

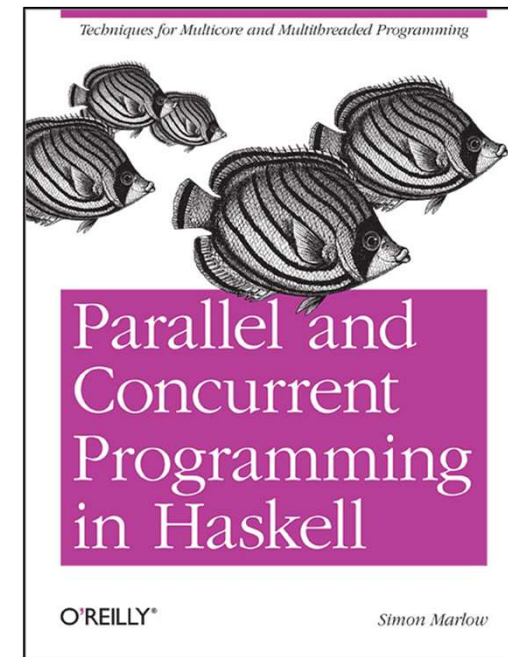
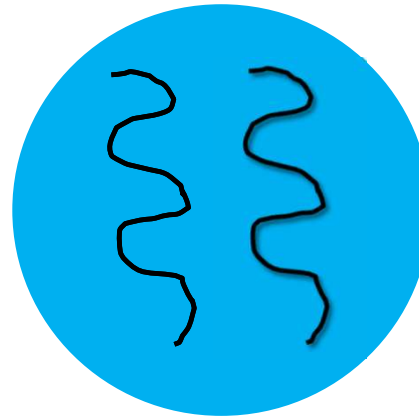
IMPLEMENTAREA CONCURENTEI IN LIMBAJE DE PROGRAMARE

Concurenta

Threaduri

Memorie Partajata

Ioana Leustean

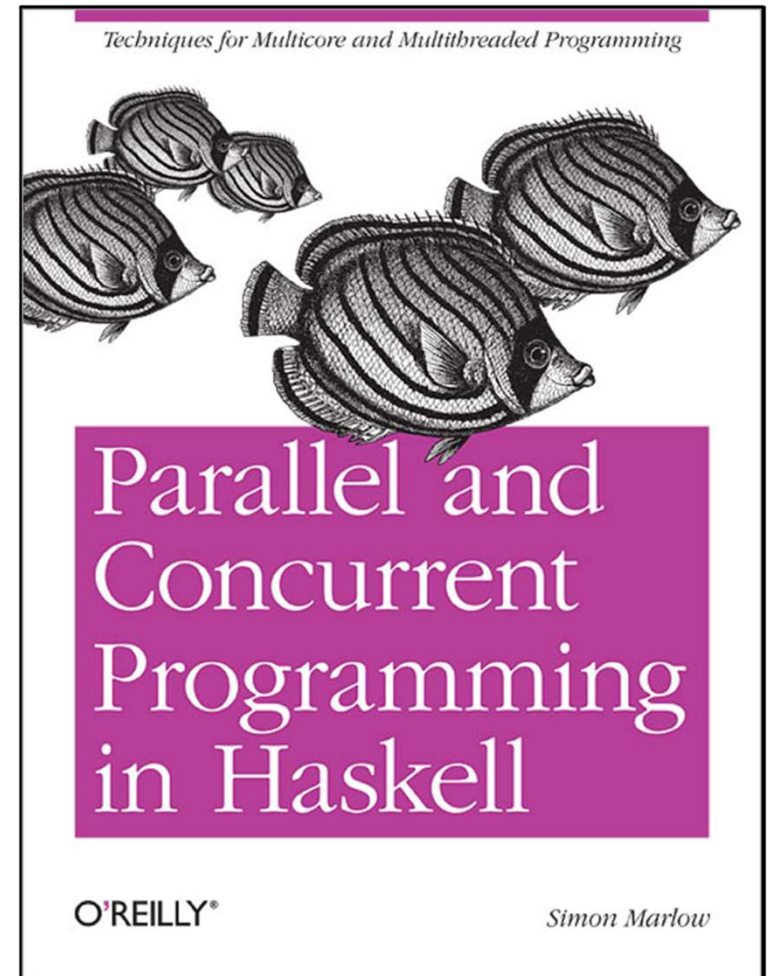


Part II. Concurrent Haskell
Cap.7 & 8

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

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

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



➤ Thread-urile in Haskell:

Thread-urile au efecte si interactioneaza cu lumea exterioara.

Programarea concurenta in Haskell are loc in [monada IO](#).

La rulare, efectele thread-urilor sunt intercalate nedeterminist.

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

Implementarea threadurilor asigura verificarea anumitor conditii de corectitudine.



➤ Crearea thread-urilor

`forkIO :: IO () -> IO ThreadId`

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

```
import Control.Concurrent
import Control.Monad
```

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

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

[replicateM](#)



```
forkIO :: IO () -> IO ThreadId
```

```
import Control.Concurrent
import Control.Monad

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

[illegible]

La rulari diferite se pot obtine rezultate diferite!



```
forkIO :: IO () -> IO ThreadId
```

```
import Control.Concurrent
import Control.Monad
```

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

Daca fisierul se numeste *thread.hs*
atunci el poate fi compilat si executat:

ghc thread.hs -threaded thread

```
C:\Users\igleu\Documents\DIR\ICLP\ICLP2023\haskell12023>ghc thread.hs -threaded
Loaded package environment from C:\Users\igleu\AppData\Roaming\ghc\x86_64-mingw32-8.10.7\environments\default
```

[illegible]

La rulari diferite se pot obtine rezultate diferite!

[illegible]

`forkIO :: IO () -> IO ThreadId`

`myThreadId :: IO ThreadId`

```
import Control.Concurrent
import Control.Monad
```

```
main = do
    forkIO (replicateM_ 100 (myThreadId >=> print)) -- child thread
    replicateM_ 100 (myThreadId >=> print) -- main thread
```

```
PS C:\Users\igleu\Documents\DIR\ICLP22\Curs-2022\Haskell22\pgh\haskell2022> ./threadID
ThreadId 3
ThreadId 4
ThreadId 3
ThreadId 4
ThreadId 3
ThreadId 4
ThreadId 3
ThreadId 4
ThreadId 3
ThreadId 4
ThreadId 3
ThreadId 4
ThreadId 3
ThreadId 4
```



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

S. Marlow, PCPH

"`forkIO` is asymmetrical: when a process executes a `forkIO` it spawns a child process that executes concurrently with the continued execution of the parent"

SL Peyton Jones, A Gordon, S Finne, Concurrent Haskell

"GHC's runtime system treats the program's original thread of control differently from other threads.

When this thread finishes executing, the runtime system considers the program as a whole to have completed.

If any other threads are executing at the time, they are terminated."

B. O'Sullivan, D. Stewart, J. Goerzen, Real World Haskell



➤ Interleaving

```
import Control.Concurrent
import Control.Monad

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

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

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

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



➤ Executie secventiala vs executie concurenta

```
fib 0 =1
fib 1 =2
fib n = fib (n-1) + fib (n-2)

act n = do
    let x = (fib n)
    putStrLn ("Fib " ++ (show n) ++ " is " ++ (show x))

act4 = do
    act 10
    act 20
    act 30
    act 35
    getLine
```

```
main = do
    forkIO $ act 10
    forkIO $ act 20
    forkIO $ act 30
    forkIO $ act 35
    getLine
```



➤ Executie secventiala vs executie concurenta

```
act4 = do
    act 10
    act 20
    act 30
    act 35
    getLine
```

```
main = do
    forkIO $ act 10
    forkIO $ act 20
    forkIO $ act 30
    forkIO $ act 35
    getLine
```

```
Prelude> :set +s
Prelude> :l test.hs
[1 of 1] Compiling Main          ( test.hs, interpreted )
Ok, one module loaded.
(0.09 secs,)
*Main> act4
Fib 10 is 144
Fib 20 is 17711
Fib 30 is 2178309
Fib 35 is 24157817

""
(115.38 secs, 11,248,521,824 bytes)
*Main> main
FiFFFbiii bbb1 0233 005i siii sss1 4147
711
2178309
24157817

""
(114.36 secs, 57,352 bytes)
```



➤ Compilare cu -threaded

```
act4 = do
    act 10
    act 20
    act 30
    act 35
    getLine
```

```
main = do
    forkIO $ act 10
    forkIO $ act 20
    forkIO $ act 30
    forkIO $ act 35
    getLine
```

```
*Main> :! ghc --make -threaded test.hs
```

```
> .\test +RTS -N4 -s
Fib 10 is 144
Fib 20 is 17711
Fib 30 is 2178309
Fib 35 is 24157817
```

```
INIT  time  0.000s ( 0.001s elapsed)
MUT   time  4.672s ( 7.394s elapsed)
GC    time  0.156s ( 0.120s elapsed)
EXIT  time  0.000s ( 0.001s elapsed)
Total time  4.828s ( 7.515s elapsed)
```

programul este compilat cu optiunea
-threaded



-threaded

"Link the program with the “threaded” version of the runtime system.

The threaded runtime system [...] enables the -N <x> RTS option to be used, which allows threads to run in parallel on a multiprocessor or multicore machine.

Note that you do not need -threaded in order to use concurrency; the single-threaded runtime supports concurrency between Haskell threads just fine."

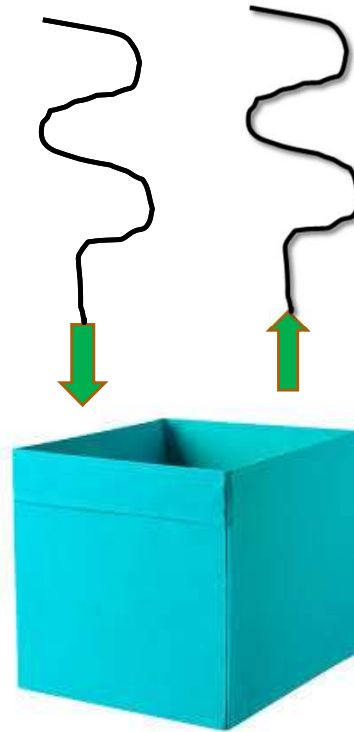
https://downloads.haskell.org/~ghc/9.10.1/docs/users_guide/phases.html

"**Concurrent Haskell** is a collective name for the facilities that Haskell provides for programming with multiple threads of control. Unlike parallel programming, where the goal is to make a program run faster by using more CPUs, the goal in concurrent programming is usually to write a program with multiple interactions."

S. Marlow, PCHP



➤ Comunicarea thread-urilor

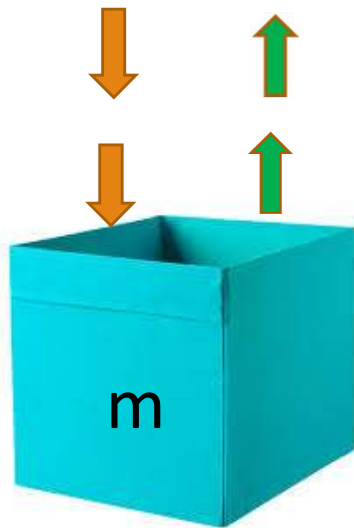


MVar
mutable variable

➤ Comunicarea folosind **MVar** se face in **monada IO**

data MVar a

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



m :: MVar a

poate fi vazuta ca:

- un semafor binar
- un monitor cu o variabila



➤ takeMVar

- `takeMVar` este o operatie care blocheaza thread-urile
- `takeMVar` este *single-wakeup*:
daca variabila MVar este goala, toate thread-urile care vor sa execute `takeMVar` sunt blocate; cand variabila devine plina, un singur thread este trezit si acesta va executa `takeMVar`
- daca mai multe thread-uri sunt blocate pe acelasi `MVar`, ele vor fi trezite in ordinea FIFO

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



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


```
import Control.Concurrent
```

```
main = do
```

```
    m <- newEmptyMVar
```

```
    forkIO $ do
```

```
        putMVar m 'x'
```

```
        putMVar m 'y'
```

```
    x <- takeMVar m
```

```
    print x
```

```
    x <- takeMVar m
```

```
    print x
```

```
newEmptyMVar :: IO (MVar a)
```

```
putMVar :: MVar a -> a -> IO()
```

```
takeMVar :: MVar a -> IO a
```

```
*Main> main
```

```
'x'
```

```
'y'
```



```
import Control.Concurrent
```

```
main = do
```

```
    m <- newEmptyMVar
```

```
    takeMVar m
```

```
*Main> main
```

```
*** Exception: thread blocked indefinitely in an MVar operation
```



➤ takeMVar vs readMVar

readMVar

Citeste atomic continutul unui `MVar`.

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

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

Implementarea veche

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

Implementarea actuala garanteaza ca `readMVar` este o operatie atomica.

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

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



➤ Comunicarea folosind **MVar** se face in **monada IO**

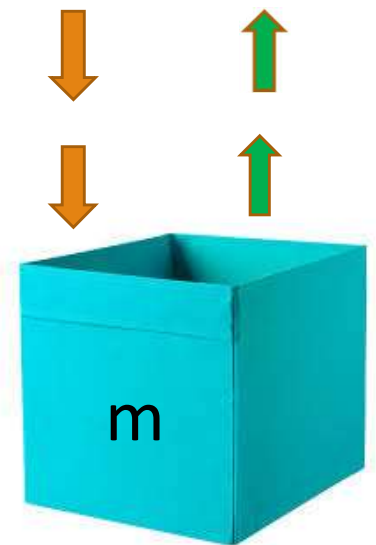
data MVar a

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 si **goleste** 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



➤ MVar ca semafor binar

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

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

`forever` repeta o actiune monadica
de un numar infinit de ori



➤ MVar ca semafor binar

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

act1 m = do

```
    acquireLock m  
    print "I have the lock"  
    releaseLock m
```

act2 m = do

```
    acquireLock m  
    print "Now I am have the lock"  
    releaseLock m
```

main = do

```
    m <- newLock  
    forkIO $ act1 m  
    forkIO $ act2 m  
    getLine
```



➤ MVar ca semafor binar

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

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

```
"Now I have the lock"  
"I have the lock"  
"Now I have the lock"  
"I have the lock"  
"Now I have the lock"  
"I have the lock"  
"Now I have the lock"  
"I have the lock"  
"Now I have the lock"  
"I have the lock"  
"Now I have the lock"  
"I have the lock"  
"Now I have the lock"  
"I have the lock"  
"Now I have the lock"  
"I have the lock"  
"Now I have the lock"  
"I have the lock"  
"Now I have the lock"  
"I have the lock"  
"Now I have the lock"  
"I have the lock"  
"Now I have the lock"  
"I have the lock"  
"Now I have the lock"  
"I have the lock"  
"Now I have the lock"  
"I have the lock"
```



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

```
add m = replicateM_ 1000 $ do
```

```
    x <- takeMVar m  
    putMVar m (x + 1)
```

```
main = do
```

```
    m <- newMVar 0  
    forkIO (add m )  
    forkIO (add m )  
    x <- takeMVar m  
    print x
```



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

```
add m = replicateM_ 1000 $ do
```

```
    x <- takeMVar m  
    putMVar m (x + 1)
```

```
main = do
```

```
    m <- newMVar 0  
    forkIO (add m )  
    forkIO (add m )  
    x <- takeMVar m  
    print x
```

```
*Main> :l cont.hs  
[1 of 1] Compiling Main  
Ok, one module loaded.  
*Main> main  
0  
*Main> main  
0
```



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

```
add m = replicateM_ 1000 $ do
```

```
  x <- takeMVar m  
  putMVar m (x + 1)
```

```
main = do
```

```
  m <- newMVar 0  
  forkIO (add m )  
  forkIO (add m )  
  x <- takeMVar m  
  print x
```

```
*Main> :l cont.hs  
[1 of 1] Compiling Main  
Ok, one module loaded.  
*Main> main  
0  
*Main> main  
0
```

trebuie sa ne asiguram
ca ambele thred-uri
au terminat



➤ Sincronizare

```
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
```

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

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



➤ Sincronizare

```
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
```

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

```
*Main> main
2000
```



➤ Sincronizare: doua thread-uri incrementeaza acelasi contor

threadDelay nr

suspenda thread-ul pt.nr microsecunde

```
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"
```

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



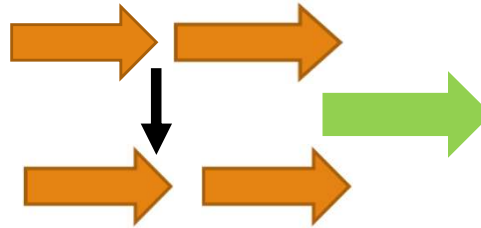
➤ Comunicarea thread-urilor

primește o valoare citită **msg**,
o pune în **a**

A

citeste valoarea din **a**
și o afisează

B



```
main = do
  aMVar <- newEmptyMVar
  forkIO (threadA aMVar )
  forkIO (threadB aMVar )
  putStrLn ("main thread ends")
  getLine
```

```
threadA a = do
  putStrLn "mesaj: "
  msg <- getLine
  if (msg == "end")
  then
    putMVar a msg
  else do
    putMVar a msg
    threadA a
```

```
threadB a = do
  x <- takeMVar a
  if x == "end"
  then putStrLn "I will stop now"
  else do
    putStrLn ("primit: " ++ x)
    threadB a
```

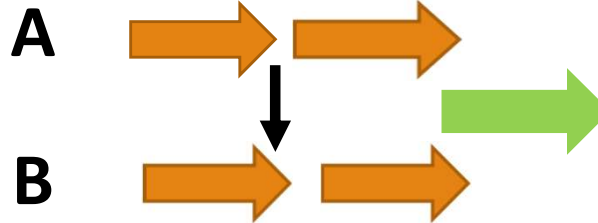
Ce problema de sincronizare poate să apară?



➤ Comunicarea thread-urilor

primește o valoare citită **msg**,
o pune în **a**

citeste valoarea din **a**
și o afisează



```
main = do
  aMVar <- newEmptyMVar
  forkIO (threadA aMVar )
  forkIO (threadB aMVar )
  putStrLn ("main thread ends")
  getLine
```

```
threadA a = do
  putStrLn "mesaj: "
  msg <- getLine
  if (msg == "end")
    then
      putMVar a msg
    else do
      putMVar a msg
      threadA a
```

```
threadB a = do
  x <- takeMVar a
  if x == "end"
    then putStrLn "I will stop now"
    else do
      putStrLn ("primit: " ++ x)
      threadB a
```

```
*Main> main
mmaeisna jt:h r
ead ends
```

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

Ce problema de sincronizare poate să apară?



➤ Comunicarea thread-urilor

Atentie!

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

```
main = do
  aMVar <- newEmptyMVar
  stdo <- newLock
  forkIO (threadA aMVar stdo)
  forkIO (threadB aMVar stdo)
  putStrLn ("main thread ends")
  getLine
```

```
tswrite stdo msg = do
  acquireLock stdo
  putStrLn msg
  releaseLock stdo

tsread stdo = do
  acquireLock stdo
  putStrLn "mesaj: "
  msg <- getLine
  releaseLock stdo
  return msg
```



➤ Comunicarea thread-urilor

```
threadA a s = do
    msg <- tsread s
    if (msg == "end")
    then
        putMVar a msg
    else do
        putMVar a msg
        threadA a
```

```
threadB a s = do
    x <- takeMVar a
    if x == "end"
    then tswrite s "I will stop now"
    else do
        tswrite s ("primit: " ++ x)
        threadB a
```

```
main = do
    aMVar <- newEmptyMVar
    stdo <- newLock
    forkIO (threadA aMVar stdo )
    forkIO (threadB aMVar stdo)
    putStrLn ("main thread ends")
    getLine
```

```
*Main> main
main thread ends
mesaj:
m1
"m1"
```

Thread-ul principal nu asteapta finalizarea activitatii thread-urilor.



➤ Comunicarea thread-urilor

```
threadA a s = do
  msg <- tsread s
  if (msg == "end")
    then
      putMVar a msg
    else do
      putMVar a msg
      threadA a
```

```
threadB a s = do
  x <- takeMVar a
  if x == "end"
    then do
      tswrite s "I will stop now"
      acquireLock sem
    else do
      tswrite s ("primit: " ++ x)
      threadB a
```

```
main = do
  aMVar <- newEmptyMVar
  stdo <- newLock
  sem <- newLock
  forkIO (threadA aMVar stdo)
  forkIO (threadB aMVar stdo sem)
  releaseLock sem
  putStrLn ("main thread ends")
  getLine
```

```
*Main> main
mesaj:
m1
mesaj:
m2
primit: m1
mesaj:
end
primit: m2
i will stop now
main thread ends
```



- 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



```
import Control.Concurrent
import Control.Monad
```

```
main = do
  m <- newEmptyMVar --buffer
  l <- newLock
  forkIO (producer m )
  consumer m 10 -- consuma 10 produse
  releaseLock l
  putStrLn "main thread ends"
```

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

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



➤ Producer-Consumer problem
MVar ca monitor



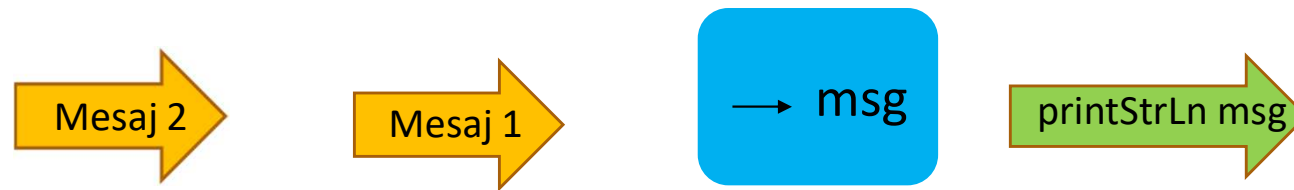
```
*Main> main
m1
>m1
m2
>m2
m3
>m3
m4
>m4
m5
>m5
main thread ends
```

```
import Control.Concurrent
import Control.Monad

main = do
    m <- newEmptyMVar --buffer
    l <- newLock
    forkIO (producer m )
    consumer m 5 -- consuma 5 produse
    releaseLock l
    putStrLn "main thread ends"
```



- Sincronizare: serviciu de logare
modelarea unui canal de comunicare simplu folosind **MVar**
http://chimera.labs.oreilly.com/books/1230000000929/ch07.html#sec_conc-logger

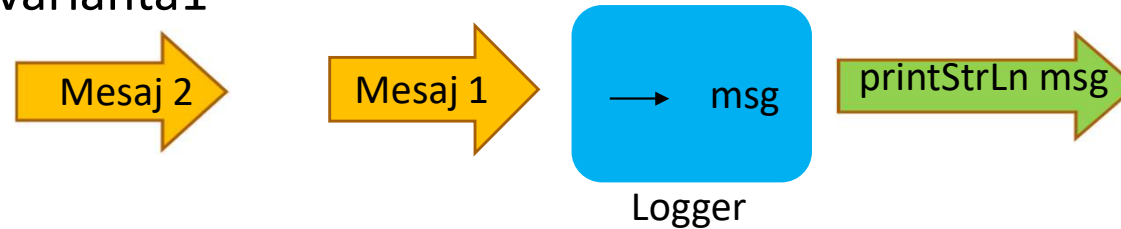


Cerinte:

- 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



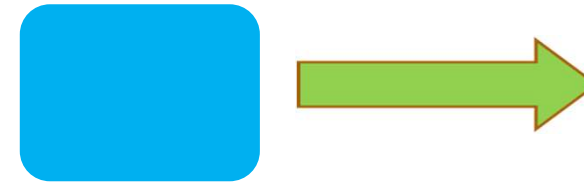
```
logMessage :: Logger -> String -> IO ()  
logMessage (Logger m) s = putMVar m s
```

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

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

Thread-ul principal trimite mesajele

Creaza thread-ul logger



```
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
```

Thread-ul logger le citeste si le scrie



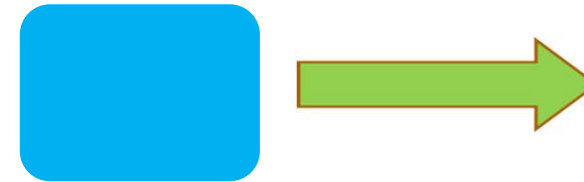
Exemplu: serviciu de logare- varianta1



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

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

main = do
    log <- initLogger
    logMessThread 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
```

Ce problema de sincronizare poate sa apara?



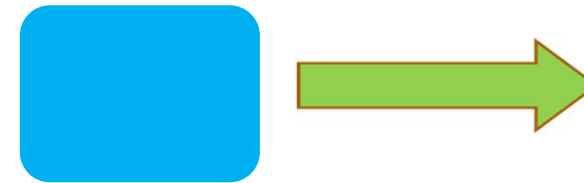
Exemplu: serviciu de logare- varianta1



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

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

main = do
    log <- initLogger
    logMessThread 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

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

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

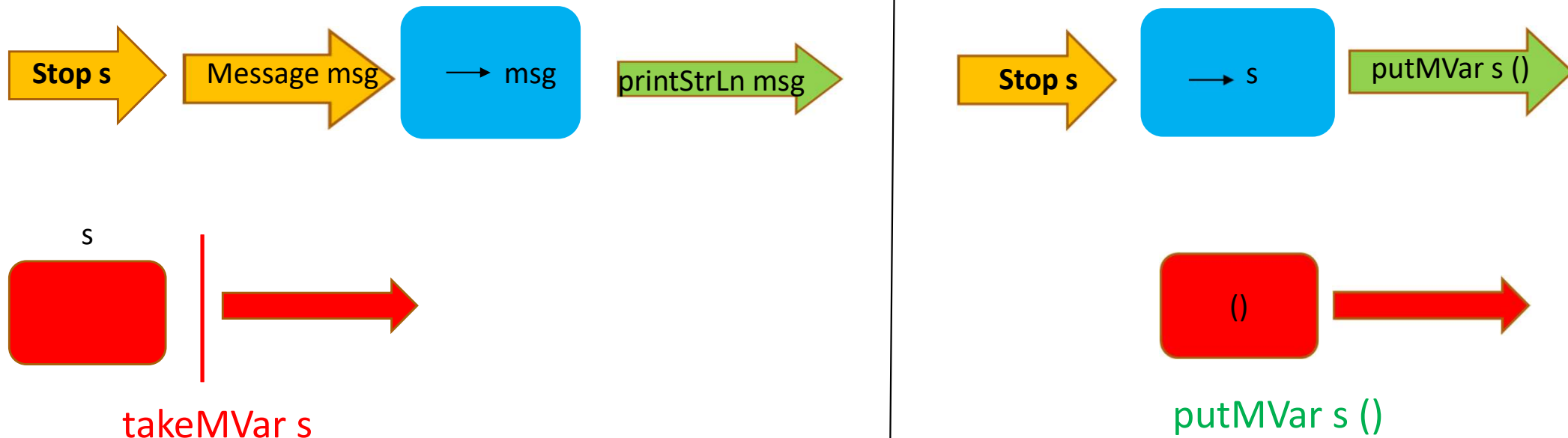
-- la fel

```
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
```

```
logMessThread :: Logger -> IO()
logMessThread 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
```

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



Exemplu: serviciu de logare

```
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

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

```
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

```
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
```



Pe saptamana viitoare!

