Programming Assignment 2: Process Synchronization
Due: Mar 26th 2015

Assignment guidelines

In this assignment you will implement process synchronization.

- The best way to work on this assignment is incrementally. Since we are working on synchronization, it is difficult to get it right on the fist attempt: so do not put off the assignment! Work a little on it every day.
- Start with the logic of the code, and work out what the **critical sections** of each problem are.
- Identify the **shared resource** that has to be protected.
- Consider how many threads can **produce** a resource at a time and how many can **consume** that resource at a time. (Hint: Refer back to the bounded-buffer problem from the slides).
- Look at the pseudo code provided to get you started towards thinking about this problem.
- **Please do not put this off to the end!!**

What should you submit?

Submit your assignment as a `.tgz` file on `aurora`. The submission instruction is `submit proj2 yourProj.tgz`. You can name 'yourProj.tgz' a name of your choice. The zipped file should contain:

(a) All your (java) code files, and any other files that might be needed for executing your code.
(b) README.txt
(c) A PDF file (report) containing answers to the pseudo code questions in Task1 and Task2.

I. Task 1: Thirsty Threads (30)

Consider a system with three thirsty threads and one server thread. To drink water, each thread needs three ingredients: water, ice, and a cup. One of the threads has water, another has ice, and the third has a cup. The server has an infinite supply of all three. The server places two of the ingredients (chosen randomly) on the table. The thread that has the remaining ingredient then prepares and has a drink, signaling the server on completion. The server then puts out another two of the three ingredients (at random), and the cycle repeats.

(A) To make your task easier, a skeleton of the solution for this problem is shown below. Before writing and implementing the code, complete this pseudocode. (10)
(B) Write a semaphore based solution in Java to synchronize the server and the thirsty threads. (20)

```
/* a[0] for water, a[1] for ice, a[2] for cup */
semaphore a[3] = 0;
semaphore server = 1;

Server(void) {
int i,j;
  While(TRUE){
    i = random(3); /* returns a random integer 0, 1 or 2 for i */
    j = random(3); /* returns a random integer 0, 1 or 2 for j */
    if (i != j) { /* i and j must be different */
       Wait(________);
          k = 3 - (i+j); /*the drinker with the k-th ingredient
                                           is identified*/
          Signal(________);
        }
    }//end of while
}//end of Server

Drinker(int r) {
/* r indicates which ingredients this drinker has */
   While(TRUE){
       Wait(________);
       Drink( );
       Signal(________);
      }
}//end of Drinker
```

II. TASK 2: THE LAZY DENTIST (30)

There is a dental clinic with N chairs for waiting patients; the clinic has one doctor. If a patient enters the clinic and there are no free chairs, the patient leaves. If a patient enters the clinic and the dentist is sleeping, the patient wakes up the dentist and consults him. Otherwise, the patient enters the clinic, takes a seat, and waits. If the dentist finishes with one patient and there are waiting patients, the dentist takes the next patient. Otherwise, the dentist goes to sleep in his chair.

(A) Below is a skeleton solution for this problem. Before proceeding to code, complete this pseudocode. And to make your task (very) simple look at the comments corresponding to each of the statements. Use these as hints for filling in the blanks! (10)

(B) Write a semaphore based solution in Java to control the actions of patients and the dentist. (20)

```
/* The first two are semaphores are mutexes
(only 0 or 1 possible)*/
Semaphore dentistReady = 0
Semaphore seatCountWriteAccess = 1
/* if 1, the number of seats in the waiting room can be
incremented or decremented*/

/* the number of patients currently in the waiting room,
ready to be served*/
Semaphore patientReady = 0

/* total number of seats in the waiting room*/
int numberFreeWRSeats = N
```

Dentist Method:

```
def Dentist():
 while true: // Run in an infinite loop.
    /* Try to acquire a patient:
       if none is available, go to sleep*/
    wait(____________)
    /*  Awake: try to get access to modify
        # of available seats otherwise sleep*/
   wait(____________)

   /*  One waiting room chair becomes free*/
   numberFreeWRSeats += 1
   /* Doctor is ready to consult.
   signal(____________)
   /* Don't need the lock on the
       chairs anymore. */
    signal(____________)
    // (Talk to patient here.)
```

Customer Method:

```
def Customer():
  /* Run in an infinite loop to
     simulate multiple patients */
 while true:
 /* Try to get access to the
     waiting room chairs. */
  wait(_____________)
  /* If there are any free seats:*/
  if numberFreeWRSeats > 0:
     /* sit down in a chair */
     numberFreeWRSeats -= 1
     /* notify the dentist, who's waiting
         until there is a patient */
     signal(____________)
     /* don't need to lock the
         chairs anymore */
     signal(_____________)
     /* wait until the dentist is ready */
     wait(_____________)
     /* (Consult dentist here.)*/
  else:
  /* otherwise, there are no free seats; tough luck  */
   /* but don't forget to release the
      lock on the seats! */
   signal(_____________)
    // Leave without consulting the dentist.
```

Testing your code: Add a main method to test your code. Assume N = 3.

2.1 Test with one dentist thread and one patient thread.
2.2 Test with one dentist thread and two patient threads.
2.3 Test with one dentist thread and three patient threads.
2.4 Test with one dentist thread and five patient threads.

III. Task 3: Espionage (25)

PeaceLover and PeaceBuff are two neighboring countries battling for the disputed land of PeaceShire. To win the war, PeaceLovers army has infiltrated the army of PeaceBuff with several spies. These spies employ the following protocol for sending secret information to their headquarters: A spy writes the message on a piece of paper and drops it at a predetermined location. An agent from PeaceLover then picks up the paper and delivers it to the headquarters. A spy will drop a new message only if the previous message has already been picked up. If not, the spy will wait until the message is collected by an agent. Obviously, an agent cannot pick up the message unless a spy drops one. To avoid being discovered and intercepted by PeaceBuff's army, the spies write only one word messages.

Write an Espionage class to simulate the above communication protocol. This class has two methods: `dropSpyMsg(String message)` and `agentPickMsg()`. **"Message" is the one word secret message**. In your implementation you can assume that `dropSpyMsg()` waits until `agentPickMsg()` is called on the same Espionage object, and then transfers the word over to `agentPickMsg()`. Once the transfer is made, both can return. Similarly, `agentPickMsg()` waits until `dropSpyMsg()` is called, at which point the transfer is made, and both can return (`agentPickMsg()` returns the word).

Write your Espionage class as a **monitor**. Your solution should work even if there are multiple spies and agents for the same Espionage object. As we discussed in the class, you would not need any locks besides what the monitor already provides.

Hint: This is equivalent to a zero-length bounded buffer. Since the buffer has no room, the producer and consumer must interact directly, requiring that they wait for one another. We saw a similar example in class, when we talked about choosing a sequence of operations, independent of the CPU scheduling using thread.join, and thread.finish.

Testing your code: Add a main method to test your code. Assume N = 3.

3.1 Test with a single spy thread and a single agent thread.

3.2 Test with multiple spies and multiple agent threads.

IV. Task 4: Documentation (15)

4.1 Prepare a "README.txt" file for your submission. The file should contain the following (10):
  - **Names** of all the group members
  - **Instructions** for compiling and executing your program(s). Include an example command line for the same.
  - If your implementation does not work, you should also **document the problems** in the "README.txt" preferably with your explanation of why it does not work and how you would solve it if you had more time.

4.2 You should **comment your code** well. The best way to go about it is to write comments while coding (5).