[Problem 3]

20185999 임태규

1. BlockingQueue & ArrayBlockingQueue

   A. Explanation

Blocking Queue is data structure class which used in multi-thread environment. Blocking Queue is used for data processing and synchronizing in Producer – consumer pattern. Producing thread keep prodcing new Object and push it into Blocking Queue, and Cousuming thread keep takes object out from the queue.

Blocking Queue methods :

put(Object object) : add Object into Blocking Queue(if there is no place in queue, **block** Producing thread)

add(Object object) : same as put, but when full, instead of block producing thread, throws exception.

offer(Object object) : same as put, but when full, instead of block producing thread, return false.

take() : take out(==return and remove) a Object from Blocking Queue(if queue is empty, **block** Consuming thread)

peak() : get(without remove) first Object from Blocking Queue(if queue is empty, return null)

element() : same as peak, but throws Exception when empty

ArrayBlockingQueue : ArrayBlockingQueue is implement of BlockingQueue which use Array data structure. So, It is FIFO because it is a implementation of queue.

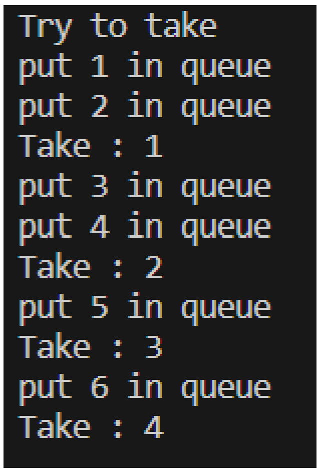
   B. Source code

```java
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.ArrayBlockingQueue;
public class ex1 {
    public static void main(String args[])throws InterruptedException{

        BlockingQueue<String> queue = new ArrayBlockingQueue<>(2);
        ProducingThread p_thread = new ProducingThread(queue);
        ConsumingThread c_thread = new ConsumingThread(queue);
        new Thread(c_thread).start();
        new Thread(p_thread).start();

        Thread.sleep(5000);
    }
}

class ProducingThread implements Runnable{
    protected BlockingQueue<String> queue = null;
    public ProducingThread(BlockingQueue<String> queue){
        this.queue = queue;
    }

    public void run(){
        try{
            Thread.sleep(1000);
            queue.put("1");
            System.out.println("put 1 in queue");
            queue.put("2");
            System.out.println("put 2 in queue");
            queue.put("3");
            System.out.println("put 3 in queue");
            queue.put("4");
            System.out.println("put 4 in queue");
            queue.put("5");
            System.out.println("put 5 in queue");
            queue.put("6");
            System.out.println("put 6 in queue");

        }catch(InterruptedException e){
            e.printStackTrace();
        }
    }
}

class ConsumingThread implements Runnable{
    protected BlockingQueue<String> queue = null;
    public ConsumingThread(BlockingQueue<String> queue){
        this.queue = queue;
    }

    public void run(){
        try{
            System.out.println("Try to take");
            System.out.println("Take : "+queue.take());
            System.out.println("Take : "+queue.take());
            System.out.println("Take : "+queue.take());
            System.out.println("Take : "+queue.take());
        }catch(InterruptedException e){
            e.printStackTrace();
        }
    }
}
```

C. Simple Explanation for code

first, try to take before put. it can't take because ArrayBlockingQueue is empty. when put String in ArrayBlockingQueue, it can take 1. and capacity of queue is 2, so when two object(string) putted into queue, than put another string is blocked. so put 3 is blocked until the take 1. output is little strange, I think this is because it takes time between take an object and print a line that we take an object. when we set capacity 1 and put 5 times, it work as we intended, And when we put 6 times but take 3 times, it doesn't terminated because of Lock. So the logic goes right but just printed line is little strange.

D. Output



```
Try to take
put 1 in queue
put 2 in queue
Take : 1
put 3 in queue
put 4 in queue
Take : 2
put 5 in queue
Take : 3
put 6 in queue
Take : 4
```

2. ReadWriteLock

A. Explanation

ReadWriteLock is a Locking rule that allow multiple reader can read resource compare to a typical Locking rule. General Locking rule is blocking **all access** when resource is using, whether reading or writing, but ReadWriteLock is blocking **only write access** when resource is reading, and blocking **all access** when resource is writing

ReadWriteLock methods :

readlock().lock() : lock the resorce with writelock

readlock().unlock() : unlock the readlock

writelock().lock() : lock the resorce with writelock

writelock().unlock() : unlock the writelock

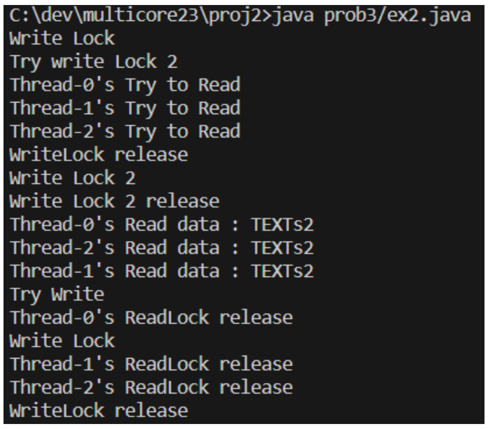
B. Source code

```java
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;
public class ex2 {
    static String data = "TEXT";
    public static void main(String[] args)throws InterruptedException{
        ReadWriteLock readWriteLock = new ReentrantReadWriteLock();
        ReadingThread r_thread1 = new ReadingThread(readWriteLock);
        ReadingThread r_thread2 = new ReadingThread(readWriteLock);
        ReadingThread r_thread3 = new ReadingThread(readWriteLock);
        WritingThread w_thread1 = new WritingThread(readWriteLock);
        WritingThread2 w_thread2 = new WritingThread2(readWriteLock);
        new Thread(r_thread1).start();
        new Thread(r_thread2).start();
        new Thread(r_thread3).start();
        new Thread(w_thread1).start();
        new Thread(w_thread2).start();

        Thread.sleep(4000);

    }

}

class ReadingThread implements Runnable{
    protected ReadWriteLock readWriteLock = null;
    public ReadingThread(ReadWriteLock readWriteLock){
        this.readWriteLock = readWriteLock;
    }

    public void run(){
        try{
            Thread.sleep(1000);
            System.out.println(Thread.currentThread().getName()+"\'s Try to Read");
            readWriteLock.readLock().lock();
            System.out.println(Thread.currentThread().getName()+"\'s Read data : " +
ex2.data);
            Thread.sleep(2000);
            readWriteLock.readLock().unlock();
            System.out.println(Thread.currentThread().getName()+"\'s ReadLock
release");
        }catch(InterruptedException e){
            e.printStackTrace();
        }
    }
}

class WritingThread implements Runnable{
    protected ReadWriteLock readWriteLock = null;
    public WritingThread(ReadWriteLock readWriteLock){
        this.readWriteLock = readWriteLock;
    }
    public void run(){
        try{
            readWriteLock.writeLock().lock();
            System.out.println("Write Lock");
            ex2.data = ex2.data.concat("s");
            Thread.sleep(2000);
            readWriteLock.writeLock().unlock();
            System.out.println("WriteLock release");
            Thread.sleep(2000);
            System.out.println("Try Write");
            readWriteLock.writeLock().lock();
            System.out.println("Write Lock");
            ex2.data = ex2.data.concat("s");
            readWriteLock.writeLock().unlock();
            System.out.println("WriteLock release");
        }catch(InterruptedException e){
            e.printStackTrace();
        }

    }
}
class WritingThread2 implements Runnable{
    protected ReadWriteLock readWriteLock = null;
    public WritingThread2(ReadWriteLock readWriteLock){
        this.readWriteLock = readWriteLock;
    }

    public void run(){
        try{
            Thread.sleep(500);
            System.out.println("Try write Lock 2");
            readWriteLock.writeLock().lock();
            System.out.println("Write Lock 2");
            ex2.data = ex2.data.concat("2");
            System.out.println("Write Lock 2 release");
            readWriteLock.writeLock().unlock();
        }catch(InterruptedException e){
            e.printStackTrace();
        }

    }
}
```

C. Simple Explanation for code

3 reading thread try to read and write when writeLocked, but they blocked until the writelock has unlocked. after that, three reading thread readlock, but they can read at the same time. we can't just write until readlock unlocked.

D. Output



```
C:\dev\multicore23\proj2>java prob3/ex2.java
Write Lock
Try write Lock 2
Thread-0's Try to Read
Thread-1's Try to Read
Thread-2's Try to Read
WriteLock release
Write Lock 2
Write Lock 2 release
Thread-0's Read data : TEXTs2
Thread-2's Read data : TEXTs2
Thread-1's Read data : TEXTs2
Try Write
Thread-0's ReadLock release
Write Lock
Thread-1's ReadLock release
Thread-2's ReadLock release
WriteLock release
```

3. AtomicInteger

   A. Explanation

AtomicInteger is a class that provide int variable atomically. It prevent multiple thread access at the same time so it helps the code runs as intended. For example, when multiple Thread access at the varible at same time, and thread runs the code that varible = variable + 10;, if it runs two times, variable goes to first value +20, but goes to first value+10. AtomicInteger provides you these methods to help.

AtomicInteger methods :

get() : get AtomicInteger variable's value.

set(int a) : set AtomicInteger variable's value to a.

getAndAdd(int a) : get AtomicInteger variable's value before add a to value.

addAndGet(int a) get AtomicInteger variable's value after add a to value.

compareAndSet(int expectedValue, int newValue) : if varible's value equals expectedValue, then set the value to newValue.

   B. Source code

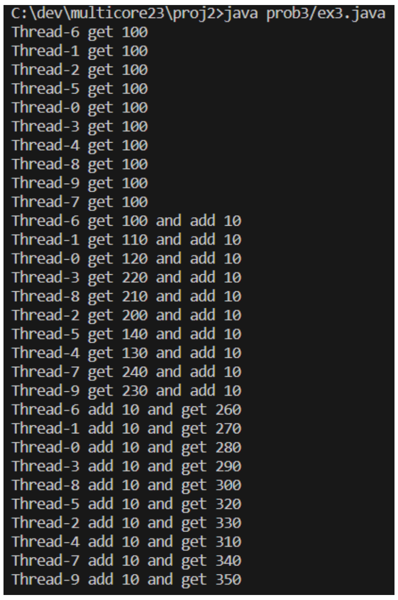
```java
import java.util.concurrent.atomic.AtomicInteger;
public class ex3 {
    public static void main(String[] args)throws InterruptedException{
        AtomicInteger atom = new AtomicInteger(100);
        for(int i=0;i<10;i++){
            Thread t = new Thread(new AtomThread(atom));
            t.start();
        }

        Thread.sleep(1500);
    }

}
```

```
class AtomThread implements Runnable{
    protected AtomicInteger atom = null;
    public AtomThread(AtomicInteger atom){
        this.atom = atom;
    }

    public void run(){
        try{
        String name = Thread.currentThread().getName();
        System.out.println(name+" get "+atom.get());
        Thread.sleep(400);
        int middle = 150;
        atom.compareAndSet(middle, 200);
        System.out.println(name+" get "+atom.getAndAdd(10)+" and add 10");
        Thread.sleep(400);
        System.out.println(name+" add 10 and get "+atom.addAndGet(10));
        }catch(InterruptedException e){
            e.printStackTrace();
        }
    }
}
```

C. Simple Explanation for code

10 thread access at the static variable at the same time, but atomic integer provides atomic access at the variable. To show that, I use compareAndSet method when value is 150 than set to 200. If it can't provide atomical properties, the final value will be less than 350 because compareAndSet executed more than once.

D. Output



```
C:\dev\multicore23\proj2>java prob3/ex3.java
Thread-6 get 100
Thread-1 get 100
Thread-2 get 100
Thread-5 get 100
Thread-0 get 100
Thread-3 get 100
Thread-4 get 100
Thread-8 get 100
Thread-9 get 100
Thread-7 get 100
Thread-6 get 100 and add 10
Thread-1 get 110 and add 10
Thread-0 get 120 and add 10
Thread-3 get 220 and add 10
Thread-8 get 210 and add 10
Thread-2 get 200 and add 10
Thread-5 get 140 and add 10
Thread-4 get 130 and add 10
Thread-7 get 240 and add 10
Thread-9 get 230 and add 10
Thread-6 add 10 and get 260
Thread-1 add 10 and get 270
Thread-0 add 10 and get 280
Thread-3 add 10 and get 290
Thread-8 add 10 and get 300
Thread-5 add 10 and get 320
Thread-2 add 10 and get 330
Thread-4 add 10 and get 310
Thread-7 add 10 and get 340
Thread-9 add 10 and get 350
```

4. CyclicBarrier

A. Explanation

CyclicBarrier is synchronization mechanism to synchronize threads at some point. It works like a halfway point. It check that number of thread reach at the point, and when one or more thread didn't reached, it makes reached thread await until other thread reach at that point.

ReadWriteLock methods :

await() : await until other Thread (CyclicBarrier parties) reach.

B. Source code

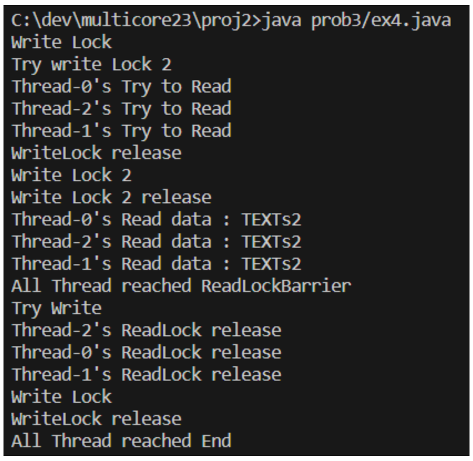
```
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;
public class ex4 {
    static String data = "TEXT";
    public static void main(String[] args){

        Runnable readLockBarrierAction = new Runnable() {
            public void run(){
                System.out.println("All Thread reached ReadLockBarrier");
            }
        };
        Runnable readEndBarrierAction = new Runnable() {
            public void run(){
                System.out.println("All Thread reached End");
            }
        };

        CyclicBarrier readLockBarrier = new CyclicBarrier(4, readLockBarrierAction);
        CyclicBarrier readEndBarrier = new CyclicBarrier(3, readEndBarrierAction);

        ReadWriteLock readWriteLock = new ReentrantReadWriteLock();
        ReadingThread r_thread1 = new
ReadingThread(readWriteLock,readLockBarrier,readEndBarrier);
        ReadingThread r_thread2 = new
ReadingThread(readWriteLock,readLockBarrier,readEndBarrier);
        ReadingThread r_thread3 = new
ReadingThread(readWriteLock,readLockBarrier,readEndBarrier);
        WritingThread w_thread1 = new WritingThread(readWriteLock,readLockBarrier);
        WritingThread2 w_thread2 = new WritingThread2(readWriteLock);
        new Thread(r_thread1).start();
        new Thread(r_thread2).start();
        new Thread(r_thread3).start();
        new Thread(w_thread1).start();
        new Thread(w_thread2).start();

    }

}
```

```java
class ReadingThread implements Runnable{
    protected ReadWriteLock readWriteLock = null;
    protected CyclicBarrier readLockBarrier = null;
    protected CyclicBarrier readEndBarrier = null;
    public ReadingThread(ReadWriteLock readWriteLock,CyclicBarrier readLockBarrier,
CyclicBarrier readEndBarrier){
        this.readWriteLock = readWriteLock;
        this.readLockBarrier = readLockBarrier;
        this.readEndBarrier = readEndBarrier;
    }

    public void run(){
        try{
            Thread.sleep(1000);
            System.out.println(Thread.currentThread().getName()+"\'s Try to Read");
            readWriteLock.readLock().lock();
            System.out.println(Thread.currentThread().getName()+"\'s Read data : " +
ex4.data);
            this.readLockBarrier.await();
            System.out.println(Thread.currentThread().getName()+"\'s ReadLock release");
            readWriteLock.readLock().unlock();
            this.readEndBarrier.await();
        }catch(InterruptedException e){
            e.printStackTrace();
        }catch(BrokenBarrierException e){
            e.printStackTrace();
        }
    }
}

class WritingThread implements Runnable{
    protected ReadWriteLock readWriteLock = null;
    protected CyclicBarrier readLockBarrier = null;
    public WritingThread(ReadWriteLock readWriteLock,CyclicBarrier readLockBarrier){
        this.readWriteLock = readWriteLock;
        this.readLockBarrier = readLockBarrier;
    }

    public void run(){
        try{
            readWriteLock.writeLock().lock();
            System.out.println("Write Lock");
            Thread.sleep(2000);
            ex4.data = ex4.data.concat("s");
            readWriteLock.writeLock().unlock();
            System.out.println("WriteLock release");
            readLockBarrier.await();
            System.out.println("Try Write");
            readWriteLock.writeLock().lock();
            System.out.println("Write Lock");
            ex4.data = ex4.data.concat("s");
            readWriteLock.writeLock().unlock();
            System.out.println("WriteLock release");
        }catch(InterruptedException e){
            e.printStackTrace();
        }catch(BrokenBarrierException e){
            e.printStackTrace();
        }

    }
}
class WritingThread2 implements Runnable{
    protected ReadWriteLock readWriteLock = null;
    public WritingThread2(ReadWriteLock readWriteLock){
        this.readWriteLock = readWriteLock;
    }

    public void run(){
        try{
            Thread.sleep(500);
            System.out.println("Try write Lock 2");
            readWriteLock.writeLock().lock();
            System.out.println("Write Lock 2");
            ex4.data = ex4.data.concat("2");
            System.out.println("Write Lock 2 release");
            readWriteLock.writeLock().unlock();
        }catch(InterruptedException e){
            e.printStackTrace();
        }

    }
}
```

C. Simple Explanation for code

use ex2.java code. We use CyclicBarrier instead of Thread.sleep() to schedule the method execution point in time to show readwriteLock Blocking. We can't replace all Thread.sleep into CyclicBarrier because it can occur deadlock.(If we place barrier.await before unlock, deadlock occur, and if we place barrier.await after unlock, it is meaningless because we can't know the output is cause from barrier or lock)

D. Output



```
C:\dev\multicore23\proj2>java prob3/ex4.java
Write Lock
Try write Lock 2
Thread-0's Try to Read
Thread-2's Try to Read
Thread-1's Try to Read
WriteLock release
Write Lock 2
Write Lock 2 release
Thread-0's Read data : TEXTs2
Thread-2's Read data : TEXTs2
Thread-1's Read data : TEXTs2
All Thread reached ReadLockBarrier
Try Write
Thread-2's ReadLock release
Thread-0's ReadLock release
Thread-1's ReadLock release
Write Lock
WriteLock release
All Thread reached End
```