Department of Information and Computer Science
Utrecht University

# INFOB3CC: Sharing state (solutions)

Trevor L. McDonell

November 28, 2021

## Introduction

Use these tasks to practice the topics of the lectures. You may have to do some research to read up on terms or topics not (yet) covered in the lectures.

## Questions

1. With a single `MVar` you can create a multi-threaded lock.
   (a) Which operation(s) in Haskell can you use to implement this operation?
   (b) Give the code for the `lock` and `unlock` methods.

   Example implementation:

   ```
   type Lock = MVar ()

   lock :: Lock -> IO ()
   lock = takeMVar

   unlock :: Lock -> IO ()
   unlock lock = putMVar lock ()
   ```

   (c) Which of the properties does the `MVar`-based lock not have? Explain why:
   - Mutual exclusion
   - Deadlock freedom
   - No starvation

   Unlike the `IORef` based lock of the tutorial set 2 (Threads), the `MVar` lock does not have these problems, in particular starvation. The Haskell runtime system ensures fairness because threads blocked on the `MVar` are woken up in first-in-first-out order.

   Of course, as with the `IORef` lock, sets of `MVar` based locks can cause deadlock when used incorrectly, even if individually the lock is deadlock-free.

2. We now have discussed two types of locking mechanism, the spin lock (using `IORef`, see tutorial set 2 (Threads)) and the blocking lock (using `MVar`).
   (a) Explain the difference between these two types of locks.

With a spin lock, the thread actively waits by looking at a variable until it has a desired value. With the blocking lock, the thread is descheduled by the runtime system so does not consume CPU resources while waiting for its turn at the lock.

(b) In which situation is a spin lock be better? In which situation is a blocking lock better?

You have to weigh the cost of actively spinning (the processor time as well as the associated memory traffic) against the thread scheduling cost. If you expect the thread to wait only a very short time (low contention) then spinning *may* be better; as the waiting time (contention) increases, blocking will be preferred.

The user-space threads in Haskell (managed by the Haskell runtime system) are very lightweight, so the scheduling cost is lower than using kernel threads directly, as is common in some other languages like C and C#.

3. What is the behaviour when multiple threads concurrently execute on an `MVar`:

(a) `putMVar  :: MVar a -> a -> IO ()`

If the `MVar` is empty, the first thread will put its value into the `MVar`, which is then full. When the `MVar` is full, the threads will be blocked and get woken up in first-in-first-out order whenever the `MVar` is emptied (by another thread).

(b) `takeMVar :: MVar a -> IO a`

If the `MVar` is full, the first thread will remove the value out of the `MVar` and then leave it empty. When the `MVar` is empty, the threads will be blocked and are woken up in FIFO order whenever the `MVar` is full.

(c) `readMVar :: MVar a -> IO a`

If the `MVar` is full the thread will read the value, and leave the `MVar` full. If it is empty, once a value is put into it, all threads blocked reading it will be woken up at once.

4. In the lecture we discussed the `Async` wrapper to manage asynchronous computations. Write a function `race` which, given two asynchronous computations, returns the result of the first computation to complete:

```
race :: Async a -> Async b -> IO (Either a b)
```

A partial solution:

```
race :: Async a -> Async b -> IO (Either a b)
race a b = do
  var <- newEmptyMVar
  _   <- forkIO $ do
           ra <- wait a
           putMVar var (Left ra)
           cancel b
```

```
_   <- forkIO $ do
         rb <- wait b
         putMVar var (Right rb)
         cancel a
takeMVar var
```

This code has a “thread leak”: if a finishes first, b is cancelled and the second forkIO thread in race will block indefinitely. Try to figure out how to fix this problem.

5. In the lecture we discussed building an unbounded queue using MVars. One additional operation we might like is to “undo” a dequeue operation; pushing an item back onto the read end of the queue. Consider the following implementation:

```
undequeue :: Queue a -> a -> IO ()
undequeue (Queue readEnd _) val = do
  newReadEnd <- newEmptyMVar
  stream     <- takeMVar readEnd
  putMVar newReadEnd (Item val stream)
  putMVar readEnd newReadEnd
```

Is the implementation correct? Consider what happens in different situations, such as when there are multiple concurrent calls to dequeue and/or undequeue, or if the queue is empty/non-empty.

The implementation works (concurrently) only if the queue is non-empty. Consider what happens when the queue is empty, a thread (T1) is blocked on dequeue, and then another thread (T2) calls undequeue. We would like that T1 should return the new element given by undequeue. However, T1 is already blocked and is holding the readEnd MVar, so undequeue will also block trying to take readEnd. The queue is deadlocked.

There is no known solution for this problem using this representation for Queue.

6. 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.

```
data Lock = Lock (IORef Bool)

newLock :: IO Lock
newLock = do
  r <- newIORef False                 -- True=locked, False=unlocked
  return (Lock r)

lock :: Int -> Lock -> IO ()
lock myId l@(Lock ref) = do           -- myId is in the range [0..9]
  time <- getCurrentTime              -- current program run time, in milliseconds
  if time `mod` 10 == myId
     then do                          -- this is my timeslot; try to acquire the lock
       locked <- readIORef ref
       if locked
          then lock myId l            -- retry
```

```haskell
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.

- 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.
- deadlock freedom: yes. If two threads try to acquire the lock at the same time, the implementation will not deadlock (or livelock) itself.
- starvation freedom: no. A thread can be infinitely unlucky and always ask for the lock (getCurrentTime) when it is not its timeslot.

7. 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.

```haskell
data Lock = Lock (IORef Int) (IORef Int)

init :: IO Lock
init = do
  refTicket  <- newIORef 0
  refCounter <- newIORef 0
  return (Lock refTicket refCounter)

lock :: Lock -> IO ()
lock (Lock refTicket refCounter) = do
  ticket <- atomicModifyIORef' refTicket (\t -> (t + 1, t))
  let
      wait = do
        current <- readIORef refCounter
        if current == ticket
          then return ()                    -- take lock
          else wait                         -- not my turn; retry
  wait

unlock :: Lock -> IO ()
unlock (Lock _ refCounter) =
  atomicModifyIORef' refCounter (\c -> (c + 1, ()))
```

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

- 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.
- deadlock freedom: yes.

(b) Does `unlock` require `atomicModifyIORef`, or could it also be updated non-atomically?

It does not require atomicity between the read of the old value and the write of the new value. It just requires atomicity of writes and can thus be implemented without `atomicModifyIORef` or atomic compare-and-swap. Only a thread holding the lock can update `refCounter`, and at most one thread can hold the lock at a time.

(c) The use of `atomicModifyIORef` in `lock` causes that this function may starve. Suppose we change this function to use atomic fetch-and-add, which atomically reads, returns and increments a variable, and is wait-free. Does that make this lock starvation free?

Yes. A thread will always get a unique ticket number, so it will eventually be able to take the lock (as usual, assuming threads do not crash while a lock is held).

8. 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) 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.

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

(b) 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.

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.

(c) 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.

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.

(d) 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.

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.

9. A large furniture company wants to develop an application to manage the stock in their warehouse. This will be used to check which products are in stock. Each product consists of one or multiple parts. A part may also be used for different products. For instance, different tables may use the same legs, as seen in these example products:

   - Table Utrecht
     - 1 × tabletop oak
     - 4 × oak leg
   - Table Amsterdam
     - 1 × tabletop oak
     - 2 × steel leg
   - Table Amersfoort
     - 1 × tabletop glass
     - 2 × steel leg

   In this database, we have three different products (three tables), which are built from four different parts (two kinds of tabletops, two kinds of table legs).

   The application keeps track of the number of each part is in stock. When making a new order, the system should check the stock of each part, and if everything is in stock, decrease the stock count of the used parts.

   (a) The system should handle multiple orders in parallel. Alex proposes to use a single lock for the whole database. Explain why this is a bad idea.

   The single lock will mean all orders must be processed sequentially.

   (b) Billy suggests to use a lock per part. When ordering a product, we acquire the lock of each part of that product, in the same order as listed in the product description. We verify the stock of each part and if everything is in stock, decrease the stock counts. Finally we release all the locks.

   Using the following product database, demonstrate an execute sequence where this method would fail:

   - Wardrobe Australia
     - 1 × frame
     - 1 × door
     - 1 × clothes rail
   - Wardrobe New Zealand
     - 1 × drawer
     - 1 × frame
     - 1 × door
   - Wardrobe accessories
     - 1 × clothes rail

  - 1 × drawer

Example execution:

| **Thread 1** | **Thread 2** | **Thread 3** |
|---|---|---|
| acquire lock: frame | | |
| acquire lock: door | | |
| | acquire lock: drawer | |
| | *wait: frame* | |
| | | acquire lock: clothes rail |
| | | *wait: drawer* |
| *wait: clothes rail* | | |

(c) How should a product database be constructed to prevent these issues?

The parts (locks) must be acquired in some fixed *global* order, for example by product number or alphabetically by name.

(d) Consider the following product database:

- Couch Neptune
  - 1 × seat module (small)
  - 1 × seat module (large)
- Couch Mars
  - 1 × seat module (small)
- Couch Venus
  - 1 × seat module (large)

The couches Mars and Venus are ordered very frequently. Omar is afraid that this will prevent an order containing couch Neptune from completing, since that order will need to be busy for a longer time as it contains twice as many modules. Explain what general property a lock should hold to prevent this situation being a problem. Explain why this situation cannot occur when that property holds.

Starvation freedom: every thread which wants access to the critical resource will eventually get it.

(e) An `MVar` can be empty or contain a value. Ivar says that you can use this to implement the locks. The stock counts will be stored in a value of type `MVar Int` per part, where the `Int` is the actual stock count. Explain how this can be done and how the empty state and blocking functions make this easy to implement.

In order to read or update the stock count, a thread has to take the `MVar`. Assume that all of the `MVar`s initially start full. If the `MVar` is empty when a thread wants it, that means another thread is currently updating the stock for that part, so the thread will wait for it to become full again and be woken up once that happens.