Inleiding tot Unix

Peter Lucas
Instituut voor Informatica en Informatiekunde
Radboud Universiteit Nijmegen
E-mail: peterl@cs.ru.nl

27 september 2006

1 Inleiding

Dit document is een korte inleiding tot het gebruik van Unix-achtige bedrijfssysteem, zoals Linux en Solaris.

Een *bedrijfssysteem*, of operating system, verzorgt de communicatie tussen de computerhardware en de programmatuur, en zorgt ervoor dat de uitvoering van gebruikersprogramma's op ordelijke wijze verloopt. Daarnaast zorgt een modern bedrijfssysteem voor de afhandeling van de communicatie tussen computers die in een netwerk van computers zijn opgenomen. Het bestaat hiertoe uit een groot aantal systeemprogramma's.

In deze handleiding wordt slechts aan een klein deel van de functionaliteit van Unix aandacht besteed; de nadruk ligt op de behandeling van de Unix-commando's met behulp waarvan de gebruiker met het Unix-bedrijfssysteem communiceert. Deze commando's moeten door de gebruiker worden ingetikt. Sommige van deze commando's zijn programma's die kunnen worden uitgevoerd; andere commando's zijn geen programma's maar instructies aan het bedrijfssysteem om bepaalde operaties uit te voeren.

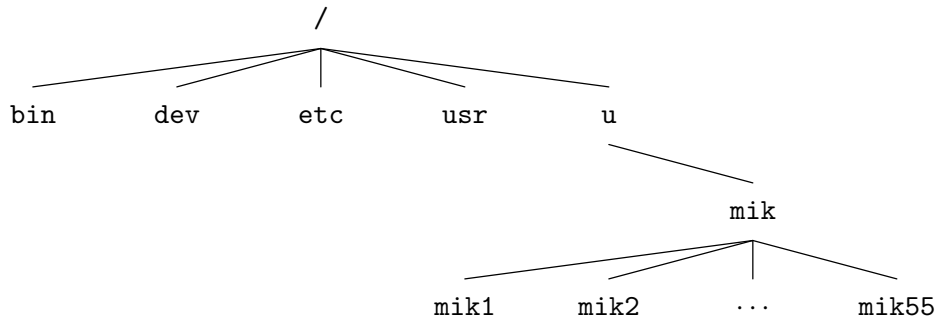
In deze handleiding zullen slechts de vaak toegepaste commando's worden besproken en zal een korte beschrijving worden gegeven van de editor 'vi'. De vi-editor gebruikt u voor het intikken en wijzigen van tekstbestanden. De belangrijkste commando's die u nodig heeft bij het gebruik van de vi-editor worden ook besproken. Tenslotte wordt kort aandacht besteed aan het gebruik van Unix als programmeeromgeving.

2 Inloggen

Vanuit de PC kan een verbinding worden gemaakt met één van Unix-servers. Elk van de werkstations heeft een naam, zoals wn5.science.ru.nl, etc. Op de PC is een putty ikoontje te zien, bijvoorbeeld met de tekst 'putty'; Door het aanklikken van deze icon wordt het communicatieprogramma 'putty' opgestart, dat verantwoordelijk is voor de verbinding tussen de PC en de Unix-server. Is de communicatie tot standgekomen dan kunt u inloggen, d.w.z. een gebruikersnaam (loginnaam) en password intikken waardoor u toegang krijgt tot het werkstation.

U kunt als volgt inloggen:

```
login: mik1
password: xwytzoiu
```



Figuur 1: Directory-boom.

Hierin is `mik1` de login-naam en `xwytzoiu` het password. Het password verschijnt niet op het scherm. Uw eigen login-naam (bijvoorbeeld mik1) en password is u door het systeembeheer verstrekt. Wanneer u inlogt, komt u automatisch in uw eigen home-directory terecht (hier dus `mik1`). Vervolgens zult u meestal programma's zoals 'mail' aanroepen, of met gebruik van 'vi' een bestand creëren of wijzigen. In de Paragraaf 4 wordt de vi-editor besproken.

3 Het Unix file-systeem

Van de meeste componenten van het Unix-bedrijfssysteem hoeft de gebruiker geen kennis te hebben. Er is echter één component waarmee u wel vertrouwd moet zijn; het zogenaamde *file-systeem*. Een file-systeem is een bepaalde organisatie van *bestanden* (*files*), die een afbeelding naar de fysieke hardware definieert. Wij zullen in deze handleiding een korte beschrijving geven van enkele aspecten van het Unix file-systeem vanuit het perspectief van de gebruiker.

Het file-systeem verzorgt de administratie en het onderhoud van de bestanden die door de gebruiker gecreëerd worden. Het Unix file-systeem is een zogenaamd *hiërarchisch* file-systeem; dat wil zeggen, de bestanden zijn gegroepeerd in eenheden, *directories* genoemd, die als knopen in een boom zijn gestructureerd. Het voordeel van de hiërarchische opzet van het file-systeem is dat de gebruiker de bestanden op onderwerp kan verdelen over verschillende directories. De gebruiker heeft de mogelijkheid met behulp van commando's de inhoud van de directories te inspecteren, en tevens van de ene naar de andere directory te gaan. In het volgende voorbeeld geven we weer hoe een deel van het file-systeem georganiseerd is (op de computer waarop u werkt is een en ander vermoedelijk niet helemaal hetzelfde).

Voorbeeld. Beschouw de directory-boom die weergegeven is in Figuur 1. Deze boom geeft een klein deel weer van de volledige directory-boom op het SUN-systeem. De knopen in de directory-boom stellen de directories voor. Zo is bijvoorbeeld `etc` de naam van een systeem-directory waarin een aantal bestanden voor algemeen gebruik opgenomen zijn. Vrijwel altijd bevat een directory één of meer bestanden. Aangezien iedere gebruiker een eigen directory heeft, wordt wel van een *home-directory* gesproken. De directory met naam `/` is de *wortel* van de directory-boom. De directories `bin`, `etc`, `dev`, `usr` en `u` zijn zogenaamde *subdirectories* van de wortel `/`. De directory `mik` is een subdirectory van de directory `u`. ■

Unix biedt de gebruiker een aantal commando's waarmee informatie over de directory-boom kan worden verkregen. Met behulp van het commando `pwd` (*p*rint *w*orking *d*irectory) kan

worden vastgesteld in welke directory gewerkt wordt.

Voorbeeld. Stel dat uw home-directory `mik1` is. Als u in deze directory het commando

```
pwd
```

intikt, krijgt u te zien:

```
/u/mik/mik1
```

Dit is het pad van van de directory waarin u zich bevindt (`mik1`) vanuit de wortel in de directory-boom. ■

Unix biedt ook de mogelijkheid stap voor stap de directory-boom te doorlopen. Het commando dat u in dat geval kunt gebruiken is `cd` (*c*hange *d*irectory).

Voorbeeld. Het `cd` commando kent diverse vormen. De belangrijkste vormen zullen we kort bespreken, uitgaande van de directory-boom weergegeven in Figuur 1. Veronderstel dat we in de directory `mik1` aan het werk zijn. Door het commando

```
cd ..
```

in te tikken, gaan we één directory omhoog in de directory-boom. De twee punten `..` verwijzen altijd naar de directory die de ouder is van directory waarin men werkt. We zijn dan in de directory `mik`. Het is ook mogelijk naar een subdirectory van een bepaalde directory te gaan, door de naam van de subdirectory als argument van `cd` op te geven. Bijvoorbeeld, met behulp van het commando

```
cd mik1
```

gaan we weer terug naar de directory `mik1`, die een subdirectory is van de directory `mik`. In het algemeen moet echter het volledige pad van de directory opgegeven worden. Wilt u bijvoorbeeld vanuit een willekeurige directory naar de directory `/u/mik/mik1` in de directory-boom, dan kunt u dat als volgt doen:

```
cd /u/mik/mik1
```

Via deze methode kan elke directory in de boom bereikt worden. ■

Elke gebruiker heeft de mogelijkheid zelf directories te creëren en te verwijderen. Het commando dat gebruikt wordt voor het creëren van een directory is `mkdir` (*m*a*k*e *dir*ectory); met behulp van het commando `rmdir` (*rem*ove *dir*ectory) kan een directory worden verwijderd.

Voorbeeld. Beschouw opnieuw de directory-boom weergegeven in Figuur 1. Veronderstel dat het commando

```
mkdir progs
```

wordt ingetikt in de directory `mik1`, dan zal de directory `progs` als subdirectory van `mik1` worden gecreëerd. Deze directory kan worden verwijderd door middel van het commando

```
rmdir progs
```

Het commando `rmdir` wordt echter dan en slechts dan uitgevoerd als de betreffende directory geen bestanden bevat. ■

Zoals we in het begin van deze paragraaf hebben beweerd, bevat een directory gewoonlijk bestanden. De namen van de bestanden in een directory kunnen zichtbaar gemaakt worden met behulp van het commando `ls`.

Voorbeeld. Veronderstel dat de directory `mik1` de bestanden `cursus1` en `cursus2` bevat. Door het intikken van `ls` wordt de inhoud van de directory `mik1` zichtbaar gemaakt:

```
ls
cursus1          cursus2
```

■

Bestanden kunnen worden gecreëerd met behulp van de vi-editor die in Paragraaf 4 besproken wordt, en door reeds bestaande bestanden te copiëren. Voor dit laatste kunt u het commando `cp` (*copy*) gebruiken.

Voorbeeld. Het `cp` commando kan op meerdere manieren gebruikt worden. We behandelen de meest voorkomende. Veronderstel dat we enkele veranderingen willen aanbrengen in het bestand `cursus1`. Door eerst het commando

```
cp cursus1 cursus1-copie
```

in te tikken blijft het oorspronkelijke bestand bewaard in `cursus1-copie`. Het is ook mogelijk bestanden naar een andere directory te copiëren. Veronderstel bijvoorbeeld dat we ons in de directory `etc` bevinden, en daar het bestand `holidays` aantreffen. Na intikken van het commando

```
cp holidays /u/mik/mik1
```

bevindt dit interessante bestand zich ook in de directory `mik1`, eveneens onder de naam `holidays`. Met `mik1` als working directory heeft

```
cp /etc/holidays .
```

hetzelfde resultaat. (De enkele punt `.` verwijst altijd naar de working directory.) ■

Bestanden kunnen worden verwijderd met behulp van het commando `rm`; u bent dan het bestand wel voor altijd kwijt!

Voorbeeld. Stel dat we het bestand `cursus2` willen verwijderen. Dit kan als volgt geschieden:

```
rm cursus2
rm: remove cursus2? y
```

Voordat een bestand verwijderd wordt, wordt dit nog eens expliciet aan de gebruiker gevraagd. In dit geval geeft de gebruiker met het antwoord '`y`' aan dat het bestand inderdaad verwijderd mag worden. ■

De vi-editor biedt de mogelijkheid tekstbestanden te inspecteren. Daarnaast kan met behulp

Commando	Verklaring
`cd <naam>`	Ga naar de directory *<naam>*.
`cp <naam1> <naam2>`	Copieer het bestand *<naam1>* naar *<naam2>*.
`ls`	Geef de namen van de bestanden in deze directory.
`mkdir <naam>`	Creëer de directory *<naam>*.
`more <naam>`	Geef een listing van bestand *<naam>* op het scherm.
`pwd`	Geef de naam van de huidige working directory.
`rm <naam>`	Verwijder het bestand *<naam>*.
`rmdir <naam>`	Verwijder de directory *<naam>*.
`vi <naam>`	Start de vi-editor voor het bestand *<naam>*.

Tabel 1: Enkele Unix-commando's.

van het commando `more` de inhoud van een bestand bekeken worden.

Voorbeeld. Veronderstel dat we de inhoud van het bestand `cursus1` willen bekijken. Dit kan met behulp van `more` als volgt geschieden:

```
more cursus1
```

Door gebruik te maken van de spatie-balk op het toetsenbord van de computer kan het bestand bladzijde voor bladzijde worden bekeken. ■

Een overzicht van de meest belangrijke Unix-commando's treft u aan in Tabel 1.

Oefenopgaven

De volgende opgaven zijn bedoeld om u enige ervaring met de meest eenvoudige filecommando's van Unix laten opdoen. Log eerst in, zoals in Paragraaf 2 is aangegeven, met behulp van een login-naam en password die u aan zijn uitgereikt.

▶ *Probeer achtereenvolgens de volgende Unix-commando's uit:*

1. `pwd`
2. `ls`
3. `cp /etc/holidays .`
4. `ls`
5. `more holidays`
6. `ls /usr/bin`

▶ *In de vorige opgaven was uw home-directory telkens uw working directory. Gewoonlijk worden bestanden echter gegroepeerd naar onderwerp bewaard in een aantal subdirectories.*

1. *Maak in uw home-directory een directory met de naam* `sub` *aan (*`mkdir sub`*). Hoe geeft het commando* `ls` *het onderscheid tussen bestanden en directories weer?*
2. *Maak* `sub` *tot uw working directory (*`cd sub`*); controleer met behulp van* `pwd`*.*
3. *Verplaats het bestand* `holidays` *naar de directory* `sub`*; controleer met behulp van* `ls`*.*

4. *Copieer nu vanuit de directory* `sub` *het bestand* `holidays` *naar de home-directory. Eén manier om dit te doen is:*

   ```
   cp holdidays ..
   ```

   Wat is hier de betekenis van de dubbele punt (..)?

5. *Ga terug naar uw home-directory en probeer de directory* `sub` *met behulp van het commando* `rmdir` *te verwijderen. Waarom lukt dit niet?*

6. *Verwijder nu het bestand met naam* `holidays` *in de directory* `sub`*, en probeer opnieuw de directory* `sub` *te verwijderen. Waarom lukt dit nu wel?*

4 Omgaan met de vi-editor

De editor vi (spreek uit 'vee-eye') is een full-screen editor. Een editor is een programma om files te creëren of te wijzigen. 'Full-screen' wil zeggen dat de resultaten van commando's direkt op het scherm getoond worden. De tekst op het scherm is dus altijd up-to-date. Het is in vi mogelijk met de cursor door die delen van het scherm te bewegen waar daadwerkelijk tekst staat. Het einde van een regel wordt beschouwd als het einde van de tekst. Indien de tekst minder regels bevat dan het aantal regels dat op het scherm past, beginnen de resterende regels onderaan het scherm met het symbool ~; deze regels behoren niet tot de tekst.

4.1 Gebruikte formalisme

Er wordt onderscheid gemaakt in het gebruik van hoofdletters en kleine letters, er is dus een verschil in interpretatie tussen *A* en *a*. Commando's van de vorm <*esc*> hebben betrekking op de toetsen met de aangeduide functie. Bijvoorbeeld, de toets met de functie <*esc*> treft u links boven op uw toetsenbord aan.

4.2 Twee modes in vi

Indien men een tekst wil creëren of wijzigen geeft men het commando `vi` <*naam*>. Binnen 'vi' zijn er twee 'modes', te weten:

- De COMMAND mode. In de COMMAND mode is het mogelijk om commando's te geven.
- De EDIT mode. In de EDIT mode is het mogelijk om tekst toe te voegen of te wijzigen.

Bij het opstarten van 'vi' komt men automatisch in de COMMAND mode. In de COMMAND mode zijn er verschillende soorten commando's:

- Commando's om de cursor over het scherm te verplaatsen (Tabel 2),
- Commando's voor het vervangen of verwijderen van tekst (Tabel 3), en
- Commando's om vi te verlaten (Tabel 4).

In het algemeen zal men een ingetikt commando **niet** op het scherm zien. Om in de EDIT mode te komen kan men uit de commando's kiezen die in Tabel 5 beschreven zijn. In de EDIT mode kan men de commando's die in Tabel 6 beschreven zijn gebruiken. Het is niet mogelijk om de cursor in de EDIT mode te verplaatsen over tekst die reeds aanwezig was voordat men

Commando	**Verklaring**
h, <*back-space*>, ←	Cursor een positie naar links in de tekst.
j, ↓	Cursor een regel naar beneden in de tekst.
k, ↑	Cursor een regel naar boven in de tekst.
l, <*spatie*>, →	Cursor een positie naar rechts in de tekst.
<*return*>	Naar het begin van de volgende regel.
−	Naar het begin van de vorige regel.
w	Naar het volgende woord.
b	Naar het voorgaande woord.
<*ctrl*>f	Een pagina naar beneden in de tekst.
<*ctrl*>b	Een pagina naar boven in de tekst.

Tabel 2: Commando's om de cursor over het scherm te verplaatsen.

Commando	**Verklaring**
r[*karakter*]	Vervang het karakter dat onder de cursor staat door [*karakter*].
x	Verwijder het karakter op de plaats van de cursor.
dw	Verwijder het woord dat op de plaats van de cursor begint.
[*aantal*]dd	Verwijder [*aantal*] regels, te beginnen bij de regel waarop de cursor staat. Indien [*aantal*] wordt weggelaten dan wordt alleen de regel waarop de cursor staat verwijderd.

Tabel 3: Commando's voor het vervangen of verwijderen van tekst.

Commando	**Verklaring**
:wq, ZZ	Verlaat vi en bewaar de tekst in de huidige toestand.
:q!	Verlaat vi, maar bewaar de veranderingen niet.

Tabel 4: Commando's om vi te verlaten.

Commando	**Verklaring**
i	Voeg tekst vóór de huidige plaats van de cursor toe.
I	Voeg tekst vóór het begin van de huidige regel toe.
a	Voeg tekst achter de huidige plaats van de cursor toe.
A	Voeg tekst achter het einde van de huidige regel toe.

Tabel 5: Commando's waardoor men in EDIT-mode komt.

Commando	**Verklaring**
<*back-space*>	Ga een positie naar links en wis het karakter.
<*return*>	Ga verder op de volgende regel. Indien men midden in een bestaande regel <*return*> geeft, wordt de regel gesplitst; het deel na de cursor komt dan op de volgende regel.
<*esc*>	Verlaat de EDIT mode.

Tabel 6: Commando's die in EDIT-mode gebruikt worden.

Commando	**Verklaring**
:w	Bewaar de tekst in de huidige toestand (zonder vi te verlaten).
:set number	Plaats regelnummers voor de regels (Alleen op het scherm!)
:set nonumber	Verwijder de regelnummering.
[*nummer*]G	Ga naar regel [*nummer*] in de tekst; indien [*nummer*] niet gespecificeerd wordt zal naar de laatste regel gegaan worden.
J	Zet de regel onder de huidige regel achter de huidige regel.
cw	Vervang het woord dat op de plaats van de cursor begint; als het vervangende woord is ingetikt, geeft men met <*esc*> aan dat men klaar is (EDIT-mode).
[*aantal*]Y	Copieer [*aantal*] regels, te beginnen bij de regel waarop de cursor staat. Indien [*aantal*] wordt weggelaten dan wordt alleen de regel waarop de cursor staat gecopieerd.
p	Voeg de laatst gecopieerde of verwijderde tekst toe achter de huidige plaats van de cursor of onder de huidige regel. (Dit is afhankelijk van de tekst.)
u	Maak de laatste wijziging ongedaan.
<*ctrl*>l	Ververs het scherm.

Tabel 7: Additionele commando's van vi.

de EDIT mode inging. Bevindt men zich in de EDIT mode, dan keert men met het commando <*esc*> terug in de COMMAND mode. (Bij twijfel enkele malen <*esc*> geven zodat zeker is dat men zich in de COMMAND mode bevindt.)

De bovenbeschreven commando's zult u frequent nodig hebben. In Tabel 7 is een aantal commando's opgenomen die u niet direct nodig hebt, maar die u het werken met 'vi' zullen veraangenamen. Tot slot wordt beschreven hoe u tekst kunt copiëren en verplaatsen. Tussen het verplaatsen en copiëren van tekst bestaat een sterke analogie. De actie bestaat uit drie stappen.

Stap 1: **Copiëren:** met [*aantal*]Y wordt aangegeven, beginnend met de regel waarop de cursor staat en naar beneden tellend, hoeveel regels er gecopieerd moeten worden; als [*aantal*] = 1 dan mag dit weggelaten worden.
Verplaatsen: met [*aantal*]dd wordt aangegeven, beginnend met de regel waarop de cursor staat en naar beneden tellend, hoeveel regels er verplaatst moeten worden; als [*aantal*] = 1 dan mag dit weggelaten worden.

Stap 2: Vervolgens verplaatst men de cursor naar de regel waaronder men het blok tekst wil plaatsen.

Stap 3: Door middel van het commando *p* geeft men vervolgens aan dat het blok na de regel waar de cursor staat, geplaatst moet worden.

De laatste twee acties, het aanwijzen van de regel en *p*, kunnen meermalen achter elkaar herhaald worden, zodat verschillende copieën van het blok gemaakt kunnen worden.

Oefenopgaven

De volgende opgaven zijn bedoeld om u enige ervaring met het Unix file-systeem en de vi-editor te laten opdoen. U zult voor het maken van de oefenopgaven die op deze paragraaf volgen, de vi-editor voldoende moeten beheersen.

▶ *Creëer met behulp van de vi-editor een tekstbestand met de naam* `test`*:*

```
vi test
```

U tikt <return> *in na het woord* `test`*.*

1. *Breng de editor in de* EDIT *mode.*
2. *Tik een paar zinnen in, en probeer delen van de laatste zin te wijzigen.*
3. *Ga nu over naar* COMMAND *mode.*
4. *Copieer een aantal woorden en zinnen.*
5. *Oefen in het verwisselen van karakters (*`x p`*) en zinnen (*`dd p`*).*
6. *Oefen met het* undo *(*`u`*) commando.*

Door voldoende tekst aan te maken kunt u ook de andere commando's (zoals het 'bladeren' door het bestand) uitproberen. Verlaat tenslotte de editor.

N.B.: Tijdens het editen bent u niet het oorspronkelijke bestand aan het wijzigen, maar een copie ervan. Pas als u de inhoud van het bestand wegschrijft (met `:w`, `:wq` of `ZZ`), zijn de wijzigingen in het oorspronkelijke bestand doorgevoerd.

▶ *Probeer achtereenvolgens de volgende Unix-commando's uit:*

1. `ls`
2. `cp test test2`
3. `ls`
4. `mv test test1`
5. `ls`
6. `more test1`
7. `rm test1`
8. `ls`

5 De Unix C-shell

In de voorgaande paragrafen heeft u enkele eenvoudige commando's leren kennen met behulp waarvan gecommuniceerd kan worden met het Unix-bedrijfssysteem. De meeste van deze commando's zijn aparte programma's die elementaire opdrachten doorgeven aan de kern van het bedrijfssysteem (de zogenaamde Unix *kernel*).

Nadat men een commando heeft ingetikt zal eerst onderzocht moeten worden of de syntaxis van dit commando correct is. Vervolgens wordt onderzocht of het commando bestaat, en pas daarna kan het worden uitgevoerd. Bij het uitvoeren wordt bijvoorbeeld nagegaan of de

Commando	Verklaring
alias	verkorte naam definiëren voor commando
cd	door directory-boom lopen
history	geschiedenis van ingevoerde commando's opvragen
echo	waarde van variabelen, of string afdrukken op scherm
jobs	overzicht geven van alle processen (jobs)
set	variabele een waarde geven
setenv	omgevingsvariabele een waarde geven
kill	een bepaald proces (job) afbreken

Tabel 8: Enkele C-shell commando's.

noodzakelijke invoerbestanden bestaan. Al deze stappen worden verricht door een programma dat een laag vormt tussen de gebruiker en de kernel, de zogenaamde *commando-interpretator* (command interpreter). Omdat de commando-interpretator als het ware een schil vormt rond de kern van Unix, wordt gewoonlijk gesproken van de *command shell*, of ook wel kortweg van de shell. Er is een aantal verschillende shells beschikbaar onder Unix.

Zodra men heeft ingelogd wordt een shell opgestart. De meest gebruikte shell (maar niet noodzakelijkerwijs de beste) is de zogenaamde *C-shell*; de naam verwijst naar het feit dat de commando's doen denken aan de programma-constructen die men beschikbaar heeft in de programmeertaal C. Andere veel gebruikte shells zijn de *Bourne shell* en de *Korn shell*.

Een shell is gewoon een programma dat wacht op gebruikersinvoer en ingevoerde commando's van de gebruiker verwerkt. Als een commando een programma is, zal de shell dit programma laten uitvoeren en mogelijkheden geven aan de gebruiker de uitvoering tijdelijk te stoppen, af te breken, verder te laten gaan, etc. Een programma dat wordt uitgevoerd wordt wel een *proces* of een *job* genoemd. Elk proces in Unix krijgt een aantal *procesidentifiers* waarmee het gekarakteriseerd wordt. Wij zullen hier straks verder op in gaan. Zoals alle programma's onder Unix correspondeert elke shell met een (niet-leesbaar) bestand. Bijvoorbeeld, de C-shell is opgeslagen in het bestand `/bin/csh`.

In Tabel 8 is een aantal veelgebruikte C-shell commando's weergegeven. Elk commando heeft een aantal *argumenten*; deze argumenten zijn òf *opties* (meestal voorafgegaan door een `-` teken), die de mogelijkheid geven de uitvoering van het commando op een bepaalde manier te laten plaatsvinden, òf *operanden*, de invoer voor het commando. De argumenten zijn in de tabel niet weergeven. Wij zullen enkele van de meest gebruikte C-shell commando's kort behandelen.

5.1 Alias

Voor commando's die men vaak gebruikt kan met behulp van het commando `alias` een andere, gewoonlijk kortere, naam worden opgeven. Men hoeft dan niet elke keer het gehele commando in te tikken.

Voorbeeld. Veronderstel dat we het bestand `cursus1` op de laserprinter willen afdrukken. Het commando dat we in dat geval moeten gebruiken is: `lpr -Ppal test`. Door nu de volgende alias te definiëren:

```
alias print lpr -Ppal
```

is het vervolgens mogelijk het bestand `test` op de printer af te drukken door `print test` in te tikken. ■

5.2 History

De C-shell biedt de mogelijkheid de laatste commando's die men heeft ingetikt te 'onthouden', zodat een bepaalde commando weer kan worden opgeroepen. Men spreekt in dit verband van de *command history*, omdat als het ware een 'geschiedenis' van ingevoerde commando's wordt bewaard. Men moet wel van te voren opgeven hoeveel commando's onthouden moeten worden; in ons geval is opgegeven dat dit er maximaal honderd zijn. Deze commando's kunnen met behulp van het commando `history` zichtbaar worden gemaakt.

Voorbeeld. Veronderstel dat wij de laatste commando's willen bekijken, om er eventueel één van te gebruiken. Hiertoe tikken wij in

```
history
```

Op het scherm zou bijvoorbeeld het volgende genummerde overzicht van eerder ingevoerde commando's zichtbaar gemaakt kunnen worden:

```
1  pwd
2  cd ..
3  cd mik1
4  cd /u/mik/mik1
5  mkdir sub
6  rmdir sub
7  ls
8  cp test test-copie
9  cd /etc
10  cp hollidays /u/mik/mik1
11  cd ~
12  cp /etc/holidays .
13  rm test-copie
14  more test
```

Merk overigens op dat het handig zou zijn het commando `history` met het hierboven besproken commando `alias` een kortere naam te geven bijvoorbeeld de naam `h`:

```
alias h history
```

■

Het is nu eenvoudig mogelijk uit de gepresenteerde history een commando uit te kiezen, en opnieuw te laten uitvoeren. Hiertoe wordt gebruik gemaakt van een `!`-commando.

Voorbeeld. Beschouw het vorige voorbeeld. We willen nu sommige van deze commando's opnieuw gebruiken. Met behulp van het `!`-teken kunnen wij deze oproepen. Door nu in te tikken:

```
!14
```

wordt het veertiende ingevoerde commando opnieuw uitgevoerd, dus

```
more test
```

Merk op hoe handig dit is; het scheelt aanzienlijk in de hoeveelheid typewerk. Er is ook een tweede manier om dit commando uit te laten voeren zonder het opnieuw volledig in te tikken, namelijk door

```
!!
```

in te tikken. Hierdoor wordt het *laatst* ingevoerde commando herhaald, in dit geval opnieuw commando 14. ■

Naast de hierboven besproken manier om commando's die bewaard zijn gebleven in de history opnieuw uit te laten voeren, is het tevens mogelijk een commando te selecteren op grond van bepaalde karakters die erin voorkomen. Wij illustreren dit met een voorbeeld.

Voorbeeld. Beschouw opnieuw de afgedrukte history in het op één na laatste voorbeeld. Stel dat we nu commando 6 (`rmdir prog`) willen herhalen, dan kan dit ook als volgt:

```
!rmd
```

Hiermee wordt het laatste commando dat met de karakters `rmd` begint herhaald. Merk op dat de invoering van

```
!rm
```

tot gevolg zou hebben dat commando 13 zou worden herhaald. Men hoeft slechts zoveel beginkarakters in te tikken als voor de selectie van het juiste commando noodzakelijk is. ■

5.3 Omgevingsvariabelen

Als de gebruiker eenmaal ingelogd is, maakt Unix voor de interactie met de gebruiker gebruik van specifieke informatie die vastgelegd is in zogenaamde *omgevingsvariabelen*, ook wel *shell-variabelen* genoemd. Zo 'weet' Unix op grond van de waarde van de variabele `term` van wat voor soort beeldscherm gebruik gemaakt wordt. Dit is van groot belang, want elk type beeldschem wordt door verschillende stuurkarakters aangestuurd. De waarde van een omgevingsvariabele kan zichtbaar gemaakt worden door middel van het commando `set`.

Voorbeeld. De waarde van de variabele `term` kan als volgt worden verkregen:

```
set term
```

Als u direct ingelogd ben op een werkstation, zal de waarde van de variabele `term` gewoonlijk `sun` zijn; bent u ingelogd via de MacIntosh, dan zal de waarde gewoonlijk `vt100` zijn. Dit zijn verschillende typen beeldschermen waarvan de besturing inderdaad niet gelijk is. Unix biedt ook een twee manier om de waarde van een variabele zichtbaar te maken, namelijk met behulp van het commando `echo`, waarbij nu de naam van de variabele voorafgegaan moet worden door een `$`-karakter:

```
echo $term
```

■

Een andere belangrijke omgevingsvariabele is `path`. Deze variabele heeft als waarde een lijst van directories waarin gezocht wordt als u een naam van een programma intikt. Als de naam van de directory van een bepaald programma dat u wilt gebruiken niet in de variabele `path` opgeven is, kunt het niet als u in uw home-directory bent aanroepen. U zult dan de volledige naam van het pad moeten opgeven.

Voorbeeld. Beschouw de volgende waarde van de variabele `path`:

```
path = (. /bin /usr/bin /usr/ucb /etc)
```

Deze specificatie houdt in dat als u een programma aanroept, eerst in uw huidige directory gezocht wordt (dit is de directory `.`); als het programma daar niet gevonden wordt, dan zoekt het systeem in directory `/bin`, etc. Stel nu dat u een programma wilt gebruiken, dat in directory `/usr/local/bin` opgenomen is, bijvoorbeeld de tekstverwerker LaTeX. Als u nu

```
latex
```

intikt, krijgt u als reactie

```
latex: Command not found.
```

De enige manier om het programma nu toch aan te roepen is door het volledige pad op te geven:

```
/usr/local/bin/latex
```

Dit is niet erg handig. ■

Met behulp van het commando `set` kunt u een variabele een (al dan niet nieuwe) waarde geven.

Voorbeeld. Veronderstel dat we de variabele `path` een nieuwe waarde willen geven, dan kan dat als volgt gebeuren:

```
set path = (. /bin /usr/bin /usr/ucb /etc /usr/local/bin)
```

Het gevolg van de toekenning van deze nieuwe waarde is dat de programma's in de directory `/usr/local/bin` nu wel kunnen worden aangeroepen in elke directory. ■

5.4 De .cshrc en .login file

In de vorige paragraaf hebben we aandacht besteed aan omgevingsvariabelen, en aan het belang daarvan. Nu is het niet erg aantrekkelijk om elke keer nadat u hebt ingelogd de waarden van deze variabelen te moeten intikken. Gelukkig is dit niet noodzakelijk. Er is in Unix in elke home-directory een tweetal bestanden aanwezig waarin alle belangrijke variabelen alvast een waarde krijgen toegekend. De namen van deze bestanden zijn:

```
.login
```

en

`.cshrc`

Merk op dat de namen van deze bestanden met een `.` beginnen (dit is niet een verwijzing naar de huidige directory, want dan zou de naam met `./` moeten beginnen). De commando's die in beide bestanden zijn gespecificeerd worden na elkaar door de C-shell na het inloggen verwerkt. Pas na de verwerking kunt uzelf iets intikken. Bestanden met dit soort commando's worden wel *scripts* genoemd.

Overigens is het niet mogelijk de naam van deze bestanden met behulp van het commando `ls` zichtbaar te maken. Bestanden waarvan de naam met een punt beginnen, laat dit commando niet zondermeer zien. Als u echter het commando

```
ls -a
```

of het commando

```
ls -la
```

intikt, krijgt u de namen van bestanden die met een punt beginnen wel te zien. U kunt deze bestanden met `vi` of `more` gewoon onderzoeken. In het file .cshrc staat bijvoorbeeld het commando `set history=100`. In de vorige paragraaf hebben reeds besproken dat dit commando tot gevolg had dat maximaal de laatste 100 ingetikte commando's bewaard werden. ■

5.5 Jobs, processen en job-controle

Elk programma dat onder Unix wordt uitgevoerd wordt een *job* of *proces* genoemd. De naam 'job' werd reeds bij de eerste bedrijfssystemen gebruikt; deze naam geeft aan dat de uitvoering als een 'klusje' van het bedrijfssysteem wordt opgevat. Het bedrijfssysteem houdt hierbij nauwkeurig bij of deze uitvoering wel naar behoren verloopt.

Gewoonlijk is een job een programma dat wordt uitgevoerd nadat u het corresponderende commando heeft ingetikt. In dat geval is er sprake van een directe interactie tussen programma en gebruiker. Men noemt zo'n programma *interactief* en zegt dat het programma in de *voorgrond* (foreground) uitgevoerd wordt. U kunt het programma dan gewoonlijk onderbreken door het controle-karakter

```
^C
```

in te tikken. Het is echter ook mogelijk een job niet-interactief te laten uitvoeren. Dit is uiteraard alleen mogelijk als het programma geen invoer verwacht van de gebruiker (maar bijvoorbeeld wel van een invoerbestand). Men noemt deze vorm van uitvoeren wel executie op de *achtergrond* (background). Vroeger was deze vorm van uitvoeren van een programma voor de gebruiker belangrijker dan tegenwoordig; men had meestal slechts één scherm, en de uitvoering van een programma duurde al snel een half uur. In plaats van een half uur te wachten totdat verder gewerkt kon worden, kon men ook het betreffende proces in de achtergrond plaatsen. Met de huidige snelle computers is de noodzaak hiervoor wat minder groot. Wel is het zo dat op elk werkstation een groot aantal systeemprocessen in de achtergrond wordt uitgevoerd. Dit betreft over het algemeen processen die nooit stoppen.

De gebruiker kan een bepaald proces van de voorgrond in de achtergrond plaatsen door het controle-karakter

Commando	Betekenis
`jobs`	laat de status van alle gebruikersjobs zien.
`fg <`*jobgetal*`>`	laat job <*jobgetal*> in de voorgrond lopen.
`bg`	laat de huidige job in de achtergrond uitvoeren.
`^Z`	stop een job tijdelijk.
`^C`	afbreken van een proces op de voorgrond.
`kill <`*pid*`>`	breek job met PID <*pid*> af.

Tabel 9: Commando's voor job-controle.

`^Z`

in te tikken. De job is nu gestopt. Als nu vervolgens

`bg`

(van bachground) wordt ingetikt, dan is het proces in de achtergrond geplaatst. Het is ook mogelijk een bepaalde job direct in de achtergrond te plaatsen door het commando te laten volgen door het karakter `&`.

Voorbeeld. Veronderstel we hebben een (niet-interactieve) rekenjob met de naam `reken`. Tijdens de uitvoering van dit programma willen we gebruikmaken van de teksteditor. Het programma `reken` kan nu als volgt in de achtergrond worden geplaatst:

```
reken &
```

Unix retourneert nu een natuurlijk getal dat het *jobgetal* (job number) heet.

Met behulp van het commando `jobs` kan een overzicht verkregen worden van alle jobs die op een bepaald moment worden uitgevoerd ('runnen'). In Tabel 9 zijn de belangrijkste commando's voor job-controle weergegeven. Naast een eenvoudig jobgetal wordt aan een job ook een *proces-identificatiegetal* (PID) gegeven. Dit nummer kan worden gebruikt om een bepaald proces te stoppen. Hiertoe moet men uiteraard eerst achter de PID's van alle processen zien te komen. Hiertoe kan men gebruikmaken van het commando `ps` (p*rocess* s*tatus*). Aan de hand van enkele voorbeelden zal de werking van dit commando uitgelegd worden.

Voorbeeld. Als u het commando

```
ps -x
```

intikt terwijl u geen enkele job in de achtergrond heeft geplaatst, zou u bijvoorbeeld het volgende op uw scherm kunnen zien:

```
  PID  TT  STAT   TIME  COMMAND
28359  p5  S      0:01  -csh (csh)
28449  p5  R      0:00  ps
```

Hierin is

`PID` het proces-identificatiegetal
`TT` de naam van de verbinding met de terminal

`STAT` status van het proces (S is 'Stopped', R is 'Running')
`TIME` de hoeveelheid CPU-tijd die tot nu toe door het proces verbruikt is
`COMMAND` de naam van het commando

U ziet dat zelfs de C-shell (csh) als een gewoon proces wordt beschouwd. ■

U kunt een volledig overzicht verkrijgen van alle processen (niet alleen van uzelf) die door de computer uitgevoerd worden door

```
ps -ax
```

in te tikken. Als u daar nog de optie `u` aan toevoegt (dus `ps -aux`) worden tevens de namen van de gebruikers gegeven die de processen hebben opgestart.

Soms voor het voor dat bepaalde processen in de achtergrond afgebroken moeten worden, bijvoorbeeld omdat het proces in een oneindige lus geraakt is. Hiertoe heeft men het commando `kill` ter beschikking. Met kill kunnen alleen de eigen processen worden afgebroken.

Voorbeeld. Stel dat het proces `prolog` dat in de achtergrond wordt uitgevoerd moet worden afgebroken. Hiertoe moet eerst het proces-identificatiegetal van dit proces worden opgevraagd met behulp van het commando `ps -x`. We krijgen de volgende uitvoer te zien op het scherm:

```
 PID TT STAT  TIME COMMAND
8396 d0 T     0:00 prolog
8398 d0 R     0:00 ps -x
```

Door nu vervolgens in te tikken

```
kill -9 8396
```

wordt het proces met nummer 8396 (het prolog-proces) afgebroken. ■

5.6 Invoer, uitvoer en I/O redirection

Bij het starten van een programma wordt standaard de invoer geassocieerd met het toetsenbord en de uitvoer met het beeldscherm, althans als geen operand (bijvoorbeeld een bestandsnaam) als argument wordt meegegeven.

Voorbeeld. Het commando `cat` is een eenvoudig programma dat de inhoud van een bestand (meestal met tekst) op het beeldscherm zichtbaar maakt. In tegenstelling tot het commando `more` laat `cat` niet telkens één pagina zien, maar drukt het gehele bestand af op het beeldscherm, zonder op gebruikersinvoer te wachten. Stel dat u in uw directory een bestand met naam `cursus1` heeft, dan kunt u dit bestand als volgt op het beeldscherm zichtbaar maken:

```
cat cursus1
```

Zou u echter slechts het commando

```
cat
```

intikken, dan verwacht het programma `cat` invoer van het toetsenbord. Stel dat u die invoer vervolgens intikt. De invoer moet vervolgens afgesloten worden door middel van het controlekarakter `^D`, dat standaard een bestand afsluit (*End Of File* karakter, kortweg EOF). ■

Overigens wordt in Unix zowel het toetsenbord als het beeldscherm behandeld alsof het een bestand is (dit bestand is zelfs in de directory-boom te vinden, namelijk in de directory `/dev`).

Enkele Unix-commando's kunnen als argument een bestandsnaam nemen, en verwerken dan dit bestand. Soms is zo'n programma slechts in staat invoer te accepteren van het toetsenbord en uitvoer door te sturen naar het beeldscherm. Unix biedt dan toch de mogelijkheid als invoer- en uitvoerbestanden bestanden in de directory-boom op te geven, namelijk door gebruik te maken van een zogenaamde *redirect*. Een redirect is als het ware een mogelijkheid om de in- en uitvoer om te leiden. Men maakt hierbij gebruik van de karakters `<` en `>`. We zullen dit aan de hand van een voorbeeld illustreren.

Voorbeeld. Veronderstel dat we een directory listing willen hebben, echter niet afgedrukt op het scherm maar in een file die we listing noemen, dan kan dat als volgt gebeuren:

```
ls > listing
```

Terwijl de uitvoer van het commando `ls` gewoonlijk naar het beeldscherm wordt gestuurd, wordt de uitvoer nu door de redirection `>` in het (nieuwe!) bestand `listing` opgeslagen. U kunt nu met behulp van het commando `more` (of `cat`) de inhoud van dit bestand inspecteren:

```
more listing
```

■

De `>`-vorm van redirection schrijft de uitvoer van het betreffende programma naar een nieuw bestand. Soms is het echter noodzakelijk de invoer aan het einde van een reeds bestaand bestand toe te voegen. In dat geval kan men gebruik maken van de `>>`-vorm van redirection.

Voorbeeld. Veronderstel dat u een twee listing van een directory aan het eerder gecreëerde bestand `listing` wilt toevoegen, zodat u beide listings kunt vergelijken. Dit kunt u als volgt doen:

```
ls >> listing
```

■

Tenslotte merken we op dat de `<`-vorm van redirection gebruikt kan worden om de invoer in plaats vanaf het toetsenbord, plaats te laten vinden vanuit het meegegeven bestand.

5.7 De Unix pipeline

Vaak is het handig om de uitvoer van het ene programma te gebruiken als invoer voor een volgend programma. Een manier om dit te realiseren is met behulp van redirection de uitvoer van het ene programma te laten wegschrijven in een bepaald bestand (`>`-redirection) en hetzelfde bestand als invoerbestand te gebruiken voor het andere programma (`<`-redirection). Dat is echter nogal omslachtig.

Een elegantere manier om de uitvoer van het ene programma als invoer te laten fungeren van het andere programma is en zogenaamde *pijp* (pipe), gesymboliseerd door het `|`-karakter (horizontal bar). In het algemeen ziet zo'n pijp er als volgt uit:

```
prog1 | prog2
```

waarin `prog1` en `prog2` twee willekeurige programma's zijn. De uitvoer van programma `prog1` is nu de invoer van `prog2`. Zo'n specificatie wordt wel een *pijplijn* (pipeline) genoemd. Uiteraard kan men ook meer dan twee programma's op deze manier met elkaar verbinden. In de volgende pijplijn

```
prog1 | prog2 | prog3
```

wordt de invoer van programma `prog1` via `prog2` uiteindelijk invoer van `prog3`. Programma's die op zo'n manier werken worden wel *filters* genoemd, omdat ze de uitvoer van een ander programma als het ware kunnen 'filteren'.

Voorbeeld. We willen weten hoeveel files en directories er in onze home-directory staan. Dat kan op volgende manier door gebruik te maken van het programma `wc` (*w*ord *c*ount) dat het aantal regels, woorden en karakters in een bestand bepaalt.

```
ls | wc
```

Stel dat de uitvoer er als volgt uitziet:

```
191  183  1593
```

dan moet hier de volgende betekenis aan worden toegekend: de uitvoer van `ls` bestaat uit 191 regels, 183 woorden en 1593 karakters.

Pipes en redirection kunnen ook gecombineerd worden.

Voorbeeld. Beschouw de volgende pijp:

```
ls | wc > wordcount
```

De uitvoer van de pijp wordt opgeslagen in het bestand `wordcount`. ■

5.8 Filenaam-expansie

Bij de meeste commando's kan men zogenaamde *wildcards* gebruiken om niet de volledige naam van een bestand te hoeven intikken. Unix kent drie van deze wildcards: `?`, `*` en `~`. Als men een naam van een bestand als operand van een commando meegeeft met daarin de `?` wildcard verwerkt, dan zal het commando (als dat mogelijk is) op elk van de bestanden worden uitgevoerd waarvan de naam correspondeert met de opgegeven naam, waarbij voor `?` een willekeurig ander karakter mag worden ingevuld. In het geval van de `*` wildcard mag voor het sterretje nul of meer karakters worden ingevuld.

Wildcards zijn erg handig als men veel bestanden in een directory heeft en bijvoorbeeld met het commando `ls` alleen een bepaalde groep wil zien. Wildcards kunnen echter in principe met elk commando worden gecombineerd, maar het gebruik is niet altijd zinvol.

Voorbeeld. Veronderstel dat in ons directory volgende files staan: `te`, `test1`, `test2`, `abc`, `bcd`, `efg`. We willen een listing volgens een bepaald criterium:

`ls *`	Laat alle bestanden in de directory zien (zelfde als kortweg `ls`).
`ls ?e`	Laat alle bestanden zien met naam bestaande uit twee karakters, waarvan de tweede letter een `e` is.
`ls ?e*`	Laat alle bestanden met als tweede letter e zien.
`ls *[1-3]`	Laat alle bestanden zien die eindigen op 1, 2 of 3.
`ls t,a*`	Laat alle bestanden zien die met `t` of `a` beginnen.

■

Er is nog een ander wildcard, die naar de home-directory verwijst namelijk `~`.

Voorbeeld. Het commando

```
cd ~mik1
```

is gelijk aan het commando

```
cd /u/mik/mik1
```

■

Oefenopgaven

De volgende opgaven zijn bedoeld om u enige ervaring met de C-shell te laten opdoen.

► *Experimenteer met het commando* `alias`*:*

1. *Onderzoek welke* `alias` *definities voor uw huidige login-sessie gelden.*
2. *Definieer met behulp van het commando* `alias` *een kortere naam voor het commando* `lpr -Ppal` *(met behulp waarvan u een bestand kunt afdrukken op de practicum-laserprinter).*

► *Experimenteer met de command history:*

1. *Onderzoek met behulp van het commando* `history` *welke commando's u inmiddels heeft ingetikt.*
2. *Probeer enkele van de reeds ingetikte commando's met behulp van het* `!`*-commando te herhalen. Wat is de betekenis van het commando* `!!`*?*

► *De omgevingsvariabelen tijdens een login-sessie bevatten alle informatie met betrekking tot terminalbesturing, loginnaam, directories waarin gezocht moet worden naar programma's als een commando wordt ingetikt, etc.*

1. *Onderzoek met behulp van het commando* `set` *welke waarden de omgevingsvariabelen bij u hebben.*
2. *Probeer vervolgens de waarde van de variabele* `term` *te veranderen in* `mac`*, en onderzoek het effect hiervan. Probeer met name met behulp van 'vi' een bestand te wijzigen. Verander vervolgens de waarde van de variabele* `term` *weer terug in de oorspronkelijke waarde.*

► *Unix biedt verschillende commando's om als gebruiker processen op te starten, af te breken en op de achtergrond te laten uitvoeren.*

1. *Onderzoek met behulp van het commando*

   ```
   ps -ax
   ```

   welke processen uitgevoerd worden op de computer waarop u ingelogd bent.
2. *Probeer nu vervolgens het proces* `reken` *op de achtergrond op te starten. U kunt dit als volgt doen:*

```
reken &
```

3. *Onderzoek met behulp van het commando* `jobs` *wat het jobgetal is van dit proces.*
4. *Onderzoek met behulp van het commando* `ps -x` *wat de procesidentifiers zijn van dit proces (in het bijzonder het proces-identificatiegetal, PID, want dat heeft u nodig om het proces af te breken).*
5. *Probeer het proces weer naar de voorgrond te halen met behulp van het commando*

   ```
   fg
   ```

6. *Stop het proces met behulp van* `^Z`*, en zorg ervoor dat het proces verder uitgevoerd wordt in de achtergrond (m.b.v.* `bg`*).*
7. *Maak nu vervolgens gebruik van het commando* `kill` *om het proces* `reken` *af te breken.*

6 Programma-ontwikkeling onder Unix

Unix is een bedrijfssysteem dat vooral bekend geworden is omdat het een bijzonder krachtige en flexibele omgeving is voor de ontwikkeling van programmatuur. In dit verband wordt Unix wel een *programmeeromgeving* genoemd. Net zoals elk bedrijfssysteem, biedt Unix een aantal compilers en interpretators voor een aantal verschillende programmeertalen. In het bijzonder worden de talen C, C++ en Pascal ondersteund. Daarnaast biedt Unix ook nog een groot aantal hulpprogramma's (*utilities*) die bij de ontwikkeling van programma's nuttig kunnen zijn. In het korte bestek van deze handleiding is het helaas niet mogelijk om aan al deze hulpprogramma's aandacht te besteden. We volstaan met een zeer sterk vereenvoudigde behandeling van het hulpprogramma `make`. Voordat we hier echter op ingaan, zal eerst kort aandacht besteed worden aan het gebruik van de Unix Pascal-compiler.

6.1 De Pascal compiler en link editor

6.1.1 De Pascal compiler

Unix biedt geen standaard-compiler voor Pascal. De meeste versies van Unix ondersteunen natuurlijk wel de ANSI-standaard van Pascal, maar ten aanzien van de extensies op deze standaard zijn er grote verschillen. Voorbeelden van gebruikelijke extensies zijn: een module-concept (soms ook wel units genoemd), een string datatype, padding van character array's, procedures voor het openen, onderzoeken van het bestaan en sluiten van bestanden, macro's, etc.

De versies van Unix die overeenkomen met Unix 4.3BSD, waaronder SUNOS, hebben meestal een compiler die afgeleid is van de Berkeley Pascal compiler. Deze compiler is de traditionele Pascal compiler voor Unix. Helaas wordt deze compiler bij andere Unix-achtige bedrijfssystemen (zoals AIX van IBM en Unix System V van onder andere HP) niet ondersteund, zodat Pascal programma's die op de SUN ontwikkeld zijn meestal aangepast moeten worden voor andere versies van Unix. Deze situatie verklaart waarom onder Unix in toenemende mate gebruik gemaakt wordt van de programmeertaal C, waarvan de programma's veel eenvoudiger uit te wisselen zijn tussen de diverse versies van Unix.

De Pascal-compiler die beschikbaar is onder SUNOS kan men aanroepen door het commando

`pc <`*filenaam*`>`

```
program CopyProg(input, output);

const EOF = -1;
      NL  = 10;

type character = -1..127;

function getc(var c : character) : character;  { one char from standard input }

var ch : char;

begin
  if eof then c := EOF
  else if eoln then
  begin
    readln;
    c := NL
  end
  else begin
    read(ch);
    c := ord(ch)
  end;
  getc := c
end;  { getc }

procedure putc(c : character); { put one char to standard output }

begin
  if c = NL then writeln
  else write(chr(c))
end;  { putc }

procedure copy; { copy input to output }

var c : character;

begin
  while getc(c) <> EOF do
    putc(c)
end;  { copy }

begin { main }
  copy
end.
```

Figuur 2: Programma voor het copiëren van een bestand.

in te tikken, waarin <*filenaam*> de naam van het bestand is waarin het Pascal-programma is opgenomen, de zogenaamde *source file*. De naam van deze file moet altijd eindigen op `.p`. Beschouw nu het programma dat in Figuur 2 is weergegeven, en waarvan we aannemen dat het opgeslagen is in het bestand met naam `test.p`. Dit Pascal-programma is in staat een invoerbestand naar een uitvoerbestand te copiëren.

Het compileren van het programma in `test.p` gaat als volgt in zijn werk:

```
pc test.p
```

De uitvoer van de compiler is een bestand met naam:

```
a.out
```

Dit is een executeerbaar programma (*executable*), d.w.z. een programma op machine-niveau dat kan worden uitgevoerd door `a.out` in te tikken. Uiteraard zou het erg onhandig zijn als de naam `a.out` gehandhaafd zou blijven; men doet er over het algemeen verstandig aan de naam van de executable met behulp van `mv` een andere naam te geven:

```
mv a.out test
```

6.1.2 Separate compilatie

Een eigenschap van programmeertalen en compilers die gebruikt kunnen worden in projecten die de ontwikkeling van grote programma's tot doel hebben, is de *separate compilatie* van delen van het volledige programma. Hierdoor is het mogelijk de ontwikkeling van het volledige programma in hanteerbare 'modulen' door verschillende personen te laten plaatsvinden. Ook de Unix Pascal-compiler biedt deze mogelijkheid. We zullen ter illustratie van de wijze waarop dit mogelijk is het in Figuur 2 gegeven programma in 'modulen' splitsen die apart kunnen worden gecompileerd. (Overigens kent het oorspronkelijke Berkeley Pascal geen echt module-concept; de nieuwere versie van de taal die door SUNOS wordt ondersteund wel, maar uit overwegingen van overdraagbaarheid gaan we hieraan voorbij. We zullen elk van de programma-delen voor het gemak toch modulen noemen.)

Berkeley Pascal biedt de mogelijkheid delen van het volledige programma, met name procedures en functies, in een apart bestand bij elkaar te plaatsen en vervolgens apart te laten verwerken door de compiler. Ter illustratie bespreken we hoe het programma in Figuur 2 over vier bestanden verspreid kan worden, en hoe elk van deze programma-delen apart kan worden gecompileerd.

Modularisatie van een programma maakt het noodzakelijk de *interface* tussen de diverse modulen vast te leggen in zogenaamde *interface-specificaties*. Om te beginnen is het noodzakelijk de datatypen in het oorspronkelijke programma bereikbaar te maken voor elk van de modulen. De datatypen worden hiertoe in het bestand `types.h` opgenomen; in elk van de modulen kan nu met behulp van de specificatie `#include "types.h"` gebruik gemaakt worden van deze datatypen. In Figuur 3 is de inhoud van het bestand `types.h` weergegeven. De vier modulen zijn weergegeven in de Figuren 4, 5, 6 en 7. In de interface-specificatie van de procedure `copy` is het noodzakelijk de typen van de functie `getc` en de procedure `putc` op te geven. Dit gebeurt door het datatype van de functie, respectievelijk procedure op te geven, gevolgd door het sleutelwoord `external`. (Let op dit is geen standaad Pascal!) In Figuur 6 is dit gedaan. Merk op dat in elk van deze aparte bestanden de specifictie `#include "types.h"` staat vermeld. Hiermee wordt de file `types.h` in het bestand tijdens het

```
const EOF = -1;
      NL  = 10;

type character = -1..127;
```

Figuur 3: Datatypen in het bestand `types.h`.

```
#include "types.h"

{ getc -- get one character from standard input }
function getc(var c : character) : character;

var ch : char;

begin
  if eof then c := EOF
  else if eoln then
  begin
    readln;
    c := NL
  end
  else begin
    read(ch);
    c := ord(ch)
  end;
  getc := c
end;  { getc }
```

Figuur 4: Het bestand `getc.p`.

```
#include "types.h"

{ putc -- put one character to standard output }
procedure putc(c : character);

begin
  if c = NL then writeln
  else write(chr(c))
end;  { putc }
```

Figuur 5: Het bestand `putc.p`.

```
#include "types.h"

function getc(var c : character) : character; external;
procedure putc(c : character); external;

{ copy -- copy input to output }
procedure copy;

var c : character;

begin
  while getc(c) <> EOF do
    putc(c)
end;  { copy }
```

Figuur 6: Het bestand `copy.p`.

```
program Copyprog(input, output);

#include "types.h"

procedure copy; external;

begin { main }
  copy
end.
```

Figuur 7: Het bestand `main.p`.

compileren tussengevoegd. Elk van de bestanden kan men nu compileren door gebruikmaking van de Pascal-compiler; echter nu moet van van de `-c` optie gebruik gemaakt worden omdat het invoerbestand van de compiler anders als een volledig Pascal-programma beschouwd wordt. Module `getc.p` kan nu bijvoorbeeld als volgt worden gecompileerd:

```
pc -c getc.p
```

Op deze wijze kan ook elk van de andere modulen worden gecompileerd. Het is echter ook mogelijk de compiler slechts eenmaal aan te roepen voor alle modulen:

```
pc -c main.p getc.p putc.p copy.p
```

Hetzelfde effect kan bereikt worden met gebruikmaking van de `*`-notatie:

```
pc -c *.p
```

De uitvoer van de Pascal-compiler is een aantal zogenaamde *object files*. Object files onder Unix eindigen op de karakters `.o`. Deze object files moeten vervolgens worden gecombineerd tot één executable. Hiertoe maakt men gebruik van de zogenaamde *link editor* `ld`. In Pascal wordt echter gebruik gemaakt van een versie van `pc` die het programma `ld` aanroept.

```
pc -o copy main.o getc.o putc.o copy.o
```

De naam van de executable die door de link editor wordt geproduceerd is `copy`. Merk op dat als invoer voor `pc` nu object files (bijvoorbeeld `main.o`) fungeren, niet de files met de source-code. De naam van de resulterende executable is de naam volgend op de `-o` optie. Het resulterend programma `copy` kan nu gewoon door de gebruiker worden ingetikt:

```
copy < cursus1 > cursus2
```

waarin `cursus1` het invoerbestand is, en `cursus2` het uitvoerbestand (dat men m.b.v. `more` weer kan bekijken).

6.2 Make en de Makefile

Zoals we hierboven hebben gezegd is de mogelijkheid een programma uit modulen op te bouwen vooral handig bij de ontwikkeling van grote programma's. Een typisch verschijnsel tijdens de ontwikkeling van een groot programma is dat telkens slechts bepaalde modulen van het programma worden veranderd. Als men nu opnieuw een executable wil maken om te onderzoeken wat het effect is van deze verandering, dient men de aangepaste modulen opnieuw te compileren. Het is echter ook noodzakelijk de modulen die van de veranderde modulen afhangen (bijvoorbeeld omdat de procedures die in de veranderde modulen zijn gedefinieerd erin worden aangeroepen) opnieuw te compileren. Vervolgens kan dan met `pc -o` een nieuwe executable worden gemaakt.

Nu is het bij de ontwikkeling van een groot programma bijna onmogelijk bij te houden op welke wijze de modulen afhankelijk zijn van andere modulen. Daarom biedt Unix een hulpprogramma waarin de programmeur deze afhankelijkheden slechts eenmaal hoeft vast te leggen. Daarna zal dit programma op grond van deze informatie kunnen besluiten welke modulen opnieuw moeten worden gecompileerd.

Het bestand waarin deze afhankelijkheden worden vastgelegd is de zogenaamde *Makefile*. We zullen nu een Makefile ontwikkelen voor het eenvoudige Pascal-programma dat in de vorige paragraaf besproken is. Beschouw Figuur 8 waarin de Makefile opgenomen is. In de eerste regel van de `Makefile` is een opsomming gegeven van alle object files die gecreëerd moeten worden voor dit programma.

```
OBJ = main.o getc.o putc.o copy.o
```

Merk op dat er eventueel andere Pascal-modulen in de directory aanwezig mogen zijn. Deze worden genegeerd.

Op de volgende regel staat vermeld dat de executable `copy` heet, en gemaakt kan worden door aanroepen van `pc -o` met als argument de lijst van object files (`$(OBJ)`).

```
copy:   $(OBJ)
        pc -o copy $(OBJ)
```

Op de regels in `Makefile` die hierop volgen wordt telkens de naam van een object file gevolgd door de bestanden waarvan compilatie van de geassocieerde source file afhankelijk is. Beschouw bijvoorbeeld de volgende regel:

```
main.o: types.h getc.o putc.o copy.o
        pc -c main.p
```

```
OBJ = main.o getc.o putc.o copy.o

copy:   $(OBJ)
        pc -o copy $(OBJ)

main.o: types.h getc.o putc.o copy.o
        pc -c main.p

getc.o: types.h
        pc -c getc.p

putc.o: types.h
        pc -c putc.p

copy.o: types.h getc.o putc.o
        pc -c copy.p

print:
        lpr -Palw types.h main.p getc.p putc.p copy.p

clean:
        rm -f $(OBJ)
```

Figuur 8: `Makefile` voor Pascal-programma.

Hierin staat vermeld dat de module `main.p` opnieuw moet worden gecompileerd (d.w.z. `pc -c main.p` moet worden uitgevoerd) zodra er een verandering is opgetreden in één van de bestanden `types.h`, `getc.o`, `putc.o` of `copy.o`. De resterende regels komen straks aan de orde.

Stel dat we nu gebruik willen maken van de `Makefile`. Dit kan door middel van het programma `make`, dat automatisch de informatie in het bestand `Makefile` verwerkt. Intikken van

```
make copy
```

leidt tot de uitvoering van de volgende commando's (die de gebruiker dus niet heeft ingetikt):

```
pc -c getc.p
pc -c putc.p
pc -c copy.p
pc -c main.p
pc -o copy main.o getc.o putc.o copy.o
```

Dit resultaat kan als volgt verklaard worden. Eerst probeert `make` de executable `copy` te maken met behulp van `pc -o`. Dit lukt echter niet, want initieel is er geen enkele object file aanwezig. Vervolgens probeert `make` de module `main.p` te compileren. Echter, volgens de specificatie van de afhankelijkheden dienen hiertoe eerst `getc.o`, `putc.o` en `copy.o` aanwezig te zijn. De file `types.h` is al aanwezig. Vervolgens gaat `make` naar de regel waar staat

'`getc.o:`'; omdat `getc.o` slechts afhankelijk is van het reeds bestaande bestand `types.h` wordt de actie die gespecificeerd is uitgevoerd:

```
pc -c getc.p
```

Op analoge wijze worden de afhankelijkheden `putc.o` en `copy.o` verwerkt. Pas als alle object files bestaan wordt `main.p` gecompileerd.

Het zal nu inmiddels wel duidelijk zijn dat de algemene vorm van regels in de `Makefile` is:

<*target*> : <*afhankelijkheden*>
<*acties*>

waarin <*target*> de naam van een object file, executable of een willekeurige andere naam is, afhankelijk van de toepassing. De specificatie van <*acties*> mag slechts vooraf gegaan worden door een TAB-karakter. Een voorbeeld van een actie is bijvoorbeeld `pc -c main.p` in Figuur 8.

Het vastleggen van deze afhankelijkheden kan het onnodig opnieuw compileren voorkomen. Veronderstel dat de source file `getc.p` iets wordt aangepast. Als we daarna opnieuw `make copy` intikken, dan zullen de volgende commando's worden uitgevoerd:

```
pc -c getc.p
pc -c copy.p
pc -c main.p
pc -o copy main.o getc.o putc.o copy.o
```

Merk op dat het bestand `putc.p` nu niet wordt gecompileerd. Kunt u dit op grond van de inhoud van de `Makefile` ook verklaren?

Tenslotte hoeft een `Makefile` niet slechts gebruikt te worden voor het opgeven van de wijze waarop modulen samenhangen. Men kan bijvoorbeeld ook informatie met betrekking tot het afdrukken van bestanden opgeven. In Figuur 8 is dit gedaan door een 'object file' `print` op te geven. Het intikken van

```
make print
```

heeft tot gevolg dat het volgende commando wordt uitgevoerd:

```
lpr -Palw types.h main.p getc.p putc.p copy.p
```

Analoog is het bijvoorbeeld mogelijk alle (relevante) object files te verwijderen (nadat de executable `copy` gemaakt is) door

```
make clean
```

in te tikken. Het volgende commando zal hierdoor worden uitgevoerd:

```
rm -f main.o getc.o putc.o copy.o
```

waardoor alle object files die gebruik zijn voor het samenstellen van de executable `copy` worden verwijderd.

Oefenopgaven

De volgende opgaven zijn bedoeld om u enige ervaring met de Pascal-compiler en het programma `make` te laten opdoen.

- *Copieer het bestand* `test.p` *dat in de directory* `/u/mik/unix` *is opgenomen naar uw home-directory. Probeer dit programma te compileren en experimenteer met de resulterende executable.*

- *Splits vervolgens het bestand* `test.p` *op in een aantal modulen, op analoge wijze als besproken in Paragraaf 6.1.2. Compileer dit programma en maak opnieuw een executable. Verwijder nu met behulp van* `mv` *het bestand* `main.o`*, en probeer opnieuw een executable te maken. Kunt u verklaren wat er fout gaat bij het aanroepen van deze executable?*

- *Probeer nu met behulp van 'vi' een* `Makefile` *te creëren, analoog aan de* `Makefile` *in Figuur 8. Probeer de voorbeelden uit die in Paragraaf 6.2 zijn besproken.*