When you approach operating system concepts there might be several confusing terms that may look similar but in fact refer to different concepts.

In this post, I will try to clarify four of such terms which often cause perplexity: those are *multiprogramming*, *multiprocessing*, *multitasking*, and *multithreading*.

In a modern computing system, there are usually several concurrent application processes which compete for (few) resources like, for instance, the CPU. As we have already introduced, the Operating System (OS), amongst other duties, is responsible for the effective and efficient allocation of those resources. Generally speaking, the OS module which handles resource allocation is called scheduler. On the basis of the type of OS to be realized, different scheduling policies may be implemented.

Multiprogramming
In a multiprogramming system there are one or more programs loaded in main memory which are ready to execute. Only one program at a time is able to get the CPU for executing its instructions (i.e., there is at most one process running on the system) while all the others are waiting their turn.
The main idea of multiprogramming is to maximize the use of CPU time. Indeed, suppose the currently running process is performing an I/O task (which, by definition, does not need the CPU to be accomplished). Then, the OS may interrupt that process and give the control to one of the other in-main-memory programs that are ready to execute (i.e. *process context switching*). In this way, no CPU time is wasted by the system waiting for the I/O task to be completed, and a running process keeps executing until either it voluntarily releases the CPU or when it blocks for an I/O operation. Therefore, the ultimate goal of multiprogramming is to keep the CPU busy as long as there are processes ready to execute.
Note that in order for such a system to function properly, the OS must be able to load multiple programs into separate areas of the main memory and provide the required protection to avoid the chance of one process being modified by another one. Other problems that need to be addressed when having multiple programs in memory is *fragmentation* as programs enter or leave the main memory. Another issue that needs to be handled as well is that large programs may not fit at once in memory which can be solved by using *pagination* and *virtual memory*. Please, refer to this article for more details on that.
Finally, note that if there are `N` ready processes and all of those are highly CPU-bound (i.e., they mostly execute CPU tasks and none or very few I/O operations), in the very worst case one program might wait all the other `N-1` ones to complete before executing.

Multiprocessing
Multiprocessing sometimes refers to executing multiple processes (programs) at the same time. This might be misleading because we have already introduced the term “multiprogramming” to describe that before.
In fact, multiprocessing refers to the *hardware* (i.e., the CPU units) rather than the *software* (i.e., running processes). If the underlying hardware provides more than one processor then that is multiprocessing. Several variations on the basic scheme exist, e.g., multiple cores on one die or multiple dies in one package or multiple packages in one system.
Anyway, a system can be both multiprogrammed by having multiple programs running at the same time and multiprocessing by having more than one physical processor.

Multitasking
Multitasking has the same meaning of multiprogramming but in a more general sense, as it refers to having multiple (programs, processes, tasks, threads) running at the same time. This term is used in modern operating systems when multiple tasks share a common processing resource (e.g., CPU and Memory). At any time the CPU is executing one task only while other tasks waiting their turn. The illusion of parallelism is achieved when the CPU is reassigned to another task (i.e. *process* or *thread context switching*).
There are subtle differences between multitasking and multiprogramming. A *task* in a multitasking operating system is not a whole application program but it can also refer to a “thread of execution” when one process is divided into sub-tasks. Each smaller task does not hijack the CPU until it finishes like in the older multiprogramming but rather a fair share amount of the CPU time called quantum.
Just to make it easy to remember, both multiprogramming and multitasking operating systems are **(CPU) time sharing** systems. However, while in multiprogramming (older OSs) one program as a whole keeps running until it blocks, in multitasking (modern OSs) time sharing is best manifested because each running process takes only a fair quantum of the CPU time.

Multithreading
Up to now, we have talked about multiprogramming as a way to allow multiple programs being resident in main memory and (apparently) running at the same time. Then, multitasking refers to multiple tasks running (apparently) simultaneously by sharing the CPU time. Finally, multiprocessing describes systems having multiple CPUs. So,

where does multithreading come in?
Multithreading is an execution model that allows a single process to have multiple code segments (i.e., *threads*) run concurrently within the "context" of that process. You can think of threads as child processes that share the parent process resources but execute independently. Multiple threads of a single process can share the CPU in a single CPU system or (purely) run in parallel in a multiprocessing system
Why should we need to have multiple threads of execution within a single process context?
Well, consider for instance a GUI application where the user can issue a command that require long time to finish (e.g., a complex mathematical computation). Unless you design this command to be run in a separate execution thread you will not be able to interact with the main application GUI (e.g., to update a progress bar) because it is going to be unresponsive while the calculation is taking place.
Of course, designing multithreaded/concurrent applications requires the programmer to handle situations that simply don't occur when developing single-threaded, sequential applications. For instance, when two or more threads try to access and modify a shared resource (*race conditions*), the programmer must be sure this will not leave the system in an inconsistent or deadlock state. Typically, this thread synchronization is solved using OS primitives, such as **mutexes** and **sempaphores**.

A Side Note on Context Switching
The concept of *context switching* applies to multiprogramming as well as to multitasking yet at a different level of granularity. In the former, the "context" refers to that of a whole process whereas in the latter the "context" may be that of a lighter thread. In fact, **process context switching** involves switching the virtual memory address space: this includes memory addresses, mappings, page tables, and kernel resources. On the other hand, **thread context switching** is context switching from one thread to another in the **same** process (i.e. there is no need of switching the virtual memory address space as the "switcher" and the "switchee" threads share the same virtual address space). This requires switching processor state (such as the program counter and register contents), which is generally very efficient.
Of course, switching from thread to thread across different processes is just like process context switching.