

LINUX SYSADMIN BASICS

INHOUD

EEN BEETJE GESCHIEDENIS KAN NOOIT KWAAD	3
UNIX HISTORIE IN VOGELVLUCHT	6
OPBOUW VAN HET SYSTEEM	8
ZO, NU AAN HET WERK	10
DE SHELL / BASISCOMMANDO'S	10
JE WEG VINDEN	14
HET FILESYSTEM	15
DIRECTORIES	16
WERKEN MET BESTANDEN	24
BESTANDEN ZOEKEN	29
FILTERS	34
GEAVANCEERDERE MOGELIJKHEDEN	40
PIPING EN REDIRECTION	40
XARGS EN EXEC	42
BESTANDEN EN DIRECTORIES COMPRIMEREN	44
SHELL VARIABELEN	45
CRONTAB & JOBS	46
PROCESSEN	47
PERFORMANCE	48
REMOTE CONNECTIONS & THIRD PARTY SOFTWARE	50
SSH / SCP	50
PASSWORDLESS SSH / REMOTE COMMANDO'S	52
FTP	54
MYSQL & GIT	56
MYSQL COMMANDLINE	58
DATABASE DUMP & GIT	60
DE SHELL SLIM(MER) GEBRUIKEN	61
EEN UITGEBREID SCRIPT	62

EEN BEETJE GESCHIEDENIS KAN NOOIT KWAAAD...

Unix varianten zijn er al sinds het begin der (automatiserings)tijden. Begonnen als een experiment is het al snel uitgegroeid tot een volwaardig OS dat met name op universiteiten en natuurlijk in defensie toepassingen werd gebruikt. De voorloper van het internet (ARPANET, de "Advanced Research Projects Agency" van het ministerie van Defensie van de VS), werd grotendeels gebouwd op de eerste versies van Unix.

De eerste implementatie waren zeer low-level, dwz dat de operator direct toegang had tot de onderliggende systeemfuncties.

Uiteraard zorgen de fabrikanten van "gebruikers software", denk aan bijvoorbeeld Microsoft en Apple, ervoor dat kritieke systeemfuncties niet zo 1-2-3 vanuit de zogenaamde GUI (Graphical User Interface) te benaderen en te modifieren zijn. Je zult hiervoor echt op de laag onder de GUI moeten zijn, bijna op het niveau van het operating system zelf. En dan bij voorkeur ook nog op dat niveau waar eventuele restricties geen rol meer spelen.

Nou is dat bij Windows-systemen meestal bijna niet te doen. Windows is van oudsher (en dan denk ik aan MS-DOS) een operating system dat alle kritieke systeemfuncties ontoegankelijk gemaakt heeft. Dat past ook wel in de filosofie van hoe Microsoft de computermarkt benadert: power to the people, en die systemen, laat dat maar aan ons over. Pas toen Microsoft met Windows NT ook de servermarkt op ging – ergens halverwege de jaren '90 van de vorige eeuw – ontstond langzaam de behoefte en de noodzaak om veel onderliggende

systeemfuncties te ontsluiten: aan servers worden toch echt wel andere eisen gesteld dan aan het gemiddelde huis-tuin-en-keuken-pcje.

De grote concurrent Apple, had dat al iets eerder in de smiezen. Met de gigantische stap die Apple begin deze eeuw gezet heeft van OS-9 naar het volledig nieuwe, en vanaf 0 opgebouwde, OSX, hebben ze bepaalde low level functies – weliswaar verborgen, maar ze zijn er – ontsloten voor de eindgebruiker. OSX is gebouwd op een apart gelicenseerde Unix-versie van Free-BSD, dat, in tegenstelling tot wat de naam doet vermoeden, geen Open Source versie is. Door deze stap sluit OSX eigenlijk naadloos aan bij het "Operating System of Choice" aan de serverkant van het internet: Unix (daar waar ik de term Unix gebruik, kun je net zo goed Linux of een van z'n varianten lezen).

Apple heeft hiermee eigenlijk de kracht van een enorm krachtige en gebruikersvriendelijke GUI gekoppeld aan het enorm krachtige Unix operating system.

HET AFZIEN VAN DE JAREN '90

In tegenstelling tot de Microsoft-approach, is de gebruiker wat Unix betreft een noodzakelijk kwaad. Als een Unix computer bezig is met opstarten, schreeuwt het hele operating system eigenlijk al: "Als je niet weet wat je aan het doen bent, dan lazer je maar op" - er flitsen allerlei onbegrijpelijke boodschappen over het scherm en uiteindelijk staat er of de tekst "Kernel Panic" – dan weet je dat je als systeembeheerder hebt zitten prutsen en wordt het overwerken – of het verlossende woord "username:" met een knipperende cursor erachter.

De meeste Linux-varianten booten tegenwoordig

over het algemeen direct door naar een grafische desktopomgeving (de zgn. Window-manager). Maar op servers is deze feature omwille van memory over het algemeen niet geïnstalleerd. Alle systeemhandelingen (netwerk instellingen, het gebruik van USB of andere disks of apparaten) kun je ook via de command line uitvoeren. Sterker nog: de meeste configuratie software is niet meer dan een grafisch omhulsel van een command line tool. Het gebruik van eindgebruikers-toepassingen zoals tekstverwerkers of tekenprogramma's is natuurlijk een ander verhaal, maar verder kun je de hele computer via de command line besturen.

Unix is gebouwd voor ontwikkelaars en systeem-jongens en meisjes, pas met het verschijnen van Linux (en dan met name de Ubuntu varianten) ging de community zich eigenlijk pas richten op het "normale" gebruik van zo'n systeem.

Maar het onderliggende systeem is nog altijd heel sterk op ontwikkelaars en infrastructuur mensen gericht. Je ziet Unix-omgevingen vanaf eind jaren '70 tot aan deze eeuw eigenlijk alleen maar in specialistische omgevingen gebruikt worden. Met name voor grote database toepassingen als Oracle of Progress in administratieve omgevingen en in productie of industriële omgevingen waarbij enorme hoeveelheden data verwerkt moeten worden. Dat was ook een markt waar met name HP (met oa. de HP9000 servers die op HP-UX draaiden) en IBM (met AIX machines) peperdure computers verkochten, uitgerust met hun eigen CPU's (de PA-RISC van HP bijvoorbeeld) en hun eigen smaak Unix (HP-UX, AIX en Ultrix etc).

Deze machines – vaak ook fysiek imponerend, ik heb servers beheerd van 2m hoog waarin een betonplaat was gemonteerd tegen het trillen – waren bedoeld voor het betere stampwerk. Al in de jaren '90 volledig voorzien van SAN (storage area

network) voorzieningen via glasvezel, full SCSI en RAID5 en nog meer van dat soort termen waarmee je op (nerd)feestjes de blits kon maken. Tegenwoordig geen cloud zonder shared storage, maar indertijd gewoon niet aanwezig in de Windows of Novell wereld.

Maar... het was natuurlijk helemaal geweldig dat dat allemaal zomaar kon. Maar je werd stevast begroet door die "username:" prompt. En nu dan?

Inloggen met de username/password combinatie die op verpakking staat, en vervolgens staat er:

\$>

op een verder volkomen leeg scherm. Goed, een ton aan ijzer onder je vingers en dat kreng vertelt je niet meer dan \$>...

Het was afzien in de jaren '90...

MINI COMPUTERS

Hiernaast een HP9000 K-klasse server met twee diskcabinets. Een typisch voorbeeld van de zgn. "mini computers" die in de jaren '90 van de vorige eeuw het echte "stampwerk" deden.

Wie de term "mini" verzonden heeft, heeft deze machines duidelijk nog nooit gezien.

REVOLUTIE

Toen een Finse student in 1991 besloot om zijn eigen operating system te gaan bouwen, kwam de zaak in een stroomversnelling. Het was namelijk zijn idee om de kracht van Unix op een huis-tuin-en-keuken-pc te kunnen gebruiken. Al gauw had deze student (Linus Torvalds) een hoop programmeurs om zich heen verzameld, en Linux was geboren.

Omdat Unix nu ook op "gewone" Intel x86 CPU's kon draaien, en met een gratis, en open sourced besturingssysteem, was het niet meer noodzakelijk voor ondernemers om in dure hardware te investeren om toch een graantje mee te kunnen pikken van dat rare, hippe, nieuwe (hoewel de oorsprong in de jaren '60 ligt) iets dat ze "internet" noemden. Het duurde dan ook niet lang voordat allerlei Linux varianten (distributies genaamd) op de markt verschenen. Niet alleen werd de populaire webserver software "Apache" gebundeld in deze distributies, maar ook mail-servers, FTP-servers, database toepassingen (bij MySQL). In de moderne versies van bijna alle distributies kun je met een druk op de knop een volledig functionele webserver installeren.

Hierdoor kreeg de hosting-markt natuurlijk een ontzettende boost. En was de server-kant van het internet niet alleen toegankelijk voor grote bedrijven, maar ineens konden ook kleine hosting providers een rol op die markt gaan spelen.

De meeste hosting providers bieden hosting aan die gebaseerd is op een Linux variant, van de instap-pakketten tot aan de gespecialiseerde VPS (Virtual Private Server) managed en unmanaged. Dit heeft eind jaren '90/begin deze eeuw mede bijgedragen aan de enorme bloei die het internet doorgemaakt heeft.

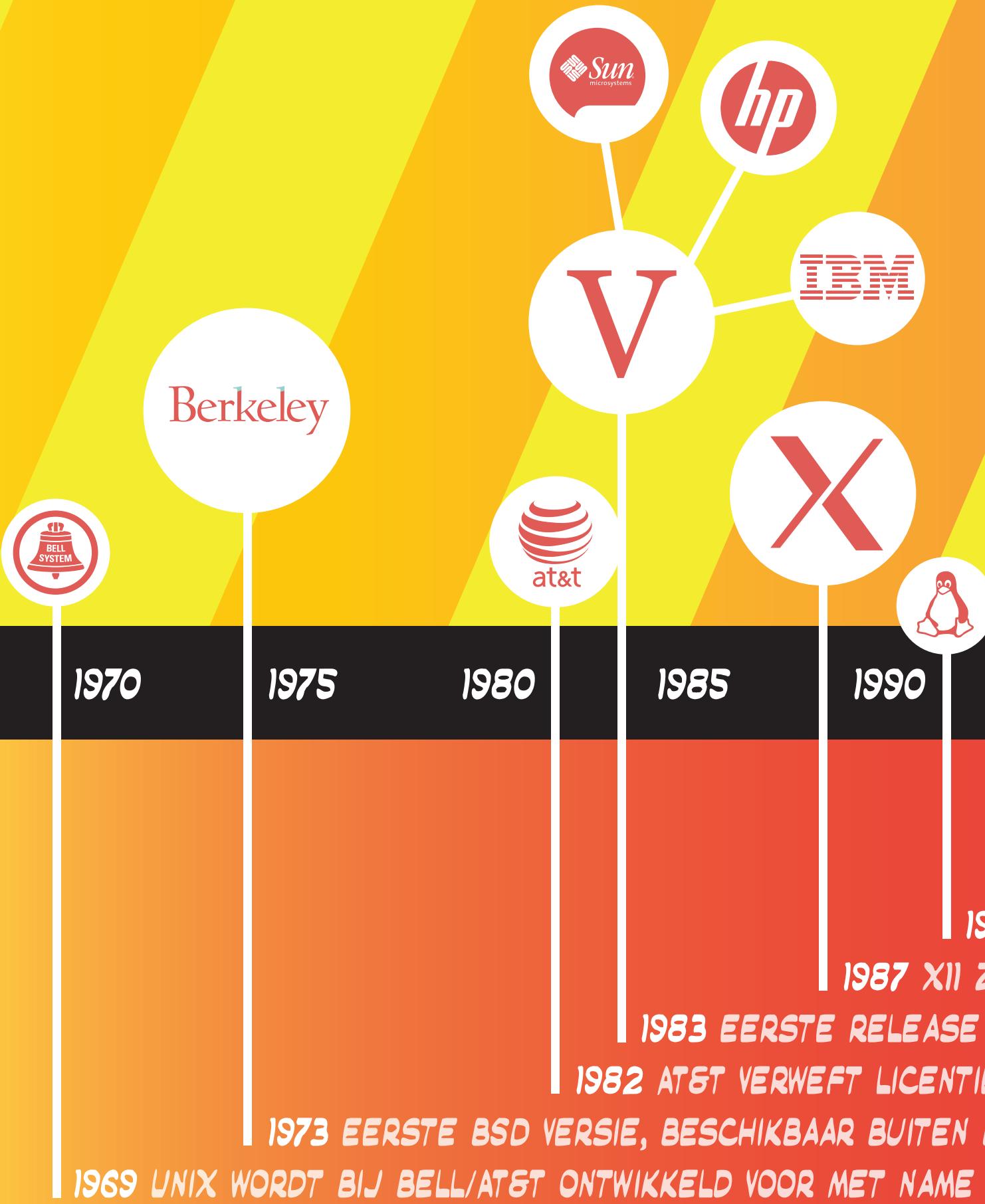
Inmiddels is Linux een volwassen operating system geworden, alle grote leveranciers van server software (zoals Oracle, SAP, Progress) ondersteunen - of hebben zelfs hun eigen distributie - Linux. HP en IBM, van oudsher de grootste spelers op de Unix markt, leveren tegenwoordig standaard een Linux versie uit met hun servers en je ziet dat HP-UX en AIX langzaam maar zeker uitgefaseerd worden. En ook in de "embedded" en PLC-markt is het Linux dat de klok slaat.

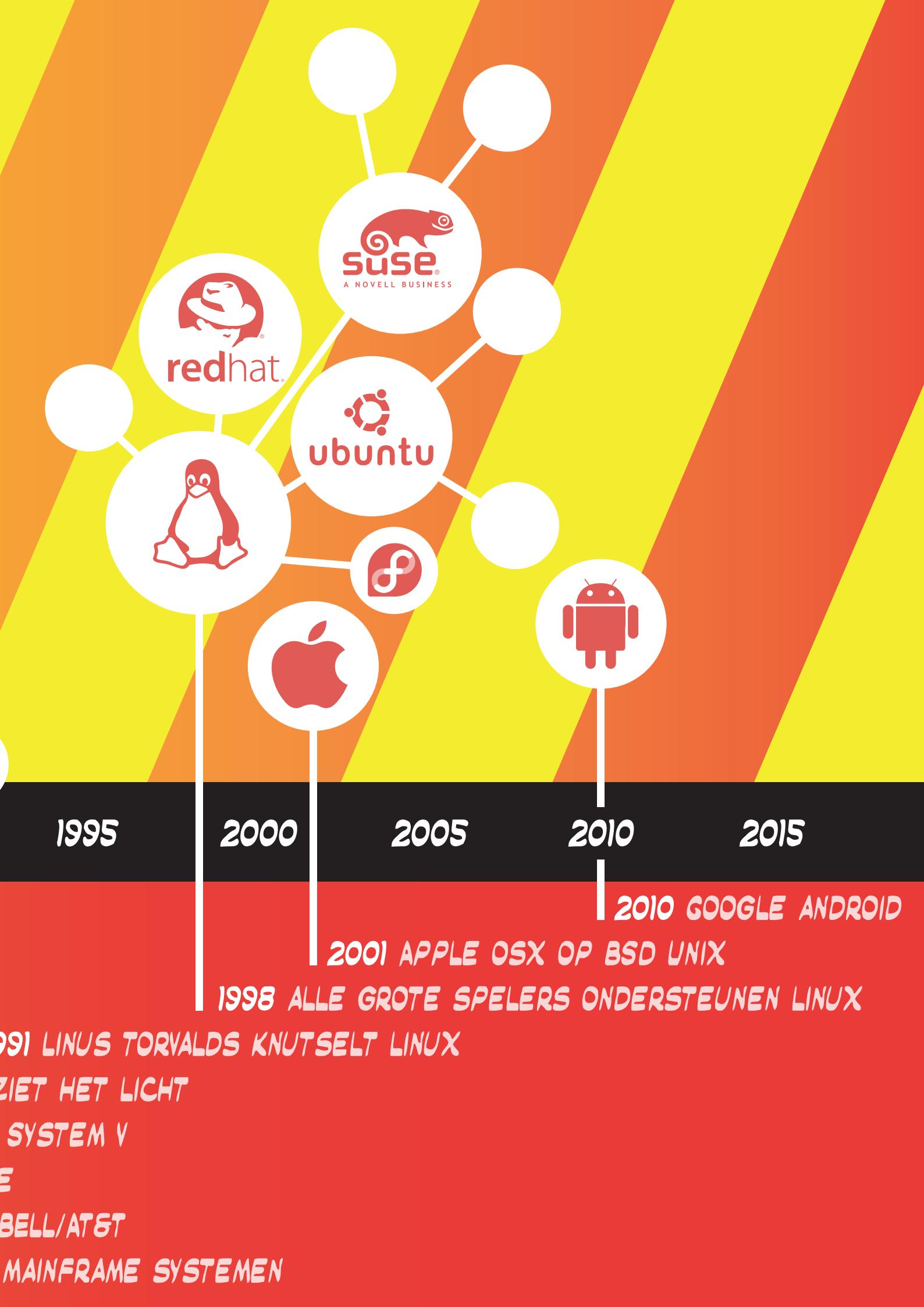
MARKTAANDEEL

De Gartner-group schat dat het aandeel van Linux servers op het internet inmiddels ongeveer 32% is. Microsoft heeft ongeveer 35% en de andere 33% is "onbekend". Geheid dat een groot deel daarvan ook een of andere Linux variant is.



UNIX HISTORIE IN VOGELVLUCHT





OPBOUW VAN HET SYSTEEM

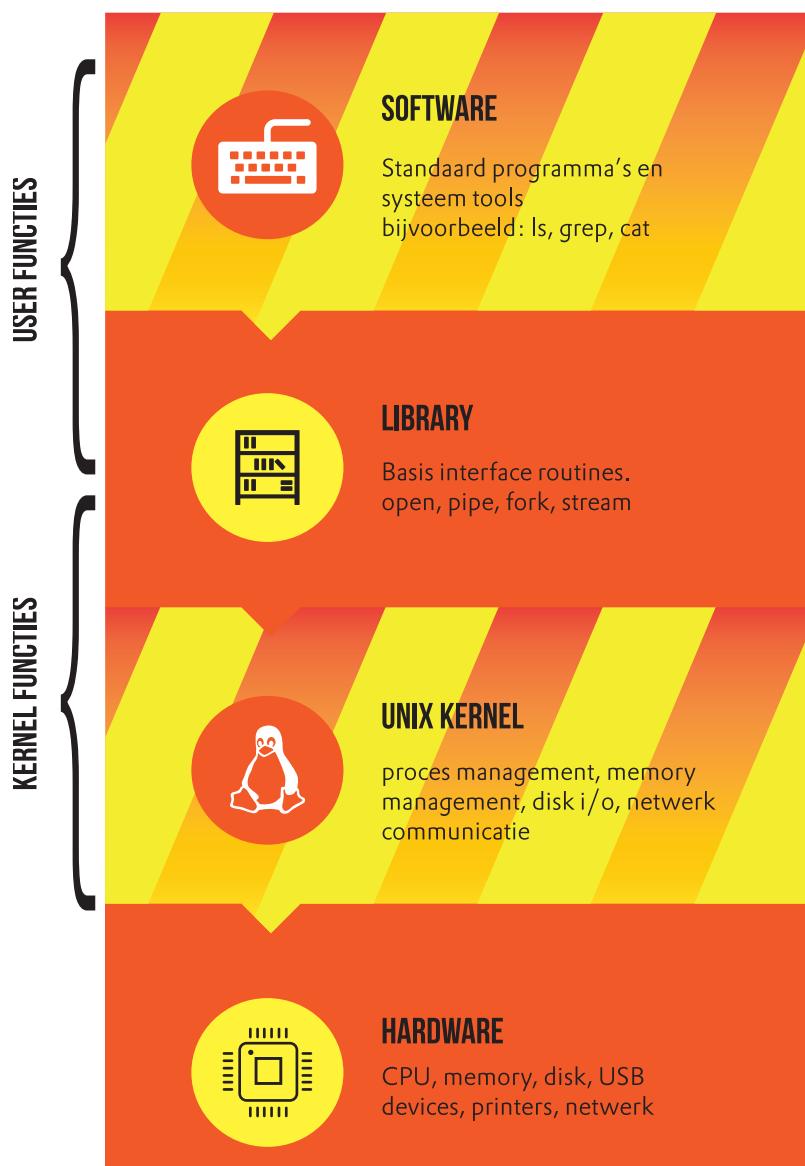
Om een beetje de structuur van het systeem te begrijpen, is het handig om even een overzichtje te schetsen hoe zo'n systeem nou daadwerkelijk opgebouwd is. In tegenstelling tot Windows-machines communiceer je in Unix altijd met behulp van bestanden.

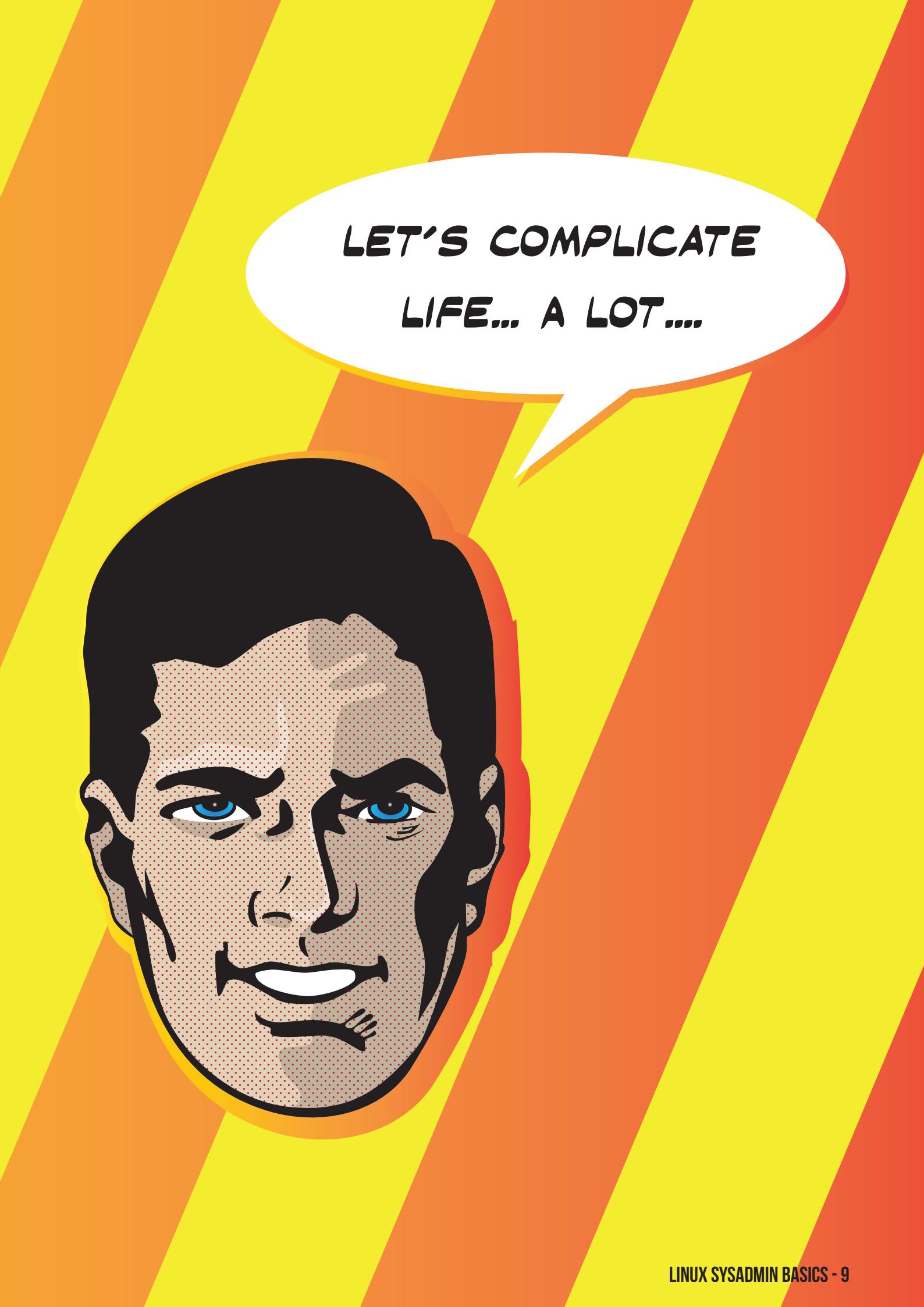
Devices (hardware) zijn op systeem niveau niet meer dan bestanden met een speciale functie. Zo is ook je beeldscherm "gewoon" een bestand en kun je met een commando uitvoer (output) naar dat bestand schrijven. Standaard wordt alle output naar het bestand dat in de software STDOUT (over het algemeen het beeldscherm van de ingelogde gebruiker) genoemd is geschreven, maar je kunt output ook "redirecten" naar een ander bestand / uitvoer medium. Dat is handig omdat je vaak de uitvoer van een commando als invoer voor een ander commando wilt gaan gebruiken. Ook kun je conditioneel werken, zodra de uitvoer van een bepaald programma "X" is, dan voer "Y" uit en anders "Z".

Dat lijkt een beetje op programmeren, en dat is het ook. Je kunt de computer bepaalde taken laten uitvoeren (dat noemen ze over het algemeen "batches") waarin je een bepaalde foutmarge opneemt. Bijvoorbeeld als de backup niet gelukt is, dan stuur een mailtje naar de systeembeheerder. We zullen straks een aantal voorbeelden behandelen en dan kun je je eigen batch schrijven.

SECURITY

De beveiliging is op een Unix systeem "ingebakken" in de kernel. Alle bestanden zijn voorzien van een zgn. ACL (Access Control List) en deze wordt op filesystem niveau door de kernel beheerd. Hierdoor is het zo goed als onmogelijk voor virusen een Unix systeem te besmetten: heb je niet de juiste rechten, dan mag je het bestand niet wijzigen! In werkelijkheid functioneert het iets anders, maar in grote lijnen komt het wel hierop neer. Het voert wat te ver om precies te gaan uitleggen hoe de zogenaamde inodes werken en hoe de kerneltransacties verlopen.





**LET'S COMPLICATE
LIFE... A LOT....**

ZO, NU AAN HET WERK!

DE SHELL

Zodra je bent ingelogd, start een specifiek programma – een systeembeheerder kan dat voor je instellen – op waarmee je toegang krijgt tot het systeem. Een bepaald set programma's stellen je in staat om commando's op het systeem uit te voeren. Dit noemen ze de "shell"-programma's. Er bestaan diverse varianten die elk met hun eigen specifieke syntax (een soort programmeertaal) werken.

De bash wordt tegenwoordig het meeste gebruikt, daarnaast heb je nog de ksh, de sh en de csh. De sh (van shell, of Bourne-shell – ontwikkeld door Stephen Bourne) is een uitgeklede variant die erg beperkt is. De csh heeft een syntax die nogal op de programmeertaal C lijkt, nogal omslachtig in het gebruik dus. De ksh (Korn-shell) was zeer populair vanwege de uitgebreide mogelijkheden en deze is later omgeschreven naar de bash (en dat staat voor

"Bourne Again Shell" – een uitbreiding op de Bourne-shell en het klinkt als "born again", welkom in Nerdistan...) met nog meer mogelijkheden. De shell stelt je dus in staat om commando's (programma's) uit te voeren. Een soort van Windows-start knop, maar dan anders.

Over het algemeen begroet zo'n shell je met een zgn. "prompt" waarin meestal je gebruikersnaam staat en de directory waarin je je bevindt. De systeembeheerder kan die prompt helemaal voor je configureren. De knipperende cursor geeft aan dat je iets kunt invoeren.

BASIS COMMANDO'S

Het eerste commando dat je over het algemeen leert is het commando 'ls' (list) en dit geeft je een overzicht van de bestanden in de actieve directory. Als je dit nu uitvoert zullen er verdacht weinig bestanden in je (home) directory aanwezig zijn, maar daar gaan we snel verandering in brengen.





Opdracht

```
#> ls <enter>
```

```
#> ls -a <enter>
```

Type nu maar eens ls -a, je zult zien dat je nu ineens meer ziet dan zonet. De bestanden die met een . beginnen (bijvoorbeeld .bash-profile) zijn "hidden files", verborgen bestanden. Dat zijn vooral configuratie files en bestanden waarvan je niet wil dat ze gemakkelijk bewerkt kunnen worden.

De "-a" in deze opdracht noemen ze een "flag". Veel commando's hebben een serie aan flags die gebruikt kunnen worden om het commando een specifieke opdracht te geven. De "-a" betekent hier dat ook verborgen bestanden getoond moeten worden. Veel flags kunnen gecombineerd worden, andere weer niet.

Aantekeningen

Download de voorbeeldbestanden!

```
*> cd $HOME <enter>
*> mkdir sampledata <enter>
*> cd sampledata <enter>
*> wget http://workshops.educom.nu/linux-cli.zip <enter>
*> ls <enter>
*> unzip linux-cli.zip <enter>
*> ls <enter>
```



Opdracht

```
#> ls -ltra /usr/bin <enter>
```

De inhoud van een specifieke directory kun je opvragen door de naam van de directory op te geven achter de flag(s). Dit noemt men een argument of parameter. Als het goed is, flitst nu de inhoud van `/usr/bin` directory over het scherm.

De `-ltra` flag vertelt het `ls` commando dat het een volledige lijst moet tonen, aflopend gesorteerd op bewerksdatum en tevens alle verborgen bestanden en directory's moet tonen.

Aantekeningen

0

De standaard notatie van een Unix commando/programma is als volgt - en dit geldt voor zo'n beetje alle standaard commando's:

● ● ● Opdracht

```
#> comando -[flag,flag,flag] [parameters] <enter>
```

"Da's leuk" zul je zeggen, "Moet ik die flags allemaal onthouden?". Nou... ja dus... Maar niet getreurd, zoals het een goed systeem betaamt, is er een "help functie" aanwezig. En zoals bij zoveel systemen, is ook hier de help functie geschreven door techneuten. Oftewel, veel plezier bij het uitvogelen ervan! Je kunt de help functie aanspreken door het "**man**" (van manual) commando uit te voeren (op voorwaarde dat de plaatselijke systeembeheerder de 'man pages' geïnstalleerd heeft).

Je ziet nu de manual page van ls op het scherm. Met de pijltjes-toetsen kun je vooruit en achteruit scrollen, met <q> beeindig je het programma (een en ander kan afhankelijk zijn van de Linux versie waar je mee werkt, **man man** zou kunnen helpen).

● ● ● Opdracht

```
#> man ls <enter>
```

Opdracht

```
#> pwd <enter>  
  
# het systeem laat je iets zien in de geest van:  
/home/[username]
```

BASIS COMMANDO'S: JE WEG VINDEN

Aangezien Unix volledig file georiënteerd is, is het natuurlijk van belang dat je de juiste files weet te vinden. Er zijn een aantal commando's van belang om door je bestanden en directory's te navigeren. We hebben net al stil gestaan bij het 'ls' commando. Om te achterhalen in welke directory je momenteel bent (en soms kun je dat niet aan de prompt zien), kun je het commando '**pwd**' (print working directory) gebruiken.

[**username**] staat voor de gebruikersnaam waaronder je bent ingelogd. Je ziet dat Unix gebruik maakt van forward-slashes om directory's en subdirectory's aan te geven (wellicht herken je dit van de URL's op een website, dat heeft nl. dezelfde oorsprong).



Opdracht

```
#> pwd <enter>  
  
/home/[username]  
  
#> cd /tmp <enter>  
  
#> pwd <enter>  
  
/tmp  
  
#> ls -ltra <enter>
```

Om van directory te wisselen, kun je het commando ‘**cd**’ (change directory) gebruiken. Als je ‘**cd**’ zonder extra argument gebruikt, dan spring je vanzelf terug naar je home directory. Gebruik je ‘**cd**’ met een slash ervoor, dan staat dit voor de directory direct onder de root-directory (de basis van het systeem).

Voer de bovenstaande reeks commando’s uit.

Je krijgt nu een overzicht van alle bestanden (mochten die er zijn) in de /tmp directory. Deze directory wordt overigens door het systeem gebruikt om allerlei tijdelijke bestanden weg te schrijven.

Aantekeningen

0

HET FILESYSTEM

Alle Unix systemen zijn volgens dezelfde structuur opgebouwd. Dit is vastgelegd in een standaard (POSIX) en als je als OS-ontwikkelaar daaraan voldoet dan mag je je OS “*nix” noemen. Dat begint met de directory (mappen) structuur.

Net als alle andere (moderne) systemen werkt een Unix systeem met directory's en subdirectory's waarin de diverse bestanden zijn ondergebracht. Er zijn vanaf de root-directory tal van subdirectory's waarin diverse software is ondergebracht.

Binnen deze context zijn ze niet allemaal even relevant, onderstaand overzicht toont de belangrijkste met een korte omschrijving van het soort bestanden dat je hierin kunt vinden.

Het Filesystem

```
/---  
  - bin  
  - dev  
  - etc  
  - home  
    -- [user1]  
    -- [user2]  
    -- [etc...]  
  - lib  
  - opt  
  - usr  
    -- bin  
    -- local  
      -- bin  
    [etc...]  
  - tmp  
  - var  
    -- log  
    -- www  
    -- [etc...]
```

Aantekeningen

3

DIRECTORIES

/bin

In de /bin directory staan zgn "systeem commando's", zoals bijvoorbeeld het 'ls' commando, maar ook de diverse shells,

/dev

In /dev wonen de devices. Als gezegd, is Unix volledig file georiënteerd, elk stuk hardware is als file in deze directory vertegenwoordigd. Zo is er bijvoorbeeld een bestand /dev/console dat de communicatie met het aangesloten beeldscherm verzorgt. Maar je vindt hier ook de ingangen voor harddisks en zelfs je terminal waar je nu in werkt staat erbij.

/etc

De /etc directory is bedoeld voor allerlei configuratie bestanden. Er zijn tal van subdirectory's voor bijvoorbeeld apache, mail en minder tot de verbeelding sprekende software (ntp, ldap, ftp).

Veruit het bekendste bestand zal /etc/hosts zijn (ook op windows voorhanden), dit is een wezenlijk onderdeel van het TCP/IP protocol en bevat bekende machine's in het netwerk met hun ip-adres.

/home

De /home directory bevat alle, de naam zegt het al, home directory's van de op het systeem geconfigureerde gebruikers - die gebruikers zitten overigens in het bestand /etc/passwd.

/lib

Deze directory bevat de zgn. shared libraries. Systeemfuncties die door elk programma

gebruikt worden (bijv. voor het openen van bestanden en het schrijven naar scherm of disk). Enigszins vergelijkbaar met de DLL bestanden van Windows.

/opt, /var en /tmp

Deze directory's zijn bedoeld voor software en gegevens die in principe niks met het systeem van doen hebben. Dus hier installeer je bijvoorbeeld Oracle oid. De /var is de vergaarbak van oa. log-bestanden maar ook Apache zet hier z'n www-directory bijvoorbeeld.

/usr

/usr tenslotte bevat "bijzaak" commando's. Zoals de cmdline tools die we later gaan gebruiken. Vanuit de history is er een /usr/bin directory (voor tools) en een /usr/local/bin directory (voor je eigen programma's en tools). Maar dat wordt nogal door elkaar gebruikt



DIRECTORIES & PERMISSIONS

Veel van de tools/programma's die we gaan gebruiken staan in de `/bin`, `/usr/bin` en `/usr/local/bin` directory. Deze directory's zijn in de shell standaard opgenomen in de **PATH**-variabele. Deze zogenaamde "omgevingsvariabele" vertelt aan het shell programma in welke directory's het moet zoeken om bepaalde programma's te vinden. Als de directory van een programma niet in de **PATH**-variabele staat, zul je het hele pad naar het programma moeten intypen.

Bijvoorbeeld: als `/bin` niet in het **PATH** zou staan, dan moet je de volledige directory en de naam van het programma - gescheiden door '/' intoetsen om het programma uit te voeren (dit noemen ze een pad).



Opdracht

```
#> /bin/ls <enter>
```

Als we straks zelf een script gaan schrijven en dat willen gaan uitvoeren, moet of de directory in het **PATH** worden opgenomen, of we moeten het volledige pad naar het script intoetsen.

Ik neem aan dat jullie nu al terug verlangen naar de Verkenner of de Finder en smachten naar een "dubbelklik"?

Hoe weet je nou of een directory wel of niet in het pad staat? Nou, dat kan door de naam van het programma in te toetsen op de commandline:

● ● ● **Opdracht**

```
#> bestaat-niet <enter>
-bash: bestaat-niet: command not found
```

De shell komt nu met de melding ‘command not found’. We kunnen beter even checken wat de inhoud van de omgevingsvariabele **PATH** is. Dat doen we als volgt:

● ● ● **Opdracht**

```
#> echo $PATH <enter>
/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
```

Hier zie je welke directory's er in het **PATH** zijn opgenomen. De shell doorzoekt de directory's in het **PATH** in de volgorde waarin ze zijn opgenomen, en zodra het programma gevonden is wordt dit uitgevoerd.



Gezien de bijna oneindige hoeveelheid bestanden en directory's verdient het de aanbeveling om je eigen werk goed te structureren. Dit kun je doen door je project(en) in te delen in een duidelijke directory structuur. Het is in de Unix- en Webdevelopment-wereld een soort van onofficiële standaard (best practice) om in grote lijnen de structuur van de Unix directory's te volgen in je projecten.

Een directory kun je maken met het **mkdir** (make directory) commando. Een directory verwijder je met het **rmdir** (remove directory) commando.

LET OP!

Dit kan alleen als de directory leeg is. Dus eerst met **ls -a** checken of er niks stiekems in staat. Zogenaamde hidden-files en directories beginnen namelijk met een '.' (punt) waardoor ze 'verborgen' zijn voor normaal gebruik.

We zien in deze opdracht ook een bewijs van de "*Als je niet weet wat je aan het doen bent, dan lazer je maar op*"-filosofie van Unix. rmdir vraagt dus niet of je het wel zeker weet of je die directory wilt verwijderen, nee, het commando wordt gewoon uitgevoerd.

In de vorige opdracht had je ook het commando '**mv**' (move) kunnen gebruiken om de directory te hernoemen:

mv [source] [destination] werkt op zowel directory's als gewone bestanden.

Opdracht

Voer onderstaande commando's uit:

```
#> cd <enter>
#> mkdir cfg <enter>
#> mkdir bin <enter>
#> mkdir lib <enter>
#> mkdir Temp <enter>
#> ls <enter>
Temp bin cfg lib
```

Hmm, Temp had tmp moeten zijn...

```
#> rmdir Temp <enter>
#> mkdir tmp <enter>
#> ls <enter>
bin cfg lib tmp
```

Opdracht

Voer onderstaande commando's uit:

```
#> mv tmp Temp <enter>
#> ls <enter>
Temp bin cfg lib
```

```
#> mv Temp tmp <enter>
#> ls <enter>
bin cfg lib tmp
```

FILESYSTEM SECURITY

Als gewone gebruiker mag je op een Unix systeem zo goed als niks. Een beetje aan je eigen files werken, misschien wat configuratie files bekijken en bepaalde programma's opstarten, maar dat was het dan ook. Iedere file en directory heeft z'n eigen permissies die individueel ingesteld kunnen worden. Die permissies kun je met het **ls -l** commando bekijken.

Opdracht

```
#> ls -l <enter>
total 0
drwxr-xr-x  2 rene  staff  68 30 aug 14:16 bin
drwxr-xr-x  2 rene  staff  68 30 aug 14:16 cfg
drwxr-xr-x  2 rene  staff  68 30 aug 14:16 lib
drwxr-xr-x  2 rene  staff  68 30 aug 14:16 tmp
```

permissions owner group size date/timestamp file/dir naam

In de eerste kolom betekent '**d**' dat het een directory is. Bij een gewone file staat er '-'. Vervolgens een matrix met de rechten in 3 groepen van 3 permissies, namelijk: **eigenaar - groep - rest van de wereld**.

r = Read
w = Write
x = Execute
- = Geen rechten

Kolom 2: is een cryptische indicatie van de directory zelf.

Kolom 3: geeft de eigenaar van het bestand/directory aan - '**rene**' in dit geval.

De vierde kolom geeft aan welke groep van gebruikers rechten hebben op deze file (meestal is het de groep waar de eigenaar toe behoort, hier: '**staff**'

Kolom 5: toont de bestandsomvang.

Kolom 6: de datum en tijd van de laatste modificatie
De laatste kolom tenslotte toont de naam van de file/directory.



Voer onderstaande commando's uit:

```
#> chmod go-rx cfg <enter>  
  
#> ls -l  
total 0  
drwxr-xr-x  2 rene  staff  68 30 aug 14:16 bin  
drwxr-xr-x  2 rene  staff  68 30 aug 14:16 lib  
drwxr-xr-x  2 rene  staff  68 30 aug 14:16 tmp  
drwx-----  2 rene  staff  68 30 aug 14:32 cfg  
  
#> chmod g+r cfg <enter>
```

De permissies kunnen gewijzigd worden door het commando **chmod** (change mode). Bijvoorbeeld: als we niet willen dat onze collega's in onze directory's kunnen snuffelen, kunnen we deze met dit commando "dichtzetten".

Hiermee nemen we "group" (g) en "others" de permissies "read" (r) en "execute" (x) af.

ls -l laat nu een andere matrix zien.

Stel nu dat we onze collega's wel inzicht in onze bestanden willen geven, maar geen "execute" rechten op programma's en scripts, dan kunnen we dat doen met het voorbeeld hiernaast.

Controleer met **ls -l** wat er gebeurd is.

Het zou natuurlijk onpraktisch zijn als we dit voor elke file of directory afzonderlijk zouden moeten doen. Daarom heeft het chmod programma - net als veel andere file/dir commando's - een extra **-R** flag (recursive), waarmee we in een keer alle permissies voor een directory en alle onderliggende subdirs en files kunnen wijzigen.

Aantekeningen

3

 Opdracht

```
#> chmod -R go-rwx * <enter>  
  
#> ls -l <enter>  
total 0  
drwx----- 2 rene staff 68 30 aug 14:16 bin  
drwx----- 2 rene staff 68 30 aug 14:16 lib  
drwx----- 2 rene staff 68 30 aug 14:16 tmp  
drwx----- 2 rene staff 68 30 aug 14:32 cfg
```

Zo, nu kunnen we alleen zelf aan onze bestanden prutsen! Oké, dat is niet waar, het algemene systeembeheer-account '**root**' heeft altijd onbeperkte toegang tot alle files op het systeem.

Je ziet dat in bovenstaand command gebruik gemaakt wordt van een sterretje (*) dit is een zgn. *wildcard*. Het sterretje gebruik je als substitutie voor bestands- of directory-namen. Typ je alleen een sterretje dan wil dat zeggen "alle bestanden/directory's", typ je bijvoorbeeld:

 Opdracht

```
#> chmod -R go+rwx c* <enter>  
#> ls -l <enter>
```

Zul je zien dat alleen de files/dirs die met een 'c' beginnen gewijzigd worden. In dit geval dus de cfg directory. Er is nog een andere wildcard, namelijk het vraagteken '?' - dit vervangt 1 letter. Kijk wat er gebeurt:

 Opdracht

```
#> chmod -R go+rwx ?i? <enter>  
#> ls -l <enter>
```

WERKEN MET FILES

Er zijn honderden, zo niet duizenden programma's (ik heb ze nooit geteld) op een Unix systeem die met bestanden werken. Van veel programma's is het redelijk duidelijk wat ze doen, en van anderen kun je alleen maar raden wat het doel of de meerwaarde is.

Door een redelijk bizarre mailtje van een transactie overzicht van het systeem kwam ik erachter dat ik een typefout in een script had gemaakt. In plaats van het date commando had ik blijkbaar ddate geschreven, en nou bleek dat toevallig ook een commando te zijn, alleen de output is, op z'n zachts gezegd, nogal bijzonder...

Gelukkig draait zo'n beetje het halve internet op Linux, dus dat soort informatie heb je redelijk snel bij elkaar gegoogled. En als het dan toch niet blijkt te bestaan, nou, dan maak je het gewoon zelf!

We gaan voor het gemak met gewone, platte, ASCII bestanden werken, maar het merendeel van de commando's werken ook met binary bestanden (afbeelding of iets dergelijks).



```
#> ddate
Today is Boomtime, the 23rd day of Bureaucracy in the YOLD 3181
```

Zo zijn er wel meer programma's (weliswaar niet zo nutteloos als dit) waarvan je in eerste instantie niet kunt achterhalen wat ze nou eigenlijk doen, maar ga er maar vanuit dat er voor ALLES wat je denkt dat je met een bestand kunt doen, wel een programma bestaat.



Een nieuw (leeg) bestand kun aanmaken met het commando ‘**touch**’. In principe is dit commando bedoeld om de date/timestamp van een bestand te wijzigen, maar als het bestand niet bestaat dan maakt het gewoon een nieuw bestand aan.

Opdracht

```
#> cd <enter>
#> touch bestand.txt <enter>
#> ls <enter>
```

Je ziet dat je nu een lege file met de naam `bestand.txt` in je home directory hebt staan.

Dit bestand willen we niet in de home directory maar in de `cfg` directory. We kunnen het bestand kopiëren met het commando `cp` (copy). En vervolgens het origineel verwijderen met `rm` (remove).

Opdracht

```
# Met cp en rm (copy en remove)

#> cp bestand.txt cfg <enter>
#> ls /cfg <enter>
bestand.txt
#> rm bestand.txt <enter>
#> ls <enter>

# Of met het mv (move commando)

#> mv bestand.txt cfg <enter>
#> ls /cfg <enter>
bestand.txt
#> ls <enter>
```

BESTANDEN BEWERKEN

Het is natuurlijk een beetje knullig om met een leeg bestand te moeten werken, dus gaan we het bewerken met de standaard editor die op elk Unix systeem aanwezig is, namelijk de beruchte vi-editor. Nu staat **vi** in dit geval voor ‘visual’, hoewel er niks visuals aan het ding is te ontdekken.

Als je een paar jaar van je leven opoffert om de editor enigszins te begrijpen, is het een fantastische tool, maar tot die tijd kun je het ding het beste gewoon ‘*Vreselijk Irritant*’ noemen...

Er zijn nog wel een paar editors beschikbaar (**nano** of **emacs** bijvoorbeeld, maar die zijn niet minder ingewikkeld).

vi draait in twee modi: de *edit mode* en de *command mode*. Met bepaalde toetscombinatie kun je tussen beide modi switchen. Dat vereist nogal wat oefening, dus in de volgende opdracht geef ik even aan op welke knoppen je moet drukken om iets voor elkaar te krijgen.

WERKEN MET VI

```
#> cd cfg <enter>
#> vi bestand.txt <enter>
```

Nu krijg je een leeg scherm te zien (je bent overigens in command mode nu). toets **<i>** (insert, switch naar edit mode)

type “Hallo, dit is een bestand.” druk op **<esc>** (nu switch je naar command mode) en type vervolgens **<:wq!>**

De **:** vertelt vi dat je er een commando komt
w = write (save)
q = quit (einde)
! = overschrijven (of als het bestand read-only is maar je wel eigenaar bent, toch gewoon uitvoeren)

In vroegere versies waren de pijltjes-toetsen niet geactiveerd. over het algemeen werken die nu wel en kun je tegenwoordig gelukkig ook met backspace en/of delete werken (in edit mode).



DE inhoud van bestanden tonen

We hebben een aantal tools tot onze beschikking om de inhoud van bestanden te tonen en te analyseren. Later volgen er nog meer commando's om je weg in bestanden te vinden, maar we beperken ons hier even tot de twee meest gebruikte en krachtigste commando's (voor in scripts) om gegevens uit bestanden te halen.

Om de inhoud van een (ASCII) bestand te bekijken kun je het **cat** (concatenate) commando gebruiken.

Opdracht

```
#> cat bestand.txt <enter>
```

Een ander veel gebruikt commando is **grep** (een patroonzoeker). Met **grep** kun je een bestand of een directory doorzoeken op bepaalde tekstpatronen. In z'n meest simpele vorm werkt het zo:

Opdracht

```
#> grep dit * <enter>
Hallo, dit is een bestand.
```

We vertellen hier met de wildcard (*) dat we **grep** willen uitvoeren op alle bestanden in de huidige directory. En we laten **grep** zoeken op de tekst 'dit'. In dit geval komt er maar 1 resultaat uit natuurlijk, maar maak maar eens een paar bestanden aan waarin het woord 'dit' niet of zelfs vaker voorkomt. **grep** is hoofdlettergevoelig. Dus 'dit' is niet hetzelfde als 'Dit'.

Dit kun je omzeilen door de **-i** flag te gebruiken:

```
grep -i DIT *
```



POW!

BESTANDEN ZOEKEN

Een aantal systeembeheer commando's moeten eigenlijk standaard tot je toolbox behoren. Zo'n beetje elke applicatie of tool die je op je server draait, zal op de een of andere manier wel een vorm van logging genereren.

De inhoud van zo'n logfile is natuurlijk sterk afhankelijk van de bouwer van de applicatie, meestal kun je er geen touw aan vast knopen – maar soms zullen deze logs wel inzicht kunnen bieden in wat er nou explicet fout ging in het systeem.

Op voorwaarde natuurlijk dat je log kunt vinden...

Hiervoor kun je het '**find**' commando gebruiken. Zoals we gezien hebben, hebben commando's een aantal flags en parameters. Find vormt hierop geen uitzondering. Je kunt het find commando met een aantal opties sturen:

Opdracht

```
# Alle bestanden vanaf de huidige directory  
#> find . <enter>  
  
# Alle bestanden vanaf een bepaalde directory  
#> find /var/log <enter>
```

'**find**' is een tooltje dat je dermate veel zult gaan gebruiken dat het verstandig is om hier uitgebreid bij stil te staan. Op de volgende pagina's tref je dan ook een werkbaar overzicht aan van de meeste opties die **find** zoal heeft.



Zoeken met 'find'

```
# zoeken op naam
#> find /tmp -name abc.txt <enter>

# zoeken op naam met wildcards
#> find /tmp -name *.txt <enter>

# negeer hoofd- en kleine letters
#> find . -iname abc.Php <enter>
#> find . -iname *.Php <enter>

# negeer hoofd- en kleine letters
#> find . -iname abc.Php <enter>
#> find . -iname *.Php <enter>

# zoek bestanden die niet aan een criterium
# voldoen
#> find . -not -name abc.php <enter>
#> find . ! -name abc.php <enter>

# of met wildcards
#> find . ! -name *.php <enter>

# of met invert case
#> find . ! -iname *.pHp <enter>

# Je kunt diverse criteria combineren,
# bijvoorbeeld:
# zoek alle bestanden die beginnen met "abc",
# maar niet de extensie "php" hebben.
#> find . -name abc* ! -name *.php <enter>

# Of alle bestanden die met "abc" beginnen
# maar wel de extensie "php" hebben
#> find . -name abc* -name *.php <enter>

# Zoek alleen bestanden van de user "applicatie"
#> find . -user applicatie <enter>
#> find -user applicatie -name *.php <enter>
```

Soms heb je natuurlijk een situatie waarbij een applicatie of systeem onderuit gaat en je niet weet *waar je het moet zoeken*. Ook hier biedt **find** een oplossing: je kunt zoeken op tijd.

● ● ● Zoeken met 'find'

```
# zoek bestanden die precies 20 dagen  
# geleden gewijzigd zijn:  
#> find . -mtime 20 <enter>  
  
# zoek bestanden die in de laatste 20 dagen  
# geraadpleegd zijn:  
#> find . -atime -20 <enter>  
  
# zoek bestanden die in de laatste 5 dagen  
# gewijzigd zijn:  
#> find . -mtime -5 <enter>  
  
# zoek bestanden die in de laatste 10 minuten  
# gewijzigd zijn:  
#> find . -cmin -10 <enter>
```

Soms dreigt je harddisk vol te lopen omdat een of ander process of programma enorm veel logging of fouten genereert. **find** kan ook hier weer de helpende hand bieden:

● ● ● Zoeken met 'find'

```
# zoek de bestanden tussen de 50 en 100MB  
#> find . -size +50M -size -100M <enter>  
  
# zoeken naar directories  
#> find / -type d -name "log" <enter>  
  
# zoeken naar bestanden met de extensie "log"  
#> find . -type f -name ".log" <enter>
```

Zoals je ziet is het **find**-commando een van de belangrijkste tools die je als systeembeheerder tot je beschikking hebt.

DE inhoud van bestanden filteren

Om de inhoud van een bestand te bekijken / analyseren hebben we een aantal tools tot onze beschikking. We hadden eerder al het commando ‘**cat**’ gezien, hiermee kunnen we de inhoud van een file bekijken. Bij grote bestanden is dit niet echt handig omdat de inhoud dan over het scherm vliegt.



```
#> cat *.log | grep -i error > fouten.txt <enter>
```

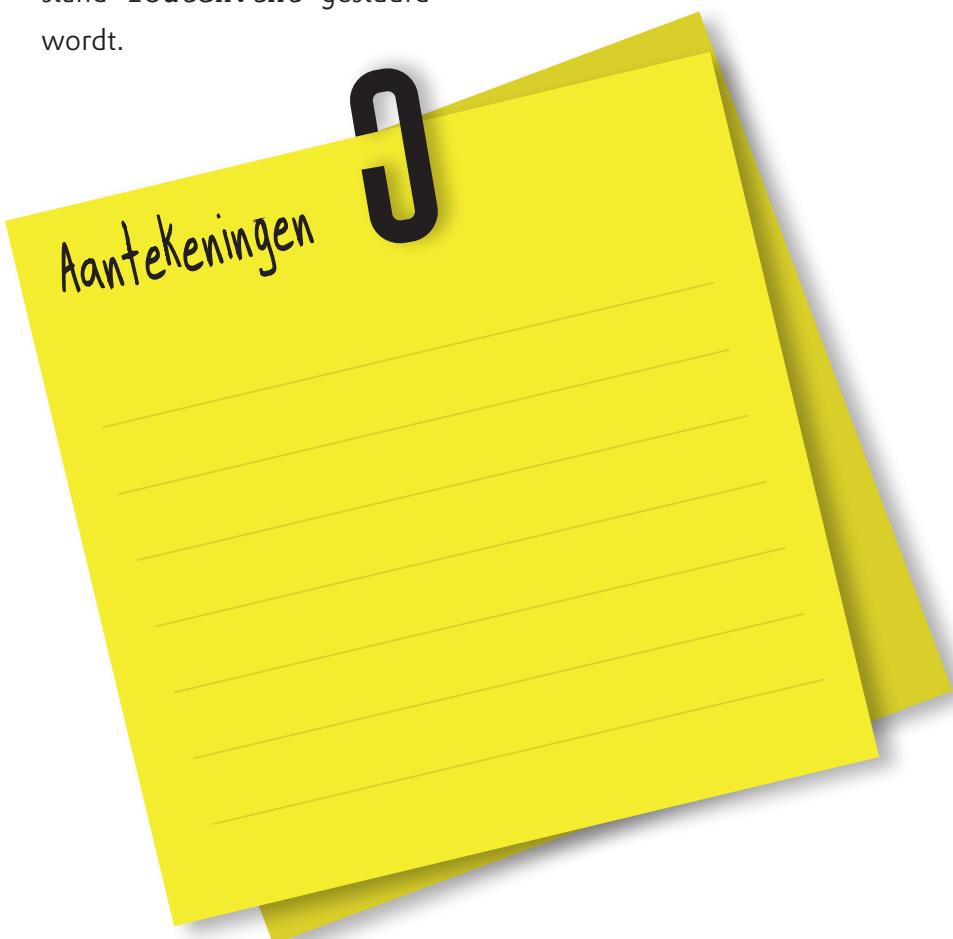
Het ‘**cat**’ commando kan prima gebruikt worden om de output van de ene file naar de andere te sturen. In combinatie met het eerdere ‘**grep**’ commando, kun je hier je eigen tool maken om informatie te analyseren.

Wat hier gebeurt is dat van elke file met de extensie ‘log’ de inhoud gescand wordt op de tekst ‘error’ (middels grep) en die output naar het bestand **fouten.txt** gestuurd wordt.

Maar om output naar het scherm te sturen en visueel te analyseren, is **cat** niet echt handig. Daartoe zijn een aantal andere commando’s een stuk praktischer in het gebruik.

DE COMMANDO’S TAIL EN HEAD

De commando’s ‘**tail**’ en ‘**head**’ doen eigenlijk hetzelfde. Met ‘**tail**’ kunnen de laatste regels van een bestand bekijken worden, met ‘**head**’ de eerste regels. Het **tail**-commando heeft nog een extra, zeer handige, optie. Dat is namelijk de **-f** flag (follow). Hiermee kan een bestand realtime een gevuld worden.



 Opdracht

```
# toon de eerste 10 regels in een bestand  
#> head -n 10 somefile.txt <enter>  
  
# toon de laatste 10 regels in een bestand  
#> tail -n 10 somefile.txt <enter>  
  
# Een bestand "volgen" met tail -f  
#> touch data.txt <enter>  
#> tail -f data.txt <enter>  
  
# open een tweede terminal en "echo" iets naar het bestand, kijk in je eerste terminal wat er gebeurt.  
#> echo "Dit is een regel" >> data.txt <enter>
```

Andere handige tools die ongeveer hetzelfde doen

more: dit doet eigenlijk hetzelfde als cat, maar laat de output 'per scherm' zien.

less: bijna gelijk aan more.

cat | more: gelijk aan more

 Opdracht

```
#> more somefile.txt <enter>  
#> less somefile.txt <enter>  
#> cat somefile.txt | more <enter>
```

FILTERS

In het vorige hoofdstuk hebben we bestanden gezocht en bekijken, nu gaan we iets dieper in op het zoeken van informatie in deze bestanden en kijken hoe we een aantal bewerkingen op deze data kunnen uitvoeren.

Hiertoe hebben we een hele set aan tools tot onze beschikking die handig kunnen zijn in de dagelijkse praktijk.

Download nu de bestanden middels het '**wget**' (web-get) commando. **wget** is een tool om via de command line allerlei HTTP-requests naar een website af te kunnen vuren. Over het algemeen is de tool handig om bestanden direct te downloaden naar je lokale systeem.



Downloaden sampledata

```
# download en unzip, middels
# onderstaande commandoreeks,
# de voorbeeldbestanden naar je
# werk-directory

#> cd $HOME <enter>
#> mkdir sampledata <enter>
#> cd sampledata <enter>

#> wget http://educom.nu/linux-cli.zip
<enter>

#> ls <enter>
#> unzip linux-cli.zip <enter>
#> ls <enter>

# Als alles gelukt is, staan er nu een
# aantal bestanden die we in de komende
# opdrachten gaan gebruiken in de
# directory "sampledata" in je
# home-directory.
```

Aantekeningen

9

SORTEREN VAN DE UITVOER

We kunnen middels het commando ‘**sort**’ de uitvoer (alfabetisch)sorteren:

Opdracht

```
#> cat sampledata.txt <enter>
#> sort sampledata.txt <enter>

# of via een pipe
#> cat sampledata.txt | sort <enter>
```

UNIEKE WAARDEN FILTEREN

Het commando ‘**uniq**’ haalt de unieke waarden (regels) uit het bestand:

Maar wat gebeurt er nu? Je ziet dat dubbele regels verdwijnen... echter... alleen als die dubbele regels **PRECIES NA** elkaar staan.

Met **uniq -c** kun je tellen hoe vaak de betreffende regels voorkomen. Dat is leuk en aardig, maar stel de gegevens komen uit een database dan kan het natuurlijk heel goed zijn dat de data niet precies achter elkaar in de output terechtkomt. Piping kan hier dus een uitkomst bieden!

Opdracht

```
#> uniq sampledata.txt <enter>

# tellen...
#> uniq -c sampledata.txt <enter>

# via een pipe...
#> cat sampledata.txt | sort | uniq -c <enter>
```

Kijk, dat ziet er al een stuk bruikbaarder uit.

Opdracht

```
# nummer de regels in een bestand  
#> nl sampledata.txt <enter>  
  
met een aantal opties:  
#> nl -s '. ' -w 1 sampledata.txt  
  
# toon het aantal regels of woorden in  
# een bestand  
# dit commando laat achtereenvolgens  
# het aantal regels, woorden en  
# karakters zien:  
#> wc sampledata.txt <enter>  
289      867     4896 sampledata.txt  
  
# regels:  
#> wc -l sampledata.txt <enter>  
  
# woorden:  
#> wc -w sampledata.txt <enter>
```

TELLEN

Vaak willen we weten hoeveel regels er in een bestand zitten, of hoeveel misschien zelfs hoeveel woorden. De commandos ‘**nl**’ (number lines) en ‘**wc**’ (word count) doen precies dat. ‘**nl**’ zet een nummer voor de regels (handig bij bijvoorbeeld source-code) en ‘**wc**’ telt het aantal woorden of regels in een bestand.

Opdracht

```
# toon het bestand telefoonnummers.txt  
#> cat telefoonnummers.txt <enter>  
  
# wijzig telefoonnummers  
#> sed 's/06/00316/g' telefoonnummers.  
txt <enter>  
  
# s: geeft aan dat je gaat zoeken naar  
# de tekst die na de eerste slash  
# staat, en dat je deze wilt  
# vervangen door de # text na  
# de tweede slash.  
# g: geeft aan dat dit voor elke regel  
# moet gebeuren (global).  
  
# De commando's die je aan 'sed' kunt  
# meegeven zijn dezelfde die je in 'vi'  
# kunt gebruiken.  
  
# LET OP! Gebruik (de goeie) quotes  
# (enkele) als je het commando  
# uitvoert!
```

GEGEVENS WIJZIGEN

Een andere handige tool is het programmaatje ‘**sed**’ - streaming editor. ‘**sed**’ is een beetje een rare omdat deze met regular expressies werkt, maar dan op zijn eigen manier.

Waar het met name voor gebruikt wordt is om gegevens te corrigeren. Bekijk het voorbeeldbestand telefoonnummers.txt. Nu willen we eigenlijk dat alle 06-nummers in dit bestand met **0031** gaan beginnen en willen we de voorloop **0** verwijderen. Dat kan heel eenvoudig met het ‘**sed**’ commando.

TABELLEN EN KOLOMMEN

'**cut**' is een handige tool om bepaalde kolommen te filteren uit een bestand (of stream). Met de **-d** optie kun je aangeven welke 'delimiter' of scheidingsteken gebruikt kan worden om je tabel te 'cussen', met de **-f[nummer]** optie geef je aan welke kolom je wil zien:

Opdracht

```
# tweede kolom:  
#> cut -d " " -f2 sampledata.txt <enter>  
  
# eerste kolom:  
#> cut -d " " -f1 sampledata.txt <enter>
```

Bij het inlezen van een bestand '**cut**' je eigenlijk al vanzelf, kijk maar wat er gebeurt:

Opdracht

```
#> while read line <enter>  
> do <enter>  
> echo "Regel: $line" <enter>  
> done < sampledata.txt <enter>  
  
# tweede kolom:  
#> while read line <enter>  
> do <enter>  
> echo "$line" | cut -d" " -f2 <enter>  
> done < sampledata.txt <enter>
```

Zo lees je het hele bestand regel voor regel. Deze regel wordt in de variabele 'line' gestopt en met '\$line' kun je waarde vervolgens ophalen en er iets mee doen. Maar vaak wil je natuurlijk alleen een specifieke kolom gebruiken. Dat kun je op twee manieren doen, met '**cut**' en met de leesopdracht zelf.



```
# Output formateren:  
#> while read naam fruit aantal  
do  
echo "$naam heeft $aantal $fruit gekocht."  
done < sampledata.txt <enter>  
  
# SQL statement maken:  
#> while read naam fruit aantal  
do  
echo "insert into tabel values('$naam', '$fruit',  
$aantal);"  
done < sampledata.txt <enter>
```

Vaak wil je meer controle over de vorm van de output. Bijvoorbeeld om een HTML-rapport te genereren, of om een SQL-insert script te maken met de data in de bron-bestanden. ‘**cut**’ en ‘**read**’ kunnen hier prima voor gebruikt worden.

Zo zie je dat je bestanden kunt lezen en verwerken - al naar gelang de behoefte natuurlijk. Door deze commando’s slim te combineren kun je zelf hele krachtige scripts bouwen die je dagelijkse werk kunnen veraangenamen.

Verder op in deze workshop gaan we deze commando’s in een uitgebreid script combineren. Je zult zien dat de meeste third-party applicaties die op een linux-systeem draaien over het algemeen ook een cmdline toolkit hebben, waardoor het combineren van commando’s en programma’s nog flexibeler wordt.

Neem je bijvoorbeeld de uitgebreide (web) scripttalen als PHP en Perl - maar ook Ruby en zelfs JavaScript (via bijvoorbeeld node.js) en combineer je deze met linux-shell scripting, dan kun je uitermate krachtige tools bouwen waar geen user-interface of interactie aan te pas komt.

Ook hebben de meeste RDBMS (Oracle, Progress, Mongo en MySQL bijvoorbeeld) systemen een cmdline interface waarmee je via de shell data kunt raadplegen of manipuleren.

Aantekeningen

3



**UNIX GUYS
DO IT ON THE
COMMAND LINE!**

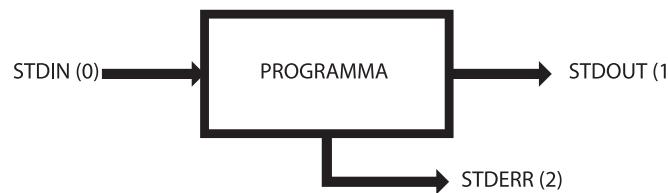
GEAVANCEERDERE MOGELIJKHEDEN

PIPING & REDIRECTION

We hebben het nu al een aantal keren gebruikt in de voorbeelden, maar wat is het nou eigenlijk.

Elk programma heeft 3 zogenaamde 'data streams' waar het aan moet voldoen om het tot de unix-commando's te kunnen laten behoren:

- **STDIN (0)** de data die we aan het programma kunnen meegeven
- **STDOUT (1)** de output die het programma genereert
- **STDERR (2)** de eventuele foutsituatie die optreedt.



We kunnen zelf gaan sturen wat we als input leveren en waar de output van een commando naar toe moet. Ook kunnen we bepalen wat we met fouten doen. Je hebt in vorige commando's vaak de foutmelding "Permission denied" gekregen (bijvoorbeeld bij **find**). Dat is natuurlijk lastig als je een lijst van bestand wil genereren waar je vervolgens met bijvoorbeeld '**cut**' of '**sort**' doorheen wil lussen.

Als je bijvoorbeeld **/var/log/syslog** wilt bekijken, krijg je waarschijnlijk een "cat: syslog: Permission denied" error.



Opdracht

```
#> cat /var/log/syslog <enter>
cat: syslog: Permission denied

Fout niet tonen:
#> cat /var/log/syslog 2> errors.txt
<enter>

Nou zie je dat je foutmelding in errors.txt staat
```

Wat we in het vorige voorbeeld gedaan hebben is dat we STDERR naar het bestand errors.txt geredirect hebben (door de toevoeging **2>**)

INPUT & OUTPUT STUREN

We hebben het al een aantal keer gedaan maar we kunnen de output van een commando naar een bestand sturen middels de volgende 2 toevoegingen:

- **>** overschrijf of maak het bestand met output van het commando
- **>>** voeg de output van het commando toe aan de file.
- **<** lees uit een specifiek file of stream



Opdracht

```
#> sed 's/06/00316/g'
      telefoonnummers.txt > modified.txt
<enter>
#> cat modified.txt <enter>

# En vervolgens:
#> sed 's/06/00316/g' telefoonnummers.txt >> modified.txt
<enter>
#> cat modified.txt <enter>

# Input:
#> wc -l < telefoonnummers.txt
<enter>
```

PIPING

Piping is de techniek waarmee we de output van het ene commando de input voor het volgende laten zijn. Een 'pipe' geef je aan met '|'.

Opdracht

```
#> cat sampledata.txt | sort | uniq -c <enter>
```

Eerst '**cat**'-ten we **sampledata.txt**, de output van **cat** is vervolgens de input van **sort**, dat vervolgens deze informatie sorteert en die gesorteerde informatie is op zijn beurt weer de input voor het **uniq** commando.

Je kunt zoveel commando's achter elkaar zetten als je zelf wil, maar zoals bij alle automatisering-dingetjes is **overzicht** uiteraard een belangrijk criterium.

Aantekeningen

9

XARGS

Een speciale vorm van piping zijn de zogenaamde “**xargs**”. Dit is eigenlijk een loop zonder te lopen. Handig als je bijvoorbeeld voor elke file in een **ls** comando iets wil uitvoeren.

Opdracht

```
#> cd sampledata <enter>
#> ls <enter>
01.PNG      02.png      03.PNG      04.png
sampledata.txt          telefoonnummers.txt

#> basename -s .PNG -a *.PNG | xargs -n1 -i mv
{}.PNG {}.png <enter>
#> ls <enter>
01.png      02.png      03.png      04.png
sampledata.txt          telefoonnummers.txt
```

TOELICHTING

- Eerst gebruiken we het commando ‘**basename**’ om de naam van het bestand zonder directory op te vragen.
- De **-s** optie geeft aan welke extensie we willen vervangen
- De **-a** optie geeft aan dat we meerdere bestanden gaan raadplegen (type ***.PNG**)
- Hier komt een lijst uit met bestanden die aan het ***.PNG** criterium voldoen
- Dit ‘pipen’ we naar **xargs**, waarbij **-n1** aangeeft dat het commando dat volgt precies 1x per input-regel moet worden uitgevoerd en vervolgens hernoemen we met **mv** (move) de file naar een nieuw bestand met een lowercase extensie.

Aantekeningen



-EXEC

Nagenoeg hetzelfde als **xargs** is de **-exec** optie. De **-exec** optie is vooral bekend bij het find commando. Met **-exec** kunnen voor elke regel input eenzelfde actie uitvoeren:

Opdracht

```
#> find /var/log -size +2M -exec ls -sh {} \; <enter>
```

{ } wil zeggen dat dit voor 'elk bestand' moet gebeuren. Het exec commando moet afgesloten worden met een ; (punt-komma), maar dit betekent op de commandline ook iets anders (nieuw commando) dus om de zaak "zeer duidelijk" te maken moet deze punt-komma 'ge-escaped' worden met een \ (backslash).

Duidelijk? Nee, natuurlijk niet.

Gewoon zo gebruiken...

Aantekeningen

9

BESTANDEN EN DIRECTORIES COMPRIMEREN

Om bestanden en directories te comprimeren (bijvoorbeeld voor backup doeleinden) kunnen we een aantal commando's gebruiken.

● ● ● Archiveren & Uitpakken

```
# Standaard syntax om een directory te archiveren  
#> tar -zcvf [archive] [dir-name] <enter>
```

Flags:

- z: zip (comprimeer)
- c: create (maak het archief bestand)
- v: verbose (toon de acties op het scherm)
- f: de filename

```
# Archiveer de /tmp directory:
```

```
#> tar -zcvf temp.tar.gz /tmp <enter>
```

```
# Uitpakken
```

```
#> tar -xvf temp.tar.gz <enter>
```

Het meest gebruikte commando is het zgn. **tar** commando. Dit is oorspronkelijk bedoeld om gegevens naar een tape te schrijven (**tar** staat dan ook voor "tape archiver"), maar wordt tegenwoordig vooral als compressie tool gebruikt.

Naast **tar** bestaat er ook een command line variant van de bekende **zip** tool. Meestal zie je dat in de unix-wereld het **tar**-commando toch de voorkeur geniet.

Aantekeningen

3

SHELL VARIABELEN

Je zag al bij het ‘**echo \$PATH**’-commando dat er ‘speciale’ elementen in de shell zitten. Door het **\$**-teken te gebruiken roepen we de zogenaamde shell-variabelen aan. We kunnen onze command line omgeving dus gebruiken om tussentijds waardes op te slaan voor later gebruik, of om onze omgeving in te richten zodra we inloggen.

Shell-variabelen (datum-tip)

```
# Zet de datum in een shell variabele
# naar goed gebruik worden variabelen altijd met
# HOOFDLETTERS geschreven.
#> DATUM=`date +%Y%m%d` <enter>
#> echo $DATUM <enter>

# Voorbeeld in het gebruik:
#> tar -zcvf $DATUM.tar.gz ./archive <enter>
```

We kunnen onze shell naar eigen gebruik inrichten. Bijvoorbeeld met het ‘**alias**’ commando. Met **alias** kunnen we complexe(re) commando’s reduceren tot een ‘shortcut’. Bijvoorbeeld als we regelmatig het ‘**ls -l**’ commando invoeren, kunnen we in **\$HOME/.bash_profile** het volgende commando instellen:

De shell configureren

```
# bewerk je .bash_profile
#> vi $HOME/.bash_profile <enter>
<i>
alias ll='ls -l'
<esc><wq!>

# Log uit en opnieuw in om de wijzigingen te zien

# Kijk wat er nu gebeurt:
#> ll <enter>
```

Het **\$HOME/.bash_profile** script is een scriptje dat elke keer dat je inlogt uitgevoerd wordt. Hier zou je allerlei veel gebruikte paden of ‘**alias**’-en in variabelen kunnen zetten bijvoorbeeld.

CRONTAB & JOBS

Een standaard feature van elk unix systeem is de zogenaamde. 'cron', oftewel een command scheduler.

De **crontab** (de 'database' waar de achtereenvolgende commando's staan) kan bewerkt worden met het commando: **crontab -e** (edit)

● ● ● Jobs schedulen

```
#> crontab -e <enter>

Positie
1 2 3 4 5 6
* * * * * /bin/execute/this/script.sh

# Voorbeelden:
# Elke vrijdag om 13:00 uur
0 13 * * 5 /bin/execute/this/linux-workshop.sh

# Elke 10 minuten:
0,10,20,30,40,50 * * * * /bin/script.sh
*/10 * * * * /bin/script.sh
```

Positie

- 1: minuten (0 t/m 59)
- 2: uur (0 t/m 23)
- 3: dag van de maand (1 t/m 31)
- 4: maand (1 t/m 12)
- 5: dag van de week (0 t/m 6, 0 = zondag, 6 = zaterdag)
- 6: script of programma dat uitgevoerd moet worden

LET OP!!

De cron maakt standaard gebruik van de /bin/sh shell, dus als je script gebruik maakt van /bin/bash, neem dit dan op in je script-header:

Denk er ook aan dat alle PATH verwijzingen die je gebruikt ook in het script aanwezig moeten zijn!!

PROCESSEN

Als systeembeheerder heb je vrijwel dagelijks te maken met processen die stuk lopen of performance problematiek of dingen die gewoon niet (willen) werken. Dat is over het algemeen lastig te onderzoeken. Een performance probleem is zelden tot nooit direct aanwijsbaar en zal wat onderzoek nodig hebben om uiteindelijk de 'root-cause' te kunnen vaststellen.

Het systeem kan ons (los van de logging van het betreffende programma) wel wat inzicht bieden in wat er zoal gaande is. Met het commando '**ps**' kunnen we zien welke processen draaien.



Processen zoeken

```
#> ps aux <enter>

# Je ziet nu per user het PID (process ID),
# het % CPU Tijd, het % Memory belasting
# en het volledige pad naar het programma.
# De uitvoer is redelijk analoog aan 'top'
# maar top ververst de info realtime.

#> top <enter>
```

Als we de boosdoener gevonden hebben kunnen we het process meestal met het commando '**kill**' stoppen. De naam is redelijk zelf-verklarend. Echter de parameters zijn dat niet.

Echt **STOPPEN** doe je met '**kill -9 [pid]**', dat is echt definitief. Het probleem is echter dat **kill** in dit geval alleen het originele process stop, heeft dit process nog zogenaamde child-processen ge'spawn'd dat zitten we opgescheept met een aantal 'zombie' processen (processen zonder parent) die resources blijven vreten. In dat geval is een reboot eigenlijk de enige optie om dit adequaat op te lossen.

PERFORMANCE

Met het commando '**uptime**' krijg je een kort overzicht hoe druk de machine is. Uptime laat je de load-average zien van 1, 5 en 10 minuten.

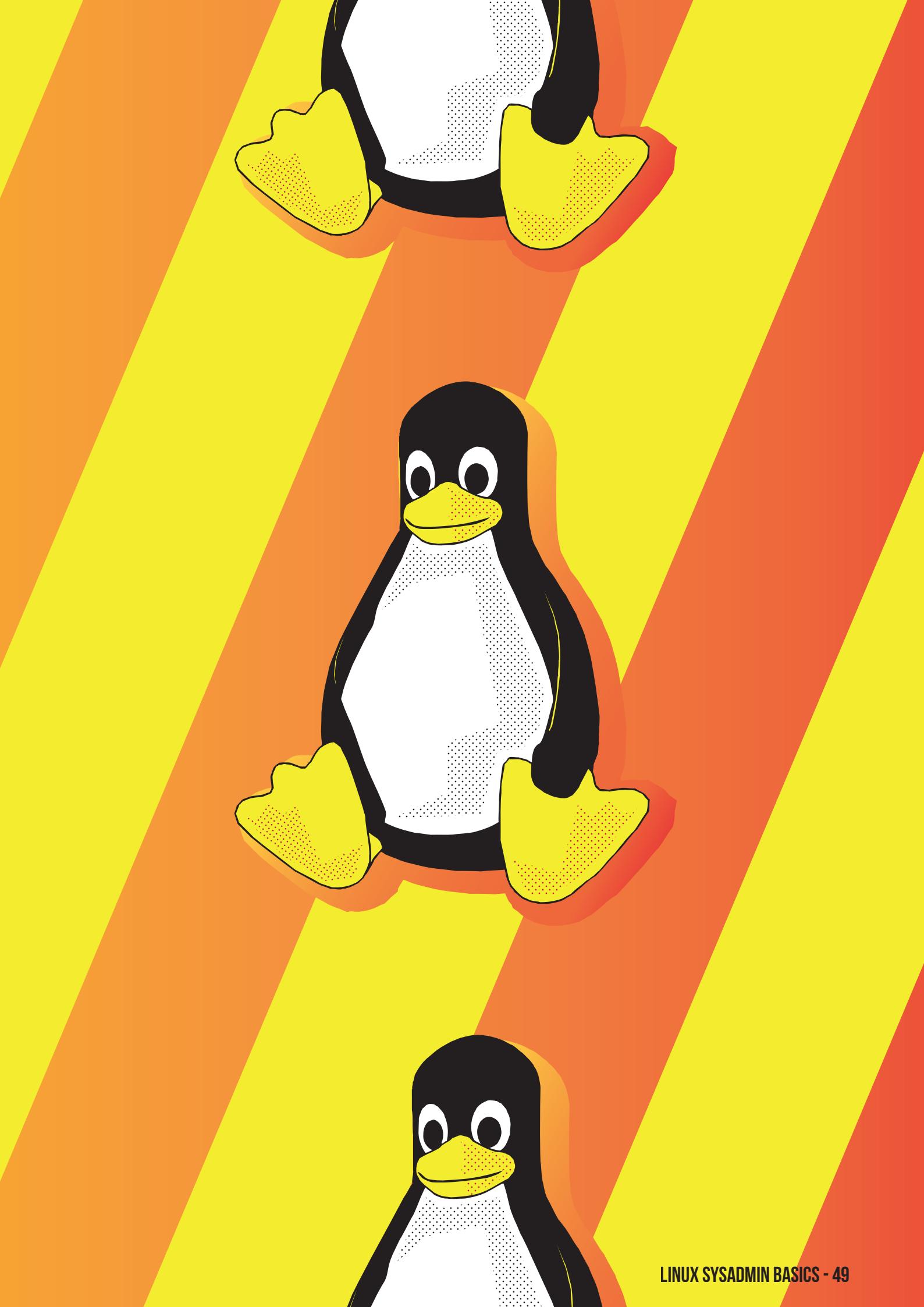
Zodra de load-average boven de 1 komt (vuistregel) dan is er meestal iets aan de hand. Dit is echter geen indicatie dat er ook daadwerkelijk iets aan de hand is, veel database systemen trekken die load average al gauw naar een hoger niveau omdat ze redelijk excessief van het RAM gebruik maken. Dus een en ander is sterk afhankelijk van wat de machine moet doen.

Betere informatie kun je krijgen door het pakket '**sysstat**' te installeren. **sysstat** is echter nogal complex in het gebruik en is meer voor een 'advanced' cursus bedoeld, we zullen hier dan ook niet verder op ingaan.



Performance insight

```
#> uptime <enter>  
  
12:17:00 up 54 days, 20:45, 1 user,  
load average: 0.01, 0.02, 0.05  
  
# sysstat package installeren  
#> sudo apt-get install sysstat <enter>  
#> sudo vi /etc/default/sysstat <enter>  
# wijzig ENABLED="false" in ENABLED="true"  
  
#> sudo service sysstat restart <enter>  
  
#> sar <enter>  
#> iostat <enter>  
#> dstat <enter>
```



REMOTE CONNECTIONS & THIRD PARTY SOFTWARE

Als het goed is, is **ssh** (Secure Shell) al per definitie op je systeem geïnstalleerd. Was het in vroegere tijden nog een extra tool die je afzonderlijk moest installeren, tegenwoordig is het de ‘preferred method’ om connectie te maken met een (remote) systeem.

SSH

Het voert wat ver om in deze basis cursus de precieze in’s- en out’s van ssh en de configuratie hiervan te bespreken, omdat dat bijna een cursus op zich is. Hier komt namelijk ook de configuratie van firewalls en networking in het algemeen bij kijken. Met een standaard installatie van ssh kun je over het algemeen prima uit de voeten.

De ssh commando-set (ook bijvoorbeeld de scp en sftp commando’s), maakt gebruik van een aantal configuratie bestanden in **/etc/ssh** (Linux versie afhankelijk) en van configuratie bestanden in je **\$HOME/.ssh** directory.

Zoals je al eerder gezien hebt zijn .-files of .-directories “hidden”, je zult ze niet zien met een reguliere ls bijvoorbeeld (met **ls -a** zie je ze wel).

In **\$HOME/.ssh/config** worden de diverse remote hosts/computers opgenomen en in **\$HOME/.ssh/known_hosts** staan de computers waarmee al eens verbinding is gemaakt. Hierin staan ook de keys/certificaten op basis waarvan de verbinding wordt gelegd.

Bij een eerste verbinding wordt een zogenaamde ‘fingerprint’ gegenereerd waarmee een tijdelijk certificaat wordt aangemaakt. Door ‘yes’ te kiezen, wordt deze key in

\$HOME/.ssh/known_hosts

opgenomen.

Hierna wordt gevraagd om in te loggen met je eigen username (van het locale systeem) en password van de corresponderende user op het remote systeem. Je zult dus op beide machines hetzelfde useraccount moeten hebben.

Een andere mogelijkheid is om in te loggen met:

ssh user@remotehost

zo log in je op de remote machine onder een specifieke user-account.

Remote Login

```
#> ssh educom.nu <enter>
The authenticity of host 'educom.
nu (91.184.11.235)' can't be estab-
lished.
ECDSA key fingerprint is SHA256:t-
baIszHp01mXfFhcIF0JRFrp7u9a2JOb-
6q3ItLYOMxk.
Are you sure you want to continue
connecting (yes/no)?

# ANTWOORD HIER MET 'no'

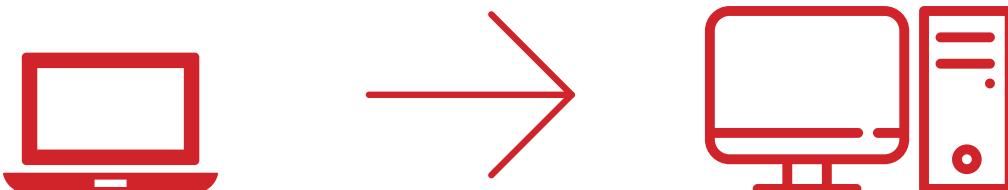
#> ssh workshop@educom.nu <enter>
workshop@educom.nu's password:
workshop <enter>

# ANTWOORD NU MET 'yes'
```

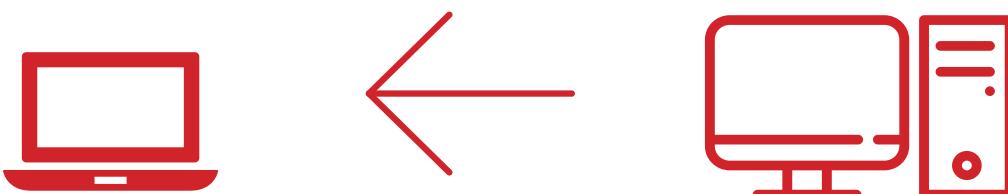
SCP

Het **scp**-commando is een tool uit de ssh-commando set en stelt je in staat om een bestand (of bestanden en directories) te kopiëren naar een remote host.

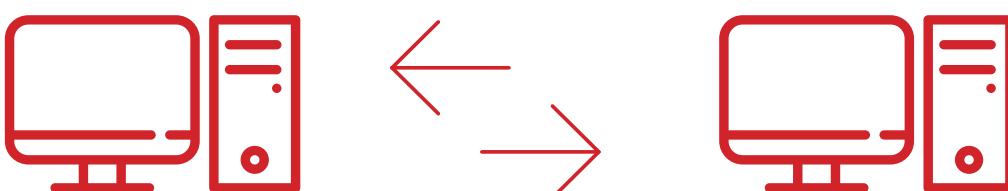
De algemene syntax is (vanaf je eigen pc/laptop):



```
scp [file/dir] username@remote:/path
```



```
scp username@remote:/path [local-path]
```



```
scp user@remote-1:/path  
user@remote-2:/path
```

PASSWORDLESS SSH

Je ziet dat we bij elke actie weer opnieuw het wachtwoord moeten invoeren. In scripts en batch-verwerkingen is dat natuurlijk reuze onhandig. Gelukkig kunnen we in de ssh tools gebruik maken van 'passwordless ssh'. Daartoe moeten we een aantal stappen uitvoeren:

● ● ● Passwordless ssh instellen

```
# we moeten eerst een reeks commando's
# uitvoeren voordat we zonder wachtwoord
# in kunnen loggen

#> ssh-keygen <enter>
# Er wordt om een 'passphrase' gevraagd voor
# extra beveiliging, deze kun je nu gewoon
# 'door-enteren'. Vervolgens krijg je wat rare
# output.

#> ssh-copy-id workshop@educom.nu <enter>
# Dit stuurt de genereerde key naar de remote
# machine. Check of het werkt met:

#> ssh workshop@educom.nu <enter>
```

Aantekeningen

REMOTE COMMANDO'S UITVOEREN

Met ssh kunnen we op de remote machine commando's uitvoeren, dit gaat vrij eenvoudig door de commando-string als parameter aan het **ssh**-commando mee te geven. Zo kun je bijvoorbeeld een directory aanmaken, maar ook – met root-toegang – hele desastreuze dingen doen.
Kijk hier dus mee uit!!

● ● ● Remote commando's uitvoeren

```
#> ssh workshop@educom.nu  
      'mkdir /home/workshop/[je-voornaam]' <enter>  
#> ssh workshop@educom.nu <enter>  
#> ls <enter>
```

Aantekeningen

9

FTP

Zodra je **scp** up and running hebt, vervalt eigenlijk de noodzaak tot **FTP** (file transfer protocol)

- **sftp** is de secure versie van **ftp**.

Binnen **(s)FTP** heb je de mogelijkheid om bijvoorbeeld directories te maken en het heeft een interactieve 'shell' omgeving. Maar zoals je in de vorige paragrafen zag, kun je met de combinatie **ssh** en **scp** hetzelfde realiseren. Vooral binnen scripts is het veel handiger om de **ssh/scp** combinatie te gebruiken dan een (s)FTP script er nog eens bij te moeten maken.

Hoewel **FTP** erg handig is vanuit een GUI omgeving (vanuit Windows bijvoorbeeld), is het als scripting tool niet echt heel geschikt.

Daarbij komt nog eens dat je op linux systemen tegenwoordig niet per definitie toegang hebt tot **ftp**-sessie en moet dit door de systeembeheerder los geconfigureerd (en dus ook beheerd) worden. Bij **ssh** en **scp** is dit niet nodig.



```
#> cd $HOME/samplefiles
#> ftp -p educom.nu <enter> # -p passive mode
Connected to educom.nu.
220 ProFTPD 1.3.5b Server (ProFTPD) [91.184.11.235]
Name (educom.nu:workshop): workshop <enter>
331 Password required for workshop
Password: workshop <enter>
ftp>

# Op de FTP-prompt kun je nu commando's uitvoeren.
```

FTP commando's op de prompt

```
ftp> ls <enter>
ftp> pwd <enter>
ftp> mkdir [je naam] <enter>
ftp> cd [je naam] <enter>
ftp> ls <enter>

# Vertel FTP dat je ascii bestanden gaat versturen
ftp> ascii <enter>
200 Type set to A

# Vertel FTP dat je binary bestanden gaat versturen
ftp> binary
200 Type set to I
ftp> put [bestand] <enter>
ftp> get [bestand] <enter>
ftp> lcd samplefiles <enter>
ftp> mput * <enter>
```

FTP commando's in een script

```
#!/bin/bash
HOST='educom.nu'
USER='workshop'
PASSWD='workshop'
FILE='file.txt'

ftp -n -p $HOST <<END_SCRIPT
quote USER $USER
quote PASS $PASSWD
put $FILE
quit
END_SCRIPT

exit 0
```

Maar zoals gezegd, het is een stuk handiger en praktischer om **ssh** en **scp** te gebruiken, je ziet hier de 'interne piping':

```
<< END_SCRIPT
END_SCRIPT
```

dat het mogelijk maakt een reeks commando's naar **ftp** te sturen. Dus los van de standaard unix-commando's moet je ook de betreffende FTP commando's leren.

MYSQL & GIT

Tot nu toe hebben we voorzichtig van de - toch wel - zeer uitgebreide mogelijkheden van de linux-command line mogen proeven. De hele set aan commando's die de revue gepasseerd zijn, vereisen natuurlijk wat oefening in het gebruik en je zult ze uiteraard veelal gaan gebruiken in de omgeving waarin je werkt.

In dit hoofdstuk gaan we uit van het gebruik van MySQL als database en BitBucket als source control tool. Andere databases en tools zullen ongeveer vergelijkbare command line tools hebben. BitBucket is echter in zoverre geen "vreemde eend" omdat de interface gewoon de '**git**' tool is - en dus eigenlijk exact hetzelfde werkt als **git** zelf en/of een eventuele lokale git-server installatie.

In dit hoofdstuk gaan we ervan uit dat er een (werkende) MySQL omgeving op de computer geïnstalleerd is en dat de git command line toolkit ook aanwezig is - dit kun je testen door achtereenvolgens de commando's '**mysql**' en '**git**' op te starten.

Als de output iets is in de geest van:

```
mysql: command not found  
      en/of  
git: command not found
```

dan zullen deze geïnstalleerd moeten worden.

● ● ● Installeren MySQL & GIT

```
# Repository updaten  
#> sudo apt-get update <enter>  
  
# GIT Installeren  
#> sudo apt-get install git <enter>  
  
# MySQL Installeren  
#> sudo apt-get install mysql-server  
<enter>
```

Het installeren van MySQL is iets uitbreider en start vervolgens een configuratie tool. Hier moet je het wachtwoord voor de MySQL-user 'root' invoeren (dit is niet de linux user `root`).

Log vervolgens in als de root-user en maak een database:

● ● ● Aanmaken database

```
#> mysql -u root -p[wachtwoord]  
<enter>  
  
# vervolgens start de commandshell  
# voor mysql  
  
mysql> create database unix_basics;  
<enter>  
  
# We kunnen eventueel nog een extra  
# gebruiker aanmaken, maar voor het  
# gemak gebruiken we  
# de root-user in de rest van de  
# opdrachten.
```

Om iets met deze database te kunnen, zullen we natuurlijk minimaal een tabel moeten hebben waarin we data gaan opnemen.

MySQL Commando's uitvoeren

```
# Eerst database "usen":  
mysql> use unix_basics; <enter>  
Database changed  
  
# Vervolgens een tabel maken:  
mysql> CREATE TABLE `orders` (  
    ->   `id` int(11) NOT NULL,  
    ->   `naam` varchar(50) NOT NULL,  
    ->   `fruit` varchar(50) NOT NULL,  
    ->   `aantal` int(11) NOT NULL  
    -> ); <enter>  
Query OK, 0 rows affected (0,02 sec)  
  
# Primary Key aanleggen:  
mysql> ALTER TABLE `orders`  
    ->   ADD PRIMARY KEY (`id`); <enter>  
  
# Autonummering toevoegen:  
mysql> ALTER TABLE `orders`  
    ->   MODIFY `id` int(11) NOT NULL  
    ->   AUTO_INCREMENT; <enter>  
  
# En wat data inserten:  
mysql> insert into orders values  
    (NULL, 'Sjaak', 'Bananen', 20); <enter>  
mysql> insert into orders values  
    (NULL, 'Sjaak', 'Bonen', 30); <enter>  
mysql> insert into orders values  
    (NULL, 'Sjaak', 'Aardbeien', 10); <enter>  
  
# Controleren:  
mysql> select * from orders; <enter>
```

Zo, nu hebben we een database en wat gegevens om mee te werken. Nu zullen we veelal de database in scripts gaan raadplegen, dus die `mysql>` prompt willen we zoveel mogelijk vermijden.

MYSQL COMMAND LINE

Gelukkig kunnen we mysql ook op de command line uitvoeren, en zodoende ook in scripts openen.

MySQL Command Line

```
#> mysql -u root -proot unix_basics  
      -e "select * from orders" <enter>  
  
opties:  
-u:   username  
-p:   password (dat om de een of andere on  
        duidelijke reden meteen aan de "p" geplakt moet  
        worden).  
databasenaam  
-e    de query die we willen uitvoeren  
  
# Via input redirect  
#> vi select.sql <enter>  
<i>  
select * from orders;  
<esc><:wq!>  
  
#> mysql -u root -proot unix_basics  
      < select.sql <enter>  
  
# Je ziet nu een keurig tabelletje met de inhoud van de  
# order-tabel. Probeer de volgende varianten en kijk  
# wat er gebeurt  
  
#> mysql -u root -proot unix_basics --table  
      < select.sql <enter>  
  
#> mysql -u root -proot unix_basics --html  
      < select.sql <enter>
```

DE UITVOER BEWERKEN

Nu zien we ons leuke tabelletje wel keurig op het scherm, maar stel we willen met 'cut' bijvoorbeeld de tweede kolom laten zien:

MySQL Command Line

```
#> mysql -u root -proot unix_basics
< select.sql | cut -d" " -f2 <enter>
```

Hmm, dat werkt dus niet... (de ruimte tussen de velden is namelijk een tab ipv een spatie). Iets anders verzinnen dus. Gelukkig hebben we het '**tr**' (transform) commando:

MySQL Command Line

```
#> mysql -u root -proot unix_basics
< select.sql
| tr "\011" "-"
| cut -d"-" -f2 <enter>
```

'**tr**' vervangt hier het karakter met ASCII code 11 (vertical tab) met een '-' dat **cut** op zijn beurt weer gebruikt om de tabel in kolommen te verdelen.

Nu nog die lastige "naam" verwijderen:

MySQL Command Line

```
#> mysql -u root -proot unix_basics -N
< select.sql
| tr "\011" "-"
| cut -d"-" -f2 <enter>
```

Dat lijkt er al meer op...

MySQL Database dump

```
#> mysqldump -u root -proot  
    unix_basics <enter>  
  
# of redirect naar een bestand:  
#> mysqldump -u root -proot  
    unix_basics > dump.sql <enter>
```

GIT Commando's

```
# Username: EducomOpleidingen  
# Wachtwoord: Workshop  
  
#> cd $HOME/sampledata <enter>  
  
#> git clone  
https://EducomOpleidingen@bitbucket.org/EducomWorkshop/linux.git  
<enter>  
  
# Update van het project  
#> git pull origin master <enter>
```

EEN DATABASE DUMP MAKEN

Je kunt de volledige database exporteren via de command line. Standaard wordt dit in een (groot) sql-script opgeslagen. Dat is natuurlijk handig om een backup van je database te maken. De tool “**mysqldump**” is daar uitermate geschikt voor.

GIT PULL

‘**git**’ is een versie control tool, het gaat wat ver om alle in’s en out’s van git hier te beschrijven, maar we zullen het strikt functioneel houden binnen een beheer-context.

Eerst moeten we een zogenaamde repository “clonen”, dwz dat we een directory gaan aanmaken waarin we onze software gaan distribueren.

Als het gelukt is, heb je nu een directory “linux” in je sampledata directory. In deze directory staan nu alle files die je binnen het project kunt gebruiken. Vanaf nu kun je met:

git pull origin master
steeds de laatste versie ophalen.

Aantekeningen

3

DE SHELL SLIM(MER) GEBRUIKEN

Het zou natuurlijk een beetje raar zijn als je in elk script dat je gaat schrijven dat op de een of andere manier de database gebruikt, steeds opnieuw username en password op zou moeten nemen. Want stel nou het wachtwoord wordt gewijzigd, dan zijn in een klap al je scripts onbruikbaar en moet je ze stuk voor stuk bewerken om het wachtwoord te veranderen.

De shell biedt (standaard) hiervoor een oplossing:

Maak een config script

```
#> vi config.sh <enter>
<i>
export MYSQL_USER="root"
export MYSQL_PASS="root"
export MYSQL_DB="unix_basics"
<esc><:wq!>

#> chmod +x config.sh <enter>
```

Maak vervolgens een shell-scriptje waarin we een query gaan uitvoeren tegen de MySQL database:

Maak een database script

```
#> vi sqlscript.sh <enter>
<i>

# Config 'include'
. ./config.sh

mysql -u $MYSQL_USER -p$MYSQL_PASS $MYSQL_DB
      -N -B < select.sql

<esc><:wq!>

#> chmod +x sqlscript.sh <enter>
```

Voer nu je sqlscript uit. Je ziet dat de waarden uit **config.sh** meteen bruikbaar zijn (door het export commando) in je script.

EEN UITGEBREID SCRIPT

We gaan het geleerde nu toepassen in een script.
We maken een klein, bruikbaar script om een mysql database te dumpen, vervolgens via tar te zippen en als dit gelukt is via scp naar een remote machine te kopiëren.

● ● ● File system structuur

```
-- application /  
  -- cfg  
    -- config.sh  
  -- backup  
  -- log  
  -- tmp  
  -- backup.sh
```

● ● ● ./cfg/config.sh

```
#!/bin/bash  
#-----  
# SCRIPT: config.sh  
# PURPOSE: configuration for mysql db connection  
#           and remote_host config  
#-----  
  
export MYSQL_USER="root"  
export MYSQL_PASS="root"  
export REMOTE_HOST="workshop@educom.nu"
```



./bin/backup.sh

```
#!/bin/bash
#-----
# SCRIPT          backup.sh
# PURPOSE         Backup MySQL Database en kopieer
#                  tar-archive naar remote systeem
#
# USES           ../cfg/config.sh
#
# USING          backup.sh [databasename]
#-----


SCRIPT=`basename ${0} | cut -d"." -f1`
SCRIPTNAME=`basename ${0}`
DATABASE="${1}"
DATE=`date +%Y%m%d`
LOG="./log/${DATE}-$SCRIPT.log"
BACKUPFILE="backup/${DATE}-${DATABASE}.tar.gz"
TMPBACKUP="tmp/${DATE}-${DATABASE}.sql"

. ./cfg/config.sh


# Function writeLog
# Writes message to log-file
#
writeLog() {

    TIME=`date +%H:%m:%S`
    MSG="${1}"
    echo "$TIME - ${MSG}" >> $LOG

}

# Function backupDB
# Dumps database and compresses it
#
backupDB() {

    writeLog "Dumping database"
    /usr/bin/mysqldump --user=$MYSQL_USER
                    --password=$MYSQL_PASS
                    $DATABASE > $TMPBACKUP
    if [ -f $TMPBACKUP ];
    then
        writeLog "Creating archive ${BACKUPFILE}"
        tar -zcvf $BACKUPFILE $TMPBACKUP
    else
        writeLog "Problem dumping database"
    fi

}
```



./bin/backup.sh (vervolg)

```
# Function copyArchive
# Purpose copies backupfile to remote server
#
copyArchive() {

    writeLog "Copy Archive"
    scp $BACKUPFILE $REMOTE_HOST:/home/workshop
    if [ $? -ne 0 ];
    then
        writeLog "Error copying archive"
    else
        # Clean up
        #
        rm $TMPBACKUP
        rm $BACKUPFILE
    fi

}

writeLog "Start Backup"

## 
# Opgangen van parameter/argument
# Ons script heeft een parameter nodig dus eerst
# controleren of we wel een parameter binnen krijgen.
# Geven we geen parameter mee, dan exit met
# status 1 (fout)

if [ $# -eq 0 ];
then
    echo "usage $SCRIPTNAME [databasename]"
    exit 1
else
    backupDB
    copyArchive
fi

writeLog "End Backup"
```

