Chalmers | GÖTEBORGS UNIVERSITET
Sandro Stucki, Computer Science and Engineering

Principles of Concurrent Programming TDA384/DIT391

Tuesday, 19 March 2019, 8:30–12:30

(including example solutions)

Exam set and supervised by: Sandro Stucki (`sandros@chalmers.se`, 076 420 8639)
Examiner: K. V. S. Prasad (`prasad@chalmers.se`)

Material permitted during the exam (hjälpmedel):
Two textbooks; four sheets of A4 paper with notes; English dictionary.

Grading: You can score a maximum of 70 points. Exam grades are:

Points in exam	Grade Chalmers	Grade GU
28–41	3	G
42–55	4	G
56–70	5	VG

Passing the course requires passing the exam and passing the labs. The overall grade for the course is determined as follows:

Points in exam + labs	Grade Chalmers	Grade GU
40–59	3	G
60–79	4	G
80–100	5	VG

The exam **results** will be available in Ladok within 15 *working* days after the exam's date.

Instructions and rules:

- Please write your answers clearly and legibly: unnecessarily complicated solutions will lose points, and answers that cannot be read will not receive any points!
- Justify your answers, and clearly state any assumptions that your solutions may depend on for correctness.
- Answer each question on a new page. Glance through the whole paper first; there are five questions, numbered Q1 through Q5. Do not spend more time on any question or part than justified by the points it carries.
- Be precise. In your answers, try to use the programming notation and syntax used in the questions. You can also use pseudo-code, *provided* the meaning is precise and clear. If need be, explain your notation.

Q1. Concurrent counter array *(14p)*

The following Java class implements a thread-safe array of counters:

```java
class LockedCounterArray {
  private int[] counters;
  private Lock lock = new ReentrantLock();

  LockedCounterArray(int size) { counters = new int[size]; /* all 0s */ }

  void increment(int index) {
    lock.lock();
    counters[index] = counters[index] + 1;
    lock.unlock();
  }

  int value(int index) {
    lock.lock();
    int cnt = counters[index];
    lock.unlock();
    return cnt;
  }
}
```

The number of counters in the array is given by the constructor argument `size`; the method `increment()` safely increments the counter at a given `index`; the method `value()` returns the value of counter `index`. See Appendix A.1 for the full Java code listing.

(Part a) Assume we are given a counter array `lca = new LockedCounterArray(100)`. What is the maximum number of threads that can call the method `increment()` on `lca` concurrently without blocking? What is the maximum number of counters that could safely be incremented concurrently in principle? Justify your answers. *(2p)*

Solution of 1.a: the maximum number of concurrent, non-blocking calls to `increment()` is one because the class uses a single lock to guard all calls to `increment()`. In principle, every counter can be incremented independently without causing any data races, so the maximum number of concurrent non-blocking calls is 100.

(Part b) How could the implementation of `LockedCounterArray` be adjusted to allow a maximal number of threads concurrent access to the counter array? Propose a lock-based solution. Describe which fields and methods of the class need to be changed and how. You may describe the changes either in words or by giving concrete Java code for new/updated fields and methods. *(4p)*

Solution of 1.b: add one lock per array cell, i.e. replace the field `lock` with an *array* of locks. Update the constructor to initialize the array, and the `increment()` and `value()` methods to only lock/unlock the lock at `index`. Code example:

```java
  private Lock[] locks; // Use an array of locks
```

```
FineLockedCounterArray(int size) {
  counters = new int[size];
  locks = new Lock[size];  // Initialize locks
  for (int i = 0; i < size; ++i) { locks[i] = new ReentrantLock(); }
}
void increment(int index) {
  locks[index].lock();  // Only guard the counter we're about to increment
  counters[index] = counters[index] + 1;
  locks[index].unlock();
}
int value(int index) {
  locks[index].lock();  // Only guard the counter we're about to read
  int cnt = counters[index];
  locks[index].unlock();
  return cnt;
}
```

Eliminating locking overhead

Locking can introduce significant overhead in some situations. For the remainder of this question, assume that the size of `counters`, the number of threads trying to access the counters, and the number of processor cores all have the same value: 16.

(Part c) Explain what type of overhead your lock-based solution from Part b may suffer from under this assumption. How costly is locking compared to just incrementing the counter? Does it take orders of magnitude fewer CPU cycles, orders of magnitude more, or about the same? *(2p)*

Solution of 1.c: there is some overhead associated with the actual locking operations and a lot of overhead if a thread blocks and has to be rescheduled. This later scenario is likely only if counter increments are very frequent and contention becomes high. The counter increment is cheap (a few cycles) if the counter value is already in a local L1 or L2 cache; if the counter value needs to be fetched from main memory, there may be up to 100x slowdown. The locking operations will likely involve memory barriers/fences (depends on the lock implementation) which may be 1–2x slower than a plain access to main memory.[1] So a "typical" locking operation may introduce 2–100x overhead. If a thread blocks and needs to be rescheduled, the OS or runtime system gets involved and context switches become necessary. This will introduce much greater overhead (>1000x).

(Part d) How could the implementation of `LockedCounterArray` be adjusted to remove locking overhead while still allowing a maximal concurrent access to the counter array? Describe which fields and methods of the class need to be changed and how. You may assume that strong synchronization primitives/instructions of the type you have seen in the course are supported by the underlying machine architecture. *(4p)*

[1] see e.g. `http://sigops.org/s/conferences/sosp/2013/papers/p33-david.pdf`

Solution of 1.d: remove the field `lock` and use compare-and-set (CAS) operations instead when incrementing the counters: repeatedly attempt updating the individual counters until the CAS operation succeeds. Concretely, replace the `counters` array with an array of `AtomicIntegers`. Update the constructor to initialize the `AtomicIntegers` in the array, and the `increment()` method to repeatedly try to update the counter at `index` until the CAS succeeds. Code example:

```
private AtomicInteger[] counters; // no more locks!
LockfreeCounterArray(int size) {
  counters = new AtomicInteger[size];
  for (int i = 0; i < size; ++i) { counters[i] = new AtomicInteger(0); }
}
void increment(int index) {
  int cnt;
  do {
    cnt = counters[index].get();
  } while (!counters[index].compareAndSet(cnt, cnt + 1));
}
int value(int index) { return counters[index].get(); }
```

(Part e) How does the performance of your solutions from Part b and Part d compare? Which of the two solutions would you expect to perform better in the given scenario? Briefly motivate your answer. *(2p)*

Solution of 1.e: the lock-free solution should perform better. Though CAS is not free, it should be at least as efficient as a locking operation. Since increments execute quickly, we expect few conflicts and no or very few CAS-retries per thread for the lock-free solution. The lock-free solution also entirely avoids context switches.

Code and transition table for Q2 and Q3

The code below implements barrier synchronization for two threads t and u using a pair of semaphores. (See Appendix A.2 for the full Java program.)

```
      Semaphore[] barrier = { new Semaphore(0), new Semaphore(0) };

                 thread t                              thread u
1   boolean done = false;                  boolean done = false;           10
2   do {                                   do {                            11
3     // synchronize                         // synchronize                12
4     barrier[t].up();                       barrier[u].up();              13
5     barrier[u].down();                     barrier[t].down();            14
6     // do work                             // do work                    15
7     done = doWork();                       done = doWork();              16
8   } while (!done);                       } while (!done);                17
9   // t terminates                        // u terminates                 18
```

Below is an incomplete state transition table for this program. Each state consists of the current values of the program counters for both threads (pc_t, pc_u), the local variable `done` in each thread (d_t, d_u), and the barrier semaphores (b_i is the value of `barrier[`i`]`). The states in the table cover only a subset of program positions: statements that access the shared variable `barrier` (lines 4, 5 for t, and 13, 14 for u), and the ends of the threads (lines 9 and 18). Remember that $pc_t = 4$ means that the *next* line executed by t is 4.

current state s = (pc_t, pc_u, d_t, d_u, b_t, b_u)		$s'(t)$ = next state if t moves	$s'(u)$ = next state if u moves
s_1	(4, 13, F, F, 0, 0)	(5, 13, F, F, 1, 0) = s_2	(4, 14, F, F, 0, 1) = s_3
s_2	(5, 13, F, F, 1, 0)	no move	(5, 14, F, F, 1, 1) = s_4
s_3	(4, 14, F, F, 0, 1)	(5, 14, F, F, 1, 1) = s_4	no move
s_4	(5, 14, F, F, 1, 1)	(4, 14, F, F, 1, 0) = s_5	(5, 13, F, F, 0, 1) = s_6
		(9, 14, T, F, 1, 0) = s_{X1}	(5, 18, F, T, 0, 1) = s_{Y1}
s_5	(4, 14, F, F, 1, 0)	(5, 14, F, F, 2, 0) = s_7	(4, 13, F, F, 0, 0) = s_1
			(4, 18, F, T, 0, 0) = s_{Y2}
s_6	(5, 13, F, F, 0, 1)	(4, 13, F, F, 0, 0) = s_1	(5, 14, F, F, 0, 2) = s_8
		(9, 13, T, F, 0, 0) = s_{X2}	
s_7	(5, 14, F, F, 2, 0)	no move	(5, 13, F, F, 1, 0) = s_2
			(5, 18, F, T, 1, 0) = s_{Y3}
s_8	(5, 14, F, F, 0, 2)	(4, 14, F, F, 0, 1) = s_3	no move
		(9, 14, T, F, 0, 1) = s_{X3}	
s_{X1}	(9, 14, T, F, 1, 0)	no move	(9, 13, T, F, 0, 0) = s_{X2}
			(9, 18, T, T, 0, 0) = s_Z
s_{X2}	(9, 13, T, F, 0, 0)	no move	(9, 14, T, F, 0, 1) = s_{X3}
s_{X3}	(9, 14, T, F, 0, 1)	no move	no move
s_{Y1}	(5, 18, F, T, 0, 1)	(4, 18, F, T, 0, 0) = s_{Y2}	no move
		(9, 18, T, T, 0, 0) = s_Z	
s_{Y2}	(4, 18, F, T, 0, 0)	(5, 18, F, T, 1, 0) = s_{Y3}	no move
s_{Y3}	(5, 18, F, T, 1, 0)	no move	no move
s_Z	(9, 18, T, T, 0, 0)	no move	no move

Q2. Barriers – state transitions *(20p)*

The pseudo-code given on page 5 implements barrier synchronization for two threads t and u using semaphores (see Appendix A.2 for the complete Java program).

Before doing any work, the threads synchronize at the barrier. After synchronization, each thread performs some computation by calling the `doWork()` method. The method returns **`true`** to indicate that the thread is done and should terminate, or **`false`** to indicate that more work needs to be done. Every call to `doWork()` terminates but not every invocation takes the same amount of time to do so.

Page 5 also contains an incomplete state transition table modeling the behavior of the two threads. To keep the table small, only program positions relevant to the concurrent behavior of the program are tracked (see page 5 for details). Remember that $pc_t = 4$ means the *next* line executed by t is 4.

Note that a thread sometimes has *more than one move*: depending on the value of the variable `done`, a thread may execute another iteration of the **`while`** loop or terminate. For example, thread u has two possible moves from state s_5: to s_1 if `doWork()` returns **`false`**, or to s_{Y2} if it returns **`true`**.

Your job is to complete the state transition table and prove (or disprove) that the code is free from *deadlocks* and *starvation*.

(Part a) Fill in the missing fields in the transition table (rows s_1–s_8 and s_{X3}). Make sure your solution is consistent with the names given for the missing fields in rows s_1, s_3 and s_5. Don't write directly into the table on page 5. Instead, note the names and fields of the incomplete rows as $s_i : (X, Y, \dots),\ (U, V, \dots)\dots$ on a separate sheet of paper with the rest of your solutions. *(6p)*

Solution of 2.a: see completed table on page 5.

(Part b) List all the final states in the table. *(1p)*

Solution of 2.b: there is only one final state: s_Z, where both t and u have terminated.

(Part c) Are any of the fields in the state tuples redundant? Is it possible to remove any of the variables $pc_t, pc_u, d_t, d_u, b_t, b_u$ without collapsing two previously distinct states? *(2p)*

Solution of 2.c: the variables d_t and d_u are redundant since their information is already encoded in the values of pc_t and pc_u: the states named s_{Xi} are those where $d_t = T$ and $d_u = F$, those named s_{Yi} have $d_t = F$ and $d_u = T$, in s_Z we have $d_t = d_u = T$ and in all other states $d_t = d_u = F$.

Deadlock

A thread is *blocked* in state s if it cannot move from s even though it has not yet terminated. For example u is blocked in s_3.

(Part d) Can the program deadlock? In other words, is there a state where both threads are blocked? Prove your answer from the state transition table. If your answer is yes, also give a trace (i.e. a sequence of states starting from the initial state s_1) that exhibits the deadlock. *(1p)*

Solution of 2.d: there is no deadlock. States s_{X3}, s_{Y3} and s_Z are candidates because neither thread can move, but in each case at least one of the threads has terminated, so it is not trying to move.

Starvation

Remember that a thread is *starving* if it is blocked indefinitely while other threads can make progress. Formally, a t-starvation path is a finite, non-empty sequence of states

$$s^1 \longrightarrow s^2 \longrightarrow s^3 \longrightarrow \cdots \longrightarrow s^n$$

where t is blocked in every state s^i, while u moves from s^i to s^{i+1} and either terminates in s^n or *loops forever* on the path, i.e. it moves from s^n back to s^1 and starts over. A u-starvation path is an analogous sequence of states where only t makes progress. (Note that a path can be of length $n = 1$.)

(Part e) Can either of the threads starve? That is, is there a t- or u-starvation path? Prove your answer from the state transition table. If your answer is yes, also give a trace (i.e. a sequence of states starting from the initial state s_1) that exhibits the starvation (i.e. that ends in a starvation path). If your answer is no, briefly explain why there cannot be any t- or u-starvation paths. *(3p)*

Solution of 2.e: both threads can starve. The state s_{Y3} is a (singleton) t-starvation path since u has terminated and t is blocked. The only non-trivial t-starvation path (s_7, s_{Y3}) also ends in s_{Y3}. Similarly, (s_{X3}) and (s_8, s_{X3}) are the only u-starvation paths. Any trace that ends in one of these starvation paths exhibits starvation, e.g. $(s_1, s_4, s_{X1}, s_{X2}, s_{X3})$.

Scenario: equal work for t and u

Now, assume that the `doWork()` methods of t and u are set up in such a way that the **`while`** loops of both threads execute *the same number of iterations*. For example, the two `doWork()` methods could execute tasks from a pair of work list l_t and l_u of the same length. Call this the *equal work* scenario.

(Part f) List all the states from the transition table that become *unreachable* in the *equal work* scenario. (A state is unreachable if no execution of the program will visit it.) Why are they unreachable? *(4p)*

Solution of 2.f: the states s_{X2}, s_{X3}, s_{Y2} and s_{Y3} become unreachable. In these states, one thread has terminated while the other thread is about to wait on the barrier (s_{X3}, s_{Y3}) or will be doing so after the next transition (s_{X2}, s_{Y2}). Since threads synchronize at the barrier at the beginning of each iteration, the thread waiting on the barrier is trying to execute one iteration more than the terminated thread.

(Part g) Does your answer to Part e change in the *equal work* scenario? If yes, prove your new answer from the updated state transition table (with unreachable states removed). If your new answer is yes, give a trace that ends in a starvation path. If your new answer is no, briefly explain why there cannot be any t- or u-starvation paths in this scenario. *(3p)*

Solution of 2.g: neither thread can starve now. The states s_{X3} and s_{Y3} are unreachable, but each starvation path ends in one of them, so all starvation paths are eliminated. There are no other starvation paths: from every remaining (non-final) state where one of the threads is blocked, the other thread is forced to move (in at most two steps) to a final state or a state where the first thread becomes unblocked again.

Q3. Barriers – temporal logic *(8p)*

Consider again the barrier synchronization problem described on page 5, and see Q2 for an explanation of the code.

Here in Q3, you must argue from the program, not from the state transition table (though you may seek inspiration from it). You get full credit for correct reasoning, whether you use formal logic, everyday language, or a mixture. Formulas and logical laws make your argument concise and precise, and help you keep track. With everyday language, be careful not to be fuzzy, or to mistake wishful thinking for proof. Appendix B reviews briefly the notation of propositional logic and linear temporal logic.

In your reasoning you may assume the following basic invariants about t and u:

1. a thread is in exactly one program location at once:
   $pc_t = i \wedge pc_t = j$ if and only if $i = j$, and similarly for pc_u;
2. the threads are only ever observed at three locations:
   if $pc_t = i$ then $i \in \{4, 5, 9\}$, and if $pc_u = i$ then $i \in \{13, 14, 18\}$.
3. the values of the semaphores b_t and b_u are non-negative: $b_t \geq 0$ and $b_u \geq 0$;
4. a semaphore has exactly one value:
   $b_t = i \wedge b_t = j$ if and only if $i = j$, and similarly for b_u.

Consider the following propositions:

- $B_t \equiv (pc_t = 5 \wedge b_u = 0)$ and $B_u \equiv (pc_u = 14 \wedge b_t = 0)$,
- $R_t \equiv (pc_t = 5 \rightarrow b_t > 0)$,
- $D \equiv \neg(B_t \wedge B_u)$,
- $S_t \equiv B_t \rightarrow \Diamond(pc_t = 4 \vee pc_t = 9)$.

(Part a) Describe the propositions B_t, D and S_t in words. Which concurrency properties do D and S_t correspond to? *(3p)*

Solution of 3.a: B_t says that the thread t is about to execute `barrier[`u`].down()` but the value of the semaphore b_u is zero, i.e. t is *blocked.* Similarly, B_u says u is blocked. R_t says that, if t is about to execute `barrier[`u`].down()`, then the semaphore b_t must have been released (i.e. its value must be strictly positive). D says that t and u are not simultaneously blocked, which implies *deadlock freedom*. S_t says that, if t is currently blocked, it will eventually make progress, i.e. move to either line 4 (next iteration) or line 9 (termination), which constitutes *starvation freedom* for t.

(Part b) Show that D is true if R_t is. *(2p)*

Solution of 3.b: assume R_t holds, but D does *not*, i.e. we have $B_t \wedge B_u$. Then the latter implies $pc_t = 5$ (by B_t) and $b_t = 0$ (by B_u). But this contradicts R_t, so we must have D.

(Part c) Give a counterexample to S_t, i.e. describe a program run where B_t is true, but $pc_t = 4 \vee pc_t = 9$ *never* becomes true. *(3p)*

Solution of 3.c: this amounts to giving a scenario where t starves. One such scenario is for u to execute one iteration of the loop and terminate, while t starts another iteration. Then $b_t = b_u = 0$ and $pc_t = 4$, $pc_u = 18$. After one step, $pc_t = 5$ but still $b_u = 0$, so B_t holds (t is blocked). Since u has terminated and t is blocked, neither thread can move, and $pc_t = 4 \vee pc_t = 9$ will never come true.

Q4. Fork/join parallelism *(10p)*

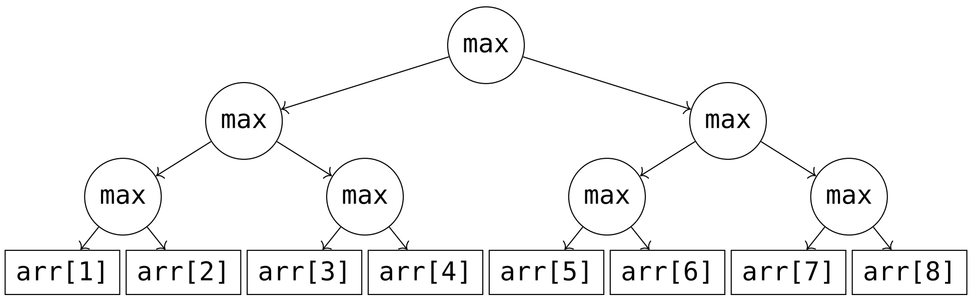
Consider the following recursive Java method `max()`, which finds the maximum value stored in the array `arr` from index `start` to index `end` $- 1$ (excluding `arr[end]`).

```java
int max(int[] arr, int start, int end) {
  if (end - start < 1)       return Integer.MIN_VALUE;
  else if (end - start == 1) return arr[start];
  else {
    int mid = (start + end) / 2;
    int m1 = max(arr, start, mid);
    int m2 = max(arr, mid, end);
    return max2(m1, m2);
  }
}
```

The method `max()` uses of the helper function `max2(int x, int y)`, which computes the maximum value of two integers `x` and `y` (see Appendix A.3 for its implementation).

(Part a) Draw the dependency graph for a call to `max(arr, 0, 8)` assuming `arr` has exactly 8 elements. Each node should represent a recursive call to `max()` with the leaf nodes representing the base cases (where `end - start <= 1`). Edges should represent data dependencies between calls. How many nodes does the graph contain? *(2p)*

Solution of 4.a: there are $8 + 4 + 2 + 1 = 15$ nodes.



(Part b) What is the maximum number of calls to `max()` that can be executed in parallel in this scenario (excluding parent calls waiting for a recursive call to finish)? How can this be inferred from the dependency graph? *(2p)*

Solution of 4.b: 8 calls – the width of the dependency graph.

(Part c) What is the approximate runtime of a fully parallel version of `max(arr, 0, 8)`, assuming each call to `max()` takes about one unit of time? How can this be inferred from the dependency graph? *(2p)*

Solution of 4.c: 4 time units – the depth of the dependency graph.

The class `MaxTask` below implements a naive fork/join parallelization of `max()`. The constructor of `MaxTask` has the same arguments as the method `max()`; the `run()` method computes the maximum value and stores the result in the public `result` field. See Appendix A.3 for a full code listing.

```java
class MaxTask extends Thread {
  private int[] arr;
  private int start, end;
  public int result;

  MaxTask(int[] arr, int start, int end) {
    this.arr = arr; this.start = start; this.end = end;
  }

  public void run() {
    if (end - start < 1)       result = Integer.MIN_VALUE;
    else if (end - start == 1) result = arr[start];
    else {
      int mid = (start + end) / 2;
      MaxTask t1 = new MaxTask(arr, start, mid);
      MaxTask t2 = new MaxTask(arr, mid, end);
      t1.start(); t2.start();
      try { t1.join(); t2.join(); } catch (InterruptedException i) {}
      result = max2(t1.result, t2.result);
    }
  }
}
```

(Part d) What is the total number of threads started by `MaxTask` when running an instance of `MaxTask(arr, 0, 8)`? How does this compare to the maximum number of parallel calls to `max()` you computed in Part b? Are there any blocked threads at any point during the execution? If yes, what is the maximum number of blocked threads? *(2p)*

Solution of 4.d: there are 15 threads – the same number as there are nodes in the dependency graph – and 7 more than the maximum number of calls that can execute in parallel. There may be up to 7 blocked threads when the "leaves" are being computed.

(Part e) Assume that an instance of `MaxTask(arr, 0, 1000000)` is being run on a machine with a 4-core processor. Will this result in an efficient parallel computation? How would the situation be improved if `MaxTask` was instead implemented as a `RecursiveTask` and scheduled using a `ForkJoinPool` from the `java.util.concurrent` package? *(2p)*

Solution of 4.e: no. Since every "recursive call" involves the creation of a new thread even for very small computations (near the "leaves"), spawning overhead will dominate the overall runtime. In addition, most threads are created unnecessarily since the number of threads ($\simeq$1M threads) far exceeds the physical capacity for parallel execution (4 cores). using a thread pool will address the latter and also reduce the spawning overhead, but the overhead will remain significant for small sub-tasks (at the leaves).

Q5. Erlang bowling alley *(18p)*

Consider the following event handler function of an Erlang server. The server models a clerk working at a bowling alley, handing out shoes to customers.

```erlang
handler(Stock) ->
  receive
    {request, From, N} when N =< Stock ->
      From ! ok,
      handler(Stock - N);
    {return, From, N} ->
      From ! ok,
      handler(Stock + N);
    {stock, From} ->
      From ! {ok, Stock},
      handler(Stock)
  end.
```

The clerk manages `Stock` pairs of shoes and needs to handle three types of events: customers may request or return shoes for `N` players, and the manager (or anyone else) may ask how many pairs of shoes the clerk currently has in stock. For the remainder of this question, assume the following *shoe dogma*:

*customers **never** return **more** shoes to the clerk than they previously obtained!*

In the following, let C_1, C_2, $\ldots C_n$ be the PIDs of customers processes, `S` the PID of the server, and `B` the PID of the boss. Assume the clerk starts off with 30 pairs of shoes, i.e. the server is spawned using `S = spawn(fun() -> handler(30) end).`

(Part a) Assume the server receives the following sequence of messages *in this order*:

`{request, C`$_1$`, 8}`, `{request, C`$_2$`, 25}`, `{return, C`$_1$`, 5}`, `{stock, B}`.

Give a sequence of *responses* sent by the server. Your answer should have the format $P_1!r_1, P_2!r_2, \ldots$ where P_i are PIDs and r_i are server responses. *(1p)*

Solution of 5.a: the only possible sequence is C$_1$**!ok**, C$_1$**!ok**, C$_2$**!ok**, B**!{ok**,2**}**. The request by C$_1$ will remain in the message queue until the stock recovers because of the guard condition in the first receive clause of `handler`.

(Part b) Is your answer to Part a the only possible sequence of responses? If so, explain why. Does the order in which the server receives the messages impact its responses? *(2p)*

Solution of 5.b: yes, this is the only possible sequence of responses because the server process is deterministic and the message queue is accessed in FIFO order.

If the messages were received in a different order, some might not be answered at all, others might differ in content. For example, if the requests by C$_2$ was received before that of C$_1$, the latter would never be answered: because of the *shoe dogma* C$_1$'s `return` message must come after its `request` message, so the stock would remain 5 – too low to serve C$_1$. The response to the manager may report the values 30, 27, 22, 5, or 2, depending on when it is received.

(Part c) Now, consider instead the following sequences of messages, given in the order they are received by the clerk (starting with an initial stock of 30 pairs of shoes):

1. `{request, C`$_1$`, 40}`, `{request, C`$_2$`, 5}`
2. `{request, C`$_2$`, 8}`, `{request, C`$_1$`, 25}`, `{request, C`$_3$`, 10}`, `{return, C`$_2$`, 8}`, `{request, C`$_4$`, 9}`, `{return, C`$_3$`, 10}`

Are all messages in these sequences eventually processed and answered by the server? If not, is it possible for customer processes to send *additional* messages to the server in such a way that all messages are answered by the server in the end? Give separate answers for each sequence and justify them. Your answers must respect the *shoe dogma*. *(3p)*

Solution of 5.c: in both cases, the message by C$_1$ never gets answered because the stock is too low when the message is received, and remains too low afterwards. For sequence 1, the message by C$_1$ can never be answered since the request exceeds the total number of shoes and, by the *shoe dogma*, no client can return more shoes than it previously requested, so the stock can never grow. For sequence 2, the message by C$_1$ will be answered if C$_4$ returns its shoes immediately.

More robust servers

You will be asked to write two short pieces of code. You may use Erlang, pseudo code, or a mixture of both. Syntax details, such as punctuation, are not important, but an experienced Erlang programmer should be able to understand your code. Your code *must use functional style*: use recursion, list comprehensions or higher-order functions instead of mutable state or loops. Each of your answers should fit into at most 10 lines of code.

(Part d) Let `Total` be the size of the clerk's initial stock, or equivalently, the total number of pairs of shoes in the bowling alley, counting both the clerk's stock and the shoes worn by customers. What happens when a customer asks for `Total` + 1 pairs of shoes? *(1p)*

Solution of 5.d: the customer never gets served, i.e. never gets a response from the server. (This corresponds to sequence 1 in Part c.)

(Part e) How could the server be changed so that customers requesting shoes for more than `Total` players receive a message `{error, Total}`? How should the state of the server (i.e. the arguments of `handler`) be adjusted? Propose *one new clause* to be inserted at the beginning of the `receive` expression in `handler`. You may write this clause in Erlang or pseudo code, but it should be of the form `P -> E` or `P when G -> E`, where `P` is a message pattern, `G` is a guard condition, and `E` is an expression. Your solution should work for arbitrary values of `Total`. *(4p)*

Solution of 5.e: the server state needs to keep track of `Total` in addition to `Stock`. Here is an example clause to be added at the beginning of the `receive` block to handle the case where `N > Total`:

```
handler(Stock, Total) ->
  receive
    {request, From, N} when N > Total ->
      From ! {error, Total},
      handler(Stock, Total);
    % ...
```

(Part f) Assume now that the server has been running for a while, handling out shoes to customers, so that `Stock < Total`. Suddenly, a customer `C` arrives requesting `N` pairs of shoes – more than the clerk has in stock, but such that `Stock < N <= Total`. Describe a scenario where `C`'s request is delayed indefinitely, while other customers continuously obtain and return shoes. What concurrency property is violated here? *(2p)*

Solution of 5.f: the sequence 1 in Part c can easily be extended into such a scenario: it suffices to extend the sequence by repeating the pattern

`{request,` C_3`, 10}`, `{return,` C_4`, 9}`, `{request,` C_4`, 9}`, `{return,` C_3`, 10}`, ...

indefinitely. In general, as long as the other customers request and return shoes in such a way that the `Stock` never exceeds `N-1`, customer `C`'s request is delayed indefinitely. This is an example of *starvation*, indicating that the server is not *fair*.

(Part g) To avoid this type of scenario, we change the `handler` function of the original server as follows, inserting a new clause in the `receive` expression:

```
handler(Stock) ->
  receive
    {request, From, N} when N =< Stock ->
      % ...
    {request, From, N} ->                % request where 'Stock < N'
```

```
      NewStock = recover(Stock, N),
      From ! ok, handler(NewStock - N);
    % ...
```

The new clause calls the helper function `recover`, which behaves like `handler` but takes an additional argument `Goal` and ignores any `request` messages until `Stock >= Goal`, at which point it simply returns the updated stock. Complete the following unfinished implementation of `recover` (only replace the bit marked "`todo`"): *(5p)*

```
recover(Stock, Goal) ->
  receive
    {return, From, M} ->  %%%%%%  Implement this clause!  %%%%%%
      todo;
    {stock, From} -> From ! {ok, Stock}, recover(Stock, Goal)
  end.
```

Solution of 5.g: the following is a possible implementation of `recover`:

```
% recover shoes until Stock >= Goal and return the new stock
recover(Stock, Goal) ->
  receive
    {return, From, M} ->
      From ! ok,
      NewStock = Stock + M,
      case NewStock < Goal of
        true  -> recover(NewStock, Goal);
        false -> NewStock
      end;
    {stock, From} ->
      From ! {ok, Stock},
      recover(Stock, Goal)
  end.
```

Appendix A. Full code listings

Appendix A.1. Code for Q1

```java
import java.util.concurrent.*;
import java.util.concurrent.locks.*;

class LockedCounterArray {
  final private int[] counters;
  final private Lock lock = new ReentrantLock();

  public LockedCounterArray(int size) {
    counters = new int[size];  /* all 0s by default */
  }
  public void increment(int index) {
    lock.lock();
    try { counters[index] = counters[index] + 1; }
    finally { lock.unlock(); }
  }
  public int value(int index) {
    lock.lock();
    try { return counters[index]; }
    finally { lock.unlock(); }
  }
  public static void main(String[] args) {
    final int N = 10;
    final int size = Integer.parseInt(args[0]);
    LockedCounterArray lca = new LockedCounterArray(size);

    Thread[] ts = new Thread[N];
    for (int i = 0; i < N; ++i) {
      ts[i] = new Thread() {
          @Override public void run() {
            for (int j = 0; j < size; ++j) { lca.increment((j + N) % size); }
          }
        };
      ts[i].start();
    }
    for (int i = 0; i < N; ++i) {
      try { ts[i].join(); } catch (InterruptedException ie) {}
    }
    for (int j = 0; j < size; ++j) {
      System.out.println("Counter " + j + ": " + lca.value(j));
    }
  }
}
```

Appendix A.2. Code for Q2 and Q3.

```java
import java.util.concurrent.*;

class BarrierThread extends Thread {
  private int id, work;
  public static Semaphore[] barrier = { new Semaphore(0), new Semaphore(0) };

  BarrierThread(int id, int work) { this.id = id; this.work = work; }

  boolean doWork() throws InterruptedException {
    if (work > 0) {           // more work?
      sleep(500 + id * 100);  // do "work"
      work = work - 1;        // update work counter
      return false;
    } else return true;       // done!
  }

  @Override public void run() {
    int me = id;
    int other = 1 - id;
    try {
      boolean done = false;
      do {
        // synchronize: wait for both threads to reach the barrier.
        System.out.println("Thread " + me + " synchronizing...");
        barrier[me].release();     // I'm done.
        barrier[other].acquire();  // Wait for the other thread.
        System.out.println("Thread " + me + " ready to work!");
        // do work.
        done = doWork();
      } while (!done);
    } catch (InterruptedException i) {}
  }

  public static void main(String[] args) {
    final int N1 = Integer.parseInt(args[0]);
    final int N2 = Integer.parseInt(args[1]);
    BarrierThread t0 = new BarrierThread(0, N1);
    BarrierThread t1 = new BarrierThread(1, N2);
    t0.start(); t1.start();
    try {
      t0.join(); t1.join();
    } catch (InterruptedException i) {}
  }
}
```

Appendix A.3. Code for Q4

```java
import java.util.concurrent.*;

class MaxTask extends Thread {
  private int[] arr;
  private int start, end;
  public int result;

  MaxTask(int[] arr, int start, int end) {
    this.arr = arr; this.start = start; this.end = end;
  }

  static int max2(int x, int y) {
    if (x > y) return x;
    else       return y;
  }

  @Override public void run() {
    if (end - start < 1)       result = Integer.MIN_VALUE;
    else if (end - start == 1) result = arr[start];
    else {
      int mid = (start + end) / 2;
      MaxTask t1 = new MaxTask(arr, start, mid);
      MaxTask t2 = new MaxTask(arr, mid, end);
      t1.start(); t2.start();
      try { t1.join(); t2.join(); } catch (InterruptedException i) {}
      result = max2(t1.result, t2.result);
    }
  }

  public static void main(String[] args) {
    final int N = 20; int[] numbers = new int[N];
    for (int i = 0; i < N; ++i) { numbers[i] = i; }
    MaxTask m = new MaxTask(numbers, 0, N);
    m.run();
    System.out.println("Max: " + m.result);
  }
}
```

Appendix A.4. Code for Q5

```erlang
-module(shoes).
-export([init/1,request/1,return/1,stock/0]).

% start the clerk process managing a stock of N pairs of shoes
init(N) ->
  register(clerk, spawn(fun () -> handler(N) end)).

% request N pairs of shoes
request(N) ->
    clerk ! {request, self(), N},
    receive ok -> ok end.

% return N pairs of shoes
return(N) ->
    clerk ! {return, self(), N},
    receive ok -> ok end.

% ask the clerk what the current stock is
stock() ->
    clerk ! {stock, self()},
    receive {ok, N} -> N end.

% the clerk's event handler
handler(Stock) ->
  receive
    {request, From, N} when N =< Stock ->
      From ! ok,
      handler(Stock - N);
    {return, From, N} ->
      From ! ok,
      handler(Stock + N);
    {stock, From} ->
      From ! {ok, Stock},
      handler(Stock)
  end.
```

Appendix B. Linear Temporal Logic (LTL) notation

1. An atomic proposition such as $q2$ (process q is at label $q2$) *holds for* a state s if and only if process q is at $q2$ in s.

2. Let ϕ and ψ be formulas of LTL. Formulas are either atomic propositions, or are built up from other formulas using the following operators: $\neg$ for “not", $\vee$ for “or", $\wedge$ for “and", $\rightarrow$ for “implies", $\Box$ for “always", and $\Diamond$ for “eventually". A convenient abbreviation is ϕ iff ψ (i.e., ϕ if and only if ψ) for $(\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$.

   These operators have the obvious meanings, but two differ from what might be your interpretation of the names. First, $\phi \vee \psi$ (“ϕ or ψ") is false iff both ϕ and ψ are false. This is an “inclusive or", so $\phi \vee \psi$ is also true if both ϕ and ψ are true. Second, $\phi \rightarrow \psi$ (“ϕ implies ψ") is false iff ϕ is true and ψ is false. So, in particular, $\phi \rightarrow \psi$ is true if ϕ is false. The meanings of the operators $\Box$ and $\Diamond$ are defined below.

3. A *path* is a possible future of the system, a possibly infinite sequence of states, each reachable from the previous state in the path. A state s *satisfies* formula ϕ if every path from s satisfies ϕ.

   A path π satisfies $\Box\phi$ if ϕ holds for the first state of π, and for all subsequent states in π. The path π satisfies $\Diamond\phi$ if ϕ holds for some state in π.

   Note that $\Box$ and $\Diamond$ are duals:

$$\Box\phi \equiv \neg\Diamond\neg\phi \qquad \text{and} \qquad \Diamond\phi \equiv \neg\Box\neg\phi.$$