Problem 3 (Multicore23 Proj #2)

Study & Summarize concurrent programming Java classes

CAU CSE 20184286 LEE DONGHWA

2023.05.10

BlockingQueue

This interface defines a queue that supports blocking operations. Blocking operations are methods that is blocked or waits for certain conditions to be met. It is useful in the Producer and Consumer problem.

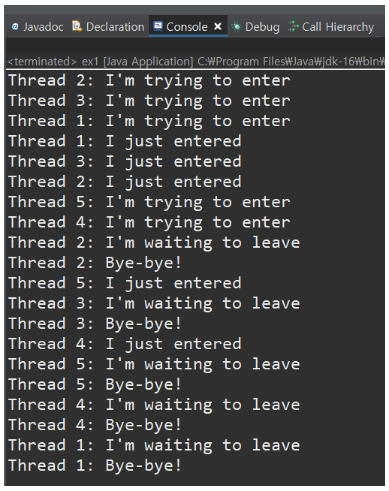
ArrayBlockingQueue

This class is a finite queue that stores elements in an array. The capacity of the queue is specified when the queue is created and cannot be changed. When the queue is full, attempts to add an element are blocked until it is removed from the queue to free up space. Similarly, if the queue is empty, attempts to remove an element by adding it are blocked until it is created.

Source code of ex1.java

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

class Ex1_Thread extends Thread {
            private BlockingQueue queue;
            public Ex1_Thread(String name, BlockingQueue q) {
                    super(name);
                    this.queue = q;
                    start();
            }

            public void run() {
                try {
                    System.out.println(getName()+": I'm trying to enter");
                    queue.put(getName());
                    System.out.println(getName()+": I just entered");
                } catch (InterruptedException e) {}

                try {
                    sleep((int)(Math.random() * 5000));
                } catch (InterruptedException e) {}

                try {
                    System.out.println(getName()+": I'm waiting to leave");
                    queue.take();
                    System.out.println(getName()+": Bye-bye!");
                } catch (InterruptedException e) {}
            }
}

public class ex1 {
        public static void main(String[] args) {
                // TODO Auto-generated method stub
                BlockingQueue queue = new ArrayBlockingQueue<String>(3);
            for (int i = 1; i <= 5; i++) {
                Ex1_Thread t = new Ex1_Thread("Thread "+i, queue);
            }
        }
}
```

Output of ex1.java



```
<terminated> ex1 [Java Application] C:₩Program Files₩Java₩jdk-16₩bin₩
Thread 2: I'm trying to enter
Thread 3: I'm trying to enter
Thread 1: I'm trying to enter
Thread 1: I just entered
Thread 3: I just entered
Thread 2: I just entered
Thread 5: I'm trying to enter
Thread 4: I'm trying to enter
Thread 2: I'm waiting to leave
Thread 2: Bye-bye!
Thread 5: I just entered
Thread 3: I'm waiting to leave
Thread 3: Bye-bye!
Thread 4: I just entered
Thread 5: I'm waiting to leave
Thread 5: Bye-bye!
Thread 4: I'm waiting to leave
Thread 4: Bye-bye!
Thread 1: I'm waiting to leave
Thread 1: Bye-bye!
```

ReadWriteLock

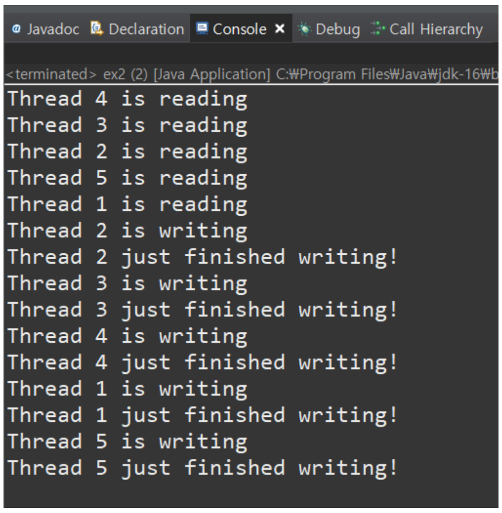
This support 'multiple threads to read' or 'write only one thread' at a time. This minimizes contention between threads in read behavior and ensures data consistency in write behavior.

Source code of ex2.java

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

class Ex2_Thread extends Thread {
        ReadWriteLock readWriteLock;

          public Ex2_Thread(String name, ReadWriteLock l) {
                  super(name);
                  this.readWriteLock = l;
                  start();
          }

          public void run() {
              try {
                  readWriteLock.readLock().lock();
                  System.out.println(getName()+" is reading");
                  sleep((int)(Math.random() * 3000));
                  readWriteLock.readLock().unlock();
```

```
                    readWriteLock.writeLock().lock();
                    System.out.println(getName()+" is writing");
                    sleep((int)(Math.random() * 3000));
                    System.out.println(getName()+" just finished writing!");
                    readWriteLock.writeLock().unlock();
               } catch (InterruptedException e) {
                    e.printStackTrace();
               }
           }
}

public class ex2 {
         public static void main(String[] args) {
                  ReadWriteLock readWriteLock = new ReentrantReadWriteLock();

                  for(int i = 1; i <= 5; i++) {
                           Ex2_Thread t = new Ex2_Thread("Thread "+i , readWriteLock);
                  }
         }
}
```

Output of ex2.java



```
<terminated> ex2 (2) [Java Application] C:₩Program Files₩Java₩jdk-16₩b
Thread 4 is reading
Thread 3 is reading
Thread 2 is reading
Thread 5 is reading
Thread 1 is reading
Thread 2 is writing
Thread 2 just finished writing!
Thread 3 is writing
Thread 3 just finished writing!
Thread 4 is writing
Thread 4 just finished writing!
Thread 1 is writing
Thread 1 just finished writing!
Thread 5 is writing
Thread 5 just finished writing!
```

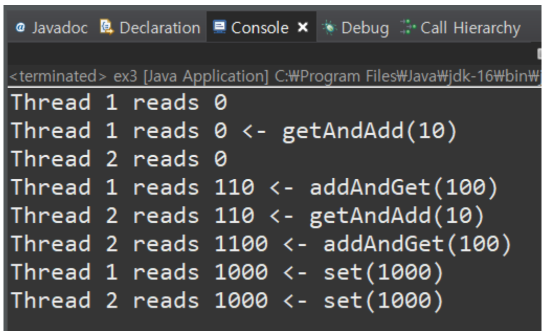
AtomicInteger

Integer variable that supports atomic operation. Atomic operations are performed in one operation, ensuring that variables are updated correctly when other threads access them simultaneously.

Source code of ex3.java

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

public class ex3 {

    private static AtomicInteger counter = new AtomicInteger(0);

    static class MyRunnable implements Runnable {

        private int now;
        private int prev;
        private int index;
        public MyRunnable(int i) {
                        this.index = i;
                }

                public void run() {
            now = counter.get();
            System.out.println("Thread " + index + " reads " + now);
            prev = counter.getAndAdd(10);
            System.out.println("Thread " + index + " reads " + prev + " <- getAndAdd(10)");
            now = counter.addAndGet(100);
            System.out.println("Thread " + index + " reads " + now + " <- addAndGet(100)");
            counter.set(1000);
            System.out.println("Thread " + index + " reads " + counter + " <- set(1000)");


        }
    }

    public static void main(String[] args) {
        for (int i = 1; i <= 2; i++) {
          Thread t = new Thread(new MyRunnable(i));
          t.start();
             }
    }
}
```

Output of ex3.java



```
<terminated> ex3 [Java Application] C:₩Program Files₩Java₩jdk-16₩bin₩
Thread 1 reads 0
Thread 1 reads 0 <- getAndAdd(10)
Thread 2 reads 0
Thread 1 reads 110 <- addAndGet(100)
Thread 2 reads 110 <- getAndAdd(10)
Thread 2 reads 1100 <- addAndGet(100)
Thread 1 reads 1000 <- set(1000)
Thread 2 reads 1000 <- set(1000)
```

CyclicBarrier

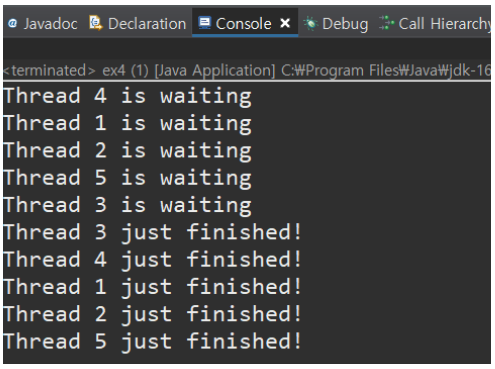
It provides the ability to wait for all threads to reach when multiple threads are waiting for each other, and then synchronize all threads to run at the same time.

Source code of ex4.java

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

class Ex4_Thread extends Thread {
           CyclicBarrier Barrier;
           public Ex4_Thread(String name, CyclicBarrier b) {
                    super(name);
                    this.Barrier = b;
                    start();
           }
```

```
            public void run() {
                try {
                        sleep((int)(Math.random() * 5000));
                        System.out.println(getName()+" is waiting");
                        Barrier.await();
                } catch (InterruptedException | BrokenBarrierException e) {} finally {
                        System.out.println(getName()+" just finished!" );
                }
            }
}

public class ex4 {
            public static void main(String[] args) {
                    CyclicBarrier newBarrier = new CyclicBarrier(5);
                    for (int i=1; i<= 5; i++) {
                    Ex4_Thread t = new Ex4_Thread("Thread "+ i, newBarrier);
                    }
            }
}
```

Output of ex4.java



```
<terminated> ex4 (1) [Java Application] C:₩Program Files₩Java₩jdk-16
Thread 4 is waiting
Thread 1 is waiting
Thread 2 is waiting
Thread 5 is waiting
Thread 3 is waiting
Thread 3 just finished!
Thread 4 just finished!
Thread 1 just finished!
Thread 2 just finished!
Thread 5 just finished!
```