Problema dos mergulhadores

Felipe Gomes Lacerda
06/83884

Pedro Garcia Freitas
04/35597

14 de dezembro de 2009

1 O problema

5 mergulhadores estão submersos a uma profundidade de 220 metros. Para eles conseguirem respirar a esta profundidade e realizar algum trabalho, precisam equilibrar a pressão interna do seu traje com a pressão externa da água. Para haver esse equilíbrio, precisam utilizar 2 balões de oxigênio: um para respiração e outro para controle de pressão. Contudo, cada metro submerso equivale a aumentar em 10 vezes o peso de um corpo em relação à pressão atmosférica. Sendo assim, cada mergulhador só pode submergir com 1 balão de oxigênio. Como possibilitar que eles compartilhem seus balões de oxigênio para que possam trabalhar, sendo que a roupa armazena e suporta oxigênio suficiente por um período de tempo, desde que os mergulhadores não realizem trabalho?

2 Solução 1: STM

A primeira solução ao problema foi feita na linguagem Haskell, usando *Software Transactional Memory* [1] (STM). STM é um novo modelo para programação concorrente, cuja filosofia é baseada em transações bancárias. Em particular, esse modelo tem uma visão *otimista* de seções críticas: várias threads podem executar uma seção crítica ao mesmo tempo. Mas em vez de as escritas serem feitas nas variáveis compartilhadas diretamente, cada thread escreve em um log as mudanças a serem feitas. Quando a seção crítica (também chamada de *transação*) termina, o log é validado: o sistema verifica se os valores das variáveis lidas—os valores são armazenados no log—são iguais aos verdadeiros valores das variáveis. Se forem iguais, as alterações do log são gravadas nas variáveis. Se forem diferentes, é porque outras threads alteraram as variáveis—a transação tinha uma visão incosistente da memória. Nesse caso, o log e a transação são reiniciados.

Esse modelo tem algumas vantagens em relação ao usual, que usa mutexes—por exemplo, nunca precisamos nos preocupar em tomar o número certo de mutexes, ou o local e a ordem em que eles são colocados. Só temos que nos preocupar em como delimitar as transações. Isso elimina uma grande classe de possíveis deadlocks. Ainda, esse modelo fornece mais concorrência que mutexes, já que várias threads podem executar uma transação ao mesmo tempo, desde que a consistência seja garantida.

O problema com o modelo é que ele é difícil de implementar nas linguagens mais populares. Variáveis usadas em transações, também chamadas de variáveis transacionais, devem ser manipuladas de forma diferente das não-transacionais, e a linguagem deve impedir que uma variável transacional seja modificada fora de uma transação. Ainda, ela deve controlar as operações que ocorrem dentro de uma transação—por exemplo, operações de entrada e saída não podem ser revertidas. É difícil implementar essas restrições em linguagens como Java e C++, mas elas são viáveis em Haskell. Previsivelmente, Haskell foi a primeira linguagem a implementar STM.

Como, então, usamos STM para resolver o problema? Primeiro, vamos modelar os balões de oxigênio. Um balão será representado por um semáforo binário, que assume os valores `False` (que indica que o balão está sendo usado) e `True` (balão livre).

```
1 type Bottle = TVar Bool
```

Essa declaração indica que um balão é uma variável transacional (`TVar`) do tipo booleano.

```
1  newBottle :: IO Bottle
2  newBottle = newTVarIO True
3
4  get :: Bottle -> STM ()
5  get bottle = do b <- readTVar bottle
6                  -- se b não estiver livre, reinicia a transação
7                  check b
8                  writeTVar bottle False
9
10 free :: Bottle -> STM ()
11 free bottle = writeTVar bottle True
```

A função `newBottle` cria um balão, inicialmente livre. A função `take` tenta adquirir um balão. Se ele já foi tomado, *toda a transação é reiniciada*. `free` simplesmente libera o balão.

Um problema pode ocorrer se a thread reiniciar a transação imediatamente após verificar que o balão está tomado. Se ela fizer isso, é bem provável que o balão ainda esteja tomado na segunda tentativa. Em geral, várias tentativas são necessárias para conseguir o balão. Assim, muito tempo de processamento pode ser perdido nessas novas tentativas. Mas como as threads mantêm logs de variáveis utilizadas durante a transação, uma otimização pode ser feita: a thread só reinicia a transação quando uma das variáveis for modificada.

Antes de partir para a solução do problema, falta mais um detalhe. Como apontado anteriormente, operações de entrada e saída não podem ser revertidas, então não faz sentido fazer I/O dentro de uma transação. Como, então, imprimimos o estado de cada mergulhador? Uma forma é escrever o estado em uma variável transacional (um buffer) e então imprimir esse buffer fora da transação.

```
1 type Buffer = TVar (Array Int String)
2
3 newBuffer :: Int -> IO Buffer
4 newBuffer n =
5     let init_list = [(i,(show i)++"") | i <- [0..n-1]] in
6     newTVarIO (array (0,n-1) init_list)
7
```

```
writeState buffer item n = do arr <- readTVar buffer
                              let a = arr // [(n,item)]
                              writeTVar buffer a

readState buffer n = do arr <- readTVar buffer
                        return (arr ! n)

output buffer n m
    | n > m = do putStr "\n"
                 threadDelay 500000
                 output buffer 0 m
    | otherwise = do str <- atomically (readState buffer n)
                     putStr str
                     output buffer (n+1) m
```

`newBuffer` cria uma array para armazenar o estado de cada mergulhador. A array em si é uma variável transacional, para permitir a atualização dos estados durante uma transação. `writeState` substitui a array por outra que tem o estado do `n`-ésimo mergulhador atualizado. `readState` lê o estado do `n`-ésimo mergulhador (isso é necessário para imprimir os estados). `output` é um loop infinito: a cada meio segundo, a função imprime o estado de todos os mergulhadores. Os estados em geral serão diferentes a cada iteração, porque cada mergulhador vai modificar o buffer com o seu estado atual.

Finalmente, a função que representa um mergulhador.

```
diver :: Int -> Buffer -> Bottle -> Bottle -> IO ()
diver i buffer bottle1 bottle2 =
    -- começa sem respirar
    do atomically (writeState buffer (show i ++ " sem respirar\t") i)

       -- tenta adquirir os dois balões
       atomically (do
         get bottle1
         get bottle2)

       -- respira por um tempo aleatório
       atomically (writeState buffer (show i ++ " respirando\t") i)
       randomDelay

       -- libera os balões
       atomically (do
         free bottle1
         free bottle2)

       -- tempo 'ocioso' (o mergulhador não precisa de oxigênio)
       atomically (writeState buffer (show i ++ " ocioso\t") i)
       randomDelay

       -- repete o procedimento
       diver i buffer bottle1 bottle2
    where
      randomDelay = do r <- randomRIO(100000,500000)
                       threadDelay r
```

O mergulhador tenta tomar dois balões. Essa operação deve ser colocada em uma transação, ou então poderia acontecer o problema de todos os mergulhadores pegarem um balão e então entrarem em deadlock porque não há mais nenhum balão livre. Uma transação é delimitada pela função `atomically`. Note que a escrita dos estados e a liberação dos balões também estão em transações.

Por fim, a função para simular o problema para `n` mergulhadores.

```
simulate :: Int -> IO ()
simulate n = do bottles <- replicateM n newBottle
                outputBuffer <- newBuffer n
                mapM_ (action outputBuffer bottles) [0..n-1]
                output outputBuffer n (n-1)
    where
      action buffer bottles i =
          forkIO (diver i buffer
                  (bottles !! i)
                  (bottles !! ((i+1) `mod` n)))
```

Essa função cuida da criação dos balões, do buffer e dos mergulhadores em si, sendo que cada um executa numa thread diferente. Para criar threads, a função `forkIO` é usada. Cada mergulhador então toma o balão "da esquerda" e o "da direita". Note que a última chamada do código é para a função `output`, que é um loop infinito que imprime o estado de cada mergulhador a cada 0.5 segundos.

3 Solução 2: Mutex

A segunda solução do problema utiliza um mutex para sincronizar o acesso das threads à região crítica, além de manter um conjunto de semáforos para indicar o estado dos recursos.

No programa, cada mergulhador é um ente independente, implementado na seguinte classe:

```
class Diver(Thread):
        def __init__(self,i):
                Thread.__init__(self)
                self._i = i

        def run(self):
                self._diver(self._i)

        def _diver(self, i):
                """ Activities of ith diver (from 0 to N-1) """

                while True:
                        idle(i)
                        take_oxygen(i)
                        work(i)
                        put_oxygen(i)
```

No código acima, o construtor da classe `__init__` serve para criar uma thread nova no sistema. O argumento `i` passado ao construtor será uma variável inteira, que é a identificação da thread, representando o *i*-ésimo mergulhador.

O método `run` é utilizado pelo sistema para reconhecer qual serão as instruções que aquela thread deverá executar, ou seja, onde é implementada sua atividade. Note que este método chama a função privada `_diver`, utilizada para conter as atividades de cada mergulhador—equivalentes a cada função chamada em seu corpo (`idle`, `take_oxygen`, `work` e `put_oxygen`).

A primeira das funções chamadas, `idle`, apenas faz com que o mergulhador aguarde um tempo aleatório até tentar obter um recurso (balão de oxigênio):

```
def idle(i):
        showStatus()
        time.sleep( randint(TIME1,TIME2) )
```

A segunda linha do código acima contém a função `showStatus`, que apenas imprime de forma amigável para o usuário os estados de cada mergulhador. A terceira linha que contém a verdadeira funcionalidade de `idle`, que é colocar a thread para dormir por um tempo aleatório, definido no intervalo entre `TIME1` e `TIME2` (no código, definimos `TIME1` e `TIME2` como 2 e 4, respectivamente).

Passado o tempo em que a thread dorme, a próxima função que ela executará será `take_oxygen`. Conforme sugere o nome, ao executar esta função a thread que representa o mergulhador tentará tomar posse dos recursos envolvidos no compartilhamento, que estão encapsulados na seguinte classe:

```
class Global:
        mutex = Lock()
        s = [Lock()] * N
        state = [BREATHLESS] * N
```

Os objetos do tipo `Lock` são a forma que o Python fornece mutexes para sincronização das threads, sendo que eles podem assumir dois estados: bloqueados ou não-bloqueados. É a partir destes estados que a função `take_oxygen` irá decidir se deve obter o recurso ou não.

O atributo `s` é uma lista, onde cada elemento *i* equivale ao estado da *i*-ésima thread. Desta forma, antes de um mergulhador adquirir um novo balão de oxigênio, ele saberá se possui todos os recursos que precisa para começar a execução de seu trabalho.

A terceira variável da classe `Global` é uma lista que representa os possíveis estados do mergulhador—trabalhando, aguardando ou necessitando repor oxigênio. O mergulhador começa precisando repor oxigênio (`BREATHLESS`), mas ele também pode assumir os estados `WORKING` e `IDLE`.

Portanto, a função `take_oxygen` terá a seguinte forma:

```
1 def take_oxygen(i):
2         down(Global.mutex)
3         Global.state[i] = BREATHLESS
4         test(i)
5         up(Global.mutex)
6         if not Global.s[i].locked():
7                 down(Global.s[i])
```

A segunda linha desta função serve para indicar que a thread entrou na região crítica. Como esta operação é atômica, certamente outra thread não irá fazer o mesmo, garantindo assim que a atual poderá realizar as operações das linhas 3 e 4 sem que o estado das outras mude, não havendo condição de corrida.

Sendo assim, a thread atual muda seu estado, indicando sua intenção de obter o recurso. Para os mergulhadores, isto equivale ao fato que todos eles querem repor seu oxigênio o quanto antes, conforme pode ser visto na linha 3.

Ainda de posse da região crítica, na linha 4 o mergulhador testa para ver se existem os 2 balões de oxigênio necessários para que ele possa trabalhar e respirar. A função desta linha simplesmente verifica se os dois mergulhadores mais próximos não estão trabalhando, ou seja, se os balões de oxigênio compartilhados estão livres. Caso estejam, a função automaticamente os libera para que o mergulhador trabalhe e reponha sua respiração.

A linha 5, quando executa `up(Global.mutex)`, libera a região crítica, fazendo com que a thread atual não bloqueie mais o estado das demais.

Nas duas últimas linhas a thread verifica que o recurso não está sendo usado. Se ele não estiver, então ela o adquire para si.

Note que a execução da thread atual só continua após o término da função `test`. Ou seja, a função `work` apenas executa após o mergulhador estar de posse dos 2 recursos necessários. No código, esta função não realiza nenhuma tarefa real, sendo utilizada apenas para representar o período de tempo em que o mergulhador estivesse trabalhando.

Por fim, a última função chamada pela classe `Diver` é a `put_oxygen`. Esta função apenas libera os recursos alocados por `take_oxygen`:

```
1 def put_oxygen(i):
2         down(Global.mutex)
3         Global.state[i] = IDLE
4         test(LEFT(i))
5         test(RIGHT(i))
6         up(Global.mutex)
```

Conforme explicado para `take_oxygen`, as linhas 2 e 6 da função acima servem para garantir e liberar o acesso da thread atual na região crítica. A terceira linha indica que o mergulhador está ocioso por não estar com os balões necessários e as 2 linhas seguintes servem para transferir o recurso para os mergulhadores vizinhos.

4 Conclusão

STM é um mecanismo de controle de concorrência muito mais simples conceitualmente que a sincronização baseada em mutexes. Ele permite que o programador tenha um modelo mais claro de como as threads do programa interagem; em particular, deadlocks são muito menos frequentes. No caso do problema dos mergulhadores, o deadlock é evitado simplesmente colocando a operação de pegar os dois balões numa transação.

Por outro lado, STM é difícil de implementar em linguagens populares devido à natureza das transações: como elas devem ser revertidas frequentemente, não podemos executar operações que não podem ser desfeitas, como a maioria das operações de I/O. No problema dos mergulhadores, a solução para a questão de imprimir o estado na tela foi resolvido colocando os estados num buffer e depois imprimindo-os fora de uma transação.

A outra solução, baseada em mutexes, foi feita para demonstrar a solução do problema dos mergulhadores em um nível de abstração mais baixo. Desta forma, podemos simular o funcionamento de como seria a comunicação entre os elementos de hardware, se a distribuição dos recursos fosse controlada por um computador, por exemplo.

Referências

[1] `http://research.microsoft.com/en-us/um/people/simonpj/papers/stm/beautiful.pdf`

Apêndice A Implementação utilizando STM

```
{-
   DivingPhilosophers.hs

   Sincroniza 5 mergulhadores que precisam compartilhar 5 balões de
   oxigênio entre eles.

   Autor: Felipe Gomes Lacerda <fegolac@gmail.com>

   Compilar com ghc -o diving diving.hs --make -threaded
-}

module Main where

import Random
import Control.Monad
import Control.Concurrent
import Control.Concurrent.STM
import Data.Array

-- Balões implementados como semáforos binários usando STM
type Bottle = TVar Bool

newBottle :: IO Bottle
newBottle = newTVarIO True

get :: Bottle -> STM ()
get bottle = do b <- readTVar bottle
                -- se b não estiver livre, reinicia a transação
                check b
                writeTVar bottle False

free :: Bottle -> STM ()
free bottle = writeTVar bottle True

type Buffer = TVar (Array Int String)

newBuffer :: Int -> IO Buffer
newBuffer n =
    let init_list = [(i,"") | i <- [0..n-1]] in
    newTVarIO (array (0,n-1) init_list)

writeState buffer item n = do arr <- readTVar buffer
                              let a = arr // [(n,item)]
                              writeTVar buffer a

readState buffer n = do arr <- readTVar buffer
                        return (arr ! n)

output buffer n m
    | n > m = do putStr "\n"
                 threadDelay 1000000
                 output buffer 0 m
    | otherwise = do str <- atomically (readState buffer n)
                     putStr str
                     output buffer (n+1) m

diver :: Int -> Buffer -> Bottle -> Bottle -> IO ()
diver i buffer bottle1 bottle2 =
    -- começa sem respirar
    do atomically (writeState buffer (show i ++ " sem respirar\t") i)
```

```
        -- tenta adquirir os dois balões
        atomically (do
          get bottle1
          get bottle2)

        -- respira por um tempo aleatório
        atomically (writeState buffer (show i ++ " respirando\t") i)
        randomDelay

        -- libera os balões
        atomically (do
          free bottle1
          free bottle2)

        -- tempo 'ocioso' (o mergulhador não precisa de oxigênio)
        atomically (writeState buffer (show i ++ " ocioso\t") i)
        randomDelay

        -- repete o procedimento
        diver i buffer bottle1 bottle2
    where
      randomDelay = do r <- randomRIO(100000,500000)
                       threadDelay r

simulate :: Int -> IO ()
simulate n = do bottles <- replicateM n newBottle
                outputBuffer <- newBuffer n
                mapM_ (action outputBuffer bottles) [0..n-1]
                output outputBuffer n (n-1)
    where
      action buffer bottles i =
          forkIO (diver i buffer
                  (bottles !! i)
                  (bottles !! ((i+1) `mod` n)))

main = simulate 5
```

Apêndice B Implementação utilizando Python

```
# -*- coding: utf-8 -*-
#
# DivingPhilosophers.py
#
# Sincroniza 5 mergulhadores que precisam compartilhar 5 balões de
# oxigênio entre eles.
#
# Autor: Pedro Garcia Freitas <sawp@sawp.com.br>
# Baseado no algoritmo de Dijkstra para resolução do problema do
# jantar dos filósofos
#
# Licença: Domínio Público
#

from threading import Thread
from threading import Lock
from random import randint
import time
import sys
import signal


TIME1 = 2
TIME2 = 4
N = 5 # number of divers

IDLE = 0 # if diver is idle
BREATHLESS = 1 # if the diver must replace oxygen
WORKING = 2 # state when diver is using 2 oxygen resources

screen_row = [  10,  6,  6,  10, 14 ]
screen_col = [ 31, 36, 44, 49, 40 ]

LEFT = lambda i: (i+N-1)%N # code of ith left neighbor
RIGHT = lambda i: (i+1)%N # code of ith right neighbor
up = lambda a: a.release()
down = lambda a: a.acquire()

class Global:
        # semaphore to indicate mutual exclusion for critical regions
        mutex = Lock()
        # initialize the semaphore set of divers
        s = [Lock()] * N
        # diver's states (0 idle, 1 breathless, 2 working)
        state = [BREATHLESS] * N


class Diver(Thread):
        """ One class for Thread (Thread for class) """

        def __init__(self,i):
                Thread.__init__(self)
                self._i = i

        def run(self):
                self._diver(self._i)

        def _diver(self, i):
                """ Activities of ith diver (from 0 to N-1) """

                while True:
```

```
                        idle(i)
                        take_oxygen(i)
                        work(i)
                        put_oxygen(i)

def showStatus():
        def position(row, col):
                s_exit = "\033[%d;%dH" % (row, col)
                sys.stdout.write(s_exit)

        def position_flush(row,col):
                s_exit = "\033[%d;%dH\n" % (row,col)
                sys.stdout.write(s_exit)

        def draw_breathless(n):
                position( screen_row[n], screen_col[n] )
                sys.stdout.write("\033[34mB\033[0m")
                position_flush( 13, 1 )

        def draw_idle(n):
                position( screen_row[n], screen_col[n] )
                sys.stdout.write("\033[33mI\033[0m")
                position_flush( 13, 1 )

        def draw_working(n):
                position( screen_row[n], screen_col[n] )
                sys.stdout.write("\033[31mW\033[0m")
                position_flush(13,1)


        def draw(n):
                for i in range(0,N):
                        code = Global.state[i]

                        if code == IDLE:
                                draw_idle(i)
                        elif code == BREATHLESS:
                                draw_breathless(i)
                        else:
                                draw_working(i)


        def mask(code):
                if code == IDLE:
                        return "   IDLE   "
                elif code == BREATHLESS:
                        return "BREATHLESS"
                else:
                        return " WORKING  "

        showStatus.i += 1

        try:
                print '\x1b[H\x1b[2J'
                print " Segundo Trabalho de NSO "
                print ""
                print "Alunos: Pedro Garcia Freitas"
                print "        Felipe Gomes Lacerda"


                draw(i)
                print "\n\n\n\n"
                print str(map(mask, Global.state))
```

```
        finally:
                pass

showStatus.i = 0

def idle(i):
        """ indicates that there is no supply of oxygen """
        showStatus()
        time.sleep( randint(TIME1,TIME2) )

def work(i):
        """ work when have two oxygen supplies """
        showStatus()
        time.sleep( randint(TIME1,TIME2) )

def take_oxygen(i):
        """ acquire two oxygens to work """

        down(Global.mutex)        # start critical work (enter critical region)
        Global.state[i] = BREATHLESS  # HEY, i need breath too
        test(i)                       # try work with 2 oxygen suplies
        up(Global.mutex)              # exit critical region
        if not Global.s[i].locked():
                down(Global.s[i])     # block if oxygen suply not acquired


def put_oxygen(i):
        """  free oxygen supply """

        down(Global.mutex)         # enter critical region
        Global.state[i] = IDLE     # diver has finished the work
        test(LEFT(i))              # see if left neighbor needs restore oxygen
        test(RIGHT(i))             # see if right neighbor needs restore oxygen
        up(Global.mutex)           # free the critical region

def test(i):
        """  Tests which diver needs to restore oxygen """

        if (Global.state[i] ==  BREATHLESS) and \
                        (Global.state[LEFT(i)] != WORKING) and \
                        (Global.state[RIGHT(i)] != WORKING) :
                Global.state[i] = WORKING
                if Global.s[i].locked():
                        up(Global.s[i])

def signal_listener():
        for sig in range(1, signal.NSIG):
            try:
                signal.signal(sig, signal.SIG_DFL)
            except RuntimeError:
                pass


if __name__ == "__main__":
        print '\x1b[H\x1b[2J'
        for i in range(0,N):
                Diver(i).start()

        signal_listener()
```