Threads

1. **Introduction**
2. **Thread model**
3. **Thread usage**
4. **Advantages of threads**
5. **User space and kernel space threads**
6. **Multithreading model**
7. **Differences between threads and processes**

1. Introduction:

A thread is a separate part of a process i.e. a thread is the smallest unit of processing that can be performed in an operating system. A process can consist of several threads, each of which execute separately. It is a flow of control within a process. It is also known as ***light weight process***. For example, one thread could handle screen refresh and drawing, another thread printing, another thread the mouse and keyboard. This gives good response times for complex programs. Windows NT is an example of an operating system which supports multi-threading.

A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack. It shares with other threads belonging to same process its code section, data section, and other operating resources, such as open files and signals. A traditional (or heavy weight) process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time.

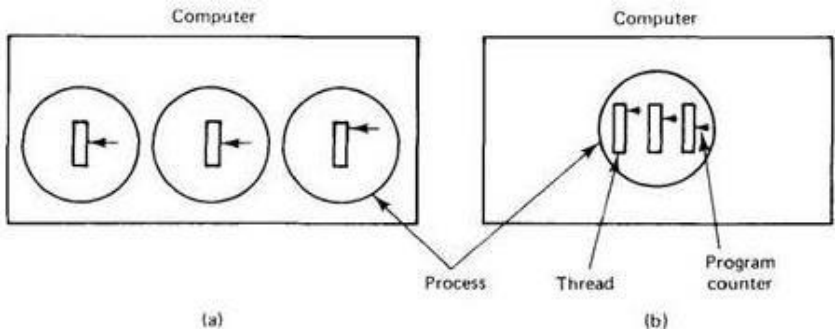

Fig: (a) Three processes with one thread each (b) A process with 3 threads

2. Multithreading:

Multithreading is the ability of a program or an operating system process to manage its use by more than one user at a time and to even manage multiple requests by the same user without having to have multiple copies of the programming running in the computer. A multithread process contains several different flow of control within the same address space. When multiple threads are running concurrently then this is known as ***multithreading***, which is similar to multitasking. Multithreading allows sub-processes to run concurrently. One example of multithreading is when we are downloading a video while playing it at the same time. Multithreading is also used extensively in computer-generated animations.

3. Thread Usage

Threads were invented to allow parallelism to be combined with sequential execution and blocking system calls. Some examples of situations where we might use threads:

- Doing lengthy processing: When a windows application is calculating it cannot process any more messages. As a result, the display cannot be updated.
- Doing background processing: Some tasks may not be time critical, but need to execute continuously.
- Concurrent execution on multiprocessors
- Manage I/O more efficiently: some threads wait for I/O while others compute
- It is mostly used in large server applications

4. Advantages of Threads

Advantage of multithread includes:

- **Resource sharing**
  Process can only share resources using shared memory or message passing method. Such techniques are explicitly arranged by the programmer. However, threads share the memory and resources of the process to which they are belong by default. It (Sharing of code and data) allows an application to have several different threads of activity within the same address space.
- **Responsiveness**
  It may allow a program to continue running even if part of it is blocked or is performing lengthy operations, which increases the responsiveness to the user.
- **Economy**
  Allocation of memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads. It is much more time consuming to create and manage processes than threads.
- **Scalability**
  In multiprocessor architecture threads may run in parallel on different processors. A single threaded process can only run on one processor. Multithreading on a multi-CPU machine increases parallelism.

5. User Space and Kernel Space Threads

User level threads are threads that are visible to the programmer and unknown to the kernel. Such threads are supported above the kernel and managed without kernel support.

The operating system directly supports and manages kernel level threads. In general, user level threads are faster to create and manage than that of kernel threads, because no intervention from the kernel is required. Windows XP, Linux, Mac OS X supports kernel threads.

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 application cannot take advantage of multiprocessing.	Kernel routines themselves can be multithreaded.

6. Multithreading Model

A relationship must exist between user threads and kernel threads. There are three models related to user and kernel threads and they are:

- **Many-to-one model:** It maps many user threads to single kernel threads.
- **One–to-one model:** It maps each user thread to a corresponding thread
- **Many-to-many model:** It maps many user threads to a smaller or equal number of kernel threads.

A. Many to one model:

It maps many user threads to single kernel threads. Thread management is done by the thread library in user space, so it if efficient but the entire process will block if a thread makes a blocking system call. Because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors.

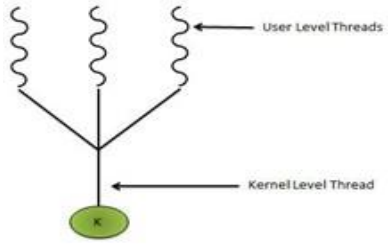

Fig: Many-to-one model

B. One to one model:

It maps each user thread to a corresponding kernel thread. It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call i.e. it also allows multiple threads to run in parallel on multiprocessor. The drawback is that creating a user thread requires creating the corresponding kernel thread and the overhead of creating the corresponding kernel thread can burden the performance of an application.

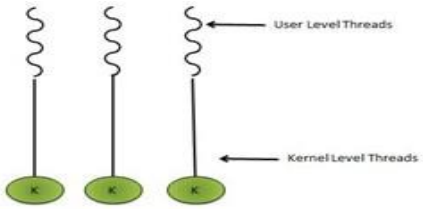

Fig: One-to-one model

C. Many- to - many model:

It multiplexes many user-level threads to a smaller or equal number of kernel threads. The number of kernel threads may be specific to either a particular application or a particular machine. This model allows the developer to create as many user threads as the user wishes and the corresponding kernel threads can run in parallel on a multiprocessor. In such case true concurrency is not gained because the kernel can schedule only one thread at a time.

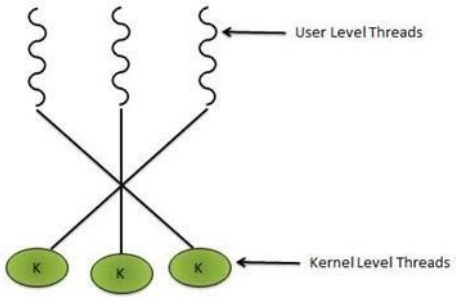

Fig: Many-to-many model

7. Difference between Process and Thread

S.N	Process	Thread
1	It is heavy weight	It is light weight, which takes lesser resources than a process
2	Process switching needs interaction with OS	Thread switching does not need interaction with OS
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, 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.