Examination, Practical Concurrent and Parallel Programming 7–8 January 2015

These exam questions comprise 9 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 Wednesday 7 January 2015 at 09:00 local time.

Your solution must be handed in no later than **Thursday 8 January 2015 at 22:00** 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/en/SDT/Your-Programme/Forms

There are 11 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.

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 form of your solution

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.

Background: Quicksort of an array of integers

The exam questions below concern the implementation in Java of parallel quicksort of an array of integers.

Standard sequential recursive quicksort of an array segment `arr[a..b]` works like this: Either `a>=b` so the segment has zero or one items and is already sorted. Or else `a<b` so the segment has at least two items, and we choose a pivot element `x` in the array segment, move items smaller than `x` to the left in the array and items greater than `x` to the right, where the left and right segments are separated by `x`, then sort the left and right segments recursively; when this is done the whole segment `arr[a..b]` obviously is sorted too. Note that the left and right array segments do not overlap.

Note that this is an in-place sorting algorithm. It modifies the given array `arr` so that the final sorted result is in the same place as the given unsorted data.

Sequential recursive quicksort

Pseudocode for sequential recursive quicksort might look like this:

```
static void qsort(int[] arr, int a, int b) {
  if (arr[a..b] contains at least 2 items) {
    choose pivot element x in arr[a..b]
    partition into arr[a..j] and x and arr[i..b]
              where arr[a..j] contains items <= x
              and   arr[i..b] contains items >= x
    recursively sort segment arr[a..j]
    recursively sort segment arr[i..b]
  }
}
```

The real Java code for sequential recursive quicksort of `arr[a..b]` is this:

```
private static void qsort(int[] arr, int a, int b) {
  if (a < b) {
    int i = a, j = b;
    int x = arr[(i+j) / 2];
    do {
      while (arr[i] < x) i++;
      while (arr[j] > x) j--;
      if (i <= j) {
        swap(arr, i, j);
        i++; j--;
      }
    } while (i <= j);
    qsort(arr, a, j);
    qsort(arr, i, b);
  }
}
```

where method `swap(arr, s, t)` swaps items `arr[s]` and `arr[t]`. To sort the entire array `arr`, initially call `qsort(arr, 0, arr.length-1)`.

In essence, recursive quicksort uses the method call stack to keep track of which array segments, such as `arr[a..j]` and `arr[i..b]`, that still need to be sorted.

If you need more information about sequential recursive quicksort, see the lecture note *Searching and sorting with Java*, section 5, at http://www.itu.dk/people/sestoft/programmering/sortering.pdf, or an algorithms textbook.

Towards a parallelizable quicksort

The above recursive version of quicksort is fast but necessarily sequential. To get closer to a parallelizable version of it, we introduce a double-ended queue (deque) called `queue` to keep track of the array segments that still need to be sorted. Instead of making a recursive call such as `qsort(arr,a,j)` we push a SortTask object on the queue that describes the segment `arr[a..j]` that needs to be sorted.

The overall implementation of queue-based quicksort now is very simple. It repeatedly pops a SortTask describing a segment `arr[a..b]` from the queue. Either `a>=b` so the segment has zero or one items and is already sorted. Or else `a<b`, so choose a pivot element `x`, partition in small items `arr[a..j]` and large items `arr[i..b]`, and push SortTask objects describing these onto the queue. Then start over popping a SortTask from the queue until the queue is empty and `queue.pop()` returns `null`.

The Java code for *single-queue single-thread* (SQST) quicksort is very similar to the above recursive one:

```
private static void sqstWorker(Deque<SortTask> queue) {
  SortTask task;
  while (null != (task = queue.pop())) {
    final int[] arr = task.arr;
    final int a = task.a, b = task.b;
    if (a < b) {
      int i = a, j = b;
      int x = arr[(i+j) / 2];
      do {
        while (arr[i] < x) i++;
        while (arr[j] > x) j--;
        if (i <= j) {
          swap(arr, i, j);
          i++; j--;
        }
      } while (i <= j);
      queue.push(new SortTask(arr, a, j));
      queue.push(new SortTask(arr, i, b));
    }
  }
}
```

Note that there is no recursion but instead a queue and a new outer while-loop. Apart from that, the code is identical to the previous one. What we have is a *worker*, running on the main thread, that repeatedly pops a task from the queue and executes that task, possibly pushing new subtasks onto the queue.

To sort the entire array `arr`, initially push `new SortTask(arr, 0, arr.length-1)` onto the `queue`, then call `sqstWorker(queue)`.

The SortTask class just contains the values `arr`, `a` and `b` as one would expect:

```
class SortTask {
  public final int[] arr;
  public final int a, b;
  public SortTask(int[] arr, int a, int b) {
    this.arr = arr;
    this.a = a;
    this.b = b;
  }
}
```

The queue interface and class

A double-ended queue (deque) is described by the Deque<T> interface:

```
interface Deque<T> {
  void push(T item);     // at bottom
  T pop();               // from bottom
  T steal();             // from top
}
```

The two ends of a deque are here called its bottom and its top. The `push` operation adds an item to the bottom of the deque; the `pop` operation removes an item from the bottom of the deque; and the `steal` operation removes an item from the top of the deque. Using `push` and `pop` together makes the deque behave like a stack.

The `pop` and `steal` methods do not block when the deque is empty, they just return `null`. The `push` operation does not block when the queue is full, but throws an exception.

A Deque<T> may be implemented as a cyclic array `items` of type `T[]`, with two indices `bottom` and `top`. Such an implementation SimpleDeque<T> is included in the source file `Quicksorts.java` provided for this exam, along with all the other code shown here.

Making a parallel quicksort

Given the single-queue single-thread version of quicksort, it is now fairly clear how one could parallelize it. Instead of letting the main thread perform all the work, you create multiple worker threads, where each worker thread repeatedly pops a SortTask from the queue and possibly pushes new SortTasks to the queue.

This is what the exam questions below asks you to do:

- Make the deque implementation threadsafe using locks. Later, test it.
- Implement a parallel quicksort using multiple worker threads and a single shared deque. Measure speed and scalability of this solution.
- Implement a parallel quicksort using multiple worker threads and multiple queues, one queue for each thread. A thread uses its own queue most of the time, but if it runs out of work, it may steal work from the queues of other threads. Measure speed and scalability of this solution.
- Implement a lock-free work-stealing queue. Later test it. Use it in the parallel quicksort with multiple worker threads and multiple queues, one for each thread. Measure speed and scalability of this solution.

Question 1 (5 %):

The given deque implementation in class SimpleDeque<T> in file `Quicksorts.java` is not threadsafe. Use locking to make it threadsafe so that `push` and `pop` and `steal` may be called concurrently from multiple worker threads.

Explain exactly what modifications you make and explain why they make the class threadsafe.

Describe where you would add `@GuardedBy` annotations, what they would look like, and what they are useful for?

Question 2 (5 %):

Explain why class `SortTask` is threadsafe.

Question 3 (20 %):

Create a *single-queue multi-thread* (SQMT) version of quicksort. Write a method

```
static void sqmtWorkers(Deque<SortTask> queue, int threadCount)
```

that creates and starts `threadCount` threads, where each thread behaves basically like the `sqstWorker` shown previously. Each thread must repeatedly pop a SortTask representing an array segment `arr[a..b]` from the shared work queue, test whether `a<b` and if so do the partitioning and push two new tasks on the work queue.

A new complication is to decide when each of the worker threads should terminate. A thread should not terminate just because the work queue becomes empty so that `pop` returns `null`: Another task may be working on a partitioning and then push new sorting tasks to the queue just a moment later. To make sure that tasks do not terminate too early, you may maintain a counter (a Java 8 LongAdder is best, else an AtomicLong), shared between the threads. The counter holds the number of sort tasks pushed to the queue but not yet processed. The counter should be incremented by one every time a new sort task is pushed on the queue (both the initial task and those pushed after the do-while loop) and decremented by one when a thread has finished processing a sort task from the queue, that is, just before the end of the outer while loop.

Then, whenever a thread finds that `pop` returns `null`, it checks whether the counter (LongAdder) is greater than zero and in that case calls `Thread.yield()`; otherwise the thread terminates by returning from the thread's `run` method.

It is best to encapsulate this logic in a separate method `getTask`, so that `getTask` returns `null` only if there really is no more work to do:

```
private static SortTask getTask(final Deque<SortTask> queue, LongAdder ongoing) {
  SortTask task;
  while (null == (task = queue.pop())) {
    if (ongoing.longValue() > 0)
      Thread.yield();
    else
      return null;
  }
  return task;
}
```

and then call this method instead of `queue.pop()` in the worker thread's while-loop's condition:

```
while (null != (task = getTask(queue, ongoing))) {
  ...
}
```

We must also decide on the capacity of the queue (because it throws an exception if it runs full). Experimentally it seems that a queue capacity of 100,000 is large enough to sort an array of 100 million random integers or more.

Show your implementation of the `sqmtWorkers` method, explain why it works. In particular, explain why it is safe for multiple threads to work concurrently (1) on the same queue object, and (2) on the same array object.

Also, use the setup to sort a small array, and show the input and output. You can use method `printout` from class IntArrayUtil in file `Quicksorts.java` to print the array before and after sorting.

Question 4 (15 %):

Write test cases for the locking-based SimpleDeque<T> implementation from Question 1, both precise sequential functional tests (for instance, that if you push an item and then immediately pop, you get the same item back), and approximate concurrent tests, as discussed in course week 9.

In the concurrent test you should create a single queue instance and then have multiple threads calling `push` and multiple threads calling `pop` or `steal` on that instance. For example, you may push one million random Integers, and concurrently pop or steal one million random Integers, and afterwards check that the sum of the pushed numbers equals the sum of the popped or stolen numbers.

Try to avoid overflowing the queue (which will throw an exception and ruin the test). You may attempt this by running fewer pushing threads than popping and stealing threads.

Use a CyclicBarrier to make sure all the testing threads are ready to start at the same time, and use a CyclicBarrier to make sure all the testing threads terminate before checking the results.

Show your test code, explain what it does, explain what parameters (how many threads, how many pushes, and so on) you run it with, and show results of running it.

Then use mutation of the SimpleDeque<T> implementation to find out how good your test cases are. Explain what parts of the implementation you mutate and whether your tests discover the faults that you add to the implementation.

Question 5 (10 %):

Measure the wall-clock execution time taken for the entire sorting process in the single-queue multi-thread setup (Question 3), for 1, 2, ..., 8 threads, or even more threads if the machine you use has more than four cores. Measure only the time to sort the array, not the time it takes to create and fill the array with random integers.

Make sure that the entire computation runs for at least 5 seconds; this may require sorting an array of 10 million or 100 million integers. You may use a Timer instance instead of the methods (Mark7 or similar) from the *Microbenchmarks* lecture note. Use a CyclicBarrier to make sure all worker threads start at the same time, and to wait for all worker threads to terminate.

Show your measurement code and explain how it works. Make a table of the wall-clock execution times as a function of the number of threads.

Question 6 (10 %):

Create a *multi-queue multi-thread* (MQMT) version of quicksort. Write a method

```
static void mqmtWorkers(Deque<SortTask>[] queues, int threadCount)
```

that creates and starts `threadCount` threads, where each thread behaves almost like the worker threads you wrote for Question 3. The difference is that now each thread has its own deque, so that thread number `t` will push and pop from `queues[t]`. Note that `queues.length` must equal `threadCount`.

If thread number `t` runs out of tasks, that is, `queues[t].pop()` returns `null`, then the thread should try to steal a task from one of the queues belonging to the other threads.

More precisely, use a counter (LongAdder) to count unfinished sort tasks as in the single-queue multi-thread setup. If a thread finds that its own queue's `pop` returns `null`, it tries to `steal` from the queues of the other threads, and if all of these return `null` as well, it checks the LongAdder, and if that is greater than zero, calls `Thread.yield()` and afterwards then tries to steal again. (There is no point in trying to `pop` from its own queue again, because no other thread pushes to that queue). If the LongAdder is zero, the thread terminates by returning from its `run` method.

As before, it is best to encapsulate this logic in a separate `getTask` method:

```
private static SortTask getTask(final int myNumber, final Deque<SortTask>[] queues,
                                LongAdder ongoing) {
  SortTask task = queues[myNumber].pop();
  if (null != task)
    return task;
  else {
    do {
      ... try to steal from other threads ...
      Thread.yield();
    } while (ongoing.longValue() > 0);
    return null;
  }
}
```

Show your implementation of `mqmtWorkers` and `getTask`, and explain why they work. Also run it on a small example, and show the input and output.

Question 7 (5 %):

Measure the wall-clock execution time taken for the entire sorting process in the multi-queue multi-thread (Question 6) setup, for 1, 2, . . . , 8 threads, or even more threads if the machine you are using has more than four cores. Measure only the time to sort the array, not the time it takes to create and fill the array with random integers.

Make sure that the entire computation runs for at least 5 seconds; this may require sorting an array of 10 million or 100 million integers. As before, use a Timer instance and CyclicBarrier.

Show your measurement code and explain how it works. Make a table of the wall-clock execution times as a function of the number of threads.

Compare the execution times of the single-queue multi-thread setup and the multi-queue multi-thread setup. Which one is faster? Which one scales better on your hardware?

Question 8 (10 %):

In the multi-queue multi-thread setup (Question 6), only the thread that owns a queue calls `push` and `pop` on that queue (whereas other threads may call `steal`). Moreover, `push` and `pop` operations happen at one end of the queue (the bottom), whereas `steal` operations happen at the other end (the top). This enables a lock-free version of the work-stealing queue, as proposed by Arora, Blumofe and Plaxton around 1999, and further improved by Chase and Lev in 2005. The idea is that the very frequent thread-confined `push` and `pop` operations can be performed without locks and (almost always) without compare-and-swap operations, whereas the less frequent `steal` operation can be implemented using compare-and-swap and still no locks.

Here is the pseudocode for a Chase-Lev work-stealing queue (simplified to not dynamically grow the `items` array):

```
class ChaseLevDeque<T> implements Deque<T> {
  long bottom = 0, top = 0;
  T[] items;
  ... constructor and auxiliary methods omitted ...

  public void push(T item) { // at bottom
    final long b = bottom, t = top, size = b - t;
    if (size == items.length)
      throw new RuntimeException("queue overflow");
    items[index(b, items.length)] = item;
    bottom = b+1;
  }

  public T pop() { // from bottom
    final long b = bottom - 1, t = top, afterSize = b - t;
    bottom = b;
    if (afterSize < 0) { // empty before call
      bottom = t;
      return null;
    } else {
      T result = items[index(b, items.length)];
      if (afterSize > 0) // non-empty after call
```

```
          return result;
        else { // became empty, update both top and bottom
          if (!CAS(top, t, t+1)) // somebody stole result
            result = null;
          bottom = t+1;
          return result;
        }
      }
    }

    public T steal() { // from top
      final long b = bottom, t = top, size = b - t;
      if (size <= 0)
        return null;
      else {
        T result = items[index(t, items.length)];
        if (CAS(top, t, t+1))
          return result;
        else
          return null;
      }
    }
  }
```

This pseudocode is simplified from sections 1 and 2.1-2.2 in the paper Chase, Lev: *Dynamic circular work-stealing deque*, 2005, which is available from http://dl.acm.org/citation.cfm?doid=1073970.1073974 provided you are inside the IT University's network. There is also a copy of the paper on the course's LearnIT site.

The Chase-Lev paper builds on previous work in section 3 of Arora, Blumofe, Plaxton: *Thread scheduling for multiprogrammed multiprocessors*, 2001. You should not need to look at this paper, but if you are curious, it is here: http://www.csd.uwo.ca/~moreno/CS433-CS9624/Resources/arora-blumofe-toc01.pdf

In the pseudocode above (and in the Arora and Chase papers), the `steal` operation may occasionally return `null` even though the queue is non-empty, namely, when a concurrent `pop` operation updates the `top` index and causes the CAS to fail. This is not a problem in our sorting application, because the worker thread will just retry the `steal` operation a moment later, and possibly succeed.

Program the lock-free deque in Java as a class ChaseLevDeque<T> that implements the Deque<T> interface. As shown by the pseudocode, the `top` field is shared between threads and will be updated using compare-and-swap and hence must be an AtomicLong. The `bottom` field is updated only by one thread (but read by others) and should just be a `long`. Think about where to use `final` and `volatile`.

Show your real-Java implementation of ChaseLevDeque<T>, and explain in your own words why it works. In particular, explain visibility: Why are updates to the `top` and `bottom` indexes visible across threads, and why are updates to the `items` array's elements visible across threads.

Question 9 (10 %):

Test that your ChaseLevDeque<T> works as intended. You may use the same sequential (single-threaded) test setup as in Question 4, but the concurrent test needs to be different. In particular, you must have only one thread that performs `push` and `pop` operations on the ChaseLevDeque<T> instance being tested, but multiple other threads may perform concurrent `steal` operations on it.

Show your test code and explain what it does. Describe the parameters you use (number of concurrent threads and so on) and show the output from the test.

Question 10 (5 %):

Implement a version of the multi-queue multi-thread (Question 6) setup that uses the ChaseLevDeque<T> instead of the SimpleTaskDeque<T>. You should be able to reuse the `mqmtWorkers` method without any change because the two queues implement the same Deque<T> interface.

Show the code and the input and output from an example running of it.

Question 11 (5 %):

Measure the time taken for the entire Question 10 sorting process in the multi-queue multi-thread setup with ChaseLevDeque<T>, using P=1..8 threads, or more threads if the machine you use has more than four cores. Make a table of the wall-clock execution times as a function of the number of threads.

Show your measurement code and explain how it works. Make a table of the wall-clock execution times as a function of the number of threads.

Compare the execution times of the previous setups. Which one is faster? Which one scales better on your hardware?

Additional information (not part of the exam)

- The lock-free work-stealing queue in Questions 8–11 is a simplified version of the work-stealing queues used inside the Java's fork-join framework, Java 8's newWorkStealingPool, the Akka library, and the .NET Task Parallel Library. In other words, it is central infrastructure in modern concurrency frameworks.
- A more ambitious and more efficient version of the lock-free work-stealing queue would grow the underlying array dynamically, and perform many other improvements as suggested in the Chase-Lev paper mentioned in Question 8.
- Our notion of sort task (class SortTask) is very simple compared to tasks in the above-mentioned frameworks. Basically our sort task is "fire and forget": there is no way to check whether a sort task has completed. This is the reason we need the counter (LongAdder) for unfinished tasks discussed in Question 3, so we can determine when all worker threads are idle.
- On a single-core machine the parallel quicksorts developed in Questions 3 and 6 and 10 are likely to be twice as slow as the sequential recursive quicksort (whose performance you are not asked to measure) described in the background section. This slowdown is likely because of the overhead of creating all the SortTask objects, and doing all the pushing and popping of these objects. The overhead can be substantially reduced by using a sequential recursive quicksort when the array segment is small (for instance, when `b-a<50`), instead of creating a new SortTask for parallel processing. This will drastically reduce the number of SortTask objects created and may considerably speed up the parallel quicksort. For simplicity, this improvement is not part of the exam questions.
- As always, a poor choice of pivot element `x` before quicksort's partitioning phase may lead to very poor performance and in addition may overflow the work deque. An industrial-strength implementation of parallel quicksort should make sure to choose the pivot element more carefully than the implementation studied here.