NTNU
Kunnskap for en bedre verden

Scripting

Helge Hafting, Institutt for informatikk og e-læring ved NTNU
Lærestoffet er utviklet for faget «TDAT3020 Sikkerhet i programvare og nettverk»

Resymé: I dette dokumentet ser vi på scripting i linux, og en del nyttige kommandoer.

Innhold

3 Scripting

3.1 Generelt om shellscript i linux

Jeg anbefaler å prøve ut de fleste testscriptene i denne øvingen. En lærer mye mer av å prøvekjøre kommandoer, enn av å bare lese om dem!

`http://www.freeos.com/guides/lsst/` har mer informasjon om scriptprogrammering.

3.1.1 Hva er shellscript, hva bruker vi dem til?

Et shellscript er på en gang en tekstfil og et program. Det inneholder stort sett samme slags kommandoer som vi gir på kommandolinja. I tillegg har vi noen konstruksjoner som sjelden brukes direkte, som løkker og if-tester. Shellscript kompileres ikke, de interpreteres (tolkes) hver gang de kjøres.

Kommandoene i shellscript kjøres stort sett én om gangen, i den rekkefølgen de står. Det fins noen unntak:

- Vi kan kjøre en kommando i bakgrunnen ved å sette tegnet «&» sist på linja. Dermed fortsetter scriptet uten å vente på at kommandoen blir ferdig. Praktisk hvis scriptet skal starte en webtjener eller lignende – tjenerprogrammer blir som kjent aldri «ferdige».
- Løkker og if-tester griper inn i rekkefølgen. Med en løkke kan vi kjøre de samme kommandoene flere ganger, og tester kan hoppe over kommandoer. Dette ser vi mer på lenger ut i leksjonen.

Shellscript er ment å være enkle å tilpasse, enklere enn programkode som må kompileres før den kan brukes. Derfor er shellscript praktiske for å automatisere administrative gjøremål. Når du som administrator finner ut at du gjør samme sort rutinearbeid om og om igjen – en daglig/ukentlig/månedlig oppgave, eller noe som må gjøres for hver eneste mappe i /home, så er tiden inne for å lage et lite script som gjør jobben lettere.

Linuxdistribusjoner har alltid en del ferdige script, bl.a. boot-script som du finner i */etc/init.d* eller */etc/rc.d*. Vil du tilpasse oppstarten kan du enkelt gjøre det ved å tilpasse scriptene – mye enklere enn å laste ned kildekode, kompilere og installere. En god

administrator er ikke nødvendigvis også C-programmerer, selv om det i blant kan være en fordel[1].

Ettersom script ikke kompileres, bør de heller ikke gjøre de tyngste jobbene. Å konvertere en fil fra postscript til jpeg er ikke en jobb for shellscript, men for et kompilert program. Å finne alle postscript-filene i et mappetre og sette i gang det kompilerte konverteringsprogrammet med passende parametre for hver av dem, er derimot en jobb som shellscript gjør glimrende.

De som har en del erfaring med windows, kjenner til *batchfiler,* som er windows-varianten av shellscript. Shellscript for linux er imidlertid kraftigere og har mange fler muligheter enn batchfilene i windows.

3.1.2 Ulike sorter shellscript

Mange shellscript er ment for det som for tiden er det vanligste shell-programmet: */bin/sh*. Men det behøver ikke være slik. Shellscript kan selv angi hvilket program som skal brukes for å interpretere det. Det gjøres ved at første linja starter med «`#! programnavn`». Et eksempel fra shutdown-scriptet *umountfs*:

```
#! /bin/sh
#
# umountfs    Unmount all
# local file systems.
umount -ttmpfs -a -r
echo "Ferdig!"
```

Som vi ser, ønsker dette scriptet å interpreteres av */bin/sh*[2]. Bash passer for mange jobber, men noen jobber gjøres bedre av andre scriptspråk. Script skrevet i f.eks. python eller perl starter med henholdsvis «`#! /usr/bin/python`» eller «`#! /usr/bin/perl`». Denne bruken av «`#! programnavn`» fungerer med alle slags programmer som kan lese en fil, ikke bare interpretere. I blant ser man denne mekanismen brukt i helt andre sammenhenger enn shellscript. F.eks. et dokument som selv starter opp tekstbehandleren når en forsøker å eksekvere det:

```
#! /usr/bin/emacs
Dette dokumentet redigeres ved å eksekvere det!
Prøv det ut, husk chmod oug+x så filen blir eksekverbar.
```

[1] De aller fleste linuxprogrammer er skrevet i C, det er også en betydelig minoritet som er skrevet i C++. Andre programmeringsspråk (python, perl, java, ...) brukes unntaksvis.

[2] */bin/sh* er som regel en link til */bin/bash* nå for tiden, da *bash* kan gjøre alt som gamle *sh* kunne, og mye mer. Navnet *sh* brukes av kompatibilitetsgrunner, mange programmer forventer at default-shellet på systemet heter *sh*, kort å skrive er det også.

Eksempelet forutsetter at emacs er installert. Hvis du har en annen favoritt-editor (f.eks. *vi*, *vim*, *pico* eller *nano*) tilpasser du den første linja. Prøv![3]

Linjer som starter med #-tegnet, er kommentarlinjer. (Bortsett fra den aller første linja, som vi nettopp har sett på.) Kommentarer er tekstlinjer som ikke utføres. De bare er der, til hjelp for den som leser og vedlikeholder scriptet. Mange script har noen kommentarer i starten, som forteller hva scriptet gjør. Hvis du legger en lang og komplisert kommando inn i et script, er det lurt å forklare den med en kommentar.

3.1.3 Hvor vi lagrer shellscript

Når en har laget et nyttig shellscript, hvor skal en lagre det? Det er flere standardmapper:

/usr/local/bin/ Dette er standardmappa for programmer som brukere kan ha nytte av. Her legger vi script som brukerne kan ha nytte av. Merk at «local» betyr programmer som er utviklet lokalt, de fulgte altså ikke med debian-distribusjonen.

/usr/local/sbin/ Standardmappe for programmer som brukes i systemarbeid, f.eks. script for å opprette nye brukere.

/etc/init.d/ Mappe for oppstartscript, som kjører når tjenermaskinen starter opp og/eller stopper. Husk å også lage lenker fra en passene */etc/rcX.d*-mappe.

3.2 En del kommandoer, verktøy og teknikker

3.2.1 find

Til tider har vi bruk for å gjøre samme jobb i alle mapper i et helt subtre. Når det f.eks. er lite plass igjen på */home*, kan vi ha bruk for å gå gjennom alle brukermappene og slette unødvendige filer. Hva som er «unødvendig» kan jo variere fra sted til sted. I dette eksempelet anser vi filer hvis navn ender på «~» for unødvendige. Mange editorer og andre programmer lager en sikkerhetskopi når de redigerer filer, og konvensjonen er at slike sikkerhetskopier får samme navn som originalen men med en «~» på slutten. Med tiden kan det bli mange slike filer på et linux-system som er aktivt brukt av mange brukere.

Programmet `find` går gjennom alle undermapper i et tre. I hver mappe leter den etter filer som passer med et eller flere kriteria, det kan være slike ting som filnavn, alder,

[3] Ikke glem *chmod* for å gjøre testscriptet ditt eksekverbart.

eierskap og mye annet. For hver fil som passer med kriteriene gjør `find` et eller annet. Det kan være noe så enkelt som å skrive filnavnet på skjermen, som er praktisk når vi leter etter filer. Det kan også være mer kompliserte ting, som å utføre en vilkårlig kommando, gjerne med filnavnet som en av parametrene. Hvis det vi skal gjøre med filen er riktig komplisert får vi `find` til å kjøre et script for hver fil som finnes.

Hvis vi ikke ber find om noe annet, skrives filnavnet ut:

```
find /home -name "*~"
```

Denne kommandoen søker gjennom treet */home*, og viser oss alle filnavn som slutter på «~». `find` er grei å ha for å finne filer en ikke helt vet hvor er. Et mer praktisk eksempel:

```
find . -name "*~" -exec rm {} \;
```

Denne kommandoen starter på stående mappe, og fjerner alle filer som slutter på «~» der og i alle undermapper. Parameteren `-exec` forteller at vi skal kjøre et program, i dette tilfellet sletteprogrammet `rm`. Parameteren `{}` byttes ut med filnavn som `find` finner. Konstruksjonen `\;` avslutter kommandoen. Kommandoen `man find` forteller mye mer om hvordan `find` kan brukes. Søk på navn er bare en av mange muligheter. Kommandoen kan også lete opp filer basert på eierskap, størrelse, tilgangstillatelser eller tidsstempel.

Hvis vi ønsker å sjekke flere betingelser samtidig, skriver vi opp alle sammen. Da vil `find` kreve at alle er oppfylt. Hvis en betingelse *ikke* skal være oppfylt, setter vi utropstegn foran den. Om det er nok at en av betingelsene holder, setter vi `-o`[4] mellom dem. For å finne alle filer som *ikke* er word-dokumenter:

```
find . ! \( -name "*.rtf" -o -name "*.doc" \)
```

For mer informasjon, bruk `man find`. Det er mange fler muligheter.

3.2.2 xargs

Verktøyet «`xargs`» leser inn en liste, og genererer en serie med kommandoer hvor elementer fra lista inngår. Vanligvis er den aktuelle lista en liste med filnavn, f.eks. en liste produsert av `find`.

Hvis eksemplene med `xargs` er vanskelige å forstå, kan du bruke parameteren `--interactive`. Da spør `xargs` om lov til å utføre hver enkelt kommando. Dermed får du se hver enkelt kommando før `xargs` utfører den. Dette kan også være praktisk hvis du gjør noe destruktivt, som f.eks:

[4] : or/eller

```
find . | xargs --interactive rm
```

Denne kommandoen fjerner alle filer, men spør om lov først.

Eksempler med *find* og *xargs*

Anta at filene *lek01.pdf, lek02.pdf, lek03.pdf, lek04.pdf, lek05.pdf* og *lek06.pdf* fins i den stående mappa.

```
find . | xargs echo Filnavn:
Filnavn: ./lek01.pdf ./lek02.pdf ./lek03.pdf ./lek04.pdf ./lek05.pdf ./lek06.pdf
```

Eksempelet var enkelt, men ikke så veldig nyttig. Vi vet jo at `find` kan skrive ut filnavnene uten hjelp av `xargs`. Vi kan imidlertid bytte ut **`echo`** med andre kommandoer. Vi har allerede sett at vi kan bruke `rm` for å fjerne filene. Nå kan forsåvidt `find` fjerne filer ved hjelp av parametereren **`-exec`**, men det er en viktig forskjell. `find` **`-exec`** vil alltid kjøre én kommando per fil, mens `xargs` vanligvis kjører én kommando med alle filnavnene som parameter. Det siste er mye mer effektivt, da vi kjører igang én prosess i stedet for mange.

Slik bruk av `xargs` kan gi veldig lange kommandoer, en `find`-kommando kan fort komme med titusenvis av filer. Ikke alle programmer takler å få så mange parametre. Derfor har `xargs` parameteren `-n` for å begrense antall argumenter. Når vi bruker den, vil `xargs` aldri gi fler argumenter enn vi vil, i stedet kjøres den aktuelle kommandoen flere ganger for å få med alle kommandoene i lista.

Ett filnavn per kommando:

```
find . | xargs -n 1 echo Fil:
Fil: lek01.pdf
Fil: lek02.pdf
Fil: lek03.pdf
Fil: lek04.psd
Fil: lek05.pdf
Fil: lek06.pdf
```

Dette var ikke noe bedre enn hva `find` klarer alene. Men vi kan be om to filnavn per kommando:

```
find . | xargs -n 2 echo Filer:
Filer: lek01.pdf lek02.pdf
Filer: lek03.pdf lek04.pdf
Filer: lek05.pdf lek06.pdf
```

Tre filnavn per kommando:

```
find . | xargs -n 3 echo Filer:
Filer: lek01.pdf lek02.pdf lek03.pdf
Filer: lek04.pdf lek05.pdf lek06.pdf
```

Programmer som tar en vilkårlig mengde filer som parametre, vil normalt klare mange hundre eller tusenvis av filer. Men en del sliter litt mer hvis det blir snakk om millioner. Det fins eksempler på at selv enkle kommandoer som `rm *` har mislyktes, fordi det var så altfor mange filer i mappa. Slike problemer løser vi med `xargs`.

Avansert bruk av parametre

Så langt har filnavnene blitt hektet på xargs-kommandoene automatisk. Ofte er det enkelt nok, men ikke alltid. La oss si at vi ønsker å forandre navnet på eksempelfilene, slik at *lek01.pdf* blir *lek01.pdf-gammel* og så videre. Det kan vi gjøre slik:

```
find . | xargs -I{} mv {} {}-gammel
```

Parameteren «`-I{}`» fører til at teksten «`{}`» byttes ut med filnavnene som `find` finner. Vi får altså utført denne serien med kommandoer:

```
mv lek01.pdf lek01.pdf-gammel
mv lek02.pdf lek02.pdf-gammel
mv lek03.pdf lek03.pdf-gammel
```

Vi trenger ikke bruke akkurat krøllparentereser, det er bare en vanlig måte å gjøre det på. Den følgende kommandoen gir akkurat samme resultat, og er lettere å lese for nybegynnere. Her er det ordet «filnavn» som byttes ut med *lek01.pdf, lek02.pdf*, osv.

```
find . | xargs -I filnavn mv filnavn filnavn-gammel
```

`xargs` kan også kjøre flere programmer samtidig for oss. For å vise frem pdf-filene våre tre og tre om gangen:

```
find . -name "*.pdf" | xargs -n 1 -P 3 xpdf
```

Her sier `-n 1` at hver kommando skal se på én fil, mens `-P 3` kjører opp til 3 kommandoer parallelt[5]. Parallell kjøring kan lett bli rotete med interaktive programmer som pdf-lesere, det er best egnet for programmer som ikke er interaktive. På en tjenermaskin med flere prosessorer, vil parallell kjøring få opp farten på prosessorintensive oppgaver.

For mer informasjon og flere eksempler, bruk `man xargs`

[5] NB! Dette fungerer ikke med den populære pdf-leseren *acroread*. Denne tror av en eller annen grunn at du bare er interessert i å kjøre én instans av programmet om gangen.

Sikkerhetsrisiko med filnavn som inneholder mellomrom

Så langt har vi brukt `find` og `xargs` på en måte som bruker mellomrom til å skille mellom filnavn. Nå går det imidlertid an å lage filnavn som inneholder mellomrom, og andre spesielle tegn (som f.eks. linjeskift). Spesielt windowsbrukere har det med å lage slike filer på filtjeneren vår, og de kan forvirre `xargs`.

Dette kan til og med bli en sikkerhetsrisiko. Tenk deg ei mappe som inneholder filene *a1, a2, a3, ...* og *b1, b2, b3, ...* For å slette alle a-filene, kan vi bruke en kommando som «`find . -name "a*"| xargs rm`». Men hva om noen lager en fil med navnet «a␣b1»? Hvis du prøver dette, vil du se at filen *b1* blir slettet, fordi `find` oppdager det spesielle filnavnet som begynner på «a», men `xargs` deler det opp i de to delene *a* og *b1* og får begge slettet. Eksempelet er litt søkt, men illustrerer hvordan en hacker kan lure `xargs` bare ved å konstruere passende filnavn.

Dette kan vi unngå ved å be programmene bruke ascii-0[6] som skilletegn. Dette tegnet forekommer aldri i filnavn, fordi filsystemet bruker det internt for å markere slutten på filnavnet. Sikker bruk av `find` og `xargs`:

```
find . -name "a*" -print0 | xargs -0 rm
```

Her sørger `-print0` for at `find` leverer filnavn adskilt med nulltegn, og `-0` sørger for at `xargs` oppfatter disse nulltegnene som skilletegn. På denne måten fungerer kommandoene også når filnavn inneholder mellomrom, linjeskift eller andre rariteter. Når en bruker `find` og `xargs` i script, bør det gjøres på denne måten.

For å se akkurat hva som skjer, prøv disse kommandoene i en ellers tom mappe:

```
touch a1 a2 b "a b"
find . -name "a*" -print0 | xargs -0 -n1 --interactive rm
```

Kommandoen `touch`[7] lager filer med navnene «*a1*», «*a2*», «*b*» og «*a␣b*». Sjekk gjerne med `ls -l` etterpå. Den neste kommandoen fjerner alle *a*-filene, en om gangen. Trykk *y* og ENTER for å bekrefte hver sletting. For å se hvor galt det kan gå, prøv på nytt uten `-print0` og `-0`. Altså:

```
touch a1 a2 b "a b"
find . -name "a*" | xargs -n1 --interactive rm
```

Hele tre ting går galt. Filen «*a␣b*» fjernes ikke, selv om den begynner på «a». Dernest får vi en feilmelding om at det ikke gikk å fjerne en fil kalt *«a»*. Det fins ingen fil kalt

[6] ascii 0 er det første tegnet i ascii-tabellen. Det har ikke noe med sifferet null (0) å gjøre, sifferet null har ascii-verdien 48.

[7] brukes egentlig for å oppdatere tidsstempelet på filer. Men hvis filene ikke fins, vil lage dem. Det er den enkleste måten å lage en tom fil.

«a», dette er et resultat av at «*a*␣*b*» ble delt opp. Det verste er at filen *«b»* forsvinner, det var jo ikke det vi mente!

3.2.3 if-tester

Grammatikken for en if-setning, hentet fra «`info bash`»:

```
if betingelsesliste; then liste; [ elif liste; then liste; ] ... [ else liste; ] fi
```

I denne sammenhengen er «`liste`» en eller flere kommandoer, som kan returnere 0 eller en annen verdi. Til forskjell fra andre programmeringsspråk regnes betingelsen i **`if`** (eller **`elif`**[8]) som oppfylt hvis returverdien er 0. Dette fordi betingelsen som evalueres vanligvis er resultatet fra et kjørbart program. Det er tradisjon for at programmer returnerer 0 når alt går bra, og en feilkode ellers.

Hvis betingelsen er oppfylt, kjøres kommandolisten etter **`then`**, hvis ikke kjøre kommandolisten etter **`else`**, om man da har brukt **`else`**. Eksempler, som også kan prøves direkte på kommandolinja:

```
$ if ls ; then echo ok! ; else echo feil! ; fi
lek09.lyx       lek09.pdf
ok!
$
$ if asdf ; then echo ok! ; else echo feil! ; fi
bash: asdf: command not found
feil
$
```

Som vi ser lyktes **`else`**, mens kommandoen `asdf` mislyktes fordi den ikke fins. En kommando kan mislykkes på mange andre vis også, f.eks:

- Kommandoen bruker en ikke-eksisterende fil: **`cat`** `asdfghjklø`
- Kommandoen får en ukjent parameter: `ls --ugyldig`
- Ytre årsaker; som full disk, brukeren mangler nødvendig tilgang, timeout på nettverk, feil i programmets konfigurasjonsfil, og mye mer.

`if`-testen avsluttes med **`fi`**, som er **`if`** baklengs.

[8] er en kortform for

3.2.4 case-tester

Når en trenger en test med mange alternativer er **case** et ryddigere alternativ enn en lang kjede med **if elif elif elif fi**. Et eksempel, `http://www.iie.hist.no/fag/lsd/nedlastbart/12/fargedemo`:

```
echo -e "\033[m"
case $1 in
  gul)
    echo -e "\033[33;1m"
    ;;
  hvit)
    echo -e "\033[37;1m"
    ;;
  lilla)
    echo -e "\033[35;1m"
    ;;
  turkis)
    echo -e "\033[36;1m"
    ;;
esac
echo $1 tekst
```

Prøv å kjøre `fargedemo` med de ulike fargenavnene. Her brukes **echo**-kommandoen på en spesiell måte, for å skifte farge på teksten. En **case**-test brukes for å velge farge.

Testen avsluttes med ordet «esac», som er «case» baklengs.

3.2.5 Testkommandoen «[»

I shellscript ser vi ofte tester av denne typen: **if** `[ -e /etc/konfigfil]` **; then**... I dette tilfellet er det en test for å se om */etc/konfigfil* eksisterer. Det kan se ut som om klammene er en del av scriptspråket, men det er de ikke! Det er rett og slett et program som heter `/usr/bin/[`, hvis oppgave er å utføre en del enkle tester. Mange oppstart-script tester hvorvidt programfiler, konfigurasjonsfiler eller datafiler fins, før de kjører i gang et større program. Dermed unngår vi unødvendig venting og feilmeldinger hvis de nødvendige filene ikke er der. Prøv `man [` for mer informasjon om dette testprogrammet. Programmet «`[`» kan teste mye annet også, som f.eks. hvilken av to filer som er eldst, om et navn refererer til en fil, mappe, lenke eller annen spesialfil, hvem som eier en fil, eller hvorvidt filen er lesbar eller skrivbar. Programmet kan også teste og sammenligne strenger og tallverdier, dette brukes ofte til å sammenligne parametre og miljøvariabler[9].

[9] Eng.: *environment variable*

3.2.6 Miljøvariabler

Shellscript refererer til innholdet i miljøvariabler ved å sette et dollartegn foran variabelnavnet. Eksempel[10]:

```
$ echo Variabelen PATH har verdien $PATH
```

Slike variabler opprettes automatisk ved behov. Et enkelt eksempel:

```
$ VARIABEL="Testing testing"
$ echo Innholdet i VARIABEL er nå $VARIABEL
```

Et mer komplisert eksempel:

```
$ TESTVAR="Programmet 'ls' ligger på "`which ls`
$ echo $TESTVAR
```

Hvis vi vil at variabelen skal være tilgjengelig for andre script vi kjører, eksporterer vi den med kommandoen *export*, slik:

```
export TESTVAR
```

Kommandoen *export* kan kombineres med tilordningen, eksempel:

```
export PATH=/bin:/sbin:/usr/bin:/usr/sbin
```

Et script som f.eks. tilpasser *PATH* for oss, må bruke `export PATH` for at andre script vi kaller opp skal ha nytte av tilpasningen.

Det er en sterk tradisjon for at miljøvariabler alltid skrives med store bokstaver, men små fungerer også. Jeg anbefaler imidlertid å følge tradisjonen. Da er det tydeligere hva som er miljøvariabler og hva som ikke er det.

3.2.7 which

I avsnittet om miljøvariabler brukte jeg `` `which ls` ``. Kommandoen `which program` leter gjennom mappene i *PATH* og forteller oss i hvor programmet *program* er å finne. Eksempel:

```
$ which ls
/bin/ls
```

[10] Du må selv kjøre eksempelkommandoen for å se hva som skjer.

Kommandoen er nyttigere enn en skulle tro, da den oppklarer mysteriet med «hvilken programfil brukes». La oss si at du installerer en ny og bedre versjon av `ls`, men `ls` virker bare på gamlemåten. `` `which ls` `` forteller oss hvilket `ls`-program som brukes, kanskje det finnes et annet `ls`-program på maskinen din, i en mappe som ligger tidligere i *PATH*.

3.2.8 ‘Grav aksent‘

Grav aksenter[11] brukes for å fange inn output fra en kommando slik at teksten kan brukes i tester eller tilordnes til en miljøvariabel. Dette så vi et eksempel på i `` `which ls` ``-eksempelet, hvor veien til *ls* ble lagt inn i en miljøvariabel. Noen andre eksempler du kan prøve:

```
MASKINNAVN=`cat /etc/hostname`
FILER=`ls`
echo Brukernavnet ditt er `whoami`
```

Hvordan skrive inn ‘grav aksenter‘

På et norsk tastatur får du grav aksenten ved å trykke «skift» og «\» samtidig. Hvis tastaturet ditt er satt opp med dødtaster, må du trykke mellomrom etterpå også.

`Grav aksent` må ikke forveksles med de vanligere ' apostrofer '. Apostrofer har en helt annen (og mye enklere) virkning i script — de brukes til å avgrense tekststrenger, på omtrent samme måte som "doble apostrofer ".

Detaljert forklaring på grav aksenter

Vi ser på denne kommandoen:

```
echo Navnet ditt er `whoami`
```

Når en slik kommando skal kjøres, utføres først det som står inni grav aksentene. I dette tilfellet kommandoen `whoami`. Denne kommandoen skriver vanligvis ut brukernavnet. Når kommandoen står inni grav aksenter, vil i stedet denne utskriften fanges opp, og erstatte delen med grav aksenter. I mitt tilfelle er brukernavnet *helgehaf,* så kommandoen transformeres til

```
echo Navnet ditt er helgehaf
```

[11] Eng.: *back ticks,* Lat: *gravis accent*

Når dette utføres, får jeg skrevet «Navnet ditt er helgehaf» på skjermen.

Nok et detaljert eksempel:

```
export NAVN=`whoami`
echo Navn: $NAVN
```

På grunn av grav aksentene kjøres programmet `whoami`, og det hele transformeres til:

```
export NAVN=helgehaf
echo Navn: $NAVN
```

Her ender jeg altså opp med beskjeden *«Navn: helgehaf»* Prøv dette selv, spesielt hvis dette med grav aksenter ikke er helt opplagt.

3.2.9 Doble og enkle apostrofer

Som nevnt brukes disse for å avgrense strenger. Dermed kan vi lage strenger som inneholder blanktegn. Prøv f.eks. disse kommandoene:

```
mkdir "mappe med langt navn"
mkdir 'ennå en lang mappe'
```

Her fungerer enkle og doble apostrofer likt. Det er imidlertid en forskjell, miljøvariabler ekspanderes inne i doble apostrofer, men ikke i de enkle. Eksempel:

```
echo "test: $PATH"
test: /usr/local/bin:/usr/bin:/bin:/usr/games:/home/helgehaf/bin

echo 'test: $PATH'
test: $PATH
```

3.2.10 Parametre

Første parameter til et shellscript finnes i variabelen `$1`, neste parameter i variabelen `$2`, og så videre. Har man bruk for alle parametrene på en gang, bruker man `$*`. (F.eks. hvis scriptet kjører en kommando med alle parametrene.) Parameteren `$0` er navnet på shellscriptet. Det er *ikke* alltid det samme, det kan være anderledes hvis scriptet kalles via en lenke med et annet navn. Scriptet kan dermed vite hvilket navn det ble kalt under. Et eksempelscript kalt *testscript:*

```
#! /bin/sh
echo \$1 = $1
echo \$2 = $2
echo \$* = $*
echo \$0 = $0
```

Hvis vi kjører det, går det slik:

```
./testscript første andre tredje fjerde femte
$1 = første
$2 = andre
$* = første andre tredje fjerde femte
$0 = ./testscript
```

Parametre brukes ofte for å fortelle script hva de skal jobbe med. F.eks. en eller flere filer.

3.2.11 *basename* og *dirname*

Disse to deler opp stier og filnavn, og forklares best med noen eksempler:

```
$ basename /etc/samba/smb.conf
smb.conf
$ dirname /etc/samba/smb.conf
/etc/samba
$ basename `dirname /etc/samba/smb.conf`
samba
```

Disse kommandoene er ofte kjekke å ha i shellscript. Vær oppmerksom på at kommandoene ikke egentlig bryr seg om hva som er fil- eller mappenavn, eller om filene fins i det hele tatt. De bare deler opp tekststrenger med skråstreken, «/», som skilletegn. Kommandoen `basename` gir oss den siste delen, som typisk er filnavnet. Kommandoen `dirname` gir oss alt utenom den siste delen, typisk mappa filen ligger i.

3.2.12 for-løkke

Shellscript har løkkekommandoen «**for**», som kan gjenta et sett kommandoer for hvert element i en mengde. Ofte er «mengden» et sett med filer.

Advarsel til de som har vært borti «for-løkker» i vanlige progragrammeringsspråk som Java og C, – dette er noe helt annet.

Det enkleste er å demonstrere med noen eksempler:

```
for ting in første andre tredje ; do echo "Dette er $ting test" ; done
Dette er første test
Dette er andre test
Dette er tredje test
```

Her skrev jeg alt på en linje, det er lettest når vi tester på kommandolinja. I en scriptfil ville jeg gjort det slik:

```
for ting in første andre tredje
do
    echo "Dette er $ting test"
done
```

I scriptfilen brukes linjeskift i stedet for semikolon. Resultatet blir det samme. **for**-løkka kjøres en gang for hvert element i lista, og hver gang er det et nytt element i løkkevariabelen `$ting`.

Vi kan putte mer enn én kommando i ei slik løkke:

```
for noe in a b c prøve
do
    echo "Test: 3x $noe"
    echo $noe$noe$noe
done
```

Resultatet blir noe slikt som dette:

```
Test: 3x a
aaa
Test: 3x b
bbb
Test: 3x c
ccc
Test: 3x prøve
prøveprøveprøve
```

Så langt har vi bare brukt faste lister. Men vi kan også bruke filnavn, slik som dette:

```
for fil in /etc/*.conf
do
    cp $fil /backup/
    cp $fil /minnepenn/backup/
exit
```

Dette scriptet kopierer alle filer i */etc* som slutter på *conf*. Filene kopieres to ganger, så de er godt sikret. Dette virker fordi «*/etc/*.conf*» ekspanderes. Prøv f.eks. «*echo /etc/*.conf*» for å se hvordan ekspansjon av *-tegnet virker.

3.2.13 wget

Forkortelse for «web get», og er en enkel kommando som henter filer fra Internettet. Praktisk både i script og på kommandolinja. Eksempel, hente fargedemo-scriptet mitt:

```
wget http://www.iie.hist.no/fag/lsd/nedlastbart/12/fargedemo
```

Hvis alt går greit, får du fila *fargedemo* lastet ned, sjekk med `ls` etterpå. Hvorfor ta bryet med å starte opp en nettleser og klikke seg rundt, bare for å hente en fil?

3.3 Håndtere tekst

Script kan få bruk for å plukke ut tekst fra filer, eller fra output som andre programmer lager. Dette for å ta avgjørelser eller lage rapporter til brukeren som kjører scriptet.

3.3.1 Jobbe med linjer: grep, uniq, head, tail, wc

Kommandoen `grep` kan mye, det vanligste er å plukke ut linje(r) som matcher et søkeord. Maskinen har en oversikt over minne i */proc/meminfo*. Kommandoen `cat /proc/meminfo` vil vise all denne informasjonen. Det er ganske mye. Hvis vi bare vil vite hvor mye minne maskinen har, uten detaljene om hva det brukes til, kan vi gjøre slik:

```
$ cat /proc/meminfo | grep MemTotal
MemTotal:        8084092 kB
```

Tilsvarende kan vi finne ut hva slags prosessor vi har:

```
$ cat /proc/cpuinfo | grep "model name"
model name : Intel(R) Core(TM) i7-4600U CPU @ 2.10GHz
model name : Intel(R) Core(TM) i7-4600U CPU @ 2.10GHz
model name : Intel(R) Core(TM) i7-4600U CPU @ 2.10GHz
model name : Intel(R) Core(TM) i7-4600U CPU @ 2.10GHz
```

Legg merke til anførselstegnene, som er nødvendige når vi søker etter tekst med mellomrom i. Denne maskinen har fire kjerner, så informasjonen kom fire ganger. Det kan vi unngå med filteret `uniq`:

```
$ cat /proc/cpuinfo | grep "model name" | uniq
model name : Intel(R) Core(TM) i7-4600U CPU @ 2.10GHz
```

Dette filteret plukker vekk linjer som er like. I dette tilfellet kunne vi også bedt om første eller siste linje fra output:

```
$ cat /proc/cpuinfo | grep "model name" | head -1
model name : Intel(R) Core(TM) i7-4600U CPU @ 2.10GHz
$ cat /proc/cpuinfo | grep "model name" | tail -1
model name : Intel(R) Core(TM) i7-4600U CPU @ 2.10GHz
```

Kommandoene `head` og `tail` plukker ut de første eller siste linjene fra det de ser på. Parameteren sier hvor mange linjer vi vil ha.

Vi så at det var fire prosessorer. Men å telle slikt kan bli slitsomt hvis det er noe særlig mer enn fire. Da bruker vi heller `wc`, som står for «word count». Eksempel:

```
$ cat /proc/cpuinfo | grep processor | wc
4       12      56
```

Her fikk vi vite at det var fire linjer, og altså fire prosessorkjerner. Videre var det 12 ord og 56 byte som ble undersøkt av `wc`. Et script som bare trenger antall linjer, kan bruke `wc -l` for å bare vise linjetallet.

3.3.2 Plukke ut informasjon med cut

Da vi sjekket minnet i maskinen, fikk vi antallet i kB. Det kan regnes om til MB eller GB, men da må vi kunne plukke ut tallet uten teksten som er rundt. For eksempel:

```
cat /proc/meminfo|grep MemTotal|cut -c 10-24
        8084092
```

Her fikk vi antallet, fordi jeg ba om tegnene fra posisjon 10 til posisjon 25. Ledeteksten «MemTotal:» er de 9 første tegnene, og tallet slutter i posisjon 24. For å dessuten få regnet om til gigabyte, kan man installere pakken «apcalc» og gjøre slik:

```
calc `grep MemTotal /proc/meminfo|cut -c 10-24` /1024/1024
7.709590911865234375
```

På min maskin kan altså 7,7GB (av 8GB) brukes. I eksempelet bruker jeg `-c` for å angi et intervall med tegn. `cut` kan evt. plukke ut felter i stedet, med parameteren `-f`. Felter er vanligvis skilt fra hverandre med tab-tegn, men parameteren `-d ':'` kan brukes for å angi andre skilletegn; i dette tilfellet, et kolon.

3.4 Logonscript for linux

3.4.1 Hva er et logonscript

Logonscript er script som kjøres når brukeren logger inn. Når shellet *bash* starter kjører det en del script automatisk, avhengig av hvordan det startet opp. Når bash kjører som interaktivt login-shell, kjører det først */etc/profile* hvis denne filen finnes. Deretter kjøres *~/.bash_profile*, *~/.bash_login* og *~/.profile*. Gunnen til at det er tre slike filer er sannsynligvis bakoverkompatibilitet med andre shell, det holder å ha én slik fil.

Shellet *bash* kan også kjøres interaktivt uten å være loginshell, hvis vi gir kommandoen *bash* på kommandolinja, hvis vi kjører en terminalemulator i det grafiske brukergrensesnittet, eller hvis et program vi arbeider med kjører *bash* for oss for at vi skal kunne gi noen linux-kommandoer. I slike tilfeller kjører *bash* scriptene */etc/bash.bashrc* og *~/.bashrc*.

Filene i */etc* er der for at administratoren skal kunne legge inn felles oppstartsting der. Filene på hjemmemappen er der så den enkelte bruker kan gjøre personlige tilpasninger.

Inndelingen i login-shell og andre shell er for å unngå unødvendig (og til tider skadelig) dobbeltarbeid. Noen ting, som f.eks. å stille inn en passende PATH trenger bare gjøres en gang, ved innlogging. Shell som starter senere «arver» denne innstillingen. Administratoren kan sette en standard-PATH i */etc/profile*, og brukeren kan gjøre egne tilpasninger som f.eks. *PATH=$PATH:~/mine-programmer:/usr/games* i *~/.bash_profile.*

Legg merke til at bash enten kjører *~/.bashrc* eller *~/.bash*_profile, men aldri begge. Et login-shell har sine egne spesielle behov, men det har gjerne bruk for samme oppsett som andre shell også. Derfor er det vanlig å la *~/.bash_profile* kjøre en *source .bashrc* til slutt. Dermed slipper en å ha felles kode i de to scriptene.

Installasjoner som bare kjører grafisk innlogging bruker ikke nødvendigvis de loginspesifikke scriptene, fordi systemet starter en windowmanager i stedet for et loginshell. Brukerne får tilsvarende funksjonalitet enten ved å konfigurere sin windowmanager, eller ved å tilpasse filen *~/.xsession*. Vær oppmerksom på at grafiske brukere har glede av *~/.bashrc* når de kjører terminalvinduer som *xterm, gterm*, o.l.

3.4.2 Vanlige oppgaver for linux-logonscript

Felles-scriptene brukes gjerne til sette opp standard `PATH`, standard språkvalg og lignende for alle. De private scriptene kan brukes til så mangt. Mange legger inn *aliaser* for mye brukte kommandoer og opsjoner. Hvis en vil ha en kommandoprompt som er litt mer interessant enn bare «$» er logonscriptet et passende sted. Hvis en bruker farger, bør en teste hvorvidt terminalen/terminalemulatoren støtter dette. Et eksempel fra min .bashrc:

```
alias cd..="cd .."
alias rd=rmdir
alias md=mkdir
case $TERM in
  linux | rxvt | xterm)
    LSOPT="--color -F"
  *)
    LSOPT="-F"
```

```
esac
alias ls='ls '$LSOPT
```

Fordi jeg ofte glemmer mellomrommet i kommandoen **cd ..**, har jeg laget en alias for dette tilfellet. Jeg har også kortversjoner av `rmdir` og `mkdir`. Til slutt en alias for `ls`, som tar med parameteren `-F`. Dette gjør det lettere å se hva som er filer og hva som er mapper. **case**-testen bruker jeg fordi noen terminaler ikke støtter farger.

Noen velger også å la logonscriptet starte opp programmer som de ofte bruker, som f.eks. nettleseren. Dette er mest anvendelig når man benytter grafisk brukergrensesnitt.

Folk som har jobbet en del med windows-kommandolinja definerer i blant aliaser som lar dem bruke vanlige windowskommandoer slik:

```
alias dir='ls -l'
alias type='cat'
alias copy='cp'
alias move='mv'
```

Slik kan de skrive de lange kommandoene de er vant til.

3.5 Standard unixverktøy

3.5.1 ifconfig – nettverkskonfigurasjon

Noen distribusjoner bruker den nyere kommandoen «ip» og har ikke «ifconfig». Se i så fall seksjon 3.5.3 på side 22

Kommandoordet er en forkortelse for «interface config», og da menes «interface» mot nettverk. Denne kommandoen brukes til meget, det vanligste er slike ting som å stille inn ip-adresse og nettmaske for nettverkskort. Det første nettkortet i en maskin heter vanligvis *eth0*, det neste *eth1*, osv. En del spesielle nettverk har navn som ikke går på ethX. Alle har *lo*[12] som brukes når prosesser på maskinen snakker med hverandre via 127.0.0.1. Varianter av *ppp* brukes av point-to-point forbindelser, *tun* brukes for ip-tunneler. Noen eksempler på *ifconfig*:

[12] local loopback

Eksempel	Forklaring
`ifconfig eth0 down`	Slår av nettkortet eth0.
`ifconfig eth1 up`	Aktiverer kortet eth1
`ifconfig eth0 12.34.56.78`	Gir kortet eth0 en adresse
`ifconfig eth1 1.2.3.4 netmask 255.254.0.0`	Stiller inn adr. og maske
`ifconfig eth0`	Gi informasjon om eth0
`ifconfig`	Informasjon om alle lokale nettverk

Konfigurasjonsfiler for nettverksoppsett

Vær oppmerksom på at nettkonfigurasjon med `ifconfig` ikke er permanent. Hvis vi f.eks. forandrer ip-adressen blir den altså ikke lagret noe sted, og endringen vår går tapt ved neste omstart av maskinen. Nettverksproblemer kan løses kjapt og uten omstart ved hjelp av `ifconfig`, men permanente forandringer krever at vi også forandrer på konfigurasjonsfiler. Når linux starter opp, kjøres en mengde startscript, blant annet */etc/init.d/networking*[13]. Disse scriptene leser konfigurasjonsfiler og bruker `ifconfig` for å sette ip-adresser og lignende.

Konfigurasjonsfilen for ip-adresser heter */etc/network/interfaces* på debian. Den har et enkelt format for statiske ip-adresser. Dokumentasjon finnes med kommandoen `man interfaces`. Et kort eksempel på hvordan denne fila kan se ut:

```
auto lo eth0
iface lo inet loopback
iface eth0 inet static
        address 158.38.51.68
        netmask 255.255.254.0
        gateway 158.38.50.1
iface eth1 inet dhcp
```

Ordet «*auto*» sørger for at */etc/init.d/networking* starter opp de aktuelle interfacene. I dette tilfellet *lo* og *eth0*. Som vi ser er oppsettet for *lo* enkelt, det er av typen *inet,* altså vanlig ipv4, og konfigureres med metoden *loopback.* Denne metoden stiller inn standard loopbackadresse 127.0.01.

Interfacet *eth0* er mer komplisert. Det bruker en statisk ip-adresse, og derfor er adressen, nettmasken og ruteren oppgitt. Til slutt har vi *eth1* som bruker dhcp. Her trengs ingen ekstra opplysninger, Alt vil bli hentet fra dhcp-tjeneren. Siden vi ikke har *auto eth1* noe sted, vil ikke *eth1* starte opp automatisk. Det må vi gjøre manuelt, f.eks. med `ifup eth1`.

[13] Det kan være små forskjeller mellom distribusjonene her, alle har startscript men navnene kan variere. Noen bruker *rc.d* i stedet for *init.d.*

Andre distribusjoner kan være temmelig forskjellige. Noen bruker «network-manager» programmer, som enten har egne konfigfiler, eller grafiske brukergrensesnitt.

Jeg vil advare mot å bruke grafiske brukergrensesnitt. De er OK for enkle tilfeller, som å få en brukermaskin «på nett». De er som regel tungvindte hvis man skal sette opp en maskin med fire nettverkskort og ruting mellom dem. Da er det enklere med script og konfigfiler!

3.5.2 ifup og ifdown – spesielt for debian

For å «skru på» nettverksinterfacet *eth0* kan vi bruke `ifup eth0`. For å skru det av igjen, `ifdown eth0` Disse programmene bruker filen */etc/networking/interfaces.* Hvis du har gjort forandringer i *interfaces*, bruker du *ifdown eth0* fulgt av *ifup eth0* for å aktivere det nye oppsettet.

3.5.3 ip – nyere nettverkskonfigurasjon

Dette er et nyere alternativ til `ifconfig` og `route`. Kjekt å kunne, noen distribusjoner har droppet `ifconfig` nå. Legg merke til at et interface kan ha flere ip-adresser

Eksempler, ip-adresser	**Forklaring**
`ip link`	Hvilke interface fins, og deres status
`ip addr`	Hvilke adresser/masker fins (alle interface)
`ip addr show eno1`	Adresse(r) og maske(r) for interface eno1. NB! vær oppmerksom på at et nettkort kan ha mer enn én adresse.
`ip addr add 10.1.1.1/23 dev eno1`	Gi interfacet eno1 en ekstra adresse og nettmaske. (/23 tilsvarer 255.255.254.0)
`ip addr del 10.1.1.1/23 dev eno1`	Fjerner en ip-adresse fra eno1

Ruter angir hvilke rutere pakker skal sendes til, for å nå maskiner utenfor lokalnettet. Vi kan f.eks. ha en regel om at pakker til 11.12.13.0/24 skal rutes via ruteren 10.1.1.2. Det er også vanlig å ha en default-rute, som brukes for «alle andre ukjente nettverk, som ikke har en eksplisitt rute». Hvis man bruker Internettet, må man praktisk talt alltid ha en default-rute.

Det går an å ha flere ruter til samme sted, via ulike rutere. I slike fall bruker linux den ruteren som har lavest «metric». Metric er et tall vi kan sette selv, det sier noe om «hvor

langt unna» den aktuelle ruteren er. En vanlig anvendelse er å sette lav metric på kablet nett, og høy metric på trådløst. Da rutes pakker gjennom kablet nett når det fins, og trådløst ellers. En annen anvendelse er når man har flere rutere for redundans. Da brukes metric til å velge den beste ruteren.

En linuxmaskin kan godt være på flere nett samtidig – både kablet og trådløst, eller flere kablede/trådløse nettverk. Enkle oppsett med bare ett nettverk trenger bare en defaultrute.

Eksempler, ruter	Forklaring
ip route list	Hvilke rute(r) fins?
ip route add default via 10.1.1.5	Legg til en default-rute
ip route add default via 10.1.1.5 metric 300	Legg til defaultrute, med «metric»
ip route del default via 10.1.1.5	Slett en default-rute
ip route add 11.12.13.0/24 via 10.1.1.2	Legg til rute for å nå 11.12.13.0/24
ip route del 11.12.13.0/24 via 10.1.1.2	Fjerne en eksplisitt rute

3.5.4 iwconfig – for trådløse nett

Kommandoen `iwconfig` brukes i tillegg til `ifconfig` for å stille inn tilleggsparametre for trådløse nett. Normal kjører man `iwconfig` først for å stille inn trådløse parametre, deretter setter man interfacet i drift med `ifconfig` eller `ifup`.

I de trådløse eksemplene mine er *eth1* et trådløst kort. Vær oppmerksom på at noen trådløse drivere bruker helt andre navn, som f.eks. *ra01, wlan0* og lignende. Programmet *iwconfig* fins i pakka *wireless-tools*.

Trådløst eksempel

```
iwconfig eth1 essid "Mitt nett" mode Managed key s:passw
```

I dette eksempelet er ssid *Mitt nett*. Her brukes krypteringsnøkkelen *passw.* I åpne nett oppgir vi ingen nøkkel, eller vi bruker *key off*. Når nøkkelen oppgis som en tekststreng, setter vi «*s:*» foran. En nøkkel kan godt bestå av uleselige byte-verdier også, i så fall oppgis nøkkelen heksadesimalt, som f.eks. «key 02ab4f66be» eller «key 02ab-4f66-be»

Parameteren *mode* har følgende muligheter:

mode	resultat
Managed	Oppkobling mot aksesspunkt
Ad-Hoc	Direkte forbindelse med annen pc, uten aksesspunkt
Master	PCen blir selv et aksesspunkt
Secondary	PCen blir reserve for annet aksesspunkt
Repeater	PCen videresender trafikk mellom andre trådløse noder
Monitor	Brukes for overvåking/avlytting

Parametrene *freq* og *channel* kan brukes for å velge bestemte frekvenser/kanaler. Det trengs for Ad-Hoc og Master, i Managed mode er det like greit å la driveren selv velge kanalen til det aksesspunktet som har sterkest signal.

I tillegg til disse parametrene har `iwconfig` en del parametre for å optimalisere ytelsen i nettverk med mye støy. En kan f.eks. få kortet til å droppe forbindelser under en viss signalstyrke, eller sørge for oppdeling av store pakker for bedre å tåle kortvarige men kraftige forstyrrelser.

Trådløst oppsett i debian

Som før nevnt, legger vi det permanente nettverksoppsettet inn i */etc/network/interfaces.* Om vi installerer pakken *wireless-tools* kan vi også legge trådløse innstillinger inn i denne filen.

Trådløs klient i debian (wep)

```
auto eth1
iface eth1 inet dhcp
    wireless-essid   Hjemmenett
    wireless-mode    Managed
    wireless-key     s:passw
    wireless-channel auto
```

Trådløst aksesspunkt implementert i debian (wep-basert)

```
auto eth1
iface eth1 inet static
    address          10.0.0.1
    netmask          255.255.0.0
    wireless-essid   Hjemmenett
    wireless-key     7061-7373-77
    wireless-channel 6
    wireless-mode    Master
```

I tillegg vil det være naturlig å kjøre en DHCP-tjener når man implementerer et trådløst aksesspunkt slik. Dette for at klientene skal kunne få tildelt ip-adresse automatisk. DHCP-tjener er imidlertid ikke tema i denne leksjonen. Vær oppmerksom på at ikke alle nettkort har drivere for å fungere som aksesspunkt.

Tips for trådløse bærbare

Dette kurset handler mest om tjenermaskiner, men for de som kjører linux på en bærbar anbefaler jeg pakkene *ifplugd* og *wpasupplicant.* Den siste oppdager når du kommer

innen rekkevidde av kjente trådløse nett, og setter automatisk opp trådløskortet så snart det er mulig. Pakken *ifplugd* oppdager når du setter inn pluggen for kablede nett, og når *wpasupplicant* tar opp et trådløst kort. Pakken tar da opp kortet med *ifup,* og dermed er maskinen på nett uten at du trenger å gjøre noe mer selv.

Trådløse nett med wpa, wpa2 og wpa-enterprise

Wpa bruker mer avansert kryptering, og bytter krypteringsnøkler systematisk for at det ikke skal gå an for hackere å gjette nøkkelen. Dette er for komplisert for nettkort-driveren å håndtere selv, derfor bruker man tilleggsprogrammer som *wpa_supplicant*. Dette programmet tar seg av detaljer som å oppdage trådløse nett, logge på, og deretter håndtere krypteringsnøkler så lenge du bruker nettet. Oppsettet for *wpa_supplicant* kan være komplisert, de fleste bruker en eller annen form for «network manager» for å slippe dette. For de interesserte tar jeg med noen eksempler. Først filen */etc/wpa_supplicant/wpa_supplicant.conf*, med eksempler for eduroam (wpa-enterprise), vanlig wpa, og wep:

```
network={
  #Oppsett for eduroam på NTNU (og resten av verden)
  #Eduroam bruker wpa-enterprise
  ssid="eduroam"
  scan_ssid=1
  key_mgmt=WPA-EAP
  eap=PEAP
  ca_cert="/etc/ssl/certs/AddTrust_External_Root.pem"
  identity="brukernavn@ntnu.no"
  password="passordet"
  phase1="peaplabel=0"
  phase2="auth=MSCHAPV2"
}

network={
  #Eksempel på vanlig wpa/wpa2, slik det gjerne er på hjemmenett
  ssid="nettverksnavn"
  scan_ssid=1
  key_mgmt=WPA-PSK
  psk="wpa-passordet"
}

network={
  #Eksempel på gammeldags wep-basert nett
  ssid="nettnavn"
  scan_ssid=1
  key_mgmt=NONE
  wep_key0="passw"
}
```

Åpne nett (uten passord) blir som det siste eksempelet, bare uten linja med `wep_key0`.

For å bruke wpa_supplicant kjøres programmet i gang slik, gitt at trådløskortet heter wlan0:

```
wpa_supplicant -s -B -i wlan0 -W -D wext -c /etc/wpa_supplicant/wpa_supplicant.conf
```

Pakkene som trengs er *wpasupplicant* og *ca-certificates*. Den siste inneholder et digitalt sertifikat som er nødvendig for å få kontakt med eduroam.

3.5.5 iw

Noen distribusjoner bruker det nyere programmet `iw` i stedet for `iwconfig`. I praksis er det sjelden man bruker iw/wconfig selv, man har gjerne nettverksmanagere (eller i det minste wpa_supplicant) til dette. Med `iw` kan man gjøre alt `iwconfig` kan, og mer.

Legg merke til at iw kan operere både på «fysiske forbindelser» (phy0, phy1, . . .) og på «trådløse nettkort» (wlan0, wlp2s0, . . .) Et trådløst nettkort er typisk koblet mot en «fysisk forbindelse»

Det er ikke så ofte man bruker `iw` direkte, men kommandoen kan være nyttige i script eller for å finne ut av trådløse problemer.

Eksempler	**Forklaring**
`iw help`	Oversikt, hva du kan gjøre med iw (mye)
`iw list`	Oversikt over alt trådløst utstyr på maskinen, og hva det kan gjøre (mye)
`iw phy0 info`	Egenskaper for en bestemt fysisk forbindelse (phy0) (mye)
`iw dev wlp2s0 info`	Info om et bestemt trådløst kort (wlp2s0)
`iw dev wlp2s0 link`	Sjekk om nettkortet har forbindelse med aksesspunkt
`iw dev wlps20 scan`	Hvilke aksesspunkter fins, med alle detaljer. (Signalstyrke, støttede protokoller, og mye annen teknisk informasjon.) Filtrer med `| grep SSID` for å bare se tilgjengelige nettverk.
`iw event`	Skriv ut trådløse hendelser etterhvert som de skjer. Typisk informasjon om at maskinen kobler seg til og fra ulike aksesspunkter når man går rundt i bygget. Nyttig for å finne ut av problemer med roaming.

3.5.6 route – ruter til andre nettverk

Kommandoen `route` brukes for å håndtere ruter. Pakker ment for maskiner som ikke er på samme nett må sendes via nærmeste ruter, og `route` lar oss fortelle linux om ruteren.

Mange maskiner har bare tilgang på én ruter. Denne kan da med fordel settes opp som default-ruter. Hvis den har adresse 158.38.56.22 gjøres det slik:

```
route add default gw 158.38.56.22
```

Noen ganger har vi tilgang på flere rutere som kjenner til hver sine nett. Ruteren som ruter videre til flest nett (typisk hele internettet) får default-ruten, de andre får eksplisitte ruter som i dette eksempelet:

```
route add –net 192.56.76.0 gw 158.38.56.23
```

Her angir vi en annen ruter for pakker som skal til 192.56.76.X. Slike pakker vil bli sendt til ruteren 158.38.56.23. En kan forsåvidt bruke DNS-navn i stedet for ip-adresser, *hvis* DNS er tilgjengelig når en gir kommandoen. Det er ikke nødvendigvis tilfelle, f.eks. hvis en holder på å sette opp en rute til nettet hvor DNS-tjeneren befinner seg. En kan få rede på hvilke ruter som gjelder i øyeblikket ved å gi kommandoen *route* uten parametre. Ruter kan slettes ved behov. For å få eksempler på sletting er det bare å bytte ut ordet *add* med *del* i eksemplene over. Flere detaljer om `route` skaffes med kommandoen `man route`.

Nett-endringer gjort med `route` er ikke permanente, akkurat som med `ifconfig`. Permanente endringer gjøres derfor i maskinens konfigurasjonsfiler. Default-ruten spesifiseres gjerne i */etc/network/interfaces*. Se eksempelet i avsnitt 3.5.1 på side 21, default-ruten spesifiseres her med nøkkelordet *gateway*.

En maskin trenger ikke nødvendigvis selv bruke `route`-kommandoen når den selv ruter mellom flere nett. Kommandoen trengs bare for å opplyse maskinen om nettverk som bare nåes via *andre* rutere. Ulike nett som er koblet direkte på maskinen kjenner den allerede til via interface-oppsettet.

3.5.7 sudo – kjør programmer under en annen brukeridentitet

Installer pakken *sudo*, eller kildekode fra `ftp://ftp.sudo.ws/pub/sudo`. Med debian er det frivillig å bruke *sudo,* men noen distribusjoner nærmest legger opp til at du skal bruke *sudo* i stedet for *su.*

Hva sudo gjør

Formålet med *sudo* er å la bestemte brukere kunne kjøre bestemte programmer under en annen brukers identitet, uten å kjenne den andres passord. Dette kan brukes for å la vanlige brukere gjøre visse oppgaver som *root* eller som andre privilegerte brukere (*daemon, games, man, mail*, . . .) Brukeren må i stedet oppgi sitt eget passord, mest for å sjekke at ikke noen har funnet en terminal som er forlatt i innlogget tilstand.

Hvorfor

Hvorfor ikke bare gi bort rotpassordet, så slipper en å sette opp sudo? Poenget er at det å kjenne til rotpassordet i en stor organisasjon, er et stort ansvar. Problemer med at mange kjenner rotpassordet er ikke bare at noen kan fristes til å misbruke det. Det er like mye faren for tabber gjort av folk som bruker *root* i utide.

Hvis en f.eks. har en forretningskritisk database kan det være lurt at mange har mulighet til å kjøre den i gang hvis den kræsjer. Det kan være nødvendig å gjøre dette som *root*, men de som arbeider med databasen trenger sannsynligvis ikke å kunne opprette brukere, omkonfigurere nettverket, slette systemfiler eller andre farlige operasjoner som *root* kan gjøre.

Hvordan

Hvem som kan gjøre hva er definert i konfigurasjonsfilen */etc/sudoers*. Den redigeres helst med kommandoen *visudo* som sjekker at det ikke bli noen feil. Filen er godt dokumentert med *man sudoers*, for debian-brukere finnes også en eksempelfil i */usr/share/doc/sudo/examples/sample.sudoers*. Filen finnes også her: `http://www.iie.hist.no/fag/lsd/nedlastbart/12/sample.sudoers`.

Et enklere eksempel:

```
User_Alias SKRIVERADM = per,kari
# Cmnd alias specification
# Kommandoer for å avbryte utskrifter,
# og starte/stoppe spooleren
Cmnd_Alias PRINTING = /usr/bin/lprm, /etc/init.d/cupsys
# User privilege specification
root    ALL=(ALL) ALL
#Skriverfolk kan bruke skriverkommandoer uten passord
SKRIVERADM   ALL=NOPASSWD:PRINTING
#Nils kan drepe andres prosesser
nils         ALL=/bin/kill
```

Her er brukerne per og kari skriveradministratorer, de kan bruke kommandoer som

```
sudo /usr/bin/lprm 47
sudo /etc/init.d/cupsys restart
```

Den første fjerner en skriverjobb, selv om det er en annen sin utskrift. Det går fordi *sudo* er satt opp til å gi *root*-privilegier for denne kommandoen. Den andre kommandoen restarter utskriftsystemet. Brukerne per og kari vil ikke bli spurt om passord.

Brukerne nils får lov til å bruke kommandoen *kill* til å drepe andres prosesser. Slikt kan det f.eks. være behov for når noen kjører spill og lignende på tider hvor maskinkapasiteten

trengs til viktigere jobber. Brukeren nils vil bli spurt om passord når han kjører *sudo kill 1367*, men det er bare sitt eget passord han trenger å oppgi. Nils trenger altså ikke å kjenne *root*-passordet. Poenget med å måtte oppgi sitt eget passord er at folk som tilfeldigvis kommer over Nils sin maskin ikke skal kunne gi sudo-kommandoer uten videre.

3.5.8 cron – prosesser som kjøres til faste tider

En del programmer har vi bruk for å kjøre til faste tider. Et eksempel er *updatedb* som oppdaterer den databasen *locate* bruker. Dette skjer normalt en gang i døgnet, vanligvis om natten fordi dette er en disk- og minnekrevende jobb som ikke bør gå i veien for interaktiv bruk av maskinen.

Verktøyet *cron* lar oss kjøre prosesser til faste tider. Det kan være alt fra hvert minutt eller time, til hver dag, en gang i uka, måneden eller året. Mer kompliserte tidsangivelser er også mulig, som f.eks. hver time mellom $08^{\underline{00}}$ og $16^{\underline{00}}$, eller hver torsdag kl. $18^{\underline{45}}$.

Hva som skal kjøres, og når, spesifiseres i filen */etc/crontab*. Filen leses en gang i minuttet av *cron*, som sjekker hvorvidt noen av kommandoene skal kjøres i det aktuelle minuttet. Filen består av linjer med sju felter. De fem første angir tidspunkt for kjøringen. Det sjette feltet spesifiserer hvilken bruker som skal kjøre programmet, f.eks. *root*. Det siste feltet inneholder kommandoen som skal kjøres og eventuelle parametre. Vær oppmerksom på at */etc/crontab* opererer med sin egen path, ofte er det enkleste å angi hele veien frem til det aktuelle programmet.

De fem tidsangivelsene i */etc/crontab* er følgende: minutt, time, dag i måned, måned og til slutt dag i uke. Man fyller vanligvis ikke ut alle feltene. Hvis et program skal kjøres daglig, setter man en passende time og et passende minutt. Det er lurt å sette ulik minuttangivelse for ulike programmer, dermed unngår vi belastningstopper som ellers kunne oppstå ved at mange programmer starter samtidig. De andre feltene fylles ut med *, som alltid passer. Det går an å angi intervaller også. «Hver time fra og med 8 til og med 10, og 16 til og med 22» angis som «0 8-10,16-22 * * *». Hvis vi hadde satt «*» i minuttfeltet, ville programmet kjørt i gang *hvert* minutt i de aktuelle timene. I tillegg til tidspunkter og intervaller, kan vi også spesifisere oppdelinger. «*/5 * * * *» betyr «hvert femte minutt». Slike oppdelinger kan kombineres med intervaller også, så «15 8-16/2 * * 1-5» betyr at programmet skal kjøres annenhver time i arbeidstiden (arbeidstid fra 8 til 16, mandag til fredag). Minuttangivelsen 15 betyr at programmet kjører kvart over, altså $8^{\underline{15}}, 10^{\underline{15}}, \ldots, 16^{\underline{15}}$.

Vær oppmerksom på at intervaller er *fra og med, til og med*. En tidsangivelse som «*/20 10-12 * * *» dekker altså følgende tidspunkter: $10^{\underline{00}}$,$10^{\underline{20}}$, $10^{\underline{40}}$, $11^{\underline{00}}$, $11^{\underline{20}}$,$11^{\underline{40}}$,$12^{\underline{00}}$, $12^{\underline{20}}$ og $12^{\underline{40}}$. Mange nybegynnere mistolker denne crontab-linja som «hvert 20. minutt

mellom 10 og 12», og får seg en overraskelse når *cron* aktiveres $12^{\underline{20}}$ og $12^{\underline{40}}$ også. Korrekt tolking er «hvert 20. minutt, så lenge *timeverdien* er i intervallet 10–12, endepunkter inkludert.

Personlig *crontab*

Konfigurasjonsfilen */etc/crontab* er bare tilgjengelig for *root*. Brukerne kan derfor ha sine egne *crontab*-filer, som de vedlikeholder selv med kommandoen *crontab*. Denne kommandoen starter opp en editor hvor brukeren kan redigere sin *crontab*, og sjekker for skrivefeil ved lagring. Formatet er det samme som for */etc/crontab* bortsett fra at man ikke spesifiserer hvilken bruker som skal kjøre programmet. En personlig *crontab* kjører alltid programmene under brukerens egen identitet.

Hvis administrator ikke ønsker at alle skal kunne bruke personlige *crontab*-filer, kan tilgangen til dette reguleres med */etc/cron.allow* og */etc/cron.deny*. Brukere som står i den første får lov til å ha sin egen crontab, brukere som står i den andre får det ikke. Det er ikke nødvendig å ha begge filene, hvis en f.eks. bare har */etc/cron.deny* får alle lov som ikke står på listen.

Noen eksempler på */etc/crontab*

Her ser vi noen eksempler på daglige, månedlige og ukentlige jobber. Legg merke til hvordan minutter og timer varieres for å unngå at flere jobber starter samtidig. Uke- og månedsjobber kan jo komme til å kjøre samme dag. Diskrydding gjøres dessuten før backup, så blir det mindre å kopiere.

```
13 1 * * * root /usr/local/sbin/daglig-diskrydding
27 4 * * * backup /usr/local/sbin/nattbackup
45 2 * 1 * backup /usr/local/sbin/mnd-backup
0  8 * * 5 regnskap /usr/bin/auto-purring
```

3.6 Oppsummering

Vi har sett på scripting, med en del eksempler.