CSc 422/522 — Homework 3

due Tuesday, April 6

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

Be sure to read each question carefully and to do all that is asked. As usual, the work you turn in must be your own. Please explain your answers and programs clearly.

1. **The Unisex Bathroom Problem.** Suppose there is only one bathroom in an office. It can be used by both men and women, but not at the same time.

(a) This problem is an example of selective mutual exclusion, as described in Section 4.4 of the text. Develop a solution to the problem using semaphores. Your solution should have a style similar to the readers/writers solution in Figure 4.10.

(b) This is also an example of a condition synchronization problem. Develop a second solution to the problem using semaphores. *First develop a solution that is similar to the program in Figure 4.11; start by stating a global invariant.* Then convert your coarse-grain solution into a fine-grain one that uses the technique of passing the baton. Your final program should be similar to the program in Figure 4.13.

(c) Modify your answer to (b) so that it is fair. You do not have to make your solution FIFO in order for it to be fair; all that is required is that neither men nor women have to wait forever to use the bathroom. [You might also find it instructive to consider how you might modify your answer to (a) to make it fair, but that is not required.]

2. **The Dining Philosophers Using Monitors.** The dining philosophers problem is introduced in Section 4.3 of the text.

(a) Develop a monitor to synchronize the actions of the philosophers. The monitor should have two operations: `getforks(id)`, and `relforks(id)`, where $1 \le$ `id` ≤ 5. Declare the variables you need *and* specify a monitor invariant. Then develop the bodies of the procedures. Assume that `signal` follows the Signal and Continue discipline. Your answer to part (a) must ensure that neighboring philosophers do not eat at the same time and it must avoid deadlock, but it need not ensure fairness.

(b) Explain whether your answer to (a) is correct if `signal` follows the Signal and Wait discipline. If so, why? If not, why not?

(c) Modify your answer to (a) to ensure fairness, namely that a philosopher that calls `getforks` eventually gets to eat (assuming, as usual, that no philosopher dies while eating :-) Again use the Signal and Continue discipline. [If your answer to (a) is fair, explain why.]

3. **The Roller Coaster Problem.** Suppose there are `n` passengers and one roller coaster car. The passengers repeatedly wait to take rides on the car. The car has a capacity of `S` seats, `S < n`, but the cheapskate owners won't let the car leave the station until it is full.

Write a program to simulate a solution to this problem. Each passenger should be a process that waits to take a ride, then naps for a random amount of time. The car should be a process that waits until `S` passengers want rides, then goes around the track for some fixed amount of time, then lets the passengers depart.

The processes are to synchronize by means of a "monitor" that has three operations: `takeRide`, `loadCustomers`, and `unloadCustomers`. The first operation is called by passengers; a call does not return until the passenger has taken a ride. The last two operations are called by the car process. [This is similar to the sleeping barber problem in Section 5.2.]

See Section 1.5 of the SR book for an example of a simulation that is somewhat similar to this, but do not use a process and the `in` statement to implement the monitor; we will cover that topic later in Chapter 8.

You may write your program either in SR or in C plus `pthreads`. If you use SR, then use a global component to implement the monitor. The interface (spec) of the `global` should declare the above three operations; the body of the `global` should implement them. *Use semaphores to implement mutual exclusion,* `wait` *and* `signal`. See Section 6.5 of the text for how to implement monitors using semaphores.

If you use C plus `pthreads`, you will not be able to have a clean implementation, but you could put the monitor in a separate file. See Section 5.5 of the text for how to simulate monitors using `pthreads`.

Your program should have three command-line arguments:

> `n`, the number of passenger processes,
>
> `C`, the capacity of the car, and
>
> `T`, the total number of times the car runs around the roller coaster.

Stop the simulation after the car has completed `T` runs. (There may be waiting passengers at this point.)

The output of your program should be a trace of the calls of the procedures in the "monitor". Each line of output should contain a "time stamp", a short message indicating which procedure was called, and—in the case of calls of `takeRide`—which passenger process made the call. [If you really want to get fancy and to have some fun—at the price of doing more work—check out the SRWin package, which provides a nice clean interface between SR and X windows.]

Turn in a commented listing of your program and sample output from at least the following three executions of the program:

```
a.out 1 1 10
a.out 3 1 10
a.out 5 3 10
```

The arguments are the number of passengers `n`, the capacity of the car `C`, and the total number of rides `T`, in that order.

4. **The Savings Account Problem.** A savings account is shared by several people (processes). Each person may deposit or withdraw funds from the account. The current balance in the account is the sum of all deposits to date minus the sum of all withdrawals to date. The balance must never become negative.

Write a program to simulate a solution to this problem. Each customer should be a process that repeatedly does a deposit or withdrawal, the goes out to lunch for a random amount of time. The bank is to be represented by a monitor that has two operations: `deposit(amount)` and `withdraw(amount)`. You may assume that `amount` is positive. You may also use two kinds of customer processes if you wish, one kind for deposits and one for withdrawals.

A customer calls `deposit` to add `amount` to the account balance. A customer calls `withdraw` to extract `amount` from the account. Unlike in real life, however, a customer that calls `withdraw` waits until the balance is sufficient (instead of tracking down one of the people who has some money). Note that if two customers are waiting to withdraw funds and then another customer deposits money, either zero, one, or both of the waiting customers might be able to withdraw funds (one at a time).

You may write your program either in SR or in Java. If you use SR, then you are also to use the `m2sr` package to implement the monitor. That package supports the monitor constructs described in the text; it implements them using semaphores, but if you use SR for this problem, you must use the package rather than the approach used in problem 3. See the manual page for a description of the `m2sr` package, and see `/home/sr/examples/m2sr` for sample programs.

If you use Java, see Section 5.4 of the text for descriptions of Java's threads and synchronized methods. That section also contains a series of readers/writers programs. [I will put that code in `/home/cs522/SamplePrograms` so you can copy and play with it if you wish.]

It is up to you to decide what command-line arguments to employ and what tests to run. (See problem 3 for a few ideas.) Think about interesting ways to show off how your program works. Be sure to test the three cases described above for what can happen when `deposit` is called and two customers are waiting.