Introduktion till Linux och små nätverk

Kommandon och skalprogram

Många använder web- och grafiska gränssnitt när de arbetar med datorer. Vilket fungerar utmärkt när man skall göra enstaka saker eller när man jobbar med eget data, som att skriva rapporter eller att läsa sin datorpost. Men det är inte alltid det optimala verktyget för att administrera en eller flera datorer i ett nätverk.

Varför påstår jag det? Jo, för att installera en eller flera datorer så vill man automatisera installationen så att om man behöver installera om datorn, exempelvis pga att disken gått sönder eller att det har blivit datorintrång[1], så vill man att det skall gå snabbt och felfritt. Att göra en installation manuellt ökar risken för att gör ett misstag, som att glömma ett steg eller att skriva fel. Därför vill man automatiserat de saker som man gör ofta, eller väldigt sällan.

Att automatisera innebär att man exempelvis kan skriva ner de kommandon som man vill utföra i ett så kallat skalprogram (även *kommandotolk*, *shell programming*). När programmet senare utförs, så kommer alltid samma kommandon att utföras. Det innebär att om programmet gjort rätt en gång, så kommer det troligen göra rätt i fortsättningen med.

Dessa skalprogram är textfiler med de kommandon som skall utföras. Så därför är det viktigt för en administratör att kunna hantera kommandoraden. Eftersom Apple OS X är baserat på Unix BSD, så har även de tillgång till samma textbaserade kommandoskal som i Linux. Standard i OS X är `zsh`. Även Microsoft har upptäckt att kommandoskal är bra och har ett textbaserat kommandoskal som heter Powershell samt ett äldre som heter Cmd.exe.

I Microsoft Windows 10 och 11 så kan man numera även installera olika Linux-distributioner, som exempelvis Debian eller Ubuntu, via deras App-sida. Då får man tillgång till samma kommandoskal som från Debian, men man kan enkelt köra grafiska program[2] i den miljön. Den är mer tänkt för programmerare som vill ha samma miljö att programmera i som de servrar som kör programmen senare. En annan begränsning är att varje användare på Windows 10-maskinen har (minst) en egen installation av Debian-miljön, eftersom det är en Microsoft Windows 10-app.

I Unix, och Linux, så finns det i huvudsak två familjer av kommandoskal som ser lite olika ut, nämligen Bourne Shell (`sh`) och C Shell (`csh`). Till dessa så finns det några utökade skal som framför allt underlättar för en människa att hantera skalen interaktivt. Vanligaste av dessa är Bourne Again Shell (`bash`) och TENEX C Shell (T C Shell, `tcsh`), även om det finns andra, som Z Shell (`zsh`). Vi kommer att koncentrera oss på `bash` i den här kursen. Dels för att `bash` följer en standard som heter POSIX, men även när man kan `bash`, så är det rätt enkelt att använda de andra.

Kommandoskalet Bash

När man loggar in på en Linux-maskin med exempelvis SSH[3], eller via någon av textkonsolerna[4], eller genom att i den grafiska miljön starta ett kommadoskal (`xterm`, `gnome-terminal` eller

1 Enda säkra sättet att hantera ett datorintrång är att säkerhetskopiera inställningen och sedan göra en ominstallation av datorn. Detta eftersom vi inte kan vara säkra på vad som gjorts av den som gjort intrånget.
2 Dvs så vida du inte kör ett program i Microsoft Windows 10 som hantera X11. MS Windows 11 har stöd för grafik.
3 Exempelvis med programmet `Putty` i MS Windows eller `ssh` i OSX, Linux och MS Windows 10 och 11.
4 I Linux från den grafiska konsolen kan du normalt komma åt med någon av `Ctrl-Alt-F1` till `Ctrl-Alt-F6`.

liknande) så kommer du att ha ett skal framför dig.

Det första du ser är en prompt, vilket har ett `$`-tecken. Det visar att skalet är klart att ta emot ett kommando. Notera att prompten hos dig troligen är längre, men avslutas med '`$`'-tecknet. Prompten skall *inte* skrivas in när det är i början av en rad i exemplen nedan. Men prova gärna dem när du läser igenom texten, för att se hur de fungerar hos dig.

En kommandorad består först av ett kommando och sedan ett antal argument. Vissa argument kallas växel (*switch* på engelska) och förändrar beteendet på programmet, dessa inleds med en eller två ’`-`’-tecken och ett eller flera bokstäver/siffror. Dessa växlar kan ofta kombineras för att få en kombinerad effekt. Alla argument som inte inleds med ett ’`-`’-tecken är vanliga argument, som exempelvis filnamn. Man kan även lägga till andra tecken som förändrar hur programmet körs av skalprogrammet, men det är instruktioner till kommandoskalet och inte programmet.

Om vi provar kommandot `ls`, så kommer det att skriva ut en lista på de filer och kataloger som finns i vår hemmakatalog, där vi kan spara våra privata saker. Varje användare har en hemmakatalog som de får läsa och skriva i, men ingen annan såvida inte användaren godkänner det. Den hemmakatalogen kan man ange med tilde (`~/`) eller genom att skriva `$HOME`.

Så, om vi skriver kommandot `ls,` så kan det se ut så här (prova gärna själv för att se hur de fungerar), där vi först skapar dessa filer, utan innehåll med kommandot `touch`:

```
$ touch a.txt b.txt
$ ls
a.txt  b.txt
$
```

Om man vill ha mera information om filerna, så kan man lägga till växeln `-l,` som betyder *long listing*. Så kan det se ut så här istället:

```
$ ls -l
totalt 0
-rw-rw-r-- 1 anders anders 0 jan 15 13:49 a.txt
-rw-rw-r-- 1 anders anders 0 jan 15 13:49 b.txt
$
```

Här ser vi en rad per fil. Varje rad visar först typ av fil och sedan rättigheterna (`-rw-rw-r--`). Därefter vem som äger filen (`anders`) och vilken grupp av användare som filen tillhör (gruppen `anders` i detta fall). Därefter storleken (`0` tecken), när ändrad (den 15 januari), samt slutligen namnet på filen (`a.txt`).

Men alla filer listas inte av `ls`, utan bara filer vars namn **inte** börjar med en punkt (`.`). För att få med dem i listningen så kan man använda växeln `-a` (all), och då får man detta resultat, se nedan. Detta görs eftersom filnamn och kataloger som börjar med en punkt är konfigurationsfiler, som man normalt inte vill se.

```
$ ls -a
.  ..  a.txt  b.txt
$
```

Notera att det nu skrivs ut två nya element i raden, nämligen punkt (`.`) samt punkt punkt (`..`). Dessa är ett alias på katalogen själv (`.`) samt namnet på katalogen ovanför katalogen (`..`), den så kallade förälderkatalogen. Dessa två kataloger finns alltid i alla kataloger.

Om man nu vill ha reda på allt om alla dessa filer och kataloger, så kan man kombinera de två växlarna med antingen som `ls -a -l` eller `ls -al`, och får då det kombinerade resultatet:

```
$ ls -la
totalt 8
drwxrwxr-x   2 anders anders 4096 jan 15 13:49 .
drwxr-xr-x 108 anders anders 4096 jan 15 13:49 ..
-rw-rw-r--   1 anders anders    0 jan 15 13:49 a.txt
-rw-rw-r--   1 anders anders    0 jan 15 13:49 b.txt
$
```

Som ni ser på rättigheterna så betyder ett minus (`-`) på första positionen att det är en vanlig fil och ett `d` att det är en katalog (*directory*). Sedan kommer rättigheterna för filen/katalogen i tre grupper om tre tecken.

Vi kommer att titta mer på det senare, men kort så betyder `r` – läsrättighet, `w` – skrivrättighet, `x` – exekveringsrättighet samt `-` – att motsvarande rättighet på den positionen inte existerar.

Första gruppen av `rwx` gäller för användaren (den kolumn som i exemplet är den första `anders`), den andra gruppen `r-x` gäller för gruppen (den kolumn som den andra `anders` står i) samt tredje och sista gruppen `r-x` för alla som inte är användaren eller tillhör gruppen.

Så filen `a.txt` ovan så har användaren `anders` läs och skrivrättighet, gruppen `anders` läs och skrivrättighet medans alla övriga bara har läsrättighet. Med kommandot `id` så ser man vilken användarnamn man har samt vilka grupper man tillhör.

Normalt i Linux så tillhör alla användare en grupp som har samma namn som användaren själv. Men en användare kan även tillhöra flera andra grupper.

Manualsidor med information

De allra flesta kommandon har en manualsida tillgänglig från Linux kommandorad. Dessa är kompakta och kan verka kryptiska innan man blir van. Men de är väldigt användbara när man har glömt vilka växlar som kommandon har, vilka argument som man behöver ange samt i vilken ordning de skall komma.

Så exempelvis för att se vilka växlar som kommandot `ls(1)` har, så använder man sig av kommandot `man(1)`. Så manualsidan för `ls` kan läsas om man skriver in kommandot `man ls` (eller `man 1 ls` där 1:an är vilken manualsektion man vill titta i. Där sektion ett innehåller alla vanliga kommandon, vilket kan skrivas så här i text: `ls(1)`).

Bra att vet är att `man(1)` även kan användas för att söka efter kommandon utifrån nyckelord i dess beskrivning. Så med kommandot `man -k change` kommer man se en lista med alla manualsidor som har ”`change`” i beskrivningen av respektive kommando. För att se hur man använder kommandot för att se manualsidor, se manualsidan för `man(1)`, vilket kan skriva så här: `man man`

Filkommandon i skal

Vanliga kommandon som man använder sig av för att hantera filer i Linux är uppräknade i tabell 1 längre ned.

Ta reda på med `man(1)`-kommandot (och referenserna nedan) vilka växlar de har och hur de fungerar. Glöm inte bort att ni kan prova hur de fungerar i ett kommandoskal.

Skal och omgivningsvariabler

För att programmera i skalprogram så behöver man mellanlagra resultat, och det kan man göra i skalvariabler. Dessa skrivs oftast med små bokstäver, som exempelvis `fil` och `destination`. Dessa kan skapas närhelst man vill och kommer bara att vara tillgängliga i skalet som man kör i.

Det finns dock andra variabler som varje program som körs har en egen kopia av, som kallas omgivningsvariabler (*environment variables*). Dessa skrivs alltid med stora bokstäver, som `HOME`, `USER` och `PATH`.

Dessa variabler påverkar hur program och operativsystemet fungerar, och dessa sätts i program som startas från kommandoskalet. Oftast behöver man inte bry sig om dem, men några är bra att veta om. Alla omgivningsvariabler kan skrivas ut med kommandot `printenv(1)`, om man vill se vilka som finns definierade i sitt kommandoskal.

Tabell 1. Vanliga förekommande kommandon med beskrivning

Kommando	Beskrivning
`ls`	Lista innehåll i en eller flera kataloger
`pwd`	Skriver ut aktuell katalog
`cd`	Byter till angiven katalog, om inget argument så byt till hemmakatalogen
`mkdir, rmdir`	Skapar respektive tar bort angivna kataloger
`touch`	Skapar en tom datafil
`rm`	Raderar angivna filer (kan även radera filer och kataloger)
`cp`	Kopierar en eller flera filer till den sist angivna filen/katalogen
`mv`	Flyttar en eller flera filer till den sist angivna filen/katalogen
`ln`	Skapar ett alias, sk länk. Det kan vara en hård eller mjuk länk
`echo, printf`	Skriver ut argumenten som står efter. `printf` kan formatera utskrifter bättre än `echo`
`./skalpgm.sh`	Startar det skalprogrammet `skalpgm.sh` som finns i aktuell katalog
`chmod`	Ändrar rättigheterna på angivna filer/kataloger
`chown, chgrp`	Ändrar ägare respektive grupp av angivna filer/kataloger
`cat`	Läser från fil (eller `stdin` om ej fil angiven) och skriver ut på `stdout`
`more, less`	Som `cat`, men gör det en sida i taget. Bläddra med mellanslag avsluta med `Q`
`man`	Ger en manualsida för det kommando som ges som argument
`help`	Ger hjälp om Bash-kommandot

De viktigaste omgivningsvariabler är `HOME` och `PATH`.

Variabeln `HOME` anger vilken katalog som är användarens hemmakatalog. Den har samma värde som `~` som nämndes ovan. Så vill man se vilket värde den har, så kan man använda kommandot `echo` så här: `echo "$HOME"`.

Dollar-tecknet (`$`) framför variabelnamnet talar om att man vill ha variabelns värde. Citat-tecknen (`"`) runt anger att man skall inte tolka mellanslag i variabelns värde som avskiljare mellan argument eller kommandon. Vill man skriva ut `$`-tecknet, så använder man `'`-tecken istället, exempelvis: `echo '$HOME'`. Prova i kommandoskal för att se skillnaden.

Variabeln PATH talar om vilka kataloger som Linux skall leta efter kommandon att starta, där varje katalog skiljs med ett kolon-tecken[5] (:). Så när du skriver ls(1), så kontrolleras om det är ett i skalet inbyggt kommando eller om det är definierat som ett alias i skalet[6]. Om det inte är något av dessa, så kommer Linux att leta genom de i PATH angivna katalogerna efter en körbar fil med det namnet, dvs ett kommando. Ett kommando är en fil som har exekveringsrättigheterna (x) satta för att markera att det kan exekveras.

Det finns många fler variabler, men dessa två är vanliga. För att se vilka omgivningsvariabler som de olika kommandona använder, så kan man titta i manualsidan för respektive kommando.

Man skapar och sätter dessa omgivningsvariabler i bash(1) med följande kommandorad: export *NAMN=värde*. Notera att det inte får finns mellanslag runt =-tecknet.

Dessa nya värden finns bara tillgängligt för kommandon som startar efter detta. Det ändrar alltså inte omgivningsvariabelns värden för kommandon som startas tidigare.

In- och utmatning

Varje program som kör i Linux har tre filer öppna, nämligen stdin, stdout och stderr.

Dessa används för att läsa in data (stdin), skriva ut resultat från kommandon (stdout) samt skriva ut status eller felmeddelanden från programmet (stderr). Dessa kommer man åt via fil-nummer 0, 1 respektive 2.

Normalt är stdin ansluten/bunden till tangentbordet (specialfilen /dev/tty) samt stdout och stderr till skärmen (även de specialfilen /dev/tty). Så när ett program läser in data, så läser det från stdin och när det är klart så skriver det ut resultat/data på stdout.

Finessen med detta är att man kan ganska enkelt testa program på kommandoraden för att sedan kombinera flera program för att göra något i exempelvis ett skalprogram.

Exempelvis så kan vi prova programmet wc(1), som räknar rader, ord och tecken som läses från stdin och sedan skriver ut resultatet på stdout. Så för att räkna antalet rader som vi matar in via tangentbordet så är det bara att starta programmet.

För att avsluta inmatningen från tangentbordet, så tryck ned Ctrl-d[7]. Då kommer programmet nämligen tro att det kommit till slutet av filen (tangentbordet), sluta läsa in mer data samt skriva ut resultatet av programmet (nedan betyder C-d att man matat in Ctrl-d).

```
$ wc
gfd sdkl gf
fasanvm afam agre
C-d
      2       6      30
```

Så vi har matat in två rader, 6 ord och 30 tecken. Om man nu vill att wc(1) istället skall läsa från filen /etc/passwd, hur gör man då? Jo, då talar man om för kommandoskalet att det skall starta programmet wc(1), men att stdin skall sättas till /etc/passwd och inte tangentbordet (filen /dev/tty).

5 I MS Windows så finns även PATH, men där skiljs varje katalog åt med semikolon (;) istället, eftersom kolon används för att ange diskenhet.

6 Ett alias är i skalet definierat namn på kommandon. Kan användas för att göra det enklare att köra vanliga kommandon.

7 Det genererar nämligen ett EOF, vilket betyder End Of File. Dvs slut på filen, och inget mer skall läsa från den.

Det görs enklast så här, där < talar om att `stdin` skall sättas till filen (en pil som pekar på kommandot):

```
$ wc < /etc/passwd
  49   80 2513
```

Dvs filen hade 49 rader, men ert resultat kan vara ett annat, beroende på vad er fil innehåller.

Men om vi inte vill att resultatet skall skrivas ut på skärmen (filen `/dev/tty`), utan i filen `resultat.txt`, hur gör man då?

Jo, man gör så här istället, där > talar om att `stdout` skall sättas till filen `resultat.txt` (en pil som pekar från programmet):

```
$ wc > resultat.txt
fdas sadfjkal sasf
fadjl jfda öhdfa
C-d
$ cat resultat.txt
      2       6      37
$ cat < resultat.txt
      2       6      37
```

Kommandot `cat(1)` läser filerna som anges efteråt och resultatet skrivs ut på skärmen. Om man inte skriver något filnamn, så kommer `cat(1)` att läsa från `stdin` istället, dvs normalt tangentbordet. Så det därför ger det tredje kommandot samma resultat, även om det inte är samma sak för programmet `cat(1)`.

Vill man istället ha utskriften en sida i taget, så använder man med fördel programmet `more(1)` eller `less(1)` istället för `cat(1)`.

Naturligtvis så kan man kombinera både < och > så att kommandoskalet ställer om både `stdin` och `stdout` för programmet.

```
$ wc < /etc/passwd > resultat.txt
$ cat resultat.txt
  49   80 2513 wc < /etc/passwd > resultat.txt
```

Om man vill ha utmatning från ett program som inmatning till ett annat program, så kan man ju starta programmen efter varandra där man mellanlagrar data i en temporär fil. Men det är slöseri med diskutrymme och tid, så därför kan man tala om för kommandoskalet att `stdout` från ett program skall vara `stdin` på nästa, och det gör man med pipe-symbolen (`|`).

Så om vi vill att utmatningen från programmet `cat(1)` skall sorteras, så kan vi använda oss av programmet `sort(1)` på det här viset.

```
$ cat < /etc/passwd | sort
anders:x:1000:1000:Anders Jackson,,,,:/home/anders:/bin/bash
```
[… data som tagits bort]
```
www-data:x:33:33:www-data:/var/www:/bin/sh
```

Vill jag nu räkna dessa sorterade rader för att se att inga rader försvinner, då är det bara att ställa om utmatningen från `sort(1)` till inmatningen på `wc(1)`.

Experimentera gärna med detta samtidigt som du läser.

Så prova gärna detta:

```
$ cat < /etc/passwd | sort | wc
     49      80    2513
```

Utskriften kommer från kommandot `wc`(1), som skriver till `stdout`. Så då vet vi att alla raderna gått igenom eftersom antal rader inte ändrats och inga nya lagts till.

Om man nu vill mata in en konstant text i ett skalprogram, så kan man styra om den texten till `stdin` på programmet. Då använder man med en speciell version av `<`, som kallas *HERE document*.

Det kan skrivas på detta sättet.

```
$ wc <<EOF
> ett två tre
> fyra
> fem
> EOF
 3  5 22
```

Dvs allt från raden efter `<<EOF` till nästa förekomst av en rad som bara innehåller texten `EOF` kommer att användas som `stdin` till kommandot `wc(1)`. Notera att texten `EOF` kan bytas ut till annan text om man vill, exempelvis `FILSLUT`, prova gärna.

Om man vill ställa om `stderr` så kan göra omställningen så här: `2&> felfil.txt`. Dvs fil nummer `2`, `stderr`, skrivs ut på filen `felfil.txt`. Som de övriga ovan, så kan det kombineras så att resultatet skrivs till en fil och status/felmeddelanden skrivs till en annan fil.

Vill lägga till omstyrningen från ett program till slutet av en fil istället för att skriva över innehållet, så gör man så här: `>> utfil.txt`

Dvs två `>>` lägger till i slutet av filen, medans `>` skriver över.

Skriva skalprogram

Hur skriver man nu ett skalprogram då?

Jo, man skapar en vanlig textfil som börjar med en speciell första rad. Efter den raden är det bara att skriva de kommandon som skall utföras, exempelvis i filen `skal.sh`. Sedan kan man starta programmet med `bash ./skal.sh` . Alternativt görs filen exekverbar med hjälp av kommandot `chmod(1)` och sedan starta det som vilket annat program som helst med `./skal.sh`.

Så här kan man då göra för att skapa ett program som räknar rader i den fil som man anger som argument. Notera att `#` är kommentar och `$1` är första argumentet. Prova gärna vad `$0` , `$1` och `$*` ger (samt kontrollera mot manualen till `bash(1)`). Notera att ni inte skall skriva in `>` i början av raderna.

```
 $ cat > skal.sh <<EOF
> #! /bin/bash
> # Ett testprogram
> echo Test "$1" "$2"
> wc -l < "\$1"
> echo Test "\\\$1"
> exit
```

```
> # EOF
> EOF
$ ls -l skal.sh
-rw-rw-r-- 1 anders anders 84 jan 16 17:36 skal.sh
$ chmod +x skal.sh
$ ls -l skal.sh
-rwxrwxr-x 1 anders anders 84 jan 16 17:36 skal.sh
$ ./skal.sh skal.sh a b
Test skal.sh a
7
Test $1
```

Notera bakvända divisionstecknet (`\`) framför dollar-tecknet (`$`). Det måste vara där, annars så kommer `$1` att ersättas med argument ett till detta skal, vilket är tomt. Detta kallas slarvigt att man *quotar* värdet av `$`, dvs man betraktar `$` som vilket tecken som helst och inte något specialtecken.

Vill man mata in ett `\`-tecken, så dubblerar man bara tecknet, så här: `\\`. Man kan quota med `\`-tecknet framför ett annat tecken eller skriva en text mellan två `"`-tecken eller två `'`-tecken. Prova gärna skillnaden mellan dem.

Tips: Prova `echo \$PATH`, `echo "$PATH"` samt `echo '$PATH'` och jämför utskriften. Prova även vad texten `AB C DE` ger för resultat (dvs med två mellanslag mellan `B` och `C` samt ett mellanslag mellan `C` och `D`) om man skriver ut texten med `echo`(1)-kommandot.

Så ett program som använder argument kan se ut så här:

```
$ more skal.sh
#! /bin/bash
# Ett testprogram
echo Test ""
wc -l < "$1"
echo Test "\$1"
exit
# EOF
```

Loopar i skalprogram

Man kan även göra loopar i skalprogram som repetera saker. Det gör man med `for`-kommandot, där man först anger en variabel som för varje varv tilldelar ett varde från de uppräknade värdena efter `in`. De kommandon som skall utföras för varje värde i loopen skrivs mellan `do` och `done`. Där kan man komma åt värdet hos variabeln i `for`-kommandot.

För att få hjälp hur det ser ut, så titta i `bash(1)` manual eller skriv `help for` i ett skal. Det kan se ut så här i ett litet skript. Skriv in kommandona i en fil `exempel1.sh`, så har ni ett litet skript. Glöm inte att ge filen `x`-rättigheterna (se ovan hur man gör det):

```
#! /bin/bash
# Skriv ut orden i listan
for ordet in aa b ccc d ; do
    echo "Ordet är: $ordet"
done
```

Ett annat exempel är att man räknar upp alla filer som finns i en katalog och sedan i loopen kan man göra saker med filerna. Det enklaste är att skriva ut dem. Notera då att * kommer att bytas ut mot alla filer i katalogen, prova `echo *` i ett kommandoskal.

```
#! /bin/bash
echo *
for filnamn in * ; do
    echo "Filnamn: $filnamn"
    # cat "$filnamn" # Bortkommenterat
done
```

Man kan skriva flera kommandon mellan `do` och `done` , som då kommer att exekveras per varv.

Ett annat sätt att använda en `loop` är att skriva ut alla argument som man har startat skriptet med. Om man skriver `$0` så får man reda på vad programmet heter. Sedan är `$1` till `$9` argument `1` till `9` till programmet. Alla argumenten uppdelade som ord får man med `$@` och antalet argument med `$#`. Det finns fler specialvariabler, som man med fördel hittar i manualsidan för `bash(1)` under sektionen `Special Parameters`. Här är ett exempelprogram som ni kan mata in och prova nedan. Döp gärna programmet till `skal2.sh`.

```
#! /bin/bash
# Skriv ut alla argument
for argument in $@ ; do
    echo "Argument: $argument"
done
```

Man kan även testa om exempelvis en fil är exekverbar. Då kan man använda sig av `if-else`. Det kan se ut så här om vi går igenom alla filer i aktuell katalog. Vilka tester som kan göras ser man med `help test` eller `man test`.

```
#! /bin/bash
# Skriv ut filnamn och om den som kör programmet har x-rättighet
for filnamn in $@ ; do
    if [ -x "$filnamn" ] ; then
        echo "Filen $filnamn är exekverbar"
    else
        echo "Filen $filnamn är inte exekverbar"
    fi
done
```

Noter att `if` slutar med `fi`. Skriv in skripten ovan och provkör dem med lite olika argument till kommandot.

Det är viktigt att sätta ut `"`-tecknet. Ni kan prova vad som händer om man tar bort det i skripten eller ersätter det med `'`-tecknet. Prova även att starta första skriptet med dessa argument:

```
./skal2.sh skal.sh a b
./skal2.sh "skal.sh a" b
./skal2.sh 'skal.sh a' b.
```

Prova även med att byta ut `skal.sh` mot * som argument i exemplen till kommandon ovan.

Linux katalogstruktur

I Linux så finns det bara en katalog från vilken man hittar alla andra filer och kataloger, nämligen den så kallade rot-katalogen (`/`). Så det finns inte några enhetsnamn eller liknande som i exempelvis MS Windows, där varje enhet har en egen rot-katalog. På engelska kallas den följaktligen då *root directory*.

Så för att inte vara begränsad till en hård-disk, så kan man *montera* diskar med filsystem på olika kataloger. När man sedan går ned i den katalogen, så kan man automatiskt byta till det filsystem som är monterad på katalogen. Detta är praktiskt då användarna och programmen inte behöver bry sig om på vilka enhet som finns. För användare och program finns bara en logisk filstruktur.

Standard Linux katalogstruktur

Debian använder en filstruktur som kallas *Filesystem Hierarchy Standard* (FHS). Det innebär att program kan finnas på ett av flera ställen, beroende på vilka typer av program det är.

Program som behövs för att starta datorn och för att köra vissa serverprogram ligger normalt under `/bin/` (*binary*, program kallas binary) och `/sbin/` (*system binaries*). I katalogen `/sbin/` så är program som normalt inte används av vanliga användare utan av serverprogram eller administratör.

Program som används när datorn har startat (bootat) finns då i `/usr/bin/` och `/usr/sbin/`, där *usr* kan uttalas *user*. Varför inte allt under `/bin/` utan vissa delar i `/usr/bin/`? Historiskt så var inte diskar så stora, så man var tvungen att dela upp program och filer på flera diskar. Därför blev det denna uppdelning, där `/usr/` är tillgängligt efter att maskinen är klar med starten.

Numera går fler distributioner över till att göra `/bin/` ett alias till `/usr/bin/`.

Program som den lokala systemadministratören skapat för alla användare på maskinen ligger normalt under katalogerna `/usr/local/bin/` och `/usr/local/sbin/`.

Slutligen så kan vissa program som kommer från tredje-part hamna under `/opt/`*`paket`*`/`, där `/opt/`*`paket`*`/bin` har programmen. `paket` är normalt förkortning på företagets eller programsystemets namn.

Bredvid katalogerna `bin/` och `sbin/` kan katalogen `lib/` finnas, som används till att lagra programmeringsbibliotek och data till program. Katalogen `man/` för manualsidor, `share/` för datafiler som kan delas mellan flera program och datorer, `var/` för att lagra filer som kan variera i storlek, som loggfiler etc.

Under *root*-katalogen finns även `/tmp/` och `/boot/`. Katalogen `/tmp/` används för att lagra temporära filer och `/boot/` för att lagra de filer som behövs för att starta operativsystemet. Numera så använder man helst `/var/tmp/` istället för `/tmp/`.

En katalog som kan vara extra bra att känna till för Debian-användare är `/usr/share/doc/` som innehåller information om installerade Debian-paket. De ***bör*** man alltid titta i om man är nyfiken eller behöver konfigurera ett program. Dessa kataloger kan då innehåller information som kan vara viktigt att veta för just Debian-användare. Ibland innehåller de mycket information och ibland bara några få filer.

Katalogen `/var/log/` är även den väldigt bra att känna till. Vill man veta vad som händer i systemet, så kommer program att skriva information i några av dessa log-filer som finns där. Här använder man med fördel kommandot `tail -f` *`filer`* i ett terminalfönster för att se vad som händer i dem. Viktiga är filerna `syslog`, `messages` och `dmesg`. Så titta gärna i dem.

Det finns flera kataloger som kan vara bra att känna till, men de beskrivs mer noggrant i referenserna nedan.

Interaktiva tips när man använda skalprogram

När man skriver in kommandon i `bash(1)`, så kan man alltid prova att trycka på `Tab`-tangenten. Om `bash` hittar någon kommando som börjar på samma bokstäver, så kommer `bash` att antingen skriva kommandot, eller om det finns många alternativ lista alla alternativen.

`Tab` fungerar både på kommandon och på filnamn, så använd `Tab` ofta! På så sätt så slipper man skriva fel, och är ett av de viktigaste sätten för att jobba effektivt i kommandoskalet.

När jag jobbar med kommandoraden, så händer det ofta att samma/liknande rad som jag gjort tidigare skall göras igen. Då använder jag gärna uppåtpilen eller `Ctrl-p` (`p` för *previous*[8]) för att få föregående rad. För att gå åt andra hållet om man stegat för långt, så kan man prova nedåtpilen eller `Ctrl-n` (`n` för *next*).

Om jag vill modifiera så flyttar jag markören med vänster och höger pil (eller `Ctrl-f` för *forward* eller `Ctrl-b` för *back*) för att flytta mig framåt eller bakåt på raden.

För att ta bort tecken använder jag mig av `Backspace` eller `DEL`-tangenterna. För att lägga till tecken så är det bara att skriva in de nya.

Vill jag snabbt flytta mig till början eller slutet av en rad, så använder jag mig av `Ctrl-a` eller `Ctrl-e` (`a` för början av alfabetet och `e` för *end of line*).

Ni kommer att tjäna mycket tid genom att använda dessa kortkommandon, så lär er gärna dem. Ett tips kan vara att dessa kommer från textredigeraren `Emacs`, vilket är en riktigt kapabel editor. :-)

Bra att ha-länkar

- https://sv.wikipedia.org/wiki/Bourne_shell
- https://sv.wikipedia.org/wiki/C_shell
- https://sv.wikipedia.org/wiki/Bash
- https://sv.wikipedia.org/wiki/Tcsh
- https://sv.wikipedia.org/wiki/Emacs
- https://tldp.org/LDP/Bash-Beginners-Guide/html/index.html
- https://tldp.org/LDP/abs/html/index.html
- https://bash.cyberciti.biz/guide/Main_Page
- https://sv.wikipedia.org/wiki/Filesystem_Hierarchy_Standard
- https://www.howtogeek.com/117435/htg-explains-the-linux-directory-structure-explained/
- https://sv.wikipedia.org/wiki/Emacs
- https://www.gnu.org/software/emacs

8 Den text som jag skriver kursivt här är minnesregler som jag använder för att komma ihåg kortkommandona.