Laboratorul 1

la PPC

“Multithreading”

Efectuat de: st. gr. FAF-101, Ursu Dumitru
Verificat de: Lector superior, Ciorbă Dumitru

Scopul: Crearea și sicronizarea firelor

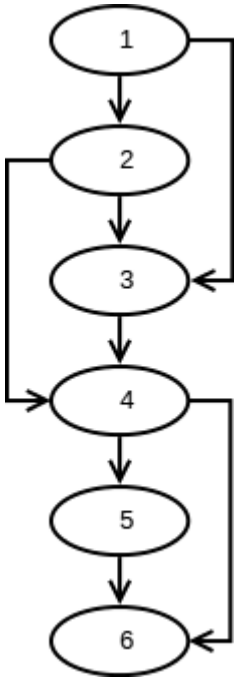

Fig 1. Varianta 6, dependența între fire.

Am decis să aleg pentru lucrările de laborator limbajul Ruby. Ruby este un "limbaj dinamic, cu codul sursă liber, fiind focusat pe simpliciate și productivitate. Are o sintaxă elegantă, care e naturală la citire, și ușor descris" – https://ruby-lang.org
Există mai multe implementări ale interpretatorului de cod ruby, cel mai comun fiind MRI, Matz Ruby Interpreter. Însă, un neajuns al acestei implementări este faptul că el oferă doar "green threads", și pe lîngă aceasta mașina virtuală are implementat un GIL (Global Interpreter Lock). Asta din cauză că gem-urile (biblioteci ruby), pot fi extensii C(cu fire separate), ceea ce ar duce la curse critice, în caz că extensia nu e scrisă bine. Fiind peste 60 000 de biblioteci disponibile, asta motivează dezvoltatorii MRI să mai păstreze aceast detaliu de implementare.

Însă, vestea bună este că pe lîngă MRI, mai există Rubinius, care are fire native (cu costul incompatibilității unor gemuri), și Jruby, intepretatorul ruby implementat pe JVM(Java Virtual Machine), care mapează firele ruby pe cele din JVM, care la rîndul lor pot fi fire native.

Ruby oferă următoare mecanisme de sincronizare a firelor: Mutex-uri, și Monitoare.
Prima mea încercare de rezolvare a laboratorului, a fost prin Mutex-uri și ConditionalVariable. Eventual codul a devenit prea complicat, și în MRI se ajugea la Dead Lock, în Rubinius/Jruby programul pur și simplu se bloca.

```ruby
#! /usr/bin/env ruby
# encoding:utf-8

require 'thread'

mutex1 = Mutex.new
resource1 = ConditionVariable.new

def expensive_operation
  File.open("lab6.dia")
end

t2 = Thread.new {
  mutex1.synchronize {
    resource1.wait(mutex1)
    expensive_operation()
    puts "t2"
    #send signal for t3
    resource2.signal()
  }
}

t1 = Thread.new {
  mutex1.synchronize {
    expensive_operation()
    puts "t1"
    resource1.signal
    resource2.signal
  }
}
mutex2 = Mutex.new
#used to synchronize 3 with 1
resource2 = ConditionVariable.new

t3 = Thread.new {
  mutex2.synchronize {
    #wait t1
    resource2.wait(mutex1)
    #wait t2 to signal
    resource2.wait(mutex1)
    expensive_operation()
    puts "t3"
  }
}

mutex3 = Mutex.new
resource3 = ConditionVariable.new
```

```
t4 = Thread.new {
  mutex3.synchronize {
    #wait t1
    resource3.wait(mutex1)
    #wait t2 to signal
    resource3.wait(mutex1)
    expensive_operation()
    puts "t4"
  }
}

t1.join
t2.join
t3.join
```

Programul l-am realizat, după ce am văzut cum se poate de implementat chestia asta în Java: cu CountDownLatch. Într-o bibliotecă separată, countdownlatch există și o asemenea funcționalitate:

```
#! /usr/bin/env ruby
require 'countdownlatch'

latch2 = CountDownLatch.new 1
latch3 = CountDownLatch.new 2
latch4 = CountDownLatch.new 2
latch5 = CountDownLatch.new 1
latch6 = CountDownLatch.new 2

Thread.new {
  puts "t1"
  latch2.countdown!
  latch3.countdown!
}

Thread.new {
  latch2.wait
  puts "t2"
  latch3.countdown!
  latch4.countdown!
}

Thread.new {
  latch3.wait
  puts "t3"
  latch4.countdown!
}

Thread.new {
```

```
  latch4.wait
  puts "t4"
  latch5.countdown!
  latch6.countdown!
}

Thread.new {
  latch5.wait
  puts "t5"
  latch6.countdown!
}

Thread.new {
  latch6.wait
  puts "t6"
}
```

Programul funcționează corect în Rubinius/Jruby, și arătă aceasta:
t1
t2
t3
t4
t5
t6

Concluzie:
Efectuînd această lucrare de laborator, am învățat cum post coordona mai multe fire de execuție. De asemenea, implementarea clasei CountDownLatch are la bază aceleași mutex-uri, și implementarea e destul de simplă, deși eu nu am putut să creez o clasă asemănătoare.