멀티코어컴퓨팅(Spring, 2023)
20171248 안재형

Multicore Project #2

Report i

a. Explain the interface/class BlockingQueue and ArrayBlockingQueue with your own English sentences.

BlockingQueue is just an interface that defines what '*blocking queue*' does. Regardless of how a queue is internally implemented, BlockingQueue guarantees that a class provides all operations that *blocking queue* does. For example, BlockingQueue contains put() and take() function. A new element is put into a queue with put() function and it can be removed with take() function. As we can see the name, it should protect the values from race condition.

ArrayBlockingQueue is one of the implementation of *blocking queue*. It uses an array to manage the internal values. A number of threads can access to the queue and it provides mutual exclusion so that it can avoid race condition.

b. Create and include (in your document) your own example of multithreaded JAVA code (ex1.java) that is simple and executable. (DO NOT copy&paste) Your example code should use put() and take() methods. Also, include example execution results (i.e. output) in your document.

This is a sample code implementing Producer-Consumer problem. Producer puts elements into a queue twice faster than consumer.

Code :

```
package prob3;
import java.util.concurrent.ArrayBlockingQueue;

public class ex1 {
    static ArrayBlockingQueue<Integer> queue = new ArrayBlockingQueue<>(1);

    public static void main(String[] args) {
        Producer producer = new Producer(queue);
        Consumer consumer = new Consumer(queue);
    }
}

class Producer extends Thread {
    ArrayBlockingQueue<Integer> queue;
    Producer(ArrayBlockingQueue<Integer> queue) {
        this.queue = queue;
        this.start();
    }

    @Override
    public void run() {
        while(true) {
            try {
                sleep((int)(Math.random() * 500));
```

```
                System.out.println("Producer trying to put");
                queue.put(0);
                System.out.println("Producer put");
            } catch (InterruptedException e) {}
        }
    }
}

class Consumer extends Thread {
    ArrayBlockingQueue<Integer> queue;
    Consumer(ArrayBlockingQueue<Integer> queue) {
        this.queue = queue;
        this.start();
    }

    @Override
    public void run() {
        while(true) {
            try {
                sleep((int)(Math.random() * 1000));
                System.out.println("Consumer trying to take");
                queue.take();
                System.out.println("Consumer took");
            } catch (InterruptedException e) {}
        }
    }
}
```

Execution Output :

Consumer trying to take
Producer trying to put
Producer put
Consumer took
Producer trying to put
Producer put
Producer trying to put
Consumer trying to take
Producer put
Consumer took
Producer trying to put
Consumer trying to take
Consumer took
Producer put
Producer trying to put
Consumer trying to take
Producer put
Consumer took
Producer trying to put
Consumer trying to take
Consumer took
Producer put
Producer trying to put
Consumer trying to take
Consumer took
Producer put
Producer trying to put

Report ii

a. Do the things similar to (i)-a for the class ReadWriteLock.

ReadWriteLock is a kind of lock that provides readLock and writeLock. It is also an interface so you should make a class or just use ReentrantReadWriteLock, which is a pre-made class in java.util.concurrent. A class that inherits ReadWriteLock manages a value from race condition by readLock and writeLock. To read or write a value, a thread should have lock on readLock or writeLock. Then, the class automatically manages them by blocking and releasing each locks.

ReadWriteLock is useful when there are much more readers than writers. If a data is managed with FIFO access policy, it would be inefficient because mutual exclusion is not needed between readers. So ReadWriteLock provides mutual exclusion when a writer has an access. Also, it prevents writer to access until all readers finishes reading.

b. Do the things similar to (i)-b for lock(), unlock(), readLock() and writeLock() of ReadWriteLock. (ex2.java)

```
package prob3;

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

public class ex2 {
    static ReadWriteLock readWriteLock = new ReentrantReadWriteLock();
    static Data data = new Data();
    static int numOfReaders = 3;

    public static void main(String[] args) {
        Reader[] readers = new Reader[numOfReaders];
        for(int i=0; i<numOfReaders; i++) readers[i] = new Reader(i+1, data, readWriteLock);
        Writer writer = new Writer(1, data, readWriteLock);
    }
}

class Data {
    int data = 0;
}

class Reader extends Thread {
    int num;
    Data data;
    ReadWriteLock lock;
    Reader(int num, Data data, ReadWriteLock lock) {
        this.num = num;
        this.data = data;
        this.lock = lock;
        this.start();
    }

    @Override
    public void run() {
        while(true) {
            try {
                sleep((int)(Math.random() * 1000));
                System.out.println("Reader " + num + " try to read");
                lock.readLock().lock();
                System.out.println("Reader " + num + " : " + data.data);
                lock.readLock().unlock();
            } catch (InterruptedException e) {}
        }
    }
}

class Writer extends Thread {
    int num;
    Data data;
    ReadWriteLock lock;
```

```
    Writer(int num, Data data, ReadWriteLock lock) {
        this.num = num;
        this.data = data;
        this.lock = lock;
        this.start();
    }

    @Override
    public void run() {
        while(true) {
            try {
                sleep((int)(Math.random() * 1000));
                System.out.println("Writer " + num + " try to modify");
                lock.writeLock().lock();
                sleep((int)(Math.random() * 800));
                data.data = (int)(Math.random()*100)+1;
                System.out.println("Writer modified");
                lock.writeLock().unlock();
            } catch (InterruptedException e) {}
        }
    }
}
```

Execution Output :

```
Reader 3 try to read
Reader 3 : 0
Reader 2 try to read
Reader 2 : 0
Reader 3 try to read
Reader 3 : 0
Writer 1 try to modify
Reader 1 try to read
Reader 3 try to read
Writer modified
Reader 1 : 98
Reader 3 : 98
Reader 3 try to read
Reader 3 : 98
Reader 2 try to read
Reader 2 : 98
Reader 1 try to read
Reader 1 : 98
Writer 1 try to modify
Reader 3 try to read
Reader 2 try to read
Writer modified
Reader 3 : 54
Reader 2 : 54
Reader 1 try to read
Reader 1 : 54
Reader 3 try to read
```

Report iii

a. Do the things similar to (i)-a for the class AtomicInteger.

AtomicInteger is a variable that guarantees mutual exclusion through threads. It has functions such as get(), set(), getAndSet() and etc. It internally has a protected variable and it is accessible only through the functions. A thread can read the variable through get() and write it through set().

b. Do the things similar to (i)-b for get(), set(), getAndAdd(), and addAndGet() methods of AtomicInteger. (ex3.java)

```
package prob3;

import java.util.concurrent.atomic.AtomicInteger;

public class ex3 {
    static int numOfThreads = 3;

    public static void main(String[] args) {
        AtomicInteger variable = new AtomicInteger(0);

        NumThread[] threads = new NumThread[numOfThreads];
        for(int i=0; i<numOfThreads; i++) threads[i] = new NumThread(i, variable);
    }
}

class NumThread extends Thread {
    AtomicInteger variable;
    int threadNumber;
    NumThread(int threadNumber, AtomicInteger variable) {
        this.threadNumber = threadNumber;
        this.variable = variable;
        this.start();
    }

    @Override
    public void run() {
        while(true) {
            try {
                sleep(10);
                double rand = Math.random();
                if (rand < 0.25) {
                    System.out.println("Thread " + threadNumber + " read " + variable.get());
                } else if(rand < 0.5) {
                    System.out.println("Thread " + threadNumber + " read and add 3 " + vari-
able.getAndAdd(3));
                } else if(rand < 0.75) {
                    System.out.println("Thread " + threadNumber + " add 3 and read " + vari-
able.addAndGet(3));
                } else {
                    System.out.println("Thread " + threadNumber + " trying to write");
                    int newVariable = (int) (Math.random() * 100) + 1;
                    variable.set(newVariable);
                    System.out.println("Thread " + threadNumber + " wrote " + newVariable);
                }
            } catch (InterruptedException e) {
            }
        }
    }
}
```

Execution Output :

Thread 1 add 3 and read 9
Thread 0 add 3 and read 3
Thread 2 add 3 and read 6
Thread 1 add 3 and read 12
Thread 2 read 12
Thread 0 read 12
Thread 1 add 3 and read 15
Thread 2 add 3 and read 18
Thread 0 read and add 3 18

Because there is a gap between get() and println, the printed result is not the same order as the actual operations. But we can see that the value of the variable is shown as we expected.

Report iv

a. Do the things similar to (i)-a for the class CyclicBarrier.

CyclicBarrier is literally used as a barrier which blocks other threads until all threads reaches to a point. At first, CyclicBarrier is created with the number of threads to reach barrier. When a thread reached to the barrier, it uses barrier.wait() to block itself. The barrier counts how many threads should reached and when all threads are reached, it releases all threads.

b. Do the things similar to (i)-b for await() methods of CyclicBarrier. (ex4.java)

```
package prob3;

import java.util.concurrent.CyclicBarrier;

public class ex4 {
    static int numOfThreads = 5;

    public static void main(String[] args) {
        WaitingThread[] threads = new WaitingThread[numOfThreads];
        CyclicBarrier barrier = new CyclicBarrier(numOfThreads);
        long startTime = System.currentTimeMillis();
        for(int i=0; i<numOfThreads; i++) threads[i] = new WaitingThread(i, startTime, barrier);
    }
}

class WaitingThread extends Thread {
    int threadNum;
    long startTime;
    CyclicBarrier barrier;
    WaitingThread(int threadNum, long startTime, CyclicBarrier barrier) {
        this.threadNum = threadNum;
        this.startTime = startTime;
        this.barrier = barrier;
        this.start();
    }

    @Override
    public void run() {
        try {
            long diff;
            diff = System.currentTimeMillis() - startTime;
            System.out.println(diff + " Thread " + threadNum + " start sleeping");
            sleep((int)(Math.random() * 10000));
            diff = System.currentTimeMillis() - startTime;
            System.out.println(diff + " Thread " + threadNum + " finished sleeping & waiting");
            barrier.await();
            diff = System.currentTimeMillis() - startTime;
            System.out.println(diff + " Thread " + threadNum + " finished waiting");
        } catch (Exception e) {}
    }
}
```

Execution Output :

```
0 Thread 2 start sleeping
0 Thread 3 start sleeping
0 Thread 0 start sleeping
0 Thread 4 start sleeping
0 Thread 1 start sleeping
284 Thread 2 finished sleeping & waiting
1449 Thread 0 finished sleeping & waiting
2985 Thread 1 finished sleeping & waiting
7305 Thread 3 finished sleeping & waiting
8893 Thread 4 finished sleeping & waiting
8894 Thread 0 finished waiting
8894 Thread 4 finished waiting
8895 Thread 1 finished waiting
8895 Thread 3 finished waiting
8894 Thread 2 finished waiting
```