# Lab 6

*30 Apr 2009*

A *thread* is essentially a lightweight process, where you can have multiple threads of execution within the same process that each have their own call stack. It's different from a regular process, however, in that it shares certain resources at the split point rather than making copies of them; this means that some of the inter-process communication we've been discussing, which requires shared variables, can be most straightforwardly implemented using threads.

In this lab, you will gain some exposure to the POSIX thread library, a widely-implemented standard for threads, and also see some of the issues involved in parallel programming. Begin with the code `serial_primes.c` 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 20,000,000. Copy it into your own directory and compile with the command line

```
  cc serial_primes.c -o serial_primes
```

(This creates the executable as `serial_primes` rather than `a.out`.) Use the shell construct `time` to time the run as follows:

```
  time serial_primes
```

Approximately 18 seconds later, you should be told that there are 1,270,607 primes in that range.

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.*

## 1 Threads

Now that you are familiar with the serial (i.e. non-parallel) implementation, it is time to take a look at a naïve multi-threaded implementation. Get a copy of `threaded_primes.c` from the course directory and look through its code. The grunt work is still done by `count_primes`, but in a more indirect fashion and in two batches. What `main` now does is create two threads, each of which runs a single function (`prime_thread`) with a single argument (a pointer to `t1arg` or `t2arg`). That function is then responsible for the call to `count_primes`.

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

```
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
```

after the threads are started. (Actually, you should check their return values as well—like `pthread_create`, they return 0 on success.) Much like the process-oriented system call `wait`, the `pthread_join` function does not return until the thread has died. If the function called at the start of the thread (in this case `prime_thread`) had a return value that we cared to process, we could retrieve that value via the second argument to `pthread_join`. 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. Compare the timed results to those of the serial version. There are two important differences: first, the threaded version runs faster, due to both threads being able to run simultaneously. Second, the threaded version slightly undercounts the number of primes. Try to identify where the problem lies before continuing to the next section. (Hint: run the threaded version a few more times. What do these results suggest?)

## 2 Fixing the problem

The problem is that our program suffers from a race condition when it updates `pCount`. Although `++*pCount` is a single statement, it is compiled into multiple statements in machine language (at least on these systems), and so 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 create a mutex—a semaphore specifically aimed at controlling access to a critical region—to prevent interruption during the increment line. Copy your existing `threaded_primes.c` to a backup file so that you will be able to compare its performance to your later versions.

First, a single mutex variable needs to be created that both threads will have access to. So, at the start of `main`, create the mutex and init it:

```
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL);
```

Then, modify the `prime_arg` struct, the `prime_thread` function, and the `count_primes` function so that a pointer to the mutex can be passed along into the actual prime-counting code.

Finally, establish the actual critical region by preceding the increment with a call to `pthread_mutex_lock` and following it with a call to `pthread_mutex_unlock`. With this, the increment can only occur after the lock associated with the `mutex` object has been acquired.

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, once again, save a copy for later comparison and then modify your solution so that each `count_primes` call 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 results to the serial version and the earlier threaded versions. 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 multiple processors and we are creating two threads, each thread runs on a different processor and 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?