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 five questions; each is worth 15 points. Graduate and honors students are to answer all questions. Undergraduates are to answer any four—or answer all five and I will count only your four best scores. *For full credit, you must explain your answer or show how you arrived at it.*

1. Suppose we want to program rendezvous synchronization between pairs of processes. In particular, the first process to arrive at its rendezvous point waits for the second process, the third waits for the fourth, and so on. Consider the following code, which is executed by each process. It uses semaphores for synchronization:

```
sem e = 1, go = 0;          # shared variables
int count = 0;

P(e);                       # code executed by each process
count++;
if (count == 1)
  { V(e); P(go); }
else
  { count = 0; V(go); V(e); }
```

(a) What is wrong with the above "solution." Explain.

(b) Fix the code so that it is correct. Write your new code below, or modify the above code.

2. Suppose there are `m` producer processes and `n` consumer processes. The producer processes periodically call `broadcast(m)` to send a copy of message `m` to *all* consumers. Each consumer receives a copy of the message by calling `fetch(m)`, where `m` is a result argument.

Write a *monitor* that implements `broadcast` and `fetch`. Use the Signal and Continue discipline. The monitor should store only one message at a time, which means that after one producer calls `broadcast`, any future call of `broadcast` has to delay until *every* consumer has received a copy of the first message. Assume that messages are single integers.

3. Suppose a message passing library provides a `broadcast` primitive in addition to `send`, `receive`, and `empty`. In particular, if `C[1:n]` is an array of channels, then execution of

```
broadcast C(m)
```

puts a copy of `m` on *every* channel `C[i]`. Assume that execution of `broadcast` is *atomic*: It puts a copy of the message on all channels as a single atomic action. This means that every process will see broadcast messages in the same order.

Using `broadcast` and the other message passing primitives, develop a solution to the *distributed mutual exclusion problem.* This is the familiar mutual exclusion problem, but in a distributed system—hence, we are protecting access to something external that is shared, such as a file, rather than protecting access to shared variables.

Assume there are `n` processes. Your task is to develop critical section entry and exit protocols that the processes execute. They can only communicate with each other using message passing. Be sure to declare the channels that you employ.

Channels:

Entry protocol:

Exit protocol:

4. A computer center has two printers, `A` and `B`. Clients that want to use a printer execute

```
which = request(kind)
```

The value of `kind` is `A`, `B`, or `EITHER`. When `request` returns, the value of `which` is the identity of the printer that was allocated. If a client requests printer `A` (or `B`) then it must be allocated that printer, and `which` will be the same as `kind`. If a client requests `EITHER` printer, then it can be allocated `A` or `B`, and `which` will indicate which printer was allocated.

A client releases a printer by executing

```
release(which)
```

Write a server process that implements `request` and `release`. Use the `in` statement of Chapter 8 (rendezvous) to implement the server. (If you cannot solve the problem using the `in` statement, you may use message passing for partial credit.)

5. Suppose we have a shared channel that is declared as:

```
chan bag(int)    # same as "op bag(int)" in SR
```

Assume that some process has initialized the `bag` by sending `n` values to it.

The problem is to use `M` workers to compute and print the sum of all the elements in `bag`. Each can compute only one sum at a time. (This is not a realistic problem, but what the heck, this is an exam!) Someone suggests the following program:

```
process Worker[i = 1 to M] {
  declarations of local variables;
  while (true) {
    receive two elements from the bag;
    compute their sum;
    if (empty(bag))
      { print the sum; break; }
    else
      send bag(sum);
  }
}
```

(a) This program will work some of the time but not always. Explain when it will work and when not.

(b) Explain how to fix the program so that it always computes the sum and prints the result once. It is OK if some workers are blocked when the program terminates. You can add fields to the messages stored in `bag`, add additional channels, and/or add a manager process. You do not need to develop detailed code. It is sufficient to give a good explanation and/or high-level pseudo-code.