Besturingssystemen

1 Inleiding

1.1 Wat is een besturingssysteem

1.1.1 Definitie

Besturingssysteem = programma dat het mogelijk maakt de hardware van een computer te gebruiken.

1.1.2 Functies

Enkele taken van een besturingssysteem:

- programma's de mogelijkheid geven informatie op te slaan en terug te halen;
- programma's afschermen van specifieke hardwarezaken;
- de gegevensstroom door de componenten van de computer regelen;
- programma's in staat stellen te werken zonder door andere programma's te worden onderbroken;
- onafhankelijke programma's de gelegenheid geven tijdelijk samen te werken en informatie gemeenschappelijk te gebruiken;
- reageren op fouten of aanvragen van de gebruiker;
- een tijdsplanning maken voor programma's die resources willen gebruiken.

1.2 Historisch overzicht

Eerste computers → geen OS

Jaren 50 → eenvoudige OS:

- sequentieel opladen en opstarten van programma's
- alle bronnen bruikbaar door slechts 1 programma

Begin jaren 60 → geavanceerder OS:

- verscheidene programma's konden tegelijkertijd opgeslagen worden in het geheugen
- beurtelingse uitvoering van programma's
- gemeenschappelijke bronnen

Midden jaren 60 → verschillende computers van hetzelfde type gebruikten één OS

Begin jaren 70 → OS kan computers met meer dan 1 processor aan

Begin jaren 80 → gemeenschappelijk gebruik van informatie door verschillende computers/systemen

Jaren 90 → distributed computing, parallelle verwerking, …

1.3 Soorten besturingssystemen

1.3.1 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.

1.3.2 Multitasking (single-user)

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.

1.3.3 Multi-user-systemen

Multiuser-systemen, ook wel multiprogrammering-systemen 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.

Er bestaan verscheidene soorten multiuser-computers, afhankelijk van de soorten programma's die ze aankunnen:

1.3.3.1 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.

1.3.3.2 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. Batch-gebruikers verschillen van interactieve gebruikers omdat zij geen directe respons verwachten. Bij scheduling houdt het besturingssysteem hiermee rekening.

1.3.3.3 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.

1.3.4 Virtuele machines

1.3.4.1 Definitie

Een virtuele machine is een computerprogramma dat een computer nabootst, waar andere programma's op kunnen worden uitgevoerd. Deze techniek wordt gebruikt om de overdraagbaarheid

van programmatuur te verbeteren, en ook om het gelijktijdig gebruik van de computer door verschillende gebruikers robuuster te maken.

1.3.4.2 Programmeertaal-specifiek

Een virtuele machine voor een programmeertaal biedt een *abstractielaag* voor de werkelijke computer: een verzameling basisfuncties waar programma's in de programmeertaal gebruik van moeten maken om de functies van de computer aan te spreken. Dit is precies wat een besturingssysteem doet.

De reden dat van een virtuele machine wordt gesproken is dat de virtuele machine zelf vaak een programma is dat wordt uitgevoerd op een bestaand besturingssysteem. De virtuele machine is de verbindende laag tussen de uitgevoerde code en de computerhardware (zoals de microprocessor) waarop het uiteindelijke programma wordt uitgevoerd, eventueel via een ander besturingssysteem. De op een virtuele machine uitgevoerde code spreekt niet direct de 'echte' hardware aan, maar gebruikt alleen de functies die worden aangeboden door de virtuele machine. Wanneer voor een bepaald platform (hardware en eventueel besturingssysteem) een virtuele machine gemaakt is, kan elk programma dat voor deze virtuele machine geschreven is, worden uitgevoerd.

De implementatie van een virtuele machine kan in elke andere programmeertaal gebeuren. Op deze manier wordt platform-onafhankelijkheid bereikt: programma's kunnen worden uitgevoerd op elk systeem waarvoor de virtuele machine is geïmplementeerd. Overdraagbaarheid wordt een kwestie van het implementeren van een virtuele machine, in plaats van een probleem dat voor elk programma afzonderlijk moet worden opgelost.

1.3.4.3 Emulator

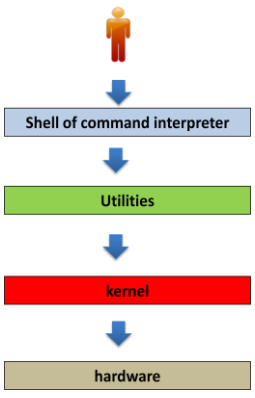
Een virtuele machine die de hardware van de computer emuleert, is niet gericht op een programmeertaal, maar emuleert een fysieke computer (dat wil zeggen de processor en andere hardware), zodanig dat een bestaand besturingssysteem op deze emulatie kan draaien, alsof het een fysieke computer betreft.

1.3.5 Virtuele monitor

Er bestaat ook nog een ander type systeem, de virtual machine monitor. Deze geeft elke gebruiker de mogelijkheid met een uniek beeld van de computeromgeving te werken. Bij gebruik van een monitor kunnen verscheidene besturingssystemen tegelijkertijd bestaan. Elke gebruiker kan dan opgeven welk systeem hij wenst te gebruiken. Dit geeft elke gebruiker de illusie van een eigen computer (virtuele machine). Voor elke gebruiker bestaat de virtuele machine echt.

1.4 Concepten van een besturingssysteem

De gebruiker communiceert met de bovenste laag. Deze bestaat uit routines van het besturingssysteem die zijn ontworpen om op opdrachten van een gebruiker te reageren. We noemen deze laag de shell of de command interpreter. Dit is het deel van het besturingssysteem waarmee de gebruiker het meest vertrouwd is.



In werkelijkheid is het echter niet de shell die de opdracht van de gebruiker uitvoert. De reden daarvoor is dat veel opdrachten eigenlijk erg ingewikkeld zijn. Zo is het mogelijk dat er voor een eenvoudige opdracht die het besturingssysteem vraagt veel moet gebeuren.

De laag utilities bevat vele routines die voor deze dingen zorg dragen.

De laatste laag is de kernel of kern. Dit is het hart van het besturingssysteem. Deze laag bevat de routines die het vaakst worden gebruikt en waar het meest op aan komt. Wanneer een gebruiker een opdracht geeft, verzorgen de utilities het grootste deel van de controle en de voorbereiding die voor de uitvoering nodig zijn.

1.4.1 Processen

Een of meer reeksenopdrachten die door een besturingsprogramma worden beschouwd als een werkeenheid.

1.4.2 Resources

Het besturingssysteem moet open staan voor de behoeften van een proces. In de eerste plaats spreken processen resources aan.

1.4.2.1 Geheugen

Een proces heeft geheugen nodig waarin het zijn instructies en gegevens kan opslaan. Geheugen is 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.

1.4.2.2 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.

1.4.2.3 Devices

Randapparatuur of devices zijn onder andere printers, tapedrives en disk-drives. Net als bij de CPU zijn er gewoonlijk meer gebruikers dan devices. 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.

1.4.2.4 Files

Het besturingssysteem wordt verondersteld snel een bepaalde file te kunnen lokaliseren. Ook wordt verwacht dat het snel een bepaald record in die file kan lokaliseren.

1.4.3 Concurency

Processen zijn meestal niet onafhankelijk, processen zijn concurrent.

OS regelt in welke volgorde processen afgehandeld worden = synchronisatie

1.4.4 Ontwerpcriteria

1.4.4.1 Consistentie

Als het aantal processen, dat van de computer gebruik maakt, vrijwel constant blijft, hoort dat ook voor de respons te gelden.

1.4.4.2 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.

1.4.4.3 Overdraagbaarheid

Dit houdt in dat het besturingssysteem op verscheidene soorten computers werkt. Overdraagbaarheid geeft de gebruiker meer flexibiliteit.

1.4.4.4 Compromissen

Al deze ontwerp-criteria zijn belangrijk; helaas is het meestal onmogelijk om aan alle te voldoen. Vaak moet het ene criterium worden opgeofferd ten gunste van het andere. De ontwerpers moeten de omgeving waarin het besturingssysteem werkt, grondig kennen. Op die manier kunnen ze bepalen welke criteria voor de gebruiker het belangrijkst zijn.

2 Scheduling

2.1 Noodzaak tot scheduling

Om efficiënt middelen (bronnen, resources) in te zetten om de taken (opdrachten, jobs) uit te voeren.

Bij multiprogrammering (multitasking) moet het OS efficiënt aan de verschillende behoeften van vele gebruikers voldoen. Processen moeten resources benaderen en binnen een redelijke tijd uitgevoerd worden.

Scheduling is dus een belangrijk concept in het ontwerp van multitasking- en multiprocessing-besturingssystemen en in het ontwerp van een realtimebesturingssysteem.

Scheduling verwijst dus naar de manier waarop processen prioriteiten worden gegeven. Deze taak wordt uitgevoerd door software die bekend staat als een scheduler.

2.2 Doelstellingen van scheduling

- doelmatigheid en tevredenheid van de gebruiker
- resources moeten effectief gebruikt worden
- op een snelle en rendabele manier

2.2.1 Doorvoersnelheid

#processen/tijdseenheid

Lage doorvoersnelheid: weinig processen
Hoge doorvoersnelheid: veel processen

2.2.2 Responstijd

Interactieve gebruikers: snelle responstijd
Batch gebruikers: redelijke responstijd

→ responstijd kan vergroot worden

2.2.3 Consistentie

Eisen aan het systeem:

- responstijd moet ca. gelijk zijn
    - Als het sterk varieert:
        - Problemen met werkindeling
        - Gebruikers weten niet wat ze mogen verwachten

2.2.4 Houdt de CPU aan het werk

OS moet resources aan het werk houden.
→ ideaal geval: OS houdt elke processor aan het werk zonder deze zwaar te belasten

2.2.5 Prioriteiten

Elk proces krijgt een prioriteit, hoe hoger, hoe belangrijker

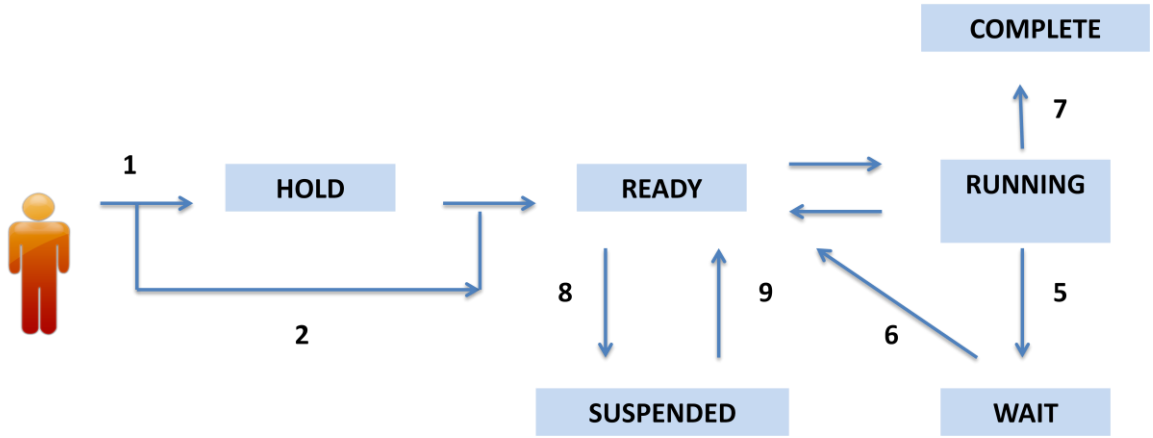
2.2.6 Realtime sysstemen

Snelle respons voor besturing van doorgaand proces → hoogste prioriteit
Rechtvaardigheid e, systeemefficiëntie → lage prioriteit

2.2.7 Besluit

Scheduling is een ingewikkelde zaak. Er moet rekening gehouden worden met eerlijkheid, behoefte van het proces, systeemefficiëntie, hardware…

2.3 Systeembeeld van een proces



- HOLD → is aangeboden
- READY → gereed om uit te voeren
- RUNNING → wordt uitgevoerd en onder besturing van de CPU
- WAIT → wacht op iets
- SUSPENDED → is opgeschort (idle t.o.v. de CPU)
- COMPLETE → volledig afgewerkt

2.3.1 Proces Control Block

Process Control Blocks (PCB) bevatten:

- proces-ID
- procestoestand
- maximale en actuele looptijd
- huidige resources en limieten
- procesprioriteit
- opslaggebieden
- locatie van de code of de segmenttabel van een proces

2.3.2 Niveaus van scheduling

- high-level scheduling(job scheduling): regelt de toestandsovergangen 1,2 en 7
- scheduling op middelniveau (intermediate level): regelt de toestandsovergangen 5,6,8 en 9
- Low-level scheduling: regelt de toestandsovergangen 3 en 4

2.4 Strategieën voor low-level scheduling

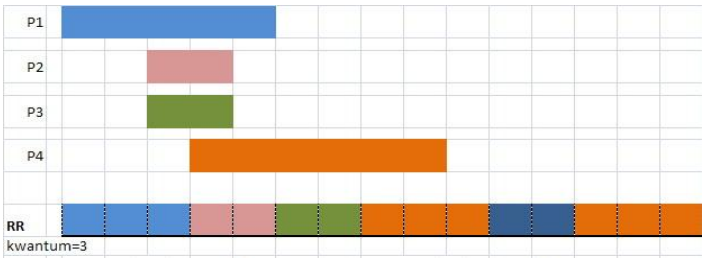
2 hoofdcategorieën:

- **algoritme voor preëmptive scheduling**
  - processen worden onderbroken om een ander proces te hervatten
- **algoritme voor nonpreëmptive scheduling**
  - een ander proces kan pas gestart worden nadat het huidige proces klaar is

Criteria voor scheduler:

- CPU-gebruik
- througput (doorvoersnelheid)
- wachttijd: som van alle tijden die een proces moet wachten
- responstijd

2.4.1 Round Robin

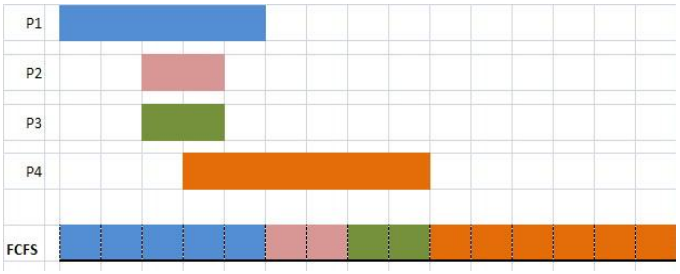


Hier wordt gebruik gemaakt van, wanneer dit overschreden wordt, zal de scheduler het proces onderbreken en een volgend proces inladen.

Round Robin wordt vaak gebruikt als er veel interactie is.

Te groot quantum: lijkt op FIFO
Te klein quantum: veel overhead door de vele switches tussen processen

2.4.2 First-in-First-Out scheduling (FIFO)



Wanneer een proces als eerste de CPU vraagt zal hij die ook krijgen, waarbij de andere processen die erna komen zullen moeten wachten.

FIFO is interessant bij lange processen en is minder geschikt als er interactie is.

Weinig overhead door de weinige wissels in processen.

Een FIFO-scheduler kan deel uitmaken van een ingewikkelder methode zoals bijvoorbeeld bij systemen die zowel batch-gebruikers als interactieve gebruikers hebben.

Een manier om hybride methode van scheduling te implementeren is met batch-partities (= virtuele geheugenconstructie die 1 batch-proces bevat).

2.4.3 Multilevel feedback queues (MFQ)

Hierbij lijkt de scheduling-methode op Round-Robin als er veel I/O-activiteit is en op FIFO wanneer er weinig of geen I/O-activiteit is.

De beste scheduling-methode is afhankelijk van de soorten processen in de ready-toestand en het MFQ is gevoelig voor wijzigingen in die activiteiten (**adaptieve methode**).

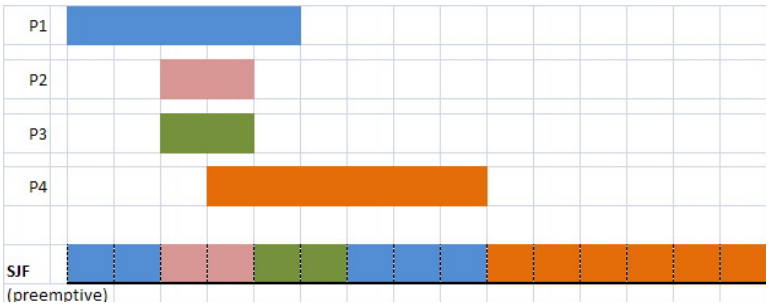
Als je in de wait terecht komt ga je naar een hogere queue (hoge prioriteit), als je quantum is opgebruikt dan ga je naar een lagere queue (lagere prioriteit).

→rekenintensief in lage queue
→I/O in hoge queue

Starvation mogelijk bij CPU gebonden (=rekenintensieve) processen.

2.4.4 Shortest-job-first-scheduling (SJF)

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



- Shortest Remaining Job Next (SRJN)
  → preëmptieve versie van SJF
- Shortest Job First (SJF) → hier zal de scheduler het proces met de kleinste lengte uitvoeren

Lang proces kan in de niet preëmptieve versie eeuwig wachten.

Het is ook bijna onmogelijke om te bepalen hoe lang iets nog zal duren.

Besturing wordt afgenomen als er een korter proces beschikbaar is.

2.4.5 Starvation

Wanneer een heel lang proces nooit uitgevoerd zal worden, noemen we dit starvation.

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

Mogelijkheden:

- Negeren: in de meeste gevallen lost het probleem zichzelf toch op
- opschorten van een aantal READY-processen
- Periodiek prioriteiten opnieuw berekenen

Bij Round Robin en FIFO kan geen starvation optreden.

2.4.5 Overzicht van de scheduling-methoden

	Round Robin	**FIFO**	**MFQ**	**SJF**	**SRJN**
Doorvoer-snelheid	kan laag zijn als quantum te klein is		kan laag zijn als de quanta te klein zijn	hoog	hoog
Responstijd	kortste gemiddelde responstijd, als quantum juist is gekozen	kan gebrekkig zijn, vooral als een lang proces de besturing over de CPU heeft	goed voor I/O-gebonden processen, maar kan gebrekkig zijn voor de CPU-gebonden processen	goed voor korte processen, maar kan gebrekkig zijn voor langere processen	goed voor korte processen, maar kan gebrekkig zijn voor langere processen
Overhead	laag	de laagste van alle methoden	kan hoog zijn; ingewikkelde datastructuren en routines zijn nodig om na elke reschedule de juiste queue te vinden	kan hoog zijn; vereist een routine om voor elke reschedule de kortste job te vinden	kan hoog zijn; vereist een routine om voor elke reschedule de minimale resterende tijd te vinden
CPU-gebonden processen	geen onderscheid tussen CPU-gebonden en I/O-gebonden processen	geen onderscheid tussen CPU-gebonden en I/O-gebonden processen	krijgt lage prioriteit als de I/O-gebonden processen aanwezig zijn	geen onderscheid tussen CPU-gebonden en I/O-gebonden processen	geen onderscheid tussen CPU-gebonden en I/O-gebonden processen
I/O-gebonden processen	geen onderscheid tussen CPU-gebonden en I/O-gebonden processen	geen onderscheid tussen CPU-gebonden en I/O-gebonden processen	krijgt hoge prioriteit om I/O-processors actief te houden	geen onderscheid tussen CPU-gebonden en I/O-gebonden processen	geen onderscheid tussen CPU-gebonden en I/O-gebonden processen
Onbepaald uitstel	treedt niet op	treedt niet op	kan optreden bij CPU-gebonden processen	kan optreden bij processen met lange geschatte runtijden	kan optreden bij processen met lange geschatte runtijden

3 Concurrency – parallelle processen

3.1 Wat is concurrency

- 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).

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.

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 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.

3.2 Wederzijdse uitsluiting (mutual exclusion)

3.2.1 Concurrency met meerdere processors

Niet alleen processen, maar ook activiteiten binnen één proces kunnen gelijktijdig worden uitgevoerd. Als parallelle processen niets gemeenschappelijk gebruiken, is er geen probleem. De moeilijkheden ontstaan wanneer de processen het gemeenschappelijke geheugen aanspreken.

3.2.2 Concurrency met 1 processors

Hier zijn er 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.

3.2.3 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 en ander proces in zijn kritieke sectie zit. Dit noemen we wederzijdse uitsluiting (mutual exclusion).

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.

3.3 Het programmeren van wederzijdse uitsluiting

3.3.1 Eerste poging

We declareren een booleaanse 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.

3.3.2 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 een globale variabele “welk”, met een waarde van of 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.

3.3.3 Derde poging

Een zwakke plek in de tweede poging is het 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 “bezet”, en definieert deze dan pas als true. 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.

3.4 Het algoritme van Dekker

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.

```
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
   ...
   // kritieke sectie van proces1
   ...
   // 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
         }  // if
   // einde van
ENTERMUTUALEXCLUSION
   ...
   // kritieke sectie van proces2
   ...
   // begin van
EXITMUTUALEXCLUSION
   welk = 1;
   bezet2 = 0;  // false
   // einde van
EXITMUTUALEXCLUSION
   ...
}  // proces2
```

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.

De logica achter 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: 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 andere 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 andere 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.

3.5 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.

```
boolean flag [2];
int turn;
void P0()
{  while (true)
   {  flag[0] = true;
      turn = 1;
      while (flag[1] && turn == 1)
            /* do nothing */;
      /* 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);
}
```

- 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 kritiek 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.

3.6 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 ander 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.

3.7 Semaforen

Een semafoor is een onderdeel van een synchronisatie-mechanisme voor parallelle of gedistribueerde programma's ontworpen door Edsger Dijkstra.

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. Men kan aan elke coördinatie-eis, hoe complex ook, voldoen door de keuze van de juiste signaalstructuur. Voor het signaleren worden speciale variabelen gebruikt, zogenoemde semaforen.

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 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 geblokkeerd.

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

```
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;
   }
```

De primitieven **wait** en **signal** worden verondersteld atomair te zijn. D.w.z. ze kunnen niet worden onderbroken en elke routine kan worden behandeld als 1 ondeelbare stap.

Definitie van binaire semafoorprimitieven:

```
struct binary_semaphore {
   enum (zero, one) value;
   queType queue;
};
```

```
void waitB(binary_semaphore s)
{   if (s.value == 1)
        s.value = 0;
    else
    {   place this process in s.queue;
        block this process;
    }
}
void signalB(binary_semaphore s)
{   if (s.queue.is_empty())
        s.value = 1;
    else
    {   remove a process P from s.queue;
        place process P on ready list;
    }
}
```

Voor zowel semaforen als binaire semaforen wordt een wachtrij (queue) gebruikt, die alle processen bevat die op de semafoor wachten. Hierbij speelt de vraag in welke volgorde de processen uit de wachtrij worden verwijderd. De meest eerlijke strategie is FIFO.

Bevat de definitie van een semafoor deze 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**.

3.7.1 Het algoritme van wederzijdse uitsluiting:

```
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));
}
```

3.7.2 Implementatie semaforen

1. implementeren in hardware of firmware
2. softwarebenadering zoals algoritme van Dekker of Peterson => leidt tot een aanzienlijke overhead in de verwerking
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. (zie slide 39)
4. bij een systeem met 1 processor is het mogelijk interrupts te verbieden tijdens de bewerkingen wait en signal

3.7.3 Testset

In deze implementatie heeft de semafoor weer een datastructuur, maar bevat een nieuwe integer als component, s.flag. Het gebruik van een vorm van actief wachten (busy waiting) is hier onvermijdelijk. De bewerkingen wait en signal zijn echter relatief kort, dus de tijd die wordt besteed aan wachten, is minimaal.

3.7.4 Interrupt

Bij een systeem met 1 processor is het mogelijk interrupts te verbieden tijdens de bewerkingen wait en signal. Wederom betekent de relatief korte duur van deze bewerkingen dat deze benadering aanvaarbaar is.

3.8 Uitwisselen van berichten

Bij de interactie tussen processen moet aan twee fundamenten eisen worden voldaan: **synchronisatie** en **communicatie**. Processen moeten worden gesynchroniseerd om wederzijdse uitsluiting af te dwingen; samenwerkende processen moeten soms informatie uitwisselen.

Een benadering die voorziet in beide functies, is het uitwisselen van berichten (message passing). Het uitwisselen van berichten heeft het bijkomende voordeel dat dit kan worden gebruikt in gedistribueerde systemen en in systemen met een of meer processors en bij gedeeld geheugen.

Er bestaan tal van manieren voor het uitwisselen van berichten. De feitelijke functie voor het uitwisselen van berichten wordt meestal geformuleerd als een stel primitieven:

- **send (bestemming, bericht)**
- **receive (bron, bericht)**

Dit is minimale aantal bewerkingen dat nodig is bij het uitwisselen van berichten door processen.

Een proces verstuurt met de primitieve verzenden (send) informatie in de vorm van een bericht (message) aan een proces dat wordt aangeduid als bestemming of doel (destination). Een proces ontvangt informatie met de primitieve ontvangen (receive), waarbij de bron (source) van het verzendende proces en het bericht worden aangegeven.

3.8.1 Synchronisatie

Het uitwisselen van een bericht tussen 2 processen vraagt om een zekere mate van synchronisatie tussen beide processen: de ontvanger kan pas een bericht ontvangen als dat is verzonden door een ander proces. Bovendien moeten we opgeven wat met een proces gebeurt nadat het de primitieve send. Wordt een primitieve send of receive heeft uitgevoerd.

Beschouw eerst de primitieve send. Wordt een primitieve send uitgevoerd in een proces, dan zijn er 2 mogelijkheden: het verzendende proces wordt geblokkeerd totdat het bericht is ontvangen of het wordt niet geblokkeerd. Op overeenkomstige wijze zijn er 2 mogelijkheden wanneer een proces een primitieve receive uitvoert:

1. Is eerder al een bericht verzonden, dan wordt het bericht ontvangen en wordt de uitvoering voortgezet.
2. Is er geen wachtend bericht, dan (a) wordt het proces geblokkeerd totdat het bericht aankomt of (b) wordt het proces voortgezet en staakt het zijn poging een bericht te ontvangen.

Zowel de zender als de ontvanger kunnen dus geblokkeerd of niet geblokkeerd raken. Drie combinaties komen veel voor, hoewel in de meeste systemen maar 1 of 2 combinaties worden geïmplementeerd:

1. **Blokkerende send** en **blokkerende receive**. Zowel de zender als de ontvanger worden geblokkeerd totdat het bericht is ontvangen. Dit wordt soms een rendez-vous genoemd. Deze combinatie maakt een sterke synchronisatie van processen mogelijk.
2. **Niet-blokkerende send** en **blokkerende receive**. De zender kan doorgaan maar de ontvanger wordt geblokkeerd totdat het gevraagde bericht is aangekomen. Waarschijnlijk is dit de nuttigste combinatie. Het biedt een proces de mogelijkheid zo snel mogelijk een of meer berichten te verzenden naar uiteenlopende bestemmingen. Een proces dat een bericht moet ontvangen voordat het nuttig werk kan uitvoeren, moet worden geblokkeerd totdat een bericht is aangekomen. Een voorbeeld is een serverproces dat een dienst of bron beschikbaar stelt aan andere processen.

De **niet-blokkerende send** is de meest natuurlijke benadering voor veel taken bij het programmeren van gelijktijdigheid (concurrent programming). Wordt het bijvoorbeeld gebruikt om te vragen om een uitvoerbewerking, zoals afdrukken, dan kan een proces het verzoek versturen in de vorm van een bericht en vervolgens doorgaan.

Een potentieel gevaar van de niet-blokkerende **send** is dat een fout kan leiden tot een situatie waarin een proces herhaaldelijk berichten verstuurt. Aangezien het proces niet wordt afgeremd door een blokkering, verbruiken deze berichten systeembronnen, waaronder processortijd en bufferruimte, ten koste van andere processen en van het besturingssysteem.

Bij een niet blokkerende **send** is de programmeur bovendien verantwoordelijk voor het controleren van de ontvangst van een bericht: processen moeten antwoordberichten gebruiken om de ontvangst van een bericht te bevestigen.

Bij de primitieve **receive** lijkt de blokkerende versie de meest natuurlijke voor veel taken bij het programmeren van gelijktijdigheid. Doorgaans heeft het proces dat verzoekt om een bericht de verwachte informatie nodig voordat het verder kan gaan.

Gaat een bericht verloren, wat kan gebeuren in een gedistribueerd systeem, of wordt een proces afgebroken voordat het een verwacht bericht verzendt, dan kan het ontvangende proces voor onbepaalde tijd worden geblokkeerd. Dit probleem kan worden opgelost door het toepassen van de niet-blokkerende **receive**. Het gevaar van deze benadering is echter dat een bericht verloren gaat als het wordt verstuurd nadat een proces de bijhorende receive heeft uitgevoerd.

Andere mogelijke benaderingen zijn een proces te laten controleren of een bericht wacht voordat de **receive** wordt uitgevoerd en een proces de mogelijkheid bieden meer dan 1 bron op te geven in een primitieve **receive**. De laatste benadering is nuttig als een proces wacht op berichten van meer dan 1 bron en kan doorgaan na de ontvangst van een van deze berichten.

3.8.2 Adressering

Vanzelfsprekend is een manier nodig om in de primitieve op te geven welk proces het bericht moet ontvangen. Op dezelfde wijze bieden de meeste implementaties een ontvangend proces ook de mogelijkheid de bron op te geven van een te ontvangen bericht.

De verschillende manieren voor het opgeven van processen in de primitieven **send** en **receive** kunnen worden ondergebracht in 2 categorieën: **directe adressering** en **indirecte adressering**.

Bij **directe adressering** bevat de primitieve **send** de identificatiecode van het bestemmingsproces. De primitieve **receive** kan op 2 manieren worden behandeld. Eén mogelijkheid is te eisen dat het proces expliciet een verzendend proces aanwijst. Het proces moet dan van tevoren weten van welk proces een bericht wordt verwacht. Dit is vaak effectief bij samenwerkende, gelijktijdige processen. In andere gevallen is het echter onmogelijk het verwachte bronproces op te geven. Een voorbeeld is een printer-server-proces dat een verzoek om afdrukken accepteert van alle andere processen. Bij dergelijke toepassingen is het gebruiken van impliciet adressering meer effectief. In dit geval bevat de parameter **bron** van de primitieve **receive** een waarde die wordt teruggegeven wanneer de bewerking voor de ontvangst is uitgevoerd.

De andere algemene benadering is **indirecte adressering**. Hierbij worden berichten niet rechtstreeks verzonden van de zender naar de ontvanger maar naar een gedeelde gegevensstructuur die bestaat uit wachtrijen waarin berichten tijdelijk kunnen worden opgeslagen. Dergelijke wachtrijen worden doorgaans **postvakken (mailboxes)** genoemd. Twee processen kunnen communiceren als het ene proces een bericht verzendt naar het juiste postvak en het andere proces het bericht ophaalt uit dat postvak.

De kracht van het gebruik van indirecte adressering is dat dit, door het loskoppelen van zender en ontvanger, een grotere flexibiliteit biedt bij het gebruik van berichten. De relatie tussen zenders en ontvangers kan 1 op 1, 1 op veel, veel op 1 of veel op veel zijn. Een 1-op-1relatie maakt het instellen van een koppeling voor privé-communicatie tussen processen mogelijk. Dit beschermt hun interactie tegen foutieve verstoringen door andere processen. Een veel-op-1relatie is nuttig voor de interactie van clients en een server: 1 proces verzorgt diensten voor meerdere ander processen. In deze situatie wordt het postvak vaak **een poort (port)** genoemd.

Bij een 1-op-veelrelatie bestaan er 1 zender en meerdere ontvangers; dit is nuttig voor toepassingen waarbij een bericht of bepaalde informatie moet worden verzonden naar een aantal processen (broadcast).

De relatie tussen processen en postvakken kan **statisch** of **dynamisch** zijn. Poorten zijn vaak statisch verbonden aan een bepaald proces: een poort wordt gecreëerd en wordt permanent toegewezen aan het proces.

Op vergelijkbare wijze wordt een 1-op-1relatie vaak gedefinieerd als statisch en permanent. Zijn er veel zenders, dan kan de relatie tussen een zender en een postvak dynamisch zijn. Hiervoor kunnen primitieve zoals **verbinden (connect)** en **verbinding verbreken (disconnect)** worden gebruikt.

Een verwant aandachtspunt heeft te maken met het eigendom van een postvak. Een poort wordt veelal gecreëerd door en is het eigendom van het ontvangende proces. Wordt een proces verwijderd, dan wordt ook de poort verwijderd. Voor algemene postvakken kan het besturingssysteem een dienst voor het creëren van een postvak aanbieden. Zulke postvakken kunnen worden beschouwd als eigendom van het creërende proces, wat impliceert dat ze tegelijk met het proces worden verwijderd, of als het eigendom van het besturingssysteem, wat betekent dat een expliciete opdracht nodig is voor het verwijderen van het postvak.

3.8.3 Indeling van berichten

De indeling (**format**) van berichten is afhankelijk van het gebruiksdoel van de berichtenfaciliteit en van de vraag of deze op 1 computer of op een gedistribueerd systeem functioneert.

Bij sommige besturingssystemen hebben de ontwerpers de voorkeur gegeven aan korte berichten met een vaste lengte om de overhead van de verwerking en opslag te minimaliseren.

Moet een grote hoeveelheid gegevens worden doorgegeven, dan kunnen de gegevens in een bestand worden geplaatst en volstaat een verwijzing naar dat bestand in het bericht.

Een meer flexibele benadering is het toestaan van berichten met een variabele lengte.

Een bericht bestaat uit 2 delen:

- een **kop (header)**, die informatie over het bericht bevat.
- een **romp (body)**, die de feitelijke inhoud van het bericht bevat.

3.8.4 Beslissingsregel wachtrij

De eenvoudigste beslissinsregel voor de wachtrij is **FIFO**. Deze regel zal echter niet toereikend zijn indien sommige berichten urgenter zijn dan andere berichten.

Een alternatief is het bieden van de mogelijkheid om de prioriteit van een bericht te specificeren; bijvoorbeeld aan de hand van het soort bericht of een vermelding door de afzender.

Nog een andere manier is de ontvanger de mogelijkheid te bieden de wachtrij met berichten te inspecteren en het volgende te ontvangen bericht te kiezen.

3.9 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. 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.

4 Processen

4.1 Wat is een proces

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

Multi:

- **multi-user** (veel gebruikers): verschillende gebruikers kunnen tezelfdertijd processen opstarten.
- **multi-tasking** (veel taken uitvoeren): een gebruiker kan meerdere processen tezelfdertijd opstarten.
- Processen moeten beheerd worden: geheugen - schijfruimte - processor capaciteit - ….

4.2 Soorten processen

Er bestaan drie soorten processen:

- interactieve
- automatische
- daemons

4.2.1 Interactieve processen

opstarten en controleren vanuit een terminal sessie, m.a.w. er moet iemand aangemeld zijn op het systeem.

- Interactieve processen kunnen zowel in de voorgrond (foreground) als in de achtergrond (background) draaien.
- Foreground processen houden de terminal bezet zolang ze lopen.
- Background processen bezetten de terminal niet en kunnen andere taken uitgevoerd worden.

4.2.2 Automatische processen

Deze processen wachten 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”, wat staat voor “first in, first out”.

Twee manieren:

1. at
2. batch

4.2.3 Daemons

Daemons 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.

4.3 Background

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.

Een proces opstarten in background: **commandonaam & PID**: process identification - proces volgnummer.

Jobnumber: dit is een nummer dat door de shell gebruikt wordt.

4.4 Eigenschappen

Elk proces heeft een aantal vaste eigenschappen:

- 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- letterlijk: het PID van de ouder.
- Het zogenaamde nice number: de mate van vriendelijkheid van dit proces: laat het nog veel processorcapaciteit 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.

4.5 Informatie weergeven

4.5.1 Korte info

ps zonder opties - process:

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.

4.5.2 Uitgebreide info

ps -ef:

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.

top commando:

Geeft ongeveer dezelfde informatie als ps -ef, maar het wordt om de 5 seconden opgefrist.

Bovendien hebben we hier al automatisch een zeker 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.

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 plaatsje 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.

pstree commando:

Samenhang van de processen

De meeste processen stammen af van **init of systemd (redhat)**, het initiële proces waarmee het systeem gestart wordt.

4.6 Processen beheren

(deel van) commando	Betekenis
Commandonaam	Draait het commando in de voorgrond
Commandonaam &	Draait het commando in de achtergrond en geeft de terminal vrij
Jobs	Toont de commando's die in de achtergrond draaien
Crtl+Z	Bevriest het commando
Crtl+C	Beëindigd het commando dat in de voorgrond draait
%n	Elk commando in de achtergrond krijgt een jobnummer
Bg	Activeer een bevroren commando terug
Fg	Breng een commando van de achtergrond naar de voorgrond
Kill	Beëindig een programma dat in de achtergrond draait

Enkele handige commando’s:

- 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 vóór het commando dat je wilt uitvoeren.
- Het commando **uptime** geeft informatie over de belasting van het systeem.
- Het commando **w** geeft een overzicht van de aangemelde gebruikers en hun activiteit(en).

Grafisch gereedschap

Er is een heel gamma aan grafische programma's ter beschikking, die het makkelijker maken om de output van de vaak cryptische tekstcommando's te analyseren. Een voorbeeld is **gnome-system-monitor**.

Iets eenvoudiger is het programma **xload**. Je moet het even laten draaien voor je beeld krijgt. Dit programma geeft een periodiek bijgewerkt histogram van de belasting op het systeem.

4.7 Een programma onderbreken

Indien één van de processen die je zelf hebt opgestart, te veel middelen gebruikt, heb je twee mogelijkheden:

1. Zorg dat het proces minder belastend is door middel van het **renice** commando; hiervoor moet je het proces niet onderbreken.
2. Stop het proces.

4.7.1 De prioriteit veranderen

Het werken met **nice** en **renice** vereist een uitgebreide kennis van het systeem. Er is echter een gemakkelijker manier: **top**.

Een belastend proces zal vermoedelijk een negatieve nice waarde hebben in de kolom “NI”.

renice -n prioriteit -p PID

4.7.2 Het kill commando

Een proces stoppen omdat het hangt, op hol slaat of teveel of te grote bestanden aanmaakt, doe je met **kill**.

Als je daartoe de gelegenheid hebt, probeer dan eerst de zachtaardige manier en stuur een **SIGTERM** (waarde 15) signaal.

Dit instrueert het proces om af te handelen waar het mee bezig is volgens de procedures zoals beschreven in de code van het programma. Op die manier wordt alle rommel opgeruimd en worden er geen bestanden beschadigd.

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 ook dat 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.

4.7.3 Het xkill commando

Grafische programma's die vasthangen kan je proberen 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.

4.8 Processen programmeren voor automatische uitvoering

Er zijn **drie manieren uitgestelde taken in te plannen**:

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**.**

4.8.1 Het sleep commando

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

4.8.2 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.

Je kan een overzicht krijgen van alle at 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.

4.8.3 Het cron systeem

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.

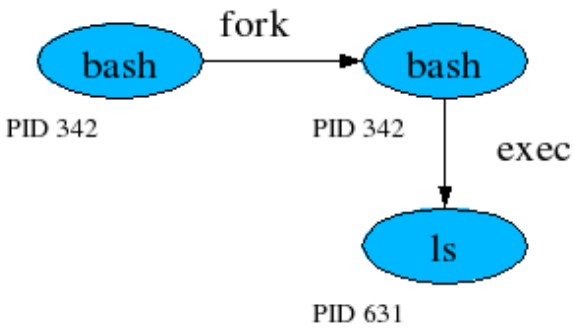
Andere taken kunnen zijn: het maken van backups, rapporten opmaken en doorsturen, systeeminformatie analyzeren en doormailen naar de administrator, herinneringsbrieven mailen, enzovoorts.

Alle taken die periodiek uitgevoerd moeten worden, komen voor opname in het cron systeem in aanmerking.

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

4.9 De levenscyclus

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 een **exec**.

Het geheel wordt **fork-and-exec** genoemd**.**

4.9.1 De rol van init

Zoals je kunt zien aan de output van het **pstree** commando, hebben veel processen init 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 init proces neemt de rol van peetvader van zulke processen: als de ouder sterft, vallen ze onder de verantwoordelijkheid van init.

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 zombie-proces, het blijft in het geheugen hangen tot je de computer herstart.

4.9.2 Een proces beëindigen

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**.