First Name: _____ Last Name: _____

Sample Test #2 — CSc 422, Fall 2017

This is a **50** minute exam, and contains five pages and four questions worth **20** points each.

Answer **THREE** of the four questions.

Either leave one question blank, or mark one question Do Not Grade. Answering all four questions without marking one as Do Not Grade will cost you four points!

This 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.

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 pages 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. (20 points) — Develop a solution for a monitor that has one operation:

```
int exchange(int value)
```

The procedure is called by two processes to exchange the values of their actual parameters. The monitor should be reusable in the sense that it should exchange the parameters of the first pair of callers, then the next pair of callers, and so on.

For example, if processes **A**, **B**, **C**, **D**, and **E** call **exchange** in the order listed, then **A** and **B** will exchange values, **C** and **D** will exchange values, and **E** will be blocked waiting for another process to call **exchange**.

You may use either Signal and Continue (**SC**) or Signal and Wait (**SW**). *State which one* you are using.

2. (20 points) — Consider the following proposed monitor solution to the shortest-job-next allocation problem:

```
monitor SJN {
   bool free = true;
   cond turn;
   procedure request(int time) {
      if ( ! free )
         wait(turn, time);
      free = false;
   } # request

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

The **`wait(turn, time)`** function will block the caller on the **`turn`** condition variable. Processes are blocked on **`turn`** in order of their request time in shortest to longest order.

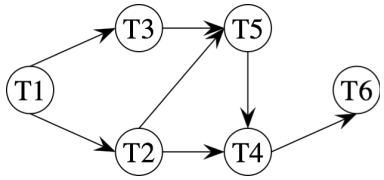
For each of a.) and b.), provide a justification for your answer. If you think the answer is "no", it is sufficient to provide an example. If you think the answer is "yes", provide an argument as to why the solution is correct.

a.) Does this solution work correctly for Signal and Continue (**`SC`**)?
b.) Does it work correctly for Signal and Wait (**`SW`**)?

3. (20 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, **`T3`** and **`T2`** can execute concurrently if their predecessors have finished.
For full credit, use the fewest number of semaphores that can still solve the problem correctly.

4. (20 points) — Suppose there is only one bathroom in a building. It can be used by both men and women, but not at the same time.

Develop a monitor to synchronize the use of the bathroom. The monitor has four public procedures: **`womanEnter`**, **`womanExit`**, **`manEnter`**, and **`manExit`**. A woman process calls **`womanEnter`** to get permission to use the bathroom and calls **`womanExit`** when finished. A man process calls **`manEnter`** and **`manExit`**. State the monitor invariant for your monitor solution.

Use Signal-and-Continue (**`SC`**) semantics. Your solution does not have to be fair.

It is possible for more than one person to be in the bathroom at the same time. Assume the bathroom has infinite capacity (obviously this is unrealistic, but makes for a reasonable test question :-).

Solutions to Sample Test #2 — CSc 422, Fall 2017

1. (20 points) — Develop a solution for a monitor that has one operation:

```
int exchange(int value)
```

The procedure is called by two processes to exchange the values of their actual parameters. The monitor should be reusable in the sense that it should exchange the parameters of the first pair of callers, then the next pair of callers, and so on.

For example, if processes **A**, **B**, **C**, **D**, and **E** call **exchange** in the order listed, then **A** and **B** will exchange values, **C** and **D** will exchange values, and **E** will be blocked waiting for another process to call **exchange**.

You may use either Signal and Continue (**SC**) or Signal and Wait (**SW**). *State which one* you are using.

Solution:

Note: For **SC**, it is necessary to use extra storage to hold the values to be exchanged, and a private condition variable on which to block each odd numbered process. In short, it is easier to solve this problem using **SW** semantics!

SC semantics:

```
monitor ValueSwapper {
      int  FirstValue[], SecondValue[];
      int  count = 0;
      cond NeedAnother[]; # array of private condition variables

procedure exchange( int value ) returns int {
   int localCount = count;
   count++;
   if ( localCount % 2 == 0 ) {
      # Need to put my value into FirstValue, then wait for the
      # next process to call exchange
      FirstValue[localCount / 2] = value;
      wait(NeedAnother[localCount / 2]);
      value = SecondValue[localCount / 2];
      return value;
   }
   else {
      # Second process.  Put my value into SecondValue, pick up
      # FirstValue, wake up other process.
      SecondValue[localCount / 2] = value;
      value = FirstValue[localCount / 2];
      signal(NeedAnother[localCount / 2]);
      return value;
   }
} # exchange
} # ValueSwapper monitor
```

SW semantics:

```
monitor ValueSwapper
      int  FirstValue, SecondValue;
      bool First = true;
      cond NeedAnother;

procedure exchange( int value ) returns int {
   if (First) {
      # Need to put my value into FirstValue, then wait for the
      # next process to call exchange
      FirstValue = value;
      First = false;
      wait(NeedAnother);
      value = SecondValue;
      return value;
   }
   else {
      # Second process.  Put my value into SecondValue, pick up
      # FirstValue, wake up other process.
      SecondValue = value;
      value = FirstValue;
      First = true;
      signal(NeedAnother);
      return value;
   }
} # exchange
} # ValueSwapper monitor
```

2. (20 points) — Consider the following proposed monitor solution to the shortest-job-next allocation problem:

```
monitor SJN {
   bool free = true;
   cond turn;
   procedure request(int time) {
      if ( ! free )
         wait(turn, time);
      free = false;
   } # request

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

The **`wait(turn, time)`** function will block the caller on the **`turn`** condition variable. Processes are blocked on **`turn`** in order of their request time in shortest to longest order.

For each of a.) and b.), provide a justification for your answer. If you think the answer is "no", it is sufficient to provide an example. If you think the answer is "yes", provide an argument as to why the solution is correct.

a.) Does this solution work correctly for Signal and Continue (**`SC`**)?
b.) Does it work correctly for Signal and Wait (**`SW`**)?

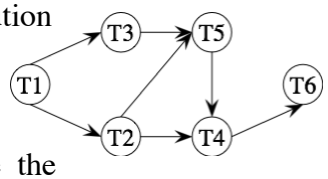
Solution:
a.) Signal and Continue:
No. The process that does a **`release`** wakes up a blocked process. However, the awakened process does not execute right away, since the signaler continues to execute. Further, it is not guaranteed by **`SC`** that the awakened process will execute next. Since the releasing process set **`free`** to **`true`**, a different process may be the next to execute **`request`**, and gain access to the resource. Then, the awakened process might run and also gain access to the resource!

b.) Signal and Wait:
Yes. The awakened process gets to run next, and sets **`free`** to **`false`**. This will prevent any other process from getting the resource. The releasing process will (later) get to run in order to complete the **`release`** procedure. However, there is nothing to complete, so it will simply wake up, gain exclusive access to the monitor, and then exit the monitor.

3. (20 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:

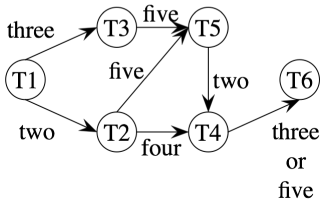


Do not impose constraints not specified in the graph. For example, **`T3`** and **`T2`** can execute concurrently if their predecessors have finished.

For full credit, use the fewest number of semaphores that can still solve the problem correctly.

Solution:

This drawing shows how semaphores can be re-used:




```
   semaphore two = 0, three = 0, four = 0, five = 0

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

4. (20 points) — Suppose there is only one bathroom in a building. It can be used by both men and women, but not at the same time.

Develop a monitor to synchronize the use of the bathroom. The monitor has four public procedures: **`womanEnter`**, **`womanExit`**, **`manEnter`**, and **`manExit`**. A woman process calls **`womanEnter`** to get permission to use the bathroom and calls **`womanExit`** when finished. A man process calls **`manEnter`** and **`manExit`**. State the monitor invariant for your monitor solution.

Use Signal-and-Continue (**`SC`**) semantics. Your solution does not have to be fair.

It is possible for more than one person to be in the bathroom at the same time. Assume the bathroom has infinite capacity (obviously this is unrealistic, but makes for a reasonable test question :-).

Solution:

Monitor invariant: (*numMen* = 0) ∨ (*numWomen* = 0)

```
monitor bathroom {
  # Globals:
  int numWomen = 0, numMen = 0;
  cond goWoman, goMan;

procedure manEnter() {
   while (numWomen > 0)
     wait(goMan);
   numMen++;
 } # manEnter

 procedure manExit() {
   numMen--;
   if (numMen == 0)
     signal_all(goWoman);
 } # manExit

 procedure womanEnter() {
   while (numMen > 0)
     wait(goWoman);
   numWomen++;
 } # womanEnter

 procedure womanExit() {
   numWomen--;
   if (numWomen == 0)
     signal_all(goMan);
 } # womanExit

} # monitor end
```