CSc 422/522 — Homework 2

due Tuesday, February 23

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. Consider the following atomic instruction:

```
procedure Swap(var x, y: int)
  ⟨ temp := x;  x := y;  y := temp ⟩
```

This, as one machine instruction, swaps the values of `x` and `y`.

(a) Using `Swap`, develop a solution to the critical section problem for `n` processes. Do not worry about the eventual entry (fairness) property. Describe clearly how your solution works and why it is correct.

(b) Modify your answer to (a) so that it will perform well on a multiprocessor system with caches. Explain what changes you make (if any) and why.

2. Using the `Swap` instruction, develop a fair solution to the critical section problem. The key is to order the processes, where FCFS (first come, first served) is the most obvious ordering. Note that you cannot just use your answer to the first problem to implement the atomic action in the ticket algorithm, because you cannot get a fair solution using unfair components. Hint: You may assume that each process has a unique identity, say the integers from 1 to `n`.

Be sure to explain what you are doing, and to give convincing arguments for why your solution is correct and why it is fair.

3. Consider a reusable dissemination barrier for 14 processes.

(a) Give complete details for the code executed by each process. In particular, write a procedure `barrier(id)` that is called by each process `id`. You may use `await` to specify delay points. Show the declaration and initialization of all the flag variables you need; these should be global to the procedure.

(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? Do not count procedure call overhead; just count the code in the body of the procedure.

4. The purposes of this problem are to introduce you to the `pthreads` library and to let you see the effects (I hope) of not protecting a critical section. The `pthreads` library is described in Section 4.6 of the text. You might also want to look at the manual pages for `pthreads`, `mutex`, and `semaphore`. The program `clock.c` handed out in class shows how you can measure execution times.

(a) Write a simple `pthreads` program that has two shared variables, `x` and `y`, that are initially zero. Have 4 identical processes increment `x` then increment `y` 1000 times. Do not protect the variables. At the end of the program, write out the final values of the variables and the execution time. Repeat this test five times. Run your tests on `par`.

(b) Modify your program so that the increments of `x` and `y` are *separate* atomic actions. Again run your tests five times and write the final values and execution time for each test.

(c) Modify your program so that both increments are in the *same* atomic action. Again run your tests five times and write the final values and execution times.

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.

5. 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. 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 for any delays (simple `await` statements).

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.

Write a program that implements the second approach. At the end of your program, print the execution time for the computational part, then print the last 10 primes you found.

Hand in a commented listing of your program together with the output from at least six tests in which `W` is 1, 2, or 4, and `L` is 100 or 10000 (ten thousand). Feel free to try even larger values of `L`, such as one million or even one billion. Run your tests on `par`. Please also use `turnin` to provide us with a copy of your program.