



SOFTWARE · PRODUCTIVITY · GROUP



Memória Transacional de Software



UNIVERSIDADE FEDERAL
DE PERNAMBUCO

cin.ufpe.br

Memória transacional de software (STM)

- Regiões da memória usadas dentro de **transações**
- **Alternativa** ao uso de exclusão mútua
 - Lembrando: **MVars** criam **zonas de exclusão mútua**

Memória transacional de software (STM)

- Regiões da memória usadas dentro de **transações**
- **Alternativa** ao uso de exclusão mútua
 - Lembrando: **MVars** criam **zonas de exclusão mútua**
- Ainda é um tópico de pesquisa
 - Mas Haskell tem uma implementação **robusta**
- Memória transacional também pode ser implementada em **hardware**

Por que memória transacional?

- Menos complicação
 - Travas são muito difíceis de gerenciar
 - Com STM, o *runtime* é quem trabalha
- Não há bloqueio
 - Logo, não há *deadlocks*
 - Quando só exclusão mútua é necessária
- **É novidade! :)**
 - Mas já está começando a se tornar prática



IBM Sequoia (Lawrence Livermore National Laboratory)

3o computador mais rápido do mundo (jun/2013 – <http://www.top500.org/lists/2013/06/>)

Usa memória transacional de hardware na *cache* L2

<http://img.gawkerassets.com/img/17q5vj0579zd9jpg/original.jpg>

Mais sobre STM

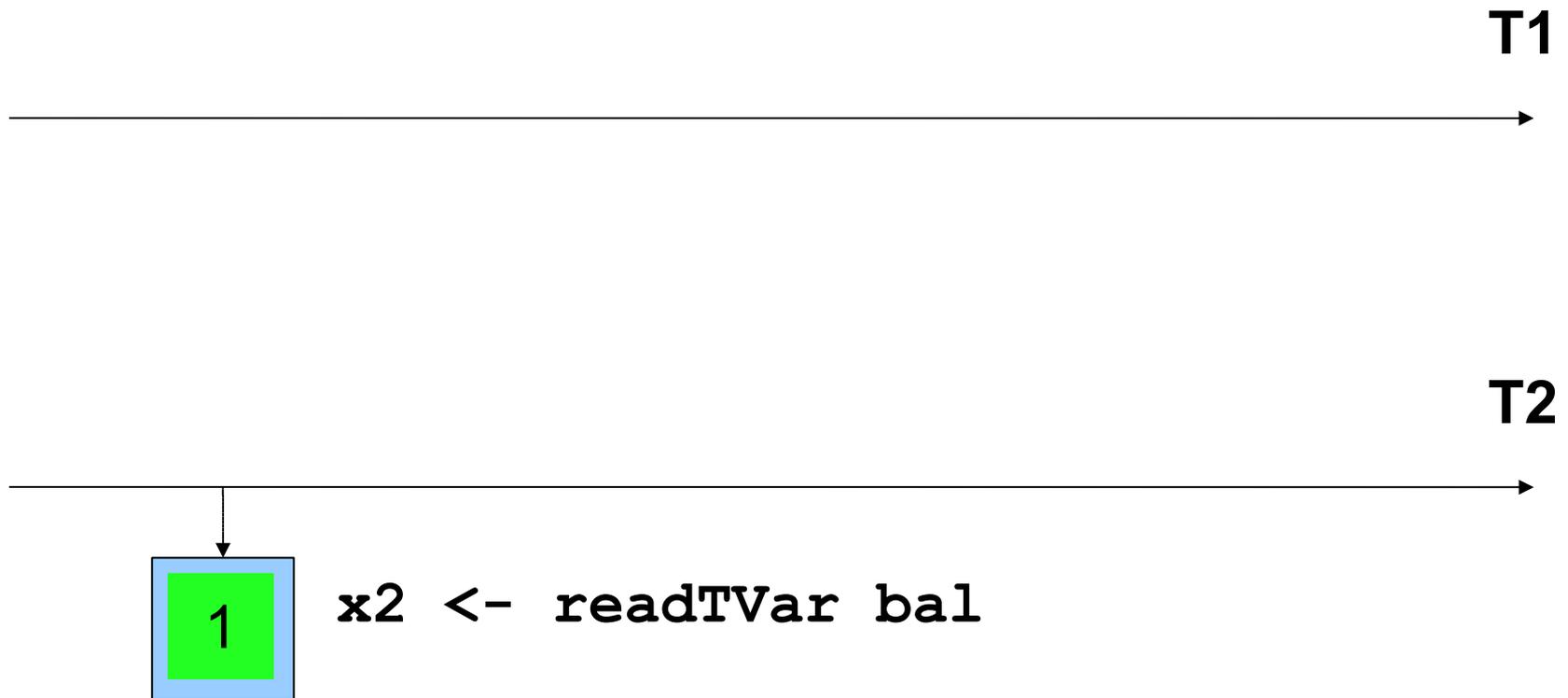
- Pacote `Control.Concurrent.STM`
- STM => Atomicidade + Isolamento
- Transações são definidas pela monad **STM**
 - Similar a **IO**
 - `atomically :: STM a -> IO a`
- Uma transação **não pode** realizar operações de entrada e saída
 - E **comunicação** entre *threads* precisa ser controlada

Exemplo simplíssimo

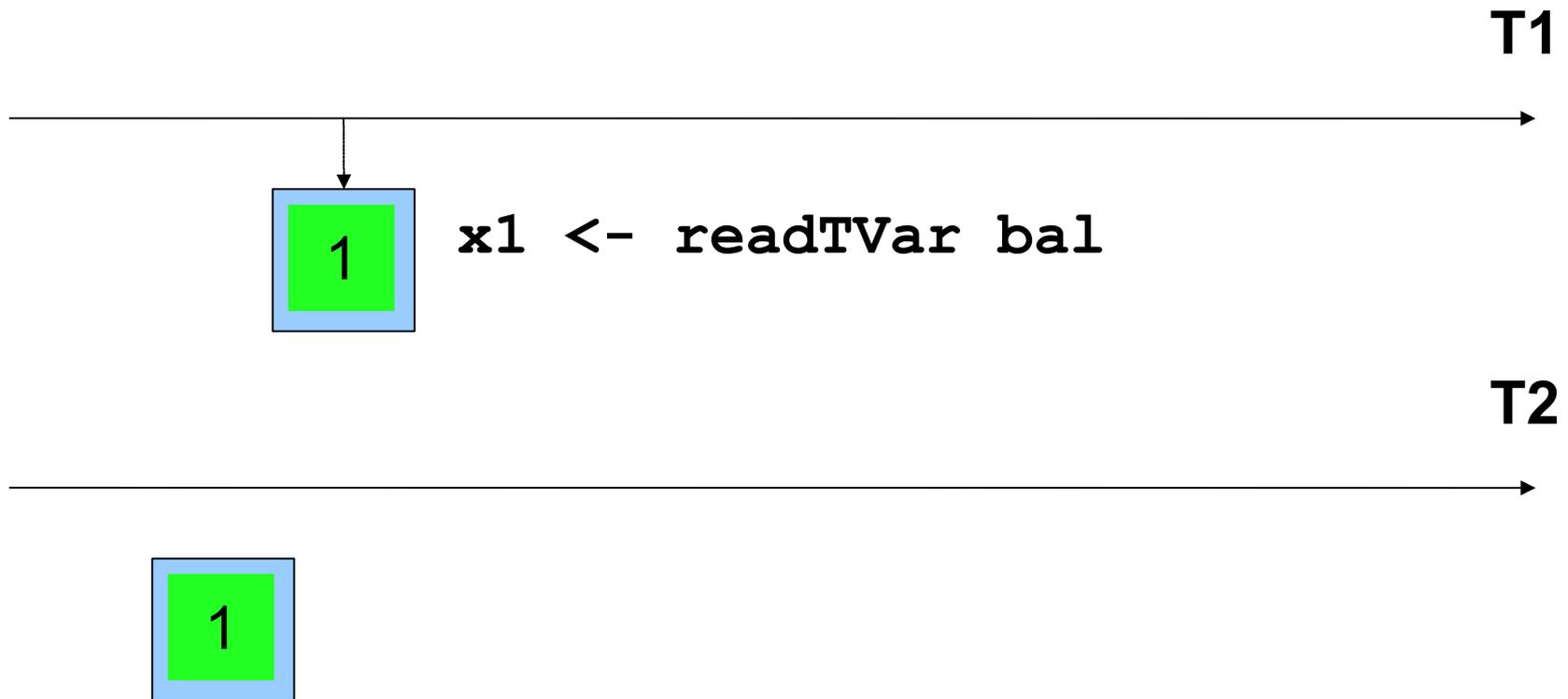
```
-- thread T1
atomically (do x1 <- readTVar bal
               writeTVar bal (x1+10)
            )
-- thread T2
atomically (do x2 <- readTVar bal
               writeTVar bal (x2+2)
            )
```

- Variáveis compartilhadas: **TVars**
 - Usadas apenas dentro de **blocos atômicos**
 - E se as duas *threads* tentarem escrever em **bal**?

STM em ação

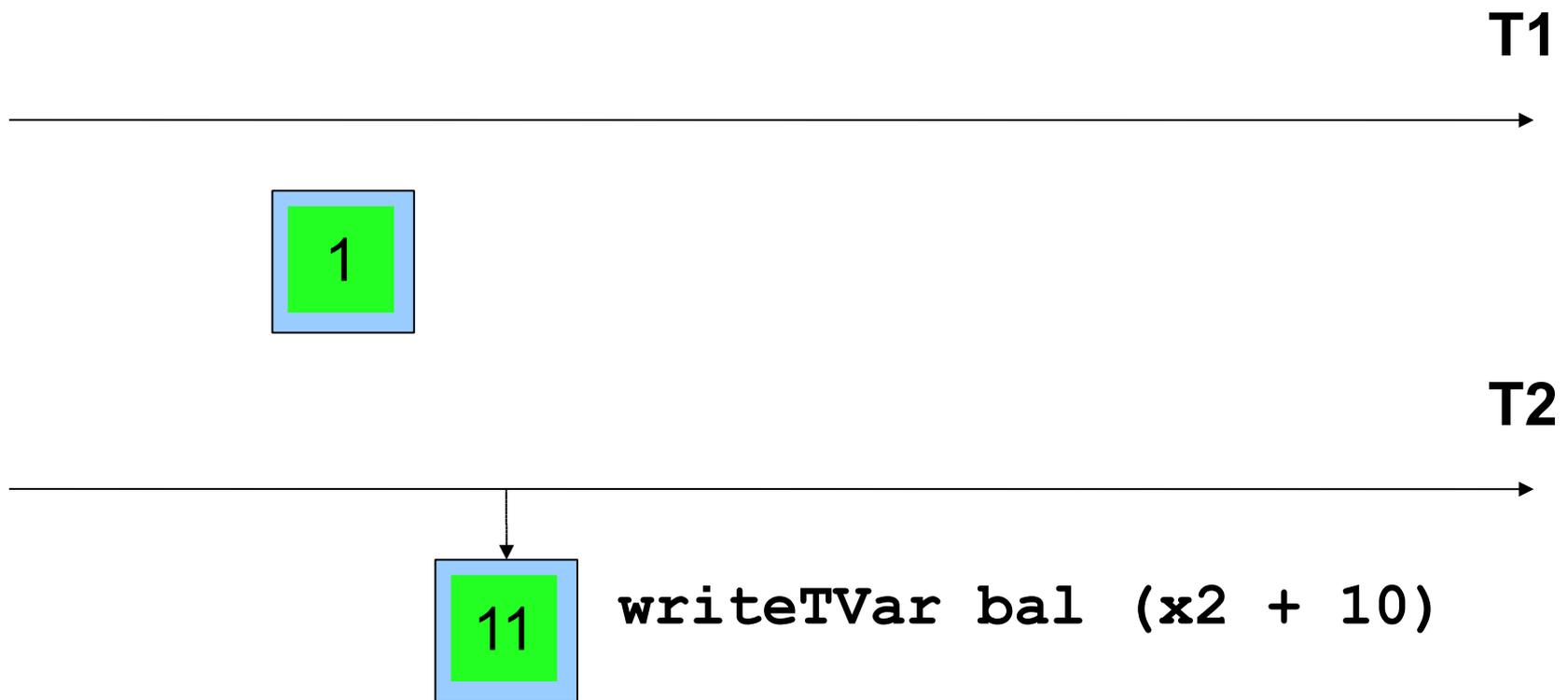


STM em ação

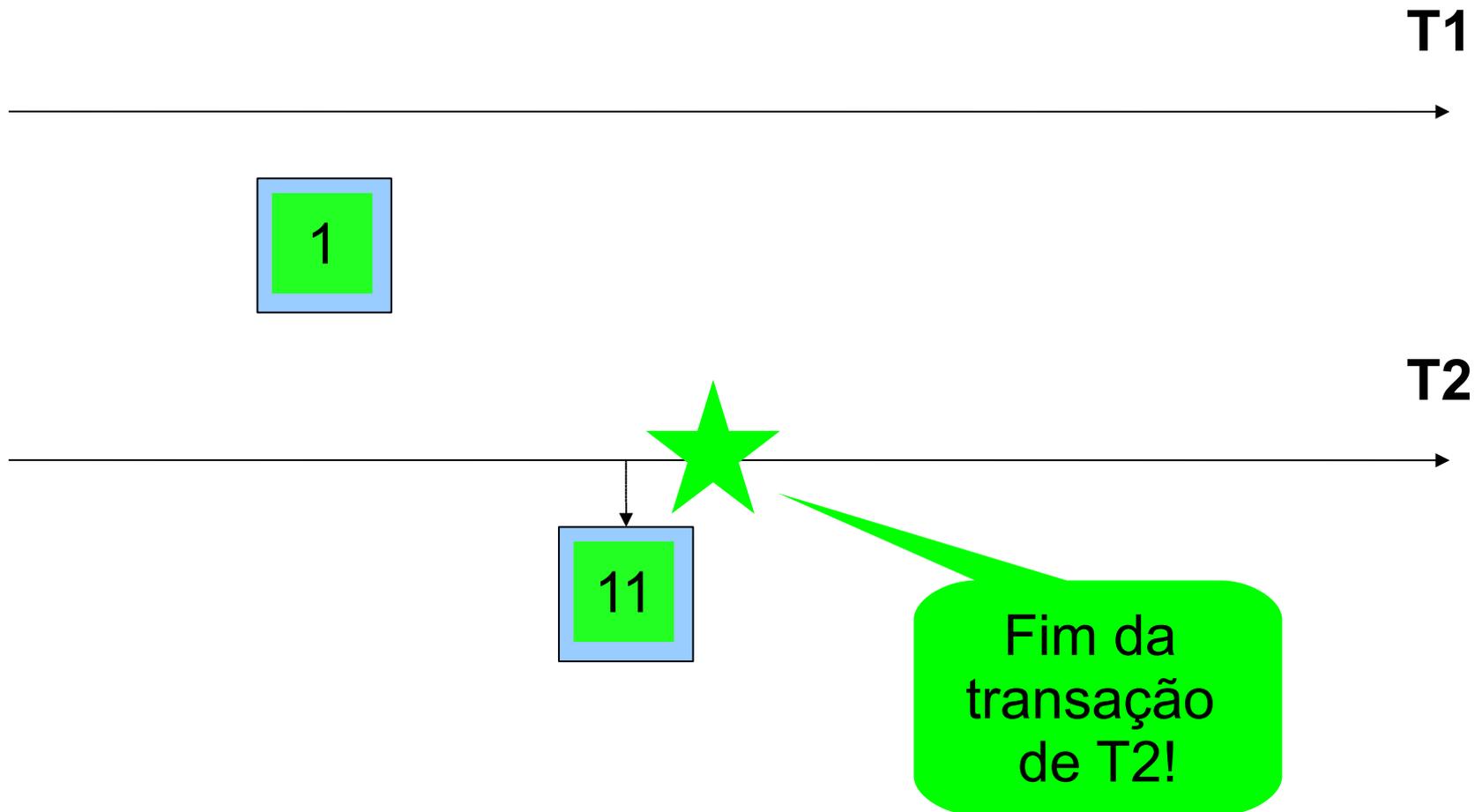


Cada transação mantém uma cópia da variável!

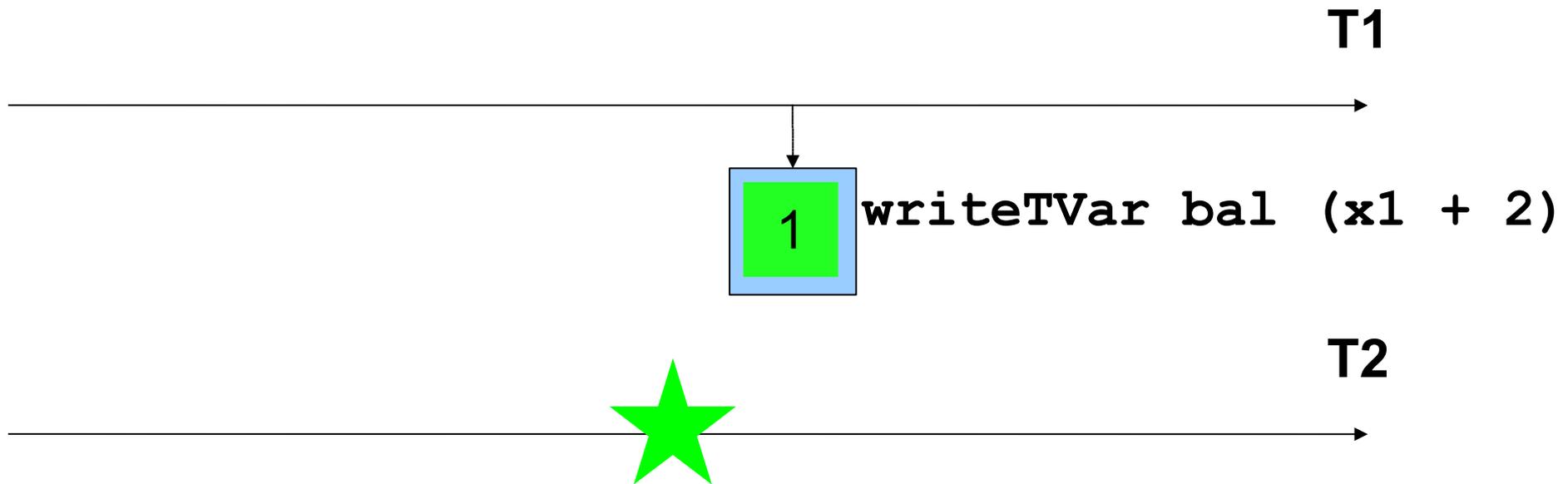
STM em ação



STM em ação

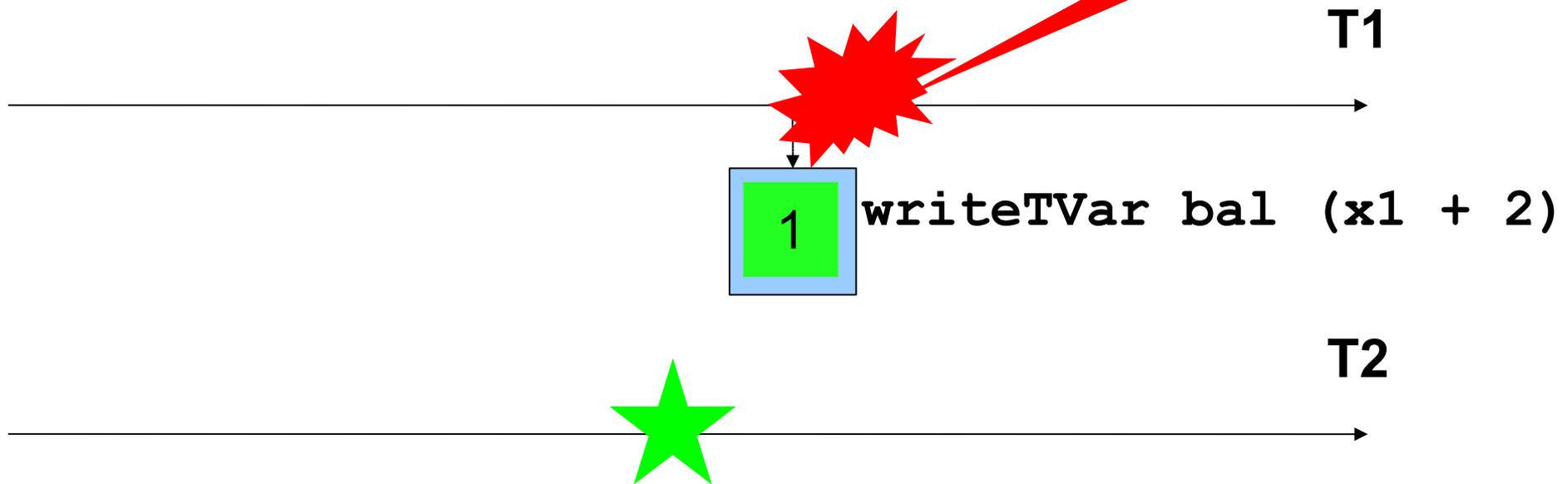


STM em ação



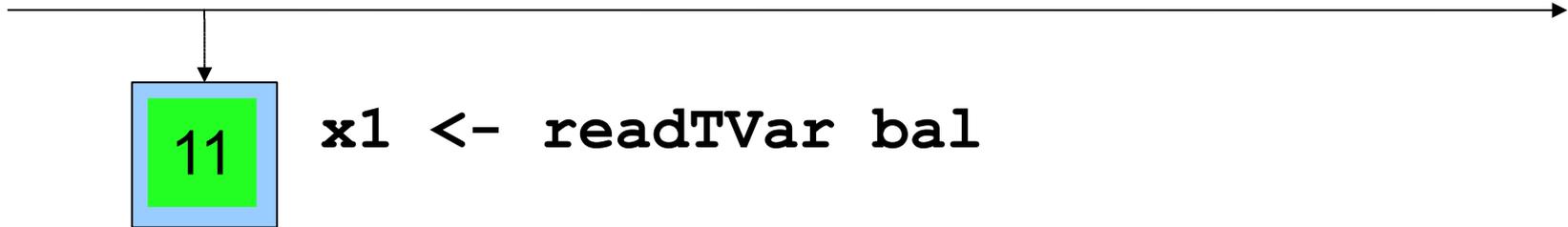
STM em ação

Transação de T1 é abortada!



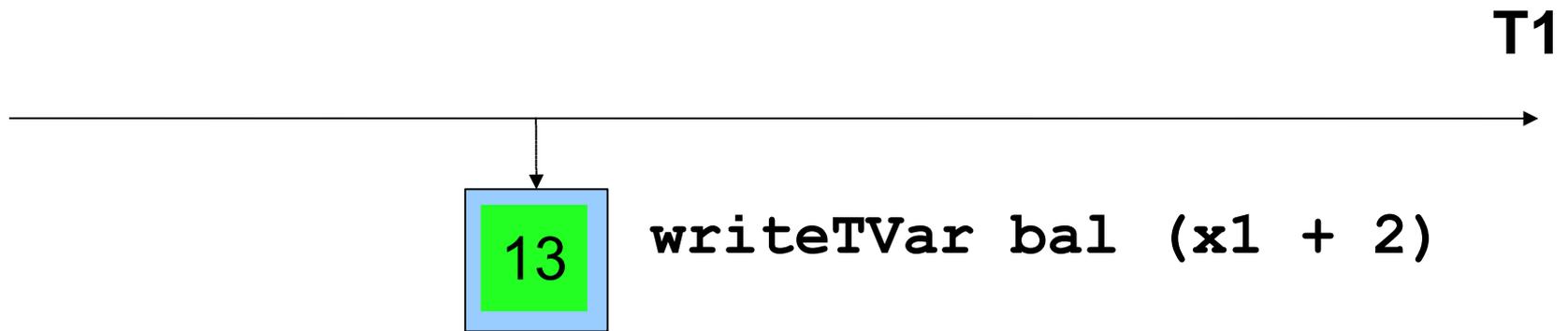
STM em ação

T1

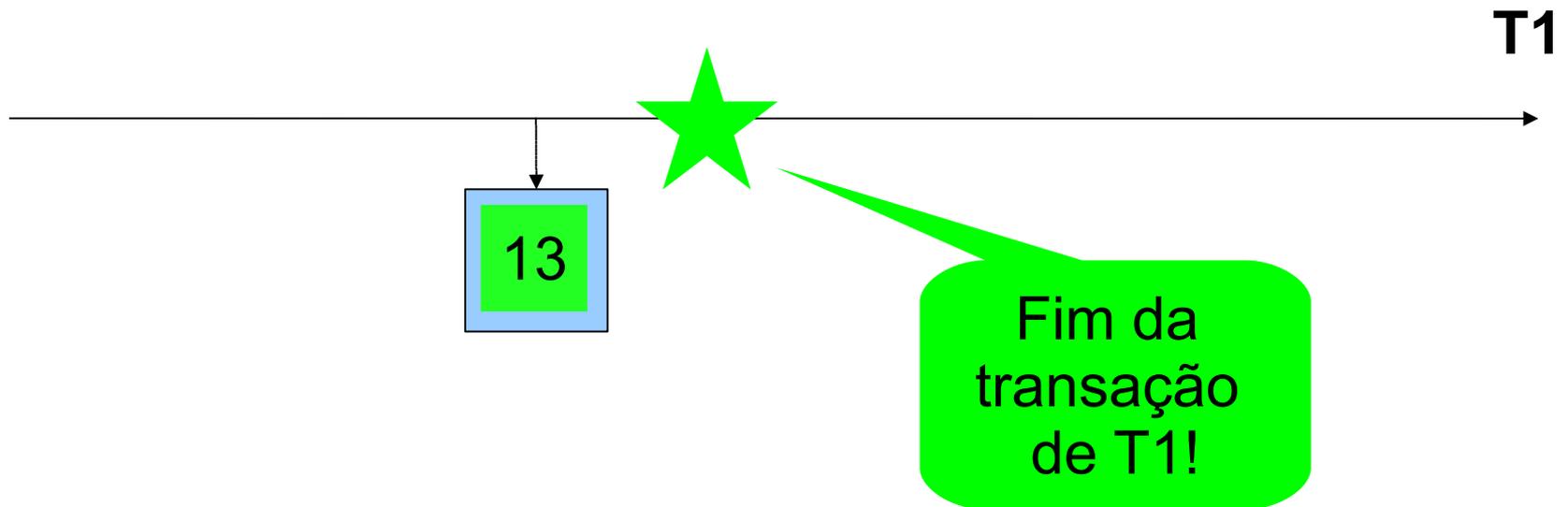


A transação
é então
reiniciada!

STM em ação



STM em ação



Algumas definições importantes

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

Contador transaccional

```
import Control.Concurrent.STM
main :: IO ()
main = do contador <- atomically (newTVar 0)
        fim <- newMVar 2
        forkIO (oper (+) contador fim 100000)
        forkIO (oper (-) contador fim 100000)
        waitThreads fim
        v <- atomically (readTVar contador)
        putStr (show v)

waitThreads :: MVar Int -> IO ()
waitThreads fim =
  do f <- takeMVar fim
     if (f > 0)
     then do { putMVar fim f; waitThreads fim }
     else return ()
```

Contador transaccional

```
oper :: (Int->Int->Int) -> TVar Int ->
        MVar Int -> Int -> IO()
oper op cont fim 0
  = do f <- takeMVar fim
        putMVar fim (f-1)
oper op cont fim num
  = do v <- atomically (readTVar cont)
        atomically (writeTVar cont (op v 1))
        oper op cont fim (num-1)
```

Código rodando...

Contador transacional ERRADO

```
oper :: (Int->Int->Int) -> TVar Int ->
      MVar Int -> Int -> IO()
oper op cont fim 0
  = do f <- takeMVar fim
      putMVar fim (f-1)
oper op cont fim num
  = do v <- atomically (readTVar cont)
      atomically (writeTVar cont (op v 1))
      oper op cont fim (num-1)
```

Leitura e escrita precisam estar **na mesma transação!!!**

Contador transaccional CORRIGIDO

```
oper :: (Int->Int->Int) -> TVar Int ->
      MVar Int -> Int -> IO()
oper op cont fim 0
  = do f <- takeMVar fim
      putMVar fim (f-1)
oper op cont fim num
  = do atomically (do v <- readTVar cont
                    writeTVar cont (op v 1)
                  )
      oper op cont fim (num-1)
```

Tentando de novo: função `retry`

- Tenta executar uma transação novamente, **do começo**
- Usada quando uma condição necessária não é satisfeita
 - **Bloqueia** a transação temporariamente
 - Tenta de novo quando os **TVars** envolvidos mudam

Um exemplo bancário

```
import ...
waitThreads :: MVar Int -> IO()
...
main::IO()
main = do
    bal <- atomically (newTVar 0.0)
    invest <- atomically (newTVar 0.0)
    fim <- newMVar 2
    forkIO (makeInvestment bal invest fim)
    forkIO (incrementBalance bal fim)
    waitThreads fim
    e <- atomically (readTVar bal)
    f <- atomically (readTVar invest)
    putStr ("Balance: " ++ show e ++ "\n")
    putStr ("Investment: " ++ show f ++ "\n")
    return ()
```

Um exemplo bancário (cont.)

```
incrementBalance :: TVar Float -> MVar Int -> IO ()
incrementBalance tvb fim = do
    atomically(ib tvb)
    f <- takeMVar fim
    putMVar fim (f-1)

ib :: TVar Float -> STM ()
ib tvb = do
    y <- readTVar tvb
    if y < 10000
    then do{ writeTVar tvb (y + 1);
             ib tvb;
           }
    else return ()
```

...

Um exemplo bancário (cont².)

```
makeInvestment :: TVar Float -> TVar Float -> MVar Int -> IO()
makeInvestment acc inv fim =
  do
    atomically(
      do
        a <- readTVar acc
        if a < 1000
        then retry --condição não satisfeita
        else do {
          writeTVar inv 1000;
          writeTVar acc (a - 1000);
        }
    )
  f <- takeMVar fim
  putMVar fim (f-1)
```

Rodando o exemplo bancário

- Saída:
 - Balance: 9000.0
 - Investment: 1000.0
 - Esperada?
- **ib** roda sempre na **mesma transação**
 - Em que isso afeta **makeInvestment**?

Um outro exemplo bancário

```
import Control.Concurrent.STM
Import Control.Concurrent

main :: IO()
main = do
    bal <- atomically (newTVar 0.0)
    invest <- atomically (newTVar 0.0)
    fim <- newEmptyMVar
    forkIO (incrementBalance bal fim)
    forkIO (makeInvestment bal invest)
    dummy <- takeMVar fim
    e <- atomically (readTVar bal)
    f <- atomically (readTVar invest)
    putStr ("Balance: " ++ show e ++ "\n")
    putStr ("Investment: " ++ show f ++ "\n")
    return ()
```

Um outro exemplo bancário (cont.)

```
incrementBalance :: TVar Float -> MVar Int -> IO ()
incrementBalance tvb fim = do
    atomically(ib tvb)
    x <- atomically(readTVar tvb)
    if (x < 10000)
    then incrementBalance tvb fim
    else putMVar fim 0

-- ib é uma transação menor
ib :: TVar Float -> STM ()
ib tvb = do
    y <- readTVar tvb
    writeTVar tvb (y + 1)
```

Um outro exemplo bancário (cont².)

```
makeInvestment :: TVar Float -> TVar Float -> IO ()
makeInvestment acc inv =
  do {
    atomically(
      do { a <- readTVar acc;
          if a < 1000 then retry
            else do { writeTVar inv 1000;
                     c <- readTVar acc;
                     writeTVar acc (c - 1000);
                   };
        }
    )
    f <- takeMVar fim
    putMVar fim (f-1); }
```

- Saída:

- Balance: 10000.0
- Investment: 1000.0