Name: ______________________________

CSc 422/522 — Examination 2

You may use up to four pages of notes for this exam, but otherwise it is closed book. There are six questions; the first three are worth 10 points each, the last three are worth 15 points each. Graduate students are to answer all questions (75 points). Undergraduates are to answer five or six questions; I will count your five best scores, up to a maximum of 60 points.

You must explain your answer or show how you arrived at it. This is required for full credit and is helpful for partial credit. Do your work on these sheets, using additional sheets if necessary.

(1) [10 points] Suppose you are writing programs for a machine that has atomic increment and decrement instructions. `INC(x)` atomically adds 1 to integer variable `x`, and `DEC(x)` atomically subtracts 1 from integer variable `x`.

The semaphore operations are defined as:

```
P(s):  〈 await (s > 0) s = s-1; 〉
V(s):  〈 s = s+1; 〉
```

The `V(s)` operation can easily be implemented as `INC(s)`. *Develop a busy-waiting (spinning) implementation of* `P(s)`.

(2) [10 points] With asynchronous message passing, the `send` statement is non-blocking and the `receive` statement is blocking. With *synchronous* message passing, a message is sent using the `synch_send` statement. Both `synch_send` and `receive` are blocking statements. That is to say, when two processes wish to communicate, whichever one executes `synch_send` or `receive` first waits for the other to execute the other statement; then the message is transfered and both processes continue.

Using semaphores for mutual exclusion and signaling, develop an implementation of `synch_send(msg)` and `receive(msg)`, where `msg` is a message. Assume there is just one channel, so it is not named in these statements. Also assume there is a single buffer `buf` that is used to copy a message from the sender to the receiver. [Hint: We have seen how to use semaphores to implement `deposit` and `fetch` for a single slot buffer; `synch_send` is like a `deposit` and `receive` is like `fetch`, except that neither `synch_send` nor `receive` can finish until the other one has been called.]

(3) [10 points] Consider programming a self-scheduling disk driver using the rendezvous primitives (`call` and `in`). There are several clients. A client uses the disk by executing:

`call access(cylinder,` other arguments`);`

(a) Give a code outline for the disk driver process. It should implement a FIFO scheduling policy.

(b) Modify your answer to (a) so that the disk driver uses the shortest-seek-time (SST) scheduling policy. In particular, if there is more than one pending invocation of `access`, the disk driver should service one that wants to access the cylinder closest to the current head position.

(4) [15 points] Develop a monitor that solves the following problem, which I will call the *hungry birds problem*.

Given are `n` baby birds and one parent bird. Each bird is represented by a process. The baby birds eat out of a common dish that initially contains `W` worms. Each baby repeatedly sleeps for a while, wakes up and eats one worm, then goes back to sleep. If the dish is empty, the baby bird who finds the dish empty awakens the parent bird. The parent then flies off, finds `W` more worms, puts them all into the dish, and goes back to sleep.

(a) Develop a monitor to synchronize the actions of the birds. Define the monitor's operations, show how they are used by the baby and parent birds, and show how they are implemented by the monitor. Be sure to declare and initialize all the variables you need. *Also state which monitor signaling discipline you are using.*

(b) Is your solution fair? Explain.

(5) [15 points] Given are `n` processes in a distributed system. Each process executes on a processor that is connected to just a few of the other processors. In particular, the processors form an undirected graph, but it is not a complete graph. A process on one processor can communicate directly only with the processes on neighboring processors.

Local to each process `i` is a symmetric matrix `topology`. In process `i`, the initial values of `topology[i,j]` and `topology[j,i]` are true if and only if process `j` is a neighbor of process `i`. The initial values of `topology` will be different in different processes.

The problem is for *every* process to determine the topology (structure) of the entire graph. When the processes terminates, the value of `topology[p,q]` *in every process* should be true if and only if processes `p` and `q` are neighbors in the graph.

(a) Develop an outline of the code that the processes execute to solve this problem. They are to communicate using asynchronous message passing. Show all communications, but you may use words to describe other actions the processes take. Be sure to declare the channels that you need.

(b) How can the processes know when to terminate? Explain, for your solution, how each process decides when it has the answer.

(6) [15 points] We can use a pipeline to sort `n` values as follows. (This is not a very efficient sorting algorithm, but what the heck, this is an exam.)

Given are `W` worker processes and a coordinator process. Assume that `n` is a multiple of `W` and that each worker process can store at most `n/W + 1` values at a time. The processes form a closed pipeline. Initially, the coordinator has `n` unsorted values. It sends these one at a time to worker 1. Worker 1 keeps some of the values and sends others on to worker 2. Worker 2 keeps some of the values and sends others on to worker 3, and so on. Eventually, the workers send values back to the coordinator. (They can do this directly.)

(a) Develop code for the coordinator and the workers so that the coordinator gets back *sorted* values. Each process can insert a value into a list or remove a value from a list, but it may not use an internal sorting routine.

(b) How many messages are used by your algorithm? Give your answer as a function of `n` and `W`. Be sure to show how you arrived at the final answer.