Systeemarchitectuur

Piet van Oostrum

herziene versie

november 2005

Universiteit Utrecht

Departement Informatica

Padualaan 14 3584 CD Utrecht
Corr. adres: Postbus 80.089
3508 TB Utrecht
Telefoon 030-2531454
Fax 030-2513791

Inhoudsopgave

Voorwoord

Dit is het collegedictaat voor een deel van het vak Netwerken. In dit deel kijken we naar de manier waarop programma's (stukken software) binnen een computersysteem met elkaar samenwerken en ook een beetje naar de manier waarop de software met de hardware samenwerkt. In het bijzonder zullen we aandacht schenken aan de opbouw van een Operating System (we gebruiken vaak de afkorting O.S.) en de manier waarop programma's samenwerken met het O.S. en met elkaar. Hiervoor is het ook nodig te kijken naar de opbouw van een computer. We doen dit echter vanuit een meer abstract standpunt; we kijken bijvoorbeeld niet naar de details van de elektronica waaruit een computer is opgebouwd.

De stof is niet toegespitst op een specifiek operating system, maar we kijken vooral naar algemene principes. Om niet te theoretisch te blijven gebruiken we voorbeelden uit de O.S's Unix en MS Windows. In diverse opzichten worden in deze O.S's dezelfde principes gebruikt vooral als we kijken naar Windows 95/98 of Windows XP. Andere dingen zijn echter juist weer heel verschillend en dat geeft ons de mogelijkheid om te laten zien hoe je dezelfde problemen op verschillende manieren kan aanpakken.

MS Windows 95/98, Windows NT, Windows 2000 en Windows XP hebben veel dingen gemeenschappelijk. In feite zijn de verschillen voor dit college nauwelijks interessant en we zullen ze dan ook meestal onder één noemer behandelen. De gemeenschappelijke naam die we dan gebruiken is "Win32". Win32 is eigenlijk de *abstracte* specificatie van de diensten die deze operating systems leveren. Ook de opvolgers die de komende jaren uitgebracht zullen worden vallen hier waarschijnlijk nog onder. Natuurlijk zijn er verschillen, maar die zitten meer in de details.

Als we het af en toe over oudere versies van MS Windows hebben (bijvoorbeeld Windows 3.1 of 3.11) dan zullen we de term Win16 gebruiken. Het feit dat de een 16-bits software is en de ander 32-bits (zie voor deze termen de volgende hoofdstukken), is overigens niet eens zo relevant. Het gaat meer over de ontwerpprincipes die er achter zitten. Als iets voor zowel Win32 als Win16 geldt gebruiken we de term "MS Windows".

Onder de naam "Unix" vallen verschillende O.S's van diverse fabrikanten, bijvoorbeeld *Solaris* (van Sun), *HP/UX* (van HP), *AIX* (van IBM), het public domain systeem Linux en zelfs *Mac OS X* (van Apple). Hoewel deze Unix varianten onderling ook verschillen vertonen, is de globale opzet in grote lijnen hetzelfde, vandaar dat we de generieke naam "Unix" gebruiken, en af en toe wat verschillen zullen noemen. Om een beetje eenheid te brengen is een deel van de Unix functies onder de naam "Posix" gestandaardiseerd. De meeste Unix systemen houden zich aan deze Posix standaard, en voegen dan hun eigen specifieke functies toe. Ook andere O.S's geven soms de Posix functionaliteit bijvoorbeeld bij Windows NT is het mogelijk een Posix subsysteem te krijgen. Op deze manier is het gemakkelijker om Unix programma's over te zetten naar zo'n systeem.

Overigens begon Unix ook op een 16-bits computer, en is later op 32-bits overgestapt, terwijl de algemene ideeën ervan hetzelfde bleven. Modernere systemen zijn trouwens al 64-bits, maar daar merk je als gebruiker niet zoveel van.

De relatie met andere vakken

In het tweede/derde jaar van de bachelor-opleiding informatica is er een vak “Gedistribueerd Programmeren” waarin vooral de theorie van synchronisatie wordt behandeld, bijvoorbeeld met behulp van semaforen en monitors, de problemen die optreden bij parallellisme en gemeenschappelijke toegang tot variabelen. We zullen in dit vak daar enigszins aandacht aan besteden, voornamelijk vanuit de praktische kant.

De aspecten van de opbouw van computer-netwerken en protocollen worden behandeld in de tweede helft van dit college. Hiervoor wordt een apart boek gebruikt.

In de studierichting Informatica zijn er verschillende vakken die aansluiten op Netwerken, met name Gedistribueerd Programmeren en Internet Programmeren in de bachelorfase en Gedistribueerde Objectsystemen in de masterfase.

In deze editie (november 2005) zijn een aantal delen tekstueel herzien. Ook is de hoofdstukindeling veranderd: het deel over assemblerprogrammeren is een eigen hoofdstuk geworden. Bovendien zijn van de belangrijkste begrippen definities in kaders toegevoegd zodat deze snel gevonden kunnen worden.

Inleiding

Veel computers staan tegenwoordig niet meer alleen te werken maar zijn met andere verbonden in een netwerk. De onderlinge verbondenheid tussen deze computers kan heel sterk zijn, of vrij losjes.

- We spreken van een gewoon computer-netwerk wanneer elke computer op zichzelf ook goed zou kunnen werken. Wanneer in zo'n systeem de netwerkverbindingen tijdelijk zouden uitvallen, is de computer in ieder geval nog te gebruiken.
- We spreken over een gedistribueerd O.S. wanneer het in feite zo is dat een aantal computers in het netwerk zo sterk met elkaar verbonden zijn dat de functies van het O.S. verdeeld zijn over deze computers. Als een gebruiker dan op een van die computers zit te werken, dan is het in feite het netwerk dat zijn of haar taken uitvoert. Dit heeft dan tot gevolg dat één computer alleen niet meer in staat is om het werk goed uit te voeren.
- Een tussenvorm vinden we in een netwerk waar weliswaar elke computer op zich zelfstandig kan blijven werken, maar waarbij belangrijke delen die voor het normale gebruik nodig zijn, op andere computers uitgevoerd worden. Zo kan bijvoorbeeld de opslag van files voor de gebruikers uitbesteed zijn aan aparte *file servers*, terwijl het besturen van de printers op een *print server gebeurt*. De computers waarop de gebruikers zelf werken heten dan *werkstations*.

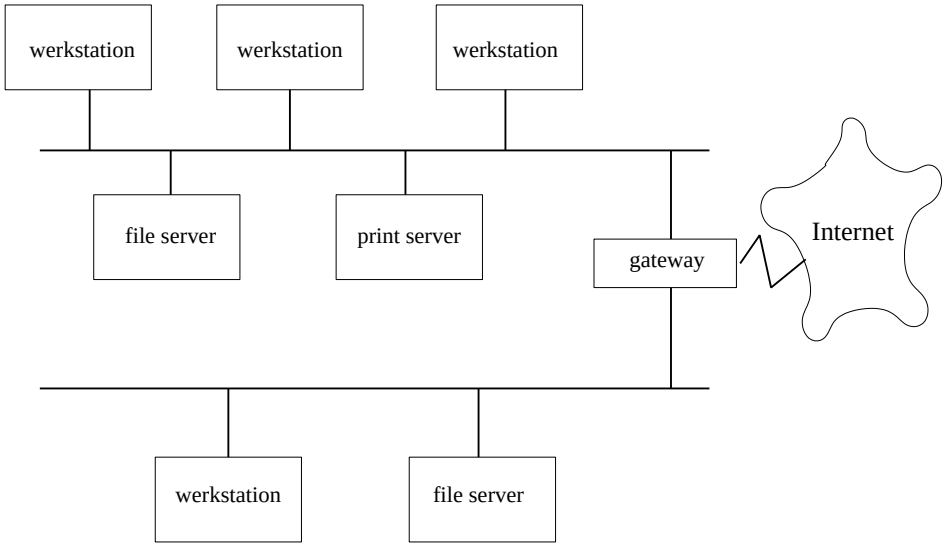
In figuur 1 zien we een voorbeeld, waarbij zelfs meerdere netwerken betrokken zijn die door een *gateway* met elkaar verbonden zijn.

In een dergelijk computersysteem zijn allerlei componenten actief, zowel software als hardware, en deze componenten communiceren voortdurend met elkaar. Voorbeelden hiervan zijn:

1. gebruikersprogramma's en O.S.
2. O.S en hardware (apparatuur)
3. gebruikersprogramma's onderling
4. onderdelen van het O.S. met elkaar
5. programma's binnen het netwerk op verschillende computers

De rode draad die door dit vak loopt, is telkens op dit thema terug te voeren. Belangrijke punten die we dan telkens tegenkomen zijn:

- Hoe geven we informatie over? Met andere woorden: hoe komt de informatie van onderdeel A naar onderdeel B?



Figuur 1: Netwerken met werkstations, servers en gateway

- Hoe verzorgen we de synchronisatie tussen communicerende componenten. Dat wil zeggen: hoe zorgen we dat de verschillende acties in de juiste tijdsvolgorde uitgevoerd worden?

Synchronisatie:
Het zorgen dat acties in verschillende onderdelen van een systeem in de juiste tijdsvolgorde uitgevoerd worden.

Abstractie en architectuur

Wanneer we het gebruik van een programma, in het bijzonder in een netwerkomgeving, vanuit de gebruiker bekijken, dan zien we dat hiermee een aantal, meestal ingewikkelde, operaties uitgevoerd kunnen worden. Als voorbeeld kunnen we nemen het ophalen van een document op het WWW via een muisklik in het programma Netscape (je kunt natuurlijk ook Firefox, Internet Explorer of een andere browser nemen). Wat de gebruiker doet is vrij simpel, maar de hoeveelheid werk die door de computer van de gebruiker, en waarschijnlijk nog vele andere computers op het Internet, verricht moet worden is gigantisch. Als je zou willen proberen alles wat er gebeurt in één keer te begrijpen dan duizelt je hoofd en zie je door de bomen het bos niet meer. We moeten dus iets doen om het begrijpelijk en overzichtelijk te maken. De manier waarop we dit doen is door in eerste instantie niet naar de details te kijken maar het systeem in eerste instantie alleen in grote lijnen te bekijken. We zeggen dan dat we op een “hoog niveau” ernaar kijken. Later kunnen we dan onderdelen nemen en die meer gedetailleerd gaan bekijken. Ook daar kunnen we hetzelfde principe toepassen: eerst in grote lijnen ernaar kijken en later de details bestuderen. Uiteindelijk komen we dan tot de elementaire acties die door de processor (CPU) in de computers uitgevoerd worden en die de hardware van de computer aansturen. De onderdelen waaruit deze operaties opgebouwd zijn noemen we “lagere niveau” operaties. Op deze manier hebben we ons systeem in lagen opgebouwd.

Het weglaten van details bij het bestuderen van een systeem noemen we *abstractie*. Elke laag geeft ons een abstractieniveau, waarover we kunnen praten zonder ons over de details van hoe die laag opgebouwd is druk te maken. Dit is de enige manier om het overzicht te bewaren, en de enige manier om het systeem zo te structureren dat we het kunnen blijven begrijpen, het kunnen verbeteren en zelfs om het in eerste instantie te kunnen ontwerpen en bouwen. Als we dit niet doen dan zien we echt door de bomen het bos niet meer.

Abstractie:
Abstractie is het weglaten van alle niet essentiële informatie of aspecten om meer fundamentele structuren zichtbaar te maken (`http://nl.wikipedia.org`).

Je kunt dit vergelijken met de bouw van een huis. Voor de architect die het huis ontwerpt en voor de koper van het huis bestaat het huis uit muren, vloeren, dak, deuren, ramen en dergelijke. Maar een muur is weer opgebouwd uit stenen. Voor de metselaar is elke steen belangrijk, voor de architect niet. En de stenen zijn opgebouwd uit moleculen, maar dat is ook voor de metselaar niet interessant. Voor een chemicus die de stenen ontwerpt echter weer wel.

In de informatica spreken we daarom ook over architectuur. Dat is een abstracte specificatie van wat je van een systeem mag verwachten. Hoe het systeem dan in werkelijkheid opgebouwd is noemen we de implementatie. Een systeem kan hierbij een computer zijn, een onderdeel van een computer, zoals een Pentium processor, een stuk software of de combinatie van hardware en software. De 80386, 80486 en Pentium chips zijn verschillende implementaties van dezelfde architectuur. Het eerder genoemde Win32 en Posix zijn ook architecturen, en Windows XP is een implementatie van de Win32 architectuur.

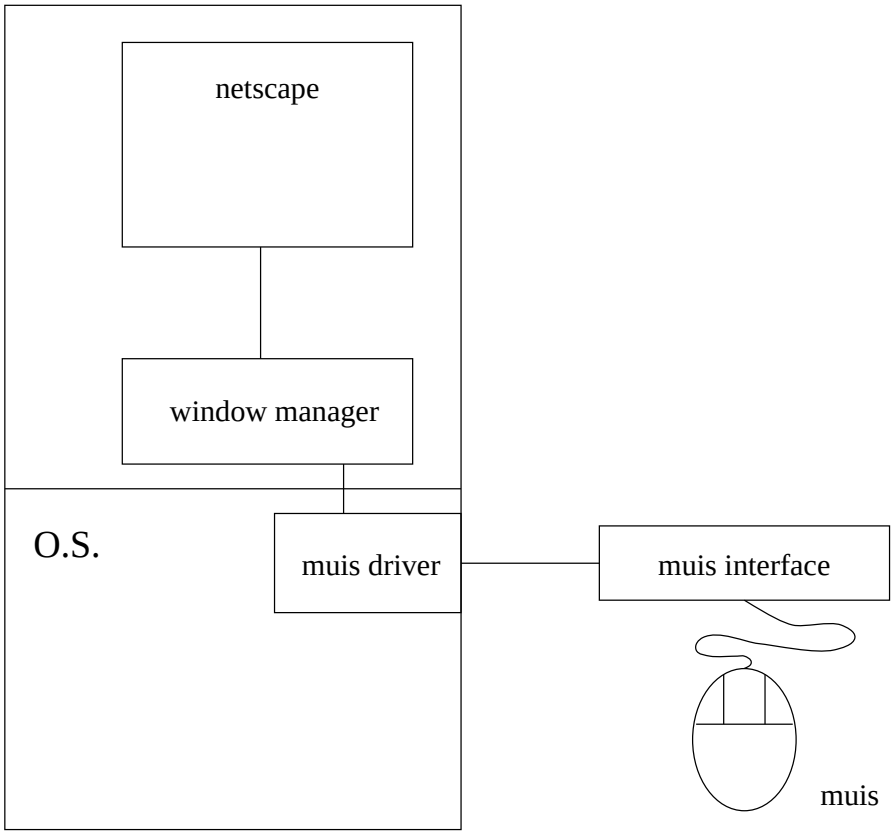
Architectuur:
Een abstracte specificatie van wat een systeem aan functies of mogelijkheden levert.

Implementatie:
De manier waarop een systeem met een bepaalde architectuur gerealiseerd wordt.

Laten we eerstgenoemde voorbeeld eens gaan opdelen. Dit voorbeeld bevat in feite praktisch alle elementen die we in dit vak zullen behandelen.

1. De muisklik van de gebruiker zal door de hardware gedetecteerd worden. Ergens in het O.S. is er een stukje software dat verantwoordelijk is voor het in de gaten houden van de muis. Dit wordt de “driver” genoemd. De driver ontvangt signalen van de muis en kan hieruit afleiden hoe de muis bewogen wordt en wanneer er op een knop gedrukt wordt. Wat een beweging betekent en wat er met de muisklikken moet gebeuren zal in het algemeen afhangen van de programma’s die actief zijn, en van de inhoud van het scherm. In de meeste O.S’s is er een speciaal stuk software verantwoordelijk voor alles wat met het beeldscherm te maken heeft, meestal windowmanager of iets soortgelijks genoemd. De muis-driver geeft een seintje aan de windowmanager dat er een muisklik is gedaan.

2. De windowmanager zoekt uit welk programma verantwoordelijk is voor het deel van het scherm waarin de muis-pijl wijst. De windowmanager stuurt een bericht met de relevante gegevens (zoals welke knop ingedrukt is en de coördinaten van de muispijl) naar het betreffende programma



Figuur 2: Componenten bij het gebruik van Netscape

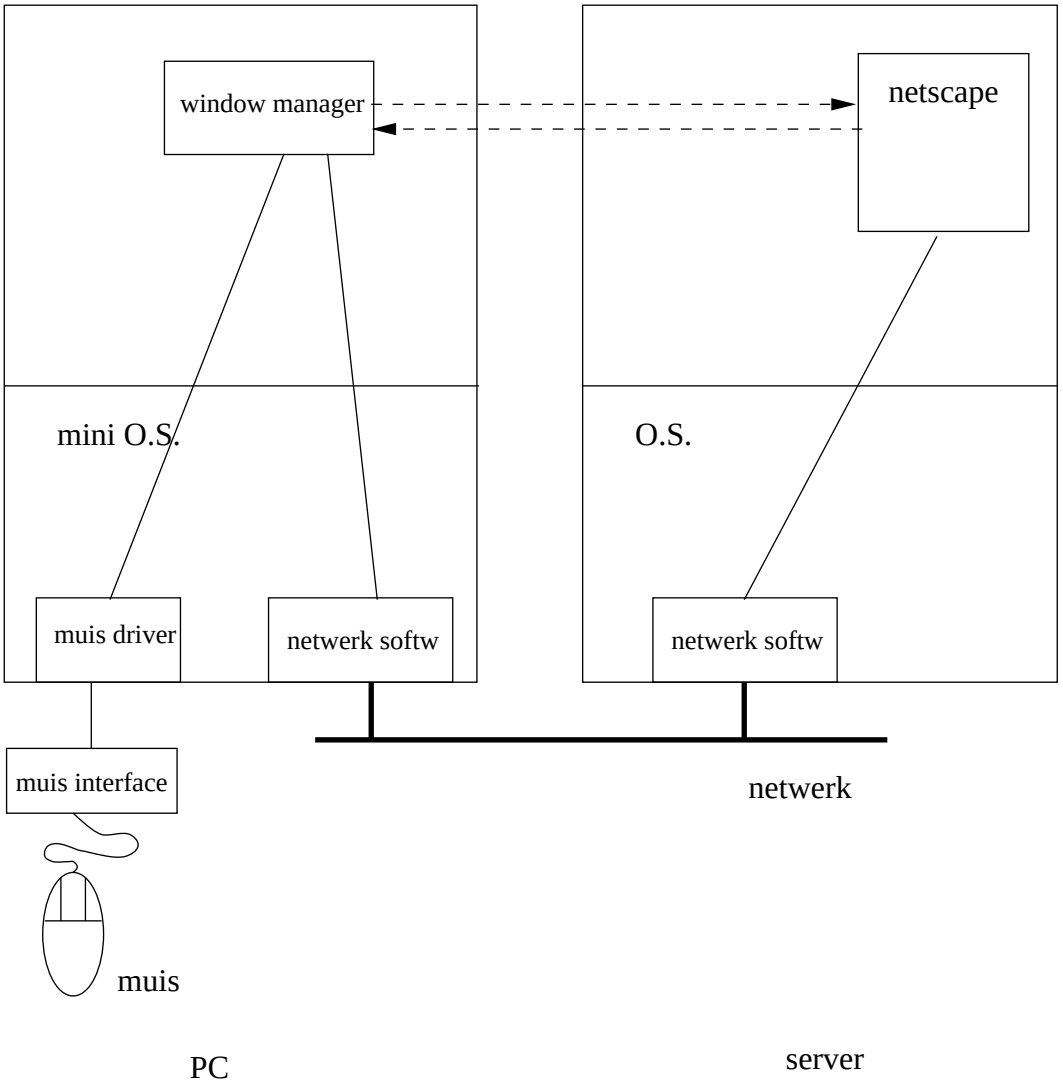
(in dit geval dus Netscape). Het kan zelfs gebeuren dat de windowmanager op een andere computer zit dan het verantwoordelijke programma; we hebben dan een gedistribueerd systeem. In dat geval zal er dus via het netwerk gecommuniceerd worden – zie hiervoor punt 6.

3. Het programma Netscape vindt uit waar de muisklik heeft plaatsgevonden. Hiervoor wordt meestal een voorgeprogrammeerde functie gebruikt, die in een z.g. bibliotheek is opgeslagen: het heeft weinig zin om dit soort acties iedere keer weer opnieuw te moeten programmeren.

Bibliotheek:
Een stuk programma dat je niet zelf hoeft te schrijven maar die bij je computer of je software meegeleverd worden. Bijvoorbeeld de verschillende standaardklassen van Java.

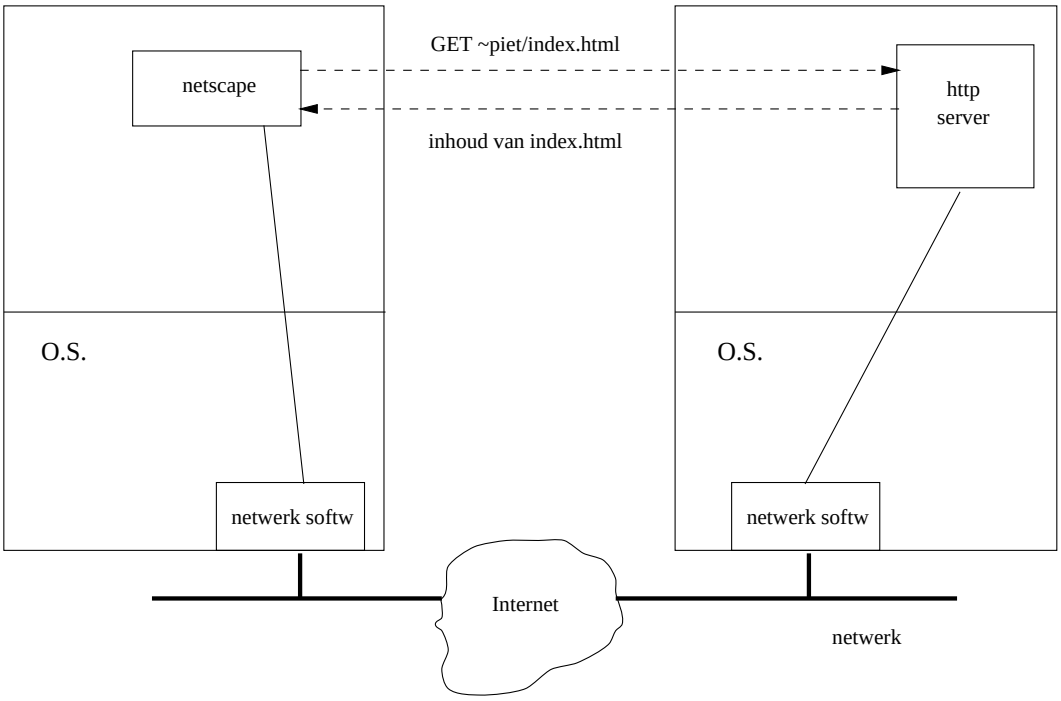
4. Netscape besluit om het document op te halen, waarvan de URL “onder de muis” zat. Voor het gemak doen we even of hiervoor ook een standaard bibliotheekfunctie aanwezig is (laten we deze “geturl” noemen).

5. De bibliotheekfunctie weet hoe documenten van het WWW opgehaald moeten worden. We gaan er voor het gemak even van uit dat het een HTML document betreft dat met het HTTP protocol opgehaald moet worden. De URL is bijvoorbeeld
“`http://www.cs.ruu.nl/people/piet/index.html`”. De functie geturl begint nu met het



Figuur 3: Componenten bij een gedistribueerd systeem

maken van een Internet verbinding met de webserver op de machine “`www.cs.ruu.nl`”. Wanneer deze verbinding tot stand gekomen is dan wordt over deze verbinding een verzoek gestuurd “GET /people/piet/index.html”. Dit verzoek wordt door de webserver ontvangen (via die Internet verbinding) en wanneer het document beschikbaar is wordt de inhoud terug gestuurd naar Netscape.



Figuur 4: Interactie tussen Netscape en webserver

6. Bij het opzetten en gebruiken van een Internet verbinding komen nogal wat dingen kijken. Zo is het voor het opzetten van een Internet verbinding nodig om de machine waarmee een verbinding gemaakt moet worden te identificeren met een IP adres (bijvoorbeeld 131.211.80.17 voor www.cs.ruu.nl). De relatie tussen de namen en de nummer wordt in het Internet bijgehouden door een collectie programma’s die “nameservers” genoemd worden. Elke nameserver kent een deel van de vertaling, bijvoorbeeld de nameserver van informatica kent de namen die eindigen op “cs.ruu.nl” en weet welke nameserver gebruikt kan worden voor namen die hij zelf niet kent. Op deze manier is een “gedistribueerde nameservice” op het Internet aanwezig, die zelf ook weer gebruik maakt van de Internet protocollen en de netwerk structuur. Dus alleen al het uitvinden van het IP adres kan een hoeveelheid werk veroorzaken die minstens zo veel is als het ophalen van het HTML document.

7. Bij het opzetten van de connectie en het versturen en ontvangen van de boodschappen zullen er verscheidene aanroepen gebeuren naar functies die het O.S hiervoor beschikbaar stelt. Misschien zullen er als gevolg hiervan berichten gestuurd worden naar andere programma’s die in dezelfde computer draaien en die een deel van het werk voor hun rekening nemen. Uiteindelijk zullen er boodschappen aangeboden worden die over een netwerk hardware verbinding (bijvoorbeeld een ethernet of een modem) verstuurd moeten worden. Deze boodschappen komen

dan bij een geschikte driver terecht, de ethernet driver of de modemdriver bijvoorbeeld.

Er kunnen in een systeem hiervoor verschillende drivers aanwezig zijn. Het interessante is nu dat een programma dat een boodschap via het netwerk wil versturen niet hoeft te weten via welke driver dit gebeurt. Dit komt omdat alle drivers op dezelfde manier gebruikt worden, hoewel ze intern heel veel kunnen verschillen. De drivers hebben dus dezelfde "interface" (manier van gebruik), net zoals verschillende automerken op dezelfde manier gebruikt kunnen worden. Je hoeft geen apart rijbewijs te hebben om met een Volvo te rijden. Hier zie je ook weer het principe van abstractie.

8. De driver stuurt de hardware aan, bijvoorbeeld door byte voor byte aan te bieden of door een gespecialiseerd stuk hardware de hele boodschap te laten oversturen. Tijdens dit oversturen kan de computer best allerlei andere activiteiten uitvoeren, zoals het bijhouden van een klokje, het bewegen van de muispijl, het printen van een ander document of het spelen van Patience. De hardware zal af en toe seintjes (interrupts) geven aan de computer om aan te geven dat er weer een volgende actie moet plaatsvinden.

9. Op zijn weg naar de uiteindelijke bestemming zal een bericht misschien door diverse andere computers heengaan, waar telkens weer soortgelijke operaties plaatsvinden als in de bron- en doelcomputer.

10. Bij de doelcomputer aangekomen gaat de boodschap weer door een driver heen, en moet het O.S. uitzoeken voor welk programma deze boodschap bestemd is. Hiervoor houdt het O.S. administratie bij van de Internet verbindingen die actief zijn. Het programma (in dit geval de webserver) krijgt een seintje dat er een boodschap is en zal deze gaan behandelen. Uiteindelijk zal er een bericht terug gestuurd worden op ongeveer dezelfde manier als het verzoek, en zal het programma Netscape het document inlezen en op het scherm laten zien. Deze operatie is ook weer een "hoog niveau" operatie die op dezelfde manier opgesplitst kan worden.

11. Sommige programma's zullen in de tijd dat ze wachten op het document niets doen, maar Netscape is efficiënter opgezet. Tijdens het wachten op het document kan het andere dingen doen, bijvoorbeeld een ander document afdrukken of erdoorheen bladeren, of een ander document ophalen. We zeggen dat Netscape "multithreaded" is. Dit maakt het werken voor de gebruiker efficiënter. We spreken ook wel over het "parallel" uitvoeren van verschillende taken. Let op dat we nu in feite al over verschillende niveaus van parallelliteit spreken: Netscape kan parallel werken met andere programma's zoals Patience, binnen Netscape kan parallelliteit optreden, en de drivers kunnen ook nog eens parallel aan dit hele gebeuren werken. Zelfs op hardware niveau kan er nog parallel gewerkt worden als het versturen en ontvangen van boodschappen van het netwerk (of gegeven van een harde schijf) door gespecialiseerd hardware parallel aan het uitvoeren van instructies gedaan wordt.

Dit was intussen heel wat. We zullen in de rest van het dictaat de verschillende lagen één voor één behandelen, waarbij we beginnen vanaf de hardware, naar een steeds groter abstractieniveau toe.

Hoofdstuk 1

Computer Architectuur

Een computer is een apparaat dat gegevens verwerkt.Het moet daarom onderdelen bevatten die

- gegevens kunnen invoeren, zoals een toetsenbord en een muis
- gegevens kunnen uitvoeren, zoals een beeldscherm en een printer
- gegevens kunnen opslaan, zoals harde schijven, floppies en CD-Roms
- gegevens kunnen manipuleren (optellen, aftrekken, vermenigvuldigen)

Om met het laatste te beginnen: het onderdeel dat de bewerkingen op de gegevens uitvoert wordt meestal een 'Centrale VerwerkingsEenheid' (CVE), in het Engels 'Central Processing Unit' (CPU) genoemd. Soms zeggen we gewoon *processor*. De CPU is het "intelligente" deel van de computer, hier worden de opdrachten uitgevoerd, de beslissingen genomen en het rekenwerk gedaan. Sommige computers hebben twee CPU's of zelfs nog meer. We spreken in zo'n geval van een *multiprocessor* systeem.

CPU/CVE/processor:
Het deel van de computer dat de bewerkingen en berekeningen uitvoert (het 'brein').

Multiprocessor systeem:
Een systeem met meer dan één CPU (processor).

Voor het opslaan van gegevens maken we onderscheid tussen het *interne geheugen* dat heel snel is en externe geheugens zoals harde schijven en CD-Roms. De termen intern/extern stammen nog uit de tijd dat harde schijven zo groot waren dat die in aparte kasten stonden. Het verschil in snelheid tussen intern en extern geheugen is enorm: modern intern geheugen heeft minder dan 10 nsec (zie kader op pagina 14) nodig om gegevens op te leveren of op te slaan. Bij een harde schijf kan dit 1000 tot zelfs 100000 keer langzamer zijn. Behalve de snelheid is er nog een belangrijk verschil: Het interne geheugen is rechtstreeks toegankelijk vanuit de CPU, dus de CPU kan hier direct mee werken (bijvoorbeeld twee getallen die in het interne geheugen staan, optellen). De gegevens van een extern geheugen moeten echter eerst naar het interne geheugen gekopieerd worden voor de CPU er iets mee kan doen. Omgekeerd kan de CPU niet direct iets op de harde schijf zetten: eerst moet het in het interne geheugen geplaatst worden.

Talstelsels

Voor het noteren van getallen gebruiken wij gewoonlijk de cijfers 0 t/m 9. De reden hiervoor is dat we 10 vingers hebben en dus wat makkelijker met 10 verschillende cijfers kunnen omgaan. Voor een computer is dit geen argument. Een ouderwetse mechanische kilometerteller gebruikt wieltjes met de 10 cijfers erop. Tandwielen maken met 10 verschillende standen is niet zo moeilijk. Maar een elektronische schakeling met 10 verschillende standen is lastig en inefficiënt. In elektronica is het veel handiger te werken met schakelingen die maar twee standen kennen (aan/uit, open/dicht, wel/geen stroom). Daarom gebruiken computers intern niet 10 cijfers om getallen op te slaan, maar slechts 2 (0 en 1). Net zoals je met 10 cijfers alle getallen kunt weergeven, kun je dat met 2 cijfers ook.
Onze manier van getallen noteren is een *positioneel talstelsel*: De waarde van een cijfer wordt bepaald door de positie. In het getal 333 heeft de eerste 3 de waarde 300, de tweede 30 en de laatste gewoon 3. Samen 333. De formule voor de waarde van een getal met cijfers $d_n d_{n-1} \ldots d_2 d_1 d_0$ is

$$d_n \cdot 10^n + d_{n-1} \cdot 10^{n-1} + \cdots + d_2 \cdot 10^2 + d_1 \cdot 10^1 + d_0 \cdot 10^0 = \sum_{i=0}^{n} d_i \cdot 10^i$$

Het getal 10 wordt de basis of het grondtal genoemd en we spreken daarom van een 10-tallig of decimaal stelsel.
Je kunt dit zelfde systeem toepassen met andere grondtallen, als je maar evenveel cijfers hebt als het grondtal. Bijvoorbeeld bij een 7-tallig stelsel heb je 7 cijfers nodig (0 t/m 6) en is de waarde van een getal:

$$\sum_{i=0}^{n} d_i \cdot 7^i$$

Algemeen bij een grondtal g:

$$\sum_{i=0}^{n} d_i \cdot g^i$$

Als het grondtal niet uit de context duidelijk is schrijven we $ddddd_g$. Met een grondtal g en n cijfers kun je g^n verschillende getallen weergeven: elk extra cijfer vermenigvuldigt het aantal mogelijkheden met g. Bij een computer die alleen de cijfers 0 en 1 gebruikt heb je dus grondtal 2 en spreken we over een 2-tallig of binair systeem. Eén zo'n cijfer wordt een *bit* genoemd, een afkorting van *binary digit*. In het geheugen worden de bits opgeslagen in groepjes van 8, die een *byte* of *octet* genoemd worden. Meestal verwerkt de CPU de informatie echter in grotere eenheden bijvoorbeeld van 32 bits, ook wel een *(geheugen-)woord* genoemd.
Hoewel dus intern in een computer het 2-tallig systeem gebruikt wordt is dit voor ons (homo sapiens) niet handig: het aantal cijfers wordt gewoon teveel. Wij gebruiken daarom decimaal en de computer zet de getallen wel om van de decimale notatie naar de binaire en omgekeerd. Dit heeft echter alleen zin als we de getallen gebruiken om te rekenen. Een computer kan echter de woorden uit het geheugen ook op andere manieren gebruiken, bijvoorbeeld voor tekst, plaatjes, geluid e.d. Als programmeur ben je soms geïnteresseerd in de bits die in zo'n woord zitten, maar is de binaire notatie te omslachtig. In zo'n situatie gebruiken we vaak de hexadecimale (16-tallige) of octale (8-tallige) notatie. De reden hiervan is dat je makkelijk van octaal naar binair kunt en omgekeerd: je pakt telkens 3 bits bij elkaar en vormt hiervan een 8-tallig cijfer. Je begint hierbij aan de rechterkant. Bij hexadecimaal op dezelfde manier: alleen neem je dan 4 bits bij elkaar. Hexadecimaal heeft nog het voordeel dat je voor een byte dan precies twee cijfers nodig hebt. Je hebt voor hexadecimaal

Talstelsels (vervolg)

natuurlijk 16 cijfers nodig. We gebruiken hiervoor de cijfers 0 t/m 9, gevolgd door de letters a t/m f (of de hoofdletters)
Hieronder zie je wat getallen in de decimale, binaire, octale en hexadecimale notatie:

decimaal	binair	octaal	hexadecimaal
2	10	2	2
3	11	3	3
4	100	4	4
8	1000	10	8
12	1100	14	c
15	1111	17	f
16	10000	20	10
64	1000000	100	40
100	1100100	144	64
218	11011010	332	da

Negatieve getallen

Tot nu toe hebben we alleen positieve getallen (en 0) gebruikt. Hoe moeten we nu negatieve getallen weergeven in bits? er zijn in het verleden verschillende systemen in gebruikt geweest, maar het systeem dat tegenwoordig gebruikt wordt heet 2-complement. Het is vergelijkbaar met wat er gebeurt als je een mechanische kilometerteller vanaf 0 terugdraait. Als we bijvoorbeeld een kilometerteller met 4 cijfers hebben en je draait hem één stap terug dan krijg je 9999. Nog een stap terug en je krijgt 9998 enzovoort. Je zou kunnen zeggen dat -1 overeenkomt met 9999 en -2 met 9998.
In een binair systeem betekent dit dat -1 voorgesteld wordt door $111\ldots11_2$, -2 door $111\ldots10_2$ enzovoort. Hiernaast een tabel met een deel van de getallen als er 5 bits gebruikt worden. Je kunt de binaire representatie van $-n$ krijgen door die van $n-1$ te nemen en dan de 0'en en 1'en om te flippen (ga dit na!). Als je n bits hebt, heb je in totaal 2^n combinaties. Daarvan is de helft (2^{n-1}) negatief. Van de andere helft is er één 0, en de rest ($2^{n-1}-1$) is positief. Er is dus 1 negatief getal meer dan er positieve getallen zijn. Het kleinste negatieve getal (-2^{n-1}) heeft niet een overeenkomstig positief getal. Bij 5 bits is er dus wel -32, maar niet $+32$: het grootste getal is $+31$!

decimaal	binair
3	00011
2	00010
1	00001
0	00000
-1	11111
-2	11110
-3	11101

Rekenen met binaire getallen

Rekenen met binaire getallen is simpel en gaat vergelijkbaar met decimale getallen. Omdat je alleen met cijfers 0 en 1 werkt is het makkelijker. De optelling en vermenigvuldiging van losse bits is simpel: Uit de operaties op losse bits kunnen we makkelijk die op getallen afleiden (bijvoorbeeld $13+9$ en 13×9 (het kleine 1-tje is 'één onthouden):

a	b	a+b	a×b
0	0	00	0
0	1	01	0
1	0	01	0
1	1	10	1

13=	1	1	0	1	
9=	1	0	0	1	+
1	0	1	1^1	0	

			1	1	0	1	
			1	0	0	1	×
			1	1	0	1	
1	1	0	1	.	.	.	
1	1	1^1	0	1	0	1	

Eenheden

In de informatica gebruiken we meestal dezelfde eenheden voor de machten van 10 als in de natuurkunde. Maar in sommige gevallen (in het bijzonder als het over intern geheugen gaat) worden afwijkende waarden gebruikt. De gebruikelijke natuurkundige (eigenlijk het internationale systeem van eenheden) afkortingen die in de informatica populair zijn vind je in onderstaande tabel:

afkorting	voluit	waarde	afwijkend
p	pico	10^{-12}	
n	nano	10^{-9}	
μ	micro	10^{-6}	
m	milli	10^{-3}	
k	kilo	10^{3}	$1024(=2^{10})$
M	mega	10^{6}	$1048576(=1024^2=2^{20})$
G	giga	10^{9}	$1073741824(=1024^3=2^{30})$
T	tera	10^{12}	$1099511627776(=1024^4=2^{40})$
P	peta	10^{15}	$1125899906842624(=1024^5=2^{50})$

Omdat de groottes van intern geheugen altijd iets met machten van twee zijn wordt hiervoor de rechterkolom gebruikt. Een intern geheugen van 1 GB (megabyte) bevat dus 1073741824 bytes. Voor harde schijven wordt vaak (maar niet altijd) de macht van 10 gebruikt. Dus een harde schijf met evenveel bytes zou dan ca. 1.07 GB zijn en dat lijkt wat meer. Vandaar dat fabrikanten van harde schijven dit graag zo doen.

Het interne geheugen wordt verdeeld in ROM (Read-Only Memory) en RAM (Random Access Memory). De laatste naam betekent dat elke geheugenplaats even snel bereikt kan worden, maar dat geldt ook voor ROM. Het eigenlijke verschil is dat ROM gegevens niet veranderbaar zijn (en dus ook niet wegraken als de computer uitgezet wordt) en RAM gegevens wel. RWM (read/write memory) zou dus een betere naam zijn, maar de geschiedenis heeft ons opgezadeld met de term RAM.

Het interne geheugen van de computer bestaat uit een groot aantal bytes (elke byte is 8 bits) en deze zijn allemaal genummerd. Het nummer van een byte wordt het *adres* genoemd. Bij de meeste huidige computers lopen de adressen vanaf 0 tot ca 4 miljard, al zullen veel computers minder hebben. Voor 4 miljard nummers heb je 32 bits nodig, daarom zijn de adressen in de meeste computers inderdaad 32 bits. Maar grotere computers hebben tegenwoordig 64-bits adressen, omdat 4 miljard bytes (4 GB of Gigabyte) niet meer genoeg is voor grote systemen.

Voor in- en uitvoer van gegevens hebben we verschillende soorten apparaten zoals toetsenbord, muis, beeldscherm, geluidskaart, printer. Ook netwerkapparatuur zoals modems worden hiertoe gerekend. Dit soort apparaten noemen we vaak *randapparatuur*.

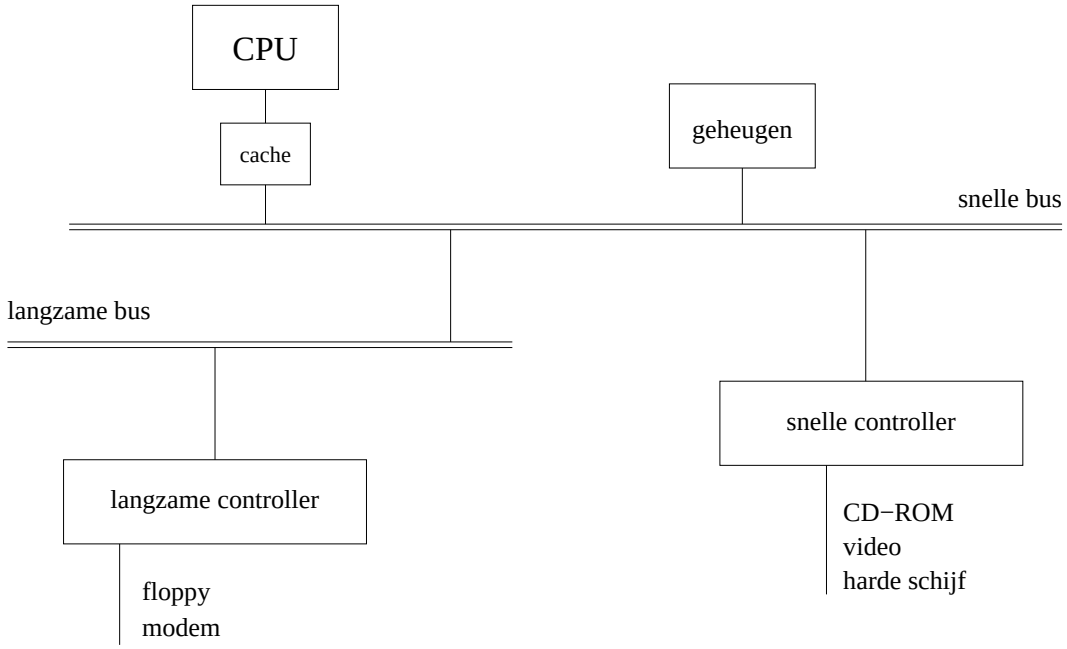
1.1 De Bus

De randapparatuur wordt bestuurd door speciale stukjes electronica, interfaces of controllers geheten. Eén controller kan soms een aantal randapparaten aan. De interfaces, het interne geheugen en de CPU zijn met elkaar verbonden door een z.g. *bus*, een aantal draden met connectoren, waarin de verschillende onderdelen geprikt kunnen worden. Er zijn draden voor de gegevens die heen en weer gestuurd

moeten worden, voor de adressen (geheugenadressen of nummers van interfaces) en besturingssignalen (bijvoorbeeld lezen of schrijven, timing). Grotere computersystemen hebben vaak meer dan één bus, bijvoorbeeld een langzame en een snelle. Bussen kunnen verschillen in de hoeveelheid data die ze per keer kunnen transporteren, snelheid en zo. Om apparatuur van verschillende fabrikanten goed met elkaar te kunnen laten samenwerken, moeten er duidelijke afspraken over de bussen zijn: dus wat voor doel elke draad heeft, hoe de connectors eruit zien, wat de eigenschappen van de signalen moeten zijn, welke volgorde van signalen gebruikt moeten worden, en welke snelheid toegestaan is. In oudere PC's wordt bijvoorbeeld vaak gebruik gemaakt van een 16 bits bus. Dit betekent dat in één operatie 16 bits aan gegevens tegelijk over de bus gestuurd kunnen worden. Als zowel de data als de adressen 16 bits groot zijn, dan kan hier weer een keus gemaakt worden: worden de adresbits en de databits tegelijk verstuurd of na elkaar. Als ze tegelijk verstuurd worden moeten er aparte draden zijn voor de adressen en de data, wat de bus ingewikkelder en dus duurder maakt. Als ze apart verstuurd worden, moet er een apart signaal mee verstuurd wat aangeeft of er een adres of data op de bus staat. De bus wordt wel goedkoper, maar ook langzamer, want alle operaties kosten nu twee keer zoveel tijd. Het is zelfs mogelijk om nog verder te gaan, namelijk door de adressen en de data op te splitsen in 2 keer 8 bits, en die twee stukjes na elkaar op te sturen. Dan hebben we weer een extra signaal nodig om aan te geven of we het eerste of het tweede stukje hebben. De bus wordt dan weer eenvoudiger en goedkoper, maar ook weer langzamer. Moderne bussen zijn meestal 32 bits voor snelle computers en apparatuur, en 16/8 bits voor langzamere.

Bus:
Een structuur (draden en electronica), die verschillende onderdelen van een computer met elkaar verbindt en waarover deze onderdelen gegevens met elkaar kunnen uitwisselen.

In figuur 1.1 zien we een voorbeeld van een computersysteem met twee bussen.



Figuur 1.1: Computer architectuur

1.2 De CPU

Ook CPU's verschillen enorm in hun eigenschappen en mogelijkheden. De meeste CPU's die tegenwoordig gebruikt worden zitten op één chip. Soms zitten er zelfs meerder CPU's op één chip. De chip bevat aansluitingen voor o.a. de bus. Het belangrijkste dat de CPU moet doen is de berekeningen van onze programma's uitvoeren. De CPU heeft hiervoor electronica waarmee de elementair bewerkingen uitgevoerd kunnen worden, zoals optellen, aftrekken, vermenigvuldigingen en logische bewerkingen.

Een programma wordt nu opgebouwd uit deze elementaire bewerkingen die we instructies noemen. De instructies staan in het geheugen, net als de gegevens waarop de bewerking uitgevoerd moet worden. Een instructie bestaat uit een opdrachtcode en een aanwijzing van waar de waarde(n) staan waarop de operatie uitgevoerd moet worden en waar het resultaat naar toe moet. De CPU moet nu de instructies één voor één ophalen, interpreteren en uitvoeren. Het instructierepertoire van een CPU kan heel simpel zijn (bijvoorbeeld bij een microprocessor in een horloge) of heel uitgebreid, zoals in PC's of werkstations. De laatste hebben bijvoorbeeld instructies om te rekenen met floating-point getallen, terwijl de eerste alleen simpele rekenkundige bewerkingen op 4- of 8-bits getallen kunnen uitvoeren. (Meestal hebben ze niet eens een instructie voor vermenigvuldigen of delen. Als dit nodig is wordt er een programmaatje gemaakt om de vermenigvuldiging te doen door meerdere malen op te tellen.)

De code van deze instructies wordt machinecode genoemd. In feite bestaat deze uit nullen en enen en het is dus niet erg handig om daarmee te programmeren. Daarom is er ook een *assemblercode* of *assemblertaal* wat een leesbare schrijfwijze is voor de instructies. De code voor optellen wordt dan bijvoorbeeld aangegeven met `ADD`, en er kunnen variabele namen gebruikt worden. Het verschil met een programmeertaal als Java is dat je elke instructie individueel moet opschrijven, en de assemblertaal is dus ook verschillend voor verschillende soorten CPU's. Je kunt er ook heel makkelijk fouten mee maken omdat bijvoorbeeld niet getest wordt of een variabele van het juiste type is.

Met machineinstructies en dus ook met assemblercode kun je in principe elke byte uit het geheugen ophalen, er iets mee doen, of er iets inzetten. Dit in tegenstelling tot Java of andere hogere programmeertalen, waar het systeem zelf bepaalt waar een variabele in het geheugen komt.

Verderop in dit dictaat beschrijven we een eenvoudige assemblertaal van een hypothetische computer.

De programmeertaal C (en C++) is een soort tussenvorm. Deze talen lijken op Java (of beter gezegd Java lijkt op C en C++), maar ze hebben wel mogelijkheden om meer rechtstreeks in het geheugen te werken. In dat opzicht staan ze dicht bij assemblercode, maar zijn ze minder afhankelijk van de specifieke CPU. C en C++ hebben ook mogelijkheden om met adressen te werken, alleen worden die dan *pointers* genoemd. Een verschil tussen een pointer en een adres is dat een pointer een type heeft, bijvoorbeeld pointer naar een `int` (aangegeven als `int *`) of een pointer naar een `float` (aangegeven als `float *`). Een pointer is dus eigenlijk een waarde die verwijst naar een geheugenplaats. In C en C++ zijn er ook operaties die met pointers werken: Als p een pointer is, dan is $*p$ de waarde waar p naar verwijst, en omgekeerd als x een variabele is dan is $\&x$ de pointer die naar x wijst (ook wel het adres van x of een referentie naar x genoemd). Verder kun je met pointers rekenen: Door 1 bij een pointer op te tellen krijg je een pointer naar de volgende plaats in het geheugen van het juiste type (de volgende `int`, `float` etc.). Omdat C en C++ zo flexibel zijn, grotendeels machine-onafhankelijk en toch een redelijke bescherming tegen programmeerfouten bieden worden ze vaak gebruikt voor het maken van Operating Systems, compilers, editors en soortgelijke basisprogramma's.

We geven nu een voorbeeld om te laten zien hoe je met pointers kunt werken: Stel we hebben variabelen x en y die een string bevatten, m.a.w. een rijtje `char`'s. We veronderstellen dat de string afgesloten

wordt met een `char` dat de waarde 0 heeft (dit is een gebruikelijke manier in de meeste Operating Systems, en in de programmeertaal C). We willen nu de string die in x staat, kopiëren naar de variabele y. We doen dit door twee pointers te nemen p en q, die we naar x resp. y laten wijzen. Door nu telkens $*p$ te nemen (het `char` waar p naar wijst) en dat te kopiëren naar $*q$ (de plaats waar q naar wijst) en beide pointers op te hogen lopen we de hele string af. We stoppen als we 0 tegenkomen, maar die moet nog wel bij q erin zetten. Het programma wordt dan (het kan overigens nog best compacter opgeschreven worden):

```
char *p = &x;
char *q = &y;
while (*p != 0)
{
    *q = *p;
    p++; q++;
}
*q = 0;
```

1.2.1 Registers

De argumenten en resultaten van instructies kunnen natuurlijk in het geheugen opgeslagen worden, maar de meeste CPU's bevatten één of meer registers. Dit zijn opslagplaatsen voor gegevens in de chip zelf. De registers zijn veel sneller toegankelijk dan het geheugen, enerzijds omdat de toegang niet via de bus gaat, maar intern in de chip, anderzijds omdat het soort electronica dat voor de registers gebruikt wordt veel snellere operaties mogelijk maakt dan wat voor het geheugen gebruikt wordt (het zou te duur zijn om dit ook voor het geheugen te gebruiken). Het aantal registers verschilt nogal, en ook de grootte van elk register. Microprocessors die in wasmachines gebruikt worden hebben meestal een paar registers van 8 of evt. 16 bits per stuk, terwijl een moderne microprocessor als de PowerPC 32 registers van elk 32 bits heeft. Bij een microprocessor die voornamelijk 8-bits registers heeft en dus vooral instructies heeft die op 8-bits data werken zeggen we dat het een 8-bits processor heeft, en analoog voor 16 of 32. De PowerPC is dus een 32-bits CPU. De oorspronkelijke CPU in de IBM PC was een 8086 van het merk Intel. Dit was een 16-bits CPU. In de huidige IBM-compatibele PC's worden de Pentium of compatibele AMD processors gebruikt. Deze hebben nagenoeg dezelfde architectuur als hun voorgangers, de 80386 en zijn opvolger 80486. Dit zijn 32-bits CPU's maar terwille van oude programma's, zoals MS-DOS kunnen zij zich ook nog als een 16-bits processor gedragen.

Er worden nog steeds veel 8-bits processors gebruikt, maar dan vooral in wasmachines en andere huishoudelijke apparatuur, en in de simpelere spelcomputers zoals de Gameboy.

Er zijn eigenlijk 3 eigenschappen van een processor die samen bepalen een hoeveel-bits processor het is.

1. Op wat voor eenheden kan de CPU de meeste operaties uitvoeren, en hoe groot zijn de registers? Als de CPU intern de meeste operaties op 16 bits getallen kan uitvoeren is het dus een 16-bits processor. Het kan best zijn dat de CPU ook op kleinere eenheden kan werken, en misschien een paar bewerkingen kan doen op grotere. Maar doorslaggevend is wat de meeste instructies doen.

2. Hoe groot zijn de adressen? Dit hoeft niet hetzelfde te zijn als het vorige. Bijvoorbeeld de Intel 8086 (van de originele IBM PC) had 20 bits adressen, terwijl er intern met 16 bits gerekend werd. En veel 8-bits processoren hadden 16 bits adressen.

3. Hoeveel data gaat er tegelijk tussen de CPU en de bus? Dit kan ook nog anders zijn. Bijvoorbeeld de Motorola 68000 (toegepast in de originele Amiga, Atari, en Macintosh computers) was intern 32 bits, had 24-bits adressen, en had een 16-bits communicatie met de bus. Sommige CPU's kunnen zich zelfs aanpassen aan de bus. Als de bus te "smal" is dan halen ze de data in een paar keer op.

Wanneer bovengenoemde getallen niet hetzelfde zijn wordt meestal de interne maat van de processor genomen, maar soms worden ook wel eens uitdrukkingen als 16/32 bits processor gebruikt.

De 16-bits CPU's hebben belangrijke nadelen boven de 32-bits: ten eerste kunnen integers van 16 bits in de praktijk te weinig waarden aannemen (-32768 t/m 32767), maar het is via software mogelijk om 32-bits berekeningen te maken, alleen kost dit dan diverse instructies per berekening, en is dus langzamer. Ten tweede betekent het dat adressen ook maar 65536 waarden kunnen aannemen. Voor moderne programma's is dit veel te weinig. De 8086 CPU had een aantal trucs om de adresruimte op te voeren, maar ook hier geldt dat dit extra instructies kost en de programma's veel ingewikkelder maakt. We zullen verderop zien hoe dit o.a. gedaan wordt. Tegenwoordig zijn er voor zware toepassingen on onder andere servers als 64-bits processoren vooral vanwege de toegenomen adresruimte.

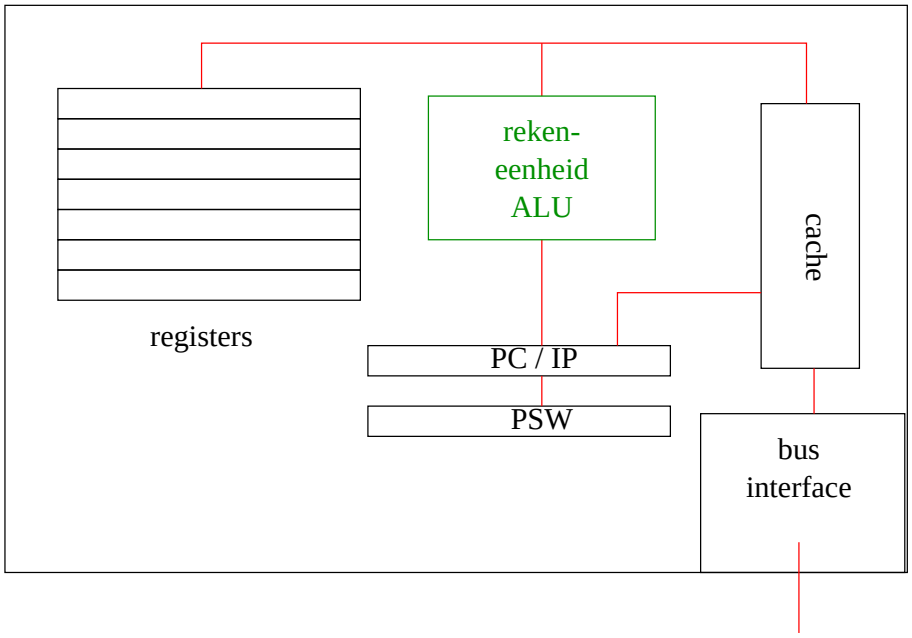
Behalve in het aantal en de grootte van registers kunnen CPU's ook verschillen in de manier waarop de registers gebruikt kunnen worden. We zeggen dat een processor een *regelmatige* registerverzameling heeft als ieder register voor hetzelfde doel gebruikt kan worden. In principe betekent dit dat elke instructie één of meer delen heeft waar een registernummer in staat. Er is dan bijvoorbeeld een instructie

```
ADD Rx, Ry, Rz
```

die de inhoud van de registers Rx en Ry optelt en het resultaat in Rz opbergt. x, y, z kunnen dan willekeurig zijn binnen het aantal registers dat aanwezig is. Een processor met een *onregelmatige* registerverzameling heeft een aantal registers die (bijna) allemaal een verschillende functie hebben. Daar kan dan bijvoorbeeld register A en B gebruikt worden om op te tellen, en register I alleen als index voor adressen. Het nadeel van de laatste opzet is dat je als (assembler) programmeur goed in de gaten moet houden welk register je voor welke taak kunt gebruiken. Er zullen dan vaak extra instructies nodig zijn om een register vrij te maken. Voor hogere programmeertalen zoals C zal de compiler dit moeten doen. Er is zoveel onderzoek naar dit laatste probleem gedaan, dat voor beide systemen genoeg algoritmen bekend zijn om dit probleem op te lossen. De processoren van het merk Intel (zoals de 80*86 familie waartoe ook de Pentium behoort) hebben meestal een onregelmatige registerverzameling, terwijl processoren van Motorola (de 68*00 familie en de PowerPC) meestal een regelmatige registerverzameling hebben.

1.2.2 CISC vs. RISC

De laatste jaren is de snelheid waarmee processors werken enorm toegenomen. De snelheid wordt meestal weergegeven als de *klokfrequentie*, wat aangeeft hoe vaak de interne klok van de processor tikt. Tegenwoordig ligt die voor processoren in PC's en werkstations meestal tussen de 1000 en 3000



Figuur 1.2: CPU opbouw (versimpeld)

MHz[1] (1 MHz = 1 miljoen keer per seconde). De klokfrequentie zegt niet alles, want ook het aantal kloktikken dat een instructie duurt kan verschillend zijn.

Uit sommige onderzoekingen is gebleken dat bepaalde ontwerpbeslissingen van een processor het versnellen ervan ernstig kunnen bemoeilijken. Bepaalde processoren hebben bijvoorbeeld instructies die heel gecompliceerde operaties uitvoeren, waarbij verschillende argumenten betrokken zijn. De processor moet dan zorgen dat alle argumenten op tijd op hun plaats zijn voordat de operatie uitgevoerd kan worden. Bij een zeer snelle processor is het moeilijker om te zorgen dat de argumenten allemaal op tijd op hun plaats zijn. Daarom heeft men op een gegeven moment processoren ontwikkeld waarbij elke instructie alleen een simpele operatie uitvoert. Deze processoren worden RISC (Reduced Instruction Set Computer) genoemd en de andere CISC (Complex Instruction Set Computer). Oorspronkelijk sloeg dit vooral op het aantal verschillende soorten instructies, tegenwoordig gaat het vooral om het feit of een instructie één simpele operatie of een gecompliceerde, samengestelde operatie uitvoert. Twee voorbeelden: CISC processoren hebben meestal instructies die een operatie uitvoeren waarbij een of meer argumenten uit het geheugen komen of ernaar toe gaan. Omdat de snelheid van het geheugen niet dezelfde spectaculaire groei meegemaakt heeft als de CPU's, betekent dit dat zo'n instructie opgehouden wordt door de toegang tot het geheugen. We zullen verderop zien dat er methoden zijn om de pijn te verzachten, maar deze lossen niet alles op. In een RISC computer werken de meeste instructies op registers, en zijn er voor het geheugen alleen instructies om gegevens tussen een register en het geheugen uit te wisselen. Het transport van en naar het geheugen is dan weliswaar nog steeds langzaam, maar dit kan onafhankelijk van de operaties uitgevoerd worden. Een compiler kan er dan voor zorgen dat het ophalen van een argument uit het geheugen ruim voor de operatie gebeurt. Het ophalen en opbergen door de CPU gebeurt dan *parallel* (=gelijktijdig) aan het uitvoeren van de andere operaties. RISC computers hebben dan meestal ook veel registers. Het tweede voorbeeld betreft het

[1] Deze getallen veranderen elk jaar: grofweg verdubbelen ze om de anderhalf jaar.

opslaan van condities door CISC processoren. In deze systemen wordt na het uitvoeren van de meeste instructies één of meer conditiecodes gezet, bijvoorbeeld of het resultaat van de operatie positief, nul of negatief is, of er overflow is opgetreden, etc. Deze condities worden in een speciaal register in de CPU bewaard Er zijn dan speciale conditionele sprongopdrachten die het resultaat hiervan weer gebruiken t.b.v. *if* of *while* constructies. De conditiecodes vormen echter een bottleneck, die het parallel uitvoeren van instructies bemoeilijken. Als je bijvoorbeeld een instructie wilt uitvoeren die een conditiecode zet, dan moet de CPU wachten tot alle voorafgaande instructies uitgevoerd zijn die de vorige waarde van de conditiecode gebruiken. Bij een RISC processor worden vaak geen conditiecodes gebruikt, maar wordt het resultaat van een test in een van de algemene registers opgeborgen. De conditionele opdrachten testen dan de waarde van zo'n register. Als nu opvolgende series { zet conditie; gebruik conditie } instructies verschillende registers gebruiken om de resultaten van de test in op te slaan, dan zitten ze elkaar niet in de weg, en kunnen dus parallel uitgevoerd worden, wat de snelheid verhoogt.

RISC processoren zijn minder ingewikkeld van interne structuur dan CISC processoren, en kunnen dus op een kleinere chip gebouwd worden. Omdat hierbij minder warmte geproduceerd wordt, kan een RISC chip eerde op een hogere klokfrequentie draaien dan een CISC chip[2].

De strijd tussen RISC en CISC is nog niet beslecht. RISC biedt voordelen voor het versnellen van processoren, omdat zowel het opvoeren van de klokfrequentie als het parallel uitvoeren van instructies makkelijker is. Anderzijds doet een RISC instructie minder dan een CISC instructie, dus zijn er meer instructies nodig om hetzelfde resultaat te bereiken. Al deze instructies moeten uit het geheugen gehaald worden, terwijl dat geheugen nu net veel langzamer is dan de processor. Dit is dus weer een vertragende factor. De CISC fabrikanten (zoals Intel) proberen nu hun processoren zo te ontwerpen dat ze er aan de buitenkant als aan CISC CPU uitzien, maar van binnen als een RISC werken. Er zit dan een vertaal-laag in tussen die twee, maar die maakt de chip natuurlijk weer ingewikkelder.

1.3 Stack en Heap

Een *stack* of stapel is een deel van het geheugen wat gebruikt wordt voor tijdelijke opslag van gegevens. Een stack is niet een speciaal soort geheugen, maar het is alleen een speciaal gebruik van het geheugen. Er kunnen meerdere stacks zijn, bijvoorbeeld één voor het O.S. zelf, en één voor elk programma dat in de computer loopt.

Een stack is te vergelijken met een stapel borden in een restaurant of kantine: Je pakt altijd het bovenste bord en als er nieuwe borden komen worden die weer bovenop de stapel gelegd. Die operaties noemen we *push* (zet iets op de stack) en *pop* (haal iets van de stack af). Omdat het laatste dat op de stack gezet is er het eerst weer afgaat noemen we dat *LIFO* (Last In, First Out). Dit is in de informatica een veel voorkomend mechanisme, vooral omdat de administratie ervan zo gemakkelijk is. Je hoeft namelijk alleen maar een pointer naar de top van de stack bij te houden, de z.g. *stackpointer*. Moderne CPU's hebben hiervoor een apart register, zodat veranderingen in de stackpointer snel zijn.

Hoewel je een stack voor van alles en nog wat kunt gebruiken zijn er een paar toepassingen die je bijna altijd tegenkomt:

1. Als in een programma een methode (ook wel functie of procedure genoemd) aangeroepen wordt, dan moet de CPU onthouden waar het programma verder moet gaan als de methode

[2]Bij een hogere klokfrequentie wordt er namelijk meer warmte geproduceerd

klaar is. In feite is dit het adres van de instructie na de aanroep. Dit adres wordt op de stack gezet. Ga zelf na dat dit klopt, dus wanneer binnen de methode weer een andere methode aangeroepen wordt, inderdaad het LIFO principe gebruikt wordt.

2. Als een ingewikkelde expressie uitgerekend moet worden en er zijn niet genoeg registers om alle tussenresultaten op te slaan dan worden de andere tussenresultaten opgeslagen op de stack.

3. Argumenten en locale variabelen van een methode worden ook op de stack opgeslagen.

Niet alle gegevens kunnen op een stack opgeslagen worden: Bijvoorbeeld objecten die langer moeten bestaan dan de methode (functie, procedure) waarin ze gemaakt worden kunnen niet op de stack omdat het stuk stack al opgeruimd wordt terwijl de gegevens nog nodig zijn. Denk bijvoorbeeld aan een Java methode die een String aanmaakt. De resulterende String kan niet op de stack staan omdat deze nog nodig is nadat de methode beëindigd is. Dit soort objecten, en in het algemeen stukken geheugen die zich niet aan de LIFO discipline houden worden in een apart stuk geheugen gezet dat de *heap* genoemd wordt.

Stack:
Een gereserveerd stuk geheugen waarin de gegevens volgens een LIFO (Last In, First Out) discipline opgeslagen worden.

Heap:
Een gereserveerd stuk geheugen voor gegevens die niet volgens de LIFO discipline opgeslagen kunnen worden.

In het geheugen van een programma moet dus tenminste plaats zijn voor de instructies, de stack en de heap.

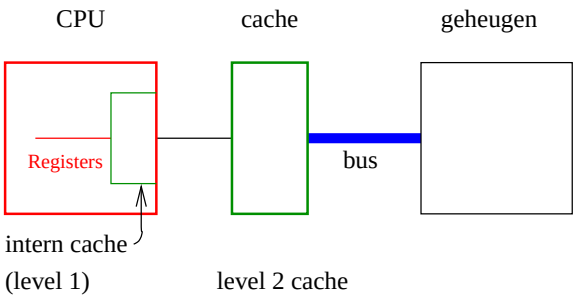
1.4 Cache

Zoals we al gezien hebben is bij moderne computers de snelheid van de CPU groter dan die van het geheugen. Gedeeltelijk wordt dit gecompenseerd als de CPU veel registers heeft, maar toch willen we niet dat de CPU moet wachten als er iets uit het geheugen gehaald (*gelezen*) of teruggeschreven moet worden. Daarom wordt er tussen de CPU en het geheugen een stuk snel (en dus duurder) geheugen gezet. Dit wordt een *cache* genoemd.

Cache:
Een stuk snel geheugen waarin veel gebruikte gegevens uit een langzamer geheugen tijdelijk opgeslagen worden om sneller bereikbaar te zijn. Bijvoorbeeld een supersnel geheugen als cache voor het RAM, het RAM als cache voor de harde schijf of de harde schijf als cache voor WWW-pagina's.

Sommige processoren hebben een cache op de chip zitten, maar daar is slechts beperkte ruimte. Meestal wordt daarom nog een extra cache buiten de chip geplaatst. Het cache in de CPU wordt wel een *intern* of level 1 cache genoemd, het andere *extern* of level 2.

Een cache bewaart (in principe) een kopie van de meest gebruikte geheugenlocaties. Als de CPU een bepaald geheugenwoord nodig heeft wordt eerst in de cache gekeken of het aanwezig is. Als dat het geval is heeft de CPU snel de benodigde waarde binnen. Zo niet, dan moet het echte geheugen aangesproken worden. Zie figuur 1.3. Wanneer zowel een intern als een extern cache aanwezig is



Figuur 1.3: Cache(s)

wordt eerst in het interne cache gekeken, als de locatie hierin niet aanwezig is, in het externe cache, en tenslotte in het hoofdgeheugen. Natuurlijk gebeurt dit allemaal in hardware, en wel supersnel. Er zijn een aantal belangrijke ontwerpbeslissingen bij een cache:

1. Een cache heeft natuurlijk een kleinere capaciteit dan het hoofdgeheugen. Wanneer het cache vol is zal moeten worden beslist wat er verwijderd kan worden om plaats te maken voor de nieuwe data. Dit heeft ook te maken met de manier waarop het cache geadresseerd wordt. In een simpel cache zouden we een hash-functie op het geheugenadres kunnen loslaten om het cache-adres te vinden. Dit kan zo iets simpels zijn als de laagste n bits nemen van het adres. In dat geval wordt gewoon de oude waarde die op dat cache-adres stond weggegooid. Het cache moet dan zowel het echte adres als de waarde van de geheugenlocatie in het cache opnemen. Als het adres klopt hebben we een *hit*, anders een *miss*.

   In een meer geavanceerd cache zouden we per cache geheugenplaats het echte geheugenadres kunnen opnemen, en de waarde, maar de plaats waar in het cache deze opgenomen worden vrij kunnen laten. Het cache moet dan bij een verzoek razendsnel alle opgeslagen adressen doorzoeken om te vinden waar in het cache de waarde zit. Dit kan met een z.g. *associatief geheugen* een stukje electronica dat parallel een waarde zoekt in een verzameling. Als nu het adres niet aanwezig is moet via een of andere algoritme beslist worden welke cache-plaats vrij gemaakt kan worden. Aangezien niet te voorspellen is welke plaatsen in de toekomst weer gebruikt zullen worden, wordt meestal een plaats hergebruikt die het langst niet gebruikt is (LRU = *least recently used*). Een andere mogelijkheid is om random een cache-locatie te kiezen.

   Voor grotere caches is een volledig associatief geheugen i.h.a. te duur. Dan kan een combinatie van bovenstaande principes gebruikt worden: bijvoorbeeld een hash-functie op het adres wijst een verzameling van k (bijv. 4) cache-locaties aan, waarbinnen associatief gezocht wordt, en op een van genoemde manieren een keuze gemaakt wordt.

2. Wat moet er gebeuren wanneer gegevens naar het geheugen toe geschreven worden? Omdat het waarschijnlijk is dat de weggeschreven geheugenplaats binnen niet al te lange tijd wel weer gebruikt zal worden, ligt het voor de hand om de waarde in de cache op te slaan. Ook moet de waarde naar het echte geheugen geschreven worden. Moet de CPU echter wachten tot dit laatste ook gebeurd is of niet? Als niet gewacht wordt, dan is de CPU sneller. Maar dit kan tot gevolg hebben dat er een queue ontstaat van woorden die nog naar het geheugen geschreven moeten worden. Dit moet als een aparte queue in hardware geïmplementeerd worden, omdat anders de betreffende cache-locaties niet opnieuw gebruikt kunnen worden. Als er in de tussentijd ook

woorden uit het geheugen naar de cache moeten, dan zouden deze voorrang kunnen krijgen. Al met al geeft dit een behoorlijk gecompliceerde cache-implementatie. Een cache waarbij wel gewacht wordt tot de gegevens weggeschreven zijn heet een *writethrough* cache. Een cache waarbij niet gewacht hoeft te worden heet een *delayed-write* of *write-behind* cache.

3. Hoeveel woorden halen we tegelijkertijd uit het geheugen naar de cache? De meeste programma's hebben een eigenschap die *lokaliteit van referentie* wordt genoemd. D.w.z. wanneer aan een bepaalde geheugenplaats gerefereerd wordt, is het waarschijnlijk dat in de buurt gelegen geheugenplaatsen niet lang daarna ook nodig zijn. Daarom zijn veel caches zo ingericht dat niet één geheugenplaats tegelijkertijd opgehaald wordt, maar meer. Wanneer bijv. N woorden tegelijk in het cache gestopt worden kan dit zo gedaan worden dat alle adressen die modulo N gelijk zijn bij elkaar gestopt worden. Meestal wordt voor N een 2-macht genomen, zodat we dan de laagste bits niet hoeven op te slaan, resp. te vergelijken.

4. Een *multiprocessor* systeem is een computersysteem waar meer dan één CPU in zit. Vaak wordt het geheugen dan gedeeld. Wanneer echter elke CPU zijn eigen cache heeft dan kan de ene CPU een geheugenplaats wijzigen, terwijl de andere CPU nog een oude kopie in het cache heeft. Er moet dus voor gezorgd worden dat de caches up to date blijven. Dit heet het *cache consistentie probleem*. Moderne caches zijn zo ontworpen dat zij de bus waarop ze aangesloten zitten in de gaten houden (*snooping*), en wanneer ze ontdekken dat een schrijfopdracht naar een bepaald geheugenadres plaatsvindt, dan maken ze een eventuele kopie die ze hebben ongeldig of ze nemen de nieuwe waarde over.

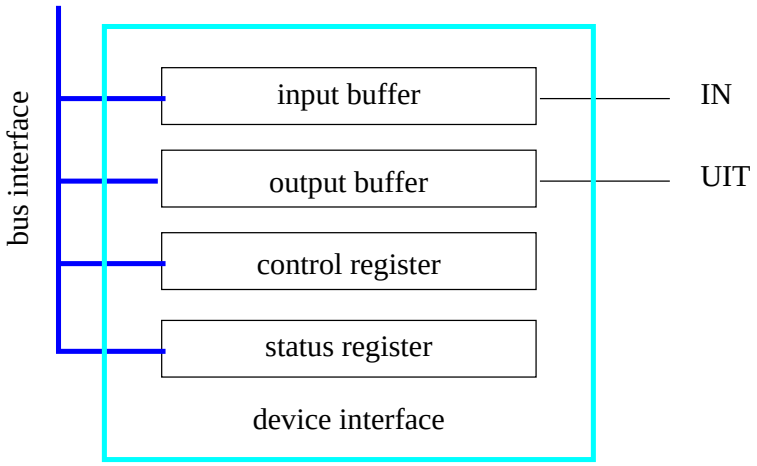
Sommige cache implementaties hebben aparte caches voor instructies en data, andere stoppen ze beide in dezelfde cache. Het is niet duidelijk welk systeem het beste is.

Het idee van caches is overigens algemeen toepasbaar, niet alleen tussen de CPU en het interne geheugen. We spreken van een cache wanneer een (deel van een) sneller geheugen gebruikt wordt om veel gebruikte (dat hopen we althans) delen van een langzamer geheugen tijdelijk te onthouden. Op soortgelijke manier kunnen we intern geheugen gebruiken als cache voor een extern geheugen (floppy, harde schijf, CD-ROM), en een deel van een schijf als cache voor documenten uit het Internet.

1.5 In- en uitvoer

De CPU zal met een groot aantal randapparaten moeten communiceren, zoals toetsenbord, beeldscherm, schijven, printers. Zoals we al gezegd hebben zijn er voor de apparaten *interfaces* ingebouwd, waar gegevens naartoe gestuurd en vandaan gehaald moeten worden. Elke interface heeft één of meer eigen registers waar gegevens in gezet of uitgehaald worden. Voor een toetsenbord is dat heel simpel een register waar de code van de ingetypte toets instaat en een register (bit) waarin staat of er een toets ingedrukt is. Voor een schijfgeheugen zijn er veel meer gegevens nodig: opdrachten om een bepaalde plaats op de schijf op te zoeken, blokken gegevens die heen en weer gestuurd worden etc. Elk register van elke interface zal een eigen nummer of adres krijgen om het te onderscheiden van alle andere.

In figuur 1.4 zien we een simpele interface, bestaande uit een input- en een outputbuffer, een control- en een statusregister. Een dergelijke interface wordt wel gebruikt om met een printer, toetsenbord of een modem te communiceren. In de outputbuffer kan een waarde gezet worden die naar buiten de computer moet, bijv een teken dat geprint moet worden. Op de zelfde manier komt een waarde die naar binnen moet (bijv een toets die ingedrukt is) in de inputbuffer te staan. Het controlregister



Figuur 1.4: Simpele device interface

is er om aan de interface aan te geven of deze bepaalde acties moet ondernemen. Bijvoorbeeld of een “reset” operatie uitgevoerd moet worden en of de interface interrupts (zie 1.6) moet geven. Het statusregister wordt door de interface gebruikt om extra informatie aan de CPU te geven, bijvoorbeeld of er een teken afgedrukt is, of een toets ingedrukt. In het algemeen of de gevraagde operatie klaar is. Daarnaast kunnen er ook bits instaan voor bepaalde foutcondities (papier op, apparaat staat niet aan, modem heeft verbinding etc.)

Er zijn twee manieren waarop een CPU met de interfaces kan communiceren:

1. Via speciale in- en uitvoer (I/O) instructies. Dit kan natuurlijk alleen als de CPU ontworpen is met deze instructies. Voor in- resp. uitvoer zouden dan instructies van de vorm:

   ```
   IN  iadres, geheugenlocatie
   OUT iadres, waarde
   ```

   gebruikt kunnen worden, waarbij `iadres` het speciale interface adres is. In de 80386 processors zijn er bijv. zulke instructies, waarbij de interface adressen 16-bits waarden kunnen hebben.

2. Een andere veel gebruikte mogelijkheid is om een deel van de gewone geheugenadressen te gebruiken voor de aansluiting van interfaces. Dit kan door de electronica die geheugenadressen decodeert, bij gebruik van deze adressen niet het echte geheugen te laten selecteren maar een signaal naar de interfaces te laten genereren. We spreken dan over *memory-mapped* I/O.

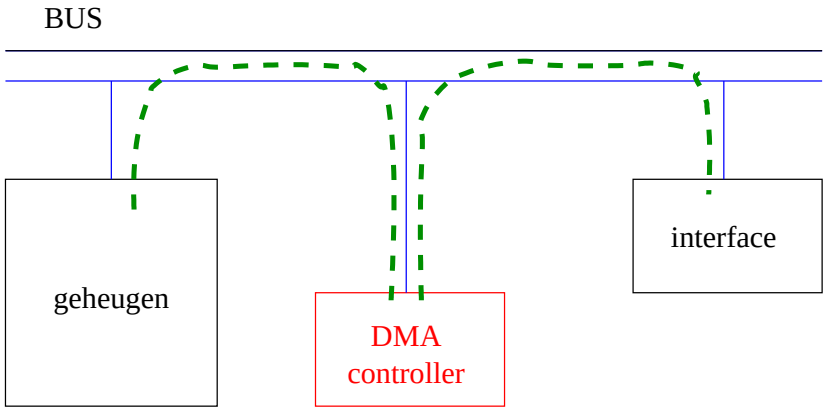
Memory-mapped I/O:
Een manier om gegevens tussen CPU en randapparaten uit te wisselen, waarbij geen speciale I/O instructies gebruikt worden, maar de registers van de interface van het apparaat reageren op daarvoor gereserveerde geheugenadressen.

   Bij een CPU die geen speciale I/O instructies heeft is dit de enige manier om in- en uitvoer te plegen maar ook bij een CPU met deze instructies wordt dit gebruikt. I.h.a. is dit een wat flexibelere manier voor de ontwerper van de computer, maar het nadeel is dat een deel van de

geheugenadressen opgeofferd wordt. Bij de originele IBM PC waren de ontwerpers er vanuit gegaan dat de hoeveelheid geheugen die ze in de computer ingebouwd hadden ruim voldoende was voor ieder denkbaar gebruik. De 8086 CPU die ze gebruikten kon een maximum van 1MB geheugen aan (20 adresbits), maar men vond 640KB genoeg voor normaal gebruik. De hoogste 384KB werd gereserveerd voor interfaces. De IBM PC computers gebruiken overigens zowel IN/OUT instructies als memory-mapped I/O.

Moderne computers hebben meestal 32 of 64 bits adressen, en daarbij is het veel minder erg om een deel van de adresruimte voor I/O te reserveren.

3. Bij het verplaatsen van grotere hoeveelheden data tussen interface en CPU is het nogal traag wanneer dit woord voor woord moet gebeuren. Het is dan beter om dit verplaatsen direct tussen de interface en het geheugen te laten plaatsvinden, zonder tussenkomst van de CPU. Dit wordt Direct Memory Access (DMA) genoemd en kan zowel voor invoer als voor uitvoer gebruikt worden. Er is dan een aparte DMA *controller* (of meer dan één) die dit werk doet. De DMA controller kan ingebouwd zijn in de CPU of als aparte chip op de bus aangesloten zijn. De DMA controller kan worden beschouwd als een uiterst primitieve processor die alleen maar blokken geheugen van/naar een interface kan sturen. Sommige DMA controllers kunnen ook een blok in het geheugen kopiëren. De DMA controller moet wel een seintje naar de CPU kunnen geven dat het werk voltooid is (of een fout opgetreden). Zie de volgende sectie. Bij



Figuur 1.5: DMA controller

gebruik van DMA en een cache moet erop gelet worden dat de CPU niet een waarde uit het cache haalt als de DMA controller de geheugenwaarde veranderd heeft. Wanneer het cache niet zelf hierop let, zal de software die de DMA aanstuurt moeten zorgen dat het geheugen dat door de DMA controller gewijzigd is, na de DMA operatie in het cache ongeldig gemaakt wordt. Evenzo moeten waarden die nog in het cache wachten om weggeschreven te worden eerst naar het geheugen gaan voor een DMA operatie gestart wordt die deze gegevens van het geheugen naar een interface transporteert.

Bij gebruik van memory mapped I/O moet erop gelet worden dat de bijbehorende geheugenlocaties niet gecached worden. We willen nl. dat elke toegang tot zo'n locatie door de interface opgemerkt wordt.

1.6 Interrupts

De meeste gebeurtenissen waar een CPU op moet reageren gebeuren op onvoorspelbare tijdstippen. Zo is het niet bekend wanneer de gebruiker de volgende toets op het toetsenbord zal indrukken. Zelfs een simpele gebeurtenis als het einde van een schrijfoperatie naar een schijf is binnen zekere grenzen onvoorspelbaar. In een simpele toepassing zoals het regelen van een CV of een wasmachine, is het mogelijk een programma te schrijven dat periodiek alle interfaces test om te zien of er iets voor gedaan moet worden. Dit wordt *polling* genoemd. In deze simpele situaties komen er meestal zo weinig signalen binnen dat de CPU ruimschoots de tijd heeft om ze allemaal één voor een af te tasten. Bovendien heeft de CPU daarnaast weinig te doen.

In een algemeen computer systeem, waarin veel apparaten actief kunnen zijn, en de snelheden van de apparaten onderling veel kunnen verschillen, kan polling een te grote belasting zijn. Bovendien wordt de software ingewikkeld als voortdurend even afgeweken moet worden van de berekening die aan de gang is om te kijken of er een apparaat is dat aandacht vraagt. De kans is ook nog groot dat we bij een bepaald apparaat net te laat zijn. Tenslotte is het jammer van al die tijd die verloren gaat met het kijken naar een apparaat dat toch niets te doen heeft. Om dit probleem op te lossen zijn CPU's uitgerust met een *interrupt* mogelijkheid.

Polling:
Een manier om uit te vinden of er iets interessants gebeurd is, waarbij je in een programma regelmatig controleert of de interessante gebeurtenis plaatsgevonden heeft.

Interrupt:
Een signaal dat een apparaat aan de CPU geeft, zodat de CPU zijn werk zal onderbreken om iets voor dat apparaat te doen.

Een interrupt is een signaal dat (meestal) van buiten de CPU komt, bijv. van een interface, en dat de normale volgorde van de uitvoering van instructies onderbreekt, om een ander stuk programma te gaan uitvoeren. Omdat we na afloop het originele programma weer ongestoord verder willen laten gaan, zal de interrupt de plaats waar het programma onderbroken werd moeten opbergen. Deze plaats (geheugenadres) staat meestal in een apart register, de *program counter* (PC) of *instruction pointer* (IP). Het opslaan van de program counter moet gebeuren door het deel van de CPU dat de interrupt herkent. Bij een CPU met aparte conditiecodes moeten deze ook opgeslagen worden omdat de eerste instructie die door de interrupt uitgevoerd wordt, deze kan veranderen. Meestal zijn er nog andere interne machinetoestanden die ook bewaard (*gered*) moeten worden en die samen met de conditiecodes in een *program status word* (PSW) opgeslagen worden. De PC en het PSW kunnen in vaste geheugenlocaties opgeborgen worden of op een stack. Bij RISC machines worden ze ook wel opgeborgen in een apart gereserveerde verzameling registers (omdat RISC machines niet meer dan één ding tegelijk naar het geheugen willen schrijven).

Na het opbergen van de PC/PSW moeten er instructies uitgevoerd gaan worden die ergens anders in het geheugen staan. Sommige processoren beginnen bij een interrupt op een vaste geheugenplaats, bij andere kan de interface die de interrupt geeft, specificeren welke instructies uitgevoerd moeten worden. Een veel gebruikt systeem is de *vectored interrupt*. Hierbij geeft de interface die de interrupt veroorzaakt een index in een array (vector) van adressen. Het adres op de juiste plaats wordt opgehaald en de instructies die op dat adres staan (de z.g. interrupt routine) worden na de interrupt uitgevoerd. In feite wordt dit adres in de Program Counter gezet. Vaak staat er niet alleen een programma-adres, maar een ook een nieuwe waarde voor het PSW. Dan worden dus beide vanuit de vector locatie gehaald,

wat dus het omgekeerde is als het opbergen aan het begin van de interrupt. De eerste instructies van de interrupt routine zullen meestal ook nog andere registers redden die in de interrupt-routine gebruikt worden. Aan het eind van de interrupt-routine moeten de originele opgeborgen PC en PSW waarde weer teruggezet worden. Dit gebeurt meestal d.m.v. een speciale instructie.

Vectored interrupt:

Een interrupt die een nummertje bij zich heeft, waardoor het adres waar de interrupt routine staat uit een vector gehaald wordt met het nummer als index.

Interrupt routine:

De instructies die uitgevoerd moeten worden als een bepaalde interrupt binnenkomt.

Wanneer meer dan een apparaat van dezelfde interrupt vector gebruik maakt, dan moet de interrupt routine uitvinden welk apparaat de interrupt veroorzaakt heeft. Dit gebeurt dan door polling, maar alleen binnen de verzameling apparaten die deze interrupt kunnen genereren. Dit is efficiënter dan alle apparaten te moeten pollen, bovendien gebeurt het alleen op een tijdstip waarop het zeker is dat er iets gedaan moet worden.

Wanneer een interrupt plaatsvindt is het i.h.a. nodig om te voorkomen dat direct daarna weer een interrupt plaatsvindt. Het signaal dat de interface genereert om de interrupt te activeren wordt meestal pas weggehaald als de CPU actie heeft ondernomen om de toestand die de interrupt veroorzaakt ongedaan te maken. Bijvoorbeeld bij een interrupt van het toetsenbord (als er een toets ingedrukt is), zal het signaal weggenomen worden als de code van de toets door de CPU uitgelezen is. De meeste CPU's hebben een mogelijkheid om interrupts te "disabelen", d.w.z. tijdelijk uit te schakelen. De interrupt-signalen verdwijnen dan niet, maar de CPU geeft er tijdelijk geen aandacht aan. Dit gebeurt vaak doordat een speciaal bit in het PSW gezet wordt. Wanneer de interrupt een nieuwe waarde van de PSW zet, kan dit mooi daarin gedaan worden. Er moet echter op gelet worden dat interrupts niet te lang uitstaan, omdat sommige apparaten snel behandeld moeten worden. Anders kunnen er gegevens verloren gaan. Omdat er nogal wat verschil is tussen het ene apparaat en het andere, zijn interrupts vaak in *prioriteiten* ingedeeld. Een interrupt van prioriteit 1 zal dan bijv. minder urgent zijn dan een interrupt van prioriteit 3. De interrupt prioriteit wordt bepaald door het signaal dat de interface genereert, er kan bijv (op de bus) voor elk prioriteitsniveau een aparte draad zijn. Om binnen de CPU de voorrang te regelen is er dan in het PSW een getal dat aangeeft welke prioriteiten doorgelaten worden en welke niet. In het PSW van een bepaalde interrupt routine staat dan een waarde die tot gevolg heeft dat deze interrupt routine alleen nog maar door hogere prioriteits interrupt mag worden onderbroken. Omdat interrupt routines onderbroken kunnen worden door andere interrupts zullen de PSW en PC (en evt. andere registers) op een stack opgeborgen moeten worden. Bij een simpele computer, bijv de reeds genoemde wasmachinebesturing, kan dat op dezelfde stack als de gewone programma's. Maar bij een wat ingewikkelder O.S, en vooral bij een multi-user systeem (waar dus meer dan een gebruiker op zit), is het verstandig als hiervoor een aparte stack gebruikt wordt. Anders zou een programma dat de stackpointer vernielt, interrupts de mist in kunnen laten gaan. Terwijl die interrupts misschien wel iets doen voor een andere gebruiker. Let op dat het gebruik van aan aparte *interrupt stack* alleen mogelijk is als dit in het ontwerp van de CPU zit.

1.7 Traps

Er zijn situaties waarbij een interrupt gegenereerd wordt door de CPU zelf i.p.v. een externe interface. Voorbeelden hiervan zijn: een deling door 0, het gebruik van een niet-bestaand adres, het uitvoeren van een illegale instructie. In deze gevallen spreken we meestal van een *trap* i.p.v. een interrupt. Het woord interrupt reserveren we dan voor extern veroorzaakte onderbrekingen. Het grote verschil is dat een interrupt op een willekeurig tijdstip kan gebeuren, het is dus niet te zeggen welke instructie de CPU op dat moment aan het uitvoeren is, terwijl een trap altijd op een precies gedefinieerde instructie plaatsvindt, omdat deze door die instructie veroorzaakt wordt. De meeste CPU's hebben zelfs één of meer speciale instructies die een trap veroorzaken, vaak TRAP of INT of iets dergelijks genoemd. Deze worden gebruikt o.a. voor de communicatie van gebruikersprogramma's met het O.S., voor communicatie tussen onderdelen van het O.S, of voor het debuggen van programma's.

Trap:
Een signaal vanuit de CPU dat het programma dat uitgevoerd wordt onderbreekt om (tijdelijk) andere instructies te gaan uitvoeren. Verder vergelijkbaar met een interrupt.

1.7.1 Interrupts en exceptions in RISC processors

In RISC processors doet iedere instructies een simpele operatie. In het bijzonder zal elke instructie meestal maximaal één geheugenoperatie uitvoeren. Een interrupt of exception kan ook als een instructie beschouwd worden. De voorgaande behandeling gaf echter aan dat bij een interrupt diverse dingen moeten gebeuren: zowel de PC als het PSW moeten opgeborgen worden en nieuwe waarden hiervoor moeten uit een tabel in het geheugen gevist worden. Bovendien moet de stackpointer veranderd worden. Het is duidelijk dat dit tegen de RISC principes indruist.

Daarom wordt bij RISC processors meestal een simpeler interrupt mechanisme toegepast. In overeenstemming met de RISC filosofie moeten de ingewikkelder operaties opgebouwd worden door meerdere instructies uit te voeren. De manier die bijv. bij de MIPS processors (o.a. toegepast in SGI werkstations en de Nintendo-64) gebruikt wordt is dat de CPU een paar aparte registers heeft voor interrupts. Eén van deze wordt gebruikt om de PC in op te bergen, een andere krijgt een code waarin staat wie de interrupt veroorzaakt heeft. Dit register kan gebruikt worden om via een tabel te springen. Het opbergen van de oude PC op een stack kan dan met normale instructies gebeuren. Verder bevat het PSW een mini-stackje voor het opbergen van status bits en interrupt-enable bit.

1.8 Supervisor mode

In een multi-user systeem, of elk ander systeem waarbij beveiliging belangrijk is, moet voorkomen worden dat elk willekeurig programma elke actie kan uitvoeren. Als iedereen de interface van de harde schijf kan programmeren, dan is geen enkel gegeven veilig voor niet-geautoriseerde gebruikers. Dit heeft dus tot gevolg dat bepaalde instructies (zoals in- en uitvoer-instructies) niet door elk programma gebruikt mogen worden. Bij memory-mapped I/O zal het deel van de adresruimte dat voor I/O gereserveerd is beschermd moeten worden tegen oneigenlijk gebruik.

Wanneer meer dan één programma in het geheugen aanwezig is, dan zal ook het geheugen dat bij het ene programma hoort, beschermd moeten worden tegen verandering of zelfs ingluren van een ander

programma. Aan de andere kant moet het O.S. toegang tot het gehele geheugen kunnen hebben, o.a. omdat het programma's moet kunnen laden van de harde schijf naar het interne geheugen, en om de I/O voor de programma's te kunnen doen.

Het is dus duidelijk dat in zo'n systeem er onderscheid moet zijn tussen de privileges die bepaalde programma's mogen hebben. In het simpelste geval zijn er twee soorten: geprivilegieerde programma's (meestal de onderdelen van het O.S), die alles mogen, en "gewone" programma's die alleen ongevaarlijke dingen mogen doen. In een uitgebreider schema is het mogelijk om nog tussenvormen te hebben. We zullen ons voornamelijk beperken tot de simpele vorm. Hierbij is op elk tijdstip de CPU in één van twee *modes*, meestal user mode en system mode (of supervisor mode) genoemd. Gevaarlijke acties mogen alleen in supervisor mode uitgevoerd worden. De mode wordt aangegeven door een bit in het PSW (bij een uitgebreidere structuur door meer bits). Het veranderen van deze bits is natuurlijk ook een beschermde actie. De vraag die zich dan voordoet is hoe we van de ene mode naar de andere omschakelen. De overgang van supervisor mode naar user mode is simpel: als we in supervisor mode zitten, hebben we het recht om naar user mode over te schakelen. Omgekeerd niet. Eén van de manieren om van user mode naar supervisor mode te gaan, is door een interrupt: interrupt routines behoren bij het O.S., ze moeten i.h.a. I/O doen en moeten dus in supervisor mode draaien. Aangezien de PSW van de interrupt routine uit de interrupt vector gehaald wordt, kan hier het supervisor-mode-bit in aangezet worden, en zal de interrupt routine automatisch in supervisor-mode draaien. Als de interrupt een gebruikersprogramma onderbrak, dan wordt de PSW van dit programma gered, en na afloop van de interrupt routine weer teruggezet, zodat de CPU daarna weer in user mode staat.

User mode:
De toestand van de CPU waarbij alleen instructies uitgevoerd mogen worden die niet gevaarlijk zijn voor het Operating System. Gebruikersprogramma's draaien normaliter in user mode.

System mode:
De toestand van de CPU waarbij alle instructies mogelijk zijn; normaal alleen toegestaan voor het Operating System zelf.

Nu zien we ook waarom de traps belangrijk zijn: Hiermee kunnen we het O.S. vanuit user mode binnenkomen. Als in de PSW van de trap vector supervisor mode aanstaat dan kan het O.S. op de juiste manier verder. Bij de meeste O.S's wordt daarom een trap instructie gebruikt voor aanroepen van O.S. functies (de systeemaanroepen of *system calls*). De parameters voor deze systeemaanroepen worden dan op de stack of in vaste registers gezet. De eerste parameter geeft meestal een nummer aan dat de soort aanroep bepaalt. De trap routine begint dan met te testen of deze parameter binnen de juiste grenzen ligt en springt met een *switch* constructie naar de juiste routine. Door de instructies van het O.S. ook af te schermen van de gebruikersprogramma's kunnen deze niet meer willekeurige functies uit het O.S. aanroepen, maar komen ze altijd binnen via een duidelijke "toegangspoort" waar gecontroleerd wordt of het programma geen illegale operaties wil uitvoeren. Het is overigens best mogelijk dat delen van het O.S. die daarvoor in aanmerking komen in user mode draaien.

De oorspronkelijke 8086 CPU van de IBM PC en de 68000 CPU van de Atari, Amiga en Macintosh computers hadden geen bescherming. Latere versies zoals de 80386 en de 68030 wel.

1.9 Geheugenbeheer

Het geheugen in een computer wordt door allerlei onderdelen van het O.S. gebruikt. Het O.S. zelf staat erin, maar het is niet altijd nodig om het gehele O.S. in het geheugen te hebben. Sommige stukken kunnen ingeladen worden als ze nodig zijn, en na afloop kan het gebruikte geheugen weer vrijgegeven worden. Het O.S. heeft datastructuren nodig om zijn administratie bij te houden, en buffers voor het doen van in- en uitvoer. De benodigde hoeveelheid zal in de loop van de tijd variëren. Verder worden er regelmatig gebruikersprogramma's gestart die geheugen nodig hebben voor hun instructies, data en stack. Ook hier kan de benodigde hoeveelheid geheugen voortdurend variëren. Vooral in een multitasking systeem kan het dan nodig zijn om te gaan zitten schuiven om ruimte te maken. Stel bijvoorbeeld dat twee programma's direct achter elkaar in het geheugen staan, en het eerste heeft meer ruimte nodig. In veel situaties is het een vereiste dat de nieuwe ruimte direct achter de oude komt. Het tweede programma moet dan naar achteren geschoven worden, of het eerste moet teruggeschoven worden. Als we geen bijzondere maatregelen getroffen hebben, kan dat echter niet zomaar, omdat de adressen in de instructies, de variabelen en de stack dan niet meer kloppen. Een simpele oplossing voor dit probleem kan verkregen worden als de CPU een apart register, het z.g. BASE register heeft dat als beginpunt van een programma genomen wordt. De adressen in een programma beginnen dan vanaf 0, maar om de echte adressen te krijgen wordt bij elke geheugenreferentie de waarde van het BASE register erbij geteld. Op deze manier is het simpel om een programma in het geheugen te verschuiven: het BASE register wordt evenveel aangepast en intern in het programma worden nog steeds dezelfde adressen gebruikt. Om het verschil tussen de adressoorten aan te geven gebruiken we de term "virtuele adressen" voor de adressen die in het programma gebruikt worden, en "fysieke adressen" voor de "echte" adressen in het geheugen. En het verband is simpel: een fysiek adres is het virtuele adres plus het BASE register. Wanneer het O.S. wisselt tussen twee programma's moet het BASE adres ook omgewisseld worden: het O.S. houdt dus per programma een BASE register bij. Een ander voordeel is dat bij het laden van een programma in het geheugen er geen aanpassing van de adressen aan de gekozen locatie hoeft te gebeuren. Het denkbeeldige geheugen dat bij de virtuele adressen hoort, noemen we *virtueel geheugen*. Het 'echte' geheugen noemen we dan *fysiek geheugen*. Voor virtueel geheugen gebruik je dus virtuele adressen en voor fysiek geheugen gebruik je fysiek geheugen. Het zou nu duidelijk moeten zijn dat je als programmeur meestal alleen te maken hebt met virtuele adressen.

Virtueel geheugen:
Het denkbeeldige geheugen dat in een programma gebruikt wordt.

Fysiek geheugen:
Het echte geheugen dat uiteindelijk door het programma gebruikt wordt als het nodig is.

Virtueel adres:
Een adres in een programma dat een plaats in het virtueel geheugen aanwijst.

Fysiek adres:
Een adres dat een plaats in het fysieke geheugen aanwijst.

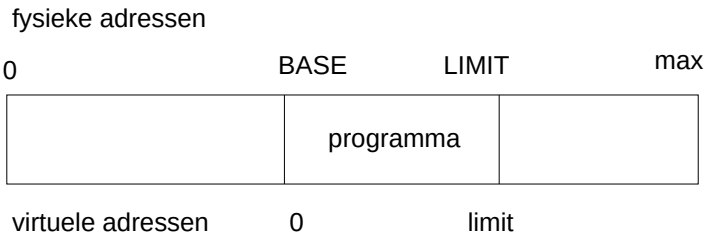
Het O.S. zelf zal waarschijnlijk liever fysieke adressen gebruiken omdat het in principe overal bij moet kunnen en zelf toch niet verschoven wordt. Dit kan bijv. simpel gekoppeld worden aan de system mode: in system mode zijn dan virtuele en fysieke adressen gelijk.

We hebben in de vorige sectie al gezien dat het nuttig kan zijn als een deel van het geheugen be-

schermd kan worden tegen toegang door een bepaald programma. De allereenvoudigste manier om dit te realiseren is om in de CPU twee registers te hebben die aangeven wat de onder- en bovengrenzen zijn van de adressen die een programma in user-mode mag gebruiken. Het ondergrens-register is dan de BASE en het bovengrens-register (dat eigenlijk de grootte van het geheugendeel van het user-mode programma aangeeft) de LIMIT. Wanneer een programma een adres gebruikt wat buiten deze grenzen ligt, of het nu is om een variabele te gebruiken of een instructie op te halen dan wordt er een speciale trap gegenereerd[3]. De betreffende registers moeten natuurlijk weer door het O.S. in system mode ingevuld worden. Wanneer een programma meer geheugen wil hebben kan het d.m.v. een system call een verzoek daartoe doen.

In de rest van deze sectie zullen we dit simpele idee steeds verder uitwerken tot een uitgebreid systeem van *geheugenbeheer* of *memory management*.

Memory management:
Een systeem van hardware en software dat virtuele adressen koppelt aan fysieke adressen.



1.9.1 Segmentering

Het komt bij verschillende computersystemen nogal eens voor dat de hoeveelheid geheugenruimte die aanwezig is, niet in overeenstemming is met de hoeveelheid geheugen die programma's zouden willen of kunnen gebruiken. Er zijn (vooral in het verleden) twee verschillende vormen van deze *mismatch*. En daarom zijn er ook verschillende oplossingen hoewel ze op elkaar lijken.

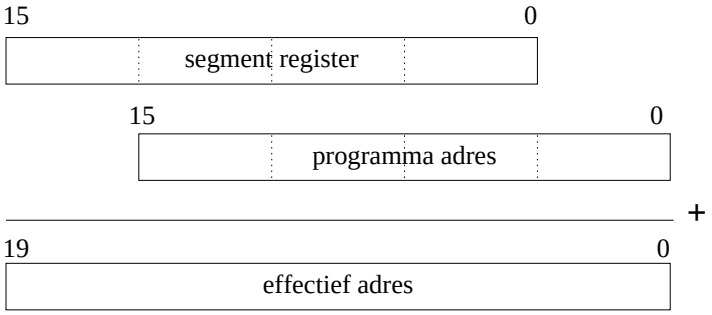
Onder de *adresruimte* van een programma verstaan we de verzameling mogelijke adressen die in het programma gebruikt zouden kunnen worden. Op een 16-bits CPU is deze adresruimte in principe $2^{16} = 65536$ geheugenplaatsen, waarbij tegenwoordig een geheugenplaats meestal 1 byte is. Dit is voor gewoon gebruik veel te weinig, en in de computer is bijna altijd meer aanwezig[4]. Dit betekent dat op de een of andere manier de te gebruiken adressen in het programma meer geheugenplaatsen moeten kunnen aanwijzen. Bij een 32-bits CPU is het aantal mogelijkheden $2^{32} = 4294967296$. Dit is meer dan de meeste computers hebben, dus nu zitten we met het omgekeerde probleem: hoe halen we meer uit het aanwezige geheugen.

Het eerste probleem was bijvoorbeeld aanwezig bij de originele IBM-PC en alle klonen daarvan, en het O.S. MS-DOS. De oplossing die de fabrikant Intel van de 80*86 CPU's gekozen had is het gebruik van *segment registers*. In de CPU zijn 4 segment registers aanwezig, één voor instructies, één voor stack referenties, één voor andere data referenties, en een laatste voor bijzondere instructies (bijv. op strings). Wanneer de CPU een geheugenadres nodig heeft, wordt het bijbehorende segment register

[3] Met wat knutselwerk is het zelfs mogelijk deze controle buiten de CPU (op de bus) te doen

[4] Behalve in toepassingen zoals wasmachines.

gebruikt om het adres dat de instructie gebruikt uit te breiden. Het segment register is ook 16 bits, maar het wordt met 16 vermenigvuldigd (4 plaatsen opgeschoven) en het adres uit de instructie wordt er bij geteld. Zo wordt effectief een 20 bits adres verkregen waardoor 1MB geheugen geadresseerd kan worden. Omdat dit op den duur ook niet voldoende was, moesten er daarna weer trucs uitgehaald worden om nog meer geheugen te kunnen benaderen.



Hoewel op deze manier een grotere adresruimte benaderd kan worden dan de adresgrootte van het programma in eerste instantie toestaat, is dit systeem toch niet erg handig. Als we een segmentregister niet veranderen kunnen we met behulp van de adressen in het programma max. 65536 bytes (ook wel 64KB genoemd) adresseren. Zo'n deel van het geheugen wat dan bij één waarde van het segment register hoort, noemen we een segment. Segmenten kunnen dus max. 64KB groot zijn. Als we over deze grens heen gaan, moet het segment register aangepast worden. Als je bijvoorbeeld een array zou hebben dat groter is dan 64KB en je wilt dat element voor element aflopen (bijv. met een *for* statement), dan moet om de zoveel keer het segment register aangepast worden. Dit is een bijzonder onhandige manier van programmeren.

Een voorbeeld van hoe die code eruit zou kunnen zie, volgt hier. We gaan er dan vanuit dat een adres voorgesteld wordt door twee 16-bits woorden, één die als inhoud van een segmentregister gebruikt kan worden, en één die als offset dient. Wanneer we een getal bij het adres moeten optellen, kunnen we dat zo doen:

tel het getal bij de offset op.
als er een carry komt komen we 2^{16} tekort.
dit kunnen we compenseren door bij het segmentdeel 2^{12} op te tellen.

```
    ADD    getal, offset
    BCC    ok
    ADD    4096, segment
ok:
```

Er zijn dan ook compilers die eisen dat een array niet groter dan 64KB wordt, en andere datastructuren ook niet. Dezelfde eis kan gesteld worden voor de hoeveelheid instructies in één functie of procedure. Wanneer aan deze eis niet voldaan kan worden, dan wordt de code die de compiler genereert veel ingewikkelder. Zelfs wanneer wel aan deze eis voldaan is, moet telkens wanneer naar een andere functie gegaan wordt of een ander stack frame benaderd wordt, een segment register aangepast worden. In sommige gevallen is het mogelijk om een groep stackframes of een groep functies als één geheel te behandelen, maar bij het gebruik van pointers is dit weer lastiger.

Compilers voor MS-DOS of voor 16-bits programma's onder MS Windows hebben vaak verschillende "memory models". Dit heeft te maken met de manier waarop de segmentregisters gebruikt worden, en hoe adressen voorgesteld worden. Bij een "small" memory model moet het hele programma in één segment passen. Evenzo de stack, en ook de heap met de globale data. In zo'n programma hoeven er dan geen segmentwisselingen te gebeuren. Pointers kunnen dan ook als 16-bits waarden voorgesteld worden.

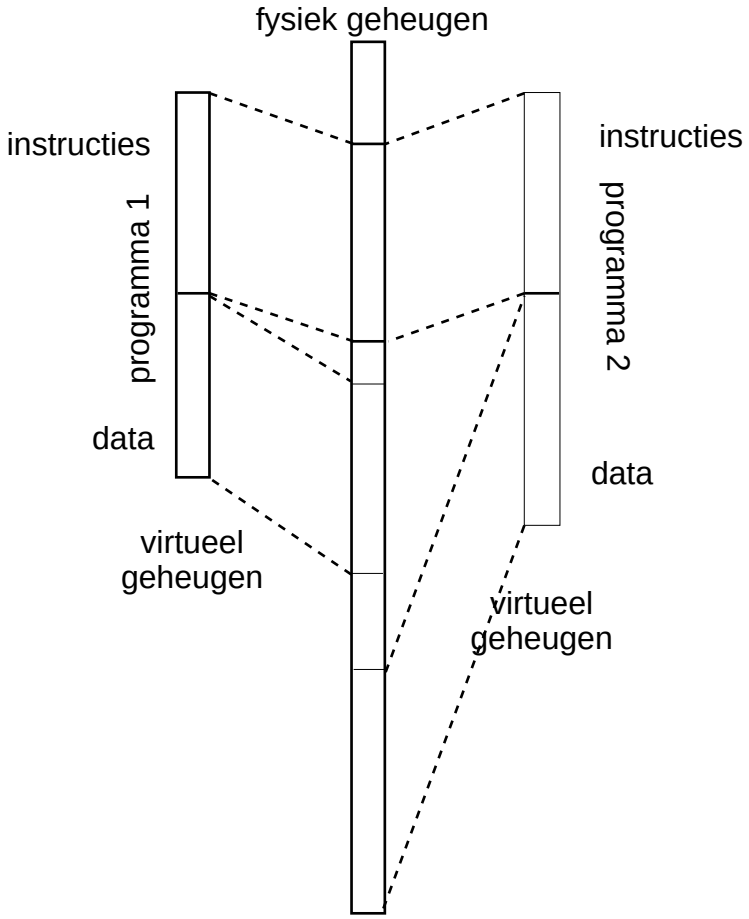
Een "large" memory model heeft dan wel segmentwisselingen nodig, maar elke functie apart past wel in een segment. Evenzo elke variabele (dus vooral structs en arrays). Het benaderen van velden uit een struct en elementen van een array kan dan zonder segmentveranderingen gebeuren. Bij een "huge" memory model zijn er geen beperkingen. Het is i.h.a. niet mogelijk om componenten die met verschillende memory modellen gecompileerd zijn, door elkaar in één programma te gebruiken. In het bijzonder is het dus nodig om algemeen bruikbare bibliotheken in verschillende memory modellen voorradig te hebben.

Segmentering kan ook nuttig zijn in systemen, waarbij er helemaal geen probleem is dat de virtuele adressen kleiner zijn dan de fysieke adressen. In de vorige sectie hebben we het idee gelanceerd dat elk programma door een BASE en LIMIT register aangewezen kan worden. Dit heeft als nadeel dat twee programma's die exact dezelfde instructies gebruiken (bijvoorbeeld twee keer dezelfde editor die voor twee personen aan het werk is) onmogelijk dit stuk geheugen kunnen delen. Als dat wel kon, dan zou dit een besparing in het geheugengebruik opleveren. We kunnen dit probleem simpel oplossen als er twee segmenten voor ieder programma zijn: één voor de instructies, en één voor de data+stack. Wanneer twee keer hetzelfde programma opgestart wordt, kan het instructiedeel één keer in het geheugen geladen worden en in beide programma's kunnen de BASE en LIMIT van het instructiedeel hiernaar verwijzen. Elk programma krijgt wel een eigen stuk geheugen voor de data en stack. Zie figuur 1.6.

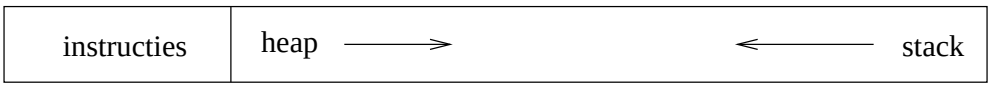
Dit idee kan verder uitgewerkt worden. De meeste programma's hebben een variabele behoefte aan stack- en heapruimte, en het is meestal niet van te voren te voorspellen hoe groot die kunnen worden. De handigste oplossing is om één van beide op een laag adres te laten beginnen, en de andere op een hoog adres, en ze dan naar elkaar te laten groeien. Op deze manier hoeven we niet voor elk van beide een stuk geheugen te reserveren, maar kunnen ze gezamenlijk de ruimte verdelen. Met een oplossing zoals hierboven met twee segmenten, waarbij de stack en heap beide in één segment zitten, moeten we echter de maximaal benodigde ruimte voor dit segment reserveren, omdat we er geen "gat" in kunnen hebben. Door nu de stack en de heap beide een eigen segment te geven, en ze ook op de beschreven manier naar elkaar toe te laten groeien, kunnen we wel de maximale hoeveelheid virtueel geheugen gebruiken, zonder dat er overbodig fysiek geheugen gebruikt wordt. Dit vereist wel dat segmenten zowel naar hogere als naar lagere adressen kunnen groeien. Zie figuur 1.7.

Nog meer segmenten kunnen nuttig zijn als we gegevens tussen twee programma's willen uitwisselen (via een segment dat ze samen delen) en wanneer we niet alleen de instructies van gehele programma's willen delen, maar ook bibliotheken van veel gebruikte functies. Er kan dan een speciaal segment gereserveerd worden voor zo'n bibliotheek, en twee verschillende programma's kunnen dan elk een segment hebben dat in het fysieke geheugen naar hetzelfde deel wijst.

Wanneer een behoorlijk aantal segmenten gewenst is, kan een aantal (meest significante) bits van het adres gereserveerd worden om het segmentnummer aan te geven en de rest voor de offset binnen het segment. Bijvoorbeeld 4 bits voor het segmentnummer (16 segmenten) en 28 bits voor de offset, waardoor een segment maximaal 2^{28} bytes groot kan worden. Als een groter segment nodig is, kunnen twee opeenvolgende segmenten gebruikt worden. Let wel, dat dit een keus is van het ontwerp van de processor! Het segmentnummer wordt dan gebruikt als index in een (hardware) lijst van segmentbe-



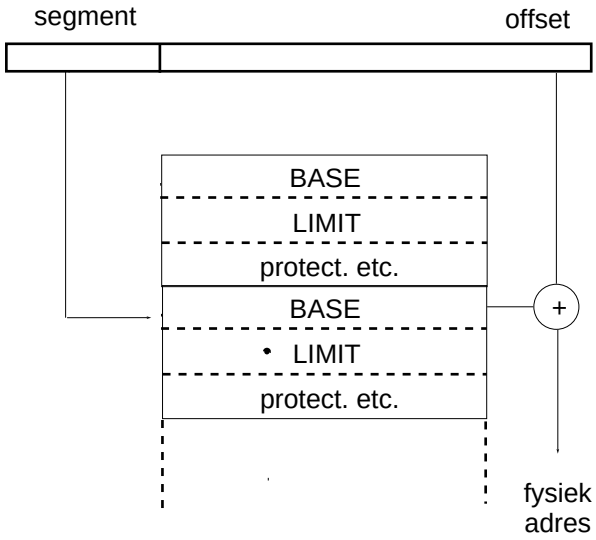
Figuur 1.6: Situatie met 2 segmenten



Figuur 1.7: Heap en stack

schrijvers, die elk het (fysieke) beginadres van het segment bevatten, de lengte van het segment, en verder welke toegangsrechten het programma heeft (lezen, schrijven, executeren).

Bij het gebruik van meerdere segmenten wordt het ook aantrekkelijk om de mogelijkheid te hebben om een segment tijdelijk uit het geheugen te verwijderen en op de schijf op te slaan, bijvoorbeeld wanneer er tekort aan geheugen is. De CPU kan dan een trap genereren als het programma een adres in het segment gebruikt en het O.S. kan dan alsnog het segment in het geheugen laden, en daarna het programma door laten gaan op de onderbroken instructie. Er zal dan waarschijnlijk eerst ruimte gemaakt moeten worden door bijvoorbeeld een ander segment te verwijderen. Segmenten die alleen een kopie waren van iets dat toch al op de schijf aanwezig is (bijvoorbeeld de instructies), kunnen zonder meer tijdelijk verwijderd worden, andere segmenten, waar gewijzigde data in opgeslagen zijn, moeten eerst veilig op een plekje op de schijf opgeborgen worden. De ruimte die hiervoor gebruikt wordt, wordt vaak “swap”ruimte genoemd. Op Windows NT gebeurt dit in de file `pagefile.sys`. Op Windows 95/98 heeft het `win386.swp`. Sommige mensen ‘ontdekken’ deze file op hun computer en willen hem dan weghalen omdat hij zo groot is. Niet doen dus!



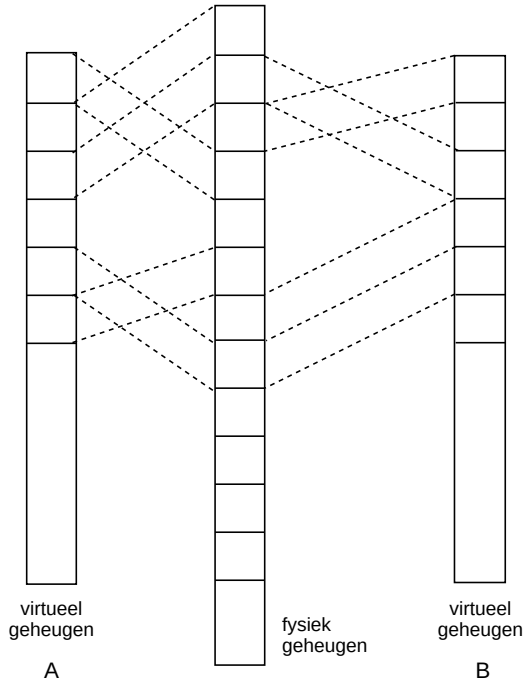
Figuur 1.8: Adresstructuur bij segmentatie

Segmentering:
Een systeem van memory management waarbij het virtuele geheugen opgedeeld wordt in enkele stukken (segmenten) en waarbij elk segment aaneengesloten in het fysieke geheugen geplaatst wordt.

1.9.2 Paginering

Een belangrijk nadeel van segmentatie is dat het beheer van het fysieke geheugen ingewikkeld is. Segmenten kunnen allerlei verschillende groottes hebben, ze kunnen tijdens het gebruik groter en kleiner gemaakt worden. Op een gegeven moment kan dan *fragmentatie* optreden, d.w.z. dat er allerlei kleine stukjes ongebruikt geheugen overblijven, die samen weliswaar genoeg zijn om de behoefte

van een aanvraag te dekken, maar omdat ze niet aaneengesloten zijn, daarvoor niet gebruikt kunnen worden. In dat geval moet er heen en weer geschoven worden om het vrije geheugen “bij elkaar te vegen”. Weliswaar is het segment systeem juist bedoeld om (o.a.) dit probleem voor de programma’s onzichtbaar te maken, maar het O.S. heeft dan toch veel tijd nodig om het geheugen optimaal te kunnen gebruiken. De fragmentatieproblemen verdwijnen, als het geheugen alleen in vaste stukjes verdeeld wordt, die ieder afzonderlijk uitgedeeld kunnen worden. In dat geval noemen we deze stukjes *pagina’s*, en we spreken van paginering i.p.v. segmentering. We kunnen paginering ook zien als een ver doorgevoerde vorm van segmentering, waar nl. het aantal segmenten heel groot geworden is, en de afzonderlijke blokjes vrij klein (4096–16384 is tegenwoordig een gebruikelijke grootte voor een pagina). Elke pagina afzonderlijk heeft dan een afzonderlijke “pointer” naar het fysieke geheugen, en elke pagina kan eigen protectiewaarden hebben(d.w.z. er kan aangegeven worden of deze pagina gelezen en/of geschreven mag worden); tevens kan per pagina beslist worden of deze gedeeld wordt met andere programma’s. Het zal duidelijk zijn dat dit een grote flexibiliteit geeft.



Figuur 1.9: Paginering

Niet alle pagina’s hoeven toegewezen te zijn: er kunnen in de adresruimte “gaten” zitten, bijvoorbeeld tussen de stack en de heap. Ook hoeven niet alle pagina’s in het geheugen aanwezig te zijn. Wanneer er niet genoeg geheugen is, kunnen pagina’s naar en van de schijf getransporteerd worden. De pagina’s worden dan pas van de schijf naar het interne geheugen getransporteerd als ze echt nodig zijn (*demand paging*). In feite is dit de belangrijkste reden waarom systemen gebruik maken van virtueel geheugen en paginering: op deze manier kan een programma virtueel veel groter worden dan het fysieke geheugen toestaat. Bij segmentering geldt dat in ieder geval het geheugen groter moet zijn dan het grootste segment; bij paginering is er niet meer een dergelijke eis. Natuurlijk is het wel zo dat er veel tijd verloren zal gaan met het heen en weer halen van pagina’s tussen het geheugen en de schijf,

wanneer het totale virtuele geheugen veel groter wordt dan het fysieke geheugen. Meestal wordt een grens aangehouden van 2 à 4 maal de hoeveelheid fysiek geheugen voor de swapruimte. Wanneer het heen en weer schuiven van pagina's zoveel tijd in beslag gaat nemen dat er nauwelijks tijd overblijft voor het echte werk, spreken we van "thrashing".

Bij paginering moet er voor elke pagina in de virtuele adresruimte een "pointer" zijn naar het fysieke geheugen. Deze bevat ook extra informatie, zoals de protectie. Deze pointers samen vormen de z.g. *pagetable*. De pagetable entries behoeven niet een compleet 32-bits adres te bevatten omdat de pagina's altijd op een veelvoud van de paginagrootte beginnen. Bij het voorbeeld van een paginagrootte van 4096 bytes zijn de laagste 12 bits dus altijd 0, dus deze hoeven niet opgenomen te worden. In de plaats hiervan worden dan de protecties en andere administratiebits opgenomen. De pagetable kan nogal groot worden, als we bijv. 32 bits adressen hebben en pagina's van 4096 bytes dan heeft de pagetable $2^{32}/4096 = 1048576$ entries, van ieder (waarschijnlijk) 4 bytes, dus ca. 4MB aan pagetables. Dit is een substantieel deel van het geheugen van sommige computers! Wat daarom meestal gedaan wordt is dat de pagetable in stukjes verdeeld wordt, en dat niet alle stukken in het geheugen aanwezig hoeven te zijn. Ook kan de pagetable van een bepaald programma zelf weer in virtueel geheugen gezet worden, waarvoor dan ook weer een pagetable nodig is. In het bovenstaande voorbeeld hoeft deze slechts 1024 entries te bevatten, en kan dus vast in het geheugen genomen worden.

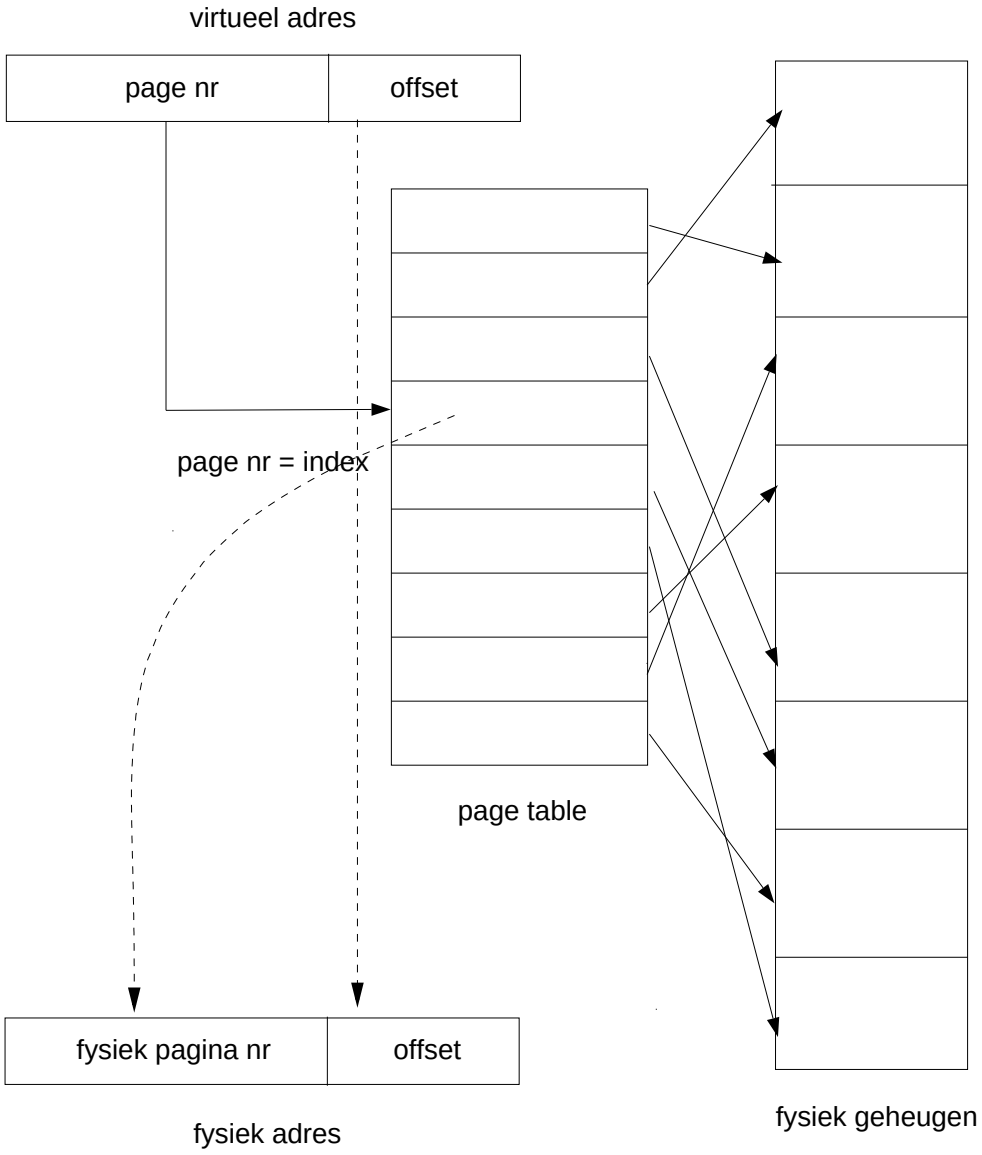
Paginering:
Een systeem van memory management waarbij het virtuele geheugen verdeeld wordt in een groot aantal even grote blokjes, en waarbij elk blokje afzonderlijk in het fysieke geheugen geplaatst kan worden.

Demand paging:
Paginering waarbij pagina's naar behoefte vanuit de harde schijf naar het fysieke geheugen getransporteerd worden. De harde schijf functioneert dan als een uitbreiding van het fysieke geheugen.

Het nadeel van deze constructie is dat voor iedere geheugentoegang er tenminste één en misschien zelfs twee extra geheugenreferenties moeten plaats vinden, nl. om de pagetable entries op te halen. Dit zou de computer ontoelaatbaar vertragen, temeer daar het geheugen i.h.a. langzamer is dan de CPU. Zelfs het gebruik van een cache helpt niet genoeg. Daarom hebben de processoren een apart zeer snel geheugen aan boord waarin de laatst gebruikte pagetable entries opgeslagen worden, zowel het virtuele als het fysieke paginanummer, en de bijbehorende administratie. Deze *translation lookaside buffer* (TLB) is een z.g. associatief geheugen, waar een virtueel pagina vliegensvlug in wordt opgezocht (het vergelijken van het paginanummer met alle entries gebeurt niet sequentieel, maar parallel), en het fysieke paginanummer wordt opgeleverd. Wanneer het virtuele paginanummer niet aanwezig is, wordt de pagetable uit het geheugen gebruikt en de gevonden entry wordt in de TLB gezet. Bij sommige CPU's vindt een trap plaats waardoor het O.S. de pagetable entry kan ophalen en in de TLB zetten.

Omdat verschillende programma's dezelfde virtuele adressen gebruiken, met verschillende vertalingen naar fysieke adressen, moet er iets gebeuren bij het omschakelen van één programma naar een ander, of naar het O.S. zelf. Het is mogelijk om dan de TLB leeg te maken, maar ook kan een nummer van het programma in de TLB opgenomen worden.

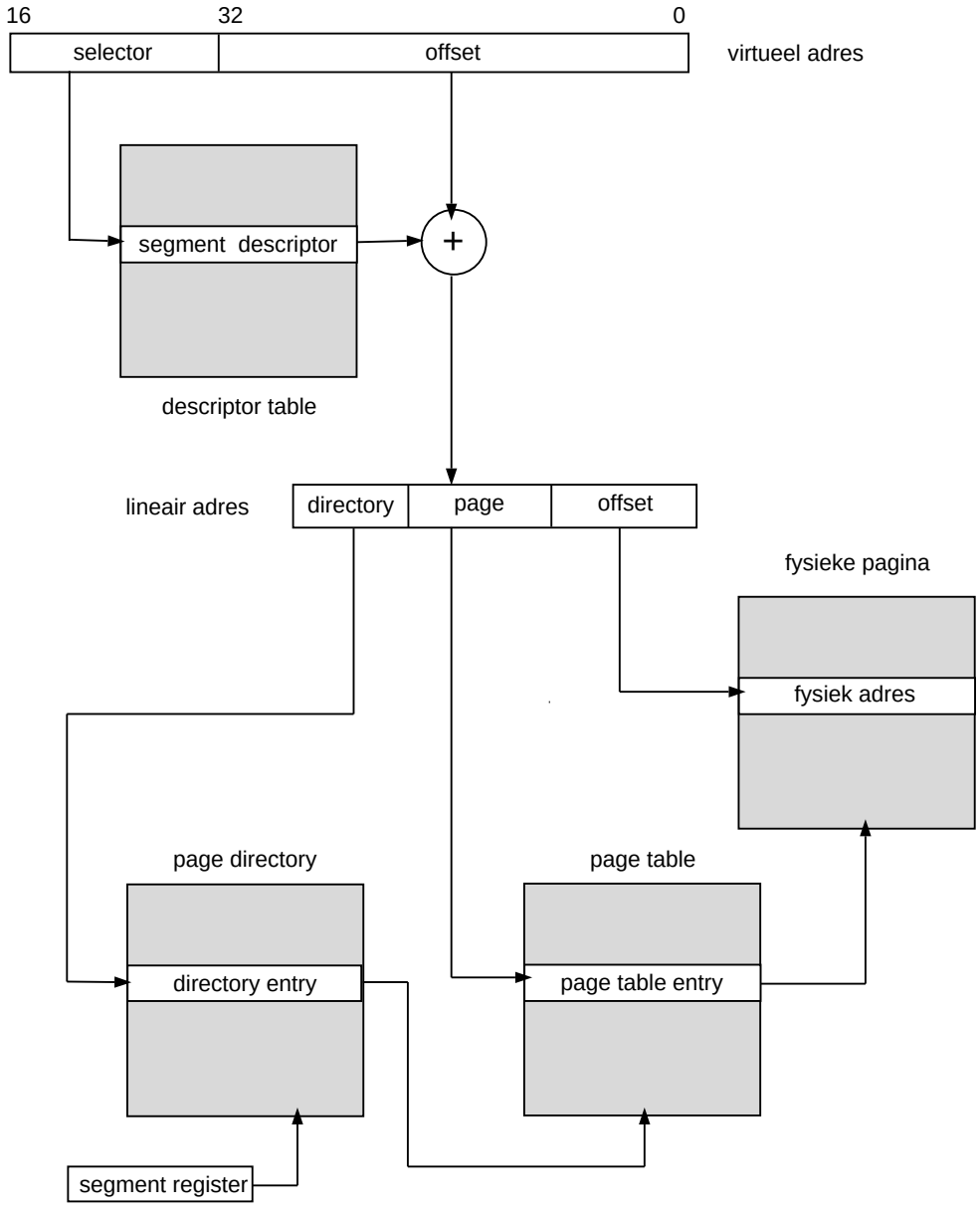
Het is ook nog mogelijk zowel segmentering als paginering te gebruiken. Op de 80386 processor en zijn opvolgers worden bijvoorbeeld een combinatie van segmentering en paginering gebruikt. Er zijn aparte segmentregisters (16 bits) en daarnaast virtuele adressen van 32 bits. Hierdoor wordt een effectief adres van 48 bits verkregen. De segmentregisters bevatten nu niet direct een fysiek adres maar een index in een *segment descriptor table* waarin o.a. de adressen, groottes en administratie van de



Figuur 1.10: Vertaling van virtuele adressen naar fysieke adressen bij paginering

De page table bevat voor elke virtuele pagina het nummer van de pagina in het fysieke geheugen. Het virtuele adres wordt opgesplitst in twee stukken: het paginanummer en de offset. Het paginanummer wordt gebruikt als index in de page table. Het daar gevonden fysieke paginanr wordt gecombineerd met de offset en samen vormt dit het fysieke adres.

segmenten staan. De pagetables zijn in stukken gehakt, die weer door een pagetable directory worden beschreven. De totale adresberekening ziet er dan uit zoals in figuur 1.11:



Figuur 1.11: Adresberekening in de 80386

1.9.3 Copy-on-write

Virtueel geheugen (vooral met paginering) is handig om dezelfde data in twee programma's te gebruiken zonder dat ze toegang hebben tot alle data van elkaar. Alleen die pagina's waarin gemeenschappelijke data staan, hoeven dan lees- en schrijftoegang te hebben voor beide. Hierdoor is het o.a. ook mogelijk om gegevens van een programma naar een ander over te brengen,of van het O.S. naar een programma of omgekeerd, zonder een echte kopie actie te ondernemen. Daartoe worden dan de pagina descriptors van beide programma's op hetzelfde stuk fysieke geheugen gezet. Wanneer een van beide programma's of beide deze ruimte echter later voor iets anders willen gebruiken is er een probleem. Een verandering in het ene programma moet in dit geval niet in het andere zichtbaar zijn. Om dit op te lossen wordt de techniek van *copy-on-write* gebruikt. Dit houdt in dat het O.S. bij het opzetten van de "kopie" actie (die dus niet een echte kopie is), de betreffende pagina's beschermt tegen schrijven voor beide programma's. Wanneer een van de programma's een wijziging wil aanbrengen dan volgt er een exceptie, en op dat moment maakt het O.S. een kopie van de betreffende pagina, en geeft beide programma's een eigen exemplaar, waarbij dus de pagetable entry van het ene naar het oorspronkelijke exemplaar verwijst en die van de andere naar het andere exemplaar. Deze nieuwe pagetable entries kunnen natuurlijk wel schrijfpermissie hebben. Hoewel dit extra werk kost blijkt in de praktijk dat het voorkomen van kopiëeracties toch een gunstig effect heeft op de prestaties van een systeem.

Copy-on-write:
Een optimalisatie van het kopiëren van geheugen waarbij de kopie pas gemaakt wordt op het moment dat een van de twee kopieën gewijzigd wordt. Maakt gebruik van de memory management hardware.

1.9.4 Page faults

We besluiten dit deel met de beschrijving van wat er gebeurt als er een *page fault* optreedt, d.w.z. wanneer een programma of het O.S. zelf een adres refereert waarvan de pagetable entry niet naar een fysieke pagina wijst.

Page fault:
Een trap die optreedt als er een virtueel geheugenadres gebruikt wordt dat niet gekoppeld is aan een fysiek adres.

1. Als het virtuele paginanummer niet in de TLB buffer aanwezig is, laadt het O.S. de pagetable entry in de TLB (tenzij dit al automatisch gebeurt). Dit gebeurt alleen als de pagina fysiek in het geheugen aanwezig is, anders heeft dit geen zin.

2. Als de pagina niet in het geheugen is, maar op de schijf, dan wordt er een vrije pagina opgezocht (of een pagina vrij gemaakt), en er wordt een lees actie van de schijf gestart. Er wordt een merkteken gezet dat deze pagina bezig is geladen te worden, zodat een volgende pagefault dit niet opnieuw zal doen. Het betreffende programma moet tijdelijk gestopt worden, omdat het alleen verder kan als de pagina ingelezen is. Na het inlezen wordt de pagetable entry bijgewerkt en in de TLB gezet.

3. Als de pagina niet op de schijf is (bijvoorbeeld omdat de pagina nodig is voor uitbreiding van de heap of stack, dan wordt gewoon een vrije pagina opgezocht, en eventueel al vast schijfruimte gereserveerd om deze later weg te kunnen schrijven.

4. Het O.S. doet er goed aan om een voorraadje beschikbare pagina's aan te houden omdat anders bij het inladen van een pagina eerst gewacht moet worden tot er een andere weggeschreven is. Dit zou het systeem teveel vertragen.

5. Een pagina die van de schijf geladen is, en niet is gewijzigd hoeft niet weggeschreven te worden, omdat de schijf nog een kopie bevat. Deze komen dus het eerst in aanmerking om hergebruikt te worden. Pagina's die wel gewijzigd zijn, moeten eerst weggeschreven worden voor ze hergebruikt mogen worden. We noemen deze *dirty*. (Dirty betekent dat de waarde in het interne geheugen niet overeenkomt met de waarde op de schijf.) Na het wegschrijven is de pagina niet meer dirty. Tijdens het wegschrijven moet natuurlijk ook aangemerkt worden dat de pagina bezig is weggeschreven te worden, om te voorkomen dat dit twee keer gebeurt.

6. Pagina's van programma's die gestopt zijn, of delen van het geheugen die vrijgegeven zijn hoeven niet meer weggeschreven te worden, maar zijn onmiddellijk voor hergebruik beschikbaar.

1.10 Opgaven

1. Hoeveel adressen zijn er mogelijk wanneer er 24 bits gebruikt worden? En hoeveel bij 64 bits?

2. Beargumenteer welke van de volgende apparaten het beste op een snelle controller aangesloten kunnen worden zoals in figuur 1.1 en voor welke een langzame ook wel goed is.

   (a) netwerkkaart
   (b) muis
   (c) toetsenbord
   (d) geluidskaart
   (e) DVD speler/brander

3. Stel we hebben een geheugen met een toegangstijd van 100 nsec. Dat wil zeggen dat het 100 nsec tijd kost om een woord uit het geheugen te halen (1 nsec = 10^{-9} sec). We gebruiken een cache dat een toegangstijd van 10 nsec. De *hit rate* van het cache is 90%, dat wil zeggen dat 90% van elk opgevraagd woord in het cache te vinden is. Hoeveel tijd kost het gemiddeld om een woord op te halen?

4. Schrijf een programmaatje zoals in sectie 1.2 voor het volgende probleem: Je hebt een pointer naar een rij *int*s. Bereken de som van alle achtereenvolgende *int*s tot je er een tegenkomt die < 0 is (deze wordt niet meegeteld).

5. In een 32-bits computer willen we segmenten hebben. Als we 64 segmenten willen hebben, hoeveel bits moet het segmentnummer dan hebben en hoe groot kan elk segment maximaal zijn?

6. Bij een simpel pagineringssysteem zoals in figuur 1.10 (32-bits adressen, 16kB pagina's): hoe groot kan de paginatabel maximaal worden? Hoeveel % van het geheugen is dit?

7. Hetzelfde als de vorige opgave, maar nu met 64-bits adressen en een maximaal geheugen van 64GB.

Hoofdstuk 2

Assemblerprogrammeren

In deze sectie geven we de beschrijving van een eenvoudige assemblertaal. Omdat de meeste gangbare CPU's een nogal ingewikkelde instructiecode hebben, gebruiken we niet een van de bestaande CPU's, maar een zelfbedachte hypothetische CPU. We noemen dit een virtuele machine (VM). De VM die we gebruiken is bedacht door Atze Dijkstra (zie het collegedictaat IPT voor meer informatie). Het is echter mogelijk om deze virtuele machine echt te gebruiken: er is een Java programma dat zich net gedraagt als deze 'CPU'; we noemen dit programma een emulator. We zeggen dan ook dat dit programma de CPU emuleert. Dat wil zeggen: het programma doet de CPU na.[1]

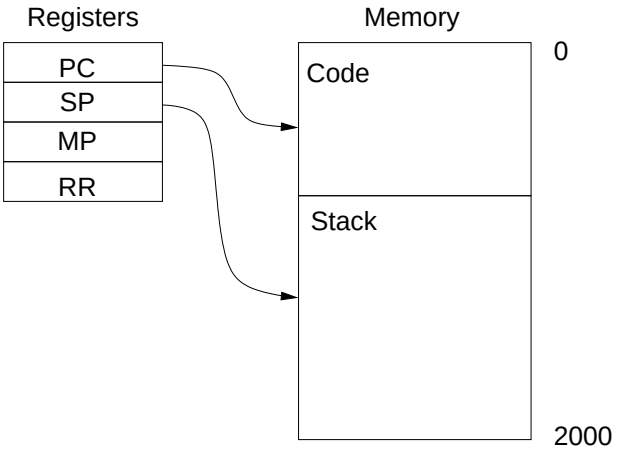
Onze virtuele CPU heet SSM (Simple Stack Machine) en is te downloaden van `http://www.cs.uu.nl/~atze/SSM/`. De naam 'stack machine' geeft aan dat de belangrijkste operaties van deze machine altijd op de stack plaatsvinden: dit maakt deze machine simpeler dan andere CPU's. In een programma moeten verschillende operaties uitgevoerd worden, zoals optellen, aftrekken en vermenigvuldigen. Op pagina 18 hebben we een voorbeeld van zo'n operatie (ADD) gegeven. Een operatie heeft in het algemeen één of meer operanden waarop de operatie uitgevoerd wordt: bijvoorbeeld de getallen die opgeteld moeten worden. Verder moet het resultaat ergens naar toe. In ingewikkelde CPU's kan voor de operanden en het resultaat een keuze gemaakt worden uit de registers en het geheugen. Dit heeft tot gevolg dat er veel combinaties zijn. Dit maakt het ingewikkeld. Door nu de keus te maken dat alle operanden en het resultaat op de stack geplaatst worden wordt het aantal keuzes drastisch beperkt en wordt de structuur dus eenvoudiger. Overigens zullen we in dit hoofdstuk niet alle mogelijkheden van de SSM behandelen.

De SSM heeft voor een paar speciale toepassingen wel registers, maar verder doen we dus alles op de stack. De CPU ziet er dan als volgt uit (zie figuur 2.1):

Het geheugen bestaat in deze CPU uit een reeks van 32-bit woorden. In een echte CPU zijn meestal afzonderlijke bytes te adresseren, maar voor de eenvoud is er hier gekozen voor het adresseren van woorden. Hoe zo'n woord geïnterpreteerd wordt hangt af van de instructie die het woord gebruikt. Gebruikelijke interpretaties zijn als 32-bit geheel getal (*int*) of als adres.

Gewoonlijk wordt in het lage gedeelte van het geheugen (vanaf adres 0) het programma geladen. Het programma bestaat ook uit 32-bits woorden, maar we zullen hier niet ingaan op de gebruikte codering.

[1] Het is ook mogelijk om echte CPU's te emuleren; op die manier is het vaak gemakkelijker om het gedrag van een CPU te bestuderen.



Figuur 2.1: De structuur van SSM

Achter het programma begint de stack, waarbij deze groeit in de richting van hogere adressen[2]. Verder zijn er acht register waarvan er drie een vaste betekenis hebben:

De *program counter* PC (=R0), die wijst naar de volgende instructie die uitgevoerd moet worden. Tijdens het verwerken van een instructie wordt dit register opgehoogd. Er zijn instructies voor het springen naar een andere locatie, deze veranderen gewoon de waarde van de PC. Bij het aanroepen van een methode (functie) wordt de waarde van de PC op de stack geplaatst zodat we aan het einde van de methode kunnen terugkeren naar de instructie achter de methodeaanroep. Na het opbergen van de PC op de stack krijgt de PC dan de nieuwe waarde, nl. het adres van de eerste instructie van de methode.

De *stackpointer* SP (=R1) wijst naar de top van de stack, dat wil zeggen het hoogste adres van de stack dat in gebruik is. Omdat bijna alle bewerkingen op de stack plaatsvinden gaat de stackpointer voortdurend op en neer. Dit gaat voor het grootste gedeelte automatisch: het programma hoeft niet expliciet de waarde van de stackpointer bij te werken, want dit gebeurt door de opdrachten zelf. Alleen als we bijvoorbeeld extra ruimte nodig hebben voor variabelen, of als we die ruimte weer willen vrijgeven manipuleren we de stackpointer expliciet.

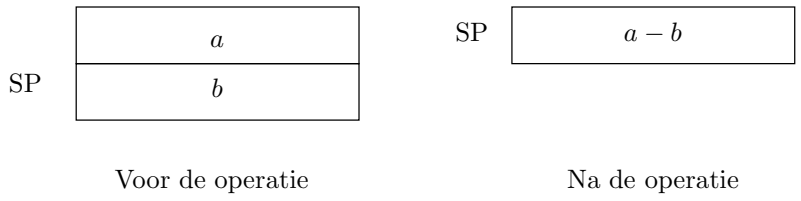
De *markpointer* MP (=R2) wijst naar het stuk stack waar de locale variabelen van een methode zijn opgeslagen. Ook de parameters van een methode kunnen via de markpointer gevonden worden. Hoewel het mogelijk is om zonder de markpointer te werken, maakt het gebruik ervan programma's gemakkelijker. Er zijn dan ook speciale instructies om de variabelen via de markpointer te benaderen.

We gebruiken verderop nog een conventie (gewoonte) om het resultaat van een methode in register R3 te bewaren. Dit noemen we dan het resultaatregister (RR). Dit is echter niet iets dat in de SSM ingebouwd is: we hadden net zo goed een ander register kunnen gebruiken. Let op dat dit alleen werkt als het resultaat in een woord past, maar we zullen geen andere voorbeelden gebruiken.

[2] In veel computers wordt de stack in de andere richting gebruikt maar dat is niet essentieel.

2.1 Rekenkundige instructies

In deze sectie geven we voorbeelden van de rekenkundige instructies (optellen, aftrekken, vermenigvuldigen etc.). De SSM werkt alleen met gehele getallen (*int*), dus niet met floats. De operaties werken altijd op de bovenste twee woorden op de stack en vervangen deze door het resultaat. Bij operaties die niet-commutatief zijn zoals aftrekken ($a - b$ is niet hetzefde als $b - a$) is het van belang te weten hoe de volgorde van de operanden is. Zie figuur 2.2 voor de werking van de sub instructie als voorbeeld. In tabel 2.1 staat een lijst van de instructies die op dezelfde manier werken. De instructies eq, ne, lt,



Figuur 2.2: De werking van de sub instructie

gt, le en ge zetten een boolean op de stack, waarbij false door 0 wordt weergegeven en true door een woord met allemaal 1-bits. De instructie neg is verschillend, omdat deze maar één operand heeft en deze vervangt door het negatieve ervan (dus bijv. -3 wordt vervangen door 3). De grootte van de stack verandert bij deze instructie dus niet.

instructie	operatie
add	$a + b$
sub	$a - b$
mul	$a \times b$
div	a/b
mod	$a\%b$
neg	$-a$
eq	$a == b$
ne	$a != b$
lt	$a < b$
gt	$a > b$
le	$a \leq b$
ge	$a \geq b$

Tabel 2.1: Rekenkundige operaties

2.2 Push en Pop operaties

We moeten natuurlijk ook waarden op de stack kunnen krijgen en waarden van de stack ergens anders kunnen opbergen. Het op de stack zetten noemen we *push*, het ervan afhalen noemen we *pop*. We geven in deze sectie de simpelste push en pop operaties.

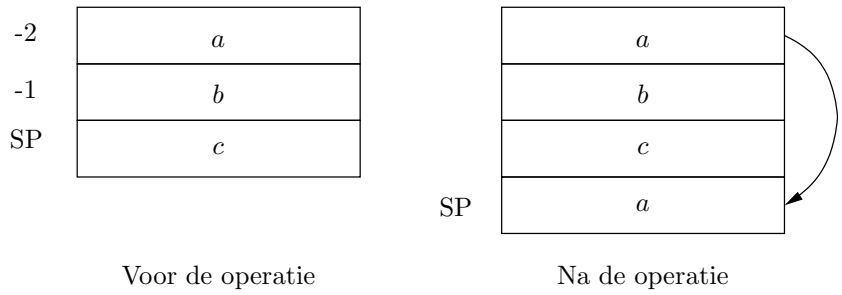
De push instructies voegen één woord toe aan de stack, dus na afloop is de stackpointer één plaats opgeschoven, zie figuur 2.3 voor een voorbeeld. De waarde kan een constante zijn, uit een register

komen of van een variabele in de stack. Zie tabel 2.2 voor een overzicht. ld staat voor ‘load’, r voor ‘register’, s voor ‘stack’ en l voor ‘local’. De operaties die met de markpointer werken komen later aan bod.

instructie	nieuwe waarde op de stack
ldc X	constante X
ldr R_i	inhoud van register R_i
lds n	inhoud van het woord n t.o.v. SP
ldl n	inhoud van het woord n t.o.v. MP

Tabel 2.2: Push instructies

De instructie lds n verdient enige toelichting: n is hier een getal (meestal negatief). Deze instructie neemt het woord dat n geheugenplaatsen verwijderd staat van de SP (zoals die voor de instructie was) en pusht dat op de stack. Dus als $n = 0$ dan wordt het woord op de top van de stack gedupliceerd, als $n = -1$ dan wordt het daaronder liggende woord als het ware over het topelement heen getild en daar gecopieerd.[3] De plaats waar het woord vandaan komt verandert niet.



Figuur 2.3: Werking van de instructie lds -2.

De *pop* instructies werken de andere kant op, alleen hebben we natuurlijk geen pop-instructie met een constante. Zie tabel 2.3. De afkorting st staat voor ‘store’. Let erop dat bij sts de stackpointer nu één hoger is dan bij de overeenkomstige lds instructie zodat we de bijbehorende n één minder moeten nemen. We zullen dat in het volgende voorbeeld duidelijk zien.

instructie	waar de waarde naar toe gaat
str R_i	inhoud van register R_i
sts n	inhoud van het woord n t.o.v. SP
stl n	inhoud van het woord n t.o.v. MP

Tabel 2.3: Pop instructies

[3] De n wordt ook de *offset* van het geheugenwoord ten opzichte van de SP genoemd.

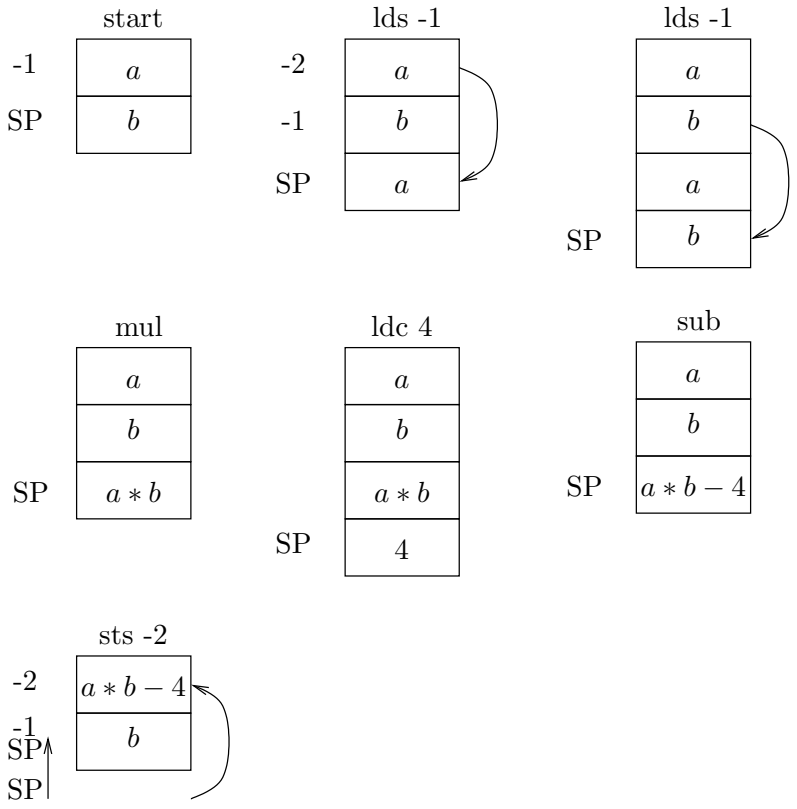
2.3 Een simpel voorbeeld.

We willen het volgende stukje Java in assemblertaal omzetten.

```
int a;
int b;
...
a = a*b-4;
```

We veronderstellen dat *a* en *b* al op de stack staan en dat de SP naar b wijst. Hoe *a* en *b* aan hun waarde gekomen zijn laten we nu even in het midden.We moeten nu de operanden van de operatie (*a*, *b*, en 4) op de stack zetten en de operaties erop uitvoeren. Daarna bergen we het resultaat weer in *a* op. Het volgende stuk assemblercode doet dit. In figuur 2.4 zien we de stack in de opeenvolgende stadia.

```
lds    -1
lds    -1
mul
ldc     4
sub
sts    -2
```

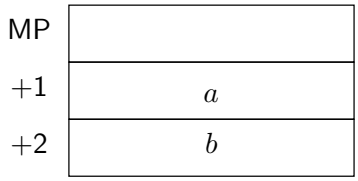


Figuur 2.4: Stack verloop

2.4 De markpointer

In het vorige voorbeeld kunnen we zien dat het adresseren van variabelen via de stackpointer vervelend werk is. Omdat de stackpointer voortdurend verandert, moet je de offset voortdurend opnieuw uitrekenen en een foutje is zo gemaakt. Al je bijvoorbeeld dezelfde berekening ($a * b - 4$) zou willen doen terwijl er al andere dingen op de stack staan dan moet je ook andere offsets gebruiken.

Hiervoor komt de *markpointer* (MP) goed van pas. De MP wijst naar het stuk van de stack waar de locale variabelen van de methode staan. Om redenen die later duidelijk zullen worden begint de eerste locale variabele van de methode in het woord volgend op dat waar de MP naar wijst. Zie figuur 2.5.



Figuur 2.5: De markpointer en twee locale variabelen.

De code uit het vorige voorbeeld kunnen we nu herschrijven als volgt, waarbij het niet meer uitmaakt waar de SP staat:

```
ldl     1
ldl     2
mul
ldc     4
sub
stl     1
```

2.5 Besturingsinstructies

Tot nu toe hebben we alleen stukjes programma geschreven waar de instructies uitgevoerd worden in dezelfde volgorde als waarin ze in het programma staan. Als je echter constructies gebruikt als if-then-else of while of een methodeaanroep dan wordt deze 'natuurlijke' volgorde doorbroken. De instructies in deze sectie hebben alle tot doel dit mogelijk te maken. Deze instructies worden *besturingsopdrachten* of *sprongopdrachten* genoemd. Ze staan in tabel 2.4. De afkorting br staat voor 'branch', t staat voor 'true', f voor 'false' en a voor 'always' of 'altijd'.

Al deze instructies (behalve ret en halt) hebben als parameter een adres van de instructie waar het programma zich voortzet. Om te voorkomen dat we deze adressen zelf moeten uitrekenen heeft de assembler de mogelijkheid om zogenaamde *labels* te gebruiken. Deze zetten we voor de instructie waar we naar toe willen gaan en in de instructie die ernaar toe gaat gebruiken we de label in plaats van het adres.

Om te zien hoe deze instructies gebruikt worden nemen we een simpele if opdracht uit Java als voorbeeld:

```
if (b < 0)
    a = 5;
```

instructie	springconditie
brt	top van stack is true
brf	top van stack is false
bra	altijd
bsr	altijd – zie tekst
ret	zie tekst
halt	stopt SSM

Tabel 2.4: Besturingsinstructies

We gaan uit van de situatie uit figuur 2.5. Wat we willen doen is de code voor `a = 5` overslaan als `b` false is. Hiervoor gebruiken we de brf instructie. Deze neemt het element op de top van de stack en als het false is springt hij naar de label. Als het true is dan gaan we gewoon door met de volgende instructie. De boolean wordt in beide gevallen van de stack verwijderd. Dan blijft nog over het probleem om de juiste boolean op de stack te krijgen. Dit doen we door eerst *b* en 0 op de stack te zetten en dan de instructie lt uit te voeren

```
        ldl    2     // b
        ldc    0
        lt
        brf    else
        ldc    5
        stl    1     // a
else:
```

Als we ook een else gedeelte gebruiken moeten we de code iets uitbreiden: Na de code van het then gedeelte moeten we over het else deel heenspringen. Dit doen we met de bra instructie. Als we bovenstaand voorbeeld als volgt uitbreiden:

```
    if (b < 0)
        a = 5;
    else
        a = -3;
```

dan wordt de code:

```
       ldl    2     // b
       ldc    0
       lt
       brf    else
       ldc    5
       stl    1     // a
       bra    einde
else:  ldc   -3
       stl    1     // a
einde:
```

Tenslotte hebben we nog de bsr (branch to subroutine) instructie die bedoeld is om methodes (functies) aan te roepen. In onze voorbeelden gebruiken we alleen statische methoden; het gebruik van niet-statische methoden zou te ver voeren. Bij de aanroep van een methode moet de PC veranderd worden, namelijk naar het begin van de methode. Maar na afloop van de methode (bij de expliciete of impliciete return) moet weer teruggegaan worden naar het stuk programma waar we vandaan komen. Aangezien we in een methode weer een andere methode kunnen aanroepen, en daarin weer een etc. kunnen we de oude PC niet op een vaste plaats opbergen. De aangewezen plaats hiervoor is natuurlijk: de stack! Dit is precies wat de instructie bsr doet: eerst wordt de waarde van de (opgehoogde) PC op de stack gepusht, daarna krijgt de PC de waarde van de parameter, waardoor het programma verder gaat met de aangeroepen methode. De tegenhanger van bsr is de ret instructie. Deze heeft geen label bij zich maar popt het adres van de top van de stack en springt daar naar toe (dus zet dat in de PC).[4]

Het aanroepen van een eenvoudige methode zou bijvoorbeeld als volgt kunnen gebeuren:

```
       bsr    methode
       .....           // hier gaan we verder na de methode

methode:
       .....           // instructies van de methode
       ret
```

2.6 Methoden met parameters

We gaan nu kijken naar methodes met parameters, en die eventueel ook een resultaat teruggeven. Bijvoorbeeld de volgende:

```
    int sumsquare (int a, int b)
    {
        int x;
        x = a*a + b*b;
        return x;
    }
```

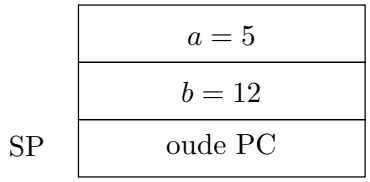
(De x is strict genomen niet nodig maar die is er voor de discussie.) We moeten nu afspreken hoe we de parameters doorgeven en hoe we het resultaat teruggeven aan de aanroeper. De SSM dwingt ons niet tot een bepaalde oplossing dus we hebben hierin vrijheid. We zouden ervoor kunnen kiezen om de parameters in registers door te geven. Het nadeel is dat er maar een beperkt aantal registers is, dus als je veel parameters hebt lukt dat niet. Een tweede nadeel is dat als de aangeroepen methode weer een andere methode aanroept dat de registers vrijgemaakt moeten worden: ze kunt ze bijvoorbeeld op de stack dumpen. Dus is het handiger om de parameters gelijk maar op de stack te zetten. Dit gebeurt natuurlijk voordat de methode aangeroepen wordt.

We zouden de methode sumsquares kunnen aanroepen met als parameters a=5 en b=12 met de volgende code:

[4] In feite zou hetzelfde bereikt kunnen worden met de instructie str PC.

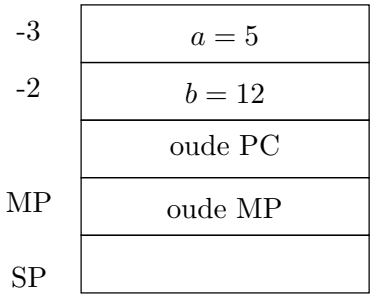
```
ldc          5    // a
ldc         12    // b
bsr sumsquare
```

De stack ziet er dan als volgt uit (figuur 2.6):



Figuur 2.6: Stack met parameters en terugkeeradres

In de methode moeten we nu ruimte reserveren voor de variabele *x* en de markpointer zetten zodat we de variabele *x* en de parameters *a* en *b* gemakkelijk kunnen adresseren. Ruimte reserveren kan eenvoudig door de stackpointer op te hogen (hiervoor is de instructie ajs (adjust stack) beschikbaar. Deze heeft als argument de hoeveelheid woorden die gereserveerd worden. Als we de markpointer zetten moeten we echter opletten, want de waarde die de markpointer heeft voor we de methode ingaan kan ook van belang zijn (nl. voor de methode van waaruit sumsquare wordt aangeroepen). We moeten dus vóór we de markpointer zetten, eerst de oude markpointer opbergen, en het zal intussen wel duidelijk zijn dat de beste plaats om hem op te bergen de stack is. Omdat deze combinatie (markpointer opbergen, nieuwe markpointer zetten, ruimte op de stack reserveren voor locale variabelen door de stackpointer een nieuwe waarde te geven) zo vaak voorkomt, is er een aparte instructie die dit in een keer doet: link *n*, waarbij *n* de ruimte voor de locale variabelen is, in ons geval dus 1. De stack ziet er dan na de link opdracht als volgt uit (figuur 2.7). De offsets zijn ten opzichte van de MP.



Figuur 2.7: Stack na de link instructie.

De link instructie heeft een logische tegenhanger, de unlink *n* instructie die het tegenovergestelde doet. De stackpointer wordt met *n* afgelaagd, en het topelement wordt naar de MP gepopt. Hierdoor zijn we weer terug in de situatie van figuur 2.6.

Het laatste probleem betreft de vraag hoe we het resultaat van de methode (als het niet void is) teruggeven aan de aanroeper. Er zijn in principe twee manieren: via de stack of via een register. Via de stack is lastig, want het terugkeeradres zit in de weg. Daarom doen we het via een register. Hiervoor gebruiken we het RR (=R3) register. De aanroeper kan het antwoord dan naar zijn uiteindelijke bestemming transporteren (bijvoorbeeld naar een variabele of naar de stack als parameter voor een volgende aanroep).

De methode kan nu terugkeren naar zijn aanroeper met de ret instructie. Het enige dat dan nog op de stack staat zijn de parameters *a* en *b*. Het gemakkelijkst is om deze er door de aanroeper vanaf te laten halen en dit kan door gewoon de stackpointer af te lagen: ajs -2. Het is in principe ook mogelijk om het opruimen van de parameters ook door de aangeroepen methode te laten doen, maar dan moet het terugkeeradres naar beneden geschoven worden, omdat anders de ret instructie niet meer werkt.

We geven nu het totale voorbeeld waarbij het resultaat van de aanroep sumsquares(5, 12) in een variabele gezet wordt (waarbij we aannemen dat deze op offset 1 ten opzichte van de markpointer staat).

```
    ldc         5    // a
    ldc        12    // b
    bsr sumsquare
    ajs        -2    // verwijder parameters
    ldr        RR    // resultaat op de stack
    stl         1    // berg het op in een variabele
    ....
sumsquare:
    link        1    // 1 locale variabele
    ldl        -3    // a
    ldl        -3    // a
    mul              // a*a
    ldl        -2    // b
    ldl        -2    // b
    mul              // b*b
    add              // a*a + b*b
    stl         1    // x
    ldl         1    // x
    str        RR    // berg op in RR
    unlink      1
    ret
```

instructie	actie
link *n*	zet MP, reserveer locale variabelen
unlink *n*	omgekeerde van link *n*
ajs *n*	SP = SP+*n*

Tabel 2.5: Extra instructies

2.7 Opgaven

1. Schrijf de SSM code voor de methode abs die de absolute waarde van een parameter x uitrekent en teruggeeft als resultaat. D.w.z. x als $x \geq 0$ en $-x$ als $x < 0$. Schrijf er ook een methode main bij die abs uitrekent voor zowel een positief als een negatief getal.

2. Geef de SSM code voor het volgende Java programma:

```
meth(int a, int b, int c)
{
    int d, x;
    d = b*b - 4*a*c;   // checkpoint
    if (d<0)
       return 0;
    x = (-b + sqrt(d))/(2*a);
    return x;
}
```

Er mag aangenomen worden dat de code voor sqrt ergens in het geheugen aanwezig is.

Teken ook de stack op het checkpoint met de SP, MP, de parameters en de locale variabelen.

3. De link en unlink instructies zijn strikt genomen niet nodig omdat hetzelfde effect ook met andere instructies bereikt kan worden. Geef voor beide een implementatie (een reeks instructies die hetzelfde doen), natuurlijk zonder link en unlink erin.

4. In de tekst is genoemd dat het resultaat van een methode ook via de stack teruggegeven zou kunnen worden. Verander het voorbeeld van sumsquares uit de tekst zodat het op die manier gebeurt. Geef de code voor zowel de aanroep als de methode.

5. In de tekst is genoemd dat het weghalen van de parameters ook in de aangeroepen methode kan gebeuren. Verander het voorbeeld van sumsquares uit de tekst zodat het op die manier gebeurt. Geef de code voor zowel de aanroep als de methode.

instructie	pop	push	operatie
add	2	1	$a+b$
sub	2	1	$a-b$
mul	2	1	$a \times b$
div	2	1	a/b
mod	2	1	$a\%b$
neg	1	1	$-a$
eq	2	1	$a == b$
ne	2	1	$a != b$
lt	2	1	$a < b$
gt	2	1	$a > b$
le	2	1	$a \leq b$
ge	2	1	$a \geq b$
ldc X	0	1	load constante X
ldr R_i	0	1	load register R_i
lds n	0	1	load woord n n t.o.v. SP
ldl n	0	1	load woord n n t.o.v. MP
str R_i	1	0	store register R_i
sts n	1	0	store woord n t.o.v. SP
stl n	1	0	store woord n n t.o.v. MP
brt	1	0	spring als b true is
brf	1	0	spring als b false is
bra	0	0	spring altijd
bsr	0	1	branch to subroutine
ret	1	0	keer terug na methodeaanroep
halt	0	0	stopt SSM
link n	O	n + 1	zet MP, reserveer locale variabelen
unlink n	n + 1	0	omgekeerde van link n
ajs n	*	*	SP = SP+n

Tabel 2.6: Overzicht van alle SSM instructies

Hoofdstuk 3

Operating Systems

Het Operating System (nederlands: besturingssysteem of bedrijfssysteem) bevat die software die standaard op een computersysteem aanwezig is, en die het mogelijk maakt zonder veel moeite gebruikersprogramma's uit te voeren. Enkele taken van een O.S. zijn:

- Het besturen van randapparaten, zoals printers, toetsenbord, harde schijven etc.
- Het bijhouden van de tijd.
- De gebruiker de mogelijkheid geven om programma's op te starten en andere taken uit te voeren, bijv. d.m.v. muis-kliks of in te typen commando's.
- Het beheren van bestanden (files).
- Het uit elkaar houden van verschillende gebruikers zodat ze elkaars informatie niet kunnen zien en/of veranderen.
- Het verzorgen van verbindingen met andere computers via een netwerk.
- Het leveren van allerlei standaardprogramma's zoals editors en compilers.

Niet elk O.S zal àl deze taken uitvoeren, en sommige O.S's zullen meer taken uitvoeren. Sommige mensen zullen het leveren van standaardprogramma's (zoals editors, tekenprogramma's, compilers) niet tot de wezenlijke taken van een O.S. rekenen, maar dit doet niets af aan het O.S. concept.

Er zijn een aantal redenen waarom het nuttig is dat er een standaard O.S. op een computer meegeleverd wordt:

- Het zou ontzettend lastig zijn als je voor elk programma apart functies en procedures zou moeten schrijven om standaard taken zoals het lezen en schrijven op de harde schijf uit te voeren. Voor apparaten als printers is het nog te doen (en bijv. op MS-DOS gebeurt dat ook), maar voor een harde schijf is dat bijna onmogelijk omdat alle programma's gezamenlijk van deze schijf gebruik moeten maken, en er dus gemeenschappelijke afspraken moeten zijn. Dit is het makkelijkst als de afspraken in software gegoten zijn.

- Er is veel variatie in apparatuur, en dit verandert ook voortdurend. Het zou een slechte zaak zijn als je programma's niet meer zouden werken als je het nieuwste model harde schijf zou kopen (maar bij printers e.d. gebeurt dat helaas wel eens). Het O.S. voorkomt dit door een *abstractie*laag te definieren om de apparatuur te benaderen. In het O.S. vindt de vertaling plaats tussen de "abstractie" en de echte apparatuur. Als er een nieuw apparaat uitkomt hoeft alleen de vertaling voor dit apparaat toegevoegd te worden en alle programma's kunnen er mee werken. De software die dit verzorgt wordt meestal een "driver" genoemd, en wordt vaak door de leverancier van het apparaat erbij geleverd.
- Computers waar meer personen op moeten kunnen werken hebben bescherming nodig tegen fouten of opzet van gebruikers die nadelig voor anderen kunnen zijn. Dit kan alleen wanneer de software die dit verzorgt onafhankelijk is van die gebruikers.

3.1 Onderdelen van het O.S.

3.1.1 Besturing randapparatuur

Het deel van het O.S. dat een randapparaat bestuurt heet een *device driver*. Wanneer van een bepaald soort randapparaat er meer exemplaren aanwezig zijn, is slechts één driver nodig, maar wel voor elk apparaat een datastructuur. De device driver moet twee dingen doen:

1. aanroepen vanuit het O.S. behandelen (bijv. read en write). Deze aanroepen zullen meestal vanuit de processen komen maar ook intern kan het O.S. aanroepen doen, bijv. voor het filesysteem bij een monolitisch systeem.
2. Interrupts afhandelen in de z.g. interrupt routine. Het synchroniseren van interrupts met de andere afhandeling in de driver vereist een zorgvuldige opzet. Wanneer een interrupt komt is meestal een ander deel van het O.S. of een proces actief, of misschien zelfs een andere interrupt routine, en dus kan niet zomaar naar de code van de driver die een verzoek afhandelt, gesprongen worden. De interrupt zal i.h.a op een aparte stack afgehandeld worden, en zal een indicatie in de O.S. datastructuren neerzetten als de interrupt tot gevolg heeft dat de device driver weer verder kan met een onderbroken actie. Pas wanneer alle interrupt routines beëindigd zijn, wordt dan gekeken of het O.S. het huidige proces tijdelijk moet onderbreken voor een andere activiteit. Het is daarom bijna nooit mogelijk om de devicedriver echt te laten wachten tot een ander deel van het O.S. klaar is met de datastructuur. De meeste O.S's gebruiken daarom voor het synchroniseren van kritische secties, wanneer daar een interrupt routine bij betrokken is, de simpele manier om interrupts tijdelijk uit te zetten. Dit mag niet te lang duren omdat andere interrupts dan wel eens te lang zouden moeten wachten en er data verloren kan gaan.

3.1.2 Het filesysteem

Het filesysteem (het onderdeel dat zorgt voor de indeling en het beheer van verschillende soorten schijven: harde schijven, diskettes, CD-ROMS e.d) kan als een soort apparaatbeheer gezien worden, maar het is natuurlijk ingewikkelder dan een apparaat zoals een printer. Daarom is het filesysteem ook als een echt wezenlijk ander onderdeel van het O.S. te beschouwen. Het is ook een zeer vitaal onderdeel om dat alle programma's die we draaien als files zijn opgenomen. Ja, zelfs het O.S. zelf

is als file opgenomen. Het is dus een centraal deel! Je kunt je afvragen hoe het mogelijk is dat het filesysteem onderdeel is van het O.S. als het O.S. zelf in het filesysteem opgenomen is. Dit is een soort kip-en-ei probleem: we hebben het O.S. nodig om files te kunnen lezen en we moeten files kunnen lezen om het O.S. te starten. We noemen dit een *bootstrap* probleem: (Hoe trek je jezelf aan je laarzen omhoog?) De oplossing is dat er eerst een soort minifilesysteempje gestart wordt dat net goed genoeg is om het O.S. op te starten en dat daar dan de echte filesysteem software in zit. Hoofdstuk 4 gaat uitgebreider hierop in.

3.1.3 Resource management

In de machine zijn verschillende *resources* (hulpbronnen) die gedeeld moeten worden door onderdelen van het O.S. en de processen die er draaien. Denk bijvoorbeeld aan:

- geheugen
- CPU-tijd
- schijfruimte
- andere apparaten

Het O.S. moet hiervoor de toegang regelen. Dit houdt in: het beheer, d.w.z. bijhouden welk onderdeel iets in gebruik heeft, en of er conflicten bij het gebruik zijn (het readers en writers probleem). Verder synchronisatie, ervoor zorgen dat onderdelen kunnen wachten op een resource en verwittigd of doorgestart worden als een resource vrijkomt. Verder is een belangrijk probleem het schedulen, d.w.z. bepalen wie op welk moment aan de beurt komt. Het schedulen is vooral belangrijk i.v.m. het gebruik van de CPU (omdat die maar één ding tegelijk kan doen), en in iets mindere mate het gebruik van het geheugen.

3.1.4 Bescherming

Het O.S. moet zichzelf en diverse hulpbronnen beschermen tegen oneigenlijk gebruik en tegen fouten in programma's. Zo moet het niet mogelijk zijn om de randapparaten op verkeerde manier aan te spreken. Daarom gaat toegang tot randapparaten via (hopelijk) goed uitgeteste device drivers, en mogen gebruikersprogramma's meestal niet rechtstreeks een randapparaat aanspreken. Het geheugen van het O.S. en de diverse processen kan beschermd worden d.m.v. de memory management voorzieningen. Het filesysteem in een computer die door meerdere personen gebruikt wordt, levert bescherming op gebruikersniveau.

Om een goede bescherming van het O.S. tegen de te draaien programma's te krijgen, en om deze programma's tegen elkaar te beschermen.Dit kan heel goed met behulp van de al besproken memory management (virtueel geheugen). Zo'n programma wordt dan in een "beheersbare" eenheid gezet, die we een proces noemen. Een proces heeft min of meer een eigen leven binnen het O.S. en kan door het O.S. ook afgebroken worden als het iets onbehoorlijks doet. Processen worden verder besproken in hoofdstuk 5.

Het is vaak nuttig om gegevens te kunnen uitwisselen tussen het O.S. en processen of tussen processen onderling. Dit kan door bepaalde stukken geheugen toegankelijk te maken voor één of meer

processen. We spreken dan van “shared memory”. Shared memory kan naar keuze alleen voor lezen beschikbaar gesteld worden of zowel voor lezen als schrijven (veranderen). In het laatste geval moeten er natuurlijk maatregelen genomen worden om te voorkomen dat er ongelukken gebeuren als meer dan één proces tegelijk de data wil veranderen, of om te voorkomen dat onbevoegde processen de data veranderen. Hierbij wordt o.a. gebruik gemaakt van de memory management hardware. Ook kan shared memory gebruikt worden om stukken programma die in meer dan één proces nodig zijn, gezamenlijk te gebruiken.

3.2 De structuur van een Operating System

Je zou een O.S. in principe als een verzameling procedures en functies kunnen aanbieden, die standaard in het geheugen van de computer geladen wordt. Programma’s die je maakt, moeten dan deze procedures vinden en aanroepen. Dit zou gedaan kunnen worden door een lijst van namen en adressen bij te houden en deze door de compiler/linker te laten gebruiken. Er zijn inderdaad Operating Systems die min of meer zo werken. Er zijn echter een aantal nadelen aan deze werkwijze:

- Als er een nieuwe versie van het O.S. uitkomt is het waarschijnlijk dat de adressen veranderen. Gecompileerde programma’s zijn dan niet meer correct en moeten opnieuw gecompileerd worden. Voor programma’s waarvan je niet zelf de broncode hebt is dat onmogelijk. Dit probleem is overigens best te omzeilen. Hier zijn twee mogelijkheden:

  1. Bouw op een vast adres een lijst van de adressen van de functies die het O.S. aanbiedt. Programma’s roepen nooit direct de functies aan, maar altijd indirect via deze lijst. Het enige dat constant moet blijven is de plaats waar een functie in de lijst staat. Bij een wijziging in het O.S. wordt de inhoud van de lijst aangepast, en evt. nieuwe functies worden achteraan bijgeplaatst.
  2. Via een TRAP instructie. Elke functie (systeemaanroep) krijgt een nummer, en dit nummer wordt voor de TRAP in een vast afgesproken register gezet, of op de stack. Het eerste dat de exception handler doet is dit nummer bekijken, testen of het binnen de juiste grenzen zit, en dan via een `switch` opdracht de juiste functie aanroepen. Het voordeel van deze methode is, dat we dan ook al in system mode zitten.

Het O.S. kan in feite ook als een apart programma beschouwd worden, echter in tegenstelling tot andere programma’s is het altijd aanwezig. We maken onderscheid tussen dit deel van het O.S., dat dan ook wel de *kernel* genoemd wordt, en andere programma’s die per stuk opgestart kunnen worden, en weer stoppen als ze klaar zijn. De laatste worden (als ze draaien) processen genoemd (zie hoofdstuk 5). Er kunnen meestal meerdere processen tegelijk actief zijn. We spreken dan van een *multitasking* operating system.

Het is mogelijk om delen van een O.S. af te splitsen en die als normale processen te draaien. Dit heeft het voordeel dat deze delen gemakkelijker vervangen kunnen worden, en bovendien dat het gemakkelijker is om *concurrency* (d.w.z. dat er verschillende activiteiten gelijktijdig optreden) in het O.S. te krijgen. Immers voor processen krijgen we concurrency er gratis bij, terwijl we anders in het O.S. weer een apart concurrency mechanisme zouden moeten maken.

Als we in het O.S. alleen de minimaal benodigde delen in de kernel stoppen, en alle andere delen is processen, dan spreken we over een “microkernel” O.S. Wanneer we alles van het O.S. in één dikke

kernel stoppen, dan noemen we het een “macrokernel” of “monolithisch” O.S. De meeste O.S’s zitten er een beetje tussenin.

Diensten van het O.S. die in aanmerking komen om buiten de kernel in processen gestopt te worden zijn:

- *Het filesysteem.* Dit beheert de structuren op de schijf en moet daarom I/O op de schijf doen. Omdat I/O typisch iets is wat goed geregeld is voor processen, ligt het voor de hand om dit af te splitsen.
- *Netwerk beheer.* Hier geldt min of meer hetzelfde. Het pure versturen en ontvangen van boodschappen kan in een driver gebeuren. De hogere protocollen echter hebben te maken met concurrency, het beheren van buffers, het wachten op I/O etc, en dat gaat in een proces meestal handiger dan wanneer hier weer aparte besturingsmechanismen voor gemaakt moeten worden.

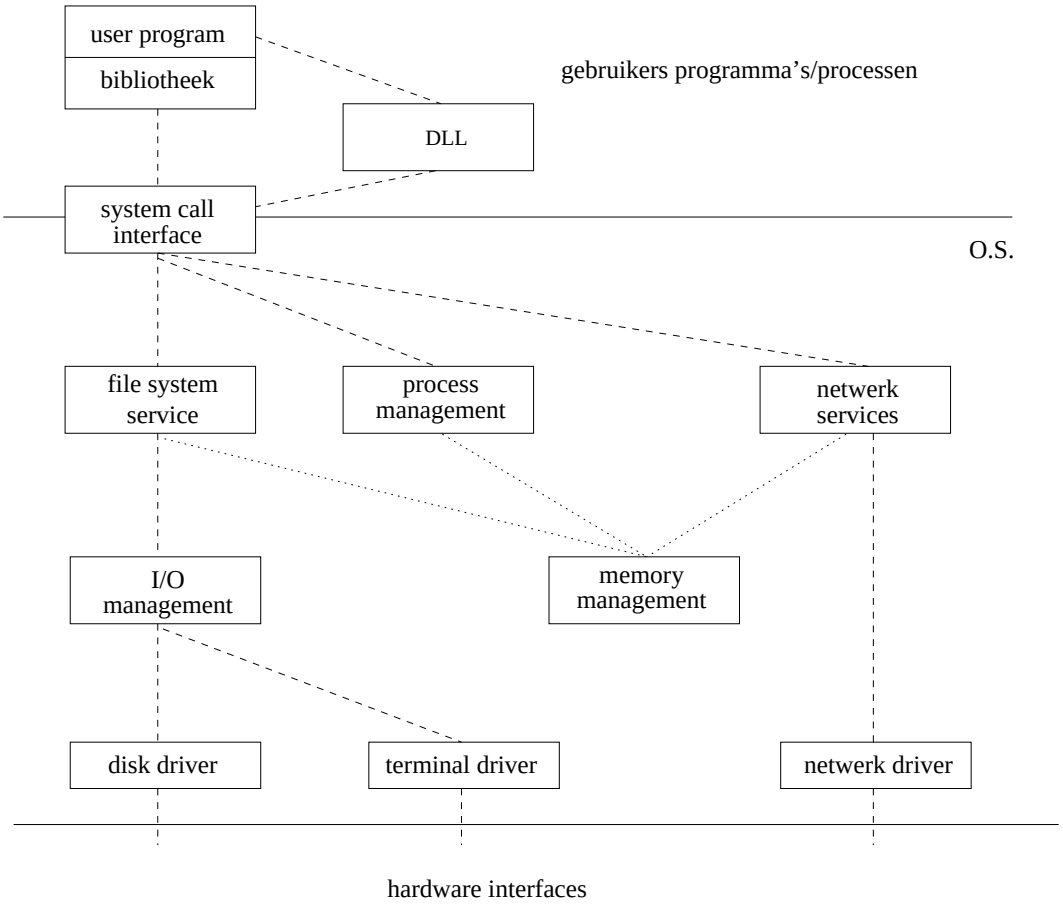
Een andere manier om een O.S. wat meer in delen op te zetten is het gebruik van “shared libraries” (zie 5.6). Hier bij worden delen van het O.S. in een stuk shared memory gezet wat dan naar behoefte in het geheugen geladen wordt. Als er een proces is dat deze instructies gebruikt komt het in het geheugen, maar hoeveel processen er ook gebruik van maken, er hoeft maar één exemplaar van in het geheugen te komen. Op deze manier wordt geheugenruimte bespaard. Omdat deze shared libraries (ook wel DLL’s genoemd, dynamic link libraries) gemakkelijk vervangen kunnen worden als er een fout inzit, of toegevoegd als er nieuwe functies nodig zijn, geeft dit ook een flexibele manier om het O.S. aan te passen. Deze methode wordt vooral in MS Windows gebruikt.

Een DLL wordt tijdens het uitvoeren van de instructies die erin staan gewoon als een onderdeel van het proces beschouwd en het heeft dus dezelfde privileges. Als de DLL dingen moet doen die bijv. de interne datastructuren van het O.S. moeten veranderen of in- en uitvoer doen, dan moet eerst een overgang naar supervisor mode gedaan worden, bijv. met een TRAP instructie. In Win32 wordt voor dit soort gevallen ook wel gebruik gemaakt van een “*virtuele*” device driver, d.w.z. een stuk software dat zich als een driver gedraagt, en dezelfde privileges als een driver heeft, maar die niet bij een echt apparaat hoort. Ook in Unix systemen worden wel virtuele device drivers gebruikt.

Tenslotte geven we in figuur 3.1 op de volgende pagina een schematische voorstelling van de onderdelen van een O.S. en hun onderlinge relatie. We zien hier duidelijk een lagenstructuur aanwezig.

3.3 Opgaven

1. Bekijk figuur 3.1 op pagina 60. Probeer van zoveel mogelijk stippellijntjes te beschrijven wat de relatie tussen de verbonden onderdelen is. (Voorbeeld: I/O management gebruikt de disk driver om data van de harde schijf te lezen en ernaar te schrijven.)
2. Beargumenteer van de onderdelen uit figuur 3.1 op pagina 60 welke wel en welke niet in de kernel van een *microkernel* operating system thuishoren.



Figuur 3.1: Structuur van een O.S.

Hoofdstuk 4

Files

Het filesysteem is een belangrijk onderdeel van een computersysteem, omdat hierin al onze data en programma's opgeslagen zijn. Het ontwerp van het filesysteem kan een belangrijke factor zijn in de efficiëntie van een computersysteem.

Verder is er vaak een verregaande integratie tussen de toegang tot files en andere I/O zoals netwerktoegang, zodat het ontwerp van de filetoegang consequenties heeft voor het hele systeem. Tenslotte zijn programma's ook als files opgeslagen en het uitvoeren daarvan is dus ook verweven met het filesysteem. In dit hoofdstuk zullen we het filesysteem zowel van de kant van het systeem, als van de kant van gebruikersprogramma's bekijken.

- De diensten (services) die het O.S. gewoonlijk aanbiedt voor het filesysteem (de architectuur dus)
- implementatie-aspecten van het filesysteem
- De manier waarop we vanuit programma's deze diensten kunnen gebruiken

4.1 File services

De diensten die het O.S. levert met betrekking tot het beheren van files kunnen in een aantal categorieen ingedeeld worden:

Storage service Dit is het beheer van de schijfruimte, het indelen ervan in files, het bijhouden waar deze files op de schijf staan, en het lezen en schrijven van de files

Directory service Dit is het bijhouden van namen voor files, en het organiseren van deze namen in directories (mappen)

Bescherming en autorisatie Het regelen van de toegang tot files tussen gebruikers onderling

Concurrency service Het regelen van gelijktijdige toegang tot files

Hoewel deze diensten op zich goed te onderscheiden zijn, zijn deze in de meeste operating systems niet helemaal strikt gescheiden. Toch zullen we ze hier in eerste instantie afzonderlijk behandelen.

4.1.1 Storage service en directory service

De *storage service* is dat deel van het O.S. dat zorgt dat files (bestanden) op de schijf opgeslagen worden, en dat deze gelezen en geschreven kunnen worden. Deze service deelt de schijf in en houdt bij op welke plaatsen op de schijf een file staat.

De *directory service* is het deel dat de namen van de files bijhoudt. Hiertoe behoort ook het verzorgen van directories en het opzoeken van een file als de naam aangeboden wordt. Directories kunnen ook weer als files opgeslagen worden dus de directory service heeft de storage service nodig. In de meeste operating systems zijn deze twee services niet los van elkaar te gebruiken, maar het is toch nuttig om ze zoveel mogelijk als afzonderlijke diensten te zien. De voordelen hiervan zijn o.a.

- Het is gemakkelijker om de interne structuur van de opslag van files op de schijf te veranderen zonder dat dit gevolgen heeft voor de directories
- Het is mogelijk de directories apart op te slaan van de files, bijvoorbeeld wanneer de files in een netwerk staan en de directories lokaal.
- Het is mogelijk files centraal op te slaan en verschillende soorten namen voor dezelfde files te gebruiken, bijvoorbeeld zowel Unix namen op een Unix werkstation als MS Windows namen op een PC.

Storage service:
Het onderdeel van het O.S. dat bijhoudt waar de files opgeslagen zijn op de harde schijf of ander opslagmedium (CD, DVD).

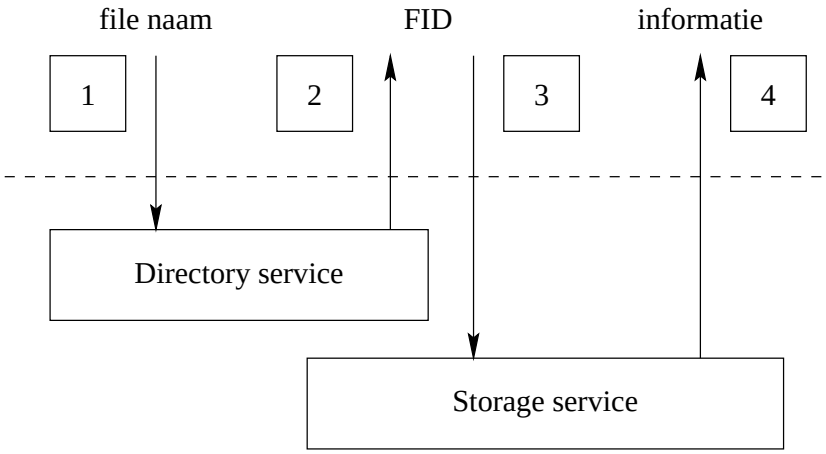
Directory service:
Het onderdeel van het O.S. dat namen (meestal in een hiërarchische directory structuur) koppelt aan files.

In het “ideale” geval kan een programma een file benaderen door eerst de directory service aan te roepen met de filenaam, waarbij dan een interne “File Identifier” teruggekregen wordt. De FID is een interne aanduiding van de file, bijv. een nummer, of een pointer. Met deze FID kan dan de file benaderd worden, zie figuur 4.1.

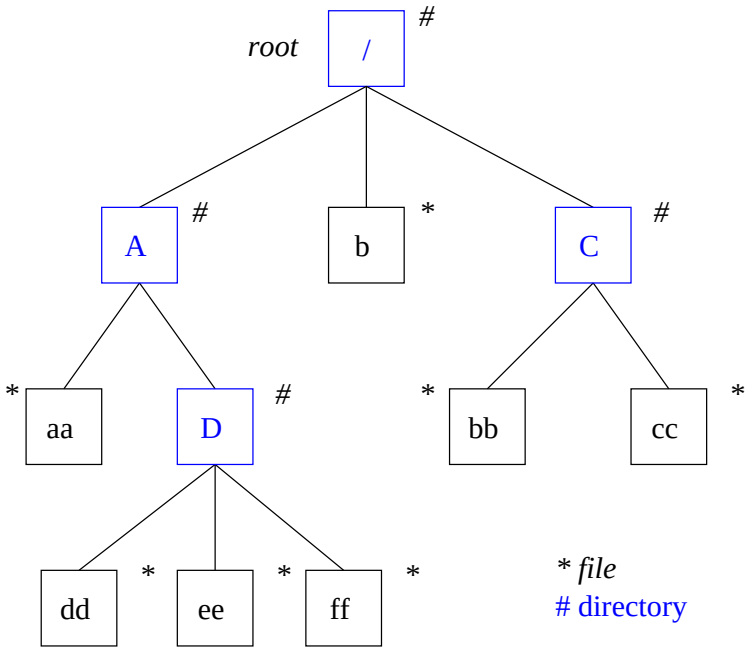
In moderne operating systems worden de directories en files in een hiërarchische structuur georganiseerd, d.w.z. in een boomstructuur waarin de bladeren de files zijn en de directories de interne knopen. Zie bijv. figuur 4.2.

Een directory kan eenvoudig geïmplementeerd worden als een lijst van paren: (naam, FID). Een directory wordt ook als file opgeslagen en heeft dus een eigen FID. Er wordt wel een aantekening gemaakt dat dit geen “gewone” file is, zodat programma’s alleen via de directory service er iets mee kunnen doen. Een wat meer “sophisticated” directory service zou een directory kunnen implementeren als één van de vormen van een B-boom. Dit versnelt het zoeken, vooral in grote directories.

In een systeem waarin de directory service en de storage service geïntegreerd zijn, kan de storage service informatie zelf in de directory entries opgenomen worden. In dat geval bestaat er dus niet een aparte FID.



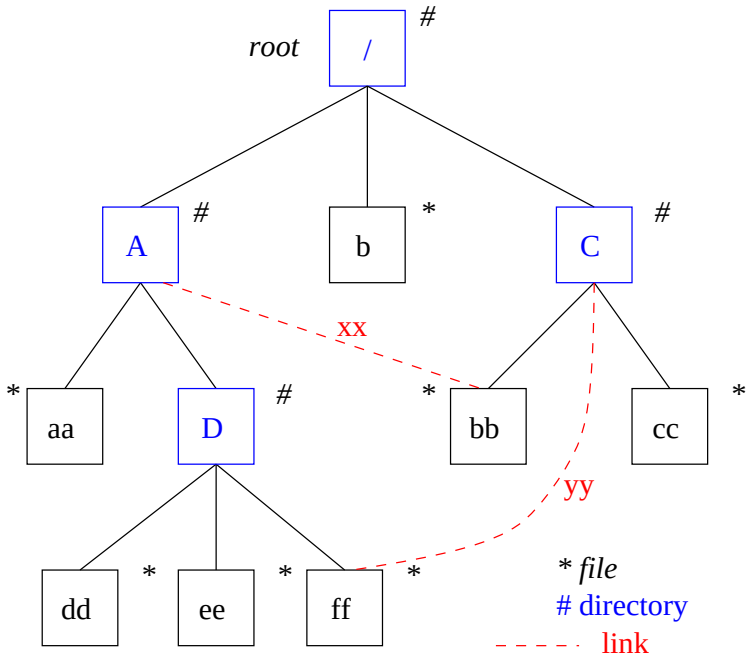
Figuur 4.1: Directory service en storage service
(1) Geef naam aan Directory Service (2) Directory Service geeft FID terug
(3) Geef FID aan Storage Service (4) Storage Service geeft data uit de file.



Figuur 4.2: Hiërarchische filestructuur

4.1.2 Links

In het Unix filesysteem is het mogelijk dat een file meer dan één naam heeft. We zeggen dan dat er meerdere links naar de file zijn. We merken op dat dit een voorbeeld is van een scheiding tussen de directory service en de storage service, immers de file is maar één keer opgeslagen. In systemen waarin de directory service en de storage service nauw verweven zijn, is dit veel moeilijker te realiseren. Een structuur met links kan bijvoorbeeld zijn zoals in figuur 4.3. Hierin zijn /A/xx en /C/bb twee verschillende namen voor dezelfde file, evenals /A/D/ff en C/yy. In feite is het zo dat er geen verschil tussen deze namen is, dus het onderscheid tussen de doorlopende en gestreepte lijnen is niet aanwezig. Wanneer links gebruikt worden dan hoeft de directory structuur geen boom meer te zijn.

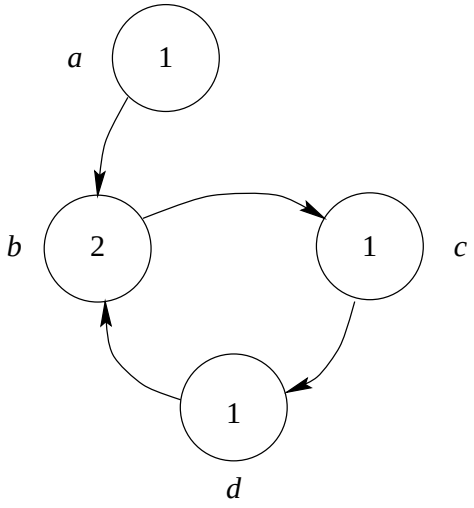


Figuur 4.3: Directory structuur met extra links

Een probleem dat optreed wanneer meer links mogelijk zijn, is dat het verwijderen van een file uit een directory niet tot gevolg hoeft te hebben dat de file verdwijnt. Immers er kan nog een andere naam zijn die naar dezelfde file verwijst. Dit is een algemeen probleem dat op kan treden als de directory service en de storage service gescheiden worden. Dit kan alleen opgelost worden als bijgehouden wordt hoeveel verwijzingen er naar een file zijn. Dit wordt een *reference count* genoemd. Zolang dit aantal groter dan 0 is moet de file blijven bestaan. Het Unix commando “rm” (of de bijbehorende system call “unlink” verlagen in eerste instantie alleen de reference count, en pas wanneer deze 0 wordt (en er geen programma is dat de file open heeft), wordt deze verwijderd. Dit vereist een goede coördinatie tussen de directory service en de storage service. In MS-DOS en MS Windows zijn de directory service en storage service nauw met elkaar verbonden en daar betekent het verwijderen van een file uit de directory ook echt het weggooien van de file (zij het dat de informatie nog niet direct helemaal weg is). Het modernere NTFS filesysteem (Windows NT/XP) heeft wel de mogelijkheid van links maar het Windows O.S. heeft geen mogelijkheden om die te gebruiken.

Link:
Een (extra) naam voor een file.
Reference count:
Een teller die bijhoudt hoeveel verwijzingen er naar een object zijn.

Reference counts zijn een vorm van *garbage collection*, het opruimen van overbodige spullen. Reference counts werken echter niet goed wanneer cyclische structuren mogelijk zijn. Zie bijv. figuur 4.4. Wanneer in deze figuur de link van *a* naar *b* weggehaald wordt, zijn de knopen *b*, *c* en *d* niet meer bereikbaar, terwijl ze toch een reference count van 1 hebben. Ze zullen dus niet opgeruimd worden terwijl ze niet meer gebruikt kunnen worden.



Figuur 4.4: Structuur met cycles

Om dit soort problemen (en andere soortgelijke) is het in Unix niet toegestaan om links naar directories te maken[1]. De directory structuur is hierdoor een acyclische graaf (DAG=*directed acyclic graph*).

4.1.3 Symbolische en harde links

In moderne Unix systemen wordt onderscheid gemaakt tussen “symbolische” en “harde” links. Bij een harde link behoort bij de verschillende namen (evt. in verschillende directories) dezelfde FID. Bij een symbolische link wordt niet verwezen naar de FID maar naar de oorspronkelijke naam van de file. Bijvoorbeeld in figuur 4.3 als we de links vervangen door symbolische links, dan wijst de directory entry /A/xx niet naar dezelfde FID als /C/bb maar naar de *naam* “/C/bb”. Een symbolische link zorgt dus niet voor een extra referentie naar de FID en levert dus ook geen bijdrage aan de reference count. Dit heeft tot gevolg dat de originele file verwijderd kan worden en dat de symbolische link daarna in het niets wijst (totdat er weer een nieuwe file aangemaakt wordt onder de oorspronkelijke naam). Het bestaan van symbolische links heeft tot gevolg dat in de directory entries een indicatie toegevoegd

[1] In feite wordt er in elke directory een link naar zichzelf gemaakt onder de naam . en van elke subdirectory naar de ouder directory onder de naam . . , maar het systeem zorgt ervoor dat deze zorgvuldig verwijderd worden als de bijbehorende directory verwijderd wordt

moet worden die aangeeft of de naam een symbolische link is of een harde (met andere woorden of er in de entry een naam staat of een FID).

Harde link:
Een link waarbij de naam verwijst naar de FID.
Symbolische link:
Een link waarbij de naam verwijst naar een andere naam.

Het verschil wordt direct duidelijk als we naar de implementatie kijken: bij een harde link hebben we twee of meer directory entries waarin dezelfde FID voorkomt. Bijvoorbeeld met de files uit figuur 4.3 (de FID getallen zijn willekeurig, maar voor de links zijn ze natuurlijk gelijk):

directory A

naam	FID
aa	237
D	523
xx	768

directory C

naam	FID
bb	768
yy	836
cc	945

directory D

naam	FID
dd	117
ee	481
ff	836

Voorbeelden van harde links

directory A

naam	H/S	FID
aa	H	237
D	H	523
xx	S	“/C/bb”

directory C

naam	H	FID
bb	H	768
yy	S	“/A/D/ff”
cc	H	945

directory D

naam	H	FID
dd	H	117
ee	H	481
ff	H	836

Voorbeelden van symbolische links

4.1.4 Metadata

Behalve de inhoud van een file (de data) moet het O.S. diverse andere informatie over een file opslaan. Deze informatie noemen we de *metadata*. De volgende gegevens vallen hier bijvoorbeeld onder:

- type van de file: directory of gewone file
- de grootte van de file
- op welke plaats op de schijf staat de data van de file
- de eigenaar van de file
- welke toegangspermissies zijn er
- de datum van creatie, laatste wijziging en laatste toegang tot de file
- reference count

Metadata:
Data (gegevens) die informatie over andere gegevens bevatten.

Niet alle metadata is in elk O.S. aanwezig. Zo heeft het FAT filesysteem op MS-DOS/MS Windows niet de notie van eigenaar/groep, en geen reference count, terwijl de toegangspermissies ook beperkt zijn[2]. De vraag is nu waar deze metadata opgeslagen wordt. Er zijn in principe drie mogelijkheden:

1. In de directory
2. Bij de file zelf
3. In een apart deel van de schijf.

Op MS-DOS en MS Windows wordt de eerste methode gebruikt: de metadata staat in de directory. In feite betekent dit dat de directory service en de storage service geïntegreerd zijn. Een gevolg hiervan is dat het gebruik van harde links onmogelijk is. Immers de directory entries van de links moeten naar dezelfde metadata wijzen. Het gebruik van symbolische links zou met deze opzet mogelijk zijn, maar wordt door deze systemen niet gebruikt.

Unix systemen daarentegen gebruiken methode 3: de metadata staat op een apart gedeelte van de schijf. Hiervoor wordt een tabel gebruikt waarvan elk element één file beschrijft. Zo'n element wordt een *inode* genoemd, en de tabel heet de *inode tabel*. De index in deze tabel voor een specifieke file heet het *inode number*, en dit is in feite de FID die in het voorgaande genoemd is.

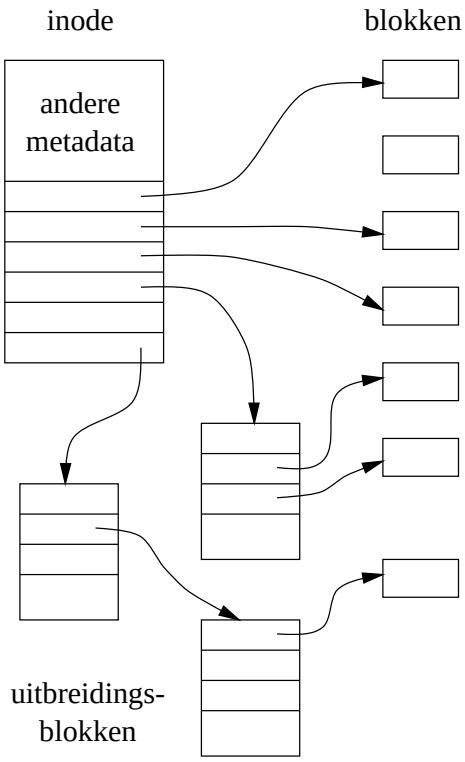
Een schijf is in blokken ingedeeld van een vaste grootte (bijv 1024 bytes). De interface kan lezen en schrijven in gehele aantallen blokken. Een file zal dus i.h.a. ook in een geheel aantal blokken op de schijf opgeslagen worden, hoewel het laatste blok niet vol hoeft te zijn.

De simpelste manieren om aan te geven waar een file op de schijf staat zijn:

1. het bloknummer van het begin van de file op geven en het aantal blokken dat de file in beslag neemt. Dit vereist echter dat alle blokken van de file achter elkaar staan. Omdat een file groter gemaakt kan worden lange tijd nadat hij is aangemaakt kan dit niet altijd, want de achterliggende ruimte kan al in beslag genomen zijn door een andere file. Bovendien wanneer meer files tegelijk aangemaakt worden (door hetzelfde programma of door gelijktijdig lopende verschillende programma's) dan is het moeilijk om van te voren genoeg ruimte te reserveren. Deze methode wordt dan ook niet zoveel gebruikt.

2. een lijst opnemen van alle blokken waarop de file staat. Voor een grote file kan deze lijst heel groot worden. Als de maat van kleine en grote files erg uiteenloopt, kun je dan niet meer met één vaste entry van bijv. een inode tabel (Unix) of directory entry (MS-DOS) uitkomen. Anders kunnen grote files niet groot genoeg worden, of verspil je ruimte voor kleine files, of beide. In het MS-DOS systeem worden in zo'n geval meerdere directory entries gebruikt die aan elkaar gelinkt worden. In het Unix systeem wordt een wat intelligenter systeem gebruikt: In de inode entry wordt een vast aantal woorden gereserveerd voor verwijzingen naar de eerste blokken van de file. Wanneer de file groter wordt, worden alle volgende bloknummers in een apart diskblok gezet en het nummer hiervan wordt in de inode gezet. Er wordt dus een indirectie toegepast. In feite creëert dit een boomstructuur (zie figuur 4.5). Nieuwere filesystemen gebruiken vaak een gebalanceerde boom. Dit is efficiënter als er veel door de file heen en weer gesprongen wordt.

3. Sommige O.S. gebruiken een combinatie waarbij een file zoveel mogelijk in grote aaneengesloten stukken op de schijf gezet wordt, en waarbij van elk stuk het beginbloknummer en het aantal blokken opgeslagen wordt. De technieken hierboven kunnen dan ook toegepast worden.



Figuur 4.5: Inode structuur

Voor grote schijven worden meestal een aantal fysieke blokken van de schijf tot “logische” blokken samengevoegd. Deze worden vaak *clusters* genoemd. Het voordeel is dat er dan minder bloknummers nodig zijn, dus voor grote files minder metadata. Het nadeel is dat in ieder geval voor kleine files er meer ruimteverlies is, omdat altijd minstens één cluster gereserveerd wordt. Ook voor grotere files is er verlies omdat bij een random verdeling van de filegroottes, het laatste cluster gemiddeld half leeg zal zijn. Wanneer er veel files zijn, dan is er dus ook veel verlies. In het oorspronkelijke FAT filesysteem op MS-DOS en MS Windows zijn deze problemen nog nadrukkelijker aanwezig omdat hier (in de directory entries) voor een bloknummer 16 bits gereserveerd worden[3]. Het aantal clusters van een schijf kan dus niet meer dan 65536 bedragen (of de rest is onbereikbaar). Voor grote schijven zijn er dan twee mogelijkheden:

1. Verdeel de schijf in kleinere partities, die elk als afzonderlijke schijven beschouwd worden in het O.S. Nadeel: een file kan niet groter worden dan een partitie, en soms moet je met files gaan zitten schuiven als de ene partitie vol zit en de andere heeft nog ruimte over

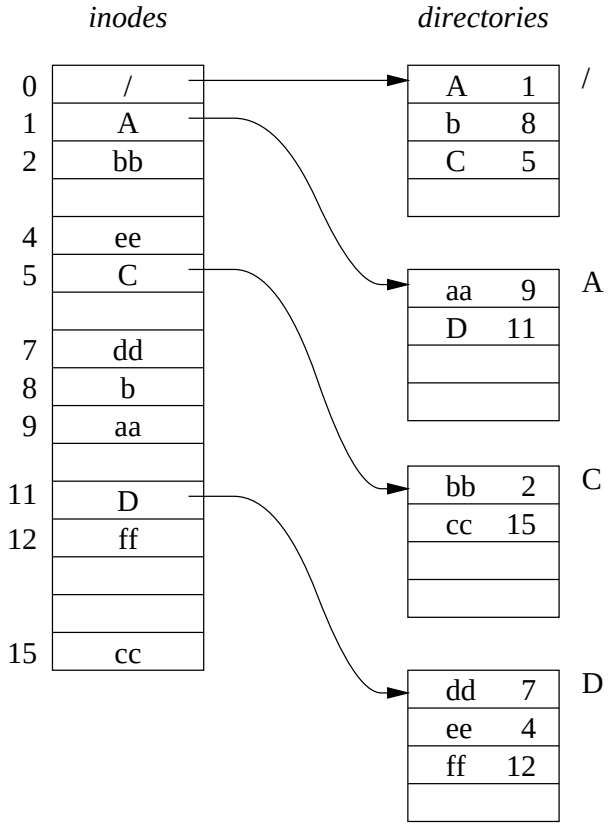
2. Maak de clusters groter. Nadeel: veel verlies.

Moderne Unix filesystemen hebben verschillende mogelijkheden om deze problemen te verminderen, bijv. door voor de eindblokken in een file niet een heel cluster te reserveren maar een deel van een cluster.

[2]MS Windows NT heeft de meeste wel

[3]Het nieuwere FAT32 filesysteem gebruikt 32 bits bloknummers

Hier volgt nog een voorbeeld van een deel van de inode tabel voor de directory structuur uit figuur 4.2. In een Unix systeem staat de inode entry van directory "/" altijd op inode nummer 0. De rest is via de directory structuur te vinden. De inode nummers (behalve nr 0) in de figuur zijn willekeurig gekozen.



Figuur 4.6: Inodes en directories

4.2 Disk cache

De toegang tot een opslagmedium als een harde schijf of CD-ROM is behoorlijk traag. Om een bepaalde plaats op zo'n apparaat te vinden kan al ca. 10 msec nodig zijn, dan moet nog gewacht worden tot het de juiste hoeveelheid rondgedraaid heeft en dan moet de data nog heen- of weergeschoven worden. Dit laatste kan ook gemakkelijk enige microseconden per byte duren. Vergeleken met een toegangstijd van 10 nanosec per woord voor het interne geheugen is dat zeer langzaam. Daarom wordt vaak een deel van het interne geheugen voor een *disk cache* gereserveerd. Blokken die van een schijf of CD-ROM gelezen worden, of er naar toe geschreven, worden hierin bewaard, zodat ze de volgende keer snel beschikbaar zijn. Dit is hetzelfde principe als het snelle-geheugen cache tussen de CPU en het interne geheugen (zie pagina 21), maar nu op een ander niveau toegepast. Ook hier kan bij het schrijven weer gekozen worden tussen een delayed-write cache en een write-through cache. Een delayed-write cache heeft tot gevolg dat een programma dat een schrijfopdracht naar een file doet, direct verder kan gaan (mits er ruimte in de cache is), maar er is dan geen garantie dat de schrijfopdracht echt gelukt is. Mocht er iets fout gaan op de schijf, of mocht de machine uitvallen dan zijn de gegevens verlo-

ren. Bovendien ligt de volgorde waarin de gegevens uiteindelijk naar de schijf geschreven worden dan niet meer vast. Voor sommige toepassingen (vooral databases) is het belangrijk om die volgorde zelf te kunnen bepalen. In zulke gevallen is het beter om een write-through cache te gebruiken, maar bijvoorbeeld voor tijdelijke files is dat overbodig. Het Unix systeem heeft een speciale systeem aanroep `sync()` die tot gevolg heeft dat de hele cache naar de schijf geschreven wordt. Standaard loopt er op een Unix systeem een programmaatje dat niet anders doet dan om de paar seconden `sync()` aanroepen.

Disk cache:
Een stuk geheugen dat wordt gebruikt om blokken uit een harde schijf of ander langzaam opslagmedium tijdelijk op te slaan om snellere toegang te verkrijgen.

Op moderne Unix systemen kan ook nog per file aangegeven worden of de blokken voor deze file als write-through behandeld dienen te worden. En het O.S. doet er goed aan om directories en inodes ook zo te behandelen.

4.3 File service en devices

Wanneer de storage service aangesproken wordt, zal deze uiteindelijk een device driver moeten aanroepen. Hoe weet de storage service welk device gebruikt wordt voor een bepaalde file? Hiervoor zijn twee methoden in gebruik: òf deze informatie is af te leiden uit de filenaam òf hij is indirect af te leiden. We geven van elk een voorbeeld.

4.3.1 MS-DOS en MS Windows: filenaam afhankelijk

In MS-DOS en MS Windows is het file systeem zo opgezet dat uit de filenaam af te leiden is welk device gebruikt moet worden. Dit gebeurt op twee niveaus:

1. Met behulp van *drive letters*, zoals A: en C: wordt aangegeven dat een file op een specifieke schijf staat. De letter wordt gebruikt als index in een tabel van devices (de letter kan ook een deel van een schijf, een partitie aanduiden). De letter bepaalt niet alleen welke device driver gebruikt moet worden, maar kan ook een andere filestructuur aangeven. Zo wordt op een CD-ROM een ander soort filesysteem gebruikt dan op een harde schijf of een diskette. Er zijn zelfs nog uitgebreidere mogelijkheden: een drive letter kan gekoppeld worden aan een “netwerk” drive, waarbij de file op een andere computer staat, of de drive letter kan gekoppeld worden aan een “virtueel” device, bijvoorbeeld een gecomprimeerd gedeelte van een schijf. De device driver zal in zo’n geval niet een echte interface aansturen maar net doen alsof, terwijl de echte dan ergens anders vandaan komt en naartoe gaat.

2. Het tweede mechanisme dat gebruikt wordt is dat bepaalde filenamen speciaal behandeld worden. Dit wordt gebruikt voor speciale apparaten, zoals het toetsenbord (CON), printers (LPT1, LPT2) en communicatiepoorten (COM1, COM2), of het virtuele null device (NUL). De manier waarop dit gedaan is is als volgt: Tijdens het opstarten van het systeem is het mogelijk device drivers voor deze (fysieke of virtuele) apparaten te laden. Elk van deze drivers heeft de code voor zichzelf (bijv. “CON”) op een speciale plaats staan. Tijdens het laden worden de drivers

in een lijst gehangen. Bij het openen van een file wordt de filenaam vergeleken met de namen in deze lijst en als er een gelijk is wordt de I/O naar deze driver gestuurd, zonder dat het filesysteem daarbij aan de pas komt. Bij het vergelijken wordt alleen naar het filenaamgedeelte voor de punt gekeken, en niet naar het deel achter de punt, en ook niet naar de directory. Dit is nogal gevaarlijk want data die naar de file "`C:\MYNDIR\NUL.TXT`" geschreven wordt, wordt dus niet naar een file geschreven maar naar het NUL device (waarvan de functie is om de data te laten verdwijnen). Bij het ontwerp van deze voorziening kunnen wel enige kanttekeningen gezet worden!!

4.3.2 Unix, special files en mounting

De Unix manier om aan te geven bij welk device een file hoort, bestaat ook uit twee delen. Hierbij wordt niet naar de filenaam op zich gekeken, maar deze informatie wordt apart aan de filenaam gekoppeld. Dit komt ook meer overeen met de Unix filosofie dat de directory service en de storage service zoveel mogelijk uit elkaar getrokken zijn. De twee fasen zijn:

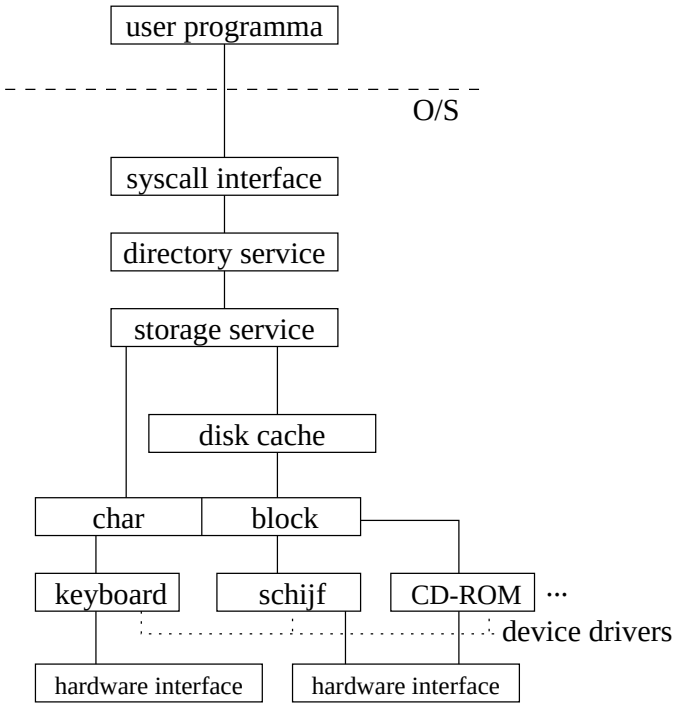
1. Een file kan als "speciaal" aangemerkt zijn. Dit gebeurt doordat de naam niet gecreëerd wordt met een normaal filecreatie systeemaanroep, maar met een speciale aanroep "mknod". Er wordt dan wel een inode voor de file gemaakt, maar hierin staat aangetekend dat het geen gewone file is (en dus geen diskruimte inneemt). In plaats van de grootte van de file worden er twee getallen opgenomen, het "major device number" en "minor device number". Het "major device number" geeft aan welke soort device het is (bijvoorbeeld welke soort schijf, CDROM of DVD), en is in feite een index in een tabel met device drivers. Het "minor device number" geeft een volgnummer aan wanneer er meer apparaten van hetzelfde soort zijn, en is in feite een index in de lijst van records die de device driver heeft om de verschillende apparaten van dezelfde soort uit elkaar te houden. Inodes van deze soort worden traditioneel in de directory "/dev" aangemaakt, maar er is geen enkel bezwaar tegen om ze in een andere directory te plaatsen, of in een andere directory er links naar te maken. Bekijk zelf eens op een Unix systeem welke special files er in /dev aanwezig zijn.

   In Unix zijn er twee soorten "special" files: "block" en "character". In een directory listing met het commando "ls -l" worden ze aangegeven met "b" resp. "c" als eerste letter.

   Block devices zijn apparaten die in blokken gelezen en geschreven moeten worden, zoals schijven en CD-ROMS. Deze blokken worden ook in een cache bewaard.

   Character devices zijn apparaten die teken voor teken gelezen en geschreven worden, bijvoorbeeld toetsenbord, printer en modem. Voor deze apparaten is het zinloos, of beter fout, om de data in een cache te bewaren. In figuur 4.7 op de pagina hierna is de relatie tussen filesysteem, cache, en block- en characterdevices weergegeven.

2. Voor de schijven geldt dat elke schijf (in traditionele Unix systemen) zijn eigen inode tabel heeft. Het is zelfs mogelijk om, net als op MS-DOS of MS Windows, een schijf in partities te verdelen, en dan heeft elke partitie zijn eigen inode tabel. Het voordeel hiervan is dat een fout of een beschadiging op één partitie de andere partities niet beïnvloedt. Het nadeel is natuurlijk dat de grootte van files wordt beperkt door de grootte van de partitie. Verder is het niet mogelijk om harde links naar andere partities te maken omdat in de directory entry alleen het inode number staat, en geen schijf- of partitienummer. Op moderne Unix systemen wordt vaak maar één partitie per schijf gebruikt, en bij deze systemen is het zelfs mogelijk om meer schijven

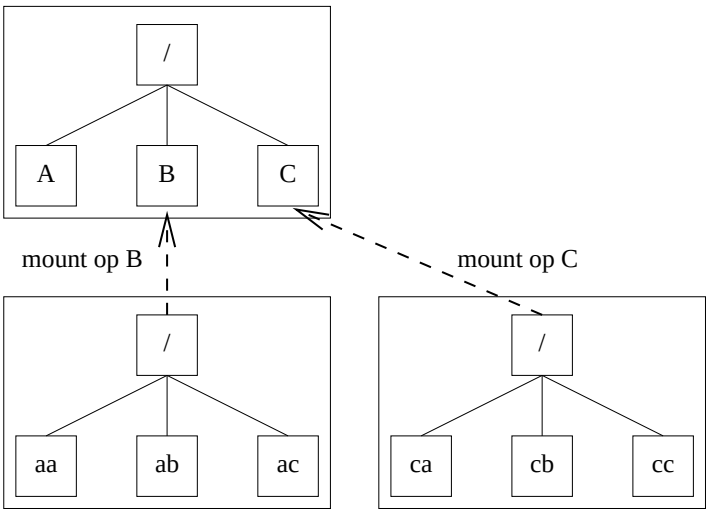


Figuur 4.7: File system, cache, en drivers

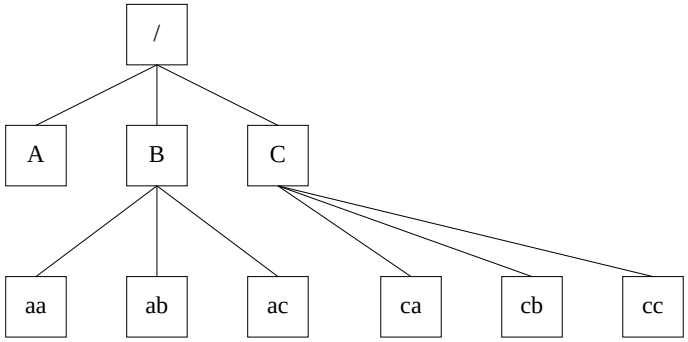
gezamenlijk als één logische schijf te gebruiken, waarbij ze dus samen één inode tabel hebben en waarbij de bloknummers doorlopen van de ene schijf naar de andere. We spreken dan over een *multi-volume* filesysteem.

Wanneer we meer partities of meer schijven hebben moeten we een manier hebben om deze aan elkaar te knopen in de directory service want op dat niveau heeft Unix slechts één directory structuur. Dit gebeurt door het *mount* commando (systeem aanroep). Met dit commando kan een partitie of schijf geknoopt worden aan een directory in een andere partitie. De oorspronkelijke directory informatie wordt hierdoor onzichtbaar en de directory gaat wijzen naar de *root* (/) van de te mounten partitie. Dit gebeurt alleen in het geheugen van de computer, de originele directory inhoud op de schijf blijft intact. Toch wordt meestal een lege directory voor dit doel gebruikt. Na het mounten is er voor de gebruiker niet meer te zien dat hier een speciale overgang zit (het mount-punt is “naadloos”). Zie figuur 4.8 en 4.9 op de rechter pagina.

Tegenwoordig wordt “mount” niet alleen gebruikt om partities aan elkaar te knopen maar ook voor file systemen op andere computers. Omdat het aantal mount-punten hierdoor nogal groot kan zijn, en voortdurend wisselen, wordt voor dit doel vaak gebruik gemaakt van een “auto-mounter”, een stuk software in het O.S. dat als een robot in actie komt als er een mount punt nodig is, en dan de mount opdracht onzichtbaar uitvoert. Als het mount punt een tijdje niet gebruikt wordt, zal de automounter zelfstandig besluiten om een “unmount” te doen. Hierdoor blijft de hoeveelheid geheugen die voor de mount informatie nodig is, binnen de perken.



Figuur 4.8: Mount points



Figuur 4.9: Resulterende directory structuur

4.4 Toegangsbescherming

In een systeem waarin meerdere gebruikers tegelijk kunnen werken is het belangrijk dat files niet door onbevoegden gelezen, gewijzigd, aangemaakt of weggegooid kunnen worden.

Ook hier zijn weer verschillende manieren om dit te bereiken. De belangrijkste hiervan zijn:

1. **Access Control Lists:** Een Access Control List (ACL) houdt bij een file (of directory) bij welke gebruikers toegang tot deze file hebben, en wat zij ermee mogen doen:

gebruiker	toegang
john	read
mary	write + read

   Het Unix systeem heeft een primitieve vorm van ACL's waarin alle gebruikers in 3 categorieën ingedeeld worden (eigenaar, groepsgenoten en anderen). De toegangsrechten worden dan aan deze categorieën toegekend. Het zou natuurlijk fraaier zijn als voor elke gebruiker afzonderlijk rechten toegekend zouden kunnen worden, en moderne Unix systemen hebben dit dan ook toegevoegd aan het standaard systeem. Windows XP heeft ook een uitgebreid systeem van ACL's op het NTFS filesysteem dat zowel afzonderlijke gebruikers als groepen gebruikers kent.

ACL (Access Control List):
Een lijst die aangeeft welke toegangsrechten verschillende gebruikers tot een file hebben.

   Bij ACL's (en dus ook bij het standaard Unix protectiemechanisme) komt de vraag naar voren, waar dit thuishoort: in de directory service of in de storage service? Als het in de directory service opgenomen wordt dan is het mogelijk om verschillende protecties te verbinden aan verschillende namen voor de file. Echter omdat de directory entries het eigendom kunnen zijn van iemand anders dan de file zelf, zou er met deze protecties gesjoemeld kunnen worden. Daarom is het beter om deze informatie bij de file zelf te houden, in het geval van Unix dus in de inode. Dan is het duidelijk wie de baas is over de protecties, namelijk de eigenaar van de file (die ook in de inode staat). Het zou mogelijk zijn om een systeem te bedenken waarbij er een extra beveiligingslaag zit tussen de directory service en de storage service.

   Op MS-DOS en MS Windows waarbij er geen aparte storage service is (deze informatie staat in de directory), wordt toegangsinformatie ook in de directory entry opgenomen. Overigens bevat deze informatie alleen of de file geschreven mag worden, en of deze in een directory listing getoond wordt. MS Windows NT heeft een apart filesysteem (NTFS) waar wel een geavanceerd ACL systeem in zit.

2. **Capabilities:** Bij dit systeem wordt niet meer bij een file bijgehouden welke gebruikers toegang hebben, maar kunnen gebruikers "toegangskaartjes" voor een file krijgen. Deze elektronische toegangskaartjes bevatten de identiteit van de file (een of ander bitpatroon dat uniek de file identificeert), de toegangsrechten (een bit voor elk recht), en een controledeel om te voorkomen dat een gebruiker zelf vervalste toegangskaartjes maakt. Een programma kan aan het systeem vragen om een nieuw toegangskaartje te maken waarin minder rechten zitten (bijv. om dat aan een andere gebruiker te geven), maar nooit om er een met meer rechten te maken.

Capability:
Een elektronisch toegangskaartje voor (bijvoorbeeld) een file, dat ook de toegestane operaties bevat.

unieke identifier	toegangsrechten	controlebits

Voorbeeld van een capability

De controlebits zijn zodanig gekozen dat een gebruiker deze niet zelf kan berekenen. Alleen het systeem is in staat om deze uit te rekenen bij een bepaalde combinatie van unieke identifier en toegangsrechten. De hoeveelheid bits voor deze onderdelen worden zo groot gekozen dat het ondoenlijk is om alle combinaties uit te proberen.

4.5 Systeeminterfaces voor files

We zullen in deze sectie laten zien hoe de systeemaanroepen voor files eruit zien. We zullen hiervoor de Unix interface nemen, omdat deze simpel is, met uitstapjes naar MS-DOS en MS Windows.

De belangrijkste Unix systeemaanroepen voor files zijn:

```
fd = open (filepath, manier, mode)
fd = creat (filepath, mode)
nr = read (fd, buffer, nr_bytes)
nr = write (fd, buffer, nr_bytes)
position = lseek (fd, offset, vanwaar)
status = close (fd)
```

De enige keer dat de filenaam (pad) gebruikt wordt is bij `open` en `creat`. Creat[4] wordt gebruikt voor het maken van een nieuwe file, en open voor het openen van een bestaande file (en ook voor het maken van een nieuwe – creat is eigenlijk verouderd). In feite is dit, samen met de systeemaanroepen om files te verwijderen en om directories te creëren en te verwijderen, de directory service interface. Deze twee aanroepen geven bij een geslaagde operatie een z.g. *filedescriptor*, in bovenstaand voorbeeld `fd` genoemd. Dit is een interne referentie in het O.S. naar de geopende file. De andere aanroepen (die in feite naar de storage service gaan) gebruiken deze filedescriptor als referentie. In Unix is de filedescriptor een klein getal, in feite de index in een tabel waarin het O.S. de administratie van de geopende files bijhoudt. Voor elk proces is er zo'n tabel, dus elk proces heeft zijn eigen filedescriptors die hetzelfde kunnen zijn als die in een ander proces, maar dan niet dezelfde file hoeven aan te duiden. De filedescriptors zijn dus niet de inodes!!

Een aantal filedescriptors wordt voor standaard doeleinden gebruikt:

0 standaard invoer
1 standaard uitvoer
2 standaard error

[4]Toen aan de oorspronkelijke ontwerpers van Unix een keer werd gevraagd wat ze anders zouden doen als ze het nog een keer opnieuw zouden mogen doen, was het antwoord dat ze in plaats van "creat" het woord "create" gebruikt zouden hebben.

Dit is in feite gewoon een afspraak, er is in het O.S zelf verder niets dat deze filedescriptors anders behandelt dan de andere. Alleen bij het inloggen worden deze drie verbonden met een terminal of window waarin ingelogd wordt, en de diverse shells gebruiken ze voor I/O-redirectie. Dit wordt in het volgende hoofdstuk behandeld (sectie 5.2).

Tenslotte geven we nog wat informatie over de (parameters van de) systeemaanroepen:

manier	hierin staat of de file gelezen en/of geschreven moet worden, wat er moet gebeuren als de file niet bestaat, e.d.
mode	protecties voor het geval de file nieuw aangemaakt moet worden
buffer	pointer naar geheugen waar de data staat of moet komen
nr_bytes	hoeveelheid die gelezen of geschreven moet worden
nr	hoeveelheid die echt gelezen of geschreven is. Als bij lezen `nr` $\neq$ `nr_bytes` dan is er iets fout. Bij lezen betekent dit dat we aan het eind van een file zijn, of bij een toetsenbord dat er eerder een RETURN ingedrukt is.
lseek	verschuift de plaats waar we aan het lezen of schrijven zijn naar een andere locatie in de file. Offset geeft aan hoeveel, vanwaar geeft aan of we rekenen vanaf het begin, eind of de huidige positie. Opgeleverd wordt de nieuwe positie (de huidige positie kun je krijgen door 0 bytes vanaf de huidige positie te springen).
status	geeft aan of de operatie geslaagd is. De meeste systeemaanroepen geven -1 of een andere negatieve waarde als resultaat als er een fout optreedt.

Onder MS-DOS en MS Windows zijn er soortgelijke systeemaanroepen, zij het dat onder MS Windows de filedescriptors (die worden daar filehandles genoemd) willekeurige waarden hebben, en er meer parameters zijn. Het idee is echter hetzelfde.

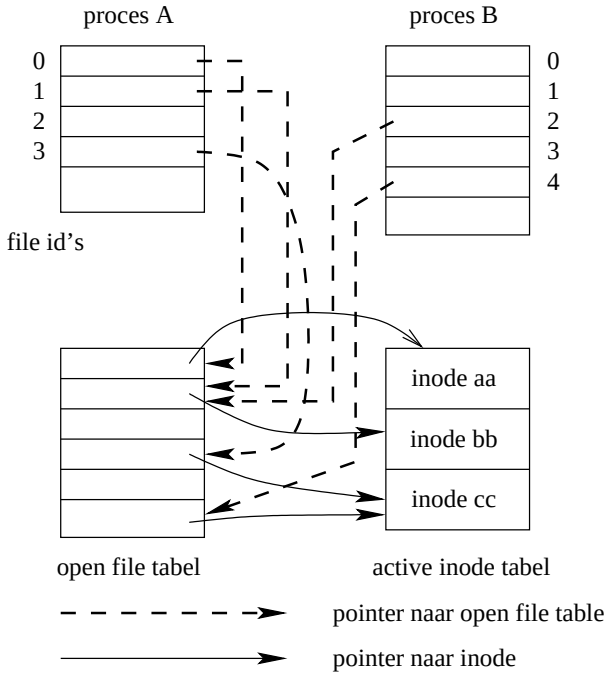
De datastructuren die Unix bijhoudt voor open files zien we in figuur 4.10 op de pagina hiernaast

Voor elke file die geopend is, heeft het O.S. een kopie van de inode in het geheugen. Hierdoor kan snel de plaats op de schijf gevonden worden. Verder is er een ‘open file tabel’ die voor elke open file extra informatie bevat, die niet in de inode staat omdat deze verandert: bijvoorbeeld de positie in de file (van lseek), een aanduiding of het proces dat de file open heeft erin mag schrijven of lezen, e.d. In figuur 4.10 op de rechter pagina heeft proces A de file “aa” open op filedescriptor 0, “bb” op 1 en “cc” op 3. Proces B heeft file “bb” op filedescriptor 2 en “cc” op 4. Zowel “bb” als “cc” zijn dus “geshared”. Er is echter een belangrijk verschil: filedescriptor 1 van proces A en 2 van B gaan via dezelfde entry van de open file tabel naar de inode van “bb”, maar filedescriptor 3 van A en 4 van B gaan via verschillende entries naar de inode entry van “cc”.

Dit heeft tot gevolg dat proces A en B voor “bb” ook een gezamenlijke filepositie hebben. Als A dus iets leest uit “bb”, en B leest daarna, dan leest B op de plaats waar A opgehouden was. Net zo met schrijven. Dit is bijvoorbeeld heel zinvol als twee processen samen iets in een file willen schrijven en de informatie moet niet door elkaar heen raken.

Bij de file “cc” hebben proces A en B elk hun eigen positiepointer, dus als ze bijvoorbeeld allebei lezen, dan zijn die operaties onafhankelijk van elkaar.

De manier waarop dit verschil tot stand komt is als volgt: Elke ‘open’ of ‘creat’ aanroep (of een andere vergelijkbare operatie) creëert een nieuwe entry in de open file tabel. Dit is de normale manier waarop processen files benaderen. Het is echter mogelijk om in een proces een duplicaat van een filedescriptor te maken: de twee filedescriptors wijzen dan naar dezelfde open file tabel entry. Ook is



Figuur 4.10: Datastructuren voor open files

het mogelijk dat een proces een filedescriptor "erft" van een ander proces; in dat geval hebben ze een gemeenschappelijke open file tabel entry. In het volgende hoofdstuk komen we hier verder op terug.

Het O.S. moet in de open file tabel entry een reference count bijhouden, zodat de entry opgeruimd kan worden als alle refererende filedescriptors gesloten zijn. Bovendien zal een zichzelf respecterend O.S. de betreffende inode ook een extra reference geven zolang de file nog geopend is. Anders kan het gebeuren dat een programma met een file bezig is en dat via een ander programma de file weggegooid wordt. Wat zou er dan moeten gebeuren bij een read of write opdracht? Anderzijds kan hierdoor een file ten onrechte blijven bestaan als de machine plotseling uitgezet wordt en de enige referentie was via de open file tabel. De file bestaat dan nog wel, maar heeft geen naam meer!!

Als voorbeeld van de behandelde systeemaanroepen geven we hier een simpel programmaatje om een file te kopiëren: Het programma is niet erg robuust, want het controleert op geen enkele fout. Maar als er geen fouten optreden wordt de file inderdaad gekopiëerd. argv[1] en argv[2] zijn de filenamen van de oorspronkelijke file en de kopie.

```
#include <fcntl.h>
#define BUFSIZ 1000
main(int argc, char *argv[])
{
  char buf[BUFSIZ];
  int infile, outfile, n;

  infile = open(argv[1], O_RDONLY);
  outfile = open(argv[2], O_WRONLY|O_CREAT, 0644);
```

```
    do {
      n = read (infile, buf, BUFSIZ);
      if (n>0) write (outfile, buf, n);
    } while (n>0);
    close(infile);
    close(outfile);
  }
```

De `open(argv[2], O_WRONLY|O_CREAT, 0644)` is hetzelfde als `creat(argv[2], 0644)`.

4.6 Gebufferde I/O

Er zijn nogal wat situaties waarin de invoer teken voor teken bekeken moet worden, of uitvoer in kleine brokjes gegenereerd wordt. Aan de invoerkant is dit bijvoorbeeld wanneer we tekst invoeren en we naar een newline teken moeten zoeken. Als we voor elk teken een systeemaanroep (`read`) zouden doen, dan wordt het programma verschrikkelijk langzaam. Een systeemaanroep moet door zoveel lagen van het O.S. heen dat er vaak duizenden instructies voor uitgevoerd worden. Het is daarom beter om per systeemaanroep een flinke hoeveelheid bytes heen- of weer te schuiven. In het programma kunnen dan op simpele wijze, met pointer- of arraymanipulaties de tekens individueel behandeld worden. We spreken dan over gebufferde I/O, omdat tussen het behandelen en de systeemaanroepen buffers zitten.

Het is natuurlijk mogelijk om voor elk individueel programma zelf iets dergelijks te verzinnen, maar dat is nauwelijks effectief. Elke programmeertaal heeft hiervoor echter i.h.a. standaard voorzieningen en die van de taal C zijn zo bekend geworden (onder de naam “stdio”= standaard I/O) dat ze *bijna* als onderdeel van het O.S. beschouwd kunnen worden. **Maar let even goed op**: stdio wordt geleverd als *bibliotheek*. De instructies maken deel uit van het gebruikersprogramma, en worden niet in het O.S. zelf uitgevoerd. Anders zou het effect ervan weg zijn. Want dan zou je telkens door de langzame systeemaanroep interface van het O.S. moeten en dat wilden we nu net voorkomen! Bekijk nog een figuur 3.1 op pagina 60 waar het woord bibliotheek staat en zie dat dat boven de streep staat die de gebruikersprogramma’s scheiden van het O.S. Vul zelf figuur 4.7 op pagina 72 aan.

Een extra voordeel is dat de stdio bibliotheek onafhankelijk is van het operating system, dus voor C programma’s is dat meestal de aanbevolen manier voor I/O.

Gebufferde I/O:
Een bibliotheek die I/O systeemaanroepen in grote blokken doet en een programma de gelegenheid geeft om de informatie toch op efficiënte wijze teken voor teken te bewerken.

Files in stdio worden voorgesteld door een pointer naar een *struct* waarin o.a. staan: het adres van de buffer, de grootte ervan, een pointer voor het vullen resp. leeghalen, de filedescriptor en nog wat van dit soort informatie. Voor het gebruik ervan hoeven we niet te weten hoe dit in elkaar zit, er is een type FILE gedefinieerd en we gebruiken een pointer hiernaar (FILE*). In sommige literatuur worden deze “files” worden ook “streams” genoemd.

De belangrijkste functies zijn de volgende:

```
file = fopen (filepath, manier)
nr = fread (pointer, grootte, aantal, file)
nr = fwrite (pointer, grootte, aantal, file)
char = getc(file)
status = putc(char, file)
status = fseek (file, offset, vanwaar)
position = ftell(file)
status = fclose (file)
status = fflush(file)
```

`file` is hierbij altijd iets van type `FILE*`. `fread` en `fwrite` hebben als parameters een pointer (bijv. naar een array), de grootte van een element en het aantal elementen. `getc` en `putc` zijn voor het lezen/schrijven van één teken, i.h.a. is dit niet veel meer dan wat simpele pointer operaties. `ftell` levert de file positie op die later weer in `fseek` gebruikt kan worden.

Een probleem bij output van streams naar bijv. een beeldscherm kan zijn dat de output nog niet op het scherm verschijnt, omdat de buffer nog niet vol is. Daarvoor is de `fflush` operatie. Deze "leegt" de buffer door een echte `write` te doen.

Hieronder volgt het filekopieerprogramma met gebruik van gebufferde I/O, en weer zonder de vereiste foutcontroles. Het moet opgemerkt worden dat dit programma iets langzamer zal lopen dan het programma dat direct `read` en `write` opdrachten doet, omdat in dit programma de bytes van de file van de ene buffer naar de andere gekopieerd worden terwijl in het vorige programma slechts één buffer gebruikt werd. Aan de andere kant is het onderstaande programma meer operating system onafhankelijk.

```
#include <stdio.h>
main(int argc, char *argv[])
{
  FILE *infile, *outfile;
  int ch;

  infile = fopen(argv[1], "rb");
  outfile = fopen(argv[2], "wb");
  while ((ch = getc(infile)) != EOF)
    putc (ch, outfile);
  fclose(infile);
  fclose(outfile);
}
```

4.7 Locking

Een file kan in het algemeen door meerdere processen in een O.S. geopend worden. Het is dus een *shared resource* en als zodanig zijn er meestal voorzieningen nodig om gemeenschappelijke toegang te reguleren. De belangrijkste operaties zijn lezen en schrijven en dus treedt het zogenaamde probleem van *readers* en *writers* op: we willen niet dat er meer dan één proces (programma) tegelijk veranderingen in de file aanbrengt, anders kan de ene verandering de andere teniet doen. Evenzo is het vaak niet

gewenst wanneer er een leesopdracht tegelijk met een schrijfopdracht plaatsvindt. Daarentegen zitten verschillende leesopdrachten elkaar niet in de weg. Bijna alle O.S's hebben hiervoor voorzieningen in de vorm van een *lock*: Er kan aangegeven worden dat een file *gelockt* moet worden, hetzij met een *shared lock*, hetzij met een *exclusive lock*. Een shared lock kan samen met andere shared locks opereren, maar een exclusive lock kan niet samen met andere locks.

Shared resources:
Een resource (file of ander onderdeel van het O.S) dat verschillende onderdelen van een systeem **tegelijk** willen gebruiken.

Lock:
Een mechanisme waardoor toegang tot een bepaalde resource tijdelijk beperkt kan worden.

Exclusive lock:
Een lock die geen andere locks tegelijkertijd op dezelfde resource toestaat.

Shared Lock:
Een lock die op dezelfde resource gelijktijdig met andere shared locks kan functioneren.

We kunnen een lock te zetten op een gehele file (we spreken dan van *file locking*). Maar het is niet altijd nodig om een hele file te locken. Wanneer slechts een deel van de file gelezen of geschreven wordt, dan is het voldoende om dat gedeelte van de file te locken en we spreken dan van *record locking*. Op deze manier zijn er minder conflicten te verwachten. Zowel op Unix systemen als op MS Windows zijn beide mogelijkheden aanwezig, hoewel oudere Unix systemen wel eens alleen de locking op gehele files hebben.

File locking:
Een lock waarbij een gehele file als gelockt object genomen wordt.

Record locking:
Een lock waarbij een deel van een file (een *record* genoemd) als gelockt object genomen wordt.

onder Unix geldt meestal dat *file locking* niet verplicht is, dus een programma kan een file openen zonder lock terwijl een ander programma er een exclusive lock op heeft staan. We zeggen dan de we *advisory locks* hebben, d.w.z. de lock adviseert, maar verplicht niet. Bij *record locking* is het meestal wel verplicht. Helaas zijn er in het verleden verschillende systemen tot ontwikkeling gekomen, waardoor dit niet op alle Unix systemen op dezelfde manier gedaan wordt.

Bij file locking kan de operatie eventueel gecombineerd worden met het openen en sluiten (*unlock*) van de file, voor record locking ligt het meer voor de hand om dit apart te doen, of eventueel te combineren met de lees- en schrijfopdrachten. Unix en MS Windows hebben verschillende ideeën hierover, zie hiervoor sectie 7.3 op pagina 126.

4.8 Synchrone en asynchrone I/O

Een I/O operatie zal i.h.a niet direct volledig uitgevoerd kunnen worden. Bij file I/O kan de disk cache ervoor zorgen dat het programma direct verder kan, wanneer we een write doen en er is ruimte in de

cache. Bij een read operatie kan dat wanneer de gevraagde gegevens in de cache aanwezig zijn. Maar anders moet het programma wachten.

Bij in- en uitvoer via een modem, netwerkverbinding, of van het toetsenbord is er zelfs niet aan te geven hoe lang het maximaal kan duren. In die gevallen zou het nuttig kunnen zijn om het programma intussen iets nuttigs te laten doen.

Als we niet eens weten waar de invoer vandaan kan komen, wordt het nog moeilijker. Als we het voorbeeld uit de inleiding nemen, waarbij Netscape een document ophaalt, dan kan tijdens de invoer vanaf het netwerk een toets ingedrukt worden of een muisklik gegeven worden. Tijdens de input van het netwerk willen we dus ook kunnen reageren op toetsenbord en muis, en omgekeerd ook. Hiervoor moeten we I/O kunnen doen die niet wacht (blokkeert) tijdens de operatie.

We onderscheiden daarom *synchrone* en *asynchrone* I/O: synchroon wil zeggen dat het programma wacht tot elke operatie klaar is (bij een write operatie kan dat betekenen dat de data in de cache is). asynchroon wil zeggen dat de I/O operatie en het programma onafhankelijk verder kunnen gaan.

Er zijn nogal wat oplossingen voor dit probleem bedacht en we zullen er een paar bespreken zonder in al te veel details te treden. Tenslotte doet elk O.S. dit weer op zijn eigen manier.

1. Op sommige systemen is het mogelijk een I/O operatie aan te vragen met een optie die zegt dat de operatie alleen gedaan moet worden als deze zonder wachten uitgevoerd kan worden. Dus bijv. bij het lezen van een toetsenbordinvoer: alleen als er iets ingetypt is. Het programma kan dan af en toe eens kijken of er wat is, een vorm van polling dus. Wanneer de opdracht wegens gebrek aan invoer niet uitgevoerd kan worden komt hij terug met een speciale foutcode.

2. Het kan mogelijk zijn om een “asynchrone” operatie te starten waarbij de operatie wel gestart wordt, maar ook direct terugkeert naar het programma. Het programma moet dan een manier hebben om later uit te vinden of de operatie klaar is. Dit systeem wordt bijvoorbeeld in MS Windows gebruikt waarbij een van de parameters van de aanroep een datastructuur is, die het O.S. gebruikt om aan te geven of de operatie al voltooid is. Het programma kan dan later een andere systeemaanroep doen om te vragen of de operatie klaar is, of te wachten tot het zover is. Bij MS Windows kan hierbij ook gebruik gemaakt worden van het standaard “message” mechanisme, waarbij door allerlei gebeurtenissen boodschappen naar het programma gestuurd worden (zoals ook muis-kliks e.d.).

   Sommige Unix systemen hebben de mogelijkheid om een functie aan te laten roepen, bijvoorbeeld wanneer er een toets ingedrukt wordt.

3. Op Unix systemen is er een systeemaanroep `select()`, die als parameters rijtjes filedescriptors heeft (zowel een rij read- als een rij write-filedescriptors). De `select` wacht tot er op minstens één van de filedescriptors een I/O operatie van de bijbehorende soort gedaan kan worden zonder te wachten. Dus als er bij de read descriptors een input gedaan kan worden die zonder wachten de data teruggeeft, of op de write descriptors een output die niet hoeft te wachten, dan stopt de `select` met wachten en geeft terug op welke filedescriptors de I/O dan zonder wachten kan gebeuren. Er is ook nog een extra parameter die aangeeft hoe lang maximaal gewacht mag worden. Als deze op 0 gezet wordt, en er is geen operatie die onmiddellijk kan gebeuren, dan komt de systeemaanroep direct terug, waardoor we weer een vorm van polling hebben. Als we dus het voorbeeld nemen dat we zowel van het toetsenbord als van de muis invoer verwachten en ook nog van een netwerkverbinding dan nemen we deze drie filedescriptors in de read parameter op.

Het moet opgemerkt worden dat lees- en schrijfopdrachten op een file die op schijf staat niet altijd erg zinvol zijn wanneer ze asynchroon gebeuren. Het kan namelijk verwacht worden dat deze nooit lang duren. Anders is het wanneer locking gebruikt wordt. In dat geval kan de lock opdracht namelijk wel lang duren. Dus daarvoor zijn asynchrone varianten wel nuttig.

Synchrone I/O:
Een I/O operatie waarbij het programma wacht met verder gaan tot het O.S. de operatie uitgevoerd heeft.

Asynchrone I/O:
Een I/O operatie waarbij het programma direct verder gaat nadat het O.S. de opdracht ontvangen heeft. Het programma kan verder werken terwijl het O.S. bezig is met het afmaken van de opdracht.

4.9 Memory-mapped files

In een systeem met virtueel geheugen (paginering) en een disk cache hebben we nu twee manieren om pagina's (blokken) geheugen van- en naar een schijf te transporteren: de paginering en de I/O. Waarom zouden we deze niet combineren? Er is in elk programma dat draait (in het volgende hoofdstuk zullen we dat een proces noemen) al een deel van de virtuele adresruimte dat verbonden is met een file, namelijk de instructies die uitgevoerd worden. Die komen van een file die uit een compiler komt, in MS Windows `iets.exe` genoemd. Deze file wordt vaak een *executable* (een vertaald programma) genoemd. We kunnen dit idee van het koppelen van een file en een stuk virtueel geheugen simpel uitbreiden door een programma de gelegenheid te geven een deel van zijn virtuele adresruimte te verbinden met een willekeurige file. We spreken dan van een *memory-mapped* file[5]. Als die verbinding eenmaal gelegd is hoeft het programma geen read systeemaanroep meer te doen om gegevens uit de file te gebruiken. In plaats daarvan gebruikt het programma gewoon het juiste adres in dat stuk virtueel geheugen. Dus bijvoorbeeld als virtueel adres 100000-110000 verbonden wordt met een file van 10000 bytes, en je wilt byte 300 van de file hebben, dan hoeft het programma alleen maar byte 100300 te adresseren. Let op: Als het de eerste keer is na het verbinden dat je iets in dit stuk virtueel geheugen doet, dan krijg je een page fault (sectie 1.9.4 op pagina 40). Het O.S. zal hierop reageren door genoeg pagina's van de file in het fysieke geheugen te plaatsen, waarna het programma kan doorgaan en de data uit de file kan gebruiken. Evenzo als we iets in de file willen wijzigen: we veranderen de bijbehorende geheugenlocaties en de pagineringssoftware van het O.S. zal het op een gegeven moment terugschrijven naar de file.

Memory-mapped file:
Het mechanisme om een file te gebruiken door via de paginering een verbinding te maken tussen een stuk virtuele adresruimte en een file of een gedeelte van een file.

Op deze manier is het ook mogelijk gegevens uit te wisselen tussen twee processen: als ze beide dezelfde file "mappen" dan zal een wijziging in het ene proces direct te zien zijn in het andere (omdat het O.S maar één kopie van de betreffende pagina in het geheugen heeft). In figuur 4.11 op pagina 84 kun je hiervan een voorbeeld zien.

In Unix is het eenvoudig om een memory-mapped file te benaderen:

[5] Let er even op dat dit niet verward moet worden met het begrip memory-mapped I/O uit sectie 1.5.

```
fd = open (filenaam, ...);
adres = mmap (address, lengte, protectie, vlaggen, fd, offset);
.... bewerken ....
munmap (adres, lengte);
close (file);
```

address	het verlangde virtuele adres waar de mapping moet plaatsvinden. Als hier 0 wordt meegegeven zoekt het systeem zelf een geschikt adres op
lengte	hoeveel bytes van de file gemapt moeten worden
protectie	vertelt of er read, write of execute permissie gegeven moet worden
vlaggen	geeft o.a aan of de file geshared moet worden, of dat er een privé mapping moet plaatsvinden (het is zelfs mogelijk om wijzigingen privé te laten zijn – met copy-on-write) en of de ruimte mag groeien
fd	filedescriptor van een open file die gemapt moet worden
offset	welke byte van de file als eerste gemapt moet worden
adres	waar de file uiteindelijk gemapt wordt (het systeem mag een anders adres gebruiken dan wordt opgegeven als dat beter uitkomt (in vlaggen is het mogelijk op te geven dat het exacte adres gebruikt moet worden).

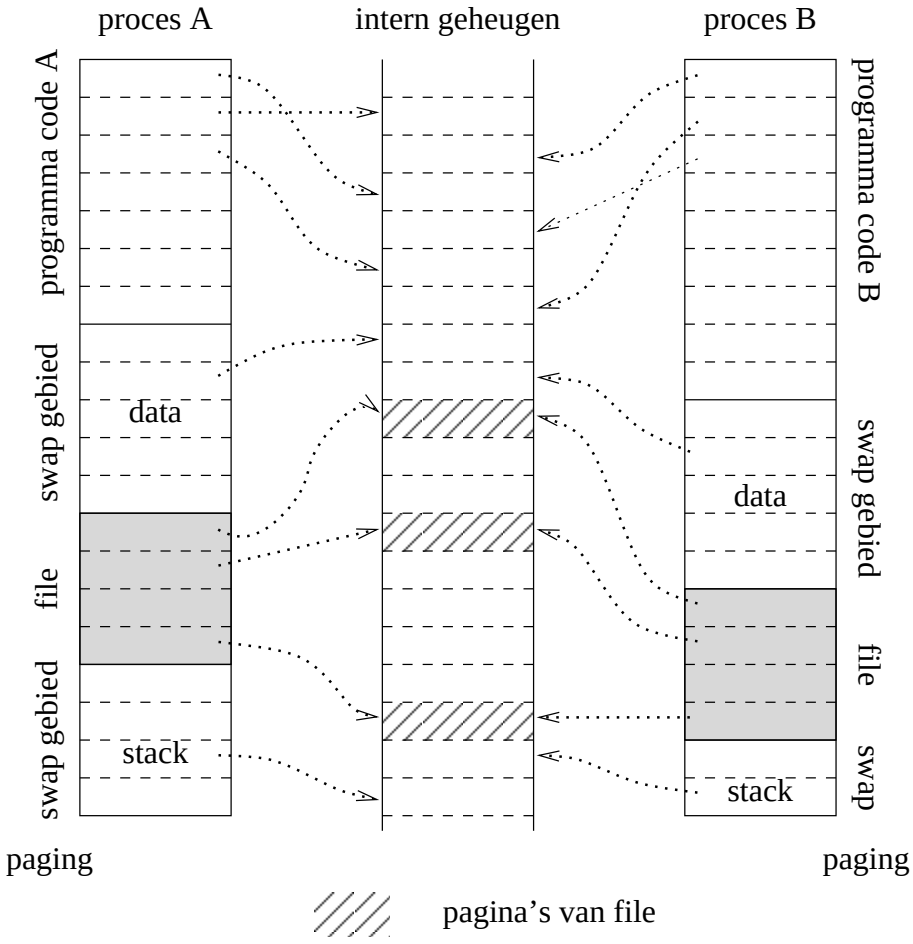
In Win32 is ongeveer hetzelfde systeem in gebruik, alleen zit er tussen het openen en het mappen nog een creatie van een speciaal FileMappingObject.

Met de komst van memory-mapped files hoeft er in het O.S. geen onderscheid meer gemaakt te worden tussen het disk-cache en het andere geheugen. Het is zelfs onwenselijk om dat de doen omdat een ouderwetse write naar een file, die tegelijk ook via memory mapping benaderd wordt, ook via de mapping zichtbaar moet worden. Een pagina moet dus nooit dubbel aanwezig zijn (in de cache en als virtuele pagina).

Een andere reden om te integereren is dat het geheugen zo efficiënter gebruikt kan worden: Als we voor de cache een apart stuk geheugen reserveren dan is die vaak te groot (waardoor er meer paginering optreedt) of te klein (waardoor het cache niet erg effectief is). Wanneer we een geïntegreerd geheugenbeheer gebruiken dan vallen de pagina's in het fysieke geheugen in de volgende categorieën:

vrij	nog niet gebruikt
vast	vaste pagina's van het O.S. die niet gepagineerd mogen worden (bijvoorbeeld page tables, de pagineringssoftware zelf)
nieuw	pagina's die door een programma gebruikt worden maar nog geen waarde hebben (bijv aangevraagde heap): deze worden meestal met nullen gevuld of bevatten troep
file	pagina's die van/naar een file gepagineerd moeten worden: instructies, memory-mapped files en **cache** pagina's
swap	pagina's die niet bij een specifieke file horen, maar gepagineerd worden van en naar een speciale swap-ruimte op de schijf: dit wordt gebruikt voor data, heap en stack pagina's, en kan ook gebruikt worden voor speciale pagina's zoals die van *pipes* (zie volgend hoofdstuk)

Van elke pagina moet bijgehouden worden hoeveel keer er een mapping naar is, waar op de schijf deze thuishoort, of het een copy-on-write pagina is, en of de pagina hetzelfde is als de bijbehorende kopie op de schijf. Een pagina die veranderd is, en nog niet teruggeschreven naar de schijf is *dirty*, en zal t.z.t. teruggeschreven moeten worden (tenzij het bijv. een data pagina is en het programma stopt



Figuur 4.11: Memory mapped file in twee processen

voordat de pagina uitgeswapt is). Pagina's die niet dirty zijn kunnen direct voor iets anders gebruikt worden als er geheugen tekort is: ze kunnen immers altijd weer van de schijf teruggeladen worden. Dirty pagina's moeten eerst naar de schijf geschreven worden voor ze voor een ander doel gebruikt kunnen worden. Tenslotte moet het aangegeven worden als een pagina van de schijf gelezen of ernaar geschreven wordt. In dat geval moet er even niets mee gebeuren omdat de inhoud dan onzeker is.

4.10 Netwerk File Systemen

Het komt tegenwoordig nogal eens voor dat de meeste files die gebruikt worden in een systeem op een centrale plaats worden opgeslagen (op een z.g. *file server*, terwijl de gebruikers op een werkstation of een PC werken. Deze laatste wordt dan gebruikt voor tijdelijke file ruimte, en soms voor de files van het O.S. zelf, maar niet voor toepassingsprogramma's en gebruikers files. Het O.S. of beter gezegd het filesysteem deel ervan moet toegang tot deze files herkennen en de opdrachten doorsturen naar de file server. In principe kunnen hiervoor dezelfde methoden gebruikt worden als in 4.3 genoemd zijn, op Unix dus bijv. met het mount commando. Alleen staat er nu geen schijf-partitie, maar een netwerkprogramma aan de andere kant.

Netwerk Filesysteem:

Een filesysteem waarbij de files op een andere plaats in het netwerk opgeslagen kunnen zijn dan waar ze gebruikt worden.

File server:

Een computer die speciaal gebruikt wordt op files op te slaan ten behoeve van andere computers in het netwerk.

Bij het gebruik van zo'n netwerk file systeem zijn er echter een paar speciale problemen:

4.10.1 Scheiding tussen directory service en storage service

Bij het gebruik van een NFS (netwerk filesysteem) ligt het voor de hand om ons af te vragen of dit niet een geschikte gelegenheid is om de scheiding tussen directory service en storage service hier door te voeren: het zou bijvoorbeeld mogelijk zijn om de directory service op het werkstation of de PC (de *cliënt*) te zetten en de storage service op de file server. Een groot voordeel hiervan is dat hiermee hetzelfde filesysteem op verschillende manieren benaderd kan worden, bijvoorbeeld op de Unix manier, de MS Windows manier en de Macintosh manier. Het probleem dat zich dan onmiddellijk voordoet is, hoe je dan de vertaling maakt van het ene systeem naar het andere. In principe zijn hiervoor wel oplossingen aan te dragen, waar we hier echter niet verder op zullen ingaan.

Een belangrijker probleem is dat wat we eerder in dit hoofdstuk hebben geconstateerd: Hoe weten we wanneer we een file kunnen weggooien? Immers een file zal weggegooid moeten worden wanneer er in geen enkele directory service een verwijzing meer naar is. En wanneer de directory services losgemaakt worden van de storage service, dan weet de storage service dat niet meer.

De oplossing hiervan is niet zo moeilijk: Voeg aan de storage service twee opdrachten toe die de reference count van een file verhogen, resp. verlagen. De directory service moet dan bij het toevoegen van een filenaam in een directory de verhoogfunctie aanroepen, en bij het verwijderen de verlaagfunctie. Bij deze laatste aanroep kan de storage service constateren of de reference count 0 geworden is, en dan de file weglaten.

Verder betekent het natuurlijk dat de file identifiers van het systeem (in Unix: inodes), dus de interne identificatie van files binnen de storage service, heen en weer gestuurd kunnen worden tussen de directory service en de storage service. Het is mogelijk om deze FID's geheel binnen het O.S. te beheren, maar anderzijds zou het ook mogelijk zijn om gebruikersprogramma's deze dingen zelf in handen te geven. Gebruikersprogramma's zouden dan een soort eigen directory service kunnen opzetten. Je moet dan denken aan speciale gebruikersprogramma's zoals database systemen, of WWW services. Voor een "gewoon" programma is dit minder voor de hand liggend. In deze gevallen is dan het gebruik van *capabilities* als FID een uitstekende keus.

4.10.2 Stateless servers

Bij een "gewoon" filesysteem (dus niet via een netwerk), bevat het O.S. op elk moment alle informatie over de geopende files, dus bijvoorbeeld hoeveel files er open zijn, hoe vaak elke file geopend is, de filepointer voor elk proces dat de file open heeft etc. Bij een netwerk file systeem is deze informatie verdeeld over het O.S. van de cliënt, en de server. Dit maakt dat de server een critische plaats in het systeem inneemt: Als de server door een storing uitvalt, dan zitten alle cliënten die een file geopend hebben in de problemen. Als de server een snelle computer is (en dat zijn servers meestal) dan kunnen we de server wel herstarten, maar dan zijn al deze essentiële gegevens weg. Er zit dan niets anders op dan allen programma's op de cliënten die een file van de file server geopend hadden ook maar te laten stoppen. Het is duidelijk dat dit niet een erg wenselijke oplossing is.

Een beter systeem is om alle essentiële informatie niet in de server maar in de cliënten te laten bewaren. De cliënt moet dan bij elke opdracht aan de server alle relevante informatie meesturen, dus: de volledige filenaam, de positie in de file, etc. Als de server nu ermee stopt en snel genoeg weer opgestart wordt, dan kan de server uit de meegestuurde informatie precies weer reconstrueren wat er gedaan moest worden. Alleen wanneer een cliënt ermee ophoudt is er een probleem, maar dan alleen voor de betreffende cliënt, niet voor de andere.

Deze aanpak wordt een "stateless" server genoemd, omdat de server geen "toestand" bijhoudt. Om niet voor elke lees- of schrijfopdracht weer de file te moeten openen zal de server in de praktijk natuurlijk zoveel mogelijk informatie toch bijhouden, maar dit is dan niet essentieel voor het afhandelen van de opdrachten van de cliënten, maar alleen een verbetering van de snelheid: deze informatie functioneert dan in feite als een soort cache.

Stateless server:
Een server die geen informatie bijhoudt over de toestand waarin zijn cliënten zich bevinden tussen hun opdrachten in.

Er is echter één probleem dat niet op deze manier af te handelen is, namelijk het locken van files of delen ervan. Hiervoor is het namelijk nodig dat er *globale* informatie wordt bijgehouden, d.w.z. informatie die niet bij één cliënt hoort, maar over alle cliënten samen gaat. Het is dus niet mogelijk *stateless* locking te doen. Het in de cliënten bijhouden van deze informatie gaat ook niet echt goed, want als de server na een *crash* weer opgestart wordt, dan weet de server niet wie alle cliënten zijn. En een cliënt die een file gelockt heeft kan best wel een weekend lang niets van zich laten horen.

Mogelijke oplossingen van dit probleem, zijn dan bijv:

1. Toch een stuk toestand bewaren, maar dan op een zodanige manier dat dit een herstart kan overleven. Mogelijkheden hiervoor zijn bijv. een stuk geheugen dat met accu's tegen stroomuitval

beschermd wordt (dit is een dure oplossing), of het opslaan van deze informatie op schijf (wat nogal vertragend kan werken).

2. Als er een manier is om een boodschap naar alle mogelijke cliënten te sturen, dan kan de server na het opstarten naar alle cliënten de vraag sturen welke files of delen ervan zij gelockt hebben. Wanneer alle cliënten een antwoord gegeven hebben dan kan de server deze informatie reconstrueren. Cliënten die geen antwoord geven worden geacht ook gestopt te zijn en dus geen files meer gelockt te houden.

3. Een lock operatie zou een tijdsduur kunnen hebben waarin deze maximaal geldig is. Een client die dan langer wil locken moet de lock operatie binnen deze tijdsduur vernieuwen. Voor de meeste programma's is dit onhandig te programmeren en dit is dus niet een aantrekkelijke manier.

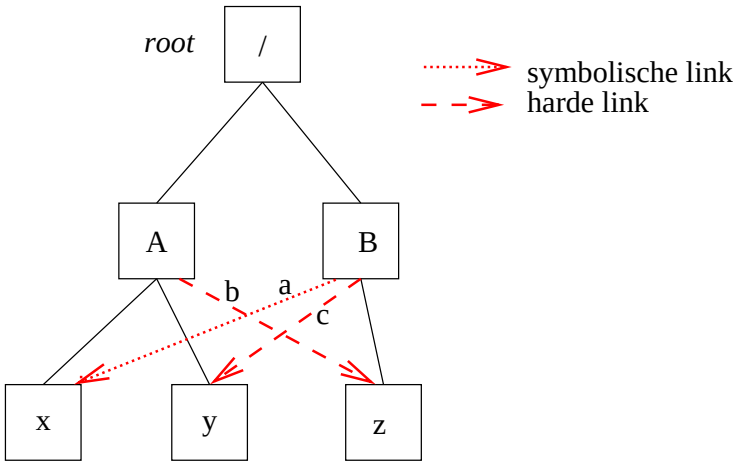
4. De simpelste oplossing is om te zeggen dat file locking niet gegarandeerd wordt, maar dit is, zeker in een zakelijke omgeving, meestal niet een acceptabele oplossing.

Let erop dat het opsturen van lock informatie tegelijk met bijv. de lees- en schrijfopdrachten, geen oplossing is (waarom niet?).

4.11 Opgaven

1. Is er een verschil tussen de FID uit figuur 4.1 en de filedescriptor uit sectie 4.5? Zo ja, wat is het verschil? Zo nee, leg uit waarom ze hetzelfde zijn.

2. Bekijk de directory structuur in figuur 4.12 en maak een inode tabel zoals in sectie 4.1.3.



Figuur 4.12: Directory structuur

3. Stel we hebben in een filesysteem met links een file met naam *x*. We maken nu een harde link naar *x* onder de naam *y*. Het commando 'mv a b' verandert de naam van a naar b. Het commando 'rm a' verwijdert de naam a en ook de file zelf als er geen harde links meer zijn.

   (a) Beschrijf het effect van de volgende twee commando's: (A) 'mv *x z*' (B) 'rm *x*'.

(b) Doe hetzelfde als we in plaats van een harde link een symbolische link hadden gemaakt.

4. Beschrijf de voor- en nadelen van drive letters zoals in MS Windows ten opzichte van mount points zoals in Unix.

5. Waarom kan het probleem van een crashende stateless fileserver niet opgelost worden door bij iedere lees- en schrijfopdracht mee te sturen welk deel van de file door de cliënt gelockt is?

Hoofdstuk 5

Processen

Processen zijn de eenheden waarin programma's uitgevoerd worden in een O.S. Een proces is een "executie van een programma". Als we een programma gestart hebben, bijvoorbeeld door op een icoon te dubbelklikken of door een commando in te typen, dan *loopt* het programma daarna. Andere termen die hiervoor gebruikt worden zijn: het programma *draait*, het `runt`, het wordt *uitgevoerd* of *geëxecuteerd*. Allemaal verschillende woorden voor hetzelfde. Wat er gebeurt is dat dan een nieuwe kopie van dit programma in het geheugen geladen wordt, en de instructies van dit programma worden uitgevoerd. Deze kopie en alles wat met het uitvoeren ervan te maken heeft, noemen we een proces. Een reden om dit te onderscheiden van het programma op zich is dat in huidige O.S's er best twee verschillende kopieën van een programma actief kunnen zijn, bijvoorbeeld voor verschillende gebruikers. Of voor dezelfde gebruiker. Start maar eens twee keer het programma "Notepad" op MS Windows op. Je kunt dit vergelijken met een toneelstuk waarvan best twee uitvoering tegelijk bezig kunnen zijn (in verschillende theaters). Bij programma's spreken we dan dus niet over een uitvoering maar over een *proces*.

Proces:
Een onafhankelijke uitvoering (executie) van een programma.

Een andere reden is dat het begrip *proces* belangrijk is, is het feit dat zo'n proces een aantal hulpbronnen (resources) van de machine in beslag neemt (bijvoorbeeld geheugen), en dat het O.S. dat moet administreren zodat dit na het beëindigen van het proces weer vrijgegeven kan worden. Daarom is het belangrijk bij te kunnen houden waar de verschillende resources bij horen. Een proces is dus ook een administratieve eenheid voor het O.S.

In het algemeen bestaat een proces uit:

- een stuk programmacode dat uitgevoerd wordt
- een hoeveelheid geheugen waarin de code uitgevoerd wordt en waarin de data, stack e.d. opgeslagen worden
- een program counter (register dat aangeeft welke instructie uitgevoerd wordt), en andere machine registers
- eventuele andere resources, zoals open files, exclusieve toegang tot apparatuur, netwerkverbindingen e.d.

- gegevens over de gebruiker die het proces gestart heeft, toegangsrechten, prioriteit t.o.v. andere processen en andere administratieve gegevens van het O.S.

5.1 Het creëren van processen

Processen worden normaliter gestart door andere processen. Het proces dat de actie onderneemt wordt vaak het *ouderproces* genoemd en het nieuw gestarte proces het *kindproces*[1]. Ook als je op een icoon dubbelklikt is er een proces actief dat deze muisklik interpreteert als een verzoek om een proces te starten.

Ouderproces:
Een proces dat een ander proces opstart.
Kindproces:
Een proces dat door het ouderproces gestart is.

Het eerste proces wordt meestal gecreëerd bij het opstarten (booten) van het O.S. In een Unix systeem is dit vaak een proces dat het programma "init" draait. Dit proces houdt het toetsenbord, inkomende communicatielijnen en netwerkverbindingen in de gaten en start nieuwe processen op wanneer er zich ergens een nieuwe gebruiker aandient. Ook start het init-proces diverse andere processen op, zoals printer spoolers, netwerk-processen, window manager, etc.

Op MS-DOS wordt als eerste proces i.h.a. het programma "COMMAND.COM" opgestart, een simpele commando interpreter. Op MS Windows wordt het programma opgestart dat het initiële beeldscherm laat zien.

Elk O.S. heeft een systeemaanroep om nieuwe processen te starten; echter de manier waarop dit gebeurt is nogal verschillend.

- Op MS-DOS is er een systeemaanroep Pexec(file) die een nieuw proces start waarin het programma uit file uitgevoerd wordt. Het proces dat deze aanroep doet wordt tijdelijk gestopt, want MS-DOS kent geen multitasking (dat wil zeggen: er kan niet meer dan één proces tegelijk actief zijn). Wanneer het gestarte proces stopt (via de exit systeemaanroep), dan wordt het aanroepende proces weer doorgestart. De processen die op enig moment in het geheugen zijn, vormen dus strikt een LIFO (stack), en alleen het proces op de top van de stack is actief. Op de bodem van de stack bevindt zich de SHELL. Vanwege deze simpele structuur is het ook niet nodig dat de identiteit van processen beschikbaar moet zijn aan andere processen. Na afloop van de Pexec is het gecreëerde proces dus beëindigd en het aanroepende proces kan opvragen wat de exit code van het gestarte proces was. Normaliter geeft dit een aanduiding of het proces goed of fout beëindigd is.

Multitasking:
De mogelijkheid in een O.S. om meer dan één proces tegelijk actief te hebben.

- In Win32 is er een systeemaanroep CreateProcess, die een nieuw proces opstart met als opstartprogramma de file die door een van de parameters aangegeven wordt. In een administratieblok

[1] Er kunnen natuurlijk vele ouder- en kindprocessen zijn. Er wordt meestal niet over grootouders en kleinkinderen gesproken.

(dat ook als parameter meegegeven wordt) schrijft deze systeemaanroep o.a. een process handle, die weer gebruikt kan worden om te wachten tot het proces klaar is.

- In het Unix systeem zijn de functies van het creëren van een nieuw proces en het specificeren van het programma dat uitgevoerd moet worden van elkaar gescheiden. Om een nieuw proces te creëren wordt de systeemaanroep `fork()` gebruikt. Deze aanroep maakt een nieuw proces, maar dit proces is een kopie van het aanroepende proces. Het voert dus dezelfde instructies uit en krijgt ook een kopie van de data (variabelen) en de stack. De instructiecode zal i.h.a. geshared worden, d.w.z. er is slechts één kopie van in het geheugen. De data en stack wordt niet gedeeld, zowel het ouderproces als het kindproces hebben hun eigen kopieën, waarin ze veranderingen mogen aanbrengen. Omdat beide processen na deze aanroep de instructies volgend op de `fork()` aanroep uitvoeren moet er een manier zijn om te beslissen of we in het ouder- of het kindproces verder gaan, want waarschijnlijk zullen we in beide verschillende dingen willen doen. Dit gebeurt doordat de aanroep een verschillend resultaat oplevert: in het ouderproces wordt de proces-id van het kindproces opgeleverd, en in het kindproces de waarde 0. De proces-id (pid) is een getal dat uniek is voor ieder proces. We zien dan vaak ook een soortgelijke code:

```
childpid = fork();
switch (childpid) {
    case -1:
            # error code
            break;
    case 0:
            # kind proces code
            break;
    default:
            # ouder proces code
    }
```

of iets soortgelijks met if statements.

De tweede functie die Unix heeft (`exec`), heeft als doel om in een bestaand proces een ander programma (executable) te laten uitvoeren. Dit creëert dus geen nieuw proces, maar gooit de instructies, data en stack van het bestaande proces weg, en laat dit proces met een schone lei beginnen met het uitvoeren van een nieuw programma (waarvan de filenaam als parameter meegegeven wordt). Hetzelfde effect als de MS-DOS of MS Windows operaties kan in Unix verkregen worden door de `fork()` in het kindproces direct te laten volgen door een `exec`.

Fork/exec:
Het systeem (op Unix-achtige O.S.) waarbij het creëren van een nieuw proces (`fork`) gescheiden is van het aangeven welk programma uitgevoerd moet worden (`exec`).

De voordelen van de fork/exec methode zijn:

1. Met de `fork()` kan een kindproces gestart worden dat de data van het ouderproces ter beschikking heeft. Op deze manier kan dus gemakkelijk een grote hoeveelheid gegevens van een proces

overgeheveld worden naar een nieuw proces. In MS-DOS is dit veel moeilijker: de gegevens moeten naar een file geschreven worden of er moeten (semi-illegale) trucs uitgevoerd worden om het geheugen van het ouderproces vanuit het kindproces te bekijken. In MS Windows kan dit gedaan worden dmv. stukken shared memory, maar dit is ook ingewikkelder. Anderzijds kan op Unix na de `fork()` geen gewijzigde informatie meer uitgewisseld worden via dit mechanisme omdat elk proces een eigen kopie krijgt (het is dus geen shared memory).

2. Tussen de `fork()` en de `exec()` kunnen wijzigingen aangebracht worden in de omgeving, bijvoorbeeld in de environment variabelen en in de standaard files (stdin, stdout en stderr). Dit is de gebruikelijke techniek voor het implementeren van I/O redirection, bijvoorbeeld in de shell. In MS-DOS en MS Windows moet dit gebeuren door de oude situatie voor het starten van het proces te kopieren en na afloop weer terug te zetten, wat iets ingewikkelder is.

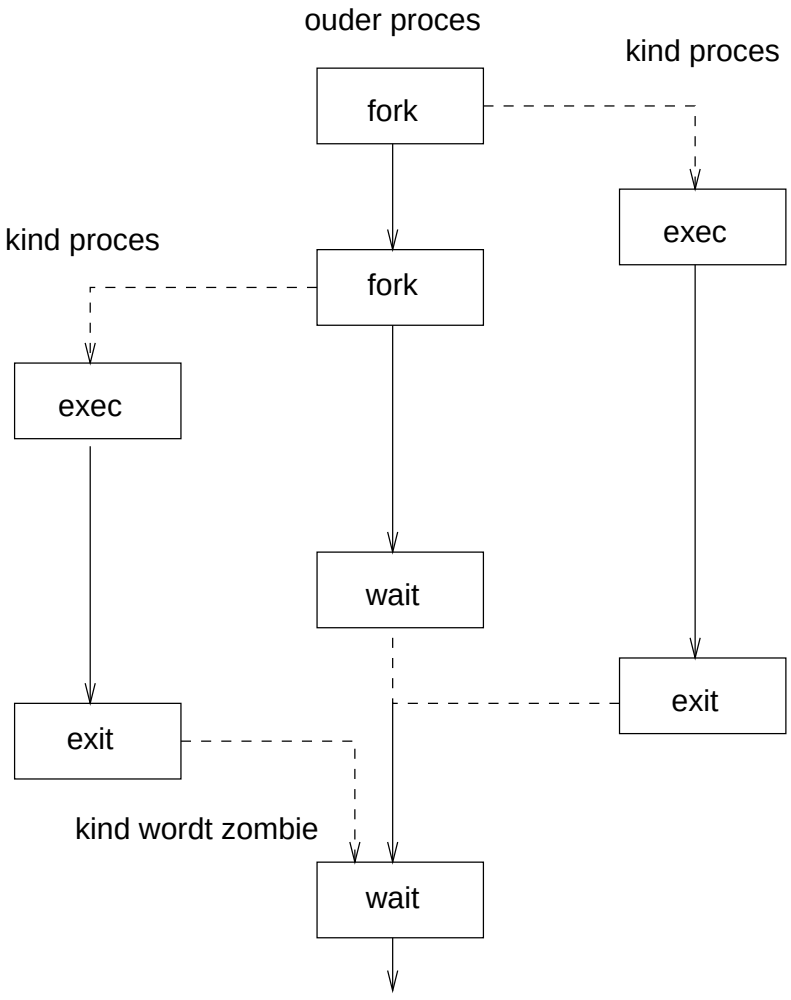
Er zijn echter ook nadelen verbonden aan de fork/exec methode. Het belangrijkste is wel dat wanneer de `fork()` direct gevolgd wordt door een exec, er een overbodige kopie is gemaakt van de data en de stack van het ouderproces. Om hier omheen te komen, heeft men in de BSD (Berkeley Software Distribution) versie van Unix indertijd een aparte systeemaanroep `vfork()` ofwel *virtual fork* geintroduceerd, waarbij niet echt een kopie gemaakt werd, maar via virtueel geheugen de data en stack geshared werd. Dit is gevaarlijk omdat een van beide processen de data of stack van de ander kon veranderen. De eis bij het gebruik van deze operatie was dat het kindproces direct daarna een `exec()` zou doen. Het was zelfs niet toegestaan om de `fork()` binnen een functie te doen en de `exec()` erbuiten omdat de terugkeer uit deze functie de stack zou veranderen en dus ook de stack van het ouderproces zou aantasten. Het O.S. gaf bij deze aanroep de garantie dat het kindproces eerder aan de beurt zou komen dan het ouderproces, omdat anders het ouderproces de stack van het kindproces zou kunnen vernielen. Al met al was dit een dubieuze manier om de tekortkomingen van het systeem te repareren.

In moderne Unix systemen wordt een betere manier toegepast, gebaseerd op *copy-on-write*. Dit is een memory management techniek die we al in sectie 1.9.3 gezien hebben, waarbij de kopie (van de data en de stack) in eerste instantie niet echt gedaan wordt, maar waarbij beide processen alleen via virtueel geheugen toegang tot dezelfde data hebben. Dit deel van het virtueel geheugen wordt in eerste instantie alleen read-only gemaakt, zodat beide processen de gegevens kunnen gebruiken maar niet veranderen. Wanneer een van beide processen een wijziging wil maken, wordt dit door het O.S. opgemerkt en dan wordt van de betreffende pagina een extra kopie gemaakt, en krijgen beide processen in hun virtueel geheugen voor deze pagina een eigen referentie die nu read/write is. Op deze manier worden alleen kopieën gemaakt die echt nodig zijn. Wanneer de fork() (bijna) onmiddellijk gevolgd wordt door een exec(), is de data en stack in het kindproces helemaal niet meer nodig en wordt er dus niets gekopieerd. Maar ook als het kindproces bijvoorbeeld eerst nog terugkeert uit een functie, dan hoeft alleen een kopie gemaakt te worden van de top van de stack.

Het wachten op een kindproces kan in Unix met de systeemaanroep wait(). Deze systeemaanroep wacht tot er een kindproces beëindigd is en levert dan de pid van dit proces op. Er is ook een waitpid() aanroep waarmee gewacht kan worden tot een proces met een specifieke pid beëindigd is. Het kan natuurlijk gebeuren dat een kindproces beëindigd is voordat het ouderproces een wait() of waitpid() kan doen. Om te voorkomen dat de informatie verloren gaat (of nog erger: dat de pid alweer opnieuw gebruikt wordt), zal een beëindigd proces waarvoor nog geen wait() gedaan is, blijven voortbestaan als een z.g. *zombie*-proces. Het O.S. verwijdert dan alle systeembronnen die het proces had (zoals geheugen, open files, etc.) en houdt alleen een klein stukje administratie over, dat net genoeg is om een evt. komende wait() of waitpid() te behandelen. De zombie-processen nemen wel plaats in in de

proces-tabel, dus het is zaak dat een proces dat veel andere processen kan opstarten (bijvoorbeeld de shell), regelmatig wait() of waitpid() uitvoert. Wanneer een ouderproces stopt terwijl er nog kindprocessen actief zijn, dan krijgen deze kindprocessen het *init* proces (proces nr. 1) als ouder, omdat er anders niemand zou zijn die hun zombie-status ongedaan kan maken. Het init-proces voert, naast het wachten op nieuwe logins, voornamelijk wait() aanroepen uit. In figuur 5.1 zien we een voorbeeld van de genoemde systemcalls.

Zombie-proces:
Een restant van een beëindigd proces dat nog een minimale hoeveelheid informatie bevat over de reden tot beëindiging.



Figuur 5.1: Unix proces management calls

5.2 I/O redirection

Onder Unix, Win32 en verscheidene andere systemen is het mogelijk om een proces op te starten op zodanige manier dat de in- en uitvoer van dit nieuwe proces gekoppeld zijn aan files (of andere I/O kanalen) die door het ouderproces bepaald worden. Dit kan omdat in deze systemen een proces een aantal file descriptors voor standaard invoer, uitvoer en errormessages gebruikt wordt (zie 4.5). Wanneer een kindproces gecreëerd wordt, dan "erft" dit deze filedescriptors. Wanneer het ouderproces nu vlak voor het opstarten deze filedescriptors koppelt aan andere files dan kan het kindproces deze gebruiken *zonder dat er in het kindproces speciale actie voor ondernomen moet worden*.

I/O redirection:
Het omleiden van een filedescriptor door een ouderproces (voor het starten van een kindproces) zodat die in het kindproces de gewenste waarde heeft.

Het ouderproces moet dan wel eerst de oude inhoud van filedescriptor 0 tijdelijk opbergen, zodat het later weer teruggezet kan worden. Dit kan met de systeemaanroep `dup`, die een kopie maakt van de filedescriptor en die kopie oplevert als resultaat. Na het starten van het kindproces wordt in het ouderproces dan de filedescriptor 0 weer teruggezet met de systemcall `dup2`:

```
fd = open (‘‘filenaam’’, O_READ);
saveid = dup(0);
dup2 (fd, 0);
close (fd);
... start nu het kind ...
close (0);
dup2 (saveid, 0);
close (saveid);
```

In de Unix shells kan het iets gemakkelijker: Als de shell bijvoorbeeld het commando 'sort <invoer' krijgt dan moet eerst een `fork` aanroep gedaan worden, en daarna een *exec* aanroep. Het opzetten van de redirection gebeurt dan tussen de `fork` en de *exec* aanroep, en wel als volgt (voor input redirection):

```
fd = open (‘‘filenaam’’, O_READ);
dup2 (fd, 0);
close (fd);
```

De `dup2` aanroep maakt een kopie van de filedescriptor `fd` naar filedescriptor 0. De vroegere inhoud van filedescriptor 0 gaat hierbij verloren (het systeem doet eerst een `close` indien nodig). Maar omdat we dit tussen de `fork` en de `exec` doen geeft het niet: dit heeft geen effect op het ouderproces want het gebeurt al in het kindproces, en de oude filedescriptor is in het kindproces niet meer nodig.

Voor output redirection doen we iets soortgelijks met file descriptor 1. Ook het koppelen van programma's via een pipe verloopt op soortgelijke wijze, zie hiervoor het volgende hoofdstuk.

5.3 Scheduling

In een multi-tasking systeem zijn er meestal meer processen die de CPU zouden kunnen gebruiken. Sommige processen zullen misschien niet verder kunnen omdat ze moeten wachten, bijvoorbeeld op

een toetsenbordaanslag, op een disk operatie of op een netwerkverbinding. Maar de andere processen willen iets uitrekenen. Als we slechts één processor hebben moet er dus een keuze gemaakt worden. Ook wanneer er meerdere processoren aanwezig zijn, en het aantal processen dat de CPU kan gebruiken is groter moet er een keus gemaakt worden. Het maken van deze keus heet *scheduling*. Er zijn verschillende algoritmen mogelijk. Behalve het feit dat er geen algemeen beste algoritme bekend is, hebben we ook te maken met het feit dat verschillende situaties verschillende eisen stellen aan een scheduler. Hier zijn wat voorbeelden:

- Wanneer we een computer hebben die, zonder dat er iemand achter het beeldscherm zit, een hoop rekenwerk moet doen (bijvoorbeeld 's nachts), dan is het doel om het totale computersysteem optimaal te gebruiken, zodat de totale hoeveelheid werk in zo kort mogelijke tijd afgewerkt wordt. Of zodat zoveel mogelijk werk in een vastgestelde tijdslimiet gedaan wordt. Dit soort werk wordt vaak *batch-verwerking* genoemd.

- Wanneer iemand achter een beeldscherm zit te werken is het doel om zo snel mogelijk op de acties van de gebruiker te reageren. Hierdoor kan de persoon die de computer gebruikt zijn of haar werk zo efficiënt mogelijk doen. Dit soort werk wordt *interactief* genoemd.

- Wanneer een proces een audio- of videoclip moet weergeven is het heel belangrijk dat het proces op het juiste moment de CPU krijgt om het volgende stukje weer te geven. Anders krijgen we hakkelende en stotterende beelden of geluid. Ook wanneer een computer ingezet wordt als besturing van bijvoorbeeld een olieraffinaderij, een vliegtuig of een chemische fabriek is het belangrijk dat de processen binnen vastgestelde tijdstippen kunnen reageren omdat er anders misschien fatale fouten optreden. Dit soort werk noemen we *real-time*.

Scheduling:
Het bepalen welk proces (of thread[a]) aan de beurt komt.
Scheduler:
Het onderdeel van het O.S. dat de scheduling verzorgt.

[a] Threads worden verderop in dit hoofdstuk behandeld.

Al deze soorten werk hebben eigen eisen die aan de scheduler gesteld worden. En deze eisen zijn vaak met elkaar in conflict. Zo is bijvoorbeeld batch verwerking bijna tegengesteld aan interactief of real-time werk. Een scheduler die goed werkt voor interactief- of real-time werkt zal i.h.a. slechte resultaten geven voor batch werk en omgekeerd. Vergelijk dit bijvoorbeeld met het verkeer: om veel mensen te transporteren is het openbaar vervoer (batch verwerking) het beste, maar om persoonlijk zo snel mogelijk van A naar B te gaan is een auto handiger (interactief). Alleen wanneer veel mensen dit doen, leidt het tot files. Aan de andere kant kost het voor een individu bijna altijd meer tijd om met het OV ergens naar toe te gaan.

In veel gevallen zal een O.S. een mengsel van processen van verschillende soort hebben. De samenstelling kan van moment tot moment veranderen. Het O.S. moet dan zijn scheduling proberen aan te passen aan de samenstelling van moment tot moment. Meestal wordt er bij een proces echter niet aangegeven wat voor soort het is. Het O.S. moet dan op grond van het gedrag van het proces gissingen maken over de soort van het proces. Bijvoorbeeld:

- Een proces dat langdurig de CPU gebruikt zonder te wachten op in- of uitvoer kan tot de categorie batch-processen gerekend worden.

- Een proces dat regelmatig wacht op invoer van de gebruiker (toetsenbord of muis) kan tot de categorie interactieve processen gerekend worden.

Een veel gebruikte manier om op deze manier de scheduling te sturen is om ieder proces een *prioriteit* te geven. De processen met de hoogste prioriteit krijgen dan eerder de CPU dan processen met een lagere prioriteit. Real-time processen zullen dan de hoogste prioriteit krijgen, interactieve een lagere en batch processen de allerlaagste.

Om elk proces een redelijke kans te geven om de CPU te krijgen is het belangrijk dat een proces niet te lang de CPU kan vasthouden. We spreken van *preemptive* scheduling als het O.S. op elk willekeurig moment kan beslissen dat het proces dat de CPU heeft, deze zal moeten opgeven. Dit kan bijvoorbeeld gebeuren als er een interrupt binnenkomt die tot gevolg heeft dat een proces met een hogere prioriteit niet langer hoeft te wachten. De scheduler kan dan besluiten het lopende proces tijdelijk stop te zetten en het andere proces voor te laten gaan. Een andere manier die gebruikt wordt is om een actief proces een maximale tijdsduur (*time slice*) te geven, waarna een ander proces een kans krijgt. De simpelste manier om dit te doen is alle processen in een circulaire lijst op te nemen, en ze een voor een een vaste time-slice te geven. Na afloop van de time-slice, of wanneer het proces moet wachten op een externe gebeurtenis wordt het volgende proces gedraaid. Dit heet *round-robin* scheduling. Het tegenovergestelde van *preemptive* scheduling is *non-preemptive* of *co-operative* scheduling. Hierbij wordt een proces nooit de CPU onvrijwillig afgenomen, maar alleen op bepaalde momenten, of wanneer het proces dit zelf aangeeft. Bijvoorbeeld op Win16 wordt alleen scheduling gedaan wanneer het lopende proces een systeemaanroep doet. Een proces dat bijvoorbeeld een loop uitvoert waarin alleen gerekend wordt en geen I/O, kan daarmee het hele systeem "vastzetten". Dergelijke processen moeten dus regelmatig in hun loops een systeemaanroep doen, evt. eentje die niets essentieels doet. Het zal duidelijk zijn dat dit gevaarlijk is: een klein foutje in een van de processen kan het hele systeem beïnvloeden. Een voordeel van non-preemptive scheduling is dat er in het systeem minder voorzieningen getroffen hoeven te worden voor het beschermen van kritische secties, omdat de plaatsen waar parallelliteit kunnen optreden nauwkeurig gedefinieerd zijn. Aan de andere kant kan dit slordig programmeren bevorderen.

Non-preemptive scheduling:

Een schedulingssysteem waarbij een proces[a] zelf aangeeft wanneer het zijn beurt wil afstaan. Ook **cooperative scheduling** genoemd.

Preemptive scheduling:

Een schedulingssysteem waarbij een proces onvrijwillig zijn beurt moet kunnen afstaan.

Timeslice:

Een maximale tijd dat een proces mag doorgaan voor het zijn beurt moet afstaan. Is alleen van toepassing bij preemptive scheduling.

Round-robin scheduling:

Scheduling waarbij elk proces op zijn beurt een timeslice krijgt.

[a] In plaats van 'proces' mag hier ook 'thread' gelezen worden.

In feite is de scheduling van een MS Windows systeem nog iets ingewikkelder: Er kunnen in een MS Windows systeem een aantal *virtuele machines* actief zijn. “Onderin” het O.S. zit een virtuele machine manager die deze machines schedulet. Dit gebeurt met preemptive scheduling. Elke MS-DOS “box” die onder MS Windows draait, draait in een eigen virtuele machine, en MS-DOS boxen worden dus preemptive gescheduled. Onder Win32 draait elk 32-bits programma ook in een eigen virtuele machine. De 16-bits programma’s, en onder Win16 alle programma’s draaien echter met zijn allen in één virtuele machine, waarbinnen weer non-preemptive scheduling plaatsvindt. Deze programma’s zijn ook niet tegen elkaar beschermd wat betreft geheugentoegang, want dat gebeurt door de virtuele machine manager.

Traditionele Unix systemen hebben op gebruikerscode niveau preemptive scheduling, maar niet in de O.S. code. In de O.S. code (de kernel) zijn er vaste punten waarop gescheduled kan worden, en wordt dus non-preemptive scheduling gebruikt. Wanneer de kernel een operatie doet waarop gewacht moet worden (bijvoorbeeld een disk in- of output) dan wordt de functie sleep() aangeroepen. Als argument wordt een pointer naar een datastructuur gegeven die iets over de gebeurtenis zegt waarop gewacht moet worden (bijvoorbeeld een buffer voor I/O). De enige punten waarop in kernel mode gescheduled wordt zijn de sleep() aanroepen. Wanneer de kernel constateert dat een gebeurtenis waarop gewacht kan worden voltooid is wordt een wakeup() aanroep gedaan met dezelfde pointer als parameter. Er kunnen evt. meerdere processen zijn die op deze gebeurtenis stonden te wachten en deze krijgen dan een kans om weer verder te gaan. De Unix kernel functioneert op deze manier in feite als een grote monitor met de sleep() en wakeup() aanroepen als de wait() en signal(). Het kan in de Unix kernel echter gebeuren dat er geruime tijd doorgebracht wordt zonder dat een sleep() gedaan wordt. Bijvoorbeeld wanneer en operatie op een directory gedaan wordt, zoals het zoeken van een file, en alle diskblokken van de directory waren al in het geheugen aanwezig, dan zal tijdens het zoeken geen sleep() gedaan worden. De tijd die een proces doorbrengt in de kernel heeft daarom niet een vaste bovengrens. Daarom is een traditioneel Unix systeem niet geschikt voor real-time werk.

5.4 Context switch en proces status

Voor elk proces moet het O.S. een hoeveelheid gegevens bewaren. Bijvoorbeeld:

- Het geheugen dat het proces gebruikt
- De files die het proces geopend heeft
- Andere resources die het proces in gebruik heeft
- De hoeveelheid CPU tijd die het proces gebruikt heeft
- De prioriteit van het proces
- ruimte om registers te saven
- waarop het proces staat te wachten

Deze informatie samen wordt ook wel eens de context genoemd. Onder Unix staat deze informatie in een grote tabel, de *proces tabel*. De index in deze tabel is in feite de *proces id* of *pid*. Het overschakelen van één proces naar een ander wordt een *context switch* genoemd. Bij een context-switch moeten

gegevens van een proces die niet een eigen geheugenlocatie hebben opgeborgen worden. Hieronder vallen bijvoorbeeld de machine registers (waaronder de program counter, de stackpointer en de conditie codes), memory management informatie, e.d. Wanneer de scheduler een ander proces uitgekozen heeft om verder te gaan worden de opgeborgen waarden van dat proces geladen zodat dit door kan gaan.

Context:
In het kader van scheduling: de informatie die het O.S. en de computer over een proces hebben.

Proces tabel:
Tabel in het O.S. met informatie over de processen.

Context switch:
Het overschakeling van één proces naar een ander door de scheduler, waarbij de context wordt omgezet van het ene proces naar het andere.

Tijdens de levensduur van een proces zal het door verschillende stadia gaan. Als we ervan uitgaan dat een proces pas als zodanig bij het O.S. bekend is wanneer alle belangrijke informatie in het geheugen geladen is dan begint het proces in de toestand *runnable*, d.w.z. het kan de CPU gaan gebruiken zodra deze toegewezen wordt. Er kunnen meerdere processen in de toestand runnable zijn, en zodra dit het geval is zal de scheduler een keuze uit hen moeten maken. Een van deze processen krijgt de CPU toegewezen (of bij een multi-processor systeem zoveel als er processoren zijn), en deze heeft/hebben de toestand *running*. Een proces beëindigt de toestand running doordat

1. de scheduler de CPU afneemt, ten gevolge van een interrupt of het aflopen van een time-slice (wat ook het gevolg is van een interrupt, namelijk de klok-interrupt). Het proces gaat dan weer naar toestand runnable.

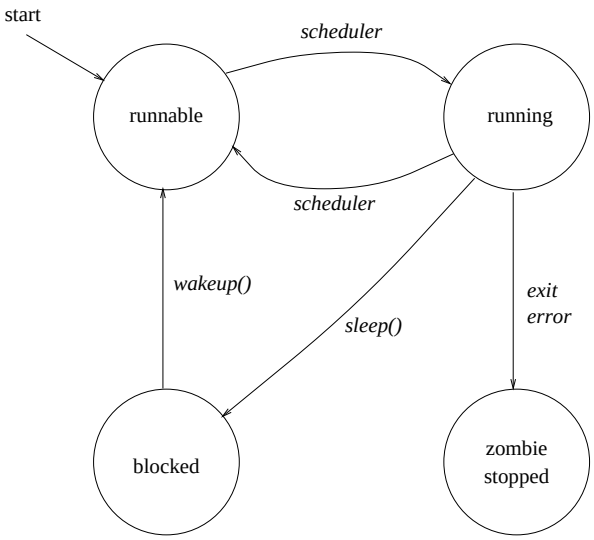
2. het proces een systeemaanroep doet die ergens op moet wachten. Het proces komt dan in de toestand *blocked*. Een proces kan uit de blocked toestand geraken doordat de gebeurtenis waarop gewacht wordt, voltooid is: het komt dan weer in de toestand runnable, en kan t.z.t weer running worden.

3. het proces eindigt, en komt dan in de toestand *stopped*. In principe verdwijnt het dan, maar bijvoorbeeld in Unix kan het nog een tijdje als *zombie* blijven bestaan. In de figuren 5.2–5.3 op de rechter pagina zien we een schema met de overgangen tussen de verschillende toestanden.

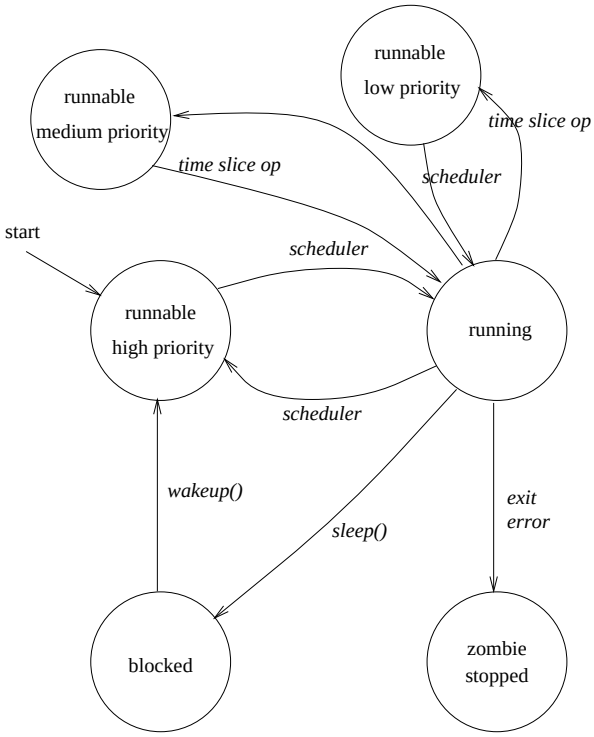
5.5 Threads

De belangrijkste manier om concurrency te verkrijgen in een modern O.S. is door middel van het starten van verschillende processen. Het O.S. zorgt via de scheduling dat elk proces op zijn tijd aan de beurt komt, en wanneer één proces moet wachten kan een ander de CPU gebruiken.

Helaas is dit niet altijd voldoende om een goede prestatie te leveren. Neem als voorbeeld het in de inleiding gepresenteerde draaien van Netscape. Wanneer Netscape moet wachten op een hoeveelheid data vanuit het netwerk kan er geen andere activiteit **binnen** Netscape plaatsvinden. Toch kan dit nuttig zijn, want Netscape zou meer dan één verbinding kunnen onderhouden, en tevens de muis en



Figuur 5.2: Simpele scheduler



Figuur 5.3: Scheduler met verschillende prioriteiten

het toetsenbord in de gaten kunnen houden. Als het O.S. asynchrone I/O voorzieningen levert dan kunnen deze gebruikt worden om een deel van het probleem op te vangen, maar erg gemakkelijk is dit niet.

Een ander voorbeeld is het gebruik van een *server* in een netwerk, bijvoorbeeld een file server of een HTTP server. De server krijgt verzoeken binnen en moet dan meestal een aantal I/O operaties doen. Tijdens het wachten op de I/O operaties kan de server geen andere verzoeken van cliënten afhandelen. Dit heeft tot gevolg dat de server een bottleneck wordt. Op Unix systemen wordt dit vaak ondervangen door voor elk verzoek een nieuw proces op te starten (de oorspronkelijke server doet een `fork`), maar dit heeft ook weer nadelen: Het starten van een nieuw proces kost relatief veel tijd en de verschillende processen kunnen geen gemeenschappelijke administratie voeren, omdat hun geheugens gescheiden zijn. Deze administratie kan belangrijk zijn, bijvoorbeeld om een cache van de laatst gevraagde informatie bij te houden. Of bijvoorbeeld bij een database server, om locking informatie bij te houden. Hier kan wel weer gewerkt worden met speciale shared memory segmenten, als het O.S. dit levert, maar dat is ook omslachtig.

Een betere oplossing is het gebruik van z.g. *threads*, dwz. dat binnen een proces meerdere op zichzelf sequentiële executies parallel kunnen lopen. Een proces kan dan zelf dus weer uit een verzameling parallelle mini-procesjes bestaan. Deze threads worden daarom ook wel *lightweight* processen genoemd.

Thread:
Een mini-procesje binnen een proces dat onafhankelijk gescheduled wordt en gelijktijdig met andere threads kan lopen. Ook wel **light-weight proces** genoemd.

De meeste moderne O.S's hebben tegenwoordig deze mogelijkheid. Een proces kan dan gewoon als een puur sequentieel proces draaien, en dit is ook de standaard methode als geen speciale actie ondernomen wordt. We kunnen ook zeggen dat het proces uit één thread bestaat. Het proces kan daarna zelf nieuwe threads opstarten en weer laten stoppen. In dat geval zeggen we dat het proces *multithreaded* is.

De eenheid van scheduling (voor de CPU) is bij een multithreaded systeem niet meer het proces maar de thread. Het O.S. kan dan dus ook op elk moment besluiten om van de ene thread naar de andere over te schakelen, zowel binnen hetzelfde proces als binnen een ander proces. Daarom moet elke thread ook een eigen stack hebben. De stacks zullen hoogstwaarschijnlijk allemaal binnen dezelfde (virtuele) adresruimte van het proces zitten. Het is denkbaar dat het O.S. telkens bij het omschakelen van een thread de virtuele adressen omschakelt naar de stack van de nieuwe thread, maar dit kan problemen geven omdat er in globale variabelen pointers naar de stack kunnen zitten. Bovendien kost dit weer tijd. Het geheugenbeheer bij threads wordt dus moeilijker, omdat er ruimte voor de threads gereserveerd moet worden: als we hiervoor te weinig nemen, dan kan een stack van één van de threads vol raken; als we teveel nemen, dan kan er misschien te weinig geheugen zijn voor alle stacks en de globale variabelen samen.

De threads delen dezelfde instructies, namelijk die van het "hoofdprogramma", en ook de globale data (dat was namelijk een belangrijke reden om threads te hebben). Het laatste betekent dat er ook synchronisatieprimitieven moeten zijn om kritieke secties af te schermen zoals semaforen of conditievariabelen[2]. Deze voorzieningen moeten door het O.S. geleverd worden of minstens aan het O.S bekend zijn. het O.S. moet namelijk een andere thread schedulen als er een moet wachten tot een kritieke sectie betreden kan worden. Anderzijds zullen we proberen zoveel mogelijk van de administratie

[2] zie college Gedistribueerd Programmeren

van de threads binnen het proces te houden, omdat het aanroepen van het O.S. ook een dure operatie is.

Het is mogelijk om bij een O.S. dat geen threads kent, zelf een bibliotheek te maken die de functionaliteit van threads imiteert. Het schedulen van de threads kan dan niet door het O.S. gebeuren, dus het O.S. schedulet dan processen, maar een proces kan dan zijn eigen threads schedulen. Dit geeft een slechtere prestatie dan wanneer het O.S. het doet, omdat een proces met veel threads in dit systeem niet vaker aan de beurt komt dan een proces met weinig threads. Het scheelt vooral op een systeem met meer dan één CPU, want als het O.S. threads levert, dan kunnen twee threads van hetzelfde proces allebei een CPU krijgen, terwijl bij "imitatie-threads" er binnen een proces altijd hoogstens één thread een CPU heeft, zelfs als er geen enkel ander proces iets te doen heeft.

Bij het "zelf" schedulen moet er op gelet worden dat een "pseudo-thread" niet op een I/O operatie staat te wachten, terwijl er andere threads zouden kunnen doorgaan. daarom wordt in zo'n systeem meestal asynchrone I/O gebruikt, bijvoorbeeld onder Unix met de `select` aanroep. Dit werkt alleen handig als het runtime systeem van de taal waarin het programma geschreven is deze voorzieningen levert. Of wanneer we de standaard bibliotheek van het runtime systeem vervangen door een aangepaste versie waarin dit gebeurt. Een voorbeeld van het zelf schedulen van verschillende threads is het programma Netscape op Win16 en op Unix systemen die geen threads hebben. Een ander voorbeeld is Java op zulke systemen.

Bij een multithreaded O.S. is het begrip proces dus niet meer voor de eenheid van scheduling zoals in een single-threaded O.S. Dat is de thread geworden. In sectie 5.5 kun je daarom overal in plaats van 'proces' ook 'thread' lezen. Het proces daarentegen is dan de eenheid van resource-allocatie. Een proces heeft verschillende resources zoals geheugen en open files, en deze worden gedeeld door alle threads in het proces. Een file die door één thread geopend is, kan door alle gelezen en geschreven worden. Idem voor netwerkverbindingen. Het geheugen wordt door alle threads gedeeld. En wanneer het proces een fout maakt (die dan door één van de threads uitgevoerd wordt) wordt het hele proces beëindigd, want het proces geldt als één geheel. Meestal kan het ook niet anders, want een fout in een proces betekent vaak dat de datastructuren niet meer kloppen, en de threads zijn door deze datastructuren aan elkaar verbonden. Als het proces bij een fout speciale actie wil nemen om slechts één thread te laten stoppen, dan moet het zelf de fout opvangen.

Hier volgt nog een overzicht van de belangrijkste diensten die een thread systeem moet leveren:

thread_create	maak een nieuwe thread aan (de parameters zijn o.a. de functie die door de thread uitgevoerd moet worden en de grootte van de stack)
thread_exit	stop de thread die deze opdracht uitvoert
thread_kill	breek een andere thread af
thread_join	wacht tot een andere thread stopt
mutex_lock	zet een lock op een mutex (semafoor)
mutex_unlock	geef een lock op een mutex vrij
cond_wait	wacht op een conditie variabele
cond_signal	signaleer een conditie variabele (zodat een wachtende thread door kan gaan)
cond_broadcast	broadcast op een conditie variabele (zodat alle wachtende processen door kunnen gaan)

5.6 Shared Libraries

In vorige hoofdstukken hebben we al de mogelijkheid genoemd om bibliotheken van functies die door veel programma's gebruikt worden, te delen met verschillende processen. In oudere systemen worden de functies en procedures die een programma nodig heeft, tijdens het *linken* met de specifieke code van het programma samengebonden tot één *executable* of *binary*, een file waarin alle instructies en constanten staan die het programma nodig heeft. Bij het sharen van de instructies via de memory management hardware, kunnen alleen kopieën van hetzelfde programma gedeeld worden. In moderne systemen hebben de bibliotheken de neiging om heel groot te worden, vooral wanneer we te maken hebben met grafische software, zoals user interfaces. In dat geval wordt het aantrekkelijk om ook de bibliotheken te delen tussen programma's die verder verschillend zijn. Dit kan door gebruik te maken van stukken shared memory, bijvoorbeeld met de memory-mapped file operaties. De bibliotheken worden in aparte files gezet en bij het opstarten van een programma (bijvoorbeeld de `exec` operatie in Unix) worden deze files gebonden aan het proces door de juiste *mmap* operatie te doen. De protectie die gebruikt wordt is dan alleen "execute". We spreken dan van een "*shared library*". Wanneer de bibliotheek globale variabelen gebruikt kan de linker van tevoren hiervoor ruimte reserveren in het aanroepende programma. Of als alternatief kan een deel van de bibliotheekfile als read/write worden gemapped. Omdat elk proces zijn eigen variabelen moet hebben moet wel gezorgd worden dat dit deel van de file niet geshared wordt voor schrijven. Copy-on-write is hiervoor een goed mechanisme.

Op MS Windows, waar deze shared libraries veel gebruikt worden (het grootste deel van het O.S. bestaat hieruit) wordt gesproken van DLL's (dynamic load libraries), omdat een proces niet alleen met deze DLL's gelinked wordt, maar ook op runtime dynamisch deze libraries kan laden en weer loslaten. Op Win16 kan een DLL ook gesharede data bevatten, d.w.z. dat er data structuren kunnen zijn die niet per gebonden proces aanwezig zijn, maar slechts één keer voor alle processen samen. Dit is een gevaarlijke situatie, want een fout in één proces kan de data voor alle andere vernielen. Maar onder Win16 hebben de diverse programma's toch al toegang tot elkaars data (en instructies). Onder Win32 waar elk proces in zijn eigen virtuele machine draait is deze mogelijkheid (althans voor 32-bits programma's) niet meer aanwezig. Op Unix systemen ook niet, hoewel in beide gevallen het gemakkelijk gedaan zou kunnen worden, door een `mmap` aanroep met read/write permissies en sharing. Wanneer iets dergelijks toch nodig is kan een aparte `mmap` operatie gebruikt worden.

Shared library:
Een bibliotheek, die bij het starten of tijdens het draaien van een proces via de memory management aan het virtuele geheugen van een proces gekoppeld wordt. Een shared library kan dan door diverse processen tegelijk gebruikt worden, terwijl er maar één kopie in het fysieke geheugen aanwezig is. Wordt ook wel **DLL** (**dynamic link library**) genoemd.

Er zijn een paar problemen bij het gebruik van DLL's die het O.S. moet oplossen:

- Op welk adres wordt de library gelinkt in het programma? Bij het samenstellen van de library weten we niet op welk adres een programma het gaat gebruiken. We kunnen als maker van een library wel een keus maken waaraan het programma zich moet houden, maar als een programma verschillende libraries nodig heeft en in het totale systeem er veel zijn, dan kunnen hier simpel conflicten ontstaan. Op sommige systemen wordt daarom de eis gesteld dat een shared library moet bestaan uit *position independent code* (PIC). Dit zijn instructies die op elke plaats in de virtuele adresruimte kunnen draaien, zonder dat ze aangepast hoeven te worden. Dus bij-

voorbeeld sprongopdrachten worden altijd relatief t.o.v. de instructie gedaan, en adressen in de bibliotheek worden uitgereken t.o.v. het instructie adres. Dit gebeurt dan door de compiler die hiervoor een speciale optie heeft. Deze methode wordt bij veel Unix systemen gebruikt.

Bij MS Windows wordt een andere methode gebruikt: een DLL wordt daarbij gelinkt met een voorkeurs adres, maar alle informatie om de code aan te passen aan een ander adres wordt erbij opgeslagen. Wanneer de bibliotheek op het voorkeursadres wordt aangevraagd, dan kan deze zonder meer geladen en gemapped worden. Wanneer een programma echter een ander adres opgeeft, wordt eerst de code van de DLL aangepast om op dat adres te kunnen draaien (het gaat hier natuurlijk over virtuele adressen). Als twee programma's elk een ander adres opgeven, dan kan de DLL niet geshared worden, maar moeten er twee kopieën in het geheugen komen.

- Het tweede probleem is dat een DLL van tijd tot tijd aangepast zal moeten worden, bijvoorbeeld om fouten te herstellen. De adressen van de functies die erin zitten kunnen dan veranderen, en bestaande programma's zouden ongeldig kunnen worden omdat ze naar de oude adressen verwijzen. Dit is natuurlijk ontoelaatbaar. Daarom wordt meestal een indirectie tussengevoegd: Aan het begin van de DLL wordt een lijst van adressen van functies opgenomen, en de programma's roepen de functies aan via deze adressen. Bij een aanpassing moet er alleen gezorgd worden dat de volgorde van de functies in deze lijst hetzelfde blijft. Nieuwe functies kunnen achteraan toegevoegd worden, maar er mogen geen functies weggehaald worden.

5.7 Signals en software interrupts

Signals zijn op het niveau van processen (en threads) wat interrupts zijn op het niveau van de hardware. Een signal is dus een onderbreking van een proces of thread om tijdelijk iets anders te gaan doen. De signals worden door het O.S. "afgeleverd" aan een proces, in feite als onderdeel van de scheduling. Na het afhandelen van de signal instructies wordt in het algemeen het proces voortgezet op de plaats waar de onderbreking plaatsvond.

We bespreken in dit gedeelte het signal mechanisme zoals het in Unix gebruikt wordt. Niet alle Unix systemen doen het op exact dezelfde manier (helaas) maar we bespreken hier de gemeenschappelijke karakteristieken.

Er zijn een aantal manieren waarop een signal kan ontstaan:

1. Er is een systeemaanroep `kill` waarmee een specifiek signal naar een proces of een groep processen gestuurd kan worden. Er zijn verschillende "soorten" signals die allemaal door een nummer aangegeven worden. Het nummer is één parameter, het procesnummer is een andere. Er zit natuurlijk een bescherming op: in principe kan een proces alleen een signal naar een ander proces sturen als ze dezelfde eigenaar hebben. Sommige signals mogen ook van een ouder- naar een kindproces of afstammeling gestuurd worden, of binnen een groep die bij elkaar hoort.

   Het Unix commando `kill` doet niets anders dan de bijbehorende systeemaanroep uitvoeren.

2. Sommige signals worden veroorzaakt door speciale tekens die op het toetsenbord ingetypt worden. Zo als het Control/C teken meestal het signal SIGINT (INT genoemd bij het commando) veroorzaken, en Control/Z het signal SIGSTOP. Op deze manier is het mogelijk lopende processen te onderbreken. Voor het proces maakt het verder niet uit hoe het signal er gekomen is, dus een aanroep `kill(SIGINT, pid)` heeft hetzelfde effect als Control/C.

3. Bepaalde fouten of uitzonderlijke situaties veroorzaken ook hun specifieke signals. Het verschil met de vorige oorzaken is dat hier i.h.a. de oorzaak bij het proces zelf ligt. Dit is vergelijkbaar met het verschil tussen een interrupt en een exception op hardware niveau, waar bij een exception ook meestal de oorzaak in het lopende programma zit.

   Voorbeelden van dit soort signals zijn:

   - SIGILL (Illegal instruction)
   - SIGPIPE (Het schrijven op een pipe waar de leeskant van gesloten is)

Signal:
Een seintje van het O.S. aan een proces dat ervoor zorgt dat het proces tijdelijk even iets anders gaat doen, of dat het voortijdig gestopt wordt.

Een proces kan zelf aangeven wat er met de signals moet gebeuren. Het proces kan aangeven of een signal genegeerd moet worden, dat het de standaard actie moet uitvoeren of dat er een eigen functie uitgevoerd moet worden als het signal binnenkomt. De standaard actie is verschillend voor allerlei signals: Bij sommige signals is de standaard actie om het proces te stoppen, bij andere is de standaard actie niets doen. Het proces specificeert de actie met de systeemaanroep:

```
signal (signalnummer, functie)
```

waarbij functie het adres van een gedeclareerde functie uit het proces is of een van een aantal constanten voor negeren of default. Als waarde levert signal de vorige toestand op, zodat het proces deze kan bewaren om hem later terug te zetten.

Als een functie meegegeven wordt en het gespecificeerde signal komt aan dan roept het O.S. deze functie aan met o.a. het signalnummer als parameter, zodat het makkelijker is om meer dan één signal met dezelfde functie op te vangen.

In een signal functie moet men uitkijken welke andere functies men aanroept, want niet alle functies kunnen onderbroken en weer aangeroepen worden. Functies die dat wel kunnen worden reëntrant genoemd. Een niet-reëntrant functie is in feite een functie die een kritieke sectie kent, en die dus niet parallel uitgevoerd kan worden.

Het is nogal moeilijk om deze kritieke secties met semaforen (locks) te beschermen. Omdat het niet mogelijk is een gesignaleerde functie tijdelijk te stoppen en met het hoofdprogramma verder te gaan: de gesignaleerde functie wordt op een stack uitgevoerd, en moet dus eerst terugkeren om het hoofdprogramma weer verder te kunnen laten gaan. Hetzelfde geldt voor een gesignaleerde functie die zelf weer onderbroken wordt door een ander signal. De laatste moet eerst eindigen voor de eerste weer door kan gaan (Last In First Out principe). Aan de andere kant kun je er met het ontwerpen van kritieke secties dus wel van uitgaan dat een hoofdprogramma door een signal routine onderbroken kan worden, maar omgekeerd niet. Wanneer er kritieke secties in het spel zijn, kan meestal een oplossing met extra variabelen gebruikt worden, bijvoorbeeld als de “doe operatie” in de signal routine interfereert met de kritieke sectie van het hoofdprogramma:

In het hoofdprogramma:

```
signalled = false;
```

```
incrit = true;
< kritieke sectie >
incrit = false;
if (signalled)
   < doe operatie >
```

In de signal routine:

```
if (incrit)
   signalled = true;
else
   < doe operatie >
```

5.8 Opgaven

1. Geef eens stukje Unix programma dat met behulp van fork, exec en wait of waitpid hetzelfde doet als de MS-DOS aanroep Pexec(file).

2. Geef de code voor I/O redirection zoals in sectie 5.2, maar dan voor standard output.

3. Waarom lopen er in figuur 5.2 geen pijlen van runnable naar blocked en van blocked naar running?

4. Is de scheduler in een microkernel operating system onderdeel van de kernel of een apart proces? Leg uit waarom.

5. Linux heeft threads op een andere manier geïmplementeerd dan andere operating systems: in Linux is er eigenlijk geen verschil tussen een proces en een thread. Er is in Linux een nieuwe system call: *clone()*; deze doet net zoiets als *fork()*, maar heeft een parameter die aangeeft wat het nieuwe proces gemeenschappelijk heeft met het oude. De resources die geshared kunnen worden zijn: de instructies (text), de data (stack en heap), de open files, de signal handlers en de proces-id (PID). Bij *fork()* worden alleen de instructies geshared; bij *clone()* kan met de parameter aangegeven worden welke andere onderdelen ook geshared moeten worden. Vergelijk de voor- en nadelen van deze manier van threads met de traditionele manier.

6. Wat kan er fout gaan als in de code aan het eind van sectie 5.7 de regels:

   ```
   signalled = false;
   incrit = true;
   ```

   verwisseld worden?

Hoofdstuk 6

Inter Process Communication

Onder Inter Process Communication of IPC verstaan we de mechanismes waarmee een proces met een ander proces kan communiceren. Veel operaties in een systeem zijn te groot of te ingewikkeld om door één proces uitgevoerd te worden. Dan moeten we de operatie opsplitsen in verschillende processen maar deze moeten wel met elkaar gegevens kunnen uitwisselen. Wanneer delen van een operatie uitgevoerd moet worden op verschillende knopen in een netwerk dan is het zelfs onontkoombaar dat er verschillende processen bij betrokken zijn. Je kunt namelijk niet een proces hebben dat op twee verschillende computers loopt. Ook niet met verschillende threads.

Inter Process Communication:
Communicatie tussen processen.

We zullen in dit hoofdstuk de verschillende vormen van IPC bespreken waarbij we ons zullen beperken tot die situaties waarbij de betrokken processen binnen één computersysteem vallen. Communicatie tussen processen op verschillende computers in een netwerk komt later aan de beurt.

6.1 Ouder-kind relatie

De simpelste relatie tussen processen is die tussen een ouder-proces en een kind-proces (zie het vorige hoofdstuk). Bij het starten van een kind-proces kan i.h.a. een hoeveelheid gegevens meegegeven worden als commando-argumenten (zoals bij de shell). Het kind kan deze gebruiken om zijn acties te sturen. Bij de meeste systemen kan een kind echter geen informatie teruggeven aan het ouder proces, of alleen maar een klein getalletje na afloop dat aangeeft of de operatie al dan niet gelukt is. Bovendien kan met deze methode tijdens het verdere verloop van de processen geen informatie meer uitgewisseld worden. Voor algemeen gebruik is dit dan ook ongeschikt.

6.2 Communicatie via files

Een iets flexibeler manier om gegevens tussen processen uit te wisselen is via het gebruik van files. Een proces dat informatie naar een ander proces wil doorgeven schrijft dit in een file met een vooraf afgesproken naam. Het proces dat de informatie wil hebben leest het uit deze file.

Er zijn een aantal problemen met deze aanpak:

1. Synchronisatie tussen de processen is moeilijk te verwezenlijken. Nadat het schrijvende proces de data in de file gezet heeft, moet het lezende proces hier een seintje van krijgen. Het best zou dit kunnen door met een systeemaanroep te wachten tot er data in de file is, maar de meeste O.S's hebben niet zo'n aanroep. Het ontvangende proces af en toe te laten proberen of er al data in de file staat is erg inefficiënt, vooral omdat het openen en sluiten van files kostbaar is. Bovendien kan op sommige systemen een file al geopend worden terwijl een ander proces nog ermee bezig is, en kan het ontvangende proces dus onvolledige informatie krijgen. Wanneer het systeem file locking heeft kan hier wat mee gedaan worden, maar dit is ingewikkeld. Op Unix zouden signals gebruikt kunnen worden om het ontvangende proces een seintje te geven.

2. Het openen en sluiten van files is meestal een redelijk dure operatie, en deze methode is dus niet geschikt wanneer informatie in veel kleine brokken doorgegeven moet worden. Het doorgeven van informatie terwijl de file open is, geeft problemen m.b.t. synchronisatie (file locking). Wanneer echter een proces eenmalig een grotere hoeveelheid informatie moet doorgeven kan dit een goede oplossing zijn. Veel compilers op Unix werken op deze manier, waarbij telkens een proces een deel van het werk doet, het resultaat naar een file schrijft, en daarna een volgend proces opstart dat deze file weer verder verwerkt. We spreken dan van een *multi-pass* compiler.

6.3 Communicatie via pipes

Een iets flexibeler manier van communiceren, vooral tussen ouder- en kind-processen is d.m.v. een *pipe*, zoals dit op Unix systemen gedaan wordt[1].

Een pipe is een file-achtig ding waarop geschreven en gelezen kan worden. Het functioneert als een FIFO queue. Een pipe dient dus voor eenrichtingsverkeer tussen processen.

In Unix wordt een pipe aangemaakt met de systeemaanroep pipe(pd), waarbij pd een pointer is naar een array van twee integers. Het O.S. maakt een nieuwe pipe aan, en geeft in het array pd twee filedescriptors: een voor het schrijf-einde van de pipe en een voor het lees-einde. Deze file descriptors kunnen op dezelfde manier gebruikt worden als de file descriptors voor een file. Op deze manier is de pipe echter nog niet bekend aan twee processen. Dit gebeurt doordat het proces dat de pipe gecreëerd heeft, een of meer kind-processen opstart. Deze kunnen de filedescriptors erven en dus ook hierop schrijven of ervan lezen. In een simpele toepassing wordt er alleen een kind-proces opgestart en is er communicatie tussen ouder en kind mogelijk. Het proces dat wil schrijven, sluit het lees-einde, en het proces dat wil lezen, sluit het schrijf-einde. Op deze manier kan naar keuze van de ouder naar het kind geschreven worden, of andersom. Theoretisch is het mogelijk om beide processen in de pipe te laten schrijven, en eruit te laten lezen, maar in de praktijk geeft dat aanleiding tot te ingewikkelde synchronisatie.

Pipe:
Een communicatiemechanisme in het O.S. met twee uiteinden. Wat aan het ene uiteinde erin geschreven wordt kan aan het andere einde eruit gelezen worden.

[1] Op MS-DOS en Win16 bestaan geen pipes. De MS-DOS commando-interface geeft de mogelijkheid om commando1|commando2 uit te voeren maar daarbij wordt stiekem van files gebruik gemaakt, zoals in de vorige sectie beschreven

Een andere simpele mogelijkheid is dat het ouder-proces twee kind-processen opstart, waarvan de ene het schrijf-einde van de pipe houdt en het lees-einde sluit, en het ander omgekeerd. De ouder kan dan de hele pipe sluiten en communicatie vindt alleen plaats tussen de beide kinderen (van het ene kind naar het andere).

In de shell wordt dit systeem bijvoorbeeld toegepast voor het implementeren van commando1|commando2. Hiervoor zal de shell twee kind-processen creëren (met fork()), en in het ene kind-proces het schrijf-einde van de pipe als standaard uitvoer gebruiken, en in het andere kind-proces het lees-einde als standaard invoer. Na het omzetten van de pipe einden zullen de kind-processen een exec() uitvoeren om het betreffende programma te gaan uitvoeren. Schematisch voorgesteld:

```
pipe(pd)
pid1=fork()
if (pid1 == 0)          /* kind1                                */
{  close(pd[0]);        /* sluit lees-einde                     */
   dup2(pd[1], 1);      /* kopieer schrijf-einde naar stdout    */
   close(pd[1]);        /* sluit originele schrijf-einde        */
   exec("command1");
} else
{                       /* ouder                                */
   pid2=fork()
   if (pid1 == 0)       /* kind2                                */
   {  close(pd[1]);     /* sluit schrijf-einde                  */
      dup2(pd[0], 0);   /* kopieer lees-einde naar stdin        */
      close(pd[0]);     /* sluit originele lees-einde   */
      exec("command2");
   } else
   {                    /* ouder                                */
      close(pd[0]);     /* sluit lees-einde                     */
      close(pd[1]);     /* sluit schrijf-einde                  */
   }
}
```

De opdracht dup2(fd1, fd2) maakt een kopie van de filedescriptor fd1, en maakt deze ook beschikbaar als fd2. Daarna kan dus zowel fd1 als fd2 gebruikt worden voor in- en uitvoer op dezelfde file of pipe, totdat een van de twee gesloten wordt (zoals hier ook gebeurt).

Situaties waarbij pipes een prima mechanisme vormen zijn vergelijkbaar met de multi-pass compiler: maar in plaats van files worden dan pipes gebruikt om van de ene pass naar de andere te communiceren. Het voordeel boven files is, dat beide processen tegelijk kunnen draaien, waardoor dit meer parallellisme geeft. Het lezende proces zal moeten wachten wanneer het schrijvende proces de data nog niet in de pipe heeft gezet. Omgekeerd kan het schrijvende proces nog wel even doorgaan ook al loopt het lezende proces wat achter. Echter de meeste Unix systemen beperken de hoeveelheid data die ongelezen in de pipe mag zitten, en dus zal het schrijvende proces op een gegeven moment ook moeten wachten wanneer het te ver voorloopt. Een pipe geldt dus als een eindige buffer.

Deze eigenschap maakt het in het algemeen onmogelijk of ongewenst om tweerichtingverkeer via pipes te doen. Als we bijvoorbeeld via een pipe van A naar B willen schrijven en via een tweede pipe

van B naar A dan kan er deadlock optreden onder de volgende omstandigheden: A schrijft veel data naar B en wordt geblokkeerd omdat de pipe vol zit. B leest een beetje data uit de pipe, maar genereert op grond hiervan veel data die terug moet naar A; B wordt geblokkeerd omdat de pipe terug vol zit, maar A kan nog niet de data eruit lezen omdat A nog wacht op de volle pipe naar B. Evenzo is het i.h.a. ongewenst om meer processen in een cyclische structuur met pipes aan elkaar te verbinden.

We hebben gezien dat een pipe altijd door één proces aangemaakt wordt en alleen aan andere processen doorgegeven kan worden via de ouder-kind-relatie. Dit geeft dus beperkingen op de keuze van processen die met elkaar via pipes willen communiceren: ze moeten van dezelfde voorouder afstammen, die dan van te voren de pipe gemaakt moet hebben. Het is dus onmogelijk dat twee processen zomaar besluiten om via een pipe te gaan communiceren.

Omdat dit toch een serieuze beperking kan zijn hebben moderne Unix systemen ook z.g. *named pipes*. Dit zijn pipes die al van tevoren bestaan (of via een speciale systeemaanroep gemaakt kunnen worden), en die een naam in het file systeem hebben. Een proces kan de pipe openen via deze naam, alsof het een file is, en er dan in schrijven of ervan lezen. De gegevens worden echter niet op een disk file opgeslagen maar als pipe behandeld. De processen die op deze manier willen communiceren moeten wel op een andere manier afspraken maken over welke pipe te gebruiken. Vaak worden named pipes voor vaste functies gebruikt. Stel bijvoorbeeld dat een named pipe gebruikt wordt om de namen van files door te geven aan een programma dat de printer bestuurt. Dit programma (de printer spooler), kan dan een named pipe openen met een vaste, bekende naam, en daaruit gaan lezen. Zolang er geen files aangeboden worden, wacht de spooler op de read-opdracht. Wanneer een programma een file wil aanbieden om die te laten printen dan opent dit programma de named pipe met dezelfde naam, maar met de intentie om erin te schrijven, en schrijft de filenaam erin. Daardoor zal de read-opdracht in de spooler voltooien en kan de spooler het werk doen.

Named pipes kunnen beschermd worden tegen ongeoorloofde toegang op dezelfde manier als files, dus met behulp van eigenaar, groep en andere protecties.

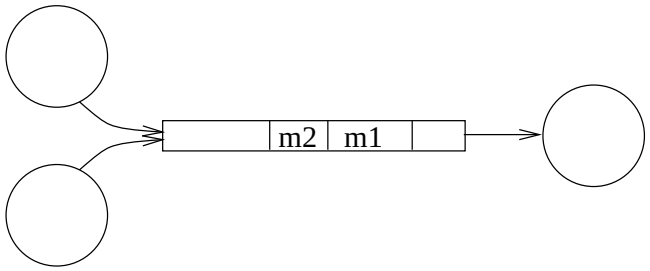
6.4 Message Passing

Het *pipe* model voor communicatie zoals het in Unix gebruikt wordt heeft een belangrijke tekortkoming: De data in de pipe wordt als een amorfe stroom bytes beschouwd waarbinnen verder geen structuur te herkennen is. We noemen dit dan een *byte stream*. Twee processen die via een pipe communiceren kunnen (en zullen i.h.a.) natuurlijk afspreken hoe de data geïnterpreteerd moet worden, maar het O.S. levert geen voorzieningen om deze informatie in de pipe vast te leggen. Een voorbeeld van zo'n afspraak zou kunnen zijn dat van elke "boodschap" de eerste twee bytes de lengte van de boodschap aangeeft. Als we alleen te maken hebben met twee processen die met elkaar communiceren kan dit voldoende zijn. Wanneer echter meer processen bij de communicatie betrokken zijn dan kan dit tot problemen aanleiding geven.

Byte stream:
Een gegevensstroom waar geen andere structuur in zit dan een rij bytes.

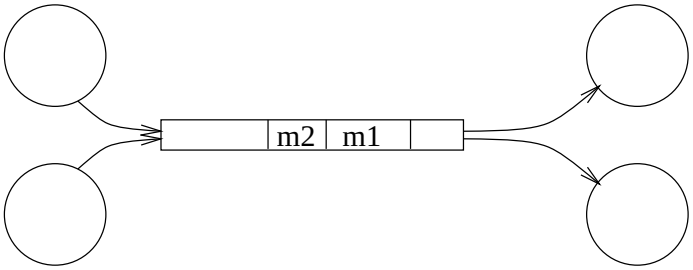
Laten we om te beginnen eens kijken naar het geval dat twee of meer processen boodschappen sturen naar een pipe, die gelezen wordt door één proces. Wat we nu moeten voorkomen dat twee boodschappen (m1 en m2) die door de twee processen worden geschreven door elkaar heen raken. Dus het

schrijven van een boodschap naar een pipe moet een ondeelbare (atomaire) actie worden. In Unix is dit inderdaad het geval, wanneer het schrijven met een enkele `write` systeemaanroep gebeurt. Wanneer we gebufferde I/O gebruiken (zie 4.6), dan is dit echter niet gegarandeerd omdat de boodschap over een buffergrens heen kan gaan, en de write opdrachten door de bufferingsfuncties per hele buffer gegeven worden. Dat maakt gebufferde I/O voor dit soort toepassingen in feite onbruikbaar.



Figuur 6.1: Twee processen die naar dezelfde pipe schrijven

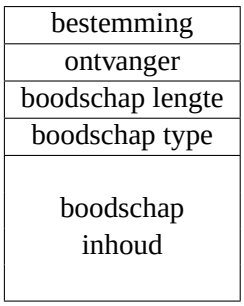
Nog erger wordt het wanneer meer dan één proces gaat lezen van de pipe. Is het vorige voorbeeld nog oplosbaar door directe `write` aanroepen te gebruiken, bij het lezen is het probleem onoplosbaar. Wat is namelijk het geval?



Figuur 6.2: Twee processen die van dezelfde pipe lezen

Om een boodschap uit de pipe te lezen moet een ontvangend proces eerst weten hoe lang de boodschap is (dit moet namelijk `read` opdracht opgegeven worden). Het proces zal dus eerst een `read` moeten doen om de lengte van een boodschap te lezen, bijvoorbeeld een read voor 2 bytes als de boodschaplengte met twee bytes aangegeven wordt, en daarna een read voor de aangegeven lengte. Dat kan dus niet als atomaire actie. Tussen deze twee read opdrachten kan een anders proces komen dat ook probeert een lengte te lezen, maar die krijgt dan de eerste twee bytes van de inhoud van de boodschap die het eerste proces had moeten lezen, waardoor de chaos compleet is. De enige oplossing is dus dat het O.S. een methode geeft om boodschappen als ondeelbare eenheden te behandelen, waarbij de lengte van de boodschap niet door de processen geïnterpreteerd wordt, maar door het O.S. in het communicatiekanaal opgenomen wordt. Het O.S. kan dan ook nog extra informatie opnemen, zoals de identiteit van de ontvanger en verzender. Zo'n boodschap zou er dan bijvoorbeeld kunnen uitzien als in figuur 6.3 op de pagina hierna.

We zullen daarom in de rest van deze sectie aannemen dat we een O.S. hebben dat voorzieningen levert om expliciet boodschappen te versturen en te ontvangen waarbij de grenzen tussen de boodschappen door het O.S. bijgehouden worden, in plaats dat dit overgelaten wordt aan de processen. Er zullen dan



Figuur 6.3: Voorbeeld van een boodschap layout

ook SEND en RECEIVE aanroepen zijn om dit te realiseren. Er zijn echter binnen dit model nog vele variaties mogelijk, waarvan we er een aantal zullen bespreken. Hier volgt eerst een lijst:

- Wachten zendende en ontvangende processen op elkaar of niet?
- Kennen de communicerende processen elkaars identiteit?
- Geeft het ontvangende proces aan van welke afzender de boodschap moet komen?
- Wordt een boodschap naar één of meer processen tegelijk verzonden?
- Kan de ontvanger onderscheid maken tussen verschillende soorten boodschappen?
- Wacht de ontvanger oneindig lang tot er een boodschap aankomt, en wacht de verzender oneindig lang of de boodschap ontvangen wordt?
- Wacht de verzender automatisch tot er een antwoord terug komt?
- Gaat het antwoord altijd terug naar hetzelfde proces?
- Kunnen boodschappen verouderen (ongeldig worden)?

Message passing:
Een manier van communicatie waarbij gestructureerde blokken informatie (**messages**) van een proces naar een ander gestuurd worden.

6.4.1 Synchrone en asynchrone communicatie

We zeggen dat communicatie tussen twee processen asynchroon is als de processen niet op elkaar hoeven te wachten om een boodschap over te kunnen dragen. Het zal duidelijk zijn dat een proces dat een boodschap wil ontvangen in ieder geval moet wachten tot de verzender de boodschap verstuurd heeft, anders is er namelijk niets te ontvangen.

Voor de verzender ligt dit anders: In principe is er geen noodzaak om te wachten tot de boodschap bij de ontvanger is aangekomen. Wanneer het O.S. de verzender toestaat om door te gaan na de SEND opdracht zeggen we dat er *asynchrone* communicatie is (er vindt geen synchronisatie tussen zender

en ontvanger plaats. Wanneer de afzender moet wachten tot de ontvanger de boodschap heeft spreken we van *synchrone* communicatie.

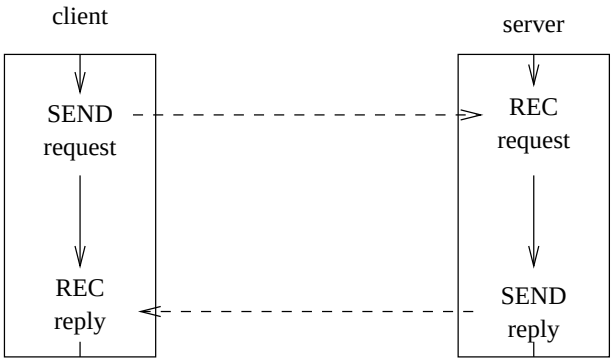
Synchrone communicatie:
Communicatie waarbij de zender moet wachten tot de ontvanger het bericht ontvangen heeft.
Asynchrone communicatie:
Communicatie waarbij de zender niet op de ontvanger hoeft te wachten.

Beide vormen van communicatie hebben voor- en nadelen. Bij synchrone communicatie kan de boodschap rechtstreeks van de zender naar de ontvanger gekopieerd worden (als we veronderstellen dat ze op dezelfde machine zitten). Dit maakt het ontwerp van de communicatiefaciliteit in het O.S. een stuk simpeler. Bij asynchrone communicatie mag de zender doorgaan en zou het geheugen waarin de boodschap was opgeslagen voor iets anders kunnen gebruiken (bijvoorbeeld voor een nieuwe boodschap). Daarom moet het O.S. eerst een kopie van deze boodschap maken en deze bewaren tot de ontvanger een RECEIVE opdracht geeft. Dit maakt het ontwerp een stuk moeilijker, want er komt een flink stuk bufferbeheer bij. Als de ontvanger eerst was met de RECEIVE opdracht is deze kopieeractie overbodig en kan het O.S. optimaliseren door rechtstreeks van de zender naar de ontvanger te kopieren. Maar zo'n speciaal geval maakt het weer ingewikkelder.

Aan de andere kant geeft asynchrone communicatie meer mogelijkheden om parallellisme uit te buiten en kaan daardoor tot een efficiënter systeem leiden.

6.5 Client/server

Er zijn gevallen waarin een asynchroon systeem niet veel oplevert omdat de communicatie in de vorm van een vraag- en antwoordspelletje gebeurt. Bijvoorbeeld proces A stuurt een verzoek naar proces B en B geeft antwoord op de vraag. De communicatie ziet er dan ongeveer zó uit (figuur 6.4):

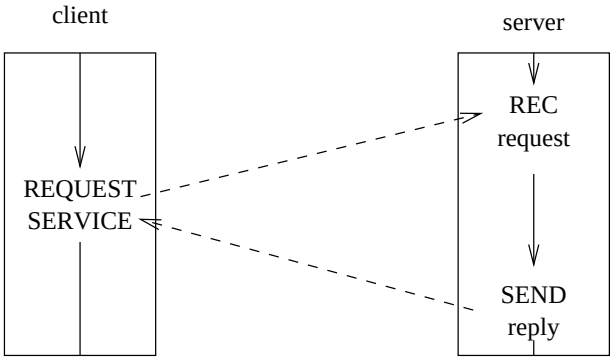


Figuur 6.4: Vraag en antwoord communicatie (client-server)

In veel gevallen zal het systeem zo opgezet zijn dat het proces dat de verzoeken beantwoord altijd op deze manier werkt en dat er verschillende ander processen zijn die dergelijke verzoeken doen. We spreken dan van een *client-server* systeem. De *server* is het proces dat de verzoeken krijgt en beantwoordt, de *cliënten* zijn de processen die de verzoeken doen. Een voorbeeld is een database

server die verzoeken om informatie of updates van cliënten krijgt en die in de database verwerkt. De cliënten is zo'n systeem zullen altijd met het paar (SEND-request, RECEIVE-reply) werken, i.h.a zonder tussen deze twee iets anders te doen. Omdat de cliënt in dit geval toch op het antwoord moet wachten, is het overbodig om de SEND-request asynchroon te laten doen, met alle overhead die het O.S. dan moet uitvoeren. Het is dan zelfs nog beter om niet twee afzonderlijke O.S. aanroepen te hebben, maar slechts één aanroep die het verzenden van het verzoek en het wachten op het antwoord beide na elkaar doet. Dit kan dan een synchrone vorm van communicatie zijn.

Client-server systeem:
Een systeem waarbij een **server**proces diensten levert die door **cliënt**processen door middel van verzoeken afgenomen worden. Cliënt- en serverprocessen kunnen op verschillende computers draaien.



Figuur 6.5: Vraag en antwoord communicatie (client-server)

We geven hier nog een overzicht van de belangrijkste eigenschappen van het *client-server* model:

- Servers in het systeem verrichten bepaalde *diensten (services)*. Voorbeelden: File server, database server, print server, mail server, news server, http server (WWW).
- Servers draaien meestal altijd, een enkel wordt echter naar behoefte opgestart. Servers draaien meestal op de wat zwaardere computers. Deze computers worden zelf ook wel servers genoemd, het woord server kan dus zowel de computer als het proces betekenen.
- Cliënten zijn programma's die van deze diensten gebruik maken. Heel vaak draaien cliënten op PC's of werkstations, en vaak worden deze door gebruikers gestart en hebben ze een grafische user interface. Bekende voorbeelden van cliënten zijn Netscape, email en newsreader programma's.
- Een server kan zelf ook weer cliënt van een andere server zijn. Een voorbeeld is een http (WWW) server die zelf weer cliënt is van een file server of database server om daar de HTML documenten vandaan te halen.

6.5.1 Emulatie van asynchrone communicatie op een synchroon systeem

Wanneer we werken op een systeem met synchrone communicatie (waarbij dus een SEND opdracht moet wachten tot de ontvanger een RECEIVE doet, kan het soms wenselijk zijn om toch het effect van

een asynchroon systeem te bereiken. Bijvoorbeeld om het voordeel van de extra parallelliteit te kunnen uitbuiten. In het algemeen zal het niet mogelijk zijn om de voorzieningen in het O.S. te veranderen. Zelfs wanneer dat wel mogelijk zou zijn dan nog kan het wel eens ongewenst zijn omdat we gezien hebben dat een asynchroon communicatiesysteem een veel ingewikkelder organisatie vereist i.v.m. buffering e.d.

In sommige situaties is het mogelijk om het effect van asynchrone communicatie toch te bereiken in user-mode, dus op het niveau van processen. Dit heeft dan het voordeel dat het O.S. niet aangepast hoeft te worden, en bovendien dat processen die tevreden zijn met synchrone communicatie geen last ondervinden van de extra overhead die gepaard gaat met asynchrone communicatie.

De twee manieren om het effect van asynchrone communicatie te bereiken in een synchroon systeem, zijn (1) met behulp van multithreading, en (2) met behulp van extra bufferprocessen.

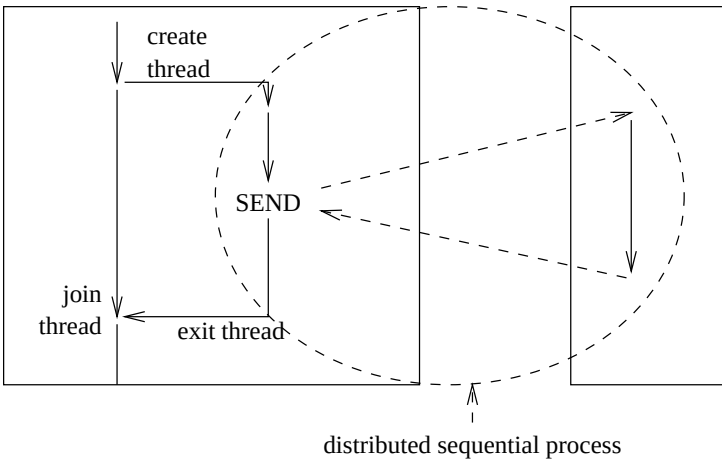
Om dit te bereiken met behulp van multithreading, moeten we kijken wat het effect van asynchrone communicatie is: Het proces dat een SEND opdracht doet kan doorgaan met zijn werk, terwijl de verzonden boodschap wacht op de ontvanger. In een multithreaded systeem kan dit door de communicatie in een aparte thread te doen. De originele thread kan doorgaan met het werk en de nieuwe thread doet een (synchrone) SEND en wacht dus automatisch totdat de ontvanger de boodschap overgenomen heeft. Wanneer de originele thread wil wachten tot de boodschap ontvangen is, doet hij dat door een "join" operatie met de nieuwe thread te doen. De "communicatie-thread" vormt samen met de operatie in de ontvanger een sequentieel stuk werk, dat echter gedistribueerd is over twee (of evt meer) processen. We spreken dan weleens van een *distributed sequential process*.

Distributed sequential proces:
Een berekening die sequentieel uitgevoerd wordt (dat wil zeggen er worden geen delen ervan gelijktijdig gedaan) maar wel verdeeld over verschillende processen en/of computers.

Merk op dat het probleem van het kopiëren van de boodschap wat in een asynchroon communicatie systeem door het systeem zelf gedaan wordt, hier niet automatisch plaatsvindt. Het is mogelijk om dit in de apart opgezette thread te laten doen. Ook kan het in de originele thread gebeuren. Het zal duidelijk zijn dat dit alleen hoeft te gebeuren als het echt nodig is. Omdat beide threads in het algemeen door dezelfde persoon (of groep personen) ontworpen worden, kan dit kopieren naar behoefte gedaan worden, wat moeilijker is wanneer het in het O.S. gebeurt.

Wanneer multithreading niet in het O.S. aanwezig is, kan i.p.v. een aparte thread een apart proces opgestart worden. Dit kost echter meer tijd. In Unix kan hiervoor gebruik gemaakt worden van de `fork` systeemaanroep, die als extra voordeel heeft dat automatisch een kopie van de boodschap gemaakt wordt. Op andere systemen, bijvoorbeeld MS Windows is dit niet zo en zou de boodschap in zo'n geval bijvoorbeeld via shared memory doorgegeven moeten worden. De Unix oplossing met `fork` heeft het nadeel dat het moeilijker is om een antwoord dat in het nieuwe proces terugkomt, door te geven naar het oorspronkelijke proces. Hiervoor zou evt. ook shared memory gebruikt kunnen worden.

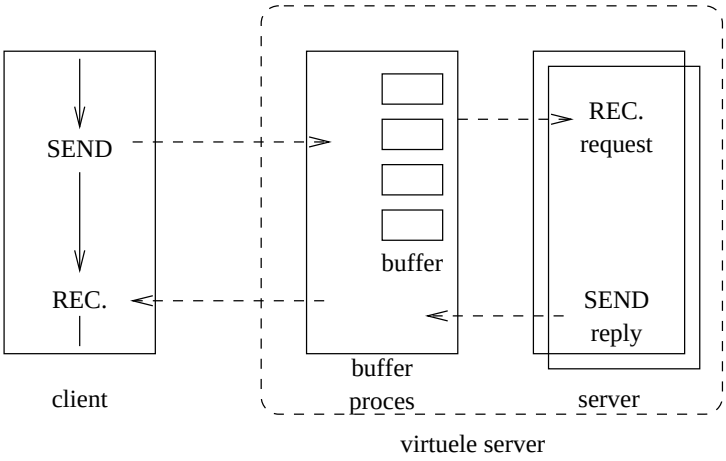
Een tweede mogelijkheid om op een synchroon communicatiesysteem een asynchrone vorm van communicatie te verkrijgen is het gebruik van een apart buffer proces. In dit geval wordt er aan de kant van de ontvanger een extra proces opgestart dat in eerste instantie de boodschappen ontvangt. Als we het voorbeeld van een client-server systeem nemen, dan krijgen we dus bij de server een extra proces, waarmee de cliënt communiceert. Dit proces doet in eerste instantie niets anders dan de boodschap aannemen en in een buffer opbergen. Het buffer proces moet nu echter de boodschappen doorsturen naar het echte server proces. Dit kan natuurlijk weer met SEND, maar dan is er het probleem dat de



Figuur 6.6: Asynchrone communicatie via threads

server nog niet klaar is om te ontvangen, waardoor het bufferproces moet wachten. Het kan dan geen nieuwe RECEIVE doen, waardoor andere cliënten toch weer in een synchrone communicatie verzeild raken. Dit lost het probleem dus niet op, en geeft hoogstens één boodschap extra asynchroniteit. Er zijn twee mogelijkheden om dit op te lossen: De eerste is dat er een polling mogelijkheid moet zijn, bijvoorbeeld door een timeout op de SEND of RECEIVE op te geven. Het buffer proces kan dan opeenvolgend SENDs met timeout naar de server doen en RECEIVEs met timeout naar cliënten. Telkens als er een echt iets te doen heeft, wordt de betreffende operatie uitgevoerd. Een andere methode is het gebruik van shared memory. Als de server en het buffer proces een stuk gemeenschappelijk geheugen hebben kunnen ze de boodschappen hierin uitwisselen, waardoor er geen wachten nodig is.

De antwoorden van de server kunnen in principe ook via hetzelfde bufferproces terug, omdat de server dan niet hoeft te wachten tot de cliënt een RECEIVE voor het antwoord doet. Als dat zou gebeuren kan de server namelijk geen nieuwe verzoeken afwerken, en zou een slecht geprogrammeerde of op hol geslagen cliënt de hele server plat kunnen leggen.



Figuur 6.7: Asynchrone communicatie via een buffer proces

Het is ook mogelijk om één buffer proces te hebben en meer dan één server proces. Hierdoor kan een extra prestatie verkregen worden, bijvoorbeeld op een multiprocessor systeem, terwijl de cliënten zich niet bewust hoeven te zijn van dit feit en slechts één proces hebben om mee te communiceren.

Tenslotte is het ook nog mogelijk om in een multithreading systeem het bufferen niet in een apart proces te doen maar in een aparte thread. Het probleem van het doorgeven van de boodschappen via shared memory wordt dan gemakkelijker omdat de threads al in een gemeenschappelijke adresruimte lopen.

6.6 Shared memory

Het gebruik van shared memory is een snelle manier van IPC. Twee of meer processen kunnen een stuk shared memory reserveren en volgens een of andere afspraak kunnen hier boodschappen ingezet worden. Het is mogelijk om de vorm van message passing te gebruiken door voor elk ontvangend proces een queue in het shared memory op te nemen. de toegang tot de queue moet dan beschermd worden om te voorkomen dat twee of meer processen tegelijk de queue veranderen omdat anders de administratie in de war kan raken. Dit kan bijvoorbeeld doordat het O.S semaforen levert waarmee dit gedaan kan worden. Op een uniprocessor systeem is het ook wel mogelijk om dit buiten het O.S. om te doen, maar bij een multiprocessor systeem kan dit alleen efficiënt als er speciale instructies in de CPU zijn. Het gaat te ver om hierover in meer details te treden.

Hoewel shared memory dus een snelle manier is om boodschappen tussen processen uit te wisselen, blijft er wel een probleem: hoe weet een proces dat er een boodschap gearriveerd is? Het is nogal verspillen voor een proces om continu te blijven kijken of er iets binnenkomt. Een van de mogelijkheden is om een signal te geven als je iets voor een ander proces klaar zet.

Het gebruik van shared memory geeft echter ook mogelijkheden die verder gaan dan message-passing: bijvoorbeeld een ingewikkelde beeldverwerking kan opgesplitst worden in een aantal processen, waarbij het te bewerken beeld in shared memory opgeslagen is. De processen die het beeld bewerken zullen vaak kris-kras door het geheugen aan het werk zijn, iets wat met de andere behandelde IPC manieren bijna niet doenlijk is.

6.7 Opgaven

1. Op MS-DOS en Win16 bestaan geen pipes. Beschrijf hoe het O.S. het effect van een pipe kan emuleren met behulp van een file. Geef ook aan wat het verschil is met een echte pipe.

2. Geef met een tekening aan hoe de verschillende filedescriptors in de code van sectie 6.3 met elkaar gerelateerd zijn.

3. Geef van alle in dit hoofdstuk genoemde IPC technieken aan of ze in een gedistribueerd systeem gebruikt kunnen worden (dus tussen processen die in verschillende computers draaien). Beargumenteer uw antwoord.

4. Geef met behulp van (pseudo-)code aan hoe emulatie van asynchrone communicatie zoals in figuur 6.6 gedaan kan worden.

Hoofdstuk 7

Synchronisatie

Omdat in moderne O.S's veel activiteiten parallel (concurrent) plaatsvinden, is er in allerlei situaties synchronisatie van activiteiten nodig. (Voor de definitie van synchronisatie zie pagina 4.) In de meeste gevallen gaat het dan om het wachten tot een bepaalde activiteit klaar is, of het regelen van de toegang tot een gemeenschappelijke bron. Voorbeelden van beide zijn:

Wachten tot een activiteit klaar is:

- Bij in- en uitvoer: wachten tot de operatie voltooid is
- Bij threads: een thread wacht tot een andere thread klaar is
- Bij processen: een proces wacht tot een ander proces klaar is. Meestal wacht hierbij een ouder-proces tot een kind-proces klaar is, zie ook deel 1, sectie 5.1.
- In een netwerk: wacht tot een boodschap verstuurd is
- Idem: wacht tot het antwoord komt

Toegang tot een gemeenschappelijke bron:

Wanneer twee of meer threads (of processen) dezelfde gegevens (bijvoorbeeld een verzameling variabelen) moeten kunnen bewerken dan moet gezorgd worden dat ze dit niet tegelijkertijd doen, anders kan er chaos ontstaan. Zo'n gegeven of verzameling gegevens noemen we een *shared resource*. Een stuk programma waarin zo'n shared resource bewerkt wordt heet een *kritieke sectie*.

Shared resource:
Een gegeven of verzameling gegevens dat tegelijk vanuit verschillende processen of threads bewerkt zou kunnen worden.

Kritieke sectie:
Een stuk programma waarin een shared resource bewerkt wordt.

- Een proces of thread moet wachten tot een andere uit de kritieke sectie komt, omdat er een update op een gemeenschappelijke datastructuur plaatsvindt.
- wachten op resources, bijvoorbeeld de printer, een modem, of een stuk geheugen
- in het filesysteem: gemeenschappelijke operaties op directories
- diverse kritieke secties in de kernel voor het bijwerken van kernel datastructuren.

Verschillende van bovengenoemde wachtsituaties zijn "automatisch" of impliciet, omdat je als programmeur er niets speciaals voor hoeft te doen. Zo zijn bijvoorbeeld bij het gebruik van synchrone I/O de wachtoperaties tot het voltooien van de I/O impliciet. Ook de kritieke secties in de kernel worden al geregeld, al heeft natuurlijk de kernel-programmeur hiervoor het nodige moeten doen.

Andere situaties kunnen echter alleen door de programmeur zelf aangegeven worden, zoals wanneer gewacht moet worden tot een kind-proces klaar is, of wanneer een kritieke sectie begonnen, resp. beëindigd moet worden.

We zullen in dit hoofdstuk bekijken welke synchronisatievoorzieningen aanwezig zijn in Unix en Win32. De meeste van deze voorzieningen zijn gebaseerd op het principe van de semaforen of de monitors[1].

De operaties die het systeem ons levert noemen we *synchronisatieprimitieven*. Om te beginnen zullen we beknopt uitleggen wat de verschillende synchronisatieprimitieven doen.

Een *semafoor* is een teller die aangeeft hoeveel processen/threads een kritieke sectie mogen binnengaan zonder te wachten. Een semafoor heeft twee operaties (methodes) P en V. De P operatie wordt uitgevoerd als de thread de kritieke sectie wil binnengaan. Dit wordt alleen toegestaan als de teller van de semafoor groter dan 0 is, anders moet de thread wachten. Als de thread klaar is met de kritieke sectie dan doet hij de V operatie op de semafoor en als er threads stonden te wachten dan mag er een doorgaan. Heel vaak zal het voorkomen dat er maximaal één thread in de kritieke sectie mag zijn, dan kan de semafoorteller alleen de waarde 0 en 1 aannemen. We spreken dan van een *binaire semafoor*.

Semafoor:
Een mechanisme om gelijktijdige toegang tot een kritieke sectie te beperken tot een maximaal aantaal processen of threads.

Een andere manier van synchroniseren die voorkomt is de *monitor*. Dit is een stuk programma (bijvoorbeeld een klasse met methoden) waar maximaal één thread tegelijkertijd in mag bezigzijn. Dit wordt o.a. in Java gebruikt. Een Java object waarin methoden gedefiniëerd zijn met het keyword `synchronized` is automatisch een monitor. Er geldt dan dat in de synchronized methoden van een bepaald object maximaal één thread actief mag zijn. Je zou dus kunnen zeggen dat deze methodes een kritieke sectie vormen. Methodes die niet `synchronized` zijn tellen niet mee.

Monitor:
Een object waarbij niet meer dan één proces of thread tegelijk een methode mag uitvoeren.

De vraag rijst natuurlijk of deze operaties in het O.S. (de kernel) geïmplementeerd moeten zijn of in de gebruikerscode (bijvoorbeeld in een bibliotheek). Het antwoord hangt af van de plaats van de scheduler. Bij een synchronisatie operatie kan het gebeuren dat een proces of thread geblokkeerd raakt,

[1] Dit wordt behandeld in het college Gedistribueerd Programmeren

bijvoorbeeld omdat een kritieke sectie al ergens anders in gebruik is. In dat geval zal er tijdelijk een ander proces of een andere thread uitgevoerd moeten worden. Omgekeerd: als de kritieke sectie of een andere bron vrijkomt, kan de geblokkeerde thread of proces weer gescheduled worden. De beste plaats voor de synchronisatieprimitieven is dus dezelfde plaats waar de scheduler zit, zodat de synchronisatieoperaties de informatie over al dan niet blokkeren kunnen doorgeven aan de scheduler. Wanneer we het hebben over synchronisatie van processen dan betekent dit dus de kernel. Wanneer we het hebben over synchronisatie van threads binnen een proces dan is het afhankelijk van waar de thread scheduling gedaan wordt. Als het O.S. threads kent, dan horen de thread synchronisatieprimitieven ook in het O.S. thuis. Wanneer threads in user mode geïmplementeerd zijn dan moeten de bijbehorende synchronisatieprimitieven dat ook zijn. Vandaar dat thread-bibliotheken vaak ook synchronisatie-operaties bevatten.

Bij het gebruik van semaforen om een kritieke sectie te beschermen is er altijd de kans dat een proces of thread een kritieke sectie ingaat, en deze daardoor blokkeert voor andere processen of threads, en dan door een programmafout “crasht” voordat de kritieke sectie weer vrijgegeven wordt. Hierdoor hangt de rest van het systeem dat van de door de kritieke sectie beschermde shared resource zou willen gebruik maken. Zolang afgesproken is dat het proces of de thread die de shared resource blokkeert hem ook weer moet vrijgeven kan het O.S[2] detecteren dat er een geblokkeerde resource is, en die alsnog vrijgeven als het proces of de thread stopt zonder de vrijgave te doen. Als er een techniek gebruikt wordt waarbij het vrijgeven door een ander proces of een andere thread kan gebeuren dan het reserveren dan is dit niet meer mogelijk. Bijvoorbeeld bij de oplossing van het bounded buffer probleem met semaforen, waarbij verschillende processen op dezelfde semafoor opereren kan dit niet. Daarom wordt er vaak onderscheid gemaakt tussen algemene semaforen en binaire semaforen (mutexen). Een mutex heeft een *lock* en een *unlock* operatie, en is voornamelijk bedoeld om een kritieke sectie te implementeren, of een shared resource te beschermen waar slechts één proces of thread tegelijk bijmag. De *unlock* operatie moet gedaan worden door hetzelfde proces of dezelfde thread die de *lock* doet. Als deze voortijdig sterft kan het O.S. dan alsnog de unlock doen. Let op dat het dan wel duidelijk moet zijn of deze operaties bij threads horen of bij processen. We zullen in de rest van dit hoofdstuk hier een aantal voorbeelden van zien.

Mutex:
Een mechanisme om exclusieve toegang tot een shared resource te krijgen.

7.1 Synchronisatie met Pthreads

Pthreads (Posix threads) is een bibliotheek voor het gebruik van threads op Unix systemen. Posix is een standaard die eenheid probeert te brengen tussen de diverse Unix systemen. In feite gaat Posix nog iets verder: het is een voorstel voor een standaard O.S. interface (API), en ook niet-Unix systemen kunnen deze interface ondersteunen (Windows NT en DEC’s VMS hebben Posix mogelijkheden).

De Pthreads standaard specificeert aanroepen voor o.a. het creëren van threads, het wachten op threads, en synchronisatieprimitieven gebaseerd op binaire semaforen (mutexen) en conditievariabelen (zie 7.1.2. In de bibliotheek zijn geen operaties voor algemene semaforen opgenomen, maar deze kunnen geïmplementeerd worden met de aanwezige primitieven. In deel 1, het einde van sectie 5.5 wordt

[2]Tenminste als het O.S. de synchronisatie doet.

een lijst van gewenste thread operaties gegeven. In feite zijn dit de operaties zoals die in Pthreads aanwezig zijn, waarbij de echte namen in de Pthreads bibliotheek met `pthread_` beginnen. De Pthreads standaard zegt niets over de plaats waar deze operaties geïmplementeerd zijn. Naar keuze kan dit in het O.S. of in user-mode zijn.

7.1.1 Pthreads Mutexen

Een *mutex* is (min of meer) een binaire semafoor, met een *lock* en een *unlock* operatie. Zoals boven beschreven is een vereiste dat de unlock door dezelfde thread gebeurt als de lock operatie.

Een *mutex* wordt gedeclareerd met een speciaal type (`pthread_mutex_t`); deze declaratie moet natuurlijk globaal zijn t.o.v. de threads die er gebruik van maken. Voordat de mutex gebruikt kan worden moet hij geïnitialiseerd worden met de aanroep `pthread_mutex_init(mutex,...)`. Hiermee worden de velden in de variabele geïnitialiseerd en wordt de mutex in de administratie opgenomen. Er is ook een omgekeerde operatie `pthread_mutex_destroy(mutex)`, waarmee de mutex uit de administratie verwijderd wordt.

Het gebruik van de mutex voor het beschermen van een kritieke sectie is als volgt:

```
pthread_mutex_lock(mutex)
/* kritieke sectie */
pthread_mutex_unlock(mutex)
```

Wanneer een andere thread nu een `pthread_mutex_lock(mutex)` doet op een reeds gelockte mutex dan blokkeert deze thread en wordt – indien mogelijk – een andere thread gescheduled. Wanneer de mutex wordt geunlockt kan een van de wachtende threads weer gescheduled worden.

Er is ook nog een operatie `pthread_mutex_trylock(mutex)` die de mutex lockt indien dat mogelijk is maar niet blokkeert als de mutex al gelockt is. In het laatste geval wordt er een specifieke foutcode teruggegeven en kan de thread besluiten om tijdelijk iets anders te doen. Zowel `pthread_mutex_lock` als `pthread_mutex_trylock` controleren op deadlocks en geven een foutcode terug als er een deadlock zou ontstaan.

Een verschil tussen mutexen en binaire semaforen zoals soms beschreven in de literatuur is dat een binaire semafoor daar ook wacht als een unlock (V) operatie gedaan wordt op een niet gelockte semafoor. Maar omdat in pthreads alleen de thread die een lock gedaan heeft ook de unlock mag doen is dat hier geen probleem.

7.1.2 Pthreads conditie variabelen

Problemen waarbij een simpele lock/unlock niet voldoende is voor een oplossing zorgen ervoor dat we ook andere primitieven nodig hebben. Er zijn twee redenen waarom de mutexen niet voldoen voor ingewikkelder problemen:

1. Een mutex moet geunlockt worden door dezelfde thread die hem gelockt heeft. Het is dus onmogelijk om vanuit één thread een andere door te starten

2. Een mutex heeft maar twee toestanden, en er kan dus maar één thread tegelijk is een kritieke sectie zijn. Er zijn echter situaties waarbij meer threads onder bepaalde voorwaarden in hun kritieke sectie mogen zijn. Voorbeelden hiervan zijn het *bounded buffer probleem* en het *reader en writers* probleem. In beide gevallen kunnen we dit oplossen door extra variabelen te gebruiken, bijvoorbeeld bij het bounded buffer probleem de hoeveelheid ruimte die nog vrij is in de buffer, en bij het readers en writers probleem de aantallen readers en writers. Omdat deze variabelen een "shared resource" zijn moeten wijzigingen hierop beschermd worden met een mutex. Ook wanneer we een conditie willen testen waarin meer dan één variabele betrokken is moet dit in een kritieke sectie gebeuren d.m.v. een mutex. Maar wanneer een thread moet wachten, bijvoorbeeld wanneer de buffer vol is of wanneer er een writer actief is, dan moet de wachttoestand opgeheven worden door een andere thread en dit kan niet met een mutex gebeuren. Hiervoor heeft pthreads *conditievariabelen*.

Een conditievariabele is een ding waarop een thread kan wachten en waardoor ook aangegeven kan worden dat een wachtende thread verder kan gaan. Een verschil tussen pthreads conditievariabelen en monitors is dat in een monitor gegarandeerd is dat altijd hoogstens één operatie uit de monitor actief kan zijn. De monitor code is dus altijd een kritieke sectie. In pthreads is dat niet zo, maar waar dit nodig is kan met een mutex een kritieke sectie gemaakt worden. In feite is het zo dat een conditievariabele bedoeld is om het mogelijk te maken te wachten op algemene condities (de **await** $B \rightarrow S_1; \ldots$ operatie) waarbij B een willekeurige expressie kan zijn. We hebben dan in ieder geval een mutex nodig om de operaties op de variabelen in B te beschermen. Daarom hebben de conditievariabele synchronisatieprimitieven ook een mutex als parameter.

Net als bij een mutex moet een conditievariabele eerst gedeclareerd (type `pthread_cond_t`) en geinitialiseerd worden: `pthread_cond_init(cv,...)`.
Omgekeerd is er ook een `pthread_cond_destroy(cv)` die de conditievariabele verwijdert uit de pthreads administratie. De synchronisatie primitieven zijn:

- `pthread_cond_wait(cv, mutex)` wacht
- `pthread_cond_timedwait(cv, mutex, tijd)` wacht een maximale tijd
- `pthread_cond_signal(cv)` signaleert dat de conditie vervuld is
- `pthread_cond_broadcast(cv)` signaleert aan alle wachtende threads.

De `pthread_cond_wait(cv, mutex)` aanroep mag alleen gegeven worden als de aanroepende thread de mutex gelockt heeft. Dit komt conceptueel overeen met het aanwezig zijn in de monitor. De aanroep zal gegeven worden als de thread gedetecteerd heeft dat aan de conditie B niet voldaan is, en de mutex werd gebruikt om de variabele(n) in deze conditie te beschermen tegen gemeenschappelijke toegang. De aanroep blokkeert de thread maar unlockt eerst de meegegeven mutex. Dit is noodzakelijk omdat tijdens de blokkade een andere thread de gelegenheid moet krijgen om de conditie waar te maken en dus moeten de bijbehorende variabelen geunlockt zijn. De thread wordt pas weer gedeblokkeerd als een andere thread een signal geeft. Wanneer de thread gedeblokkeerd wordt wordt de mutex weer gelockt zodat we conceptueel weer in de monitor zitten. Mocht intussen een andere thread de mutex gelockt hebben dan wacht de thread tot deze geunlockt is en lockt hem dan zelf.

Een alternatieve aanroep is `pthread_cond_timedwait(cv, mutex, tijd)` waarbij dezelfde regels gelden, maar waarbij de thread slechts een maximale tijd geblokkeerd blijft. Wanneer de tijd

verstreken is voordat er een signal gegeven is dan wordt de thread in ieder geval gedeblokkeerd (mits de mutex geunlockt is).

De aanroep `pthread_cond_signal(cv)` signaleert dat de conditie mogelijkerwijs vervuld is. Het heeft tot gevolg dat één thread die op de conditievariabele wacht gedeblokkeerd wordt. Als er meer threads wachten bepaalt de scheduler welke dat is. Vaak wordt deze aanroep gegeven als de aanroepende thread weet dat de wachtconditie vervuld is, maar noodzakelijk is dit niet. Het is een kwestie van afspraak tussen degene(n) die de threads geschreven hebben (heeft). Maar zelfs als de signal alleen gegeven wordt als de conditie vervuld is, kan door de wachtende threads niet zomaar aangenomen worden dat na deblokkering de conditie waar is. Immers de scheduler zou best een andere thread kunnen laten voorgaan die de conditie weer onwaar maakt. Daarom zien we bij het wachten meestal een constructie van de vorm:

```
pthread_mutex_lock(mtx)
while (! B) pthread_cond_wait(cv, mtx)
S1; ...
pthread_mutex_unlock(mtx)
```

als implementatie van **await** $B \rightarrow S_1; \ldots$

De aanroep `pthread_cond_broadcast(cv)` deblokkeert alle wachtende threads. Vanwege de mutex kan er telkens maar één tegelijk verder, maar op den duur komen ze allemaal aan de beurt. Of wanneer de waits verschillende mutexen gebruiken, wat in bijzondere omstandigheden denkbaar is, dan kunnen er meerdere direct verder.

Een probleem waarop gelet moet worden is dat een signal op een conditievariabele waarop niet gewacht wordt verloren gaat. Er wordt niet onthouden dat de signal al gebeurd is. Als we echter zorgen dat zowel het waar maken van de conditie B als het testen op de conditie voor een wait beschermd is door een mutex dan kan dit niet tot problemen leiden. Om deze reden zullen we zelfs een mutex nodig hebben als de conditie alleen maar de waarde van één variabele is.

Tenslotte moet er nog op gewezen worden dat in Pthreads alleen threads binnen hetzelfde proces gesynchroniseerd kunnen worden. Voor synchronisatie tussen verschillende processen of tussen threads in verschillende processen hebben sommige Unix systemen aparte semaforen.

7.1.3 Implementatie van semaforen in Pthreads

Met behulp van een conditievariabele, een mutex en een teller kunnen we gemakkelijk algemene semaforen implementeren. Dit komt in feite overeen met de implementatie van semaforen met behulp van monitors, waarbij het monitor aspect door de mutex geïmplementeerd wordt. We geven de implementatie in pseudo-code, waarbij een semafoor een object is met een conditievariabele (`cv`), een `mutex` en een `count`. De wachtconditie B is `count` > 0.

P(sem):

```
lock (sem.mutex)
while (sem.count <= 0)
      wait (sem.cv, sem.mutex)
sem.count = sem.count - 1
unlock (sem.mutex)
```

V(sem):

```
lock (sem.mutex)
sem.count = sem.count + 1
unlock (sem.mutex)
signal (sem.cv)
```

7.2 Synchronisatie in Win32

Win32 heeft (o.a.) semaforen en mutexen als kernel objecten. Deze kunnen gebruikt worden zowel om threads als om processen te synchroniseren. In tegenstelling tot Pthreads is het hierbij ook mogelijk om te synchroniseren tussen threads in verschillende processen. Omdat het in dat geval onmogelijk is (of alleen met gecompliceerd gebruik van shared memory) om semafoorvariabelen in de adresruimte van het proces te hebben (want dan kunnen verschillende processen niet dezelfde semafoor hebben) is er een naamruimte voor semaforen. Een semafoor kan dus een naam hebben net als een file, en die naam is in het hele systeem bekend.

De Win32 systeemaanroep `CreateSemaphore` maakt een nieuwe semafoor aan. De semafoor heeft o.a. een beginwaarde, en een naam die als parameters meegegeven worden.

De aanroep `OpenSemaphore` gebruikt een bestaande semafoor (de naam moet meegegeven worden). Er zijn twee operaties die overeenkomen met de P() en V() operatie:

- `WaitForSingleObject(sema)` is de P operatie. De waarde van de semafoor wordt afgelaagd als deze > 0 is, anders wordt er gewacht.
- `ReleaseSemaphore(sema,...)` is de V operatie. Er kan ook meegegeven worden dat de waarde met meer dan 1 opgehoogd wordt.

Behalve semaforen heeft Win32 ook Mutexen, dus binaire semaforen vergelijkbaar met die van Pthreads. Ze hebben de volgende eigenschappen:

- ze hebben slechts twee toestanden (gelockt en geunlockt)
- ze horen bij een thread i.p.v. proces: als de thread termineert terwijl de mutex gelockt is wordt de lock opgeheven. Semaforen zijn eigendom van een proces, en ze kunnen niet in een default toestand gezet worden als het proces crasht.
- ze kunnen ook gebruikt worden om threads in verschillende processen te synchroniseren.

De operaties op de mutexen zijn `WaitForSingleObject(mutex)` en `ReleaseMutex(mutex)`.

De operatie `WaitForSingleObject()` is een algemene systeemaanroep die ook voor andere wachtobjecten gebruikt wordt. Bijvoorbeeld voor file handles (asynchrone I/O), proces handles (wacht tot proces klaar is), timers e.d. Er is ook een overeenkomstige aanroep `WaitForMultipleObjects()` die gebruikt kan worden om op meer objecten te wachten. Er is dan een extra parameter die aangeeft of er gewacht wordt tot één van de opgegeven objecten klaar is, of tot ze allemaal klaar zijn.

Zowel `WaitForSingleObject()` als `WaitForMultipleObjects()` doen een P() operatie als het te wachten object een semafoor of een mutex is. Wanneer bij `WaitForMultipleObjects()` wordt gespecificeerd dat op alle objecten gewacht moet worden dan worden de P() operaties pas uitgevoerd als ze op alle opgegeven objecten tegelijkertijd uitgevoerd kunnen worden. Anders zou gemakkelijk een deadlock kunnen ontstaan. Deze operaties hebben ook de mogelijkheid om een maximale wachttijd als parameter op te geven.

7.3 File locking

In sectie 4.7 op pagina 79 is reeds over file locking gesproken. We zullen er hier iets dieper op in gaan vanwege het feit dat ook dit een synchronisatiemechanisme is.

Processen (of threads) die gelijktijdig op dezelfde file willen werken zullen zich moeten beschermen wanneer er veranderingen op de file plaatsvinden. Dit kan natuurlijk gebeuren door middel van de eerder in dit hoofdstuk behandelde primitieven, omdat een file een shared resource is. Alle oplossingen die bekend zijn voor het readers en writers probleem kunnen hier toegepast worden. Echter als er wijzigingen op een file optreden is het nogal grof om de file in zijn geheel als shared resource te beschouwen. Immers als de operaties op verschillende disjuncte delen van de file plaatsvinden is er geen conflict. Daarom hebben de moderne O.S's file locking operaties waarbij niet de file in zijn geheel, maar een gespecificeerd deel van de file aangegeven wordt. De principes van file locking zijn op Unix en Win32 te vergelijken maar verschillen in de details van wanneer de detectie van conflicten tussen readers en writers plaatsvinden.

Om te beginnen zullen we resumeren wat het readers en writers probleem is met een simpel voorbeeld. Om een wijziging in een file aan te brengen zal een proces een deel van de file inlezen in het geheugen[3] de wijzigingen aanbrengen en dat deel weer terugschrijven. Wanneer gelijktijdig een ander proces ook een wijziging op hetzelfde deel wil aanbrengen gaat hoogstwaarschijnlijk tenminste één van de wijzigingen verloren. Er moet dus voor gewaakt worden dat ten hoogste één "writer" op de file bezig kan zijn. Een voorbeeld is het invoegen van een record in een B-boom: het kan gebeuren dat één of meer knopen in de boom gewijzigd en/of gesplitst moeten worden. Niet alleen moet elke knoop beschermd worden tegen gelijktijdige wijzigingen door andere processen, maar ook moet de hele verzameling operaties niet onderbroken worden door wijzigingen van andere processen in dezelfde verzameling anders kan de informatie inconsistent worden. Operaties op disjuncte stukken zijn geen probleem.

Bij processen die alleen maar lezen van de file zijn er geen onderlinge conflicten, maar een proces dat leest kan wel een conflict opleveren met een proces dat schrijft. Neem als voorbeeld een "reader" proces dat door de boom heenloopt, terwijl tegelijkertijd een ander proces een wijziging aanbrengt, bijvoorbeeld een knoop splitst. Het kan zijn dat de reader een knoop leest, daarna de writer deze knoop verandert en ook de knopen die in de boom de kinderen ervan zijn, en dat de reader pas daarna de kinderen leest. De reader heeft dan een ouderknoop en kindknopen die niet meer met elkaar in overeenstemming zijn. Het omgekeerde is natuurlijk ook mogelijk. Daarom moeten ook readers en writers tegen elkaar beschermd worden.

Als we het hebben over synchronisatie tussen verschillende processen dan moeten deze operaties in de kernel of in het filesysteem deel van het O.S. plaatsvinden. In sommige database systemen wordt

[3] Bij het gebruik van alleen memory-mapped file operaties is de inhoud van de file als shared memory aanwezig en kunnen andere technieken zoals semaforen gebruikt worden.

al het werk op de database gedaan door een apart database proces, en dan kan dit allemaal in dat betreffende proces gedaan worden.

7.3.1 File locking in Win32

Bij het openen van de file in Win32 moeten we o.a. specificeren:

- wat we met de file willen doen: READ en/of WRITE
- wat we andere processen toestaan: niets (exclusief), READ, WRITE of beide

De meeste "gewone" programma's zoals editors openen een file exclusief, en dan is er slechts één proces dat de file kan benaderen. In dat geval is het zinloos om nog delen van de file te beschermen. Bij de andere "share" mogelijkheden kunnen we delen van de file locken:

- `LockFile(filehandle, beginbyte, eindbyte)` lockt het deel tussen `beginbyte` en `eindbyte`. (De echte aanroep ziet er iets ingewikkelder uit omdat de `beginbyte` en `eindbyte` parameters als twee 32-bits woorden worden gegeven.)
- `UnlockFile(filehandle, beginbyte, eindbyte)` unlockt het deel tussen `beginbyte` en `eindbyte`
- Als een ander proces een read resp. write wil doen op een gelockt stuk krijgt het een foutcode terug; het deel dat buiten locks viel is dan wel gelezen of geschreven. Het is in Windows 95 niet mogelijk om aan te geven dat de lock een read-lock of een write-lock is, in Windows NT is er een uitgebreidere functie waarmee dit wel kan.

7.3.2 File locking in Unix

In Unix wordt file sharing niet opgegeven bij openen van een file. Wel wordt aangegeven of men wil lezen of schrijven. Pas bij het locken van een del van een file wordt aangegeven of anderen dit deel mogen lezen of schrijven. Op deze manier is wat meer flexibiliteit mogelijk omdat een proces dan op een deel van de file als reader en op een ander deel als writer kan optreden.

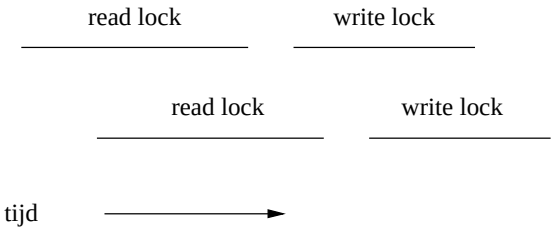
We geven hier in het kort een overzicht van de functionaliteit. Let op dat bij Unix een lock conflict optreedt bij het locken, en niet zoals op Win32 bij het lezen resp. schrijven.

- De `fcntl()` system call wordt gebruikt om een deel van de file te locken
- Hierbij kan opgegeven worden of het een read lock of write lock betreft.
- `fcntl(fid, F_SETLK, param)` lockt of unlockt een deel van de file open op `fid` zoals beschreven door `param`. In `param` staat het soort lock en de begin- en eindposities van het te locken deel.
- Een *read lock* laat geen andere write locks toe op hetzelfde stuk, maar wel read locks
- Een *write lock* laat geen andere read locks of write locks toe op hetzelfde stuk

- Een lock conflict treedt op als er een overlap is van de betrokken stukken.
- De operatie `F_SETLK` geeft een error-code als er een lock conflict is
- De operatie `F_SETLKW` blokkeert in zo'n geval
- De operatie `F_GETLK` geeft informatie over locks op een file (welk stuk gelockt is en door welk proces)
- afhankelijk van de file protecties is locking *advisory* (`read` en `write` trekken zich er niets van aan) of *enforced* (ze doen dat wel).

7.4 Opgaven

1. Kan het probleem in sectie 5.7 worden opgelost met een mutex? Zo ja, leg uit hoe. Zo nee, leg uit waarom niet.
2. Waarom zijn er in Win32 zowel mutexen als semaforen? Is een mutex niet gewoon een semafoor met minder mogelijkheden?
3. Bedenk zelf een voorbeeld waarbij file locking nodig is. Leg ook uit wat de juiste grootte is voor de te locken delen van de file.
4. De processen A en B locken telkens hetzelfde gedeelte van een file. De tijdsvolgorde is als volgt:

   Geef aan wanneer er lock conflicten optreden en wanneer deze opgelost zijn.
5. Vertaal bovenstaande voorbeeld naar de file locking opdrachten van Win32.

Hoofdstuk 8

Inter Proces Communicatie in Netwerken

8.1 Adressering

In het hoofdstuk over IPC hebben we steeds gesproken over “communicatie tussen twee processen”. De gemakkelijkste manier om dit voor te stellen is dat bij het verzenden van een boodschap het zendende proces de “naam” of een andere identificatie van het ontvangende proces opgeeft. Dit is echter niet altijd de beste methode om dit te doen. Hoe komt het verzendende proces bijvoorbeeld te weten op welke machine het bestemmingsproces staat, en welk procesnummer dit proces heeft? Het laatste is ook een probleem zelfs als we op één machine blijven.

8.1.1 Ports

Daarom gebruiken veel moderne systemen niet meer de processen als bestemming van een boodschap, maar z.g. *ports*. Een port is te vergelijken met een brievenbus aan een huis. De boodschappen worden geadresseerd aan de port en het ontvangende proces haalt ze eruit.

In de meeste systemen is het dan ook mogelijk om per ontvangend proces meer dan één port te hebben. Het voordeel hiervan is dat verschillende soorten boodschappen naar verschillende ports gestuurd kunnen worden. Dat kan bijvoorbeeld om voor verschillende operaties verschillende ports te hebben of bijvoorbeeld een aparte port voor urgente boodschappen. Sommige systemen gebruiken ook bij servers een port per object. Een file server in zo’n systeem kan dan bijvoorbeeld een port hebben die gebruikt wordt voor verzoeken om een file te creëren of te openen, terwijl daarna voor elke geopende file een aparte port wordt gebruikt waar lees en schrijfopdrachten voor die file naar toe gestuurd worden.

Port:
Een abstract eindpunt voor communicatie.

In Win32 hebben we *message queues* die vergelijkbaar zijn met ports. Per window en per thread is er zo’n message queue[1]

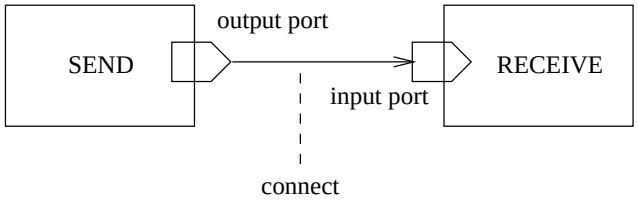
[1] Win16 heeft alleen message queues voor elk window. Een proces dat geen windows wil laten zien maar wel messages

Als alle communicatie via ports gebeurt, moeten antwoorden van een server ook naar een port van een cliënt gestuurd worden. Hiervoor kan een vaste port van de cliënt gebruikt worden, of er kan voor elk verzoek een tijdelijke port gemaakt worden waar het antwoord naar toe gestuurd wordt. Meestal worden deze antwoord ports automatisch door het O.S. verzorgd zonder dat de cliënt en server (of zender en ontvanger in het algemene geval) zich daar expliciet druk over hoeven te maken.

Ook bij ports doet zich de vraag voor hoe deze benoemd worden. Sommige systemen gebruiken port nummers per machine en een zender moet dan zowel het adres van de machine gebruiken als het nummer van de port (dit is de manier waarop Internet communicatie werkt). Op andere systemen hebben ports “namen” (meestal een heel groot nummer), en zoekt het systeem zelf uit waar de port zich bevindt. Er zijn zelfs systemen (geweest) waarbij ports per proces genummerd werden, en dan moeten een machineadres, een proces nummer en een port nummer opgegeven worden. In zo’n geval gaat een deel van de voordelen van het gebruik van ports verloren.

8.1.2 Getypeerde ports

Wannee voor elke soort boodschap een aparte port gebruikt wordt is het mogelijk om aan een port een *type* boodschap te verbinden. Met type bedoelen we hier een type zoals gebruikt in een programmeertaal. Een voordeel hiervan is dat de compiler kan controleren dat het juiste type boodschap naar de port gestuurd wordt. en dat de ontvanger ervan kan uitgaan dat de informatie in de ontvangen boodschap ook van het juiste type is. In zo’n geval moet er zowel in het zendende als in het ontvangende proces het type gecontroleerd worden, en het gemakkelijkst gaat dit als er zowel *output ports* als *input ports* bestaan. Het zendende proces stuurt dan de boodschappen naar zijn eigen output port en het ontvangende proces leest van zijn input port. De beide ports moeten met elkaar “verbonden” worden voor de communicatie begint (figuur 8.1).



Figuur 8.1: Input- en output ports

In zo’n geval kan de typecontrole in beide processen op compile tijd gebeuren, maar de controle dat de input port en de output port hetzelfde type hebben gebeurt dan op het moment dat de connectie plaatsvindt. Hiervoor is het nodig dat er in het systeem een manier is om type informatie uit te wisselen tussen verschillende processen, of tussen een proces en het O.S.

Getypeerde port:
Een port waarbij alleen boodschappen van een specifiek type (in de betekenis van een programmeertaal) toegestaan zijn.

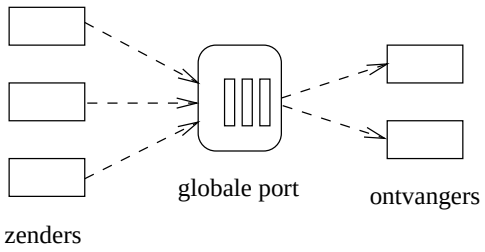
De communicatie hoeft niet te gaan tussen één zender en één ontvanger. Het is ook mogelijk om meer dan één output port aan één input port te koppelen.

ontvangen moet dan een onzichtbaar window hiervoor aanmaken.

8.1.3 Globale ports

Een verdere generalisatie van het port begrip is de z.g. globale port, ook wel mailbox genoemd. Bij een gewone port uit de vorige sectie, wordt weliswaar niet rechtstreeks naar de port gezonden, maar de port hoort nog wel bij een specifiek proces. Bij een globale port is de port helemaal losgemaakt van het ontvangende proces en kan meer dan één proces uit de port (mailbox) lezen. Dit zou bijvoorbeeld gebruikt kunnen worden om de werklast van een bepaalde service over meer dan één server te verdelen. Cliënten sturen dan hun verzoeken naar een bepaalde mailbox en een server die niets te doen heeft leest een verzoek en behandelt dat. Natuurlijk is dit concept ook bruikbaar buiten een client-server structuur, maar vooral in een client-server systeem kan dit een bijzonder nuttige voorziening zijn.

Globale port:
Een port waarbij er meer dan één ontvanger kan zijn, maar waarbij elke boodschap maar naar één van de ontvangers gaat.

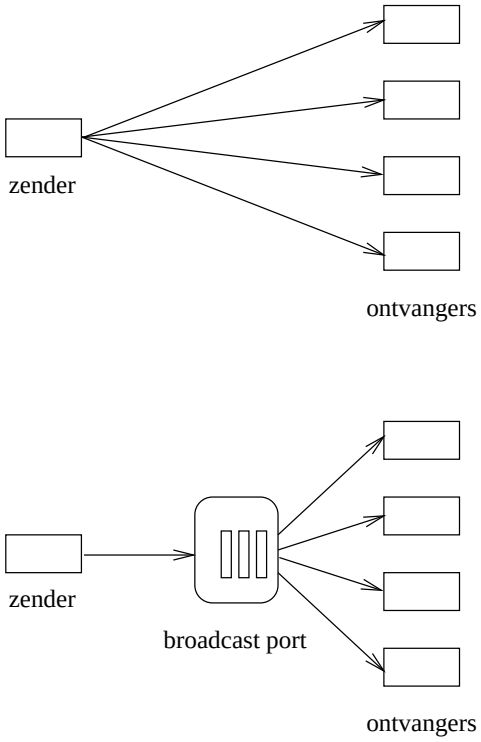


Figuur 8.2: Een globale port

8.1.4 Broadcast en multicast

Bij een globale port wordt een verzonden boodschap opgepikt door één van de “aangesloten” ontvangers, namelijk degene die het eerst een RECEIVE doet. Er zijn echter situaties waarin het wenselijk is dat een boodschap niet naar één, maar naar een hele verzameling ontvangers gaat. We spreken dan van een *broadcast* of *multicast*. Natuurlijk kan dit altijd gedaan worden door een boodschap naar een hele lijst van ontvangers (of ports) te sturen, maar het is gemakkelijker als het systeem hiervoor voorzieningen heeft. Dan kan namelijk de administratie welke processen of ports er in zo’n multicast groep behoren centraal geregeld worden. Bovendien is het in sommige gevallen noodzakelijk dat de verzonden boodschappen bij alle aangesloten processen in dezelfde volgorde aankomen, iets wat moeilijker te realiseren is als processen zelf de broadcast moeten verzorgen. Bijvoorbeeld als proces A een boodschap m1 verstuurt en proces B verstuurt boodschap m2, dan zou proces X best eerst m1 en daarna m2 kunnen ontvangen en proces Y eerst m2 en daarna m1. Als A en B op verschillende computers draaien, en X zit bij A op de machine en Y bij B is dat zelfs het meest waarschijnlijk. Bij een door het O.S. geregeld broadcast mechanisme is het mogelijk dat het O.S. ervoor zorgt dat X en Y beide de volgorde (m1,m2) of (m2,m1) krijgen, maar niet ieder een andere.

Eén va de manieren waarop het systeem dit kan doen is door een speciaal soort *broadcast port* of *group port* te leveren waarnaar de boodschappen in eerste instantie gestuurd worden. Vandaar stuurt



Figuur 8.3: Broadcast

het systeem dan de boodschappen naar alle ontvangers.

Broadcast:
Het gelijktijdig versturen van een boodschap naar alle ontvangers binnen een bepaald netwerk.

Multicast:
Het gelijktijdig versturen van een boodschap naar een gespecificeerde groep ontvangers.

Broadcast port:
Een port die meer dan één ontvanger kan hebben en waarbij elke boodschap naar *alle* ontvangers gaat.

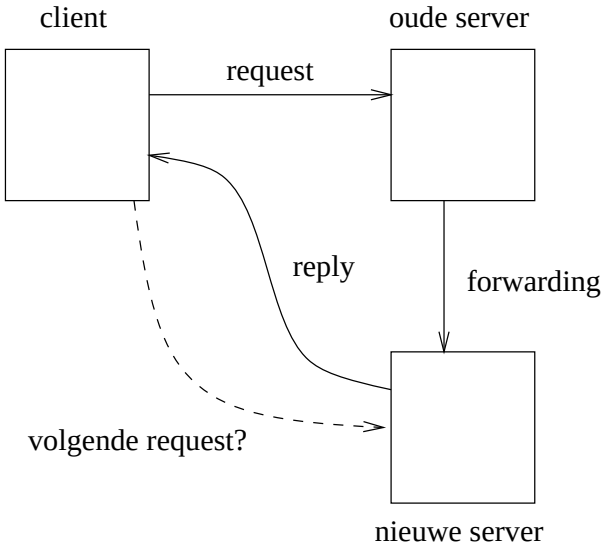
Let erop dat het verschil tussen een broadcast en een globale port is, dat bij een globale port slechts één van de ontvangers het bericht krijgt en bij een broadcast allemaal.

Voorbeelden waarbij een broadcast een belangrijk mechanisme is:

- het synchroniseren van klokken op verschillende computers in een netwerk
- Het bijhouden van de werklast van verschillende CPU's in een systeem (om het werk eerlijk te kunnen verdelen)
- Het bijhouden van kopieën van een file systeem of database, zodat in geval van een crash de data niet verloren gaat.

8.1.5 Message forwarding

Normaliter zal een server die een verzoek krijgt van een cliënt het antwoord naar deze cliënt terugsturen. Aan de andere kant betekent dit dat een cliënt het antwoord kan verwachten van het proces (of de port) waarnaar het verzoek gezonden is. Er zijn echter situaties waarin een omweg noodzakelijk is. Neen als voorbeeld een systeem waarin ports niet vast aan processen verbonden zijn, maar kunnen verhuizen van het ene proces naar het andere. Of een systeem waarbij een proces van de ene machine naar de andere kan verhuizen. In zo'n systeem kan het soms nuttig zijn als een deel van het werk van de ene machine naar de andere verschoven wordt. Als een file server het bijvoorbeeld te druk krijgt terwijl ergens anders een server machine niet veel meer te doen heeft, dan is het beter om een deel van het werk door te schuiven naar de minder belaste machine. Er kunnen echter nog cliënten zijn die "denken" dat ze met de oorspronkelijke server moeten communiceren. Deze server kan dan het verzoek gewoon doorsturen naar de nieuwe server, maar er is geen reden voor de nieuwe server om het antwoord via de oude server terug te laten gaan. Het is veel efficiënter om het antwoord direct naar de cliënt te laten gaan. Het doorsturen van boodschapen van de ene ontvanger naar de andere noemen we *message forwarding* en sommige systemen hebben voorzieningen om dit automatisch te doen. Het is dan natuurlijk handig als volgende verzoeken van hetzelfde soort direct naar de nieuwe ontvanger gaat. Of dit expliciet door de verzender moet gebeuren (die dit kan ontdekken als de nieuwe server als afzender in het reply bericht voorkomt), of dat dit door het O.S. gebeurt hangt helemaal af van het ontwerp van het systeem.



Figuur 8.4: Message forwarding

8.2 Timeouts bij communicatie

Een van de ontwerpbeslissingen bij een communicatie systeem is het gedrag bij communicatieproblemen. Wanneer alles goed gaat dan zal een verstuurde boodschap, of het nu een verzoek of een reply is, uiteindelijk de bestemming bereiken. Het is echter mogelijk dat een van beide partijen voortijdig

stopt, hetzij dat de machine waarop het proces draait ermee ophoudt, hetzij dat het proces een ontijdige dood sterft. Een andere mogelijke bron van problemen is het netwerk, dat plotseling overbelast kan worden, of waarbij zelfs de verbinding verbroken kan worden. In zo'n geval moeten we voorkomen dat een proces oneindig lang gaat wachten op een te ontvangen bericht. Sommige systemen hebben hiervoor de mogelijkheid om bij het verzenden of ontvangen van een boodschap een timeout op te geven. Waneer de operatie niet binnen de opgegeven tijd lukt, wordt deze afgebroken en krijgt het proces een foutcode terug. Het ligt dan aan het proces wat voor verdere actie ondernomen gaat worden. In sommige gevallen kan zelfs het O.S. de timeout opvangen, en actie ondernemen, bijvoorbeeld door naar een andere server te gaan zoeken.

Wanneer de waarde van de tijdslimiet expliciet opgegeven kan worden is het vaak zelfs mogelijk een vorm van *polling* te implementeren door als timeout de waarde 0 mee te geven. Een RECEIVE met timeout waarde 0 betekent dan dus feitelijk "kijk of er een boodschap in de queue aanwezig is". Op sommige systemen wordt een waarde van 0 echter geïnterpreteerd als "oneindig". Timeouts op een SEND opdracht hebben i.h.a. alleen betekenis bij synchrone operaties.

8.3 Sockets

Een socket is een generalisatie van een pipe die ook voor netwerk communicatie gebruikt kan worden. Het kan dus beschouwd worden als een pipe waarvan de einden niet noodzakelijk op dezelfde computer hoeven te zitten. Sockets zijn oorspronkelijk ontworpen als netwerk IPC mechanisme voor de Berkeley (BSD) versie van Unix, maar zijn intussen ook op O.S's als MS Windows en OS/2 de gangbare manier om netwerkcommunicatie te plegen. In tegenstelling tot gewone pipes kunnen willekeurige processen met elkaar communiceren, ook al stammen ze niet van hetzelfde proces af (anders zouden processen op verschillende computers niet met elkaar kunnen communiceren), en in tegenstelling tot named pipes hebben ze geen naam. In Unix kunnen sockets gebruikt worden alsof het files zijn (dus met read en write), maar er is ook een meer gestructureerde communicatie mogelijk die gebaseerd is op boodschappen (send en receive). Op sommige andere systemen is dit de enige manier.

Socket:
Een pipe-achtig mechanisme voor communicatie tussen verschillende computers of binnen een computer.

Sockets worden het meest gebruikt voor verbindingen via het Internet, en met gebruikmaking van de bijbehorende protocollen TCP/IP en UDP, maar ze kunnen ook gebruikt worden voor andere netwerken en protocollen. Het hangt van het O.S. af welke netwerken en protocollen aanwezig zijn op een bepaalde machine. De principes zijn echter steeds hetzelfde.

In tegenstelling tot een pipe is een socket een tweerichtingskanaal. De twee richtingen zijn wel onafhankelijk van elkaar te gebruiken. Op deze manier kunnen er dialogen tussen twee processen gevoerd worden. Veel Internet protocollen zijn op dergelijke dialogen gebaseerd. We zullen verderop hiervan een voorbeeld zien.

Bij een werkende socket horen een aantal parameters:

- de netwerksoort
- het te gebruiken protocol

- het netwerkadres van de machine waarop het einde zit (*)
- het proces dat het einde heeft (*)
- een *port nr* voor het einde (*)

De met (*) aangegeven parameters zijn voor elk uiteinde aanwezig.

De *port nrs* verdienen enige uitleg. In de Internet protocollen worden nummers (max 16 bits) gebruikt om bepaalde diensten aan te geven. Zo wordt bijvoorbeeld voor file transfer het nummer 21 gebruikt en voor het WWW (http) nummer 80. Wanneer een aanvraag voor een connectie met een bepaald nummer bij het O.S. binnenkomt dan weet het O.S. bij welk programma dit nummer hoort. De verbindingsaanvraag wordt dan naar dat betreffende proces gestuurd. Op deze manier kunnen verschillende O.S's dezelfde diensten op heel verschillende manier implementeren.

Het netwerk adres is voor de Internet protocollen momenteel een 32-bits nummer[2]. De meeste mensen werken echter liever met namen zoals "www.cs.ruu.nl". Er zijn in het systeem voorzieningen om bij een bepaalde naam het nummer te vinden, we zullen deze later beschrijven.

Om een socket te kunnen gebruiken moet meer werk verricht worden dan bij een pipe, voornamelijk omdat er zoveel parameters zijn. Het opzetten van een socket gebeurt in een aantal stappen:

1. creëer de socket (waarbij de netwerk- en protocolsoort aangegeven wordt)
2. laat de socket weten hoe wij onszelf aan de buitenwereld bekend willen maken
3. vertel de socket met wie aan de andere kant contact gemaakt moet worden.

Er wordt onderscheid gemaakt tussen *connection-oriented* en *connection-less* communicatie. Bij *connection-oriented* wordt van te voren een "verbinding" opgezet tussen twee processen (op de manier zoals je de telefoon gebruikt), waarna over deze verbinding gecommuniceerd kan worden. Na afloop wordt de verbinding verbroken. Deze manier van communicatie heeft de voorkeur als grotere hoeveelheden data overgeheveld moeten worden, bijvoorbeeld bij het transporteren van een file. Het opzetten van de verbinding kost extra tijd, maar daarna hoeft niet meer per boodschap gezegd te worden waar deze heen moet. Bovendien garandeert het systeem dat opeenvolgende boodschappen over de verbinding in de juiste volgorde aankomen en wordt er voor gezorgd dat boodschappen niet verloren gaan of verdubbeld worden.

Bij communicatie die verbindingsloos is, wordt zomaar een boodschap van het ene proces naar het andere gestuurd. Elke boodschap moet dan dus het ontvangeradres meekrijgen. Deze manier van communicatie is handig als slechts af en toe korte boodschappen overgestuurd moeten worden. Bijvoorbeeld wanneer computers de tijd met elkaar uitwisselen om elkaars klokken gelijk te kunnen laten lopen. Of wanneer ze samen bij willen houden welke gebruikers er op het netwerk aanwezig zijn.

In veel situaties is het belangrijk om onderscheid te maken tussen *cliënten* en *servers*: Een cliënt is een proces dat een bepaalde dienst uitgevoerd wil hebben door een bepaald soort proces op een andere machine. Denk bijvoorbeeld aan een WWW browser die een bepaald HTML document van een machine wil opvragen. De cliënt weet met welk ander proces gecommuniceerd moet worden[3].

[2] Er wordt gewerkt aan een uitbreiding naar 128-bits nummers omdat door de sterke groei van het Internet de 32-bits nummers te krap beginnen te worden

[3] Beter gezegd: welk port nummer

Aan de andere kant weet de server niet met welke cliënten gecommuniceerd zal gaan worden, want deze kunnen overal vandaan komen, en het doet er zelfs niet toe wat voor soort programma aan de andere kant van de socket staan. Anderzijds zal de server moeten vertellen welk port nummer erbij hoort.

Deze verschillen zijn alleen relevant voor het opzetten van de verbinding, daarna is er geen verschil meer. Beide kunnen dan lezen en beide kunnen schrijven op de verbinding. In sectie 8.5 geven we een voorbeeld van beide met uitleg van de verschillende aanroepen die nodig zijn.

8.4 Remote Procedure Call

Bij het aanvragen van een dienst door een cliënt aan een server is de algemene werkwijze als volgt:

- stuur een bericht met het verzoek, waarin op de een of andere manier het soort verzoek en de benodigde parameters opgeslagen zijn.
- wacht op een antwoord van de server, en haal het resultaat en/of de status uit de boodschap

Of de server op dezelfde machine draait of op een andere als de cliënt maakt hierbij niet zoveel uit. Als we dit vergelijken met het gebruiken van andere diensten, zoals die in in bibliotheek (bijvoorbeeld het berekenen van de lengte van een string), of die van het O.S. dan zien we dat daar i.h.a. een heel andere stijl van werken gebruikt wordt: deze diensten worden aangevraagd door het aanroepen van een functie, procedure of methode, en het resultaat komt terug als functieresultaat of via een meegegeven (pointer naar) een parameter van de functie:

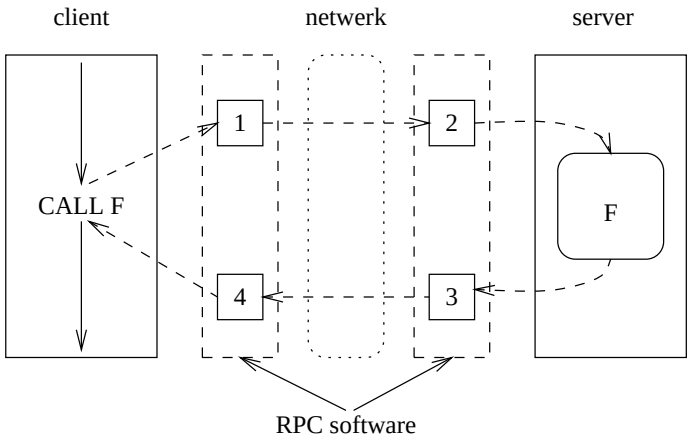
```
y = length (x);
write (fileno, buf, length);
```

Omdat het gebruiken van functie- of procedure aanroepen een stijl van werken is die veel mensen kennen, en die klaarblijkelijk een prettige manier van werken is, waarvan bovendien goed bekend is hoe deze in correctheidsbewijzen gebruikt kan worden, heeft men gezocht naar een mogelijkheid om deze stijl van werken ook bij communicatie tussen verschillende processen en zelfs tussen verschillende computers te gebruiken. Hier is een manier van werken uitgekomen die *Remote Procedure Call* (RPC) genoemd wordt.

Bij RPC wordt net gedaan alsof de cliënt een procedure aanroept voor de gevraagde dienst maar de procedure bevindt zich niet in hetzelfde proces als de cliënt, maar in de server. Onder water zal er dus het een en ander moeten gebeuren om de RPC om te zetten in de benodigde boodschappen:

Remote Procedure Call:
Een communicatiemechanisme dat lijkt op een normale functie-, procedure- of methodeaanroep.

In figuur 8.5 zien we de stadia waar de RPC doorheen gaat. Bij de cliënt wordt een RPC gedaan, vermomd als functie- of procedureaanroep. Bij punt 1 wordt een boodschap samengesteld waarin voldoende informatie zit voor de server, deze wordt (al dan niet via het netwerk) verstuurd naar de server. Bij de server wordt de informatie uit deze boodschap bekeken en er wordt een aanroep naar de echte procedure gedaan (punt 2). De procedure doet zijn werk en na afloop pakt de RPC software



Figuur 8.5: RPC

het resultaat in een boodschap in (punt 3), die weer teruggestuurd wordt. Tenslotte wordt bij punt 4 de echte terugkeer gedaan naar het punt in de cliënt waar de RPC gedaan was.

Wanneer voor de communicatie gebruik gemaakt wordt van een standaard protocol zoals TCP, dan zullen er intussen ook nog de diverse acknowledge boodschappen rondgestuurd worden. Wanneer een speciaal RPC protocol gebruikt wordt dan kan er hier en daar eventueel op bezuinigd worden. Bijvoorbeeld de reply van de RPC kan dienst doen als acknowledge voor de aanroep. De reply zelf moet dan nog wel een acknowledge krijgen. Het bijbehorende protocol wordt dan RRA (Request–Reply–Acknowledge) genoemd. Wanneer de functie lang kan duren dan is het beter om ook op de aanroep een acknowledge te genereren en het protocol heet dan RARA.

Om fouten te kunnen detecteren (bijvoorbeeld het verloren gaan van een boodschap. het stoppen van een machine of het server proces) moeten we ook timers gebruiken. Bij een timeout moet dan een boodschap herhaald worden of teruggemeld worden aan het proces dat er een probleem is. We hebben zowel bij de request als bij de reply een timer nodig; de timers worden gereset als blijkt dat de boodschap goed is overgekomen (doordat een ACK wordt ontvangen of doordat een reply op de request wordt ontvangen).

Bij een timeout van de request zal in eerste instantie geprobeerd worden om deze nog eens te sturen. Hierbij lopen we het risico dat de server de request nogmaals uitvoert, bijvoorbeeld omdat niet de request verloren gegaan was, maar de ACK of de reply. Voor sommige verzoeken is het niet erg dat de request nog eens gedaan wordt (bijvoorbeeld het uitrekenen van een cosinus), voor andere echter wel (bijvoorbeeld het overboeken van een bedrag van een bankrekening naar een andere rekening). Welke discipline gebruikt wordt bij een fout is dus sterk afhankelijk van de operatie. Het is zelfs afhankelijk van de manier waarop de operatie gespecificeerd wordt.

Neem bijvoorbeeld als operatie een schrijfopdracht naar een file. Als we de gangbare Unix manier nemen dan betekent `write (fileno, buffer, lengte)` dat de inhoud van de buffer geschreven moet worden op de plaats waar de “file pointer” staat, en dat deze filepointer daarna opgeschoven moet worden. In dat geval betekent twee keer schrijven iets anders dan één keer. Als echter behalve deze parameters ook de positie in de file nog meegegeven wordt dan heeft tweemaal schrijven hetzelfde effect als éénmaal. We zeggen dan dat de operatie *idempotent* is. Dit soort operaties verdient meestal de voorkeur, maar dit is niet altijd mogelijk.

idempotent:
De eigenschap van een operatie dat één keer uitvoeren hetzelfde effect heeft als meer dan een keer uitvoeren.

Het Network File System (NFS), een file server systeem dat veel op Unix gebruikt wordt is gebaseerd op deze methode, waarbij elke operatie alle benodigde informatie bij zich heeft. We hebben dit al eerder een *stateless server* genoemd (sectie 4.10.2). Het stateless server concept houdt dus in dat er idempotente operaties gebruikt worden, maar gaat in feite nog een stapje verder omdat het bestand is tegen het uitvallen van de server.

Een uitvoering van de RPC waarbij gegarandeerd wordt dat de operatie in ieder geval uitgevoerd wordt, maar waarbij het kan voorkomen dat de operatie meer dan eens gedaan wordt, noemen we *At Least Once* (semantiek)

Het omgekeerde waarbij de operatie nooit meer dan één keer uitgevoerd wordt, maar waarbij het kan voorkomen dat de operatie niet uitgevoerd wordt, heet dan *At Most Once* (semantiek). Dit kan eenvoudig gerealiseerd worden door niet de request te herhalen bij een fout. Er zijn niet echt veel toepassingen waarbij dit zinnig is, maar soms kan dit nuttig zijn. Bijvoorbeeld wanneer het doel is om regelmatig wat informatie van de cliënt naar de server te sturen, of omgekeerd (bijvoorbeeld over de belasting van de server). In zo'n geval mag er best eens een keer iets gemist worden.

In veel gevallen is het echter belangrijk dat de operatie precies één keer uitgevoerd wordt (*Exactly Once* semantiek). Om dit voor elkaar te krijgen, moet er ook bij het opsturen van de request een volgnummer gegenereerd worden, en dit moet meegestuurd worden. Bij het ontvangen moet dan gecontroleerd worden of dit volgnummer al gebruikt is. De protocollen die hiervoor nodig zijn, zijn gelijk aan degene die bij een datalink protocol of een transportlaag protocol zoals TCP gebruikt worden. We zullen er daarom hier niet verder op ingaan.

Wat nog wel een probleem kan geven is de situatie dat òf de cliënt òf de server stopt en opnieuw gestart moet worden. In dat geval gaan de volgnummers verloren en werkt bovenstaande methode niet meer. Er zullen dan weer beter manieren gebruikt moeten worden, bijvoorbeeld het bijhouden van informatie op permanent geheugen (harde schijf) en/of het synchroniseren van volgnummers met behulp van hardware klokken.

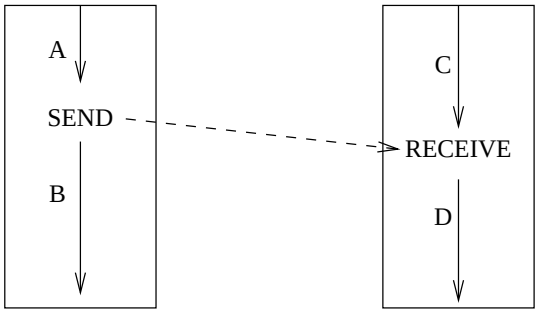
8.4.1 Transparante RPC

Zoals uit het voorafgaande is gebleken kunnen we wel proberen om RPC zoveel mogelijk te laten lijken op een gewone functie- of procedureaanroep, er blijven echter verschillen. De belangrijkste verschillen worden veroorzaakt door de volgende omstandigheden:

- Netwerk fouten waardoor we de verschillende *-Once semantieken moesten onderscheiden. Bij een gewone aanroep kan dit niet voorkomen.

- Bij een RPC zijn verschillende processen en/of machines betrokken die afzonderlijk fouten kunnen maken. Bij een gewone aanroep zitten de aanroeper en de aangeroepen procedure in hetzelfde proces, en als er een fout optreedt, dan verdwijnen beide dus. Het is daarbij onmogelijk dat er een losse aanroeper of aangeroepene blijft rondhangen die de andere partij is kwijtgeraakt.

- Synchronisatie: Bij RPC vooral als de aanroeper en de aangeroepene op verschillende machines draaien is het begrip “gelijktijdig” niet altijd goed gedefinieerd. Binnen een proces kan van elke twee gebeurtenissen ondubbelzinnig vastgesteld worden welke het eerst is opgetreden en welke het laatst. Bij verschillende processen binnen één computer meestal ook wel, maar bij processen op verschillende systemen niet meer. Althans niet op een manier waarop de programma’s zelf dit ondubbelzinnig kunnen vaststellen. Zo is in figuur 8.6 wel zeker dat punt B later is dan A (omdat dit een sequentiële executie is), evenzo is D later dan C. Verder geldt dat D later is dan A, omdat het ontvangen van een boodschap alleen maar later kan gebeuren dan het verzenden. Maar wat de tijdsrelatie tussen B en C is, kan niet ondubbelzinnig worden vastgesteld uit deze informatie. De tijd in een gedistribueerde toepassing heeft dus een *partiële ordening*.



Figuur 8.6: Partiële ordening van tijdstippen binnen een gedistribueerde toepassing

- Er is geen globale toestand die ondubbelzinnig vastligt. De toestand van een gedistribueerde toepassing ligt verdeeld over een aantal computers en de veranderingen erop zijn niet gesynchroniseerd. Bovendien is het moeilijk te realiseren om een bepaald proces dat deelneemt, een totaalbeeld van de toestand te laten zien. Daarvoor moet namelijk informatie uitgewisseld worden tussen de onderdelen maar tijdens het uitwisselen kunnen delen van de toestand weer veranderen.

Omdat een gedistribueerde toepassing op zoveel punten verschilt van een “gewone”, is het de vraag of het zo zinvol is om een RPC precies eender te gebruiken als een gewone aanroep. In feite is dit een ontwerpbeslissing die een aanbieder van RPC (wat best het O.S. zou kunnen zijn) moet maken. Wanneer we besluiten om de RPC precies eender te maken als de gewone aanroep hebben we het voordeel dat dit voor de programmeur gemakkelijker is. Aan de andere kant is het dan niet meer mogelijk om aan te geven welk soort semantiek een bepaalde RPC moet gebruiken. Het systeem moet daar dan standaard beslissingen voor nemen.

Sommige systemen hebben ervoor gekozen om voor RPC weliswaar iets te gebruiken dat heel veel lijkt op de gewone procedure aanroep, maar toch anders opgeschreven moet worden. Wanneer de RPC er precies zo uitziet als een gewone aanroep dan spreken we van een *transparante RPC*. Een veel gebruikt systeem om een transparante RPC te realiseren is met gebruikmaking van z.g. *stubs*. Hierbij wordt de RPC en de aangeroepen procedure gewoon gecompileerd als elke andere. Dus bij de cliënt wordt een gewone procedure aanroep gegenereerd door de compiler, met de normale naam van de procedure. Dit werkt natuurlijk alleeen als we er ook een procedure tegenaan linken met diezelfde naam, die bovendien nog eens dezelfde parameters moet accepteren en hetzelfde resultaat moet afleveren. We doen dit door een soort “dummy” procedure te maken die hier inderdaad aan voldoet. Dus als

we een RPC naar `write` willen maken, leveren we een "dummy" procedure `write` op. Deze dummy procedure noemen we de *client stub*.

Transparante RPC:
RPC die op exact dezelfde wijze opgeschreven wordt als een gewone functie-, procedure- of methodeaanroep.

De client stub is niet helemaal dummy, want wat deze doet, is de meegegeven parameters bij elkaar verzamelen en in een geschikt formaat inpakken in een boodschap die naar de server gestuurd wordt. Dit proces heet *marshallen*.

Aan de server kant gebeurt iets soortgelijks maar dan precies tegenover gesteld. Daar bevindt zich de echte procedure (in dit voorbeeld dus `write`), en die zal op een gegeven moment aangeroepen moeten worden. Daarom wordt er op de server ook een *stub* gemaakt (dit keer *server stub* of ook wel eens *skeleton* geheten. Deze stub ontvangt de boodschap met de "gemarshalde" parameters en pakt deze weer uit, wat dan *unmarshallen* genoemd wordt. Wanneer de parameters op een juiste manier neergezet zijn (bijvoorbeeld op de stack) dan roept de stub de echte procedure aan. Deze is zich op geen enkele manier ervan "bewust" dat de parameters van een andere machine of een ander proces komen, en voert zijn werk uit, om daarna terug te keren tot zijn aanroeper: de server stub. De server stub pakt het resultaat op en zet dit in een reply buffer (ook hier heet het proces om dit te doen *marshallen*). Het wordt weer teruggestuurd en daar door de client stub weer geunmarshalled en de stub keert terug naar zijn aanroeper. Ook deze aanroeper, de cliënt is zich op geen enkele manier "bewust" geweest van het feit dat de functie in feite op een andere computer of in een ander proces is uitgevoerd.

De stubs worden meestal met behulp van een speciale *stub compiler* gemaakt. Deze genereert vanuit de declaratie van de procedure beide stubs. De type-informatie van de procedure wordt gebruikt om de marshal instructies te genereren.

Stub:
Een stukje programma dat opgenomen wordt in een programma en dat de communicatie verzorgt bij een RPC.

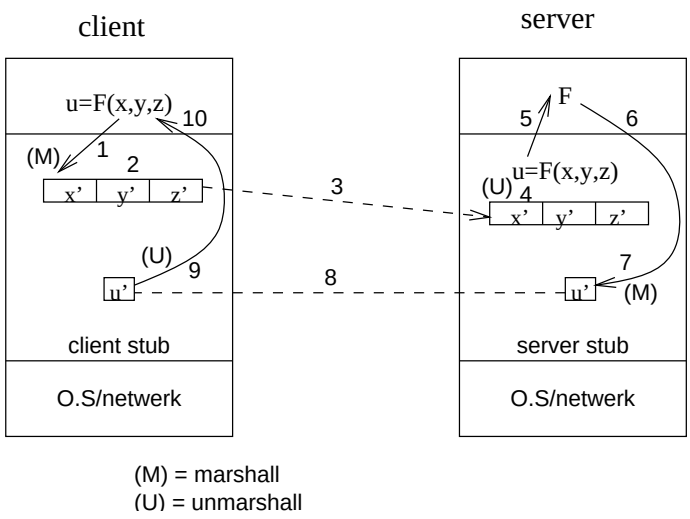
Skeleton:
Stub aan de serverkant.

Marshalling:
Het omzetten van gegevens (parameters en resultaten) in een vorm die geschikt is om over het netwerk gestuurd te worden.

Unmarshalling:
Het terugzetten van ontvangen gegevens naar het normale machineformaat.

De gewone type informatie zoals die bijvoorbeeld in Pascal, C of Java gebruikt wordt is niet altijd voldoende voor het maken van de stubs. De volgende problemen kunnen alleen opgelost worden als extra informatie voorhanden is:

- Wat doen we met *reference* parameters en pointers? Een pointer (een reference parameter is in feite een pointer in vermomming), is niet erg zinvol om over te sturen. De pointer wijst naar een



Figuur 8.7: Client en server stubs in actie:

1. RPC aanroep
2. marshallen van parameters
3. request boodschap
4. unmarshallen van parameters
5. aanroep echte functie
6. terugkeer echte functie
7. marshallen resultaat
8. reply boodschap
9. unmarshallen resultaat
10. terugkeer naar aanroeper

adres in de cliënt (wanneer het in een parameter is) en de server heeft daar geen toegang toe. Bij het marshallen moet een pointer dus niet zelf opgestuurd worden maar de data waar de pointer naar wijst. Bij het unmarshallen wordt dan een nieuwe pointer gemaakt die naar de kopie van de data wijst. Wanneer de pointer wijst naar een grotere datastructuur, zoals een lijst, boom of graaf, dan komt de vraag aan de orde hoeveel er gekopieerd moet worden: alleen de data waar de pointer naar wijst of moeten de pointers die daarin zitten ook weer gevolgd worden? En wanneer houden we dan op? Dit kan alleen goed gedaan worden als er extra informatie aan de stub compiler gegeven wordt.

- Wat doen we met parameters die door de aangeroepen procedure veranderd worden? Meestal gaat het hierbij ook om data die via pointers benaderd wordt. Moeten de veranderde waarden teruggestuurd worden?

- Wat doen we met grote objecten? Bijvoorbeeld een groot array? Als de aangeroepen procedure slechts een paar elementen van het array gebruikt is het erg inefficiënt om het hele array op te sturen. Misschien kunnen we aan de stubcompiler aangeven welke array elementen gebruikt worden en/of welke in ieder geval niet.

- Verschillende datatypes worden op verschillende computers op verschillende manier weergege-

ven. Sommige computers hebben 32-bits integers, andere 64-bits. OP sommige CPU's staat de bytes van een integer in een andere volgorde dan op een andere. Floating point getallen worden verschillend weergegeven etc. In een werkelijk transparant geval moet ervoor gezorgd worden dat er in zo'n geval de juiste omzettingen plaatsvindt, want de aanroeper en de aangeroepenen zijn meestal niet geïnteresseerd in een specifiek bitpatroon, maar in de betekenis ervan, bijvoorbeeld het getal 237 of 3.14 of een tekst "abc". Ook hiervoor is de type-informatie nodig.

8.5 Voorbeelden van clients en servers met sockets

We sluiten dit hoofdstuk af met twee voorbeelden van simpele client-server systemen, waarvoor we gebruik maken van sockets. Het eerste voorbeeld maakt gebruik van Unix sockets, die in feite gewoon "named pipes" zijn, maar dan verpakt als sockets.

De voorbeelden zijn geschreven in de taal "Python" een objectgeorienteerde "scripting" taal. De reden hiervoor is dat ze er meer uitzien als "pseudocode" en ze kunnen ook het beste als zodanig gelezen worden. Het zijn echter echt werkende programma's. Het belangrijkste verschil is dat sockets e.d. objecten zijn, en dat er daarom `s.connect(x)` gebruikt wordt, terwijl in C connect(s,x) gebruikt zou worden.

```
1       from socket import *
2       FILE = 'blabla'
3       s = socket(AF_UNIX, SOCK_STREAM)
4       s.connect(FILE)
5       s.send('Hello, world')
6       data = s.recv(1024)
7       s.close()
8       print 'Received', `data`
```

Figuur 8.8: Simpele client met Unix socket

in figuur 8.8 wordt de cliënt getoond. Eerst creëren we een socket (regel 3), waarbij we zeggen dat dit een Unix socket moet worden en dat de socket een "stream" verzorgt (dit betekent in feite dat de communicatie verbindingsgericht wordt; een alternatief is SOCK_DGRAM voor datagram communicatie).

Regel 4 "connect" de socket aan een naam, waarbij de server dit ook gedaan moet hebben. Hierdoor worden de verschillende sockets van elkaar onderscheiden. Tenslotte sturen we een boodschap naar de server (regel 5, het `send` commando, en wachten op een antwoord van de server (regel 6, het `recv` commando). Het opgeleverde antwoord wordt tenslotte afgedrukt, nadat we de socket gesloten hebben (regel 8 resp. 7).

De bijbehorende server is te zien in figuur 8.9

Bij de server gebruiken we niet `connect` maar `bind` (regel 4). Het verschil is dat `bind` de verbinding tussen socket en naam maakt, terwijl `connect` een bestaande verbinding opzoekt. Ter controle wordt de naam nog even afgedrukt.

Een tweede verschil is dat de server eerst een `listen` op de socket doet (regel 6). Deze aanroep wacht tot er een cliënt in het systeem een connect doet en komt dan pas terug. De parameter van `listen`

```
1       from socket import *
2       FILE = 'blabla'
3       s = socket(AF_UNIX, SOCK_STREAM)
4       s.bind(FILE)
5       print 'Sock name is: ['+s.getsockname()+']'
6       s.listen(1)
7       conn, addr = s.accept()
8       print 'Connected by', addr
9       while 1:
10          data = conn.recv(1024)
11          if not data: break
12          conn.send(data)
13      conn.close()
```

Figuur 8.9: Simpele server met Unix socket

is het maximale aantal connecties dat het systeem wil bufferen. De `listen` aanroep keert pas terug als er een connectie staat te wachten, echter dan is er nog geen connectie gemaakt. Hiervoor is de `accept` aanroep nodig. Door `accept` geeft de server te kennen de connectie te willen accepteren. Deze aanroep levert een nieuwe socket op, en deze socket is verbonden met de socket van de cliënt. In regel 9–12 wordt nu voortdurend van de socket gelezen (recv) tot er een end-of-file optreedt (dit gebeurt als de socket aan de andere kant gesloten wordt). Omdat dit een simpel voorbeeld is stuurt de server alleen maar alle ontvangen data terug, een echte server zou natuurlijk eerst een bewerking uitvoeren. We zien dus dat bij de server er twee sockets zijn: Een wordt gebruikt om connecties te accepteren, en de andere om de werkelijke communicatie te doen.

In dit simpele voorbeeld is dat op zich niet zo zinvol, maar in het algemene geval geeft dit de mogelijkheid voor de server om een nieuwe thread of een nieuw proces op te starten dat de communicatie doet, terwijl op de originele socket nieuwe verbindingen geaccepteerd worden. In dit simpele voorbeeld kan een cliënt die niet snel genoeg de socket sluit, alle andere cliënten blokkeren.

In het volgende voorbeeld gebruiken we Internet sockets, en starten we ook nieuwe processen op in de server. Bovendien gebruiken we iets meer van een protocol. Dit is een simpele HTTP server, die ook hier echter geen HTML documenten van files haalt, maar wel een HTML document opstuurt, dat echter een kopie van de request van de cliënt bevat. We geven geen code voor de cliënt, want elke WWW browser kan als cliënt gebruikt worden.

Het belangrijkste verschil voor de cliënt is echter het volgende:

```
s = socket(AF_INET, SOCK_STREAM)
s.connect(host, port)
s.send('GET filenaam\r\n')
```

Bij de socket creatie wordt aangegeven dat het een Internet socket betreft, en bij de `connect` moet een host (IP adres) en een port nummer opgegeven worden. Het IP adres kan afgeleid worden uit de hostnaam (die i.h.a. uit de URL gehaald wordt) door middel van de aanroep `gethostbyname`. Het portnummer is meestal vast gegeven; voor http is 80 vastgesteld maar omdat dit geen officiële server is heb ik hier 8080 genomen. Tenslotte stuurt de cliënt een “GET” opdracht met de gevraagde filenaam als parameter.

Hier volgt de server:

```
1       from socket import *
2       import sys, os, string, time
3       host = ''
4       port = 8080
5       header = """HTTP/1.0 200 OK
6       Server: httpd.py
7       MIME-version: 1.0
8       Content-Type: text/html
9       """
10      bodystart="""<html><head><title>HTTPD.PY response</title></head>
11      <body><h1>HTTPD.PY response</h1>
12      <h2>This is a copy of your request</h2>
13      <pre>
14      """
15      bodyend="""</pre>
16      </body></html>
17      """
18      newline="\r\n"
19
20      active_children = []
21
22      def collect_children():
23          """routine to wait for died children."""
24          global active_children
25          while active_children:
26              pid, status = os.waitpid(0, os.WNOHANG)
27              if not pid: break
28              active_children.remove(pid)
29
30      s = socket(AF_INET, SOCK_STREAM)
31      s.bind(host, port)
32      s.listen(SOMAXCONN)
33      while 1:
34          conn, addr = s.accept()
35          pid = os.fork()
36          if pid == 0: # child
37              inp = conn.makefile()
38              data = string.rstrip(inp.readline())
39              req = string.split(data)
40              conn.send(header)
41              now = time.ctime(time.clock()) + newline
42              conn.send("Date: " + now)
43              conn.send("Last-modified: " + now)
44              conn.send(newline)
45              if req[0] == "GET":
```

```
46                  conn.send(bodystart)
47                  while 1:
48                      conn.send(data+newline)
49                      data = string.rstrip(inp.readline())
50                      if not data: break
51                  conn.send(bodyend)
52              inp.close()
53              sys.exit(0)
54          else: # parent
55              active_children.append(pid)
56          conn.close()
57          collect_children()
```

Een HTTP server stuurt eerst een “header”, die niet getoond wordt in de browser. Deze wordt gedefinieerd in regel 5–8, en verstuurd in regel 40. De header is vergelijkbaar met de header in email en nieuws berichten. De body bevat de echte HTML, en wordt gescheiden van de body door een lege regel. De body wordt in dit voorbeeld begonnen met een aantal HTML codes (regel 10-13) en afgesloten met de codes uit regel 15–16.

In regel 30–34 zien we de code voor het creëren van de socket, het binden van de naam, `listen` en `accept` zoals in het vorige voorbeeld. Ook hier zien we dat `bind` een host en een port nummer nodig heeft. In dit voorbeeld wordt “host” leeg gelaten wat betekent dat de locale host genomen wordt. In tegenstelling tot de eerste server starten we hier echter voor iedere binnenkomende connectie een kindproces met behulp van `fork` (regel 35-36). In het kind doen we nu de verdere communicatie, waarna het kind stopt (regel 37–53).

In regel 42–44 worden nog een paar header regels verstuurd (o.a. de tijd), en de lege regel die de header van de body scheidt. De body bestaat in dit voorbeeld uit een kopie van de binnenkomende data, met wat HTML codes eromheen. In dit voorbeeld wordt de input van de socket gedaan alsof de socket een file is (regel 37, 38, 49). Dit kan wel op Unix omdat een socket daar ook als een file nummer gegeven wordt. Op MS Windows en andere systemen kan dat echter niet en moet dus `recv` gebruikt worden zoals in het eerste voorbeeld.

Regel 20–28 zijn bedoeld om de kindprocessen die opgestart worden telkens als er een connectie binnenkomt, weer op te ruimen. Dit wordt gedaan omdat er anders zombie processen blijven hangen. Het ouder proces doet niets anders dan deze opruimoperatie, om daarna weer op een nieuwe connectie te wachten.

Tenslotte geven we hierbij dezelfde server in Java. Hierbij gebruiken we echter threads in plaats van processen voor het afhandelen van de requests. M.a.w. voor elke binnenkomende request wordt een aparte thread gemaakt. Er zijn nog enkele verschillen die met de implementatie van sockets in Java te maken hebben:

1. Java maakt expliciet onderscheid tussen de socket die de verbindingen ontvangt (`ServerSocket`) en de gewonde sockets. Zie regel 24–35.

2. De bind opdracht is in Java samengevoegd met de aanmaak van de socket (op regel 27).

```
1   import java.net.*;
```

```
2   import java.io.*;
3   import java.util.Date;
4   import java.text.DateFormat;
5
6   class HTTPServer implements Runnable {
7
8     Socket mysocket;
9     String header = "HTTP/1.0 200 OK\nServer: HTTPServer.java\n"+
10                    "MIME-version: 1.0\nContent-Type: text/html";
11    String bodystart = "<html><head>\n"+
12              "<title>HTTPServer.java response</title>\n"+
13              "</head><body><h1>HTTPServer.java response</h1>\n"+
14              "<h2>This is a copy of your request</h2>\n<pre>\n";
15    String bodyend = "</pre>\n</body></html>\n";
16
17    HTTPServer(Socket s) {
18          mysocket = s;
19    }
20
21    // This is for starting the server
22    public static void main(String args[]) {
23      int portnr = 8080;
24      ServerSocket in = null;
25
26      try {
27        in = new ServerSocket(portnr);
28      } catch (IOException e) {
29        System.out.println ("Cannot create ServerSocket");
30            System.exit(1);
31      }
32
33      while (true) {
34        try {
35          Socket s = in.accept();
36          // start thread
37          new Thread(new HTTPServer(s)).start();
38        } catch (Exception e) {
39            System.out.println ("Cannot accept");
40        }
41      }
42    }
43
44    // The rest is used for the threads.
45    public void run() {
46      BufferedReader inp = null;
47      PrintStream out = null;
48
```

```
49      try {
50        inp = new BufferedReader(
51                new InputStreamReader(mysocket.getInputStream()));
52        out = new PrintStream(mysocket.getOutputStream(), true);
53
54        out.println(header);
55        Date now = new Date();
56        String datetime =
57                    DateFormat.getDateTimeInstance().format(now);
58        out.println("Date: " + datetime);
59        out.println("Last-modified : " + datetime);
60        out.println("\n");
61
62        String line = inp.readLine();
63        if (line.substring(0,4).equals("GET ")) {
64           out.println(bodystart);
65           while (true) {
66             line = inp.readLine();
67             if (line == null || line.equals("")) break;
68             out.println(line);
69           }
70           out.println(bodyend);
71           out.close();
72        }
73      } catch (IOException e) {}
74      try {
75        if (inp != null) inp.close();
76        mysocket.close();
77      } catch (IOException e) {}
78    }
79  }
```

8.6 Opgaven

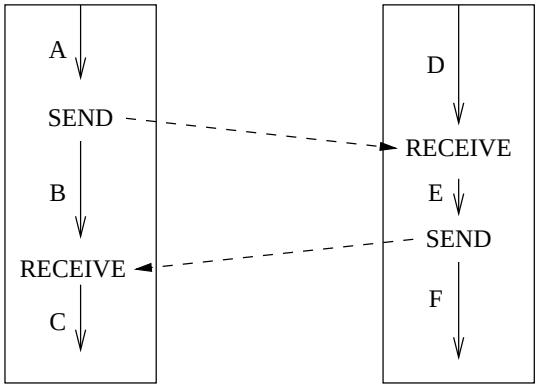
1. Geef nog twee toepassingen van broadcast/multicast die niet in het dictaat genoemd worden.
2. Geef een toepassing van een globale port. Leg uit waarom een globale port hier wenselijk is.
3. Geef een tijdschema van de boodschappen die heen en weer gaan tussen cliënt en server bij RPC met het RRA protocol. Hetzelfde voor het RARA protocol.
4. Welke van de volgende operaties zijn *idempotent*? Beargumenteer uw antwoorden.
   (a) Het overmaken van een bedrag van bankrekening A naar bankrekening B.
   (b) Het ophalen van een file en deze opslaan op de harde schijf.
   (c) Het opvragen van klantgegevens in een database en het resultaat op het scherm tonen.

(d) Het plaatsen van een bestelling door een klant.

5. Welke waarden worden er bij RPC in de cliënt stub gemarshalled? En welke geunmarshalled? En welke in de server stub?

6. Bepaal de partiële ordening van de tijdstippen in het schema hiernaast. Welke tijdsvolgordes liggen ondubbelzinnig vast en welke niet?

7. Wat staat er in de 'header' die de Java server in sectie 8.5 naar de browser stuurt?



Index