# CS 390
# Chapter 4 Homework Solutions

**4.1** `Provide three programming examples in ...`

There are many examples. Think of programming tasks where there are independent computations to be performed. A single-threaded solution can only do one of these computations at a time, but a multithreaded solution could work on several concurrently. The performance gain is especially clear when one of the computational tasks involves waiting.

1. A web browser: A multithreaded web browser could divide up the display of web pages by having one thread draw the web page on the screen, while one or more additional threads are reading the contents of the page (text, images, videos, advertisements, etc.) from the network or from the browser's cache.
2. A compiler: A multithreaded compiler could have one thread performing lexical analysis while another thread does syntax analysis.
3. A database engine: One thread could read and parse queries from users, while another thread could execute the queries by reading / writing information from/to the database tables.

**4.2** `Using Amdahl's Law, calculate the ...`

Since $S = 0.4, P = 0.6$.

**(a)** $N = 2$, so speedup $\leq \frac{1}{0.4+\frac{0.6}{2}} \approx 1.43$.

**(b)** $N = 4$, so speedup $\leq \frac{1}{0.4+\frac{0.6}{4}} = \approx 1.82$.

**4.4** `What are two differences between ...`

Here are four differences.

- Kernel threads must be supported by the kernel. User threads do not require any kernel support.
- User threads are invisible to the kernel. With kernel threads, the kernel is aware of the processes in the system, and also any threads contained within the processes.

- When one user thread blocks, all threads in the same process block. In a kernel-thread, threads that are not in the same process as the blocked thread can continue to execute.
- With user threads, the threading library schedules the threads. The threading library is just another segment of code in the process. With kernel threads, the kernel schedules the different processes that make up the multi-threaded computation.

User threads are the only option if the kernel does not provide thread support. Under this model, if one thread is blocked, the entire computation is blocked. If the kernel provides support for threads, than the system can use the one-to-one or many-to-many threading model. Under these models, a blocked thread does not necessarily block the entire computation.

**4.5** `Describe the actions taken by ...`

Kernel threads are regular user processes except that they share text, data and heap segments. Thus, switching between kernel threads is the same as switching between processes.

Additional Exercises

**A** `Give some examples where multithreading does not provide better performance than a single-threaded solution.`

**B** `Which of the following components of a program state are shared across threads in a multithreaded computation: register values, heap memory, global variables, stack memory?`

**C** `Can a multithreaded solution using multiple user-level threads achieve better performance on a multiprocessor system than on a single-processor system? Explain.`

**D** `Explain how a system can achieve concurrency without parallelism.`

Answers to Additional Exercises

**A** Any computation that is inherently sequential (that is, one that cannot be parallelized) will not benefit from multithreading.

Imagine an application where you need to perform a series of ten transformations on an item of data. Each transformation must be completed before the next transformation can be done. If we create a thread to perform each transformation, then no thread can execute before the thread before it completes its transformation, and after a transformation is complete, there is no more work for the thread to perform. This is no faster than having a single thread perform all ten transformations sequentially, and in fact is probably slower due to the overhead of managing the threads.

I/O-bound computations typically do not see a performance benefit from multithreading. Suppose an application needs to create a file consisting of 100 pieces of structured data. (As an example, consider a database application initializing a new table.) If we create 100 threads to perform this task, each thread may make a write request concurrently, but the writes themselves will still take place sequentially, since a file resides on a single physical disk, and a single disk cannot perform multiple writes at the same time. (In chapter 10, we will discuss RAID, a technique that allows the OS to parallelize disk operations.)

**B** **register values** Not shared. Since each thread is potentially executing a different statement in the program, each thread must have its own private set of registers.

**Heap memory** Shared.

**Global variables** Shared.

**Stack memory** Not shared. Since each thread may be executing a different function in the program, each thread needs its own stack.

**C** No. If a system supports only user-level threads, then all threads reside in a single process. Since the OS assigns each process to a single processor, the multithreaded solution cannot take advantage of multiple processors.

**D** The term "concurrent computation" means that multiple threads or

processes *appear* to execute simultaneously. Parallelism means that multiple threads or programs actually execute simultaneously.

A multithreaded application running under any threading model on a uni-processor system will be concurrent, even though the system never executes more than one thread at the same time. Multithreaded applications running under the one-to-one model or the many-to-many model on a multiprocessor system can actually execute multiple instructions at the same time, achieving parallelism.

The same argument holds true for systems without threads. Concurrency is achieved on uni-processor machines by switching between processes. This is just our old friend multitasking.