COMP4075/G54RFP Coursework Part II

Benjamin Charlton — psybc3 — 4262648

5th December 2018

1 Task II.1 — Dining Philosophers

1.1 Solution Design

The problem is to develop a system simulating a number of philosophers at a table all alternating between thinking and eating. The issue is the limited amount of eating implements (sporks were used in this solution as to distinguish from forking threads) and the fact each philosopher needs 2 sporks to eat.

The main idea behind the solution is the use of Software Transactional Memory (STM), where each spork is held in a piece of STM. If a philosopher is hungry they will wait until they can grab both sporks at the same time and then begin eating. After they have finished eating they return the sporks to the table (adding them back into the relevant STM) so others can get them if required.

The solution is designed to generate a number of sporks equal to the number of philosophers. This means that the system can work with an arbitrary number of philosophers that is ≥ 1 as 1 philosopher will cause a deadlock as they want 2 sporks but only 1 exists.

1.2 Sample Output

Here is some sample output for my solution running with 7 philosophers around the table. It shows the first 50 lines of output and the time of each line being printed.

```
14:16:42 | Socrates is thinking
14:16:42 | Delaying Socrates 6 seconds
14:16:42 | Kant is thinking
14:16:42 | Delaying Kant 2 seconds
14:16:42 | Aristotle is thinking
14:16:42 | Delaying Aristotle 6 seconds
14:16:42 | Descartes is thinking
14:16:42 | Delaying Descartes 2 seconds
14:16:42 | Plato is thinking
14:16:42 | Delaying Plato 4 seconds
14:16:42 | Aquinas is thinking
14:16:42 | Delaying Aquinas 6 seconds
14:16:42 | Marx is thinking
14:16:42 | Delaying Marx 7 seconds
14:16:44 | Kant is hungry
14:16:44 | Kant is eating
14:16:44 | Delaying Kant 5 seconds
14:16:44 | Descartes is hungry
14:16:44 | Descartes is eating
14:16:44 | Delaying Descartes 5 seconds
14:16:46 | Plato is hungry
14:16:48 | Aristotle is hungry
14:16:48 | Socrates is hungry
14:16:48 | Aquinas is hungry
14:16:48 | Aquinas is eating
14:16:48 | Delaying Aquinas 8 seconds
14:16:49 | Kant is thinking
14:16:49 | Delaying Kant 7 seconds
14:16:49 | Marx is hungry
14:16:49 | Descartes is thinking
14:16:49 | Delaying Descartes 5 seconds
14:16:49 | Socrates is eating
14:16:49 | Delaying Socrates 4 seconds
14:16:49 | Aristotle is eating
14:16:49 | Delaying Aristotle 6 seconds
14:16:53 | Socrates is thinking
14:16:53 | Delaying Socrates 3 seconds
14:16:54 | Descartes is hungry
14:16:55 | Descartes is eating
14:16:55 | Delaying Descartes 3 seconds
14:16:55 | Aristotle is thinking
14:16:55 | Delaying Aristotle 3 seconds
14:16:56 | Aquinas is thinking
14:16:56 | Delaying Aquinas 5 seconds
14:16:56 | Kant is hungry
14:16:56 | Kant is eating
14:16:56 | Delaying Kant 5 seconds
14:16:56 | Marx is eating
14:16:56 | Delaying Marx 2 seconds
14:16:56 | Socrates is hungry
```

1.3 Implementation

1.3.1 Helper Functions

Microseconds to Seconds This function converts a integer representing a number of seconds to the same time in microseconds

```
ustos :: Int -> Int
ustos i = i * 1000000
```

Getting a String containing the current time This function uses the library Data.Time to get the current time and format it.

```
getTimeString :: IO String
getTimeString = do
                    zt <- getZonedTime
                    return $ formatTime defaultTimeLocale "%H:%M:%S" zt
```

Printing a Philosopher doing an Action This function prints a formatted string with the current time, the philosophers name and their intent (Thinking, Hungry, Eating)

```
printPhilosopherIntent :: String -> String -> IO ()
printPhilosopherIntent name i =
    do
        timeString <- getTimeString
        putStrLn (timeString ++ " | " ++ name ++ " is " ++ i)
```

Delaying a Philosopher and Printing This function will print how long the philosopher thread will be delayed (used to simulate the thinking/eating time) and then delays the thread for the given number of seconds.

```
delayPhilosopherPrint :: String -> Int -> IO ()
delayPhilosopherPrint name s =
    do
        timeString <- getTimeString
        putStrLn (timeString ++ " | Delaying " ++ name ++
                  " " ++ show(s) ++ " seconds")
        threadDelay (ustos s)
```

1.4 Philosopher function

The main outline of the philosopher function is an infinite loop by recursively calling itself without changing any of the inputs. It takes a string for the philosophers name and the pair of sporks that the philosopher will use while doing some IO action. The work that happens during this function is described in the following paragraphs but this code segment gives an idea of the overall structure.

```
philosopher :: String -> ((TMVar Int), (TMVar Int)) -> IO ()
philosopher name (ls, rs) =
    do
    -- Work happens here
     philosopher name (ls, rs)
```

Thinking The thing the philosopher does first is thinking. This is done by simply declaring the intent and then delaying for a random number of seconds (between 1 and 10).

```
printPhilosopherIntent name "thinking"
seconds <- randomRIO (1, 10)
delayPhilosopherPrint name seconds
```

Hungry Once the philosopher has stopped thinking they will become hungry. While hungry they try and take the 2 sporks on either side of them, this action is done atomically to ensure that their are no dead locks.

```
printPhilosopherIntent name "hungry"
(l, r) <- atomically $ do
    l <- takeTMVar ls
    r <- takeTMVar rs
    return (l, r)
```

Eating Once the philosopher has both sporks they can start eating, this happens in the same way as the thinking with a stating the intent and then delaying for a random amount of time. Once they have finished eating, they return the sporks by atomically putting them back correctly. Although the value replaced strictly doesn't matter they return the same value to the TMVar.

```
printPhilosopherIntent name "eating"
seconds <- randomRIO (1, 10)
delayPhilosopherPrint name seconds
atomically $ do
    putTMVar ls l
    putTMVar rs r
```

1.5 Main

The main function is what is executed when the program first runs, this is split into 3 major sections.

Generating pairs of sporks In the first 2 lines of the main function all of the synchronising variables are created. First of all the correct number of sporks equal to the number of philosophers and numbers them, the numbering isn't required for the system to run but allows for an easy and efficient way of creating enough TMVars in a map. After this the sporks are zipped into relevant pairs so that they can be passed onto the philosophers.

Forking philosophers The next lines deals with the forking separate threads for each of the philosophers. It zips the philosopher and pairs of sporks together running each one under the philosopher function. For each element in this list it will fork a separate thread to run the philosopher under.

Blocking Main The final is a way to allow main to continue running as if main ever terminates the philosopher threads will also end. By continuously calling threadDelay it means that main will never terminate.

```
main = do
    sporks <- mapM newTMVarIO [1..length(philosophers)]
    let pairs = zip sporks (tail(sporks) ++ [(head sporks)])

    mapM forkIO $ zipWith philosopher philosophers pairs

    forever $ threadDelay (ustos 1)
```

1.6 STM vs Classical Solutions

1.6.1 Why STM is free of deadlocks

During attempts to pick up sporks both actions are done atomically, meaning that there is no way for a philosopher to have access to a single spork. If a philosopher can't access the sporks it will simply wait and retry until it can access them both.

1.6.2 Resource Hierarchy Solution

A pro of using STM a hierarchy solution is that a STM doesn't generate a situation where many philosophers are holding a single spork waiting for the other to become available. This is because an STM allows for both sporks to be picked up at the same time allowing more of the philosophers eating at the same time. As stated in the wikipedia article if there was 5 philosophers at a table all hungry at the same it would mean that only 1 philosopher would have access to both sporks required for eating, in the STM system 2 philosophers would manage to get access to both required sporks. This situation doesn't get any better for more philosophers as in the hierarchy system there will always only be 1 philosopher able to get both resources, this would often result in a long wait time for the others to eat and the available fork trickling one by one around the table, only having one philosopher eating at a time. The STM system would allow $\frac{1}{2}$ of the philosophers to eat at once giving a higher utilisation of resources and sooner freeing of those resources for other philosophers.

Another problem with the hierarchy system is if at a later date a thread requires more resources it needs to systematically free the higher resources are require them all starting at the lowest resource. This can result in complex code to write as you have to remember to acquire resources in the correct order. With an STM this can be resolved by freeing all of the resources and then reacquiring them all atomically, as the order doesn't matter its much simpler to reacquire the resources you need correctly.

1.6.3 Arbitrator Solution

The arbitrator solution creates a single synchronisation variable that tells the philosophers if they are allowed to pick up sporks. This simplifies the amount of synchronisation needed to be managed but can reduce the parallelism as several philosophers that could safely pick up sporks have to wait and it also makes the synchronising logic more complex.

You can leave the synchronisation logic fairly simple if you simply let a single philosopher attempt to pick up their sporks at once and if they fail to do so let them continue trying until they succeed (This works as replacing sporks is always allowed). The problem with this solution is the reduction in parallelism achieved as many threads are simply waiting for their turn, an issue that is reduced with the STM solution.

You could create a more sophisticated synchronisation logic if you allowed philosophers to revoke its permission to collect sporks if it fails to collect them. This would allow the arbitrator to give another philosopher a chance to try and access their sporks and they may succeed. This would increase parallelism but not to the same levels as the STM solution, while also requiring much more complex logic leaving more places where bugs could occur.