

# H2 Locking

15/9/11

- Iwan: run <sup>Critical</sup> Dedicated 1 [ ... ]
- Prak Partner mailen naar Sander

Wo 11/9/13 Unnik-Groen

- ① Wat willen we.
- ② Twee-thread locks
- ③ Multi-Threaded: Bakery
- ④ Aantal Registers
- ⑤ TAS Lock.
- ⑥ Conclusies  
(Kun je mee beginnen!)

## ① Wat willen we

Locking beschermt een object tegen gedeeld gebruik. Het zorgt voor feitelijke sequentialisering en daar mee voor atomiciteit. Het maakt de progress van operaties wel "blocking".

lock ; methode ; unlock.

De locks in C#, bv `lock (this) { ... }`, worden door het OS verzorgd.

Zitten wat "eenmalige" kosten aan om threads buiten verdeling van processor tijd te houden.

"Wat als lock niet vrij? Nogmaals doen, while-NEE! Zit 'in' locken"

Vandaag kijken we hoe je het wachten (locken) kunt regelen zonder OS-hulp: met spinning ofwel busy wait.

## Kun je locks implementeren met RW-atomicity & Fairness

Aannamen:

R/W atomicity (van gedeeld geheugen)

Fairness.

Threads gebruiken locks "well-formed", dwz er zijn stukjes code (Critical Sections) met lock ervoor en unlock erachter. (Geen unlock zonder lock!)

De eisen op de locking:

Safety  
"Eindig  
falsifieerbaar"

Mutual Exclusion: hoogstens 1 thread is kritiek.

No Deadlock : als het lock gevraagd wordt en niet bezet is, wordt het uitgedeeld  
(Globale progres)

No Starvation : als een thread het lock vraagt, dan krijgt die thread een keer het lock mits alle threads het lock teruggeven.  
(Locale progres)

Liveness  
"Oneindig  
falsifieerbaar"

## ② Locks voor Twee Threads

### LockZero

Communicatie dmv gedeelde booleans, voor elke thread een.  
Kijk of de ander bezig is, zo niet: flag aan

Threads  $i=0, 1$ , flag[2]

```
lock() { while (flag[j]) {}  
(voor i)   flag[i] = T  
j = 1 - i }
```

Busy wait.

> Als i kritiek is, is  $flag[i] == T$

```
unlock() { flag[i] = False }
```

Dit is niet veilig! Omdat de lock zelf niet atomair is,  
van wege access naar 2 variabelen.  
Er is geen "uitdeler" van het lock, de threads moeten  
het met elkaar doen.

### LockOne

```
lock() { flag[i] = T  
        while (flag[j]) {}  
      }
```

```
unlock() { flag[i] = F }
```

Safe Sluice:

- Set & Unset eigen flag rond kritiek
- Andere is een keer false tussen set en enter.

Probleem: Deadlock als ze allebei willen.

LockTwo Gebruikt maar één variabele: vic (slachtoffer dat moet wachten)

```
lock { vic = i  
      while (vic == i) {} }
```

```
unlock { }
```

Thread komt pas door while als de ander de vic verzet heeft, maar die wacht dan in de while.  
Dus Mutual Exclusion!

Probleem: Deadlock als er maar eenje wil.

## Peterson Lock

lock      1.  $\text{flag}[i] = \text{True}$   
            2.  $\text{vic} = i$   
            3.  $\text{while} (\text{flag}[j] \wedge \text{vic} == j) \{ \}$

Gedachte. je mag door als LockOne OF LockTwo dat toestaat.

// Zie  $\text{flag}[j] == \text{False}$   
          OF  $\text{vic} == j$

unlock       $\text{flag}[i] = \text{False}$

Dat het aan Mutual Exclusion voldoet, vereist wel bewijs!

Bewering (Mutual Exclusion): Threads 0 en 1 zijn nooit tegelijk kritiek.

Bewijs Stel op enig moment zijn ze beide kritiek.  
Voorafgaand doortoert Th i de lock procedure met  
① moment dat  $\text{flag}[i]$  gezet wordt tot na CS  
② zetten van  $\text{vic}$  op  $i$   
③ constatering dat  $\text{vic} \neq i$  OF andere flag false.

Laat 0 en 1 beide kritiek;  
stel dat  $i$  de thread is die de tweede stap als laatste deed.

Die overschrijft  $\text{vic}$ , dus vanaf dat moment is  $\text{vic} = i$ .



Komt  $i$  door de wacht?

$\text{vic} \neq i$  kan hy niet gezien hebben

$\text{flag}[j] = \text{False}$  kan hy niet gezien hebben want de andere flag is al True vanaf voor het veranderen van  $\text{vic}$ .

Geen van beide is mogelijk, dus: Mutual Exclusion.

Bewering: No Deadlock.

Stel er is een thread in Lock, geen kritiek (ert in unlock).

Bewijs. Beide wachten kan niet, eentje zal  $\text{vic} \neq i$  zien  
Eentje in wacht: kan alleen als flag van andere True, binnen enige tijd zal die op False komen of de ander gaat ook wachten.

## No Starvation

Een wachtende thread wordt sowieso definitief vrij gezet als de ander het lock wil.

Conclusie Locking is te implementeren met R/W atomicity.

### ③ Meer threads: Bakery

Idee van "nummertjes automaat": pak een ticket, wacht tot alle klanten met lager nummer uit het systeem zijn.

Mooist is: gebruik maken van een atomaire incrementeerbare teller!

Statement  $\text{label}[i] = \text{ticket}.$  FetchAndIncrement() garandeert dat ieder een verschillend ticket krijgt.

De atomaire increment is niet te realiseren met R/W atomicity, tenzij je locks gebruikt, maar die gaan we juist implementeren!

Idee van ticket: - lees ieders label, neem 1 hoger.  
- mogelijk gelijke labels als berekening overlapt  
- daarom arbitrage op thread nummer.  
ticket:  $(\text{label}[i], i).$

Bakery Lock gebruikt array van bools Flag init F  
ints label init 0.

lock voor  $i$ :

①  $\text{flag}[i] = T$   
②  $L = 0$ ; for (j) { if ( $\text{label}[j] > L$ ) {  $L = \text{label}[j]$  } }

③ for (k) while (  $\text{flag}[k] \wedge (\text{label}[k], k) < (\text{label}[i], i)$  ) { }

Dus "zie"  $\neg \text{flag}[k]$  OF  $\text{label}_k > \text{label}_i$ .

unlock  
{  $\text{flag}[i] = F$  }

De lock kent nu 3 fasen

① Flag zetten

② label/ticket kiezen

niet atomaire

③ wachtkamer.

① en ② heten samen de doorway, er zit geen wacht in, elke thread komt er in beperkte tijd door.

Wait-free: begrensd aantal stappen van thread, fairness  $\Rightarrow$  dit eindigt.

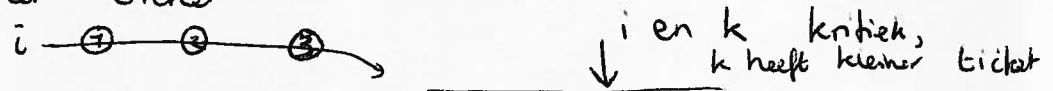
Merk aller eerst op, dat voor elke  $i$ ,  $\text{label}[i]$  niet kan afnemen: elke nieuwe waarde is groter dan de vorige.

Ten tweede: als thread  $i$  kritiek is, is  $\text{flag}[i] == \text{true}$

Ten derde: als twee "label berekeningen" niet overlappen, levert de laatste een groter label

## Stelling Bakery voldoet aan Mutual Exclusion

Thread  $i$  is nooit tegelijk kritiek met een thread  $k$  met een kleiner ticket



Voordat  $i$  kritiek werd, ging  $hy$  in de wachtkamer, waar  $hy$  heeft gezien dat

- $flag[k]$  is False; of
- $k$  heeft groter gelijk ticket.

1<sup>e</sup> geval:  $k$  is met lock begonnen NA dat  $i$  dit zag.  $Hy$  heeft dan zijn label bepaald nadat  $i$  zijn label heeft vastgesteld, dus is  $k$ 's label groter.

2<sup>e</sup> geval:  $k$ 's ticket was op moment (3) al groter gelijk dan  $i$ 's. Dat van  $i$  is niet veranderd, dus die van  $k$  kan alleen maar groter zijn.  $\boxtimes$

Kan  $i$  dan gelijk kritiek zijn met een thread met groter ticket?

NEE, want die kan niet met een kleinere!

### Gelyk ticket?

Leuke van Bakery: er zit een soort eerlykheid in. Als thread  $i$  eenmaal zijn ticket heeft, dan krijgen alle threads die dan met lock beginnen een hoger ticket. Die mogen niet voor  $i$ .

### Twee nadelen

- Groei van labels. Je moet ooit resetten  
Zie boek 2.7
- Veel geheugens:  $\Omega(n)$  ruimte!  
En  $\Omega(n)$  tijd/operaties per thread.

#### ④ Het aantal registers

Leuk aan concurrency: je kunt van allerlei dingetjes bewijzen dat ze onmogelijk zijn, als je precies formuleert wat, en zorgvuldig redeneert.

Voor lockings: als je locking met R/W implementeert voor  $n$  threads, dan heb je  $n$  registers nodig.

(Bakery gebruikt  $2n$  gedeelde geheugens).

Wij pakken het iets bescheidener aan:

Stelling Twee-thread lock kan niet met 1 variabele.

En LockTwo dan? Die had wel Mutex, maar geen No Deadlock.

Probleem van geheugen: waarde erin kan overschreven worden zonder dat het gezien is.

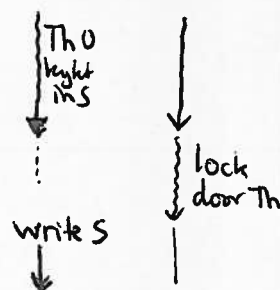
Stel we willen lock implementeren met 1 geheugen  $S$

Lemma A In de lock methode moet je minstens 1x schrijven.  
B. Als Th 1 lock doet (alleen) moet hij door wegens No-D.  
Th 1 lock doet terwijl Th 0 net lock heeft gedaan moet hij wachten dgu Mutex.  
Dus: er moet een andere waarde in  $S$  zitten na de lock door Th 0.

Nu kan, vlak voor de eerste write van Th 0, T1 de gehele lock methode doorlopen.

Na het doorlopen van lock door T1 gaat T0 verder en doet een write: alles wat T1 heeft veranderd in  $S$  is verdwenen!

T0 gaat door alsof er niets gebeurd is:  
schending van Mutex.



Conclusie: voor 2-thread lock zijn minstens 2 reg. nodig.

Boek bewijst: voor 3 threads zijn 3 reg nodig.

Er geldt voor  $n$  threads zijn  $n$  registers nodig.



## ⑤ Het Tas lock

De x86 architectuur staat toe in 1 atomaire stap, een waarde te lezen en schrijven.  
Aan 1 bit heb je genoeg voor locking, met willekeurige veel threads.

Bit: write(0) write(1) read: atomair  
tas : zet op 1, return oude waarde.

(Een geheugen kan dit niet, er is expliciet locking op nivo van machine-instructie nodig).

We gebruiken 1 bit L: 0 - lock is vrij  
1 - lock is bezet

```
lock { while (L.tas == 1) {} }
```

```
unlock { L.write(0) }
```

De Tas heet in C#: Interlocked.Exchange(t, 1)  
werkt op 32b

Er is geen NoStarvation!

## ⑥ Conclusies

- Locking: implementeert atomiciteit van willekeurige operaties, ten koste van: progress is blocking.
- Locking kan met R/W worden geïmplementeerd.
- Geeft wel "busy wait", vaak zal OS-block beter zijn.
- Algoritmen: Peterson (2 threads) Bakery (n threads).
- Sterke instructies zijn beter dan R/W:  
Met Tas kan locken met 1 bit  
Met R/W heb je n locaties nodig.  
Om dit laatste te substantiëren, wil je kunnen bewijzen dat iets met R/W niet kan.

⑦ tryLock? Voorkomen van deadlock by Din. Phil,  
"kijk" of beide vorken vrij zijn, dan locken.

Werkt niet!

"tryLock" is iha niet zo nuttig.

RMW-instr.

x86: load  
① store

②

instructie:  
add -- 1  
werkt op unicode.

⑦

atomaire instr.  
lock; add -- 1.

6++, niet atomaire  
in machine nivo!