Thread Safe Deferred Callback

Asynchronous programming involves being able to execute functions at a future occurrence of some event. Designing a thread-safe deferred callback class becomes a challenging interview question.

Problem Statement

Design and implement a thread-safe class that allows registeration of callback methods that are executed after a user specified time interval in seconds has elapsed.

Solution

Let us try to understand the problem without thinking about concurrency. Let's say our class exposes an API called `registerCallback()` that'll take a parameter of type `Callback`, which we'll define later. Anyone calling this API should be able to specify after how many seconds should our executor invoke the passed in callback.

One naive way to solve this problem is to have a busy thread that continuously loops over the list of callbacks and executes them as they become due. However, the challenge here is to design a solution which doesn't involve a busy thread.

If we restrict ourselves to use only concurrency constructs offered by Java then one possible solution is to have an execution thread that maintains a priority queue of callbacks ordered by the time remaining to execute each of the callbacks. The execution thread can sleep for the duration equal to the time duration before the earliest callback in the min-heap becomes due for execution.

Consumer threads can come and add their desired callbacks in the min-heap within a critical section. However, whenever a consumer thread requests a callback be registered, the caveat is to wake up the execution thread and recalculate the minimum duration it needs to sleep for before the earliest callback becomes due for execution. It is possible that a callback with an earlier due timestamp gets added by a consumer thread while the executor thread is current;y asleep for a duration, calculated for a callback due later than the one just added.

Consider this example: initially, the execution thread is sleeping for 30 mins before any callback in the min-heap is due. A consumer thread comes along and adds a callback to be executed after 5 minutes. The execution thread would need to wake up and reset itself to sleep for only 5 minutes instead of 30 minutes. Once we find an elegant way of capturing this logic our problem is pretty much solved.

Let's see how the skeleton of our class would look like:

Class Skeleton

```
public class DeferredCallbackExecutor {

    PriorityQueue<CallBack> q = new PriorityQueue<CallBack>(new Comparator<CallBack>() {

        public int compare(CallBack o1, CallBack o2) {
            return (int) (o1.executeAt - o2.executeAt);
        }
    });

    // Run by the Executor Thread
    public void start() throws InterruptedException {

    }

    // Called by Consumer Threads to register callback
    public void registerCallback(CallBack callBack) {

    }

    /**
     * Represents the class which holds the callback. For simplicity instead of
     * executing a method, we print a message.
     */
    static class CallBack {

        long executeAt;
        String message;

        public CallBack(long executeAfter, String message) {
            this.executeAt = System.currentTimeMillis() + executeAfter * 1000;
            this.message = message;
        }
    }
}
```

We define a simple `CallBack` class, an object of which will be passed into the `registerCallback()` method. This method will add a new callback to our min heap. In Java the generic `PriorityQueue` is an implementation of a heap which can be passed a comparator to either act as a min or max heap. In our case, we pass in a comparator in the constructor so that the callbacks are ordered by their execution times, the earliest callback to be executed sits at the top of the heap.

For guarding access to critical sections we'll use an object of the `ReentrantLock` class offered by Java. It acts similar to a mutex. Also we'll introduce the use of a Condition variable. The execution thread will wait on it while the consumer threads will signal it. The condition variable allows the consumer threads to wake up the execution thread whenever a new callback is registered. Let's write out what we just discussed as code.

```
public class DeferredCallbackExecutor {

    PriorityQueue<CallBack> q = new PriorityQueue<CallBack>(new Comparator<CallBack>() {

        public int compare(CallBack o1, CallBack o2) {
            return (int) (o1.executeAt - o2.executeAt);
        }
    });
    // Lock to guard critical sections
    ReentrantLock lock = new ReentrantLock();

    // Condition to make execution thread wait on
    Condition newCallbackArrived = lock.newCondition();

    public void start() throws InterruptedException {

    }

    public void registerCallback(CallBack callBack) {
        lock.lock();
        q.add(callBack);
        newCallbackArrived.signal();
        lock.unlock();
    }

    static class CallBack {

        long executeAt;
        String message;

        public CallBack(long executeAfter, String message) {
            this.executeAt = System.currentTimeMillis() + executeAfter * 1000;
            this.message = message;
        }
    }
}
```

Note how in `registerCallback()` method we lock the critical section before adding the callback to the queue. Also, we signal the condition associated with the lock. As a reminder, note the execution thread, if waiting on the condition variable, will not be able to make progress until the consumer thread gives up the lock, even though the condition has been signaled.

Now lets come to the meat of our solution which is to design the execution thread's workflow. The thread will run the `start()` method and enter into a perpetual loop. The flow will be as follows:

- Initially the queue will be empty and the execution thread should just wait indefinitely on the condition variable to be signaled.
- When the first callback gets registered, we note how many seconds after its arrival does it need to be executed and `await()` on the condition variable for that many seconds.
- Now two things are possible at this point. No new callbacks arrive, in which case the executor thread completes waiting and polls the queue for tasks that should be executed and starts executing them.

  Or that another callback arrives, in which case the consumer thread will signal the condition variable `newCallbackArrived` to wake up the execution thread and have it re-evaluate the duration it can sleep for before the earliest callback becomes due.

This flow is caputed in the code below:

```
1  private long findSleepDuration() {
2      long currentTime = System.currentTimeMillis();
3      return q.peek().executeAt - currentTime;
4  }
5
6  public void start() throws InterruptedException {
7      long sleepFor = 0;
8
9      while (true) {
10         // Lock the critical section
11         lock.lock();
12
13         // if no item in the queue, wait indefinitely for one to arrive
14         while (q.size() == 0) {
15             newCallbackArrived.await();
16         }
17
18         // Loop till all callbacks have been executed
19         while (q.size() != 0) {
20
21             // find the minimum time execution thread should
22             // sleep for before the next callback becomes due
23             sleepFor = findSleepDuration();
24
25             // If the callback is due break from loop and start
```

Executor thread's workflow

The working of the above snippet is explained below

- Initially, the queue is empty and the executor thread will simply `await()` indefinitely on

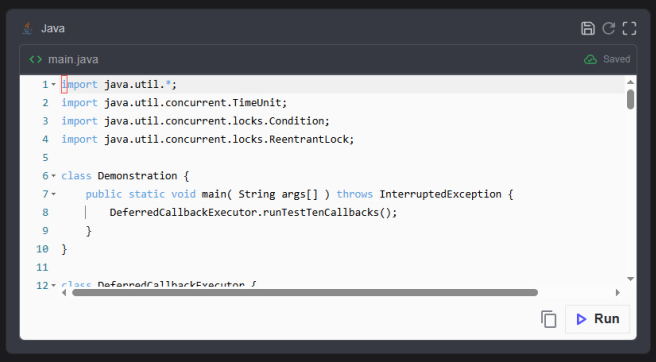
the condition `newCallbackArrived` to be signaled. Note we wrap the waiting in a while loop to cater for spurious wakeups.

- If the queue is not empty, say if the executor thread is created later than the consumer threads, then the executor will fall into the second while loop and either wait for the callback to become due or if one is already due break out of the while loop and execute the due callback.
- For all other happy path cases, adding a callback to the queue will always signal the awaiting executor thread to wake up and recalculate the time it needs to sleep before the next callback is ready to be executed.
- Note that both the `await()` calls are properly enclosed by while loops to cater for spurious wakeups. In the second while loop, if a spurious wakeup happens, the executor thread recalculates the sleep time, find it to be greater than zero and goes back to sleeping until a callback becomes due.

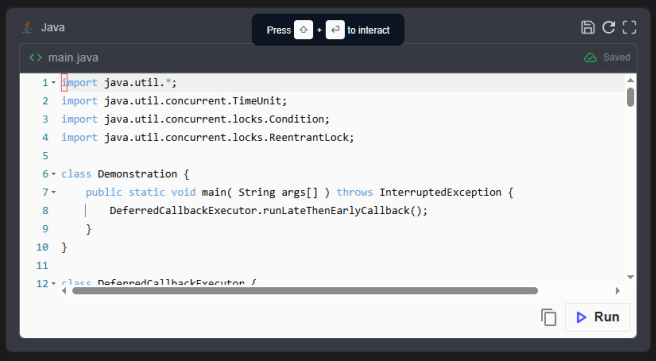
Complete Code

The complete code with the test case appears below. We insert ten callbacks sequentially waiting randomly between insertions. The output shows the epoch seconds at which the callback was expected to be executed and the actual time at which it got executed. Both of these values, for all the callbacks, should be the same or differ very slightly to account for the minuscule time it for the executor thread to wake up and execute the callback.

The code includes a test case where ten callbacks are executed. Since the code runs in the browser, it might timeout before printing the complete output.



Here's another test-case, which first submits a callback that should get executed after eight seconds. Three seconds later another call back is submitted which should be executed after only one second. The callback being submitted later should execute first. The test run would timeout if run in the browser since the callback service is a perpetual thread but from the output you can observe the callback submitted second execute first.



Back lesson
... continued
Mark As Completed
Next
Implementing Semaphore