

Systeemprogrammatuur

Inleiding tot GNU/Linux

Chris Raymaekers
chris.raymaekers@luc.ac.be

Kris Luyten
kris.luyten@luc.ac.be

Karin Coninx
karin.coninx@luc.ac.be

academiejaar 2001 - 2002

bijdragen door:

Jori Liesenborgs (jori@lumumba.luc.ac.be)

Tom Van Laerhoven (tom.vanlaerhoven@luc.ac.be)

Stysteemprogrammatuur inleiding tot GNU/Linux

Kris Luyten



WORLD DOMINATION
THROUGH
WORLD COOPERATION

academiejaar 2001-2002
Limburgs Universitair Centrum
Expertisecentrum Digitale Media

Inhoudsopgave

Inhoudsopgave	1
Lijst van Figuren	5
Lijst van Tabellen	6
Lijst van Code Listings	7
1 Inleiding	9
2 Eerste kennismaking	11
2.1 Inloggen	11
2.2 Gebruikersinformatie veranderen	12
2.2.1 Paswoord	12
2.2.2 Finger informatie	12
2.3 Informatie over commando's	12
3 Directories en files	14
3.1 Directories	14
3.2 De directory ordening onder Unix	14
3.3 Veranderen van directory	14
3.4 Directory maken	16
3.5 Directory-gegevens bekijken	17
3.6 Randapparaten als files	17
3.7 Pipes	18
4 Enkele simpele operaties	20
4.1 Creëren van files	20
4.2 Bekijken van files	20
4.3 Copy, Move en Link	21
4.4 Directories en files verwijderen	22
4.5 Rechten op files en directories	22
4.6 Archiveren van bestanden	23
4.7 Coderen van bestanden	24
5 Reguliere Expressies	25
5.1 Wat zijn reguliere expressies?	25
5.2 grep en egrep	25
5.3 De mogelijke reguliere expressies met grep	26
5.4 Enkele grep opties	28
5.5 Extended reguliere expressies met egrep	28
5.6 Overzicht van de metakarakters	29

5.7	e-mailadressen ontleden en zoeken	29
6	Editors	31
6.1	Inleiding	31
6.2	Vi/Vim	31
6.2.1	Vi	31
6.2.2	Zoeken en Vervangen in Vim	32
6.3	Emacs	32
6.4	Toetsencombinaties	34
7	Shell scripting	35
7.1	Shell	35
7.2	Aliases	35
7.3	Programmeren in de Shell	36
7.4	Variabelen	36
7.5	Speciale karakters	37
7.6	Vergelijken van expressies	38
7.7	Conditionele expressies	40
7.8	Iteraties	41
7.8.1	for	41
7.8.2	while	42
7.8.3	until	42
7.9	Functies	43
7.10	Scripts onderbreken	43
7.11	eval	44
7.12	Opgave	44
8	Gnu Awk	46
8.1	Principes van AWK	46
8.2	Output	48
8.3	Input	48
8.4	Variabelen	50
8.5	Reguliere expressies	50
8.6	Conditionele expressie	52
8.7	Iteraties	53
8.8	Functies	54
8.9	Voorbeeld	54
9	Programmeren onder Linux	56
9.1	Beschikbare middelen	56
9.2	De GNU C Compiler	56
9.3	Bibliotheken	57
9.3.1	Statische bibliotheken	58
9.3.2	Gedeelde bibliotheken	58
9.3.3	Dynamisch geladen bibliotheken	59
9.3.4	Relevante utilities	60
10	Een device driver in Linux	61
10.1	Randapparaten	61
10.2	Programmeren voor de kernel	62
10.3	Een eerste kernel module	63
10.3.1	Hello world	63

10.3.2	Uitgebreider voorbeeld in het proc filesystem	65
10.4	Devices als files	70
10.4.1	De API	70
10.4.2	Gebruikte entries	70
10.4.3	Module gebruik beschermen	71
10.4.4	I/O control: ioctl	71
10.5	Een device driver voor de keyboard LEDs	71
10.5.1	De opdracht	71
10.5.2	Verduidelijking	71
10.5.3	De code	74
10.5.4	Praktisch gebruik van de device driver	80
10.6	Device Driver Practicum FAQ	80
10.6.1	In de functie <code>write_leds</code> , moet je gebruik maken van <code>copy_from_user</code> . Je moet dan van de buf naar de kernel space schrijven. Is dit dan <code>led_on</code> ?	81
10.6.2	Er staat ook dat <code>ppos</code> moet aangepast worden. Is dit voor de volgende keer dat je de functie gebruikt of moet er een lus gemaakt worden?	81
10.6.3	Om <code>ppos</code> aan te passen volstaat het dan om <code>*ppos++</code> te schrijven of moeten we dat als een pointer beschouwen en iets in de aard van <code>ppos = ppos->next</code> schrijven?	81
10.6.4	Waar kunnen we informatie vinden over de module usage counter?	81
10.6.5	Bij de <code>read_leds</code> : moet je een waarde teruggeven? Hoe controleer je dan of er een foutief resultaat is? Moet dat met de waarde die kar heeft?	82
10.6.6	Moeten we in <code>init_module</code> de functie <code>read_leds</code> en <code>write_leds</code> oproe- pen, of wordt dit gedaan door het systeem?	82
10.6.7	Moeten <code>open_leds</code> en <code>close_leds</code> ook ergens worden aangeroepen?	82
10.6.8	moeten we <code>unregister_chrdev</code> eigenlijk met een waarde vergelijken zoals <code>register_chrdev</code> ?	82
10.7	Verdere informatie	83
11	Een kernel compileren en installeren	84
11.1	De kernel broncode bekomen	84
11.2	De kernel patchen	84
11.3	De kernel configureren	84
11.4	De kernel en bijbehorende modules compileren	84
11.5	De nieuwe kernel installeren	84
12	Hoe software installeren	85
12.1	Red Hat Package Manager	85
12.2	Source distributies installeren	85
13	Linux op het web	87
13.1	Distributies	87
13.2	Kantoor-applicaties	87
13.3	Software ontwikkeling	88
13.4	Meer informatie	88
A	Pointers naar functies	89
B	Voorbeeld van een Makefile	91
C	Oplossingen oefeningen	94
	Bibliografie	97

<i>INHOUDSOPGAVE</i>	4
Over dit document	98
Auteurs en bijdragen	99

Lijst van Figuren

1.1	Ruwe opbouw besturingssysteem Linux	10
2.1	De prompt	11
2.2	Het passwd commando	12
3.1	Linux directory tree	15
10.1	Interne kernel architectuur	62
10.2	Ontstaan van een dangling pointer	64
10.3	Character device switch table	73

Lijst van Tabellen

3.1	Voorbeeld van een Unix directory structuur	15
3.2	Linux directory structuur	16
3.3	belangrijke device files	18
5.1	Reguliere expressies syntax ondersteuning	29
6.1	Veel gebruikte vim commando's	33
6.2	Veel gebruikte Emacs toetsencombinaties	34
7.1	Integer expressies	38
7.2	String expressies	39
7.3	Bestand expressies	39
7.4	Logische expressies	39
8.1	Voorgedefinieerde variabelen	48
8.2	Format specifiers	49
8.3	Vergelijkingsoperatoren	49
8.4	Logische operatoren	49
8.5	Wiskundige operatoren	50
8.6	Metakarakters	51
10.1	Gestructureerde kijk op /dev listing	62

Lijst van Code Listings

5.1	grmail.txt; bestand met emailadressen	29
7.1	de read opdracht	36
7.2	de read opdracht, voorbeeld 2	37
7.3	Ingebouwde variabelen	37
7.4	Toekenning aan variabelen	38
7.5	if statement syntax	40
7.6	if statement voorbeeld	40
7.7	case statement syntax	40
7.8	case statement voorbeeld	40
7.9	for statement syntax	41
7.10	for statement voorbeeld	41
7.11	untarren en gunzippen m.b.v. een for lus	41
7.12	while statement syntax	42
7.13	while statement voorbeeld	42
7.14	until statement syntax	42
7.15	until statement voorbeeld	43
7.16	functie syntax	43
7.17	functie voorbeeld	43
7.18	shift voorbeeld	44
7.19	zonder gebruik te maken van eval	44
7.20	met gebruik te maken van eval	44
7.21	Wat doet dit script?	45
8.1	count.awk	47
8.2	count met awk	47
8.3	Schrijf gebruikersnamen uit met groep ID=20	49
8.4	Tel alle woorden	50
8.5	Schrijf de achternaam uit	50
8.6	factuur.awk	52
	language = bash	53
8.7	Statistieken van een log file	54
9.1	“hello world” voorbeeld programma	57
9.2	dllib.c	60
10.1	Listing van de /dev directory	62
10.2	modhelloworld.c	63
10.3	proctext.c (kernel 2.2)	67
10.4	proctext.c (kernel 2.4)	68
10.5	Het framework voor de LEDs device driver	74
10.6	De eerste code voor de LED device driver	75
10.7	De startcode voor de LED device driver	78
10.8	Aansturen van het device vanuit een C programma	80
A.1	callbackf.c	89

B.1 de Makefile voor dit document	91
B.2 Een simpele Makefile	92
C.1 oplossing oef 2	94
C.2 oplossing oef 3	94
C.3 oplossing oef 4	94
C.4 oplossing oef 5	95
C.5 oplossing oef 6	95
C.6 oplossing oef 7	95
C.7 oplossing oef 8	95

Hoofdstuk 1

Inleiding

Given enough eyeballs, all bugs are shallow.

Eric S. Raymond: Linus's Law

Put on your pointy hat, grow a beard, drink Jolt Cola and come join in the fun.

Alan Cox

Linux is een op UNIX gebaseerd besturingssysteem, gestart door *Linus Torvalds* in 1991. Linus liet zich hierbij inspireren door het besturingssysteem *Minix* [Tan], speciaal ontwikkeld voor didactische doeleinden door Andrew S. Tanenbaum. Omdat Linux gebaseerd is op UNIX, zullen vele aspecten van het Unix besturingssysteem terug te vinden zijn in Linux. De meeste programma's die men kan gebruiken op Unix systemen hebben een Linux equivalent. UNIX is een operating system dat werd ontwikkeld in de jaren 70, en voortdurend verder ontwikkeld en uitgebreid is. Omdat UNIX het eerste besturingssysteem is dat grotendeels werd geschreven in een high-level programmeertaal, namelijk C, kon het op alle systemen, waarvoor een C compiler voorhanden was, werken. Ook Linux is dus beschikbaar voor de meeste hardware.

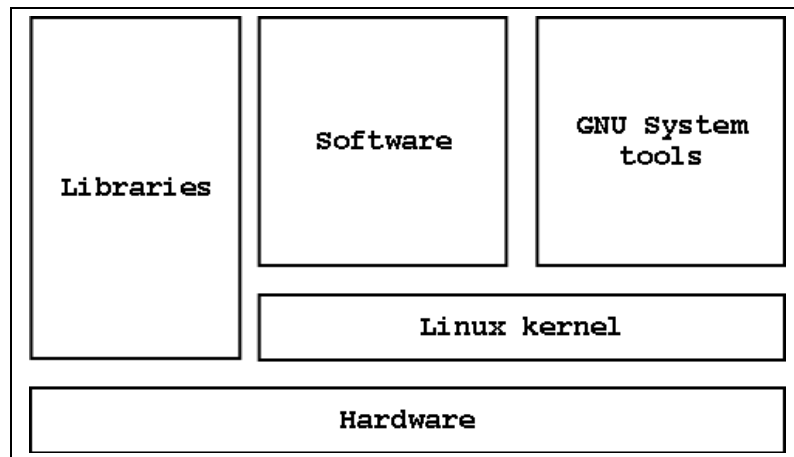
UNIX is een multi-user, multitasking operating system; dit wil zeggen dat meerdere gebruikers op hetzelfde moment kunnen inloggen en dat elk van deze gebruikers meerdere programma's tegelijkertijd kunnen uitvoeren. Ook voor Linux zijn deze eigenschappen geïmplementeerd in het besturingssysteem.

Tegenwoordig bestaan er veel operating systems die zijn afgeleid van UNIX. Naast Linux zijn er ook enkele commerciële voorbeelden: SunOS (nu gekend als *Solaris*) en *SGI Irix*. Tegenwoordig zijn er ook gratis UNIX versies verkrijgbaar onder de zogenaamd GNU General Public License (GNU = GNU's not Unix)¹ die aangeeft dat de source code van deze operating systems verkrijgbaar is en dat iedere gebruiker de code mag aanpassen aan zijn of haar eigen behoeften mits vrijgave van de code. De belangrijkste voorbeelden hiervan zijn Linux en *FreeBSD* (alhoewel FreeBSD een andere licentie hanteert, die ook vrije beschikbaarheid en aanpasbaarheid van de broncode garandeert). Daarnaast verzorgt GNU ook een set van zeer handige system tools die bij elke Linux distributie terug te vinden zijn. Figuur 1.1 bevat een ruwe afbeelding van de gehele structuur, samen met de GNU tools.

GNU is gestart door Richard Stallman, ook weer geïnspireerd op UNIX. In feite kunnen we stellen dat Linux onbruikbaar is zonder GNU tools. Deze tools omvatten onder meer compilers, editors, shells, ... Verschillende van deze tools zullen later nog uitgebreid aan bod komen in deze cursus. Ondertussen is de bibliotheek GNU al uitgegroeid tot honderden toepassingsprogramma's.

Het gebruik van Linux voor een praktische toelichting bij het vak Systemprogrammatuur ligt voor de hand. Mede omdat Linux vrij beschikbare broncode voorziet is het van grote

¹GNU General Public License, een software licentie voor vrij beschikbare herverdeelbare software, zie <http://www.gnu.org>.



Figuur 1.1: Ruwe opbouw besturingssysteem Linux

didactische waarde. We kunnen zien hoe de kernel van het besturingssysteem werkt, en hoe de aangeleerde technieken van besturingssystemen in de praktijk geïmplementeerd kunnen worden. Vooral in verband met het hoofdstuk over device drivers zal dit zeer handig blijken.

Al de hoger vermelde operating systems gebruiken ongeveer dezelfde commando's als UNIX. In deze cursus worden commando's beschreven die overal moeten werken, maar sommige Unices operating systems kunnen kleine afwijkingen vertonen. Om zeker te zijn over de syntax van het systeem waarop je werkt, kun je best de manual pages bekijken (syntax: `man commando`, bv.: `man chfn`. Je kan de man pagina verlaten door op de "q"-toets te drukken).

Hoofdstuk 2

Eerste kennismaking

2.1 Inloggen

De meeste Linux versies hebben een grafische interface (The X Window System¹) die kan gebruikt worden als men op de computer inlogt. Ofwel moet er dan op de computer zelf of op een terminal (bij mainframes) worden ingelogd ofwel kan dit vanuit een pc gebeuren indien de juiste software is geïnstalleerd. De computer waarop Linux draait moet dan ook X Windows ondersteunen, dit doet het door een zogenaamde **X Server**. De X Server is verantwoordelijk voor de grafische output. Een belangrijk voordeel van het gebruik van zo een X Server is dat de output ook naar andere schermen kan gestuurd worden, bijvoorbeeld naar een scherm in een naburige kamer² (of een naburig land!) dat niet rechtstreeks aan de computer zelf verbonden is. Een andere manier om in te loggen is via telnet. Dit kan op bijna ieder platform gebruikt worden om op een server in te loggen. Het enige nadelen zijn dat telnet volledig karaktergeoriënteerd is en zeer onveilig. SSH (Secure Shell) is een veiliger alternatief, indien mogelijk moet dit verkozen worden boven telnet. Eens je (via ssh of telnet) ingelogd bent, krijg je een prompt te zien zoals in figuur 2.1 waarachter je commando's kan typen. In hoofdstuk 4 vind je meer informatie over beschikbare commando's.

```
kris@lumumba:~$
```

Figuur 2.1: De prompt

Voor dit opleidingsonderdeel wordt gebruik gemaakt van de computer `lumumba.luc.ac.be` of van een centrale server, die je toelaat een grafische omgeving op te starten. Om te programmeren voor de Linux kernel zijn er 4 andere PC's beschikbaar waarop je kan inloggen als root. Deze 4 machines zijn niet op het netwerk aangesloten.

Ruwweg kunnen we twee soorten gebruikers op een Linux systeem herkennen: de root en de gewone gebruiker. De root is de beheerder van het systeem, en heeft overal op het systeem toegang tot bestanden, kan deze verwijderen, aanpassen, . . . De gewone gebruiker kan enkel gegevens veranderen in de eigen home directory en soms in enkele andere directories zoals de temp directory (`/tmp`). Zo is het de root die meestal de nieuwe software zal installeren, device drivers kan bijvoegen, . . . De speeltuin van de gewone gebruiker is beperkt tot zijn eigen home directory.

¹<http://www.xfree.org>, <http://www.x.org>

²als er een netwerkverbinding is

```
kris@lumumba:~$ passwd
Changing password for kris
Old password:
Enter the new password (minimum of 5, maximum of 127 characters)
Please use a combination of upper and lower case letters and numbers.
New password:
```

Figuur 2.2: Het passwd commando

2.2 Gebruikersinformatie veranderen

2.2.1 Paswoord

Iedere gebruiker heeft een paswoord om zijn persoonlijke bestanden te beveiligen. Om dit te veranderen moet er `passwd` achter de prompt worden getypt (zie figuur 2.2). De computer zal eerst om het oude paswoord vragen en zal daarna het nieuwe vragen. Dit laatste gebeurt tweemaal, zodat je niet per ongeluk een verkeerd paswoord kan intypen. Vergeet dit paswoord niet, anders kan je niet meer inloggen en moet je de systeembeheerder vragen om een nieuw paswoord.

2.2.2 Finger informatie

Het systeem houdt informatie bij over iedere gebruiker. Andere gebruikers kunnen deze informatie opvragen via het zogenaamde finger protocol (let wel, deze service moet aanwezig zijn op het systeem alvorens men finger kan gebruiken). Om informatie over Kris Luyten op te vragen, typ achter de prompt `finger lucp1315@alpha`³. Hierbij is `lucp1315` zijn login en `alpha` de computer, waar hij zijn login heeft (voluit `alpha.luc.ac.be`). Er verschijnt dan de volgende informatie op het scherm :

```
alpha.luc.ac.be> finger
[alpha.luc.ac.be]
Login name: lucp1315   In real life: Kris Luyten
Directory: /home/lucp1315      Shell: /usr/local/bin/bash
Last login Tue Mar 20 18:57 on tty2 from edm-062.edm.luc.
Plan:
Linux world domination!
```

Hiervoor moet je wel de login naam van de persoon die je zoekt kennen. Om op een stuk van iemands echte naam (In real life) te zoeken, kan je ook het finger commando gebruiken bv. `finger raymaekers@alpha.luc.ac.be`. Om deze mogelijkheid expliciet te vermijden gebruik je de `-m` switch. Op de meeste systemen is de echte naam van een gebruiker voor-naam.achternaam. Indien je wilt dat je gebruikersinformatie wordt aangepast, moet je `chfn` (change full name) intypen. Dit programma laat toe om gebruikersinformatie te veranderen. De informatie die kan worden opgegeven is afhankelijk van het systeem.

2.3 Informatie over commando's

In de volgende hoofdstukken zullen er heel wat commando's ingevoerd worden die je op een Linux systeem kan gebruiken. Voor de meeste commando's kan je een zogenaamde *man*-pagina opvragen. Als je bijvoorbeeld de syntax en de betekenis van het `finger` commando wilt kennen,

³Indien u dit vanop *lumumba* probeert zal dat niet lukken, omdat men vanaf de *lumumba* geen toegang heeft tot de *alpha*

kan je `man finger` intypen, waarna je een overzicht krijgt van onder andere de syntax van dat commando, de betekenis ervan en de mogelijke opties.

Naast het `man` commando om informatie over andere commando's op te vragen, kan je ook `info` gebruiken om wat uitgebreidere informatie te verkrijgen. Tenslotte is er ook `apropos`, om vertrekkende vanaf één of meer keywords op te vragen welke commando's hiermee gerelateerd zijn. Als je bijvoorbeeld een wiskundige functie wil plotten, maar je weet niet of er geschikte⁴ programma's hiervoor op je computer staan kan je het commando `apropos` oproepen met als argument “plot” of “math” of beide. Als dit commando op mijn systeem wordt uitgevoerd dan krijgt je het volgende te zien:

```
[kris@localhost course]$ apropos plot
gnuplot          (1) - an interactive plotting program
mathplot         (1) - interactive (or batch) function grapher
mathplot [mathplot2ps] (1) - interactive (or batch) function grapher
mathplot2ps      (1) - interactive (or batch) function grapher
mathplot2ps [mathplot] (1) - interactive (or batch) function grapher
pbmtoplot        (1) - convert a portable bitmap into a Unix plot(5) file
.Vb 1 PDL::Graphics::PGPLOT [PDL::Graphics::PGPLOT] (3) - PGPLOT enhanced interface for
PDL::Graphics::PGPLOT::Window (3) - A OO interface to PGPLOT windows
PDL::Graphics::PGPLOTOptions (3) - Setting PGPLOT options
```

Opgave Probeer je eigen gebruikersinformatie op te vragen via `finger`. Verander deze informatie met `chfn` en vraag dan nog eens je eigen gebruikersinformatie op.

⁴`apropos` staat trouwens voor “appropriate search”

Hoofdstuk 3

Directories en files

3.1 Directories

Bestanden in Linux zijn hiërarchisch geordend. Dit wil zeggen dat zij in een directory tree staan, die is te vergelijken met de directory structuur van DOS¹. Er zijn een aantal afwijkingen: ten eerste is Linux case-sensitive voor directories en files: Sample.txt is niet hetzelfde als sample.txt. Verder wordt niet het “\” teken om directories te scheiden, maar “/”. De subdirectory C van de directory src is dus src/C. UNIX (en dus ook Linux) systemen kennen ook geen drive aanduidingen zoals in DOS (zoals A: en C: bijvoorbeeld). Alle drives zijn toegankelijk via een subdirectory van de root (/) directory. Op de meeste systemen is de CD-ROM drive bijvoorbeeld toegankelijk via /cdrom/ of /mnt/cdrom/. Hoe dat dit praktisch gebeurt, kan je vinden in sectie 3.6. Om de actieve directory op te vragen kan het commando `pwd` gebruikt worden (`pwd` staat voor “print working directory”).

3.2 De directory ordening onder Unix

Anders dan bij het Microsoft Windows besturingssysteem, waar er slechts enkele “verplichte” directories zijn zoals “Program Files”, “Windows” en “Windows/System” bijvoorbeeld, is de directory structuur op een UNIX (en dus ook Linux) systeem voor een groot deel vooraf vastgelegd. Tabellen tabel 3.1 en tabel 3.2 geven een overzicht van de structuur van respectievelijk een UNIX systeem en een Linux systeem. In figuur 3.1 vind je een grafische voorstelling van de directory structuur, met als wortel van de boom de root directory.

Opgave Ter verduidelijking van de /proc directory (welke u op een Linux systeem kan vinden) kan u de volgende commando’s proberen:

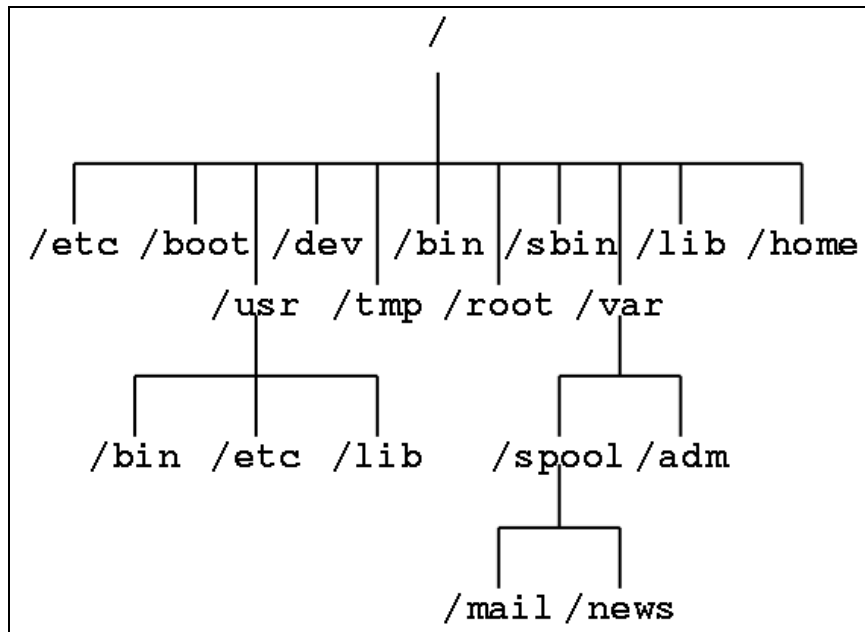
```
cat /proc/meminfo
cat /proc/interrupts
cat /proc/partitions
cat /proc/devices
```

Door middel van het `cat` commando wordt de inhoud van de file naar het scherm weggeschreven.

3.3 Veranderen van directory

Het commando om van directory te veranderen is `cd` (change directory). Enkele voorbeelden zijn:

¹Disk Operating System



Figuur 3.1: Linux directory tree

/	de root directory
/etc	bevat data om het systeem op te starten. Deze directory bevat ook de meeste configuratiebestanden
/lib	bevat functie bibliotheken die gebruikt worden door de C compiler, evenals de gedeelde bibliotheken (zie sectie 9.3 voor meer uitleg over bibliotheken)
/tmp	bevat tijdelijke bestanden
/bin	bevat de meest noodzakelijke en gebruikte binaire en uitvoerbare bestanden
/usr	bevat al de rest
/usr/spool	bevat data die meestal in een queue staat, wachtend om verwerkt te worden, bijvoorbeeld printjobs
/usr/bin	bevat binaire en uitvoerbare bestanden zoals /bin
/usr/include	bevat de source code die gebruikt wordt door <code>#include</code> statements
/usr/adm	bevat diagnostische informatie en de boekhouding van het systeem
/usr/lib	bevat diverse bibliotheken die door andere programma's aangeroepen worden

Tabel 3.1: Voorbeeld van een Unix directory structuur

/	de root directory
/etc	bevat data om het systeem op te starten. Deze directory bevat ook de meeste configuratiebestanden
/etc/passwd	bevat de gebruikers database (is <i>geen</i> directory, wel een bestand)
/etc/rc.d	bevat systeem-initialisatiescripts
/lib	bevat functiebibliotheken die gebruikt worden door de C compiler, evenals de gedeelde bibliotheken (zie sectie 9.3 voor meer uitleg over bibliotheken)
/tmp	bevat tijdelijke bestanden
/var	bevat systeem definitie tabellen
/home	bevat de accounts van de gebruikers
/home/<zet uw naam hier>	dit is de user account van de gebruiker <zet uw naam hier>
/bin	bevat de meest noodzakelijke en gebruikte binaire en uitvoerbare bestanden
/sbin	bevat alle systeem-programma's
/usr	bevat al de rest niet aanwezig in de andere directories
/usr/bin	bevat binaire en uitvoerbare bestanden zoals /bin
/proc	bevat een pseudo-filesysteem dat dient als een interface naar kernel data structuren. Het wordt ondermeer gebruikt om proces informatie uit het geheugen te lezen.
/dev	bevat bestanden die de hardware voorstellen van het systeem

Tabel 3.2: Linux directory structuur

<code>cd ..</code>	Ga naar de onmiddellijk hogere directory in de directorytree (“parent directory”);
<code>cd /</code>	Ga naar de hoogste directory (root);
<code>cd /home/craymaek</code>	Ga naar de home directory van de gebruiker <code>craymaek</code> ;
<code>cd ~</code>	Ga naar je eigen home directory;
<code>cd dir</code>	Ga naar de directory <code>dir</code> die onder de huidige directory staat;

Directory specificaties zijn niet steeds relatief ten opzichte van de huidige directory of absoluut (ten opzichte van de root directory).

Opgave Probeer zelf enkele directories te activeren. U kan de huidige directory opvragen door het commando `pwd`.

3.4 Directory maken

Je kunt pas naar een directory gaan als deze directory bestaat natuurlijk. Om een directory te maken, moet je het commando `mkdir` (make directory) gebruiken met als parameter de directory die je wilt maken. Je kan natuurlijk alleen maar op de plaatsen waar je hiervoor toestemming hebt nieuwe directories aanmaken. Voor een gewone gebruiker is dat meestal alleen in de eigen home directory (`~/`) of de temp directory (`/tmp/`), voor de beheerder van het systeem (root of *super user*) is dat nagenoeg overal.

Opgave Maak een directory oefeningen aan onder je home directory. Deze directory gaat nog verder worden gebruikt tijdens de oefeningen.

3.5 Directory-gegevens bekijken

Met het commando `ls` (list) kan de inhoud van een directory worden opgevraagd. Enkele voorbeelden zijn:

```
ls                Geeft de inhoud van de huidige directory;
ls oefeningen    Geeft de inhoud van de oefeningen directory;
ls h*            Geeft de lijst van alle files waarvan de naam met h begint;
ls ?allo        Geeft de lijst van alle files waarvan de naam met
                een willekeurig karakter begint en eindigt op "allo";
```

Vergeet niet dat ook hier directories relatief of absoluut kunnen zijn. Zoals je in de vorige voorbeelden kan zien is het ook mogelijk *joker*-tekens te gebruiken die door de shell ge-expand worden. Als je het teken `*` gebruikt wil dit zeggen dat het met 0 of meer achtereenvolgende willekeurige tekens mag matchen. Het teken `?` wil zeggen dat er met 0 of 1 willekeurig teken gematcht mag worden. Zo matcht `ch*ter1*` bijvoorbeeld met `chapter1`, `chputer1`, `chter105689`,... en `ch?ter1?` met `chater12`, `chter1`, `choter1b`,...

Het `ls` commando heeft een reeks van parameters (switches). Enkele belangrijke zijn

```
-a                Geeft alle files weer, ook de verborgen "." files;
-l               Geeft de directory meer gedetailleerd weer;
-R               Gaat ook alle subdirectories af;
```

Een bestand met een naam beginnende met een punt ".", is een speciaal bestand, meestal een configuratie bestand of iets dergelijks. Een gewone listing toont deze bestanden niet, we moeten hiervoor expliciet opgeven dat we alle bestanden willen zien met de optie `-a`.

Deze switches kunnen ook worden gecombineerd tot bv. `ls -la`. Indien de directory te groot is, zodat je niet alles in een keer op het scherm kan worden getoond, kun je | `more` achter het `ls` commando types. Vb: `ls -la /home | more`. Dit geeft de directory listing pagina per pagina weer. Om naar de volgende pagina te gaan, moet je op de spatiebalk drukken. Bij sommige systemen moet je na de laatste pagina op "q" drukken (quit). Bijna alle commando's die uitvoer op het scherm geven kunnen met | `more` worden gecombineerd. Meer informatie over het "|" symbool vind u in sectie 3.7. Een alternatief voor `more` is `less`. Hiermee kan men zowel voorwaarts als achterwaarts scrollen, waardoor `less` een meer ingewikkeld commando is dan `more`. Na iedere pagina moet een commando worden gegeven om het vervolg te zien. De belangrijkste zijn `f` (forward, soms ook de PageDown toets), `b` (backward, soms ook de PageUp toets) en `q` (quit). Meer informatie over `less` staat in de man pages.

3.6 Randapparaten als files

Voordat men de directories kan gebruiken om de randapparaten aan te spreken, moet men eerst het geviseerde "device" *mounten*. Dit wil zeggen dat men een directory en het apparaat logisch zal verbinden. Dit kan men bijvoorbeeld doen als volgt: `mount -t iso9660 /dev/hdc /mnt/cdrom`. *iso9660* duidt op het soort bestandssysteem dat men mount (andere mogelijkheden zijn *vfat* voor fat bestandssystemen en *extf2* voor Linux extended filesystem 2 bestandssystemen). */dev/hdc* is een voorbeeld van een file die de connectie legt met de eigenlijke hardware, welke file voor welk randapparaat staat is afhankelijk van de hardware configuratie van je computer. Het hoeft bijvoorbeeld niet */dev/hdc* te zijn voor je cdrom, op een andere PC had dat ook */dev/hdb* kunnen zijn. Meestal kan je een floppy zelfs mounten zonder het type van

/dev/console	de monitor die aan de computer hangt
/dev/hd	de IDE randapparaten (harde schijven, cdrom), zo geeft /dev/hda2 de tweede partitie weer op de schijf hda
/dev/sd	de SCSI randapparaten
/dev/fd	de floppy drive
/dev/null	dit is de “vuilbak” van het systeem; alle data die men ernaar toe schrijft is voor altijd verloren. Indien het gebruikt wordt als input file wordt er een file gecreëerd van lengte 0

Tabel 3.3: belangrijke device files

bestandssysteem eraan mee te geven: `mount /dev/fd0 /mnt/floppy` bijvoorbeeld. `/dev/fd0` stelt een bestand voor dat een abstractie van de hardware is (meer bepaald de floppy disk), en we willen deze mounten op de directory `/mnt/floppy`. Let op: de directory `/mnt/floppy` moet bestaan om dit commando uit te voeren.

In UNIX en Linux worden randapparaten en andere hardware dus als bestanden voorgesteld. Deze speciale files kan men vinden in de directory `/dev`. Om output naar deze randapparaten te sturen kan men dan simpelweg naar deze files schrijven. Vb.: `cat /usr/share/sounds/generic.wav > /dev/dsp` zorgt ervoor dat de data in de wave file naar de geluidskaart gestuurd wordt. We gebruiken hier weer de redirectie operator `>` voor. Merk op dat het bestand waarnaar geschreven wordt in rechtstreeks contact staat met de eigenlijke hardware. Anderzijds kan men van deze files ook informatie lezen. Zo geeft de file `/dev/mouse`² gegevens over de muis. Tabel 3.3 geeft een overzicht van de belangrijkste device files weer.

3.7 Pipes

In het voorbeeldje van het gebruik van het `more` commando zagen we het symbool `|`. Dit wordt een pipe (pijp) genoemd. In Linux wordt het pipe symbool `()` gebruikt om twee of meerdere commando's te combineren. Hierbij dient het resultaat van een commando als invoer voor het volgende commando. De betekenis van dit symbool is makkelijk uit te leggen aan de hand van het voorbeeldje `ls -l /home | less`. het commando voor de pipe, namelijk `ls -l /home` genereert veel output, meer dan in één keer op het scherm kan. Het commando na de pipe, namelijk `less`, zorgt ervoor dat als men het input geeft langer dan het scherm dit deel per deel kan bekeken worden op het scherm. Nu moeten we een manier hebben om de output van `ls -l /home` als input aan `less` te geven. Hiervoor zorgt de pipe: `|`. Het neemt de output van de commando die voor het symbool `|` staat en voedt het als input aan het commando dat achter het symbool `|` staat.

Een ander voorbeeldje van het gebruik van pipes is `ls /home | sort | less`. In het eerste deel, voor de eerste pipe wordt een listing gegeven van al de bestanden en subdirectories in `/home`. Deze listing wordt als input gegeven aan het commando `sort`. Dit commando sorteert de lijnen input die het krijgt volgens de alfabetische volgorde. Als output van het commando `sort` kan je dus een gesorteerde lijst regels verwachten. Deze output wordt door de tweede pipe doorgegeven aan het laatste commando, nl. `less`. Dit zorgt ervoor dat de gebruiker al de output kan doornemen op het scherm.

Het commando `grep` wordt ook regelmatig gebruikt in combinatie met andere commando's. `grep` drukt lijnen af die aan een meegegeven patroon voldoen. Hierbij zoekt `grep` in de files die het aangeduid krijgt via de argumenten of de standaard input als er geen files aangeduid zijn. Als we bijvoorbeeld alle zinnen uit de file `/etc/rc.d/rc.local` waarin het woord “echo”

²Dit is meestal een link naar een ander file!

voorkomt willen zien, dan kan dit door het commando: `grep echo /etc/rc.d/rc.local`. Als eerste argument geeft men het patroon mee waarnaar men zoekt, en als tweede argument de plaats waar `grep` dit moet gaan zoeken (zie hoofdstuk 5 voor meer uitleg). Nu kunnen we bijvoorbeeld ook via een pipe een lijst tekstlijnen doorgeven en `grep` gebruiken om de ongewenste lijnen weg te filteren. Zo kunnen we bijvoorbeeld alle bestanden die als bestandsnaam het patroon “oef” bevatten (in de huidige directory) opvragen door het commando `ls | grep oef`.

Hoofdstuk 4

Enkele simpele operaties

4.1 Creëren van files

Er is een simpel commando om een *lege* file te creëren in Linux, namelijk `touch`. `touch mijnbestand` maakt een leeg bestand met de naam `mijnbestand` aan indien dit bestand nog *niet* bestond. Anders verandert het gewoon de “modification time” van het bestand. Let op: een bestandsnaam mag geen karakters bevatten die een speciale betekenis hebben voor de shell. Concreet wil dit zeggen dat de volgende tekens niet mogen gebruikt worden in een bestandsnaam:

```
! @ # $ % ^ & * ( ) [ ] { } ' " \ / | ; < > `
```

Beschouw het volgende voorbeeld: Bij het `finger` commando van paragraaf 2.2.2 kreeg je op de laatste regel “No plan” te zien. Op deze plaats kun je extra informatie geven en hiervoor moet er een tekst file met de naam `.plan` worden aangemaakt (vergeet de “.” niet, dit geeft aan dat dit een speciale file is). De tekst in deze file zal afgebeeld worden op de plaats waar nu “No plan” staat. Je kan het bestand, w iliswaar zonder enige inhoud creëren door middel van `touch ~/.plan` De eenvoudigste manier om een file met tekst erin aan te maken is natuurlijk gebruik te maken van een tekst editor. De meeste UNIX systemen hebben drie karaktergeoriënteerde tekst editors: *ed*, *vi* en *pico*. Hiervan is *pico* (onderdeel van het email en news programma *pine*) het eenvoudigste om te gebruiken. Een andere veel gebruikte editor is *emacs*. Onder software ontwikkelaars zijn *emacs* en *vi* zeer populaire editors. Niet zozeer vanwege hun gebruiksvriendelijkheid (de leercurve mag wel steil genoemd worden) maar wel vanwege de efficiëntie die ermee kan bereikt worden. Een korte introductie tot *vim* kan je vinden in hoofdstuk 6.

Opgave Maak een `.plan` file aan door `pico .plan` te typen. Schrijf een paar zinnen met behulp van de editor *pico* en sluit af met `ctrl+x`. Probeer nu je eigen gebruikersinformatie op te vragen met `finger`.

4.2 Bekijken van files

Een tekst file kan met een tekst editor worden bekeken. Maar indien je enkel een file wil bekijken is een tekst editor hiervoor niet altijd het beste hulpmiddel. Belangrijke files¹ kunnen beter niet worden geëditeerd, omdat je het gevaar loopt om niet meer te kunnen inloggen als je per ongeluk een verkeerde waarde geeft. Het kan ook zijn dat je wel het recht hebt om een file te bekijken maar niet om deze file te wijzigen. Hiervoor is het `cat` commando zeer handig

¹Onder “belangrijke files” verstaan we configuratie bestanden zoals die bijvoorbeeld in de `/etc` directory voorkomen. Zonder een goede kennis van hun inhoud kan men deze beter niet zomaar veranderen.

om een file te bekijken. De syntax is `cat filename`. Ook `more` kan gebruikt worden samen met `cat` door middel van een pipe. Hierbij kan `cat filename | more` op de meeste systemen worden afgekort tot `more filename`. Dit is omdat `more` ook op files kan werken, welke dan per pagina op de standaard output (het scherm) te zien zijn.

`cat` wordt vaak gecombineerd met `head` of `tail`. Hiermee kunnen respectievelijk het begin en het einde van een file worden bekeken. Vb.:

<code>cat readme head</code>	Toont de 10 eerste regels van de file <code>readme</code> ;
<code>cat readme head -n 5</code>	Toont de 5 eerste regels van de file <code>readme</code> ;
<code>cat log tail</code>	Toont de 10 laatste regels van de file <code>log</code> ;
<code>cat log tail -n 5</code>	Toont de 5 laatste regels van de file <code>log</code> ;

`Cat` kan ook gebruikt worden om twee of meerdere files samen te voegen (concatenate), vb.: `cat file1 file2 > file3`. Dit wil zeggen dat de inhoud van `file1` en `file2` achtereenvolgens in `file3` moet worden geschreven. “>” is de redirectie operator en zegt dat de output niet naar de standaard output (dit is meestal de console) geschreven moet worden, maar naar het vermelde bestand (`file3` in dit geval).

Opgave Bekijk de file waarin jouw history staat (`.sh_history`) met behulp van het `cat` commando. Indien je de `bash` gebruikt, moet je de file `.bash_history` gebruiken.

4.3 Copy, Move en Link

Een operating system is niet compleet als files niet kunnen worden gekopieerd of verplaatst. Het commando `cp` (copy) kopieert een file of directory. Om `file1` naar `file2` te kopiëren moet het volgende commando worden gegeven: `cp file1 file2`. De volledige inhoud van een directory kan worden gekopieerd door gebruik te maken van de wildcard “*”. Het commando `cp /etc/* ./` zal de volledige inhoud van de `/etc` directory naar de huidige directory kopiëren (doe dit niet: de `/etc` directory is zeer uitgebreid!!!). Door de `-r` switch te gebruiken kan een complete directory (inclusief de subdirectories) worden gekopieerd, vb: `cp -r dir1 dir2`, dat `dir2` aanmaakt als kopie van `dir1`. De switch `-r` geeft aan dat het commando `cp` hier recursief moet werken.

Het verplaatsen van files en directories is analoog aan het kopiëren ervan. Hiervoor moet gebruik worden gemaakt van het commando `mv` (move). Met dit commando kan ook een file worden hernoemd, vb.: `mv .pan .plan`.

In UNIX omgevingen kan men een file laten wijzen naar een andere file, deze file wordt een link genoemd. De meest gebruikte links zijn symbolische links. Als men een symbolische link wil editeren, wordt de oorspronkelijke file geëditeerd als die bestaat. Het commando om een symbolische link te maken is `ln -s file1 file2`.

Opgave

- Kopieer de file `.plan` in je home directory naar een file genaamd `plan` in de oefeningen directory. Kopieer de hele inhoud van de oefeningen directory naar een directory genaamd `backup`.
- Maak in de oefeningen directory een symbolische link naar `.plan`, genaamd `planlink`. Bekijk de inhoud van de oefeningen directory met `ls -l`. Is er een verschil tussen de twee files?
- Editeer nu de file `planlink` en bekijk je eigen gebruikersinformatie met `finger`. Is er nu iets veranderd?

4.4 Directories en files verwijderen

Files kunnen worden verwijderd met het commando `rm` (remove), waarvan de syntaxis `rm files` is. Ook hier kunnen wildcards worden gebruikt. Om een (lege!) directory te verwijderen, moet het commando `rmdir` (remove directory) worden gebruikt. Het verwijderen van alle files en subdirectories van een directory en het verwijderen van een directory zelf kan worden gecombineerd door de `-r` switch: `rm -r dirname`. Zoals bij het copy commando geeft de `-r` switch hier ook aan dat `rm` recursief te werk moet gaan. Als je er zeker van bent dat je de bestanden wilt verwijderen kan je de optie `f` gebruiken. Door `rm -f naam` in te typen zal het `rm` commando dit bestand verwijderen zonder nog eens expliciet toestemming hiervoor te vragen aan de gebruiker.

Opgave Verwijder de backup directory.

4.5 Rechten op files en directories

In UNIX omgevingen behoort iedere gebruiker tot een *groep*. Aan de hand hiervan kan de beveiliging van het systeem worden opgezet. Iedere file in een UNIX systeem heeft namelijk een owner (user), een group en een beveiligingsmodus van de volgende vorm `TUUUGGG000`. Dit kan men in 4 delen opsplitsen

T	type van de file, vb.: directory (d), gewone file (-);
UUU	owner read (r), write (w) en execute (x);
GGG	group read (r), write (w) en execute (x);
000	others read (r), write (w) en execute (x);

De file `public_html` in een home directory kan de volgende beveiligingsmodus hebben: `drwxr-xr-x`. Dit wil zeggen dat deze file een directory is (d) en dat de owner deze file mag lezen (ls), schrijven (files creëren en verwijderen) en uitvoeren (cd). Anderen (de group en alle andere gebruikers van het systeem) mogen de file lezen en uitvoeren, maar niet schrijven. Alleen de owner van een file (of directory) en de root gebruiker hebben het recht om deze informatie te veranderen. De owner van de directory bepaalt wie in de directory bestanden mag verwijderen of aanmaken.

Het belangrijkste commando voor de beveiliging verandert de beveiligingsmodus zelf: `chmod` (change modus). Dit commando kan op twee manieren worden gebruikt: `chmod [ugoa] {+,-,=} [rwx] files` of `chmod mode files`. De eerste manier zorgt ervoor dat voor de user (u), group (g), others (o) of iedereen (a, all) een bepaalde permissie (Read, Write, eXecute) wordt toegevoegd (+) of afgenomen (-). Ook kunnen de permissies aan een bepaalde permissie worden gelijkgesteld (=). Om bijvoorbeeld andere gebruikers van dezelfde group, waar je zelf inzit, permissie te geven om de `.plan` file te lezen moet `chmod g+r .plan` worden uitgevoerd.

De tweede manier bekijkt de permissie op een binaire wijze. De permissie `rw- r-- r--` kan binair worden gezien als `110 100 100`. Octaal is dit `644`. Om nu de `.plan` file deze permissie te geven, kan het volgende worden ingetypt: `chmod 644 .plan`.

Om de owner van een file of directory te veranderen, moet het commando `chown` (change owner) worden gebruikt. Dit commando mag alleen maar worden gebruikt als je 100% zeker bent van wat je doet. De syntax is `chown user files`. Alleen de nieuwe eigenaar van de file of directory en de systeembeheerder (super user) kunnen je opnieuw de eigenaar maken van deze files. Meestal kan enkel de superuser `chown` toepassen gebruiken.

De group kan worden veranderd met het commando `chgrp` (change group): `chgrp group files`.

Opgave Zorg ervoor dat alle gebruikers de directory listing van de oefeningen directory kunnen vragen en dat de gebruikers van de groep ook naar deze directory kunnen gaan. De owner moet zijn of haar schrijfpermissies behouden.

4.6 Archiveren van bestanden

Wanneer met grote projecten wordt gewerkt is het handig om de files van dit project te archiveren. Hierbij worden de files samengevoegd tot één file. Het commando dat dit doet is `tar` (tape archiver). Historisch gezien wordt `tar` gebruikt om back-ups te maken op tape, maar het resultaat van `tar` kan eender welke file zijn. De syntaxis is: `tar [key] [file...]`. `key` kan de volgende waarden hebben:

<code>r</code>	archiveert files;
<code>x</code>	extract files;
<code>t</code>	geeft een lijst van de files in de archive;
<code>c</code>	creëert een nieuw archive (impliceert “r”);
<code>v</code>	verbose: geeft op de console weer wat er gebeurt;
<code>f</code>	de volgende parameter is de naam van de archive;

De `public.html` directory kan gearchiveerd worden met het commando `tar -cf pages.tar public.html`. De switch `c` geeft aan dat er een tar-bestand wordt gecreëerd, `f` geeft aan dat we dit naar een bestand willen doen. Omdat de switch `f` aangeeft dat we naar een bestand gaan archiveren is het eerste argument dat verwacht wordt de bestandsnaam voor het resultaat, hier is dat `pages.tar`. Daarna verwacht `tar` al de directories en bestandsnamen die gearchiveerd moeten worden, hier is dat enkel de directory `public.html`. De directorystructuur wordt op een relatieve manier ook mee gearchiveerd. De gemaakte file is niet gecomprimeerd: de files zijn gewoon in de archive geplaatst. Om een file te comprimeren kan `gzip` worden gebruikt. Er zijn meerdere GNU compressie programma’s beschikbaar, zo is `bzip2` een goed alternatief. Hiermee kan eender welke file worden gecomprimeerd. Het commando `gzip pages.tar` verwijdert de file `pages.tar` en creëert de gecomprimeerde file `pages.tar.gz`. Het commando `gunzip` decomprimeert het bestand weer. Deze commando’s kunnen natuurlijk ook gecombineerd worden. Om de `public.html` directory te archiveren en te comprimeren kan men het volgende intypen: `tar -c ./public.html/ | gzip > pubhtml.tgz`². Door het `tar` commando worden al de files in de directory samengezet in een enkele file. De output van het `tar` commando wordt rechtstreeks als input van het `gzip` commando doorgegeven via de pipe. Meer uitleg over het gebruik van een pipe (`|`) vind je in sectie 3.7. `gzip` wijst door `>` te gebruiken `pubhtml.tgz` als output file aan (dit wordt ook wel “redirectie naar een bestand” genoemd). Hierbij moet nog opgemerkt worden dat `gzip` slechts op individuele files werkt. Als men geprobeerd had `gzip ./public.html/` in te typen zouden alle files in de directory in aparte files met `.gz` als extensie gecomprimeerd worden. Om de file `pubhtml.tgz` nu terug uit te pakken kan men `tar -xvzf pubhtml.tgz`³ intypen (waarbij `x` aangeeft dat het om extractie gaat, `v` aangeeft dat er informatie over de operatie moet gegeven worden⁴, `z` aangeeft dat de file eerst gedecomprimeerd moet worden met `gunzip` en `f` aangeeft dat het om een file gaat).

² `tar` heeft ook een optie om de compressie met `gzip` dadelijk toe te passen, namelijk de switch `z`, bijvoorbeeld `tar -czf pubhtml.tgz ./public.html/`

³We kunnen dit ook in 2 stappen doen: eerst `gunzip` gebruiken om te unzippen en daarna pas untarren.

⁴de optie `v` is beschikbaar voor de meeste commando’s om informatie te geven over hetgeen het commando aan het doen is

4.7 Coderen van bestanden

Moderne e-mail programma's zoals outlook en netscape hebben ondersteuning voor MIME⁵. Dit maakt het verzenden van binaire files zeer gemakkelijk. E-mail programma's, zoals *pine* en *elm* die standaard op UNIX systemen staan kunnen slechts e-mails aan die gebruik maken van de eerste 128 ASCII karakters. Om via zo een e-mail programma een binaire file te sturen of om een binaire file te sturen naar iemand die alleen maar zo een e-mail programma heeft, moet de binaire file op een speciale manier worden gecodeerd. Het programma dat een binaire file encodeert, heet `uuencode`. Het programma dat een gecodeerde file terug omzet naar een binaire file heet `uudecode`. Om een file te encoderen moet het volgende commando worden ingetypt: `uuencode pages.tar.gz pages.tar.gz`. De eerste parameter geeft aan wat de naam van de file is die moet worden geëncodeerd. De tweede parameter geeft de naam aan die de file moet krijgen bij het decoderen. Het is vaak verstandig om twee maal dezelfde parameter te gebruiken. Er is nu nog één probleem: het resultaat wordt naar het scherm geschreven in plaats van naar een file. Dit kan worden opgelost door redirectie te gebruiken: `: uuencode pages.tar.gz pages.tar.gz > pages.uue`. Dit wil zeggen dat het resultaat niet naar het scherm moet worden geschreven, maar naar de file met naam `pages.uue`. UNIX systemen behandelen het scherm namelijk op dezelfde manier als een file. Vaak wordt `uue` als extensie genomen om aan te geven dat de file is geëncodeerd met `uuencode`. De file kan nu in een tekst editor worden ingelezen. Het decoderen gebeurt met het commando `uudecode pages.uue`. Het maakt niet uit als er in de file nog andere informatie staat, zodat een uuencoded file gemakkelijk in een e-mail kan worden gezet.

Opgave Maak wat lege en niet lege bestanden aan in de oefeningen directory. Maak een archive van de oefeningen directory en comprimeer het resultaat. Encodeer de gecomprimeerde file. Verwijder de oefeningen directory en de gecomprimeerde file. Voer nu `uudecode`, `gunzip` en `tar` uit. De oefeningen directory zou nu terug in zijn oorspronkelijke staat moeten zijn hersteld.

⁵Multipurpose Internet Media Extensions

Hoofdstuk 5

Reguliere Expressies

5.1 Wat zijn reguliere expressies?

Een reguliere expressie is een patroon, en dit patroon beschijft een taal. In dit specifieke geval beschrijft het patroon een verzameling van strings. Naar analogie met arithmetische expressies is het mogelijk kleinere reguliere expressies te combineren tot een grote reguliere expressie met behulp van operatoren. Het alfabet dat hier gebruikt wordt is de verzameling van letters en digits. Zo zal een lijst van karakters omringd door [en] elke karakter tussen deze haken matchen. Bijvoorbeeld: `[af05]` matcht de letters *a*, *f* en de digits *0* en *5*. We kunnen ook een bereik gebruiken, bijvoorbeeld `[d-i]` matcht met de letters *d*, *e*, *f*, *g*, *h* en *i*. De verzameling karakters tussen [en] wordt ook wel een *character class* genoemd.

In dit hoofdstuk laten we zien hoe reguliere expressies in vele GNU applicaties kunnen gebruikt worden. Als we bijvoorbeeld naar een bepaald patroon zoeken in een tekst, en alle regels willen weergeven waarin dat patroon voorkomt moeten we reguliere expressies gebruiken. Deze expressies worden gebruikt om te kijken of een bepaalde sequentie beantwoordt aan het opgegeven patroon: dit wordt ook wel *pattern matching* genoemd. We laten aan de hand van `grep` en `egrep`¹ zien hoe we pattern matching met behulp van reguliere expressie kunnen gebruiken. `grep` staat trouwens voor *General Regular Expression Parser*. Andere applicaties die regelmatig gebruik maken van reguliere expressies zijn de editors `ed`, `sed`, `vim` en `emacs` (om zoekopdrachten in een tekst te doen), de scripting talen `awk` en `perl` en de utility `find`. Een kleine waarschuwing hier: we zullen hier enkel `grep` en `egrep` ter illustratie gebruiken, bij andere applicaties kan de syntax voor het specificeren van een reguliere expressies verschillen. Raadpleeg de gebruikershandleiding en de man-pagina's om de juiste syntax te weten te komen. Zie ook [[Man99](#), [Hek97](#)] voor meer uitleg.

5.2 grep en egrep

`grep` en `egrep` worden gebruikt om regels uit te printen waarin een bepaald opgegeven patroon voorkomt. De syntax wordt als volgt gedefiniëerd:

```
grep [options] <PATROON> [BESTAND(EN)...]
```

`egrep` is hetzelfde als `grep -E`. De optie `-E` zorgt er voor dat het opgegeven patroon als een “extended” reguliere expressie opgevat. Dit betekent dat men krachtigere reguliere expressies kan gebruiken, we zullen later hier nog op terug komen.

Ter illustratie, stel dat we alle lijnen die het woord “Linux” bevatten in de (L^AT_EX bestanden van de) hoofdstukken “Programmeren onder Linux” en “Een device driver in Linux” willen zoeken. We zouden dan het volgende commando intypen: `grep 'Linux' shellprogr.tex`

¹extended *grep*

devicedr.tex. Het resultaat hiervan is:

```
devicedr.tex:\chapter{Een device driver in Linux}
devicedr.tex:over de Linux kernel is te vinden op \url{http://www.kernelnotes.org/}.
devicedr.tex:Al de devices waarvan je Linux systeem gebruik kan maken staan opgesomd
in de /dev directory, zoals reeds vermeld
devicedr.tex:Er is een duidelijk verschil in aanpak wanneer we voor de kernel moeten
programmeren. Het besturingssysteem Linux kent
devicedr.tex:we er in Linux mee rekening houden dat lezen van en schrijven naar
een randapparaat dezelfde
linprogr.tex:\chapter {Programmeren onder Linux}
linprogr.tex:Op zowat elk Linux systeem is er een \href{http://gcc.gnu.org/}{GNU
C compiler} aanwezig,
linprogr.tex:omdat deze de meest gebruikte compilers zijn op Linux systemen.
linprogr.tex:%voor het Linux platform. Daarnaast zijn er (net als op het Microsoft
platform)
```

Telkens wordt eerst de naam van het bestand afgebeeld, en daarna de regel waarin het woord “Linux” voorkomt. Men kan ook het lijn nummer mee laten afbeelden door de optie `-n` mee te geven aan `grep`. Voor meer opties, raadpleeg de man-pagina’s.

Opmerking: het is altijd veiliger om quotes rond het patroon te zetten, anders wou de shell wel eens het patroon kunnen expanden.

5.3 De mogelijke reguliere expressies met grep

Naast de reguliere expressies die op het begin van dit hoofdstuk reeds getoond worden, biedt `grep` nog een heel wat andere mogelijkheden om zulke expressies te vormen. We zetten deze hier op een rijtje met telkens enkele voorbeelden. Als voorbeeld bestand gebruiken we de Makefile die u vindt in appendix B.

[]: de character class; matcht een karakter uit een verzameling van karakters tussen [en], zoals reeds getoond in de inleiding van dit hoofdstuk (zie sectie 5.1). Binnen in [en] kan de verzameling karakters voorafgegaan worden door `^`. Dit betekent dan dat deze karakters *niet* mogen gematcht worden. Zo zal `b[^A-Da-d]` ervoor zorgen dat de letters a, b, c, d, A, B, C en D niet mogen voorkomen als letter tussen *b* en *l*. De volgende woorden zouden dus mogelijk wel matchen: *bol, bOl, b5l, bil* en *bwl*.

.: matcht eender welk karakter;

commando	grep '[iI]...[xX]' Makefile
output	<pre>BIB = bibtex # for generating the BibTeX entries INDEX = makeindex # for generating the index \$(INDEX) \$(FILE) \$(INDEX) \$(FILE)</pre>

Dit voorbeeld matcht elk woord dat begin met *i* of *I*, gevolgd door *juist* 3 karakters en afgesloten door *x* of *X*. Zo matcht `k.st` ondermeer de woorden *kast, kbst, kost, k3st,...*

: het voorgaande karakter zal nul, één of meer keer herhaald worden. Let op hierbij: als je `` gebruikt in de shell zal dit ge-expand worden naar alle mogelijke opeenvolgingen van karakters. Bij `grep` is dit anders; zo matcht `hoo*fd` de woorden *hofd, hoofd, hooofd, hooofd, hooofd, hooofd,...*

commando	grep 'hoo*1' Makefile
output	# de hoofdfile van de tekst # de hoofdregel van deze Makefile
commando	grep 'ho*1' Makefile
output	
commando	grep 'ho*1*' Makefile
output	BIB = bibtex # for generating the BibTeX entries INDEX = makeindex # for generating the index # de hoofdfile van de tekst # de hoofdregel van deze Makefile linprogr.tex shellprogr.tex softinstall.tex\ app-callback.tex app-makefile.tex bibliotheken.tex\ oploefshell.tex editors.tex regexpr.tex # ruim de boel op, alles behalve de benodigde sources

\wedge : matcht het begin van een lijn

commando	grep '^#' Makefile
output	# De compilers: # de hoofdfile van de tekst # de flags voor de DVI compiler en de output file # welke bestanden mogen weg gedaan worden # de hoofdregel van deze Makefile # ruim de boel op, alles behalve de benodigde sources

Dit matcht dus alle regels die met '#' beginnen.

commando	grep '^[BP][DI]' Makefile
output	PDF = pdflatex # for a pdf file BIB = bibtex # for generating the BibTeX entries

Opmer-

king: let op het verschil met het gebruik van \wedge binnen [en], daar wordt het gebruikt als negatie.

$\$$: matcht het einde van een lijn

commando	grep 'file\$' Makefile
output	DVI = latex # for a DVI file PS = dvips # for a postscript file PDF = pdflatex # for a pdf file # de flags voor de DVI compiler en de output file # de hoofdregel van deze Makefile

Dit matcht alle regels die eindigen met het woord "file".

\backslash : zorgt ervoor dat we ook de gereserveerde karakters kunnen gebruiken zoals +, ?, |, (,) en

commando	<code>grep '\\$(DELETABLE)' Makefile</code>
output	<pre>rm -rfv \$(DELETABLE) #verwijder intermediate files rm -rfv \$(DELETABLE) #verwijder intermediate files rm -rfv \$(CLEANABLE) \$(DELETABLE)</pre>

`\{ \}` : het voorgaande karakter van `\{n\}` wordt juist n keer gematcht. Bijvoorbeeld: de expressie `0[123456789]\{2\}/[0123456789]\{6\}` matcht een telefoonnummer van de vorm `044/124578`.

Opmerking: deze metakarakters werken *niet* bij `egrep`.

Een veel voorkomende patroon om te matchen is gewoon de willekeurige string, ongeacht de lengte of welke karakters erin voorkomen. Dit kan men matchen door de reguliere expressie `.*` te gebruiken.

5.4 Enkele grep opties

Naast de uitgebreide mogelijkheden om reguliere expressies te specificeren bij `grep`, heeft `grep` ook nog heel wat opties. Hier volgt een opsomming van de meest gebruikte opties:

- `-i` : maakt de zoekactie case insensitive
- `-v` : inverteer de zoektocht; laat enkel de regels zien die *niet* matchen met het patroon
- `-n` : toon de regelnummers bij de gevonden regels
- `-c` : toont enkel het *aantal* gevonden regels per bestand
- `-f BESTAND` : gebruik de patronen in BESTAND (één per regel) om de zoekactie uit te voeren
- `-w` : toon enkel de regels waar een heel woord gematcht wordt

5.5 Extended reguliere expressies met egrep

Alhoewel de opdeling tussen `grep` en `egrep` eerder artificeel is, willen we toch aanduiden dat er verschillen zijn tussen de mogelijke expressies die men kan vormen. Hier volgt een lijst van extra mogelijkheden die men met `egrep` kan gebruiken.

- `+` : het voorgaande karakter wordt één of meerdere keren herhaald. Zo zal `ho+fd` matchen met de woorden *hofd*, *hoofd*, *hooofd*, *hooofd*,...
- `?` : het voorgaande karakter wordt nul of één keer gematcht. Zo zal `pee?r` matchen met de woorden *per* en *peer*.
- `|` : Geeft een aantal *mogelijkheden* op om mee te matchen. Bijvoorbeeld; de reguliere expressie `(haar|zijn)` geeft aan dat zowel de zin *is dit zijn boek?* als *is dit haar boek?* moeten matchen. De zin *dit is hun boek!* zal echter niet matchen.
- `()` : men noemt de haakjes ook wel “groeper” operatoren. Het vorige voorbeeld gaf reeds aan hoe deze gebruikt worden: om verschillende mogelijkheden te groeperen. De reguliere expressie `b(a|o|e)l` matcht enkel met de woorden *bal*, *bol* en *bel*.

tekens	grep	egrep	awk	vi
.	✓	✓	✓	✓
[]	✓	✓	✓	✓
^	✓	✓	✓	✓
\$	✓	✓	✓	✓
\{ \}	✓			
?		✓	✓	
+		✓	✓	
		✓	✓	
()		✓	✓	

Tabel 5.1: Reguliere expressies syntax ondersteuning

`\< \>` : zorgt ervoor dat enkel de karakters op het begin van een woord of het einde van een woord gematcht worden.

commando	<code>grep 'dvi\>' Makefile</code>
output	<code>DVIFILE = \$(FILE).dvi</code> <code>CLEANABLE = *.*[^e]ps *.pdf *.tex~ *.dvi *.ps.gz</code>
commando	<code>grep '\<make' Makefile</code>
output	<code>INDEX = makeindex # for generating the index</code> <code>app-callback.tex app-makefile.tex bibliotheken.tex\</code>
commando	<code>grep '\<the\>' Makefile</code>
output	<code>BIB = bibtex # for generating the BibTeX entries</code> <code>INDEX = makeindex # for generating the index</code>

5.6 Overzicht van de metakarakters

In tabel 5.1 wordt getoond welke tool en/of scripting taal de in dit hoofdstuk getoonde metakarakters kan gebruiken. De syntax en semantiek blijven ongeveer hetzelfde voor de verschillende GNU tools. Raadpleeg bij twijfel de man pagina.

5.7 e-mailadressen ontleden en zoeken

Gegeven is een bestand (`grmail.txt`) met als inhoud een lijst namen die geassocieerd worden met email-adressen (zie listing 5.1). Sommige namen worden geassocieerd met meerdere email-adressen simpelweg omdat ze meer dan 1 keer voorkomen in het bestand. We gaan enkele zoekacties in dit bestand uitvoeren met behulp van `grep` en `egrep`.

Listing 5.1: `grmail.txt`; bestand met emailadressen

<code>drs. Chris Raymaekers</code>	<code>chris.raymaekers@luc.ac.be</code>
<code>Jori Liesenborgs</code>	<code>jori.liesenborgs@luc.ac.be</code>
<code>Jori Liesenborgs</code>	<code>jori@lumumba.luc.ac.be</code>

```

drs. Tom Van Laerhoven tom.vanlaerhoven@luc.ac.be
drs. Tom Van Laerhoven tom@lumumba.luc.ac.be
prof. dr. Karin Coninx karin.coninx@luc.ac.be
prof. dr. Eddy Flerackers eddy.flerackers@luc.ac.be
drs. Kris Luyten kris.luyten@luc.ac.be

drs. Kris Luyten kris@lumumba.luc.ac.be
Jo Segers jo.segers@luc.ac.be
drs. Jan Van den Bergh jan.vandenbergh@luc.ac.be
student endroew eldritsj endroew.eldritsj@student.luc.ac.be
student sjef vanknutsel sjef.vanknutsel@student.luc.ac.be

student soja boon soja.boon@luk.ac.be
student Jozef Lumumba lumumba@congo.org

```

- Geef alle emailadressen in de lijst die een emailadres hebben op de server *lumumba*. Het is duidelijk dat `grep 'lumumba' grmail.txt` onvoldoende is. We krijgen dan als output:

```

Jori Liesenborgs jori@lumumba.luc.ac.be
drs. Tom Van Laerhoven tom@lumumba.luc.ac.be
drs. Kris Luyten kris@lumumba.luc.ac.be
student Jozef Lumumba lumumba@congo.org

```

De laatste hit is hier duidelijk geen email adres op de server *lumumba*. We kunnen `grep` ook laten weten dat *lumumba* moet voorafgegaan worden door een `@`!

`grep '@lumumba' grmail.txt` geeft ons:

```

Jori Liesenborgs jori@lumumba.luc.ac.be
drs. Tom Van Laerhoven tom@lumumba.luc.ac.be
drs. Kris Luyten kris@lumumba.luc.ac.be

```

Dit is wat we zochten!

- Verdergaand op de vorige vraag: we willen zoeken welke mensen een emailadres hebben op de server *lumumba* en aan het doctoreren zijn (*drs.*). We kunnen dit op twee manieren doen:

1. De output van de vorige oplossing pipen naar een nieuwe `grep`:

```
grep '@lumumba' grmail.txt | grep 'drs\.'
```

2. Het `grep` commando uitvoeren met een wat moeilijkere expressie:

```
grep 'drs\..*@lumumba' grmail.txt
```

Let op de `.*` hierzo, dit betekent dat een willekeurige reeks karakters kan gematcht worden.

- Tel de lege lijnen in de file met behulp van `grep`. Een lege lijn is een lijn zonder inhoud: er komt niets voor tussen het begin van de lijn (`^`) en het einde van de lijn (`$`). Met behulp van het patroon `^$` kunnen we dus zo lijnen vinden. Het aantal lijnen dat je vindt met behulp van `grep` kan je laten zien door de optie `-c` te gebruiken.

`grep -c "^$"grmail.txt` geeft ons:

Hoofdstuk 6

Editors

PENDING: *Pas begonnen, moet nog grondig afgewerkt worden, misschien aan de hand van andere (L)GPL documentatie van LDP*

6.1 Inleiding

Er zijn zeer veel verschillende editors beschikbaar voor Linux. De juiste keuze is dan ook meestal een kwestie van smaak. De meeste beginnende Linux gebruikers zullen `pico` gebruiken om tekstbestanden te editeren. Pico is een editor die bij de emailclient `pine` hoort.

Pico is echter verre van een krachtige editor. Het kan enkel de basisbewerkingen doen. In dit hoofdstuk worden er twee editors besproken die tot de krachtigste editors die er te vinden zijn behoren, namelijk `vim`¹ en `emacs`².

6.2 Vi/Vim

6.2.1 Vi

Vim staat voor *Vi improved*. Vim is een uitbreiding van Vi, met meer flexibiliteit en meer opties. Eerst zullen we wat uitweiden over de basis, namelijk Vi, waarna er wat wordt ingegaan op Vim. De tekst die je nu aan het lezen bent is trouwens geschreven met behulp van Vim!

Je kan vim opstarten en een bestand laten laden door het commando: `vim test.txt`. Dit zorgt ervoor dat het bestand “test.txt” geladen wordt en, indien dit nog niet bestond, zal het gecreëerd worden. Je kan vim afsluiten door `:q` in te drukken (let wel, je moet hiervoor in command mode zijn, zie sectie 6.2.1. Om ervoor te zorgen dat vim ook de aangebrachte wijzigingen vergeet druk je `:q!` in.

Verschillende modes

Waarmee beginnende Vi gebruikers nogal eens last hebben zijn de verschillende modes die je moet gebruiken om “commando’s” te geven en tekst te typen. Als je gewoon tekst kan intypen ben je bezig in *insert mode*, ook wel *input mode* genoemd. Voor het uitvoeren van commando’s, zoals het bestand opslaan, vi afsluiten of een tekst te plakken, moet je in *command mode* zijn. Meestal zit je standaard in de command mode als je opstart. Je kan dan naar insert mode gaan door gewoon `i` in te drukken. Om zeker te zijn dat je in command mode zit kan je dan eerst de Escape ([ESC]) indrukken. Je gebruikt [ESC] ook als je gewoon naar command mode wil gaan.

¹<http://www.vim.org>

²<http://www.gnu.org/software/emacs/>, <http://www.xemacs.org>

Enkele standaard commando's

In feite zijn er twee verschillende soorten commando's: degene die direkt op de tekst werken door een toets of een toetsencombinatie in te drukken, of degene die op de laatste regel van het scherm verschijnen die op een bepaald stuk tekst werken (ook wel *last line mode* of *ex mode* genoemd). Zo een *ex mode* commando begint meestal met een dubbele punt ':'. Als je een commando intypt (in *command mode*) dan zie je dit onderaan het scherm verschijnen.

In tabel 6.1 is een lijstje van mogelijke commando's voor vi. Deze commando's kunnen gebruikt worden wanneer men zich in de *command-mode* bevindt. Na het uitvoeren van sommige van deze commando's kan het zijn dat men zich in *insert mode* bevindt. Naast de commando's die in tabel 6.1 opgesomd worden, zijn er nog vele andere mogelijkheden. Hiervoor verwijzen we door naar de documentatie en `:help`.

Een belangrijk aspect van deze commando's is dat men ze kan combineren, door de verschillende "codes" simpelweg na elkaar te plaatsen vooraleer op [enter] te drukken. Door een getal x voor een commando te zetten kan je dit ook x keer laten herhalen. Enkele voorbeelden hiervan zijn:

d5\$ verwijdert tot op het einde van de huidige lijn en verwijdert ook de volgende 4 lijnen

y10w kopieert de volgende 10 woorden na de cursor

d10w cut de volgende 10 woorden (kopieert en verwijdert)

PENDING: *Nog voorbeeldjes verzinnen*

6.2.2 Zoeken en Vervangen in Vim

Vim heeft een uitgebreide ondersteuning voor het gebruik van *reguliere expressies* om zoekacties uit te voeren en stukken tekst te vervangen. De mogelijkheden en een korte praktische introductie tot reguliere expressies vind je in hoofdstuk 5. Tabel 5.1 uit hoofdstuk 5 geeft al een idee wat je allemaal kan doen met reguliere expressies in Vim.

Om in Vim naar een patroon van karakters te zoeken moeten we on eerst in *command mode* bevinden. Zorg ervoor dat je dus eerst [ESC] gedrukt hebt vooraleer de zoekopdracht uit te voeren. We geven aan dat we naar een patroon willen zoeken door eerst '/' in te typen. Daarna typen we het patroon in dat we zoeken. Dus `/modes` invoeren en daarna [enter] drukken brengt ons naar het eerste voorkomen van het woord "modes" achter de cursor. Als man naar het volgende voorkomen wil verdergaan moet op de toets "n" gedrukt worden.

Naast simplele woorden kan men ook gesofisticeerdere zoekacties uitvoeren met behulp van reguliere expressies. Wil men bijvoorbeeld de woorden vinden die de letters *s* en *t* bevatten en daartussen exact drie andere willekeurige letters staan hebben dan geeft met de zoekopdracht `/s...t` in.

6.3 Emacs

Emacs kan je opstarten door het commando `emacs` in te geven. Op deze manier start emacs op met een *buffer* met de naam `*scratch*`. Emacs laad bestanden namelijk in buffers. Je kan ook één of meerdere bestanden die je wil editeren als argument meegeven. Zo zal `emacs mijntext.txt oef27.sh` de bestanden `tt mijntext.txt` en `oef27.sh` in twee buffers laden die geassocieerd worden met deze bestanden. Deze buffers kunnen dan naar het eigenlijke bestand op de schijf geschreven worden.

commando	betekenis
:q	afsluiten
:q!	afsluiten zonder de laatst gedane wijzigingen op te slaan
:w	het huidige bestand opslaan
:w <i>filename</i>	sla het bestand op met bestandsnaam <i>filename</i>
:qw	afsluiten en het bestand opslaan
:e <i>filename</i>	open het bestand <i>filename</i> in een aparte buffer
:next	ga naar de volgende buffer
:previous	ga naar de vorige buffer
:split	laat meerdere buffers op het scherm zien
:help <i>commando</i>	Open een nieuwe buffer waar de help informatie voor <i>commando</i> wordt getoond.
a	begin met tekst in te voegen achter de huidige positie van de cursor
i	begin met tekst in te voegen voor de huidige positie van de cursor
A	begin met tekst in te voegen aan het einde van de huidige lijn
I	begin met tekst in te voegen aan het begin van de huidige lijn
o	begin met een nieuwe lijn tekst in te voegen onder de huidige lijn
O	begin met een nieuwe lijn tekst in te voegen boven de huidige lijn
dd	wis een lijn
d	delete commando, doet een “cut” operatie
y	kopieer (<i>yank</i>) de geselecteerde tekst
p	plak de geselecteerde tekst
\$	ga naar het einde van de regel
^	ga naar het begin van de regel
w	ga naar het volgende woord
v	ga naar “visual mode”, de gebruiker kan nu een stuk tekst selecteren

Tabel 6.1: Veel gebruikte vim commando's

bla	bla
-----	-----

Tabel 6.2: Veel gebruikte Emacs toetsencombinaties

6.4 Toetsencombinaties

In tegenstelling tot Vi, waar men met “modes” werkt (zie sectie sectie 6.2.1), bestaat dit concept niet onder Emacs. Deze editor vereist speciale toetsencombinaties om commando’s te geven. Om deze combinaties te vormen wordt er gebruik gemaakt van de **control**-toets ([Ctrl]) en de **meta**-toets (meestal [Alt]).

PENDING: *Zelf eerst nog wat emacs bekijken of andere expert vinden*

Hoofdstuk 7

Shell scripting

PENDING: *input redirectie ook nog vermelden, bijv mail bla < tekst*

7.1 Shell

De shell is een programma dat zorgt voor de “interface” tussen de gebruiker en de kernel. Het laat toe om opdrachten door de computer te laten uitvoeren. Ieder commando dat wordt ingetypt wordt door de shell geïnterpreteerd en doorgegeven aan de kernel.

UNIX heeft verschillende soorten shells. De belangrijkste zijn: de Bourne shell, de Korn shell en de C shell.

De Bourne shell (sh) is de originele UNIX shell en is daarom beschikbaar in alle UNIX versies. Deze shell is zeer geschikt voor shell programmeren, maar is niet erg gebruiksvriendelijk.

De C shell (csh) is krachtiger dan de Bourne shell. Hoewel veel mensen deze shell niet zo goed vinden als de Bourne shell, wordt deze shell door veel C programmeurs gebruikt omdat de syntax sterk lijkt op de C syntax.

De Korn shell (ksh) combineert de voordelen van de Bourne shell en de C shell en is compatibel met de Bourne shell.

Andere shells die op deze shells zijn gebaseerd, zijn o.a. de *Bourne again shell* (bash) en *tcsh*. Dit hoofdstuk is gebaseerd op sh, bash en ksh. Vergelijkbare technieken bestaan ook voor csh en tcsh, soms met kleine verschillen o.a. in de syntaxis.

7.2 Aliases

Ingewikkelde commando's die vaak worden gebruikt zouden best een kortere vorm krijgen. Dit gebeurt door het commando `alias`. Een voorbeeld is `alias ll="ls -l"`. Als dit commando is uitgevoerd kan `ll` worden gebruikt in plaats van `ls -l`.

Om het aantal alias definities in de file `.login` te tellen kunnen we het volgende commando uitvoeren: `cat .login | grep alias | wc -l`. Het eerste commando toont de file met omgevingsvariabelen (zie sectie 4.2). Op dit resultaat wordt het `grep` commando toegepast. Dit commando gaat alleen de lijnen weerhouden, waarin het woord “alias” voorkomt. Tenslotte wordt met het `wc` (word count) commando het aantal lijnen geteld (dankzij de optie `-l`). Het resultaat van de hele regel is het aantal aliases dat in de `.login` file is gedefinieerd. Als je met de *Bourne Again Shell* werkt moet je het bestand `.bashrc` doorzoeken in plaats van `.login`.

7.3 Programmeren in de Shell

De Linux shell wordt niet alleen gebruikt om shell commando's door te geven aan de kernel. De shell kan ook scripts parsen en interpreteren. Zo kunnen reeksen van statements worden samengevoegd tot een *shell script*. Dit maakt shell programming zeer krachtig en zeer populair.

Een shell script is een tekst file, die uitvoerbaar is. De eerste regel van deze tekst file moet aangeven wat de interpreter van het script is. Omdat dit eigenlijk een regel commentaar is, moet het met het commentaar symbool beginnen (`#`), gevolgd door een uitroepteken om aan te geven dat de interpreter hierop volgt. Alle voorbeelden en oefeningen van deze cursus zullen steeds beginnen met `#!/bin/sh`, om aan te geven dat de Bourne shell de interpreter is van het script.

Deze tekst file kan op drie manieren worden uitgevoerd. Ten eerste kan expliciet worden aangegeven wat de interpreter is van de tekst file door de tekst file als argument aan de shell mee te geven: `/bin/sh scriptname`. Het shell script kan ook door de huidige shell worden geïnterpreteerd: `./scriptname` intypen achter de prompt voert het script uit. Vaak wordt het script hiervoor executable gemaakt met behulp van `chmod` (zie ook paragraaf 4.5). Het script kan dan worden uitgevoerd door de naam in te typen: `scriptname`.

7.4 Variabelen

Zoals de meeste talen kent ook shell scripting variabelen. In tegenstelling tot variabelen in de meeste programmeertalen, hebben shell variabelen geen type. Een nieuwe variabele wordt aangemaakt door gewoon een nieuwe naam te gebruiken. Daarom is het bij scripting zeer belangrijk om de file op fouten te controleren. Enkele voorbeelden zijn.

```
aantal=5
```

```
voornaam=Chris
```

```
achternaam=Raymaekers
```

Het is zeer belangrijk dat er voor en na = geen spaties worden geplaatst, anders weet de shell niet hoe het commando moet worden geïnterpreteerd. De waarde van een variabele wordt opgevraagd door er een dollarteken (`$`) voor te plaatsen. Vb:

```
naam="$voornaam $achternaam"
```

Met behulp van het `echo` statement kan de waarde ook op het scherm uitgeschreven worden. Ook hier moet het dollarteken worden gebruikt. Vb.:

```
echo naam           resultaat: naam
echo $naam          resultaat: Chris Raymaekers
```

Anderzijds kan je met de `read` opdracht een waarde inlezen. Het voorbeeld in listing 7.1 laat zien hoe dit in zijn werk gaat

Listing 7.1: de read opdracht

```
#!/bin/sh
eentekst="Shell programmeren is fijn"
echo $eentekst
echo "Typ nu zelf een regel in"
read eentekst
echo $eentekst
```

Dit voorbeeldje laat zien hoe de variabele `eentekst` eerst een initiële waarde toegekend krijgt welke in de volgende stap uitgeschreven wordt. Daarna wordt er gevraagd om zelf een regel in te typen, dat ingelezen wordt in de variabele `eentekst` en daarna uitgeschreven. Een ander voorbeeldje vind je in listing 7.2

Listing 7.2: de read opdracht, voorbeeld 2

```
#!/bin/sh
echo "geef je voor- en achternaam, gevolgd door je universiteit,"
echo "je richting en je hobbies"
read voornaam achternaam universiteit richting hobbies
echo "voornaam: $voornaam"
echo "achternaam: $achternaam"
echo "universiteit: $universiteit"
echo "richting: $richting"
echo "hobbies: $hobbies"
```

Als je bij het programma voorgesteld in listing 7.2 de invoer: `Kris Luyten LUC informatica mensen irriteren met domme opmerkingen` intypt zal je de volgende uitvoer bekomen:

```
voornaam: Kris
achternaam: Luyten
universiteit: LUC
richting: informatica
hobbies: mensen irriteren met domme opmerkingen
```

Linux kent ook nog een reeks ingebouwde variabelen, waarvan de waarde kan worden opgevraagd.

<code> \$# </code>	Het aantal argumenten dat aan het script is meegegeven
<code> \$? </code>	De exit waarde van het commando dat het laatste is uitgevoerd
<code> \$0 </code>	De naam van het shell script (dit is impliciet het eerste argument)
<code> \$* </code>	De positionele parameters <code>\$1</code> , <code>\$2</code> ,... die de parameters bevatten Vervang <code>*</code> door de gewenste parameter nummer.
<code> @\$ </code>	De volledige lijst van parameters (een “array” van alle argumenten)
<code> \$\$ </code>	De ID van het huidige proces

Deze ingebouwde variabelen kunnen gebruikt worden in shell scripts, zoals getoond in listing 7.3.

Listing 7.3: Ingebouwde variabelen

```
#!/bin/sh
echo the name for this script is $0 and it has $# arguments
echo the first argument of this script is $1
echo the fifth argument of this script is $5
```

Nu kunnen we desgewenst ook argumenten meegeven als we shell scripts gaan schrijven.

7.5 Speciale karakters

Linux heeft een reeks van speciale karakters. In deze paragraaf worden vier belangrijke karakters behandeld: double quotes (`"`), single quotes (`'`), back slash (`\`) en back tick (```).

Double quotes worden gebruikt om een string die spaties bevat uit te schrijven of toe te kennen aan een variabele. Als de double quotes worden weggelaten, interpreteert de shell

Operator	Betekenis
int1 -eq int2	int1 = int2 (equal)
int1 -ge int2	int1 \geq int2 (greater than or equal)
int1 -gt int2	int1 > int2 (greater than)
int1 -le int2	int1 \leq int2 (less than or equal)
int1 -lt int2	int1 < int2 (less than)
int1 -ne int2	int1 \neq int2 (not equal)

Tabel 7.1: Integer expressies

ieder woord als een nieuw commando. Hierdoor zouden ernstige fouten kunnen ontstaan bij de uitvoering van het script. De waarde van variabelen worden nog steeds ingevuld.

Single quotes zijn sterker dan double quotes. Een string die tussen single quotes staat wordt letterlijk geïnterpreteerd. Indien slechts een gedeelte van een expressie letterlijk moet worden genomen, dan kan men best double quotes gebruiken in combinatie met de back slash. Dit teken zorgt er namelijk voor dat het volgend teken letterlijk wordt genomen. Vb.:

```
echo "Hello $LOGNAME"          resultaat: Hello craymaek
echo 'Hello $LOGNAME'         resultaat: Hello $LOGNAME
echo "Hello \$LOGNAME"        resultaat: Hello $LOGNAME
```

Met behulp van de back tick (`) en het `expr` commando kan het resultaat van een bewerking aan een variabele worden toegekend. Dit wordt getoond in listing 7.4

Listing 7.4: Toekenning aan variabelen

```
een=1
twee='expr $een + 1'
echo $twee
datum='date'
echo $datum
```

De back tick tenslotte wordt gebruikt om een string *uit te voeren*. Het commando `info='ls'` stelt de variabele `info` niet gelijk aan "ls", maar aan de inhoud van de huidige directory.

7.6 Vergelijken van expressies

Het commando om een logische expressie te evalueren is `test`. De syntax is `test expressie`. Vaak worden vierkante haakjes gebruikt in plaats van het `test` commando: `[expressie]`. Het resultaat is in beide gevallen hetzelfde. Tabel 7.1, tabel 7.2, tabel 7.3 en tabel 7.4 geven de verschillende expressie weer. Enkele voorbeelden van expressies zijn:

- `[5 -eq 6]` geeft vals
- `[-n lala]` geeft waar
- `[-d /dev]` geeft waar
- `[-f /home]` geeft vals

Deze expressies komen pas helemaal tot hun recht bij conditionele expressies en condities voor het uitvoeren van herhalingslussen. Dit wordt besproken in de volgende secties.

Operator	Betekenis
<code>str1 = str2</code>	<code>str1 = str2</code> (equal)
<code>str1 != str2</code>	<code>str1 ≠ str2</code> (not equal)
<code>str1</code>	<code>str1</code> is niet null
<code>-n str1</code>	de lengte van <code>str1</code> is groter dan 0
<code>-z str1</code>	de lengte van <code>str1</code> is 0

Tabel 7.2: String expressies

Operator	Betekenis
<code>-d bestandsnaam</code>	bestand (<i>bestandsnaam</i>) is een directory ^a
<code>-f bestandsnaam</code>	bestand is een gewone file
<code>-r bestandsnaam</code>	bestand kan worden gelezen door het proces
<code>-s bestandsnaam</code>	bestand heeft een lengte groter dan 0
<code>-w bestandsnaam</code>	bestand kan worden beschreven door het proces
<code>-x bestandsnaam</code>	bestand kan worden uitgevoerd door het proces
<code>-e bestandsnaam</code>	bestand bestaat

^ain UNIX is een directory in weze ook een bestand

Tabel 7.3: Bestand expressies

Operator	Betekenis
<code>! expr1</code>	Negatie van <code>expr1</code>
<code>expr1 -a expr2</code>	<code>expr1</code> en <code>expr2</code>
<code>expr1 -o expr2</code>	<code>expr1</code> of <code>expr2</code>

Tabel 7.4: Logische expressies

7.7 Conditionele expressies

De Bourne shell kent twee conditionele expressies: *if* en *case*. De syntax van het *if* statement is:

Listing 7.5: *if* statement syntax

```
if [ expressie ]
then
    statements
elif [ expressie ]
then
    statements
else
    statements
fi
```

Een voorbeeldje van het gebruik van een *if*-statement in een shell script vind je in listing 7.6.

Listing 7.6: *if* statement voorbeeld

```
if [ $var = "Yes" ]
then
    echo "Value is Yes"
elif [ $var = "No" ]
then
    echo "Value is No"
else
    echo "Invalid value"
fi
```

Het *case* commando is vergelijkbaar met het *switch* commando van C. De syntax is beschreven in listing 7.7

Listing 7.7: *case* statement syntax

```
case word in
    str1)
        statements;;
    str2 | str3)
        statements;;
    *)
        statements;;
esac
```

Een voorbeeldje van het gebruik van een *case*-statement in een shell script vind je in listing 7.8.

Listing 7.8: *case* statement voorbeeld

```
case $1 in
    01 | 1) echo "Month is January";;
    02 | 2) echo "Month is February";;
    03 | 3) echo "Month is March";;
    04 | 4) echo "Month is April";;
```

```
05 | 5) echo "Month is May";;
06 | 6) echo "Month is June";;
07 | 7) echo "Month is July";;
08 | 8) echo "Month is August";
09 | 9) echo "Month is September";;
10) echo "Month is October";;
11) echo "Month is November";;
12) echo "Month is December";;
*) echo "Invalid Parameter";;
esac
```

7.8 Iteraties

Een iteratie in een shell script kan op drie verschillende manieren worden geïmplementeerd: for, while en until.

7.8.1 for

De syntax van het for statement is:

Listing 7.9: for statement syntax

```
for var in list
do
    statements
done
```

Een voorbeeldje van het gebruik van een for-statement in een shell script vind je in listing 7.10. Het script in listing 7.10 schrijft de huidige directory uit.

Listing 7.10: for statement voorbeeld

```
#!/bin/sh
dirlist='ls'
for file in $dirlist
do
    echo $file
done
```

Stel dat je in een directory heel wat tar.gz files staan hebt die je allemaal wilt decomprimeren en uitpakken, dan kan je daarvoor een soortgelijk script schrijven. Een voorbeeldje hiervan zie je in listing 7.11.

Listing 7.11: untarren en gunzippen m.b.v. een for lus

```
#!/bin/sh
for file in `ls *tar.gz`
do
    tar -xvzf $file
done
```

Merk op dat 'ls *tar.gz' een verzameling files teruggeeft waarover geïtereerd wordt. Je kan in plaats van het ls commando uit te voeren, het te expanden patroon meegeven: `for file in *tar.gz`. De shell zal dit dan eerst expanden tot all tar.gz files in de huidige directory. In

feite hoeft dit helemaal niet in een bestand gezet te worden. We zouden ook het script kunnen intypen achter de shell prompt, waarbij we de enters vervangen door `;`. Zo zou je in plaats van listing 7.11 in een bestand te plaatsen ook kunnen intypen:

```
for file in 'ls *tar.gz'; do tar -xvzf $file ; done
```

Dit heeft hetzelfde effect.

7.8.2 while

De syntax van het while statement is:

Listing 7.12: while statement syntax

```
while expressie
do
    statements
done
```

Een voorbeeldje van het gebruik van een while-statement in een shell script vind je in listing 7.13.

Listing 7.13: while statement voorbeeld

```
#!/bin/sh
i=1
while [ $i -le 10 ]
do
    echo $i
    i='expr $i + 1'
done
```

Het script in listing 7.13 telt tot 10 en schrijft de getallen uit.

7.8.3 until

Tenslotte is er nog het until statement waarvan de syntax beschreven wordt in listing 7.14. Het until statement voert de code tussen `do` en `done` uit tot de conditie die achter `until` staat voldaan is. Als deze conditie voldaan is voordat de lus voor de eerste keer uitgevoerd is, wordt de lus gewoon niet uitgevoerd.

Listing 7.14: until statement syntax

```
until expressie
do
    statements
done
```

Een voorbeeldje van het gebruik van een until-statement in een shell script vind je in listing 7.15.

Listing 7.15: until statement voorbeeld

```
#!/bin/sh
i=1
until [ $i -gt 10 ]
do
    echo $i
    i='expr $i + 1'
done
```

Ook het voorbeeld uit listing 7.15 telt tot 10 en schrijft de getallen uit.

7.9 Functies

Scripts voor de Bourne shell kunnen ook functies bevatten. De syntax is voorgesteld in listing 7.16.

Listing 7.16: functie syntax

```
functienaam () {
    statements
}
```

De functie kan worden opgeroepen met de functienaam. Ook kunnen er parameters worden doorgegeven: `functienaam param1 param2 ...`. Deze parameters kunnen dan in de functie worden aangeroepen door middel van de positionele parameters (`$1`, `$2`, ...). Een voorbeeldje van een functie die argument1 argument2-keer met zichzelf optelt (zoals een gewone vermenigvuldiging dus) vind je in listing 7.17. Merk op dat de functie voor de aanroep moet gedefinieerd worden.

Listing 7.17: functie voorbeeld

```
#!/bin/sh
telkeerop(){
    result=$1
    teller=$2
    until [ $teller -eq 1 ]
    do
        result='expr $result + $1'
        teller='expr $teller - 1'
    done
    echo $result
}
```

```
telkeerop 5 7
```

7.10 Scripts onderbreken

Een iteratie (for, while en until) kan worden onderbroken door middel van het commando `break`. Het hele script kan beëindigd worden met het commando `exit`.

Een laatste commando is `shift`. Hiermee worden de positionele parameters één plaats opgeschoven. Een voorbeeld hiervan vind je in listing 7.18. Dit voorbeeld schrijft alle parameters uit. De parameters kunnen ook meerdere plaatsen (n) plaatsen worden opgeschoven door `shift n`.

Listing 7.18: shift voorbeeld

```
#!/bin/sh
while [ $# -ne 0 ]
do
    echo $1
    shift
done
```

7.11 eval

Beschouw het stukje code in listing 7.19. We zouden verwachten dat de uitvoer van dit stukje code “blaiwaai” is, maar we krijgen “\$bla” als output te zien.

Listing 7.19: zonder gebruik te maken van eval

```
#!/bin/bash
bla=blaiwaai
wa=bla
z='$'$wa
echo $z
```

De reden hiervoor is eenvoudig. Door de statement `z='$'$wa` wordt aan `z` niet de inhoud van de variabele `$wa` toegekend, maar wel de string “\$bla”. `$wa` evalueert namelijk als de string “bla”, en wordt aan het karakter ‘\$’ vastgehangen.

We hebben dus iets nodig dat de *waarde van de waarde van een variabele* teruggeeft: `eval`. Dit stelt ons in staat om zeer krachtige, dynamische programma’s te bouwen. Alleen wordt een script door het gebruik van `eval` niet alleen krachtiger maar ook complexer en dus ook moeilijker leesbaar en moeilijker te debuggen. Listing 7.20 Laat zien hoe `eval` werkt. Na het uitvoeren van het scriptje in listing 7.20 krijgen we wel de verwachte uitvoer “blaiwaai”.

Listing 7.20: met gebruik te maken van eval

```
#!/bin/bash
bla=blaiwaai
wa=bla
eval z='$'$wa
echo $z
```

7.12 Opgave

1. Leg uit wat het script in listing 7.21 doet. Maak gebruik van de man paginas voor de opdrachten die je niet kent.
2. Schrijf een script dat een reeks van files aanvaardt en naar een back-up directory kopiëert. Zorg ervoor dat de kopies read-only zijn en maak een gezippt archive van de back-up directory. Vergeet niet om alle tijdelijke files en directories op te ruimen.
3. Maak een script dat een reeks van files aanvaardt en voor iedere file een functie aanroept. Deze functie schrijft de gegevens van de file (`ls -l`) en de inhoud uit.
4. Schrijf een script dat een directory als parameter krijgt en de inhoud per email naar de gebruiker stuurt. Tip: bekijk de man pages van mail.

5. Maak een script dat een lijst van email-adressen meekrijgt en als laatste argument een tekst. Zorg ervoor dat deze tekst doorgestuurd wordt naar alle email-adressen.
6. Maak een script dat als argument de login van een gebruiker meekrijgt en een melding op het scherm geeft wanneer dat die gebruiker inlogt. Dit wil zeggen dat het script moet blijven werken totdat de gebruiker inlogt, waarna het een melding geeft en dan stopt. Tip: u kan hiervoor het `sleep` en het `grep` commando gebruiken.
7. Breid het vorige programma uit zodat je een lijst van loginnamen kan meegeven om te melden. Het script is dan bedoeld om vanaf het starten ervan altijd te blijven lopen.
8. Maak een script dat als invoer shell-commando's krijgt en deze zelf uitvoert, waarbij er een minimale boekkeeping gedaan wordt. Schrijf de naam van het process, uitvoeringstijd en starttijd in een file, samen met de gebruiker die het process opgestart heeft. Dit programma moet altijd blijven werken. Gebruik `read` voor de invoer. Zorg ervoor dat je het script kan beëindigen door `quit` in te typen.

PENDING: *Enkele oefeningen met grep toevoegen*

Listing 7.21: Wat doet dit script?

```
#!/bin/bash
if [ -d ~/dustbin ]
then
:
else
mkdir ~/dustbin
fi

i=1
currentdir='pwd'
mkdir -p ~/dustbin$currentdir
movedate='date +%s'
while [ $i -le $# ]
do
if [ -e $1 ]
then
mv $1 ~/dustbin$currentdir/$1.$movedate
echo moving $1 to dustbin (" $1 "->"$1.$movedate)"
fi
shift
i='expr $i + 1'
done
```

Hoofdstuk 8

Gnu Awk

GNU bevat een hoop programma's om taken te automatiseren. Deze programma's zijn meestal heel goed om één taak uit voeren. En uitzondering hierop is *gawk*, de GNU versie van *awk*. *awk* is namelijk een scripting taal die kan gebruikt worden om verschillende soorten taken te automatiseren. *awk* is genoemd naar de 3 auteurs (Aho, Weinberger en Kernighan).

Deze cursus behandelt *gawk* omdat deze versie wordt meegeleverd bij de meeste Linux distributies. De verschillen tussen de verschillende *awk* versies zijn echter miniem. De principes die in deze cursus worden uitgelegd gelden dus grotendeels voor alle *awk* versies.

8.1 Principes van AWK

AWK is een scripting taal die gebruikt wordt om informatie uit tekstfiles te verwerken. De programmeur moet zelf geen files openen, lezen of sluiten; dit zal *awk* automatisch doen. De verschillende regels van de tekst files worden zelfs opgedeeld in tekstvelden.

awk kan worden opgestart vanuit de commandolijn, zoals in dit eerste voorbeeld:

```
gawk '{print NF ": "$0}' file.txt
```

Stel dat de file "file.txt" de volgende inhoud heeft:

Dit is een eerste voorbeeld.

Deze file bevat 3 regels.
zoals een test!

Dan is het resultaat van het script:

```
5: Dit is een eerste voorbeeld.  
5: Dit voorbeeld bevat       3           regels.  
3: zoals een test!
```

Het script telt dus de woorden in iedere regel in de tekst file en schrijft dit uit. De invoer van het *awk* script is het resultaat van `cat file.txt`, de inhoud van deze file. *awk* zal vervolgens iedere regel van de invoer in velden onderverdelen. Omdat de verschillende velden door "whitespace" (een combinatie van spaties en tabs) worden gescheiden, komen de verschillende velden overeen met de woorden van het script. Vervolgens wordt met het `print` commando, het aantal velden van iedere regel (aangegeven door de variable `NF`) en de regel zelf (aangegeven door de variable `$0`) uitgeprint. Het is ook mogelijk om de verschillende velden uit te printen:

```
gawk '{print $1 " ... "$NF}' file.txt
```

Het resultaat is dan:

```
Dit ... voorbeeld.  
Dit ... regels.  
zoals ... test!
```

Ingewikkeldere *awk* scripts worden in een file gezet. Op deze manier kan een script meerdere keren worden uitgevoerd. Bv. het script listing 8.1 zal hetzelfde resultaat geven als het vorige voorbeeld, wanneer het wordt aangeroepen zoals afgebeeld in listing 8.2.

Listing 8.1: count.awk

```
{print $1 " ... "$NF}
```

Listing 8.2: count met awk

```
gawk -f count.awk file.txt
```

De meeste van de voorbeelden in deze tekst zullen gebruik maken van een voorbeeld password file (genaamd “voorbeeld”). Dit is de file (`/etc/passwd`) waarin UNIX systemen alle gebruikersinformatie opslaan:

```
root:x:0:0:Super User:/root:/bin/sh
chris:x:101:20:Chris Raymaekers:/home/chris:/bin/tcsh
john:x:102:20:John Doe:/home/john:/bin/csh
jane:x:103:20:Jane Doe:/home/jane:/bin/sh
peter:x:104:30:Peter Icshello:/home/peter:/bin/sh
```

Iedere regel in deze file geeft informatie over één gebruiker. Het eerste veld geeft de gebruikersnaam aan. In het tweede veld staat het geëncrypteerde password van deze gebruiker (x geeft aan dat het password in een meer beveiligde file staat). De 2 volgende velden geven een getal aan dat de gebruiker uniek aanduidt en het getal dat zijn groep¹ aanduidt (gebruikers worden gegroepeerd, zodat bepaalde rechten aan een groep van gebruikers kunnen worden toegekend). Het vijfde veld bevat de volledige naam van de gebruiker. Uiteindelijk komen de home directory en de shell van de gebruiker (zie ook hoofdstuk 7).

Als we deze file willen splitsen in meerdere regels, dan moeten we aangeven dat de verschillende velden niet gescheiden worden door whitespace, maar door `:` (de field separator is `:`). Dit kan aan *awk* worden aangegeven door de optie `-F ":"` aan *awk* mee te geven. Het is echter vervelend om dit iedere keer aan te geven als *awk* wordt aangeroepen. Een *awk* script kan echter een initialisatieblok bevatten, waarin dit soort variabelen wordt geïnitieerd:

```
BEGIN { FS = ":" }
{print $5}
```

De aanroep

```
gawk -f users.awk voorbeeld
```

heeft als resultaat:

```
Super User
Chris Raymaekers
John Doe
Jane Doe
Peter Icshello
```

Het blok, aangegeven door `BEGIN`, is het initialisatieblok en wordt dus slechts eenmaal aangeroepen. Het andere blok (op de volgende lijn) vormt het eigenlijke script en wordt voor iedere regel van het voorbeeldbestand aangeroepen. Een overzicht van de belangrijkste voorgedefiniëerde variabelen (zoals `FS`) wordt gegeven in tabel 8.1.

Op dezelfde manier kunnen we een blok specificeren dat enkel op het einde wordt uitgevoerd:

```
BEGIN { FS = ":" }
{print $5}
END { print NR " gebruikers"}
```

¹Een groep heeft ook gebruiksvriendelijke naam, zoals “users”. Deze informatie staat in `/etc/groep`.

Variabele	Betekenis	Standaardwaarde
FILENAME	Naam van het invoerbestand	
FS	Input field separator	whitespace
NF	Aantal velden in het huidige record	
NR	Aan records (lijnen) die al zijn gelezen	
ORS	Output record separator	Newline (enter)
RS	Input record separator	Newline (enter)

Tabel 8.1: Voorgedefinieerde variabelen

De uitvoer van dit script geeft:

```
Super User
Chris Raymaekers
John Doe
Jane Doe
Peter Icshello
5 gebruikers
```

Ook *awk* scripts kunnen commentaar bevatten: het teken `#` geeft aan dat de rest van de regel commentaar bevat.

8.2 Output

Tot nu toe, werd in alle voorbeelden gebruik gemaakt van het `print` commando. Dit commando print de de gebruikte variabelen rechtstreeks uit (en voegt een newline toe). Eventuele getallen worden tot op 6 cijfers na de comma afgeprint. Het is echter ook mogelijk om zelf het formaat van de output te kiezen. Dit gebeurt met de functie `printf(format-specifier, variabele1, variabele2, ...)`. De “format-specifier” is een string (reeks van karakters) die aangeeft hoe de variabelen *variabele1*, *variabele2*, ... worden weergegeven. De lijst van de meest belangrijke formatie specifiers staat in tabel 8.2.

Een voorbeeld van deze specifiers is:

```
printf( "%i %f %e\n", 45, 3.2, 10 )
printf( "%g %g\n", 10, 10000000000 )
printf( "%c %c %s\n", 64 "a", "abc")
```

Resultaat:

```
45 3.2 1.00000E+01
10 1.00000E+10
@ a abc
```

Vaak volstaat echter het `print` commando.

8.3 Input

De voorgaande voorbeelden printen alle lijnen uit. Soms willen we echter de invoer gaan filteren: we willen niet uit alle record gegevens gaan uitschrijven. De verschillende velden kunnen gecontroleerd worden met een aantal vergelijksoperatoren (zie ook tabel 8.3). Voorbeeld listing 8.3 schrijft de namen van de gebruikers uit, waarvan de groeps ID (veld #4) gelijk is aan 20.

Variabele	Betekenis
<code>\n</code>	newline (enter)
<code>\t</code>	tab
<code>\"</code>	"
<code>\\</code>	\
<code>%i</code>	integer (geheel getal)
<code>%f</code>	floating point (reëel getal)
<code>%e</code>	floating point in wetenschappelijke notatie
<code>%g</code>	<code>%f</code> of <code>%e</code> (wat het kortste is)
<code>%c</code>	ASCII karakter
<code>%s</code>	string (reeks van karakters)

Tabel 8.2: Format specifiers

Listing 8.3: Schrijf gebruikersnamen uit met groep ID=20

```
BEGIN { FS = ":" }
$4 == 20 {print $5}
```

De uitvoer van het script in listing 8.3 geeft:

```
Chris Raymaekers
John Doe
Jane Doe
```

Operator	Betekenis
<code>==</code>	gelijk
<code>!=</code>	niet gelijk
<code>></code>	groter dan
<code>>=</code>	groter dan of gelijk aan
<code><</code>	kleiner dan
<code><=</code>	kleiner dan of gelijk aan
<code>~</code>	Voldoet aan een reguliere expressie(zie paragraaf 8.5)
<code>!~</code>	Voldoet niet aan een reguliere expressie(zie paragraaf 8.5)

Tabel 8.3: Vergelijkingsoperatoren

Verschillende testen kunnen met behulp van logische operatoren worden gecombineerd (zie tabel 8.4).

Operator	Betekenis
<code>&&</code>	logische en
<code> </code>	logische of
<code>!</code>	logisch eniet
<code>()</code>	combineren van logische expressies

Tabel 8.4: Logische operatoren

8.4 Variabelen

In *awk* kunnen ook variabelen worden gedefinieerd om berekeningen en tussentijdse waarden op te slaan. Een variabele moet niet expliciet worden gedefinieerd. De naam is case-sensitive; pas dus goed op dat je de naam van een variabele altijd op dezelfde manier schrijft, anders kun je per ongeluk een nieuwe variabele aanmaken. Een nieuwe variabele wordt steeds op 0 geïnitieerd.

Voorbeeld listing 8.4 telt alle woorden in het voorbeeldbestand “file.txt” (een overzicht van de belangrijkste wiskundige operatoren staat in tabel 8.5)

Listing 8.4: Tel alle woorden

```
{ teller += NF }
END { print teller }
```

Operator	Betekenis
+	optelling
−	afrekking
*	vermenigvuldiging
/	deling
%	rest na deling
^	macht
=	toekenning

Tabel 8.5: Wiskundige operatoren

Met behulp van een de functie `split`, kan een veld worden opgesplitst in meerdere velden. Voorbeeld listing 8.5 schrijft de achternaam uit van alle gebruikers van de voorbeeld password file.

Listing 8.5: Schrijf de achternaam uit

```
BEGIN { FS = ":" }
$3 != 0 { split( $5, naam, " "); print naam[2] }
```

Merk op dat `split` het resultaat in een array opslaat. Het verschil tussen arrays in *awk* en veel andere talen is dat de indices van *awk* arrays beginnen bij 1.

8.5 Reguliere expressies

Vaak wil je gaan zoeken naar een string die op een andere string lijkt, maar niet identiek hetzelfde is. Dit kan worden bereikt door het gebruik van reguliere expressies. Een reguliere expressie is een string die metakarakters bevat (zie tabel 8.6 en hoofdstuk 5 voor meer informatie).

Reguliere expressies zijn zeer krachtig in het verwerken van teksten en worden in Unices veel gebruikt. Deze cursustekst geeft slechts een inleiding, aangezien dit een zeer uitgebreid onderwerp is en volledige boeken² zijn geschreven over het gebruik van reguliere expressies.

Bijvoorbeeld: de string “linkermuisknop” en “rechtermuisknop” voldoen aan `/(linker|rechter)muisknop/`

De strings “muisknop”, “middelste muisknop” en “Linkermuisknop” voldoen echter niet. Deze twee eerste strings voldoen echter aan

`/(linker|rechter)?muisknop/`

²Een uitstekend boek is *Mastering regular expressions* (Jeffrey Field, O’Reilly).

Metakarakter	Komt overeen met
\	escape karakter: plaats dit voor een metakarakter dat je letterlijk wilt gebruiken
^	begin van een string
\$	einde van een string
/^\$/	een lege lijn
.	een willekeurig karakter
[ABC]	A, B of C
[A-Ca-c]	A,B,C, a, b of c
[^ABC]	ieder karakter, behalve A, B en C
Stoel—Tafel	“Stoel” of “Tafel”
[ABC][DEF]	A,B of C, gevolgd door D, E of F
[ABC]*	0 of meerdere opeenvolgingen van A, B of C
[ABC]+	1 of meerdere opeenvolgingen van A, B of C
[ABC]?	A, B, C of niets
()	Combineert reguliere expressies

Tabel 8.6: Metakaracters

Omdat `?` aangeeft dat (`linker|rechter`) ten hoogste éénmaal mag voorkomen. De volledige string moet echter niet voldoen aan de reguliere expressie. Het is voldoende dat een deel van de string voldoet aan de (volledige!) reguliere expressie. Daarom voldoen bovenstaande strings aan

```
/muisknop/
```

Stel dat we nu willen dat alleen “muisknop” voldoet aan de reguliere expressie, dan moeten we aangeven dat de string begint met “muisknop”:

```
/^muisknop/
```

Als we in het voorbeeld password file willen controleren welke gebruikers `csch` of `tcsh` als shell hebben, moeten we in principe enkel controleren welke velden overeenkomen met “`csch`”:

```
BEGIN { FS = ":" }
/csh/ { print $5 }
```

De uitvoer van dit script geeft dan:

```
Chris Raymaekers
John Doe
Peter Icshello
```

De gebruikers “Chris Raymaekers” (`tcsh`) en “John Doe” (`csch`) voldoen hier inderdaad aan, maar “Peter Icshello” heeft `sh` als shell. De “`csch`” in zijn naam komt overeen met de reguliere expressie. Er zijn 2 manieren waarop we dit kunnen oplossen:

```
BEGIN { FS = ":" }
/(\/bin\/)t?(csch)/ { print $5 }
BEGIN { FS = ":" }
$NF ~ /csh/ { print $5 }
```

De eerste oplossing controleert of `/bin/csch` of `/bin/tcsch` in de regel voorkomt. De haakjes zijn niet nodig, maar werden toegepast om de oplossing iets leesbaarder te maken. De tweede oplossing controleert enkel of `csch` in het laatste veld voorkomt. Alhoewel beide oplossing correct zijn, is de tweede oplossing beter:

- De tweede oplossing controleert enkel het veld dat relevant is.

- De tweede oplossing maakt gebruik van een eenvoudigere reguliere expressie en wordt daarom sneller uitgevoerd.
- De tweede oplossing is beter leesbaar en daarom gemakkelijker om uit te breiden en te onderhouden.

8.6 Conditionele expressie

awk kent ook constructies die je in de meeste scripting en programmeertalen tegenkomt, zoals *if*.

Stel dat we een databank file (genaamd “bestelling”) van bestellingen hebben die er als volgt uitziet:

```
Computer:50000:1:2
```

```
Boek:1000:4:1
```

Het eerste veld geeft een beschrijving van het artikel, het tweede geeft de eenheidsprijs zonder BTW, het derde veld geeft aan hoeveel artikels werden besteld, het laatste veld bevat een code die aangeeft hoeveel BTW moet worden aangerekend. Het volgende script (listing 8.6, genaamd “factuur.awk”) maakt een factuur van deze file.

Listing 8.6: factuur.awk

```
BEGIN {
# de velden worden gescheiden door :
  FS = ":"
# print de hoofding
  printf( "%-15s | %-13s | %-11s | %-7s | %-7s\n", "Artikel", "Eenheidsprijs",
    "Hoeveelheid", "Prijs", "BTW inc")
  printf( "-----")
  printf( "-----\n")
}

{
# totale prijs voor dit artikel
  prijs = $2 * $3
# tel dit op bij het volledige totaal
  totaal += prijs;
# bereken de BTW
  if ( $4 == 1 )
    btw = prijs * 1.15 #15%
  else
    btw = prijs * 1.21 #21%
# tel dit op bij het volledige totaal
  totaalbtw += btw
# schrijf het rapport uit
  printf( "%-15s | %13i | %11i | %7i | %7i\n", $1, $2, $3, prijs, btw )
}

END {
  printf( "-----")
  printf( "-----\n")
# print de totalen
  printf( "%-18s: %6i BF (%7.2f euro)\n", "Totaal zonder BTW", totaal,
```

```

    totaal / 40.3399 )
    printf( "%-18s: %6i BF (%7.2f euro)\n", "Totaal met BTW", totaalbtw,
    totaalbtw / 40.3399 )
}

```

Het resultaat van de uitvoer is dan:

Artikel	Eenheidsprijs	Hoeveelheid	Prijs	BTW inc
Computer	50000	1	50000	60500
Boek	1000	4	4000	4600

Totaal zonder BTW :	54000 BF	(1338.63 euro)		
Totaal met BTW :	65100 BF	(1613.79 euro)		

Met behulp van het programma `sort` kunnen we ervoor zorgen dat de artikels in alfabetische volgorde staan:

```
sort bestelling | awk -f factuur.awk
```

Het resultaat van deze oproep is:

Artikel	Eenheidsprijs	Hoeveelheid	Prijs	BTW inc
Boek	1000	4	4000	4600
Computer	50000	1	50000	60500

Totaal zonder BTW :	54000 BF	(1338.63 euro)		
Totaal met BTW :	65100 BF	(1613.79 euro)		

De getallen die bij de format specifiers van `printf` staan, geven de (minimale) breedte van de verschillende velden aan. De velden worden standaard rechts uitgelijnd; een minteken geeft aan dat een veld links moet worden uitgelijnd. `7.2f` betekent dat 7 karakters moeten worden voorzien voor het uitschrijven van een reëel getal, waarvan 1 karakter voor de decimale punt en 2 karakters voor de cijfers na het decimale punt.

`awk` kent ook een conditionele operator: de `if` constructie van het vorige voorbeeld kan afgekort worden tot:

```
btw = ( $4 == 1 ) ? prijs * 1.15 : prijs * 1.21
```

8.7 Iteraties

`awk` kent ook enkele lus-structuren:

```

for ( initialisatie ; test ; increment )
    ...
for ( teller in array )
    ...
while ( test )
    ...

```

De eerste en derde iteratie constructie ben je al in Java en JavaScript tegengekomen. De tweede constructie is een constructie die in sommige scripting talen bestaat en kan gebruikt worden om alle waarden in een array te doorlopen (zie ook paragraaf 8.4).

Met behulp van een iteratie kan code die de horizontale lijn uitschrijft in het vorige voorbeeld geschreven worden als:

```

for ( i = 0 ; i < 65 ; i++ )
    printf( "-" )
printf( "\n" )

```

8.8 Functies

Zoals de meeste talen kent *awk* functies om veel gebruikte code te groeperen.

In het voorbeeld van paragraaf 8.6 wordt tweemaal een horizontale lijn getekend. We kunnen hiervoor een functie definiëren.

```
function printline( count )
{
    for ( i = 0 ; i < count ; i++ )
        printf( "-" )
    printf( "\n" )
}
```

De functie kan dan op de volgende manier worden opgeroepen in het script:

```
printline( 65 )
```

Met behulp van het `return` commando kan een functie een waarde teruggeven.

8.9 Voorbeeld

Deze sectie geeft nog een extra voorbeeld, waarin enkele constructies worden herhaald. Ook wordt extra informatie gegeven over de werking van arrays.

Stel dat op een computer een log file staat van de volgende vorm:

```
boot:disk2 failed
network:telnet:connection refused from beezer.mysite.be
network:ftp:connection refused from beezer.mysite.be
login:John:bad password
network:telnet:connection refused from www.mysite.be
```

De systeembeheerder wil statistieken maken van de log file. Er moet bijgehouden hoeveel “boot” (opstart) errors, hoeveel “login” errors en hoeveel “network” errors zijn opgetreden. Verder moeten statistieken worden bijgehouden over de oorzaken van deze errors. Het script houdt ook bij hoeveel onbekende fouten er zijn opgetreden, zodat het script kan worden aangepast als een nieuw soort fout optreedt.

Listing 8.7: Statistieken van een log file

```
BEGIN {
    FS = ":"
}

$1 == "boot"{ #boot error
    boot++;
    known++
}

$1 == "login"{ #login error
    login++;
    logins[$2]++; #user who caused the error
    known++
}

$1 == "network"{ #network error
    network++;
    services[$2]++ #service that reported the error
```

```

    split( $NF, messages, " ");
    sites[ messages[4] ]++ #site that caused the error
    known++
}

{
    print $0 #print all errors
    total++
}

END {
    printf( "\nSummary\n")
    print ( "-----")
    printf( "Boot: %i error(s)\n", boot )

    printf( "Login: %i error(s)\n", login )
    for ( user in logins ) #print login errors
        printf( " %s: %i bad login(s)\n", user, logins[ user ] )

    printf( "Network: %i error(s)\n", network )
    for ( service in services ) #print network errors
        printf( " %s: %i connection(s) refused\n", service, services[ service ] )
    for ( site in sites ) #print network errors
        printf( " %i connection(s) refused from %s\n", sites[ site ], site )

    if ( total - known > 0 ) #there are unknown errors
        printf( "%i unknown error(s)\n", total - known )
}

```

Merk op dat in tegenstelling tot programmeertalen *awk* willekeurige waarden kan gebruiken als index van een array (dit is typische voor scripting talen zoals *awk* en *perl*).

Het uitvoeren van dit script geeft het volgende resultaat.

```

boot:disk2 failed
network:telnet:connection refused from beezer.mysite.be
network:ftp:connection refused from beezer.mysite.be
login:John:bad password
network:telnet:connection refused from www.mysite.be

```

Summary

```

Boot: 1 error(s)
Login: 1 error(s)
    John: 1 bad login(s)
Network: 3 error(s)
    telnet: 2 connection(s) refused
    ftp: 1 connection(s) refused
    2 connection(s) refused from beezer.mysite.be
    1 connection(s) refused from www.mysite.be

```

Hoofdstuk 9

Programmeren onder Linux

9.1 Beschikbare middelen

Op zowat elk Linux systeem is er een **GNU C compiler** aanwezig, namelijk *gcc*. Met deze compiler kan men elk programma dat correct geschreven is in ANSI C compileren. De volgende lijst geeft een overzicht van andere programma's om software mee te schrijven:

ar om bibliotheek bestanden te creëren;

as om object bestanden te creëren (assembler);

bison om parsing tabellen te genereren;

flex lexical analyzer;

cpp C code preprocessor;

ld link editor (dynamisch libraries aanmaken);

make om programma's mee te creëren, hercreëren, updaten en onderhouden. zie ook appendix **B**;

patch om patches op broncode toe te passen;

gdb GNU debugger;

9.2 De GNU C Compiler

We zullen ons enkel concentreren op het gebruik van *gcc* en *g++*, omdat deze de meest gebruikte compilers zijn op Linux systemen. *gcc* staat voor Gnu C Compiler en *g++* is de GNU C++ compiler. Een volledig overzicht van alle opties (switches) die bruikbaar zijn met *gcc* zou ons te ver brengen, daarom zullen we ook enkel de meest relevante opties verduidelijken en aangeven.

Om het versienummer van de compiler te weten te komen kan men de switch `-v` gebruiken: `gcc -v`. De output ziet er dan als volgt uit, afhankelijk van de versie van *gcc* en de installatie:

```
Reading specs from /usr/lib/gcc-lib/i586-mandrake-linux/2.95.3/specs
gcc version 2.95.3 19991030 (prerelease)
```

Met de `-o` switch kan je aangeven welke output file het moet gebruiken om het programma weg te schrijven, bijv.: `gcc -o disaster.out disaster.c` compileert de file `disaster.c` en genereert de output file `disaster.out` van de gecompileerde code. Stel dat we het zeer eenvoudige programma `hello.c` in listing **9.1** zouden compileren met `gcc hello.c`, zonder de `-o` switch te gebruiken. Dan genereert *gcc* standaard een output file `a.out` die men kan uitvoeren door in

de output directory `./a.out` in te typen. Indien we nu de broncode gecompileerd hadden door `gcc -o hello hello.c`, dan werd het uitvoerbare bestand `hello` gegenereerd, wat we kunnen uitvoeren door `./hello` in te drukken.

Listing 9.1: “hello world” voorbeeld programma

```
/* hello.c */
int main()
{
    printf("hello world\n");
}

```

Indien de standaard foutmeldingen onvoldoende blijken kan gebruik gemaakt worden van de switch `-Wall`. Dit zorgt ervoor dat er zeer uitgebreide foutmeldingen gegenereerd worden, die meer informatie bevatten dan de standaard foutmeldingen.

Eigenaardig genoeg is `gcc` niet de compiler zelf, maar wel het deel dat men kan bedienen vanop de commandline. In feite zijn de C compiler en de C++ compiler (`g++`) geïntegreerd in een enkele tool, namelijk `gcc`. Toch is het aangeraden om `g++` te gebruiken voor expliciete C++ compilatie.

Hier is een lijstje van relevante switches voor `gcc`:

<code>-o file</code>	plaats de output in <code>file</code>
<code>-ansi</code>	ondersteun al de ANSI standaard C programma's
<code>-include file</code>	verwerk <code>file</code> eerst, en daarna pas het gewone input bestand
<code>-static</code>	verbiedt de compiler dynamisch linken toe te staan
<code>-shared</code>	zorgt ervoor dat de compiler een gedeelde bibliotheek genereert (zie ook sectie 9.3)
<code>-Idir</code>	voeg directory <code>dir</code> toe om erin naar include files te zoeken
<code>-W</code>	print extra warnings
<code>-Wall</code>	Laat <i>zoveel mogelijk</i> warnings zien
<code>-g</code>	produceer debug informatie, handig voor <code>gdb</code> (kan ook met <code>-ggdb</code>)
<code>-O</code>	optimaliseer de output: probeert de grootte te beperken en uitvoeringssnelheid te maximaliseren

opgave Schrijf zelf een programma, implementeer bijvoorbeeld uw favoriete sorteer algoritme, met behulp van `gcc`.

9.3 Bibliotheken

Deze sectie geeft een beknopte inleiding tot het programmeren met bibliotheken in Linux en een opsomming van de voordelen ervan bij het gebruik in eigen applicaties. Een bibliotheek is een bestand dat code en data bevat en als doel heeft later geïntegreerd te worden in, of gelinkt te worden aan een applicatie. Het gebruik van bibliotheken kan ervoor zorgen dat applicaties meer uit aparte modules bestaan, sneller te compileren zijn en bovendien gemakkelijker up-to-date gehouden kunnen worden. Bibliotheken stimuleren ook herbruikbaarheid: andere applicaties kunnen er ook gebruik van maken, zonder dat de bibliotheek gehercompileerd hoeft te worden.

Indien het tijdstip waarop een bibliotheek wordt geladen als criterium wordt genomen om ze op te delen, kunnen we drie soorten onderscheiden: *statische bibliotheken* (static libraries), *gedeelde bibliotheken* (shared libraries) en *dynamisch geladen bibliotheken* (DL libraries). Verder wordt er ook een onderscheid gemaakt tussen verschillende formaten. We hanteren hier echter enkel het *ELF formaat*, het Executable and Linking Format, omdat dit tegenwoordig het meest gebruikt wordt. Voor een vollediger overzicht en meer voorbeelden verwijzen we naar [Whe00].

[kleine uitleg
ELF toevoegen]

9.3.1 Statische bibliotheken

Een statische bibliotheek bestaat uit een collectie van object files en wordt volledig in de executable gekopieerd tijdens het compileren van de applicatie. Programmeurs kunnen de bibliotheek aan hun applicaties linken zonder dat deze bibliotheek gehercompileerd moet worden, iets wat behoorlijk wat tijd uitspaart. Ze zijn ook bruikbaar indien men andere gebruikers wil toelaten hun applicatie aan jouw bibliotheek te linken zonder dat ze beschikken over de broncode ervan. Een laatste voordeel is dat ze in theorie bij de uitvoer iets sneller zijn dan de andere soorten bibliotheken. In de praktijk blijkt dit echter zelden het geval te zijn wegens allerlei andere complicaties.

Ondanks deze voordelen worden statische bibliotheken niet zoveel meer gebruikt als vroeger. De voordelen van gedeelde bibliotheken overheersen namelijk, zoals in de volgende sectie zal blijken. Het volgende voorbeeld toont hoe een statische bibliotheek wordt aangemaakt aan de hand van enkele object bestanden die met het programma *ar* (sectie 9.1) tot één archief gecombineerd worden :

```
ar rcs mijn_bibliotheek.a bestand1.o bestand2.o
```

De opties *r* en *c* staan respectievelijk voor het toevoegen van object bestanden (insert with replacement), het aanmaken van een archief (create) en het creëren of overschrijven van een object bestand index.

9.3.2 Gedeelde bibliotheken

Gedeelde bibliotheken worden geladen zodra een gelinkte applicatie wordt opgestart. De eigenlijke bibliotheekcode wordt dus niet in de binary meegecompileerd maar slechts als verwijzing in de applicatie bewaard. Gedeelde bibliotheken hebben de volgende interessante eigenschappen:

- Een bibliotheek kan bijgewerkt of aangevuld worden terwijl applicaties die *niet-backwards-compatibele versies van de bibliotheek* gebruiken nog steeds ondersteund worden. Dit wordt mogelijk gemaakt door de manier waarop bibliotheken beheerd worden, namelijk door het onderscheid te maken tussen verschillende namen van een bibliotheek, zoals verder wordt uitgelegd. De verschillende versies kunnen dus naast elkaar blijven bestaan.
- Een bibliotheek, of zelfs specifieke functies in die bibliotheek, kunnen overriden worden bij de uitvoer van een gelinkte applicatie.
- Dit alles is mogelijk terwijl andere applicaties kunnen en gebruik maken van bestaande bibliotheken.

Om deze eigenschappen te kunnen ondersteunen is men verplicht een aantal richtlijnen te volgen. Het is noodzakelijk het verschil te kennen tussen de verschillende namen van de bibliotheek, en verder moet men ook enige notie hebben van waar de bibliotheek in kwestie uiteindelijk geplaatst moet worden.

Elke gedeelde bibliotheek heeft een drietal namen:

- De **soname** begint met de prefix “lib”, gevolgd door de naam van de bibliotheek, de letters “.so”, en afgesloten door een punt en de versienummer die wordt verhoogd telkens de interface van de bibliotheek gewijzigd werd. Meestal is de *soname* een symbolische link naar de *echte naam*.
- Met de **echte naam** wordt het bestand aangeduid dat de eigenlijke bibliotheekcode bevat. Het formaat voegt aan de *soname* een punt, het minor nummer, nog een punt, en het optionele release nummer. De twee extra nummers laten gebruikers weten wat de exacte versie is van een geïnstalleerde bibliotheek, zodat ook meerdere versies geïnstalleerd kunnen worden zonder verwarring te creëren.

- De **linker naam** wordt gebruikt door de compiler wanneer deze de bibliotheek wil gebruiken. Ze wordt gevormd door de *soname* zonder enige versienummer. Meestal is dit ook een symbolische link naar een *soname* of naar een *echte naam*.

Voordat getoond wordt hoe een gedeelde library gemaakt kan worden, eerst nog een paar opmerkingen over de compatibiliteit tussen verschillende versies van een bibliotheek. Zoals eerder vermeld moet de “soname” aangepast worden indien een nieuwere versie van de bibliotheek binair-incompatibel is met de oudere versie. [Whe00] somt een aantal oorzaken op, voor zowel C als C++, waardoor de ABI (Application Binary Interface) van een bibliotheek verandert en een bibliotheek dus incompatibel wordt met oudere versies.

Als eenvoudig voorbeeld worden eerst een tweetal object bestanden gecreëerd (bestand1.o en bestand2.o), en daarna een gedeelde bibliotheek die beiden bevat.

```
g++ -fPIC -g -c -Wall bestand1.cpp
g++ -fPIC -g -c -Wall bestand2.cpp
g++ -shared -Wl,-soname,libmijn_bib.so.1 -o libmijn_bib.so.1.0.1
    bestand1.o bestand2.o -lc
```

Naast de gekende parameters gebruikten we `-fPIC`, een vereiste bij gedeelde bibliotheken om “Position Independent Code” te genereren, `-c` om object bestanden te creëren en `-shared` om aan te geven dat we een gedeelde bibliotheek willen. `-Wl` laat toe enkele parameters aan de linker door te geven, gescheiden door komma’s zoals hier de optie `-soname` met de naam die we gekozen hebben. De parameter `-lc` lijkt nieuw, maar geeft aan dat de bibliotheek `libc` meegelinkt moet worden.

Als nu nog de nodige symbolische links aangemaakt worden voor de *soname* en de *linker naam* kan de bibliotheek gelinkt worden aan een applicatie:

```
g++ [...] -lmijn_bib
```

Zodra echter de bibliotheek door meerdere applicaties gebruikt moet kunnen worden, is het noodzakelijk dat ze ergens op een centrale plaats staat. Volgens twee standaarden (de *GNU standard* en de *Filesystem Hierarchy Standard*¹) is `/usr/local/lib` de aangewezen plaats voor bibliotheken die mogelijk nog fouten bevatten, en in `/usr/lib` zouden de latere, stabielere versies moeten komen. Het programma `ldconfig` zorgt ervoor dat een verwijzing naar de bibliotheek in een speciaal bestand wordt toegevoegd zodat er snellere toegang tot de bibliotheek mogelijk is (zie verder).

9.3.3 Dynamisch geladen bibliotheken

Dynamisch geladen bibliotheken of DL bibliotheken worden geladen op eender welk tijdstip dat verschilt van het moment dat een applicatie wordt opgestart. Dit biedt de mogelijkheid tot het creëren van plug-ins of modules. Het tijdstip waarop de module geladen wordt kan dan immers uitgesteld worden totdat de inhoud ervan voor het eerst gebruikt wordt.

Het formaat van een DL bibliotheek verschilt niet van dat van statische en gedeelde bibliotheken, enkel het tijdstip waarop ze geladen wordt. Een speciale API zorgt voor het openen van een DL bibliotheek, het opzoeken van symbolen, het afhandelen van fouten en het sluiten van de bibliotheek. Deze interface is echter niet op elk platform hetzelfde [Whe00], we behandelen hier dus enkel de interface ondersteund door Linux.

De DL bibliotheek API (gedefinieerd in de `dlfcn.h`) biedt de volgende vier functies:

¹<http://www.pathname.com/fhs>

```
void* dlopen(const char* filename, int flag); // Opent een bibliotheek.
void* dlsym(void* handle, char* symbol);     // Geeft de waarde van een symbool.
char* dlerror();                            // Een beschrijving van de foutmelding.
int dlclose(void* handle);                  // Sluit de bibliotheek.
```

De parameter “flag” in de functie *dlopen* kan drie waarden aannemen: `RTLD_LAZY` zorgt ervoor dat symbolen enkel *resolved* worden zodra ze worden aangeroepen vanuit de code van de bibliotheek. `RTLD_NOW` verlegt dit tijdstip tot vóór de functie *dlopen()* is afgelopen. Tenslotte, `RTLD_GLOBAL` mag optioneel ge-or’d worden met de twee vorige waarden en geeft aan dat alle externe symbolen die in de bibliotheek gedefinieerd werden ter beschikking gesteld worden van bibliotheken die achteraf nog geladen worden.

Listing 9.2 haalt een pointer naar een functie uit de bibliotheek “libm” en gebruikt ze om de cosinus van 2.0 te berekenen (voor meer uitleg over pointers naar functies, zie appendix A).

Listing 9.2: `dllib.c`

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <dlfcn.h>
4
5 int main(int argc, char **argv) {
6
7     void *handle;
8     double (*cosine)(double);
9     char *error;
10
11     handle = dlopen ("/lib/libm.so", RTLD_LAZY);
12     if( !handle) {
13         fputs(dlerror(), stderr);
14         exit(1);
15     }
16
17     cosine = dlsym(handle, "cos");
18     if((error = dlerror()) != NULL) {
19         fputs(error, stderr);
20         exit(1);
21     }
22
23     printf ("%f\n", (*cosine)(2.0));
24     dlclose(handle);
25
26     return 0;
27 }
```

9.3.4 Relevante utilities

Het gereedschap dat u nodig heeft om aan de slag te gaan en je eigen libraries te creëren bestaat uit de volgende utilities: `ldconfig`, `ldd`, `nm`. Deze worden meestal standaard mee geïnstalleerd bij je Linux distributie.

Hoofdstuk 10

Een device driver in Linux

Alhoewel de laatste stabiele Linux kernel versie 2.4.18 is, wordt in dit hoofdstuk nog kernel 2.2.x gebruikt.

10.1 Randapparaten

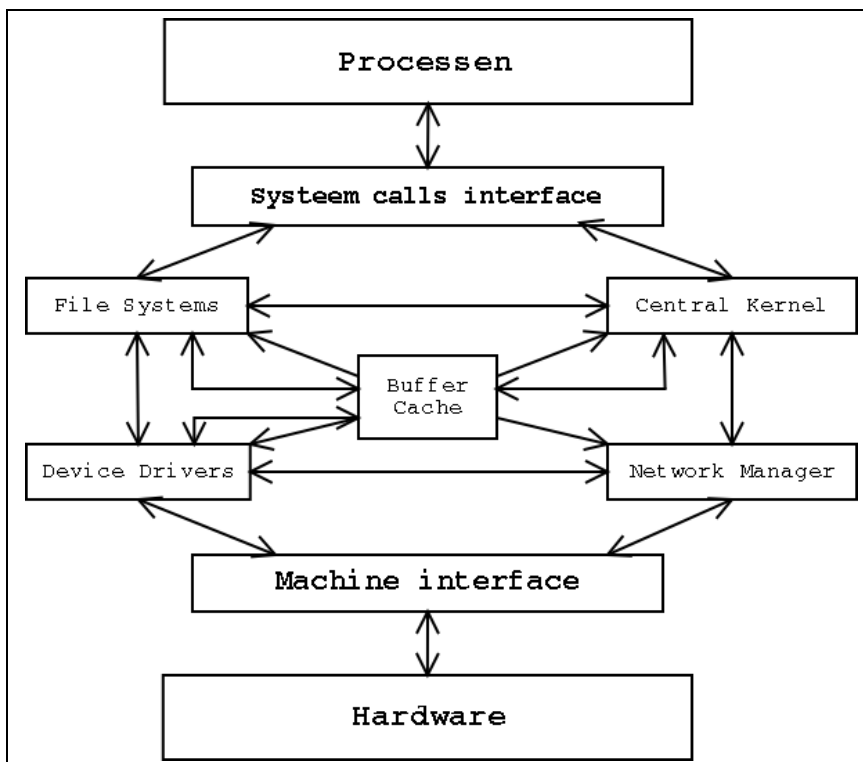
De kern van een besturingssysteem is de **kernel**¹. De kernel is een stuk code dat steeds in het geheugen aanwezig is en verantwoordelijk is voor procesbeheer, geheugenbeheer, bestandsbeheer en hardwarebeheer. Om het systeem toe te laten met randapparatuur te communiceren gebruikt het device drivers. Dit kunnen modules zijn die in de kernel geladen worden of code die reeds met de kernel mee gecompileerd is en die toelaat om gegevens van de randapparatuur (de zgn. devices) te lezen en/of ernaar te schrijven. Device drivers zijn de “software interfaces” om met een bepaald randapparaat te werken. Let wel: dit randapparaat hoeft *niet* een fysisch hardware apparaat te zijn; het kan bijvoorbeeld ook een random data generator of een virtuele console zijn. We hebben twee soorten devices: *character* devices en *block* devices. Character devices lezen de data enkel sequentieel en worden niet gebufferd. Block devices bieden random access aan en hun data wordt wel gebufferd. Een harde schijf, CDROMS, . . . zijn voorbeelden van block devices. Een console, de geluidskaart en de seriële poort zijn voorbeelden van character devices. In feite wordt er nog een onderscheid gemaakt met de network devices, zodat er eigenlijk 3 soorten devices zijn. Network devices werken dan ook op een totaal andere manier dan character en block devices. Op netwerk devices wordt in deze cursus echter niet dieper ingegaan.

De kernel heeft eigenlijk twee belangrijke interfaces: een interface voor de applicaties en een interface voor de hardware. Op deze manier kan het een uniforme toegang tot processen en hardware aanbieden wat de uitbreidbaarheid en onderhoudbaarheid van het besturingssysteem in grote mate ten goede komt. Een afbeelding van de kernel architectuur met de twee interfaces vind je in figuur 10.1.

Al de devices waarvan je Linux systeem gebruik kan maken staan opgesomd in de `/dev` directory, zoals reeds vermeld in sectie 3.1. In listing 10.1 kan je zien dat de letter *b* op het begin van een regel erop wijst dat we met een block device te maken hebben (in dit geval is dat de floppy) en dat de letter *c* staat voor een character device (in dit geval de DSP² van de geluidskaart). De nummers die volgen op de owner en group zijn de major en minor nummer. `/dev/fd0` heeft major 2 en minor 0 en `/dev/dsp` heeft major 14 en minor 3. De major nummer duidt op een bepaald randapparaat en de minor op een instantie hiervan. We kunnen ook zelf een file aanmaken waar later een randapparaat kan gemapt worden met behulp van het `mknod` commando. Dit commando heeft de syntax: `mknod name type major minor`. Indien we de file `/dev/fd0` zelf zouden aanmaken, dan zouden we het commando `mknod /dev/fd0 b 2 0` moeten uitvoeren.

¹De laatste informatie over de Linux kernel is te vinden op <http://www.kernelnotes.org/>

²Digital Signal Processor



Figuur 10.1: Interne kernel architectuur

<i>type device en protecties</i>				<i>major nummer device driver</i>	<i>minor nummer id apparaat</i>		
brw-rw----	1	root	disk	3,	0	May 5 1998	/dev/hda
brw-rw----	1	root	disk	3,	1	May 5 1998	/dev/hda1
brw-rw----	1	root	disk	3,	64	May 5 1998	/dev/hdb
crw-rw--w-	1	root	sys	14,	4	May 5 1998	/dev/audio
crw-rw--w-	1	root	sys	14,	20	May 5 1998	/dev/audio1

Tabel 10.1: Gestructureerde kijk op /dev listing

Listing 10.1: Listing van de /dev directory

```

[kris@localhost course]$ls -l /dev/
...
brw-rw----  1 kris   floppy   2,  0 Sep 27 12:31 /dev/fd0
...
crw-----  1 kris   audio    14,  3 Sep 27 12:31 /dev/dsp
    
```

De betekenis van de /dev listing wordt overzichtelijk voorgesteld in de tabel 10.1

10.2 Programmeren voor de kernel

Er is een duidelijk verschil in aanpak wanneer we voor de kernel moeten programmeren. Het besturingssysteem Linux kent twee verschillende modes: *user-mode* en *supervisor-mode*. De eerste mode is de “normale” mode; de mode waarin we normaal gezien als gebruiker van het systeem werken. De user-mode heeft meer restricties in gebruik van stukken geheugen en aanspreken van de hardware bijvoorbeeld. De tweede mode, supervisor-mode, is de mode waarin we in dit hoofdstuk gaan werken. De supervisor-mode laat ons toe heel het geheugen te

gebruiken, dus ook in kernel-space te werken en rechtstreeks hardware aan te spreken. Vooraleer we ons op het harde coderen gooien nog een opmerking: het is een algemene vuistregel dat wanneer men iets niet in de kernel-space hoeft te programmeren, men dat gewoon *niet* doet. Het voorbeeldje gegeven in sectie 10.3.1 is louter didactisch, en doorbreekt deze vuistregel.

10.3 Een eerste kernel module

10.3.1 Hello world

We beginnen aan onze eerste kernel module: “Hello world” voor de kernel. Listing 10.2 geeft een voorbeeldje van de module.

Listing 10.2: modhelloworld.c

```

1  /* modhelloworld.c; a kernel module version */
2
3  #include <linux/kernel.h>
4  #include <linux/module.h>
5
6  #if CONFIG_MODVERSIONS==1
7  #define MODVERSIONS
8  #include <linux/modversions.h>
9  #endif
10
11 int init_module()
12 {
13     /* printk is the kernel version of printf */
14     printk("Hello world - this is your kernel speaking...\n");
15     return 0;
16 }
17
18 void cleanup_module()
19 {
20     printk("bye bye\n");
21 }

```

De twee include statements zijn minimale include statements voor module-code. `kernel.h` en `module.h` bevatten define statements en andere include statements die de nodige parameters at compile time aanreiken om een geschikte module voor de kernel te maken.

Wat ongetwijfeld eerst opvalt is het gebruik van `printk` in plaats van `printf`. `printk` schrijft een string weg naar een kernel console³ en kan geen floating point getallen wegschrijven. Het wordt trouwens (zeer) sterk afgeraden om met floating point getallen te werken wanneer men voor de kernel programmeert: de floating point staat kan verloren gaan omdat de kernel de staat van een floating point getal niet opslaat.

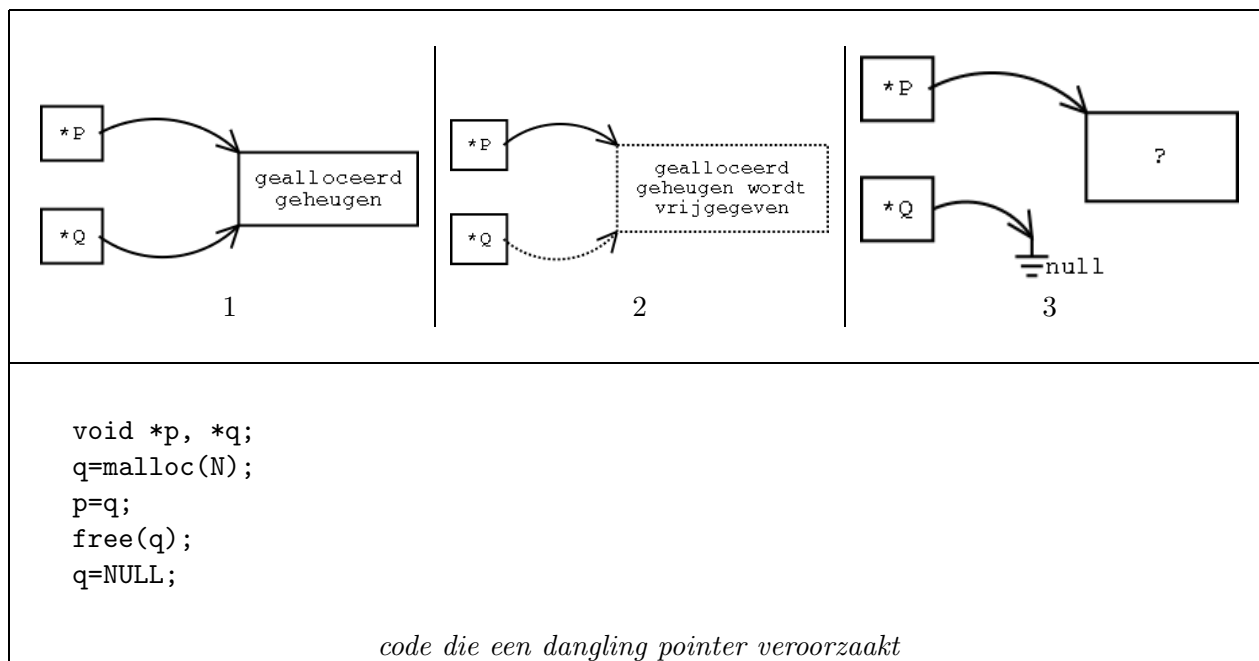
Bovenaan zie je ook een `#if...#endif` statement waarin gekeken wordt of `CONFIG_MODVERSIONS` reeds gedefinieerd is. Als dit zo is dan definiëren we `MODVERSIONS` en moeten we `modversions.h` includen. We hadden dit natuurlijk ook kunnen schrijven als:

```

#if defined(CONFIG_MODVERSIONS)
#define MODVERSIONS

```

³Eigenlijk kent `printk` verschillende “niveaus”. Als `printk` blijkbaar niets uitschrijft probeer dan “<7>” vooraan bij in de string te zetten, werkt dat nog niet probeer dan “<6>” enzoverder tot en met “<0>”, dit is het meest kritische niveau van `printk`. Soms schrijft `printk` ook niet naar de console, maar naar een file, gecontroleerd door `syslogd` en aangegeven in `/etc/syslog.conf`



Figuur 10.2: Ontstaan van een dangling pointer

```

#include <linux/modversions.h>
#endif

```

Later zullen we zien dat er eigenlijk ook met de kernel versie moet rekening gehouden worden.

De twee functies tenslotte zijn bijna zelf-verklarend: `init_module` wordt door de kernel aangeroepen als de module in de kernel geladen wordt. `cleanup_module` wordt aangeroepen als de module uit de kernel gehaald wordt om alles op te kuisen. Zeker als men voor kernel programmeert moeten we zeer voorzichtig omspringen met geheugengebruik. Een dangling pointer (zie figuur 10.2; stap 1 laat de pointers naar het geheugen wijzen, in stap 2 wordt het geheugen waar `q` naar wijst verwijderd, maar `p` blijft in stap 3 naar dezelfde geheugenplaats wijzen waar nu mogelijk rommel in staat.) is een onvergeefbare fout voor een kernel programmeur, want dit brengt een zeer groot risico voor een kernel crash met zich mee! De kleinste fout die men maakt in code bestemd voor de kernel heeft meestal het vastlopen van de computer als resultaat. Een crash van de kernel is moeilijk op te vangen en te ontleden. In plaats van een *core dump* zoals bij een crash van een gewone applicatie zal de kernel enkel een “Oops message” geven als het misloopt (dan weet je dat het hoogstwaarschijnlijk tijd is om de computer te herstarten). Het ontleden van zo een “Oops message” valt buiten deze cursus. Als de functie `init_module` als return value een 0 teruggeeft betekent dit dat de module succesvol geladen is.

Tenslotte moeten we ons programma nog gecompileerd krijgen. Om deze module te compileren gebruiken we `gcc`. De volgende opdrachtregel geeft aan hoe we de module moeten compileren:

```

gcc -Wall -D__KERNEL__ -DMODULE -I/usr/src/linux/include -O2
-c modhelloworld.c -o hello.o

```

Welke switch dient nu voor wat (-D betekent “define” en -I betekent “include directory”)?

-Wall : geeft aan dat de compiler *alle* warnings moet laten zien tijdens het compileren;

-D__KERNEL__ : geeft aan dat we voor de kernel aan het compileren zijn;

-DMODULE : geeft aan dat we een module aan het compileren zijn;

- I**/**usr/src/linux/include** : geeft aan dat de compiler in die directory voor de includes moet zoeken. Dit kan verschillend zijn voor verschillende distributies;
- O2** : een kernel module moet met een optimalisatie level van O2 of hoger gecompileerd worden omdat dit ervoor zorgt dat inlined functies geëxpandeerd worden in de kernel.

Als de compilatie gelukt is hebben we de module `hello.o`. Switch nu naar een console en log in als root. Je kan de module nu laden door `insmod hello.o` in te typen. Let op, dit moet in een console gebeuren omdat printk geen output naar X-terminals geeft. Als je `insmod hello.o` ingetypt hebt krijg je normaal het volgende te zien:

```
[root@localhost lists]# insmod hello.o
Hello world - this is your kernel speaking...
[root@localhost lists]#
```

Hoe kan je nu zien of de module wel degelijk geladen is? `/proc/modules` bevat een lijst van alle geladen modules, dus als we hier een cat van doen zien we of “hello” erbij is:

```
[root@localhost lists]# cat /proc/modules
hello                400      0 (unused)
tuner                2960     1 (autoclean)
bttv                 47952    0 (unused)
i2c-algo-bit         7696     1 [bttv]
...
vfat                 9408     1 (autoclean)
fat                  30432    1 (autoclean) [vfat]
ide-scsi             7664     0
reiserfs             128592   1
[root@localhost lists]#
```

Bovenaan het lijstje prijkt nu onze module. Om de module te verwijderen typ je `rmmmod hello` in. Dan krijg je normaal de boodschap van `cleanup_module` te zien:

```
[root@localhost lists]# rmmmod hello
bye bye
[root@localhost lists]#
```

10.3.2 Uitgebreider voorbeeld in het proc filesystem

Nu de fundamenten gelegd zijn, kunnen we een stapje verder gaan. We gaan een kernel module maken die een teller bijhoudt voor elke keer dat er informatie opgevraagd wordt over deze module. We zorgen ervoor dat de module ook een entry in de `proc` directory⁴ heeft zodanig dat we via het `cat` commando informatie kunnen opvragen. Bij het programmeren van deze module moeten we de input-output omdraaien. Wat voor de gebruiker van het systeem *data lezen* is, is voor de kernel *data schrijven* en omgekeerd. Dus als we gegevens van onze module opvragen is dat voor ons lezen, maar voor de module zal dat schrijven betekenen. Door middel van het `cat` commando *lezen* we dus de module.

Onze module, `proctext.c`, is afgebeeld in listing 10.3 (voor kernel 2.2, de code voor kernel 2.4 vind je in listing 10.4). Er is een extra macro definitie bijgekomen waarmee de kernel versie nummer kunnen berekenen. Dit is nodig vanwege de functie-aanroep `proc_register` in de functie `init_module`, vóór kernel versie 2.2 had deze een andere definitie en daar moet

⁴Het `/proc` filesystem is origineel gemaakt om makkelijk informatie over processen op te vragen. Tegenwoordig wordt het ook gebruikt om alle relevante informatie over de kernel op te vragen

rekening mee gehouden worden. Daarnaast zijn er ook nog wat include statements bijgekomen; `malloc.h` omdat we geheugen moeten gaan alloceren en `proc_fs.h` omdat we het proc filesystem gebruiken.

In de functie `init_module` registreren we dus onze module bij het `/proc` system. We geven daar een variabele van het type `proc_dir_entry` aan mee om wat extra informatie te voorzien:

```
struct proc_dir_entry Our_Proc_File=
{
0,          → lage inode nummer, zal automatisch ingevuld worden door
            proc_register[_dynamic]
6,          → lengte van de naam van de module
"teller",  → naam van de module
S_IFREG | S_IRUGO, → de mode van de module: type en permissies
1,          → aantal links naar de node
0 ,        → de uid van de module, meestal 0
0 ,        → de gid van de module, meestal 0
80,        → de grootte van de module, zoals aangegeven door ls
NULL,      → definieert de set van operaties, hier zijn dat er geen dus
            schrijven we NULL

procfile_read, → de functie die opgeroepen wordt om informatie van de mo-
            dule te verkrijgen

NULL,      → Dit altijd op NULL zetten
NULL, NULL, NULL, → Deze entries altijd op NULL zetten (beter is deze gewoon
            niet in te vullen)

NULL      → Dit altijd op NULL zetten (beter is deze gewoon niet in te
            vullen)
};
```

`S_IFREG | S_IRUGO` betekent dat het een gewone (regular) file is die kan gelezen worden door de eigenaar, de groep en iedereen daarbuiten. Voor meer informatie over deze structure (en andere) kan meestal ook de man-pagina geraadpleegd worden (`man proc_dir_entry`). De algemene definitie van de `proc_dir_entry` structure ziet er als volgt uit:

```
struct proc_dir_entry {
    unsigned short    low_ino;
    unsigned short    namelen;
    const char *      name;
    mode_t            mode;
    nlink_t           nlink;
    uid_t             uid;
    gid_t             gid;
    unsigned long     size;
    struct inode_operations *ops;
    int               (*get_info)(char *buffer, char **start, off_t offset,
                                int length, int unused);
    void              (*fill_inode)(struct inode *);
    struct proc_dir_entry *next, *parent, *subdir;
    void *            data;
};
```

Vanaf kernel 2.4 is het niet meer nodig om deze struct expliciet te gebruiken. Men kan met behulp van functie oproepen een entry in het proc filesystem aanmaken.

Een entry in het `/proc` system is in dit geval handig omdat we geen entry hebben in de `/dev` directory. Hiervoor zouden we met `mknod` eerst een entry moeten maken. Nu wordt er in de `/proc` directory automatisch een entry gemaakt als we de module laden. Merk op dat de module die we hier schrijven ook nog geen "echte" device driver is, het stuurt namelijk niets

aan. Om de entry in de /proc directory te voorzien, roepen we in de functie `init_module` `proc_register` aan als de kernel versie hoog genoeg is en de kernel deze functie ondersteunt, anders gebruiken we `proc_register_dynamic`, wat op hetzelfde neerkomt. Om de entry weer te verwijderen gebruiken we de functie `proc_unregister` in `cleanup_module`.

Listing 10.3: `proctext.c` (kernel 2.2)

```
1 #ifndef PROCTEXT
2 #define PROCTEXT
3 #endif
4
5 #include <linux/kernel.h>
6 #include <linux/errno.h>
7 #include <linux/module.h>
8
9
10 #if CONFIG_MODVERSIONS==1
11 #define MODVERSIONS
12 #include <linux/modversions.h>
13 #endif
14
15 #ifndef KERNEL_VERSION
16 #define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
17 #endif
18
19 #include <linux/config.h>
20 #include <linux/malloc.h>
21 #include <linux/sysctl.h>
22 #include <linux/swapctl.h>
23 #include <linux/proc_fs.h>
24
25 int procfile_read( char *buffer,
26                  char **buffer_location,
27                  off_t offset,
28                  int buffer_length,
29                  int zero)
30 {
31     int len;
32     static int count = 1;
33     static char my_buffer[80];
34
35     if(offset>0)
36         return 0;
37
38
39     len = sprintf(my_buffer,
40                 "You have asked me that already %d times!\n",
41                 count);
42     count++;
43     *buffer_location = my_buffer;
44     return len;
45 }
46
```

```

47
48 struct proc_dir_entry Our_Proc_File=
49 {
50     0,
51     6,
52     "teller",
53     S_IFREG | S_IRUGO,
54     1,
55     0 ,0,
56     80,
57     NULL,
58     procfile_read,
59     NULL
60 };
61
62
63 int init_module()
64 {
65     #if LINUX_VERSION_CODE > KERNEL_VERSION(2,2,0)
66     return proc_register(&proc_root, &Our_Proc_File);
67     #else
68     return proc_register_dynamic(&proc_root, &Our_Proc_File);
69     #endif
70 }
71
72
73 void cleanup_module()
74 {
75     proc_unregister(&proc_root, Our_Proc_File.low_ino);
76 }

```

Listing 10.4: proctext.c (kernel 2.4)

```

1  #ifndef PROCTEXT
2  #define PROCTEXT
3  #endif
4
5  #include <linux/kernel.h>
6  #include <linux/errno.h>
7  #include <linux/module.h>
8
9  #if CONFIG_MODVERSIONS==1
10 #define MODVERSIONS
11 #include <linux/modversions.h>
12 #endif
13
14 #ifndef KERNEL_VERSION
15 #define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
16 #endif
17
18 #include <linux/config.h>
19 #include <linux/malloc.h>

```

```

20 #include <linux/sysctl.h>
21 #include <linux/swapctl.h>
22 #include <linux/proc_fs.h>
23
24 int read_proc( char *buffer,
25               char **buffer_location,
26               off_t offset,
27               int buffer_length)
28 {
29     int len;
30     static int count = 1;
31     static char my_buffer[80];
32
33     if(offset>0)
34         return 0;
35
36     len = sprintf(my_buffer,
37                 "You have asked me that already %d times!\n",
38                 count);
39     count++;
40     *buffer_location = my_buffer;
41     return len;
42 }
43
44 int init_module()
45 {
46     create_proc_info_entry("teller", S_IFREG|S_IRUGO, &proc_root, read_proc);
47     return 0;
48 }
49
50
51
52 void cleanup_module()
53 {
54     remove_proc_entry("teller", &proc_root);
55 }

```

Wat gebeurt er nu als we de module inserteren en dan de opdracht `cat /proc/teller` uitvoeren? De module reageert hierop door de functie op te roepen die we in de `proc_dir_entry` structure hebben aangegeven hiervoor, namelijk `procfile_read`. De output die moet verschijnen als reactie op `cat /proc/teller` wordt aangewezen door `buffer_location`. `len` geeft aan hoeveel bytes er in feite gebruikt worden. De offset is altijd gelijk aan 0 in dit voorbeeldje, omdat we data niet in chunks (verschillende afgelijnde delen) maar in één keer versturen. Als de read functie opgeroepen wordt met een offset>0 geven we niets terug, omdat met een offset groter dan 0 een volgende chunk bedoeld wordt. Daar onze data niet zo groot is, is het makkelijk om die in één keer door te geven. Als deze echter wat groter wordt kunnen we zo de data wel in stukken versturen (en zo de processor beter delen met andere processen).

10.4 Devices als files

10.4.1 De API

Tot nu toe hebben we nog geen echte device drivers geschreven, maar enkel modules die in de kernel geladen worden. Als we een echte device driver willen schrijven moeten we er in Linux mee rekening houden dat lezen van en schrijven naar een randapparaat dezelfde software architectuur gebruikt als operaties op bestanden (dit wordt ook weerspiegeld in het normale gebruik van het besturingssysteem, zie ook sectie 3.1 en sectie 3.6). Dit wil zeggen dat de verzameling geldige functies om met bestanden te werken ook geldt voor device drivers. Bij het schrijven van zulk een device driver moeten we aangeven welke functies uit deze verzameling we gaan gebruiken. Al de mogelijke functies voor het werken met bestanden zijn samengepakt in een structure:

```
struct file_operations {
    loff_t      (*llseek)      (struct file *, loff_t, int);
    ssize_t    (*read)        (struct file *, char *, size_t, loff_t *);
    ssize_t    (*write)       (struct file *, const char *, size_t, loff_t);
    int        (*readdir)     (struct file *, void *, filldir_t);
    unsigned int(*poll)       (struct file *, struct poll_table_struct *);
    int        (*ioctl)       (struct inode *, struct file *,
                               unsigned int, unsigned long);
    int        (*mmap)        (struct file *, struct vm_area_struct);
    int        (*open)        (struct inode *, struct file *);
    int        (*flush)       (struct file *);
    int        (*release)     (struct inode *, struct file *);
    int        (*fsync)       (struct file *, struct entry *);
    int        (*fasync)      (int, struct file *, int);
    int        (*check_media_change)(kdev_t dev);
    int        (*revalidate)   (kdev_t dev);
    int        (*lock)        (struct file *, int, struct file_lock *);
};
```

De code voor de device driver declareert een variabele van het type `file_operations` en zet vervolgens alle functies die *niet* ondersteund zullen worden door de device driver op NULL. De functies die wel ondersteund worden door de device driver, worden als verwijzing naar de echte implementatie in de code van de device driver wel in de structure geplaatst. Let erop dat de structure *pointers naar functies* bevat; op deze manier is het mogelijk nieuwe device drivers te schrijven waarvan de kernel niet “op de hoogte” was. Door enkel verwijzingen in `file_operations` door te geven naar de functies die ter ondersteuning van de device driver geprogrammeerd zijn, is het niet nodig dat de code statisch met de kernel gelinkt was. De functionaliteiten die een device driver aanbiedt worden bepaald door een tabel met wijzers (pointers) naar functies die bijgehouden en doorgegeven wordt. Functies worden door middel van hun verwijzingen (die terug te vinden zijn in die tabel) opgeroepen en niet “statisch” via hun naam (anders zou deze reeds kernel compile-time moeten gekend zijn!). In appendix A is wat meer uitleg verschaft over pointers naar functies.

10.4.2 Gebruikte entries

De entries in de `file_operation` structure die voor ons het meest interessant zijn, zijn ongetwijfeld `read`, `write`, `open` en `close`. Op `ioctl` wordt kort nog teruggekomen in sectie 10.4.4. Voor de `read` en `write` functies moeten we oppassen of we nu in kernel space of in user space aan het lezen en/of schrijven zijn. Het besturingssysteem is hier zeer strikt in! We kunnen gebruik maken van de twee functies `copy_to_user` en `copy_from_user`, met de definities:

```
copy_to_user(void *to, void *from, unsigned long size)
copy_from_user(void *to, void *from, unsigned long size)
```

De oorsprong en het doel van de kopieer operatie worden aangegeven, samen met de grootte van de te kopiëren data. In feite zijn deze twee functies de “uitgebreide” versie van `get_user` en `put_user` die enkel geschikt zijn voor kleine hoeveelheden data (een enkele primitieve variabele) te kopiëren. De `copy_xx_user` is beter geschikt voor grotere hoeveelheden data.

De functie `copy_to_user` laat toe om data van de kernel space naar de user space te kopiëren. De functie `copy_from_user` doet het omgekeerde: het kopiëert data van de user space naar de kernel space. Indien de functies succesvol uitgevoerd worden zullen ze 0 teruggeven. Indien dit niet het geval is geven ze een niet nul waarde terug, en wil dit meestal zeggen dat de toegang tot de doel-memory space ontzegt is.

10.4.3 Module gebruik beschermen

Wanneer we gebruik maken van de device driver, bijvoorbeeld door deze te openen om ernaar te schrijven of ervan te lezen, mag er geen `rmmmod` op de module uitgevoerd worden. We mogen immers de module die voor de communicatie met het randapparaat instaat niet verwijderen terwijl we met het apparaat aan het praten zijn. Hiervoor zijn er twee handige defines in `include/linux/module.h` aanwezig, namelijk `MOD_INC_USE_COUNT` en `MOD_DEC_USE_COUNT`. Je kan de eerste in de open-operatie gebruiken en de tweede in de close-operatie. Hierdoor houdt de kernel een teller bij van het gebruik van de module en zorgt ervoor dat de module niet kan verwijderd worden als er iemand ervan gebruik maakt. `MOD_INC_USE_COUNT` verhoogt de gebruikersteller met één en `MOD_DEC_USE_COUNT` verlaagt de gebruikersteller met één.

10.4.4 I/O control: `ioctl`

Een zeer belangrijke entry in de structure is de `ioctl` verwijzing. `ioctl` laat toe om I/O-apparaten in te stellen zonder hiervoor nieuwe functies te hoeven schrijven. Meer zelfs; het laat toe instellingen van een I/O-apparaat te lezen en/of te veranderen terwijl de driver reeds in gebruik is. Dit is zeer handig voor device drivers die permanent in de kernel aanwezig zijn. Wij zullen hier echter geen gebruik meer van maken.

10.5 Een device driver voor de keyboard LEDs

Deze sectie is gebaseerd op de tekst van Michiel Ronsse [Ron00].

10.5.1 De opdracht

Om de opgedane kennis in dit hoofdstuk in de praktijk om te zetten, gaan we een device driver voor de 3 LEDs⁵ van het toetsenbord schrijven. Eerst moeten we hiervoor het geschikte device voor aanmaken in de `/dev` directory, en dit doen we met het `mknod` commando, namelijk: `mknod /dev/led c 120 0`. Merk op dat het hier over een character device driver gaat.

Het is de bedoeling dat we door `+` of `-` kunnen aangeven of een ledje uit of aan moet zijn. `echo +++ > /dev/led` zal de 3 ledjes laten oplichten. Als je `echo +-+ > /dev/led` uitvoert mag het middenste led niet oplichten, maar de twee buitenste wel.

10.5.2 Verduidelijking

Als u aandachtig de vorige sectie (sectie 10.5.1) heeft gelezen, ziet u dat voor de led driver er niet veel file operations nodig zijn: u hoeft enkel een read en write functie te definiëren naast

⁵Light Emitting Diodes; op het toetsenbord bedoelen we hiermee “Num Lock”, “Caps Lock” en “Scroll Lock”

de vereiste open en close (release) functies. In de `file_operations` structure zullen er dus 4 niet-NULL entries terug te vinden zijn.

poorten

Om de eigenlijke hardware aan te sturen moeten we de juiste “poorten” van de hardware kennen en zorgen dat we het device met de juiste major nummer aanspreken. Communicatie van het systeem met de hardware verloopt via deze poorten. De data poort van het toetsenbord staat op IO adres `0x60`, de status poort op IO adres `0x64`. Voor de LEDS tenslotte moeten we IO adres `0xED` gebruiken om de LEDs op het toetsenbord aan te sturen. De volgende defines kan je gebruiken in de code om aan te geven welke adressen je voor het keyboard moet gebruiken:

```
#define DATA      0x60      /* data poort v/d keyboard controller */
#define STATUS    0x64      /* status poort v/d keyboard controller */
#define SETLEDS   0xED      /* commando om leds aan te sturen */
```

Niet alleen voor het toetsenbord zijn er poorten voorzien, maar ook voor de andere rand-apparatuur natuurlijk. Zo heeft data poort van de printer IO adres `0x378`. De IO adressen van deze poorten zijn hardware afhankelijk en we werken hier met een PC compatibele architectuur! Voor verschillende klassen van hardware (devices) zijn bereiken van poorten voorzien. Voor de hard disk controller is bijvoorbeeld het bereik `0x320` tot `0x32F` gereserveerd en voor de parallelle poort het bereik `0x378` tot `0x37F`.

Low level poort input en output

Nu we de abstractie van *poorten* kennen in de hardware is het nodig om enkele operaties te leren kennen om echt datatransport mogelijk te maken. Er worden functies voorzien om datatransport in de kernel naar en van de hardware mogelijk te maken: namelijk `outb`, voor output naar de poort toe en `inb` voor input van de poort ⁶. De definities van `inb` en `outb` in `include/asm/io.h` zijn als volgt:

```
inline void outb(char value, unsigned short port)
inline unsigned int inb(unsigned short port)
```

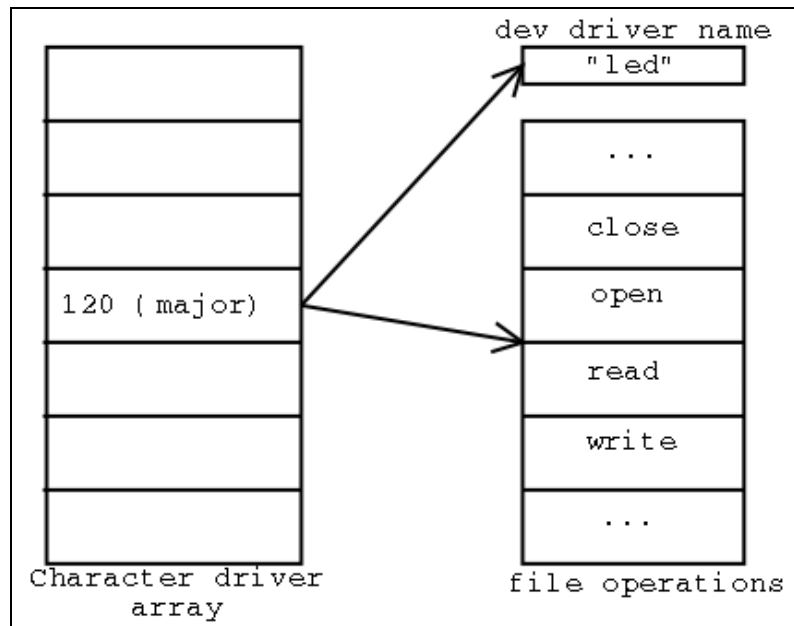
Bij `outb` geven we als argumenten de data (d.i. een byte) en het poort nummer mee aan de functie. Bij `inb` geven we enkel het poort nummer en verwachten we een byte, gelezen van die poort terug. `inb` en `outb` proberen zo snel mogelijk respectievelijk te lezen of te schrijven naar de poort, maar sommige hardware heeft liever dat dit niet gebeurt, maar dat er gewacht kan worden vooraleer te lezen/schrijven. Omdat we hier niet onderhevig zijn aan timing constraints, zoals bij realtime systemen e.d. wel het geval is, kunnen we hier gebruik maken van de “pauserende” tegenhangers van `outb` en `inb` namelijk `outb_p` en `inb_p`. Deze twee functies worden in `include/asm/io.h` als volgt gedefiniëerd:

```
inline void outb_p(char value, unsigned short port)
inline unsigned int inb_p(unsigned short port)
```

driver registreren

Anders dan bij het vorige voorbeeld, moeten we nu de module niet registreren bij het proc filesystem, maar wel als driver registreren. Dit moet in de functie `init_module` gebeuren.

⁶Er wordt zo een hele “familie” van functies voorzien, te vinden in `include/asm/io.h`:
`outb`, `outw`, `outl`, `outsb`, `outsw`, `outsl` voor poort *output*
`inb`, `inw`, `inl`, `insb`, `insw`, `inl` voor port *input*
`outb_p`, `outw_p`, `outl_p`, `inb_p`, `inw_p`, `inl_p` voor *paused I/O*



Figuur 10.3: Character device switch table

Hiervoor gebruiken we de functie `register_chrdriver`, waaraan we als parameters de major nummer, de naam en een verwijzing naar de `file_operations` structure meegeven (in deze volgorde). Deze functie dient om *character* device drivers te registreren. Bij de registratie van deze drivers wordt door de kernel een *character device switch table* bijgehouden zoals getoond in figuur 10.3. Hiervoor wordt een array (`chrdevs`) gebruikt die als entries variabelen van het type `struct device_struct` heeft. Deze structure wordt als volgt gedefinieerd:

```
struct device_struct
{
    const char *name;
    struct file_operations *fops;
};
```

De major nummer zorgt voor de indexering, zodaning dat `chrdevs[major] → fops` de file operaties teruggeeft en `chrdevs[major] → name` de naam van het device teruggeeft.

error codes

Om beter te kunnen checken op errors en soorten errors, is er de header file `errno.h`. Deze include zelf meestal `asm/errno.h`, bekijk deze file om meer te weten te komen over de betekenis van de error codes (meestal kan je de header files vinden in `/usr/src/linux/include/`, dit kan lichtjes verschillen afhankelijk van het systeem). Die laat je bijvoorbeeld toe de return value van `register_chrdriver` te controleren. Als de code `-EBUSY` wordt teruggegeven dan weet je dat het device dat of de resource die je probeert aan te spreken bezet is, en dat de operatie ten gevolge mislukt is. Indien bij `write_leds` of `read_leds` iets misloopt moet er de error code `-EFAULT` teruggegeven worden. Let op de negatie van de error codes. Zo kan simpel gecheckt worden of een return waarde `< 0` is, en aan de hand daarvan concluderen dat er “iets” is misgelopen⁷.

⁷Dit is enkel zo omdat de device drivers in plain C worden geprogrammeerd. Als u een programma in C++ of Java schrijft is het “bad programming practice” foutwaarden mee te geven met de return waarde. Het gebruik van excepties is te verkiezen boven deze methode van fout indicatie en afhandeling.

10.5.3 De code

Framework

Om te beginnen laten we een framework voor de applicatie zien in listing 10.5. Dit zijn de functies en structures die je moet vervolledigen om tot een werkende device driver te komen. Daar we als major nummer voor het device 120 bij `mknod` hadden meegegeven, kunnen we dit best ook definiëren in de source code. `#define LED_DEV 120` bij de andere defines plaatsen volstaat zodat we `LED_DEV` kunnen gebruiken in plaats van 120.

Listing 10.5: Het framework voor de LEDs device driver

```

1 #define LED_DEV 120      /* major nummer van /dev/led */
2 #define DATA 0x60     /* data poort v/d keyboard controller */
3 #define STATUS 0x64   /* status poort v/d keyboard controller */
4 #define SETLEDS 0xED   /* commando om leds aan te sturen */
5
6 #define __KERNEL__
7 #define MODULE
8
9 #include <asm/io.h>
10 #include <linux/module.h>
11 #include <linux/kernel.h>
12 #include <linux/ioport.h>
13 #include <linux/errno.h>
14 #include <linux/major.h>
15 #include <asm/uaccess.h>
16
17 static void set_leds(){}
18 static ssize_t read_leds(struct file * file, char * buf,
19                          size_t count, loff_t *ppos){}
20 static ssize_t write_leds(struct file * file, const char * buf,
21                           size_t count, loff_t *ppos){}
22 static int open_leds(struct inode * inode, struct file * file){}
23 static int close_leds(struct inode * inode, struct file * file){}
24 static struct file_operations led_fops = {};
25 int init_module(void){}
26 void cleanup_module(void){}

```

Om hiermee te kunnen werken hebben we eerst nog wat extra informatie over de functie declaraties nodig.

read_leds

parameters

struct file * file : uit deze “file” vraagt de applicatie om te lezen;

char * buf : de buffer waarin het resultaat geplaatst moet worden;

size_t count : het aantal bytes dat de applicatie vraagt om te lezen;

loff_t *ppos : de offset waar we in de buffer aan het lezen zijn, de “leeswijzer”;

return waarde *ssize_t*: indien >0, het aantal bytes dat gelezen is, anders wordt een error waarde teruggegeven. Deze functie mag maar 1 karakter teruggeven! Een foutief resultaat moet met `-EFAULT` aangeduid worden. Anders zal er 1 teruggegeven worden als er een karakter gelezen is, anders 0 (“einde bestand”).

write_leds

parameters

struct file * file : naar deze “file” vraagt de applicatie om te schrijven (hebben we niet nodig voor onze device driver);
const char * buf : wat er ons gevraagd wordt om te schrijven;
size_t count : het aantal bytes dat de applicatie vraagt om te schrijven;
loff_t *ppos : de offset waar we in de buffer aan het schrijven zijn, de “leeswijzer”;
return waarde *ssize_t*: indien >0, het aantal bytes dat geschreven is, anders wordt een error waarde teruggegeven;

open_leds**parameters**

struct inode * inode zullen we niet hoeven te gebruiken;
struct file * file zullen we niet hoeven te gebruiken;
return waarde *int*: indien dit <0 is wordt er een error waarde teruggegeven;

close_leds**parameters**

struct inode * inode zullen we niet hoeven te gebruiken;
struct file * file zullen we niet hoeven te gebruiken;
return waarde *int*: indien dit <0 is wordt er een error waarde teruggegeven;

De eerste code

Nu we de grondvesten gelegd hebben, rest er ons alleen nog wat code voor de bodies van de functies te schrijven. In listing 10.6 wordt een eerste korte uitleg gegeven over wat er juist in welke body moet komen. Merk op dat de enige functie die echt met de hardware zal communiceren `set_leds()` is.

Listing 10.6: De eerste code voor de LED device driver

```

1 #define LED_DEV 120      /* major nummer van /dev/led */
2 #define DATA 0x60     /* data poort v/d keyboard controller */
3 #define STATUS 0x64    /* status poort v/d keyboard controller */
4 #define SETLEDS 0xED   /* commando om leds aan te sturen */
5
6 #define __KERNEL__
7 #define MODULE
8
9 #include <asm/io.h>
10 #include <linux/module.h>
11 #include <linux/kernel.h>
12 #include <linux/ioport.h>
13 #include <linux/errno.h>
14 #include <linux/major.h>
15 #include <asm/uaccess.h>
16
17 #define FALSE 0
18
19 /* Aantal gebruikers, mag hoogstens 1 worden in deze oefening*/
20 static int led_users=0

```

```
21 /* Houdt de stand van de leds bij */
22 static int led_on[3] = {FALSE, FALSE, FALSE};
23
24 static void set_leds()
25 {
26     /* We zetten het ingelezen patroon om in een geschikte
27      * vorm om aan outb_p mee te geven. Vervolgens
28      * zeggen we dat we de LEDs willen gaan aansturen
29      * waarop we wachten totdat deze klaar zijn om aangestuurd te worden.
30      * In de laatste stap zetten we de LEDs volgens het patroon.
31      *
32      * Dit is de enige functie die echt met de hardware via de poorten
33      * commmunicert!
34      */
35 }
36
37
38 static ssize_t read_leds(struct file * file, char * buf,
39                          size_t count, loff_t *ppos)
40 {
41     /* We gaan in deze functie de stand van de leds uitlezen.
42      * We lezen deze vervolgens uit door de offset *ppos
43      * te verhogen en de LED stand uit te lezen van
44      * de array led_on.
45      * Hierbij moet copy_to_user gebruikt worden om
46      * de gevraagde data in user space te krijgen.
47      */
48 }
49
50 static ssize_t write_leds(struct file * file, const char * buf,
51                           size_t count, loff_t *ppos)
52 {
53     /* Dit is het moeilijkste gedeelte van het geheel.
54      * In deze functie wordt de stand van de LEDs geschreven
55      * Dus iets wat we invoeren vanuit user space moet naar
56      * kernel space gekopieerd worden. Hierbij moet
57      * copy_from_user gebruikt worden.
58      */
59 }
60
61 static int open_leds(struct inode * inode, struct file * file)
62 {
63     /* Zorg ervoor dat slechts 1 gebruiker de driver kan openen,
64      * gebruik makend van led_users.
65      * Gebruik ook de voorgedefinieerde module usage counter
66      */
67 }
68
69 static int close_leds(struct inode * inode, struct file * file)
70 {
71     /* Hier wordt de device driver gesloten. Denk eraan
```

```

72  * dat dit een "illegale" operatie is als voorafgaand
73  * de driver niet geopend was!
74  * Gebruik ook de voorgedefinieerde module usage counter
75  */
76  }
77
78  static struct file_operations led_fops =
79  {
80  /* Hier vul je al de file operations in
81  * die deze device driver ondersteunt,
82  * de overige entries worden NULL gezet.
83  * Merk op dat deze device driver maar 4
84  * soorten operaties ondersteunt (lezen,
85  * schrijven, openen en sluiten)
86  */
87  };
88
89  int init_module(void)
90  {
91  /* Hier gebeurt het registreren van de character device
92  * driver. Zorg ervoor dat hier ook de juiste fout-
93  * afhandeling plaatsvindt en gepaste boodschappen
94  * uitgeschreven worden.
95  */
96  }
97  void cleanup_module(void)
98  {
99  /* Hier wordt de character device driver verwijderd.
100  * Zorg ook hier voor de juiste foutafhandeling
101  * en de gepaste boodschappen
102  */
103  }

```

Vooral bij `write_leds` is het uitkijken geblazen! Dit is een functie die data van user space naar kernel space moet kopiëren om de van hem verwachte functie te vervullen. Besef dat het zeer gevaarlijk is nu buiten het gealloceerde geheugen te schrijven. Omdat we met super user rechten werken in kernel space is het mogelijk dat u daardoor data van andere processen overschrijft of andere data spaces, die niet voor de device driver gereserveerd zijn, vervuult. Dit kan leiden tot een paginafout. We moeten er ook voor zorgen dat we niet meer bytes als `count` kopiëren, anders lezen we teveel uit de user space, wat weer tot een paginafout kan leiden.

Nog enkele kleinigheden in verband met de programmeertaal C die u dient te weten vooraleer u zelf kan beginnen te coderen:

- *Bitgewijze AND (&) operator:*
deze operator kan aangewend worden om een bit te testen, bijvoorbeeld:
 - `x = bbyte&0x02` zet `x` op 2 als de 2e bit van `bbyte` aan staat, anders 0. We kunnen meestal ook gewoon `bit2 = bbyte&2` schrijven hiervoor.
 - `y = bbyte&4` (binair is 4 nl. 000100) zet de stand van `y` op 4 als de 3e bit van `bbyte` aanstaat, anders op 0.
- *De Left-Shift operator <<:*
verschuift de bits in de variabele een opgegeven aantal bits naar links. Hiervoor worden

er rechts nullen ingeschoven. Als we rekening moeten houden met het teken en we hebben een negatieve waarde worden er enen in plaats van nullen ingeschoven.

De startcode

Na al deze informatie is het nu mogelijk om de device driver zelf af te maken. De start code wordt u gegeven in listing 10.7. U moet de code vervolledigen en werkend krijgen. U moet ook alle code begrijpen en kunnen uitleggen. Met andere woorden moet u vragen kunnen beantwoorden in verband met de structuur, opbouw en betekenis van de code.

Listing 10.7: De startcode voor de LED device driver

```

1 #define LED_DEV 120      /* major nummer van /dev/led */
2
3 #define DATA  0x60     /* data poort v/d keyboard controller */
4 #define STATUS 0x64     /* status poort v/d keyboard controller */
5 #define SETLEDS 0xED    /* commando om leds aan te sturen */
6
7 #define __KERNEL__
8 #define MODULE
9
10 #include <linux/module.h>
11 #include <linux/kernel.h>
12 #include <linux/ioport.h>
13 #include <linux/errno.h> /* Definitie van EFAULT, EBUSY, EIO, ... */
14 #include <linux/major.h>
15 #include <asm/io.h>
16 #include <asm/uaccess.h>
17
18 #define FALSE 0
19
20 static int led_on[3] = {FALSE, FALSE, FALSE}; /* status v/d LEDs */
21
22
23
24 static void set_leds(){
25     int pattern = (led_on[1]<<2) + (led_on[0]<<1) + led_on[2];
26
27     outb_p(SETLEDS, DATA);
28     while (inb_p(STATUS)&2)
29         ;
30     outb_p(pattern, DATA);
31 }
32
33
34 static ssize_t read_leds(struct file * file, char * buf,
35                         size_t count, loff_t *ppos){
36     char kar;
37
38     kar = (*ppos<3) ? (led_on[*ppos] ? '+' : '-') : (*ppos==3) ? '\n' : 0;
39
40 }
41

```

```
42
43 static ssize_t write_leds(struct file * file, const char * buf,
44                          size_t count, loff_t *ppos){
45     int i,start,stop;
46
47
48     if (*ppos<3) {
49         start = *ppos;
50         stop = *ppos+count;
51         if (stop > 3) stop = 3;
52         for (i=start; i<stop; i++)
53             switch(temp[i-start]){
54                 case '+': led_on[i] = TRUE; break;
55                 case '-': led_on[i] = FALSE; break;
56             }
57     }
58
59     set_leds();
60
61     /* zorg ervoor ppos aan te passen! */
62 }
63
64
65 static int open_leds(struct inode * inode, struct file * file){
66
67     return 0;
68 }
69
70
71 static int close_leds(struct inode * inode, struct file * file){
72
73     return 0;
74 }
75
76
77 static struct file_operations led_fops = {
78
79 };
80
81
82 int init_module(void){
83     if (register_chrdev(LED_DEV, "LED", &led_fops) == -EBUSY){
84         return -EIO;
85
86     }else{
87
88     }
89     return 0;
90 }
91
92
```

```

93 void cleanup_module(void){
94     if (unregister_chrdev(LED_DEV, "LED"))
95
96     else
97 }

```

10.5.4 Praktisch gebruik van de device driver

Het gebruik van de device driver ontwikkeld in dit hoofdstuk is enkel geïllustreerd door het shell commando `echo ++ > /dev/led`. Wat als je nu deze device driver wil gebruiken vanuit een ander programma? Door de voorstelling van devices als files, kunnen we ook de traditionele file-operaties gebruiken om een device driver aan te sturen. We kunnen eerst een “open” operatie op de device driver uitvoeren, waarna we met een “write” functie naar het device kunnen schrijven.

We kunnen de device driver vanuit een C programma met de functie `fopen` openen en met behulp van `fprintf` kan er data naar geschreven worden. Listing 10.8 illustreert hoe dit gecodeerd kan worden.

Listing 10.8: Aansturen van het device vanuit een C programma

```

#include <stdio.h>
int main(int argc, char *argv[])
{
    FILE *file = fopen("/dev/led", "w+");
    if(file<=0)
    {
        printf("Kan leds niet aansturen\n");
        return -1;
    }
    else
    {
        fprintf(file, "%s\n", "+++");
        return 1;
    }
}

```

Nu is het mogelijk om programma's te maken die de LEDs van het toetsenbord gebruiken. Enkele mogelijkheden zijn:

- een programma dat binnenkomende emails meldt door een LED te laten knipperen
- een programma dat de LEDs laat knipperen op de maat van muziek
- een programma dat de processor belasting laat zien aan de hand van de LEDs

Opgave Schrijf een programma dat uit een bestand een sequentie van LED standen leest en deze uitvoert. Zorg ervoor dat de overgang van de vorige naar de volgende toestand met een redelijk tijdsinterval rekening houdt. Zoek zelf de nodige functies hiervoor op.

10.6 Device Driver Practicum FAQ

PENDING: *dit moet nog geïntegreerd worden in de rest van de tekst*

10.6.1 In de functie `write_leds`, moet je gebruik maken van `copy_from_user`. Je moet dan van de `buf` naar de kernel space schrijven. Is dit dan `led_on`?

De device driver zelf wordt in de kernel gestoken door middel van `insmod`. Dus alle datastructuren die in de code van de device driver gedeclareerd worden, zoals `led_on`, behoren tot kernel space. De beste manier om nu die data uit `constant char * buf` (dit is een buffer in de user space) naar je kernel space te krijgen is gebruik te maken van een tijdelijke datastructuur (bijv. `char temp[3]`), waarin je met `copy_from_user` de inhoud van `buf` kopieert. Je kan dit niet rechtstreeks naar `led_on` doen, omdat `led_on` een 1 of 0 bevat, en `buf` bevat '+' of '-'. Als je bijvoorbeeld `cat ++- > /dev/led` uitvoert vanaf de shell-prompt zal de data die naar `/dev/led` gestuurd worden in die `buf` bijgehouden worden en in dit geval wordt “++-” naar `/dev/led` gestuurd.

Als je dus `buf` naar `temp` hebt gekopieerd is alle gewenste data in kernel space en kan je `temp` doorlopen om de juiste waarden in `led_on` te zetten ('+' uit `temp` wordt 1 in `led_on` en '-' uit `temp` wordt 0 in `led_on`). Hierbij geeft `*ppos` aan waar je mag beginnen in `temp` en `*ppos+count` tot waar je mag lezen in `temp`.

10.6.2 Er staat ook dat `ppos` moet aangepast worden. Is dit voor de volgende keer dat je de functie gebruikt of moet er een lus gemaakt worden?

De enige lus die nodig is in `write_leds` staat reeds in de startcode. Je moet hier dus zelf geen lus meer implementeren (het besturingssysteem zorgt zelf voor het opdelen van de te verwerken data in “chunks”, dit kan je zien doordat je uit de `temp` enkel van `*ppos` tot `*ppos+count` gebruikt). Dit wil natuurlijk wel zeggen dat als je “count” data verwerkt hebt, je `*ppos` moet aanpassen tot `*ppos+count`, zodat bij de volgende oproep je weet waar je laatst data gelezen hebt uit de buffer.

10.6.3 Om `ppos` aan te passen volstaat het dan om `*ppos++` te schrijven of moeten we dat als een pointer beschouwen en iets in de aard van `ppos = ppos->next` schrijven?

`ppos` heeft als type `loff_t`, in feite is dit gewone een herdefinitie van een gewoon primitief type (`long; typedef long loff_t;`), `ppos->next` bestaat dus niet, want `ppos` is van een primitief type. Je kan gewoon de *inhoud* van `ppos` (want het is een pointer) gelijkstellen aan een nieuwe waarde alsof het een pointer naar een int betreft.

10.6.4 Waar kunnen we informatie vinden over de module usage counter?

Dit is niet in de cursus gezet, maar had er wel in moeten staan. Hier is hetgeen wat je moet doen: in `open_leds` zet je gewoon `MOD_INC_USE_COUNT` bij en in `close_leds` zet je `MOD_DEC_USE_COUNT`, bijv.:

```
static int close_leds(struct inode * inode, struct file * file){
    ...
    MOD_DEC_USE_COUNT;
    ...
}
```

Dit zorgt ervoor dat een module usage counter voor onze module verhoogt wordt indien deze ergens gebruikt wordt (bijv. voor een `fopen(/dev/led/)`) en verlaagt indien deze niet meer gebruikt wordt.

10.6.5 Bij de `read_leds` : moet je een waarde teruggeven? Hoe controleer je dan of er een foutief resultaat is? Moet dat met de waarde die kar heeft?

Je moet bij `read_leds` een waarde teruggeven. Pagina 45 zegt hierover: “indien >0 , het aantal bytes dat gelezen is, anders wordt een error waarde teruggegeven. Deze functie mag maar 1 karakter teruggeven! Een foutief resultaat moet met `-EFAULT` aangeduid worden. Anders zal er 1 teruggegeven worden als er een karakter gelezen is, anders 0”

Dus:

1. Als alles goed gegaan is moet er dus '1' teruggegeven wordt (`ssize_t` is een `typedef` van het type `int`, dus dit kan). Dit kunnen we standaard zo doen omdat er telkens maar 1 karakter gelezen wordt. de “return 1” komt dus onderaan te staan
2. Wanneer moeten we dan `-EFAULT` teruggeven? Wel, als het kopiëren van die ene karakter van kernel space naar user space misloopt.
3. Wat als `ppos` nu voorbij de laatst beschikbare plaats in de `led_on` array wijst (`ppos` is groter dan de lengte van de array)? Dan hebben we gedaan en kunnen we een 0 teruggeven. Let wel dat dit moet gecontroleerd worden vooraleer er data van kernel naar user space wordt verzonden, anders zou het kunnen dat er gelezen wordt buiten het beschikbare geheugen. Vergeet ook niet `ppos` te verhogen met het aantal karakters dat de functie gelezen heeft (en dat is dus 1 hier).

Idem bij `write_leds`, alleen dat het daar over “count” bytes gaat. Dus ipv 1 terug te geven als alles goed is gegaan, moet je `count` teruggeven als alles goed gegaan is. Hier moet nergens 0 teruggegeven worden omdat je op voorhand al weet dat je niet meer als “count” bytes kan lezen uit `buf`.

10.6.6 Moeten we in `init_module` de functie `read_leds` en `write_leds` oproepen, of wordt dit gedaan door het systeem?

Als je bedenkt dat dit gewone `file_operaties` zijn kan je de lijn verder trekken tot het gebruik van de gewone files. Je moet het uit een verschillend perspectief bekijken als je een device driver aan het programmeren bent: stel je voor dat jij de “file” bent en dat er iemand een `fopen("gijzelf");` op u doet! Dan wordt `open_leds` opgeroepen op u door het systeem. Jij voorziet de functies die de eigenlijke gebruiker van het systeem kan gebruiken: om data van je te lezen en naar je te schrijven, om je te openen en terug te sluiten. Je moet dus zelf nergens die functies oproepen, dat zal het systeem doen.

10.6.7 Moeten `open_leds` en `close_leds` ook ergens worden aangeroepen?

Uitleg idem als in sectie [10.6.6](#)

10.6.8 moeten we `unregister_chrdev` eigenlijk met een waarde vergelijken zoals `register_chrdev`?

Nee, dat is niet nodig. Enkel als je wil registreren wil je dat er gekeken wordt of het lukt. Als je wil de-registreren trek je je daar niks meer van aan. Gewoon oproepen dus ,meer kan er toch niet gedaan worden als je aangeeft dat je de module niet meer wil gebruiken. De functie `cleanup_module` heeft ook als return type `void`, dus is het niet de bedoeling daar een waarde terug te geven. Je kan best wel met `printk` uitschrijven of `unregister_chrdev(LED_DEV, "LED")` gelukt is (geeft 1 terug dan) of dat het niet gelukt is (geeft 0 terug dan).

10.7 Verdere informatie

Voor meer informatie over het programmeren van device drivers worden de volgende bronnen aanbevolen: [Ron00], [AJ00], [Pom99], [LDP] en [SM99].

Hoofdstuk 11

Een kernel compileren en installeren

PENDING: *Nog afmaken, veel steunen op [Man01] en [War99]*

11.1 De kernel broncode bekomen

11.2 De kernel patchen

11.3 De kernel configureren

11.4 De kernel en bijbehorende modules compileren

11.5 De nieuwe kernel installeren

Hoofdstuk 12

Hoe software installeren

12.1 Red Hat Package Manager

Red Hat is een bedrijf dat Linux distributies verzorgt. Om het de gebruikers te vergemakkelijken heeft het een formaat ontworpen om software te installeren op een systeem, namelijk *RPM*¹, het *Red Hat Package Management systeem*. De software zit verpakt in een bestand dat de extensie `.rpm` draagt. Dit bestand kan met behulp van het `rpm` programma geïnstalleerd worden. De syntax van het `rpm` programma is `rpm [opties] package-naam`. Om nu bijvoorbeeld het package `hugs-1.33-2.i386.rpm` te installeren kan men `rpm -i hugs-1.33-2.i386.rpm` intypen. Let wel dat men voor de meeste software installaties als beheerder of superuser op het systeem moet ingelogd zijn. In de naam van de package zit ook informatie over het package namelijk: *hugs* staat voor de naam van de software, *1.33* staat voor het versie nummer, *2* staat voor de release nummer en *i386* geeft aan dat het package voor een intel 386 processor architectuur bedoeld is.

Om een stuk software te updaten met behulp van `rpm`, bijvoorbeeld versie 1.47 van `hugs` kan men `rpm -U hugs-1.47-1.i386.rpm` intypen. Indien men niet weet of een voorgaande versie van de software reeds aanwezig is op het systeem kan men ook `rpm -Ui hugs-1.47-1.i386.rpm` intypen wat de software gewoon installeert indien deze niet reeds aanwezig was op het systeem.

Een `rpm` package kan natuurlijk ook “ondervraagd” worden over de inhoud ervan. Dit doet men door `rpm -qip package-naam` in te typen. Om te verifiëren of er niet met het package geknoeid is alvorens het te installeren kan men `rpm -V package-naam` gebruiken (let op, de `V` is uppercase).

Een van de grootste voordelen van `rpm` is toch wel de mogelijkheid om software met dezelfde moeite te verwijderen als het gekost heeft om het te installeren. Als men `hugs-1.47-1.i386.rpm` wil verwijderen kan men simpelweg `rpm -e hugs-1.47-1.i386.rpm` intypen. Tenslotte is het nog nuttig te weten dat de *verbose* optie ook met `rpm` werkt. Indien men meer informatie wenst kan men altijd een lowercase `v` toevoegen aan de opties.

12.2 Source distributies installeren

Veel software voor Linux systemen kan afgehaald worden in de vorm van broncode vergezeld van een *make* file. Deze *make* file beschrijft de afhankelijkheden en vereisten van de code en geeft aan hoe de code moet gecompileerd worden. De *make* file kan de volgende soorten lijnen bevatten:

- lege lijnen;
- commentaar lijnen, wordt altijd voorafgegaan door `#`;

¹<http://www.rpm.org>

- lijnen die afhankelijkheden aangeven;
- lijnen die suffixen voor bestandsnamen vastleggen;
- lijnen die commando's bevatten;
- macro definities, bijv.: `define naam\ string\ endef` of `naam =string`;
- include statements;

Een voorbeeld van een Make file met wat uitleg vindt u in appendix B.

Meestal zal de broncode verpakt zijn in een gecomprimeerde archive (zie sectie 4.6), zoals een tar.gz file. Pak dit bestand uit in een directory. Het eerste wat men dan doet is de README file lezen, indien de archive er eentje bevat. Meestal is er ook een INSTALL file, waarin de instructies voor de installatie van de broncode beschreven staan. Meestal moet men de volgende stappen doorlopen:

1. `./configure`, indien dit commando erbij zit moet men het uitvoeren. Het zal controleren of het systeem voldoet aan de vereisten van de software en de make files generen;
2. `make`, simpelweg `make` intypen zal de compilatie procedure starten, waarvan u de output te zien krijgt;
3. `make check`, dit is een niet verplichte tussenstap die de huidige toestand nakijkt (of alles wel in orde is);
4. `make install`, als *superuser* kan u dit uitvoeren om de gecompileerde en gelinkte bestanden te installeren op het systeem;
5. `make clean`, dit ruimt de bestanden op die ontstaan zijn bij de compilatie, maar na installatie overbodig zijn;

Hoofdstuk 13

Linux op het web

Het Internet is het belangrijkste distributie kanaal voor Linux en software voor het Linux systeem. Deze sectie geeft een aantal links naar websites waar men software kan afhalen of meer informatie kan vinden. De volgende links zijn een greep uit het ruime aanbod, en slechts enkele categoriën. Voor meer uitgebreide lijsten kan je surfen naar <http://www.freshmeat.net>, <http://linux.davecentral.com> en naar <http://www.linux.com>. Je kan ook op de nieuwsgroep *be.comp.os.linux* terecht met je vragen.

13.1 Distributies

Er zijn verschillende Linux distributies beschikbaar, die men van het web kan afhalen. Dit zijn de bekendste distributies:

RedHat Linux <http://www.redhat.com>

Mandrake Linux <http://www.mandrake.com>

Suse Linux <http://www.suse.de>

Slackware Linux <http://www.slackware.com>

Caldera Linux <http://www.caldera.com>

Debian Linux <http://www.debian.org>

13.2 Kantoor-applicaties

Onder kantoorapplicaties verstaan we tekstverwerkingsprogramma's, spreadsheets en presentatie-programma's.

Sun Star Office <http://www.sun.com/products/staroffice/>

Corel Wordperfect http://www.linux.corel.com/products/wpo2000_linux/index.htm

KOffice <http://www.koffice.org/>

Gnumeric <http://www.gnome.org/projects/gnumeric/>

Abiword <http://www.abiword.org/>

13.3 Software ontwikkeling

Programmeren De lijst met alle beschikbare software om aan software ontwikkeling te doen is zeer lang. We beperken ons hier tot degene die bruikbaar kunnen zijn voor de richting informatica-kennistechnologie zoals gegeven aan het LUC.

Free Pascal <http://www.freepascal.org>

KDevelop <http://www.kdevelop.org>

gcc <http://gcc.gnu.org/>

gdb <http://sources.redhat.com/gdb/>

Borland JBuilder <http://www.borland.com/jbuilder/>

Qt <http://www.trolltech.com/products/qt/>

SWI Prolog <http://swi.psy.uva.nl/projects/SWI-Prolog/>

GNU Prolog <http://pauillac.inria.fr/~diaz/gnu-prolog/>

Scheme <http://www.swiss.ai.mit.edu/projects/scheme/>

Hugs <http://www.haskell.org/hugs/>

IBM JSDK 1.3 <http://www.ibm.com/java/jdk/index.html>

Sun JSDK 1.3 <http://java.sun.com/j2se/1.3/>

Freebuilder <http://www.freebuilder.com/>

Gegevensbanken Er zijn ook vele gegevensbanken beschikbaar voor Linux.

MySQL <http://www.mysql.com/>

PostgreSQL <http://www.postgresql.org/>

Sybase <http://www.sybase.com/linux/>

IBM DB2 <http://www-4.ibm.com/software/data/db2/linux/>

13.4 Meer informatie

Linux Documentation Project <http://www.linuxdoc.org/>

UK.LINUX.ORG <http://www.linux.org.uk/>

Brave GNU World <http://www.bgw.org/tutorials/>

GNU's Not Unix! <http://www.gnu.org/>

Bash Reference Manual <http://www.gnu.org/manual/bash/index.html>

Linux Online <http://www.linux.org/>

Linux.com <http://www.linux.com/>

ddj Linux <http://www.ddj.com/topics/linux/>

Bijlage A

Pointers naar functies

De volgende code listing laat een paar voorbeelden zien van functies die door middel van een verwijzing naar de functie als parameter doorgegeven worden aan andere functies. Een functie is eigenlijk niet meer als een verwijzing naar een adres waar de code van de desbetreffende functie daadwerkelijk begint. Functies zijn in feite een beetje zoals datastructuren: omdat zij ook in het geheugen opgeslagen worden is er ook een beginadres hiervoor te vinden.

Listing A.1: callbackf.c

```
1 #include<stdio.h>
2
3 int function1(void){
4     printf("this is function 1\n");
5 }
6
7 int function2(int (*p)()){
8     printf("function 2 printing:\n");
9     p();
10    printf("function 2 printed\n");
11 }
12
13 double somfun(int n, double (*f)(int k)){
14     double s = 0;
15     int i;
16     for (i=1;i<=n;i++)
17         s += f(i);
18     return s;
19 }
20
21 double term(int k){
22     return 2*k+1/k;
23 }
24
25 int main(void){
26     double som;
27     int (*p)();
28     int function1();
29     p = function1;
30     function1();
31     function2(p);
32     function2(function1);
```

```
33 p();
34 som = somfun(3, term);
35 printf("%lf\n", som);
36 return 0;
37 }
```

Als we het programma van listing [A.1](#) compileren en uitvoeren krijgen we als output:

```
this is function 1
function 2 printing:
this is function 1
function 2 printed
function 2 printing:
this is function 1
function 2 printed
this is function 1
13.000000
```

We kunnen `p` laten wijzen naar `function1`, door `p` als volgt te definiëren: `int (*p)();`. Nu heeft `p` de juiste typering om er `function1` aan toe te kennen: `p = function1;`. Vervolgens kan `function2` als parameter een variabele ontvangen die op dezelfde manier als `p` en dus ook `function1` gedefinieerd is. M.a.w. we geven `p` mee als argument aan `function2`, en in de body van deze function wordt `p` dan opgeroepen. Anderzijds kunnen we ook gewoon `function1` als parameter meegeven aan `function2`. In feite wordt dan het adres van de functie meegegeven, zodanig dat als men dan een verwijzing naar dat adres neemt, men eigenlijk aan de eerste statement van de functie zelf zit.

Tenslotte wordt er nog een praktisch voorbeeldje gegeven in listing [A.1](#) door de functie `term` als argument aan `somfun` mee te geven. Probeer zelf eens een andere implementatie voor `term` te verzinnen en dan `somfun` terug op te roepen. Je zal zien dat het resultaat in `som` dan ook verandert.

Bijlage B

Voorbeeld van een Makefile

Het programma `make` is zeer handig om een “onderhoudbaar” en makkelijk compilatie proces te voorzien. Het gebruikt hiervoor de zogenaamde *Makefile*. In het voorbeeld (listing B.1) kan je een aantal macro-definities zien zoals `PDF = pdfelatex`. Alles wat achter `#` staat is commentaar. De macro-definitie `PDF` wordt opgeroepen door `$(PDF)`. Het `$` teken geeft aan dat het de variabele moet “expanden”. Een lijn als `Linux2001v2.pdf:Linux2001v2.tex ...` wil zeggen dat als er een verandering waargenomen is in een van de `TEX` files (algemener één van de files achter het dubbel punt) de pdf-file moet geupdate worden. Dit geeft aan dat er een afhankelijkheid bestaat tussen het bestand voor het dubbelpunt en de bestanden achter het dubbelpunt. Deze regel is de “hoofdregel” van de Makefile, als je het commando `make` intypt zal het standaard de eerste regel van de vorm `all : <bronA> <bronB> ...` nemen. Hier is dit gedefinieerd als `all: Linux2001v2.pdf`. De update gebeurt door de lijnen die op de afhankelijkheidsregel volgen uit te voeren. Als je `make clean` intypt zal alleen de sectie die begint met `clean:` in de Makefile uitgevoerd worden. Idem voor `make html`.

Listing B.1: de Makefile voor dit document

```
1 # De compilers:
2 DVI = latex # for a DVI file
3 PS = dvips # for a postscript file
4 PDF = pdfelatex # for a pdf file
5 BIB = bibtex # for generating the BibTeX entries
6 INDEX = makeindex # for generating the index
7
8 # de hoofdfile van de tekst
9 FILE = Linux-2002-v2.1
10
11 # de flags voor de DVI compiler en de output file
12 DVI2PSFLAGS = -o $(FILE).ps
13 DVIFILE = $(FILE).dvi
14
15 # welke bestanden mogen weg gedaan worden
16 DELETABLE = *.aux *.toc *.log *.out *.ind *.lof *.idx *.ilg *.lot\
17 *.blg *.bbl *.lol *.lo *.brf *.*~
18 CLEANABLE = *[^e]ps *pdf *.tex~ *.dvi *.ps.gz *.*~ *.bak
19
20 all: $(FILE).pdf
21
22 # de hoofdregel van deze Makefile
23 $(FILE).pdf: $(FILE).tex basiscommandos.tex devicedr.tex\
24 dir-and-files.tex inleiding.tex linonweb.tex\
```

```

25         linprogr.tex shellprogr.tex softinstall.tex\
26         app-callback.tex app-makefile.tex bibliotheken.tex\
27         oploefshell.tex editors.tex regexpr.tex linuxcourse.bib\
28         bijdragen.tex grmail.txt edm.sty awk.tex
29     make pdf
30
31 pdf: $(FILE).tex basiscommandos.tex devicedr.tex\
32     dir-and-files.tex inleiding.tex linonweb.tex\
33     linprogr.tex shellprogr.tex softinstall.tex\
34     app-callback.tex app-makefile.tex bibliotheken.tex\
35     oploefshell.tex editors.tex regexpr.tex linuxcourse.bib\
36     grmail.txt edm.sty awk.tex
37     $(PDF) $(FILE) #pdfelatex moet je een paar keer uitvoeren
38     $(BIB) $(FILE)
39     $(PDF) $(FILE)
40     $(INDEX) $(FILE).idx
41     $(PDF) $(FILE)
42     rm -rfv $(DELETABLE) #verwijder intermediate files
43
44 ps:
45     $(DVI) $(FILE) #latex moet je een paar keer uitvoeren
46     $(BIB) $(FILE)
47     $(DVI) $(FILE)
48     $(INDEX) $(FILE).idx
49     $(DVI) $(FILE)
50     $(PS) $(DVI2PSFLAGS) $(DVIFILE)
51     rm -rfv $(DELETABLE) #verwijder intermediate files
52     cp $(FILE).ps temp.ps
53     rm -fv $(FILE).ps.gz
54     gzip $(FILE).ps
55     mv temp.ps $(FILE).ps
56
57 # ruim de boel op, alles behalve de benodigde sources
58 clean:
59     rm -rfv $(CLEANABLE) $(DELETABLE)

```

Let erop dat de regels voor de Makefile met een tab moeten beginnen!

Nu werkt dit voorbeeldje met \LaTeX files als input en een pdf file als output. Hetzelfde kan gedaan worden voor een hoop object files, waarvoor c files moeten gecompileerd worden waarvan een programma afhangt. Een voorbeeld hiervan vind je in listing [B.2](#).

Listing B.2: Een simpele Makefile

```

1 quantumsearch: main.o q.o bubble.o
2     gcc -o quantumsearch main.o q.o bubble.o
3
4 main.o: main.c
5     gcc -c main.c -o main.o
6
7 q.o: q.c q.h
8     gcc -c q.c -o q.o
9
10 bubble.o: bubble.c bubble.h

```

11 `gcc -c bubble.c -o bubble.o`

De listing [B.2](#) zegt dat de (executable) file `quantumsearch` afhangt van `main.o`, `q.o` en `bubble.o`. (door de regel `quantumsearch: main.o q.o bubble.o`). De volgende regel in listing [B.2](#) geeft aan wat er moet gebeuren om `quantumsearch` terug up-to-date te krijgen. We geven op hun beurt ook aan waarvan `main.o`, `q.o` en `bubble.o` afhangen. Indien blijkt dat één van deze bestanden niet up-to-date is, zullen eerst de bijbehorende opdrachten geactiveerd worden, waarna verder gegaan wordt met `quantumsearch` bij te werken.

De twee voorbeelden, listing [B.1](#) en listing [B.2](#), geven duidelijk aan dat Makefiles handig zijn in allerlei software projecten. Ze worden vooral gebruikt om software projecten bestaande uit meerdere bestanden van broncode op een efficiënte manier te compileren, het overbodige compileer werk vermijdend. Het zal de up-to-date files niet hercompileren, en enkel de files die aangepast zijn hercompileren. Een Makefile kan dus gebruikt worden wanneer we een bepaalde output willen verkrijgen, afhankelijk van één of meerder input files.

Bijlage C

Oplossingen oefeningen

Oplossingen oefeningen uit sectie 7.12:

Listing C.1: oplossing oef 2

```
#!/bin/bash
if [ ! -d ~/backup ]
then
    mkdir ~/backup
fi

movedate='date +%s'
tar -c $@ | bzip2 > ~/backup/$movedate.tar.bz2
chmod uago-wx ~/backup/$movedate.tar.bz2
```

Listing C.2: oplossing oef 3

```
#!/bin/bash
show_me_your_stuff(){
    ls -l $1
    cat $1
}

i=1
while [ $i -le $# ]
do
    show_me_your_stuff $1
    shift
done
```

Listing C.3: oplossing oef 4

```
#!/bin/bash
blaai=/tmp/attachment.tar.gz
if [ -d $1 ]
then
    tar -cv $1 | gzip > $blaai
    mail 'whoami' < $blaai
else
    echo "$1 is geen directory"
fi
```

Listing C.4: oplossing oef 5

```
#!/bin/bash
eval text='$'$#
maxnr=$#
i=1
while [ $i -lt $maxnr ]
do
    eval receiver='$'$i
    mail $receiver < $text
    i='expr $i + 1'
done
```

Listing C.5: oplossing oef 6

```
#!/bin/bash
until who | grep "$1">/dev/null
do
    sleep 10
done

echo "$1 is ingelogd"
```

Listing C.6: oplossing oef 7

```
#!/bin/bash
while true
do
    i=1
    while [ $i -le $# ]
    do
        eval user='$'$i
        iser='who | grep "$user"'
        if [ -n "$iser" ]
        then
            echo "user $user is er"
        else
            echo "user $user is er niet"
        fi
        i=' expr $i + 1 '
    done
    sleep 10
done
```

Listing C.7: oplossing oef 8

```
#!/bin/bash
while true
do
    echo "geef een commando:"
    read jobname
    if [ "$jobname"= "quit" ]
    then
        exit
    fi
```

```
starttijd='exec date'  
echo "$jobname starts at $starttijd">> jobaccounting  
$jobname  
eindtijd='exec date'  
echo "$jobname ends at $eindtijd">> jobaccounting  
done
```

Bibliografie

- [Aiv00] Tigran Aivazan. *Linux Kernel Internals*. World Wide Web, <http://www.linuxdoc.org>, 2000.
- [AJ00] Alessandro and Jonathan. *Linux Device Drivers, 2nd edition*. O'Reilly, 2000.
- [Bar96] Daniel Barlow. *The Linux gcc HOWTO*. LDP, <http://www.linuxdoc.org>, 1996.
- [Bow98] Ivan T. Bowman. Conceptual architecture of the linux kernel. Technical report, University of Waterloo, <http://plg.uwaterloo.ca/~itbowman/papers/CS746G-a1.html>, 1998.
- [BST98] Ivan T. Bowman, Saheem Siddiqi, and Meyer C. Tanuan. Concrete architecture of the linux kernel. Technical report, University of Waterloo, <http://plg.uwaterloo.ca/~itbowman/papers/CS746G-a2.html>, 1998.
- [Das01] Sumitabha Das. *Your UNIX, The Ultimate Guide*. McGrawHill, 2001.
- [GG] Arne Georg Gleditsch and Per Kristian Gjermshus. The linux kernel hackers' guide. Worl Wide Web. <http://www.linuxdoc.org/LDP/khg/HyperNews/get/khg.html>.
- [Hek97] Jessica Perry Hekman. *Linux in a nutshell, a desktop quick reference*. O'Reilly, 1997.
- [JB98] Jack Tacket Jr. and Steve Burnett. *Special Edition Using Linux (fourth edition)*. 1998.
- [LDP] The linux documentation project. World Wide Web. <http://www.linuxdoc.org>.
- [Man99] Steve Mansour. A tao of regular expressions. World Wide Web, 1999.
- [Man01] MandrakeSoft. *MandrakeLinux Reference Manual*, April 2001. <http://www.linux-mandrake.com>.
- [Pom99] Ori Pomerantz. *Linux Kernel Module Programming Guide*. LDP, <http://www.linuxdoc.org>, 1999.
- [Ron00] Michiel Ronsse. *Cursus besturingssystemen, Linux-practica: handleiding*, 2000.
- [SG94] Abraham Silberschatz and Peter B. Galvin. *Operating System Concepts (fourth edition)*. Addison-Wesley, 1994.
- [SM99] Richard Stones and Neil Matthew. *Beginning Linux Programming (second edition)*. Wrox, 1999.
- [Tan] Andy Tanenbaum. Minix. <http://www.cs.vu.nl/~ast/minix.html>.
- [War99] Brian Ward. *The Linux Kernel HOWTO*. LDP, <http://www.linuxdoc.org>, 1999. <http://www.linuxdoc.org>.
- [Whe00] David A. Wheeler. *Program Library HOWTO*. LDP, <http://www.linuxdoc.org>, 2000. <http://www.linuxdoc.org>.

Over dit document

Deze tekst is geschreven in L^AT_EX, een verzameling macro packages gebouwd rond de typesetting taal T_EX¹. Om dit naar het dvi-formaat om te zetten is er het latex programma gebruikt dat het bron bestand als argument aannam. Deze dvi is vervolgens geconverteerd naar het postscript² formaat met behulp van dvips. Een pdf³ bestand is uit de bron tekst gegenereerd door pdfelatex. Alle tekeningen zijn gemaakt met dia⁴. Eén tabel is gemaakt met Gnumeric⁵, gebruik makend van de latex2e output mogelijkheid.

De laatste versie van deze cursus is te vinden op <http://lumumba.luc.ac.be/kris/courses/systeemprogrammatuur/>. Via CVS⁶ kan je ook de L^AT_EX source files verkrijgen. U kan hier anoniem op inloggen en het paswoord “hiephoi” gebruiken. De volgende commando’s zijn hiervoor nodig (aangenomen dat CVS op uw systeem geïnstalleerd is).

```
export CVSROOT=:pserver:anoncvs@lumumba.luc.ac.be:/home/kris/CVSROOT
cvs login
cvs checkout courses/linux/course
```

In een Microsoft Windows omgeving vervangt u `export` door `set`. Alle opmerkingen, kritiek, eventuele bijdragen zijn zeer welkom bij:

kris.luyten@luc.ac.be

Veel dank aan degenen die dit al gedaan hebben (zie ook volgend blad).

Laatste update: 22 april 2002

¹<http://www.tug.org>

²<http://www.adobe.com>

³<http://partners.adobe.com/asn/developer/technotes/acrobatpdf.html>

⁴<http://www.lysator.liu.se/~alla/dia>

⁵<http://www.gnome.org/gnumeric>

⁶<http://www.cvshome.org>

Auteurs en bijdragen

Deze tekst kwam tot stand met de hulp van de volgende mensen:

Chris Raymaekers

- originele auteur
- GNU Awk scripting

Kris Luyten

- auteur
- omzetting \LaTeX

Karin Coninx

- grondige reviews en correcties

Jan Van den Bergh

- aanpassingen
- correcties

Jori Liesenborgs

- grondige reviews en correcties
- suggesties

Tom Van Laerhoven

- Bibliotheken
- correcties

Herman Bruyninckx

- correcties
- suggesties
- aanvullingen

Fabian Di Fiore

- correcties

Jo Segers

- correcties

Chris Vandervelpen

- correcties

Panagiotis Issaris

- suggesties
- correcties

Gioti Arkoudopoulos

- tekst bijgevoegd in draft

Tom Haber

- correcties

Simon Vandemoortele

- correcties

Tom Janssens

- suggesties

Index

- .plan, 20
- L^AT_EX, 98
- ABI, 59
- alias, 35
- bestandsnaam, 20
- bibliotheken, 57
 - dynamically loaded, 57
 - dynamisch geladen, 59
 - gedeelde, 58
 - statisch, 58
- bijdragen, 99
- bzip2, 23
- cat, 20, 21
- cd, 14
- chmod, 22
- copy, 21
- cp, 21
- CVS, 98
- directories, 14
 - groep, 22
 - rechten, 22
 - structuur, 14
- ed, 20
- editors, 31
 - ed, 20
 - emacs, 20, 32
 - pico, 20
 - vi, 20, 31
 - vim, 31
- emacs, 20, 32
- file
 - groep, 22
 - rechten, 22
- finger, 20
- grep, 18
- groep, 22
- gzip, 23
- head, 21
- latex, 98
- less, 18
- link, 21
- ln, 21
- ls, 17
- mkdir, 16
- more, 17
- mount, 17
- move, 21
- mv, 21
- pico, 20
- pine, 20, 24
- pipe, 18, 23
- pwd, 14
- randapparaten, 17
 - mount, 17
- rechten, 22
- redirectie, 18, 23
- remove, 22
- rm, 22
- script
 - echo, 36
 - lees, 36
 - schrijf, 36
- scripts, 35
 - read, 36
 - variabele, 36
- shared libraries, 57
- shel, 35
- shell
 - bash, 35
 - csh, 35
 - ksh, 35
 - scripts, 35
 - sh, 35
- so, 58
- static libraries, 57
- tail, 21
- tar, 23
- touch, 20

uuencode, 24

vi, 20, 31

vim, 31

 modes, 31

 reguliere expressies, 32

 zoeken, 32

websites, 87

 distributies, 87

 programmeren, 88