

Programowanie w Ruby

Wykład 8

Marcin Młotkowski

28 listopada 2012

Plan wykładu

Jak robić kilka rzeczy na raz

Większy przykład

Testowanie programów

Wątki – wyzwania

- ▶ Tworzenie wątków
- ▶ Dostęp do wspólnych zasobów
- ▶ Przekazywanie wartości

Program jednowątkowy

Przykład

```
for url in %w{ftp.com www.org http.net}  
  ściągnij url  
end
```

To samo, ale wielowątkowo

```
require 'net/http'
threads = []
for page in page_list
  threads << Thread.new(page) do | url |
    h = Net::HTTP.new(url, 80)
    resp = h.get('/', nil )
  end
end
threads.each { | t | t.join }
```

Dostęp do wspólnych zasobów

Chcemy zorganizować wyścigi wątków

Implementacja

```
threads = []  
5.times do | num |  
  threads << Thread.new(num) do | n |  
    dystans = 42196  
    dystans -= 1 while dystans > 0  
    puts "#{n} Meta!!!"  
  end  
end  
threads.each { | t | t.join() }
```

A ile przebiegli wszyscy razem?

```
total = 0
```

Mała modyfikacja

```
while dystans > 0
    total += 1
    dystans -= 1
end
```


Próbujemy!

Próba 1.

180634

Próbujemy!

Próba 1.

180634

Próba 2.

161237

Próbujemy!

Próba 1.

180634

Próba 2.

161237

Próba 3.

159984

Próbujemy!

Próba 1.

180634

Próba 2.

161237

Próba 3.

159984

Powinno być

$5 * 42196 = 210980$

Próbujemy!

Próba 1.

180634

Próba 2.

161237

Próba 3.

159984

Powinno być

$5 * 42196 = 210980$

Dlaczego

Implementacja `total += 1`

`val = fetch_current(@count)`

add 1 to val

store val back into `@count`

Przebieg, 1. scenariusz

```
val = fetch_current(@total)
add 1 to val
store val back into @total
```

```
val = fetch_current(@total)
add 1 to val
store val back into @total
```

Przebieg, 2 scenariusz

total = 2

val = fetch_current(@total)

add 1 to val # val = 3

store val back into @total

total = 3

total = 2

val = fetch_current(@total)

add 1 to val # val = 3

store val back into @total

total = 3

Dostęp do zasobów wspólnych

Monitory

`require 'monitor'`

`lock = Monitor.new`

`lock.synchronize { total += 1 }`

Dostęp do zasobów wspólnych

Monitory

```
require 'monitor'
```

```
lock = Monitor.new
```

```
lock.synchronize { total += 1 }
```

Wady

Wolne

Dostęp do zmiennych wątk

Wątki można traktować jak tablice asocjacyjne

Dostęp do zmiennych wątki

Wątki można traktować jak tablice asocjacyjne

```
th = Thread.new do
  Thread.current['zmienna'] = 1024
end
puts th['zmienna']
```

Wielowątkowe systemy asynchroniczne

Są dwa wątki, które wymieniają dane między sobą.

Pierwsze podejście

\$BUFOR # zmienna globalna

PRODUCENT

```
while true
  while $BUFOR; end
  $BUFOR = dane
end
```

KONSUMENT

```
while true
  while not $BUFOR; end
  Thread.new { foo($BUFOR) }
  $BUFOR = nil
end
```

Wady

Wątki działają synchronicznie; prościej można to zaprogramować tak:

```
foo(dane)
```

Wady

Wątki działają synchronicznie; prościej można to zaprogramować tak:

```
foo(dane)
```

Dużo pochłania aktywne czekanie:

```
while warunek; end
```


Schemat dobrego rozwiązania

- ▶ Wątki można usypiać (`Thread#stop`) i budzić (`Thread.run`)
- ▶ Można uśpić KONSUMENTA i obudzić gdy będą dane do przetwarzania
- ▶ Dane do przetworzenia są przechowywane w kolejce

Kolejka Queue (SizedQueue)

require 'thread'

```
qu = Queue.new # SizedQueue(1024)
```

enq(obj)

wkłada do kolejki obiekt. Jeśli kolejka jest pełna (w przypadku SizedQueue) wątek wkładający do kolejki jest usypiany

deq

pobranie elementu z kolejki; jeśli kolejka jest pusta wątek który wywołuje tę metodę jest usypiany

Implementacja

Producent

loop do

 dane = gets.chomp

 qu.enq (dane)

end

Implementacja

Producent

loop do

 dane = gets.chomp

 qu.enq (dane)

end

Konsument

loop do

 obj = qu.deq

 Thread.new(arg) { | arg | foo(arg) }

end

Plan wykładu

Jak robić kilka rzeczy na raz
Większy przykład

Testowanie programów

Tesy jednostkowe

- ▶ Program testujący pojedynczy fragment kodu (funkcję czy metodę)
- ▶ Podczas testowania sprawdzane są asercje
- ▶ Testy można łączyć w pakiety

Testowanie w Rubym

- ▶ Narzędzia są w module 'test/unit'
- ▶ Testy umieszcza się w metodach klasy będącej podklasą `Test::Unit::TestCase`

Testowanie implementacji

Stos

- ▶ Klasa implementuje stos o ograniczonym rozmiarze
- ▶ rozmiar stosu jest ustalany w konstruktorze
- ▶ `enq(obj)` – wkłada obiekt na stos; jeżeli stos jest pełny zgłaszany jest wyjątek `StackOverflow`
- ▶ `deq` – zdejmuję element ze stosu; jeśli stos jest pusty zgłaszany jest wyjątek `EmptyStack`

Implementacja testów

```
require 'test/unit'  
require 'Stack'  
class TestStack < Test::Unit::TestCase
```

Testowanie poprawnej kolejności

```
def test_list
  q = MyStack.new(3)
  0.upto(2) { | i | q.enq(i) }
  2.downto(0) { | i | assert_equal(i, q.deq) }
end
```

Testowanie wyjątków

```
def test_extreme
  q = MyStack.new(2)
  assert_raise(EmptyStack) { q.deq }
  2.times { |n| q.enq(n) }
  assert_raise(StackOverflow) { q.enq(234) }
end
```

Różne testy

```
def test_random
  assert_nothing_raised() { MyStack.new(0) }
  assert_nothing_raised() { MyStack.new(-10) }
end
```

Wynik

Loaded suite test_qu

Started

...

Finished in 0.001094 seconds.

3 tests, 7 assertions, 0 failures, 0 errors