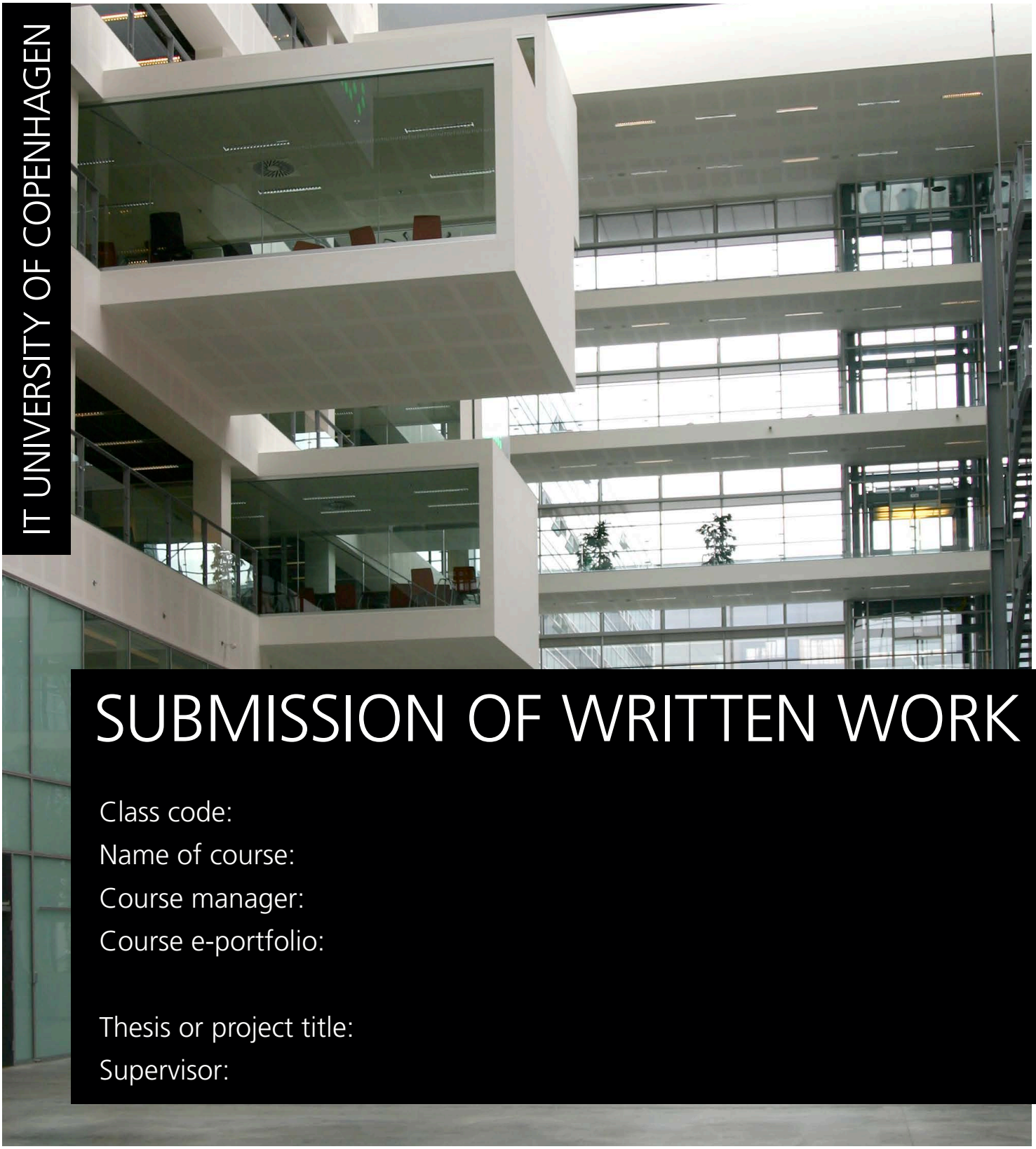
IT UNIVERSITY OF COPENHAGEN



SUBMISSION OF WRITTEN WORK

Class code:

Name of course:

Course manager:

Course e-portfolio:

Thesis or project title:

Supervisor:

Full Name:	Birthdate (dd/mm-yyyy):	E-mail:
1. _____	_____	_____@itu.dk
2. _____	_____	_____@itu.dk
3. _____	_____	_____@itu.dk
4. _____	_____	_____@itu.dk
5. _____	_____	_____@itu.dk
6. _____	_____	_____@itu.dk
7. _____	_____	_____@itu.dk

Practical Concurrent and Parallel Programming

Emil Lynegaard

December 11, 2017

I hereby declare that I have answered thede exam questions myself without any outside help.

Through all tests, the same machine will be used. Below are the results of `SystemInfo`:

OS	Linux; 4.13.12-1-ARCH; amd64
JVM	Oracle Corporation; 1.8.0_144
CPU	null; 8 "cores"
Date	2017-12-11T09:20:13+0100

Table 1: System Info

1 Question 1

1.1

Seeing as we are interested in seeing how well each implementation performs on random input of different sizes, we use Mark9 for the benchmarking, as it calculates the per element mean time and standard deviation.

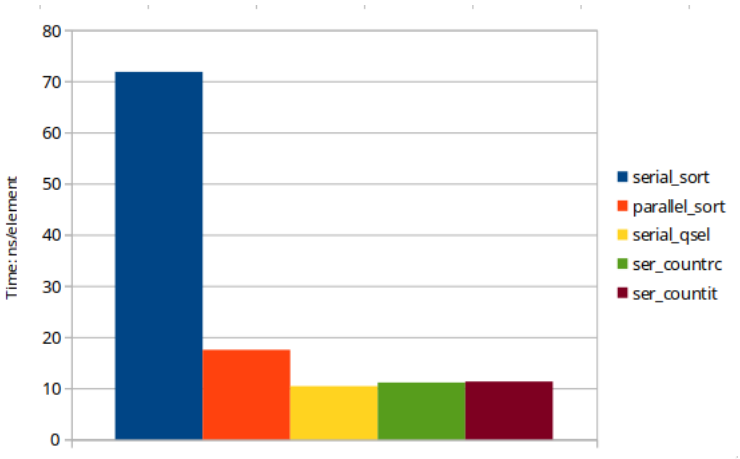


Figure 1: Plot of running times of given implementations. Tested with size = 10000000.

From graph **??** we see that, perhaps unsurprisingly, the serial quickselect and quickcount implementations beat out serial sort entirely. We do however see parallel sort get rather close to the expected linear running time implementations, due to the test machine having a quad-core CPU, allowing an approximate 4 time speedup from the serial implementation. Between the expected linear running time algorithms, it seems serial quickselect beat out its quickcount counterparts, however the difference is so small that we have yet to find a definitive winner.

1.2

2 Question 2

2.1

```
1       (int) Arrays.stream(inp).skip(1).filter(i -> i < p).count();
```

Where `inp` is our input array, and `p` is our current partition candidate. Skip the first element as this is the index of the partition element with which we do not wish to compare.

2.2

```
1           Arrays.stream(inp).skip(1).filter(i -> i < p).toArray();
2           Arrays.stream(inp).skip(1).filter(i -> i >= p).toArray();
```

2.3

```
1  public static int quickCountStream(int[] inp) {
2      int partition=-1, count=0, n=inp.length;
3      int target = n/2;
4      do {
5          partition=inp[0];
6          final int p = partition;
7          n=inp.length;
8          count = (int) Arrays.stream(inp).skip(1).filter(i -> i < p).count();
9          if (count == target) break;
10         if (count > target){
11             inp = Arrays.stream(inp).skip(1).parallel.()filter(i -> i < p).toArray();
12         }else{
13             inp = Arrays.stream(inp).skip(1).parallel().filter(i -> i >= p).toArray();
14             target=target-count-1;
15         }
16     } while( true );
17     return partition; // we are on target
18 }
```

Combining the two we get above implementation, which yields correct results.

2.4

```
1     Arrays.stream(inp).parallel().skip(1).filter(i -> i < p).count();
2     Arrays.stream(inp).parallel().skip(1).filter(i -> i < p).toArray();
3     Arrays.stream(inp).parallel().skip(1).filter(i -> i >= p).toArray();
```

Here we simply throw `.parallel()` onto the pipelines from before.

2.5

```
1  public static int quickCountStream(int[] inp) {
2      int partition=-1;
3      int target = inp.length/2;
4      // Since we have to be working with boxed Integers. We start off by converting.
5      List<Integer> list = Arrays.stream(inp).boxed().collect(Collectors.toList());
6      do {
7          partition = list.get(0);
8          final Integer p = partition;
9          Map<Boolean, List<Integer>> res = list.stream().skip(1).parallel()
10             .collect(Collectors.partitioningBy(i -> i < p));
11
12         List<Integer> smaller = res.get(true);
13         System.out.println(Arrays.toString(smaller.toArray()));
14         List<Integer> bigger = res.get(false);
15         System.out.println(Arrays.toString(bigger.toArray()));
16
17         if (smaller.size() == target) break;
18         if (smaller.size() > target) list = smaller;
19         else {
20             target=target-smaller.size()-1;
21             list = bigger;
22         }
23    } while( true );
24     return partition; // we are on target
25 }
```

To avoid having to constantly do boxing, since Collectors does not work with primitives, we start off by converting our ([] inp) to `List<Integer>`.

The `partitioningBy` collector gives us a map of all with two entries. The `true` entry on line 12, holding all elements larger than our partition element, and false entry on line 14 holding all the elements larger than or equal to our partition element. The size of `smaller` now represents our `count` from before, and the remainder of the code is similar to the given code.

2.6 TODO

3 Question 3 - TODO

By initially deciding what ranges threads will work on for the entire algorithm, we risk a thread winding up with a range of the input that we can discard as useless after a single iteration.

4 Question 4

4.1

We are not guaranteed to always grab locks in the same order, hence we are prone to deadlocks.

4.2

```
union(0,1)
union(1,0)
```

Above example when executed in parallel may lead to the first call grabbing the lock on `nodes[0]`, the second call grabbing the lock on `nodes[1]` and both calls thereafter waiting to get the lock on the node that their counterpart already grabbed.

4.3

We modify the given `concurrent()` method in class `UnionFindTest` in file `MyUnionFind.java` to target the issue we identified in 4.1.

```
public void deadlock(final int size, final UnionFind uf) throws Exception {
    final int[] numbers = new int[size];
    for (int i = 0; i < numbers.length; ++i) numbers[i] = i;
    final int threadCount = 32;
    final CyclicBarrier startBarrier = new CyclicBarrier(threadCount+1),
         stopBarrier = startBarrier;
    Collections.shuffle(Arrays.asList(numbers));
    for (int i = 0; i < threadCount; ++i) {
        final boolean reverse = i%2==0;
        Thread ti = new Thread(new Runnable() { public void run() {
            try { startBarrier.await(); } catch (Exception exn) { }
            if (reverse)
                for (int j=0; j<100; j++)
                    for (int i = 0; i < numbers.length - 1; ++i)
                        uf.union(numbers[i], numbers[i + 1]);
            else
                for (int j=0; j<100; j++)
                    for (int i = 0; i < numbers.length - 1; ++i)
```

```
19                      uf.union(numbers[i + 1], numbers[i]);
20              try { stopBarrier.await(); } catch (Exception exn) { }
21          }});
22          ti.start();
23      }
24      startBarrier.await();
25      stopBarrier.await();
26      final int root = uf.find(0);
27      for (int i : numbers) {
28          assertEquals(uf.find(i), root);
29      }
30      System.out.println("No deadlocks");
31  }
```

As seen from line 10 to 20, half the threads will now be attempting to union nodes in reverse order of the other half. From my tests, calling the above defined `deadlock` method with parameters shown below, deadlocked every time.

```
1  UnionFindTest test = new UnionFindTest();
2  test.deadlock(itemCount, new BogusFineUnionFind(itemCount));
```

By merely adding a check to the given `union()` method in class `BogusFineUnionFind` to ensure we always lock the lowest entry of the `nodes` array first, the deadlock test executes without deadlocking.

5 Question 5

Specifications:

1. pop returns an inserted item or the value null. It might block until another concurrent operation completes, but it will return without delay if no other operation is happening simultaneously. In particular, it will not block until another thread inserts some element.
2. for each element that is pushed, there is at most one pop operation that returns that element.
3. If there are no further concurrent operations, pop will succeed (i.e. return a non-null value) if so far there have been more successful push than pop operations.
4. If processor A pushed two elements x and y in this order, and processor B pops both elements, then this happens in reverse order. (There is no further constraint on ordering).

5.1

```
1  import java.util.LinkedList;
2  public class MyStack<T> {
3      private Object lock;
4      private LinkedList<T> stack;
5
```

```
6       public MyStack(){
7           lock = new Object();
8           stack = new LinkedList<T>();
9       }
10
11      public void push(T obj) {
12          synchronized(lock){
13              stack.push(obj);
14          }
15      }
16
17      public T pop() {
18          synchronized(lock){
19              return stack.peek() != null ? stack.pop() : null;
20          }
21      }
22  }
```

Above generic implementation utilizes that Java's `LinkedList` ships with `push` and `pop`. Alternatively we could use the combination `addLast` and `removeLast` or `addFirst` and `removeFirst` of which the second pair by Java's documentation is equivalent to `push` and `pop`[1]. In `pop`, given that the specification states that we should return null if the list is empty, we use `peek` to check if there is a first element, if there isn't return null, otherwise `pop`. The locking could also be done implicitly on `this` by marking the methods as synchronized, but here we use an explicit lock object as it seems more in line with specification number 1.

5.2

Given the initially stated specifications, below are added bullet points (letters) matching specifications, describing why the implementation from **??** is sufficient.

1. (a) We take care of this explicitly in the implementation for `MyStack` in section **??**, where we on line 19, `peek` prior to popping. If we blindly popped on an empty list, we would get a `NoSuchElementException`[2].
2. (a) Given that `pop` removes the single first element, and `push` only adds the element once, `pop` may only return an element pushed once.

   (b) Furthermore, since we are using a single lock, everything happens sequentially, removing any chance of reading off an element that was removed in a different thread.
3. (a) Same as 2.a.
4. (a) Since everything is handled sequentially due to the global lock, we have the guarantee that if one thread pushes two items, these will be pushed in the order of which the thread called push.

   (b) Given 3.a. and Java's Documentation stating that `push` and `pop` adds/removes from the head of the list, elements are bound to be popped in reverse compared to the order of which they were pushed.

[1] https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html
[2] https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.htmlpop()

5.3

5.3.1 Concurrency

To test for everything except reverse ordering, we define the below shown test `concurrentTest`.

```
1  public static void concurrentTest(final int size, final int threads,
2          MyStack<Integer> stack) throws Exception {
3      final CyclicBarrier startBarrier = new CyclicBarrier(threads+1),
4            stopBarrier = startBarrier;
5
6      final int range = size/threads;
7      for (int i = 0; i < threads; ++i) {
8          final int nr = i;
9          Thread ti = new Thread(new Runnable() { public void run() {
10             try { startBarrier.await(); } catch (Exception exn) { }
11                 for(int j = range*nr; j<range*nr+range; j++)
12                     stack.push(j);
13             try { stopBarrier.await(); } catch (Exception exn) { }
14         }});
15         ti.start();
16     }
17     startBarrier.await();
18     stopBarrier.await();
19     startBarrier.reset();
20     stopBarrier.reset();
21
22     final Set<Integer> pops = ConcurrentHashMap.newKeySet();
23     for (int i = 0; i < threads; ++i) {
24         final int nr = i;
25         Thread ti = new Thread(new Runnable() { public void run() {
26             try { startBarrier.await(); } catch (Exception exn) { }
27                 for(int j = range*nr; j<range*nr+range; j++)
28                     pops.add(stack.pop());
29             try { stopBarrier.await(); } catch (Exception exn) { }
30         }});
31         ti.start();
32     }
33
34     startBarrier.await();
35     stopBarrier.await();
36
37     if (pops.size() == size) System.out.println("Concurrency works :)");
38     else System.out.println("Concurrency doesn't work :(");
39 }
```

We create `CyclicBarrier`s on line 3, which we use to ensure that our threads are working simultaenously. We then assign different threads ranges between 0 and `size` in which they will push all numbers to the stack.

When all the threads are ready, we call `startBarrier.await()` on line 17 to start the pushing and `stopBarrier.await()` on the next line to wait for all of them to finish pushing.

We now divide the work similarly for popping threads. For each number they pop, we add it to the `ConcurrentHashMap` backed `Set` defined on line 22. When all threads are done popping, we can check whether the size of our `Set` is equal to the number of items we tried to add. Since there are no duplicates in a `Set`, and we only pushed unique numbers, if these are equal we can with reasonably confidence say that our implementation is working.

Scaling: In terms of scaling, since MyStack functions entirely sequentially, it scales poorly with multiple threads. In fact, the more threads we use, the more time will be spent locking and waiting for locks, making it slower and slower. In table **??** behavior is illustrated.

Threads	1	2	4	8	16	32
Time (sec)	7.363	8.159	8.203	8.413	10.071	10.832

Table 2: Test times for conccurentTest with size 10,000,000

5.3.2 Ordering

To test that order works for multiple threads, we push number $[0..n]$ onto the stack from thread A, and pop `n` items off the stack from thread B, checking that these are the numbers $[n..0]$. This is shown below in method `testOrder`.

```
1  public static void testOrder(int n, MyStack<Integer> stack){
2      final AtomicBoolean working = new AtomicBoolean(true);
3      Thread A = new Thread(new Runnable() { public void run() {
4          for(int i=0; i<n; i++) stack.push(i);
5      }});
6      Thread B = new Thread(new Runnable() { public void run() {
7          for(int i=0; i<n; i++)
8              working.compareAndSet(true, n-1-i == stack.pop());
9      }});
10     try {
11         A.start(); A.join();
12         B.join(); B.join();
13     } catch (Exception e) {
14         System.out.println("Order dies >:(");
15     }
16     if(working.get()) System.out.println("Order works :)");
17     else System.out.println("Order doesn't work :(");
18 }
```

This behaviour was untested in `concurrentTest`, so a separate straight forward test to clear this up was needed.

5.4

Below is shown an implentation using striping, with a total of 32 stripes. To determine the stripe use `Thread.currentThread()hashCode()%STRIPES)` as shown on line 16 and 24. We

use an `ArrayList` to store our `LinkedLists` since we cannot create arrays of parameterized types[3].

In `pop`, we use `i%STRIPES` to iterate through the stacks, starting from the stack at the computed `stripe`, with wraparound.

```
import java.lang.*;
import java.util.*;
public class MyStack<T> {
    private Object lock;
    private final List<LinkedList<T>> stacks;
    private static final int STRIPES = 32;

    public MyStack(){
        lock = new Object();
        stacks = new ArrayList<LinkedList<T>>();
        for(int i = 0; i < STRIPES; i++)
            stacks.add(new LinkedList<T>());
    }

    public void push(T obj) {
        int stripe = Thread.currentThread().hashCode()%STRIPES;
        LinkedList<T> stack = stacks.get(stripe);
        synchronized(stack){
            stack.push(obj);
        }
    }

    public T pop() {
        int stripe = Thread.currentThread().hashCode()%STRIPES;
        for (int i = stripe; i < stripe+STRIPES; i++){
            LinkedList<T> stack = stacks.get(i%STRIPES);
            synchronized(stack){
                if(stack.size() == 0) continue;
                return stack.pop();
            }
        }
        return null;
    }
}
```

5.5

With the new striping such that we are actually running concurrently we see improved performance in our tests. The fastest execution, as seen in table **??** was with 16 threads where it ran in 5.866 seconds. This is in contrast to the single threaded execution from table **??** that ran in 7.363 seconds with a single thread. Considering that we have 4 physical cores available, this is not that big of a performance improvement. This could be due to many threads hashing to the same stripe or due to the added iteration through all the stacks when we run into an empty

[3]https://docs.oracle.com/javase/tutorial/java/generics/restrictions.htmlcreateArrays

one.

Threads	1	2	4	8	16	32
Time (sec)	8.016	7.473	7.465	6.313	5.866	7.337

Table 3: Test times for MyStack with striping for conccurentTest with size 10,000,000

5.6

Given our implementation described in **??**, we modify line 29 to say:

```
1 return i == stripe ? stack.pop() : stack.removeLast();
```

If we are on our own stripe's stack, pop from the front, otherwise pop from the back.

5.7

Below is a description in pseudocode of what will go wrong given our changes in 5.6.

```
1 //Thread A // Stripe 0
2 myStack.push(0);
3 myStack.push(1);
4
5 //Thread B // Stripe 1
6 myStack.pop(); // this will return 0 - should be 1
7 myStack.pop(); // this will return 1 - should be 0
```

5.8

The code described in section **??** detects this issue and outputs "Order doesn't work :(" as expected.

6 Question 6

6.1

The given code has the flaw that it allows multiple threads to get into the else block before anyone changes the `state`. This means that in an example with thread `A` calling `consensus(x)` and thread `B` calling `consensus(y)`, both may retrieve return values indicating that parameter value `x` or `y` is the consensus, whereas only one of them will be stored in `state`. Hence we have a race condition.

6.2

Using synchronization on some lock, in this case on `this`, has a problem with termination. If one process is to fail while holding the lock, or fall asleep for an extended amount of time, the whole system will halt as they wait to retrieve the lock. As such, this implementation is not fault tolerant.

6.3

Firstly, assuming that we're talking Java, this would fail to typecheck, as we're returning an `AtomicInteger` from a method that supposedly returns an `int` and there is no implicit conversion from `AtomicInteger` to `int`. Even with this fixed, it would still fail, since two threads may enter the while loop before state changes, causing one of them to be stuck in the loop forever, since `state.compareAndSet(-1,x)` will then be returning false over and over. This will work if only one thread ever gets to enter the while loop and successfully `compareAndSet`s the `state`, since all other threads then merely get the value of the state, leading to consensus.