Concurrent and Distributed Systems

Concurrent Problem Solving Lab Assignment

January 19, 2020

Student: Marcu Andrei Cristian

Computers and Information Technology

CEN 3.2 A

3rd Year

Problems statements

Problem 1 (50p)
Implement a program to solve the producer-consumer problem using:
a) coarse synchronization
b) fine synchronization.
The code must be tested with at least 4 producers and the number of consumers will be given by the number of available virtual CPUs.

Problem 2 (50p)
Develop and implement a program to multiply 2 matrices of size 1024x1024 using divide-et-impera.
The multiplication of matrices will be realized in a concurrent way using executors and the number of threads will be given by the number of available virtual CPUs.

Both problems are implemented using Java.

Problem1: a) For the coarse implementation I've used only one lock each time a producer/consumer calls the function that modifies the queue. This method is not as efficient as the fine synchronization but it's way safer than that one. We also have two lock conditions "notFull, notEmpty" which are made in order to notify the consumer when the queue is not empty and the producer when the queue is not full

```java
    Random rand = new Random();
    int i;
    ConcurrentLinkedQueue<Integer> queue; // queue where producer inserts numbers
    Lock lock; // lock used for synchronization
    Condition notFull; // lock conditions used for notfull/notempty cases
    Condition notEmpty;
    PCQ(){
        queue = new ConcurrentLinkedQueue<Integer>(); // thread safe queue
        lock =  new ReentrantLock();
        notFull = lock.newCondition();
        notEmpty = lock.newCondition();
    }
    public void prod(int id)
    {
        for(int it = 0 ; it < 10 ; it++)
        {
            lock.lock();
            try {
                while(queue.size() == queueCapacity)
                    try {
                        notFull.await(); // producer waits until the queue is not full
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                i = rand.nextInt(100);
                System.out.println("Producer " + id + " added " + i);
                queue.add(i);// we add a random number from 0 to 100
                notEmpty.signalAll(); // we inform the consumer that a new item was produced and queue can't be empty
            }
            finally {
                lock.unlock();
            }
        }
    }

    public void cons(int id)
    {
        for(int it = 0; it < 10; it++)
        {
            lock.lock();
            try {
                while(queue.isEmpty())
                    try {
                        notEmpty.await(); // consumer waits until the queue is not empty
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                System.out.println("Consumer " + id + " removed " + queue.remove());
                notFull.signalAll(); // we inform the producer that an item was removed and the queue can't be full
            }
            finally {
                lock.unlock();
            }
        }
```

Problem1: b) For the fine implementation I've used an array of locks for each elements in queue. This method is more efficient than the coarse implementation but it's harder to get it to work right. I've made an array of conditions too. Both solutions are using the number of available CPUs as threads.

PCQ.java

```java
    ConcurrentLinkedQueue<Integer> queue; // queue where producer inserts numbers
    ReentrantLock lock[] = new ReentrantLock[queueCapacity + 1]; // lock used for synchronization
    Condition notFull[] = new Condition[queueCapacity + 1]; // lock conditions used for notfull/notempty cases
    Condition notEmpty[] = new Condition[queueCapacity + 1];
    PCQ(){
        queue = new ConcurrentLinkedQueue<Integer>(); // thread safe queue
        for(int i = 0 ; i <= this.queueCapacity; i++) {
            lock[i] =  new ReentrantLock();
            notFull[i] = lock[i].newCondition();
            notEmpty[i] = lock[i].newCondition();
        }
    }
    public void prod(int id)
    {
        int lockNumber = queue.size();
        lock[lockNumber].lock();
        try {
            while(queue.size() == queueCapacity) {
                try {
                    notFull[lockNumber].await();// producer waits until the queue is not full
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            i = rand.nextInt(100);
            System.out.println("Producer " + id + " added " + i);
            queue.add(i);// we add a random number from 0 to 100
            notEmpty[lockNumber].signal();
        }
        finally {
            lock[lockNumber].unlock();
        }
    }

    public void cons(int id)
    {

        int lockNumber = queue.size();

        lock[lockNumber].lock();
        try {
            while(queue.isEmpty()) {
                try {
                    notEmpty[lockNumber].await(); // consumer waits until the queue is not empty
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            System.out.println("Consumer " + id + " removed " + queue.remove());
            notFull[lockNumber].signal();
        }
        finally {
            lock[lockNumber].unlock();
        }
```

Experiments and results for matrix multiplication

Both producer and consumer will run forever. I've printed as much as I could from the execution.

1. Example a)

```
Producer 0 added 14
Producer 1 added 73
Producer 1 added 81
Producer 1 added 24
Producer 1 added 46
Consumer 0 removed 14
Consumer 0 removed 73
Consumer 0 removed 81
Consumer 0 removed 24
Consumer 0 removed 46
Producer 1 added 80
Producer 1 added 78
Producer 1 added 17
Producer 1 added 89
Producer 1 added 85
Consumer 2 removed 80
Consumer 2 removed 78
Consumer 2 removed 17
Consumer 2 removed 89
Consumer 2 removed 85
Producer 1 added 36
Producer 2 added 20
Producer 2 added 15
Producer 2 added 84
Producer 2 added 45
Consumer 2 removed 36
Consumer 2 removed 20
Consumer 2 removed 15
Consumer 2 removed 84
Consumer 2 removed 45
Producer 2 added 99
Producer 2 added 48
Producer 2 added 21
Producer 2 added 86
Producer 2 added 33
Consumer 5 removed 99
Consumer 5 removed 48
Consumer 5 removed 21
Consumer 5 removed 86
Consumer 5 removed 33
Producer 2 added 41
Consumer 10 removed 41
Producer 0 added 90
Producer 0 added 82
Producer 0 added 56
Producer 0 added 17
Producer 0 added 58
Consumer 11 removed 90
Consumer 11 removed 82
Consumer 11 removed 56
Consumer 11 removed 17
Consumer 11 removed 58
Producer 0 added 96
```

2. Example b)

```
Producer 0 added 42
Producer 2 added 98
Producer 1 added 55
Producer 3 added 41
Consumer 0 removed 42
Consumer 1 removed 98
Consumer 2 removed 55
Consumer 3 removed 41
Producer 0 added 44
Producer 3 added 61
Producer 2 added 98
Producer 1 added 81
Consumer 4 removed 44
Consumer 5 removed 61
Consumer 6 removed 98
Consumer 7 removed 81
Producer 0 added 39
Consumer 5 removed 39
Producer 3 added 37
Producer 1 added 14
Producer 2 added 95
Consumer 4 removed 37
Consumer 8 removed 14
Consumer 9 removed 95
Producer 1 added 48
Producer 0 added 60
Consumer 9 removed 48
Producer 3 added 66
Consumer 5 removed 60
Consumer 4 removed 66
Producer 2 added 0
Consumer 8 removed 0
Producer 1 added 13
Consumer 4 removed 13
Producer 3 added 91
Producer 0 added 67
Consumer 5 removed 91
Producer 2 added 22
Consumer 7 removed 67
Consumer 6 removed 22
Producer 3 added 26
Consumer 5 removed 26
Producer 0 added 74
Consumer 7 removed 74
Producer 1 added 59
Consumer 4 removed 59
Producer 2 added 97
Consumer 0 removed 97
Producer 3 added 77
Producer 0 added 30
Consumer 5 removed 77
Producer 1 added 45
Producer 2 added 43
```

Conclusions Producer Consumer problem

The coarse solution for the problem is more ineffective than the fine solution but it's easier to debug the code and prevent upcoming bugs. The fine method uses more locks and each of them can cause the code go wrong but it is efficient.

Problem2: In order to solve the divide-et-impera multiplication matrix I've took the add matrix solution presented on the 10th course by our teacher. I've modified the code in order to support multiplication too. In the end the code presented there proved to be way less ineffective than the classical math matrix multiply. In my code I've got both solutions in order to compare them. I've tried to use an exact number of CPUs like it's said in the problem statement but it doesn't work for this solution. I've let it with cached thread pool which is more efficient. I have added two new classes, MathMultiplication which makes the classic math matrix multiplication and MatrixGenerator which generates random matrices. I have also added one method on the Matrix class which prints the matrix

Matrix.java | MathMultiplication.java | MultiplyTask.java

```java
import java.util.concurrent.Future;

public class MultiplyTask implements Runnable {
    Matrix a, b, c;

    public MultiplyTask(Matrix a, Matrix b, Matrix c) {
        this.a = a;
        this.b = b;
        this.c = c;
    }

    public void run() {
        try {
            int n = a.getDim();
            if (n == 1) {
                c.updateCell(0, 0, (a.get(0,0) * b.get(0,0)));
            } else if (n == 2) {

                for (int i = 0; i < 2; ++i) {
                    for (int j = 0; j < 2; ++j) {
                        // paralel solving for the sum
                        c.updateCell(i, j, a.get(i, 0) * b.get(0, j) + a.get(i, 1) * b.get(1, j));
                    }
                }

            } else {
                Matrix[][] aa = a.split(), bb = b.split(), cc = c.split();
                Future<?>[][] future1 = (Future<?>[][]) new Future[2][2];
                Future<?>[][] future2 = (Future<?>[][]) new Future[2][2];
                for (int i = 0; i < 2; i++) {
                    for (int j = 0; j < 2; j++) {
                        // concurrent solving for the 8xmultiplications
                            future1[i][j] = MatrixTask.exec.submit(new MultiplyTask(aa[i][0], bb[0][j], cc[i][j]));
                            future2[i][j] = MatrixTask.exec.submit(new MultiplyTask(aa[i][1], bb[1][j], cc[i][j]));
                            future1[i][j].get();
                            future2[i][j].get();
                    }
                }
            }
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

Experiments and results for producer-consumer

Matrices we want to multiply

```
5 7 9 0 8 8 0 9
6 4 8 4 3 4 5 7
5 2 1 8 8 4 6 1
3 1 8 3 9 7 0 1
0 5 4 7 4 2 2 0
9 4 6 4 3 1 2 9
1 5 0 1 8 4 0 4
4 6 0 7 2 7 5 3

4 8 6 7 0 2 0 6
2 7 6 9 3 0 2 2
9 5 8 9 4 8 4 5
8 8 3 5 9 3 3 1
4 8 5 4 4 5 8 3
8 4 9 0 0 5 4 6
6 4 9 0 9 2 1 8
7 0 6 4 3 7 4 8
```

Result

```
Math multiplication:
274 230 310 247 116 225 182 233
259 208 274 210 158 182 125 217
204 227 210 138 171 121 122 151
209 195 211 157 101 166 147 140
146 159 139 132 128 87 89 77
225 198 234 221 129 165 110 199
114 131 139 105 68 93 109 97
199 194 217 137 143 105 94 155

Math multiplication duration in seconds : 0

Divide et impera with threads multipliction
274 230 310 247 116 225 182 233
259 208 274 210 158 182 125 217
204 227 210 138 171 121 122 151
209 195 211 157 101 166 147 140
146 159 139 132 128 87 89 77
225 198 234 221 129 165 110 199
114 131 139 105 68 93 109 97
199 194 217 137 143 105 94 155

Divite et impera multiplication duration in seconds : 0
```

Conclusions Matrix multiplication problem

I've used the solution implemented in course, which proved being very slow. In the end the code will compute the result matrix correct but the execution time for the 1024x1024 matrix is somewhere between 2-3 minutes. For the rest it goes in maximum 1 minute. I will present examples in the Experiments and results folder

References

https://blog.georgovassilis.com/2014/02/04/on-coarse-vs-fine-grained-synchronization/
https://en.wikipedia.org/wiki/Strassen$_{a}lgorithm$
$www.overleaf.com$