STORAGE MANAGEMENT

Memory Management

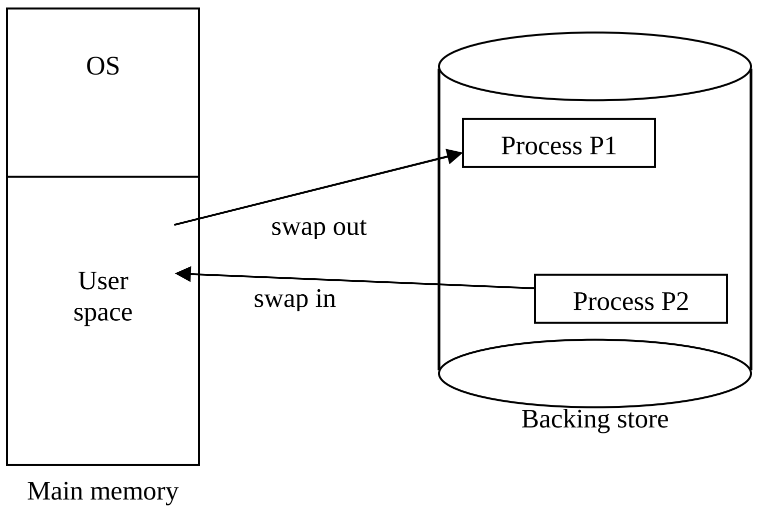
Memory management deals with the ways or methods through which memory in a computer system is managed. The method or scheme of managing memory depends upon its hardware design.

Memory is a large array of words or bytes with some addresses. Data read or written to the memory locations are memory address. An instruction execution cycle fetches an instruction from memory. This instruction is then decoded and may cause operands to be fetched from memory. When instruction is executed completely results may be stored back in the memory, so we will see how the operating system manages memory.

Swapping

A process to be executed should be in the memory. A process in memory can also be swapped out of memory temporarily to another storage e.g. hard disk and then brought back into the memory for execution. When a time quantum expires in Round Robin scheduling algorithm the memory manager swaps out a process and swaps in another process.

Similarly in Priority based scheduling when a higher priority process arrives and wants service, the memory manager swaps out the low priority process in order to load and execute high priority process. When high priority process finishes execution, the lower priority process will again be swapped in for execution. This is also called **Roll out**/**Roll-in**.



Swapping of two processes using a disk as a backing store

Normally a swapped out process is swapped in the same memory space that it was using previously. Although, it depends upon address binding. If binding is done at assembly or load time then process, cannot be moved to different location. If binding is done at execution time, then it is possible to swap a process into different memory location.

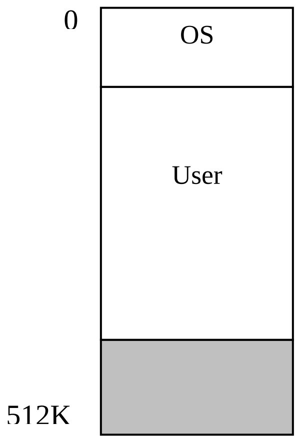
Swapping needs a backing store, that is commonly a disk etc. Disk should be large enough to hold copies of memory images of all users and should provide direct access to these memory images. The Operating System keeps information of all processes that are in memory or on the backing store. When the Scheduler decides to execute a process it calls the dispatcher. The dispatcher checks the memory regions to place the process in the memory, if there is no free memory region, the dispatcher swaps out the process currently in memory and swaps in the desired process.

Single-Partition Allocation

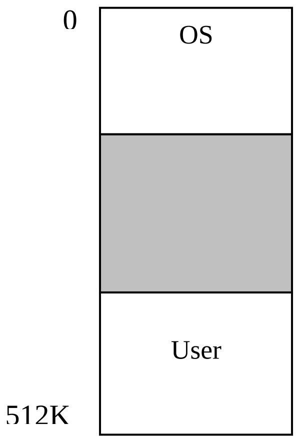
In Single-Partition Allocation scheme will divide memory into two partitions one is for the user and the other for the Operating System. Operating System is either kept in low or in high memory. Normally Operating System is kept in low memory. So when the Operating System is in low memory the user programs will be in high memory, and we'll have to protect Operating System code from the user process. This protection is implemented using the Base and Limit Registers.

The value of the Base should be fixed during the execution of the program, if the user addresses are bound to physical address by the use of base then it the base changes then these addresses will become invalid. So if the Base Address is changed, it should only be changed before the execution of the program.

Sometimes the size of the Operating System changes, e.g. a device driver that is not in use has occupied the memory. So it is better to release that memory so that other programs can use the free space but removing the device driver from memory changes the size of operating system.



Single User System



Loading User Program into high memory

So, in such cases when the size of Operating System changes dynamically, we load the user process into high memory down towards the base register value rather then from the base register towards high memory. Its advantage is that all unused space is in the middle and user as well as operating system can use this unused memory.

Multiple-Partition Allocation

In Multi Programming more than one process is kept in memory for execution and CPU switches rapidly between these processes. A problem is to allocate memory to different processes waiting to be brought into memory.

A simple way is to divide memory into fixed-size partition and each partition will have one process in it. When a partition becomes available another waiting process is brought in that place for execution.

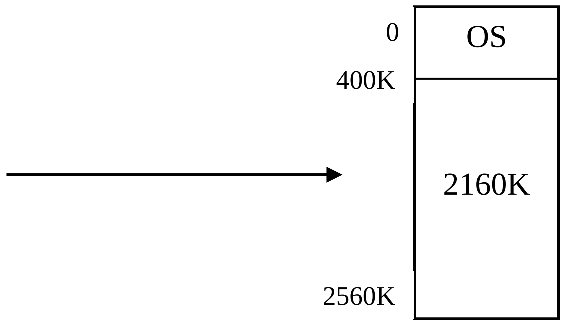
Problem in this solution is that the number of partitions bound degree of Multi-programming. Another problem is size e.g. if the size is large enough then small processes waste the space and if the size is small then it becomes difficult for larger programs to execute. This kind of scheme was used by IBM OS/360 Operating System, it is no longer in use.

Another technique is that operating system maintains a table indicating which parts of the memory are available and which are occupied. Initially, all the memory is available for user processes and is taken as one large block of memory also called a hole. When a process needs memory, we search the hole that should be enough for this process. If the hole is found it is allocated to that process and rest of the hole i.e. free memory is available to satisfy the requests of other processes or future requests of same process etc.

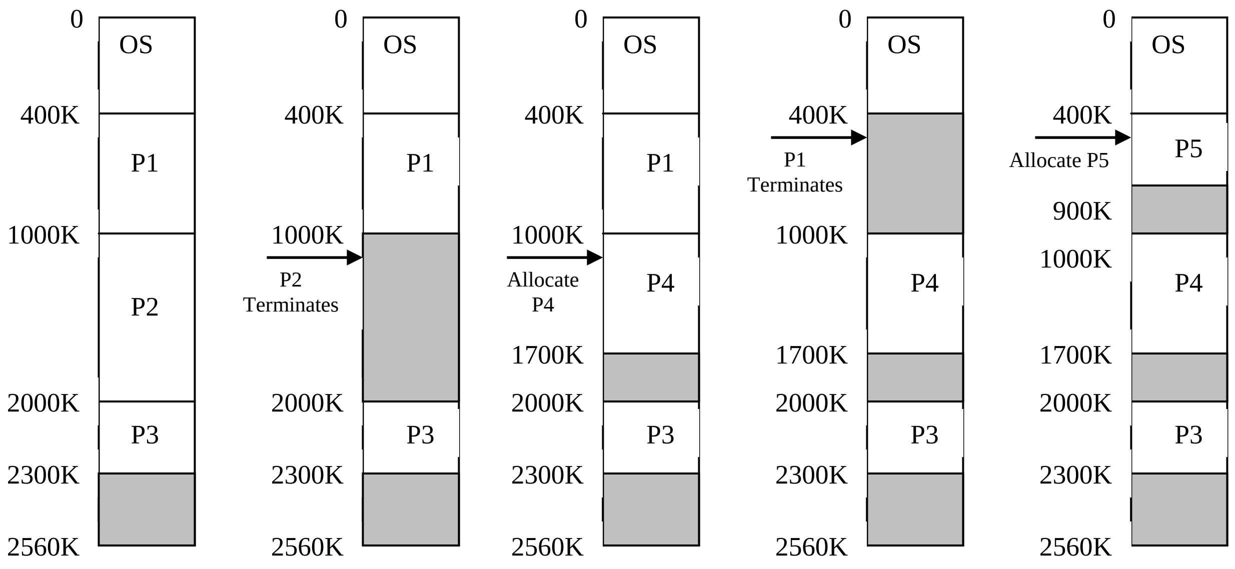
Consider a situation where we have 2560K memory available. Operating System is using 400K memory out of 2560K. So, 2160K memory is available to user processes. The memory allocated to P1 is 600K, P2 = 1000K and P3 = 300K using FCFS scheduling algorithm.

Job Queue

Process	Memory	Time
P1	600K	10
P2	1000K	5
P3	300K	20
P4	700K	8
P5	500K	15



So we have a hole of 260K that cannot be allocated to next process P4 because its size is 700K. Using RR Scheduling, process P2 terminates first and releases its memory, now process P4 will be given that memory hole which is of 1000K. P4 is of 700K, so we again have a block of 300 k. Similarly, after sometime P1 will be completed and P5 of size 500K will be given the memory hole of 600K freed by P1. So, another hole of 100K will be created.



Memory allocation using Multiple Partition Allocation

Now we have a set of holes scattered throughout the memory. When a process needs memory, we search a hole in which the process can be fitted, if the hole is large than after storing the process in larger hole the remaining space is split to create a new hole. When the process terminates, it releases its memory and again a block is available where new processes can be stored again. This is the example of Dynamic-Storage Allocation i.e. how to assign/allocate memory to processes. In reality we use First-fit, Best-fit or Worst-fit algorithms to allocate free hole to processes.

First-Fit

Allocates the first hole that is big enough where the process that needs execution can be easily fitted.

Best-Fit

In Best-Fit all holes are kept in some order by size and entire list is checked to allocate the smallest hole that can be best fitted.

Worst-Fit

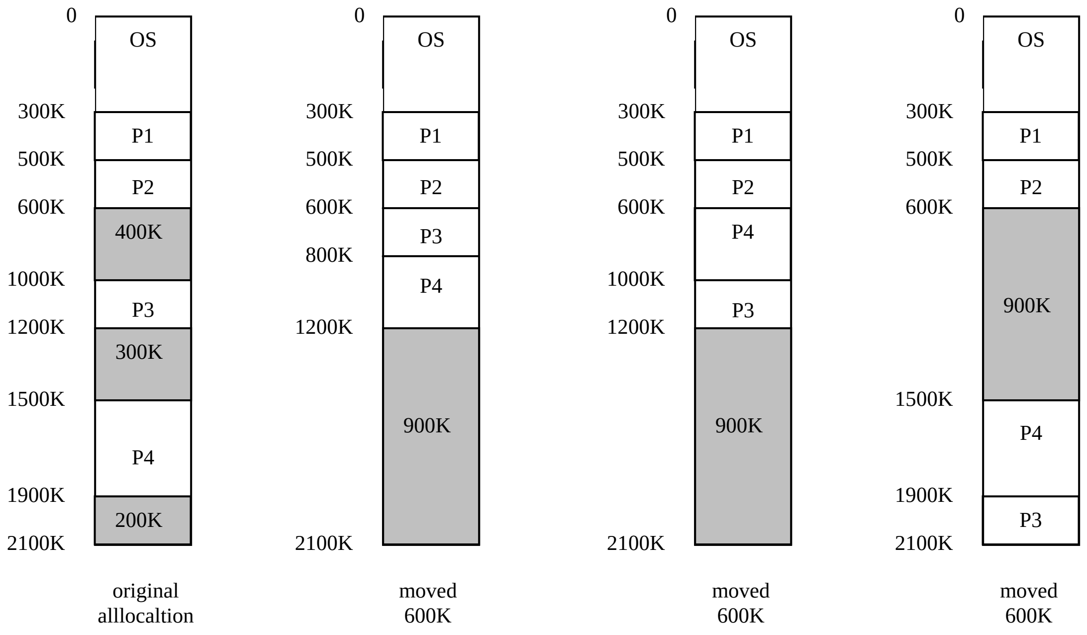
In worst fit all holes are kept in some order by size and entire list is checked to allocate the largest hole.

The problem with these algorithms is that they suffer from External Fragmentation. Processes are loaded and removed from memory, and free space is broken in to pieces. External Fragmentation is that we have enough total free space to satisfy a request but it is not contiguous but it is fragmented into small holes.

Compaction

Compaction is a solution for solving the problem caused by External Fragmentation. In Compaction, memory contents are shuffled to place all free memory together in one large block. Compaction is possible only if reallocation is dynamic and is done at execution time. If relocation is static and is done at load or assembly time then compaction will not be possible.

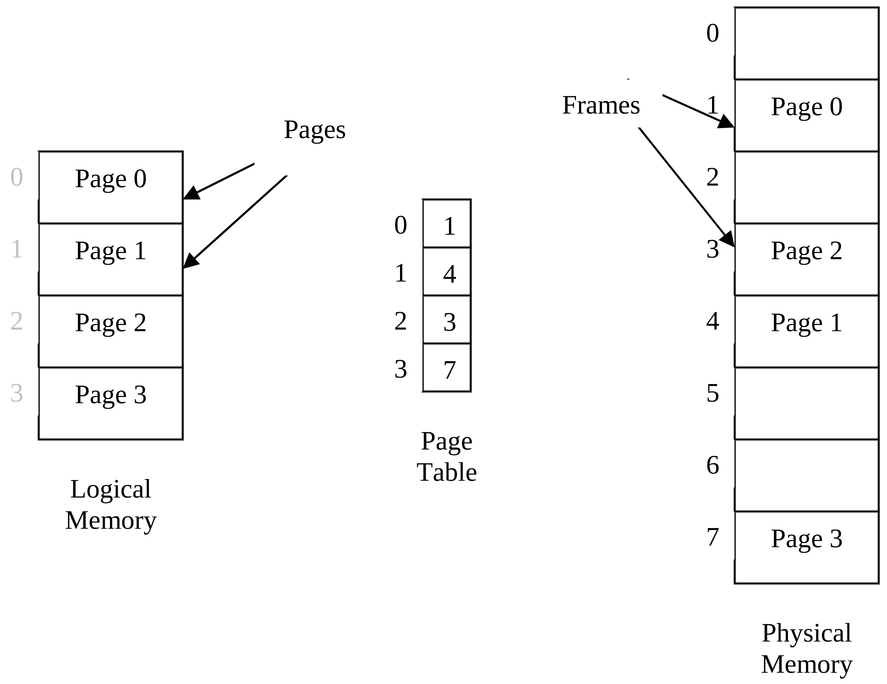
Algorithms are developed that move process in order to form one large hole of memory. One technique is to move all processes towards one end of memory and move all holes in other directions to form one big hole. Another technique may be to move some processes at one end and some other at the other end to form one big hole of memory in the center.



Comparison of some different ways to compact memory

Paging

Another solution of External Fragmentation is Paging. Normally all the processes need contiguous space in memory, but paging allows a process's memory to be non-contiguous and allowing a process to allocate memory wherever it is available. So problem of External Fragmentation is solved by Paging. Paging is used in many operating systems.

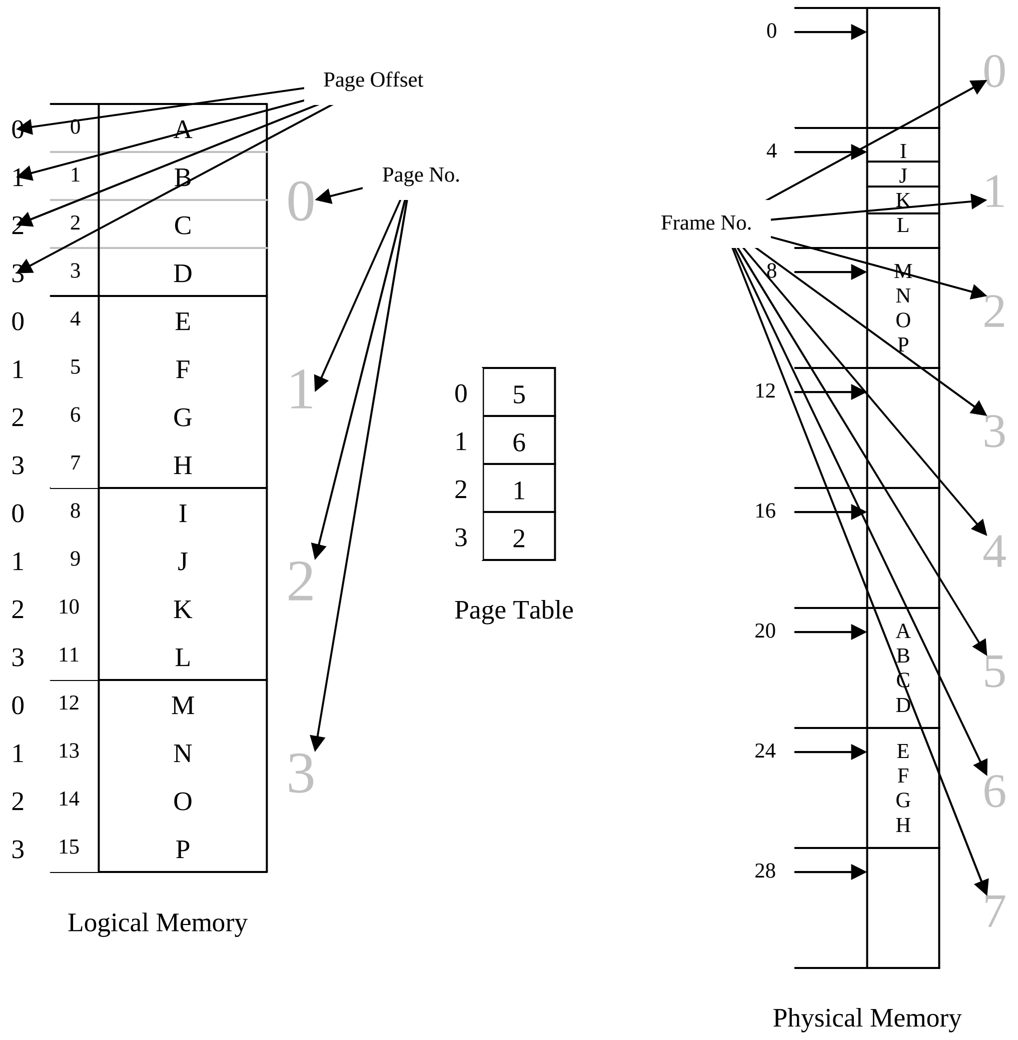


Paging Model of Logical and Physical Memory

In Paging, physical memory is broken into fixed size blocks called "**frames**". Logical memory is broken into blocks of the same size called "**pages**". When a process needs execution, its pages are loaded into any available frames from the backing store. The address generated by the CPU is divided into two parts.

i) Page number
ii) Page offset

Page number is used as an index into a page table.
Page offset is combined with the base address to define physical memory address.
Page Table contains the base address of each page in physical memory.



Paging example for a 32-word memory with 4-word Pages

We have a system that has page size of 4 words and a physical memory of 32 words (eight pages). As an example we'll see how the user's view of memory can be mapped into physical memory.

In the above mentioned diagram, Logical address 0 is page 0, offset 0. Indexing into the page table, we can check that page 0 is in frame 5. So, logical address 0 maps to physical address 20 that is
((5 x 4) + 0) = 20.

Similarly, logical address 3 (page 0, offset 3) maps to physical address 23 i.e. ((5 x 4) + 3) = 23. Logical address 4 is page 1, offset 0. According to the page table, page 1 is mapped to frame 6. Logical address 4 maps to physical address 24 i.e. ((6 x 4) + 0) = 24. Similarly, logical address 13 maps to physical address 9.

Segmentation

Users or programmers view about memory is that memory is not a linear array of words, but is a collection of variable-sized segments, i.e. each procedure, subroutine and functions etc are present in separate segments.

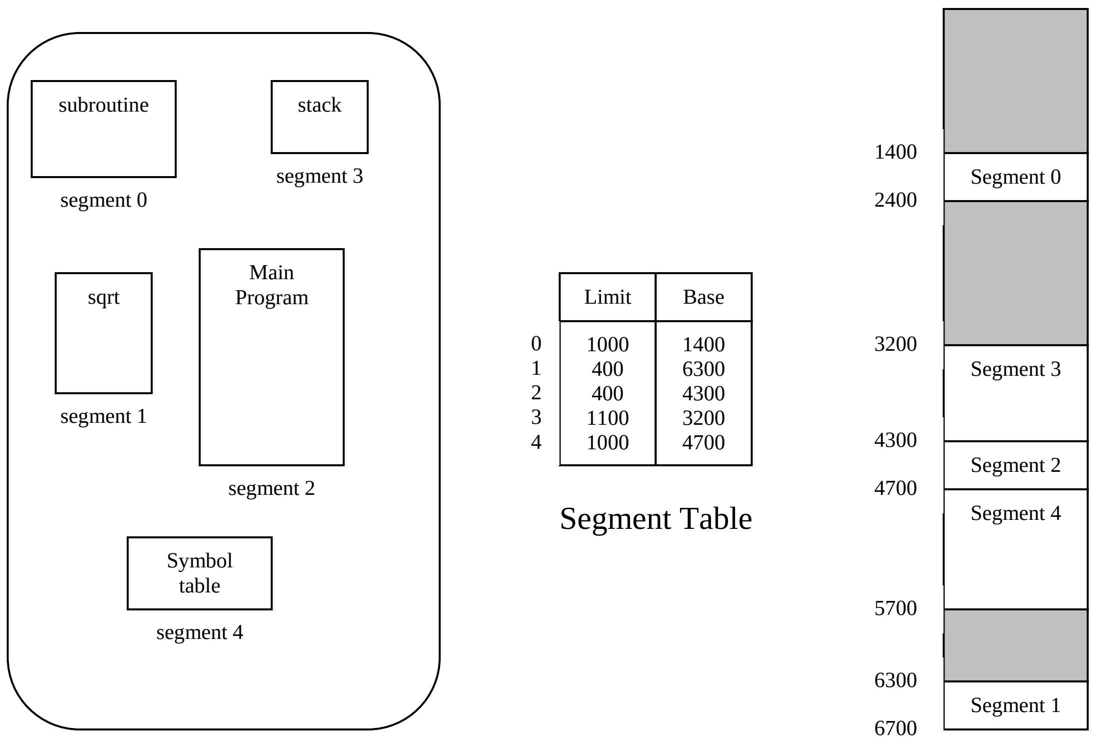
Addresses of segmentation specify both the segment name and the offset within the segment, and in actuality instead of segment name we use segment numbers. Intel 8086 supports segmentation as its only memory-management scheme and programs in this environment are separated into CODE, DATA and STACK segments etc.

Segment-Table

In order to map a two-dimensional user-defined address into one-dimensional physical address we use segment table. So, a logical address consists of two parts

i) Segment Number
ii) Offset into the Segment

Each entry of the Segment Table has a **Segment-Base** and **Segment-Limit**. The offset of the logical address must be between 0 and the Segment-Limit. If it is not, we trap the Operating System. If offset is legal, it is added to the Segment-Base to produce the address in physical memory of the desired word.



	Limit	Base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

Logical Address Space Physical Memory

Segmentation

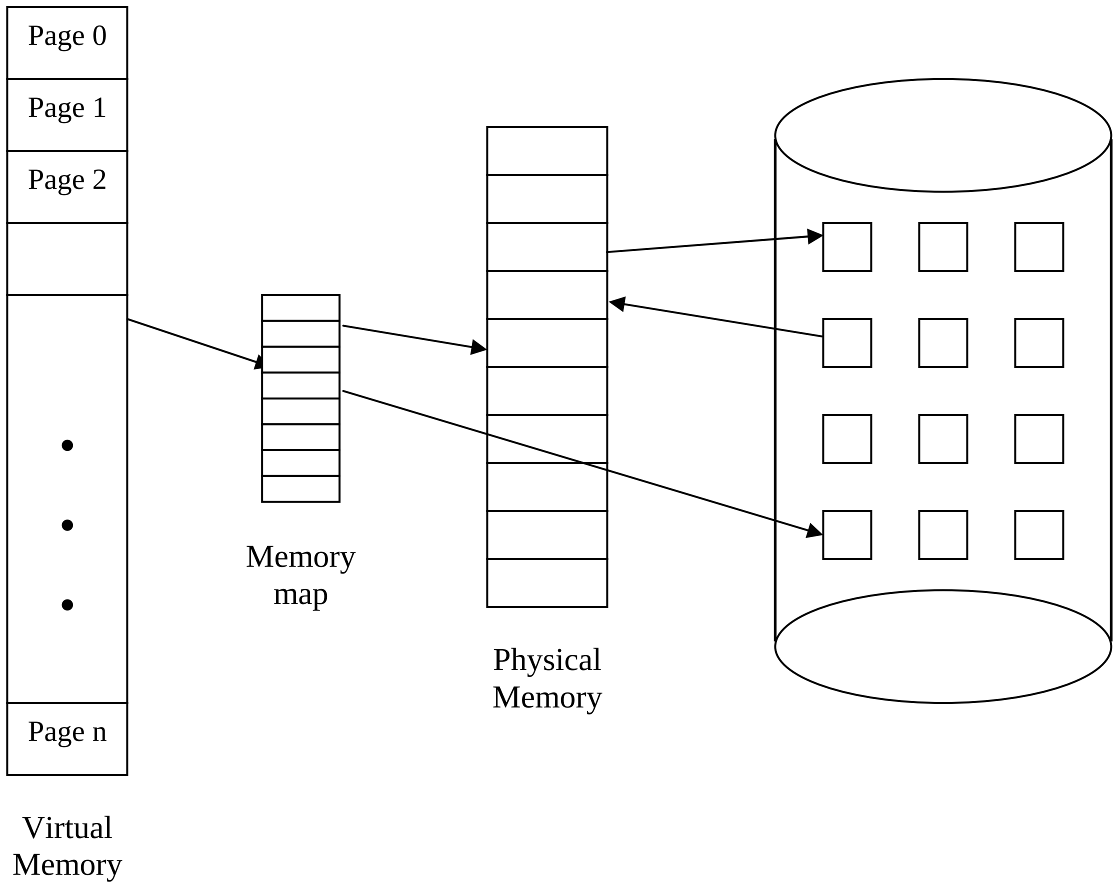
Suppose we have five segments i.e. 0-4 stored in physical memory. The segment table has separate entry for each segment i) The beginning address of the segment in physical memory i.e. (The Base) ii) the length of the segment (Limit). Segment 2 is 400 bytes long and begins at location 430 in physical memory. Now to generate on address of 53rd byte of 2nd Segment 4300+53=4353. Similarly, byte 1222 of segment 0 would result in a trap to Operating System as this segment is only 1000 bytes long.

Virtual Memory

Virtual Memory is a technique that allows the execution of processes that may not be completely in memory. So, the advantage of using this scheme is that programs can be larger than physical memory and users doesn't need to bother about the memory capacity or program size etc. Disadvantages are that this scheme decreases the performance and is difficult to implement.

So, the concept or methodology in Virtual Memory is

- It allows the combined size of program, data and stack to exceed the size of physical memory.
- Operating System to keep only those parts in memory which are currently in use. (We can run 1MB program in a machine having 256K of Physical RAM)



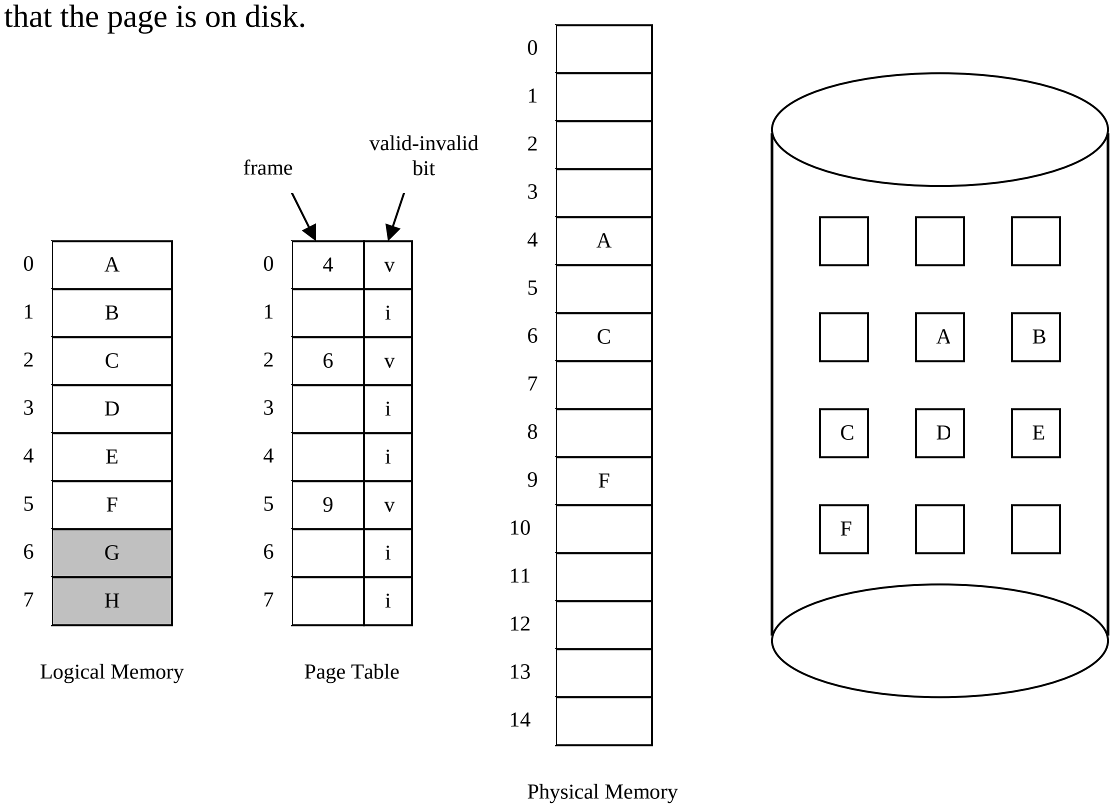
Virtual Memory can be much larger than physical memory

Demand Paging

Demand Paging is similar to Paging with Swapping and is a technique of accessing Virtual Memory.

Processes reside on secondary memory i.e. hard disk etc and when we want to execute a process, we swap it into memory. Now instead of swapping the whole process into memory we swap only those pages of the process that are needed, as we are talking about a process as a sequence of pages. So, bringing only those pages into memory from a process that are needed, our swap time decreases and our physical memory contains the information that is required.

To implement this scheme we need hardware support i.e. only bit called "Valid-Invalid" bit is attached to each entry in the page table. When this bit is set to "Valid", it indicates that the associated page is in memory. If the bit is set to "Invalid", it indicates that the page is on disk.



Page Table
When some pages are not in main memory

If a process tries to use a page that was not brought into memory than a Page-Fault will occur. When a Page-Fault occurs

i) We check whether the reference was a valid or invalid memory address (with the help of PCB)
ii) If it was invalid, we terminate the process. If it was valid but we have not brought that page

iii) We'll have to find the free frame on the physical memory to bring the page from backing store.
iv) The desired page is then brought from backing store to physical memory.
v) Now we update the page table entries to indicate that the desired page is in memory.
vi) We again start the instruction that was interrupted by the illegal address trap.

One case may be that we start executing a process with no pages in memory. This process will immediately cause a page fault and the desired page will be brought into physical memory and similarly, whenever page is not found in page table page fault will bring that page into physical memory and will update the page table and will continue to execute and so on till the process finishes.

So, in Demand Paging only those pages are brought into memory that are required.

Page Replacement

A user process is executing and a page fault occurs. The operating system checks where the desired page is residing on the disk, but another problem arises that there are not free-frames on the free-frame list as all the memory is in use.

One was to solve this problem is that Operating System should terminate that process, but this is not a good technique as Operating System tries to improve the CPU utilization and manage resources in a better way.

Another way to solve this problem is to swap out a process and freeing all its frames so that the desired page can now be placed in the page table.

So, Page Replacement technique is that if no frame is free than find the frame that is not currently in use and free it. A frame is freed by writing its contents etc to indicate that the page is not longer in memory. The freed frame can now be used to hold the page for which the process faulted. The page-fault service routine is now modified to include Page-Replacement

1. Find the location of desired page on disk.
2. Find a free frame
    a. If there is a free frame, use it.
    b. Otherwise, use a Page-Replacement algorithm to select a victim frame.
    c. Write the victim page to the disk, change the page and frame tables.
3. Read the desired page into the newly free frame and change the page and frame tables.
4. Restart the use process.

When no frames are free, two page transfers (one out and one in) are required that increases time and reduced performance.

Modify (Dirty) bit

The overhead of the two page transfers when no frames are free can be reduced with the use of modify (dirty) bit. In the hardware, each page or frame may have a modify bit associated with it. The modify bit for a page is set by the hardware whenever any word or byte in the page is written, thus indicating that the page has been modified.

When a page is selected for replacement, its modify bit is examined. If bit is set, it means that the page is modified since it was read in from the disk, so, in this case we write that page to the disk. If modify bit is not set, it means that the page is not modified since it was read in from the disk so there is not need for writing that page to the disk. Like this instead of two page transfers we transferred only one page that reduced our time and increases the performance.

Page-Replacement Algorithms

Page-Replacement Algorithm discusses about the selection of frames that are to be replaced. We know that disk I/O is expensive and a slight improvement in Page-Replacement Algorithm will improve system performance.

FIFO Page-Replacement Algorithm

The First-In, First-Out is the simplest Page-Replacement Algorithm. In FIFO Page-Replacement Algorithm, when a page needs to be replaced, the oldest-page is chosen. FIFO queue can be created to hold all pages in memory.

The performance of FIFO Page-Replacement Algorithm is not always good. The Page Replaced may be the one that was used a long time ago and is no longer needed or it could be an active page that is in use. Everything will be working correctly even if an active page is replaced but a bad replacement choice increases the page-fault rate and slows process execution.

Optimal Page-Replacement Algorithm

An Optimal Page-Replacement Algorithm has the lowest Page-Fault rate of all algorithms. In Optimal Algorithm we replace the page that will not be used for the longest period of time.

The Optimal Page-Replacement Algorithm is difficult to implement, as it requires the future knowledge of the reference. So, Optimal Algorithm is used mainly for comparison studies.

LRU Page-Replacement Algorithm

In FIFO we were using the time when a page was brought into memory. In Optimal Algorithm we were using the time when a page is to be used. In LRU, Algorithm we use recent past i.e. we use the time when that page was last used. So, LRU Page

Replacement Algorithm chooses the page that has not been used for the longest period of time.

LRU Algorithm is used as a Page-Replacement Algorithm and is considered to be quite good. LRU Page-Replacement Algorithm requires a little hardware support too.

Second-Chance Page-Replacement Algorithm

In Second-Chance Algorithm when a page is selected, we first check its reference bit. If the value is 0 we replace this page if the value is 1 we give that page a second chance and move on to select the next FIFO Page.

When a page gets a second chance, its reference bit is cleared and its arrival time is reset to the current time. So, a page that is given a second chance will not be replaced until all other pages are replaced (or given second chance). Similarly, if a page is used enough and its reference bit is set i.e. 1, then it will not be replaced.

LFU Page-Replacement Algorithm

Least Frequently Used Page Replacement Algorithm keeps a counter of the number of references that have been made to each page. The Page with the smallest count is replaced. It is obvious that an actively used page will have a large reference count and will not be replaced.

A problem occurs when a page is used very heavily during the initial phase, but then it is never used, its reference count is high but it is no longer in use, so it will not be replaced and will continue to occupy the space.

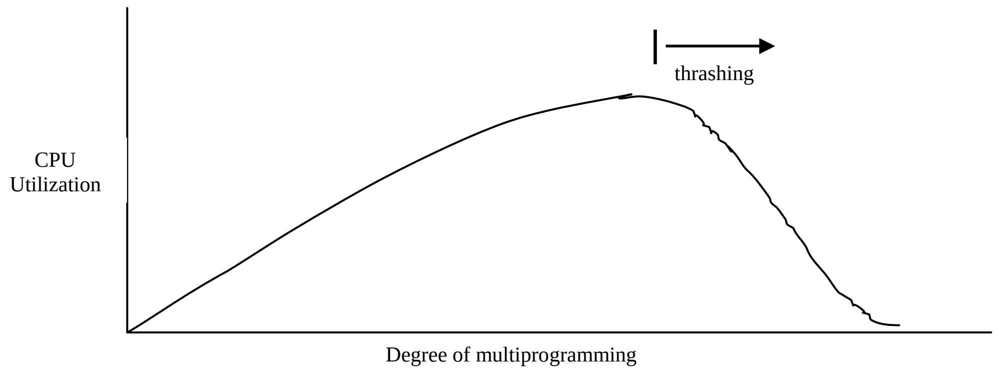
A solution for this problem is to shift the count down by 1 at regular intervals

Thrashing

When the paging activity in a system is very high we call it Thrashing. If a process spends more of its time in paging instead of executing then it is called that the process is Thrashing.

Consider an example where all the pages of a process are in use and we also have limited number of frames available in the physical memory. Process page faults, so it must replace a page, as all the pages are in active use, it will replace a page that will be needed again quickly. So, for that page it will again page fault and will replace an active page that too will be needed quickly, and like this process page faults very quickly again, and again, and again. The process continues to fault, replacing pages for which it will again page fault and bring back that page again.

The problem caused by Thrashing is performance. Thrashing reduces the performance of the system.



Thrashing

Operating System monitors CPU Utilization. If CPU Utilization is low, we increase the degree of multiprogramming by introducing a new process to the system. Now many processes are running and one out of them needs more frames. It starts faulting and takes pages from other processes. Other processes do the same and for this reason the speed or CPU Utilization decreases. Operating System sees that the CPU Utilization has decreased, it introduces new processes in order to increase the degree of multiprogramming but due to the high page-fault rate thrashing starts and slows down the speed even more.

So, at this point to increase the CPU Utilization and stop Thrashing, we must decrease the degree of multiprogramming.

Thrashing can be limited as if a process starts thrashing it can not take frames from other processes and cause them to thrash. Thrashing can be reduced if pages are replaced with regard to the process of which they are a part.