Konkurens programozás

Mitől konkurens egy program?

Konkurens programozás során több vezérlési szálat hozunk létre.

- A szálak időben egyszerre hajtódnak végre – hatásaik összefésülődnek
- A tényleges futási eredmény nem determinisztikus

A konkurens programok fókuszában a külvilággal történő, többirányú kommunikáció áll.

- A konkurenciának köszönhetően az ilyen programok modulárisan építhetőek fel
- Például a program futása során a felhasználók felé egy reszponzív felületet akarunk adni, miközben a nem interaktív feladatok a háttérben mennek

Konkurencia tisztán funkcionális esetben

Nincs értelme szálak alkalmazásának tisztán funkcionális programok esetében:

- Nincsenek (implicit) mellékhatások
- A kiértékelés sorrendje nem kötött

A konkurens programozást ezért ilyen esetekben valójában mellékhatással rendelkező ("effectful") programrészek strukturálására használhatjuk.

Következmények:

- Az *IO* monád
- Nem determinisztikus viselkedés

Nem azonos a párhuzamos programozással, lásd később!

Szálak létrehozása

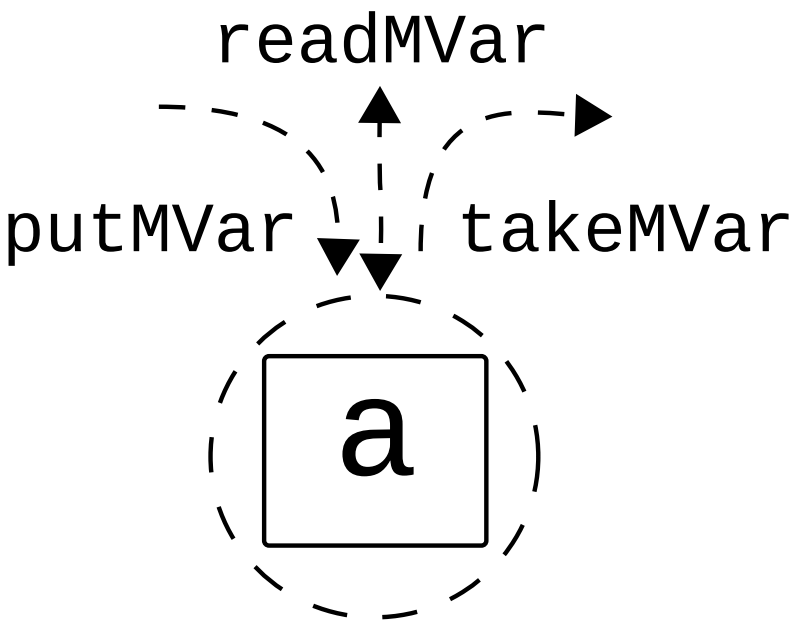
```
import Control.Concurrent
-- forkIO :: IO () → IO ThreadId
import System.IO

multiple :: Char → IO ()
multiple c = forM_ [1 ... 10000] $ λ _ . putChar c

concurrentAction :: IO ()
concurrentAction = do
  hSetBuffering stdout NoBuffering
  forkIO (multiple '#')
  forkIO (multiple '*')
  return ()
```

Figyelem: a *main* befejeződésével az összes többi aktív szál is befejeződik.

Kommunikáció szálak között

MVar ("mutable variable", védett mutató): egy *alacsonyszintű* de mégis *sokoldalú* absztrakció, amely szálak közti kommunikációt tesz lehetővé.

Két állapota lehet: *üres* vagy *teli*.



Kommunikáció szálak között (példák)

```
communicate :: IO ()
communicate = do
  m ← newEmptyMVar
  forkIO $ do
    putStrLn "Waiting..."
    () ← takeMVar m
    putStrLn "Got signal."
  getLine
  putStrLn "Sending..."
  putMVar m ()

concurrentFileRead :: [FilePath] → IO [String]
concurrentFileRead files = do
  mvars ← replicateM (length files) newEmptyMVar
  forM_ (files `zip` mvars) $ λ (f, m) .
     forkIO $ readFile f >>= putMVar m
  forM mvars $ λ i . takeMVar i
```

Az `MVar` értékek használata

Az `MVar` több különféle kommunikációs és szinkronizációs mintát általánosít (és egyesít). Néhány példa az alkalmazásra:

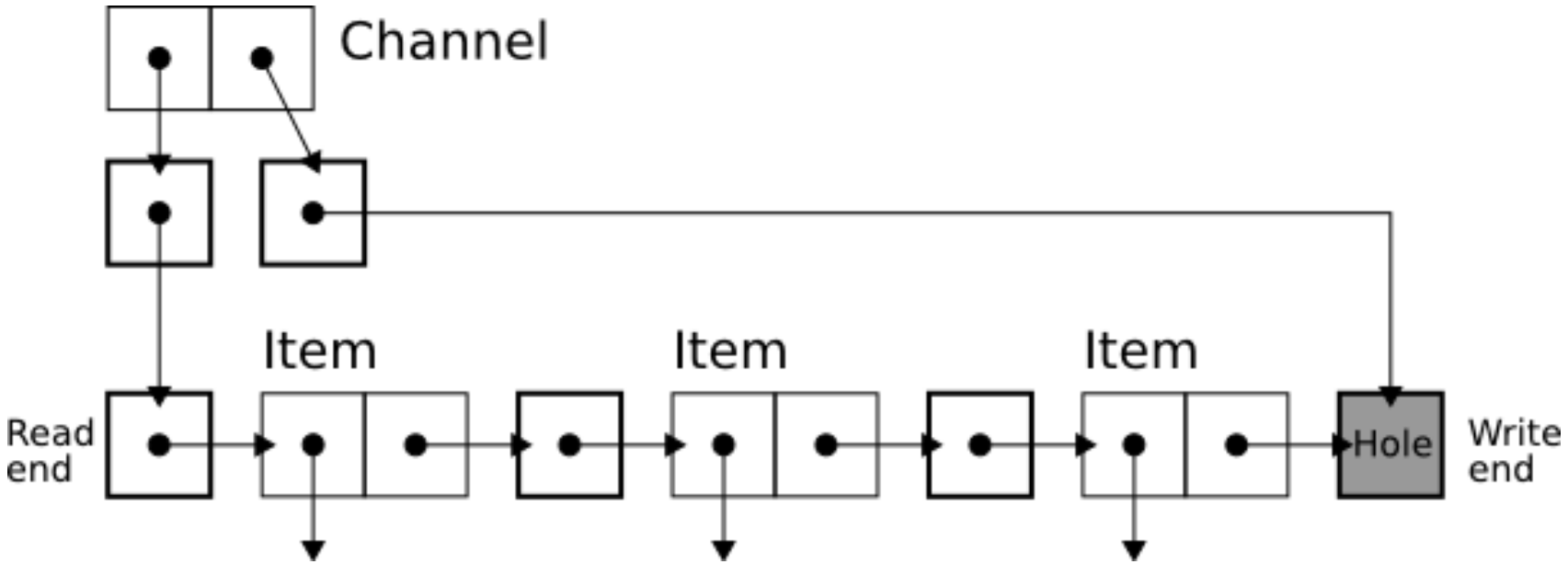
- Egyelemes üzenetküldés csatornával
- Kölcsönös kizárás
- Osztott, felülírható(!) állapot – amely állapot nem feltétlenül egy Haskellben megvalósított adatszerkezet
- Nagyobb konkurens adatszerkezetek felépítése

Az ütemezés *viszonylag fair*:

- Mindegyik szál előbb-utóbb sorra kerül
- Nem mindegyik szál kap viszont ugyanannyi időt
- Megvalósítás: FIFO sorokkal, az ébresztés és a művelet elvégzése atomi

Csatornák: *MVar* értékek kompozíciója

$$
\begin{aligned}
&\textbf{type}\ \mathit{Stream}\ \alpha &&= \mathit{MVar}\ (\mathit{Item}\ \alpha) \\
&\textbf{data}\ \mathit{Item}\ \alpha &&= \mathit{Item}\ \alpha\ (\mathit{Stream}\ \alpha) \\
&\textbf{data}\ \mathit{Chan}\ \alpha &&= \\
&\quad \mathit{Chan}(\mathit{MVar}\ (\mathit{Stream}\ \alpha)) \\
&\qquad\qquad (\mathit{MVar}\ (\mathit{Stream}\ \alpha))
\end{aligned}
$$



Csatornák: műveletek (implementáció)

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

newChan :: IO (Chan α)
newChan = do
  hole     ← newEmptyMVar
  readVar  ← newMVar hole
  writeVar ← newMVar hole
  return (Chan readVar writeVar)

writeChan :: Chan α → α → IO ()
writeChan (Chan _ writeVar) val = do
  newHole ← newEmptyMVar
  oldHole ← takeMVar writeVar
  putMVar oldHole (Item val newHole)
  putMVar writeVar newHole

readChan :: Chan α → IO α
readChan (Chan readVar _) = do
  stream        ← takeMVar readVar
  Item val tail ← takeMVar stream
  putMVar readVar tail
  return val
```

Csatornák: *MVar* értékek kompozíciója (példa)

```
import Control.Concurrent
import Control.Concurrent.Chan

channels :: IO ()
channels = do
  ch ← newChan
  t ← forkIO $ forever $ getLine >>= writeChan ch
  contents ← getChanContents ch
  print $ takeWhile (≠ "q") contents
  killThread t
```

Közjáték: (aszinkron) kivételek

```
import Control.Exception

data Async α = Async ThreadId (MVar (Either SomeException α))

-- try :: Exception ε ⇒ IO α → IO (Either ε α)
async :: IO α → IO (Async α)
async action = do
  m ← newEmptyMVar
  t ← forkIO $ do
    r ← try action
    putMVar m r
  return (Async t m)

waitCatch :: Async α → IO (Either SomeException α)
waitCatch (Async _ m) = readMVar m

-- throwIO :: Exception ε ⇒ ε → IO α
wait :: Async α → IO α
wait x = do
  r ← waitCatch x
  case r of
    Left e  → throwIO e
    Right x → return x
```

Közjáték: (aszinkron) kivételek (példák)

```
-- throwTo :: Exception ε ⇒ ThreadId → ε → IO ()
cancel :: Async α → IO ()
cancel (Async t _) = throwTo t ThreadKilled

longAction :: Int → IO Int
longAction n = do
  threadDelay n
  return n
cancellableAction :: [Int] → IO ()
cancellableAction ns = do
  as ← forM ns $ λ i . async (longAction i)
  t ← async $ do
    hSetBuffering stdin NoBuffering
    forever $ do
      c ← getChar
      when (c ≡ 'q') $
        forM_ as $ λ i . cancel i
  rs ← forM as $ λ i . waitCatch i
  cancel t
  print rs
```

Közjáték: (aszinkron) kivételek (magyarázat)

A szálak leállíthatóságának támogatásához fontos eldöntenünk, hogy:

- a leállítandó szálnak kell-e figyelnie valamilyen külső állapotot a megálláshoz (veszélyes, mert esetleg mégsem teszi), vagy
- tőle függetlenül is leállíthatjuk-e, viszont akkor a védett osztott változók konzisztenciájáról gondoskodnunk kell.

A második megoldás imperatív nyelvek esetében szinte elképzelhetetlen, azonban Haskell esetén lényegében az egyetlen értelmes lehetőség.

Közjáték: (aszinkron) kivételek (további példák)

```
unsafeModifyMVar :: MVar α → (α → IO α) → IO ()
unsafeModifyMVar m f = do
  x ← takeMVar m
  r ← f x `catch` λ e . do
    putMVar m x
    throw (e :: SomeException)
  putMVar m r

-- mask :: ((IO α → IO α) → IO β) → IO β
-- mask: ne legyen megszakítható az akció
modifyMVarWithMask :: MVar α → (α → IO α) → IO ()
modifyMVarWithMask m f = mask $ λ restore . do
  x ← takeMVar m -- még ilyenkor is megszakítható
  r ← restore (f x) `catch` λ e . do
    putMVar m x
    throw (e :: SomeException)
  putMVar m r -- ez is megszakítható, ha blokkol

-- "compare and swap"
casMVar :: Eq α ⇒ MVar α → α → α → IO Bool
casMVar m old new =
  modifyMVar m $ λ cur .
    if cur ≡ old
      then return (new, True)
      else return (cur, False)
```

Közjáték: holtpontok

Naiv, de abszolúte lehetséges eset:

$$
\begin{aligned}
&\mathit{nonTerminatingWait} :: \mathit{IO}\ \alpha \\
&\mathit{nonTerminatingWait} = \mathbf{do} \\
&\quad m \leftarrow \mathit{newEmptyMVar} \\
&\quad \mathit{takeMVar}\ m
\end{aligned}
$$

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

A valós életben ilyen bagatell hibát valószínűleg senki sem fog elkövetni, viszont...

Közjáték: holtpontok (folytatás)

```
type Dir α β = Map α (MVar (Set β))

moveElem :: (Ord α, Ord β) ⇒ Dir α β → β → α → α → IO ()
moveElem dir x a b = do
  let ma = dir ! a
  let mb = dir ! b
  va ← takeMVar ma
  vb ← takeMVar mb
  putMVar ma (Set.delete x va)
  putMVar mb (Set.insert x vb)

deadlockAction :: Int → Int → IO ()
deadlockAction n m = do
  ma ← newMVar (Set.fromList [1 ... n])
  mb ← newMVar (Set.fromList [(n + 1) ... (n + m)])
  let d = Map.fromList [('a', ma), ('b', mb)]
  forkIO $
    forM_ [1 ... n] $ λ i . moveElem d i 'a' 'b'
  forkIO $
    forM_ [(n + 1) ... (n + m)] $ λ i . moveElem d i 'b' 'a'
  readMVar ma >>= print
  readMVar mb >>= print
```

... lásd az *„étkező filozófusok”* problémáját.

Közjáték: összefoglalás

Nem könnyű az élet az aszinkron kivételekkel és a holtpontokkal. Szerencsére ritkán kell velük közvetlenül foglalkoznunk, mert:

- az összes, *IO*-t nem tartalmazó Haskell-kód alapból biztonságos;
- rendelkezésre állnak már a megfelelő absztrakciók, például az *MVar* esetén a *modifyMVar*;
- az állapottal dolgozó, megszakításra érzékeny, nagyobb kódrészletek mindig becsomagolhatóak a *mask* segítségével;
- ha ügyelünk több *MVar* használata esetén a sorrendre, akkor a program működni fog.

Vagy...