

Name: _____

CSc 422, Spring 2009 — Examination 2

You may use up to *two* pages of notes for this exam, but otherwise it is closed book. Write your answers on these sheets, using additional sheets if necessary.

The exam is worth 60 points. There are 5 problems; each is worth 12 points. *You must show your work and/or explain your answers.* This is required for full credit and is helpful for partial credit.

1. 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. Consider the following problem, which I call the *Hungry Birds*. 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.

3. Consider a memory allocator that manages n equal-size blocks of memory. A client requests free blocks by calling `request(amount)`; a request delays until there are at least `amount` free blocks of memory. You may assume that `amount` is between 1 and n . A client releases memory blocks by calling `release(amount)`, where `amount` is the number of blocks being released. Blocks may be released in different quantities than they are acquired. You may assume that a client only releases blocks that it had previously acquired.

Write a **monitor** to implement the memory allocator. Do not worry about representing the blocks or identifying them. Focus just on properly synchronizing requests and releases. Assume that all n blocks are initially free. Your solution does not have to be fair.

4. Assume that a class has n students numbered 1 to n . The teacher wants students to form groups of two for a project, and asks the students to develop a single algorithm that every student executes. The algorithm should use random numbers so that the outcome is nondeterministic. The teacher starts the algorithm by picking one student at random and sending a message to that student saying in effect "your turn to pick a partner."

Use **asynchronous message passing** for process interaction; do *not* use shared variables. When the code terminates, a local variable `partner` in each student should indicate the student's partner. An ideal solution will use only n messages. You may use pseudo-code for sequential parts, and you may use high-level data structures such as sets or lists.

The students share the channel declared below; the teacher starts the selection process by sending a message as shown. The set `left` indicates which students do not yet have partners.

```
chan choose[n] (int partner, set left);
```

```
Teacher:: first = random number between 1 and n;  
          send choose[first] (0, left = {1,2,...,n});
```

5. Develop a peer-to-peer algorithm for solving the mutual exclusion problem in a distributed system. Assume that each of n Client processes has a Helper process. The Helpers form a ring. A token starts at Helper[1] and continuously circulates around the ring of Helpers from 1 to 2 to 3 ... to n and then back to 1.

When a client wants to have exclusive access to some resource, it sends a message to its Helper. That Helper waits to get the token, then tells its Client to proceed; when the Client leaves its critical section, it informs the Helper, who puts the token back into circulation. Thus the token keeps circulating except when a Helper's Client actually wishes to enter the critical section.

Develop code for the Helper processes; each should execute the same code. Also show how the a Client interacts with its Helper. Use **asynchronous message passing** for process interaction. Be sure to declare all channels that you need.

channel declarations:

Client[i = 1 to n]::

Helper[i = 1 to n]::