Threads in Operating System

A thread is a sequential flow of tasks within a process. Each thread has its own set of registers and stack space. There can be multiple threads in a single process having the same or different functionality. Threads are also termed lightweight processes.

Scope of Article

- This article contains the definition of thread, types of thread, and why threading is essential in the operating system.
- This article also explains the advantages and disadvantages of threading.

What is a Thread?

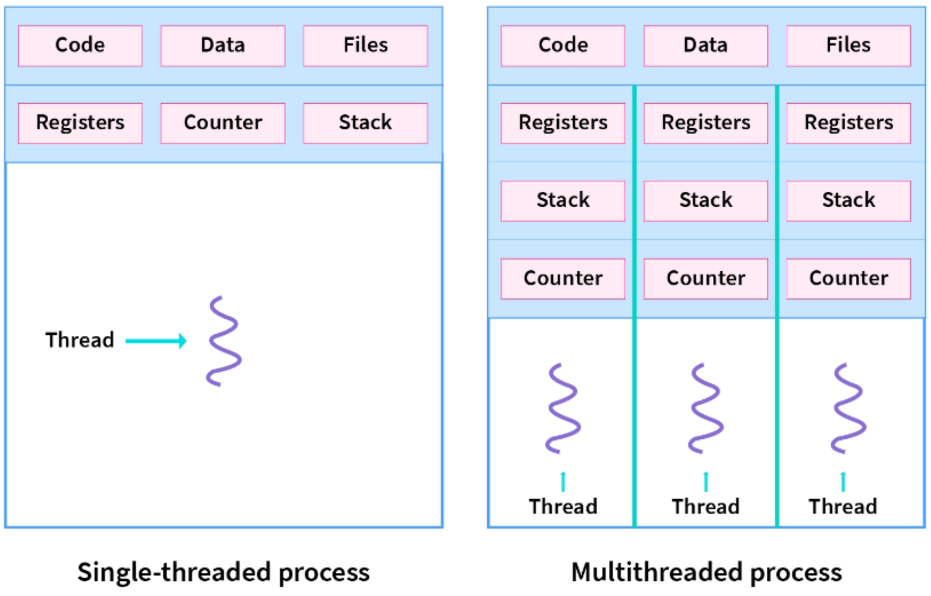
Let us take an example of a human body. A human body has different parts having different functionalities which are working parallelly (Eg: *Eyes*, *ears*, *hands*, etc). Similarly in computers, a single process might have multiple functionalities running parallelly where each functionality can be considered as a thread.

What is Thread in OS?

Thread is a sequential flow of tasks within a process. Threads in OS can be of the same or different types. Threads are used to increase the performance of the applications.

Each thread has its own program counter, stack, and set of registers. But the threads of a single process might share the same code and data/file. **Threads are also termed as lightweight processes as they share common resources**.

Eg: While playing a movie on a device the audio and video are controlled by different threads in the background.



The above diagram shows the difference between a single-threaded process and a multithreaded process and the resources that are shared among threads in a multithreaded process.

Components of Thread

A thread has the following three components:

1. Program Counter
2. Register Set
3. Stack space

Why do we need Threads?

Threads in the operating system provide multiple benefits and improve the overall performance of the system. Some of the reasons threads are needed in the operating system are:

- Since threads use the same data and code, the operational cost between threads is low.
- Creating and terminating a thread is faster compared to creating or terminating a process.
- Context switching is faster in threads compared to processes.

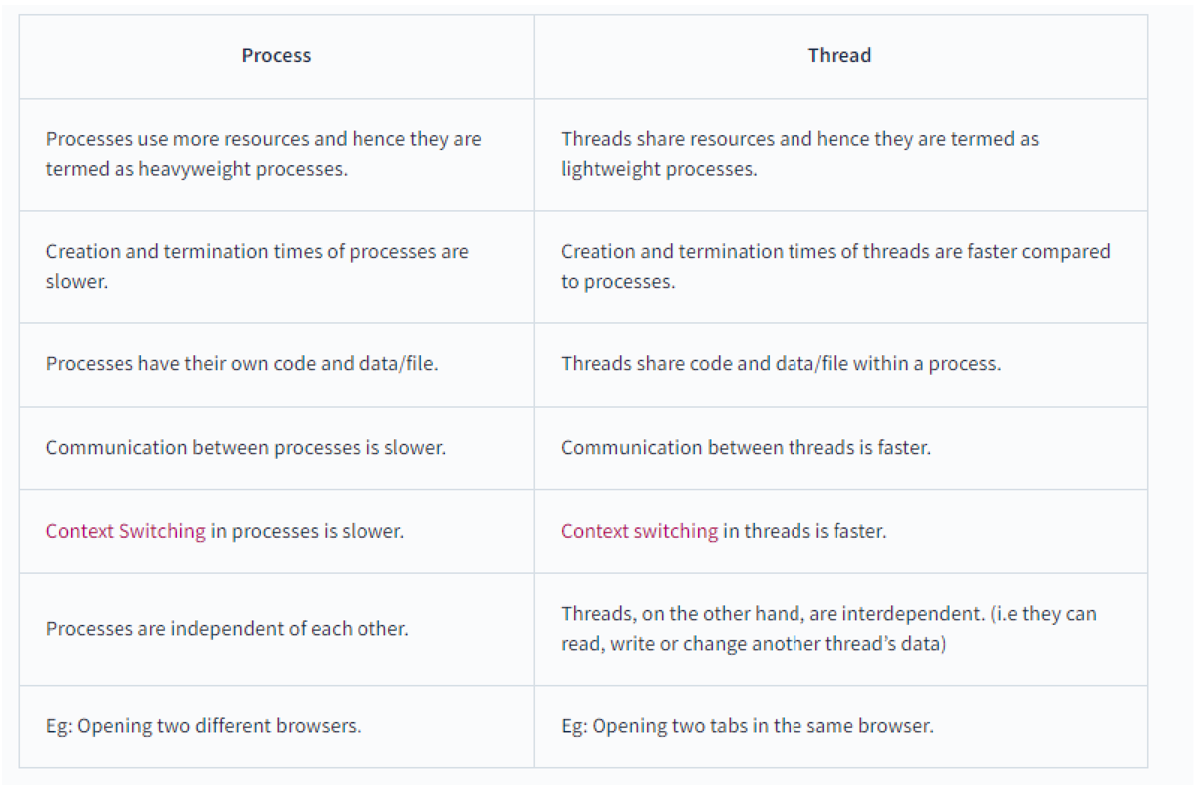
Why Multithreading?

In Multithreading, the idea is to divide a single process into multiple threads instead of creating a whole new process. Multithreading is done to achieve parallelism and to improve the performance of the applications as it is faster in many ways which were discussed above. The other advantages of multithreading are mentioned below.

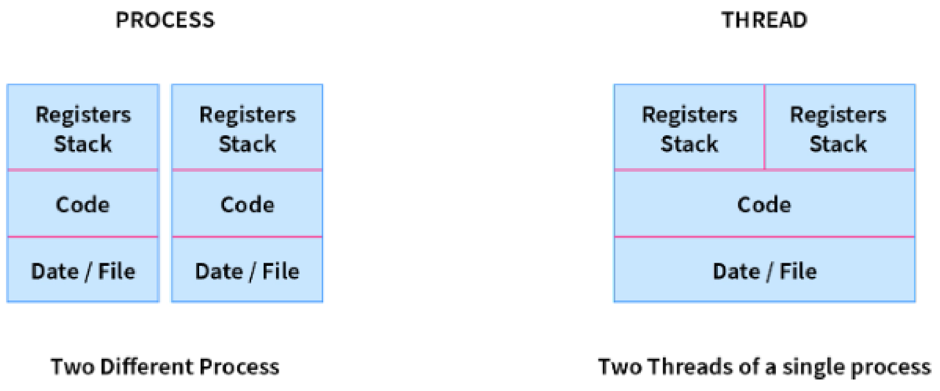
- **Resource Sharing:** Threads of a single process share the same resources such as code, data/file.
- **Responsiveness:** Program responsiveness enables a program to run even if part of the program is blocked or executing a lengthy operation. Thus, increasing the responsiveness to the user.
- **Economy:** It is more economical to use threads as they share the resources of a single process. On the other hand, creating processes is expensive.

Process vs Thread

Process simply means any program in execution while the thread is a segment of a process. The main differences between process and thread are mentioned below:



Process	Thread
Processes use more resources and hence they are termed as heavyweight processes.	Threads share resources and hence they are termed as lightweight processes.
Creation and termination times of processes are slower.	Creation and termination times of threads are faster compared to processes.
Processes have their own code and data/file.	Threads share code and data/file within a process.
Communication between processes is slower.	Communication between threads is faster.
Context Switching in processes is slower.	Context switching in threads is faster.
Processes are independent of each other.	Threads, on the other hand, are interdependent. (i.e they can read, write or change another thread's data)
Eg: Opening two different browsers.	Eg: Opening two tabs in the same browser.



The above diagram shows how the resources are shared in two different processes vs two threads in a single process.

Types of Thread

1. User Level Thread:

User-level threads are implemented and managed by the user and the kernel is not aware of it.

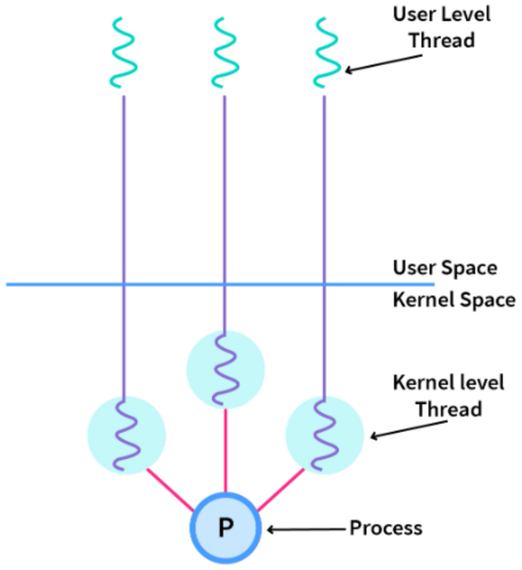
- User-level threads are **implemented using user-level libraries and the OS does not recognize these threads**.
- User-level thread is **faster to create and manage compared to kernel-level thread**.
- **Context switching in user-level threads is faster**.

- If one user-level thread performs a blocking operation then the entire process gets blocked. Eg: POSIX threads, Java threads, etc.

2. Kernel level Thread:

Kernel level threads are implemented and managed by the OS.

- Kernel level threads are **implemented using system calls and Kernel level threads are recognized by the OS**.
- Kernel-level threads are **slower to create and manage compared to user-level threads**.
- **Context switching in a kernel-level thread is slower**.
- Even if one kernel-level thread performs a blocking operation, it does not affect other threads. Eg: **Window Solaris**.



The above diagram shows the functioning of user-level threads in userspace and kernel-level threads in kernel space.

Advantages of Threading

- Threads improve the overall performance of a program.
- Threads increases the responsiveness of the program
- Context Switching time in threads is faster.
- Threads share the same memory and resources within a process.
- Communication is faster in threads.
- Threads provide concurrency within a process.
- Enhanced throughput of the system.
- Since different threads can run parallelly, threading enables the utilization of the multiprocessor architecture to a greater extent and increases efficiency.

Issues with Threading

There are a number of issues that arise with threading. Some of them are mentioned below:

- **The semantics of fork() and exec() system calls: The fork() call is used to create a duplicate child process**. During a fork() call the issue that arises is whether the whole process should be duplicated or just the thread which made the fork() call should be duplicated. **The exec() call replaces the whole process that called it including all the threads in the process with a new program**.
- **Thread cancellation:** The termination of a thread before its completion is called thread cancellation and the terminated thread is termed as target thread. Thread cancellation is of two types:

1. **Asynchronous Cancellation:** In asynchronous cancellation, one thread immediately terminates the target thread.
2. **Deferred Cancellation:** In deferred cancellation, the target thread periodically checks if it should be terminated.

- **Signal handling:** In UNIX systems, a signal is used to notify a process that a particular event has happened. Based on the source of the signal, signal handling can be categorized as:

1. **Asynchronous Signal:** The signal which is generated outside the process which receives it.
2. **Synchronous Signal:** The signal which is generated and delivered in the same process.

Conclusion

- Thread is a sequential flow of tasks within a process.
- There can be multiple threads in a single process.
- A thread has three components namely Program counter, register set, and stack space.
- Thread is also termed as the lightweight process as they share resources and are faster compared to processes.
- Context switching is faster in threads.
- Threads are of two types:

1. User Level Thread: User-level threads are created and managed by the user.
2. Kernel Level Thread: Kernel-level threads are created and managed by the OS.

- Issues related to threading are fork() and exec() system call, thread cancellation, signal handling, etc.
- Some of the advantages of threading include responsiveness, faster context switching, faster communication, concurrency, efficient use of the multiprocessor, etc.