Process and Communication

A process is a program in execution. A process needs CPU time, Memory, Files and I/O devices etc and these resources are allocated to it either when it is created or when it is executing. A process is not the same as a program. A process is more than a program code. A process is an active entity as opposite to a program which is considered to be a passive entity. As we all know that a program is an algorithm expressed in some suitable notation, (e.g., programming language). Being passive, a program is only a part of process.

A Process, includes

- Current value of Program Counter (PC)
- Contents of the processors registers
- Value of the variables
- The process stack (SP) that typically contains temporary data such as subroutine parameter, return address, and temporary variables.
- A data section that contains global variables.

A process is a unit of work in most systems. Operating System processes execute system code, while user processes execute user code. For creation and deletion of user and system processes, Operating System is responsible for the scheduling of processes, communication, synchronization and deadlock handling for processes.

Process State

A program by itself is not a process whereas a process is a program in exec ution that has its own program counter and internal states. A process has the following five states.

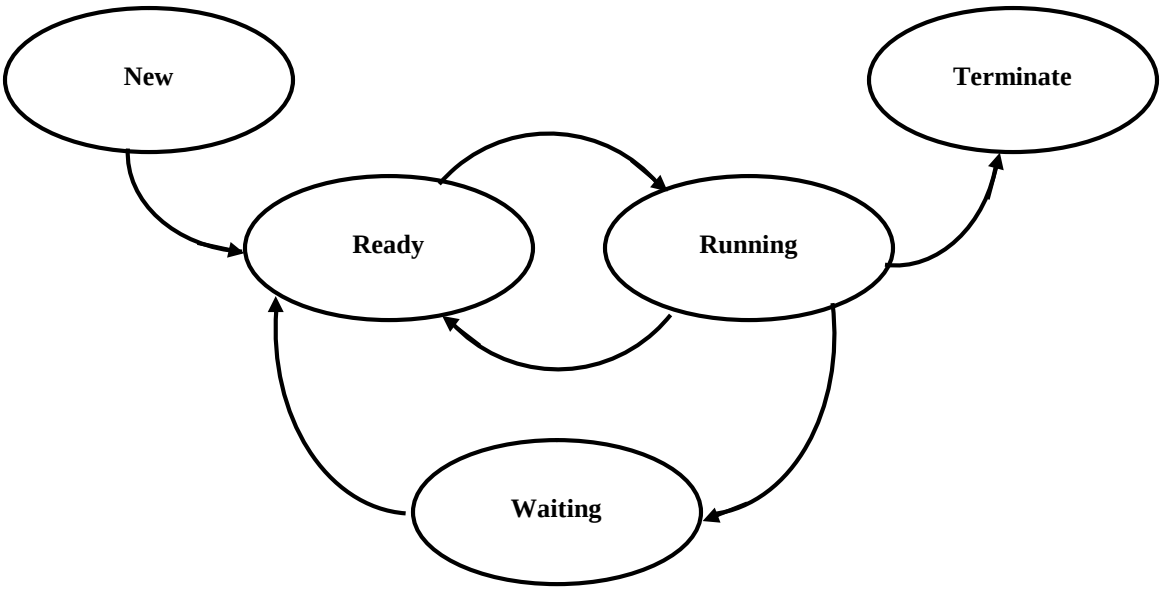
1. New The process being created.

2. Ready A process is ready to be executed and waiting for the CPU time.

3. Running A process is said to be running if it has the CPU, that is, process actually using the CPU at that particular instant.

4. Waiting A process is said to be blocked/waiting if it is waiting for some event to happen such that as an I/O completion before it can proceed.

5. Terminate The process has finished execution.



Process States

Concurrent Processes

Many processes can be Multitasked on a CPU so that they can be executed concurrently. So executing processes in a time-shared Operating System is called concurrent processing.

Reasons for concurrent processing are

- **Physical Resource Sharing** As hardware resources are limited, so we share them in a multi user environment.

- **Logical Resource Sharing** If many user are in need of sharing the same information i.e. a file etc, then we should provide an environment to allow concurrent access to such resources.

- **Speedup the Computation** In order to make a task run faster, we can break it into subtasks and will execute all the subtasks at the same time. Such kind of speed can be achieved in Multiprocessing.

- **Modularity** A system can be constructed in a modular design i.e. the functions of the system may be divided into different modules or dividing the system functions into separate processes.

- **Convenience** An individual user may want to perform many tasks at one time i.e. Editing, Printing and Compiling etc.

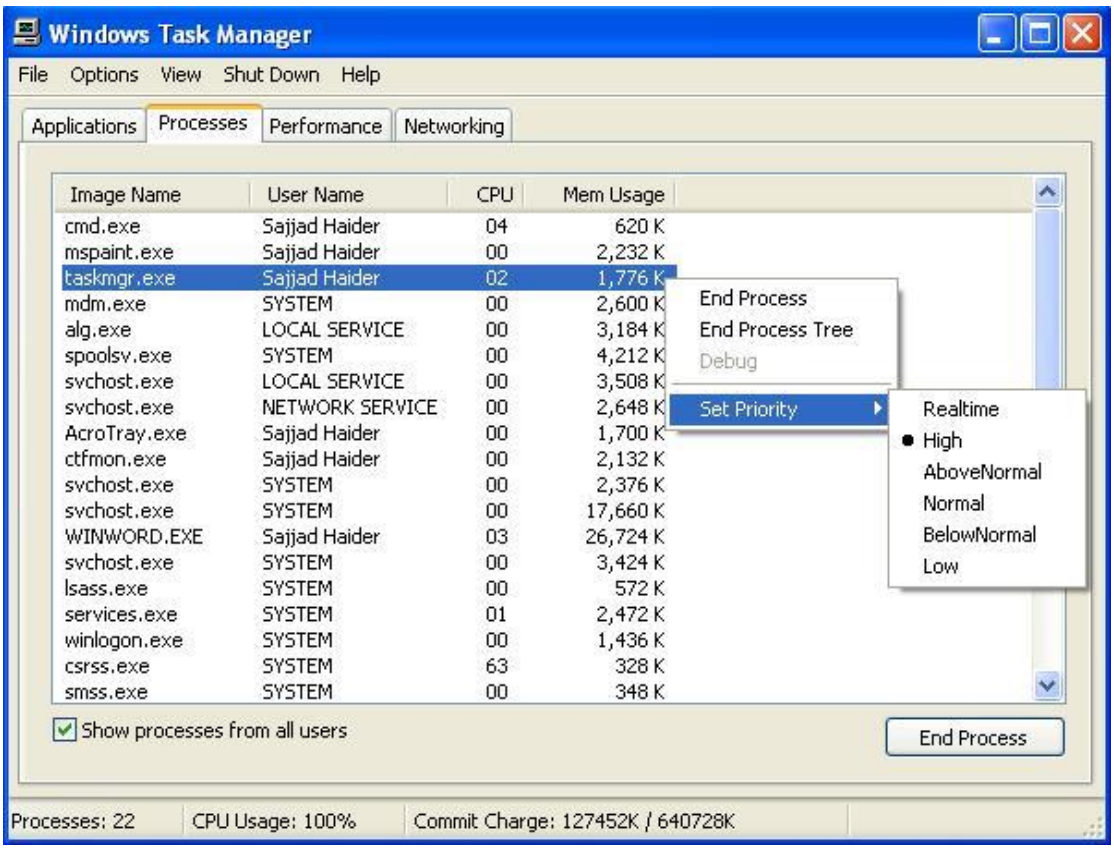
Concurrent execution that requires cooperation among the processes also requires a mechanism for process synchronization and communication.

Process Control Block (PCB)

Every process in Operating System has its Process or Task Control Block. A Process Control Block or PCB is basically a record containing different kinds of information about a specified process.

Different kinds of information contained in a PCB are

- **Process State** Ready, Running or Waiting
- **Program Counter** Indicating the address of the next instruction that will be executed for the process.
- **Accounting Information** includes the amount of CPU and real time used. Similarly information about job or process number etc.



- **Registers** depend upon the architecture of the system. Registers include accumulators, Index registers, stack pointers and general purpose registers etc.

In a computer, A Register is one of a small set of data holding places that are part of a computer microprocessor. Registers provide a place for passing data from one instruction to the next sequential instruction or to another program that the operating system has just given control to. A register must be large enough to hold an instruction - for example, in a 32-bit instruction computer, a register must be 32 bits in length. In some computer designs, there are smaller registers - for example, *half-registers* - for shorter instructions. Depending on the processor design and language rules, registers may be numbered or have arbitrary names.

- **Scheduling Information** contains information about the priority of the process etc.

- **Memory Management Information** contains information about limit registers or Page tables etc.

- **I/O Status** contains information about the requests and usage of I/O devices etc.

Pointer	Process States
Process Number	
Program Counter	
Registers	
Memory Limits	
List of Open files	
….	

Process Control Block

Independent and Cooperating Processes

The processes executing in the Operating System may be either Independent Processes or Cooperating Processes.

Independent Process A process is Independent if it can not affect or be affected by the other processes executing in the system.

An Independent process has the following characteristics.

- Its state is not shared in any way by the other process.
- Its execution is deterministic i.e. the result of the execution completely depends upon the state of Input.

- Its execution is reproducible, i.e. the result of execution will always be the same for that same Input.
- Its execution can be stopped and restarted without causing ill effects.

So, a process that does not share any data etc with other processes is an Independent process.

Cooperating Process A process is cooperating if it can affect or be affected by the other processes executing in the system.

Cooperating process has the following characteristics.

- State of cooperating process is shared among other processes.
- The result of its execution can not be predicted in advance, because it depends on the relative execution sequence.
- The result of execution is non deterministic as it will not always be the same for the same input.

So, a process that shares data with other processes is a Cooperating process. Cooperating processes may either directly share a logical address space (code and data) or may be allowed to share data through files.

Threads or Light Weight Processes are used in order to achieve the sharing of logical address space (code and data) in Cooperating processing.

Hierarchy of Process

A process can create several new processes. The process that creates processes is called Parent process and the newly created processes are called children of that process. Similarly, the newly created processes can also create other processes, so we get a tree of processes.

Inter Process Communication or IPC

Processes communicate with each other so, the issues related to the communication of processes is called Inter Process Communication or IPC.

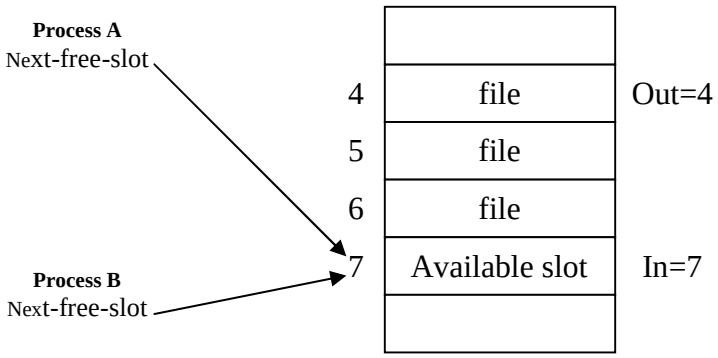
Race Condition

In some operating systems, multiple processes use shared resources (Memory, storage or printer etc) for reading and writing.

Race condition is situation where two or more processes are reading or writing some shared data and final result depends on who runs exactly when.

An operating system is providing printing services, “Printer System Process ” picks a print job from SPOOLer. Each job gets a job number, and this is done by the SPOOLer using two global variables “In” and “Out”.

Two process, A & B need printing services, both of these process have their own local variables named “next-fee-slot”.



- Process A found from “In” that slot 7 is free, so it recorded this number in its variable i.e. next free slot = 7.
- Process A consumed its time and scheduler now gives CPU to Process B.
- Process B has some need so it also recorded next free slot = 7. Process B submitted the job and added 1 in its local variable and placed the value in global variable i.e. now “In” = 8.
- Process A comes back for execution and starts from where it was proceeded out. As this time it has 7 in its local variable, so it will submit the job in slot number 7, overwriting the job submitted by process B and will again overwrite the global variable too i.e. “In = 8”.

Mutual Exclusion

Mutual Exclusion forces all processes to avoid the use of shared variable, if in use by one process. In order to avoid race conditions there should be a way that if one process is using a shared variable etc then other processes should not be allowed to use the same shared variable. If a process is using a shared variable then other processes will be excluded from using same-shared variable and this is called the Mutual Exclusion.

Critical Section

The problem occurred in the above example because process B started using the shared variable before process A was finished with its. So, the part of a program where shared memory is accessed is called the Critical Section. If we arrange that two processes will not be in their critical section at the same time, then we can avoid Race Condition.

Various techniques are used for achieving Mutual Exclusion so that when one process is busy updating shared memory in its critical region, no other process can enter its region and cause problems.

Disabling Interrupts

Each process is allowed to disable all the Interrupts immediately after it has entered the critical section, and re-enable the interrupts just before exiting the critical section.

Problem here is that if a process disable interrupts and due to any reason, didn't enable them, then it will cause a complete halt of the system.

Lock Variables

Here we have a single shared lock variable and its initial value is set to 0. 0 means Resource is available. Process sets it to 1 and enters the critical section. Another process wants to use the resource but lock variables value is 1, so the process will have to wait for the value to be 0.

Problem here is that if a process reads the lock as 0 but before setting it to 1. CPU scheduled another process that sets lock to 1. So when the first process runs again, it will also set the lock to 1 and two processes will be in their critical regions at the same time.

The Producer-Consumer or Bounded Buffer Problem

Two processes share a common buffer. One is producer, which puts information into the buffer, and the other one is consumer that takes it out.

Producer wants to put a new item in the Buffer, but it is already full, then the producer should go to sleep and awakened by the consumer on removing one or more items. Similarly, consumer wants to take items but buffer is empty, so it goes to sleep until the producer puts something in the buffer and wakes it up.

To check the number of items in the buffer we need a variable, Count. If the maximum number of items, the buffer can hold is N, the producer will first test if count is N, if it is, the producer will got to sleep, if it is not, producer will add an item and increments counter.

Similarly consumer tests count to see if it is 0. If it is, it goes to sleep, if it is not 0, remove an item and decrement the counter.

Problem here is that if buffer is empty and consumer finds the count to 0, at that time if scheduler starts running the producer, producer enters the item in the buffer, increments count and sends a wakeup call to consumer. Consumer was not sleeping so the wakeup signal is lost. Similarly, when consumer runs again, it read previously that count is 0 although count now is 1, but it will sleep, as it knows that the count is 0. Like this producer will fill up the buffer and will also go to sleep and both will sleep forever.

Semaphore

A Semaphore 'S' is an integer variable that other initialization, is accessed only through two standard atomic operations i.e. wait and signal.

So, when Semaphore = 0 it means 'No wakeups' i.e. no job is pending and
When Semaphore > = then it means 'Pen ding wakeups' i.e. jobs are pending

Operations of Semaphore

A Semaphore can have two operations that are called

i) **Wait** or **Sleep** or **Down**
ii) **Signal** or **Wakeup** or **Up**

Wait Operation

- tests semaphore for value > 0.
- if true, then decrement the value and cont inue the processing.
- (Decrements means one process has consumed one wakeup turn).
- if = 0 then process is put to sleep.

Signal Operation

- Increment the value of semaphore.
- If more than one process is sleeping then pick one randomly.

In programming, especially in UNIX systems, semaphores are a technique for coordinating or synchronizing activities in which multiple process compete for the same operating system resources. A semaphore is a value in a designated plac e in operating system (or kernel) storage that each process can check and then change. Depending on the value that is found, the process can use the resource or will find that it is already in use and must wait for some period before trying again. Semaphon es can be binary (0 or 1) or can have additional values. Typically, a process using semaphores checks the value and then, if it using the resource, changes the value to reflect this so that subsequent semaphore users will know to wait. Semaphores are comm only use for two purposes: to share a common memory space and to share access to files. Semaphores are one of the techniques for interprocess communication (interprocess communication). The C/C++ programming language provides a set of interfaces or "functi ons" for managing semaphores.