CSc 422/522 — Homework 2

due Tuesday, February 22

The first four problem are worth 10 points each. The last problem is worth 20 points. Graduate students are to solve all problems (60 points). Undergraduates are to solve any combination of problems that adds to 40 points.

Again, the work you turn in must be your own. Be sure to explain clearly and succinctly what you are doing; don't just give an answer.

1. Critical sections using spin locks.

(a) MPD book, Exercise 3.2. `DEC` and `INC` are machine instructions, so their arguments are passed by reference (address). If you wish you can assume that `DEC` and `INC` are functions that return the value of `sign`.

(b) Discuss the performance of your solution on a shared-memory multiprocessor with `N` processors. Assume that `N` processes, one per processor, arrive at the critical section at the same time. What will happen?

(c) Modify your solution so that it performs better on a multiprocessor. Explain what you would change and discuss the performance of your new solution assuming that `N` processes arrive at the same time as in part (b).

2. Critical sections using a coordinator process.

(a) MPD book, Exercise 3.12. Just solve part (b); namely, develop a fair solution.

(b) Discuss the performance of your solution on a shared-memory multiprocessor with `P` processors. Assume that there are `P-1` regular processes and one coordinator, that each process executes on its own processor, and that the regular processes arrive at the critical section at the same time. What will happen?

3. Barrier synchronization.

(a) Assume that there are `N` worker processes in a parallel program; each has a unique identity between `1` and `N`. Develop a procedure `barrier(int id)` that the workers can call for barrier synchronization. In particular, after *all* `N` workers have called `barrier`—but not before—then each call should return. Your procedure should implement a reusable dissemination barrier. Use the `await` statement or `while` loops within the body of the procedure in order to delay processes. Write your procedure in SR or in the pseudo-C notation in the textbook.

(b) Suppose that each assignment to a flag variable takes 1 unit of time and that each `await` statement or spin loop takes 3 units of time. What is the *best case* execution time for the barrier? In particular, if *all* workers call `barrier` at the same time, how long will it take before every call has returned? Do not count procedure call overhead; just count the code in the body of the procedure.

4. Concurrent execution in Pthreads.

The purposes of this problem are to introduce you to the Pthreads library and to let you see the effects of not protecting critical sections. The Pthreads library is described in Section 4.6 of the text. You might also want to look at the manual pages for `pthreads` and for the various functions used in the four sample programs handed out in class. (These programs are stored in `/home/cs522/SamplePrograms`.)

Assume that `x`, `y`, and `z` are global integer variables that are all initially zero. Consider the following three statements:

```
    S1:  x = x+1;
    S2:  y = y+1;
    S3:  z = x+y;
```

(a) Write a Pthreads program that has two processes. Each process executes the above three statements 1000 times. At the end of the program, write out the final values of the three variables. Repeat this test five times. Compile your programs on `lec`, but run your tests on `par`.

(b) Repeat part (a), but now make *each* assignment statement atomic. Thus, the body of the loop in each process will look like:

```
    ⟨S1;⟩ ⟨S2;⟩ ⟨S3;⟩
```

Use Pthreads semaphores to protect critical sections.

(c) Repeat part (a), but this time combine all three assignment statements into a single atomic action:

```
    ⟨S1; S2; S3;⟩
```

Hand in commented listings of your programs together with the output from the five test runs of each program. Please also use `turnin` to provide us with copies of your programs. See below for details.

5. Prime number generation using a bag of tasks.

Write a parallel program that generates prime numbers using the *bag of tasks* paradigm described in Section 3.6. There should be `W` worker processes, where `W` is a command-line argument between `1` and `4`. The program should calculate all primes up to a limit `L`, which is also a command-line argument. You may write your program in SR or in C plus Pthreads. Use semaphores to implement any atomic actions you need, but use busy waiting (spin loops) to program delays. Also use static arrays; *do not use* dynamic memory allocation.

One way to solve this problem is to mimic the sieve of Eratosthenes in which you have an array of `L` integers and repeatedly cross out multiples of primes: 2, 3, 5, ... In this case the bag would contain the next prime to use. This approach is easy to program, but it uses lots of storage. In particular, you need to have an array of `L` integers.

A second approach is to check all odd numbers, one after the other. Hence, the bag would contain odd numbers. For each candidate, see whether it is prime, and if so, add the number to a growing list of known primes. The list of primes is used to check future candidates. This second approach requires far less space than the first approach. However, it has a tricky little synchronization problem to keep the list of known primes sorted.

(a) Write a program that implements the second approach. Represent the bag by a single integer, whose value is the next odd candidate. When a worker needs a new task, it atomically reads the value of the next candidate, then increments that value by two. If the worker finds that the candidate is prime, it atomically inserts it into the sorted list of known primes. Initialize the bag to the value `9` and initialize the list of known primes to `3`, `5`, and `7`. At the end of your program, print the execution time for the computational part, then print the last 10 primes you found.

(b) Modify your program to make it faster. Your goal is to find the largest prime that you can in 20 seconds. The bag is a bottleneck, because the workers compete for access to it. You can get rid of *all* contention for the bag by using `W` bags, one per worker. It is also possible to separate the list of known primes into `W` separate lists, but this is trickier to do because each worker needs to be sure it has checked all possible prime factors before concluding that a candidate is prime.

Compile your programs on `lec` and run timing tests on `par`. Test your programs for various values of `W` and `L`. Use up to four workers. Try small values of `L` when you are doing debugging, then try *large* values of `L`.

Hand in a commented listings of your two programs. Also turn them in electronically.

Electronic Turnin.

Use the `turnin` program on Lectura to turn in copies of your programs. For problem 4, the assignment name to use is `hw2.problem4`. The file names should be `parta.c`, `partb.c`, and `partc.c`.

For problem 5, the assignment name to use is `hw2.problem5`. The file names should be `primes.c` or `primes.sr` and `fastprimes.c` or `fastprimes.sr`.