Thread in Operating System

A thread is a path of execution within a process also known as a lightweight process. A process can contain multiple threads.
The main difference between threads and processes is that threads within the same process run in a shared memory space, while processes run in separate memory spaces. But stack and registers can't be shared among the threads. Each thread has its own stack and registers.

Advantages of Thread over Process

1. Responsiveness
2. Faster context switch
3. Effective utilization of multiprocessor system
4. Resource sharing
5. Communication

Each thread has

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

Types of Threads:

	Definition	**Advantages**	**Disadvantages**
User Level thread (ULT)	It is implemented in the user level library. Kernel doesn't know about the user level thread and manages them as if they were	Implementable on an OS that does't support multithreading. Simple representation.	No coordination among threads and kernel. Entire process gets blocked if one thread causes page fault.

	single-threaded processes.	Thread switching is fast.	
Kernel Level Thread (KLT)	Kernel itself has a thread table which keeps track of all threads. OS kernel provides system call to create and manage threads.	As the kernel knows about all threads, schedulers may decide to give more time to processes having large numbers of threads. Good for applications that frequently block.	Inefficient and slow. Requires thread control block so it is an overhead.

Multitasking programming is of two types –

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

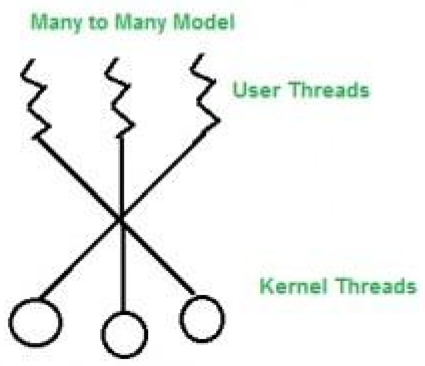
Process-based Multitasking	Thread-based Multitasking
In this, 2 or more processes or programs can be run concurrently.	In this, 2 or more threads can be run concurrently.
It requires more overhead.	It requires less overhead.
Process requires its own address space.	Threads share the same address space.
Process to process communication is expensive.	Thread to thread communication is not expensive.

It is heavy weight	It is lightweight.
Ex: Music player and browser at same time.	Ex: Typing text in one thread and spell checker in another thread in MS Word.

Many operating systems support kernel thread and user thread in a combined way. Multi threading model are of three types:

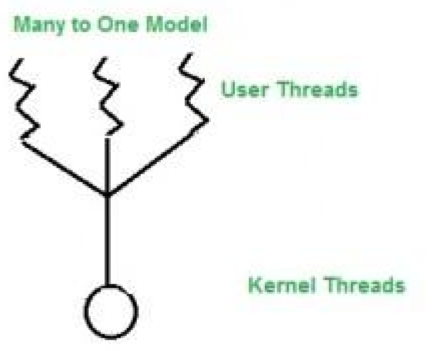
1. **Many to Many Model**

In this model, we have multiple user threads multiplexing to the same or lesser number of kernel level threads. If a user thread is blocked we can schedule others user thread to other kernel thread. Thus, System doesn’t block if a particular thread is blocked.



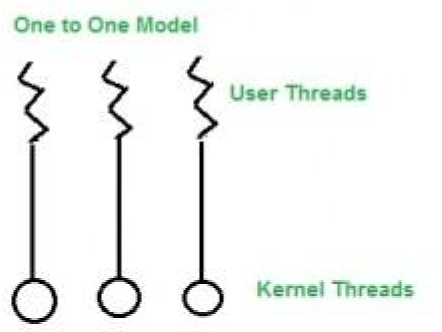
2. **Many to One Model**

In this model, we have multiple user threads mapped to one kernel thread. When a user thread makes a blocking system call entire process blocks. As we have only one kernel thread and only one user thread can access the kernel at a time, so multiple threads are not able access multiprocessor at the same time.



3. One to One Model

In this model, one to one relationship between kernel and user thread. Multiple threads can run on multiple processors. It requires the corresponding kernel thread for creating a user thread.



fork() in C

Fork system call is used for creating a new process, which is called a child ***process***, which runs concurrently with the process that makes the fork() call (parent process) after which both processes will execute the next instruction following the fork() system call. A child process uses the same pc(program counter), same CPU registers, same open files which are used in the parent process.
It takes no parameters and returns an integer value.

Negative Value: creation of a child process was unsuccessful.

Zero: Returned to the newly created child process.

Positive value: Returned to parent or caller. The value contains the process ID of the newly created child process.

Parent process and child process are running the same program, but it does not mean they are identical. OS allocate different data and states for these two processes, and the control flow of these processes can be different.

Zombie state : It is a state of child when there is an entry of child in the process table even after its termination.

When a process is created in UNIX using fork() system call, the address space of the Parent process is replicated. If the parent process calls wait() system call, then the execution of the parent is suspended until the child is terminated after which a 'SIGCHLD' signal is generated which is delivered to the parent by the kernel. When the parent collects the status, this entry is deleted. But if the parent decides not to wait for the child's termination then at the termination of the child, the exit status is not read. Hence, there remains an entry in the process table even after the termination of the child. This state of the child process is known as the Zombie state.

Now, we need to prevent this because the size of the process table is finite. If too many zombie processes are generated, then the process table will be full and we won't be able to generate new processes. So the different ways are:-

1. By ignoring the SIGCHLD signal: If we call the 'signal(SIGCHLD,SIG_IGN)', then the SIGCHLD signal is ignored by the system, and the child process entry is deleted from the process table. But the parent can't know about the exit status of the child.

2. By using a signal handler: When the child is terminated, the SIGCHLD is delivered to the parent and hence, the corresponding handler is activated, which in turn calls the wait() system call. Hence, the parent collects the exit status almost immediately and the child entry in the process table is cleared. Thus no zombie is created.

Remote Procedure Call (RPC) in Operating System

It is a powerful technique for constructing distributed, client-server based applications. It is based on extending the conventional local procedure calling so that the called procedure need not exist in the same address space as the calling procedure. The two processes may be on the same system, or they may be on different systems with a network connecting them.

The following steps take place during working of a RPC:

1. A client invokes a client stub procedure, passing parameters in the usual way. The client stub resides within the client's own address space.

2. The client stub marshalls(pack) the parameters into a message and passes it to the transport layer which sends it to the remote server machine.

3. On the server, the transport layer passes the message to a server stub, which demarshalls(unpack) the parameters and calls the desired server routine using the regular procedure call mechanism.

4. When the server procedure completes, it returns to the server stub (e.g., via a normal procedure call return), which marshalls the return values into a message. The server stub then hands the message to the transport layer.

5. The transport layer sends the result message back to the client transport layer, which hands the message back to the client stub.

6. The client stub demarshalls the return parameters and execution returns to the caller.

<u>Terms:</u>

1. RPC Runtime: This system is a library of routines and a set of services that handle the network communications that underlie the RPC mechanism.

2. Stub: A **stub** in distributed computing is a piece of code that converts parameters passed between client and server during a remote procedure call (RPC).

3. Binding: It is used to acknowledge the client to whom it will call and where the service resides.

Binding consists of two parts:

- <u>Naming:</u>

  Remote procedures are named through interfaces. An interface uniquely identifies a particular service, describing the types and numbers of its arguments.
- <u>Locating:</u>

  Finding the transport address at which the server actually resides. Once we have the transport address of the service, we can send messages directly to the server.

ADVANTAGES

1. RPC provides ABSTRACTION i.e message-passing nature of network communication is hidden from the user.

2. RPC often omits many of the protocol layers to improve performance. Even a small performance improvement is important because a program may invoke RPCs often.

3. RPC enables the usage of the applications in the distributed environment, not only in the local environment.

4. With RPC code re-writing / re-developing effort is minimized.

5. Process-oriented and thread oriented models supported by RPC.