CS 226: Operating systems and networking
Spring Term, 2008

Lab 5

Multiple threads

In this lab, you will gain some exposure to the Java `Thread` class and also see some of the issues in parallel programming. Begin with the code `SerialPrimes.java` in the course directory. This is a program that counts the number of prime numbers (those only divisible by 1 and themselves) between 1 and 2,000,000. Look through this program, compile it, and then time a run of it with the command line

```
time java SerialPrimes
```

Approximately 19 seconds later, you should be told that there are 148,933 primes in the range 1–2,000,000.

In addition to the output from the program itself, the `time` program will print out some information on your program's running time. The first two numbers are the amount of CPU time the program took while in user mode (i.e. not during system calls) and the amount of system time (i.e. during system calls). The third number is the total amount of time between when the program started and when it ended; this is called the *wall clock time*.

Now that you are familiar with the serial (i.e. non-parallel) implementation, it is time to take a look at a naive multi-threaded implementation. Get a copy of `ThreadedPrimes.java` from the course directory and look through its code. Instead of a prime-finding loop in `main`, the work is done by `PrimeFinder` objects, each of which is responsible for finding primes within a range specified during object construction. The `main` method creates two of these objects with complementary ranges and assigns one of them to each of two threads. To allow this, `PrimeFinder` implements the `Runnable` interface, which simply requires the `run` method. Once the work is assigned to each thread, the threads are started by calling their `start` methods.

Compile and run this program. The results are unimpressive; the program dramatically undercounts the number of primes. The problem is that `main` is printing the number of primes before the `PrimeFinder` objects have completed their counts. To fix this, add the lines

```
t1.join();
t2.join();
```

after the threads are started. The `join` method does not return until the thread has died.[1] Thus, these calls delay the printing of results until both of the counting threads have completed.

After adding these lines, recompile the program and run it again. The counted number of primes still does not agree with the value given by the serial program. Can you identify the cause of this difference? See if you can figure it out before turning to the other side of the page.

[1] Technically, the thread can also have been interrupted by another part of the program, though this will not occur in our program. If this had occurred, an `InterruptedException` would be thrown, which is why our `main` method is noted to throw this type of exception.

The problem is that our program suffers from a race condition when it updates `pCount`. Although `pCount++` is a single Java statement, the thread can be interrupted in the middle of it, leading to some primes not being included in the count. To fix this, we will add a lock around the increment line. Create a `static` object called `lock` in the `ThreadedPrimes` class. (The type does not matter; I made mine an `Object`.) Then, embed the line incrementing `pCount` inside a block

```
synchronized(lock) {
  ...
}
```

With this, the increment can only occur after the lock associated with the `lock` object has been acquired. In Java, every object has a lock associated with it; we just made a static object so it could be shared by the two occurrences of `PrimeFinder`. The `synchronized` statement automatically acquires the lock before entering the protected block of code and automatically releases it at the block's end.

After this change, the number of primes should be properly calculated. One potential criticism of this solution, however, is that the two threads constantly need the lock. This does not cause the program much overhead since the critical section is so short, but a better solution is possible. Our prime algorithm is what is called *embarrassingly parallel* because it is so easy to break into subproblems that can run independently; testing the primality of each number is completely independent of all the other numbers. With a problem like this, it seems wasteful to use the lock so heavily. Instead, modify the solution so that each `PrimeFinder` object keeps its own count of the number of primes it has found. Then, after it has counted all the primes within its range, it should update `pCount` (using a lock of course). With this modification, each thread only claims the lock once.

So how fast is the resulting program? Run it using `time` and compare the result to the serial version. The first two numbers should be nearly identical between the two implementations because they do almost exactly the same things and therefore need the same amount of CPU time. The wall clock times should differ, though. The *speedup* of our parallel implementation is the serial wall clock time divided by the parallel wall clock time. Since the lab machines have two processors and we are creating a thread to run on each, an ideal speedup would be 2. In practice, parallel programs cannot normally achieve *linear speedup*, as this would be called, since some parts of the program run serially and communication (the lock in our case) incurs some overhead. However, we should be able to get a speedup extremely close to 2 for this program because it splits into parallel subproblems so well.

Unfortunately, I get a speedup of only about 1.6. Can you explain this? (Hint: What happens when you set the two parts to find primes within the ranges 1-1,100,000 and 1,100,001–2,000,000 instead splitting the test region evenly?) Once you understand the problem, see if you can fix it. (There are at least two fairly straightforward ways to do so.)

If you have extra time, continue to experiment with threaded programs to find prime numbers. What happens if you create more than two threads? Another way to speed up the computation is to only divide a candidate number by the primes smaller than its square root rather than all values smaller than its square root; can you use this to further improve your performance? (You will probably want to use the `Vector` class, which is a synchronized version of `ArrayList`.)