Middle East Technical University - Department of Computer Engineering

CENG 443

Introduction to Object Oriented
Language and Systems

Fall '2022-2023

Homework 2 - Printer Queue

Due Date: 20 December 2022, Tuesday, 23:55
Late Submission Policy will be explained below

Objectives

The objective of this assignment is to familiarize you with the development of multi-threaded application and synchronization using java. Your task is to implement a Multiple-Producer Multiple-Consumer (MPMC) Queue with different prioritization of elements. Then such queue will be used to simulate a printer room with multiple printers (consumers). And multiple print operations submitted by multiple people; namely Students and Instructors. Instructor print jobs will have precedence over the Student print jobs. Specifications of the game is given Section 1.
Keywords: Thread, MPMC Queue, Condition Variable, Atomic, Runnable.

1 Specifications

As discussed above you are expected to implement a printer room class that are going to use your own implementation of MPMC Queue. Specifications are as follows:

1. There will be printer room with multiple printers. Amount of printers will be determined by the user using **`void PrinterRoom(...)`** constructor. Lets name it P_n.
2. Buffer size of the queue is also be provided by such constructor. Lets name it Q_n.

   > **Info:** This does not mean that you are required to have exactly Q_n size memory of each element of the queue. For example you can use Java data structure(s) (i.e. **`ArrayList<T>`**) to hold data internally, such classes may pre-allocate more memory than you need, this is not important.

3. There will be a **single** MPMC Queue of this room.
4. MPMC Queue items have a single type **`PrintItem`**.
5. However, each print item may have different origin. This origin is denoted by the **`enum PrintType`**. These are namely **`INSTRUCTOR`** and **`STUDENT`**.
6. Each printer have its own thread and try to access items over this shared queue, in First in First out (FIFO) manner. However, print submissions from **Instructors will have precedence over the Students**. Meaning that, if instructor submits an item to the queue it should be processed before students but FIFO order between Instructors and Students should remain.

7. This precedence does **not** mean that if queue is full and an instructor tries to submit, it will replace a student over the queue. It should wait just like any other item. We will discuss waiting routines shortly.
8. As discussed, queue can only hold Q_n amount of print items. If queue is full, print submitter should **wait** until there is a space on the queue. This wait should **not be busy wait**. After acquiring a print item form the queue, print submitter (producer) will **notify** the waiting printer threads.
9. Similar to the item above, printers will **wait** if there are no elements on the queue. Likely; if a new print is submitted by other threads (producers), printers will be **notified** and wake up.
10. When a producer thread tries to submit to the printer room, it should log its effort to on a log file, **before potentially waiting on the queue**.
11. Similarly, if a printer (consumer) accomplishes the printing job it should log its effort on a log file.
12. Another thread which not a Producer or a Consumer (probably the main thread) can try to close the room at anytime. When this event occurs, printer room should not accept any other printing events other than the items are already in the queue. All waiting producer threads, should be notified that queue is now closed, and items that are tried to be submitted will not be processed.
13. Consumer threads (printers) should not terminate until every item in the queue is processed. After that they will terminate. Just before termination, these threads log their termination to the log file.
14. Printers should log their creation as well.
15. Only after these two event is finalized, the thread that closes the room should continue executing.
16. Logging functionality is already provided for you. Formatting strings are also provided in order to ensure consistency of your outputs. An example can bee seen on Listing 1.
17. Log files are important, these will be checked to see if your functionality is proper.
18. A single producer thread may submit different types of print items. Do not assume a producer thread can only submit one type of print item.
19. You are only partially responsible for the implementation of the producer threads. **`boolean SubmitPrint(PrintItem item, int producerId)`** will be called from the different producer threads. This function should return false only when the queue is closed. **`int producerId`** argument is provided to log which consumer thread is called this function.
20. You can assume Producer thread only produces values, and consumer thread do consume items. (Testing function should not be able to access to the Queue anyway if you stick OOP principles, we may test this though.)
21. Rest of the routines of the producers will be implemented by the testing class. You should test your application thoroughly by implementing test cases.
22. You can share these test cases over the ODTUClass. **Please make sure that you are sharing only producer related code. Sharing PrinterRoom, PrintQueue related code is strictly forbidden.**
23. For testing, I would suggest writing inner class on the Main.java class and use it for testing. It will be easier to share with your colleagues.
24. Your queue should implement **`interface IMPMCQueue<T>`**. Implementation of this interface is required to throw exceptions after **`void CloseQueue()`** is called and according to the functions:
    - **`void Add(..)`** function should start to throw immediately after queue is closed.

- **`void Consume(..)`** function should start to throw only after queue is closed **and** there are no elements are left on the queue.

25. You should use these exceptions to handle printer room close functionality.
26. Exceptions should be handled internally, you can not assume consumer thread or main thread will catch exceptions that you throw.

```
// This is called by the producer thread just before add
SyncLogger.Instance().Log(SyncLogger.ThreadType.PRODUCER, producerId,
                          String.format(SyncLogger.FORMAT_ADD, item));
```

Listing 1: An example Log routine that is called by the producer thread

2 Class Details

2.1 `public class SyncLogger`

SyncLogger class is already implemented singleton class that will be used to log your outputs. It is already thread safe; thus you can directly call the **`public void Log(ThreadType t, int uniqueId, String s)`** function on the appropriate places. Please use the format strings provided (can be seen on Listing 2) inside of the class. use **`String.format(String format, Args...)`** to format your strings. These strings have appropriate spacing for monospace fonts, console output or a proper text editor should output the strings neatly.

```
// Format String to prevent print mistakes
public static final String FORMAT_PRINTER_LAUNCH    = "Printer %d is launched.";
public static final String FORMAT_PRODUCER_LAUNCH   = "Creating Producer %d";
public static final String FORMAT_PRINT_DONE        = "Printing        %s is done!";
public static final String FORMAT_ADD               = "Trying to Add   %s";
public static final String FORMAT_ROOM_CLOSED       = "Room is closed, %s is skipped!";
```

Listing 2: Format strings in order to prevent mistakes

SyncLogger will output every "Log" function execution on a single line with semicolon separated manner:

- First column will show the thread name with an unique thread type specific id
- Second column will show the elapsed *nanoseconds* since the start of the program.
- Third column will show the actual output of the log function.

SyncLogger may not output the strings in time order. Make sure you are aware of that. Testing will sort the log file wrt. time and check if proper operations are happening on the system.

You are not allowed to change this class.

2.2 `public class PrintItem`

PrintItem class is also provided for you. This class have two properties; a print duration and an unique id. PrintItem class has a mock print function which will just force the invoked thread to sleep as long as its "printDuration" property. Additionally, it has overridden **`String toString()`** method.

2.3 `public class QueueIsClosedExecption`

Exception class that will be thrown by the queue to the producers and consumers.

2.4 `public interface IMPMCQueue<T>`

Already defined interface that you are going to implement. Every operation except **`int RemainingSize()`** should wait (**not busy-wait**) when appropriate. Please check javadoc comments for further explanations. Entirety of this interface can be seen on Listing 3.

You are not allowed to change this interface.

2.5 `public class Main`

Class that contains the main method. This is given to you for testing your code. It already has a simple code snippet that does a simple test. This test produces print items on a single thread. Actual tests will utilize multiple producer threads. Do not forget that the final evaluation(s) will be more complex and covers many other cases.

Do not provide this class when submitting. It may clash with the tester main functions!

2.6 `public class PrinterRoom`

PrinterRoom class is the hub class of the system. It holds printers (consumers) and the shared queue. Provides functionality to the producers. It is partially implemented, please fill the gaps that are indicated in the comments.

Function Explanations:

- **`public boolean SubmitPrint(PrintItem item, int producerId)`**: This function will be called by the producer threads, thus it should be thread safe. It will log the add operation **before** a potential wait. Returns **false** only if the room is closing (**`void CloseRoom()`** is called) returns true otherwise.
- **`public PrinterRoom(int printerCount, int maxQueueCount)`**: Constructor is already implemented for you. It just instantiates your shared queue implementation (**`public class PrinterQueue`**) with the given size and creates P_n amount of printers with unique ids.
- **`public void CloseRoom()`**: This function will be called from another thread that is not either a producer nor a consumer. Calling thread will wait until all the printer threads are terminated. When this function is called, waiting producer threads should be notified and **boolean SubmitPrint(....)** should start to return false. Only after queue is empty, printer threads (consumers) should be allowed to terminate.

2.6.1 `public class Printer`

This is the actual consumer class. When instantiated, it should create its own thread and run over the shared queue of the *PrinterRoom* class. When **`void CloseRoom()`** function is called, instance of this class should terminate when all of the elements in the queue is consumed.

> **Info:** If you do not like this class as an inner class, you can change it. It is provided like this in order to reduce the file clutter.

```
1  public interface IMPMCQueue<T>
2  {
3      /**
4       * Adds data to the queue, waits if queue is full
5       *
6       * @param data data to be added into the queue
7       * @throws QueueIsClosedExecption when adding operation is suspended.
8       */
9      void Add(T data) throws QueueIsClosedExecption;
10
11     /**
12      * Consumes the first element with respect to the priority of the type T,
13      * waits until an element is available in the queue
14      *
15      * In this HW, if there is an instructor print item it should be prioritized
16      * from the student print items (but 'first come, first served' is still true
    ↪  internally
17      * between students and instructors).
18      *
19      * @throws QueueIsClosedExecption when there are no elements left on the queue
    ↪  and queue is closed
20      *          (with CloseQueue function)
21      * @return returns the element
22      */
23     T Consume() throws QueueIsClosedExecption;
24
25     /**
26      * Non-blocking query function, this is technically an approximate value
27      * since after function succeeds, another thread may remove/add an item.
28      * @retun remaining size
29      */
30     int RemainingSize();
31
32     /**
33      * Notifies every thread that is waiting on this queue (threads that are waiting
34      * on the functions will return throw QueueIsClosedExecption as a notification)
35      *
36      * After this call,
37      *   - Consumers should not be terminated until every item on the queue is
    ↪  processed. After that
38      *     they should be terminated.
39      *   - Queue should not accept any add operations.
40      *
41      * This function should return only after all elements on the queue are
    ↪  processed
42      */
43     void CloseQueue();
44 }
```

Listing 3: MPMC Queue Interface

2.7 public class PrinterQueue

This is the implementation of the IMPMCQueue, which will be fully implemented by you.

3 FAQ

1. **Q: Should we assume that the consumer is smart and when it received false from the function `void SubmitPrint(...)`, it will not call this function again?**

   A: No. Always assume a user of your class can do mistakes. Prevent that consumer to wait over your system again. After **`void CloseRoom()`** is called there should not be any waiting by any of the producers.

2. **Q: Is there any limit on the Producer/Consumer amount or Queue size?**

   A: There are no limitations over these parameters. Testers will check many configurations. Only limitation that queue has a **static** size meaning that when there are "size" amount items in the queue submitters (producers) should wait.

3. **Q: Is there any "print.log" sample that we can check?**

   A: ODTUClass course page have a sample log file.

4. **Q: I have getting exceptions, deadlocks while using concurrency constructs (`.wait()`, `.notify()` over Object wrappers of the integral types (Integer, Float etc.). What is going on?**

   A: Object wrappers of the integral types may be shared between instances. For example **`new Integer(0)`** call from the separate portions of the code could be the same Integer object even if you create it using new operation. This is a design element of the Java Programming Language. Since these types are immutable by default. I would not suggest you to use these when using synchronization, you create custom wrapper object of the integral types (from int for example) and use these classes for synchronization and wait.

Regulations

- **Programming Language:** You will use Java (version 8 or above).
- **Late Submission Policy:** A penalty of $5 \times Lateday^2$ will be applied for submissions that are late at most 3 days. If unusual circumstances truly beyond your control prevent you from submitting an assignment, you should discuss this with the course staff as soon as possible. If you contact us well in advance of the deadline, we may be able to show more flexibility in some cases.
- **Newsgroup:** You must follow the Forum (`odtuclass.metu.edu.tr`) for discussions and possible updates on a daily basis.
- You are allowed to use java multi-threading classes. However there should not be any busy waiting on your code.
- Evaluation will be done using both using black-box method and manually. Black-box method will check the "print.log" file that is outputted by your implementation. Thus, formatting is important and please use the provided format string on the "SyncLogger" class when outputting to the log file.
- Manual checking is done to make sure that you are not busy waiting anywhere on the code and to check that you comply with the Object Oriented Programming principles. Both would cause a penalty. Non-terminating submissions will get **zero** on that corresponding black-box test.
- Everything you submit should be your own work. Usage of binary source files and codes found on internet is strictly forbidden.
- **Submission:** Submission will be made via ODTUClass. Create a zip file named "hw2.zip" that contains all your source code files. **Please do not include Main.java file on your submissions! Evaluation scripts will use different main function(s) and it may clash with the submitted Main.java file.**
- Your code should be able to be compiled using **`"java *.java"`** command line statement.