Concurrency in Haskell

Assignment Project Exam Help

ECS713

https://tutorcs.com

Functional Programming

WeChat: cstutorcs

Dr. Paulo Oliva / Prof. Edmund Robinson

Queen Mary
University of London

Assignment Project Exam Help

Techniques for Multicore and Multithreaded Programming

Parallel and Concurrent Programming in Haskell

https://tutorcs.com
WeChat: cstutorcs

O'REILLY®

Simon Marlow

https://smunix.github.io/chimera.labs.oreilly.com/books/1230000000929/

Parallel vs Concurrent

Parallel	Concurrent
Computing a value faster	Combining interacting systems
Task broken into smaller (similar) sub-tasks that can be processed independently	Various processes performing different tasks and communicating between them
Lives in the pure world	Lives in the IO monad
Results often combined in main program	Usually the system is not meant to terminate

Assignment Project Exam Help
https://tutorcs.com
WeChat: cstutorcs

GHC supports running Haskell programs in parallel on an SMP (symmetric multiprocessor).

There's a fine distinction between **concurrency** and **parallelism**: **parallelism** is all about making your program run faster by making use of multiple processors simultaneously. **Concurrency**, on the other hand, is a means of abstraction: it is a convenient way to structure a program that must respond to multiple asynchronous events.

Assignment Project Exam Help
https://tutorcs.com

However, the two terms are certainly related. By making use of multiple CPUs it is possible to run concurrent threads in parallel, and this is exactly what GHC's SMP parallelism support does. But it is also possible to obtain performance improvements with parallelism on programs that do not use concurrency.

WeChat: cstutorcs

https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/using-concurrent.html#using-smp

Control.Concurrent

A common interface to a collection of useful concurrency abstractions.

```
forkIO :: IO () -> IO ThreadId                                    # Source
```

Creates a new thread to run the `IO` computation passed as the first argument, and returns the `ThreadId` of the newly created thread.

Assignment Project Exam Help
https://tutorcs.com
WeChat: cstutorcs

as soon as main program finishes all threads are killed

need way to synchronise threads (shared memory)

```
newEmptyMVar :: IO (MVar a)
```

Create an `MVar` which is initially empty.

create an empty "box" of type a

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

Create an `MVar` which contains the supplied value.

box of type a with some initial value

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

Put a value into an `MVar`. If the `MVar` is currently full, `putMVar` will wait until it becomes empty.

put a value in a box

Assignment Project Exam Help
https://tutorcs.com
WeChat: cstutorcs

```
takeMVar :: MVar a -> IO a
```

Return the contents of the `MVar`. If the `MVar` is currently empty, `takeMVar` will wait until it is full. After a `takeMVar`, the `MVar` is left empty.

remove the value from a box

```
readMVar :: MVar a -> IO a
```

Atomically read the contents of an `MVar`. If the `MVar` is currently empty, `readMVar` will wait until its full. `readMVar` is guaranteed to receive the next `putMVar`.

read the value without removing it

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

data Coin = Head | Tail deriving (Show,Eq)

coinFlip :: IO Coin
coinFlip = do
    r <- randomIO :: IO Bool
    return $ if r then Head else Tail
```

we will be putting “coins” in the box

Assignment Project Exam Help
https://tutorcs.com
WeChat: cstutorcs

A call to coinFlip returns either Head or Tail
(with uniform probability)

Coin-flip Game

- Initially, we flip a coin **c** an place in a box
- Three treads will take turns by, flipping their own coin, and checking if their coin is the same as the coin **c** in the box
  - If it is that player is the winner!
  - If not, that player places the coin **c** back in the box, so the other can try again

Assignment Project Exam Help
https://tutorcs.com
WeChat: cstutorcs

Each process will also flip a coin and win if it is the same as the coin in the box

```
process :: String -> MVar String -> MVar Coin -> IO ()
process name winner box = do
    c1 <- takeMVar box
    putStrLn $ name ++ " 's turn"
    c2 <- coinFlip
    putStrLn $ " -- got " ++ (show c2)
    if c1 == c2 then
        putMVar winner $ "Process " ++ name ++ " wins!"
    else do
        putStrLn $ " -- putting coin back in the box"
        putMVar box c1
        threadDelay 5
        process name winner box
```

We will also have a box of type String, to store the name of the winner

wait a bit before trying again

Assignment Project Exam Help
https://tutorcs.com
WeChat: cstutorcs

The “main” program spawns three processes, which will be taking turns trying to get the same coin values as the initial coin

```
main = do
    coin <- coinFlip
    putStrLn $ "Random coin is: " ++ (show coin)
    box <- newMVar coin
    winner <- newEmptyMVar
    forkIO (process "A" winner box)
    forkIO (process "B" winner box)
    forkIO (process "C" winner box)
    w <- takeMVar winner
    putStrLn $ w
```

Assignment Project Exam Help
https://tutorcs.com
WeChat: cstutorcs

At this point we have three threads running, and the main thread will wait until one of them wins

```
lecture 11» stack ghc concurrent.hs -- -threaded -rtsopts
[1 of 1] Compiling Main ( concurrent.hs, concurrent.o )
Linking concurrent ...
lecture 11» ./concurrent
Random coin is: Head
B's turn
 -- got Tail
 -- putting coin back in the box
C's turn
 -- got Head
Process C wins!
lecture 11» ./concurrent
Random coin is: Head
B's turn
 -- got Tail
 -- putting coin back in the box
C's turn
 -- got Tail
 -- putting coin back in the box
A's turn
 -- got Tail
 -- putting coin back in the box
B's turn
 -- got Head
Process B wins!
```

Assignment Project Exam Help
https://tutorcs.com
WeChat: cstutorcs