Examination, Practical Concurrent and Parallel Programming 11–12 January 2016

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 January 2016 at 09:00 local time.

Your solution must be handed in no later than **Tuesday 12 January 2016 at 15:00** according to these rules:

- Your solution must be handed in through LearnIT.
- Your solution must be handed in in the form of 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 12 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 answers 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, in the course LearnIT discussion forum. You are encouraged to 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, that is, you describe 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 signature (result type and parameter types) required by the question. Similarly, when defining a particular class, you are welcome to define auxiliary classes and methods, provided the requested class has at least the required public methods (and signatures).

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 no later than 19:00 on Tuesday. The selected students must present themselves at Peter's office 4D15 on Wednesday 13 January at 09:00, and each will be questioned for 5–8 minutes about his/her own exam handin. (This is only to discover possible cheating, and otherwise does not influence the grade).

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. In addition, you will probably need to use tables and charts and possibly other figures. Take care that the program code retains a sensible layout in the report so that it is readable.

Part 1: Locks

The first major part of this exam concerns the meaning and use of basic lock operations.

Question 1 (5 %):

Consider the small artificial program in file TestLocking0.java. In class Mystery, the single mutable field `sum` is private, and all methods are synchronized, so superficially the class seems to be threadsafe.

1. Compile the program and run it several times. Show the results you get. Do they indicate that class Mystery is thread-safe or not? Discuss.
2. Explain why class Mystery is not thread-safe. Hint: Consider (a) what it means for an instance method to be synchronized, and (b) what it means for a static method to be synchronized.
3. Explain how you could make the class threadsafe. Do it, and rerun the program to see whether it works; report the results.

Question 2 (5 %):

Consider class DoubleArrayList in TestLocking1.java. It implements an array list of numbers, and like Java's ArrayList it dynamically expands the underlying array when it has become full.

1. Explain the simplest natural way to make class DoubleArrayList threadsafe so it can be used from multiple concurrent threads.
2. Discuss how well the threadsafe version of the class is likely to scale if a large number of threads call `get`, `add` and `set` concurrently.
3. Now somebody suggests to improve scalability by introducing a separate lock for each method, roughly as follows:

```
private final Object sizeLock = new Object(), getLock = new Object(),
  addLock = new Object(), setLock = new Object(), toStringLock = ...;
public boolean add(double x) {
  synchronized (addLock) {
    if (size == items.length) {
      ...
    }
    items[size] = x;
    size++;
    return true;
  }
}
public double set(int i, double x) {
  synchronized (setLock) {
    if (0 <= i && i < size) {
      double old = items[i];
      items[i] = x;
      return old;
    } else
      throw new IndexOutOfBoundsException(String.valueOf(i));
  }
}
```

   Does this achieve thread safety? Explain why not. Does it achieve visibility? Explain why not.

4. Is there some version of the proposed (but flawed) locking scheme that actually achieves thread safety?

Question 3 (5 %):

Consider the extended class DoubleArrayList in TestLocking2.java. Like the class in the previous question it implements an array list of numbers, but now also has a static field `totalSize` that maintains a count of all the items ever added to any DoubleArrayList instance, and a static field `allLists` that contains a hashset of all the DoubleArrayList instances created. There are corresponding changes in the `add` method and the constructor.

1. Explain how one can make the class threadsafe enough so that the `totalSize` field is maintained correctly even if multiple concurrent threads work on multiple DoubleArrayList instances at the same time. You may ignore the `allLists` field for now.
2. Explain how one can make the class threadsafe enough so that the `allLists` field is maintained correctly even if multiple concurrent threads create new DoubleArrayList instances at the same time.

Part 2: Pipeline sort of a sequence of numbers

The second major part of this exam concerns sorting of a sequence of floating-point numbers (type `double`). Assume that you are faced with this problem, and that you have a large computer with many processor cores. Your notorious colleague Ulrik Funder helpfully suggests to put all the processor cores to work by sorting the numbers using a sorting pipeline.

A *sorting pipeline* consists of one or more stages. The first stage (to the left) receives the sequence of numbers to sort, and the last stage (to the right) outputs the sorted sequence. Each stage maintains a collection of the numbers it has received but not yet output or forwarded; this collection has a limited size. When a stage receives a new number `x` from the left, and its collection is not yet full, it just puts `x` in the collection. If the collection is full, and `x` is smaller than the smallest number in the collection, it forwards `x` to the next stage; otherwise it takes the smallest number from the collection and forwards that to the next stage, and puts `x` into the collection instead.

When all input numbers have been processed by the pipeline in this way, the first stage will contain the largest numbers, the second stage will contain the next largest numbers, and the last stage will contain the smallest numbers. The resulting sorted sequence can be "washed out" of the pipeline (that is, emitted by the last stage) by feeding sufficiently many infinitely large numbers into the first stage of the pipeline.

Figure 1 shows a small sorting pipeline with 2 stages each having an internal collection of size 4. The pipeline is shown after the first stage has received the input sequence $4, 7, 2, 8, 6, 1$. At this point the first stage contains the items $4, 6, 7, 8$ and has forwarded the items $2, 1$ to the second stage in that order.

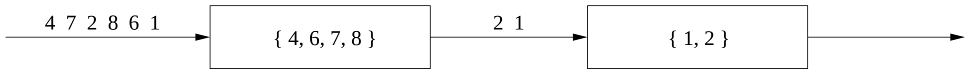

Figure 1: A sorting pipeline with two stages each having collection size four. The first number in the input sequence on the left is 4, the next 7, and so on.

If the first stage now receives number 5, it will replace 4 with 5 in its collection and forward 4 to the second stage. If it then receives number 3, it will forward it directly to the second stage (because the stage collection is full *and* 3 is smaller than all elements already in the stage). After that, the stage collections contain $5, 6, 7, 8$ and $1, 2, 3, 4$ respectively. Feeding eight infinitely large numbers into the first stage will cause the second stage to output the sorted sequence $1, 2, 3, 4, 5, 6, 7, 8$ in this order.

In general a sorting pipeline has `P>=1` stages each with collection size `S>=1`; above `P=2` and `S=4`. Such a pipeline can sort a sequence with up to `P*S` numbers.

As you can see from the description above, each state needs to efficiently find the smallest element in its `S`-element collection, and possibly replace it by another one. For this purpose we use a *minheap*, stored in an `S`-element array `heap[0..S-1]`. When there are `k` valid elements in a minheap `heap[0..k-1]`, it has the invariant property that `heap[(i-1)/2] <= heap[i]` for all `i=0..k-1`. It follows that `heap[0]` is the smallest element in the heap.

To insert an additional element `x` in a `k`-element minheap, store `x` in `heap[k]` and apply a "sift up" operation to restore the invariant; see method `minheapSiftup` in file SortingPipeline.java. To remove the least element and replace it with `x`, put `x` in `heap[0]` and "sift down" from `heap[0]` to restore the invariant; see method `minheapSiftdown` in file SortingPipeline.java.

In the exam questions below you will be asked to do the following:

- You must implement the sorting pipeline using threads, one thread for each stage, where two neighbour threads communicate through a blocking queue.
- You must make a number of different blocking queue implementations for the above one-thread-per-stage implementation.
- You must implement the sorting pipeline using actors and message passing, in the Java Akka framework.
- You must implement the sorting pipeline using Java 8 streams.

Interface for a queue of numbers

A first-in first-out queue of floating-point numbers is described by this interface:

```
interface BlockingDoubleQueue {
  double take();
  void put(double item);
}
```

The operations are intended to be blocking: if the queue is empty then a call to `take` will not return until another thread has put an element in the queue; and if the queue is full, then a call to `put` will not return until another thread has taken an element from the queue.

The BlockingDoubleQueue interface is defined in file SortingPipeline.java.

Question 4 (10 %):

Create the general sorting pipeline setup, as illustrated in Figure 2.

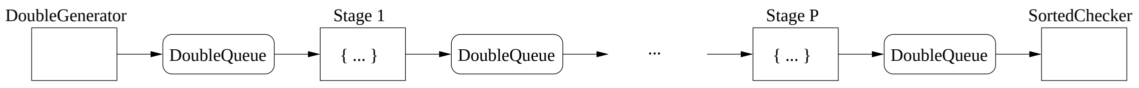

Figure 2: A parallel sorting pipeline with an input sequence generator, `P` sorting stages, and an output checker. There are `P+2` threads (shown as rectangles) connected by `P+1` queues (shown as rounded rectangles).

The first and last threads in the figure are implemented by classes DoubleGenerator and SortedChecker in file SortingPipeline.java.

1. To implement the sorting stages you must declare a class SortingStage; it must implement Runnable so that you can create a Thread object from it. A SortingStage instance needs to know (a) its input and output queues, (b) the size `S` of its internal collection, and (c) how many items it needs to produce as output before it terminates. The latter is necessary only to achieve termination of the sorting stage threads. Sorting stage `i`, where `1<=i<=P`, must process all the `N` numbers from the input plus the number of infinites needed to flush out the results from the subsequent stages: `itemCount = N+(P-i)*S` numbers in total.

   The action of a SortingStage thread should follow this pseudo-code outline:

```
while (itemCount > 0) {
  double x = ... get next number from input ...
  if (heapSize < heap.length) { // heap not full, put x into it
    heap[heapSize++] = x;
    DoubleArray.minheapSiftup(heap, heapSize-1, heapSize-1);
  } else if (x <= heap[0]) {    // x is small, forward
    output x
    itemCount--;
  } else {                      // forward least, replace with x
    double least = heap[0];
    heap[0] = x;
```

```
          DoubleArray.minheapSiftdown(heap, 0, heapSize-1);
          output least
          itemCount--;
        }
      }
```

2. Declare a method `sortPipeline(arr, P, queues)` that receives an array of `P+1` new queues and then creates the `P+2` threads. It must pass the relevant queue instances to each thread, pass the array `arr` to the DoubleGenerator instance, start all the threads, and finally wait for the threads to complete.

Question 5 (10 %):

The next step is to implement a (bounded) blocking queue so that you can try out your pipeline.

1. Declare a class WrappedArrayDoubleQueue that implements the BlockingDoubleQueue interface, as a wrapper around Java's thread-safe ArrayBlockingQueue<Double>. Basically you need to discard the exceptions that may be thrown by the ArrayBlockingQueue<Double> methods. Give the queue a capacity of 50 numbers. Show the code.
2. Now create a 4-stage pipeline to sort 40 numbers, and run it with `DEBUG` in class SortedChecker set to `true`. Show the output.
3. Measure the time to sort an array of 100,000 numbers using a pipeline with `P=4` stages, connected by WrappedArrayDoubleQueue instances. Thus each stage collection should have size `S=25000`. Here and in all later questions, `DEBUG` in SortedChecker should be `false`. Here and in later questions, use the microbenchmark measurement infrastructure presented in the course. Record also system information, number of cores, and so on. Show and discuss the results.

Question 6 (10 %):

Now implement your own thread-safe fixed-size bounded blocking queue.

1. Declare class BlockingNDoubleQueue, a blocking queue with a capacity of 50 numbers. Use locks for thread safety. The class must implement interface BlockingIntegerQueue.

   Use a `double[]` array for storing the (up to 50) numbers, and indexes `head` (for taking) and `tail` (for putting) numbers in the array.

   Show the code for BlockingNDoubleQueue, with some comments.
2. Explain why your BlockingNDoubleQueue can be safely used by multiple threads.
3. Measure the time to sort an array of 100,000 numbers using a pipeline with `P=4` sorting stages, connected by BlockingNDoubleQueue instances. Show and discuss the results.

Question 7 (10 %):

Now implement your own thread-safe *unbounded* blocking queue.

1. Declare class UnboundedDoubleQueue, a blocking queue with unbounded capacity. Use locks for thread safety. The class must implement interface BlockingDoubleQueue.

   Store the queue elements in a linked list of nodes, each holding one `double` and a reference to the next node (or `null`). Maintain references `head` and `tail` to the nodes at which taking and putting happen.

   Hint: The code becomes simpler if you use a sentinel (or dummy) node so that `head` and `tail` are never `null`.

   Show the code for UnboundedDoubleQueue, with some comments.
2. Explain why your UnboundedDoubleQueue can be safely used by multiple threads.
3. Measure the time to sort an array of 100,000 numbers using a pipeline with `P=4` sorting stages, connected by UnboundedDoubleQueue instances. Show and discuss the results.

Question 8 (10 %):

Observe that for a queue used in a sorting pipeline, only one thread calls the `put` method, and only one thread calls the `take` method. Therefore it is possible to implement a sufficiently thread safe queue without locks, as shown by the WaitFreeQueue class in Herlihy and Shavit chapter 3.

But note that Herlihy and Shavit's WaitFreeQueue's `put` and `take` methods return immediately if the queue is full (resp. empty). Instead your implementation of method `put` should spin—that is, execute a while loop—so long as the queue is full, and similarly method `take` should spin so long as the queue is empty. That is, your `put` method should return only when it successfully put something into the queue, and your `take` method should return only when it successfully took something from the queue.

1. Implement your own thread-safe NolockNDoubleQueue, a blocking queue with a capacity of 50 numbers. The class must implement interface BlockingDoubleQueue. You should use no locks. Show the code.
2. Explain the reason for each `volatile` and `final` modifier used in your implementation.
3. Explain why the absence of locks prevents you from using `wait` and `notify` to wait for the queue becoming non-full respectively non-empty.
4. Explain why your NolockNDoubleQueue can be safely used by two different threads, provided only one of them calls `put` and only one of them calls `take`. Explain in particular why all relevant memory writes performed by one thread are visible to the other thread.
5. Now delete any `volatile` modifiers you may have used in the queue implementation. Run the sorting pipeline (possibly with `DEBUG` in SortedChecker set to `true`) and describe what happens. Discuss why it happens.
6. Measure the time to sort an array of 100,000 numbers using a pipeline with `P=4` sorting stages, connected by NolockNDoubleQueue instances. Show and discuss the results.
7. Similarly, measure the time to sort an array of 100,000 numbers when using a pipeline with `P=2` sorting stages. Also measure the time when using a pipeline with `P=8` sorting stages. Remember to adapt the stage collection size when you change the number of stages. Show and discuss the results.

Question 9 (10 %):

Now implement a thread-safe lock-free unbounded blocking queue, as a variant of the Michael-Scott queue.

1. Declare a class MSUnboundedDoubleQueue, a blocking queue with unbounded capacity. Do not use locks for thread safety; instead make a variant of the Michael-Scott lockfree queue, using a linked list of nodes to store the elements (as in the UnboundedDoubleQueue above), and use compare-and-swap operations when manipulating the node lists. The class must implement interface BlockingDoubleQueue.

   As in the NolockNDoubleQueue, use spinning to ensure that the `take` method does not return until its has successfully taken a number from the queue. (Since the queue is unbounded the `put` method should not block).
2. Explain why your MSUnboundedDoubleQueue can be safely used by multiple threads.
3. Measure the time to sort an array of 100,000 numbers using a pipeline with `P=4` sorting stages, connected by MSUnboundedDoubleQueue instances. Show and discuss the results.

Question 10 (5 %):

Now implement your own thread-safe bounded blocking queue using transactional memory instead of locks.

1. Declare class StmBlockingNDoubleQueue, a blocking queue with a capacity of 50 numbers, using transactional memory (via the Multiverse library) but no locks. The class must implement interface BlockingDoubleQueue. Apart from not using locks, the implementation should be fairly similar to that of class BlockingNDoubleQueue.

   Show the code for StmBlockingNDoubleQueue, with some comments.

2. Explain why your StmBlockingNDoubleQueue can be safely used by multiple threads.
3. Measure the time to sort an array of 100,000 numbers using a pipeline with `P=4` sorting stages, connected by StmBlockingNDoubleQueue instances. Show and discuss the results.

Question 11 (10 %):

Implement the sorting pipeline without explicit use of threads or shared mutable memory, instead using message passing and the Java Akka library. Each sorting stage (as well as the output consumer) should be an actor. Each stage works by accepting messages containing a number from the previous stage and sending messages containing a number to the next stage.

The code below shows how a sorting pipeline with `P=2` stages, each having an internal collection size `S=4`, can be programmed in Erlang. For simplicity, the internal collection of a stage is here represented by a sorted list; using a minheap instead would give (much) better performance when the collection is large.

```
-module(helloworld).
-export([start/0,sorter/2,echo/0]).

add(N,L) ->
  lists:sort(lists:merge(L, [N])).

sorter(L, Out) ->
  receive
    {init, Pid} -> sorter(L, Pid);
    {num, N} when length(L) < 4 ->
      sorter(add(N, L), Out);
    {num, N} ->
      [Smallest|Rest] = add(N, L),
      Out ! {num, Smallest},
      sorter(Rest, Out)
  end.

echo() ->
  receive {num, N} ->
    io:write(N),
    echo()
  end.

transmit([], _) -> done;
transmit([N|L], Pid) ->
  Pid ! {num, N},
  transmit(L, Pid).

start() ->
  First = spawn(helloworld, sorter, [[], none]),
  Second = spawn(helloworld, sorter, [[], none]),
  Echo = spawn(helloworld, echo, []),
  First ! {init, Second},
  Second ! {init, Echo},
  transmit([4,7,2,8,6,1,5,3], First), % input
  transmit([9,9,9,9,9,9,9,9], First). % washout
```

Figure 3 shows the communication diagram for the sorting pipeline working on the 8-element input sequence $4, 7, 2, 8, 6, 1, 5, 3$.

1. Use the Java Akka library to implement the two-stage actor-based sorting pipeline, sorting eight numbers, as shown in Erlang above. You may use a sorted list or a minheap as you like. Show the code with some comments.
2. Run the pipeline and demonstrate that it produces the correct result.

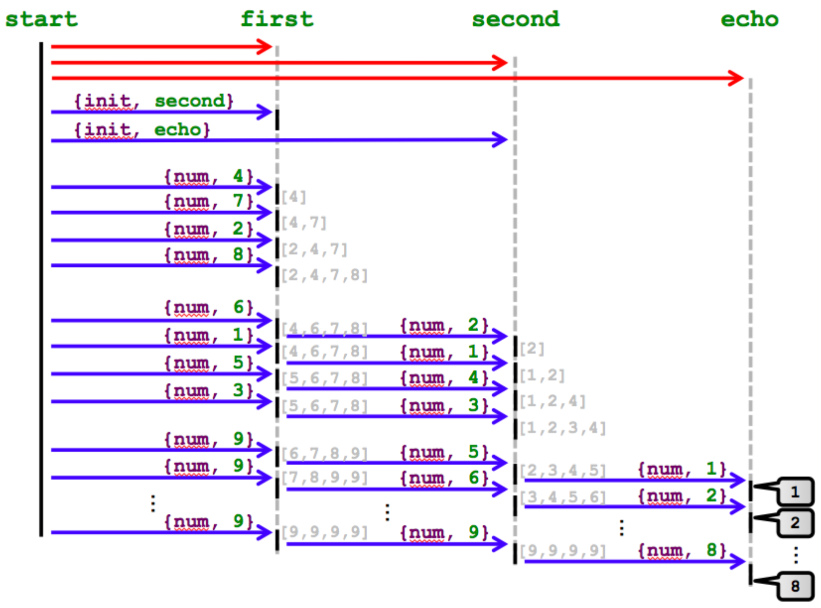

Figure 3: Communication diagram for the pipeline sort.

Question 12 (10 %):

A sorting pipeline can be also implemented using Java 8 streams (in the sense of course week 3), as follows. Each sorting stage is implemented using a function `stage` of type DoubleFunction<DoubleStream>; it takes an incoming number and returns either a zero-element DoubleStream (no output) or a one-element DoubleStream (a single number output). As in the thread-based implementation, each `stage` instance maintains a minheap of size `S` containing the largest elements it has seen. A pipeline is then built by flatmapping each stage onto its input DoubleStream, thus producing an output DoubleStream.

1. Implement a three-stage sorting pipeline using this approach, and apply it to a DoubleStream created from an unsorted `double` array. Do not attempt to parallelize the stream. Show the code you have written. Show the output from applying it to the result of DoubleArray.randomPermutation(60).

2. Try to apply the three-stage stream-based sorting pipeline to a *parallel* DoubleStream with 100,000 elements and print the first 500 numbers from the output. Does the result look sorted? Why doesn't it work to parallelize such a Java 8 stream-based sorting pipeline? What dogma about Java 8 stream processing is violated by your implementation of the sorting stages?