KU LEUVEN

Besturingssystemen

Samenvatting gebaseerd op de lessen van Naessens Vincent,
alsook op het boek,
Operating Systems: Internals and Design Principles.

Gilles Callebaut

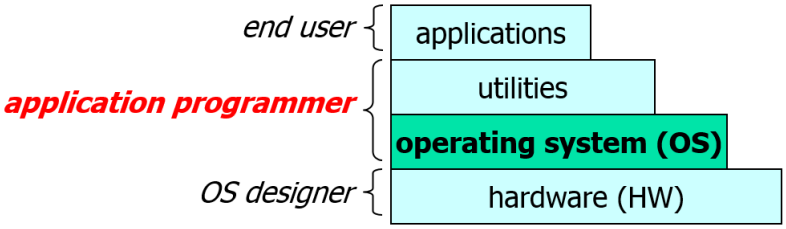
1 INHOUD

H2 - OPERATING SYSTEM OVERVIEW

1 DOELSTELLINGEN EN FUNCTIES VAN EEN BESTURINGSSYSTEEM

Het OS is een interface tussen de gebruiker en de hardware.
Waarbij het OS 3 doelstellingen heeft:

- Gemak
- Efficiëntie
- Evolutie is mogelijk



1.1 HET OS ALS USER/COMPUTER INTERFACE

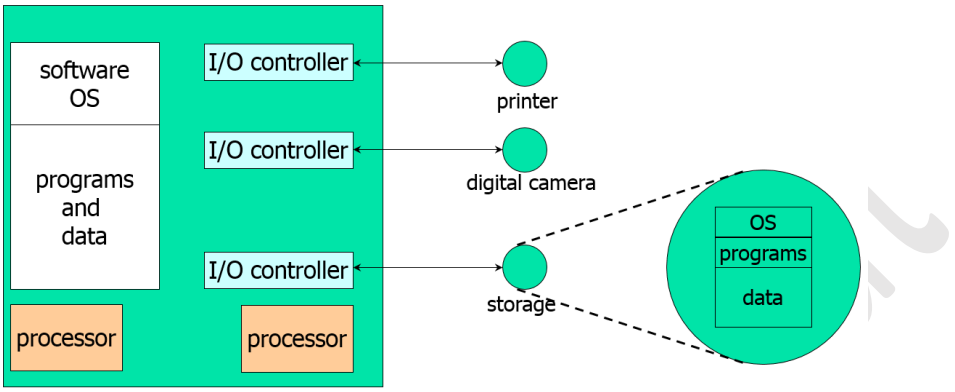
Services die het OS biedt:

- Program development
  Application program develoment tools –meestal in de vorm van utilities)
- Program execution
- Acces to I/O devices
- Controlled access to files
  Het OS kent de structuur van de data die opgeslagen zit op het medium –bestandsindeling-.
  Ook kan het OS bescherming bieden bij multiple user-systems.
- System access
  Bescherming van data en resources voor onbevoegden, en conflicten zien op te lossen.
- Error detection and response
- Accounting –Administratie-
  Statistieken bijhouden voor het optimaliseren van het OS
- Instruction set architecture (ISA)
  Definieert alle mogelijke instructies die de hardware kent.
- Application binary interface (ABI)
  Een standard voor binaire system calls.
- Application programming interface (API)
  Hetzelfde als een ABI maar met high-level language (HLL) –dus niet gecompileerd- calls.

1.2 HET OS ALS RESOURCE MANAGER

Het besturingssysteem is verantwoordelijk voor het beheer van resources –zoals storage, processing van data,..- die een computer kan uitvoeren.

Het OS staat de controle van de processor af om nuttig werk te leveren, om dan nadien weer de controle te grijpen om de processor klaar te maken voor het volgende werkje.
Daardoor zit een deel van het OS –de kernel- geladen in main-memory.



computer system

1.3 FLEXIBELE ONTWIKKELINGSMOGELIJKHEDEN

- Kunnen inspelen op nieuwe hardware
- Kunnen inspelen op nieuwe services
- Verbeteringen, updates, patches[1] toevoegen

Net hierom moet een OS modulair opgebouwd zijn met duidelijke interfaces tussen de modulus –en moet goed gedocumenteerd-.

[1] Een softwarematige patch is een klein stukje software dat door de uitgever van software gebruikt wordt om fouten op te lossen of updates uit te voeren aan zijn software.

2 EVOLUTIE VAN BESTURINGSSYSTEMEN

2.1 SERIËLE VERWERKING

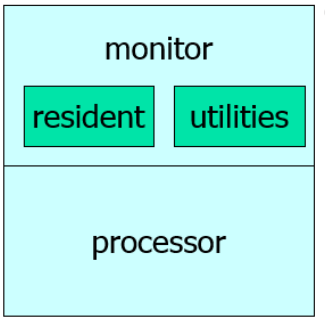
Machine instructies werden ingeladen via een kaartlezer, de uitvoer kwam terecht op een printer.
Er werd toen nog geen gebruik gemaakt van een OS, waardoor er 2 grote problemen voorkwamen:

- Scheduling
  Uitvoeringstijd van een job kon korter zijn dan de ingestelde tijd, of het omgekeerde –forced stop-.
- Setup time
  De insteltijd was zeer groot door het laden van compiler en laden van het HLL programma, om dan nadien deze te linken met veelgebruikte functies. Het laden/verwijderen van deze tapes/kaarten vergde veel tijd.

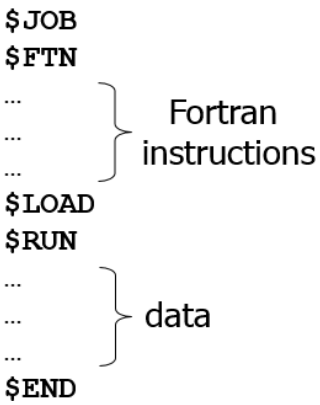
Een oplossing voor deze problemen omvatten; het includeren van libraries, linkers, loaders, debuggers en I/O driver routines als common software voor alle gebruikers.

2.2 EENVOUDIGE BATCHSYSTEMEN

Voor het verbeteren van het gebruik van de kostbare processor tijd, werd het concept batch[2] OS geïntroduceerd.
Bij dit soort OS werd er gebruikt gemaakt van een monitor, zodat de gebruiker niet rechtsreeks aan de processor kon. De monitor –software- lade automatisch programma's in een batch. De monitor had de controle over de sequentie van jobs, daarom werd er een groot deel van de monitor –residente monitor-geplaatst in het hoofd geheugen.



Via job control languages –JCL- worden er instructies door de monitor uitgevoerd. Zo kan de monitor de insteltijd van programma's versnellen. De monitor leest de `$FTN` lijn en laad de correcte compiler voor deze taal. Als het programma niet in het geheugen zit –maar op een tape- dan wordt er gebruik gemaakt van de lijn `$LOAD` die ook alweer wordt uitgevoerd door de monitor.



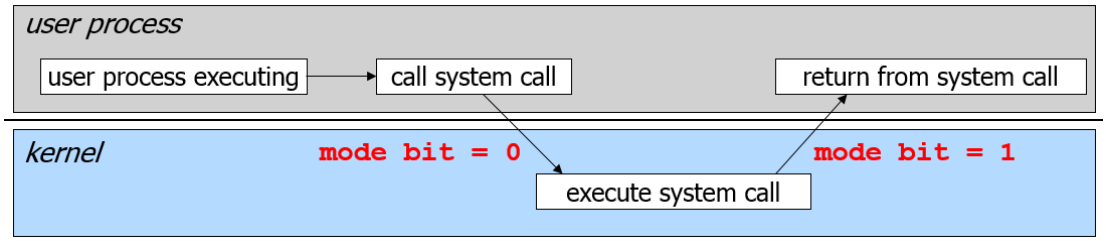
[2] Batch staat voor een rij van jobs.

Hardware matig worden er nog enkele zaken toegevoegd:

- Geheugenbeveiliging
  Niet overschrijven van de monitor
- Timer
  Beperken van de tijd die een taak krijgt om uit te voeren –monopolie vermijden-.
- Privileged instructions
  Voorkomen van rechtstreekse toegang tot I/O devices
- Interrupts
  OS meer flexibiliteit geven in het geven en nemen van de controle

Een user program zal uitgevoerd worden in **user mode**, waardoor bepaalde delen van het geheugen en instructies niet toegankelijk zijn.
Wanneer de monitor uitgevoerd wordt zal dit gebeuren in **kernel mode**, zo heeft deze toegang tot geprivilegieerde instructies en meer geheugen kan worden benaderd.

Hierdoor is er overhead namelijk, het hoofdgeheugen en processor tijd wordt nu gedeeltelijk ge-shared met de monitor.



2.3 MULTI-PROGRAMMING BATCHSYSTEMEN

De processor is nog te vaak ongebruikt, vermits I/O apparaten erg traag zijn.
De oplossing hierin ligt over te gaan van uni-programming naar multi-programming –multi-tasking-.
Wanneer er een programma wacht op een I/O apparaat dan wordt er overgeschakeld naar een ander programma. Deze nieuwe manier van uitvoering heeft nieuwe Software en Hardware nodig, namelijk

- Bij het starten van een I/O opdracht → uitvoeren andere taak –I/O interrupt-
- Indien I/O opdracht is beëindigd → interrupt naar de processor

Waardoor de complexiteit van het geheugenbeheer stijgt –verschillende jobs in geheugen-, alsook de nood aan een scheduling algoritme –deze beslist welke job de processor krijgt-.

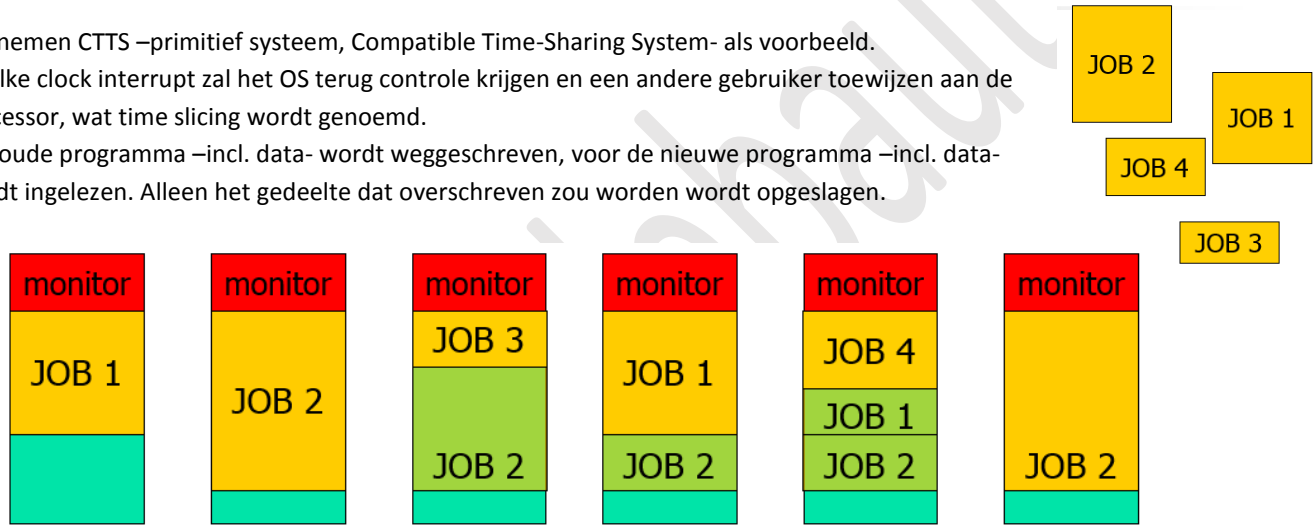
2.4 TIME-SHARING SYSTEMEN

Time sharing is een techniek om interactieve taken te kunnen afhandelen –door multi-programming-, de processortijd wordt dus gedeeld met verschillende gebruikers.

Beiden –batchsystems en time sharing- gebruiken multiprogramming, maar er zijn duidelijke verschillen:

	batchmultiprogrammering	time-sharing
hoofddoel	maximaliseren processorgebruik	minimaliseren antwoordtijd
OS instructies	instructies in JCL	opdrachten aan terminal

We nemen CTTS –primitief systeem, Compatible Time-Sharing System- als voorbeeld.
Bij elke clock interrupt zal het OS terug controle krijgen en een andere gebruiker toewijzen aan de processor, wat time slicing wordt genoemd.
Het oude programma –incl. data- wordt weggeschreven, voor de nieuwe programma –incl. data- wordt ingelezen. Alleen het gedeelte dat overschreven zou worden wordt opgeslagen.



Nieuwe problemen duiken op bij systeem waarbij time-sharing wordt gebruikt –alsook bij muti-programming- namelijk:

- Verschillende taken in het geheugen mogen niet met elkaar storen –bv. elkaars data wijzigen-
- Alleen geautoriseerde gebruikers mogen bepaalde bestanden bewerken/bekijken
- Gelijktijdig gebruik maken van resources moet behandeld worden

3 BELANGRIJKE PRESTATIES

3.1 PROCESSEN

Het grootste probleem bij de groei van systeemsoftware is de coördinatie van de activiteiten:

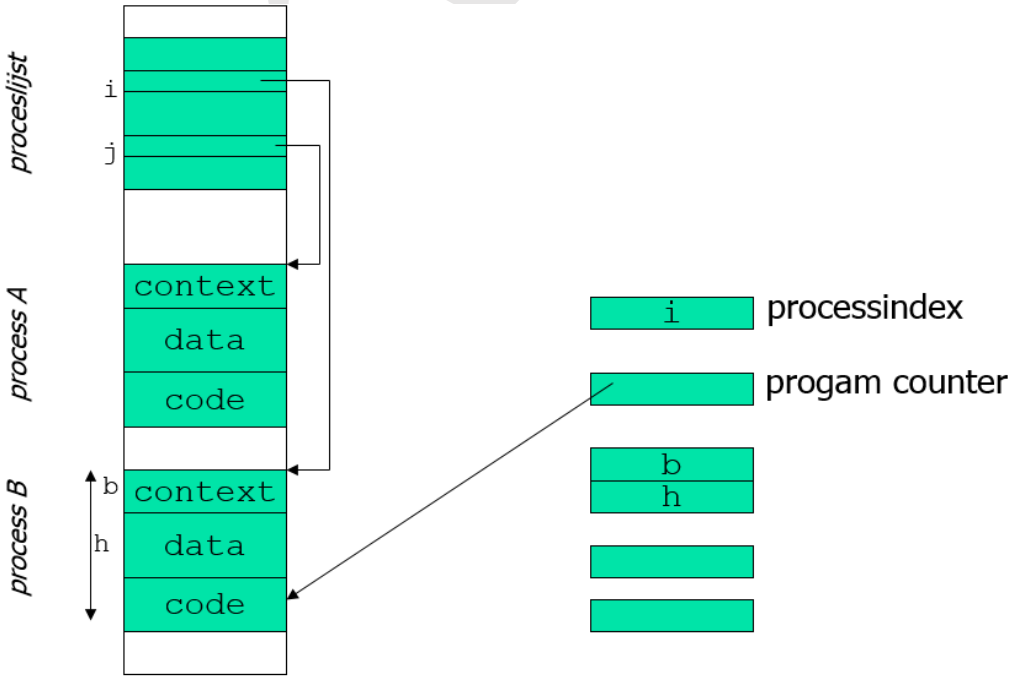
- Onjuiste synchronisatie
  Signaleringsmechanisme na I/O bewerking kan een duplicaat zijn of verloren gaan.
- Mislukte wederzijdse uitsluiting
  Gecontroleerde toegang tot resources
- Niet-vast omschreven programmaverwerking
  Overschrijven van geheugen (andere programma's *interferen*)
- Deadlock
  Eindeloos wachten, veroorzaakt door het op elkaar te wachten om de resources vrij te geven.

De oplossing voor deze problemen is een systematische aanpak voor het bewaken en beheren van uitvoering programma's. Zo kunnen we een proces aanschouwen bestaande uit 3 componenten:

- Een uitvoerbaar programma
- Geassocieerd met data
- En met de executie context van het programma

De executie context –of process state- is de interne data waarbij het OS het proces kan beheren en controleren. In deze context staat onder andere verschillende processor registers –zoals programma counter en data registers-, prioriteit en de staat door een I/O event.

In de processor registers staat de proces index –welk proces is bezig-, program counter –wijst naar de volgende instructie- , base –begint adres- en limit –grootte-. Het program counter en alle data verwijzingen worden geïnterpreteerd relatief aan het base register, deze mogen niet groter zijn dan het limit register. Zo voorkomen we interprocess interference.



3.2 GEHEUGENBEHEER

Het OS heeft 4 doelen i.v.m. geheugenbeheer:

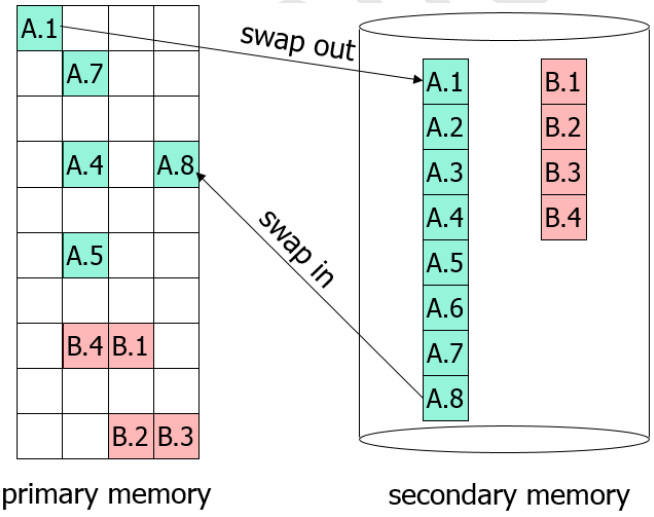
- Procesisolatie
  Processen mogen elkaars instructies –en data- niet verstoren
- Automatisch toewijzing en beheer
  Allocatie is transparant t.o.v. de programmeur, i.e. de programmeur moet zich niet buigen over geheugen limitaties en efficiënt geheugen toewijzen.
- Bescherming en toegangsbeheer
  De OS zorgt ervoor dat er maar porties van het geheugen toegankelijk zijn voor verschillende gebruikers
- Lange termijn opslag

De oplossingen voor deze doelen zijn:

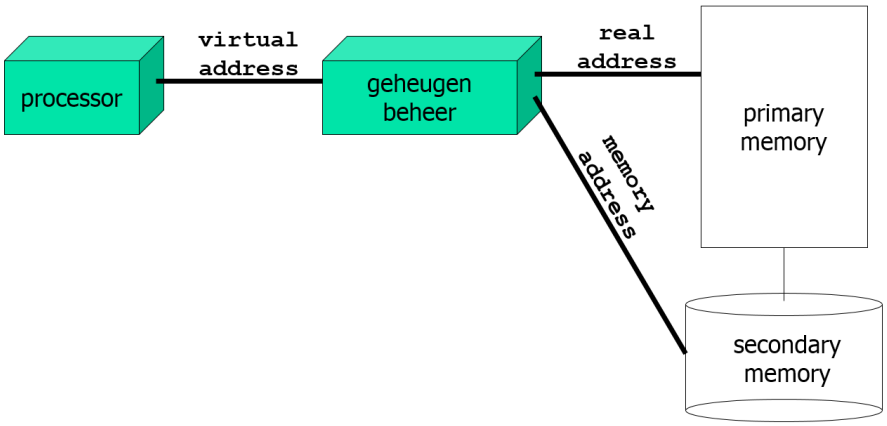
- Bestandensystemen
  Lange termijn opslag en toegangsbeheer
- Virtueel geheugen
  Automatische toewijzing en procesisolatie

Bij virtueel geheugen kan een programma geheugen adresseren zonder rekening te houden met de hoeveelheid –fysiek- geheugen die toegankelijk is.
Door paging systems worden processen opgedeeld in blokken –pages- van gelijke lengtes.

Een programma refereert naar een woord via een virtueel adres –deze bestaat uit de pagina nummer en de offset binnen deze pagina-.



Wanneer een programma wordt uitgevoerd, zal er een deel van zijn *pages* in het hoofd geheugen zitten. Als er een referentie wordt gemaakt naar een pagina dat niet in het hoofdgeheugen zit, dan zal *the memory managment hardware* dit detecteren en deze pagina laden in het hoofd geheugen –vanuit de *disk-*, dit wordt virtueel geheugen genoemd.



De processors hardware –samen met het OS- bieden de gebruiker met een virtuele processor die toegang heeft tot het virtuele geheugen.

Procesisolatie kan verzekerd worden door elk proces een uniek –niet overlappend- virtueel geheugen te geven, het omgekeerde kan gerealiseerd worden bij *memory sharing*.

3.3 INFORMATIE BEHEER EN BEVEILIGING

De 4 hoofdzakelijke doelen bij bescherming en beveiliging van informatie is:

- Availability
  Beheer tegen interruptie
- Confidentiality
  Gebruikers kunnen geen onbevoegde data benaderen
- Data integrity
  Data kan niet onbevoegd worden veranderd
- Authenticity
  Er moet een goede verificatie zijn van de identiteit van gebruikers, berichten en data.

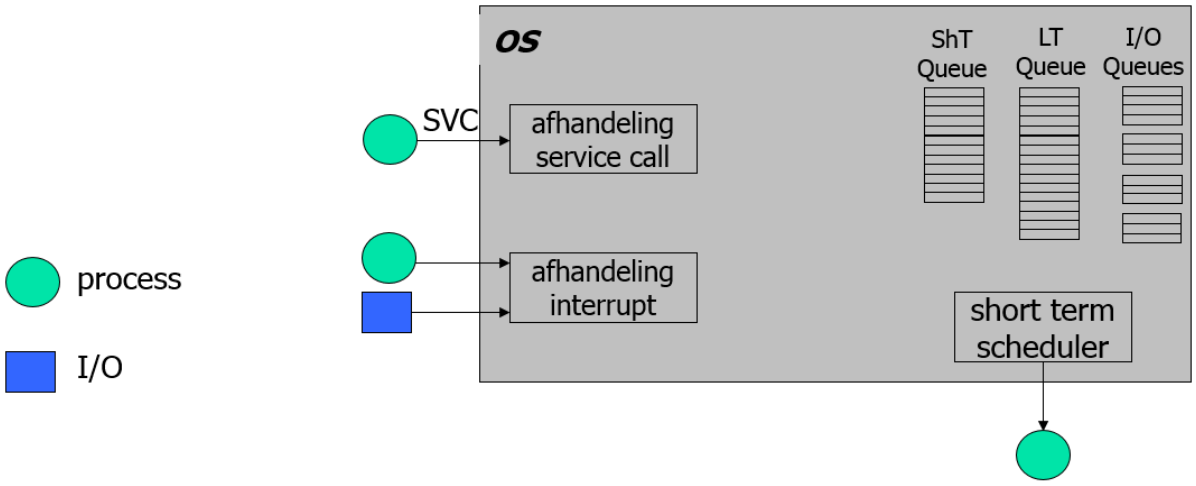
3.4 SCHEDULING EN BEHEER VAN BRONNEN

Criteria:

- Rechtvaardigheid
- Dynamisch en reactief
  Het dynamisch *alloceren* en *schedulen* zodat alle *sets of requirements* voldaan zijn.
- Efficiëntie
  Maximum *throughput*, minima aan *response time*, en – bij time sharing- zoveel mogelijk gebruikers ondersteunen.

Het OS beheert een aantal *queues* –elk I/O *device* heeft zijn eigen *queue*-.
In de *long-term queue* staat een lijst van processen die wachten om de processor te gebruiken. Processen worden aan het systeem toegevoegd als een proces –door het OS- naar een *short tem queue* wordt getransporteerd. Dan wordt een portie van het hoofd geheugen *geacloceerd* voor het inkomend proces.

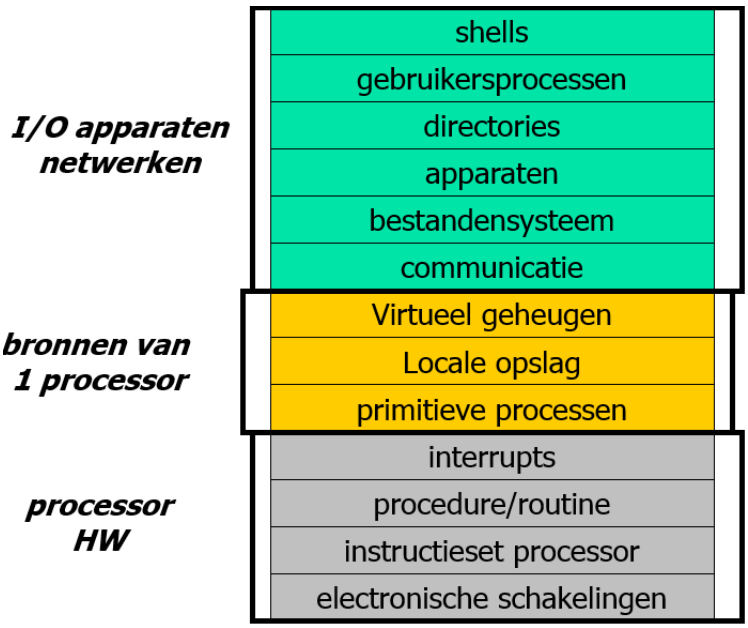
Wanneer een *interrupt* of *service call* wordt behandelt zal de *short-term scheduler* opgeroepen worden om een proces te kiezen voor executie.



3.5 SYSTEEMSTRUCTUUR

We stellen vast dat het aantal taken van het OS stijgt samen met de complexiteit van de hardware. Met als gevolg dat het OS e laat wordt uitgebracht, het aantal bugs stijgt, de prestaties lager zijn dan eerst geanticipeerd en dat het moeilijk omgaan is met externe aanvallen.

De complexiteit wordt beheerst door de modulaire opbouw met eenvoudige interfaces en een hiërarchisch model.



4 ONTWIKKELINGEN IN DE AANLOOP NAAR MODERNE BESTURINGSSYSTEMEN

4.1.1 MICROKERNEL ARCHITECTUUR

Recent wordt er gebruik gemaakt van een microkernel, daarvoor was het OS uitgerust met een *monolithic kernel*[3]. Waarbij het OS dus een monolitisch blok was.

Een microkernel architectuur daarentegen bestond enkel uit essentiële functies, de rest van de functies werden gezien als aparte processen, servers genaamd. Deze processen/diensten werden uitgevoerd in user modus.

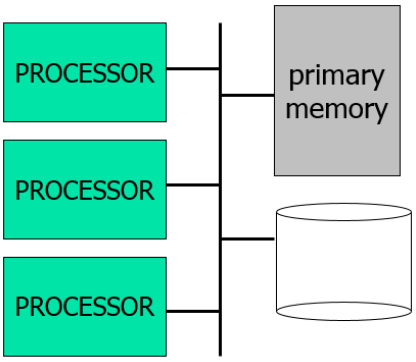
4.1.2 MULTITHREADING

Een proces kan bestaan uit meerdere threads[4] die parallel kunnen lopen.

Multithreading is nuttig voor applicaties die bepaalde losstaande taken uitvoeren.
Het omschakelen tussen processen gaat gepaard met een zware context switch in tegenstelling tot het omschakelen tussen threads.

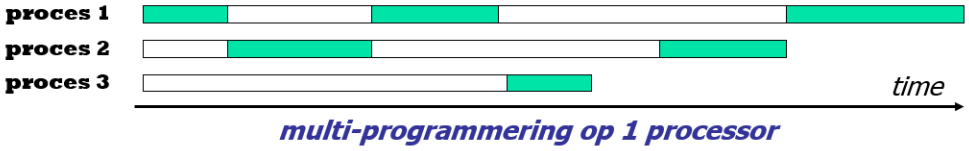
4.1.3 SYMMETRISCHE MULTIPROCESSING (SMP)

Bij SMP wordt er gebruikt gemaakt van meerdere –gelijke- processoren die memory en I/O apparaten delen.

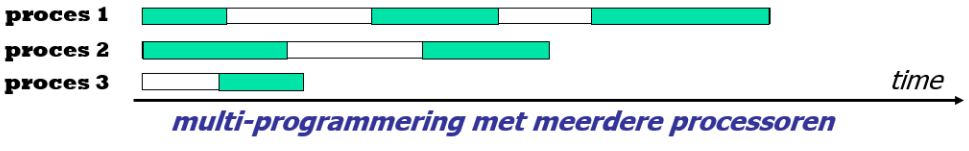


Bij SMP worden volgende voordelen gerealiseerd:

- *Performance*
  Meerdere processen kunnen gelijktijdig –dus parallel- worden uitgevoerd (multi-processing)
- *Availability*
  De andere processors kunnen de taken van een defecte processor overnemen.
- Schaalbaarheid



multi-programmering op 1 processor

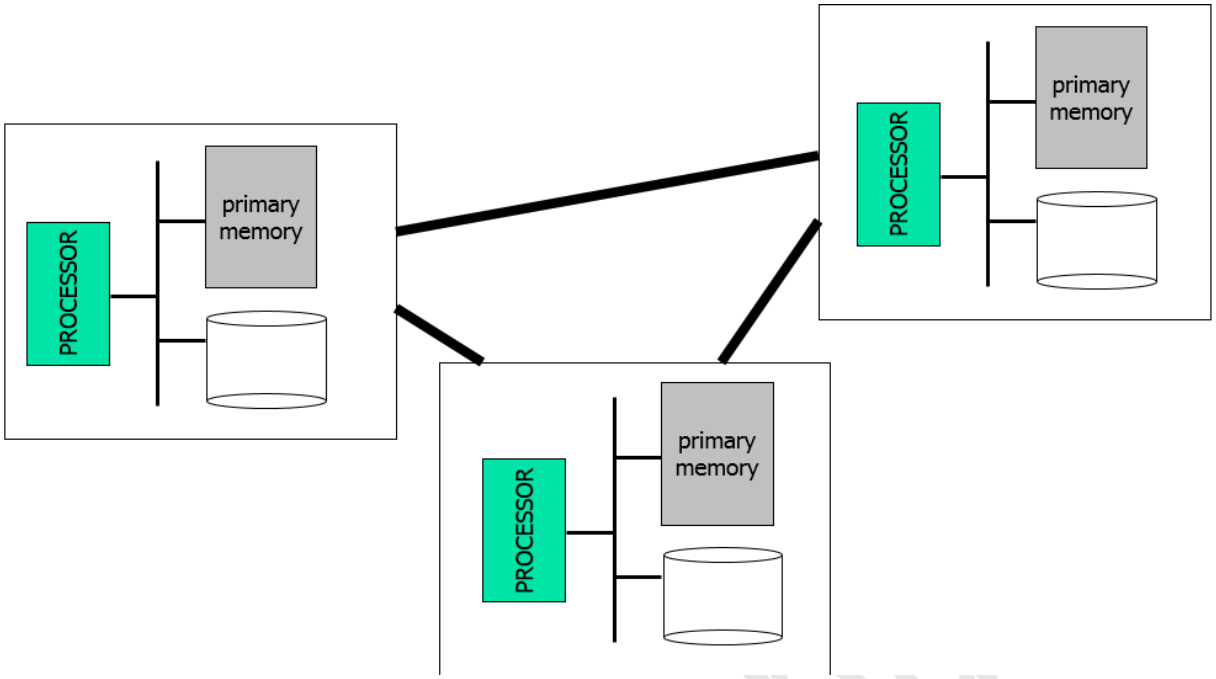


multi-programmering met meerdere processoren

4.1.4 GEDISTRIBUEERDE BESTURINGSYSTEMEN

[3] Een *monolithic kernel* is een geïmplementeerd als een proces waarbij alle functies van het OS geladen zijn in dat proces.
[4] Een thread is een uitschakelbare *unit of work*. Deze bevat een processor context en zijn eigen data veld. Een thread wordt sequentieel uitgevoerd en is ook *interruptible* zodat een andere thread kan uitgevoerd worden. Een proces kan bestaan uit één of meerder threads.

Bij gedistribueerde besturingssystemen werken we met clusters van computers.



Hierbij wordt er de illusie gecreëerd dat men werkt met één *main memory* en één *secondary memory space*, plus nog andere verenigde voorzieningen.

H3 - PROCESS DESCRIPTION AND CONTROL

1 WAT IS EEN PROCES

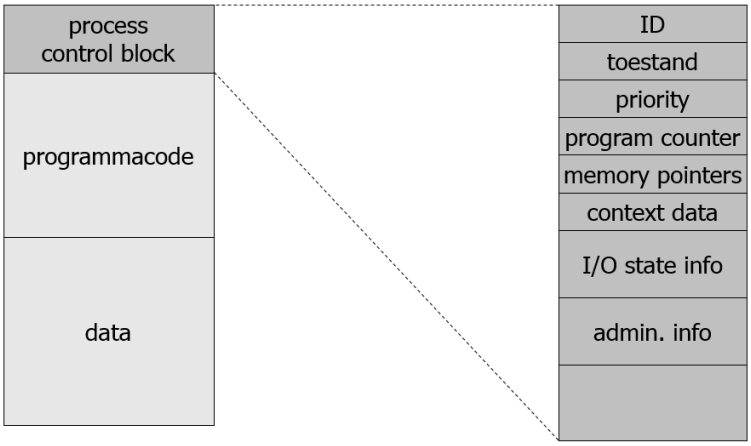
1.1 PROCESSEN EN PROCESS CONTROLE BLOCKS

Een proces bestaat uit –programma- code, data en een proces controle blok.

Het proces controle blok wordt gecreëerd en beheerd door het OS.
Het controle blok maakt het mogelijk om een proces te onderbreken om nadien weer verder uit te voeren alsof de *interrupt* nooit had plaatsgevonden.

Het controle blok bestaat uit:

- Identifier
  Uniek id. per proces, die gegeven wordt wanneer het proces wordt opgestart
- Toestand
- Prioriteit
- Program counter
  pointer naar de volgende instructie in het programma
- Memory pointers
  Code staat verspreidt over het RAM, de pointers houden de plaatsen bij.
- Context data
  Data in registers tijdens uitvoering
- I/O state info
- Accounting information
  Hoeveelheid processortijd, tijd limieten,…

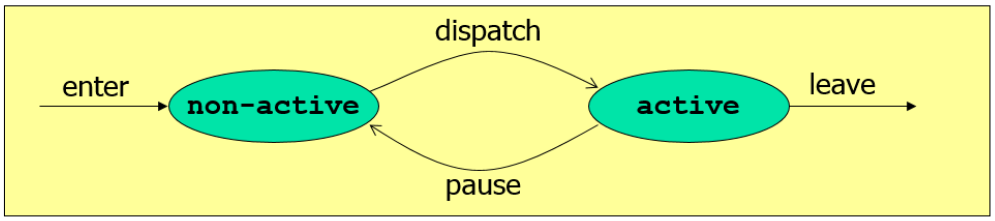


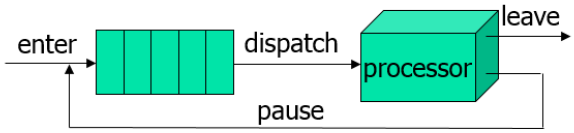
2 PROCESTOESTANDEN

Om te wisselen tussen processen maken we gebruik van een dispatcher programma.
Een dispatcher is een eenvoudig OS dat geladen zit in het hoofd geheugen.

Een proces neemt de processor over als er een I/O request of een time-out optreedt.
Wanneer dit gebeurt zal de dispatcher commando's uitvoeren –om te kunnen wisselen van proces-.

2.1 PROCESMODEL MET TWEE TOESTANDEN





In de *waiting queue* gebruikt men FIFO en de dispatch werkt via round-robin[5].

Hierbij worden er enkele problemen veroorzaakt namelijk I/O bewerkingen. Ee proces kan namelijk nog niet klaar zijn met het ontvangen van de gegevens van een I/O device.

2.2 CREEREN EN BEEINDIGEN VAN PROCESSEN

2.2.1 CREATIE VAN EEN PROCES

Wanneer een nieuw proces wordt toegevoegd zal het OS de data structuren –die nodig zijn om het proces te beheren– maken en adres ruimte *alloceren* in het hoofd geheugen.

Het OS is verantwoordelijk voor de creatie van nieuwe processen –bij nieuwe batchtaak, interactieve aanmelding of verwerkt door een bestaand proces-, maar ook kan het OS zelf een proces maken –*process spawing*[6]-.

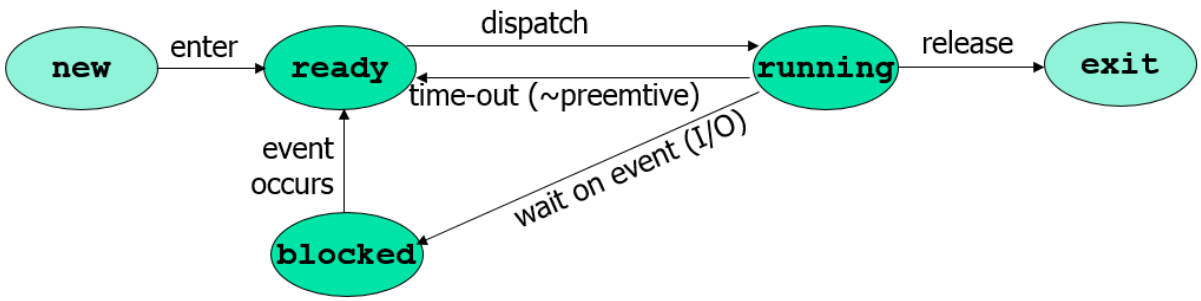
2.2.2 BEEINDING VAN EEN PROCES

Een proces kan op allerlei manieren beëindigd worden:

- Normale voltooiing (~HALT instructie)
- Tijdslimiet is overschreden
- Onvoldoende geheugen
- Time-out
- *Parent process* is beëindigd
- …

[5] Round-robin maakt gebruikt van tijdskwanta, in dit geval komt dit neer op, elk proces krijgt een gegeven hoeveelheid processortijd zo kan er geen *starvation* ontstaan –waarbij één proces alle processortijd inneemt-.
[6] Process spawing is een actie waarbij het OS een nieuw proces creëert op het verzoek van een ander proces, bijvoorbeeld bij het printen.

2.3 PROCESMODEL MET VIJF TOESTANDEN



De vijf staten:

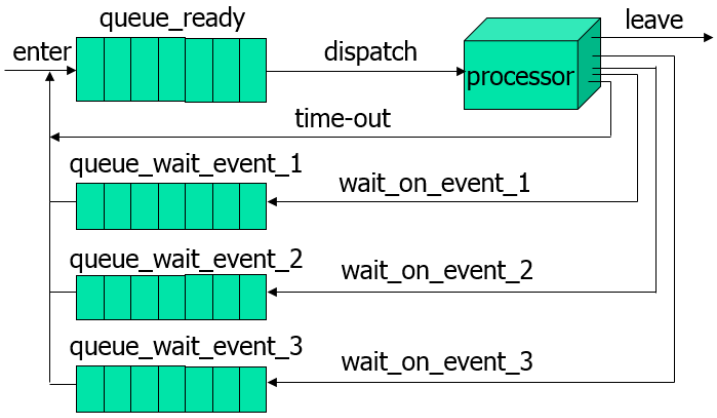
Running	Het proces wordt momenteel uitgevoerd
Ready	Het proces kan uitgevoerd worden wanneer het de mogelijkheid krijgt
Blocked/waiting	Het proces kan niet uitgevoerd worden tot een gebeurtenis plaatsvindt
New	Het proces is net gecreëerd, het controle blok is aangemaakt maar er is nog geen geheugen gealloceerd –het programma bevindt zich nog in het secundaire geheugen-. Het proces is nog niet klaar voor uitvoering.
Exit	Het proces is gestopt (HALT, *aborted*)

Een proces kan van ready naar exit gaan of van blocked naar exit gaan wanneer het *parent* proces het *child* proces stopt of wanneer het *parent* proces beëindigd wordt.

Wanneer een proces in *blocked state* komt staan zal hij terecht komen in een *queue* die toegespitst is op het verwerken van één soort *event*.

Als er ook een prioriteit toegekent wordt aan processen kan het ook handig zijn om per *priority level* een *ready queue* te voorzien.

Ook I/O apparaten hebben elk hun eigen *queue*.



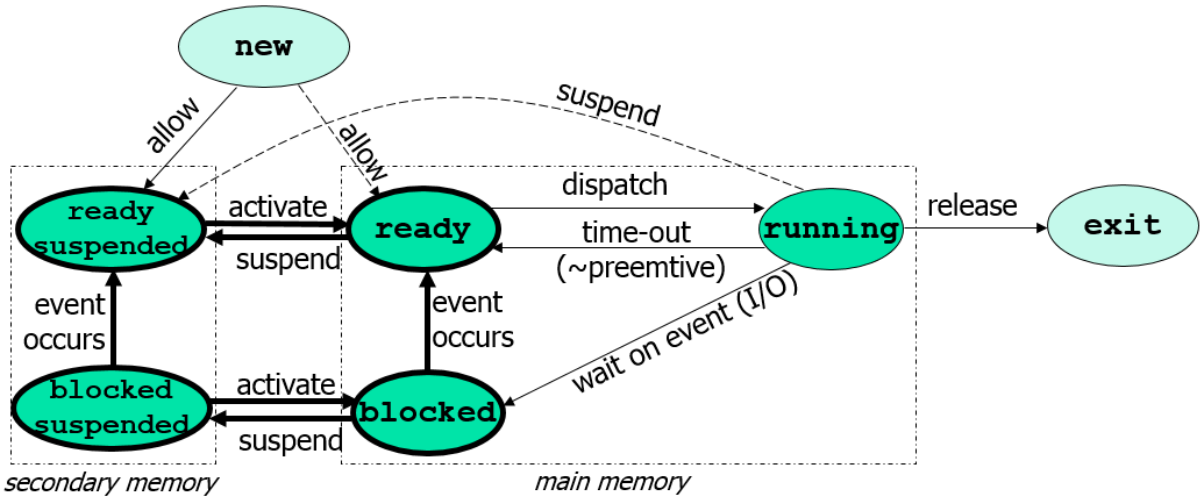
2.4 SUSPENDED PROCESSEN

Als een proces een I/O request uitvoerde kwam deze in de *blocked state* zodat de processor maximaal zou gebruikt worden. Maar net omdat de processor zo snel werkt zullen al snel alle processen terecht komen in de *queues* voor I/O.

Een oplossing voor dit probleem is *swapping*, het verplaatsen van processen naar het secundaire geheugen –*disk*-.
Opmerking: Swapping is ook een I/O operatie maar deze is –meestal- de snelste I/O op het systeem, zodat *swapping* toch de prestaties verbeterd.

Hierdoor komen er nog 2 staten bij, namelijk

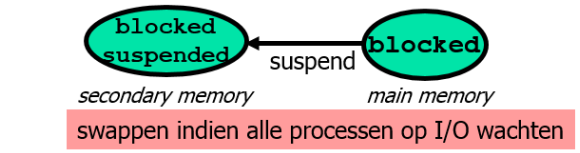
- Blocked/suspended
  Het proces zit in het secundaire geheugen is aan het wachten op een *event*
- Ready/suspended
  Het proces zit in het secundaire geheugen, maar zodra het proces in het hoofd geheugen terecht komt kan het uitgevoerd worden



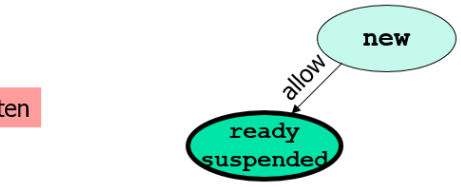
Redenen waarom processen worden opgeschort

- Swapping
  onvoldoende primary geheugen
- Proces veroorzaakt problemen
  Deadlock
- Timing
  Periodiek uit te voeren processen (per tijdsinterval)
- *Parent process* schort *child process* tijdelijk op

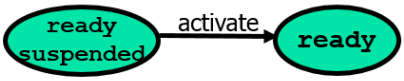
Enkele procesovergangen:



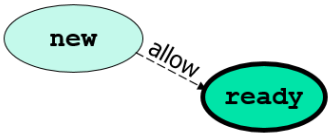
swappen indien alle processen op I/O wachten



reserve van nieuwe processen aanleggen

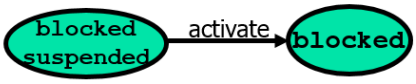


swappen van proces met hoge prioriteit



just-in-time creatie van processen

swappen van proces met lage prioriteit



bij verlaten van ander proces

3 BESCHRIJVING VAN PROCESSEN

3.1 CONTROLE STRUCTUREN VAN HET BESTURINGSSYSTEEM

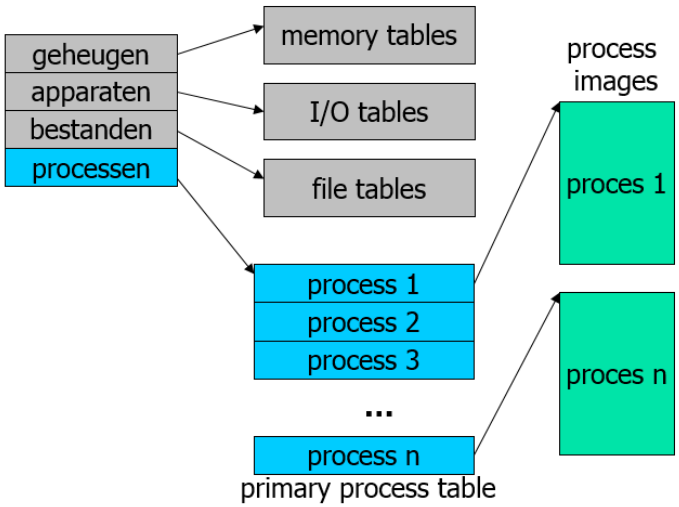
Een OS is een entiteit dat het gebruik van systeem *recources* door processen beheert, ook doet het OS aan *scheduling* en *dispatching* van processen.

Hiervoor houdt het OS *tables of information* bij.

Geheugen *tables* houden informatie bij over het *primary* en *secundary* geheugen.

I/O *tables* houden informatie bij over de beschikbaarheid van I/O apparaten.

File tables houden informatie bij over de toestand van bestanden –*het file management system* heeft ook veel info daarover-.



3.2 CONTROLE STRUCTUREN VAN EEN PROCES

3.2.1 LOCATIE VAN HET PROCES

Een proces wordt voorgesteld als een proces image, deze bevat:

- User Data
  Dit gedeelte wordt gemodificeerd.
- User Program
  Dit gedeelte –het programma- wordt uitgevoerd.
- Stack
  Elke proces heeft minstens één –LIFO- stack.
- Process Control Block
  Data die nodig is zodat het OS het proces kan besturen.

Om een proces uit te voeren met het proces image volledig in het hoofdgeheugen geladen zijn, of minstens in virtueel geheugen geladen zijn. Vermits het proces image in het hoofd- als secundaire geheugen kan zitten moet het OS de locatie weten van elke pagina van het proces image.

3.2.2 ATTRIBUTEN VAN HET PROCES

De informatie over het proces wordt opgeslagen in het controle block. Dit gedeelte kan opgesplitst worden in 3 algemene categorieën.

- Process identification
  Van het proces, parent proces, user
- Processor state information (afhankelijk van type processor)
  - User-Visible Registers (user mode vs kernel mode)
  - Control and Status Registers (PSW)
    Program counter, Condition codes[7], Status information[8].
  - Stack Pointers
    Wijzen naar de top van de stack.
- Process Control Control Information
  - Scheduling and State Information
    Prioriteit, process toestand, scheduling info
  - Priviles, geheugen info, …

[7] Resultaat van recente arithmetische/logische operaties
[8] Interrupt enabled/disabled flags en executie mode

4 PROCESBESTURING

4.1 UITVOERINGSMODI

Bepaalde instructies kunnen enkel uitgevoerd worden in system (of kernel) mode. Deze instructies omvatten het lezen en het bewerken van een controle register –zoals PSW-, primitieve I/O instructies en instructies dat te maken hebben met geheugen beheer. Ook zullen bepaalde regio's van het geheugen toegankelijk zijn voor system mode executie –die dus niet toegankelijk zijn in user mode-.

Het is nodig om het onderscheidt te maken tussen deze 2 modi, vermits het noodzakelijk is om het OS en de *key operating system tables* –zoals process control blocks- te beschermen interferentie door *user programs*.

In het PSW staat er een bit die aangeeft in welke mode het proces uitgevoerd moet worden. Deze kan bit kan verandert worden tijdens executie door bepaalde events. Deze events zijn een oproep aan een OS service of een interrupt. Wanneer het proces klaar is met uitvoeren in kernel mode keert deze terug in de user mode, dit door de bit-wisseling 1 → 0.

4.1.1 PROCES CREATIE

De creatie van een nieuw proces gaat als volgt:

- Proces ID toewijzen (door OS)
  Deze ID is de nummer van de *entry* in de *primary process table*.
- Alloceren van het geheugen (door OS)
  Dit voor alle elementen van het *process image*.
- Initialiseren van het *process control block* (door OS)
- Koppelingen maken (door OS)
  Opnemen van het nieuwe proces in rijen (*Ready* of *Ready/Suspended queues*) als het OS doet aan *scheduling*.

4.1.2 WISSELEN VAN PROCESSEN

Mechanisme voor *interrupting* van het uitvoeren van een proces:

	Oorzaak/Gevolg
Interrupt	Extern
• Clock interrupt	• Maximum toegelaten eenheid van tijd (Time slice) is afgelopen. Het proces komt in de ready staat en een ander proces wordt ge-*dispatched*.
• I/O interrupt	• I/O device is toegankelijk (I/O is voltooid) De processen die staan te wachten om dit I/O device te nuttigen worden uit de ready staat gehaald.
• Memory fault	• Pagina zit niet in het hoofdgeheugen Het process komt een *virtual memory address reference* tegen die niet in het hoofdgeheugen zit. Het OS brengt het *block* –die in het sec. geheugen zat- in het hoofd geheugen.
Trap	Intern komt er een fout voor. Het OS bepaald of de fout fataal is. Zo ja, geraakt het proces in de *Exit state*.
Supervisor call	Intern wordt er expliciet zelf een oproep gedaan van het programma aan het OS. Dit proces kan in de *blocked state* terecht komen. Bijvoorbeeld het verzoek tot I/O.

4.1.3 WISSELEN VAN MODUS

Als er een *interrupt* aan het wachten is zal de processor de PC op het begin van de routine zetten die de *interrupt* afhandelt en van user naar kernel mode overgaan.

Hierdoor moet de context van het proces dat *interrupted* is moet opgeslagen worden in het controle blok van het –*interrupted*- proces. Als we weten welke informatie wijzigt door de *interrupt handler* dan weten we welke informatie we moeten opslaan. Dus de *processor state information* dient opgeslagen te worden.

Het afhandelen van de *interrupt* is afhankelijk van het soort *interrupt*.
Bij een I/O *interrupt* wordt er een bevestiging gestuurd naar het I/O apparaat.
Bij een clock *interrupt* zal de controle worden overgedragen aan de dispatcher.

Opmerking: In de meeste OS wilt een interrupt niet noodzakelijk zeggen dat er een proceswisseling zal zijn.

We zien dus ook in dat er zeer weinig overhead is.

4.1.4 WISSELEN VAN PROCESTOESTAND

In tegenstelling tot een mode switch waarbij het mogelijk is dat het proces in *Running state* blijft, zal er veel overhead gemoeid zijn met een *full process switch*.

Volgende stappen worden dan ondernomen:

1. Opslaan van de context van de processor (= van het proces)
2. Bijwerken van het PCB[9] van het huidige proces
   Veranderen van de staat onder andere (Running naar …)
3. Verplaatsen van het PCB naar gepaste queue
4. Selecteren van een ander proces
5. Bijwerken PCB van het nieuwe proces
6. Bijwerken van data structuren voor geheugenbeheer
7. Terugbrengen van de context van dat proces die het had toen hij de laatste keer in de *Running state* verkeerde.

[9] Process Control Block

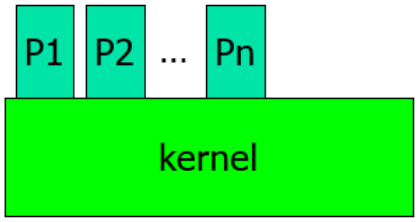
5 UITVOERING VAN HET OS

5.1 PROCESLOZE KERNEL

De kernel wordt buiten de processen uitgevoerd. Het OS heeft zijn eigen geheugengedeelte en zijn eigen *system stack* voor *controlling procedure calls* en *returns*.

Een OS *call* gaat gepaard met veel overhead.

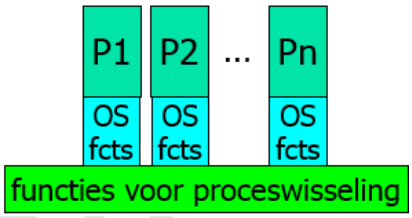
Het concept van processen is enkel toepasbaar bij *user programs* en het OS wordt dus aparte entiteit uitgevoerd in *privileged mode*.



5.2 UITVOERING BINNEN GEBRUIKERSPROCESSEN

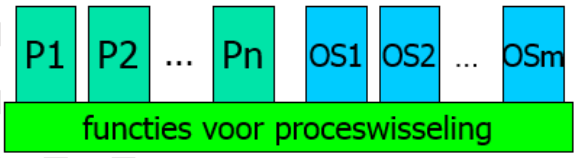
Het OS wordt in de context van een *user process* uitgevoerd. Hierdoor is er enkel een mode wisseling nodig, en zo is er dus weinig overhead bij een OS *call*.

OS code en data zitten dan wel weer in *shared address space* en worden door alle *user processes* gedeeld.



5.3 PROCES GEBASEERD OPERATING SYSTEM

Dit is een complexe kernel die bestaat uit meerder aparte processen. Het OS wordt dus geïmplementeerd als een collectie van *system processes.* Hierdoor wordt het OS modulair.



Opmerking:
Voor sommige OS *calls* zullen er zeer lichte *context switches* nodig zijn.

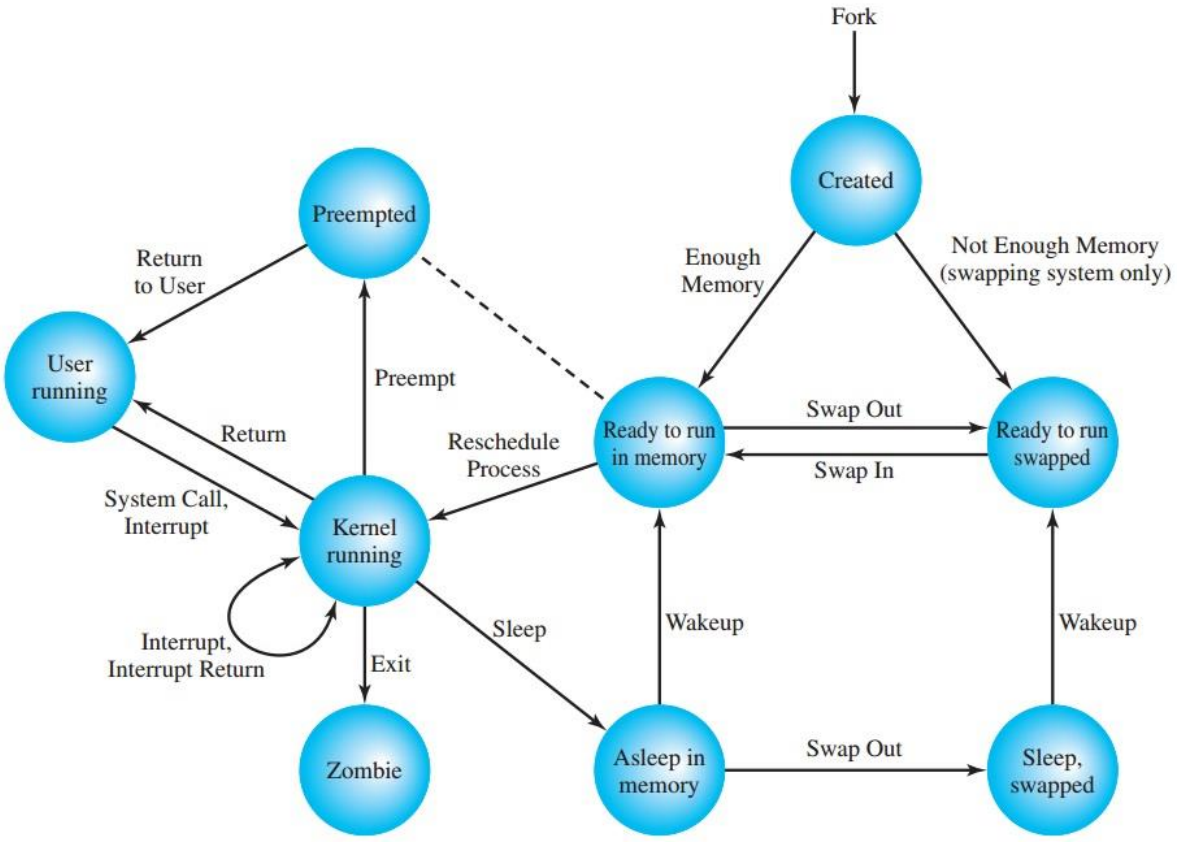
6 PROCESBEHEER IN UNIX SVR4

UNIX System V maakt gebruik van de uitvoering binnen gebruikersprocessen. Waardoor er 2 categorieën van processen zijn; *system processes* en *user processes*.
System processes runnen in *kernel mode* en voeren OS code uit. *User processes runnen* in *user mode* om gebruikersprogramma's uit te voeren en in *kernel mode* om instructies uit te voeren die in de *kernel* zitten –dit door een *system call*-.

6.1 PROCES STATEN

User Running ~Running	Uitvoeren in user mode
Kernel Running ~Running	Uitvoeren in kernel mode
Ready to run, in Memory ~Ready	Klaar om uit te voeren zodra de kernel het proces *schedulded*
Asleep in Memory ~Blocked	Proces zit in main memory en kan niet uitvoeren tot een bepaalde gebeurtenis plaatsvindt.
Ready to run, in Memory ~Ready Suspended	De *swapper* moet eerst het proces in *main memory* laden voor de *kernel* deze kan *schedulen*.
Sleeping, swapped ~Blocked Suspended	Proces zit in *secundary* memory en kan niet uitvoeren tot een bepaalde gebeurtenis plaatsvindt en het proces naar *main memory* wordt *geswapped*.
Preempted ~Ready	Het proces wilt terug komen van *kernel* naar *user mode*, maar de kernel stopt en voert een *process switch* uit. Opmerking: ready to run in Memory en Preempted zitten beiden te wachten aan éénzelfde *queue*.
Created ~New	Proces is net gecreëerd en is nog niet klaar om uit gevoerd te worden.
Zombie ~Exit	Proces bestaat niet meer, maar laat wel nog een *record* achter voor zijn *parent process*.

6.2 PROCES CONTROLE



6.2.1 CREATIE VAN EEN PROCES IN LINUX

`Fork()`
Creatie van een *child* proces waarbij de *parent* verder op de processor blijft uitvoeren.

`System()`
Creatie van een *child* proces waarbij het *parent process* om *blocked* komt te staan tot wanneer e *child* volledig is uitgevoerd.

`Exec()`
Vervangen van het huidige proces door een nieuw *child* proces.

6.2.2 CREATIE VAN EEN PROCES DOOR `FORK()`

1. Toewijzen van een *slot* in het *proces table* voor het nieuwe proces
2. Toekennen van een unieke ID code
3. Kopie maken van het *process image* van de *parent* (met uitzondering van het *shared memory*)
4. Verhogen van de tellers voor alle bestanden die in bezit zijn van de *parent*, om zo duidelijk te maken dat een ander proces ook deze bestanden bezit
5. Plaatsen van *child process* in *ready-to-run state*
6. ID code van *child* aan *parent* geven en een 0 *value* aan het *child process*

Dit alles wordt in *kernel mode* –in het *parent process* -uitgevoerd. Hierna kunnen volgende dingen gebeuren:

1. DEFAULT: blijven in het *parent* proces
2. Besturing overlaten aan *child process*
3. Besturing overdragen aan een ander proces

Bekijk de oefening 3.6 Intermezzo in de PowerPoint!

H4 - THREADS

1 PROCESSEN EN THREADS

Proceskenmerken waren het bezet van resources, *scheduling* en executie.
Een thread wordt nu gezien als een eenheid van *dispatching* en de eenheid van bezit van *resources* wordt gerefereerd naar een proces (of taak).

1.1 MULTITHREADING

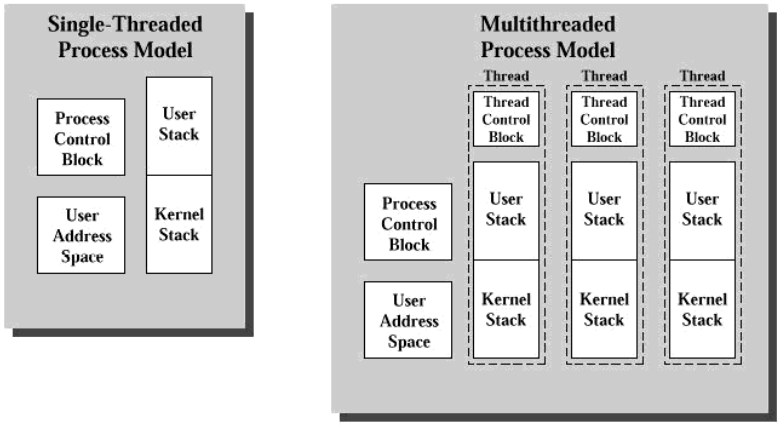
In het geval van multithreading ondersteunt het OS meerder, gelijktijdige paden van executie binnen één proces.

In deze omgeving wordt een proces (*unit of resource/*protection) geassocieerd met volgende zaken:

- Virtueel adres ruimte die het *proces image* bevat
- *Protected access to procesoors, other processes, files* en *I/O resources*

Een proces bevat één of meer threads, die het volgende bevatten:

- Een thread *execution state* (*Running, ready* en *blocked*)
- Een opgeslagen thread context (*not running*)
- Een executie *Stack*
- Toegang tot *shared memory* en *resources* van het proces.



De voordelen van threads zijn *performance* gerelateerd:

- snellere creatietijd (10x)
- snellere beëindigingstijd
- thread *switching* is sneller dan proces *switching*
- Efficiënte communicatie tussen threads
  dit door *shared memory* en *files*, waardoor de kernel niet telkens moet tussenkomen

Voorbeelden van het gebruik van threads in een *single-user mutliprocessing system*:

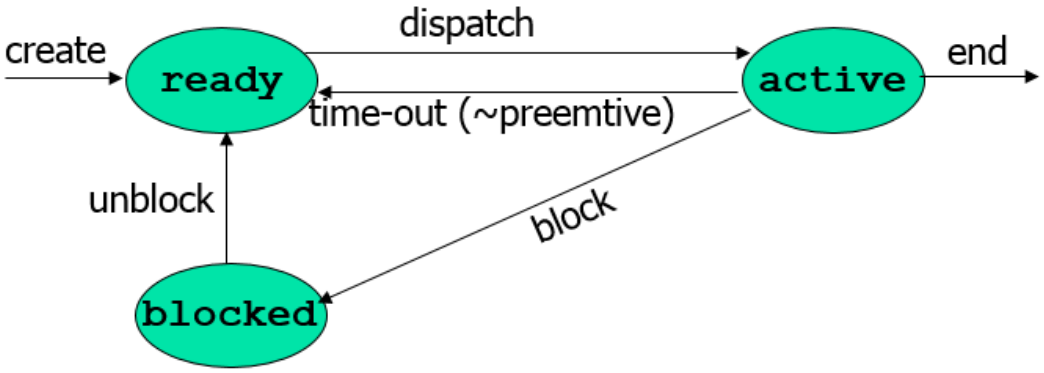
- Werk op achtergrond/voorgrond
  Spreadsheet: bewerkingen uitvoeren (achtergrond) en visualisatie + user input (voorgrond)
- Asynchrone verwerking
  RAM buffer periodiek naar schijf plaatsen
- Uitvoeringssnelheid verhogen
  Gegevens inlezen en de andere thread kan deze dan verwerken, men moet zo niet wachten op I/O operaties voor het verwerken van data.
- Modulaire programmastructuur
  Eenvoudigere en meer gestructureerde implementatie

Een applicatie zoals een file server maakt best gebruik van threads. Zo kan één thread de data ophalen van een DB en de andere kan dan de data visualiseren. Vermits data moet gedeeld worden in een file server is het handig om *shared memory* te hebben i.p.v. *mesaage passing* (proces niveau).

1.2 THREAD FUNCTIONALITEIT

1.2.1 THREAD STATEN

Opmerking:
Vermits alle *threads shared memory* delen zal dus een *swapping* en *termination* op proces-niveau gebeuren en niet op *thread*-niveau.



Voordelen van threads zijn de hogere performanties bij uni-processors, namelijk wanneer een I/O-request is uitgevoerd gaat deze over naar een ander proces.

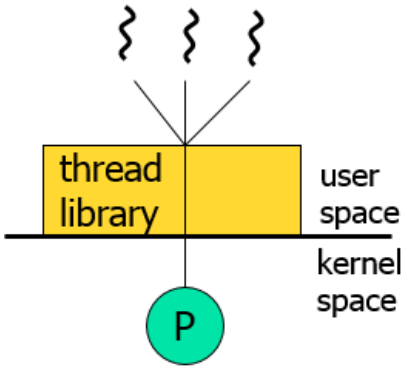
1.2.2 THREAD SYNCHRONISATIE

Vermits thread shared *memory* hebben zullen er dus synchronisatie-technieken nodig zijn.

2 TYPES VAN THREADS

2.1 USER-LEVEL THREADS

Bij een Pure ULT wordt het proces gecreëerd door de kernel. Binnen dit proces bevindt zich –default- één thread. Via een *Thread Library* kan deze meerdere threads '*spawnen*'. De kernel is dus niet op de hoogte van de creatie van meerdere *threads*.



Voordelen:

- Geen privileges van *kernel mode* nodig bij threadwisseling
  Hierdoor wordt de *overhead* van 2 *mode switches* uitgespaard
- *Scheduling* kan applicatie specifiek zijn.
- ULT's kan op elke OS gedraaid worden

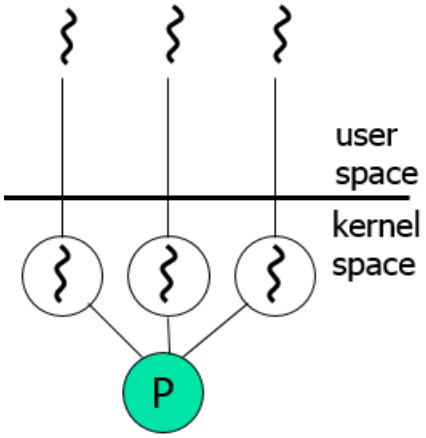
Nadelen:

- Alle threads binnen één proces worden geblokkeerd bij een *system call*.
  Opmerking: De staten van de threads blijven onveranderd ze hebben enkel 'geen betekenis' als het proces in de *blocked state* staat.
- Geen voordeel van multiprocessing

Om het probleem van *blocking threads* op te lossen kan men gebruik maken van *jacketing*. Hierbij worden blokkerende I/O oproepen omgezet in niet-blokkerende I/O omroepen. Via de *thread library* kan een *jacket routine* kijken of het I/O apparaat bezet is om zo de controle te geven aan een ander thread vooraleer een system I/O routine te benaderen.

2.2 KERNEL-LEVEL THREADS

Het thread *managment* wordt nu gedaan door de kernel. Men voorziet een API naar de *thread facility* in de kernel.



Voordelen:

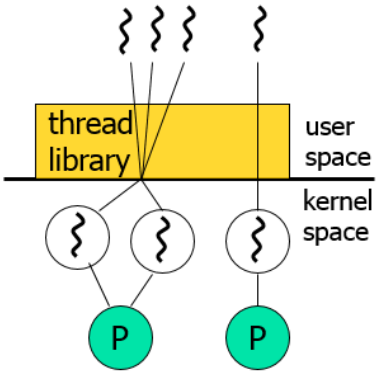
- Multiprocessing van threads binnen 1 proces mogelijk
  Dit door context informatie van de threads die worden onderhouden door de kernel
- Blokkering van thread heeft geen invloed op andere threads
- Kernelroutines kunnen ook threads bevatten

Nadeel:

- Het wisselen van threads kost een *mode switch* in de kernel (overhead).

2.3 GECOMBINEERDE METHODE

Voordelen van ULT en KLT combineren en nadelen ervan minimaliseren, mits een goed ontwerp!



3 SYMMETRISCHE MULTIPROCESSING

Voordelen van parallelle verwerking zijn prestatieverbetering en betrouwbaarheidsverhoging.

3.1 SMP ARCHITECTUUR

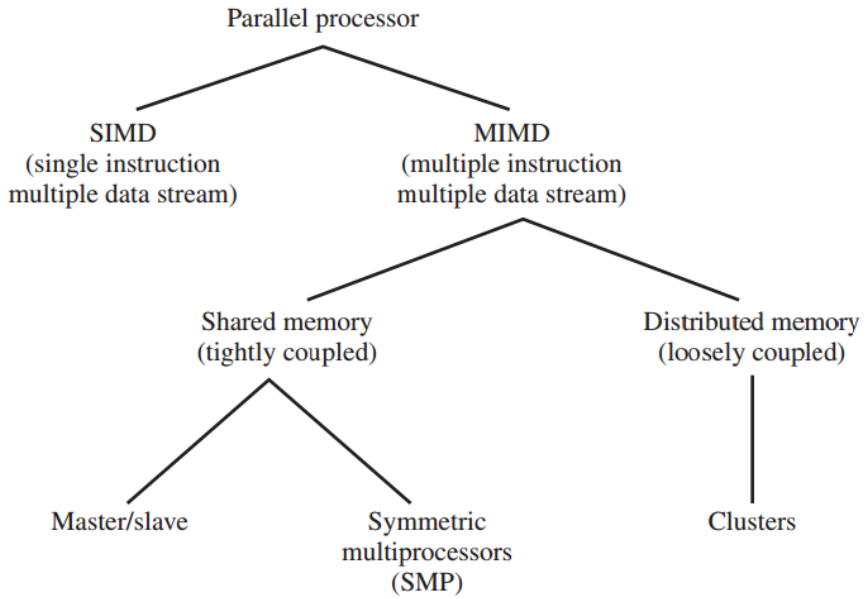
Single Instruction Single Data (SISD) stream
Een *single* processor –uniprocessor- voert één instructie *stream* uit op datatdie opgeslagen zit in een *single memory*.

Single Instruction Multiple Data (SIMD) stream
Meerdere *processing elements* –bronnen van data- worden bewerkt via één instructie. Dit kan handig zijn bij multimedia, verduisteren alle pixels of volume verhogen. Vector en Array processoren vallen onder deze categorie.

Multiple Instruction Multiple Data (MIMD) stream
Meerdere processoren voeren simultaan verschillende instructies op verschillende *data sets* uit.
Deze MIMD's kunnen verder worden opgedeeld a.d.h.v. hoe de processoren communiceren. Clusters zullen communiceren via *fixed paths* of via *network facility*. De *shared-memory multiprocessoren* daarentegen zullen communiceren via hun *shared memory*.

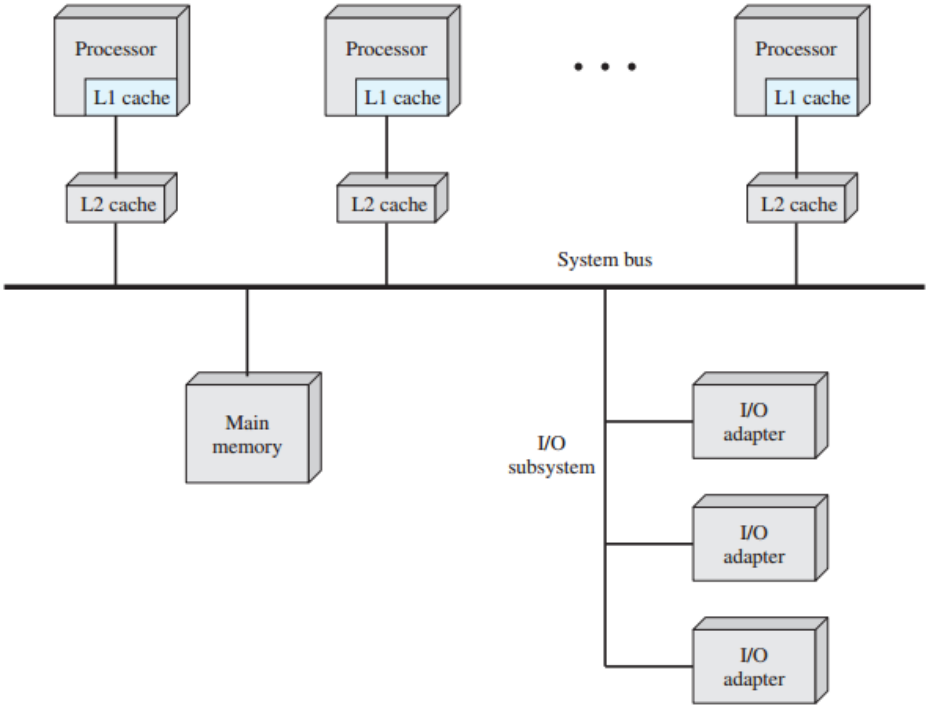
Bij een Master/Slave architectuur zal de OS kernel draaien op 1 vaste processor. Hierdoor is het ontwerp eenvoudig, maar een fout in de master kan het systeem neerhalen en de master is de bottleneck.

Bij een SMP architectuur zal de kernel op elke processor uitgevoerd kunnen worden. Hierdoor zit de kernel verspreidt en kan dus parellel worden uitgevoerd waardoor de prestaties verbeteren. Het nadeel hierbij is het synchroniseren van resources, kiezen van processen enz.



3.2 SMP ORGANISATIE

Het private cache geheugen houdt een *image* van een portie van het hoofgeheugen bij. Als nu een *word* in een cache wordt gewijzigd en deze niet wordt aangepast in het hoofdgeheugen, na een time-out bijvoorbeeld, dan kunnen er problemen optreden. Dit probleem noemt het *cache coherence problem* ~ correctheid van de informatie in de cache.



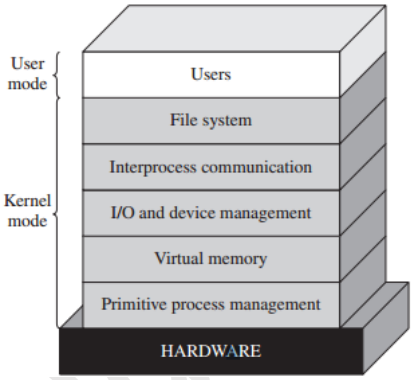
4 MICROKERNELS

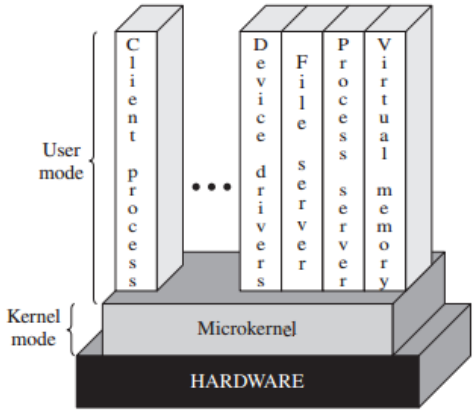
Een *microkernel* is een kleine kern van het OS die de basis is voor modulaire uitbreidingen.

4.1 MICROKERNEL ARCHITECTUUR

Het OS kan gestructureerd worden als een monolithisch blok maar deze structuur is niet beheersbaar en ongestructureerd.

Een andere vorm was de *layered kernel*, dit is en hiërarchische organisatie van functies. De meeste lagen zullen dan ook in kernel mode uitgevoerd worden. Door de interactie tussen de aanliggende lagen is de veiligheid bemoeilijkt alsook als men iets verandert in de ene laag zal dit consequenties hebben voor de andere.





Ben microkernel bevat de kernel enkel de essentiële functies, de kern, van het OS. De minder essentiële functies en applicaties worden ingebouwd in de microkernel en worden uitgevoerd in user mode. Deze -in gebruikersmodus- processen worden *server processes* genoemd.

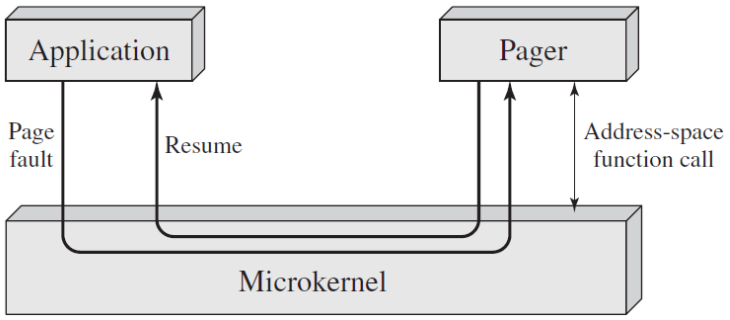
4.2 MICROKERNEL DESIGN

De microkernel moet de functies die 'vasthangen' aan de hardware en functies voor ondersteuning van servers en applicaties in gebruiker modus bevatten.

4.2.1 FUNCTION 1: LOW-LEVEL MEMORY MANAGEMENT

Wanneer er een *Page fault* wordt opgegooid in de applicatie ,door referentie naar een *page* die zicht niet bevindt in het hoofdgeheugen. De kernel zend hierbij een bericht naar het *pager* proces met de indicatie naar welke *page* is gerefereerd. De *pager* beslist dan of de *page* wordt geladen en alloceerd een *page frame*. Als de *page* toegankelijk is zal de *Pager* een *resume* bericht sturen naar de *kernel*.

De *kernel* zorgt dus voor de *mapping* tussen logische adresruimte en fysische adresruimte.



4.2.2 FUCNTION 2: INTERPROCESS COMMUNICATION (IPC)

De *kernel* beheert de poorten en wachtrijen aan die poorten van processen. Berichten kunnen verzonden worden tussen processen. Deze berichten kunnen wachten aan een specifieke poort van een specifiek proces.

4.2.3 FUNCTION 3: I/O AND INTERRUPT MANAGEMENT

De *kernel* zal niet instaan voor het afhandelen van *interrupts* maar deze worden doorgegeven aan *user-level* processen, via *messages*. De *kernel* houdt de *mapping* tussen het specifiek proces dat de *interrupt* afhandelt en de *interrupt*.

H5 - MUTUAL EXCLUSION AND SYNCHRONIZATION

1 PRINCIPLES OF CONCURRENCY

1.1 COMPETITION AMONG PROCESSES FOR RESOURCES

In het geval van *competition for resources*[10] zijn er 3 controle problemen.

1. **Mutual Exclusion**
   We moeten ervoor zorgen dat er maar één proces zijn *critical section* kan uitvoeren op een *critical resource*.
2. **Deadlock**
   Deadlock komt voor als er twee of meerdere processen op elkander moet wachten tot ze klaar zijn, en juist hierdoor nooit klaar geraken. Dit door de wederzijdse uitsluiting.
3. **Starvation**
   Dit komt ook weer door de wederzijdse uitsluiting. Een *starvation* treedt op wanneer een proces een bepaalde resource -telkens opnieuw- wordt geweigerd door de OS.

1.2 COOPERATION AMONG PROCESSES BY SHARING

In het geval van *cooperation among processes by sharing*[11] hebben we weer te maken met *mutual exclusion*, *deadlock* en *starvation*. Dit omdat de gedeelde data zijn opgeslagen op *resources.*

Opmerking:
Er komt een nieuw probleem bij kijken, namelijk *data coherence*.

1.3 COOPERATION AMONG PROCESSES BY COMMUNICATION

Door de communicatie is er een manier om de activiteiten te synchroniseren/coördineren tussen processen.

Nu hebben we niet te maken met *shared resources* dus is het *mutual exclusion* probleem al opgelost.
Maar omdat processen kunnen blijven wachten op berichten van elkander kan er wel *deadlock* ontstaan, alsook *starvation*.

1.4 REQUIREMENTS FOR MUTUAL EXCLUSION

1. *Mutual Exclusion* moet opgedrongen worden.
2. Een proces dat stopt in het *noncritical section* mag andere processen niet hinderen. De andere processen mogen dus komen in het mutex gedeelte.
3. Geen *deadlock* of *starvation.*
4. Wanneer geen enkel proces in de *critical section* zit, elk proces die toegang wilt tot de mutex krijgt deze zonder *delay*.
5. Geen assumpties rond precessorsnelheid of aantal processors.
6. Een proces blijft ni de *critical section* voor een eindige tijd.

[10] Dit komt voor wanneer twee of meerdere processen *geen weet* van elkaar hebben.
[11] Hierbij zijn de processen onbewust op de hoogt van elkaar, dit door gedeelde data. Dit is typisch bij *Threads*.

2 MUTUAL EXCLUSION: HARDWARE SUPPORT

Hieronder bespreken we enkele strategieën om hardware matig *mutual exclusion* te verkrijgen.

2.1 STRATEGY 1: INTERRUPT DISABLING

Deze oplossing is enkel geldig voor *uniprocessor systems*. Dit omdat processen elkaar niet kunnen overlappen en enkel kunnen *interleaven*. Vermits een proces enkel gestopt wordt door een *OS call* of een *interrupt* is het voldoende om het volgende te doen:

```
while(true){
        /* disable interrupts */
        /* critical section */
        /*enable interrupts */
}
```

Maar deze oplossing is niet efficiënt en is helemaal geen *multi-programming* meer.

2.2 STRATEGY 2: SPECIAL MACHINE INSTRUCTIONS

Er bestaan speciale machine instructies die twee actie *atomically* kunnen uitvoeren, d.w.z. zonder onderbroken te worden.

Een gedeelde variabele `bolt` is geïnitialiseerd op 0.
Het enige process dat toegang heeft tot de kritische sectie is degene waarbij `bolt` gelijk is aan 0.
In het andere geval zal het proces in *busy waiting mode* komen te staan.

```
/* ATOMAIRE MACHINE INSTRUCTIE */
boolean testset(int i){
        if(i==0){i=1; return true;}
        else {return false;}
}
/* mutex programma */
void p(int i){
        while(true){
                while(!testset(bolt)) {
                        //busy waiting
                }
                /* critical section */
                bolt=0;
                /* non-critical section
*/
        }
}
```

Een andere manier is via de `exchange(int x, int y)` instructie.
Dit komt op hetzelfde neer een proces waarbij bolt staat op nul zal toegang hebben tot de kritische sectie, de andere processen zullen terug in *busy mode* staan.

```
/* mutex programma */
void p(int i){
        while(true){
                int key=1;
                do{

        exchange(key,bolt);
                        //busy waiting
                }
                while(key!=0)
                /* critical section */
                bolt=0;
                /* non-critical section
*/
        }
}
```

Als bolt op 0 staat zal er geen process in de kritische sectie zitten. Als bolt gelijk is aan 1 zal er maar één proces in de kritische sectie zitten –waarbij de *key value* nul is.

De voordelen van het gebruik van speciale machine instructies zijn:

- Werkt op zowel uni- als multiprocessors
- Het is simpel en gemakkelijk te verifiëren
- Er kunnen verschillende kritische secties onderhouden met elk hun eigen variabele

Maar er zijn een aantal grote nadelen:

- *busy waiting*
  Tijdens het wachten zullen de processoren kostbare processortijd verspillen.
- *Starvation* is mogelijk
  Arbitraire selectie van het volgende proces kan ervoor zorgen dat een proces geen toegang krijgt.
- *Deadlock* is mogelijk
  Als een proces de *lock* neemt en daarna wordt ge-*interrupted* waarbij een proces met een hogere prioriteit de processor krijgt. Deze voert telkens de lus uit, waardoor het eerste proces nooit uit zijn kritische sectie geraakt.

3 SEMAPHORES

Een *Semaphore* is een object gebruikt om processen te stoppen of signaleren. Een *Semaphore* heeft de attributen `int count`, die de waarde voorstelt van de *Semaphore* en dan `queueType queue`, die een lijst van wachtende processen voorstelt.

Er kunnen 3 methodes losgelaten wordne op de *semaphore* namelijk `init()`, `semWait()` en `semSignal()`.

Wanneer de `count` groter of gelijk is aan nul zullen er geen wachtende processen zitten in de `queue`. In het andere geval zullen er dus |count| processen wachten.

Als een proces `semWait(s)` uitvoert zal de `count` met 1 verlaagd worden en in het geval de `count` hierdoor negatief of nul wordt, zal het proces ge-*blocked* worden.

Als een proces `semSignal(s)` uitvoert zal de `count` met 1 verhogen en in het geval hierdoor de `count` groter wordt dan nul, zal er een proces opgewekt worden.

Opmerkingen:
Men weet niet voor men de teller decrementeerd of het proces ge-*blocked* gaat worden.
Als er een proces een semaphore incrementeerd en een ander proces opwekt beginnen beiden *concurrent* uit te voeren. Hierdoor weet men niet, op een uniprocessor, welk proces zal starten, als er zelfs één van beiden begint.
Als je een *signal* verstuurt weet je niet zeker of er nog processen staan te wachten.

VARIANT: *BINARY SEMPAHORES*

Hierbij zal de semaphore geïnitialiseerd worden op 0 of 1.
Bij een `semWaitB` zal er gekeken worden of hij de waarde nul heeft, zo ja zal het proces ge-*blocked* worden. In het andere geval zal men de semaphore op nul zetten en het proces verder uitvoeren.
Bij een `semSignalB` wordt er weer gekeken of de waarde nul is, zo ja zal het proces ge-*unblocked* worden.
Als er geen processen ge-*blocked* zijn zal de waarde van de semaphore op één gezet worden.

Het enige verschil hierbij is dat we nu niet weten hoeveel processen er staan te wachten aan de Semaphore.

3.1 MUTUAL EXCLUSION

VARIANT: MUTEX

Deze *mutual exclusion lock* is een programmeerbare *flag* die een proces kan nemen en vrijgeven.
Het verschil tussen een *mutex* en een gewone *semaphore* ligt in het feit dat een proces die de *mutex* neemt deze ook moet vrijgeven.

Bij sterke semaphoren hebben geen kans op *starvation*, door de FIFO implentatie.

Bekijk volgende problemen: Producer/Consumer problem; Bounded buffer

3.2 IMPLEMENTATION OF SEMAPHORES

De `semWait` en `semSignal` operaties moeten atomische primitieven zijn!

Deze kunnen verwezelijkd worden door Hardware of firmware implementaties, alsook pure software *schemes*. Maar in de praktijk wordt er gebruik gemaakt van hardware-supported *schemes*. Via de compare_and_swap

methode met het nadeel dat het proces in *busy waiting mode* staat –maar dit is gelimiteerd- en via *interrupt disabling* –enkel voor uniprocessoren-.

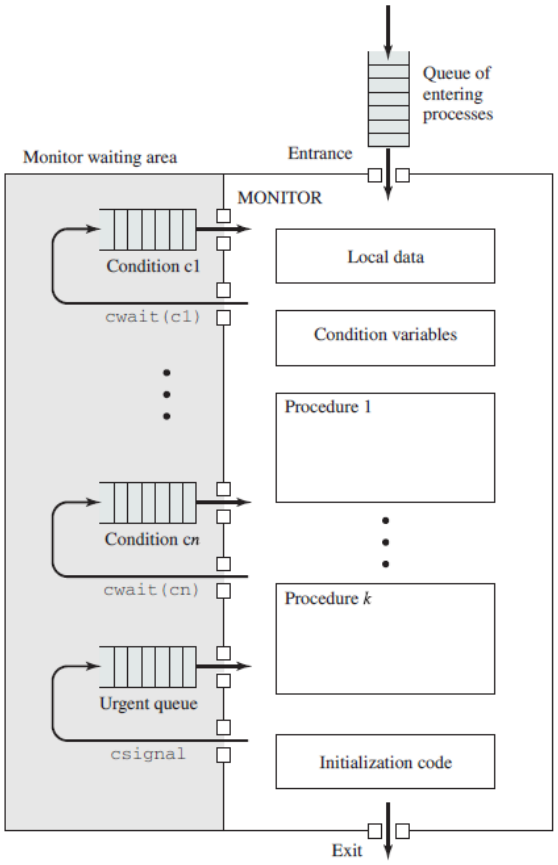
4 MONITORS

Het probleem bij semaphoren is het feit dat de P() en V() helemaal verspreidt zit over het programma, waardoor het overzicht nogal vaag wordt.

4.1 MONITOR WITH SIGNAL

Karakteristieken van een monitor:

- Lokale variabelen zijn alleen toegankelijk door de monitors *procedures*.
- Een proces komt in de monitor door het aanroepen van zo'n *procedure*.
- Enkel één proces kan uitgevoerd worden binnen en monitor.



Als een proces aan het uitvoeren is binnen een monitor en deze in *blocked* toestand komt te staan, is het de bedoeling dat de monitor terug vrijgegeven wordt. Dit is mogelijk door het gebruik van *condition variables* die zich in de monitor bevinden.

`cwait(c)` *Suspend execution* van het proces aan de conditie `c` en geef de monitor terug vrij voor andere processen.

`csignal(c)` *Resume execution* van een proces die geblokkeerd is door `cwait` op dezelfde conditie.

Deze operaties zijn verschillend van de operaties bij semaphoren.

Bij Hoare's definitie van een monitor moet het proces die een `csignal` uitvoert ofwel direct de monitor verlaten ofwel moet deze onmiddellijk ge-*blocked* worden aan de monitor. Dit omdat het opgewekte proces aan de queue actief wordt –door de `csignal`-.

Hierdoor zal het proces die signaleert twee *process switches* moeten maken als hij nog niet klaar is met de monitor, één om het opgewekte proces te *blocken* en een om verder uit te voeren op de monitor wanneer deze vrij komt. Men moet er ook voor zorgen dat het proces dat opgewekt wordt door de `csignal` deze is die geactiveerd wordt **voor** een proces die staat te wachten aan de monitor –en niet aan die conditievariabele-. Deze –opgewekt proces- komt dan ook in de *urgent queue* terecht.

Een alternatief voor het eerste probleem is enkel signaleren op het einde van een monitor.

Het opgewekte proces heeft dus voorrang daarom dat er een `if-` wordt geplaatst i.p.v. een `while`-lus. Men weet zeker dat het opgewekte proces niet meer moet wachten en zo ook niet meer achteraan de *queue* komt te staan.

4.2 MONITOR WITH NOTIFY

Lampson and Redell 's monitoren zien er hierdoor iets anders uit.

Wanneer een proces die aan het uitvoeren is in de monitor een `cnotify(x)` stuurt, wordt de `x queue` genotificeerd maar het gesignaleerde proces blijft uitvoeren. Maar omdat men niet zeker is dat er geen proces voor hen de monitor krijgt zal men opnieuw de conditie moeten controleren. De problemen bij Hoare zijn

hierdoor opgelost, maar men is minder eerlijk vermits het opgewekte proces nog moet *strijden* met andere processen.

5 MESSAGE PASSING

Via *message passing* is het dus mogelijk om zowel *mutual exclusion* als communicatie te voorzien tussen processen.

5.1 SYNCHRONISATION

Wanneer een *send primitive* is uitgevoerd kunnen er 2 dingen gebeuren; het zendende proces wordt geblokkeerd tot het bericht is toegekomen (*blocking send*) of het proces blijft gewoon verder uitvoeren (*non-blockign send*).

Dit is gelijkaardig aan een *receive primitive* waarbij een *blocking receive* ervoor zorgt dat het proces moet wachten tot een bericht aangekomen is. Het proces kan ook gewoon verder uitvoeren (*non-blocking receive*). Er bestaat ook nog zoiets als *polling* waarbij er af en toe een non-blocking *receive* wordt uitgevoerd.

Hiermee kunnen volgende combinaties gemaakt worden:

- ***Blocking Send – Blocking Receive***
  Rendezevous protocol, beide processen zijn geblokt tot het bericht aangekomen is.
- ***Non-Blocking Send – Blocking receive***
  Voorbeeld, Server process. Hierbij doet de server enkel nuttig werk wanneer er een bericht is aangekomen.
- ***Non-Blocking send – Non-Blocking receive***
  Niemand moet hier wachten.

5.2 ADDRESSING

5.2.1 DIRECT ADDRESSING

Bij directe adressering moet er op voorhand geweten zijn welk *processID* het proces, die je wil bereiken, heeft. (Impliciet)

```
send(processID,msg)
receive(processID, msg)
```

Bij een printer-server process, bijvoorbeeld, kan het nuttig zijn om alle berichten te ontvangen. (Expliciet)

```
receive(processID)
```

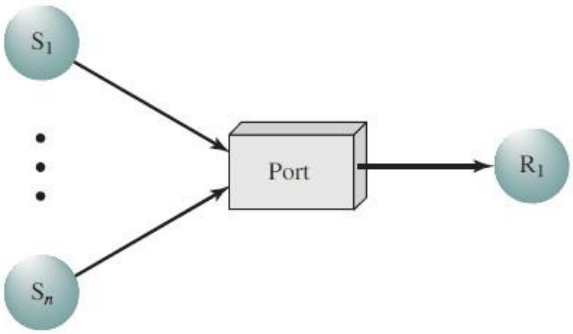
5.2.2 INDIRECT ADDRESSING

In het geval we op voorhand het *processID* niet kennen, maken we gebruik van een *shared data structure*, *mailboxes*.

De *mailbox* is dan een *queue* (buffer) waarbij een proces berichten achterlaat en een ander proces het bericht terug opneemt.

5.2.2.1 RELATIONSHIP BETWEEN SENDER AND RECEIVER

- **1 Zender – 1 Ontvanger**
  private communicatie en geen *interference* door anderen
- **Veel Zenders – 1 Ontvanger**
  Client/server interactie, waarbij de mailbox wordt gezien als een *port* van de ontvanger. (C/P problem)
- **1 Zender – Veel Ontvangers**
  Bericht broadcast
- **Veel Zenders – Veel Ontvangers**
  Meerdere servers bieden *concurrent service* aan

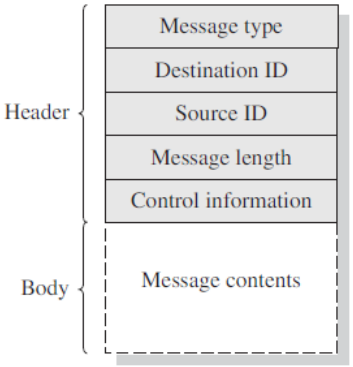


(b) Many to one

5.3 MESSAGE FORMAT

Een bericht kan een vaste lengte hebben die gemakkelijker door het OS kan behandeld worden.

Een bericht kan ook een variabele lengte hebben die dan meer flexibel is voor het proces.



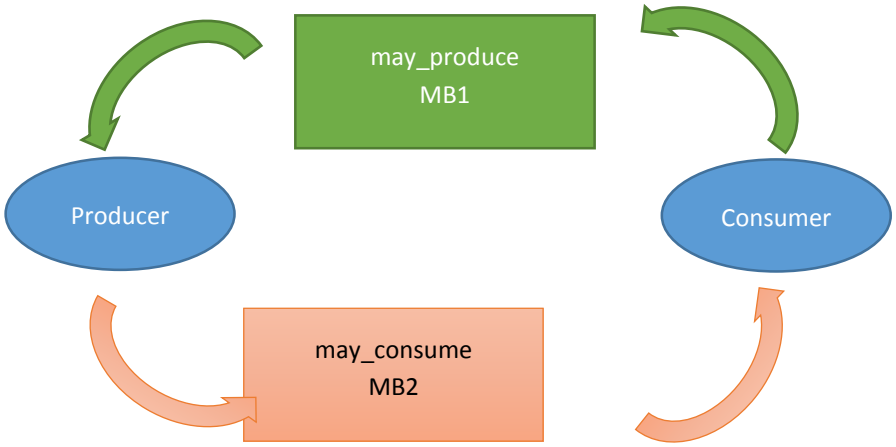
5.4 MUTUAL EXCLUSION

Voor deze implementatie gaan we uit dat we te maken hebben met *blocking receive* en *non-blocking send*.

De *mailbox* wordt geïnitialiseerd waarbij hij één *null* bericht bevat, de *token*. Als een proces zijn kritische sectie wilt uitvoeren zal hij eerst een bericht uit de *mailbox* willen halen. Als deze leeg is zal het proces geblokkeerd worden. In het andere geval zal deze de token nemen en terug plaatsen in de mailbox als hij zijn kritische sectie heeft uitgevoerd.

Er wordt dus vanuit gegaan –bij concurrent benaderen van mailbox- dat er maar één proces de token krijgt –de rest in *block* toestand- en dat als de *mailbox* leeg is dat alle processen ge-*blocked* worden. Als deze token opnieuw vrijgegeven is dan zal eer maar 1 proces geactiveerd worden en het bericht krijgen.

Een voorbeeld voor dit probleem is het *producer/consumer problem* waarbij er 2 mailboxen worden aangemaakt die elk *n tokens* bevatten. De *n tokens* staan voor de *n* plaatsen in buffer. De *Producer* moet telkens een token uit de eerste *mailbox* halen om deze nadien te plaatsen in de tweede *mailbox*. Deze kan dan weer genomen worden door een *Consumer* die dit token –uit MB2- terug plaats in MB1 na zijn uitvoering –van zijn kritische sectie-.

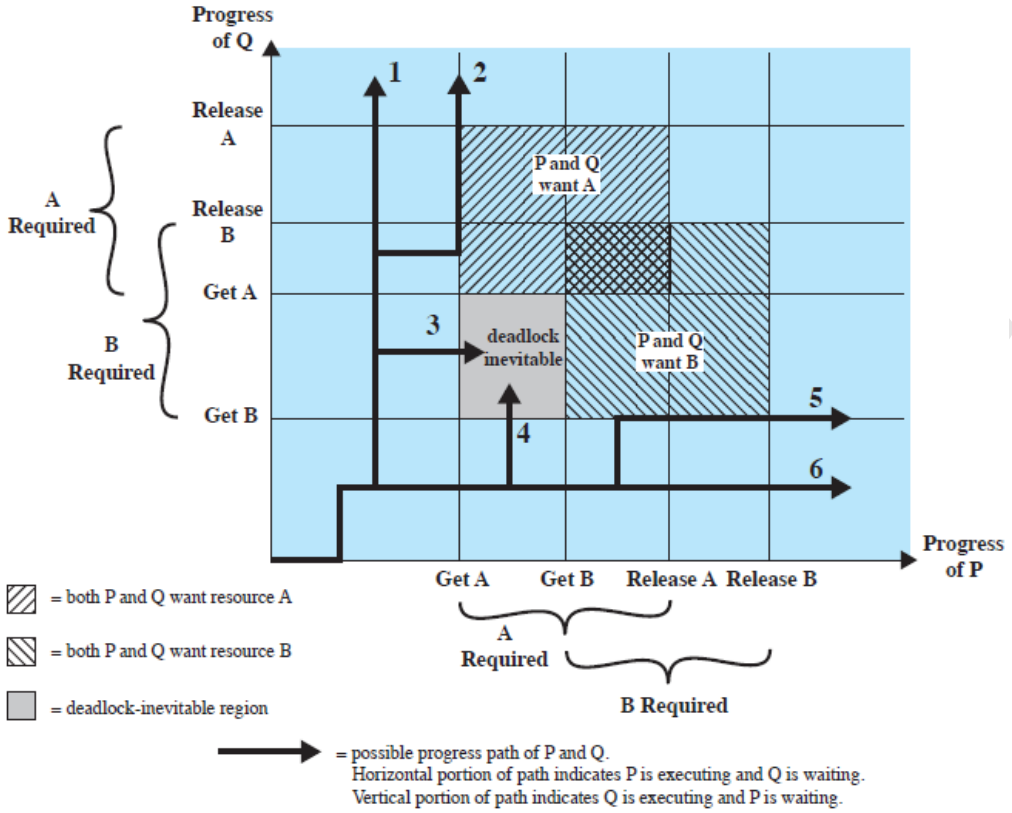


H6 – CONCURRENCY
DEADLOCK & STARVATION

1 PRINCIPLES OF DEADLOCK

Deadlock is het permanent geblokkeerd zijn van processen die '*het tegen elkander opnemen*' om *system resources* of die met elkaar communiceren.

Via een *joint progress diagram* kunnen we, in dit geval, 2 processen die 2 *recourses* willen bemachtigen illustreren.



We kunnen 2 algemene catogorien van *resources* onderscheiden, de herbruikbare en de consumeerbare.

1.1 REUSABLE RESOURCES

Het proces kan deze *resource* bemachtigen om nadien deze terug vrij te geven om opnieuw te gebruiken door andere processen.

Voorbeelden zijn processoren, geheugen, databanken, semaphoren,…

Process P

Step	Action
p_0	Request (D)
p_1	Lock (D)
p_2	Request (T)
p_3	Lock (T)
p_4	Perform function
p_5	Unlock (D)
p_6	Unlock (T)

Process Q

Step	Action
q_0	Request (T)
q_1	Lock (T)
q_2	Request (D)
q_3	Lock (D)
q_4	Perform function
q_5	Unlock (T)
q_6	Unlock (D)

Wanneer het multiprogramming systeem net na p1 overgaat naar het proces Q die dan weer wordt gestopt bij q2 omdat D al in bezit is van P kan deadlock optreden. Vermits T al ge-*locked* is door proces Q. Beiden staan te wachten op het vrijgeven van de *resource*die ze bezitten, P heeft D en Q heeft R.

1.2 CONSUMABLE RESOURCES

In dit geval kan een *resources* gemaakt (*produced*) worden en vernietigd (*destroyed*) worden.

Voorbeelden zijn *interrupts*, *signals*, berichten, informatie in de I/O buffers,…

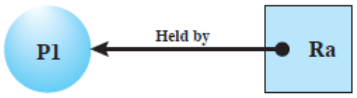
P1	P2
…	…
Receive (P2);	Receive (P1);
…	…
Send (P2, M1);	Send (P1, M2);

Deadlock kan voorkomen wanneer we in de situatie zitten van *Blocking receive*. Hierbij zal het ontvangend proces geblokkeerd worden tot het bericht is ontvangen.

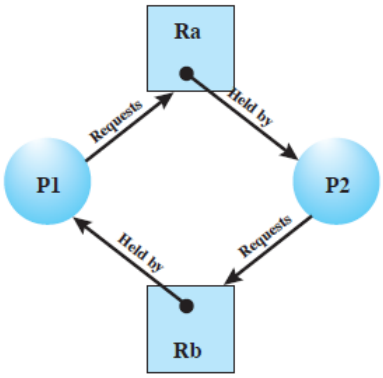
1.3 RESOURCE ALLOCATION GRAPHS

Een *resource allocation graph* karakteriseert de allocatie van *resources* door processen. Waarbij een cirkel staat voor een proces en een vierkant een *resource*, het aantal resources wordt aangeduid door zwarte punten.

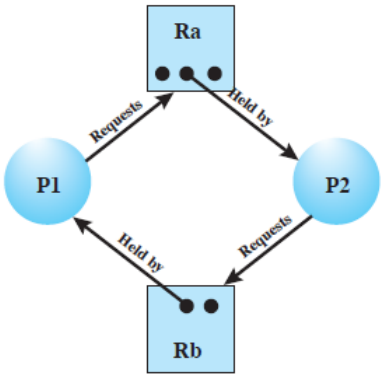
(a) Resouce is requested



(b) Resource is held



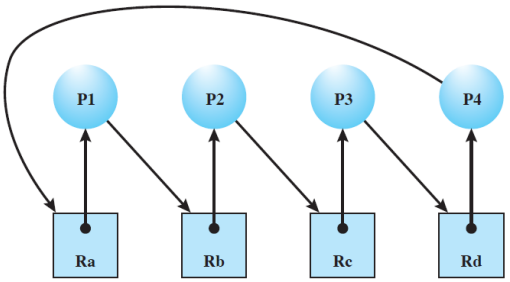
(c) Circular wait



(d) No deadlock

1.4 THE CONDITIONS FOR DEADLOCK

1. Mutual exclusion
2. Hold and wait
   Vasthouden van *resources* tijdens het wachten op de toegang tot andere *resources*.
3. No preemption
   Resources kunnen niet afgenomen worden van een proces.
4. Circular wait
   Dit is een gesloten ketting van processen, waarvan elk proces minstens 1 *resource* van het volgende proces nodig heeft.
   Dit is eigenlijk een potentiele consequentie van de eerste 3 condities.



2 DEADLOCK PREVENTION

Om deadlock te voorkomen moeten we ofwel één van de 3 eerste condities zien te voorkomen, dit is de indirecte methode. De Directe methode zal zorgen dat de laatste conditie niet kan voorkomen.

- Mutual Exclusion
  Het is niet mogelijk om dit te voorkomen.
- Hold and Wait
  Alle *requests* van het proces worden op hetzelfde moment uitgevoerd, en het proces is geblokkeerd tot alle *requests* simultaan voldaan zijn. Het proces moet dus wachten tot de volledige allocatie klaar is, alle *resources* moeten dus tegelijkertijd vrij zijn. Dit kan lang duren en gealloceerde *resources* kunnen lang niet gebruikt worden. Een proces weet niet altijd op voorhand alle nodige *resources*.
- No preemption
  We willen de vastgehouden *resources* afnemen van één proces en deze geven aan een andere. Dit is enkel mogelijk als de *resource* zijn staat kan opgeslagen worden en nadien verder opnieuw gebruikt kan worden. Een voorbeeld is hier RAM allocatie waarbij het swappen van het proces zo'n *resource preemption* is. (En is niet mogelijk bij een printer)
- Circular Wait
  We definiëren hierbij een lineaire ordering van *resource types*. Als er allocatie is van het type R kan er enkele gevraagd worden voor types die volgen op R. Hierdoor zal er geen circulaire vorm meer zijn. Ook dit is een inefficiënte manier van werken.

3 DEADLOCK AVOIDANCE

Het verschil bij deadlock *avoidance* is dat de 3 condities mogen doorgaan maar dat ze zo gekozen worden dat het deadlock *point* nooit bereikt zal worden, waardoor deze meer *concurrency* toelaat.

Hierdoor zal er dynamisch moeten gekeken worden of de *resource* allocatie wordt toegestaan of niet, om dit mogelijk te maken moeten we weten wat de toekomstige *resource requests* van de processen zullen zijn.

3.1 PROCESS INITIATION DENIAL

We kunnen hier dus gebruik maken van 2 staten, de `safe_state` die niet kan leiden tot deadlock en de `unsafe_state` die kan leiden tot deadlock.

```
if (request > avail) suspend_process
simulate_resource_allocation
if (safe_state) then do_allocation
else    restore_previous_state
        suspend_proces
```

Vectoren

R	*Resources*	totaal aantal *resources* van het system
V	*Available*	Totaal aantal *resources* die nog niet zijn gealloceerd

Matrices

C	*Claim*	C_{ij} benodigde resource j voor proces i
A	*Allocation*	A_{ij} gealloceerde resource j door proces i

Volgende formules vloeien hieruit voort:

$$R_j = V_j + \sum_{i=1}^{n} A_{ij}$$

Alle *resources* zijn ofwel vrij ofwel gealloceerd.

$$C_{ij} \leq R_j$$

Geen enkel proces kan meer *resources* opeisen dan er aanwezig zijn in het systeem.

$$A_{ij} \leq C_{ij}$$

Geen enkel proces kan meer *resources* alloceren (van een bepaald type) dan de oorspronkelijk opgeëiste hoeveelheid (van dat type).

Er mag dus enkel een Proces P_{n+1} als

$$R_j \geq C_{(n+1)j} + \sum_{i=1}^{n} C_{ij}$$

Opmerking:
De matrix $C - A$ wordt ook vaak gebruikt omdat die het aantal *resource* bevat die maximaal nog opgeëist kunnen worden.

3.2 RESOURCE ALLOCATION DENIAL

We gaan als volgt te werk:
Wanneer een proces een *request* maakt voor een set van *resources*, gaan we ervan uit dat deze worden toegestaan. We *updaten* het systeem en bekijken of we nog altijd in een *safe state* zitten, zo ja zal de *request* toegestaan worden, zo niet zal het proces in *blocked state* komen te staan.

De voordelen van het *Banker's algorithm* is dat het niet nodig is om processen te '*rollbacken*', alsook dat het minder restrictief is dan deadlock *prevention*.

De nadelen daarentegen zijn dat het maximale benodigde *resources* op voorhand moet gekend zijn, dat het aantal *resources* vast liggen en dat het proces niet mag stoppen terwijl hij nog *resources* vasthoudt.

```
struct state {
       int resource[m];
       int available[m];
       int claim[n][m];
       int alloc[n][m];
}
```

(a) global data structures

```
if (alloc [i,*] + request [*] > claim [i,*])
     < error >;                                  /* total request > claim*/
else if (request [*] > available [*])
     < suspend process >;
else {                                                  /* simulate alloc */
     < define newstate by:
     alloc [i,*] = alloc [i,*] + request [*];
     available [*] = available [*] - request [*] >;
}
if (safe (newstate))
     < carry out allocation >;
else {
     < restore original state >;
     < suspend process >;
}
```

(b) resource alloc algorithm

```
boolean safe (state S) {
    int currentavail[m];
    process rest[<number of processes>];
    currentavail = available;
    rest = {all processes};
    possible = true;
    while (possible) {
        <find a process Pk in rest such that
            claim [k,*] – alloc [k,*] <= currentavail;>
        if (found) {                          /* simulate execution of Pk */
            currentavail = currentavail + alloc [k,*];
            rest = rest - {Pk};
        }
        else possible = false;
    }
    return (rest == null);
}
```

(c) test for safety algorithm (banker's algorithm)

$$\exists i \, \forall j \colon C_{ij} - A_{ij} \leq V_j \;\; \Rightarrow \forall j \colon V_j = V_j + A_{ij}$$

4 DEADLOCK DETECTION

Nu zullen *resource requests* altijd toegelaten worden waardoor een *deadlock detection algorithm* de *circular wait* conditie zal controleren. Als het algoritme frequent wordt uitgevoerd zal er veel processortijd naar deze uitvoering gaan, maar als de niet frequent wordt uitgevoerd is de kans dat de deadlock pas laat wordt gedetecteerd.

4.1 DEADLOCK DETECTION ALGORITHM

Vectoren

R	*Resources*	totaal aantal *resources* van het system
V	*Available*	Totaal aantal *resources* die nog niet zijn gealloceerd

Matrices

Q	*Request*	Q_{ij} gevraagde resource j voor proces i
A	*Allocation*	A_{ij} gealloceerde resource j door proces i

Het algoritme baseert zich op de processen dat niet behoren tot de deadlock set. Initieel zijn alle processen niet gemarkeerd.

```
Mark each process in matrix A with only zeros
(that has a row in the Allocation Matrix)

/* temp vector W = Available vector*/
Initialize vector(W) = vector(V)

/*      zoek proces i die ongemarkeerd is
        en waarvoor de i^de rij van Q < of = aan W
        Als geen gevonden, stop algoritme
*/
while [exists(process[i]:Q_ik ≤ W_k, 1≤k≤m)]{
        //gevonden -> markeer proces i
        //en voeg de correspondeerde rij van de allocatiematrix in W
        W_k = W_k+A_ik, 1≤k≤m
        mark proces[i]
}

//Unmarked processes are in deadlock
```

4.2 RECOVERY

1. Stoppen van alle deadlocked processen
2. Terugrollen naar vroegere staat, dit door het back-uppen naar een vorig checkpoint en opnieuw opstarten.
3. Stop proces per proces tot er geen deadlock meer is.
4. *Preempt resources* tot er geen deadlock meer is.

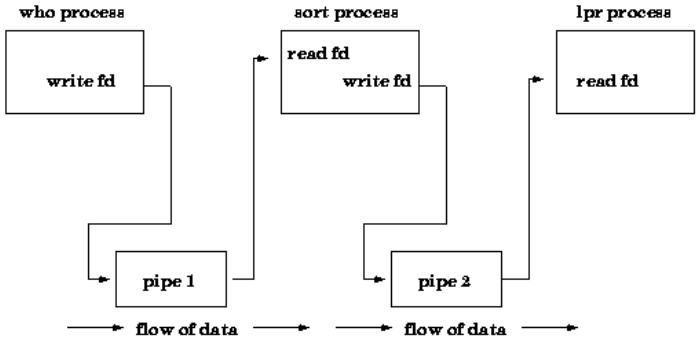
De selectiecriteria tussen 2 processen kunnen zijn: die met het minst gebruikte processortijd, die met de minst geproduceerde output, deze met de nog langste uitvoertijd of die met de laagste prioriteit.

5 UNIX CONCURRENCY MECHANISMS

UNIX voorziet een variëteit aan mechanismen voor *interprocessor communication* en synchronisatie.

5.1 PIPES

Een *pipe* is een circulaire buffer die toelaat om te kunnen communiceren tussen 2 processen volgens het *producer-consumer* model.
Dit is een FIFO *queue* (mutex) met een vaste lengte in bytes (tijdens de creatie vastgelegd).
De schrijver wordt *geblocked* als er niet genoeg ruimte is, de lezer wordt ge*blocked* als hij teveel wilt lezen (hoe wil meer lezen dan er momenteel in de *pipe* zitten).
Voorbeeld: who | sort | lpr



5.2 MESSAGES

Elke proces zal een *queue* krijgen die een functie heeft zoals een mailbox. De zender zal geblokkeerd geraken als de mailbox vol zit en de ontvanger als hij probeert te lezen uit een lege mailbox.

5.3 SHARED MEMORY

Dit is een blok van virtueel geheugen die wordt gedeeld door verschillende processen. Dit is ook de snelste vorm van *interprocess communication*. De *mutual exclusion* moet door de processen zelf worden voorzien.

5.4 SEMAPHORES

Dit is een veralgemening van de `semWait/semSignal`. Waarbij het mogelijk is om de *semaphore* met meer dan 1 te verhogen of te verlagen. Deze kunnen ook gemaakt/geïnitialiseerd worden in sets.

5.5 SIGNALS

Een signaal is een software mechanisme dat een proces informeert over een gebeurtenis van een asynchroon event. Dit gebrut door een veld aan te passen in de *process table*. *Signals* lijken dus sterk op interrupts enkel zal er geen prioriteiten worden gegeven bij *signals*. Een *signal* is verwerkt wanneer een proces wakker wordt om verder uit te voeren of wanneer het proces zich klaarmaakt om terug te komen van een *system call*.

6 LINUX KERNEL CONCURRENCY MECHANISMS

Linux bevat alle *concureency mechanisms* van UNIX, maar linux heeft ook een grote set aan *concurrency mechanisms* speciaal bedoelt om te gebruiken wanneer een thread aan het uitvoeren is in *kernel mode*.

6.1 ATOMIC OPERATIONS

Het is mogelijk om atomische operaties uit te voeren op een variabele.

Op een uniprocessor systeem zal een thread niet worden onderbroken vanaf hij begint met de operatie tot het klaar is. Op een multiprocessor systeem zal de variabele *gelocked* zijn zodat andere processen er geen toegang tot hebben, tot de operatie klaar is, hierbij zal de variabele gemeenschappelijk in het RAM zitten.

6.2 SPINLOCKS

Spinlocks worden gebruikt om kritische secties te beschermen. Een integer wordt gecheckt als de variabele op 1 staat heeft een thread al de lock in bezit en zal je moeten *spinnen*, busy waiting.

De implementatie van de spinlock is verschillend per soor processor systeem:

- Uniprocessor & *kernel preemption* is *disabled*
  De spinlocks worden verwijdert *at compile time.*
- Uniprocessor & *kernel preemption* is *enabled*
  De spinlocks worden verwijdert *at compile time*, en worden vervangen door *interrupt enabling/disabling*, wat dus neerkomt op de kritische sectie uit te voeren als een atomische operatie.
- Multiprocessor
  Hier zal de lock blijven staan in de code.

Er bestaat ook zoiets als de reader-writer spinlock implementatie waarbij meerdere threads een datastructuur kunnen lezen, maar er mag maar 1 thread de datastructuur updaten. Deze implementatie heeft een lezers-prioriteit.

6.3 SEMAPHOREN

Binary, counting en reader-writer semaphoren worden in Linux ondersteund.

Bij de eerst twee wordt er gewerkt met `up` (= `semSignal`) en `down` (= `semWait`), hierbij bestaan er meerdere down functies. Buiten de gewone funcies bestaat ook nog de `down_interruptable` waarbij de thread opgewekt kan worden door een *kernel* signaal en de `down_trylock` waarbij de thread de *semaphore* neemt als hij vrij is anders krijgt hij een niet-nul waarde en gaat door zonder geblokkeerd te worden.

Bij een *reader-writer semaphore* zullen er meerder lezers (zonder schrijver) toegang kunnen krijgen, maar er mag maar 1 schrijver toegang hebben (zonder *concurrent* lezers).

H13 – EMBEDDED OPERATING SYSTEMS

1 EMBEDDED SYSTEMS

Een embedded system is een combinatie van computer hardware en software ontworpen om een bepaalde functie uit te voeren. In vele gevallen zal een embedded systeem een gedeelte zijn van een groter systeem of product, sterk gekoppeld aan zijn omgeving.

2 CHARACTERISTICS OF EMBEDDED OPERATING SYSTEMS

Er bestaan enkele unieke karakteristieken en ontwerpen voor *embedded systems*

- Real-time operation
- Reactive operation
  Reactie op externe gebeurtenissen
- Configurability
  Het moet mogelijk zijn om delen van het OS op *embedded* systemen te plaatsen.
- I/O device flexibility
  Er moet ondersteuning zijn voor vele I/O devices.
- Low protection mechanisms
  De gebruikte software werd eerst goed getest voor ze het gebruiken. De vaste code die ze dus uitvoeren is dus betrouwbaar waardoor er geen privilege instructies nodig zijn. *Memory protection* moet ook niet hoog zijn, zodat er weinig *overhead* is.

Hierbij zijn er 2 mogelijkheden we kunnen een bestaand commercieel OS aanpassen of we kunnen een *purpose-built embedded OS* gebruiken.

Een voorbeeld van het tweede geval is TinyOS. We bespreken eerst het verschil tussen Linux en *embedded* Linux.

3 EMBEDDED LINUX

De kernel grootte zal bij een desktop Linux groter zijn door de grote variëteit aan ondersteunende apparaten en configuraties, dit in tegenstelling bij een *embedded* Linux waarbij *customization* mogelijk is.

Bij een desktop/server Linux distributie wordt de software op één platform gecompileerd en wordt ook gebruikt om datzelfde platform. Dit in tegenstelling tot *embedded systems* waarbij de software wordt gecompileerd op een platform, waarbij de uitvoering zal gebeuren op een ander platform, *cross-compiling*.

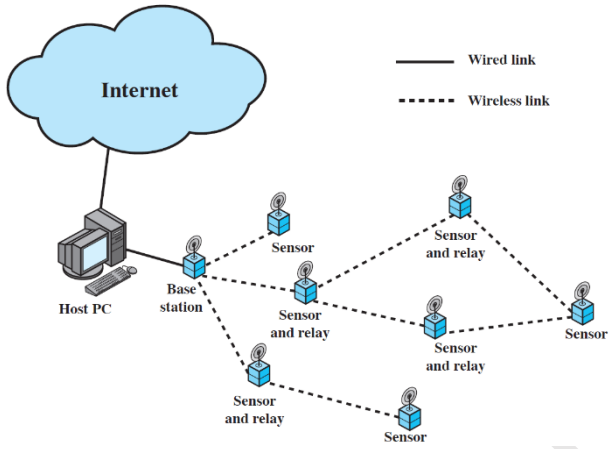
Een *file system* wordt opgeslagen in *persisent memory* (disk opslag) bij een desktop distributie, maar bij *embedded Linux* zal dit opgeslagen worden in *flash memory*. De bestanden moeten dan ook zo compact mogelijk zijn, *compressed file system*.

4 TINYOS

eCos en gelijkaardige systemen zijn beter voor kleine *embedded systems* omdat deze sterke eisen hebben op geheugen, *processing time*, *real-time response, power consumption*,... TinyOS is nog compacter omdat deze maar 400 bytes aan geheugen nodig heeft voor zijn code en data.

4.1 WIRELESS SENSOR NETWORKS

TinyOS wordt vaak gebruikt in *wireless* sensors. De applicatie/systeem software moet compact genoeg zijn voor *sensing*, communicatie en berekeningen mogelijkheden in zo een klein mogelijke architectuur.



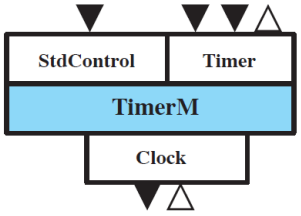
Individuele sensors ontvangen data en geven die door aan het basis station. Er is dus *routing functionality* nodig. Deze sensors moeten zich zelf ook in een ad hoc netwerk kunnen opstellen, zonder enige configuratie.

4.2 TINYOS GOALS

- Allow High concurrency
  Meerdere *flows* aan data: *Processing* data en *transmitting/forwarding* resultaten.
  Externe controle van *remote sensors* en/of base stations.
- Operate with limited resources
  Kbytes voor geheugen, RAM heft een geheugen van 100 byte (grote orde)
- Adapt to HW evolution
- Support a wide range of applications
- Support a diverse set of platforms
- Be robust
  Sensors moeten voor een lange tijd zonder overzicht gelaten worden.

4.3 TINYOS COMPONENTS

TinyOS systeem bestaat uit een set van kleine modules (componenten) die elke een simpele taak of een set van taken uitvoert. De componenten voor de basis functies (single-hop networking, ad-hoc routing, power management,..) zijn open-source. De componenten zijn gelaagd opgebouwd.



Waarbij componenten door een *shaded boxes* en interface door *white boxes* worden voorgesteld.

Elk component kan maar naar maximum één lower-level component linken, waarbij hij commands stuurt naar zijn low-level component en daarvan *event signals* ontvangt. Hij kan ook maar maximaal met 1 upper-level component linken, waarvan hij commands ontvangt en signalen naar terug verstuurd.

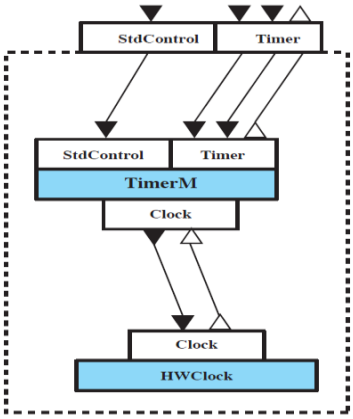
Hierdoor krijgen we onderaan de Hardware componenten en bovenaan de applicatie componenten.

Elke taak die een component implementeert zal gelijkaardig zijn aan een thread in een gewoon OS (met zeker limitaties). De taak loopt tot hij klaar is (*atomic*), hij kan dus niet worden ge-*preempted* door een andere taak in hetzelfde component. Er is ook geen *time-slicing*, *blocking* of *spin waiting*. Maar *Preemption* door een event is wel nog altijd mogelijk!

De commands, naar een lower-level component, zijn *non-blocking requests*. Commands zijn taken die gepland zijn voor latere executie.

Een event kan direct of indirect gelinkt worden met hardware events. De lowest-level componenten *interface* direct met de hardware *interrupts*. Deze kan de *interrupt* zelf handelen of doorsturen in de hiërarchie.
Een command kan ook een event teweegbrengen, het *signal event* is dan het gevolg van een bepaalde taak die werd opgeroepen door een command.

De componenten worden verbonden, *wiring*, door hun interfaces bij configuratie.




```
configuration TimerC {
   provides {
      interface StdControl;
      interface Timer;
   }
}

implementation {
   components TimerM, HWClock;
   StdControl = TimerM.StdControl;
   Timer = TimerM.Timer;
   TimerM.Clk -> HWClock.Clock;
}
```

4.4 TINYOS SCHEDULER

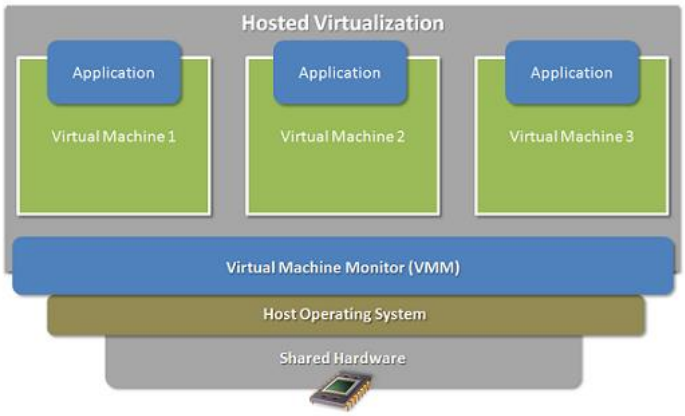
Bij TinyOS systemen wordt er voornamelijk enkel gewerkt met uniprocessoren. De default scheduler is een simpele FIFO *queue*. Als er geen taken in de *queue* zitten zal de processer in SLEEP modus komen, waarbij de hardware nog altijd *events* kan *triggeren* in SLEEP modus.

- TinyOS 1.x
  shared taks queue voor alle taken
  Elke component kan meerdere taken '*submitten*'
  Nadeel: taken mislukken als de *queue* vol zit
- TinyOS 2.x
  1 slot per taak in *task queue*
  Een taak kan maar eenmalig gepost worden, maar kan wel, door een interne variabele, zichzelf reposten (als men meerdere keren deze taak laten uitvoeren)

H14 – VIRTUAL MACHINES

Traditioneel draaide men 1 OS op 1 pc/server. Gedeeltes van de code moesten dus herschreven worden om op andere platforms te draaien. Een oplossing voor dit probleem is meerdere OS'en te draaien op dezelfde hardware, dit noemt men *virtualization*. Het *hos OS* kan meerdere virtuele machines draaien, zowel meerdere malen dezelfde PS ofwel verschillende soorten van OS'en.

Een ander probleem wast dat er teveel servers waren die niet optimaal werden gebruikt en die grote kosten met zich mee brachten (koeling, *power*, *network switches*,…).



De *hypervisor* of de *virtual machine monitor* (VMM) zit tussen de hardware en de VMs. De VMM zorgt ervoor dat meerdere VMS samen op 1 fysische server host kunnen draaien en dat de VMS de host's *resources* kunnen delen. Het aantal VMs dat op een *single host* staan noemt men het *consolidation ratio*. Met als doel om de hardware optimaal te gebruiken.

Nog voordelen van het gebruik van VMs zijn is het migreren van een server van de ene machine naar een andere zonder down-time, big data applicaties,…

1 APPROACHES TO VIRTUALIZATION

De VMM gedraagt zich als proxy voor VMs naar hardware toe. Een VM bootst de karakteristieken van een fysische server na. Deze wordt geconfigureerd met enkel processoren, RAM, opslag *resources* en maakt connectie door netwerk poorten. Een VM ziet dus enkel de *resources* die hij gekregen heeft. De hardware wordt abstract gemaakt voor een VM door de hypervisor die de translaties en I/O overziet van een VM naar hardware en omgekeerd. Hierdoor zal de performantie wat omlaag gaan.

Een VM bestaat uit *files*. De *configuration file* bevat de attributen van de VM, de allocaties. Deze zijn de virtuele CPU's (vCPU), RAM, I/O *devices* die de VM kan bereiken en virtuele disks.
Doordat de VM bestaat ui *files* is het gemakkelijk om VMs te back-uppen, te kopiëren (enkel configuraties veranderen).

1.1 BENEFITS

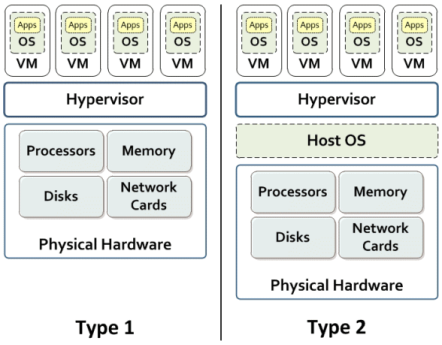
Zoals al eerder besproken zijn er voordlen als *consolidation* en snele *provisioning* door *templates*. Maar er zijn ook andere voordelen zoals *increased availability*, waarbij VMs gemakkelijk om een andere machine kunnen opgestart worden. De migratie naar een andere machine heeft nog zo zijn voordelen, namelijk dat het onderhoudt van de servers gemakkelijk wordt en wanneer een VM meer *resources* nodig heeft op zijn server, dit door het overzetten naar een andere server zonder *downtime*.

1.2 TYPE VIRTUAL MACHINE MONITORS

Een Type-1 hypervisor is geladen met een dunne software laag op de fysische server. Deze types worden voornamelijk gebruikt in *clustered servers*.

Een Type-2 hypervisor zal gebruik maken van een OS waarbij de hypervisor als applicatie wordt gedraaid.

Type-1 heeft betere *performance* en een grotere *consolidation ratio*. Het is voor *developers* wel beter om gebruik te maken van Type-2.



1.3 TACKLING PERFORMANCE BOTTLENECK

Paravirtualization is een *software-assisted virtualization* techniek die gebruik maakt van APIs om VM te linken met hun hypervisor, dit om de prestaties te verbeteren. Hierdoor zal het OS en de hypervisor meer efficiënt samenwerken met de *overhead* van de hypervisor translaties.

Sommige processoren hebben functionaliteit om hun prestaties met hypervisors te verbeteren. Toevoegen van een extra instructie set die opgeroepen kunnen worden door VMs. Deze *hardware-assisted support* hoeft geen aangepaste OS te hebben in tegenstelling tot *paravirtualization*.

2 PROCESSOR ISSUES

De eerste strategie om processor *resources* te beheren is de chip te emuleren als software. Hierdoor is het gemakkelijk te transporteren vermits het niet platform *dependent* is. Het nadeel is echter dat de ge-emuleerde processor *resource intensive* is en dus traag is.

De tweede strategie geeft segmenten van processor tijd (op pCPU[12]s) van de *virtualization host* aan de *virtual processors* van de VMs die draaien op de fysische server.
De hypervisor onderschept de *requets* voor de processor, *schedules* tijd op de processor, zend dan de *request* naar de processor en het resultaat wordt dan verzonden naar de VM.

Er zijn ook verschillende strategieën om te bepalen hoeveel processoren er per VM moeten gebruikt worden.

Bij de eerste strategie maakt men gebruik van een tool dat kijkt wat de *resource usage* is. Een tweede strategie is zeker niet te veel vCPU te alloceren, aantal is n, vermits we dan moeten wachten tot n pCPUs simultaan vrij zijn. Voor vele applicaties is het voldoen om 1 vCPU te gebruiken, dit is dan ook de derde strategie.

3 MEMORY MANAGEMENT

Het RAM kan verdeeld worden onder de VMs en de hypervisor. De hypervisor beheert de *memory requests* door het gebruik van *translation tables*.

Oplossingen om geheugen verspilling tegen te gaan:

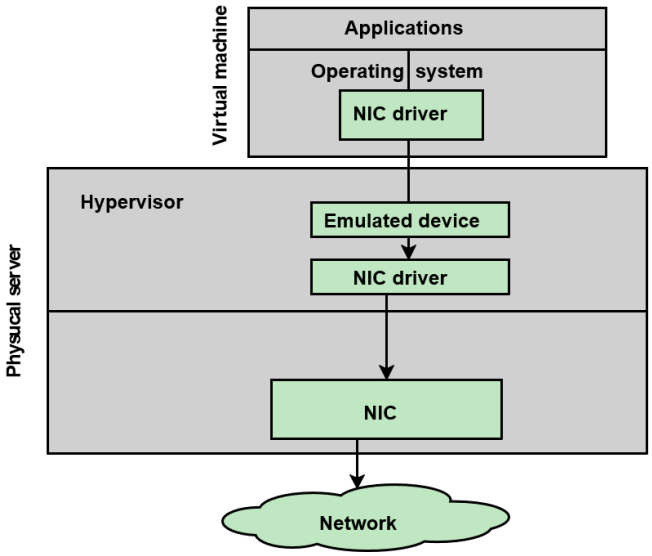
- Right-sizing VM
  Vuistregel: 1 GB per applicatie en 1 GB hypervisor
- Page sharing tussen VMs (~ de-*duplication*)
  Als VMs dezelfde OS of applicatie draaien zullen sommige geheugenblokken hetzelfde zijn, de hypervisor detecteert dit en geeft een link naar de *shared page* in de VM *translation table*.
- Memory overcommit door Ballooning
  Hierbij alloceren we meer geheugen dan er eigenlijk is. Door ballooning kan men virtueel *'inflate'*en' hierdoor gaat het gast OS zijn *pages flush*'en naar een *disk*. Als de *pages* ge-*cleared* zijn zal de balloon *driver deflate*'en en zal de hypervisor het fysische geheugen toewijzen aan andere VMs.

[12] *Physical CPU*

4 I/O MANAGEMENT

De bandbreedte bottlenecks kunnen het verkeer zijn naar het secundaire geheugen of naar het netwerk.

Hoe wordt nu het I/O beheert in de virtuele omgeving?
Het OS van het VM *calls* het device driver van het ge-emuleert apparaat.
De hypervisor controleert/*monitors* de *requests* tussen de VMs driver en het echte apparaat driver, en terug.



De voordelen hiervan zijn is dat het HW *independent* is, waardoor het gemakkelijk te migreren is naar andere HW. De porties naar individuele VMs worden door de hypervisor gecontroleerd en gegeven hierdoor wordt QoS gegarandeerd. Alsook de verbetering van *security* en *availibility*.

Maar doordat de hypervisor al het verkeer beheert zal er veel processor overhead gecreëerd worden. De oplossingen voor deze problemen zijn:
het gebruik van muticores, *sophisticated supervisors*, HW verandering voor *virtualization support* en *bypass the hypervisor's I/O stack* voor betere prestaties maar voor mindere flexibiliteit.

5 VMWARE ESXI

Bekijk de PowerPoint presentaties. Snel over geweest tijdens de les.

6 MICROSOFT HYPER-V AND XEN VARIANTS

Bekijk de PowerPoint presentaties. Snel over geweest tijdens de les.

7 JAVA JVM

Bekijk de PowerPoint presentaties. Snel over geweest tijdens de les.

8 LINUX VSERVER VIRTUAL MACHINE ARCHITECTURE

Bekijk de PowerPoint presentaties. Snel over geweest tijdens de les.