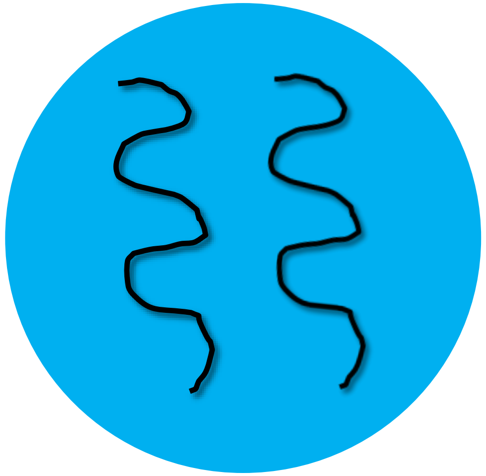
IMPLEMENTAREA CONCURENTEI IN LIMBAJE DE PROGRAMARE

Concurenta

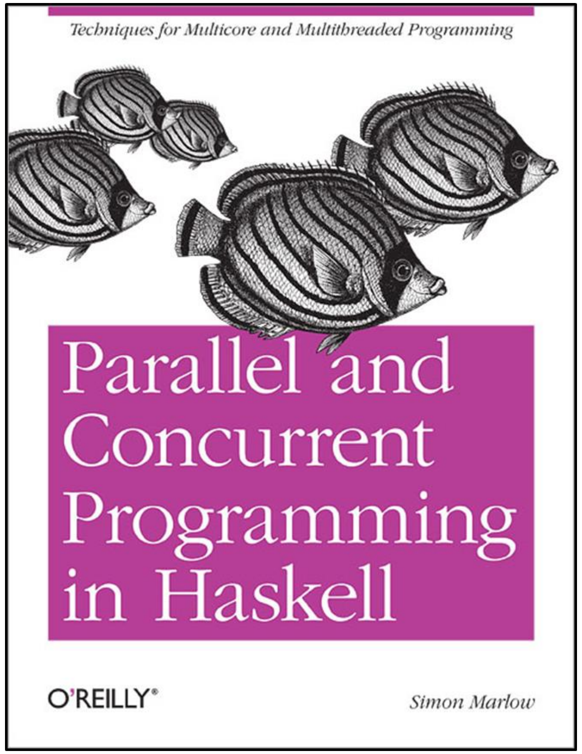
Threaduri

Memorie Partajata

Ioana Leustean

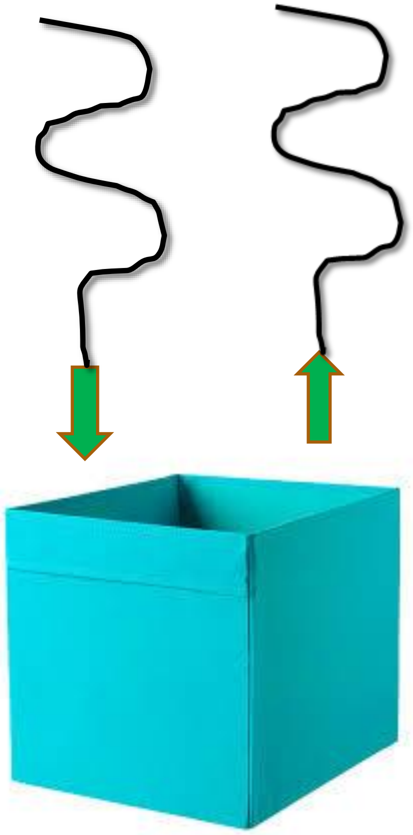


Haskell



[Part II. Concurrent Haskell]()
Cap.7 & 8

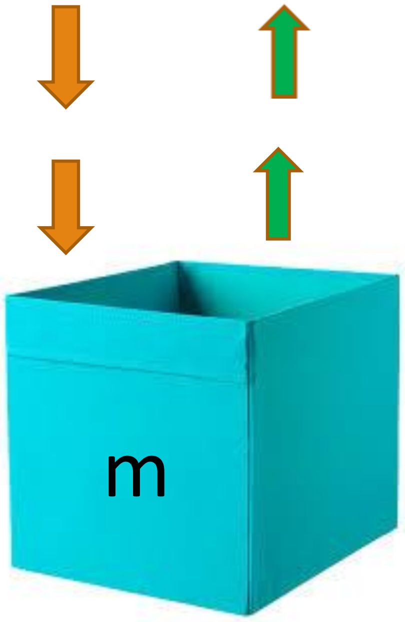
forkIO :: IO () -> IO ThreadId



MVar
mutable variable

➢ Comunicarea folosind MVar se face in monada IO

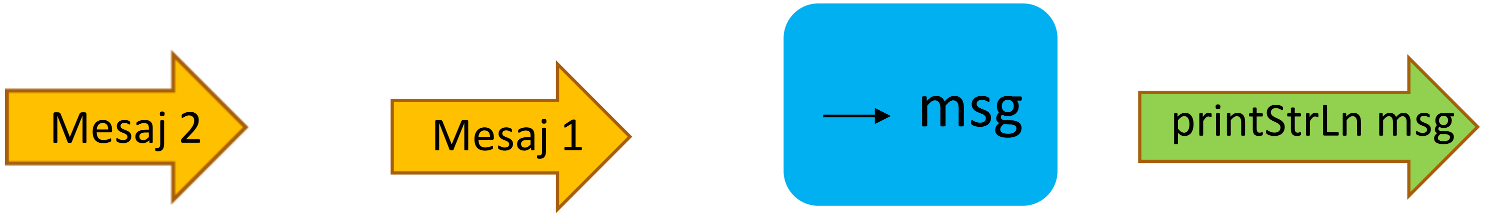
```
data MVar a

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
```



- Sincronizare: serviciu de logare

  modelarea unui canal de comunicare simplu folosind MVar

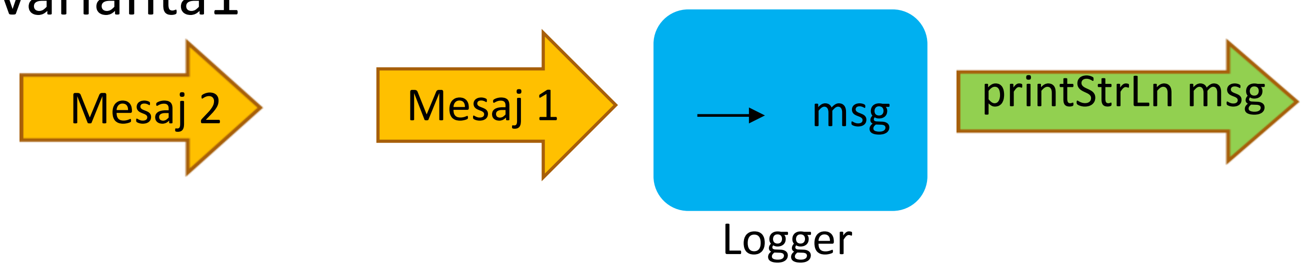
  http://chimera.labs.oreilly.com/books/1230000000929/ch07.html#sec_conc-logger



Cerinte:

- serviciul de logare prelucreaza mesajele intr-un thread separat
- mesajele trebuie prelucrate in ordinea in care sunt logate
- cand programul se termina toate mesajele logate trebuie sa fie prelucrate

Exemplu: serviciu de logare – varianta1




```
data Logger = Logger MVar  String

initLogger  ::  IO Logger
initLogger = do
                m <- newEmptyMVar

                let log = Logger m        -- log = 
```

```
                forkIO (logger log)       -- creeaza 
```

```
                return log

logger  :: Logger -> IO()  -- prelucreaza mesajele din Logger
```

```
logger :: Logger -> IO ()
logger (Logger m) = loop
                      where
                        loop = do
                               msg<- takeMVar m
                               putStrLn msg
                               loop
```

Exemplu: serviciu de logare- varianta1


```
logMessage :: Logger -> String -> IO ()
logMessage (Logger m) s = putMVar m s

logMessTh:: Logger -> IO()
logMessTh l = do
                msg <- getLine
                if (msg == "bye")
                then return()
                else do
                     logMessage log msg
                     logMessTh log
main = do
         log <- initLogger
         logMessTh log
```

Thread-ul principal trimite messajele

```
data Logger = Logger MVar  String

initLogger :: IO Logger
initLogger = do
             m <- newEmptyMVar
             let log = Logger m
             forkIO (logger log)
             return log
logger :: Logger -> IO ()
logger (Logger m) = loop
                    where
                       loop = do
                              msg<- takeMVar m
                              putStrLn msg
                              loop
```

Thread-ul logger le citeste si le scrie

Exemplu: serviciu de logare- varianta1


```
logMessage :: Logger -> String -> IO ()
logMessage (Logger m) s = putMVar m s

logMessTh:: Logger -> IO()
logMessTh log = do
                msg <- getLine
                if (msg == "bye")
                then return()
                else do
                     logMessage log msg
                     logMessTh log
main = do
         log <- initLogger
         logMessTh log
```

```
data Logger = Logger MVar  String

initLogger :: IO Logger
initLogger = do
             m <- newEmptyMVar
             let log = Logger m
             forkIO (logger log)
             return log
logger :: Logger -> IO ()
logger (Logger m) = loop
                    where
                      loop = do
                           msg<- takeMVar m
                           putStrLn msg
                           loop
```

programul nu se asigura ca toate mesajele logate sunt prelucrate

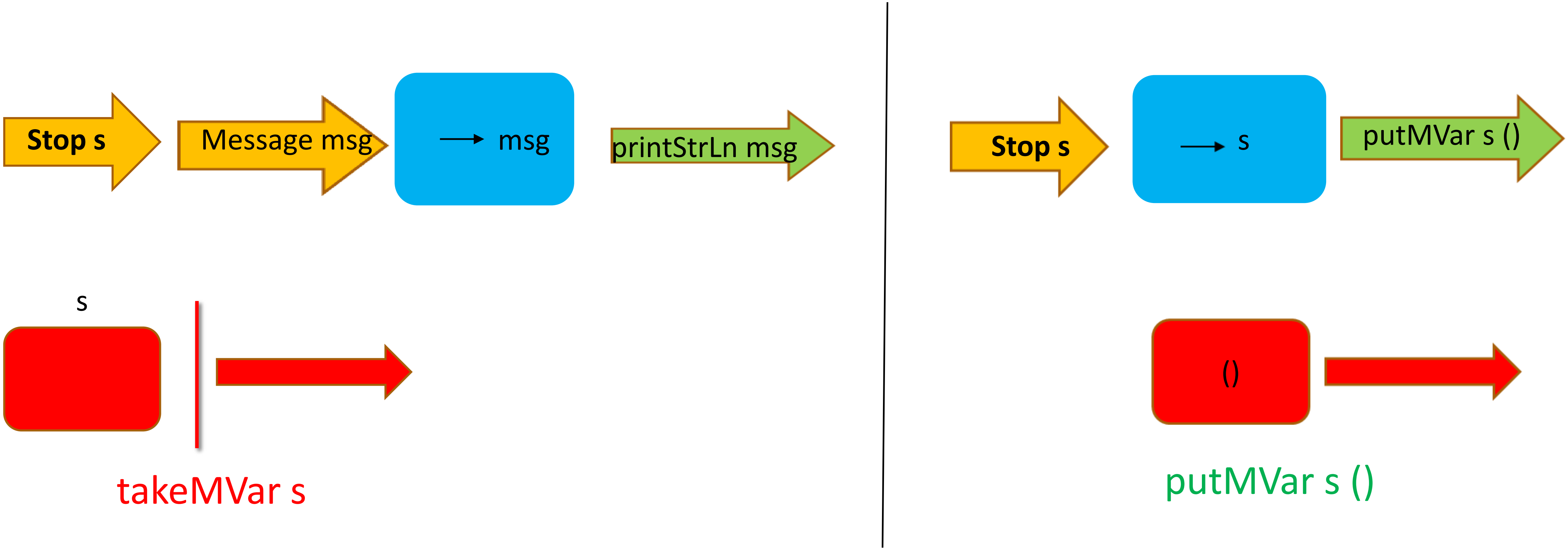
- Exemplu: serviciu de logare

```haskell
logMessage :: Logger -> String -> IO ()
logMessage (Logger m) s = putMVar m s

logMessTh:: Logger -> IO()
logMessTh log = do
                 msg <- getLine
                 if (msg == "bye")
                 then logStop log
                 else do
                      logMessage log msg
                      logMessTh log
```

```haskell
-- la fel

initLogger :: IO Logger
initLogger = do
               m <- newEmptyMVar
               let log= Logger m
               forkIO (logger log)
               return log

main = do
         log <- initLogger
         logMessTh log
```

Exemplu: serviciu de logare

```
data Logger = Logger (MVar LogCommand)
data LogCommand = Message String | Stop  (MVar ())
```



- Exemplu: serviciu de logare

```
data Logger = Logger (MVar LogCommand)
data LogCommand = Message String | Stop  (MVar ())
```

```
logMessage :: Logger -> String -> IO ()
logMessage (Logger m) s = putMVar m s

logMessTh:: Logger -> IO()
logMessTh log = do
              msg <- getLine
              if (msg == "bye")
              then logStop log
              else do
                  logMessage log msg
                  logMessTh log
```

```
logStop :: Logger -> IO ()
logStop (Logger m) = do
              s <- newEmptyMVar
              putMVar m (Stop s)
              takeMVar s
```

Exemplu: serviciu de logare

```
data Logger = Logger (MVar LogCommand)
data LogCommand = Message String | Stop  (MVar ())
```

```
logger :: Logger -> IO ()
logger (Logger m) = loop
        where  loop = do
                    cmd <- takeMVar m
                    case cmd of
                      Message msg -> do
                                  putStrLn  ("mesaj: " ++ msg)
                                  loop
                      Stop s -> do
                            putStrLn "logger: stop"
                            putMVar s ()
```

Thread-ul logger va debloca s cand cand ajunge la Stop s

```
logStop :: Logger -> IO ()
logStop (Logger m) = do
            s <- newEmptyMVar
            putMVar m (Stop s)
            takeMVar s
```

logger.hs ©2012, Simon Marlow

```
*Main> main
mes:
mes1
mesm:e
saj: mes1
mes2
memseasj::
 mes2
mes3
mesm:e
saj: mes3
bye
```

Atentie!
Accesul la stdout nu este thread-safe, deci trebuie sincronizat

```haskell
stdo <- newMVar ()

tswrite stdo s = do
          takeMVar stdo
          putStrLn s
          putMVar stdo ()
```

```
*Main> main
mes:
mesajul 1
mesaj: mesajul 1
mes:
mesajul 2
mes:
mesaj: mesajul 2
mesajul 3
mes:
mesaj: mesajul 3
mesajul 4
mes:
mesaj: mesajul 4
bye
```

Semafoare

```
import Control.Concurrent.QSem

data  QSem

newQSem :: Int -> IO  Qsem

waitQSem ::  QSem -> IO()        -- aquire, il ocupa
signalQSem :: QSem -> IO()       -- release, il elibereaza
```

un semafor care sincronizeaza accesul la n resurse se defineste astfel:

```
qs <- newQsem  n
```

Exemplu: qsemrcmy.hs

O multime de taskuri acceseaza simultan o resursa reprezentata printr-un **QSem**;
pentru a se executa, fiecare task trebuie sa acceseze resursa, pe care o elibereaza la sfarsitul executiei.

```
import Control.Concurrent
import Control.Monad


main :: IO ()
main = do
        q <- newQSem 3
        stdo <- newEmptyMVar
        let workers = 5
            prints  = 2 * workers
        mapM_ (forkIO . worker q m) [1..workers]
        replicateM_ prints $ takeprint  stdo
```

```
takeprint :: MVar String  -> IO()
takeprint stdo = do
          s <- takeMVar stdo
          print s

worker ::  QSem -> MVar String -> Int -> IO ()
worker q m w= do
   waitQSem q
   putMVar stdo $ "Worker " ++ show w ++ " acquired the lock."
   threadDelay 2000000      -- microseconds
   signalQSem q
   putMVar stdo $ "Worker " ++ show w ++ "released the lock."
```

q este semaforul care controleaza resursele
stdo coordoneaza accesul la stdout

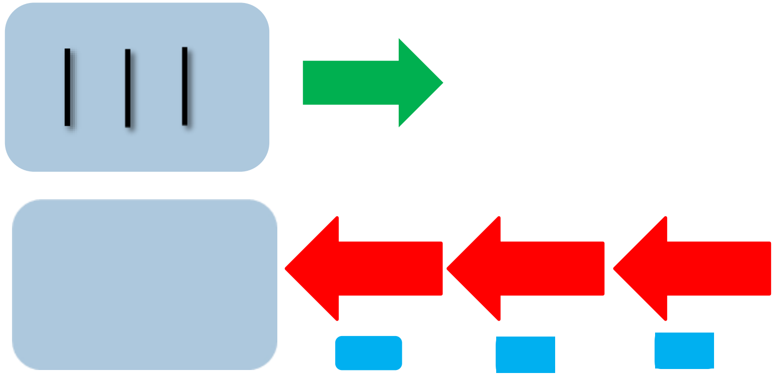
http://rosettacode.org/wiki/Metered_concurrency

```
Prelude> :l qsemrcmy.hs
[1 of 1] Compiling Main             ( qsemrcmy.hs, interpreted )
Ok, modules loaded: Main.
*Main> main
"Worker 1 has acquired the lock."
"Worker 2 has acquired the lock."
"Worker 3 has acquired the lock."
"Worker 2 has released the lock."
"Worker 3 has released the lock."
"Worker 1 has released the lock."
"Worker 5 has acquired the lock."
"Worker 4 has acquired the lock."
"Worker 4 has released the lock."
"Worker 5 has released the lock."
```

```
*Main> main
"Worker 1 has acquired the lock."
"Worker 2 has acquired the lock."
"Worker 3 has acquired the lock."
"Worker 1 has released the lock."
"Worker 5 has acquired the lock."
"Worker 2 has released the lock."
"Worker 4 has acquired the lock."
"Worker 3 has released the lock."
"Worker 4 has released the lock."
"Worker 5 has released the lock."
```

in Concurrent Haskell
concurenta este nedeterminista

➢ Implementarea QSem



```
type  QSem = MVar (Int, [MVar ()])

newQSem :: Int -> IO  QSem

newQSem n =  newMVar (n,[])
                    --  qsem <- newQSem 3

waitQSem ::  QSem -> IO()        -- ocupa
signalQSem :: QSem -> IO()       -- elibereaza
```

n = nr. de resurse
blki = un thread care cere acces la resursa
este blocat pe variabila blki

daca n > 0 atunci qsem = (n, [])
altfel qsem = (0, [blk1, blk2, …])

Implementarea din:
Concurrent Haskell
SL Peyton Jones, A Gordon, S Finne, 1996

➢ Implementarea QSem - *Concurrent Haskell SL Peyton Jones, A Gordon, S Finne, 1996*

```haskell
type  QSem = MVar (Int, [MVar ()])

newQSem :: Int -> IO  QSem
newQSem n =  newMVar (n,[])
```

daca n > 0 atunci qsem = (n, [])
altfel qsem = (0, [blk1, blk2, ...])

ocuparea resursei

```haskell
waitQSem ::  QSem -> IO()
waitQSem qsem = do
              (avail,blkd) <- takeMVar qsem
              if avail > 0
                   then    putMVar qsem (avail-1, [])
                   else
                       do
                         blk <- newEmptyMVar
                         putMVar qsem (0,blk:blkd)
                         takeMVar blk  – threadul e blocat pe
                                         variabila proprie
```

eliberarea resursei

```haskell
signalQSem :: QSem -> IO()
signalQSem qsem = do
                        (avail,blkd) <- takeMVar qsem
                        case blkd of
                           [] -> putMVar qsem (avail+1,[])
                           (blk:blkd') -> do
                                    putMVar qsem (0,blkd')
                                    putMVar blk ()
```

fiecare thread elibereaza variabila
proprie a unui thread in asteptare

- Readers/Writers problem

  - Mai multe threaduri au acces la o resursa.
  - Unele threaduri scriu (writers), iar altele citesc (readers).
  - Resursa poate fi accesata simultan de mai multi cititori.
  - Resursa poate fi acessata de un singur scriitor.
  - Resursa nu poate fi accesata simultan de cititori si de scriitori.

```
import  Control.Concurrent.ReadWriteLock
new  :: IO RWLock
acquireRead ::  IO RWLock -> IO ()
releaseRead ::  IO RWLock -> IO ()
acquireWrite ::  IO RWLock -> IO ()
releaseWrite ::  IO RWLock -> IO ()
```

➢ Readers/Writers problem

Mai multe threaduri au acces la o resursa.
Unele threaduri scriu (writers), iar altele citesc (readers).
Resursa poate fi accesata simultan de mai multi cititori.
Resursa poate fi acessata de un singur scriitor.
Resursa nu poate fi accesata simultan de cititori si de scriitori.

Pentru sincronizare folosim:

- un semafor binar care da acces la citit sau la scris: writeL
- un monitor in care se inregistreaza nr. de cititori: readL

```
data MyRWLock = MyRWL {readL :: MVar Int, writeL :: MyLock}
```

➢ Reader/Writer Lock

```
type MyLock = MVar ()
newLock = newMVar ()
aquireLock  m = takeMVar m
releaseLock m = putMVar m ()
```

```
data MyRWLock = MyRWL {readL :: MVar Int, writeL :: MyLock}
newMyRWLock :: IO MyRWLock
newMyRWLock = do
                readL  <-  newMVar 0
                writeL <-  newLock
                return (MyRWL readL writeL)
```

```
aquireWrite :: MyRWLock -> IO ()
aquireWrite (MyRWL readL  writeL) = aquireLock writeL

releaseWrite :: MyRWLock -> IO ()
releaseWrite (MyRWL readL writeL) = releaseLock writeL
```

➢ Reader/Writer Lock

```
data MyRWLock = MyRWL {readL :: MVar Int, writeL :: MyLock}
```

```
aquireRead :: MyRWLock -> IO ()
aquireRead (MyRWL readL writeL) = do
                        n <- takeMVar readL  -- n readers
                        if (n == 0)   then do
                                           aquireLock writeL
                                           putMVar readL 1
                        else  putMVar readL (n+1)

releaseRead :: MyRWLock -> IO ()
releaseRead (MyRWL readL writeL) = do
                        n <- takeMVar readL
                        if (n == 1)   then do
                                           releaseLock writeL
                                           putMVar readL 0
                        else  putMVar readL (n-1)
```

- Exemplu: Readers/Writers

```
genread n rwl lib = if (n==0)
                    then putStrLn "no more readers"
                    else do
                          reader n rwl lib
                          threadDelay 20
                         genread (n-1) rwl lib
genwrite n rwl lib = if (n==0)
                    then putStrLn "no more writers"
                    else do
                          writer n rwl lib
                          threadDelay 100
                          genwrite (n-1) rwl lib

main = do
         lib <- newMVar 0    -- resursa
         rwl <- newMyRWLock
         forkIO $  genread 10 rwl lib
         forkIO $  genwrite 5 rwl lib
         getLine
```

```
reader i rwl lib = do
                  aquireRead rwl
                  c <- readMVar lib -- non blocking
                  putStrLn $  (show i) ++ (show c)
      --  "Reader " ++ (show i) ++ " reads: " ++ (show c)
                  releaseRead rwl

writer i rwl lib = do
                   aquireWrite rwl
                   putStrLn $ show  i
      --    "Writer " ++ (show i) ++ " writes "  (show i)
                    c <- takeMVar lib
                    putMVar lib i
                    releaseWrite rwl
```

➢ Readers/Writers

```
genread n rwl lib = if (n==0)
                        then putStrLn "no more readers"
                        else do
                              reader n rwl lib
                              threadDelay 20
                             genread (n-1) rwl lib
genwrite n rwl lib = if (n==0)
                        then putStrLn "no more writers"
                        else do
                              writer n rwl lib
                              threadDelay 100
                              genwrite (n-1) rwl lib

main = do
         lib <- newMVar 0     -- resursa
         rwl <- newMyRWLock
         forkIO $  genread 10 rwl lib
         forkIO $  genwrite 5 rwl lib
         getLine
```

```
Prelude>  :l myrw.hs
[1 of 1] Compiling Main
Ok, modules loaded: Main.
*Main> main
Reader 10 reads: 0
Writer 5 writes 5
Reader 9 reads: 5
Reader 8 reads: 5
Writer 4 writes 4
Reader 7 reads: 4
Reader 6 reads: 4
Writer 3 writes 3
Reader 5 reads: 3
Writer 2 writes 2
Reader 4 reads: 2
Writer 1 writes 1
Reader 3 reads: 1
no more writers
Reader 2 reads: 1
Reader 1 reads: 1
no more readers
```

➢ Canale de comunicare: canale implementate cu MVar

readChan
se
blocheaza
cand
canalul
este gol

writeChan
nu se blocheaza
niciodata

```
import Control.Concurrent.Chan

newChan :: IO (Chan a)

writeChan :: Chan a -> a -> IO ()

readChan :: Chan  a -> IO a
```

- Exemplu: doua canale: **cin** si **cout**

- thread –ul parinte citeste siruri si le pune pe canalul **cin**.
- un thread citeste sirurile de pe **cin**, le imparte in cuvinte iar cuvintele le pune pe canalul **cout**.
- un alt thread ia cuvintele de pe **cout**, si le scrie la iesire cu litere mari

```
import Control.Monad
import Control.Concurrent
import Data.Char

mymain = do
          cin  <- newChan
          cout <- newChan
          forkIO $ forever (move cin cout)
          forkIO $ forever (upout cout)
          load cin
```

```
move c1 c2 = do
            v1 <- readChan c1
            let ls = words v1
            mapM_ (writeChan c2) ls

upout c  = do
            str <- readChan c
            putStrLn (map toUpper str)

load c = do
            str <- getLine
            if (str == "exit")
            then return()
            else do
                  writeChan c str
                  load
```

➢ Canale de comunicare formate din variabile MVar

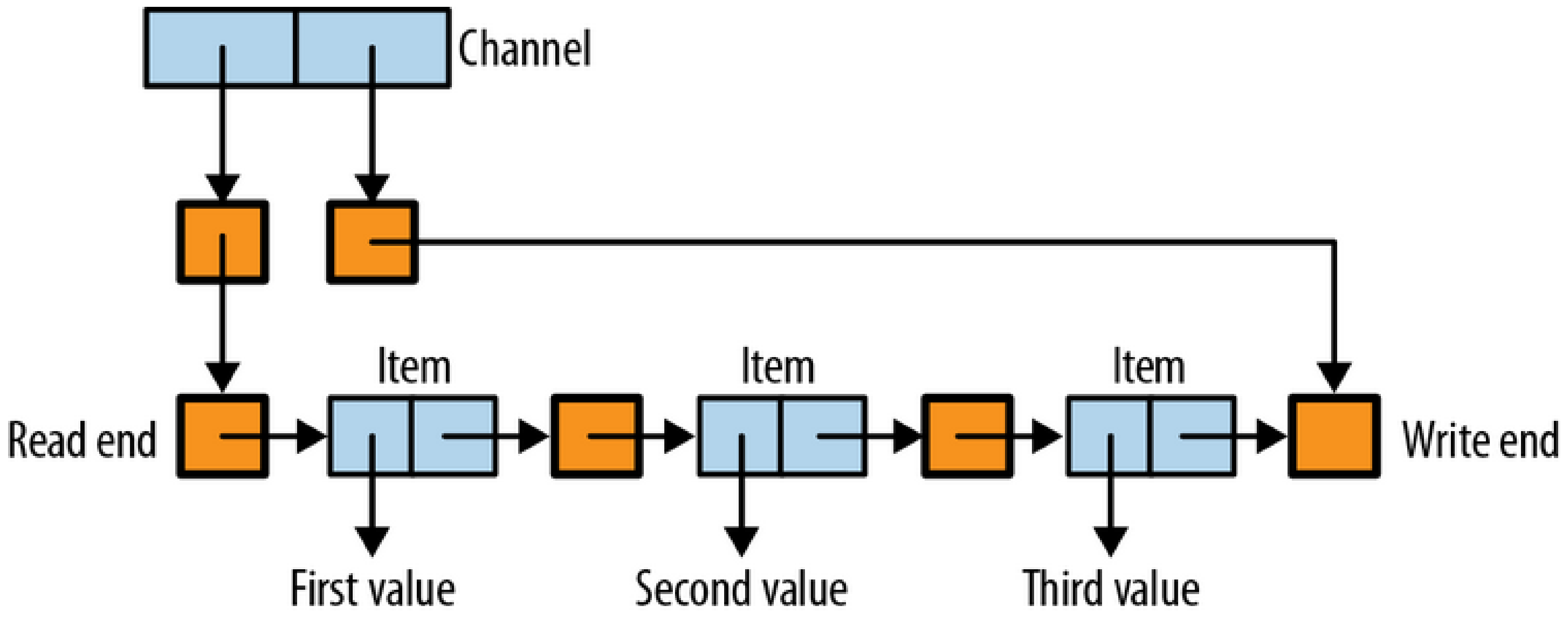


Figure 7-1. Structure of the buffered channel implementation

http://chimera.labs.oreilly.com/books/1230000000929/ch07.html#sec_channels

Canale formate din variabile MVar

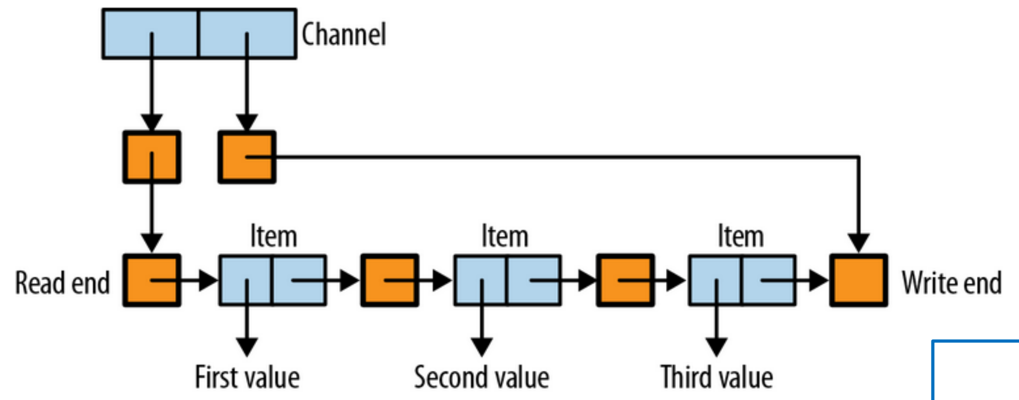
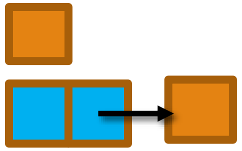


Figure 7-1. Structure of the buffered channel implementation

```
type Stream a = MVar (Item a)
data Item a   = Item a (Stream a)

data Chan a = Chan (MVar (Stream a)) (MVar (Stream a))

c <- newChan
v <- readChan c
putChan c v
```



chan.hs ©2012, Simon Marlow

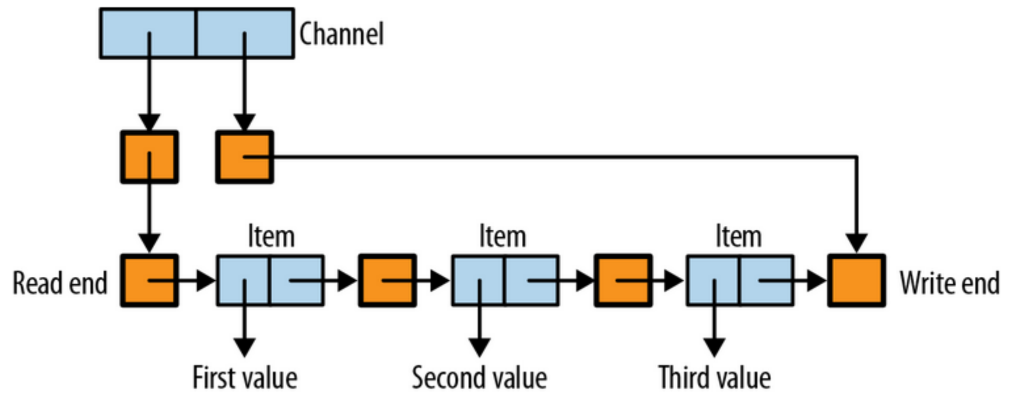
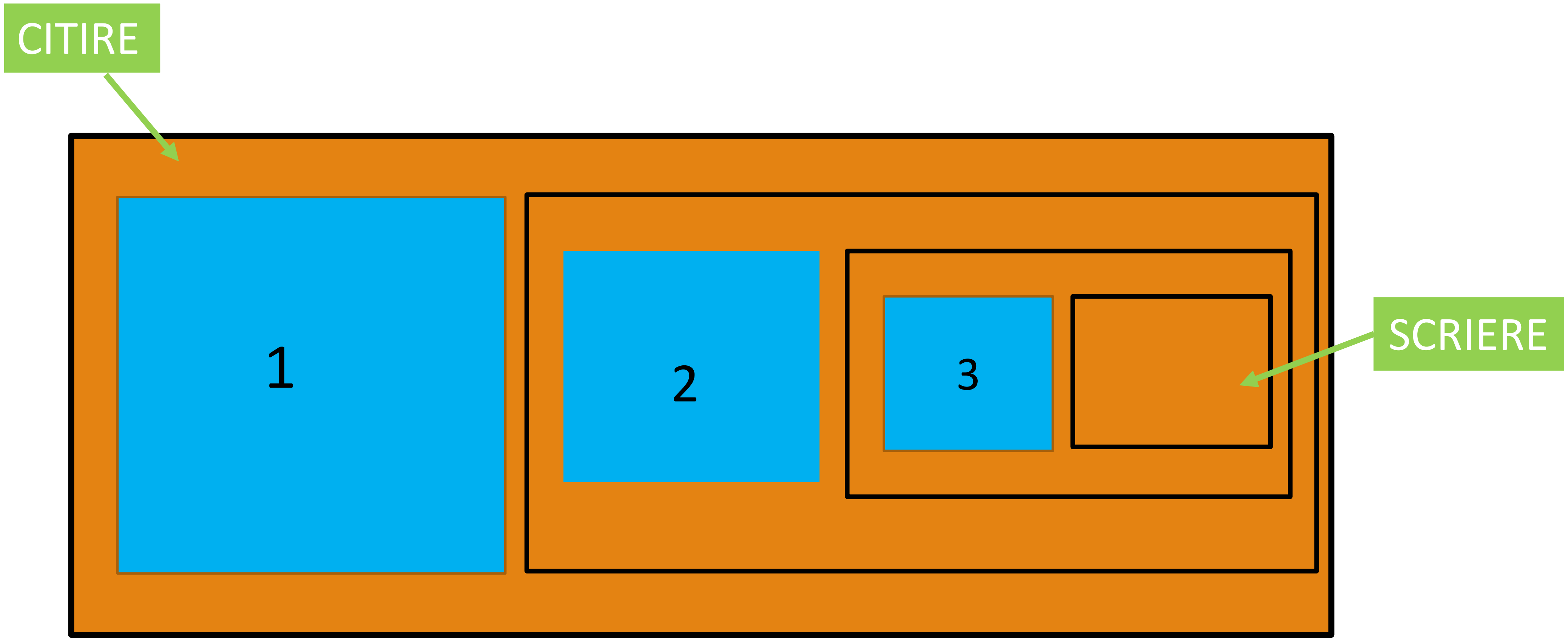


Figure 7-1. Structure of the buffered channel implementation

```
type Stream a = MVar (Item a)
data Item a   = Item a (Stream a)
data Chan a = Chan (MVar (Stream a)) (MVar (Stream a))

c <- newChan
v <- readChan c
putChan c v
```

"If multiple threads concurrently call readChan, the first one will successfully call takeMVar on the read end, but the subsequent threads will all block at this point until the first thread completes the operation and updates the read end. If multiple threads call writeChan, a similar thing happens: the write end of the Chan is the synchronization point, allowing only one thread at a time to add an item to the channel. However, the read and write ends, being separate MVars, allow concurrent readChan and writeChan operations to proceed without interference."

http://chimera.labs.oreilly.com/books/1230000000929/ch07.html#sec_channels


CITIRE
1
2
3
SCRIERE

Implementarea canalelor

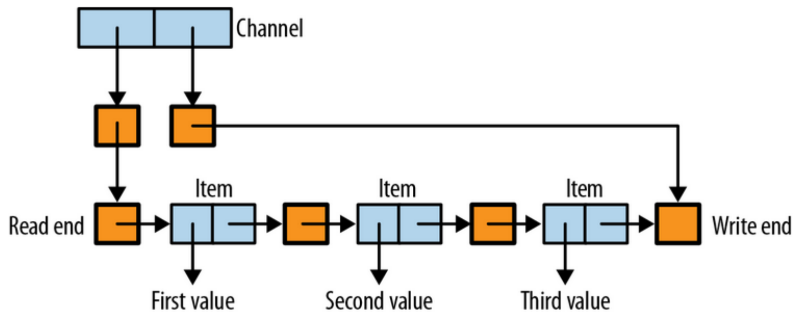


Figure 7-1. Structure of the buffered channel implementation

```
type Stream a = MVar (Item a)
data Item a   = Item a (Stream a)
data Chan a = Chan (MVar (Stream a)) (MVar (Stream a))
```

```
newChan :: IO(Chan a)
newChan = do
    emptyStream <-  newEmptyMVar
    readVar <- newMVar emptyStream
    writeVar <-newMVar emptyStream
    return (Chan readVar writeVar)
```

contine Item-ul care va fi citit

contine variabila in care se va scrie noul Item

```
readChan :: Chan a -> IO a
readChan (Chan rV wV) = do
                stream <- takeMVar rV
                Item val str <- takeMVar stream
                putMVar rV <- str
                return val
```

```
writeChan :: Chan a  -> a -> IO()
writeChan (Chan rV wV) val = do
            newStream <- newEmptyMVar
            writeEnd <- takeMVar wV
            putMVar writeEnd (Item val newStream)
            putMVar wV newStream
```

http://chimera.labs.oreilly.com/books/1230000000929/ch07.html#sec_channels

➢ Implementarea canalelor

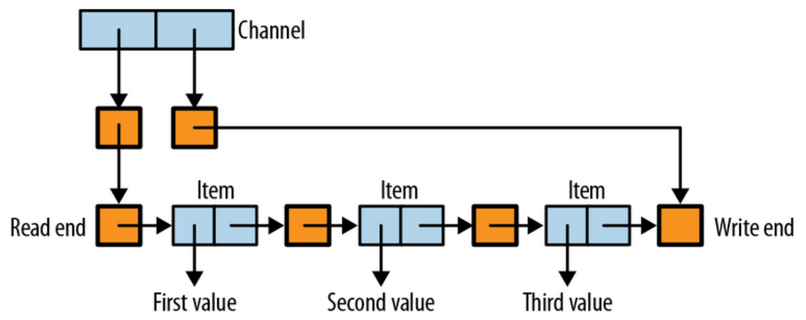


Figure 7-1. Structure of the buffered channel implementation

```
type Stream a = MVar (Item a)
data Item a   = Item a (Stream a)
data Chan a = Chan (MVar (Stream a)) (MVar (Stream a))
newChan :: IO (Chan a)
writeChan :: Chan a -> a -> IO ()
 readChan :: Chan a -> IO a
```

```
dupChan  ::  Chan a -> IO (Chan a)
```

- noul canal este initial gol
- dupa crearea canalului duplicat, ceea ce se scrie pe oricare din canale poate fi citit de pe oricare cele doua canale
- citirea de pe un canal nu elimina elementul de pe celalalt canal.

```
main = do  c <- newChan
           writeChan c 'a'
           readChan c >>= print
           c2 <- dupChan c
           writeChan c 'b'
           readChan c >>= print
           readChan c2 >>= print
```

```
Prelude> :l chan2.hs
[1 of 1] Compiling Main
Ok, modules loaded: Main.
*Main> main
'a'
'b'
'b'
```

http://chimera.labs.oreilly.com/books/1230000000929/ch07.html#sec_channels

➢ Implementarea canalelor

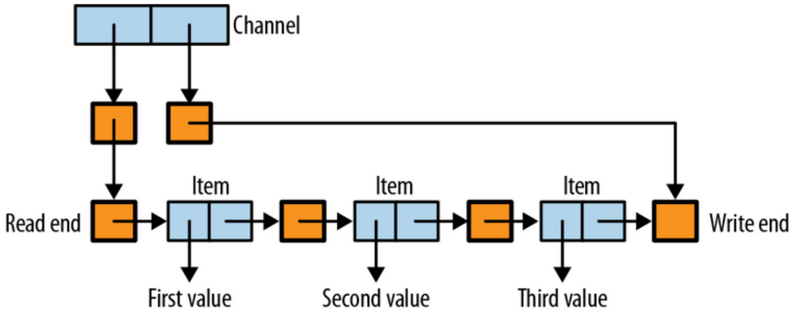


Figure 7-1. Structure of the buffered channel implementation

```
type Stream a = MVar (Item a)
data Item a   = Item a (Stream a)
data Chan a = Chan (MVar (Stream a)) (MVar (Stream a))
newChan :: IO (Chan a)
writeChan :: Chan a -> a -> IO ()
 readChan :: Chan a -> IO a
```

```
 dupChan  ::  Chan a -> IO (Chan a)
dupChan (Chan _ writeVar) = do
                    writeEnd <- readMVar writeVar
                    newReadVar <- newMVar writeEnd
                    return (Chan newReadVar writeVar)
```

```
readChan :: Chan a -> IO a
readChan (Chan rV wV) = do
             stream  <- takeMVar rV
             Item val str <- readMVar  stream --!!
             putMVar rV  str
             return val
```

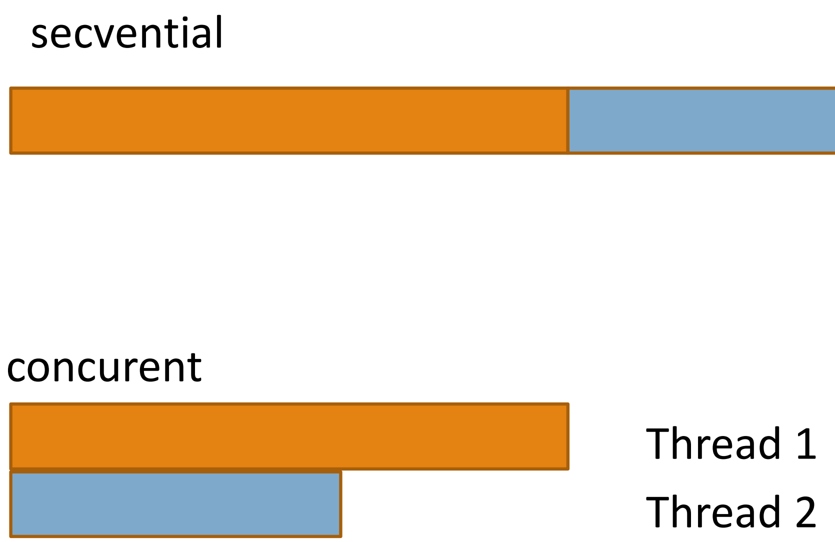
readMVar este necesar deoarece continutul trebuie sa ramana accesibil celuilalt canal.

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

http://chimera.labs.oreilly.com/books/1230000000929/ch07.html#sec_channels

"**Concurrent computing** is a form of computing in which several computations are executing during overlapping time periods—*concurrently*—instead of *sequentially* (one completing before the next starts)[…]
A *concurrent system* is one where a computation can advance without waiting for all other computations to complete; where more than one computation can advance at *the same time*."
Operating System Concepts 9th edition, Abraham Silberschatz

Exemplu: incarcarea mai multor pagini web




```
import Data.ByteString as B
import GetURL     -- parconc-examples
main = do
        m1 <- newEmptyMVar
        forkIO $ do
                r  <- getURL "http://..."
                putMVar m1 r

        m2 <- newEmptyMVar
        forkIO $ do
                r <- getURL "http://..."
                putMVar m2 r

        r1 <- takeMVar m1
        r2 <- takeMVar m2

     print (B.length r1, B.length r2)
```

geturl1.hs ©2012, Simon Marlow

➢ Comunicare asincrona

Se creaza un thread separat pentru fiecare actiune si se asteapta rezultatul

```
m1 <- newEmptyMVar
forkIO $ do
            r  <- getURL "http://www.fmi.ro "
            putMVar m1  r
r1 <- takeMVar m
```

```
a <- async (getURL "http://www.fmi.ro " )
r <- wait a
```

```
data Async a = Async (MVar a)

async :: IO a -> IO (Async a)
async action = do
  var <- newEmptyMVar
  forkIO (do r <- action; putMVar var r)
  return (Async var)

wait :: Async a -> IO a
wait (Async var) = readMVar var
```

readMVar nu blocheaza threadul, deci mai multe apeluri **wait** pot fi facute pentru aceeasi operatie asincrona

➢ Comunicare asincrona

```
import Control.Concurrent
import Text.Printf
import qualified Data.ByteString as B
import GetURL  --  parconc-examples
import TimeIt    --   parconc-examples

timeDownload :: String -> IO ()
timeDownload url = do
        (page, time) <- timeit $ getURL url
        printf " %s (%d bytes, %.2fs)\n" url (B.length page) time
```

```
data Async a = Async (MVar a)

async :: IO a -> IO (Async a)
async action = do
    var <- newEmptyMVar
    forkIO (do r <- action; putMVar var r)
    return (Async var)

wait :: Async a -> IO a
wait (Async var) = readMVar var
```

```
sites =["url1","url2",...]
main =  do
            as <- mapM (async . timeDownload) sites
            mapM_ wait as
```

geturl3.hs ©2012, Simon Marlow

➢ Comunicare asincrona

In exemplul anterior vrem sa scriem un mesaj cand s-a descarcat prima pagina

```
sites = ["url1",  "url2", ...]

download m url = do
                    r <- getURL url
                    putMVar m (url, r)

main :: IO ()main = do
                      m <- newEmptyMVar
                      mapM_ (forkIO . download m)  sites
                      (url, r) <- takeMVar m
                      printf "%s was first (%d bytes)\n" url (B.length r)
                      replicateM_ (length sites - 1) (takeMVar m)
```

threadul principal va accesa variabila **m** in momentul in care primeste o valoare

geturl5.hs ©2012, Simon Marlow

➢ Async - comunicare asincrona (folosind MVar)

geturl3.hs ©2012, Simon Marlow

```
import Control.Concurrent
import Text.Printf
import qualified Data.ByteString as B
import GetURL  --  parconc-examples
import TimeIt   --   parconc-examples

timeDownload :: String -> IO ()
timeDownload url = do
        (page, time) <- timeit $ getURL url
        printf " %s (%d bytes, %.2fs)\n" url (B.length page) time
```

```
data Async a = Async (MVar a)

async :: IO a -> IO (Async a)
async action = do
    var <- newEmptyMVar
    forkIO (do r <- action; putMVar var r)
    return (Async var)

wait :: Async a -> IO a
wait (Async var) = readMVar var
```

```
main =  do
        as <- mapM (async . timeDownload) sites     -- sites =["url1","url2",...]
        mapM_ wait as
```

asteapta ca toate actiunile asincrone sa se termine, monitorizand fiecare actiune in parte; un alt thread ar putea interveni inainte ca toate actiunile sa se termine

Vom rezolva acesta problema folosind STM