Week 4 – Coursework

There is no pop quiz this week – you will be focusing solely on the coursework which is a programming exercise! Before providing details some reminders pertaining to the coursework:

1) You must submit your coursework onto Moodle before demoing.

2) We will be unable to provide support in Week 5 labs, which are dedicated to solely marking. I will be running evening sessions in Week 4 – this is your best chance to seek clarification on the exercise.

3) You are not allowed to use any pre-existing concurrency libraries – **doing so will result in an automatic fail on any submitted coursework**. Using operations that 'hide the complexity' of the underlying concurrency primitive makes it impossible for us to determine whether you fully understand what is occurring. If you are unsure what this entails let me know ASAP.

4) Part of your marks will be based on appropriate use of classes, commenting, and formatting. Make sure that the editor which displays your code formats correctly.

5) Further details on the grading criteria can be found in the marking guidelines document.

Programming Exercise

You will be working with the files found under **WK4_coursework**, which are incomplete and will not compile. This program shares similar functionality to the one provided in Week 3.

In this program X users will add elements onto a single buffer object, which are removed by Y webservers. The number of users, webservers and elements are all specified by user input, and resembles the following:

```
Enter buffer capacity
20
Enter number of users
10
Enter number of servers
10
Enter number of total elements
100
```

The program must automatically divide the number of inputted elements evenly across all users (even for odd values).

The action of adding and removing elements from the buffer must be performed concurrently, and will resemble the following in mid execution:

User 8 adds an element 1/20
User 6 adds an element 2/20
User 9 adds an element 3/20
User 0 adds an element 4/20
Serv 2 removed element 3/20
User 1 adds an element 4/20
Serv 5 removed element 3/20
Serv 6 removed element 2/20

The program must also continue to operate even if the buffer is empty or full, with the user or webserver waiting until the buffer is again available. An example of this will resemble the following when the buffer is empty:

Serv 6 removed element 2/20
Serv 3 removed element 1/20
Serv 2 removed element 0/20
Buffer empty – web server wait
Buffer empty – web server wait
User 2 added an element 1/20
User 6 added an element 2/20

And will resemble the following when the buffer is full:

User 9 added an element 19/20
User 5 added an element 20/20
Buffer full – User now sleeping
Serv 1 removed element 19/20
User 8 added an element 20/20
Buffer full – User now sleeping
Buffer full – User now sleeping
Serv 2 removed element 19/20

Once all users and server have completed processing all elements, the program should output something resembling the following:

User 7 created a total of 10 elements
User 5 created a total of 10 elements
User 0 created a total of 10 elements
User 6 created a total of 10 elements
User 8 created a total of 10 elements
User 1 created a total of 10 elements
User 2 created a total of 10 elements
User 3 created a total of 10 elements
User 9 created a total of 10 elements
User 4 created a total of 10 elements
Consumer 9 consumed a total of 10 elements
Consumer 1 consumed a total of 10 elements
Consumer 3 consumed a total of 10 elements

```
Consumer 6 consumed a total of 10 elements
Consumer 4 consumed a total of 10 elements
Consumer 0 consumed a total of 10 elements
Consumer 7 consumed a total of 10 elements
Consumer 2 consumed a total of 10 elements
Consumer 5 consumed a total of 10 elements
Consumer 8 consumed a total of 10 elements
```

As well as check that the buffer is empty, and that all created threads have completed:

```
----------------------
Buffer has 0 elements remaining
----------------------
Server thread 0 is alive: false
Server thread 1 is alive: false
Server thread 2 is alive: false
Server thread 3 is alive: false
Server thread 4 is alive: false
Server thread 5 is alive: false
Server thread 6 is alive: false
Server thread 7 is alive: false
Server thread 8 is alive: false
Server thread 9 is alive: false
User thread 0 is alive: false
User thread 1 is alive: false
User thread 2 is alive: false
User thread 3 is alive: false
User thread 4 is alive: false
User thread 5 is alive: false
User thread 6 is alive: false
User thread 7 is alive: false
User thread 8 is alive: false
User thread 9 is alive: false
----------------------
Program took 10052 milliseconds to complete
```

Please note that the time does not need to be exactly the same – this will be entirely dependent on the number of elements processed by user defined values for users and webservers.

Your assignment is to make this program function correctly – allowing for concurrent access between users and servers on a single buffer. You must demonstrate to us that the program will operate in a thread-safe manner under the following scenarios:

1) Program correctly operated under any scale of buffer size, number of users, web servers, and elements
2) Buffer is frequently empty (i.e. web servers removes elements faster than users add)
3) Buffer is frequently full (i.e. users adds elements faster than web servers remove)

To demonstrate 2 and 3 I would advise that you use the java thread sleep method on users and web servers to control their execution.

Important: You are NOT allowed to use the synchronized keyword within any of your methods EXCEPT within synchronization primitive classes that you have designed yourself (i.e. semaphores, locks).

When you are ready to demo, myself or one of the TAs will sign off your work.
