OPERATING SYSTEM NOTES
UNIT-II

Introduction of Process Management

Program vs Process: A process is a program in execution. For example, when we write a program in C or C++ and compile it, the compiler creates binary code. The original code and binary code are both programs. When we actually run the binary code, it becomes a process.

A process is an 'active' entity instead of a program, which is considered 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, multiple instances begin (multiple processes are created).

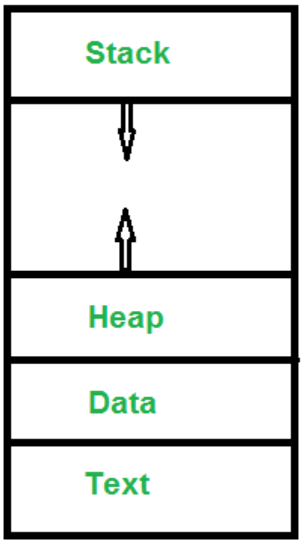
Process Management

If operating system supports multiple users than services under this are very important . In this regard operating systems has to keep track of all the completing processes , Schedule them, dispatch them one after another. But user should feel that he has the full control of the CPU.

some of the systems calls in this category are as follows.

1. create a child process identical to the parent.
2. Terminate a process
3. Wait for a child process to terminate
4. Change the priority of process
5. Block the process
6. Ready the process
7. Dispatch a process
8. Suspend a process
9. Resume a process
10. Delay a process
11. Fork a process

What does a process look like in memory?



Text Section: A Process, sometimes known as the Text Section, also includes the current activity represented by the value of the ***ProgramCounter***.

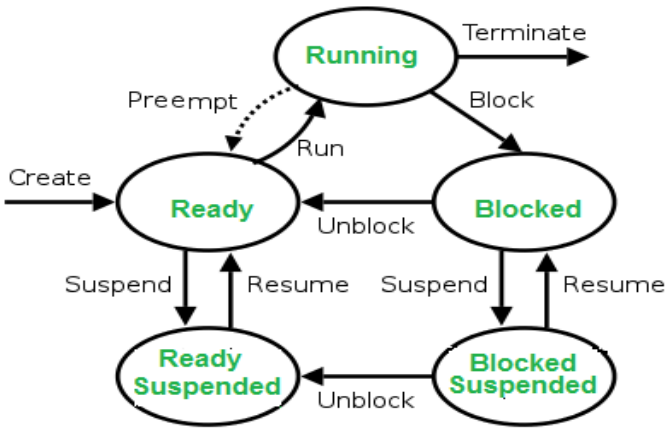
Stack: The stack contains temporary data, such as function parameters, returns addresses, and local variables.

Data Section: Contains the global variable.
Heap Section: Dynamically allocated memory to *process during its run time.*
Attributes or Characteristics of a Process: A process has the following attributes.

1. **Process Id:** A unique identifier assigned by the operating system
2. **Process State:** Can be ready, running, etc.
3. **CPU registers:** Like the Program Counter (CPU registers must be saved and restored when a process is swapped in and out of CPU)
4. **Accounts information:** Amount of CPU used for process execution, time limits, execution ID etc
5. **I/O status information:** For example, devices allocated to the process, open files, etc
6. **CPU scheduling information:** For example, Priority (Different processes may have different priorities, for example a shorter process assigned high priority in the shortest job first scheduling)

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. the 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 a process requests I/O access.
5. **Complete (or Terminated):** The process completed its execution.
6. **Suspended Ready:** When the ready queue becomes full, some processes are moved to suspended ready state
7. **Suspended Block:** When waiting queue becomes full.



Process Schedulers in Operating System

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

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

There are three types of process scheduler.

1. **Long Term or job scheduler :**
   It brings the new process to the 'Ready State'. It controls ***Degree of Multi-programming***, i.e., number of process present in ready state at any point of time. It is important that the long-term scheduler make a careful selection of both I/O and CPU-bound processes. I/O bound tasks are which use much of their time in input and output operations while CPU bound processes are which spend their time on CPU. The job scheduler increases efficiency by maintaining a balance between the two.

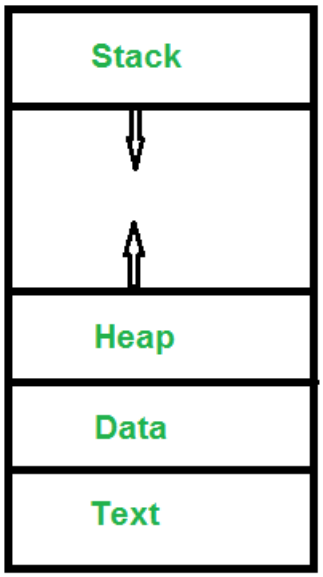
2. **Short term or CPU scheduler :**
   It is responsible for selecting one process from ready state for scheduling it on the running state. Note: Short-term scheduler only selects the process to schedule it doesn't load the process on running. Here is when all the scheduling algorithms are used. The CPU scheduler is responsible for ensuring there is no starvation owing to high burst time processes.
   Dispatcher is responsible for loading the process selected by Short-term scheduler on the CPU (Ready to Running State) Context switching is done by dispatcher only. A dispatcher does the following:
   1. Switching context.
   2. Switching to user mode.
   3. Jumping to the proper location in the newly loaded program.

3. **Medium-term scheduler :**
   It is responsible for suspending and resuming the process. It mainly does swapping (moving processes from main memory to disk and vice versa). Swapping may be necessary to improve the process mix or because a change in memory requirements

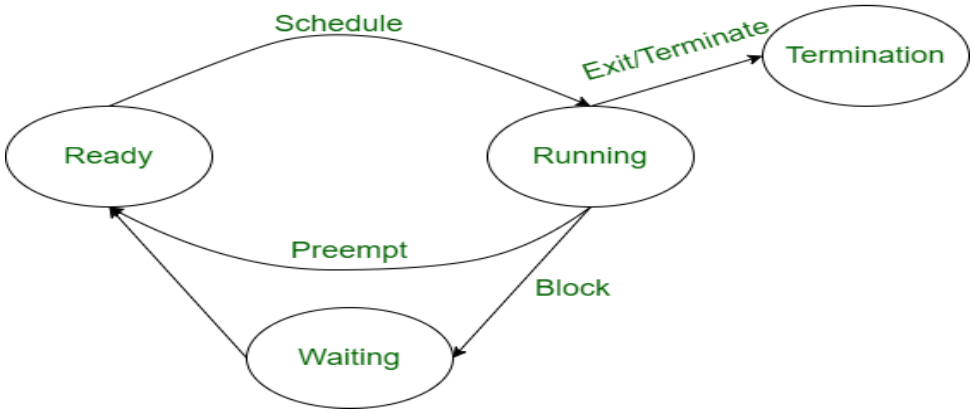
has overcommitted available memory, requiring memory to be freed up. It is helpful in maintaining a perfect balance between the I/O bound and the CPU bound. It reduces the degree of multiprogramming.

Operations on Processes

Process: A process is an activity of executing a program. Basically, it is a program under execution. Every process needs certain resources to complete its task.



Operation on a Process: The execution of a process is a complex activity. It involves various operations.



Following are the operations that are performed while execution of a process:

1. **Creation:** This is the initial step of process execution activity. Process creation means the construction of a new process for the execution. This might be performed by system, user or old process itself. There are several events that leads to the process creation. Some of the such events are following:

- When we start the computer, system creates several background processes.
- A user may request to create a new process.
- A process can create a new process itself while executing.
- Batch system takes initiation of a batch job.

2. **Scheduling/Dispatching:** The event or activity in which the state of the process is changed from ready to running. It means the operating system puts the process from ready state into the running state. Dispatching is done by operating system when the resources are free or the process has higher priority than the ongoing process. There are various other cases in which the process in running state is preempted and process in ready state is dispatched by the operating system.

3. Blocking: When a process invokes an input-output system call that blocks the process and operating system put in block mode. Block mode is basically a mode where process waits for input-output. Hence on the demand of process itself, operating system blocks the process and dispatches another process to the processor. Hence, in process blocking operation, the operating system puts the process in 'waiting' state.

4. Preemption: When a timeout occurs that means the process hadn't been terminated in the allotted time interval and next process is ready to execute, then the operating system preempts the process. This operation is only valid where CPU scheduling supports preemption. Basically this happens in priority scheduling where on the incoming of high priority process the ongoing process is preempted. Hence, in process preemption operation, the operating system puts the process in 'ready' state.

5. Termination: Process termination is the activity of ending the process. In other words, process termination is the relaxation of computer resources taken by the process for the execution. Like creation, in termination also there may be several events that may lead to the process termination. Some of them are:

- Process completes its execution fully and it indicates to the OS that it has finished.
- Operating system itself terminates the process due to service errors.
- There may be problem in hardware that terminates the process.
- One process can be terminated by another process.

Cooperating Process in Operating System

In this article, you will learn about the cooperating process in the operating system and its various methods.

What is Cooperating Process?

There are various processes in a computer system, which can be either independent or cooperating processes that operate in the operating system. It is considered independent when any other processes operating on the system may not impact a process. Process-independent processes don't share any data with other processes. On the other way, a collaborating process may be affected by any other process executing on the system. A cooperating process shares data with another.

Advantages of Cooperating Process in Operating System

There are various advantages of cooperating process in the operating system. Some advantages of the cooperating system are as follows:

1. Information Sharing
Cooperating processes can be used to share information between various processes. It could involve having access to the same files. A technique is necessary so that the processes may access the files concurrently.

2. Modularity
Modularity refers to the division of complex tasks into smaller subtasks. Different cooperating processes can complete these smaller subtasks. As a result, the required tasks are completed more quickly and efficiently.

3. Computation Speedup
Cooperating processes can be used to accomplish subtasks of a single task simultaneously. It improves computation speed by allowing the task to be accomplished faster. Although, it is only possible if the system contains several processing elements.

4. Convenience
There are multiple tasks that a user requires to perform, such as printing, compiling, editing, etc. It is more convenient if these activities may be managed through cooperating processes.

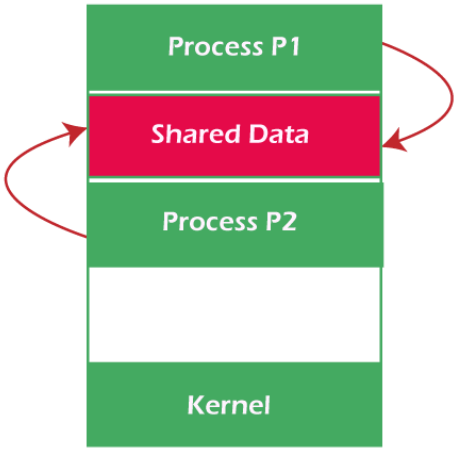
Concurrent execution of cooperating processes needs systems that enable processes to communicate and synchronize their actions.

Methods of Cooperating Process

Cooperating processes may coordinate with each other by sharing data or messages. The methods are given below:

1. Cooperation by sharing

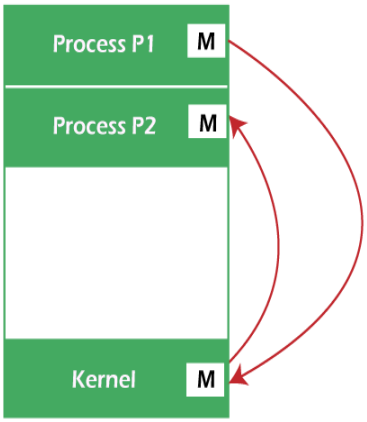
The processes may cooperate by sharing data, including variables, memory, databases, etc. The critical section provides data integrity, and writing is mutually exclusive to avoid inconsistent data.



Here, you see a diagram that shows cooperation by sharing. In this diagram, Process P1 and P2 may cooperate by using shared data like files, databases, variables, memory, etc.

2. Cooperation by Communication

The cooperating processes may cooperate by using messages. If every process waits for a message from another process to execute a task, it may cause a deadlock. If a process does not receive any messages, it may cause starvation.



Here, you have seen a diagram that shows cooperation by communication. In this diagram, Process P1 and P2 may cooperate by using messages to communicate.

Inter Process Communication (IPC)

What is Inter Process Communication?

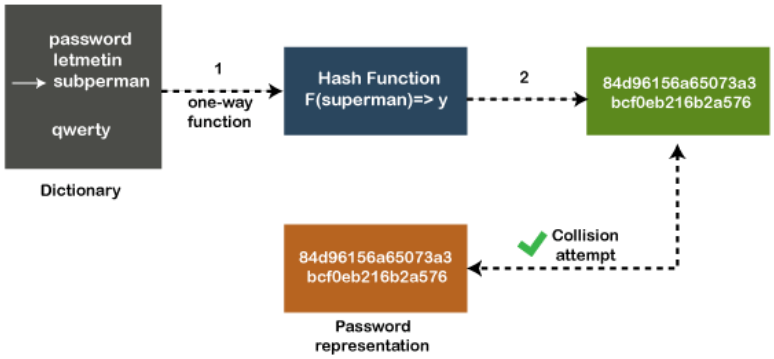
In general, Inter Process Communication is a type of mechanism usually provided by the operating system (or OS). The main aim or goal of this mechanism is to provide communications in between several processes. In short, the intercommunication allows a process letting another process know that some event has occurred.

Let us now look at the general definition of inter-process communication, which will explain the same thing that we have discussed above.

Definition

"Inter-process communication is used for exchanging useful information between numerous threads in one or more processes (or programs)."

To understand inter process communication, you can consider the following given diagram that illustrates the importance of inter-process communication:



Role of Synchronization in Inter Process Communication

It is one of the essential parts of inter process communication. Typically, this is provided by interprocess communication control mechanisms, but sometimes it can also be controlled by communication processes.

These are the following methods that used to provide the synchronization:

1. **Mutual Exclusion**
2. **Semaphore**
3. **Barrier**
4. **Spinlock**

Mutual Exclusion:-

It is generally required that only one process thread can enter the critical section at a time. This also helps in synchronization and creates a stable state to avoid the race condition.

Semaphore:-

Semaphore is a type of variable that usually controls the access to the shared resources by several processes. Semaphore is further divided into two types which are as follows:

1. Binary Semaphore
2. Counting Semaphore

Barrier:-

A barrier typically not allows an individual process to proceed unless all the processes does not reach it. It is used by many parallel languages, and collective routines impose barriers.

Spinlock:-

Spinlock is a type of lock as its name implies. The processes are trying to acquire the spinlock waits or stays in a loop while checking that the lock is available or not. It is known as busy waiting because even though the process active, the process does not perform any functional operation (or task).

Approaches to Interprocess Communication

We will now discuss some different approaches to inter-process communication which are as follows:

These are a few different approaches for Inter- Process Communication:

1. **Pipes**
2. **Shared Memory**
3. **Message Queue**

4. **Direct Communication**
5. **Indirect communication**
6. **Message Passing**
7. **FIFO**

To understand them in more detail, we will discuss each of them individually.

Pipe:-

The pipe is a type of data channel that is unidirectional in nature. It means that the data in this type of data channel can be moved in only a single direction at a time. Still, one can use two-channel of this type, so that he can able to send and receive data in two processes. Typically, it uses the standard methods for input and output. These pipes are used in all types of POSIX systems and in different versions of window operating systems as well.

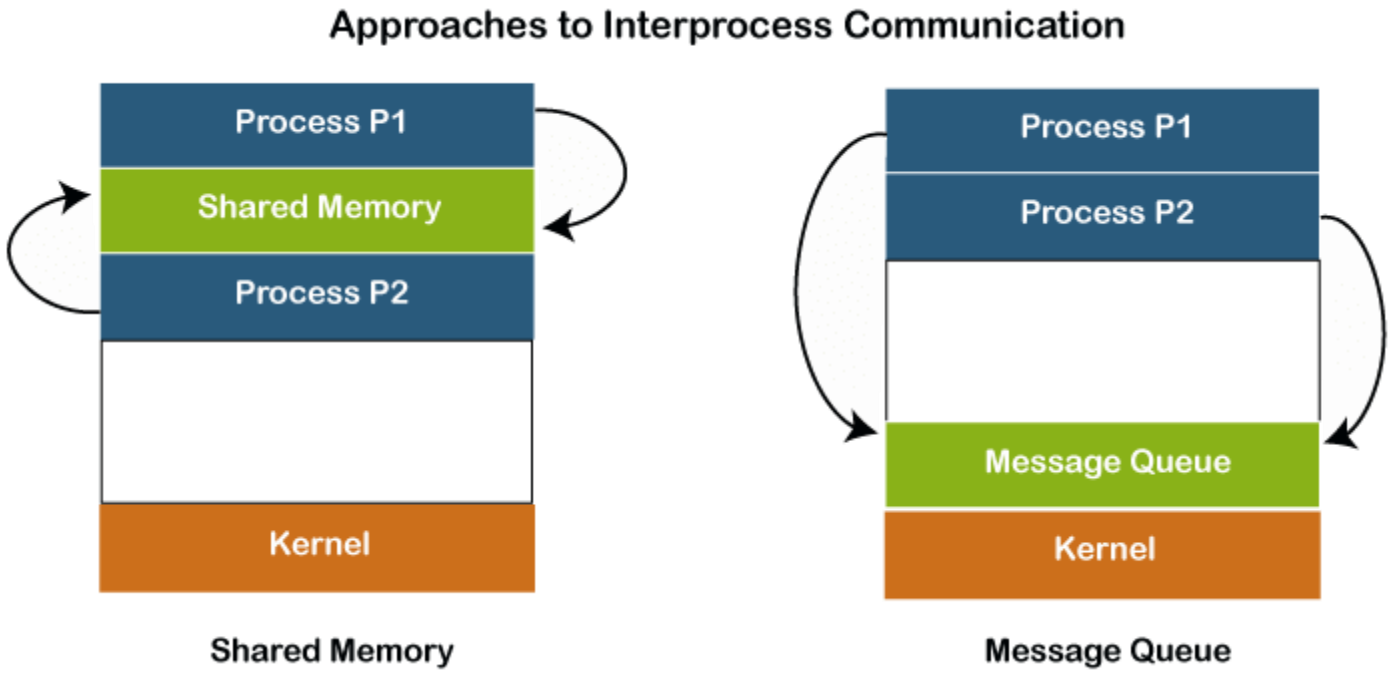
Shared Memory:-

It can be referred to as a type of memory that can be used or accessed by multiple processes simultaneously. It is primarily used so that the processes can communicate with each other. Therefore the shared memory is used by almost all POSIX and Windows operating systems as well.

Message Queue:-

In general, several different messages are allowed to read and write the data to the message queue. In the message queue, the messages are stored or stay in the queue unless their recipients retrieve them. In short, we can also say that the message queue is very helpful in inter-process communication and used by all operating systems.

To understand the concept of Message queue and Shared memory in more detail, let's take a look at its diagram given below:



Message Passing:-

It is a type of mechanism that allows processes to synchronize and communicate with each other. However, by using the message passing, the processes can communicate with each other without restoring the hared variables.

Usually, the inter-process communication mechanism provides two operations that are as follows:

- send (message)
- received (message)

Direct Communication:-

In this type of communication process, usually, a link is created or established between two communicating processes. However, in every pair of communicating processes, only one link can exist.

Indirect Communication

Indirect communication can only exist or be established when processes share a common mailbox, and each pair of these processes shares multiple communication links. These shared links can be unidirectional or bi-directional.

FIFO:-

It is a type of general communication between two unrelated processes. It can also be considered as full-duplex, which means that one process can communicate with another process and vice versa.

Some other different approaches

- **Socket:-**

  It acts as a type of endpoint for receiving or sending the data in a network. It is correct for data sent between processes on the same computer or data sent between different computers on the same network. Hence, it used by several types of operating systems.

- **File:-**

  A file is a type of data record or a document stored on the disk and can be acquired on demand by the file server. Another most important thing is that several processes can access that file as required or needed.

- **Signal:-**

  As its name implies, they are a type of signal used in inter process communication in a minimal way. Typically, they are the massages of systems that are sent by one process to another. Therefore, they are not used for sending data but for remote commands between multiple processes.

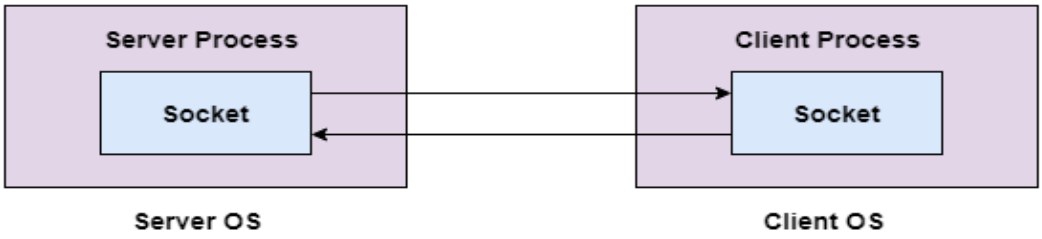
Operating Systems Client/Server Communication

Client/Server communication involves two components, namely a client and a server. They are usually multiple clients in communication with a single server. The clients send requests to the server and the server responds to the client requests.

There are three main methods to client/server communication. These are given as follows –

Sockets

A diagram that illustrates sockets is as follows –



Sockets facilitate communication between two processes on the same machine or different machines. They are used in a client/server framework and consist of the IP address and port number. Many application protocols use sockets for data connection and data transfer between a client and a server.

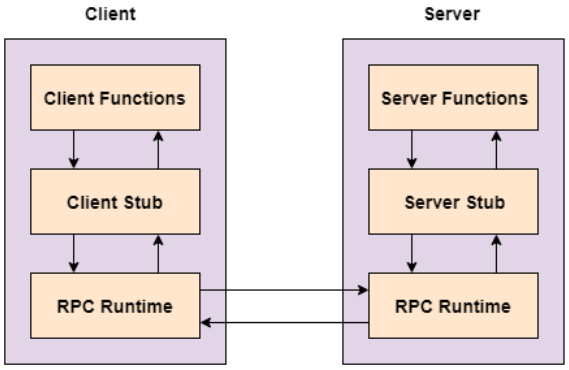
Socket communication is quite low-level as sockets only transfer an unstructured byte stream across processes. The structure on the byte stream is imposed by the client and server applications.

Remote Procedure Calls

These are interprocess communication techniques that are used for client-server based applications. A remote procedure call is also known as a subroutine call or a function call.

A client has a request that the RPC translates and sends to the server. This request may be a procedure or a function call to a remote server. When the server receives the request, it sends the required response back to the client.

A diagram that illustrates remote procedure calls is given as follows –



Pipes

These are interprocess communication methods that contain two end points. Data is entered from one end of the pipe by a process and consumed from the other end by the other process.

The two different types of pipes are ordinary pipes and named pipes. Ordinary pipes only allow one way communication. For two way communication, two pipes are required. Ordinary pipes have a parent child relationship between the processes as the pipes can only be accessed by processes that created or inherited them.

Named pipes are more powerful than ordinary pipes and allow two way communication. These pipes exist even after the processes using them have terminated. They need to be explicitly deleted when not required anymore.

A diagram that demonstrates pipes are given as follows –

Thread in Operating System

What is a Thread?

A thread is a path of execution within a process. A process can contain multiple threads.

Why Multithreading?

A thread is also known as lightweight process. The idea is to 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, another thread to process inputs, etc. More advantages of multithreading are discussed below

Process vs Thread?

The primary difference is that threads within the same process run in a shared memory space, while processes run in separate memory spaces.
Threads are not independent of one another like processes are, and as a result threads share with other threads their code section, data section, and OS resources (like open files and signals). But, like process, a thread has its own program counter (PC), register set, and stack space.

Advantages of Thread over Process

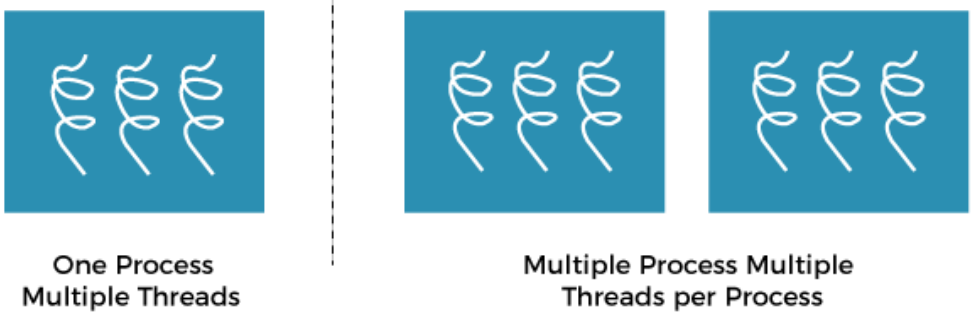
1. Responsiveness: If the process is divided into multiple threads, if one thread completes its execution, then its output can be immediately returned.
2. Faster context switch: Context switch time between threads is lower compared to process context switch. Process context switching requires more overhead from the CPU.
3. Effective utilization of multiprocessor system: If we have multiple threads in a single process, then we can schedule multiple threads on multiple processor. This will make process execution faster.
4. Resource sharing: Resources like code, data, and files can be shared among all threads within a process.
Note: stack and registers can't be shared among the threads. Each thread has its own stack and registers.

Multithreading Models in Operating system

In this article, we will understand the multithreading model in the Operating system.

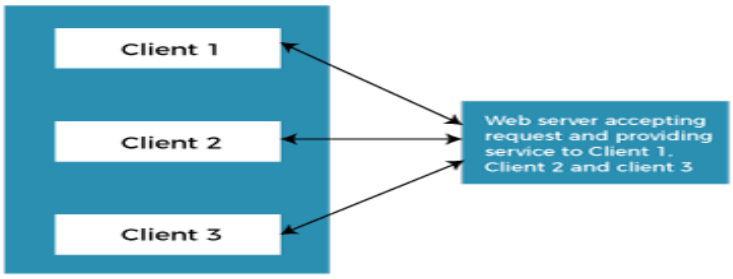
Multithreading Model:

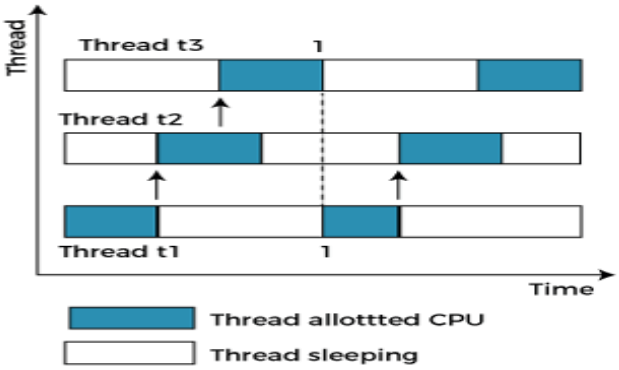
Multithreading allows the application to divide its task into individual threads. In multi-threads, the same process or task can be done by the number of threads, or we can say that there is more than one thread to perform the task in multithreading. With the use of multithreading, multitasking can be achieved.



The main drawback of single threading systems is that only one task can be performed at a time, so to overcome the drawback of this single threading, there is multithreading that allows multiple tasks to be performed.

For example:





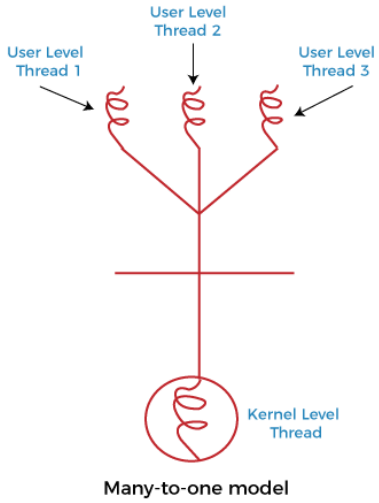
In the above example, client1, client2, and client3 are accessing the web server without any waiting. In multithreading, several tasks can run at the same time.

In an operating system , threads are divided into the user-level thread and the Kernel-level thread. User-level threads handled independent form above the kernel and thereby managed without any kernel support. On the opposite hand, the operating system directly manages the kernel-level threads. Nevertheless, there must be a form of relationship between user-level and kernel-level threads.

There exists three established multithreading models classifying these relationships are:

Many to one multithreading model
One to one multithreading model
Many to Many multithreading models

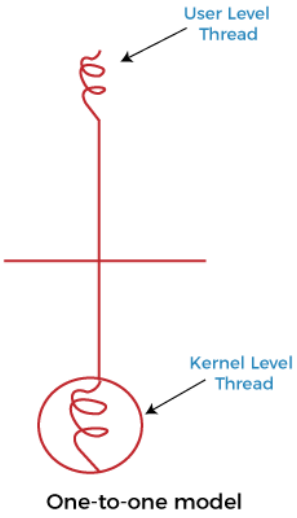
Many to one multithreading model:



Many-to-one model

The many to one model maps many user levels threads to one kernel thread. This type of relationship facilitates an effective context-switching environment, easily implemented even on the simple kernel with no thread support.

The disadvantage of this model is that since there is only one kernel-level thread schedule at any given time, this model cannot take advantage of the hardware acceleration offered by multithreaded processes or multi-processor systems. In this, all the thread management is done in the userspace. If blocking comes, this model blocks the whole system.



One-to-one model

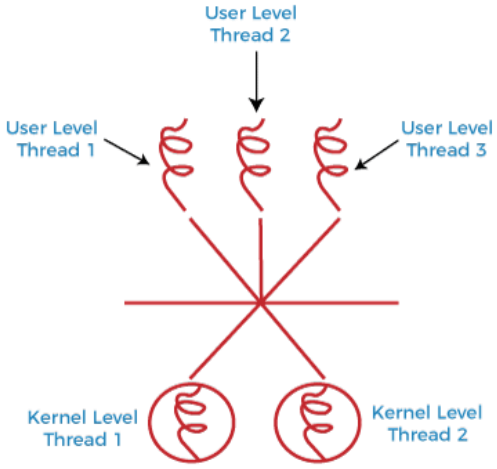
In the above figure, the many to one model associates all user-level threads to single kernel-level threads.

One to one multithreading model

The one-to-one model maps a single user-level thread to a single kernel-level thread. This type of relationship facilitates the running of multiple threads in parallel. However, this benefit comes with its drawback. The generation of every new user thread must include creating a corresponding kernel thread causing an overhead, which can hinder the performance of the parent process. Windows series and Linux operating systems try to tackle this problem by limiting the growth of the thread count.

In the above figure, one model associates that one user-level thread to a single kernel-level thread.

Many to Many Model multithreading model



Many-to-Many model

In this type of model, there are several user-level threads and several kernel-level threads. The number of kernel threads created depends upon a particular application. The developer can create as many threads at both levels but may not be the same. The many to many model is a compromise between the other two models. In this model, if any thread makes a blocking system call, the kernel can schedule another thread for execution. Also, with the introduction of multiple threads, complexity is not present as in the previous models. Though this model allows the creation of multiple kernel threads, true concurrency cannot be achieved by this model. This is because the kernel can schedule only one process at a time.

Many to many versions of the multithreading model associate several user-level threads to the same or much less variety of kernel-level threads in the above figure.

Threading Issues in OS

1. System Calls
2. Thread Cancellation
3. Signal Handling
4. Thread Pool

5. Thread Specific Data

1. fork() and exec() System Calls

The fork() and exec() are the system calls. The fork() call creates a duplicate process of the process that invokes fork(). The new duplicate process is called child process and process invoking the fork() is called the parent process. Both the parent process and the child process continue their execution from the instruction that is just after the fork().

Let us now discuss the issue with the fork() system call. Consider that a thread of the multithreaded program has invoked the fork(). So, the fork() would create a new duplicate process. Here the issue is whether the new duplicate process created by fork() will duplicate all the threads of the parent process or the duplicate process would be single-threaded.

Well, there are two versions of fork() in some of the UNIX systems. Either the fork() can duplicate all the threads of the parent process in the child process or the fork() would only duplicate that thread from parent process that has invoked it. Which version of fork() must be used totally depends upon the application.

Next system call i.e. exec() system call when invoked replaces the program along with all its threads with the program that is specified in the parameter to exec(). Typically the exec() system call is lined up after the fork() system call.

Here the issue is if the exec() system call is lined up just after the fork() system call then duplicating all the threads of parent process in the child process by fork() is useless. As the exec() system call will replace the entire process with the process provided to exec() in the parameter.

In such case, the version of fork() that duplicates only the thread that invoked the fork() would be appropriate.

2. Thread cancellation

Termination of the thread in the middle of its execution it is termed as 'thread cancellation'. Let us understand this with the help of an example. Consider that there is a multithreaded program which has let its multiple threads to search through a database for some information. However, if one of the thread returns with the desired result the remaining threads will be cancelled.

Now a thread which we want to cancel is termed as target thread. Thread cancellation can be performed in two ways:

Asynchronous Cancellation: In asynchronous cancellation, a thread is employed to terminate the target thread instantly.

Deferred Cancellation: In deferred cancellation, the target thread is scheduled to check itself at regular interval whether it can terminate itself or not.
The issue related to the target threads are listed below:

- What if the resources had been allotted to the cancel target thread?
- What if the target thread is terminated when it was updating the data, it was sharing with some other thread.

Here the asynchronous cancellation of the thread where a thread immediately cancels the target thread without checking whether it is holding any resources or not creates troublesome.
However, in deferred cancellation, the thread that indicates the target thread about the cancellation, the target thread crosschecks its flag in order to confirm that it should it be cancelled immediately or not. The thread cancellation takes place where they can be cancelled safely such points are termed as **cancellation points** by Pthreads.

3. Signal Handling

Signal handling is more convenient in the single-threaded program as the signal would be directly forwarded to the process. But when it comes to multithreaded program, the issue arrives to which thread of the program the signal should be delivered.
Let's say the signal would be delivered to:

- All the threads of the process.
- To some specific threads in a process.
- To the thread to which it applies
- Or you can assign a thread to receive all the signals.

Well, how the signal would be delivered to the thread would be decided, depending upon the type of generated signal. The generated signal can be classified into two type's synchronous signal and asynchronous signal.
Synchronous signals are forwarded to the same process that leads to the generation of the signal. Asynchronous signals are generated by the event external to the running process thus the running process receives the signals asynchronously.
So if the signal is synchronous it would be delivered to the specific thread causing the generation of the signal. If the signal is asynchronous it cannot be specified to which thread of the multithreaded program it would be delivered. If the asynchronous signal is notifying to terminate the process the signal would be delivered to all the thread of the process.
The issue of an asynchronous signal is resolved up to some extent in most of the multithreaded UNIX system. Here the thread is allowed to specify which signal it can

accept and which it cannot. However, the Window operating system does not support the concept of the signal instead it uses asynchronous procedure call (ACP) which is similar to the asynchronous signal of the UNIX system.
UNIX allow the thread to specify which signal it can accept and which it will not whereas the ACP is forwarded to the specific thread.

4. Thread Pool

When a user requests for a webpage to the server, the server creates a separate thread to service the request. Although the server also has some potential issues. Consider if we do not have a bound on the number of actives thread in a system and would create a new thread for every new request then it would finally result in exhaustion of system resources.
We are also concerned about the time it will take to create a new thread. It must not be that case that the time require to create a new thread is more than the time required by the thread to service the request and then getting discarded as it would result in wastage of CPU time.
The solution to this issue is the **thread pool**. The idea is to create a finite amount of threads when the process starts. This collection of threads is referred to as the thread pool. The threads stay in the thread pool and wait till they are assigned any request to be serviced.
Whenever the request arrives at the server, it invokes a thread from the pool and assigns it the request to be serviced. The thread completes its service and return back to the pool and wait for the next request.
If the server receives a request and it does not find any thread in the thread pool it waits for some or the other thread to become free and return to the pool. This much better than creating a new thread each time a request arrives and convenient for the system that cannot handle a large number of concurrent threads.

5. Thread Specific data

We all are aware of the fact that the threads belonging to the same process share the data of that process. Here the issue is what if each particular thread of the process needs its own copy of data. So the specific data associated with the specific thread is referred to as **thread-specific data.**
Consider a transaction processing system, here we can process each transaction in a different thread. To determine each transaction uniquely we will associate a unique identifier with it. Which will help the system to identify each transaction uniquely.

As we are servicing each transaction in a separate thread. So we can use thread-specific data to associate each thread to a specific transaction and its unique id. Thread libraries such as Win32, Pthreads and Java support to thread-specific data. So these are threading issues that occur in the multithreaded programming environment. We have also seen how these issues can be resolved.