**TAKE-HOME ASSIGNMENT-1**

**Q2. What are Threads? What is the difference between user level thread 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. 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 a **program counter**, a **register set** and a **stack space** but other threads can read and write in the stack memory. Threads are not independent of each other as they share the code, data, OS resources etc.

**Advantages of Threads:**

* Context-switching time is reduced.
* 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.
* **Kernel-Level Threads** are managed by the Kernel. Instead of thread table in each process, the kernel itself has a master thread table that keeps track of all the threads 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. OS doesn’t know about them. 3. These threads are generic and can run on any OS. 4. Easier implementation. 5. Context-switching time is less and NO hardware support is required. 6. If one thread performs any blocking operation, then the entire process is blocked. 7. They are designed as dependent threads. 8. **Example:** POSIX threads. | 1. Implemented by OS. 2. OS recognizes them. 3. These threads are specific to a particular OS. 4. Complex implementation. 5. Context-switching takes more time and hardware support is required. 6. One thread’s blocking operation does not affect the other thread executions. 7. They are designed as Independent threads. 8. **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:**

Concurrency in threads is achieved by **Multi-Threading**. [[4]](#endnote-4)When there are more threads than processors, concurrency is simulated by **Time-Slicing**, which means that the processor switches between threads. On most of the systems, time-slicing happens unpredictably and non-deterministically which means that a thread may be paused or resumed at any time.

[[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. 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:** 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 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 operating system 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 an User-Level thread requires the corresponding Kernel-Level thread. **Examples:** OS/2, windows NT and windows 2000 etc. use this model.

1. https://www.geeksforgeeks.org/threads-and-its-types-in-operating-system/ [↑](#endnote-ref-1)
2. https://www.geeksforgeeks.org/difference-between-user-level-thread-and-kernel-level-thread/ [↑](#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)