

H4 Een Hierarchie van Primitieven

- ① Snapshot: Wat
- ② Lock-free Snapshot
- ③ Ticket met Snapshot
- ④ Registers en 2-Consensus
- ⑤ TaS superieur aan Reg
- ⑥ Baas boven baas: CaS
- ⑦ Consensus Getal
- ⑧ Conclusie

Uit vorige colleges: met registers (R/W atomair) kun je veel, bv locks implementeren, en: "sterke" uit "zwakke". Vandaag. Snapshot-object, met toepassing: Counting en: grens van registers in zicht! Counting is bijna Ticketing... maar dat is "lastig" met R/W... Het kan nl niet!

Consensus-probleem: Registers kunnen niet oplossen
TaS / Ticket wel voor $n=2$!
CaS voor $n>2$.

Consensus is Universeel.

① Snapshot: Wat

Snapshot-object lijkt een beetje op een register, je kunt lezen en schrijven. Wat maakt hem anders: hij heeft een "eigen stukje" voor elke thread, dat die thread kan schrijven: update
je kunt geheel lezen : scan.

Het "werkt" dus als (voor type V)

S: V[n]

update(x) door Thread i : s[i] = x

scan return s

NB: hele array tegelijk!

Wat kun je ermee?

Een teller maken met inc dec read.

Voor de eerste 2, laat Th[i] een locale l byhouden.

dec voor i : { l--; s.update(l) }

read : { a = s.scan

t = 0;

for(i) t += a[i]

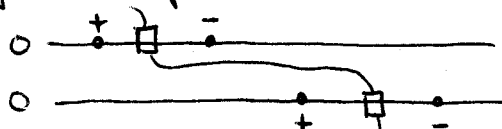
return t

}

De vraag is: hoe kun je "atomair" uitlezen.

Een Read die overlapt met inc dec inc dec mag

0 of 1 op leveren



Ongecoördineerde Scan: 2.

② Een Lock-Free Snapshot

Het Snapshot object is wait-free te bouwen met registers. Dwz zo, dat elke operatie gegarandeerd in begrensde tyd afloopt, met atomaire consistency.

Hoe het niet kan:

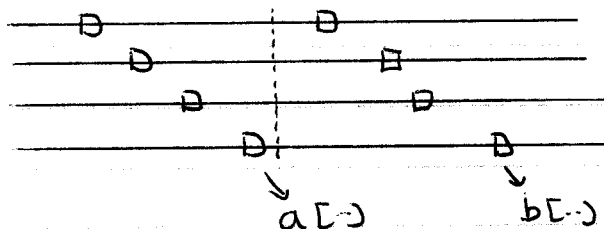
- Alles na elkaar lezen: niet gegarandeerd dat de waarden ook tegelyk zijn voorgekomen.
- Maak 1 groot register waar hele state in past. Scan is dan makkelijk hoe doe je Update?
 - lees alles
 - verander stukje
 - schryf terug

Gaat natuurlijk mis als er tussendoor updates zijn.

Ik ga 'm lock-free (obstruction-free) laten zien.

Idee: double collect - lees alles 2x.

Als er niets veranderd is, is de reeks waarden geldig op het moment tussen de twee rondes.



By verschil: herhaal!

Dus idee van double collect:

Gedeelte S: $V[]$.

scan: a, b $V[]$

```
for(i) a[i] = s[i].read
```

```
for(i) b[i] = s[i].read
```

```
while ( $\exists i: a[i] \neq b[i]$ )
```

```
    { a = b; for(i) b[i] = s[i].read }
```

```
return a
```

a Waarom is het lock-free?

Het is gegarandeerd dat er globale progress is. Na de tweede collect kan de scannende thread ofwel zelf termineren, of er is een succesvolle update tussendoor gekomen.

In dit geval is dat niet zo bevredigend. Want er zijn twee methoden, en de enige progress die hieruit volgt is, dat updates kunnen voltoren.

b Individuele garantie:

De enige manier om een thread te laten falen is anderen ertussen door te sturen.

Als een thread alleen in het object bezig is

Dan kan hij zijn operatie (binnen $3n-1$ reads) afmaken.

Dit is de enige individuele garantie die we hier hebben, en deze eigenschap heet obstruction-free

c ABA en serienummers.

Het is niet correct om op waarden te vergelijken omdat kan gelden $b[i] == a[i]$, terwijl Thread i , tweemaal een update heeft gedaan!

Oplossing: voeg serienummers toe.

d Hoe maak je het wait-free?

Lemma Als j een dubbelverstoorder is voor i ,

Dan ligt het begin van j 's 2^e update binnen de scan van i .

Uit te werken tot: Reg \models_{WF} Snapshot

③ Tickets met Snapshot?

De Counter heeft gescheiden inc en read, en je kunt hem maken met een snapshot. (Aannemende dat je die atomair en wacht-vrij hebt).

De Ticket doet een inc+read samen: de 1^e aanroeper krijgt 1, de 2^e 2, enz.

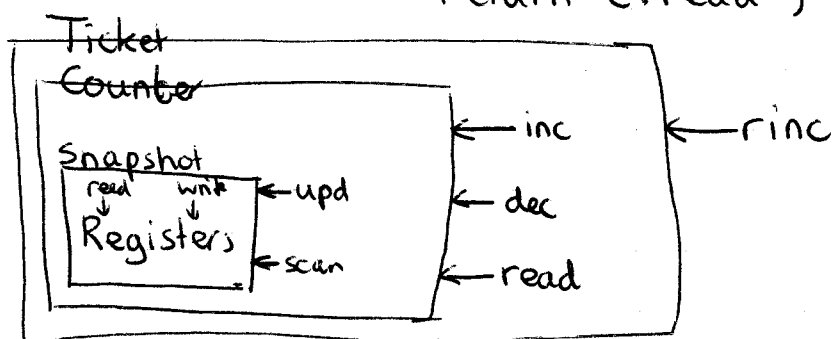
idea Class Ticket

```
c: Counter  
rinc() {c.inc;  
        return c.read }
```

Werkt dit?

Waarom (niet)?

Is er een andere oplossing?

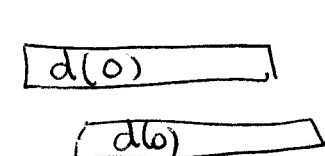


Ik ga bewijzen:
Ticket kan niet
met registers.

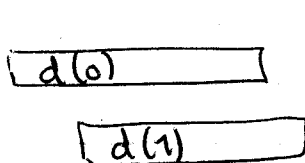
Consensus-Probleem: fundamenteel probleem, kunnen threads het met elkaar eens worden?

Consensus object heeft 1 methode: decide (...),

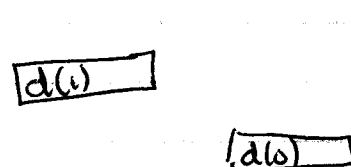
- Mag 1x per thread aangeroepen worden
- Met input (bv. bit, 0 of 1)
- Output is voor alle threads gelijk
- Moet één van de invoeren zijn.



Geeft natuurlijk:
0



hier mag 0
of 1 uitkomen
WEL by allebei!



Wat zou hier
uitkomen?

Conclusie van laatste scenario:

De (gemeenschappelijke) uitvoer van `decide()` is het argument van de eerste aanroep

Je kunt de werking dus zo voorstellen; het ziet er uit of je het met een register kunt maken, maar:

dit werkt natuurlijk maar voor 1 thread!

```
st : init a
decide(x)
{ if (st == a) { st = x }
  return st
}
```

Het belangrijkste, wat ik straks ga laten zien is: je kunt Consensus NIET wacht-vrij implementeren met registers.

(Blocking natuurlijk WEL, zet om deze code een lock).

Maar eerst: Met een TaS (en twee registers) WEL.

```
arb : TaS init 0
prop : bool[2]
```

`decide(x)` voor `i`:

```
{ prop[i] = x
  if (arb.TaS == 0)
    { return prop[i] }
  else { return prop[1-i] }
}
```

Aanpassing voor Ticket:
arb Ticket init 0.

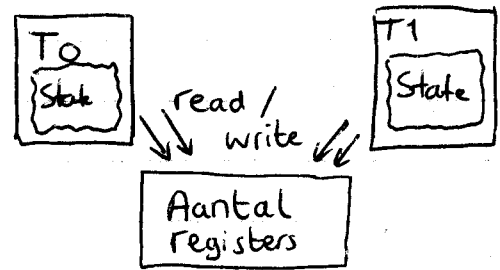
`arb.ticket == 1`

Dus een Ticket-object kan Consensus implementeren (voor 2 threads)

④ Registers en 2-Thread Consensus

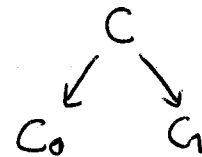
Bewyzen dat je met registers geen Consensus kunt bereiken is lastig, maar niet onoverkomelijk. Stel dat we een implementatie A hebben, bestaande uit een stuk code A_0 en A_1 , voor threads 0 en 1, die de decide implementeren.

De threads hebben lokaal geheugen en delen alleen Reg, d.w.z. ze kunnen het gemeenschappelijke deel alleen benaderen via read's en write's.



Een globale toestand C , een "foto" van het systeem bestaat uit (State T_0 , State T_1 , Waarde Registers)

Een stap van T_0 of T_1 brengt het systeem in een nieuwe toestand, waarbij bv



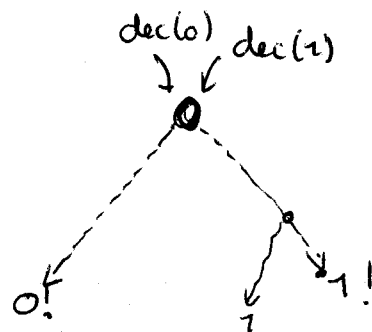
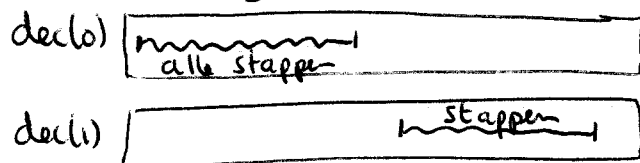
voor C_0 zal gelden dat de State T_0 anders is dan in C , mogelijk ook de Registers (als er een write is gedaan), maar de State T_1 zal nog hetzelfde zijn.

Stap kan zijn: intern, read op r , write(x) op r .

Vanuit een configuratie (toestand) wordt steeds de volgende stap non-deterministisch gekozen.

Soms/Vaak kan de keuze van de stap de uiteindelijke uitkomst nog beïnvloeden.

Stel bv dat: T_0 begint aan $decide(0)$ en T_1 aan $decide(1)$
 Als er eerst alleen maar stappen van T_0 worden gekozen, zal de $decide(0)$ feitelijk voor de $decide(1)$ liggen:



liggen: met als gevolg dat de uitkomst 0 moet zijn, en Vv.

Je kunt je voorstellen dat, als eerst T_1 wat stappen zet, dan T_0 gaat werken, misschien toch T_1 al zoveel heeft gedaan dat er 1 uitkomt.

Def Config C heet bivalent als beide uitkomsten mogelijk zijn
 $C^{(0)}$ $C^{(1)}$ 0-valent als alleen 0 nog mogelijk is
1-valent als alleen 1 nog mogelijk is.

Wait-free eigenschap: Vanuit elke toestand kan er een reeks stappen zijn van alleen T_0 (of T_1), waarin T_0 zijn decide afmaakt.

De threads communiceren alleen via de gedeelde variabelen, de toestand van de ene thread heeft geen rechtstreekse invloed op de ander.

Def C_a en C_b zijn i-gelyk als ze dezelfde State T_i en dezelfde Registerwaarden hebben.

Distributie: Reeks van stappen "onder C_a " van T_i , is ook toepasbaar onder C_b (als C_a en C_b i-gelyk).

Dus Onmogelijk: C_a i-gelyk C_b en de ene 0-, andere 1-valent.

Wat kunnen we bewyzen over zo'n Consensus-
implementatie?

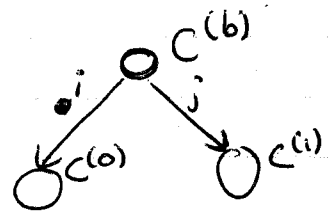
Lemma 1 Er bestaat een bivalente C zonder
bivalent kind.

Bewys: Stel elke bivalente C heeft een bivalent
kind. Dan is er een oneindig lang bivalent
pad: een willekeurig lange executie zonder return! \square

Lemma 2. Een bivalente C zonder bivalent kind
heeft een 0-valent en een 1-valent kind.

Bewys. Zijn beide kinderen zijn univalent, dus
0- of 1-valent. Maar, als beide dezelfde valentie
hebben, dan heeft C zelf die valentie ook al. \square

Kortom, er moet een toestand C
zijn, waarvoor geldt: als nu de ene thread
een stap doet wordt het 0-valent, als
de andere een stap doet, 1-valent.



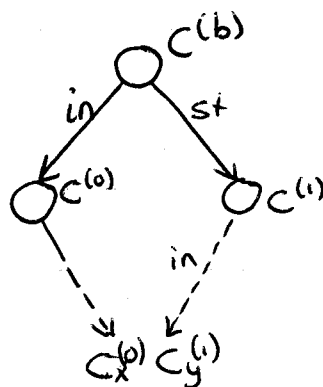
Bedenk ook: "Onder" een 0-valente zitten alleen
0-valente, voor 1 idem dito.

En: Als T_i een stap doet, staat T_j nog steeds
aan het begin van dezelfde stap.

Wat kunnen het voor stappen zijn, die de overgang naar 0- of 1-valentie forceren?

Geval 1: Een interne stap

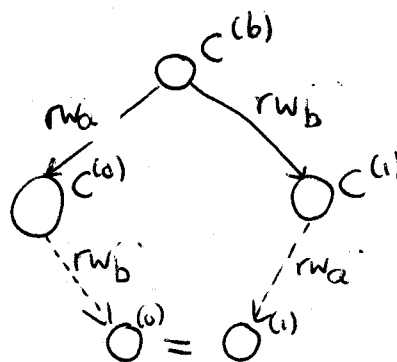
Een interne stap commuteert met een stap van de andere thread: $st(in(C)) = in(st(C))$, dit is in tegenspraak met de verschillende valentie



Geval 2: Register-stappen op verschillende registers

Ook die commuteren:

$$rw_b(rw_a(C)) = rw_a(rw_b(C))$$



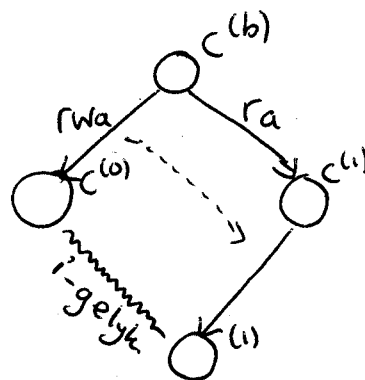
Geval 3 Stappen op 1 register, die van j een read.

Na de read van j wil i nog steeds de zelfde stap doen en, omdat een

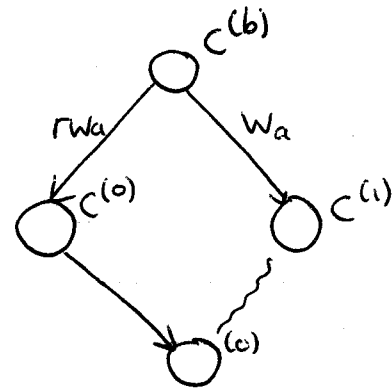
read geen sporen nalaat

in het register, is de (eventuele) uitkomst gelijk.

Dus: $rwa(C)$ is i-gelyk $rwa(ra(C))$, strydig met verschillende valentie!



Geval 4 Stappen op 1 register,
die van j is een write.



Ook na de stap van i wil
 j nog steeds writen en, om-
dat een write de complete
historie van een register wist,
is daarna "voor j niet te zien of er wel of niet
geschreven is!" tegenspraak.

$$w_a(C) =_j w_a(rw_a(C))$$

Alle gevallen zijn onmogelijk!



Redenering gebruikt karakteristiek van register-operaties:

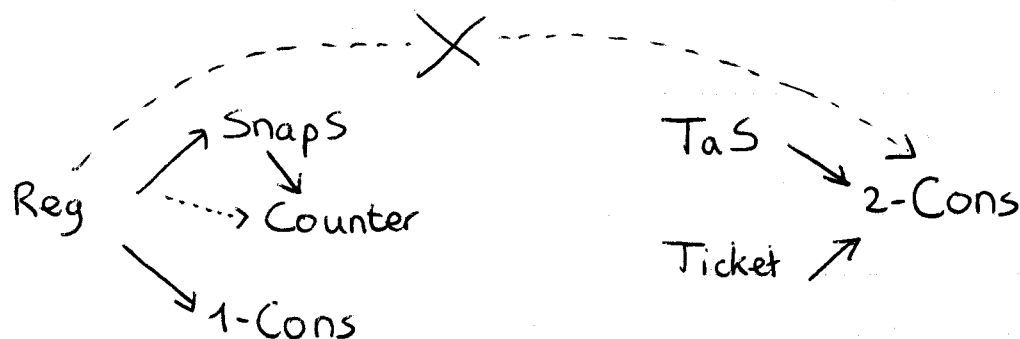
- Read is onzichtbaar (heeft geen effect)
- Write gooit alles weg wat daarvoor is gebeurd.

⑤ TaS is Supérieur!

De Test-and-Set (en Inc-and-Read) zijn als machine-instructie, duur, dus je wilt kijken of je hem kunt vermijden.

Met Registers kun je locks implementeren (Hf 2) en daarmee elke operatie blocking atomair maken.

Als je wachtvrij wilt zijn, kan Consensus wel met TaS of Ticket, niet met Registers.



Hieruit volgt: met geen van de objecten links kan ik eentje rechts maken!

Dus Reg \neq TaS
 Reg \neq Ticket
 Counter \neq Ticket

Verrassend want ze doen bijna 't zelfde!

Dit was reden om TaS toe te voegen aan de x86 architectuur vanaf de 386.

⑥ Baas boven Baas: Compare and Swap.

Met de Tas of Ticket kun je wel Consensus doen voor 2, maar niet voor meer. Reden: je kunt wel zien "of je de 1^e was", maar niet: wie de eerste was.

```
decide(x):  
  { prop[i] = x  
    if (arb.Tas == 0)  
      { return prop[i] }  
    else { return prop[j] }  
  }
```

Twé feiten zijn interessant:

- ① Met Compare-and-Swap kun je Consensus voor willekeurig veel threads
- ② Met Tas of Ticket kan 3-Cons NIET.

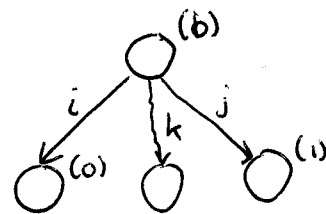
Hieruit volgt: de CaS is superieur aan de Tas.
Reden om de CaS in de x86 te stoppen vanaf Pentium.

Eerste feit:

```
arb: CaS init @  
decide(x)  
  { y = arb.CaS(@, x)  
    if (y == @) { return x }  
    else      { return y }  
  }
```

Deze implementatie van decide(x) werkt voor willekeurig veel threads.

Dat 3-Cons niet kan met Tas.
 Bekijk de situatie C is bi
 $\text{stap}_i(C)$ is 0
 $\text{stap}_j(C)$ is 1.



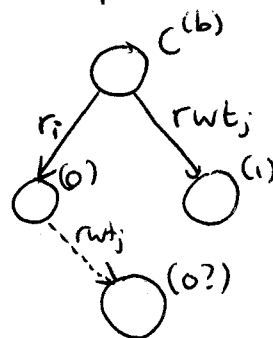
Een stap kan zijn: intern

read/write op gedeeld reg

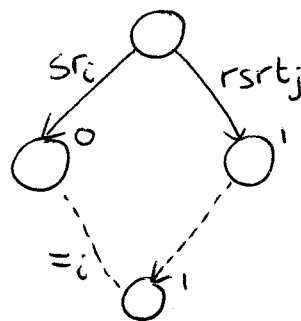
read/set/reset/tas op gedeelde tas.

De situatie van interne stappen, verschillende objecten of dezelfde registers is al behandeld als Geval 1-4. Kijk alleen naar: i en j werken op dezelfde Tas.

Geval 5 Een van hen doet een read. Als by registers is de read onzichtbaar voor de ander:
 $\text{rwt}_j(\text{r}_i(C)) = \text{rwt}_j(C)$,
 tegenspraak.



Geval 6 Een van hen doet set of reset.
 Omdat die (als een write) alles uitwist, geldt
 $\text{sr}_i(C) = \text{sr}_i(\text{rsrt}_j(C))$



Geval 7 Beide doen een Tas.

Voor 2 threads was dit

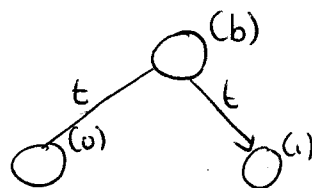
inderdaad de reddende

handeling. Maar by 3 niet,

want: $t_i(C) =_k t_j(C)$.

"k kan niet zien wie er geTasT heeft".

Tegenspraak!



⑦ Consensus Getal

Consensusgetal van een soort object:

Voor hoeveel threads kun je Consensus Oplossen?

CN=1: Registers, plus wat je ermee kunt maken:

Snapshot, Counter

CN=2: Test-and-Set, Ticket

ook: Queue, Stack

en: Alles wat je hiermee kunt maken is
2 of 1!

Deze objecten kun je NIET maken met registers!

CN= ∞ : Compare-and-Swap

Sterke Electie - geeft Thread nr van 1^e aanroeper.

SwapArray - read(i), write(i, x), swap(i, j).

Deze kun je NIET maken met registers en ook niet met Test-and-Set.

⑧ Conclusie

Registers kunnen veel,
De TaS kan nog meer,
De CaS kan het meest.

Alle registersoorten kunnen evenveel,
van Safe SRSW bit
tot Atomic MRMW register.

Met registers kun je locks implementeren.
Dus: als je blocking progress OK vindt,
is elke functionaliteit atomair te maken.

TaS en CaS zijn op x86-architectuur beschikbaar,
kosten ≈ 100 reads/writes.

Met 1 TaS kun je multi-thread locken.

TaS kan Ticket, en synchroniseren tussen 2 threads.

CaS kan alles wachtvrij, willekeurig veel threads.