Assignment 3

Spiros-Daniel Mavroidakos - 260689391
Jastaj Virdee - 260689027

December 3, 2018

Contents

1 Question 1

1.1

From the question, we know the following.

- We are reading L words from cache
- Cache Line is 4 words

L' is the number of words that can be stored in the cache given a maximum word seperation of stride s. A stride can be a maximum size of $\left\lfloor \frac{L}{2} \right\rfloor$. We also know that the cache can have several cache lines. We will denote the amount of cache lines by k.
Knowing this, we can come up with the following equation for L'.

$$L' = \left\lceil \frac{4k}{\left\lfloor \frac{L}{2} \right\rfloor} \right\rceil \tag{1}$$

Furthermore, t0 shows the cache access latency as it does take some time to read from the cache. GIven that L ¡ L', the stride size will either be 0 or a size that fits within the case. Therefore, all the time delay for access comes from reading the case.

1.2

Given that L ¿ L', the stride will be large enough that we cannot only read from the cache. The words will be too seperated to read only from cache. However, at t1, we have peaked in the maximum read time. This implies that the stride is larger than the actual cache size. That is why it is a constant line. t1 will therefore be showing the memory access latency since we are no longer reading from the cache. t1 might also take into account the time to read the cache as we need to generate a cache miss to read from other storage.

1.3

Part 1 is when L ¡ L' so as stated in section 1.1, all of the data of the array will fit into the cache and will therefore be accessed at the time that it takes to read the cache. This is why it is the lowest of the 2 curves.
Part 2 is when L ¿ L' however, the stride has not become so big that we cannot use the cache in a meaningful way. Throughout part 2, the stride is continuously increasing which is why the curve is increasing as you must read from cache and then main memory. At the end of part 2 (the peak value where it becomes constant) is where part 3 begins and when the stride has become so large that the stride takes up all of the cache and you will always incur a cache miss and have to then go to main memory which is why it is a peak. Part 2 has a mix between some values are in cache and some are not while part 3 is where everything is not in the cache.

1.4

2 Question 2

2.1

All of the source code can be found in the appendix in the specified sections:

- The implementation for a node can be found in section A.1.
- The contains method implementation can be found in section A.2

2.2

All of the source code can be found in the appendix in the specified sections:

- The implementation for the contains Test can be found in section A.3
- The output of the test can be found in Section A.4.

The implementation of the contains method was doen using a hand-over-hand locking style. This assured that you would lock the previous node and the current node. This is needed to ensure that when you are selecting the next node, that you do not get preempted by another operation that came after you. For example, if you only lock the current node, then it is possible that you unlock the current node and then you set your next node but you have not locked this node. Now, the thread gets suspended and another one comes along and removes the node you currently have selected. You resume and now you are on an orphan node. This is not safe and is why you need to lock both the previous and current. You release the previous and then lock the current node's next node. Like this, the current node blocks anyone from overtaking you.

The test implemented here if several threads tried to check for the existance of the same node. From the output, you can clearly see that hand-over-hand locking is occuring due to the order of locks and release. Furthermore, the locks are working as expected since no lock is being locked without an unlock occuring prior. The output clearly demonstrates a previous and current node being locked. Then the previous node is unlocked and the current node's next node is locked. It can also be noticed that the released previous node is then being locked by another waiting node. Since the add and remove methods that have been seen in class are implemented using the same hand-over-hand method, those operations cannot interfere with the contains method as they are designed to be thread safe.

A Question 2 Code

A.1 Node Implementation

```
1 import java.util.concurrent.locks.ReentrantLock;
2
3 public class Node<T>{
4
5     public ReentrantLock nodeLock = new ReentrantLock();
6
7     public T item;
8     public int key;
```

```
    public volatile Node next;

    public void lock()
    {
        this.nodeLock.lock();
        System.out.println("Node with key = " + key + " locked");
    }

    public void unlock()
    {
        System.out.println("Node with key = " + key + " unlocked");
        this.nodeLock.unlock();
    }

}
```

A.2 Contains Implementation

```
import java.util.LinkedList;
import java.util.*;
import java.util.concurrent.*;
import java.util.concurrent.locks.ReentrantLock;

public class FineGrainedContains<T>
{
    public Node head = new Node();

    public boolean contains(T item)
    {
        Node pred = head;
        pred.lock();
        Node curr = pred.next;
        curr.lock();
        int key = item.hashCode();

        try
        {

            while (curr.key <= key)
            {
                // check if they are equal
                if (curr.item == item)
                {
                    return true;
                }

                // Hand over hand locking
                pred.unlock();
                pred = curr;
                if(curr.next != null)
                {
                    curr = curr.next;
                    curr.lock();
                }
                else
                {
                    curr = null;
```

```
                    break;
                }
            }

        }
        finally
        {
            pred.unlock();
            if (curr != null)
            {
                curr.unlock();
            }
        }

        // nothing was found
        return false;
    }

}
```

A.3 Contains Test

```

public class ContainsTest
{
    public static FineGrainedContains test = new FineGrainedContains<Object>();

    public static void main(String args[]){
        Node c = new Node();
        c.item = 3;
        c.key = c.item.hashCode();


        Node b = new Node();
        b.item = 2;
        b.key = b.item.hashCode();
        b.next = c;


        Node a = new Node();
        a.item = 1;
        a.key = a.item.hashCode();
        a.next = b;

        test.head.next = a;

        Tester[] testers = new Tester[5];

        for(int i = 0; i < testers.length; i++)
        {
            testers[i] = new Tester();
            testers[i].start();
        }
    }

    private static class Tester extends Thread {
        public void run() {
```

```
36              System.out.println("Thread with id = " + this.getId() + " Searching for
                    value = 3.");
37
38              boolean found = test.contains(3);
39
40              if (found)
41              {
42                  System.out.println("Thread with id = " + this.getId() + " found value"
                        );
43              }
44              else
45              {
46                  System.out.println("Thread with id = " + this.getId() + " did not find
                        value");
47              }
48          }
49      }
50
51  }
```

A.4 Contains Test Output

```
1   Done execution
2   Thread with id = 15 Searching for value = 3.
3   Thread with id = 14 Searching for value = 3.
4   Thread with id = 12 Searching for value = 3.
5   Thread with id = 13 Searching for value = 3.
6   Thread with id = 16 Searching for value = 3.
7   Node with key = 0 locked
8   Node with key = 1 locked
9   Node with key = 0 unlocked
10  Node with key = 2 locked
11  Node with key = 0 locked
12  Node with key = 1 unlocked
13  Node with key = 3 locked
14  Node with key = 1 locked
15  Node with key = 2 unlocked
16  Node with key = 0 unlocked
17  Node with key = 3 unlocked
18  Node with key = 0 locked
19  Node with key = 2 locked
20  Node with key = 1 unlocked
21  Node with key = 3 locked
22  Node with key = 2 unlocked
23  Node with key = 3 unlocked
24  Node with key = 1 locked
25  Thread with id = 15 found value
26  Node with key = 0 unlocked
27  Node with key = 2 locked
28  Node with key = 0 locked
29  Node with key = 1 unlocked
30  Thread with id = 12 found value
31  Node with key = 1 locked
32  Node with key = 3 locked
33  Node with key = 0 unlocked
34  Node with key = 2 unlocked
35  Node with key = 0 locked
```

```
Node with key = 2 locked
Node with key = 3 unlocked
Node with key = 1 unlocked
Thread with id = 16 found value
Node with key = 1 locked
Node with key = 3 locked
Node with key = 0 unlocked
Node with key = 2 unlocked
Node with key = 3 unlocked
Node with key = 2 locked
Thread with id = 13 found value
Node with key = 1 unlocked
Node with key = 3 locked
Node with key = 2 unlocked
Node with key = 3 unlocked
Thread with id = 14 found value
```