Examination, Practical Concurrent and Parallel Programming
11–12 December 2017

These exam questions comprise 8 pages; check immediately that you have them all.

The exam questions are handed out in digital form from LearnIT and from the public course homepage on Monday 11 December 2017 at 08:00 local time.

Your solution must be handed in no later than **Tuesday 12 December 2017 at 14:00** according to these rules:

- Your solution must be handed in through LearnIT.
- Your solution must be handed in as a single PDF file, including source code, written explanations in English, tables, charts and so on, as further specified below.
- Your solution must have a standard ITU front page, available at `http://studyguide.itu.dk/SDT/Your-Programme/Forms`

There are 7 main questions. For full marks, all these questions must be satisfactorily answered.

If you find unclarities, inconsistencies or misprints in the exam questions, then you must describe these in your answers and describe what interpretation you applied when answering the questions.

Your solutions and answers must be made by you and you only. This applies to program code, examples, tables, charts, and explanatory text (in English) that answer the exam questions. You are not allowed to create the exam solutions as group work, nor to consult with fellow students, pose questions on internet fora, or the like. You are allowed to ask for clarification of possible mistakes, misprints, and so on, by private email to `rikj@itudk`. Expect the answer in the course LearnIT discussion forum. You should occasionally check the forum for news about mistakes and unclarities.

Your solution must contain the following declaration:

> **I hereby declare that I have answered these exam questions myself without any outside help.**
> (name) (date)

When creating your solution you are welcome to use all books, lecture notes, lecture slides, exercises from the course, your own solutions to these exercises, internet resources, pocket calculators, text editors, office software, compilers, and so on.

You are **of course not allowed to plagiarize** from other sources in your solutions. You must not attempt to take credit for work that is not your own. Your solutions must not contain text, program code, figures, charts, tables or the like that are created by others, unless you give a complete reference, describing the source in a complete and satisfactory manner. This holds also if the included text is not an identical copy, but adapted from text or program code in an external source.

You need not give a reference when using code from these exam questions or from the mandatory course literature, but even in that case your solution may be easier to understand and evaluate if you do so.

If an exam question requires you to define a particular method, you are welcome to define any auxiliary methods that will make your solution clearer, provided the requested method has exactly the result type and parameter types required by the question. Similarly, when defining a particular class, you are welcome to define auxiliary classes and methods.

The exam will be followed by a **cheat check** ("snydtjek"): The study administration randomly selects 20 percent of students; the list will be published on the course LearnIT page at 14:00 on Tuesday. The selected students must present themselves in the room 2A18 on Tuesday 12 December at 15:00. Each will be questioned for 5 minutes about his/her own exam handin. (This is only to discover possible cheating, and otherwise does not influence the grade). Students who sit the exam from some remote location must be available for interviews via Skype, and must seek permission for this beforehand.

What to hand in Your solution should be a short report in PDF consisting of text (in English) that answers the exam questions, with relevant program fragments shown inline or supplied in appendixes, and a clear indication which code fragments belong to which answers. You may need to use tables and charts and possibly other figures. Take care that the program code retains a sensible layout and indentation in the report so that it is readable.

Finding the median – Quickselect

In a sorted array of of n integers it is easy to find the median: It is the element at position $n/2$ in the array, because of the sortedness. So, we can find the median by sorting the array, but that is not necessary. If we run quicksort and know in advance that the only thing we are interested in is the element at a particular position k ($k = n/2$ for the median), one of the recursive calls is unnecessary. The resulting algorithm is known as quickselect, and it has expected linear running time (improving over the expected $O(n \log n)$ running time of quicksort).

More background information can be found at
`https://en.wikipedia.org/wiki/Median`
`https://en.wikipedia.org/wiki/Selection_algorithm`
`https://en.wikipedia.org/wiki/Quickselect`

In classical quicksort and quickselect, the partitioning algorithm is based on intertwined scanning and swapping from the beginning and the end of the array until the two scans meet at the position where the pivot element is to be placed. There are several variants of this approach, you can find an implementation in the accompanying source code as the methods `partition(...)` and its use in `quickselect(...)`. As it stands, this partitioning algorithm cannot be parallelized.

There is an alternative that lends itself to parallelization, inspired by the partitioning step of sample sort (also spelled out as `quickCountRec()` in the file `TestQuickSelect.java`). Instead of modifying the array (by swapping), it creates one new array consisting only of the elements containing the rank k element. We recurse on that array with an adjusted k. The actual work is split into first counting how many elements are smaller than the pivot, and then filtering out either the elements smaller than the pivot or the elements larger than the pivot into the new array.

The advantage of this approach is the possibility to paralellize the two steps. We divide the input array equally into regions A_i. In the first stage, we can in parallel count the number s_i of elements smaller than the pivot in the region A_i. Now the total number of elements smaller than the pivot is the sum of all s_i, telling in which direction we need to filter. Additionally, the s_i (or their complements $s'_i = |A_i| - s_i$) determine the regions of the new array that is populated by the filtered elements of A_i, namely it starts at $\sum_{j=0}^{i-1} s_i$ (or $\sum_{j=0}^{i-1} s'_i$ if the pivot was smaller than the rank k element). With this information computed by the main thread, the filtering of the regions itself can be done in parallel without any further coordination.

Here is a Java implementation of the quickselect algorithm (in file TestQuickSelect.java):

```
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
}
```

Here is a Java implementation of the quickselect algorithm based on first counting and then filtering (in file TestQuickSelect.java):

```
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);
  }
  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
}
```

Source code for the questions File `TestQuickSelect.java` contains:

- static method `medianSort`, `medianPSort` reference implementation using library sort
- static method `quickSelect` and `partition` implementing sequential recursive quickSelect.
- static method `quickSelectIt` also using `partition` implementing sequential iterative quickSelect.
- static method `quickCountRec` (also `quickCountIt` in an iterative version) implementing sequential iterative quickSelect using counting and filtering.
- (Class `Timer` for simple wall-clock time measurements.)
- (Method `Mark7` (from the lectures) to microbenchmark functions.)
- Method `Mark9` (from the lectures) to microbenchmark functions running on an instance of size n and show the running time per element.

Question 1 (20 %):

Implement the above paralellisation using first Threads or Tasks (not streams). Benchmark your solution. More precisely:

1. Benchmark the serial reference implementations given in the file `TestQuickSelect.java`, comparing the versions based on (parallel) sorting and quickselect. (`medianSort(...)`, `medianPSort(...)`, `quickSelect(...)`, `quickSelectIt(...)`, `quickCountRec(...)`, and `quickCountIt(...)`. Use the appropriate "`Mark`" benchmarking methods on random input of different sizes, up to the biggest that runs in less than a second on your system. Show the the information produced by the benchmark method `SystemInfo` for your system. Focus on the running time per element. Produce graphs comparing the different implementations. Discuss your findings.
2. Parallelize `quickCountIt(...)` as outlined above. It is up to you if you want to use threads or tasks. Make sure that the number of threads or tasks is a (compile time) parameter so that you can easily change it. Convince yourself of the correctness of your implementation (observe that the deterministic choice of the first element as pivot leads to very similar intermediate arrays for the different algorithms). Describe your approach. Are there visibility issues? Is there shared mutable state? How do you achieve synchronization? Show the resulting code, explain what you are doing and why.

3. Benchmark your parallel solution. Compare it to `quickSelectIt()` for different number of threads/tasks, using the benchmarking framework you already created. How many threads are beneficial? For which parameters does the parallel implementation outperform the serial one?
4. Benchmark a single partitioning step. Analyze the influence of the array size and the number of threads/tasks.
5. Under the assumption that the serial solutions outperform the parallel one for small array sizes (we would expect this), create a hybrid implementation that switches to serial once the array is smaller than the threshold.

Question 2 (15 %):

Implement the above parallisation using parallel streams.

1. Write a stream based version of counting the number of elements in the array that are smaller than the pivot.
2. Write a stream based version of filtering the elements smaller/larger than the pivot into a new array.
3. Combine the two and check that the resulting implementation is correct.
4. Use `.parallel()` to parallelize the stream based solution.
5. Combine the counting and filtering step into one partitioning step using `Collectors.partitioningBy()`
6. Benchmark the serial and parallel stream based solution against the reference implementation and your implementation from the previous question. Discuss your findings.

Question 3 (5 %):

This question is discussion only. Don't implement this version!
Discuss the possibility of implementing a thread based version:

- Each thread owns a certain part of the data;
- there are phases guarded by a (cyclic) barrier;
- a global pivot is chosen;
- each thread counts below pivot;
- this is added up globally;
- each thread filters and counts against the next pivot;
- until there is only one element left.

Is there a potential advantage of thread pinning? Are there potential load balancing issues?

Question 4 (10%):

(This question is unrelated to the preceding ones).

Consider the implementation of union find given in file `MyUnionFind.java`.

```
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);
      }
    }
  }
}
```

1. Why is this implementation dead-lock prone?
2. Give an example program using this data structure and a schedule that deadlocks.
3. Implement a test program that uses the above data structure and provokes a deadlock with reasonable probability and run it. Do you see it deadlocking? Discuss your findings.

Question 5 (20%):

(This question is unrelated to the preceding ones).

Consider a stack data structure. The interface has a push and a pop operation. The data structure needs to be thread safe and comply to the following:

1. for each element that is pushed, there is at most one pop operation that returns that element.
2. 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.
3. 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).

Here are the questions:

1. Implement the data structure as a classical linked list (e.g. using Java's `LinkedList` type) with a single global lock.
2. Argue that it meets the specification.
3. Write a test program that tests:
   - Correctness: are the three functional requirements above met in a concurrent setting?

- Performance: how does the implementation scale with more threads?

4. We want to improve the performance by lock striping, using the following ideas:
   - There is an array (of size 32) of stacks, each with a lock of its own.
   - The threads are hashed to (by their hash code) to one of the stacks. We call this its assigned stack.
   - A push of a thread is always done on the assigned stack.
   - A pop is attempted from the assigned stack first.
   - If popping from the assigned stack fails (i.e. the assigned stack is empty), the stacks of the array are tried in turn; if none of these is successful, the pop fails (i.e. returns `null`).

   Implement the above idea.
5. Run a performance test. Do you see the hoped for improvement in performance?
6. Change your implementation by pulling from the non-assigned stacks from the wrong side (i.e. in FIFO order).
7. Describe an example execution with two threads (and two different assigned (non-)stacks) that shows that the change violates the specification.
8. Make sure your test program detects the mistake.

Question 6 (10%)

(This question is unrelated to the preceding ones).

Consider the following naive attempt at implementing a consensus data structure. The interface is one method

```
volatile int state = -1;
int consensus(int x) { // invariant: x > 0
  assert( x > 0 ) ;
  if( state > 0 )
     return state;
  else
     state = x;
     return state;
}
```

Questions:

1. Describe a situation where the above does not achieve consensus.
2. Now consider the variant where the `consensus` method is declared `synchronized`. Which requirement for consensus protocols is violated by this solution? Give an example execution.
3. Consider the following variant:

   ```
   AtomicInteger state = new AtomicInteger(-1);
   int consensus(int x) { // invariant: x > 0
     assert( x > 0 ) ;
     if( state.get() > 0 ) return state;
     while( true ) {
       if( state.compareAndSwap(-1,x) ) return state.get();
     }
   }
   ```

   Does this work? If yes, give an argument, if not, describe a situation in which it fails, and if possible one where it works as intended?

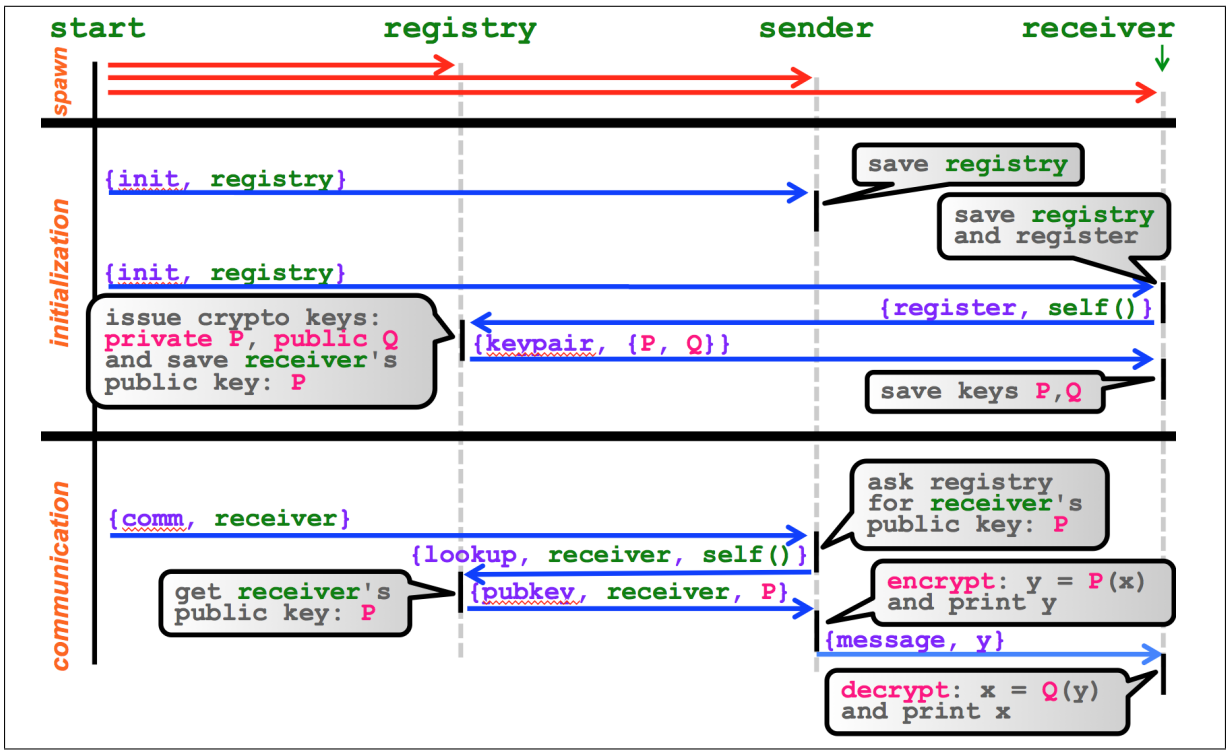

Figure 1: Communication diagram for the *secret communication system*.

Question 7 (20%)

(This question is unrelated to the preceding ones).

The file `SecComSys.erl` contains an ERLANG specification of a *secret communication system* that has four distinct actors and three distinct phases: *spawn*, *initialization*, and *communication*, as shown in orange to the left of the communication diagram (cf. Figure 1).

The `start` actor is responsible for *spawning* the other actors, send appropriate *initialization* messages, and finally send a *comm(unication)* message which will cause a secret to be communicated between two actors. The `registry` actor issues a key pair (a *public key* and a *private key*) whenever someone registers with it. The registry also maintains a mapping from actors to public keys, so that it can perform a *lookup* operation whenever someone asks it for the public key of a particular (registered) actor. The `sender` actor will, when it receives an initialization message, save a reference to the registry so that it can contact it later. When the `sender` actor later receives a *comm(unication)* message (instructing it to communicate a secret with a recipient), it will ask the registry for the public key of the recipient. Then, it will use the public key of the recipient to encrypt the secret message, print it out, and send it to the recipient. The `receiver` actor will, when it receives an initialization message, try to register with the registry. When it subsequently receives a key pair, it will save the keys for later use. When it finally receives an encrypted message, it will use its private key to decrypt the message and print it.

Since, this is about *message passing concurrency* and *not cryptographic protocols*, we will provide you with the JAVA code for generating *public* and *private* key pairs and for *en-* and *de*-crypting messages (see further below); simply use the *public key* for *encryption* and the *private key* for *decryption*. (This is a simple ROT13-style encryption; however, obviously, in a real cryptographic system it should not be possible to derive the private key from the public key, but that is an independent issue which we will not elaborate in this simplified system.)

Here is a run of the system (it should produce the same output in ERLANG and JAVA+AKKA):

```
[press return to terminate...]
public key: 19
private key: 7
cleartext: 'SECRET'
encrypted: 'LXVKXM'
decrypted: 'SECRET'
```

TASKS:

- 1. Implement the system in JAVA+AKKA (as close to the ERLANG version as reasonably possible).
- 2. Run your system and document the output (three distinct runs is enough).

```
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;
    }
}
```