

# Les 4: Synchronisatie

If debugging is the process of removing bugs, then programming  
must be the process of putting them in.  
*(Edsger W. Dijkstra)*

best4-1

# Overzicht

- Wat is synchronisatie?
- Software-oplossingen
- Hardware-oplossingen
- Semafoor
- Monitor
- Transactioneel geheugen
- Boodschappen
- Impasses

best4-2

# Doelstelling



- Het in goede banen leiden van de interactie tussen processen of draden
  1. Doorgeven van informatie
  2. Vastleggen van volgorde
  3. Wederzijdse uitsluiting
- Bij foute synchronisatie
  1. Raceconditie ('te weinig synchronisatie')
  2. Impasse ('te veel synchronisatie')

best4-3

Bij processen die met elkaar wensen te interageren, maar zeker bij draden is het belangrijk dat de interactie in goede banen geleid wordt (zie verder). Indien dat niet gebeurt, dan kunnen er allerlei dingen fout lopen. Indien er te weinig gesynchroniseerd wordt, lopen we het risico dat processen elkaars gegevens gaan overschrijven en op die manier corrupt maken. Indien er dan weer te veel synchronisatie is kunnen processen onnodig staan wachten, en zelfs in een impasse terechtkomen.

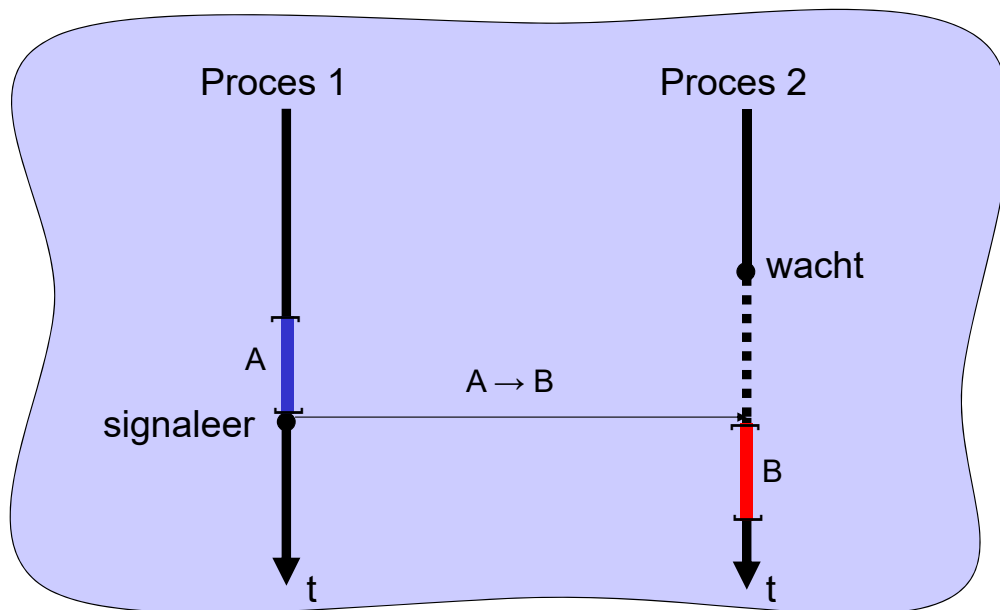
# Doorgeven van informatie

- B.v.: “ls | wc”
- Methoden:
  - Via bestanden, pipes, sockets
  - Via boodschappen (send/receive)
  - Via een blok gemeenschappelijk geheugen
  - Via kernstructuren (b.v. semaforen)
- Bij draden is het gebruik van gemeenschappelijk geheugen het meest voor de hand liggend

best4-4

Een interactie waarbij synchronisatie nodig is, is o.a. het doorgeven van informatie tussen processen of draden. Bij draden zal dat meestal aan de hand van een gemeenschappelijke datastructuur gebeuren. Bij processen kan dat gebeuren door middel van het doorgeven van boodschappen, of door middel van het alloceren van een gemeenschappelijk blok geheugen (shared memory).

# Volgorde vastleggen



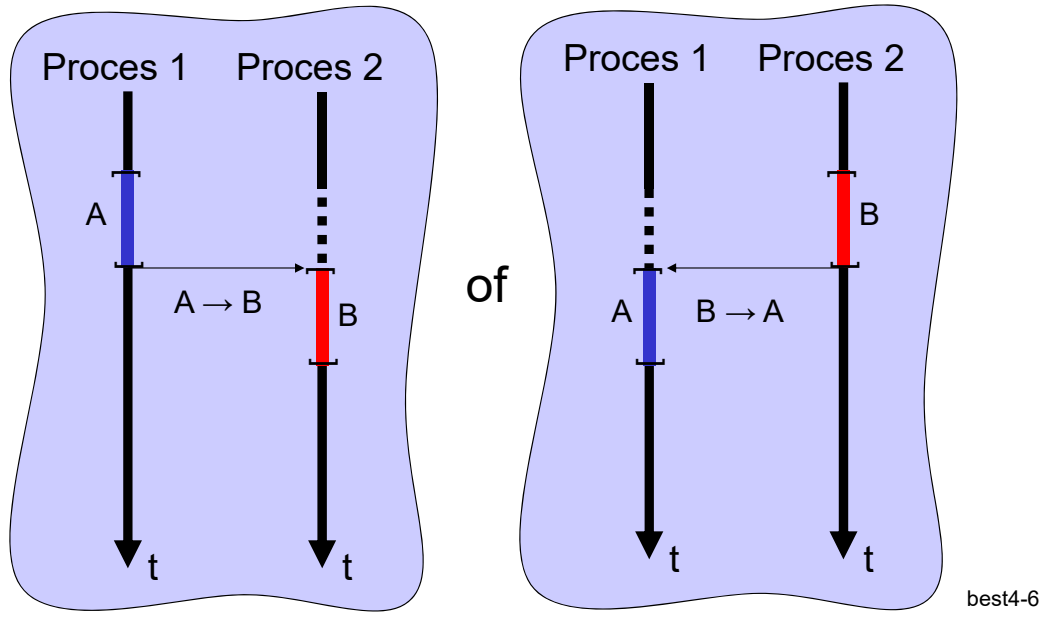
best4-5

Soms wensen processen of draden op elkaar te wachten. Om dit gedrag te implementeren zullen we gebruik maken van synchronisatieoperaties. In de bovenstaande figuur wacht proces 2 op proces 1. De notatie  $A \rightarrow B$  betekent 'A komt voor B' en wordt in het Engels aangeduid als de 'happens before' relatie. Naast  $A \rightarrow B$  zijn er enkel nog afhankelijkheden binnen de processen zelf die geordend zijn. Zo moet 'signaleer' steeds na A komen, moet 'wacht' steeds voor B komen. Merk op dat 'wacht' niet geordend is met 'signaleer'. Het programma legt niet vast in welke volgorde deze moeten voorkomen. In de bovenstaande figuur wordt de indruk gewekt dat 'wacht' voor 'signaleer' komt, maar bij een andere uitvoering kan 'wacht' in de tijd na 'signaleer' komen waardoor er dan uiteraard niet meer hoeft gewacht te worden (geen stippellijn meer).

# Wederzijdse uitsluiting



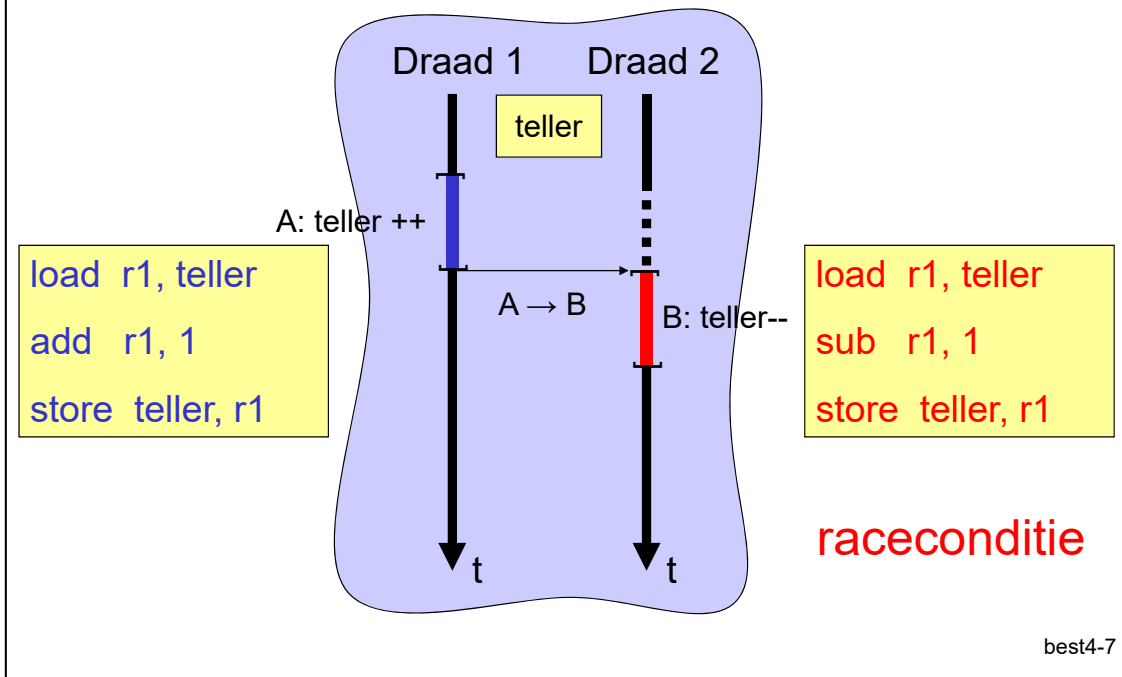
De secties A en B zijn wederzijds uitgesloten



Tenslotte zullen we ook synchronisatieoperaties willen gebruiken om afspraken te maken over het gebruik van gemeenschappelijke systeemmiddelen die niet simultaan kunnen of mogen gebruikt worden. In de bovenstaande figuur kan A voor B komen, of B voor A, maar ze kunnen niet tegelijk uitgevoerd worden (op een multicore) – of de gekleurde sectie in het ene proces mag niet onderbroken worden om de andere gekleurde sectie uit te voeren (op een uncore).

Merk op dat wederzijdse uitsluiting geen vaste volgorde tussen A en B oplegt (en dit in tegenstelling met de vorige dia). Deze extra vrijheid laat meer flexibiliteit bij de uitvoering toe. Wederzijdse uitsluiting is een heel belangrijke synchronisatiemethode om gemeenschappelijke datastructuren adequaat te kunnen beschermen zoals hierna duidelijk zal worden.

# Gebruik van gemeenschappelijke data



Dat het atomair kunnen aanpassen van gemeenschappelijke datastructuren van belang is mag blijken uit het bovenstaande voorbeeld. Twee draden wensen een gemeenschappelijke teller aan te passen (incrementeren en decrementeren).

Ze doen dat aan de hand van de operatoren ++ en --. Op het eerste zicht zou men kunnen denken dat dit op een atomaire manier gebeurt. Afhankelijk van de onderliggende architectuur en van de gebruikte compiler zal dit echter door 1 of meer instructies geïmplementeerd worden. Indien er meer dan 1 instructie gebruikt wordt kunnen er zich heel wat vreemde effecten voordoen zoals blijkt uit de volgende dia.

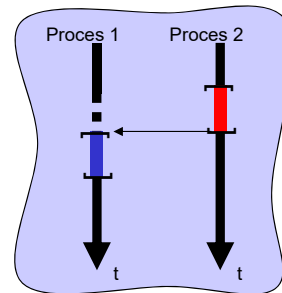
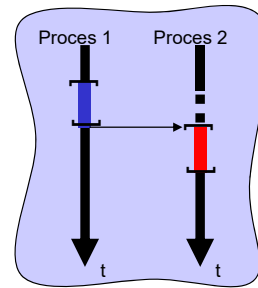
# Raceconditie → fout resultaat

Begin	Volgorde	Einde
teller = 10	L + S L - S *	teller = 10
teller = 10	L + L S - S	teller = 9
teller = 10	L + L - S S	teller = 9
teller = 10	L + L - S S	teller = 11
teller = 10	L L + S - S	teller = 9
teller = 10	L L + - S S	teller = 9
teller = 10	L L + - S S	teller = 11
teller = 10	L L - + S S	teller = 9
teller = 10	L L - + S S	teller = 11
teller = 10	L L - S + S	teller = 11
teller = 10	L L + S - S	teller = 9
teller = 10	L L + - S S	teller = 9
teller = 10	L L + - S S	teller = 11
teller = 10	L L - + S S	teller = 9
teller = 10	L L - + S S	teller = 11
teller = 10	L L - S + S	teller = 11
teller = 10	L - L + S S	teller = 9
teller = 10	L - L + S S	teller = 11
teller = 10	L - L S + S	teller = 11
teller = 10	L - S L + S *	teller = 10

← Juist

Fout

← Juist

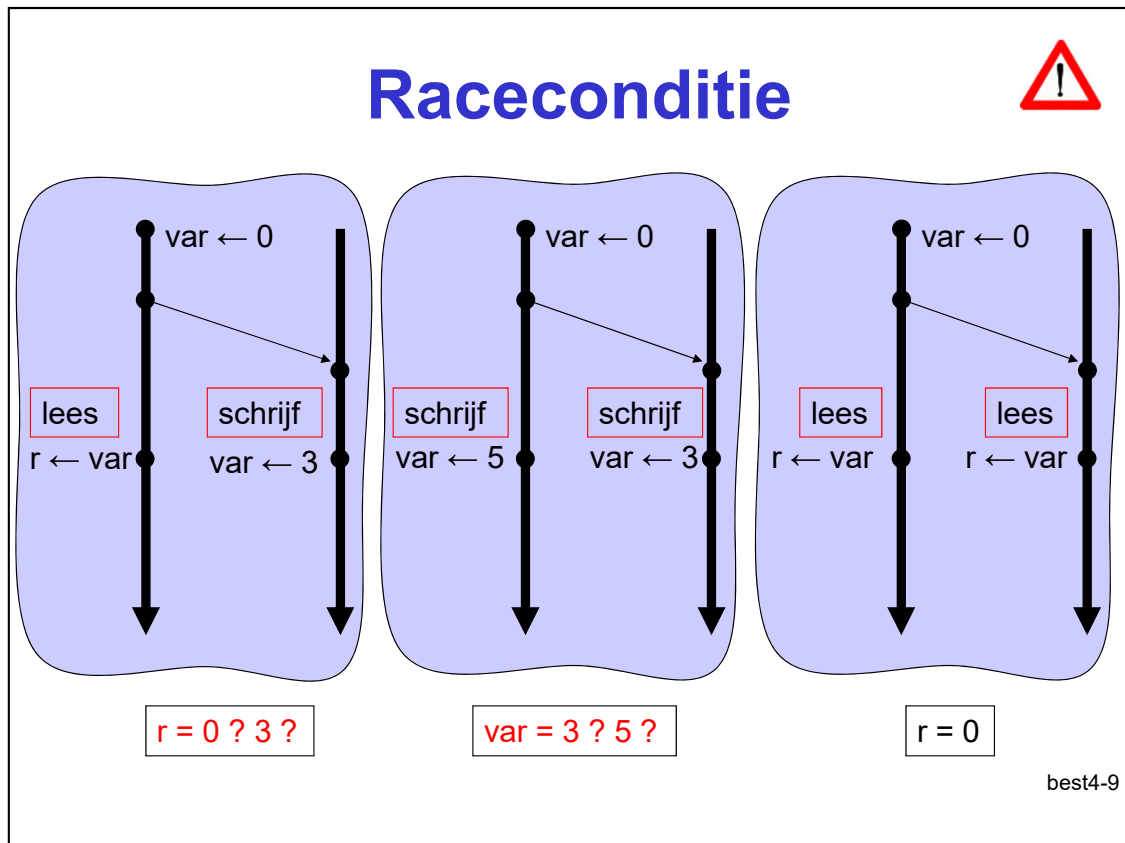


best4-8

Afhankelijk van de manier waarop de twee draden uitgevoerd worden zullen de instructies in één van de volgorden uit de bovenstaande tabel uitgevoerd worden. Er blijken niet minder dan 20 mogelijke uitvoeringen te bestaan, waarvan er slechts 2 correct zijn. Alle andere uitvoeringen blijken een fout resultaat op te leveren. Gelukkig is de kans dat een correcte uitvoering optreedt (gemarkt met een \*) veel groter dan de kans dat een foutieve uitvoering optreedt. Als het noodlot echter toeslaat is het mogelijk dat de uitvoering van een draad op een ongepaste plaats onderbroken wordt door een andere draad en dat er op die manier toch een fout resultaat verschijnt.

De twee correcte uitvoeringen worden gekenmerkt doordat de instructies van één draad vóór of na de instructies van de andere draad uitgevoerd worden. Dit zijn precies de gevallen die door wederzijdse uitsluiting gegarandeerd worden.





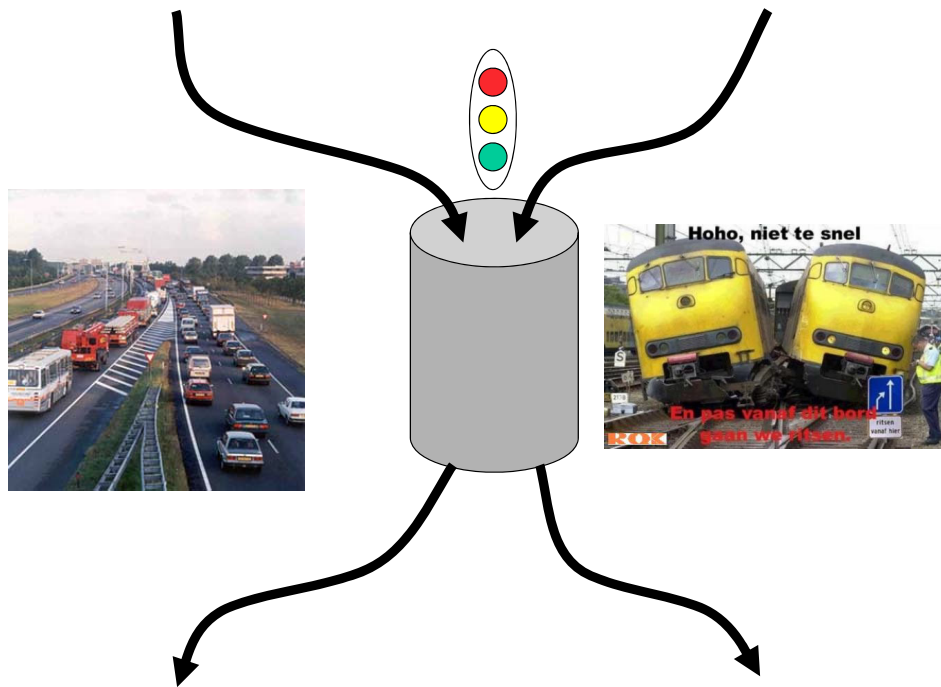
Men spreekt van een raceconditie indien er twee of meer ongeordende (die parallel uitgevoerd kunnen worden) lees/schrijfinstructies zijn waarvan er minstens één een schrijfinstructie is.

Zoals hierboven geïllustreerd wordt, is het resultaat van de uitvoering van deze twee parallelle instructies niet gedefinieerd indien er minstens één schrijfoperatie is. De instructies worden door het programma niet geordend, en men kan dus niet weten in welke volgorde ze zullen uitgevoerd worden. Links zal de veranderlijke  $r$  hetzij de waarde 0 hebben – indien de toekenning in de rechterdraad nog niet gebeurd is, of zal deze de waarde 3 hebben indien de toekenning wel reeds gebeurd is. Meer nog, per uitvoering kan dit verschillend zijn. Het zal afhangen van de procesplanner in welke volgorde de twee instructies zullen uitgevoerd worden. In het midden gaat het over twee schrijfinstructies waarvan de volgorde niet vastligt. In dit geval zal de uiteindelijke waarde van de veranderlijke  $var$  afhangen van welke draad als laatste de schrijfoperatie kan uitvoeren. Rechts gaat het over twee leesinstructies. In dat geval is er geen raceconditie, en is er dus ook geen probleem.

Hierbij dient nog vermeld te worden dat er bij de hierboven geschetste gevallen verondersteld wordt dat we met een sequentieel consistent geheugensysteem te maken hebben, wat betekent dat alle draden in het systeem het resultaat van de uitvoering van instructies van andere draden ogenblikkelijk zien. In de praktijk worden zwakkere consistentiemodellen gebruikt waarbij men zelfs daarvan niet mag uitgaan. In die gevallen is het b.v. Mogelijk dat in het linkergeval ‘ $var \leftarrow 3$ ’ effectief kort vóór ‘ $r \leftarrow var$ ’

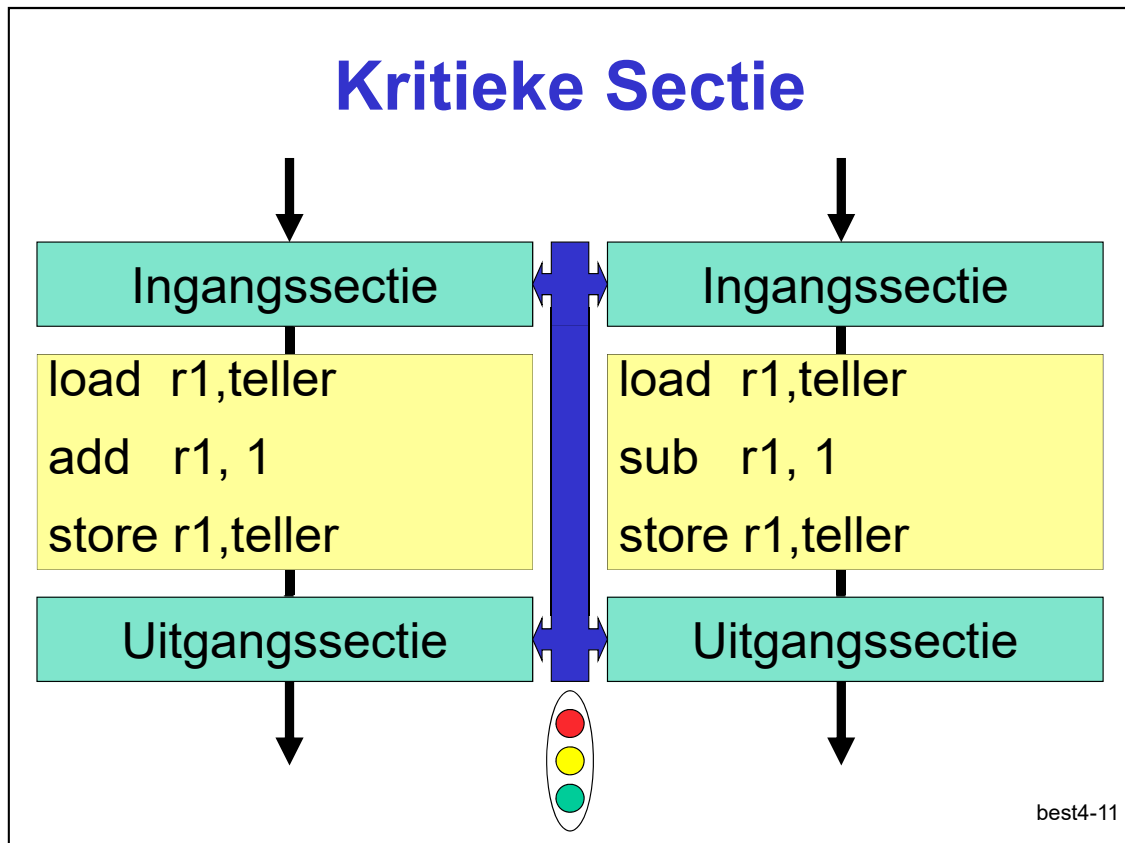
uitgevoerd wordt, maar er toch de waarde 0 in r zit omdat de 3 zich nog in de load-store buffer van de rechtercore bevindt, of omdat de caches tussen de twee cores geen informatie aan elkaar doorsturen.

## Kritieke Sectie



best4-10

Om ervoor te zorgen dat er maar instructies van één draad uitgevoerd kunnen worden, zullen we gebruik maken van een zgn. kritieke sectie (critical section). Een kritieke sectie zorgt ervoor dat er gedurende een bepaald deel van de uitvoering van een proces enkel instructies van precies één draad kunnen uitgevoerd worden. Het is vergelijkbaar met een smalle doorgang waar de auto's één per één doormoeten. Een kritieke sectie is een manier om wederzijdse uitsluiting te implementeren.



Uiteraard dient de wederzijdse uitsluiting tussen de instructies van twee draden niet gedurende de hele uitvoering van de draden van kracht te zijn, maar enkel gedurende die stukken van de uitvoering waarbij ze gemeenschappelijke datastructuren aanpassen. In de praktijk zullen we die gevoelige code laten voorafgaan door een ingangssectie, en afsluiten met een uitgangssectie. De ingangs- en uitgangssecties van de twee draden communiceren met elkaar om af te spreken welke draad er mag uitvoeren, en welke draad er moet wachten.

Deze afspraken noemen we een protocol.

# Protocolvoorwaarden



1. wederzijdse uitsluiting garanderen  
slechts 1 proces per keer in een kritieke sectie
2. vooruitgang garanderen  
een lege sectie moet kunnen betreden worden
3. eindige wachttijden garanderen  
het aantal vóór te laten processen moet eindig zijn

Er mogen geen veronderstellingen over snelheid of over het aantal processors gemaakt worden.

best4-12

Opdat een protocol bruikbaar zou zijn, moet het voldoen aan een drietal voorwaarden.

Het protocol moet **wederzijdse uitsluiting garanderen**, hetgeen betekent dat het maximaal één proces of draad in de kritieke sectie mag toelaten.

Het protocol moet **vooruitgang garanderen**. Dit wil zeggen dat indien er zich geen enkel proces in de kritieke sectie bevindt, en er een proces de kritieke sectie wenst te betreden, er niet oneindig lang mag gewacht worden om de toelating te verlenen.

De **wachttijden moeten eindig zijn**. Indien een proces een kritieke sectie wil betreden en indien deze kritieke sectie in gebruik is, moet er een limiet staan op het aantal keren dat de andere processen de kritieke sectie kunnen verlaten en opnieuw betreden.

Verder mogen er geen veronderstellingen over snelheid of over het aantal processors gemaakt worden, wat wil zeggen dat alle mogelijke combinaties van uitvoeringen mogelijk zijn, zo b.v. ook alle mogelijke uitvoeringsvolgorden van het voorbeeld van het begin van deze les.

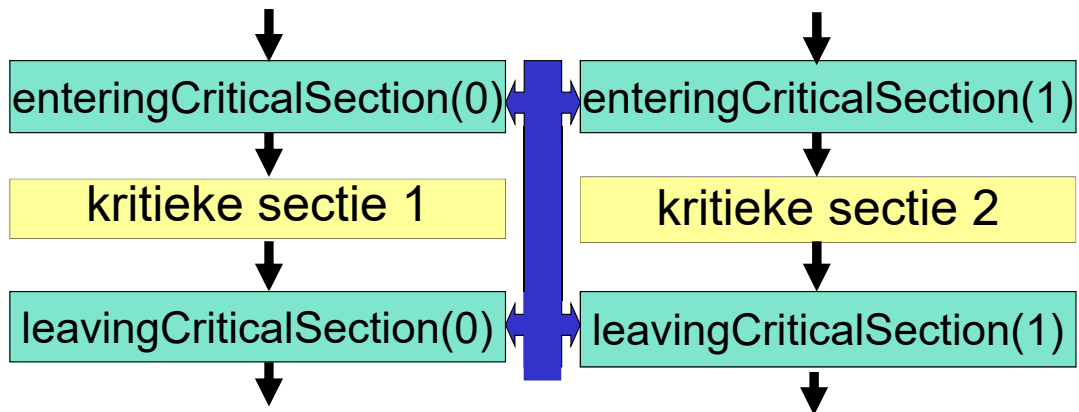
# Overzicht

- Wat is synchronisatie?
- **Software-oplossingen**
- Hardware-oplossingen
- Semafoor
- Monitor
- Transactioneel geheugen
- Boodschappen
- Impasses

best4-13

Er worden nu een aantal software-oplossingen voor het probleem van de kritieke sectie voorgesteld. Deze oplossingen houden geen rekening met de onderlinge snelheidsverschillen van de processen. De enige veronderstelling die zij maken is dat de load, store en test instructies atomair uitgevoerd worden. Dit wil zeggen dat het resultaat van de parallelle uitvoering van twee store-instructies naar dezelfde geheugenlocatie uiteindelijk één van beide waarden zal opleveren, maar geen kruising van de twee.

## Oplossing met twee processen



best4-14

Er volgen nu een aantal softwareoplossingen voor twee processen (genummerd 0 en 1). Indien proces  $n$  de kritieke sectie wenst te gebruiken, dan moet het de methode `enteringCriticalSection(n)` oproepen, en bij het verlaten `leavingCriticalSection(n)`. Deze twee methodes worden hier gedefinieerd als interface die moet geïmplementeerd worden door de diverse softwareoplossingen die nu meteen voorgesteld worden.

# Beurtelings protocol



```
volatile int turn = 0;

void enteringCriticalSection(int t) {
    while (turn != t)
        yield();
}

void leavingCriticalSection(int t) {
    turn = 1 - t;
}
```

Geen vooruitgang  
gegarandeerd

best4-15

Het beurtelings protocol laat de twee draden om beurten de kritieke sectie gebruiken. Om bij te houden wie er als volgende aan de beurt is, heeft het protocol nood aan 1 gemeenschappelijke veranderlijke 'turn'. Deze veranderlijke houdt bij welk proces er als volgende de kritieke sectie mag gebruiken (of aan het gebruiken is). Zolang de betrokken draad niet aan de beurt is, wordt de processor vrijgegeven (met de methode yield). Het sleutelwoord volatile zorgt ervoor dat de veranderlijke niet in een register zal opgeslagen worden. Op die manier blijft hij toegankelijk voor de verschillende draden (registers worden immers niet gedeeld tussen draden).

Dit protocol garandeert wederzijdse uitsluiting, een eindige wachttijd (de andere draad kan maar 1 keer voorgaan), maar kan geen vooruitgang garanderen. Het is immers niet omdat de kritieke sectie vrij is dat een proces ze ook kan gebruiken. Een extra voorwaarde is dat het proces ook aan de beurt moet zijn. Twee draden waarvan de eerste de kritieke sectie éénmaal betreedt (bijvoorbeeld voor de initialisatie van de gemeenschappelijke veranderlijken), en een tweede draad die intensief van deze veranderlijken gebruik maakt (en dus de kritieke sectie verschillende keren wenst te betreden en te verlaten) kunnen dus niet op deze manier gesynchroniseerd worden.

Anderzijds is het wel zo dat dit protocol wel bruikbaar is op plaatsen waar het de bedoeling is dat processen een kritieke sectie altemnerend of in een vaste volgorde betreden (dit kan bijvoorbeeld voorkomen in toepassingen waar men aan gecontroleerde heruitvoering wil doen ten behoeve van het cyclisch debuggen. In dat geval eist men dat bij de heruitvoering de kritieke secties in precies dezelfde volgorde als in de originele



uitvoering betreden en verlaten worden).



## Hoffelijk protocol

```
volatile int flag[2] = { 0, 0 };

void enteringCriticalSection(int t) {
    flag[t] = 1;
    while(flag[1-t] == 1)
        yield();
}

void leavingCriticalSection(int t) {
    flag[t] = 0;
}
```

Geen vooruitgang  
gegarandeerd

best4-16

Om de draden toe te laten om de kritieke sectie te betreden als deze vrij is, laten we elke draad bijhouden of hij zich al dan niet in de kritieke sectie bevindt (of wil begeven) door het aanzetten van de corresponderende vlag. Een draad die de sectie wil betreden kan dan na inspectie van de toestand van de andere draad beslissen of de kritieke sectie vrij is of niet. Dit komt in de praktijk neer op het feit dat een draad steeds de voorrang geeft aan de andere draad (vandaar de naam van het protocol).

Dit protocol garandeert wederzijdse uitsluiting, maar geen vooruitgang. Gesteld dat beide draden hun vlag simultaan op 1 zetten, zullen beide draden voor altijd blijven wachten op het ogenblik dat de kritieke sectie door de andere draad vrijgegeven wordt, ofschoon geen van beide draden zich echt in de kritieke sectie bevindt. Het protocol zal wel een eindige wachttijd garanderen in die zin dat de draad die de kritieke sectie verlaat deze niet opnieuw zal kunnen betreden indien de andere draad te wachten staat.



## Algoritme van Peterson

```
volatile boolean flag[2] = { 0, 0 };  
volatile int turn = 1;  
  
void enteringCriticalSection(int t) {  
    flag[t] = 1; turn = 1-t;  
    while (flag[1-t] == 1 && turn == 1-t)  
        yield();  
}  
  
void leavingCriticalSection(int t) {  
    flag[t] = 0;  
}
```

best4-17

Het algoritme van Peterson is een verbetering van het hoffelijk protocol waarbij nu wel aan de drie protocolvoorwaarden voldaan is (bij een blokkering zal de veranderlijke turn de blokkering opheffen). In deze versie werkt het slechts voor 2 processen, maar het kan uitgebreid worden tot n. Een softwareversie die werkt voor n processen is bv het bakkerijprotocol van Leslie Lamport.

# Overzicht

- Wat is synchronisatie?
- Software-oplossingen
- **Hardware-oplossingen**
- Semafoor
- Monitor
- Transactioneel geheugen
- Boodschappen
- Impasses

best4-18

# Hardware-oplossingen

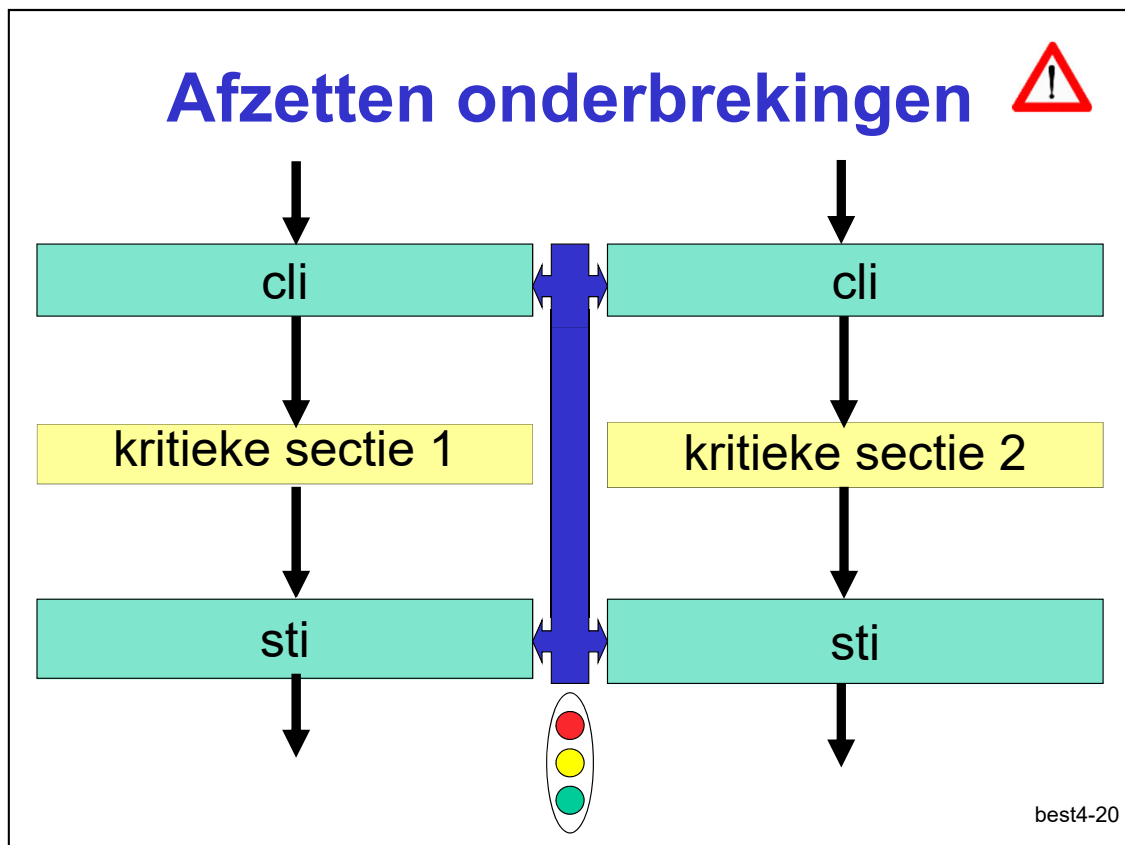
- Afzetten van de onderbrekingen
- Read-modify-write instructies
  - SWAP
  - TAS
  - CAS

best4-19

Naast het afzetten en terug aanzetten van de onderbrekingen (met de cli en sti, die systeemmode vereisen) zijn er ook nog andere machine-instructies die voor de synchronisatie gebruikt kunnen worden – en die in gebruikersmode kunnen uitgevoerd worden.

De drie frequent voorkomende instructies zijn: **SWAP** die toelaat om de inhoud van twee locaties (meestal een geheugenlocatie en een register) te verwisselen, **TAS (test-and-set)** die nagaat of een geheugenlocatie nul is en er nadien 1 in schrijft, en **CAS (compare-and-swap)** die een geheugenlocatie met een register vergelijkt en indien ze gelijk zijn de geheugenoperand overschrijft, zoniet de geheugenoperand kopieert.

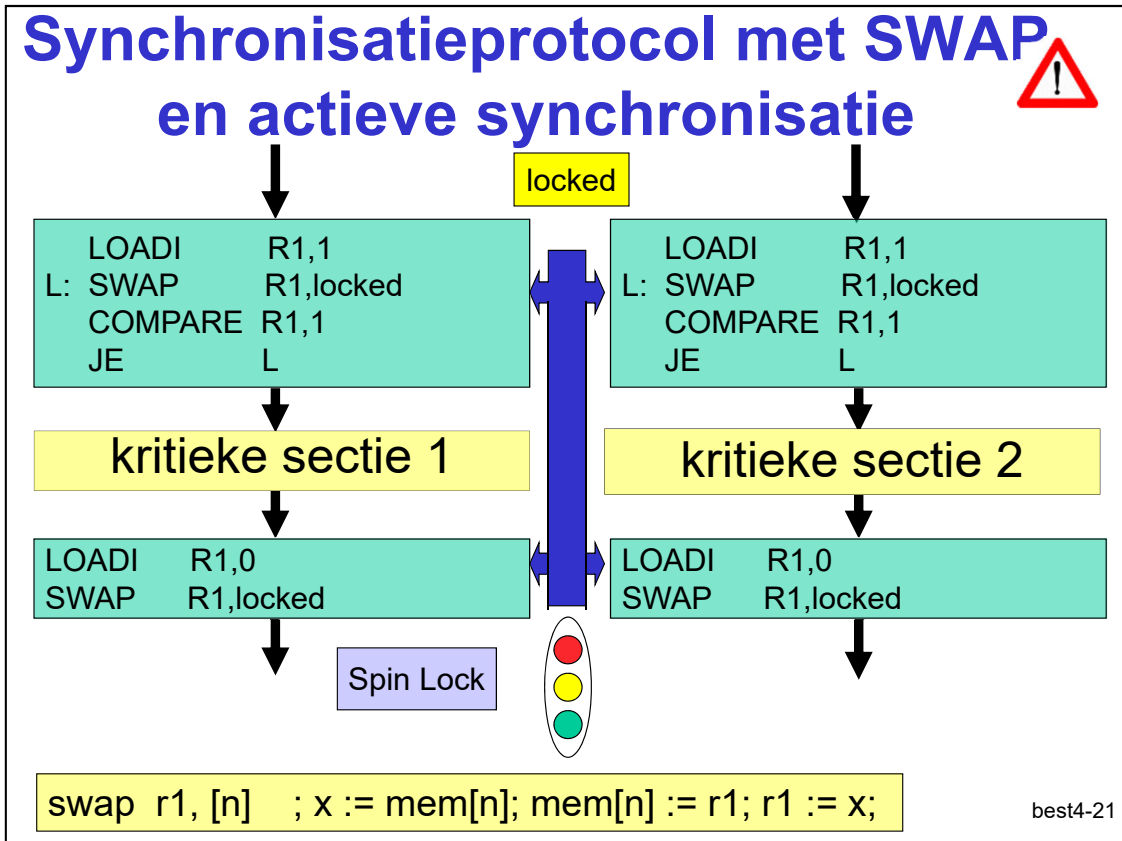
Het kenmerkende van al deze instructies is dat ze op de één of andere manier een load en een store in dezelfde atomaire instructie verenigen door gebruik te maken van een zogenaamde read-modify-write geheugencyclus. Deze cyclus is ononderbreekbaar (zowel voor de CVE, DMA, IO, enz.). Bij SWAP en TAS is de operatie onvoorwaardelijk, bij CAS is deze voorwaardelijk.



De eenvoudigste hardwaremethode om een kritieke sectie te beschermen is het afzetten en terug aanzetten van de onderbrekingen. Deze methode is echter enkel bruikbaar in zeer eenvoudige gevallen. Ze is bruikbaar op monoproductoren voor zeer korte kritieke secties. Aangezien in principe enkel de kern van het besturingssysteem het recht heeft om de onderbrekingsvlag te manipuleren is het gebruik van het onderbrekingssysteem om te synchroniseren beperkt tot de kern van het besturingssysteem. Indien er langere kritieke secties voorkomen is dit niet aanvaardbaar omdat het afzetten van de onderbrekingen ervoor zorgt dat heel wat andere diensten van het besturingssysteem eveneens voor langere tijd afgezet worden. Verder heeft het gebruik van onderbrekingen als nadeel dat alle kritieke secties van alle processen samen gesynchroniseerd worden. Er wordt geen onderscheid gemaakt tussen de individuele kritieke secties.

Op multiprocessors is het afzetten van de onderbrekingen onbruikbaar omdat (i) het afzetten van de onderbrekingen op alle processors veel te lang duurt, en (ii) het onaanvaardbaar is dat andere processors geen onderbrekingen meer kunnen ontvangen omdat er 1 processor wil synchroniseren.

Op een monoproductor garandeert het afzetten van de onderbrekingen wederzijdse uitsluiting, vooruitgang (de cli kan niet blokkeren, een draad die de kritieke sectie wenst te betreden zal dit steeds kunnen zonder te moeten wachten; een draad kan enkel maar in uitvoering gebracht worden indien de kritieke sectie vrij is = indien de onderbrekingen aan staan). Aangezien processen niet kunnen wachten (cli blokkeert nooit), is de wachttijd ook steeds eindig.

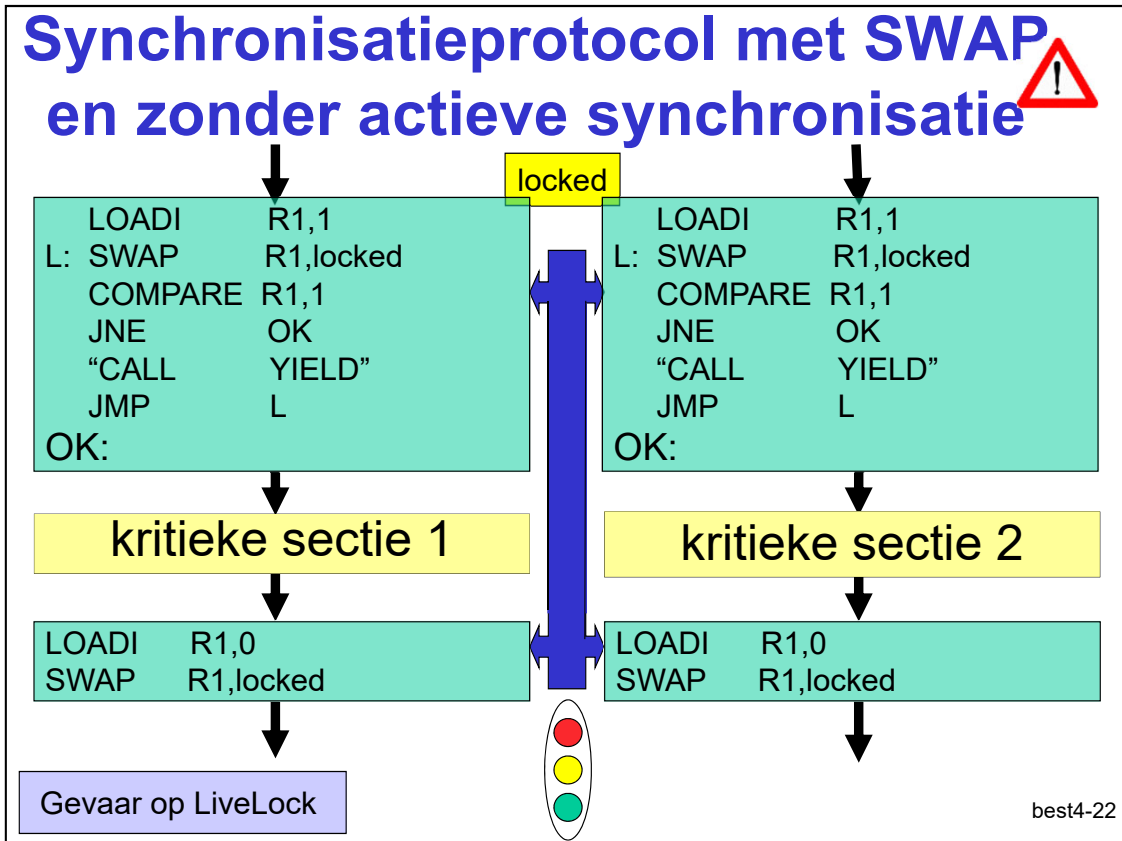


Hier bespreken we het gebruik van SWAP. De andere instructies zoals TAS en CAS leiden tot vergelijkbare oplossingen. De toestand van de kritieke sectie wordt in dit geval bijgehouden door de veranderlijke `locked` (staat op 1 indien de kritieke sectie in gebruik is, zoniet staat deze op 0).

De ingangssectie bestaat uit de SWAP instructie die de inhoud van `locked` verwisselt met de inhoud van register R1 dat de waarde 1 heeft. Naderhand wordt de inhoud van R1 getest. Indien deze 1 is (vorige waarde van `locked`), dan was de sectie bezet door een andere draad en wordt er herbegonnen. Indien de waarde 0 is, dan was de kritieke sectie kennelijk vrij, en kan de kritieke sectie betreden worden.

Doordat SWAP een atomaire instructie is, zal er zelfs bij de simultane uitvoering op verschillende processors toch maar 1 draad de kritieke sectie in handen krijgen (de swap-instructie zorgt ervoor dat er rechtstreeks naar het geheugen gegaan wordt, en niet naar de cache). Het vrijgeven van de kritieke sectie gebeurt door opnieuw 0 in `locked` in te schrijven.

Deze implementatie garandeert wederzijdse uitsluiting en vooruitgang, maar geen eindige wachttijd. Een draad kan na het verlaten van zijn kritieke sectie deze sectie meteen opnieuw betreden zonder dat een ander wachtende (actief synchroniserende) draad de kans krijgt om tussen te komen.



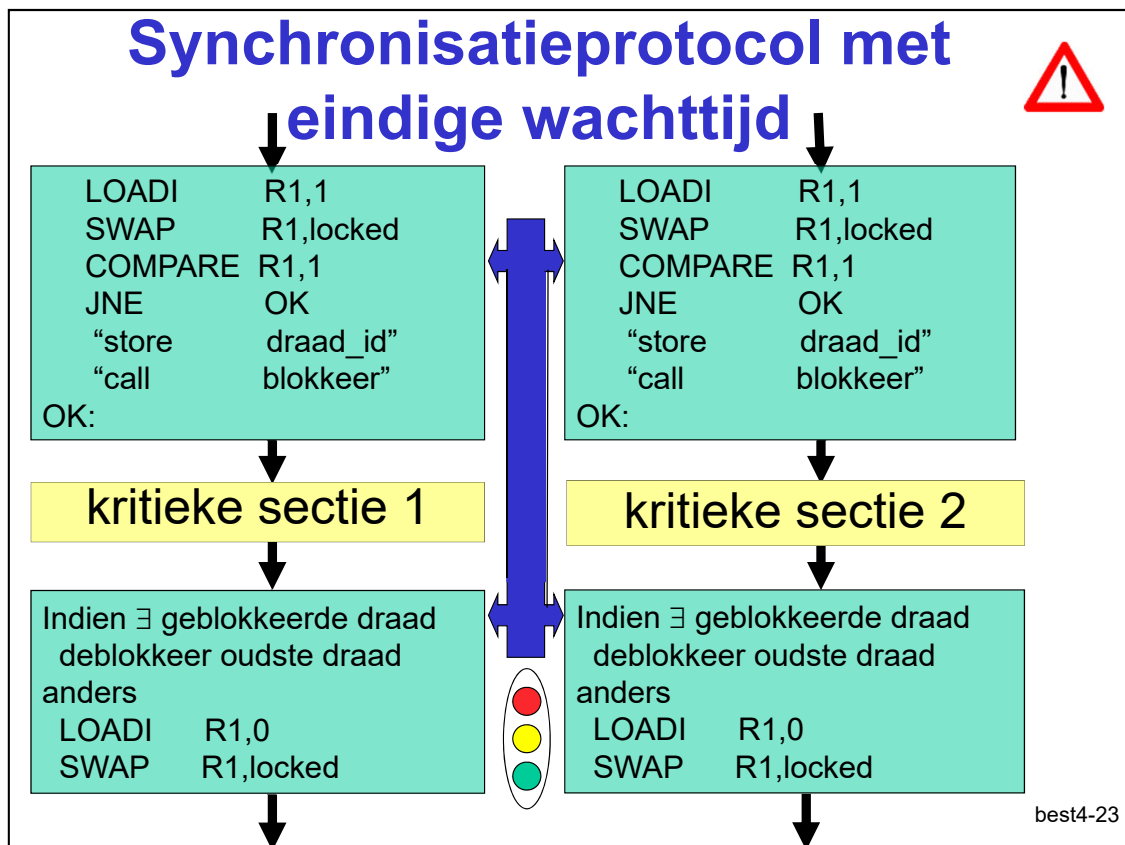
In de plaats van actief te staan wachten op het veranderen van de lock (wat niet zinvol is als men de enige draad in uitvoering is; zolang men zelf aan het uitvoeren is, kan er toch niets gebeuren), kan men de processor ook vrijgeven door de draad naar de klaarlijst te brengen (met yield()). De lock zal dus pas voor het eerst opnieuw gecontroleerd worden nadat de draad opnieuw door de kortetermijnplanner geselecteerd werd. Hopelijk is tegen dan de lock reeds vrijgegeven.

Deze ingreep maakt het wachten efficiënter, maar lost het probleem van de eindige wachttijd niet op (draad 1 kan zijn kritieke sectie nog steeds verlaten en opnieuw betreden zonder dat draad 2 de kans krijgt om tussen te komen). Daarvoor is er nog een bijkomende ingreep nodig.

De hier voorgestelde oplossing kan sporadisch leiden tot zgn livelock. Dit is een situatie waarbij een draad of proces wel degelijk instructies blijft uitvoeren, maar daarmee geen stap vooruit komt in zijn berekening. Dit zou hier b.v. het geval kunnen zijn indien de lock om de een of andere reden op 1 komt te staan en de draden proberen om de kritieke sessie te betreden. Als zij de enige twee processen in de klaarlijst zijn, zullen zij alle cycli opgebruiken, maar ter plaatse staan trappelen. Livelock is moeilijker te detecteren dan een impasse (deadlock) omdat in het geval van een impasse alle processen in de geblokkeerde toestand verzeild zijn geraakt (zie verder) en de machine als



het ware stilvalt op een ogenblik dat men dat niet verwacht. Livelock is een situatie van gas geven met geblokkeerde remmen.



Om ook voor een eindige wachttijd te kunnen zorgen moeten we de processor niet alleen vrijgeven indien we de lock niet meteen kunnen krijgen, maar moeten we ook bijhouden in welke volgorde de draden de lock hebben proberen vergrendelen. Dit is eenvoudig realiseerbaar door een geordende lijst van draden op te bouwen en te blokkeren totdat het signaal komt dat de langst wachtende draad de kritieke sectie kan betreden.

Om dit gedrag te implementeren zullen we de exitsectie ook wat moeten aanpassen. In de plaats van de lock steeds gewoon vrij te geven en dan af te wachten wie de lock opnieuw te pakken krijgt in een open competitie, zullen we de lock nu niet vrijgeven indien er draden staan te wachten. In de plaats hiervan kunnen we de kritieke sectie gewoon doorgeven aan de langst geblokkeerde draad. Enkel indien er geen enkele draad aan het wachten is, wordt de lock terug vrijgegeven. Deze manier van werken garandeert effectief een eindige wachttijd. Merk op dat de ingangssectie nu de lock maar 1 keer test. Indien hij niet vrij is, wordt er gewoon gewacht totdat de lock toegekend wordt aan de draad.

Dit mechanisme kan verfijnd worden door b.v. niet steeds de langst geblokkeerde draad te selecteren, maar b.v. de draad met de hoogste prioriteit.

# Overzicht

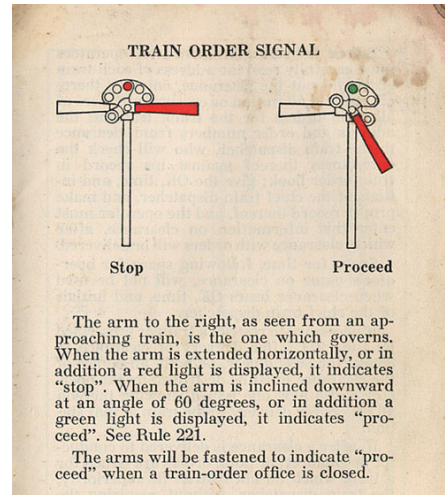
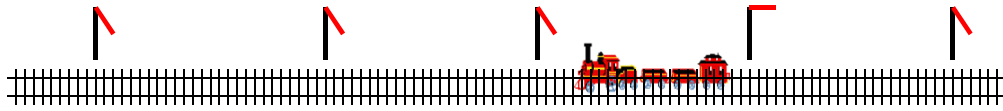
- Wat is synchronisatie?
- Software-oplossingen
- Hardware-oplossingen
- **Semafoor**
- Monitor
- Transactioneel geheugen
- Boodschappen
- Impasses



Edsger Wybe Dijkstra  
(1930– 2002)

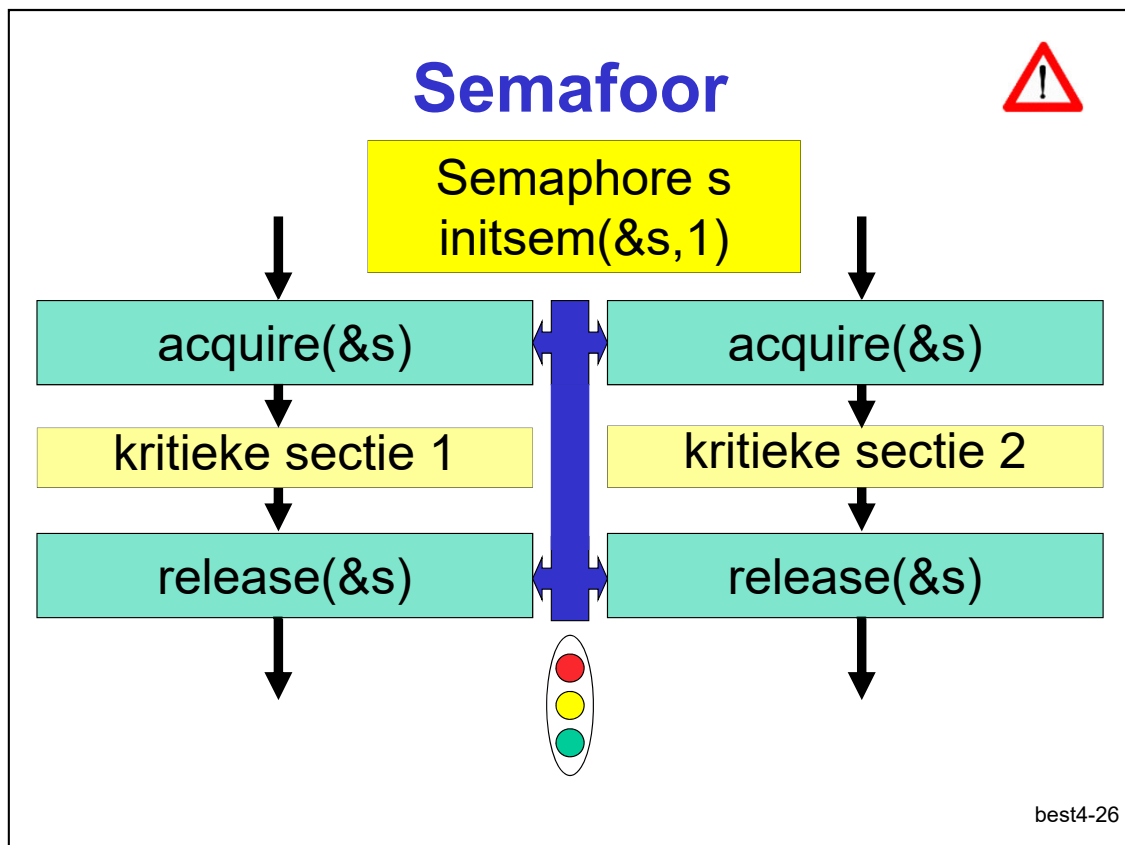
best4-24

# Semafoor (Dijkstra, 1965)



best4-25

Semaforen werden voor het eerst gedefinieerd door E. Dijkstra in 1965. Hij haalde zijn inspiratie bij het treinwezen waar gelijkaardige signalen gebruikt worden om aan te geven of de volgende spoorsectie vrij is of niet. Een spoorsectie mag immers maar 1 trein tegelijk bevatten en is om die reden dus ook een kritieke sectie.



best4-26

Een semafoor is in essentie een geheel getal dat door slechts drie operaties gemanipuleerd kan worden: initialisatie die het getal op 1 zet, acquire die het getal van 1 op 0 probeert te brengen, en release die het getal terug van 0 op 1 zet. De acquire-operatie lukt enkel indien de semafoor oorspronkelijk op 1 staat. Indien de semafoor op 0 staat betekent dit dat een ander proces de semafoor vergrendeld heeft, en er dus moet gewacht worden op het terug vrijgeven van de semafoor.

In de oorspronkelijke teksten van Dijkstra wordt acquire 'P' genoemd (van proberen verlagen), en release 'V' (van verhogen). Dit is één van de weinige plaatsen waar een Nederlandse term (korte tijd) ingang gevonden heeft in het informaticajargon.

De acquire-operatie is bruikbaar als ingangssectie voor een kritieke sectie. Initieel wordt de semafoor op 1 gezet. Bij het betreden wordt ze dan op 0 gezet. Andere draden die een acquire-operatie wensen uit te voeren om dezelfde kritieke sectie te betreden moeten wachten totdat de semafoor terug op 1 gezet wordt door een release-operatie.

# Implementatie binaire semafoor

LOADI R1,0  
L: SWAP R1,sema  
COMPARE R1,0  
JE L

kritieke sectie

LOADI R1,1  
STORE R1,sema

wederzijds uitgesloten  
vooruitgang  
geen eindige wachttijd

best4-27

De semafooroperaties zijn eenvoudig te implementeren d.m.v. de hardware-instructies. Hierboven staat een implementatie van een binaire semafoor door middel van de swap-instructie.

De oplossing die hier voorgesteld wordt werkt met actieve synchronisatie, wat niet bijzonder efficiënt is. Beter is het om bij het botsen op een vergrendelde kritieke sectie, de processor vrij te geven (b.v. door het uitvoeren van een yield()-operatie, of nog beter door een wait()-operatie), en om de release-operatie een signal()-operatie te laten genereren om het geblokkeerde proces terug vrij te geven. Door de geblokkeerde processen in een afgesproken volgorde te bewaren kan men dan ook een eindige wachttijd garanderen. Een dergelijk semafoorobject bevat dus naast de waarde van de semafoor ook nog een lijst van draden die staan wachten op het vrijkomen van de semafoor.

Indien men weet dat men een kritieke sectie maar zeer kort nodig heeft, dan kan men op een uncore steeds overwegen om gewoon de onderbrekingen even af te zetten. Op een multicore kan men bij het ontmoeten van een vergrendelde kritieke sectie beslissen om toch actief te wachten als men weet dat de kritieke sectie maar voor heel korte te vergrendeld zal zijn. Indien de kritieke sectie door een andere parallelle core vergrendeld werd zal deze na een paar iteraties toch vrijkomen. Dat is efficiënter dan de proces in een ander toestand te brengen.

In Solaris spreekt men over adaptieve mutexen (mutex = binaire semafoor). Indien de acquire-operatie moet wachten op een draad die 'in uitvoering' is (op een andere processor), wordt er actief gesynchroniseerd (een zgn. spin lock). Indien de andere draad niet aan het uitvoeren is, heeft wachten geen zin, en wordt er meteen geblokkeerd. Als er niet lang gewacht moet worden is de overhead van de actieve synchronisatie kleiner dan de overhead van blokkeren en terug deblokkeren.

## Binaire semafoor vs. tellende semafoor

- Tellende semafoor (Counting semaphore)
  - Geïnitieerd met een natuurlijk getal
  - Acquire() decrementeert; bij 0 wordt er geblokkeerd
  - Release() incrementeert
- Binaire semafoor = tellende semafoor met initiële waarde = 1
- Mutex = binaire semafoor waarbij acquire() en release() door dezelfde draad moeten gebeuren

best4-28

Naast de klassieke binaire semafoor bestaat er ook nog een tellende semafoor (counting semaphore). Een tellende semafoor kan met een willekeurig positief getal geïnitieerd worden.

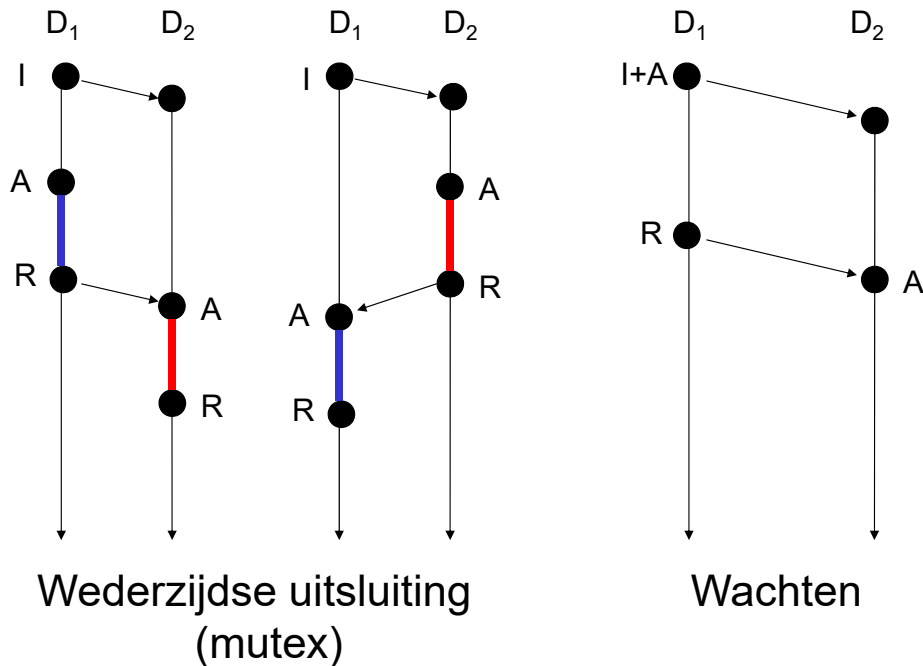
De semantiek van de acquire-operatie is dat de semafoor met 1 verminderd wordt, ongeacht haar waarde, en dat de draad die de acquire-operatie uitvoert blokkeert indien het resultaat van deze bewerking negatief is. De semantiek van de release-operatie is dat de semafoor met 1 verhoogd wordt en indien het resultaat van deze bewerking kleiner of gelijk aan 0 is, er een geblokkeerde draad gedeblokkeerd wordt.

Door een semafoor te initialiseren op 3 zal men dus 3 acquire-operaties kunnen uitvoeren alvorens een acquire-operatie voor het eerst zal blokkeren. Dit garandeert dus geen wederzijdse uitsluiting meer, maar kan wel nuttig zijn voor het beheer van systeemmiddelen waarvan er meer dan 1 ter beschikking is (printers, aantal netwerkconnecties, enz.). Aan de semafoor kan men dan zien hoeveel er nog ter beschikking zijn (indien positief), of hoeveel draden er staan te wachten op het systeemmiddel (indien negatief). Een tellende semafoor die op 1 geïnitieerd wordt gedraagt zich als een klassieke semafoor.

Een mutex is een speciaal type van binaire semafoor waarbij de acquire- en de release-operatie door dezelfde draad moeten uitgevoerd worden (een mutex is dus meer gestructureerd dan een klassieke semafoor). In dat geval worden de acquire- en de release-operaties `mutex_lock` en `mutex_unlock` operaties genoemd. De voornaamste toepassing van een mutex is het beschermen van een kritieke sectie.



# Gebruik Semaforen



best4-29

Het voordeel van het niet-gestructureerd zijn van semaforen is dat acquire()- en de release()-operaties niet gekoppeld moeten voorkomen en dat ze gebruikt kunnen worden voor andere toepassingen dan enkel het beschermen van een kritieke sectie. Er kan bijvoorbeeld een voorrangregeling mee geprogrammeerd worden (zie rechter deel van de figuur). Hiervoor volstaat het dat de draad die de voorrang moet verlenen een acquire-operatie uitvoert op een semafoor die initieel op 0 gezet werd. Hierdoor zal deze draad blokkeren. De draad die voorrang heeft ( $D_1$ ) moet na zijn doortocht een release-operatie uitvoeren op dezelfde semafoor waardoor de geblokkeerde draad ( $D_2$ ) verder kan werken. Indien de draad met voorrang ( $D_1$ ) het synchronisatiepunt reeds voorbij is, zal de draad die voorrang moet verlenen ( $D_2$ ) natuurlijk niet blokkeren.

Een typische toepassing voor dit soort van synchronisatie komt voor bij het wachten op invoer. De draad die de invoer nodig heeft blokkeert op een acquire-operatie en wordt vrijgemaakt door een release-operatie van zodra de invoer voorhanden is. Indien de invoer reeds voorhanden is, moet er natuurlijk niet gewacht worden.

## Problemen met semaforen

- Ongestructureerd – kan leiden tot synchronisatiefouten zoals race condities en impasses.

best4-30

Semaforen zijn eenvoudig te begrijpen en gemakkelijk in gebruik. Semaforen zijn echter niet-gestructureerd. Dit wil zeggen dat men de acquire- en de release-operaties op willekeurige plaatsen in het programma mag gebruiken (ook afzonderlijk) – vergelijkbaar met sprong in een programmeertaal. Het nadeel is dat de minste fout in het gebruik ervan aanleiding kan geven tot synchronisatiefouten (het vergeten van een release-operatie zal elke andere draad verhinderen om de betrokken kritieke sectie te betreden, het vergeten van een acquire-operatie zorgt ervoor dat de kritieke secties niet meer beveiligd zijn, enz.), en dat dit niet gedetecteerd wordt door het besturingssysteem of de compiler van het programma. Er bestaan wel programma's die de zogenaamde 'locking discipline' van een programma kunnen nagaan en merkwaardige wendingen in de synchronisatie van een programma zullen aanduiden (b.v. `lock_lint` onder Solaris, of `Eraser` voor data race detectie op basis van de locking discipline). De meeste van deze programma's zijn echter wel beperkt in het aantal ofwel het type van synchronisatie-operaties of zijn gebaseerd op heuristieken en daarom niet 100% waterdicht. Dit moet ons niet verbazen omdat veel van deze problemen onbeslisbaar zijn.

# Klassieke synchronisatieproblemen

- Eindige buffer
- Lezers-schrijversprobleem
- Dinerende filosofen
  
- Barrier

best4-31

Hierna worden een aantal klassieke synchronisatieproblemen besproken. Deze problemen dienen enkel ter illustratie van het gebruik van synchronisatieprimitieven. Zij komen voor in ongeveer elk tekstboek over besturingssystemen.

Daarnaast zijn er nog een hele resem van andere voorbeelden met ronkende namen zoals ‘het slapende kapperprobleem’, ‘het sigarettenrollerprobleem’, enz.

# Eindige buffer met semaforen

```
#define BUFFER_SIZE 5
volatile Object buffer[BUFFER_SIZE];
volatile int in;
volatile int out;
Semaphore insertmutex;
Semaphore removemutex;
Semaphore empty;
Semaphore full;
```

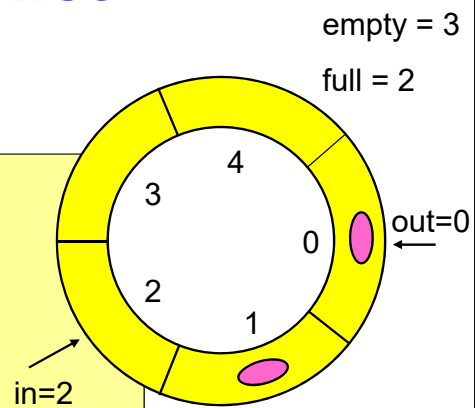
best4-32

Hierboven staan de gegevensstructuren voor een eindige (circulaire) buffer. De circulaire buffer is 5 objecten groot, en houdt intern een wijzer bij naar het eerste en het laatste element. Verder zijn er drie semaforen.

Mutex is een binaire semafoor die gebruikt wordt om de bufferdatastructuur te beschermen. Empty is een tellende semafoor die gebruikt wordt om het aantal lege plaatsen in de buffer bij te houden. Full is een tellende semafoor die gebruikt wordt om het aantal opgeslagen objecten in de buffer bij te houden. Initieel staat deze semafoor op 0. De som van de waarde van empty en full moet buiten de kritieke sectie altijd BUFFER\_SIZE zijn.

# Eindige buffer met semaforen

```
init() {  
    // buffer is initially empty  
    in = 0;  
    out = 0;  
    initsem(&insertmutex, 1);  
    initsem(&removemutex, 1);  
    initsem(&empty, BUFFER_SIZE);  
    initsem(&full, 0);  
}
```



best4-33

Hier worden de verschillende veranderlijken gealloceerd en geïnitieerd. Deze veranderlijken worden ook gebruikt de kritieke secties van de volgende dia, maar men mag ervan uitgaan dat insert() en remove() pas zullen opgeroepen worden na de initialisatie en er ten tijde van de initialisatie maar 1 proces is, en er hier dus geen synchronisatie vereist is.

## Eindige buffer met semaforen

```
insert(Object item) {  
    acquire(&empty);  
    acquire(&insertmutex);  
    // add an item to the buffer  
    buffer[in] = item;  
    in = (in+1) % BUFFER_SIZE;  
    release(&insertmutex);  
    release(&full);  
}
```

```
Object remove() {  
    acquire(&full);  
    acquire(&removemutex);  
    // remove an item from the buffer  
    Object item = buffer[out];  
    out = (out+1) % BUFFER_SIZE;  
    release(&removemutex);  
    release(&empty);  
    return item;  
}
```

best4-34

De semafoor empty wordt hier gebruikt om de methode insert te laten blokkeren indien er geen vrije plaatsen in de buffer meer zijn. Indien er wel nog vrije plaatsen zijn, dan mag er een item aan de lijst toegevoegd worden.

Vervolgens wordt toegang tot de kritieke sectie van insert() verworven om vervolgens het item in de buffer te kunnen plaatsen en de in-wijzer met 1 te verhogen. De in-wijzer verwijst dus steeds naar de eerste lege plaats. Uiteindelijk wordt de kritieke sectie van insert() verlaten en wordt er aangegeven dat er een item bijgekomen is door full met 1 te verhogen. Het gebruik van insertmutex kan misschien overbodig lijken, maar het is best mogelijk dat er twee draden zijn die simultaan iets in de buffer willen schrijven (dit is perfect mogelijk indien er nog twee vrije elementen in de buffer zijn en empty.acquire() lukt voor de beide draden). Het verlagen van empty gebeurt buiten de kritieke sectie omdat we willen vermijden dat de kritieke sectie vergrendeld zou blijven tijdens het wachten op empty. De methode remove werkt op precies dezelfde manier.

Merk op dat het aantal 'empty' eerst verkleind wordt (waardoor empty+full tijdelijk op 4 komen te staan), en dat men pas op het einde aangeeft dat er iets aan de buffer werd toegevoegd. Dit om te vermijden dat remove al een element uit de buffer zou weghalen nog voor het goed en wel opgeslagen werd. Merk op dat de rol van empty en full (communicatie tussen draden, voorrangsregeling) van een totaal andere orde is dan die van mutex (beschermen kritieke sectie)! Empty en full beschermen helemaal geen kritieke sectie.

Met op dat ofschoon de buffer zowel door insert als door remove gebruikt worden, ze toch niet door een gemeenschappelijke kritieke sectie beschermd worden. Dit mag in dit geval omdat door de werking van empty en full, er nooit kan geschreven en gelezen

worden in hetzelfde element van buffer. Verder wordt ‘in’ enkel gebruikt in insert en ‘out’ enkel in remove.

# Lezers-schrijversprobleem: Interface

```
void acquireReadLock();  
void acquireWriteLock();  
void releaseReadLock();  
void releaseWriteLock();
```

best4-35

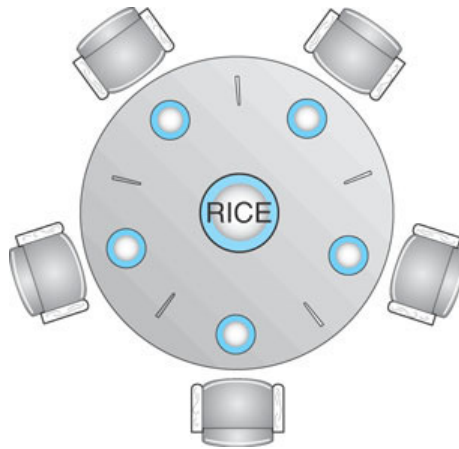
Het gebruik van een mutex om een kritieke sectie te beschermen is soms te sterk. Sommige operaties dienen effectief serieel uitgevoerd te worden, maar andere kunnen perfect parallel uitgevoerd worden. In de praktijk moeten operaties slechts beschermd worden indien er in de verzameling van parallel uit te voeren instructies minstens één instructie is die schrijft. Parallele leesoperaties zijn dus perfect aanvaardbaar.

Deze eigenschap wordt uitgebuit bij **lezers-schrijverssynchronisatie**. Hierbij synchroniseert men in functie van de operatie die men wenst uit te voeren (lezen of schrijven). Van de leesoperaties kunnen er verschillende simultaan toegelaten worden tot de kritieke sectie, van de schrijfoperaties slechts 1. Hierboven staat de interface die men kan gebruiken om lezers-schrijverssynchronisatie te implementeren.

Ofschoon op het eerste zicht eenvoudig, kunnen er zich ook bij een functioneel correcte implementatie problemen van eindige wachttijd voordoen. Indien b.v. omwille van de efficiëntie steeds maar bijkomende lezers toegelaten worden zolang er lezers actief zijn (verhogen van het parallelisme), dan kunnen bij een niet-aflatende stroom van lezers de schrijvers verhongeren. Anderzijds kan de strategie dat het zich aandienen van één schrijver ervoor zorgt er geen bijkomende lezers meer toegelaten worden (om de schrijver zo snel mogelijk te kunnen bedienen) leiden tot het zelfde fenomeen voor lezers. In de praktijk zal men een goed uitgebalanceerde strategie moeten kiezen om de globale prestatie te maximaliseren.



# Dinerende filosofen



Semaphore chopStick[5];

best4-36

Het probleem van de dinerende filosofen bestaat uit  $n$  filosofen (hier 5) die afwisselend rijst eten en denken. Om te kunnen eten hebben ze de twee stokjes nodig die links en rechts van hun bord liggen. Een filosoof mag slechts één stokje per keer nemen. Een stokje dat in gebruik is mag niet afgenomen worden. Van zodra de filosoof twee stokjes heeft kan hij beginnen eten. Na het eten legt hij de twee stokjes terug op tafel.

Dit probleem illustreert wat er allemaal fout kan lopen bij het oplossen van dergelijke problemen. Aangezien de stokjes hier het schaars systeemmiddel zijn, wordt er een binaire semafoor bijgehouden per stokje.

# Dinerende filosofen

```
void philosopher(int i) {  
    while (true) {  
        // get left chopstick  
        acquire(&chopStick[i]);  
        // get right chopstick  
        acquire(&chopStick[(i + 1) % 5]);  
        eating();  
        // return left chopstick  
        release(&chopStick[i]);  
        // return right chopstick  
        release(&chopStick[(i + 1) % 5]);  
        thinking();  
    }  
}
```

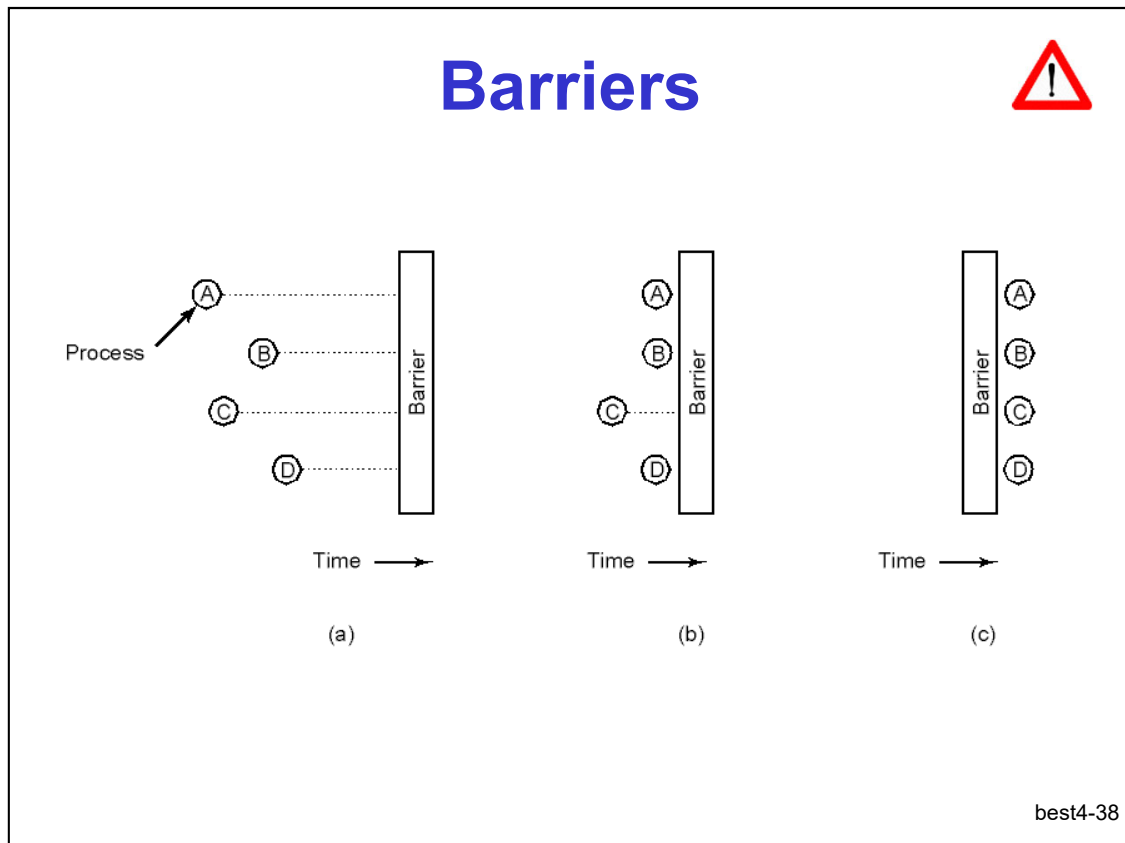
best4-37

Hier ziet U een foute oplossing van het probleem van de dinerende filosofen.

Deze oplossing garandeert dat twee naburige filosofen nooit gelijktijdig kunnen eten, maar met deze oplossing kan het wel grondig fout lopen. Als alle filosofen tegelijk hun linkerstokje nemen, dan zal geen van de filosofen erin slagen om nog een rechterstokje te pakken te krijgen, dan kan komen ze allemaal om van de honger.

Men kan een aantal oplossingen bedenken voor dit probleem:

1. Men kan slechts 4 filosofen toelaten aan tafel. Dan zal er altijd minstens 1 kunnen eten.
2. Men kan eisen dat de filosoof beide stokjes simultaan moet opnemen (in een kritieke sectie). Indien er maar 1 stokje is, mag dit niet opgenomen worden.
3. Men kan het gedrag van de filosofen wat minder gelijk maken. De even filosofen kunnen b.v. eerst het linkerstokje nemen en dan het rechterstokje, en de oneven filosofen net andersom.



Een semafoor laat toe dat 1 draad op een andere wacht. Er kunnen zich echter situaties voordoen waarbij verschillende draden op elkaar moeten wachten. Hiervoor kan men dan een zgn barrier gebruiken. Een **barrier** is een synchronisatiepunt tussen verschillende draden. De draden moeten bij het synchronisatiepunt wachten totdat alle draden die door de barrier gesynchroniseerd worden zijn aangekomen, en kunnen pas dan hun eigen berekening weer verder zetten.

Dit kan b.v. gebruikt worden bij de parallelle uitvoering van iteratieve algoritmen. Elke iteratie wordt b.v. parallel uitgevoerd (b.v. 4 draden die elk 25% van de data bewerken), maar de volgende iteratie mag niet starten alvorens de vorige volledig afgelopen is. Een barrier kan dan gebruikt worden om het einde van elke iteratie te synchroniseren met het begin van de volgende.

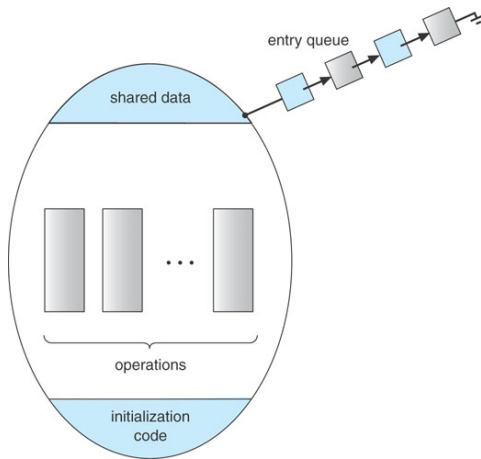
Een barrier kan geïmplementeerd worden aan de hand van een voorrangsregeling met een matrix van semaforen. Bij aankomst laat een draad aan alle andere draden weten dat hij aangekomen is (met een `release()`). Naderhand begint hij te wachten totdat hij ditzelfde signaal van alle andere draden ontvangen heeft (met een `acquire()`).

# Overzicht

- Wat is synchronisatie?
- Software-oplossingen
- Hardware-oplossingen
- Semafoor
- **Monitor**
- Transactioneel geheugen
- Boodschappen
- Impasse

best4-39

# Monitor



```
monitor monitor-name
{
    // variable declarations

    procedure p1(...) {
        ...
    }

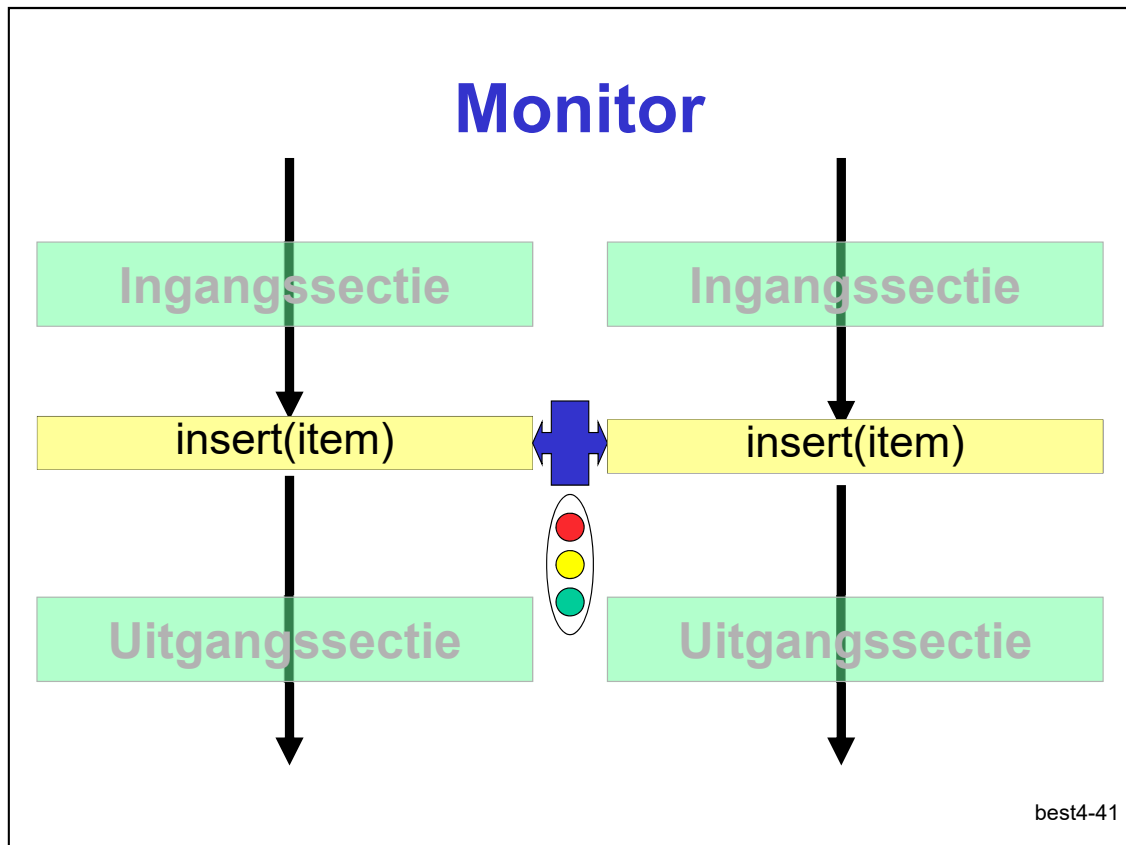
    procedure p2(...) {
        ...
    }
}
```

best4-40

Een monitor bestaat uit een aantal private gegevens en een aantal publieke methoden die deze gegevens kunnen manipuleren. Kenmerkend voor een monitor is dat er van de publieke methoden op elk ogenblik slechts 1 actief kan zijn per monitor. Indien een tweede draad een routine van dezelfde monitor wenst op te roepen zal deze oproep blokkeren totdat de monitor vrijgegeven wordt. Men kan stellen dat alle methoden van een monitor dus behoren tot dezelfde kritieke sectie. De monitorroutines hebben enkel toegang tot de veranderlijken die in de monitor zelf gedefinieerd werden. Verder hoeft er niet meer gesynchroniseerd te worden. Alle synchronisatie is impliciet.

Afhankelijk van de implementatie kan een monitormethode al dan niet andere monitorroutines oproepen. In het geval waarbij dit wel het geval is, gaat men ervan uit dat de monitor vergrendeld werd door de eerste oproep, en kan de monitorroutine zelf andere monitorroutines oproepen. In het geval waarbij dit niet mogelijk is, zal een monitorroutine blokkeren indien ze een andere monitorroutine wenst op te roepen (recursie in de monitor is in dit geval evenmin mogelijk).

Een eenvoudige implementatie van een monitor bestaat erin om per monitor een afzonderlijke mutex of semafoor te definiëren die bij het begin van een monitorroutine vergrendeld wordt, en terug vrijgegeven wordt bij het verlaten ervan. Deze implementatie laat niet toe dat monitorroutines elkaar oproepen. Indien dat wel mogelijk gemaakt dient te worden, kan men b.v. twee varianten van de monitorroutine voorzien: een private routine die de semafoor niet meer vergrendelt, een publieke die dat wel doet.



De ingangs-en uitgangssectie van de kritieke secties zijn in het geval van een monitor zo goed als transparant. Dit reduceert aanzienlijk de kans op het maken van synchronisatiefouten.

## Eindige buffer met monitor

```
Monitor {  
    #define BUFFER_SIZE 5  
  
    Object buffer[BUFFER_SIZE];  
    int in, out, count;  
  
    init() {  
        in = 0; out = 0; count = 0;  
    }  
  
    insert(Object item) {  
        if (count < BUFFER_SIZE) {  
            count++; buffer[in] = item; in = (in + 1) % BUFFER_SIZE;  
        } else throw(exception("buffer vol"));  
    }  
  
    Object remove() {  
        if (count > 0) {  
            Object item = buffer[out];  
            count--;  
            out = (out + 1) % BUFFER_SIZE;  
            return item;  
        } else throw(exception("buffer leeg"));  
    }  
}
```

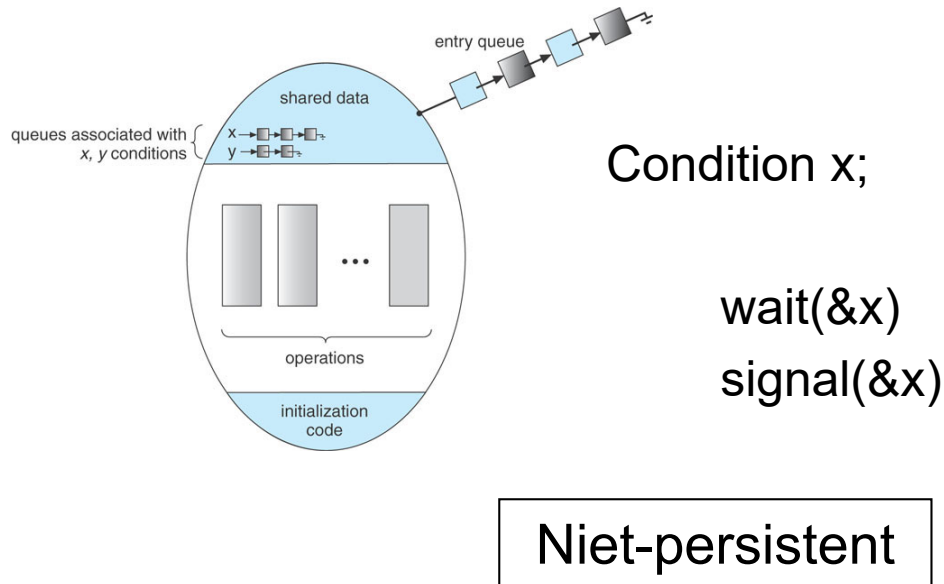
In pseudo-C-stijl

best4-42

De eindige buffer met een monitor zou er als hierboven kunnen uitzien. Alle veranderlijken (buffer, in en out) zijn nu private velden van de monitor en insert en remove zijn nu publieke methoden die door de draden kunnen opgeroepen worden. Deze oplossing is echter niet elegant. Indien de toestand van de buffer niet toelaat dat er een insert of remove operatie uitgevoerd wordt, wordt er een exceptie gegenereerd i.p.v. gewoon te blokkeren.

Het is duidelijk dat we nood hebben aan een manier om in de monitor zelf te wachten totdat er aan de voorwaarde om verder te kunnen rekenen voldaan is.

# Conditieveranderlijken



best4-43

Conditieveranderlijken zijn signalen waarop men kan blokkeren met `wait()`. Een geblokkeerd signaal kan men vrijgeven door `signal()` uit te voeren. Dergelijke signalen zijn **niet-persistent**. Daarmee wordt bedoeld dat indien men een signaal uitstuurt waarop niemand staat te wachten, dit signaal onherroepelijk verloren gaat. Daarnaast bestaan er persistente signalen zoals b.v. semaforen. Indien men een release-operatie uitvoert gaat deze informatie niet verloren (de semafoor wordt verhoogd), zelfs indien er geen geblokkeerde acquire-operatie staat te wachten.

Indien er meer dan 1 draad staat te wachten op een signaal kan `signal()` hetzij 1 wachtende draad deblokkeren, hetzij allemaal. Dit hangt af van de gekozen implementatie.



# Eindige buffer met monitor en condities



```
Monitor {
#define BUFFER_SIZE 5
Object buffer[BUFFER_SIZE];
int in, out, count;
Condition notempty, notfull;

init() {
    in = 0; out = 0; count = 0;
    initcond(&notempty); initcond(&notfull);
}

insert(Object item) {
    while (count == BUFFER_SIZE) wait(&notfull);
    count++; buffer[in] = item; in = (in + 1) % BUFFER_SIZE;
    if (count == 1) signal(&notempty);
}

Object remove() {
    Object item;

    while (count == 0) wait(&notempty);
    count--; item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    if (count == BUFFER_SIZE-1) signal(&notfull);
    return item;
}
}
```

signal(&c)  
vs.  
broadcast(&c)

In pseudo-C stijl

best4-44

Binnen de monitor blokkeren (erbuiten kan niet omdat we daar de waarde van count niet kunnen kennen) moet op een voorzichtige manier gebeuren omdat we de monitor moeten vrijgeven alvorens te blokkeren. De wait-operatie op een conditieveranderlijke zorgt ervoor dat de monitor vrijgegeven wordt alvorens te blokkeren, en bij het deblokkeren van de wait moet de monitor opnieuw vergrendeld worden.

Bij het sturen van een signaal kunnen er zich twee gevallen voordoen. Ofwel kan de draad die de signal-operatie uitvoert blokkeren (met tijdelijke vrijgave van de monitor) zodat het gedeblokkeerde proces kan verder gezet worden en de monitor verlaten. Het proces dat het signaal genereerde kan dan naderhand zijn eigen berekening in de monitor verder afmaken (Sir Tony Hoare, 1974). Per Brinch Hansen (1975) stelde voor om de signal-operatie als laatste operatie in de monitorroutine op te nemen waardoor de routine de monitor automatisch verlaat bij het geven van een signaal (in de taal concurrent Pascal).

Ofwel kan de draad die het signaal gegeven heeft gewoon verder uitvoeren, zal deze op termijn de monitor verlaten en op die manier het gedeblokkeerde proces de kans geven om de monitor opnieuw te betreden en verder te rekenen. In dat geval is er echter geen garantie dat het gedeblokkeerde proces meteen na het proces uitgevoerd wordt dat het signaal genereerde, waardoor er aan de voorwaarde om te deblokkeren misschien niet langer voldaan is als de draad begint uit te voeren. Dat probleem kan men oplossen door de test in een lus te plaatsen waardoor na het deblokkeren van de wait-routine de voorwaarde nogmaals getest wordt. Indien er aan de voorwaarde niet langer voldaan is, dan blokkeert de draad opnieuw, indien wel, dan wordt de rest van de monitorroutine uitgevoerd.

```

#define BUFFER_SIZE 5
volatile Object buffer[BUFFER_SIZE];
volatile int in, out, count;
Condition notempty, notfull;
Semaphore mutex;

init() {
    in = 0; out = 0; count = 0; buffer = new Object[BUFFER_SIZE];
    initcond(&notempty); initcond(&notfull); initsem(&mutex, 1);
}

insert(Object item) {
    acquire(&mutex);
    while (count == BUFFER_SIZE) { release(&mutex); wait(&notfull); acquire(&mutex); }
    count++; buffer[in] = item; in = (in + 1) % BUFFER_SIZE;
    if (count == 1) signal(&notempty);
    release(&mutex);
}


Object remove() {
    Object item;
    acquire(&mutex);
    while (count == 0) { release(&mutex); wait(&notempty); acquire(&mutex); }
    count--; item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    if (count == BUFFER_SIZE-1) signal(&notfull);
    release(&mutex);
    return item;
}

```

## Eindige buffer met Semaforen & condities

notfull.signal()

Lost signal race



Niet persistent

best4-45

Bovenstaande code tracht de methoden insert en remove wederzijds uitgesloten te maken door gebruik te maken van een binaire semafoor om het lichaam van de methoden tot dezelfde kritieke sectie te laten behoren. Aangezien er zowel bij insert als bij remove moet kunnen gewacht worden, wordt er gebruik gemaakt van een conditieveranderlijke waarop gewacht wordt. Uiteraard moeten we erop toezien dat in dat geval de semafoor vrijgegeven wordt en terug vergrendeld wordt. Anders zal er aan de oorzaak van het wachten niet veel veranderd kunnen worden door de andere draden.

De hierboven getoonde aanpak faalt echter: tussen het vrijgeven van de semafoor (`mutex.release()`) en het blokkeren van de draad is het immers mogelijk dat een andere draad de inhoud van de buffer aanpast (en het bijhorende signaal genereert), en op die manier het wachten overbodig maakt. Aangezien de signalen die hier gegenereerd worden met `signal()` niet persistent zijn, zal de draad die nog niet geblokkeerd is, het signaal niet horen, en uiteindelijk blokkeren op een signaal dat misschien nooit meer gegenereerd zal worden. Het zou bovendien kunnen dat deze situatie zich slechts hoogst uitzonderlijk voordoet waardoor deze moeilijk op te sporen is.

Men spreekt in dit geval over een ‘lost signal race’. Er is voor dit probleem eigenlijk maar één afdoende oplossing: het vrijgeven van de semafoor en het blokkeren van de draad moeten atomair gebeuren.

```

#define BUFFER_SIZE 5
volatile buffer[BUFFER_SIZE];
volatile int in, out, count;
Condition notempty, notfull;
Semaphore mutex;

init() {
    in = 0; out = 0; count = 0;
    initcond(&notempty); initcond(&notfull); initsem(&mutex, 1);
}

insert(Object item) {
    acquire(&mutex);
    while (count == BUFFER_SIZE) wait(&notfull, &mutex);
    count++; buffer[in] = item; in = (in + 1) % BUFFER_SIZE;
    if (count == 1) signal(&notempty);
    release(&mutex);
}

Object remove() {
    Object item;

    acquire(&mutex);
    while (count == 0) wait(&notempty, &mutex);
    count--; item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    if (count == BUFFER_SIZE-1) signal(&notfull);
    release(&mutex);
    return item;
}

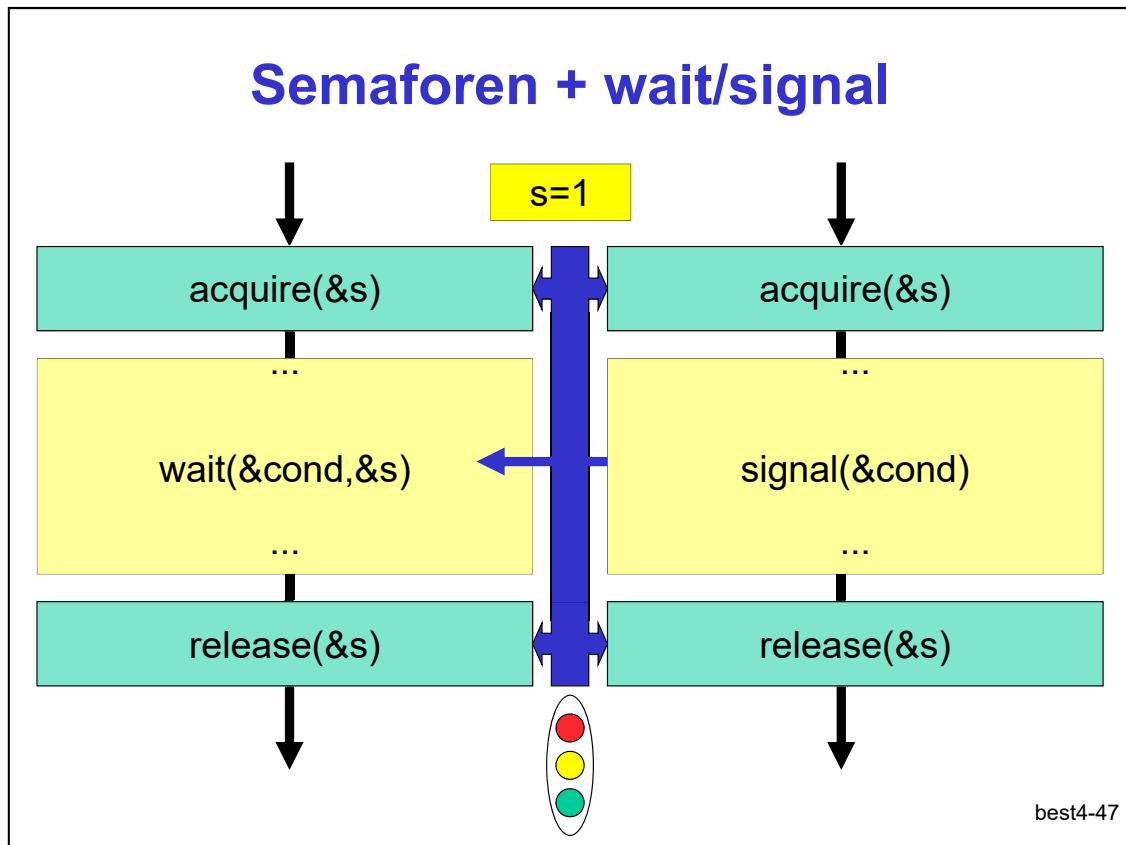
```



## Eindige buffer met semaforen en condities

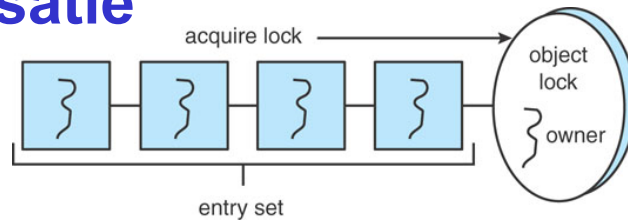
best4-46

Door gebruik te maken van wait() methodes die zelf de semafoor op atomaire manier vrijgeven, kan er een correcte synchronisatie gegarandeerd worden.



Het monitorconcept wordt aangeboden in programmeertalen, maar zelden door de kern van een besturingssysteem. In de praktijk zal men daar semaforen en mutexen tegenkomen, samen met signalen. Deze laten ook toe dat er binnen een kritieke sectie gewacht wordt.

# Eindige buffer met Java synchronisatie



```
public class BoundedBuffer implements Buffer
{
    private static final int BUFFER_SIZE = 5;
    private volatile int count, in, out;
    private volatile Object[] buffer;
    public BoundedBuffer() { // buffer is initially empty
        count = 0; in = 0; out = 0;
        buffer = new Object[BUFFER_SIZE];
    }

    public synchronized void insert(Object item) { ... }

    public synchronized Object remove() { ... }
}
```

intrinsieke lock

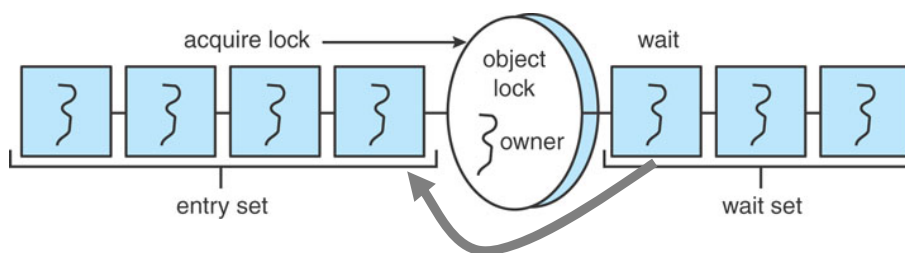
best4-48

In Java wordt er met elk object een synchronisatieobject geassocieerd (de zgn. intrinsieke lock). Dit synchronisatieobject kan gebruikt worden om de toegang tot het object in goede banen te leiden. Indien de methoden voorzien zijn van het sleutelwoord `synchronized` dient de draad die de methode wenst uit te voeren eerst het bijhorende synchronisatieobject te vergrendelen. Pas nadien kan de methode uitgevoerd worden. Bij het verlaten van de methode wordt het synchronisatieobject opnieuw vrijgegeven.

Indien een draad er niet in slaagt om het synchronisatieobject te vergrendelen, dan wordt de draad in de 'entry set' voor het object geplaatst.

# insert() met wait/notify methoden

```
public synchronized void insert(Object item) {  
    while (count == BUFFER_SIZE)  
        { try { wait(); } catch (InterruptedException e) { } }  
    ++count;  
    buffer[in] = item;  
    in = (in + 1) % BUFFER_SIZE;  
    notify();  
}
```



Als een draad `wait()` oproept, (i) geeft de draad het synchronisatieobject van het object vrij, (ii) wordt de draad in de toestand geblokkeerd gebracht, en (iii) wordt de draad in de 'wait set' van het object geplaatst. Een geblokkeerde draad moet ook kunnen onderbroken worden door middel van de methode `interrupt()`. In dat geval zal er een exceptie optreden die hier opgevangen wordt in het try-catch-blok.

Als een draad `notify()` uitvoert, dan (i) wordt er een willekeurige draad uit de 'wait set' geselecteerd, (ii) wordt de geselecteerde draad naar de 'entry set' overgebracht, en (iii) wordt de betrokken draad in de toestand 'runnable' gebracht. Vanaf dan kan de draad opnieuw de controle over het synchronisatieobject van object proberen krijgen. Naast `notify()` bestaat er ook een `notifyall()` die ervoor zorgt dat alle draden uit de wait set in één keer overgeheveld worden naar de 'entry set'.

# remove() met wait/notify methoden

```
public synchronized Object remove() {  
    Object item;  
    while (count == 0) {  
        try {  
            wait();  
        }  
        catch (InterruptedException e) { }  
    }  
    --count;  
    item = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    notify();  
    return item;  
}
```

best4-50

Remove() is volledig analoog.

Wait() en notify() zijn eigenlijk methodes die op conditieveranderlijken moeten uitgevoerd worden. Uit het feit dat er hier geen conditieveranderlijke opgegeven wordt, kan men besluiten dat er per monitor precies 1 (impliciete) conditieveranderlijke aanwezig is. Dit is een gemak, maar ook een beperking. Een draad die gedeblokkeerd wordt moet expliciet nagaan waarom hij gedeblokkeerd werd. Het gegenereerde signaal bevat immers geen dergelijke informatie.

# Bloksynchronisatie

```
Object mutexLock = new Object();  
...  
public void someMethod() {  
    nonCriticalSection();  
    synchronized(mutexLock) {  
        criticalSection();  
    }  
    nonCriticalSection();  
}
```

best4-51

Soms is een volledige methode een te groot blok om te beschermen als kritieke sectie. Java laat toe om ook kleinere blokken te beschermen, opnieuw door gebruik te maken van het synchronisatieobject dat in elk object aanwezig is. In bovenstaand voorbeeld wordt de kritieke sectie beschermd door middel van `synchronized(mutexLock)`. Alle blokken die via hetzelfde object gesynchroniseerd worden, zullen wederzijds uitgesloten uitgevoerd worden. Binnen een dergelijke kritieke sectie kunnen ook `wait()` en `notify()` gebruikt worden, maar dan wel op het synchronisatieobject (`mutexLock.wait()` en `mutexLock.notify()`).

Een monitorfunctie zou ook op deze manier kunnen gesynchroniseerd worden door als object 'this' te gebruiken.



## synchronisatieprimitieven vanaf Java 1.5

- Semaforen:
  - Semaphore sema = new Semaphore(1);
- Reentrant locks:
  - Lock Key = new ReentrantLock();
  - Key.lock()
  - Key.unlock()
- Conditieveranderlijken
  - Condition condvar = key.newCondition();
  - condvar.await()
  - condvar.signal()

best4-52

Met de komst van Java 1.5 werden er een hele resem synchronisatieprimitieven toegevoegd aan de taal. Deze zijn terug te vinden in `java.util.concurrent` en `java.util.concurrent.locks`.

Vooreerst biedt men nu de klasse `Semaphore` aan. Dit is een implementatie van een tellende semafoor.

Verder biedt men ook zgn ‘Reentrant locks’ aan, dit zijn synchronisatieobjecten waarop men `lock()` en `unlock()` kan uitvoeren. Deze locks zijn reentrant hetgeen wil zeggen dat ze meer dan eens door dezelfde draad kunnen verworven worden. De `lock()`-operatie zal dus enkel blokkeren indien de lock op dat ogenblik door een **andere** draad bezet is. Verder kan de lock er ook voor zorgen dat bij een `unlock()` de langst wachtende draad eerst zal toegelaten worden. Deze lock is met andere woorden een verfijning van de intrinsieke lock die bij objecten gebruikt worden om `synchronized` methodes te ondersteunen (ofschoon deze laatste ook al reentrant is).

Ten slotte zijn er ook conditieveranderlijken. Deze laten toe om op een meer gedetailleerde manier geblokkeerde draden te signaleren (ipv dmv het algemene `notify()` en `notifyall()`). Net zoals `notify()` en `notifyall()` gekoppeld zijn aan de intrinsieke lock, zijn de conditieveranderlijken gekoppeld aan een `ReentrantLock`. Dit maakt het overbodig om bij de `await()`-operatie de bijhorende lock als argument op te geven.

# Gebruik van semaforen en locks

```
Lock key = new ReentrantLock();
```

```
key.lock();
try {
    // kritieke sectie
}
finally {
    key.unlock();
}
```

```
Semaphore sem = new Semaphore(1);
```

```
try {
    sem.acquire();
    try {
        // kritieke sectie
    } finally {
        sem.release();
    }
}
catch (InterruptedException ie) {
    // ...
}
```

best4-53

Men moet de nodige omzichtigheid aan de dag leggen bij het gebruik van de synchronisatieprimitieven van Java. Het gevaar bestaat steeds dat door het optreden van een exceptie er van het normale controlepad van het Javaprogramma afgeweken wordt waardoor de kritieke sectie niet vrijgegeven worden.

Bij de `ReentrantLock` is het verstandig om van zodra de lock verkregen werd, de volledige kritieke sectie op te nemen in een `try`-block en de lock te ontgrendelen in het `finally`-block. Op die manier zal de lock – eenmaal vergrendeld – zeker ook ontgrendeld worden. `Lock()` genereert geen `InterruptedException`, die hoeven we dus niet op te vangen. Indien we wel op de hoogte willen gebracht worden van een onderbreking moeten we `lockInterruptibly()` gebruiken.

Bij semaforen is het iets complexer omdat `acquire()` ook excepties kan genereren. Deze moeten dan afzonderlijk opgevangen worden. Bij het optreden van `InterruptedException` zal er moeten nagegaan worden of deze effectief van `sem.acquire()` afkomstig is om de gepaste acties te kunnen ondernemen.

# Overzicht

- Wat is synchronisatie?
- Software-oplossingen
- Hardware-oplossingen
- Semafoor
- Monitor
- Transactioneel geheugen
- Boodschappen
- Impasses

best4-54

# Transactioneel geheugen

```
#define BUFFER_SIZE 5

volatile Object buffer[BUFFER_SIZE];
volatile int in, out, count;

init() {
    in = 0; out = 0; count = 0;
}

void insert(Object item) {
    atomic {
        if (count == BUFFER_SIZE) retry;
        else {
            count++; buffer[in] = item; in = (in + 1) % BUFFER_SIZE;
        }
    }
}

Object remove() {
    Object item;

    atomic {
        if (count == 0) retry;
        else {
            count--; item = buffer[out]; out = (out + 1) % BUFFER_SIZE;
        }
    }
    return item;
}
}
```

commit

retry

best4-55

Het afbakenen van een kritieke sectie met synchronisatieprimitieven zorgt ervoor dat instructies mutueel exclusief uitgevoerd worden, maar eigenlijk dienen niet de instructies maar de datastructuren beschermd te worden tegen simultaan gebruik. Transactioneel geheugen probeert precies dat te doen. Het garandeert dat een aantal geheugentoeegangen atomair gebeuren waardoor aanpassingen aan datastructuren ofwel compleet ofwel gewoon niet doorgevoerd worden. Transacties worden gedefinieerd als een atomic-blok en een transactie kan ofwel doorgaan (commit) of herstart worden (retry).

Tijdens de uitvoering wordt er gecontroleerd of andere draden dezelfde data proberen te gebruiken. Indien dat niet het geval is, dan is de transactie succesvol, indien ze wel gebruikt werden, moet één van de betrokken transacties gestopt en herbegonnen worden, net zoals bij een databanktransactie. Alle manipulaties in een transactie zijn pas definitief als de transactie afloopt. Transactioneel geheugen kan geïmplementeerd worden in hardware of in software. Als een transactie niet kan uitgevoerd worden (b.v. buffer vol of leeg), dan kan ze herstart worden – een manier van werken die eventueel kan leiden tot livelock.

# Transactioneel geheugen

```
#define BUFFER_SIZE 5

volatile Object buffer[BUFFER_SIZE];
volatile int in, out, count;

init() {
    in = 0; out = 0; count = 0;
}

void insert(Object item) {
    atomic (count < BUFFER_SIZE) {
        count++; buffer[in] = item; in = (in + 1) % BUFFER_SIZE;
    }
}

Object remove() {
    Object item;

    atomic (count > 0) {
        count--; item = buffer[out]; out = (out + 1) % BUFFER_SIZE;
    }
    return item;
}
```

best4-56

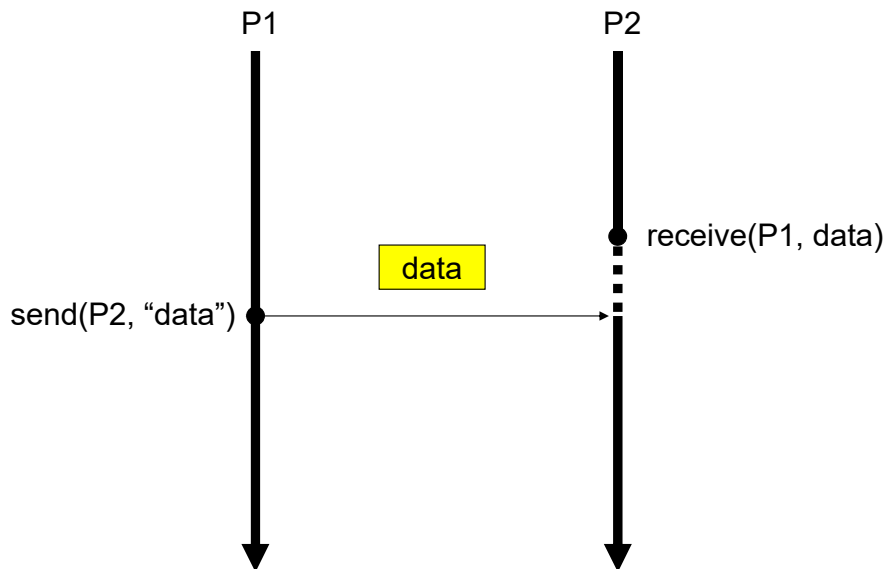
Eventueel kunnen transacties ook conditioneel uitgevoerd worden zoals in het bovenstaande voorbeeld. Dit vermijdt in dit geval het optreden van livelock.

# Overzicht

- Wat is synchronisatie?
- Software-oplossingen
- Hardware-oplossingen
- Semafoor
- Monitor
- Transactioneel geheugen
- Boodschappen
- Impasses

best4-57

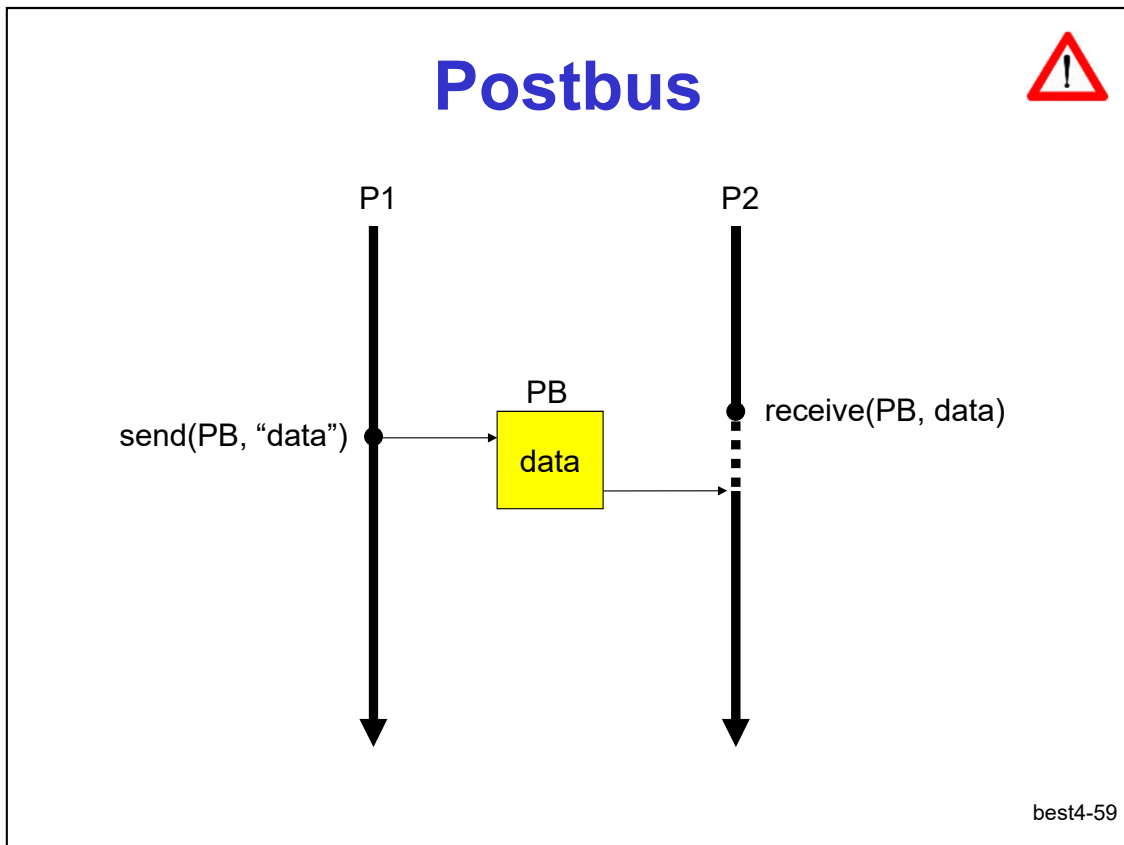
# Boodschappen



best4-58

Op gedistribueerde systemen kan men niet gemakkelijk werken met gemeenschappelijk geheugen (er bestaat wel een abstractie die men gedistribueerd gemeenschappelijk geheugen noemt (distributed shared memory of DSM) maar dit is een softwarelaag bovenop een boodschappegebaseerd systeem). Het basisprimitief op gedistribueerde systemen is het versturen en ontvangen van boodschappen. Hierbij is het belangrijk te beseffen dat een boodschap zowel een tijdstip van communicatie vastlegt, als de informatie die moet overdragen worden zelf. Bij het gebruik van gemeenschappelijk geheugen wordt de synchronisatie in hoofdzaak gebruikt om aan te geven wanneer men iets mag gaan ondernemen in het gemeenschappelijk geheugen. De boodschap ligt in dat geval opgeslagen in het geheugen, en zit niet in de synchronisatiemethode.

Boodschappen zijn niet enkel mogelijk tussen computers, maar kunnen ook tussen processen op dezelfde computer uitgewisseld worden. Er zijn slechts 2 basisprimitieven: het verzenden van een boodschap, en het ontvangen ervan. De ontvanger kan geen boodschap ontvangen nog voor ze verstuurd werd. Hij zal dus moeten wachten op het ontvangen van een boodschap.



Het versturen van een bericht naar een proces houdt belangrijke beperkingen in: men moet de identificatie van het andere proces kennen (procesidentificaties zijn doorgaan pas tijdens de uitvoering gekend), en een proces heeft maar 1 procesnummer. Alle boodschappen naar het proces zullen dus via datzelfde ene kanaal verstuurd moeten worden (ongeacht hun type). Verder is het ook niet eenvoudig om boodschappen naar een groep van processen te sturen.

Een beter alternatief kan erin bestaan om de booschap naar een gedickeerd communicatiekanaal te versturen. Een voorbeeld van een dergelijk kanaal is een postbus (mailbox). Berichten kunnen daar achtergelaten worden, en kunnen daar later afgehaald worden. Zender en ontvangen moeten het enkel maar eens zijn over de naam van de postbus. Verdere voordelen van postbuscommunicatie is dat de postbus een eigen opslagcapaciteit kan hebben, dat er meerdere zenders en ontvangers kunnen zijn en dat de zender(s) de ontvanger(s) niet moet kennen. Zij moeten enkel maar de postbus kennen.



# Boodschappencommunicatie

- Buffering
  - 0 elementen
  - Eindig
  - Oneindig
- Synchronisatie
  - Blocking send (indien buffer vol)
  - Non-blocking send (faalt indien vol)
  - Blocking receive (indien buffer leeg)
  - Non-blocking receive (faalt indien leeg)

best4-60

De boodschappenkanaal of de postbus kan een zekere buffercapaciteit bezitten. De lees- en schrijfoperaties kunnen al dan niet blokkerend zijn.

# Bounded buffer met boodschappen

```
#define BUFFER_SIZE 5
MailBox mailbox;

init() {
    initmb(&mailbox, BUFFER_SIZE);
}

insert(Object item) {
    send(&mailbox, item);
}

Object remove() {
    return receive(&mailbox);
}
```

best4-61

De implementatie van een BoundedBuffer met een boodschappengebaseerde communicatie is bijna triviaal. Na de creatie van een postbus met aangepaste capaciteit valt de semantiek van insert en remove nagenoeg volledig samen met blokkerende send en receive.

## Voorbeelden

- sockets
- MPI
- RPC
- RMI

best4-62

Belangrijke bibliotheken voor het verzenden en ontvangen van boodschappen zijn MPI (Message Passing Interface), RPC (Remote Procedure Call) en RMI (Remote Message Invocation). Sockets zijn het basismechanisme om op het niveau van het besturingssysteem communicatie tussen twee systemen tot stand te brengen.

# Sockets: server

```
import java.net.*;
import java.io.*;

public class DateServer
{
    public static void main(String[] args) {
        try {
            ServerSocket sock = new ServerSocket(6013);
            while (true) {
                Socket client = sock.accept();
                PrintWriter pout = new PrintWriter(client.getOutputStream(), true);
                pout.println(new java.util.Date().toString());
                client.close();
            }
        } catch (IOException ioe) { System.err.println(ioe);}
    }
}
```

best4-63

Dit is een voorbeeld van een datumserver die geïmplementeerd werd met sockets. De server luistert in dit geval naar poort 6013. De `sock.accept()` wacht op een inkomende verbinding (eigenlijk is dit het equivalent van een receive-operatie). Het poortnummer functioneert als identificatie van een postbus zodat de identificatie van het communicatiekanaal losgekoppeld wordt van de procesidentificatie. Van zodra er een communicatiekanaal tot stand is gekomen stuurt de server een boodschap naar de client en sluit de verbinding af.

# Sockets: client

```
import java.net.*;
import java.io.*;

public class DateClient
{
    public static void main(String[] args) throws IOException {
        Socket sock = null;
        try {
            sock = new Socket("127.0.0.1",6013);
            InputStream in = sock.getInputStream();
            BufferedReader bin = new BufferedReader(new InputStreamReader(in));

            String line;
            while( (line = bin.readLine()) != null)
                System.out.println(line);
        } catch (IOException ioe) { System.err.println(ioe);
        } finally { sock.close(); }
    }
}
```

best4-64

Aan de clientzijde wordt er contact opgenomen met de server (hier via het loopback adres 127.0.0.1 dat naar dezelfde machine wijst, maar in de praktijk kan dit gelijk welke machine zijn). Van zodra die verbinding tot stand gekomen is, wordt de binnenkomende informatie gelezen en uitgeprint totdat er geen informatie meer ontvangen wordt.

## RMI: server

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Date;

public class RemoteDateImpl extends UnicastRemoteObject
    implements RemoteDate
{
    public RemoteDateImpl() throws RemoteException { }

    public Date getDate() throws RemoteException {
        return new Date();
    }

    public static void main(String[] args) {
        try {
            RemoteDate dateServer = new RemoteDateImpl();
            Naming.rebind("RMIDateObject", dateServer);
            System.out.println("RMIDateObject bound in registry");
        } catch (Exception e) { System.err.println(e);
        }
    }
}
```

best4-65

Hetzelfde gedrag kan ook met RMI geïmplementeerd worden. Het serverobject biedt de methode `getDate()` aan. Het remote object moet geregistreerd worden via de methode `rebind()`. Hier wordt het remote object geboden aan een publieke naam "RMIDateObject". Via deze naam zal het kunnen geadresseerd worden.

# RMI: client

```
import java.rmi.*;

public class RMIClient
{
    public static final String server = "127.0.0.1";

    public static void main(String args[]) {
        try {
            String host = "rmi://" + server + "/RMIDateObject";
            RemoteDate dateServer = (RemoteDate)Naming.lookup(host);
            System.out.println(dateServer.getDate());
        } catch (Exception e) { System.err.println(e); }
    }
}
```

best4-66

De RMI client zoekt de RMI server op via de URL `rmi://127.0.0.1/RMIDateObject` (opnieuw loopbackadres). Eenmaal een referentie naar `dateServer` verworven werd, kunnen de remote methodes gebruikt worden als waren het methoden van een lokaal object. Dit is uiteraard een zeer transparante manier van werken. Dit mag echter niet de indruk wekken dat de prestatie ook vergelijkbaar is met een lokale oproep. Deze oproepen gaan wel degelijk naar een andere JVM (hier op dezelfde machine, maar dat hoeft niet), de argumenten en het resultaat voor de methodeoproep moeten geserialiseerd en doorgestuurd worden. Er komt dus heel wat overhead bij te pas.

# Overzicht

- Wat is synchronisatie?
- Software-oplossingen
- Hardware-oplossingen
- Semafoor
- Monitor
- Transactioneel geheugen
- Boodschappen
- **Impasses**

best4-67



# Allocatie systeemmiddelen

## Draad 1

```
r1.acquire(); // A1
r2.acquire(); // A2

// ...

r2.release(); // R2
r1.release(); // R1
```

## Draad 2

```
r2.acquire(); // A2
r1.acquire(); // A1

// ...

r1.release(); // R1
r2.release(); // R2
```

r1, r2 systeemmiddelen

best4-68

Allocatie van fysieke (randapparaten) en logische systeemmiddelen (PCB's, FCB, semaforen...) gebeuren steeds op dezelfde manier: aanvraag – gebruik – vrijgave.

Processen die systeemmiddelen aanvragen zullen blokkeren indien deze niet beschikbaar zijn.

## Impasse (deadlock)

Volgorde	resultaat
A1 A2 R1 R2 A2 A1 R1 R2	OK
A1 A2 a2 a1	Impasse
A1 A2 a1 a2	Impasse
A2 A1 a2 a1	Impasse
A2 A1 a1 a2	Impasse
A2 A1 R1 A1 R2 A2 R1 R2	OK
A2 A1 R1 R2 A1 A2 R1 R2	OK

70 verwevingen van de twee draden  
Leidend tot 7 unieke uitvoeringen.

- 3 correct
- 4 impasses

best4-69

Net zoals bij het inleidende voorbeeld kunnen de synchronisatie-instructies van beide draden ook verweven worden. Dit leidt in dit geval tot 70 verwevingen, waarvan de uitvoering leidt tot een 7-tal gevallen. Van die 7 gevallen zijn er maar drie die zullen termineren, de andere vier leiden tot een impasse. Hieruit leren we dat impasses niet noodzakelijk optreden, maar dat ze in de ene uitvoering wel, en in de andere uitvoering niet kunnen optreden, afhankelijk van de timing van de verschillende draden.

## Impasse: vier nodige voorwaarden van (Coffman, 1971)



1. Wederzijdse uitsluiting
2. Vasthouden en wachten
3. Geen preëemptie
4. Circulair wachten

best4-70

Om een echte impasse te hebben zijn er vier nodige voorwaarden:

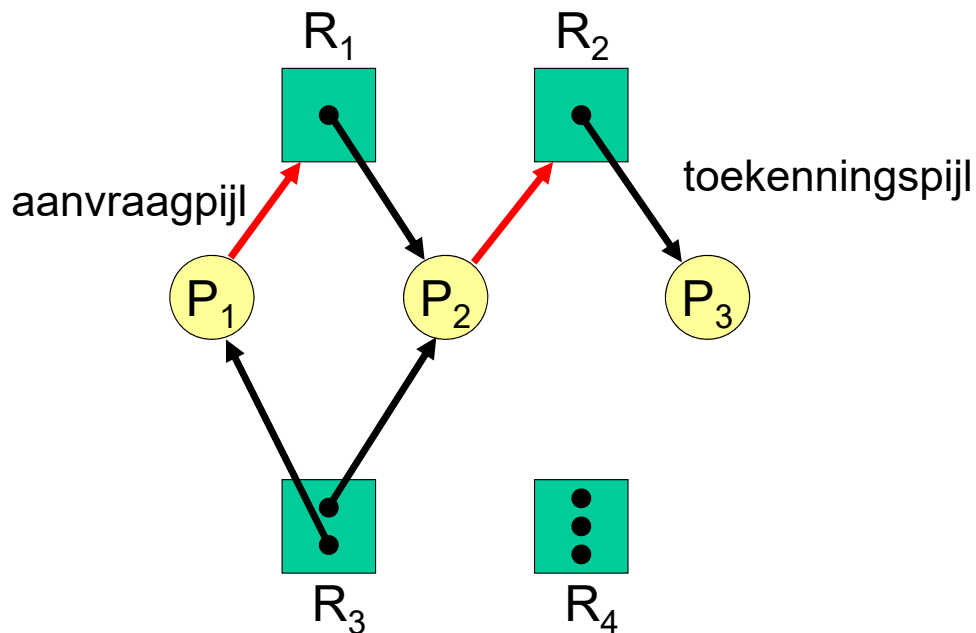
**Wederzijdse uitsluiting:** er moet minstens 1 systeemmiddel zijn dat niet gedeeld kan worden door de processen. Het zal de allocatie van dit systeemmiddel zijn dat de processen zal doen blokkeren.

**Vasthouden en wachten (hold and wait):** een proces mag de reeds gealloceerde systeemmiddelen niet vrijgeven terwijl het staat te wachten op een bijkomend systeemmiddel. Alles wat verworven is, blijft verworven.

**Geen preëemptie:** systeemmiddelen kunnen enkel vrijwillig afgestaan worden, nadat het systeemmiddel niet meer nodig is. Systeemmiddelen kunnen niet afgenomen worden.

**Circulair wachten:** de verschillende processen die betrokken zijn bij een impasse, moeten circulair op elkaar staan wachten.

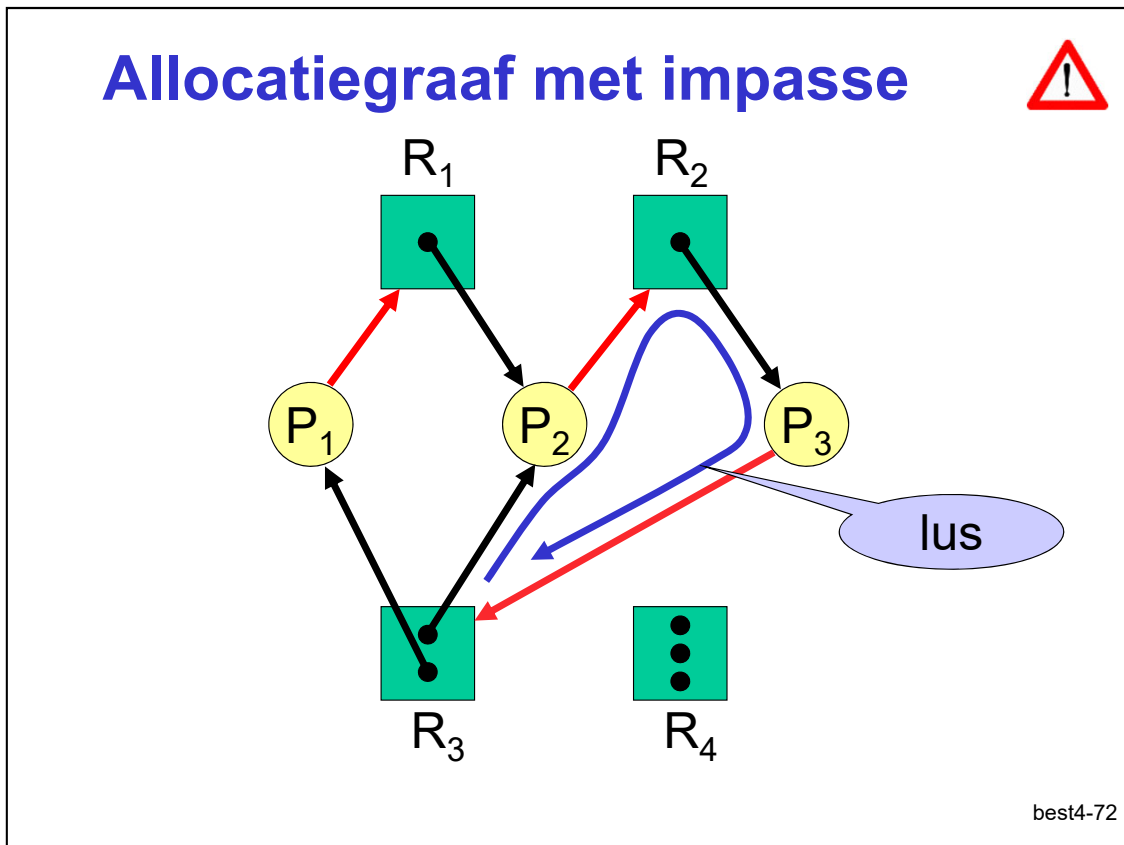
## Allocatiegraaf zonder impasse



best4-71

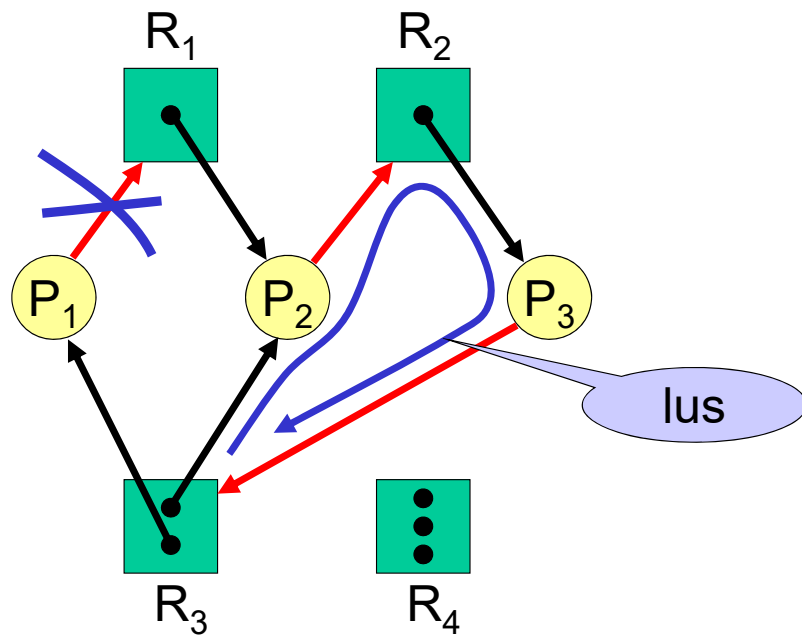
Systeemmiddelen die gealloceerd worden door processen kunnen voorgesteld worden in een allocatiegraaf. De ronde knopen stellen de processen voor, de vierkante knopen stellen de systeemmiddelen voor. Het aantal bolletjes in een vierkante knoop bepaalt hoeveel exemplaren er van een bepaald systeemmiddel voorhanden zijn. Een pijl van een systeemmiddel naar een procesknoop geeft aan dat er een dergelijk systeemmiddel gealloceerd werd door de procesknoop (dit is een zgn. toekenningspijl). Een pijl van de procesknoop naar het systeemmiddel is een aanvraagpijl en geeft aan dat het proces geblokkeerd is tot het vrijkomen van een systeemmiddel.

In bovenstaand voorbeeld zijn er twee aanvraagpijlen. Enkel proces  $P_3$  heeft geen aanvraagpijl en kan beginnen uitvoeren en finaal termineren. Bij de terminatie van  $P_3$  komt  $R_2$  vrij.  $R_2$  kan dan toegewezen worden aan  $P_2$  die daardoor ook kan gaan rekenen. Op die manier komt o.a.  $R_1$  vrij die dan kan gealloceerd worden aan  $P_1$ .  $P_1$  kan dan rekenen, waarna alle processen getermineerd zijn. Er is in dit geval dus geen impasse opgetreden.



Indien er een lus optreedt in de allocatiegraaf bestaat de kans dat er een impasse optreedt. Dit is een nodige voorwaarde, doch geen voldoende voorwaarde (zie verder). In dit voorbeeld zijn alle processen geblokkeerd, en is er dus geen voortgang meer mogelijk.

## Allocatiegraaf met impasse



best4-73

Door een aanvraagpijl te verbreken, krijgen we een andere situatie.  $P_1$  kan nu termineren, en  $R_3$  vrijgeven. In de praktijk zal men proces  $P_1$  extern termineren waardoor zowel de toekenningspijl als de aanvraagpijl verdwijnt. Merk op dat er in dit voorbeeld nog steeds een lus is, maar toch geen impasse meer.  $P_3$  krijgt nu een instantie van  $R_3$  en kan rekenen, waardoor de lus zal verdwijnen bij het termineren van  $P_3$ .

# Afhandeling van een impasse

1. preventie
2. vermijding
3. detectie
4. ontkenning

best4-74

# Preventie



Ervoor zorgen dat aan één van de vier Coffmanvoorwaarden niet voldaan is

- **Wederzijdse uitsluiting**: spooling, read-write synchronisatie.
- **Vasthouden en wachten**: alle systeemmiddelen ineens alloceren of heralloceren.
- **geen preëemptie**: systeemmiddelen ontnemen of terugnemen (vooral voor CVE en geheugen).
- **circulair wachten**: in volgorde alloceren

best4-75



## Vermijding

Het besturingssysteem moet ervoor zorgen niet in een onveilige toestand te belanden. Onveilige toestand = toestand van waaruit een impasse mogelijk is.

De maximale hoeveelheid systeemmiddelen moet bij het begin meegedeeld worden

Gekende wiskundige oplossing:  
bankiersalgoritme

best4-76

## Detectie

Geen preventie of vermijding, maar een oplossing bij het optreden ervan. Impasse detectie geeft ons de lijst van processen die betrokken zijn bij een impasse.

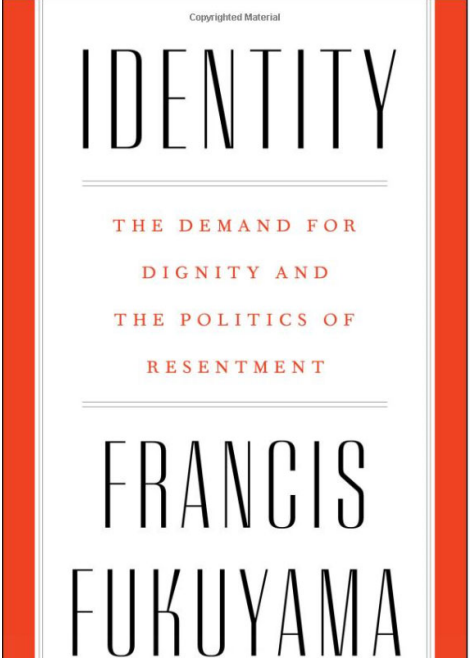
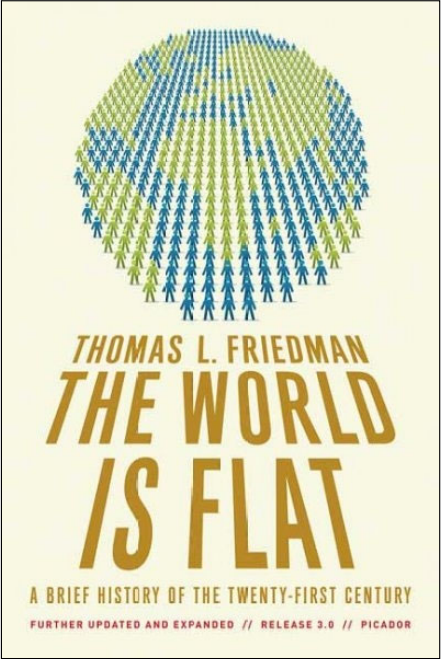
Men kan 1 of meerdere processen uit deze lus weghalen om het systeem uit de impasse te helpen.

best4-77

# Ontkenning

Manuele oplossing van de impasse  
= situatie in de meeste commerciële OS'en

best4-78



best4-79