REPORT

#Assignment 02

Class	Multicore Computing 01class
Prof.	Bong-Soo Sohn
ID	20206319
Dept	Software dept
Name	Ga-Yeun Kim

(I)-a. Explain the interface BlockingQueue and class ArrayBlockingQueue

ⅰ. BlockingQueue

A queue that causes a thread to wait in certain situations, such as when an attempt to remove an element from the queue is empty or when an attempt to insert an element is full and the thread calling deque()/enque() is waited. A thread attempting to remove an element from an empty queue waits until another thread puts an element into the queue. Conversely, a thread attempting to insert an element from a full queue waits until another thread removes the element from the queue or cleans the queue to free up space in the queue.

The core methods are put() and take(). Put() waits until there is space to add a value when the queue is full, and take() waits until a value arrives when the queue is empty.

BlockingQueue is non-nullable, can be limited in capacity, supports the collection interface, and implements thread safety.

BlockingQueue supports Throw exception, Special value, Blocks, and Time out. Throwing exception means that an exception is raised when the corresponding method cannot be used immediately. Special value is a method that returns null or boolean value. Blocks means that the current thread waits indefinitely until the corresponding task is performed when the corresponding method is not used immediately. Time out is to block for a given amount of time.

ⅱ. ArrayBlockingQueue

BlockingQueue implemented as an Array, used when used in a multi-threaded environment. It's concurrency-safe, so you don't need the synchronized statement. When creating a queue, set the size and internally use an array to store elements. If the queue is empty when trying to retrieve an element, wait until the element is added. If the queue is full when adding an element, an exception may occur or wait for a certain amount of time.

put() is a method to add an element after waiting until there is free space.

offer(timeout) is a method for adding an element after waiting for a certain amount of time.

take() is a method to take an element.

poll(timeout) is a method to wait for a certain period of time until an element is returned.

(I)-b. Create multithread and include example of put() and take() methods of ex1.java and execution results

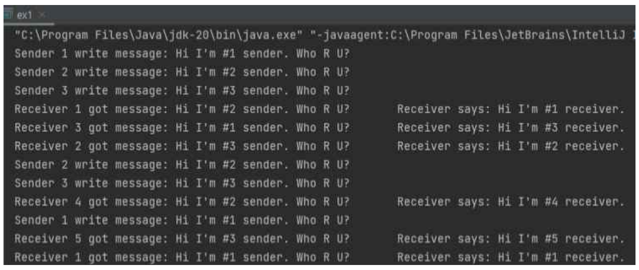
#ex1.java code

```
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.ArrayBlockingQueue;

class Sender implements Runnable {
```

```
private BlockingQueue<String> queue;
private int num;
public Sender(BlockingQueue<String> queue, int num) {
this.queue = queue;
this.num = num;
}

public void run() {
while (true) {
try {
String message = "Hi I'm #" + num + " sender. Who R U?";
queue.put(message);
System.out.println("Sender " + num + " write message: " + message);
Thread.sleep(1000);
} catch (InterruptedException e) {
e.printStackTrace();
}
}
}
}

class Receiver implements Runnable {
private BlockingQueue<String> queue;
private int num;
public Receiver(BlockingQueue<String> queue, int num) {
this.queue = queue;
this.num = num;
}

public void run() {
while (true) {
try {
String message = queue.take();
System.out.println("Receiver " + num + " got message: " + message + "        Receiver says:
Hi I'm #" + num + " receiver.");
Thread.sleep(2000);
} catch (InterruptedException e) {
e.printStackTrace();
}
}
}
}

public class ex1 {
public static void main(String[] args) {
BlockingQueue<String> queue = new ArrayBlockingQueue<String>(10);

for (int i = 1; i <= 3; i++) {
Sender s = new Sender(queue, i);
Thread t = new Thread(s);
t.start();
}

for (int i = 1; i <= 5; i++) {
Receiver r = new Receiver(queue, i);
Thread t = new Thread(r);
t.start();
}
}
}
```

Sender writes a message and puts it into the BlockingQueue, and Receiver gets the message from the BlockingQueue and processes it. BlockingQueue puts data into the queue using the put() method and takes data from the queue using the take() method. Put() method blocks when the queue is full and take() method blocks when the queue is empty.

#execution result



```
ex1
"C:\Program Files\Java\jdk-20\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ
Sender 1 write message: Hi I'm #1 sender. Who R U?
Sender 2 write message: Hi I'm #2 sender. Who R U?
Sender 3 write message: Hi I'm #3 sender. Who R U?
Receiver 1 got message: Hi I'm #2 sender. Who R U?      Receiver says: Hi I'm #1 receiver.
Receiver 3 got message: Hi I'm #1 sender. Who R U?      Receiver says: Hi I'm #3 receiver.
Receiver 2 got message: Hi I'm #3 sender. Who R U?      Receiver says: Hi I'm #2 receiver.
Sender 2 write message: Hi I'm #2 sender. Who R U?
Sender 3 write message: Hi I'm #3 sender. Who R U?
Receiver 4 got message: Hi I'm #2 sender. Who R U?      Receiver says: Hi I'm #4 receiver.
Sender 1 write message: Hi I'm #1 sender. Who R U?
Receiver 5 got message: Hi I'm #3 sender. Who R U?      Receiver says: Hi I'm #5 receiver.
Receiver 1 got message: Hi I'm #1 sender. Who R U?      Receiver says: Hi I'm #1 receiver.
```

(Ⅱ)-a. Explain the class ReadWriteLock

If a resource is being read, other threads cannot write to that resource. Since the content consistency is not broken when read operations occur consecutively, synchronization problems do not occur even if the same resource is read at the same time.

Read-lock is a state that occurs when a thread reads a resource. It passes when a read request comes in and blocks when a write request comes in. If there are pending write requests, it also blocks other read requests so that write operations do not starve.

Write-lock is a state that occurs when a resource is being written and blocks all resource requests. This is because resource consistency cannot be guaranteed until the write operation completes.

In the two locks, lock() method is used for locking, and unlock() method is used for unlocking.

(Ⅱ)-b. Create multithread and include example of lock(), unlock(), readLock() and writeLock() methods of ReadWriteLock (ex2.java) and execution results

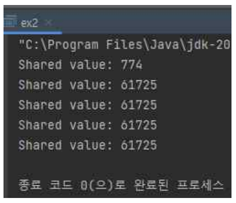
#ex2.java code

```
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

public class ex2 {
private int value = 0;
private ReadWriteLock rw = new ReentrantReadWriteLock();

public void jump() {
rw.writeLock().lock();
try {
value++;
} finally {
rw.writeLock().unlock();
}
}

public int getVariable() {
rw.readLock().lock();
try {
return value;
} finally {
rw.readLock().unlock();
}
```

```java
}

public static void main(String[] args) {
ex2 ex = new ex2();

for (int i = 1; i <= 5; i++) {
Thread t = new Thread(() -> {
for (int j = 0; j < 12345; j++) {
ex.jump();
}
});
t.start();
}

Thread t = new Thread(() -> {
for (int i = 0; i < 5; i++) {
System.out.println("Shared value: " + ex.getVariable());
try {
Thread.sleep(1000);
} catch (InterruptedException e) {
e.printStackTrace();
}
}
});
t.start();
}
}
```

Create a ReadWriteLock object named rw using the ReentrantReadWriteLock class. The jump() method obtains a write lock and increments a shared variable, and the getVariable() method obtains a read lock and returns a shared variable. This allows multiple threads to read concurrently, and while writes are in progress, other threads will wait for reads. The 5 threads increment the shared variable 12345 times, and the other threads output the value of the shared variable every second. The output values should all be the same.

#execution result



```
"C:\Program Files\Java\jdk-20
Shared value: 774
Shared value: 61725
Shared value: 61725
Shared value: 61725
Shared value: 61725

종료 코드 0(으)로 완료된 프로세스
```

<u>(Ⅲ)-a. Explain the class AtomicInteger</u>

It is an Interger that guarantees atomicity, and is a class to solve synchronization problems in a multithread environment without a synchronized statement. It is solved by Compare And Swap (CAS) method. CAS is a method of assigning a new value only when the value it had before changing the value of a variable is the same as the value you expect.

Set(new) is a method that changes a value to new, and get() is a method that reads a value.

GetAndAdd(new) is a method that returns the current value and adds a new value to the current value.

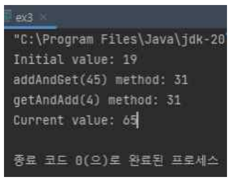
AddAndGet(new) is a method that adds a new value to the current value and returns the result.

(Ⅲ)-b. Create and include example of get(), set(), getAndAdd(), and addAndGet() methods of AtomicInteger (ex3.java) and execution results

#ex3.java code

```java
import java.util.concurrent.atomic.AtomicInteger;

public class ex3 {
private AtomicInteger atom = new AtomicInteger();

public void set(int value) {
atom.set(value);
}

public int get() {
return atom.get();
}

public int addAndGet(int value) {
return atom.addAndGet(value);
}

public int getAndAdd(int value) {
return atom.getAndAdd(value);
}

public static void main(String[] args) {
ex3 ex = new ex3();

ex.set(19);
System.out.println("Initial value: " + ex.get());

int r1 = ex.addAndGet(12);
System.out.println("addAndGet(45) method: " + r1);

int r2 = ex.getAndAdd(34);
System.out.println("getAndAdd(4) method: " + r2);
System.out.println("Current value: " + ex.get());
}
}
```

#execution result



```
"C:\Program Files\Java\jdk-20
Initial value: 19
addAndGet(45) method: 31
getAndAdd(4) method: 31
Current value: 65

종료 코드 0(으)로 완료된 프로세스
```

(Ⅳ)-a. Explain the class CyclicBarrier

It is a class that allows you to wait at a desired point inside a thread that runs concurrently, that is, a barrier, and release the waiting of all waiting threads by calling the await() method as many times as the value passed as a parameter.

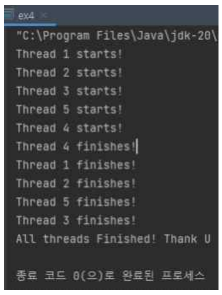
(Ⅳ)-b. Create multithread and include example of await() methods of CyclicBarrier (ex4.java) and execution results

#ex4.java code

```java
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

public class ex4 {
private static final int THREAD_NUM = 5;
private static final CyclicBarrier bar = new CyclicBarrier(THREAD_NUM, new Runnable() {
@Override
public void run() {
System.out.println("All threads Finished! Thank U");
}
});

public static void main(String[] args) {
for (int i = 1; i <= THREAD_NUM; i++) {
Thread t = new Thread(new DoThread(i));
t.start();
}
}

private static class DoThread implements Runnable {
private final int id;

public DoThread(int id) {
this.id = id;
}

public void run() {
System.out.println("Thread " + id + " starts!");
try {
Thread.sleep(1000 + (long) (Math.random() * 2000));
System.out.println("Thread " + id + " finishes!");
bar.await();
} catch (InterruptedException | BrokenBarrierException e) {
e.printStackTrace();
}
}
}
}
```

This is the code implemented so that 5 threads do the work at the same time. The bar variable is an object of the CyclicBarrier class, which calls the await() method to wait for all threads to complete their work. When all threads have completed their work, the Runnable object passed to CyclicBarrier's constructor is executed and outputs the message "All threads Finished! Thank U".

#execution result



```
"C:\Program Files\Java\jdk-20\
Thread 1 starts!
Thread 2 starts!
Thread 3 starts!
Thread 5 starts!
Thread 4 starts!
Thread 4 finishes!
Thread 1 finishes!
Thread 2 finishes!
Thread 5 finishes!
Thread 3 finishes!
All threads Finished! Thank U

종료 코드 0(으)로 완료된 프로세스
```