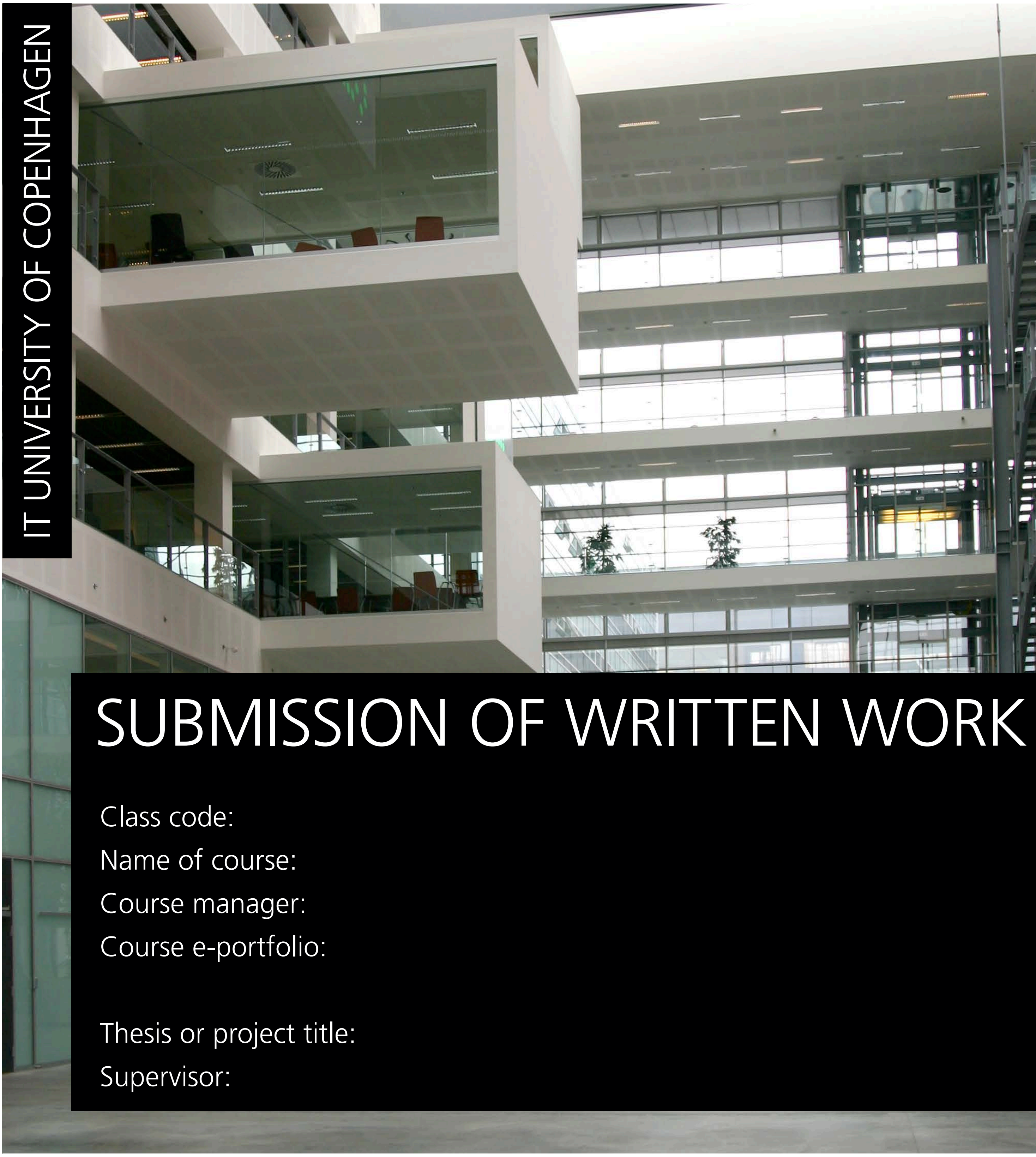

IT UNIVERSITY OF COPENHAGEN

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

Contents

Practical Concurrent and Parallel Programming

Emil Lynegaard

December 12, 2017

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

Thoughout the report, I will be using inline code snippets, but include full files in the appendix. For all tests, the same machine will be used. Table 1 contains 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.

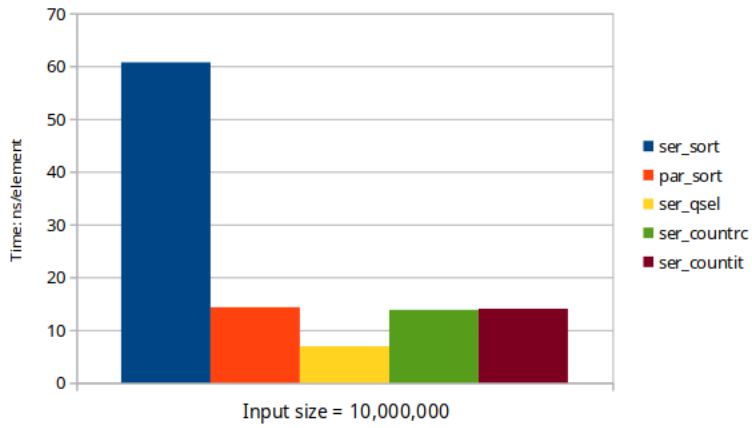

Figure 1: Plot of running times of given implementations. Tested with input size 10^7.

From Graph 1 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, serial `quickSelect` beats out its `quickCount` counterparts. This may be due to `quickSelect` avoiding spending time allocating new memory, other than its initial copy.

1.2

Below is shown a parallel `quickCount` implementation using tasks, where `quickCountItTask` is the main function, and `filter` is the auxiliary method in which the parallel filtering is done. The filter method mainly serves to reduce code duplication as the original implementation is near identical for the filtering cases where we either have a too larger or a too small selected partition number.

To allow multiple tasks to safely increment `count`, we use a `final AtomicInteger` on line 28. In general for each iteration, all variables that we need to access from within the threads are made final, as we get compile errors otherwise. For task partitioning, we try to split up the input array in even intervals skipping the partition element. For undivisible numbers, we give the leftovers to the last created task. This can slow us down a little bit, since one task may end up doing more work than the others. In `filter` we take the same approach to partitioning, and now use an `AtomicInteger` on line 8, to safely keep track of what index in the new array `m` we want to insert the next value in. This ensures that we never try to write to the same index of `m` twice.

Other than these parallization measures, the overall structure of `quickCountItTask` is similar to that of the given `quickCountTask`.

```java
// Takes an arr, a partition, the size of the output array and a BiFunction,
// returning an array of the given size containing elements from arr for which
// f.apply(arr[i], partition) returns true.
public static int[] filter(int[] arr, int partition, int size,
     BiFunction<Integer,Integer,Boolean> f){
    int[] m = new int[size];
    ArrayList<Callable<Void>> filterers = new ArrayList<>();
    final AtomicInteger j = new AtomicInteger(0);
    final int step = arr.length/threadCount;
    for(int i=0;i<threadCount;i++) {
        final int from = i==0 ? 1 : i*step;
        final int to = i==threadCount-1 ? arr.length : i*step+step;
        filterers.add(() -> {
            for(int h= from; h<to; h++)
                if(f.apply(arr[h],partition)) m[j.getAndIncrement()]=arr[h];
            return null;
        });
    }
    try{ executor.invokeAll(filterers);
    } catch (InterruptedException e) { System.err.println("Threads interrupted");}
    return m;
}

static ExecutorService executor = Executors.newWorkStealingPool();
public static int quickCountItTask(int[] in) {
    int target = in.length/2;
    do {
        final AtomicInteger count = new AtomicInteger(0);
        final int[] inp = in;
        final int n = inp.length, p = inp[0];
        final int step = n/threadCount;

        //Counting
        ArrayList<Callable<Void>> counters = new ArrayList<>();
        for(int i=0;i<threadCount;i++) {
            final int from = i==0 ? 1 : i*step; //skip pivot
            //for undivisible numbers, just let the last thread take a larger chunk
            final int to = i==threadCount-1 ? inp.length : i*step+step;
            counters.add(() -> {
                for(int j = from; j<to; j++)
                    if(inp[j]<p) count.getAndIncrement();
                return null;
            });
        }
        try{ executor.invokeAll(counters);
        } catch (InterruptedException e) { System.err.println("Threads interrupted");}

        if (count.get() == target) return p; //Terminated
```

```
        //Filtering
        boolean tooLargeP = count.get() > target;
        int size = tooLargeP ? count.get() : n-count.get()-1;
        if(tooLargeP) {
            in = filter(inp, p, size, (x,y) -> x < y);
        } else {
            in = filter(inp, p, size, (x,y) -> x >= y);
            target=target-count.get()-1;
        }
    } while( true );
}
```

1.3

Since the machine used for testing has 4 physical cores, this is the ideal amount of threads for the quickSelectIt. There is however only little difference between 4 and 8 since it supports hyperthreading[1]. Because of this, I have set the `threadCount` to 4, and tested with various input sizes. Figure 2 depicts the results.

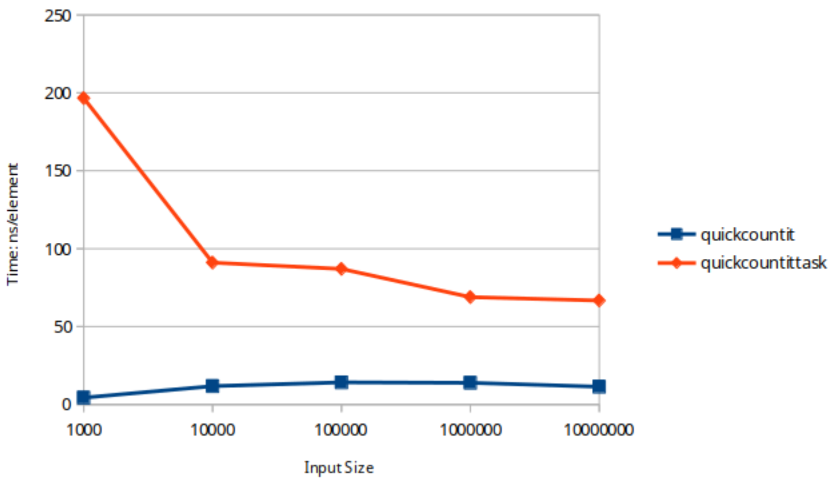

Figure 2: Plot of running times per element between `quickCountIt` and `quickCountItTask`

Perhaps in contrast to the expected outcome, `quickCountItTask` remains slower than `quickCountIt` even for large inputs. This can be due to how much it slows down on small inputs, with near $200ns/element$ for inputs of size 10^3. Based on the Figure 2 it seems that it will not be able to catch up to the serial implementation no matter the input size.

[1] https://ark.intel.com/products/80806/Intel-Core-i7-4790-Processor-8M-Cache-up-to-4_00-GHz

1.4

We assume that a single partioning step indicates one entire iteration of the `do-block` including counting and filtering. To test a single partioning step we temporarily modify `quickCountItTask` to return after a single iteration. Table 2 depicts the running time per element for inputs of size 10^4 with a varying number of threads for a single step. Here we see that on smaller input sizes, the overhead of using multiple threads becomes increasingly noticable.

Threads	1	2	4	8	16	32
Time: ns/element	15.0	28.4	32.7	35.7	35.9	36.4

Table 2: Running times per element for a single partitioning step in `quickCointItTask`, with input size 10^4, and a varying thread count

Similarly we also try to pin the thread count to 4 and vary the input size. This is depicted in Table 3. Here we see that with the pinned thread count, we seem to scale positively in the size of the input up until 10^6. This makes sense in terms of the benefits of multithreading outweighing the overhead, when each thread gets a sufficiently large part of the input to work with.

Input size	10^3	10^4	10^5	10^6	10^7	10^8
Time: ns/element	26.7	32.5	32.4	18.1	19.7	22.4

Table 3: Running times per element for a single partitioning step in `quickCointItTask`, with input size of thread count of 4 and varying input size

1.5

To make our implementation a hybrid, we declare a constant `CUTOFF` and add the following line to the top of the `do-block` in `quickCountItTask`.

```
if (in.length <= CUTOFF) return quickCountIt(in);
```

This change does make `quickCountItTask` faster, but even as a hybrid, it is unable to beat the sequential version, meaning that it still remains best to use the sequential version. This finding may either be due to the overhead of thread synchronization, or that the parallelization could be implemented in a more thread-friendly way.

2 Question 2

2.1

```
    (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

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

2.3

```java
public static int quickCountStream(int[] inp) {
    int partition=-1, count=0, n=inp.length;
    int target = n/2;
    do {
        partition=inp[0];
        final int p = partition;
        n=inp.length;
        count = (int) Arrays.stream(inp).skip(1).filter(i -> i < p).count();
        if (count == target) break;
        if (count > target){
            inp = Arrays.stream(inp).skip(1).filter(i -> i < p).toArray();
        }else{
            inp = Arrays.stream(inp).skip(1).filter(i -> i >= p).toArray();
            target=target-count-1;
        }
    } while( true );
    return partition; // we are on target
}
```

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

2.4

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

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

2.5

```java
public static int quickCountStream(int[] inp) {
    int partition=-1;
    int target = inp.length/2;
    // Since we have to be working with boxed Integers. We start off by converting.
    List<Integer> list = Arrays.stream(inp).boxed().collect(Collectors.toList());
    do {
        partition = list.get(0);
        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       List<Integer> bigger = res.get(false);
14
15       if (smaller.size() == target) break;
16       if (smaller.size() > target) list = smaller;
17       else {
18           target=target-smaller.size()-1;
19           list = bigger;
20       }
21   } while( true );
22    return partition; // we are on target
23 }
```

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 13 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

In Figure 3 we have tested all implementations from Figure 1 again, and added the hybrid implementation as well as the stream implementations. Here the `CUTOFF` for the parallel iterative `quickCount` is set to 10^4. Based on these tests, it appears that every parallel implementation is slower than the serial `quickSelect` and the serial recursive and iterative `quickCount`. While this is perhaps somewhat surprising, we can deduce that we either need smarter ways of parallelizing the implementations, or that `quickCount` does not lend itself positively to parallelization on my machine, despite it being very reasonable to implement.

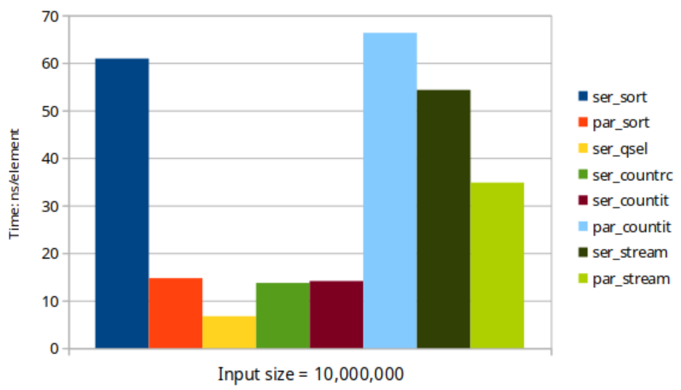

Figure 3: Plot of running times including parallel iterative and stream based. Tested with input size 10^7.

3 Question 3

3.1 CPU pinning:

If we assume that the total number of threads is equal to the number of cores on the machine of execution, CPU pinning can be advantageous if the program gets all of the machine's CPU time, and the list is evenly partitioned between threads. In these specific conditions, not having to schedule threads may be beneficial as these conditions should cause various threads to finish their iteration's work simultaneously. On the other hand, if the input is not evenly distributed and the system is using resources on other tasks as well, one partition may end up being poorly scheduled on a core with less free CPU time than one of the other available cores. Another benefit of thread pinning, which may be more significant, is that this will allow the threads to keep most of the relevant values in their designated cores' L1 cache, leading to less cache misses[2]. This benefit may even extend to L2 cache for some CPUs[3].

3.2 Load balancing:

By having each thread own certain data ranges, we may end up with several threads having nothing or very little to do after few iterations. For example, if the input is near sorted and we assign threads consecutive ranges, some threads may up filtering out all their elements in the first iteration depending on the selected pivot.

4 Question 4

4.1

Since `x` and `y` are parameters, we are not guaranteed to always grab locks in the same order, hence we are prone to dead-locks.

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.

[2]https://en.wikipedia.org/wiki/CPU_cache#Cache_miss
[3]https://en.wikipedia.org/wiki/CPU_cache#Cache_hierarchy_in_a_modern_processor

```
1 public void deadlock(final int size, final UnionFind uf) throws Exception {
2    final int[] numbers = new int[size];
3    for (int i = 0; i < numbers.length; ++i) numbers[i] = i;
4    final int threadCount = 32;
5    final CyclicBarrier startBarrier = new CyclicBarrier(threadCount+1),
6         stopBarrier = startBarrier;
7    Collections.shuffle(Arrays.asList(numbers));
8    for (int i = 0; i < threadCount; ++i) {
9       final boolean reverse = i%2==0;
10      Thread ti = new Thread(new Runnable() { public void run() {
11         try { startBarrier.await(); } catch (Exception exn) { }
12         if (reverse)
13            for (int j=0; j<100; j++)
14               for (int i = 0; i < numbers.length - 1; ++i)
15                  uf.union(numbers[i], numbers[i + 1]);
16         else
17            for (int j=0; j<100; j++)
18               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.

```
UnionFindTest test = new UnionFindTest();
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

```java
import java.util.LinkedList;
public class MyStack<T> {
    private Object lock;
    private LinkedList<T> stack;

    public MyStack(){
        lock = new Object();
        stack = new LinkedList<T>();
    }

    public void push(T obj) {
        synchronized(lock){
            stack.push(obj);
        }
    }

    public T pop() {
        synchronized(lock){
            return stack.peek() != null ? stack.pop() : null;
        }
    }
}
```

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`[4]. 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 global 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 5.1 is sufficient.

[4] https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html

1. (a) We take care of this explicitly in the implementation for `MyStack` in section 5.1, where we on line 19, `peek` prior to popping. If we blindly popped on an empty list, we would get a `NoSuchElementException`[5].
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.

5.3

5.3.1 Concurrency

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

```java
public static void concurrentTest(final int size, final int threads,
      MyStack<Integer> stack) throws Exception {
   final CyclicBarrier startBarrier = new CyclicBarrier(threads+1),
        stopBarrier = startBarrier;

   final int range = size/threads;
   for (int i = 0; i < threads; ++i) {
      final int nr = i;
      Thread ti = new Thread(new Runnable() { public void run() {
         try { startBarrier.await(); } catch (Exception exn) { }
            for(int j = range*nr; j<range*nr+range; j++)
               stack.push(j);
         try { stopBarrier.await(); } catch (Exception exn) { }
      }});
      ti.start();
   }
   startBarrier.await();
   stopBarrier.await();
   startBarrier.reset();
   stopBarrier.reset();

   final Set<Integer> pops = ConcurrentHashMap.newKeySet();
   for (int i = 0; i < threads; ++i) {
      final int nr = i;
      Thread ti = new Thread(new Runnable() { public void run() {
```

[5]https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html#pop()

```
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 `CyclicBarriers` on line 3, which we use to ensure that our threads are working simultaneously. 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 4 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 4: Test times for conccurentTest with size 10^7

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`.

```
public static void testOrder(int n, MyStack<Integer> stack){
   final AtomicBoolean working = new AtomicBoolean(true);
   Thread A = new Thread(new Runnable() { public void run() {
      for(int i=0; i<n; i++) stack.push(i);
   }});
   Thread B = new Thread(new Runnable() { public void run() {
      for(int i=0; i<n; i++)
         working.compareAndSet(true, n-1-i == stack.pop());
```

```
    }});
    try {
        A.start(); A.join();
        B.join(); B.join();
    } catch (Exception e) {
        System.out.println("Order dies >:(");
    }
    if(working.get()) System.out.println("Order works :)");
    else System.out.println("Order doesn't work :(");
}
```

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[6].

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

```
1  import java.lang.*;
2  import java.util.*;
3  public class MyStack<T> {
4      private Object lock;
5      private final List<LinkedList<T>> stacks;
6      private static final int STRIPES = 32;
7  
8      public MyStack(){
9          lock = new Object();
10         stacks = new ArrayList<LinkedList<T>>();
11         for(int i = 0; i < STRIPES; i++)
12             stacks.add(new LinkedList<T>());
13     }
14 
15     public void push(T obj) {
16         int stripe = Thread.currentThread().hashCode()%STRIPES;
17         LinkedList<T> stack = stacks.get(stripe);
18         synchronized(stack){
19             stack.push(obj);
20         }
21     }
22 
23     public T pop() {
24         int stripe = Thread.currentThread().hashCode()%STRIPES;
25         for (int i = stripe; i < stripe+STRIPES; i++){
26             LinkedList<T> stack = stacks.get(i%STRIPES);
```

[6]https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html#createArrays

```
27            synchronized(stack){
28                if(stack.size() == 0) continue;
29                return stack.pop();
30            }
31        }
32        return null;
33    }
34 }
```

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 5 was with 16 threads where it ran in 5.866 seconds. This is in contrast to the single threaded execution from table 4 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 one.

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

Table 5: Test times for MyStack with striping for conccurentTest with size 10^7

5.6

Given our implementation described in 5.4, we modify line 29 to say:

```
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.

```
//Thread A // Stripe 0
myStack.push(0);
myStack.push(1);

//Thread B // Stripe 1
myStack.pop(); // this will return 0 - should be 1
myStack.pop(); // this will return 1 - should be 0
```

5.8

The code described in section 5.3.2 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.

7 Question 7

7.1

Implementation found in appendx section 8.4.

7.2

Below is shown the output of three distinct runs of the Secure Communication System with Java+Akka.

```
$ java -cp scala.jar:akka-actor.jar:akka-config.jar:. SecComSys
public key: 18
private key: 8
cleartext: 'SECRET'
encypted: 'KWUJWL'
decrypted: 'SECRET'

$ java -cp scala.jar:akka-actor.jar:akka-config.jar:. SecComSys
public key: 2
private key: 24
cleartext: 'SECRET'
encypted: 'UGETGV'
decrypted: 'SECRET'

$ java -cp scala.jar:akka-actor.jar:akka-config.jar:. SecComSys
public key: 5
private key: 21
cleartext: 'SECRET'
encypted: 'XJHWJY'
decrypted: 'SECRET'
```

8 Appendix

8.1 TestQuickSelect.java

```java
import java.util.Arrays;
import java.util.Collections;
import java.util.Random;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Comparator;
import java.util.stream.Collectors;
import java.util.stream.IntStream;
import java.util.stream.Stream;
import java.util.function.IntFunction;
import java.util.function.IntToDoubleFunction;
import java.util.function.Function;
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
import java.util.function.BiFunction;

class TestQuickSelect {
    public static int medianSort(int[] inp) {
        int w[] = Arrays.copyOf(inp, inp.length);
        Arrays.sort(w);
        return w[w.length/2];
    }
```

```java
    public static int medianPSort(int[] inp) {
        int w[] = Arrays.copyOf(inp, inp.length);
        Arrays.parallelSort(w);
        return w[w.length/2];
    }

    public static int partition(int[] w, int min, int max) {
        int p = min; // use w[p] as pivot
        int left=min+1, right = max-1;
        while(left <= right) {
            while( w[left] <= w[p] && left < right ) left++;
            while( w[right] > w[p] && left <= right ) right--;
            if(left >= right) break;
            int t=w[left]; w[left]=w[right]; w[right]=t;
        }
        int t=w[p]; w[p]=w[right]; w[right]=t;
        return right;
    }

    public static int quickSelect(int[] inp) {
        int w[] = Arrays.copyOf(inp, inp.length);
        return quickSelect(w,0,w.length,w.length/2);
    }
    public static int quickSelect(int[] w, int min, int max, int target) {
        int p = partition(w,min,max);
        if( p < target ) return quickSelect(w,p+1,max,target);
        if( p > target ) return quickSelect(w,min,p,target);
        return w[target]; // p==target
    }

    public static int quickSelectIt(int[] inp) {
        int w[] = Arrays.copyOf(inp, inp.length);
        int target = w.length/2;
        int p = -1, min=0, max=w.length;
        do{
            p = partition(w,min,max);
            if( p < target ) min=p+1;
            if( p > target ) max=p;
            //      System.out.println(" "+p+" "+target);
        } while(p!=target);
        return w[p];
    }

    public static int quickCountRec(int[] inp, int target) {
        final int p=inp[0], n=inp.length;
        int  count=0;
        for(int i=1;i<n;i++) if(inp[i]<p) count++;
        if(count > target) {
            int m[] = new int[count];
            int j=0;
            for(int i=1;i<n;i++) if(inp[i]<p) m[j++]=inp[i];
            return quickCountRec(m,target);
        }
```

```java
        if(count < target) {
            int m[] = new int[n-count-1];
            int j=0;
            for(int i=1;i<n;i++) if(inp[i]>=p) m[j++]=inp[i];
            return quickCountRec(m,target-count-1);
        }
        return p; // we are on target
    }

    public static int quickCountIt(int[] inp) {
        int p=-1, count=0, n=inp.length;
        int target = n/2;
        do {
            p=inp[0];
            count=0;
            n=inp.length;
            for(int i=1;i<n;i++) if(inp[i]<p) count++;
            if(count > target) {
                int m[] = new int[count];
                int j=0;
                for(int i=1;i<n;i++) if(inp[i]<p) m[j++]=inp[i];
                inp = m;
                continue;
            }
            if(count < target) {
                int m[] = new int[n-count-1];
                int j=0;
                for(int i=1;i<n;i++) if(inp[i]>=p) m[j++]=inp[i];
                inp =m;
                target=target-count-1;
                continue;
            }
            break;
        } while( true );
        return p; // we are on target
    }

    // Takes an arr, a partition, the size of the output array and a BiFunction,
    // returning an array of the given size containing elements of arr for which
    // f.apply(arr[i], partition) returns true.
    public static int[] filter(int[] arr, int partition, int size,
        BiFunction<Integer,Integer,Boolean> f){
        int[] m = new int[size];
        ArrayList<Callable<Void>> filterers = new ArrayList<>();
        final AtomicInteger j = new AtomicInteger(0);
        final int step = arr.length/threadCount;
        for(int i=0;i<threadCount;i++) {
            final int from = i==0 ? 1 : i*step;
            final int to = i==threadCount-1 ? arr.length : i*step+step;
            filterers.add(() -> {
                for(int h= from; h<to; h++)
                    if(f.apply(arr[h],partition)) m[j.getAndIncrement()]=arr[h];
                return null;
```

```
            });
        }
        try{ executor.invokeAll(filterers);
        } catch (InterruptedException e) { System.err.println("Threads interrupted");}
        return m;
    }

    final static ExecutorService executor = Executors.newWorkStealingPool();
    final static int CUTOFF = 10_000;
    public static int quickCountItTask(int[] in) {
        int target = in.length/2;
        do {
            if (in.length <= CUTOFF) return quickCountIt(in);
            final AtomicInteger count = new AtomicInteger(0);
            final int[] inp = in;
            final int n = inp.length, p = inp[0];
            final int step = n/threadCount;

            //Counting
            ArrayList<Callable<Void>> counters = new ArrayList<>();
            for(int i=0;i<threadCount;i++) {
                final int from = i==0 ? 1 : i*step; //skip pivot
                // for undivisible numbers, just let the last thread take a larger chunk
                final int to = i==threadCount-1 ? inp.length : i*step+step;
                counters.add(() -> {
                    for(int j= from; j<to; j++)
                        if(inp[j]<p) count.getAndIncrement();
                    return null;
                });
            }
            try{ executor.invokeAll(counters);
            } catch (InterruptedException e) { System.err.println("Threads
                interrupted");}

            if (count.get() == target) return p; //Terminated

            //Filtering
            boolean tooLargeP = count.get() > target;
            int size = tooLargeP ? count.get() : n-count.get()-1;
            if(tooLargeP) {
                in = filter(inp, p, size, (x,y) -> x < y);
            } else {
                in = filter(inp, p, size, (x,y) -> x >= y);
                target=target-count.get()-1;
            }
        } while( true );
    }

    public static int quickCountStreamP(int[] inp) {
        int partition=-1;
        int target = inp.length/2;
        // Since we have to be working with boxed Integers. We start off by converting.
        List<Integer> list = Arrays.stream(inp).boxed().collect(Collectors.toList());
```

```
        do {
            partition = list.get(0);
            final Integer p = partition;
            Map<Boolean, List<Integer>> res = list.stream().skip(1).parallel()
                .collect(Collectors.partitioningBy(i -> i < p));

            List<Integer> smaller = res.get(true);
            List<Integer> bigger = res.get(false);

            if (smaller.size() == target) break;
            if (smaller.size() > target) list = smaller;
            else {
                target=target-smaller.size()-1;
                list = bigger;
            }
        } while( true );
        return partition; // we are on target
    }

    public static int quickCountStream(int[] inp) {
        int partition=-1;
        int target = inp.length/2;
        // Since we have to be working with boxed Integers. We start off by converting.
        List<Integer> list = Arrays.stream(inp).boxed().collect(Collectors.toList());
        do {
            partition = list.get(0);
            final Integer p = partition;
            Map<Boolean, List<Integer>> res = list.stream().skip(1)
                .collect(Collectors.partitioningBy(i -> i < p));

            List<Integer> smaller = res.get(true);
            List<Integer> bigger = res.get(false);

            if (smaller.size() == target) break;
            if (smaller.size() > target) list = smaller;
            else {
                target=target-smaller.size()-1;
                list = bigger;
            }
        } while( true );
        return partition; // we are on target
    }

    public static final int threadCount = 4;
    public static void main( String [] args ) {
        SystemInfo();
        int a[] = new int[Integer.parseInt(args[0])];
        Random rnd = new Random();
        if( args.length == 1 ) {
            int nrIt = 10;
            for(int ll=0;ll<nrIt;ll++) {
                rnd.setSeed(23434+ll); // seed
                for(int i=0;i<a.length;i++) a[i] = rnd.nextInt(4*a.length);
```

```java
                final int ra = quickCountRec(a,a.length/2);
                final int rb = medianPSort(a);
                if( ra !=rb ) {
                    System.out.println(ll);
                    System.out.println(ra);
                    System.out.println(rb);
                    System.exit(0);
                }
            }
            System.out.println();
        } else {
            rnd.setSeed(23434+Integer.parseInt(args[1])); // seed
            for(int i=0;i<a.length;i++) a[i] = rnd.nextInt(4*a.length);
            System.out.println(medianPSort(a));
            System.out.println(quickCountRec(a,a.length/2));
        }
        //    System.exit(0);
        int[] testArray = new int[]{9,2,4,3,5,7,1,8,9,6};
        double d=0.0;
        d += Mark9("serial sort", a.length, x -> medianSort(a));
        d += Mark9("parall sort", a.length, x -> medianPSort(a));
        d += Mark9("serial qsel", a.length, x -> quickSelect(a));
        d += Mark9("ser countRc", a.length,x -> quickCountRec(a,a.length/2));
        d += Mark9("ser countIt", a.length,x -> quickCountIt(a));
        d += Mark9("par countIt", a.length,x -> quickCountItTask(a));
        d += Mark9("countStream", a.length,x -> quickCountStream(a));
        d += Mark9("countStreamP", a.length,x -> quickCountStreamP(a));
        // d += Mark9("task countR", a.length,x -> quickCountRecTask(a,a.length/2));
        System.out.println(d);
    }

    public static double Mark7(String msg, IntToDoubleFunction f) {
        int n = 10, count = 1, totalCount = 0;
        double dummy = 0.0, runningTime = 0.0, st = 0.0, sst = 0.0;
        do {
            count *= 2;
            st = sst = 0.0;
            for (int j=0; j<n; j++) {
                Timer t = new Timer();
                for (int i=0; i<count; i++)
                    dummy += f.applyAsDouble(i);
                runningTime = t.check();
                double time = runningTime * 1e9 / count;
                st += time;
                sst += time * time;
                totalCount += count;
            }
        } while (runningTime < 0.25 && count < Integer.MAX_VALUE/2);
        double mean = st/n, sdev = Math.sqrt((sst - mean*mean*n)/(n-1));
        System.out.printf("%-25s %15.1f ns %10.2f %10d%n", msg, mean, sdev, count);
        return dummy / totalCount;
    }

```

```java
    public static double Mark9(String msg, int size, IntToDoubleFunction f) {
        int n = 5, count = 1, totalCount = 0;
        double dummy = 0.0, runningTime = 0.0, st = 0.0, sst = 0.0;
        do {
            count *= 2;
            st = sst = 0.0;
            for (int j=0; j<n; j++) {
                Timer t = new Timer();
                for (int i=0; i<count; i++)
                    dummy += f.applyAsDouble(i);
                runningTime = t.check();
                double time = runningTime * 1e9 / count; // microseconds
                st += time;
                sst += time * time;
                totalCount += count;
            }
        } while (runningTime < 0.25 && count < Integer.MAX_VALUE/2);
        double mean = st/n/size, sdev = Math.sqrt((sst - mean*mean*n)/(n-1))/size;
        System.out.printf("%-25s %15.1f ns %10.2f %10d%n", msg, mean, sdev, count);
        return dummy / totalCount;
    }

    public static void SystemInfo() {
        System.out.printf("# OS: %s; %s; %s%n",
                System.getProperty("os.name"),
                System.getProperty("os.version"),
                System.getProperty("os.arch"));
        System.out.printf("# JVM: %s; %s%n",
                System.getProperty("java.vendor"),
                System.getProperty("java.version"));
        // The processor identifier works only on MS Windows:
        System.out.printf("# CPU: %s; %d \"cores\"%n",
                System.getenv("PROCESSOR_IDENTIFIER"),
                Runtime.getRuntime().availableProcessors());
        java.util.Date now = new java.util.Date();
        System.out.printf("# Date: %s%n",
                new java.text.SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ssZ").format(now));
    }
}
```

8.2 MyUnionFind.java

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

import java.util.concurrent.Callable;
import java.util.concurrent.CyclicBarrier;

import java.util.Arrays;
import java.util.Collections;

```

```java
public class MyUnionFind {
    public static void main(String[] args) throws Exception {
        final int itemCount = 10_000;
        {
            UnionFindTest test = new UnionFindTest();
            test.sequential(new FineUnionFind(5));
            test.concurrent(itemCount, new FineUnionFind(itemCount));
            test.deadlock(itemCount, new FineUnionFind(itemCount));
        }
        // Question 4.3
        {
            UnionFindTest test = new UnionFindTest();
            test.sequential(new BogusFineUnionFind(5));
            test.concurrent(itemCount, new BogusFineUnionFind(itemCount));
            test.deadlock(itemCount, new BogusFineUnionFind(itemCount));
        }
    }
}

interface UnionFind {
    int find(int x);
    void union(int x, int y);
    boolean sameSet(int x, int y);
}

// Test of union-find data structures, adapted from Florian Biermann's
// MSc thesis, ITU 2014

class UnionFindTest extends Tests {

    public void sequential(UnionFind uf) throws Exception {
        System.out.printf("Testing %s ... ", uf.getClass());
        // Find
        assertEquals(uf.find(0), 0);
        assertEquals(uf.find(1), 1);
        assertEquals(uf.find(2), 2);
        // Union
        uf.union(1, 2);
        assertEquals(uf.find(1), uf.find(2));

        uf.union(2, 3);
        assertEquals(uf.find(1), uf.find(2));
        assertEquals(uf.find(1), uf.find(3));
        assertEquals(uf.find(2), uf.find(3));

        uf.union(1, 4);
        assertEquals(uf.find(1), uf.find(2));
        assertEquals(uf.find(1), uf.find(3));
        assertEquals(uf.find(2), uf.find(3));
        assertEquals(uf.find(1), uf.find(4));
        assertEquals(uf.find(2), uf.find(4));
        assertEquals(uf.find(3), uf.find(4));
    }
```

```java

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;
    // Populate threads
    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)
                        uf.union(numbers[i + 1], numbers[i]);
            try { stopBarrier.await(); } catch (Exception exn) { }
        }});
        ti.start();
    }
    startBarrier.await();
    stopBarrier.await();
    final int root = uf.find(0);
    for (int i : numbers) {
        assertEquals(uf.find(i), root);
    }
    System.out.println("No deadlocks");
}

public void concurrent(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;
    // Populate threads
    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) {
        Thread ti = new Thread(new Runnable() { public void run() {
            try { startBarrier.await(); } catch (Exception exn) { }
            for (int j=0; j<100; j++)
                for (int i = 0; i < numbers.length - 1; ++i)
                    uf.union(numbers[i], numbers[i + 1]);
            try { stopBarrier.await(); } catch (Exception exn) { }
        }});
        ti.start();
    }
```

```java
        startBarrier.await();
        stopBarrier.await();
        final int root = uf.find(0);
        for (int i : numbers) {
            assertEquals(uf.find(i), root);
        }
        System.out.println("passed");
    }
}

class Tests {
    public static void assertEquals(int x, int y) throws Exception {
        if (x != y)
            throw new Exception(String.format("ERROR: %d not equal to %d%n", x, y));
    }

    public static void assertTrue(boolean b) throws Exception {
        if (!b)
            throw new Exception(String.format("ERROR: assertTrue"));
    }
}
// Fine-locking union-find. Union and sameset lock on the intrinsic
// locks of the two root Nodes involved. Find is wait-free, takes no
// locks, and performs no compression.

// The nodes[] array entries are never updated after initialization
// inside the constructor, so no need to worry about their visibility.
// But the fields of Node objects are written (by union and compress
// while holding locks), and read by find without holding locks, so
// must be made volatile.

class FineUnionFind implements UnionFind {
    private final Node[] nodes;

    public FineUnionFind(int count) {
        this.nodes = new Node[count];
        for (int x=0; x<count; x++)
            nodes[x] = new Node(x);
    }

    public int find(int x) {
        while (nodes[x].next != x)
            x = nodes[x].next;
        return x;
    }

    public void union(final int x, final int y) {
        while (true) {
            int rx = find(x), ry = find(y);
            if (rx == ry)
                return;
            else if (rx > ry) {
                int tmp = rx; rx = ry; ry = tmp;
```

```java
            }
            // Now rx < ry; take locks in consistent order
            synchronized (nodes[rx]) {
                synchronized (nodes[ry]) {
                    // Check rx, ry are still roots, else restart
                    if (nodes[rx].next != rx || nodes[ry].next != ry)
                        continue;
                    if (nodes[rx].rank > nodes[ry].rank) {
                        int tmp = rx; rx = ry; ry = tmp;
                    }
                    // Now nodes[rx].rank <= nodes[ry].rank
                    nodes[rx].next = ry;
                    if (nodes[rx].rank == nodes[ry].rank)
                        nodes[ry].rank++;
                    compress(x, ry);
                    compress(y, ry);
                } }
        }
    }

    // Assumes lock is held on nodes[root]
    private void compress(int x, final int root) {
        while (nodes[x].next != x) {
            int next = nodes[x].next;
            nodes[x].next = root;
            x = next;
        }
    }

    public boolean sameSet(int x, int y) {
        return find(x) == find(y);
    }

    class Node {
        private volatile int next, rank;

        public Node(int next) {
            this.next = next;
        }
    }
}

// Bogus Fine-locking union-find. Union and sameset lock on the intrinsic
// locks of the two root Nodes involved. Find is wait-free, takes no
// locks, and performs no compression.

// The nodes[] array entries are never updated after initialization
// inside the constructor, so no need to worry about their visibility.
// But the fields of Node objects are written (by union and compress
// while holding locks), and read by find without holding locks, so
// must be made volatile.

class BogusFineUnionFind implements UnionFind {
```

```java
    private final Node[] nodes;

    public BogusFineUnionFind(int count) {
        this.nodes = new Node[count];
        for (int x=0; x<count; x++)
            nodes[x] = new Node(x);
    }

    public int find(int x) {
        while (nodes[x].next != x)
            x = nodes[x].next;
        return x;
    }

    public void union(final int x, final int y) {
        while (true) {
            int rx = find(x), ry = find(y);
            if (rx == ry)
                return;
            synchronized (nodes[rx]) {
                synchronized (nodes[ry]) {
                    // Check rx, ry are still roots, else restart
                    if (nodes[rx].next != rx || nodes[ry].next != ry)
                        continue;
                    if (nodes[rx].rank > nodes[ry].rank) {
                        int tmp = rx; rx = ry; ry = tmp;
                    }
                    // Now nodes[rx].rank <= nodes[ry].rank
                    nodes[rx].next = ry;
                    if (nodes[rx].rank == nodes[ry].rank)
                        nodes[ry].rank++;
                    compress(x, ry);
                    compress(y, ry);
                }
            }
        }
    }

    // Assumes lock is held on nodes[root]
    private void compress(int x, final int root) {
        while (nodes[x].next != x) {
            int next = nodes[x].next;
            nodes[x].next = root;
            x = next;
        }
    }

    public boolean sameSet(int x, int y) {
        return find(x) == find(y);
    }

    class Node {
        private volatile int next, rank;
```

```java

        public Node(int next) {
            this.next = next;
        }
    }
}
```

8.3 MyStack.java

```java
import java.lang.*;
import java.util.*;
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
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 i == stripe ? stack.pop() : stack.removeLast();
            }
        }
        return null;
    }

    public static void concurrentTest(final int size, final int threads,
            MyStack<Integer> stack) throws Exception {
        final CyclicBarrier startBarrier = new CyclicBarrier(threads+1),
             stopBarrier = startBarrier;

        final int range = size/threads;
```

```java
        for (int i = 0; i < threads; ++i) {
            final int nr = i;
            Thread ti = new Thread(new Runnable() { public void run() {
                try { startBarrier.await(); } catch (Exception exn) { }
                    for(int j = range*nr; j<range*nr+range; j++)
                        stack.push(j);
                try { stopBarrier.await(); } catch (Exception exn) { }
            }});
            ti.start();
        }
        startBarrier.await();
        stopBarrier.await();
        startBarrier.reset();
        stopBarrier.reset();

        final Set<Integer> pops = ConcurrentHashMap.newKeySet();
        for (int i = 0; i < threads; ++i) {
            final int nr = i;
            Thread ti = new Thread(new Runnable() { public void run() {
                try { startBarrier.await(); } catch (Exception exn) { }
                    for(int j = range*nr; j<range*nr+range; j++)
                        pops.add(stack.pop());
                try { stopBarrier.await(); } catch (Exception exn) { }
            }});
            ti.start();
        }

        startBarrier.await();
        stopBarrier.await();

        if (pops.size() == size) System.out.println("Concurrency works :)");
        else  System.out.println("Concurrency doesn't work :(");
    }

    public static void testOrder(int n, MyStack<Integer> stack){
        final AtomicBoolean working = new AtomicBoolean(true);
        Thread A = new Thread(new Runnable() { public void run() {
            for(int i=0; i<n; i++) stack.push(i);
        }});
        Thread B = new Thread(new Runnable() { public void run() {
            for(int i=0; i<n; i++)
                working.compareAndSet(true, n-1-i == stack.pop());
        }});
        try {
            A.start(); A.join();
            B.join(); B.join();
        } catch (Exception e) {
            System.out.println("Order dies >:(");
        }
        if(working.get()) System.out.println("Order works :)");
        else System.out.println("Order doesn't work :(");
    }

```

```java
    public static void main(String[] args){
        int size = 10_000_000;
        int threads = 32;
        MyStack<Integer> stack = new MyStack<Integer>();
        testOrder(size, stack);
        try {
            concurrentTest(size, threads, stack);
        } catch (Exception e){
            System.out.println("Concurrent test died >:(");
        }
        System.exit(0);
    }
}
```

8.4 SecComSys.java

```java
// COMPILE:
// javac -cp scala.jar:akka-actor.jar SecComSys.java
// RUN:
// java -cp scala.jar:akka-actor.jar:akka-config.jar:. SecComSys

import java.util.*;
import java.io.*;
import akka.actor.*;

// -- HANDOUT ---------------------------------------------------
class KeyPair implements Serializable {
    public final int public_key, private_key;
    public KeyPair(int public_key, int private_key) {
        this.public_key = public_key;
        this.private_key = private_key;
    }
}

class Crypto {
    static KeyPair keygen() {
        int public_key = (new Random()).nextInt(25)+1;
        int private_key = 26 - public_key;
        System.out.println("public key: " + public_key);
        System.out.println("private key: " + private_key);
        return new KeyPair(public_key, private_key);
    }

    static String encrypt(String cleartext, int key) {
        StringBuffer encrypted = new StringBuffer();
        for (int i=0; i<cleartext.length(); i++) {
            encrypted.append((char) ('A' + ((((int)
                                    cleartext.charAt(i)) - 'A' + key) % 26)));
        }
        return "" + encrypted;
    }
```

```java
}

// -- MESSAGES ---------------------------------------------------

class InitMessage implements Serializable {
    public final ActorRef R;
    public InitMessage(ActorRef R) {
        this.R = R;
    }
}

class RegisterMessage implements Serializable {
    public final ActorRef pid;
    public RegisterMessage(ActorRef pid) {
        this.pid = pid;
    }
}

class LookupMessage implements Serializable {
    public final ActorRef pid;
    public final ActorRef returnTo;
    public LookupMessage(ActorRef pid, ActorRef returnTo) {
        this.pid = pid;
        this.returnTo = returnTo;
    }
}

class KeyPairMessage implements Serializable {
    public final KeyPair keyPair;
    public KeyPairMessage(KeyPair keyPair) {
        this.keyPair = keyPair;
    }
}

class Message implements Serializable {
    public final String Y;
    public Message(String Y) {
        this.Y = Y;
    }
}

class CommMessage implements Serializable {
    public final ActorRef pid;
    public CommMessage(ActorRef pid) {
        this.pid = pid;
    }
}

class PubKeyMessage implements Serializable {
    public final ActorRef recipient;
    public final Integer publicKey;
    public PubKeyMessage(ActorRef recipient, int publicKey) {
        this.recipient = recipient;
```

```
        this.publicKey = publicKey;
    }
}

// -- ACTORS ---------------------------------------------------

class RegistryActor extends UntypedActor {
    public final Map<ActorRef, Integer> registry = new HashMap<>();

    public void onReceive(Object o) throws Exception {
        if (o instanceof RegisterMessage) {
            RegisterMessage rm = (RegisterMessage) o;
            KeyPair keyPair = Crypto.keygen();
            registry.put(rm.pid, keyPair.public_key);
            rm.pid.tell(new KeyPairMessage(keyPair), getSelf());
        } else if (o instanceof LookupMessage) {
            LookupMessage lm = (LookupMessage) o;
            lm.returnTo.tell(new PubKeyMessage(lm.pid, registry.get(lm.pid)), getSelf());
        }
    }
}

class ReceiverActor extends UntypedActor {
    public ActorRef registry;
    public int publicKey;
    public int privateKey;

    public void onReceive(Object o) throws Exception {
        if (o instanceof InitMessage) {
            InitMessage im = (InitMessage) o;
            im.R.tell(new RegisterMessage(getSelf()), getSelf());
        } else if (o instanceof KeyPairMessage) {
            KeyPairMessage kpm = (KeyPairMessage) o;
            publicKey = kpm.keyPair.public_key;
            privateKey = kpm.keyPair.private_key;
        } else if (o instanceof Message) {
            Message m = (Message) o;
            String X = Crypto.encrypt(m.Y, privateKey);
            System.out.print("decrypted: '" + X + "'\n");
        }
    }
}

class SenderActor extends UntypedActor {
    public ActorRef registry;

    public void onReceive(Object o) throws Exception {
        if (o instanceof InitMessage) {
            InitMessage im = (InitMessage) o;
            registry = im.R;
        } else if (o instanceof CommMessage) {
            CommMessage cm = (CommMessage) o;
            registry.tell(new LookupMessage(cm.pid, getSelf()), getSelf());
```

```
        } else if (o instanceof PubKeyMessage) {
            PubKeyMessage pkm = (PubKeyMessage) o;
            String X = "SECRET";
            System.out.print("cleartext: '" + X + "'\n");
            String Y = Crypto.encrypt(X, pkm.publicKey);
            System.out.print("encypted: '" + Y + "'\n");
            pkm.recipient.tell(new Message(Y), getSelf());
        }
    }
}
//
// -- MAIN -------------------------------------------------

public class SecComSys {
    public static void main(String[] args) {
        final ActorSystem system = ActorSystem.create("SecComSys");
        final ActorRef registry = system.actorOf(Props.create(RegistryActor.class),
            "reigstry");
        final ActorRef receiver = system.actorOf(Props.create(ReceiverActor.class),
            "receiver");
        receiver.tell(new InitMessage(registry), ActorRef.noSender());
        final ActorRef sender = system.actorOf(Props.create(SenderActor.class),
            "sender");
        sender.tell(new InitMessage(registry), ActorRef.noSender());
        sender.tell(new CommMessage(receiver), ActorRef.noSender());
        system.shutdown();
    }
}
```