# Week#7

## Practice Exercises and Solutions

1. Provide two programming examples in which multithreading can provide better performance than a single-threaded solution.

   Answer:

   (1) A Web server that services each request in a separate thread.
   (2) A parallelized application such as matrix multiplication where different parts of the matrix may be worked on in parallel.
   (3) An interactive GUI program such as a debugger where a thread is used to monitor user input, another thread represents the running application, and a third thread monitors performance.

2. What are two differences between user-level threads and kernel-level threads? Under what circumstances is one type better than the other?

   Answer:

   (1) User-level threads are unknown to the kernel (they are just managed within the process), whereas the kernel is aware of kernel threads.
   (2) On systems using M:1 mapping, user threads are scheduled by the thread library *within* the process - the kernel schedules kernel threads.
   (3) Kernel threads need not be associated with a process whereas every user thread belongs to a process. Kernel threads are generally more expensive for the operating system to maintain than user threads as they must be represented with a kernel data structure.

3. Describe the actions taken by a kernel to context-switch between kernel-level threads.

   Answer: Context switching between kernel threads (within the same process) typically requires saving the value of the CPU registers from the thread being switched out and restoring the CPU registers of the new thread being scheduled.

4. What resources are used when a thread is created? How do they differ from those used when a process is created?

   Answer: Because a thread is smaller than a process, thread creation typically uses fewer resources than process creation. Creating a process requires allocating a process control block (PCB), a rather large data structure. The PCB includes a memory map, list of open files, and environment variables. Allocating and managing the memory map is typically the most time-consuming activity. Creating either a user or kernel thread involves allocating a small data structure to hold a register set, stack, and priority.

5. What is the relationship between threads and processes?
   Answer: A processes is a container for threads, it has its own memory space. A process may contain one or more threads, which share that memory space, all of the file descriptors and other attributes. The threads are the units of execution *within* the process, they each possess a register set, stack, program counter, and scheduling attributes.

6. Describe how a multi-threaded application can be supported by a user-level threads package. It may be helpful to consider (and draw) the components of such a package, and the function they perform.
   Answer: From the kernel's perspective, each process has its own address space, file descriptors, etc. and at least one thread of execution. To support multiple threads at *user-level*, the process contains code to create, destroy, schedule and synchronize user-level threads - which can be thought of as multiplexing many user-level threads onto the single kernel thread, all managed within the process. The scheduler can run any arbitrary scheduling algorithms, and is independent of the kernel's scheduler.

   User-level threads can also implement a huge number of threads if necessary, as it can take advantage of virtual memory to store user-level thread control blocks and stacks.

   These are often cooperative schedulers, as there is usually only rudimentary support (if any) for delivering timer ticks to the user-level scheduler. However the application must be specifically written for cooperative scheduling, contains yields such that other threads have an opportunity to run; a single badly written thread without enough yields can monopolise CPU time. Alternatively, these can also be preemptive multitaskers, where the user level thread package receives regular signals from the operating system, which may initiate the context switch. These have high granularity though and aren't really practical.

   Another important consideration is making system calls non-blocking, so that the user-level thread scheduler can schedule (overlap) execution of another thread without blocking the whole process. This is not always possible (either because of a lack of async I/O system calls), or because events such as page faults (which are always synchronous) will block the process and all the user-level threads inside.