IMPLEMENTAREA CONCURENTEI IN LIMBAJE DE PROGRAMARE

STM

Santa Claus problem

Ioana Leustean

Simon Peyton Jones, Beautiful Concurrency

➢ Santa Claus problem

"Santa repeatedly sleeps until wakened by either all of his nine reindeer, back from their holidays, or by a group of three of his ten elves. If awakened by the reindeer, he harnesses each of them to his sleigh, delivers toys with them and finally unharnesses them (allowing them to go off on holiday). If awakened by a group of elves, he shows each of the group into his study, consults with them on toy R&D and finally shows them each out (allowing them to go back to work). Santa should give priority to the reindeer in the case that there is both a group of elves and a group of reindeer waiting."

S. Peyton Jones, Beautiful concurrency, in Beautiful Code,
Leading Programmers Explain How They Think, O'Reilly, 2007

Problema a fost initial formulata in
JA Trono. A new exercise in concurrency. *SIGCSE Bulletin*, 26:8–10, 1994

➢ Santa Claus problem

- Programul va avea 10 threaduri elf, 9 threaduri ren, threadul Santa .
- Threadurile elf/ren vor forma un grup de capacitate data.
- Cand s-a realizat o grupare aceasta este preluata de Santa
  - renii au prioritate,
  - santa trebuie sa fie liber.
- Toate threadurile functioneaza la infinit.
- Cand grupul s-a format, fiecare elf/ren din grup va intra la Santa, va desfasura o activitate si apoi va pleca .
- Santa va lasa sa intre un grup nou numai dupa ce toti elfii/renii din grupul anterior au plecat .

Formarea grupurilor si activitatile desfasurate cu Santa sunt concurente!

```haskell
main = do
    elf_group <- newGroup 3
    sequence_ [ elf elf_group n    | n <- [1..10] ]

    rein_group <- newGroup 9
    sequence_ [ reindeer rein_group n | n <- [1..9] ]

    forever (santa elf_group rein_group)
```

```
*Main> main
----------
Ho! Ho! Ho! let's deliver toys
Reindeer 1 delivering toys

Reindeer 2 delivering toys

Reindeer 3 delivering toys

Reindeer 4 delivering toys

Reindeer 5 delivering toys

Reindeer 6 delivering toys

Reindeer 7 delivering toys

Reindeer 8 delivering toys

Reindeer 9 delivering toys

----------
Ho! Ho! Ho! let's meet in my study
Elf 1 meeting in the study

Elf 2 meeting in the study

Elf 3 meeting in the study
```

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

- Thread ren/elf

Ciclul de viata al unui elf:

1. incearca sa intre intr-un grup
2. dupa ce grupul s-a format intra la Santa
3. lucreaza cu Santa
4. pleaca de la Santa
5. se intoarce la 1.

Ciclul de viata al unui ren:

1. incearca sa intre intr-un grup
2. dupa ce grupul s-a format intra la Santa
3. lucreaza cu Santa
4. pleaca de la Santa
5. se intoarce la 1.

Cand un grup de reni/elfi este format, primeste doua porti (gates), una pentru intrare, alta pentru iesire.
Fiecare membru al grupului
intra la Santa prin poarta de intrare a grupului sau si
iese de la Santa prin poarta de iesire a grupului sau

➢ Thread ren/elf

```
helper1 :: Group -> IO () -> IO ()
helper1 group do_task = do
    (in_gate, out_gate) <- joinGroup group
    passGate in_gate
    do_task
    passGate out_gate
```

Ciclul de viata al unui elf/ren:

1. incearca sa intre intr-un grup
2. dupa ce grupul s-a format intra la Santa
3. lucreaza cu Santa
4. pleaca de la Santa
5. se intoarce la 1.

```
elf1, reindeer1 :: Group -> Int -> IO ()

elf1 gp id = helper1 gp (meetInStudy id)
reindeer1 gp id = helper1 gp (deliverToys id)
```

elf1/reindeer1
definesc ciclul de viata al unui singur elf/ren a carui identitate este data de id

➢ Thread ren/elf

```
helper1 :: Group -> IO () -> IO ()
helper1 group do_task = do
    (in_gate, out_gate) <- joinGroup group
    passGate in_gate
    do_task
    passGate out_gate
```

Ciclul de viata al unui elf/ren:

1. incearca sa intre intr-un grup
2. dupa ce grupul s-a format intra la Santa
3. lucreaza cu Santa
4. pleaca de la Santa
5. se intoarce la 1.

```
elf1, reindeer1 :: Group -> Int -> IO ()

elf1 gp id = helper1 gp (meetInStudy id)
reindeer1 gp id = helper1 gp (deliverToys id)
```

identitatea/nr thread-ului ren/elf

```
meetInStudy :: Int -> IO ()
meetInStudy id = putStr ("Elf " ++ show id ++
"meeting in the study\n")

deliverToys :: Int -> IO ()
deliverToys id = putStr ("Reindeer " ++ show
id ++ " delivering toys\n")
```

➢ Thread ren/elf

Cand un grup de reni/elfi este format, primeste doua porti (gates), una pentru intrare, alta pentru iesire.
Fiecare membru al grupului
intra la Santa prin poarta de intrare a grupului sau si
iese de la Santa prin poarta de iesire a grupului sau

```
joinGroup :: Group -> IO (Gate, Gate)
joinGroup group = do
        putStr "joinGroup\n"
        return (MkGate, MkGate)
```

```
passGate :: Gate -> IO ()
passGate gate = putStr "passGate\n"
```

Atentie! Folosim aceste variante numai pentru testare.
Functiile joinGroup si passGate vor contoriza gruparea/accesul numarului fixat de reni/elfi.

```
elf1, reindeer1 :: Group -> Int -> IO ()
elf1 gp id = helper1 gp (meetInStudy id)
reindeer1 gp id = helper1 gp (deliverToys id)
```

Group si Gate
vor fi definite detaliat!
acesta variant este numai
pentru a testa elf1/reindeer1

```
data Group = MkGroup
data Gate = MkGate

main = do
    elf1 MkGroup 3
    reindeer1 MkGroup 4
```

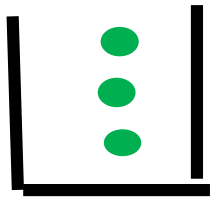
```
*Main> main
joinGroup
passGate
Elf 3 meeting in the study
passGate
joinGroup
passGate
Reindeer 4 delivering toys
passGate
```

```
main = do
      sequence_ [ elf1 MkGroup n | n <- [1..10] ]
      sequence_ [ reindeer1 MkGroup n | n <- [1..9] ]
```

➢ Grupuri si porti

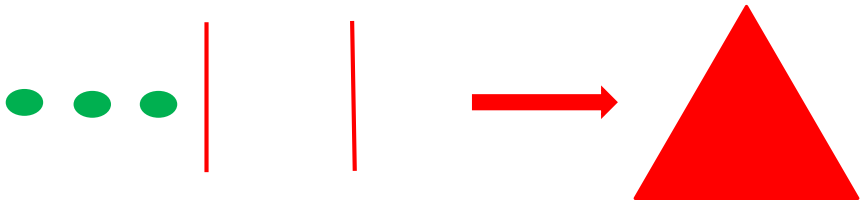
```
joinGroup :: Group -> IO (Gate, Gate)
joinGroup group = do
        putStr "joinGroup\n"
        return (MkGate, MkGate)
```

Grup



capacitatea unui grup
este limitata
Ex: 3 elfi

cand grupul se umple se creaza doua porti prin care
membrii grupului vor intra/iesi la Santa



portile sunt operate de Santa

Fiecare grup are propriile porti.

Astfel, un grup poate fi la Santa
in timp ce alt grup este in curs de formare.

➢ Porti

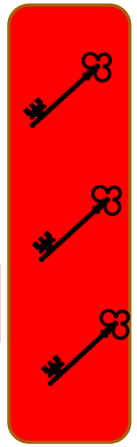
```
data Gate = MkGate Int (TVar Int)
```

nr. maxim de chei

nr. chei disponibile

```
newGate :: Int -> STM Gate
newGate n = do
   tv <- newTVar 0    -- !!!!
   return (MkGate n tv)
```

!!!! Cheile vor fi date de Santa



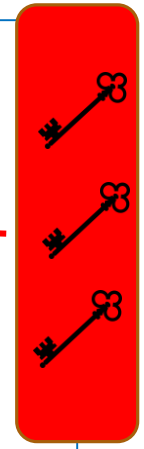
```
passGate :: Gate -> IO ()
passGate :: Gate -> IO ()
passGate (MkGate n tv)  = atomically $ do
                n_left <- readTVar tv
                if (n_left == 0)
                 then retry
                 else writeTVar tv (n_left-1)
```

un ren/elf care apeleaza
passGate
ia o cheie pentru a intra/iesi de la Santa

- Porti

```
data Gate = MkGate Int (TVar Int)
```

```
operateGate :: Gate -> IO ()
operateGate (MkGate n tv) = do
   atomically (writeTVar tv n)
   atomically $ do
                    n_left <- readTVar tv
                    if (n_left > 0)
                        then retry
                        else return ()
```

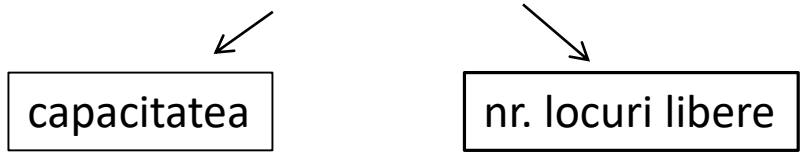


cheile sunt date de Santa

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

- Grupuri

```
data Group = MkGroup Int (TVar (Int, Gate, Gate))
```




```
data Gate = MkGate Int (TVar Int)
```

```
joinGroup :: Group -> IO (Gate, Gate)
joinGroup (MkGroup n tv) = atomically $ do
                (n_left, g1, g2) <- readTVar tv
                if (n_left == 0)
                 then retry
                 else do
                      writeTVar tv (n_left-1, g1, g2)
                      return (g1,g2)
```

```
newGroup   :: Int -> IO Group
newGroup n = atomically $ do
                      g1 <- newGate n
                      g2 <- newGate n
                      tv <- newTVar (n, g1, g2)
                      return (MkGroup n tv)
```

- Santa controleaza
  - formarea grupurilor: awaitGroup
  - accesul la porti: operateGate

```
data Group = MkGroup Int (TVar (Int, Gate, Gate))
data Gate = MkGate Int (TVar Int)
```

```
awaitGroup :: Group -> STM (Gate,Gate)
awaitGroup (MkGroup n tv) = do
   (n_left, g1, g2) <- readTVar tv
   if (n_left > 0)
    then retry
    else do
         new_g1 <- newGate n
         new_g2 <- newGate n
         writeTVar tv (n,new_g1,new_g2)
         return (g1,g2)
```

Testarea implementarii grupurilor si portilor

```
main = do
    grp <- newGroup 2       -- grup de capacitate 2
    forkIO $ elf1 grp 1     --  elful 1 vrea sa intre la Santa
    forkIO $ elf1 grp 2     --  elful 2 vrea sa intre la Santa
    (in_gate, out_gate) <- atomically (awaitGroup grp)   -- Santa asteapta formarea grupului
    operateGate in_gate     --  Santa deschide poarta de intrare si asteapta sa intre toti elfii
    operateGate out_gate    -- Santa deschide poarta de iesire si asteapta sa intre toti elfii
```

```
*Main> main
Elf 1 meeting in the study

Elf 2 meeting in the study
```

➢ **Threadul** elf/ren

Ciclul de viata al unui elf/ren:
1. incearca sa intre intr-un grup
2. dupa ce grupul s-a format intra la Santa
3. lucreaza cu Santa
4. pleaca de la Santa
5. se intoarce la 1.

```
import System.Random
import Control.Monad
randomDelay :: IO ()
randomDelay = do
   waitTime <- getStdRandom (randomR (1, 5000000))
   threadDelay waitTime
```

```
helper1 :: Group -> IO () -> IO ()
helper1 group do_task = do
   (in_gate, out_gate) <- joinGroup group
   passGate in_gate
   do_task
   passGate out_gate

elf1, reindeer1 :: Group -> Int -> IO ()
elf1 gp id = helper1 gp (meetInStudy id)
reindeer1 gp id = helper1 gp (deliverToys id)
```

```
elf, reindeer :: Group -> Int -> IO ThreadID

elf gp id  = (forkIO . forever) $ do
                    elf1 gp id
                    randomDelay

reindeer gp id  = (forkIO . forever) $ do
                    reindeer1 gp id
                    randomDelay
```

Fiecare elf/ren se executa intr-un thread separat

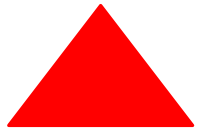
```
elf gp id  = (forkIO . forever) $ do
                elf1 gp id
                randomDelay
```

```
reindeer gp id  = (forkIO . forever) $ do
                    reindeer1 gp id
                    randomDelay
```

```
main = do
   elf_group <- newGroup 3
   sequence_ [ elf elf_group n | n <- [1..10] ]

   rein_group <- newGroup 9
   sequence_ [ reindeer rein_group n | n <- [1..9] ]

   forever (santa elf_group rein_group)
```

➢ Santa 

```
chooseGroup :: Group -> String -> STM (String, (Gate,Gate))
chooseGroup gp task = do
                      gates <- awaitGroup gp
                      return (task, gates)
```

```
santa :: Group -> Group -> IO ()
santa elf_gp rein_gp = do
   putStr "----------\n"
   (task, (in_gate, out_gate)) <- atomically  $ orElse
                                                (chooseGroup rein_gp "deliver toys") --!!!
                                                (chooseGroup elf_gp "meet in my study")

   putStr ("Ho! Ho! Ho! let's " ++ task ++ "\n")
   operateGate in_gate
                                  -- elfii/renii lucreaza cu Santa
   operateGate out_gate
```

!!! Renii au prioritate

```
main = do
   stdw  <- newMVar ()
   elf_group <- newGroup 3
   sequence_ [ elf elf_group n stdw | n <- [1..10] ]

   rein_group <- newGroup 9
   sequence_ [ reindeer rein_group n stdw | n <- [1..9] ]

   forever (santa elf_group rein_group)
```

```
*Main> main
----------
Ho! Ho! Ho! let's deliver toys
Reindeer 1 delivering toys

Reindeer 2 delivering toys

Reindeer 3 delivering toys

Reindeer 4 delivering toys

Reindeer 5 delivering toys

Reindeer 6 delivering toys

Reindeer 7 delivering toys

Reindeer 8 delivering toys

Reindeer 9 delivering toys

----------
Ho! Ho! Ho! let's meet in my study
Elf 1 meeting in the study

Elf 2 meeting in the study

Elf 3 meeting in the study
```