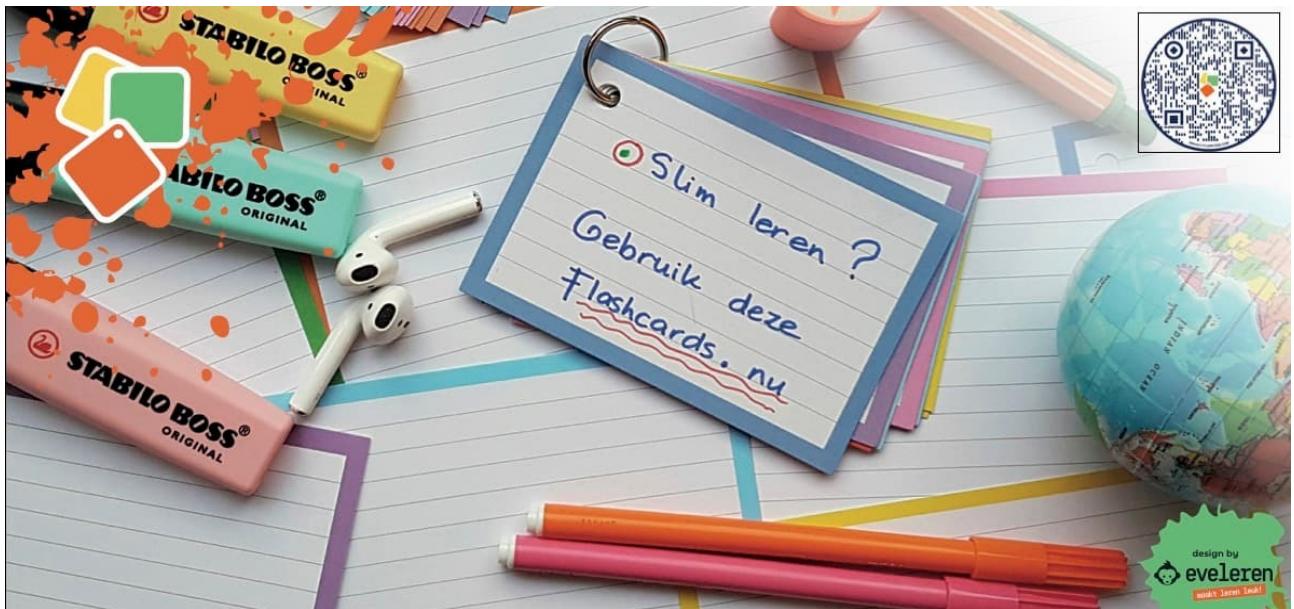


Besturingssystemen - Theorie

geschreven door

bert16



www.stuvia.be

Besturingssystemen Theorie

Hoofdstuk 1 Inleiding Linux

1. Wat is Linux ?
 - **Multi-usersysteem**
 - **Overal** te vinden
 - Google, YouTube, Amazon, IBM, Novell, Wikipedia, Facebook, Twitter, LinkedIn, Politie, Oracle, SAP, Nasa, Euronext,...
 - Ontwikkeld vanuit **UNIX**
 - Ontwikkeld door **Linus Torvalds**
 - Studeerde computerwetenschappen
 - Wilde graag een gratis beschikbaar, academische versie van UNIX maken
 - Verschillende **distributies** (wij kiezen **Fedora**)
 - Linux is nu een **volledige UNIX kloon**, geschikt voor werkposten, **mid-range en high-end servers**
 - In het begin: concentratie op netwerkaspect: netwerkkaarten aan de praat krijgen, diensten zoals mail en web aanbieden. Bureauautoepassingen zoals een grafische tekstverwerker en een rekenblad vormden één van de laatste barrières om te overwinnen
 - Vandaag is Linux **klaar voor de dekstopmarkt**
 - RedHat, SuSe, Mandriva en Debian begonnen hun versie van Linux samen te stellen
 - Specifieke set van pakketten en tools werd geschikt gemaakt voor massaconsumptie
 - GUI
 - Eerste distributies
- Terminal
 - Biedt een **tekstinterface** aan
 - Voordelen:
 - **Snelste manier** om met Linux te werken
 - Via de terminal kan je **ook op andere computers op het netwerk**. Dat kan natuurlijk ook met andere programma's maar die bieden telkens maar een deel van de functionaliteit aan.
 - Laat ons toe verder te zien dan de grafische verpakking van een systeem.
 - Verschillende manieren om een terminal af te sluiten
 - File → Close Window
 - Kruisje

- CTRL + D
- Commando exit
- Shift + CTRL + Q
- Basiscommando's
 - comando [-opties] [argument(en)]
 - Bv. ls -l
 - commando's zijn dezelfde voor alle linux distributies
 - Prompt: hierin geef je de commando's in **user@hostname:~\$**
 - Voor de apenstaart: de gebruikersnaam waarmee je op het systeem aan het werken bent.
 - Na de apenstaart: de computernaam van het systeem
 - Na het dubbele punt: de map waarin je je bevindt
 - Commando met opties: de optie bepaalt hoe het commando zich moet gedragen
 - Optie wordt voorafgegaan door een streepje (-): **ls -a**
 - **ls** commando geeft de inhoud van een map weer. De -a optie specificeert dat alle bestanden, ook diegene die verborgen zijn, getoond moeten worden
 - Na het streepje kunnen meerdere opties gecombineerd worden bv **ls -al == ls -l -a**
 - -l zorgt ervoor dat we ook alle eigenschappen van de volledige lijst willen zien
 - Commando's met argumenten: dit zijn objecten waarop het commando zal inwerken bv ls /etc
 - Commando's kunnen gevuld worden door zowel opties als argumenten bv. **ls -al /etc**
 - Geeft een overzicht van de inhoud van de configuratiemap /etc
- Het Linux bestandssysteem
 - Heel eenvoudig kan men het volgende stellen over Linux: "On a Linux system, **everything is a file**. If something is not a file, it is a process."
 - Met cd veranderen we van directory

- Oriëntatie in het bestandssysteem
 - Het pad zorgt ervoor dat we niet steeds het volledige (absolute) pad moeten ingeven om een commando of om een bestand te openen.
 - **Path:** Environment variable - bevat één of meerdere map verwijzingen. Deze worden onderzocht om een commando te lokaliseren en uit te voeren.
 - **Absoluut pad:** start steeds vanaf de root en begint steeds met een **slash** (/)
 - **Relatief pad:** **start vanaf de huidige map** en begint nooit met een slash
 - In relatieve paden kan ook gewerkt worden met . (dot) → verwijst naar de huidige map en .. (dot dot) → verwijst naar hoger liggende map
 - Inhoud van PATH bekijken dmv **echo** command: **echo** toont de inhoud (\$) van de variabel PATH
 - De inhoud van de PATH variabele kan gewijzigd worden met het **export** commando: **export PATH=/usr/bin:/usr/bin**
 - Deze wijzigingen zijn tijdelijk, enkel voor deze sessie
 - Bv. **export PATH=\$PATH:~/Muziek**
- Mappen in het bestandssysteem
 - Aanmaken van mappen: **mkdir** mapnaam (meerdere namen mag)
 - **mkdir** staat voor *make directory*
 - Vb: **mkdir -p dir/dir1**
 - Verwijderen van mappen: **rmdir** mapnaam (meerdere namen mag)
 - **rmdir** staat voor *Remove directory*
- Inhoud van tekstbestanden
 - Commando **cat** (concatenate): laat de inhoud van een tekstbestand over het scherm lopen
 - Commando **less**: laat je toe om de tekst, scherm per scherm te bekijken
 - Druk op de **spatiebalk** om de **volgende pagina** te zien te krijgen
 - Druk op **B** om **terug te gaan**
 - Druk op **Q** om het less programma te **verlaten**
 - Je kan ook de pijltjes omhoog en omlaag gebruiken om te navigeren
 - Commando **tac**: Toont tekst in omgekeerde volgorde (regels)
 - Commando **head**: Toont de 10 eerste lijnen van een bestand
 - Commando **tail**: Toont de laatste 10 lijnen van een bestand - interessant om de inhoud van logfiles te bekijken (laatste gebeurtenissen)
 - Commando **more**: Voorganger van less (more is less than less)



- Tekst zoeken in bestanden doe je met het commando **grep** [opties] tekenreeks [bestanden]
 - Enkele opties:
 - **-i** --ignore-case
 - **-n** --line-number
 - **-s** --silent: onderdrukt alle foutmeldingen
 - Het programma grep is eenvoudig maar zeer krachtig voor het filteren van lijnen output die een bepaalde string (=zoekwoord) bevatten. Er zijn letterlijk duizenden toepassingen die van het grep commando gebruik maken.
- Shell
 - Een **shell** is een programma dat wordt opgestart in een terminal venster. Het is de interface naar het hart van je computer
 - **Verschillende shelles:** sh -bash -csh -tcsh -ksh
 - Overzichten terug te vinden in **/etc/shell**
 - Welke shell ? **echo \$SHELL**
 - Snel werken met de shell:
 - **CTRL+A**: Plaatst de cursor aan het begin van de lijn, vlak achter de prompt
 - **CTRL+C**: Beëindigt een lopend programma en geeft je de prompt terug zodat je een nieuw commando kan starten
 - **CTRL+D**: Verlaat de huidige shell sessie, dit staat gelijk met exit of logout
 - **CTRL+E**: Plaatst de cursor aan het einde van de lijn
 - **CTRL+H**: Genereert een backspace, verwijdert het karakter links van de cursor
 - **CTRL+L**: Maakt de terminal leeg, zodat je prompt bovenaan komt te staan
 - **CTRL+R**: Zoek in de commandogeschiedenis
 - **CTRL+Z**: Bevriest een programma
 - **Pijltje links en rechts**: Beweeg de cursor over en weer op de commandolijn
 - **Pijltje omhoog en omlaag**: Overloopt de commandogeschiedenis. Ga naar de lijn die je opnieuw wil uitvoeren, editeer eventueel en druk Enter
 - **Shift+PageUp en Shift+PageDown**: Overloop de terminalbuffer om tekst te zien die al van het scherm gerold is
 - **Tab**: Commando- of bestandsnaam vervolledigen. Als er meerdere mogelijkheden zijn, zal de shell je met een geluidje of flits waarschuwen
 - **Tab Tab**: Toont de mogelijke bestandsnamen of commandonamen om te vervolledigen

- Werken met bestanden
 - Tekstbestanden aanmaken: **touch** bestand1 bestand2 ...
 - Bestanden manipuleren:
 - **Kopiëren:** **cp** bronmap/bronbestand doelmap/doelbestand
 - Geen mappen gespecificeerd: naar huidige map
 - Enkel doelmap en geen doelbestand: bestand behoudt zelfde naam
 - Meerdere bestanden kopiëren: doel moet enkel een map zijn
 - Gebruik van wildcards is toegestaan
 - Opties:
 - **-r** = kopieert mappen, inclusief hun inhoud
 - **-v** = verbose, toont alle kopieacties op het scherm
 - **Verplaatsen:** **mv** map/bestand
 - Opties:
 - **-r** = recursive, verwijder map, inclusief bestanden
 - **-f** = force, geen waarschuwing
 - **-v** = verbose
 - Let op met rm -rf als root op / → Alles kwijt !
 - **Zoeken:**
 - Op **bestandsnamen** - eenvoudig
 - Commando **locate** (lokaliseer) - maakt gebruik van een 's nachts opgebouwde database bv. Locate Bureaublad
 - Op **bestandsnamen** - geavanceerd
 - **find** zoekmap zoekopties [tests] [acties]
 - Zoekopties: zie man pages
 - Naar **commando's**
 - **which** commandonaam
 - Which zoekt in de mappen gedeclareerd in de PATH variable
 - Vb. which ls
 - Hulp
 - Gebruik maken van **man** pages met het commando **man**
 - **whatis** geeft een korte index van verklaringen voor een commando
 - **apropos** niet enkel op commando gezocht maar er wordt ook gekeken naar de eigenlijke beschrijving
 - **info** heeft gewoonlijk meer recente informatie en sommigen vinden ze gemakkelijker te gebruiken. De man pages zijn geïntegreerd in de info pagina's: als er geen info pagina beschikbaar is voor een bepaald commando of bestand, zal info gewoon de man pagina voor dat commando of bestand tonen.

Hoofdstuk 2 Inleiding besturingssystemen

1. Wat is een besturingssysteem ?

Definitie: Een besturingssysteem is een programma dat het mogelijk maakt de hardware van een computer te gebruiken.

- In hoofdzaak is een besturingssysteem een **programma dat mensen de mogelijkheid geeft gebruik te maken van de hardware van een computer** (CPU's, geheugen en secundaire opslagapparatuur).
- **Gebruikers geven geen instructies aan de computer maar in plaats daarvan aan het besturingssysteem.** Het besturingssysteem geeft de hardware de opdracht de gewenste taken uit te voeren.
- De Engelse term voor besturingssysteem is "**Operating System**", de afkorting **OS** wordt vaak gebruikt.
- **Traditioneel** denken veel gebruikers dat **Windows** het **enige besturingssysteem** is, of dat er slechts **één alternatief** is genaamd **OSX**. Er zijn er echter veel meer. Onder IT'ers is het **Linux** besturingssysteem wijd verspreid, een OS dat we verder in deze cursus zullen behandelen. De parallelen met één van de oudste besturingssystemen, **Unix**, zijn groot. O.a. het bedrijf Oracle verkoopt nog steeds Unix.
- Vergeet niet dat er ook heel wat besturingssystemen bestaan hebben die nu niet meer bijgewerkt worden. **Rekenmachine, klassieke GSM** toestellen,... hebben ook allemaal hun **eigen besturingssysteem, ontworpen voor de specifieke hardware** van die toestellen.

Functies:

- **Opslaan** en ophalen van **informatie**
- **Programma's afschermen** van **specifieke hardwarezaken**
- **Gegevensstroom** door de **componenten** van de computer **regelen**
- **Programma's** in staat stellen te **werken zonder** door andere programma's **onderbroken te worden**.
- Het mogelijk maken om **bronnen te delen**
- **Tijdelijke samenwerking** tussen **onafhankelijke programma's** mogelijk maken
- **Reageren** op **fouten** van de **gebruiker**
- **Tijdsplanning maken** voor **programma's** die **resources** willen gebruiken

2. Kort historisch overzicht

- Besturingssysteem → Dynamisch iets.
- Veel geavanceerde systemen van tegenwoordig hebben nauwelijks punten van overeenkomst met de eerste ontwerpen. De eerste elektronische computer had geen besturingssysteem. Hij moest met de hand bediend worden.
- In de **jaren 50** werden er eenvoudige besturingssystemen ontwikkeld waarmee **programma's na elkaar** in de computer konden worden **ingevoerd** en **opgeslagen**. Wanneer het ene programma klaar was, laadde en startte het besturingssysteem het volgende programma. **Alle resources** konden slechts **door één programma** gebruikt worden
- In de **jaren 60** konden **verscheidene programma's tegelijkertijd in het geheugen** worden **opgeslagen**. De programma's gebruikten de resources van de computer dan gemeenschappelijk. De programma's werden nu niet na elkaar, maar **om beurten** uitgevoerd. Elk programma liep een tijdje en dan schakelde het besturingssysteem over naar een ander programma. Hierdoor kwamen de **korte programma's eerder aan de beurt**, en omdat de resources gemeenschappelijk werden gebruikt, waren ze **sneller afgelopen**. Een gebruiker kon vanaf een terminal inloggen en vrijwel direct toegang tot de resources krijgen
- In het **midden van de jaren 60** kon **één besturingssysteem voor verschillende computers van hetzelfde type worden gebruikt**, zodat een upgrade eenvoudiger werd.
- In het **begin van de jaren 70** kunnen besturingssystemen **computers met meer dan één processor** aan
- In het **begin van de jaren 80** zijn besturingssystemen geschikt voor **gemeenschappelijk gebruik van informatie door verschillende computersystemen**
- Besturingssystemen zorgen ervoor dat de **hardware** van de computer **bruikbaar wordt**. Ook al is de **gebruikersinterface** (user interface) van **uitzonderlijk belang**, er is nog meer. Een besturingssysteem **bepaalt waartoe de gebruiker in staat is en hoe efficiënt** hij of zij dit kan. Het bepaalt **welke hardware** op een computer kan worden **aangesloten**. Verder bepaalt het **welke programma's** de computer **accepteert**. Niet enkel professionele informatici, maar ook mensen die maar af en toe achter een computer zitten, weten waarschijnlijk meer over een besturingssysteem dan over de computer zelf.

Single-tasking - Multitasking - Multi-user systemen

Single tasking: Een systeem waarin één gebruiker één applicatie tegelijk draait, heet een single-tasking systeem. Onder dit systeem kan slechts één programma (task) tegelijk actief zijn.

Multitasking: Meestal 1 gebruiker die verscheidene taken kan uitvoeren tezelfdertijd.

- Aangezien een gebruiker verscheidene werkzaamheden gelijktijdig kan laten verrichten, worden bepaalde functies van het besturingssysteem, bijvoorbeeld geheugenbeheer, ingewikkelder.

Multi-user-systemen: Meerdere gebruikers maken simultaan gebruik van de computerresources.

- Multiuser-systemen, ook wel **multiprogrammering-systeem** genoemd, moeten niet alleen alle gebruikers bijhouden, er moet ook voorkomen worden dat deze elkaar hinderen of in het werk van de ander rondneuzen
- In een multiuser-systeem wordt **scheduling** belangrijker. Bij een single-user computer hoeft het besturingssysteem slechts de behoeften van één gebruiker te vervullen. In een multi-user computer gaat het om de **behoeften van velen**. Dit kan moeilijk of zelfs onmogelijk zijn. Aangezien **veel programma's** de **resources** van de computer **gemeenschappelijk** moeten **gebruiken**, moet het besturingssysteem beslissen wie wat krijgt en wanneer. Vaak zal het aan elk programma de hoogste prioriteit toekennen. Maar hoe beslist het besturingssysteem welke prioriteit een bepaald programma krijgt? Elke gebruiker denkt dat **zijn** of haar programma de hoogste prioriteit zou moeten hebben en de programma's van alle andere een lagere. Hoe lost het besturingssysteem dit probleem op? Hoe stelt het vast welke programma's werkelijk een hoge prioriteit verdienen?
- Er bestaan verscheidene soorten multiuser-computers, afhankelijk van de soorten programma's die ze aankunnen.
 - **Interactieve programma's:** Programma dat een gebruiker vanaf de terminal activeert. Over het algemeen voert de gebruiker een korte opdracht in. Het besturingssysteem vertaalt deze opdracht en onderneemt actie. Het zet vervolgens een prompt-teken op het scherm en geeft daarmee aan dat de gebruiker een volgende opdracht kan invoeren. De gebruiker voert weer een opdracht in en het proces gaat door. De gebruiker werkt met het besturingssysteem op een conversatie-achtige manier, interactieve mode genoemd. Interactieve gebruikers verwachten **snelle respons**. Daarom moet het besturingssysteem interactieve gebruikers **voorrang** geven.
 - **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. Uiteindelijk zal het besturingssysteem de opdrachten uitvoeren. Batch-gebruikers verschillen van interactieve gebruikers omdat zij **geen directe respons** verwachten. Bij scheduling houdt het besturingssysteem hier rekening mee.

- **Real-time programma's:** Real-time programmering legt aan de respons een **tijdsbeperking** op. Het wordt gebruikt wanneer een snelle respons essentieel is. Interactieve gebruikers geven de voorkeur aan **snelle respons**, maar real-time gebruikers **eisen** dit zelfs. Voorbeelden van real-time verwerking: controlesysteem voor het luchtverkeer op een vliegveld, robots,...

Virtuele machines

- **Definitie** virtuele machine: Computerprogramma dat een *volledige* computer nabootst, waar andere (besturings)programma's op kunnen worden uitgevoerd
- **Soorten:**
 - **Programmeertaal-specifiek:** vb. **JVM**
 - **Emulator:** vb. **VirtualBox, VMWare, Parallels**
 - **Applicatie-specifiek:** vb. **Docker**

Virtual Machine Monitor - Hypervisor

- Een Hypervisor of Virtual Machine Monitor is in de informatica een opstelling die ertoe dient om meerdere besturingssystemen tegelijkertijd op een hostcomputer te laten draaien. Met de term hypervisor wordt de eerdere term supervisor uitgebreid, die gewoonlijk toegepast werd op besturingssysteemkernels. Een hypervisor regelt een vorm van virtualisatie. Hypervisors zijn in twee soorten ingedeeld
 - Native or Bare-Metal (type 1)
 - Een Type1 Hypervisor ligt direct op de hardware, dat wil zeggen dat er geen Besturingssysteem tussen ligt, hierdoor kunnen er meer resources aan de virtuele machines gegeven worden. Een van de nadelen van een Type1 is dat je enige technische kennis moet hebben om ermee te kunnen werken. Type1 Hypervisors worden dan vooral ook gebruikt voor en door Servers.
 - Voorbeelden: VMWare ESXi, Citrix Xen, KVM en Microsoft Hyper-V
 - Hosted (Type2)
 - Een Type2 Hypervisor ligt in tegenstelling tot een Type1 niet direct tegen de hardware aan, dit wil zeggen dat er wel een besturingssysteem onder ligt. Dit kunnen er verschillende zijn zoals: Microsoft Windows, Apple macOS en Linux. Voordelen die je hebt bij Type2 is dat hij aan de praat te krijgen is zonder al te veel technische kennis, meestal kunnen Type2 Hypervisors geïnstalleerd worden zoals een programma. Een nadeel is dat Type2 niet zo krachtig en efficiënt is als Type1. Enkele voorbeelden van Type2 Hypervisors zijn: Oracle VirtualBox, VMWare Workstation en Parallels Desktop

Hoofdstuk 3 Scheduling

Noodzaak tot scheduling

- Efficiënt middelen (bronnen, resources) inzetten om taken (opdrachten, jobs uit te voeren)
- Bij **multiprogrammering** moet het besturingssysteem aan de verschillende behoeften van vele gebruikers voldoen. Het moet ook efficiënt zijn. Processen moeten resources benaderen en binnen een redelijke tijd moeten de processen worden uitgevoerd. De kans hierop is groter als de machinecode voor een proces, of tenminste een deel daarvan, in het geheugen staat. Multiprogrammering is de mogelijkheid de code van meerdere processen gelijktijdig in het geheugen te bewaren.
- We moeten echter benadrukken dat **multiprogrammering niet hoeft te betekenen dat er veel processen gelijktijdig worden uitgevoerd**. Dit is iets dat vaak verkeerd wordt begrepen. Het geheugen kan van vele processen tegelijk de code bevatten, maar als er maar één CPU is, kan er maar één proces tegelijk worden uitgevoerd. Multiprogrammering is iets anders dan multiprocesssing.
- **Multiprocessing** betekent dat het systeem meerdere processors bevat. Elke CPU kan de code van een afzonderlijk proces uitvoeren. Multi-programmering in een systeem met één CPU levert vele interessante problemen op. De verschillende processen willen allemaal toegang tot de CPU.
- De processor (CPU) moet regelmatig lange tijd wachten op in- en uitvoer van een bepaald proces. Tijdens deze wachttijd kan dan een deel van een *ander* proces uitgevoerd worden. Om te bepalen welk proces op welk moment mag uitgevoerd worden, wordt een **scheduler** gebruikt. De scheduler moet de processorbelasting balanceren en voorkomen dat één proces alle CPU-tijd gebruikt, of juist geen CPU-tijd krijgt. In realtime omgevingen, zoals industriële robots, zorgt de scheduler er ook voor dat processen zich aan hun deadline kunnen houden; dit is cruciaal om het systeem stabiel te houden
- De term **scheduler** wordt ook gebruikt als benaming voor een programma dat op gezette tijden andere programma's start. Een voorbeeld hiervan is het programma **cron** in Unix-achtige besturingssystemen zoals **Linux**. Scheduling met cron gebeurt op een enkele machine. Scheduling op meerdere machines kan met Cronacle van Redwood, AutoSys van ComputerAssociates of Tivoli Workload Scheduler van IBM. Databases zoals Oracle en MySQL kennen ook een ingebouwd schedulingmechanisme.

Doelstellingen van scheduling

Veel scheduling strategieën hebben doelmatigheid en tevredenheid van de gebruiker tot doel. Maar ook de resources moeten effectief worden gebruikt. Met andere woorden, niet alleen moet het besturingssysteem iedereen gelukkig maken, maar dat moet ook snel en rendabel gebeuren.

De efficiëntie wordt niet bepaald door één metriek, maar kent meerdere metrieken die in evenwicht wordt gebracht door de scheduler.

We bespreken hierna de verschillende metrieken waarmee een scheduler rekening houdt.

Doorvoersnelheid (throughput)

- Het aantal processen per tijdseenheid door het systeem.
- Gevolg:
 - Lage doorvoersnelheid zorgt voor weinig processen
 - Hoge doorvoersnelheid zorgt voor veel processen
 - lijkt het interessant
 - Maar dit houdt geen rekening met procesgrootte
 - Het is het meest efficiënte systeem die sommige processen kan negeren
 - De redelijkheid wordt opgeofferd aan efficiëntie

Responstijd

- Interactieve gebruikers hebben nood aan een snelle respons
- Batch-gebruikers hebben nood aan een redelijke responstijd
- Men wil een snelle reactie op elk proces
 - Meer processen per tijdseenheid zijn noodzakelijk
 - De responstijd lijkt hetzelfde als de doorvoersnelheid
- De doorvoersnelheid kan vergroot worden door enkel korte processen te behandelen en lange processen te negeren
 - Lange processen worden niet beantwoord

Consistentie

- Er worden eisen gesteld aan het systeem
 - Bv. Dezelfde werking om 10u en om 15u
 - De responstijd moet ongeveer gelijk zijn
- Als de respons sterk varieert, dan weten de gebruikers niet wat ze mogen verwachten
 - Er zijn problemen met de werkindeling

Houd de processor aan het werk

- Het besturingssysteem moet resources aan het werk houden
- Bijvoorbeeld I/O-processors
 - Er moet meer aandacht zijn aan processen die veel I/O vragen
 - Hoe meer processors aan het werk, hoe meer er gedaan wordt
 - Als ze allemaal bezig zijn kunnen ze geen I/O rekest meer genereren
 - De CPU verwerkt andere processen die op CPU wachten
 - Het besturingssysteem moet dus minder ingrijpen en de systeem-efficiëntie neemt toe
- Het ideale geval: Het besturingssysteem houdt elke processor aan het werk, zonder deze zwaar te beladen

Prioriteiten

- Elk proces krijgt een prioriteit
 - Hoe hoger de prioriteit, hoe belangrijker
- Het besturingssysteem baseert zich hierop bij de scheduling
- Dit levert ook problemen
 - Wie bepaalt de prioriteiten ?
 - Wie kent ze toe ?
 - Welke richtlijnen zijn er ?
 - Wanneer is een proces belangrijker en verdient het daarom een hogere prioriteit ?
 - Wat gebeurt er als scheduling op basis van prioriteiten de systeemefficiëntie verlaagt ?

Real-time systemen

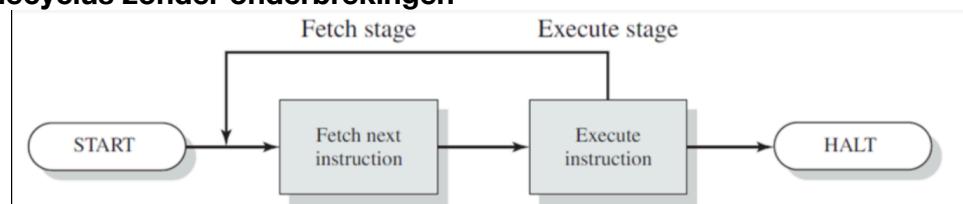
- Snelle respons voor besturing van doorgaand proces
 - Hoogste prioriteit
- Rechtvaardigheid en systeemefficiëntie
 - Lagere prioriteit voor de andere processen

Scheduling

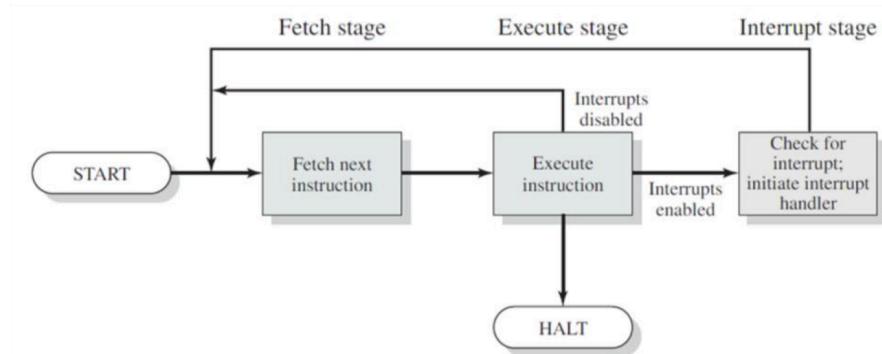
Scheduling is geen eenvoudige zaak: processen eisen soms verschillende zaken, en de taak van de scheduler is om dit voor elk proces zo veel mogelijk tegemoet te komen. Wat voor het ene proces efficiënt is, is dit niet voor het andere. Er wordt door de scheduler optimalisatie nastreefd tussen de verschillende metrieken.

Het precieze algoritme van een scheduler is dus behoorlijk complex - dit valt buiten de scope van deze cursus 😊

Instructiecyclus zonder onderbrekingen



Instructiecyclus met onderbrekingen



Uitvoeren van een proces

PC: Program Counter
AC: Accumulator
IR: Instruction Register

1XXX: copy to AC
 2XXX: copy from AC
 5XXX: add to AC

Fetch stage		Execute stage	
Memory	CPU registers	Memory	CPU registers
300 [1 9 4 0]	[3 0 0] PC	300 [1 9 4 0]	[3 0 1] PC
301 [5 9 4 1]	[] AC	301 [5 9 4 1]	[0 0 0 3] AC
302 [2 9 4 1]	[1 9 4 0] IR	302 [2 9 4 1]	[1 9 4 0] IR
:		:	
940 [0 0 0 3]		940 [0 0 0 3]	
941 [0 0 0 2]		941 [0 0 0 2]	
Step 1		Step 2	
Memory	CPU registers	Memory	CPU registers
300 [1 9 4 0]	[3 0 1] PC	300 [1 9 4 0]	[3 0 2] PC
301 [5 9 4 1]	[0 0 0 3] AC	301 [5 9 4 1]	[0 0 0 5] AC
302 [2 9 4 1]	[5 9 4 1] IR	302 [2 9 4 1]	[5 9 4 1] IR
:		:	
940 [0 0 0 3]		940 [0 0 0 3]	
941 [0 0 0 2]		941 [0 0 0 2]	
Step 3		Step 4	
Memory	CPU registers	Memory	CPU registers
300 [1 9 4 0]	[3 0 2] PC	300 [1 9 4 0]	[3 0 3] PC
301 [5 9 4 1]	[0 0 0 5] AC	301 [5 9 4 1]	[0 0 0 5] AC
302 [2 9 4 1]	[2 9 4 1] IR	302 [2 9 4 1]	[2 9 4 1] IR
:		:	
940 [0 0 0 3]		940 [0 0 0 3]	
941 [0 0 0 2]		941 [0 0 0 5]	
Step 5		Step 6	
Memory	CPU registers	Memory	CPU registers
300 [1 9 4 0]		300 [1 9 4 0]	
301 [5 9 4 1]		301 [5 9 4 1]	
302 [2 9 4 1]		302 [2 9 4 1]	
:		:	
940 [0 0 0 3]		940 [0 0 0 3]	
941 [0 0 0 5]		941 [0 0 0 5]	

Een proces bestaat uit...

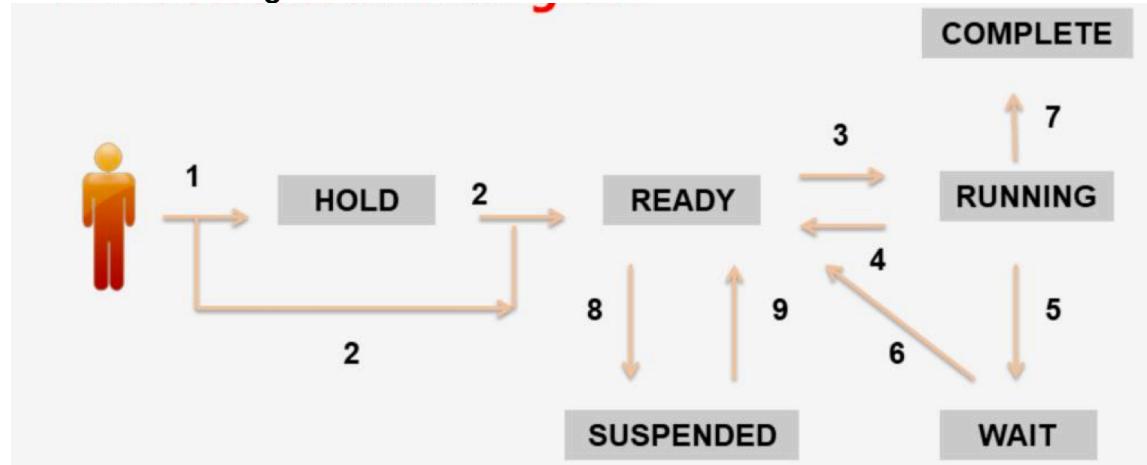
- Een context
 - ID / nummer
 - Status of toestand
 - PC (Proces Control)
 - Prioriteit
- Instructies
- Data

Besturingssysteem

- Bekijkt de processen die het computersysteem binnenkomen
- **Procestoestand**
 - = Status
 - Gedefinieerd in termen van de mogelijkheid om te worden uitgevoerd
 - Rechthoek in procestoestandsdiagram
- **Toestandsovergangen**

- Pijlen in procestoestandsdiagram

Procestoestand-diagram



Bij scheduling is het nodig de processen zoals deze de handen van de gebruiker verlaten en het computersysteem binnengaan, zorgvuldig te bekijken. Wij gaan na hoe een besturingssysteem de verschillende processen bekijkt. Laten wij de toestand (status) van een proces definiëren in termen van de mogelijkheid van uitvoering.

Het procestoestand-diagram toont de toestanden waarin een proces kan bestaan. De rechthoeken in het diagram vertegenwoordigen de toestanden. De pijlen ertussen vertegenwoordigen wijzigingen in de toestand: toestandovergangen.

Procestoestanden

- De toestand **HOLD** duidt aan dat het proces is aangeboden. Het besturingssysteem staat het alleen nog niet toe te worden uitgevoerd of resources aan te vragen. Deze toestand treedt op wanneer iemand een proces aanbiedt met de instructies dat het later moet worden gestart. Veel systemen kunnen een aangeboden proces initialiseren op een bepaald tijdstip, of nadat een bepaalde tijd is verstreken.
- De **READY**-toestand geeft aan dat het proces ready-to-run is: gereed om te worden uitgevoerd. Een proces in deze toestand is *idee* (ruststand: toestand waarin een systeem wacht, omdat er niets te doen valt), maar kan worden uitgevoerd als het de besturing over de CPU krijgt. In systemen met één processor kunnen veel READY processen staan te wachten, er kan er maar één tegelijk worden uitgevoerd. Deze toestand is niet van toepassing op een proces dat op I/O wacht.
- De toestand **RUNNING** geeft aan dat het proces wordt uitgevoerd en op dat moment onder de besturing van de CPU staat. In een single-processorsysteem is er maar één proces ind deze toestand. Bevat het systeem echter meerdere CPU's, dan kunnen verscheidene processen gelijktijdig worden uitgevoerd.
- De toestand **WAIT** geeft aan dat het proces op iets staat te wachten. Het kan bijvoorbeeld een I/O request hebben geproduceerd. Het kan niet worden uitgevoerd omdat het moet wachten tot de I/O actie klaar is
- De toestand **SUSPENDED** (opgeschort) is in één opzicht vergelijkbaar met de WAIT-toestand: het proces kan niet naar de CPU meedingen. Het verschilt echter

in die zin, dat processen in een weit-toestand nog wel om sommige resources mogen vragen. Het proces dat een I/O aanvraag gedaan heeft, staat in feite te wachten op een I/O-processor of een I/O-controller. Het proces is idle ten opzichte van de CPU, maar het vordert nog steeds. Een proces in de SUSPENDED-toestand kan geen enkele resource-aanvraag doen. Het proces wordt tijdelijk compleet stilgelegd. Processen kunnen bijvoorbeeld worden opgeschort wanneer het systeem te veel processen bevat. De intensieve strijd om resources kan de voortgang van alle processen vertragen. Een van de oplossingen daarvoor is het opschorten van een paar processen, zodat het systeem de rest efficiënter kan afhandelen. Wanneer de zware competitie om resources verminderd, kan het systeem de opgeschorte processen weer activeren.

- Een proces in de toestand **COMPLETE**, tenslotte, is volledig afgewerkt

Toestandsovergangen

Een toestandsverandering is een wijziging in de toestand van een proces. Bepaalde events kunnen een dergelijke wijziging vereisen. De toestandsbeschrijvingen geven u waarschijnlijk wel een indicatie wat sommige van deze events kunnen zijn, maar toch gaan we alle toestandsveranderingen beschrijven. **De opdrachten die nodig zijn om alle programma's aan te bieden, duiden we aan met Job Control Language (JCL).**

In het procestoestand-diagram zijn volgende toestandsveranderingen:

1. De overgang van **niet-aangeboden naar HOLD** vindt plaats wanneer een gebruiker een programma aanbiedt met instructies de uitvoering van dat programma tot later uit te stellen. De JCL kan een start- of vertragingstijd bevatten. Het programma blijft in de HOLD-status tot het juiste ogenblik is aangebroken
2. De **overgang van HOLD of niet-aangeboden naar READY** vindt plaats wanneer een gebruiker een programma aanbiedt dat van plan is direct een resource-aanvraag te doen. Deze overgang treedt ook op bij de start van een programma dat eerder op basis van vertraging is aangeboden
3. De overgang van **READY naar RUNNING** vindt plaats wanneer het besturingssysteem de besturing van de CPU aan een bepaald proces overdraagt. De basis waarop de CPU de processen selecteert, is belangrijk en daarom het onderwerp van een volgende paragraaf (strategieën voor scheduling).
4. De overgang van **RUNNING naar READY** vindt plaats wanneer het besturingssysteem de besturing van de CPU terugkrijgt van een proces dat, indien toegestaan, nog kan runnen. Sommige systemen bepalen bijvoorbeeld hoeveel tijd een proces zonder onderbreking mag runnen (het quantum). Als het quantum is verbruikt, neemt het besturingssysteem de besturing over de CPU weer terug. Het zet het proces in de READY-toestand en geeft de CPU aan het volgende proces in de rij. Op deze manier kan geen enkel proces de CPU monopoliseren
5. De overgang van **RUNNING naar WAIT** vindt plaats wanneer het proces iets aanvraagt waarop het moet wachten. We hebben bijvoorbeeld al gezien dat een proces dat om I/O verzoekt, moet wachten tot de I/O-actie is afgelopen
6. De overgang van **WAIT naar READY** vindt plaats wanneer een wachtend proces krijgt wat het nodig heeft. Zo kan bijvoorbeeld het besturingssysteem een signaal ontvangen dat een I/O-actie is afgerond. Daarna zet het het wachtende proces in de READY-toestand terug
7. De overgang van **RUNNING naar COMPLETE** vindt plaats wanneer het proces

afgelopen is. Dit betekent dat de noodzakelijke werkzaamheden zijn afgerond of dat er bij het proces iets fout is gegaan en het wordt afgebroken. In beide gevallen wordt het proces beëindigd, voor zover dat het besturingssysteem betreft.

8. De overgang van **READY** naar **SUSPENDED** vindt plaats wanneer er teveel READY processen zijn om nog adequaat service te verlenen. Elk computersysteem heeft beperkte resources en als het aantal processen dat toegang tot deze resources vraagt, zonder beperking kan groeien, raakt het systeem verzadigd als een overvolle snelweg, zodat er filevorming optreedt en elk proces maar een gebrekige service krijgt. Een manier om dit te vermijden is het aantal READY processen te beperken. Een andere manier om dit te vermijden is het aantal READY processen te beperken. Een andere manier is het verplaatsen van processen van de READY-toestand naar de SUSPENDED_toestand wanneer de responsitiden slecht worden. Welke processen hiervoor in aanmerking komen, is natuurlijk een beslissing die het besturingssysteem moet nemen.
9. De overgang van **SUSPENDED** naar **READY** vindt plaats wanneer het besturingssysteem besluit dat een opgeschort proces weer naar resources kan meedingen. Vermoedelijk is de belasting tot normale niveaus teruggebracht en is het niet langer nodig om processen in de suspendeer toestand te brengen.

Process Control Blocks (PCB)

Het besturingssysteem moet de informatie over het aantal processen op een systematische manier bijhouden. **Het besturingssysteem houdt een lijst bij van Process Control Blocks (PCB's).** In principe is er **één PCB voor elk proces**. Wanneer een proces wordt geïnitialiseerd, creëert het besturingssysteem een PCB voor dit proces en houdt een lijst van PCB's bij. Wordt een proces beëindigd, dan verwijdert het systeem het PCB uit de lijst.

Een PCB bevat alles wat het besturingssysteem over het proces zou moeten weten:

- **Identificatienummer van het proces** (proces-ID)
- **Procestoestand.** Als het proces van toestand verandert, werkt het besturingssysteem het PCB van dit proces bij.
- **Maximale looptijd en actuele looptijd** Gedurende de uitvoering van een proces gaat het systeem na hoeveel CPU-tijd het gebruikt. Bovendien mag de actuele looptijd de maximaal toelaatbare looptijd niet overschrijden. De maximale looptijd is een door het systeem gedefinieerde parameter en wordt bij de initialisatie van het PCB opgeslagen.
- **Huidige resources en limieten.** Hieronder vallen onder andere het aantal printerpagina's, de hoeveelheid geheugen en de hoeveelheid schijfruimte
- **Procesprioriteit.** Het systeem kan een proces schelen of het op basis van zijn prioriteit toegang tot resources geven. Sommige processen (bijvoorbeeld routines van het besturingssysteem) hebben een hoge prioriteit
- **Opslaggebieden.** Als een proces tijdelijk wordt stopgezet, moet het systeem bepaalde registerwaarden bewaren. Het systeem kan deze waarden dan later herstellen, zodat het proces de uitvoering op dezelfde plek kan hervatten als waar het stopte
- **Locatie an de code of de segmenttabel van een proces.** Wanneer het besturingssysteem een proces laat runnen, moet het weten waar de code of de

segmenttabel van dat proces zich bevindt.

Niveaus van scheduling

Scheduling vindt plaats op hoog, middel en laag niveau: high-level, intermediate level en low-level scheduling.

High-level scheduling (job scheduling) stelt vast welke programma's of jobs toegang tot het systeem krijgen. Besturingssysteem-routines die de high-level scheduling verzorgen, controleren en de Job Control Language (JCL) van een job. High-level scheduling regelt de toestandovergangen 1, 2 en 7 in het procestoestand-diagram.

High-level schedulers controleren een checklist voordat ze een job toestemming geven het systeem binnen te komen. Ze reageren eenvoudig op events en beschermen de integriteit en de veiligheid van het systeem. Er is relatief zelden behoefte aan high-level schedulers.

Scheduling op middelniveau (intermediate level) bepaalt welke processen in feite kunnen meedingen naar de CPU. Processen die I/O requests hebben geproduceerd kunnen daarbij niet meedoelen. Soms schort het systeem een proces op wanneer de vraag ongewoon hoog is. Intermediate-level scheduling houdt zich voornamelijk bezig met de toestandovergangen 5, 6, 8 en 9 in het procestoestand-diagram.

Net als high-level schedulers zijn ook de intermediate-level schedulers event-gestuurd. Events als een I/O-aanvraag of I/O-einde bepalen meestal welke toestandovergang plaatsvindt.

Low-level scheduling is verantwoordelijk voor de toestandovergangen 3 en 4 in het procestoestand-diagram.

In vele opzichten is scheduling op laag niveau het moeilijkst te implementeren. In tegenstelling tot andere niveaus is deze niet hoofdzakelijk event-gestuurd. Vaak zijn er veel processen die om CPU-tijd vragen en low-level scheduling moet vaststellen welk proces toegang krijgt.

In time sharing-systemen, waar processen de CPU om de beurt gebruiken, vindt low-level scheduling veelvuldig plaats. Is er veel I/O-activiteit, dan kan het elke paar milliseconden wel nodig zijn. Algoritmen voor low-level scheduling moeten proberen de CPU en de andere resources efficiënt toe te wijzen. Ze moeten een rechtvaardige respons naar de gebruikers garanderen.

Strategieën voor low-level scheduling

Er zijn twee hoofdcategorieën algoritmen voor low-level scheduling: algoritmen voor preemptive (preventieve) scheduling en voor nonpreemptive scheduling.

Bij **nonpreemptive scheduling** houdt een proces dat de besturing over de CPU krijgt, deze besturing tot het proces is afgelopen. Het besturingssysteem neemt het proces de besturing niet af. Het besturingssysteem geeft de besturing van de CPU alleen aan een proces wanneer een ander proces afgelopen is. Daarom is scheduling op een laag niveau zelden nodig bij nonpreemptive scheduling. Nonpreemptive scheduling heeft het nadeel dat het niet op andere processen reageert. Het proces dat de besturing over de CPU heeft, krijgt volledige service, maar alle andere processen moeten wachten.

Bij **preemptive scheduling** kan een proces niet oneindig lang de besturing over de CPU houden. Na een bepaalde periode kan het besturingssysteem besluiten de CPU van dit proces weg te halen. De processen in de READY-toestand moeten de CPU om beurten gebruiken. Om kort te zijn: het besturingssysteem beslist hierover.

In het algemeen kan het besturingssysteem een runningproces de besturing van de CPU ontnemen om verschillende redenen, bijvoorbeeld: het proces is beëindigd; het proces genereert een request waarop het moet wachten; de uitvoering van het proces heeft al een hele tijd geduurd. De maximale tijd die het besturingssysteem een proces zonder onderbreking laat runnen is een parameter van het besturingssysteem en wordt quantum genoemd. Als het quantum wordt bereikt, slaat het besturingssysteem de context van het proces op (statusword, de registers en de programmateller), en geeft het de CPU gedurende een bepaalde tijd aan een ander proces.

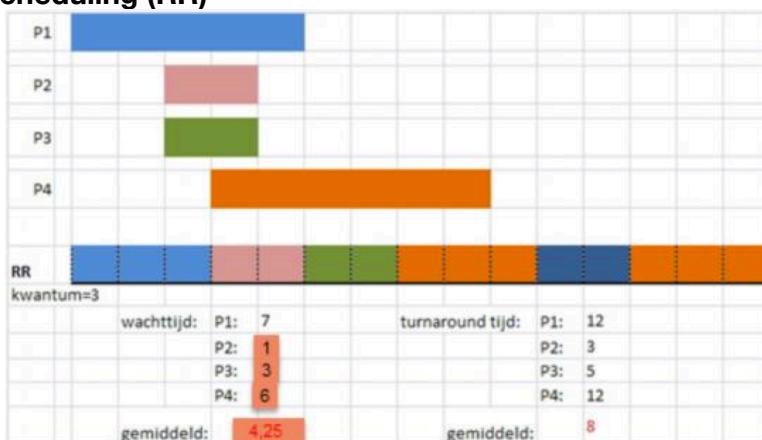
Criteria voor scheduler

Vereisten voor de scheduler: De verschillende algoritmen hebben verschillende eigenschappen en bij het kiezen van het algoritme voor de scheduler zal het karakter van het besturingssysteem bepalen welk algoritme gekozen wordt. Sommige systemen (bijvoorbeeld realtimebesturingssystemen) stellen strikte eisen aan het algoritme.

Een aantal criteria voor de keuze van de scheduler zijn:

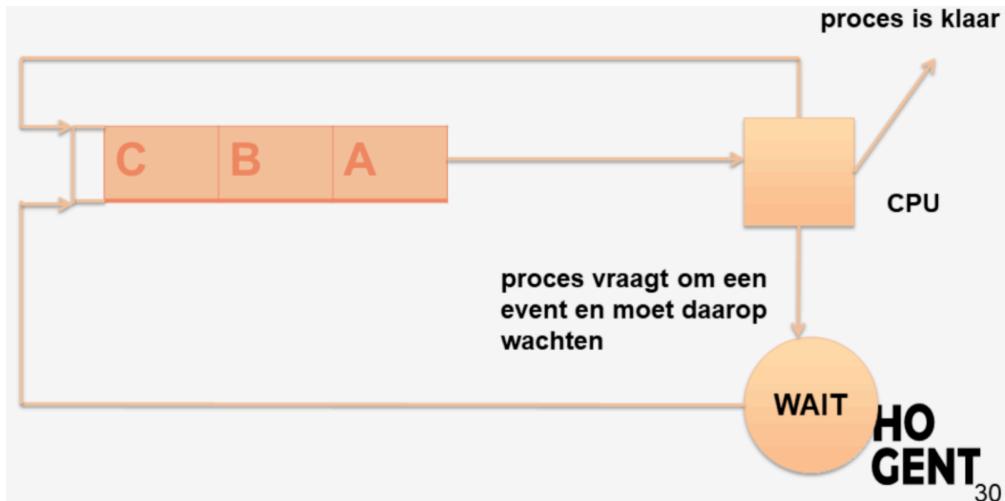
- **CPU-gebruik:** De mate waarin de processor gebruikt wordt door de verschillende processen
- **Debit:** Het aantal processen dat zijn werk voltooit in een bepaalde tijd, Turnaround-tijd: de tijd die nodig is om een proces te voltooien
- **Wachttijd:** De som van alle tijd die een proces doordringt in een wachtrij
- **Responstijd:** De tijd die nodig is om een reactie te krijgen. Het is de tijd tussen het indienen van een aanvraag en het krijgen van een antwoord

Round Robin Scheduling (RR)



In dit scheduling algoritme wordt gebruikgemaakt van een vaste tijdsmaarde, ook wel tijdkwantum genoemd. Wanneer dit tijdkwantum overschreden wordt, zal de scheduler het proces onderbreken en een volgend proces inladen. Een moeilijkheid is het bepalen

van de grootte van het tijdkwantum. Een te groot tijdkwantum zal er voor zorgen dat we een FCFS (First Come First Served) karakter krijgen en een te klein tijdkwantum zal voor een overhead aan context switches zorgen. Een context switch is het wisselen van processen. Wanneer de scheduler een proces onderbreekt zal hij de huidige status van het proces bewaren. Bij een te klein tijdsquantum zal de processor meer bezig zijn met het verwerken van context switches dan met het effectief uitvoeren van processen.



De preëemptieve Round Robin methode → Elk proces in de READY-toestand heeft een entry in de queue (gewoonlijk het PCB van het proces). Wanneer de CPU beschikbaar komt, geeft het besturingssysteem de besturing aan het proces waarvan het PCB aan het begin van de wachtrij staat, zodat het proces met de uitvoering begint. Het proces wordt verder uitgevoerd totdat één van de drie eerder aangegeven events optreedt. Op dat punt haalt het besturingssysteem de besturing van de CPU bij het proces weg. Als het proces afgelopen is, verwijdert de high-level scheduler het uit het systeem en verwijdert zijn PCB. Meestal creëert het besturingssysteem een overzicht van de gebruikte systeentijd en resources, en brengt dit gebruik in rekening. Daarna bestaat het proces niet langer in het denken van het besturingssysteem.

Als het proces niet stopt en evenmin een aanvraag genereert waarop het moet wachten, neemt het besturingssysteem het heft in handen. Het proces mag de CPU niet langer besturen dan een kwantum. Als een kwantum is verbruikt en het proces nog steeds wordt uitgevoerd, stopt het besturingssysteem het proces en zet het PCB aan het einde van de queue. Dan geeft het de CPU aan het proces dat vooraan in de queue staat. Het voortijdig onderbroken (preempted) proces moet wachten tot alle andere processen in de wachtrij een gelegenheid hebben gehad om de CPU te gebruiken. Preemption voorkomt dat processen de CPU monopoliseren. Geen enkel proces kan langer dan de kwantumtijd worden uitgevoerd zonder onderbroken te worden. Hierdoor krijgt elk proces in de READY-toestand gelegenheid gedurende een bepaalde tijd de CPU te gebruiken.

Round Robin Scheduling wordt het meest gebruikt in systemen met veel, achter terminal werkende, interactieve gebruikers.

Round Robin scheduling vereist echter enige overhead. Het besturingssysteem moet de activiteiten van elk proces van heel dichtbij volgen. Een ingebouwde timer moet elk proces onderbreken dat langer dan een quantum van de CPU gebruik probeert te maken. Wanneer een dergelijke interrupt optreedt, voert het besturingssysteem een rescheduling

uit en geeft de besturing van de CPU aan een ander proces. Hierdoor is het mogelijk dat het besturingssysteem meestal dezelfde CPU gebruiken als de processen die CPU-tijd willen hebben, vermindert dat de totale hoeveelheid tijd die voor processen beschikbaar is.

Daarnaast is de keuze van een quantumwaarde kritisch. Deze is bij Round Robin scheduling en sleutelwaarde en moet zorgvuldig gekozen worden

First-in-First-Out scheduling (FIFO) of First-come-first-served-scheduling (FCFS)



Dit is het eenvoudigste algoritme. Wanneer een proces als eerste om de cpu vraagt dan zal hij die ook krijgen. Processen die erna komen moeten wachten. Deze vorm van scheduling kan geïmplementeerd worden door een First-in-first-out (FIFO) model. Wanneer een proces lang zal duren, zullen korte processen die erna komen lang moeten wachten.



Deze nonpreemptive-methode is heel eenvoudig : Geef de besturing van de CPU aan het proces dat zich het langst in het systeem bevindt. Het proces dat het eerste aankwam mag de CPU gebruiken.

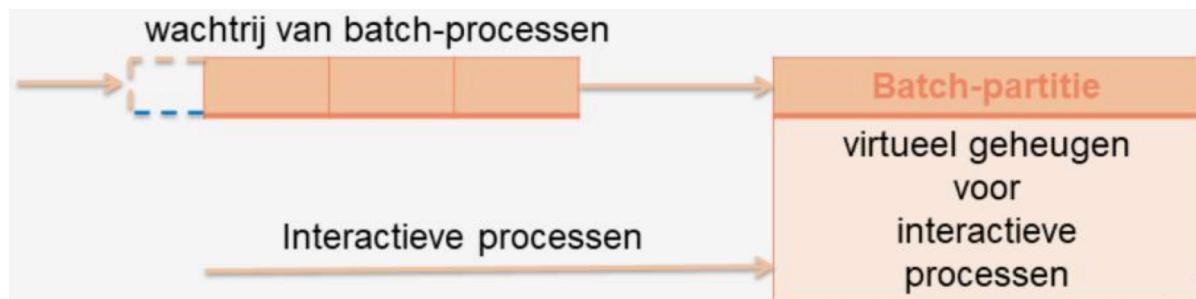
Wanneer een proces is geïnitialiseerd, plaatst het systeem het PCB van het proces aan het einde van de queue. Wanneer de CPU beschikbaar komt, geeft het besturingssysteem de besturing van de CPU aan een ander proces..

Het belangrijkste voordeel van deze methode is zijn eenvoud. Dit is iets wat nooit mag worden onderschat. Het besturingssysteem besluit alleen tot rescheduling als dat absoluut noodzakelijk is. De overhead is daardoor klein.

In vele systemen kan FIFO onrealistisch zijn (bijvoorbeeld in een hoog-interactief systeem met veel I/O-activiteit, toch zijn er gevallen waarin FIFO gewenst is (bijvoorbeeld een systeem waarin de meeste processen zwaar rekenwerk verrichten, maar erg weinig I/O-aanvragen produceren).

Een FIFO-scheduler kan ook deel uitmaken van een ingewikkeldere methode: bijvoorbeeld bij systemen die zowel batch-gebruikers als interactieve gebruikers hebben. Interactieve gebruikers laten korte programma's draaien, hebben weinig informatie nodig of ontwikkelen nieuwe applicatie en hebben een snelle respons nodig. Interactieve

gebruikers op terminals verwachten bijna onmiddellijk respons van het systeem. Korte onderbrekingen in de uitvoering worden direct opgemerkt. Een batch-gebruiker daarentegen biedt zijn of haar programme op een bepaalde tijd aan, samen met de JCL-opdrachten die nodig zijn om het te laten draaien. Nadat het is aangeboden, is de gebruiker echter vrij om iets anders te gaan doen. In tegenstelling tot de interactieve gebruiker hoeft hij of zij tijdens de verwerking van het proces niet aanwezig te zijn. De programma's van batch-gebruikers zijn meestal langer of willen veel uitvoer produceren. In tegenstelling tot interactieve gebruikers merken batch-gebruikers niets van korte onderbrekingen. Het besturingssysteem kan dit gebruiken om aan beide gebruikers een goede service te bieden.



Wanneer batch-processen en running processen worden gemengd, is een Round Robin schedule van alle processen niet gewenst, met name als er veel batch-gebruikers zijn. Deze hebben geen snelle respons nodig. Waarom zouden we dan niet wat CPU-tijd van batch-processen stelen en die aan de interactieve processen ter beschikking stellen ? De batch-gebruikers merken het verschil niet eens en de interactieve gebruikers zullen een verbeterde respons waarderen.

Een manier om deze hybride methode van scheduling te implementeren is met batch-partities. Een batch-partitie is een virtuele geheugenconstructie die één batch-proces bevat. Een virtueel geheugen is een geheugenbeeld dat het besturingssysteem aan de gebruiker verschaft. Er kunnen verscheidene batch-partities zijn, maar in dit voorbeeld gaan we uit van één.

Als de processen het systeem binnenkomen, maakt het systeem verschil tussen de batch-processen en de interactieve processen. Alle interactieve processen komen direct in de READY-toestand en dingen mee naar de CPU-tijd. Het systeem zet de batch-processen echter in een wachtrij voor de batch-partitie. Aangezien de partitie slechts één proces kan bevatten, moeten alle andere wachten.

Als de partitie leeg is, wordt het proces aan het begin van de queue in de partitie geplaatst. Het proces komt in de READY-toestand en dingt samen met de interactieve processen mee naar de CPU-tijd. Is de partitie vol, dan moet het proces dat vooraan in de queue staat wachten. Het resultaat is een quota-systeem waarbij slechts één batch-proces tegelijk de gelegenheid krijgt naar de CPU mee te dingen.

Hierin is de low-scheduler Round Robin, maar er is nooit meer dan één batch-proces READY, en de intermediate-level FIFO-scheduler bepaalt welke processen in de READY-toestand komen.

Round Robin en FIFO zijn twee veel voorkomende methoden bij de scheduling op laag-

en middelniveau, maar het zijn geenszins de enige methoden.

Round Robin is het meest geschikt wanneer gebruikers een snelle respons willen.

Round Robin past heel goed in systemen die batch-processen verwerken die veel I/O requests genereren. In dit geval willen we dat de processen hun eigen aanvragen zo spoedig mogelijk genereren. Dit houdt de I/O-processors actief en verkleint het aantal processen in de READY-toestand. Aangezien Round Robin garandeert dat alle READY-processen een kans krijgen om te worden uitgevoerd, vergroot deze de kansen dat er I/O requests worden gegenereerd.

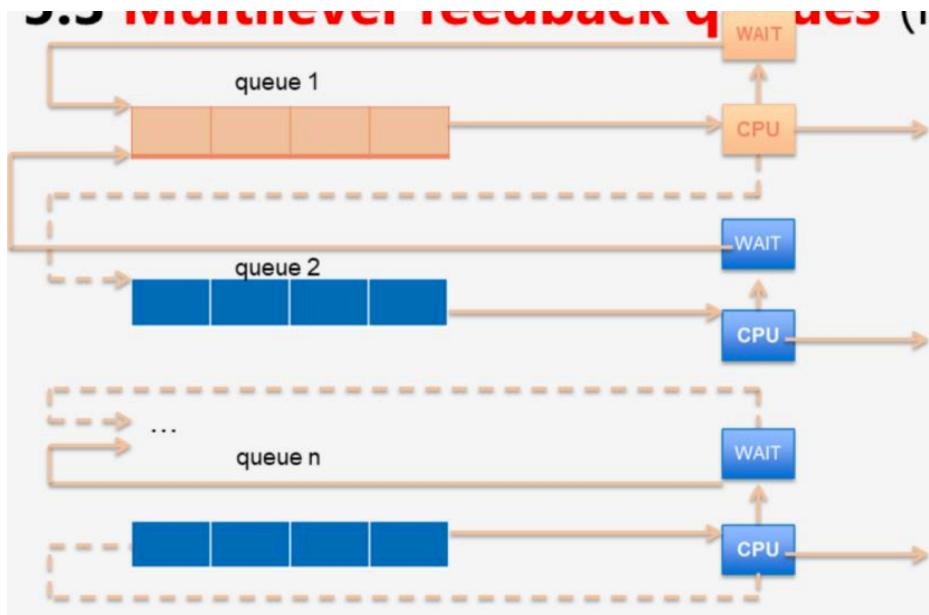
FIFO daarentegen is het meest geschikt voor omgevingen waarin zwaar rekenwerk moet worden verricht of als onderdeel van een ingewikkeldere scheduling-benadering. Er is niets te winnen door FIFO in interactieve omgevingen te gebruiken of wanneer er veel I/O-activiteit bestaat. Omgekeerd is er evenmin winst als we Round Robin in een rekenomgeving zouden toepassen.

Multilevel feedback queues

Computeromgevingen zijn dynamisch en het soort processen dat het systeem binnenkomt kan enorm variëren. Als we óf Round Robin óf FIFO gebruiken, zullen er momenten zijn waarop de scheduler zijn werk niet zo best doet.

Er bestaat een manier waarop de scheduling-methode op Round Robin kan lijken als er veel I/O-activiteit is en op FIFO kan lijken wanneer er weinig of geen I/O-activiteit is: dit wordt **Multilevel Feedback Queues (MFQ)** genoemd. Bij deze methode is de scheduling afhankelijk van de activiteiten die op een bepaald moment plaatsvinden. De beste scheduling-methode is afhankelijk van de soorten processen in de READY-toestand en het MFQ-systeem is gevoelig voor wijzigingen in die activiteiten (adaptieve methode).

Wanneer er veel I/O-activiteit plaatsvindt, lijkt MFQ op een Round Robin scheduler. Is er daarentegen maar weinig of geen I/O-activiteit, dan lijkt MFQ op een FIFO-scheduler. MFQ is nooit compleet FIFO, omdat het dan niet op veranderingen zou kunnen reageren in de tijd dat een bepaald proces de besturing over de CPU heeft. Het voert echter minder reschedules uit door elk proces de gelegenheid te geven gedurende langere tijd de CPU te bestuderen.



Zoals de naam suggereert, bestaat MFQ uit vele queue. Ze zijn genummerd van 1 tot n en elke wachtrij heeft een eigen prioriteit (q_1 heeft de hoogste prioriteit en q_n heeft de laagste prioriteit). Het PCB voor een proces in de READY-toestand is in één van de queues geplaatst. De prioriteit van het PCB wordt bepaald door de queue waarin het zich bevindt.

De low-level scheduler werkt altijd op basis van prioriteit. Een proces met lage prioriteit krijgt alleen de besturing van de CPU als er geen proces met een hogere prioriteit bestaat. Heeft een aantal processen dezelfde prioriteit, en is er geen met een hogere, dan kiest de scheduler het proces aan het begin van de queue (FIFO).

Tekens wanneer een nieuw proces wordt geïnitialiseerd, zet het systeem het in de READY-toestand en wordt het PCB van het proces in de eerste queue geplaatst. Elk nieuw proces heeft aan het begin dus de hoogste prioriteit. Is de CPU beschikbaar, dan zoekt de low-level scheduler de queue met de hoogste prioriteit die niet leeg is. Dan geeft de low-level scheduler de besturing van de CPU aan het proces dat vooraan in die queue staat.

Het proces krijgt de besturing van de CPU tot één van de volgende dingen gebeurt:

- Het proces wordt beëindigd (elegant of door het af te breken);
- Het proces vraagt iets aan waarop het moet wachten (bijvoorbeeld een I/O request);
- Het quantum is verbruikt

Het quantum is afhankelijk van de queue waaruit het proces is gekozen. Het is kenmerkend dat elke queue een ander quantum heeft. Queue 1 heeft het kleinste quantum en queue n heeft het grootste quantum.

Als het proces afgelopen is, verlaat het het systeem en speelt het verder geen rol.

Als het proces iets aanvraagt waarop het moet wachten, zet de intermediate-level

scheduler het in de WAIT-toestand. Wanneer het proces krijgt wat het nodig heeft, keert het terug naar de READY-toestand. Het besturingssysteem plaatst het PCB aan het einde van de queue met net iets hogere prioriteit dan de queue waarin het heeft gestaan. Het besturingssysteem verhoogt de prioriteit van het proces.

Stel dat het proces geen enkele aanvraag doet en niet stopt. Dan haalt het besturingssysteem de besturing van de CPU bij het proces weg nadat het quantum is verbruikt. In dit geval blijft het proces in de READY-toestand en het besturingssysteem plaatst het PCB aan het einde van de queue met net een iets lagere prioriteit dan de queue waarin het heeft gestaan.

Hoe zorgt het besturingssysteem ervoor dat de low-level scheduling zich aanpast aan het type processen die READY zijn ?

Al de processen ontvangen in eerste instantie de hoogste prioriteit.

Als de processen interactief zijn of I/O-gebonden, ontvangt het besturingssysteem geregeld I/O-aanvragen. Stel dat het quantum iets langer is dan de gemiddelde tijd tussen I/O requests. Hierdoor genereren de meeste processen I/O-requests voordat het quantum is verbruikt.

Wanneer een I/O-request is afgelopen, komt het proces uiteindelijk opnieuw in de READY-toestand, met een hogere of dezelfde prioriteit. Daarom heeft elk I/O-gebonden proces de neiging een hoge prioriteit te behouden en meer I/O-requests te genereren.

Als het proces enige tijd wordt uitgevoerd zonder een I/O-aanvraag, raakt het quantum op. In dit geval blijft het proces in de READY-toestand, maar met een lagere prioriteit (tenzij het de laagste prioriteit heeft). Is het proces echter nog I/O-gebonden, dan zal het spoedig meer I/O-aanvragen genereren en weer hogere prioriteit krijgen.

Stel dat de binnenkomende processen echt rekenintensief zijn. Deze zullen dan weinig I/O requests genereren, maar de meeste tijd spenderen aan het uitvoeren van code. Wanneer ze de besturing van de CPU ook krijgen, meestal zal het quantum worden verbruikt. Het gevolg is dat ze met een lagere prioriteit naar de READY-toestand terugkeren (tenzij ze al de laagste prioriteit hebben).

Als alle READY-processen rekenintensief zijn, krijgen ze uiteindelijk allemaal de laagste prioriteit. De low-level scheduler heeft dus vooral te maken met processen met een lage prioriteit. Het wordt een Round Robin Scheduler met een groter quantum dan voor I/O-gebonden processen wordt gebruikt. De grotere quanta zijn gerechtvaardigd omdat er met geregelde rescheduling van rekenintensieve processen niets wordt gewonnen. Hoewel de scheduler nog steeds Round Robin is, zorgen de grotere quanta voor een werkwijze die meer op FIFO lijkt.

Laten we vervolgens aannemen dat het gaat om een mengsel van I/O-gebonden en CPU gebonden processen. De I/O-gebonden processen behouden een hoge prioriteit, terwijl de prioriteit van de CPU-gebonden processen afneemt. De scheduler begunstigt de I/O-gebonden processen boven de CPU-gebonden processen. Door dit te doen, blijven de I/O-processors actief en neemt de graad van multiprocessing toe. CPU-gebonden processen worden alleen uitgevoerd wanneer er geen I/O-gebonden processen READY zijn.

Niet alleen is MFQ gevoelig voor verschillen tussen processen, het is ook gevoelig voor veranderingen binnen een proces. Zo kan een proces van start gaan als I/O-gebonden, maar opeens veranderen in een CPU-gebonden proces. Het omgekeerde kan ook voorkomen.

Het systeem met queues op verschillende niveaus (MFQ) is een flexibele methode die zorgt voor een automatische aanpassing aan de werkbelasting en aan veranderingen in het gedragspatroon van processen. Als een systeem hoofdzakelijk interactieve of CPU-gebonden processen heeft, is MFQ ingewikkelder dan nodig. In een onvoorspelbare omgeving kan deze methode echter productief zijn. Dit is in feite de scheduling-methode die door VMS wordt toegepast.

Shortest-job-first-scheduling (SJF)

De vorige methoden zijn best geschikt voor verschillende soorten computeromgevingen. Ze doen het echter allemaal wat minder goed als er te veel processen zijn. Als bijvoorbeeld de scheduler van het type FIFO is en de queue veel processen bevat, betekent dat lange wachttijden voor de processen achter in de rij. De vertraging is met name vervelend voor korte processen die achter langere processen staan te wachten. Vergelijkbare problemen kunnen zich voordoen bij een Round Robin scheduler. Als er te veel processen zijn, heeft de scheduler meer tijd nodig om ze allemaal een beurt te geven. Het gevolg is dan aanmerkelijk langere responstijden.

Is het logisch om korte processen een hogere prioriteit te geven om ze sneller uit het systeem te krijgen ? Als dat zo is, kan het systeem het aantal processen dat om resources vraagt kleiner houden.

Het welslagen of mislukken van de vorige methoden is sterk afhankelijk van het type activiteit dat de running processen nodig hebben. Daarom maakten we onderscheid tussen I/O-gebonden en CPU-gebonden processen, maar nooit tussen lange en korte processen. De bovenstaande voorbeelden suggereren dat we wel een dergelijk onderscheid zouden moeten maken.

Er zijn twee strategieën die aan korte processen een hoge prioriteit geven:

- **Shortest-job-first-scheduling (SJF):** Bij dit algoritme zal de scheduler het proces met de kleinste lengte uitvoeren. Een probleem is het voorspellen van de lengte van een proces. Dit kan gedaan worden door bijvoorbeeld het aantal instructies te tellen of het aantal in- en uitvoer aanvragen. Een voordeel ten opzichte van het FCFS-algoritme is dat de wachttijd bij SJF kleiner is. Het SJF-algoritme bestaat in twee vormen, de preemptive en de non-preemptive vorm. Een gevaar bij SJF-scheduling is dat een heel lang proces nooit uitgevoerd zal worden. Dit noemt men "verhongering" (Eng: starvation)
- **Shortest Remaining Job Next (SRJN):** Ze zijn in die zin vergelijkbaar, dat de low-level scheduler rekening houdt met de hoeveelheid tijd die een READY-proces nodig heeft. Eenvoudig gesteld, het kiest het proces dat de minste tijd nodig heeft. Het verschil tussen de twee strategieën is dat SRJN preemptive is en SJF nonpreemptive is.

Een nadeel van beide methoden is dat het besturingssysteem het moeilijk heeft om te berekenen hoeveel tijde en proces nodig heeft. Het is moeilijk (in feite meestal onmogelijk)

voor een besturingssysteem om te berekenen hoeveel tijd een proces nodig heeft. De persoon die het proces heeft aangeboden, zal echter best een redelijke schatting kunnen geven (zo niet, dan zou hij of zij de job waarschijnlijk beter niet kunnen aanbieden). Dergelijke ramingen maken in feite deel uit van de vereiste JCL. De low-level scheduler kan deze schatting dus gebruiken om te beslissen welk proces de besturing over de CPU krijgt. Bij SJF neemt de low-level scheduler het READY-proces met de kortst geschatte run-tijd. Heeft dat eenmaal de besturing van de CPU, dan draait het tot het is afgelopen (non preemptive).

Nonpreemptive scheduling creëert problemen als er I/O-requests zijn. Maar, als er maar weinig of geen I/O wordt verwacht, is SJF uitvoerbaar. Korte processen krijgen een prima respons omdat ze de hoogste prioriteit hebben. De doorvoer-snelheid is indrukwekkend. In feite genereert SJF de hoogst mogelijke doorvoer-snelheid. Als het erom gaat een zo groot mogelijk aantal tevreden gebruikers te krijgen, is dit de beste methode.

Het is echter mogelijk dat de doorvoersnelheid een verkeerde indruk van de productiviteit van het systeem geeft. Kijk maar naar de gebruiker wiens proces iets langer loopt dan veel andere. Het systeem straft dit af. Uiteraard kan het leiden tot indefinite postponement (onbepaald uitstel) of starvation (uithongeren). Dat wil zeggen, het kan eeuwig staan wachten. Zolang kortere processen het systeem blijven binnenkomen, zullen de langere niet worden uitgevoerd.

Het kan in hoge mate onrechtvaardig zijn om een proces uren te laten wachten, terwijl een iets korter proces wel wordt ingevoerd en het systeem al heel snel weer verlaat. Dit is vooral het geval als het langere proces van uzelf is.

De preemptive versie van SJF is SRJN. Een proces dat de besturing over de CPU heeft, kan hiervan afstand doen als het om iets vraagt waarop het moet wachten. Als een nieuw proces een kortere geschatte run-tijd heeft dan het proces dat op dat moment wordt uitgevoerd, krijgt het de besturing van de CPU. Net als SJF, genereert SRJN een hoge doorvoersnelheid, maar het lijdt ook aan dezelfde nadelen. Langere processen kunnen in onaanvaardbare mate worden vertraagd.

Starvation

We hebben gezien dat de methoden SJF en SRJN starvation van langere processen kunnen veroorzaken. De MFQ-methode kan ook starvation van processen in de lagere queues tot gevolg hebben. Zolang de interactieve processen READY zijn, worden de CPU-gebonden processen genegeerd. Kunnen we iets aan starvation doen of moeten we ermee leven en het als bijproduct van een bepaalde methode beschouwen ?

Eén mogelijkheid is dat we het negeren en hopen dat het geen ernstige problemen veroorzaakt. Dit is niet altijd een onrealistische benadering. Onder MFQ bijvoorbeeld verlaten interactieve processen de queues snel. Als gevolg van het verschil tussen de reken- en de I/O-snelheid duurt het voor deze processen veel langer om terug te keren. Daarom is het niet ongewoon te constateren dat de queues met hoge prioriteit snel leeg zijn. Natuurlijk ontstaat hierdoor voor de processen met lage prioriteit een gelegenheid om te runnen. Bij de methoden SJF en SRJN kan men vergelijkbare observaties uitvoeren. De ongelijkheid tussen de verwerkingssnelheden van de computer en de mens is hier zelfs nog groter. ER zijn waarschijnlijk niet veel gebruikers die korte processen sneller kunnen genereren dan een CPU deze kan uitvoeren. In deze gevallen is starvation zeldzaam. Zelfs

wanneer het optreedt, houdt het niet lang aan. OOK al zeggen we dat events waarschijnlijk niet voorkomen, dat wil niet zeggen dat zij onmogelijk zijn. Starvation van CPU-gebonden processen onder MFQ is aannemelijker als het aantal interactieve processen toeneemt. Op dezelfde manier is starvation van lange processen onder SJF of SRJN waarschijnlijker als het aantal gebruikers toeneemt. In dergelijke gevallen is starvation een ernstiger probleem.

Als we ervoor kiezen het niet te negeren, wat kan het systeem er dan aan doen ? Eén methode is het opschorten van een aantal READY-processen. Dit reduceert het aantal processen en vermindert starvation. Het systeem hervat de processen wanneer het besluit dat het er meer aankan.

Een ander alternatief is dat het systeem de prioriteiten periodiek opnieuw berekent. Het systeem kan dit doen met behulp van het veld LastTime dat zich in het PCB van elk proces bevindt. In het begin wordt daar de tijd opgeslagen waarop het proces het systeem is binnengekomen. Telkens wanneer een proces wordt onderbroken (preempted), vervangt het systeem de waarde in het veld door het tijdstip van preemption. Onder nonpreemptive scheduling verandert het veld nooit. Wanneer het systeem de prioriteit van een proces evalueert, bekijkt het de waarde ‘huidige tijd - LastTime’. Een grote waarde betekent dat het proces al lange tijd is genegeerd. Als de waarde toeneemt tot voorbij een bepaalde drempelwaarde, verhoogt het systeem de prioriteit voor dat proces. Onder MFQ betekent dit dat het PCB in een hogere queue wordt geplaatst. Onder SJF of SRJN betekent het dat het proces al lange tijd is genegeerd. Als de waarde toeneemt tot voorbij een bepaalde drempelwaarde, verhoogt het systeem de prioriteit voor dat proces. Onder MFQ betekent dit dat het PCB in een hogere queue wordt geplaatst. Onder SJF of SRJN betekent het dat het prioriteitsveld van het PCB wordt veranderd.

Bij Round Robin en FIFO kan geen starvation optreden

Scripting in Linux

Inleiding

Een **shell script** is een bestand met instructies die door de shell (bash) gelezen en begrepen moeten worden. Met de scripttaal die in bash is ingebouwd is het mogelijk een volledig functionerend programma te schrijven.

Shell soorten: sh, bash, csh, tcsh, ksh

In het bestand **/etc/shells** vind je een overzicht van alle gekende shells op jouw linux systeem.

Shell types:

- **sh of Bourne Shell:** de originele shell. Nog steeds gebruikt op UNIX systemen en in UNIX-gerelateerde omgevingen. Dit is de basic shell, een klein programma met extra mogelijkheden. Niettegenstaande het niet de standaard shell is, is het nog steeds beschikbaar op elk Linux systeem voor compatibiliteit met UNIX programma's.
- **bash of Bourne Again shell:** De standaard GNU shell, intuïtief en flexibel. Misschien de meest aanraadbare voor beginnende gebruikers alsook gevorderde gebruikers. Op Linux is **bash** de standaard shell voor de meeste gebruikers. Deze shell is zogezegd *superset* van de Bourne shell, een set van add-ons en plug-ins. Dit betekent dat de Bourne Again shell compatibel is met de Bourne shell: commando's die werken in **sh** werken ook in **bash**. Hoewel het omgekeerde niet altijd het geval is.
- **csh of C shell:** De syntax van deze shell komt overeen met dat van de **C programmeertaal**.
- **tcsh of TENEX C shell:** Een superset van de gekende C shell, verbetert gebruiksvriendelijkheid en snelheid. Dit is waarom het soms ook Turbo C Shell genoemd wordt.
- **ksh of Korn Shell:** Soms geapprecieerd door mensen met een UNIX achtergrond. Het is een superset van de Bourne shell.

De shell die standaard opgestart wordt bij het aanmelden staat vermeld in het bestand **/etc/passwd**

Enkele eigenschappen van een goed script

- Een script moet foutloos zijn
- Een script moet de taak uitvoeren waarvoor het bestemd is
- De programma logica in een script moet eenduidig en duidelijk omschreven zijn
- Een script doet geen onnodig werk
- Een script moet herbruikbaar zijn

De structuur van een shell script

De structuur van een shell script is zeer flexibel. Hoewel in Bash veel vrijheid wordt verleend, moet je zorgen voor een juiste logica, een correcte flow control en efficiency, zodat gebruikers gemakkelijk en correct het script kunnen laten uitvoeren.

Een goede gewoonte is elk script te beginnen met

- **#!/bin/bash** --> Hash bang: de interpreter voor dit script
 - De "**hash-bang**" of "**sha-bang**" (**#!**) aan het begin van het script bepaalt welke "interpreter" moet gebruikt worden om het script uit te voeren. Meestal zie je hier /bin/bash, maar dat kan bijvoorbeeld ook /usr/bin/perl zijn. Als je het script uitvoerbaar maakt (chmod +x *mijn_script*), dan kan je het oproepen met ./*mijn_script*, ongeacht de taal. Een perl-script hoef je dan niet op te roepen met "perl *mijn_script.pl*"
- # een korte beschrijving van wat het script doet of hoe het te gebruiken

Voorzie je script ook steeds van voldoende regels commentaar, dit komt de leesbaarheid ten goede.

Het is ook aangeraden volgende 2 instructies bovenaan je script op te nemen:

- # Script stoppen als instructie faalt
 - **set -o errexit**
- # Script stoppen bij gebruik van niet-gedefinieerde variabele
 - **set -o nounset**

Een voorbeeld van een Bash script: mijnsysteem.sh

```
#!/bin/bash
clear
cat << _EOF_
Deze informatie wordt aangeboden door mijnsysteem.sh
Het programma start nu

Hallo ${USER}
Vandaag zijn we $(date) en dit is week $(date + "%V") .

Deze gebruikers zijn aangemeld:
$(w | cut -d " " -f 1 | grep -v USER | sort -u)

Dit OS is $(uname -s) en draait op een $(uname -m) processor.
```

Uptime informatie:

```
$(uptime)
```

```
_EOF_
```

Een script begint altijd met #!. Dan zal de daarop volgende commando sequentieel uitvoeren.

Het script begint met het scherm te ledigen en drukt de volgende boodschappen af: <<Deze info>> <<Het programma...>> om de gebruiker in te lichten van wat er zal gebeuren. Daarna wordt de huidige datum en nummer van de week afgedrukt op het scherm. Daarna wordt een alfabetische lijst getoond van de gebruikers die aangemeld zijn op het systeem. Dan wordt het besturingssysteem en de CPU informatie getoond en als laatste de uptime informatie.

Echo of printf zijn **built-in commando's**. Wanneer de naam van een built-in commando gebruikt wordt als eerste woord van een eenvoudig commando, zal de shell die onmiddelijk uitvoeren zonder een nieuw proces aan te maken.

Opmerking over het script:

- Als je veel tekst op het scherm wil afdrukken is het beter "here documents" (cfr. Labo 2) te gebruiken ipv het "echo" commando. Zo vermindert je het herhalen van de "echo's" en dat maakt het script makkelijker leesbaar.
- De notatie \$(COMMANDO) wordt 'command substitution' genoemd (zie verder).
- Rond de naam van variabelen worden vaak accolades gezet (bv \${USER}) om het te onderscheiden van tekst er rond. \$USER mag ook.

Het script kan je laten uitvoeren aan de hand van het commando **bash mijnsysteem.sh** of **./mijnsysteem.sh**

```
$ chmod +x mijnsysteem.sh  
$ ./mijnsysteem.sh  
Deze informatie wordt aangeboden door mijnsysteem.sh  
Het programma start nu
```

Hallo Bert

Vandaag zijn we wo apr 18 11:05:22 CEST 2012 en dit is week 16

Deze gebruikers zijn aangemeld:
Bert

Dit OS is Linux en draait op een x86_64 processor.

Uptime informatie:
11:05:23 up 1 day, 45 min, 1 user, load average: 0.11, 0.29,
0.30

Het schrijven van een script

Om een shell script te schrijven open je met een teksteditor (vim, emacs, gedit, nano,...) naar keuze een leeg bestand.

Het beste is om te kiezen voor een geavanceerde teksteditor zoals vim of emacs, omdat deze geconfigureerd kunnen worden om shell en Bash syntax te herkennen. Deze kunnen dus een grote hulp zijn om beginnersfouten te vermijden zoals het vergeten van haakjes en puntkomma's.

Om de markering van de syntax te activeren in vim kan je dat doen met het commando :syntax enable.

Om deze setting standaard te maken kan je deze bijvoegen in het bestand .vimrc

Onder fedora krijg je syntax-markering als je "vi" opstart met het commando "vim". Met het commando "vi" krijg je de "basisversie" van vi die voorkomt op alle UNIX-varianten. Vim (VI IMproved) heeft meer features, maar is niet volledig compatibel meer met vi op ander UNIX-systemen.

Typ jouw commando's in dit leeg bestand zoals je ze zou invoeren in de commandoregel. Bewaar dit bestand onder een zinvolle naam die reeds duidelijk maakt wat het script doet. Om duidelijk te maken dat het een script is wordt meestal de extensie .sh gebruikt.

Maak gebruik van de commando **which**, **whereis**,... om te kijken of er niet reeds een shell script met deze naam bestaat.

Je kan ook scripts schrijven in een andere "taal" dan Bash, bijvoorbeeld Perl of Python. De conventie is om altijd een extensie toe te voegen aan een script dat aangeeft in welke taal het geschreven is (bv. .sh voor Bash, .pl voor Perl, .py voor Python enz,...). Als het script echter in een bin/ directory in het \$PATH staat, wordt de extensie weggelaten. Dankzij de Hash-bang wordt dan automatisch de juiste interpreter gebruikt. Als er geen hash-bang voorkomt, zal de Bash interpreter proberen om het script uit te voeren.

Het activeren van een script

Om een shell script als zelfstandige opdracht te activeren moet het script eerst worden voorzien van de execute-permissie als volgt: **chmod u+x naam_van_het_script**.

Een mogelijkheid om een shell script te activeren is door het aan te roepen als argument van de shell als volgt: **bash -x naam_van_het_script**. Hier is het nodig de execute-permissie aan het script te verbinden.

Een andere mogelijkheid is met het commando **source naam_van_het_script**

Voorbeeld:

`chmod u+x voorbeeld1.sh`

Kijken of de rechten toegekend zijn:

`ls -l voorbeeld.sh`

Uitvoeren van het script

`./voorbeeld1.sh`

Source gebruik je als je geen nieuwe shell wil opstarten, maar je het script wil uitvoeren in je huidige shell. Dit heeft als gevolg dat indien er wijzigingen gebeuren in jouw werkomgeving ten gevolge van de uitvoering van het script die onmiddellijk zichtbaar zullen zijn.

Het debuggen van een bash script

Bash voorziet in uitgebreide debugging mogelijkheden. De meest voorkomende is het opstarten van de subshell met de **-x** optie, die het hele script in debug mode zal draaien.

```
nathalie@nathalie-laptop:~$ bash -x mijnsysteem.sh
+ clear

+ echo 'Deze informatie wordt aangeboden door de mijnsysteem.sh.'
Deze informatie wordt aangeboden door de mijnsysteem.sh.
+ echo 'Het programma start nu!'
Het programma start nu!
+ echo

+ echo 'Hallo, '
Hallo,
+ echo

++ date
++ date +%%V
+ echo 'Vandaag zijn we ma apr 19 18:41:32 CEST 2010, en dit is week 16.'
Vandaag zijn we ma apr 19 18:41:32 CEST 2010, en dit is week 16.
+ echo

+ echo 'Volgende gebruikers zijn momenteel aangemeld:'
Volgende gebruikers zijn momenteel aangemeld:
+ w
```

Met behulp van het commando **set**, kan je de delen waarvan je zeker bent dat ze foutloos zijn laten uitvoeren in normale mode.

Door **set -x** in te voegen in jouw script, activeer je vanaf daar de debugging en door **set +x** stop je de debugging.

Je kan ook het commando **echo** invoegen in jouw script om de inhoud van variabelen te tonen en zo de werking van jouw script te controleren.

```
#!/bin/bash
clear
echo "Deze informatie wordt aangeboden door de mijnsysteem.sh."
echo "Het programma start nu!"
echo
echo "Hallo, $user"
echo
echo "Vandaag zijn we `date`, en dit is week `date +\"%V\"`."
echo
echo "Volgende gebruikers zijn momenteel aangemeld:"
echo
set -x
w | cut -d " " -f 1 | grep -v USER | sort -u
set +x
echo
echo "Dit is `uname -s` die draait op een `uname -m` processor."
echo
echo "Ziehier de uptime informatie:"
uptime
echo
```

Deze informatie wordt aangeboden door de mijnsysteem.sh.
Het programma start nu!

Hallo,

Vandaag zijn we ma apr 19 18:56:18 CEST 2010, en dit is week 16.

Volgende gebruikers zijn momenteel aangemeld:

+ w
+ sort -u
+ cut -d ' ' -f 1
+ grep -v USER

nathalie

+ set +x

Dit is Linux die draait op een i686 processor.

Ziehier de uptime informatie:
18:56:18 up 3:57, 3 users, load average: 0.00, 0.00, 0.00

Variabelen en parameters in scripts

Variabelen:

Je kunt op volgende manieren een waarde aan een variabele koppelen:

1. **variabele=waarde**
2. **variabele="meerdere waarden"**
3. **variabele=**

Waarde van een variabele kan aangeroepen worden

door **\$naam_varabele** of **\$(naam_varabele)**

Gebruik **steeds** de 2e notatie om problemen te vermijden.

In 1 en 2 wordt een waarde aan de variabele gekoppeld; in 3 wordt de variabele gedefinieerd, maar er wordt geen waarde gegeven.

In een script hebben we volgende types variabelen:

- **Read-only variabelen**
 - Variabelen die beschermd zijn tegen overschrijven. Een variabele wordt read-only gemaakt door het commando **readonly naam_varabele**
- **Lokale variabelen**
 - Variabelen die alleen beschikbaar zijn voor de shell waarin ze gedefinieerd zijn
- **Omgevingsvariabele**
 - Variabele die ook beschikbaar is in subshells van de huidige shell

Aan de hand van het commando **export** kan je een lokale variabele een omgevingsvariabele maken.

Shell-variabelen zijn variabelen die nodig zijn om de shell op de juiste wijze te laten functioneren. Ze worden ook wel systeemvariabelen genoemd. Voorbeelden: PWD, UID,...

Algemene toekenning:

variabele=waarde	#geen SPATIES rond = !!!
variabele=string	#geen SPATIES rond = !!!

Met het commando **declare**

Syntax: declare optie(s) variabele=waarde

Enkele opties:

- -a --> Variabele is een tabel
- -I --> Variabele is een integer
- -p --> Toont de kenmerken en de waarde van de variabele
- -r --> Variabele is readonly

Opmerking ! Bash heeft een optie om een numerieke waarde te declareren, maar geen voor alfanumerieke waarden. Dit komt omdat standaard ervan uitgegaan wordt als geen declaratie plaatsvindt de variabele van alles kan bevatten en dus als string beschouwd wordt.

Voorbeeld:

```
declare -i VARIABELE=12
```

VARIABELE=string

```
echo $VARIABELE
```

Het resultaat is 0

Voorbeeld2:

VAR=test

```
declare -p VAR
```

--> Het resultaat is declare VAR="test"

Constanten:

Aanmaken door een variabele readonly te maken **readonly OPTION VARIABELE(s)**

Lokale variabelen:

Toevoeging van het prefix "local" aan een variabele binnen functies van een script.

--> Variabele bestaat enkel tijdens het uitvoeren van de functie

Voorbeeld:

```
readonly TUX=pinguin
```

TUX=eend

Het resultaat is bash: TUX: readonly variabele

Belangrijkste omgevingsvariabelen

Variabele	Beschrijving
\$HOME	Home map van de gebruiker
\$USER / \$USERNAME	Naam van de gebruiker
\$PWD	Huidige werkmap
\$BASH / \$SHELL	Locatie van bash of shell
\$BASH_VERSION	Versie van Bash
\$OSTYPE	Info over het OS
\$PATH	Alle locaties voor uitvoerbare bestanden
\$IFS	Bevat alle scheidingstekens

Belangrijkste shellvariabelen:

Variabele	Beschrijving
\$0	Scriptnaam
\$#	Aantal meegegeven parameters
\$@	Alle meegegeven parameters
\$1, \$2, ... \${x}	Positionele parameter - x is getal voor laatste positie
\$?	Uitvoerstatus laatste commando
\$\$	Proces-id van het script
\$!	Proces-id van het laatste commando opgestart met &

Werken met invoer

Met het commando **read** kan je tekst inlezen van het toetsenbord en deze tekst opslaan in een variabele.

Voorbeeld:

```
#!/bin/bash
# lees
clear
echo "Geef een naam: ";read naam1
echo "Geef nog een naam: ";read naam2
echo "De eerste naam is $naam1."
echo "De tweede naam is $naam2."
echo
```

Geeft een naam:
Jan

Geeft nog een naam:
Johan

De eerste naam is Jan.
De tweede naam is Johan.

Probeer over het algemeen het gebruik van "read" te vermijden: een bash script moet taken automatiseren, en tijdens het uitvoeren nog informatie vragen is vervelend. Je kan beter alle nodige informatie doorgeven als opties of argumenten (zie verder), of eventueel in het begin van het script als waarde toewijzen aan variabelen.

Positionele parameters:

Alle argumenten die aan een script of een opdracht meegegeven kunnen worden, worden ook wel positionele parameters genoemd.

Het eerste argument is de pp nummer 1, het tweede argument is de pp nummer 2, enz,...

In de opdracht **ls -l /etc** is -l dus de eerste en /etc de tweede pp. In shell scripts worden deze pp aangeduid als **\$1** en **\$2**. Om een pp op te vragen hoger dan 9 gebruik je accolades, dus **\${10}**.

De naam van het script zelf is **\$0**.

```
#!/bin/bash
# parameters [parameter1, parameter2, ... parametern]
# Dit script wordt gebruikt om te tonen
# welke parameters zijn gebruikt
clear
echo "De opdrachtnaam is " $0
echo "De eerste parameter is " $1
echo "De negende parameter is " ${9}
echo
```

nathalie@nathalie-laptop:~\$ bash parameters.sh Hallo, wie ben jij?



```
De opdrachtnaam is parameters.sh
De eerste parameter is Hallo,
De negende parameter is
```

Als je een argument wil doorgeven aan een script waar spaties in voorkomen (bijvoorbeeld "Bert Verschaeve" in één keer), dan zet je er quotes rond. Probeer bijvoorbeeld bash parameters.sh "Hallo, wie ben jij ?"

De waarden van de positionele parameters zijn niet rechtstreeks aanpasbaar zoals bij het toekennen van waarden aan (omgevings)variabelen. Door het interne commando **shift** is het wel mogelijk om de positionele parameters een positie door te schuiven. Elke keer de opdracht shift wordt uitgevoerd, wordt de waarde van \$2 in \$1 gezet, de waarde van \$3 in \$2, de waarde van \$4 in \$3,...

Deze operatie vermindert dan ook de waarde van \$# met 1 !

```
#!/bin/bash
# parameters [parameter1, parameter2, ... parametern]
# Dit script wordt gebruikt om te tonen
# welke parameters zijn gebruikt
clear
echo "De opdrachtnaam is " $0
echo "De eerste parameter is " $1
echo "De tweede parameter is " $2
echo "De negende parameter is " ${9}
shift
echo "De tiende parameter is " ${10}
shift
echo "De twaalfde parameter is " ${12}
echo
```

nathalie@nathalie-laptop:~\$ bash parameters.sh a b c d e f g h i j k l m n o p



```
De opdrachtnaam is parameters.sh
De eerste parameter is a
De tweede parameter is b
De negende parameter is i
De tiende parameter is j
De twaalfde parameter is k
```

Een tweede manier om toch de oorspronkelijke parameters te overschrijven is door gebruik te maken van het commando **set**. Als je aan set een waarde van een variabele meegeeft dan zal Linux deze op **basis van \$IFS** opdelen en elk stuk plaatsen in een pp te beginnen vanaf \$1.

```
#!/bin/bash

echo "Meegegeven parameters ($#): $@"
x="een twee drie vier"
echo "Oproepen set met volgende waarde: ${x}"

set ${x}
echo "Nieuwe parameters ($#): $@"
```



```
ubuntu@ubuntu:~$ bash setex.sh 1 2 3 4
Meegegeven parameters (4): 1 2 3 4
Oproepen set met volgende waarde: een twee drie vier
Nieuwe parameters (4): een twee drie vier
```

Speciale parameters:

Dit zijn tekens met een speciale betekenis, die gebruikt worden om te praten over de pp van een opdracht.

Speciale parameter	Betekenis
\$#	Verwijst naar het aantal gegeven parameters
\$*	Geeft als resultaat één tekenreeks waarin alle parameters voorkomen. Elke parameter wordt van de vorige gescheiden door het scheidingsteken dat is gedefinieerd als waarde van de systeemvariabele IFS
\$@	Geeft als output alle parameters waarbij elke parameter als individuele tekenreeks kan worden gebruikt

```
#!/bin/bash
# telarg [par1, par2, ..., parn]
# Dit script definieert een functie waarin met
# $# het aantal argumenten wordt geteld.
# Deze functie wordt daarop eerst aangeroepen waarbij
# naar het aantal argumenten wordt verwezen met $*.
# Daarna wordt naar het aantal argumenten verwezen met $0.

clear
function telze
{
    echo "Het aantal argumenten is gelijk aan $#."
}

telze "$*"
telze "$@"
echo
```

```
nathalie@nathalie-laptop:~$ bash telarg.sh 1 2 3 4
```



```
Het aantal argumenten is gelijk aan 1.
Het aantal argumenten is gelijk aan 4.
```

Hier blijkt duidelijk de werking van \$ en \$@*

Command substitution

- Hiermee kan je de uitvoer van een commando opvangen en bv. in een variabele stoppen
- Syntax: **\$(commando)** of `commando`
- Om bijvoorbeeld de output van **whoami** in variabele **wiebenik** te plaatsen: **wiebenik=\$(whoami)**
- Normaal stuurt een commando zijn resultaat naar standard-output of standard-error, nu wordt de output in een variabele gezet.

Tegenwoordig wordt over het algemeen de nieuwere notatie **\$()** gebruikt ipv ``(backticks).

\$() is makkelijker leesbaar: de verschillende soorten quotes zijn moeilijker te onderscheiden. In een tekstverwerker die "smart quotes" invoegt, gebeuren er regelmatig fouten in het gebruik van quotes, wat tot verwarring kan leiden. Met **\$()** kan je de bedoeling beter duidelijk maken

```

#!/bin/bash
# In dit script wordt het resultaat van de opdracht date,
# opgeslagen in positionele parameters.
clear
echo $1
set `date` # Date command enclosed in backticks
echo $1
echo
echo $2
echo
echo $3
echo

```

nathalie@nathalie-laptop:~\$ bash opdrachtnsub.sh

↓

ma
apr
19

Reguliere expressies

Reguliere expressies (afgekort regex) is een (bijna) **gestandaardiseerde**, krachtige manier om **zoekpatronen** te definiëren.

Door te standaardiseren voorkom je dat elk commando een eigen definitie voor zoekpatronen hanteert (met nadelige gevolgen voor de gebruiker)

Reguliere expressies worden oa gebruikt door grep, sed, awk, vi, less, nano maar ook in Java, .NET, PHP, Perl, Python, JavaScript,...

Teken	Doel
.	Jokerteken. Vervangt elk willekeurig teken met uitzondering van \n
^	Verwijst naar het begin van de regel
\$	Verwijst naar het einde van de regel
<	Verwijst naar het begin van een woord
>	Verwijst naar het einde van een woord
[]	Matcht met één van de tekens tussen de haken. Vb. [ab] is hetzelfde als [a b]; [a-d] = [a b c d]
[^]	Matcht met alle tekens die niet tussen de haken staan Voorbeeld: ^[^a-z] regel die niet begint met een kleine letter
()	Groepering. Voorbeeld: (va moe)der matcht met 'vader' en 'moeder'; (groot)?vader matcht met 'vader' en 'groot'vader
?	Voorgaand teken of reguliere expressie komt nul of één keer voor
*	Voorgaand teken of reguliere expressie mag 0, 1 of meerdere keren voorkomen
+	Voorgaand teken komt één of meerdere keren voor
{n}	Voorafgaand teken komt juist n keer voor
{n,}	Voorafgaand teken komt minstens n keer voor
{,n}	Voorafgaand teken komt hoogstens n keer voor
\	Escape teken: voorkomt dat de shell de speciale regex tekens interpreteert en vervangt
Lin.x	De tekens 'Lin' gevuld door een willekeurig teken, gevuld door 'x' ergens in de tekst
\$[M W]ortel	Matcht met Mortel en Wortel aan het begin van een regel
ab*c	'a' gevuld door 0 of meer 'b'-s gevuld door 'c' (dus 'ac', 'abc', 'abbc', 'abbcc',...)
a(bc){2,4}	Matcht met 'abcbc', 'abcbcbc' en 'abcbcbcbc'
,[a-zA-Z0-9]\$	Matcht regels die eindigen op een komma, gevuld door een alfanumeriek karakter

Om interpretatieproblemen te voorkomen, is het aan te raden reguliere expressies altijd tussen aanhalingstekens te zetten, zo zorg je ervoor dat ze niet door de shell geïnterpreteerd worden.

Programmeerbare filters

In veel gevallen is het wenselijk tekst in bestanden te bewerken op basis van woorden die voorkomen in de regels van dat bestand. Hiervoor kan gebruik worden gemaakt van **programmeerbare filters** zoals **gawk** en **sed**.

Ook al kunnen tegenwoordig met uitermate gebruiksvriendelijke editors heel eenvoudig bewerkingen op tekstbestanden worden uitgevoerd, toch blijven deze hulpmiddelen ook nu nog hun nut bewijzen.

Vooral als je complexe shell-scripts wilt schrijven, kan het handig zijn deze hulpmiddelen te gebruiken.

SED

sed: Stream EDitor

De streameditor sed is een afgeleide van de oereditor ed.

Perl wordt vaak gezien als de opvolger van sed.

Op elke **regel** v/h bestand wordt de **filteropdracht** uitgevoerd.

Syntax: **sed** 'lijst van opdrachten' bestandsnaam

De filteropdrachten kunnen ook uit een tekstbestand genomen worden (-f optie). Gebruik de -n optie om te voorkomen dat het oorspronkelijk bestand ook uitgevoerd (getoond) wordt.

Regel selectie gebeurt met een **reguliere expressie** tussen / gevuld door een **opdracht** (bijvoorbeeld: **p** voor afdrukken, **d** voor delete, **a** voor add).

gedicht.sh ziet er als volgt uit

Pieter Paashaas

```
Daar komt Pieter Paashaas aan
voor het kippenhok blijft hij staan
hij doet alle deurtjes open
kan ik hier ook eitjes kopen?
tok, tok, tok
tok, tok, tok
alle kippetjes zijn op stok
kukeleku, kukeleku
alle eitjes zijn voor u.
```

Voorbeeld 1:

-n om te voorkomen dat het oorspronkelijk bestand ook uitgevoerd (getoond) wordt.

Alle regels afdrukken die eindigen op ok.

```
sed -n '/ok$/p' gedicht.sh
-- UITVOER --
tok, tok, tok
tok, tok, tok
alle kippetjes zijn op stok
-- --
```

Voorbeeld 2:

Verwijdert alle regels waarin het volgend patroon voorkomt: geen spatie gevolgd door een willekeurig teken en ok

```
sed '/\S.ok/d' gedicht.sh
-- UITVOER --
Pieter paashaas
```

Daar komt Pieter paashaas aan
 hij doet alle deurtjes open
 kan ik hier ook eitjes kopen?
 tok, tok, tok
 tok, tok, tok
 kukeleku, kukeleku
 alle eitjes zijn voor u.

```
-- --
```

Selectie op basis van regelnummer**Voorbeeld 1:**

Derde regel wordt afgedrukt op scherm

```
sed -n '3p' gedicht.sh
-- UITVOER --
Daar komt Pieter Paashaas aan
-- --
```

Voorbeeld 2:

Vanaf de derde tot de vijfde regel afdrukken

```
sed -n '3,5p' gedicht.sh
-- UITVOER --
Daar komt Pieter Paashaas aan
voor het kippenhok blijft hij staan
hij doet alle deurtjes open
-- --
```

Voorbeeld 3:

Regel 3 tot 5 worden verwijderd uit het gedicht

```
sed '3,5d' gedicht.sh
-- UITVOER --
Pieter Paashaas
```

kan ik hier ook eitjes kopen?
 tok, tok, tok
 tok, tok, tok
 alle kippetjes zijn op stok
 kukeleku, kukeleku
 alle eitjes zijn voor u

```
-- --
```

Substitutieopdrachten

Vervangen van tekstgedeelten

Syntax: s/patroon/vervanging/[opties]

Opties:

getal --> vervang alleen de n-de keer

g --> vervang elke keer

p --> druk elke regel af waarin vervangen werd

w bestand --> schrijf gewijzigde regels weg naar bestand

Voorbeelden:

```
nathalie@nathalie-laptop:~$ sed 's/hok/huis/' gedicht.sh
Pieter Paashaas

Daar komt Pieter Paashaas aan
voor het kippenhuis blijft hij staan
hij doet alle deurtjes open
kan ik hier ook eitjes kopen?
tok, tok, tok
tok, tok, tok
alle kippetjes zijn op stok
kukeleku, kuksukeleku
alle eitjes zijn voor u.
```

```
nathalie@nathalie-laptop:~$ sed 's/tok/boink/g' gedicht.sh
Pieter Paashaas

Daar komt Pieter Paashaas aan
voor het kippenhok blijft hij staan
hij doet alle deurtjes open
kan ik hier ook eitjes kopen?
boink, boink, boink
boink, boink, boink
alle kippetjes zijn op snoink
kuksukeleku
alle eitjes zijn voor u.
```

```
nathalie@nathalie-laptop:~$ sed -n 's/tok/boink/2p' gedicht.sh
tok, boink, tok
tok, boink, tok
nathalie@nathalie-laptop:~$ sed -n 's/tok/boink/1p' gedicht.sh
boink, tok, tok
boink, tok, tok
alle kippetjes zijn op snoink
nathalie@nathalie-laptop:~$ sed 's/ha\+s/beest/' gedicht.sh
Pieter Paasbeest

Daar komt Pieter Paasbeest aan
voor het kippenhok blijft hij staan
hij doet alle deurtjes open
kan ik hier ook eitjes kopen?
tok, tok, tok
tok, tok, tok
alle kippetjes zijn op stok
kuksukeleku
alle eitjes zijn voor u.
```

Flow Control

Het is mogelijk om bepaalde gedeelten alleen uit te voeren als aan één of meer voorwaarden zijn voldaan.

Voorwaarde	Uitleg
if/else	Instructies worden alleen uitgevoerd als wel of niet aan een bepaalde conditie wordt voldaan
case	Voert instructies uit als een variabele een bepaalde waarde heeft
for	Voert instructies een bepaald aantal keer uit
while	Voert instructies uit zolang aan een bepaalde conditie wordt voldaan
until	Voert instructies uit totdat aan een bepaalde conditie wordt voldaan

Exit-status, exit

- Wat er ook gedaan wordt met flow-control, het komt er altijd op neer dat gekeken wordt of wel of niet aan een bepaalde conditie wordt voldaan.
- Elk commando geeft een code, de **exit-status**, terug aan het proces waardoor het commando is opgeroepen. Normaliter geeft de exit-status 0 aan dat het goed gegaan is; exit-status 1 tot en met 255 worden alleen gegeven als er een fout is.
- De exit-status wordt bewaard in een shell-variabele **\$?**
- Het commando **exit** wordt gebruikt om een shell-script te beëindigen. Daarbij kan de exit-status aan het OS worden doorgegeven

Return en test

- Met het commando **return** doet men hetzelfde als exit, maar het wordt alleen gebruikt binnen een functie of een script dat is aangeroepen met het commando **source**
- Het commando **test** wordt gebruikt om een conditie te 'testen'
- Voorbeeld:** if **test** -z \$1 in bash kan dit vervangen worden door gebruik te maken van vierkante haken if [-z \$1]

```
if [ -z $1 ]
then
    echo "foutmelding"
    exit 1
fi
```



```
nathalie@nathalie-laptop:~$ bash flowif.sh
foutmelding
nathalie@nathalie-laptop:~$ echo $?
1
nathalie@nathalie-laptop:~$ bash flowif.sh a
nathalie@nathalie-laptop:~$ echo $?
0
```

- Met het commando **test** wordt een expressie geëvalueerd
 - Is de uitkomst **positief**, wordt een exit-status **0** gegeven
 - Is de uitkomst **negatief**, wordt een exit-status **1** gegeven
- Er kunnen 5 soorten tests worden uitgevoerd
 - file testers**, waarmee gekeken wordt naar de eigenschappen van een bestand
 - file comparisons**, waarmee bestanden vergeleken worden
 - file tests**, waarmee gekeken wordt naar de uitkomst van expressies
 - integer tests**, waarmee 'getallen' vergeleken kunnen worden
- Voor elke test kan ook gekeken worden of **niet** aan de voorwaarde wordt voldaan; hiervoor dient het uitroepteken (!).

FILE TESTER

Tester	Beschrijving
[-b FILE]	True als FILE bestaat en een block-special file is
[-c FILE]	True als FILE bestaat en een character-special file is
[-d FILE]	True als FILE bestaat en een directory is
[-e FILE]	True als FILE bestaat
[-s FILE]	True als FILE bestaat en een grootte heeft van meer dan 0
[-f FILE]	True als FILE bestaat en een gewone file is
[-h FILE]	True als FILE bestaat en een symbolische link is
[-r FILE]	True als FILE bestaat en leesbaar is
[-w FILE]	True als FILE bestaat en schrijfbaar is
[-x FILE]	True als FILE bestaat en uitvoerbaar is
[-O FILE]	True als FILE bestaat en het bezit is van een effectieve UserID
[-G FILE]	True als FILE bestaat en het bezit is van een effectieve groupID
[-L FILE]	True als FILE bestaat en een symbolische link is
[-N FILE]	True als FILE bestaat en gewijzigd is sinds het gelezen is
[-S FILE]	True als FILE bestaat en een socket is

FILE COMPARISONS

Tester	Beschrijving
[FILE1 -nt FILE2]	True als FILE1 recenter gewijzigd is dan FILE2 of als FILE1 bestaat en FILE2 niet
[FILE1 -ot FILE2]	True als FILE1 ouder is dan FILE2 of als FILE2 bestaat en FILE1 niet
[FILE1 -ef FILE2]	True als FILE1 en FILE2 refereren naar hetzelfde toestel en inode nummers

STRING TESTS

Tester	Beschrijving
[-z STRING]	True als de lengte van "STRING" 0 is
[-n STRING] of [STRING]	True als de lengte van "STRING" niet 0 is
[STRING1 == STRING2]	True als de strings gelijk zijn
[STRING1 != STRING2]	True als de strings niet gelijk zijn

INTEGER TESTS - operators

Tester	Beschrijving
[ARG1 OP ARG2]	"OP" is één van -eq, -ne, -lt, -le, -gt of -ge Deze arithmetische binaire operatoren geven true terug als "ARG1" gelijk is aan, niet gelijk is aan, kleiner dan of gelijk is aan, groter dan, groter dan of gelijk aan "ARG2" respectievelijk

IF THEN ELSE

Algemene vorm van **if then else** statement

```
if conditie 1
then
commandolijst1
[elif conditie2
then
commandolijst2]
[else
commandolijst3]
fi
```

Sinds Bash 2.02 is het zogenaamde "uitgebreide testcommando" ingevoerd, genoteerd als `[[...]]` (dubbele haken). Naast meer mogelijkheden is de syntax ook iets meer voor de hand liggend.

Gebruik dus zoveel `[[...]]` om vaak voorkomende fouten te vermijden.

Bijvoorbeeld: testen of een bestand leesbaar én schrijfbaar is doe je met de gewone test zo:

```
if [-r "${file}"] && [-w "${file}"]; then
echo "readable and writable"
fi
```

Je mag de logische operatoren `&&` en `||` dus niet gebruiken binnen `[]`.

Met de uitgebreide test wordt dit:

```
if [[-r "${file}" && -w "${file}"]]; then
echo "readable and writeable"
fi
```

Tip:

Gebruik zoveel mogelijk de `${} notatie` om de variabelennamen af te bakenen van de omringende tekst.

Schrijf telkens dubbele quotes rond variabelen, in het bijzonder in condities. Zoniet zal je script nooit werken met strings waar spaties in voorkomen.

```
#!/bin/bash
# ifthenelse.sh - een voorbeeld van het gebruik van "if"
# Dit script geeft een melding of $1 een bestaand bestand is

if [[ -e "${1}" ]]; then
echo "${1} bestaat"
else
echo "${1} bestaat niet"
fi
```



```
nathalie@nathalie-laptop:~$ bash ifthenelse.sh gedicht
gedicht bestaat niet
nathalie@nathalie-laptop:~$ bash ifthenelse.sh gedicht.sh
gedicht.sh bestaat
```

CASE

Algemene vorm van case statement:

```
case string in str1)
  commandoreeks1;;
str2)
  commandoreeks2;;
*)
  commandoreeks3;;
esac
```

```
#!/bin/bash
# Voorbeeld: case.sh --> case [argument]
# Voer code uit op basis van een gespecificeerd argument

case $1 in
-a)
echo u hebt optie -a ingegeven;;
-b)
echo u hebt optie -b ingegeven;;
*)
echo Fout!! U hebt een onbekende optie ingegeven
exit 1;;
esac
```



```
nathalie@nathalie-laptop:~$ bash case.sh -t
Fout!! U hebt een onbekende optie ingegeven
nathalie@nathalie-laptop:~$ bash case.sh -a
u hebt optie -a ingegeven
```

FOR

Algemene vorm van for statement:

```
for x [in list]
do
  commandoreeks
done
```

```
#!/bin/bash
# Voorbeeld: for.sh
# Druk alle directory's uit PATH onder elkaar af
IFS=:

for dir in $PATH
do
  ls -ld $dir
done
```



```
nathalie@nathalie-laptop:~$ bash for.sh
drwxr-xr-x 2 root root 4096 2009-10-28 21:55 /usr/local/sbin
drwxr-xr-x 2 root root 4096 2009-10-28 21:55 /usr/local/bin
drwxr-xr-x 2 root root 12288 2010-03-24 18:42 /usr/sbin
drwxr-xr-x 2 root root 36864 2010-03-24 18:42 /usr/bin
drwxr-xr-x 2 root root 4096 2010-01-26 12:31 /sbin
drwxr-xr-x 2 root root 4096 2010-01-26 12:30 /bin
drwxr-xr-x 2 root root 4096 2009-10-28 21:59 /usr/games
```

Generieke vorm (indien geen lijst) van for statement

```
for((startwaarde teller; stopvoorwaarde telelr; aanpassen
teller))
do
commandoreeks
done
```

WHILE/UNTIL**Algemene vorm van while of until statement:**

```
while/until conditie
do
commandoreeks
done
```

```
#!/bin/bash
# Voorbeeld: whileuntil.sh

while who | grep elvis >/dev/null
do
sleep 10
done
echo "Elvis heeft net het gebouw verlaten"

until who | grep elvis >/dev/null
do
sleep 10
done
echo "Elvis is zojuist aangekomen"
```

```
#!/bin/bash
# Voorbeeld: while.sh

count=1
while [ -n "$*" ]
do
echo "Dit is parameter nummer $count: $1"
shift
count=`expr $count + 1`
done
```



```
nathalie@nathalie-laptop:~$ bash while.sh a b c
Dit is parameter nummer 1: a
Dit is parameter nummer 2: b
Dit is parameter nummer 3: c
```

Rekenen in een script

Voorbeelden:

```
#!/bin/bash
teller=1
while true
do
    teller=$((teller + 1))
    echo teller is $teller
done
```

```
#!/bin/bash
#
#reken $1 $2 $3
#${$1} is het eerste getal
#${$2} is de operator
#${$3} is het tweede getal
x=`expr $1 + $2 + $3`
echo $x
```

Nieuwere en betere alternatieven voor "expr" zijn

- \$((...))
- ((...))
- let

Functies

Aanmaken in code als volgt:

- FUNCTIENAAM () { commando's }

Voorbeeld:

```
#!/bin/bash
# Laat het bestandstype zien
#
# Gebruik: bestandstype.sh $1
function fout
{
    echo "Je hebt een fout gemaakt!"
    echo "Voer bij gebruik van de script altijd
        de naam in van het bestand dat je wilt zien"
    exit 1
}

clear
if [ -z $1 ]; then
    fout
else
    file $1
fi
exit 0
```

- Oproepen in de code door
 - functienaam Param1 Param2...
- Waarden uit functie teruggeven
 - Exitwaarde
 - Echo "waarde"
 - Globale scriptvariabele(n) instellen

```
#!/bin/bash
product(){
    echo "$(($1 * $2))"
}
k=$(product $1 $2)
echo "$1 * $2 = $k"

```

```
#!/bin/bash
k=
product(){
    k=$(( $1 * $2 ))
}
product $1 $2
echo "$1 * $2 = $k"

```

Opties

- In een shell script wil je opties meestal op een andere manier behandelen dan gewone argumenten
- Er zijn verschillende manieren om in een shell script duidelijk te maken dat een argument een optie is
 - manueel parsen
 - met **grep** zoeken naar het koppelteken
 - met **getopts**

Manueel parsen

```
#!/bin/bash
# Optie [-a -b -c] parameters

while [ "$#" -gt "0" ]
do
    if [ "$1" = "-a" ]
    then
        echo "Dit is de eerste optie"
    elif [ "$1" = "-b" ]
    then
        echo "Dit is de tweede optie"
    elif [ "$1" = "-c" ]
    then
        echo "Dit is de derde optie"
    else
        echo "$1 is geen geldige optie"
    fi
    shift
done

echo "parameter 1 is $1"

```

Met grep zoeken naar het koppelteken

```
#!/bin/bash
# Optie [-a -b -c] parameters
# Script dat laat zien hoe gecheckt kan worden op opties.

clear
while [ -n "$(echo $1 | grep '-')" ]; do
    case $1 in
        -a) echo dit is de eerste optie;;
        -b) echo dit is de tweede optie;;
        -c) echo dit is de derde optie;;
        *) echo foute optie; exit 1;;
    esac
    shift
done
echo parameter 1 is $1 # en verdere afhandeling
```

Met getopts

```
#!/bin/bash
# getopt.sh [-a] [-b arg] [-c] args...
# Toont de werking van getopts
clear
while getopts ":ab:c" opt
do
    case $opt in
        a ) echo Je hebt optie -a gekozen;;
        b ) echo Je hebt optie -b gekozen
             echo het argument van -b is $OPTARG;;
        c ) echo Je hebt optie -c gekozen;;
        * ) echo 'optie [-a] [-b argument] [-c] args...'
             exit 1;;
    esac
done
shift $((OPTIND -1))
echo parameter 1 is $1
echo enz...
```

While getopts "":ab:c" opt -->

Het rijtje opties begint met een dubbelpunt om te voorkomen dat een onduidelijke foutmelding gegenereerd wordt als een ongeldige optie wordt gegeven. Het dubbelpunt bij optie b, duidt aan dat deze optie een argument kan hebben. Na de optie wordt de variabele opt gedefinieerd.

Het argument van optie b wordt automatisch in de var OPTARG geplaatst.

Shift \$((OPTIND -1)) --> Hier een shift uitgevoerd op basis van de waarde van de variabele OPTIND. De opdracht getopts houdt bij met welke optie hij bezig is dmv de var OPTIND, de option index. Elke keer dat getopts een optie bewerkt wordt de waarde 1 opgeteld bij de huidige waarde van OPTIND. Daardoor kan verwijzen worden naar de volgende optie die moet worden bewerkt. Dit betekent dat op het moment dat alle opties verwerkt zijn. OPTIND verwijst naar de eerste positionele parameter.

Op de waarde van de var **OPTIND** wordt vervolgens een berekening uitgevoerd, nl. OPTIND -1. Nadat alle opties zijn behandeld, is de waarde van OPTIND gelijk aan alle opties +1. Door hier weer 1 vanaf te trekken blijft dus een waarde over die precies gelijk is aan het aantal opties. Deze waarde wordt weer doorgegeven aan de opdracht shift die ervoor zorgt dat het eerste argument dat geen optie is op de plaats van de eerste positionele parameter komt te staan.

Hoofdstuk 5 – Concurrency

Wat is concurrency ?

Concurrency (ofwel parallelle processen) is bij computerprocessen een belangrijke plaats gaan innemen. Doordat het mogelijk werd enorme rekencapaciteit in een kleine chip te stoppen, zijn multiprocessors gemeengoed geworden. Dergelijke systemen kunnen vele taken gelijktijdig uitvoeren. Dit verhoogt de productiviteit, maar schept ook problemen

- **Multiprogrammering:** het beheer van meerdere processen in een systeem met 1 processor
- **Multiprocessing:** het beheer van meerdere processen in een systeem met meerdere processors
- **Gedistribueerde verwerking:** het beheer van meerdere processen die worden uitgevoerd op een aantal verspreide computersystemen.

Aan de basis van al deze zaken, en daarmee aan de basis van het ontwerp van besturingssystemen ligt **concurrency (gelijktijdig, samenlopen)**.

Concurrency hangt samen met tal van ontwerpkwesties zoals: de communicatie tussen processen, het delen van en het vechten om bronnen, de synchronisatie van meerdere procesactiviteiten en het toedelen van processortijd aan processen.

In de systemen met I/O channels (I/O-processors) zijn verscheidene acties tegelijkertijd gaande. De CPU werkt aan één proces, terwijl de I/O channels aan andere werken. Het is duidelijk dat het gebruik van meerdere processors de verwerkingscapaciteit vergroot.

Stel dat er een programmeertaal bestaat waarin je onafhankelijke processen kan specificeren, en dat er meerdere processors beschikbaar zijn om aan een proces te werken

```
PARBEGIN
statement1;
statement2;
...
statementn;
PARENDD
```

Zulke talen bestaan: Ada, Modula-2, Concurrent C, Concurrent Pascal,...

Concurrency treedt op in 3 verschillende situaties:

- **Meerdere toepassingen:** Multiprogrammering werd uitgevonden om verwerkingstijd dynamisch te kunnen verdelen tussen een aantal actieve toepassingen
- **Gestructureerde toepassing:** Als uitbreiding op de beginselen van modulair ontwerpen en gestructureerd programmeren kunnen sommige toepassingen effectief worden geprogrammeerd als een verzameling gelijktijdige processen
- **Structuur van het besturingssysteem:** Dezelfde voordelen van het structureren gelden voor de systeemprogrammeur en we hebben gezien dat besturingssystemen zelf vaak worden geïmplementeerd als een verzameling processen of threads

Wederzijdse uitsluiting (mutual exclusion)

Concurrency met meerdere processors

Niet alleen processen, maar ook activiteiten binnen één proces kunnen gelijktijdig worden uitgevoerd.

We zullen zulke processen bespreken, maar de principes gelden eveneens voor activiteiten binnen één proces. Als parallelle processen niets gemeenschappelijk gebruiken, is er geen probleem.

De moeilijkheden ontstaan wanneer de processen het gemeenschappelijke geheugen aanspreken.

Er bestaan verschillende niveaus van concurrency. Niet alleen processen, maar ook activiteiten binnen één proces kunnen gelijktijdig worden uitgevoerd. We zullen processen bespreken, maar de principes gelden eveneens voor activiteiten binnen één proces. Als parallelle processen niets gemeenschappelijk gebruiken, is er geen probleem. De moeilijkheden ontstaan wanneer de processen het gemeenschappelijke geheugen aanspreken.

Concurrency met 1 processor

Hier zijn ook parallelle processen mogelijk. In een dergelijk geval kunnen de processen 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.

Voorbeeld:

Beschouw een computersysteem met veel terminals. Stel dat de gebruikers elke regel, bestemd voor het computersysteem, beëindigen met de <enter> toets. Stel dat wij het totaal aantal lijnen voor alle gebruikers samen wensen bij te houden in een variabele "LinesEntered". Veronderstel dat twee processen proberen de variabele "LinesEntered" simultaan te verhogen met 1.

Elk proces heeft dan zijn eigen kopij van volgende code:

```
load LinesEntered
add 1
store LinesEntered
```

Dit is een vervelende situatie: **de informatie in LinesEntered is fout !** En bovendien wordt de fout veroorzaakt door een zwakke plek in het systeem dat toestaat dat twee processen op het verkeerde moment het gemeenschappelijke geheugen aanspreken. Merk op dat de twee processen het gemeenschappelijk geheugen niet op hetzelfde tijdstip benaderen. Door een wat ongelukkige timing ondermijnt het ene proces het andere. Het grootste probleem is dat bovenstaand voorbeeld 99,99% van de tijd misschien wel goed werkt. Het opsporen van dergelijke fouten kan daarom vrij moeilijk zijn.

Wederzijdse uitsluiting

--> De **kritieke sectie** van een proces is de code die naar gemeenschappelijke data verwijst

Als de uitvoering van een proces in de kritieke sectie is aangeland, moeten wij ervoor zorgen dat elk ander proces zijn eigen kritieke sectie niet betreedt. Omgekeerd moeten wij ook opletten dat een proces zijn kritieke sectie niet binnenkomt op het moment dat een ander proces in zijn kritieke sectie zit. Dit noemen we **wederzijdse uitsluiting (mutual exclusion)**.

We moeten processen gelijktijdig laten uitvoeren, en tegelijkertijd toch voorkomen dat bepaalde delen van die processen, de kritieke secties (critical sections), parallel worden verwerkt. Wanneer parallelle processen zich toegang verschaffen tot het gemeenschappelijke geheugen, bevatten hun kritieke secties de opdrachten die deze resources aanspreken. De kritieke sectie van een proces is dus de code die naar gemeenschappelijke data verwijst.

Hoe kunnen we voor wederzijdse uitsluiting zorgen ? Hoe garanderen we dat de uitsluiting er zal zijn, al vóór het proces zijn kritieke sectie heeft bereikt ? Moeten we ook iets doen als een proces aan het einde van zijn kritieke sectie is ?

Net voor de kritieke sectie van een proces wordt ENTERMUTUALEXCLUSION uitgevoerd en na de kritieke sectie wordt EXITMUTUALEXCLUSION uitgevoerd. ENTERMUTUALEXCLUSION doet het volgende:

- Controleren of een ander proces in zijn kritieke sectie is en, als dat het geval is, wachten;
- Doorgaan met de uitvoering van de kritieke sectie als er geen ander proces in de kritieke sectie bezig is. EXITMUTUALEXCLUSION moet alle andere processen vertellen dat een proces klaar is met de uitvoering van zijn kritieke sectie.

Met ENTERMUTUALEXCLUSION en EXITMUTUALEXCLUSION in een proces kunnen we van wederzijdse uitsluiting verzekerd zijn. Rest ons nog één probleem: hoe schrijven we dat? Hoe ziet dat eruit? Is het wel mogelijk zo'n code te creëren? Het schrijven van ENTERMUTUALEXCLUSION en EXITMUTUALEXCLUSION is namelijk niet zo makkelijk...

Het programmeren van wederzijdse uitsluiting

We gaan uit van slechts twee gelijktijdige processen.

Eerste poging

We declareren een **boolaanse variabele "bezet"**, die voor beide processen globaal is. "Bezet" krijgt de waarde true als één van de processen zijn kritieke sectie ingaat en is false als dit niet het geval is.

Zo kan een proces dat aan zijn kritieke sectie moet beginnen, "bezet" controleren om te zien of het andere proces in zijn kritieke sectie is.

Het "wachten" en "wekken" kan op verschillende manieren worden geïmplementeerd. Een proces kan wachten en een ander proces kan het wekken.

Er is hier een probleem omdat de processen in ENTERMUTUALEXCLUSION gemeenschappelijk geheugen aanspreken. Beide verwijzen naar "bezet". Een ongelukkige timing kan tot gevolg hebben dat het ene proces het andere ondermijnt.

Tweede poging

Eén manier om te voorkomen dat beide processen bijna gelijktijdig "bezet" controleren, is een tweede voorwaarde te gebruiken. We nemen aan dat beide processen op bijna hetzelfde moment proberen hun kritieke sectie in te gaan. Welk proces wanneer voorrang heeft, wordt geregeld door met een waarde van 1, of 2, te declareren. Beide processen moeten de waarde van "welk" controleren voordat zij hun kritieke secties ingaan. Eén proces mag doorgaan, het andere moet wachten. Zo wordt wederzijdse uitsluiting afgedwongen.

Helaas is er een ongewenst neveneffect. De twee processen kunnen niet meer onafhankelijk worden uitgevoerd. Ze moeten hun kritieke secties om beurten uitvoeren. Zo kan een proces door onbepaald uitstel worden getroffen: het moet voor onbepaalde tijd wachten. Deze oplossing brengt wederzijdse uitsluiting tot stand, maar er moet wel veel voor worden ingeleverd. Processen worden misschien helemaal niet uitgevoerd.

Derde poging

Zwakke plek in tweede poging --> Gebruik van de variabele "welk" om te bepalen welk proces zijn kritieke sectie kan ingaan. De waarde van "welk" staat dit slechts aan één proces toe, zonder rekening te houden met wat het tweede proces aan het doen is. Dit is een te zware beperking. We hadden iets nodig om ervoor te zorgen dat twee processen niet gelijktijdig hun kritieke secties ingaan, maar dit was duidelijk een verkeerde benadering

Twee processen kunnen hun kritieke secties op hetzelfde moment ingaan omdat beide een "bezetting" pas claimen nadat gecontroleerd is of er een proces met zijn kritieke sectie bezig is. Anders gezegd: **een proces controleert eerst de waarde van de globale variabele "bezett", en definieert deze dan pas als true.** Idee: verwissel deze twee opdrachten in ENTERMUTUALEXCLUSION.

Zwakke plek --> Wanneer een proces "bezet" op true zet, moet het wachten omdat "bezet" true is. Het proces maakt het zichzelf onmogelijk in zijn kritieke sectie te komen. De poging mislukt omdat het hier geen verschil maakt welk proces in zijn kritieke sectie zit. Een proces moet onderscheid kunnen maken tussen zichzelf en andere processen.

Mogelijke oplossing --> Twee globale booleaanse variabelen gebruiken "bezet1" en "bezet2". "Bezet1" is true als proces 1 in zijn kritieke sectie is en false als dit niet het geval is. "Bezet2" is true als proces 2 in zijn kritieke sectie is en false als het dat niet is. Een proces declareert dus het betreden van zijn kritieke sectie en controleert dan of het andere proces dat ook heeft gedaan.

Als proces 1 in zijn kritieke sectie is en deze vervolgens verlaat, zet het "bezet1" op false. Als proces 2 staat te wachten, wordt het hervat. Het omgekeerde vindt plaats als proces 2 zijn kritieke sectie verlaat. Elk proces kan zijn kritieke sectie dus diverse malen uitvoeren indien het ander proces inactief is.

Deze constructie zorgt voor wederzijdse uitsluiting zonder de processen te dwingen bij toerbeurt hun kritieke secties in te gaan. Alleen als het andere proces niet in zijn kritieke sectie is of bezig is daar in te gaan, kan een proces zijn kritieke sectie betreden. Wederzijdse uitsluiting is dus gegarandeerd.

Helaas is er nog een probleem dat zich kan voordoen. Stel dat beide processen tegelijk, of vrijwel op hetzelfde moment hun kritieke sectie proberen in te gaan. Het probleem is dat elk proces denkt dat het andere in zijn kritieke sectie is. Elk wacht dus tot de ander EXITMUTUALEXCLUSION heeft uitgevoerd. Omdat beide wachten, gebeurt er niets, nooit meer.

Een dergelijke situatie, waarin twee processen elk erop wachten dat de ander iets doet, noemen we een **deadlock (impasse)**. Het resultaat is dat geen van de processen verder kan. Gewoonlijk moet één proces worden afgebroken, waarbij al het verrichte werk geheel of gedeeltelijk verloren gaat. Dan moet het proces opnieuw worden gestart.

Het algoritme van Dekker

De drie pogingen tot wederzijdse uitsluiting illustreren de complexiteit van het probleem. Elk heeft een andere kwaal waar een systeem met parallelle processen kan aan lijden. Elk demonstreert ook een ander manier van denken om tot een oplossing te komen.

De Nederlandse wiskundige Dekker heeft een algoritme ontwikkeld dat wederzijdse uitsluiting zonder ongewenste neveneffecten garandeert.

Het staat in voor wederzijdse uitsluiting voor twee parallelle, asynchrone processen door ideeën uit de eerder beschreven algoritmen te gebruiken.

```
int bezet1 = 0; //false
int bezet2 = 0; //false
int welk = 1;

int main(void)
{ //main
void proces1(void);
void proces2(void);
parbegin
proces1();
proces2 ();
parend
return 0;
} //main
```

Het algoritme van Dekker komt in die zin met de derde poging overeen, dat twee booleaanse variabelen het binnengaan in een kritieke sectie aangeven. Het lijkt op de tweede poging waarin ook een globale variabele een ommezwaai in prioriteit aangeeft. Het algoritme lijkt ook op de eerste poging: elk proces controleert of een kritieke sectie is betreden voordat het probeert zijn eigen kritieke sectie in te gaan.

```

void proces1 (void)
{ //proces1
...
// begin van ENTERMUTUALEXCLUSION
bezet1 = 1; //true
while (bezet2)
if (welk == 2)
{ //if
bezet1 = 0; //false
while (welk == 2) //wacht tot welk 1 wordt
bezet1 = 1; //true
} //if
// einde van ENTERMUTUALEXCLUSION
...
// begin van EXITMUTUALEXCLUSION
welk = 2;
bezet1 = 0; //false
// einde van EXITMUTUALEXCLUSION
...
} //proces1

void proces2 (void
{ //proces2
...
// begin van ENTERMUTUALEXCLUSION
bezet2 = 1; //true
while (bezet1)
if (welk == 1)
{ //if
bezet2 = 0; //false
while (welk == 1); //wacht tot welk 2 wordt
bezet2 = 1; //true
}
// einde van ENTERMUTUALEXCLUSION
...
// kritieke sectie van proces2
...

```

```
// begin van EXITMUTUALEXCLUSION
welk = 1;
bezett2 = 0; //false
// einde van EXITMUTUALEXCLUSION
...
}
```

De logica achter het algoritme van Dekker is de volgende. Voordat een proces in zijn kritieke sectie gaat, moet het:

1. Zijn bezettingsvariabele op true zetten. Dit betekent dat het probeert zijn kritieke sectie in te gaan
2. Controleren of het andere proces in zijn kritieke sectie is of probeert daarin te komen. Is dat niet het geval: kritieke sectie ingaan. Anders: verder gaan met de volgende stap
3. Wachten als het andere proces aan de beurt is om zijn kritieke sectie uit te voeren. De bezettingsvariabele op false zetten en wachten tot het ander proces zijn kritieke sectie verlaat.
4. In het geval dat het huidige proces aan de beurt is om zijn kritieke sectie uit te voeren - als het ander proces toch in zijn kritieke sectie is: wachten tot het deze verlaat. Probeerdaarentegen het andere proces ook zijn kritieke sectie in te gaan, dan moet het wachten, zodra het ontdekt dat het huidige proces aan de beurt is. In die situatie: de kritieke sectie ingaan.

Er zijn twee grote verschillen tussen deze oplossing en de vorige

1. De booleaanse variabelen geven niet aan of een proces daadwerkelijk in zijn kritieke sectie is; ze maken slechts kenbaar dat een proces dat wil gaan doen
2. De voorrang niet streng wordt voorgeschreven, tenzij beide processen op vrijwel hetzelfde moment proberen hun kritieke secties in te gaan.

Het algoritme van Peterson

Biedt een eenvoudigere elegante oplossing

```
boolean flag [2];
int turn;
void P0()
{
    while (true)
    {
        flag [0] = true;
        turn = 1;
        while (flag [1] && turn == 1)
        /* do noting */;
        /* critical section*/;
        flag [0] = false;
        /* remainder */
    }
}
void P1 ()
{
    while (true)
    {
        flag [1] = true;
        turn = 0;
        while (flag [0] && turn == 0)
        /* do nothing */;
        /* critical section*/;
        flag [1] = false;
        /* remainder */
    }
}
void main()
{
    flag [0] = false;
    flag [1] = false;
    parbegin (P0, P1);
}
```

Bespreking algoritme:

Het algoritme van Dekker lost het probleem van wederzijdse uitsluiting op, maar gebruikt daarvoor een nogal complex programma, dat moeilijk te volgen is en waarvan de juistheid lastig is te bewijzen.

De globale variabele **flag** duidt de positie van elk proces ten aanzien van wederzijdse uitsluiting aan.

De globale variabele **turn** lost de conflicten van gelijktijdigheid op.

De wederzijdse uitsluiting blijft hier behouden --> wanneer P0 de flag[0] op true instelt, kan P1 zijn kritieke sectie niet uitvoeren. Bevindt P1 zich al in zijn kritieke sectie, dan geldt dat flag[1] = true en is de uitvoering van de kritieke sectie van P0 geblokkeerd.

Een wederzijdse blokkering wordt echter voorkomen.

Veronderstel dat P0 is geblokkeerd in zijn while lus. Dit betekent dat flag[1] true is en dat turn = 1. P0 kan zijn kritieke sectie uitvoeren als flag[1] false wordt of turn 0 wordt.

Beschouw nu 3 uitputtende gevallen:

1. P1 heeft geen interesse in zijn kritieke sectie. Dit geval is onmogelijk, omdat het impliceert dat flag[1] = false.
2. P1 wacht op zijn kritieke sectie. Dit geval is ook onmogelijk, omdat als turn = 1, P1 zijn kritieke sectie kan uitvoeren.
3. P1 gebruikt zijn kritieke sectie herhaaldelijk en monopoliseert daarmee de toegang. Dit kan niet gebeuren, omdat P1 aan P0 een kans moet geven door turn in te stellen op 0 voordat P1 probeert zijn kritieke sectie uit te voeren.

Wederzijdse uitsluiting bij n processen

Het algoritme van Dekker is niet gemakkelijk op meer dan twee processen toe te passen. Daarvoor hebben we dus een algoritme nodig zoals bijvoorbeeld het algoritme van Peterson.

Er zijn er vele. Sommige garanderen dat een proces dat zijn kritieke sectie probeert uit te voeren, nooit erg lang wordt uitgesteld. Door hun complexiteit zijn deze algoritmen heel moeilijk in de praktijk toe te passen.

Semaforen

Inleiding

In 1965 introduceerde Dijkstra het begrip semafoor om wederzijdse uitsluiting tussen processen af te dwingen. Hij definiereerde een semafoor als **een integer-variabele die door slechts 2 primitieve operaties kan worden veranderd**.

Een primitieve kan niet worden onderbroken; eenmaal begonnen, kan het proces tot het klaar is niet worden onderbroken of opgeschort. Primitieve operaties zijn afhankelijk van het systeemontwerp, daarom moet er bij het ontwerpen van computersystemen al rekening mee worden gehouden.

Ter info: De semafoor of optische telegraaf was het eerste, bruikbare middel voor optische telecommunicatie. Het woord semafoor is een samenstelling van de Griekse woorden teken en dragen.

De primitieve operaties voor een semafoor zijn P en V, en worden als volgt gedefinieerd. Als S een semafoor is, dan is

P(S): if ($S > 0$)

$S = S - 1;$

else (proces uitstellen)

V(S): if (een proces uitgesteld is als gevolg van P(S))

(hervat een proces)

else $S = S+1;$

Een proces dat de primitieve P uitvoert, moet misschien wachten (WAIT).

Een proces dat de primitieve V uitvoert, geeft misschien het signaal dat een ander proces kan worden hervat. Het feit dat P en V niet onderbroken kunnen worden is essentieel. Een eenmaal begonnen operatie eindigt zonder onderbreking.

P --> Prolaag. "Prolaag" is een woord dat Dijkstra heeft bedacht en het betekent "probeer te verlagen".

V --> Verhoog

Het algoritme wordt dan:

```
int S = 1; //semafoor
int main(void)
{
//main
void proces1 (void);
void proces2 (void);
parbegin
proces1();
proces2();
parend
return 0;
} //main
void proces1 (void)
{
//proces1
...
//begin van ENTERMUTUALEXCLUSION
P(S);
//einde van ENTERMUTUALEXCLUSION
...
// kritieke sectie van proces1
...
// begin van EXITMUTUALEXCLUSION
V(S);
//einde van EXITMUTUALEXCLUSION
...
} //proces1
void proces2 (void)
{
//proces2
...
//begin van ENTERMUTUALEXCLUSION
P(S);
//einde van ENTERMUTUALEXCLUSION
...
//kritieke sectie van proces2
...
//begin van EXITMUTUALEXCLUSION
V(S);
//einde van EXITMUTUALEXCLUSION
...
```

```
} //proces2
```

Naast de eenvoud en de elegantie van semaforen hebben ze nog een ander belangrijk voordeel. Het algoritme kan gemakkelijk worden uitgebreid voor een situatie geval met n parallelle processen. Als 1 proces P(S) uitvoert, zijn alle andere gedwongen te wachten.

Sterke semaforen

Een **semafoor** is dus een onderdeel van een synchronisatiemechanisme voor parallelle of gedistribueerde programma's.

Het grondbeginsel luidt als volgt:

- Twee of meer processen kunnen samenwerken d.m.v. eenvoudige signalen, waarbij een proces kan worden gedwongen te stoppen op een opgegeven plaats totdat het een specifiek signaal heeft ontvangen.
- Voor het signaleren worden speciale variabelen gebruikt, zogenoemde **semaforen**

Men kan aan elke coördinatie-eis, hoe complex ook, voldoen door de keuze van de juiste signaalstructuur.

De eerste grote sprong voorwaarts bij het oplossen van problemen van gelijktijdige processen was het proefschrift van Dijkstra. Dijkstra hield zich bezig met het ontwerpen van een besturingssysteem als een verzameling samenwerkende, sequentiële processen en met het ontwikkelen van efficiënte en betrouwbare mechanisme voor het ondersteunen van de samenwerking. Deze mechanismen kunnen even gemakkelijk ook worden gebruikt door gebruiksprocessen als de processor en het besturingssysteem de mechanisme beschikbaar maken.

Het grondbeginsel luidt als volgt. Twee of meer processen kunnen samenwerken d.m.v. eenvoudige signalen, waarbij een proces kan worden gedwongen te stoppen op een opgegeven plaats totdat het een specifiek signaal heeft ontvangen. Voor het signaleren worden speciale variabelen gebruikt, zogenoemde semaforen. Men kan aan elke coördinatie-eis, hoe complex ook, voldoen door de keuze van de juiste signaalstructuur.

- Voor het verzenden van een signaal via semafoor s voert een proces de primitieve **signal(s)** uit.
- Voor het ontvangen van een signaal via semafoor s voert een proces de primitieve **wait(s) uit**; is het corresponderende signaal nog niet verzonden, dan wordt het proces onderbroken totdat het versturen ervan plaatsvindt.

Om het gewenste effect te bereiken kunnen we de semafoor beschouwen als een variabele die een gehele waarde heeft en waarvoor **3 bewerkingen zijn gedefinieerd**.

1. Een semafoor kan worden **geinitialiseerd op een niet-negatieve waarde**
2. De bewerking **wait** verlaagt de semafoorwaarde. Wordt de waarde negatief, dan wordt het proces dat de opdracht wait uitvoert, geblokkeerd.
3. De bewerking **signal** verhoogt de semafoorwaarde. Is de waarde niet positief, dan wordt een proces dat is geblokkeerd door een bewerking wait gedeblokkeerd.

Er bestaat geen mogelijkheid, anders dan deze 3 bewerkingen, om semaforen te inspecteren of te bewerken.

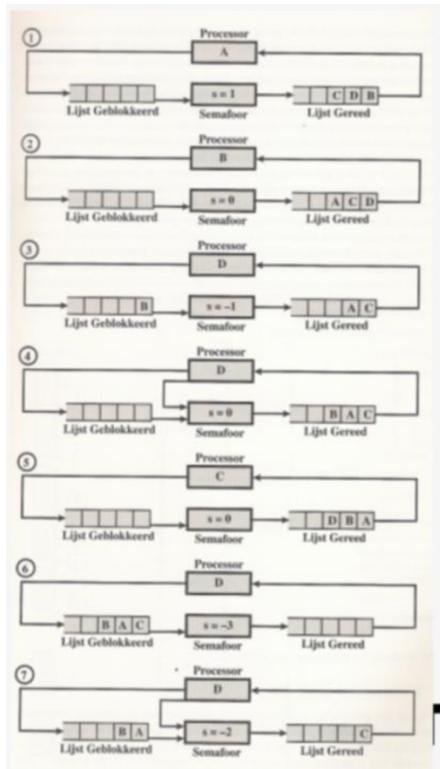
Formele definitie van primitieven voor semaforen:

```
struct semaphore {
    int count;
    queueType queue;
}
void wait(semaphore s)
{
    s.count--;
    if (s.count < 0)
    {
        place this process in s.queue;
        block this process;
    }
}
void signal(semaphore s)
{
    s.count++;
    if (s.count <= 0)
    {
        remove a process P from s.queue;
        place process P on ready list;
    }
}
```

Bevat de definitie van een semafoor deze FIFO)strategie, dan wordt dit een **sterke** semafoon genoemd.

Als niet is vastgelegd in welke volgorde processen uit de wachtrij worden verwijderd, is er sprake van een **zwakke semafoon**.

Voorbeeld van een sterke semafoon:



Voorbeeld van een FIFO operatie:

Hierbij zijn de processen A, B en C afhankelijk van een resultaat van proces D.

In het begin (1) is A in uitvoering: B, C en D staan gereed; de semafoorteller is gelijk aan 1, wat aangeeft dat 1 resultaat van D beschikbaar is. Wanneer A een wait uitvoert, passeert het onmiddellijk de semafoon en kan doorgaan met de uitvoering; vervolgens kan het weer aansluiten in de rij gereed. **Vervolgens gaat B in uitvoering (2)**, voert ook een wait uit en wordt geblokkeerd; **dit geeft D de mogelijkheid voor uitvoering (3)**. Als D voor een nieuw resultaat heeft gezorgd, geeft het een signaal dat het mogelijk maakt dat **B verplaatst wordt naar de rij gereed (4)**. D sluit weer aan bij de rij gereed en **C komt in uitvoering (5)**, maar **raakt geblokkeerd** bij het geven van een wait. Op gelijke manier komen ook A en B in uitvoering en raken geblokkeerd op de semafoon, wat **D de gelegenheid geeft om verder te gaan met zijn uitvoering (6)**. Wanneer D een resultaat heeft bereikt, geeft het een signaal zodat C verplaatst naar de rij gereed. Bij volgende optredens van D komen ook A en B uit de wachtrij van geblokkeerde processen.

Het algoritme van wederzijdse uitsluiting

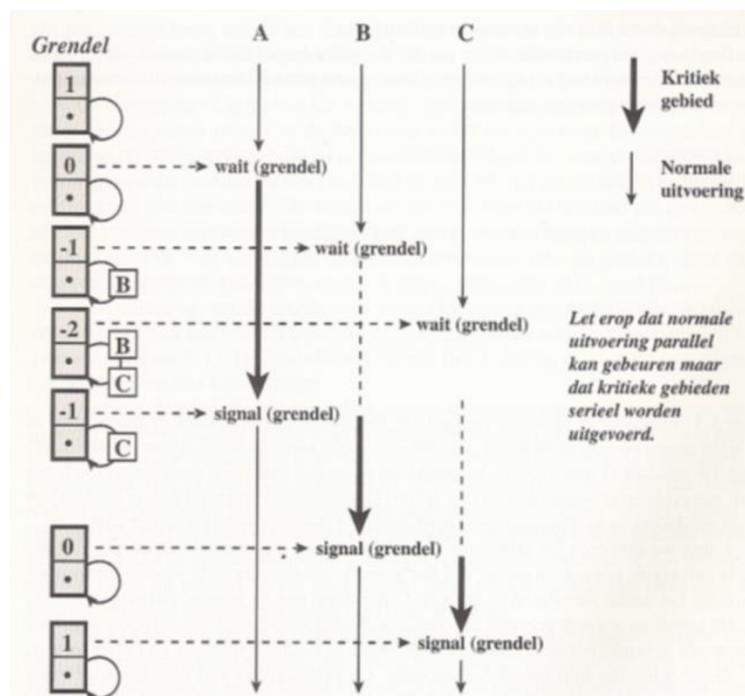
```
/* program mutual exclusion */
const int n = /* number of processes */;
semaphore s = 1;
void P(int i)
{
while(true)
{
wait(s);
/* critical section */;
signal(s);
/* remainder */
}
}
void main()
{
parbegin (P(1), P(2), ..., P(n));
}
```

We zullen uitgaan van sterke semaforen omdat deze gemakkelijker zijn en omdat meestal besturingssystemen deze vorm van semaforen bieden.

Dit toont een eenvoudige oplossing van het probleem van wederzijdse uitsluiting met een semafoor s . Er zijn n processen, die worden aangegeven in de array $P(i)$. In elk proces wordt een $\text{wait}(s)$ uitgevoerd vlak voor de kritieke sectie. Wordt de waarde s negatief, dan wordt het proces opgeschorst. Is de waarde 1, dan wordt deze verlaagd tot 0 en voert het proces onmiddellijk zijn kritieke sectie uit. Omdat s niet meer positief is, kan geen enkel ander proces zijn kritieke sectie uitvoeren. De semafoor wordt geïnitialiseerd op 1. Daardoor kan het eerste proces dat een wait uitvoert, onmiddellijk zijn kritieke sectie binnengaan, waarbij de waarde van s gebracht wordt op 0. Elk proces dat probeert zijn kritieke sectie te betreden, merkt dat deze bezet is en wordt geblokkeerd, waarbij de waarde van s met 1 vermindert. Nog meer processen kunnen proberen toegang te krijgen: al deze pogingen leiden tot een verdere daling van de waarde van s .

Verlaat het proces dat zijn kritieke sectie het eerst heeft gestart, de kritieke sectie, dan wordt s verhoogd en wordt een van de geblokkeerde processen die geblokkeerd zijn voor de semafoor en in de toestand gereed geplaatst. Wordt het daarna ingeroosterd door het besturingssysteem, dan kan het zijn kritieke sectie uitvoeren.

Voorbeeld:



In dit voorbeeld benaderen 3 processen A, B en C een gedeelde bron die wordt bewaakt door de semafoor grendel. Proces A voert een wait(grendel) uit. Omdat de semafoorwaarde 1 is op het moment van de wait, kan A direct de kritieke sectie binnengaan en de semafoor krijgt de waarde 0. Terwijl A binnen zijn kritieke sectie zit, voeren B en C beiden een wait uit en worden geblokkeerd in afwachting van de beschikbaarheid van de semafoor. Wanneer A zijn kritieke sectie verlaat en signal (grendel) uitvoert, mag B zijn kritieke sectie binnengaan want B stond vooraan in de wachtrij.

Implementatie semaforen:

1e mogelijkheid: Implementeren in hardware of firmware

2e mogelijkheid: Softwarebenadering zoals algoritme van Dekker of Peterson => leidt tot een aanzienlijke overhead in de verwerking

3e mogelijkheid: Het gebruiken van een in hardware ondersteund mechanisme voor wederzijdse uitsluiting zoals bijvoorbeeld het gebruik van een instructie test and set waarbij de semafoor weer een datastructuur heeft en een nieuwe integer als component, s.flag bevat.

4e mogelijkheid: Bij een systeem met 1 processor is het mogelijk interrupts te verbieden tijdens de bewerking wait en signal

Zoals eerder is gezegd, is het essentieel dat de bewerkingen wait en signal worden geïmplementeerd als atomaire primitieven. Een mogelijkheid die voor de hand ligt, is ze te implementeren in hardware of firmware. Is dat niet mogelijk, dan zijn er een aantal alternatieven. De kern van het probleem is de wederzijdse uitsluiting: slechts 1 proces tegelijk mag een semafoor wijzigen met de bewerking wait of signal. Daarvoor zou elk van de softwarebenaderingen kunnen worden gebruikt, bijvoorbeeld het algoritme van Dekker of het algoritme van Peterson; dit zou leiden tot een aanzienlijke overhead in de verwerking. Een ander alternatief is het gebruiken van een in hardware ondersteund mechanisme voor wederzijdse uitsluiting.

Monitoren

Semaforen zijn een primitief maar krachtig en flexibel gereedschap voor het afdwingen van wederzijdse uitsluiting en voor het coördineren van processen. Het kan echter moeilijk zijn een correct programma te maken met semaforen. De moeilijkheid is dat de bewerkingen **wait** en **signal** verspreid kunnen zijn over het programma en het is lastig te zien welke algehele invloed bewerkingen hebben op de betreffende semaforen.

De monitor is een constructie in een programmeertaal die een functionaliteit biedt die vergelijkbaar is met die van semaforen, maar die gemakkelijker te besturen is.

Om problemen te vermijden moet wederzijdse uitsluiting echter verplicht zijn. Oplossing hiervoor is: de kritieke secties in een gebied te plaatsen waartoe maar 1 proces tegelijk toegang heeft. Dan gebruiken de processen de code op een manier die automatisch wederzijdse uitsluiting afdwingt. Voor de speciale gebieden gebruiken we de term **monitors**.

Een monitor is een constructie die code kan bevatten die naar gemeenschappelijke gebruikte data verwijst. Oppervlakkig gezien lijkt het een verzameling datatypen, datastructuren en procedures, onder de monitor heading. Maar een monitor is veel meer.

Hij bevat procedures en variabelen, maar de procedures zijn van een bijzonder type. Als parallelle processen verschillende procedures in een monitor aanroepen, dwingt de monitor de processen deze procedures na elkaar uit te voeren.

2 procedures binnen dezelfde monitor kunnen niet tegelijk actief zijn. Door de taal gedefinieerde **aanroep-protocollen** (calling protocollen) dwingen dit automatisch af.

Een kritieke sectie kunnen we daarom i.p.v. deze in een proces te coderen als **monitor-procedure** schrijven. De code wordt dan niet geduplicateerd. Wanneer een proces gemeenschappelijke data moet gebruiken, roept het een monitor-procedure aan.

Door een compiler gegenereerde code, die de besturing aan een monitor-procedure overdraagt, wordt wederzijdse uitsluiting gegarandeerd.

Op deze manier is er een aanzienlijk verschil tussen een monitor en een eenvoudige collectie procedures. Een monitor dwingt heel streng wederzijdse uitsluiting af tussen processen die proberen zijn procedures uit te voeren. Om dit verschil te benadrukken wordt een monitor-procedure een **procedure-entry** genoemd.

Via conditionele variabelen kan de monitor processen uitstellen of hervatten om events te synchroniseren.

Monitoren werken het best wanneer ze centraal geïnstalleerd kunnen worden. Omdat alle processen de monitor aanspreken, is deze het middelpunt van alle discussies en analyses. Maar vele systemen hebben geen centrale component.

De monitorconstructie is geïmplementeerd in enkele programmeertalen waaronder Modula-3, Java,... Ook is ze geïmplementeerd als een programmabibliotheek. Dit biedt de mogelijkheid grendels op elk object te plaatsen. Vooral bij zoiets als een verbonden lijst kan het zinvol zijn alle verbonden lijsten te vergrendelen met 1 grendel, 1 grendel te gebruiken voor elke lijst of 1 grendel te gebruiken voor elk element van elke lijst.

Een monitor kunnen we dus kort definiëren **als een constructie in een programmeertaal die voorziet in abstracte gegevenstypen en toegang, met wederzijdse uitsluiting, tot een aantal procedures.**

Synchronisatie

Het gemak waarmee semaforen wederzijdse uitsluiting afdwingen, maakt het ons mogelijk ook andere problemen, bijvoorbeeld het **synchroniseren van processen**, op te lossen.

We definiëren dit als het opleggen van een dwingende volgorde aan events die door concurrente, asynchrone processen worden uitgevoerd.

Wij moeten garanderen dat processen in een bepaalde volgorde verlopen.

Illustratie van de problematiek van synchroniseren a.d.h.v. het filosofen probleem:

De vijf filosofen zitten aan een tafel en kunnen twee dingen doen: spaghetti eten of filosoferen. Als ze eten kunnen ze niet denken en als ze denken kunnen ze niet eten. De spaghetti staat midden op de ronde tafel en 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, maar alleen een voor een.

Het probleem is nu om de filosofen zodanige instructies te geven dat ze niet zullen verhongeren.

Dit soort problemen zijn in het algemeen niet zo eenvoudig op te lossen.

Stel bijvoorbeeld dat elke denker als filosofie heeft: ik pak een vork zo gauw ik kan, als beide beschikbaar zijn eerst de linkervork; zo gauw ik beide vorken heb eet ik wat; dan leg ik de vorken weer neer.

Op het eerste gezicht een redelijk plan, maar nu kan de situatie ontstaan dat elke filosoof de linkervork in de linkerhand heeft, eeuwig wachtend tot de rechtermuur vrijkomt. Dit is een voorbeeld van '**deadlock**': er is helemaal geen voortgang in het systeem meer mogelijk. Elke filosoof zal verhongeren.

Er zijn technieken om tot oplossingen te komen die deadlock bewijsbaar voorkomen; Dijkstra heeft het probleem verzonden om zulke technieken te demonstreren.

We kunnen bijvoorbeeld de denkers nummeren en elke denker alleen een vork laten pakken als er geen hoger genummerde denker al een vork vastheeft. Nu is deadlock onmogelijk.

Deadlock is echter niet het enige soort situatie dat in het ontwerp moet worden uitgesloten.

Stel bijvoorbeeld dat we een denker zelfs geen vork laten pakken als tegelijk een hoger genummerde hetzelfde probeert. Dan zal de hoogstgenummerde altijd eten, terwijl de rest verhongert. Zo'n situatie wordt **starvation** genoemd.

We kunnen dit nog verder aanscherpen, bijvoorbeeld door te eisen dat het systeem **eerlijk** is, in de zin dat de filosofen niet alleen allemaal altijd nog ooit de kans krijgen te eten, maar ze die kans zelfs even vaak krijgen; of door te eisen dat de totale wachttijd zo klein mogelijk is.

Deze situatie illustreert de problemen die zich kunnen voordoen bij het synchroniseren van toegang tot resources (de vorken), bijvoorbeeld door verschillende threads (de filosofen) in een computerprogramma.

Als verschillende threads gebruik maken van dezelfde variabelen of bestanden is het niet veilig dat ze die tegelijk proberen aan te passen; daarom kan het onvermijdelijk zijn dat threads op elkaar moeten wachten. Als deze synchronisatie niet correct wordt ontworpen kan het voorkomen dat een thread helemaal nooit meer aan de beurt komt (**starvation**) of dat dat zelfs voor elke thread geldt (**deadlock**).

Deadlocks

Een deadlock of een impassestoestand treedt op wanneer 2 of meer processen voor onbepaalde tijdwachten op een gebeurtenis die alleen door 1 van de wachtende processen kan worden veroorzaakt.

In principe zijn **er 2 methoden voor het behandelen van deadlock:**

- Gebruik één of ander protocol (afspraak) om te garanderen dat het systeem nooit in een deadlock-situatie zal komen;
- Laat toe dat het systeem in een deadlock-situatie geraakt en los deze dan op

Voornaamste aspecten van deadlock

- **Deadlock-preventie**
 - Het besturingssysteem beperkt het gemeenschappelijk gebruik van resources om deadlock onmogelijk te maken
- **Deadlock-vermijding**
 - Het besturingssysteem onderzoekt alle aanvragen voor resources heel nauwkeurig. Ziet het besturingssysteem dat de toewijzing van een resource het risico van deadlock met zich meebrengt, dan weigert het de gevraagde toegang en vermijdt zo het probleem.
- **Deadlock-signalering**
 - Als er een deadlock optreedt, moet het besturingssysteem dit kunnen signaleren. Het besturingssysteem ziet elk proces in een wachttoestand. Hoe kan het besturingssysteem erachter komen dat dit wachten permanent is ?
- **Deadlock-herstel**
 - Wat moet er gebeuren nadat het besturingssysteem een deadlock ontdekt ? De processen moeten daar toch een keer uit bevrijd worden. Het besturingssysteem moet dit probleem oplossen

Deadlock-preventie

Een deadlock kan alleen dan optreden indien er tegelijkertijd aan de volgende 4 voorwaarden is voldaan

1. **Wederzijdse uitsluiting** (mutual exclusion)
2. **Bezet houden en wachten** (hold and request)
3. **Geen voortijdig ontnemen** (non-preemption)
4. **Wachten in een kring** (circular wait)

Om deadlock te voorkomen zorgen we ervoor dat tenminste 1 van de voorwaarden nooit optreedt

Er ontstaan aanzienlijke problemen als we proberen deadlock te voorkomen door een noodzakelijke conditie op te heffen.

Wederzijdse uitsluiting:

Gemeenschappelijk gebruik van de resources moet onder wederzijdse uitsluiting plaatsvinden, d.w.z. als een proces tot een resource toegang heeft, mag geen enkel ander proces deze benaderen tot de resource is vrijgegeven.

Bezet houden en wachten:

Een proces kan meerdere resources aanvragen zonder de eerder toegewezen resources vrij te geven. Er moet dus een proces bestaan dat ten minste 1 resource bezet houdt en dat tevens wacht op het verkrijgen van nog andere resources die op dat ogenblik door andere processen bezet zijn.

Geen voortijdig ontnemen

Resources kunnen niet voortijdig worden afgenomen d.w.z. dat een resource alleen vrijwillig kan worden vrijgegeven door het proces dat deze resource in bezit heeft, nadat het proces zijn taak heeft beëindigd.

Wachten in een kring

Er moet een verzameling $\{p_0, p_1, \dots, p_n\}$ van wachtende processen bestaan d.w.z. dat een resource alleen vrijwillig kan worden vrijgegeven door het proces dat deze resource in bezit heeft, nadat het proces zijn taak heeft beëindigd.

Problemen:

- Als er geen wederzijdse uitsluiting wordt afgedwongen, kunnen de activiteiten van het ene proces de voortgang van het andere proces beïnvloeden. Deze conditie mag dus niet worden verwijderd.
- Alvorens met zijn uitvoering te beginnen moet elk proces al de resources, die het nodig heeft, verkrijgen. Als een proces alle resources voor lange tijd onder beheer zondert ze daadwerkelijk te gebruiken. Dit beperkt de beschikbaarheid van de resources.
- Als het proces enkele resources vasthoudt en het vraagt nog een resource aan, en deze resource kan niet onmiddellijk aan dat proces worden toegewezen (d.w.z. het proces moet wachten), dan moet het proces al de resources die het op dat ogenblik vasthoudt, vrijgeven. Als we deze conditie verwijderen, dan kan een resource met geweld van een proces ontnomen worden.
- Onderwerp alle processen aan een lineair ordeningsschema. Ieder proces kan alleen resources in opklimmende volgorde verkrijgen

Deadlock vermijden

Het verschil tussen het vermijden en het voorkomen van een deadlock is dat in het eerste geval deadlock niet onmogelijk is. Het idee is de aanvragen die eventueel tot deadlock kunnen leiden, te weigeren.

Deadlock signaleren

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

Eén manier om deadlock te signaleren, is een resource allocation graf. Dit is een georiënteerde graf die gebruikt wordt om de resource-toewijzingen weer te geven.

Een deadlock kunnen we signaleren door de resource allocation graf te bekijken. Als deze een cyclus bevat, is er een deadlock. Om cycli in een georiënteerde graf te signaleren, heeft het besturingssysteem diverse algoritmen ter beschikking.

Herstel in een deadlock-situatie

Nu we weten hoe we een deadlock signaleren, rest ons nog 1 vraag: wat doen we eraan?

Eén mogelijkheid is een proces gewoon maar af te breken en de eraan toegewezen resources verwijderen. 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 resources verwijderd. Het proces verliest alle updates die het met gebruik van deze resources 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 resources. Dit kan overeenkomen met de oorspronkelijke start van het proces, of met een checkpoint. Een checkpoint treedt op wanneer een proces vrijwillig alle resources vrijgeeft. Door het gebruik van checkpoints kan elk proces eventueel verlies van werk echter zo klein mogelijk houden.

Threads

Een proces bestaat uit 2 afzonderlijke en mogelijk onafhankelijke concepten: een concept dat samenhangt met de eigendom van bronnen en een concept dat samenhangt met de uitvoering. Dit onderscheid heeft in enkele besturingssystemen geleid tot de ontwikkeling van een constructie die bekendstaat als een **thread**.

Om een onderscheid te maken tussen de 2 concepten, wordt de eenheid voor de verdeling (uitvoering) doorgaans een thread of lichtgewicht proces genoemd en de eenheid voor de eigendom van bronnen een proces of een taak.

Multithreading verwijst naar de mogelijkheid van een besturingssysteem binnen een proces meerdere threads of draden te gebruiken voor de uitvoering. De traditionele benadering met één uitvoeringsthread per proces, waarin het concept thread in feite niet bestaat, wordt ook wel een benadering met één thread genoemd.

In een omgeving met multithreading wordt een proces gedefinieerd als beveiligings- en brontoewijzingseenheid.

Het volgende is verbonden met processen:

- Een **virtuele adresruimte**, die het procesbeeld bevat
- **Beveiligde toegang** tot processors, andere processen (voor de communicatie tussen processen), bestanden en I/O bronnen (apparaten en kanalen)

Binnen een proces kunnen er een of meer threads zijn, elk met het volgende (eigenschappen van threads):

- Een **uitvoeringstoestand** van de thread (actief, gereed,...)
- Een **context** die wordt opgeslagen als de thread niet actief is, een thread kan onder meer worden gezien als een onafhankelijke programmateller die binnen een proces werkt
- **Een stack** voor de uitvoering
- **Enige statische opslagcapaciteit per thread** voor lokale variabelen
- Toegang tot het geheugen en de bronnen van het bijhorende proces, die wordt gedeeld door alle threads binnen dat proces

Alle threads van een proces delen de toestand en de bronnen van dat proces. Ze bevinden zich in dezelfde adresruimte en hebben toegang tot dezelfde gegevens.

De grootste voordelen van threads hangen samen met de gevolgen voor de prestaties: het creëren van een nieuwe thread binnen een bestaand proces kost veel minder tijd dan het creëren van een geheel nieuw proces en ditzelfde geldt voor het overschakelen tussen 2 threads binnen hetzelfde proces.

Threads verbeteren ook de efficiëntie van de communicatie tussen verschillende actieve programma's. Aangezien threads binnen hetzelfde proces echter geheugen en bestanden delen, kunnen ze rechtstreeks met elkaar communiceren.

Net zoals processen hebben threads uitvoeringstoestanden en kunnen ze met elkaar worden gesynchroniseerd.

BS Theorie

Geef de definitie van een besturingssysteem

In hoofdzaak is een besturingssysteem een programma dat mensen de mogelijkheid geeft gebruik te maken van de hardware van een computer (CPU's, geheugen en secundaire opslagapparatuur). Gebruikers geven geen instructies aan de computer, maar in plaats daarvan aan het besturingssysteem. Het besturingssysteem geeft de hardware de opdracht de gewenste taken uit te voeren.

Geef de 7 taken van een besturingssysteem

1. Programma's de mogelijkheid geven informatie op te slaan en terug te halen
2. Programma's afschermen van specifieke hardware zaken
3. De gegevensstroom door de componenten van de computer regelen
4. Programma's in staat stellen te werken zonder door andere programma's te worden onderbroken
5. Onafhankelijke programma's de gelegenheid geven tijdelijk samen te werken en informatie gemeenschappelijk te gebruiken
6. Reageren op fouten of aanvragen van de gebruiker
7. Een tijdsplanning maken voor programma's die resources willen gebruiken

Hoe werkten besturingssystemen in de jaren 50 ?

Programma's werden na elkaar in de computer ingevoerd en opgeslagen. Wanneer het ene programma klaar was, laadde en startte het besturingssysteem het volgende programma. Alle resources van de computer konden slechts door één programma gebruikt worden.

Wat waren de verbeteringen in de jaren 60 ?

Verscheidene programma's konden tegelijkertijd in het geheugen worden opgeslagen. De programma's gebruikten de resources van de computer dan gemeenschappelijk. De programma's werden nu niet na elkaar, maar om beurten uitgevoerd. Elk programma liep een tijdje en dan schakelde het besturingssysteem naar een ander programma over. Hierdoor kwamen de korte programma's eerder aan de beurt, en omdat de resources gemeenschappelijk werden gebruikt, waren ze sneller afgelopen. Een gebruiker kon vanaf een terminal inloggen en vrijwel direct toegang tot de resources krijgen. Daarnaast kon 1 besturingssysteem voor verschillende computers van hetzelfde type worden gebruikt, zodat een upgrade eenvoudiger werd.

Welke grote verandering vond plaats in de jaren 70 ?

Besturingssystemen konden computers met meer dan één processor aan.

Wat gebeurde er in de jaren 80 ?

Besturingssystemen werden geschikt voor gemeenschappelijk gebruik van informatie door verschillende computersystemen.

Wat kenmerkt de jaren 90 ?

Begrippen als distributed computing, parallelle verwerking, GUI

VRAGENSTELLING BESTURINGSSYSTEMEN THEORIE

Welke 3 soorten besturingssystemen bestaan er ?

1. Single-tasking
2. Multitasking
3. Multi-user systemen

Wat is Single-tasking ?

Een systeem waarin één gebruiker één applicatie tegelijk draait. Onder dit systeem kan slechts één programmat tegelijk actief zijn.

Wat is multitasking ?

Meestal 1 gebruiker die verscheidene taken kan uitvoeren tezelfdertijd.

Veel van de multitasking systemen staan nog steeds maar één gebruiker toe, maar hij of zij kan verscheidene bezigheden op hetzelfde moment laten afwikkelen. Aangezien een gebruiker verscheidene werkzaamheden gelijktijdig kan laten verrichten, worden bepaalde functies van het besturingssysteem, bijvoorbeeld geheugenbeheer, ingewikkelder.

Wat zijn multi-user systemen ?

Bij multi-user systemen maken meerdere gebruikers simultaan gebruik van de computerresources.

Multiuser-systemen, ook wel multiprogrammeringssystemen genoemd, moeten niet alleen alle gebruikers bijhouden, er moet ook voorkomen worden dat deze elkaar hinderen of in het werk van de ander rondneuzen.

Hier is scheduling een belangrijk concept. Scheduling verwijst naar de manier waarop processen prioriteiten worden gegeven in een prioriteitenwachtrij.

In een multiuser-systeem wordt scheduling belangrijker. Bij een single-user computer hoeft het besturingssysteem slechts de behoeften van één gebruiker te vervullen. In een multi-user-computer gaat het om de behoeften van velen. Dit kan moeilijk of zelfs onmogelijk zijn. Aangezien veel programma's de resources van de computer gemeenschappelijk moeten gebruiken, moet het besturingssysteem beslissen wie wat krijgt en wanneer.

VRAGENSTELLING BESTURINGSSYSTEMEN THEORIE

Welke 3 soorten programma's bestaan er ?

1. **Interactieve programma's:** Een interactief programma is een programma dat een gebruiker vanaf de terminal activeert. Over het algemeen voert de gebruiker een korte opdracht in. Het besturingssysteem vertaalt deze opdracht en onderneemt actie. Het zet vervolgens een prompt-teken op het scherm en geeft daarmee aan dat de gebruiker een volgende opdracht kan invoeren. De gebruiker voert weer een opdracht in en het proces gaat door. De gebruiker werkt met het besturingssysteem op een conversatie-achtige manier, interactieve mode genoemd. Interactieve gebruikers verwachten een snelle respons. Daarom moet het besturingssysteem interactieve gebruikers voorrang geven.
2. **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. Uiteindelijk zal het besturingssysteem de opdrachten uitvoeren. Batchgebruikers verschillen van interactieve gebruikers omdat zij geen directe respons verwachten. Bij scheduling houdt het besturingssysteem hiermee rekening.
3. **Real-time programma's:** Real-time programmering legt aan de respons een tijdsbeperking op. Het wordt gebruikt wanneer een snelle respons essentieel is. Interactieve gebruikers geven de voorkeur aan een snelle respons, maar real-time gebruikers eisen dit zelfs. Voorbeelden van real-time verwerking: controlesysteem voor het luchtverkeer op een vliegveld, robots,...

Wat is de definitie van een virtuele machine ?

Een virtuele machine is een computerprogramma dat een volledige computer nabootst, waar andere (besturing)programma's op kunnen worden uitgevoerd.

Welke 3 soorten virtuele machines bestaan er ?

1. Programmeertaal-specifieke: bv JVM
2. Emulator: Bv. VirtualBox, VMWare, Parallels Desktop
3. Applicatie-specifieke: Bv. Docker

Wat is een Hypervisor ?

Een Hypervisor of Virtual Machine Monitor is in de informatica een opstelling die ertoe dient om meerdere besturingssystemen tegelijkertijd op een hostcomputer te laten draaien. Met de term hypervisor wordt de eerdere term supervisor uitgebreid, die gewoonlijk toegepast werd op besturingssysteemkernels. Een hypervisor regelt een vorm van virtualisatie.

Hypervisors zijn in twee soorten ingedeeld. Welke ?

1. **Type 1 Hypervisor:** deze ligt direct op de hardware, dat wil zeggen dat er geen besturingssysteem tussen ligt, hierdoor kunnen er meer resources aan de virtuele machines gegeven worden. Een van de nadelen van een Type1 is dat je enige technische kennis moet hebben om ermee te kunnen werken. Type1 hypervisors worden dan vooral ook gebruikt voor en door servers. Enkele voorbeelden van type1 hypervisors zijn VMWare ESXI, Citrix Xen, KVM en Microsoft Hyper-V
2. **Type 2 Hypervisor:** deze ligt in tegenstelling tot een Type1 niet direct tegen de hardware aan, dit wil zeggen dat er wel een besturingssysteem onder ligt. Dit kunnen er verschillende zijn, zoals Microsoft Windows, Apple macOS en Linux. Voordelen die je hebt bij Type2 is dat hij aan de praat te krijgen is zonder al te veel technische kennis, meestal kunnen Type2 Hypervisors geïnstalleerd worden zoals een programma. Een nadeel is dat Type2 Hypervisors niet zo krachtig en efficiënt zijn als Type1. Enkele voorbeelden van Type2 Hypervisors zijn: Oracle VirtualBox, VMWare Workstation en Parallels Desktop.

Wat is een proces ?

Een proces is elke onafhankelijk uitgevoerde entiteit die meedingt naar het gebruik van resources. Het zijn een of meer reeksen opdrachten die door een besturingsprogramma worden beschouwd als een werkeenheid.

Welke 4 soorten resources bestaan er ?

1. **Geheugen:** Een proces heeft geheugen nodig waarin het zijn instructies en gegevens kan opslaan. Een besturingssysteem moet er daarom voor zorgen dat het proces voldoende geheugen krijgt. Geheugen is echter een eindige resource. Het besturingssysteem mag niet toelaten dat een proces zoveel geheugen heeft dat de andere processen niet kunnen 'runnen'. Bovendien stellen privacy en beveiliging de eis dat een proces niet in staat mag zijn zich eigenmachtig toegang tot het geheugen van een ander proces te verschaffen. Het besturingssysteem moet deze resource niet alleen toewijzen, maar ook de toegang regelen.
2. **CPU:** De CPU is ook een resource die elk proces nodig heeft om zijn instructies te kunnen uitvoeren. Aangezien er gewoonlijk meer processen dan CPU's zijn, moet het besturingssysteem het gebruik van de CPU regelen; dat moet echter wel eerlijk gebeuren. Als het goed is, krijgen belangrijke processen de CPU snel ter beschikking en mogen minder belangrijke processen de CPU niet gebruiken ten koste van andere.
3. **Devices:** Randapparatuur of devices zijn onder andere printers, typedrives en disk-drives. Net als bij de CPU zijn er gewoonlijk meer gebruikers dan devices. Wat gebeurt er als verscheidene processen naar dezelfde printer of dezelfde drive willen schrijven ? Het besturingssysteem moet uitzoeken wie tot wat toegang heeft. Het moet ook de gegevensstroom regelen wanneer de processen van devices lezen of naar devices schrijven.
4. **Files:** Het besturingssysteem wordt verondersteld snel een bepaalde file te kunnen lokaliseren. Ook wordt verwacht dat het snel een bepaald record in de file kan lokaliseren. Omdat devices vaak vele duizenden files bevatten en een file vele duizenden records kan bevatten, is dit een ingewikkelde taak.

Wat is concurrency ?

Het feit dat processen meestal niet onafhankelijk zijn, maar concurrent.

Geef 3 voorbeelden waar conflicten (concurrency) kan optreden.

1. 2 processen willen dezelfde printer gebruiken
2. 2 processen willen dezelfde file lezen of schrijven
3. 2 processen willen communiceren

Welke ontwerpcriteria zijn er ?

1. **Consistentie:** Consistentie is een belangrijk ontwerp-criterium. Als het aantal processen dat van de computer gebruik maakt, vrijwel constant blijft, hoort dat ook voor de respons te gelden.
2. **Flexibiliteit:** Een besturingssysteem hoort zo te zijn geschreven dat een nieuwe versie het draaien van oude applicaties niet onmogelijk maakt. Bij een besturingssysteem moeten ook eenvoudig nieuwe randapparaten kunnen worden toegevoegd.
3. **Overdraagbaarheid:** Dit houdt in dat het besturingssysteem op verschillende soorten computers werkt. Overdraagbaarheid geeft de gebruiker meer flexibiliteit.

Wat is scheduling ?

Efficiënt middelen (bronnen, resources) inzetten om taken (opdrachten, jobs) uit te voeren.

Betekent multiprogrammering dat er veel processen gelijktijdig worden uitgevoerd ?

Nee, dit is iets dat vaak verkeerd begrepen wordt. Het geheugen kan van vele processen tegelijk de code bevatten, maar als er maar één CPU is, kan er maar één proces tegelijk worden uitgevoerd. Multiprogrammering is iets anders dan multiprocessing.

Wat is multiprocessing ?

Multiprocessing betekent dat het systeem meerdere processors bevat. Elke CPU kan de code van een afzonderlijk proces uitvoeren. Multi-programmering in een systeem met één CPU levert veel interessante problemen op. De verschillende processen willen namelijk allemaal toegang tot de CPU.

Wat is doorvoersnelheid/throughput ?

Het aantal processen per tijdseenheid door het systeem. Lage doorvoersnelheid zorgt voor weinig processen – hoge doorvoersnelheid zorgt voor veel processen. Hoge doorvoersnelheid lijkt het meest interessant, maar houdt geen rekening met procesgrootte.

Uit wat bestaat een proces ?

Een proces bestaat uit een context met een ID, status, Proces Control en prioriteit. Daarnaast bestaat het ook nog uit instructies en data.

Wat is de procestoestand ?

De procestoestand is de status. Die is gedefinieerd in termen van de mogelijkheid om te worden uitgevoerd. Het is de rechthoek in een procestoestandsdiagram.

Geef de verschillende procestoestanden.

1. HOLD
2. READY
3. RUNNING
4. WAIT
5. SUSPENDED
6. COMPLETE

Wat betekent HOLD ?

De toestand HOLD duidt aan dat het proces is aangeboden. Het besturingssysteem staat het alleen nog niet toe te worden uitgevoerd of resources aan te vragen. Deze toestand treedt op wanneer iemand een proces aanbiedt met de instructies dat het later moet worden gestart. Veel systemen kunnen een aangeboden proces initialiseren op een bepaald tijdstip, of nadat een bepaalde tijd is verstreken.

Wat betekent READY ?

De READY toestand geeft aan dat een proces ready-to-run is: gereed om te worden uitgevoerd. Een proces in deze toestand is *idle* (ruststand: toestand waarin een systeem wacht, omdat er niets te doen valt), maar kan worden uitgevoerd als het de besturing over de CPU krijgt. In systemen met één processor kunnen veel READY processen staan te wachten, er kan er maar één tegelijk worden uitgevoerd. Deze toestand is niet van toepassing op een proces dat op I/O wacht.

Wat betekent RUNNING ?

De toestand RUNNING geeft aan dat het proces wordt uitgevoerd en op dat moment onder de besturing van de CPU staat. In een single-processorsysteem is er maar één proces in deze toestand. Bevat het systeem echter meerdere CPU's, dan kunnen verscheidene processen gelijktijdig worden uitgevoerd.

Wat betekent WAIT ?

De toestand WAIT geeft aan dat het proces op iets staat te wachten. Het kan bijvoorbeeld een I/O request hebben geproduceerd. Het kan niet worden uitgevoerd omdat het moet wachten tot de I/O actie klaar is.

Wat betekent SUSPENDED ?

De toestand SUSPENDED (opgeschort) is in één opzicht vergelijkbaar met de WAIT-toestand: het proces kan niet naar de CPU meedingen. Het verschilt echter in die zin, dat processen in een wait-toestand nog wel om sommige resources mogen vragen. Het proces dat een I/O aanvraag gedaan heeft, staat in feite te wachten op een I/O-processor of een I/O-controller. Het proces is idle ten opzichte van de CPU, maar het vordert nog steeds. Een proces in de SUSPENDED-toestand kan geen enkele resource-aanvraag meer doen. Het proces wordt tijdelijk compleet stilgelegd. Processen kunnen bijvoorbeeld worden opgeschort wanneer het systeem te veel processen bevat. De intensieve strijd om resources kan de voortgang van alle processen vertragen. Een van de oplossingen daarvoor is het opschorten van een aantal processen, zodat het systeem de rest efficiënter kan afhandelen. Wanneer de zware competitie om resources verminderd, kan het systeem de opgeschorte processen weer activeren.

VRAGENSTELLING BESTURINGSSYSTEMEN THEORIE

Wat betekent COMPLETE ?

Een proces is in de toestand COMPLETE wanneer het volledig is afgewerkt.

Bespreek de toestandsovergang niet-aangeboden => HOLD

De overgang van niet-aangeboden naar HOLD vindt plaats wanneer een gebruiker een programma aanbiedt met instructies de uitvoering van dat programma tot later uit te stellen. De JCL kan een start- of vertragingstijd bevatten. Het programma blijft in de HOLD-status tot het juiste ogenblik is aangebroken.

Bespreek de toestandsovergang HOLD/niet-aangeboden => READY

De overgang van HOLD of niet-aangeboden naar READY vindt plaats wanneer een gebruiker een programma aanbiedt dat van plan is direct een resource-aanvraag te doen. Deze overgang treedt ook op bij de start van een programma dat eerder op basis van vertraging is aangeboden.

Bespreek de overgang van READY naar RUNNING

De overgang van READY naar RUNNING vindt plaats wanneer het besturingssysteem de besturing van de CPU aan een bepaald proces overdraagt. De basis waarop de CPU de processen selecteert, is belangrijk en daarom het onderwerp van een volgende paragraaf (strategieën voor scheduling).

Bespreek de overgang van RUNNING naar READY

De overgang van RUNNING naar READY vindt plaats wanneer het besturingssysteem de besturing van de CPU terugkrijgt van een proces dat, indien toegestaan, nog kan runnen. Sommige systemen bepalen bijvoorbeeld hoeveel tijd een proces zonder onderbreking mag runnen (het quantum). Als het quantum is verbruikt, neemt het besturingssysteem de besturing over de CPU weer terug. Het zet het proces in de READY-toestand en geeft de CPU aan het volgende proces in de rij. Op deze manier kan geen enkel proces de CPU monopoliseren.

Bespreek de overgang van RUNNING naar WAIT

De overgang van RUNNING naar WAIT vindt plaats wanneer het proces iets aanvraagt waarop het moet wachten. We hebben bijvoorbeeld al gezien dat een proces dat om I/O verzoekt, moet wachten tot de I/O-actie is afgelopen.

Bespreek de overgang van WAIT naar READY

De overgang van WAIT naar READY vindt plaats wanneer een wachtend proces krijgt wat het nodig heeft. Zo kan bijvoorbeeld het besturingssysteem een signaal ontvangen dat een I/O-actie is afgerond. Daarna zet het het wachtende proces in de READY-toestand terug.

Bespreek de overgang van RUNNING naar COMPLETE

De overgang van RUNNING naar COMPLETE vindt plaats wanneer het proces afgelopen is. Dit betekent at de noodzakelijke werkzaamheden zijn afgerond of dat er bij het proces iets fout is gegaan en het wordt afgebroken. In beide gevallen wordt het proces beëindigd, voor zover dat het besturingssysteem betreft.

Bespreek de overgang van READY naar SUSPENDED

De overgang van READY naar SUSPENDED vindt plaats wanneer er teveel READY processen zijn om nog adequaat service te verlenen. Elk computersysteem heeft beperkte resources en als het aantal processen dat toegang tot deze resources vraagt, zonder beperking kan groeien, raakt het systeem verzadigd als een overvolle snelweg, zodat er filevorming optreedt en elk proces maar een gebrekkige service krijgt. Een manier om dit te vermijden is het aantal READY processen te beperken. Een andere manier is het verplaatsen van processen van de READY-toestand naar de SUSPENDED-toestand wanneer de responsitiden slecht worden. Welke processen hiervoor in aanmerking komen, is natuurlijk een beslissing die het besturingssysteem moet nemen.

Bespreek de overgang van SUSPENDED naar READY

De overgang van SUSPENDED naar READY vindt plaats wanneer het besturingssysteem besluit dat een opgeschorst proces weer naar resources kan meedingen. Vermoedelijk is de belasting tot normale niveaus teruggebracht en is het niet langer nodig om processen in de suspendeer toestand te brengen.

Wat is een Process Control Block (PCB) ?

Het besturingssysteem houdt een lijst van Process Control Blocks bij. In principe is er één PCB voor elk proces. Wanneer een proces wordt geïnitialiseerd, creëert het besturingssysteem een PCB voor dit proces en houdt een lijst van PCB's bij. Wordt een proces beëindigd, dan verwijdert het systeem het PCB uit de lijst.

Wat bevat een PCB ?

Een PCB bevat alles wat het besturingssysteem over het proces zou moeten weten:

- **Identificatienummer van het proces** (proces-ID)
- **Procestoestand:** Als het proces van toestand verandert, werkt het besturingssysteem het PCB van dit proces bij.
- **Maximale looptijd en actuele looptijd:** Gedurende de uitvoering van een proces gaat het systeem na hoeveel CPU-tijd het gebruikt. Bovendien mag de actuele looptijd de maximaal toelaatbare looptijd niet overschrijden. De maximale looptijd is een door het systeem gedefinieerde parameter en wordt bij de initialisatie van het PCB opgeslagen.
- **Huidige resources en limieten:** Hieronder vallen onder andere het aantal printerpagina's, de hoeveelheid geheugen en de hoeveelheid schijfruimte
- **Procesprioriteit:** Het systeem kan een proces schedulen of het op basis van zijn prioriteit toegang tot resources geven. Sommige processen (bijvoorbeeld routines van het besturingssysteem) hebben een hoge prioriteit
- **Opslaggebieden:** Als een proces tijdelijk wordt stopgezet, moet het systeem bepaalde registerwaarden bewaren. Het systeem kan deze waarden dan later herstellen, zodat het proces de uitvoering op dezelfde plek kan hervatten als waar het stopte.
- **Locatie van de code of de segmenttabel van een proces:** Wanneer het besturingssysteem een proces laat runnen, moet het weten waar de code of de segmenttabel van dat proces zich bevindt.

Wat zijn de niveaus van scheduling ?

Hoog, middel en laag niveau.

Wat is scheduling op hoog niveau ?

Scheduling op hoog niveau (job scheduling) stelt vast welke programma's of jobs toegang tot het systeem krijgen. Besturingssysteem-routines die de high-level scheduling verzorgen, controleren en de Job Control Language (JCL) van een job. High-level scheduling regelt de toestandovergangen 1, 2 en 7 (niet-aangeboden => HOLD, HOLD/niet-aangeboden => READY, RUNNING => COMPLETE) in het procestoestand-diagram. High-level schedulers controleren een checklist voordat ze een job toestemming geven het systeem binnen te komen. Ze reageren eenvoudig op events en beschermen de integriteit en de veiligheid van het systeem. Er is relatief zelden behoefte aan high-level schedulers.

Wat is scheduling op middenniveau (intermediate level) ?

Scheduling op middenniveau bepaalt welke processen in feite kunnen meedingen naar de CPU. Processen die I/O requests hebben geproduceerd kunnen daarbij niet meedoen. Soms schort het systeem een proces op wanneer de vraag ongewoon hoog is. Intermediate-level scheduling houdt zich voornamelijk bezig met de toestandovergangen 5, 6, 8 en 9 (RUNNING => WAIT, WAIT => READY, READY => SUSPENDED, SUSPENDED => READY) van het procestoestand-diagram. Net als high-level schedulers zijn ook de intermediate-level schedulers event-gestuurd. Events als een I/O-aanvraag of I/O-einde bepalen meestal welke toestandovergang plaatsvindt.

Wat is Low-level scheduling ?

Low-level scheduling is verantwoordelijk voor de toestandovergangen 3 en 4 (READY => RUNNING, RUNNING => READY) in het procestoestand-diagram. In vele opzichten is scheduling op laag niveau het moeilijkst te implementeren. In tegenstelling tot andere niveaus is deze niet hoofdzakelijk event-gestuurd. Vaak zijn er veel processen die om CPU-tijd vragen en low-level scheduling moet vaststellen welk proces toegang krijgt.

Wat zijn time-sharing-systemen ?

In time-sharing-systemen, waar processen de CPU om beurt gebruiken, vindt low-level scheduling plaats. Is er veel I/O-activiteit, dan kan het elke paar milliseconden wel nodig zijn. Algoritmen voor low-level scheduling moeten proberen de CPU en de andere resources efficiënt toe te wijzen. Ze moeten een rechtvaardige respons naar de gebruikers garanderen.

Wat is nonpreemptive scheduling ?

Bij nonpreemptive scheduling houdt een proces dat de besturing over de CPU krijgt, deze besturing tot het proces is afgelopen. Het besturingssysteem neemt het proces de besturing niet af. Het besturingssysteem geeft de besturing van de CPU alleen aan een proces wanneer een ander proces afgelopen is. Daarom is scheduling op een laag niveau zelden nodig bij nonpreemptive scheduling. Nonpreemptive scheduling heeft het nadeel dat het niet op andere processen reageert. Het proces dat de besturing over de CPU heeft, krijgt volledige service, maar alle andere processen moeten wachten.

Wat is preemptive scheduling

Bij preemptive scheduling kan een proces niet oneindig lang de besturing over de CPU houden. Na een bepaalde periode kan het besturingssysteem besluiten de CPU van dit proces weg te halen. De processen in de READY-toestand moeten de CPU om beurten gebruiken. Om kort te zijn: het besturingssysteem beslist hier over.

In het algemeen kan het besturingssysteem een running proces de besturing van de CPU ontnemen om verschillende redenen, bijvoorbeeld: het proces is beëindigd; het proces genereert een request waarop het moet wachten; de uitvoering van het proces heeft al een hele tijd geduurd. De maximale tijd die het besturingssysteem een proces zonder onderbreking laat runnen is een parameter van het besturingssysteem en wordt quantum genoemd. Als het quantum wordt bereikt, slaat het besturingssysteem de context van het proces op (statusword, de registers en de programmateller), en geeft het de CPU gedurende een bepaalde tijd aan een ander proces.

Geef de criteria voor een scheduler.

1. **CPU-gebruik:** de mate waarin de processor gebruikt wordt door de verschillende processen
2. **Debit:** Het aantal processen dat zijn werk voltooit in een bepaalde tijd, Turnaround tijd: de tijd die nodig is om een proces te voltooien.
3. **Wachttijd:** De som van alle tijd die een proces doordrengt in een wachtrij
4. **Responstijd:** De tijd die nodig is om een reactie te krijgen. Het is de tijd tussen het indienen van een aanvraag en het krijgen van een antwoord.

Wat is Round Robin Scheduling ?

In dit scheduling algoritme wordt gebruikgemaakt van een vaste tijdswaarde, ook wel tijdkwantum genoemd. Wanneer dit tijdkwantum overschreden wordt, zal de scheduler het process onderbreken en een volgend process inladen. Een moeilijkheid is het bepalen van de grootte van het tijdkwantum. Een te groot tijdkwantum zal er voor zorgen dat we een FCFS (First Come First Served) karakter krijgen en een te klein tijdkwantum zal voor een overhead aan context switches zorgen.

Wat is een context switch ?

Een context switch is het wisselen van processen. Wanneer de scheduler een proces onderbreekt zal hij de huidige status van het proces bewaren. Bij een te klein tijdkwantum zal de processor meer bezig zijn met het verwerken van context switches dan met het effectief uitvoeren van processen.

Leg de preemptive Round Robin Methode uit

Elk proces in de READY-toestand heeft een entry in de queue (gewoonlijk het PCB van het proces). Wanneer de CPU beschikbaar komt, geeft het besturingssysteem de besturing aan het proces waarvan het PCB aan het begin van de wachtrij staat, zodat het proces met de uitvoering begint. Het proces wordt verder uitgevoerd totdat één van de drie eerder aangegeven events optreedt. Op dat punt haalt het besturingssysteem de besturing van de CPU bij het proces weg. Als het proces afgelopen is, verwijdert de high-level scheduler het uit het systeem en verwijdert zijn PCB. Meestal creëert het besturingssysteem een overzicht van de gebruikte systemtijd en resources, en brengt dit gebruik in rekening. Daarna bestaat het proces niet langer in het denken van het besturingssysteem.

Als het proces niet stopt en evenmin een aanvraag genereert waarop het moet wachten, neemt het besturingssysteem het heft in handen. Het proces mag de CPU niet langer besturen dan een kwantum. Als een kwantum is verbruikt en het proces nog steeds wordt uitgevoerd, stopt het besturingssysteem het proces en zet het PCB aan het einde van de queue. Dan geeft het de CPU aan het proces dat vooraan in de queue staat. Het voortijdig onderbroken (preempted) proces moet wachten tot alle andere processen in de wachtrij een gelegenheid hebben gehad om de CPU te gebruiken. Preemption voorkomt dat processen de CPU monopoliseren. Geen enkel proces kan langer dan de kwantumtijd worden uitgevoerd zonder onderbroken te worden. Hierdoor krijgt elk proces in de READY-toestand gelegenheid gedurende een bepaalde tijd de CPU te gebruiken.

Waar wordt Round Robin vaak gebruikt ?

Round Robin scheduling wordt het meest gebruikt in systemen met veel, achter terminal werkende, interactieve gebruikers. Round Robin scheduling vereist echter enige overhead. Het besturingssysteem moet de activiteiten van elk proces van heel dichtbij volgen. Een ingebouwde timer moet elk proces onderbreken dat langer dan een quantum van de CPU gebruik probeert te maken. Wanneer een dergelijke interrupt optreedt, voert het besturingssysteem een rescheduling uit en geeft de besturing van de CPU aan een ander proces. Hierdoor is het mogelijk dat het besturingssysteem vrij veel moet ingrijpen. Aangezien routines van het besturingssysteem meestal dezelfde CPU gebruikt als de processen die CPU-tijd willen hebben, vermindert de totale hoeveelheid tijd die voor processen beschikbaar is. Daarnaast is de keuze van een quantumwaarde kritisch. Deze is bij Round Robin scheduling een sleutelwaarde en moet zorgvuldig gekozen worden.

Wat is het First-in-First-out scheduling principe (FIFO) of First-come-first-served (FCFS) ?

Dit is het eenvoudigste algoritme. Wanneer een process als eerste om de CPU vraagt dan zal hij die ook krijgen. Processen die erna komen moeten wachten. Deze vorm van scheduling kan geïmplementeerd worden door een First-in-first-out (FIFO) model. Wanneer een process lang zal duren, zullen korte processen die erna komen lang moeten wachten.

Hoe werkt dit principe ?

Deze nonpreemptive-methode is heel eenvoudig: Geef de besturing van de CPU aan het proces dat zich het langst in het systeem bevindt. Het proces dat het eerste aankwam mag de CPU gebruiken. Wanneer een proces is geïnitialiseerd, plaatst het systeem het PCB van het proces aan het einde van de queue. Wanneer de CPU beschikbaar komt, geeft het besturingssysteem de besturing van de CPU aan een ander proces. Het belangrijkste voordeel van deze methode is zijn eenvoud. Dit is iets wat nooit mag worden onderschat. Het besturingssysteem besluit alleen tot rescheduling als dat absoluut noodzakelijk is. De overhead is daardoor klein.

In vele systemen kan FIFO onrealistisch zijn (bijvoorbeeld in een hoog-interactief systeem met veel I/O-activiteit, toch zijn er gevallen waarin FIFO gewenst is (bijvoorbeeld een systeem waarin de meeste processen zwaar rekenwerk verrichten, maar erg weinig I/O-aanvragen produceren)).

Wat zijn hybride-scheduling-methoden ?

Bijvoorbeeld een FIFO-scheduler kan ook deel uitmaken van een ingewikkeldere methode: bijvoorbeeld bij systemen die zowel batch-gebruikers als interactieve gebruikers hebben. Interactieve gebruikers laten korte programma's draaien, hebben weinig informatie nodig of ontwikkelen nieuwe applicatie en hebben een snelle respons nodig. Interactieve gebruikers op terminals verwachten bijna onmiddellijk respons van het systeem. Korte onderbrekingen in de uitvoering worden direct opgemerkt. Een batch-gebruiker daarentegen biedt zijn of haar programma op een bepaalde tijd aan, samen met de JCL-opdrachten die nodig zijn om het te laten draaien. Nadat het is aangeboden, is de gebruiker echter vrij om iets anders te gaan doen. In tegenstelling tot de interactieve gebruiker hoeft hij of zij tijdens de verwerking van het proces niet aanwezig te zijn. De programma's van batch-gebruikers zijn meestal langer of willen veel uitvoer produceren. In tegenstelling tot interactieve gebruikers merken batch-gebruikers niets van korte onderbrekingen. Het besturingssysteem kan dit gebruiken om aan beide gebruikers een goede service te bieden.

Wanneer batch-processen en running processen worden gemengd, is een Round Robin schedule van alle processen niet gewenst, met name als er veel batch-gebruikers zijn. Deze hebben geen snelle respons nodig. Waarom zouden we dan niet wat CPU-tijd van batch-processen stelen en die aan de interactieve processen ter beschikking stellen ? De batch-gebruikers merken het verschil niet eens en de interactieve gebruikers zullen de verbeterde respons waarderen.

Wat zijn batch-partities ?

Een batch-partitie is een virtuele geheugenconstructie die één batch-proces bevat. Een virtueel geheugen is een geheugenbeeld dat het besturingssysteem aan de gebruiker verschaft. Er kunnen verscheidene batch-partities zijn.

Als de processen het systeem binnengaan, maakt het systeem verschil tussen de batch-processen en de interactieve processen. Alle interactieve processen komen direct in de READY-toestand en dingen mee naar CPU-tijd. Het systeem zet de batch-processen echter in een wachtrij voor de batch-partitie. Aangezien de partitie slechts één proces kan bevatten, moeten alle andere wachten. Als de partitie leeg is, wordt het proces aan het begin van de queue in de partitie geplaatst. Het proces komt in de READY-toestand en dingt samen met de interactieve processen mee naar de CPU-tijd. Het resultaat is een quota-systeem waarbij slechts één batch-proces tegelijk de gelegenheid krijgt naar de CPU mee te dingen. Hierin is de low-scheduler Round Robin, maar er is nooit meer dan één batch-proces READY, en de intermediate-level FIFO-scheduler bepaalt welke processen in de READY-toestand komen.

Round Robin en FIFO zijn twee veel voorkomende methoden bij scheduling op laag-en middelniveau, maar het zijn geenszins de enige methoden.

Wanneer is Round Robin het meest geschikt en wanneer is FIFO het meest geschikt ?

Round Robin is het meest geschikt wanneer gebruikers een snelle respons willen. Het pas heel goed in systemen die batch-processen verwerken die veel I/O requests genereren. In dit geval willen we dat de processen hun eigen aanvragen zo spoedig mogelijk genereren. Dit houdt de I/O-processors actief en verkleint het aantal processen in de READY-toestand. Aangezien Round Robin garandeert dat alle READY-processen een kans krijgen om te worden uitgevoerd, vergroot deze de kansen dat er I/O-requests worden gegenereerd.

FIFO daarentegen is het meest geschikt voor omgevingen waar zwaar rekenwerk moet worden verricht of als onderdeel van een ingewikkeldere scheduling-benadering. Er is niets te winnen door FIFO in interactieve omgevingen te gebruiken of wanneer er veel I/O-activiteit bestaat. Omgekeerd is er evenmin winst als we Round Robin in een rekenomgeving zouden toepassen.

Wat zijn Multilevel Feedback Queues (MFQ) ?

Er bestaat een manier waarop de scheduling-methode op Round Robin kan lijken als er veel I/O-activiteit is en op FIFO kan lijken wanneer er weinig of geen I/O-activiteit is: dit wordt Multilevel Feedback Queues genoemd. Bij deze methode is de scheduling afhankelijk van de activiteiten die op een bepaald moment plaatsvinden. De beste scheduling-methode is afhankelijk van de soorten processen in de READY-toestand en het MFQ-systeem is gevoelig voor wijzigingen in die activiteiten. Wanneer er veel I/O-activiteit plaatsvindt, lijkt MFQ op een Round Robin scheduler. Is er daarentegen maar weinig of geen I/O-activiteit, dan lijkt MFQ op een FIFO-scheduler. MFQ is nooit compleet FIFO, omdat het dan niet op veranderingen zou kunnen reageren in de tijd dat een bepaald proces de besturing over de CPU heeft. Het voert echter minder reschedules uit door elk proces de gelegenheid te geven gedurende langere tijd de CPU te bestuderen.

Hoe ziet een MFQ eruit en hoe werkt het ?

Zoals de naam suggereert, bestaat MFQ uit vele queue. Ze zijn genummerd van 1 tot n en elke wachtrij heeft een eigen prioriteit (q1 heeft de hoogst prioriteit en qn heeft de laagste prioriteit). Het PCB voor een proces in de READY-toestand is in één van de queues geplaatst. De prioriteit van het PCB wordt bepaald door de queue waarin het zich bevindt.

De low-level scheduler werkt altijd op basis van prioriteit. Een proces met lage prioriteit krijgt alleen de besturing van de CPU als er geen proces met een hogere prioriteit bestaat. Heeft een aantal processen dezelfde prioriteit, en is er geen met een hogere, dan kiest de scheduler het proces aan het begin van de queue (FIFO).

Telkens wanneer een nieuw proces wordt geïnitialiseerd, zet het systeem het in de READY-toestand en wordt het PCB van het proces in de eerste queue geplaatst. Elk nieuw proces heeft aan het begin dus de hoogste prioriteit die niet leeg is. Dan geeft de low-level scheduler de besturing van de CPU aan het proces dat vooraan in die queue staat.

Het proces krijgt de besturing van de CPU tot één van de volgende dingen gebeurt:

- Het proces wordt beëindigd (elegant of door het af te breken)
- Het proces vraagt iets aan waarop het moet wachten (bijvoorbeeld een I/O-request)
- Het quantum is verbruikt.

Het quantum is afhankelijk van de queue waaruit het proces is gekozen. Het is kenmerkend dat elke queue een ander quantum heeft. Queue 1 heeft het kleinste quantum en queue n heeft het grootste quantum.

Als het proces afgelopen is, verlaat het het systeem en speelt het verder geen rol.

Als het proces iets aanvraagt waarop het moet wachten, zet de intermediate-level scheduler het in de WAIT-toestand. Wanneer het proces krijgt wat het nodig heeft, keert het terug naar de READY-toestand. Het besturingssysteem plaatst het PCB aan het einde van de queue met net iets hogere prioriteit dan de queue waarin het heeft gestaan. Het besturingssysteem verhoogt de prioriteit van het proces.

Stel dat het proces geen enkele aanvraag doet en niet stopt. Dan haalt het besturingssysteem de besturing van de CPU bij het proces weg nadat het quantum is verbruikt. In dit geval blijft het proces in de READY toestand en het besturingssysteem plaatst het PCB aan het einde van de queue met net een iets lagere prioriteit dan de queue waarin het heeft gestaan.

Als de processen interactief zijn of I/O-gebonden, ontvangt het besturingssysteem geregeld I/O-aanvragen. Stel dat het quantum iets langer is dan de gemiddelde tijd tussen I/O-requests. Hierdoor genereren de meeste processen I/O-requests voordat het quantum is verbruikt.

Wanneer een I/O-request is afgelopen, komt het proces uiteindelijk opnieuw in de READY-toestand met een hogere of dezelfde prioriteit. Daarom heeft elk I/O-gebonden proces de neiging een hoge prioriteit te behouden en meer I/O-requests te genereren.

VRAGENSTELLING BESTURINGSSYSTEMEN THEORIE

Als het proces enige tijd wordt uitgevoerd zonder een I/O-aanvraag, raakt het quantum op. In dit geval blijft het proces in de READY-toestand, maar met een lage prioriteit (tenzij het de laagste prioriteit heeft). Is het proces echter nog I/O-gebonden, dan zal het spoedig meer I/O-aanvragen genereren en weer hogere prioriteit krijgen.

Stel dat de binnenkomende processen echter rekenintensief zijn. Deze zullen dan weinig I/O-requests genereren, maar de meeste tijd spenderen aan het uitvoeren van code. Wanneer ze de besturing van de CPU krijgen, meestal zal het quantum worden verbruikt. Het gevolg is dat ze met een lagere prioriteit naar de READY-toestand terugkeren (tenzij ze al de laagste prioriteit hebben).

Als alle READY-processen rekenintensief zijn, krijgen ze uiteindelijk allemaal de laagste prioriteit. De low-level scheduler heeft dus vooral te maken met processen met een lage prioriteit. Het wordt een Round Robin Scheduler met een groter quantum dan voor I/O-gebonden processen wordt gebruikt. De grotere quanta zijn gerechtvaardigd omdat er met geregelde rescheduling van rekenintensieve processen niets wordt gewonnen. Hoewel de scheduler nog steeds Round Robin is, zorgend e grotere quanta voor een werkwijze die meer op FIFO lijkt.

Laten we vervolgens aannemen dat het gaat om een mengsel van I/O-gebonden en CPU-gebonden processen. De I/O-gebonden processen behouden een hoge prioriteit, terwijl de prioriteit van de CPU-gebonden processen afneemt. De scheduler begunstigt de I/O-gebonden processen boven de CPU-gebonden processen. Door dit te doen, blijven de I/O-processors actief en neemt de graad van multiprocessing toe. CPU-gebonden processen worden alleen uitgevoerd wanneer er geen I/O-gebonden processen READY zijn.

Niet alleen is MFQ gevoelig voor verschillen tussen processen, het is ook gevoelig voor veranderingen binnen een proces. Zo kan een proces van start gaan als I/O-gebonden maar opeens veranderen in een CPU-gebonden proces. Het omgekeerde kan ook voorkomen.

Het systeem met queues op verschillende niveaus (MFQ) is een flexibele methode die zorgt voor een automatische aanpassing aan de werkbelasting en aan veranderingen in het gedragspatroon van processen. Als een systeem hoofdzakelijk interactieve of CPU-gebonden processen heeft, is MFQ ingewikkelder dan nodig. In een onvoorspelbare omgeving kan deze methode echter productief zijn. Dit is in feite de scheduling-methode die door VMS wordt toegepast.

Wat is Shortest-job-first scheduling (SJF) ?

Bij dit algoritme zal de scheduler het proces met de kleinste lengte uitvoeren. Een probleem is het voorspellen van de lengte van een proces. Dit kan gedaan worden door bijvoorbeeld het aantal instructies te tellen of het aantal in-en uitvoer aanvragen. Een voordeel ten opzichte van het FCFS-algoritme is dat de wachttijd bij SJF kleiner is. Het SJF-algoritme bestaat in twee vormen, de preemptive en de non-preemptive vorm. Een gevaar bij SJF-scheduling is dat een heel lang proces nooit uitgevoerd zal worden. Dit noemt men "verhongering" of "starvation".

Wat is Shortest Remaining Job Next (SRJN) ?

Shortest Remaining Job Next en Shortest Job First scheduling zijn in die zin vergelijkbaar, dat de low-level scheduler rekening houdt met de hoeveelheid tijd die een READY-proces nodig heeft. Eenvoudig gesteld, het kiest het proces dat de minste tijd nodig heeft. Het verschil tussen de twee strategieën is dat SRJN preemptive is en SJF nonpreemptive.

Wat is het nadeel van SJF en SRJN ?

Een nadeel van beide methoden is dat het besturingssysteem het moeilijk heeft om te berekenen hoeveel tijd een proces nodig heeft. Het is moeilijk (in feite meestal onmogelijk) voor een besturingssysteem om te berekenen hoeveel tijd een proces nodig heeft. De persoon die het proces heeft aangeboden, zal echter beste een redelijke schatting kunnen geven (zo niet, dan zou hij of zij de job waarschijnlijk beter niet kunnen aanbieden). Dergelijke ramingen maken in feite deel uit van de vereiste JCL. De low-level scheduler kan deze schatting dus gebruiken om te beslissen welk proces de besturing over de CPU krijgt. Bij SJF neemt de low-level scheduler het READY-proces met de kortst geschatte run-tijd. Heeft dat eenmaal de besturing van de CPU, dan draait het tot het is afgelopen (non preemptive).

Nonpreemptive scheduling creëert problemen als er I/O-requests zijn. Maar, als er maar weinig of geen I/O wordt verwacht, is SJF uitvoerbaar. Korte processen krijgen een prima respons omdat ze de hoogste prioriteit hebben. De doorvoer-snelheid is indrukwekkend. In feite genereert SJF de hoogst mogelijke doorvoersnelheid. Als het erom gaat een zo groot mogelijk aantal tevreden gebruikers te krijgen, is dit de beste methode.

Het is echter mogelijk dat de doorvoersnelheid een verkeerde indruk van de productiviteit van het systeem geeft. Kijk maar naar de gebruiker wiens proces iets langer loopt dan veel andere. Het systeem straft dit af. Uiteraard kan het leiden tot indefinite postponement (onbepaald uitstel) of starvation (uithongeren). Dat wil zeggen, het kan eeuwig staan wachten. Zolang kortere processen het systeem blijven binnenkomen, zullen de langere niet worden uitgevoerd.

Het kan in hoge mate onrechtvaardig zijn om een proces uren te laten wachten terwijl een iets korter proces wel wordt ingevoerd en het systeem al heel snel weer verlaat. Dit is vooral het geval als het langere proces van uzelf is.

De preemptive versie van SJF is SRJN. Een proces dat de besturing over de CPU heeft, kan hiervan afstand doen als het om iets vraagt waarop het moet wachten. Als een nieuw proces een kortere geschatte runtijd heeft dan het proces dat op dat moment wordt uitgevoerd, krijgt het de besturing van de CPU. Net als SJF, genereert SRJN een hoge doorvoersnelheid, maar het lijdt ook aan dezelfde nadelen. Langere processen kunnen in onaanvaardbare mate worden vertraagd.

Wat is starvation ?

Een proces wordt nooit uitgevoerd. We hebben gezien dat de methoden SJF en SRJN starvation van langere processen kunnen veroorzaken. De MFQ methode kan ook starvation van processen in lagere queues tot gevolg hebben. Zolang de interactieve processen READY zijn, worden de CPU-gebonden processen genegeerd.

Kunnen we iets aan starvation doen of moeten we ermee leven en het als bijproduct van een bepaalde methode beschouwen ?

Eén mogelijkheid is dat we het negeren en hopen dat het geen ernstige problemen veroorzaakt. Dit is niet altijd een onrealistische benadering. Onder MFQ bijvoorbeeld verlaten interactieve processen de queues snel. Als gevolg van het verschil tussen de reken- en de I/O-snelheid duurt het voor deze processen veel langer om terug te keren. Daarom is het niet ongewoon te constateren dat de queues met hoge prioriteit snel leeg zijn. Natuurlijk ontstaat hierdoor voor de processen met lage prioriteit een gelegenheid om te runnen. Bij de methoden SJF en SRJN kan men vergelijkbare observaties uitvoeren. De ongelijkheid tussen de verwerkingssnelheden van de computer en de mens is hier zelfs nog groter. Er zijn waarschijnlijk niet veel gebruikers die korte processen sneller kunnen genereren dan een CPU deze kan uitvoeren. In deze gevallen is starvation zeldzaam.

Zelfs wanneer het optreedt, houdt het niet lang aan. Ook al zeggen we dat events waarschijnlijk niet voorkomen, dat wil niet zeggen dat zijn onmogelijk zijn. Starvation van CPU-gebonden processen onder MFQ is aannemelijker als het aantal interactieve processen toeneemt. Op dezelfde manier is starvation van lange processen onder SJF of SRJN waarschijnlijker als het aantal gebruikers toeneemt. In dergelijke gevallen is starvation een ernstiger probleem.

Als we ervoor kiezen het niet te negeren, wat kan het systeem er dan aan doen ? Eén methode is het opschorten van een aantal READY-processen. Dit reduceert het aantal processen en vermindert starvation. Het systeem hervat de processen wanneer het besluit dat het meer aankan.

Een ander alternatief is dat het systeem de prioriteiten periodiek opnieuw berekent. Het systeem kan dit doen met behulp van het veld LastTime dat zich in het PCB van elk proces bevindt. In het begin wordt daar de tijd opgeslagen waarop het proces het systeem is binnengekomen. Telkens wanneer een proces wordt onderbroken (preempted), vervangt het systeem de waarde in het veld door het tijdstip van de preemption. Onder nonpreemptive scheduling vervangt het veld nooit. Wanneer het systeem de prioriteit van een proces evalueert, bekijkt het de waarde 'huidige tijd – LastTime'. Een grote waarde betekent dat het proces al lange tijd is genegeerd. Als de waarde toeneemt tot voorbij een bepaalde drempelwaarde, verhoogt het systeem de prioriteit voor dat proces. Onder MFQ betekent dit dat het PDB in een hogere queue wordt geplaatst. Onder SJF of SRJN betekent het dat het prioriteitsveld van het PCB wordt veranderd.

Bij Round Robin en FIFO kan geen starvation optreden.

Wat is concurrency ?

Concurrency (ofwel parallelle processen) is bij computerprocessen een belangrijke plaats gaan innemen. Doordat het mogelijk werd enorme rekencapaciteit in een kleine chip te stoppen, zijn multiprocessors gemeengoed geworden. Dergelijke systemen kunnen vele taken gelijktijdig uitvoeren. Dit verhoogt de productiviteit, maar schept ook problemen.

Wat zijn de problemen die veroorzaakt worden door het gelijktijdig uitvoeren van taken ?

1. **Multiprogrammering:** het beheer van meerdere processen in een systeem met 1 processor
2. **Multiprocessing:** het beheer van meerdere processen in een systeem met meerdere processors
3. **Gedistribueerde verwerking:** het beheer van meerdere processen die worden uitgevoerd op een aantal verspreide computersystemen.

Concurrency hangt samen met tal van ontwerpquestiones. Welke ?

Communicatie tussen processen, het delen van en het vechten om resources, de synchronisatie van meerdere procesactiviteiten en het toedelen van processortijd aan processen.

Concurrency treedt op in 3 verschillende situaties. Welke ?

1. **Meerdere toepassingen:** Multiprogrammering werd uitgevonden om verwerkingsstijd dynamisch te kunnen verdelen tussen een aantal actieve toepassingen
2. **Gestructureerde toepassing:** Als uitbreiding op de beginselen van modulair ontwerpen en gestructureerd programmeren kunnen sommige toepassingen effectief worden geprogrammeerd als een verzameling van gelijktijdige processen
3. **Structuur van het besturingssysteem:** Dezelfde voordelen van het structureren gelden voor de systeemprogrammeur en we hebben gezien dat besturingssystemen zelf vaak worden geïmplementeerd als een verzameling processen of threads.

Wanneer is er geen probleem voor concurrency ?

Wanneer parallelle processen niets gemeenschappelijk gebruiken.

Wanneer ontstaan er dan moeilijkheden ?

Wanneer processen het gemeenschappelijke geheugen aanspreken.

Leg concurrency met 1 processor uit

Bij systemen met 1 processor zijn er ook parallelle processen mogelijk. In een dergelijk geval kunnen de processen niet tegelijkertijd uitgevoerd worden, maar ze kunnen wel op hetzelfde moment proberen de besturing van de CPU te krijgen.

Wanneer twee van zulke processen het gemeenschappelijke geheugen aanspreken, kunnen er nog steeds problemen ontstaan.

Wat is een kritieke sectie ?

De kritieke sectie van een proces is de code die naar gemeenschappelijke data verwijst.

Wat is wederzijdse uitsluiting ?

Als de uitvoering van een proces in zijn kritieke sectie is aangeland, moeten wij ervoor zorgen dat elk ander proces zijn eigen kritieke sectie niet betreedt. Omgekeerd moeten wij ook opletten dat een proces zijn kritieke sectie niet binnentreedt op het moment dat een ander proces in zijn kritieke sectie zit.

We moeten processen gelijktijdig laten uitvoeren, en tegelijkertijd toch voorkomen dat bepaalde delen van die processen, de kritieke secties, parallel worden verwerkt. Wanneer parallelle processen zich toegang verschaffen tot het gemeenschappelijke geheugen, bevatten hun kritieke secties de opdrachten die deze resources aanspreken. De kritieke sectie van een proces is dus de code die naar gemeenschappelijke data verwijst.

Hoe kunnen we voor wederzijdse uitsluiting zorgen ?

Net voor de kritieke sectie van een proces wordt ENTERMUTUALEXCLUSION uitgevoerd en na de kritieke sectie wordt EXITMUTUALEXCLUSION uitgevoerd.

ENTERMUTUALEXCLUSION doet het volgende

- Controleren of een ander proces in zijn kritieke sectie is en, als dat het geval is, wachten.
- Doorgaan met de uitvoering van de kritieke sectie als er geen ander proces in de kritieke sectie bezig is. EXITMUTUALEXCLUSION moet alle andere processen vertellen dat een proces klaar is met de uitvoering van zijn kritieke sectie.

Met ENTERMUTUALEXCLUSION en EXITMUTUALEXCLUSION in een proces kunnen we van wederzijdse uitsluiting verzekerd zijn. Nu alleen nog uitvissen hoe we dit schrijven.

Hoe werkt het algoritme van Dekker ?

Voordat een proces in zijn kritieke sectie gaat, moet het:

1. Zijn bezettingsvariabele op true zetten. Dit betekent dat het probeert zijn kritieke sectie in te gaan.
2. Controleren of het andere proces in zijn kritieke sectie is of probeert daarin te komen. Is dat niet het geval: kritieke sectie ingaan. Anders: stap 3
3. Wachten als het andere proces aan de beurt is om zijn kritieke sectie uit te voeren. De bezettingsvariabele op false zetten en wachten tot het ander proces zijn kritieke sectie verlaat.
4. In het geval dat het huidige proces aan de beurt is om zijn kritieke sectie uit te voeren – als het ander proces toch in zijn kritieke sectie is: wachten tot het deze verlaat. Probeert daarentegen het andere proces ook zijn kritieke sectie in te gaan, dan moet het wachten, zodra het ontdekt dat het huidige proces aan de beurt is. In die situatie: de kritieke sectie ingaan.

Bespreek het algoritme van Peterson

Het algoritme van Dekker lost het probleem van wederzijdse uitsluiting op, maar gebruikt daarvoor een nogal complex programma, dat moeilijk te volgen is en waarvan de juistheid lastig is te bewijzen.

De globale variabele **flag** duidt de positie van elk proces ten aanzien van wederzijdse uitsluiting aan. De globale variabele **turn** lost de conflicten van gelijktijdigheid op.

De wederzijdse uitsluiting blijft hier behouden → wanneer P0 de flag[0] op true instelt, kan P1 zijn kritieke sectie niet uitvoeren. Bevindt P1 zich al in zijn kritieke sectie, dan geldt dat flag[1] = true en is de uitvoering van de kritieke sectie van P0 geblokkeerd.

Een wederzijdse blokkering wordt echter voorkomen. Veronderstel dat P0 is geblokkeerd in zijn while lus. Dit betekent dat flag[1] true is en dat turn = 1. P0 kan zijn kritieke sectie uitvoeren als flag[1] false wordt of turn 0 wordt.

Toon aan dat de uitputtende gevallen opgelost zijn door het algoritme van Peterson

1. P1 heeft geen interesse in zijn kritieke sectie. Dit geval is onmogelijk, omdat het impliceert dat flag[1] = false.
2. P1 wacht op zijn kritieke sectie. Dit geval is ook onmogelijk, omdat als turn = 1, P1 zijn kritieke sectie kan uitvoeren.
3. P1 gebruikt zijn kritieke sectie herhaaldelijk en monopoliseert daarmee de toegang. Dit kan niet gebeuren, omdat P1 aan P0 een kans moet geven door turn in te stellen op 0 voordat P1 probeert zijn kritieke sectie uit te voeren.

Wat is een semafoor ?

Een semafoor is een integer-variabele die door slechts 2 primitieve operaties kan worden veranderd.

Wat is een primitieve ?

Een primitieve kan niet worden onderbroken; eenmaal begonnen, kan het proces tot het klaar is niet worden onderbroken of opgeschort. Primitieve operaties zijn afhankelijk van het systeemontwerp, daarom moet er bij het ontwerpen van computersystemen al rekening mee gehouden worden.

Wat zijn de primitieve operaties voor een semafoor ?

P en V. Ze zijn als volgt gedefinieerd. Als S een semafoon is:

```
P(S): if (S>0)
      S = S - 1;
      else (proces uitstellen);
```

```
V(S): if (een proces uitgesteld is als gevolg van P(S))
      (hervat een proces)
      else S = S+1;
```

Een proces dat de primitieve P uitvoert, moet misschien wachten (WAIT).

Een proces dat de primitieve V uitvoert, geeft misschien het signaal dat een ander proces kan worden hervat. Het feit dat P en V niet onderbroken kunnen worden is essentieel. Een eenmaal begonnen operatie eindigt zonder onderbreking.

P → Prolaag. "Prolaag" is een woord dat Dijkstra heeft bedacht en het betekent "probeer te verlagen".

V → Verhoog

Wat is een ander belangrijk voordeel van semaforen naast eenvoud en elegantie ?

Het algoritme kan gemakkelijk worden uitgebreid voor een situatie geval met n parallelle processen. Als 1 proces P(S) uitvoert, zijn alle andere gedwongen te wachten.

Wat is het grondbeginsel van semaforen ?

- Twee of meer processen kunnen samenwerken d.m.v. eenvoudige signalen, waarbij een proces kan worden gedwongen te stoppen op een opgegeven plaats totdat het een specifiek signaal heeft ontvangen.
- Voor het signaleren worden speciale variabelen gebruikt, zogenaamde semaforen

Welke 3 bewerkingen zijn er gedefinieerd voor semaforen ?

1. Een semafoor kan worden **geïnitialiseerd op een niet-negatieve waarde**
2. De bewerking **wait** verlaagt de semafoorwaarde. Wordt de waarde negatief, dan wordt het proces dat de opdracht wait uitvoert, geblokkeerd.
3. De bewerking **signal** verhoogt de semafoorwaarde. Is de waarde niet positief, dan wordt een proces dat is geblokkeerd door een bewerking wait gedeblokkeerd.

Wanneer is een semafoor sterk en wanneer zwak ?

Bevat de definitie van een semafoor een FIFO-strategie, dan wordt dit een **sterke** semafoor genoemd. Als niet is vastgelegd in welke volgorde processen uit de wachtrij worden verwijderd, is er sprake van een **zwakke semafoor**.

Hoe werkt het algoritme van wederzijdse uitsluiting met semaforen ?

Er zijn n processen die worden aangegeven in de array P(i). In elk proces wordt een wait uitgevoerd vlak voor de kritieke sectie. Wordt de waarde s negatief, dan wordt het proces opgeschort. Is de waarde 1, dan wordt deze verlaagd tot 0 en voert het proces onmiddellijk zijn kritieke sectie uit. Omdat s niet meer positief is, kan geen enkel ander proces zijn kritieke sectie uitvoeren. De semafoor wordt geïnitialiseerd op 1. Daardoor kan het eerste proces dat een wait uitvoert, onmiddellijk zijn kritieke sectie binnengaan, waarbij de waarde van s gebracht wordt op 0. Elk proces dat probeert zijn kritieke sectie te betreden, merkt dat deze bezet is en wordt geblokkeerd, waarbij de waarde van s met 1 vermindert. Nog meer processen kunnen proberen toegang te krijgen: al deze pogingen leiden tot een verdere daling van de waarde van s. Verlaat het proces dat zijn kritieke sectie het eerst heeft gestart, de kritieke sectie, dan wordt s verhoogd en wordt een van de geblokkeerde processen die geblokkeerd zijn voor de semafoor en in de toestand gereed geplaatst. Wordt het daarna ingeroosterd door het besturingssysteem, dan kan het zijn kritieke sectie uitvoeren.

Hoe worden semaforen geïmplementeerd ?

1. **Mogelijkheid 1:** Implementeren in hardware of firmware
2. **Mogelijkheid 2:** Softwarebenadering zoals algoritme van Dekker of Peterson → leidt tot een aanzienlijke overhead in de verwerking
3. **Mogelijkheid 3:** Het gebruiken van een in hardware ondersteund mechanisme voor wederzijdse uitsluiting zoals bijvoorbeeld het gebruik van een instructie test and set waarbij de semafoor weer een datastructuur heeft en een nieuwe integer als component s.flag bevat.
4. **Mogelijkheid 4:** Bij een systeem met 1 processor is het mogelijk interrupts te verbieden tijdens de bewerking wait en signal

Wat is een monitor ?

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

Om problemen te vermijden moet wederzijdse uitsluiting echter verplicht zijn. Oplossing hiervoor is: de kritieke secties in een gebied te plaatsen waartoe maar 1 proces tegelijk toegang heeft. Dan gebruiken de processen de code op een manier die automatisch wederzijdse uitsluiting afdwingt. Voor de speciale gebieden gebruiken we de term monitors.

Een monitor is een constructie die code kan bevatten die naar gemeenschappelijke gebruikte data verwijst. Oppervlakkig gezien lijkt het een verzameling datatypen, datastructuren en procedures, onder de monitor heading. Maar een monitor is veel meer.

Hij bevat procedures en variabelen, maar de procedures zijn van een bijzonder type. Als parallelle processen verschillende procedures in een monitor aanroepen, dwingt de monitor de processen deze procedures na elkaar uit te voeren. 2 procedures binnen dezelfde monitor kunnen niet tegelijk actief zijn. Door de taal gedefinieerde calling protocollen dwingen dit automatisch af.

Een kritieke sectie kunnen we daarom i.p.v. deze in een proces te coderen als **monitor-procedure** schrijven. De code wordt dan niet geduplicateerd. Wanneer een proces gemeenschappelijke data moet gebruiken, roept het een monitor-procedure aan.

Door een compiler gegenereerde code, die de besturing aan een monitor-procedure overdraagt, wordt wederzijdse uitsluiting gegarandeerd.

Op deze manier is er een aanzienlijk verschil tussen een monitor en een eenvoudige collectie procedures. Een monitor dwingt heel streng wederzijdse uitsluiting af tussen processen die proberen zijn procedures uit te voeren. Om dit verschil te benadrukken wordt een monitor-procedure een **procedure-entry** genoemd.

Via conditionele variabelen kan de monitor processen uitstellen of hervatten om events te synchroniseren.

Monitoren werken het best wanneer ze centraal geïnstalleerd kunnen worden. Omdat alle processen de monitor aanspreken, is deze het middelpunt van alle discussies en analyses. Maar vele systemen hebben geen centrale component.

De monitorconstructie is geïmplementeerd in enkele programmeertalen waaronder Modula-3, Java,... Ook is ze geïmplementeerd als een programmabibliotheek. Dit biedt de mogelijkheid grensels op elk object te plaatsen. Vooral bij zoiets als een verbonden lijst kan het zinvol zijn alle verbonden lijsten te vergrendelen met 1 grenzel, 1 grenzel te gebruiken voor elke lijst of 1 grenzel te gebruiken voor elk element van de lijst.

Wat is synchronisatie ?

Het opleggen van een dwingende volgorde aan events die door concurrente, asynchrone processen worden uitgevoerd.

Wat is een deadlock ?

Een deadlock of een impassestoestand treedt op wanneer 2 of meer processen voor onbepaalde tijd wachten op een gebeurtenis die alleen door 1 van de wachtende processen kan worden veroorzaakt.

Welke 2 methoden zijn er voor het behandelen van een deadlock ?

1. Gebruik één of ander protocol (afspraak) om te garanderen dat het systeem nooit in een deadlock-situatie zal komen.
2. Laat toe dat het systeem in een deadlock-situatie geraakt en los deze dan op

Wat zijn de voornaamste aspecten van deadlock ?

1. **Deadlock preventie:** Het besturingssysteem beperkt het gemeenschappelijk gebruik van resources om deadlock onmogelijk te maken.
2. **Deadlock vermindering:** Het besturingssysteem onderzoekt alle aanvragen voor resources heel nauwkeurig. Ziet het besturingssysteem dat de toewijzing van een resource het risico van deadlock met zich meebrengt, dan weigert het de gevraagde toegang en vermijdt zo het probleem.
3. **Deadlock-signalering:** Als er een deadlock optreedt, moet het besturingssysteem dit kunnen signaleren. Het besturingssysteem zet elk proces in een wachttoestand. Hoe kan het besturingssysteem erachter komen dat dit wachten permanent is ?
4. **Deadlock-herstel:** Wat moet er gebeuren nadat het besturingssysteem een deadlock ontdekt ? De processen moeten daar toch een keer uit bevrijd worden ? Het besturingssysteem moet het oplossen.

Wat is deadlock preventie ?

Een deadlock kan alleen optreden indien er tegelijkertijd aan de volgende 4 voorwaarden is voldaan.

1. **Wederzijdse uitsluiting** (mutual exclusion)
2. **Bezet houden en wachten** (hold and request)
3. **Geen voortijdig ontnemen** (non-preemption)
4. **Wachten in een kring** (circular wait)

Om deadlock te voorkomen zorgen we dat tenminste 1 van de voorwaarden nooit optreedt. Er ontstaan aanzienlijke problemen als we proberen deadlock te voorkomen door een noodzakelijke conditie op te heffen.

Hoe signaleren we een deadlock ?

Eén manier om een deadlock te signaleren, is een resource allocation graf. Dit is een georiënteerd graf die gebruikt wordt om de resource-toewijzingen weer te geven.

Een deadlock kunnen we signaleren door de resource allocation graf te bekijken. Als deze een cyclus bevat, is er een deadlock. Om cycli in een georiënteerd graf te signaleren, heeft het besturingssysteem diverse algoritmen ter beschikking.

VRAGENSTELLING BESTURINGSSYSTEMEN THEORIE

Hoe herstellen we een deadlock situatie ?

Eén mogelijkheid is een proces gewoon maar af te breken en de eraan toegewezen resources verwijderen. 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 resources verwijderd. Het proces verliest alle updates die het met gebruik van deze resources 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 resources. Dit kan overeenkomen met de oorspronkelijke start van het proces, of met een checkpoint. Een checkpoint treedt op wanneer een proces vrijwillig alle resources vrijgeeft. Door het gebruik van checkpoints kan elk proces eventueel verlies van werk echter zo klein mogelijk houden.

Wat is multithreading ?

Multithreading verwijst naar de mogelijkheid van een besturingssysteem binnen een proces meerdere threads of draden te gebruiken voor de uitvoering. De traditionele benadering met één uitvoeringsthread per proces, waarin het concept thread in feite niet bestaat, wordt ook wel een benadering met één thread genoemd.

In een omgeving met multithreading wordt een proces gedefinieerd als beveiligings- en brontoewijzingseenheid. Het volgende is verbonden met processen

- Een **virtuele adresruimte**, die het procesbeeld bevat
- **Beveiligde toegang** tot processors, andere processen (voor de communicatie tussen processen), bestanden en I/O bronnen (apparaten en kanalen).

Binnen een proces kunnen er een of meer threads zijn, elk met het volgende (eigenschappen van threads)

- Een **uitvoeringstoestand** van de thread (actief, gereed,...)
- Een **context** die wordt opgeslagen als de thread niet actief is, een thread kan onder meer worden gezien als een onafhankelijke programmateller die binnen een proces werkt.
- **Een stack** voor de uitvoering
- **Enige statische opslagcapaciteit per thread** voor lokale variabelen
- Toegang tot het geheugen en de bronnen van het bijhorende proces, die wordt gedeeld door alle threads binnen dat proces

Alle threads van een proces delen de toestand en de bronnen van dat proces. Ze bevinden zich in dezelfde adresruimte en hebben toegang tot dezelfde gegevens.

De grootste voordelen van threads hangen samen met de gevolgen voor de prestaties: het creëren van een nieuwe thread binnen een bestaand proces kost veel minder tijd dan het creëren van een geheel nieuw proces en ditzelfde geldt voor het overschakelen tussen 2 threads binnen hetzelfde proces.

Threads verbeteren ook de efficiëntie van de communicatie tussen verschillende actieve programma's. Aangezien threads binnen hetzelfde proces echter geheugen en bestanden delen, kunnen ze rechtstreeks met elkaar communiceren.

Net zoals processen hebben threads uitvoeringstoestanden en kunnen ze met elkaar worden gesynchroniseerd.

Wat is een proces ?

Een proces is een uitvoerbaar deel van een programma, dat in het geheugen geladen wordt en daar instructies doorgeeft naar de processor.

Welke drie soorten processen bestaan er ?

1. Interactieve processen
2. Automatische processen
3. Daemons

Wat is een interactief proces ?

Processen die opgestart en gecontroleerd kunnen worden vanuit een terminal sessie m.a.w. er moet iemand aangemeld zijn op het systeem.

Interactieve processen kunnen zowel in de voorgrond als in de achtergrond draaien.

Foreground processen houden de terminal bezet zolang ze lopen.

Background processen bezetten de terminal niet en er kunnen andere taken uitgevoerd worden

Wat is een automatisch proces ?

Een automatisch proces – ook wel batch-proces genoemd – wacht eerst op uitvoering in een daartoe bestemde map. Vandaar uit worden ze opgeroepen door een programma dat de wachtrij analyseert en de programma's systematisch uitvoert. Het programma dat het eerste in de wachtrij terecht kwam, wordt ook eerst uitgevoerd. De naam van dit systeem is "FIFO".

Twee manieren:

1. **at**
2. **batch**

Wat zijn daemons ?

Dit zijn server processen die continu draaien. Meestal worden ze opgestart wanneer het systeem opstart, waarna ze in de achtergrond wachten tot hun diensten vereist zijn.

Wat is job control ?

Door middel van job control beheer je processen in foreground of background. Een proces in background draaien heeft enkel zin als het over processen gaat die geen input verwachten en veel tijd nodig hebben. Je doet dit door de commandonaam gevolgd door &.

PID: process identification – proces volnummer

Jobnummer: Dit is een nummer dat door de shell gebruikt wordt.

Welke vaste eigenschappen heeft een proces ?

- Het **procesidentificatienummer of PID**: een uniek nummer dat gebruikt wordt om naar het proces te verwijzen.
- Het PID van het proces dat dit proces gestart heeft: parent process ID of **PPID**
- Het zogenaamde **nice number**: de mate van vriendelijkheid van dit proces: laat het nog veel procescapaciteit over aan andere processen, of juist niet ?
- De terminal van waaruit dit proces opgestart werd, als het om een interactief proces gaat. Dit wordt aangeduid met een **tty number**.
- De **gebruikersnaam** van de gebruiker aan wie het proces toebehoort
- De **groepsnaam** van de groep aan wie het proces toebehoort.

Wat bevat het ps commando ?

1. **PID**: het procesidentificatienummer
2. **TTY**: Het terminal type en nummer waaraan het proces verbonden is. Wij gebruiken pts, pseudo-terminals, in tegenstelling tot echte terminals waarbij je een toetsenbord en een scherm hebt, waarmee je niets anders kan doen dan 1 enkele shell openen, in een tekstuele omgeving (te vergelijken met DOS vroeger). Pseudo-terminals zijn terminal vensters in een grafische omgeving, of verbindingen vanop een netwerk..
3. **TIME**: Een relatieve indicatie van de tijd die het aantal processorcycli dat het process al verbruikt heeft. Gewone processen van gebruikers verbruiken slechts een klein deel van de totale processorkracht
4. **CMD**: De naam van het commando

Het ps -ef commando is uitgebreider. Wat bevat dit ?

1. **UID**: De naam van de gebruiker die het proces opstartte
2. **PID**: Het procesidentificatienummer
3. **PPID**: Procesidentificatienummer van het *parent process, het proces dat dit proces opstartte*
4. **TTY**: De terminal waaraan het proces verbonden is, “?” wil zeggen dat het proces niet aan een terminal verbonden is
5. **CMD**: de naam van het commando

Wat doet het top commando ?

Het top-commando geeft ongeveer dezelfde informatie als ps -ef, maar het wordt om de 5 seconden opgefrist.

Bovendien hebben we hier al automatisch een zekere vorm van sorteren; de zwaarste processen, dat wil zeggen de processen die het meeste processortijd verbruiken, worden bovenaan in de lijst getoond. We krijgen ook niet alle processen te zien. Al naargelang de grootte van je terminal venster wordt de lijst ingekort. We krijgen dus een “top” van de processen te zien.

Wat doet het uptime commando ?

De output van het **uptime** commando, met daarin informatie over hoe lang het systeem al draait, hoeveel gebruikers er verbonden zijn en wat de belasting is.

Het aantal processen en de status ervan: er draait altijd slechts 1 proces tegelijk op de CPU, terwijl de andere in een wachtrij staan.

De belasting van de processor(s): moet de processor veel berekeningen maken, dan is de belasting hoog.

Gebruik van het geheugen: alle programma's die actief zijn, nemen een plaats in op het geheugen.

Gebruik van de swap space (het virtuele geheugen): als er teveel programma's draaien, wordt alle beschikbare plaats in het geheugen opgevuld. Een speciale plek op de harde schijf wordt dan gebruikt als extra geheugen.

Wat is het pstree commando ?

Dit commando toont de samenhang van processen. De meeste processen stammen af van **systemd**, het initiële proces waarmee het systeem gestart wordt.

Wat doet het time commando ?

Het time commando werkt als een chronometer. Het geeft aan hoeveel uur, minuten en seconden een opdracht duurt om uit te voeren. Je gebruikt het door het te plaatsen voor het commando dat je wil uitvoeren.

Wat doet het w commando ?

Het commando **w** geeft een overzicht van de aangemelde gebruikers en hun activiteit(en)

Wat doet het renice commando ?

Het **renice** commando zorgt dat het proces minder belastend is. Hiervoor moet je het proces niet onderbreken.

Wat doet het kill commando ?

Kill stopt een proces omdat het hangt, op hol slaat of te veel of te grote bestanden aanmaakt. Als je daar toe gelegenheid hebt, probeer dan eerst de zachte manier een stuur een **SIGTERM** (waarde 15) signaal. Dit instrueert het proces om af te handelen waar het mee bezig is volgens de procedures beschreven in het programma. Op die manier wordt alle rommel opgeruimd en worden er geen bestanden beschadigd.

Hoe gebruik je dit commando nu ?

1. Zoek het procesidentificatienummer van het proces dat je wilt stoppen met behulp van **ps -ef**
2. Gebruik het commando **kill -15 PID_nummer**
3. Ga na met **ps** of het proces echt wel weg is
4. Van sommige processen raak je echter niet zo makkelijk af. Probeer dan in eerste instantie **kill -2**, een interruptiesignaal. Dat is hetzelfde als een programma onderbreken met **CTRL+C** als het in de voorgrond draait.
5. Als dat ook niet helpt, zit er niet veel anders op dan het sterkste signaal te sturen, een **SIGKILL** met **kill -9**
6. Kijk in elk geval altijd na met **ps** of het stoppen gelukt is

Wat doet het xkill commando ?

Je kan grafische programma's die vasthangen proberen te stoppen met **xkill**. Na het ingeven van dit commando, verandert de muispijl in een doodshoofd. Beweeg het doodshoofd over het venster van het programma dat je wilt stoppen en klik met de linker muistoets.

Op welke 3 manieren kan je uitgestelde taken inplannen ?

1. Een korte tijd wachten en daarna de taak uitvoeren, gebruik makend van het **sleep** commando. Het tijdstip van uitvoering hangt af van het tijdstip waarop de taak gepland werd
2. De taak uitvoeren op een welbepaald tijdstip met het **at** commando. Het tijdstip van uitvoering is niet afhankelijk van het tijdstip van planning.
3. Een taak steeds opnieuw uitvoeren, maandelijks, wekelijks, dagelijks, elk uur of elke minuut, door gebruik te maken van de **cron** faciliteit

Wat doet het sleep commando ?

Het enige dat sleep doet, is wachten. Standaard wordt de wachttijd uitgedrukt in seconden

Wat doet het at commando ?

Geef het **at** commando in, gevolgd door het tijdstip waarop de geplande taak uitgevoerd moet worden. Daarmee kom je in de **at** omgeving, gekenmerkt door de **at** prompt. Hier geef je het commando of de commando's in die gepland moeten worden. Sluit af met **CTRL+D** en de taak wordt gepland.

Hoe krijg ik een overzicht van alle jobs die ge-at zijn ?

Je kan een overzicht krijgen van alle jobs met het commando **atq**. Met het **atrm** commando kan je de job verwijderen. Gebruik het **jobnummer** uit de eerste kolom van de output van **atq** als argument

Hoe werkt cron ?

De **cron daemon** draait constant op je systeem. Deze dienst gaat elke minuut na of er taken uit te voeren zijn voor de gebruikers of voor de diensten die op een systeem draaien. De taken worden opgeslagen in zogenaamde **crontabs** (tabellen).

Elke gebruiker kan een crontab hebben, waarin elke lijn een taak voorstelt die regelmatig herhaald moet worden.

Verder is er ook nog een crontab waarin de **systeem-specifieke taken** vernoemd worden, zoals bijvoorbeeld:

- Dagelijks de index maken waarvan het **locate** commando gebruik maakt
- Dagelijks nagaan of er **updates** zijn voor de software op het systeem
- Er dagelijks voor zorgen dat **logbestanden**, waarin informatie wordt opgeslagen over wat er allemaal op het systeem gebeurd is, niet te groot worden
- Dagelijks en wekelijks een index maken van alle **man pagina's**, zodat apropos en whatis kunnen werken.

Andere taken kunnen zijn: het maken van backups, rapporten opmaken en doorsturen, systeeminformatie analyseren en doormailen naar de administrator, herinneringsbrieven mailen, enzovoort. *Alle taken die periodiek uitgevoerd moeten worden, komen voor opname in het cron systeem in aanmerking.*

De crontabs van het systeem vind je in de **/etc** map, die van de gebruikers is **/var/spool/cron/crontabs**, maar die is niet toegankelijk voor niet-geprivilegerde gebruiker. In **/var/spool/cron** vind je ook nog atjobs en atspool, omdat de at jobs onder de verantwoordelijkheid van de cron daemon vallen.

Hoe wordt een nieuw proces aangemaakt ?

Een nieuw proces wordt aangemaakt doordat een bestaand proces een exacte kopie van zichzelf maakt. Dit child process is eigenlijk net hetzelfde als het ouderproces, enkel het procesidentificatienummer verschilt. Deze procedure heet men een **fork** (letterlijk: een vork of splitsing).

Na de fork wordt de geheugenruimte van het kindproces overschreven met de nieuwe procesdata: het commando dat gevraagd werd, wordt in het geheugen geladen. Dit noemt men **exec**. Het geheel wordt **fork-and-exec** genoemd.

Wat is de rol van systemd ?

Zoals je kan zien aan de output van het **ps tree** commando, hebben veel processen systemd als ouderproces, terwijl dat helemaal niet mogelijk is.

Veel programma's "demoniseren" hun kindprocessen, zodanig dat die kunnen blijven draaien als de ouder stopt. Het systemd proces neemt de rol van peetvader van zulke processen: als de ouder sterft, vallen ze onder de verantwoordelijkheid van systemd.

Heel af en toe wil het nog wel eens mislopen met de "adoptie" van processen. Een proces dat geen ouderproces heeft, noemt men een **zombie**. Het systeem heeft geen vat meer op zo'n zombieproces, het blijft in het geheugen hangen tot je de computer herstart.

Wat gebeurt er als een proces beëindigd wordt ?

Wanneer een proces normaal eindigt, geeft het een code, de **exit status**, door aan de ouder. Als alles goed verlopen is, is de exit status nul.

De waarde van de exit status van shell commando's wordt opgeslagen in een speciale variabele, aangeduid met **\$?**. Met het **echo** commando kan je de inhoud van deze variabele bekijken.

Processen eindigen omdat ze een signaal krijgen. Je kan verschillende signalen naar een proces sturen. Om dat te doen gebruik je het **kill** commando.