**$ ssh aleksawo@itstud**

**scp -r aleksawo@itstud.hiof.no:htdocs/oblig? C:\Users\alexg\Documents\GitHub\innf\_os**

Kommandolinjen i Linux er et tegnbasert grensesnitt som gjør at du kan kjøre både OS-kommandoer og andre programmer på maskinen. Når du bruker kommandolinjen kommuniserer du med et program som kalles et **shell**:

* Shellet tilbyr et "Command Line Interface" (CLI). CLI lar deg kontrollere datamaskinen ved å bruke kommandoer som legges inn bare med tastaturet, i stedet for å navigere i et grafisk brukergrensesnitt (GUI) med en mus eller annen pekeredskap.
* Shellet lar deg kjøre andre programmer eller script, og er også svært nyttig for å administrere datamaskinen når du vet hvilke kommandoer du skal bruke.
* De fleste Linux-distribusjoner bruker programmet **bash** (Bourne-Again Shell) som standard shell.

Merk at shellet ikke må forveksles med et annet program som kalles en **terminalemulator**. Terminalemulatoren etterligner/emulerer gammeldagse og tegnbaserte dataterminaler, som tidligere ble brukt som "arbeidsstasjoner" for brukere som delte på én stor, sentral datamaskin (mainframe). Emulatoren tilbyr bare et GUI-terminalvindu som "rammer inn" shellets kommandolinjegrensesnitt.

Et bilde som inneholder tekst, Elektronisk anordning, elektronikk, Kontorutstyr

Automatisk generert beskrivelseBildet viser en typisk terminal fra slutten av 70-årene, med skjerm og tastatur. Dette var **ikke** en datamaskin, men en tegnbasert (ikke-grafisk) I/O-enhet der man kunne kommunisere med den sentrale datamaskinen som den var koblet opp mot:

Et bilde som inneholder tekst, skjermbilde, programvare, Multimedieprogramvare

Automatisk generert beskrivelseSkjermbildet nedenfor er tatt fra desktopen til et moderne Linux-system med GUI, med en aktiv terminalemulator (det svarte vinduet). Det tegnbaserte shellet kjører linje-for-linje interaktiv kommunikasjon med brukeren inne i emulatoren, på samme måte som i de gamle terminalene:

Når du starter bash inne i et terminalvindu, er det første du vil se en **ledetekst** (aka "command prompt"), som betyr at shellet venter på at du skal skrive inn en kommando. Ledeteksten kan tilpasses og settes slik du vil ha den, men ser typisk slik ut: **username@hostname:currentdirectory$**

Når jeg logger på serveren it.stud.hiof.no, får jeg denne ledeteksten: **aleksawo@itstud:~$**

Symbolet '~' (tilde) i ledeteksten betyr at stående katalog er brukerens hjemmeområde (mer om kataloger og slikt kommer nedenfor i dette dokumentet). Kommandoene du gir til maskinen skrives etter "prompt-tegnet" '$' og avsluttes med en return/linjeskift. Shellet tolker da kommandoen du har gitt og setter i gang et program som utfører kommandoen, hvis den er korrekt. Hvis ikke, gis det en feilmelding.

**$ who**

Du vil da få se en liste med informasjon om påloggede brukere, bl.a. brukernavn og tidspunkt de logget på systemet.

Merk at alle Linux-kommandoer har såkalte **opsjoner** og **argumenter** som kan endre måten de fungerer på. Disse angis på kommandolinjen etter selve kommandonavnet, oftest som en eller to bindestreker etterfulgt av et tegn eller et ord.

Kommandoen *who* har mange opsjoner, blant annet:

* *-q* : viser alle påloggende brukernavn og antall brukere pålogget
* *-b* : viser tidspunktet for siste oppstart av systemet
* *--version:* skriver ut versjonsinformasjon for programmet who

**$ who -q**

**$ who -b**

**$ who --version**

For se alle mulige opsjoner som kommandoen *who* kan ha (og også mer informasjon om hvordan den virker og kan brukes), kan du gi følgende kommando som viser deg **manualsiden** for *who*:

**$ man who**

Kommandoen *man* brukes for å få hjelp og informasjon om Linux-kommandoer. Du kan "bla gjennom" hjelpesider fra *man* ved å trykke *SPACE*. *man* (og alle andre Linux-kommandoer) kan avbrytes ved å trykke *Ctrl-C* (som sender et såkalt interrupt som stopper det kjørende programmet). Mer om manualsider finner du nedenfor.

**$ cal**

Som standard skriver denne kommandoen ut kalenderen for nåværende måned. Hvis du ønsker å skrive ut en kalender for et bestemt år og måned, bruk denne syntaksen: *cal month year*.

Bruk ***uptime*** til å finne ut hvor lenge det er siden maskinen du er logget på var ute av drift.

Bruk ***finger*** til å finne ut mere om brukeren

Send en melding med ***write*** til en annen bruker på systemet.

Lær deg å bruke den enkle, selvforklarende teksteditoren ***nano***.

Finn ut hva kommandoen ***apropos*** kan brukes til:

I Linux er apropos et kommandoverktøy som brukes til å søke etter manualsidetitler og beskrivelser som inneholder et bestemt nøkkelord. Det er nyttig når du vet hva en kommando gjør eller hva du leter etter, men ikke husker den nøyaktige kommandoen.

**Hvordan bruke apropos:**

* Hvis du for eksempel leter etter kommandoer relatert til "network", kan du skrive:

Bash

Kommandoen ***which*** forteller hvor på disken de eksekverbare programmene for hver Linux-kommando er lagret. Finn ut hvor Java Virtual Machine ("**java**") ligger.

Programmet ***cat*** kan brukes til å se på innholdet av tekstfiler. Kjør kommandoen ***cat /etc/passwd*** for å se den "rå" informasjonen om alle brukerne på systemet.

Les brukerinformasjonen fra systemet en "side" om gangen ved å gi kommandoen ***cat /etc/passwd | more***. bruk også ***man*** for se hva som utføres av kommandoen *more*

Tegnet |, kjent som "pipe" i en kommandolinje i Linux (og andre Unix-lignende systemer), brukes til å sende utdata (output) fra én kommando som input til en annen kommando. Dette gjør det mulig å kjede flere kommandoer sammen for å utføre komplekse oppgaver i en enkelt kommandolinje.

**Hvordan | fungerer:**

* **Eksempel 1:** La oss si at du vil se innholdet i en katalog og deretter telle hvor mange filer som er i den katalogen. Dette kan du gjøre ved å kombinere ls-kommandoen (som lister opp filer) med wc -l (som teller antall linjer).

bash

I Linux brukes kommandoen *echo* til å skrive ut en linje med tekst på skjermen

**echo Cashmere Cat**

Linux-shell har en rekke innebygde **systemvariabler** som lagrer informasjon om bl.a. oppsett av systemet, opsjoner som styrer hvordan shellet virker, informasjon om bruker etc. Variablene lagrer tekststrenger og identifiseres med variabelnavn. Du kan se på innholdet i systemvariable i *bash* ved å sette '$' (et dollartegn) foran variabelnavnet.

**$ echo Hei $USER**

**$ echo $HOSTNAME**

**$ echo $TERM**

**$ echo $PATH**

Linux tilbyr også kommandoer for å sjekke maskinvare- og systeminformasjon. For eksempel vil kommandoen *lscpu* gi informasjon om prosessorene, men *lspci* lister opp alle PCI-bussene (PCI: Peripheral Component Interconnect) og alle enhetene som er koblet til en buss.

**$ lscpu**

**$ lspci**

Her er noen flere kommandoer som gir systeminformasjon. Prøv å forstå hva disse kommandoene gjør, ved å lese manualsider og studere utskriften:

**$ cat /proc/version**

Viser informasjon om kjernen (kernel) som kjører på systemet. Denne kommandoen leser innholdet av filen /proc/version, som inneholder versjonsinformasjon om Linux-kjernen, kompilatorbruk, og når kjernen ble kompilert.

**$ cat /proc/partitions**

Viser en liste over alle partisjonene som er registrert på systemet. Denne kommandoen leser innholdet av filen /proc/partitions, som gir informasjon om diskpartisjoner på systemet, inkludert størrelsen på hver partisjon og deres enhetsnavn.

**$ cat /proc/cpuinfo**

Viser detaljer om systemets CPU(er). Denne kommandoen leser innholdet av filen /proc/cpuinfo, som inneholder informasjon om CPU-arkitekturen, som prosessormodell, antall kjerner, klokkefrekvens, cache-størrelse, og andre CPU-relaterte detaljer. Utdataene kan være omfattende og typisk inneholde detaljer for hver prosessor (kjerne) i systemet.

**$ df**

Viser diskbruk for alle monterte filsystemer. df står for "disk free", og kommandoen viser hvor mye plass som er brukt og tilgjengelig på hvert montert filsystem, inkludert prosentandelen av brukte ressurser.

**$ lsblk**

Viser en liste over alle blokkenheter (som disker og deres partisjoner) på systemet. lsblk står for "list block devices". Den gir en hierarkisk visning av disker og tilhørende partisjoner, sammen med deres størrelse og monteringspunkter. Dette kan hjelpe deg å forstå diskstrukturen på systemet.

**$ free -m**

Viser informasjon om systemets minnebruk, inkludert RAM og bytteområde (swap). Kommandoen free viser hvor mye av det fysiske minnet (RAM) og swap-plassen som er brukt, tilgjengelig, og cachet. Flagget -m viser informasjonen i megabyte.

Etter at du har logget på et Linux-system, vil din arbeidskatalog (eller "stående katalog"/"current directory") oftest være hjemmeområdet ditt på disken, der du kan lagre dine egne filer.

Lag nå en katalog med *mkdir*

**$ mkdir katlognavn**

Bytt deretter stående katalog til å være denne nye katalogen ved å bruke kommandoen *cd* (change directory):

**$ cd katlognavn**

Deretter oppretter du en ny fil og redigerer den med den enkle, selvdokumenterende editoren *nano*:

**$ nano nyfil.txt**

Du vil da gå inn i *nano*-redigeringsgrensesnittet. Skriv inn litt tekst og trykk ***Ctrl-X*** for å lagre filen og avslutte *nano*. Tilbake på kommandolinjen, bruk *ls* til å liste ut navnene på alle filer i stående katalog:

**$ ls**

Filen ***nyfil.txt*** vil da skrives ut som eneste fil i katalogen ***katalognavn***.

Opprett noen flere tekstfiler i katalogen ***katalognavn***, f.eks. ved å bruke editoren ***nano*** eller kommandoen ***touch***, eller ved å laste ned noen filer fra internett med bruk av kommandoen ***wget***. Gjøre deretter følgende for å teste litt håndtering av filer og kataloger:

***ls***: Skriver en liste med informasjon om filene i en katalog

***cp***: Kopierer filer og kataloger

***mv***: Flytter eller bytter navn på filer og kataloger

***rm***: Sletter filer og kataloger (vær forsiktig med denne...)

***chmod***: Setter tilgangskontroll for filer (mer om dette senere i kurset)

Only the owner of a file can use chmod to change the permissions of a file. The options of chmod are as follows

|  |  |
| --- | --- |
| **Symbol** | **Meaning** |
| u | user |
| g | group |
| o | other |
| a | all |
| r | read |
| w | write (and delete) |
| x | execute (and access directory) |
| + | add permission |
| - | take away permission |

***man mv*** for å se hvordan ***mv*** virker

Se på innholdet av tekstfilene dine med ***cat*, *more* og *less***.

// ikke ferdig med 1.6

Opprett en ny katalog som heter ***katalog2*** i katalogen ***katalognavn***. Lag flere nivåer av forskjellige underkataloger inne i***katalog2***. Bruke kommandoen ***cd*** til å flytte deg rundt i denne katalogstrukturen du har laget. Finn deretter ut hvordan du kan fjerne hele "katalogtreet" som starter i ***katalog2*** (og alle filene som evt. ligger der), med bare én enkel Linux-kommando.

**$ ls -a**

Kommandoen ls -a viser alt innholdet i en katalog. For f.eks. å vise informasjon om bl.a. alle filenes størrelser (i et lettlest "human-readable" format), bruk følgende kommando:

**$ ls -a -l -h**

eller "klump" de tre opsjonene sammen på denne måten:

**$ ls -alh**

I oversikten over filene som ***ls -a*** skriver ut, vil du først se en linje for en fil som heter bare "." (punktum) og deretter en som heter ".." (to punktum). Dette er to spesialfiler som finnes i alle Linux-kataloger. "." refererer til stående katalog (working directory), mens ".." refererer til katalogen på nivået over; forelderkatalogen (parent directory).

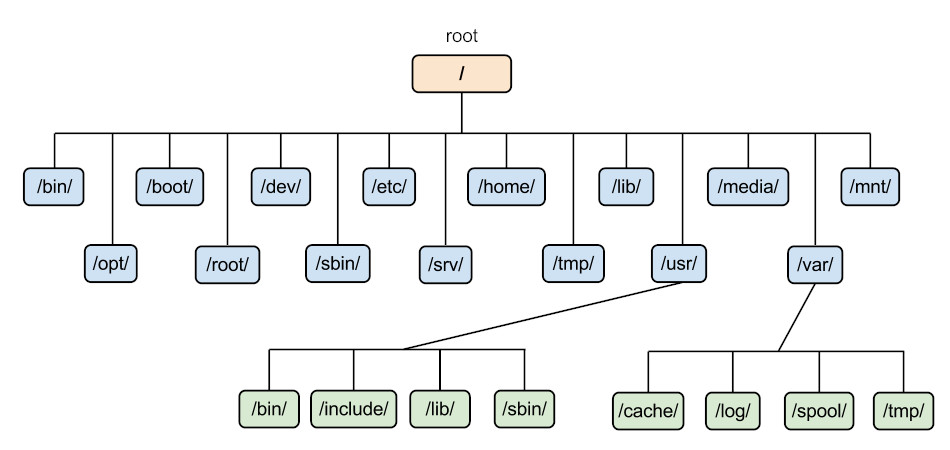
**$ cd .**

forandres ikke stående katalog. Kommandoen:

**$ cd ..**

vil sette stående katalog til å være forelderkatalogen. Prøv disse to kommandoene på katalogene som du har laget i ditt eget hjemmeområde.

Alle katalogene i et Linux-system er ordnet i et hierarkisk **tre** med mange nivåer. Hver katalog ligger inne i en annen (forelder)katalog som er på nivået ovenfor. Helt øverst i treet ligger en katalog som ikke har noen forelderkatalog, dette er **rotkatalogen** (root) som betegnes bare med en '/'. Figuren nedenfor viser et (relativt) standard oppsett av systemkataloger på de øverste nivåene i katalogtreet i Linux:



**Hjemmekatalogen** (home directory) til en bruker er den katalogen som settes til å være stående katalog etter pålogging. Som oftest er dette en katalog som heter det samme som brukernavnet og ligger inne i systemkatalogen */home*. I shellet det et eget, reservert tegn for å angi hjemmekatalogen i kommandoer som har kataloger som parametre: '~' (tilde).

**$ cd ~**

vil alltid sette stående katalog til å være hjemmeområdet ditt. Kommandoen ***cd*** uten et katalognavn som parameter gjør det samme som ***cd ~***.

En søkesti i Linux er en angivelse av hvor i filsystemet/katalogtreet du finner en bestemt katalog. Søkestien er en liste av katalognavn adskilt med spesialtegnet '/' (skråstrek/slash). Kommandoen *pwd* (print working directory) vil skrive ut søkestien til stående katalog.

Prøv å gi kommandoen *pwd* etter pålogging. Du vil da få skrevet ut søkestien til hjemmekatalogen din.

$ pwd

/home/aleksawo

Det finnes to ulike måter å angi søkestier på:

Absolutt søkesti:

Starter enten på toppen av filsystemet i rotkatalogen som angis med tegnet '**/**', eller i hjemmekatalogen som angis med tegnet '**~**'. Deretter kommer listen av katalognavn adskilt med '**/**' som går helt frem til katalogen der filen vi skal bruke befinner seg.

Relativ søkesti:

Stien starter med et katalognavn som ligger inne i stående katalog (working directory). Deretter kommer listen av katalognavn adskilt med '**/**'.

Prøv å bruke en absolutt søkesti for å bytte stående katalog -- du skal allerede ha laget katalogen ***katalognavn*** i din hjemmekatalog -- ved å gi kommandoen

**$ cd /home/aleksawo/Intro2OS**

Du skal nå være i ***Intro2OS***-katalogen din. For å bekrefte dette, skriv inn:

**$ pwd**

For å gå tilbake til hjemmekatalogen, skriv inn:

**$ cd**

Bytt nå til underkatalogen igjen ved å bruke en relativ søkesti:

**$ cd Intro2OS**

Bruk ***pwd*** for å bekrefte at du nå er i underkatalogen.

I eksemplet ovenfor er det ingen forskjell mellom disse to måtene å komme til en katalog på. Anta imidlertid at du var i en annen brukers hjemmekatalog. I det tilfellet ville kommandoen ***cd Intro2OS*** ta deg til hans eller hennes ***Intro2OS***-katalog. Men hvis du brukte den absolutte søkestien med brukernavnet ditt, ville du gå til din egen katalog.

Du kan bruke relative og absolutte søkestier for å beskrive filer og kataloger i mange kommandoer. For eksempel, hvis du er i hjemmekatalogen din, bør følgende to kommandoer gi samme resultat (en liste av filene i rotkatalogen på Linux-systemet):

**$ ls /**

**$ ls ../..**

Den første kommandoen bruker en absolutt søkesti (den starter med **/**), så den vil alltid returnere samme resultat uansett hvor du er. Hvis du er i katalogen ***/home/brukernavn*, vil ../..** referere til katalogen to nivåer over hjemmekatalogen (forelder til forelderkatalogen). Hvis du var i ***/home/brukernavn/Intro2OS*** og skrev ***ls ../..***, ville du se filene i katalogen ***/home*** i stedet.

Linux har to standard editorer, ***vi* og *emacs***. Det finnes også enklere editorer som f.eks. ***nano* og *joe***. Vi anbefaler **sterkt** at dere lærer å bruke ***emacs*,** som er et svært bra verktøy for det aller meste og spesielt for å skrive programkode i Linux.

stopp den igjen med ***:q***

Start (den fantastiske!) editoren ***emacs*** i terminal-modus med kommandoen ***emacs -nw*.**

Gå gjennom hele **nybegynneropplæringen** i ***emacs***, som du får frem ved å trykke ***Ctrl-h t*.**

Prøv psykoanalyse i **emacs** med kommandoen ***Esc-x doctor***

**Ctrl** **x** and then press **Ctrl c**. to exit and get save prompt

**$ iconv -t UNICODE -o tmp.txt file.txt**

**$ mv tmp.txt file.txt**

**iconv -t UNICODE -o tmp.txt file.txt && mv tmp.txt file.txt**

Det interaktive shellet ***bash*** har en snarvei-funksjon som lar deg skrive inn bare de første tegnene i en kommando. Trykk deretter på tabulatortasten og shellet vil skrive ferdig resten av kommandoen for deg (tab completion). Ved å trykke to ganger på tabulator vises alle kommandoer som matcher det du hittil har skrevet.

Du kan bruke opp- og ned-piltastene til å bla gjennom en liste med nylig innskrevne kommandoer. ***Ctrl-R*** vil søke gjennom kommandohistorikken din

Manualsidene er dine beste venner i Linux. De gir informasjon om bl.a. alle kommandoer, applikasjoner, systemkall, programmeringsspråk osv. Det krever litt trening/tilvenning for å få godt utbytte av manualsidene -- de er presise og korrekte, men også detaljerte, tekniske og omfattende.

Et eksempel: For å vise manualsiden til kommandoen ***ping***, som brukes for å sjekke hastighet og kvalitet på forbindelsen til andre maskiner på nettet, skriv inn *man ping* i shellet. Du vil da se noe slikt:

PING(8) iputils PING(8)

**NAME**

ping - send ICMP ECHO\_REQUEST to network hosts

**SYNOPSIS**

**ping** [**-aAbBdDfhLnOqrRUvV46**] [**-c** count] [**-F** flowlabel] [**-i** interval]

[**-I** interface] [**-l** preload] [**-m** mark] [**-M** pmtudisc\_option]

[**-N** nodeinfo\_option] [**-w** deadline] [**-W** timeout] [**-p** pattern]

[**-Q** tos] [**-s** packetsize] [**-S** sndbuf] [**-t** ttl]

[**-T** timestamp option] [hop...] {destination}

**DESCRIPTION**

**ping** uses the ICMP protocol mandatory ECHO\_REQUEST datagram to elicit

an ICMP ECHO\_RESPONSE from a host or gateway. ECHO\_REQUEST datagrams

("pings") have an IP and ICMP header, followed by a struct timeval and

then an arbitrary number of “pad” bytes used to fill out the packet [...]

**Manualsidene er delt inn i flere seksjoner**. F.eks. betyr **PING(8)** i eksemplet ovenfor at ***ping* sine manualsider finnes i seksjon 8**. For en beskrivelse av hva som ligger i de forskjellige manualseksjonene, gi kommandoen:

**$ man man**

**Noen viktige seksjoner: Shellkommandoer og en del systemprogrammer er i seksjon 1, systemkall i programmeringsspråket C er i seksjon 2, biblioteksfunksjoner er i seksjon 3, administrative systemverktøy er i seksjon 8.**

Du kan angi hvilken seksjon du vil se manualsider fra. For eksempel vil ***man write* vise manualsidene i seksjon 1 for shellkommandoen *write* som sender meldinger til andre brukere. For å se dokumentasjon for C-systemkallet *write()*, må du gi kommandoen *man 2 write*.**

I tillegg til manualsider tilbyr Linux også såkalte "**infosider**" for en del kommandoer og programvare. Infosider leses med kommandoen ***info***. De er ofte bedre oppdatert enn manualsider og kan være mer lettleste. Et eksempel er infosidene for kommandoen ***sort***, som inneholder mer informasjon om hva dette programmet gjør enn bare en beskrivelse av syntaks og funksjonalitet. Prøv kommandoen:

**$ info sort**

Andre nyttige hjelpeverktøy i Linux er ***apropos* og *whatis***. Prøv:

**$ man apropos**

**$ man whatis**

De fleste Linux-kommandoene som du bruker er egne, ferdig kompilerte C-programmer som ligger lagret et sted på disken. Når du gir en kommando til *bash*, vil shellet lete etter et program med samme navn som kommandoen. Katalogene som det letes i ligger lagret som en liste i shellvaraiabelen ***PATH***. For å se denne listen, gi kommandoen:

**$ echo $PATH**

Hvis ***bash*** finner kommandoprogrammet, lastes dette inn i RAM og startes. Hvis ikke, får du en feilmelding. For å finne ut hvor på disken en Linux-kommando ligger lagret, kan du bruke kommandoen ***which*.**

Shellet ***bash*** har i tillegg en del kommandoer som er **"innebygget",** dvs. at de **ikke** ligger på disk men er en del av shellprogrammet selv. Slike kommandoer kalles **"bash-builtins".** Du kan få en oversikt over dem ved å gi denne kommandoen:

**$ man builtins**

Builtins er **ikke** dokumentert i manualsystemet. I stedet tilbyr ***bash*** kommandoen ***help*** som gir kortfattet dokumentasjon om **builtins**. Et eksempel er builtin-kommandoen ***alias***, som gjør at du kan definere egne navn/shortcuts for f.eks. kommandoer du bruker ofte. Hvis du gir kommandoen *man alias* vil du kanskje (avhengig av versjonen av Linux so brukes) se at det ikke finnes noen manualside for ***alias***. Prøv i stedet:

**$ help alias**

Sjekk også om det finnes **manualside for kommandoen *cd*. Eller må du bruke *help cd*?**

Merk at en del kommandoer finnes **både** som en **builtin i *bash*** og som en vanlig Linux-kommando på disk, f.eks. kommandoen ***echo***. Det finnes også C-systemkall som har samme navn som **builtins, f.eks. *read***. Det betyr at det kan finnes manualsider for en kommando eller et systemkall med samme navn som en **builtin.**

Mange kommandoer i Linux er tekstbaserte "filterprogrammer", som:

* Leser en strøm av tekst/tegn, linje for linje.
* Gjør noe med ("filtrerer") tekstlinjene.
* Skriver deretter ut filtrert tekst linje for linje.

Et av filterprogrammene i Linux er kommandoen ***tr*: "translate or delete characters". *tr*** leser tekstlinjer og bytter ut eller fjerner tegn før linjene skrives ut igjen. Her er noen eksempler på bruk av ***tr*** til å gjøre om tekst, test gjerne disse med input fra tastaturet (se også *man* ***tr***):

* Små til store (engelske) bokstaver:

**$ tr a-z A-Z**

* Slett alle siffer i input:

**$ tr -d 0-9**

* Erstatt alle siffer med tegnet X:

**$ tr 0-9 X**

* Erstatt alle ikke-siffer(!) med X:

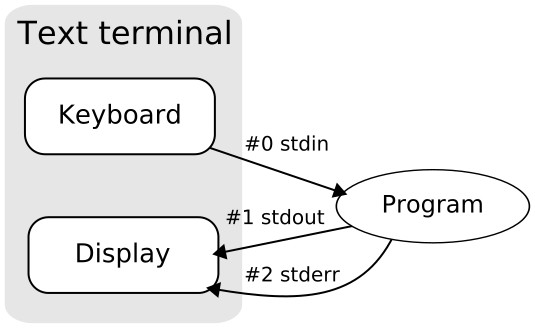
**$ tr -c 0-9 X**

* Erstatt alle påfølgende siffer med én X:

**$ tr -s 0-9 X**

Alle filterprogrammer bruker i utgangspunktet tre input/output enheter:

* Input leses fra standard innenhet: ***stdin*. *stdin*** er i utgangspunktet (**default)** satt til å være tastaturet
* Output skrives til standard utenhet: ***stdout*. *stdout*** er i utgangspunktet **terminalvinduet (skjermen).**
* Feilmeldinger skrives til standard feilenhet: ***stderr*. *stderr*** er i utgangspunktet også **terminalvinduet.**

Figuren nedenfor viser sammenhengen mellom filterprogram, terminalvindu/tastatur og de tre standard I/O-enhetene:

Shellet i Linux tilbyr **redirigering** (redirection) av input og output for filterprogrammer. Vi kan sette ***stdin* og *stdout*** til å være vanlige filer i stedet for tastatur og skjerm. Dette betyr at det er lettvint (og elegant) å få Linux-program og -kommandoer til å lese fra og skrive til filer.

Spesialtegnene **'<' og '>'** brukes på kommandolinjen, sammen med et filnavn, for å gjøre **redirigering av I/O:**

* Les input fra fil i stedet for fra tastatur:

**< filnavn**

* Skriv output til fil i stedet for til skjerm, overskriv hvis filen finnes fra før:

**> filnavn**

* Skriv output til fil, men legg til på slutten av filen **("append")** hvis filen finnes fra før:

**>> filnavn**

For å forstå hvordan **redirigering** virker, kan vi se på filterprogrammet ***cat***. Dette er en kommando som bare kopierer **input (fra *stdin*) til output (*stdout*). *cat*** kan også ta filnavn som **parametre**, den vil da skrive innholdet av filen ut til skjermen.

Kjør kommandoen ***cat*** og skriv inn noen linjer med tekst. Du vil da se at linjene bare **kopieres tilbake (som et ekko). *cat* kan avsluttes ved å trykke *Ctrl-D* (som er end-of-file tegnet i Linux) eller *Ctrl-C* (interrupt).**

Hvis du kombinerer ***cat* med redirigering av I/O**, kan du få denne enkle kommandoen til å gjøre andre nyttige ting:

* Viser innholdet av filen *fil\_1* på skjermen:

**$ cat < fil\_1**

* Lager en kopi av *fil\_1* og lagrer denne i *fil\_2*:

**$ cat < fil\_1 > fil\_2**

* Legger til innholdet av *fil\_1* på slutten av *fil\_2*:

**$ cat < fil\_1 >> fil\_2**

* Det som tastes inn blir lagret i *fil\_1*:

**$ cat > fil\_1**

Lag noen filer og prøv eksemplene ovenfor selv, for å se at det virker som det skal. Merk at i det siste eksemplet har ***cat*** faktisk blitt til en enkel teksteditor!

Noen flere eksempler på redirigering av I/O (les manualsider for å forstå hvordan kommandoene her virker):

* Lagrer en liste med navn på alle filene i stående katalog på en egen fil:

**$ ls > liste-fil.txt**

* Legger til nåværende tidspunkt og en liste med alle påloggede brukere på den samme filen:

**$ date >> liste-fil.txt  
$ who >> liste-fil.txt**

* Legger til de 20 siste linjene på en fil på slutten av en annen fil:

**$ tail -20 < fil\_1.txt >> fil\_2.txt**

We should note here that a directory is merely a special type of file. So the rules and conventions for naming files apply also to directories.

In naming files, characters with special meanings such as / \* & % , should be avoided. Also, avoid using spaces within names. The safest way to name a file is to use only alphanumeric characters, that is, letters and numbers, together with \_ (underscore) and . (dot).

|  |  |
| --- | --- |
| Good filenames | Bad filenames |
| project.txt | project |
| my\_big\_program.c | my big program.c |
| fred\_dave.doc | fred & dave.doc |

File names conventionally start with a lower-case letter, and may end with a dot followed by a group of letters indicating the contents of the file. For example, all files consisting of C code may be named with the ending .c, for example, prog1.c . Then in order to list all files containing C code in your home directory, you need only type ls \*.c in that directory.

Vi kan også redirigere I/O slik at output fra en kommando sendes som input til neste kommando, dette kalles for **piping** ("rørlegging" på norsk -- vi "kobler" to kommandoer sammen). Til dette brukes spesialtegnet '**|**' (vertikal strek eller "pipe") som skrives mellom de to kommandoene. Noen eksempler du kan teste (du vil antagelig trenge å lese manualsider for å forstå hvordan de virker):

* Telle antall filer (og kataloger) i stående katalog:

**$ ls | wc -l**

* Sortere listen over innloggede brukere på brukernavn:

**$ who | sort**

* Liste med alle brukere som heter "Magnus":

**$ cat /etc/passwd | grep -w Magnus**

* Hvor mange brukere heter "Magnus"?:

**$ cat /etc/passwd | grep -w Magnus | wc -l**

* Lage tilfeldig passord med 32 tall/bokstaver (denne er litt tricky):

**$ tr -cd '[:alnum:]' < /dev/urandom | fold -w32 | head -n1**

**\*** **The grep command is case sensitive**

To ignore upper/lower case distinctions, use the **-i option**

To search for a phrase or pattern, you must enclose it in **single quotes**

Some of the other options of grep are:

**-v display those lines that do NOT match   
-n precede each matching line with the line number   
-c print only the total count of matched lines**

**The character \* is called a wildcard,** and will match against none or more character(s) in a file (or directory) name.

**The character ?** will match exactly one character. **So ?ouse** will match files like house and mouse, but not grouse.

The operating system is the code that carries out the system calls. Editors, compilers, assemblers, linkers, utility programs, and command interpreters definitely are not part of the operating system, even though they are important and useful. At the risk of confusing things somewhat, in this section we will look briefly at the UNIX command interpreter, the shell. Although it is not part of the operating system, it makes heavy use of many operating system features and thus serves as a good example of how the system calls are used. It is also the main interface between a user sitting at his terminal and the operating system, unless the user is using a graphical user interface. Many shells exist, including sh, csh, ksh, and bash. All of them support the functionality described below, which derives from the original shell (sh). When any user logs in, a shell is started up. The shell has the terminal as standard input and standard output. It starts out by typing the prompt, a character such as a dollar sign, which tells the user that the shell is waiting to accept a command. If the user now types:

**date**

for example, the shell creates a child process and runs the date program as the child. While the child process is running, the shell waits for it to terminate. When the child finishes, the shell types the prompt again and tries to read the next input line. The user can specify that standard output be redirected to a file, for example,

**date >file**

Similarly, standard input can be redirected, as in

**sort <file1 >file2**

which invokes the **sort** program with input taken from file1 and output sent to **file2**. The output of one program can be used as the input for another program by connecting them with a **pipe**. Thus

**cat file1 file2 file3 | sort >/dev/lp**

invokes the **cat** program to concatenate three files and send the output to sort to arrange all the lines in alphabetical order. The output of **sort** is redirected to the file

**/dev/lp**, typically the printer.

If a user puts an ampersand after a command, the shell does not wait for it to complete. Instead it just gives a prompt immediately. Consequently,

**cat file1 file2 file3 | sort >/dev/lp &**

starts up the sort as a background job, allowing the user to continue working normally while the **sort** is going on. The shell has a number of other interesting features, which we do not have space to discuss here. Most books on UNIX discuss the shell at some length (e.g., Kernighan and Pike, 1984; Quigley, 2004; Robbins,

2005) Most personal computers these days use a GUI. In fact, the GUI is just a program running on top of the operating system, like a shell. In Linux systems, this fact is made obvious because the user has a choice of (at least) two GUIs: Gnome and KDE or none at all (using a terminal window on X11). In Windows, it is also possible to replace the standard GUI desktop (Windows Explorer) with a different program by changing some values in the registry, although few people do this.

To background a process, type an **&** at the end of the command line. For example, the command sleep waits a given number of seconds before continuing. Type

**$ sleep 10**

This will wait 10 seconds before returning the command prompt **$**. Until the command prompt is returned, you can do nothing except wait.

To run sleep in the background, type

**$ sleep 10 &**

**[1] 6259**

The **&** runs the job in the background and returns the prompt straight away, allowing you do run other programs while waiting for that one to finish.

The first line in the above example is typed in by the user; the next line, indicating job number and PID, is returned by the machine. The user is be notified of a job number (numbered from 1) enclosed in square brackets, together with a PID and is notified when a background process is finished. Backgrounding is useful for jobs which will take a long time to complete.

You can suspend the process running in the foreground by typing **^Z $ bg**

Note: do not background programs that require user interaction e.g. **vi**

When a process is running, backgrounded or suspended, it will be entered onto a list along with a job number. To examine this list, type

**$ jobs**

An example of a job list could be

**[1] Suspended sleep 1000  
[2] Running netscape  
[3] Running matlab**

To restart (foreground) a suspended processes, type

**$ fg $*jobnumber***

For example, to restart sleep 1000, type

**$ fg $1**

Typing **fg** with no job number foregrounds the last suspended process.

It is sometimes necessary to kill a process (for example, when an executing program is in an infinite loop). To kill a job running in the foreground, type **^C** (control c).

Alternatively, processes can be killed by finding their process numbers (PIDs) and using kill *PID\_number.* If a process refuses to be killed, uses the **-9** option

All students are allocated a certain amount of disk space on the file system for their personal files, usually about 100Mb. If you go over your quota, you are given 7 days to remove excess files.

The df command reports on the space left on the file system.

The du command outputs the number of kilobyes used by each subdirectory.

The **-s** flag will display only a summary (total size) and the \* means all files and directories.

**gzip** This reduces the size of a file, thus freeing valuable disk space. and note the size of the file using ls -l . Then to compress. To expand the file, use the gunzip command.

**zcat** will read gzipped files without needing to uncompress them first. If the text scrolls too fast for you, **pipe** the output though **less**.

The C shell keeps an ordered list of all the commands that you have entered. Each command is given a number according to the order it was entered.

**$ history (show command history list)**

If you are using the C shell, you can use the exclamation character (**!**) to recall commands easily.

**$ !! (recall last command)**

**$ !-3 (recall third most recent command)**

**$ !5 (recall 5th command in list)**

**$ !grep (recall last command starting with grep)**

You can increase the size of the history buffer by typing

**$ set history=100**

An example of an environment variable is the OSTYPE variable. The value of this is the current operating system you are using. Type

% echo $OSTYPE

More examples of environment variables are

* USER (your login name)
* HOME (the path name of your home directory)
* HOST (the name of the computer you are using)
* ARCH (the architecture of the computers processor)
* DISPLAY (the name of the computer screen to display X windows)
* PRINTER (the default printer to send print jobs)
* PATH (the directories the shell should search to find a command)

**Finding out the current values of these variables.**

ENVIRONMENT variables are set using the setenv command, displayed using the printenv or env commands, and unset using the unsetenv command.

To show all values of these variables, type

% printenv | less

**8.3 Shell Variables**

An example of a shell variable is the history variable. The value of this is how many shell commands to save, allow the user to scroll back through all the commands they have previously entered. Type

% echo $history

More examples of shell variables are

* cwd (your current working directory)
* home (the path name of your home directory)
* path (the directories the shell should search to find a command)
* prompt (the text string used to prompt for interactive commands shell your login shell)

**Finding out the current values of these variables.**

SHELL variables are both set and displayed using the set command. They can be unset by using the unset command.

To show all values of these variables, type

% set | less

**So what is the difference between PATH and path ?**

In general, environment and shell variables that have the same name (apart from the case) are distinct and independent, except for possibly having the same initial values. There are, however, exceptions.

Each time the shell variables home, user and term are changed, the corresponding environment variables HOME, USER and TERM receive the same values. However, altering the environment variables has no effect on the corresponding shell variables.

PATH and path specify directories to search for commands and programs. Both variables always represent the same directory list, and altering either automatically causes the other to be changed.

**8.4 Using and setting variables**

Each time you login to a UNIX host, the system looks in your home directory for initialisation files. Information in these files is used to set up your working environment. The C and TC shells uses two files called .login and .cshrc (note that both file names begin with a dot).

At login the C shell first reads **.cshrc** followed by **.login**

**.login** is to set conditions which will apply to the whole session and to perform actions that are relevant only at login.

**.cshrc** is used to set conditions and perform actions specific to the shell and to each invocation of it.

The guidelines are to set ENVIRONMENT variables in the **.login** file and SHELL variables in the **.cshrc** file.

**WARNING:** NEVER put commands that run graphical displays (e.g. a web browser) in your .cshrc or .login file.

**8.5 Setting shell variables in the .cshrc file**

For example, to change the number of shell commands saved in the history list, you need to set the shell variable history. It is set to 100 by default, but you can increase this if you wish.

% set history = 200

Check this has worked by typing

% echo $history

However, this has only set the variable for the lifetime of the current shell. If you open a new xterm window, it will only have the default history value set. To PERMANENTLY set the value of history, you will need to add the set command to the .cshrc file.

First open the **.cshrc** file in a text editor. An easy, user-friendly editor to use is nedit.

% nedit ~/.cshrc

Add the following line AFTER the list of other commands.

set history = 200

Save the file and force the shell to reread its .cshrc file buy using the shell source command.

% source .cshrc

Check this has worked by typing

% echo $history

**8.6 Setting the path**

When you type a command, your path (or PATH) variable defines in which directories the shell will look to find the command you typed. If the system returns a message saying "command: Command not found", this indicates that either the command doesn't exist at all on the system or it is simply not in your path.

For example, to run units, you either need to directly specify the units path (**~/units174/bin/units**), or you need to have the directory **~/units174/bin** in your path.

You can add it to the end of your existing path (the **$path** represents this) by issuing the command:

% set path = ($path ~/units174/bin)

Test that this worked by trying to run units in any directory other that where units is actually located.

% cd  
% units

To add this path PERMANENTLY, add the following line to your .cshrc AFTER the list of other commands.

set path = ($path ~/units174/bin)

● fork() og de andre systemkallene for spawning er dokumentert i man-systemet (Section 2: System Calls)

Prosesstrær i Linux: En Linux-prosess kan spawne flere barn, Alle prosessene som kjører på et Linux-system er derfor ordnet hierarkisk i et prosesstre, pstree – kommando for å se (deler av) prosesstreet

$ inneholde av variabel

Linux systemkall\* for terminering  
● exit: Normal terminering/kontrollert avslutning ved feil  
– Avslutter prosessen som kaller exit  
– Returnerer "exit status" (et heltall) til forelderprosessen  
● sigaction: Håndtering av fatale feil/avbruddssignaler fra OS  
– Gir mulighet til å håndtere feilen  
– Evt. terminere kontrollert i stedet for "bråstopp"  
● kill: Terminere/sende avbruddssignal til en annen prosess  
– Må ha rettigheter til å terminere prosessen  
– En forelderprosess kan vanligvis "drepe" egne barn(!)  
\*: Systemkallene i C er dokumentert i del 2 av manualsystemet i Linux, se f.eks. man 2 exit

Oppstart av prosesser fra shellet: Prosesser startes ved å gi en kommando som svarer til  
navnet på det eksekverbare programmet. Hvis programmet ikke ligger i stående katalog:  
– Katalogen der programmet ligger må enten være lagret i shell-variabelen PATH, eller søkesti må angis eksplisitt  
Hvis programmet ligger i stående katalog:  
– Angi ./ foran kommandonavnet, hvis ikke “.” ligger i listen av kataloger i PATH  
Input/output til prosessen kan redirigeres fra/til fil: ./somescript < my-data > my-results

Forgrunns- og bakgrunnsprosesser: Linux-programmer som startes på vanlig måte fra  
kommandolinjen, kjøres interaktivt “i forgrunnen”: $ program\_navn  
Shellet oppretter (spawner) en ny prosess med fork, og bruker deretter systemkallet wait – shellet blir suspendert og må vente til den nye prosessen er ferdig  
Ved å bruke tegnet & (“og-tegn”) etter en Linux-kommando, kjøres kommandoen “i bakgrunnen”: $ program\_navn &  
Shellet lager nå i stedet ny prosess ved bare å bruke systemkallet fork – suspenderes ikke med wait

Håndtering av bakgrunnsprosesser: Kan bare være én aktiv forgrunnsprosess i shellet, Prosessen i forgrunnen kan suspenderes (settes i wait-  
tilstand) ved å taste CTRL-Z  
Kommandoen bg vil starte opp igjen siste suspenderte jobb som en bakgrunnsprosess  
For å se liste over alle prosesser i bakgrunnen, med status og jobbnummer: jobs  
Kommandoen fg jobbnummer brukes for å flytte prosesser fra bakgrunn til forgrunn

PID: Identifisering av prosesser i Linux, Alle prosesser har en unik PID – ProsessIDentifikator: – Et heltall – prosessene nummereres fortløpende etterhvert  
som de opprettes – Angir indeksen i prosesstabellen der prosessdatene lagres – PID-nummere gjenbrukes når de blir ledige – Den første prosessen som startes, init, har PID lik 1. Prosesser har også bl.a.: – Parent ID (PPID), som er PID til forelderprosessen – User ID (UID), identifiserer brukeren som startet prosessen

Prosessinformasjon og -status i Linux: Kan bruke GUI-verktøy for “System Monitoring”, Fra shellet kan disse kommandoene brukes: top Løpende systemstatus og informasjon om alle prosessene på systemet, ps Skriv ut hele eller deler av prosesstabellen, pstree Skriver ut hele eller deler av "slekstreet" for prosessene

top – dynamisk prosessinformasjon: Kommandoen top viser: – Systemstatus øverst på skjermen – Liste med info om alle prosesser på systemet – Sortert på ressursbruk – “grådige” prosesser øverst – Listen oppdateres hvert tredje sekund (default)  
Fullskjerm – blokkerer kommandolinjen inntil exit  
Interaktiv – tastetrykk fra bruker forandrer virkemåten\*  
\*: Trykk ‘h’ når top kjører for å se en oversikt over ulike opsjoner/kommandoer

ps – skriver ut prosesstabellen: Kommandoen ps gir et “snapshot” av prosessene på  
systemet “akkurat nå” Svært mange opsjoner til kommandoen for å angi: – Hvilke prosesser som skal vises – Hva slags informasjon som skal vises

Noen av outputkolonnene fra ps: PID Prosessidentifikator %CPU CPU-bruk  
%MEM Minnebruk VSZ Forbruk av (virtuelt) minne (KB) RSS Forbruk av fysisk RAM (KB)  
TTY Terminalvinduet (? hvis ingen TTY) START Når prosessen startet TIME Totalt forbruk av CPU-tid COMMAND Kommando som startet prosessen STAT Prosess-status, en tegnkode

ps – koder for prosess-status (STAT): D uninterruptible sleep (usually IO) R running or runnable (on run queue) S interruptible sleep (waiting for an event to complete) I idle (for kernel threads) T stopped Z "zombie" process, terminated, not killed by parent

nice – prioritering av prosesser i Linux: Prosesser får en “nice-verdi” (default 0) ved oppstart, Nice-verdier er heltall med verdier fra -20 til 19   
19 : “Super-nice” prosess, prioriteres lavt, langsom  
-20 : “Grådig” prosess, prioriteres på topp, rask  
Brukere kan sette positiv nice-verdi for en prosess: nice -15 program  
Bare sys.adm./root kan bruke negative nice-verdier, Prioritet til kjørende prosess kan endres med renice

The UNIX system has a command, nice, which allows a user to voluntarily reduce the priority of his process, in order to be nice to the other users. Nobody ever uses it.

Stoppe/drepe prosesser: Prosesser terminerer vanligvis når de kommer til slutten  
av programkoden under eksekvering, En prosess kan avslutte “unormalt” pga. run-time feil, OS legger da ofte igjen en fil med navn core, Dette er en “core dump”, som inneholder minnet for prosessen slik det var da den terminerte unormalt, Noen ganger kan prosesser henge eller oppføre seg unormalt, og må stoppes av bruker, Ofte vil en Ctrl-C (interrupt) fra kommandolinjen stoppe forgrunnsprossessen i shellet

Stopping av barneprosesser: Hvis en barneprosess terminerer uten at forelder er  
tilgjengelig (f.eks. i sleep-status): – Barnet terminerer, men tas ikke ut av systemet – Barn må først “rapportere” exit til forelder – Barn blir til en “zombie”-prosess inntil forelder våkner. Hvis forelderprosessen dør mens barnet fortsatt kjører, får vi en “orphan” prosess – Feilsituasjon, ingen andre enn init får være “foreldreløs” – init (evt. forelders forelder) vil da vanligvis “adoptere” barneprosessen

Brukerkommandoer for å drepe prosesser: Tre ulike alternativer: 1. Prøv å terminere programmet normalt med en “exit”- kommando e.l. 2. Stop prosessen fra en “GUI process monitor”: Menyvalg: “Stop process” / “Kill process” 3. Stopp prosessen ved å sende den et signal med kommandoen kill

Linux-prosesser og signaler: Alle prosesser i Linux responderer på signaler, Signaler er beskjeder fra OS-et om at en prosess skal avslutte eller endre “oppførsel” fordi “noe uventet har skjedd”, Kommandoen kill kan brukes for å sende et signal fra  
OS til en prosess, Signalene identifiseres med nummer eller navn

Name # Description  
SIGHUP 1 Hangup  
SIGINT 2 Terminal interrupt  
SIGQUIT 3 Terminal quit  
SIGILL 4 Illegal instruction  
SIGIOT 6 IOT Trap  
SIGBUS 7 BUS error  
SIGFPE 8 Floating point exception  
SIGKILL 9 Kill (can't be caught or ignored)  
SIGSEGV 11 Invalid memory segment access  
SIGPIPE 13 Write on a pipe with no reader, broken pipe  
SIGTERM 15 Termination  
SIGSTKFLT 16 Stack fault  
SIGCHLD 17 Child process has stopped or exited  
SIGCONT 18 Continue executing, if stopped  
SIGURG 23 Urgent condition on socket  
SIGXCPU 24 CPU limit exceeded  
SIGXFSZ 25 File size limit exceeded  
SIGPWR 30 Power failure restart

kill: Send et signal til en prosess, Brukere kan bare sende signaler til egne prosesser, Liste over alle signaler: kill -l, Spesielt signal: SIGKILL (9) for å stoppe prosess: – Sendes ikke videre til prosessen fra OS-et, – Kalles en “sure kill”: Prosessen kan ikke selv stoppe eller håndtere en SIGKILL

kill PID Terminér en bestemt prosess, kill -signal PID Send et gitt signal til en prosess, kill -KILL PID “Sure kill”, kill -KILL -1 Massemord, kill -HUP PID Brukes ofte for å restarte server-prosesser (web, ftp, o.l.)  
\*: Se også kommandoene pkill og killall

Avslutte og stenge ned Linux: shutdown [options] time [message] – Starter nedstenging av Linux til angitt tid – Stenger for nye login – Tid kan angis på mange måter, svært fleksibelt  
shutdown opsjoner : -H : Halt (stopper alle CPU'er) -P : Power-off -r : Reboot

Hvordan er kjernetråder implementert i Linux? Linux-kjernen håndterer "egentlig" bare prosesser. En kjernetråd i Linux er en "lettvekts"-prosess som deler minne, kode og andre prosessdata med andre prosesser. Lettvektsprosessene er raskere å opprette og håndtere. "Tråder" opprettes fra et C-program med systemkallet clone, som gir mulighet for bl.a. å dele hele minneområdet til kallende prosess med lettvektsprosessen som opprettes. All informasjon om både vanlige prosesser og lettvekts-prosesser/tråder lagres internt i OS-kjernen.

Linux-kommandoer for å se kjernetrådinfo: Linux-kjernen holder oversikt over hvilke prosesser som har flere "tråder" med tilgang til samme kode/data i minnet. Informasjonen ligger tilgjengelig i systemkatalogen /proc:  
cat /proc/PID/status/ Info. om PID, inkludert ant. tråder  
ls /proc/PID/task/ Alle trådene i prosessen PID  
Kan også se trådinformasjon med f.eks. top og ps:  
ps -eLF Antall "lettvektsprosesser" pr PID  
top -H Vis alle tråder (se også feltene nTH og TGID)

De fleste nyere implementasjoner av pthreads i Linux er hybride, med én-til-én mapping av hver brukertråd til en tilsvarende kjernetråd i OS – "nesten kjernetråder"  
pthreads i Linux kan bruke dette til bl.a. å utnytte prosessorene bedre og unngå at tråder blokkerer andre tråder i samme prosess

gcc -pthread -o

For å måle effektivitet av et program, kan vi se på hvor lang tid programmet bruker for å løse et problem, *time*, som kan brukes i shellet til å måle tidsforbruket til et program. Programmet vil da kjøre som normalt, men etter at det er ferdig vil *time* skrive ut informasjon om tidsforbruket til programmet. ***real:*** Dette er den faktiske tiden — drøyt 2.2 sekunder — som har gått fra prosessen startet til den er ferdig, kalles også "wall clock time". Dette inkluderer både tiden prosessen har kjørt i CPU, og tiden den har ventet på f.eks. input eller på å få bruke en CPU som er opptatt av andre prosesser. ***user:*** Tiden som prosessen har fått kjøre i CPU i user mode, når den eksekverer sin egen kode. ***sys:*** Tiden som prosessen har kjørt kode i kernel mode, typisk utførelse av systemkall. Summen av *user* og *sys* er den totale CPU-tiden som prosessen har brukt. Merk at både *user* og (spesielt) *sys* kan være null sekunder i utskriften fra *time*, selv om prosessen har brukt tid både i user og kernel mode. Dette skjer når kjøretidene er så små/korte at de ikke kan måles.

Tiden som et program bruker for å løse et gitt problem avhenger av hvor rask maskinens hardware er, hvilken kompilator og operativsystem som brukes, og hvor opptatt maskinen er med å utføre andre prosesser som er aktive samtidig med vårt program.

Eksempel på IPC: Pipes i Linux  
Alle prosesser har tre åpne "I/O-kanaler" når de starter: – Standard input (stdin), default satt til å være tastaturet – Standard output (stdout), default skjermen/terminalvindet   
– Standard feilmeldingsenhet, default skjermen  
Shellet i Linux tilbyr redirigering av I/O til/fra en prosess:  
– stdin kan redirigeres til å lese fra en fil med operatoren <  
– stdout kan redirigeres til å skrive til en fil med >  
– stderr kan redirigeres til å skrive til en fil med >2  
Output (stdout) fra én prosess kan sendes som input (stdin) til en annen prosess gjennom en "pipe" i operativsystemet

Hvis vi setter tegnet | ("pipe") mellom to (eller flere) Linux-kommandoer, vil prosessene startes samtidig og kommunisere med hverandre mens de kjører:  
tail -f access.log | grep 'hiof.no'  
Output til stdout fra første prosess (tail) redirigeres til et "pipe-buffer" i RAM  
Input fra stdin til andre prosess (grep) redirigeres til å lese fra det samme bufferet  
Første prosess må vente med å skrive hvis bufferet blir fullt  
Andre prosess må vente med å lese til det ligger noe i bufferet

mkfifo

For å lære mer om hvordan named pipes virker og er implementert, se man 7 fifo.

Et eksempel er det som skjer når du ønsker å stoppe et program med *Ctrl-C*. Hvis du kjører en forgrunnsprosess i terminalvinduet og trykker *Ctrl-C*, vil shellet be OS sende et avbruddssignal (signal #2: SIGINT) til prosessen. Hvis programmet som prosessen kjører ikke har implementert en egen avbruddshåndtering for dette signalet, vil prosessen termineres.

For en oversikt over alle signaler som kan sendes, med signalnummer og -navn, gi kommandoen *kill -l* til shellet:

$ kill -l

1) SIGHUP 2) SIGINT 3) SIGQUIT 4) SIGILL 5) SIGTRAP

6) SIGABRT 7) SIGBUS 8) SIGFPE 9) SIGKILL 10) SIGUSR1

11) SIGSEGV 12) SIGUSR2 13) SIGPIPE 14) SIGALRM 15) SIGTERM

16) SIGSTKFLT 17) SIGCHLD 18) SIGCONT 19) SIGSTOP 20) SIGTSTP

21) SIGTTIN 22) SIGTTOU 23) SIGURG 24) SIGXCPU 25) SIGXFSZ

26) SIGVTALRM 27) SIGPROF 28) SIGWINCH 29) SIGIO 30) SIGPWR

...

For mer informasjon om hvordan delt POSIX-minne er implementert og brukes i Linux, kan du gi kommandoen:

$ man 7 shm\_overview

Nedenfor er korte beskrivelser av de viktigste funksjonene for å håndtere delte POSIX minnområder:

* shm\_open():

Brukes til å opprette og åpne et nytt delt minneområde, eller åpne et minneområde som allerede eksisterer. Funksjonen er nært beslektet med systemkallet *open()* for å åpne vanlige filer og I/O-strømmer. Et kall på *shm\_open()* vil returnere en fildeskriptor som brukes av de andre metodene gitt nedenfor.

Merk at et delt minneområde som opprettes med *shm\_open()* vil ligge og oppta plass i minnet inntil en prosess fjerner det med et kall på *shm\_unlink()* (se nedenfor). Det blir ikke slettet/gjenbrukt selv om prosessen som opprettet det terminerer.

* ftruncate():

Setter størrelsen/lengden (i bytes) på et delt minneområde. Et nytt minneområde som er opprettet/åpnet med *shm\_open()* vil ha lengde lik null. Lengden må settes eksplisitt med et kall på *ftruncate()* før området brukes til å lese/skrive data.

* mmap():

Denne funksjonen må også kalles før minneområdet brukes. Den gjør at minneområdet inkluderes i prosessens adresserom, slik at det kan behandles som f.eks. en array eller en struct i C-koden.

* munmap():

Funksjonen skal kalles etter at prosessen er ferdig med å bruke minneområdet. Den gjør "det motsatte" av *mmap()* og fjerner minneområdet igjen fra prosessens adresserom.

* shm\_unlink():

Brukes for å slette et delt minneområde med et gitt navn.

Minnehåndtering i Linux: Linux er laget for å få et portabelt OS som skal virke for "alle" typer hardware, også RISC-systemer som vanligvis ikke støtter segmentert hukommelse i hardware. Memory Manager kan derfor konfigureres på mange ulike måter. Standardoppsettet av Linux bruker: – Virtuelt minne med paging, uten segmentering \* – 4 KB pages – Pagetabeller med tre nivåer – En variant av aging-algoritmen for page replacement  
\*: Det brukes en slags segmentering for de ulike hoveddelene av minnet for hver prosess men, ikke "småsegmenter" for de ulike delene av et program, slik som beskrevet i læreboken

Kommandoer for å se minnebruk i Linux Hele systemet:  
more /proc/meminfo  
top  
free -m  
vmstat  
ps aux --sort -rss  
gnome-system-monitor  
Enkeltprosesser:  
ps v PID  
cat /proc/PID/maps

Bourne Again Shell:  
Shell: Interaktiv linjebasert "kommandotolker" i Linux. Mest brukt: Bourne Again Shell – Bash  
Shellet leser en og en linje med tekstlig input fra bruker/ tastaturet (egentlig fra standard innhet – stdin)  
Input er Linux-kommandoer som skal utføres \*  
Shellet interpreterer (tolker) input, finner ut hva som skal utføres og starter utføringen  
:\* En "kommando" er angivelse av hvor på disken OS finner et ferdig kompilert program. Shellet ber OS om å laste dette programmet inn i RAM og starte det som en prosess på maskinen

Shellet kan også lese input fra en fil. Hver linje på filen interpreteres som en Linux-  
kommando og eksekveres på vanlig måte. Shellet tilbyr også et programmeringsspråk med bla. kontrollstrukturer og variabler som kan brukes til å styre og kontrollere kjøringen av kommandoer. En fil med shellkommandoer og programkode kalles for  
et shellprogram eller shellscript. Shellprogrammering kalles også scripting

Hva brukes shellprogrammer til? En av verdens meste brukte programmerings-  
omgivelser. For effektivisering og automatisering av Linux-bruk. Essensielt verktøy for drift og systemadministrasjon. Shellprogrammer kompileres ikke: – De interpreteres og kjøres linje for linje av shellet – Interpretering er mye langsommere enn å kjøre ferdig  
kompilert kode – Ikke for "vanlig" programmering der effektivitet er viktigst

Hvorfor lære shellprogrammering? Alle Linux-brukere bør kunne scripting. Gjør deg til en bedre og mer effektiv bruker. Lett å lage gode løsninger raskt og enkelt. Øker forståelsen for operativsystemet. Etterspurt og relevant kompetanse i arbeidslivet. Shellprogrammering dukker opp både i lab.oppgavene og på eksamen(?) i dette kurset

Innhold av tekstfilen/shellprogrammet hello.sh :  
#!/bin/bash  
echo "Hello, world!"

Kjøring av shellprogrammet hello.sh:

chmod u+x hello.sh  
./hello.sh

Eller:

bash hello.sh

Utskrift fra shellprogrammer:  
echo – Utskrift av linjer med tekst  
Opsjoner:  
-e Tillat escape-koder for å legge inn spesialtegn\*  
-n Ikke gjør linjeskift etter utskrift av teksten  
\*: En escape-kode er en "backslash" etterfulgt av et tegn: \t: TAB \n: NEWLINE

Data og variabler i shell: Tekst og heltall  
Shellvariabler brukes til å lagre dataverdier  
Refereres til med variabelnavn: – Variabelnavn kan inneholde a-z A-Z 0-9 og \_  
– Må begynne med en bokstav eller \_  
Variable i shell er uten datatype: – Lagrer bare bytes – Tolkes som enten tekststrenger eller heltal

Variable i shellet trenger ikke deklareres: Opprettes automatisk av shellet første gangen det er en referanse til variabelen  
Variable kan deklareres med kommandoen declare, som gir mulighet til å gi variabelen spesielle attributter/egenskaper – se help declare\* for dokumentasjon  
\*: declare er en såkalt "builtin" kommando i Bash. Dokumentasjon for Bash-builtins finnes med kommandoen help og ikke med man som for vanlige Linux-kommandoer (se man builtins)

Se på innholdet i variabler: For å få tilgang til dataverdien lagret i en variabel, brukes tegnet '$' (dollartegn) foran variabelnavnet: $variabelnavn  
eller, for å separere variabelnavnet fra annen tekst: ${variabelnavn}  
For å fjerne eller nullstille innholdet i en variabel: unset variabelnavn

Tilordning av verdier til variabler  
Syntaks\*: variabelnavn=verdi  
verdi kan være: – En konstant (heltall eller streng) – Verdien til en annen variabel (hentes ut med $) – Resultatverdien fra en regne- eller streng-operasjon – Output fra en Linux-kommando (aka "substitusjon")  
\*: Space før eller etter likhetstegnet vil gi feilmelding

Innlesing av verdier til variable read variabel\_1 variabel\_2...  
Leser ett og ett item fra standard input og lagrer verdiene i angitte variable  
Opsjoner: -p string Bruk string som ledetekst/prompt -n number Ikke les flere enn number antall tegn -t secs Time-out etter secs sekunder

Regning med heltall\*  
Regnestykker i Bash kan skrives inne i doble paranteser: ((A = A + 1))  
Merk at vi da ikke trenger å bruke $ foran variabelnavnene  
Aritmetiske operatorer: + - \* / De fire vanlige regneartene   
++ – Positivt og negativt inkrement  
% Rest ved heltallsdivisjon  
\*\* Potensuttrykk med eksponenter  
\*: Regning med floating point tall i shellet/fra kommandolinjen kan gjøres med Linux-"kalkulatoren" bc

Håndtering av strenger: Variable i shellet består av bytes som kan tolkes som  
en tegnstreng  
Bash-syntaks for strengoperasjoner på variabler: ${variabel[operasjon(er)]}

Noen strengoperasjoner:  
${#navn} Antall tegn i skallvariabelen navn  
${navn:3} Substreng fra 4. tegn (0 er første tegn)  
${navn:2:3} Substreng fra 3. til 5. tegn  
${navn/en/an} Bytt første forekomst av en med an  
${navn//en/an} Bytt alle forekomster av en med an  
${navn//en/} Fjern alle forekomster av en  
${navn/#en/an} Bytt bare hvis en er i starten på streng  
${navn/%en/an} Bytt bare hvis en er i slutten av streng

Tegnstreng-konstanter i shellet: Forskjellen på '' og ""  
• Single quotes '' : Hele strengen brukes akkurat slik den er skrevet  
• Double quotes " " : Alle tegn beholdes uendret, unntatt $ , ` og \ $ brukes til å hente ut verdier av variable   
\ kan brukes som “escape-tegn” (foran f.eks. $ , eller ")

Kommandosubstitusjon: Output fra en Linux-kommando tilordnes som verdi til en variabel. Kan også bruke output fra en kommando som input/ argument(er) til en annen kommando  
Syntaks: var=$(kommando)

Eksempler på kommandosubstitusjon  
antall\_filer=$(ls | wc -l) echo "Antall filer: $antall\_filer"  
echo "Hællæ, $(whoami), tidspunktet er nå \$(date)"  
rm $(cat filer\_som\_skal\_fjernes.txt)  
echo -e "Filer med navn som inneholder 'd'\ eller 'o':\n$(ls -d \*[od]\*)"  
script\_liste=$(ls \*.sh) echo $script\_liste

Parametre til shellprogrammer: Shellprogrammer kan ta parametre som input  
Parameterverdier angis på kommandolinjen: ./shellprogram param1 param2 param3  
Verdiene er tilgjengelige som $1 , $2 , $3  
$0 er navnet på selve shellprogrammet  
$# er antallet parametre angitt på kommandolinjen  
$@ og $\* angir begge alle parametre til programmet

Betinget eksekvering \*?  
Programmet utfører operasjon(er) bare hvis en logisk betingelse er sann  
Bash tilbyr to kontrollstrukturer for å kunne gjøre betinget eksekvering: if case  
Både if og case krever at vi kan gjøre logiske tester i shellet for å sjekke om noe er sant eller usant  
\*: Kalles også for seleksjon i programkoden

Exit-status og variabelen $?  
For å forstå hvordan logiske tester håndteres i shell-programmer, må vi kjenne til begrepet “exit-status”  
Alle Linux-kommandoer returnerer en exit-status når de avslutter – en tallverdi som lagres i shellet  
Standard er at exit-status lik 0 (null) betyr “alt i orden”  
En exit-status > 0 brukes til å signalisere at noe er feil  
Vi kan sette exit-status for våre egne shellprogrammer med termineringskommandoen exit verdi  
Exit-status for siste utførte kommando ligger alltid lagret i shellet. Denne verdien kan hentes frem med $?

test – Logiske tester i shellet  
● Kommandoen test : – Beregner logiske utrykk – Returnerer med exit-status 0 hvis uttrykk er true – Returnerer med exit-status 1 hvis uttrykk er false  
● Syntaks: test logisk-uttrykk  
● For å få ryddigere kode, tillater Linux alternativt denne syntaksen for å kjøre kommandoen test : [ logisk-uttrykk ]

Sammenligning og test av tekststrenger  
Logisk uttrykk Uttrykket er true hvis:  
[ s1 = s2 ] De to strengene er like  
[ s1 != s2 ] De to strengene ikke er like  
[ -n s ] s har lengde større enn 0  
[ s ] s har lengde større enn 0  
[ -z s ] s har lengde lik 0 (tom)

Sammenligning av to tallverdier  
Logisk uttrykk Uttrykket er true hvis:  
[ n1 -eq n2 ] De to tallene er like  
[ n1 -ne n2 ] De to tallene ikke er like  
[ n1 -gt n2 ] n1 er større enn n2  
[ n1 -ge n2 ] n1 er større eller lik n2  
[ n1 -lt n2 ] n1 er mindre enn n2  
[ n1 -le n2 ] n1 er mindre eller lik n2

Logiske tester knyttet til filer: Kan også teste egenskaper ved filer ved å bruke  
logiske uttrykk og kommandoen test i Linux  
Nyttig for unngå run-time feil i shellprogrammer: – Kan teste om filer eller kataloger finnes før vi prøver å gjøre noe med dem – Kan teste om filer er av riktig type – Kan sjekke om vi har rettigheter til filer før vi forsøker å lese, skrive eller eksekvere  
Gir også mulighet til å sammenligne metadataene for filer (tidsstempel, inodenummer etc.)

Sammenligning og testing av filer \*  
Logisk uttrykk Uttrykket er true hvis filen:  
[ -e fil ] Eksisterer  
[ -f fil ] Er en vanlig (regulær) fil  
[ -d fil ] Er en katalog  
[ -r fil ] Er lesbar (tilsvarende for -w og -x )  
[ -O fil ] Eies av deg/bruker ( -G for gruppe)  
[ fil1 -ef fil2 ] fil1 og fil2 er samme fil (inode)  
[ fil1 -nt fil2 ] fil1 er nyere enn fil2  
[ fil1 -ot fil2 ] fil1 er eldre enn fil2  
\*: Det finnes flere logiske operatorer for filer, se man test

Sammensatte uttrykk: NOT, AND og OR  
Logisk uttrykk Uttrykket er true hvis:  
[ ! u ] Det logiske uttrykket u er false  
[ u1 -a u2 ] Begge uttrykkene u1 og u2 er true  
[ u1 -o u2 ] Minst ett av u1 og u2 er true  
( ) Brukes til gruppering/presedens  
Eksempel: [ ! \( u1 -o u2 \) -a \( u3 -o u4 \) ]

Syntaks: if TEST-KOMMANDO; then FLERE-KOMMANDOER; fi  
Virkemåte: – Hvis TEST-KOMMANDO returnerer med exit-status lik 0, så utføres FLERE-KOMMANDOER , ellers utføres de ikke

Vanlig skrivemåte for bruk av if i shellprogrammer: Dropper vanligvis semikolon ...skriver i stedet hver del av if - setningen på hver sin linje, med indentering for lesbarhet  
Bruke oftest kommandoen test , skrevet med [] , som "test-kommando" ... men en hvilken som helst Linux-kommando kan brukes etter if

case - setningen i shellprogrammer: case-setningen er alternativ måte å skrive en flergrenet if-test på  
Hvert "case" er en liste med verdier som matches mot et gitt uttrykk  
For hvert "case" angis det hvilke kommandoer som skal kjøres  
Tilsvarende konstruksjoner finnes i svært mange programmeringsspråk, gir ofte ryddigere kode

case – syntax:  
case expression in  
list1)  
action(s)  
;;  
list2)  
action(s)  
;;  
.  
.  
.  
\*)  
action(s)  
;;  
esac  
Kan bruke regulæruttrykk i angivelsene av list  
\*) angir det som utføres hvis ingen av de andre angitte case matcher  
expression  
Hvis hvert case avsluttes med ;& i stedet for ;; , vil eksekveringen fortsette med neste case, i stedet for å hoppe til esac

&& og || : Betinget eksekvering  
● Kan også bruke spesialtegnene && og || i Bash til å gjøre seleksjon i shellprogrammer  
● Spesialtegnene settes mellom to kommandoer:  
&& Utfør neste kommando bare hvis forrige OK  
|| Utfør neste kommando bare hvis forrige ikke OK  
● "OK" betyr at kommando terminerer med exit-kode lik 0  
● && og || er egentlig bare "forkortede if-tester"

Komprimering av kode med && og ||  
if [ -f $filnavn ] then  
rm filnavn  
else  
echo "No regular file $filnavn found"  
exit 1  
fi  
[ -f $filnavn ] && rm $filnavn || { echo "No\  
regular file $filnavn found"; exit 1; }

Løkkestrukturer i Bash:  
while – Går så lenge en løkkebetingelse er true  
until – Går inntil en løkkebetingelse er true  
for in – Itererer over hvert element i en angitt liste

while/until condition   
do   
commands  
done   
● condition håndteres likt som i if – setningen  
● Gjør ofte tallregning i løkken som endrer betingelsen  
● Løkker kan også styres med disse to kommandoene:  
break Avbryt og hopp ut av løkken  
continue Avbryt og gå til toppen av løkken igjen

shift  
● Kan brukes for å gå gjennom alle parametrene til et shellprogram med en løkke  
● Et kall på shift flytter alle innparametre til scriptet "et hakk til venstre":  
– $1 overskrives med $2  
– $2 overskrives med $3  
– Etc...

for in – løkke  
for var in list  
do  
command(s)  
done  
list er en liste med verdier, adskilt med whitespace. Listen kan angis eksplisitt eller med jokernotasjon, lages med en Linux-kommando eller hentes fra en variabel. Løkken går én gang per element i listen. I hver iterasjon lagres nåværende verdi i listen i var

Bruk av seq sammen med løkker\*  
seq [first] [step] last  
Returnerer liste av tall fra og med first (default 1) til og med last , med steg på step (default 1)

\*: Bash tilbyr også en for-løkke som er mer lik C og Java, se f.eks. <https://linuxconfig.org/writing-a-c-style-bash-for-loop-example>

Løkker og input / output: Vi kan redirigere I/O for løkker i Linux, akkurat som for andre kommandoer. All lesing fra stdin og skriving til stdout som foregår inne i løkken, mellom nøkkel-ordene do og done , kan redirigeres til/fra en pipe eller en fil

Kommandoen read kan brukes sammen med en while - løkke for å behandle input linje for linje:

#!/bin/bash  
SUM=0  
while read X Y  
do  
SUM=$((SUM+X\*Y))  
done  
echo $SUM

➔ Én linje med data leses  
fra stdin i hvert  
gjennomløp av løkken  
➔ Eller fra en fil hvis I/O  
for hele scriptet  
redirigeres

Array (tabell) i shellprogrammer: En datastruktur som kan lagre flere verdier. Verdiene ligger 'etter hverandre' i en indeksert liste. En array i Bash indekseres fra 0. Bash tilbyr bare éndimensjonale arrayer. Hvert element i en array er en shellvariabel

Eksempler på opprettelse av array  
● A=()  
● A2=(Jan Per Ole Kari)  
● A3=([0]=Jan [1]=Per [2]=Ole [3]=Kari)  
● A4=([1]=Per [3]=Kari [2]=Ole [0]=Jan)  
● read -a A5  
● A6=($(ls))  
● A7=({a..z} {A..Z} {0..9})

Tilgang til elementene i en array A  
● Legge inn en verdi: A[index]=value  
● Hente ut en verdi: a=${A[index]}  
● Alle elementene: ${A[@]} eller ${A[\*]}  
● Hente ut delarray: A2=${A[@]:first:last}  
● Lengde av array \* : ${#A[@]} eller ${#A[\*]}  
\*: Lengden av en array er definert som antall elementer som ikke er tomme

Tomme variabler og defaultverdier: En tom variabel inneholder bare verdien NULL. Bash tilbyr mulighet til å bruke en alternativ defaultverdi hvis en variabel er (eller ikke er) tom  
Defaultverdier kan anvendes til: – Fleksibel håndtering av parametre til shellscript  
– Mulighet for default svar ved interaktiv input

Operatorer for default-håndtering  
${variabel:-streng} Bruk verdien streng hvis variabel er tom  
${variabel:=streng} Bruk verdien streng hvis variabel er tom, og sett variabel lik streng  
${variabel:+streng} Bruk verdien streng hvis variabel ikke er tom  
${variabel:?melding} Skriv ut (feil)melding hvis variabel er tom

Shellfunksjoner  
● Tilsvarer metoder \* i språk som f.eks. Java: – Samler flere kommandoer i en gruppe eller kodeblokk – Kommandoene kan da senere brukes bare ved å referere til shellfunksjonens navn – En shellfunksjon kjøres som en vanlig kommando i shellet, men starter ikke noen ny prosess  
● Brukes til: – Gjenbruk og modularisering av kode – Lage shortcuts for lengre, kompliserte kommandoer  
\*: Aka «funksjoner», «prosedyrer», «subrutiner»

Deklararasjon av shellfunksjoner  
● Syntaks:  
function funksjonsnavn {  
kommando  
kommando  
...  
}

Bruk av shellfunksjoner  
● Funksjoner må alltid defineres før de kan brukes  
● Kalles ved bare å angi funksjonsnavnet  
● Funksjoner kan ta i mot innparametre:  
– Fungerer som for et vanlig shellscript  
– Listes opp etter funksjonsnavnet  
– Refereres til som $1 , $2 , $3 etc.  
● Se koden til en funksjon: type funksjonsnavn

trap – håndtering av signaler i script  
● Operativsystemet kan sende signaler til prosesser, ofte for å signalisere at noe er 'galt', f.eks.: – Divisjon med null – Bruker har trykket interrupt ( Ctrl-C ) – Et device har feilet... og mye annet  
● Signaler får ofte OS til å terminere prosessen  
● Vi kan bruke kommandoen trap til å håndtere signaler inne i shellprogrammer, for f.eks. å terminere kontrollert hvis noe går galt

Bruk av trap:  
trap [kommandoer] signalkode[r]  
● Når et av signalene som er listet opp mottas, utføres kommandoer i stedet for default signalhåndtering i OS  
● Oversikt over alle signalkoder: trap -l  
● Kan også angi noen andre signalkoder til trap , bl.a.:  
EXIT kode utføres når shellprogramet terminerer  
ERR kode utføres når kommandoer feiler

Debugging og testing av shellprogrammer  
● Shellprogrammering krever prøving og feiling  
● Test interaktivt hver ny programlinje som skrives  
● Debuggingsopsjonene -v og -x :  
bash -v program Skriver ut kodelinjene etter hvert som de utføres  
bash -x program Skriver ut kommandoer som utføres og verdien av variable som brukes

Definisjon av filbegrepet i Linux: En fil i Linux er enten: – En kilde som data kan leses fra, eller: – Et medium som data kan skrives til  
● Betyr at “alt er filer”, også f.eks. skjerm, mus og printer: – Applikasjoner kan behandle I/O-enheter på samme måte som vanlige diskfiler – Muliggjør redirigering av I/O i Linux   
– Implementeres i Linux ved å representere alle "devices" med en egen spesialfil/driver (i systemkatalogen /dev) som kan leses fra og/eller skrives til som en vanlig fil

Fil- og katalognavn i Linux: Kommandoene for filhåndtering i Linux bruker filnavn til å  
identifisere både filer og kataloger/mapper. Filnavn kan inneholde alle tegn unntatt / og NULL-tegnet. Lengde på filnavn opptil 255 tegn. Ingen regler for “extensions” eller formatering – men det er alltid lurt å holde seg til standarder og konvensjoner

"Regler" for bruk av tegn i filnavn i Linux: Unngå whitespace. Unngå spesialtegn for shell som $ \* @ ! < > |  
Unngå særtegn som f.eks. norske æ ø å. Bruk: 0-9 a-z A-Z . - \_  
Bruk i størst mulig grad små bokstaver

Tre ulike filtyper (i Linux): 1. Vanlige filer (regular files) 2. Kataloger (directories)  
3. Spesialfiler: ▻ Block device file: For disk/I/O-enheter som leser/skriver data en og  
en blokk (en større sammenhengende bytesekvens) om gangen ▻ Character device file: For serielle I/O-enheter som leser/skriver et og et tegn om gangen – tastatur, mus, printere, ▻ Named pipe file: Buffer for data-utveksling mellom prosesser ▻ Symbolic link file: Lenke (soft) til en annen fil på systemet ▻ Socket file: For kommunikasjon over nettverk

Se filtype i Linux: ls -l:   
- Regular file  
d Directory  
b Block device  
c Character device  
p Named pipe (se man mkfifo )  
l Symbolic link  
s Socket (se f.eks. i /run )

Vanlige filer – Regular files: Enten ASCII-filer (linjer med tekst) eller binære filer (ikke tekst). Vanlige filer kan være datafiler, eksekverbare scripts (tekst) eller eksekverbar kompilert kode (binært). Binære filer er oftest strukturerte iht. et internt lagringsformat for f.eks. bilder, video, lyd eller eksekverbar kode. OS kjenner oftest igjen en del ulike typer filer og vet hvilke applikasjoner som skal startes for å håndtere dem

Metadata i Linux, for vanlige filer: – Filstørrelse i antall bytes – Device som filen ligger på   
– Eier av filen og filens brukergruppetilhørighet – Tilgangsinformasjon – hvem kan lese/skrive/eksekvere fil – Tidspunkter for siste endring av filen og metadataene   
– Tidspunkt for siste gang filen ble brukt. – Antallet linker til denne filen – Adresser til diskblokkene der filen fysisk ligger lagret

Søkesti (path name): For å angi hvor en fil ligger i katalogtreet. En søkesti i Linux er en liste av katalognavn skilt med tegnet "/" (skråstrek/slash) \*  
● Absolutt søkesti: – Starter på toppen av filsystemet, i toppnivåkatalogen  
root , som angis med / – Går helt frem til katalogen der filen vi skal bruke befinner seg  
● Relativ søkesti: – Stien starter i stående katalog ("working/current directory") og går frem til katalogen der filen befinner seg  
\*: Andre OS bruker andre skilletegn enn "/", f.eks. MS-Windows som bruker "\" (aka. "bakslask")

Søkestier, shell og Linux-kommandoer \*  
● Input til en Linux-kommando er typisk en fil eller katalog  
● Shellet må da vite hvor det kan finne: – Den eksekverbare filen som svarer til kommandoen som skal utføres (f.eks. /bin/ls ) – Filene som brukes som input og output  
● Filene kan ligge i en hvilken som helst katalog  
● Vi må enten angi søkestien til filene eksplisitt, eller legge inn default søkestier i systemvariabelen PATH i shellet  
\*: Filer kan også håndteres med pek-og-klikk / drag-and-drop i et GUI for Linux, omtrent på samme måte som i Windows og OS X

Spesialtegn i søkestier i Linux:  
. Stående katalog  
.. Katalogen ett nivå over stående katalog  
~ Innlogget brukers hjemmekatalog (typisk /home/brukernavn )  
~brukernavn En annen brukers hjemmekatalog

Ta ut deler av søkestier: Linux tilbyr kommandoer som tar en søkesti som input og returnerer deler av den. Nyttig i bl.a. shellprogrammering  
● basename søkesti: – Returnerer bare siste element i søkestien  
● dirname søkesti: – Returnerer søkestien frem til siste element

Innholdet i en diskpartisjon (Linux):   
● Boot block: Brukes bare for partisjonen som inneholder OS'et  
● Superblock: Inneholder metadata/parametre for filsystemet på  
partisjonen – type FS, antall diskblokker, administrativ info.  
● Free space management: Info. om ledige/ubrukte diskblokker  
● I-nodes: Array med datastrukturer som inneholder info. om alle  
filene i filsystemet. En i-node i Linux inneholder metadata for én fil.  
● Root dir.: Rotkatalogen, øverste nivå i hele filsystemets katalogtre.  
● Files and directories: Innholdet/dataene i alle filer og kataloger

Filsystemet i Linux: • Linux har støtte for mange ulike filsystemer, også FAT  
• Vanligst: The Extended File System – ext : – Opprinnelig fra MINIX (“leke-Unix” for PC, ca 1990) – ext er inspirert det gamle Unix File System (UFS) – Stor fordel med ext: Journaling\* – ext er i dag betegnelse på en familie av filsystemer – Mest brukt i Linux er ext 3 \*: Se også avsnitt 4.3.6 i læreboken  
● Sentralt element i Linux-filsystemer: i-node : – En i-node lagrer informasjonen om en enkelt fil

i-noder\*   
En i-node i Linux er datastrukturen som lagrer alle metadataene, unntatt filnavnet, om en fil. Diskadresser lagres også internt i i-noden. Alle i-nodene ligger i en tabell lagret på disk, med fast lengde: – Typisk én i-node per 2-8 KB med diskplass – Hvis i-nodetabellen er full er filsystemet fullt – i-nodenummeret til en fil er indeksen i i-nodetabellen – Når en fil fjernes, gjenbrukes i-nodenummeret av neste nye fil  
● Når en fil åpnes/brukes, leses dataene i i-noden inn i RAM  
● Bedre enn FAT: Bare aktive filer vil ha diskadressen lagret i RAM  
\*: i-node er antagelig en forkortelse for “index node”

Sammenhengen mellom kataloger, filnavn og i-noder i Linux: En katalogfil (directory file) i Linux inneholder bare en liste med i-nodenummer og filnavn for hver fil (inkludert katalog-filer for underkataloger) som hører til i katalogen  
Linux-kommandoer som håndterer filer: – Bruker filnavnet som er angitt til å hente ut i-nodenummeret fra katalogfilen – Henter deretter nødvendig informasjon om filen fra disk ved å lese i-nodetabellen  
Merk: Det finnes flere andre måter å implementere kataloger/directories på, se avsnitt 4.3.3 i læreboken

Innholdet i en i-node, for regulære filer: Filstørrelse i antall bytes. Device som filen ligger på. Eier av filen og filens brukergruppetilhørighet. File mode / tilgangsinformasjon. Tidspunkter for siste endring av filen og i-noden. Tidspunkt for siste gang filen ble brukt. Antallet link-filer som peker til denne i-noden. Adresser til de første blokkene der filen fysisk ligger lagret

i-noder og lagring av diskblokker: En i-node inneholder et lite antall (typisk 8-12) pekere direkte til de første diskblokkene på filen  
i-noden har i tillegg indirekte pekere: – En peker til en indirekte blokk der det er lagret flere direkte pekere til diskblokker – Det kan også brukes en peker til en dobbelt indirekte  
blokk (med pekere til indirekte blokker) og en pekere til en trippel indirekte blokk (med pekere til dobbelt indirekte blokker)

File name generation (globbing)  
● Kan angi samlinger med filnavn med spesialtegn (jokernotasjon, wildcards) i shellet:  
\* Matcher en vilkårlig tegnsekvens  
? Matcher et vilkårlig tegn  
[] Et av tegnene mellom firkantparantesene  
[!] Et av tegnene ikke mellom firkantparantesene  
● Shellet leser spesialtegnene og utvider dem til en liste med filnavn, før kommandoen som skal håndtere filene utføres  
Brace expansion: Genererer alle mulige kombinasjoner av en tekst-streng og en liste med ord/items (ligner for-løkker)  
● Eksempel: streng{ord1,ord2,ord3}  
ekspanderer til: strengord1 strengord2 strengord3  
● Kan typisk brukes til å spare skriving og gå gjennom lister av filer og kataloger på en systematisk måte

Brace-expansion: Eksempler  
● Gå gjennom flere kataloger med én kommando:ls /home/bash/test/{foo,bar,baz,cat,dog}  
● “Massenedlasting” av filer fra nettet:  
wget http://domain.com/book/page{1,2,3,4,5,6}.html  
wget http://domain.com/book/page{1..6}.html  
● Skrive ut alle mulige kombinasjoner av en stor bokstav og et siffer (nesting av braces):  
echo {A..Z}{0..9}

Lenker/Links til filer i Linux:  
• Alle filer i Linux må “pekes til” – det må finnes en referanse til filen i OS  
• En katalog inneholder en liste med filnavnene i katalogen og en “peker” til informasjon om hver av filene  
• Pekeren fra katalogen til filen – som egentlig er filens indeks i i-nodetabellen i OS – kalles for en “hard link”  
• Det kan finnes flere “hard links” til en og samme fil fra flere ulike kataloger – en fil kan deles mellom flere kataloger og  
flere brukere\*  
\*: Se også avsnitt 4.3.4 i læreboken: "Shared Files"

Hva skjer når en fil fjernes med rm?  
• Selve filen slettes ikke av rm  
• I stedet slettes filnavnet og hard-linken fra den katalogfilen der filen (logisk) ligger  
• Det kan fortsatt finnes andre katalogfiler som “peker” til den samme filen \*  
• Først når antallet hard links blir lik 0 – ingen katalogfiler inneholder referanse til filen – fjerner OS'et selve filen  
\*: Kommandoen ls –l viser antallet hard links til filer

ln: Opprettelse av en ny lenke til en fil  
ln gammelfil nyfil  
• gammelfil er den originale filen (egentlig søkestien til en hard link til den originale filen)  
• nyfil er den nye lenken som opprettes (i stående katalog hvis ikke annet er angitt)  
• Antall hard links til filen øker med 1  
• Begge de harde lenkene må ligge på samme partisjon i filsystemet

Soft links  
ln -s gammelfil nyfil  
• Opsjonen -s lager en symbolsk lenke, en “soft link”  
• En soft link lagres ikke i en katalogfil  
• Opprettelse av en soft link lager ny fil som bare inneholder en peker til den harde lenken  
• Soft links kan fjernes uten at selve filen den peker til påvirkes  
• Nyttig for å dele filer mellom flere brukere

Beskyttelse og deling av filer: Standard Linux tilbyr filsikkerhet på brukernivå. Alle filer har tilhørighet til en eier, som er en av de registrerte brukerne på systemet. Alle brukere tilhører en eller flere grupper av brukere, med ett eller flere gruppemedlemmer. Alle filer har også tilhørighet til en gruppe av brukere  
● Tilgangsrettighetene til en fil settes separat for: u eier (user) g gruppe (group) o alle andre brukere (others, “rest of the world”)

Tilgang og rettigheter til filer  
En bruker kan ha tre rettigheter til en fil: r Lesetilgang / read permission  
Filen kan åpnes (read-only hvis ikke skrivetilgang) og vises med kommandoer som cat og more  
w Skrivetilgang / write permission Innholdet av filen kan endres, filen kan overskrives  
x Kjøretilgang / execute permission Filen kan eksekveres (kjøres) hvis den er et eksverbart program (ferdig kompilert maskinkode) eller et script som kan utføres av en interpreter

Tilgang og rettigheter til kataloger  
For kataloger (som også er filer) betyr rettighetene:  
r Lesetilgang / read permission Innholdet i katalogen kan listes (med ls )  
w Skrivetilgang / write permission Kan opprette, flytte og fjerne filer i katalogen  
x Kjøretilgang / execute permission Kan gjøre en cd til katalogen

Første tegn angir filtypen ( - betyr “regulær fil”)

chmod – Endre tilgangsrettighetene til en fil  
chmod tilgang filnavn  
● Kan angi endringer i tilgang med + og - :  
u+w Legger til skrivetilgang for eier  
o-r Fjerner lesetilgang for “others”  
a+r Gir lesetilgang til alle brukere (a = all)  
● Kan også angi tilgang eksplisitt med = :  
u=rwx Gir alle rettigheter til eier  
g=rx Gruppen får lese- og skrivetilgang, ikke kjøretilgang  
o= “Others” får ingen tilgang

Representasjon av tilgangsrettigheter  
● Linux lagrer tilgangsrettighetene til en fil som tre samlinger av tre enkle bits (0 eller 1)  
● 1 betyr at man har en rettighet, 0 ingen rettighet  
● Tre bits representerer rwx for hver “brukergruppe”  
● Rettighetene for en brukergruppe kan derfor tolkes som et oktalt tall (i åttetallsystemet)  
● Informasjonen om tilgangen til en fil for alle tre brukergrupper kan representeres med tre oktale siffer  
● Dette er en kompakt skrivemåte for tilgangsrettighetene til en fil som kalles for en “file mode”

File mode  
● File mode: Et oktalt tall der tre sifre brukes til å lagre rettighetene til brukerne: 1. siffer eier av filen u 2. siffer filens gruppe g 3. siffer andre brukere o  
● Hver rettighet har en verdi: 4 read r 2 write w 1 execute x 0 ikke rettighet –

Beregning av file mode  
● Legger sammen rettighetene for hver brukergruppe  
● Får ett unikt tall for hver mulig kombinasjon rettigheter:  
7 = 4+2+1 read/write/execute rwx  
6 = 4+2 read/write rw-  
5 = 4+1 read/execute r-x  
4 = 4 read r--  
3 = 2+1 write/execute -wx  
2 = 2 write -w-  
1 = 1 execute --x  
0 ingen rettigheter ---

Noen eksempler på file modes  
400 r-------- Lesetilgang bare for bruker. For “hemmelige” dokumenter(?)  
644 rw-r--r-- Lese/skrive for bruker, andre lese. Mye brukt.  
664 rw-rw-r-- Bruker og gruppe lese/skrive, andre lese. Vanlig ved fildeling i arbeidsgrupper.  
755 rwxr-xr-x Legger til kjøretilgang på 644 for alle. Svært vanlig for delte kataloger og scripts.  
745 rwxr--r-x Kjøretilgang for eier og “verden”. Alternativ til 755.  
711 rwx--x--x Gruppe og “verden” kan kjøre. Vanlig for kataloger på webservere.  
666 rw-rw-rw- Farlig (number of the beast)  
777 rwxrwxrwx Jeg er idiot