COSC 2P13 – Assignment 2 – Now That's a Spicy Meatball!

Due: Thursday, July 2\textsuperscript{nd}, 11:00 AM

Consider the problem of trying to cook a large number of dishes in a woefully underequipped kitchen. Sally decided to start her own restaurant. Because it isn't at all awkward or weird when people do this, she decided to just convert the bottom of her house. Seriously, that isn't at all weird, and patrons really like seeing Sally's family going into the living room to eat and watch tv. In a restaurant. Yup. Totes.

Erm, moving on... because it was adapted from a tiny home, there are only two stove burners free, and room in the oven for a single dish at a time. Currently, Sally's menu consists solely of steak, pizza, and spaghetti sauce (don't ask why there's no noodles. A wizard did it). Largely because she's going from nearly-completed meals from frozen, all three dishes are very simple, with only a few steps each. Additionally, she sometimes has her family helping her cook.

Each dish can be made as follows:

- For spaghetti:
  - Get the only canister of spices in the apartment and season
  - Place on a burner (stove) and simmer
  - While it's still simmering \textbf{on the stove}, grab the spices and season again

- For pizza:
  - Toss it in the oven and bake
  - While it's still \textbf{in the oven}, grab the spices and carefully season

- For steak:
  - Grab the spices and season
  - Place on a burner (stove) and fry
  - Move from stove to oven (it can't be taken off of the stove unless it can be put into the oven)

For the sake of this exercise, assume:

1. Unlimited counter space (in other words, the only limited resources are the single cannister of spices, single oven, and two stove burners)
2. Unrealistically flexible cooking times (to the point where only the relative sequence of events matters)
3. The possibility of anywhere from a single chef dividing time between all meals to a dedicated chef for each meal (i.e. uniprocessor or multiprocessor)
4. You don't need to bother managing resources for starting or ejecting meals (e.g. in my own sample below, I just announced 'Pasketti' to show when spaghetti was being started, and all meals were finished with 'done')

Clearly, a \textbf{counting semaphore} would be appropriate here, due to the dual burners. As such, assume counting semaphores for the entirety of this assignment.

1. Write a sequence for cooking spaghetti that would work perfectly fine for only a single dish at a time, but could potentially lead to deadlock if more than two spaghetti dishes were allowed at the same time. Use pseudocode similar in style to the in-class examples. Show an example of how deadlock could occur.
2. Write a sequence for cooking spaghetti that will avoid deadlocks.
3. Write sequences for pizza and steak such that:
   1. If all threads were only pizza, there would be no deadlocks.
   2. If all threads were only steak, there would be no deadlocks.
   3. If there is a combination of the two, a deadlock is possible.
   Show an example of how deadlock could occur.

4. Write sequences for cooking pizza and steak together where deadlock is not possible.

5. This is the fun part: You're going to write a Java simulation to implement your sequences with counting semaphores.
   It's worth noting that, instead of wait() and signal(), the Java semaphores instead use acquire() and release() (respectively). Other than that, they're fairly straight forward. Review that documentation to see how to use them.
   I'd personally suggest having a separate Semaphores class to hold public static references to the semaphores so that all of the threads can easily access them.

Your program must include the following:
- Each dish is prepared in an infinite loop (e.g. pasketti → season → cook → season → done → pasketti → season...)
- Each dish is represented by a different class which extends Thread
- Each dish must announce at least its current action to the console (standard out)
  - This output must be thread-safe, which means you'll need another semaphore for output
  - To make this cleaner to read, I'd suggest having a separate class with static output methods
  - The output must somehow make it clear which thread is saying what
- You may assume there will always be 5 threads if you wish, but must provide some easy mechanism for deciding which dishes are chosen for each thread.
  - For example, the marker should be able to have 5 spaghetti, 3 pizzas and 2 steaks, etc.
  - Alternatively, you may let the user choose how many threads to have, but must still provide for the ability to have 5 threads
- It's recommended (though not necessary) to include some notion of a randomly-selected delay between steps (e.g. anywhere between 0 and 50 ms)

As mentioned above, you need to make it clear which thread is generating each message. Passing an identifying string to each thread in the constructor is a good way to do this. You could then prepend this string to the beginning of any subsequence output. One option would be to pass an ID number (e.g. "1."). Personally, I went with different numbers of 'tab' characters ('\t') to create a column for each thread.

   e.g.
   
   'Za
   seasoning
   frying

   'Pasketti
   seasoning
   broiling
   frying
   done
   Beef!
   broiling
seasoning  done
baking
simmering
Beef!
seasoning
simmering
seasoning
seasoning
seasoning
seasoning
seasoning
done
'dZa
done

There are five columns in the example above, with the first and third being spaghetti, the second being pizza, and the fourth and fifth being steak.

Submission: Both physical and electronic submission are required. The posted due date applies to both physical and electronic submission. Seriously.

To submit electronically, slap everything into a single folder on sandcastle, SSH/PuTTY in, navigate to that folder, and run submit2p13.
For physical submission, include your answers for questions 1-4, printouts of your code, a few sample outputs, and don't forget the signed departmental cover page. Deposit it in the appropriate dropbox in J-block.