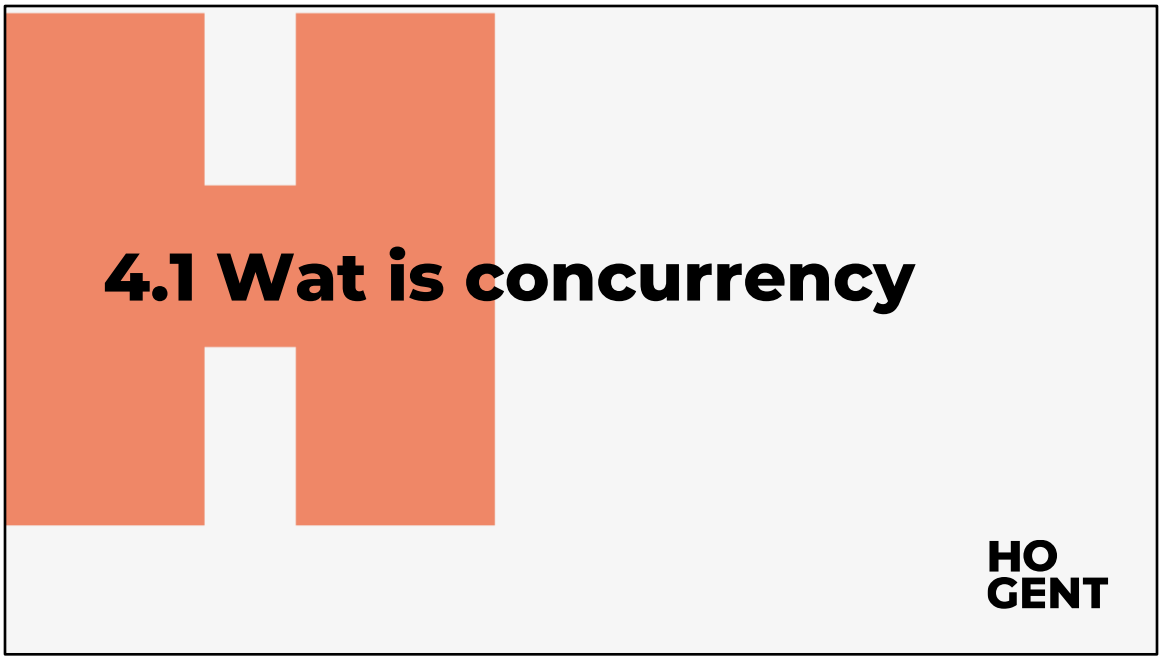
H4 - Concurrency

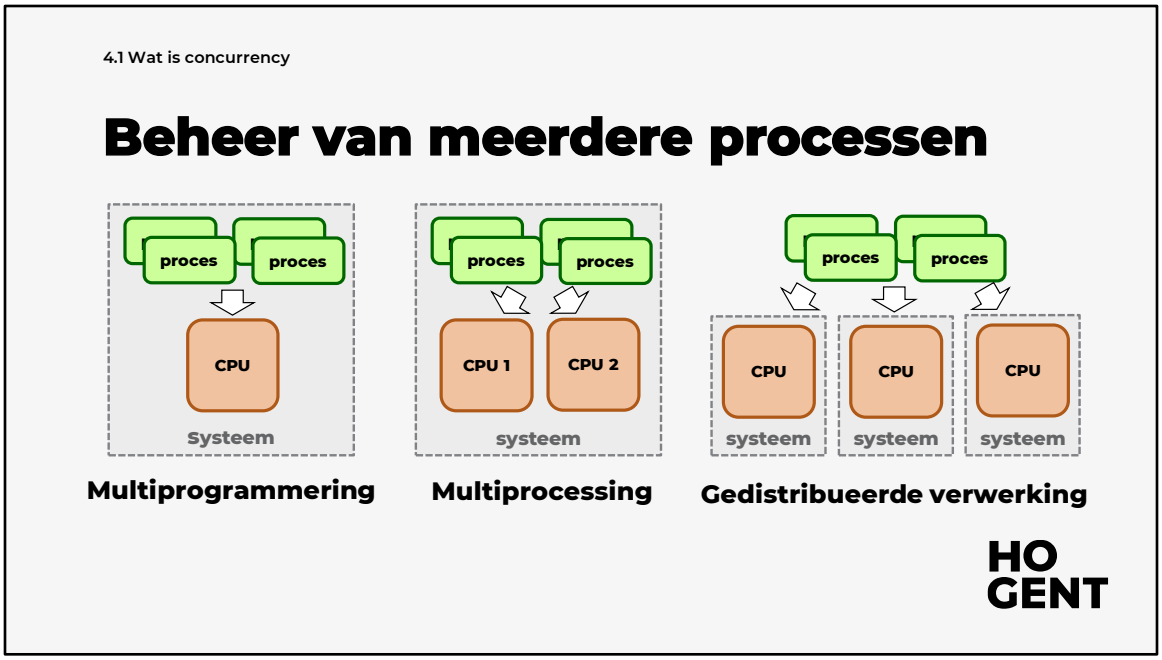
HO GENT

4. Concurrency

4.1 Wat is concurrency
4.2 Wederzijdse uitsluiting (mutual exclusion)
4.3 Synchronisatie
4.4 Deadlocks

HO GENT

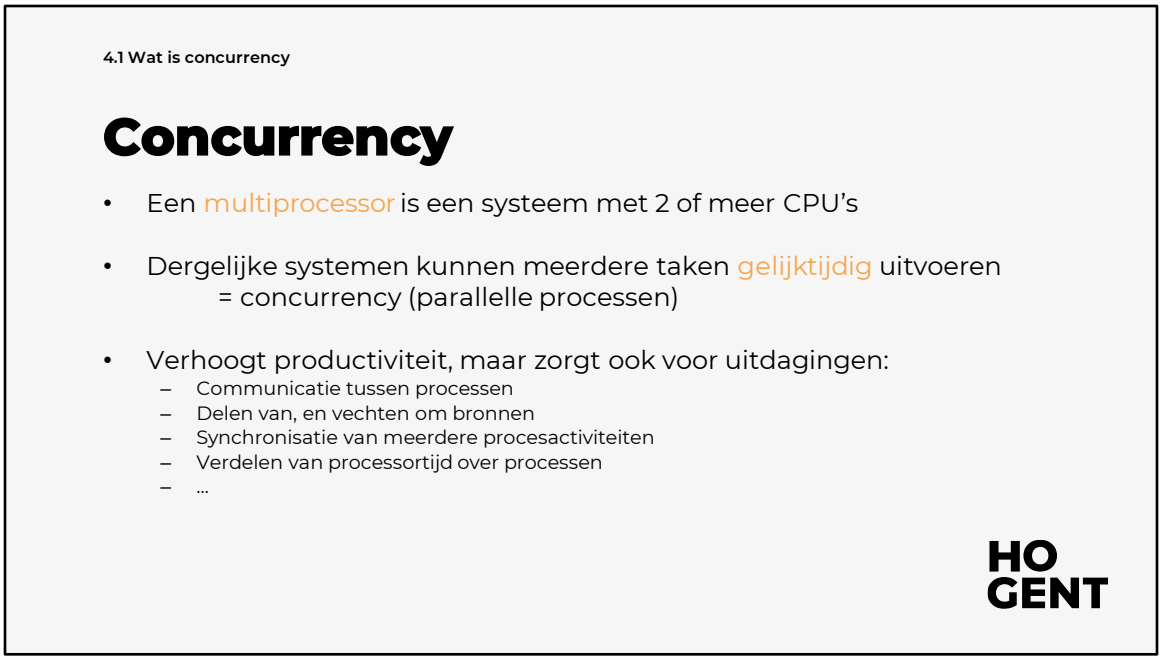

4.1 Wat is concurrency
HO
GENT



Beheer van meerdere processen

In het vorig hoofdstuk hebben we gezien hoe op de meeste moderne computersystemen meerdere processen gelijktijdig actief kunnen zijn. Deze processen kunnen al dan niet gelijktijdig uitgevoerd worden. Concreet kunnen we een onderscheid maken tussen:

- Multiprogrammering: het beheer van meerdere processen in een systeem met 1 processor
- Multiprocessing: het beheer van meerdere processen in een systeem met meerdere processoren
- Gedistribueerde verwerking: het beheer van meerdere processen die worden uitgevoerd op een aantal verspreide (= gedistribueerde) computersystemen



Concurrency

Concurrency (ofwel parallelle processen) is bij computerprocessen een belangrijke plaats gaan innemen. Doordat het mogelijk werd enorme rekencapaciteit in een kleine chip te stoppen, zijn multiprocessors gemeengoed geworden. Een multiprocessor is een computersysteem met twee of meer processoren. Dergelijke systemen kunnen meerdere taken gelijktijdig uitvoeren. Dit verhoogt de productiviteit, maar zorgt ook voor enkele uitdagingen:

- Hoe kunnen verschillende processen met elkaar communiceren?
- Wat moet het OS doen wanneer processen computerbronnen delen, of vechten om bepaalde bronnen?
- Hoe kunnen activiteiten binnen verschillende processen gesynchroniseerd worden?
- Hoe moet de beschikbare processortijd verdeeld worden over de verschillende processen?

Algemeen kunnen we stellen dat concurrency verwijst naar processen of activiteiten die gelijktijdig uitgevoerd worden. Wanneer deze moeten samenwerken, bijvoorbeeld wanneer ze informatie moeten uitwisselen of bronnen delen, is het een uitdaging om

dit probleemloos te laten verlopen. Het tegengestelde van concurrency zijn sequentiele processen, waarbij alle stappen strikt na elkaar worden uitgevoerd, en er dus van gelijktijdigheid geen sprake is.

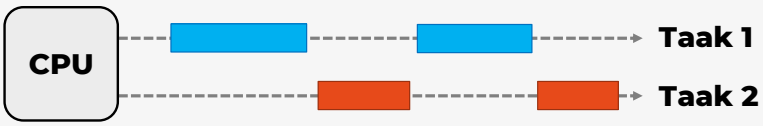
Verschillende situaties

Concurrency beperkt zich niet tot computersystemen met meerdere processoren. Ook bij systemen met maar 1 CPU kan concurrency optreden. Concurrency treedt op in verschillende situaties:

- Meerdere toepassingen: bij multiprogrammering kunnen er verschillende processen gelijktijdig actief zijn, en multiprogrammering werd uitgevonden om de verwerkingstijd (processortijd) dynamisch te kunnen verdelen over een aantal actieve toepassingen
- Gestructureerde toepassingen: als uitbreiding op de beginselen van modulair ontwerpen en gestructureerd programmeren kunnen sommige toepassingen effectief worden geprogrammeerd als een verzameling van gelijktijdige processen
- Structuur van besturingssysteem: dezelfde voordelen van het structureren gelden ook voor de systeemprogrammeur, en besturingssystemen zelf worden vaak geïmplementeerd als een verzameling processen

4.1 Wat is concurrency

Concurrency - 1 CPU



Gelijktijdige uitvoering

- Toepassing boekt vooruitgang op meer dan één taak: (schijnbaar) gelijktijdig
- Bij systemen met één CPU: schakelen tussen verschillende taken tijdens uitvoering

HO GENT

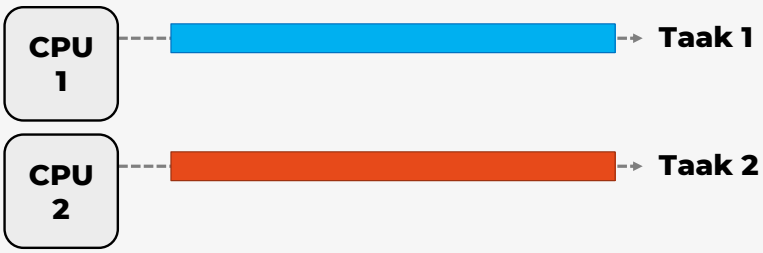
Concurrency bij systemen met 1 processor

Concurrency betekent dat een toepassing vooruitgang boekt op meer dan één taak - tegelijkertijd of op zijn minst schijnbaar tegelijkertijd (gelijktijdig).

Als de computer slechts één CPU heeft, kan de toepassing op exact hetzelfde moment geen vooruitgang boeken, maar is er meer dan één taak aan de gang op een moment binnen de toepassing. Om tegelijkertijd vooruitgang te boeken met meer dan één taak schakelt de CPU tussen de verschillende taken tijdens de uitvoering.

4.1 Wat is concurrency

Parallel Execution



Parallelle uitvoering
- Systemen met meer dan één CPU of CPU-kern
- Meer dan één taak van een toepassing worden gelijktijdig (parallel) uitgevoerd
- ≠ parallelisme (zie verder)

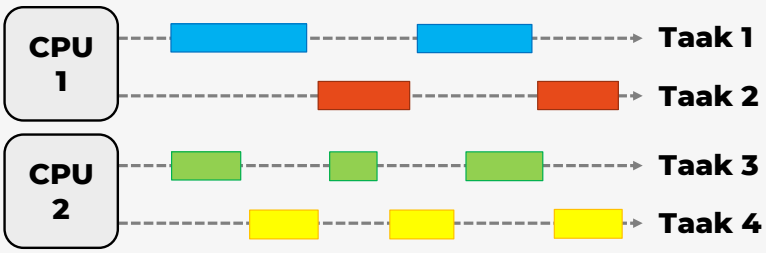
HO GENT

Parallelle uitvoering

Parallelle uitvoering *(EN: Parallel Execution)* treedt op wanneer een computersysteem meer dan één CPU of CPU-kern heeft en tegelijkertijd vooruitgang boekt op meer dan één taak. Parallelle uitvoering verwijst echter niet naar hetzelfde fenomeen als parallellisme. Parallelle uitvoering wordt geïllustreerd in het diagram in deze slide.

4.1 Wat is concurrency

Parallel Concurrent Execution



Parallelle gelijktijdige uitvoering

- Taken worden verdeeld over meerdere CPU's
- Binnen één CPU: schakelen tussen verschillende taken – (schijnbaar) gelijktijdig
- Taken op verschillende CPU's worden parallel uitgevoerd

HO GENT

Parallelle gelijktijdige uitvoering

We kunnen ook de vorige 2 slides combineren, wat het principe is van parallelle gelijktijdige uitvoering *(EN: parallel concurrent execution)*. Hierbij worden de verschillende taken verdeeld over meerdere CPU's. Taken die op dezelfde CPU worden uitgevoerd, worden gelijktijdig uitgevoerd, terwijl taken die op verschillende CPU's worden uitgevoerd parallel worden uitgevoerd.

Parallelism

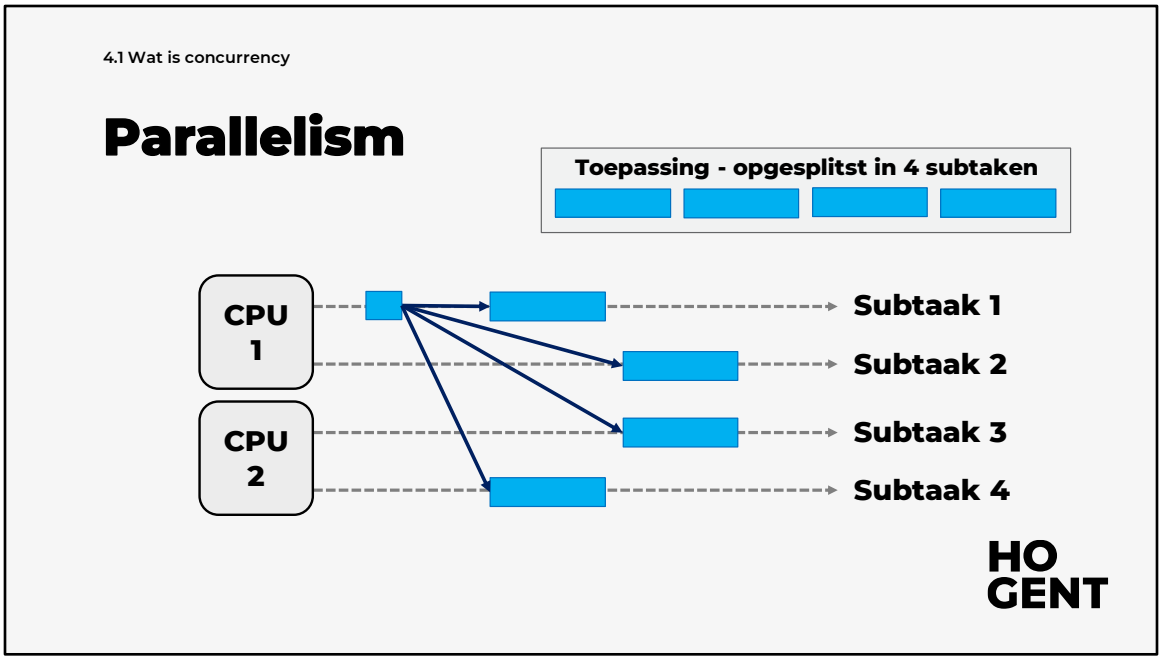
- Toepassing splitst zijn werk op in subtaken die parallel kunnen worden verwerkt
- Verwijst niet naar hetzelfde uitvoeringsmodel als parallelle gelijktijdige uitvoering
- Hoe parallellisme bereiken:
  - Meer dan 1 subtaak
  - Elke subtaak draait parallel op afzonderlijke CPU's of CPU-cores

HO GENT

Parallellisme

De term parallellisme *(EN: parallelism)* betekent dat een toepassing zijn werk opsplitst in kleinere subtaken die parallel kunnen worden verwerkt, bijvoorbeeld op meerdere CPU's op exact hetzelfde moment. Parallellisme verwijst dus niet naar hetzelfde uitvoeringsmodel als parallelle gelijktijdige uitvoering - zelfs als ze er op het oppervlak vergelijkbaar uitzien.

Om echte parallellisme te bereiken moet een toepassing meer dan subtaak hebben die wordt uitgevoerd - en elke subtaak moet op afzonderlijke CPU's / CPU-cores / GPU-cores draaien.

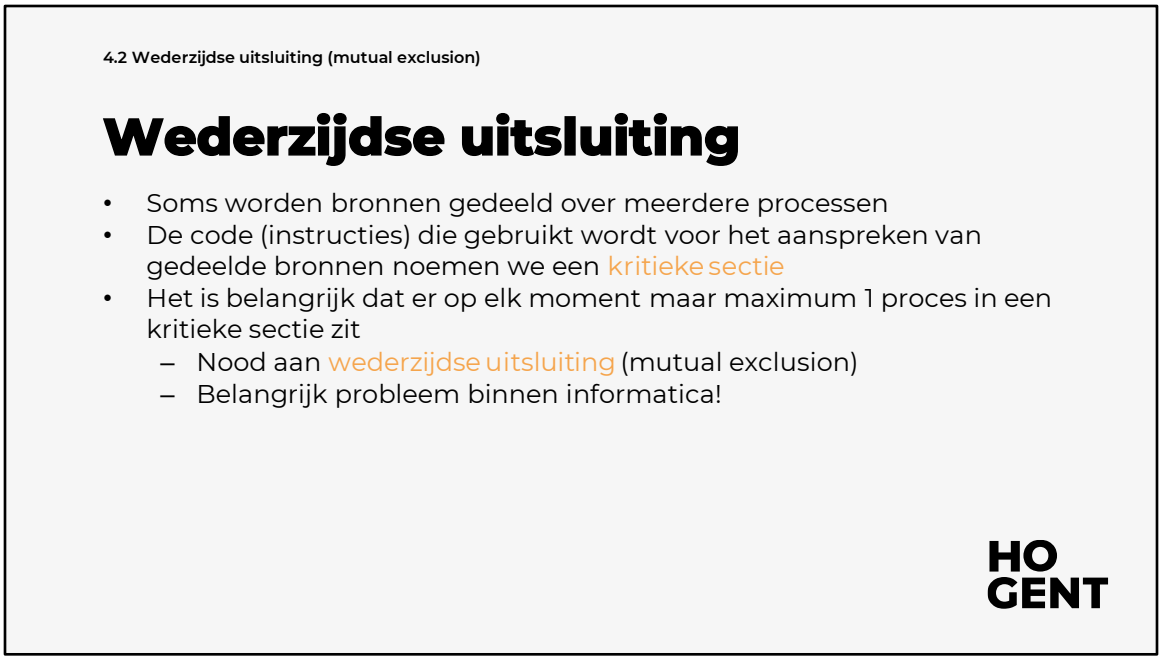


Parallellisme

Het diagram in deze slide illustreert een grotere taak die wordt opgesplitst in 4 subtaken. Deze 4 subtaken worden uitgevoerd op 2 verschillende CPU's. Dit betekent dat delen van deze toepassing gelijktijdig worden uitgevoerd (op dezelfde CPU) en delen parallel worden uitgevoerd (op verschillende CPU's).

4.2 Wederzijdse uitsluiting (mutual exclusion)

HO GENT



Wederzijdse uitsluiting

Soms willen meerdere taken of processen gelijktijdig gebruikmaken van dezelfde gedeelde computerbronnen, bijvoorbeeld wanneer ze hetzelfde deel van het gemeenschappelijk RAM geheugen willen aanspreken. Dit kan echter voor conflicten en inconsistenties zorgen. Zoals we reeds gezien hebben in het vorig hoofdstuk, bestaat een proces uit meerdere instructies die uitgevoerd worden op een processor. De instructies (code) voor het aanspreken van die gedeelde bronnen (bijvoorbeeld lezen of schrijven naar gemeenschappelijke data) noemen we een kritieke sectie. Het is belangrijk dat er op elk moment maar 1 proces in een kritieke sectie zit.

We willen processen gelijktijdig laten uitvoeren, en tegelijkertijd toch voorkomen dat bepaalde delen van die processen, de kritieke secties, parallel worden verwerkt. Wanneer parallelle processen zich toegang verschaffen tot het gemeenschappelijke geheugen, bevatten hun kritieke secties de opdrachten die deze resources aanspreken. De kritieke sectie van een proces is hier dus de code die naar gemeenschappelijke data verwijst. Als de uitvoering van een proces in de kritieke sectie is aangeland, moeten wij er voor zorgen dat elk ander proces zijn eigen kritieke sectie niet betreedt. Omgekeerd moeten wij ook opletten dat een proces zijn kritieke

sectie niet binnenkomt op het moment dat een ander proces in zijn kritieke sectie zit.

Wederzijdse uitsluiting is een term uit de informatica waarmee de eis bedoeld wordt dat wanneer een proces zich in een kritieke sectie bevindt en er gebruikgemaakt wordt van gedeelde bronnen, er geen andere processen zijn die zich ook in een kritieke sectie bevinden waarbij dezelfde gedeelde bronnen worden gebruikt. Het regelen van de toegang tot gedeelde bronnen is een belangrijk probleem in de computerwetenschappen. Dit komt doordat taken op elk willekeurig moment kunnen starten of stoppen.

4.2 Wederzijdse uitsluiting (mutual exclusion)

Wederzijdse uitsluiting: voorbeeld

- Stel: je hebt een globale variabele getal (geheel getal)
- 2 processen willen deze variabele aanpassen, via volgende instructies:
  1. Lees de huidige waarde van de variabele getal vanuit het geheugen
  2. Verhoog deze waarde met 1
  3. Schrijf de nieuwe waarde van getal weg naar het geheugen
- Als beide processen om beurt de instructies uitvoeren is er geen probleem, maar wat als het eerste proces onderbroken wordt na uitvoeren van de eerste instructie?
  - → De variabele getal zal, afhankelijk van de volgorde van uitvoering, verhoogd zijn met 1 of 2
  - → Een oplossing is om de 3 instructies te groeperen als kritieke sectie, en hiervoor wederzijdse uitsluiting af te dwingen

HO GENT

Wederzijdse uitsluiting: voorbeeld

We kunnen het belang van wederzijdse uitsluiting illustreren aan de hand van een eenvoudig voorbeeld. Stel dat je in het geheugen een globale variabele hebt, **getal**, die een geheel getal voorstelt. Een proces kan deze waarde aanpassen via volgende 3 instructies:

1. Lees de huidige waarde van de variabele **getal** in vanuit het geheugen
2. Verhoog deze waarde met 1
3. Schrijf de nieuwe waarde voor **getal** weg naar het geheugen

Stel nu dat er 2 processen zijn, proces A en proces B, die parallel uitgevoerd worden, en beide processen willen bovenstaande instructies uitvoeren. We kunnen dit voorstellen als A1, A2, A3 (instructies 1, 2 en 3 voor proces A) en B1, B2, B3 (voor proces B).

Als beide processen na elkaar de instructies uitvoeren is er geen probleem, en zal de waarde van **getal** met 2 verhoogd zijn. De volgorde van uitvoering is dan bijvoorbeeld: A1 – A2 – A3 – B1 – B2 – B3

In het vorig hoofdstuk hebben we echter gezien dat een proces onderbroken kan worden tijdens de uitvoering. Stel bijvoorbeeld dat proces A onderbroken wordt na uitvoeren van de eerste instructie, en proces B van aan bod komt. De volgorde van uitvoeren is dan bijvoorbeeld: A1 – B1 – B2 – B3 – A2 – A3

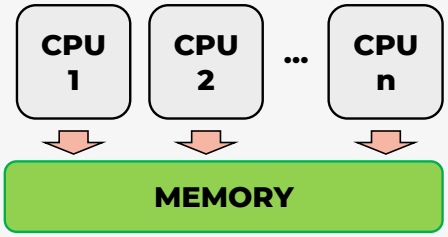
Merk op dat de eerste instructie hier niet opnieuw uitgevoerd wordt, waardoor proces A nog de 'oude' waarde kent van de variabele **getal**. Na uitvoeren van alle stappen zal **getal** hier dus niet verhoogd zijn met 2, maar met 1.

Om dit te voorkomen, zullen we de 3 instructies dus moeten groeperen in een kritieke sectie, en afdwingen dat proces B deze instructies niet mag uitvoeren wanneer proces A hier reeds mee begonnen is.

4.2 Wederzijdse uitsluiting (mutual exclusion)

Wederzijdse uitsluiting bij multiprocessing

- Niet alleen processen, maar ook activiteiten binnen één proces kunnen parallel worden uitgevoerd
- We zullen processen bespreken, maar de principes gelden eveneens voor activiteiten binnen één proces
- De moeilijkheden ontstaan wanneer de processen het gemeenschappelijke geheugen aanspreken



HO GENT

Wederzijds uitsluiting bij multiprocessing

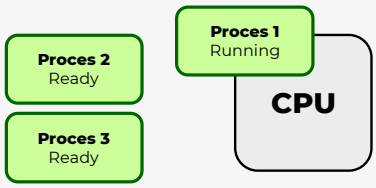
Zoals we reeds gezien hebben bestaan er verschillende niveaus van concurrency. Niet alleen processen, maar ook activiteiten binnen één proces kunnen parallel worden uitgevoerd. In dit hoofdstuk zullen we hoofdzakelijk processen bespreken, maar de principes gelden eveneens voor activiteiten binnen één proces.

Als parallelle processen niets gemeenschappelijk gebruiken, is er geen probleem. De moeilijkheden ontstaan wanneer de processen het gemeenschappelijke geheugen aanspreken.

4.2 Wederzijdse uitsluiting (mutual exclusion)

Wederzijdse uitsluiting bij multiprogrammering

- Ook in een systeem met maar 1 CPU zijn gelijklopende processen mogelijk
- Processen kunnen niet tegelijkertijd worden uitgevoerd, maar ze kunnen wel op hetzelfde moment proberen de besturing van de CPU te krijgen
- Wanneer twee van zulke processen het gemeenschappelijk geheugen aanspreken, kunnen er nog steeds problemen ontstaan



HO GENT

Wederzijdse uitsluiting bij multiprogrammering

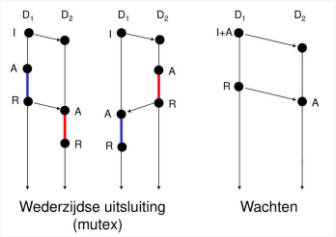
Ook wanneer een computersysteem maar één processor heeft, zijn gelijklopende processen mogelijk. De processen kunnen in dergelijk geval uiteraard niet parallel (gelijktijdig) uitgevoerd worden, maar ze kunnen wel op hetzelfde moment proberen de besturing van de CPU te krijgen.

Wanneer twee van zulke processen het gemeenschappelijk geheugen willen aanspreken, kunnen er nog steeds problemen ontstaan, en kan wederzijds uitsluiting nodig zijn.

4.2 Wederzijdse uitsluiting (mutual exclusion)

Wederzijdse uitsluiting afdwingen

- Niet evident om wederzijdse uitsluiting af te dwingen!
- Mogelijke oplossingen: algoritme van Dekker (2 processen) en wederzijds uitsluitingsalgoritme van Peterson (2 of meer processen)
- Alternatieven: wederzijds uitsluiting afdwingen via semaforen (Dijkstra) of monitoren



HO GENT

Wederzijdse uitsluiting afdwingen

Het is in de praktijk is het niet zo eenvoudig om wederzijdse uitsluiting af te dwingen. Een eenvoudige oplossing zou kunnen zijn om met 1 globale variabele (bijvoorbeeld een boolean) bij te houden of er een proces in een kritieke sectie zit, maar dan is het probleem verschoven naar de toegang tot deze globale variabele.

Er bestaan algoritmes die een oplossing bieden voor het probleem van wederzijdse uitsluiting. Het algoritme van Dekker bijvoorbeeld is de eerste bekende juiste oplossing voor dit probleem, maar is beperkt tot wederzijdse uitsluiting voor 2 parallelle processen. In 1981 werd er een oplossing geformuleerd, Peterson's algoritme, die ook bruikbaar is voor wederzijdse uitsluiting bij meer dan 2 processen. De details van deze algoritmes vallen echter buiten de scope van deze cursus.

Daarnaast bestaat er ook een alternatieve oplossing, die gebruik maakt van semaforen. Een semafoor is een soort van integer variabele, bedacht door Dijkstra, die slecht door enkele primitieve operaties kan gewijzigd worden. Een primitieve operatie is een operatie die niet onderbroken kan worden: ofwel wordt de operatie volledig uitgevoerd, ofwel wordt deze volledig ongedaan gemaakt. Een ander

alternatief is om monitoren te gebruiken. Een monitor is een constructie in een programmeertaal die een functionaliteit biedt die vergelijkbaar is met die van semaforen, maar gemakkelijker te besturen is.

Extra informatie (ter info):

- https://en.wikipedia.org/wiki/Dekker%27s_algorithm
- https://nl.wikipedia.org/wiki/Wederzijds_uitsluitingsalgoritme_van_Peterson
- https://nl.wikipedia.org/wiki/Semafoor_(computer)
- https://nl.wikipedia.org/wiki/Monitor_(gedistribueerd_programmeren)

Meer dan toegang tot gedeeld geheugen

- In vorige slides: wederzijdse uitsluiting om toegang te regelen van CPU naar gemeenschappelijk geheugen
- Andere vormen van wederzijdse uitsluiting:
  - Toegang tot bestanden en records
    (bv: meerdere processen willen gelijktijdig schrijven naar zelfde bestand)
  - Toegang tot hardware bronnen
    (bv: 2 processen willen gelijktijdig iets sturen naar printer)
  - ...
- Het is de taak van het besturingssysteem om in deze situaties wederzijdse uitsluiting te garanderen

HO GENT

Meer dan toegang tot gedeeld geheugen

In de vorige slides hebben we wederzijdse uitsluiting besproken in de context van het regelen van de toegang van verschillende processen die uitgevoerd worden op de CPU naar gemeenschappelijk geheugen. Er bestaan echter ook andere vormen van wederzijdse uitsluiting:

- Wederzijdse uitsluiting kan ook gebruikt worden om de toegang naar bestanden te regelen, en bijvoorbeeld te voorkomen dat 2 processen gelijktijdig naar hetzelfde bestand willen schrijven (en zo elkaars wijzigingen ondermijnen).
- Daarnaast kan wederzijdse uitsluiting ook nuttig zijn om de toegang tot bepaalde hardware bronnen te regelen. Stel bijvoorbeeld dat 2 processen op exact hetzelfde moment een taak willen versturen naar een printer... Gelukkig gaat het besturingssysteem dit niet toelaten, en de printertaken bijhouden in een wachtrij zodat de documenten één voor één (sequentieel) afgeprint worden.

Algemeen kunnen we stellen dat het de taak is van het besturingssysteem om (waar nodig) wederzijdse uitsluiting te garanderen tot gedeelde computerbronnen.

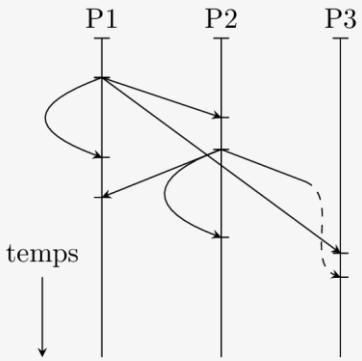
4.3 Synchronisatie

HO
GENT

4.3 Synchronisatie

Wat is synchronisatie?

- Synchronisatie is het proces, of het resultaat van iets gelijktijdig maken
  - Ontstaan: 19e eeuw, treinen werden zo snel dat een verschil in lokale tijd opviel
  - Synchronisatie van klokken ook nodig om botsingen te voorkomen op enkelspoor
- Hier: het opleggen van een dwingende volgorde aan events die door concurrente, asynchrone processen worden uitgevoerd.
- Wij moeten garanderen dat processen in een bepaalde volgorde verlopen



HO GENT

Wat is synchronisatie?

Synchronisatie is het proces of het resultaat van iets gelijktijdig maken. Het is afgeleid van het Griekse συν (sýn) 'samen' en χρόνος (chrónos) 'tijd'. Pas in de negentiende eeuw is synchronisatie in de geschiedenis van de mens een rol van betekenis gaan spelen. De treinen gingen zo snel rijden dat een verschil in lokale tijd op ging vallen. Het gelijk zetten van de klokken langs de spoorlijn werd noodzakelijk. Synchronisatie van de klokken was ook een vereiste voor het veilig gebruik van enkelspoor. De treinen konden zo volgens het spoorboekje blijven rijden, zodat vermeden werd dat twee treinen tegelijkertijd op hetzelfde spoorvak op elkaar aanstormden. Sindsdien is het belang van synchronisatie alleen maar groter geworden (bron: https://nl.wikipedia.org/wiki/Synchronisatie).

Binnen de context van concurrency kunnen we synchronisatie ook definiëren als het opleggen van een dwingende volgorde aan events die door concurrente, asynchrone processen worden uitgevoerd. Bij concurrente processen weten we niet welk proces wanneer aan bod komt, maar toch willen we garanderen dat de uitvoering van (delen van) bepaalde processen in een bepaalde volgorde verlopen.

4.3 Synchronisatie

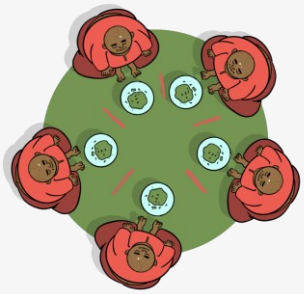
Het filosofen probleem

- 5 filosofen aan ronde tafel
- 5 vorken op tafel, tussen elke filosoof

- Filosoof kan denken of eten (niet tegelijkertijd)
- Om te eten heeft een filosoof 2 vorken nodig (L + R)

- Filosoof kan vork oppakken als die op tafel ligt
- Filosoof moet de vorken één voor één oppakken

- In welke volgorde moeten filosofen de vorken oppakken en hoe lang mag een filosoof eten, zodat geen enkele filosoof zal verhongeren?



HO GENT

Het filosofen probleem

We kunnen het probleem van synchronisatie illustreren aan de hand van het filosofen probleem.

Stel: er zitten 5 filosofen aan een ronde tafel. Op die tafel liggen 5 vorken, er ligt één vork tussen elke twee filosofen dus elke filosoof heeft een vork aan zijn linker- en rechterkant. Een filosoof kan twee dingen doen: eten of denken (maar niet tegelijkertijd). Als een filosoof denkt kan hij dus niet eten en als een filosoof eet kan hij niet denken.

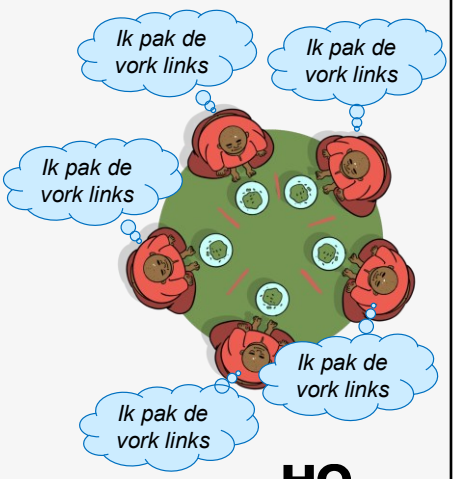
Om te eten heeft elke filosoof twee vorken nodig. Er zijn echter slechts vijf vorken. Zo heeft elke filosoof één vork aan zijn linker en één aan zijn rechterhand; de filosoof kan die oppakken als die op tafel ligt, maar moet de vorken één voor één oppakken. Het probleem is nu om de filosofen zodanige instructies te geven dat ze niet zullen verhongeren. In welke volgorde moeten de filosofen de vorken oppakken? En hoe lang mag een filosoof eten, voor hij de vork terug op tafel legt?

We moeten dus met andere woorden een planning vinden voor het oppakken en neerleggen van de vorken. Als we dit bovendien eerlijk willen doen, moeten we er in de planning voor zorgen dat elke filosoof even veel tijd krijgt om te eten, met andere woorden: elke filosoof zou na verloop van tijd even lang 2 vorken vastgehad moeten hebben. Dit soort problemen zijn in het algemeen niet zo eenvoudig op te lossen, maar illustreren mooi het probleem van synchronisatie.

4.3 Synchronisatie

Een eerste poging

- Stel: elke filosoof wil meteen eten, pakt eerst vork aan linkerkant op van tafel
- Elke filosoof heeft 1 vork vast, andere vork is reeds gepakt door filosoof rechts
- Geen enkele filosoof wil vork neerleggen, maar blijft wachten tot andere vork vrij is
- Filosofen blijven eeuwig op elkaar wachten: er is een deadlock



HO GENT

Een eerste poging

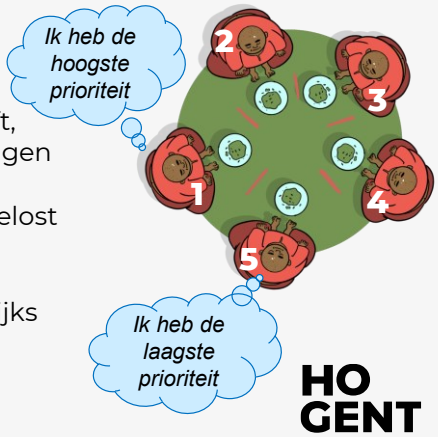
Stel bijvoorbeeld dat elke filosoof meteen wil eten, en hiervoor eerst de vork aan de linkerkant oppakt van de tafel. Elke filosoof heeft nu één vork vast, maar de andere vork is reeds bezet door de filosoof aan de rechterkant. Bovendien wil geen enkele filosoof de vork terug neerleggen tot ze iets gegeten hebben.

De filosofen blijven nu dus eeuwig op elkaar wachten, en elke filosoof zal uiteindelijk verhongeren. Een dergelijke situatie noemen we een **deadlock**: er is geen voortgang meer mogelijk. Zoals we verderop in dit hoofdstuk zullen zien, kunnen processen binnen een besturingssysteem ook in een deadlock zitten, wanneer elk proces wacht op een ander proces dat ook aan het wachten is op een ander proces in een circulaire structuur (zoals de ronde tafel).

4.3 Synchronisatie

Een tweede poging

- Stel: elke filosoof heeft nummer, een lager nummer heeft voorrang
- Als buur met lager nummer een vork heeft, moet filosoof wachten en vork(en) neerleggen
- Het probleem van een deadlock is nu opgelost
- Het systeem is echter niet eerlijk
- Filosoof met hoogste nummer kan mogelijks verhongeren: starvation



HO GENT

Een tweede poging

Om toch vooruitgang te boeken, zullen filosofen dus de vork moeten neerleggen voor ze iets gegeten hebben. We kunnen hiervoor bijvoorbeeld elke filosoof een nummer geven, en een filosoof met een lager nummer krijgt voorrang bij het eten. Als een filosoof merkt dat één van zijn buren een vork vastheeft en een lager nummer heeft, dan moet hij zijn vork terug op tafel leggen en mag hij nog niet eten tot de buur klaar is met eten.

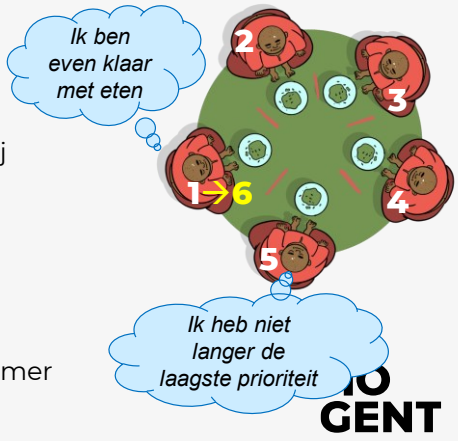
Het probleem van een deadlock is nu opgelost: filosofen zullen nooit oneindig op elkaar blijven wachten, want door invoeren van de nummers hebben we een volgorde opgelegd. Er is echter een ander probleem: het systeem is nu niet **eerlijk**. De filosoof met het laagste nummer krijg steeds voorrang, en als die eeuwig zou blijven eten kunnen zijn buren nooit eten. Bovendien zal de filosoof met het hoogste nummer het langst moeten wachten voor hij kan eten, want hij moet wachten tot alle andere filosofen klaar zijn met eten. Indien de andere filosofen elk om beurt zouden blijven eten, kan de filosoof met het hoogste nummer uiteindelijk verhongeren, dit noemen we **starvation**. Ook dit is een fenomeen dat zich kan voordoen bij processen,

bijvoorbeeld wanneer een proces eeuwig moet wachten omdat er telkens een ander proces voorrang krijgt (zie ook vorig hoofdstuk).

4.3 Synchronisatie

Een eerlijke poging

- Filosoof mag niet oneindig blijven eten, moet na maximumtijd even stoppen om te filosoferen
- Wanneer filosoof stopt met eten krijgt hij hoogste nummer + 1
- Elke filosoof zal uiteindelijk kunnen eten: geen kans op deadlock of starvation
- Het systeem is eerlijk: elke filosoof heeft om beurt laagste nummer



GENT

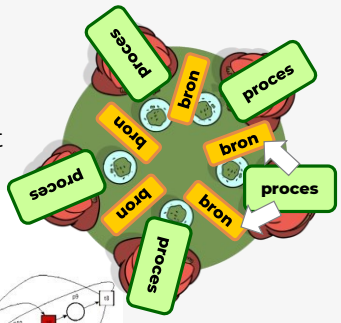
Een eerlijke poging

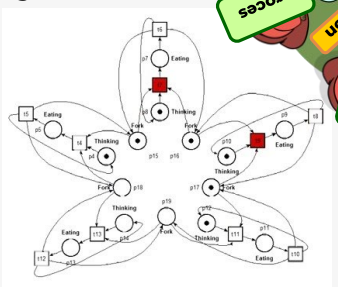
Om er voor te zorgen dat de oplossing eerlijk is, zullen we 2 extra regels moeten toevoegen aan onze poging. Om te beginnen kunnen we eisen dat een filosoof niet oneindig mag blijven eten, maar na een bepaalde maximumtijd verplicht is om eten te stoppen met eten om te filosoferen. Hierbij moet de filosoof beide vorken op tafel neerleggen. Bovendien krijgt een filosoof die even stopt met eten een nieuwe nummer, namelijk het hoogste nummer aan tafel verhoogd met 1. Dit om te voorkomen dat hij meteen nadat hij gestopt is, opnieuw zou kunnen beginnen eten.

Het probleem van een deadlock of starvation is nu opgelost: elke filosoof zal uiteindelijk iets kunnen eten, want elke filosoof zal ooit het laagste nummer hebben (als alle andere filosofen iets gegeten hebben). Het systeem is nu dus ook eerlijk, want elke filosoof heeft dus vroeger of later even de hoogste prioriteit

Link naar processen

- Filosofen = taken of processen
  Vorken = gedeelde bronnen (resources)
- Bron kan niet door 2 processen gelijktijdig gebruikt worden, wederzijdse uitsluiting nodig





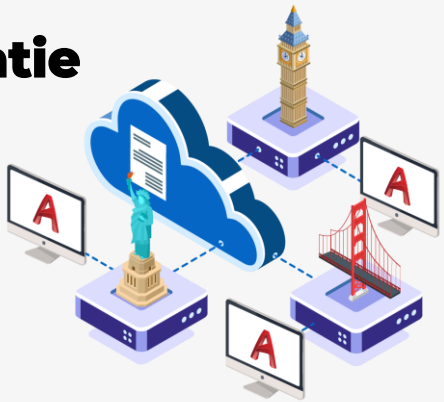
HO GENT

https://www.youtube.com/watch?v=Ar6ezhwVNdg

4.3 Synchronisatie

Bestandssynchronisatie

- 2 of meer identieke kopieën van bestand
  - Wijzigingen in één kopie zichtbaar in andere
  - Synchronisatie aan hand van vaste regels
- Vroeger: vaak manueel (bv. USB-stick)
  Nu: automatisch over netwerk / internet
  - Veel programma's beschikbaar,
    bv. DropBox, OneDrive, Google Drive, ...
  - Vaak met centrale kopie in de cloud
  - Ook gebruikt voor maken van backups



HO GENT

Bestandssynchronisatie

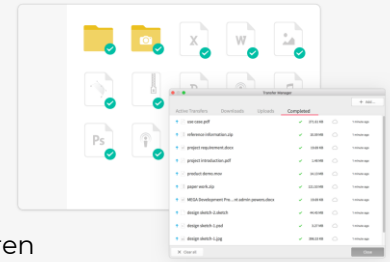
Bestandssynchronisatie is een voorbeeld van synchronisatie waarbij twee of meer bestanden aan elkaar gelijkgesteld worden. Bij bestandssynchronisatie worden, aan de hand van bepaalde regels, bestanden automatisch gesynchroniseerd zodat de inhoud van deze bestanden op twee of meerdere plaatsen hetzelfde zijn.

Voor de opkomst van netwerken en het internet gebeurde bestandssynchronisatie vaak manueel, bijvoorbeeld door een bestand te kopiëren naar een externe gegevensdrager zoals een floppydisk of flashdrive, en zo op een ander toestel te kopiëren. Deze manier van werken was echter vrij omslachtig. Sinds de opkomst van netwerken en het internet is het echter veel eenvoudiger om bestandssynchronisatie automatisch te laten verlopen via het netwerk. Vandaag de dag bestaan er veel programma's die hiervoor gebruikt kunnen worden.

4.3 Synchronisatie

Bestandssynchronisatie: werking

- Selectie van mappen/bestanden
- Vaak: bron ➔ doel
  - 1 bron (source)
  - 1 of meerdere doelen (lokaal of remote)
- Mogelijkheid om regels toe te voegen:
  - Bepaalde mappen uitsluiten?
  - Filter op type(s) bestanden?
  - Verborgen bestanden?
- Synchronisatie kan meteen (continu) gebeuren of periodiek (op vastgelegde tijdstippen)
  - Online samenwerken ⇔ back-up maken



HO GENT

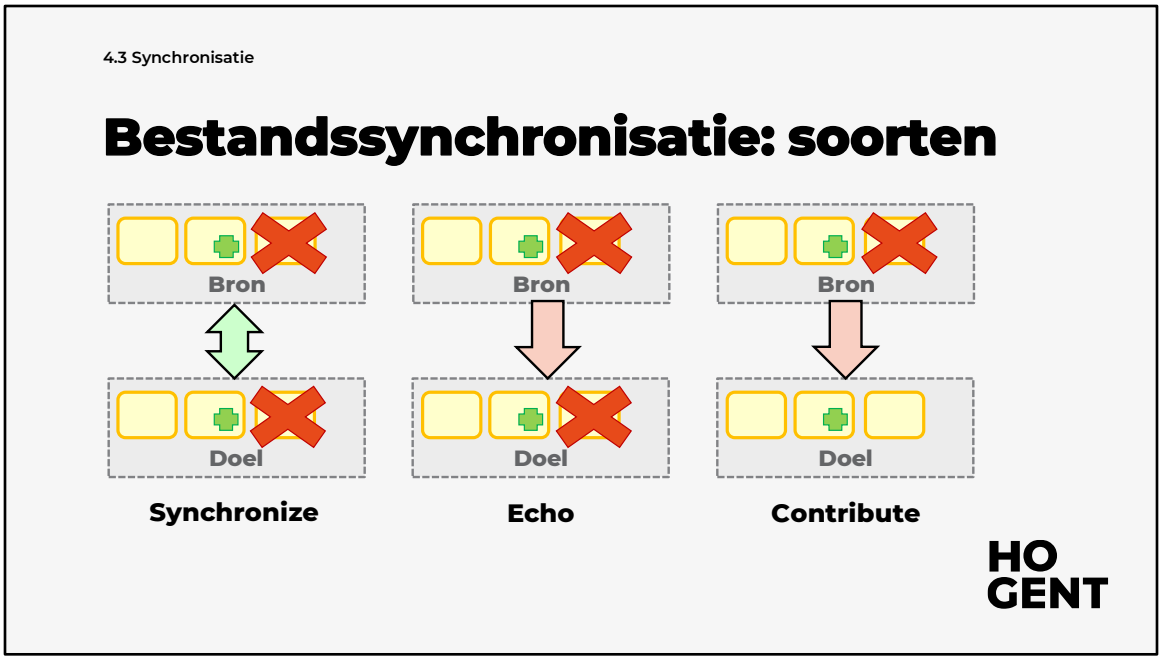
Bestandssynchronisatie: werking

Een eerste stap bij bestandssynchronisatie is het kiezen van de mappen en/of bestanden die gesynchroniseerd moeten worden. Vaak wordt er gewerkt met één bronmap (die de originele bestanden bevat) en één of meerdere doelmappen (waar je een kopie wil hebben van de bronbestanden). De doelmappen kunnen lokaal (op hetzelfde toestel) of remote (ander toestel op netwerk/internet) zijn.

Naast de selectie van de bronbestanden en –mappen is het ook vaak mogelijk om extra regels toe te voegen, bijvoorbeeld om te filteren welke bestanden wel/niet gesynchroniseerd moeten worden. Zo is het bijvoorbeeld mogelijk om bepaalde sub mappen of bestanden uit te sluiten, om te filteren op basis van bestandstype, of om aan te geven of verborgen bestanden wel of niet gesynchroniseerd moeten worden.

Tenslotte kan er voor gekozen worden om de synchronisatie meteen te doen, en de bestanden constant (continu) gesynchroniseerd te houden, of de synchronisatie periodiek uit te voeren op vastgelegde tijdstippen (bijvoorbeeld elke dag om middernacht). Dit laatste is vooral nuttig voor het maken van back-ups, terwijl de eerste optie vooral bruikbaar is wanneer je met meerdere personen wil

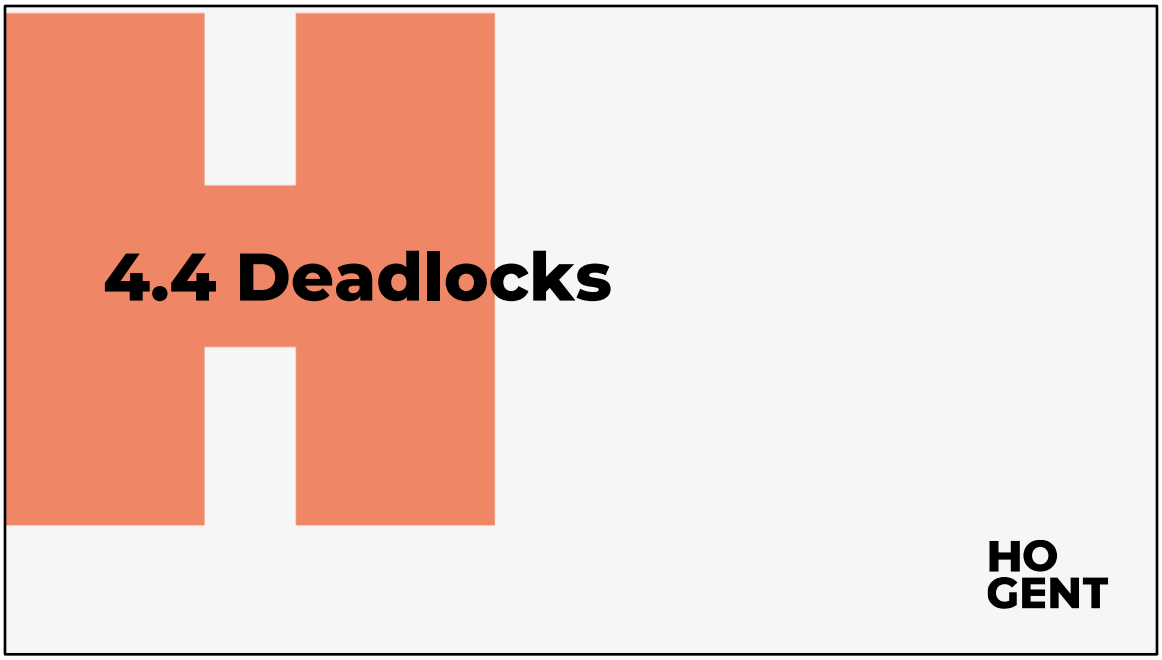
samenwerken.



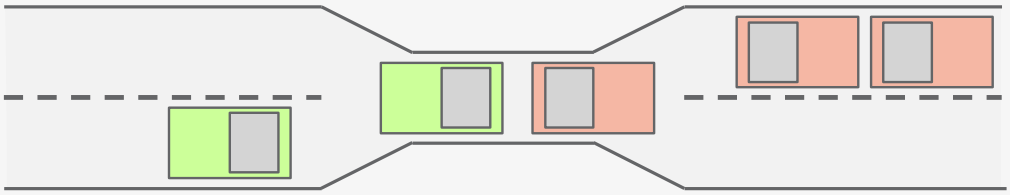
Bestandssynchronisatie: soorten

We kunnen een onderscheid maken tussen verschillende soorten van bestandssynchronisatie:

- Bij **synchronize** wordt de inhoud van de bron- en doelmap gelijk gehouden. Elke wijziging in een kopie zal dus altijd zichtbaar zijn in alle andere kopieën
- Bij **echo** worden nieuwe en gewijzigde bestanden gekopieerd van de bronmap naar de doelmap. Hernoemde en verwijderde bestanden in de bronmap worden in doelmap ook hernoemd of verwijderd. De synchronisatie gebeurt echter maar in één richting.
- De werking van **contribute** is gelijkaardig aan deze van **echo**, maar er worden geen bestanden verwijderd uit de doelmap als dit in de bronmap wel gebeurd is.


4.4 Deadlocks
HO
GENT

Wat is een deadlock?



HO GENT

Wat is een deadlock?

In de vorige slides hebben we reeds gezien dat we bij concurrency moeten opletten om niet in een deadlock te komen. Bij het filosofen probleem bijvoorbeeld zou het kunnen zijn dat elke filosoof één vork vastheeft, en die niet wil neerleggen tot hij iets gegeten heeft. Als elke filosoof hetzelfde denkt, zal geen enkele filosoof kunnen eten en zal elke filosoof uiteindelijk verhongeren. Er is geen voortgang meer mogelijk, en algemeen kunnen we zeggen dat het systeem in een deadlock zit.

Deadlocks bestaan er in vele maten en vormen. De figuur in deze slide toont bijvoorbeeld een eenvoudige situatie uit het verkeer waarbij een deadlock kan optreden. Een stuk weg bestaat uit 2 rijstroken, maar onderweg is er een versmalling, waardoor auto's in beide richtingen even over één enkele rijstrook moeten. Voor de eenvoud gaan we er hier van uit dat auto's enkel vooruit kunnen rijden. De auto's in het groen rijden van links naar rechts, de auto's in het rood van rechts naar links.

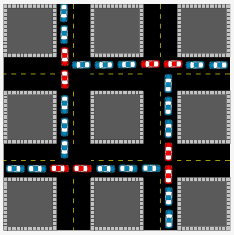
Indien er veel verkeer is kan dit voor lastige situaties zorgen. Indien de auto's elkaar mooi afwisselen, en er dus telkens één rode en nadien één groene auto door de wegversmalling rijdt is er geen probleem, maar als een rode en groene auto

gelijktijdig over de wegversmalling willen rijden zullen ze elkaar blokkeren waardoor geen enkele auto nog verder kan (auto's kunnen immers enkel vooruit rijden). Mochten auto's achteruit kunnen rijden, zouden we dit probleem kunnen oplossen, tenminste als er geen andere wagens staan te wachten. Als er echter achter de rode én groene auto veel auto's staan, kan het nog gebeuren dat de auto's niet meer achteruit kunnen rijden en kan het lastig zijn om uit deze situatie te geraken.

De bovenstaande situatie illustreert dus ook het probleem van de deadlock. Algemeen kunnen we zeggen dat een deadlock (ook impasse genaamd) een situatie is waarbij een bepaalde actie is vastgelopen op wederzijdse uitsluiting. In het voorbeeld van deze slide is het stuk wegversmalling het deel waar we wederzijdse uitsluiting voor nodig hebben, en dus willen afdwingen dat er op elk moment maar één wagen gebruik maakt van deze "kritieke sectie".

Deadlocks in de echte wereld: gridlock





HO
GENT

Deadlocks in de echte wereld: gridlock

Deadlocks komen trouwens niet enkel voor binnen de computerwereld, maar ook binnen de echte wereld, bijvoorbeeld in het verkeer. Een mooi voorbeeld hiervan is de gridlock (in het Nederlands ook verkeersinfarct genoemd). Een gridlock is een situatie waarbij het verkeer zodanig gehinderd is door uitzonderlijke omstandigheden (file, weersomstandigheden, ...) dat voertuigen nog nauwelijks vooruit kunnen komen, meestal veroorzaakt doordat weggebruikers elkaar indirect blokkeren (kop van file sluit aan bij de staart).

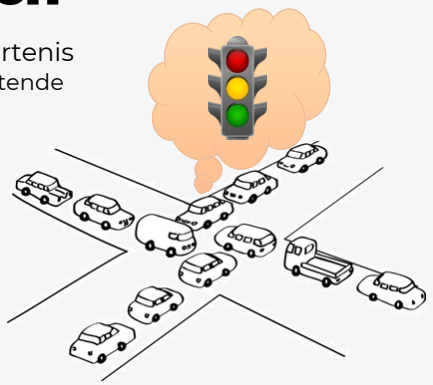
De afbeeldingen op deze slide illustreren enkele mogelijke gridlocks.

In het vervolg van dit hoofdstuk zullen we uiteraard op deadlocks binnen computersystemen, maar je kan de analogie trekken met het verkeer: elk voertuig stelt een proces voor, en een rijvak is een computerbron die het proces nodig heeft.

4.4 Deadlocks

Deadlocks bij processen

- Twee of meer processen wachten op gebeurtenis
  - Gebeurtenis kan enkel door één van de wachtende processen worden veroorzaakt
- Hoe deadlock behandelen?
  - Gebruik protocol om deadlock te voorkomen
    OF
  - Laat deadlocks toe, maar detecteer deadlock-situatie en herstel deze



HO GENT

Deadlocks bij processen

Binnen de informatica treedt een deadlock (of impassetoestand) dus op wanneer 2 of meer processen voor onbepaalde tijd wachten op een gebeurtenis die enkel door 1 van de wachtende processen kan worden veroorzaakt. Zoals we gezien hebben bij het filosofen probleem kan het bijvoorbeeld zijn dat 2 processen 2 verschillende bronnen nodig hebben om verder te kunnen. Als elk van beide processen reeds één bron toegekend heeft, en weigert om deze bron vrij te geven, dan zullen de processen dus eeuwig op elkaar blijven wachten.

Het is de taak van een besturingssysteem om deadlocks te voorkomen, of wanneer deze toch zouden voorkomen, deze op te sporen en op te lossen. Algemeen zijn er dus 2 mogelijkheden om met deadlocks om te gaan:

- Gebruik één of ander protocol om een deadlock te voorkomen, zodat het systeem nooit in een deadlock kan zitten, of..
- Laat toe dat het systeem in deadlock-situatie kan komen, maar zorg dat het besturingssysteem een deadlock-situatie kan detecteren en los deze situatie dan op.

4.4 Deadlocks

Deadlocks behandelen

- Deadlock voorkomen
  - OS beperkt gebruik van gemeenschappelijke bronnen
  - Doel: deadlock onmogelijk maken
- Deadlock vermijden
  - Alle aanvragen voor bronnen in detail onderzoeken
  - Kans op deadlock verminderen
- Deadlock signaleren
  - Hoe kan het OS weten dat er een deadlock is?
  - Processen zitten in wacht-toestand, is wachten permanent?
- Deadlock herstellen
  - Hoe kan OS een deadlock-situatie oplossen?

HO GENT

Deadlocks behandelen

Het OS kan er dus voor kiezen om deadlocks te voorkomen (of strenger: vermijden), of deadlocks toe te laten maar deze op te sporen en dan te behandelen.

- Om **deadlocks te voorkomen** kan het besturingssysteem het gemeenschappelijk gebruik van bronnen beperken met als doel om een deadlock-situatie zo goed als onmogelijk te maken.
- Om **deadlocks te vermijden** kan het besturingssysteem alle aanvragen voor resources heel nauwkeurig onderzoeken. Ziet het besturingssysteem dat de toewijzing van een resource het risico van deadlock met zich meebrengt, dan weigert het de gevraagde toegang en vermijdt het zo het probleem.
- Als er een deadlock optreedt, moet het besturingssysteem dit kunnen **signaleren**. Het besturingssysteem ziet elk proces in een wachttoestand. Hoe kan het besturingssysteem erachter komen dat dit wachten permanent is?
- Als een deadlock optreedt, moet het systeem dit tenslotte ook kunnen **herstellen**. Wat moet er gebeuren nadat het besturingssysteem een deadlock ontdekt? De processen moeten daar toch een keer uit bevrijd worden. Het besturingssysteem moet dit probleem oplossen.

Voorwaarden deadlock

Een deadlock kan enkel ontstaan indien er tegelijkertijd aan de volgende 4 voorwaarden is voldaan:

- **Wederzijdse uitsluiting (Mutual Exclusion):** Gemeenschappelijk gebruik van bronnen moet onder wederzijdse uitsluiting plaatsvinden, d.w.z. als een proces toegang heeft tot een bron *(resource),* mag geen enkel ander proces deze bron benaderen tot de bron is vrijgegeven.
- **Bezet houden en wachten (Hold and Wait):** Een proces kan meerdere bronnen aanvragen zonder de eerder toegewezen bronnen vrij te geven. Er moet dus een proces bestaan dat ten minste 1 bron bezet houdt en dat tevens wacht op het verkrijgen van één of meer andere bronnen die op dat ogenblik door andere processen bezet zijn.
- **Geen voortijdig ontnemen (No preemption):** Bronnen kunnen niet voortijdig worden afgenomen d.w.z. dat een bron alleen vrijwillig kan worden vrijgegeven door het proces dat deze bron in bezit heeft, nadat het proces zijn taak heeft beëindigd.
- **Wachten in een kring (Circular Wait):** Er moet een verzameling {*p0*,*p1*,…,*pn*} van

wachtende processen bestaan zodanig dat *p0* wacht op een bron die in het bezit is van *p1*, *p1* wacht op een bron die in het bezit is van *p2*, enz…, en *pn* tenslotte wacht op een bron dit in het bezit is van *p0*. Met andere woorden: elk proces wacht op een bron die reeds in gebruik is door een ander proces in de verzameling.

Deadlock preventie

Deadlock voorkomen: zorgen dat minstens één voorwaarde nooit optreedt?

- ~~Bron met wederzijdse uitsluiting~~ → noodzakelijk
- Proces zal bron bezet houden ~~en wachten~~ → vrijgeven indien nodig
- Er is ~~geen~~ voortijdig ontnemen → eerlijk? volgorde volgens schema
- Processen wachten in een kring

HO GENT

Deadlock preventie

Om een deadlock te voorkomen kunnen we er voor proberen zorgen dat minstens één van de voorwaarden voor een deadlock dus nooit optreedt. Sommige voorwaarden zijn echter **noodzakelijk**, en er kunnen dus aanzienlijke problemen ontstaan als we deze zouden opheffen.

Als er geen wederzijdse uitsluiting wordt afgedwongen, kunnen de activiteiten van het ene proces de voortgang van het andere proces beïnvloeden. Deze conditie mag dus niet worden verwijderd. Als processen wachten in een kring (hier kunnen we niet veel aan doen) zullen we dus aan de overige 2 voorwaarden zaken moeten aanpassen.

Alvorens met zijn uitvoering te beginnen moet elk proces alle bronnen die het nodig heeft verkrijgen. Als een proces alle bronnen tegelijkertijd moet aanvragen, heeft het mogelijks een aantal bronnen voor lange tijd onder beheer zonder ze daadwerkelijk te gebruiken. Dit beperkt de beschikbaarheid van de bronnen. Als het proces enkele bronnen vasthoudt en het vraagt nog een bron aan, en deze bron kan niet onmiddellijk aan dat proces worden toegewezen (d.w.z. het proces moet wachten),

dan moet het proces al de bronnen die het op dat ogenblik vasthoudt, **vrijgeven**.

Door de tweede conditie te verwijderen kan een bron nu echter wel met geweld van een proces ontnomen worden. We hebben dus nog iets nodig om het toekennen en afnemen van bronnen **eerlijk** te laten verlopen. Een mogelijkheid hiervoor is om alle processen te onderwerpen aan een lineair ordenings**schema**, en elk proces kan alleen bronnen in opklimmende **volgorde** verkrijgen.

Deadlock vermijden

Deadlock voorkomen: geen kans op deadlock

Deadlock vermijden:
- Kans op deadlock bijna onmogelijk maken, maar kan nog optreden
- Aanvragen die tot deadlock kunnen leiden weigeren
  - Omgeving blijft in veilige status
  - Moeilijk (onmogelijk?) om te implementeren:
    Hoe aanvragen beoordelen?

HO GENT

Deadlock vermijden

Het verschil tussen het vermijden en het voorkomen van een deadlock is dat bij deadlock vermijden een deadlock niet onmogelijk is, maar we proberen de kans op een deadlock zo goed als onmogelijk te maken.

Het idee is om de aanvragen voor bronnen die eventueel tot deadlock kunnen leiden, te weigeren. Hiervoor moet het OS alle aanvragen heel nauwkeurig onderzoeken. Ziet het OS dat de toewijzing van een bron het risico van deadlock met zich meebrengt, dan weigert het de gevraagde toegang en vermijdt het zo het probleem.

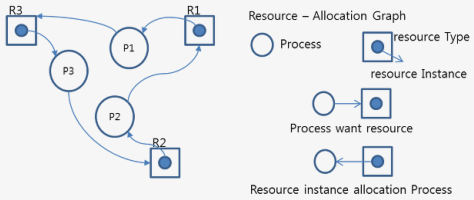
Door risicovolle aanvragen tot bronnen te weigeren blijft het systeem dus in een veilige toestand. Helaas is dit niet eenvoudig om te implementeren: hoe kan het OS oordelen of een aanvraag gevaarlijk is? Het systeem heeft hiervoor veel historische informatie nodig over het gebruik van elk bron per proces. Een algoritme die gebruikt kan worden om deadlocks te vermijden op basis van deze informatie is Banker's Algorithm (https://en.wikipedia.org/wiki/Banker's_algorithm).

4.4 Deadlocks

Deadlock signaleren

Hoe weet het OS dat er een deadlock is?

- Processen in BLOCKED toestand, is dit permanent?
- Deadlock opsporen via Resource Allocation Graph
  - Deadlock = cyclus in RAG
  - Algoritmes om cyclus in graaf op te sporen



HO GENT

Deadlock signaleren

Steeds wanneer een proces een resource aanvraagt is er deadlock mogelijk. We vragen ons af hoe het besturingssysteem deadlock kan signaleren en wat het besturingssysteem eraan doet als er een deadlock is.

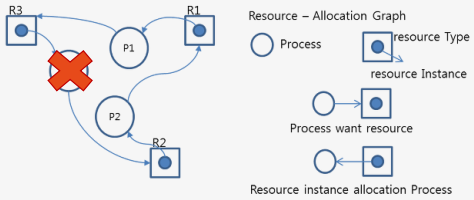
Eén manier om deadlock te signaleren, is een resource allocation graph. Dit is een georiënteerde graaf die gebruikt wordt om de resource-toewijzingen weer te geven. Een deadlock kunnen we signaleren door de **resource allocation graph** te bekijken. Als deze een cyclus bevat, is er een deadlock. Om cycli in een georiënteerde graaf te vinden, heeft het besturingssysteem diverse algoritmen ter beschikking.

4.4 Deadlocks

Deadlock herstellen

Wat doen als er een deadlock gevonden is?

- proces stoppen via KILL → bronnen vrijgeven ten koste van proces
- Doe rollback op proces
  - Deel van werk ongedaan maken en bronnen vrijgeven
  - Terugkeren naar starttoestand of checkpoint



HO GENT

Deadlock herstellen

Nu we weten hoe we een deadlock signaleren, rest ons nog 1 vraag: wat doen we eraan? Eén mogelijkheid is een proces gewoon maar af te breken en de eraan toegewezen bronnen terug vrijgeven. Hierdoor wordt de cyclus en dus ook de deadlock geëlimineerd ten koste van het proces.

Een andere mogelijkheid is een rollback op het proces uit te voeren. Hierbij worden alle eraan toegewezen bronnen vrijgegeven. Het proces verliest alle updates die het met gebruik van deze bronnen heeft gemaakt, en al het werk dat inmiddels was gedaan, maar wordt niet afgebroken. Het besturingssysteem brengt het terug in de toestand van vóór de aanvraag en toewijzing van de verwijderde bronnen. Dit kan overeenkomen met de oorspronkelijke start van het proces, of met een checkpoint. Een checkpoint treedt op wanneer een proces vrijwillig alle bronnen vrijgeeft. Door het gebruik van checkpoints kan elk proces eventueel verlies van werk echter zo klein mogelijk houden.