UNIT-III (12 Hrs)

Memory-Management Strategies: Introduction, Swapping, Contiguous memory allocation, Paging, Segmentation, Examples. Virtual Memory Management: Introduction, Demand paging, Copy on-write, Page replacement, Frame allocation, Thrashing, Memory-mapped files, Kernel memory allocation, Examples.

MAIN MEMORY

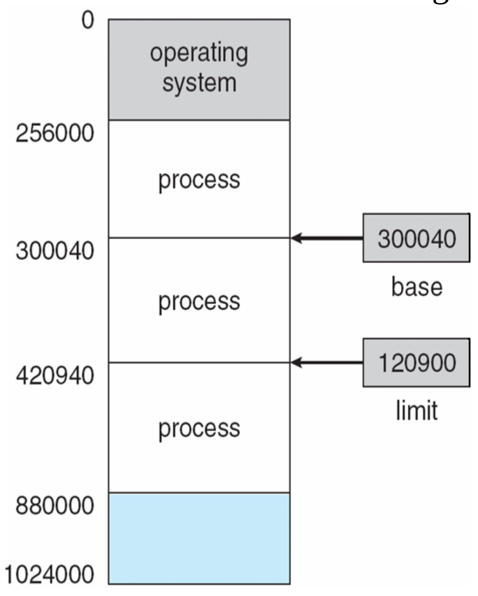
The main purpose of a computer system is to execute programs.
→During the execution of these programs together with the data they must be stored in main memory.
→**Memory consists of a large array of bytes. Each Byte has its own address.**
→CPU fetches instructions from memory according to the value of the program counter.

BASIC HARDWARE

CPU can access data directly only from Main memory and processor registers.
→ Main memory and the Processor registers are called Direct Access Storage Devices.
→Any instructions and data being used by the instructions must be in one of these direct-access storage devices.
→If the data are not in memory then the data must be moved to main memory before the CPU can operate on them.
→Registers that are built into the CPU are accessible within one CPU clock cycle.
→Completing a memory access from main memory may take many CPU clock cycles. Memory access from main memory is done through memory bus.
→In such cases, the processor needs to stall, since it does not have the required data to complete the instruction that it is executing.
→To avoid memory stall, we need to implement Cache memory in between Main memory and CPU.

BASE REGISTER & LIMIT REGISTER

→Each process has a separate memory space that protects the processes from each other. It is fundamental to having multiple processes loaded in memory for concurrent execution.
There are two register that provides protection:
Base Register holds the smallest legal physical memory address.
Limit register specifies the size of the range (i.e. process size).
Example: if the base register holds 300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420939 (inclusive).



→The base and limit registers can be loaded only by the operating system by using a special privileged instruction that can be executed only in kernel mode.
→Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode with the registers.
→Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a trap to the operating system, which treats the attempt as a Fatal Error.
Operating system executing in kernel mode is given unrestricted access to both operating-system memory and users' memory. This provision allows the operating system to do certain tasks such as:

→ Load users' programs into users' memory
→To dump out those programs in case of errors
→To access and modify parameters of system calls
→To perform I/O to and from user memory etc.

Example: A Multiprocessing Operating system must execute context switches, storing the state of one process from the registers into main memory before loading the next process's context from main memory into the registers.

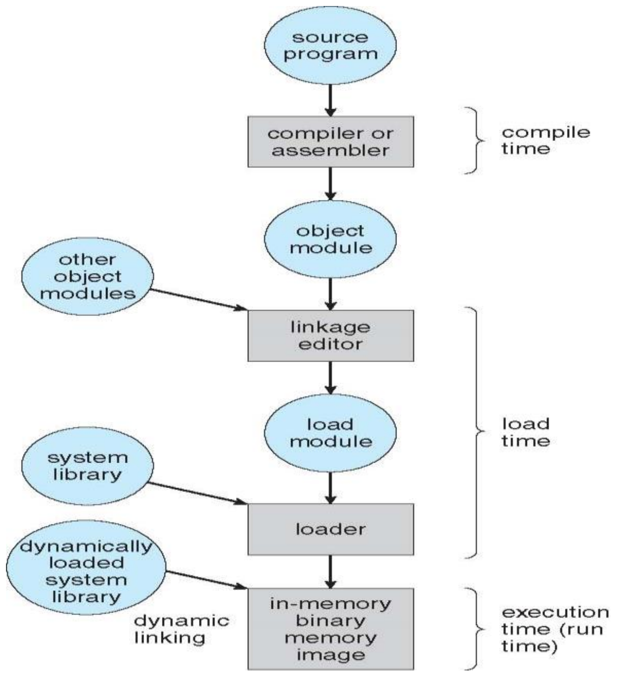
Address Binding

→A program resides on a disk as a binary executable file. The program must be brought into memory and placed within a process for execution.
→The process may be moved between disk and memory during its execution.
→The processes on the disk that are waiting to be brought into memory for execution are put into the Input Queue.
→Addresses may be represented in different ways during these steps.
→Addresses in the source program are generally symbolic, such as the variable count.
· A compiler typically binds these symbolic addresses to Relocatable addresses such as "14 bytes from the beginning of this module".
· The Linkage editor or Loader in turn binds Relocatable addresses to Absolute addresses such as 74014 (i.e. 74000+14=74014).
· Each binding is a mapping from one address space to another address space.
Binding of instructions and data to memory addresses can be done at any of following steps:

Compile time.
If you know at compile time where the process will reside in memory then Absolute code can be generated.

Load time
If it is not known at compile time where the process will reside in memory, then the
compiler must generate Relocatable code.
· In this case, final binding is delayed until load time. If the starting address changes, we need to reload only the user code to incorporate this changed value.

Execution time
If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time.
· Most general-purpose operating systems use this method



SWAPPING

A process must be in Main memory to be executed. A process can be swapped temporarily out of main memory to a backing store and then brought back into main-memory for continued execution.

→Swapping makes it possible for the total physical address space of all processes to exceed the real physical memory of the system, thus increasing the degree of multiprogramming in a system.

Standard Swapping

→Standard swapping involves, moving processes between main memory and a backing store.

→The backing store is commonly a Hard Disk.

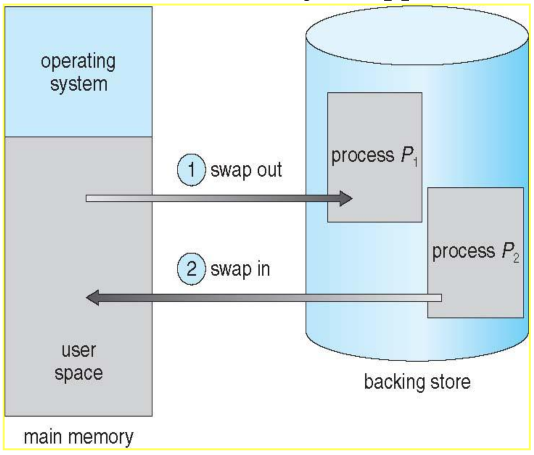
→The system maintains a Ready Queue consisting of all processes whose memory images are on the backing store or in memory and the processes are ready to run.

→Whenever the CPU scheduler decides to execute a process, it calls the dispatcher.

→The dispatcher checks to see whether the next process in the queue is in main memory.

→If it is not in main memory and if there is no free memory region, the dispatcher swaps out a process currently in main memory and swaps in the desired process. It then reloads registers and transfers control to the selected process.

→The context-switch time in such a swapping system is fairly high. The major part of the swap time is transfer time. The total transfer time is directly proportional to the amount of memory swapped.



Swapping on Mobile systems

Mobile systems such as iOS and Android do not support swapping.

· Mobile devices generally use flash memory rather than more spacious Hard disks as their persistent storage.

· Mobile operating-system designers avoid swapping because of the less space constraint.

· Flash memory can tolerate only the limited number of writes before it becomes unreliable and the poor throughput between main memory and flash memory in these devices.

Alternative methods used in Mobile systems instead of swapping:

· Apple's iOS asks applications to voluntarily relinquish allocated memory when free memory falls below a certain threshold.

· Read-only data (i.e. code) are removed from the system and later reloaded from flash memory if necessary.

· Data that have been modified such as the stack are never removed.

· Any applications that fail to free up sufficient memory may be terminated by the operating system.

· Android may terminate a process if insufficient free memory is available. Before terminating a process android writes its Application state to flash memory so that it can be quickly restarted.

CONTIGUOUS MEMORY ALLOCATION

Memory allocation can be done in two ways:

1. Fixed Partition Scheme (Multi-programming with Fixed Number of Tasks)
2. Variable partition scheme (Multi-programming with Variable Number of Tasks)

Fixed Partition Scheme (MFT)

The memory can be divided into several Fixed-Sized partitions.

→Each partition may contain exactly one process. The degree of multiprogramming is bound by the number of partitions.

→In this Multiple-Partition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition.

→When the process terminates, the partition becomes available for another process.

Variable partition scheme (MVT)

In the variable-partition scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied.

→Initially, all memory is available for user processes and it is considered one large block of available memory called as Hole.

→The memory contains a set of holes or fragments or parttitions of various sizes.

→As processes enter the system, they are put into an Input Queue.

→The operating system takes into account the memory requirements of each process and the amount of available memory space in determining which processes are allocated memory.

→When a process is allocated space, it is loaded into memory and it can then compete for CPU time.

→When a process terminates, it releases its memory. The operating system may use this free fill with another process from the input queue.

Memory is allocated to processes until the memory requirements of the next process cannot be satisfied (i.e.) there is no available block of memory is large enough to hold that process. Then operating system can wait until a large block is available for the process or it can skip the process and moves down to the input queue to see whether the smaller memory requirements of some other process can be met.

→When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process.

→If the hole is too large, it is split into two parts. One part is allocated to the arriving process and the other part is returned to the set of holes.

→When a process terminates, it releases its block of memory, which is then placed back in the set of holes.

Dynamic Storage Allocation Problem:

The above procedure leads to Dynamic storage allocation problem which concerns how to satisfy a request of size n from a list of free holes.

There are 3-solutions for this problem: First fit, Best fit, worst fit.

First fit: It allocates the first hole that is big enough. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.

Best fit. It allocates the smallest hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.

Worst fit. It allocates the largest hole.

Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

FRAGMENTATION

There are 2-problems with Memory allocation

1. Internal Fragmentation
2. External Fragmentation

Internal Fragmentation

→Consider a multiple-partition allocation scheme with a hole of 18,464 bytes.

→ Suppose that the next process requests 18,462 bytes.
If we allocate exactly the requested block, we are left with a hole of 2 bytes.

→The overhead to keep track of this hole will be substantially larger than the hole itself.

→The general approach to avoiding this problem is to break the physical memory into fixed-sized blocks and allocate memory in units based on block size. With this approach, the memory allocated to a process may be slightly larger than the requested memory.

→The difference between these two numbers(memory address) is Internal Fragmentation. It is unused memory that is internal to a partition.

External Fragmentation

→The first-fit and best-fit strategies for memory allocation suffer from External Fragmentation.

→As processes are loaded and removed from main memory, the free memory space is broken into small pieces.

→External fragmentation exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous, the storage is fragmented into a large number of small holes.

→External fragmentation problem can be severe. In the worst case, we could have a block of free memory between every two processes that is wasted.

→If all these small pieces of memory were in one big free block instead, we might be able to run several more processes.

Solution to External fragmentation

One solution to the problem of external fragmentation is Compaction.

· The goal is to shuffle the memory contents so as to place all free memory together in one large block.
· Compaction is possible only if relocation is dynamic and is done at execution time.
· If addresses are relocated dynamically, relocation requires only moving the program and data and then changing the base register to reflect the new base address.
· If relocation is static and is done at assembly or load time, compaction cannot be done.

Note: Compaction can be expensive, because it moves all processes toward one end of memory. All holes move in the other direction and produces one large hole of available memory.

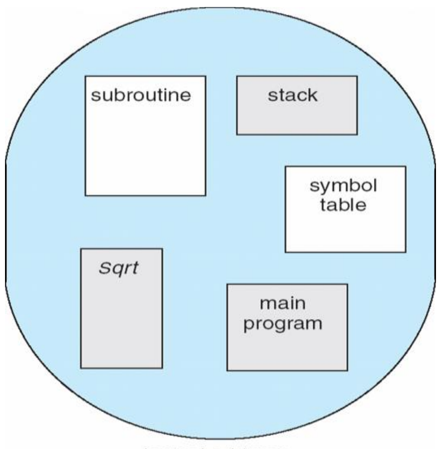
Other solutions to External fragmentation are Segmentation and Paging. They allow a process to be allocated physical memory wherever such memory is available. These are Non-contiguous memory allocation techniques.

SEGMENTATION

Segmentation is a memory-management scheme that permits the physical address space of a process to be non-contiguous.

A logical address space is a collection of segments. Each segment has a name and a length.

· Logical addresses specify both the segment name and the offset within the segment.
· The programmer specifies each address by two quantities: a segment name and an offset.
· The segments are referred to by Segment Number.
· A logical address consisting of two tuples: <Segment Number, offset>



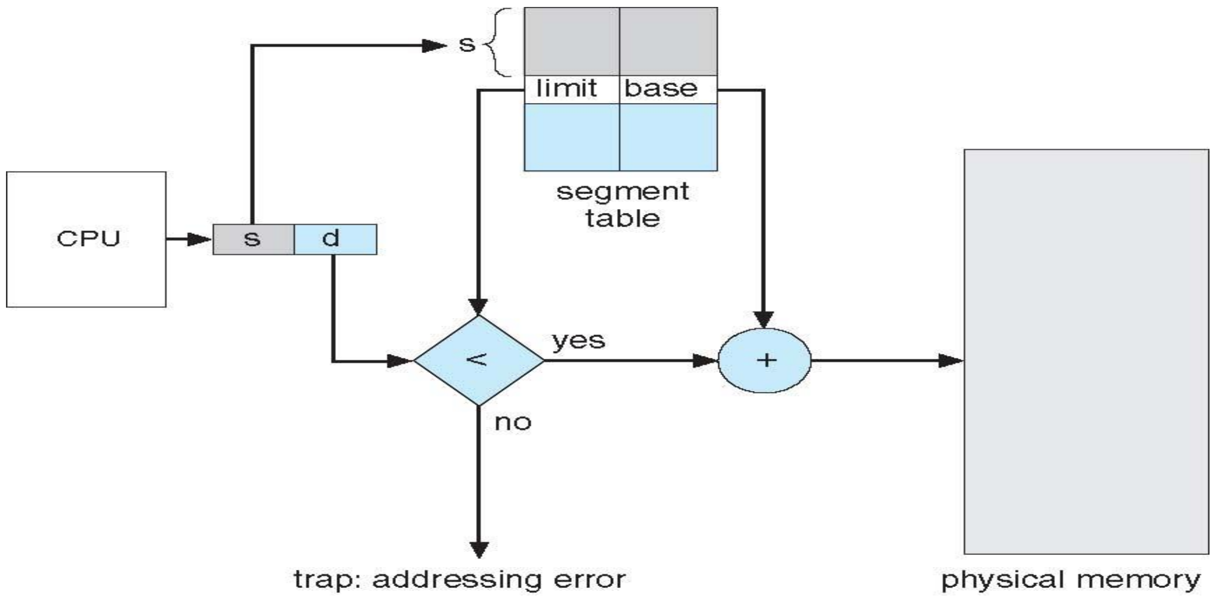
logical address

A C compiler might create separate segments for the following:
· The code
· Global variables
· The heap, from which memory is allocated
· The stacks used by each thread
· The standard C library

Note: Libraries that are linked in during compile time might be assigned separate segments. The loader would take all these segments and assign them segment numbers.

Segmentation Hardware

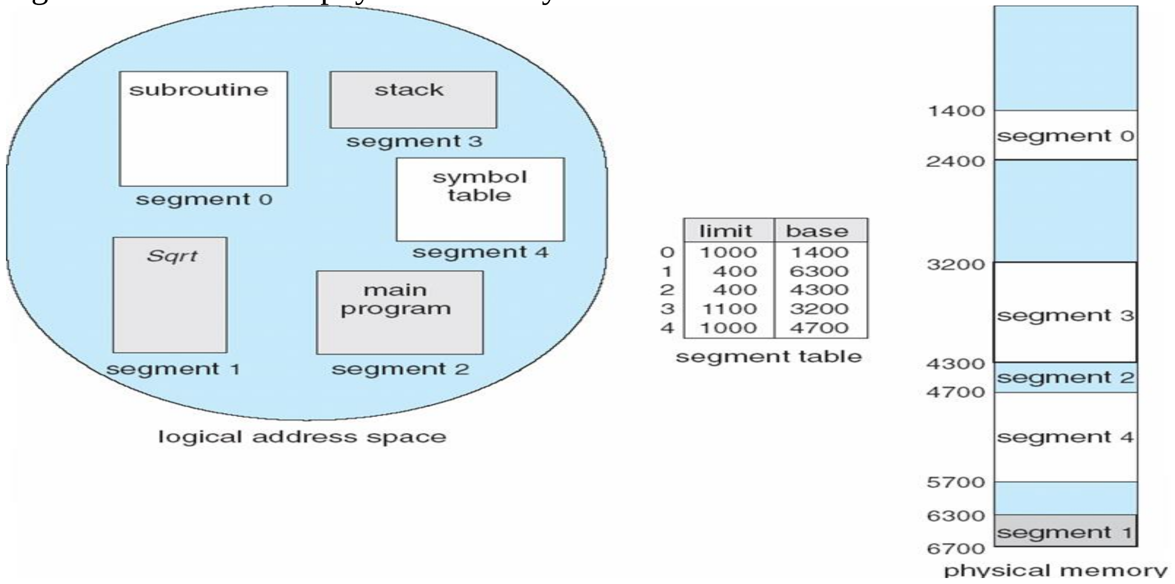
Logical address can be viewed by a programmer as a two dimensional address and where as actual Physical address is a one dimensional address.
· The Memory Management Unit (MMU) maps two-dimensional user-defined addresses into one-dimensional physical address.
· This mapping is effected by a Segment table.
· Each entry in the segment table has a segment base and a segment limit.
· The segment base contains the starting physical address where the segment resides in memory and the segment limit specifies the length of the segment.



A logical address consists of two parts: segment number s and an offset into that segment d.
· The segment number is used as an index to the segment table.
· The offset d of the logical address must be between 0 and the segment limit.

· When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte.
· If d>=segment limit, it is illegal then an addressing error trap will be generated that indicates logical addressing attempt beyond end of segment.
· The segment table is essentially an array of base–limit register pairs.

Example: Consider the below diagram that have five segments numbered from 0 to 4. The segments are stored in physical memory.

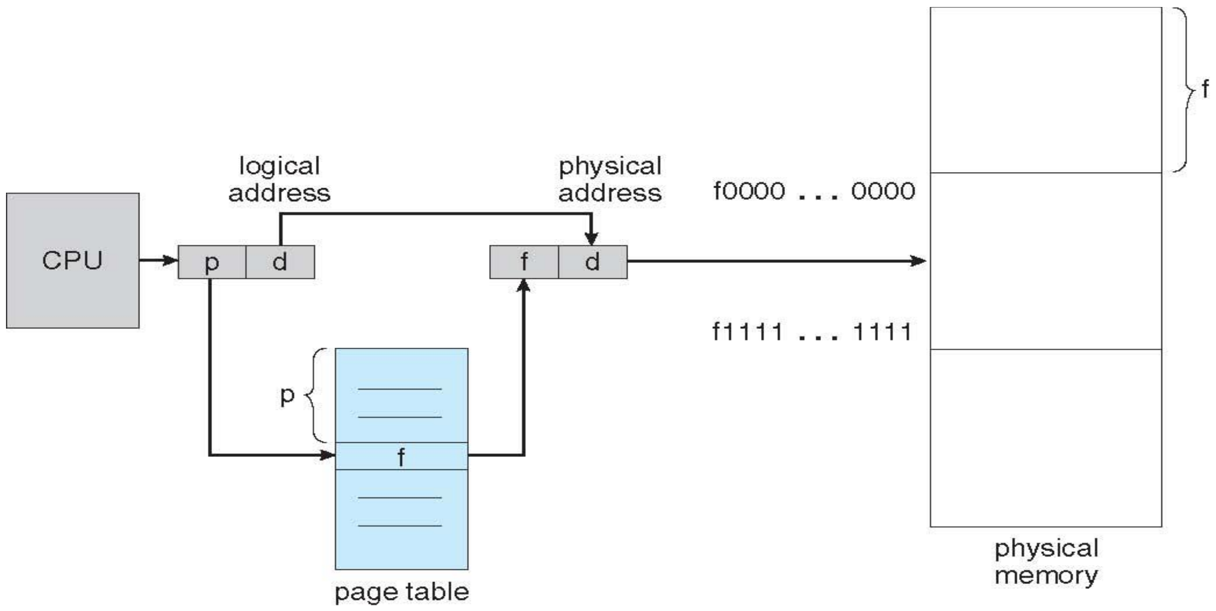


	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

The segment table has a separate entry for each segment, giving the beginning address of the segment in physical memory (i.e. base) and the length of that segment (i.e. limit).
1. 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.
2. A reference to segment 3, byte 852 is mapped to 3200 (base of segment 3) + 852 = 4052.
3. A reference to byte 1222 of segment 0 would result in a trap to the operating system, as this segment is only 1,000 bytes long.

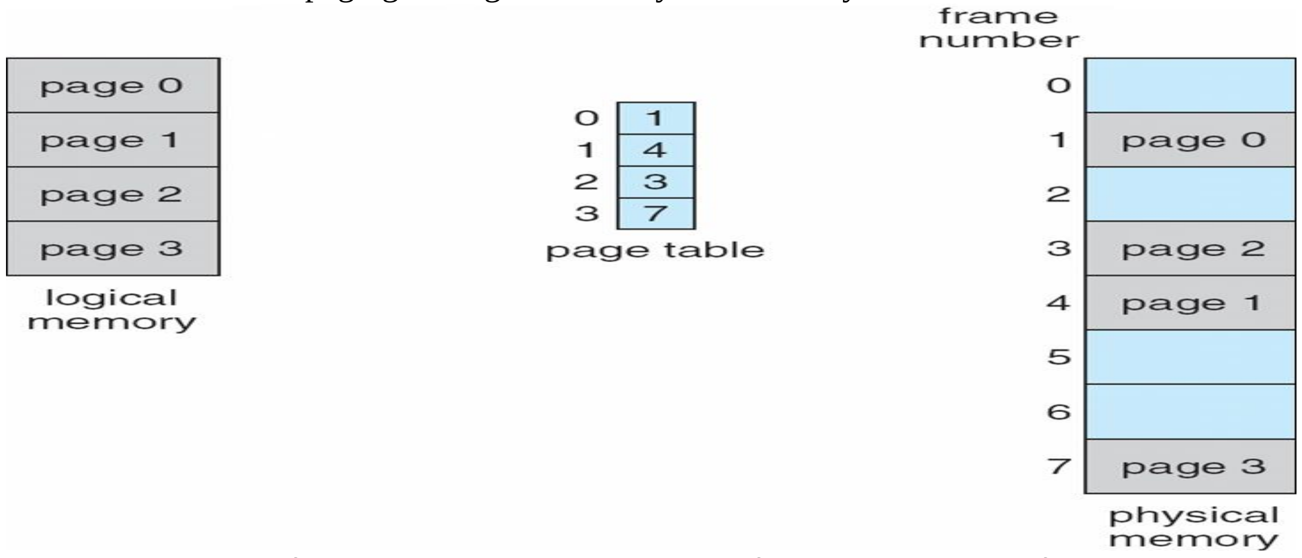
PAGING

Paging also permits the physical address space of a process to be non-contiguous.
Paging avoids External fragmentation and need for compaction.
Paging is implemented through cooperation between the operating system and the computer hardware.
· Physical memory is divided into fixed-sized blocks called Frames.
· Logical memory is divided into blocks of the same size called Pages.
· Frame size is equal to the Page size.
· When a process is to be executed, its pages are loaded into any available memory frames from their source such as a file system or backing store.
· The backing store is divided into fixed-sized blocks that are the same size as the memory frames or clusters of multiple frames.
· Frame table maintains list of frame and the allocation details of the frames (i.e.) A frame is free or allocated to some page.
· Each process has its own page table. When a page is loaded into main memory the corresponding page table is active in the system and all other page tables are inactive.
· Page tables and Frame tables are kept in main memory. A Page-Table Base Register (PTBR) points to the page table

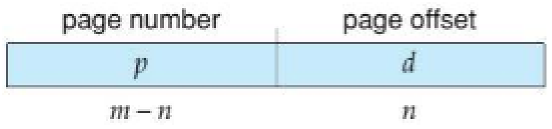


· Every address generated by the CPU is divided into two parts: a page number (p) and a page-offset (d).
· The page number is used as an index into a Page table. The page table contains the base address of each page in physical memory.
· This base address is combined with the page offset to define the physical memory address that is sent to the memory unit.

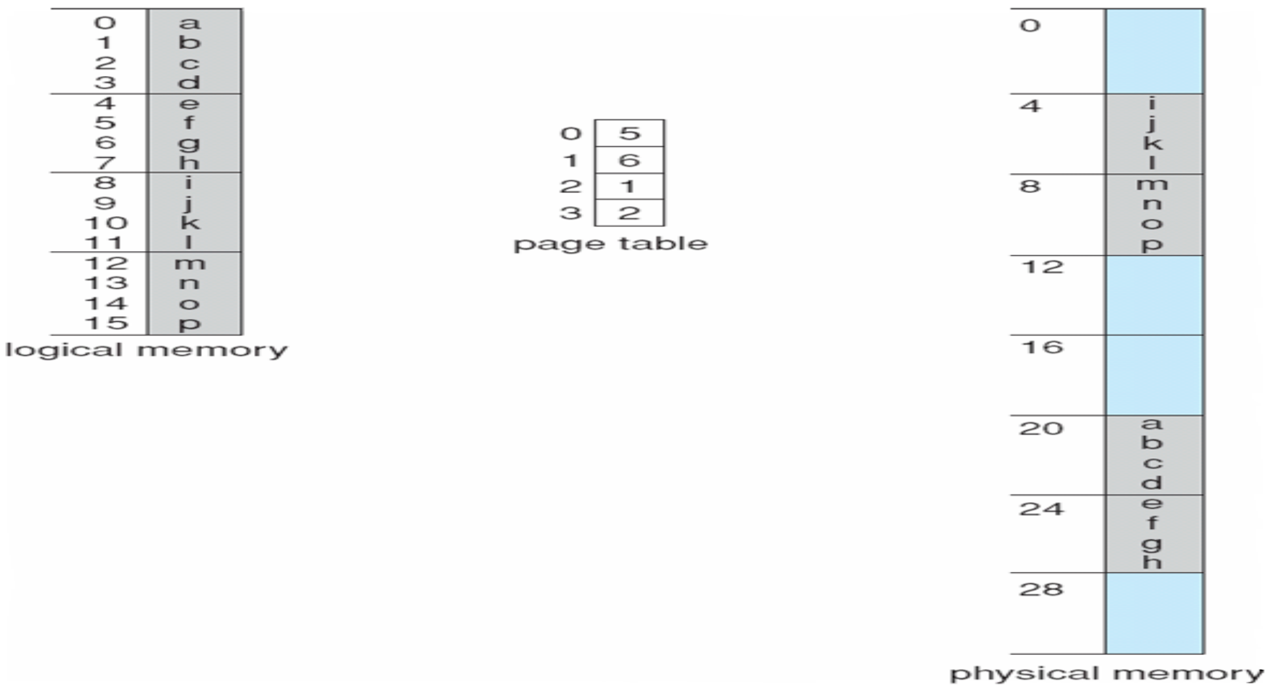
The below shows the paging of Logical and Physical memory:



· The page size is defined by the hardware. The size of a page is a power of 2.
· Depending on the computer architecture the page size varies between 512 bytes and 1 GB per page.
· The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy.
· If the size of the logical address space is 2m and a page size is 2n bytes, 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.
The logical address contains: p is an index into the page table and d is the isplacement within the page.



Example: consider the memory in the below figure where n= 2 and m = 4.
Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages), we show how the programmer’s view of memory can be mapped into physical memory



n=2 and *m*=4 32-byte memory and 4-byte pages

· 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 × 4) + 0].
· Logical address 3 (page 0, offset 3) maps to physical address 23 [= (5 × 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 × 4) + 0].
· Logical address 13 maps to physical address 9.

Paging scheme avoids external fragmentation but it creates internal fragmentation because of fixed size pages.
· If page size is 2048 bytes, a process of 20489 bytes will need 10 pages plus 9 bytes.
· It will be allocated 11 frames, resulting in internal fragmentation of 2048−9 = 2037 bytes.
· In the worst case, a process would need n pages plus 1 byte. It would be allocated n + 1 frames resulting in internal fragmentation of almost an entire frame.
· If the page size is small then the number of entries in page table is more this will leads to huge number of context switches.
· If the page size is large then the number of entries in page table is less and the number of context switches is less

VIRTUAL MEMORY

INTRODUCTION

· Virtual Memory is a technique that allows the execution of processes that are not completely in Main-memory.
· Virtual memory involves the separation of Logical Memory as perceived by users from Physical Memory.
· This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available.
· A program would no longer be constrained by the amount of physical memory that is available. Users would be able to write programs for a large virtual address space.
· Because each user program could take less physical memory, more programs could be run at the same time with a corresponding increase in CPU utilization and throughput but it will not increase the Response time or Turnaround time.
· Less I/O would be needed to load or swap user programs into memory, so each user program would run faster. This process will benefit both system and user.

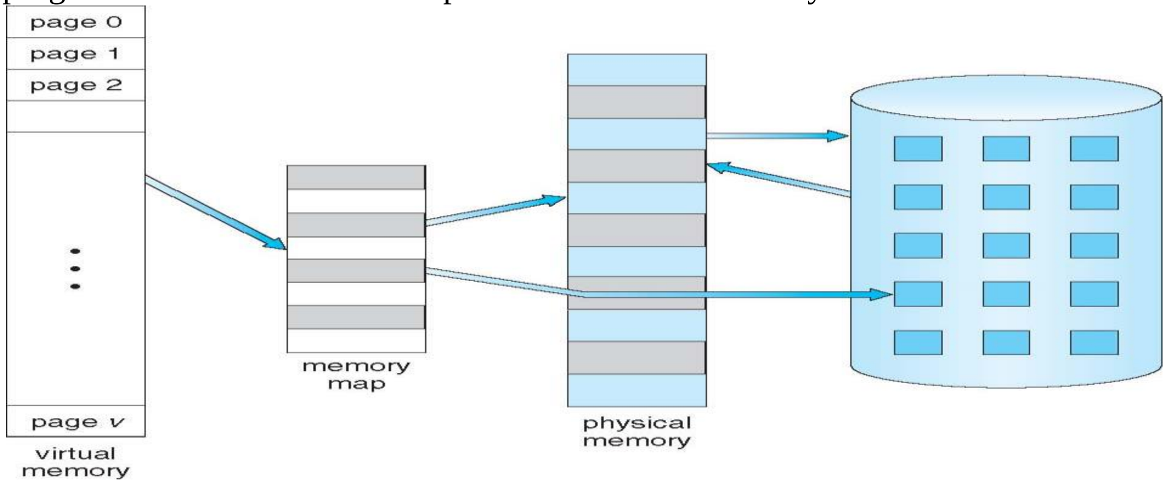


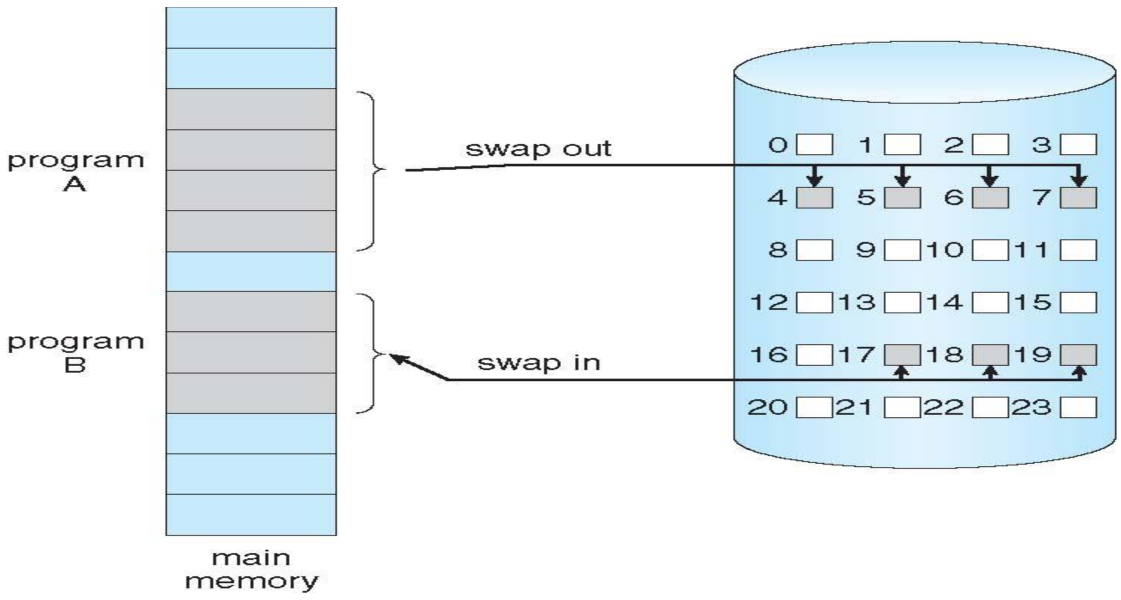
Fig. Diagram showing virtual memory that is larger than physical memory.

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

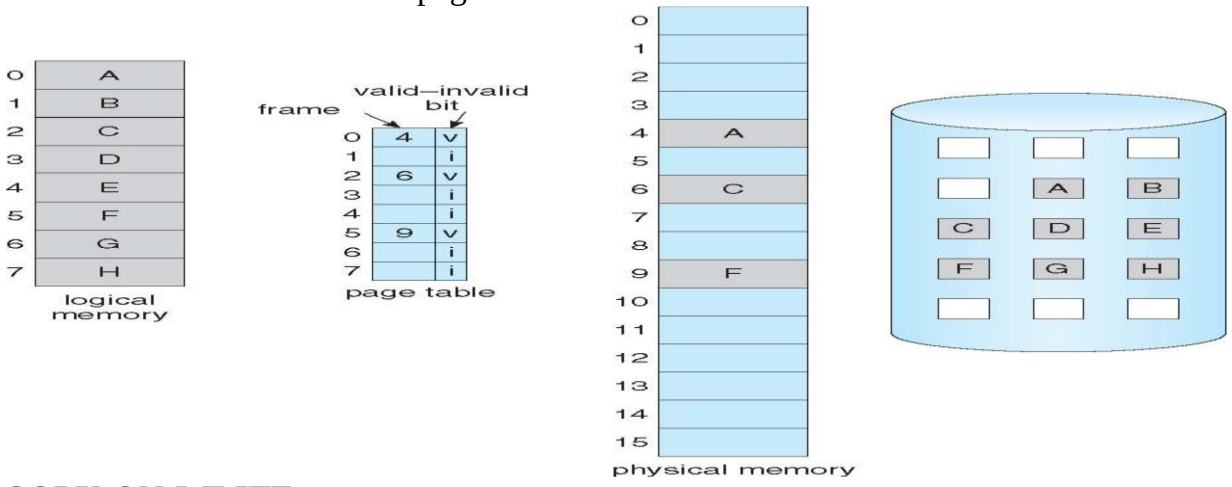
Benefits of having Virtual Memory :
1. Large programs can be written, as virtual space available is huge compared to physical memory
2. Less I/O required, leads to faster and easy swapping of processes.
3. More physical memory available, as programs are stored on virtual memory, so theyoccupy very less space on actual physical memory.

Demand Paging

With Demand-paged virtual memory, pages are loaded only when they are demanded during program execution. Pages that are never accessed are never loaded into physical memory.

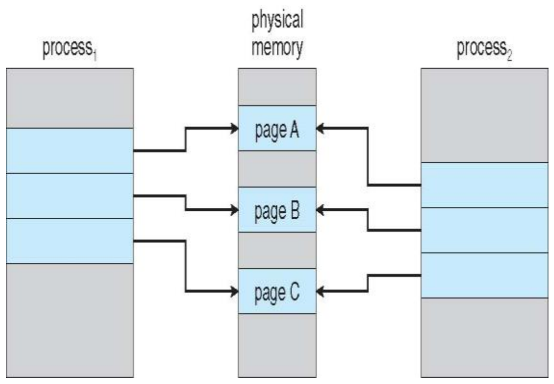


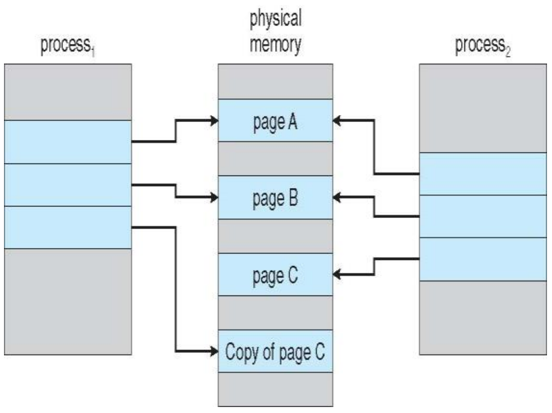
· A demand-paging system is similar to a paging system with swapping, where processes reside in secondary memory (i.e. disk).
· Demand paging uses the concept of Lazy Swapper or Lazy Pager. A lazy swapper or pager never swaps a page into memory unless that page will be needed.
· Valid–Invalid bit is used to distinguish between the pages that are in memory and the pages that are on the disk.
· When this bit is set to "valid," the associated page is both legal and is in main memory.
· If the bit is set to "invalid," the page either is not valid (i.e. not in the logical address space of the process) or is valid but is currently on the disk.
· The page-table entry for a page that is brought into main memory is set to valid, but the page-table entry for a page that is not currently in main memory is either marked as invalid or contains the address of the page on disk



COPY-ON-WRITE

· The fork( ) system call creates a child process that is a duplicate of its parent.
· fork( ) worked by creating a copy of the parent's address space for the child, duplicating the pages belonging to the parent.
· Many child processes invoke the exec( ) system call immediately after creation and the copying of the parent's address space may be unnecessary.
· Copy-on-Write is a technique which allows the parent and child processes initially to share the same pages.
· These shared pages are marked as Copy-on-Write pages, meaning that if either process writes to a shared page, a copy of the shared page is created.



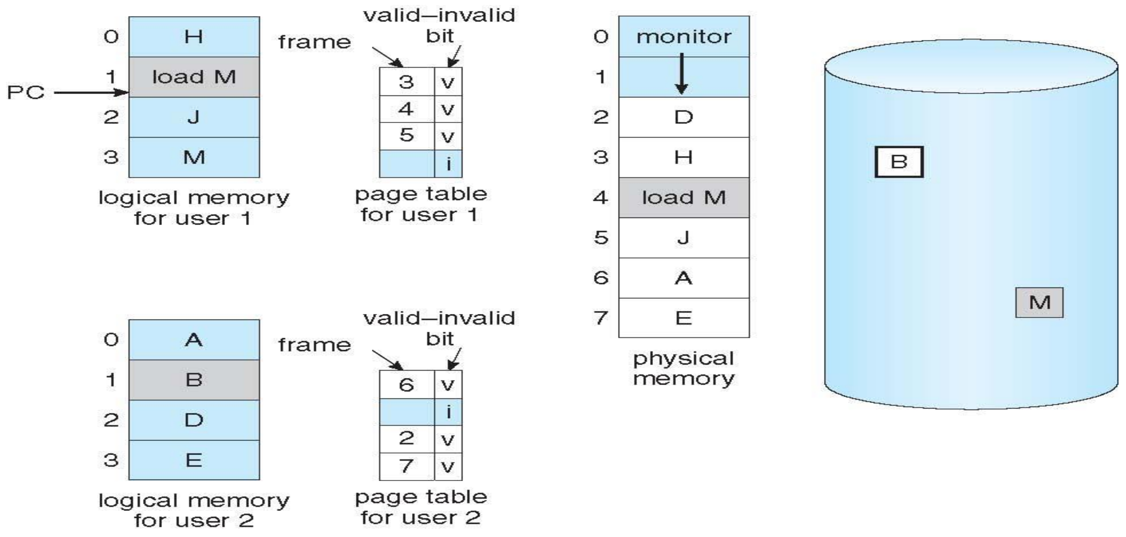


Before Process 1 Modifies Page C **After Process 1 Modifies Page C**

Example: Assume that the child process attempts to modify a page containing portions of the stack, with the pages set to be Copy-on-Write.
· Operating system will create a copy of this page, mapping it to the address space of the child process.
· The child process will then modify its own copied page but not the page belonging to the parent process.
· When the Copy-on-Write technique is used, only the pages that are modified by either process are copied.
· Only pages that can be modified need be marked as Copy-on-Write. Pages that cannot be modified can be shared by the parent and child processes.
· Windows XP, Linux and Solaris operating systems uses Copy-on-Write technique.

<u>PAGE REPLACEMENT ALGORITHMS</u>

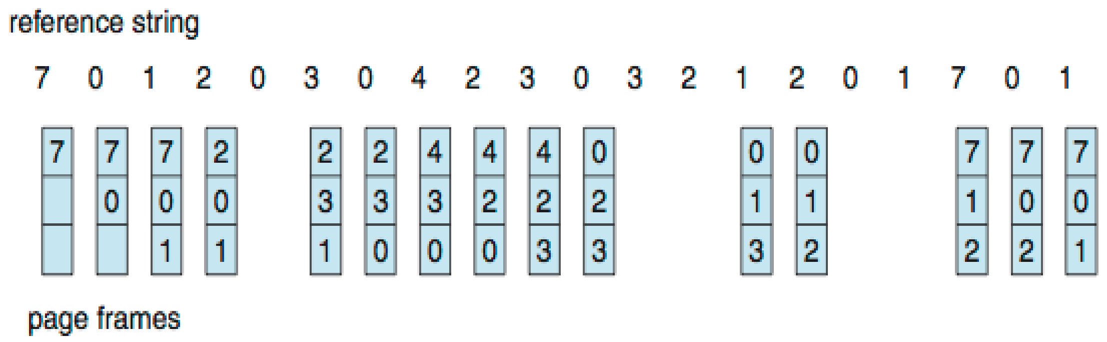
· If no frame is free, we find one that is not currently being used and free it.
· We can free a frame by writing its contents to swap space and changing the page table to indicate that the page is no longer in main memory.
· We can now use the freed frame to hold the page for which the process faulted.
We modify the page-fault service routine to include Page Replacement:
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 frame to the disk and change the page table and frame table.
3. Read the desired page into newly freed frame and change the page table and frame table.
4. Continue the user process from where the page fault occurred.



Modify bit or Dirty bit
Each page or frame has a modify bit associated with it in the hardware.
· The modify bit for a page is set by the hardware whenever any byte in the page has been modified.
· When we select a page for replacement, we examine its modify bit.
· If the bit is set, the page has been modified since it was read in from the disk. Hence we must write the page to the disk.
· If the modify bit is not set, the page has not been modified since it was read into memory. Hence there is no need for write the memory page to the disk, because it is already there.
There is several Page Replacement Algorithms are in use:
1. FIFO Page Replacement Algorithm
2. Optimal Page Replacement Algorithm
3. LRU Page Replacement Algorithm
4. Counting Based Page Replacement Algorithm

First-In-First-Out Page Replacement Algorithm

FIFO algorithm associates with time of each page when it was brought into main memory.
· When a page must be replaced, the oldest page is chosen.
· 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: Consider the below reference string and memory with three frames
7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1



· First 3-references (7, 0, 1) cause 3-Page faults and are brought into these empty frames.
· The next reference (2) replaces page 7, because page 7 was brought in first.
· Since 0 is the next reference and 0 is already in memory, we have no fault for this reference.
· The first reference to 3 results in replacement of page 0, since it is now first in line.

Because of this replacement, the next reference to 0, will fault. Page 1 is then replaced by page 0.

· By the end, there are Fifteen page faults altogether.

Problem: Belady's Anomaly

Belady's Anomaly states that: the page-fault rate may increase as the number of allocated frames increases. Researchers identifies that Belady's anomaly is solved by using Optimal Replacement algorithm.

Optimal Page Replacement Algorithm (OPT Algorithm)

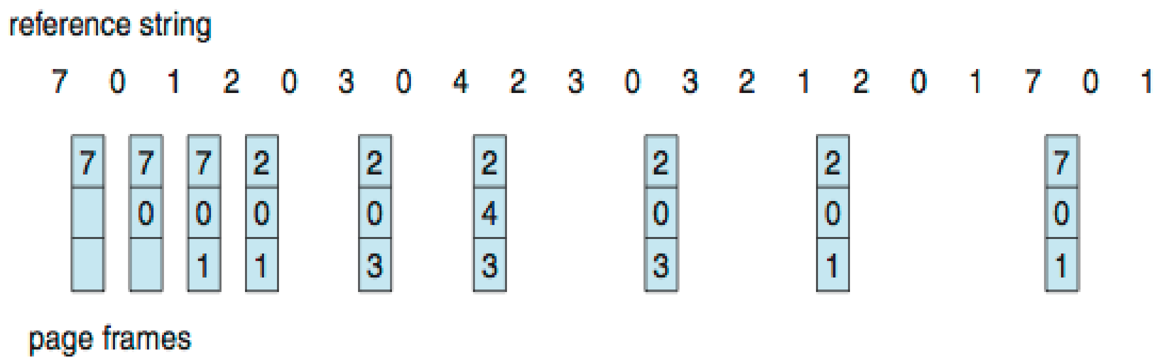
· It will never suffer from Belady's anomaly.

· OPT states that: Replace the page that will not be used for the longest period of time.

· OPT guarantees the lowest possible page fault rate for a fixed number of frames.

Example: Consider the below reference string and memory with three frames

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



· The first 3-references cause faults that fill the 3-empty frames.

· The reference to page 2 replaces page 7, because page 7 will not be used until reference number 18, whereas page 0 will be used at 5 and page 1 at 14.

· The reference to page 3 replaces page 1 because page 1 will be the last of the three pages in memory to be referenced again.

· At the end there are only 9-page faults by using optimal replacement algorithm which is much better than a FIFO algorithm with 15-page faults.

Note: No replacement algorithm can process this reference string in 3-frames with fewer than 9-faults.

Problem with Optimal Page Replacement algorithm

The optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string. The optimal algorithm is used mainly for comparison studies (i.e. performance studies)

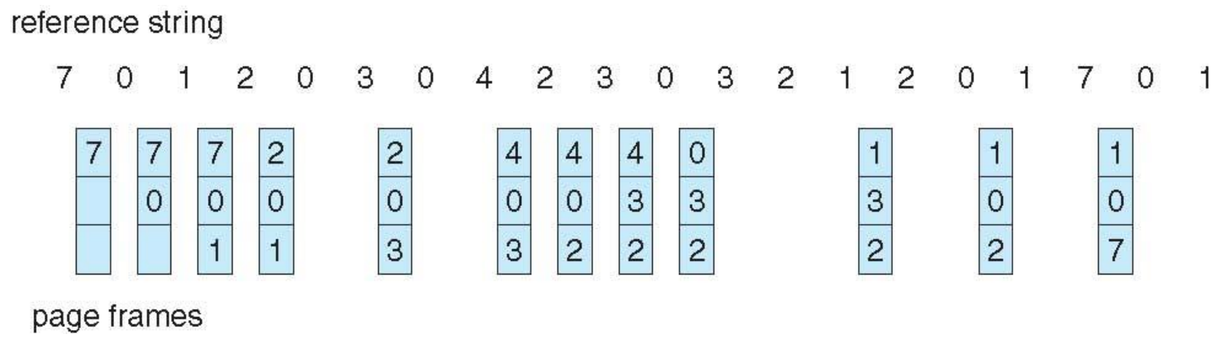
LRU Page Replacement Algorithm

In LRU algorithm, the page that has not been used for the longest period of time will be replaced (i.e.) we are using the recent past as an approximation of the near future.

LRU replacement associates with each page the time of that page's last use.

Example: Consider the below reference string and memory with three frames

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



· The first five faults are the same as those for optimal replacement.
· When the reference to page 4 occurs LRU replacement sees that out of the three frames in memory, page 2 was used least recently.
· Thus, the LRU algorithm replaces page 2, not knowing that page 2 is about to be used.
· When it then faults for page 2, the LRU algorithm replaces page 3, since it is now the least recently used of the three pages in memory.
· The total number of page faults with LRU is 12 which is less as compared to FIFO.
LRU can be implemented in two ways: Counters and Stack
Counters
· Each page-table entry is associated with 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
LRU replacement is implemented by keeping 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 most recently used page is always at the top of the stack and the least recently used page is always at the bottom.
· Because entries must be removed from the middle of the stack, it is best to implement this approach by using a doubly linked list with a Head pointer and a Tail pointer.
· Removing a page and putting it on the top of the stack then requires changing six pointers at worst.
· 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 implementations of LRU replacement.
.
ALLOCATION OF FRAMES
There are different approaches used for allocation of frames:
1. Minimum Number of Frames
2. Allocation Algorithms
3. Global versus Local Allocation
4. Non-Uniform Memory Access
Minimum Number of Frames
Allocation is based on minimum number of frames per process is defined by the computer architecture. The maximum number is defined by the amount of available physical memory.

We must also allocate at least a minimum number of frames.
· One reason for allocating at least a minimum number of frames involves performance.
· When a page fault occurs before an executing instruction is complete, the instruction must be restarted.
· As the number of frames allocated to each process decreases, the page-fault rate increases and slows the process execution.
· Hence the process must have enough frames to hold all the different pages that any single instruction can reference.

Allocation Algorithms

There are two algorithm are used: Equal allocation and Proportional allocation.

Equal Allocation

· This algorithm splits m frames among n processes is to give everyone an equal share, m/n frames.
· Example: If there are 93 frames and five processes, each process will get 18 frames (93/5=18). The 3-leftover frames can be used as a free-frame buffer pool.

Problem with Equal Allocation

Consider a system with a 1-KB frame size and two processes P1 and P2.
· Process P1 is of 10 KB and process P2 is of 127 KB are the only two processes running in a system with 62 free frames.
· Now if we apply Equal allocation then both P1 and P2 will get 31 frames.
· It does not make sense to give P1 process to 31 frames where its maximum use is 10 frames and other 21 frames are wasted.

Proportional Allocation

· In this algorithm we allocate available memory to each process according to its size.
· Let the si be the size of the virtual memory for process Pi then S = si
· If the total number of available frames is m, we allocate ai frames to process Pi, where ai is approximately: ai = si/S × m.
· We must adjust each ai to be an integer that is greater than the minimum number of frames required by the instruction set, with a sum not exceeding m.

Example: With proportional allocation, we would split 62 frames between 2-processes, one of 10 pages and one of 127 pages, by allocating 4 frames for P1 and 57 frames for P2.
P1-> 10/137 × 62 ≈ 4
P2-> 127/137 × 62 ≈ 57

In this way both processes share the available frames according to their “needs,” rather than equally.

Global versus Local Allocation

With multiple processes competing for frames, 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 (i.e.) one process can take a frame from another process.
· Local replacement requires that each process select from only its own set of allocated frames.

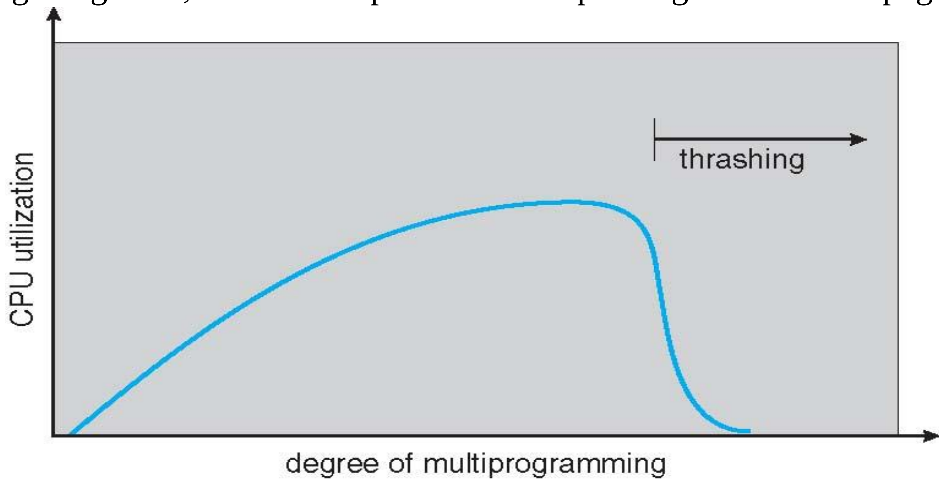
Example: Consider an allocation scheme wherein we allow High-priority processes to select frames from low-priority processes for replacement.
· A process can select a replacement from its own frames or the frames of any lower-priority process.
· This approach allows a High-priority process to increase its frame allocation at the

expense of a low-priority process.
· With a local replacement strategy, the number of frames allocated to a process does not change.
· With global replacement, a process may happen to select only frames allocated to other processes, thus increasing the number of frames allocated to it.
Non-Uniform Memory Access (NUMA)
· Consider a system is made up of several system boards, each containing multiple CPUs and some memory.
· In systems with multiple CPUs, a one CPU can access some sections of main memory faster than it can access others.
· The system boards are interconnected in various ways, ranging from system buses to High-speed network connections.
· The CPUs on a particular board can access the memory on that board with less delay than they can access memory on other boards in the system.
· Systems in which memory access times vary significantly are known collectively as Non-Uniform Memory Access (NUMA) systems.
· NUMA systems are slower than systems in which memory and CPUs are located on the same motherboard.

<u>THRASHING</u>
A process is thrashing if it is spending more time for paging than executing.
· If the number of frames allocated to a low-priority process falls below the minimum number required by the computer architecture, we must suspend that process's execution.
· We should then page out its remaining pages, freeing all its allocated frames.
· This provision introduces a swap-in, swap-out level of intermediate CPU scheduling.
· If the process does not have the number of frames it needs to support pages in active use, it will quickly page-fault and the process must replace some page

· Since all of its pages are in active use, it must replace a page that will be needed again right away. Consequently, it quickly faults again and again by replacing pages that it must bring back in immediately.
· This high paging activity is called Thrashing.
<u>Cause of Thrashing</u>
The operating system monitors CPU utilization. If CPU utilization is too low, we increase the degree of multiprogramming by introducing a new process to the system.
· If a global page-replacement algorithm is used then it replaces pages without regard to the process to which the pages are belongs to.
· Now suppose that a process enters a new phase in its execution and needs more frames. It starts faulting and taking frames away from other processes.
· These processes need those pages which have been faulted earlier so they also fault taking frames from other processes.
· These faulting processes must use the paging device to swap pages in and out.
· As they queue up for the paging device, the ready queue empties. As processes wait for the paging device, CPU utilization decreases.
· The CPU scheduler sees the decreasing CPU utilization and increases the degree of multiprogramming by introducing new process in to the system again.
· The new process tries to get started by taking frames from running processes, causing more page faults and a longer queue for the paging device.
· As a result, CPU utilization drops even further and the CPU scheduler tries to increase the degree of multiprogramming even more.

· Thrashing has occurred and system throughput decreases. The page fault rate increases tremendously. As a result, the effective memory-access time increases.
· No work is getting done, because the processes are spending all their time paging



Consider the above figure that show how thrashing will occur:
· As the degree of multiprogramming increases, CPU utilization also increases until a maximum is reached.
· If the degree of multiprogramming is increased even further then thrashing occurs and CPU utilization drops sharply.
· At this point, we must stop thrashing and increase the CPU utilization by decreasing the the degree of multiprogramming.

<u>Solutions to Thrashing</u>

1. Local Replacement Algorithm (or) Priority Replacement Algorithm
2. Locality Model

<u>Local Replacement Algorithm</u>

· With local replacement, if one process starts thrashing, it cannot steal frames from another process.
· Local replacement Algorithm limits thrashing but it cannot avoid thrashing entirely.
· If processes are thrashing, they will be paging device queue most of the time.
· The average service time for a page fault will increase because of the longer average queue for the paging device.
· Thus, the effective access time will increase even for a process that is not thrashing.

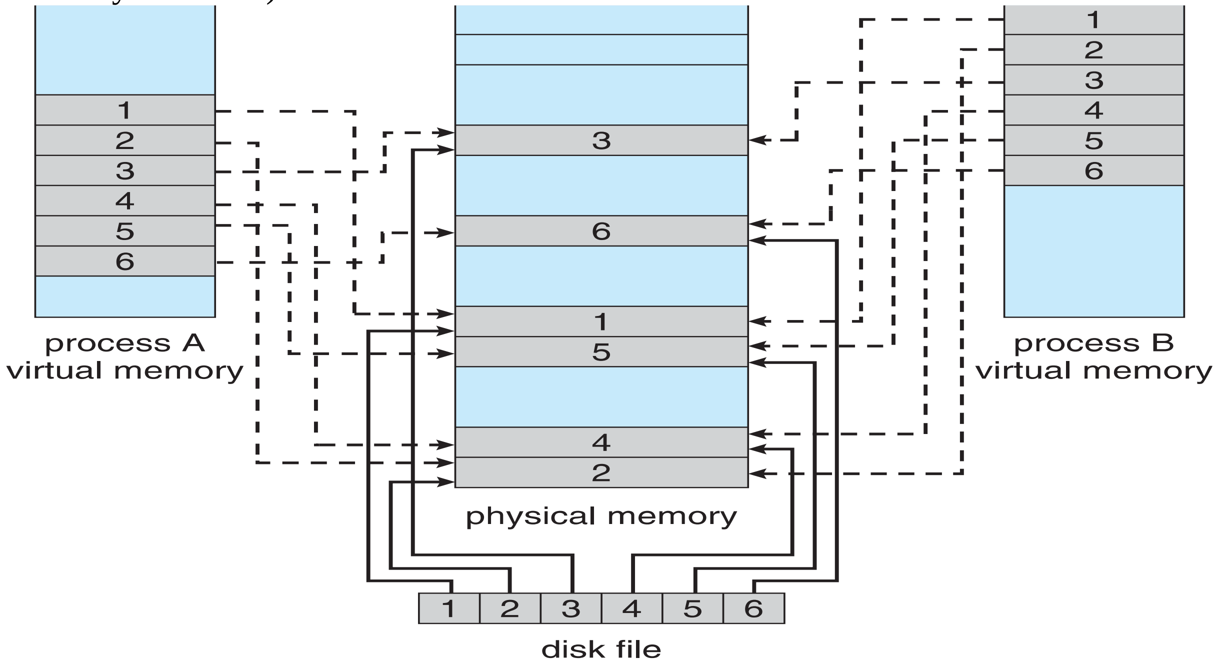
Locality Model

· The locality model states that, as a process executes, it moves from locality to locality.
· A locality is a set of pages that are actively used together. A program is generally composed of several different localities, which may overlap.

Example: When a function is called, it defines a new locality. In this locality, memory references are made to the instructions of the function call, its local variables and a subset of the global variables. When we exit the function, the process leaves this locality, since the local variables and instructions of the function are no longer in active use. We may return to this locality later.
Note: Localities are defined by the program structure and its data structures.
Suppose we allocate enough frames to a process to accommodate its current locality.
· It will fault for the pages in its locality until all these pages are in memory; then, it will not fault again until it changes localities.
· If we do not allocate enough frames to accommodate the size of the current locality, the process will thrash, since it cannot keep in memory all the pages that it is actively using.

Memory-Mapped Files

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by **mapping** a disk block to a page in memory
- A file is initially read using demand paging
  - A page-sized portion of the file is read from the file system into a physical page
  - Subsequent reads/writes to/from the file are treated as ordinary memory accesses
- Simplifies and speeds file access by driving file I/O through memory rather than read() and write() system calls
- Also allows several processes to map the same file allowing the pages in memory to be shared
- But when does written data make it to disk?
  - Periodically and / or at file close() time
  - For example, when the pager scans for dirty pages
- Some OSes uses memory mapped files for standard I/O
- Process can explicitly request memory mapping a file via mmap() system call
  - Now file mapped into process address space
- For standard I/O (open(), read(), write(), close()), mmap anyway
  - But map file into kernel address space
  - Process still does read() and write()
    - Copies data to and from kernel space and user space
  - Uses efficient memory management subsystem
    - Avoids needing separate subsystem
- COW can be used for read/write non-shared pages
- Memory mapped files can be used for shared memory (although again via separate system calls)



Shared Memory in Windows API

- First create a **file mapping** for file to be mapped
  - Then establish a view of the mapped file in process’s virtual address space
- Consider producer / consumer
  - Producer create shared-memory object using memory mapping features

- Open file via CreateFile(), returning a HANDLE
- Create mapping via CreateFileMapping() creating a **named shared-memory object**
- Create view via MapViewOfFile()