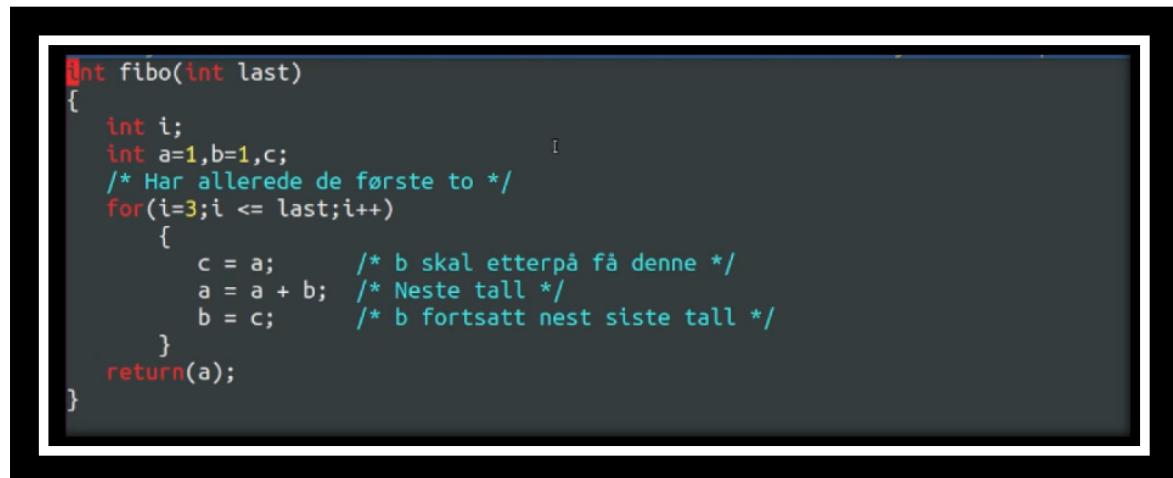


NOTATER UKE 7

Eksempel på kode som går i loop og bygger på hverandre. Denne koden kan ikke utføres i parallelle, men hvordan kan man utføre kode effektiv i en CPU?

Noen instruksjoner utføres bare på registrene, da står man inni CPU med registre tilgjengelig f.eks: R0 + R1, som vi gjorde i simuleringen. Vi kunne også legge verdier ute i RAM, det er noe man typisk gjør når man har variabel som int a=1 nede, variabler har egne plass i RAM, en integer har 4 byte (32 bit). Det tar ti ganger så lengre tid å lagre ett tall i RAM enn registeret.



```
int fibo(int last)
{
    int i;
    int a=1,b=1,c;
    /* Har allerede de første to */
    for(i=3;i <= last;i++)
    {
        c = a;      /* b skal etterpå få denne */
        a = a + b; /* Neste tall */
        b = c;      /* b fortsatt nest siste tall */
    }
    return(a);
}
```

Men hvordan lager en kompilator maskinkode ut ifra koden over? Under vises en main som skal kompileres og linkes sammen med fibo(int last) over.



```
haugerud@studssh:~/2020/fibo$ cat main.c
#include <stdio.h>

extern int fibo();

int main(void)
{
    int last=10;
    int res = fibo(last);
    printf("Res: %d \n",res);
}

haugerud@studssh:~/2020/fibo$
```

Extern int fibo() er C-metoden som skal brukes (bildet øverst på siden). I main sendes tallet 10 inn til fibo.

Med gcc main.c fibo.c kan man kompilere disse i samme setning, og kjøre ved ./a.out.

Det vi vil se på er hvordan ser maskinkoden ut, som det kompileres til fra høynivåkode? Man kan spørre gcc hvordan assembly koden ser ut som. Da skriver man `gcc -S <navnetPåKoden>`. Og det vil da i `ls -l` dukke opp en fil som heter `<navnetPåKoden>.s` (MERK! ikke `.c`). Da kan man hente innholdet til fila med kommandoen `cat`:

```

.LFB0:
    .cfi_startproc
    pushq   %rbp
    .cfi_offset %rbp, 16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    .cfi_offset %rbp, 0
    movl    %edi, -20(%rbp)
    movl    $1, -12(%rbp)
    movl    $1, -8(%rbp)
    movl    $1, -16(%rbp)
    jmp    .L2
.L3:
    movl    -12(%rbp), %eax
    movl    %eax, -4(%rbp)
    movl    -8(%rbp), %eax
    addl    %eax, -12(%rbp)
    movl    -4(%rbp), %eax
    movl    %eax, -8(%rbp)
    addl    $1, -16(%rbp)
.L2:
    movl    -16(%rbp), %eax
    cmpl    -20(%rbp), %eax
    jle    .L3
    movl    -12(%rbp), %eax
    popq    %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size    fibo, .-fibo
    .ident   "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.11) 5.4.0 20160609"

```

Her brukes hele tiden RAM. Vi har tidligere sett at en `-4(%rbp)` konstruksjon er en peker til en integer i RAM. Det som er inni `()`, er et register, adressen til ram ligger inni registeret, og `-12` er en relativ adresse som sier at akkurat den integeren vi skal hente ligger `-12` byte unna starten av det området. Herifra må vi få med oss at begge er en variabel. Vi kan konkludere med at alle operasjonene som utføres som f eks det **oransje**, utføres på integer som ligger i RAM. Der sies at legg tallet (adder) 1 til variabelen som ligger i RAM. Vi kan se at det er tellevariablen i løkken fordi det øker med 1 hver gang.

Når gcc lager kode, skriver den inn og ut av RAM hele tiden. Når det gjelder add så er det et register og et element fra RAM.

`%eax` er et register i x86-arkitekturen som brukes i intel-kompatible prosessorer. Det kan brukes til å lagre data eller adresseinformasjon. «eax» delen av navnet står for «utvidet akkumulator», og «%» symbolet brukes til å angi at det er et register. Alle sånne referanser er

til registeret, og det er et extended ax register, 32 bits register. Når gcc kompilerer så lager den 62 bits kode, og man kan endre det ved kompilering. Når vi kompilerer denne koden, så lager kompilatoren kode som går ut i minne, og lagrer ting hele tiden i minne. I x86 er det ikke lov/definert operasjoner som bruker to minneadresser samtidig. Men man kan legge til et tall med en variabel eller registeret i et variabel direkte. gcc default lager maskinkode det er raskest mulig å kompilere. Hvis man ønsker den skal lage et mest mulig effektivt program (kjører så fort som mulig) skriver man -O:

```
gcc -O main.c fibo.c
```

Men hvordan ser koden ut da? Hvordan vil gcc endre koden slik at koden skal gå enda forttere?

```
gcc -O -S fibo.c
```

Stort sett vil da operasjonene foregå inne i CPU-en med registere. Hovedpoenget er at alt som gjøres, gjøres inne i CPU uten å referere til RAM. Det mest effektive er å kjøre det inne i CPU-en. Det er også begrenset lagringskapasitet i RAM, og hvis man f eks har store array, kan man ikke ha alt i RAM, dermed må de laste inn og ut av RAM.

NB! Fra og med 2022 har default konfigurasjon av kompilatoren gcc endret seg, slik at man nå må legge på oppsjonen -no-pie for å kompilere assembly-kode som deklarerer variabler i et data-segment. Derfor må man kompilere med

```
gcc -no-pie enlinje.c en.s
```

Under er et datafelt: det er måten man legger variabler på i RAM på. Den første linjen sier lag et variabelt svar og legg i RAM. .quad betyr lagre et 64 bit (8 byte) og vil inneholde tallet 32.

```
.data
svar:  .quad 32    # deklarerer variablen svar i RAM
memvar: .quad 10   # 8 byte = 64 bit variable
```

```
.globl enlinje
# C-signatur:int enlinje ()

enlinje:      # Standard start av funksjon

# add memvar, svar
mov memvar, %rbx # Man trenger to linjer kode for å
add %rbx, svar  # gjøre en høynivålinje svar = svar + memvar
mov svar, %rax  # Returnerer svar

ret # Verdien i rax returneres

# Følgende avsnitt av koden viser hvordan man definerer
# variabler som lagres i minnet.
# Andre linje tilsvarer linjen
# int svar=32;
# i et C-program
# Dette avsnittet kunne også stått øverst i filen

.data
svar: .quad 32  # deklarerer variablene svar i RAM
memvar: .quad 10 # 8 byte = 64 bit variable
```

Inni CPU-en har vi flere register: ax (16 bit), bx og cx osv. eax er på 32 bit, rax er på 64 bit.
Inni CPU har vi en ALU. Det som skjer, er at man har en kanal fra registrene som går inn i ALU og resultatene kommer tilbake til registrene.

ASSEMBLY-KODE FRA SCRATCH:

Under er koden til iftest() i høynivåkode.

```
haugerud@studssh:~/2020/as2/if$ cat if.c
int iftest()
{
    int svar = 32;
    if(svar > 42)
    {
        return(1);
    }
    else
    {
        return(0);
    }
}
```

globl. Iftest → definerer at den heter det
label iftest → standard måte å definere en funksjon på
under er iftest, greater og return med kolon «labels».



```
.globl iftest

iftest:
# int svar = 32;
#     if(svar > 42)
#     {
#         return(1);
#     }
#     else
#     {
#         return(0);
#     }

    mov $42,%rbx
    cmp %rbx, svar
    jg greater

    mov $0, %rax
    #jmp return
    ret

greater:
    mov $1, %rax

return:
    ret # Verdi returneres i %eax

.data
svar: .quad 42
```

Over er assembly koden han lagde fra bunns i videoen.

Forenklinger ved CPU-simuleringen vi hadde

Den virker i prinsipp som vanlig men har en rekke forenklinger. For det første bruker instruksjoner mer tid enn en CPU-sykel på å utføres. Hver instruksjon hentes inn fra RAM før den utføres (ikke ROM). En x86-instruksjon deles inn i flere små deler (mikro-operasjoner), uops

WHILE CPU LØKKE

Vi har en CPU-løkke, som while(not HALT), så lenge maskinen ikke blir skrudd av så utfører den instruksjoner. Hvis den ikke har noe å gjøre så må den utføre en not-instruksjon (no operasjon) og slår i lufta. En CPU- er også koblet med annen hardware og kan dermed av mange andre komponenter bli avbrutt, og da blir den stoppet. Den må da behandles. F eks hvis man taster inn noe med tastaturet så må CPU her stoppes opp og det som ble tastet inn må behandles.

```

while(not HALT)
{
    IR = mem[PC];      # IR = Instruction Register
    PC++;               # PC = Program counter
    execute(IR);
    if(InterruptRequest)
    {
        savePC();
        loadPC(IRQ); # IRQ = Interrupt Request
                        # Hopper til Interrupt-rutine
    }
}

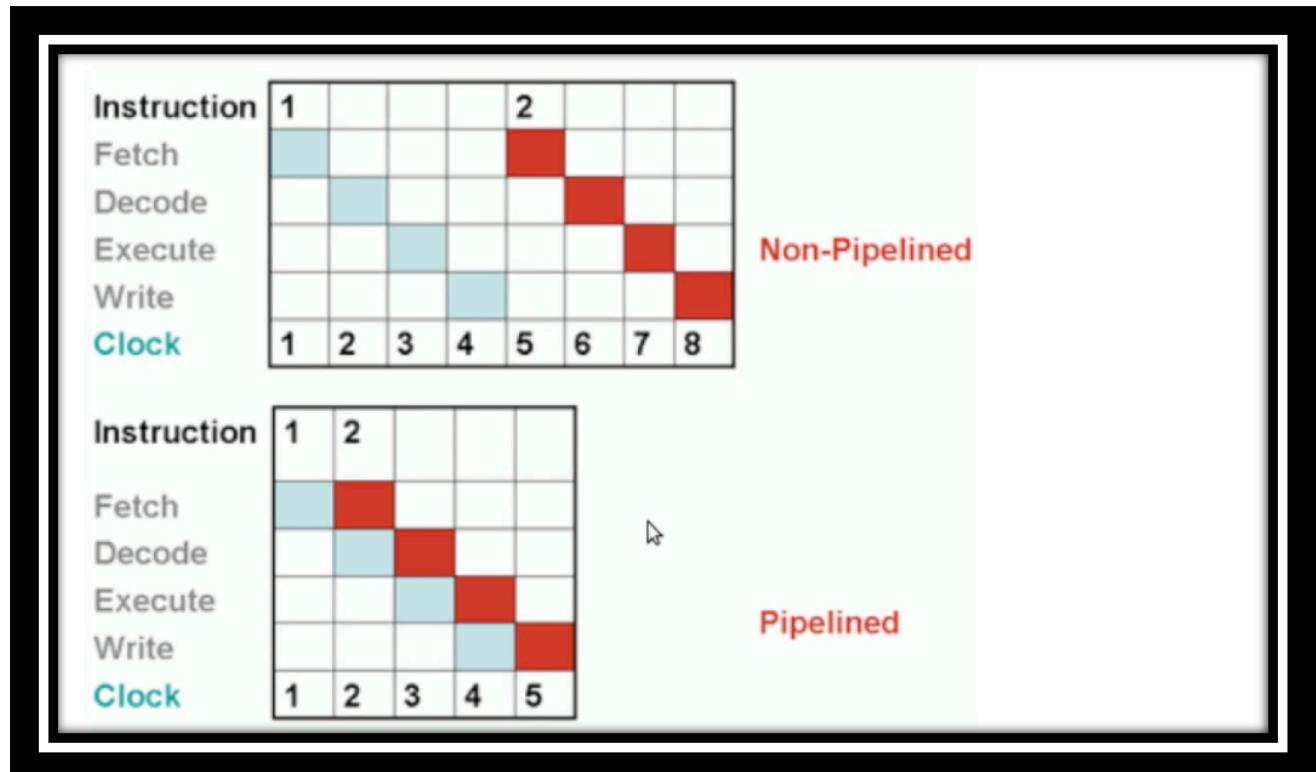
```

PIPELINING

En instruksjon kan deles inn i flere deler, stages, 4 er vanlig i Intel-CPUer. Eksempel på 4 stages:

1. Fetch (hent instruksjonen fra RAM)
2. Decode (hvilke knapper skal trykkes på i ALU og Datapath)
3. Execute (utfør instruksjonen)
4. Write (skriv resultater til RAM)

Tid spares ved at neste instruksjon starter før den første er ferdig.



En mikroarkitektur er hvordan et instruksjonssett er implementert i en CPU.

SUPERSCALAR ARKITEKTUR

Dette er en type datamaskinarkitektur som er designet for å øke ytelsen ved å utføre flere instruksjoner samtidig. I en slik arkitektur har datamaskinen flere aritmetiske logiske enheter (ALUer), registerbåndbredde og kontrollstrukturer som gjør at den kan utføre flere instruksjoner parallelt. I stedet for å utføre instruksjoner en etter en i rekkefølge, utfører den flere instruksjoner samtidig ved å bruke ALUene og andre ressurser samtidig.

Operasjonene kan utføres i en annen rekkefølge enn det sekvensielle programmet tilsier.

Slike arkitekturer er et viktig steg i utviklingen av moderne datamaskiner, og har bidratt til økt ytelse ved å redusere tiden det tar å utføre en oppgave. De er vanlige i mange moderne prosessorer, inkludert de som brukes i bærbare datamaskiner, stasjonære datamaskiner og mobiler.

LINUX UKE 7

PASSORD-KRYPTERING OG -CRACKING

All info om innlogging (brukernavn, brukerID, hjemmemappe, default shell) ligger i /etc/passwd. Her ligger ikke passordet. Vanligvis pleide passordet og ligge her også, men det har endret seg, så nå skriver man i stedet for passwd:

```
sudo grep brukernavn /etc/shadow
```

og hvis man ikke skriver sudo så får man en feilmelding om at man ikke har tilgang. Men output av den kommandoen kan gi:

```
rex:~$ sudo grep haugerud /etc/shadow
haugerud:$6$46M1Y1bk$H6vckKiUDTVYbNe/M9Wj4Nn/Ufk4GMvYK7uxbEmiusl9qoH7MzBRC6
fYetjZHdClWy5N2PZ2zcCt4wSB37oGu/:18292:0:99999:7:::
rex:~$
```

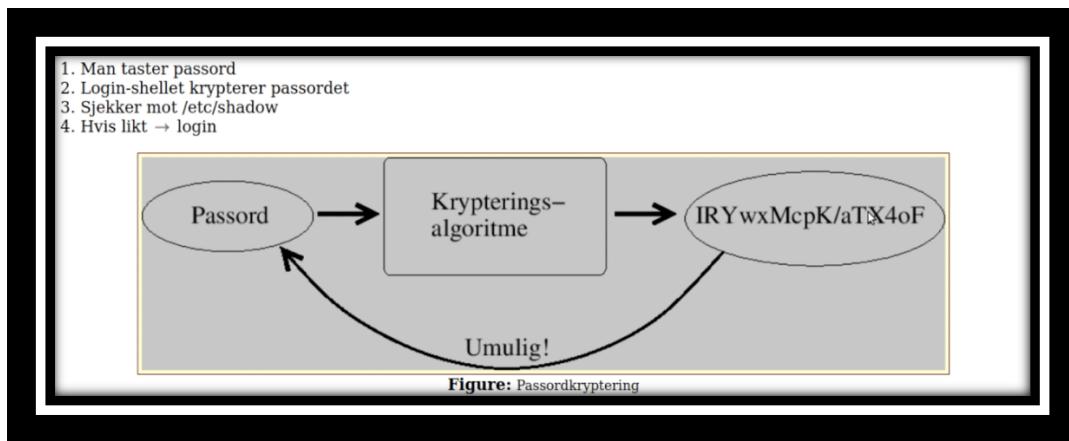
Som er en hash. Da man logger inn taster man passordet, og en algoritme lager dette passordet, hvis disse er like kan man logge inn. Mellom dollartegn nummer 2 og 3 er det et såkalt salt (en streng).

```
rex:~$ mkpasswd -m sha-512 -S 46M1Y1bk
```

Hvor -m er hvilken algoritme, sha-512 i vårt tilfelle, -S for saltet og selve saltet på slutten. Da kan man kjøre passordalgoritme.

| Første tegn | Algoritme |
|-------------|-----------------|
| To tegn | DES (13 totalt) |
| \$1\$ | md5 |
| \$5\$ | sha256 |
| \$6\$ | sha512 |

265 og 512 er hvor mange bit som er brukt i krypteringen. Jo flere bit → desto vanskeligere er det å knekke. I Haugerud sitt eksempel brukes sha512.



Hvis man har tilgang til shadow-filen på Linux, og dermed tilgang til hash-strengene, kan et crack-program kryptere alle ord og kombinasjoner av ord i en ordbok og sammenlikne med de krypterte passordene. Hvis ett av de riktige passordene velges, vil det avsløres ved at det gir en av hashene. Slike program kan teste hundretusener av passord per sekund, slik at passord fra ordbøker veldig raskt kan knekkes- jo lengre passordene er jo flere tegn som brukes i passordene, desto mer tidskrevende er det for passord-knekker program å teste ut alle mulige kombinasjoner.

BRUTE FORCE

Brute force er en metode for å løse et problem eller å finne en løsning ved å prøve alle mulige alternativer, ofte ved hjelp av en datamaskin, til en løsning er funnet. Det brukes ofte i sikkerhetsbransjen for å knekke passord eller andre typer krypterte data ved å prøve alle mulige kombinasjoner av tegn eller tall til en korrekt kombinasjon blir funnet.

Brute force-angrep kan være veldig tidskrevende og ressurskrevende, men de er også veldig effektive siden de tar hensyn til alle mulige alternativer. Imidlertid kan sikkerhetssystemer implementere forholdsregler, for eksempel å begrense antall påloggingsforsøk eller å aktivere en «sperre» etter en viss tidsperiode.

Kommandoen **find . -name “*.c” hvor .** vil si hvor vi vil lete, -name betyr at vi vil lete etter navn på noe som starter med noe og slutter på c.

Kommndoen **locate <filnavn>** vil i hele datamaskinen finne steder hvor oppgitt filnavn er med og ikke spesifikt det alene. Locate kommandoen er avhengig av at det er en oppdater locate database, og den kan være sjappere, men find leter fra scratch.

Kommndoen **find . -newermt** står for newer modified time og da man skriver en dato etter vil man få listet opp alt som er endret etter datoens oppgitt. Som eksempel:

```
find . -newermt «6 Feb 2021» -ls
```

-ls vil liste det resultatet i listeform.

```
haugerud@studssh:~$ find . -newermt "8 Feb 2021 22:00" -ls
9714      4 drwxr-xr-x  37 haugerud drift          4096 feb.  8 22:06 .
13593      4 -rw-----   1 haugerud drift          318 feb.  8 22:03 ./Xauthority
 728      4 -rw-----   1 haugerud drift          222 feb.  8 22:06 ./lessht
```

Linja under vil bety at det skal være nyere enn det til venstre for utropstegnet, men ikke nyere enn det til høyre for utropstegnet.

```
haugerud@studssh:~$ find . -newermt "8 Feb 2021 20:00" ! -newermt "8 Feb 2021 22:00" -ls
```

SED OG SORT

Sed er en kommando som kan brukes til å bytte forekomster av ord. Under ved den første / skriver det man ønsker å bytte ut og etter den andre / det man vil bytte det ut med. Hvis man rett bak den siste / skriver på en g vil det bytte ut alle forekomster og ikke bare det første som kan finnes.

```
haugerud@studssh:~$ echo test og test
test og test
haugerud@studssh:~$ echo test og test | sed s/test/fisk/
fisk og test
```

Sor kan brukes sortere filer linje for linje. Dersom man bare skriver sort rett ut vil den sortere fra første ord (h>k>s), men dersom man legger til en -2 k vil den da se på andre kolonner (i dette tilfellet bilene). I tredje kolonne er det numeriske verdier så det vil ikke holde med å kun skrive sort -k 3 <filnavn>. Da må vi ha med en -n for numerisk (sorterer synkende) slik:

```
sort -l 3 -n biler
```

Hvis man vil sortere stigende vil man da skrive -nr (numeric reverse)

```
haugerud@studssh:~$ cat biler
student bmw 500000
haugerud berlingo 150000
kyrre elbil 90000
haugerud@studssh:~$ sort biler
haugerud berlingo 150000
kyrre elbil 90000
student bmw 500000
haugerud@studssh:~$ sort -k 2 biler
haugerud berlingo 150000
student bmw 500000
kyrre elbil 90000
```

Hvis man vil sende over alt til en fil man kan skrive

Sort -k 3 -nr bilde ><filnavn> så får man det opp i en fil.

HEAD OG TAIL

Nyttige kommandoer for å kunne se starten og slutten av filer.

```
group100@os100:~$ head -n 5 /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
group100@os100:~$ head -n 1 /etc/passwd
root:x:0:0:root:/root:/bin/bash
group100@os100:~$ tail -n 5 /etc/passwd
messagebus:x:104:105::/nonexistent:/usr/sbin/nologin
```

En annen måte å gjøre det på med pipe:

```
group100@os100:~$ cat /etc/passwd | tail -n 3
group100:x:1000:998::/home/group100:/bin/bash
s123456:x:1001:1000:,,,:/home/s123456:/bin/bash
```

Under vil -t endre default skiller fra mellomrom til kolon slik at den kan skjønne hva som skiller hvilke kolonner i feks /etc/passwd.

```
group100@os100:~$ sort -t: -k 3 /etc/passwd | tail -n 5
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
```

Kommandoen under er brukt for å vise innholdet i «/var/log/auth.log» fila i ekte-tid. -f står for «follow» og den lar «tail» kommandoen overvåke filen og vise fram ny data samtidig som det skrives til fila. Dette er nyttig å se når man skal se logg filer samtidig som det oppdateres. Hvis det fortsetter å runne må man trykke kontroll + c for å stoppe det.

```
group100@os100:~$ sudo tail -f /var/log/auth.log
Feb 10 13:58:03 os100 sshd[34171]: Failed password for root from 159.75.120.229 port 37650 ssh2
Feb 10 13:58:04 os100 sshd[34171]: Received disconnect from 159.75.120.229 port 37650:11: Bye Bye
Feb 10 13:58:04 os100 sshd[34171]: Disconnected from authenticating user root 159.75.120.229 port 37650
Feb 10 13:58:16 os100 sshd[34173]: pam_unix(sshd:auth): authentication failure; logname= uid=0 e
Feb 10 13:58:18 os100 sshd[34173]: Failed password for root from 159.75.120.229 port 37650 ssh2
Feb 10 13:58:19 os100 sshd[34173]: Received disconnect from 159.75.120.229 port 37650:11: Bye Bye
Feb 10 13:58:19 os100 sshd[34173]: Disconnected from authenticating user root 159.75.120.229 port 37650
Feb 10 13:58:26 os100 sudo: group100 : TTY=pts/1 ; PWD=/home/group100 ; USER=root ; COMMAND=/usr
Feb 10 13:58:26 os100 sudo: pam_env(sudo:session): Unable to open env file: /etc/default/locale:
```

CUT

Kommandoen «cut» lar oss ta ut spesifikke deler eller kolonner fra en fil eller input. Det fungerer ved å dele hver linje i deler, også velger vi hva som skal tas ut. I 2 linje er det mulig å se at vi må spesifisere hvordan vi skal skille ulike delene i en streng. -d« » med et mellomrom imellom anførselstegnene spesifiserer dette.

```
group100@os100:~$ echo a b c | cut -f 2
a b c
group100@os100:~$ echo a b c | cut -d" " -f 2
b
```

Man kan få cut til å løpe gjennom tabeller og hente ut det vi trenger: her henter vi de tre første linjene i en fil. Også kutter vi feltene med : og henter første kolonne, og i den andre kommandolinjen ber vi cut bare hente første bokstav med -c (character = bokstav). etter cut -c kommer tallet en som betyr første bokstav. man kan også skrive 1-3 som vil hente ut bokstavene 1-3. man kan også skrive -3 som betyr alle tall fra starten av til tredje.

```
group100@os100:~$ head -n 3 /etc/passwd | cut -d: -f 1
root
daemon
bin
group100@os100:~$ head -n 3 /etc/passwd | cut -d: -f 1 | cut -c 1
r
d
b
```

INPUT FRA BRUKER TIL SCRIPT

Til venstre er det en kode som tar input og legger til på slutten i «Du svarte». Til høyre kan man se hvordan det kjøres. I koden vil echo -n be om input på en vanligere måte.

| | |
|-------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>F10 key ==> File Edit Search Buffers Wi #!/bin/bash echo -n "Svar: " read svar echo "DU svarte \$svar"</pre> | <pre>haugerud@lap:~/Documents\$./input.sh Svar: svaret DU svarte svaret haugerud@lap:~/Documents\$./input.sh Svar: HALLO DU svarte HALLO haugerud@lap:~/Documents\$</pre> |
|-------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

LESE FIL LINJEVIS MED WHILE OG TREKKE UT KOLONNER

| |
|-----------------------------------------------------------------------------------------------|
| <pre>#!/bin/bash while read linje do ((i++)) echo "Linje nr \$i: \$linje" done</pre> |
|-----------------------------------------------------------------------------------------------|

Denne koden kan kjøres med:

cat fil.txt | ./while.sh

./while.sh fil.txt

./while < fil.txt

Alternativ: etter done i koden legge til «< fil.txt» også kjøre shellelet

Alternativ: over while i koden skrive cat «fil.txt»

PASSORDFIL LINJEVIS MED WHILE OG NOEN KOLONNER

IFS: endrer fra default skille mellomrom til kolon som angitt under. Istedet for å skrive «while read linje» velger vi variabler, og de vi vil ha skrevet ut er gitt under do med echo.

```
#!/bin/bash

IFS=:
while read brnavn x uid gid navn home shell
do
    echo "Brukernavn $brnavn uid $uid"
done < /etc/passwd
```

PS AUX

ARRAY I BASH

Under vises hvordan man angir verdier til indeksplasser i et array. Merk under hvordan man får ut



```

linux:~$ tall[1]=en
linux:~$ tall[2]=to
linux:~$ tall[3]=tre
linux:~$ echo ${tall[1]}
null[1]
linux:~$ echo ${tall[2]}
null[2]
linux:~$ echo ${tall[2]}
to
linux:~$ echo ${tall[0]}
null
linux:~$ echo ${tall[1]}
en
linux:~$ echo ${tall[@]}
null en to tre
linux:~$ echo ${!tall[@]}
0 1 2 3

```

Skriver ut alle verdier

Skriver ut alle verdier

For løkke som skriver ut alle verdier med indeksplass.

```

linux:~$ for i in ${!tall[@]}
> do
> echo "element $i er ${tall[$i]}"
> done
element 0 er null
element 1 er en
element 2 er to
element 3 er tre

```

Som vist under vil # angi antall elementer i et array.

```

linux:~$ echo ${#array[@]}
1

```

VANLIG PROBLEM MED PIPE TIL WHILE OG READ

Her er problemet at når man vil pipe noe fra en prosess, altså output fra en prosess, intil en while konstruksjon, så vil while-konstruksjonen lage et eget subshell slik at alt som skjer i det, blir der. Så etterpå vil man ikke ha de datakonstruksjonene som ble lagd inne, i tilfellet under et array. Så da har ikke arrayet utenfor en verdi. Men en løsning kan sees ved å lage en temporary fil.

```
#!/bin/bash

ps aux | awk '{print $2}' |
while read pid
do
    ((i++))
    pidArr[$i]=$pid
    echo "$i $pid"
done

echo "Antall elementer ${#pidArr[@]}"
```

Som vist under er løsningen:

```
#!/bin/bash

ps aux | awk '{print $2}' > tmp.txt
while read pid
do
    ((i++))
    pidArr[$i]=$pid
done < tmp.txt

echo "Antall elementer ${#pidArr[@]}"
```

På bildet under er det enda en alternativ løsning i stedet for å lage og sette innholdet i en alternativ fil. Etter done kan man se <<() og hele kommandolinjen inni kan vi late som er en fil.

```
#!/bin/bash

while read pid
do
    ((i++))
    pidArr[$i]=$pid
done < <(ps aux | awk '{print $2}'[])
echo "Antall elementer ${#pidArr[@]}"
```

ASSOSIATIVE ARRAY

Assosiativ array er en type datastruktur som lagrer nøkkel-verdi-par. I en slik type array kan du bruke unike nøkler til å indeksere og hente verdier, i stedet for å bruke heltallindekser som i tradisjonelle arrays. Dette gjør det enklere å finne og håndtere data, da nøklene kan være beskrivende tekstverdier.

Disse må mann deklarer på en spesiell måte:

```
declare -A <mann>
```

```
linux:~$ for i in "${!mann[@]}"; do echo "Mannen til $i er ${mann[$i]}"
Mannen til kari er per
Mannen til eva er adam
Mannen til Gunn Kari er Per Olav
linux:~$
```