

H3 Register Implementatie.

- ① Uitleg typen
- ② Safe bit (SRSW)
- ③ Multi Reader Safe
- ④ Een Regular Bit
- ⑤ Regular m-valued
- ⑥ Atomic m-valued
- ⑦ Atomic Multi Reader
- ⑧ Conclusie.

We denken in termen van geheugenplaatsen dat je ze atomair kunt lezen en schrijven, met 32 of 64 b tegelijk vanuit vele threads.

Doorgaans is zoiets wel beschikbaar in je programmeertaal of machine.

Vandaag kijken we hoe je "sterke" registers kunt maken uit zwakke.

① Typen register.

Registers kennen read (plaats rechts van =)
en write (plaats links van =)

De sequentiële specificatie zegt dat de opgeleverde waarde van een read, de parameter van de laatste voorgaande write is.

Waarde. m verschillende.

Een bit is een twee-waardig register.

Normaal heeft een register meerdere bits, bv 32, en kan dan 2^{32} verschillende waarden op slaan.

Aantal gebruikers.

- Single of Multiple Writer: de write operatie is toegankelijk voor slechts één, dan wel meerdere threads.
- Single of Multiple Reader: de read operatie.

We eisen steeds dat registers wacht-vrij zijn: voor het aflopen van een operatie is een thread dus nooit afhankelijk van een andere.

Nu de toegestane uitkomsten bij concurrency.

Het zwakst heet Safe:

- Als een read niet overlapt (met een write) is de uitkomst de waarde van meest recente write. (Een overlappende read mag "alles" returnen)
- Regular als een read die overlapt met writes, de waarde van de voorlaatste, of een van de overlappende teruggeeft.

Vraagje: Is dat wat we willen?

- Atomic behoudt ook volgorde: een strikt latere read kan geen oudere waarde returnen.

We gaan zien:

Safe SRSW bit

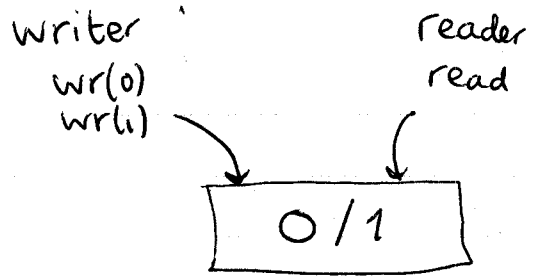
implementeert

Atomic MRMW m-valued register.

② De Safe Bit

Zwakst mogelijke object waarmee communicatie mogelijk is.

- Kent twee toestanden, 0 en 1
- De writer beschikt over $write(x)$ operatie
- De reader heeft $read$.

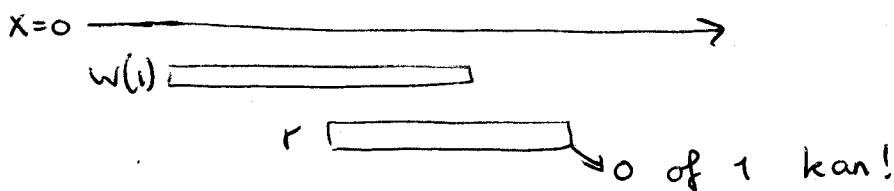


Gestarte $read/write$ lopen gegarandeerd af.
 R/R overlap kan niet omdat er maar 1 reader is.
 W/W overlap ook niet.
Safe wil zeggen: R/W overlap maakt 'm niet stuk,

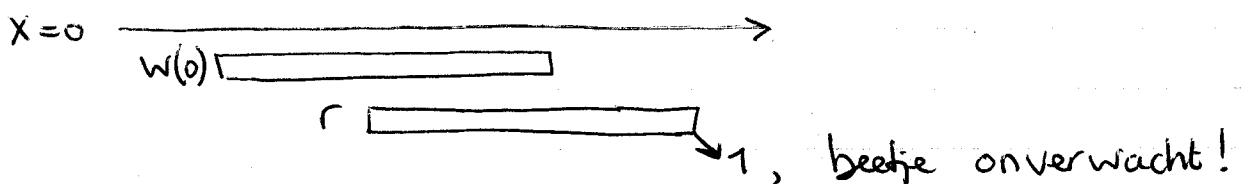
Van uitkomst is alleen bekend:

Als $Read$ niet overlapt met een $write$,
dan geeft hy de waarde van de meest recente $write$.

Overlappende ^{read} $write$ kan alles geven:



Maar ook (!):



③ Multireader Safe.

Om een waarde voor meerdere readers zichtbaar te maken, kun je hem in verschillende SR-registers schrijven. Voor "Safe" is dit genoeg, maar voor atomic niet!

Conventie: ik gebruik Read/Write voor een hoog-nivo primitief, en read/write voor het gebruikte.

Zo maak je een MultiReaderSW safe register.

$s[n]$: SRSW safe registers. n : aantal readers

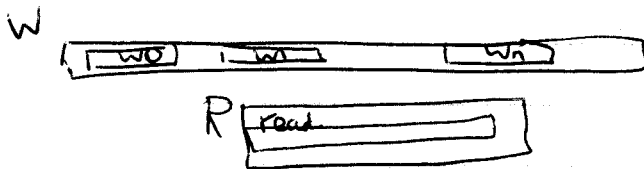
Write(x):

for ($i=0$; $i < n$; $i++$) $s[i].write(x)$

Read voor Reader i :

return $s[i].read$

Een Write bestaat uit n writes, een Read uit één read.



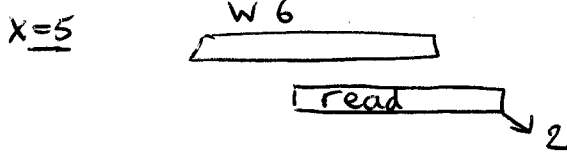
Het gaat hier om Safe dus ik hoef niet te kijken naar een Read die overlapt met een Write.

Als een Read^{door i} niet overlapt met een Write, dan overlapt de read op $s[i]$ (binnen de Read door i) niet met de write op $s[i]$ (binnen de Write).

Dus: die read geeft de laatste waarde die in $s[i]$ is geschreven (want $s[i]$ is zelf safe!), wat de input is van de laatst voltooide Write.

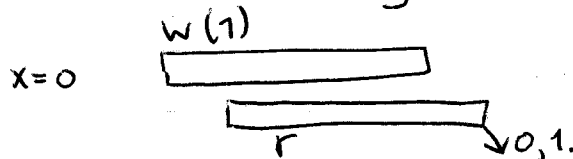
④ Een Regular Bit

Probleem met Safe: als de read overlapt met een write, is de uitkomst willekeurig!



Dit is toegestaan by een "Safe Int".

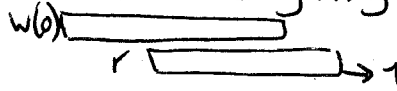
Bij een Safe Bit kan er, by overlap een willekeurige bit uitkomen:



Dat ziet er niet zo schadelijk uit!

Een "willekeurige" bit is al gauw gelijk aan de oude OF nieuwe waarde, en dus acceptabel voor een Regular Bit.

Is een Safe BIT altijd Regular?

Nee, want je kunt een overschrijving met dezelfde waarde doen: $x=0$  is Safe maar niet Regular.

De betreffende write lijkt overbodig...

Dit geeft wel een idee om een Regular BIT te maken van een Safe. De Writer onthoudt de laatst geschreven waarde; een Write van dezelfde waarde doet geen write:

s. Safe bit (gedeeld door Reader en Writer)
last: local voor Writer

Write(x):

```
if (x ≠ last)
{ s.write(x);
  last = x }
```

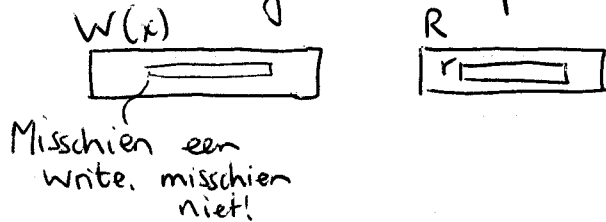
Read

return s.read.

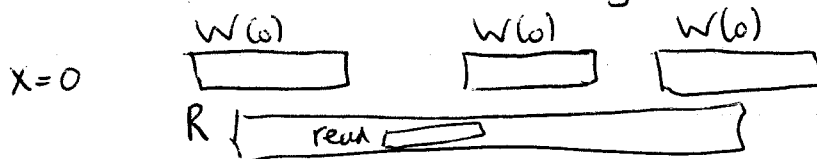
Dit idee werkt voor zowel een MR als SR bit.

Waarom is het nu Regular?

- Stel dat een Read niet overlapt met een Write. De laatste Write (x) laat waarde x achter in s, en omdat er geen overlap is wordt die gereturnt:



- Stel dat een Read overlapt met een aantal PASSIEVE Writes (die niet gaan schrijven).



De passieve Writes hebben allemaal dezelfde waarde, nl. die al in s zit.

Ze doen geen write, dus de read overlapt niet en (omdat s Safe is) retourneert dus de gewenste waarde.

- Stel dat een Read overlapt met een ACTIEVE Write.



Volgens de eisen van Regular, zijn als uit-

komst van de Read, de oude en nieuwe waarde acceptabel: dat zijn 0 en 1.

De read kan alleen die twee waarden geven!

⑤ Regular m-Valued

Met een bit heb je een beetje geluk dat tijdens een "actieve Write" ALLE bitwaarden acceptabel zijn. By een groter bereik wil je dat niet: als een 3 een 0 overschrijft, wil je in een (overlappende) Read, de 3 of 0 zien maar niet de 1 of 2.

Lastig als je het grotere bereik met meerdere bits gaat representeren:

0 :	00
(binair)	3 : 11

De Write(3) is opgebouwd uit operaties op de bits, dus zal ooit een "tussenstand" 01 oid voorkomen, die correspondeert met een niet-reguliere waarde. Je mag geen locks gebruiken om die ontoegankelijk te maken.

We gebruiken daarom geen binaire maar een unaire representatie. Voor een m-waardig register gebruiken we m bits 0₀0₁0₂110₃01101, waarbij de waarde x wordt weergegeven door: $s[x] = 1$, $s[x'] = 0$ voor $x' < x$. Vb stelt dus voor: 3.

Er kan op positie x alleen een 1 staan als de waarde ooit echt x is geweest!

De Writer werkt van hoog naar laag, de Reader andersom:
 r : Regular Bit [m], init 11000

Write(x):

```
r[x].write(1)
for (i = x-1, i ≥ 0, i--)
    r[i].write(0)
```

Read

```
for (int i = 0; i < m; i++)
    if (r[i].read == 1)
        return i
```

Als Reads en Writes niet overlappen is wel duidelijk dat de laatst geWrite waarde wordt geRead.

Let op: een 1 kan alleen corresponderen met een x die ooit echt geschreven is!

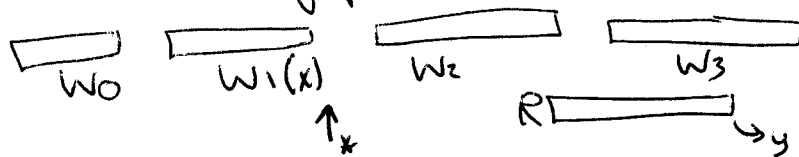
Het is onmogelijk dat een Read alleen nullen ziet en "uit" de array loopt.

Als een Reader bit $r[j]$ gaat lezen

Dan zit er daar of op een hogere positie een 1.

Reden hiervoor: wanneer er een r op 0 wordt gezet, dan zit er op dat moment op een hogere positie een 1.

Dus: de Read geeft altijd een x die ooit geschreven is.



Welke zou dat mogen zijn?
(1, 2, 3 !!)

Zij x de waarde van de laatste Write die afliep voor de Read. Op het aflopen van die Write(*) is $r[x]=1$ en alle lagere = 0. Zij y de waarde, opgeleverd.

Als $y < x$: $r[y]$ is op 1 gezet NA (*),

dus in een Write overlappend met Read.

$y > x$: $r[x]$ is op 0 gezet NA (*),

dus er is een Write overlappend met Read voor een waarde $> x$.

(Argument niet helemaal compleet!)

Dus: Read geeft waarde van laatste voorgaande of overlappende Write.

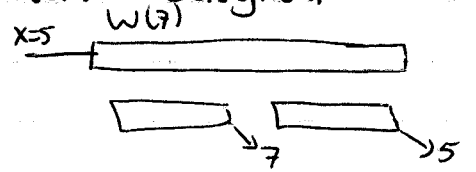
Dwz. Regular.

Deze constructie werkt zowel SR als MR

⑥ Atomic m-Valued

Atomair wordt lastiger omdat je de juistheid van een antwoord niet meer los per Read kunt bekijken, je moet naar een hele executie kijken.

Voor Regular zou het inversie-scenario hiernaast OK zijn. Voor Atomic mag elk van de Reads op zich wel maar de combinatie niet! Het is een inversie.



Idee: als de Reader nou zou kunnen zien dat de 7 "nieuwer" was, dan kon hij die bewaren en bij de tweede Read, de vorige waarde opleveren! Serienummers zijn de oplossing om een Regular register, Atomic te maken.

We gebruiken dus een Regular register dat een combinatie van een waarde (T in het boek) en een getal kan bevatten.

Lokaal voor Writer: sn int

Reader: (y, t) T, int

s : Regular (T, int)

Write(x):

$sn++$

$s.write((x, sn))$

Read:

$(x, s) = s.read$

if $(s > t)$ { $(y, t) = (x, s)$ }

return x

Voorbeelden: $x = 5^{(44)}$ — $W(7)$ $W(7^{45})$

Scen. 1

Read: 5^{44} → 5 is OK

Scen 2

$r \rightarrow 7^{45}$ → 7 is OK

Scen 3

$r \rightarrow 7^{45}$ → 7, $r \rightarrow 5^{44}$ → 5 is OK.

Serienummers voorkomen inversies, maar: alleen lokaal, dwz voor 1 Reader!

Dwz we zijn weer terug bij Single Reader!

Voor Multi-Reader zou je ook inversies in verschillende threads moeten oprangen.

⑦ MultiReader Atomic

Om Multi-Reading te maken uit Single-Reading Atomic, is het niet genoeg een array te maken als by Safe.

Write(x)

```
for(i  
    a[i].write x
```

Read voor th i:

```
return a[i].read.
```

Stel Writer begint, schryft in a[0], dan komt eerst Th 1 een complete Read doen en dan Th 0 een complete. inversie in verschillende readers.

Oplossing: Readers moeten, net als voor SR, hun al opgeleverde waarde bijhouden, maar: zo, dat ze die waarden van elkaar kunnen zien!

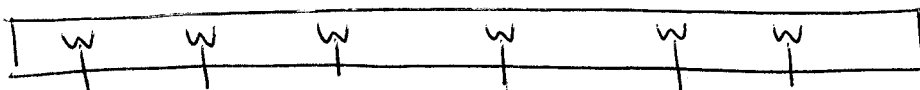
In onze oplossing zullen dus ook SRSW-registers voorkomen, waar van Readers gaan schrijven.

Dit is onvermijdelijk.

Stelling Er bestaat geen
 constructie van Atomic MRSW uit SRSW registers
 waarin geen enkele Reader een write doet
 op de onderliggende SRSW registers

Bewijs. Kyk naar situatie: er zijn 2 Readers, R_0 en R_1 .
 Een Write bestaat uit een reeks "writes", waarvan
 sommige alleen zichtbaar zijn voor R_0 en sommige
 alleen voor R_1 . Maar geen voor beide want het zijn
 SR-registers!

$x=0, W(1)$



Wat gebeurt er als een van de Readers (zeg R_0) een
 complete Read afwikkelt tussen twee van die writes?

R_0 .

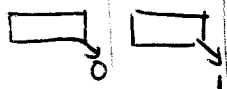


Volgens de eisen van atomair komt hier steeds
 0 of 1, en zonder inversie: dwz na een aantal keren
 0 krijg je een 1 en dan alleen maar 1.

Tussen die twee Reads is er iets veranderd voor R_0 ,
 dus de tussenliggende write was op een SR-register
 waarvan R_0 de reader was. Dus niet R_1 !

Gevolg: als ik ook zijn reeks Reads van R_1 bekijk is
 er een "ander" omslagpunt:

R_1



Er is dus een periode, waarin 1 vd threads de
 nieuwe waarde al kan zien en de andere niet.

Of dat ook gebeurt is voor de ander onzichtbaar ☒

Gebruik een rij $a[n]$ van SRSW-registers waarmee de Writer zijn waarde vertelt aan Reader $i]$
 blok $b[n,n]$ waarin Reader i zijn waarde vertelt aan R_j .

Lokaal voor Writer: sn .

Write(x):

```

    sn++
    for( $i$  )
         $a[i].write(x, sn)$ 

```

Read voor thread i :

```

    ( $x, s$ ) =  $a[i].read$ 
    for( $j$  )
        {( $y, t$ ) =  $b[j, i].read$ 
         if ( $t > s$ ) {  $x, s = y, t$  }
        }
    for( $j$  )
        {  $b[i, j].write(x, s)$  }
    return  $x$ 

```

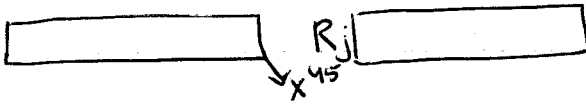
- Alle registers worden SRSW gebruikt:

$a[i]$ heeft writer = Writer en reader = Reader i

$b[i, j]$ heeft writer: Reader j en Writer: Reader i

- Het is wacht-vrij:

Alle constructies van vandaag bestaan uit een vast (begrensd) aantal instructies.

Geen inversie: R_i  R_j

Voordat de eerste R afloopt heeft Th_i al x^{45} geschreven in $b[i, j]$, die wordt dus gezien door de tweede Read.

⑧ Conclusie

Het kan ook nog naar MW, maar dat is meer van hetzelfde, zie boek.

De constructies laten zien:

MWMR m-valued Atomic

is te maken uit

SWSR bit Safe.

Dus: alles wat met registers kan, kun je doen met Safe Bits.

Volgende week gaan we zien:

Je kunt er leuke dingen mee: Snapshot, Counter
maar ook leuke dingen niet: TaS, Ticket.

Dus. Atomaire R/W registers "kunnen" hetzelfde als Safe Bits

Maar Test-And-Set is fundamenteel anders!