Parallelverarbeitung mit Ruby

Prozess-Ebene

Multithreading

Parallelisierung auf Prozess-Ebene

Prozesse starten und verbinden

fork & exec: Parallele Prozesse

Interrupt handler: Signale abfangen

Prozesse starten

system()

- String mit Kommando übergeben
- Rückgabe: true (falls ok) oder false
- *Exit code* in $?
- Beispiel:

```
system "tar xzf apache-2.2.1.tar.gz" # → true
```

Nachteile:

- Output des Programms landet im selben Output, den auch das Skript erzeugt (meist **stdout**).

` … ` (backticks)

- Wie system(), aber Output wird als String zurückgegeben

```
`date`          # → "Do, 4. Dez 14:07:21 CET 2008"
```

%x|…|, %X|…| (Alternative Notation)

- Wie `…`, mit Trennzeichenkonvention analog zu %w bzw. %W

```
%x/date/        # → "Do, 4. Dez 14:07:25 CET 2008"
```

Prozesse verbinden: Pipes

- **Externen Prozess als Pipe nutzen**
  - Hier: GZIP-Daten extern entpacken lassen

```
unzip = IO.popen("/bin/zcat", "w+")
unzip.print some_gzipped_data     # In Pipe schreiben
unzip.close_write                 #  force a flush!
uncompressed_data = unzip.read    # Aus Pipe lesen
```

- Hinweise:
  - So NICHT entpacken – ZLIB ist eingebaut!
  - Hinweis: „close_write“ erweist sich leider als notwendig…

Prozesse verbinden: Pipes

- **Pipe-Technik für ein implizites fork()**

```
pipe = IO.popen("-", "w+")
if pipe                          # Eltern-Zweig
   pipe.puts "Studiere fleißig!"
   STDERR.puts "Antwort vom Kind: '#{pipe.gets.chomp}'"
else                             # Kind-Zweig
   STDERR.puts "Vater sagt: '#{gets.chomp}'"
   puts "Klar doch ;-)"
end
```

  - Ergebnis:

```
Vater sagt: 'Studiere fleißig!'
Antwort vom Kind: 'Klar doch :-)'
```

(Frei nach „Pickaxe 2nd ed., p, 149“)

Langläufer-Prozess starten

- **fork() und exec() zur Prozess-Parallelisierung**

```
exec("./do_some_numbercrunching") if fork.nil?
# Elternprozess kann nun weiterarbeiten!
# …
# Später dann:
Process.wait  # Wartet auf Ende vom Kindprozesses
```

- Wirkung:
  - „fork“ spaltet den Prozess in einen Eltern- und Kind-Prozess
  - Der Kindprozess erhält „nil“ von fork() und gibt die Kontrolle an das externe Programm „do_some_numercrunching“ unter seiner Prozess-ID ab
  - Der Elternprozess erhält die PID des Kindprozesses zurück und kann parallel weiterarbeiten
  - Am Ende wartet der Elternprozess auf die Beendigung des Kindprozesses

Parallelisierung eines Ruby-Skripts

- **fork() und exec() zur Parallelisierung des aktuellen Ruby-Programms**
  - Live-Demo „fork_env.rb“
- Neue Besonderheiten:
  - Block-Modus von „fork“
  - Mehrstufigkeit: Kind von Kind

Prozesse und Signale

- Signale (Software-Interrupts) lassen sich in Ruby abfangen

```
# Interrupt-Handler:
trap "INT"  { puts "'Strg-C'  erhalten" }
trap "USR1" { puts "'SIGUSR1' erhalten" }
# Hauptprogramm:
loop { print "Echo: #{gets}" }
```

```
$ ruby trap_echo.rb
Hallo
Echo: Hallo
<Strg-C>
'Strg-C'  erhalten
<Strg-Z>
[1]+ Stopped                 ruby trap_echo.rb
$ ps   # PID erfahren, z.B. 8857
$ kill -SIGUSR1 8857
$ fg
'SIGUSR1' erhalten
```

Multithreading

Multithreading

Warum *Multithreading*?

- Auf *Single core*-Maschinen:
  - Beschleunigung IO-begrenzter Prozesse
- Auf *Multiple core*-Maschinen:
  - Beschleunigung auch *compute*-begrenzter Prozesse
- Generell:
  - Effizienter und flexibler als Parallelisierung auf Prozess-Ebene
  - Nutzung paralleler Algorithmen
  - Manchmal elegantere Formulierung von Lösungen

Nachteile?

- Synchronisierung erforderlich
- Debugging (teils erheblich!) schwieriger
- Notorisch fehleranfällig – offenbar sind menschliche Denkmuster schlecht auf die Möglichkeiten und Konsequenzen nebenläufiger Algorithmen angepasst.

Status

- Ruby 1.8
  - Alle Ruby-Threads laufen in einem einzigen *native thread*, daher noch kein Performance-Gewinn
  - „Kooperatives Multi-Threading“: Thread-Wechsel findet automatisch durch den Ruby-Interpreter statt, sofern dieser dazu Gelegenheit bekommt
  - Konsequenz: Ein blockierender Thread blockiert alle anderen auch!
  - Beispiel: Blockierender OS-Aufruf
- Ruby 1.9
  - Ruby-Threads bereits auf *native threads* abgebildet
  - Sperre: Jeweils nur ein Thread darf laufen!
  - Grund: Noch nicht alle C-Bibliotheken sind *thread safe*
- JRuby
  - Verwendet Java *threads*, die *native threads* nutzen – Parallelität!

Ruby *threads*

- Im Prinzip Kontrollstrukturen für nebenläufige Verarbeitung
- OO-Kapselung mit der Klasse „Thread“

- Prinzipielles Vorgehen
  - Nebenläufig auszuführender Code bildet Blöcke
  - Diese Blöcke werden von Thread-Exemplaren gekapselt
  - Rückgabewert jedes Blocks per „value“-methode
  - „value“ blockiert, bis dass der Block/Thread geendet hat

- Beispiel

```
# Paralleles Lesen von Dateien:
threads = filenames.map {|n| Thread.new { File.read(f)}}
threads.map {|t| t.value }          # → Array von Strings
```

Generatoren und *fibers* (Ruby 1.9)

- Die Klasse „Fiber“ stellt eine Kontrollstruktur für Nebenläufigkeit unterhalb der Thread-Ebene bereit
  - Fibers ermitteln Zwischenergebnisse und kehren zurück!
  - Sie starten nur auf Aufforderung („resume“)
  - Grundlagen für Generatoren. Hier: Lottozahlen-Generator

```
class LottozahlenGenerator
  def initialize(n=6, out_of=49)
    @n, @pool = n, Array(1..out_of)
    @fiber = Fiber.new do
       loop do
          x = @pool.delete_at rand(@pool.size)
          Fiber.yield x # Kontrolle zurück an
       end              # Aufrufenden
     end
  end
```

Generatoren und *fibers* (Ruby 1.9), Forts.

```
  def next
    @fiber.resume
  end
end # LottoGenerator
```

```
# Anwendung

ziehung = LottozahlenGenerator.new
6.times { print "#{ziehung.next} "}
puts "Zusatzzahl: #{ziehung.next} "
```

- Warnung
  - Der gezeigte Code ist noch nicht getestet (kein Ruby 1.9)

Der *main thread*

- Sonderrolle:
  - Interpreter stoppt, wenn dieser endet
  - Dies gilt auch, wenn andere *threads* noch laufen!
  - *Exceptions* im *main thread* bewirken Abbruch (wie gewohnt), solche in anderen *threads* beenden nur diesen *thread*

```
Thread.abort_on_exception = true # ändert dies!
```

- Synchronisierung am Ende

```
threads = (1..5).map do |i|
  Thread.new { sleep rand(20); puts "Thread #{i}" }
end

# Auf Beendigung aller Threads warten
threads.each {|t| t.join }
```

Exklusiver Zugriff, Mutex

- Situation:
  - Mehrere *threads* müssen ein gemeinsames Objekt aktualisieren
  - Ist die Aktualisierung nicht atomar, darf sie nicht unterbrochen werden!
- Beispiel:

```
c = 0          # ein gemeinsamer Zähler
threads = (1..5).map do
   Thread.new { 1000.times { c += 1 } }
end
threads.each {|t| t.join }
puts c         # → 1716              < 5000 !
```

- Ursache:
  - c += 1 ist nicht atomar: lese c, inkrementiere, schreibe zurück

Exklusiver Zugriff, Mutex

- Abhilfe:
  - ***Locking*** des kritischen Code-Abschnitts
- Beispiel:

```
require "thread"     # Ruby 1.9: unnötig (eingebaut)
mutex = Mutex.new
c = 0                # ein gemeinsamer Zähler
threads = (1..5).map do
    Thread.new do
      1000.times { mutex.synchronize { c += 1} }
    end
end
threads.each {|t| t.join }
puts c          # → 5000              nun OK !
```

Zum Schluss: Vorsicht – *deadlock*!

- Beispiel:

```
require "thread"     # Ruby 1.9: unnötig (eingebaut)
m, n = Mutex.new, Mutex.new
t = Thread.new do
  m.lock; puts "Thread t sperrte Mutex m" ; sleep 1
  puts "Thread t will Mutex n sperren"; n.lock
end

s = Thread.new do
  n.lock; puts "Thread s sperrte Mutex n" ; sleep 1
  puts "Thread s will Mutex m sperren"; m.lock
end

t.join; s.join       # Gegenseitige Blockade!
```

Quelle: The Ruby Programming Language, Kap. 9.9.7.1