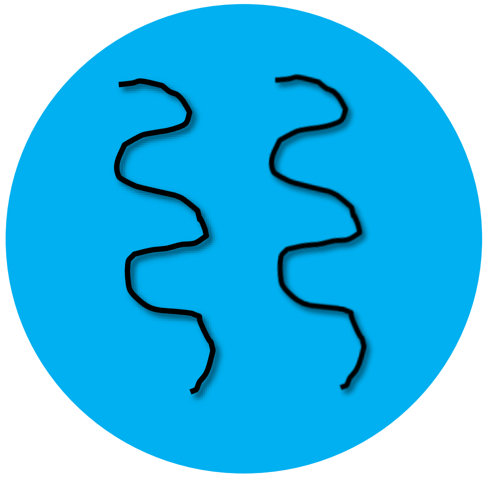
IMPLEMENTAREA CONCURENTEI IN LIMBAJE DE PROGRAMARE

Concurenta

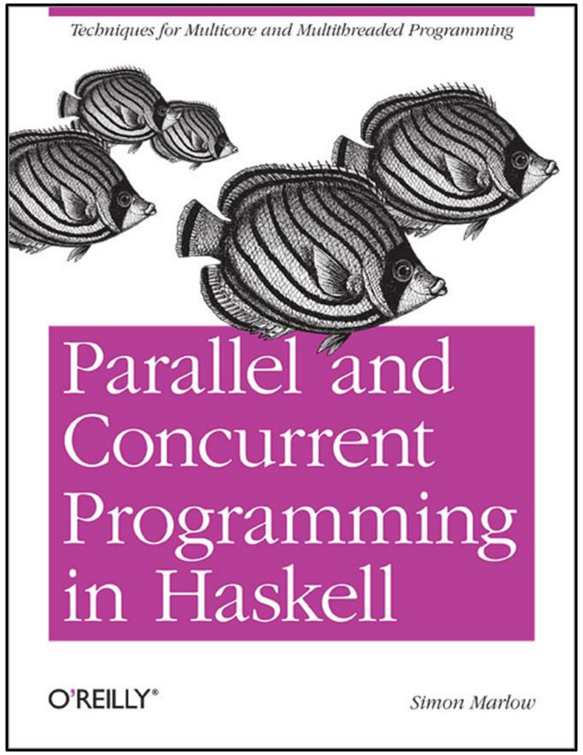
Threaduri

Memorie Partajata

Ioana Leustean



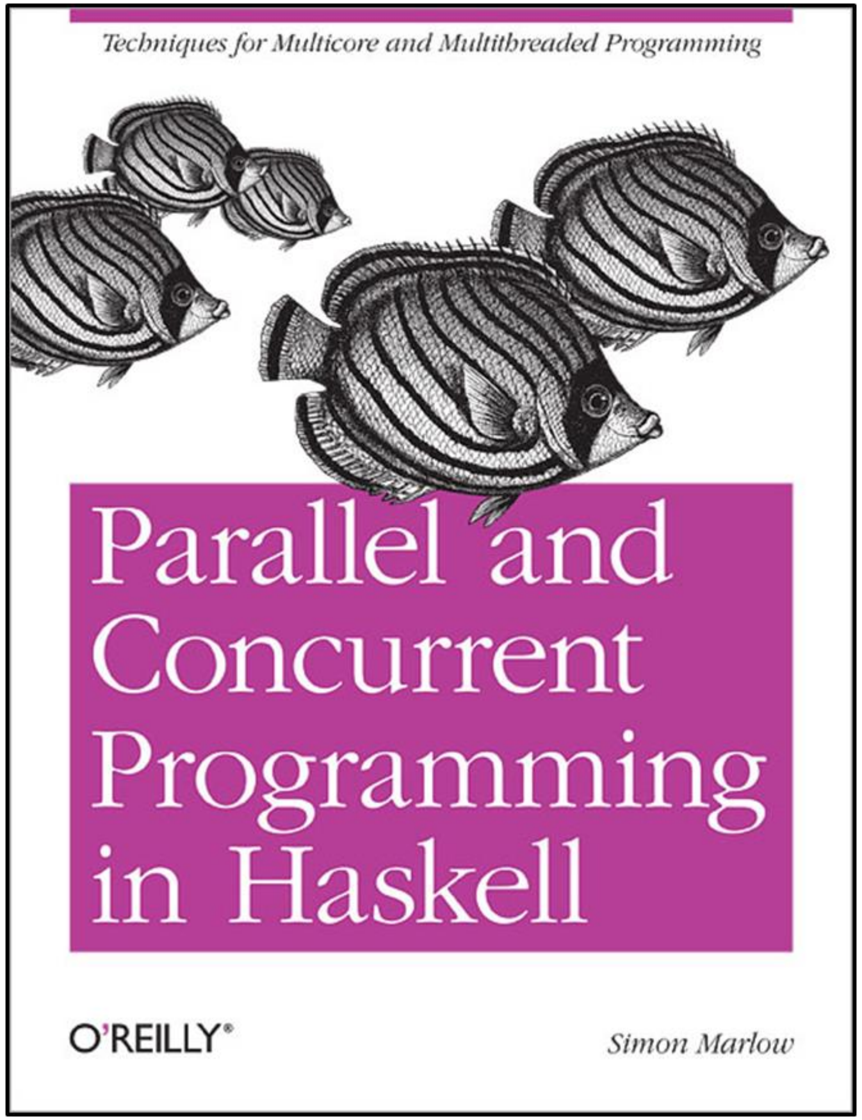
Haskell


Techniques for Multicore and Multithreaded Programming
Parallel and Concurrent Programming in Haskell
O'REILLY
Simon Marlow

Part II. Concurrent Haskell
Cap.7 & 8

"Haskell does not take a stance on which concurrent programming model is best: actors, shared memory, and transactions are all supported, for example."

"Haskell provides all of these concurrent programming models and more - but this flexibility is a double-edged sword. The advantage is that you can choose from a wide range of tools and pick the one best suited to the task at hand, but the disadvantage is that it can be hard to decide which tool is best for the job."
S. Marlow


Techniques for Multicore and Multithreaded Programming
Parallel and Concurrent Programming in Haskell
O'REILLY
Simon Marlow

➢ Thread-urile in Haskell:

Thread-urile au efecte si interactioneaza cu lumea exterioara.

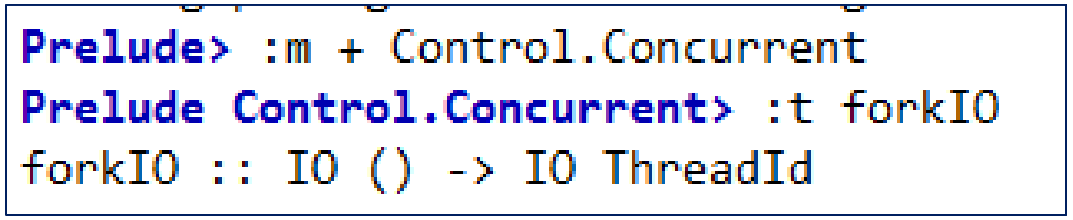
Programarea concurenta in Haskell are loc in monada IO.

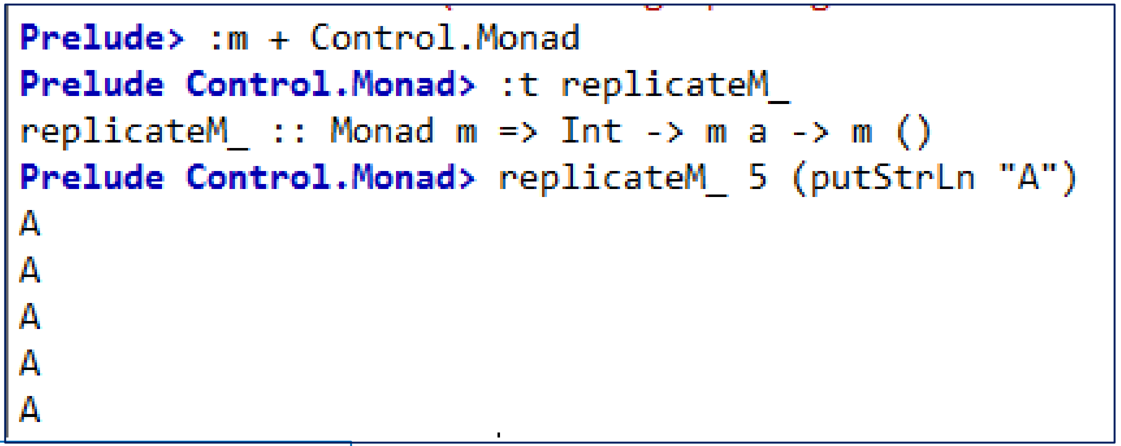
La rulare, efectele thread-urilor sunt intercalate nedeterminist.

Thread-urile in Haskell sunt create si gestionate intern,
fara a folosi facilitati specifice sistemului de operare.

Implementarea threadurilor asigura verificarea anumitor conditii de corectitudine (fairness)

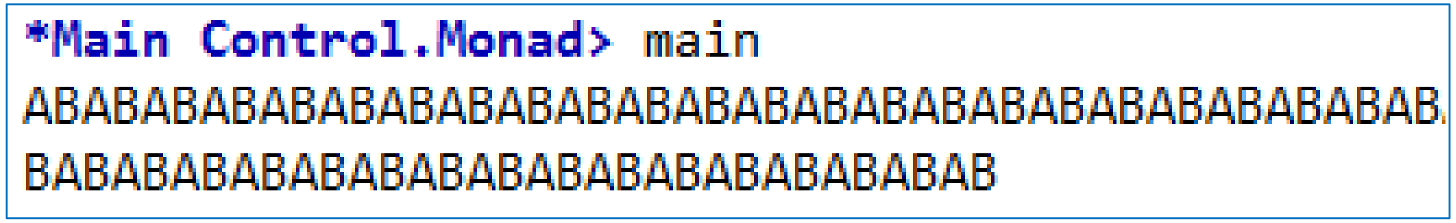
forkIO :: IO () -> IO ThreadId


```
Prelude> :m + Control.Concurrent
Prelude Control.Concurrent> :t forkIO
forkIO :: IO () -> IO ThreadId
```


```
Prelude> :m + Control.Monad
Prelude Control.Monad> :t replicateM_
replicateM_ :: Monad m => Int -> m a -> m ()
Prelude Control.Monad> replicateM_ 5 (putStrLn "A")
A
A
A
A
A
```

replicateM

```
import Control.Concurrent
import Control.Monad

main = do
        forkIO (replicateM_  100 (putChar 'A'))   -- child thread
        replicateM_ 100 (putChar 'B')  -- main thread
        putStrLn " "
```


```
*Main Control.Monad> main
ABABABABABABABABABABABABABABABABABABABABABABABAB
BABABABABABABABABABABABABABABABAB
```

La rulari diferite se pot obtine rezultate diferite!

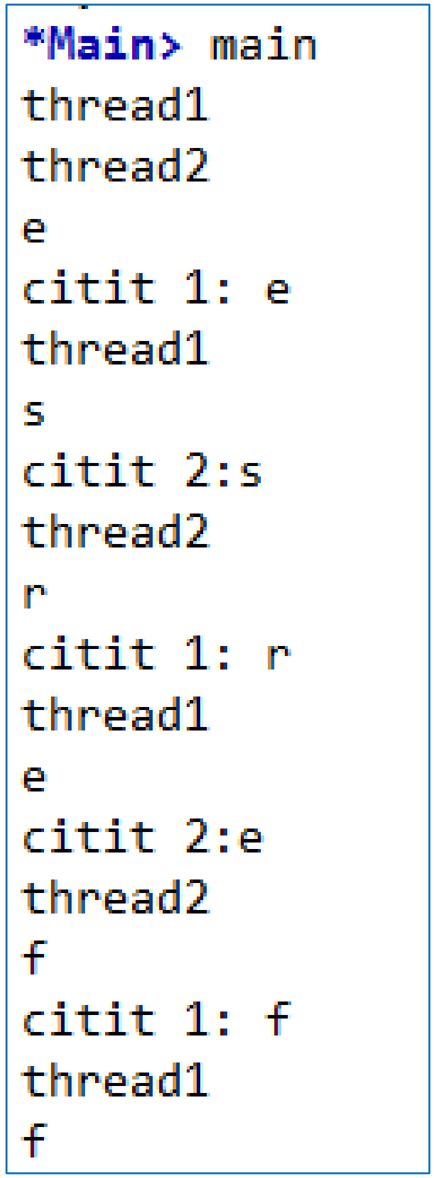
"The computation passed to forkIO is executed in a new thread that runs concurrently with the other threads in the system. If the thread has effects, those effects will be interleaved in an indeterminate fashion with the effects from other threads."
S. Marlow, PCPH

"forkIO is assymetrical: when a process executes a forkIO it spawns a child process that executes concurrently with the continued execution of the parent"
SL Peyton Jones, A Gordon, S Finne, Concurrent Haskell

"GHC's runtime system treats the program's original thread of control differently from other threads.
When this thread finishes executing, the runtime system considers the program as a whole to have completed.
If any other threads are executing at the time, they are terminated."
B. O'Sullivan, D. Stewart, J. Goerzen, Real World Haskell

➢ Interleaving

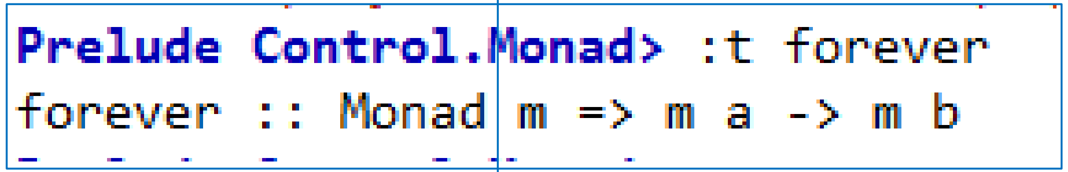
```
import Control.Concurrent
import Control.Monad

myread1 = do
        putStrLn "thread1"
        s<- getLine
        putStrLn $ "citit 1: " ++ s

myread2 = do
        putStrLn "thread2"
        s<- getLine
        putStrLn $ "citit 2:" ++ s


main = do
          forkIO (replicateM_  10 myread1)
          replicateM_ 10 myread2
```



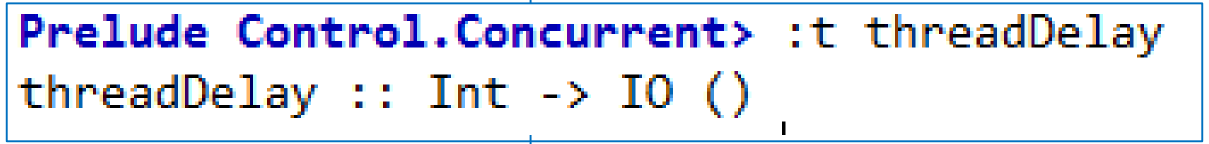
```
*Main> main
thread1
thread2
e
citit 1: e
thread1
s
citit 2:s
thread2
r
citit 1: r
thread1
e
citit 2:e
thread2
f
citit 1: f
thread1
f
```

reminders2.hs: thread-ul principal creaza thread-uri "reminder"

```
import Control.Concurrent
import Control.Monad

main =  do
        s <- getLine
        if (s =="exit")
          then return ()
          else do
                 forkIO $  setReminder s
                 main


setReminder :: String -> IO ()
setReminder s  = do
               let t = (read s) :: Int
               putStrLn  ("Ok, I'll remind you in "++s++  " seconds")
               threadDelay (10^6 * t)         -- suspenda threadul pentru t secunde
               putStrLn ("Reminder for "++s++"seconds is up! BING!\BEL")
```



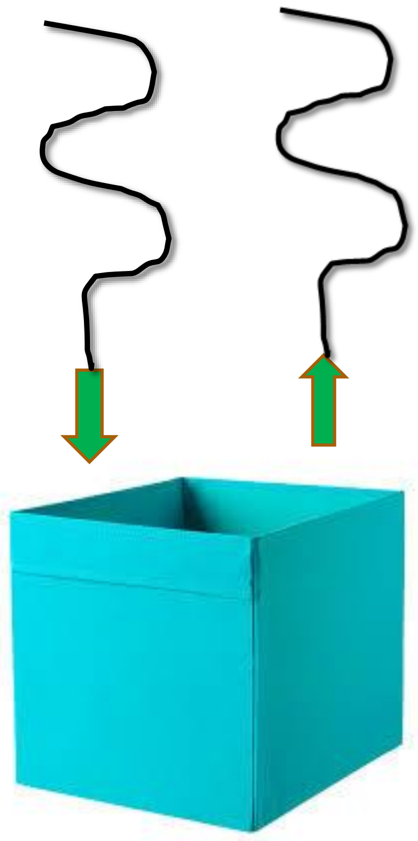
```
Prelude Control.Monad> :t forever
forever :: Monad m => m a -> m b
```



```
Prelude Control.Concurrent> :t threadDelay
threadDelay :: Int -> IO ()
```

Rulati exemplul!
Observati:
la "exit" thread-urile neexecutate sunt abandonate

➢ Comunicarea thread-urilor



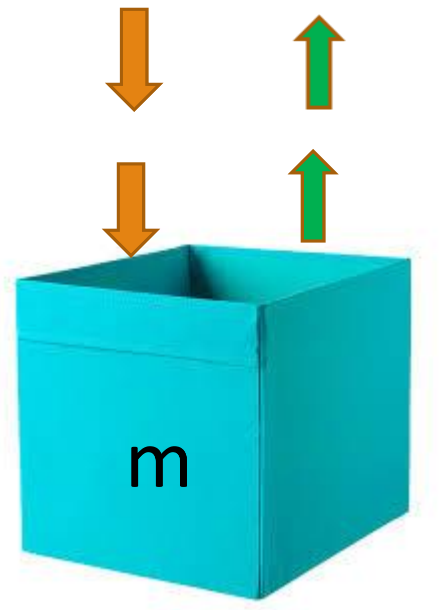
MVar

mutable variable

➢ Comunicarea folosind MVar se face in monada IO

data MVar a

- o data de tipul MVar a reprezinta o locatie mutabila care poate fi goala sau
- poate contine o singura valoare de tip a
- thread-urile pot comunica prin intermediul datelor de tip MVar



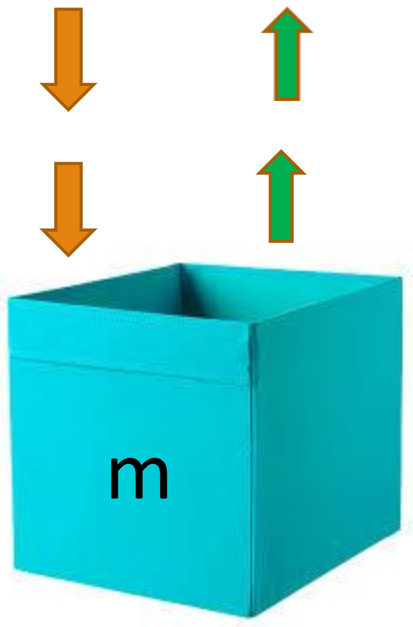
m :: MVar a

poate fi vazuta ca:

- un semafor binar
- un monitor cu o variabila

➢ Comunicarea folosind MVar se face in monada IO

```
data MVar a


newEmptyMVar :: IO (MVar a)  -- m <- newEmptyMVar
                              -- m este o locatie goala

newMVar :: a -> IO (MVar a)    -- m <- newMVar v
                              -- m este o locatie care contine valoarea v

takeMVar :: MVar a  -> IO a    -- v <- takeMVar m
                              -- intoarce in v valoarea din m
                              -- asteapta (blocheaza thread-ul)  daca m este goala

putMVar :: Mvar a -> a -> IO()  -- putMVar m v
                              -- pune in m valoarea v
                              -- asteapta (blocheaza thread-ul) daca m este plina
```



➢ takeMVar

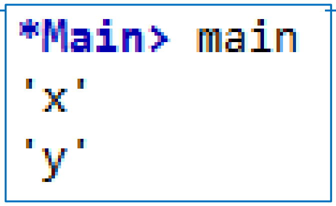
- takeMVar este o operatie care blocheaza thread-urile

- takeMVar este *single-wakeup*:
  daca variabila Mvar este goala, toate thread-urile care vor sa execute takeMVar sunt blocate; cand variabila devine plina, un singur thread este trezit si acesta va executa takeMVar

- daca mai multe thread-uri sunt blocate pe acelasi MVar, ele vor fi trezite in ordinea FIFO

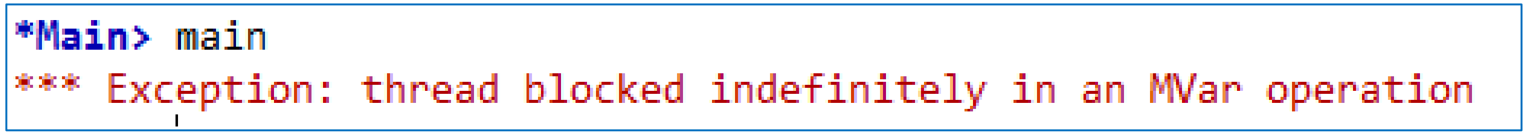
https://www.haskell.org/hoogle/?hoogle=MVar

```
import Control.Concurrent

main = do
        m <- newEmptyMVar
        forkIO $ do
                putMVar m 'x'
                putMVar m 'y'
        x <- takeMVar m
        print x
        x <- takeMVar m
        print x
```



```
*Main> main
'x'
'y'
```

```
import Control.Concurrent

main = do
        m <- newEmptyMVar
        takeMVar m
```



```
*Main> main
*** Exception: thread blocked indefinitely in an MVar operation
```

➢ takeMVar vs readMVar

readMVar

Citeste atomic continutul unui MVar.

Daca variabile Mvar este goala, thread-ul care apeleaza readMVar va astepta pana cand MVar primeste o valoare si va citi valoarea pusa de urmatoarea operatie putMVar.

readMVar este *multiple-wakeup*, deci toate threa-urile care asteapta sa citeasca din MVar vor fi trezite in acelasi timp.

Implementarea veche

```
readMVar :: MVar a -> IO a
readMVar m =   do
                  a <- takeMVar m
                  putMVar m a
                  return a
```

Implementarea actuala garanteaza ca readMVar este o operatie atomica.

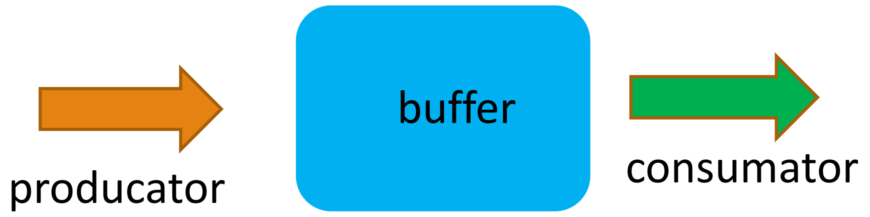
https://www.haskell.org/hoogle/?hoogle=MVar

➢ MVar ca semafor binar

```
newLock = newMVar ()      -- MVar care contine ()
aquireLock  m = takeMVar m
releaseLock  m = putMVar m ()
```

```
act1 m = do
            aquireLock m
            print "I have the lock"
            releaseLock m

act2 m = do
            aquireLock m
            print "Now I am have the  lock"
            releaseLock m

main = do
          m <- newLock
          forkIO $  act1 m
          forkIO $  act2 m
          getLine
```

- Producer-Consumer problem
  MVar ca monitor



```
producer :: MVar String-> IO ()
producer m = forever $ do
                        mes <- getLine
                        putMVar m mes
```

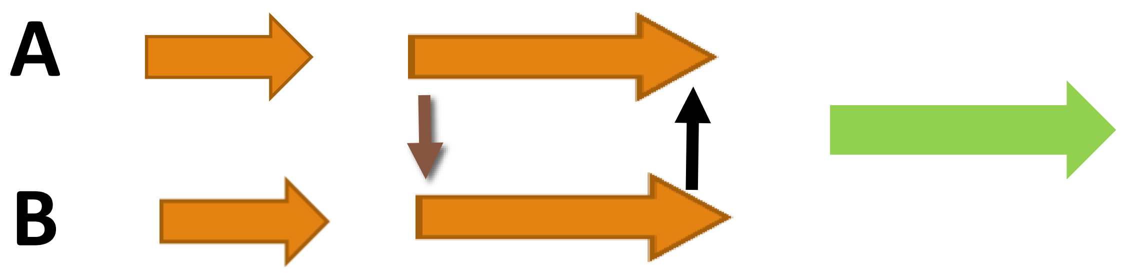
```
import Control.Concurrent
import Control.Monad


main = do
       m <- newEmptyMVar  --buffer
       forkIO (producer m )
       consumer  m 10
                    -- consuma 10 produse
```

```
consumer :: MVar String -> Int -> IO()
consumer m  n =  if (n == 0)
                 then return ()
                 else
                     do
                       mes <- takeMVar m
                       putStrLn  (">"++ mes)
                       consumer m (n-1)
```

➢ Rendezvous

primeste o valoare citita **msg**, o pune in **a** si afiseaza valoarea din **b**

citeste valoarea din **a** si pune in **b** dublul ei



```
import Control.Concurrent
import Control.Monad

main = do
      aMVar <- newEmptyMVar
      bMVar <- newEmptyMVar
      fMVar <- newEmptyMVar
      putStrLn "mesaj:"
      msg <- getLine
      forkIO (threadA aMVar bMVar fMVar msg)
      forkIO (threadB aMVar bMVar)
      takeMVar fMVar
```

fMVar functioneaza ca un semafor binary, asigurand sincronizarea thread-urilor

```
threadA a b f  msg =  do
                        putMVar a msg
                        y <- takeMVar  b
                        putStrLn ("raspuns:  " ++ y)
                        putMVar f ()


threadB a b  = do
                      x <- takeMVar a
                      let y =  x ++ x
                      putMVar b y
```

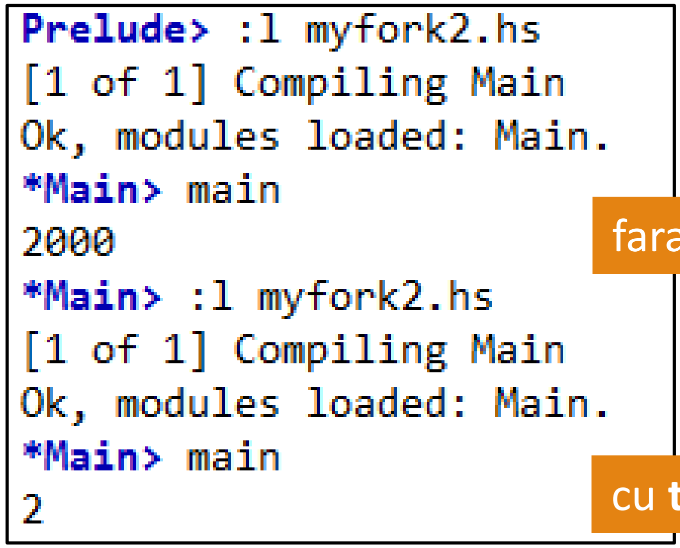
- **Sincronizare** : doua thread-uri incrementeaza acelasi contor
  vrem sa citim valoarea contorului dupa ce ambele thread-uri au terminat

```
add1 m  =  replicateM_ 1000 $ do
                    x <- takeMVar m
                    threadDelay 100
                    putMVar m (x +1)
```

```
add2 m  = replicateM_ 1000 $ do
                    s<- takeMVar m
                    putMVar m (s + 1)
```

```
main = do
        m <- newMVar 0
        forkIO (add1 m )
        forkIO (add2 m )
        x  <- takeMVar m
        print x
```

trebuie sa ne asiguram ca ambele thread-uri au terminat



```
Prelude> :l myfork2.hs
[1 of 1] Compiling Main
Ok, modules loaded: Main.
*Main> main
2000
*Main> :l myfork2.hs
[1 of 1] Compiling Main
Ok, modules loaded: Main.
*Main> main
2
```

fara **threadDelay**

cu **threadDelay**

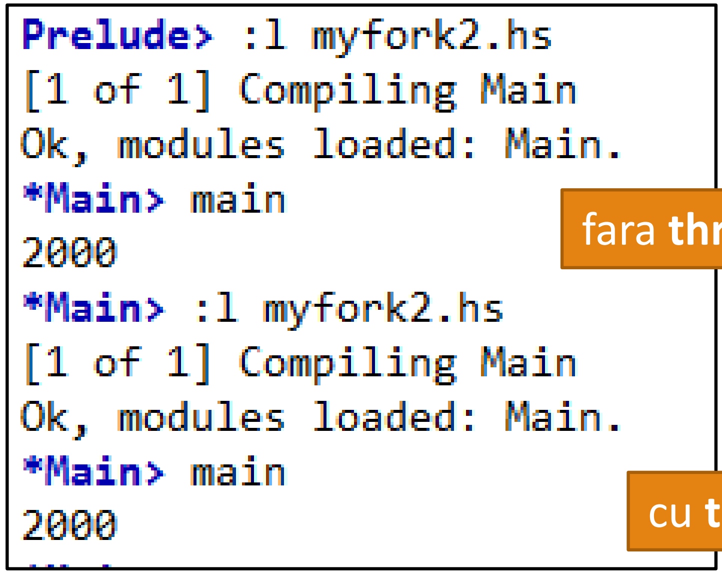
myfork2.hs

➢ Sincronizare

```
add1 m  ms1 = do
                  replicateM_ 1000 $ do
                                x <- takeMVar m
                                threadDelay 100
                                putMVar mv (x +1)
                  putMVar ms1 "ok"
```

```
add2 m ms2   = do
               replicateM_ 1000 $ do
                                  s<- takeMVar m
                                  putMVar mv (s + 1)
               putMVar ms2 "ok"
```

```
main = do
          m <- newMVar 0
          ms1 <- newEmptyMVar
          ms2 <- newEmptyMVar
          forkIO (add1 m ms1)
          forkIO (add2 m ms2)
          takeMVar ms1
          takeMVar ms2
          x  <- takeMVar m
          print x
```



```
Prelude> :l myfork2.hs
[1 of 1] Compiling Main
Ok, modules loaded: Main.
*Main> main
2000
*Main> :l myfork2.hs
[1 of 1] Compiling Main
Ok, modules loaded: Main.
*Main> main
2000
```

fara **threadDelay**

cu **threadDelay**

variabilele ms1 si ms2 actioneaza ca niste semafoare ; astfel ne asiguram ca ambele thread-uri au terminat