Department of Information and Computer Science
Utrecht University

# INFOB3CC: Sharing state

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.

When questions have tags like "[Threads 2]", this means that should expect to be able to do this exercise after the mentioned lecture.

## Questions

1. With a single `MVar` you can create a multi-threaded lock. [Threads 2]
   (a) Which operation(s) in Haskell can you use to implement this operation?
   (b) Give the code for the `lock` and `unlock` methods.
   (c) Which of the properties does the `MVar`-based lock not have? Explain why:
   - Mutual exclusion
   - Deadlock freedom
   - No starvation

2. We now have discussed two types of locking mechanism, the spin lock (using `IORef`; also appears in tutorial set 3 (Threads)) and the blocking lock (using `MVar`). [Threads 2/3]
   (a) Explain the difference between these two types of locks.
   (b) In which situation is a spin lock be better? In which situation is a blocking lock better?

3. What is the behaviour when multiple threads concurrently execute on an `MVar`: [Threads 2]
   (a) `putMVar  :: MVar a -> a -> IO ()`
   (b) `takeMVar :: MVar a -> IO a`
   (c) `readMVar :: MVar a -> IO a`

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: [Threads 2]

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

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: [Threads 3]

```
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.

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. [Threads 1]

```
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
          else writeIORef ref True     -- take lock
     else                              -- not my timeslot; retry
       lock myId l

unlock :: Lock -> IO ()
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.

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. [Threads 3]

```
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
```

```
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 (\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.

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

(c) The use of `atomicModifyIORef` in `lock` means 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?

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. [Threads 2]

   (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.

   (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.

   (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.

   (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.

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: [Threads 2]

   - 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.

(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

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

(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.

(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.