Kapitel 11

Programmering och skalskript

EN GÅNG ÄR INGEN GÅNG -- TVÅ GÅNGER ÄR ETT SKRIPT.

11.1 Programmering

Som systemadministratör måste man inte vara expert på programmering, men det är bra att kunna grunderna. Både för att kunna automatisera vissa saker och inte minst för att förstå sina användare om de är programmerare.

11.1.1 Maskinkod

Maskinkod är det enda som en dators CPU förstår. Denna kod är ettor och nollor, binära tal, som man sällan skriver direkt i. Men det är bra att känna till att alla andra program till slut, på olika vis, blir maskinkod.

11.1.2 Assembler

Assembler är i stort sett en-till-en-mappning från korta instruktioner till maskinkod. Alltså fortfarande på mycket maskinnära nivå, men lite mer läsbart än maskinkod. En CPU har interna register, en sorts snabba minnesplatser. I assembler anger man vad som ska sparas och räknas med direkt i dessa register. Rader från ett assemblerprogram för x86 (PC) kan se ut så här:

```
tal1 dw 4
tal2 dw 5 ;två variabler, tal1=4 och tal2=8
mov Ax, tal1 ;lägg tal1 i register Ax
add Ax, tal2 ;addera Ax och tal2, spara svaret i Ax
```

Assembler används numera ganska sällan. Det används i vissa delar av Linuxkärnan där det är viktigt med snabbhet och för inbyggda system där man måste optimera för liten minnesanvändning. Olika CPU-familjer har helt olika assemblerspråk, t.ex. x86, MIPS, Motorola 68k.

11.1.3 C

Språket C används mycket fortfarande. Nästan alla program du använder dagligen i GNU/Linux är skrivna i C. Historiskt har UNIX och C samma ursprung och upphovsman (Dennis Ritchie). De första UNIX-versionerna var skrivna i PDP-assembler, men för 4:th edition (1973) skrevs det mesta i

det då nya språket C. C är ett kompilerat språk, det innebär att ett program kallat kompilator (numera ofta GCC) översätter från programspråk till maskinkod en gång och sen använder man enbart den kompilerade versionen vid körning av programmet. Jag förklarar med ett exempel (det vanliga Hello World men med en variabel tillagd):

```
#include <stdio.h>
int main()
{
  int siffra = 1;
  printf("hello, world number: %d", siffra);
}
```

Om du sparar den texten i en fil som heter `hello.c` kan du kompilera den med t.ex. `gcc -o hello hello.c` och när du sen kör ./hello skrivs texten `hello, world number: 1` ut i terminalen.

Detta lilla exempel kräver en lite längre förklaring. En variabel är ett namn på ett värde, i det här fallet ett heltal (int=integer) som heter siffra och har värdet 1. Rader med # i början är direktiv till preprocessorn, de kan bland annat vara #include, #define och #ifdef. Den #include vi har här lägger till stdio.h från glibc där funktionen printf() finns. Funktioner används mycket i C, båda de som finns i olika programbibliotek och programmerarens egna. Argumenten till funktioner anges inom parenteser, i det här fallet en sträng där %d anger var decimaltalet i variabeln siffra ska stoppas in. En funktion som alltid ska finnas i ett komplett program är main( ), som anger att det är där man ska starta exekveringen.

Linuxkärnan är skriven i C, förutom någon procent assembler. Så är även de flesta kommandon du har i `/bin/` och `/usr/bin/`. Även demoner som Apache, BIND och Sendmail är C-program.

Det är inte så svårt att lära sig grunderna i C, men det är lätt att göra misstag, främst vad gäller indata från användare eller nätverk. Indata lagras i en buffert, och om den inte skrivs rätt kan elaka personer, eller program, exekvera kod via så kallad buffertöverskridning (engelska: buffer overflow). Även pekare kan ställa till problem.

Med C programmerar man på ganska låg nivå, alltså nära maskinen. Med `gcc -S filnamn.c` kan man få sin kod visad som assembler. För programmet ovan blir det 31 rader assemblerkod. Det är inte så användbart, men kan vara kul att prova för att förstå mera.

Vill du lära dig C är boken *The C Programming Language* av Brian Kernighan och Dennis Ritchie fortfarande bland de bästa. Den kallas av de insatta kort och gott för K&R. Första upplagan kom 1978, och den andra (som är den du bör läsa) kom 1988.

11.1.4 C++

C++ skapades av Bjarne Stroustrup i början av 1980-talet, och bygger vidare på C men har lagt till objektorientering och andra standardbibliotek. I C++ kan Hello World se ut så här:

```
#include <iostream>
int main() {
   std::cout << "Hello World!" << std::endl;
   return 0;
}
```

Kända program som är skrivna i C++ är stora delar av KDE och Mozilla (Firefox och Thunderbird).

Mer om C++ kan man läsa i Stroustrups egna bok *The C++ Programming Language* (Hello World ovan är lånad från den boken).

11.1.5 Perl

Perl är kanske det språk som systemadministratörer har mest nytta av att kunna. Det skapades av Larry Wall 1987 och är mycket flexibelt, språkets motto är TIMTOWTDI: ”There is more than one way to do it’. Perlprogram kan dock bli ganska kryptiska, med många speciella egenheter.

Perl har tre datatyper för variabler. Den vanligaste är skalär och anges med ett $ först i variabelnamnet. En skalär kan heta $value och motsvarar både heltal, flyttal, tecken och sträng i andra språk, och den kan omvandlas mellan

dessa beroende på sammanhang. Nästa datatyp är array och anges med ett @-tecken i namnet, t.ex. @lista. Den är som en vektor eller lista av värden. Den tredje datatypen är hash och anges med ett %-tecken. En hashtabell är en mappning mellan nycklar och värden.

Perl har mycket bra stöd för reguljära uttryck, och det finns moduler för många saker i ett bibliotek som heter CPAN. Med Perl kompilerar man normalt sett inte sina program, utan koden sparas som vanlig text och tolkas vid varje körning.

Ett Perl-skript kan se ut så här:

```
#!/usr/bin/perl
@lines = `cd ~/jobblog ;ls -1 *redovisa*`;
foreach $rad (@lines) {
  $datum = substr($rad,14,6);
  print "$datum: ";
  $_ = substr($rad,21,3);
  s/\..*//;
  $summa += $_;
  print ("X"x$_," $_\n");
}
print "Summa: ", $summa, " timmar\n";
```

Det är ett litet skript som skriver ut en tabell över mina jobbade timmar med indata i form av filnamn som `log.redovisat.110701.51.549` och ger utdata med staplar och summa. Det är inget mästerverk och kan säkert förbättras. Om du kan lite C så ser du att det finns stora likheter, Perl har lånat idéer från många håll. Larry Wall är lingvist, så det har vissa likheter även med naturliga språk.

För mer info om Perl rekommenderar jag att börja med att läsa *Learning Perl* av Randal L. Schwartz, brian d foy och Tom Phoenix (O'Reilly) (Jag vet inte varför brian vill ha sitt namn med små bokstäver). När man har läst "Learning" kan man fortsätta med *Intermediate Perl* från samma förlag, följt av *Programming Perl* och *Perl Cookbook* (även dessa två är från O'Reilly men är tjockare och används mer som uppslagsböcker).

11.1.6 Python

Python är ett annat populärt skriptspråk. En egenhet med det är att indentering är syntax. Gör du inte indrag rätt så fungerar inte programmet alls. I andra språk är det ju ofta {} som markerar block och indrag är mest för läsarens skull.

Python har stort stöd för objektorientering, nästan allt är objekt. Python skapades i slutet av 1980-talet av Guido von Rossum.

Boktips om Python är *Learning Python* av Mark Lutz (4th ed, O'Reilly) och *Dive into Python* av Mark Pilgrim (Apress).

11.1.7 PHP

PHP används mest på webben för skript på serversidan. Men det går att skriva andra skript i PHP. Bland annat har PHP bra stöd för databaser. PHP uppfanns av Rasmus Lerdorf 1995.

11.1.8 Java

Java används ofta på grundläggande kurser i programmering. Det liknar C++. När man kompilerar ett javaprogram blir det en bytekod som körs i en virtuell maskin. Detta gör att man kan använda så kallade applets på webben, oavsett operativsystem. Java uppfanns av James Gosling på Sun Microsystems 1995.

En lättläst svensk bok om Java är *Java – steg för steg* av Jan Skansholm från 2012 (Studentlitteratur).

11.1.9 LISP

LISP skapades redan 1958 av John McCarthy på MIT. Det används mycket inom AI (Artificiell intelligens) och för editorn Emacs. En egenhet med LISP är att det blir många parenteser i koden.

En underhållande och lättläst ny bok om LISP är *Land of Lisp* av Conrad Barski (No Starch Press).

11.2 Skriva skript

Nästan allt som man kan göra vid prompten kan man spara i ett skript, och tvärtom kan man göra samma sak som i korta skript med en enradare vid prompten. Ett skalskript är bara en fil där man skriver ner en följd av kommandon. Givetvis kan man använda kontrollstrukturer som upprepning (iteration) och val (selektion). Skript som bara du ska använda sparas gärna i `~/bin/`. Skript som bara root ska använda kan sparas i `/root/bin` (eller `/usr/local/sbin`) och skript för alla kan du lägga i `/usr/local/bin`. Glöm inte att göra filen exekverbar med `chmod 755`, `chmod u+x` eller liknande. Man kan, som sagt, skriva skript i riktiga programmeringsspråk som Perl, Python eller PHP. Men jag tar här främst upp skalskript.

Ett skalskript kan se ut så här:

```
#!/bin/bash
echo -n "Hej "
whoami
```

Det gör inte så mycket nytta, men kan illustrera grunderna. Första raden inleds med tecknen #!, de kallas ofta shebang och följs direkt av en absolut sökväg till programmet som ska tolka resten av skriptet. För bash kan man utelämna denna rad om alla som använder skriptet har bash som standardskal, men det är bäst att ta med ifall någon kör annat skal (zsh kommer nog funka, men förmodligen inte csh för mer avancerade skript). Skriver du skripten i Perl ska du inleda med #!/usr/bin/perl.

Nästa rad använder kommandot `echo` för att skriva till STDOUT. Växeln `-n` gör att det inte blir någon ny rad. Tredje raden anropar kommandot `whoami` som helt enkelt skriver ut vem du är inloggad som.

Vissa tycker att man ska undvika Bash-specifik syntax i skalskript, och istället göra dem kompatibla med original Bourneshell. Då använder man #!/bin/sh på första raden och vet vad som funkar på alla system. Men ofta skriver man skripten för användning endast på en maskin, eller några, och Bash finns till de flesta system. Så jag har /bin/bash i alla exempel i denna bok. På vissa GNU/Linuxsystem är /bin/sh en länk till /bin/bash. På Debian och Ubuntu (från

version 6.10) är dash standardskal. Dessutom uppför sig bash lite annorlunda om det anropas som sh.

Här kommer ett annat exempel på ett enkelt skript. Det kollar vilka som besökt dina webbsidor:

```
#!/bin/bash
tail -10000 /var/log/httpd/access_log| grep '/~ao'
```

Det är avsett att köra på en maskin med många besökare (hits), så den kollar bara de tiotusen sista raderna, och grep:ar efter mitt användarnamn (byt ut ao mot ditt eget login). Detta är typiskt en sak som går lätt att göra vid prompten, men istället för att upprepa sparar man i ett skript. Så jag skriver bara `aoweblog` för att se de senaste besökarna.

Det går givetvis att starta grafiska saker om man kör X på sin maskin, här är ett skript för att öppna nio terminaler på specifika platser:

```
#!/bin/bash
xterm -geometry 84x27+0+0 &
xterm -geometry 84x27+533+0 &
xterm -geometry 84x27-0+0 &
xterm -geometry 84x27+0+403 &
xterm -geometry 84x27+533+403 &
xterm -geometry 84x27-0+403 &
xterm -geometry 84x27+0-30 &
xterm -geometry 84x27+533-30 &
xterm -geometry 84x27-0-30 &
```

Jag brukar kalla det `9xterm` och lägga till som startare i GNOME-panelen. Lägg märke till att man måste ha & på varje rad så programmen körs i bakgrunden. (Se 12.2 för förklaring).

11.3 Kontrollstrukturer

11.3.1 for

Även Bash är ett programmeringsspråk där man kan upprepa och välja. För att upprepa kan man till exempel använda `for` och `while` och för att välja finns bland annat `if` och

`switch`. Dessa kommandon är inbyggda i kommandotolken, och kan användas både i skript och enradare.

Kommandot `for` i Bash fungerar inte som i Java, Perl eller C, där det är tre argument: `for(i=1;i<10;i++)`, utan det är mer som foreach i Perl. Man anger en lista av namn och for-loopen går igenom dem en efter en. Det kan vara enklare att förklara med exempel:

```
for dator in zeuz hera poseidon athena
do
   ping -c 1 $dator
done
```

Variabeln dator är här alltså den som varierar för varje upprepning. Första varvet är det zeus som pingas och i nästa är det hera. Vill man göra samma sak på en rad blir det:

```
for dator in a b c d e; do ping -c 1 $dator; done
```

Det måste alltså vara ny rad eller semikolon efter listan som ska itereras och efter varje kommando inne i loopen. Nyckelorden do och done används för att ange var loopen börjar och slutar, inga måsvingar eller andra specialtecken. Indraget i första exemplet är bara för läsarens skull, Bash bryr sig inte om sådant.

Listan som kommer efter nyckelordet `in` kan skapas ”i farten”, till exempel med * eller kommandosubstitution. Till exempel:

```
for fil in * ; do mv $fil $fil.bak; done
```

eller:

```
for fil in $(find . -name *.jpg); do
   mv $fil ~/bilder
done
```

I exemplet med * kommer Bash att expandera asterisken till namnet på filer och kataloger där du står. Detta görs tidigt i tolkningen av raden, så `for` kommer se en vanlig lista separerad av mellanslag. Samma sak sker alltid, till exem-

pel vid `cat *` så ”ser” cat aldrig *, skalet byter ut det mot filnamnen (om inte katalogen är tom). Testa med `echo *`.

I nästa exempel söker man rekursivt efter jpg-bilder och flyttar dem till en specifik katalog. Syntaxen med $() går även att göra med backticks (bakåtvända citattecken) men det blir snyggare med dollar och parentes. Det som händer är att kommandot mellan parenteser körs först, och *for* får utdatan som en vanlig lista.

Dessa skript kan få problem om man har mellanslag i filnamn, så här kommer en favorit som jag kallar *tabortspace*:

```
for f in *; do mv "$f" `echo $f |tr ' ' '_'`; done
```

Den kräver kanske inte mer förklaring än att tr byter ut enstaka tecken. Resten är sådant vi nyss lärt oss (men med backticks).

Man kan även använda `for` utan `in`. Då tar den argumenten på kommandoraden som lista. Till exempel:

```
for fil
do
cp $fil /home/backup/
done
```

Om du sparar det som `/usr/local/bin/backup` kan du till exempel köra:

```
cd /etc
backup passwd group
```

och få säkerhetskopia av de filer du anger.

11.3.2 while

För att förstå `while` (och senare `if`) behöver man förstå slutstatus (exit status). Alla kommandon som körs klart lämnar en siffra till sin föräldraprocess, det vill säga till skalet om de körs från prompten. Noll betyder OK och 1–255 något fel. Man kan se slutstatus med skalets variabel $?. Testa till exempel `grep root /etc/passwd; echo $?` och `grep Root /etc/passwd; echo $?` Den första bör ge värdet 0 och den andra värdet 1. Att grep hittar en rad räknas som rätt och att inte hitta är fel.

I många programmeringsspråk använder man `for` när man vet antalet varv innan början och `while` när det är okänt och avgörs efterhand. På liknande vis är det i Bash. Syntaxen för while är:

```
while kommando1; do kommando2; done
```

Den kör alltså `kommando2` så länge som `kommando1` returnerar noll. Man kan givetvis utföra mer än ett kommando, och om man har flera som testas är det slutstatus från det sista som räknas. Jag brukar ibland kombinera `while` med en `sleep` för att upptäcka saker som bara ändras ibland. Ett trivialt exempel som man kan roa sig med på datorer med flera användare:

```
while who|grep -q kalle
do echo "kalle är inloggad"; sleep 60; done;
echo "kalle loggade ut"
```

Det är kanske inte så användbart, men det förklarar `while`. Växeln `-q` till `grep` betyder quiet, det är ju bara slutstatus vi (och `while`) bryr oss om. While kör vidare så länge raden kalle finns i utdata från `who`, och den väntar en minut mellan varje gång den kollar. När loopen är slut (raden kalle finns inte) körs nästa kommando efter done.

Ett vanligt sätt att använda while är med `true`, ett kommando som alltid ger sant (noll) som slutstatus:

```
while true; do något; done
```

Då får man själv avbryta med **ctrl-c**, eftersom `true` är ett kommando som alltid returnerar 0. Även denna kan kombineras med sleep. Till exempel kan man göra så här för att var tionde sekund dumpa processlistan till en fil:

```
while true; do (date;ps aux)>>processer; sleep 10;
done
```

I det skriptet blev det även ett exempel på hur man kan starta kommandon i ett subskal med (), för att slippa dirigera om utdata två gånger. Kommandona `date` och `ps aux` körs i ett eget skal, och båda dirigeras om med dubbla pilar, alltså append, till en fil.

11.3.3 if

Även `if` använder sig av ett programs slutstatus, men här för att välja antingen en sak eller en annan sak, selektion. Grundsyntaxen är:

```
if kommando1; then kommando2; fi
```

(jag krånglar till den mera snart). Alltså som vanligt bara ny rad eller semikolon för att avsluta respektive kommando (eller lista av kommandon). Nyckelorden för att separera är `then` och för att avsluta `fi`. Det är förstås if baklänges, äkta nördhumor.

En enkel if-sats är:

```
if grep -q '^ao:' /etc/passwd; then echo "ao finns";
fi
```

Tecknet cirkumflex (^) gör att man bara matchar början av en rad, och kolon är det man separerar fält med i filen passwd, så den matchar inte andra login som börjar på samma bokstäver.

Med bara `if then fi` kommer man fortsätta efter `fi` oavsett vad som väljs. Ofta vill man ha två eller flera val, antingen-eller. Till det använder man `else`. Exempel:

```
if grep -q .pdf /var/log/httpd/access_log
then
   echo "Någon har kollat en pdf"
else
   echo "Ingen har hittar pdf:en än"
fi
```

Även här är indenteringen frivillig, men den gör det mer lättläst. Även om man har `else` måste man avsluta med `fi`.

Ett vanligt kommando att använda direkt efter `if` är `test`. Det är så vanligt att det har fått en egen förkortning, nämligen `[`. Det betyder exakt samma sak som `test`, och är ett kommando, inte bara ett syntaxtecken. Med `test` eller `[` kan man testa många saker: om variabler har samma värde, om filer existerar och är av viss sort och mycket mera. Se help test för mer info. Som vanligt är det slutstatus 0 för sant och 1 för falskt från test. När man använder hakparentes avslutar man med en hakparentes åt andra hållet. Man måste ha mellanslag efter den första och före den sista. Till

exempel (med helt olika saker för varje if-sats):

```
if [ -f /etc/group ]; then echo "group finns"; fi

if [ $SHELL = /bin/bash ]; then
   echo "Du kör bash";
fi

if [ $(who|wc -l) -lt 10 ]; then
   echo "färre än 10 användare inne";
fi
```

I längre skript har man ofta flera if-satser, och kombinerar med `while` och `for`. Vill man välja mellan fler än två alternativ kan man även använd `elif` inne i en if-sats, men det blir ofta snyggare med `case`, den avslutas förstås med `esac`. Dessa är dock inte alltid utbytbara, med `elif` kan man ha helt olika villkor, med `case` testar man bara olika värden på ett enda uttryck.