Section 8

Goals
1. To understand what multithreading is and why it is important.
2. To understand the practical applications of multithreading and its limitations.
3. To be able to create a basic program that utilizes multiple threads.

Introduction
In the early days of computer science, hardware manufacturers were only concerned with making their CPUs faster. A CPU, or a central processing unit, is the part of the computer that carries out the instructions in a program. For a while, this strategy worked. The Commodore 64, released in 1982, had a CPU that ran at approximately 1 MHz. By 2001, Intel’s Pentium 4 processor had a clock speed of 1.3 Ghz, over 1300 times faster.

Unfortunately, hardware manufacturers hit a wall. The faster they made their CPUs run, the more heat was produced and the more power was consumed at an exponential rate. The solution was multicore processors. These processors have more than one core, the unit that reads and executes instructions. Rather than make a CPU do one task faster, the idea was to allow the CPU to do multiple tasks at once, at the same or a slightly reduced speed.

Initially this concept was slow to catch on. Programs that had been written for single core processors saw no performance gain, and multithreaded programs were more difficult to write, as programmers ran into new problems when using multiple threads. Yet despite the complications created by multithreading, the benefits that multicore processors offered made them the superior choice, and modern programmers adapted by making their programs capable of utilizing the parallel processing power of these new CPUs.

The first popular dual-core consumer CPUs were released in 2005. Compare that to today, where most flagship smartphones boast quad-core CPUs.

Multithreading
Imagine that you are running a program by reading through a Java program and calculating each instruction manually row by row. This is known as a thread - a sequence of instructions that a processor runs, and is similar to how a CPU executes instructions in a program. The idea of multithreading then, is to give the processor the capability to read multiple sequences of instructions at once. Think of it as having several classmates working on other parts of the program while you work on your own section.

In order to benefit from multithreading, the processor must have more cores to run other threads - like how you would need to have classmates to calculate other parts of the program. This is why multithreading is most effective on multicore processors. Some processors have what is known as hyperthreading, the ability to run twice the amount of threads on one core. This becomes useful because when the CPU is waiting for one instruction to finish, it can switch threads and start executing instructions from the second thread.

It becomes evident that not everything can be multithreaded. Calculating the second half of a program, for example, will not work if you need data from the first half. Most code cannot be parallelized for this reason. However, there are a few situations where multithreading comes in useful. If you want to crunch a large amount of data that does not need the results of the rest of the data, multithreading is key.
For example, imagine we are reading in several million text files and counting the number of times a certain word appears. By splitting the list of text files into two smaller lists, with a different thread handling each list and counting side by side, the program becomes approximately 200% faster. If the lists are further split into four lists, it will be about 400% faster. This can be continued until there are more threads than can be handled at one time. Remember that there will always be some part of the code that cannot be parallelized.

This introduces the interesting problem that in order for the program to continue, all of the threads must finish their calculations. If the program attempts to continue when some of the threads are still running, this may result in unexpected behavior. Furthermore, while multithreading will almost always be faster than having a single thread handle all the calculations, if the threading is done poorly, there will not be much gain at all.

Let us take the file reading example. If we continue the program before all the threads have finished - before all the text files have been read - then the wrong count value will be returned. Furthermore, we are operating under the assumption that all the text files are the same length. If one of the lists has text files that are much larger, then that thread will take much longer to finish. As you can see, properly implementing multiple threads is not an easy task.

**Program**

So, now that you have a better understanding of threads, let’s look at a problem. We want to find all the prime numbers under 1000000 and print it out. The provided program, Prime.java, already does this, but it is very slow, since it has to check every individual number’s primality on the main thread. What can we do to make it run faster? Why, using multiple threads of course! We will be modifying the code inside CS8BThread.java in order to have threads able to check primality, and modify the code inside Prime.java in order to use these threads.

First, in Prime.java, let’s comment out lines 8-12 (ie. the second for loop). We won’t be needing that logic anymore, since the majority of the workload shouldn’t be on the main thread. Then we need to split the original array of 1000000 into two equal arrays. You could use the System.arraycopy() method for this, or you could find/come up with your own way to do this. After that, create two new CS8BThreads, and pass in the array halves as the parameter. Finally, call start upon both of the CS8BThreads.

However, CS8BThread actually doesn’t actually do anything as of now. To fix this, you will need to modify the run method so that it is able to find prime numbers. This should look very similar to the code already found in Prime.java. Alternatively, feel free to use your own algorithms to check for primality (Question 6 looks fun).

The program should be ready to run after all the proper changes were made. Try running it and saving the output into a file output.txt. To do this, you will need to redirect the output like so:

```
$ java Prime > output.txt
```

Hopefully, you noticed the improvement in runtime. Splitting the array into four equal parts would theoretically double the program’s performance, and eight parts would quadruple it, and so forth. Enjoy finding relatively large prime numbers in roughly half the time!

**Questions**

1. What are the benefits of multithreading?
2. What sort of problems can occur when multithreading a program?
3. Why isn’t all code parallelizable?
4. Come up with an example of when multithreading might come in useful.
5. Note that even the current multithreaded solution is not optimal. The last thread, which will get the section of the array with the largest numbers, will take much longer to finish because it will have more operations to run. Can you come up with a better implementation? Try to split the array in a way that each thread will have approximately the same number of calculations to do.
6. Convert the isPrime() method in Prime.java to a recursive function.