UNIT -3

Memory Management

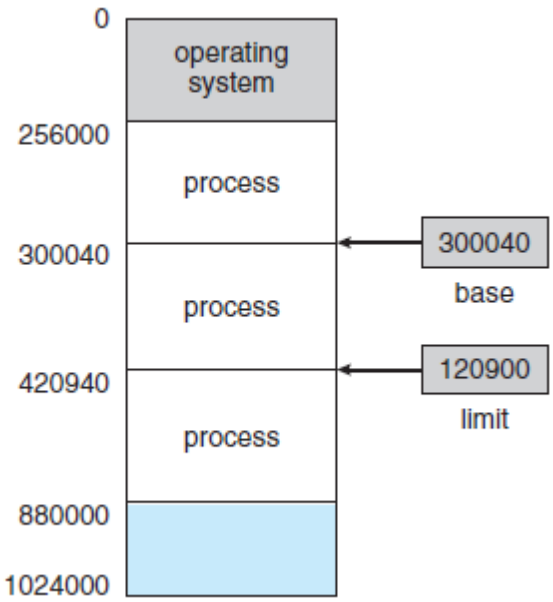
The task of subdividing the memory among different processes is called Memory Management. Memory management is a method in the operating system to manage operations between main memory and disk during process execution. The main aim of memory management is to achieve efficient utilization of memory.

Base register

A base register is a special-purpose register used in memory management to hold the starting address of a memory segment. This concept is crucial for systems that implement segmentation or paging to manage memory efficiently.

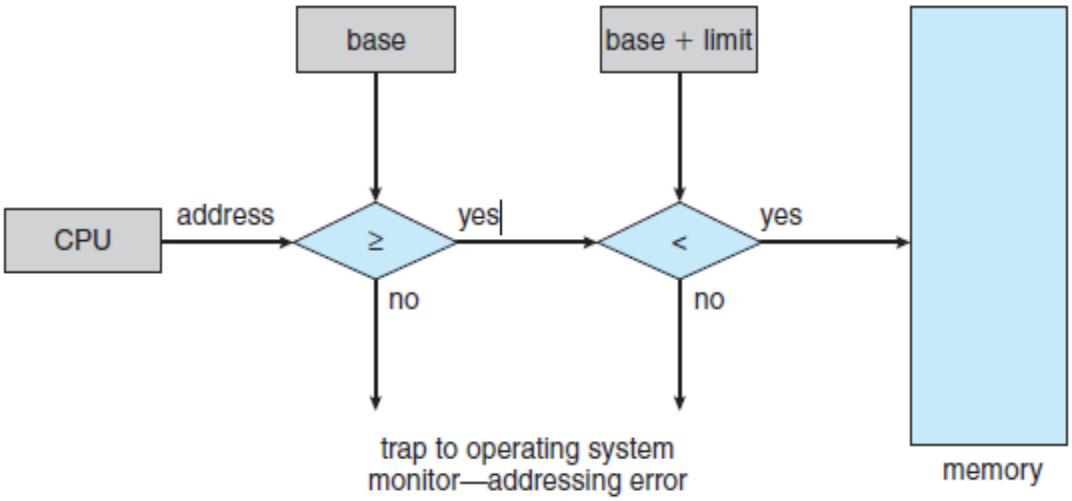
Limit register

The limit register specifies the size of the range. It works in conjunction with the base register to manage and protect memory segments effectively.



Hardware address protection with base and limit registers

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 . This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users.



The base and limit registers can be loaded only by the operating system, which uses a special privileged instruction. Since privileged instructions can be executed only in kernel mode, and since only the operating system executes in kernel mode, only the operating system can load the base and limit registers.

Logical Address

The logical address is a virtual address created by the CPU of the computer system. The logical address of a program is generated when the program is running. A group of several logical address is referred to a logical address space. The logical address is basically used as a reference to access the physical memory locations.

In computer systems, a hardware device named memory management unit (MMU) is used to map the logical address to its corresponding physical address. However, the logical address of a program is visible to the computer user.

Physical Address

The physical address of a computer program is one that represents a location in the memory unit of the computer. The physical address is not visible to the computer user. The MMU of the system generates the physical address for the corresponding logical address.

The physical address is accessed through the corresponding logical address because a user cannot directly access the physical address. For running a computer program, it requires a physical memory space. Therefore, the logical address has to be mapped with the physical address before the execution of the program.

Difference between Logical and Physical Address

S. No.	Logical Address	Physical Address
1.	This address is generated by the CPU.	This address is a location in the memory unit.
2.	The address space consists of the set of all logical addresses.	This address is a set of all physical addresses that are mapped to the corresponding logical addresses.
3.	These addresses are generated by CPU with reference to a specific program.	It is computed using Memory Management Unit (MMU).
4.	The user has the ability to view the logical address of a program.	The user can't view the physical address of program directly.
5.	The user can use the logical address in order to access the physical address.	The user can indirectly access the physical address.

SWAPPING

Swapping is a memory management scheme in which any process can be temporarily swapped from main memory to secondary memory so that the main memory can be made available for other processes. It is used to improve main memory utilization. In secondary memory, the place where the swapped-out process is stored is called swap space.

The purpose of the swapping in operating system is to access the data present in the hard disk and bring it to RAM so that the application programs can use it. The thing to remember is that swapping is used only when data is not present in RAM.

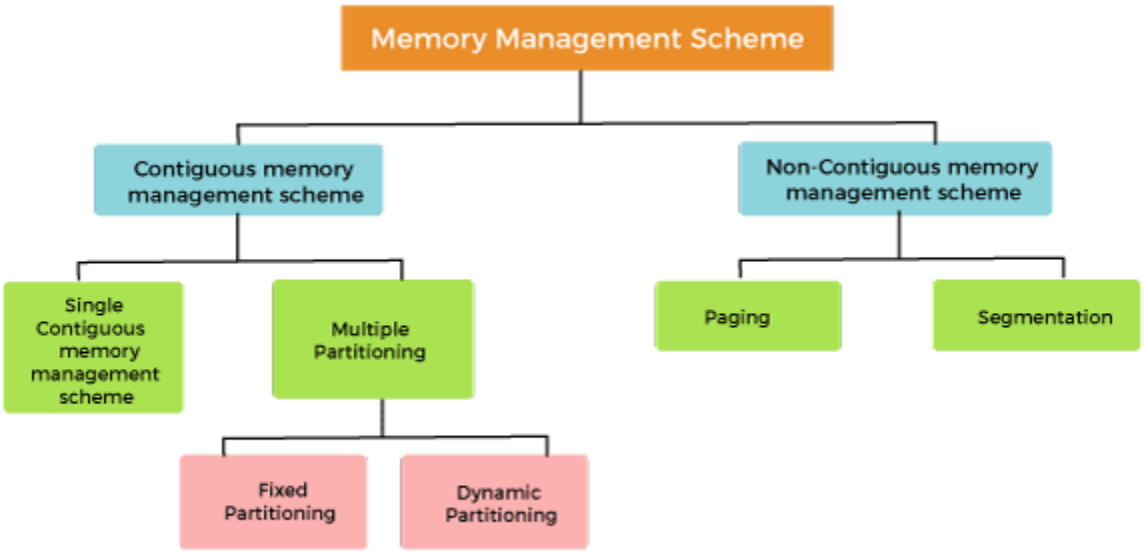
The concept of swapping has divided into two more concepts: Swap-in and Swap-out.

- **Swap-out** is a method of removing a process from RAM and adding it to the hard disk.
- **Swap-in** is a method of removing a program from a hard disk and putting it back into the main memory or RAM.

MEMORY MANAGEMENT TECHNIQUES:

The memory management techniques can be classified into following main categories:

- Contiguous memory management schemes
- Non-Contiguous memory management schemes



Classification of memory management schemes

Contiguous memory management schemes:

In a Contiguous memory management scheme, each program occupies a single contiguous block of storage locations, i.e., a set of memory locations with consecutive addresses.

Single contiguous memory management schemes:

The Single contiguous memory management scheme is the simplest memory management scheme used in the earliest generation of computer systems. In this scheme, the main memory is divided into two contiguous areas or partitions. The operating systems reside permanently in one partition, generally at the lower memory, and the user process is loaded into the other partition.

Multiple Partitioning:

The single Contiguous memory management scheme is inefficient as it limits computers to execute only one program at a time resulting in wastage in memory space and CPU time. The problem of inefficient CPU use can be overcome using multiprogramming that allows more than one program to run concurrently. To switch between two processes, the operating systems need

to load both processes into the main memory. The operating system needs to divide the available main memory into multiple parts to load multiple processes into the main memory. Thus multiple processes can reside in the main memory simultaneously.

The multiple partitioning schemes can be of two types:

- Fixed Partitioning
- Dynamic Partitioning

Fixed Partitioning

The main memory is divided into several fixed-sized partitions in a fixed partition memory management scheme or static partitioning. These partitions can be of the same size or different sizes. Each partition can hold a single process. The number of partitions determines the degree of multiprogramming, i.e., the maximum number of processes in memory. These partitions are made at the time of system generation and remain fixed after that.

Dynamic Partitioning

The dynamic partitioning was designed to overcome the problems of a fixed partitioning scheme. In a dynamic partitioning scheme, each process occupies only as much memory as they require when loaded for processing. Requested processes are allocated memory until the entire physical memory is exhausted or the remaining space is insufficient to hold the requesting process. In this scheme the partitions used are of variable size, and the number of partitions is not defined at the system generation time.

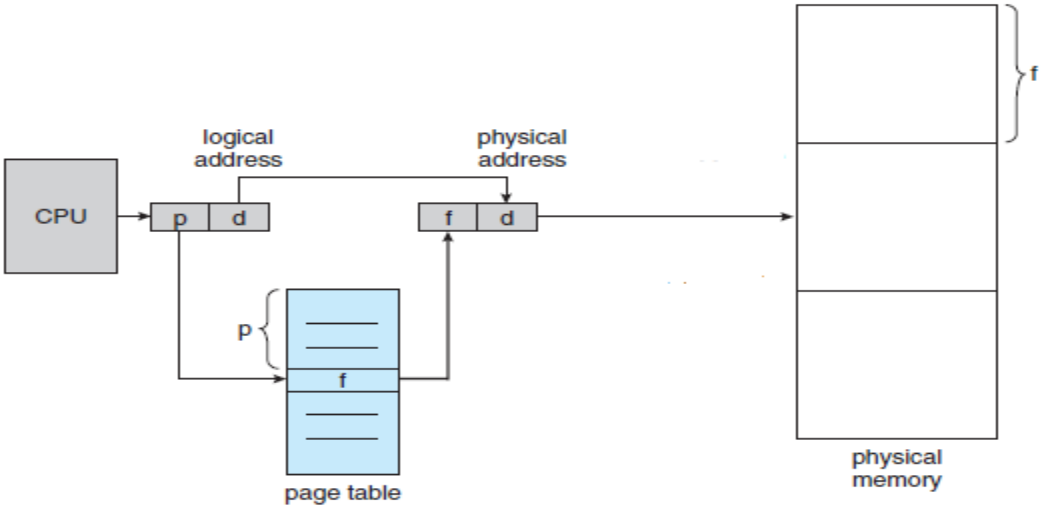
First fit. Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.
• **Best fit**. Allocate the smallest hole that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

Non-Contiguous memory management schemes:

In a Non-Contiguous memory management scheme, the program is divided into different blocks and loaded at different portions of the memory that need not necessarily be adjacent to one another. This scheme can be classified depending upon the size of blocks and whether the blocks reside in the main memory or not.

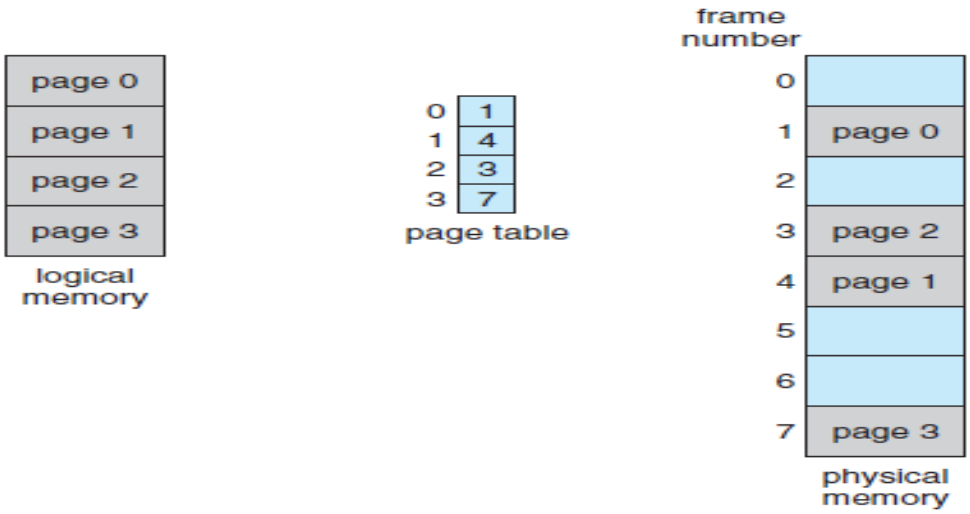
PAGING

Paging is a method used by operating systems to manage memory efficiently. It breaks physical memory into fixed-size blocks called “frames” and logical memory into blocks of the same size called “pages.” When a program runs, its pages are loaded into any available frames in the physical memory.



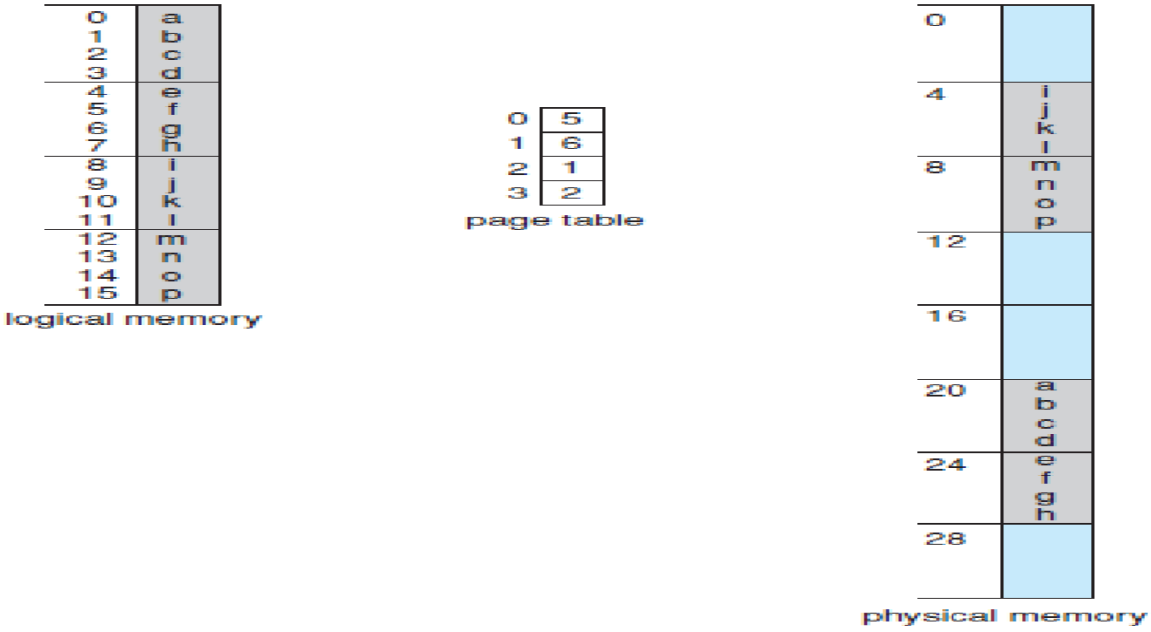
Every address generated by the CPU is divided into two parts: a **page number (p)** and a **page offset (d)**. The page number is used as an index into a **page table**. The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define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

The paging model of memory is shown in Figure below



Eample:-

Paging example for a 32-byte memory with 4-byte pages is



Here we can find the physical address by using the formula is

Frame number * size of the frame + offset value

Thus, logical address 0 maps to physical address 20 [= (5 × 4) +0].

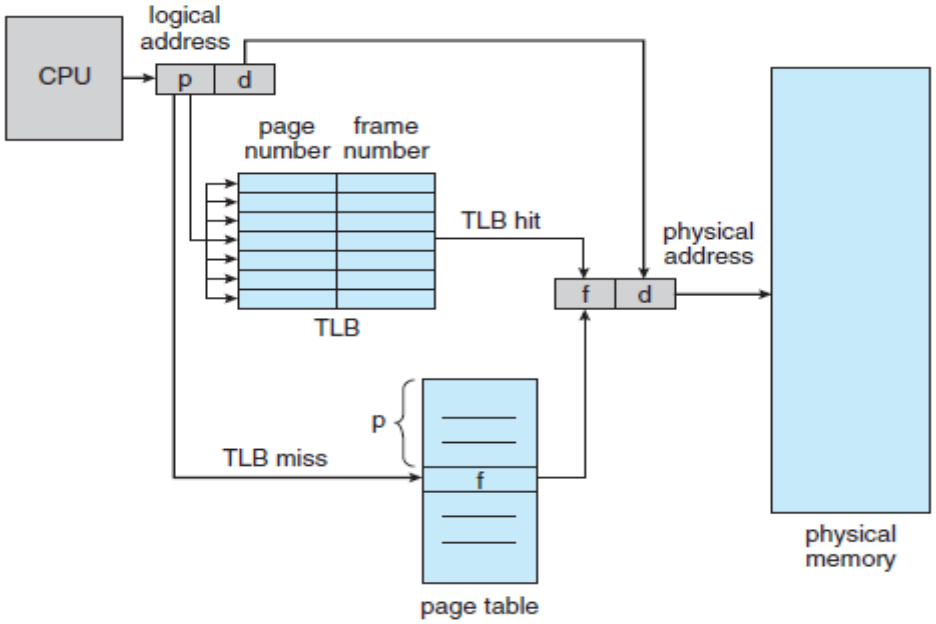
Logical address 3 (page 0, offset 3) maps to physical address 23 [= (5 × 4) +3].

Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6.

i. Hardware Support

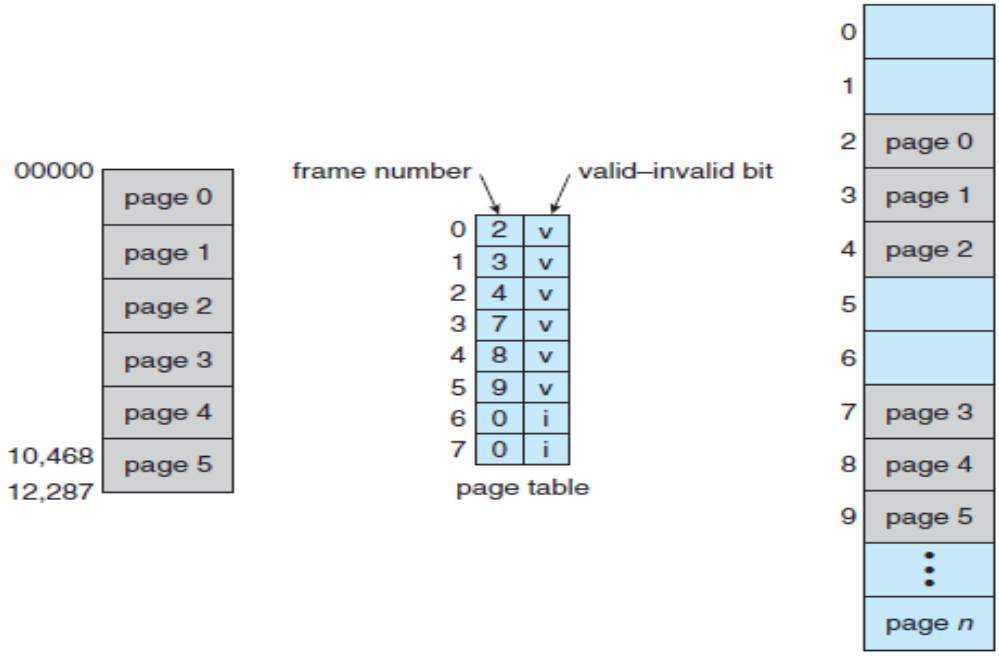
In the simplest case, the page table is implemented as a set of dedicated **registers**. These registers should be built with very high-speed logic to make the paging-address translation efficient. The use of registers for the page table is satisfactory if the page table is reasonably small. Most contemporary computers, however, allow the page table to be very large. For these machines, the use of fast registers to implement the page table is not feasible. Rather, the page table is kept in main memory, and a **page-table base register (PTBR)** points to the page table. Changing page tables requires changing only this one register, substantially reducing context-switch time.

The standard solution to this problem is to use a special, small, fast lookup hardware cache called a translation look-aside buffer **(TLB)**. The TLB is associative, high-speed memory. The standard solution to this problem is to use a special, small, fast lookup hardware cache called a translation look-aside buffer **(TLB)**. The TLB is associative, high-speed memory. If the page number is not in the TLB (known as a **TLB miss**), a memory reference to the page table must be made. Depending on the CPU, this may be done automatically in hardware or via an interrupt to the operating system.



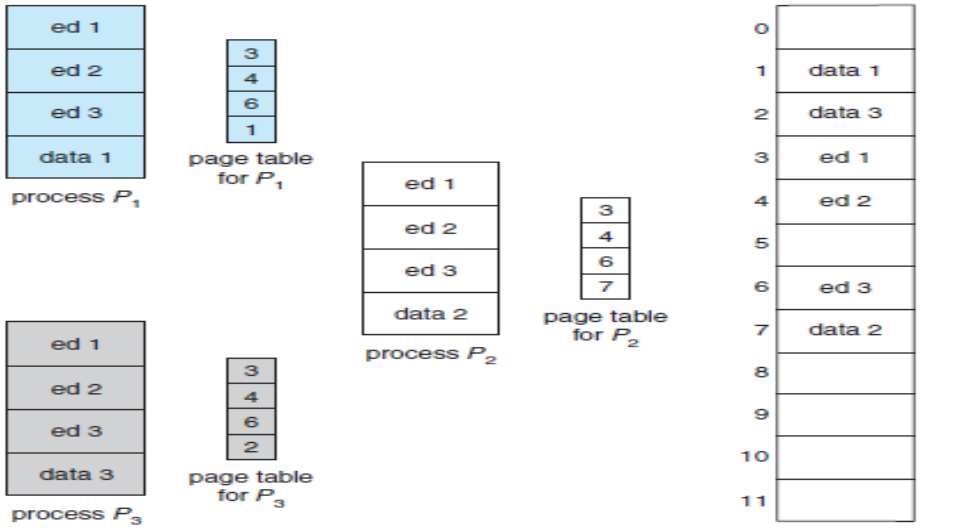
ii. Protection

One additional bit is generally attached to each entry in the page table: a valid–invalid bit. When this bit is set to valid, the associated page is in the process's logical address space and is thus a legal (or valid) page. When the bit is set to invalid, the page is not in the process's logical address space. Illegal addresses are trapped by use of the valid–invalid bit. The operating system sets this bit for each page to allow or disallow access to the page.



iii. Shared Pages

Shared pages are used to improve the performance of the paging system. There can be a scenario where multiple users might be executing the same jobs at a time. To avoid the duplication of the same pages in the same memory, it is preferable to share the pages. Shared pages are used in order to avoid having two copies of a page in memory at once. Shared pages can be used in place of physical RAM when more memory is needed. The most advantage of shared pages is that only one copy of a shared file exists in memory, reducing the overhead of pages and allowing more efficient use of RAM.

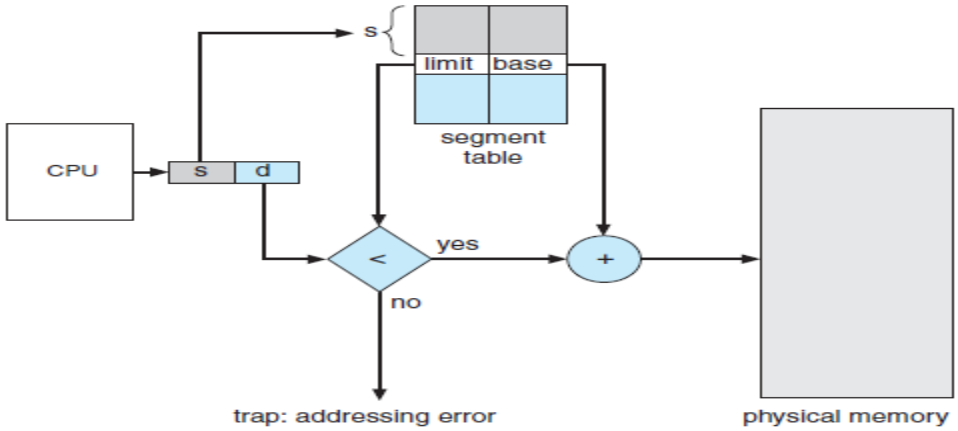


SEGMENTATION

Segmentation is a memory-management scheme that supports this programmer view of memory. In Operating Systems, Segmentation is a memory management technique in which the memory is divided into the variable size parts. Each part is known as a segment which can be allocated to a process. The details about each segment are stored in a table called a segment table. Segment table is stored in one (or many) of the segments.

Segment table contains mainly two information about segment:

1. Base: It is the base address of the segment
2. Limit: It is the length of the segment.

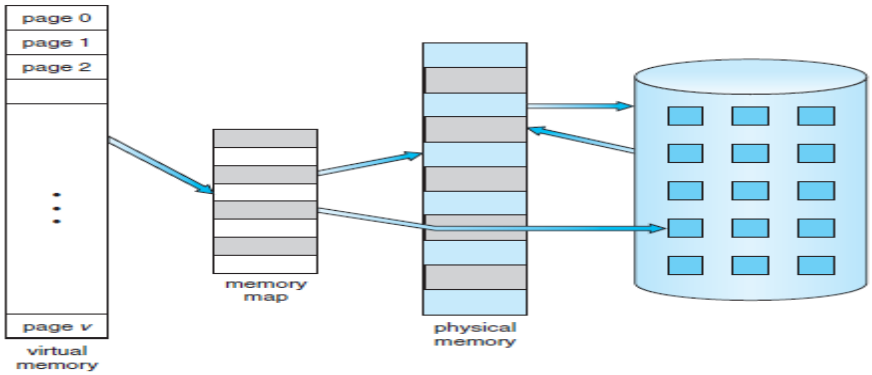


A logical address consists of two parts: a segment number, *s*, and an offset into that segment, *d*. the segment number is used as an index to the segment table. The offset *d* of the logical address must be between 0 and the segment limit. If it is not, we trap to the operating system (logical addressing attempt beyond end of segment). When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte. The segment table is thus essentially an array of base–limit register pairs.

VIRTUAL 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.

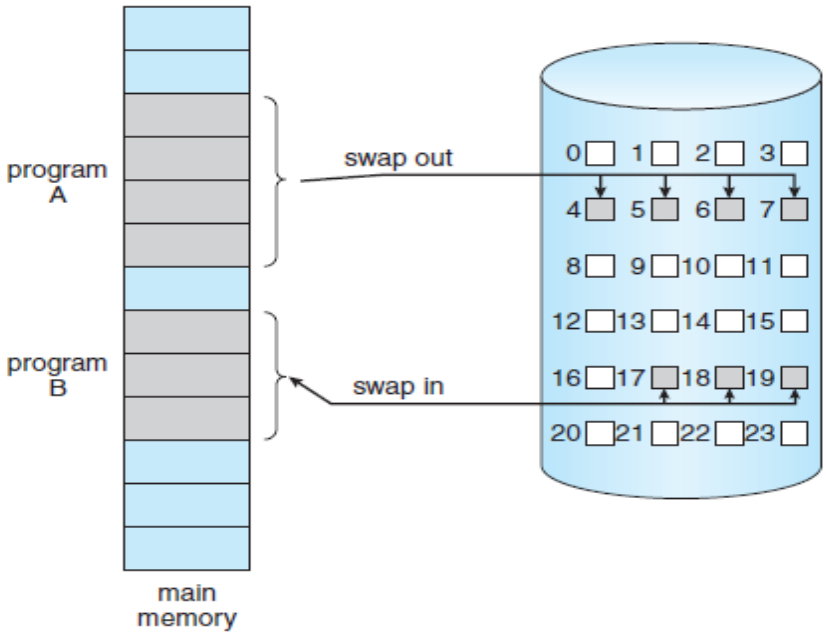
In this scheme, whenever some pages needs to be loaded in the main memory for the execution and the memory is not available for those many pages, then in that case, instead of stopping the pages from entering in the main memory, the OS search for the RAM area that are least used in the recent times or that are not referenced and copy that into the secondary memory to make the space for the new pages in the main memory.



DEMAND PAGING

Demand paging is a technique used in virtual memory systems where pages enter main memory only when requested or needed by the CPU. In demand paging, the operating system loads only the necessary pages of a program into memory at runtime, instead of loading the entire program into memory at the start. A page fault occurred when the program needed to access a page that is not currently in memory.

A demand-paging system is similar to a paging system with swapping where processes reside in secondary memory (usually a disk). When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory, though, we use a **lazy swapper**. A lazy swapper never swaps a page into memory unless that page will be needed.

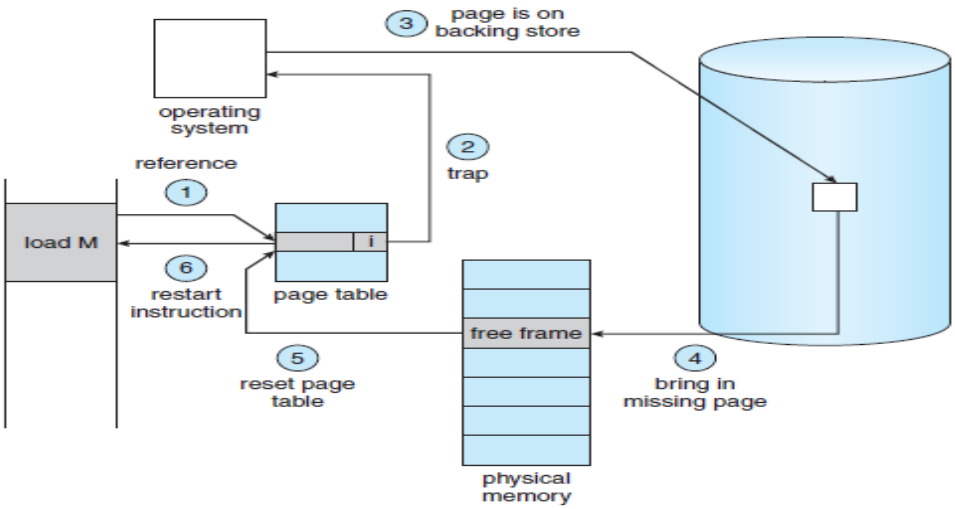


Page fault

The term “page miss” or “page fault” refers to a situation where a referenced page is not found in the main memory.

When a program tries to access a page, or fixed-size block of memory, that isn’t currently loaded in physical memory (RAM), an exception known as a page fault happens. Before enabling the program to access a page that is required, the operating system must bring it into memory from secondary storage (such a hard drive) in order to handle a page fault.

The procedure for handling this page fault is straightforward



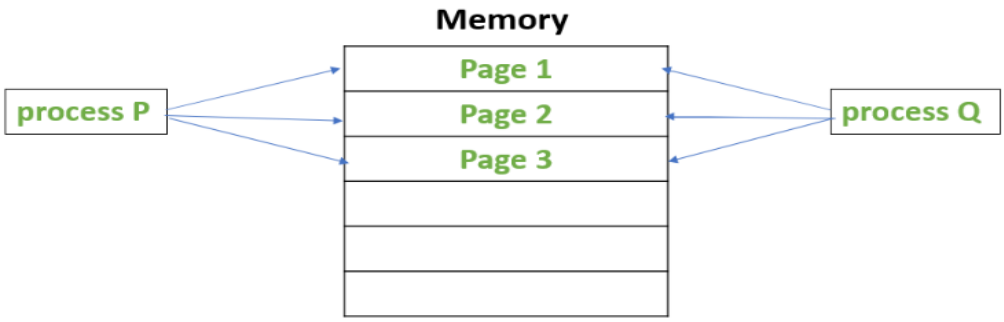
1. We check an internal table (usually kept with the process control block) for this process to determine whether the reference was a valid or an invalid memory access.
2. If the reference was invalid, we terminate the process. If it was valid but we have not yet brought in that page, we now page it in.
3. We find a free frame (by taking one from the free-frame list, for example).
4. We schedule a disk operation to read the desired page into the newly allocated frame.
5. When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.
6. We restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory.

Copy-on-Write

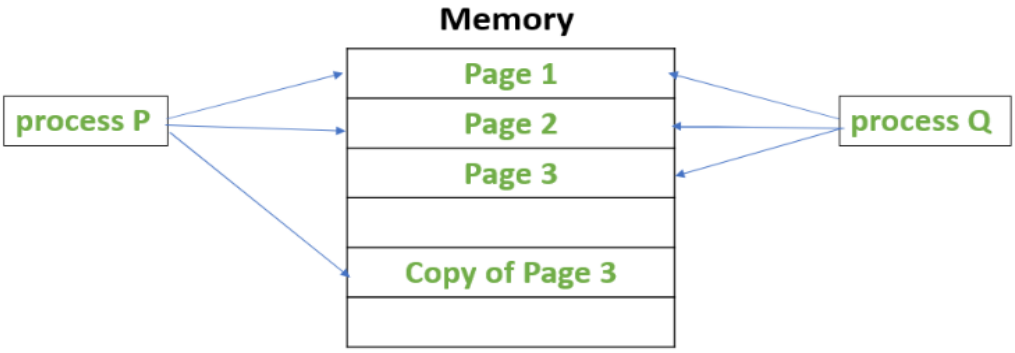
Copy on Write or simply COW is a resource management technique. One of its main uses is in the implementation of the fork system call in which it shares the virtual memory (pages) of the OS. In UNIX like OS, fork() system call creates a duplicate process of the parent process which is called as the child process.

The idea behind a copy-on-write is that when a parent process creates a child process then both of these processes initially will share the same pages in memory and these shared pages will be marked as copy-on-write which means that if any of these processes will try to modify the shared pages then only a copy of these pages will be created and the modifications will be done on the copy of pages by that process and thus not affecting the other process.

Suppose, there is a process P that creates a new process Q and then process P modifies page 3. The below figures shows what happens before and after process P modifies page 3.



Before process P modifies Page 3

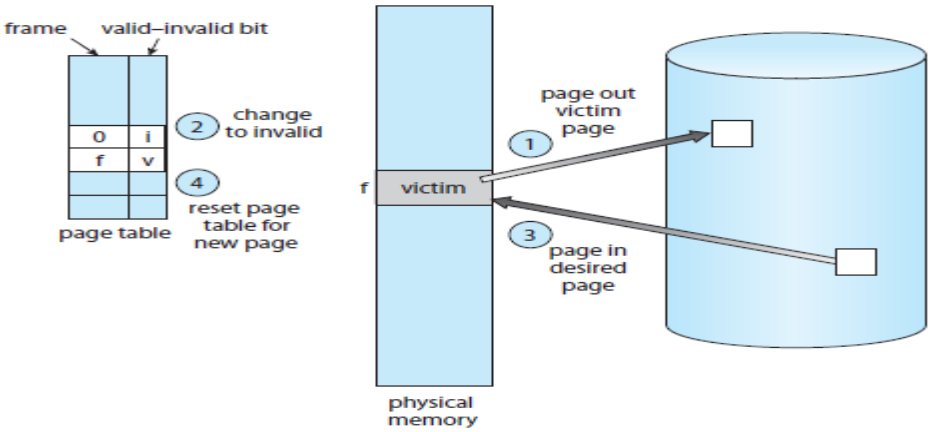


After process P modifies Page 3

PAGE REPLACEMENT

Page replacement is needed in the operating systems that use virtual memory using Demand Paging. As we know in Demand paging, only a set of pages of a process is loaded into the memory. This is done so that we can have more processes in the memory at the same time. When a page that is residing in virtual memory is requested by a process for its execution, the Operating System needs to decide which page will be replaced by this requested page. This process is known as page replacement and is a vital component in virtual memory management.

Page replacement takes the following approach. If no frame is free, we find one that is not currently being used and free it. We can free a frame by writing its contents to swap space and changing the page table (and all other tables) to indicate that the page is no longer in 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; change the page and frame tables accordingly.
3. Read the desired page into the newly freed frame; change the page and frame tables.
4. Continue the user process from where the page fault occurred.

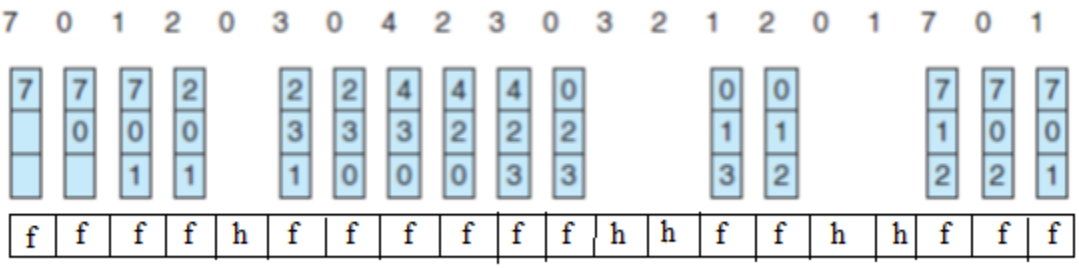
PAGE-REPLACEMENT ALGORITHM

1. FIFO Page Replacement

The simplest page-replacement algorithm is a first-in, first-out (FIFO) algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. Notice that it is not strictly necessary to record the time when a page is brought in. 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 reference string 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1 for a memory with three frames and calculate number of page faults by using FIFO (First In First Out) Page replacement algorithms.
Solution:-

1. Assume we have a fixed number of page frames (let's say 3).
2. The FIFO algorithm replaces the oldest page in memory when a new page must be loaded.
3. At the start, all the frames are empty. We will track the page faults that occur each time a new page is loaded into memory.
4. Page Not Found - - - > Page Fault(f), Page Found - - - > Page Hit(h)



Number of Page Hits = 5
Number of Page Faults = 15
The Ratio of Page Hit to the Page Fault = 5 : 15 - - - > 1 : 3 - - - > 0.33
The Page Hit Percentage = 5 *100 / 20 = 25%
The Page Fault Percentage = 100 - Page Hit Percentage = 100 - 25 = 75%

Example 2:-

Consider the reference string 6, 1, 1, 2, 0, 3, 4, 6, 0, 2, 1, 2, 1, 2, 0, 3, 2, 1, 2, 0 for a memory with three frames and calculate number of page faults by using FIFO (First In First Out) Page replacement algorithms.

Reference String:

6, 1, 1, 2, 0, 3, 4, 6, 0, 2, 1, 2, 1, 2, 0, 3, 2, 1, 2, 0

S. no	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
F3				2	2	2	4	4	4	2	2	2	2	2	2	2	2	2	2	0
F2		1	1	1	1	3	3	3	0	0	0	0	0	0	0	3	3	3	3	3
F1	6	6	6	6	0	0	0	6	6	6	1	1	1	1	1	1	1	1	1	1
Hit (H)/ Fault (F)	F	F	H	F	F	F	F	F	F	F	F	H	H	H	H	F	H	H	H	F

Number of Page Hits = 8
Number of Page Faults = 12
The Ratio of Page Hit to the Page Fault = 8 : 12 - - - > 2 : 3 - - - > 0.66
The Page Hit Percentage = 8 *100 / 20 = 40%
The Page Fault Percentage = 100 - Page Hit Percentage = 100 - 40 = 60%

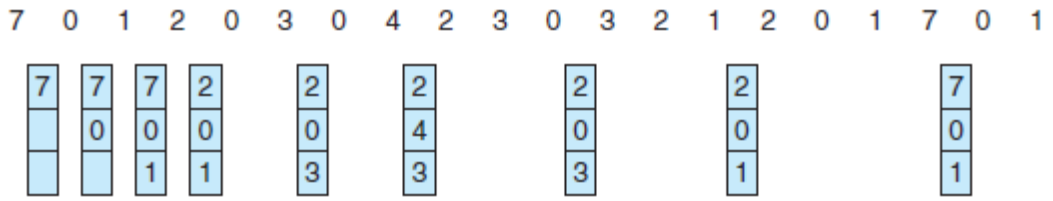
2. Optimal Page Replacement

The Optimal Page Replacement algorithm is a theoretical page replacement strategy used in memory management. Its goal is to achieve the minimum number of page faults by replacing the page that will not be used for the longest period of time in the future.

- When a page fault occurs (i.e., the requested page is not in memory), the algorithm looks ahead in the page reference string.
- It finds the page currently in memory that will not be used for the longest period of time in the future.
- That page is replaced with the new page.

Example:-

Consider the reference string 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1 for a memory with three frames and calculate number of page faults by using Optimal Page replacement algorithms.



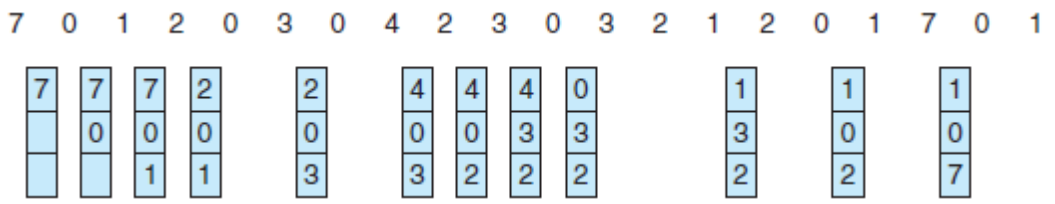
Number of Page Hits = 11
Number of Page Faults = 9

3. LRU Page Replacement

The **LRU (Least Recently Used)** page replacement algorithm is a widely used page replacement strategy in memory management. It replaces the page that has not been used for the longest period of time when a page fault occurs. The idea behind LRU is that pages that have been used recently are more likely to be used again soon, so it is better to replace the least recently used page.

Steps of the LRU Algorithm:

1. When a page fault occurs, the algorithm looks at the pages currently in memory.
2. It identifies which page was least recently used (i.e., the one with the longest time since its last use).
3. That page is replaced by the new page.



Number of Page Hits = 8
Number of Page Faults = 12

ALLOCATION OF FRAMES

The main memory of the operating system is divided into various frames. The process is stored in these frames, and once the process is saved as a frame, the CPU may run it. As a result, the operating system must set aside enough frames for each process. As a result, the operating system uses various algorithms in order to assign the frame.

Demand paging is used to implement virtual memory, an essential operating system feature. It requires the development of a page replacement mechanism and a frame allocation system. If you have multiple processes, the frame allocation techniques are utilized to define how many frames to allot to each one. A number of factors constrain the strategies for allocating frames:

1. You cannot assign more frames than the total number of frames available.
2. A specific number of frames should be assigned to each process. This limitation is due to two factors. The first is that when the number of frames assigned drops, the page fault ratio grows, decreasing the process's execution performance. Second, there should be sufficient frames to hold all the multiple pages that any instruction may reference.

There are mainly five ways of frame allocation algorithms in the OS. These are as follows:

1. Equal Frame Allocation
2. Proportional Frame Allocation
3. Priority Frame Allocation
4. Global Replacement Allocation
5. Local Replacement Allocation

Equal Frame Allocation

In equal frame allocation, the processes are assigned equally among the processes in the OS. For example, if the system has 30 frames and 7 processes, each process will get 4 frames. The 2 frames that are not assigned to any system process may be used as a free-frame buffer pool in the system.

Disadvantage

In a system with processes of varying sizes, assigning equal frames to each process makes little sense. Many allotted empty frames will be wasted if many frames are assigned to a small task.

Proportional Frame Allocation

The proportional frame allocation technique assigns frames based on the size needed for execution and the total number of frames in memory.

The allocated frames for a process **pi** of size **si** are **ai = (si/S)*m**, in which **S** represents the total of all process sizes, and **m** represents the number of frames in the system.

Disadvantage

The only drawback of this algorithm is that it doesn't allocate frames based on priority. Priority frame allocation solves this problem.

Priority Frame Allocation

Priority frame allocation assigns frames based on the number of frame allocations and the processes. Suppose a process has a high priority and requires more frames that many frames will be allocated to it. Following that, lesser priority processes are allocated.

Global Replacement Allocation

When a process requires a page that isn't currently in memory, it may put it in and select a frame from the all frames sets, even if another process is already utilizing that frame. In other words, one process may take a frame from another.

Advantages

Process performance is not hampered, resulting in higher system throughput.

Disadvantages

The process itself may not solely control the page fault ratio of a process. The paging behavior of other processes also influences the number of pages in memory for a process.

Local Replacement Allocation

When a process requires a page that isn't already in memory, it can bring it in and assign it a frame from its set of allocated frames.

Advantages

The paging behavior of a specific process has an effect on the pages in memory and the page fault ratio.

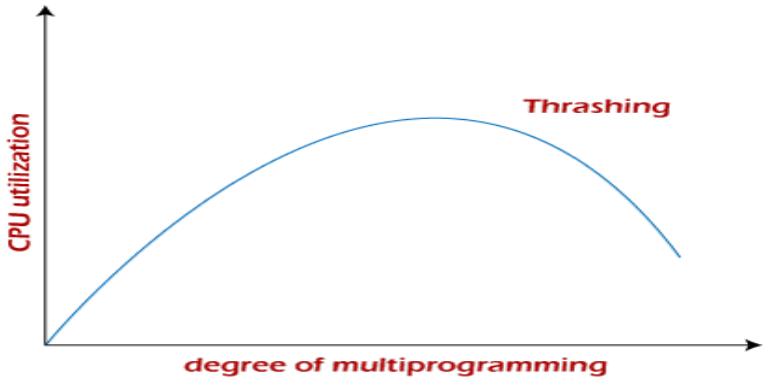
Disadvantages

A low priority process may obstruct a high priority process by refusing to share its frames.

THRASHING

In computer science, ***thrash*** is the poor performance of a virtual memory (or paging) system when the same pages are being loaded repeatedly due to a lack of main memory to keep them in memory.

Thrashing is when the page fault and swapping happens very frequently at a higher rate, and then the operating system has to spend more time swapping these pages. This state in the operating system is known as thrashing. Because of thrashing, the CPU utilization is going to be reduced or negligible.



The basic concept involved is that if a process is allocated too few frames, then there will be too many and too frequent page faults. As a result, no valuable work would be done by the CPU, and the CPU utilization would fall drastically.
The long-term scheduler would then try to improve the CPU utilization by loading some more processes into the memory, thereby increasing the degree of multiprogramming. Unfortunately, this would result in a further decrease in the CPU utilization, triggering a chained reaction of higher page faults followed by an increase in the degree of multiprogramming, called thrashing.

Techniques to Prevent Thrashing

The Local Page replacement is better than the Global Page replacement, but local page replacement has many disadvantages, so it is sometimes not helpful. Therefore below are some other techniques that are used to handle thrashing:

1. Locality Model

A locality is a set of pages that are actively used together. The locality model states that as a process executes, it moves from one locality to another. Thus, a program is generally composed of several different localities which may overlap.
For example, when a function is called, it defines a new locality where memory references are made to the function call instructions, local and global variables, etc. Similarly, when the function is exited, the process leaves this locality.

2. Working-Set Model

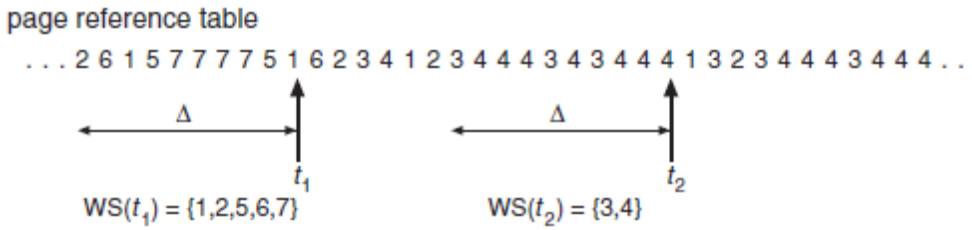
This model is based on the above-stated concept of the Locality Model.

The basic principle states that if we allocate enough frames to a process to accommodate its current locality, it will only fault whenever it moves to some new locality. But if the allocated frames are lesser than the size of the current locality, the process is bound to thrash.

According to this model, based on parameter A, the working set is defined as the set of pages in the most recent 'A' page references. Hence, all the actively used pages would always end up being a part of the working set.

The accuracy of the working set is dependent on the value of parameter A. If A is too large, then working sets may overlap. On the other hand, for smaller values of A, the locality might not be covered entirely.

If there are enough extra frames, then some more processes can be loaded into the memory. On the other hand, if the summation of working set sizes exceeds the frames' availability, some of the processes have to be suspended (swapped out of memory).



If D is the total demand for frames and WSS_i is the working set size for process i,

$D = \sum WSS_i$

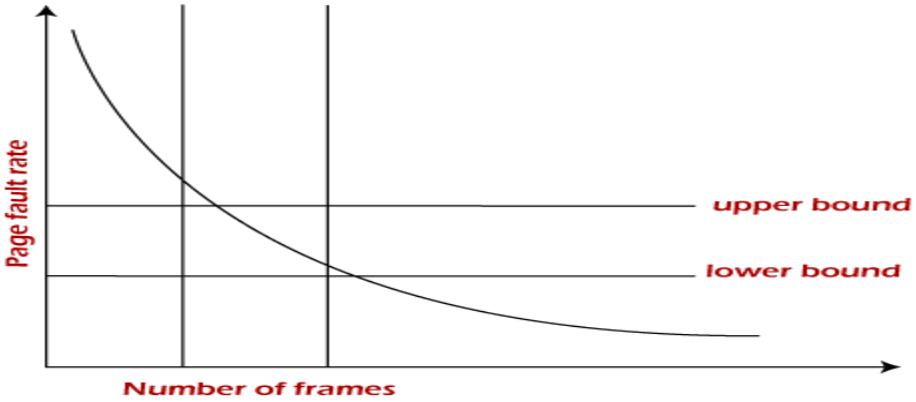
Now, if 'm' is the number of frames available in the memory, there are two possibilities:

- D>m, i.e., total demand exceeds the number of frames, then thrashing will occur as some processes would not get enough frames.
- D<=m, then there would be no thrashing.

This technique prevents thrashing along with ensuring the highest degree of multiprogramming possible. Thus, it optimizes CPU utilization.

3. Page Fault Frequency

A more direct approach to handle thrashing is the one that uses the Page-Fault Frequency concept.



The problem associated with thrashing is the high page fault rate, and thus, the concept here is to control the page fault rate.

If the page fault rate is too high, it indicates that the process has too few frames allocated to it. On the contrary, a low page fault rate indicates that the process has too many frames.

Upper and lower limits can be established on the desired page fault rate, as shown in the diagram.

If the page fault rate falls below the lower limit, frames can be removed from the process. Similarly, if the page faults rate exceeds the upper limit, more frames can be allocated to the process.

In other words, the graphical state of the system should be kept limited to the rectangular region formed in the given diagram.

If the page fault rate is high with no free frames, some of the processes can be suspended and allocated to them can be reallocated to other processes. The suspended processes can restart later.

MEMORY-MAPPED FILES

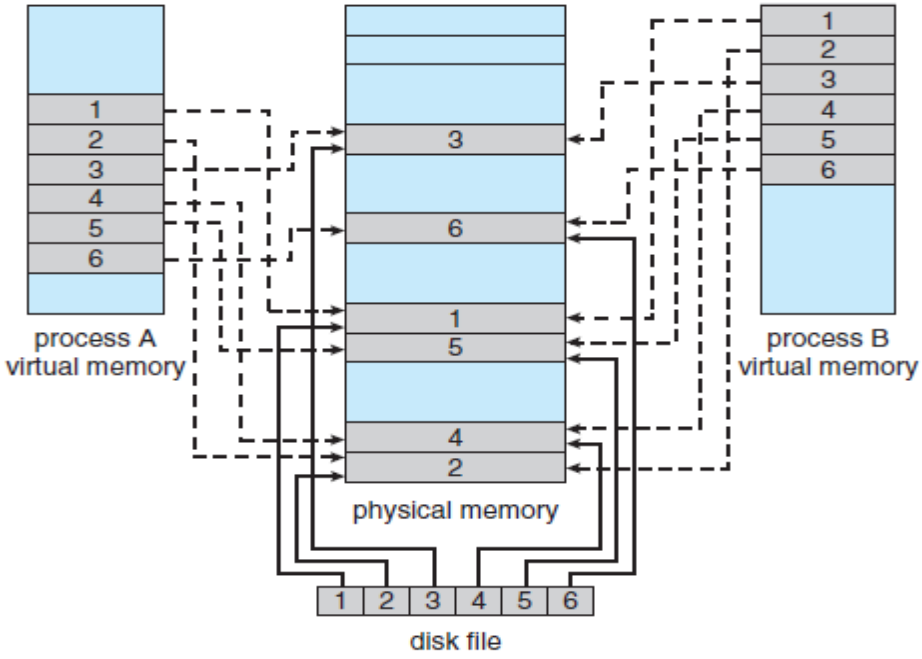
Rather than accessing data files directly via the file system with every file access, data files can be paged into memory the same as process files, resulting in much faster accesses (except of course when page-faults occur.) This is known as memory-mapping a file.

Basic Mechanism

- Basically a file is mapped to an address range within a process's virtual address space, and then paged in as needed using the ordinary demand paging system.
- Note that file writes are made to the memory page frames, and are not immediately written out to disk.
- This is also why it is important to "close( )" a file when one is done writing to it - So that the data can be safely flushed out to disk and so that the memory frames can be freed up for other purposes.
- Some systems provide special system calls to memory map files and use direct disk access otherwise. Other systems map the file to process address space if the special

system calls are used and map the file to kernel address space otherwise, but do memory mapping in either case.

- File sharing is made possible by mapping the same file to the address space of more than one process, as shown in below Figure. Copy-on-write is supported, and mutual exclusion techniques may be needed to avoid synchronization problems.
-



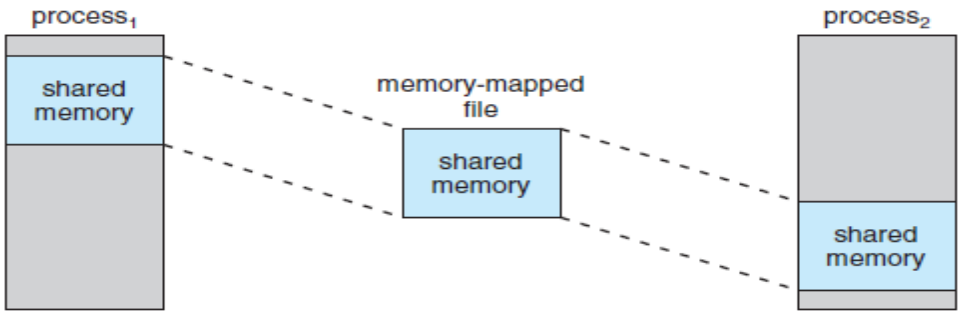
Shared memory can be implemented via shared memory-mapped files (Windows), or it can be implemented through a separate process (Linux, UNIX.)

Shared Memory in the Win32 API

• Windows implements shared memory using shared memory-mapped files, involving three basic steps: → Create a file, producing a HANDLE to the new file.

→ Name the file as a shared object, producing a HANDLE to the shared object.

→ Map the shared object to virtual memory address space, returning its base address as a void pointer (LPVOID). This is illustrated in below Figure



Memory-Mapped I/O

→ All access to devices is done by writing into (or reading from) the device's registers. Normally this is done via special I/O instructions.

→ For certain devices it makes sense to simply map the device's registers to addresses in the process's virtual address space, making device I/O as fast and simple as any other memory access. Video controller cards are a classic example of this.

→ Serial and parallel devices can also use memory mapped I/O, mapping the device registers to specific memory addresses known as I/O Ports, e.g. 0xF8. Transferring a series of bytes must be done one at a time, moving only as fast as the I/O device is prepared to process the data, through one of two mechanisms:

Programmed I/O (PIO), also known as polling. The CPU periodically checks the control bit on the device, to see if it is ready to handle another byte of data.

Interrupt Driven. The device generates an interrupt when it either has another byte of data to deliver or is ready to receive another byte.

ALLOCATING KERNEL MEMORY

Allocating kernel memory is a critical task in operating system design, as the kernel needs to manage memory efficiently and effectively to ensure optimal system performance. Kernel memory is often allocated from a free-memory pool different from the list used to satisfy ordinary user-mode processes. There are two primary reasons for this:

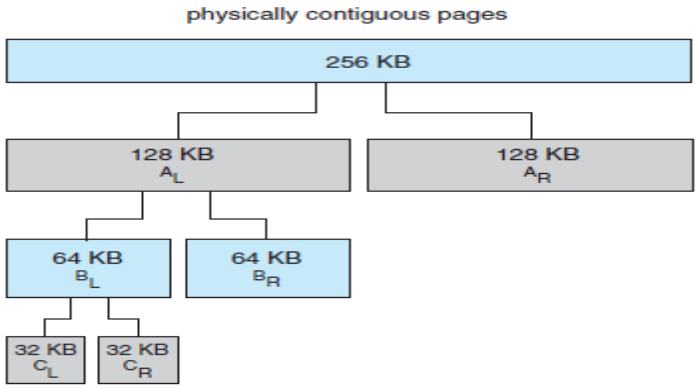
1. The kernel requests memory for data structures of varying sizes, some of which are less than a page in size. As a result, the kernel must use memory conservatively and attempt to minimize waste due to fragmentation. This is especially important because many operating systems do not subject kernel code or data to the paging system.

2. Pages allocated to user-mode processes do not necessarily have to be in contiguous physical memory. However, certain hardware devices interact directly with physical memory—without the benefit of a virtual memory interface—and consequently may require memory residing in physically contiguous pages.

Two common methods for allocating kernel memory are the buddy system and the slab system.

Buddy System

The buddy system allocates memory from a fixed-size segment consisting of physically contiguous pages. Memory is allocated from this segment using a **power-of-2 allocator**, which satisfies requests in units sized as a power of 2 (4 KB, 8 KB, 16 KB, and so forth). A request in units not appropriately sized is rounded up to the next highest power of 2. For example, a request for 11 KB is satisfied with a 16-KB segment.

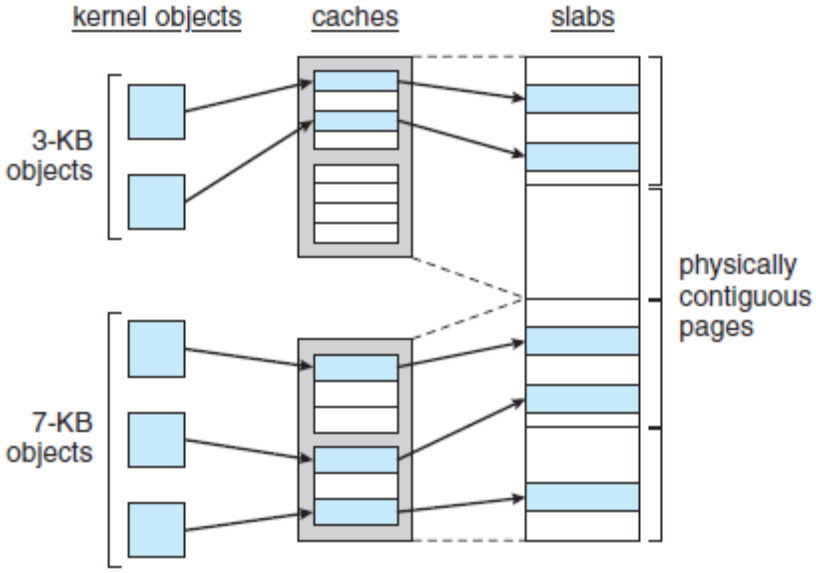


An **advantage** of the buddy system is how quickly adjacent buddies can be combined to form larger segments using a technique known as **coalescing**.

The obvious **drawback** to the buddy system is that rounding up to the next highest power of 2 is very likely to cause fragmentation within allocated segments. For example, a 33-KB request can only be satisfied with a 64- KB segment.

Slab Allocation

A second strategy for allocating kernelmemory is known as **slab allocation**. A **slab** is made up of one or more physically contiguous pages. A **cache** consists of one or more slabs. There is a single cache for each unique kernel data structure —for example, a separate cache for the data structure representing process descriptors, a separate cache for file objects, a separate cache for semaphores, and so forth. Each cache is populated with **objects** that are instantiations of the kernel data structure the cache represents. For example, the cache representing semaphores stores instances of semaphore objects, the cache representing process descriptors stores instances of process descriptor objects, and so forth.



The slab-allocation algorithm uses caches to store kernel objects. When a cache is created, a number of objects—which are initially marked as free—are allocated to the cache. The number of objects in the cache depends on the size of the associated slab.

In Linux, a slab may be in one of three possible states:

1. Full. All objects in the slab are marked as used.

2. Empty. All objects in the slab aremarked as free.

3. Partial. The slab consists of both used and free objects.

The slab allocator provides two main benefits:

1. No memory is wasted due to fragmentation.
2. Memory requests can be satisfied quickly.