EN.601.620
Parallel Computing for Data Science

Activity 4

Submission by **Rishabh Nanawati**

Implementation of BlockingQueue.java

```
// Implementation of a subset of the java.util.concurrent.BlockingQueue interface
public class BlockingQueue {

    // queue and dequeue string data -- not objects -- makes it easier to read
    private String [] queue;

    // queue metadata
    private int limit = 10;
    private int head = 0;
    private int qlen = 0;

    // Create an array of strings as the queue
    public BlockingQueue(int limit){
        this.limit = limit;
        this.queue = new String [limit];
    }

    public synchronized void put(String item)
    throws InterruptedException  {

        // variable for slot item goes in
        int slot;

        // wait and don't add if the queue is full
        while (qlen == limit) {
            wait();
          }

        // get slot and update head and length
        slot = head;
        head = (head + 1) % limit;
        qlen++;

        // notify takers if this is the first item in queue
        if (qlen == 1) {
```

```
            notifyAll();
          }

        // add the item
        this.queue[slot] = item;
    }

    public synchronized String take()
    throws InterruptedException {

        // slot to be taken and deleted
        int tail;

        // don't take from an empty queue
        while (qlen == 0) {
            wait();
          }

        //get slot
        tail = (head + limit - qlen) % limit;

        // if taking from a full queue, notify putters
        if (qlen == limit) {
            notifyAll();
          }

        // update queue length
        qlen--;

        // take the item and dereference pointer for garbage collection
        String ret_obj = this.queue[tail];
        queue[tail]=null;

        // return item
        return ret_obj;
    }
}
```

Question 1

When many producer threads are waiting on a full queue, what happens when a thread `takes` a String and calls `notifyAll()` ? Why is this thread safe?

Answer

When multiple producer threads are waiting on a full queue, they are all blocked in the `put()` method on the `wait()` call.

When a consumer thread calls `take()` and removes an item, making the queue non-full, it calls `notifyAll()`. This notifies all waiting threads to wake up and re-check their condition.

Only one of the waiting producers will be able to proceed, since the `put()` method is synchronized. That producer will re-check the condition, see that the queue is no longer full, and continue to add its item.

The other producers will wake up, re-check the condition, and go back to waiting since the queue is still full after the first producer added its item.

This ensures that only one producer at a time can add to the queue, so multiple threads accessing and modifying the shared state of head, tail, and qlen is safe. The synchronization and wait/notify allows threads to access the shared state in a coordinated way.

Question 2

Read the documentation for sycnhronized methods. Do your synchronized methods synchronize on the object or the class? Why is this the right scope? (Consider that there may be multiple queues in an application.)

Answer

The synchronized methods in the BlockingQueue class synchronize on the specific BlockingQueue object, not the class.

If we synchronized on the class instead, it would effectively make all BlockingQueue operations synchronized globally, even operations on different instances. This would lead to unnecessary blocking and contention.

By synchronizing on the object instance, we restrict the synchronization to only when necessary - when multiple threads access the same queue instance. This allows maximum parallelism while providing thread-safety on each instance.

Question 3

One might want the `String` copy operations to proceed in parallel because they are expensive (see below). This is not thread safe. Why? Give an example.

> To allow `String` copies to occur in parallel one would:

> - Remove the `synchronized` from the function definitions.
> - Implement a synchronized block within each function that makes accesses to `head` and `qlen` mutually exclusive.
> - Have only the following lines outside of mutual exclusion sections.
>
> ```
> // in put()
> this.queue[slot] = item;
>
> // in take()
> String ret_obj = this.queue[tail];
> queue[tail]=null;
> ```

Answer

Removing the `synchronized` keyword and using synchronized blocks instead would allow the `String` copy operations to proceed in parallel.

However, this introduces a thread safety issue due to the order of operations. Here is a scenario that could happen:

- Thread 1 calls `put()`, sees that `qlen < limit`, and proceeds to copy the `String` to the queue array.
- Before Thread 1 increments `qlen`, Thread 2 calls `take()` and sees `qlen` is the same value.
- Thread 2 computes `tail`, copies and returns the `String`, and decrements `qlen`.
- Now Thread 1 increments `qlen`.

The issue is that Thread 2 was able to take an item before Thread 1 finished putting it into the queue!

This violates the atomicity of the put and take operations - another thread should not be able to observe and take the item until after the put operation completes fully.

By making the entire `put()` and `take()` methods synchronized, it ensures the operations are atomic and thread-safe. Other threads cannot interleave operations in between.