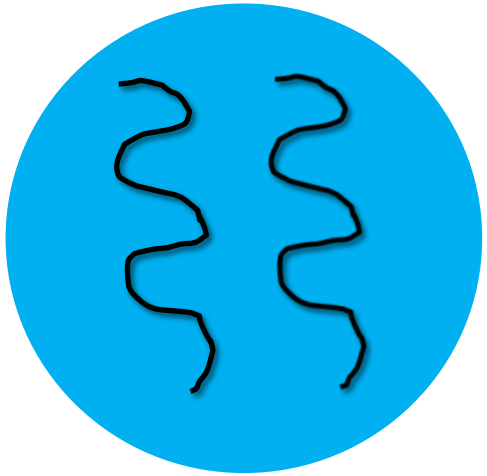
IMPLEMENTAREA CONCURENTEI IN LIMBAJE DE PROGRAMARE

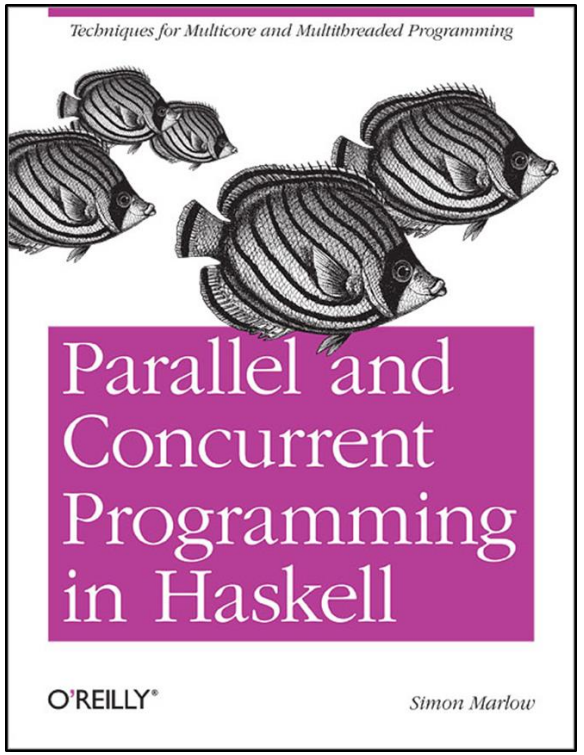
Concurenta

Threaduri

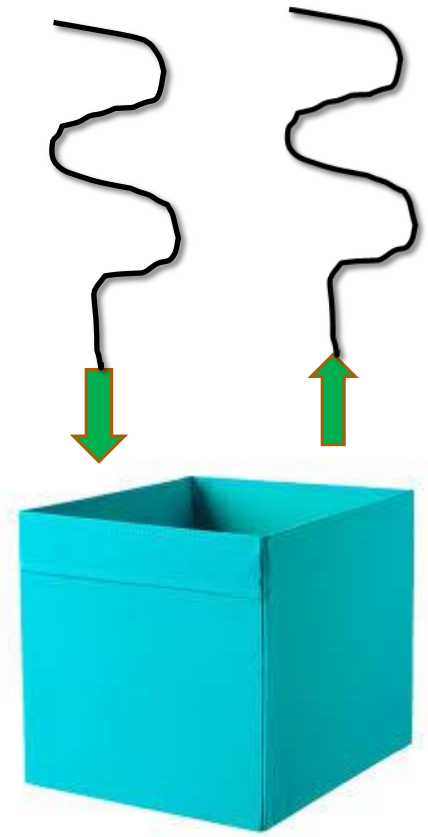
Memorie Partajata

Ioana Leustean





Part II. Concurrent Haskell
Cap.7 & 8



forkIO :: IO () -> IO ThreadId

MVar
mutable variable

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

➢ Comunicarea folosind MVar se face in monada IO

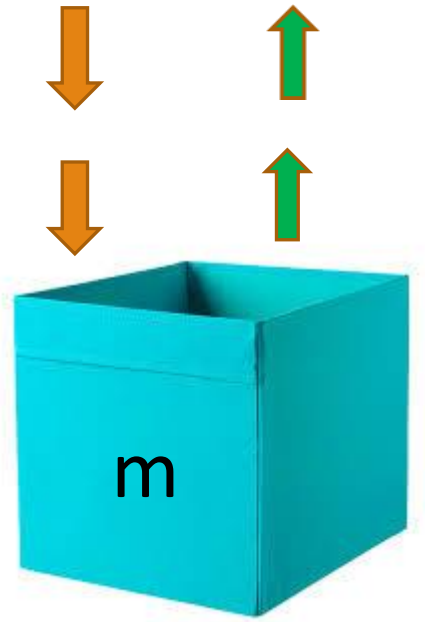
```
data MVar a


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: doua thread-uri incrementeaza acelasi contor

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

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

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

➢ Sincronizare: doua thread-uri incrementeaza acelasi contor

```
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
```

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

threadDelay nr
suspenda thread-ul pt.nr microsecunde

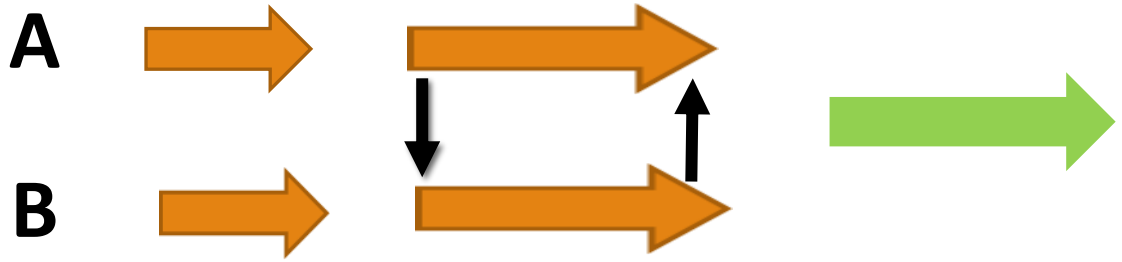
```
add2 m ms2   = do
        replicateM_ 1000 $ do
                s<- takeMVar m
                putMVar mv (s + 1)
        putMVar ms2 "ok"
```

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

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




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

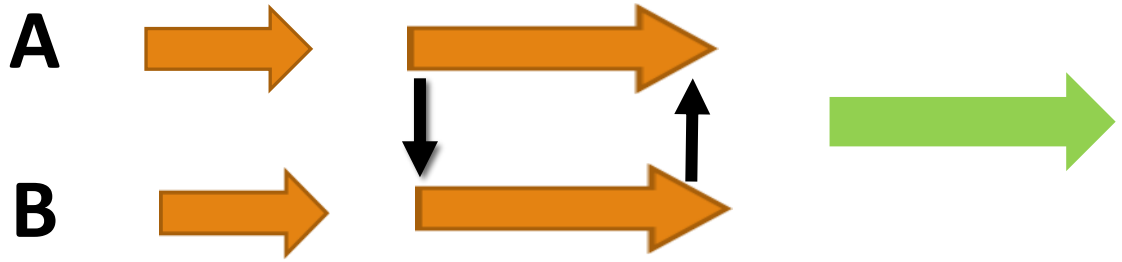
```
threadB a b  = do
                    x <- takeMVar a
                    let y =  x ++ x
                    putMVar b y
```

Este corect? Ce problema de sincronizare poate sa apara?

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
```

- Producer-Consumer problem
  MVar ca monitor


```
import Control.Concurrent
import Control.Monad


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

- 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)
```

➢ Semafor cu cantitate (quantity semaphore)

```
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
```

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

➢ Exemplu: **QSem**

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

```
import Control.Concurrent
import Control.Monad


main :: IO ()
main = do
        q <- newQSem 3
        let workers = 5
        mapM_ (forkIO . worker q m) [1..workers]
```

```
mapM_ :: (Foldable t, Monad m) => (a -> m b) -> t a -> m ()
```

➢ Exemplu: **QSem**

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

```
import Control.Concurrent
import Control.Monad


main :: IO ()
main = do
        q <- newQSem 3
        let workers = 5
        mapM_ (forkIO . worker q m) [1..workers]
```

q este semaforul care controleaza resursele

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

http://rosettacode.org/wiki/Metered_concurrency

➢ Exemplu: **QSem**

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

```
import Control.Concurrent
import Control.Monad

main :: IO ()
main = do
        q <- newQSem 3
        let workers = 5
        mapM_ (forkIO . worker q m) [1..workers]
```

q este semaforul care controleaza resursele

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

```
Ok, one module loaded.
*Main> main
WoWWo*Main> orrekkree rr1   23h  ahhsaa ssa  caaqccuqqiuuriierrdee ddt  httehh eel  ollcookcc.kk
..

WoWWrWWookoorrerrkkrkkee eerr1rr     23h54  a  hhshhaa aassrss  e  rrlaaeeeccllaqqeesuuaaeiissdrree eeddtdd  h
 ttetthh hheelee  o  llcllookoocc.cckk
kk....
```

http://rosettacode.org/wiki/Metered_concurrency

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

➢ Exemplu: **QSem**

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

```
import Control.Concurrent
import Control.Monad

main :: IO ()
main = do
        q <- newQSem 3
        let workers = 5
        mapM_ (forkIO . worker q m) [1..workers]
```

q este semaforul care controleaza resursele

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

```
Ok, one module loaded.
*Main> main
WoWWo*Main> orrekkree rr1   23h  ahhsaa ssa  caaqccuqqiuuriierrdee ddt  httehh eel  ollcookcc.kk
..

WoWWrWWookoorrerrkkrkkee eerr1rr     23h54  a  hhshhaa aassrss  e  rrlaaeeeccllaqqeesuuaaeiissdrree eeddtdd  h
 ttetthh hheelee  o  llcllookoocc.cckk
kk....
```

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

http://rosettacode.org/wiki/Metered_concurrency

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

➢ Exemplu: **QSem**

O multime de taskuri acceseaza simultan o resursa reprezentata printr-un **QSem**;
pentru a se executa, fiecare task trebuie sa acceseaze 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

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

```
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

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

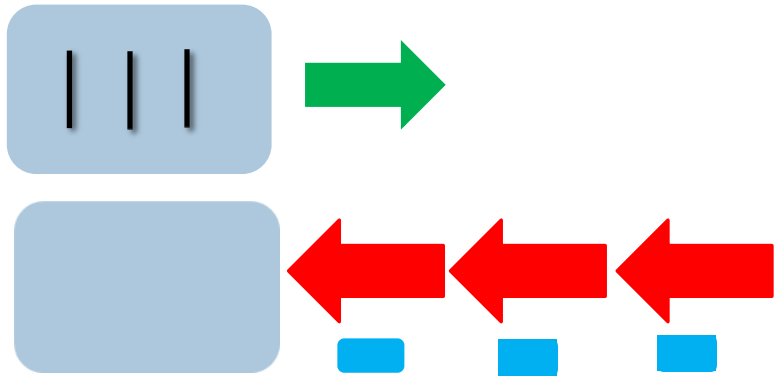
Implementarea QSem

```
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*

```
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

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

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

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

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

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

Eliberarea resursei

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

- fiecare thread elibereaza variabila proprie a unui thread in asteptare

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

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

```
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

```
waitQSem ::  QSem -> IO()
waitQSem qsem = do
            (avail,blks) <- takeMVar qsem
            if avail > 0
                then    putMVar qsem (avail-1, [])
                else
                    do
                      blk <- newEmptyMVar
                      putMVar qsem (0,blk:blks)
                      takeMVar blk
```

– threadul e blocat pe variabila proprie

Eliberarea resursei

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

fiecare thread elibereaza variabila proprie a unui thread in asteptare

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

➢ 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)
```

➢ 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}
```

```
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)
```

➢ 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

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


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

lib este resursa partajata
rwl este lacatul care sincronizeaza accesul la resursa

- 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
                    putStrLn $  (show i) ++ (show c)
                    releaseRead rwl

writer i rwl lib = do
                     aquireWrite 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
                   aquireRead rwl
                   c <- readMVar lib
                   putStrLn $  (show i) ++ (show c)
                   releaseRead rwl

writer i rwl lib = do
                    aquireWrite 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
```

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

➢ 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
```

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

➢ 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
```

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

➢ Canale de comunicare formate din variabile MVar

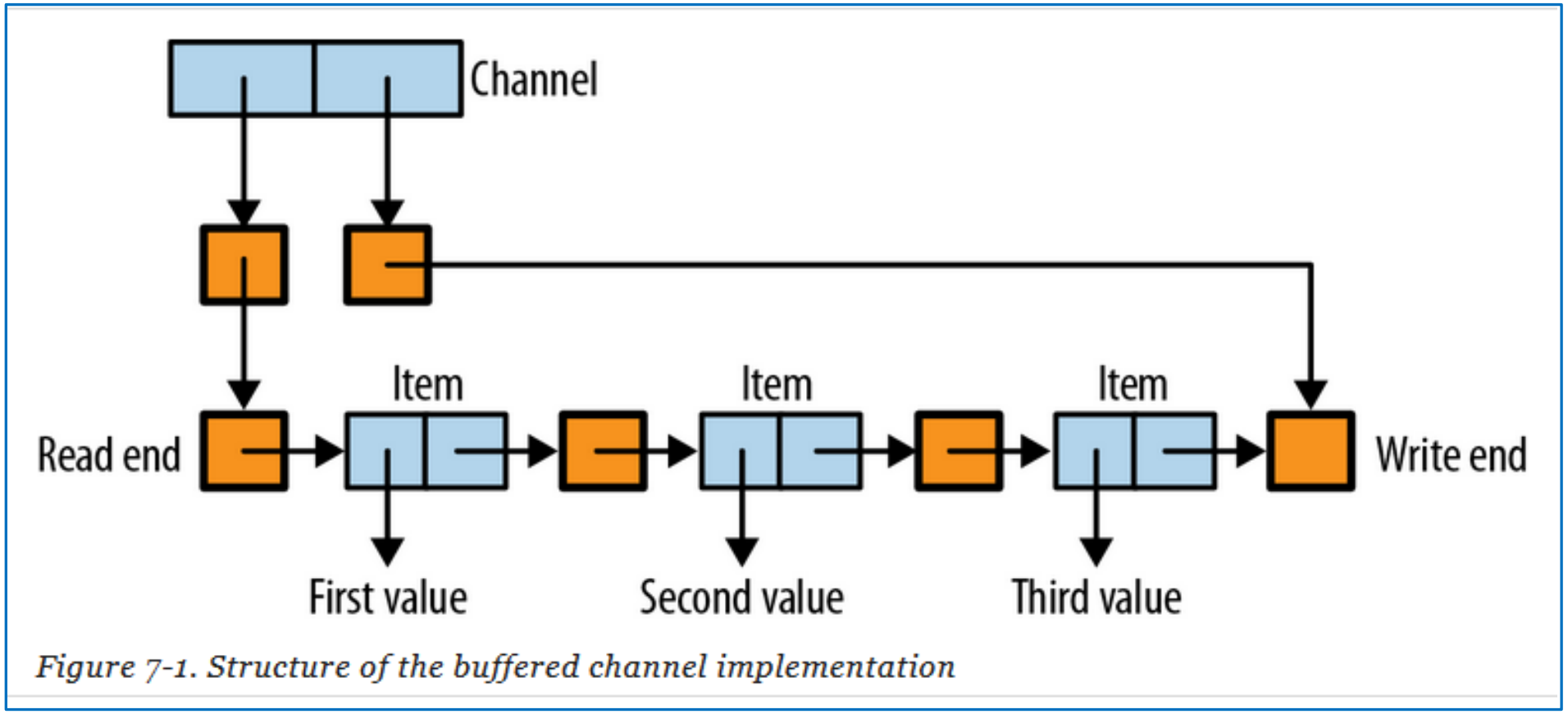


Figure 7-1. Structure of the buffered channel implementation

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

➢ Canale formate din variabile MVar

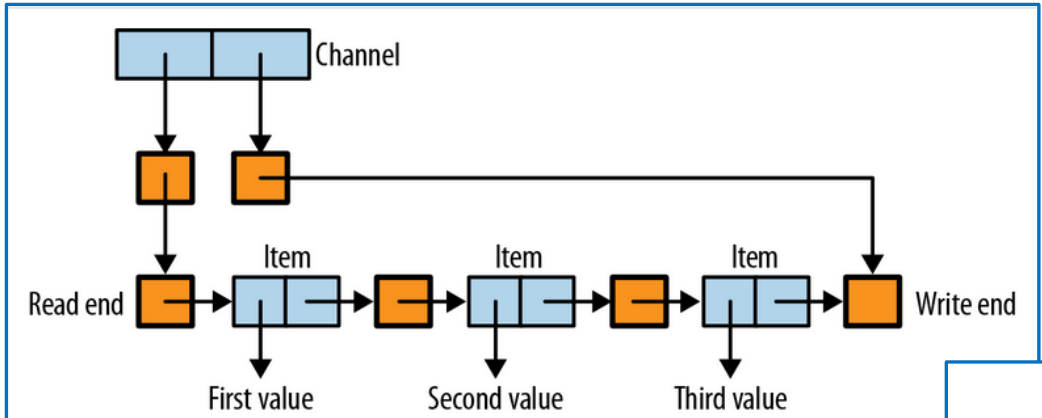
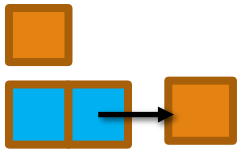


Figure 7-1. Structure of the buffered channel implementation

```
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

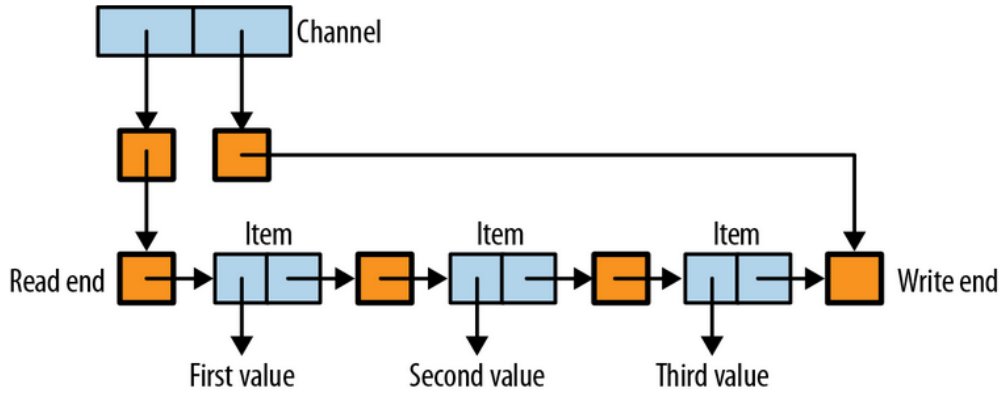


Figure 7-1. Structure of the buffered channel implementation

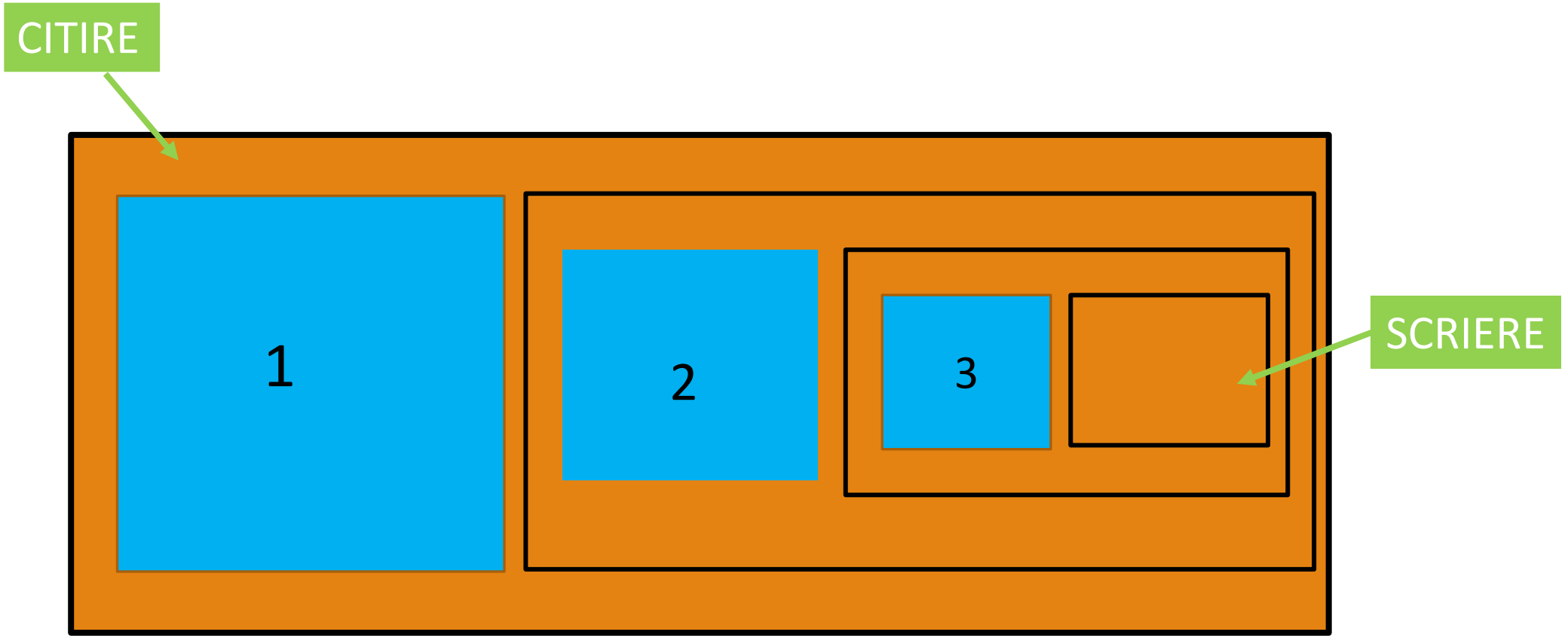
```
type Stream a = MVar (Item a)
data Item a   = Item a (Stream a)
data Chan a = Chan (MVar (Stream a)) (MVar (Stream a))

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

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


CITIRE
1
2
3
SCRIERE

Implementarea canalelor

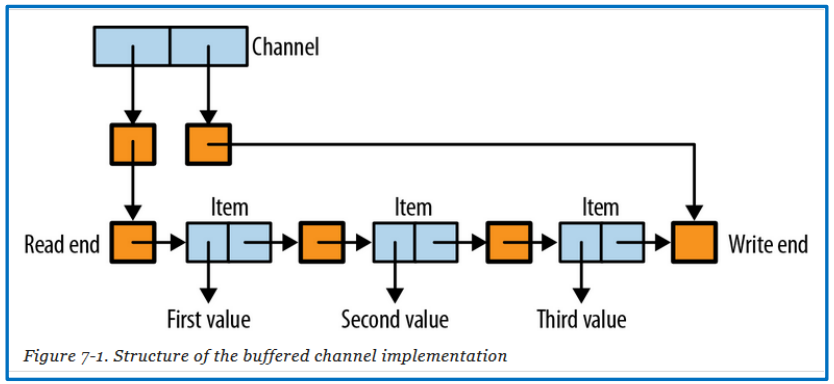


Figure 7-1. Structure of the buffered channel implementation

```
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

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

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

➢ Implementarea canalelor

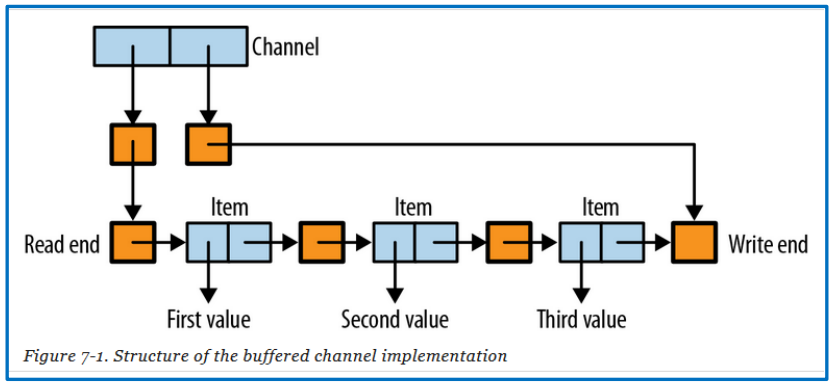


Figure 7-1. Structure of the buffered channel implementation

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

```
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/1230000000929/ch07.html#sec_channels

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

➢ Implementarea canalelor

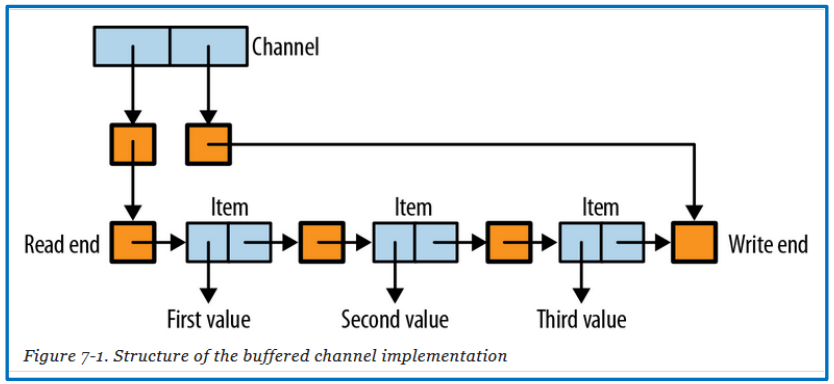


Figure 7-1. Structure of the buffered channel implementation

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

```
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/

➢ Implementarea canalelor

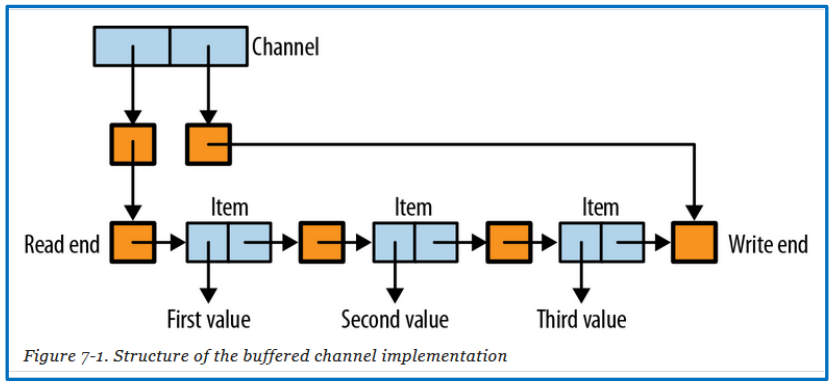


Figure 7-1. Structure of the buffered channel implementation

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

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

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

```
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/

➢ Exercitiu: implementarea canalelor multicast

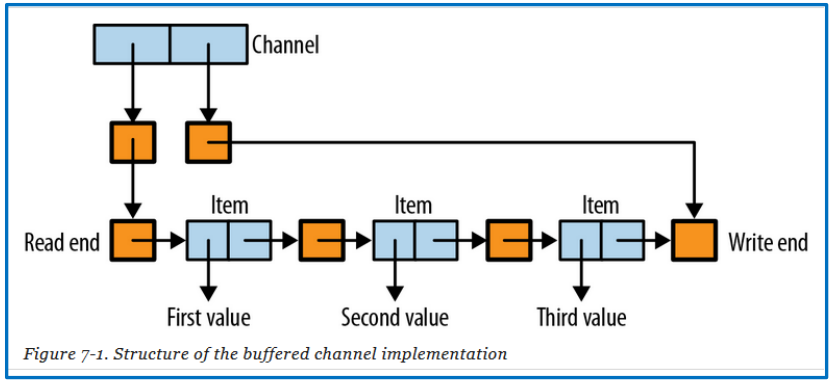


Figure 7-1. Structure of the buffered channel implementation

```
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

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

➢ Exercitiu: implementarea canalelor multicast

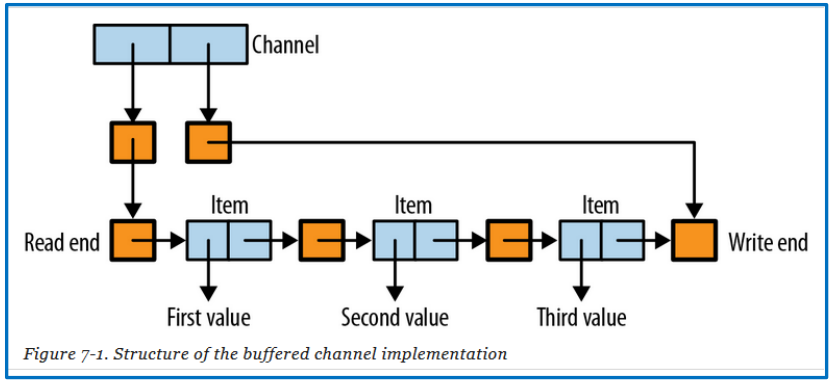


Figure 7-1. Structure of the buffered channel implementation

```
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)
```

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/

➢ Exercitiu: implementarea canalelor multicast

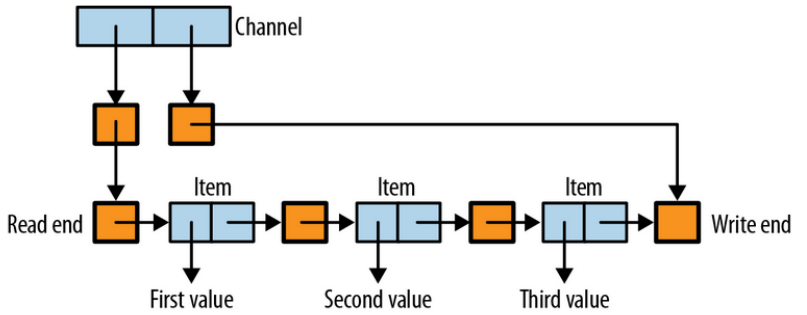


Figure 7-1. Structure of the buffered channel implementation

```
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
```

operatia **readChan** trebuie modificata

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

readMVar este folosit in locul lui **takeMVar** deoarece continutul trebuie sa ramana accesibil celuilalt canal.

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

➢ 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 putLine lineVar countVar
```

➢ Comunicare sincrona

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

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

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

http://rosettacode.org/wiki/Synchronous_concurrency

- Comunicare sincrona

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

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

```
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

Thread 1

Thread 2

```
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

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

- Comunicare asincrona

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

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

```
a <- async (getURL "http://... " )
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

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

- 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
```

```
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)
```

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

➢ 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 TimeIt
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
```

```
*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)
```

vor fi afisate statistici privind timpul de executie

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