Campaign Mailer Assignment

Overview

Consider an interesting and challenging synchronization problem, the campaign mailer's problem. Suppose that a campaign mailing requires three items: an envelope, a campaign flyer and a stamp. There are three volunteers, each of whom has only one of these items with infinite supply. There is an agent who has an infinite supply of all three items. To put together a mailing, the volunteer who has an envelope must have the other two items, a campaign flyer and a stamp. (You can figure out what happens with the volunteers who start with campaign flyers and stamps, respectively.) The agent and volunteers share a table. The agent randomly generates two items and notifies the volunteer who needs these two items. Once they are taken from the table, the agent supplies another two. On the other hand, each volunteer waits for the agent's notification. Once notified, the volunteer picks up the items, puts together a mailing, takes the mailing to the mailbox, and goes back to the table waiting for his next items. The problem is to come up with an algorithm for the volunteers using semaphores as synchronization primitives.

Now, all jokes aside, this is an actual problem related to computer science. The agent represents an operating system that allocates resources, while the volunteers represent applications that need resources. The problem is to make sure that if resources are available that would allow one or more applications to proceed, those applications should be awakened. Conversely, we want to avoid waking an application if it cannot proceed.

A restriction of the problem is that you are not allowed to modify the agent code. (Indeed, if the agent represents an operating system it makes sense to assume that you don't want to modify it every time a new application comes along.) The agent uses these binary semaphores and actually consists of three concurrent threads (one of which is given below.) Each waits on `agentSem`. Each time it's signaled, one agent thread wakes up and provides items by signaling two other semaphores.

```
Semaphore agentSem = 1;
Semaphore envelope = 0;
Semaphore flyer = 0;
Semaphore stamp = 0;

while (true)
{
  Wait(agentSem);
  Signal(envelope);
  Signal(flyer);
}
```

(One thing you will need to change when you implement this is to make the three agent threads sleep for a random period of time – up to 200 milliseconds – before beginning to wait on `agentSem`. This will hopefully mix things up and make this more interesting.)

This becomes a hard problem because you can show that the natural solution leads to deadlock. To get around this, propose the use of three additional "pusher" threads that respond to the signals from the agents, keep track of the available items and signal the appropriate volunteer. We first need three Boolean variables to indicate whether or not an item is on the table, three new semaphores to signal the volunteers, and a semaphore for preserving mutual exclusion (as given below.)

```
Boolean isEnvelope = false;
Boolean isFlyer = false;
Boolean isStamp = false;
Semaphore envelopeSem = 0;
Semaphore flyerSem = 0;
Semaphore stampSem = 0;
Semaphore mutex = 1
```

Pseudo code for one of the pushers (the one who wakes up when there's an envelope on the table) appears below. If this pusher finds flyer, it knows that the flyer pusher has already run, so it can signal the volunteer with stamps. Similarly, if it finds stamps, the volunteer with flyers is signaled. Finally, if this is the first pusher to run, it cannot signal any volunteer, so sets `isEnvelope`.

```
while (true)
{
  wait(envelope);
  wait(mutex);

  if (isFlyer)
  {
    isFlyer = false;
    signal(stampSem);
  }
  else
    if (isStamp)
    {
      isStamp = false;
      signal(flyerSem);
    }
    else
      isEnvelope = true;

  signal(mutex);
}
```

The other pushers are similar. Since they do all the work, the volunteer code is trivial. Pseudo code for the volunteer with an envelope appears below; the others are similar. As above simulate the putting together a mailer and taking the mailing to the mailbox by having the thread sleep for a short period of time (up to 50 milliseconds for both the putting together a mailer and taking the mailing to the mailbox).

```
while (true)
{
  wait(envelopeSem);
    Put together a mailer.
  signal(agentSem);
    Take the mailing to the mailbox.
}
```

Your solution to the problem is to create a simulation of this problem using Java threads. Create three agent threads, three pushers and six volunteers (two holding envelopes, two flyers, two stamps). Each volunteer finishes three campaign mailings before exiting. (They said something about being hungry as they left.) As such, rather than loop forever, each agent loops six times, and each pusher twelve times. (Think about it.) Remember to join *all* threads before your program terminates.

Design

1. Write the code for the threads as described in the overview above.

2. Create a driver class and make the name of the driver class **Assignment2** containing only one method: `public static void main(String args[]).`
   The main method itself is fairly short containing code to do the following:
   a. Create the number of instances of each thread as described in the overview above.
   b. Start each instance of the threads created in a above.
   c. The main method needs to keep track of the threads and take care of each thread before it can end. The methods of the Thread class you'll need for this are join() and isAlive(). If the thread is dead then execute a join on it in order to properly dispose of that thread. If the thread is still alive then move on to the next thread. The main method keeps doing this check until the last remaining thread has died and been properly disposed.

3. You must declare public each class you create which means you define each class in its own file.

4. You must declare private the data members in every class you create.

5. You can use "**implements Runnable**" or "**extends Thread**" to implement the classes defining the threads in this assignment. You can't use extends in this assignment in defining any class except if you are using "**extends Thread**" to define a thread class.

6. **Tip:** Make your program as modular as possible, not placing all your code in one .java file. You can create as many classes as you need in addition to the classes described above. Methods being reasonably small follow the guidance that "A function does one thing, and does it well." You will lose a lot of points for code readability if you don't make your program as modular as possible. But, do not go overboard on creating classes and methods. Your common sense guides your creation of classes and methods.

7. Do **NOT** use your own packages in your program. If you see the keyword **package** on the top line of any of your .java files then you created a package. Create every .java file in the **src** folder of your Eclipse project, if you're using Eclipse.

8. Do **NOT** use any graphical user interface code in your program!

9. Do **NOT** type any comments in your program. If you do a good job of programming by following the advice in number 6 above then it will be easy for me to determine the task of your code.

Grading Criteria

The total assignment is worth 20 points, broken down as follows:

1. If your code does not implement the task described in this assignment then the grade for the assignment is zero.
2. If your program does not compile successfully then the grade for the assignment is zero.
3. If your program produces runtime errors which prevents the grader from determining if your code works properly then the grade for the assignment is zero.

If the program compiles successfully and executes without significant runtime errors then the grade computes as follows:
Followed proper submission instructions, 4 points:

1. Was the file submitted a zip file.
2. The zip file has the correct filename.
3. The contents of the zip file are in the correct format.
4. The keyword **package** does not appear at the top of any of the .java files.

Code implementation and Program execution, 12 points:

- The driver file has the correct filename, **Assignment2.java** and contains only the method **main** performing the exact tasks as described in the assignment description.
- The code performs all the tasks as described in the assignment description.
- The code is free from logical errors.
- Program output, the program produces the correct results for the input.

Code readability, 4 points:

- Good variable, method, and class names.
- Variables, classes, and methods that have a single small purpose.
- Consistent indentation and formatting style.
- Reduction of the nesting level in code.

Late submission penalty: assignments submitted after the due date are subjected to a 2 point deduction for each day late.

Late submission policy: you **CAN** submit your assignment early, before the due date. You are given plenty of time to complete the assignment well before the due date. Therefore, I do **NOT** accept any reason for not counting late points if you decide to wait until the due date (and the last possible moment) to submit your assignment and something happens to cause you to submit your assignment late.

Submission Instructions

Go to the folder containing the .java files of your assignment and select all (and **ONLY**) the .java files which you created for the assignment in order to place them in a Zip file. The file can **NOT** be a **7z** or **rar** file! Then, follow the directions below for creating a zip file depending on the operating system running on the computer containing your assignment's .java files.

Creating a Zip file in Microsoft Windows (any version):

1. Right-click any of the selected .java files to display a pop-up menu.
2. Click on **Send to**.
3. Click on **Compressed (zipped) Folder**.
4. Rename your Zip file as described below.
5. Follow the directions below to submit your assignment.

Creating a Zip file in Mac OS X:

1. Click **File** on the menu bar.
2. Click on **Compress ? Items** where ? is the number of .java files you selected.
3. Mac OS X creates the file **Archive.zip**.
4. Rename **Archive** as described below.
5. Follow the directions below to submit your assignment.

Save the Zip file with the filename having the following format:

your last name,
followed by an underscore _,
followed by your first name,
followed by an underscore _,
followed by the word **Assignment2**.

For example, if your name is John Doe then the filename would be: **Doe_John_Assignment2**

Once you submit your assignment you will not be able to resubmit it!
Make absolutely sure the assignment you want to submit is the assignment you want graded.
There will be **NO** exceptions to this rule!

You will submit your Zip file via your CUNY Blackboard account.
The only accepted submission method!

Follow these instructions:

Log onto your CUNY BlackBoard account.
Click on the CSCI 340 course link in the list of courses you're taking this semester.
Click on **Content** in the green area on the left side of the webpage.
You will see the **Assignment 2 – Campaign Mailer Assignment**.
Click on the assignment.
Upload your Zip file and then click the submit button to submit your assignment.

Due Date: Submit this assignment by Thursday, December 12, 2019.