

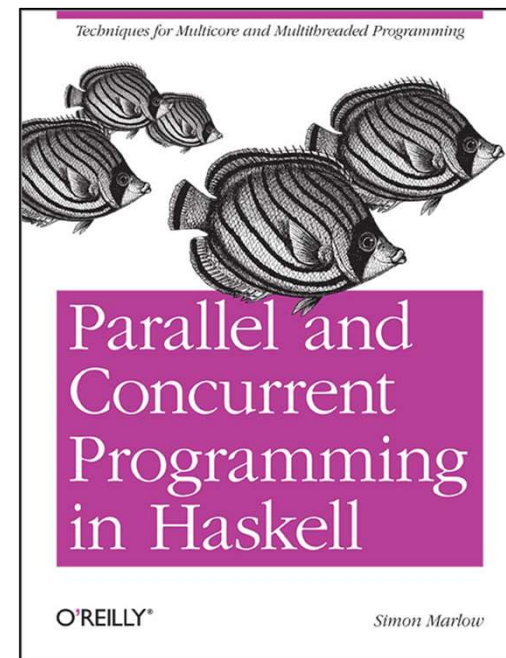
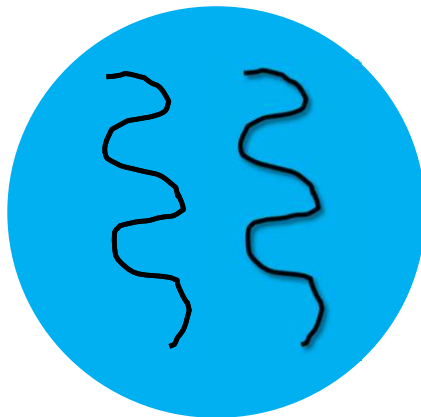
IMPLEMENTAREA CONCURENTEI IN LIMBAJE DE PROGRAMARE

Concurenta

Threaduri

Memorie Partajata

Ioana Leustean



Part II. Concurrent Haskell
Cap.7 & 8

<https://hackage.haskell.org/package/base-4.20.0.1/docs/Control-Concurrent.html>

➤ data MVar a

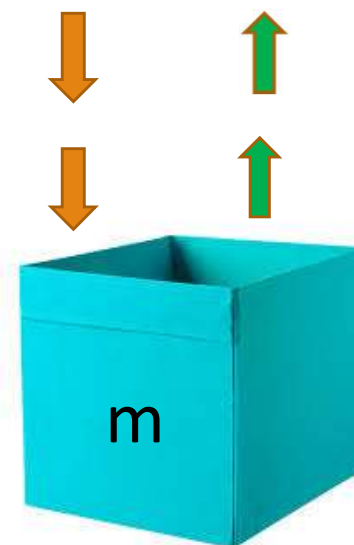
`newEmptyMVar :: IO (MVar a)` -- `m <- newEmptyMVar`
-- `m` este o locatie goala

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

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

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

`readMVar :: MVar a -> IO a` -- `v <- readMVar m` citeste atomic continutul lui `m`
-- intoarce in `v` valoarea din `m` dar aceasta ramane in `m`
-- asteapta (blocheaza thread-ul) daca `m` este goala



➤ Cititori/scriitori (Readers/Writers)

- 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 accesata 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 ()
```



➤ Cititori/scriitori (Readers/Writers)

Mai multe thread-uri au acces la o resursa.

Unele thread-uri scriu (writers), iar altele citesc (readers).

Resursa poate fi accesata simultan de mai multi cititori.

Resursa poate fi accesata 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 :: MVar ()}
```



➤ Reader/Writer Lock

```
type MyLock = MVar ()  
newLock = newMVar ()  
acquireLock 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)
```



➤ Reader/Writer Lock

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

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

```
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}
```

```
acquireRead :: MyRWLock -> IO ()
acquireRead (MyRWL readL writeL) = do
    n <- takeMVar readL    -- n cititori
    if (n == 0) then do
        acquireLock writeL
        putMVar readL 1
    else putMVar readL (n+1)
```



➤ Reader/Writer Lock

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

```
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

lib este resursa partajata

rwl este lacatul care sincronizeaza accesul la resursa

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



➤ Exemplu: Readers/Writers

lib este resursa partajata

rwl este lacatul care sincronizeaza accesul la resursa

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

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



➤ 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  -- lacatul rw
    forkIO $ genread 10 rwl lib  -- creez 10 thread-uri cititor
    forkIO $ genwrite 5 rwl lib  -- creez 5 thread-uri scriitor
    getLine
```

```
reader i rwl lib = do
    acquireRead rwl
    c <- readMVar lib
    putStrLn $ (show i) ++ (show c)
    releaseRead rwl

writer i rwl lib = do
    acquireWrite rwl
    putStrLn $ 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
```

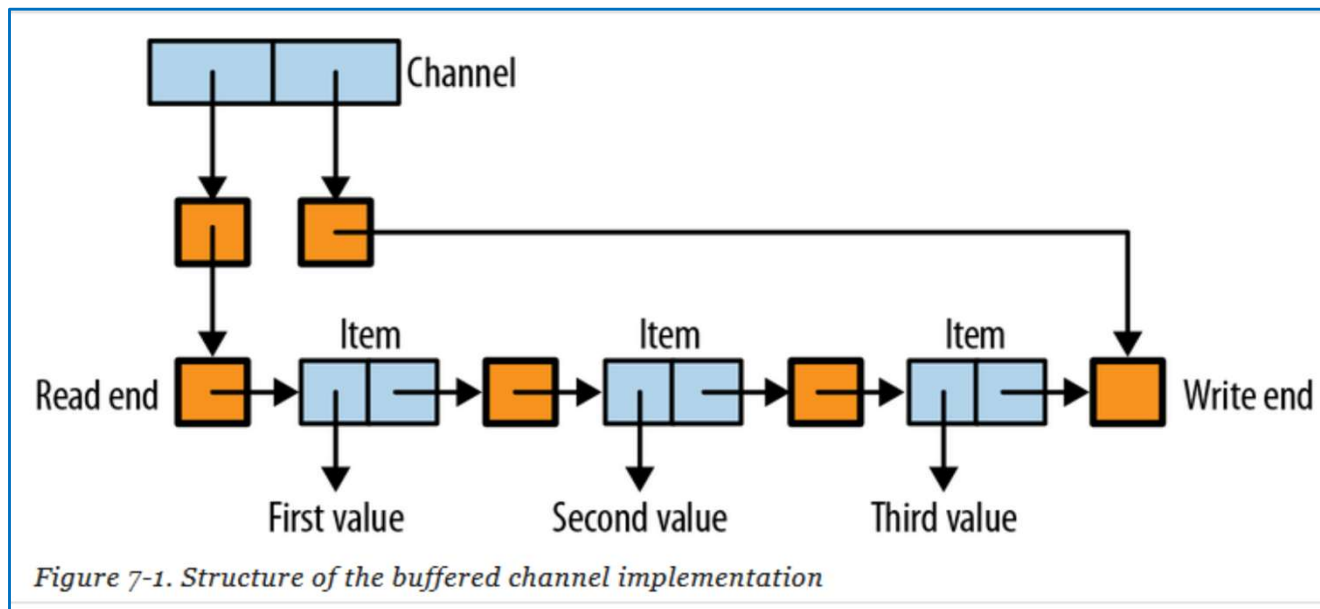
```
main = do
    lib <- newMVar 0
    rwl <- newMyRWLock
    forkIO $ genread 10 rwl lib
    forkIO $ genwrite 5 rwl lib
    getLine
```

```
reader i rwl lib = do
    acquireRead rwl
    c <- readMVar lib
    putStrLn $ (show i) ++ (show c)
    releaseRead rwl

writer i rwl lib = do
    acquireWrite rwl
    putStrLn $ show i
    c <- takeMVar lib
    putMVar lib i
    releaseWrite rwl
```

```
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
```

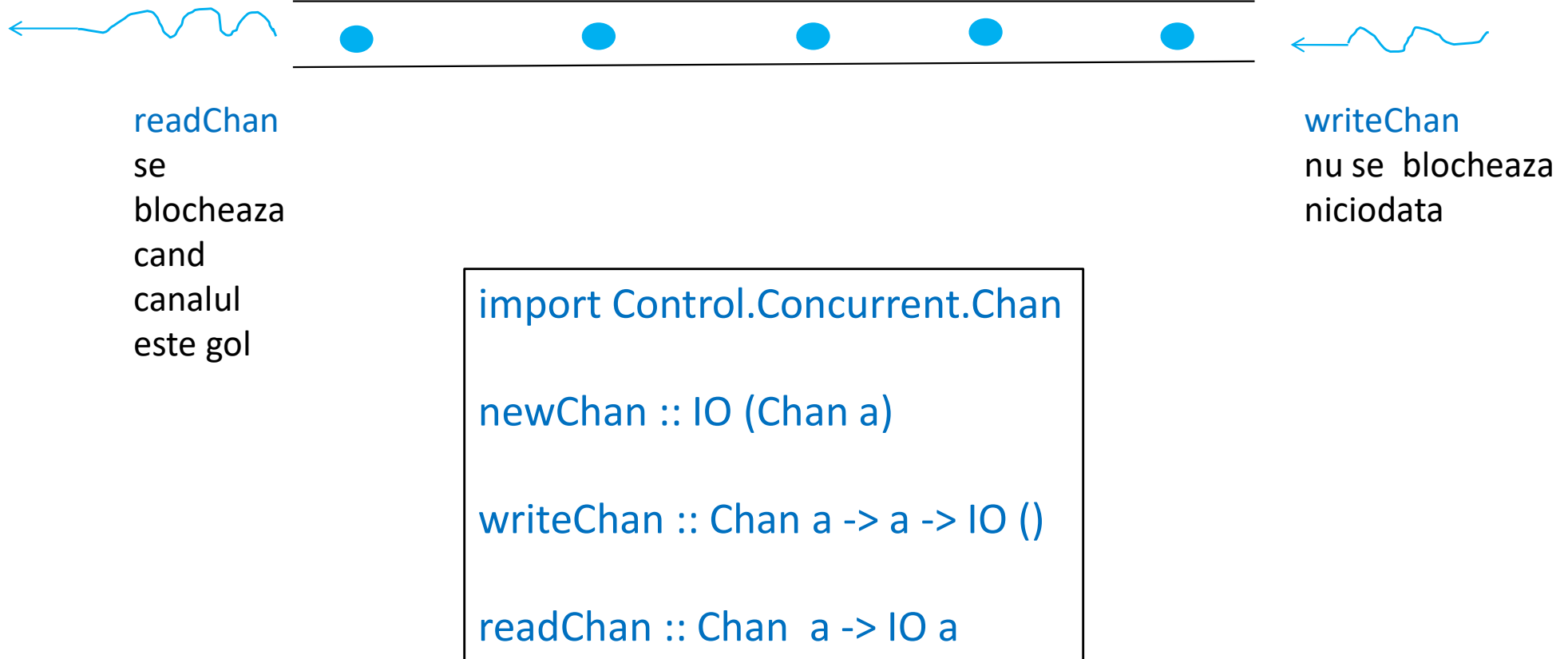
Canale de comunicare formate din variabile MVar



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



➤ Canale de comunicare: canale implementate cu **MVar**



➤ 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
```

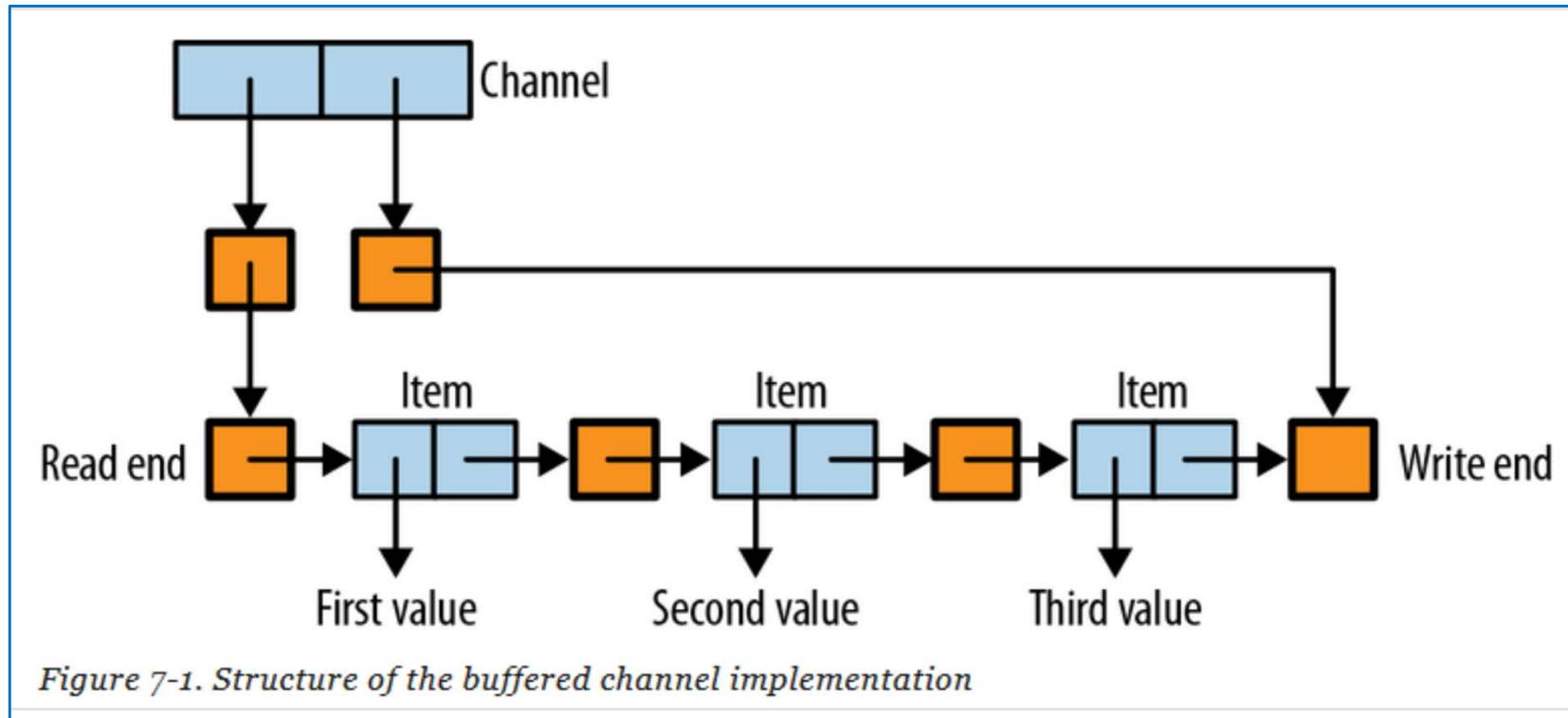
```
move c1 c2 = do
    v1 <- readChan c1
    let ls = words v1
    mapM_ (writeChan c2) ls

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
```



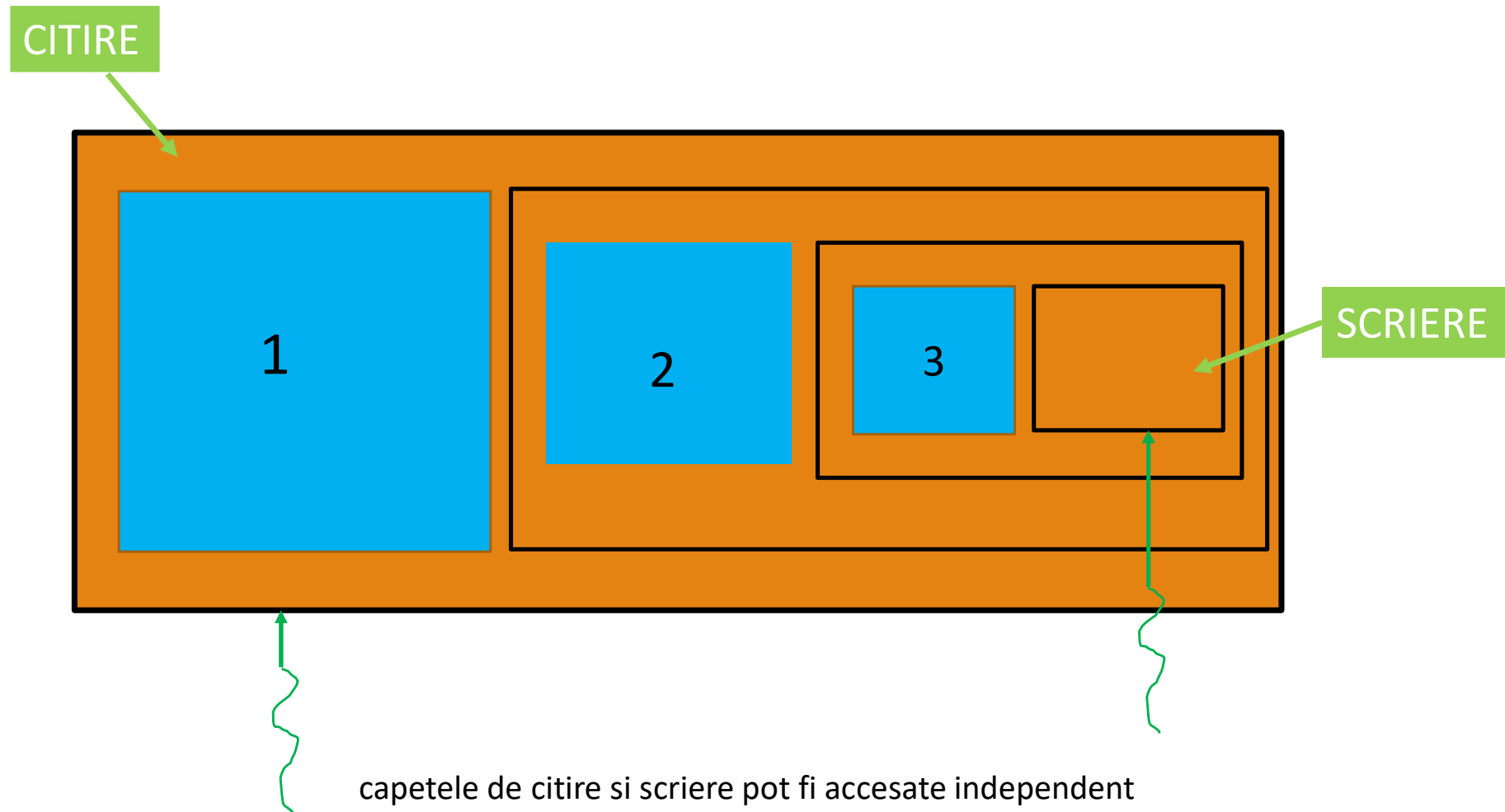
➤ Canale de comunicare formate din variabile MVar



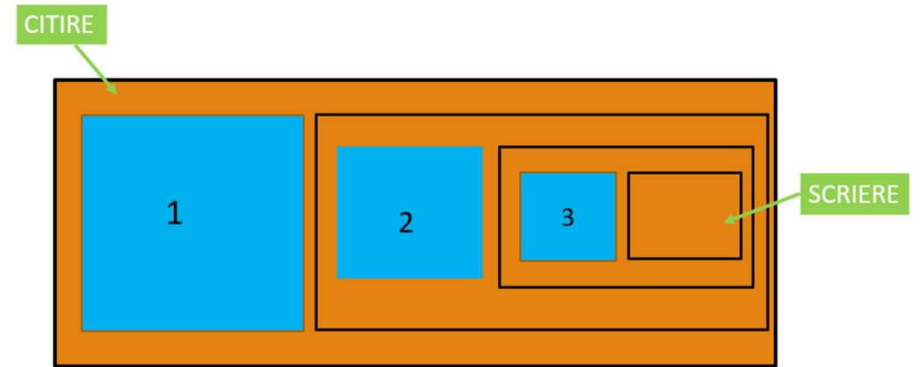
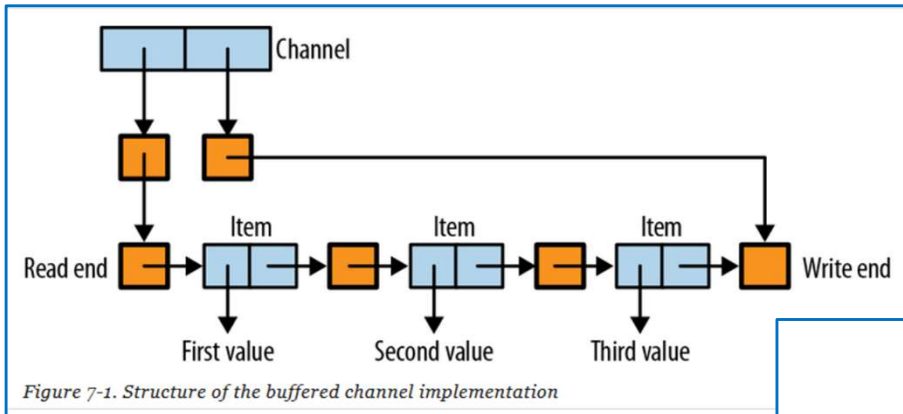
http://chimera.labs.oreilly.com/books/1230000000929/ch07.html#sec_channels



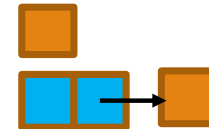
<https://www.haskell.org/hoogle/>



➤ Canale formate din variabile MVar



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

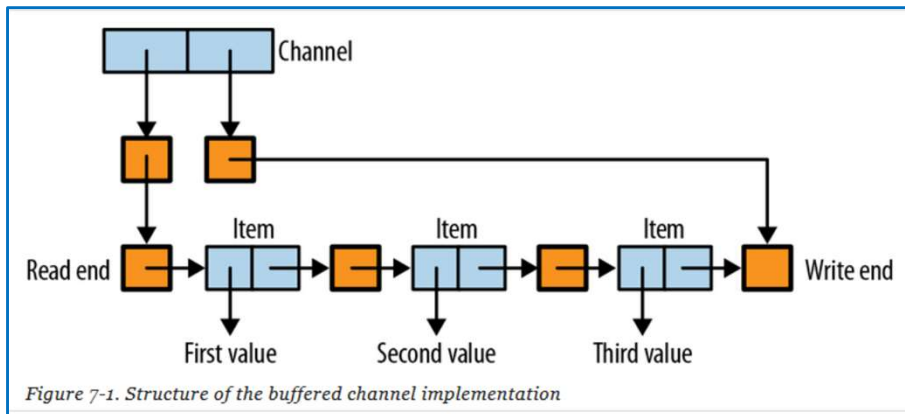


```
data Chan a = Chan (MVar (Stream a)) (MVar (Stream a))
```

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

chan.hs ©2012, Simon Marlow





```
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
```

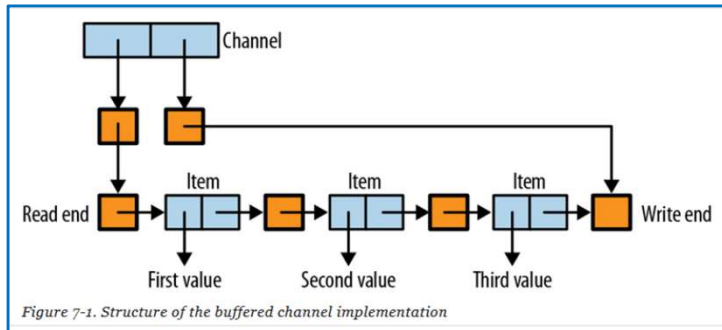
"If multiple threads concurrently call `readChan`, the first one will successfully call `takeMVar` on the read end, but the subsequent threads will all block at this point until the first thread completes the operation and updates the read end. If multiple threads call `writeChan`, a similar thing happens: 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

<https://www.haskell.org/hoogle/>



➤ Implementarea canalelor



```
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

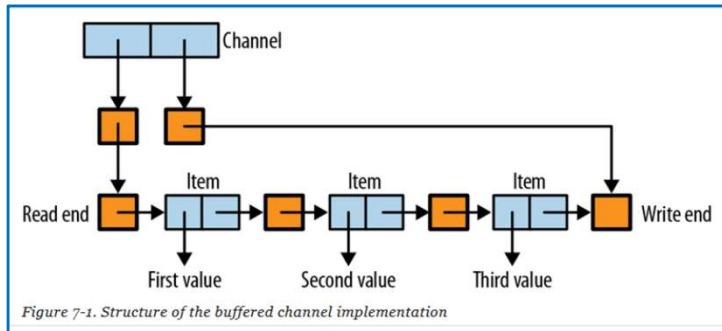
contine variabila in care
se va scrie noul **Item**

http://chimera.labs.oreilly.com/books/12300000000929/ch07.html#sec_channels

<https://www.haskell.org/hoogle/>



➤ Implementarea canalelor



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

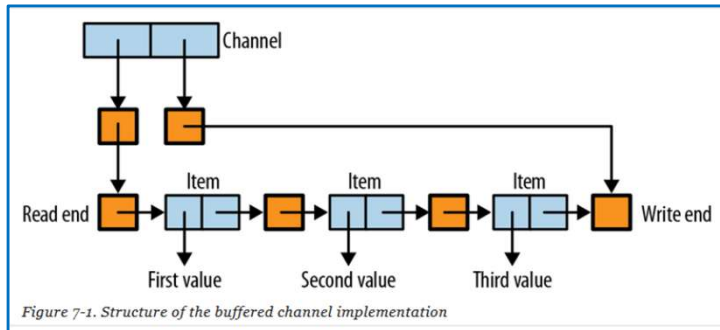
```
readChan :: Chan a -> IO a
readChan (Chan rV wV) = do
    stream <- takeMVar rV
    Item val str <- takeMVar stream
    putMVar rV str
    return val
```

http://chimera.labs.oreilly.com/books/12300000000929/ch07.html#sec_channels

<https://www.haskell.org/hoogle/>



➤ Implementarea canalelor



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

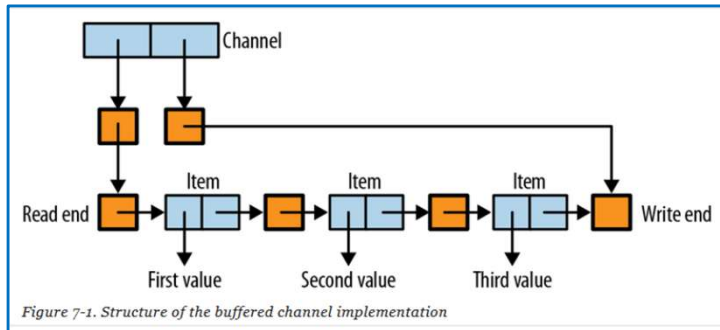
```
writeChan :: Chan a -> a -> IO()
writeChan (Chan rV wV) val = do
    newStream <- newEmptyMVar
    writeEnd <- takeMVar wV
    putMVar writeEnd (Item val newStream)
    putMVar wV newStream
```

http://chimera.labs.oreilly.com/books/12300000000929/ch07.html#sec_channels

<https://www.haskell.org/hoogle/>



➤ Implementarea canalelor



```
newChan :: IO(Chan a)
```

```
newChan = do
```

```
    emptyStream <- newEmptyMVar
    readVar <- newMVar emptyStream
    writeVar <- newMVar emptyStream
    return (Chan readVar writeVar)
```

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

```
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)
```

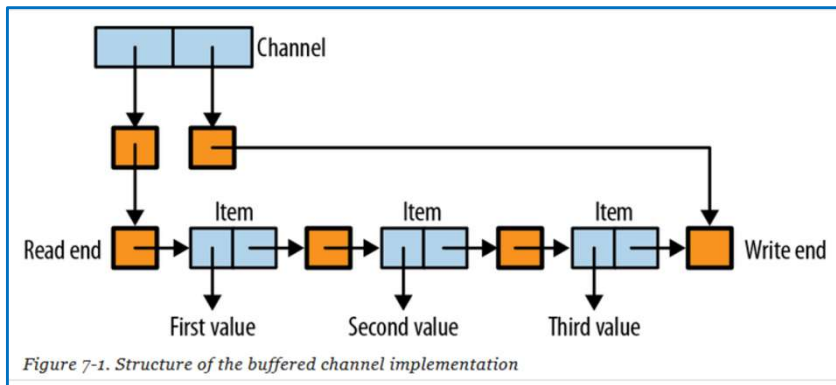
```
    putMVar wV newStream
```

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

<https://www.haskell.org/hoogle/>



➤ Exercițiu: implementarea canalelor multicast



```
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 dintre canale poate fi citit de pe oricare dintre cele doua canale
- citirea de pe un canal **nu** elimina elementul de pe celalalt canal.

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



<https://www.haskell.org/hoogle/>

➤ Exercițiu: implementarea canalelor multicast

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

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

```
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
```

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

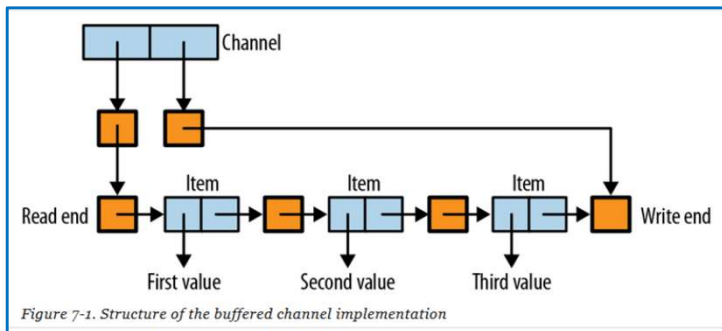
```
Prelude> :l chan2.hs
[1 of 1] Compiling Main
Ok, modules loaded: Main.
*Main> main
'a'
'b'
'b'
```

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

<https://www.haskell.org/hoogle/>



➤ Exercițiu: implementarea canalelor multicast



```
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 _ wV) = do
```

```
    writeEnd <- readMVar wV
```

```
    newReadVar <- newMVar writeEnd
```

```
    return (Chan newReadVar wV)
```

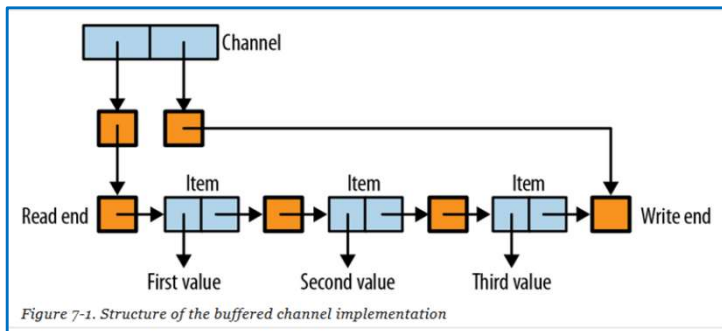
Canalul duplicat are **acelasi cap de scriere**,
dar **un alt cap de citire**

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

<https://www.haskell.org/hoogle/>



➤ Exercițiu: implementarea canalelor multicast



```
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)
```

implementarea reala asigura atomicitate

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

operatia **readChan** trebuie modificata

```
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 folosit in locul lui **takeMVar** deoarece continutul trebuie sa ramana accesibil celui alt canal.

http://chimera.labs.oreilly.com/books/12300000000929/ch07.html#sec_channels

<https://www.haskell.org/hoogle/>



➤ Comunicare sincrona

Un thread – **reader**- citeste un fisier text linie cu linie

Liniile sunt trimise, pe rand,
unui al doilea thread – **writer**- care le afiseaza
in ordinea trimisa si le numara.

La sfarsit, thread-ul writer trimite thread-ului reader
numarul de linii si acesta il afiseaza,

http://rosettacode.org/wiki/Synchronous_concurrency

```
import Control.Concurrent
import Control.Concurrent.MVar

main = do
    lineVar <- newEmptyMVar
    countVar <- newEmptyMVar
    forkIO $ writer lineVar countVar
    reader lineVar countVar
```



➤ Comunicare sincrona

```
import Control.Concurrent
import Control.Concurrent.MVar

main = do
    lineVar <- newEmptyMVar
    countVar <- newEmptyMVar
    forkIO $ writer lineVar countVar
    reader lineVar countVar
```

```
-- main thread
reader lineVar countVar = do
    ls <- fmap lines (readFile "input.txt")
    mapM ((putMVar lineVar) . Just) ls
    putMVar lineVar Nothing
    n <- takeMVar countVar
    print n
```

```
readFile :: FilePath -> IO String
lines :: String -> [String]
fmap :: Functor f => (a -> b) -> f a -> f b
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
```

http://rosettacode.org/wiki/Synchronous_concurrency

<https://www.haskell.org/hoogle/>



➤ Comunicare sincrona

```
import Control.Concurrent
import Control.Concurrent.MVar

main = do
    lineVar <- newEmptyMVar
    countVar <- newEmptyMVar
    forkIO $ writer lineVar countVar
    reader lineVar countVar
```

```
readFile :: FilePath -> IO String
lines :: String -> [String]
fmap :: Functor f => (a -> b) -> f a -> f b
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
```

```
reader lineVar countVar = do
    ls <- fmap lines (readFile "input.txt")
    mapM ((putMVar lineVar) . Just) ls
    putMVar lineVar Nothing
    n <- takeMVar countVar
    print n
```

```
writer lineVar countVar = loop 0
    where
        loop n = do
            l <- takeMVar lineVar
            case l of
                Just x -> do putStrLn x
                           loop (n+1)
                Nothing -> putMVar countVar n
```

http://rosettacode.org/wiki/Synchronous_concurrency

<https://www.haskell.org/hoogle/>



➤ Comunicare sincrona

```
import GetURL
import Data.ByteString as B

action x = do
    r <- getURL x
    print (B.length r)
```

Fisierul "inputurl.txt" contine adrese web,
iar **action** **descarca** paginile respective

```
reader lineVar countVar = do
    ls <- fmap lines (readFile "inputurl.txt")
    mapM ((putMVar lineVar) . Just) ls
    putMVar lineVar Nothing
    n <- takeMVar countVar
    print n

writer lineVar countVar = loop 0
    where
        loop n = do
            l <- takeMVar lineVar
            case l of
                Just x -> do
                    action x
                    loop (n+1)
                Nothing -> putMVar countVar n
```

<https://hackage.haskell.org/package/parconc-examples-0.1/src/GetURL.hs>

<https://www.haskell.org/hoogle/>



“**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

secvential



concurent



```
import Data.ByteString as B
import GetURL
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

```
var <- newEmptyMVar
forkIO $ do
    r <- getURL "http://... "
    putMVar var r
takeMVar var
```

```
a <- async (getURL "http://... ")
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 devine goala dupa citire (multiple-wakeup)
deci mai multe apeluri **wait** pot fi facute pentru
aceeasi operatie asincrona



➤ Comunicare asincrona

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

```
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
```

```
a <- async action
r <- wait a
```

```
import Data.ByteString as B
import GetURL

main = do
    a1 <- async (getURL "http://..." )
    a2 <- async (getURL "http://..." )
    r1 <- wait a1
    r2 <- wait a2
    print (B.length r1, B.length r2)
```



➤ Comunicare asincrona

```
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
```

```
a <- async action
r <- wait a
```

```
import Data.ByteString as B
import GetURL
import TimeIt
import Text.Printf

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

main = do
  a1 <- async (timeDownload "http://...")
  a2 <- async (timeDownload "http://...")
  wait a1
  wait a2
```



➤ Comunicare asincrona

```
import Data.ByteString as B
import GetURL
import Time
import Text.Printf

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

main = do
    a1 <- async (timeDownload "http://...")
    a2 <- async (timeDownload "http://...")
    wait a1
    wait a2
```

vor fi afisate statistici privind timpul de executie

```
*Main> :set +s
*Main> main
downloaded: http://old.uefiscdi.ro/ (74599 bytes, 3.96)
downloaded: http://old.uefiscdi.ro/ (74599 bytes, 4.01)
(4.01 secs, 52,080 bytes)
```



➤ 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
```

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

```
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
```

geturl3.hs ©2012, Simon Marlow



➤ Async - comunicare asincrona (folosind MVar)

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

Cum rezolvam aceasta problema?

