Report: Problem 3

소프트웨어학부 20204898 박소은

# (i) BlockingQueue

## (i)-a. Explain about **BlockingQueue** and **ArrayBlockingQueue**

BlockingQueue is a thread-safe queue interface. This means that in a multi-threaded environment, attempting to insert an element into a full queue/attempting to take an element from an empty queue will block the thread. The blocked thread remains blocked until an element is taken/inserted. BlockingQueue supports 4 methods:

1. Throw exception: It throws an exception if add(), remove() or element() is not possible.
2. Special value: It returns a special value if offer(), poll() or peek() is not possible.
3. Blocks: It blocks if put() or take() is not possible.
4. Times out: It waits for the given timeout if offer() or poll() is not possible.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1. Throw Exception | 2. Special Value | 3. Blocks | 4. Times Out |
| Insert | add(o) | offer(o) | **put(o)** | offer(o, timeout, timeunit) |
| Remove | remove(o) | poll() | **take()** | poll(timeout, timeunit) |
| Examine | element() | peek() |  |  |

ArrayBlockingQueue is a class which implemented the BlockingQueue interface. It is a limited sized array which stores elements in FIFO order.

## (i)-b. Create an example of multithreaded JAVA code including **put()** and **take()** methods.

import java.util.concurrent.ArrayBlockingQueue;  
import java.util.concurrent.BlockingQueue;  
  
public class ex1 {  
 public static void main(String[] args) {  
 BlockingQueue queue = new ArrayBlockingQueue(2);  
  
 ProducerThread producer = new ProducerThread(queue);  
 producer.start();  
  
 ConsumerThread consumer = new ConsumerThread(queue);  
 consumer.start();  
 }  
  
 static class ConsumerThread extends Thread {  
 BlockingQueue queue;  
  
 ConsumerThread(BlockingQueue queue) { this.queue = queue; }  
  
 public void run() {  
 try {  
 Thread.*sleep*(500);  
 System.*out*.println("\n< ConsumerThread 시작 >");  
  
 for(int i=0; i<4; i++) {  
 System.*out*.println("\n[TAKE] Current Queue: " + queue.toString());  
 System.*out*.println("[TAKE] Try to take an element");  
 System.*out*.println("[TAKE] Took an element: " + queue.take());  
 System.*out*.println("[TAKE] Queue remaining capacity: " + queue.remainingCapacity());  
 }  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
 }  
 }  
  
 static class ProducerThread extends Thread {  
 BlockingQueue queue;  
  
 ProducerThread(BlockingQueue queue) { this.queue = queue; }  
  
 public void run() {  
 System.*out*.println("< ProducerThread 시작 >");  
 try {  
 System.*out*.println("\n[PUT] Queue remaining capacity: "+ queue.remainingCapacity());  
 System.*out*.println("[PUT] Try to insert 1");  
 queue.put(1);  
 System.*out*.println("[PUT] Current Queue: " + queue.toString());  
  
 System.*out*.println("\n[PUT] Queue remaining capacity: "+ queue.remainingCapacity());  
 System.*out*.println("[PUT] Try to insert 2");  
 queue.put(2);  
 System.*out*.println("[PUT] Current Queue: " + queue.toString());  
  
 System.*out*.println("\n[PUT] Queue remaining capacity: "+ queue.remainingCapacity());  
 System.*out*.println("[PUT] Try to insert 3");  
 queue.put(3);  
 System.*out*.println("[PUT] Current Queue: " + queue.toString());  
  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
 }  
 }  
}

텍스트이(가) 표시된 사진

자동 생성된 설명*Queue capacity: 2*

1. A producer thread wants to put 3 elements in the queue.
2. When it tries to put third element, it is blocked.
3. After 500ms, Consumer thread starts.
4. When consumer thread took first element, the producer thread wakes up, and put third element.
5. The consumer thread continues taking the elements out. When it tries to take an element out of the queue, it is blocked.

# (ii) ReadWriteLock

## (ii)-a. Explain about ReadWriteLock.

The ReadWriteLock interface allows multiple threads to read shared resources, while writing can only be done by one thread at a time.

* Reading: If the ReadWriteLock is not locked for writing, multiple threads can lock for reading.
* Writing: If no threads are reading or writing (if the ReadWriteLock is not locked), one thread can lock for writing.

ReentrantReadWriteLock class implemented the ReadWriteLock interface.

## (ii)-b. Create an example of multithreaded JAVA code including **lock()**, **unlock()**, **readLock()**, **writeLock()** methods.

import java.util.concurrent.locks.ReadWriteLock;  
import java.util.concurrent.locks.ReentrantReadWriteLock;  
  
public class ex2 {  
 public static void main(String[] args) {  
 ReadWriteLock lock = new ReentrantReadWriteLock();  
  
 ReadThread readThread1 = new ReadThread(lock, 1);  
 readThread1.start();  
  
 ReadThread readThread2 = new ReadThread(lock, 2);  
 readThread2.start();  
  
 WriteThread writeThread3 = new WriteThread(lock, 3);  
 writeThread3.start();  
  
 WriteThread writeThread4 = new WriteThread(lock, 4);  
 writeThread4.start();  
 }  
  
 static class WriteThread extends Thread {  
 ReadWriteLock lock;  
 int id;  
  
 WriteThread(ReadWriteLock lock, int id) {  
 this.lock = lock;  
 this.id = id;  
 }  
  
 @Override  
 public void run() {  
 // writeLock().lock()  
 System.*out*.println("Thread #" +id + ": Try to lock writeLock()");  
 lock.writeLock().lock();  
 System.*out*.println("Thread #" +id + ": Locked writeLock()");  
  
 // writeLock().unlock()  
 System.*out*.println("Thread #" +id + ": Try to unlock writeLock()");  
 lock.writeLock().unlock();  
 }  
 }  
  
 static class ReadThread extends Thread {  
 ReadWriteLock lock;  
 int id;  
  
 ReadThread(ReadWriteLock lock, int id) {  
 this.lock = lock;  
 this.id = id;  
 }  
  
 @Override  
 public void run() {  
 // readLock().lock()  
 System.*out*.println("Thread #" +id + ": Try to lock readLock()");  
 lock.readLock().lock();  
 System.*out*.println("Thread #" +id + ": Locked readLock()");  
  
 // Thread sleeps  
 try {  
 Thread.*sleep*(5000);  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
  
 // readLock().unlock()  
 System.*out*.println("Thread #" +id + ": Try to unlock readLock()");  
 lock.readLock().unlock();  
 }  
 }  
}

* Example execution

4 threads try to lock: thread1, 2 lock readLock, thread3, 4 lock writeLock.

ReadLocks sleep for 5s after locking the readLock, therefore readThreads can lock at the same time.

If the writeThread lock first, the readThreads and the other writeThread have to wait until the writeThread unlocks. On the other hand, if the readThread lock first, the other readThread can also lock the readLock, and the writeLocks have to wait until the readThreads unlock the lock.

* Execution result 1

1. 텍스트이(가) 표시된 사진

   자동 생성된 설명writeThread(Thread3) locked the writeLock first, therefore the other writeThread(Thread4) and readThreads waits until Thread3 unlock the lock.
2. After Thread3 unlock, readThreads(Thread1, Thread2) locked. The remaining writeThread(Thread4) waits until the readThreads unlock.

* 텍스트, 명판이(가) 표시된 사진

  자동 생성된 설명Execution result 2

1. This time, readThreads locked first.
2. As the writeThreads cannot lock simultaneously, after one writeThread is finished, the next thread locks.

# (iii) AtomicInteger

## (iii)-a. Explain about **AtomicInteger**.

You can use an integer variable which provides concurrency in multi-threaded environment. Multiple threads can read and write the AtomicInteger object.

## (iii)-b. Create an example of multithreaded JAVA code including **get(), set(), getAndAdd(), addAndGet()** methods.

# (iv) CyclicBarrier

## (iv)-a. Explain about **CyclicBarrier**

## (iv)-b. Create an example of multithreaded JAVA code including **await()** method.