# H6 - Threads

HO GENT

# 6 Threads



HO GENT


6.1 Wat zijn threads?
HO
GENT

6.1 Wat zijn threads

# Processen: 2 concepten

- Een proces bestaat uit 2 concepten:
  - **Eigendom** van bronnen
  - **Uitvoering** van het programma
- Deze 2 concepten worden **onafhankelijk behandeld** door het besturingssysteem
  - Bronnen zoals bestanden blijven toegewezen aan proces, ongeacht of dit proces actief of geblokkeerd is

HO GENT

**Processen: 2 concepten**

Tot nu toe hebben we processen steeds behandeld als een onscheidbaar deeltje in het besturingssysteem. Een proces kan verschillende toestanden hebben via de scheduler, heeft toegang tot bestanden en rekenkracht, kan prioriteiten krijgen , ... .

Als we dieper gaan kijken naar wat een proces nu precies is, zien we dat we het idee van een proces eigenlijk hebben gebruikt voor twee aparte concepten: eigendom van bronnen en het uitvoeren van de instructies.

- Eigendom van bronnen: Het besturingssysteem kent bronnen zoals bestanden, geheugenruimte, apparaten, ... toe aan een proces. Het besturingssysteem zorgt er ook voor dat processen elkaars bronnen niet zomaar kunnen beïnvloeden.
- Het uitvoeren van de instructies: de fetch-execute cyclus haalt continu instructies op en voert deze uit. Hiervoor wordt er gebruik gemaakt van een Program Counter om de volgende uit te voeren instructie bij te houden, registers en een stack om informatie bij te houden, ... .

Deze twee concepten staan eigenlijk los van elkaar. Bijvoorbeeld, of een proces nu actief is op de CPU of geblokkeerd is: bronnen zoals bestanden blijven toegewezen

aan het proces.

# Proces: eigendom bronnen

- Elk proces krijgt controle of **eigenaar**schap over **bronnen** (locking)
- **Afgeschermd** van andere processen door het besturingssysteem
- Voorbeelden van bronnen:
  - Address space (geheugen voor procesbeeld: instructies, data, …)
  - Geheugen (in RAM, HDD, SSD, …)
  - Bestanden
  - Apparaten (bv. CPU, …)
  - …

HO GENT

# Proces: uitvoering programma

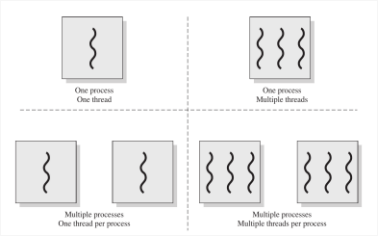
- Uitvoeren van **instructies**
  - Via fetch-execute cyclus
- Bijhouden **registers** en **stack**
  - Bv. program counter (PC)
- Scheduling
  - Bijhouden toestand/staat
  - Is proces actief/geblokkeerd/… ?

HO GENT

6.1 Wat zijn threads

# Processen vs. threads

- Uitvoeringsgedeelte proces = **thread**
- Binnen een proces:
  - Enkele thread: single-threaded
  - Meerdere threads: multi-threaded
- Multi-threading:
  - Parallelle taken binnen een proces mogelijk (via meerdere threads)




HO GENT

**Processen vs. Threads**

Een proces zal dus de verschillende instructies uitvoeren via de fetch-execute cyclus. Het uitvoeren van instructies binnen een proces gebeurt in een thread (NL: draadje). Daarom wordt in de context van uitvoering vaak over threads gesproken, terwijl we voor eigendom van bronnen meestal spreken over processen of taken.

Een proces kan echter meerdere threads bevatten. Een thread is de kleinste eenheid van geprogrammeerde instructies die onafhankelijk van elkaar kunnen worden beheerd door een scheduler, dus met andere woorden: binnen één proces kunnen gelijkertijd meerdere threads actief zijn.

Wanneer een proces bestaat uit één enkele thread voor de uitvoering, spreken we van een single-threaded proces. Tot nu toe kwamen alle processen in deze cursus overeen met single-threaded processen. Wanneer een proces echter zijn werk verdeeld over meerdere threads, spreken we van een multi-threaded proces. Deze threads kunnen parallel uitgevoerd worden door het systeem, wat vooral interessant is voor systemen met meerdere processoren en/of multi-core CPU's.

Threads kunnen zich – net zoals processen – in verschillende toestanden bevinden, zoals geblokkeerd, actief, ... . Het wisselen tussen threads op de CPU loopt bovendien analoog aan het wisselen tussen processen: er wordt een context switch voor threads uitgevoerd. Zoals we verderop zullen zien, is het wisselen tussen threads wel veel 'goedkoper' dan het wisselen tussen processen.

6.1 Wat zijn threads

# Opbouw threads

- Een thread **bestaat uit**:
  - Thread ID
  - Registers (Program Counter, …)
  - Stack
  - Toestand (status)
- Threads in een proces **delen**:
  - Instructies
  - Data
  - Toegang tot bronnen

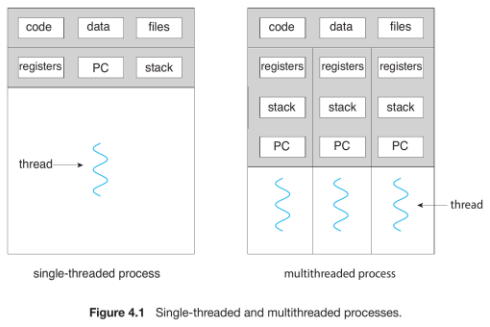



Figure 4.1 Single-threaded and multithreaded processes.

HO GENT

**Opbouw threads**

Net zoals een computer de CPU, geheugen, hardware, … deelt met één of meerdere processen, deelt een proces de instructies, data, toegewezen bronnen, … met één of meerdere threads. Elke thread van een proces heeft dus toegang tot alle bronnen toegewezen aan dat proces. Bijvoorbeeld: alle threads kunnen een bestand toegewezen aan het proces lezen en bewerken. Daarnaast heeft elk thread ook zijn eigen registers (bv. Program Counter), stack, toestand, … zodat het onafhankelijk van andere threads binnen het proces instructies kan uitvoeren.

Er is geen afscherming tussen threads binnen eenzelfde proces. Threads binnen eenzelfde proces kunnen elkaars gegevens lezen, schrijven, verwijderen, … zonder enige beperking. Dit is een groot verschil met processen waar het besturingssysteem processen van elkaar afschermt.

Threads hebben net als processen een toestand en kunnen dus ook op actief, gereed, … gezet worden. Een thread kan dus bijvoorbeeld gedeactiveerd worden en een andere thread van dat proces geactiveerd. Dit gebeurt net als bij processen met een context switch. De context voor een thread is de inhoud van diens registers (inclusief

de Program Counter) en de stack. Threads kunnen net als processen ook gesynchroniseerd worden. Omdat een thread echter minder data bevat dan een proces (instructies, data, toegang tot bronnen zijn immers gedeeld) is deze context switch dus wel 'goedkoper' dan een context switch tussen processen.

Let op! Er is vaak verwarring of de instructies behoren tot het proces of de threads. De instructies zelf (opgeslagen in de adress space in het geheugen) behoren tot het proces. Het proces heeft immers het geheugen toegewezen gekregen voor het procesbeeld door het besturingssystemen. De instructies in het geheugen zijn dus eigenlijk een soort van bron die toegewezen is aan dat proces. Het uitvoeren van de instructies is dan weer de verantwoordelijkheid van de threads: met behulp van hun eigen Program Counter kunnen ze de juiste instructie van het proces lezen om deze uit te voeren en nadien de volgende waarde voor de PC in te laden (via de fetch-execute cyclus).


6.2 Soorten threads
HO
GENT

6.2 Soorten threads

# User-Level Threads

- Programma gebruikt **bibliotheek** voor threading
- OS beschouwt programma als **single-threaded** proces

- Voordelen
  - Geen system calls van het OS nodig voor threading (**sneller**)
  - Proces heeft zelf controle over scheduling threads
  - Onafhankelijk van OS
- Nadelen
  - Als thread blokkeert op I/O, wordt hele process geblokkeerd
  - Geen gebruik van multiprocessing

HO GENT

**User-Level Threads**

Algemeen kunnen we een onderscheid maken tussen twee soorten van threads: User-Level Threads en Kernel-Level Threads.

In het geval van User-Level threads maakt het programma gebruik van een bibliotheek (library) voor het aanmaken en beheren van threads. Dit gebeurt onafhankelijk van het besturingssysteem, het besturingssysteem heeft dus geen weet van deze (software) threads en beschouwt het programma als een single-threaded proces.
Deze vorm van threading heeft enkele voordelen: het is niet nodig om gebruik te maken van system calls van het besturingssysteem voor threading, wat dus sneller is, en het proces heeft zelf de volledige controle over de scheduling van de verschillende threads. Bovendien is deze vorm van threading dus onafhankelijk van het besturingssysteem.
Doordat het OS het programma beschouwt als een single-threaded proces zijn er echter ook enkele belangrijke nadelen. Als één van de threads blokkeert, bijvoorbeeld wanneer deze thread wacht op het antwoord van een I/O operatie, wordt mogelijks het hele proces geblokkeerd door het OS. Daarnaast is het in deze vorm ook niet

mogelijk om gebruik te maken van multiprocessing – threads die behoren tot hetzelfde proces kunnen dus niet parallel uitgevoerd worden.

6.2 Soorten threads

# Kernel-Level Threads

- De logica voor threads zit in het besturingssysteem

- Voordelen
  - Scheduling mogelijk op threadniveau
  - Blokkeren van één thread geen invloed op andere threads process
  - Multiprocessing mogelijk
  - Besturingssysteem gebruikt waarschijnlijk zelf threads
- Nadelen
  - Trager (wisselen tussen programma en besturingssysteem)

HO GENT

**Kernel-Level Theads**

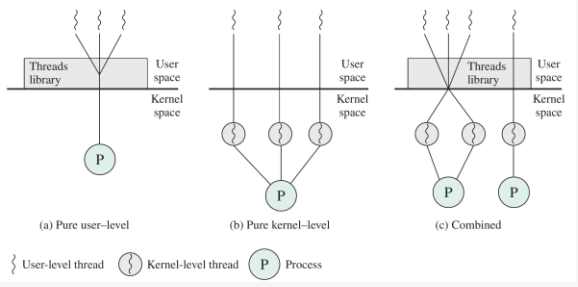
Een andere manier om threads te implementeren is door gebruik te maken van kernel-level threads. In deze vorm zit de logica voor threads in het besturingssysteem, het besturingssysteem is met andere woorden verantwoordelijk voor het aanmaken en beheren van de verschillende threads.

Deze vorm van threading heeft enkele voordelen. Een belangrijk voordeel is dat er nu scheduling mogelijk is op threadniveau (door het besturingssysteem). Het blokkeren van één thread binnen een proces zorgt er ook niet langer voor dat alle andere threads binnen dat proces geblokkeerd worden. Omdat het besturingssysteem de threads beheert, is er nu ook multiprocessing mogelijk: threads kunnen parallel uitgevoerd worden op verschillende processoren of processorkernen. Bovendien is het vaak zo dat het besturingssysteem zelf reeds threads gebruikt om verschillende taken in de achtergrond uit te voeren. Er is echter ook een belangrijk nadeel: doordat het besturingssysteem de threads beheert, is deze vorm trager. Voor aanmaken van threads moet er gewisseld worden tussen het programma en het besturingssysteem, terwijl bij user-level threads het aanmaken en beheren van threads volledig gebeurt binnen het programma zelf.

6.2 Soorten threads

# Combinatie ULT en KLT

- Combinatie van User-Level en Kernel-Level threads
- Beste van de twee werelden
- Kan op verschillende manieren




HO GENT

**Combinatie ULT en KLT**

Beide vormen – User-Level Threads en Kernel-Level Threads – hebben dus hun voor- en nadelen. Het is echter mogelijk om beide vormen te combineren, en zo het beste over te houden van beide vormen. Dit combineren kan op verschillende manieren, zoals geïllustreerd in de figuur in de slide.

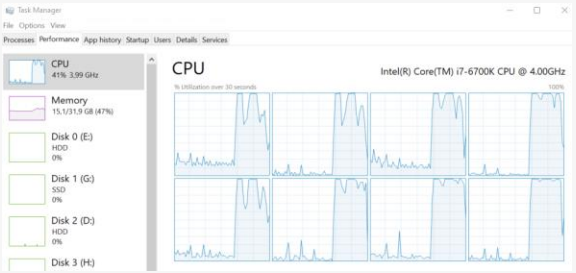
# 6.3 Voor- en nadelen

HO
GENT

6.3 Voor- en nadelen

# Voordelen multi-threading

- Niet noodzakelijk om hele proces te blokkeren
- Eenvoudiger om bronnen te delen
- Lichter en sneller dan processen (creatie, context switch, …)
- Efficiënt gebruik van multi-core CPU's (**parallellisme**)




HO GENT

**Voordelen van multi-threading**

Een proces kan zijn werk dus verdelen over meerdere threads. Het gebruik van meerdere threads binnen een proces heeft enkele voordelen ten opzichte van een single-threaded proces:

- Een interactief proces moet soms een operatie uitvoeren die lang kan duren. Bij een single-threaded proces moet het hele proces wachten tot deze operatie is afgerond. Een multi-threaded proces daarentegen kan de thread blokkeren en ondertussen een andere thread laten uitvoeren zodat het proces hier niet op hoeft te wachten. Bijvoorbeeld, indien de user interface en de langdurige operatie worden behandeld door eenzelfde thread lijkt het alsof het hele programma vast zit. De user interface zal immers niet reageren, want de thread is nog steeds bezig met de langdurige operatie. Mocht de user interface een aparte thread hebben, dan kan het proces nog steeds de thread van de user interface uitvoeren en lijkt het proces nog steeds actief te zijn voor de gebruiker.
- Het delen van bronnen tussen processen wordt nauwlettend afgeschermd door het besturingssysteem. Het is dus niet vanzelfsprekend om processen zomaar bronnen of berichten met elkaar te laten uitwisselen of te delen. Threads binnen een proces daarentegen kunnen wel zonder afscherming aan elkaar. Het is dus

eenvoudiger en efficiënter om threads binnen een proces met elkaar te laten communiceren dan processen onderling.

- Het maken van een proces verbruikt rekenkracht en geheugen. Het aanmaken van een thread binnen een proces is een stuk minder belastend en dus lichter en sneller (van x10 tot x100, afhankelijk van het systeem). Daarnaast is een context switch tussen threads ook lichter en sneller dan een context switch tussen processen. Ook het afsluiten van een thread is minder belastend voor het systeem dan het afsluiten van een proces.
- Een multi-threaded proces kan efficiënt gebruik maken van een multicore CPU's. Bijvoorbeeld, bij het unzippen van een groot bestand met 7-zip kan het werk opgesplitst worden in evenveel threads als CPU cores. Elke core voert dan een thread van het 7-zip proces uit en unzipt dus een deel van het bestand. Dit wordt ook wel parallellisme genoemd. Single-threaded processen daarentegen kunnen slechts gebruik maken van een enkele CPU core.

6.3 Voor- en nadelen

# Uitdagingen bij multi-threading

- Indelen van het rekenwerk in mogelijke threads
- Opsplitsen van data over de threads
- Afhankelijkheid van data tussen threads
- Testen en debuggen

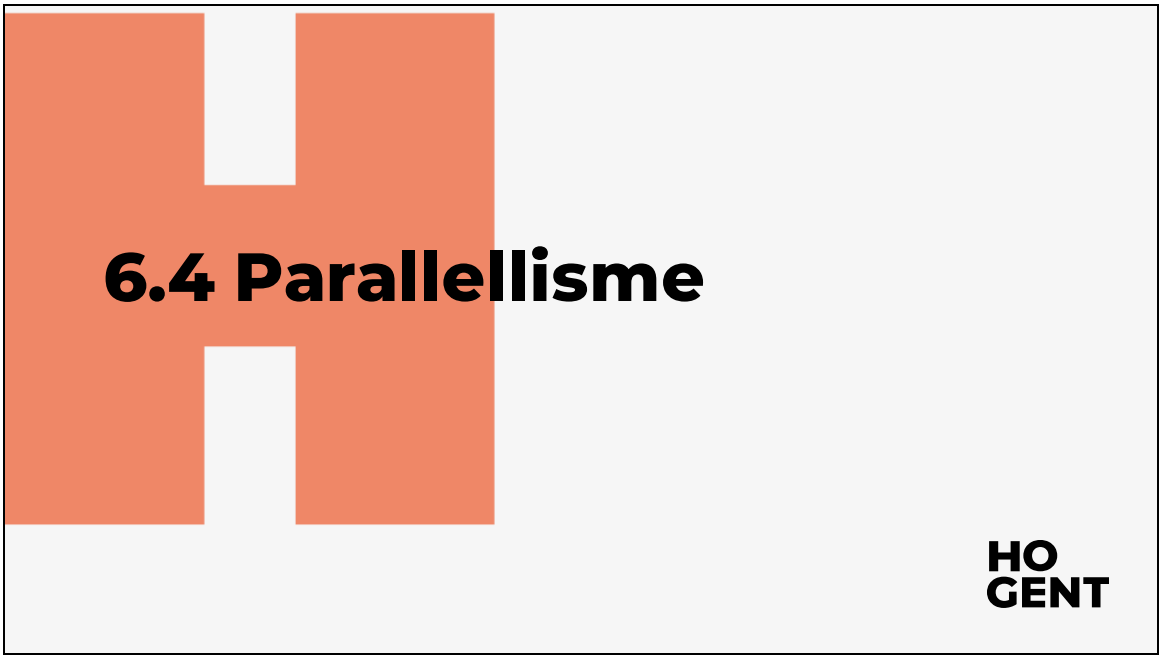
HO GENT

**Uitdagingen van multi-threading**

Het ontwikkelen van multi-threaded programma's brengt wel complexere problemen met zich mee dan het ontwikkelen van single-threaded programma's:

- Indelen van het rekenwerk in mogelijke threads: Rekenwerk dat zich kan opsplitsen in onafhankelijke problemen is ideaal voor het opsplitsen in threads. Het unzippen van een .7z-bestand is hier een mooi voorbeeld van. Er moet wel voor gezorgd worden dat het de moeite is om een thread op te richten voor een taak. Het is één ding om het rekenwerk te kunnen opsplitsen, maar als het opgesplitste rekenwerk minder werk is dan het creëren van een thread is het niet ideaal om hiervoor een aparte thread aan te maken.
- Opsplitsen van data over de threads: Als de data waarop gerekend moet worden kan opgesplitst worden over de verschillende threads, kunnen de threads onafhanklijk van elkaar goed doorwerken.
- Afhankelijkheid van data tussen threads: Als de data niet compleet kan opgesplitst worden over threads, dan moet er gesynchroniseerd worden tussen de threads. Er zal dus door sommige threads gewacht moeten worden.
- Testen en debuggen: Het schrijven van multi-threaded programma's is erg complex. Taken kunnen op meerdere manieren en in meerdere volgordes

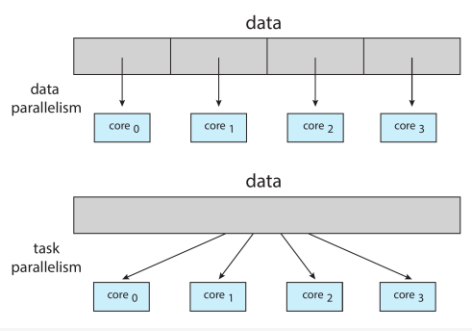
afgewerkt worden dan bij een single-threaded programma. Dit maakt het enorm moeilijk om bugs te vinden, te testen en op te lossen.


6.4 Parallellisme
HO
GENT

6.4 Parallellisme

# Soorten parallellisme

- **Data** parallellisme
- **Taken** parallellisme
- Vaak een **combinatie** van beide




HO GENT

**Soorten parallelisme**

Bij parallelisme gaan we dus een taak opsplitsen in verschillende subtaken, die onafhankelijk van elkaar uitgevoerd worden. Dit kan op verschillende manieren gebeuren.

Bij data parallellisme wordt de te verwerken data opgesplitst in stukken. Op elk stuk wordt dan dezelfde operatie uitgevoerd. De data wordt dus verdeeld over de threads. Een mooi voorbeeld hiervan is het optellen van de integers in een array. Bij een array van 240 integers en een systeem met 8 cores wordt de array verdeeld in 8 deelarrays van 30 integers elk. Er wordt dan voor elke deelarray een thread aangemaakt die de 30 integers in diens deelarray optelt. Op het eind moet enkel de resultaten van alle threads opgeteld worden. Een ander voorbeeld is het berekenen van het product van twee matrices: dit is een vrij rekenintensieve taak, en elke thread kan één rij, kolom of zelfs één cel van de resulterende matrix berekenen.

Taken parallellisme daarentegen verdeelt het werk op in verschillende threads. Niet elke thread voert dus dezelfde operatie uit. De threads kunnen werken op dezelfde of andere data. Een mooi voorbeeld hiervan is een tekstverwerkingsprogramma met

een thread voor de GUI, een thread voor spellingscontrole, een thread om het bestand weg te schrijven, een thread om wijzigingen door externe programma's aan het bestand te detecteren, een thread om te checken op updates, ... .

In de praktijk wordt vaak een hybride combinatie van de twee soorten parallellisme gebruikt.

6.4 Parallellisme

# Amdahl's Law

Geeft de **theoretische snelheidswinst** bij het toevoegen van CPU cores

- *S:* % van het programma dat niet versneld kan worden met meer CPU cores
- *N:* aantal CPU cores

$$snelheidswinst = \frac{1}{S + \frac{1-S}{N}}$$

Bv: Stel dat een programma voor 25 % bestaat uit een gedeelte dat niet versneld kan worden en voor 75 % uit een gedeelte dat wel versneld kan worden met meerdere CPU cores:

- Voor *N = 2* CPU cores: het programma zal 1.6x sneller draaien met 2 CPU cores dan met 1 core
- Voor *N = 4* CPU cores: het programma zal 2.29x sneller draaien met 4 CPU cores dan met 1 core

$$1.6 = \frac{1}{0.25 + \frac{1-0.25}{2}} \qquad 2.29 = \frac{1}{0.25 + \frac{1-0.25}{4}}$$

HO GENT

**Amdahl's Law**

De wet van Amdahl is een manier om een theoretische schatting te maken hoe het toevoegen van CPU cores de uitvoering van een programma versneld. De formule heeft 2 parameters nodig, namelijk *S* en *N*:

- *S* is het percentage van het programma dat sowieso niet versneld kan worden door meer CPU cores toe te voegen. Dit percentage wordt uitgedrukt als een kommagetal (bv. 25 % = 0.25).
- *N* is het aantal CPU cores.

De term *1 – S* in de formule is dus eigenlijk het percentage van het programma dat wel kan versneld worden door het toevoegen van CPU cores. Bv, als S = 0.25 (dus 25 %), dan is 1 - S = 1 - 0.25 = 0.75 (dus 75 %).

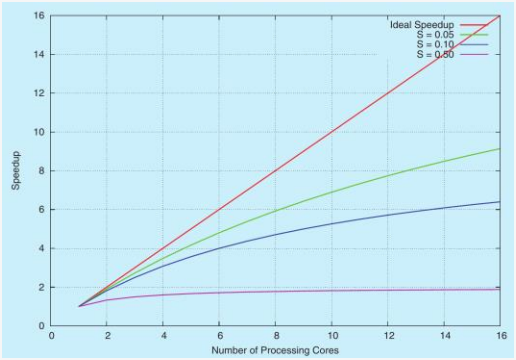
Bij een single-threaded proces is er geen snelheidswinst te behalen: het gedeelte dat niet versneld kan worden door meer CPU cores toe te voegen is immers 100 %, oftewel S = 1. Als we dit in de formule invullen bekomen we als snelheidswinst `1`: het programma zal dus 1x sneller draaien (m.a.w. er veranderd niets aan de snelheid). Bij een multi-threaded programma kan er vaak wel een snelheidswinst behaald

worden. In het voorbeeld zien we dat we een snelheidswinst van 160 % (1.6) halen bij 2 CPU cores en 229 % (2.29) bij 4 cores. We zien hier dus duidelijk dat multicore systemen met multi-threaded processen een aanzienlijke snelheidswinst kunnen geven.

6.4 Parallellisme

# Amdahl's Law

- Wat als N = ∞ (oneindig)?




HO GENT

**Amdahl's Law**

Stel nu dat we een machine hebben met 1000000 cores. Als we *N = 1000000* plaatsen in de formule van het vorig voorbeeld, dan komen we aan een versnelling van ongeveer 4. Deze winst zal niet meer verhogen, ook al voegen we bijvoorbeeld een miljard cores toe. Dit komt omdat *S = 0.25* (25 %) van het programma geen snelheidswinst heeft bij het toevoegen van meerdere cores. Zelfs als we de resterende 75 % over zoveel cores verdelen dat dat gedeelte van het programma bijna geen tijd inneemt, gaat er toch nog steeds 25 % tijd naar het *S*-gedeelte. Het programma kan dus nooit meer versnellen dan dat.

Ondanks dat threads voor een serieuze snelheidswinst kan zorgen, wordt deze snelheidswinst nog steeds beperkt door delen van het programma die niet versneld kunnen worden door multi-threading en multi-processing. Dit is bijvoorbeeld vaak het geval bij games. Veel zaken kunnen uitgespreid worden over meerdere threads, maar er is meestal één thread die de andere threads aanstuurt en beheert. Vaak is het deze thread die de snelheid bij multi-threaded games beperkt.

In de figuur zie je de impact van *S* op de snelheidswinst. Hoe groter *S*, hoe lager de

maximale snelheidswinst zal zijn ongeacht het aantal CPU cores.

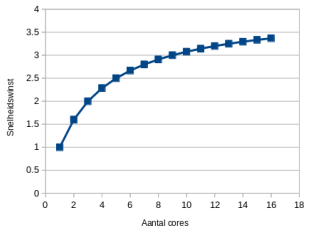
Meer informatie:

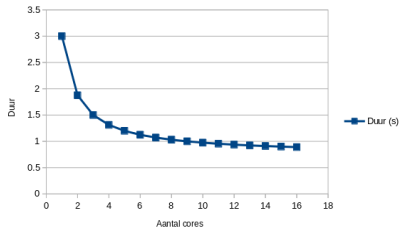
- https://de.slideshare.net/guest40fc7cd/threading-successes-02-supreme-commander
- http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=BB1D25AE0EA298CBFDF2EA7E00C94502?doi=10.1.1.567.332&rep=rep1&type=pdf
- https://vkguide.dev/docs/extra-chapter/multithreading/

# Amdahl's Law

- Wat als N = ∞ (oneindig)?

| Uitvoertijd bij 1 core (s) | 3 | |
|---|---|---|
| S | 0.25 | |
| | | |
| Aantal cores | Snelheidswinst | Duur (s) |
| 1 | 1.000 | 3.000 |
| 2 | 1.600 | 1.875 |
| 3 | 2.000 | 1.500 |
| 4 | 2.286 | 1.313 |
| 5 | 2.500 | 1.200 |
| 6 | 2.667 | 1.125 |
| 7 | 2.800 | 1.071 |
| 8 | 2.909 | 1.031 |
| 9 | 3.000 | 1.000 |
| 10 | 3.077 | 0.975 |
| 11 | 3.143 | 0.955 |
| 12 | 3.200 | 0.938 |
| 13 | 3.250 | 0.923 |
| 14 | 3.294 | 0.911 |
| 15 | 3.333 | 0.900 |
| 16 | 3.368 | 0.891 |




HO GENT

# 6.5 Voorbeelden

HO
GENT

# Voorbeeld: game

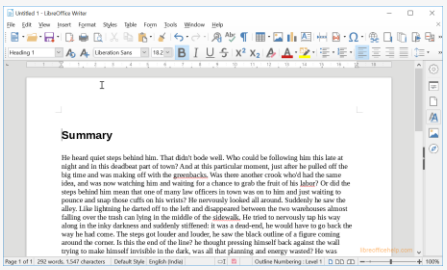
- Main thread: beheert alle andere threads
- Render thread: genereert de visuals
- Audio thread: speelt de audio af
- I/O thread: uitwisseling data met RAM, HDD/SDD, netwerk, …
- Extra threads, bijvoorbeeld:
  - Genereren wereld
  - Gedrag NPC's (AI)
  - Berekenen van paden (bv. AoE2)
  - …

Meer informatie:

- http://www.fragmentbuffer.com/docs/MultithreadingForGamedevStudents.pdf
- https://www.gdcvault.com/play/1022164/Multithreading

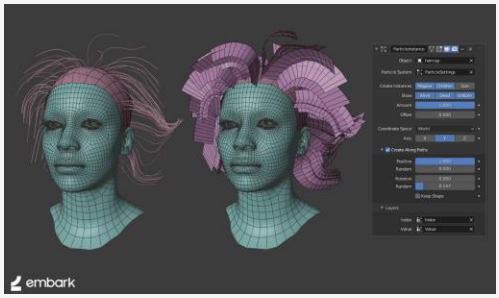
# Voorbeeld: word processor

- Thread voor UI
- Thread voor spellchecker
- Thread voor opslaan van document
- Thread voor detecteren veranderingen aan document door andere programma's
- Thread voor controleren updates
- ...



# Voorbeeld: 3D modelling

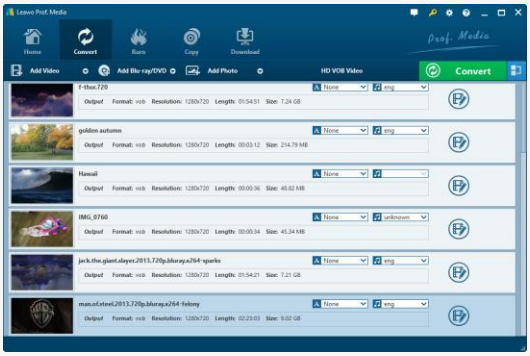
- Thread voor UI
- Werk voor rendering wordt verdeeld over meerdere threads voor parallellisme
  - Kan op CPU of op GPU




HO GENT

# Voorbeeld: video conversies

- Werk voor het converteren wordt verdeeld over meerdere threads voor parallellisme




HO GENT

# Bronnen


- Andrew S. Tanenbaum and Herbert Bos. 2014. *Modern Operating Systems* (4th. ed.). Prentice Hall Press, USA
- William Stallings. 2018. *Operating Systems: Internals and Design Principles, 9/e* (9th. ed.). Pearson IT Certification, Indianapolis, Indiana, USA
- Tanenbaum & Austin, 2013. *Structured Computer Organization* (6th. ed.). Pearson, USA
- Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. 2018. *Operating System Concepts* (10th. ed.). Wiley Publishing


HO
GENT