Lecture 20

Concurrent programming in Haskell

Haskell has a very good support for concurrent programming. In this lecture we see a very basic introduction to concurrent programming.

20.1 Threads

The Haskell runtime implements threads which are really lightweight. This means that on a decent machine you can open say 10K threads and still be have decent performance. This makes Haskell a great platform to implement high connectivity servers like http/ftp servers etc.

However you need to be carefull about FFI calls to C code. If one of your thread makes a call to a C function that blocks then all the threads will block. Sometimes the call to C functions might be indirect, you might use a library that uses Haskells FFI to talk to an already implemented C library. Then you can be in trouble if you are not careful.

One creates threads in Haskell using the `forkIO :: IO () -> IO ThreadId` function. The threads created using `forkIO` are really local to you ghc process and will not be visible in the rest of the OS. There is a similar `forkOS` function that creates an OS level thread. As long as the code uses pure haskell functions `forkIO` is all that you need.

We give here a basic function that forks a lot of worker process. The module to import is `Control.Concurrent`.

```
> import Control.Concurrent -- Concurrency related module
```

```
> import System.Environment -- For command line args in main
```

This is the worker process. In real life program this is were stuff happen.

```
> worker :: Int -> IO ()
> worker inp = do tId <- myThreadId
>                 let say x = putStrLn (show tId ++ ": " ++ x)
>                     in do say ("My input is " ++ show inp)
>                           say "Oh no I am dying."
>
```

Here is where the process is created. Note that `forkIO . worker` takes as input an integer and runs the worker action on it in a seperate thread

```
> runThreads :: [Int] -> IO ()
> runThreads = sequence_ . map (forkIO . worker)
>
```

And finally this is the main function where all the command line arguments are parsed and things done.

```
> main = do args <- getArgs
>           case args of
>                [a] -> let nthreads = read a
>                           in runThreads [1..nthreads]
>                _   -> putStrLn "Bad arguments"
>
```

20.2 Daemonic thread termination

There is a big bug in the code you just saw. All threads are terminated as soon as the main thread terminates. For real world applications one wants main thread to hang around till the others are done. We will see one way to handle this in the next lecture.