**TAKE-HOME ASSIGNMENT-1**

**Q2. What are Threads? What is the difference between user level threads and kernel level threads? During execution, how does it benefit to have kernel level threads? Describe in detail how threads, user-level or kernel-level, are treated for concurrency.**

**Answer:**

[[1]](#endnote-1)A **Thread** is a single sequence stream within a process. Threads have same properties as of the process, so they are also called as **light-weight processes**. Each thread belongs to exactly one process and no thread can exist outside a process. So, threads are designed to assist one another. Each thread represents a separate flow of control. Threads are executed one after another but gives the illusion as if they are executing in parallel.

Each thread has different states. Each thread consists of its own **program counter**, a **register set** and a **stack space.** But all the threads of a process run in **shared memory space** while processes run on separate memory spaces. So, threads are NOT independent of each other as they share the code section, data section and OS resources (like open files and signals).

**Advantages of Threads:**

* Context-switching time is lesser as compared to processes.
* Threads provide concurrency within a process.
* Threads are more economically feasible than processes in terms of creation and context-switching.
* Threads allow utilization of multiprocessor architectures to a greater scale and efficiency.

There are **Two types** of **Threads**:

* **User-Level Threads** are implemented in the user level library. They are not created using the system calls. Thread switching does not require OS intervention and does not require to cause an interrupt to Kernel. Kernel doesn’t know about the user level thread and manages them as if they were single-threaded processes. Hence, they can be even implemented in an OS which does not support threads.
* **Kernel-Level Threads** are managed by the Kernel. Since Kernel manages and schedules threads as well as processes, so it requires a **Thread Control Block (TCB)** for each thread. But instead of maintaining a thread table in each process, the kernel itself has a master thread table that keeps track of all the threads (i.e. all the TCBs) in the system. In addition, the kernel also maintains the traditional process table to keep track of the processes. OS kernel provides system calls to create and manage threads.

[[2]](#endnote-2)

|  |  |
| --- | --- |
| User-Level Thread | Kernel-Level Thread |
| 1. Implemented by the users. 2. They are invisible to the OS. So, they are not well integrated with the OS 3. They are generic and can run on any OS. 4. Easier implementation. 5. Their thread operations are much faster than Kernel-Level threads. 6. Context-switching time is less and NO hardware support is required. 7. If one thread performs any blocking operation, the entire process is blocked. 8. They are designed as dependent threads. 9. **Example:** POSIX threads. | 1. Implemented by OS. 2. They are well integrated with the OS. OS uses TCBs to maintain their information. 3. They are specific to a particular OS. 4. Complex implementation. 5. They are slower and inefficient as compared to the User-Level threads. 6. Context-switching takes more time and hardware support is required. 7. One thread’s blocking operation does not affect the other thread executions. 8. They are designed as Independent threads. 9. **Example:** Solaris2, BeOS etc. |

**Benefits of having Kernel-Level Threads during execution:**

* [[3]](#endnote-3)Since Kernel has full knowledge about all the threads, the Scheduler may decide to give more time to a process having large number of threads than a process having small number of threads.
* If one thread in a process is blocked, the Kernel can schedule another thread of the same process.
* Multiple threads of the same process can be scheduled on different processors.
* There can be multi-threaded kernel routines.

**Concurrency in Threads:**

In OS, **Concurrency** means interleaving of processes/threads in time to give the appearance of simultaneous execution. Itoccurs when multiple running processes/threads make progress and/or communicate with each other through shared memory or message-passing, but one at a time. In case of threads, concurrency is needed when there are more number of threads than processors **(**otherwise parallelism can be achieved if the number of processors are more**)**. It is achieved by **Multi-Threading**.

[[4]](#endnote-4)Concurrency is generally achieved by givingcertain time quantums to each thread typically using a **Time-Slicing** algorithm. While only one thread is executed at a time by a processor, the other threads can be switched in and out as required.

To ensure the mutual exclusion of execution of concurrent threads, the OS uses **mutex**. Mutex is a mutual exclusion object that synchronizes access to a resource. It is a locking mechanism which makes sure that only one thread can acquire it at a time and enter the **critical section**. When a thread locks a mutex, it has exclusive access to the shared resources i.e. ownership is associated with Mutex. Other threads attempting to lock the mutex are not going to succeed and have to wait till the thread which has locked the mutex (i.e. the owner thread) completes its task. The owner thread finally releases the mutex only when it exits the critical section.

[[5]](#endnote-5)Concurrency in threads can be implemented using different combinations of User-Level and Kernel-Level threads. These can be roughly classified into three types/models:

* **Many-to-Many Model**:
  + In this model, any number of User-Level threads can be multiplexed onto an equal or smaller number of Kernel-Level threads.
  + 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:** 
  + In this model, many User-Level threads are mapped to one Kernel-Level thread.
  + Thread management is done in user space by the thread library.
  + When a 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.
  + This model is used by the Kernel-Level threads when the User-Level thread libraries are implemented in the OS in such a way that the system does not support them.
* **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.
  + The main disadvantage of this model is that creating a User-Level thread requires the corresponding Kernel-Level thread.

1. https://www.geeksforgeeks.org/threads-and-its-types-in-operating-system/ [↑](#endnote-ref-1)
2. http://www.cs.iit.edu/~cs561/cs450/ChilkuriDineshThreads/dinesh's%20files/User%20and%20Kernel%20Level%20Threads.html [↑](#endnote-ref-2)
3. https://www.tutorialspoint.com/user-level-threads-and-kernel-level-threads [↑](#endnote-ref-3)
4. https://web.mit.edu/6.005/www/fa15/classes/19-concurrency/ [↑](#endnote-ref-4)
5. https://www.tutorialspoint.com/operating\_system/os\_multi\_threading.htm [↑](#endnote-ref-5)