miss ratio.

LRU page replacement algorithm:



The least recently used page (LRU) replacement algorithm associate a time (counter) with each page so as to keeps a record that which page is referenced when in past. Using this record OS can know which page is the oldest referenced page over a period of time. So when a page needs to be swapped-out, the operating system chose that page for swapping-out, which is not referenced from long time (the oldest referenced page). LRU uses the concept that pages that have been referenced a lot in the past few instructions are most likely to be referenced in future also by the upcoming instructions. LRU like Optimal page replacement algorithm does not suffer from **Belady's anomaly** phenomenon.

Ref. String: **7, 8, 1, 4, 2, 7, 2, 8, 6, 2, 8, 7, 2, 6**

▨ Box(frame) marked pink shows that required page is swapped-in this frame

	7	8	1	4	2	7	2	8	6	2	8	7	2	6
	7	7	7	4	4	4	4	8	8	8	8	8	8	6
		8	8	8	2	2	2	2	2	2	2	2	2	2
			1	1	1	7	7	7	6	6	6	7	7	7
Page fault	✓	✓	✓	✓	✓	✓	×	✓	✓	×	×	✓	×	✓

LRU can provide near-optimal performance but requires extensive hardware support to be implemented practically. There are a few implementation methods for this algorithm that attempt to reduce the cost and give quite good performance as well. These are:

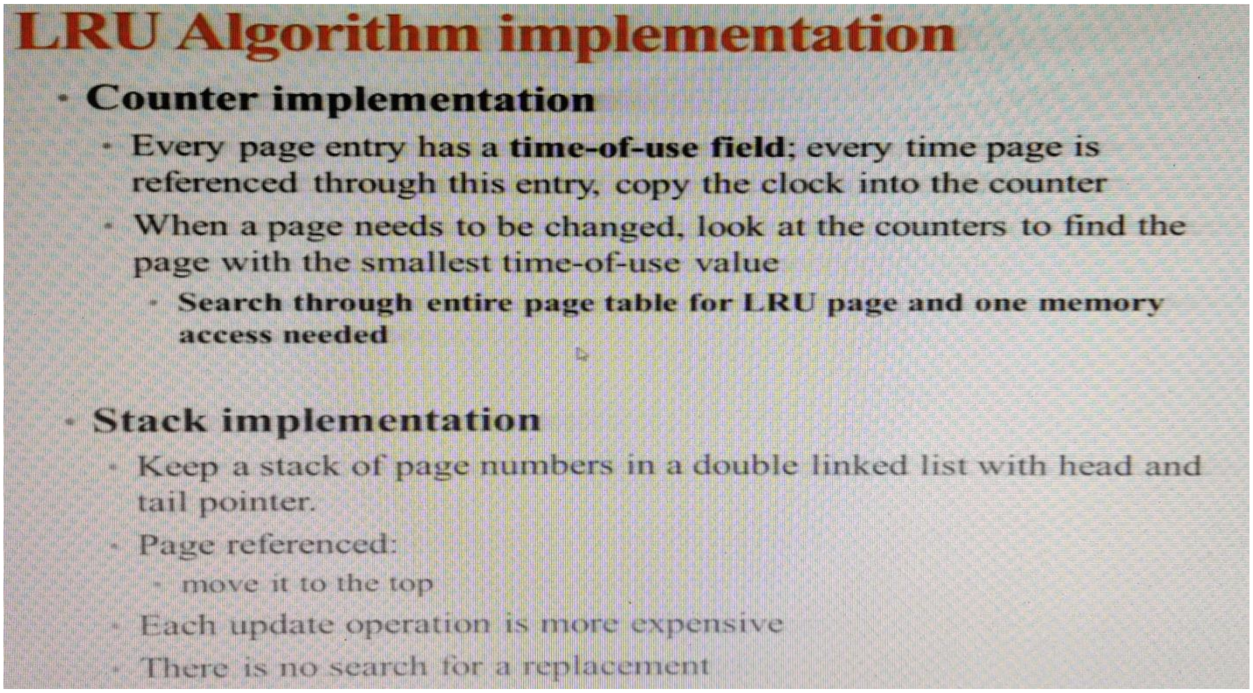
- **Use of Counters:** The page table entry of each page contains one more field the **time-of-reference** field and the CPU is provided with the counter. Whenever there is any memory reference the counter is incremented. The counter value is copied into to the corresponding page **time-of-reference** field whenever that page is accessed. Thus we have a last reference time of each page
- **Use of Stacks:** Another approach is to maintain a stack of page numbers. Whenever any page is referenced it is moved to the top of the stack. Thus top of the stack always have the most recently used page and bottom of stack has is the least recently used page.

Q. Consider a reference string: 4, 7, 6, 1, 7, 6, 1, 2, 7, 2. the number of frames in the memory is 3. Find out the number of page faults respective to: LRU replacement algorithm

Ans:

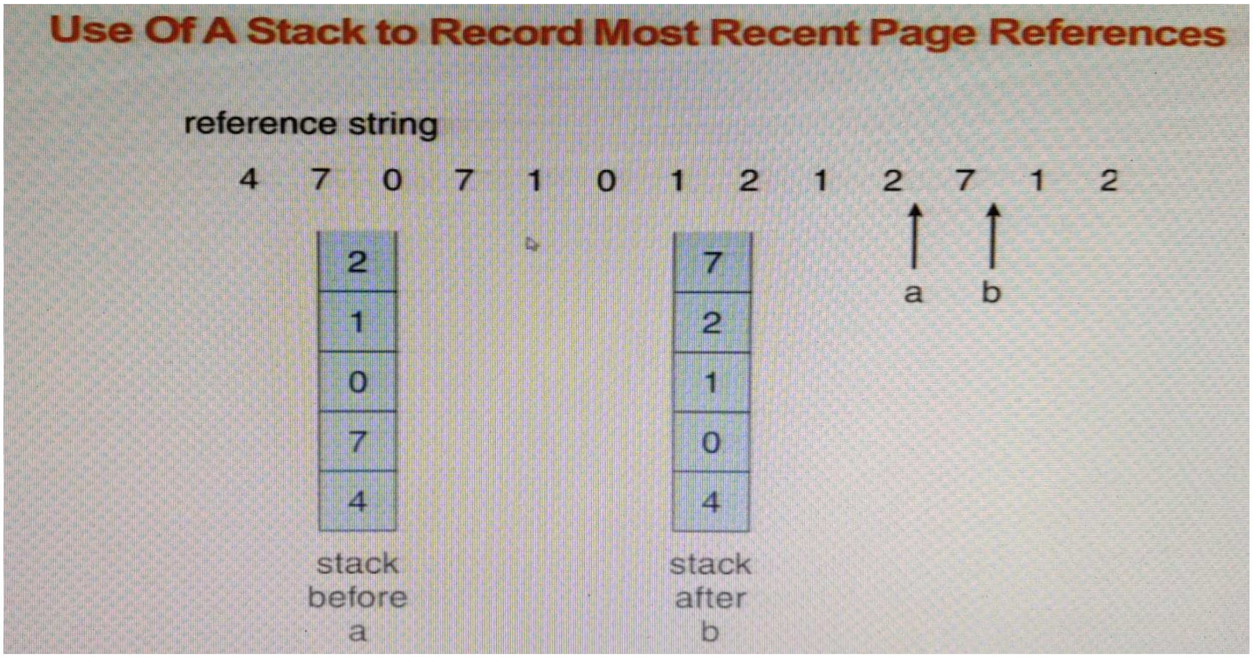
Request	4	7	6	1	7	6	1	2	7	2
Frame 3			6	6	6	6	6	6	7	7
Frame 2		7	7	7	7	7	7	2	2	2
Frame 1	4	4	4	1	1	1	1	1	1	1
Miss/Hit	Miss	Miss	Miss	Miss	Hit	Hit	Hit	Miss	Miss	Hit

Number of Page Faults in LRU = 6



LRU Algorithm implementation

- **Counter implementation**
  - Every page entry has a **time-of-use field**; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to find the page with the smallest time-of-use value
    - **Search through entire page table for LRU page and one memory access needed**
- **Stack implementation**
  - Keep a stack of page numbers in a double linked list with head and tail pointer.
  - Page referenced:
    - move it to the top
  - Each update operation is more expensive
  - There is no search for a replacement

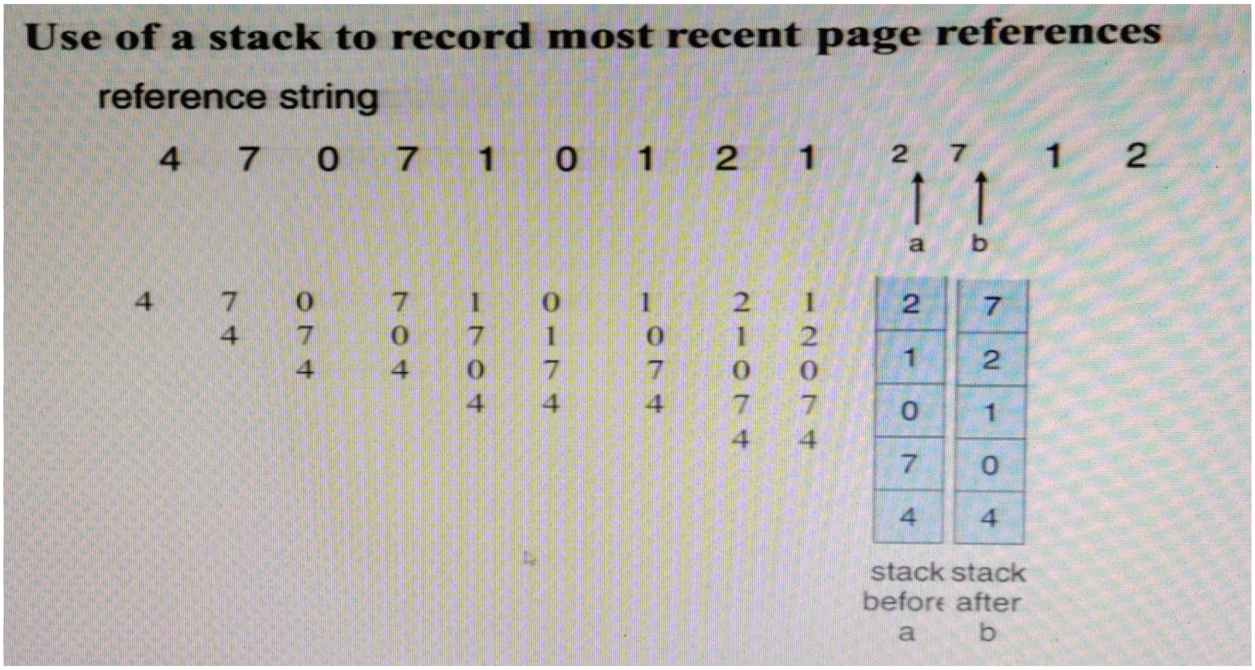


Use Of A Stack to Record Most Recent Page References

reference string

4 7 0 7 1 0 1 2 1 2 7 1 2

2	7
1	2
0	1
7	0
4	4
stack before a	stack after b



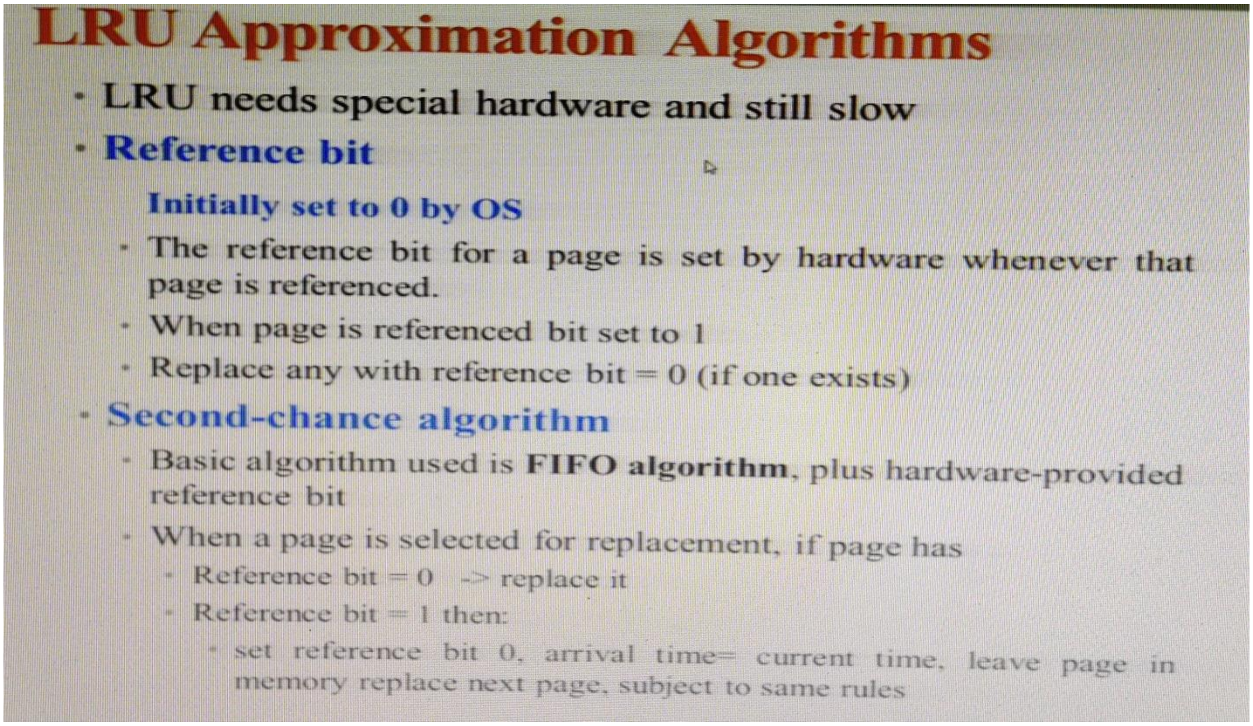
Use of a stack to record most recent page references

reference string

4 7 0 7 1 0 1 2 1 2 7 1 2

4	7	0	7	1	0	1	2	1	2	7
	4	7	0	7	1	0	1	2	1	2
		4	4	0	7	7	0	0	0	1
				4	4	4	7	7	7	0
							4	4	4	4
									stack before a	stack after b

LRU –Approximation Page replacement

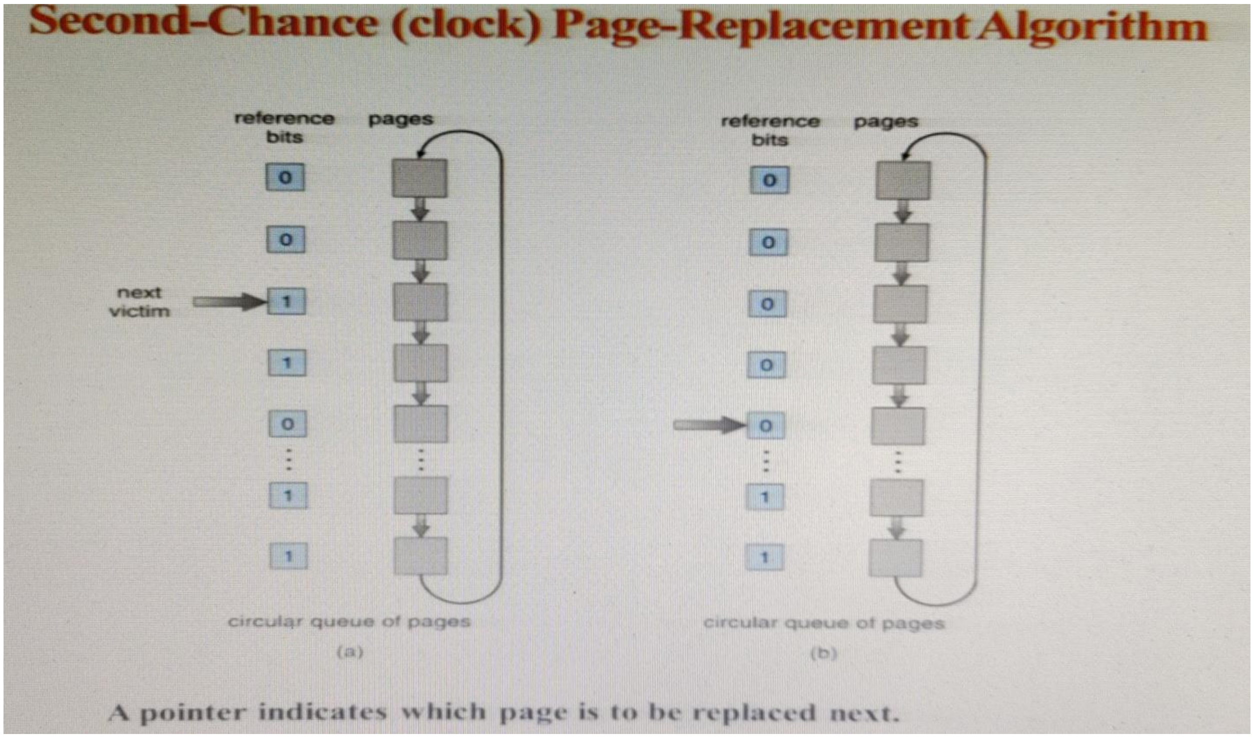


LRU Approximation Algorithms

- LRU needs special hardware and still slow
- **Reference bit**

  Initially set to 0 by OS

  - The reference bit for a page is set by hardware whenever that page is referenced.
  - When page is referenced bit set to 1
  - Replace any with reference bit = 0 (if one exists)
- **Second-chance algorithm**
  - Basic algorithm used is **FIFO algorithm**, plus hardware-provided reference bit
  - When a page is selected for replacement, if page has
    - Reference bit = 0 -> replace it
    - Reference bit = 1 then:
      - set reference bit 0, arrival time= current time, leave page in memory replace next page, subject to same rules

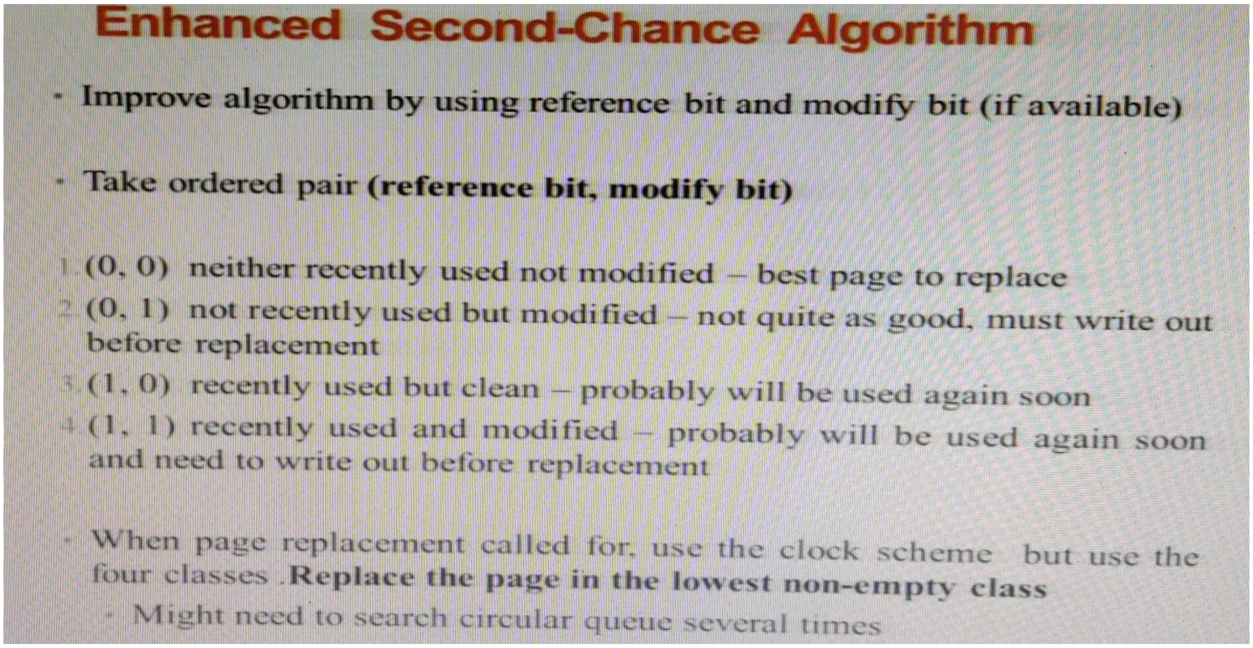


Second-Chance (clock) Page-Replacement Algorithm

A pointer indicates which page is to be replaced next.

Ex: R-string: 2 3 2 1 5 2 4 5 3 2 5 2

Request	2	Set bit	3	Set bit	2	**Set bit**	1	Set bit	5	Set bit	2	Set bit	4	Set bit	5	Set bit	3	Set bit	2	Set bit	5	Set bit	2
F1	2	**0**	2	**0**	**2**	**1**	**2**	1	**2**	**0**	**2**	**1**	**2**	**0**	**2**	**0**	**3**	**0**	**3**	**0**	**3**	**0**	**3**
F2	-		3	**0**	**3**	**0**	**3**	0	**5**	**0**	**5**	**0**	**5**	**0**	**5**	**1**	**5**	**1**	**5**	**0**	**5**	**1**	**5**
F3	-		-		-		**1**	0	**1**	**0**	**1**	**0**	**4**	**0**	**4**	**0**	**4**	**0**	**2**	**0**	**2**	**0**	**2**
Miss/Hit	Miss		Miss		Hit		Miss		Miss		hit		miss		hit		miss		miss		hit		hit

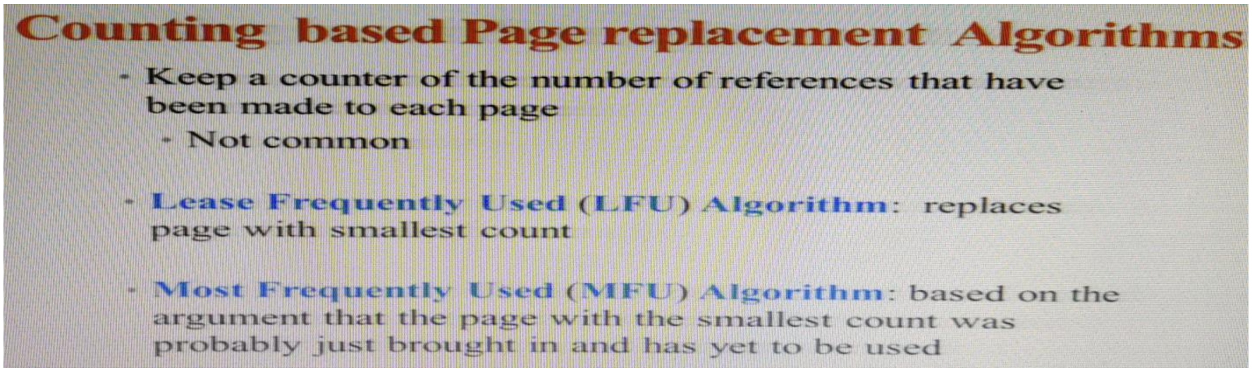


Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bit (if available)
- Take ordered pair **(reference bit, modify bit)**

1. (0, 0) neither recently used not modified – best page to replace
2. (0, 1) not recently used but modified – not quite as good, must write out before replacement
3. (1, 0) recently used but clean – probably will be used again soon
4. (1, 1) recently used and modified – probably will be used again soon and need to write out before replacement

- When page replacement called for, use the clock scheme but use the four classes .**Replace the page in the lowest non-empty class**
  - Might need to search circular queue several times



Counting based Page replacement Algorithms

- Keep a counter of the number of references that have been made to each page
  - Not common
- Lease Frequently Used (LFU) Algorithm: replaces page with smallest count
- Most Frequently Used (MFU) Algorithm: based on the argument that the page with the smallest count was probably just brought in and has yet to be used