|  |  |  |
| --- | --- | --- |
|  | **Process** | **Thread** |
| Definition | An executing instance of a program is called a process. | A thread is a subset of the process. |
| Process | It has its own copy of the data segment of the parent process. | It has direct access to the data segment of its process. |
| Communication | Processes must use inter-process communication to communicate with sibling processes. | Threads can directly communicate with other threads of its process. |
| Overheads | Processes have considerable overhead. | Threads have almost no overhead. |
| Creation | New processes require duplication of the parent process. | New threads are easily created. |
| Control | Processes can only exercise control over child processes. | Threads can exercise considerable control over threads of the same process. |
| Changes | Any change in the parent process does not affect child processes. | Any change in the main thread may affect the behavior of the other threads of the process. |
| Memory | Run in separate memory spaces. | Run in shared memory spaces. |
| File descriptors | Most file descriptors are not shared. | It shares file descriptors. |
| File system | There is no sharing of file system context. | It shares file system context. |
| Signal | It does not share signal handling. | It shares signal handling. |
| Controlled by | Process is controlled by the operating system. | Threads are controlled by programmer in a program. |
| Dependence | Processes are independent. | Threads are dependent. |

A process is an instance of a program that is being executed. It contains the program code and its current activity. Depending on the operating system, a process may be made up of multiple threads of execution that execute instructions concurrently. A program is a collection of instructions; a process is the actual execution of those instructions.

A process has a self-contained execution environment. It has a complete set of private basic run-time resources; in particular, each process has its own memory space. Processes are often considered similar to other programs or applications. However, the running of a single application may in fact be a set of cooperating processes. To facilitate communication between the processes, most operating systems use Inter Process Communication (IPC) resources, such as pipes and sockets. The IPC resources can also be used for communication between processes on different systems. Most applications in a virtual machine run as a single process. However, it can create additional processes using a process builder object.

In computers, a thread can execute even the smallest sequence of programmed instructions that can be managed independently by an operating system. The applications of threads and processes differ from one operating system to another. However, the threads are made of and exist within a process; every process has at least one. Multiple threads can also exist in a process and share resources, which helps in efficient communication between threads.

On a single processor, multitasking takes place as the processor switches between different threads; it is known as multithreading. The switching happens so frequently that the threads or tasks are perceived to be running at the same time. Threads can truly be concurrent on a multiprocessor or multi-core system, with every processor or core executing the separate threads simultaneously.

In summary, threads may be considered lightweight processes, as they contain simple sets of instructions and can run within a larger process. Computers can run multiple threads and processes at the same time.

**Process Scheduling**

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

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

Schedulers fell into one of the two general categories:

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

**Scheduling Queues**

* All processes when enters into the system are stored in the **job queue**.
* Processes in the Ready state are placed in the **ready queue**.
* Processes waiting for a device to become available are placed in **device queues**. There are unique device queues for each I/O device available.

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, once of several events could occur.

* The process could issue an I/O request, and then be placed in an 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:

**Long Term Scheduler** :

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

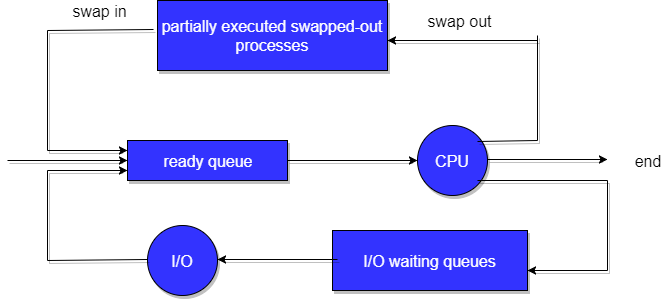
**Short Term Scheduler** :

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

**Medium Term Scheduler** :

This scheduler removes the processes from memory (and from active contention for the CPU), and thus reduces the degree of multiprogramming. At some later time, the process can be reintroduced into memory and its execution can be continued where it left off. This scheme is called **swapping**. The process is swapped out, and is later swapped in, by the medium term scheduler.

Swapping may be necessary to improve the process mix, or because a change in memory requirements has overcommitted available memory, requiring memory to be freed up. This complete process is descripted in the below diagram:



**Context Switch**

* Switching the CPU to another process requires **saving** the state of the old process and **loading** the saved state for the new process. This task is known as a **context switch**.
* The **context** of a process is represented in the **Process Control Block(PCB)** of a process; it includes the value of the CPU registers, the process state and memory-management information. When a context switch occurs, the Kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run.
* Context switch time is **pure overhead**, because the **system does no useful work while switching**. Its speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions (such as a single instruction to load or store all registers). Typical speeds range from 1 to 1000 microseconds.
* Context Switching has become such a performance **bottleneck** that programmers are using new structures (threads) to avoid it whenever possible.

**OPERATIONS ON PROCESSES**

**Process Creation**

Parent process create children processes, which, in turn create other processes, forming a tree of processes

**Resource sharing**  
- Parent and children share all resources  
- Children share subset of parent’s resources  
- Parent and child share no resources

**Execution**  
- Parent and children execute concurrently  
- Parent waits until children terminate

**Address space**  
- Child duplicate of parent  
- Child has a program loaded into it

**UNIX examples**- **fork** system call creates new process  
- **exec** system call used after a **fork** to replace the process’ memory space with a new program

**Process Termination**

Process executes last statement and asks the operating system to delete it (**exit**)  
- Output data from child to parent (**via wait**)  
- Process’ resources are deallocated by operating system

Parent may terminate execution of children processes (**abort**)  
- Child has exceeded allocated resources  
- Task assigned to child is no longer required  
- If parent is exiting  
- Some operating system do not allow child to continue if its parent terminates  
–All children terminated - cascading termination

**Process states**

A process which is Executed by the Process have various States, the State of the Process is also called as the**Status of the process**, The Status includes whether the Process has Executed or Whether the process is Waiting for Some input and output from the user and whether the Process is Waiting for the CPU to Run the Program after the Completion of the Process.

The various States of the Process are as Followings:-

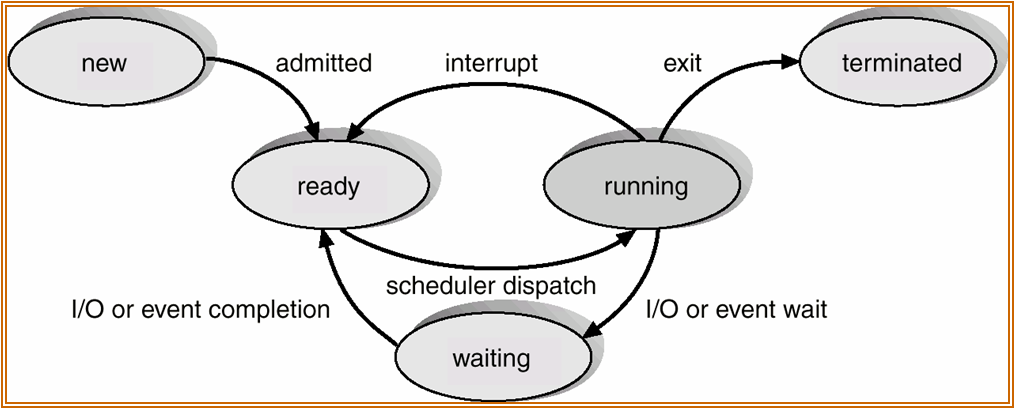
**1) New State:** When a user request for a Service from the System , then the System will first initialize the process or the System will call it an initial Process . So Every new Operation which is Requested to the System is known as the New Born Process.

**2) Running State:** When the Process is Running under the CPU, or When the Program is Executed by the CPU, then this is called as the Running process and when a process is Running then this will also provides us Some Outputs on the Screen.

**3) Waiting:** When a Process is Waiting for Some Input and Output Operations then this is called as the Waiting State. And in this process is not under the Execution instead the Process is Stored out of Memory and when the user will provide the input then this will again be on ready State.

**4) Ready State:** When the Process is Ready to Execute but he is waiting for the CPU to Execute then this is called as the Ready State. After the Completion of the Input and outputs the Process will be on Ready State means the Process will Wait for the [Processor](http://ecomputernotes.com/fundamental/introduction-to-computer/what-is-cpu) to Execute.

**5) Terminated State:** After the Completion of the Process , the Process will be Automatically terminated by the CPU . So this is also called as the Terminated State of the Process. After Executing the Whole Process the Processor will also deallocate the Memory which is allocated to the Process. So this is called as the Terminated Process.



**Interprocess communication (IPC)**

Interprocess communication (IPC) is a set of programming [interface](http://searchcio-midmarket.techtarget.com/definition/interface)s that allows a programmer to coordinate activities among different program [process](http://whatis.techtarget.com/definition/process)es that can run concurrently in an operating system. This allows a program to handle many user requests at the same time. Since even a single user request may result in multiple processes running in the operating system on the user's behalf, the processes need to communicate with each other. The IPC interfaces make this possible. Each IPC method has its own advantages and limitations so it is not unusual for a single program to use all of the IPC methods.

**Basics of Inter process Communication (IPC)**

There are numerous reasons for providing an environment or situation which allows process co-operation:

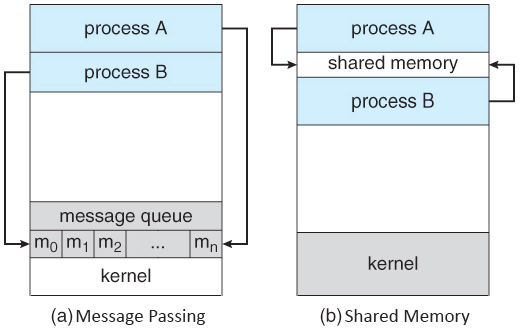
* **Information sharing:** Since a number of users may be interested in the same piece of information (for example, a shared file), you must provide a situation for allowing concurrent access to those information.
* **Computation speedup:** If you want a particular work to run fast, you must break it into sub-tasks where each of them will get execute in parallel with the other tasks. Note that such a speed-up can be attained only when the computer has compound or various processing elements like CPUs or I/O channels.
* **Modularity:** You may want to build the system in a modular way by dividing the system functions into split processes or threads.
* **Convenience:** Even a single user may work on many tasks at a time. For example, a user may be editing, formatting, printing, and compiling in parallel.

Working together multiple processes, require an inter process communication (IPC) method which will allow them to exchange data along with various information. There are two primary models of inter process communication:

* **shared memory and**
* **Message passing.**

In the shared-memory model, a region of memory which is shared by cooperating processes gets established. Processes can then able to exchange information by reading and writing all the data to the shared region. In the message-passing form, communication takes place by way of messages exchanged among the cooperating processes.

The two communications models are contrasted in figure below:



**Shared Memory Systems**

Inter process communication (IPC) usually utilizes shared memory that requires communicating processes for establishing a region of shared memory. Typically, a shared-memory region resides within the address space of any process creating the shared-memory segment. Other processes that wish for communicating using this shared-memory segment must connect it to their address space.

**THREAD**

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

A thread shares with its peer threads 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.

**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.

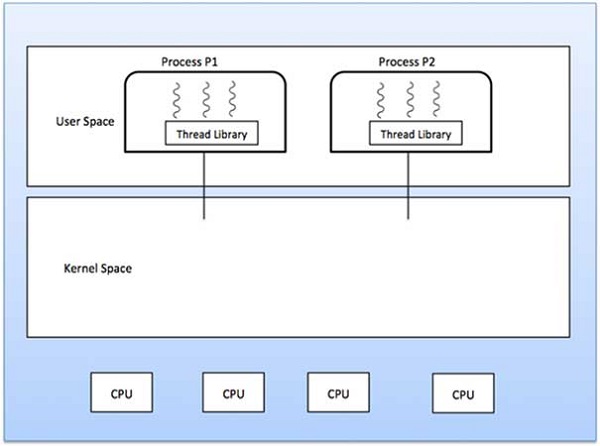
**Types of Thread**

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.

**Multithreading Models**

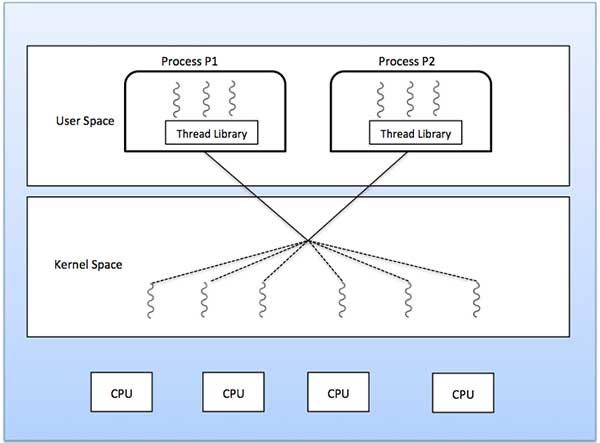
Some operating system provide a combined user level thread and Kernel level thread facility. Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the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

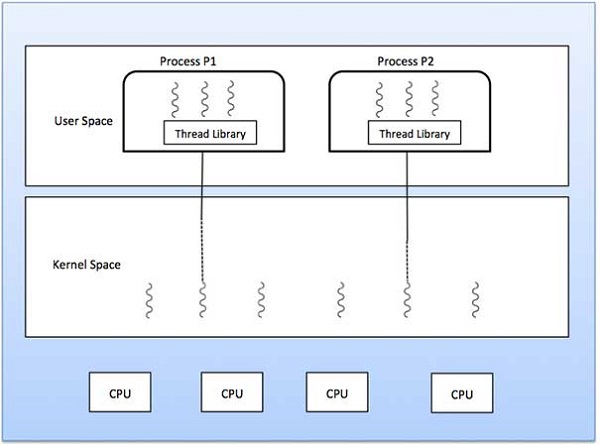
The following diagram shows the many-to-many threading model where 6 user level threads are multiplexing with 6 kernel level threads. In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallel on a multiprocessor machine. This model 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.

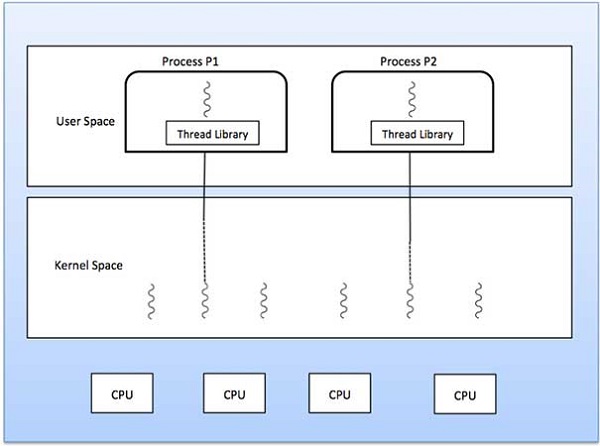
If the user-level thread libraries are implemented in the operating system in such a way that the system does not support them, then the Kernel threads use the many-to-one relationship modes.



**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.



## 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. |

**Critical Section Problem**

In simple terms a critical section is group of instructions/statements or region of code that need to be executed atomically, such as accessing a resource (file, input or output port, global data, etc.).

In concurrent programming, if one thread tries to change the value of shared data at the same time as another thread tries to read the value (i.e. data race across threads), the result is unpredictable.

The access to such shared variable (shared memory, shared files, shared port, etc…) to be synchronized. Few programming languages have built in support for synchronization.

It is critical to understand the importance of race condition while writing kernel mode programming (a device driver, kernel thread, etc.). since the programmer can directly access and modifying kernel data structures.

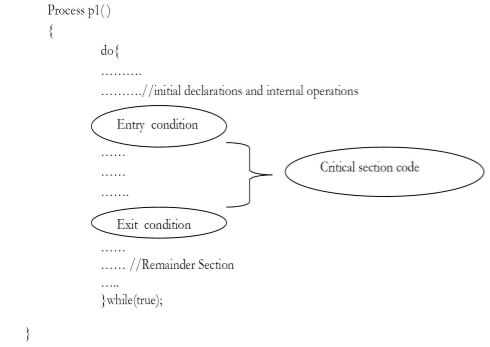
* Consider a system consisting of n processes **P0, P1, ..., Pn.**
* Each process has a segment of code, called a critical section, in which the process may be changing common variables, updating a table, writing a file, and so on. It is section of code that requires access to shared resources.
* An important fact regarding this system is that when a process A is executing in its critical section, no other process B except A is allowed to execute in its critical section.
* No-two processes can execute in its critical section at the same time.
* However many processes may have a critical section code and all of them needs to be executed. So, The Critical Section Problem deals with creating a set of protocols so as to help the process cooperate.

**A solution to a critical section problem must satisfy three conditions;**

a) **Mutual Exclusion:** If a process A is executing in its critical section, then no other processes must execute in its critical section.

b) **Progress:** If no process is currently in its critical section, then only those processes which are currently not in its remainder section can participate in deciding who will enter the Critical section next.

c) **Bounded waiting:** There exists a bound with regards to the number of times other process can enter its critical section between when a process has requested for critical section and when that request is granted



# Solutions to the Critical Section Problem

***Assumption***

* assume that a variable (memory location) can only have one value; never ``between values''
* if processes A and B write a value to the same memory location at the ``same time,'' either the value from A or the value from B will be written rather than some scrambling of bits

***Peterson's Algorithm***

* a simple algorithm that can be run by two processes to ensure mutual exclusion for one resource (say one variable or data structure)
* does not require any special hardware
* it uses busy waiting (a spinlock)

***Peterson's Algorithm***   
Shared variables are created and initialized before either process starts. The shared variables flag[0] and flag[1] are initialized to FALSE because neither process is yet interested in the critical section. The shared variable turn is set to either 0 or 1 randomly (or it can always be set to say 0).   
  
**var** flag: **array** [0..1] **of** boolean;   
turn: 0..1;   
*%flag[k] means that process[k] is interested in the critical section*   
flag[0] := FALSE;   
flag[1] := FALSE;   
turn := random(0..1)   
  
After initialization, each process, which is called process *i* in the code (the other process is process *j*), runs the following code:   
  
**repeat**   
flag[i] := TRUE;   
turn := j;   
**while** (flag[j] **and** turn=j) **do** no-op;   
CRITICAL SECTION   
flag[i] := FALSE;   
REMAINDER SECTION   
**until** FALSE;

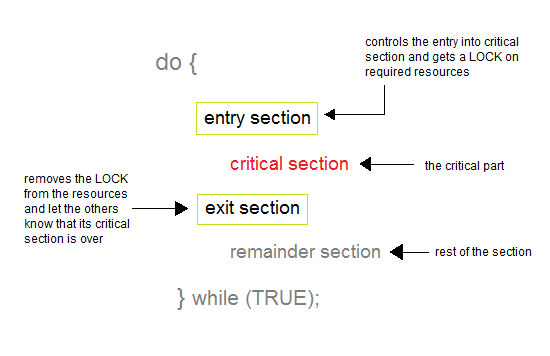
**Process Synchronization**

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

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

**Critical Section Problem**

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



**Solution to Critical Section Problem**

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

**Mutual Exclusion**

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

**Progress**

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

**Bounded Waiting**

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

**Synchronization Hardware**

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

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

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

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

**Mutex Locks**

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

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

**Semaphores**

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

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

**Wait:** decrement the value of its argument S as soon as it would become non-negative.

**Signal:** increment the value of its argument, S as an individual operation.

#### Properties of Semaphores

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

#### Types of Semaphores

Semaphores are mainly of two types:

1. **Binary Semaphore**

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

1. **Counting Semaphores**

These are used to implement bounded concurrency.

Semaphores

A Semaphore is an integer variable, which can be accessed only through two operations *wait()*and *signal()*.

**There are two types of semaphores: Binary Semaphores and Counting Semaphores**

**Binary Semaphores:** They can only be either 0 or 1. They are also known as mutex locks, as the locks can provide mutual exclusion. All the processes can share the same mutex semaphore that is initialized to 1. Then, a process has to wait until the lock becomes 0. Then, the process can make the mutex semaphore 1 and start its critical section. When it completes its critical section, it can reset the value of mutex semaphore to 0 and some other process can enter its critical section.

**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.

**Limitations of Semaphores**

* Priority Inversion is a big limitation of semaphores.
* Their use is not enforced, but is by convention only.
* With improper use, a process may block indefinitely. Such a situation is called Deadlock. We will be studying deadlocks in details in coming lessons.

**Lock Variable**

**Prerequisites –** [Process Synchronization](https://www.geeksforgeeks.org/process-synchronization-set-1/)

A lock variable provides the simplest synchronization mechanism for processes. Some noteworthy points regarding Lock Variables are-

* Its a **software mechanism** implemented in user mode, i.e. no support required from the Operating System.
* Its a busy waiting solution (keeps the CPU busy even when its technically waiting).
* It can be used for more than two processes.

When Lock = 0 implies critical section is vacant (initial value ) and Lock = 1 implies critical section occupied.

The pseudo code looks something like this –

Entry section - while (lock! = 0);

Lock = 1;

//critical section

Exit section - Lock = 0;

Now every Synchronization mechanism is judged on the basis of three primary parameters :

1. Mutual Exclusion.
2. Progress.
3. Bounded Waiting.

Of which mutual exclusion is the most **important** of all parameters. The Lock Variable doesn’t provide mutual exclusion in some cases. This fact can be best verified by writing its pseudo-code in the form of an assembly language code as given below.

1. Load Lock, R0 ; (Store the value of Lock in Register R0.)

2. CMP R0, #0 ; (Compare the value of register R0 with 0.)

3. JNZ Step 1 ; (Jump to step 1 if value of R0 is not 0.)

4. Store #1, Lock ; (Set new value of Lock as 1.)

Enter critical section

5. Store #0, Lock ; (Set the value of lock as 0 again.)

Now let’s suppose that processes P1 and P2 are competing for Critical Section and their sequence of execution be as follows (initial alue of Lock = 0) –

1. P1 executes statement 1 and gets pre-empted.
2. P2 executes statement 1, 2, 3, 4 and enters Critical Section and gets pre-empted.
3. P1 executes statement 2, 3, 4 and also enters Critical Section.

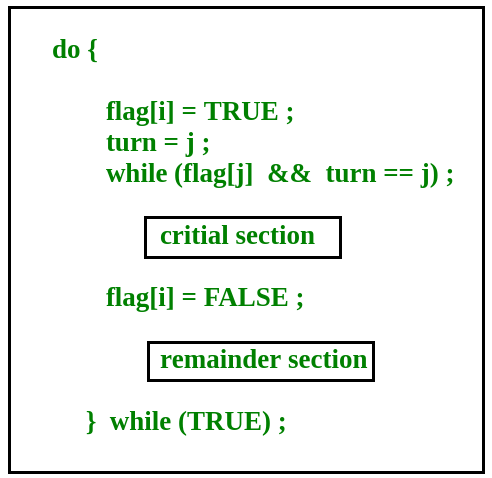
Here initially the R0 of process P1 stores lock value as 0 but fails to update the lock value as 1. So when P2 executes it also finds the LOCK value as 0 and enters Critical Section by setting LOCK value as 1. But the real problem arises when P1 executes again it doesn’t check the updated value of Lock. It only checks the previous value stored in R0 which was 0 and it enters critical section.

**Peterson’s Solution**

Peterson’s Solution is a classical software based solution to the critical section problem.

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

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



**Peterson’s Solution preserves all three conditions:**

Mutual Exclusion is assured as only one process can access the critical section at any time.

Progress is also assured, as a process outside the critical section does not block other processes from entering the critical section.

Bounded Waiting is preserved as every process gets a fair chance.

**Disadvantages of Peterson’s Solution**

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

**Monitors**

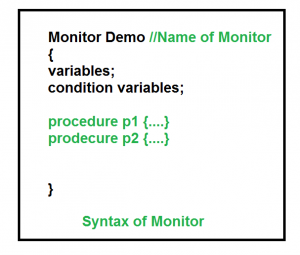
Monitor is one of the ways to achieve Process synchronization. Monitor is supported by programming languages to achieve mutual exclusion between processes. For example Java Synchronized methods. Java provides wait() and notify() constructs.

1. It is the collection of condition variables and procedures combined together in a special kind of module or a package.

2. The processes running outside the monitor can’t access the internal variable of monitor but can call procedures of the monitor.

3. Only one process at a time can execute code inside monitors.

**Syntax of Monitor**

[](https://www.geeksforgeeks.org/wp-content/uploads/gq/2015/06/monitors.png)

**Condition Variables**

Two different operations are performed on the condition variables of the monitor.

Wait.

signal.

let say we have 2 condition variables

**condition x, y; //Declaring variable**

**Wait operation**

x.wait() : Process performing wait operation on any condition variable are suspended. The suspended processes are placed in block queue of that condition variable.

**Note:** Each condition variable has its unique block queue.

**CPU Scheduling**

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

Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the short-term scheduler (or CPU scheduler). The scheduler selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them.

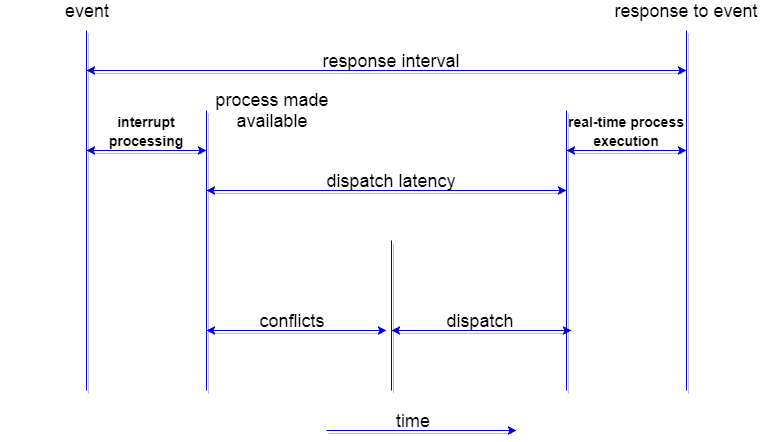
**Dispatcher**

Another component involved in the CPU scheduling function is the **dispatcher**. The dispatcher is the module that gives control of the CPU to the process selected by the **short-term scheduler**.

This function involves:

* Switching context
* Switching to user mode
* Jumping to the proper location in the user program to restart that program

The dispatcher should be as fast as possible, given that it is invoked during every process switch. The time it takes for the dispatcher to stop one process and start another running is known as the **dispatch latency**. Dispatch Latency can be explained using the below figure:



**Types of CPU Scheduling**

CPU scheduling decisions may take place under the following four circumstances:

* When a process switches from the **running** state to the **waiting** state (for I/O request or invocation of wait for the termination of one of the child processes).
* When a process switches from the **running** state to the **ready** state (for example, when an interrupt occurs).
* When a process switches from the **waiting** state to the **ready** state(for example, completion of I/O).
* When a process **terminates**.

In circumstances 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution. There is a choice, however in circumstances 2 and 3.

When Scheduling takes place only under circumstances 1 and 4, we say the scheduling scheme is **non-preemptive**; otherwise the scheduling scheme is **preemptive**.

**Non-Preemptive Scheduling**

Under non-preemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state.

This scheduling method is used by the Microsoft Windows 3.1 and by the Apple Macintosh operating systems.

It is the only method that can be used on certain hardware platforms, because It does not require the special hardware (for example: a timer) needed for preemptive scheduling.

**Preemptive Scheduling**

In this type of Scheduling, the tasks are usually assigned with priorities. At times it is necessary to run a certain task that has a higher priority before another task although it is running. Therefore, the running task is interrupted for some time and resumed later when the priority task has finished its execution.

**Scheduling Criteria**

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

**CPU utilization**

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

**Throughput**

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

**Turnaround time**

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

**Waiting time**

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

**Load average**

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

**Response time**

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

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

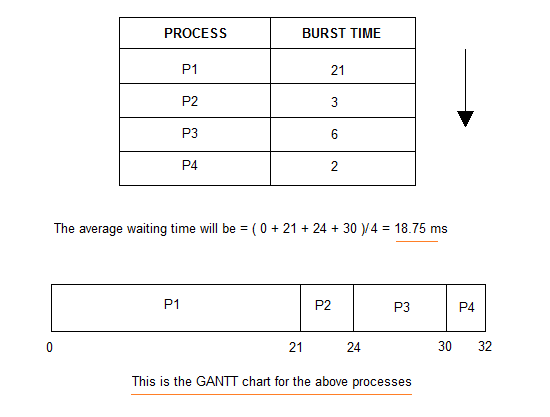
**Scheduling Algorithms**

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

* First Come First Serve(FCFS) Scheduling
* Shortest-Job-First(SJF) Scheduling
* Priority Scheduling
* Round Robin(RR) Scheduling
* Multilevel Queue Scheduling
* Multilevel Feedback Queue Scheduling

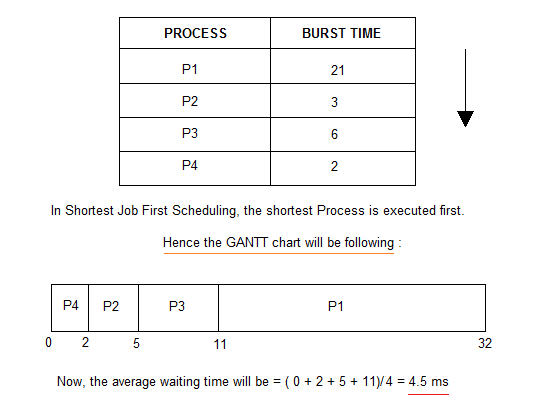
**First Come First Serve (FCFS) Scheduling**

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

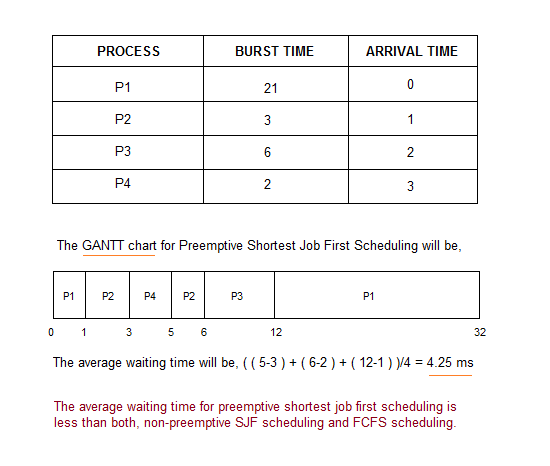


**Shortest-Job-First (SJF) Scheduling**

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



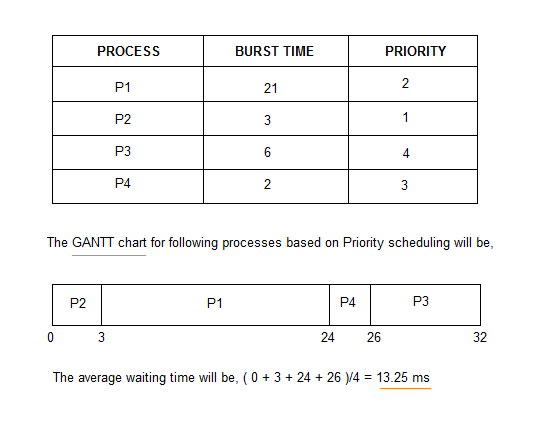
In Preemptive Shortest Job First Scheduling, jobs are put into ready queue as they arrive, but as a process with short burst time arrives, the existing process is preempted.



**Priority Scheduling**

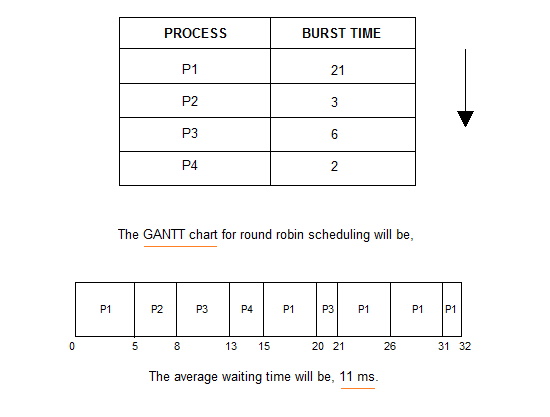
* Priority is assigned for each process.
* Process with highest priority is executed first and so on.
* Processes with same priority are executed in FCFS manner.

Priority can be decided based on memory requirements, time requirements or any other resource requirement.



**Round Robin (RR) Scheduling**

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



***Deadlock***is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.  
Consider an example when two trains are coming toward each other on same track and there is only one track, none of the trains can move once they are in front of each other. Similar situation occurs in operating systems when there are two or more processes hold some resources and wait for resources held by other(s). For example, in the below diagram, Process 1 is holding Resource 1 and waiting for resource 2 which is acquired by process 2, and process 2 is waiting for resource 1.



**Deadlock can arise if following four conditions hold simultaneously (Necessary Conditions)**

* ***Mutual Exclusion:*** One or more than one resource are non-sharable (Only one process can use at a time)
* ***Hold and Wait:*** A process is holding at least one resource and waiting for resources.  
  ***No Preemption:***  A resource cannot be taken from a process unless the process releases the resource.
* ***Circular Wait:*** A set of processes are waiting for each other in circular form.

**Methods for handling deadlock**

There are three ways to handle deadlock

1) Deadlock prevention or avoidance: The idea is to not let the system into deadlock state.

2) Deadlock detection and recovery: Let deadlock occur, then do preemption to handle it once occurred.

3) Ignore the problem all together: If deadlock is very rare, then let it happen and reboot the system. This is the approach that both Windows and UNIX take.

#### How to avoid Deadlocks

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

1. **Mutual Exclusion**

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

1. **Hold and Wait**

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

1. **No Preemption**

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

1. **Circular Wait**

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

#### Handling Deadlock

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

1. **Preemption**

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

1. **Rollback**

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

1. **Kill one or more processes**

This is the simplest way, but it works.