# Operating Systems

Samevatting Gilles De Praeter

Inhoud

[Operating Systems 1](#_Toc72178388)

[1. Inleiding besturingssystemen 3](#_Toc72178389)

[a. Wat is een besturingssysteem? 3](#_Toc72178390)

[b. Soorten besturingssystemen 3](#_Toc72178391)

[c. Concepten van besturingssystemen 3](#_Toc72178392)

[2. Virtualisatie & Cloud 5](#_Toc72178393)

[a. Wat is virtualisatie? 5](#_Toc72178394)

[b. Concepten van virtualisatie 5](#_Toc72178395)

[c. Multi-tenancy 6](#_Toc72178396)

[d. Cloud Computing 7](#_Toc72178397)

[3. Processen 9](#_Toc72178398)

[a. Van programma tot proces 9](#_Toc72178399)

[b. Opbouw van een proces 12](#_Toc72178400)

[c. Soorten processen 13](#_Toc72178401)

[d. Beheer van processen 13](#_Toc72178402)

[e. Scheduling 16](#_Toc72178403)

[4. Concurrency 21](#_Toc72178404)

[a. Wat is concurrency? 21](#_Toc72178405)

[b. Wederzijdse uitsluiting (mutual exclusion) 22](#_Toc72178406)

[c. Synchronisatie 24](#_Toc72178407)

[d. Deadlocks 26](#_Toc72178408)

[5. File systems 27](#_Toc72178409)

[a. Persistente opslag 27](#_Toc72178410)

[b. Files 28](#_Toc72178411)

[c. Directories 29](#_Toc72178412)

[d. File systems 30](#_Toc72178413)

[e. Partities 34](#_Toc72178414)

[f. Voorbeelden 35](#_Toc72178415)

[6. Threads 37](#_Toc72178416)

[a. Wat zijn threads? 37](#_Toc72178417)

[b. Soorten threads 38](#_Toc72178418)

[c. Voor- en nadelen 39](#_Toc72178419)

[d. Parallellisme 39](#_Toc72178420)

[e. Voorbeelden 40](#_Toc72178421)

# Inleiding besturingssystemen

## Wat is een besturingssysteem?

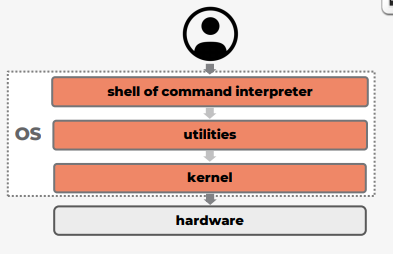
* Programma dat het mogelijk maakt de hardware van een computer te gebruiken
  + Hardware = CPU, geheugen, secundaire opslagapparatuur, randapparatuur, …
  + Gebruikers geven geen opdrachten aan de computer/hardware, maar aan het besturingssysteem
  + Het besturingssysteem geeft de hardware de opdracht om de gewenste taken uit te voeren
  + EN: Operating System (OS)
* Taken van een besturingssysteem:
  + Opslaan en ophalen van informatie
  + Programma’s afschermen
  + Gegevensstroom regelen
  + Prioriteiten regelen
  + Beheer en delen van bronnen
  + Tijdelijke samenwerking tussen programma’s mogelijk maken
  + Reageren op fouten
  + Tijdsplanning maken
* Enkele voorbeelden:
  + Windows
  + MacOS
  + Linux
  + Unix (een van de oudste)

## Soorten besturingssystemen

* Single-tasking:
  + Er is maar één gebruiker die maar één applicatie tegelijk draait. Er kan dus maar één programma (taak/task) tegelijk actief zijn.
  + Een bekend voorbeeld van een single-tasking besturingssysteem is MS-DOS, de (niet-grafische) voorloper van Windows
* Multitasking (single-user):
  + Meerdere programma’s (taken) kunnen tegelijkertijd geopend en uitgevoerd worden door één gebruiker.
* Multi-user systemen (ofwel multiprogrammering systemen)
  + Meerder gebruikers maken simultaan gebruik van de computerbronnen.
  + Moeten niet alleen alle gebruikers bijhouden, maar er moet ook voorkomen worden dat deze elkaar hinderen of in het werk van de anderen kunnen rondneuzen.

## Concepten van besturingssystemen

* Verschillende lagen:
  + Veel besturingssystemen implementeren de interface tussen gebruiker en computer als een reeks stappen of lagen.
  + De gebruiker communiceert met de bovenste laag. Deze bestaat uit routines van het besturingssysteem die zijn ontworpen om op opdrachten van een gebruiker te reageren. We noemen deze laag de shell of de command interpreter
  + De laag utilities bevat vele routines die voor deze dingen zorg dragen. Wanneer een gebruiker een opdracht geeft, verzorgen de utilities het grootste deel van de controle en de voorbereiding die voor de uitvoering nodig zijn.
  + De laatste laag is de kernel of kern. Dit is het hart van het besturingssysteem. Deze laag bevat de routines die het vaakst worden gebruikt en waar het meest op aan komt.



* Soorten taken:
  + Interactieve programma’s: een programma dat een gebruiker vanaf de terminal activeert. De interactieve gebruiker verwacht een snelle respons, en krijgt daarom voorrang. Werking: gebruiker geeft korte opdracht in -> besturingssysteem vertaalt opdracht en neemt actie -> Het zet een prompt-teken op het scherm -> gebruiker volgende opdracht..
  + Batch-programma’s: Een gebruiker kan opdrachten in een file opslaan en deze aan de batch queue (wachtrij voor batch-programma’s) van het besturingssysteem aanbieden. De batch-gebruikers verwachten geen directe respons.
  + Real-time programma’s: een programma dat een tijdbeperking legt aan de respons. Interactieve gebruikers geven de voorkeur aan een snelle respons, maar real-time gebruikers eisen dit zelf. Vb’en: robots, controlesysteem voor luchtverkeer, …
* Processen:
  + Een programma of taak bestaat uit één of meerdere processen.
  + Een proces bestaat uit één of meerdere reeksen opdrachten die door een besturingssysteem worden beschouwd als een werkeenheid.
  + Een besturingssysteem focust zich op de uitvoering van de processen. Het bepaalt welke processen momenteel worden uitgevoerd en welke bronnen toegekend worden aan welke processen.
* Resources:
  + Processen spreken in eerste instantie resources aan.
  + Persistente opslag (bestanden): files en records lokaliseren
  + Geheugen: instructies opslaan
  + CPU: instructies uitvoeren
  + Randapparatuur: printers, tapedrives, disks, …
* Een besturingssysteem moet:
  + Zorgen voor voldoende geheugen voor het proces, maar ook niet te veel en de toegang regelen.
  + Het gebruik van de CPU regelen.
  + De gegevensstroom regelen van of naar randapparatuur
  + Bestanden en records kunnen lokaliseren
* Scheduling: de manier waarop een besturingssysteem prioriteiten toekent aan elk programma of zelf elk proces.
* Concurrency:
  + Er kunnen conflicten ontstaan als processen gedeelde bronnen aanspreken.
    - Vb: 2 processen willen dezelfde printer aanspreken
  + Oplossing: synchronisatie: het besturingssysteem zal dit conflict moeten detecteren en oplossen door een volgorde op te leggen waarin processen afgehandeld worden
* Ontwerp criteria van een besturingssysteem:
  + Consistentie: Als het aantal processen vrijwel constant blijft, hoort dat ook voor de respons te gelden.
  + Flexibiliteit: een besturingssysteem hoort zodanig geschreven te zijn dat een nieuwe versie, het draaien van oude applicaties niet onmogelijk maakt. Ook moeten nieuwe randapparaten eenvoudig kunnen worden toegevoegd
  + Overdraagbaarheid: het besturingssysteem moet op verscheidene soorten computers werken. Dit zorgt voor meer flexibiliteit.
  + Vaak moet het ene criterium worden opgeofferd ten gunste van het andere.

# Virtualisatie & Cloud

## Wat is virtualisatie?

* Virtualisatie: het creëren van een virtuele versie van iets, zoals virtuele computer hardware, virtuele opslagapparaten of virtuele netwerkbronnen.
* Traditionele architectuur: Op een computer gebruiken we één besturingssysteem.
  + Je kan wel verschillende besturingssystemen naast elkaar installeren, maar deze kunnen niet tegelijk opgestart worden.
* Virtualisatiesoftware maakt het mogelijk om op één computer, meerdere computers te laten draaien.
  + Zorgt ervoor dat de fysieke hardware gedeeld wordt over de virtuele hardware
  + Virtuele harde schijven worden bewaard als bestanden op de fysieke harde schijf, zal elke VM een stukje van het fysieke geheugen reserveren en zullen virtuele processen m.b.v. scheduling elk een deel krijgen van de processorcapaciteit.
* Voordelen van virtualisatie:
  + Efficiënter gebruik van de beschikbare hardware
  + Veel goedkoper dan het gebruik van aparte fysieke computers
  + Lagere ecologische voetafdruk (minder stroom)
* Virtualisatiesoftware:
  + VMware: eerste fabrikant die virtualisatie op het Intel-platform mogelijk maakte
  + Microsoft Hyper-V
  + Xen, KVM en Oracle VirtualBox: gratis alternatieven voor VM’s
  + Parallels Desktop: geoptimaliseerd voor MAC (windows op mac laten draaien)
  + Docker: speciale vorm van virtualisatie: Linux kernel wordt gedeeld door meerdere containers

## Concepten van virtualisatie

* Virtuele machine: een computerprogramma dat een volledige computer nabootst, waar andere (besturings-) programma’s op kunnen worden uitgevoerd
  + Beheerd door een hypervisor
* Soorten virtuele machines:
  + Emulator: een VM die een volledige computer nabootst. Vb: VirtualBox
  + Programmeertaal-specifiek: Een virtuele machine voor een programmeertaal biedt een abstractielaag voor de werkelijke computer: een verzameling basisfuncties waar programma's in de programmeertaal gebruik van moeten maken om de functies van de computer aan te spreken. Dit is precies wat een besturingssysteem doet, en een dergelijke machine fungeert dus als besturingssysteem voor in die taal geschreven programma's; de reden dat van een virtuele machine wordt gesproken is dat de virtuele machine zelf vaak een programma is dat wordt uitgevoerd op een bestaand besturingssysteem. De virtuele machine is de verbindende laag tussen de uitgevoerde code en de computer hardware waarop het uiteindelijke programma wordt uitgevoerd, eventueel via een ander besturingssysteem. De op een virtuele machine uitgevoerde code spreekt niet direct de 'echte' hardware aan, maar gebruikt alleen de functies die worden aangeboden door de virtuele machine. 11Wanneer voor een bepaald platform (hardware en eventueel besturingssysteem) een virtuele machine gemaakt is, kan elk programma dat voor deze virtuele machine geschreven is, worden uitgevoerd. De implementatie van een virtuele machine kan in elke andere programmeertaal gebeuren. Op deze manier wordt platform-onafhankelijkheid bereikt: programma's kunnen worden uitgevoerd op elk systeem waarvoor de virtuele machine is geïmplementeerd. Een bekend voorbeeld hiervan is de Java Virtual Machine (JVM).
  + Applicatie-specifiek: vrij recent, er wordt meestal geen volledige computer nagebootst. Vb: Docker
* Hypervisor of Virtual Machine Monitor:
  + Is de software die gebruikt wordt om virtuele machines aan te maken en op te starten
  + Regelt de virtualisatie. Met een hypervisor kan één hostcomputer meerdere VM’s gelijktijdig laten draaien door de (hardware) bronnen virtueel te delen, zoals het geheugen en de processor.
* Type 1 hypervisor:
  + Ligt rechtstreeks op de hardware en kan dus rechtstreeks de nodige hardware bronnen aanspreken.
  + Er is geen onderliggend OS nodig
  + Vooral gebruikt op servers
  + Enige technische kennis nodig om te gebruiken
  + Je kan geen type 2 installeren als er al een type 1 is geïnstalleerd
  + Vb’en: VMWare ESXi, Citrix Xen, KVM & Microsoft Hyper-V
* Type 2 hypervisor:
  + Een programma boven op een besturingssysteem
  + Aanspreken van de hardware gebeurt via de functies van het besturingssysteem
  + Vooral gebruikt op persoonlijk toestel
  + Weinig technische kennis nodig, wel minder krachtig en efficiënt als type 1
  + Je kan meerdere type 2’s naast elkaar installeren
  + Vb’en: Oracle VirtualBox, VMWare Workstation, Parallels Desktop

## Multi-tenancy

* Single-tenant omgeving:
  + Voor één gezin
  + Lager gebruik van beschikbare ruimte
  + Meer afscherming
  + Duurder
* Multi-Tenant omgeving:
  + Bronnen worden gedeeld, in tegenstelling tot een dedicated of isolated omgeving
  + Een tenant (huurder) is een gebruiker of groep van gebruikers met gemeenschappelijke toegang
  + Multi-tenancy kan geïmplementeerd worden in verschillende vormen, zowel op niveau van hardware als software
  + Virtualisatie speelt een belangrijke rol bij multi-tenancy
  + Multi-tenancy is een belangrijk kenmerk van Cloud Computing
  + Voordelen:
    - Efficiënter gebruik van de beschikbare bronnen, want meerdere eindgebruikers kunnen bediend worden door één toestel of instantie van de applicatie
    - Daardoor zijn er lagere operationele kosten dus goedkoper voor de eindgebruiker
  + Nadelen:
    - Minder isolatie en verhoogde beveiligingsrisico’s, in het geval van een inbreuk op de beveiliging op één enkele instantie kunnen meerdere tenants worden getroffen
    - Minder prestatie-isolatie, grote tenants kunnen de prestaties van kleinere tenants negatief beïnvloeden
  + Een goede multi-tenant omgeving moet dus zorgen voor een duidelijke scheiding van zowel de gegevens (= data) als de prestaties. Hoewel multi-tenancy draait rond het delen van bronnen, zou dit eigenlijk niet merkbaar mogen zijn voor de eindgebruiker: de gedeelde instantie moet zich dus gedragen als een aparte (dedicated) instantie ten opzichte van elke tenant

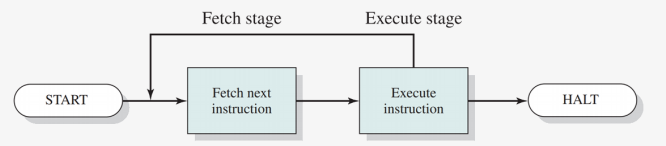
## Cloud Computing

* Cloud computing: Computerbronnen, zoals hardware, software en gegevens, worden op aanvraag beschikbaar gesteld via een netwerk (vaak het internet).
  + Een Cloud omgeving is vaak een collectie van een heel groot aantal servers (soms zelfs duizenden of meer), die zich bevinden in één of meerdere datacenters en waarop virtuele machines gemaakt kunnen worden voor de eindgebruikers.
* Kenmerken Cloud computing
  + Bronnen beschikbaar op aanvraag (on demand), vaak zonder tussenkomst van een fysiek persoon
    - Bronnen = hardware, software en/of opslag
    - Vaak: groot aantal servers in datacenter die door iedereen gehuurd kunnen worden, bijvoorbeeld als VM’s
  + Vaak via een pay-as-you-go pricing model (kostprijs afhankelijk van effectief verbruik)
  + Elasticiteit: mogelijkheid om automatisch aan te passen in functie van vraag
* Opkomst cloud computing:
  + Eerste definitie 1997 (Ramnath K. Chellappa)
  + 2000 opkomst Software as a Service
  + 2005-2007 opkomst Cloud Computing
    - Amazon, Google, IBM en universiteiten
    - Hand in hand met opkomst virtualisatie
  + VMWare, VirtualBox, Microsoft en Citrix
  + Analogie met mainframes (servers) en terminals (clients)
    - Geen centraal mainframe maar een gedistribueerde omgeving bij CP
* Cloud:
  + Wolk van computers
  + Virtualisatie van serveromgeving
    - Eindgebruiker weet niet waar de instanties precies draaien en soms ook niet op hoeveel servers
    - Eindgebruiker is geen eigenaar meer van hardware/software en niet verantwoordelijk voor onderhoud
  + Schaalbare, virtuele infrastructuur
* Deployment modellen:
  + Publieke Cloud omgeving
    - Beschikbaar voor iedereen, via het internet
  + Private Cloud omgeving
    - Toegang beperkt tot één of meerdere organisaties
    - Kan in privaat (self-hosted) of publiek (co-located) datacenter
    - De gebruiker volledige controle over data, beveiliging en kwaliteit van de dienst
    - De fysieke locatie van de infrastructuurcomponenten kan zowel de Cloud leverancier als de klant zelf zijn
  + Hybride Cloud omgeving
    - Combinatie van meerdere modellen
    - Voor- en nadelen van beide te combineren
    - vb: private data in binnenland bewaren, zware berekeningen in buitenland
  + Er bestaan ook vele afgeleide vormen
    - Bv. Community Cloud, Distributed Cloud, Multi-Cloud, …
* Service modellen:
  + Infrastructure as a Service (IaaS)
    - Infrastructuur virtueel beschikbaar voor gebruiker
    - Gebruiker heeft controle over besturingssysteem, software en (virtuele) hardware
    - Amazon EC2, …
  + Platform as a Service (PaaS)
    - Platform en diensten (bv. toegangsbeheer) aangeboden
    - Gebruiker heeft controle over software, maar geen controle over onderliggende hardware
    - Microsoft Azure, Google AppEngine, …
  + Software as a Service (SaaS)
    - Aanbieden applicaties
    - Gmail, Office 365, Salesforce, …
  + Er bestaan ook vele afgeleide vormen (BPaaS, FaaS, DaaS, DBaaS, SECaaS, …)
  + Het belangrijkste verschil tussen de modellen is welke delen (lagen) beheerd worden door de Cloud provider, en welke beheerd worden door de gebruiker van de clouddiensten.
* Virtualisatie speelt belangrijke rol binnen Cloud computing, enkele vb’en:
  + Virtuele machines, virtualisatie van opslag, netwerk virtualisatie, virtualisatie van applicaties en zelfs virtualisatie van volledige desktopomgevingen.
* Elasticiteit:
  + De mate waarin het aanbod zich afstemt op een stijgende of dalende vraag.
  + Er worden meer bronnen ingezet tijdens een piekbelasting en minder tijdens daluren.
  + Dit proces automatiseren door een aantal instanties te voorzien, en dit aantal kan wijzigen in de tijd afhankelijk van de huidige belasting
  + Het aantal benodigde instanties inschatten kan zowel proactief (voorspellen op voorhand) als reactief (als antwoord op wijziging in vraag) zijn.
  + Er worden zelflerende modellen gemaakt, die op basis van historische data een voorspelling doen voor de mogelijke belasting in de toekomst
* Over-provisioning:
  + Er worden meer middelen toegewezen dan er nodig zijn.
  + Kan onnodige kosten met zich meebrengen aangezien de meeste Cloud providers het aantal voorziene (=provisioned) bronnen aanrekenen.
* Under-provisioning:
  + Minder middelen beschikbaar dan nodig om de applicaties correct te laten draaien
  + Kan ervoor zorgen dat de applicaties heel traag reageren en soms zelf crashen
  + Mogelijks verlies van klanten tot gevolg
* Idealiter:
  + Er zijn altijd net voldoende bronnen voorzien, afhankelijk van de huidige belasting van de eindgebruikers, voor de verschillende applicaties vlot te laten draaien.
* Cloud computing is een mooi voorbeeld van multi tenancy.

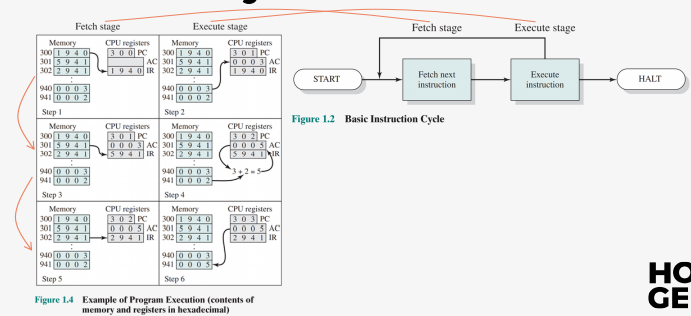
# Processen

## Van programma tot proces

* De instructieset van een type CPU: De bepaalde verzameling instructies die het type CPU begrijpt
* Compileren: Het proces waarbij een programma omgezet wordt van programmeertaal tot een verzameling instructies uit de instructieset van de CPU
* Verschillende types van de instructieset van de CPU:
  + x86: de 32-bits instructieset die vooral vroeger gebruikt werd AMD en Intel (desktops en laptops).
  + x86-64: de 64-bits instructieset die tegenwoordig door de meeste AMD en Intel processoren gebruikt wordt (desktops en laptops).
  + ARM: dit type processor word bv gebruikt op de Raspberry Pi, smartphones, tablets, … vanwege de lage kost en energieverbruik
  + MIPS: wordt gebruikt in embedded devices zoals routers, switches, printers, smartphones, tablets, supercomputers, …
  + JVM: De Java Virtuele Machine -> Java is hierop een uitzondering. Bij Java wordt er gecompileerd naar bytecodes. Dit is een soort tussentaal met instructies uit een intructieset voor een virtuele machine, de Java Virtual Machine (JVM). De programmeur moet de java code slechts eenmaal compileren naar bytecode, en het programma werkt dan op elk type CPU waarvoor een JVM bestaat. Het is dan immers de taak van de JVM om de bytecode om te zetten naar instructies voor de hardware. Dit bespaart heel wat werk voor de programmeur. De bytecode kan dus begrepen worden door elke JVM op elke processor die zo een JVM bevat.
  + 8051: wordt soms gebruikt in embedded devices en soms ook in het onderwijs
  + …. : er zijn ook nog andere
  + Een programmeur moet dus zijn programma compileren naar elk type CPU waarop het programma moet kunnen uitgevoerd worden. Hierdoor zijn er vaak verschillende versies van de software
* Instructiecyclus:
  + De computerhardware werkt steeds oneindig lang volgens hetzelfde stappenplan. Het eenmalig uitvoeren van dit stappenplan wordt ook wel een cyclus genoemd. De hardware voert dus oneindig lang cyclus na cyclus uit. In de meest eenvoudige vorm bestaat zo een cyclus uit 2 stappen, fetch en execute:
    - Fetch: Haal de volgende uit te voeren instructie op.
    - Execute: Voer deze instructie uit.
  + Na deze 2 stappen is de cyclus afgewerkt en begint de volgende cyclus.



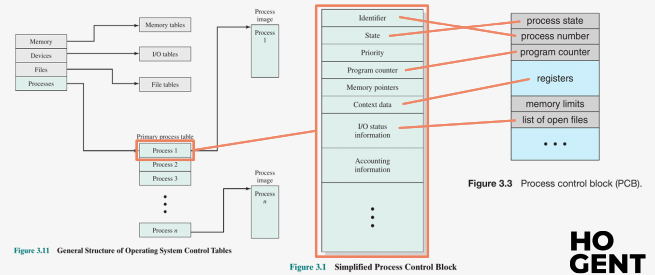
* Een vb:



* + De instructies voor de hardware van dit systeem zijn als volgt:
    - 1XXX: Kopieer de waarde op adres XXX in het geheugen naar het AC register.
    - 2XXX: Sla de waarde in het AC register op in het geheugen op adres XXX.
    - 5XXX: Tel de waarde op adres XXX in het geheugen op bij de waarde in het AC register en sla het resultaat op in het AC register.
  + We zien hier de volgende CPU-registers:
    - PC: Program Counter. Dit register houdt het adres bij van de uit te voeren instructie.
    - IR: Instruction Register. Dit register houdt de uit te voeren instructie bij. De hardware analyseert de bits in dit register om de handeling van de instructie uit te voeren.
    - AC: Accumulator. Een hulpregister voor tussenresultaten op te slaan van bewerkingen.
* Interrupts:
  + Bij computersystemen treden er soms events op: er wordt een nieuw apparaat aangesloten, de gebruiker voert iets uit, er treed ergens een fout op, er is een timer afgegaan, … . De computer moet dan hierop kunnen reageren. Zulke onderbreking noemt men een interrupt.
  + De fetch-execute cyclus kan uitgebreid worden om interrupts mogelijk te maken, hiervoor wordt na de execute stap een extra stap voor interrupts toegevoegd in de cyclus. Bij de interrupt stap wordt gekeken of er een interrupt is opgetreden:
    - Er is geen interrupt opgetreden: Dan gaat men door naar de volgende cyclus.
    - Er is een interrupt opgetreden: De huidige uitvoering van het programma wordt onderbroken. De toestand van het programma wordt opgeslagen. Zo wordt ondermeer het PC-register opgeslagen, zodat niet vergeten wordt op welke plek het programma werd onderbroken. Daarna voert het systeem de instructies om de interrupt af te handelen uit via de fetch-execute-cyclus. Het wisselen van het ene programma naar een ander programma wordt ook een context switch genoemd. Deze verzameling instructies noemt men ook wel de interrupt handler. Als de interrupt is afgehandeld, wordt de toestand van het oude programma herstelt en wordt het oude programma verder uitgevoerd.
* Binary of executable: het bestand gegenereerd door een compiler
  + Bij windows worden programma’s gecompileerd naar het PE-formaat (Portable Executable). Ook bekend als .exe-formaat.
  + Bij Linux worden programma’s gecompileerd naar het ELF formaat (Executable and Linkable Format).
  + Bij Mac worden programma’s gecompileerd naar het Mach-O formaat (Mach Object).
* Verschillende besturingssystemen bieden verschillende functies aan programmeurs om bepaalde dingen te doen.
  + Programmeurs moeten dus voor elk besturingssysteem een specifieke versie maken en compileren.
  + Programma’s gecompileerd op het ene besturingssysteem kunnen dus niet zomaar uitgevoerd worden op een ander besturingssysteem
  + Er bestaan wel programma’s die dit proberen mogelijk te maken, maar deze zijn niet altijd bruikbaar
    - Vb: WINE op linux
* Van binary tot proces:
  + Eenmaal de binary is aangemaakt, is dit simpelweg een bestand dat opgeslagen is op de harde schijf of SSD.
  + Wanneer de gebruiker het programma wil uitvoeren, worden de instructies gekopieerd naar het RAM-geheugen. Deze instantie van het programma in het RAM-geheugen wordt een proces genoemd.
  + Het bestand op de schijf (het programma) is dus niet hetzelfde als het proces. Het bestand op de schijf (het programma) is iets passief: pas wanneer een instantie daarvan wordt ingeladen in het RAM-geheugen krijgen we iets actiefs, namelijk het proces.

## Opbouw van een proces

* Proces:
  + Een instantie van een programma dat uitgevoerd wordt op het systeem
  + Een soort container die alles bevat om een programma uit te voeren
  + Dankzij processen kunnen besturingssystemen programma’s die worden uitgevoerd beheren.
  + Ook zorgt het gebruik van processen ervoor dat er meerdere programma’s tegelijkertijd uitgevoerd kunnen worden.
  + De adress space van een proces: het deel van het RAM-geheugen dat het proces kan aanspreken om de inhoud van te lezen of er naar te schijven.
  + De adress space bevat onder andere de uitvoerbare instructies van het programma en data van het programma.
* Proces image:
  + Hoe het proces er uit ziet in het RAM-geheugen.
  + Het is een momentopname van de address space van een proces. Dit bevat in het algemeen de volgende zaken:
    - Text: de uit te voeren instructies.
    - Data: globale variabelen.
    - Stack: tijdelijke opslag voor variabelen, functie parameters, adressen voor return uit functies, ... . Als er geen plaats meer is op deze stack, door bijvoorbeeld een te diepe of oneindige recursie, krijgt men de befaamde stackoverflow error.
    - Heap: dynamisch gealloceerd geheugen. Als iets te groot is om op de stack te plaatsen, of als er iets beschikbaar moet zijn over de grenzen van functies heen, dan is de stack geen bruikbare optie. In dat geval kan er gebruik gemaakt worden van de heap om iets dynamisch te alloceren. De heap werkt niet zoals een stack: wat er wordt bewaard, blijft beschikbaar tot het door de programmeur wordt opgekuist.
  + De exacte vorm van een process image is sterk afhankelijk van besturingssysteem tot besturingssysteem.
* Program control block (PCB):
  + Process table is een overzicht dat het besturingssysteem heeft van alle processen die actief zijn op het systeem. Dit overzicht heeft de vorm van een tabel. Elk element in die tabel noemt een process control block. Dit element bevat informatie over een proces.
  + Het bevat algemeen volgende informatie:
    - Informatie om het proces uniek te identificeren.
    - Informatie om de staat van het proces bij te houden bij context switches.
    - Informatie om het proces te beheren.
  + De vorm van een PCB is sterk afhankelijk van het besturingssysteem

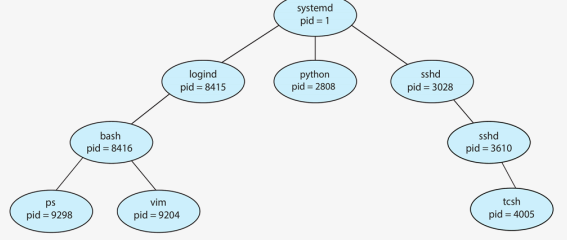


## Soorten processen

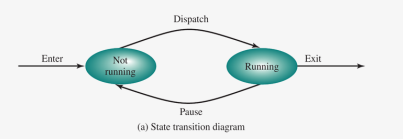
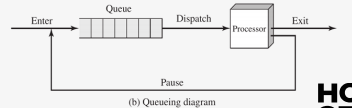
* Iteratief proces:
  + Opstarten en controleren vanuit een terminal sessie
  + Kunnen zowel in voorgrond (foreground) als achtergrond (background) draaien
  + Foreground proces: blokkeert terminal zolang het loopt
  + Background proces: blokkeert terminal enkel bij opstart van het proces, nadien kan terminal andere taken uitvoeren
* Automatische processen:
  + Ook wel “batch” processen genaamd
  + Verzameling van processen die in een wachtrij worden geplaatst wachtend op uitvoering
  + Bij uitvoering worden alle processen één voor één uit de wachtrij uitgevoerd. Dit volgens het First In, First Out (FIFO) principe
  + Voorbeeld: automatisch een back-up maken, elke dag om middernacht
* Daemons:
  + Processen die continu draaien
  + Veelal gestart bij opstarten van systeem
  + Wachten in de achtergrond tot ze nodig zijn
  + Ook gekend als services
  + Voorbeeld: onedrive synchronisatie client, e-mail server, …

## Beheer van processen

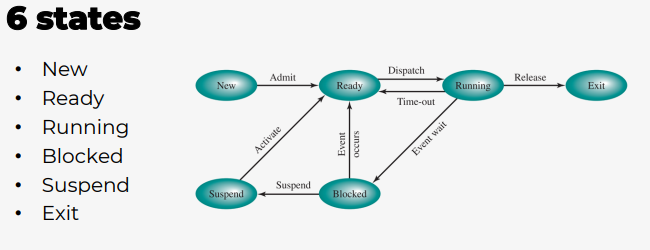
* Het ontstaan van processen verschilt van besturingssysteem tot besturingssysteem
* We bekijken hoe een proces ontstaat bij Linux:
  + Elk proces op een linux systeem heeft een uniek ID-nummer, de Process Identifier (PID).
  + Bij het opstarten start het eerste proces met PID 1. Bij nieuwere systemen is dit systemd, bij oudere systemen kan het het innit proces zijn.
  + Systemd is de moeder van alle processen bij recentere linux distrubuties.
  + Alle andere pocessen worden aangemaakt als kinderen door een ouderproces, waardoor een boomstructuur ontstaat



* + *In het voorbeeld hierboven zien we het moederproces systemd dat de kindprocessen logind (om ervoor te zorgen dat gebruikers zich kunnen inloggen), python (om python programma’s uit te voeren), en sshd (om ervoor te zorgen dat gebruikers kunnen inloggen via SSH) heeft aangemaakt. Een gebruiker heeft zich ingelogd op het toestel, waardoor logind het kindproces bashd heeft aangemaakt om de gebruiker een bash shell te geven. Daarin heeft de gebruiker de commando’s ps en vim ingetypt en zo bash de opdracht gegeven om de ps en vim kindprocessen aan te maken. Daarnaast zien we ook dat iemand zich heeft ingelogged via SSH (zie het kindproces sshd van sshd) waardoor de tcsh shell is aangemaakt als kindproces van sshd.*
* Het aanmaken van een proces in Linux bestaat uit het aanroepen van 2 functies:
  + Fork(): Deze functie maakt een exacte kopie van het proces in het RAM geheugen (een kopie van diens address space dus) en vult de PID van het proces in met een nieuw ongebruikt procesnummer. Daarnaast vult de functie het PPID (Parent Process Identifier) in met het PID van het ouderproces.
  + Exec(): Het nieuw aangemaakt kindproces wordt overschreven met de nodige waarden voor het gewenste proces. Zo worden bijvoorbeeld de juiste instructies, waarden, ... ingelezen naar het procesbeeld van het kindproces.
* De afbraak van een proces:
  + Wanneer een proces zijn taak heeft volbracht of er een fout optreedt, wordt het afgesloten.
    - Linux: exit() commando -> getal 0 als argument indien het goed is afgewerkt, een ander getal als het is afgesloten door een fout. Dit getal vertelt dan welke fout er is opgetreden.
  + Als het proces wordt afgesloten, wordt het procesbeeld verwijderd en alle resources (bv. Bestanden, geheugen, ...) dat het proces in gebruik had, worden terug vrijgegeven.
  + De PCB in de process table mag wel nog niet meteen vrijgegeven worden. Het bevat immers de foutcode dat aangeeft waarom het proces is afgesloten. Als de PCB ook meteen verwijderd zou worden, kan het ouderproces niet achterhalen waardoor zijn kindproces is afgesloten.
  + Zombieproces: een proces waarvan enkel de PCB nog bestaat. De PCB wordt snel na het afsluiten van het kindproces gelezen door het ouderproces, waardoor het PCB mag verwijderd worden en dus het kindproces volledig weg is.
  + Orphan proces: een zombieproces dat nooit wordt afgesloten doordat de foutcode van het zombieproces nooit wordt gelezen. Bv doordat Het ouderproces is afgesloten alvorens de foutcode te lezen. Het besturingssysteem kan hier niet meer aan en de PCB’s van zulke processen vervuilen dus de process table van het OS.
* 2-state scheduling model:
  + Simpelste scheduling model dat er is waarbij het Proces 2 states (=staten) heeft
  + Running: dit proces wordt uitgevoerd op de CPU.
  + Not Running: dit proces wordt niet uitgevoerd op de CPU.
  + We zien dat processen die staan te wachten op de CPU in een wachtrij (queue) worden geplaatst en aanschuiven voor CPU-tijd. Nieuwe processen worden aangemaakt, krijgen een PCB in de process table, en worden dan toegevoegd aan de queue. Nieuwe processen hebben dus steeds de Not Running toestand (tenzij ze meteen de CPU mogen gebruiken)

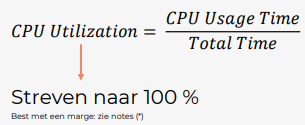
* 5 states scheduling model:
  + New: Een nieuw proces aangemaakt door het besturingssysteem. Meestal een nieuw proces waarvan de PCB al is toegevoegd aan de process table, maar dat nog niet volledig is ingeladen in het geheugen. Soms wordt er ook een limiet gezet op het aantal processen in de wachtrij: als deze vol zit mag een proces nog niet overgaan van New naar Ready tot de wachtrij vrije plaatsen heeft.
  + Ready: Een proces dat wacht tot het op de CPU mag.
  + Running: Het proces wordt uitgevoerd op de CPU.
  + Blocked: Een proces dat staat te wachten op iets (zoals lezen/schrijven uit of naar het RAM, geheugen, harde schijf, SSD, netwerk, ...).
  + Exit: Een afgewerkt proces. Let op, de exit toestand zegt niets over het feit dat het proces correct is afgewerkt of door een fout is afgesloten. De scheduler heeft daar niets mee te maken, het weet alleen dat hij met dit proces geen rekening moet houden: het is op hoe dan ook een of andere manier afgerond.
  + Enkele belangrijke overgangen:
    - Ready → Running: Het proces dat als eerste staat in de wachtrij mag op de CPU.
    - Running → Ready: Processen mogen vaak slechts een bepaalde maximumtijd op de CPU (zie bv. time sharing). Die maximumtijd wordt ook wel quantum of time slice genoemd. Wanneer de tijd verlopen is, wordt het proces van de CPU gehaald en achteraan de wachtrij geplaatst. Het onderbreken van een proces wordt ook wel pre-emption genoemd.
    - Running → Blocked: Het proces moet wachten op iets (bv. I/O) en wordt van de CPU gehaald zodat deze gebruikt kan worden door een ander proces dat de CPU-tijd kan gebruiken.
    - Blocked → Ready: het proces heeft gedaan met wachten (bv. I/O). Dit is vaak het resultaat van een gebeurtenis (event) (bv. de data is gelezen van de harde schijf). Het proces heeft opnieuw CPU-tijd nodig en wordt dus achteraan de wachtrij geplaatst.
* 6 states scheduling model:
  + Processen in het 5 states scheduling model bevinden zich in de RAM. Deze kan soms vol zitten waardoor het systeem vast loopt. Om dit op te lossen kunnen bepaalde processen die staan de wachten uit het RAM-geheugen verplaatsen naar de harde schijf of SSD om RAM-geheugen vrij te maken. Dit wordt ook wel swapping genoemd. Zulke processen bevinden zich dan in de Suspend toestand. Wanneer het systeem terug voldoende ademruimte heeft in het RAM-geheugen, kan het systeem terug processen vanop de harde schijf of SSD verplaatsen naar het RAM geheugen. De processen krijgen dan terug de Ready toestand en mogen weer aanschuiven in de wachtrij.



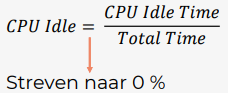
* Swapping (suspend):
  + Het is een intensief proces: de harde schijf of SSD is veel trager dan het RAM-geheugen waardoor swappen het systeem enorm vertraagd
  + Op windows wordt er weggeschreven naar de pagefile.sys en swapfile.sys bestanden
  + Op linux is er vaak een aparte partitie nodig
  + Debat of het nog relevant is vanwege de enorme RAM-capaciteit tegenwoordig

## Scheduling

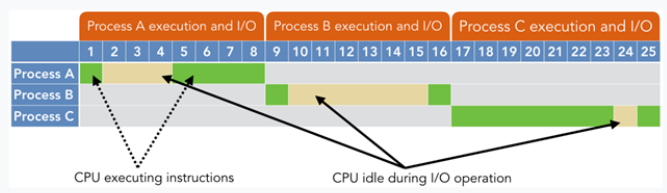
* CPU utilization:
  + Het percentage tijd dat de CPU in gebruik is

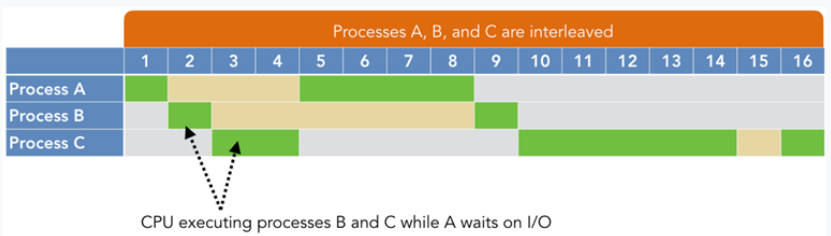


* + *(\*) Een besturingssysteem dat continue aan 100% draait is ook niet ideaal. Indien er dan een interrupt binnenkomt, is er immers geen CPU-tijd beschikbaar om deze interrupt af te werken. M.a.w. een systeem waarvan de CPU volledig aan 100% werkt, zal niet meer lijken te reageren tot de taken die de CPU opeisen afgewerkt zijn of minder CPU-tijd vragen. Er wordt dus best gestreefd naar een goed compromis tussen een zo hoog mogelijke CPU utilization en een marge voor interrupts.*
* CPU idle:
  + Als een CPU niets staat te doen

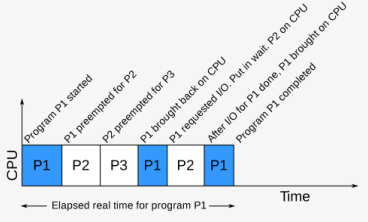


* Multiprogramming:
  + CPU zo optimaal mogelijk gebruiken
    - Sommige processen moeten wachten op iets (zoals bv. I/O) omdat deze trager zijn dan de CPU
  + Wanneer een proces moet wachten, kan een ander proces gebruik maken van de CPU. Als het proces klaar is met wachten, vraagt het terug CPU-tijd aan.

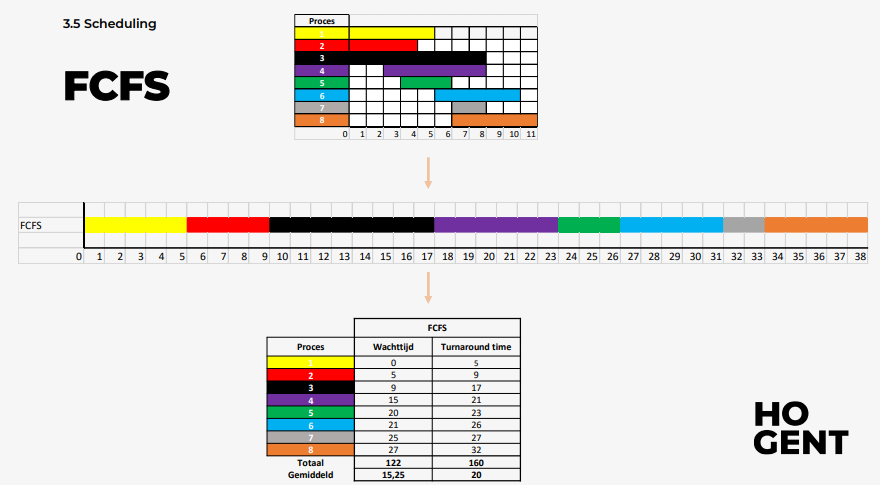




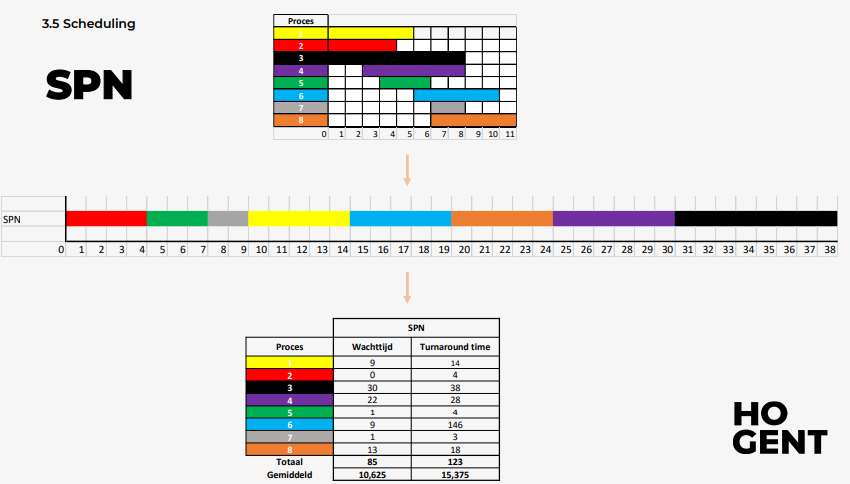
* Time sharing:
  + Multitasking
  + Multi-user
  + Processen op de CPU wisselen elkaar (snel) af
  + Geeft de illusie dat processen tegelijkertijd worden uitgevoerd



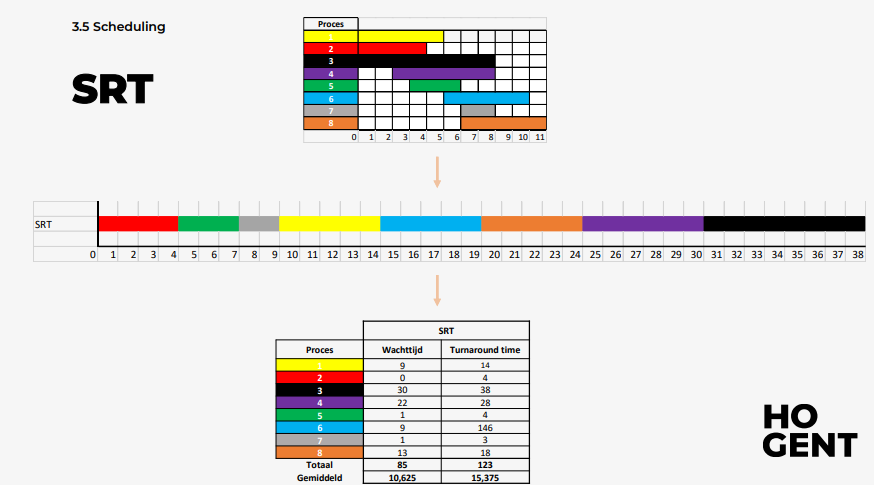
* Context switch:
  + Treed op bij het wisselen van een proces
  + Er wordt een snapshot genomen van het volledige proces en bewaard in het geheugen. Dan wordt de snapshot van het andere proces ingeladen en dat proces uitgevoerd. Als het terug de beurt is aan het eerste proces, wordt er opnieuw een snapshot van het tweede proces bewaard en de snapshot van het eerste proces ingeladen. Het proces doet verder vanwaar de snapshot is opgenomen alsof er niets is gebeurd.
  + Het nemen van een snapshot heeft wel een kleine overhead: verschillende data moeten worden opgeslagen, zoals de stack, heap, data, registers (PC, ...), ... en dit neemt een klein beetje tijd in beslag. Hetzelfde geldt voor het terug inladen van een snapshot.
* Scheduler:
  + Het onderdeel van het OS dat verantwoordelijk is voor het bepalen wanneer welk proces CPU-tijd krijgt.
  + Afweging tussen verschillende soorten processen:
    - Batch processen: processen die een na een moeten worden uitgevoerd en vaak weinig tot geen interactie hebben met de gebruiker. Vaak gaat het om een lijst van opdrachten die gegeven wordt aan de computer, die het systeem een na een afwerkt. Hierbij is een wachtrij systeem vaak voldoende.
    - Interactive processen: de processen op command line of GUI waarmee we het meeste vertrouwd zijn. Deze processen geven ook vaak resultaten terug weer. Als de scheduler slecht werkt, dan lijkt het alsof het programma vast blijft hangen.
    - Real time systemen: systemen die een hoge snelheidsrespons nodig hebben. Bv streamen van video’s. Het is dus goed voor de scheduler om deze processen voorrang te geven.
  + Het ontwikkelen en programmeren van een scheduler is dus vaak een complexe uitdaging
* Starvation:
  + In sommige gevallen kan het zijn dat bepaalde processen nooit CPU-tijd krijgen
  + Mogelijk scenario:
    - De scheduler geeft voorrang aan korte processen
    - Er komen steeds nieuwe korte processen in het systeem
    - Hierdoor worden lange processen telkens uitgesteld
* Verschillende algoritmen voor schedulers om CPU-tijd toe te wijzen aan processen:
  + Kunnen onderverdeel worden in 2 categorieën:
    - Non-preemptive: algoritmes die geen gebruik maken van context switches en dus pas wisselen van proces wanneer een proces volledig is afgewerkt.
    - Preemptive: algoritmes die gebruik maken van context switches om processen op de CPU te onderbreken en om te wisselen met een ander proces. Dit kan bv nodig zijn indien een proces het systeem kan monopoliseren. Het zorgt wel voor extra overhead.
* Wachttijd: tijd dat het proces in het systeem aanwezig was, maar niet uitgevoerd werd
* Turnaround time:
  + Tijd dat het proces in het systeem aanwezig was vanaf aanbieden tot effectief uitgevoerd zijn
  + = wachttijd + uitvoeringstijd
* FCFS (First Come First Serve):
  + Een non-preemptive algoritme
  + Proces wordt volledig uitgevoerd, geen onderbrekingen
  + Processen worden in wachtrij geplaatst indien nodig
  + Volgende proces? Eerst proces uit de wachtrij
  + + : Geen starvation, heel gemakkelijk
  + - : Korte processen kunnen lang moeten wachten, Proces kan systeem monopoliseren



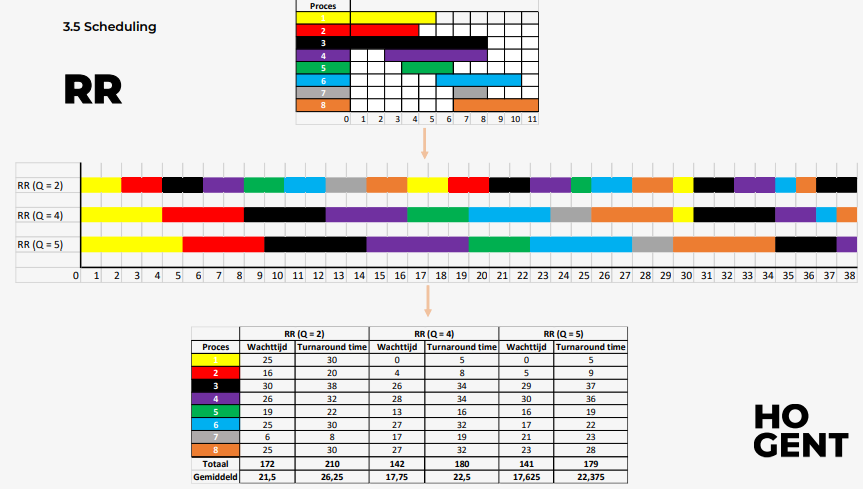
* SPN (Shortest Proces Next):
  + Een non-preemptive algoritme
  + Proces wordt volledig uitgevoerd, geen onderbrekingen
  + Processen worden in wachtrij geplaatst indien nodig
  + Volgende proces? Kortste proces uit de wachtrij
  + + : Korte processen zijn snel uitgevoerd
  + - : Starvation voor lange processen, Proces kan systeem monopoliseren

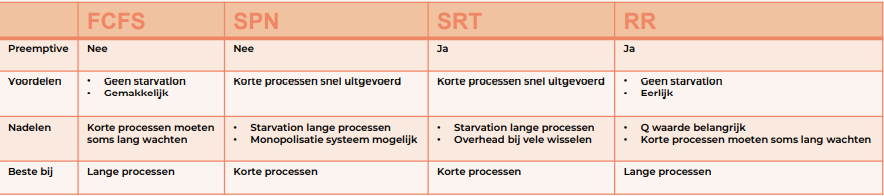


* SRT (Shortest Remaining Time):
  + Een preemptive algoritme
  + Telkens een nieuw proces ontstaat wordt een afweging gemaakt welk proces het minst tijd op de CPU zal nemen.
  + Proces wordt onderbroken, als er zich een beter (= korter) proces aandient
  + Processen worden in wachtrij geplaatst indien nodig
  + Volgende proces? Proces met kortste overgebleven uitvoeringstijd uit de wachtrij
  + + : Korte processen zijn snel uitgevoerd
  + - : Starvation voor lange processen, Overhead bij vele wisselen



* RR (Round Robin):
  + Elk proces wordt beurtelings Q eenheden uitgevoerd
  + Processen worden in wachtrij geplaatst indien nodig (bv als het proces nog niet klaar was)
  + Volgende proces? Volgende wachtrij uit wachtrij, laatst uitgevoerde wordt achteraan terug toegevoegd
  + + : Eerlijk, geen starvation
  + - : Waarde voor Q heel belangrijk, Korte processen moeten soms lang wachten, Overhead bij vele wisselen

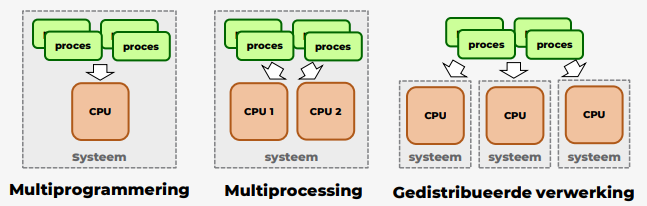




# Concurrency

## Wat is concurrency?

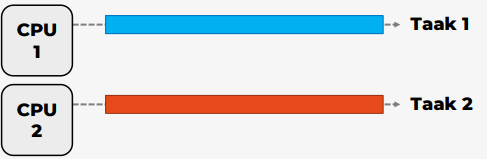
* Beheer van meerdere processen:
  + Multiprogrammering: het beheer van meerdere processen in een systeem met 1 processor
  + Multiprocessing: het beheer van meerdere processen in een systeem met meerdere processoren
  + Gedistribueerde verwerking: het beheer van meerdere processen die worden uitgevoerd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



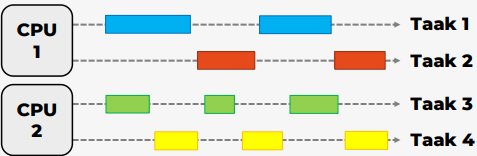
* Een multiprocessor:
  + Een computersysteem met twee of meer processoren
  + Meerdere taken gelijktijdig uitvoeren. Dit verhoogt de productiviteit, maar zorgt ook voor enkele uitdagingen:
    - Communicatie tussen processen
    - Delen van, en vechten om bronnen
    - Synchronisatie van meerdere procesactiviteiten
    - Verdelen van processortijd over processen
* Concurrency (ofwel parallelle processen):
  + 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.
* Sequentiele processen zijn het tegengestelde van concurrency, waarbij alle stappen strikt na elkaar worden uitgevoerd, en er dus van gelijktijdigheid geen sprake is.
* Concurrency treedt op in verschillende situaties:
  + Meerdere toepassingen: de verwerkingstijd (processortijd) dynamisch verdelen over een aantal actieve toepassingen
  + Gestructureerde toepassingen: sommige toepassingen kunnen geprogrammeerd worden als een verzameling van gelijktijdige processen
  + Structuur van besturingssysteem: besturingssystemen zelf worden vaak geïmplementeerd als een verzameling processen
* Concurrency bij systemen met 1 processor:
  + Toepassing boekt vooruitgang op meer dan één taak: (schijnbaar) gelijktijdig
  + Schakelen tussen verschillende taken tijdens uitvoering voor tegelijkertijd vooruitgang te boeken



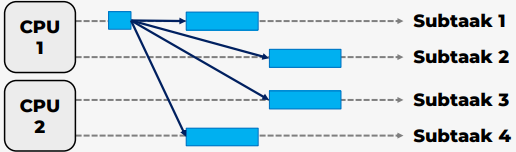
* Parallelle uitvoering (EN: Parallel Execution):
  + Systemen met meer dan één CPU of CPU-kern
  + Meer dan één taak van een toepassing worden gelijktijdig (parallel) uitgevoerd
  + ≠ parallelisme



* Parallelle gelijktijdige uitvoering (EN: parallel concurrent execution):
  + Taken worden verdeeld over meerdere CPU’s
  + Binnen één CPU: schakelen tussen verschillende taken – (schijnbaar) gelijktijdig
  + Taken op verschillende CPU’s worden parallel uitgevoerd



* Parallellisme (EN: parallelism):
  + Toepassing splitst zijn werk op in subtaken die parallel kunnen worden verwerkt
  + ≠ parallelle gelijktijdige uitvoering
  + Hoe parallellisme bereiken:
    - Meer dan 1 subtaak
    - Elke subtaak draait parallel op afzonderlijke CPU’s / CPUcores / GPU cores



## Wederzijdse uitsluiting (mutual exclusion)

* 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.
  + Een kritieke sectie: De code (instructies) die gebruikt wordt voor het aanspreken van gedeelde bronnen
  + Het is belangrijk dat er op elk moment maar maximum 1 proces in een kritieke sectie zit
  + Eisen 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.
  + VB:

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).

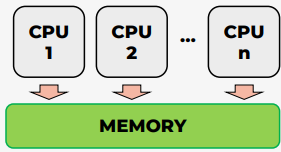
Als beide processen na elkaar de instructies uitvoeren is er geen probleem, en zal de waarde van getal met 2 verhoogd zijn. De volgorde van uitvoering is dan bijvoorbeeld: A1 – A2 – A3 – B1 – B2 – B3.

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

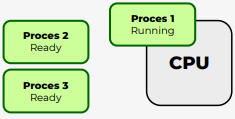
Merk op dat de eerste instructie hier niet opnieuw uitgevoerd wordt, waardoor proces A nog de ‘oude’ waarde kent van de variabele getal. Na uitvoeren van alle stappen zal getal hier dus niet verhoogd zijn met 2, maar met 1.

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

* Wederzijdse uitsluiting bij multiprocessing:
  + Als parallelle processen niets gemeenschappelijk gebruiken, is er geen probleem. De moeilijkheden ontstaan wanneer de processen het gemeenschappelijke geheugen aanspreken
  + Niet alleen processen, maar ook activiteiten binnen één proces kunnen parallel worden uitgevoerd. We zullen vooral processen bespreken, maar de principes gelden eveneens voor activiteiten binnen één proces



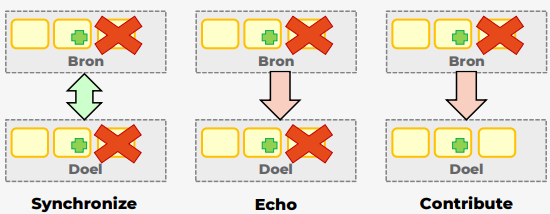
* Wederzijdse uitsluiting bij multiprogrammering:
  + Ook in een systeem met maar 1 CPU zijn gelijklopende processen mogelijk
  + Processen kunnen niet tegelijkertijd worden uitgevoerd, maar ze kunnen wel op hetzelfde moment proberen de besturing van de CPU te krijgen
  + Wanneer twee van zulke processen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



* Wederzijdse afsluiting afdwingen
  + Niet eenvoudig!
  + Mogelijke oplossingen:
    - Algoritme van Dekker (2 processen)
    - Wederzijds uitsluitingsalgoritme van Peterson (2 of meer processen)
  + Alternative oplossing:
    - Alternatieven: wederzijds uitsluiting afdwingen via 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.
    - Het gebruik van monitoren. Een monitor is een constructie in een programmeertaal die een functionaliteit biedt die vergelijkbaar is met die van semaforen, maar gemakkelijker te besturen is.
* Andere vormen van wederzijdse uitsluiting (hiervoor bespraken we alleen het gedeeld geheugen):
  + 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

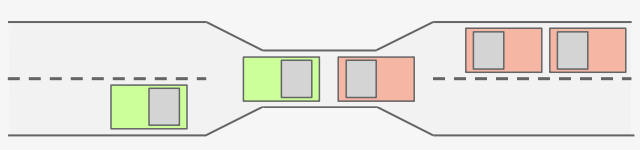
## Synchronisatie

* Synchronisatie:
  + Het proces, of het resultaat van iets gelijktijdig maken
  + Binnen de context van concurrency: 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.
* 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? 21We 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.
  + 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).
  + 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. Als 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, 23bijvoorbeeld wanneer een proces eeuwig moet wachten omdat er telkens een ander proces voorrang krijgt (zie ook vorig hoofdstuk).
  + Een eerlijke poging: Om ervoor 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 nieuw 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 vroeg of laat 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
* Bestandssynchronisatie:
  + Aan de hand van bepaalde regels, worden bestanden automatisch gesynchroniseerd zodat de inhoud van deze bestanden op twee of meerdere plaatsen hetzelfde zijn.
  + Is een voorbeeld van synchronisatie waarbij twee of meer bestanden aan elkaar gelijkgesteld worden.
  + Vroeger werd dit manueel gedaan (bv met usb-stick)
  + Nu verloopt bestandssynchronisatie automatisch via een netwerk. Je kan hier allerlei programma’s voor gebruiken zoals bv DropBox, OneDrive, …
  + Werking:
    - Selectie van mappen/bestanden
    - Vaak: bron ➔ doel
      * 1 bron (source, die de originele bestanden bevat)
      * 1 of meerdere doelen, waar je een kopie wil hebben van de bronbestanden (lokaal of remote)
    - Mogelijkheid om regels toe te voegen:
      * Bepaalde mappen uitsluiten?
      * Filter op type(s) bestanden?
      * Verborgen bestanden wel of niet synchroniseren?
    - Synchronisatie kan meteen (continu, bv bij Online samenwerken) gebeuren of periodiek (op vastgelegde tijdstippen, bv voor een back-up te maken)
  + Soorten:
    - 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.



## Deadlocks

* Deadlock (of impasse):
  + Een situatie waarbij een bepaalde actie is vastgelopen op wederzijdse uitsluiting.



* + Binnen de informatica: een deadlock (of impassetoestand) treedt op wanneer 2 of meer processen voor onbepaalde tijd wachten op een gebeurtenis die enkel door 1 van de wachtende processen kan worden veroorzaakt. Bv denk aan het filosofenprobleem
  + 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.
* 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.
  + Deadlocks voorkomen: het besturingssysteem kan het gemeenschappelijk gebruik van bronnen beperken met als doel om een deadlock-situatie zo goed als onmogelijk te maken.
  + Deadlocks vermijden: het besturingssysteem kan 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. Een mogelijk algoritme dat hiervoor gebruikt kan worden is Banker’s Algorithm.
  + Deadlock signaleren: 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.
  + Deadlock herstellen:
    - 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.
* Voorwaarden voor het ontstaan van een deadlock:
  + 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): een bron kan alleen vrijwillig 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 al in gebruik is door een ander proces in de verzameling.
* Deadlock preventie op basis van de voorwaarden:
  + Deadlock voorkomen: zorgen dat er minstens één voorwaard nooit optreedt?
  + Bron met wederzijdse uitsluiting is noodzakelijk, dus deze kunnen we niet opheffen
  + Proces zal bron bezet houden en ~~wachten~~ → vrijgeven indien nodig
    - 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
  + Er is geen voortijdig ontnemen → eerlijk? volgorde volgens schema
    - 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 ordeningsschema, en elk proces kan alleen bronnen in opklimmende volgorde verkrijgen
  + Processen wachten in een kring

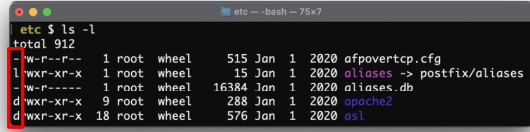
# File systems

## Persistente opslag

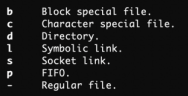
* Persistente opslag:
  + Data bewaren na afsluiten van het proces.
  + Grote hoeveelheden data opslaan.
  + Data delen tussen processen.
* Hard Disk Drive (HDD):
  + Is een magnetisch opslagmedium dat een grote opslagcapaciteit biedt voor een lage prijs.
  + Doordat een HDD werkt met draaiende schijven, is er telkens een kleine vertraging (seek time) wanneer de leeskop zich moet positioneren boven de gevraagde data.
* solid-state drive (SSD):
  + Opslag gebeurt in Integrated Circuits (IC’s)
  + Bevat geen bewegende onderdelen
  + Sneller dan HDD
  + Hogere kost per GB dan HDD
* CD/DVD:
  + Optisch opslagmedium.
  + Beperkte capaciteit en performantie.
  + Vaak read-only.
  + Voornamelijk geschikt voor distributie of backup
* USB-stick:
  + Extern opslagmedium, aangesloten via USB
  + Opslag gebeurt in ICs, net zoals SSD
  + Beperkte capaciteit en performantie

## Files

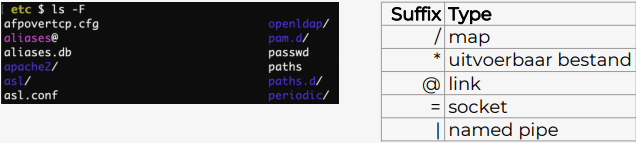
* File:
  + Een fysiek opslagmedium verdeelt zijn opslagruimte in blokken, vaak met een complexe layout. Een file (bestand) is een abstracte eenheid die data uit één of meerdere blokken groepeert en voorstelt als één geheel.
  + Het beheer van files en het toekennen van blokken behoort tot de taken van een file system (bestandssysteem). Op deze manier verbergt een file system de complexiteit van de fysieke opslag voor de gebruikers van het besturingssysteem.
* Voorstelling van een file:
  + Naar de gebruikers toe wordt een file voorgesteld als een opeenvolging van bytes.
  + Dit is echter een abstractie: in realiteit kunnen de bytes van een file verspreid zijn over verschillende blokken die niet altijd aan elkaar aansluiten.
* Eigenschappen van een file:
  + Naam, eventueel gevolgd door een extensie
  + Huidige en maximale grootte
  + Toegangsrechten
  + Datum van aanmaak
  + Datum van laatste aanpassing
  + Verwijzing naar de datablokken
  + …
  + Welke eigenschappen worden opgeslagen en hoe/waar deze worden opgeslagen hangen af van het gebruikte bestandssysteem.
* Soorten files:
  + Op UNIX (en afgeleiden zoals Linux) geldt de regel “everything is a file”
  + Gewone bestanden bevatten data
  + Directories (mappen) bevatten andere bestanden of mappen. Zij zorgen voor een hiërarchische structuur in het bestandssysteem
  + Links maken een bestand op meerdere plaatsen in het bestandssysteem zichtbaar
  + Block of character special files geven toegang tot hardware
  + Sockets zorgen voor netwerkcommunicatie
  + Pipes (FIFO) verbinden de output van een proces met de input van een ander proces
  + De output van ls –l toont het type van elk bestand:



* + De types zijn te vinden in de manpage van ls:



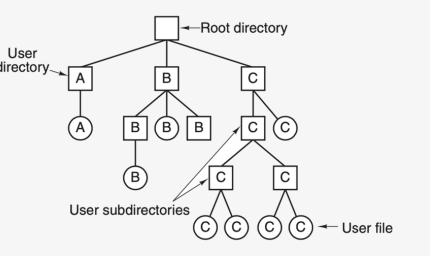
* + De output van ls –F toont het type via een suffix:



* Bytes uitlezen van een bestand kan op verschillende manieren:
  + Een bestand met sequentiële toegang dient in volgorde uitgelezen te worden.
  + Een bestand met willekeurige toegang (random access) kan in eender welke volgorde uitgelezen worden.

## Directories

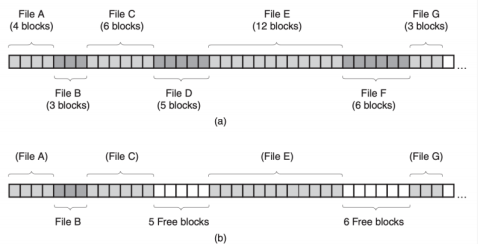
* Directory (map):
  + Groepeert files en kan zelf ook andere directories bevatten. Dit zorgt voor een hiërarchische structuur in het bestandssysteem.
  + De implementatie van directories is opnieuw de taak van een bestandssysteem.
    - Een mogelijke implementatie is bv om een directory op te slaan als een bestand met daarin de inhoudstafel van de directory.
* Hiërarchie:
  + Root directory = hoofdmap
  + Boomstructuur:



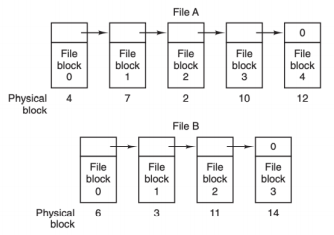
* Padnamen:
  + Het absoluut pad naar een bestand of map start bij de root directory en doorloopt de hiërarchie, bvb:
    - /home/hogent/os (Linux of Mac gebruiken “/”)
    - C:\Users\hogent\os (Windows gebruikt “\”)
    - Merk op dat een absoluut pad op Windows telkens start met een letter die gekoppeld is aan een bestandssysteem. Op Linux en Mac worden geen letters gebruikt
  + Een relatief pad start vanuit een bestaande directory en kan de speciale verwijzingen “.” (huidige directory) en “..” (parent directory) gebruiken:
    - ../hogent/os (Linux of Mac)
    - ..\hogent\os (Windows)

## File systems

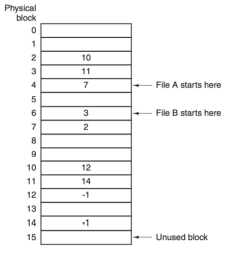
* File system:
  + Een file system (bestandssysteem) is een onderdeel van een besturingssysteem. File systems beheren de fysieke opslagruimte en wijzen deze toe aan files en directories. File systems implementeren dus files en directories.
  + De meeste besturingssystemen ondersteunen meerdere file systems. Er kunnen ook meerdere file systems tegelijkertijd actief zijn, bvb. één per opslagmedium dat aanwezig is in het systeem.
* Implementatie van files:
  + Contiguous storage (samenhangende opslag):
    - Elk bestand wordt in één of meer aansluitende blokken opgeslagen.
    - Voordelen:
      * Eenvoudig
      * Goede leessnelheid
      * Ondersteunt random access
    - Nadelen:
      * Als een bestand groeit, dan moet het mogelijk verplaatst worden naar een grotere vrije ruimte.
      * Als een bestand wordt verwijderd, en de vrije ruimte wordt ingenomen door een kleiner bestand, dan ontstaat er fragmentatie (= meer en meer kleine ruimtes die niet opgevuld geraken). Een systeem dat deze techniek gebruikt moet dus regelmatig gedefragmenteerd worden om de kleine vrije ruimtes terug samen te voegen.



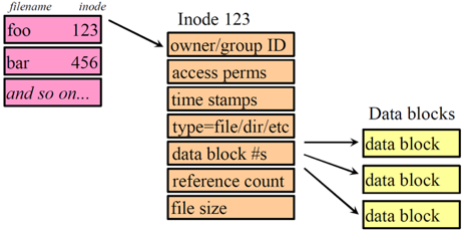
* + Linked lists (gelinkte lijsten):
    - De datablokken van een bestand hoeven niet aan te sluiten. Elk blok bevat namelijk een verwijzing (link) naar het volgende blok, dat zich eender waar mag bevinden.
    - Voordeel:
      * Geen fragmentatie
    - Nadelen:
      * Slechte performantie: Doordat de datablokken verspreid staan, daalt de leesperformantie, vooral bij opslagmedia met een seek time.
      * Geen random access, omdat de blokken in volgorde moeten worden uitgelezen.



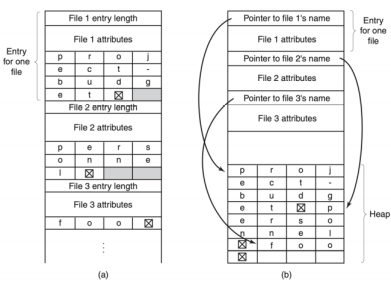
* + File allocation table (FAT) (toewijzingstabel):
    - De datablokken van elk bestand vormen nog steeds een linked list, maar alle verwijzingen tussen de blokken worden samengebracht in één tabel.
    - Voordelen:
      * Door deze tabel in het RAM te bewaren, is het mogelijk snel een lijst te vinden van alle blokken van een bestand, zonder de blokken zelf te moeten uitlezen. Dit verhoogt de leessnelheid en maakt random access mogelijk.
    - Nadelen:
      * Het nadeel van deze techniek is dat de FAT een hoog RAM-verbruik kan hebben.



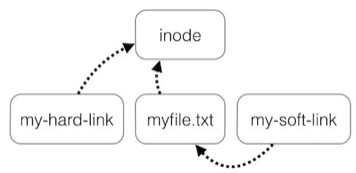
* + Index nodes (inodes):
    - Is een datastructuur die zowel de metadata van een bestand als verwijzingen naar de datablokken bevat. Een bestandssysteem dat inodes gebruikt, hoeft enkel de inodes van de geopende bestanden in het RAM te bewaren.
    - Voordelen:
      * Goede leessnelheid
      * Ondersteunt random access
      * Beperkt RAM-verbruik



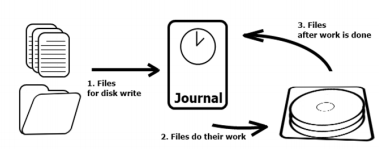
* Implementatie van directories:
  + Een directory kan opgeslagen worden in een bestand. Dit bestand bevat dan een directory entry voor elke file of subdirectory.
  + Elk file system kiest zelf welke informatie er wordt opgeslagen in een directory entry, maar deze informatie omvat zeker een verwijzing naar het eerste datablok of naar de inode (afhankelijk wat gebruikt wordt).
  + Een belangrijke keuze die moet worden gemaakt is hoe/waar de naam van elke file of subdirectory wordt opgeslagen:
    - In Figuur (a) wordt de naam opgeslagen als deel van de entry.
      * Heeft als nadeel dat directory entries een variabele grootte hebben, en er dus fragmentatie kan ontstaan binnen het directory-bestand.
    - In Figuur (b) worden de namen apart opgeslagen in een heap.
      * Heeft als nadeel dat er een heap moet beheerd worden.



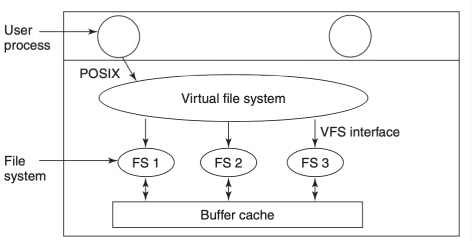
* Implementatie van links:
  + Een link is een bestand dat verwijst naar (of gekoppeld is aan) een ander bestand. Er bestaan 2 vormen:
    - Een hard link is een koppeling die wordt gecreëerd door dezelfde datablokken of inode in meerdere directory entries in te schrijven. Deze entries blijven onafhankelijk maar delen wel dezelfde data.
      * Minder flexibel omdat deze link enkel mogelijk is binnen hetzelfde bestandssysteem.
    - Een soft link (ook gekend als symbolic link of snelkoppeling) heeft een eigen datablok of inode. Dit soort link is een bestand waarvan de inhoud bestaat uit een verwijzing naar een ander bestand.
      * Is flexibeler omdat ze ook werkt tussen verschillende bestandssystemen
      * Kan leiden tot een dangling pointer: een soft link die verwijst naar een bestand dat niet meer bestaat



* Journaling:
  + Een crash tijdens een schrijfbewerking kan leiden tot corrupte of verloren data
  + Het bijhouden van een journal (logboek van bewerkingen) kan hiertegen beschermen
* Bestandssyteem met journaling werkt als volgt:
  1. Elke uit te voeren bewerking wordt eerst neergeschreven in het logboek.
  2. Vervolgens wordt de bewerking zelf uitgevoerd.
  3. Tenslotte wordt in het logboek de bewerking gemarkeerd als voltooid.
  + Wanneer het bestandssysteem of besturingssysteem crasht tijdens de uitvoering van de bewerking, dan kan het bestandssysteem deze crash detecteren omdat er een nog-niet-voltooide bewerking in het logboek staat. Vervolgens kan het bestandssysteem de onvoltooide bewerking alsnog voltooien op basis van de informatie in het logboek.



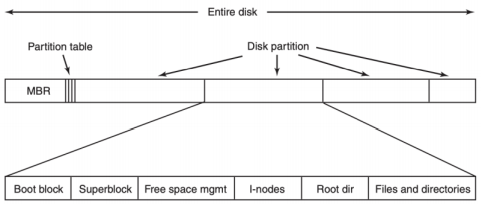
* Vritual file systems:
  + Op Windows krijgt elk bestandssysteem een drive letter toegewezen. Zo spreek je bv over het bestand C:\Users\hogent\file.txt, dat zich bevindt op “de C schijf” (het bestandssysteem van deze schijf).
  + Op Linux en Mac daarentegen is er een virtual file system. Deze systemen brengen alle actieve bestandssystemen samen onder één hiërarchische structuur. Voor de gebruiker lijkt het alsof er slechts één bestandssysteem is



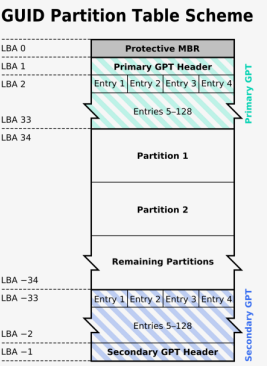
* + Processen gebruiken de algemene functies van het besturingssysteem om het virtual file system aan te spreken. Deze functies (op de figuur aangeduid als POSIX) zijn dus niet gebonden aan een specifiek file system.
  + Het virtual file systeem vertaalt deze algemene functieoproepen naar oproepen voor de drivers van elk file system.
  + Een virtual file system verbergt dus de verschillen tussen de file systems. Processen zien slechts één bestandsstructuur, die op een uniforme manier kan worden bewerkt.
  + Twee belangrijke bewerkingen bij een virtual file system zijn:
    - mount: het inladen van de root directory van een bestandssysteem in een directory van de virtuele hiërarchie. Zo kan je bv kiezen om de root directory van een USB-stick in te laden in de map /media/usb. Vanaf dan verwijst deze map naar het bestandssysteem van de USB-stick.
    - unmount: de omgekeerde bewerking. Een reeds ingeladen bestandssysteem terug loskoppelen van het virtuele bestandssysteem

## Partities

* Partities:
  + Een fysiek opslagmedium (zoals een harde schijf) kan zijn capaciteit onderverdelen in partities. Elke partitie heeft dan een eigen bestandssysteem.
  + Een opslagmedium met partities moet ook een partitietabel voorzien. Deze tabel beschrijft de partities en hun bestandssystemen. Twee systemen die hiervoor worden gebruikt zijn een Master Boot Record (MBR) of een GUID Partition Table (GPT).
* MBR vs GBT:
  + MBR: 1983 (DOS 2.0), belangrijke beperkingen:
    - Maximum grootte schijf: 2TB
    - Maximum 4 primaire partities
      * Kan omzeild worden door een extended partitie te maken die de logische partities bevat
  + GPT: opvolger voor MBR
    - Biedt ondersteuning voor schijven groter dan 2TB, en binnen GPT kan je in theorie een bijna onbeperkt aantal partities hebben (maar in Windows is het aantal partities wel beperkt tot 128).
    - Elke partitie krijgt een unieke ID, de Globally Unique Identifier (GUID), die opgeslagen wordt in de GUID partition table (GPT).
    - 44GPT enkel werkt op moderne systemen, die 64-bit zijn en gebruikmaken van UEFI (opvolger van BIOS) als interface tussen de hardware en het besturingssysteem.
* MBR:
  + Een Master Boot Record bevat naast de partitietabel ook een zogenaamde boot loader, die verantwoordelijk is voor het opstarten van het systeem. Deze boot loader gaat op zoek naar een partitie met een besturingssysteem en schakelt dan door naar de code in het boot block van deze partitie.



* GPT:
  + GPT is achterwaarts compatibel met MBR en kan ook worden gebruikt in combinatie met een BIOS. UEFI is dus niet verplicht.
  + GPT plaatst de boot loader op een speciale EFI System Partition. Uiteraard kan er via de achterwaartse compatibiliteit met MBR nog steeds gebruik worden gemaakt van boot blocks.



## Voorbeelden

* NTFS (New Technology File System):
  + Is het standaard file system op Windows. Dit file system gebruikt journaling en vervangt het oudere FAT32 file system.
  + NTFS is een propiëtair file system van Microsoft, met beperkte ondersteuning op Mac en Windows. Op deze platformen gebruik je best de open source NTFS-3G driver, die het mogelijk maakt om NTFS-partities te lezen en te schrijven.
* FAT32:
  + Is een bestandssysteem dat gebruikmaakt van een 32 bits File Allocation Table.
  + FAT32 is ontwikkeld door Microsoft als opvolger voor FAT12 en FAT16. Het nummer verwijst naar het aantal bits dat gebruikt wordt voor de adressering: bij FAT32 worden 32 bits gebruikt om de clusters te adresseren.
  + Enkele beperkingen:
    - De maximum grootte van het bestandssysteem is 2TB
    - De maximum grootte voor een bestand is 4GB
  + Tegenwoordig nog weinig gebruikt.
* exFAT (Extensible File Allocation Table):
  + Is een propiëtair file system van Microsoft. Zoals je uit de naam kan afleiden, gebruikt dit file system een file allocation table.
  + In tegenstelling tot NTFS gebruikt exFAT geen journaling, dit om schijfruimte te besparen. exFAT is namelijk gericht op verwijderbare media zoals USB-sticks en SD-kaarten.
  + exFAT wordt goed ondersteund op Mac en Linux, wat het uitermate geschikt maakt voor bestandsuitwisseling.
* HFS+ (Hierarchical File System Plus, of MacOS Extended):
  + Was het standaard file system op Mac tot enkele jaren terug
    - Beperkte ondersteuning op Windows en Linux
  + Gebruikt journaling
  + Vervangen door APFS en niet langer aan te rade
* APFS (Apple File System)
  + Is het nieuwste file system voor macOS, iOS, en aanverwanten.
  + Dit file system heeft een grote focus op SSDs en encryptie, en gebruikt GPT-containers en volumes i.p.v. klassieke partities.
  + APFS is hoofdzakelijk een Apple-product en heeft slechts een beperkte ondersteuning op Windows en Linux, via third-party (commerciële) drivers.
* ext4 (fourth extended file system):
  + Is het standaard file system op vele Linux-distributies.
  + Dit file system gebruikt journaling en is achterwaarts compatibel met zijn voorgangers ext2 en ext3.
  + ext4 wordt bijna uitsluitend gebruikt op Linux en heeft slechts beperkte ondersteuning op Windows en Mac, via third-party (commerciële) drivers.
* ZFS (Zettabyte File System):
  + Is een file system oorspronkelijk ontwikkeld door Sun Microsystems voor hun Solaris besturingssysteem.
  + ZFS is uitermate krachtig en complex (volume management, snapshots maken, klonen, integriteitscontrole, caching, ...), en voornamelijk gericht op intensieve servertoepassingen. Het is daarom erg populair op Linux en FreeBSD.
* Btrfs (B-tree File System of Butter FS of Better File System):
  + Is een file system ontwikkeld door Oracle, als antwoord op ZFS.
  + Net als ZFS is Btrfs een geavanceerd file system, met veel meer features dan ext4.
  + Btrfs is sinds kort ook het standaard file system op Fedora.
* ISO 9660 en UDF (Universal Disk Format):
  + Zijn file systems voor optische schijven (resp. CD en DVD).
  + Deze file systems verschillen heel erg van de vorige omdat ze gericht zijn op read-only of write-once media voor distributie of backup.
  + Beide file systems zijn uiteraard ondersteund op alle gangbare platformen.

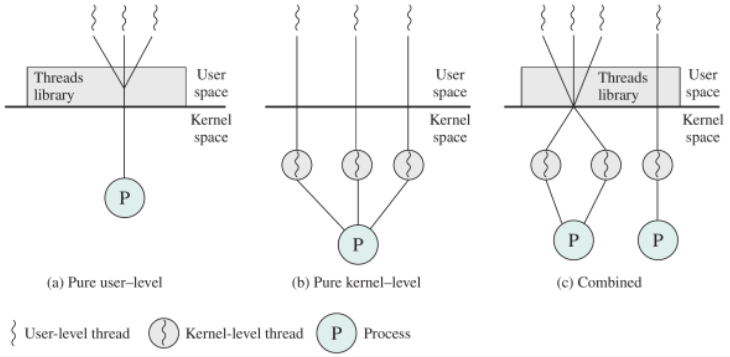
# Threads

## Wat zijn threads?

* Processen bestaan uit 2 concepten:
  + 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.
* Processen vs threads:
  + 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
  + Single-threaded proces: Wanneer een proces bestaat uit één enkele thread voor de uitvoering.
  + Multi-threaded proces: Wanneer een proces echter zijn werk verdeeld over meerdere threads. Deze threads kunnen parallel uitgevoerd worden door het systeem, wat vooral interessant is voor systemen met meerdere processoren en/of multi-core CPU’s.
* Opbouw threads:
  + Elke thread van een proces heeft toegang tot alle bronnen toegewezen aan dat proces.
  + Daarnaast heeft elk thread ook zijn eigen Thread ID, 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.
  + 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.
  + 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).

## Soorten threads

* User-Level Threads:
  + Het programma maakt 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.
  + 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, bv wachtend op I/O, wordt mogelijks het hele proces geblokkeerd
    - Geen gebruik van multiprocessing: threads die behoren tot hetzelfde proces kunnen dus niet parallel uitgevoerd worden.
* Kernel-level threads:
  + De logica voor threads zit in het besturingssysteem, het besturingssysteem is met andere woorden verantwoordelijk voor het aanmaken en beheren van de verschillende threads.
  + Voordelen:
    - Scheduling mogelijk op threadniveau
    - Blokkeren van één thread geen invloed op andere threads process
    - Multiprocessing mogelijk: threads kunnen parallel uitgevoerd worden op verschillende processoren of processorkernen.
    - Besturingssysteem gebruikt waarschijnlijk zelf threads
  + Nadelen:
    - Trager (wisselen tussen programma en besturingssysteem, terwijl bij user-level threads het aanmaken en beheren van threads volledig gebeurt binnen het programma zelf)
* Combinatie ULT en KLT:
  + Het is mogelijk om beide vormen te combineren, en zo het beste over te houden van beide vormen.
  + Kan op verschillende manieren

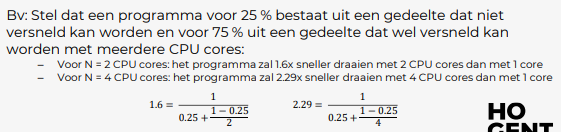


## Voor- en nadelen

* Voordelen multi-threading:
  + Een interactief proces moet soms een operatie uitvoeren die lang kan duren. 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.
  + Threads binnen een proces daarentegen kunnen zonder afscherming aan elkaar. Het is dus 14eenvoudiger en efficiënter om threads binnen een proces met elkaar te laten communiceren dan processen onderling.
  + Lichter en sneller dan processen (creatie, context switch, afsluiten, ...)
  + Een multi-threaded proces kan efficiënt gebruik maken van een multicore CPU’s (parallelisme).
    - 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.
* Uitdagingen multi-threading:
  + Indelen van het rekenwerk in mogelijke threads: Rekenwerk dat zich kan opsplitsen in onafhankelijke problemen is ideaal voor het opsplitsen in threads, 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. Dit maakt het enorm moeilijk om bugs te vinden, te testen en op te lossen.

## Parallellisme

* Soorten parallellisme:
  + Data parallellisme: Hierbij 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.
  + Taken parallellisme: 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 17een thread voor de GUI, een thread voor spellingscontrole, een thread om het bestand weg te schrijven, …
  + In de praktijk wordt vaak een hybride combinatie gebruikt van de twee.
* Amdahl’s law:
  + Geeft de theoretische snelheidswinst bij het toevoegen van CPU cores
  + 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.
  + Bij een single-threaded proces is er geen snelheidswinst te behalen (S = 1)
  + Bij een multi-threaded programma kan er vaak wel een snelheidswinst behaald worden.
  + Vb:



* + Wat als N = ∞ (oneindig)?
    - 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. Dit is 4x sneller (25 %), en 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.
  + Hoe groter S, hoe lager de 19maximale snelheidswinst zal zijn ongeacht het aantal CPU cores.

## Voorbeelden

* 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), ..
* 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
  + …
* 3D modelling:
  + Thread voor UI
  + Werk voor rendering wordt verdeeld over meerdere threads voor parallellisme
    - Kan op CPU of op GPU
* Video conversies:
  + Werk voor het converteren wordt verdeeld over meerdere threads voor parallellisme