IMPLEMENTAREA CONCURENTEI IN LIMBAJE DE PROGRAMARE

Concurenta

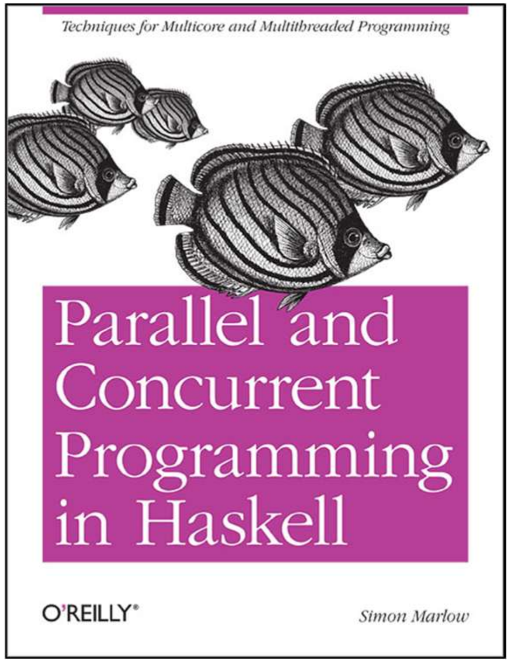
Threaduri

Memorie Partajata

Ioana Leustean

atomically

Haskell



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

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

➢Atomicitate

- Modalități de sincronizare de nivel scăzut: variabile atomice
  - Atomicitate fără sincronizare. mult mai rapide decât cu locks
  - Java: **AtomicInteger, AtomicBoolean, …**
    get(), set(), incrementAndGet(), addAndGet(int d), compareAndSet(int old, int new)
  - Haskell: **IORef a**
    newIORef, readIORef, writeIORef, atomicModifyIORef, atomicWriteIORef
  - Metodele sunt implementate folosind instrucțiuni hardware compare-and-swap

- Modalitati de sincronizare de nivel inalt: Software Transactional Memory (STM)
  - sincronizare fara lacate
  - blocuri de instructiuni executate atomic

- Exemplu: o tranzactie bancara(I)

```
type Account = MVar Int
```

```
deposit :: Account -> Int -> IO() -- depunere
deposit acc amount  = do
            x <- takeMVar acc
            putMVar acc (x + amount)
```

```
withdraw  :: Account -> Int -> IO() -- retragere
withdraw  acc amount  = do
            x <- takeMVar acc
            putMVar acc (x - amount)
```

```
showBalance  ::  Account -> String -> IO()  -- sold
showBalance  acc str = do
              x <- takeMVar acc
              putMVar acc x
              putStrLn ("Contul " ++ str ++ ": " ++ (show x))
```

- Exemplu: o tranzactie bancara (I)

```
transfer :: Account -> Account -> Int -> IO()
transfer from to amount = do
                withdraw from amount

                deposit to amount
```

data race

un alt thread ar putea observa o stare in care banii nu se gasesc in nici un cont

```
import Control.Concurrent
import Control.Monad

type Account = MVar Int)


main = do

      aMVar <- newMVar 1000
      bMVar <- newMVar 1000
      forkIO(transfer aMVar bMVar 300)
      forkIO (transfer bMVar aMVar 500)

      showBalance aMVar "a"
      showBalance bMVar "b"
```

➢ Exemplu: o tranzactie bancara (I)

```
transfer :: Account -> Account -> Int -> IO()
transfer from to amount = do
            withdraw from amount
     --     threadDelay (5^6)
            deposit to amount
```

```
transfer :: Account -> Account -> Int -> IO()
transfer from to amount = do
            withdraw from amount
     --     threadDelay (5^6)
            deposit to amount
```

compunerea unor operatii corecte poate avea ca rezultat o operatie eronata

```
main = do
    aMVar <- newMVar 1000
    bMVar <- newMVar 1000
    forkIO(transfer aMVar bMVar 300)
    forkIO (transfer bMVar aMVar 500)

    showBalance aMVar "a"
    showBalance bMVar "b"
```

```
Prelude> :l mybank.hs
[1 of 1] Compiling Main
Ok, modules loaded: Main.
*Main> main
Contul a: 700
Contul b: 500
```

```
*Main> main
Contul a: 1000
Contul b: 800
```

➢ STM

"Software transactional memory (STM) is a technique for simplifying concurrent programming by allowing multiple state-changing operations to be grouped together and performed as a single atomic operation. Strictly speaking, "software transactional memory" is an implementation technique, whereas the language construct we are interested in is "atomic blocks.""
S. Marlow, PCHP

"A transaction is a finite sequence of machine instructions, executed by a single process, satisfying the following properties:
Serializability: Transactions appear to execute serially, meaning that the steps of one transaction never appear to be interleaved with the steps of another. Committed transactions are never observed by different processors to execute indifferent orders.
Atomicity: Each transaction makes a sequence of tentative changes to shared memory. When the transaction completes, it either commits, making its changes visible to other processes (effectively) instantaneously, or it aborts, causing its changes to be discarded."
Maurice Herlihy, J. Eliot B. Moss *Transactional memory: architectural support for lock-free data structures*. *Isca '93. ACM SIGARCH Computer Architecture News - Special Issue: Proceedings of the 20th annual international symposium on Computer architecture (ISCA '93). May 1993*

➢ STM: Tranzactii bancare

```
transfer :: Account -> Account -> Int -> IO()
transfer from to amount = atomically  $ do
                          withdraw from amount
                          deposit to amount
```

```
Prelude> :m Control.Concurrent.STM
Prelude Control.Concurrent.STM> :t atomically
atomically :: STM a -> IO a
```

atomically

"takes an action as its argument, and performs it atomically. More precisely, it makes two guarantees:

Atomicity: the effects of atomically act become visible to another thread all at once. This ensures that no other thread can see a state in which money has been deposited in to but not yet withdrawn from from.

Isolation: during a call atomically act, the action act is completely unaffected by other threads. It is as if act takes a snapshot of the state of the world when it begins running, and then executes against that snapshot."

Simon Peyton Jones, Beautiful Concurrency

Monada STM
este asemanatoare monadei IO

type Account = **TVar** Int

TVar
variabile tranzactionale

```
deposit :: Account -> Int -> STM ()
deposit acc amount  = do
          x <- readTVar acc
          writeTVar acc (x + amount)
```

deposit
actiune STM

```
withdraw  :: Account -> Int -> STM ()
withdraw  acc amount  = do
          x <- readTVar acc
          writeTVar acc (x - amount)
```

withdraw
actiune STM

```
transfer :: Account -> Account -> Int -> IO()
transfer from to amount = atomically  $ do
                          withdraw from amount
                          deposit to amount
```

```
Prelude> :m Control.Concurrent.STM
Prelude Control.Concurrent.STM> :t atomically
atomically :: STM a -> IO a
```

atomically :: STM a -> IO a
executa atomic o actiune STM

- Monada STM

```
data STM a

instance Monad STM
atomically :: STM a -> IO a

data TVar a
newTVar   :: a -> STM (TVar a)
readTVar  :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()
```

Operatiile de baza ale monadei STM sunt scrierea si citirea variabilelor tranzactionale.

Variabilele tranzactionale sunt mutabile.
O variabila TVar **nu** poate fi goala.

Scrierea si citirea variabilelor tranzactionale se face **fara blocare**.

Actiunile STM au loc **atomic**.

O **tranzactie** este o actiune STM care este executata in monada IO folosind atomically

- Monada STM

```
data STM a

instance Monad STM
atomically :: STM a -> IO a

data TVar a
newTVar   :: a -> STM (TVar a)
readTVar  :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()
```

"Why is STM a different monad from IO?

The STM implementation relies on being able to roll back the effects of a transaction in the event of a conflict with another transaction [..]. A transaction can be rolled back only if we can track exactly what effects it has, and this would not be possible if arbitrary I/O were allowed inside a transaction—we might have performed some I/O that cannot be undone, like making a noise or launching some missiles."

S.Marlow, PCHP

➢ Implementarea STM

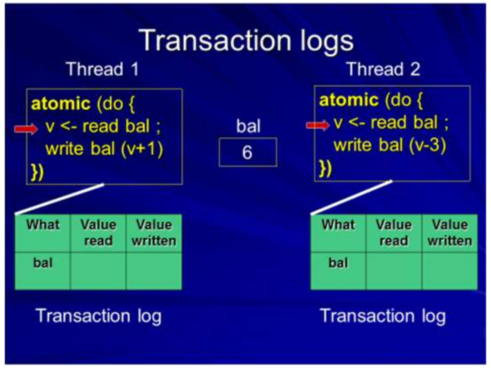
"One particularly attractive implementation is well established in the database world, namely optimistic execution. When (atomically act) is performed, a **thread-local transaction log is allocated, initially empty**. Then the action act is performed, **without taking any locks at all**. While performing act, each call to writeTVar writes the address of the TVar and its new value into the log; it does not write to the TVar itself. Each call to readTVar first searches the log (in case the TVar was written by an earlier call to writeTVar); if no such record is found, the value is read from the TVar itself, and the TVar and value read are recorded in the log. In the meantime, other threads might be running their own atomic blocks, reading and writing TVars like crazy.

When the action act is finished, the implementation first validates the log and, if validation is successful, commits the log. The validation step examines each readTVar recorded in the log, and checks that the value in the log matches the value currently in the real TVar. If so, validation succeeds, and the commit step takes all the writes recorded in the log and writes them into the real TVars.

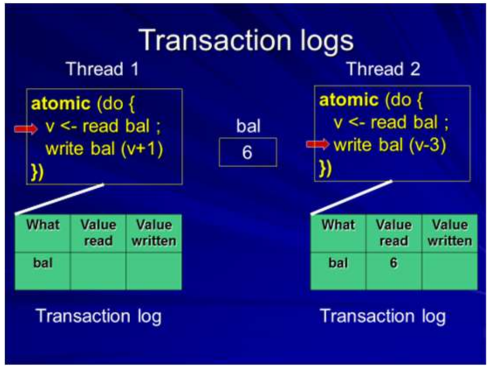
What if validation fails? Then the transaction has had an inconsistent view of memory. So we abort the transaction, re-initialise the log, and run act all over again"

Simon Peyton Jones, Beautiful Concurrency

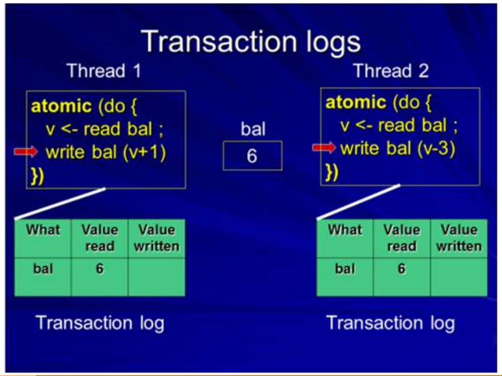
1



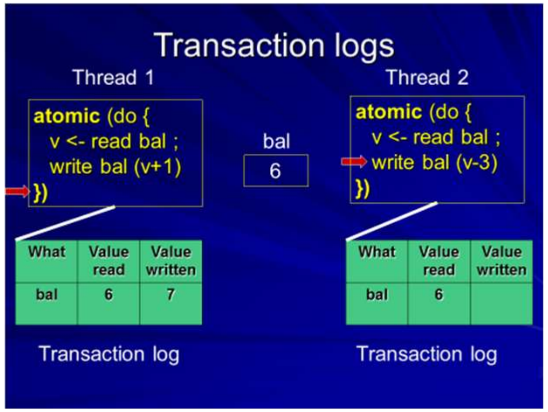
2



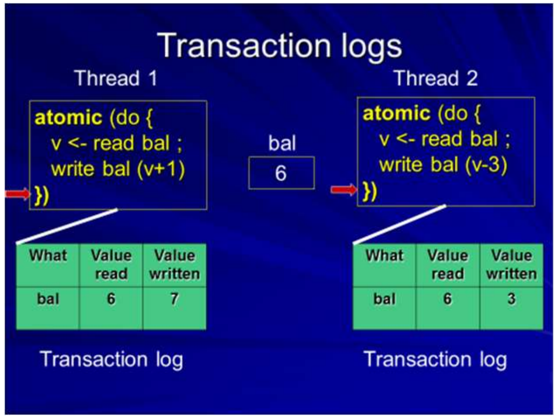
3



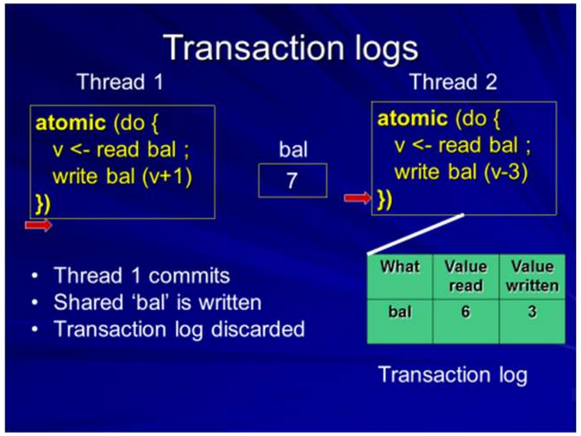
4



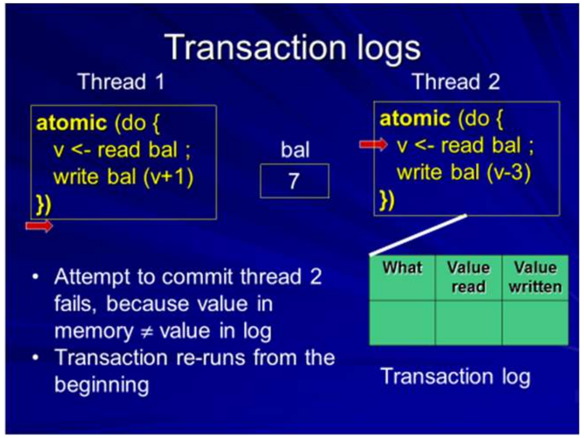
5

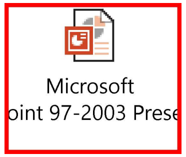


6



7





click pe prezentare

T. Harris, M. Herlihy, S. Marlow, S. Peyton Jones, Concurrency unlocked

- Variabile mutabile: IORef, MVar, **TVar**

```
import Data.IORef
-- variabile mutabile in monada IO

newIORef :: a -> IO (IORef a)
readIORef :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()
```

```
add :: IORef Int -> Int -> IO()
add rref n = do
             val <- readIORef rref
             writeIORef rref

main = do
         rref <- newIORef 0
         add rref 10
         val <- readIORef rref
         print val
```

```
import Control.Concurrent.STM.TVar
-- variabile tranzactionale
-- variabile mutabile in monada STM

newTVar :: a -> STM (TVar a)
readTVar :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()
```

```
import Control.Concurrent.MVar
-- variabile de sincronizare
-- variabile mutabile in monada IO

newEmptyMVar :: IO (MVar a)
newMVar :: a -> IO (MVar a)
takeMVar :: MVar a -> IO a       -- blocheaza thread-ul
putMVar :: MVar a -> a -> IO () -- blocheaza thread-ul
```

```haskell
import Control.Concurrent
import Control.Monad
import Control.Concurrent.STM

type Account = TVar Int

deposit :: Account -> Int -> STM ()
deposit acc amount  = do
            x <- readTVar acc
            writeTVar acc (x + amount)

withdraw  :: Account -> Int -> STM ()
withdraw  acc amount  = do
            x <- readTVar acc
            writeTVar acc (x - amount)

showBalance :: Account -> String -> IO()
showBalance acc str =  do
                    x <- atomically $ readTVar acc
                    putStrLn ("Contul " ++ str ++ ": " ++ (show x))
```

mybankstm.hs

```haskell
transfer :: Account -> Account -> Int -> IO()
transfer from to amount = atomically  $ do
                          withdraw from amount
                          deposit to amount
```

compozitionalitate

```haskell
main = do
        (a,b) <- atomically $ do
                      a <- newTVar 1000
                      b <- newTVar 1000
                      return (a,b)
       forkIO(transfer a b 300)
       forkIO (transfer b a 500)
       showBalance a  "a"
       showBalance b  "b"
```

```
Prelude> :l mybankstm.hs
[1 of 1] Compiling Main
Ok, modules loaded: Main.
*Main> main
Contul a: 1200
Contul b: 800
```

➢ Blocare (blocking)

"Suppose that a thread should *block* if it attempts to overdraw an account (i.e. withdraw more than the current balance). Situations like this are common in concurrent programs: for example, a thread should block if it reads from an empty buffer, or when it waits for an event. We achieve this in STM by adding the single function retry, whose type is

```
retry :: STM a
```

The semantics of retry are simple: if a retry action is performed, the current transaction is abandoned and retried at some later time."

```
limitedWithdraw :: Account -> Int -> STM ()
limitedWithdraw acc amount = do
   bal <- readTVar acc
   if amount > 0 && amount > bal
   then retry
   else writeTVar acc (bal - amount)
```

sau

```
check :: Bool -> STM ()
check True  = return ()
check False = retry
```

```
limitedWithdraw :: Account -> Int -> STM ()
limitedWithdraw acc amount = do
      bal <- readTVar acc
      check (amount <= 0 || amount <= bal)
      writeTVar acc (bal - amount)
```

Simon Peyton Jones, Beautiful Concurrency

➢ Blocare (blocking)

"Suppose that a thread should *block* if it attempts to overdraw an account (i.e. withdraw more than the current balance). Situations like this are common in concurrent programs: for example, a thread should block if it reads from an empty buffer, or when it waits for an event. We achieve this in STM by adding the single function retry, whose type is

retry :: STM a

The semantics of retry are simple: if a retry action is performed, the current transaction is abandoned and retried at some later time."
Simon Peyton Jones, Beautiful Concurrency

"The retry operation uses the transaction log to find out which TVars were accessed by the transaction, because changes of any of these TVars must trigger a rerun of the current transaction.

Hence, each TVar has a watch list of threads that should be woken up if the TVar is modified and retry adds the current thread to the watch list of all the TVars read during the current transaction."
S. Marlow, PCHP

➢ Alegerea (choice)

"Suppose you want to withdraw money from account A if it has enough money, but if not then withdraw it from account B? For that, we need the ability to choose an alternative action if the first one retries.
To support choice, STM Haskell has one further primitive action, called orElse, whose type is

```
Prelude Control.Concurrent.STM> :t orElse
orElse :: STM a -> STM a -> STM a
```

Its semantics are as follows: **the action (orElse a1 a2) first performs a1;**
if a1 retries (i.e. calls retry), it tries a2 instead;
if a2 also retries, the whole action retries. "

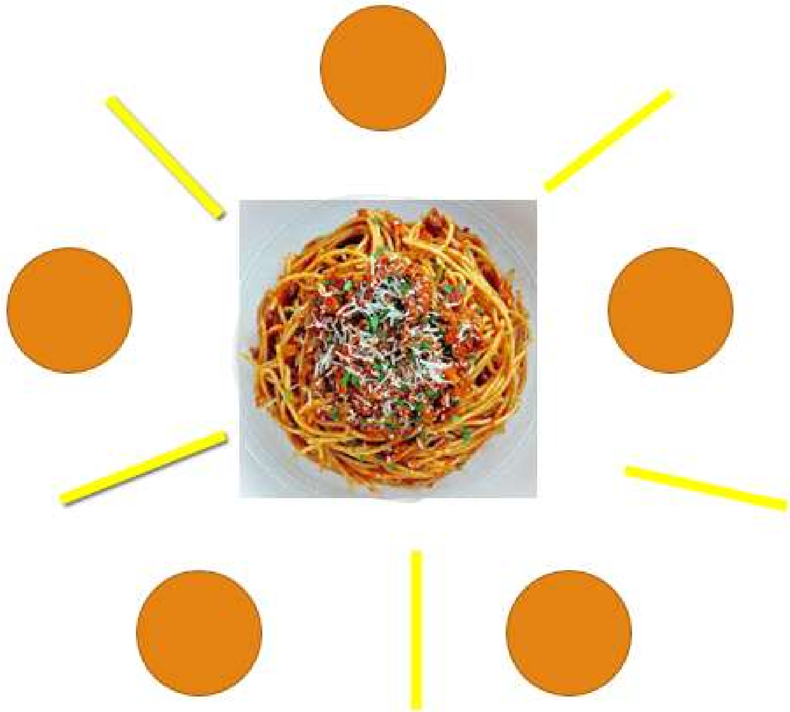
```
limitedWithdraw :: Account -> Int -> STM ()
limitedWithdraw acc amount = do
   bal.<- readTVar acc
   if amount > 0 && amount > bal
   then retry
   else writeTVar acc (bal - amount)
```

Simon Peyton Jones, Beautiful Concurrency

```
limitedWithdraw2 :: Account -> Account -> Int -> STM ()
limitedWithdraw2 acc1 acc2 amt
  = orElse (limitedWithdraw acc1 amt) (limitedWithdraw acc2 amt)
```

Exercitiu: Modificati mybankstm.hs adaugand retry si orElse

- The Dining Philosophers



http://rosettacode.org/wiki/Dining_philosophers
http://www.tobiasmuehlbauer.com/2011/07/24/stm-haskell-dining-philosophers-problem/
http://www-ps.informatik.uni-kiel.de/~fhu/projects/stm.pdf

- Dining Philosophers

Fiecare filozof executa
la infinit urmatorul ciclu

asteapta sa manance

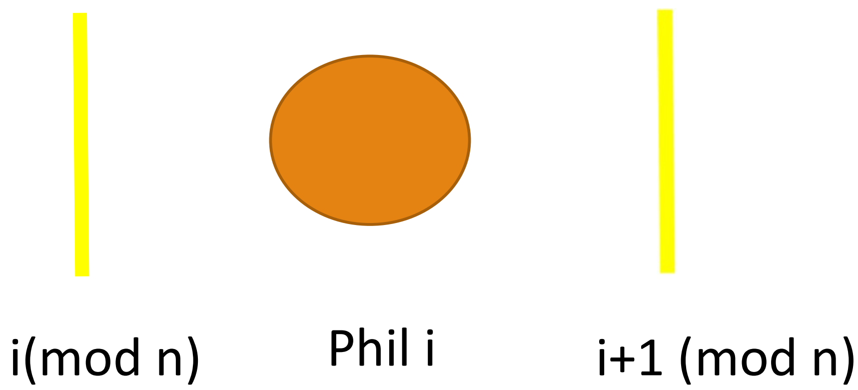
ia furculita stanga
Ia furculita dreapta

mananca

elibereaza furculita stanga
elibereaza furculita dreapta

mediteaza

n = numarul de filozofi



Fiecare filozof executa la infinit urmatorul ciclu

-- asteapta sa manance

ia furculita stanga
Ia furculita dreapta

mananca

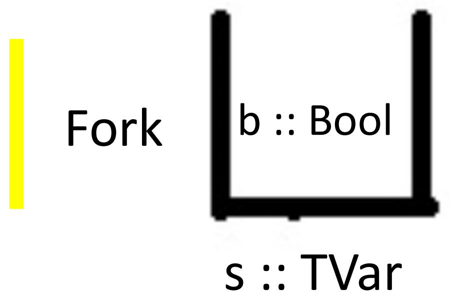
elibereaza furculita stanga
elibereaza furculita dreapta

mediteaza

actiuni atomice - elimina deadlock

durata finita – elimina starvation

- Dining Philosophers – varianta 1
  dinnersrc1.hs




```
type Fork = TVar Bool            -- True  daca furculita este libera

takeFork :: Fork -> STM ()
takeFork s = do
                 b <- readTVar s
                 if b then writeTVar s False
                     else retry   -- asteapta pana se elibereaza furculita

releaseFork ::  Fork -> STM ()
releaseFork fork = writeTVar fork True
```

Un filozof

asteapta sa manance

ia furculita stanga
Ia furculita dreapta

mananca

elibereaza furculita stanga
elibereaza furculita dreapta

mediteaza

```
import System.Random
type Name = String

runPhilosopher :: (Name, (Fork, Fork)) -> IO ()
runPhilosopher (name, (left, right)) = forever $ do
    putStrLn (name ++ " is hungry.")
    atomically $ do
        takeFork left
        takeFork right
    putStrLn (name ++ " got two forks  and is now eating.")
    delay <- randomRIO (1,10)
    threadDelay (delay * 1000000)
    putStrLn (name ++ " is done eating. Going back to thinking.")
    atomically $ do
        releaseFork left
        releaseFork right
    delay <- randomRIO (1, 10)
    threadDelay (delay * 1000000)
```

```haskell
philosophers :: [String]
philosophers = ["Aristotle", "Kant", "Spinoza", "Marx", "Russel"]


main = do
        forks <- atomically $ do
                        sticks <- mapM (const (newTVar True)) [1..5]
                        return sticks

    let forkPairs  = zip forks ((tail  forks) ++ [head forks])
        philosophersWithForks = zip philosophers forkPairs

    putStrLn "Running the philosophers. Press enter to quit."

    mapM_ (forkIO . runPhilosopher) philosophersWithForks

    getLine
```

```
Prelude> :t const
const :: a -> b -> a
Prelude> map (const True) [1..5]
[True,True,True,True,True]
```

```
Prelude> :l dinnersrc1.hs
[1 of 1] Compiling Philosophers     ( dinnersrc1
Ok, modules loaded: Philosophers.
*Philosophers> main
Loading package stm-2.4.2 ... linking ... done.
Running the philosophers. Press enter to quit.
Kant is hungry.
Kant got two forks  and is now eating.
Spinoza is hungry.
Marx is hungry.
Marx got two forks  and is now eating.
Russel is hungry.
Aristotle is hungry.
Marx is done eating. Going back to thinking.
Russel got two forks  and is now eating.
Marx is hungry.
Russel is done eating. Going back to thinking.
Marx got two forks  and is now eating.
Kant is done eating. Going back to thinking.
Aristotle got two forks  and is now eating.
Kant is hungry.
Russel is hungry.
```

➢ Implementarea MVar folosind TVar

o data de tip MVar are doua stari:

- goala - nu contine nici o valoare (blocheaza operatia takeMVar; permite operatia putMVar)
- plina - contine o valoare (permite operatia takeMVar; blocheaza operatia putMVar)

```
data   TMVar a = TMVar (TVar (Maybe a))

-- Nothing indica faptul ca variabila e goala
```

```
newEmptyTMVar :: STM (TMVar a)
newEmptyTMVar = do
        t <- newTVar Nothing
        return (TMVar t)
```

Composable Memory Transactions
T. Harris, S. Marlow, S.P. Jones, M. Herlihy
PPoPP ' 05
PCPH, Cap.10, Blocking

- TMVar – implementarea MVar folosind TVar

```
takeTMVar :: TMVar a -> STM a
takeTMVar (TMVar t) = do
        m <- readTVar t
        case m of
            Nothing -> retry -- blocare
            Just a -> do
                writeTVar t Nothing
                return a
```

```
putTMVar :: TMVar a -> a -> STM ()
putTMVar (TMVar t) = do
        m <- readTVar t
        case m of
            Just _ -> retry    -- blocare
            Nothing  -> do
                writeTVar t  (Just a)
                return ()
```

Composable Memory Transactions
T. Harris, S. Marlow, S.P. Jones, M. Herlihy
PPoPP ' 05
PCPH, Cap.10, Blocking

➢ MVar vs TMVar

```
takeBothMVar :: MVar a -> MVar b -> IO (a,b)
takeBothMVar tv tw  = do
                      v < - takeMVar tv
                      w <- takeMVar tw
                      return (v,w)
```

putMVar tv x

```
takeBothTMVar :: TMVar a -> TMVar b -> IO (a,b)
takeBothTMVar tv tw  =  atomically $ do
                        v < - takeTMVar tv
                        w <- takeTMVar tw
                        return (v,w)
```

```
Prelude Control.Concurrent.STM> :t takeTMVar
takeTMVar :: TMVar a -> STM a
```

- Dining Philosophers - varianta2
  dinnersrc3.hs

```
type Fork = TMVar Int

newFork :: Int -> STM Fork
newFork i = newTMVar i


takeFork :: Fork -> STM Int
takeFork fork = takeTMVar fork

releaseFork :: Int -> Fork -> STM ()
releaseFork i fork = putTMVar fork i
```

```
forks <- mapM newFork [1..5]
```

```
import System.Random
type Name = String

runPhilosopher :: (Name, (Fork, Fork)) -> IO ()
runPhilosopher (name, (left, right)) = forever $ do
    putStrLn (name ++ " is hungry.")
    (leftv, rightv)<- atomically $ do
                        leftv <- takeFork left
                        rightv <-takeFork right
                        return (leftv,rightv)
    putStrLn (name ++ " got forks"++ (show leftv)++","++
                              (show rightv)++  " and is now eating.")
    delay <- randomRIO (1,10)
    threadDelay (delay * 1000000)
    putStrLn (name ++ " is done eating. Going back to thinking.")
    atomically $ do
              releaseFork leftv left
              releaseFork rightv right
    delay <- randomRIO (1, 10)
    threadDelay (delay * 1000000)
```

- Async - comunicare asincrona (folosind MVar)
  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
```

➢ Async - comunicare asincrona (folosind MVar)

geturl3.hs ©2012, Simon Marlow

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

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

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

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

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

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

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

➢ Async cu TMVar

```
data Async a = Async (TMVar a)

async :: IO a -> IO (Async a)
async action = do
                 var <- atomically $ do
                                       var <- newEmptyTMVar
                                       return var
                 forkIO (do r <- action; (atomically. putTMVar var)  r)
                 return (Async var)

waitSTM :: Async a -> STM a
waitSTM (Async var) = readTMVar var
```

- Async cu TMVar

```
data Async a = Async (TMVar a)

async :: IO a -> IO (Async a)
async action = do
      var <- atomically $ do
              var <- newEmptyTMVar
              return var
   forkIO (do r <- action; (atomically. putTMVar var)  r)
   return (Async var)

waitSTM :: Async a -> STM a
waitSTM (Async var) = readTMVar var
```

```
waitAll :: [Async a] -> IO ()
waitAll asyncs = atomically $ mapM_ waitSTM asyncs
```

monitorizeaza terminarea actiunilor global,
intoarce dupa terminarea tuturor actiunilor din lista

➢ waitAll

```
putStrLn "Running the philosophers."
as0 <- async $ runPhilosopher 2  (philosophersWithForks !! 0)    --Aristotel
as1 <- async $ runPhilosopher 1  (philosophersWithForks !! 1)    -- Kant
as2 <- async $ runPhilosopher 3  (philosophersWithForks !! 2)    -- Spinoza
waitAll [as0,as1,as2]
putStrLn "WAIT RETURNED"
getLine
```

```
runPhilosopher :: Int -> (Name, (Fork, Fork)) -> IO ()
runPhilosopher n (name, (left, right)) = if (n==0) then return ()
                                          else  do
                                             putStrLn (name ++ " is hungry.")
                                                 ....
                                             runPhilosopher (n-1) (name, (left, right))
```

```
Kant is leaving.
Aristotle got two forks  and is now eating.
Aristotle is leaving.
Spinoza got two forks  and is now eating.
Spinoza is done eating. Going back to thinking.
Spinoza is hungry.
Spinoza got two forks  and is now eating.
Spinoza is leaving.
WAIT RETURNED
```

➢ Dining Philosophers – varianta in care astept ca fiecare sa manance de n ori

dinnersrc4.hs

```
runPhilosopher n (name, (left, right)) =   …….

main = do
         forks <- atomically $ do
                             sticks <- mapM (const (newTVar True)) [1..5]
                             return sticks

         let forkPairs  = zip forks ((tail  forks) ++ [head forks])
             philosophersWithForks = zip philosophers forkPairs
             n = 2
        putStrLn "Running the philosophers. "
        as <- mapM (async . (runPhilosopher n)) philosophersWithForks
        waitAll as
        getLine
```

```
Aristotle is done eating. Going back to thinking.
Kant got two forks  and is now eating.
Aristotle is hungry.
Kant is done eating. Going back to thinking.
Aristotle got two forks  and is now eating.
Marx is done eating. Going back to thinking.
Spinoza got two forks  and is now eating.
Kant is hungry.
Spinoza is done eating. Going back to thinking.
Spinoza is hungry.
Spinoza got two forks  and is now eating.
Aristotle is leaving.
Russel got two forks  and is now eating.
Russel is done eating. Going back to thinking.
Marx is hungry.
Spinoza is leaving.
Kant got two forks  and is now eating.
Marx got two forks  and is now eating.
Marx is leaving.
Russel is hungry.
Russel got two forks  and is now eating.
Russel is leaving.
Kant is leaving.
```

➢ Dining Philosophers – varianta in care astept ca fiecare sa manance de n ori

```
runPhilosopher :: Int ->  (Name, (Fork, Fork)) -> IO ()
runPhilosopher n (name, (left, right)) =  if n == 0
                                           then return ()
                                           else do
                                                 putStrLn (name ++ " is hungry.")
                                                 atomically $ do
                                                     takeFork left
                                                     takeFork right
                                                putStrLn (name ++ " got two forks  and is now eating.")
                                                delay <- randomRIO (1,10)
                                               threadDelay (delay * 1000000)
                                               if (n> 1) then   putStrLn (name ++ " is done eating. Going back to thinking.")
                                                        else   putStrLn (name ++ " is leaving.")
                                                atomically $ do
                                                          releaseFork left
                                                          releaseFork right
                                                delay <- randomRIO (1, 10)
                                                threadDelay (delay * 1000000)
                                                runPhilosopher (n-1) (name, (left, right))
```

➢ Monada Either a b

```
Prelude>  let nat x = if (x>=0) then Left x else Right "negativ"
Prelude> :t nat
nat :: (Ord a, Num a) => a -> Either a [Char]
Prelude> :t Left
Left :: a -> Either a b
Prelude> :t Right
Right :: b -> Either a b
```

```
Prelude> :t fmap
fmap :: Functor f => (a -> b) -> f a -> f b
```

```
waitEither :: Async a -> Async b -> IO (Either a b)
waitEither x y = atomically $
                    fmap Left (waitSTM x)
                            `orElse`
                    fmap Right (waitSTM y)
```

http://chimera.labs.oreilly.com/books/1230000000929/ch10.html#sec_stm-async

- Async cu TMVar

```
data Async a = Async (TMVar a)

async :: IO a -> IO (Async a)
async action = do
      var <- atomically $ do
              var <- newEmptyTMVar
              return var
   forkIO (do r <- action; (atomically. putTMVar var)  r)
   return (Async var)

waitSTM :: Async a -> STM a
waitSTM (Async var) = readTMVar var
```

```
waitAny :: [Async a] -> IO a
waitAny asyncs = atomically $ foldr orElse retry $ map waitSTM asyncs
```

intoarce cand una din actiuni se termina

waitAny

```
putStrLn "Running the philosophers."
as0 <- async $ runPhilosopher 3  (philosophersWithForks !! 0)  -- Aristotel
as1 <- async $ runPhilosopher 1  (philosophersWithForks !! 1)  -- Kant
as2 <- async $ runPhilosopher 3  (philosophersWithForks !! 2)  -- Spinoza
waitAny [as0,as1,as2]
putStrLn "WAIT RETURNED"
getLine
```

```
Kant is leaving.
Aristotle got two forks  and is now eating.
WAIT RETURNED
Aristotle is done eating. Going back to thinking.
Spinoza got two forks  and is now eating.
Spinoza is done eating. Going back to thinking.
Spinoza is hungry.
Spinoza got two forks  and is now eating.
Spinoza is leaving.
Aristotle is hungry.
Aristotle got two forks  and is now eating.
Aristotle is leaving.

""
```

Programul continua
pana se efectueaza getLine

Pe saptamana viitoare!