INFOB3CC: Concurrency
Exam 1 (solutions)
19 December 2019, 08:30 – 10:30

Please read the following instructions carefully:

- Write your *name* and *student number* here:

  .....

- Be prepared to identify yourself with your ID card when you submit your exam.
- A maximum of 85 points can be obtained by answering the questions of this exam, to be divided by 8.5 to obtain the final mark.
- Provide brief and concise answers. Overly verbose responses or nonsense added to otherwise good answers can deduct from your grade.
- If a question does not give you all the details you need, you may make reasonable assumptions. Your assumptions must be clearly stated. If your solution only works under certain conditions, state them.
- Fill in your answers in the space provided. If you run out of space, continue in a separate answer booklet and clearly indicate your name, student number, and the question number.
- You may use diagrams to help explain your answers.
- Read through the paper first and plan your time accordingly.
- Answer questions in English.
- *Good luck!* (:

Please keep in mind that there are often many possible solutions
and that these example solutions may contain mistakes.

Definitions

1. (a) (5 points) According to our definitions in the lecture, what is the difference between parallelism and concurrency?

Solution:

- (2.5) Concurrency is the ability for the computer to compute tasks out-of-order with respect to each other; that is, two or more threads are making progress at a time. It is the composition of many independently executing processes. A concurrent application can be executed on a single processor.
- (2.5) Parallelism means to execute multiple (possibly related) tasks at the same time, with the goal to reduce the overall running time of the program. Requires multiple processing elements.

(b) (5 points) Give an example of a problem/computation which is parallel but not concurrent.

Solution: There are not many examples in this category; operations which must be executed in parallel and can not be split to execute on a single processing element. The example from lectures was SIMD instructions.

- 3 for a good description of what it means but without supporting example (or incorrect example)

(c) (5 points) What does it mean for a program to be lock-free?

Solution: Lock freedom allows individual threads to starve, but guarantees that at least one thread will make progress and complete in a finite number of steps.

Locks

2. You have been asked to implement the ledger software for a bank, which will hold the account balance for each of the clients. The software will support operations such as withdrawing, depositing, and transferring money between accounts. It will use threads in order to process multiple transactions concurrently. You intend to use locks in order to control the multiple threads in the program.

(a) (5 points) Consider the following implementation of a lock data type. This is a time-based lock for at most 10 threads, where each thread has a unique identifier `myId` in the range `[0..9]` (inclusive). Each thread gets a time slot in which it may acquire the lock.

```
1  data Lock = Lock (IORef Bool)
2
3  newLock :: IO Lock
4  newLock = do
5    r ← newIORef False                    -- True=locked, False=unlocked
6    return (Lock r)
7
8  lock :: Int → Lock → IO ()
9  lock myId l@(Lock ref) = do             -- myId is in the range [0..9]
10   time ← getCurrentTime                 -- current program run time, in milliseconds
11   if time `mod` 10 == myId
12      then do                            -- this is my timeslot; try to acquire the lock
```

```
13          locked ← readIORef ref
14          if locked
15             then lock myId l              -- retry
16             else writeIORef ref True      -- take lock
17       else                                -- not my timeslot; retry
18          lock myId l
19
20  unlock :: Lock → IO ()
21  unlock (Lock ref) = atomicWriteIORef ref False
```

A correct lock implementation must fulfil the properties of mutual exclusion, deadlock freedom, and starvation freedom. Explain why each of these requirements are or are not fulfilled by this lock implementation.

Solution:

- (2) mutual exclusion: no. A thread can be descheduled between checking whether the lock is free (line 13) and entering the critical section (line 14). Similarly thread can be descheduled between checking the current time (line 10) and deciding whether it is its turn to enter the critical section (line 11), so can try to enter the critical section when it is not its turn.
- (1) deadlock freedom: yes. If two threads try to acquire the lock at the same time, the implementation will not deadlock (or livelock) itself.
- (2) starvation freedom: no. A thread can be infinitely unlucky always ask for the lock (getCurrentTime) when it is not its timeslot.

(b) (5 points) Consider the following implementation of a lock data type. This is a ticket based lock; when a thread wants the lock, it takes a ticket number, and then waits for that number, similar to tickets in a pharmacy.

```
1   data Lock = Lock (IORef Int) (IORef Int)
2
3   init :: IO Lock
4   init = do
5     refTicket  ← newIORef 0
6     refCounter ← newIORef 0
7     return (Lock refTicket refCounter)
8
9   lock :: Lock → IO ()
10  lock (Lock refTicket refCounter) = do
11    ticket ← atomicModifyIORef' refTicket (λt → (t + 1, t))
12    let
13        wait = do
14          current ← readIORef refCounter
15          if current == ticket
16            then return ()              -- take lock
17            else wait                   -- not my turn; retry
18    wait
19
20  unlock :: Lock → IO ()
21  unlock (Lock _ refCounter) =
22    atomicModifyIORef' refCounter (λc → (c + 1, ()))
```

A correct lock implementation must fulfil the properties of mutual exclusion, deadlock freedom, and starvation freedom. Explain why each of these requirements are or are not fulfilled by this lock implementation.

Solution:

- (2) mutual exclusion: yes. The ticket number is always unique due to atomicModifyIORef, so only one thread can ever have a ticket number which matches the counter.
- (1) deadlock freedom: yes.
- (2) starvation freedom: yes. A thread will always get a unique ticket number due to atomicModifyIORef, so will eventually be able to take the lock (as usual, assuming threads do not crash while a lock is held).

NOTE: In the remaining parts of this question, assume that you are using a lock implementation which correctly fulfils the requirements for a lock as stated above.

(c) (5 points) The bank proposes that in order to safely execute transactions, a single global lock should be placed around the entire account ledger. You think this will not be a good solution; explain why.

Solution: The program will be sequentialised and no threads will be able to process transactions concurrently, all waiting on the single lock.

(d) (5 points) You propose instead to have a single lock on each individual account. You know you must be careful with this arrangement, however, because it is possible to encounter a deadlock when trying to access two accounts, even if you use a correct lock implementation. Give an example execution/scenario of how this can occur.

Solution: Canonical example:

```
thread 1: transfer(from A, to B, amount1)
thread 2: transfor(from B, to A, amount2)
```

If the threads take the individual account locks in that order, then it can be that thread 1 takes the lock on A and then is descheduled, then thread 2 takes the lock on B. At this point the program is deadlocked.

(e) (5 points) How can you prevent these deadlocks, while still using only one lock per bank account. You can not change the implementation of the lock itself, only how it is used.

Solution: Threads must always take locks in a specific order, for example taking the hash of the two locks and taking the lowest hash first. For example if $hash(A) < hash(B)$ then threads should take the lock on A first.

(f) (5 points) How would you extend the answer of the previous section to handle a *variable* number of bank accounts in a *single* transaction, in particular, when it is not known beforehand which accounts will need to be accessed? For example, to withdraw money from a secondary account when there are insufficient funds available in the first account.

Solution: Extending the idea of the previous section, to take a third lock (after the first two are already acquired) then it is still required that all locks were taken in the correct order. For

example if we have that $hash(A) < hash(C) < hash(B)$, since we already have the lock on B, we must release it, take the lock on C, and re-take the lock on B.

STM

3. (a) (5 points) When writing code using STM, you cannot perform side effects in `IO`, such as reading or writing files. Why is this restriction needed?

Solution: STM relies on the ability to *undo* the effects of the atomic block when a conflict is detected, and then *retry* the transaction. Most `IO` actions, such as writing to the console, can not be undone, so the `STM` type restricts the transaction to only contain operations which can rolled-back when a transaction is retried.

(b) (5 points) How does STM guarantee the property of mutual exclusion?

Solution: STM functions are executed as a single atomic transaction using the `atomically` function. To other threads, it appears that all of the modifications in the atomically block happen instantaneously.
This is achieved by keeping track of a log of what actions a thread performs in the `atomically` section, at at the end of the block, validating the lock to ensure that the thread executed with a consistent view of memory, and if so, committing all changes in the log to memory.

(c) (5 points) Does STM guarantee the absence of starvation? Explain why or why not.

Solution: No. A thread can be continually forced to retry, if another thread finishes its transaction and commits some change which and forces it to abort. This is a particular problem with a long-running transaction competing with short running transactions.

- (3) For mentioning that all blocked threads are woken up when a `TVar` changes, rather than in FIFO order, but no mention of retries etc. in the above.

(d) (5 points) In some situations STM transactions can be slow. Give an example where this can occur, and explain why this happens.

Solution: Valid examples:

- Each `readTVar` must traverse the lock to see if it was written by an earlier `writeTVar`, which is an $O(n)$ operation.
- A transaction is woken up whenever any one of the `TVar`s in its read set changes, so calling `retry` is $O(n)$.
- Composing too many blocking operations together can cause a thread to be woken up many times. For example, if we want to wait on a list of `TMVar`s, consider
  `atomically (mapM takeTMVar ts)`; vs.
  `mapM (atomically . takeMVar) ts`.
  In the first example the transaction is re-run from the start for every element of `ts`, so is $O(n^2)$.

(e) (5 points) Disgruntled with the limitations of software transactional memory, Felix makes his own version which includes the possibility to read and write files on disk. Writing to files is directly

executed, and if the transaction fails, the operation is reverted by rewriting the original contents of the file back. Will this approach work? If so motivate your answer, or if not explain why or describe a problem which can be encountered.

Solution: This will not work because another thread can observe the intermediate state of the transaction. Thus another thread can execute with an inconsistent view of memory. Example: Thread A write a file to memory, then is descheduled. Thread B reads the contents of that file, and commits its results. Thread A resumes but detects a conflict and is forced to abort; it returns the file back to its original contents and retries.

Work & Span

4. Ada has developed a parallel algorithm with work $\Theta\left(n^{1.5}\right)$ and span $\Theta\left(\log^3 n\right)$. Gabriëlle thinks that her own algorithm for the task is better, since the span is only $\Theta\left(\log^2 n\right)$, although the work is $\Theta\left(n^2\right)$.

   (a) (5 points) Which algorithm will perform (asymptotically) better with a linear $\Theta(n)$ number of processors? Motivate your response.

   Solution: With a low number of processors $P < work/span$, both algorithms are work bound so you can estimate the time as $T_P = work/P$. Gabriëlle runs in a time $n^2/n = n$ (linear time), while Ada runs quicker in $n^{1.5}/n = \sqrt{n}$ (polynomial time).

   (b) (5 points) Which algorithm will perform (asymptotically) better with a quadratic $\Theta\left(n^2\right)$ number of processors? Motivate your response.

   Solution: With a large number of processors both algorithms are span bound, so you can estimate the running time as $T_P = span$. Ada will use $\log^3 n$ time while Gabriëlle uses only $\log^2 n$ time.

   (c) (5 points) For what number of processors is Ada's algorithm (asymptotically) faster?

   Solution: The break even point is at $P = n^2/\log^3 n$. With that number of processors Ada is span-bound, using $\log^3 n$ time and will not get any faster by adding more processors. At this point Gabriëlle is still work bound, and will continue to increase performance with more processors (up to a factor $\log n$).

THERE ARE NO MORE QUESTIONS

Enjoy the holidays.