Lab 4: Producer-Consumer and Readers and Writers Problems

In this lab, we will look into and try to solve the producer-consumer and readers and writers problems in Java.

A. Producer-Consumer Problem

In this problem, there are two parties (producer and consumer) that are trying to exchange data by using a bounded buffer. The producer (a thread) is the one that adds data to the buffer, whereas the consumer (another thread) is the one that retrieves data from the buffer. However, it does not mean that a producer will also act as a producer, or a consumer will always act as a consumer. For example, a consumer may act as the producer if it is required to pass the data to another party (just another thread). In this case, we need to ensure that a producer is not allowed to add new data to the buffer when it is full, and a consumer is not allowed to retrieve any data from the buffer when it is empty.

As shown below is an example that demonstrate how this could be done by using the wait() method and the notify() method covered in previous session. In this case, we first defined our own data type known as Task. Then, an array of type Task is created to store the Task. Two methods are included in the BoundedBuffer class. The put() method is to be executed by the producer when it is trying to add a Task, and the get() method is to be called by the consumer when it is trying to retrieve a Task.

```
// Example 1
class Task {
   private String fileName;
   Task(String fileName) { this.fileName = fileName; }
   public String getFileName() { return fileName; }
}

class BoundedBuffer {
   private Task buffer[];
   private int first;
   private int last;
   private int numberOfTaskInBuffer;
   private int size;

   BoundedBuffer(int length) {
      size = length;
      buffer = new Task[size];
      last = 0;
      first = 0;
   }

   public synchronized void put(Task task) throws InterruptedException {
      while (numberOfTaskInBuffer == size) { wait(); }
      last = (last + 1) % size;
      numberOfTaskInBuffer++;
      buffer[last] = task;
      notifyAll();
   }
```

```
    public synchronized Task get() throws InterruptedException {
        while (numberOfTaskInBuffer == 0) { wait(); }
        first =(first + 1) % size;
        numberOfTaskInBuffer--;
        notifyAll();
        return buffer[first];
    }
}
```

Exercise 1

a) A printer is waiting for printing tasks that could come from computer A or computer B. When there is no printing task, the printer should stay in the waiting state. The printer will only wake up to perform a printing task when computer A or computer B sends a signal to the printer. The printer takes one second to print a job, and the printer will not accept new tasks while it is printing. After sending each printing task, the computer should also pause for half a second before sending the next task. In this case, computer A and computer B are the producers, and the printer is the consumer. Simulate the above by using the wait() method and the notify() method. Assume that computer A and computer B will each send 3 printing tasks to the printer.

b) Implement a queue (buffer) to keep track of the printing to be completed by the printer. You just need to keep the file names of those files that needs to be printed in the queue. With this, the printer should be able to continue receive printing task while it is printing. The code given in example 1 should be able to assist you in creating the queue needed for this.

B. Readers and Writers Problem

Up to this point, we have covered how to allow multiple threads to modify a shared variable safely, at the same time, with synchronization. But so far, we only have one thread that could be reading the shared variable (no matter it is at the end of modifying the shared variable, or halfway). In this problem, we are going to see how we can allow multiple writer threads and multiple reader threads to work concurrently. In this case, the writer threads will only be modifying a shared variable, and the reader threads will only be reading the shared variable.

You might argue that the solution can be very simple, just enclosed the shared variable in synchronized blocks and all the threads will have to acquire the same lock before they can proceed to modify or read the shared variable. Although this is one of the possible solutions, it does not allow the reader threads to read at the same time. In fact, there is no harm to allow multiple reader threads to read at the same time, because they are just merely reading the shared variable. Moreover, it also does not allow us to choose whether we would like to prioritize the writer threads or the reader threads.

As shown below is an example that demonstrate how we can create a shared structure consists of a variable called counterValue, and several methods that can be used as the entry and exit protocols. The protocols are used to control the access to the critical session. For example, a writer thread will need to first call the startWrite() method to check if it is allowed to enter the critical section. If it is allowed to proceed, then it can continue to call the increaseCounterValueByOne() method to modify counterValue (assume it is the task given to the writer thread), and then call the stopWrite() method after it is done. Similarly, a reader thread will need to first call the startRead() method to check it is fine to read counterValue. If it is allowed to do so, then it can proceed to call the getCounterValue() method and call the stopRead() method after it is done.

```
//Example 2
class SharedStructureReadersPreference {
   private int readers = 0;
   private boolean writing = false;
   private int counterValue = 0;

   public synchronized void startWrite() throws InterruptedException {
      while(readers > 0 || writing) {    // Wait until no waiting readers.
         wait();
      }
      writing = true;
   }

   public synchronized void stopWrite() {
      writing = false;
      notifyAll();
   }

   public synchronized void startRead() throws InterruptedException {
      while(writing) {                    // Wait if writing is in progress.
         wait();
         readers++;
      }
   }

   public synchronized void stopRead() {
      readers--;
      notifyAll();
   }

   public void increaseCounterValueByOne() {
      counterValue++;
   }

   public int getCounterValue() {
      return counterValue;
   }
}
```

Exercise 2

a) Based on the code given in example 2, create 3 writer threads and 3 reader threads that will try to access counterValue. Each writer will call the increaseCounterValueByOne() method for 10 times, and each reader will call the getCounterValue() method for 10 times. All the threads will need to implement the entry and exit protocols to ensure that the writing and reading process can be carried out safely (e.g. no race condition and data inconsistency). Run the code several time and try to see if you can observe any specific pattern from the outputs.

b) Modify the code given in example 2 to prioritize the writer threads. In other words, the reader threads are not allowed to read when there is a waiting writer thread. Then, create 3 writer threads and 3 reader threads that will try to access counterValue in the same way as described in part (a).