Name: ______________________________

CSc 422, Spring 2002 — Examination 2

You may use up to *two* pages of notes for this exam, but otherwise it is closed book. Please put your name on the top of your notes and turn them in with your examination. Do your work on these sheets, using additional sheets if necessary.

This exam is worth 60 points. The first three problems are worth 8 points each; the last three problems are worth 12 points each. *You must show your work and/or explain your answers.* This is required for full credit and is helpful for partial credit.

1. [8 points] With asynchronous message passing, `send` is nonblocking and `receive` is blocking. With *synchronous message passing*, both statements are blocking.

Develop an implementation of synchronous message passing that uses a single shared variable for the buffer and three semaphores for synchronization. These variables are declared as shown below. For simplicity, there is just one channel, so it does not need to be named.

```
int buffer;
sem empty = 1, full = 0, done = 0;

send(value int msg):   # synchronous send of value msg
```

```
receive(result int msg):  # receive msg as a result parameter
```

2. [8 points] Remote operations can be implemented using RPC (RMI) or rendezvous. They are commonly used to program client/server interactions in distributed systems.

(a) Define the syntax and semantics of the primitives that are used with RPC.

(b) Define the syntax and semantics of the primitives that are used with rendezvous.

(c) Assume that you are using only RPC to program interactions between modules in a distributed system. How can deadlock occur? How do you avoid it?

(d) Assume that you are using only rendezvous to program interactions between modules in a distributed program How can deadlock occur? How can you avoid it?

3. [8 points] Consider the following monitor, which is proposed as a solution to the shortest-job-next (SJN) allocation problem. Client processes call `request` and then `release`. The resource can be used by at most one client at a time. When there are two or more competing requests, the one with the minimum value for argument `time` is to be serviced next.

```
monitor SJN {
  bool free = true;
  cond turn;

  procedure request(int time) {
    if (not free)
      wait(turn, time);
    free = false;
  }

  procedure release() {
    free = true;
    signal(turn);
  }
}
```

(a) Define the Signal-and-Continue (SC) signaling discipline.

(b) Define the Signal-and-Wait (SW) signaling discipline.

(c) Does the above monitor work correctly for the SC discipline? Clearly but briefly explain why or why not.

(d) Does the above monitor work correctly for the SW discipline? Clearly but briefly explain why or why not.

4. [12 points] Consider the following problem, which I will call the *Hungry Birds Problem.* Given are `n` baby birds and two parent birds. The birds shared a common dish, which can contain at most `B` bugs. The dish is initially empty.

Each parent bird flies off, finds one bug, flies back to the nest, waits until there is room in the dish, and puts the bug in the dish—then repeats these actions. Each baby bird chirps for a while, wakes up, waits for the dish to contain a bug, takes one, and eats it—then repeats these actions.

Develop a **monitor** to synchronize the actions of the birds. Assume that each bird is represented by a process. The dish is a critical variable that can be accessed by at most one bird at a time. The monitor should have two operations: `depositBug()`, which is called by the parent birds, and `fetchBug()`, which is called by the baby birds.

5. [12 points] Recall that in the roller coaster problem there are `n` passenger processes and one car process. The car has a capacity of `C` passengers, where `C < n`. In Homework 3 you developed a monitor to synchronize the actions of the passengers and the cars.

This problem is easier to solve if you use message passing, because the passengers can interact directly with the car. You are given the following channel declarations:

```
chan takeRide(int passengerID), rideOver[1:n]();
```

Every time passenger `i` wants to take a ride, it executes:

```
send takeRide(i);
receive rideOver[i]();
```

(a) Develop an implementation of the car process. You may use high-level pseudo code for sequential parts, but show exactly how the above channels are used.

(b) Suppose we add a second coaster car, which also has a capacity of `C` passengers. Car 1 loads first, then Car 2, then Car 1, and so on. Describe how you would change your answer to (a) to add a second car. Assume that the `takeRide` channel is shared, so both cars can receive from it. You may use one or more additional channels, but you may *not* change the passenger interface.

6. [12 points] Consider a distributed program with `n` processes, numbered 1 to `n`. Each process has a local value `v`. Process 1 wishes to compute the sum of all `n` values. However, process 1 has only a few neighbors; namely, process 1 is able to communicate with only a subset of the other processes. Similarly, each process has only a few neighbors with which it can communicate. In short, the interconnection network forms an undirected graph, but it is not a complete graph.

Assume that each process `i` has a vector, `neighbors[1:n]` that is initialized so that `neighbors[j] == 1` if `j` is a neighbor of `i` and `neighbors[j] == 0` otherwise. The processes communicate using the array of channels declared below. I have declared these as full arrays to simplify your programming, but process `i` can send only to its neighbors.

The code for process 1 is given below. Develop code for the other processes. You may use high-level pseudo code for sequential parts, but show exactly how the channels are used. Each process should execute the same program, but of course the values of `v` and `neighbors` are different in each process. [*Hint*: Consider how you could solve this problem for a tree, and then generalize your approach to handle a graph with cycles.]

```
chan Chan[1:n](int kind, int value);
  # kind is one of ASK or ANSWER and value is an additional field
  # depending on the kind of message

process Node1 {
  int v, neighbors[1:n]; initialize v and neighbors;
  int sum = v, numNeighbors = 0, kind, newSum;
  for [j = 2 to n st neighbors[j] == 1] {
    send Chan[j](ASK, 1);         # ask neighbor j for what it knows
    numNeighbors++;
  }
  for [j = 1 to numNeighbors] {  # collect answers in any order
    receive Chan[1](kind, newSum);    # kind should be ANSWER
    sum += newSum;                    # newSum is a partial sum
  }
}

process Node[i = 2 to n] {
```