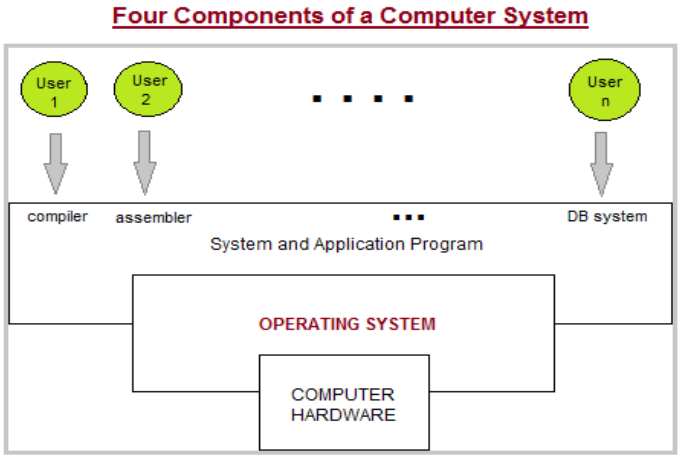
Operating Systems

- In simple terms, an operating system is the interface between the user and the machine.
- resource allocator
- manager (memory,process,file,security)



Operating System Management Tasks

1. **Process 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

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

Kundan Thakur

1. Batch operating system

The users of a batch operating system do not interact with the computer directly. Each user prepares his job on an off-line device like punch cards and submits it to the computer operator. To speed up processing, jobs with similar needs are batched together and run as a group. The programmers leave their programs with the operator and the operator then sorts the programs with similar requirements into batches.
The problems with Batch Systems are as follows −

- Lack of interaction between the user and the job.
- CPU is often idle, because the speed of the mechanical I/O devices is slower than the CPU.
- Difficult to provide the desired priority.

2. Multiprogramming Batch Systems

- In this the operating system picks up and begins to execute one of the jobs 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 the system chooses which one to run through the process of **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.

3. 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.

1. Execution of several tasks by different processors concurrently, increases the system's throughput without speeding up the execution of a single task.
2. 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.

4. Distributed operating System

Distributed systems use multiple central processors to serve multiple real-time applications and multiple users. Data processing jobs are distributed among the processors accordingly.
The processors communicate with one another through various communication lines (such as high-speed buses or telephone lines). These are referred as **loosely coupled systems** or distributed systems. Processors in a distributed system may vary in size and function. These processors are referred as sites, nodes, computers, and so on.
The advantages of distributed systems are as follows −

- With resource sharing facility, a user at one site may be able to use the resources available at another.
- Speedup the exchange of data with one another via electronic mail.
- If one site fails in a distributed system, the remaining sites can potentially continue operating.
- Better service to the customers.
- Reduction of the load on the host computer.
- Reduction of delays in data processing.

5. Real Time operating System

A real-time system is defined as a data processing system in which the time interval required to process and respond to inputs is so small that it controls the environment. The time taken by the system to respond to an

input and display of required updated information is termed as the **response time**. So in this method, the response time is very less as compared to online processing.
Real-time systems are used when there are rigid time requirements on the operation of a processor or the flow of data and real-time systems can be used as a control device in a dedicated application. A real-time operating system must have well-defined, fixed time constraints, otherwise the system will fail. For example, Scientific experiments, medical imaging systems, industrial control systems, weapon systems, robots, air traffic control systems, etc.
There are two types of real-time operating systems.

Hard real-time systems
Hard real-time systems guarantee that critical tasks complete on time. In hard real-time systems, secondary storage is limited or missing and the data is stored in ROM. In these systems, virtual memory is almost never found.

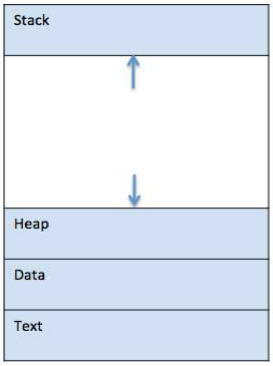
Soft real-time systems
Soft real-time systems are less restrictive. A critical real-time task gets priority over other tasks and retains the priority until it completes. Soft real-time systems have limited utility than hard real-time systems. For example, multimedia, virtual reality, Advanced Scientific Projects like undersea exploration and planetary rovers, etc

Process Management

Program vs Process
A process is a program in execution. A process is an 'active' entity as opposed to a program which is considered to be a 'passive' entity. A single program can create many processes when run multiple times, for example when we open a .exe or binary file multiple times, many instances begin (many processes are created).

What does a process look like in Memory?

- **Text Section**:A Process is also sometimes known as the Text Section. It also includes the current activity represented by the value of Program Counter.
- **Stack**: Stack contains the temporary data such as function parameters, returns address and local variables.
- **Data Section:** Contains the global variable and static variable.
- **Heap Section:** Contain Dynamically allocated memory to process during its run time.



Process Control Block
While creating a process the operating system performs several operations. To identify these process, it must identify each process, hence it assigns a process identification number (PID) to each process. As the operating system supports multi-programming, it needs to keep track of the all the processes. For this task, the process control block (PCB) is used to track the process's execution status. Each block of memory contains information about the process state, program counter, stack pointer, status of opened files, scheduling algorithms, etc. All these information is required and must be saved when the process is switched from one state to another. When the process made transitions from one state to another, the operating system must update information in the process's PCB.

There is a Process Control Block for each process, enclosing all the information about the process.

Attributes / context / Characteristics of a Process

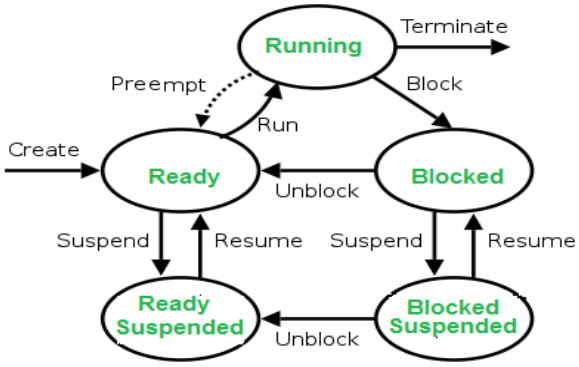
It is a data structure, which contains the following:

1. **Process Id:** A unique identifier assigned by operating system
2. **Process State:** Can be ready, running, .. etc
3. **CPU registers:** Like Program Counter (CPU registers must be saved and restored when a process is swapped out and in of CPU).
4. **Program Counter**: holds the address of the next instruction to be executed for that process.
5. **I/O status information:** For example devices allocated to process, open files, etc
6. **CPU scheduling information:** For example Priority (Different processes may have different priorities, for example a short process may be assigned a low priority in the shortest job first scheduling).
7. **Memory Management information**: For example, page tables or segment tables.
8. **Priority:** priority of the process, if it have.
9. **Open files list :** This information includes the list of files opened for a process.

States of Process:

A process is in one of the following states

1. **New:** Newly Created Process (or) being created process.
2. **Ready:** After creation Process moves to Ready state, i.e., process is ready for execution.
3. **Run:** Currently running process in CPU (only one process at a time can be under execution in a single processor).
4. **Wait (or Block):** When process request for I/O request.
5. **Complete (or Terminated):** Process Completed its execution.
6. **Suspended Ready**: When ready queue becomes full, some processes are moved to suspend ready state
7. **Suspended Block:** When waiting queue becomes full.



What is Process Scheduling:-

The act of determining which process is in the **ready** state, and 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.

Scheduling 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.

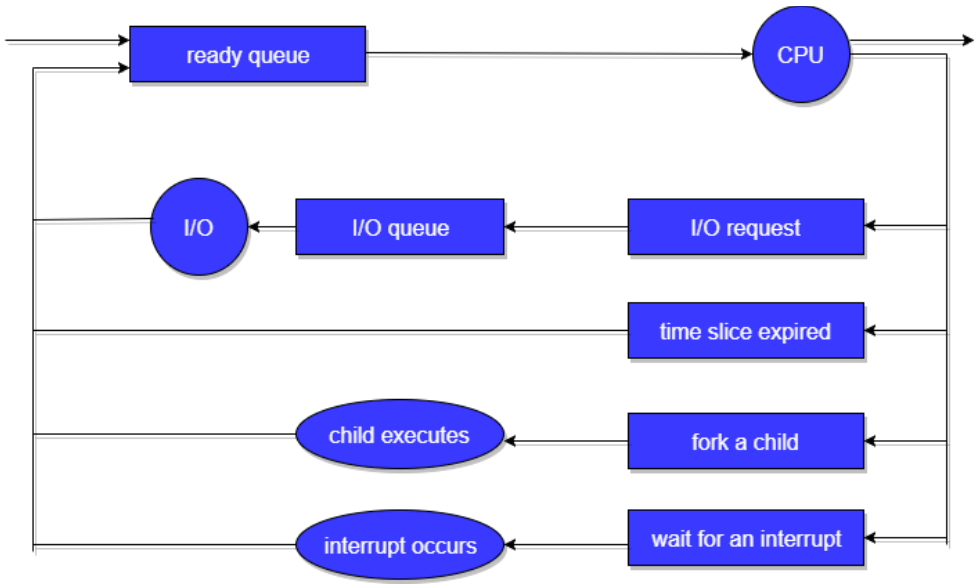
Process Scheduling Queues:-

The Operating System maintains the following important process scheduling queues −

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

A new process is initially put in the **Ready queue**. It waits in the ready queue until it is selected for execution(or dispatched). Once the process is assigned to the CPU and is executing, one of the following several events can occur:

- The process could issue an I/O request, and then be placed in the **I/O queue**.
- The process could create a new subprocess and wait for its termination.
- The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.



Types of Schedulers

There are three types of schedulers available:

1. Long Term Scheduler :

It is also called a **job 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.

Sometimes the number of processes submitted to the system are more than it can be executed immediately. Then in such cases, the processes are spooled on the mass storage, where they reside to get executed later. The **Long-Term Scheduler** then select the process from this spool which is also called as **Job Pool or job queue** and load them in the **Ready Queue** for their further execution.

2. Short Term Scheduler :

It is also called as **CPU scheduler**. It is change of ready state to running state of the process. CPU scheduler selects a process among the processes that are ready to execute and allocates CPU to one of them.
Short-term schedulers make the decision of which process to execute next. Short-term schedulers are faster than long-term schedulers. Sort term scheduler tell the dispatcher that which process is selected for execution, then dispatcher take process and give to cpu for execution.
Kundan Thakur

3. Medium Term Scheduler :

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

Degree of multiprogramming –

The number of process that can reside in the ready state at maximum decides the degree of multiprogramming, e.g., if degree of programming = 100 means 100 processes can reside in the ready state at maximum .

Context Switching

The process of saving the context of one process and loading the context of other process is known as Context Switching. In simple term, it is like loading and unloading of the process from running state to ready state.

When does Context switching happen?

1. When a high priority process comes to ready state, compared to priority of running process
2. Interrupt Occurs
3. User and Kernel mode switch: (It is not necessary though)
4. Preemptive CPU scheduling used

CPU Bound vs I/O Bound Processes:

A CPU Bound Process requires more amount of CPU time or spends more time in the running state.
I/O Bound Process requires more amount of I/O time and less CPU time. I/O Bound process more time in the waiting state.

First look on some term which use in scheduling :-

1. **Arrival Time:** Time at which the process arrives in the ready queue.
2. **Completion Time:** Time at which process completes its execution.
3. **Burst Time:** Time required by a process for CPU execution.
4. **Turn Around Time:**Time Difference between completion time and arrival time.

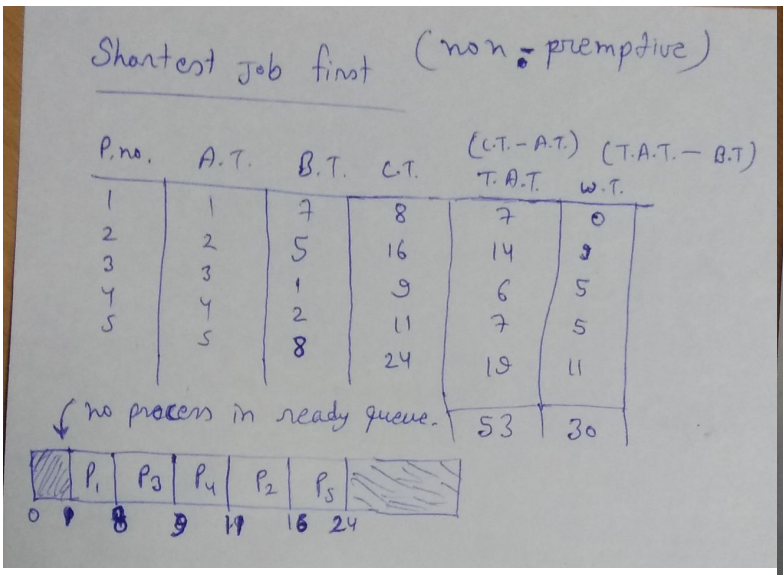
Turn Around Time = Completion Time - Arrival Time

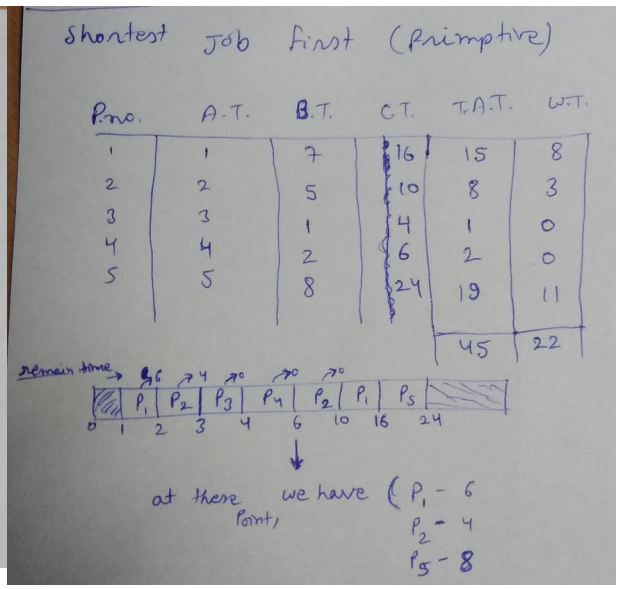
5. **Waiting Time(W.T):** Time Difference between turn around time and burst time.

Waiting Time = Turn Around Time - Burst Time

Different Scheduling Algorithms

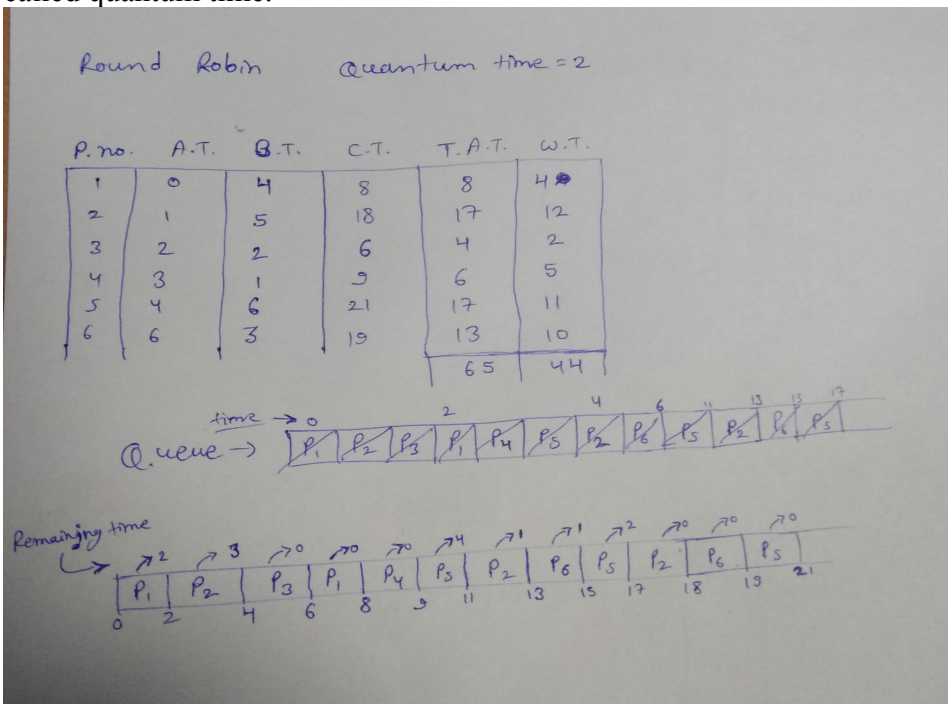
1. **First Come First Serve (FCFS):** Simplest scheduling algorithm that schedules according to arrival times of processes.
2. **Shortest Job First(SJF):** Process which have the shortest burst time are scheduled first.





3. **Shortest Remaining Time First(SRTF):** It is preemptive mode of SJF algorithm in which jobs are schedule according to shortest remaining time.

4. **Round Robin Scheduling:** Each process is assigned a fixed time in cyclic way. The fixed time is called quantum time.



5. **Priority Based scheduling (Non Preemptive):** In this scheduling, processes are scheduled according to their priorities, i.e., highest priority process is schedule first. If priorities of two processes match, then schedule according to arrival time.

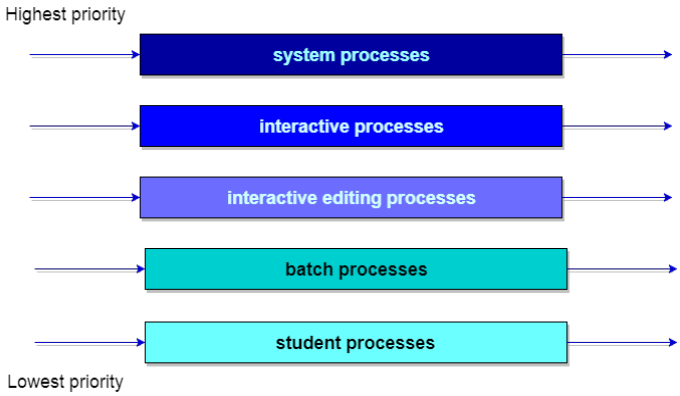
6. **Highest Response Ratio Next (HRRN):** In this scheduling, processes with highest response ratio is scheduled. This algorithm avoids starvation.

Response Ratio = (Waiting Time + Burst time) / Burst time.

7. **Multilevel Queue Scheduling:** According to the priority of process, processes are placed in the different queues. Generally high priority process are placed in the top level queue. Only after completion of processes from top level queue, lower level queued processes are scheduled.

A multi-level queue scheduling algorithm partitions the ready queue into several separate queues. The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type. Each queue has its own scheduling algorithm .

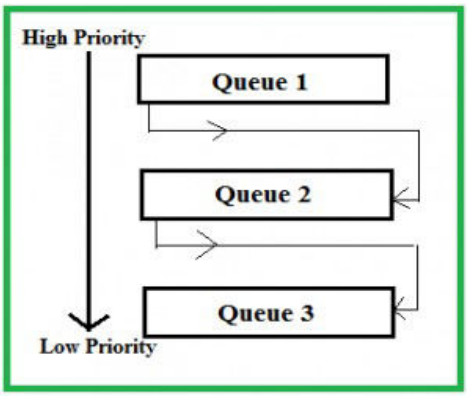
Let us consider an example of a multilevel queue-scheduling algorithm with five queues:
1. System Processes(FCFS)
2. interactive Processes(SJF)
3. Interactive Editing Processes
4. Batch Processes(RR)
5. Student Processes(PRIORITY)



There is may be starvation problem occur in it.

8. **Multi level Feedback Queue Scheduling:** This Scheduling is like Multilevel Queue(MLQ) Scheduling but in this the process's can move between the queues. **Multilevel Feedback Queue Scheduling (MLFQ)** keep analyzing the behavior (time of execution) of processes and according to which it changes its priority.Now, look at the diagram and explanation below to understand it properly.

   Now let us suppose that queue 1 and 2 follow round robin with time quantum 4 and 8 respectively and queue 3 follow FCFS.One implementation of MFQS is given below –



1. When a process starts executing then it first enters queue 1.
2. In queue 1 process executes for 4 unit and if it completes in this 4 unit or it gives CPU for I/O operation in this 4 unit than the priority of this process does not change and if it again comes in the ready queue than it again starts its execution in Queue 1.
3. If a process in queue 1 does not complete in 4 unit then its priority gets reduced and it shifted to queue 2.
4. Above points 2 and 3 are also true for queue 2 processes but the time quantum is 8 unit.In a general case if a process does not complete in a time quantum than it is shifted to the lower priority queue.
5. In the last queue, processes are scheduled in FCFS manner.

6. A process in lower priority queue can only execute only when higher priority queues are empty.
7. A process running in the lower priority queue is interrupted by a process arriving in the higher priority queue.

Some useful facts about Scheduling Algorithms:

1) FCFS can cause long waiting times, especially when the first job takes too much CPU time.
2) Both SJF and Shortest Remaining time first algorithms may cause starvation. Consider a situation when long process is there in ready queue and shorter processes keep coming.
3) If time quantum for Round Robin scheduling is very large, then it behaves same as FCFS scheduling.
4) SJF is optimal in terms of average waiting time for a given set of processes,i.e., average waiting time is minimum with this scheduling, but problems is, how to know/predict time of next job.

Convoy Effect in Operating Systems

Convoy Effect is phenomenon associated with the First Come First Serve (FCFS) algorithm, in which the whole Operating System slows down due to few slow processes.
FCFS algorithm is non-preemptive in nature, that is, once CPU time has been allocated to a process, other processes can get CPU time only after the current process has finished. This property of FCFS scheduling leads to the situation(not for indefinate time,like a long time) called Convoy Effect.

Starvation

Starvation or indefinite blocking is phenomenon associated with the Priority/SJF scheduling algorithms, in which a process ready to run for CPU can wait indefinitely because of low priority. In heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU.

A **livelock** is similar to a deadlock, except that the states of the processes involved in the **livelock** constantly change with regard to one another, none progressing. **Livelock** is a special case of resource starvation; the general definition only states that a specific process is not progressing.

A possible **solution** to **starvation** is to use a scheduling algorithm with priority queue that also uses the aging technique. Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time

---

Threads

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

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

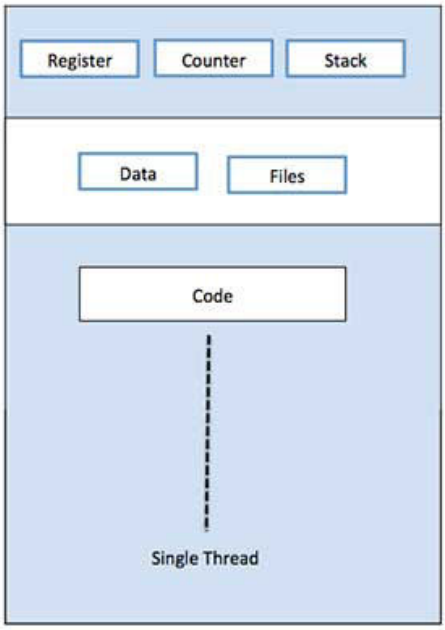
All the threads are independent of each other so they share the code, data, OS resources etc.

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

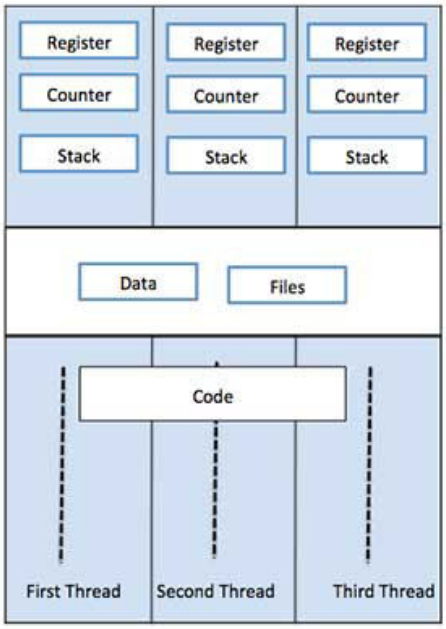
A thread shares with its peer threads few information like code segment, data segment and open files. When one thread alters a code segment memory item, all other threads see that.

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

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



Single Process P with single thread



Single Process P with three threads

Difference between Process and Thread

S.N.	Process	Thread
1	Process is heavy weight or resource intensive.	Thread is light weight, taking lesser resources than a process.
2	Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
3	In multiple processing environments, each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.
4	If one process is blocked, then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, a second thread in the same task can run.

5	Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
6	In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

Advantages of Thread

- Threads minimize the context switching time.
- Use of threads provides concurrency within a process.
- Efficient communication.
- It is more economical to create and context switch threads.
- Threads allow utilization of multiprocessor architectures to a greater scale and efficiency.

Similarity between Threads and Processes –

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

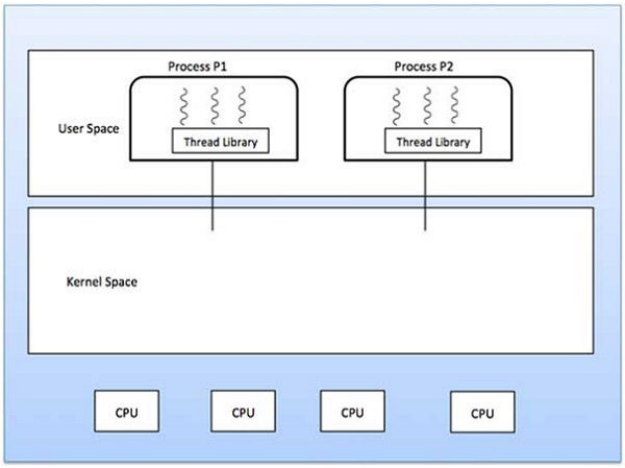
Types of Thread

Threads are implemented in following two ways −

- **User Level Threads** − User managed threads.
- **Kernel Level Threads** − Operating System managed threads acting on kernel, an operating system core.

User Level Threads

In this case, the thread management kernel is not aware of the existence of threads. The thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts. The application starts with a single thread.



Advantages

- Thread switching does not require Kernel mode privileges.
- User level thread can run on any operating system.

- Scheduling can be application specific in the user level thread.
- User level threads are fast to create and manage.

Disadvantages

- In a typical operating system, most system calls are blocking.
- Multithreaded application cannot take advantage of multiprocessing.

Kernel Level Threads

In this case, thread management is done by the Kernel. There is no thread management code in the application area. Kernel threads are supported directly by the operating system. Any application can be programmed to be multithreaded. All of the threads within an application are supported within a single process.

The Kernel maintains context information for the process as a whole and for individuals threads within the process. Scheduling by the Kernel is done on a thread basis. The Kernel performs thread creation, scheduling and management in Kernel space. Kernel threads are generally slower to create and manage than the user threads.

Advantages

- Kernel can simultaneously schedule multiple threads from the same process on multiple processes.
- If one thread in a process is blocked, the Kernel can schedule another thread of the same process.
- Kernel routines themselves can be multithreaded.

Disadvantages

- Kernel threads are generally slower to create and manage than the user threads.
- Transfer of control from one thread to another within the same process requires a mode switch to the Kernel.

Difference between User-Level & Kernel-Level Thread

S.N.	User-Level Threads	Kernel-Level Thread
1	User-level threads are faster to create and manage.	Kernel-level threads are slower to create and manage.
2	Implementation is by a thread library at the user level.	Operating system supports creation of Kernel threads.
3	User-level thread is generic and can run on any operating system.	Kernel-level thread is specific to the operating system.
4	Multi-threaded applications cannot take advantage of multiprocessing.	Kernel routines themselves can be multithreaded.

Why Multithreading?

Thread is also known as lightweight process. The idea is achieve parallelism by dividing a process into multiple threads. For example, in a browser, multiple tabs can be different threads. MS word uses multiple threads, one thread to format the text, other thread to process inputs etc.

Multithreading Models

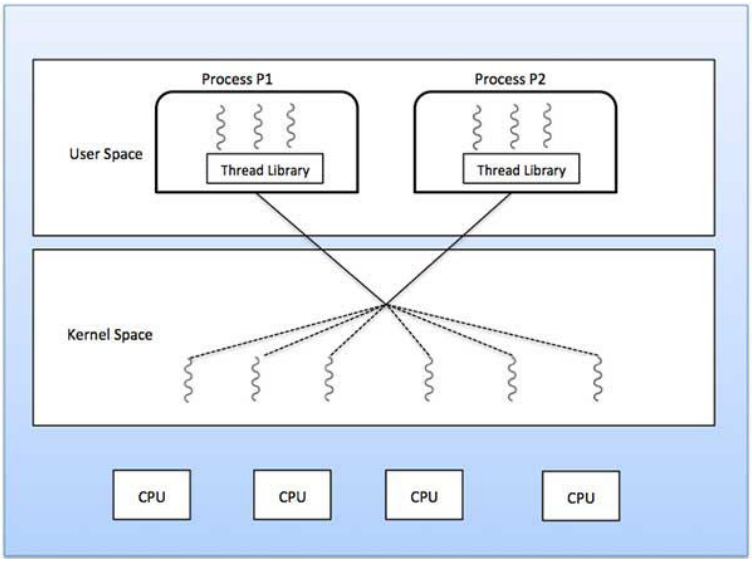
Some operating system provide a combined user level thread and Kernel level thread facility. Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the entire process. Multithreading models are three types

- Many to many relationship.
- Many to one relationship.
- One to one relationship.

Many to Many Model

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

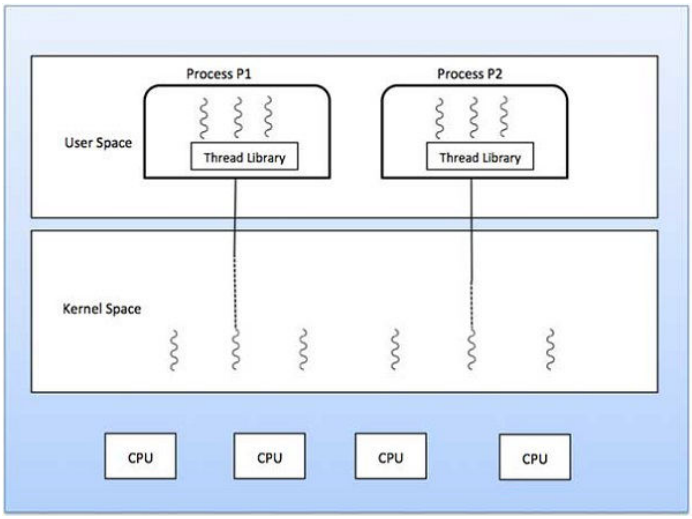
The following diagram shows the many-to-many threading model where 6 user level threads are multiplexing with 6 kernel level threads. In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallel on a multiprocessor machine. This model provides the best accuracy on concurrency and when a thread performs a blocking system call, the kernel can schedule another thread for execution.



Many to One Model

Many-to-one model maps many user level threads to one Kernel-level thread. Thread management is done in user space by the thread library. When thread makes a blocking system call, the entire process will be blocked. Only one thread can access the Kernel at a time, so multiple threads are unable to run in parallel on multiprocessors.

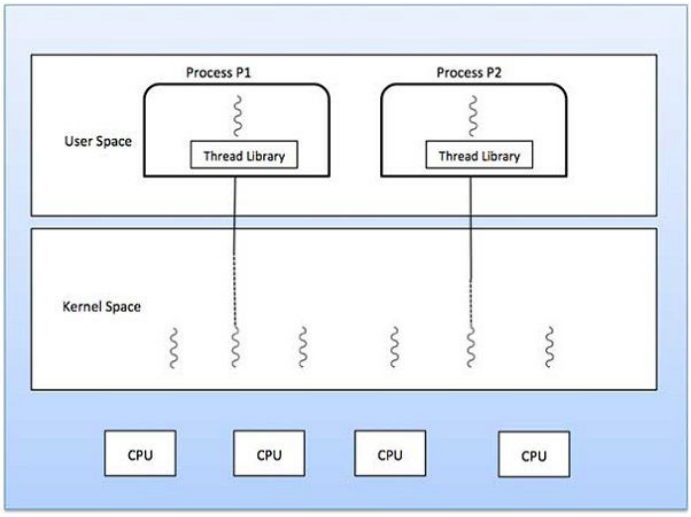
If the user-level thread libraries are implemented in the operating system in such a way that the system does not support them, then the Kernel threads use the many-to-one relationship modes.



One to One Model

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

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



Benefits of Multithreading:-

The benefits of multi threaded programming can be broken down into four major categories:

1. **Responsiveness –** Multithreading in an interactive application may allow a program to continue running even if a part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.

In a non multi threaded environment, a server listens to the port for some request and when the request comes, it processes the request and then resume listening to another request. The time taken while processing of request makes other users wait unnecessarily. Instead a better approach would be to pass the request to a worker thread and continue listening to port.

Kundan Thakur

For example, a multi threaded web browser allow user interaction in one thread while an video is being loaded in another thread. So instead of waiting for the whole web-page to load the user can continue viewing some portion of the web-page.

2. **Resource Sharing –** Processes may share resources only through techniques such as-

- Message Passing
- Shared Memory

Such techniques must be explicitly organized by programmer. However, threads share the memory and the resources of the process to which they belong by default. The benefit of sharing code and data is that it allows an application to have several threads of activity within same address space.

3. **Economy –**Allocating memory and resources for process creation is a costly job in terms of time and space. Since, threads share memory with the process it belongs, it is more economical to create and context switch threads. Generally much more time is consumed in creating and managing processes than in threads. In Solaris, for example, creating process is 30 times slower than creating threads and context switching is 5 times slower.

4. **Scalability –** The benefits of multi-programming greatly increase in case of multiprocessor architecture, where threads may be running parallel on multiple processors. If there is only one thread then it is not possible to divide the processes into smaller tasks that different processors can perform. Single threaded process can run only on one processor regardless of how many processors are available. Multi-threading on a multiple CPU machine increases parallelism

Process-based and Thread-based Multitasking

A **multitasking operating system** is an operating system that gives you the perception of 2 or more tasks/jobs/processes running at the same time. It does this by dividing system resources amongst these tasks/jobs/processes and switching between the tasks/jobs/processes while they are executing over and over again. Usually CPU processes only one task at a time but the switching is so fast that it looks like CPU is executing multiple processes at a time. They can support either **preemptive** multitasking, where the OS doles out time to applications (virtually all modern OSes) or **cooperative** multitasking, where the OS waits for the program to give back control (Windows 3.x, Mac OS 9 and earlier), leading to hangs and crashes. Also known as **Timesharing**, multitasking is a logical extension of multiprogramming.

Multitasking programming is of two types –

1. Process-based Multitasking
2. Thread-based Multitasking.

Process Based Multitasking Programming –

- In process based multitasking two or more processes and programs can be run concurrently.
- In process based multitasking a process or a program is the smallest unit.
- Program is a bigger unit.
- Process based multitasking requires more overhead.
- Process requires its own address space.
- Process to Process communication is expensive.
- Here, it is unable to gain access over idle time of CPU.
- It is comparatively heavy weight.
- It has slower data rate multi-tasking.

Example – We can listen to music and browse internet at the same time. The processes in this example are the music player and browser.

Thread Based Multitasking Programming –

- In thread based multitasking two or more threads can be run concurrently.
- In thread based multitasking a thread is the smallest unit.
- Thread is a smaller unit.
- Thread based multitasking requires less overhead.
- Threads share same address space.
- Thread to Thread communication is not expensive.
- It allows taking gain access over idle time taken by CPU.
- It is comparatively light weight.
- It has faster data rate multi-tasking.

What is a Thread? What are the differences between process and thread?
A thread is a single sequence stream within in a process. Because threads have some of the properties of processes, they are sometimes called *lightweight processes*. Threads are popular way to improve application through parallelism. For example, in a browser, multiple tabs can be different threads. MS word uses multiple threads, one thread to format the text, other thread to process inputs, etc.
A thread has its own program counter (PC), a register set, and a stack space. Threads are not independent of one other like processes as a result threads shares with other threads their code section, data section and OS resources like open files and signals.

Process Synchronization

Process Synchronization means several cooperative process run in an operating system and they are sharing the resource in such a way that the data inconsistency not arise.

On the basis of synchronization, processes are categorized as one of the following two types:

- **Independent Process :** Execution of one process does not affects the execution of other processes.
- **Cooperative Process :** Execution of one process affects the execution of other processes.

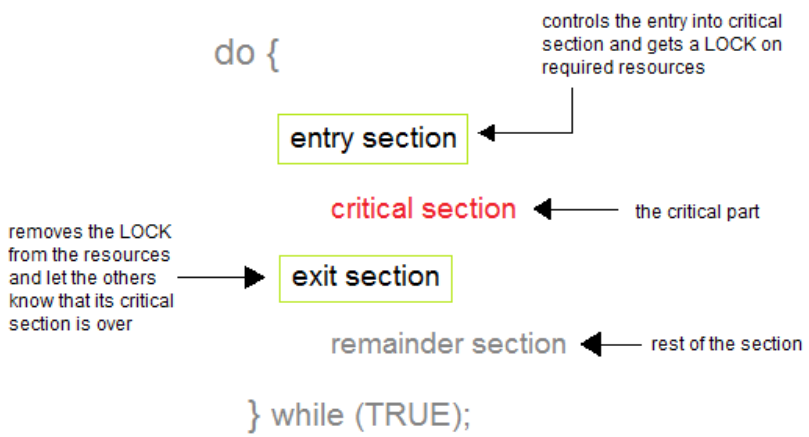
Process synchronization problem arises in the case of Cooperative process also because resources are shared in Cooperative processes.

Race Condition

A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **Race Condition.**

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.



Kundan Thakur

Solution to Critical Section Problem satisfy these criteria :-

1. Mutual Exclusion
2. Progress
3. Bounded Waiting

Mutual Exclusion:

If a process is executing in its critical section, then no other process is allowed to execute in the critical section.

Progress :

If no process is in the critical section, then no other process from outside can block it from entering the critical section.

Bounded waiting :

There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

(Operating system have right to preempt any process or context switch at any point of time but after complete execution of the any instruction of the process. mean(after completion the four face of any instruction - instruction fetch,decode,execute and store).

Peterson's Solution

Peterson's Solution is a classical software based solution to the critical section problem**.** Peterson's Solution preserves all three conditions.

In Peterson's solution, we have two shared variables:

1. boolean flag[i] :Initialized to FALSE, initially no one is interested in entering the critical section.
2. int turn : The process whose turn is to enter the critical section.

```
do {

    flag[i] = TRUE ;
    turn = j ;
    while (flag[j]  &&  turn == j) ;

        critial section

    flag[i] = FALSE ;

        remainder section

} while (TRUE) ;
```

Disadvantages of Peterson's Solution

- It involves Busy waiting
- It is limited to 2 processes.

(busy-waiting,busy-looping or spinning is a technique in which a process repeatedly checks to see if a condition is true, such as whether keyboard input or a lock is available.)

Semaphores

semaphore is a variable which can hold only a non-negative Integer value, shared between the process's, with operations wait and signal, which work as follow:

The classical definitions of **wait** and **signal** are:

- **Wait**: Decrements the value of its argument S, as soon as it would become non-negative(greater than or equal to 1).

```
wait(S)
{
  while(S<=0) ;
  S--;
}
```

- **Signal**: Increments the value of its argument S, as there is no more process blocked on the queue.

```
signal(S)
{
    S++;
}
```

Types of Semaphores

Semaphores are mainly of two types:

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

2. **Counting Semaphores:** They can have any value and are not restricted over a certain domain. They can be used to control access a resource that has a limitation on the number of simultaneous accesses. The semaphore can be initialized to the number of instances of the resource. Whenever a process wants to use that resource, it checks if the number of remaining instances is more than zero, i.e., the process has an instance available. Then, the process can enter its critical section thereby decreasing the value of the counting semaphore by 1. After the process is over with the use of the instance of the resource, it can leave the critical section thereby adding 1 to the number of available instances of the resource.

Classical Problems of Synchronization

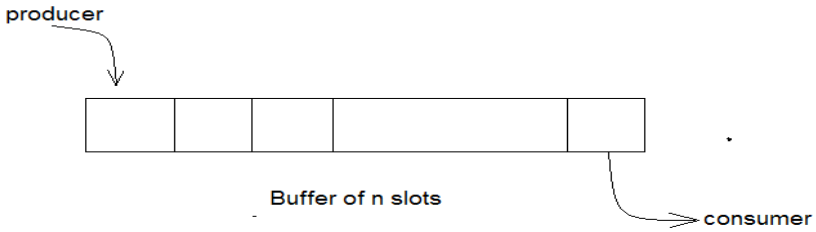
We will discuss the following three problems:

1. Bounded Buffer (Producer-Consumer) Problem
2. The Readers Writers Problem
3. Dining Philosophers Problem

Bounded Buffer Problem(producer consumer problem)

What is the 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.



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,

Here's a 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.

The Producer Operation

```
do
{
    wait (empty);
    wait (mutex) ;  // aquire lock

    // perform the insert operation in a slot

    signal(mutex);  // release lock
    signal(full);
} while (TRUE);
```

The Consumer Operation

```
do
{
    wait (full);
    wait (mutex) ;  // aquire lock

    // perform the remove operation in a slot

    signal(mutex);  // release lock
    signal(empty);
} while (TRUE);
```

Readers Writer Problem

The 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 at that time.

The 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 :

```
do
{
      wait(w);
      // writing is performed
      signal(w);
} while (TRUE);
```

And, the code for the **reader** process looks like this :

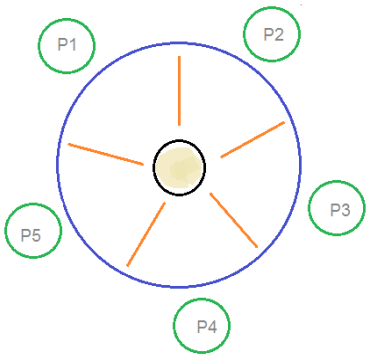
```
do
{
      wait(m);   // acquire lock
      read_count++;
      if(read_count == 1)
         wait(w);
      signal(m);   // release lock

      // perform the reading operation

     wait(m);   // acquire lock
      read_count - -;
     if(read_count == 0)
         signal(w);
      signal(m);   // release lock
} while (TRUE);
```

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.



Kundan Thakur

What is the 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.

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.

Here's the 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.

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.

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

   /* eat */
   signal(stick[i]);

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

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.

Deadlock

Under the normal mode of operation, a process may utilize a resource in only the following sequence:

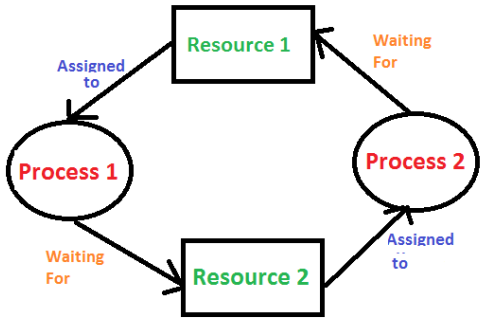
1. **Request-** The process requests the resource. If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.
2. **Use-** The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).
3. **Release-** The process releases the resource.

What is dead lock?

A Deadlock is a situation where each of the computer process waits for a resource which is being assigned to some another process. In this situation, none of the process gets executed since the resource it needs, is held by some other process which is also waiting for some other resource to be released.

Condition for Deadlocks

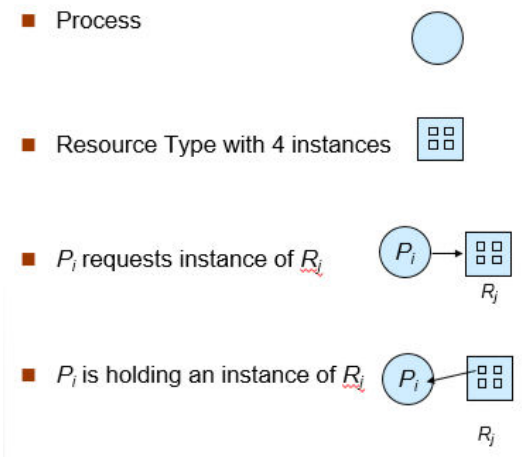
Deadlocks can be avoided by avoiding at least one of the four conditions, because all this four conditions are required simultaneously to cause deadlock



1. **Mutual exclusion.** At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource.
2. **Hold and wait.** A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
3. **No Preemption:** A resource cannot be taken from the process unless the process releases the resource.
4. **Circular wait.** A set { *P0* , Pl, ... , Pn } of waiting processes must exist such that *Po* is waiting for a resource held by P1, P1 is waiting for a resource held by P2, ... , Pn-1 is waiting for a resource held by *Pn* and Pn is waiting for a resource held by *Po*.

Resource-Allocation Graph

G is Graph in which V is vertex. V is partitioned into two types:
$P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
$R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
request edge – directed edge $P_i \rightarrow R_j$
assignment edge – directed edge $R_j \rightarrow P_i$



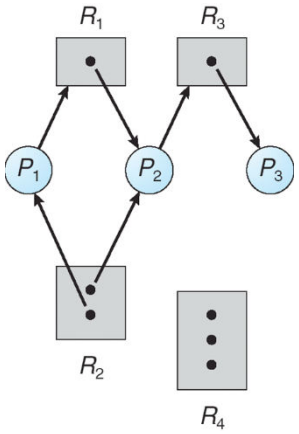
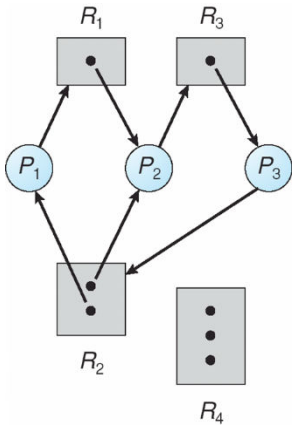


Fig (a)



Fig(b)

If graph contains no cycles ⇒ no deadlock

Kundan Thakur

If graph contains a cycle ⇒
if only one instance per resource type, then deadlock
if several instances per resource type, possibility of deadlock

Methods for Handling Deadlocks

Ensure that the system will ***never*** enter a deadlock state:

1. **Deadlock prevention**
2. **Deadlock avoidance**

Allow the system to enter a deadlock state and then recover.

3. **Deadlock detection and recovery:**

1.Deadlock prevention

if we break one of the condition:-

1. Mutual Exclusion:

Mutual section from the resource point of view is the fact that a resource can never be used by more than one process simultaneously which is fair enough but that is the main reason behind the deadlock. If a resource could have been used by more than one process at the same time then the process would have never been waiting for any resource.

2. Hold and Wait

1. Allocate all required resources to the process before start of its execution, this way hold and wait condition is eliminated but it will lead to low device utilization. for example, if a process requires printer at a later time and we have allocated printer before the start of its execution printer will remained blocked till it has completed its execution.

2. Process will make new request for resources after releasing the current set of resources. This solution may lead to starvation.

3. No Preemption

Preempt resources from process when resources required by other high priority process.

4. Circular Wait

To violate circular wait, we can assign a priority number to each of the resource. A process can't request for a lesser priority resource. This ensures that not a single process can request a resource which is being utilized by some other process and no cycle will be formed.

Each resource will be assigned with a numerical number. A process can request for the resources only in increasing order of numbering.

For Example, if P1 process is allocated R5 resources, now next time if P1 ask for R4, R3 lesser than R5 such request will not be granted, only request for resources more than R5 will be granted.

(NOTE : Possible side effects of preventing deadlocks by this method, however, are low device utilization and reduced system throughput.)

2.Deadlock avoidance

In deadlock avoidance, the request for any resource will be granted if the resulting state of the system doesn't cause deadlock in the system. The state of the system will continuously be checked for safe and unsafe states.

In order to avoid deadlocks, the process must tell OS, the maximum number of resources a process can request to complete its execution.

SAFE AND UNSAFE STATE
A state of the system is called **safe** if the system can allocate all the resources requested by all the processes without entering into deadlock.

If the system cannot fulfill the request of all processes, then the state of the system is called **unsafe**.

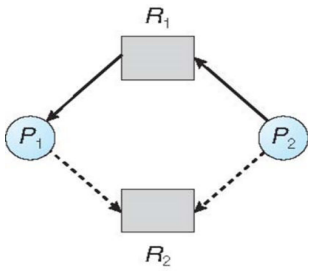
System is in **safe state** if there exists a sequence $<P_1, P_2, ..., P_n>$ of ALL the processes in the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < I$.

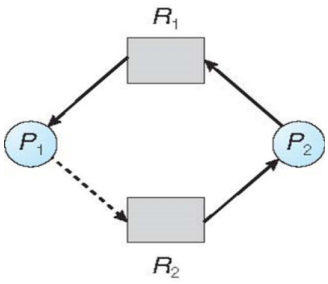
Deadlock Avoidance Algorithms:-

1. Single instance of a resource type
    a. Use a resource-allocation graph
2. Multiple instances of a resource type
    a. Use the banker's algorithm

resource-allocation graph

- Claim edge $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; represented by a dashed line
- Claim edge converts to request edge when a process requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Resources must be claimed *a priori* in the system





We note that the resources must be claimed a priori in the system. That is, before process *Pi* starts executing, all its claim edges must already appear in the resource-allocation graph. We can relax this condition by allowing a claim edge *Pi--> Rj* to be added to the graph only if all the edges associated with process *Pi* are claim edges.

Now suppose that process *P2* requests resource *R2.* The request can be granted only if converting the request edge *P2 --> R2* to an assignment edge *R2--> P2;* does not result in the formation of a cycle in the resource-allocation graph. We check for safety by using a cycle-detection algorithm. An algorithm for detecting a cycle in this graph requires an order of *n^2* operations, where *n* is the number of processes in the system.

If no cycle exists, then the allocation of the resource will leave the system in a safe state. If a cycle is found, then the allocation will put the system in an unsafe state. In that case, process *P1* will have to wait for its requests to be satisfied.

banker's algorithm

Several data structures must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system. We need the following data structures, where *n* is the number of processes in the system and *m* is the number of resource types:

1. **Available**. A vector of length *m* indicates the number of available resources of each type. If *Available[j]* equals *k,* then *k* instances of resource type *Ri* are available.
2. **Max**. An *n* x *m* matrix defines the maximum demand of each process. If *Max[i] [j]* equals *k,* then process *Pi* may request at most *k* instances of resource type Rj.
3. **Allocation**. An *n* x *m* matrix defines the number of resources of each type currently allocated to each process. If *Allocation[i][j]* equals k, then process *Pi* is currently allocated k instances of resource type *Rj*.
4. **Need.** An *n* x *m* matrix indicates the remaining resource need of each process. If *Need[i][j]* equals *k,* then process *Pi* may need *k* more instances of resource type *Ri* to complete its task. Note that
   Need[i][j] = Max[i][j] - Allocation [i][j].

1.Safety Algorithm

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

Work = Available
***Finish* [*i*] = *false* for *i* = 0, 1, …, *n*- 1**

2. Find an ***i*** such that both:

Finish* [*i*] = *false
$Need_i \leq Work$
If no such ***i*** exists, go to step 4

3. $Work = Work + Allocation_i$

Finish*[*i*] = *true
go to step 2

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

2.Resource-Request Algorithm

$Request_i$ = request vector for process P_i.
If $Request_i[j] = k$ then process P_i wants ***k*** instances of resource type R_j

1. If $Request_i \leq Need_i$ *go to step 2. Otherwise,*
   raise error condition, since process has exceeded its maximum claim
2. If $Request_i \leq Available$, *go to step 3. Otherwise*
   P_i *must wait, since resources are not available*
3. Pretend to allocate requested resources to P_i by modifying the state as follows:
   $Available = Available - Request_i;$
   $Allocation_i = Allocation_i + Request_i;$
   $Need_i = Need_i - Request_i;$

- 5 processes P_0 through P_4;
  3 resource types:
  A (10 instances), *B* (5instances), and *C* (7 instances)
- Snapshot at time T_0:

	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

→If safe ⇒ the resources are allocated to P_i
→If unsafe ⇒ P_i must wait, and the old resource-allocation state is restored.

EXAMPLE :-

We have given 5 process and three resource and there **Allocation**, **Max** matrix and **Available** now we need to check the system is in safe state or not.?
First we find **Need** matrix.
And apply safety algo.

- The content of the matrix ***Need*** is defined to be ***Max – Allocation***

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

We get a safe sequence <P1,P3,P4,P0,P2>.
Its not like only these present in system,
There is so many other sequence for this q. depend on implementation.

- The system is in a safe state since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria
- Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ true

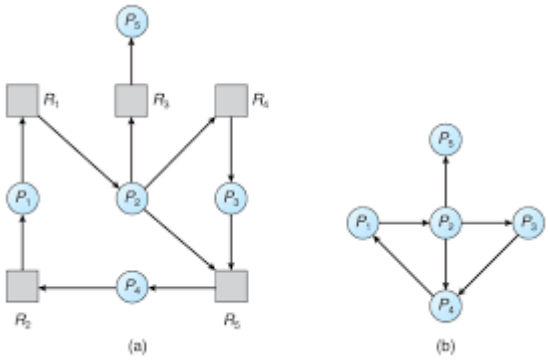
	Allocation	Need	Available
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

Now process process P1 request resource (1,0,2).

Apply resource request algo.
(1,0,2) <= (1,2,2)need
(1,0,2)<= (3,3,2) available
Request accepted

After accepting request we need to check state o system again.

- Executing safety algorithm shows that sequence < P_1, P_3, P_4, P_0, P_2 > satisfies safety requirement
- Can request for (3,3,0) by P_4 be granted?
- Can request for (0,2,0) by P_0 be granted?



3.Deadlock detection and recovery:

1. **Detection**
   - **For Single Instance of Each Resource Type**
     Maintain wait-for graph
     - Nodes are processes
     - $P_i \rightarrow P_j$ if P_i is waiting for P_j

     Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock
     An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.

   - **For Several Instances of a Resource Type**

     Available: A vector of length m indicates the number of available resources of each type
     Allocation: An n x m matrix defines the number of resources of each type currently allocated to each process

Request: An n x m matrix indicates the current request of each process. If $Request[i][j] = k$, then process P_i is requesting k more instances of resource type R_j.

1.Let ***Work*** and ***Finish*** be vectors of length ***m*** and ***n***, respectively Initialize:
(a) ***Work = Available***
(b)For ***i* = 1,2, …, *n***, if ***Allocation*$_i$ ≠ 0**, then
Finish*[i] = *false; otherwise, ***Finish*[i] = *true***
2.Find an index ***i*** such that both:
(a)***Finish*[*i*] == *false***
(b)***Request*$_i$ ≤ *Work***
If no such ***i*** exists, go to step 4

3.***Work = Work + Allocation*$_i$**
Finish*[*i*] = *true
go to step 2
4.If ***Finish[i] == false***, for some ***i***, $1 \le i \le n$, then the system is in deadlock state. Moreover, if, ***Finish*[*i*] == *false***, then ***P*$_i$** is deadlocked.

EXAMPLE:-

- Five processes P_0 through P_4; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances)
- Snapshot at time T_0:

	Allocation	Request	Available
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- Sequence <P_0, P_2, P_3, P_1, P_4> will result in ***Finish[i] = true*** for all ***i***

- P_2 requests an additional instance of type ***C***

	Request
	A B C
P_0	0 0 0
P_1	2 0 2
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

- State of system?
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - one for each disjoint cycle
- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes "caused" the deadlock.

1. **Recovery from Deadlock: Process Termination**
   - Abort all deadlocked processes
   - Abort one process at a time until the deadlock cycle is eliminated
   - In which order should we choose to abort?
     i. Priority of the process
     ii. How long process has computed, and how much longer to completion
     iii. Resources the process has used
     iv. Resources process needs to complete

v. How many processes will need to be terminated.
vi. Is process interactive or batch?

Memory Management

Memory management is the functionality of an operating system which handles or manages primary memory and swap processes between main memory and disk during execution. Memory management keeps track of each and every memory location, regardless of either it is allocated to some process or it is free. It checks how much memory is to be allocated to processes. It decides which process will get memory at what time. It tracks whenever some memory gets freed or unallocated and correspondingly it updates the status.

Static vs Dynamic Loading

The choice between Static or Dynamic Loading is to be made at the time of computer program being developed. If you have to load your program statically, then at the time of compilation, the complete programs will be compiled and linked without leaving any external program or module dependency. The linker combines the object program with other necessary object modules into an absolute program, which also includes logical addresses.

If you are writing a Dynamically loaded program, then your compiler will compile the program and for all the modules which you want to include dynamically, only references will be provided and rest of the work will be done at the time of execution.

At the time of loading, with **static loading**, the absolute program (and data) is loaded into memory in order for execution to start.

If you are using **dynamic loading**, dynamic routines of the library are stored on a disk in relocatable form and are loaded into memory only when they are needed by the program.

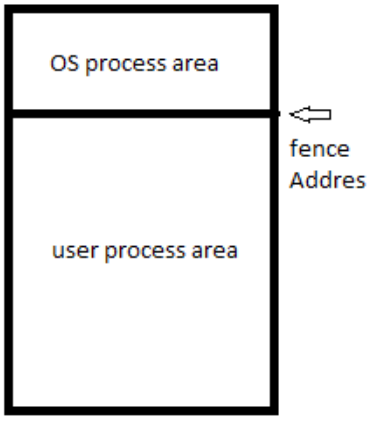
Static vs Dynamic Linking

As explained above, when static linking is used, the linker combines all other modules needed by a program into a single executable program to avoid any runtime dependency.

When dynamic linking is used, it is not required to link the actual module or library with the program, rather a reference to the dynamic module is provided at the time of compilation and linking. Dynamic Link Libraries (DLL) in Windows and Shared Objects in Unix are good examples of dynamic libraries.

Memory Allocation

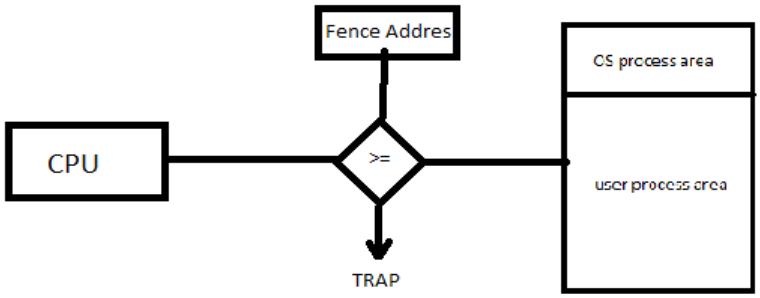
Main memory usually has two partitions −
Low Memory − Operating system resides in this memory.
High Memory − User processes are held in high memory.



In these type of system there is some problem that when the user program start execution it will may be try to access OS area, But it haven't right to access any area which is outside there limit, that may lead to Machin crash. To avoid these problem we must have some memory protection.

whenever the program executed we must ensure that the user program not permitted to access operating system area, but the operating system monitor entire computer area so it will have right to access all part.

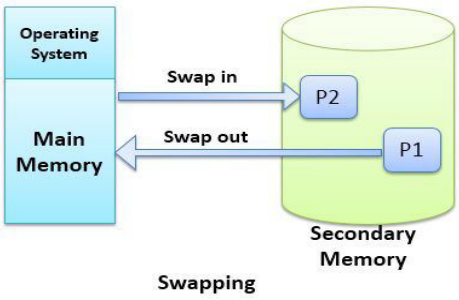
Use of fence address (hardware component):-
Fence address store in hardware and we can't able to update it. Whenever the program execute and try to access the memory CPU generate address, that address check with fence address and it must greater than or equal to the fence address otherwise process access the operating system area and we should interrupt.



Now one more problem arise if we store fence address in hardware. Suppose we have os which take 40KB so we set fence address 40KB. After some time we update os then it will take 60KB. Now 20KB is unprotected these will make a problem. So avoid these problem we take **fence register**.

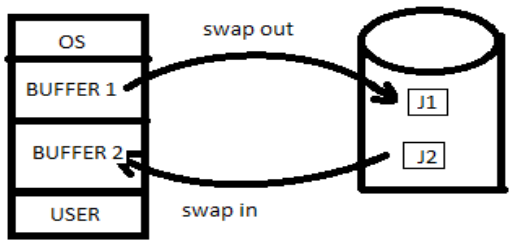
Swapping:-

For execution, each process must be placed in the main memory. When we need to execute a process, and the main memory is entirely full, then the **memory manager swaps** a process from main memory to backing store by evacuating the place for the other processes to execute. The memory manager swaps the processes so frequently that there is always a process in main memory ready for execution.



Due to **address binding** methods, the process that is swapped out of main memory occupies same address space when it is swapped back to the main memory if the binding is done at the assembly or load time. If the binding is done at execution time, the process can occupy any available address space in main memory as addresses are computed at the execution time.

Now in these technique we have spent more time in swapping the process and at that time our CPU have no work it is in ideal state. So in next memory model we divide the memory into four part. One user part that execute the process one buffer1 that contain the process which swap out, one buffer2 that contain the next process for execution.



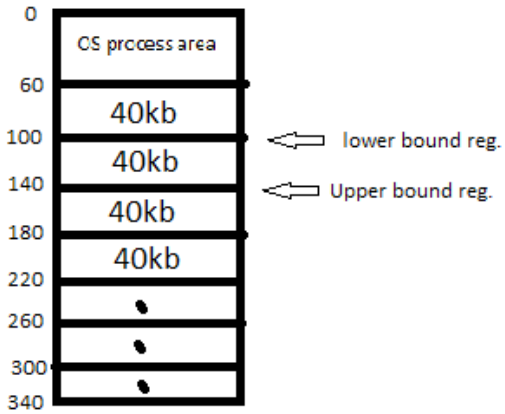
Memory allocation technique

First fit: Allocate the *first* hole that is less than or equal to our process size.
Best fit: Allocate the *smallest* hole that is big enough. We must search the entire list, unless the list is ordered by size.
Worst fit: Allocate the *largest* hole. Again, we must search the entire list, unless it is sorted by size.

(Note: Both the first-fit and best-fit strategies for memory allocation suffer from External fragmentation.)

MFT(multiprogramming with fix number of task):

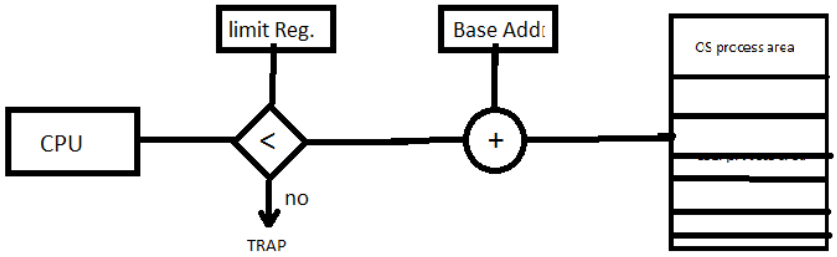
Memory is divided into several fixed-size partitions. Each partition may contain exactly one process/task.
Now at the same time we execute many process concurrently in the CPU.



For memory protection or unwanted access of memory we use two register one **lower bound register**, and **upper bound register**. But in these technique we need to compare two time. Lets optimize its, now in place of lower bound register, and upper bound register, we use two register.

I. **Limit register:** it will contain size of the block of current process.
II. **Base address:** it will contain base address of the block of current process.

Now CPU generate address, first we check it greater then limit registeror or not. If it is greater then we will trap, otherwise we add the base address in it and permit to access the memory.



Partition allocation table is contain all the detail about the partition like base address, size of block, block is empty or contain any process etc.

NOW WE UNDERSTAND WHAT IS FRAGMENTATION:

Suppose we have 2 process and we use FCFS scheduling and First fit memory allocation technique.
Now process size
p1 30kb
p2 100kb

according to First fit technique p1 store in second block 40kb but process need only 30kb. There is 10kb of memory lost is situation called **internal fragmentation**.

For process p2 in main there in enough memory for p2, but we are not able to store it in the main memory because the memory is divide into into small part these situation called **external fragmentation**.

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**.

- 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.
- 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 noncontiguous i.e. the memory is fragmented into large no. Of small holes. This phenomenon is known as **External Fragmentation**.

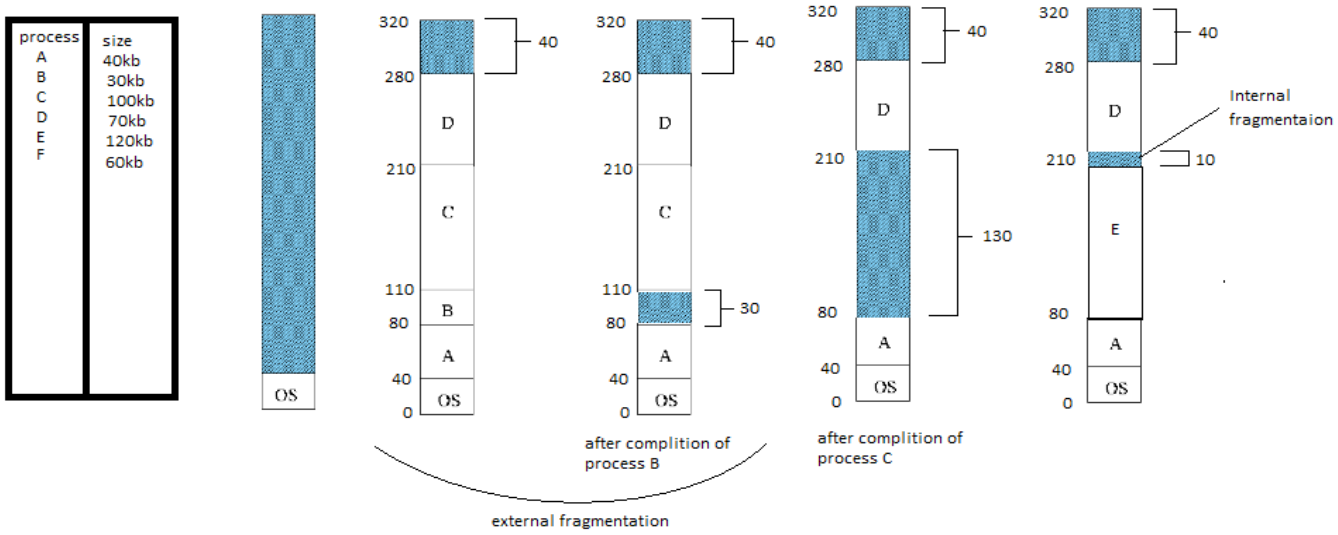
MVT(multiprogramming with variable number of task):

In variable partitioning scheme there are no partitions at the beginning. There is only the OS area and the rest of the available RAM. The memory is allocated to the processes as they enter. This method is more flexible as there is less chance of internal fragmentation and there is no size limitation.

- Both the number and size of partition change with respect to time
- job still has only one segment.
- A single ready list.
- Job can be move (might be swapped back in different place).
- This is dynamic address translation.

- **Eliminate internal fragmentation**
- find a region the exact right size.

- **Introduces external fragmentation**
- what do you do if no hole is big enough for the request.
- Can compact memory- transition from free space at the bottom. This is expensive. Not suitable for real time system.
- Can we swap one process to another so may be continuous space will be made.



MFT and MVT are different memory management techniques: -

MFT or fixed partitioning scheme

1. The OS is partitioned into fixed sized blocks at the time of installation. For example, there can be total 4 partitions and the size of each block can be 4KB. Then the processes which require 4KB or less memory will only get the memory.
2. It is possible to bind address at the time of compilation.
3. It is not flexible because the number of blocks cannot be changed.
4. There can be memory wastage due to fragmentation.

MVT or variable partitioning scheme

1. No partitioning is done at the beginning.
2. Memory is given to the processes as they come.
3. This method is more flexible.
4. Variable size of memory can be given as there is no size limitation
5. There is may be internal fragmentation. But there can be external fragmentation.
6. Compile time address binding cannot be done

Memory Compaction:

Paging

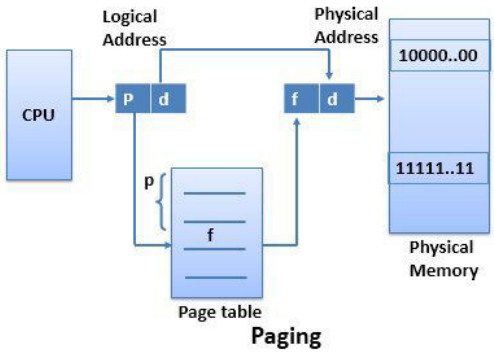
Paging is a memory management scheme, which allots a **noncontiguous address space** to a process. Now, when a process's physical address can be non-contiguous the problem of **external fragmentation** would not arise.

Paging is implemented by breaking the **main memory** into fixed-sized blocks that are called **frames**. The **logical memory of a process** is broken into the same fixed-sized blocks called **pages**. The page size and frame size is defined by the hardware. As we know, the process is to be placed in main memory for execution. So, when a process is to be executed, the pages of the process from the source i.e. back store are loaded into any available frames in main memory.

Kundan Thakur

There is may be external fragmentation when the memory is not an integer multiple of fix frame size. In that condition we can say, in worst case the internal fragmentation is (frame_size – 1).

Now let us discuss how paging is implemented. CPU generates the logical address for a process which consists of two parts that are **page number** and the **page offset**. The page number is used as an **index** in the **page table**.



Paging

The page table contains the **base address** of each page that loaded in main memory and map into the any empty frame. Then we combine the frame and offset to get correct frame.

Every operating system has its own way of storing page table. Most of the operating system has a separate page table for each process.

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.

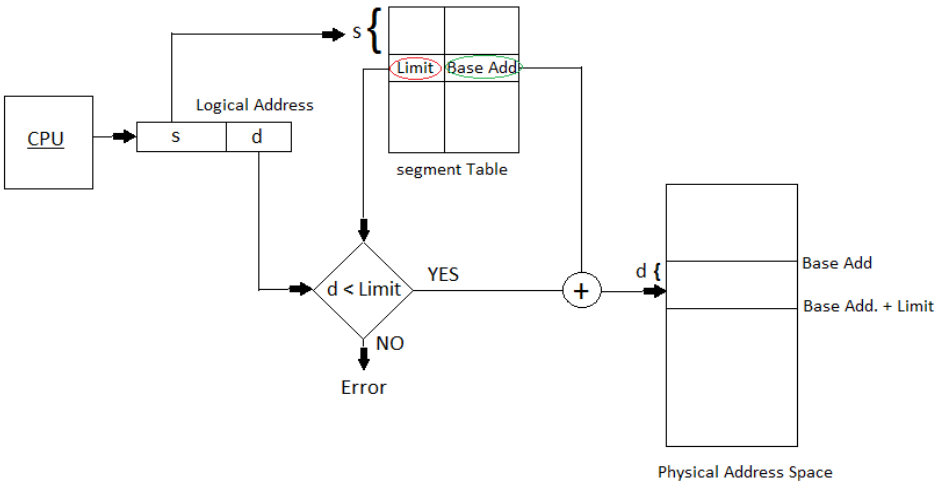
A Memory Management technique in which memory is divided into variable sized chunks which can be allocated to processes. Each chunk is called a Segment. A table stores the information about all such segments and is called Segment Table.

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

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

Address generated by the CPU is divided into:

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



Why Segmentation is required?

Till now, we were using Paging as our main memory management technique. Paging is more close to Operating system rather than the User. It divides all the process into the form of pages regardless of the fact that a process can have some relative parts of functions which needs to be loaded in the same page.

Operating system doesn't care about the User's view of the process. It may divide the same function into different pages and those pages may or may not be loaded at the same time into the memory. It decreases the efficiency of the system.

It is better to have segmentation which divides the process into the segments. Each segment contain same type of functions such as main function can be included in one segment and the library functions can be included in the other segment.

Advantages of Segmentation –

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

Disadvantage of Segmentation –

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

Segmentation with Paging:-

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

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

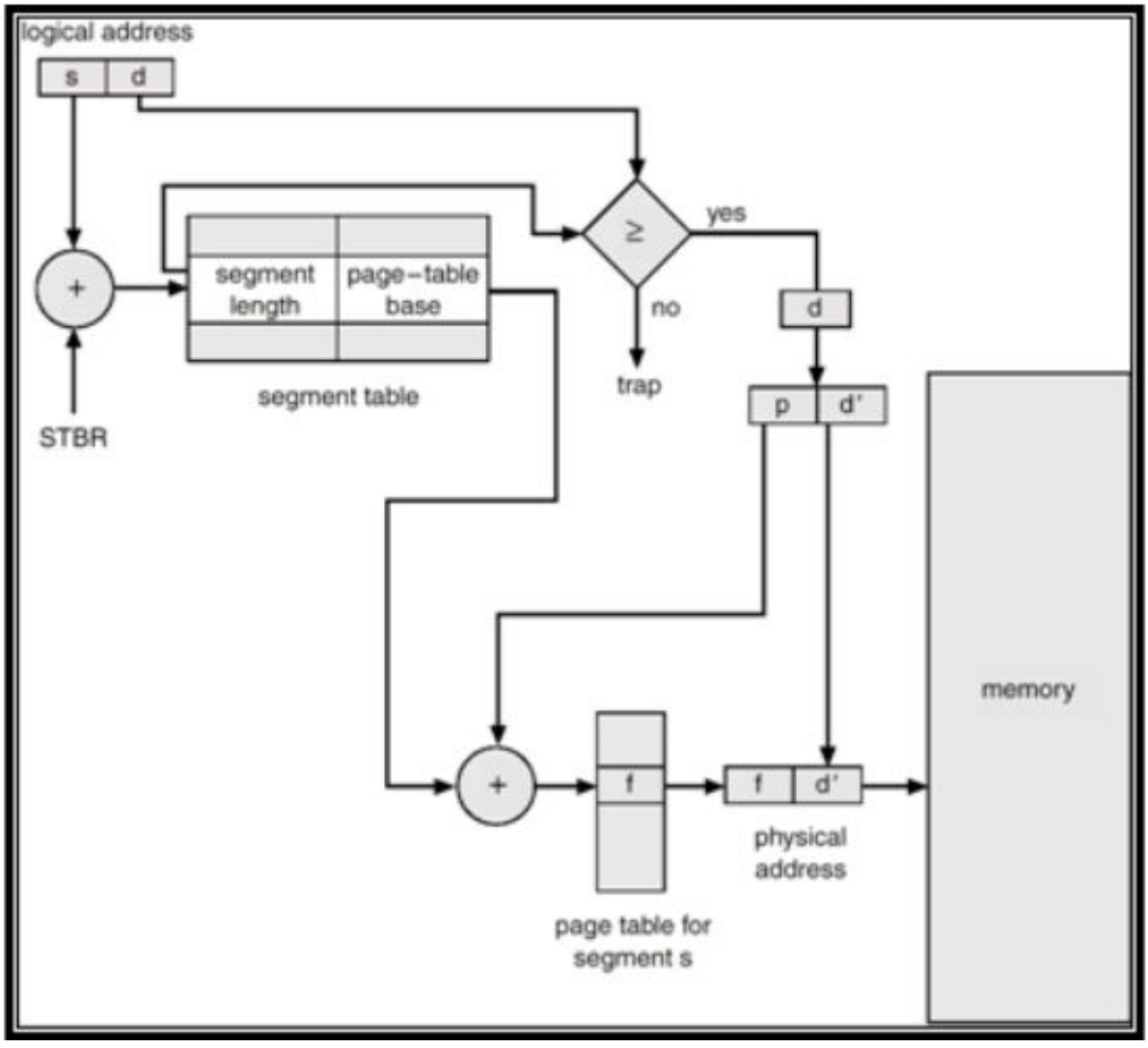
1. Pages are smaller than segments.
2. Each Segment has a page table which means every program has multiple page tables.
3. The logical address is represented as Segment Number (base address), page offset.

Segment Number → It points to the appropriate Segment Number.
Page Offset → Used as an offset within the page frame

Each Page table contains the various information about every page of the segment. The Segment Table contains the information about every segment. Each segment table entry points to a page table entry and every page table entry is mapped to one of the page within a segment.

Translation of logical address to physical address

The CPU generates a logical address which is divided into two parts: Segment Number and Segment Offset. The Segment Offset must be less than the segment limit. Offset is further divided into Page number and Page Offset. To map the exact page number in the page table, the page number is added into the page table base. The actual frame number with the page offset is mapped to the main memory to get the desired word in the page of the certain segment of the process.



Advantages of Segmented Paging

1. It reduces memory usage.
2. Page table size is limited by the segment size.
3. Segment table has only one entry corresponding to one actual segment.
4. External Fragmentation is not there.
5. It simplifies memory allocation.

Disadvantages of Segmented Paging

1. Internal Fragmentation will be there.
2. The complexity level will be much higher as compare to paging.
3. Page Tables need to be contiguously stored in the memory.

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.

Virtual Memory is a storage scheme that provides user an illusion of having a very big main memory. This is done by treating a part of secondary memory as the main memory.

In this scheme, User can load the bigger size processes than the available main memory by having the illusion that the memory is available to load the process.

Instead of loading one big process in the main memory, the Operating System loads the different parts of more than one process in the main memory.

By doing this, the degree of multiprogramming will be increased and therefore, the CPU utilization will also be increased

How Virtual Memory Works?

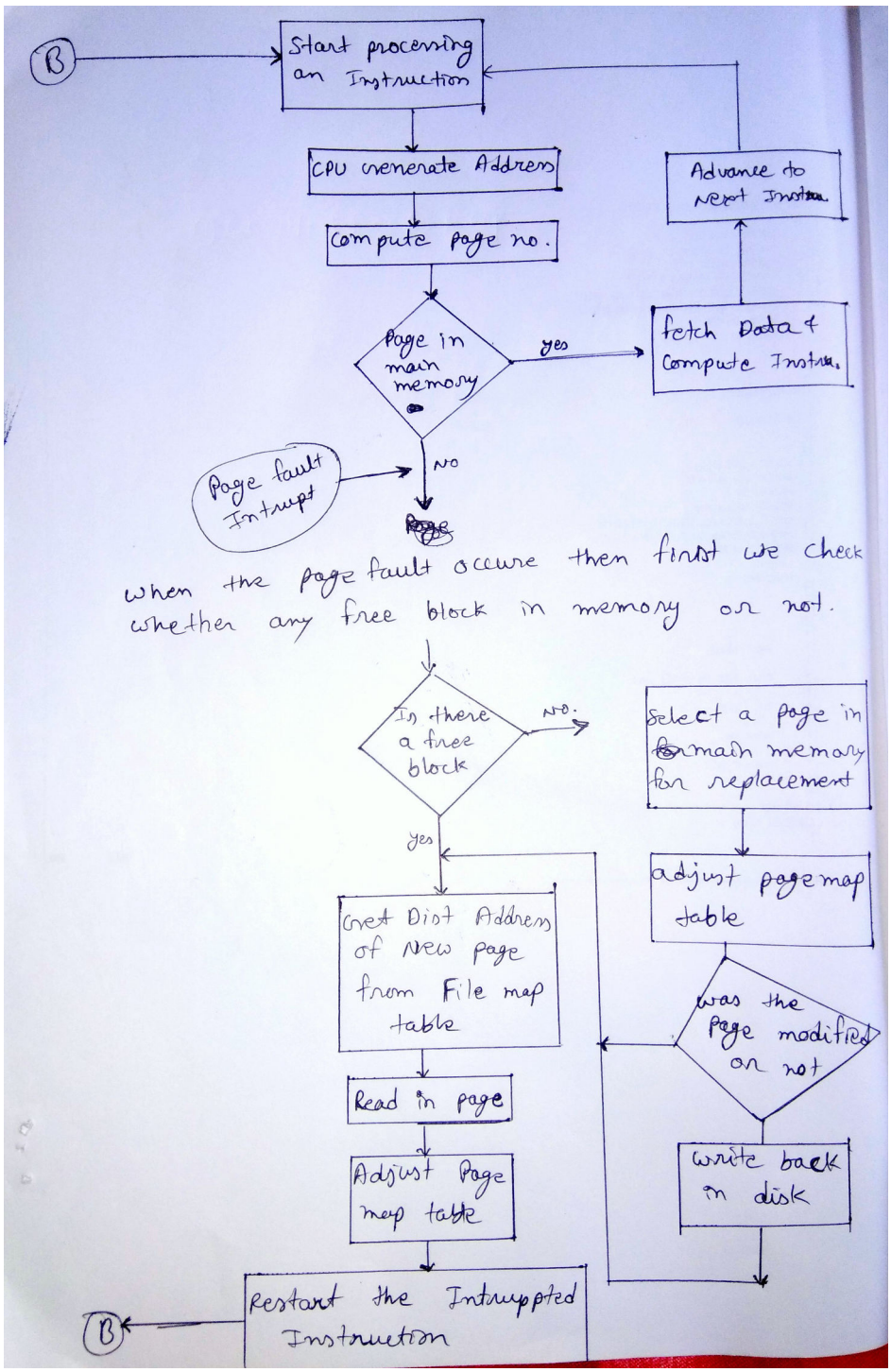
In modern word, virtual memory has become quite common these days. In this scheme, whenever some pages needs to be loaded in the main memory for the execution and the memory is not available for those many pages, then in that case, instead of stopping the pages from entering in the main memory, the OS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

Since all this procedure happens automatically, therefore it makes the computer feel like it is having the unlimited RAM.

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.

When a process incurs a page fault, a **local page replacement** algorithm selects for replacement some page that belongs to that same process (or a group of processes sharing a memory partition). A global replacement algorithm is free to select any page in memory.

Global page replacement involves competition between processes as there are a limited number of frames. This results in more page faults.



Advantages of Virtual Memory

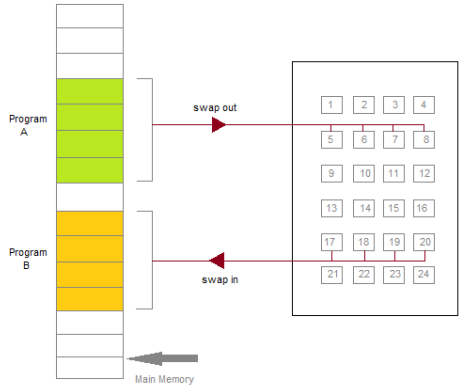
1. The degree of Multiprogramming will be increased.
2. User can run large application with less real RAM.
3. There is no need to buy more memory RAMs.

Disadvantages of Virtual Memory

1. The system becomes slower since swapping takes time.
2. It takes more time in switching between applications.
3. The user will have the lesser hard disk space for its use.

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 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.

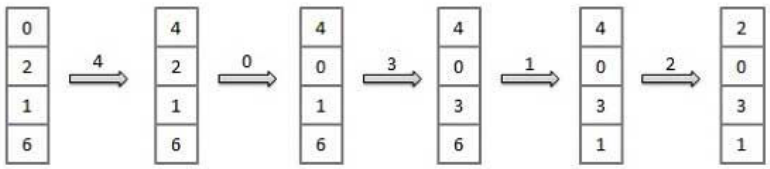
A page replacement algorithm looks at the limited information about accessing the pages provided by hardware, and tries to select which pages should be replaced to minimize the total number of page misses, while balancing it with the costs of primary storage and processor time of the algorithm itself. 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.

First In First Out (FIFO) algorithm

- 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.

Reference String : 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1

Misses : x x x x x x x x x



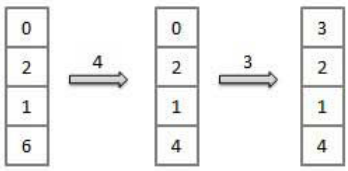
Fault Rate = 9 / 12 = 0.75

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.

Reference String : 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1

Misses : x x x x x x



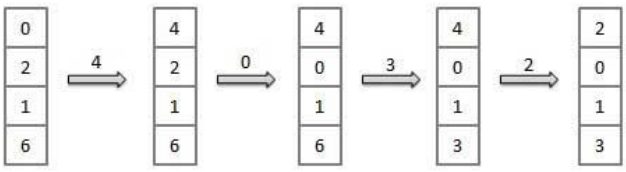
Fault Rate = 6 / 12 = 0.50

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.

Reference String : 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1

Misses : x x x x x x x x



Fault Rate = 8 / 12 = 0.67

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.

Thrashing:
Now if it happens that your system has to swap pages at such a higher rate that **major chunk of CPU time is spent in swapping** then this state is known as thrashing. So effectively during thrashing, the CPU spends less time in some actual productive work and more time in swapping.

Now the effects of thrashing and also the extent to which thrashing occurs will be decided by the type of page replacement policy.

1. **Global Page Replacement**: The paging algorithm is applied to all the pages of the memory regardless of which process "owns" them. A page fault in one process may cause a replacement from any process in memory. Thus, the size of a partition may vary randomly.

2. **Local Page Replacement**: The memory is divided into partitions of a predetermined size for each process and the paging algorithm is applied independently for each region. A process can only use pages in its partition.

What happens after Thrashing starts?
If **global** page replacement is used, situations worsens very quickly. CPU thinks that CPU utilization is decreasing, so it tries to increase the degree of multiprogramming. Hence bringing more processes inside memory, which in effect increases the thrashing and brings down CPU utilization further down. The CPU notices that utilization is going further down, so it increases the degree of multiprogramming further and the

cycle continues.

The solution can be **local** page replacement where a process can only be allocated pages in its own region in memory. If the swaps of a process increase also, the overall CPU utilization does not decrease much. If other transactions have enough page frames in the partitions they occupy, they will continue to be processed efficiently.

What is Belady's Anomaly?
Bélády's anomaly is an anomaly with some page replacement policies where increasing the number of page frames results in an increase in the number of page faults. It occurs with First in First Out page replacement is used.

Differences between mutex and semphore?