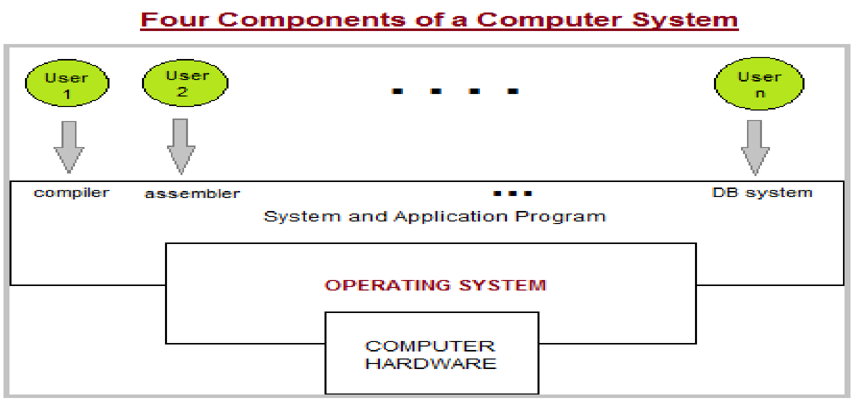
Introduction of Operating Systems

A computer system has many resources (hardware and software), which may be require to complete a task. The commonly required resources are input/output devices, memory, file storage space, CPU etc. The operating system acts as a manager of the above resources and allocates them to specific programs and users as necessary for their task. Therefore operating system is the resource manager i.e. it can manage the resource of a computer system internally. The resources are processor, memory, files, and I/O devices.



Two Views of Operating System

1. User's View
2. System View

User View :

The user view of the computer refers to the interface being used. Such systems are designed for one user to monopolize its resources, to maximize the work that the user is performing. In these cases, the operating system is designed mostly for ease of use, with some attention paid to performance, and none paid to resource utilization.

System View :

Operating system can be viewed as a resource allocator also. A computer system consists of many resources like - hardware and software - that must be managed efficiently. The operating system acts as the manager of the resources, decides between conflicting requests, controls execution of programs etc.

Operating System Management Tasks

1. **Processor management** which involves putting the tasks into order and pairing them into manageable size before they go to the CPU.

2. **Memory management** which coordinates data to and from RAM (random-access memory) and determines the necessity for virtual memory.
3. **Device management** which provides interface between connected devices.
4. **Storage management** which directs permanent data storage.
5. **Application** which allows standard communication between software and your computer.
6. **User interface** which allows you to communicate with your computer.

Functions of Operating System

1. It boots the computer
2. It performs basic computer tasks e.g. managing the various peripheral devices e.g. mouse, keyboard
3. It provides a user interface, e.g. command line, graphical user interface (GUI)
4. It handles system resources such as computer's memory and sharing of the central processing unit(CPU) time by various applications or peripheral devices.
5. It provides file management which refers to the way that the operating system manipulates, stores, retrieves and saves data.
6. Error Handling is done by the operating system. It takes preventive measures whenever required to avoid errors.

Types of Operating Systems

Following are some of the most widely used types of Operating system.

1. Simple Batch System
2. Multiprogramming Batch System
3. Multiprocessor System
4. Distributed Operating System
5. Realtime Operating System

SIMPLE BATCH SYSTEMS

- In this type of system, there is no direct interaction between user and the computer.
- The user has to submit a job (written on cards or tape) to a computer operator.
- Then computer operator places a batch of several jobs on an input device.
- Jobs are batched together by type of languages and requirement.
- Then a special program, the monitor, manages the execution of each program in the batch.
- The monitor is always in the main memory and available for execution.

Following are some disadvantages of this type of system :

1. Zero interaction between user and computer.
2. No mechanism to prioritize processes.

MULTIPROGRAMMING BATCH SYSTEMS

- In this the operating system, picks and begins to execute one job from memory.
- Once this job needs an I/O operation operating system switches to another job (CPU and OS always busy).
- Jobs in the memory are always less than the number of jobs on disk(Job Pool).
- If several jobs are ready to run at the same time, then system chooses which one to run (CPU Scheduling).
- In Non-multiprogrammed system, there are moments when CPU sits idle and does not do any work.
- In Multiprogramming system, CPU will never be idle and keeps on processing.

Time-Sharing Systems are very similar to Multiprogramming batch systems. In fact time sharing systems are an extension of multiprogramming systems.

In time sharing systems the prime focus is on minimizing the response time, while in multiprogramming the prime focus is to maximize the CPU usage.

MULTIPROCESSOR SYSTEMS

A multiprocessor system consists of several processors that share a common physical memory. Multiprocessor system provides higher computing power and speed. In multiprocessor system all processors operate under single operating system. Multiplicity of the processors and how they do act together are transparent to the others.

Following are some advantages of this type of system.

1. Enhanced performance
2. Execution of several tasks by different processors concurrently, increases the system's throughput without speeding up the execution of a single task.
3. If possible, system divides task into many subtasks and then these subtasks can be executed in parallel in different processors. Thereby speeding up the execution of single tasks.

DISTRIBUTED OPERATING SYSTEMS

The motivation behind developing distributed operating systems is the availability of powerful and inexpensive microprocessors and advances in communication technology.

These advancements in technology have made it possible to design and develop distributed systems comprising of many computers that are inter connected by communication networks. The main benefit of distributed systems is its low price/performance ratio.

Following are some advantages of this type of system.

1. As there are multiple systems involved, user at one site can utilize the resources of systems at other sites for resource-intensive tasks.
2. Fast processing.
3. Less load on the Host Machine.

REAL-TIME OPERATING SYSTEM

It is defined as an operating system known to give maximum time for each of the critical operations that it performs, like OS calls and interrupt handling.

The Real-Time Operating system which guarantees the maximum time for critical operations and complete them on time are referred to as **Hard Real-Time Operating Systems.**

While the real-time operating systems that can only guarantee a maximum of the time, i.e. the critical task will get priority over other tasks, but no assurity of completeing it in a defined time. These systems are referred to as **Soft Real-Time Operating Systems**.

What is a Process?

A program in the execution is called a Process. Process is not the same as program. A process is more than a program code. A process is an 'active' entity as opposed to program which is considered to be a 'passive' entity. Attributes held by process include hardware state, memory, CPU etc.

Process memory is divided into four sections for efficient working :

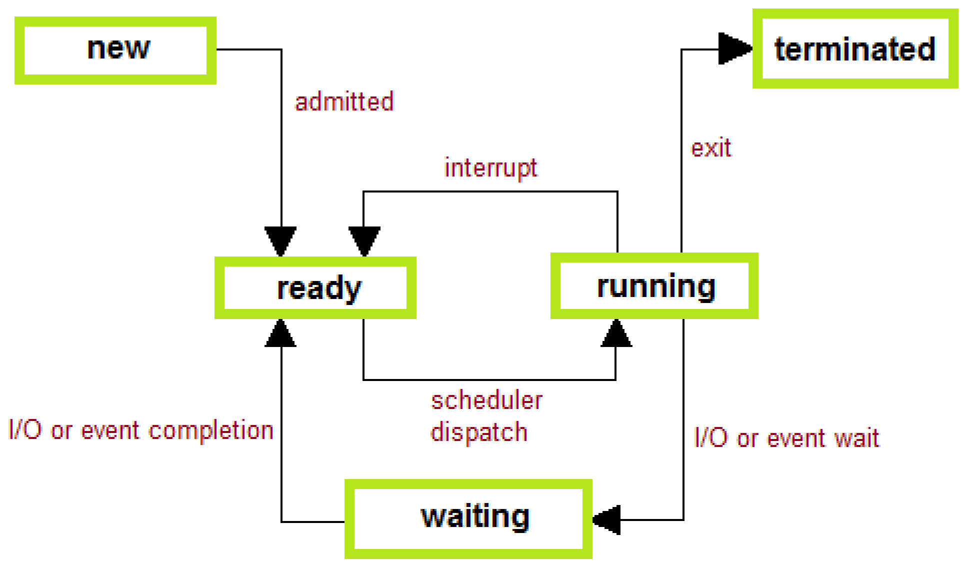
- The text section is made up of the compiled program code, read in from non-volatile storage when the program is launched.
- The data section is made up the global and static variables, allocated and initialized prior to executing the main.
- The heap is used for the dynamic memory allocation, and is managed via calls to new, delete, malloc, free, etc.
- The stack is used for local variables. Space on the stack is reserved for local variables when they are declared.

PROCESS STATE

Processes can be any of the following states :

- **New** - The process is in the stage of being created.
- **Ready** - The process has all the resources available that it needs to run, but the CPU is not currently working on this process's instructions.
- **Running** - The CPU is working on this process's instructions.

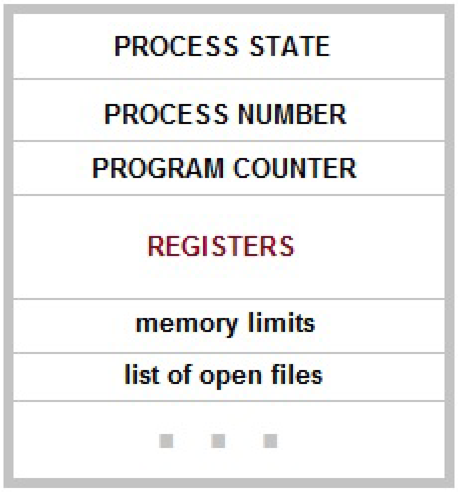
- **Waiting** - The process cannot run at the moment, because it is waiting for some resource to become available or for some event to occur.
- **Terminated** - The process has completed.



PROCESS CONTROL BLOCK

There is a Process Control Block for each process, enclosing all the information about the process. It is a data structure, which contains the following :

- Process State - It can be running, waiting etc.
- Process ID and parent process ID.
- CPU registers and Program Counter. **Program Counter** holds the address of the next instruction to be executed for that process.
- CPU Scheduling information - Such as priority information and pointers to scheduling queues.
- Memory Management information - Eg. page tables or segment tables.
- Accounting information - user and kernel CPU time consumed, account numbers, limits, etc.
- I/O Status information - Devices allocated, open file tables, etc.



Process Scheduling

The act of determining which process in the ready state should be moved to the running state is known as Process Scheduling.

The prime aim of the process scheduling system is to keep the CPU busy all the time and to deliver minimum response time for all programs. For achieving this, the scheduler must apply appropriate rules for swapping processes IN and OUT of CPU.

Schedulers fell into one of the two general categories :

- **Non pre-emptive scheduling.** When the currently executing process gives up the CPU voluntarily.
- **Pre-emptive scheduling.** When the operating system decides to favour another process, pre-empting the currently executing process.

Scheduling Queues

- All processes when enters into the system are stored in the **job queue**.
- Processes in the Ready state are placed in the **ready queue**.
- Processes waiting for a device to become available are placed in **device queues**.

There are unique device queues for each I/O device available.

Types of Schedulers

There are three types of schedulers available :

1. **Long Term Scheduler** :

   Long term scheduler runs less frequently. Long Term Schedulers decide which program must get into the job queue. From the job queue, the Job Processor, selects processes and loads them into the memory for execution. Primary aim of the Job Scheduler is to maintain a good degree of Multiprogramming. An optimal degree of Multiprogramming means the average rate of process creation is equal to the average departure rate of processes from the execution memory.

2. **Short Term Scheduler** :

   This is also known as CPU Scheduler and runs very frequently. The primary aim of this scheduler is to enhance CPU performance and increase process execution rate.

3. **Medium Term Scheduler** :

   During extra load, this scheduler picks out big processes from the ready queue for some time, to allow smaller processes to execute, thereby reducing the number of processes in the ready queue.

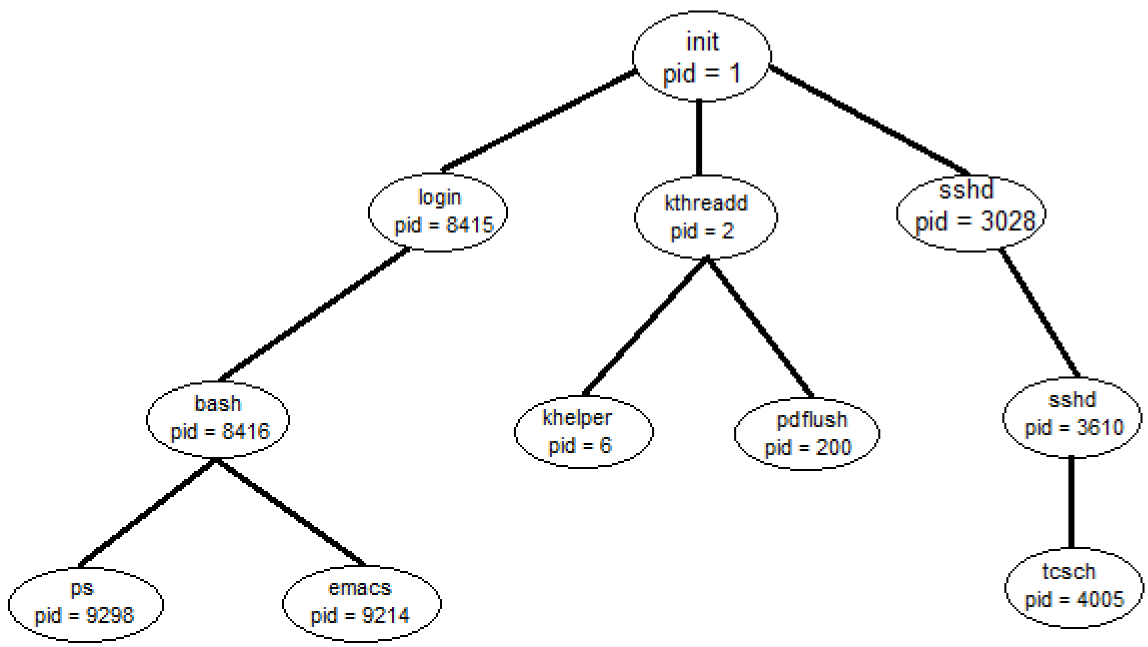
Operations on Process

Process Creation

Through appropriate system calls, such as fork or spawn, processes may create other processes. The process which creates other process, is termed the **parent** of the other process, while the created sub-process is termed its **child**.

Each process is given an integer identifier, termed as process identifier, or PID. The parent PID (PPID) is also stored for each process.

On a typical UNIX systems the process scheduler is termed as `sched`, and is given PID 0. The first thing done by it at system start-up time is to launch `init`, which gives that process PID 1. Further Init launches all the system daemons and user logins, and becomes the ultimate parent of all other processes.



A Tree of processes on a typical Linux system

A child process may receive some amount of shared resources with its parent depending on system implementation. To prevent runaway children from consuming all of a certain system resource, child processes may or may not be limited to a subset of the resources originally allocated to the parent.

There are two options for the parent process after creating the child :

- Wait for the child process to terminate before proceeding. Parent process makes a `wait()` system call, for either a specific child process or for any particular child process, which causes the parent process to block until the wait() returns. UNIX shells normally wait for their children to complete before issuing a new prompt.

- Run concurrently with the child, continuing to process without waiting. When a UNIX shell runs a process as a background task, this is the operation seen. It is also possible for the parent to run for a while, and then wait for the child later, which might occur in a sort of a parallel processing operation.

Process Termination

By making the `exit`(system call), typically returning an int, processes may request their own termination. This int is passed along to the parent if it is doing a `wait()`, and is typically zero on successful completion and some non-zero code in the event of any problem.

Processes may also be terminated by the system for a variety of reasons, including :

- The inability of the system to deliver the necessary system resources.
- In response to a KILL command or other unhandled process interrupts.
- A parent may kill its children if the task assigned to them is no longer needed i.e. if the need of having a child terminates.
- If the parent exits, the system may or may not allow the child to continue without a parent (In UNIX systems, orphaned processes are generally inherited by `init`, which then proceeds to kill them.)

When a process ends, all of its system resources are freed up, open files flushed and closed, etc. The process termination status and execution times are returned to the parent if the parent is waiting for the child to terminate, or eventually returned to init if the process already became an orphan.

The processes which are trying to terminate but cannot do so because their parent is not waiting for them are termed **zombies**. These are eventually inherited by init as orphans and killed off.

CPU Scheduling

CPU scheduling is a process which allows one process to use the CPU while the execution of another process is on hold(in waiting state) due to unavailability of any resource like I/O etc, thereby making full use of CPU. The aim of CPU scheduling is to make the system efficient, fast and fair.

Scheduling Criteria

There are many different criterias to check when considering the "best" scheduling algorithm :

- **CPU utilization**

  To make out the best use of CPU and not to waste any CPU cycle, CPU would be working most of the time(Ideally 100% of the time). Considering a real system, CPU usage should range from 40% (lightly loaded) to 90% (heavily loaded.)

- **Throughput**

  It is the total number of processes completed per unit time or rather say total amount of work done in a unit of time. This may range from 10/second to 1/hour depending on the specific processes.

- **Turnaround time**

  It is the amount of time taken to execute a particular process, i.e. The interval from time of submission of the process to the time of completion of the process(Wall clock time).

- **Waiting time**

  The sum of the periods spent waiting in the ready queue amount of time a process has been waiting in the ready queue to acquire get control on the CPU.

- **Load average**

  It is the average number of processes residing in the ready queue waiting for their turn to get into the CPU.

- **Response time**

  Amount of time it takes from when a request was submitted until the first response is produced. Remember, it is the time till the first response and not the completion of process execution(final response).

In general CPU utilization and Throughput are maximized and other factors are reduced for proper optimization.

Scheduling Algorithms

We'll discuss four major scheduling algorithms here which are following :

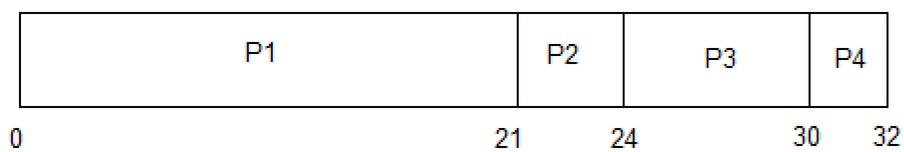
1. First Come First Serve(FCFS) Scheduling
2. Shortest-Job-First(SJF) Scheduling
3. Priority Scheduling
4. Round Robin(RR) Scheduling
5. Multilevel Queue Scheduling

First Come First Serve(FCFS) Scheduling

- Jobs are executed on first come, first serve basis.
- Easy to understand and implement.
- Poor in performance as average wait time is high.

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2

The average waiting time will be = (0 + 21 + 24 + 30)/4 = 18.75 ms



This is the GANTT chart for the above processes

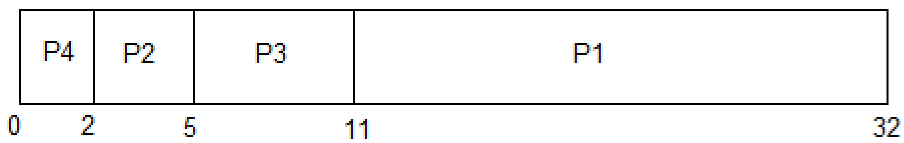
Shortest-Job-First(SJF) Scheduling

- Best approach to minimize waiting time.
- Actual time taken by the process is already known to processor.
- Impossible to implement.

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2

In Shortest Job First Scheduling, the shortest Process is executed first.

Hence the GANTT chart will be following :

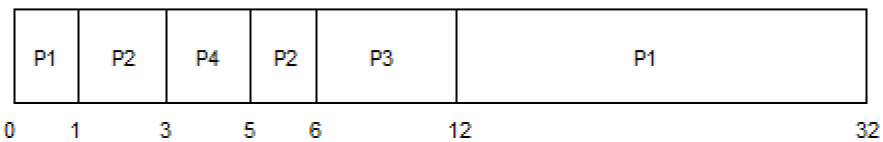


Now, the average waiting time will be = (0 + 2 + 5 + 11)/4 = 4.5 ms

In Preemptive Shortest Job First Scheduling, jobs are put into ready queue as they arrive, but as a process with short burst time arrives, the existing process is preempted.

PROCESS	BURST TIME	ARRIVAL TIME
P1	21	0
P2	3	1
P3	6	2
P4	2	3

The GANTT chart for Preemptive Shortest Job First Scheduling will be,



The average waiting time will be, ((5-3) + (6-2) + (12-1))/4 = 4.25 ms

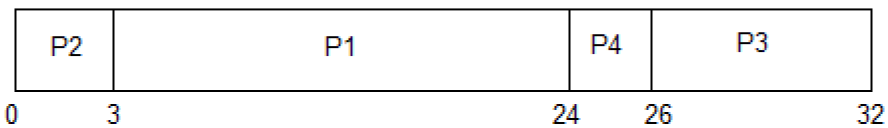
The average waiting time for preemptive shortest job first scheduling is less than both, non-preemptive SJF scheduling and FCFS scheduling.

Priority Scheduling

- Priority is assigned for each process.
- Process with highest priority is executed first and so on.
- Processes with same priority are executed in FCFS manner.
- Priority can be decided based on memory requirements, time requirements or any other resource requirement.

PROCESS	BURST TIME	PRIORITY
P1	21	2
P2	3	1
P3	6	4
P4	2	3

The GANTT chart for following processes based on Priority scheduling will be,



The average waiting time will be, (0 + 3 + 24 + 26)/4 = 13.25 ms

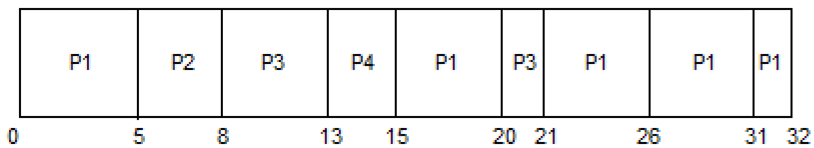
---

Round Robin(RR) Scheduling

- A fixed time is allotted to each process, called **quantum**, for execution.
- Once a process is executed for given time period that process is preempted and other process executes for given time period.
- Context switching is used to save states of preemptied processes.

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2

The GANTT chart for round robin scheduling will be,



The average waiting time will be, 11 ms.

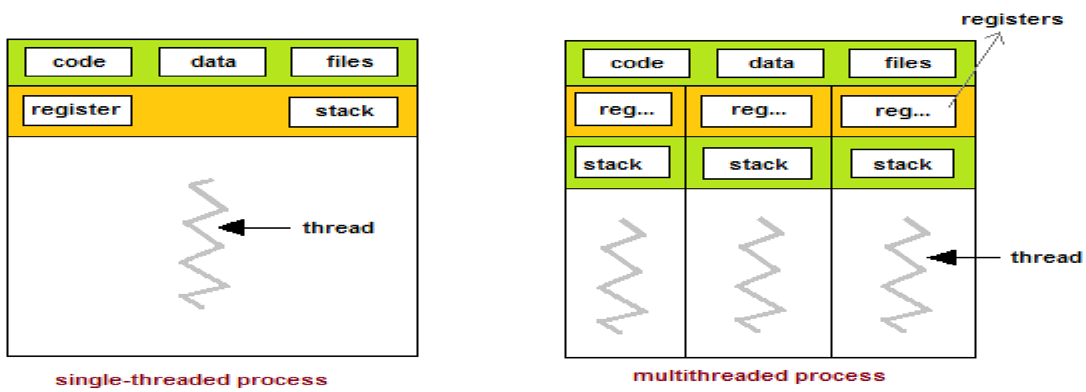
Multilevel Queue Scheduling

- Multiple queues are maintained for processes.
- Each queue can have its own scheduling algorithms.
- Priorities are assigned to each queue.

What are Threads?

Thread is an execution unit which consists of its own program counter, a stack, and a set of registers. Threads are also known as Lightweight processes. Threads are popular way to improve application through parallelism. The CPU switches rapidly back and forth among the threads giving illusion that the threads are running in parallel.

As each thread has its own independent resource for process execution, multpile processes can be executed parallely by increasing number of threads.



Types of Thread

There are two types of threads :

- User Threads
- Kernel Threads

User threads, are above the kernel and without kernel support. These are the threads that application programmers use in their programs.

Kernel threads are supported within the kernel of the OS itself. All modern OSs support kernel level threads, allowing the kernel to perform multiple simultaneous tasks and/or to service multiple kernel system calls simultaneously.

Multithreading Models

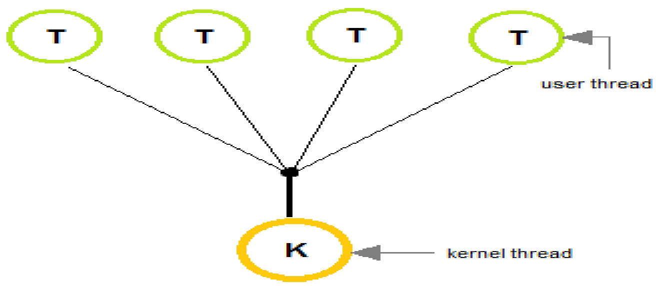
The user threads must be mapped to kernel threads, by one of the following strategies.

- Many-To-One Model
- One-To-One Model
- Many-To-Many Model

Many-To-One Model

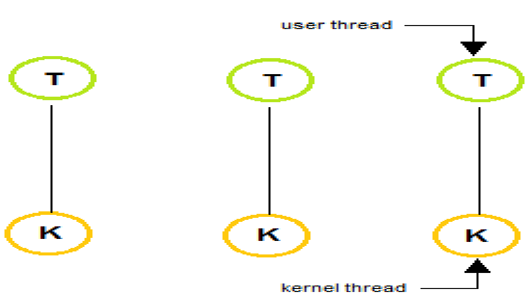
- In the many-to-one model, many user-level threads are all mapped onto a single kernel thread.

- Thread management is handled by the thread library in user space, which is efficient in nature.



One-To-One Model

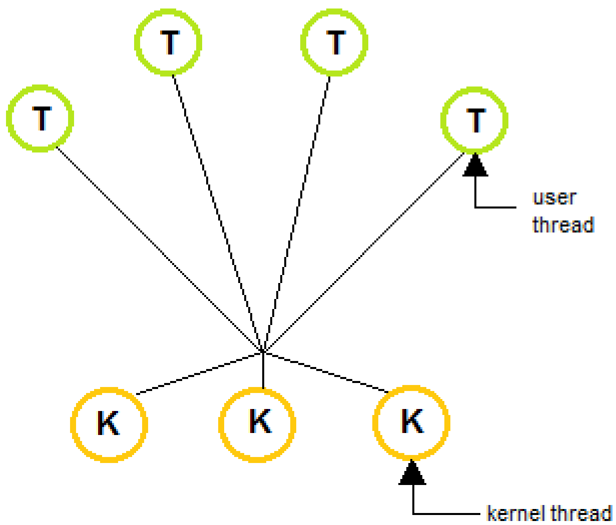
- The one-to-one model creates a separate kernel thread to handle each and every user thread.
- Most implementations of this model place a limit on how many threads can be created.
- Linux and Windows from 95 to XP implement the one-to-one model for threads.



Many-To-Many Model

- The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads, combining the best features of the one-to-one and many-to-one models.
- Users can create any number of the threads.

- Blocking the kernel system calls does not block the entire process.
- Processes can be split across multiple processors.



Thread Libraries

Thread libraries provides programmers with API for creating and managing of threads.

Thread libraries may be implemented either in user space or in kernel space. The user space involves API functions implemented solely within user space, with no kernel support. The kernel space involves system calls, and requires a kernel with thread library support.

There are three types of thread :

POSIX Pitheads, may be provided as either a user or kernel library, as an extension to the POSIX standard.

- Win32 threads, are provided as a kernel-level library on Windows systems.
- Java threads - Since Java generally runs on a Java Virtual Machine, the implementation of threads is based upon whatever OS and hardware the JVM is running on, i.e. either Pitheads or Win32 threads depending on the system

Benefits of Multithreading

1. Responsiveness
2. Resource sharing, hence allowing better utilization of resources.

3. Economy. Creating and managing threads becomes easier.
4. Scalability. One thread runs on one CPU. In Multithreaded processes, threads can be distributed over a series of processors to scale.
5. Context Switching is smooth. Context switching refers to the procedure followed by CPU to change from one task to another.

Multithreading Issues

1. **Thread Cancellation**.

   Thread cancellation means terminating a thread before it has finished working. There can be two approaches for this, one is **Asynchronous cancellation**, which terminates the target thread immediately. The other is **Deferred cancellation** allows the target thread to periodically check if it should be cancelled.

2. **Signal Handling**.

   Signals are used in UNIX systems to notify a process that a particular event has occurred. Now in when a Multithreaded process receives a signal, to which thread it must be delivered? It can be delivered to all, or a single thread.

3. **fork() System Call**.

   fork() is a system call executed in the kernel through which a process creates a copy of itself. Now the problem in Multithreaded process is, if one thread forks, will the entire process be copied or not?

4. **Security Issues** because of extensive sharing of resources between multiple threads.

There are many other issues that you might face in a multithreaded process, but there are appropriate solutions available for them. Pointing out some issues here was just to study both sides of the coin.

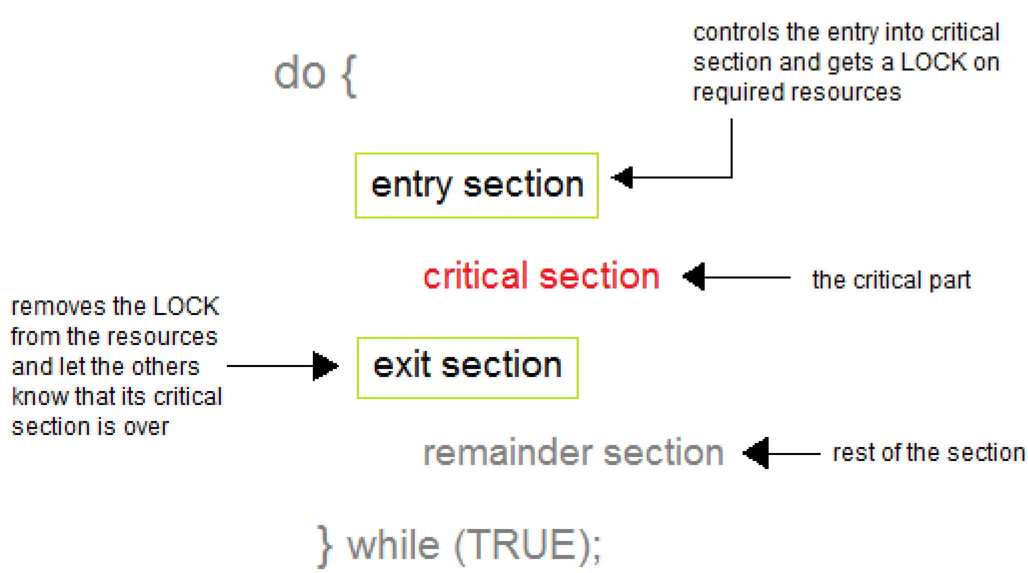
Process Synchronization

Process Synchronization means sharing system resources by processes in a such a way that, Concurrent access to shared data is handled thereby minimizing the chance of inconsistent data. Maintaining data consistency demands mechanisms to ensure synchronized execution of cooperating processes.

Process Synchronization was introduced to handle problems that arose while multiple process executions. Some of the problems are discussed below.

Critical Section Problem

A Critical Section is a code segment that accesses shared variables and has to be executed as an atomic action. It means that in a group of cooperating processes, at a given point of time, only one process must be executing its critical section. If any other process also wants to execute its critical section, it must wait until the first one finishes.



Solution to Critical Section Problem

A solution to the critical section problem must satisfy the following three conditions :

1. **Mutual Exclusion**

Out of a group of cooperating processes, only one process can be in its critical section at a given point of time.

2. **Progress**

   If no process is in its critical section, and if one or more threads want to execute their critical section then any one of these threads must be allowed to get into its critical section.

3. **Bounded Waiting**

   After a process makes a request for getting into its critical section, there is a limit for how many other processes can get into their critical section, before this process's request is granted. So after the limit is reached, system must grant the process permission to get into its critical section.

Synchronization Hardware

Many systems provide hardware support for critical section code. The critical section problem could be solved easily in a single-processor environment if we could disallow interrupts to occur while a shared variable or resource is being modified.

In this manner, we could be sure that the current sequence of instructions would be allowed to execute in order without pre-emption. Unfortunately, this solution is not feasible in a multiprocessor environment.

Disabling interrupt on a multiprocessor environment can be time consuming as the message is passed to all the processors.

This message transmission lag, delays entry of threads into critical section and the system efficiency decreases.

Mutex Locks

As the synchronization hardware solution is not easy to implement fro everyone, a strict software approach called Mutex Locks was introduced. In this approach, in the entry section of code, a LOCK is acquired over the critical resources modified and used inside critical section, and in the exit section that LOCK is released.

As the resource is locked while a process executes its critical section hence no other process can access it.

Semaphores

In 1965, Dijkstra proposed a new and very significant technique for managing concurrent processes by using the value of a simple integer variable to synchronize the progress of interacting processes. This integer variable is called **semaphore**. So it is basically a synchronizing tool and is accessed only through two low standard atomic operations, wait and signal designated by P() and V() respectively.

The classical definition of wait and signal are :

- Wait : decrement the value of its argument S as soon as it would become non-negative.
- Signal : increment the value of its argument, S as an individual operation.

Properties of Semaphores

1. Simple
2. Works with many processes
3. Can have many different critical sections with different semaphores
4. Each critical section has unique access semaphores
5. Can permit multiple processes into the critical section at once, if desirable.

Types of Semaphores

Semaphores are mainly of two types:

Binary Semaphore

It is a special form of semaphore used for implementing mutual exclusion, hence it is often called *Mutex*. A binary semaphore is initialized to 1 and only takes the value 0 and 1 during execution of a program.

Counting Semaphores

These are used to implement bounded concurrency.

Limitations of Semaphores

1. Priority Inversion is a big limitation os semaphores.
2. Their use is not enforced, but is by convention only.

3. With improper use, a process may block indefinitely. Such a situation is called Deadlock. We will be studying deadlocks in details in coming lessons.

Classical Problem of Synchronization

Following are some of the classical problem faced while process synchronaization in systems where cooperating processes are present.

Bounded Buffer Problem

- This problem is generalised in terms of the Producer-Consumer problem.
- Solution to this problem is, creating two counting semaphores "full" and "empty" to keep track of the current number of full and empty buffers respectively.

The Readers Writers Problem

- In this problem there are some processes(called readers) that only read the shared data, and never change it, and there are other processes(called writers) who may change the data in addition to reading or instead of reading it.
- There are various type of the readers-writers problem, most centred on relative priorities of readers and writers

---

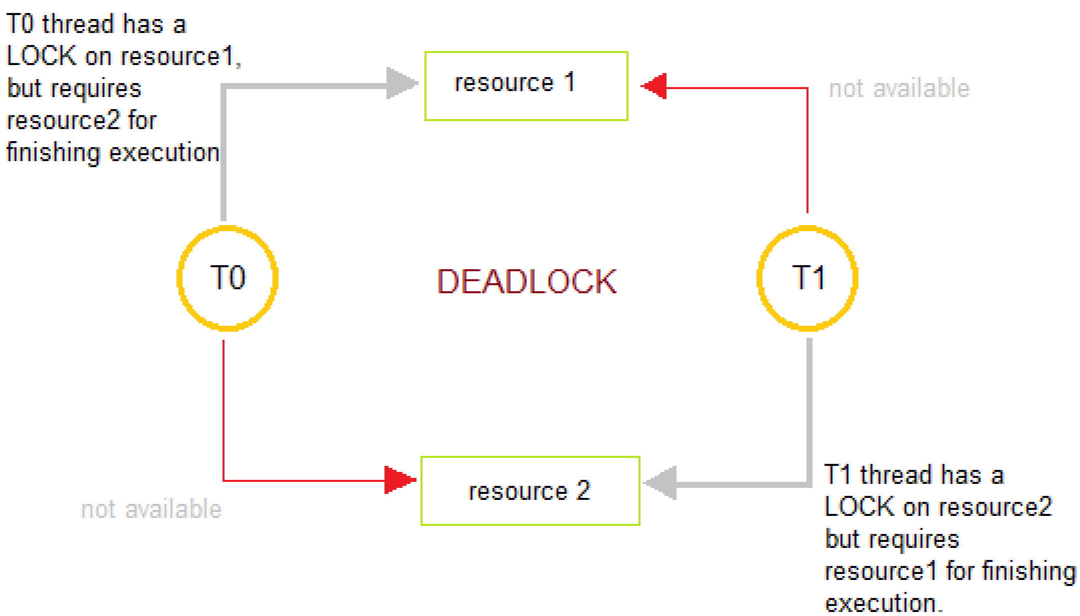
Dining Philosophers Problem

- The dining philosopher’s problem involves the allocation of limited resources from a group of processes in a deadlock-free and starvation-free manner.
- There are five philosophers sitting around a table, in which there are five chopsticks kept beside them and a bowl of rice in the centre, When a philosopher wants to eat, he uses two chopsticks -

one from their left and one from their right. When a philosopher wants to think, he keeps down both chopsticks at their original place.

What is a Deadlock?

Deadlocks are a set of blocked processes each holding a resource and waiting to acquire a resource held by another process.



How to avoid Deadlocks

Deadlocks can be avoided by avoiding at least one of the four conditions, because all this four conditions are required simultaneously to cause deadlock.

1. **Mutual Exclusion**

   Resources shared such as read-only files do not lead to deadlocks but resources, such as printers and tape drives, requires exclusive access by a single process.

2. **Hold and Wait**

   In this condition processes must be prevented from holding one or more resources while simultaneously waiting for one or more others.

3. **No Preemption**

   Preemption of process resource allocations can avoid the condition of deadlocks, where ever possible.

4. **Circular Wait**

   Circular wait can be avoided if we number all resources, and require that processes request resources only in strictly increasing(or decreasing) order.

Handling Deadlock

The above points focus on preventing deadlocks. But what to do once a deadlock has occured. Following three strategies can be used to remove deadlock after its occurrence.

1. **Preemption**

   We can take a resource from one process and give it to other. This will resolve the deadlock situation, but sometimes it does causes problems.

2. **Rollback**

   In situations where deadlock is a real possibility, the system can periodically make a record of the state of each process and when deadlock occurs, roll everything back to the last checkpoint, and restart, but allocating resources differently so that deadlock does not occur.

3. **Kill one or more processes**

   This is the simplest way, but it works.

What is a Livelock?

There is a variant of deadlock called livelock. This is a situation in which two or more processes continuously change their state in response to changes in the other process(es) without doing any useful work. This is similar to deadlock in that no progress is made but differs in that neither process is blocked or waiting for anything.

A human example of livelock would be two people who meet face-to-face in a corridor and each moves aside to let the other pass, but they end up swaying from side to side without making any progress because they always move the same way at the same time.

Memory Management

Main Memory refers to a physical memory that is the internal memory to the computer. The word main is used to distinguish it from external mass storage devices such as disk drives. Main memory is also known as RAM. The computer is able to change only data that is in main memory. Therefore, every program we execute and every file we access must be copied from a storage device into main memory.

All the programs are loaded in the main memeory for execution. Sometimes complete program is loaded into the memory, but some times a certain part or routine of the program is loaded into the main memory only when it is called by the program, this mechanism is called **Dynamic Loading**, this enhance the performance.

Also, at times one program is dependent on some other program. In such a case, rather than loading all the dependent programs, CPU links the dependent programs to the main executing program when its required. This mechanism is known as **Dynamic Linking**.

Swapping

A process needs to be in memory for execution. But sometimes there is not enough main memory to hold all the currently active processes in a timesharing system. So, excess process are kept on disk and brought in to run dynamically. Swapping is the process of bringing in each process in main memory, running it for a while and then putting it back to the disk.

Contiguous Memory Allocation

In contiguous memory allocation each process is contained in a single contiguous block of memory. Memory is divided into several fixed size partitions. Each partition contains exactly one process. When a partition is free, a process is selected from the input queue and loaded into it. The free blocks of memory are known as *holes*. The set of holes is searched to determine which hole is best to allocate.

Memory Protection

Memory protection is a phenomenon by which we control memory access rights on a computer. The main aim of it is to prevent a process from accessing memory that has not been allocated to it. Hence prevents a bug within a process from affecting other processes, or the operating system itself, and instead results in a segmentation fault or storage violation exception being sent to the disturbing process, generally killing of process.

Memory Allocation

Memory allocation is a process by which computer programs are assigned memory or space. It is of three types :

First Fit: The first hole that is big enough is allocated to program.
Best Fit: The smallest hole that is big enough is allocated to program.
Worst Fit: The largest hole that is big enough is allocated to program.

Fragmentation

Fragmentation occurs in a dynamic memory allocation system when most of the free blocks are too small to satisfy any request. It is generally termed as inability to use the available memory.

In such situation processes are loaded and removed from the memory. As a result of this, free holes exists to satisfy a request but is non contiguous i.e. the memory is fragmented into large no. Of small holes. This phenomenon is known as **External Fragmentation.**

Also, at times the physical memory is broken into fixed size blocks and memory is allocated in unit of block sizes. The memory allocated to a space may be slightly larger than the requested memory. The difference between allocated and required memory is known as **Internal fragmentation** i.e. the memory that is internal to a partition but is of no use.

Paging

A solution to fragmentation problem is Paging. Paging is a memory management mechanism that allows the physical address space of a process to be non-contagious. Here physical memory is divided into blocks of equal size called **Pages**. The pages belonging to a certain process are loaded into available memory frames.

Page Table

A Page Table is the data structure used by a virtual memory system in a computer operating system to store the mapping between *virtual address* and *physical addresses.*

Virtual address is also known as Logical address and is generated by the CPU. While Physical address is the address that actually exists on memory.

Segmentation

Segmentation is another memory management scheme that supports the user-view of memory. Segmentation allows breaking of the virtual address space of a single process into segments that may be placed in non-contiguous areas of physical memory.

Segmentation with Paging

Both paging and segmentation have their advantages and disadvantages, it is better to combine these two schemes to improve on each. The combined scheme is known as 'Page the Elements'. Each segment in this scheme is divided into pages and each segment is maintained in a page table. So the logical address is divided into following 3 parts :

- Segment numbers(S)
- Page number (P)
- The displacement or offset number (D)

Virtual Memory

Virtual Memory is a space where large programs can store themselves in form of pages while their execution and only the required pages or portions of processes are loaded into the main memory.

This technique is useful as large virtual memory is provided for user programs when a very small physical memory is there.

In real scenarios, most processes never need all their pages at once, for following reasons :

- Error handling code is not needed unless that specific error occurs, some of which are quite rare.
- Arrays are often over-sized for worst-case scenarios, and only a small fraction of the arrays are actually used in practice.
- Certain features of certain programs are rarely used.

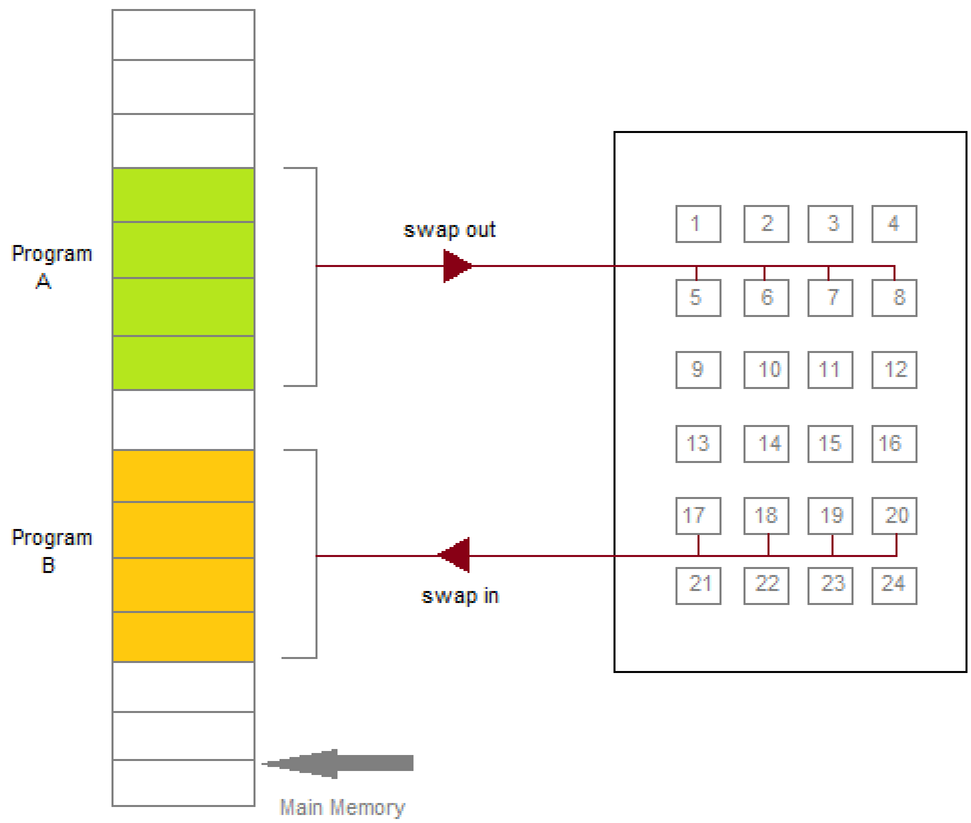
Benefits of having Virtual Memory :

Large programs can be written, as virtual space available is huge compared to physical memory.

1. Less I/O required, leads to faster and easy swapping of processes.
2. More physical memory available, as programs are stored on virtual memory, so they occupy very less space on actual physical memory.

Demand Paging

The basic idea behind demand paging is that when a process is swapped in, its pages are not swapped in all at once. Rather they are swapped in only when the process needs them(On demand). This is termed as lazy swapper, although a pager is a more accurate term.



Initially only those pages are loaded which will be required the process immediately.

The pages that are not moved into the memory, are marked as invalid in the page table. For an invalid entry the rest of the table is empty. In case of pages that are loaded in the memory, they are marked as valid along with the information about where to find the swapped out page.

When the process requires any of the page that is not loaded into the memory, a page fault trap is triggered and following steps are followed,

1. The memory address which is requested by the process is first checked, to verify the request made by the process.
2. If its found to be invalid, the process is terminated.
3. In case the request by the process is valid, a free frame is located, possibly from a free-frame list, where the required page will be moved.
4. A new operation is scheduled to move the necessary page from disk to the specified memory location. (This will usually block the process on an I/O wait, allowing some other process to use the CPU in the meantime.)
5. When the I/O operation is complete, the process's page table is updated with the new frame

number, and the invalid bit is changed to valid.

6. The instruction that caused the page fault must now be restarted from the beginning.

There are cases when no pages are loaded into the memory initially, pages are only loaded when demanded by the process by generating page faults. This is called **Pure Demand Paging**.

The only major issue with Demand Paging is, after a new page is loaded, the process starts execution from the beginning. Its is not a big issue for small programs, but for larger programs it affects performance drastically.

Page Replacement

As studied in Demand Paging, only certain pages of a process are loaded initially into the memory. This allows us to get more number of processes into the memory at the same time. but what happens when a process requests for more pages and no free memory is available to bring them in. Following steps can be taken to deal with this problem :

1. Put the process in the wait queue, until any other process finishes its execution thereby freeing frames.
2. Or, remove some other process completely from the memory to free frames.
3. Or, find some pages that are not being used right now, move them to the disk to get free frames. This technique is called **Page replacement** and is most commonly used. We have some great algorithms to carry on page replacement efficiently.

Basic Page Replacement

- Find the location of the page requested by ongoing process on the disk.
- Find a free frame. If there is a free frame, use it. If there is no free frame, use a page-replacement algorithm to select any existing frame to be replaced, such frame is known as **victim frame**.
- Write the victim frame to disk. Change all related page tables to indicate that this page is no longer in memory.
- Move the required page and store it in the frame. Adjust all related page and frame tables to indicate the change.
- Restart the process that was waiting for this page.

FIFO Page Replacement

A very simple way of Page replacement is FIFO (First in First Out)

- As new pages are requested and are swapped in, they are added to tail of a queue and the page which is at the head becomes the victim.
- Its not an effective way of page replacement but can be used for small systems.

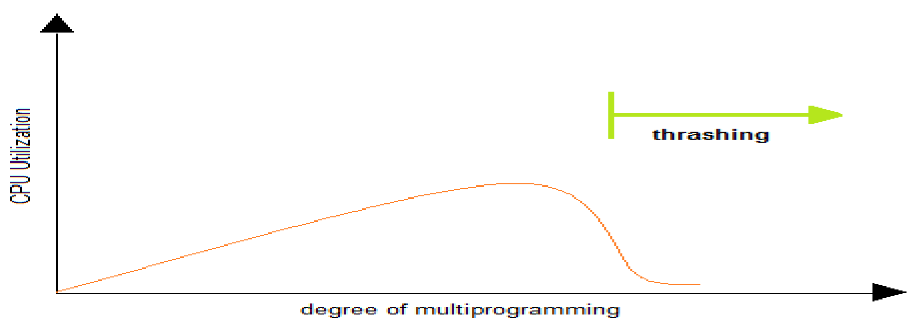
LRU Page Replacement

Below is a video, which will explain LRU Page replacement algorithm in details with an example.

Thrashing

A process that is spending more time paging than executing is said to be thrashing. In other words it means, that the process doesn't have enough frames to hold all the pages for its execution, so it is swapping pages in and out very frequently to keep executing. Sometimes, the pages which will be required in the near future have to be swapped out.

Initially when the CPU utilization is low, the process scheduling mechanism, to increase the level of multiprogramming loads multiple processes into the memory at the same time, allocating a limited amount of frames to each process. As the memory fills up, process starts to spend a lot of time for the required pages to be swapped in, again leading to low CPU utilization because most of the proccesses are waiting for pages. Hence the scheduler loads more processes to increase CPU utilization, as this continues at a point of time the complete system comes to a stop.



To prevent thrashing we must provide processes with as many frames as they really need "right now".

File System

A file can be "free formed", indexed or structured collection of related bytes having meaning only to the one who created it. Or in other words an entry in a directory is the file. The file may have attributes like name, creator, date, type, permissions etc.

File Structure

A file has various kinds of structure. Some of them can be :

- **Simple Record Structure** with lines of fixed or variable lengths.
- **Complex Structures** like formatted document or reloadable load files.
- **No Definite Structure** like sequence of words and bytes etc.

Attributes of a File

Following are some of the attributes of a file :

- **Name** . It is the only information which is in human-readable form.
- **Identifier**. The file is identified by a unique tag(number) within file system.
- **Type**. It is needed for systems that support different types of files.
- **Location**. Pointer to file location on device.
- **Size**. The current size of the file.
- **Protection**. This controls and assigns the power of reading, writing, executing.
- **Time, date, and user identification**. This is the data for protection, security, and usage monitoring.

File Access Methods

The way that files are accessed and read into memory is determined by Access methods. Usually a single access method is supported by systems while there are OS's that support multiple access methods.

Sequential Access

- Data is accessed one record right after another is an order.
- Read command cause a pointer to be moved ahead by one.
- Write command allocate space for the record and move the pointer to the new End

Of File.

- Such a method is reasonable for tape.

Direct Access

- This method is useful for disks.
- The file is viewed as a numbered sequence of blocks or records.
- There are no restrictions on which blocks are read/written, it can be dobe in any order.
- User now says "read n" rather than "read next".
- "n" is a number relative to the beginning of file, not relative to an absolute physical disk location.

Indexed Sequential Access

- It is built on top of Sequential access.
- It uses an Index to control the pointer while accessing files.

What is a Directory?

Information about files is maintained by Directories. A directory can contain multiple files. It can even have directories inside of them. In Windows we also call these directories as folders.

Following is the information maintained in a directory :

- **Name** : The name visible to user.
- **Type** : Type of the directory.
- **Location** : Device and location on the device where the file header is located.
- **Size** : Number of bytes/words/blocks in the file.
- **Position** : Current next-read/next-write pointers.
- **Protection** : Access control on read/write/execute/delete.
- **Usage** : Time of creation, access, modification etc.

- **Mounting** : When the root of one file system is "grafted" into the existing tree of another file system its called Mounting.

Banker's Algorithm

Banker's algorithm is a deadlock avoidance algorithm. It is named so because this algorithm is used in banking systems to determine whether a loan can be granted or not.

Consider there are n account holders in a bank and the sum of the money in all of their accounts is S. Everytime a loan has to be granted by the bank, it subtracts the loan amount from the total money the bank has. Then it checks if that difference is greater than S. It is done because, only then, the bank would have enough money even if all the n account holders draw all their money at once.

Banker's algorithm works in a similar way in computers. Whenever a new process is created, it must exactly specify the maximum instances of each resource type that it needs.

Let us assume that there are **n** processes and **m** resource types. Some data structures are used to implement the banker's algorithm. They are:

- **Available:** It is an array of length **m**. It represents the number of available resources of each type. If **Available[j] = k**, then there are **k** instances available, of resource type **Rj**.
- **Max:** It is an **n x m** matrix which represents the maximum number of instances of each resource that a process can request. If **Max[i][j] = k**, then the process **Pi** can request atmost **k** instances of resource type **Rj**.
- **Allocation:** It is an **n x m** matrix which represents the number of resources of each type currently allocated to each process. If **Allocation[i][j] = k**, then process **Pi** is currently allocated **k** instances of resource type **Rj**.
- **Need:** It is an **n x m** matrix which indicates the remaining resource needs of each process. If **Need[i][j] = k**, then process **Pi** may need **k** more instances of resource type **Rj** to complete its task.

```
Need[i][j] = Max[i][j] - Allocation [i][j]
```

Resource Request Algorithm:

This describes the behavior of the system when a process makes a resource request in the form of a request matrix. The steps are:

1. If number of requested instances of each resource is less than the need (which was declared previously by the process), go to step 2.
2. If number of requested instances of each resource type is less than the available resources of each type, go to step 3. If not, the process has to wait because sufficient resources are not available yet.
3. Now, assume that the resources have been allocated. Accordingly do,

```
Available = Available - Requesti
Allocationi = Allocationi + Requesti
Needi = Needi - Requesti
```

This step is done because the system needs to assume that resources have been allocated. So there will be less resources available after allocation. The number of allocated instances will increase. The need of the resources by the process will reduce. That's what is represented by the above three operations.

After completing the above three steps, check if the system is in safe state by applying the safety algorithm. If it is in safe state, proceed to allocate the requested resources. Else, the process has to wait longer.

Safety Algorithm:

1. Let Work and Finish be vectors of length **m** and **n**, respectively. Initially,

```
2.  Work = Available
3.  Finish[i] =false for i = 0, 1, ... , n - 1.
```

   This means, initially, no process has finished and the number of available resources is represented by the **Available** array.

4. Find an index **i** such that both

```
5.  Finish[i] ==false
6.  Needi <= Work
```

   If there is no such i present, then proceed to step 4.

It means, we need to find an unfinished process whose need can be satisfied by the available resources. If no such process exists, just go to step 4.

7. Perform the following:

```
8.  Work = Work + Allocation;
9.  Finish[i] = true;
```

Go to step 2.

When an unfinished process is found, then the resources are allocated and the process is marked finished. And then, the loop is repeated to check the same for all other processes.

10. If **Finish[i] == true** for all i, then the system is in a safe state.

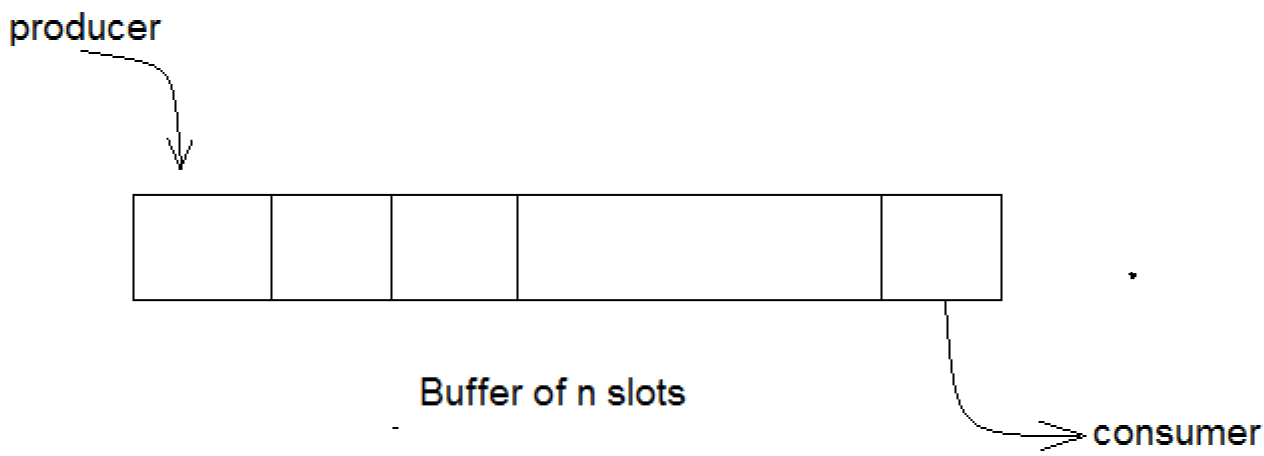
That means if all processes are finished, then the system is in safe state.

Bounded Buffer Problem

Bounded buffer problem, which is also called **producer consumer problem**, is one of the classic problems of synchronization.

Problem Statement:

There is a buffer of **n** slots and each slot is capable of storing one unit of data. There are two processes running, namely, **producer** and **consumer**, which are operating on the buffer.



Bounded Buffer Problem

A producer tries to insert data into an empty slot of the buffer. A consumer tries to remove data from a filled slot in the buffer. As you might have guessed by now, those two processes won't produce the expected output if they are being executed concurrently.

There needs to be a way to make the producer and consumer work in an independent manner.

Solution:

One solution of this problem is to use semaphores. The semaphores which will be used here are:

- **m**, a binary semaphore which is used to acquire and release the lock.
- **empty**, a counting semaphore whose initial value is the number of slots in the buffer, since, initially all slots are empty.
- **full**, a counting semaphore whose initial value is 0.

At any instant, the current value of empty represents the number of empty slots in the buffer and full represents the number of occupied slots in the buffer.

Producer Operation:

The pseudocode of the producer function looks like this:

```
do {
 wait(empty); // wait until empty>0 and then decrement 'empty'
 wait(mutex); // acquire lock
 /* perform the
 insert operation in a slot */
 signal(mutex); // release lock
 signal(full);  // increment 'full'
} while(TRUE)
```

- Looking at the above code for a producer, we can see that a producer first waits until there is atleast one empty slot.
- Then it decrements the **empty** semaphore because, there will now be one less empty slot, since the producer is going to insert data in one of those slots.
- Then, it acquires lock on the buffer, so that the consumer cannot access the buffer until producer completes its operation.
- After performing the insert operation, the lock is released and the value of **full** is incremented because the producer has just filled a slot in the buffer.

Consumer Operation:

The pseudocode of the consumer function looks like this:

```
do {
 wait(full); // wait until full>0 and then decrement 'full'
 wait(mutex);  // acquire the lock
 /* perform the remove operation
  in a slot */
 signal(mutex); // release the lock
 signal(empty); // increment 'empty'
} while(TRUE);
```

- The consumer waits until there is atleast one full slot in the buffer.
- Then it decrements the **full** semaphore because the number of occupied slots will be

decreased by one, after the consumer completes its operation.
- After that, the consumer acquires lock on the buffer.
- Following that, the consumer completes the removal operation so that the data from one of the full slots is removed.
- Then, the consumer releases the lock.
- Finally, the **empty** semaphore is incremented by 1, because the consumer has just removed data from an occupied slot, thus making it empty.

Comparison of Scheduling Algorithms

By now, you must have understood how CPU can apply different scheduling algorithms to schedule processes. Now, let us examine the advantages and disadvantages of each scheduling algorithm.

First Come First Serve (FCFS)

Advantages:

- FCFS algorithm doesn’t include any complex logic, it just puts the process requests in a queue and executes it one by one.
- Hence, FCFS is pretty simple and easy to implement.
- Eventually, every process will get a chance to run, so starvation doesn’t occur.

Disadvantages:

- There is no option for pre-emption of a process. If a process is started, then CPU executes the process until it ends.
- Because there is no pre-emption, if a process executes for a long time, the processes in the back of the queue will have to wait for a long time before they get a chance to be executed.

Shortest Job First (SJF)

Advantages:

- According to the definition, short processes are executed first and then followed by longer processes.
- The throughput is increased because more processes can be executed in less amount of time.

Disadvantages:

- The time taken by a process must be known by the CPU beforehand, which is not possible.
- Longer processes will have more waiting time, eventually they'll suffer starvation.

Note: Preemptive Shortest Job First scheduling will have the same advantages and disadvantages as those for SJF.

Round Robin (RR)

Advantages:

- Each process is served by the CPU for a fixed time quantum, so all processes are given the same priority.
- Starvation doesn't occur because for each round robin cycle, every process is given a fixed time to execute. No process is left behind.

Disadvantages:

- The throughput in RR largely depends on the choice of the length of the time quantum. If time quantum is longer than needed, it tends to exhibit the same behavior as FCFS.
- If time quantum is shorter than needed, the number of times that CPU switches from one process to another process, increases. This leads to decrease in CPU efficiency.

Priority based Scheduling

Advantages:

- The priority of a process can be selected based on memory requirement, time requirement or user preference. For example, a high end game will have better graphics, that means the process which updates the screen in a game will have higher priority so as to achieve better graphics performance.

Disadvantages:

- A second scheduling algorithm is required to schedule the processes which have same priority.
- In preemptive priority scheduling, a higher priority process can execute ahead of an already executing lower priority process. If lower priority process keeps waiting for higher priority processes, starvation occurs.

Usage of Scheduling Algorithms in Different Situations:

Every scheduling algorithm has a type of a situation where it is the best choice. Let's look at different such situations:

- **Situation 1:** The incoming processes are short and there is no need for the processes to execute in a specific order.

  In this case, FCFS works best when compared to SJF and RR because the processes are short which means that no process will wait for a longer time. When each process is executed one by one, every process will be executed eventually.

- **Situation 2:** The processes are a mix of long and short processes and the task will only be completed if all the processes are executed successfully in a given time.

Round Robin scheduling works efficiently here because it does not cause starvation and also gives equal time quantum for each process.

- **Situation 3:** The processes are a mix of user based and kernel based processes.

  Priority based scheduling works efficiently in this case because generally kernel based processes have higher priority when compared to user based processes.

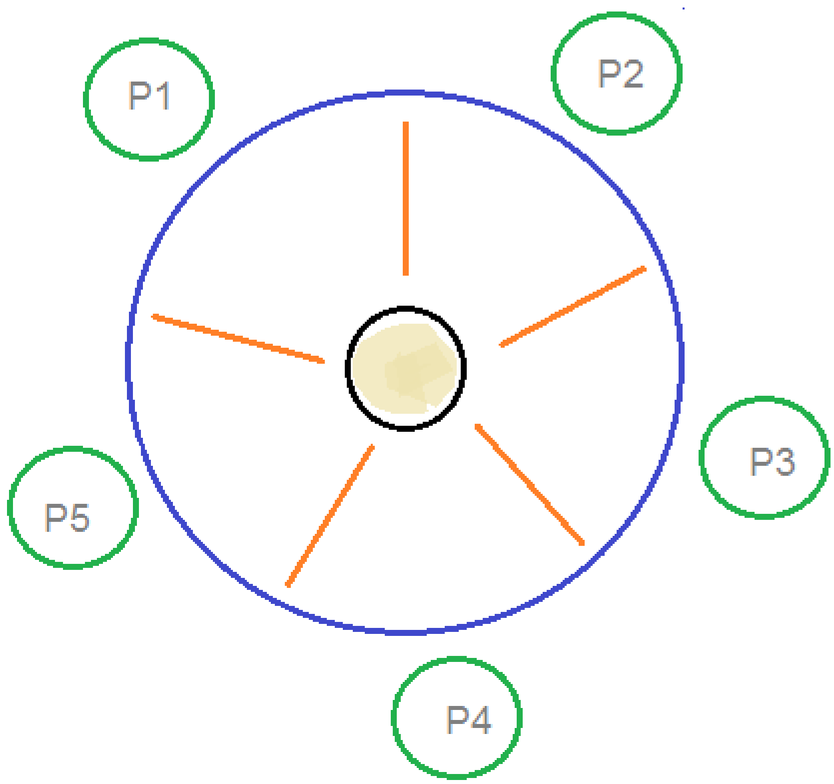
  For example, the scheduler itself is a kernel based process, it should run first so that it can schedule other processes.

Dining Philosophers Problem

The dining philosophers problem is another classic synchronization problem which is used to evaluate situations where there is a need of allocating multiple resources to multiple processes.

Problem Statement:

Consider there are five philosophers sitting around a circular dining table. The dining table has five chopsticks and a bowl of rice in the middle as shown in the below figure.



Dining Philosophers Problem

At any instant, a philosopher is either eating or thinking. When a philosopher wants to eat, he uses two chopsticks - one from their left and one from their right. When a philosopher wants to think, he keeps down both chopsticks at their original place.

Solution:

From the problem statement, it is clear that a philosopher can think for an indefinite amount of time. But when a philosopher starts eating, he has to stop at some point of time. The philosopher is in an endless cycle of thinking and eating.

An array of five semaphores, **stick[5]**, for each of the five chopsticks.

The code for each philosopher looks like:

```
while(TRUE) {
wait(stick[i]);
wait(stick[(i+1) % 5]);  // mod is used because if i=5, next
                // chopstick is 1 (dining table is circular)
/* eat */
```

```
signal(stick[i]);
signal(stick[(i+1) % 5]);
/* think */
}
```

When a philosopher wants to eat the rice, he will wait for the chopstick at his left and picks up that chopstick. Then he waits for the right chopstick to be available, and then picks it too. After eating, he puts both the chopsticks down.

But if all five philosophers are hungry simultaneously, and each of them pickup one chopstick, then a deadlock situation occurs because they will be waiting for another chopstick forever. The possible solutions for this are:

- A philosopher must be allowed to pick up the chopsticks only if both the left and right chopsticks are available.
- Allow only four philosophers to sit at the table. That way, if all the four philosophers pick up four chopsticks, there will be one chopstick left on the table. So, one philosopher can start eating and eventually, two chopsticks will be available. In this way, deadlocks can be avoided.

Readers Writer Problem

Readers writer problem is another example of a classic synchronization problem. There are many variants of this problem, one of which is examined below.

Problem Statement:

There is a shared resource which should be accessed by multiple processes. There are two types of processes in this context. They are **reader** and **writer**. Any number of **readers** can read from the shared resource simultaneously, but only one **writer** can write to the shared resource. When a **writer** is writing data to the resource, no other process can access the resource. A **writer** cannot write to the resource if there are non zero number of readers accessing the resource.

Solution:

From the above problem statement, it is evident that readers have higher priority than writer. If a writer wants to write to the resource, it must wait until there are no readers currently accessing that resource.

Here, we use one mutex **m** and a semaphore **w**. An integer variable **read_count** is used to maintain the number of readers currently accessing the resource. The variable **read_count** is initialized to 0. A value of 1 is given initially to **m** and **w**.

Instead of having the process to acquire lock on the shared resource, we use the mutex **m** to make the process to acquire and release lock whenever it is updating the **read_count** variable.

The code for the writer process looks like this:

```
while(TRUE) {
  wait(w);
  /*perform the
write operation */
  signal(w);
}
```

The code for the reader process looks like this:

```
while(TRUE) {
  wait(m);   //acquire lock
  read_count++;
  if(read_count == 1)
       wait(w);
  signal(m);  //release lock
  /* perform the
    reading operation */
  wait(m);   // acquire lock
  read_count--;
  if(read_count == 0)
       signal(w);
  signal(m);  // release lock
}
```

Code Explained:

- As seen above in the code for the writer, the writer just waits on the **w** semaphore until it gets a chance to write to the resource.
- After performing the write operation, it increments **w** so that the next writer can access the resource.
- On the other hand, in the code for the reader, the lock is acquired whenever the **read_count** is updated by a process.

- When a reader wants to access the resource, first it increments the **read_count** value, then accesses the resource and then decrements the **read_count** value.
- The semaphore **w** is used by the first reader which enters the critical section and the last reader which exits the critical section.
- The reason for this is, when the first readers enters the critical section, the writer is blocked from the resource. Only new readers can access the resource now.

- Similarly, when the last reader exits the critical section, it signals the writer using the **w** semaphore because there are zero readers now and a writer can have the chance to access the resource.

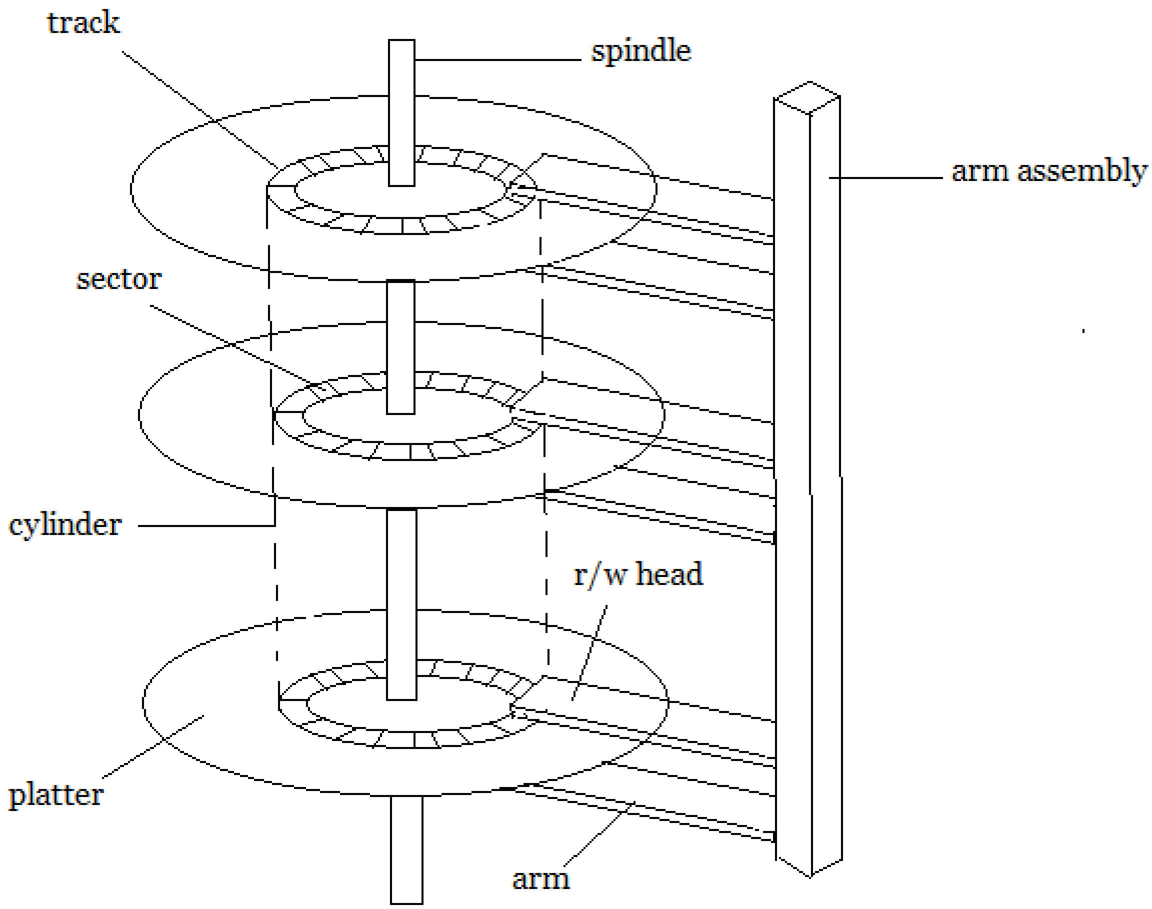
Secondary Storage Structure

Secondary storage devices are those devices whose memory is non volatile, meaning, the stored data will be intact even if the system is turned off. Here are a few things worth noting about secondary storage.

- Secondary storage is also called auxiliary storage.
- Secondary storage is less expensive when compared to primary memory like RAMs.
- The speed of the secondary storage is also lesser than that of primary storage.
- Hence, the data which is less frequently accessed is kept in the secondary storage.
- A few examples are magnetic disks, magnetic tapes, removable thumb drives etc.

Magnetic Disk Structure

In modern computers, most of the secondary storage is in the form of magnetic disks. Hence, knowing the structure of a magnetic disk is necessary to understand how the data in the disk is accessed by the computer.



Structure of a magnetic disk

A magnetic disk contains several **platters**. Each platter is divided into circular shaped **tracks**. The length of the tracks near the centre is less than the length of the tracks farther from the centre. Each track is further divided into **sectors**, as shown in the figure.

Tracks of the same distance from centre form a cylinder. A read-write head is used to read data from a sector of the magnetic disk.

The speed of the disk is measured as two parts:

- **Transfer rate:** This is the rate at which the data moves from disk to the computer.
- **Random access time:** It is the sum of the seek time and rotational latency.

Seek time is the time taken by the arm to move to the required track. **Rotational latency** is defined as the time taken by the arm to reach the required sector in the track.

Even though the disk is arranged as sectors and tracks physically, the data is logically arranged and addressed as an array of blocks of fixed size. The size of a block can be **512** or **1024** bytes. Each logical block is mapped with a sector on the disk, sequentially. In this way, each sector in the disk will have a logical address.

Disk Scheduling Algorithms

On a typical multiprogramming system, there will usually be multiple disk access requests at any point of time. So those requests must be scheduled to achieve good efficiency. Disk scheduling is similar to process scheduling. Some of the disk scheduling algorithms are described below.

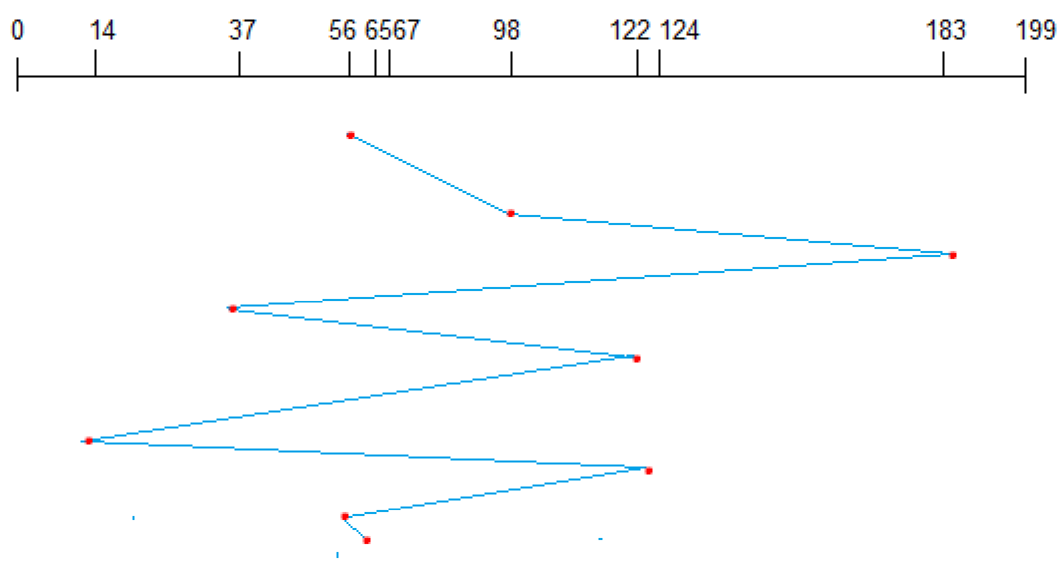
First Come First Serve:

This algorithm performs requests in the same order asked by the system. Let's take an example where the queue has the following requests with cylinder numbers as follows:

98, 183, 37, 122, 14, 124, 65, 67

Assume the head is initially at cylinder **56**. The head moves in the given order in the queue i.e., **56 ->98->183->….->67**.

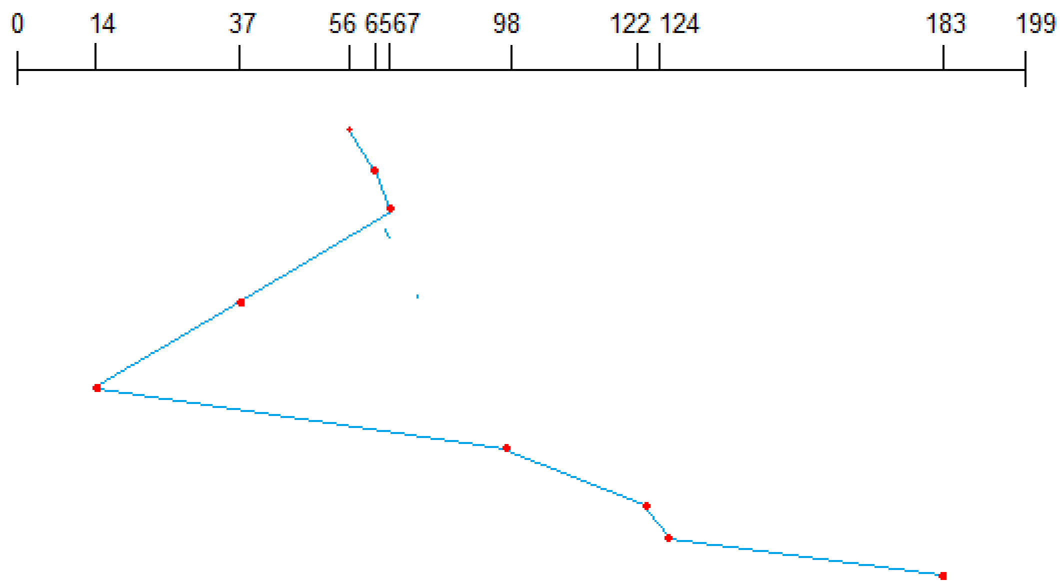
- Assume the head is initially at cylinder **56**. The head moves in the given order in the queue i.e., **56->98->183->….->67**.



-
- **FCFS Disk Scheduling**
-

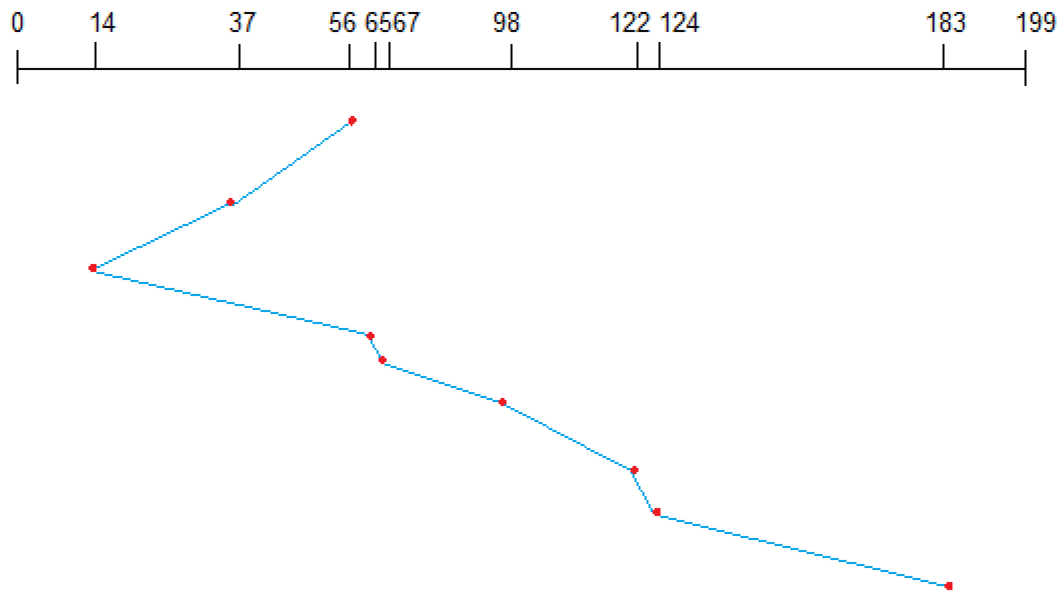
- ## Shortest Seek Time First (SSTF):

- Here the position which is closest to the current head position is chosen first. Consider the previous example where disk queue looks like,
- **98, 183, 37, 122, 14, 124, 65, 67**
- Assume the head is initially at cylinder **56**. The next closest cylinder to **56** is **65**, and then the next nearest one is **67**, then **37**, **14**, so on.



-
- **SSTF Disk Scheduling**
-

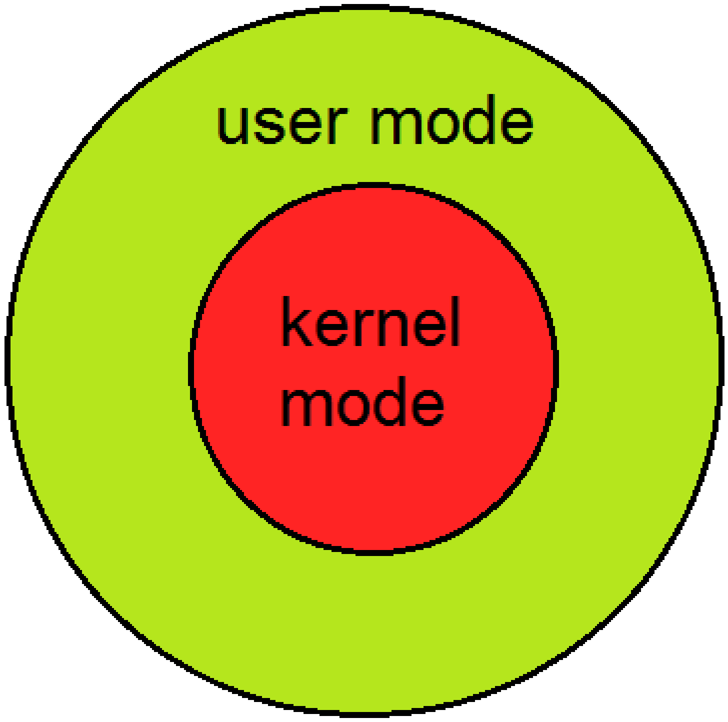
- ## SCAN algorithm:
- This algorithm is also called the elevator algorithm because of it's behavior. Here, first the head moves in a direction (say backward) and covers all the requests in the path. Then it moves in the opposite direction and covers the remaining requests in the path. This behavior is similar to that of an elevator. Let's take the previous example,
- **98, 183, 37, 122, 14, 124, 65, 67**
- Assume the head is initially at cylinder **56**. The head moves in backward direction and accesses **37** and **14**. Then it goes in the opposite direction and accesses the cylinders as they come in the path.



-
- **SCAN algorithm**

System Calls

To understand system calls, first one needs to understand the difference between **kernel mode** and **user mode** of a CPU. Every modern operating system supports these two modes.



Modes supported by the operating system

Kernel Mode

- When CPU is in **kernel mode**, the code being executed can access any memory address and any hardware resource.
- Hence kernel mode is a very privileged and powerful mode.
- If a program crashes in kernel mode, the entire system will be halted.

User Mode

- When CPU is in **user mode**, the programs don't have direct access to memory and hardware resources.
- In user mode, if any program crashes, only that particular program is halted.
- That means the system will be in a safe state even if a program in user mode crashes.
- Hence, most programs in an OS run in user mode.

System Call

When a program in user mode requires access to RAM or a hardware resource, it must ask the kernel to provide access to that resource. This is done via something called a **system call**.

When a program makes a system call, the mode is switched from user mode to kernel mode. This is called a **context switch**.

Then the kernel provides the resource which the program requested. After that, another context switch happens which results in change of mode from kernel mode back to user mode.

Generally, system calls are made by the user level programs in the following situations:

- Creating, opening, closing and deleting files in the file system.
- Creating and managing new processes.
- Creating a connection in the network, sending and receiving packets.
- Requesting access to a hardware device, like a mouse or a printer.

In a typical UNIX system, there are around 300 system calls. Some of them which are important ones in this context, are described below.

Fork()

The fork() system call is used to create processes. When a process (a program in execution) makes a fork() call, an exact copy of the process is created. Now there are two processes, one being the **parent** process and the other being the **child** process.

The process which called the **fork()** call is the **parent** process and the process which is created newly is called the **child** process. The child process will be exactly the same as the parent. Note that the process state of the parent i.e., the address space, variables, open files etc. is copied into the child process. This means that the parent and child processes have identical but physically different address spaces. The change of values in parent process doesn't affect the child and vice versa is true too.

Both processes start execution from the next line of code i.e., the line after the fork() call. Let's look at an example:

```
//example.c
#include
void main() {
  int val;
  val = fork();  // line A
```

```
  printf("%d",val);  // line B
}
```

When the above example code is executed, when **line A** is executed, a child process is created. Now both processes start execution from **line B**. To differentiate between the child process and the parent process, we need to look at the value returned by the fork() call.

The difference is that, in the parent process, fork() returns a value which represents the **process ID** of the child process. But in the child process, fork() returns the value 0.

This means that according to the above program, the output of parent process will be the **process ID** of the child process and the output of the child process will be 0.

Exec()

The exec() system call is also used to create processes. But there is one big difference between **fork()** and **exec()** calls. The fork() call creates a new process while preserving the parent process. But, an exec() call replaces the address space, text segment, data segment etc. of the current process with the new process.

It means, after an exec() call, only the new process exists. The process which made the system call, wouldn't exist.

There are many flavors of **exec()** in UNIX, one being **execl()** which is shown below as an example:

```
//example2.c
#include
void main() {
  execl("/bin/ls", "ls", 0); // line A
  printf("This text won't be printed unless an error occurs in exec().");
}
```

As shown above, the first parameter to the execl() function is the address of the program which needs to be executed, in this case, the address of the **ls** utility in UNIX. Then it is followed by the name of the program which is **ls** in this case and followed by optional arguments. Then the list should be terminated by a NULL pointer (0).

When the above example is executed, at line A, the **ls** program is called and executed and the current process is halted. Hence the **printf()** function is never called since the process has already been halted. The only exception to this is that, if the **execl()** function causes an error, then the printf() function is executed.

MEMORY MANAGEMENT
Memory:- it is a large array of words or bytes, each with its own address.
Addressing binding:-
The binding of instructions and data to memory addresses can be called address binding.
The binding can be done at 3 ways.
1) compile time 2)load time 3) execution time.

compile time:- binding can be done at the compilation time. i.e if it is known at the compile time where the process will reside in memory, then the absolute code can be generated.
Load time:-binding can be done at the load time. if it is not known at compile time, where the process will reside in memory, then the compiler must generate the relocatable code.in this case the final binding is delayed until load time.
Execution time:-the binding must be delayed until run time.

Dynamic loading:-

The loading being postponed until execution time. to obtain the better memory utilization we can use dynamic loading. In this a routine is not loaded until it is called. All routines are kept on disk in a relocartable load format.
In this unused routine is never loaded .

Dynamic linking:-the linking being postponed until execution time. most o.s support static linking, in which all system language libraries are treated as module and kept in memory.for this much memory is wasted. But in dynamic linking it used the stub. It is a small piece of code that indicates how to load the library if the routine is not already present.

OVER LAYS:-

See text book.

Logical versus physical address space:-

L.M:-
1) the address generated by the C.P.U or user process is commonly referred as logical address.
2) It is relative address.
3) The set of all logical addresses generated by a program is called logical address space. The user programmer deals with logical addresses.
4) the logical address is used in user mode.

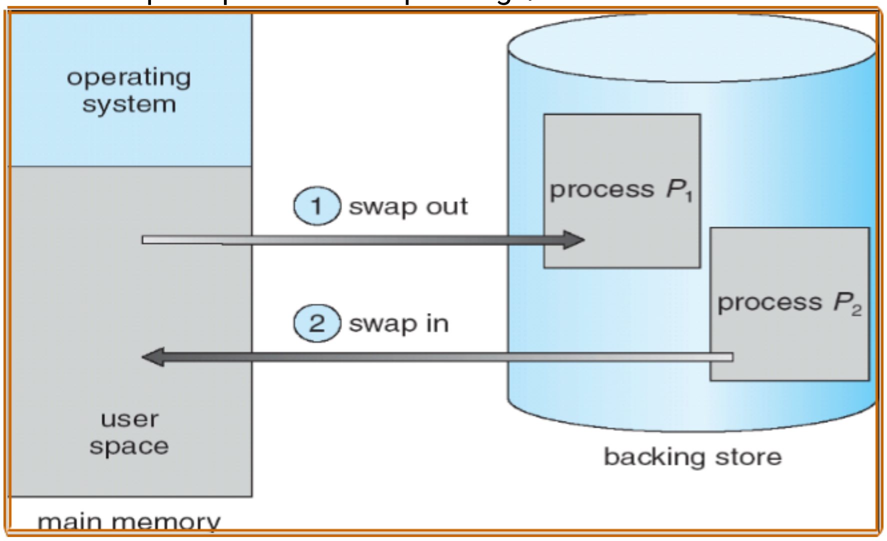
P.M:-
1) an address seen by the memory unit is called physical address.
2) It is absolute address.
3) The set of all physical addresses corresponding to these logical addresses is referred
4) A computer system has a physical memory, which is a H/Wdevice.
5) Physical address are only used in system mode.
6) to as a physical address space. User program never see the real physical address.

In compile time and load time address binding the logical address and physical addresses are same.

Swapping:-

A process needs to be in memory to be executed. A process however can be swapped temporarily out of memory to a backing store, and then brought back in to memory for continued execution.
EX:- In round robin C.P.U scheduling algorithm, when a quantum expires, the memory manager will start to swap out the process from the memory, and to swap in another process to that memory space.
EX:- In preemptive priority base algorithm if a higher priority process arrives and wants service, the memory manager can swap out the lower priority process, so that it can be load and execute the higher priority process. When higher priority process finishes, the lower priority process can be swapped back in and continued. This is called Roll Out and Roll In.

Swapping requires a backing store. the backing store is commonly a fast disk. It must be large enough to accommodate copies of all memory images for all users. And must provide direct access to these memory images.
Never swap the process with pending I/O.



CONTIGUOUS ALLOCATION:-
The memory is usually divided in to two partitions. One for resident O.S, and one for the user processes. It is possible to place the O.S in either low or high memory. But usually O.S is kept in low memory because the interrupt vectors is often in low memory. There are two types.

1)Single partition allocation
2) Multiple partition allocation
a) fixed sized partition(M.F.T)
b) Variable sized partition(M.V.T)

Single partition allocation:-
The O.S is residing in low memory and the user processes are executing in high memory. We need to protect the O.S code and data from changes by the user processes. We can provide this protection by using a relocation register and limit registers. The relocation register contains value of the smallest physical address and limit register contains the range of Logical address.

DISadvantages:-
1) Much memory is wasted.
2) Only one user process can run at a time.

Multiple partition allocation:-
If there are several user processes residing in memory at the same time, then the problem can occur, how to allocate available memory to the various processes.
One of the simplest scheme for memory allocation is to divided memory.

1) Fixed sized partition(M.F.T):-
memory is divided in to a no.of fixed sized partitions. Each partition maycontain exactly one process. Thus the degree of multiprogramming is bound by the no of partitions. When a partition is free, a process is selected from the input queue and is loaded in to the free partitions. When the process terminates the partition becomes available for another proces**s.**

Disadvantages:-

1) In M.F.T, the internal fragmentation can occur.
2) Suppose the process needs less memory than its allocated partition , then some memory is wasted.
3) Similarly a process needs large memory than its allocated partition size , then the problem will occur.

Variable sized partitions(M.V.T):- here the O.S maintain a table indicating which parts of memory are available and which are occupied. Initially all memory is available for user processes, and is considered as one large block of available memory, called **hole**. When a process arrives and needs a memory, we search for a hole large enough for this process. If we find one we allocate only as much memory as is needed, remaining is keep available to satisfy the other request.

Generally as the process enter the system , they are put in to an input queue. The O.S can takes the information from each process how much memory it needs. And O.s can alsofind how much memory is available and determining which processes are allocated memory. Process loads in to memory. It executes and when the process terminates, it release its memory. This memory is filled with other process by O.S.

We search for a hole among the set of holes to allocate the process, then 3 strategies used to select a free hole from a set of available holes.

1)first fit 2)best fit 3)worst fit

1)first fit:- allocate the first hole that is big enough.
2)**best hole:-** allocate the smallest hole that is big enough.
3) **worst hole:-** allocate the largest hole.

Disadvantages:-

1) in which external fragmentation can occurs.

Fragmentation:-

The wastage of memory space is called fragmentation. There are two types .
1)Internal fragmentation 2)External fragmentation.

1)external fragmentation:- it exists when enough total memory space is available to satisfy a request, but it is not contiguous.

EX:-

In this example the process p5 can requesting 500k. but in fig total available memory space is (300+260)=560k but that space is not contiguous.

2)**internal fragmentations:-**

the wastage of memory at the internal block.

EX:- In M.F.T a process is allocated to a fixed partition . the size of each partition is 5 bytes. The process requesting memory 4 bytes. Here 1 byte of memory is wasted in internal block. This is called internal fragmentation. Another type of problem arises in multiple partition allocation is

Suppose that the next process request 18,462 bytes. If we allocate exactly the requested block, then 2 bytes of hole is left free. To maintain the 2 bytes of free hole O.S needs more memory than this memory. So the general approach is to allocate very small holes as part of the larger request. thus the allocated memory may be slightly larger than the requested memory. The difference between these two numbers is internal fragmentation.

Compaction:- one solution to the problem of external fragmentation is **compaction.**

The goal of the compaction is to shuffle the memory contents to place all the free memory to gather in one large block.

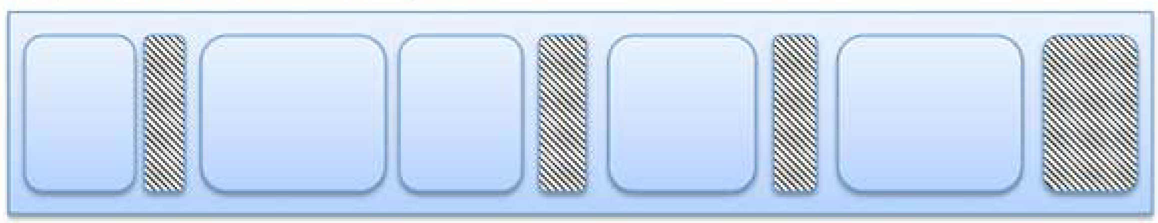
Fragmentation

As processes are loaded and removed from memory, the free memory space is broken into little pieces. It happens after sometimes that processes cannot be allocated to memory blocks considering their small size and memory blocks remains unused. This problem is known as Fragmentation.
Fragmentation is of two types –

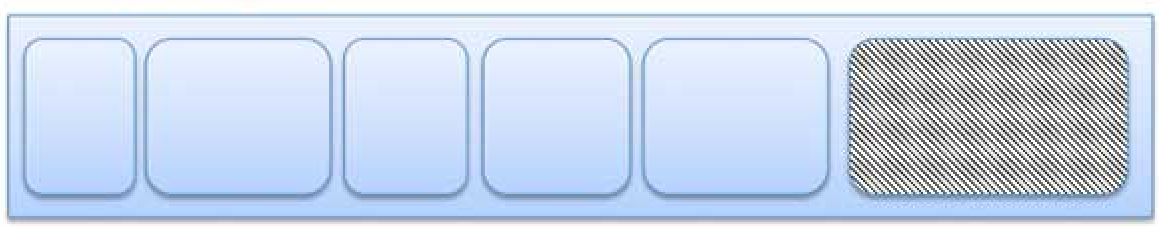
S.N.	Fragmentation & Description
1	**External fragmentation** Total memory space is enough to satisfy a request or to reside a process in it, but it is not contiguous, so it cannot be used.
2	**Internal fragmentation** Memory block assigned to process is bigger. Some portion of memory is left unused, as it cannot be used by another process.

The following diagram shows how fragmentation can cause waste of memory and a compaction technique can be used to create more free memory out of fragmented memory –

Fragmented memory before compaction



Memory after compaction



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.
Compaction is not always possible. We moved the processes, then these processes to be able to execute in their new locations, internal addresses must be relocated. If the relocation is static, compaction is not possible.
If the relocation is dynamic then compaction is possible.
When a compaction is possible, we must determine its cost. The simplest compaction algorithm is to move all processes towards one end of memory, all holes move in the

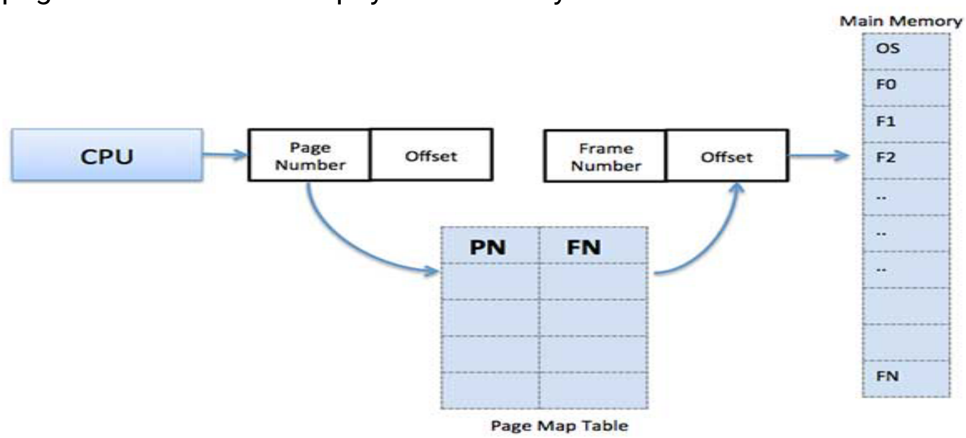
other direction, and producing one large hole of available memory. But it is very expensive. We note that one large hole of available memory is not at the end of memory, but rather is in the middle.

Paging:-

Another possible solution to the external fragmentation problem is to paging.
The physical memory is broken into fixed sized blocks called **frames.**
The logical memory is also broken in to blocks of the same size called **pages.**
When the process is to be executed, its pages are loaded in to any available memory frames from the backing store. The backing store is divided in to fixed sized blocks that are of the same size the memory frames.

Paging hardware:-

Every address generated by the C.P.U is divided in to two parts:
1) page number 2) page offset
the page number is used as an index in to a page table. The page table contains the base address of each page in physical memory. This base address is combined with the pageoffset to define the physical memory.

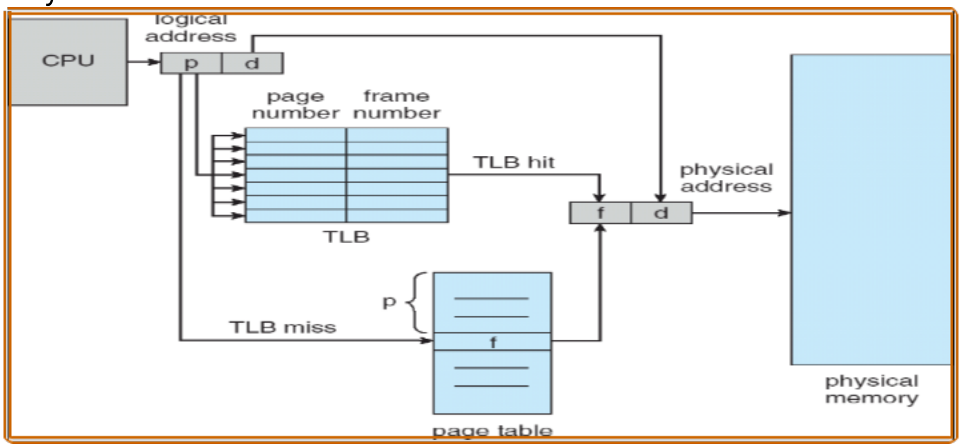


The page size is defined by the H/W. the size of a page is typically a power of 2. power of 2 is selected as page size because translation of a logical address in to a page number and page offset is easy.

Structure of the page table:-

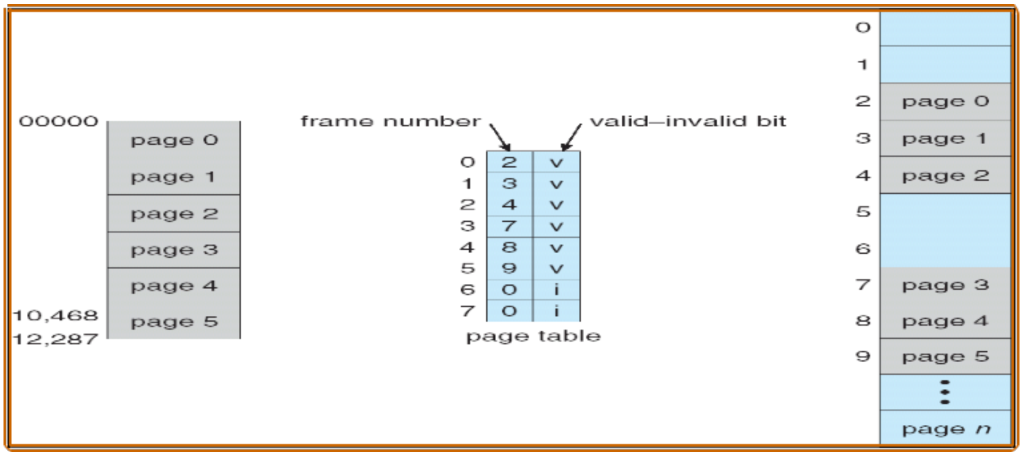
The page table can be implemented in 3 different ways.
1) the page table can be implemented as a set of dedicated registers. These registers are built with very high logic to easy translation of logic address. The C.P.U dispatcher reloads these registers.
The use of registers for page table is satisfactory if the page table is small.
The page table to be very large then the registers are not feasible.
2) in this method *the page table is kept in main memory and* page table base register(PTBR) points to the page table. Changing page table requires changing only this one register, reducing the context switch.
The problem with this approach is the time required to access a user memory location. If we want to access location I , we must first index in to the page table using PTBR, this task requires memory access. It provides us frame number, which is combined with page offset to produce the actual address. We can access the desired place in memory. This scheme requires two memory accesses.
3)The solution to this problem is to use a special registers *called associative registers*

or translation look aside buffer(TLB’s) . a set of associative registers is built with very high speed memory. Each register consist of two parts.
1)a key and 2) a value.
When the associative registers are presented with an item it is compared with all keys simultaneously. If item is found the corresponding value field is output. The searching is very fast.



the associative registers contains only a few of the page table entries.
When a logical address is generated by C.P.U its page number is presented as a set of associative registers that contains page number and their corresponding frame number. If a page number is found in associative registers, its frame number is immediately available and is used to access memory.
If page number is not in associative registers, a memory reference to the page table must be made. Suppose if TLB is full then the O.S must select one for replacement.

PROTECTION:-
Memory protection in paged environment is accomplished by two ways.
1) by using valid-invalid bit.
2) *Read only (or) read& write.*
These protection bits are attached to the each entry of the page table.
Here the valid-invalid bit is attached to the each entry in the page table.That bit containing “v” if it is valid, i.e the page of the process is in its logical address space. This bit containing “i” if it is invalid. I.e page of process is not in its logical address space.
Ex:-



Here a process containing 0 to 5 (six) pages. So page numbers 6,7,8 will have "i" in the valid-invalid bit . so if pages 6,7,8 if referenced then trap will occur.
Other method is use read& write or readonly bits to the each entry of the page table. The page table can be checked to verify that no writes are being made to a read only page.

Advantages and Disadvantages of Paging

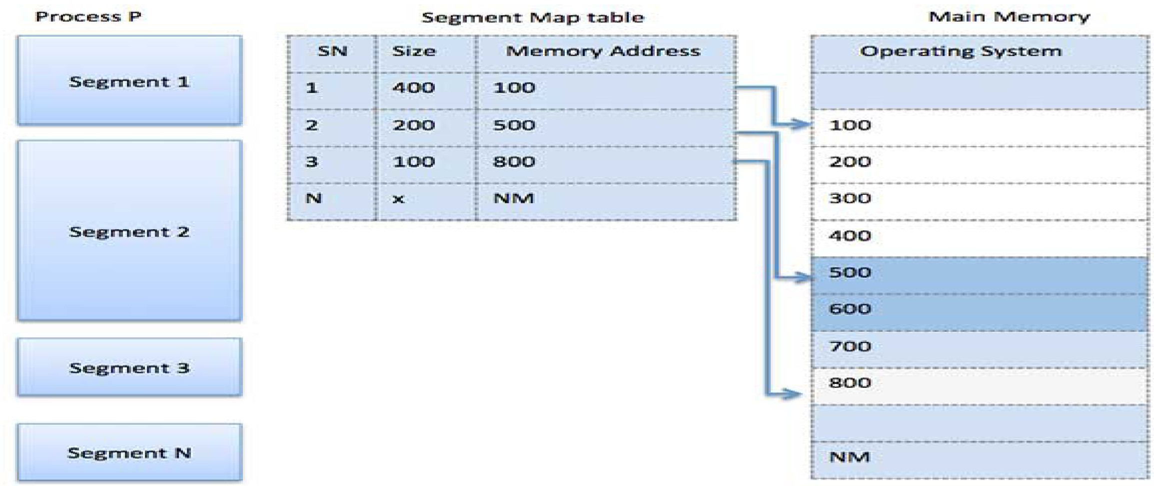
Here is a list of advantages and disadvantages of paging –

- Paging reduces external fragmentation, but still suffer from internal fragmentation.
- Paging is simple to implement and assumed as an efficient memory management technique.
- Due to equal size of the pages and frames, swapping becomes very easy.

Page table requires extra memory space, so may not be good for a system having small RAM

Segmentation

Segmentation is a memory management technique in which each job is divided into several segments of different sizes, one for each module that contains pieces that perform related functions. Each segment is actually a different logical address space of the program.
When a process is to be executed, its corresponding segmentation are loaded into non-contiguous memory though every segment is loaded into a contiguous block of available memory.
Segmentation memory management works very similar to paging but here segments are of variable-length where as in paging pages are of fixed size.
A program segment contains the program's main function, utility functions, data structures, and so on. The operating system maintains a **segment map table** for every process and a list of free memory blocks along with segment numbers, their size and corresponding memory locations in main memory. For each segment, the table stores the starting address of the segment and the length of the segment. A reference to a memory location includes a value that identifies a segment and an offset.



Operating System - Virtual Memory

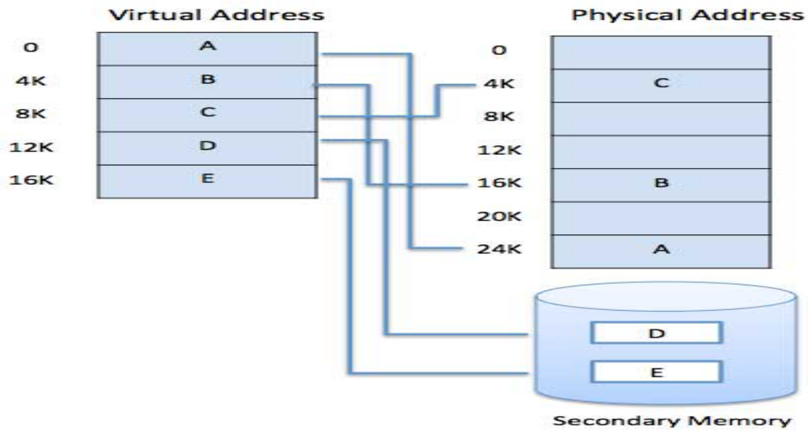
A computer can address more memory than the amount physically installed on the system. This extra memory is actually called **virtual memory** and it is a section of a hard disk that's set up to emulate the computer's RAM.

The main visible advantage of this scheme is that programs can be larger than physical memory. Virtual memory serves two purposes. First, it allows us to extend the use of physical memory by using disk. Second, it allows us to have memory protection, because each virtual address is translated to a physical address.

Following are the situations, when entire program is not required to be loaded fully in main memory.

- User written error handling routines are used only when an error occurred in the data or computation.
- Certain options and features of a program may be used rarely.
- Many tables are assigned a fixed amount of address space even though only a small amount of the table is actually used.
- The ability to execute a program that is only partially in memory would counter many benefits.
- Less number of I/O would be needed to load or swap each user program into memory.
- A program would no longer be constrained by the amount of physical memory that is available.
- Each user program could take less physical memory, more programs could be run the same time, with a corresponding increase in CPU utilization and throughput.

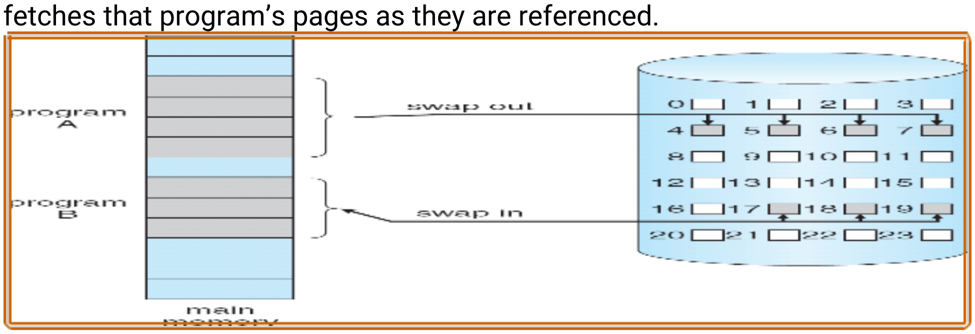
Modern microprocessors intended for general-purpose use, a memory management unit, or MMU, is built into the hardware. The MMU's job is to translate virtual addresses into physical addresses. A basic example is given below –



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.

Demand Paging

A demand paging system is quite similar to a paging system with swapping where processes reside in secondary memory and pages are loaded only on demand, not in advance. When a context switch occurs, the operating system does not copy any of the old program's pages out to the disk or any of the new program's pages into the main memory Instead, it just begins executing the new program after loading the first page and fetches that program's pages as they are referenced.



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.

Advantages

Following are the advantages of Demand Paging –

- Large virtual memory.
- More efficient use of memory.
- There is no limit on degree of multiprogramming.

Disadvantages

- Number of tables and the amount of processor overhead for handling page interrupts are greater than in the case of the simple paged management techniques.

Page Replacement Algorithm

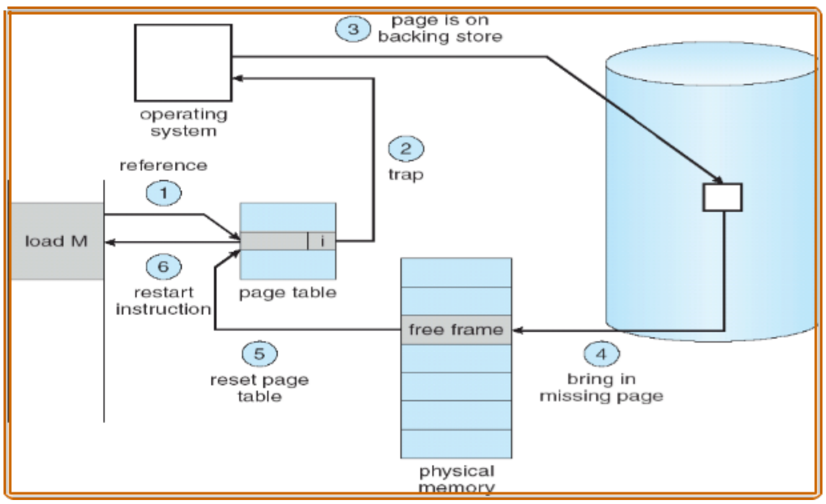
Page replacement algorithms are the techniques using which an Operating System decides which memory pages to swap out, write to disk when a page of memory needs to be allocated. Paging happens whenever a page fault occurs and a free page cannot be used for allocation purpose accounting to reason that pages are not available or the number of free pages is lower than required pages.

When the page that was selected for replacement and was paged out, is referenced again, it has to read in from disk, and this requires for I/O completion. This process determines the quality of the page replacement

algorithm: the lesser the time waiting for page-ins, the better is the algorithm.
A page replacement algorithm looks at the limited information about accessing the pages provided by hardware, and tries to select which pages should be replaced to minimize the total number of page misses, while balancing it with the costs of primary storage and processor time of the algorithm itself. There are many different page replacement algorithms. We evaluate an algorithm by running it on a particular string of memory reference and computing the number of page faults,

Page Replacement includes the following steps:

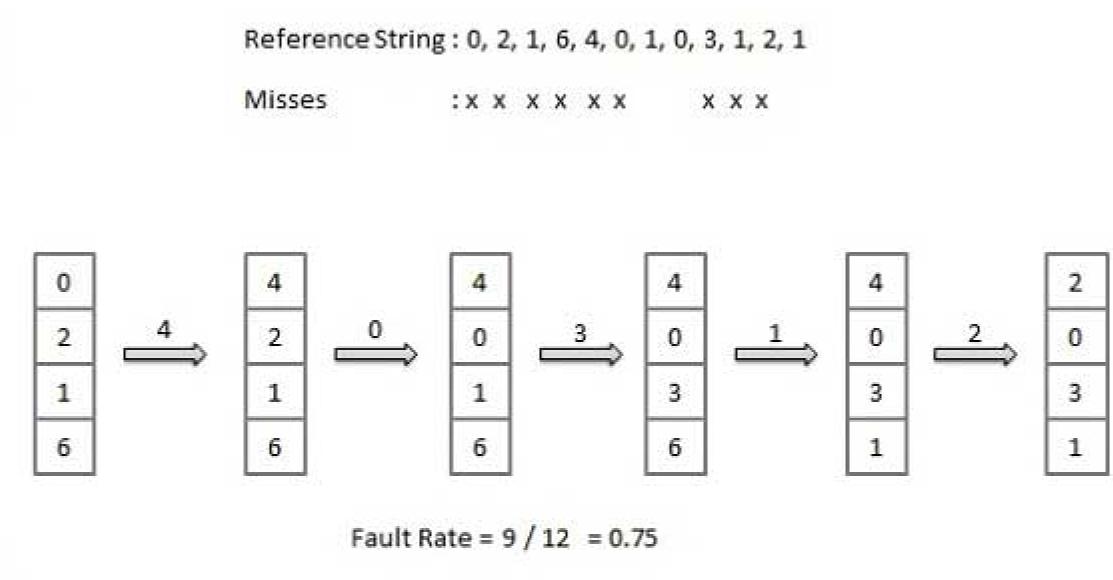
1. Find the Location of the desired page on the disk.
2. Find a free frame:
    a. If there is a free frame, use it.
    b. If there is no free frame use a page replacement algorithm to select a victim frame.
    c. Write the victim page to the disk, change the page and frame tables accordingly.
3. Read the desired page into the free frame, change the page and frame tables.
4. Restart the user process.



First In First Out (FIFO) algorithm

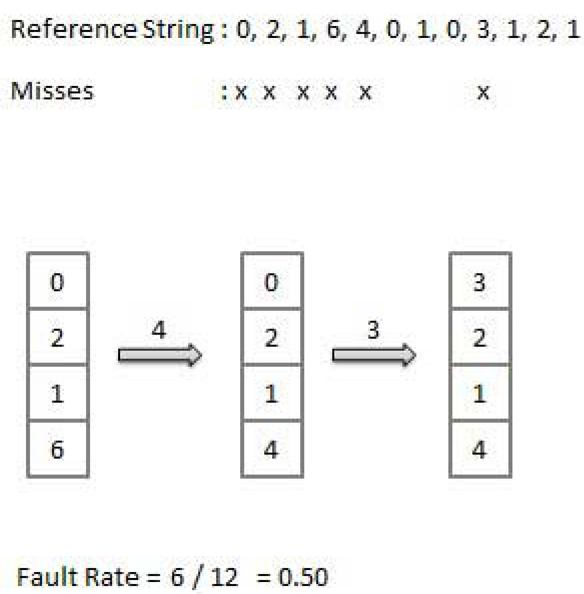
The simplest page-replacement algorithm is FIFO algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen.

- Oldest page in main memory is the one which will be selected for replacement.
- Easy to implement, keep a list, replace pages from the tail and add new pages at the head.



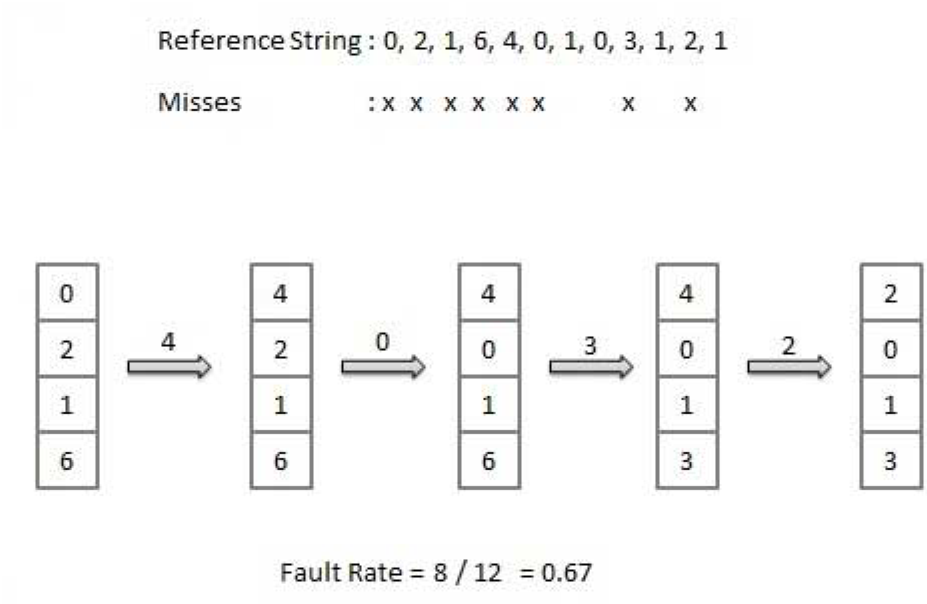
Optimal Page algorithm

- An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms. An optimal page-replacement algorithm exists, and has been called OPT or MIN.
- Replace the page that will not be used for the longest period of time. Use the time when a page is to be used.



Least Recently Used (LRU) algorithm

- Page which has not been used for the longest time in main memory is the one which will be selected for replacement.
- Easy to implement, keep a list, replace pages by looking back into time.



Page Buffering algorithm

- To get a process start quickly, keep a pool of free frames.
- On page fault, select a page to be replaced.
- Write the new page in the frame of free pool, mark the page table and restart the process.
- Now write the dirty page out of disk and place the frame holding replaced page in free pool.

Least frequently Used(LFU) algorithm

- The page with the smallest count is the one which will be selected for replacement.
- This algorithm suffers from the situation in which a page is used heavily during the initial phase of a process, but then is never used again.

Most frequently Used(MFU) algorithm

- This algorithm is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.