COMP4075 Real-world Functional Programming: Coursework Part II

Task II.1

• Brief explanations of your design and implementation

First of all, we need to use a semaphore. Different threads get signals by accessing the same semaphore, so we need to define semaphore first. In Haskell, STM newTVarIO can be easily defined. With this, we use semaphore to represent a fork. We can judge whether the current fork is available by reading the semaphore value continuously. If not, use retry to wait for it to be reread when it changes. STM implements the waiting process. The corresponding code is here.

```
15  useOneFork :: ForkSemaphore -> STM ()
16  useOneFork fork = do
17      state <- readTVar fork
18      if state then writeTVar fork False else retry
```

This is the core idea. With this, we can simulate our eating process. For a scientist named, when he/she is hungry (simulate by suspending the thread for a period of time), it will try to pick up the fork of his/her left and right hands. In order to avoid deadlock, we define picking up the left-hand fork and the right-hand fork as atomic operation atomically. If it fails to pick up two forks, it will put down the fork (i.e. the fork of the left hand) and try again We also define the operation of putting down the fork after eating as atomic. In fact, there is no need to put down the fork after eating. Of course, it does not affect anything. It is logical and appropriate. Finally, recursively call yourself to execute the process. This is what the eat function does.

In the end, for the run function, we need to simulate a group of philosophers dining at the same time. We should pay attention to this part because every philosopher is a thread, but printing uses the common thread. That's why we have Message Queue, which also relies on STM to ensure thread safety, using the same method as defining semaphores.

- Code

```haskell
import System.Random
import Control.Monad
import Control.Concurrent
import Control.Concurrent.STM

philosopherNames :: [String]
philosopherNames = ["Socrates", "Anthony", "Denny", "Edwin", "Leo"]

type ForkSemaphore = TVar Bool
type Message = TQueue String

emptySemaphore :: IO ForkSemaphore
emptySemaphore = newTVarIO True -- avaliable by default

useOneFork :: ForkSemaphore -> STM ()
useOneFork fork = do
    state <- readTVar fork
    if state then writeTVar fork False else retry

releaseOneFork :: ForkSemaphore -> STM ()
releaseOneFork = flip writeTVar True -- make one fork avaliable

delaySomeTime :: IO ()
delaySomeTime = do
    mus <- randomRIO (1000, 1200) -- microsecond
    threadDelay $ mus * 1000 -- millisecond

eat :: String -> Message -> ForkSemaphore -> ForkSemaphore -> IO ()
eat name msgs left right = do
    atomically $ writeTQueue msgs (name ++ " is hungry.")

    (leftNum, rightNum) <- atomically $ do
        leftNum <- useOneFork left
        rightNum <- useOneFork right
        return(leftNum, rightNum)
    atomically $ writeTQueue msgs (name ++ " got forks " ++ " and is now eating.")
    delaySomeTime -- start eating
    atomically $ writeTQueue msgs (name ++ " is done eating. Going back to thinking.")
    atomically $ do
        releaseOneFork left
        releaseOneFork right
    delaySomeTime -- start thinking

    eat name msgs left right -- repeat

run :: [String] -> IO ()
run names = do
    forks <- replicateM cnt emptySemaphore
    msgQueue <- newTQueueIO
    forM_ [0..cnt-1] (\i ->
        forkIO $ eat (names !! i) msgQueue (forks !! i) (forks !! (i + 1 `mod` cnt)))
    printMsg msgQueue
    return ()
    where
        cnt = length names
        printMsg :: Message -> IO String
        printMsg queue = do
                s <- atomically (readTQueue queue)
                putStrLn s
                printMsg queue

main :: IO ()
main = do
    run philosopherNames
```

• Enough sample output to provide evidence that your solution is working

```
(base) pc-200-217:task1 stephenwang$ ghc Main.hs -e main
Socrates is h<interactive>: Prelude.!!: index too large
ungry.
Socrates got forks  and is now eating.
Anthony is hungry.
Denny is hungry.
Denny got forks  and is now eating.
Edwin is hungry.
Leo is hungry.
Socrates is done eating. Going back to thinking.
Denny is done eating. Going back to thinking.
Anthony got forks  and is now eating.
Edwin got forks  and is now eating.
Edwin is done eating. Going back to thinking.
Socrates is hungry.
Denny is hungry.
Anthony is done eating. Going back to thinking.
Socrates got forks  and is now eating.
Denny got forks  and is now eating.
Denny is done eating. Going back to thinking.
Socrates is done eating. Going back to thinking.
Anthony is hungry.
Anthony got forks  and is now eating.
Edwin is hungry.
Edwin got forks  and is now eating.
Socrates is hungry.
Anthony is done eating. Going back to thinking.
Socrates got forks  and is now eating.
Denny is hungry.
Edwin is done eating. Going back to thinking.
Denny got forks  and is now eating.
^XSocrates is done eating. Going back to thinking.
Anthony is hungry.
Denny is done eating. Going back to thinking.
Anthony got forks  and is now eating.
Edwin is hungry.
Edwin got forks  and is now eating.
Anthony is done eating. Going back to thinking.
Socrates is hungry.
Socrates got forks  and is now eating.
Denny is hungry.
Edwin is done eating. Going back to thinking.
Denny got forks  and is now eating.
```

• Discussion of your STM solution in relation to the classical solutions

Arbitrator Solution

A simple solution is to introduce a restaurant waiter. If a philosopher wants to get the fork, he must first ask the waiter to give it to him. The waiter can serve a philosopher in the order of first come first, then serve the next philosopher. At this time, if the forks on both sides of the philosopher who is being served by the waiter are not occupied, the philosopher will definitely get the two forks, and will not be robbed by other philosophers, so as to avoid deadlock.

The characteristic of this method is simple, but the concurrency is very low. For example, suppose that philosopher No. 2 is already eating, and he occupies forks No. 2 and No. 3. At this time, philosopher No. 1 calls the waiter, and the waiter gives philosopher No. 1 fork No. 1, and then finds that fork No. 2 is being occupied, then the waiter must wait for philosopher No. 2 to finish eating before giving him No. 2 fork. Because the service provided by the waiter to philosopher No. 1 has not completed, so when other philosophers apply for service, the waiter cannot respond, causing the whole system to be blocked until philosopher 2's meal is over.

Resource Hierarchy Solution

Resources (forks) are numbered from 1 to 5, according to specific rules. Each work unit (philosopher) always picks up the lower numbered fork on the left and right sides first and then takes the higher numbered fork. After using the fork, he always put down the higher numbered fork first and then the lower one.

In this case, when the four philosophers simultaneously picked up the lower numbered fork at hand, only the highest numbered fork remained on the table. Thus, the fifth philosopher could not use any of them.

Moreover, only one philosopher can use the highest numbered fork so that he can eat with two forks. When he finished eating, he would put down the highest-numbered fork and then the lower one, so that another philosopher would pick up the one at the back and start eating.

STM

The implementation of software transaction memory includes atomic object and conflict manager. The implementation of atomic object is the most important, which is the medium of communication and synchronization between transactions. The implementation of atomic objects can be divided into sequential implementation and transaction implementation: the transaction implementation also requires the implementation of synchronization and recovery function. The synchronization function means the ability to detect the transaction conflict. In contrast, the recovery function means that the object needs to be rolled back to the state before the transaction execution when the transaction fails.

All in all, the STM is to improve concurrency and transactions are used to manage the read-write access of memory to avoid the use of locks.