Processes Concept

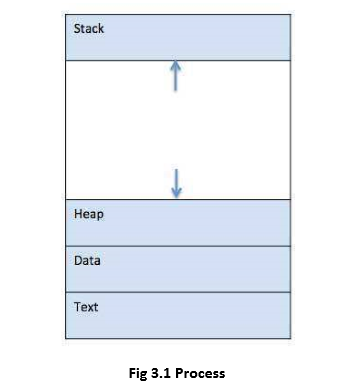
Process

A process is basically a program in execution. The execution of a process must progress in a sequential fashion.

A process is defined as an entity which represents the basic unit of work to be implemented in the system.

To put it in simple terms, we write our computer programs in a text file and when we execute this program, it becomes a process which performs all the tasks mentioned in the program.

When a program is loaded into the memory and it becomes a process, it can be divided into four sections ─ stack, heap, text and data. The following image shows a simplified layout of a process inside main memory



**Component & Description**

* **Stack -** The process Stack contains the temporary data such as method/function parameters, return address and local variables.
* **Heap -** This is dynamically allocated memory to a process during its run time.
* **Text -** This includes the current activity represented by the value of Program Counter and the contents of the processor's registers.
* **Data -** This section contains the global and static variables.

**Program**

A program is a piece of code which may be a single line or millions of lines. A computer program is usually written by a computer programmer in a programming language. For example, here is

a simple program written in C programming language –

#include<stdio.h>

intmain ()

{

printf("Hello, World! \n");

return0;

}

A computer program is a collection of instructions that performs a specific task when executed by a computer. When we compare a program with a process, we can conclude that a process is a dynamic instance of a computer program.

A part of a computer program that performs a well-defined task is known as an algorithm. A collection of computer programs, libraries and related data are referred to as software.

**Scheduling Concepts**

**Definition**

The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.

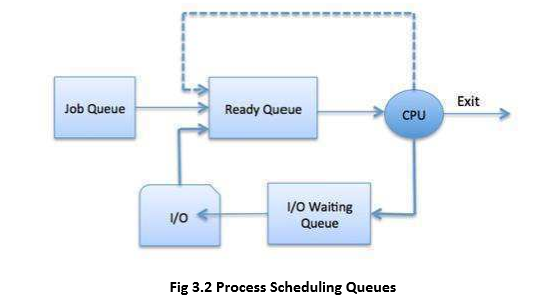
Process scheduling is an essential part of Multiprogramming operating systems. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.

**Process Scheduling Queues**

The OS maintains all PCBs in Process Scheduling Queues. The OS maintains a separate queue for each of the process states and PCBs of all processes in the same execution state are placed in the same queue. When the state of a process is changed, its PCB is unlinked from its current queue and moved to its new state queue.

The Operating System maintains the following important process scheduling queues –

* **Job queue−** this queue keeps all the proesses in the system.
* **Ready queue−** this queue keeps a set of all processes residing in main memory, ready and waiting to execute. A new process is always put in this queue.
* **Device queues−** the processes which are blocked due to unavailability of a I/O devices constitute this queue.



The OS can use different policies to manage each queue (FIFO, Round Robin, Priority, etc.). The OS scheduler determines how to move processes between the ready and run queues which can only have one entry per processor core on the system; in the above diagram, it has been merged with the CPU.

**Two-State Process Model**

Two-state process model refers to running and non-running states which are described below –

**State & Description**

**Running -** When a new process is created, it enters into the system as in the running state.

**Not Running -** Processes that are not running are kept in queue, waiting for their turn to execute. Each entry in the queue is a pointer to a particular process. Queue is implemented by using linked list. Use of dispatcher is as follows. When a process is interrupted, that process is transferred in the waiting queue. If the process has completed or aborted, the process is discarded. In either case, the dispatcher then selects a process from the queue to execute.

**Types of Schedulers**

Schedulers are special system software which handles process scheduling in various ways. Their main task is to select the jobs to be submitted into the system and to decide which process to run. Schedulers are of three types –

* Long-Term Scheduler
* Short-Term Scheduler
* Medium-Term Scheduler

**Long Term Scheduler**

* It is also called a job scheduler. A long-term scheduler determines which programs are admitted to the system for processing. It selects processes from the queue and loads them into memory for execution. Process loads into the memory for CPU scheduling.
* The primary objective of the job scheduler is to provide a balanced mix of jobs, such as I/O bound and processor bound. It also controls the degree of multiprogramming. If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system.
* On some systems, the long-term scheduler may not be available or minimal. Time-sharing operating systems have no long-term scheduler. When a process changes the state from new to ready, then there is use of long-term scheduler.

**Short Term Scheduler**

* It is also called as CPU scheduler. Its main objective is to increase system performance in accordance with the chosen set of criteria. It is the change of ready state to running state of the process. CPU scheduler selects a process among the processes that are ready to execute and allocates CPU to one of them.
* Short-term schedulers, also known as dispatchers, make the decision of which process to execute next. Short-term schedulers are faster than long-term schedulers.

**Medium Term Scheduler**

* Medium-term scheduling is a part of swapping. It removes the processes from the memory. It reduces the degree of multiprogramming. The medium-term scheduler is in-charge of handling the swapped out-processes.
* A running process may become suspended if it makes an I/O request. Suspended processes cannot make any progress towards completion. In this condition, to remove the process from memory and make space for other processes, the suspended process is moved to the secondary storage. This process is called swapping, and the process is said to be swapped out or rolled out. Swapping may be necessary to improve the process mix.

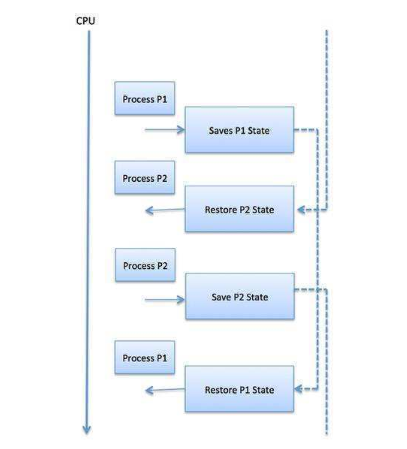
**Comparison among Scheduler**

|  |  |  |  |
| --- | --- | --- | --- |
| **S.NO** | **Long-Term Scheduler** | **Short-Term Schedule** | **Medium-Term Scheduler** |
| **1** | It is a job scheduler | It is a CPU scheduler | It is a process swapping scheduler. |
| **2** | Speed is lesser than short term scheduler | Speed is fastest among other two | Speed is in between both short and long term scheduler. |
| **3** | It controls the degree of multiprogramming | It provides lesser control over degree of multiprogramming | It reduces the degree of multiprogramming. |
| **4** | It is almost absent or minimal in time sharing system | It is also minimal in time sharing system | It is a part of Time sharing systems. |
| **5** | It selects processes from pool and loads them into memory for execution | It selects those processes which are ready to execute | It can re-introduce the process into memory and execution can be continued. |
|  |  |  |  |
|  |  |  |  |

**Context Switch**

A context switch is the mechanism to store and restore the state or context of a CPU in Process Control block so that a process execution can be resumed from the same point at a later time. Using this technique, a context switcher enables multiple processes to share a single CPU. Context switching is an essential part of a multitasking operating system features.

When the scheduler switches the CPU from executing one process to execute another, the state from the current running process is stored into the process control block. After this, the state for the process to run next is loaded from its own PCB and used to set the PC, registers, etc. At that point, the second process can start executing.

****

**Context Switch**

Context switches are computationally intensive since register and memory state must be saved and restored. To avoid the amount of context switching time, some hardware systems employ two or more sets of processor registers. When the process is switched, the following information is stored for later use.

* Program Counter
* Scheduling inform
* Base and limit register value
* Currently used register
* Changed State
* I/O State information
* Accounting information

**Process Life Cycle & Process State diagram**

When a process executes, it passes through different states. These stages may differ in different operating systems, and the names of these states are also not standardized.

In general, a process can have one of the following five states at a time.

**State & Description**

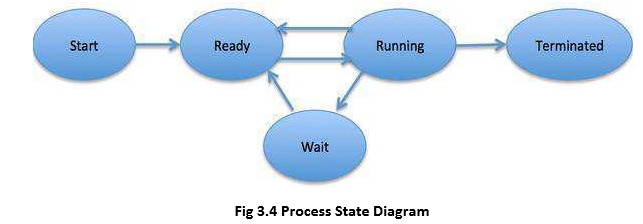
**Start -** This is the initial state when a process is first started/ created.

**Ready -** The process is waiting to be assigned to a processor. Ready processes are waiting to have the processor allocated to them by the operating system so that they can run. Process may come into this state after Start state or while running it by but interrupted by the scheduler to assign CPU to some other process.

**Running -** Once the process has been assigned to a processor by the OS scheduler, the process state is set to running and the processor executes its instructions.

**Waiting -** Process moves into the waiting state if it needs to wait for a resource, such as waiting for user input, or waiting for a file to become available.

**Terminated or Exit -** Once the process finishes its execution, or it is terminated by the operating system, it is moved to the terminated state where it waits to be removed from main memory.



**Process Control Block (PCB)**

A Process Control Block is a data structure maintained by the Operating System for every process. The PCB is identified by an integer process ID (PID). A PCB keeps all the information needed to keep track of a process as listed below in the table –

**Information & Description**

**Process State -** The current state of the process i.e., whether it is ready, running, waiting, or whatever.

**Process privileges -** This is required to allow/disallow access to system resources.

**Process ID –** Unique identification for each of the process in the operating system.

**Pointer -** A pointer to parent process.

**Program Counter -** Program Counter is a pointer to the address of the next instruction to be executed for this process.

**CPU registers -** Various CPU registers where process need to be stored for execution for running state.

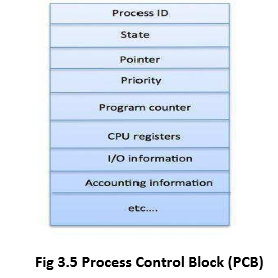
**CPU Scheduling Information -** Process priority and other scheduling information which is required to schedule the process.

**Memory management information -** This includes the information of page table, memory limits, Segment table depending on memory used by the operating system.

**Accounting information -** This includes the amount of CPU used for process execution, time limits, execution ID etc.

**IO status information -** This includes a list of I/O devices allocated to the process.

The architecture of a PCB is completely dependent on Operating System and may contain different information in different operating systems. Here is a simplified diagram of a PCB –



The PCB is maintained for a process throughout its lifetime, and is deleted once the process terminates.

**Scheduling Algorithms**

A Process Scheduler schedules different processes to be assigned to the CPU based on particular scheduling algorithms. There are six popular process scheduling algorithms

* Shortest-Job-Next (SJN) Scheduling
* Priority Scheduling
* Shortest Remaining Time
* Round Robin(RR) Scheduling
* Multiple-Level Queues Scheduling
* First-Come, First-Served (FCFS) Scheduling

These algorithms are either non-preemptive or preemptive. Non-preemptive algorithms are designed so that once a process enters the running state; it cannot be preempted until it completes its allotted time, whereas the preemptive scheduling is based on priority where a scheduler may preempt a low priority running process anytime when a high priority process enters into a ready state.

**First Come First Serve (FCFS)**

* Jobs are executed on first come, first serve basis.
* It is a non-preemptive, pre-emptive scheduling algorithm.
* Easy to understand and implement.
* Its implementation is based on FIFO queue.
* Poor in performance as average wait time is high.

**Shortest Job Next (SJN)**

* This is also known as shortest job first, or SJF
* This is a non-preemptive, pre-emptive scheduling algorithm.
* Best approach to minimize waiting time.
* Easy to implement in Batch systems where required CPU time is known in advance.
* Impossible to implement in interactive systems where required CPU time is not known.
* The processes should know in advance how much time process will take.

**Priority Based Scheduling**

* Priority scheduling is a non-preemptive algorithm and one of the most common scheduling algorithms in batch systems
* Each process is assigned a priority. Process with highest priority is to be executed first and so on.
* Processes with same priority are executed on first come first served basis.
* Priority can be decided based on memory requirements, time requirements or any other resource requirement.

**Shortest Remaining Time**

* Shortest remaining time (SRT) is the preemptive version of the SJN algorithm.
* The processor is allocated to the job closest to completion but it can be preempted by a newer ready job with shorter time to completion.
* It is often used in batch environments where short jobs need to give preference.
* Impossible to implement in interactive systems where required CPU time is not known.

**Round Robin Scheduling**

* Round Robin is the preemptive process scheduling algorithm.
* Each process is provided a fix time to execute, it is called a quantum.
* Once a process is executed for a given time period, it is preempted and other process executes for a given time period.
* Context switching is used to save states of preempted processes.

**Multiple-Level Queues Scheduling**

Multiple-level queues are not an independent scheduling algorithm. They make use of other existing algorithms to group and schedule jobs with common characteristics.

* Multiple queues are maintained for processes with common characteristics.
* Each queue can have its own scheduling algorithms.
* Priorities are assigned to each queue.

For example, CPU-bound jobs can be scheduled in one queue and all I/O-bound jobs in another queue. The Process Scheduler then alternately selects jobs from each queue and assigns them to the CPU based on the algorithm assigned to the queue.

**Algorithm Evaluation**

How do we select a CPU scheduling algorithm for a particular system?

There are many scheduling algorithms, each with its own parameters. As a result, selecting an algorithm can be difficult. The first problem is defining the criteria to be used in selecting an algorithm. Criteria are often defined in terms of CPU utilization, response time, or throughput. To select an algorithm, we must first define the relative importance of these measures. Our criteria may include several measures, such as:

* Maximizing CPU utilization under the constraint that the maximum response time is 1 second
* Maximizing throughput such that turnaround time is (on average) linearly proportional to total execution time once the selection criteria have been defined, we want to evaluate the algorithms under consideration. We next describe the various evaluation methods we can use.

**Deterministic Modeling**

One major class of evaluation methods is analytic evaluation. Analytic evaluation uses the given algorithm and the system workload to produce a formula or number that evaluates the performance of the algorithm for that workload. One type of analytic evaluation is deterministic modeling. This method takes a particular predetermined workload and defines the performance of each algorithm for that workload. For example, assume that we have the workload shown below. All five processes arrive at time 0, in the order given, with the length of the CPU burst given in milliseconds:

**Queuing Models**

Another method of evaluating scheduling algorithms is to use queuing theory. Using data from real processes we can arrive at a probability distribution for the length of a burst time and the I/O times for a process. We can now generate these times with a certain distribution.

We can also generate arrival times for processes (arrival time distribution).

If we define a queue for the CPU and a queue for each I/O device we can test the various scheduling algorithms using queuing theory.

Knowing the arrival rates and the service rates we can calculate various figures such as average queue length, average wait time, CPU utilization etc.

One useful formula is Little’s Formula.

n= λ

Where

n is the average queue length

λ is the average arrival rate for new processes (e.g. five a second)

w is the average waiting time in the queue

Knowing two of these values we can, obviously, calculate the third. For example, if we know that eight processes arrive every second and there are normally sixteen processes in the queue we can compute that the average waiting time per process is two seconds.

The main disadvantage of using queuing models is that it is not always easy to define realistic distribution times and we have to make assumptions. This results in the model only being an approximation of what actually happens.

**Simulations**

Rather than using queuing models we simulate a computer. A Variable, representing a clock is incremented. At each increment the state of the simulation is updated.

Statistics are gathered at each clock tick so that the system performance can be analyzed.

The data to drive the simulation can be generated in the same way as the queuing model, although this leads to similar problems.

Alternatively, we can use trace data. This is data collected from real processes on real machines and is fed into the simulation. This can often provide good results and good comparisons over a range of scheduling algorithms.

However, simulations can take a long time to run, can take a long time to implement and the trace data may be difficult to collect and require large amounts of storage.

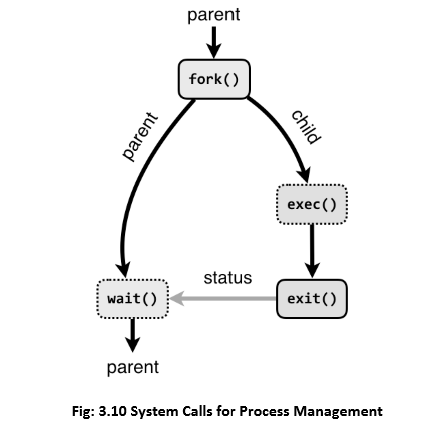
**Implementation**

The best way to compare algorithms is to implement them on real machines. This will give the best results but does have a number of disadvantages.

* It is expensive as the algorithm has to be written and then implemented on real hardware.
* If typical workloads are to be monitored, the scheduling algorithm must be used in a live situation. Users may not be happy with an environment that is constantly changing.
* If we find a scheduling algorithm that performs well there is no guarantee that this state will continue if the workload or environment changes.

**System Calls for Process Management**

Basic process management is done with a number of system calls, each with a single (simple) purpose. These system calls can then be combined to implement more complex behaviors.



The following system calls are used for basic process management.

* **fork :**A parent process uses fork to create a new child process. The child process is a copy of the parent. After fork, both parent and child executes the same program but in separate processes.
* **exec:** Replaces the program executed by a process. The child may use exec after a fork to replace the process’ memory space with a new program executable making the child execute a different program than the parent.
* **exit:** Terminates the process with an exit status.
* **wait:** The parent may use wait to suspend execution until a child terminates. Using wait the parent can obtain the exit status of a terminated child.

**Multiple Processor Schedulers:**

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 executes only the user code. This is simple and reduces the need of 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 process 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 accesses 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.**

* **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.
* **Hard Affinity –** Some systems such as Linux also provide some system calls that support Hard Affinity which allows a process to migrate between processors.

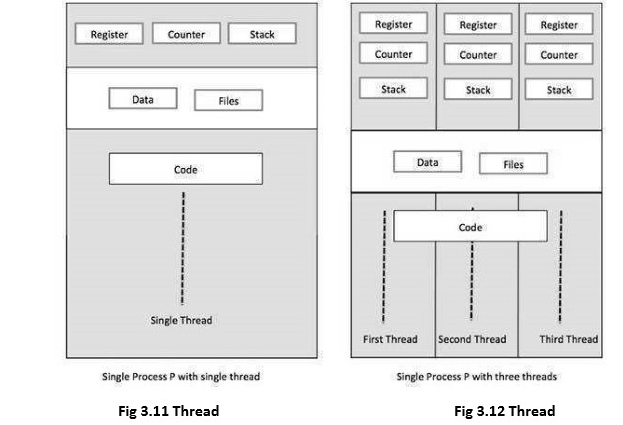
**Thread**

A thread is a flow of execution through the process code, with its own program counter that keeps track of which instruction to execute next, system registers which hold its current working variables, and a stack which contains the execution history.

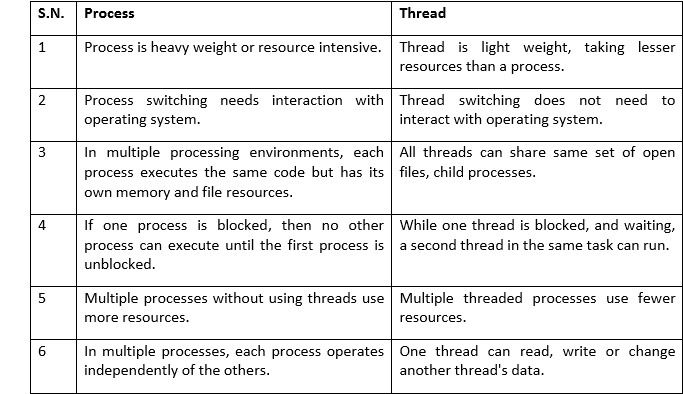
A thread shares with its peer threads little information like code segment, data segment and open files. When one thread alters a code segment memory item, all other threads see that.

A thread is also called a lightweight process. Threads provide a way to improve application performance through parallelism. Threads represent a software approach to improving performance of operating system by reducing the overhead thread is equivalent to a classical process.

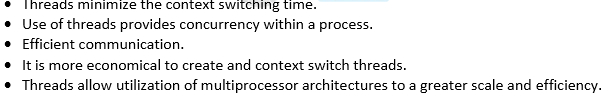
Each thread belongs to exactly one process and no thread can exist outside a process. Each thread represents a separate flow of control. Threads have been successfully used in implementing network servers and web server. They also provide a suitable foundation for parallel execution of applications on shared memory multiprocessors. The following figure shows the working of a single-threaded and a multithreaded process.

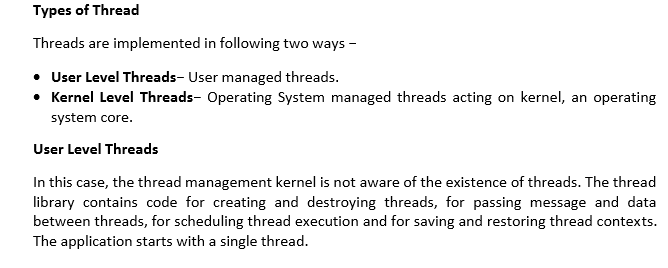
****

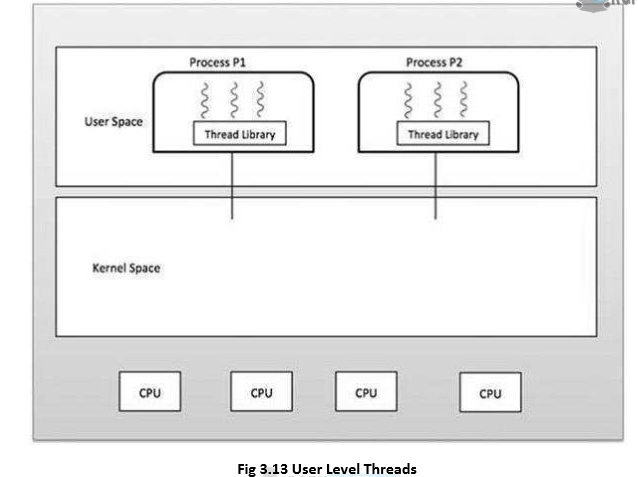
**Difference between Process and Thread**

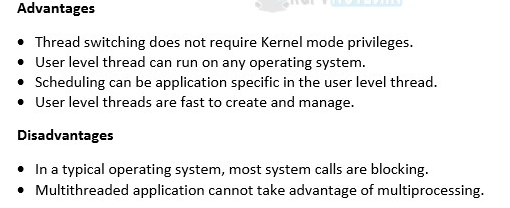
****

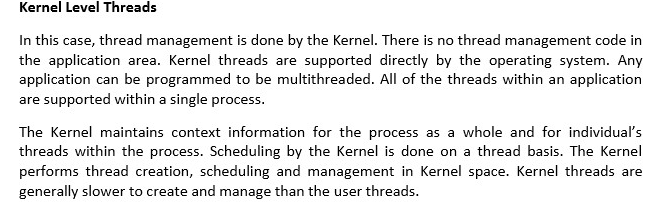
**Advantages of Thread**

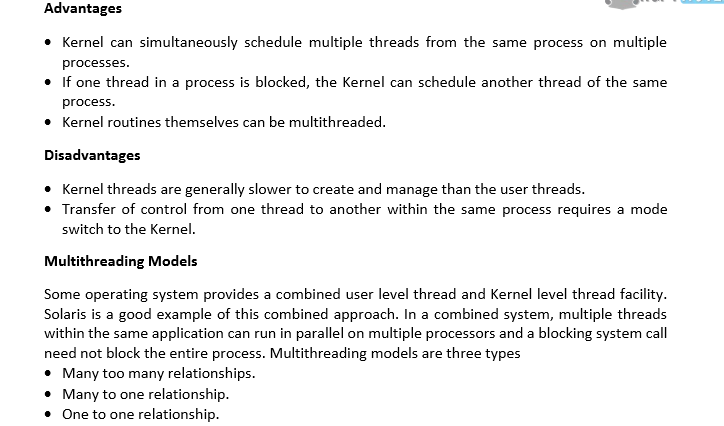
****

****

****

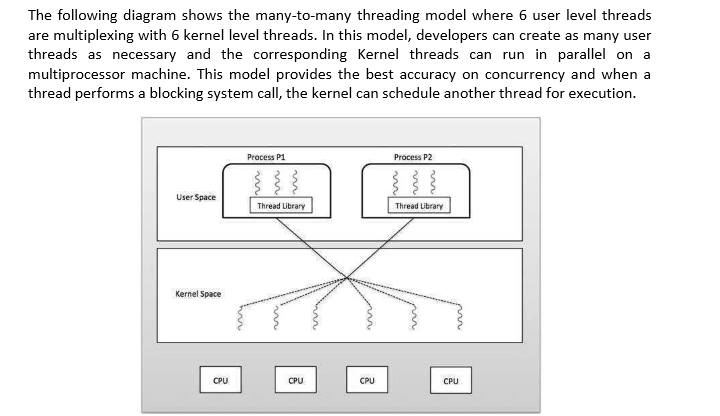
****

****

****

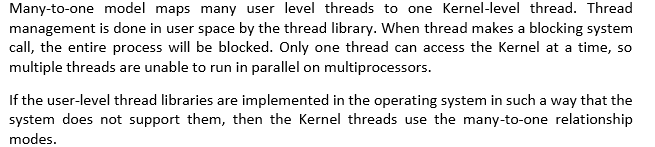
**Many too Many Model**

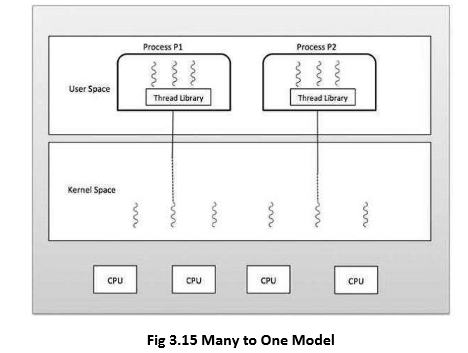
The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads.



**Fig 3.14 Many too Many Model**

**Many to One Model –**

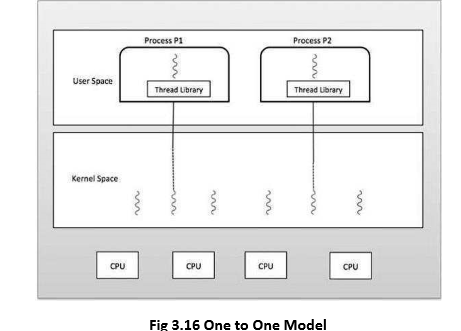
****

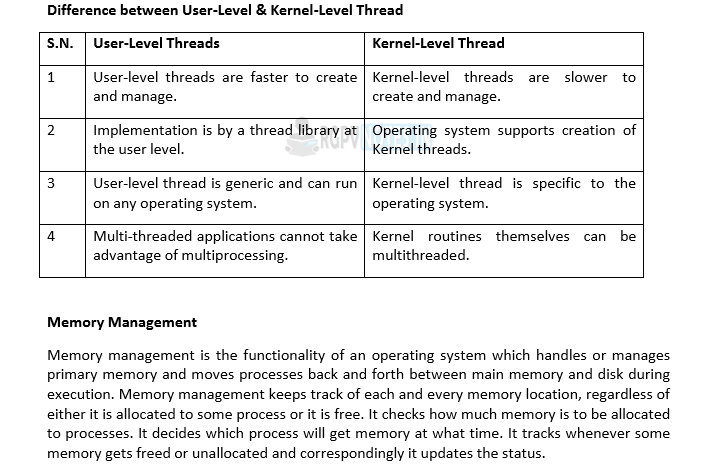


**One to One Model**

There is one-to-one relationship of user-level thread to the kernel-level thread. This model provides more concurrency than the many-to-one model. It also allows another thread to run when a thread makes a blocking system call. It supports multiple threads to execute in parallel on microprocessors.

Disadvantage of this model is that creating user thread requires the corresponding Kernel thread. OS/2, Windows NT and windows 2000 use one to one relationship model.

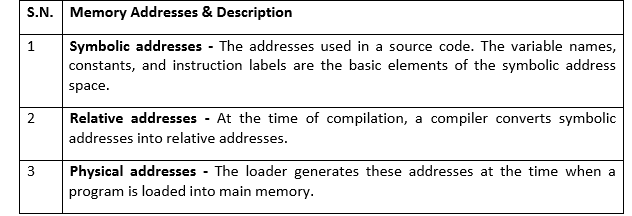


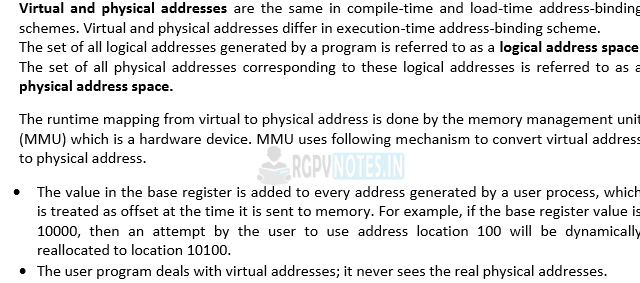


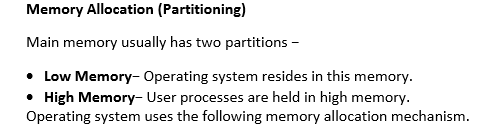
**Process Address Space**

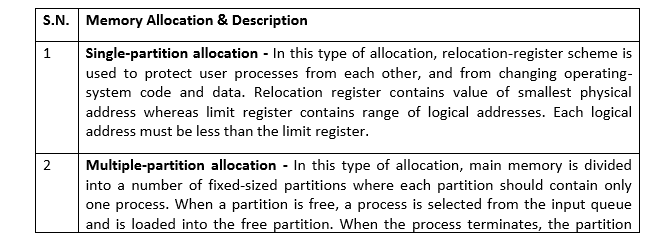
The process address space is the set of logical addresses that a process references in its code. For example, when 32-bit addressing is in use, addresses can range from 0 to 0x7fffffff; that is, 2^31 possible numbers, for a total theoretical size of 2 gigabytes.

The operating system takes care of mapping the logical addresses to physical addresses at the time of memory allocation to the program. There are three types of addresses used in a program before and after memory is allocated-

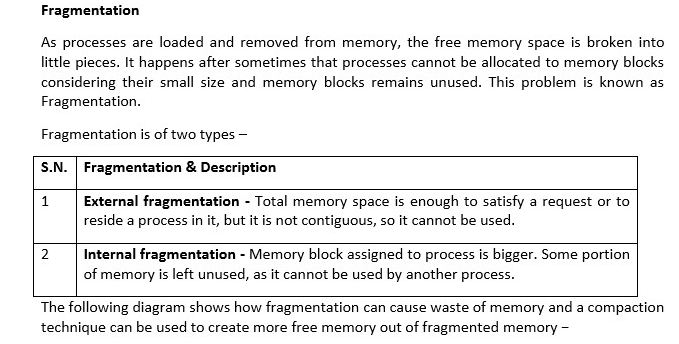


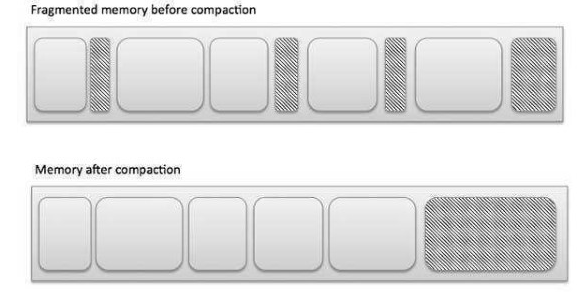












**Fig 3.17 Fragmentation**

**External fragmentation** can be reduced by compaction or shuffle memory contents to place all free memory together in one large block. To make compaction feasible, relocation should be dynamic.

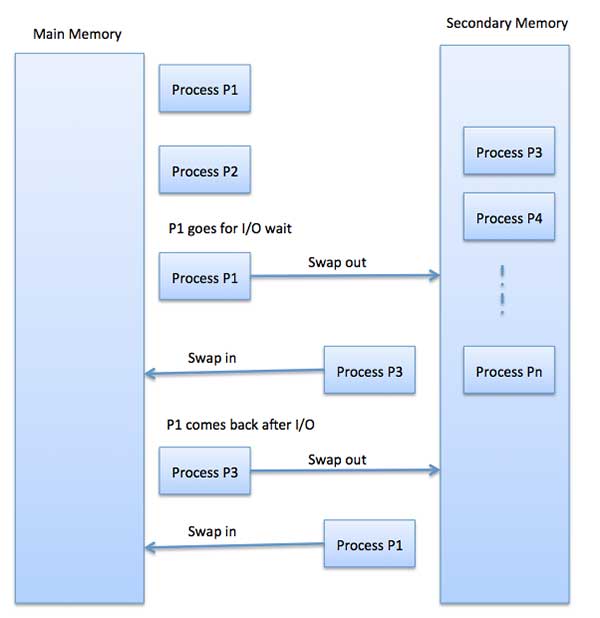
**The internal fragmentation** can be reduced by effectively assigning the smallest partition but large enough for the process.

**Swapping**

Swapping is mechanisms in which a process can be swapped temporarily out of main memory

(or move) to secondary storage (disk) and make that memory available to other processes. At some later time, the system swaps back the process from the secondary storage to main memory.

Though performance is usually affected by swapping process but it helps in running multiple and big processes in parallel and that's the reason. **Swapping is also known as a technique for memory compaction.**

****

**Fig 3.18 Swapping**

The total time taken by swapping process includes the time it takes to move the entire process to a secondary disk and then to copy the process back to memory, as well as the time the process takes to regain main memory.

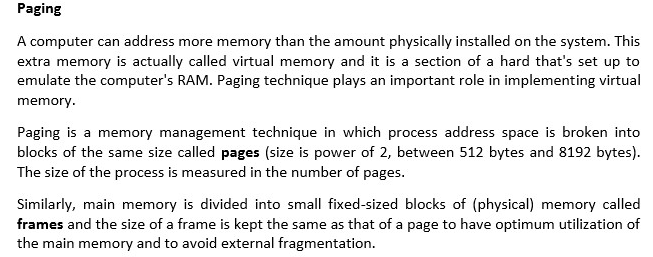
Let us assume that the user process is of size 2048KB and on a standard hard disk where swapping will take place has a data transfer rate around 1 MB per second. The actual transfer of the 1000K process to or from memory will take

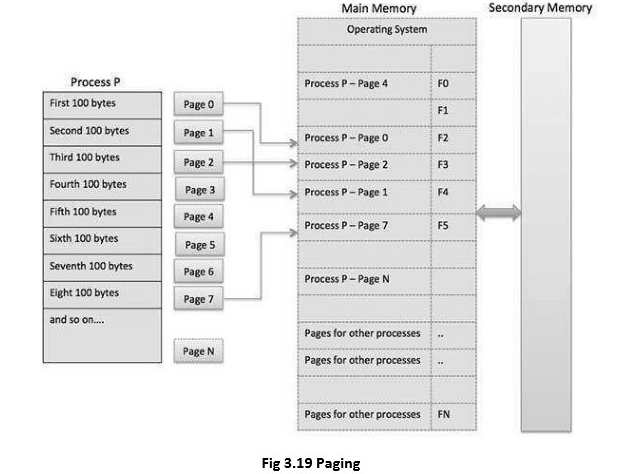
**2048KB / 1024KB per second**

**= 2 seconds**

**= 2000 milliseconds**

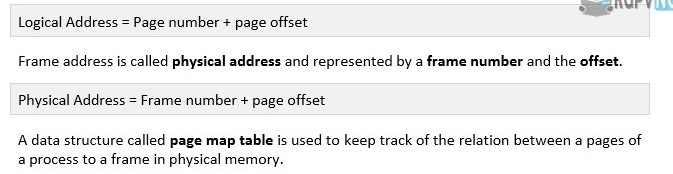
Now considering in and out time, it will take complete 4000 milliseconds plus other overhead where the process competes to regain main memory.

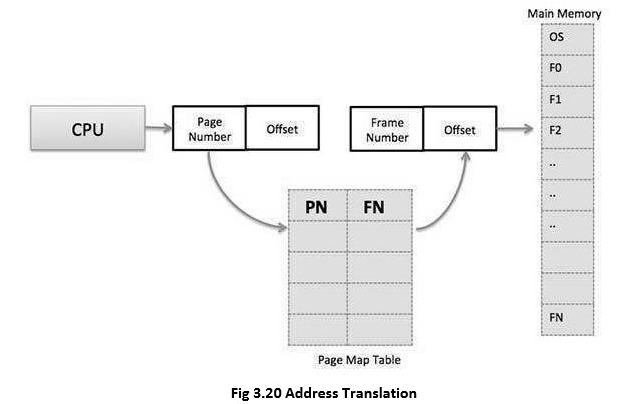
****

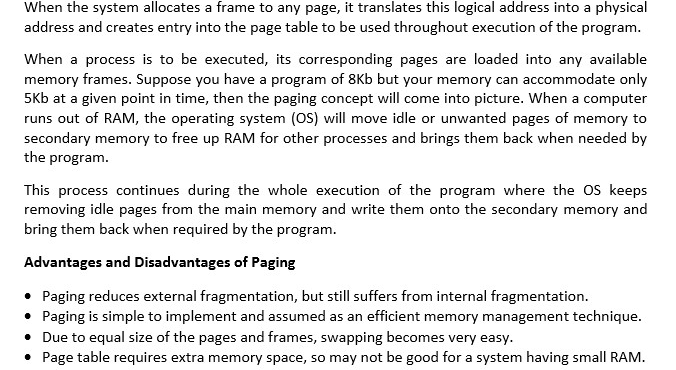


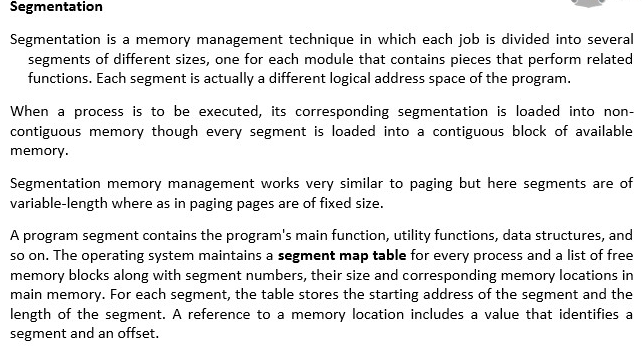
**Address Translation**

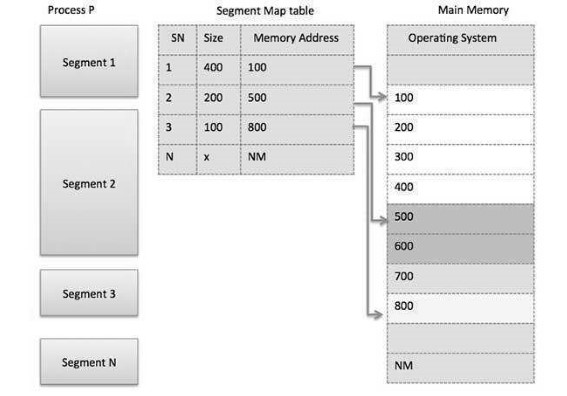
Page address is called logical address and represented by page number and the offset.









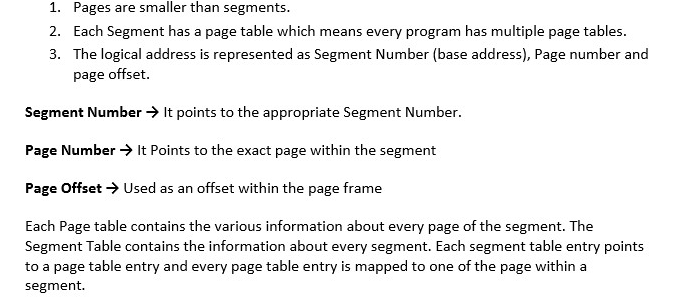


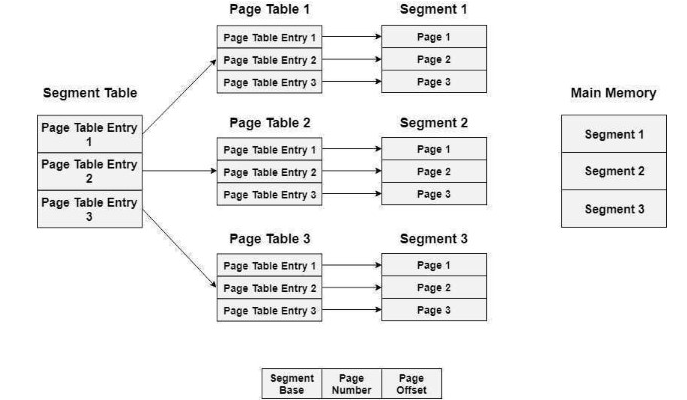
**Fig 3.21 Segmentation**

**Segmented Paging**

Pure segmentation is not very popular and not being used in many of the operating systems. However, Segmentation can be combined with Paging to get the best features out of both the techniques.

In Segmented Paging, the main memory is divided into variable size segments which are further divided into fixed size pages.



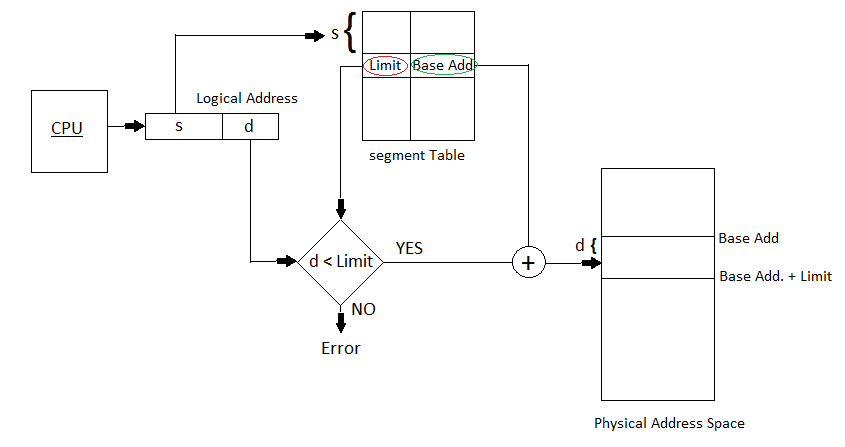


**Fig. 3.22 Segmented Paging**

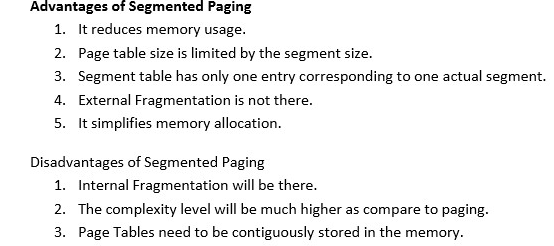
**Translation of logical address to physical address**

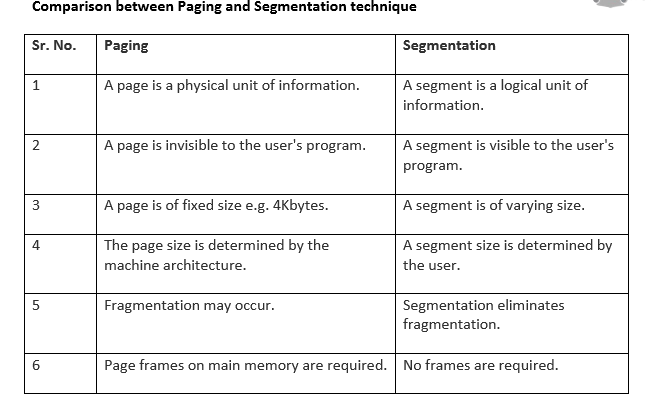
The CPU generates a logical address which is divided into two parts: Segment Number and Segment Offset. The Segment Offset must be less than the segment limit. Offset is further divided into Page number and Page Offset. To map the exact page number in the page table, the page number is added into the page table base.

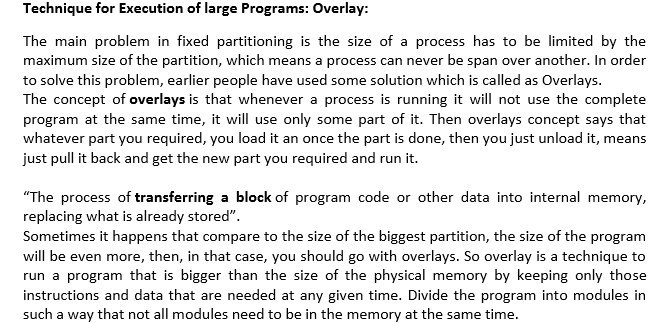
The actual frame number with the page offset is mapped to the main memory to get the desired word in the page of the certain segment of the process.

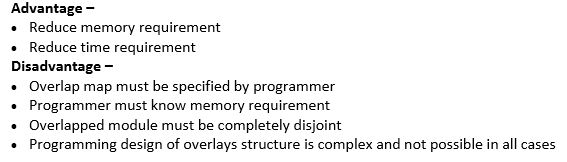


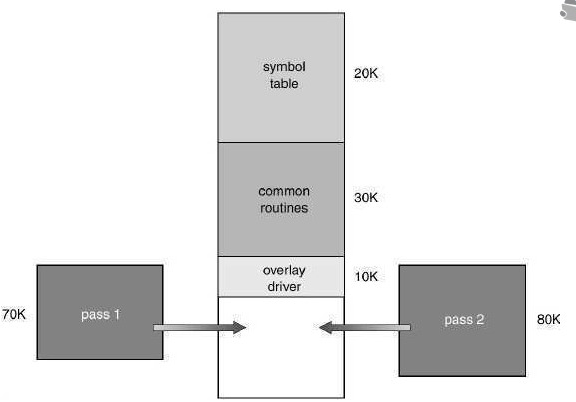
**Fig. 3.23 Translation of logical address to physical address**

****

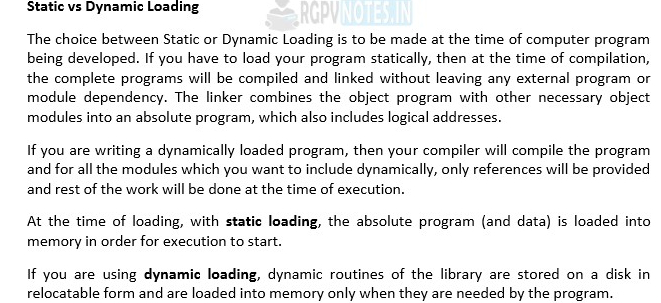


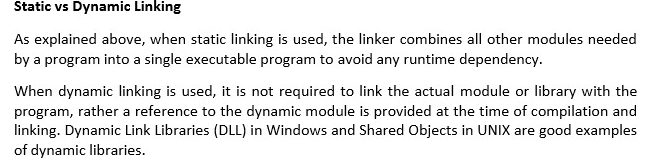


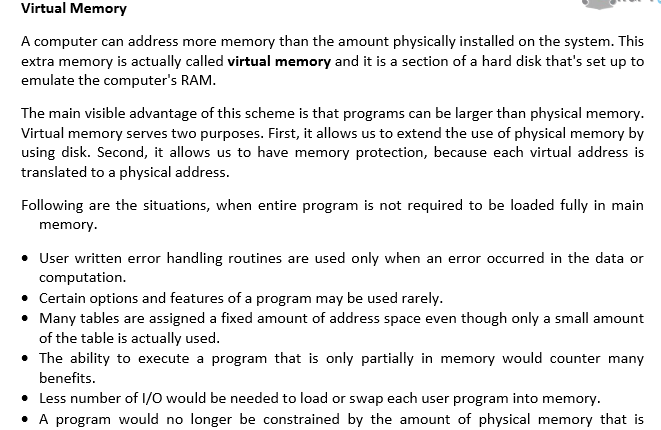


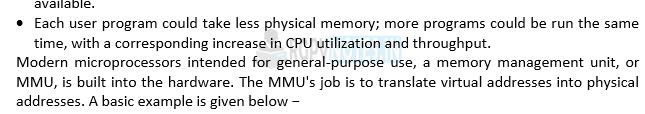


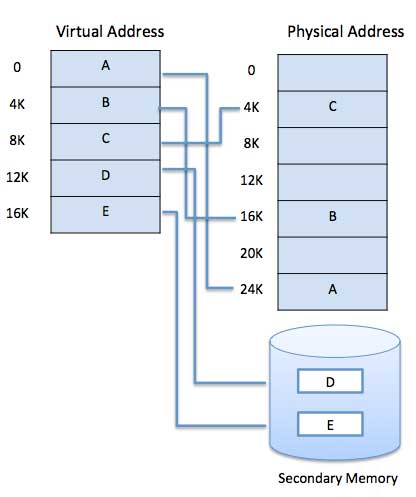
**Fig 3.24 Overlays for two pas Assembler**

****



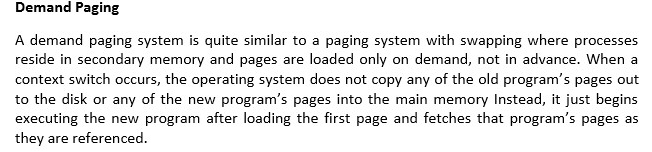


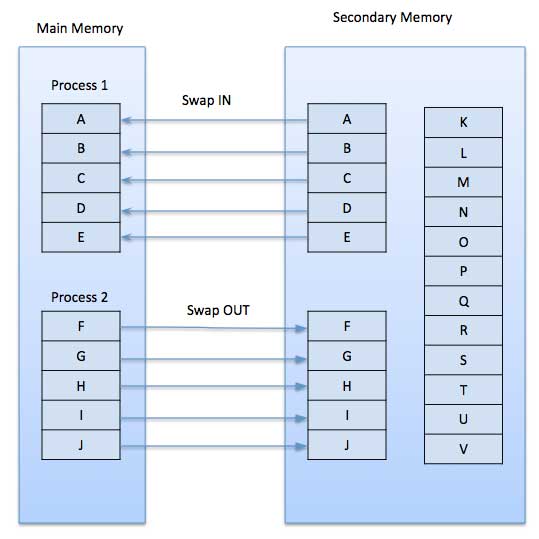




**Fig 3.25 Virtual Memory**

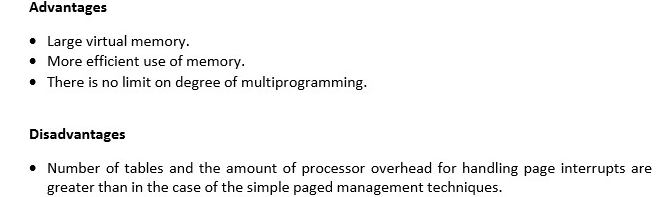
Virtual memory is commonly implemented by demand paging. It can also be implemented in a segmentation system. Demand segmentation can also be used to provide virtual memory.

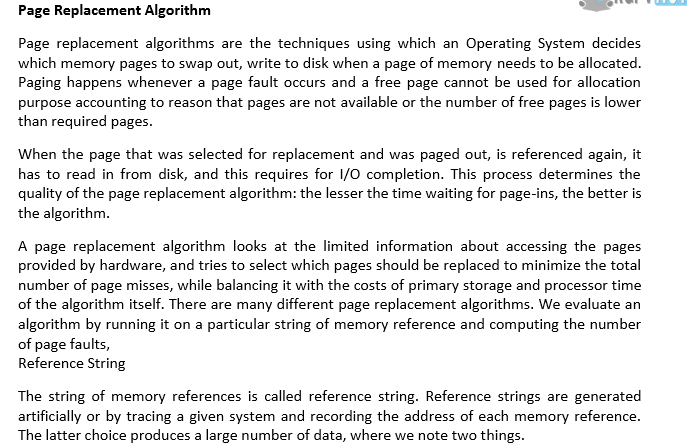


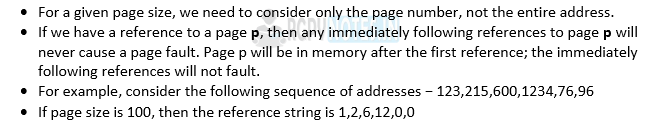


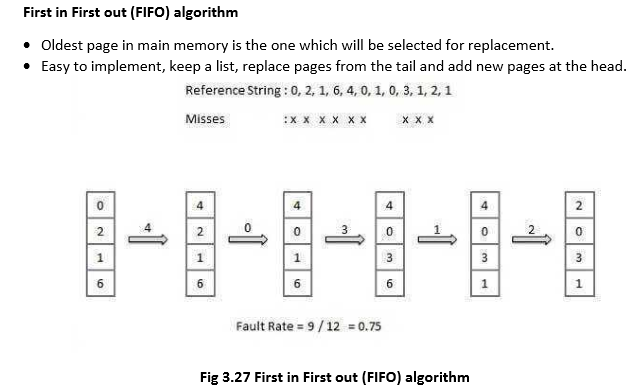
**Fig 3.26 Demand Paging**

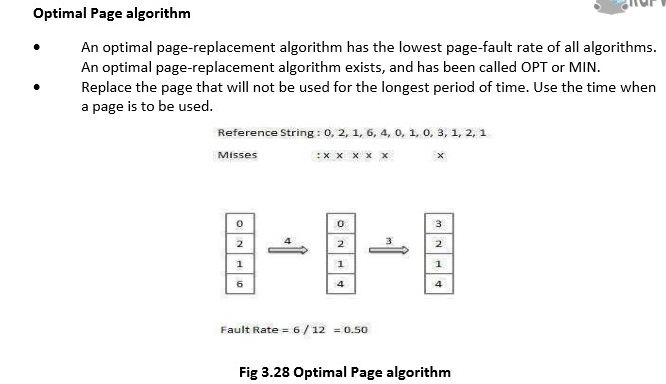
While executing a program, if the program references a page which is not available in the main memory because it was swapped out a little ago, the processor treats this invalid memory reference as a page fault and transfers control from the program to the operating system to demand the page back into the memory.

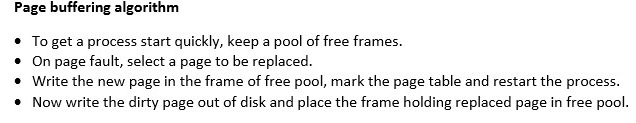


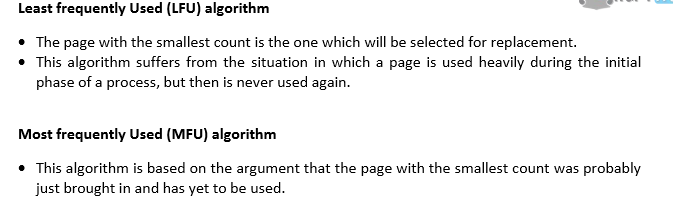












**Translation Lookaside Buffer (TLB) in Paging**

In Operating System (Memory Management Technique : [Paging](https://www.geeksforgeeks.org/operating-system-paging/)), for each process page table will be created, which will contain [Page Table Entry (PTE)](https://www.geeksforgeeks.org/operating-system-page-table-entries/). 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.

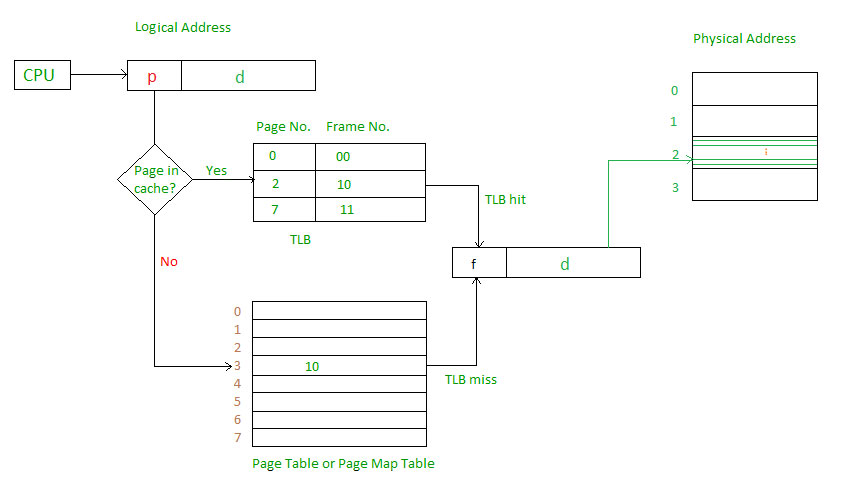
The problem initially was to fast access the main memory content based on address generated by CPU (i.e [logical/virtual address](https://www.geeksforgeeks.org/logical-vs-physical-address-in-operating-system/)). Initially, some people thought of using registers to store page table, as they are high-speed 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 is register size is small (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 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 as index while processing 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.

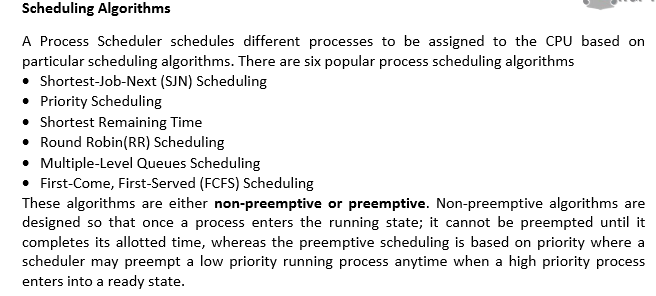


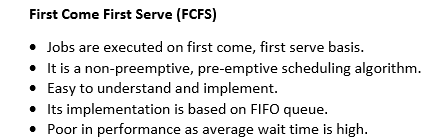
**Steps in TLB hit:** 

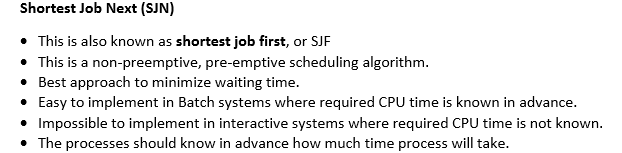
1. CPU generates virtual (logical) address.
2. It is checked in TLB (present).
3. Corresponding frame number is retrieved, which now tells where in the main memory page lies.

**Steps in TLB miss:** 

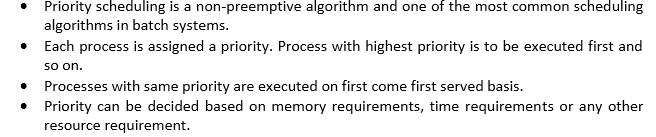
1. CPU generates virtual (logical) address.
2. It is checked in TLB (not present).
3. Now the page number is matched to page table residing in main memory (assuming page table contains all PTE).
4. Corresponding frame number is retrieved, which now tells where in the main memory page lies.
5. The TLB 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).

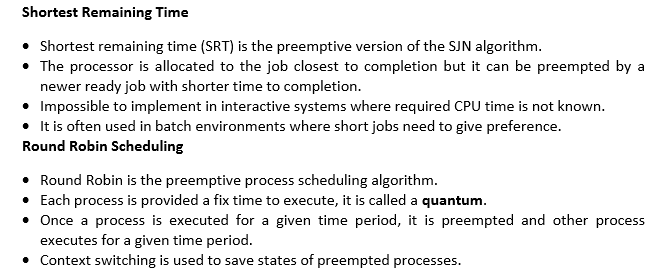






**Priority Based Scheduling**

****

****

