First Name: _____ Last Name: _____

Sample Final Exam — CSc 422, Fall 2017

This is a **120** minute exam, and contains nine pages and eight questions worth **16** points each. Answer **six** of the eight questions. Either leave two questions blank, or mark two questions "Do Not Grade". It is a closed-book exam. Calculators and foreign-language-to-English dictionaries are allowed. Put your answers in the space provided here. Show your work.

4 points will be added to everyone's score to make the exam worth a total of **100** points.

You may use one sheet of paper, no larger than 8.5" x 11", with notes written in your own handwriting. The notes may be on both sides of the paper. Turn in the notes page with your test.

You must explain your answers or show how you arrived at them. This is required for full credit and is helpful for partial credit.

1. (16 points) — Consider the following program in which processes communicate using message passing. Assume non-blocking **send** and blocking **receive**.

For each of the three communication channels indicate

- the order in which processes send messages through the channel, and
- the value that is being sent.

```
chan toA(int), toB(int), toC(int);
```

```
process A {
  int v1 = 1, new;
  send toB(v1);
  v1 = v1 + 2;
  send toC(v1);
  v1 = v1 + 2;
  receive toA(new);
  v1 = max(v1,new+1);
  v1 = v1 + 2;
  send toC(v1);
  v1 = v1 + 2;
  receive toA(new);
  v1 = max(v1,new+1);
  v1 = v1 + 1;
}
```

```
process B {
  int v2 = 1, new;
  send toC(v2);
  v2 = v2 + 1;
  receive toB(new);
  v2 = max(v2,new+2);
  v2 = v2 + 1;
  receive toB(new);
  v2 = max(v2,new+2);
  v2 = v2 + 1;
  send toA(v2);
  v2 = v2 + 1;
}
```

```
process C {
  int v3 = 1, new;
  receive toC(new);
  v3 = max(v3,new+3);
  v3 = v3 + 2;
  receive toC(new);
  v3 = max(v3,new+3);
  v3 = v3 + 2;
  send toA(v3);
  v3 = v3 + 1;
  send toB(v3);
  v3 = v3 + 1;
  receive toC(new);
  v3 = max(v3,new+3);
  v3 = v3 + 1;
}
```

2. (16 points) — Two processes, **`A`** and **`B`**, both have a local array of **`10`** integers.

- Both arrays are sorted in ascending order.
- There is at least one value common to both arrays.

Develop an algorithm in which the two processes interact until each has determined the smallest common value.

Use message passing to solve the problem.

Assume that each message contains at most one integer.

One way to solve this is to have both processes send all their numbers to the other. You may not use this solution! Your solution needs to use fewer messages.

3. (16 points) — Assume that a class has `N` students, numbered `1` to `N`, and one teacher. The teacher wants to assign the students to groups of two for a project. The teacher does this by having every student submit a request for a partner. The teacher takes the first two requests, forms a group from those students, and lets each student know the id of the partner. The teacher then takes the next two requests, forms a second group, and so on. If `N` is odd, the last student who contacts the teacher is solo (gets back her/his own id as partner).

Model the students as processes and the teacher as a server process or module. Develop code for the client (used by each student) and the server (used by the teacher).

Write an implementation using message passing.

4. (16 points) — Give all possible final values of variable **x** in the following program. Explain how you got each answer.

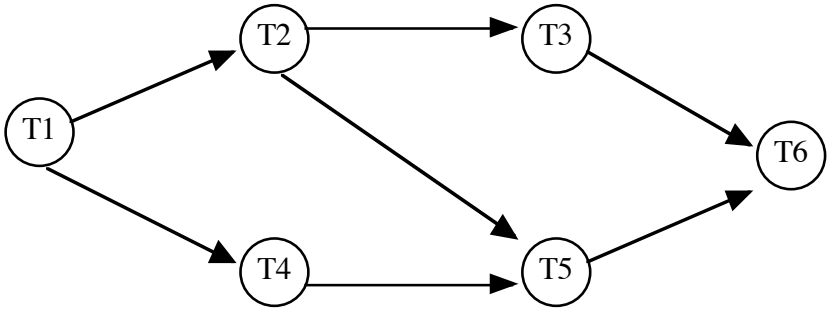
```
int x = 0;
sem s1 = 1;
sem s2 = 0;
co P(s2); P(s1); x = x * 2; V(s1);
// P(s1); x = x * x; V(s1);
// P(s1); x = x + 3; V(s2); V(s1);
oc
```

5. (16 points) — A precedence graph is a directed, acyclic graph. Nodes represent tasks, and edges indicate the order in which tasks are to be accomplished. In particular, a task can execute as soon as all its predecessors have been completed. Assume that the tasks are processes and that each process has the following outline:

```
process T {
   wait for predecessors, if any;
   body of the task;
   signal successors, if any;
}
```

Using semaphores, show the code for six processes whose permissible execution order is specified by the following precedence graph:

Do not impose constraints not specified in the graph. For example, **`T2`** and **`T4`** can execute concurrently once **`T1`** completes. For full credit, minimize the number of semaphores that you use.



6. (16 points) — Consider the following way to sort a set of **100** numbers:

There are **10** processes. Each initially has **10** integers. The processes are organized in a chain, so that each process can communicate only with its neighbors. For example, process **1** can communicate only with process **2**, process **2** can communicate only with processes **1** and **3**, process **3** can communicate only with processes **2** and **4**, etc. Process **10** can communicate only with process **9**.

Each channel between processes is bi-directional.

This is not a ring architecture since there is not a channel between processes **10** and **1**.

Design an algorithm that terminates with the numbers being sorted into ascending order. In particular, process **1** ends up with the smallest **10** numbers, process **2** ends up with the next smallest set of **10** numbers, and so on. Be sure to show how termination is detected.

Use message passing for inter-process communication. Show all uses of **send** and **receive**, but just state what the sequential code in each process does.

7. (16 points) — Complete the code below for a distributed semaphore server that uses **`send()`** and **`receive()`** primitives.

The server maintains an array of integers that are the current values for the semaphores and carries out **`P`** and **`V`** operations on those semaphores upon request from the clients.

Clients send requests to the server

- to create a new semaphore — providing an initial value for the semaphore;
- to perform a **`P`** operation; and
- to perform a **`V`** operation.

Assume non-blocking **`send()`** and blocking **`receive()`** operations.

Here is code for the proc's that a client will use to create a semaphore, to do a **`P`** operation on a specified semaphore, and to do a **`V`** operation on a specified semaphore. Clients, will use the proc's below for **`create_sem()`**, **`P()`**, and **`V()`** that contain the needed **`send()`** and **`receive()`** operations. These proc's use three different message tags (**`create`**, **`P`**, **`V`**) to indicate whether they are creating a semaphore or performing a **`P`** or **`V`** operation.

Client:

```
semaphore create_sem(init_value) {
   send(create, init_value, my_id);
   receive(sem);
   return sem;
}
P(sem) {
   send(P, sem, my_id);
   receive();
}
V(sem) {
   send(V, sem, my_id);
}
```

For the server, complete the implementation of these operations.

Server:

```
while (1) {
   receive(tag, value, client_id);
```

8. (16 points) —

a.) (8 points) Consider a four-process dissemination barrier. If the four processes are distributed across four machines, and can only communicate via **`send()`** and **`receive()`**, show the messages that each of the four processes will use in working through the dissemination barrier.

Assume **`send()`** is non-blocking and **`receive()`** is blocking.

Use process id's **`0`**, **`1`**, **`2`**, and **`3`** for the four processes.

Assume there is an array of four message channels:

```
channel ch[4];
```

b.) (8 points) Now, consider a more general case for a dissemination barrier for eight processes (numbered **`0`** through **`7`**). Write a (single) procedure that could be used by each of the processes to work through the dissemination barrier in this case. The procedure should use a loop to move the process through the barrier.

```
channel ch[8];
proc Barrier( int my_id ) {
```

Solutions to Sample Final Exam — CSc 422, Fall 2017

1. (16 points) — Consider the following program in which processes communicate using message passing. Assume non-blocking **send** and blocking **receive**.

For each of the three communication channels indicate

- the order in which processes send messages through the channel, and
- the value that is being sent.

```
chan toA(int), toB(int), toC(int);
```

```
process A {
  int v1 = 1, new;
  send toB(v1);
  v1 = v1 + 2;
  send toC(v1);
  v1 = v1 + 2;
  receive toA(new);
  v1 = max(v1,new+1);
  v1 = v1 + 2;
  send toC(v1);
  v1 = v1 + 2;
  receive toA(new);
  v1 = max(v1,new+1);
  v1 = v1 + 1;
}
```

```
process B {
  int v2 = 1, new;
  send toC(v2);
  v2 = v2 + 1;
  receive toB(new);
  v2 = max(v2,new+2);
  v2 = v2 + 1;
  receive toB(new);
  v2 = max(v2,new+2);
  v2 = v2 + 1;
  send toA(v2);
  v2 = v2 + 1;
}
```

```
process C {
  int v3 = 1, new;
  receive toC(new);
  v3 = max(v3,new+3);
  v3 = v3 + 2;
  receive toC(new);
  v3 = max(v3,new+3);
  v3 = v3 + 2;
  send toA(v3);
  v3 = v3 + 1;
  send toB(v3);
  v3 = v3 + 1;
  receive toC(new);
  v3 = max(v3,new+3);
  v3 = v3 + 1;
}
```

Solution:

channel **A**: **C** sends first sending a **10** or an **8**, then **B** sends:

the **10** if **C** receives **3** then **1**

OR

the **8** if **C** receives **1** then **3**

channel **B**: **A** sends first, sending a **1**; later **C** sends

channel **C**: **A** sends **3** and **B** sends **1**, can be in opposite order

A sends 3rd value which is either **9** or **11**

2. (16 points) — Two processes, **A** and **B**, both have a local array of **10** integers.

- Both arrays are sorted in ascending order.
- There is at least one value common to both arrays.

Develop an algorithm in which the two processes interact until each has determined the smallest common value.

Use message passing to solve the problem.

Assume that each message contains at most one integer.

One way to solve this is to have both processes send all their numbers to the other. You may not use this solution! Your solution needs to use fewer messages.

Solution:

```
Solution from Tanner Bernth's Spring 2017 Final Exam:

process FindLeastCommon {
   int values[10];
   int smallest = values[0];
   int index = 0;
   int other;

   send(smallest); // send initial value
   receive(other);

   while ( smallest != other ) {
      if ( smallest > other ) {
         // receive other smallest for comparisons
         // new other should be bigger or equal
         receive( other );
      }
      else {
         index++;

         // increase index in values until it is greater than
         // or equal to the other process's smallest
         while ( values[index] < other ) {
            index++;
         }

         smallest = values[ index ];
         if ( smallest != other ) {
            // send larger smallest value
            // otherwise, they are equal
            send( smallest );
      }
```

3. (16 points) — Assume that a class has **N** students, numbered **1** to **N**, and one teacher. The teacher wants to assign the students to groups of two for a project. The teacher does this by having every student submit a request for a partner. The teacher takes the first two requests, forms a group from those students, and lets each student know the id of the partner. The teacher then takes the next two requests, forms a second group, and so on. If **N** is odd, the last student who contacts the teacher is solo (gets back her/his own id as partner).

Model the students as processes and the teacher as a server process or module. Develop code for the client (used by each student) and the server (used by the teacher).

Write an implementation using message passing.

Solution:
Teacher Server
Student:

```
send partner(myID);
receive reply[myID](otherID);
```

Teacher:

```
for [i = 1 to n/2 ] {
   receive partner(id1);
   receive partner(id2);
   send reply[id1](id2);
   send reply[id2](id1);
}
if ( n is odd ) {
   receive partner(id1);
   send reply[id1](id1);
```

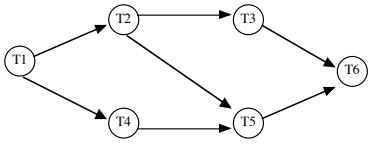
4. (16 points) — Give all possible final values of variable **x** in the following program. Explain how you got each answer.

```
int x = 0;
sem s1 = 1;
sem s2 = 0;
co P(s2); P(s1); x = x * 2; V(s1);
// P(s1); x = x * x; V(s1);
// P(s1); x = x + 3; V(s2); V(s1);
oc
```

Solution:
The possible values of **x** are **6**, **18**, and **36**.
Note that each assignment statement (and corresponding loads, stores and arithmetic) cannot be interrupted due to the **s1** semaphore. There is an additional restriction that the first arm's assignment statement must be executed after the third arm has completed. Thus, the following are possible (where the number refers to the assignment statement of the corresponding **co** arm):

```
2, 3, 1 -> x = 6
3, 2, 1 -> x = 18
3, 1, 2 -> x = 36
```

5. (16 points) — A precedence graph is a directed, acyclic graph. Nodes represent tasks, and edges indicate the order in which tasks are to be accomplished. In particular, a task can execute as soon as all its predecessors have been completed. Assume that the tasks are processes and that each process has the following outline:




```
process T {
   wait for predecessors, if any;
   body of the task;
   signal successors, if any;
}
```

Using semaphores, show the code for six processes whose permissible execution order is specified by the following precedence graph:

Do not impose constraints not specified in the graph. For example, **T2** and **T4** can execute concurrently once **T1** completes. For full credit, minimize the number of semaphores that you use.

Solution:

```
semaphores: one = 0, two = 0, three = 0;

T1:                              T4:
   body of T1                       P(one);
   V(one);                          body of T4
   V(one);                          V(four);
T2:                              T5:
   P(one);                          P(two);
   body of T2                       P(four);
   V(two);                          body of T5
   V(two);                          V(three);
T3:                              T6:
   P(two);                          P(three);
   body of T3                       P(three);
   V(three);                        body of T6
```

6. (16 points) — Consider the following way to sort a set of **100** numbers:

There are **10** processes. Each initially has **10** integers already sorted in ascending order. The processes are organized in a chain, so that each process can communicate only with its neighbors. For example, process **1** can communicate only with process **2**, process **2** can communicate only with processes **1** and **3**, process **3** can communicate only with processes **2** and **4**, etc. Process **10** can communicate only with process **9**.

Each channel between processes is bi-directional.

This is not a ring architecture since there is not a channel between processes **10** and **1**.

Design an algorithm that terminates with the numbers being sorted into ascending order. In particular, process **1** ends up with the smallest **10** numbers, process **2** ends up with the next smallest set of **10** numbers, and so on. Be sure to show how termination is detected.

Use message passing for inter-process communication. Show all uses of **send** and **receive**, but just state what the sequential code in each process does.

Solution:

There are several possibilities for the algorithm here. Possibly the simplest is to send all the numbers to one end, have the process on that end sort all of them (can be done efficiently by merging the numbers into an array as they arrive), then send the numbers back with each node keeping its **10** and sending the remainder on.

Another possibility is to send the largest number of the **10** to the right and the smallest to the left, receiving the corresponding value in return from each direction. Insert the two values into the list of numbers. Repeat until done. The problem: when is it done?

Worst case: the largest **10** numbers start in **#1** and the smallest in **#10**). It will take **9** iterations to get the first value to the other end, with the other **9** numbers trailing along behind (pipeline fashion). **8** more iterations to get them pumped along. Can count iterations to determine when done.

Be wary of halting when no change occurs, as it is possible to have a chunk of identical numbers in the original **100**. These can make changes stop, but not permanently.

7. (16 points) — Complete the code below for a distributed semaphore server that uses **`send()`** and **`receive()`** primitives.

The server maintains an array of integers that are the current values for the semaphores and carries out **`P`** and **`V`** operations on those semaphores upon request from the clients.

Clients send requests to the server

- to create a new semaphore — providing an initial value for the semaphore;
- to perform a **`P`** operation; and
- to perform a **`V`** operation.

Assume non-blocking **`send()`** and blocking **`receive()`** operations.

Here is code for the proc's that a client will use to create a semaphore, to do a **`P`** operation on a specified semaphore, and to do a **`V`** operation on a specified semaphore. Clients, will use the proc's below for **`create_sem()`**, **`P()`**, and **`V()`** that contain the needed **`send()`** and **`receive()`** operations. These proc's use three different message tags (**`create`**, **`P`**, **`V`**) to indicate whether they are creating a semaphore or performing a **`P`** or **`V`** operation.

Client:

```
semaphore create_sem(init_value) {
   send(create, init_value, my_id);
   receive(sem);
   return sem;
}
P(sem) {
   send(P, sem, my_id);
   receive();
}
V(sem) {
   send(V, sem, my_id);
}
```

For the server, complete the implementation of these operations.

Server:

```
while (1) {
   receive(tag, value, client_id);
```

Solution:
Server:

```
while (1) {
   receive(tag, value, client_id);
   if (tag == P) {
      if (sem_array[value].count == 0)
         insert client_id on sem_array[value].queue;
      else {
         sem_array[value].count--;
         send[client_id]();
      }
   else if (tag == V) {
      if ( sem_array[value].count == 0 and
           !empty( sem_array[value].queue ) {
         id = remove fm sem_array[value].queue
         send[id]();
      }
   else if (tag == create) {
      allocate a semaphore structure from the sem_array;
      sem_array[this_sem].count = value;
      sem_array[this_sem].queue = empty;
      send[client_id](this_sem);
   }
}
```

8. (16 points) —

a.) (8 points) Consider a four-process dissemination barrier. If the four processes are distributed across four machines, and can only communicate via **`send()`** and **`receive()`**, show the messages that each of the four processes will use in working through the dissemination barrier.

Assume **`send()`** is non-blocking and **`receive()`** is blocking.
Use process id's **`0`**, **`1`**, **`2`**, and **`3`** for the four processes.
Assume there is an array of four message channels:

```
channel ch[4];
```

Solution:

```
channel ch[4];
process 0:                  process 1:
  send ch[1]();               send ch[2]();
  receive ch[3]();            receive ch[0]();
  send ch[2]();               send ch[3]();
  receive ch[2]();            receive ch[3]();

process 2:                  process 3:
  send ch[3]();               send ch[0]();
  receive ch[1]();            receive ch[2]();
  send ch[0]();               send ch[1]();
  receive ch[0]();            receive ch[1]();
```

b.) (8 points) Now, consider a more general case for a dissemination barrier for eight processes (numbered **`0`** through **`7`**). Write a (single) procedure that could be used by each of the processes to work through the dissemination barrier in this case. The procedure should use a loop to move the process through the barrier.

```
channel ch[8];
proc Barrier( int my_id ) {
```

Solution:

```
channel ch[8];

proc Barrier( int my_id ) {
   int stage;

   for (int i = 0 to 2) {
      stage = 2^i;   # 2 raised to the power of i
      send[ (my_id + stage) mod n ];
      receive[ (my_+id - stage + n) mod n ]();
   }
}
```