

30/08/2011

## H1. Mult Threading.

- ① Vb: BSN's tellen.
- ② Non-Determinisme.
- ③ Gedrag van objecten.
- ④ Consistentie
- ⑤ Progress.

## 1 BSN's tellen

Hoeverel BSN's zijn er?  $a b c d e f g h i$   
 $\times g \times d \quad \times 2 \times -1$ , som: 11-voud.  
Check je eigen BSN. (Verkeerd begrepen: grote kans  $\neq 0$ ).

```
Prog    tel = 0  
        for (b=0; b < M; b++)  
        { ... elf proef...  
          if (b geslaagd) t++  
        }  
        Console.WriteLine("tel: " + tel)
```

Na 2 min antwoord: 90909091

Splitten over  $p$  threads, verdeel bereik in  $p$  stukken  $0 \dots p-1$ .

```
for (t=0; t < p; t++)  
    for (b = t * M / p; b < (t+1) * M / p; b++)
```

Eindpunt van thread  $t$  is beginpunt van thread  $t+1$ .

Thread 0 begint by  $0 * M / p$  is 0

$p-1$  eindigt by  $p * M / p$ , is  $M$ .

Dus: elk getal in  $[0, M)$  wordt een maal bekeken.

Om er  $p$  threads van te maken moet je 3 of 4 dingen doen.

Declareren & Aanmaken

Starten

Joinen

Rekentijd op Dual Core: 11 min, by elke  $p \geq 2$ .

Het gaat dus sneller.

```

using System;
using System.Collections.Generic;
using System.Threading;
using System.Linq;
using System.Text;

namespace TelBSN
{
    class Program
    {
        static int eind, tussen, p, tel;
        static Object slot = new Object();

        static void Main(string[] args)
        {
            eind = 1000000000; // Int32.Parse(Console.ReadLine());
            Console.WriteLine("Dit programma telt het aantal geldige BSNs tot " + eind + ".");
            //Console.WriteLine("Tot welk getal moet ik tellen?");
            //Console.WriteLine("Tussenrapporten elke hoeveel nummers?");
            // tussen = 50000000; // Int32.Parse(Console.ReadLine());

            // parallellisme
            p = 7;

            tel = 0; // tel de geldige BSNs

            DateTime startTime = DateTime.Now; // Bijhouden hoe lang het duurt

            // Declareer threads
            Thread[] ts = new Thread[p];

            // Creeer threads
            for (int t = 0; t < p; t++)
            {
                ts[t] = new Thread(telbereik);
            }
            // Start threads
            for (int t = 0; t < p; t++)
            {
                ts[t].Start(t);
            }
            // Join threads
            for (int t = 0; t < p; t++)
            {
                ts[t].Join();
            }

            // Antwoord afdrukken
            Console.WriteLine("Tot aan " + eind + " zijn er " + tel + " geldige BSNs.");

            // Rekening
            DateTime stopTime = DateTime.Now;
            TimeSpan elapsedTime = stopTime - startTime;
            Console.WriteLine("Rekening in milliseconds: " + elapsedTime.TotalMilliseconds);
            Console.ReadLine();
        }

        public static void telbereik(object mt)
        {
            // Uit het object mt (thread nummer) bepalen we onder-en bovenlimiet
            // De p threads samen zoeken van 0 tot (exclusief!) (eind*p)/p,
            // dus exact alles, ook als de deling niet exact opgaat!
            // Omdat eind*p groter dan maxInt kan zijn moet je even met long
            int from = (int)((int)mt * (long)eind / p);
            int to = (int)((int)mt + 1 * (long)eind / p);
            Console.WriteLine("Thread " + mt + " telt van " + from + " tot " + to
                + " (bereik-omvang: " + (to-from) + ").");

            for (int bt = from; bt < to; bt++)
            {
                // Implementeer de 11-proef voor getal bt, kopieer eerst naar b
                int b = bt;
                int s = -b % 10; b /= 10; // laatste cijfer telt -1 keer
                for (int c = 2; c <= 9; c++)

```

Hoeveel sneller? Dit ziet er uit als:  $p$  onafh. threads (dus  $p$  cores) gaat  $p \times$  zo snel.

Vaak zitten er stukjes in de code die niet parallel kunnen. Amdahl's Law gaat over Speedup  $S$   
$$S = \frac{\text{sequentiele tijd}}{\text{parallelle tijd}}$$

Je hoopt, met  $p$  cores,  $S \approx p$  te bereiken.

Amdahl: stel "fractie  $\alpha$ " van je programma is paralleliseerbaar,  $1-\alpha$  moet sequentieel.

Tijd op 1 core:  $(1-\alpha) + \alpha$   
 $p$  cores:  $(1-\alpha) + \alpha/p$

Zodra  $p$  zo groot wordt dat  $\alpha/p < (1-\alpha)$ , heeft meer processors toevoegen weinig zin. Beter algoritme verzinnen!

## Hoeveel threads?

Als je meer threads maakt dan je cores hebt: threads worden beurtelings in- en out-geswapt, effectief is je parallelisme "maar" je aantal cores.

Latency hiding: Vaak moet een thread wachten, bv op geheugen: "kort" maar toch zo'n 100 klok cycles  
page-fault: van disk halen duurt 10 mln klok cycles

Daarvoor kun het nuttig zijn om "extra" threads te hebben. <sup>user input</sup> <sup>andere thread</sup> Afhankelijk van je applicatie kan het aantal threads 5 tot 10 maal aantal cores bedragen, zolang je al je cores volledig uitput.

## 2 Non-Determinisme

Onzekerheid in executietijd van een instructie of thread!

- Niet alles duurt even lang.
- Wachten is onvoorspelbaar.

Modelleren met: Non-Determinisme  
en Asynchroniteit met Fairness.

Neem voorbeeld van twee threads, elk met reeks instructies ("stappen", zie verderop).

We beschouwen de parallelle executie als reeks van stappen.

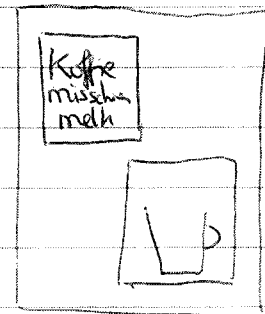
De instructies worden dus in een volgorde uitgevoerd.

Op elk "moment" (toestand) heeft elke thread een stap die hij "wil doen", uit de willige threads wordt van een-tje de stap gekozen.

### 2.1 Non-Determinisme

Welke gekozen wordt: daar kun je op enig moment niets over zeggen.

Vb Koffie - melk - misschien - melk.



Non-determinisme: afwezigheid van een argument voor wat er gaat gebeuren.

Vb Termineert dit programma:

De body kan  $i$  op 0 of op 1 zetten. Omdat het tempo van de threads onvoorspelbaar is, hebben we geen reden om te zeggen dat de laatste zal zijn.

$i = 0$

while ( $i == 0$ )

start  $i = 0$   $i = 1$

join

Vb 1

de  $i = 1$  een keer

## 2.2 Fairness

We gaan wel uit van asynchroniteit, maar ook dat elke thread wel ooit voortgang boekt.

Vb Termineert dit programma:

Voor elke  $K$  is het mogelijk dat thread 0,  $K$  maal door de body gaat en dan stopt op  $i = 1$ .

$i = 0$   $t = 0$

a while ( $i == 0$ )  
{  $t++$  }

print  $t$

$i = 1$

Vb 2

Maar: ooit zal Th 1 zijn operatie mogen doen!

Def Een reeks van stappen is unfair als er een stap is die oneindig lang klaar staat maar nooit gekozen wordt.

Aanname Fairness: De executie is fair.

De aanname van fairness impliceert dat Vb 2 termineert.

VRAAG: Impliceert fairness dat Vb 1 termineert?

## Randomness

De body van Vb3 doet hetzelfde als van Vb1, nl i op 0 of 1 zetten.

Kunnen we bewijzen dat Vb3 termineert?

Kans dat termineert na 1 ronde:  $\frac{1}{2}$  (Output 1)  
2  $\frac{1}{4}$

Kans op "een output"

————— +

Randomness is iets anders dan Non-Determinisme omdat je kunt redeneren over wat er gebeurt!

### ③ Gedrag van Objecten.

Het antwoord komt snel: Tel BSN in 1 minuut.  
Antwoord 90607546      wat zie ik dat goed? nog eens FS  
90607186

Het is niet alleen fout, maar ook steeds anders!  
Hierdoor vrijwel moeilyk om een multithreaded prog te debuggen.

Wat gaat hier mis?      tell + werkt niet zoals we  
verwachten als hij door twee threads tegelyk wordt gedaan

Uitgangspunt is specificatie van een methode wanneer  
hy geïsoleerd (dus alleen) wordt gedaan.

Sequentiële Specificatie.

- De operatie termineert
- Wat is het effect  
(Operatie INC, wanneer tel waarde K heeft,  
zal eindigen in een situatie waar tel waarde  $K+1$  heeft).

Wanneer verschillende threads tegelyk op een object werken,  
wat kunnen we dan van het object verwachten?

Elke methode heeft per thread een begin en eindpunt.  
Er is dus een interval waarin de thread



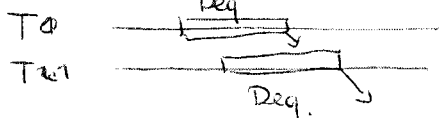
## 4 Consistentie.

Wat is de uitkomst van overlappende operaties?

Principe van Consistentie:

Uitkomst is altijd hetzelfde als wanneer de operaties na elkaar zijn gedaan.

Dus: Queue met [A B C] erin



Uitkomst is nooit: D of empty exception, of tweemaal A.

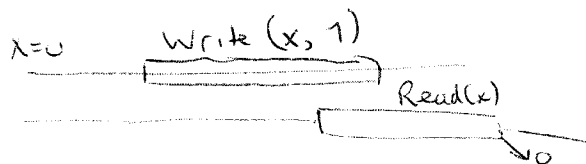
Maar: mag het wel dat T0 B krijgt en T1 A?

JA, al ziet het er naar uit dat T0 "eerder" is.

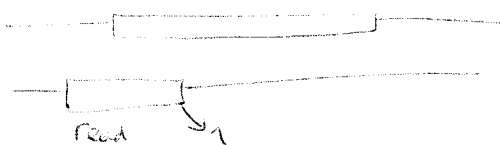
Voor de volgende gebruiken we de eis van Linearizable, werd in vak Gedistribueerd Programmeren ATOMAIR genoemd.

Definitie Operaties zijn atomair (linearizable) als je altijd aan elke operatie een contractie punt binnen het uitvoerings interval kunt toewijzen, zo, dat het lijkt alsof hij op dat punt is uitgevoerd.

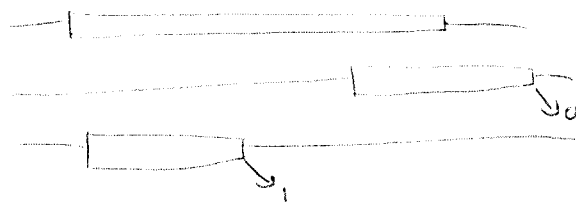
Dit mag wel:



dit ook

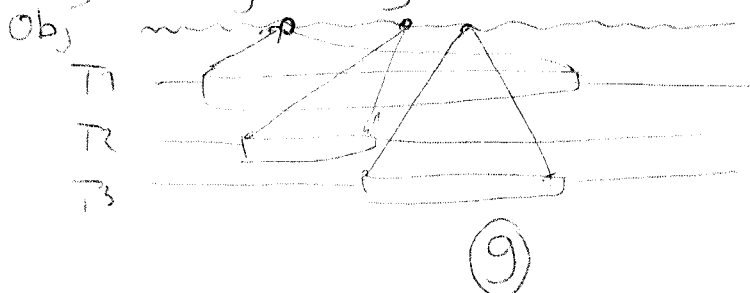


maar dit niet



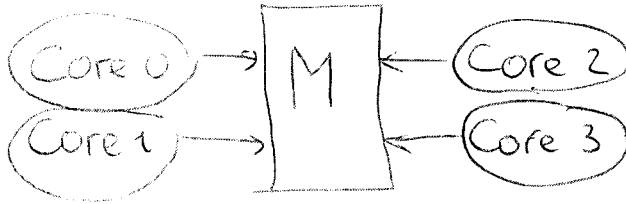
Inversie

Voorstelling die je erby hebt:

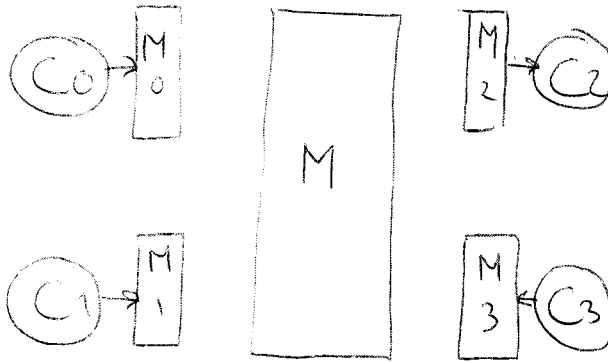


Tussen begin en eind wordt er met het object gecommuniceerd, daar worden de requests gesequentialiseerd.

Dat is niet wat er gebeurt.



Gehengen operaties zijn duur dus hebben de cores caches



Operaties van een thread kunnen op M worden gedaan, maar dat is  $\sim 100\times$  zo duur als op de cache.

Cache-operaties: "Laad" stukje geheugen in cache  
"Schrijf" stukje cache naar geheugen.

Onder concurrentie moet je in de gaten houden dat een andere thread ook het zelfde stukje geheugen in de cache heeft, zodat jouw kopie invalid kan zijn!

Je kunt atomaire Read / Write  
(en zelfs RMW-operaties als `t++`)  
krijgen maar dat's best duur.

## 5 Progress

Welke garantie heb je dat je operatie überhaupt afloopt?  
Kunnen overlappende operaties elkaar dusdanig voor de voeten lopen dat ze nooit termineren?  
Er zijn verschillende varianten van Progress.

### Zwak: Dependent Progress, "Blocking".

Vb je gebruikt een lock op teller.

lock (slot) { tel++ }

Als het lock bezet is moeten andere threads wachten.  
"Makkelijke" manier om atomiciteit te krijgen,  
maar de voortgang van een thread wordt afhankelijk van wat andere threads doen!

Meerdere locks kunnen leiden tot deadlock.

I.v.g. grote vertraging als de thread die een lock heeft zelf ook weer gaat blokken.

### Sterk: Wait-Free

De operatie gaat altijd termineren in een begrensd aantal stappen, zonder wachten.

Wachtvrije teller: Geef elke thread een eigen teller.

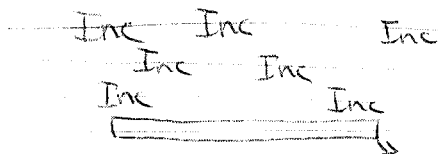
deel: int [p],

inc voor thread i: deel[i]++

uitlezen: { local s = 0  
for (i = 0; i < p; i++) { s += deel[i] }  
return s }

Omdat calls van een thread elkaar niet overlappen, zal deel[i] keurig het aantal Inc's door thread i tellen.

Een uitlezing kan met meerdere Inc's overlappen!



Termineert toch na vast aantal stappen (max p) en geeft een waarde, overeenkomend met dat sommige overlappende zijn meegeteld.

Tussen. (Soms is wacht-vrij niet makkelijk te realiseren)  
Lock-free: er is wel de garantie dat er steeds een operatie ter minste maar geen individuele garantie voor een thread.

We gaan een lock-free counter maken met de zg. Compare-And-Swap.

In x86 zit CPXGN instructie, noemen wij Cas

$t.CaS(x, y)$  doet dit atomair  
lees gehangen  $t$ ;  
als de waarde  $x$  is schrijf dan  $y$  en  
return: de oude waarde van  $t$ .

Normale increment:  $local = t$   
 $local = local + 1$   
 $t = local$

Gaat mis als de waarde in stap 3, niet meer die van stap 1 is.

Lock-free increment:

```
local = t, s = False
while (not s)
{
  r = t.CaS(local, local + 1)
  if (r == local) { s = True }
  else { local = r }
}
```

De increment-poging kan succesloos zijn als een ander heeft ge-update tussen twee accessen naart.

Dus, er is globale progress:

Als er een of meerdere threads aan een increment bezig zijn,  
Dan zal er binnen eindige tijd een inc aflopen.

Voor de individuele thread is er geen garantie, want het aantal keren dat hij moet proberen is niet met een zinnig argument te begrenzen.

Als een thread alleen is in het object, dan is natuurlijk wel zijn succes gegarandeerd.