Multicore Computing Project2

- Problem 3 -

과목명	멀티코어컴퓨팅
제출일	2023.05.10.
학 번	20183901
학 과	소프트웨어학부
이 름	김상민

목 차

1. ex1: BlockingQueue and ArrayBlockingQueue

(a). Explanation

Blocking queue is a queue which can be used in multi-threaded environment. ArrayBlockingQueue is a class which implements the Blocking queue interface. ArrayBlockingQueue has a fixed size space. When an element tries to enter the full queue, it waits for the queue to have space.

(b). Source code

```
1  package ex1;
2  import java.util.concurrent.ArrayBlockingQueue;
4  public class ex1 {
5
6      public static void main(String[] args) {
7          BlockingQueue queue = new ArrayBlockingQueue<Integer>(3);
8          Thread producer = new Producer(queue);
9          Thread consumer = new Consumer(queue);
10         producer.start();
11         consumer.start();
12     }
13
14     static class Producer extends Thread{
15         private BlockingQueue queue;
16
17         public Producer(BlockingQueue q) {
18             queue = q;
19         }
20         public void run() {
21             for(int i = 0; i<10;i++)
22             {
23                 try {
24                     System.out.println("Producer "+i+": Trying to put "+i);
25                     queue.put(i);
26                     System.out.println("Producer "+i+": Put "+i+" done. Queue size: "+queue.size());
27                     sleep((int)(Math.random()*1000));
28                 }catch(Exception e) {
29
30                 }
31
32             }
33         }
34
35     }
36
37     static class Consumer extends Thread{
38         private BlockingQueue queue;
39
40         public Consumer(BlockingQueue q) {
41             queue = q;
42         }
43         public void run() {
44             for(int i = 0; i<10;i++)
45             {
46                 try {
47                     System.out.println("Consumer "+i+":                    Trying to take");
48                     int x = (int)queue.take();
49                     System.out.println("Consumer "+i+":                    Take "+x+" done. Queue size: "+queue.size());
50                     sleep((int)(Math.random()*1500));
51                 }catch(Exception e) {
52
53                 }
54
55             }
56         }
57
58     }
59
60 }
```

(c). Execution result

```
Producer 0: Trying to put 0
Consumer 0:                                     Trying to take
Producer 0: Put 0 done. Queue size: 1
Consumer 0:                                     Take 0 done. Queue size: 0
Producer 1: Trying to put 1
Producer 1: Put 1 done. Queue size: 1
Producer 2: Trying to put 2
Producer 2: Put 2 done. Queue size: 2
Consumer 1:                                     Trying to take
Consumer 1:                                     Take 1 done. Queue size: 1
Producer 3: Trying to put 3
Producer 3: Put 3 done. Queue size: 2
Consumer 2:                                     Trying to take
Consumer 2:                                     Take 2 done. Queue size: 1
Producer 4: Trying to put 4
Producer 4: Put 4 done. Queue size: 2
Consumer 3:                                     Trying to take
Consumer 3:                                     Take 3 done. Queue size: 1
Producer 5: Trying to put 5
Producer 5: Put 5 done. Queue size: 2
Producer 6: Trying to put 6
Producer 6: Put 6 done. Queue size: 3
Consumer 4:                                     Trying to take
Consumer 4:                                     Take 4 done. Queue size: 2
Producer 7: Trying to put 7
Producer 7: Put 7 done. Queue size: 3
Producer 8: Trying to put 8
Consumer 5:                                     Trying to take
Consumer 5:                                     Take 5 done. Queue size: 2
Producer 8: Put 8 done. Queue size: 3
Consumer 6:                                     Trying to take
Consumer 6:                                     Take 6 done. Queue size: 2
Producer 9: Trying to put 9
Producer 9: Put 9 done. Queue size: 3
Consumer 7:                                     Trying to take
Consumer 7:                                     Take 7 done. Queue size: 2
Consumer 8:                                     Trying to take
Consumer 8:                                     Take 8 done. Queue size: 1
Consumer 9:                                     Trying to take
Consumer 9:                                     Take 9 done. Queue size: 0
```

This is an example of a basic producer and consumer problem. BlockingQueue's size is set to 3. Producer puts numbers from 0 to 9 into queue and consumer takes number from queue 10 times. Consumer is set a little faster than producer.

2. ex2: ReadWriteLock

(a). Explanation

ReadWriteLock is an lock which is suitable for controlling read, write problem. It has 2 locks: read lock and write lock. When readLock is locked, multiple readers can read the shared value, but writers can not access. When writeLock is locked, only the writer which acquired the lock can access and the others can not.

(b). Source code

```java
package ex2;

import java.util.concurrent.locks.ReadWriteLock;

class SharedVar{
    private ReadWriteLock lock;
    private int shared_var = 0;

    public SharedVar(ReadWriteLock l) {
        lock = l;
    }

    public void read(int id) {
        try {
            lock.readLock().lock();
            System.out.println("Reader "+id+": read locked");
            System.out.println("Reader "+id+": read: "+shared_var);
        } catch (Exception e) {

        }finally{
            lock.readLock().unlock();
            System.out.println("Reader "+id+": read unlocked");
        }
    }

    public void write(int id) {
        try{
            lock.writeLock().lock();
            System.out.println("Writer "+id+": "+"          write locked");
            shared_var += 1;
            System.out.println("Writer "+id+": "+"          write: "+shared_var);
        }catch(Exception e) {

        }finally {
            lock.writeLock().unlock();
            System.out.println("Writer "+id+": "+"          write unlocked");
        }
    }
}

public class ex2 {

    public static void main(String[] args) {
        ReadWriteLock lock = new ReentrantReadWriteLock();
        SharedVar var = new SharedVar(lock);

        Reader[] reader = new Reader[2];
        Writer[] writer = new Writer[2];
        for(int i = 0; i<2 ; i++) {
            reader[i] = new Reader(var, i);
            writer[i] = new Writer(var, i);
            reader[i].start();
            writer[i].start();
        }
        try {
            for(int i = 0; i<2 ; i++) {
                reader[i].join();
                writer[i].join();
            }
        }catch(Exception e) {

        }


    }
    static class Reader extends Thread{
        SharedVar var;
        int id;

        public Reader(SharedVar v, int i) {
            var = v;
            id = i;
        }
        public void run(){
            for(int i = 0; i<10; i++) {
                try {
                    var.read(id);
                }catch(Exception e) {

                }
            }
        }
    }
    static class Writer extends Thread{
        SharedVar var;
        int id;

        public Writer(SharedVar v, int i) {
            var = v;
            id = i;
        }
        public void run(){
            for(int i = 0; i<10; i++) {
                try {
                    var.write(id);
                }catch(Exception e) {

                }
            }
        }
    }


}
```

(c). Execution result

```
Reader 0: read locked
Reader 1: read locked
Reader 0: read: 0
Reader 1: read: 0
Reader 0: read unlocked
Reader 1: read unlocked
Writer 0:                       write locked
Writer 0:                       write: 1
Writer 0:                       write unlocked
Writer 0:                       write locked
Writer 0:                       write: 2
Writer 0:                       write unlocked
Writer 1:                       write locked
Writer 1:                       write: 3
Writer 1:                       write unlocked
Reader 1: read locked
Reader 0: read locked
Reader 1: read: 3
Reader 0: read: 3
Reader 1: read unlocked
Reader 0: read unlocked
Writer 0:                       write locked
Writer 0:                       write: 4
Writer 0:                       write unlocked
Writer 1:                       write locked
Writer 1:                       write: 5
Writer 1:                       write unlocked
Reader 1: read locked
Reader 0: read locked
Reader 1: read: 5
Reader 0: read: 5
Reader 1: read unlocked
Writer 0:                       write locked
Writer 0:                       write: 6
Writer 0:                       write unlocked
Writer 0:                       write locked
Writer 0:                       write: 7
Reader 0: read unlocked
Writer 0:                       write unlocked
Writer 0:                       write locked
Writer 0:                       write: 8
Writer 0:                       write unlocked
Writer 1:                       write locked
Writer 1:                       write: 9
Writer 1:                       write unlocked
Reader 1: read locked
Reader 1: read: 9
Reader 0: read locked
Reader 0: read: 9
```

```
Reader 0: read unlocked
Reader 1: read unlocked
Writer 0:                       write locked
Writer 0:                       write: 10
Writer 0:                       write unlocked
Writer 0:                       write locked
Writer 0:                       write: 11
Writer 0:                       write unlocked
Writer 1:                       write locked
Writer 1:                       write: 12
Writer 1:                       write unlocked
Reader 0: read locked
Reader 1: read locked
Reader 1: read: 12
Reader 1: read unlocked
Reader 0: read: 12
Reader 0: read unlocked
Writer 0:                       write locked
Writer 0:                       write: 13
Writer 0:                       write unlocked
Writer 0:                       write locked
Writer 0:                       write: 14
Writer 0:                       write unlocked
Writer 1:                       write locked
Writer 1:                       write: 15
Writer 1:                       write unlocked
Writer 1:                       write locked
Writer 1:                       write: 16
Writer 1:                       write unlocked
Reader 1: read locked
Reader 0: read locked
Reader 1: read: 16
Reader 0: read: 16
Reader 1: read unlocked
Reader 0: read unlocked
Writer 1:                       write locked
Writer 1:                       write: 17
Writer 1:                       write unlocked
Reader 1: read locked
Reader 0: read locked
Reader 1: read: 17
Reader 0: read: 17
Reader 1: read unlocked
Reader 0: read unlocked
Writer 1:                       write locked
Writer 1:                       write: 18
Writer 1:                       write unlocked
Reader 0: read locked
Reader 1: read locked
```

```
Reader 0: read: 18
Reader 1: read: 18
Reader 0: read unlocked
Writer 1:                       write locked
Reader 1: read unlocked
Writer 1:                       write: 19
Writer 1:                       write unlocked
Writer 1:                       write locked
Writer 1:                       write: 20
Writer 1:                       write unlocked
Reader 0: read locked
Reader 0: read: 20
Reader 0: read unlocked
Reader 0: read locked
Reader 0: read: 20
Reader 0: read unlocked
Reader 1: read locked
Reader 1: read: 20
Reader 1: read unlocked
Reader 1: read locked
Reader 1: read: 20
Reader 1: read unlocked
```

This is a example of a ReadWriteLock. There are 2 readers and 2 writers and they are run simultaneously. Writers add 1 by 1 into shared_var 10 times each and readers print the value of shared_var. When read is locked, you can see that multiple readers can access(read: 17) the shared var but when write is locked, only one writer can access the shared var.

3. ex3: AtomicInteger

(a). Explanation

Atomic integer is an integer which is suitable for multi-threaded environment. When threads try to access the atomic integer, they access by various methods such as incrementAndGet(), getAndIncrement(), compareAndSet(). All calculations are atomic, so it allows safe sharing between threads.

(b). Source code

```
 1  package ex3;
 2  import java.util.concurrent.atomic.*;
 3
 4  public class ex3 {
 5
 6      public static void main(String[] args) {
 7          AtomicInteger atomic_int = new AtomicInteger(0);
 8          int inc_num1 = 10;
 9          int inc_num2 = 10;
10          Thread ang = new AddAndGetInc(atomic_int, inc_num1);
11          Thread gna = new GetAndAddInc(atomic_int, inc_num2);
12          ang.start();
13          gna.start();
14          try {
15              ang.join();
16              gna.join();
17          }catch(Exception e) {
18
19          }finally {
20              System.out.println("-------------------");
21              System.out.println("final result: "+atomic_int);
22              System.out.println("-------------------");
23          }
24      }
25
26      static class AddAndGetInc extends Thread{
27          private AtomicInteger atomic_int;
28          private int num;
29          public AddAndGetInc(AtomicInteger i, int n) {
30              atomic_int = i;
31              num = n;
32          }
33          public void run() {
34              for(int i = 0; i<num; i++) {
35                  System.out.println("Trying to add&get: "+atomic_int);
36                  int x = atomic_int.addAndGet(1);
37                  System.out.println("add&get done: "+x);
38                  try {
39                      sleep((int)(Math.random()*2000));
40                  }catch(InterruptedException e) {
41
42                  }
43
44              }
45          }
46      }
47      static class GetAndAddInc extends Thread{
48          private AtomicInteger atomic_int;
49          private int num;
50          public GetAndAddInc(AtomicInteger i, int n) {
51              atomic_int = i;
52              num = n;
53          }
54          public void run() {
55              for(int i = 0; i<num; i++) {
56                  System.out.println("                    Trying to get&add: "+atomic_int);
57                  int x = atomic_int.getAndAdd(1);
58                  System.out.println("                    get&add done: "+x);
```

```
59                  try {
60                      sleep((int)(Math.random()*2000));
61                  }catch(InterruptedException e) {
62
63                  }
64              }
65          }
66      }
67
68  }
69
```

(c). Execution result

```
Trying to add&get: 0
add&get done: 1
                                    Trying to get&add: 0
                                    get&add done: 1
Trying to add&get: 2
add&get done: 3
                                    Trying to get&add: 3
                                    get&add done: 3
Trying to add&get: 4
add&get done: 5
                                    Trying to get&add: 5
                                    get&add done: 5
Trying to add&get: 6
add&get done: 7
                                    Trying to get&add: 7
                                    get&add done: 7
Trying to add&get: 8
add&get done: 9
                                    Trying to get&add: 9
                                    get&add done: 9
                                    Trying to get&add: 10
                                    get&add done: 10
Trying to add&get: 11
add&get done: 12
Trying to add&get: 12
add&get done: 13
Trying to add&get: 13
add&get done: 14
                                    Trying to get&add: 14
                                    get&add done: 14
                                    Trying to get&add: 15
                                    get&add done: 15
Trying to add&get: 16
add&get done: 17
                                    Trying to get&add: 17
                                    get&add done: 17
                                    Trying to get&add: 18
                                    get&add done: 18
Trying to add&get: 19
add&get done: 20
------------------
final result: 20
------------------
```

This is a example of an atomic integer. There are 2 types of thread that increments the atomic integer. One 'adds and gets' and the other 'gets and adds'. 2 threads increment the atomic integer 10 times each. In add&get, when add&get is done, it prints the added value. But in get&add, it prints the atomic integer first and then adds.

4. ex4: CyclicBarrier

(a). Explanation

Cyclic barrier is a kind of a barrier where threads must wait when they reach the barrier point. When cyclic barrier is initialized, an integer is given. When the number of threads that are waiting reaches the given integer, program executes the cyclic barrier action code.

(b). Source code

```java
package ex4;

import java.util.concurrent.CyclicBarrier;

public class ex4 {

    public static void main(String[] args) {
        try {
            CyclicBarrier barrier = new CyclicBarrier(2, () -> {
                System.out.println("********************************************************************************");
                System.out.println("            All threads passed the barrier");
            });
            Counter[] counter = new Counter[2];
            for (int i = 0; i < 2; i++) {
                counter[i] = new Counter(barrier, 10, i);
                counter[i].start();
            }
        }catch(Exception e) {

        }
    }

    static class Counter extends Thread{
        private CyclicBarrier barrier;
        private int num;
        private int id;
        public Counter(CyclicBarrier b, int n, int i) {
            barrier = b;
            num = n;
            id = i;
        }
        public void run() {
            for(int i = 0; i<num; i++) {
                if(id==0) {
                    System.out.println("Thread " +id+" - count: "+i);
                }else {
                    System.out.println("                        Thread " +id+" - count: "+i);
                }
                try {
                    sleep((int)(Math.random()*2000));
                }catch(Exception e) {

                }

            }
            try {
                if(id==0) {
                    System.out.println("Thread " +id+" - reached the barrier");
                }else {
                    System.out.println("                        Thread "+id+" - reached the barrier");
                }
                barrier.await();
            }catch(Exception e) {

            }
        }
    }
}
```

(c). Execution result

```
Thread 0 - count: 0
                                            Thread 1 - count: 0
Thread 0 - count: 1
Thread 0 - count: 2
                                            Thread 1 - count: 1
                                            Thread 1 - count: 2
Thread 0 - count: 3
                                            Thread 1 - count: 3
Thread 0 - count: 4
                                            Thread 1 - count: 4
                                            Thread 1 - count: 5
Thread 0 - count: 5
Thread 0 - count: 6
                                            Thread 1 - count: 6
Thread 0 - count: 7
                                            Thread 1 - count: 7
Thread 0 - count: 8
                                            Thread 1 - count: 8
Thread 0 - count: 9
                                            Thread 1 - count: 9
Thread 0 - reached the barrier
                                            Thread 1 - reached the barrier
********************************************************************************
                        All threads passed the barrier
```

This is a example of a cyclic barrier. There are 2 threads that counts 10 numbers. When threads end counting, each meets the cyclic barrier and awaits for all other threads to reach this point. When the number of threads awaiting reaches the number we set at the beginning(2), program executes cyclic barrier action code.