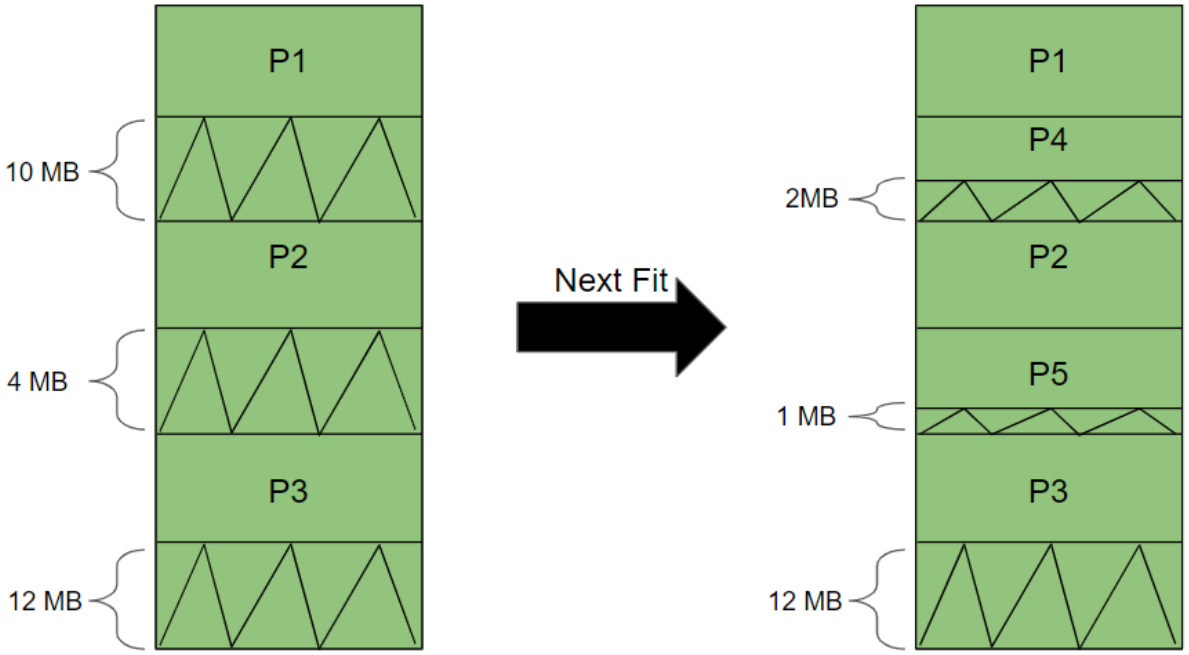
3. **Next Fit:** Next fit is similar to the first fit but it will search for the first sufficient partition from the last allocation point. Let's look at the example below:



Suppose in this setup a process P4 of size 8 MB and a process P5 of size 3 MB makes a request. At first, the linked list is traversed from the top of the memory and wherever, a size of 8 MB or more is available, the process gets allocated. In this case, the P4 gets allocated in the 10 MB hole leaving another hole of 2 MB. When the next request P5 of 3 MB arrives then, instead of traversing the whole list from the top, it is traversed from the last stop i.e., from P4 and process P5 gets allocated into the 4 MB block.

4. **Worst Fit:** Allocate the process to the partition which is the largest sufficient among the freely available partitions available in the main memory.

After doing some analysis, it has been found out that the First fit is the best approach since the best-fit algorithm needs to traverse the whole list every time a request is made.

- Paging in Operating System

Paging is a memory management scheme that eliminates the need for contiguous allocation of physical memory. This scheme permits the physical address space of a process to be non - contiguous. Let's revise various types of address and spaces:

- **Logical Address or Virtual Address (represented in bits):** An address generated by the CPU
- **Logical Address Space or Virtual Address Space(represented in words or bytes):** The set of all logical addresses generated by a program
- **Physical Address (represented in bits):** An address actually available on memory unit
- **Physical Address Space (represented in words or bytes):** The set of all physical addresses corresponding to the logical addresses

Example:

- If Logical Address = 31 bit, then Logical Address Space = 2^{31} words = 2 G words (1 G = 2^{30})
- If Logical Address Space = 128 M words = $2^7 * 2^{20}$ words, then Logical Address = $\log_2 2^{27}$ = 27 bits
- If Physical Address = 22 bit, then Physical Address Space = 2^{22} words = 4 M words (1 M = 2^{20})
- If Physical Address Space = 16 M words = $2^4 * 2^{20}$ words, then Physical Address = $\log_2 2^{24}$ = 24 bits

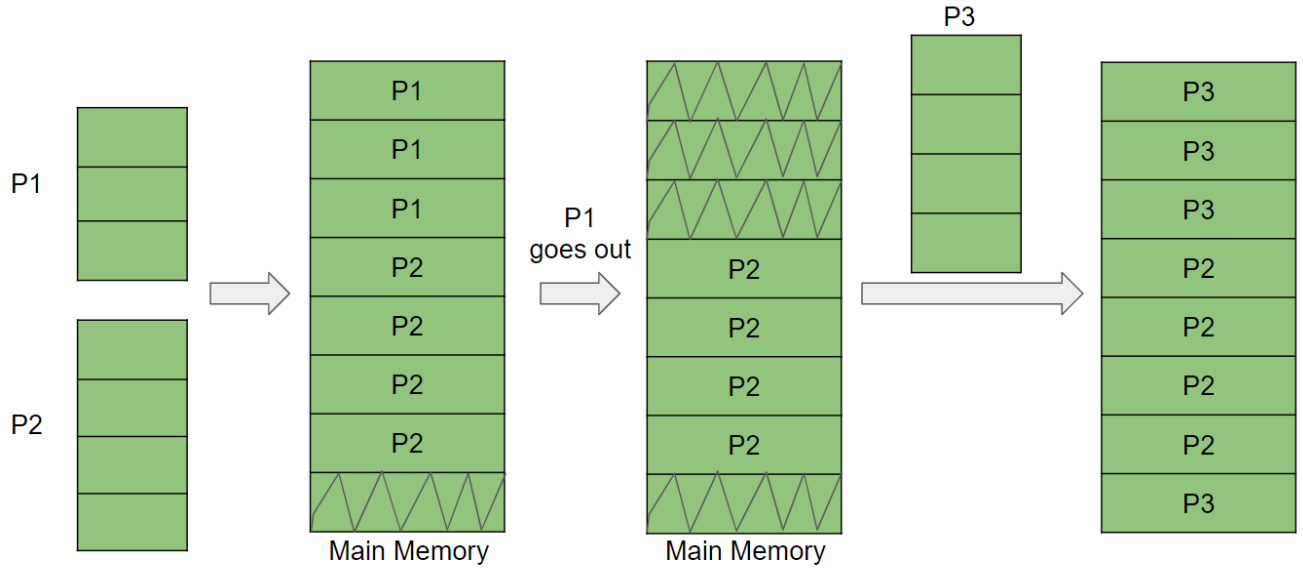
The mapping from virtual to physical address is done by the memory management unit (MMU) which is a hardware device and this mapping is known as paging technique.

- **Frames:** The Physical Address Space is conceptually divided into a number of fixed-size blocks.
- **Pages:** The Logical address Space is also split into fixed-size blocks, called .
  Note: Page Size is always equal to Frame Size

Let us consider an example:

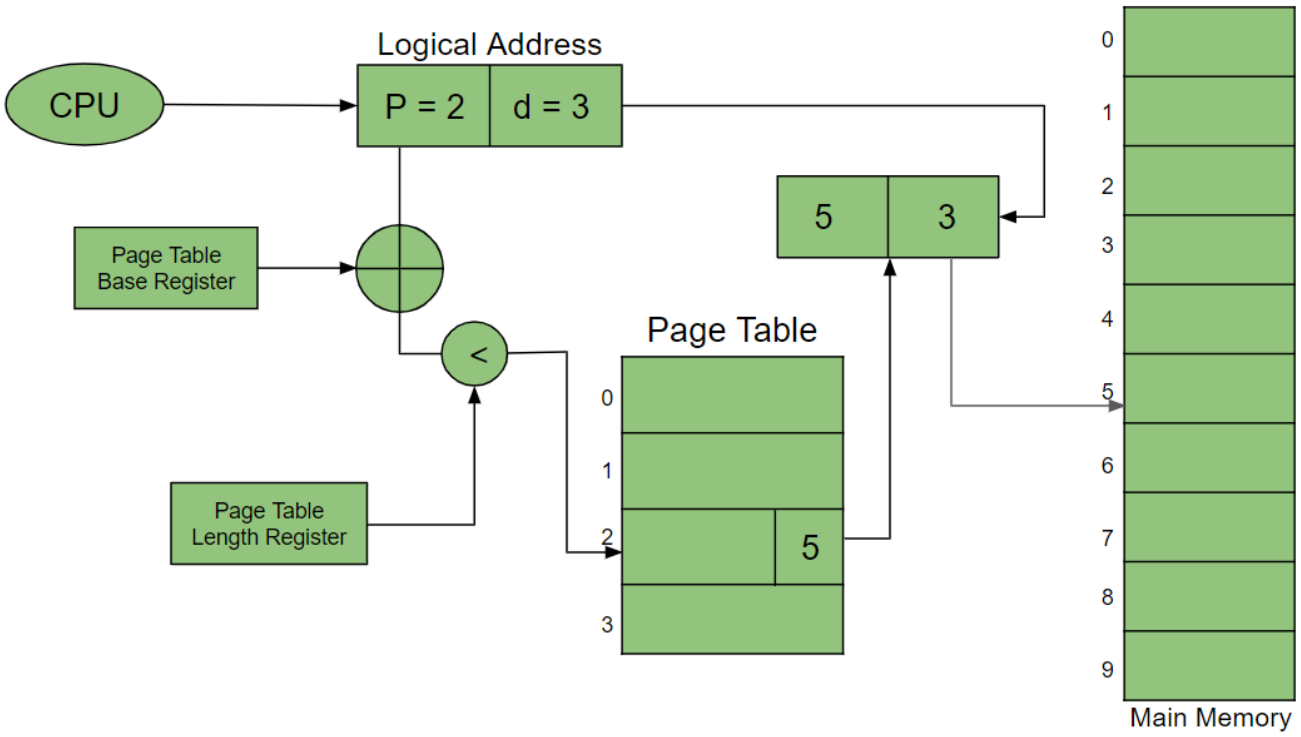
- Physical Address = 12 bits, then Physical Address Space = 4 K words
- Logical Address = 13 bits, then Logical Address Space = 8 K words
- Page size = frame size = 1 K words (assumption)

Let's see how Paging helps in Memory Management through this diagram:



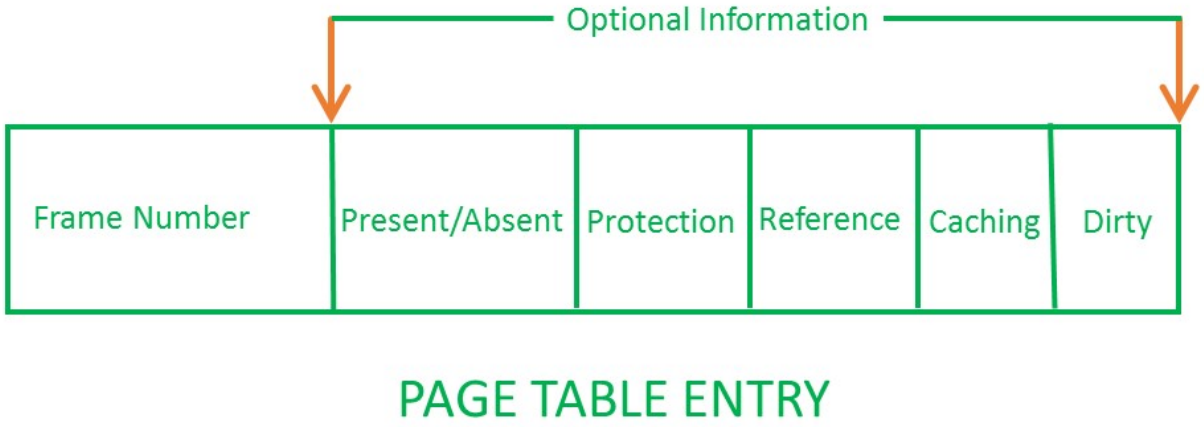
We can see that when P1 completes and goes out of the memory the remaining 4 blocks of memory is taken by P3 in a non-contiguous manner. No more memory is wasted and the problem of fragmentation is almost eliminated.

Runtime binding is used in Paging to facilitate swapping during run time. To do this logical address is needed.
Since the logical addresses are contiguous and we might need a non-contiguous address in paging, these logical addresses are converted to physical addresses. This conversion is quite complex during paging since non-contiguous memory address is available. This brings us to the concept of Page Table, which stores the mapping of logical address spaces to memory frames. So when a process is loaded into the memory, a page table is created for the same. To facilitate the task, the hardware provides us with two registers- Page Table Base Register and Page Table Length Register. These registers help during context switching. The former is used to store the starting address of the created Page Table when context switching occurs and the later is used for security purposes. For every frame in the main memory, we have a corresponding page address in the page table.



This diagram shows the control flow diagram of Paging. As we can see that the logical address stores the page number as 2 and page offset as 3. 2 signifies the page number in the page table. From the page table, we get the frame number to the main memory. Here it is 5 and from the logical address, we get the offset value within frame number. There can be many offsets within a frame. This is how we can access a physical address located inside main memory using CPU generated logical addresses.

There might arise a situation when the address in a page table points to an invalid location of main memory. This is called the Page Fault. A page fault occurs when a program attempts to access data or code that is in its address space but is not currently located in the system RAM. Page table entry has the following information.



1. **Frame Number -** It gives the frame number in which the current page you are looking for is present. The number of bits required depends on the number of frames.Frame bit is also known as address translation bit.

   ```
   Number of bits for frame = Size of physical memory/frame size
   ```

2. **Present/Absent bit -** Present or absent bit says whether a particular page you are looking for is present or absent. In case if it is not present, that is called Page Fault. It is set to 0 if the corresponding page is not in memory. Used to control page fault by the operating system to support virtual memory. Sometimes this bit is also known as **valid/invalid** bits.

3. **Protection bit -** Protection bit says that what kind of protection you want on that page. So, these bit for the protection of the page frame (read, write etc).

4. **Referenced bit -** Referenced bit will say whether this page has been referred in the last clock cycle or not. It is set to 1 by hardware when the page is accessed.

5. **Caching enabled/disabled -** Some times we need the fresh data. Let us say the user is typing some information from the keyboard and your program should run according to the input given by the user. In that case, the information will come into the main memory. Therefore main memory contains the latest information which is typed by the user. Now if you try to put that page in the cache, that cache will show the old information. So whenever freshness is required, we don't want to go for caching or many levels of the memory.The information present in the closest level to the CPU and the information present in the closest level to the user might be different. So we want the information has to be consistency, which means whatever information user has given, CPU should be able to see it as first as possible. That is the reason we want to disable caching. So, this bit **enables or disable** caching of the page.

6. **Modified bit -** Modified bit says whether the page has been modified or not. Modified means sometimes you might try to write something on to the page. If a page is modified, then whenever you should replace that page with some other page, then the modified information should be kept on the hard disk or it has to be written back or it has to be saved back. It is set to 1 by hardware on write-access to page which is used to avoid writing when swapped out. Sometimes this modified bit is also called as the **Dirty bit**.

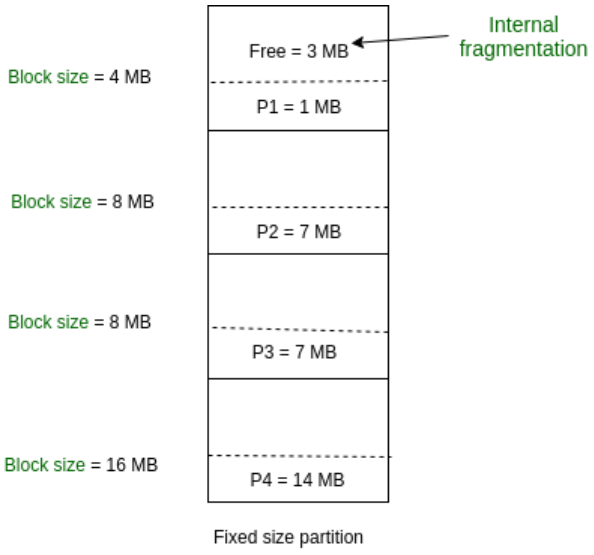
− Memory Management – Fragmentation

In operating systems, Memory Management is the function responsible for allocating and managing a computer's main memory. Memory Management function keeps track of the status of each memory location, either allocated or free to ensure effective and efficient use of Primary Memory.

There are two Memory Management Techniques: **Contiguous**, and **Non-Contiguous**. In Contiguous Technique, executing process must be loaded entirely in main-memory. Contiguous Technique can be divided into:

1. Fixed (or static) partitioning
2. Variable (or dynamic) partitioning

Fixed Partitioning: This is the oldest and simplest technique used to put more than one process in the main memory. In this partitioning, number of partitions (non-overlapping) in RAM are **fixed but size** of each partition may or **may not be same**. As it is **contiguous** allocation, hence no spanning is allowed. Here partitions are made before execution or during system configure.



Fixed size partition

As illustrated in the above figure, the first process is only consuming 1MB out of 4MB in the main memory.
Hence, Internal Fragmentation in first block is (4-1) = 3MB.

Sum of Internal Fragmentation in every block = (4-1)+(8-7)+(8-7)+(16-14)= 3+1+1+2 = 7MB.

Suppose process P5 of size 7MB comes. But this process cannot be accommodated in spite of available free space because of contiguous allocation (as spanning is not allowed). Hence, 7MB becomes part of External Fragmentation.

There are some advantages and disadvantages of fixed partitioning.

Advantages of Fixed Partitioning -

1. **Easy to implement:** Algorithms needed to implement Fixed Partitioning are easy to implement. It simply requires putting a process into certain partition without focussing on the emergence of Internal and External Fragmentation.

2. **Little OS overhead:** Processing of Fixed Partitioning require lesser excess and indirect computational power.

Disadvantages of Fixed Partitioning -

1. **Internal Fragmentation:** Main memory use is inefficient. Any program, no matter how small, occupies an entire partition. This can cause internal fragmentation.

2. **External Fragmentation:** The total unused space (as stated above) of various partitions cannot be used to load the processes even though there is space available but not in the continuous form (as spanning is not allowed).

3. **Limit process size:** The process of size greater than the size of the partition in Main Memory cannot be accommodated. The partition size cannot be varied according to the size of the incoming process's size. Hence, the process size of 32MB in the above-stated example is invalid.

4. **Limitation on Degree of Multiprogramming:** Partition in Main Memory is made before execution or during system configure. Main Memory is divided into a fixed number of the partition. Suppose if there are $n1$ partitions in RAM and $n2$ are the number of processes, then $n2 \leq n1$ condition must be fulfilled. Number of processes greater than the number of partitions in RAM is invalid in Fixed Partitioning.

Fragmentation occurs in a dynamic memory allocation system when many of the free blocks are too small to satisfy any request.
External Fragmentation: External Fragmentation happens when a dynamic memory allocation algorithm allocates some memory and a small piece is left over that cannot be effectively used. If too much external fragmentation occurs, the amount of usable memory is drastically reduced. Total memory space exists to satisfy a request, but it is not contiguous.

Internal Fragmentation: Internal fragmentation is the space wasted inside of allocated memory blocks because of restriction on the allowed sizes of allocated blocks. Allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.

- Virtual Memory Management - Description of virtual memory

Virtual Memory is a storage allocation scheme in which secondary memory can be addressed as though it were part of the main memory. The addresses a program may use to reference memory are distinguished from the addresses the memory system uses to identify physical storage sites, and program generated addresses are translated automatically to the corresponding machine addresses.
The size of virtual storage is limited by the addressing scheme of the computer system and the amount of secondary memory is available not by the actual number of the main storage locations.

It is a technique that is implemented using both hardware and software. It maps memory addresses used by a program, called virtual addresses, into physical addresses in computer memory.

1. All memory references within a process are logical addresses that are dynamically translated into physical addresses at run time. This means that a process can be swapped in and out of main memory such that it occupies different places in main memory at different times during the course of execution.
2. A process may be broken into a number of pieces and these pieces need not be continuously located in the main memory during execution. The combination of dynamic run-time address translation and use of page or segment table permits this.

If these characteristics are present then, it is not necessary that all the pages or segments are present in the main memory during execution. This means that the required pages need to be loaded into memory whenever required. Virtual memory is implemented using Demand Paging or Demand Segmentation.

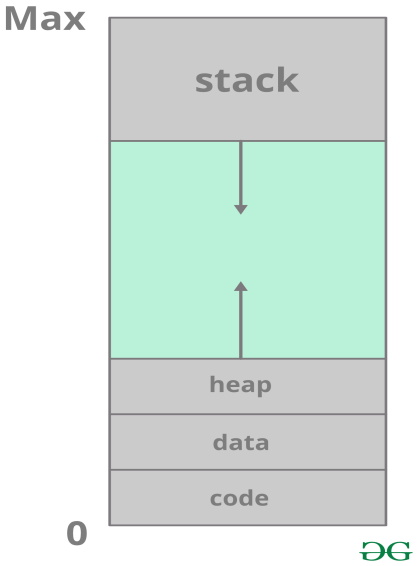
In main memory management it was discussed how to avoid memory fragmentation by breaking process memory requirements down into smaller bites (pages), and storing the pages non-contiguously in memory. But the entire process still had to be stored in memory somewhere.

- In practice, most real processes do not need all their pages, or at least not all at once, for several reasons:
    1. There is no need of Error handling code unless that specific error occurs, some of which are quite rare.
    2. Arrays are often over-sized for worst-case scenarios, and only a small fraction of the arrays are actually used in practice.
    3. Certain features of certain programs are rarely used, such as the routine to balance the federal budget.
- There are some benefits of the ability to load only the portions of processes that were actually needed (and only when they were needed):
    - Programs could be written for much larger address space (virtual memory space) than physically exists on the computer.
    - Since every process is only using a fraction of their total address space that's why there is more memory left for other programs this will improvise CPU utilization and system throughput.
    - For swapping processes less I/O is needed in and out of RAM this will speed things up.

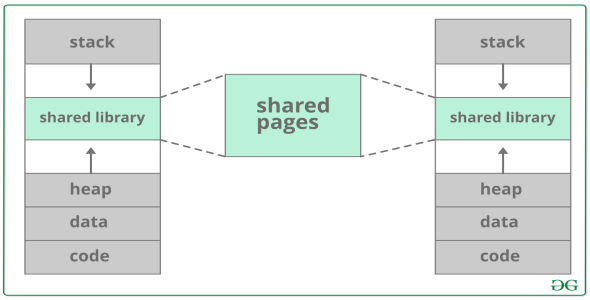
  Let's try to understand the impact of Page Fault:
  Suppose we have a 99% hit ratio of the RAM and the Ram access time is 10 ns. During page fault time taken is 5000000 ns. So let's find out the average access time.
  The average access time for the above process is = 0.99 x 10 ns + 0.01 x 5000000 ns = 50,009.9 ns which is approximately equal to 50 micro-second.
  So, we see how a 1% page fault can increase the ram access time from 10 ns to 50 micro-second.

  Figure 1 given below shows the general layout of virtual memory, which can be much larger than the physical memory:

- Figure 2 given below shows virtual address space, which is the programmer's logical view of process memory storage. The actual physical layout is controlled by the process's page table.
- Note that the address space shown in Figure 9.2 is sparse - A great hole in the middle of the address space is never used unless the stack and/or the heap grow to fill the hole.



- By multiple processes, virtual memory also allows the sharing of files and memory, with some benefits:
    - If System libraries are mapped with virtual address space of more than one process these libraries can be shared.
    - Virtual memory can also be shared with processes if it is mapped with the same block of memory to more than one process.
    - During a fork( ) system call process pages can also be shared, eliminating the need to copy all of the pages of the original (parent)process.



Memory Management – TLB

In Operating System (Memory Management Technique: Paging), for each process page table will be created, which will contain Page Table Entry (PTE). This PTE will contain information like frame number (The address of main memory where we want to refer), and some other useful bits (e.g., valid/invalid bit, dirty bit, protection bit, etc). This page table entry (PTE) will tell where in the main memory the actual page is residing.

Now the question is where to place the page table, such that overall access time (or reference time) will be less.

The problem initially was to fast access the main memory content based on an address generated by CPU (i.e logical/virtual address). Initially, some people thought of using registers to store page table, as they are high-speed memory so access time will be less.

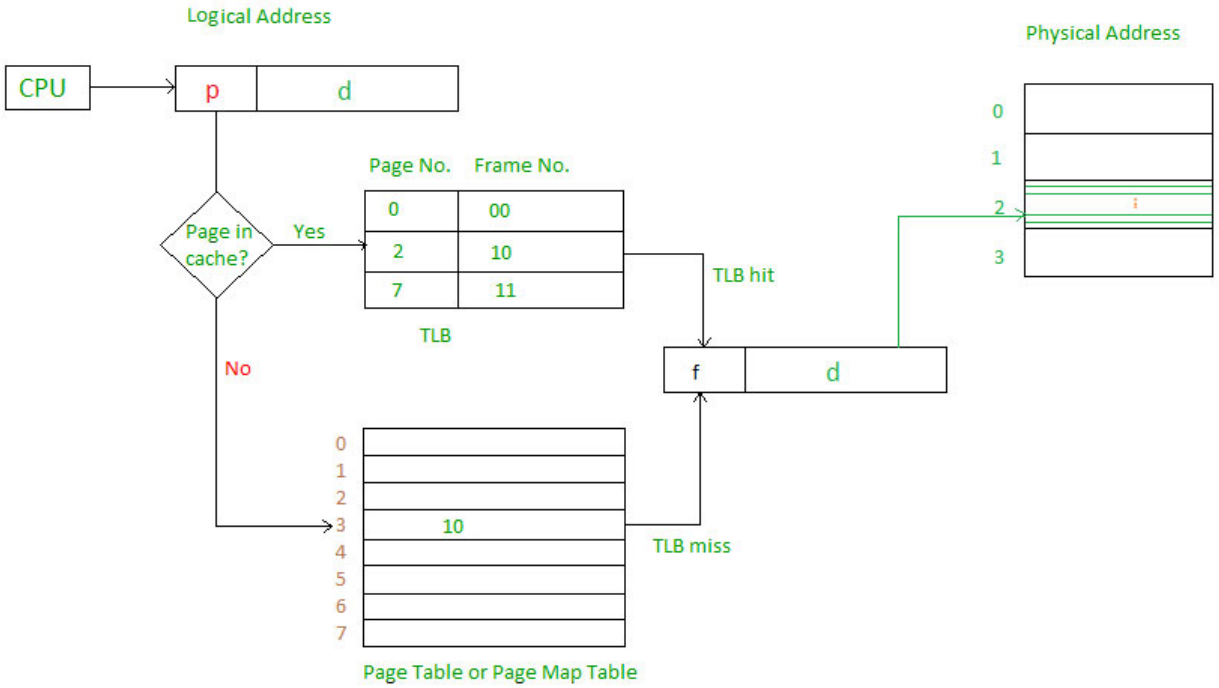
The idea used here is, place the page table entries in registers, for each request generated from CPU (virtual address), it will be matched to the appropriate page number of the page table, which will now tell where in the main memory that corresponding page resides. Everything seems right here, but the problem lies in the small register size (in practical, it can accommodate maximum of 0.5k to 1k page table entries) and process size may be big hence the required page table will also be big (lets say this page table contains 1M entries), so registers may not hold all the PTE's of Page table. So this is not a practical approach.

To overcome this size issue, the entire page table was kept in the main memory. but the problem here is two main memory references are required:

```
1. To find the frame number.

2. To go to the address specified by frame number.
```

To overcome this problem a high-speed cache is set up for page table entries called a Translation Lookaside Buffer (TLB). Translation Lookaside Buffer (TLB) is nothing but a special cache used to keep track of recently used transactions. TLB

contains page table entries that have been most recently used. Given a virtual address, the processor examines the TLB if a page table entry is present (TLB hit), the frame number is retrieved and the real address is formed. If a page table entry is not found in the TLB (TLB miss), the page number is used to index the process page table. TLB first checks if the page is already in main memory, if not in main memory a page fault is issued then the TLB is updated to include the new page entry.



Steps in TLB hit:

1. CPU generates virtual address.
2. It is checked in TLB (present).
3. Corresponding frame number is retrieved, which now tells where in the main memory page lies.

Steps in Page miss:

1. CPU generates virtual address.
2. It is checked in TLB (not present).
3. Now the page number is matched to page table residing in main memory (assuming page table contains all PTE).
4. Corresponding frame number is retrieved, which now tells where in the main memory page lies.
5. The TLB is updated with new PTE (if space is not there, one of the replacement technique comes into picture i.e either FIFO, LRU or MFU etc).

Effective memory acess time(EMAT) : TLB is used to reduce effective memory access time as it is a high speed associative cache.

EMAT = h*(c+m) + (1-h)*(c+2m)

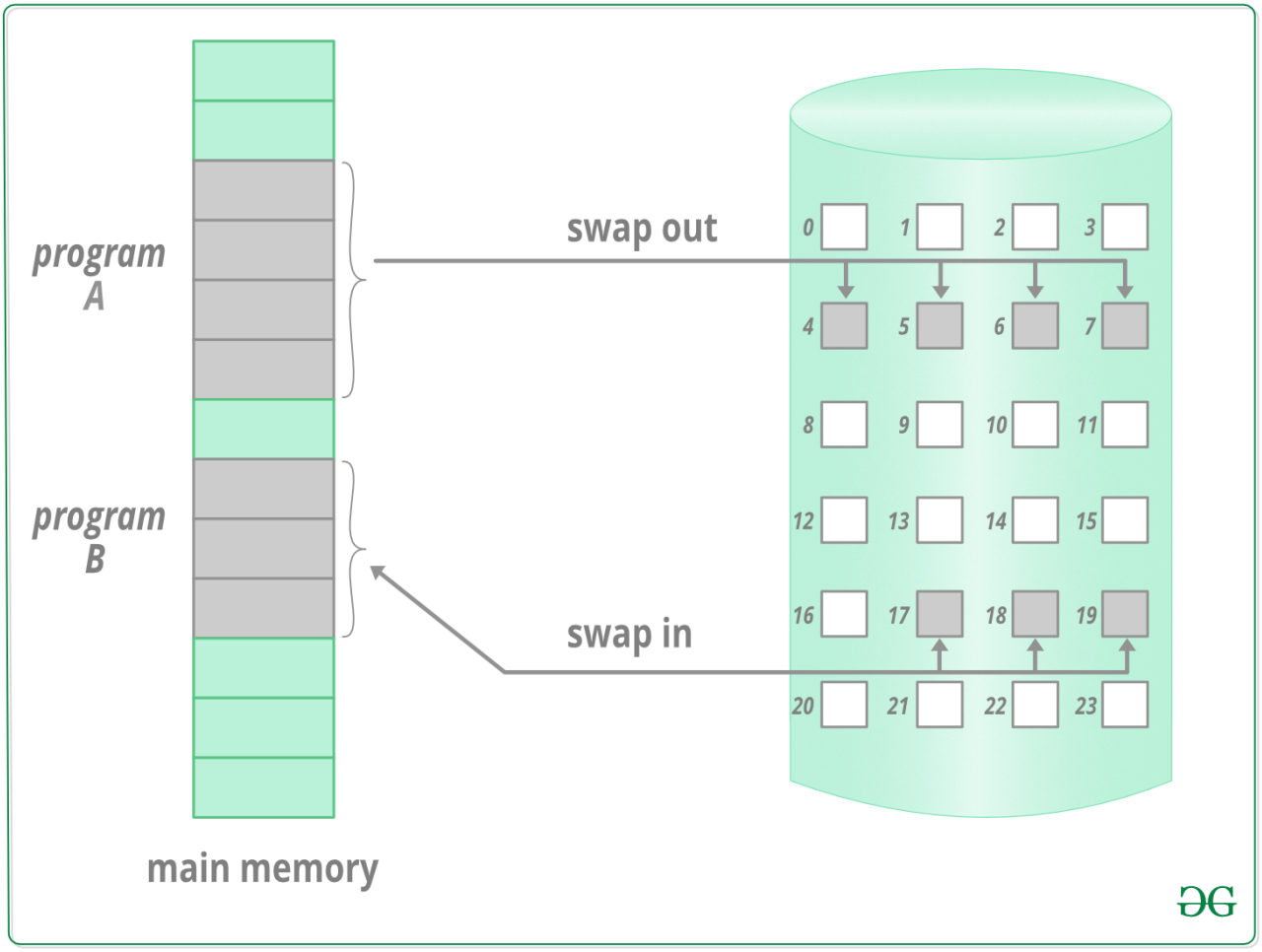
where, h = hit ratio of TLB

m = Memory access time

c = TLB access time

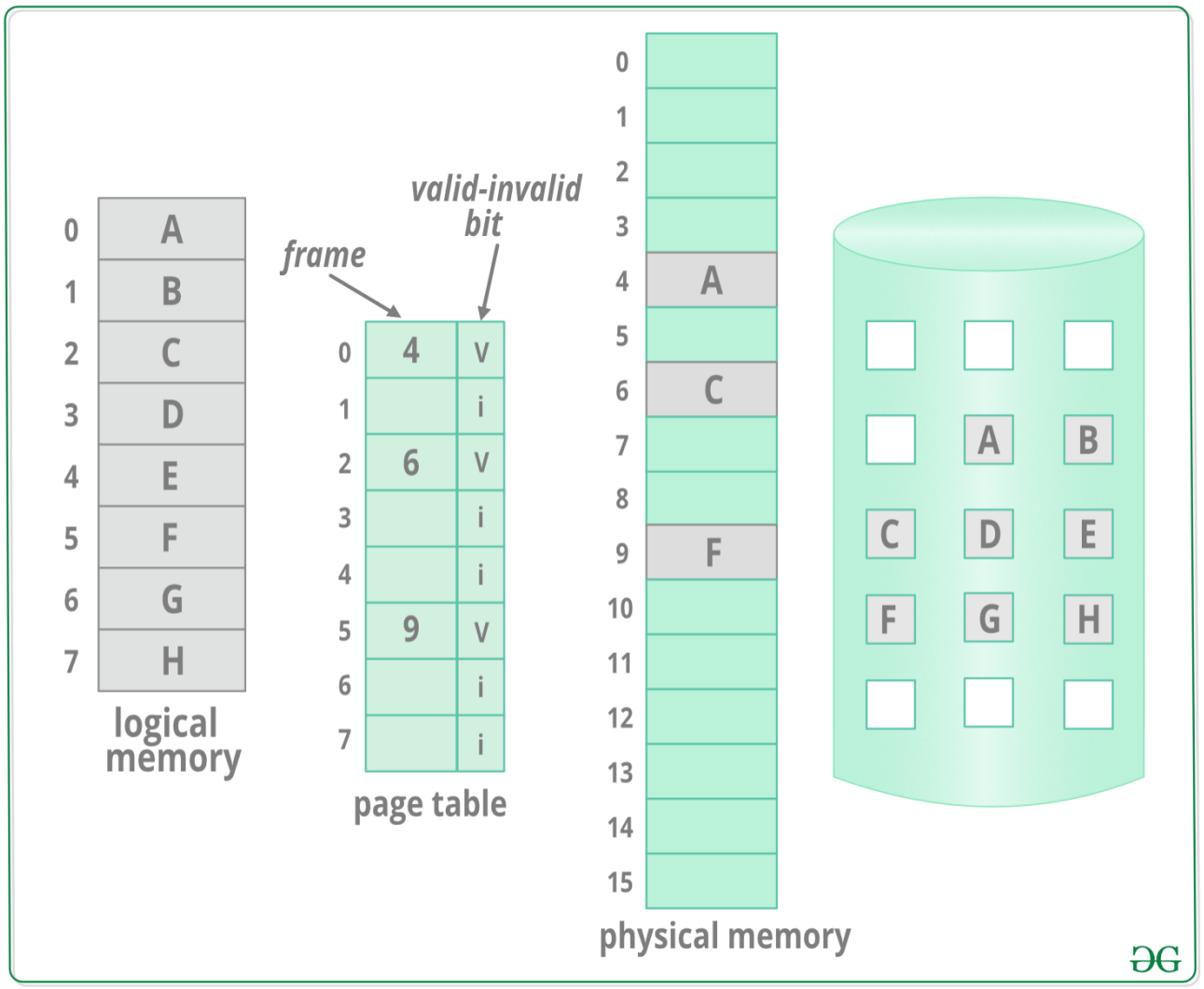
- Virtual Memory Management - Demand Paging

When a process is swapped in, all its pages are not swapped in at once. Rather they are swapped in only when the process demands them or needs them this is the basic idea behind demand paging. This is also termed as a lazy swapper.



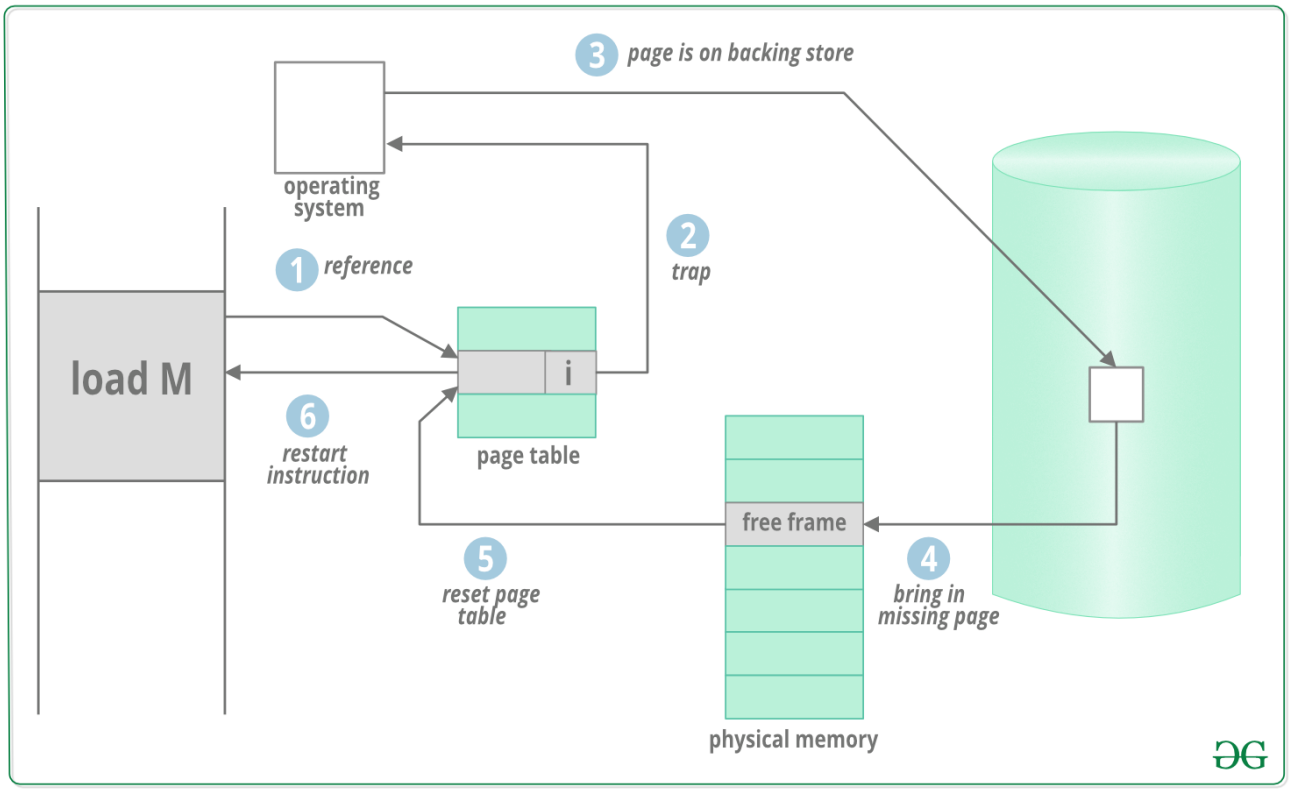
Basic Concept:

- The basic idea behind paging is that when a process is swapped in, the pager only loads into memory those pages that it expects the process to need (right away.)
- Those pages which are not loaded into memory are marked as invalid pages in the page table using the invalid bit. The remaining page table entry may either be blank or have some information about the swapped-out page on the hard drive.
- If the process only ever access pages that are loaded in memory (memory resident pages), then the process runs exactly as if all the pages were loaded into memory.



On the other hand, if a page is needed that was not originally loaded up, then a page fault trap is generated, which must be handled in a series of steps:

1. The memory address requested is first checked, to make sure it was a valid memory request.
2. A process is terminated if the reference was invalid. Else, the page must be paged in.
3. A free frame is located from a free-frame list.
4. To bring the necessary page from a disk operation is scheduled. It will allow some other process to use the CPU and block the process on an I/O wait.
5. The process's page table is updated when the I/O operation is completed, with the new frame number, and the invalid bit is changed its status to valid indicating that this is now a valid page reference.
6. The instruction will be restarted that caused the page fault.(as soon as this process gets another turn on the CPU.)



- NO pages are swapped in for a process until they are requested by page faults in the extreme case, This is termed as pure demand paging.
- Theoratically each instruction could generate multiple page faults. In practice, this is very rare, due to the locality of reference.
- A page table and secondary is necessary to support virtual memory and it is the same as for paging and swapping.
- Once the demanded page is available in memory, the instruction must be restarted from scratch(a crucial part of the demand paging). For the basic instructions, it is not a difficulty. But in some architectures, a large block of data is modified by a single instruction. And if some of the data get modified before the page fault occurs, this could cause some problems. One solution is to access both ends of the block before executing the instruction so that the necessary pages already paged before the instruction begins.

Performance of Demand Paging

- Obviously there is some slowdown and performance hit whenever a page fault occurs and the system has to go get it from memory, but just how big a hit is it exactly?
- A number of steps take place while servicing a page fault (see the book for full details), and some of the steps are optional or variable. But just for the sake of discussion, suppose that normal memory access requires 300 nanoseconds and that servicing a page fault takes 8 milliseconds. (9,000,000 nanoseconds, or 30,000 times a normal memory access.) Where p is the page fault rate, (ranges from 0 to 1), the effective access time is now:

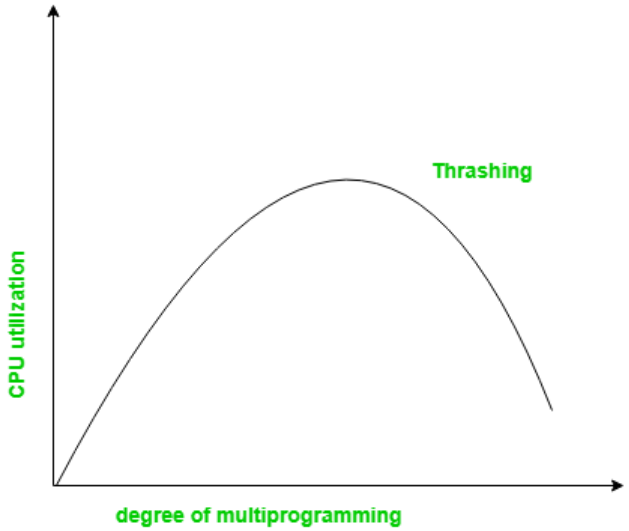
```
= ( 1 - p ) * ( 300 ) + p * 9000000

= 300 + 8,999,700 * p
```

- which clearly depends heavily on p! Even if only one access in 1000 causes a page fault, the effective access time drops from 200 nanoseconds to 8.2 microseconds, a slowdown of a factor of 40 times. In order to keep the slowdown less than 10%, the page fault rate must be less than 0.0000025, or one in 399,990 accesses.

- A subtlety is that swap space is faster to access than the regular file system because it does not have to go through the whole directory structure. For this reason, some systems will transfer an entire process from the file system to swap space before starting up the process so that future paging all occurs from the (relatively) faster swap space.
- Some systems use demand paging directly from the file system for binary code (which never changes and hence does not have to be stored on a page operation), and to reserve the swap space for data segments that must be stored. Both Solaris and BSD Unix use this approach.

- Virtual Memory Management - Thrashing

Thrashing is a condition or a situation when the system is spending a major portion of its time in servicing the page faults, but the actual processing done is very 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 useful 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. 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.

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. A program is generally composed of several different localities that may overlap.

For example, when a function is called, it defines a new locality where memory references are made to the instructions of the function call, it's local and global variables, etc. Similarly, when the function is exited, the process leaves this locality.

Techniques to handle:

1. **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 a 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 dependant 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 D is the total demand for frames and WSS_i is the working set size for a process I,

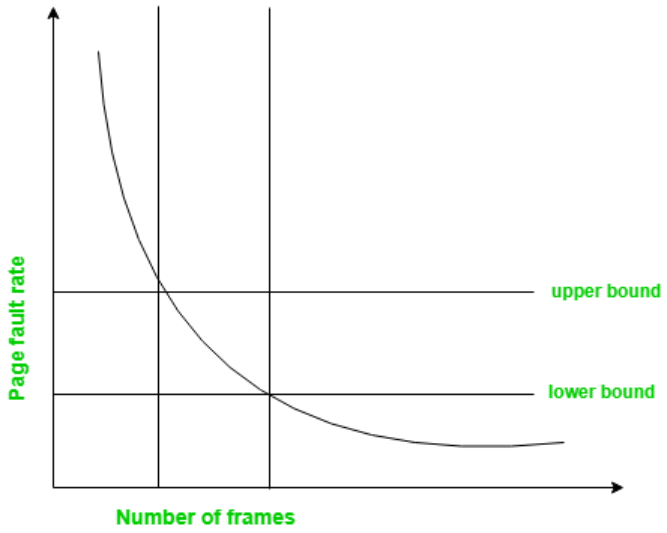
   $$D = \sum no \lim its\{WSS_i\}$$

   Now, if 'm' is the number of frames available in the memory, there are 2 possibilities:
   - (i) D>m i.e. total demand exceeds the number of frames, then thrashing will occur as some processes would not get enough frames.
   - (ii) D<=m, then there would be no thrashing.

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

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

2. 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 fault rate exceeds the upper limit, the number of 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.

Here too, if the page fault rate is high with no free frames, then some of the processes can be suspended and frames allocated to them can be reallocated to other processes. The suspended processes can then be restarted later.

Causes of Thrashing :

1. High degree of multiprogramming : If the number of processes keeps on increasing in the memory then the number of frames allocated to each process will be decreased. So, less number of frames will be available to each process. Due to this, a page fault will occur more frequently and more CPU time will be wasted in just swapping in and out of pages and the utilization will keep on decreasing.

   For example:
   Let free frames = 400
   Case 1: Number of process = 100
   Then, each process will get 4 frames.

   Case 2: Number of process = 400
   Each process will get 1 frame.
   Case 2 is a condition of thrashing, as the number of processes is increased, frames per process are decreased.
   Hence CPU time will be consumed in just swapping pages.

2. **Lacks Frames**: If a process has less number of frames then fewer pages of that process will be able to reside in memory and hence more frequent swapping in and out will be required. This may lead to thrashing. Hence sufficient amount of frames must be allocated to each process in order to prevent thrashing.

Recovery of Thrashing :

- Do not allow the system to go into thrashing by instructing the long term scheduler not to bring the processes into memory after the threshold.
- If the system is already in thrashing then instruct the midterm scheduler to suspend some of the processes so that we can recover the system from thrashing.

- Virtual Memory Management - Page Replacement

In an operating system that uses paging for memory management, a page replacement algorithm is needed to decide which page needs to be replaced when a new page comes in.

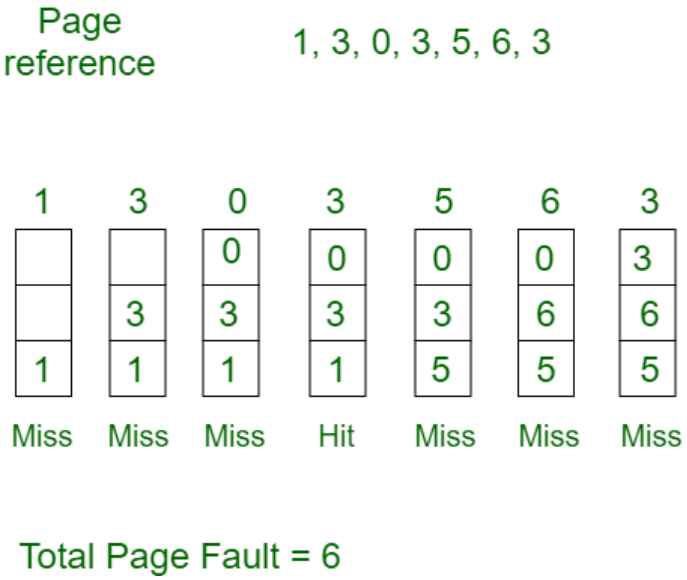
Page Fault - A page fault happens when a running program accesses a memory page that is mapped into the virtual address space but not loaded in physical memory.

Since actual physical memory is much smaller than virtual memory, page faults happen. In case of a page fault, the Operating System might have to replace one of the existing pages with the newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce the number of page faults.

Page Replacement Algorithms :

- **First In First Out (FIFO) -** This is the simplest page replacement algorithm. In this algorithm, the operating system keeps track of all pages in the memory in a queue, the oldest page is in the front of the queue. When a page needs to be replaced the page in the front of the queue is selected for removal.

  Example-1 Consider page reference string 1, 3, 0, 3, 5, 6 with 3 page frames.Find number of page faults.



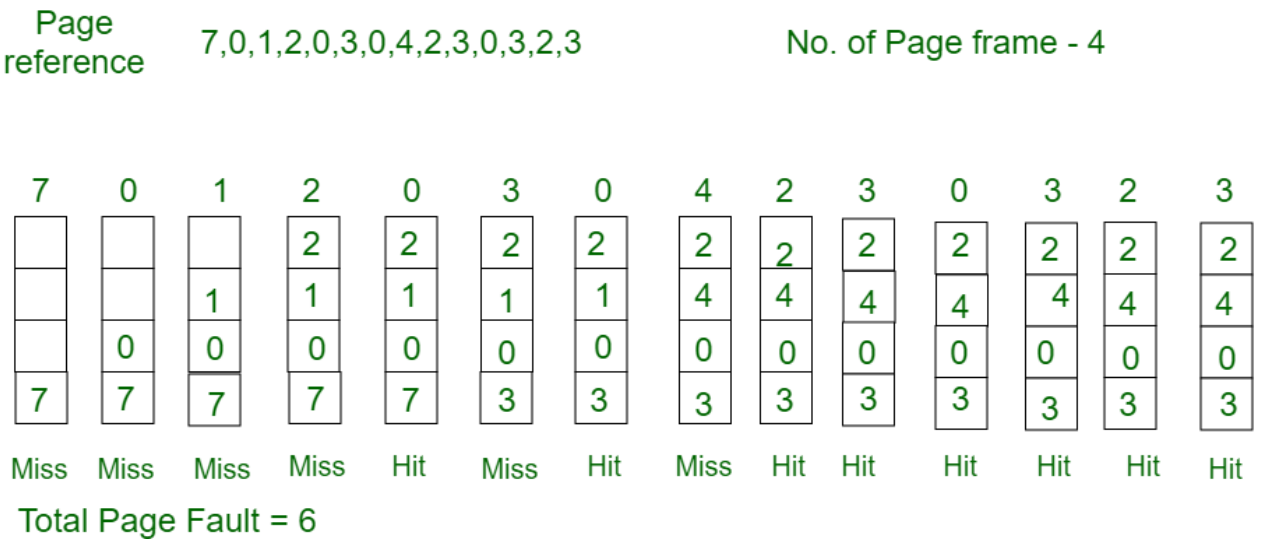
Initially all slots are empty, so when 1, 3, 0 came they are allocated to the empty slots → **3 Page Faults.** when 3 comes, it is already in memory so → **0 Page Faults.** Then 5 comes, it is not available in memory so it replaces the oldest page slot i.e 1. →**1 Page Fault.** 6 comes, it is also not available in memory so it replaces the oldest page slot i.e 3 →**1 Page Fault.** Finally when 3 come it is not avilable so it replaces 0 **1 page fault**

Belady's anomaly - Belady's anomaly proves that it is possible to have more page faults when increasing the number

of page frames while using the First in First Out (FIFO) page replacement algorithm. For example, if we consider reference string 3, 2, 1, 0, 3, 2, 4, 3, 2, 1, 0, 4 and 3 slots, we get 9 total page faults, but if we increase slots to 4, we get 10 page faults.

- Optimal Page replacement - In this algorithm, pages are replaced which would not be used for the longest duration of time in the future.

  Example-2:Consider the page references 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, with 4 page frame. Find number of page fault.

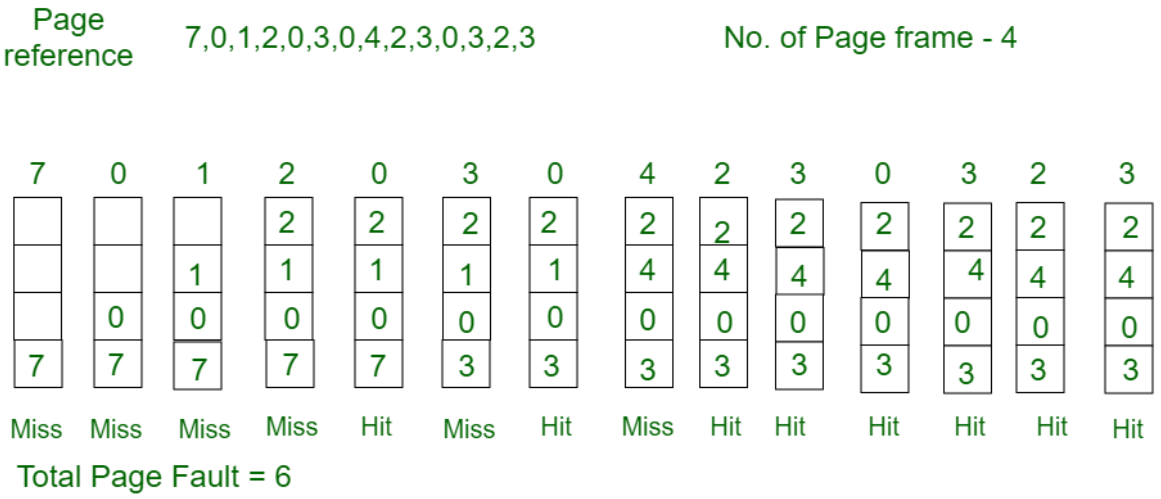


Initially all slots are empty, so when 7 0 1 2 are allocated to the empty slots → **4 Page faults** 0 is already there so → **0 Page fault.** when 3 came it will take the place of 7 because it is not used for the longest duration of time in the future. →**1 Page fault.** 0 is already there so → **0 Page fault.**.
4 will takes place of 1 → **1 Page Fault.**
Now for the further page reference string → **0 Page fault** because they are already available in the memory.

Optimal page replacement is perfect, but not possible in practice as the operating system cannot know future requests. The use of Optimal Page replacement is to set up a benchmark so that other replacement algorithms can be analyzed against it.

- Least Recently Used - In this algorithm page will be replaced which is least recently used.

  Example-3Consider the page reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2 with 4 page frames.Find number of page faults.



Here LRU has same number of page fault as optimal but it may differ according to question.

Initially all slots are empty, so when 7 0 1 2 are allocated to the empty slots → **4 Page faults** 0 is already their so → **0 Page fault.** when 3 came it will take the place of 7 because it is least recently used →**1 Page fault** 0 is already in memory so → **0 Page fault**.
4 will takes place of 1 → **1 Page Fault** Now for the further page reference string → **0 Page fault** because they are already available in the memory.

Note: You can also refer LFU and Second chance page replacement policy.

- Memory Management - Segmentation

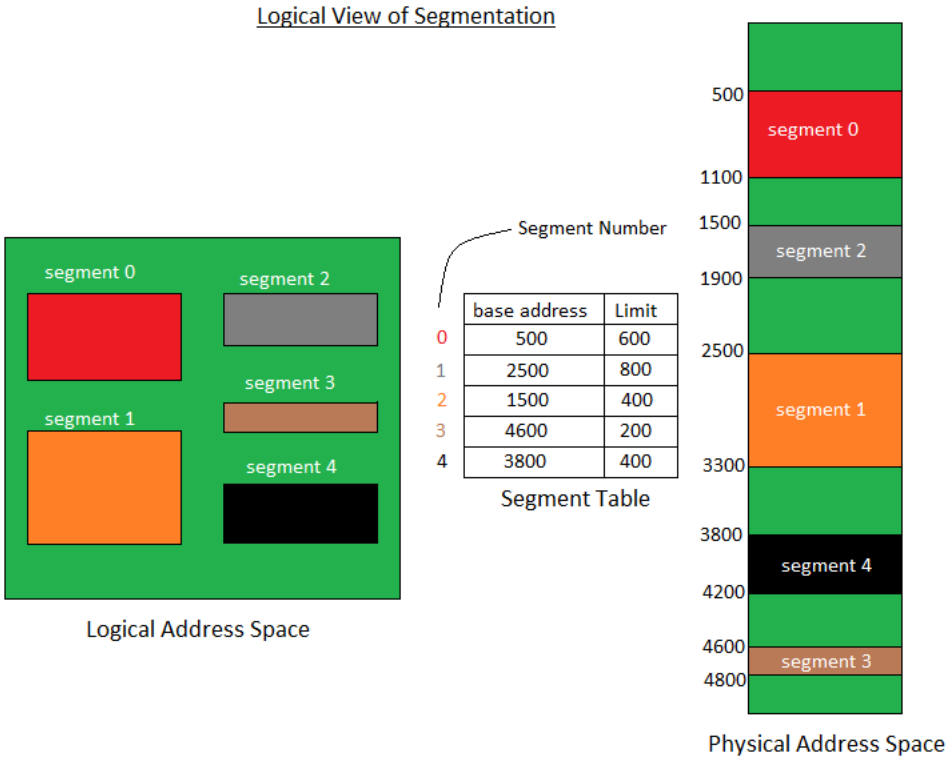
A process is divided into Segments. The chunks that a program is divided into which are not necessarily all of the same sizes are called segments. Segmentation gives the user's view of the process that paging does not give. Here the user's view is mapped to physical memory.
There are types of segmentation:

1. **Virtual memory segmentation -** Each process is divided into a number of segments, not all of which are resident at any one point in time.
2. **Simple segmentation -** Each process is divided into a number of segments, all of which are loaded into memory at run time, though not necessarily contiguously.

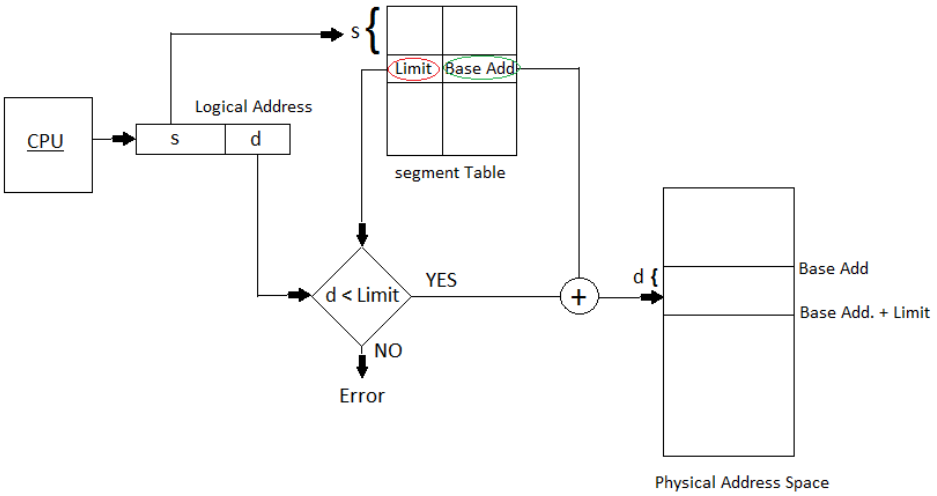
There is no simple relationship between logical addresses and physical addresses in segmentation. A table stores the information about all such segments and is called Segment Table.

Segment Table - It maps two-dimensional Logical address into one-dimensional Physical address. It's each table entry has:

- **Base Address:** It contains the starting physical address where the segments reside in memory.
- **Limit:** It specifies the length of the segment.



Translation of Two-dimensional Logical Address to one-dimensional Physical Address.



The address generated by the CPU is divided into:

- **Segment number (s):** Number of bits required to represent the segment.
- **Segment offset (d):** Number of bits required to represent the size of the segment.

Advantages of Segmentation -

- No Internal fragmentation.
- Segment Table consumes less space in comparison to Page table in paging.

Disadvantage of Segmentation -

- As processes are loaded and removed from the memory, the free memory space is broken into little pieces, causing External fragmentation.

Memory Management - Swapping

A computer has a sufficient amount of physical memory but most of the time we need more so we swap some memory on disk. Swap space is a space on the hard disk which is a substitute of physical memory. It is used as a virtual memory that contains the process memory image. Whenever our computer runs short of physical memory it uses it's virtual memory and stores information in memory on disk. Swap space helps the computer's operating system in pretending that it has more RAM than it actually has. It is also called a swap file. This interchange of data between virtual memory and real memory is called as swapping and space on disk as "swap space".

Virtual memory is a combination of RAM and disk space that running processes can use. Swap space is the portion of virtual memory that is on the hard disk, used when RAM is full.

Swap space can be useful to the computer in various ways:

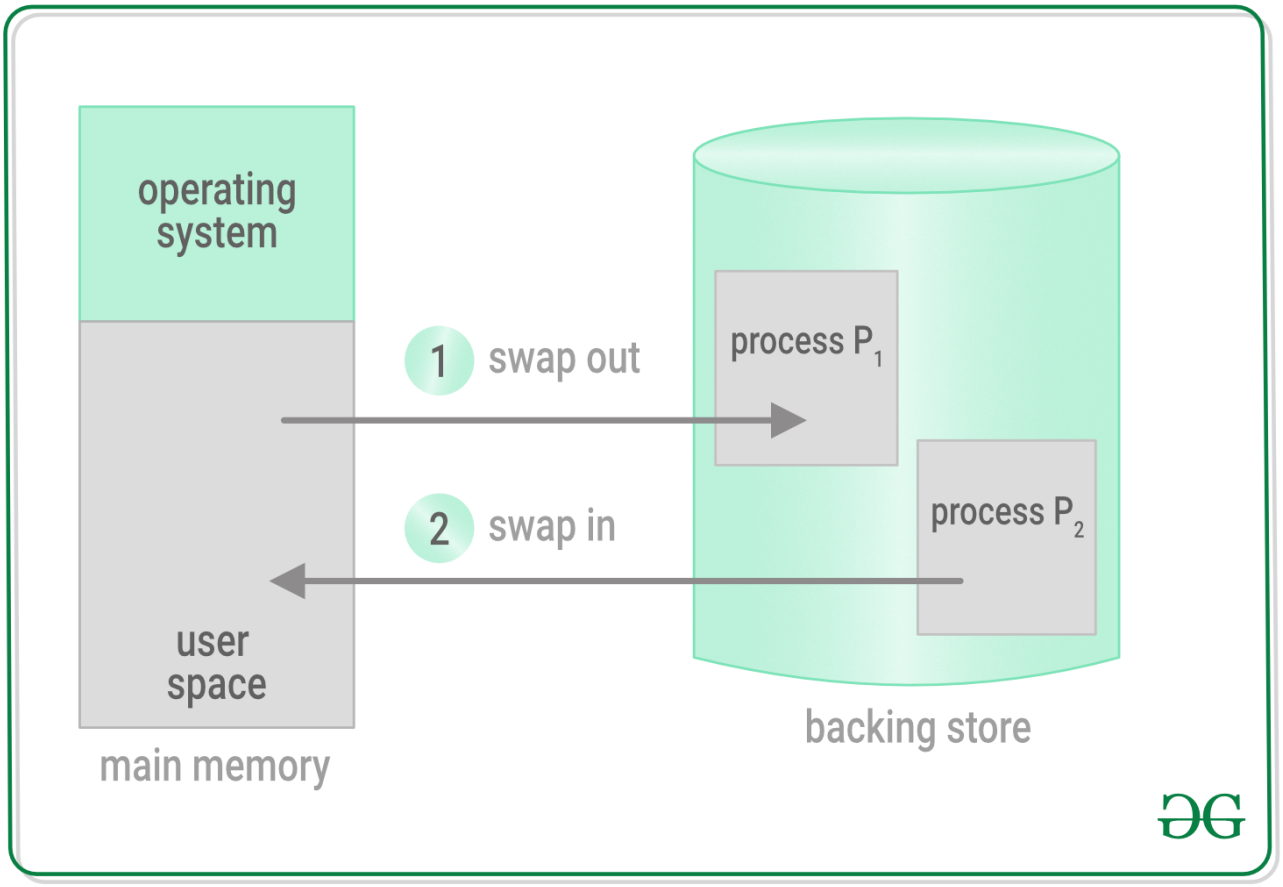
1. It can be used as a single contiguous memory which reduces i/o operations to read or write a file.
2. Applications that are not used or are used less can be kept in the swap file.
3. Having sufficient swap file helps the system keep some physical memory free all the time.
4. The space in physical memory which has been freed due to swap space can be used by OS for some other important tasks.

In operating systems such as Windows, Linux, etc systems provide a certain amount of swap space by default which can be changed by users according to their needs. If you don't want to use virtual memory you can easily disable it all together but in case if you run out of memory then the kernel will kill some of the processes in order to create a sufficient amount of space in physical memory. So it totally depends upon the user whether he wants to use swap space or not.

What is standard swapping?

- If compile-time or load-time address binding is used, the processes must be swapped back into the same memory location from which they were swapped out. The processes can be swapped back into any available location if execution time binding is used.
- Comparing with other processes in a system swapping is a very slow process. Swapping involves moving old data out as well as new data in, for efficient processor scheduling, the CPU time slice should be significantly longer than this lost transfer time.
- Swapping transfer overhead can be reduced by applying limitation over information to be transferred, which requires that the system know how much memory a process is using, as opposed to how much it might use. If programmers freeing up dynamic memory that is no longer in use, it will be helpful.
- When processes are idle(No I/O operation are pending), it can be swapped out. There are two options either swap only totally idle processes or perform pending I/O operations only into and out of OS buffers, which are then transferred to or from the process's main memory as a second step.
- Most modern OSes no longer use swapping, because it is too slow and there are faster alternatives available. (e.g. Paging.) But if the system gets extremely full, some UNIX systems will still invoke swapping, and then discontinue swapping when the load reduces again.

Figure:

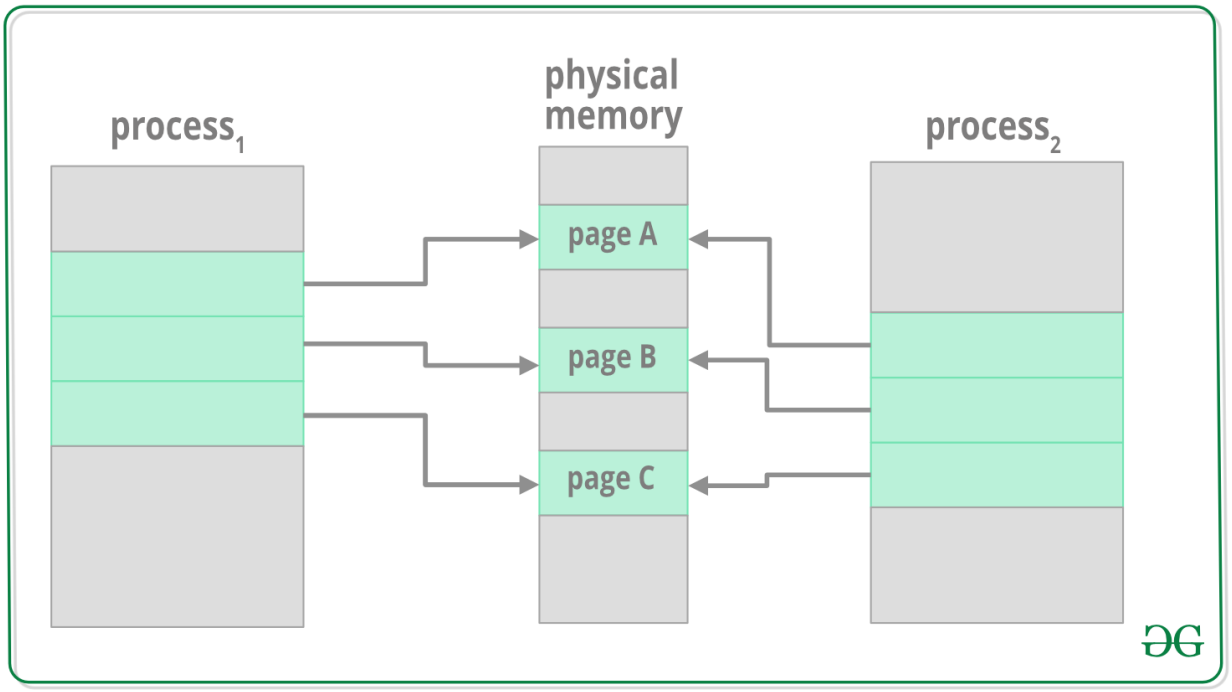


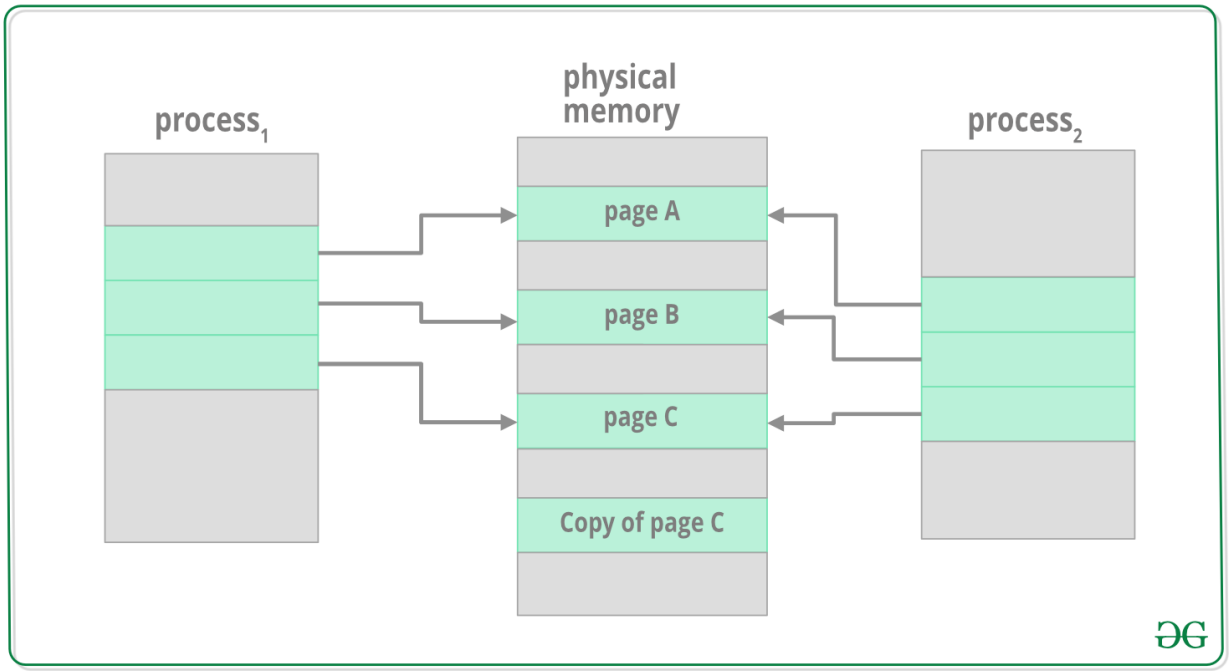
How swapping occurs on the mobile system?

- There are several reasons why swapping is not supported on mobile platforms:
  - On Mobile devices there is typically flash memory in place of more spacious hard drives, so there is not as much space available.
  - Flash memory has a limit to be written to a limited number of times before it becomes unreliable.
  - There is low bandwidth to flash memory.
- Apple's IOS asks applications to voluntarily free up memory
  - Read-only data.
  - Modified data.
  - OS must uninstall the apps which are failed to free up memeory space
- Android follows a similar strategy.
  - In android, it write application state to flash memory for quick restarting, instead to terminate a process

- Virtual Memory Management - Copy-on -write

- The idea behind a copy-on-write fork is that the pages for a parent process do not have to be actually copied for the child until one or the other of the processes changes the page. They can be simply shared between the two processes in the meantime, with a bit set that the page needs to be copied if it ever gets written to. This is a reasonable approach since the child process usually issues an exec( ) system call immediately after the fork.





- Pages that can be modified also need to be labeled as copy-on-write. Code segments can simply be shared.
- Pages used to satisfy copy-on-write duplications are typically allocated using zero-fill-on-demand, meaning that their previous contents are zeroed out before the copy proceeds.
- An alternative to the fork() system call is provided by some systems is called a virtual memory fork, vfork( ). In this case, the parent is suspended, and the child uses the parent's memory pages. This is very fast for process creation but requires that the child not modify any of the shared memory pages before performing the exec( ) system call. (In essence, this addresses the question of which process executes first after a call to fork, the parent or the child. With vfork, the parent is suspended, allowing the child to execute first until it calls exec( ), sharing pages with the parent in the meantime.

- Multi-threaded programming - Threads and its types

Thread is a single sequence stream within a process. Threads have same properties as of the process so they are called as light weight processes. Threads are executed one after another but gives the illusion as if they are executing in parallel. Each thread has different states. Each thread has

1. A program counter
2. A register set
3. A stack space

Threads are not independent of each other as they share the code, data, OS resources, etc.

Similarity between Threads and Processes -

- Only one thread or process is active at a time
- Within process both execute sequentiall
- Both can create children

Differences between Threads and Processes -

- Threads are not independent, processes are.
- Threads are designed to assist each other, processes may or may not do it

Types of Threads:

1. User Level thread (ULT) - It is implemented in the user level library, they are not created using the system calls. Thread switching does not need to call OS and to cause an interrupt to Kernel. The kernel doesn't know about the user level thread and manages them as if they were single-threaded processes.

   Advantages of ULT -
   - Can be implemented on an OS that does't support multithreading.
   - Simple representation since thread has only program counter, register set, stack space.
   - Simple to create since no intervention of kernel.
   - Thread switching is fast since no OS calls need to be made.

   Disadvantages of ULT -
   - No or less co-ordination among the threads and Kernel.
   - If one thread causes a page fault, the entire process blocks.

2. Kernel Level Thread (KLT) - Kernel knows and manages the threads. Instead of thread table in each process, the kernel itself has thread table (a master one) that keeps track of all the threads in the system. In addition kernel also maintains the traditional process table to keep track of the processes. OS kernel provides system call to create and manage threads.

   Advantages of KLT -
   - Since kernel has full knowledge about the threads in the system, scheduler may decide to give more time to processes having large number of threads.
   - Good for applications that frequently block.

   Disadvantages of KLT -
   - Slow and inefficient.
   - It requires thread control block so it is an overhead.

Summary:

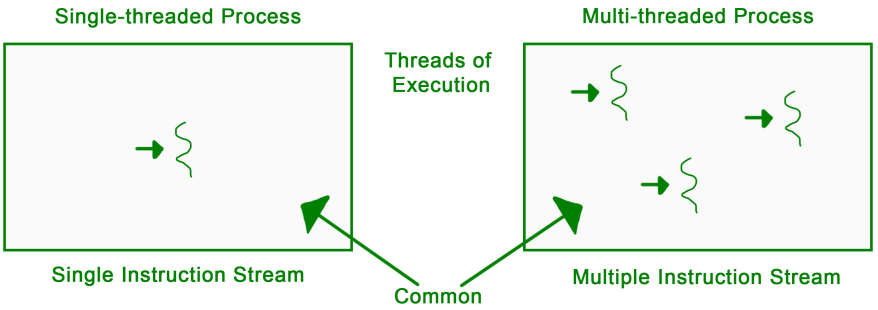
1. Each ULT has a process that keeps track of the thread using the Thread table.
2. Each KLT has Thread Table (TCB) as well as the Process Table (PCB).

A **thread** is a path which is followed during a program's execution. Majority of programs written nowadays run as a single thread. Let's say, for example, a program is not capable of reading keystrokes while making drawings. These tasks cannot be executed by the program at the same time. This problem can be solved through multitasking so that two or more tasks can be executed simultaneously.

Multitasking is of two types: Processor based and thread based. Processor-based multitasking is totally managed by the OS, however multitasking through multithreading can be controlled by the programmer to some extent.

The concept of **multi-threading** needs proper understanding of these two terms - **a process and a thread**. A process is a program being executed. A process can be further divided into independent units known as threads.

A thread is like a small light-weight process within a process. Or we can say a collection of threads is what is known as a process.



Single Thread and Multi Thread Process

Applications -

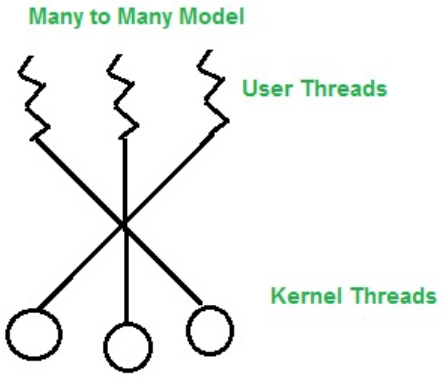
- Threading is used widely in almost every field. Most widely it is seen over the internet nowadays where we are using transaction processing of every type like recharges, online transfer, banking, etc. Threading is a segment that divides the code into small parts that are of very lightweight and has less burden on CPU memory so that it can be easily worked out and can achieve the goal in the desired field.
- The concept of threading is designed due to the problem of fast and regular changes in technology and less work in different areas due to less application. Then as says "need is the generation of creation or innovation" hence by following this approach human mind develop the concept of thread to enhance the capability of programming.

Many operating systems support kernel thread and user thread in a combined way. An example of such a system is Solaris. The multithreading model is of three types.

```
Many to many models.

Many to one model.

one to one model.
```

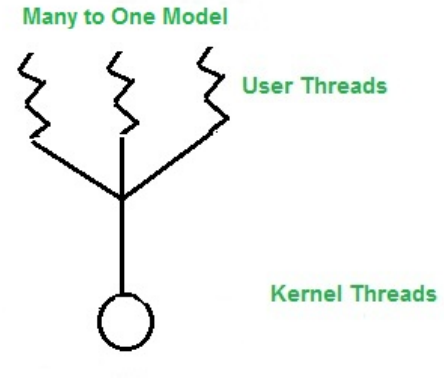
Many to Many Model

In this model, we have multiple user threads multiplex to the same or lesser number of kernel-level threads. The number of kernel-level threads is specific to the machine, advantage of this model is if a user thread is blocked we can schedule others user thread to other kernel thread. Thus, the System doesn't block if a particular thread is blocked.

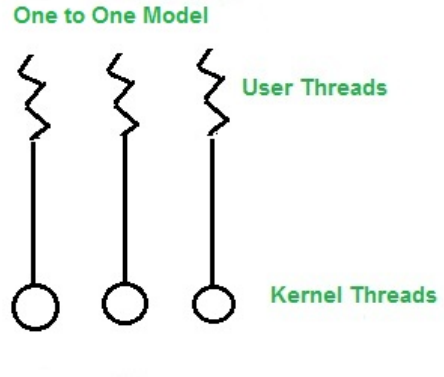


Many to One Model

In this model, we have multiple user threads mapped to one kernel thread. In this model when a user thread makes a blocking system call entire process blocks. As we have only one kernel thread and only one user thread can access kernel at a time, so multiple threads are not able to access multiprocessor at the same time.



One to One Model In this model, one to one relationship between the kernel and user thread. In this model, multiple threads can run on multiple processors. The problem with this model is that creating a user thread requires the corresponding kernel thread.



Benefits of Multithreading:

1. Resource Sharing
2. Responsiveness
3. Economy

4. Scalability

Multi-threaded programming - Thread libraries(Linux)

Three main types of libraries nowadays are used:
An API is provided by thread libraries to the programmers to create and handle the thread. It can be implemented in either kernel mode or in user mode. Former API was implemented solely within user mode, without any kernel support. The latter libraries involve system call and to do this kernel thread library support is required.

POSIX pthreads:

1. It is provided either as a kernel level of a user-level library. It extends the POSIX standards.
2. **Win32 threads:** It is a kernel level library on windows system.
3. **JAVA threads:** JAVA is not a platform independent, JAVA virtual machine (JVM) provide the atmosphere to run a JAVA file. Implementation is depending upon the Operating system and hardware on which JVM is running on; either POSIX threads or Win32 threads

- **POSIX:** The POSIX standard defines the specification of thread but implementation is not there.
- Solaris, Mac OSX, Tru 64 provide thread and for windows, it is available through public domain shareware.
- Global variables are shared among all threads.
- One thread can wait for others to rejoin before continuing.
- pThread begin execution in a specified function.

For example:

```
#include<pthread.h>
#include<stdio.h>
int add;
void *run(void *super);
int main(int argc, char *argv[])
{
pthread_p pid;
pthread_att_p att;

if (argc != 2)
{
fprintf(stderr, "usage.aout <integer value>\n");
return -1;
}
if(atoi(argv[1]) < 0)
fprintf(stderr, "%d must be >= 0\n", atoi(argv[1]));
return -1;
pthread_att_init(&att);
pthread_create(&pid, &att, super, argv[1]);
pthread_join(pid, NULL);
printf("add = %d\n" add);

void *super()
{
int u, upper = atoi(super);
add = 0;

for(u = 1; u <= upper; u++)
```

Run

Windows thread: The only difference between POSIX and windows thread is syntactic and nomenclature.

```
#include<windows.h>
#include<stdio.h>
DWORD Add;
DWORD WINAPI Summation (LPVOID Super)
{
```

```
 8  DWORD Upper = *(DWORD *)Super;
 9  for(DWORD i = 0; i <= Upper; i++)
10   add += i;
11  return 0;
12  }
13  int main (int argc, char *argv[])
14  {
15  DWORD ThreadId;
16  HANDLE ThreadHandle;
17  int Super;
18  if(argc != 2)
19  {
20  fprintf(stderr,"An integer parameter is required\n");
21  return -1;
22
23  }
24  Super = atoi(argv[1]);
25  if(Super < 0)
26  fprintf(stderr,"An integer >= 0 is required\n");
27  return -1;
28  }
29  ThreadHandle = CreateThread(Null, 0, addition, &Super, 0, &ThreadId);
30
```

Run

JAVA Thread:

- Threads are used in all JAVA programs, even in common single-threaded programs.
- The creation of new Threads requires Objects that implement the Runnable Interface, which means they contain a method "public void run( )". Any descendant of the Thread class will naturally contain such a method. (In practice the run( ) method must be overridden/provided for the thread to have any practical functionality.)
- By Creating a Thread Object does not mean that it starts the thread running - To do that the program must call the Thread's "start( )" method. Start( ) allocates and initializes memory for the Thread, and then calls the run( ) method. (No Programmers call run( ) directly.)
- It is well known that Java does not support global variables, A reference must be passed by threads to a Shared Object in order to share data, in this example the "Add" Object.
- It is noted that the JVM runs on top of a native OS and that the JVM specification does not specify what model to use for mapping Java threads to kernel threads. This decision depends on JVM implementation and may be one-to-one, many-to-many, or many to one.. (On a UNIX system the JVM normally uses PThreads and on a Windows system it normally uses windows threads.)

```
 1
 2
 3  Class Add
 4  {
 5  private int Add;
 6  public int get add()
 7  {
 8  return addition;
 9  }
10
11  public void setadd(int add)
12  {
13  this.add = add;
14  }}
15  Class Addition implements Runnable()
16  {
17  private int add;
18  private Add addValue;
19
20  public addition(int upper  add addValue)
21  {
22  int add = 0;
23  for(int i = o; i <= upper; i++)
24  add += i;
25  addvalue.setAdd(add);
```

```
26  }
27  }
28  public class Driver
29  {
30  public static void main(String args[])
```

Run

LIVE BATCHES

- Multi-threaded programming - Thread scheduling

In multiple-processor scheduling **multiple CPU's** are available and hence **Load Sharing** becomes possible. However multiple processor scheduling is more **complex** as compared to single processor scheduling. In multiple processor scheduling, there are cases when the processors are identical i.e. HOMOGENEOUS, in terms of their functionality, we can use any processor available to run any process in the queue.

Approaches to Multiple-Processor Scheduling -

One approach is when all the scheduling decisions and I/O processing are handled by a single processor which is called the **Master Server** and the other processors execute only the **user code**. This is simple and reduces the need for data sharing. This entire scenario is called **Asymmetric Multiprocessing**.

A second approach uses **Symmetric Multiprocessing** where each processor is **self scheduling**. All processes may be in a common ready queue or each processor may have its own private queue for ready processes. The scheduling proceeds further by having the scheduler for each processor examine the ready queue and select a process to execute.

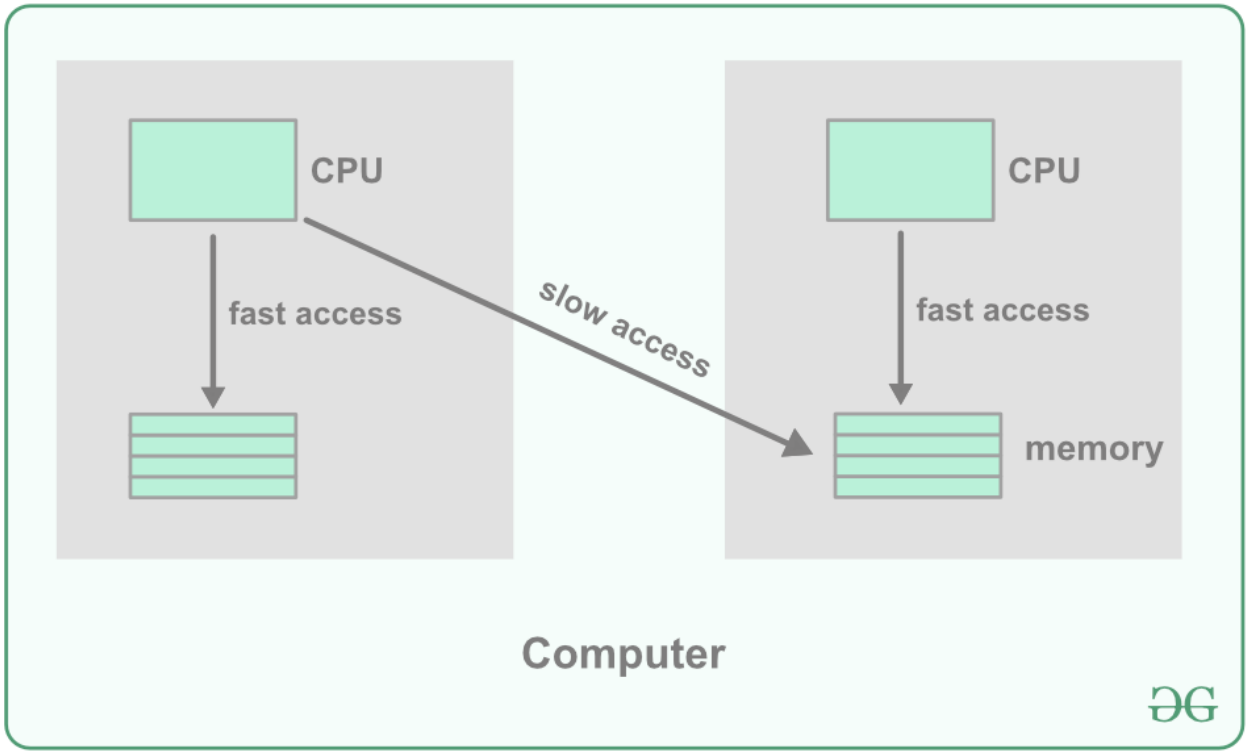
Processor Affinity -

Processor Affinity means a processes has an **affinity** for the processor on which it is currently running.
When a process runs on a specific processor there are certain effects on the cache memory. The data most recently accessed by the process populate the cache for the processor and as a result successive memory access by the process are often satisfied in the cache memory. Now if the process migrates to another processor, the contents of the cache memory must be invalidated for the first processor and the cache for the second processor must be repopulated. Because of the high cost of invalidating and repopulating caches, most of the SMP(symmetric multiprocessing) systems try to avoid migration of processes from one processor to another and try to keep a process running on the same processor. This is known as **PROCESSOR AFFINITY**.

There are two types of processor affinity:

1. **Soft Affinity -** When an operating system has a policy of attempting to keep a process running on the same processor but not guaranteeing it will do so, this situation is called soft affinity.
2. **Hard Affinity -** Some systems such as Linux also provide some system calls that support Hard Affinity which allows a process to migrate between processors.

The figure below is showing the architecture featuring non-uniform memory access which is depicting faster access to some parts than to other parts. This is the scenario of the system containing combined CPU and memory boards



Load Balancing -

Load Balancing is the **phenomena** which keeps the **workload** evenly **distributed** across all processors in an SMP system. Load balancing is necessary only on systems where each processor has its own private queue of the process which is eligible to execute. Load balancing is unnecessary because once a processor becomes idle it immediately extracts a runnable process from the common run queue. On SMP(symmetric multiprocessing), it is important to keep the workload balanced among all processors to fully utilize the benefits of having more than one processor else one or more processor will sit idle while other processors have high workloads along with lists of processors awaiting the CPU.

There are two general approaches to load balancing:

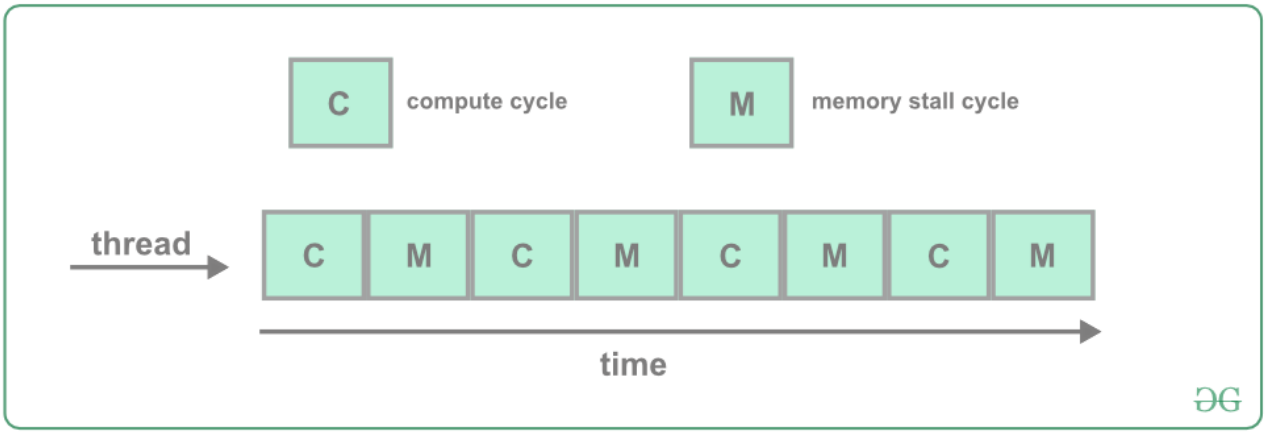
1. **Push Migration -** In push migration a task routinely checks the load on each processor and if it finds an imbalance then it evenly distributes the load on each processor by moving the processes from overloaded to idle or less busy processors.
2. **Pull Migration -** Pull Migration occurs when an idle processor pulls a waiting task from a busy processor for its execution.

Multicore Processors -

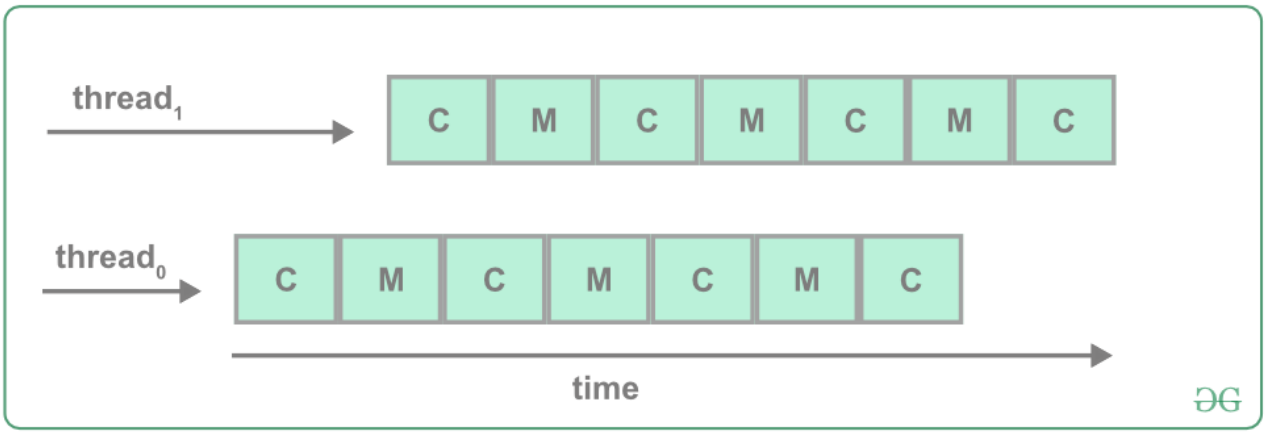
In multicore processors **multiple processors** cores are placed on the same physical chip. Each core has a register set to maintain its architectural state and thus appears to the operating system as a separate physical processor. **SMP systems** that use multicore processors are faster and consume **less power** than systems in which each processor has its own physical chip.

However multicore processors may **complicate** the scheduling problems. When a processor accesses memory then it spends a significant amount of time waiting for the data to become available. This situation is called **MEMORY STALL**. It occurs for various reasons such as cache miss, which is accessing the data that is not in the cache memory. In such cases, the processor can spend up to fifty percent of its time waiting for data to become available from the memory. To solve this problem recent hardware designs have implemented multithreaded processor cores in which two or more hardware threads are assigned to each core. Therefore if one thread stalls while waiting for the memory, the core can switch to another thread.

The figure below is depicting the memory stall:



The figure below is depicting the multithread multicore system:



There are two ways to multithread a processor:

1. Coarse-Grained Multithreading - In coarse-grained multithreading a thread executes on a processor until a long latency event such as a memory stall occurs, because of the delay caused by the long latency event, the processor must switch to another thread to begin execution. The cost of switching between threads is high as the instruction pipeline must be terminated before the other thread can begin execution on the processor core. Once this new thread begins execution it begins filling the pipeline with its instructions.

2. Fine-Grained Multithreading - This multithreading switches between threads at a much finer level mainly at the boundary of an instruction cycle. The architectural design of fine-grained systems include logic for thread switching and as a result, the cost of switching between threads is small.

Virtualization and Threading -

In this type of **multiple-processor** scheduling, even a single CPU system acts like a multiple-processor system. In a system with Virtualization, the virtualization presents one or more virtual CPU's to each of the virtual machines running on the system and then schedules the use of physical CPU'S among the virtual machines. Most virtualized environments have one host operating system and many guest operating systems. The host operating system creates and manages the virtual machines and each virtual machine has a guest operating system installed and applications running within that guest. Each guest operating system may be assigned for specific use cases, applications, and users, including time-sharing or even real-time operation. Any guest operating-system scheduling algorithm that assumes a certain amount of progress in a given amount of time will be negatively impacted by the virtualization. In a time-sharing operating system that tries to allow 100 milliseconds to each time slice to give users a reasonable response time. A given 100 millisecond time slice may take much more than 100 milliseconds of virtual CPU time. Depending on how busy the system is, the time slice may take a second or more which results in very poor response time for users logged into that virtual machine. The net effect of such scheduling layering is that individual virtualized operating systems receive only a portion of the available CPU cycles, even though they believe they are receiving all cycles and that they are scheduling all of those cycles. Commonly, the time-of-

day clocks in virtual machines are incorrect because timers take no longer to trigger than they would on dedicated CPU's.

Virtualizations can thus undo the good scheduling-algorithm efforts of the operating systems within virtual machines.

Difference between multithreading and multi-tasking:

1. The main difference between multitasking and multithreading is that in multitasking the system allows executing multiple programs and tasks at the same time, whereas, in multithreading, the system executes multiple threads of the different or same processes at the same time.
2. Multi-threading is more granular than multi-tasking. In multitasking, CPU switches between multiple programs to complete their execution in real time, while in multi-threading CPU switches between multiple threads of the same program. There is more context switching cost in multiple processes than switching between multiple threads of the same program.
3. Processes are heavyweight as compared to threads. They require their own address space, which means multi-tasking is heavy compared to multithreading.
4. For each process/program multitasking allocates separate memory and resources whereas, in multithreading, threads belonging to the same process share the same memory and resources like that of the process.

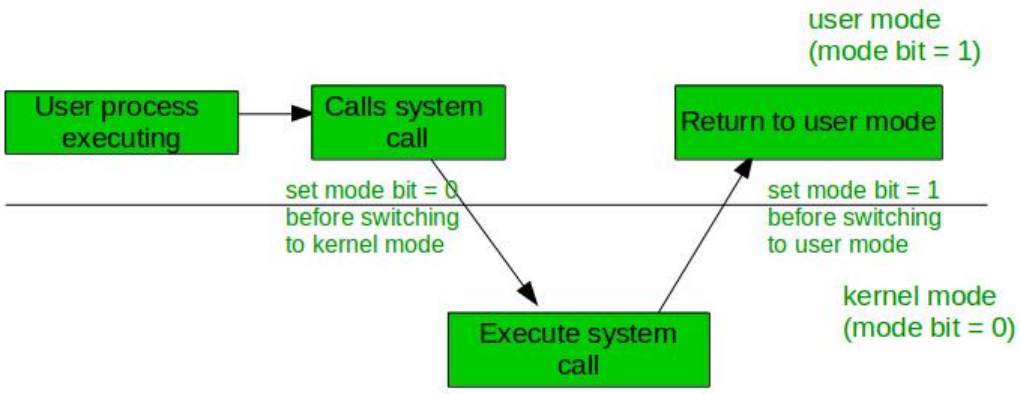
- System Calls

To understand **System Calls** in an UNIX-based system, one is required to know about the modes of operation of OS. There are basically 2 modes of operation namely, **Kernel** & **User** mode:
Kernel Mode The mode of operation in which critical OS procedures run. Only privileged instructions such as:

- Handling Interrupts
- Process Management
- I/O Management

are allowed to run. This is to ensure that no user application is able to tamper or cause a glitch in core-OS tasks, thus preventing any critical error which may cause the system to crash. When the OS boots, it begins execution in kernel-mode only. All user applications which are loaded afterwards are run on User Mode.

User Mode All user applications and high-level programs are run in User Mode so that they don't mess up with the critical OS procedures. However, sometimes a user application may require access to low-level utilities such as I/O peripherals, File System (Hard-drives), etc. For that, it requires the OS to switch from the User Mode of execution to Kernel Mode. The way of switching is done via System Calls:



We shall cover some basic system calls related to process management (creation, reaping etc.) as given below:

fork Fork system call is used to create a child process. It is a special function in the fact that this system call has a conditional return value:

- < 0: Unsuccessful Child creation
- 0: Value returned to the child process
- > 0: Value returned to the parent process. It is equal to the PID (Process ID) of the child process

If we count the parent as 1 single processes, then after each fork call, 2 processes are said to exist now (the original parent and the newly created child).
Execution continues from the point of fork() call in both child and parent processes. i.e. The same code is executed in both the processes. However, everything else (variables, stack, heap etc.) is duplicated.
Example:

```
#include <stdio.h>
#include <unistd.h>

int g_var;

int main()
{
    int l_var = 0;
    char *str;

    int pid = fork();
    if (pid < 0)
        printf ("Unsuccessful Child creation");
    else if (pid == 0) {    //Inside Child Process Only
        g_var++;
        l_var++;
        str = "From Child Process";
    }
    else {    //Inside Parent Process Only
        str = "From Parent Process";
    }

    printf ("%s: Global Var = %d, Local Var = %d\n", str, g_var, l_var);
    return 0;
}

```

Output:

```
From Parent Process: Global Var = 0, Local Var = 0
From Child Process: Global Var = 1, Local Var = 1
```

As can be seen, everything (global, local segment etc.) was duplicated in both child and parent. Only the variables corresponding to the child were incremented.

exec family Fork creates a new child process but has the same code executing afterwards. If we want to load a new program into the child process, we use the *exec family* calls. Exec calls replaces the current executing program with a new one. We demonstrate the usage of **execvp()** call:

```
#include <stdio.h>
#include <unistd.h>

int main()
{
    int pid = fork();
    if (pid < 0)
        printf ("Unsuccessful Child creation");
    else if (pid == 0) {    //Inside Child Process
        printf ("Still executing Same Code\n");
        execvp("ls");
        printf ("Unreachable Code as new Program loaded already\n");
    }
    else {    //Parent Process
        printf ("Inside Parent Process\n");
    }

    return 0;
}

```

Output:

```
Inside Parent Process
Still Executing Same Code
bin   dev  home        lib    lost+found  mnt  proc  run   srv  tmp  var
boot  etc  initrd.img  lib64  media       opt  root  sbin  sys  usr  vmlinuz
```

NOTE: Since, we loaded "ls" program into our child process, we never saw "Unreachable Code ..." string getting printed.

getpid Returns the Process ID of the currently executing process (program):

```c

#include <stdio.h>
#include <unistd.h>

int main()
{
    printf ("Process ID: %d", getpid());
    return 0;
}

```

Output:

```
Process ID: 10013
```

wait and exit Wait is a blocking-call which prevents the parent process from exiting without properly reaping any child process. Exit is used to terminate the current process. It takes the exit code as an argument, which is returned to the parent. This blocking effect is demonstrated below:

```c

#include <stdio.h>
#include <unistd.h>
#include <time.h>

int main()
{
    int pid = fork();

    if (pid < 0)
        printf ("Unsuccessful Child creation");
    else if (pid == 0) {    //Inside Child Process
        sleep(5);
        exit(0);
    }
    else {    //Parent Process
        printf ("Start: %ld\n", time(NULL));
        wait(NULL);
        printf ("End: %ld\n", time(NULL));

    }

    return 0;
}

```

Output:

```
Start: 1557231700
End: 1557231705
```

As we can see there is a difference of 5 secs. (because Child process was sleeping for 5secs). The parent waited for the child to exit.