

Thomas Nordli

# Notater til OSYDA50 – Operativsystemer

2. utgave

Horten, Høgskolen i Vestfold, 2013

Kompendium 1/2013



Kompendium 1/2013 Høgskolen i Vestfold

©Høgskolen i Vestfold / Thomas Nordli ([tn@hive.no](mailto:tn@hive.no))

ISSN 0808-1328

# Innhold

<b>1</b>	<b>01-Maskinvare-bakgrunn</b>	<b>7</b>
1.1	Hoved-deler i datamaskin . . . . .	7
1.2	Instruksjons-syklus . . . . .	9
1.3	Avbrudd . . . . .	13
1.4	Minnehierarkiet . . . . .	15
1.5	I/O - kommunikasjon . . . . .	18
1.6	Oppgaver: Bli kjent med systemet . . . . .	21
<b>2</b>	<b>OS-overblikk</b>	<b>23</b>
2.1	Repetisjon av maskinvarebakgrunn . . . . .	23
2.2	Målsetning med OS . . . . .	23
2.3	Operativsystemers utviklingshistorie . . . . .	24
2.4	Viktige teoretiske fremskritt i OS-utviklingen . . . . .	29
2.5	Utvikling mot moderne OS . . . . .	32
2.6	Eksempler . . . . .	32
2.7	Oppgaver . . . . .	33
<b>3</b>	<b>03-linux-intro</b>	<b>37</b>
3.1	Repetisjon OS-overblikk . . . . .	37
3.2	Løsningsforslag til oppgave 2.1 . . . . .	37
3.3	Historie/bakgrunn . . . . .	39
3.4	Filbehandling . . . . .	41
3.5	Redirigering . . . . .	43
3.6	Bruk av skall (shell) . . . . .	45
3.7	Oppgave 3 . . . . .	50
<b>4</b>	<b>04 - Shell scripting</b>	<b>51</b>
4.1	Repetisjon . . . . .	51
4.2	Shell-skripting . . . . .	52
4.3	Førgrunn/bakgrunn . . . . .	55
4.4	Oppgaver . . . . .	56
<b>5</b>	<b>05 - Filer og filsystemer</b>	<b>59</b>
5.1	Løsningsforslag . . . . .	59
5.2	Generelt om filsystem . . . . .	62
5.3	Filhåndtering i Linux . . . . .	66
5.4	Oppgaver . . . . .	71

<b>6</b>	<b>06 - Filsystem-sikkerhet og brukere i Linux</b>	<b>75</b>
6.1	Løsningsforslag . . . . .	75
6.2	Filsystem-sikkerhet . . . . .	79
6.3	Brukere i Linux . . . . .	81
6.4	Oppgaver . . . . .	84
<b>7</b>	<b>07-prosesser</b>	<b>87</b>
7.1	Løsningsforslag . . . . .	87
7.2	Om prosesser . . . . .	90
7.3	Representasjon/beskrivelse av prosesser . . . . .	91
7.4	Noen tilstands-modeller . . . . .	94
7.5	Kontroll av prosesser . . . . .	96
7.6	Alternativer for operativsystemets kjøring . . . . .	102
7.7	Oppgaver . . . . .	103
<b>8</b>	<b>08-Tråder</b>	<b>105</b>
8.1	Løsningsforslag . . . . .	105
8.2	Vi repeterer: To definisjoner av prosess: . . . . .	106
8.3	Om prosesser og tråder . . . . .	106
8.4	Hvorfor tråder? . . . . .	108
8.5	Fordeler med tråder fremfor prosesser . . . . .	108
8.6	Handlinger som involverer alle tråder i en prosess . . . . .	109
8.7	Trådenes tilstand . . . . .	109
8.8	Operasjoner knyttet til endring av tråders tilstand . . . . .	109
8.9	Tråder på "user level" eller "kernel level" . . . . .	109
8.10	Tråder i Linux . . . . .	111
8.11	Oppgaver . . . . .	112
<b>9</b>	<b>10 Samtidig behandling</b>	<b>115</b>
9.1	Produsent-konsument-problemet (semaforer nå igjen...) . . . . .	115
9.2	Vranglås . . . . .	118
9.3	"Message Passing" . . . . .	120
9.4	Oppgaver . . . . .	128
<b>10</b>	<b>11 - minne og signaler</b>	<b>131</b>
10.1	Løsninger . . . . .	131
10.2	Repetisjon: Pipes . . . . .	133
10.3	Signaler . . . . .	135
10.4	Minnehåndtering . . . . .	139
10.5	Oppgaver . . . . .	144
<b>11</b>	<b>12 Sideveksling, virtuelt minne og sikkerhetstrusler</b>	<b>145</b>
11.1	Kort repetisjon fra forrige gang . . . . .	145
11.2	Sideveksling (paging) . . . . .	146
11.3	Virtuelt minne . . . . .	147
11.4	Sikkerhetstrusler . . . . .	149
11.5	Oppgaver . . . . .	151

<b>12 13 - Sikkerhetstiltak (og litt mer om trusler)</b>	<b>153</b>
12.1 Autentisering . . . . .	153
12.2 Tilgangskontroll . . . . .	155
12.3 IDS - Intrusion Detection System . . . . .	156
12.4 Antivirus . . . . .	157
12.5 forsvar mot buffer overflyt-angrep . . . . .	158
12.6 Fysiske tiltak . . . . .	158
12.7 Redundans . . . . .	159
12.8 BIOS . . . . .	159
12.9 Sikkerhet på linux . . . . .	159
12.10 Et tenkt tilfelle: Innbrudd i fire akter (kursorisk pensum) . . . . .	162
12.11 Oppgaver . . . . .	167
<b>I Obligatoriske oppgaver</b>	<b>171</b>
A Obligatorisk oppgave 1 - 2011	173
B Obligatorisk oppgave 2 - 2011	175
<b>II Eksamensoppgaver</b>	<b>177</b>
C Eksamensoppgaver høst 2008 med løsningsforslag	179
D Eksamensoppgaver høst 2009 med løsningsforslag	187
E Eksamensoppgaver høst 2010 <i>uten</i> løsningsforslag	195
F Eksamensoppgaver høst 2011 <i>uten</i> løsningsforslag	199
G Oppgavesett kontinuasjonseksamen 2011 <i>uten</i> løsningsforslag	201
H Oppgavesett siste gangs eksamen 2012 <i>uten</i> løsningsforslag	203

## Om skriftet

Skriftet *Notater til OSYDA50, andre utgave* er et kompendium som inneholder undervisningsmateriale. Undervisningsmaterialet er produsert i forbindelse med undervisning i faget *Operativsystemer.* , høsten 2011. Faget ble tilbudt som obligatorisk fag, for ingeniørstudenter med *grunnleggende ferdigheter i programmering.* Det var også forventet at de fulgte kurset *Algoritmer og datastrukturer* parallellt.

Kompendiet er en sammenstilling av:

- notater
- eksempler
- oppgaver
- løsningsforslag

, som ble brukt i forbindelse med undervisningen.

## Om forfatteren

Forfatteren, Thomas Nordli, har vært fagansvarlig for kurset. Han er ansatt som høgskolelektor ved *Fakultet for teknologi og maritime fag* ved *Høgskolen i Vestfold*, hvor han underviser innen emnene *operativsystemer, databaser, nettverk, kybersikkerhet* og *programmering.* I tillegg er han lærer og kurshefte-forfatter for NKI-kurset *Operativsystemet Linux.*

## Endringer fra første til andre utgave

Foruten retting av feil og små justeringer, inkluderer denne utgaven flere oppgaver.

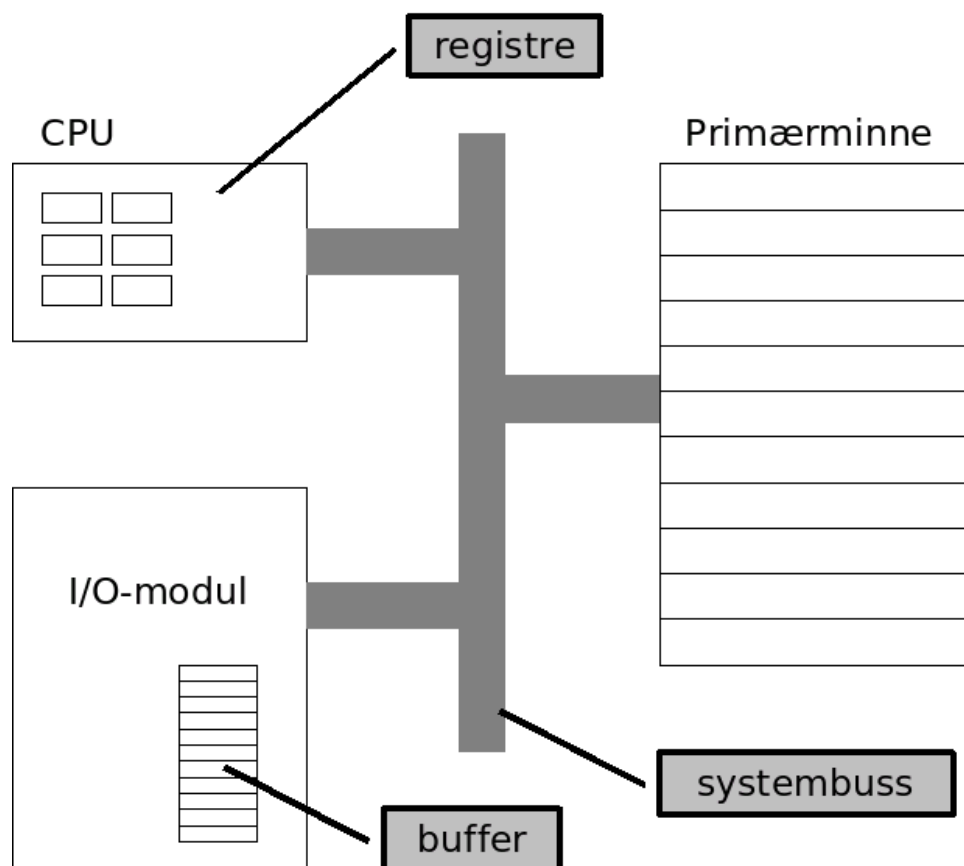
# Kapittel 1

## 01-Maskinvare-bakgrunn

- Et operativsystem kan anses som programvare som kontrollerer maskinvaren i et datasystem, og tilbyr et forenklet grensesnitt mot datasystemets tjenester. I denne forelesningen ser vi nærmere på maskinvaren.

### 1.1 Hoved-deler i datamaskin

figur



cpu - kontrollerer og beregner

- Et register består av en rekke flip-flops

## Typen av prosessor-registre

### registre tilgjengelig for brukere

#### tilgjengelige via maskinkode-instruksjoner

- generelle
- spesielle (f.eks. for flytalls-/heltalls-operasjoner)
- adresseregistre
- stakkpeker
- segmentregister
- indeksregister
- dataregistre

### kontroll- og status-registre

- MAR - Memory Address Register
- MBR - Memory Buffer Register
- I/OAR - Input/Output Address Register
- I/OBR - Input/Output Buffer Register
- PC - Program Counter
- IR - Instruction Register

### primær-minne

- "et sett av lokasjoner definert av sekvensielt nummererte adresser"
- Data kan skrives til og leses fra hver enkelt adresserbare lokasjon
- inneholder både program (instruksjoner) og data

### I/O moduler

- flytter data mellom datamaskin og eksternt miljø (f.eks. sekundært minne - disk)
- Inneholder bufre for mellomlagring
- Systembuss - kommunikasjonskanal mellom CPU, minne og I/O

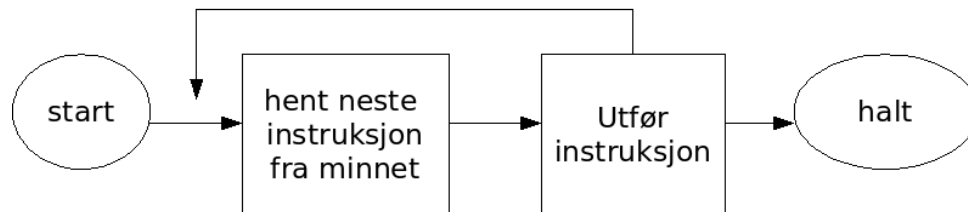


**Spørsmål:**

- Kan du se noen farer/utfordringer med at data og instruksjoner befinner seg i samme lager?

**1.2 Instruksjons-syklus**

figur

**Fire typer instruksjoner**

- Prosessor ↔ primærminnet
- Prosessor ↔ I/O
- Behandling av data
- Kontroll (endre instruksjons-sekvensen)

**Eksempel: Programkjøring** (basert på eksempelet i seksjon 1.3 i pensumboka (Operating Systems Internals and Design Principles, Stallings 2009).

**Eksempelsystemet har**

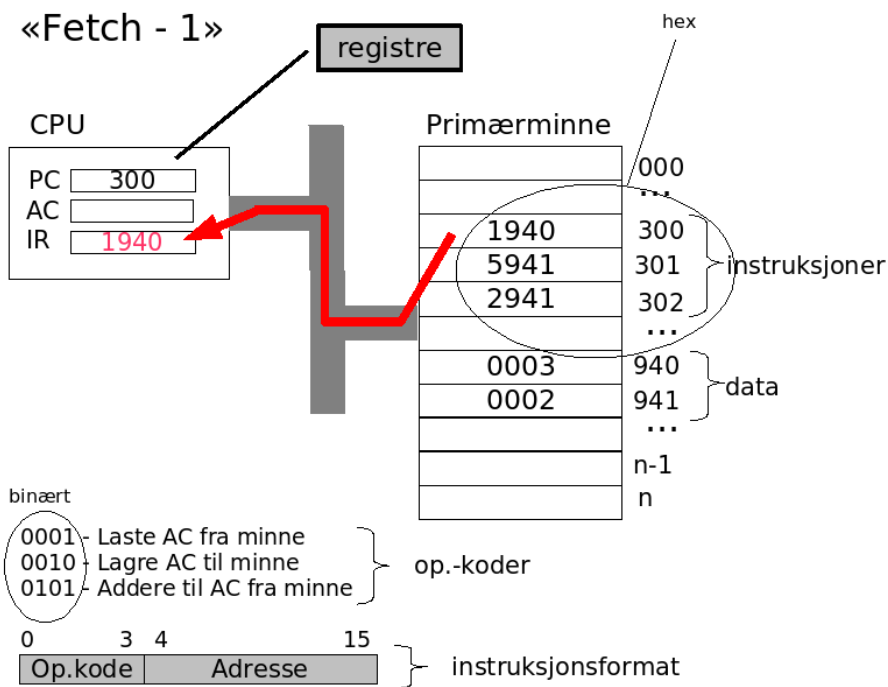
**tre registre**

- PC – programteller (innholder adressen til neste instruksjon)
- AC – akkumulator (generelt registre)
- IR – inneholder instruksjonen som sist ble hentet fra primærminnet

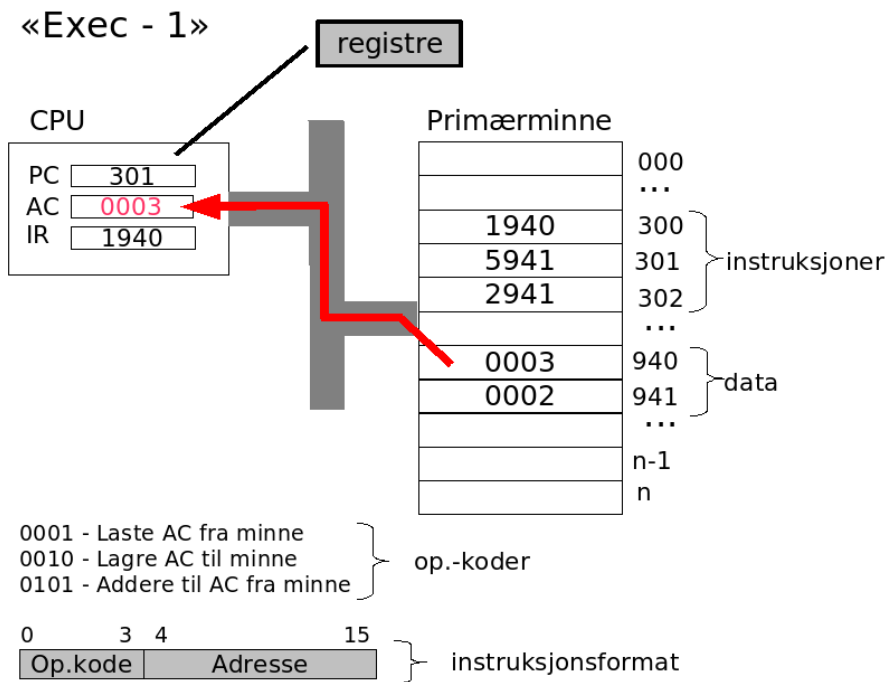
**tre 16 bits instruksjoner**

- Laste AC fra minnet
- Lagre AC til minnet
- Addere AC med minnelokasjon og skrive svaret tilbake i AC

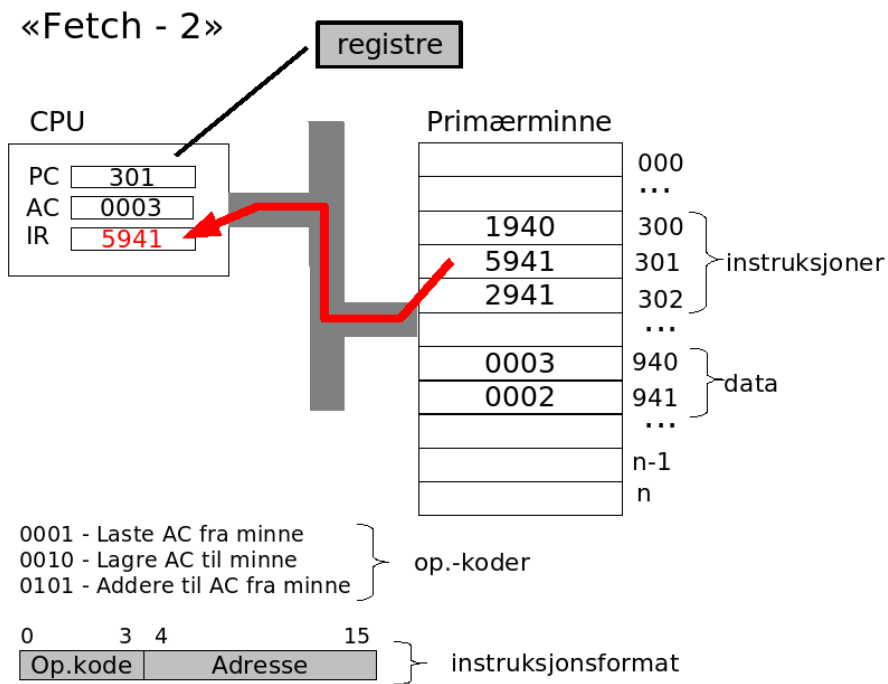
Fetch 1



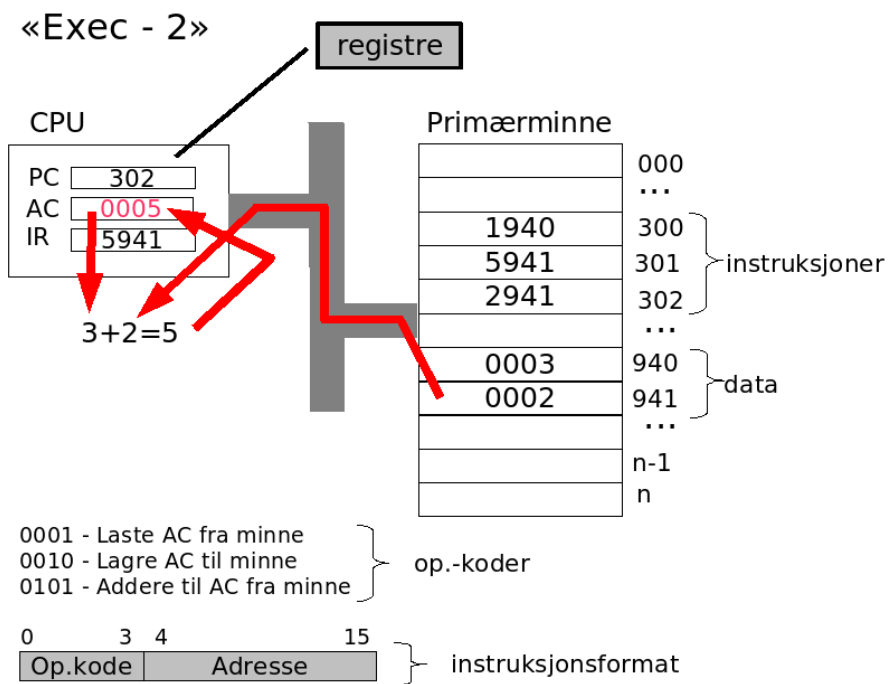
Exec 1



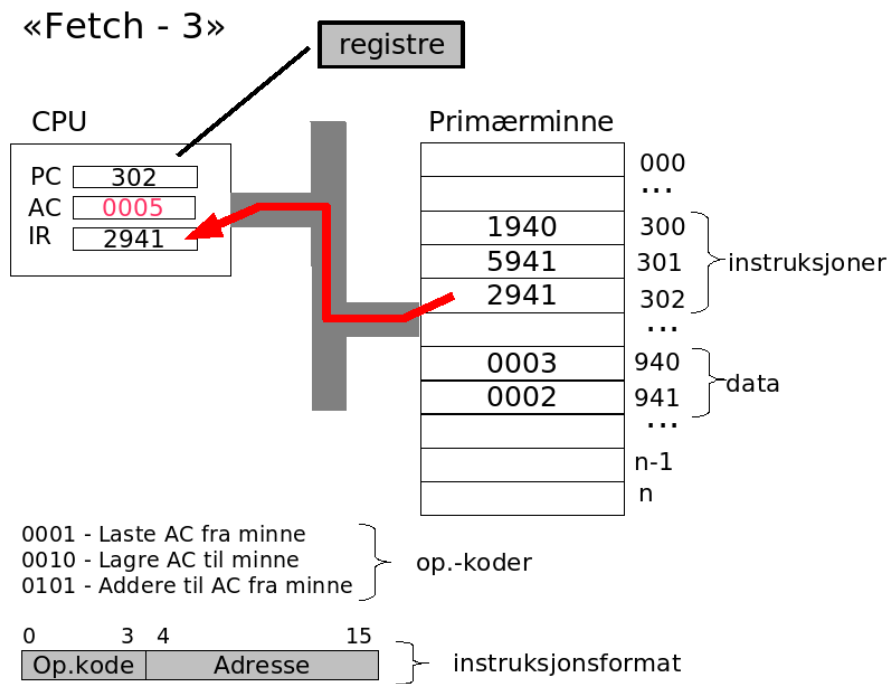
Fetch 2



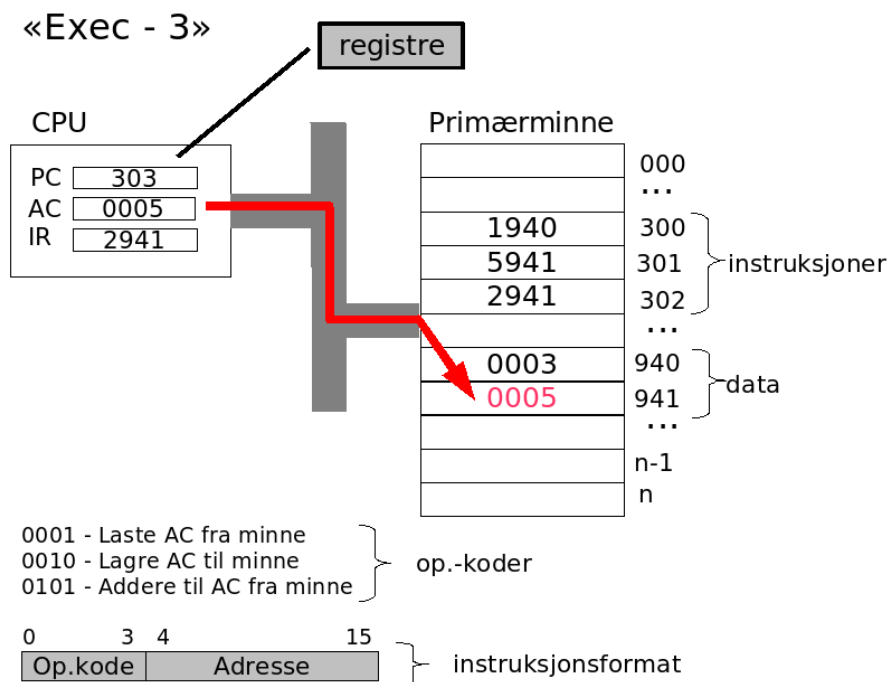
Exec 2



## Fetch 3



## Exec 3



## 1.3 Avbrudd

- Mekanisme som gjør det mulig for andre moduler, (f.eks. I/O-moduler) å avbryte den normale sekvensen (fetch execute).
- Hensikten er primært å øke prosessorens utnyttelsesgrad – den slipper å vente.

### Typer av avbrudd

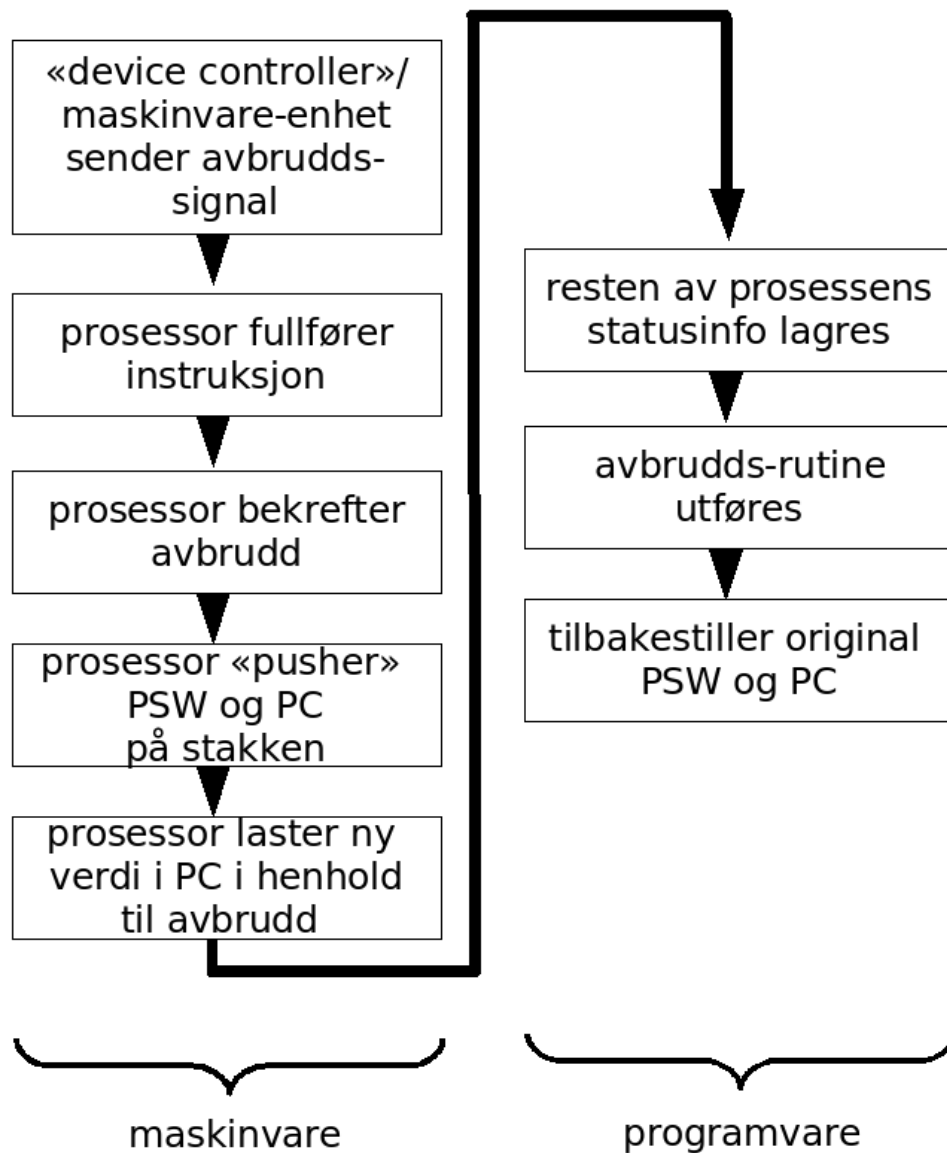
- Avbrudd ved maskinvarefeil
- I/O-avbrudd
- Avbrudd ved utløp av "timer"
- Programvareavbrudd

### Eksempel

- Regnestykket under gir en ide om hastighetsforskjeller mellom CPU og I/O-enheter
- CPU: 1GHz -> 1E9 instruksjon/sekund
- Harddisk: 7200 rpm -> 4 ms på en halv runde
- -> CPU er 4 mill. ganger raskere enn HD

## Utførelse av avbrudd

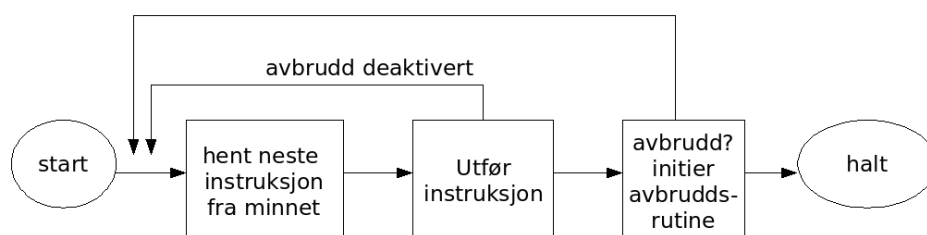
figur



- (basert på figur 1.10 i Operating Systems Internals and Design Principles, Stallings 2009)

## fetch and execute m/avbrudd

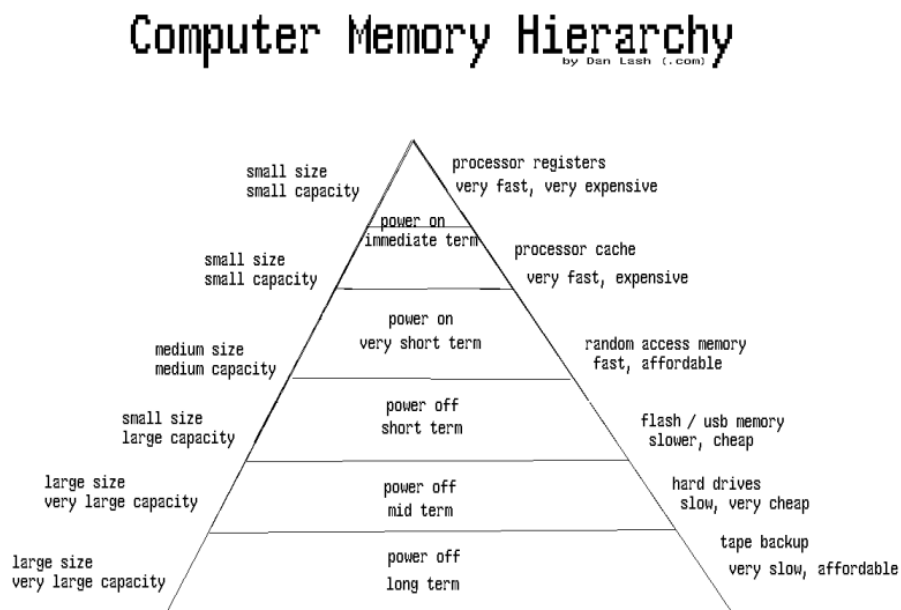
figur



- (basert på figur 1.7 i Operating Systems Internals and Design Principles, Stallings 2009)

## 1.4 Minnehierarkiet

figur



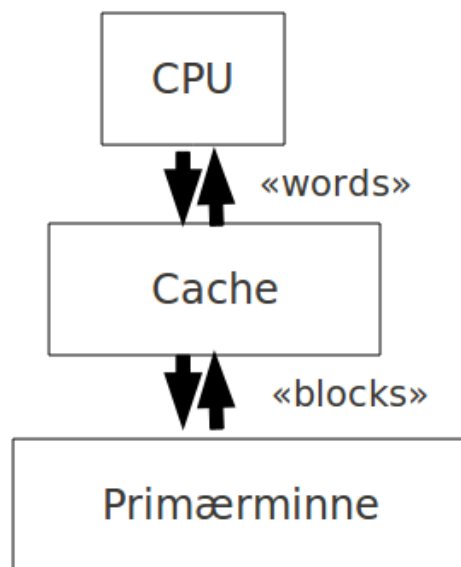
- (bildet er bildet er offentlig eiendom – ”in the public domain”)

### Nedover i hierarkiet

- Lavere kostnad
- Økende kapasitet
- Økende ”access time”
- Sjeldnere i bruk

## hurtigminne - cache

figur

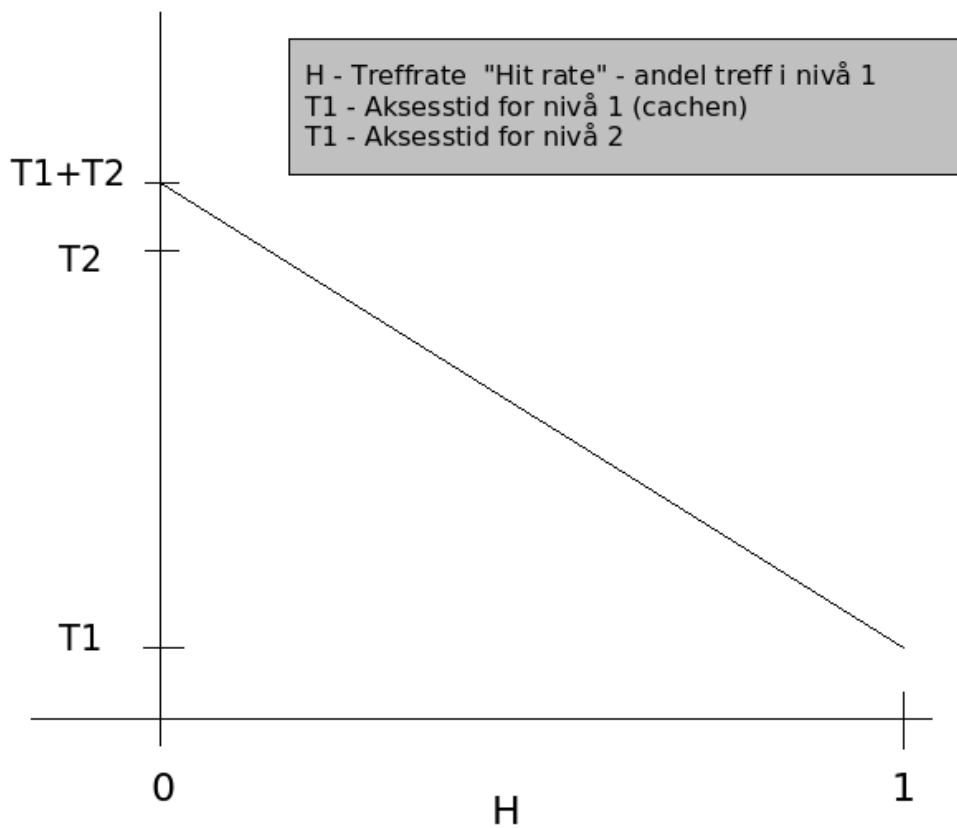


- Baserer seg på antagelsen om at vi har en tendens til at påfølgende referanser er til nærliggende lokasjoner ("Locality of Reference" )



## Treffrate

### Figur



- (basert på figur 1.15 i Operating Systems Internals and Design Principles, Stallings 2009)
- Gjennomsnittlig accesstid som funksjon av treffraten i førstenivå-cachen.

### Eksempel

- Regnestykket under er hentet fra eksemplet på s. 27 i pensumboka (Operating Systems Internals and Design Principles, Stallings 2009).
- $H=0.95$
- $T1=0.1 \text{ ms}$
- $T2=1 \text{ ms}$
- $(H)(T1)+(1-H)*(T1+T2)=$
- $(0.95)(0.1\text{ms})+(0.05)(0.1\text{ms}+1\text{ms})=$
- $0.095+0.055=0.15\text{ms}$

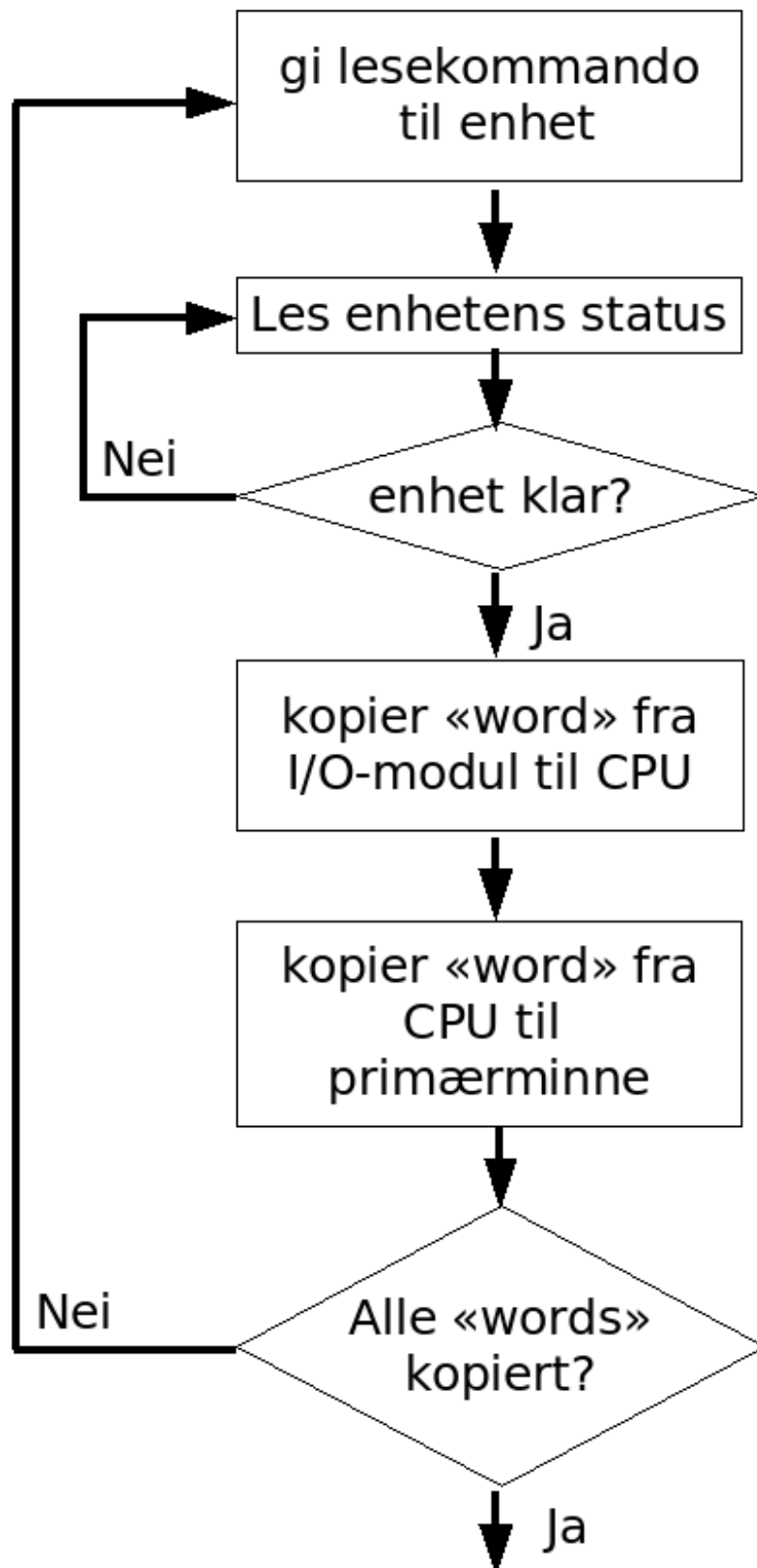
## 1.5 I/O - kommunikasjon

### Programmert I/O

- I/O modul setter bits i statusregister - ikke noe signal til prosessor når den er ferdig med en operasjon

- prosessor: Kontrollerer, sjekker status og overfører

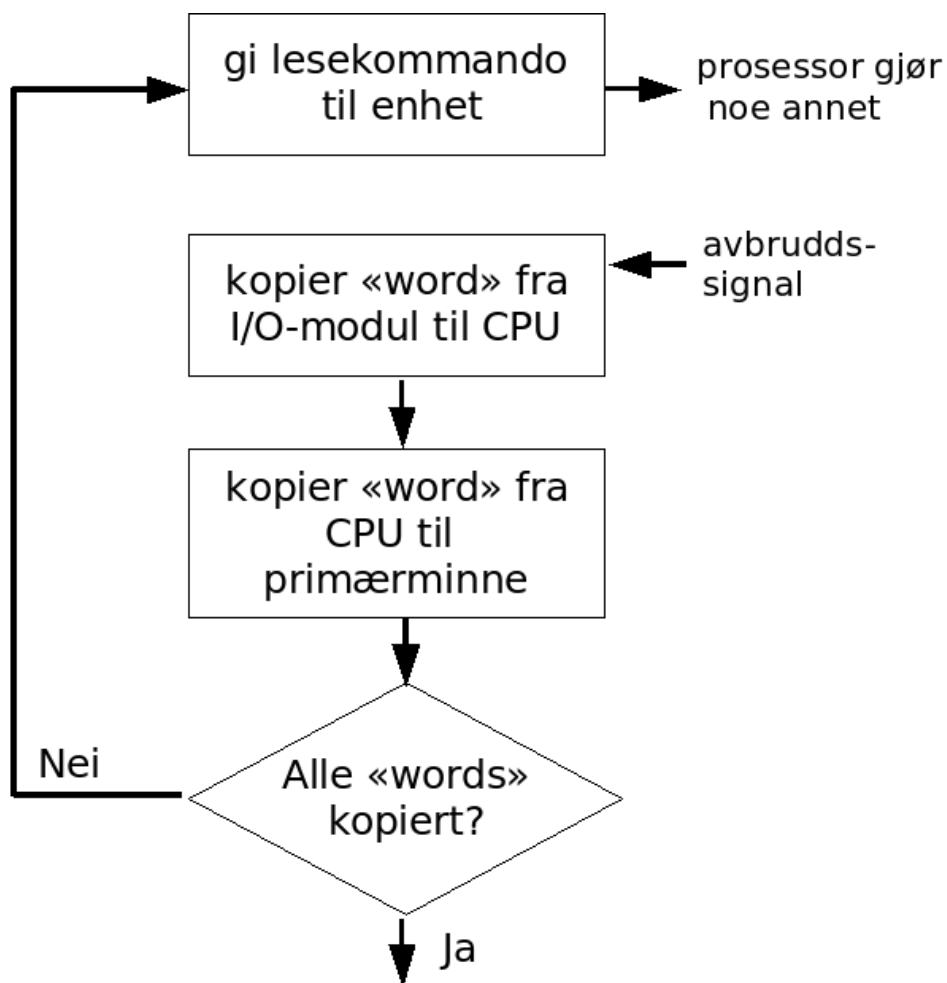
figur



- (basert på figur 1.19 i Operating Systems Internals and Design Principles, Stallings 2009)

## Avbruddsdrevet I/O

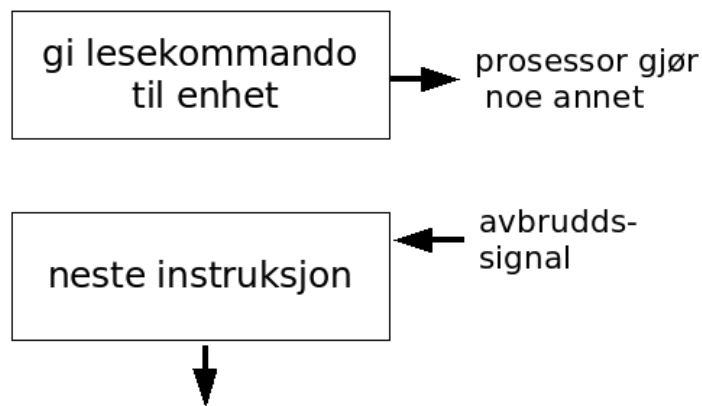
figur



- (basert på figur 1.19 i Operating Systems Internals and Design Principles, Stallings 2009)

## DMA (Direkte minnetilgang - "Direct Memory Access")

figur



- (basert på figur 1.19 i Operating Systems Internals and Design Principles, Stallings 2009)

## 1.6 Oppgaver: Bli kjent med systemet

### Oppgave 1.1: Aktivering

Kontakt faglærer og få kontoen på [debbie.hive.no](http://debbie.hive.no) aktivert.

### Oppgave 1.2: Innlogging

#### Innlogging fra windows

- Logg inn på [debbie.hive.no](http://debbie.hive.no) fra en maskin med windows ved hjelp av terminalemulatoren putty.

**Forsøk å få starte editoren emacs, slik at den kommer opp grafisk modus i et eget vindu. Kommando: emacs. For at emacs skal komme opp i grafisk modus, må**

- 1) Xming (eller tilsvarende) kjøre på windows-maskinen og
- 2) X11 forwarding må være slått på i putty (SSH->X11).

#### Innlogging fra linux/unix/mac

- Logg inn på [debbie.hive.no](http://debbie.hive.no) fra en maskin med ved hjelp av ssh.
- Kommando: `ssh -X brukernavn@debbie.hive.no`
- Forsøk å få starte editoren emacs, slik at den kommer opp grafisk modus i et eget vindu.

### Oppgave 1.3: Et C-program

- Bruk editoren emacs til å skrive et program i C. Kompiler programmet. Kommando: gcc -Wall filnavn.c
- Kjør programmet. Kommando: ./a.out

### Oppgave 1.4: Hvem der?

Finn ut hvem er pålogget. Kommandoer:

- finger
- who
- w
- Skriv melding til pålogget bruker. Kommando: write brukernavn

# Kapittel 2

## OS-overblikk

### 2.1 Repetisjon av maskinvarebakgrunn

Hva er

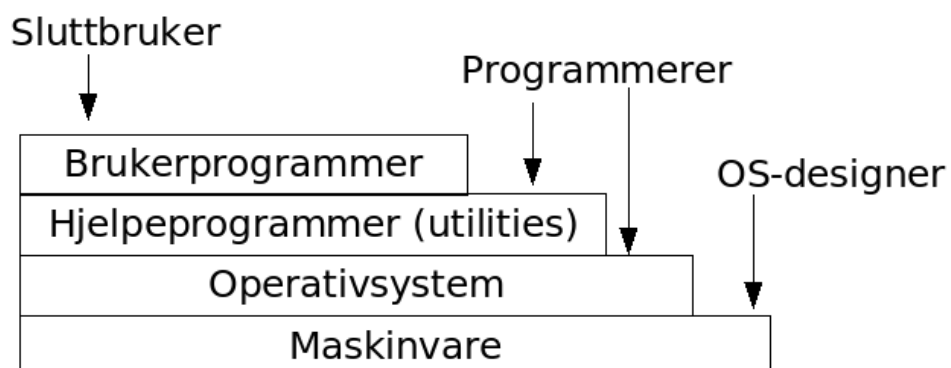
- hoved-delene i en datamaskin?
- instruksjons-syklus?
- minnehierarkiet?
- de tre I/O-kommunikasjonsmetodene?

### 2.2 Målsetning med OS

**Bekvemmelighet - det skal være enkelt å benytte datasystemet**

- De viktigste systemprogrammene kan sies å utgjøre operativsystemet. Resten kalles hjelpeprogrammer (utilities).

Figur



Typiske områder hvor OS gir tjenester

- programkjøring
- kontrollert filtilgang

- tilgangskontroll til systemet
- feildeteksjon og respons
- ”bokføring” - bruks-statistikk og overvåking av ytelesesparametre
- (programvareutvikling - vanligvis ”utilities”)

**Effektivitet - mest mulig effektiv utnyttelse av ressursene i data-systemet**

**Resursser**

- Prosessorer
- Minne
- I/O-enheter (via I/O-kontrollere)
- Operativsystemet kontrollerer flytting, lagring og behandling av data

**Skiller seg fra andre kontrollsystemer**

- er ikke eksternt system - er et vanlig program
- oppgir ofte kontrollen og avhenger av prosessoren for å få den tilbake

**Fleksibilitet**

**Et OS bør kunne utvikles over tid for å**

- håndtering av nye typer maskinvare/tjenester
- innføring av feil-/sikkerhets-fikser

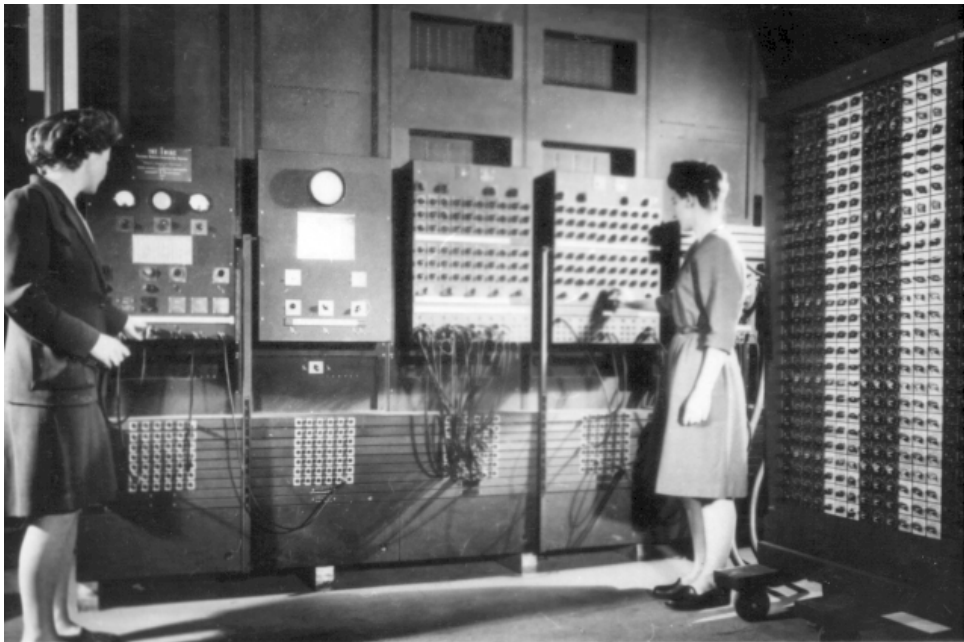
## **2.3 Operativsystemers utviklingshistorie**

**seriell prosessering**

- 1940-1950: uten OS



Foto ENIAC med to operatører (U. S. Army Photo - <http://ftp.arl.army.mil/mike/comphist/>)



- Brukeren interagerer direkte med maskinvaren - brytere
- Programmene, i form av maskinkodeinstruksjoner, lastes fra innlesings-enheter (f.eks. hullkortleser)
- Fordeling av kjøretid: Reservering av tid på papirark

#### Oppsettingstiden for hver jobb

- lasting av programvare fra sekundærlager (magnetbånd/hullkort) til primærlager
- lenking
- lagring av objekt-program (kompilert program) på sekundærlager
- ... mye manuelt arbeid - bytting av magnetbånd/hullkortbunker

#### Kjøretiden

treg maskinvare -> kjøretiden lang (timer - dager)

#### Effektiviseringstiltak: Felles programvare

- Bibliotek med fellesfunksjoner

”Linker”

”Loader”

**”Debugger”**

- I/O-driver-rutiner

**enkle system for satsvis ”batch” behandling (”simple batch system”)****Maskinvaren ble raskere**

- kjøretiden ble vesentlig kortere
- oppsettingstiden ble ikke tilsvarende forkortet (p.g.a. mye manuelle rutiner)
- Dyr maskinvare → ønske om å maksimere prosessorens utnyttelsesgrad.
- Konseptet ved enkle batch-systemer øker utnyttelsesgraden ved å forkorte jobbenes oppsettningstid.

**Første OS**

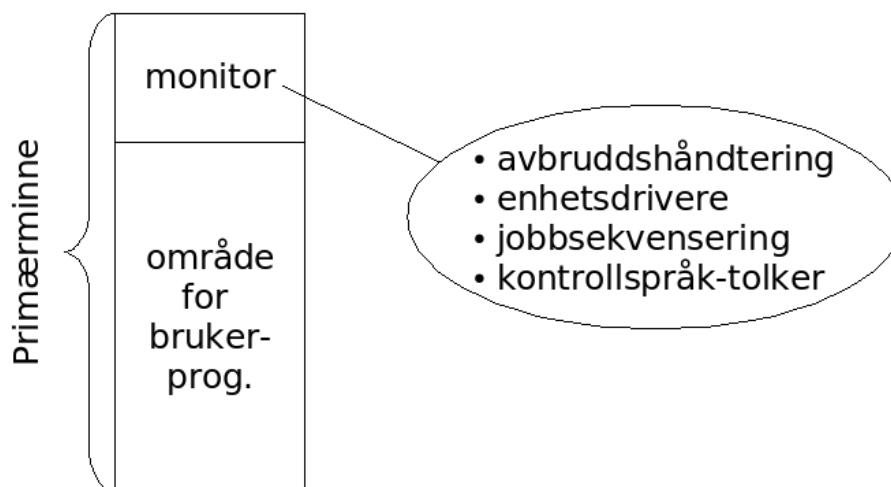
- i midten av femtiårene

for IBM 701 <http://www.computer-history.info/Page4.dir/pages/IBM.701.dir/index.html>

- av General Motors

**monitor**

- program som laster og kjører jobber fra kø på innlesningsenhet
- etter avslutning av hver jobb, tilbakeføres kontrollen til monitor
- den delen av monitoren som alltid er i minnet kalles ”resident monitor”

**Figur av monitor i minnet**

- monitor kan også innbefatte av hjelpe-programmer/-funksjoner

**Brukeren er ikke lenger direkte i kontakt med maskinvaren - kun via operatør.**

- operatør tar i mot jobber
- operatør ordner jobbene i bunker - "batches"
- operatør legger i jobbene i kø
- monitor laster og kjører jobber fra kø

### **JCL - Job Control Language**

**Eksempelet i seksjon 2.2 i pensumboka (Operating Systems Internals and Design Principles, Stallings 2009) er gjenfortalt under.**

```
$JOB  ( jobb starter her )
$FTN  ( fortrankompilator lastes og startes )
..    ( fortran-kildekode - som leses av kompilatoren og
..    kompileres. Resultatet, objektprogrammet,
..    lagres på sekundærminne )
$LOAD ( objektprogrammet lastes fra sekundærminne til primærminne )
$RUN  ( kjøring av objektprogrammet starter )
..    ( eventuelle data for inlesning av programmet )
$END  ( job avsluttes - monitor kjøres )
```

### **ønskelig HW-støtte for monitor**

- beskyttelse av minnet
- tidtaker
- privilegerte instruksjoner
- avbrudd (ikke tilgjengelig i tidlige maskiner)

### **operasjonsmoda**

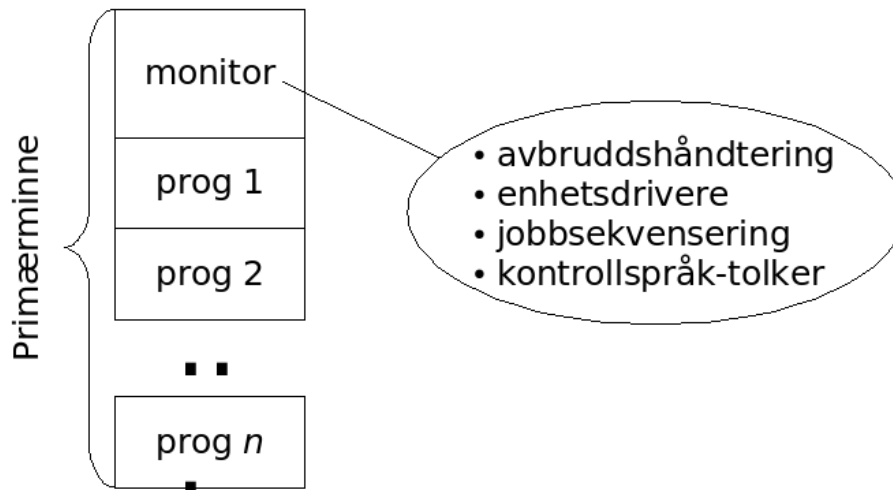
- usermode
- kernelmode

### **multiprogramkjørende system for satsvis behandling ("multiprogrammed batch system")**

- I/O-enheter tregere enn prosessor -> lav utnyttelsesgrad

flere jobber holdes i minnet samtidig

Figur



- ved I/O forespørsel velges enn annen jobb for kjøring
- ved fullføring av I/O velges ny jobb for kjøring

HW-krav

- I/O-avbrudd
- DMA
- Resultat: Høyere grad av ytelse -> større gjennomstrømning

Kjøretidsdeling ("time sharing")

- Flere bruker maskinen direkte

Foto av PDP 7 på institutt for informatikk ved Universitetet i Oslo



<http://creativecommons.org/licenses/sa/1.0/>

- N brukere; hver bruker får  $1/N$  av kapasiteten
- Lav responstid er vesentlig
- En klokke sender avbrudds-signal ved faste intervaller
- Ved klokkeavbrudd velges ny bruker

**Eksempel OS: CTSS - Compatible Time-Sharing System**

- Tidlig "time sharing"-system:
- Inspirasjonskilde for utviklerne av UNIX
- Klokke avbrøt hvert 0,2 sek.

## 2.4 Viktige teoretiske fremskritt i OS-utviklingen

### Prosesser

#### Ulike definisjoner

- Et program under utførelse
- En instans av et program som kjører på en datamaskin
- Entitet som kan tildeles og kjøres på en prosessor

**En enhet av aktivitet karakterisert av**

- en enkel sekvensiell tråd av utførelse
- en tilstand
- tilhørende system resursser
- Begrepet først brukt under utvikling av Multics på 60-tallet

**Problemer ved utviklingen av "batch"-, "time share" og "real time"-OS**

- feilaktig synkronisering
- feil ved gjensidig utelukkelse
- uberegnelig utfall av programkjøring
- vranglåser
- Problemene imøtegås med systematisk måte å kontrollere programkjøringene

**Prosess består av**

- et kjørbart program
- tilhørende data

**kjøre-kontekst**

- info nødvendig for å kontrollere prosessen
- f.eks. ulike prosessor-registre, prioritet, tid-brukt, etc.

<http://debbie.hive.no/osyda50/figurer/typiskProssesimplementasjon.png>

**Minnebehandling****Krav til fleksibelt miljø med støtte for modulær programmering**

- isolasjon av prosesser
- automatisk (og transparent) tilordning av minne (i minnehierarkiet)

**støtte for modulære programmer**

- definere moduler
- opprette/slette moduler
- endre størrelse på moduler
- Beskyttelse og tilgangskontroll
- Langtidslagring

### **Filsystemer**

- langtidslagring
- lagring i navgitte objekter - filer
- filer - enheter/objekter i tilgangskontroll og beskyttelse

### **Virtuelt minne**

- adressere minnet "fra logisk ståsted"
- virtuell adresse
- virkelig/fysisk adresse

[http://debbie.hive.no/osyda50/figurer/virtuell\\_minneadressering.png](http://debbie.hive.no/osyda50/figurer/virtuell_minneadressering.png)

### **Beskyttelse av informasjon og sikkerhet**

- Tilgjengelighet
- Konfidensialitet
- Data-integritet
- Autentisitet/ekthet

### **Skedulering (fordeling) og resurss håndtering**

#### **Hensyn**

- Rettferdighet
- Differensiering mellom klasser

#### **Effektivitet**

- høyest mulig gjennomføringshastighet/"throughput"
- lavest mulig responstid
- flest mulig brukere

<http://debbie.hive.no/osyda50/figurer/multiprogramming.png><http://cs.gmu.edu/cne/workbenches/rndrobin/rndrobin.html>

## Systemstruktur

### Størrelsen på koden blir et problem

- Forsinket levering
- Latente bugs
- Ytelse ofte lavere enn forventet
- modulær
- veldefinerte grensesnitt mellom modulene

### store OS

- millioner-titalls millioner kodelinjer

<http://debbie.hive.no/osyda50/figurer/multiprogramming.png>

## 2.5 Utvikling mot moderne OS

- mikrokjerne
- flertrådskjøring
- symmetrisk multiprosessering
- distribuerte OS
- objektorientert design

## 2.6 Eksempler

### UNIX-arkitektur

<http://debbie.hive.no/osyda50/figurer/unixarkitektur.png> Vi ser på figur 2.14: Generell UNIX-arkitektur <http://debbie.hive.no/osyda50/figurer/unixkjerne.png> Vi ser på figur 2.15: Tradisjonell UNIX-kjerne

### Linux-arkitektur

<http://debbie.hive.no/osyda50/figurer/linuxkjerne.png> Vi ser på fig. 2.18: Komponenter i linux-kjernen

### loadable modules

### dynamisk lenking

- lastes i minnet og lenkes til kjernen under kjøring
- insmod
- fjernes og avlenkes fra kjernen under kjøring
- rmmmod



**”stackable modules”**

- hierarkisk ordnet
- moduler kan forutsette at andre moduler er lastet
- reduserer kodereplikering
- sørger for å ikke fjerne moduler som er i bruk

## 2.7 Oppgaver

### 2.1

Tenk deg at du har et flerprogramkjørende satsvis system med en bunke (batch) med tre jobber:

1.

- Tidfordeling mellom CPU og I/O:
- 00-10s CPU
- 11-50s I/O
- 51-100s CPU

2.

- Tidfordeling mellom CPU og I/O:
- 00-30s CPU
- 31-50s I/O
- 51-70s CPU

3.

- Tidfordeling mellom CPU og I/O:
- 00-40s CPU
- 41-110s I/O
- 111-120s CPU

a)

Regn ut gjennomsnittelig omløpstid (Turnaround time) for batchen. Med omløpstid menes tiden det tar fra en jobb settes opp for kjøring og til den er fullført.

b)

Regn ut gjennomstrømningen (throughput) for batchen. D.v.s. antall fullførte jobber pr. tidsenhet.

c)

Regn ut prosessor-utnyttelsen (processor utilization) for batchen. Utnyttelsesgraden finner du ved å regne ut hvor stor prosent av medgått tid, som prosessoren har vært aktiv.

## 2.2

a)

Logg inn på [debbie.hive.no](http://debbie.hive.no) og start programmet 'vimtutor' ved å skrive 'vimtutor' og trykke enter på kommandolinja. Les og følg instruksjonene.

b)

Bruk deretter programmet 'vi' (eller 'vim') til å besvare oppgavene i "Repetisjon av maskinvarebakgrunn" over. Husk å lagre det du skriver i en fil. Kall filen 2.2.txt.

c)

Bruk programmet 'mail' for å sende det du skrev (og lagret i fila 2.2.txt) til faglærer med kommandoen 'cat 2.2.txt | mail tn'. Kommandoen skrives på kommandolinja på debbie.

## 2.3

- I denne oppgaven skal du sette deg inn i hvordan unix kommandoene 'at' og 'batch' fungerer.

a)

- Logg inn på debbie og åpne manualen til kommandoen 'at' med kommandoen 'man at'. Les hvordan kommandoen 'at' og 'batch' fungerer. For å avslutte lesing av manualen kan du taste 'q'.
- For å lage en pdf-fil med penere formatering av manualen kan følgende kommandoer brukes: `man -t at | ps2pdf - at.pdf`
- For å se på pdf-fila kan følgende kommando brukes: `xpdf at.pdf &`

b)

- Du skal nå bruke det du leste om i a). Sett opp en batch-jobb på debbie som laster ned fila <http://debbie.hive.no/osyda50/pdfs/01.pdf> når prosessorbelastninger er lav. Nedlastingen skal gjøres med programmet 'wget'. Hvordan wget brukes må du selv finne ut av ved å lese i manualen. Gi kommandoen 'man wget' for å lese manualen til wget. Husk at du avslutter manualen ved å taste 'q'.
- Du får utskrift av jobben pr. e-post når jobben er gjort.

- For å lese e-post på debbie, kan du bruke programmet 'mail' (som også har en manualsider).
- Hvis du vil at e-post-meldingene fra debbie skal havne sammen din vanlige e-post, kan du instruere e-post-systemet på debbie til å videresende din post til en annen konto. Dette gjør du ved å lage en tekstfil med en tekst-editor (f.eks. vim). Tekstfila må inneholde e-post-adressen du vil at systemet skal videresende posten til. Du kan også oppgi flere adresser. En adresse pr. linje. Filen må lagres med navnet '.forward'. Du kan da lese e-posten i den e-postleseren du pleier å bruke. Fila '.forward' må ligge i din hjemmekatalog. Hjemmekatalogen er den katalogen du "havner i" når du logger inn.

**c)**

Sett opp en batch-jobb på debbie som sender en e-post til faglærer nøyaktig kl. 1300 i dag. Bruk følgende kommando for å sende e-post til faglærer: 'echo Takk for i dag | mail tn'.

**d)**

Se om du få frem en liste over batch-jobbene dine.



# Kapittel 3

## 03-linux-intro

### 3.1 Repetisjon OS-overblikk

- Hva vil du si er målsetning for et operativsystem?
- Nevn noen viktige teoretiske fremskritt i OS-utviklingen. Forklar også kort hva som menes med dem.
- Nevn noen utviklingstrekk hos moderne operativsystemer.

#### Hva menes med

- seriell prosessering?
- enkle system for satsvis "batch" behandling ("simple batch system")?
- multiprogramkjørende system for satsvis behandling ("multiprogrammed batch system")?
- kjøretidsdeling ("time sharing")?

### 3.2 Løsningsforslag til oppgave 2.1

Tenk deg at du har et flerprogramkjørende satsvis system med en bunke (batch) med tre jobber:

1.

- Tidfordeling mellom CPU og I/O:
- 00-10s CPU
- 11-50s I/O
- 51-100s CPU

**2.**

- Tidfordeling mellom CPU og I/O:
- 00-30s CPU
- 31-50s I/O
- 51-70s CPU

**3.**

- Tidfordeling mellom CPU og I/O:
- 00-40s CPU
- 41-110s I/O
- 111-120s CPU

Tegner opp jobbene.

- c er 10s med CPU-aktivitet
- i er 10s med I/O-aktivitet

```
1: ciiiiiccccc
2: cccciicc
3: cccciiiiic
```

Jeg gjør følgende antagelser:

- systemet har en prosesseringsenhet (enkeltkjernet CPU).
- systemet har en "round robin" fordelingsalgoritme
- de tre jobbene ber om ulike I/O tjenester (ingen køing for I/O-tjenester)

Figuren under viser hvordan de tre jobbene fordeles i tid på prosessoren. Tiden løper fra venstre mot høyre.

- c er 10s med CPU-aktivitet
- i er 10s med I/O-aktivitet
- k er 10s hvor jobben ligger i kø, klart til å kjøre

Den nederste linjen viser hvilken jobb prosessoren jobber med til enhver tid.

```
ciiiikkkcccc---
-ccciikkkkkkkcc-
----ccciiiiic
122233331111223
```

a)

- Regn ut gjennomsnittelig omløpstid (Turnaround time) for batchen. Med omløpstid menes tiden det tar fra en jobb settes opp for kjøring og til den er fullført.

1 bruker 130s

2 bruker 140s

3 bruker 120s

snittet blir 130

b)

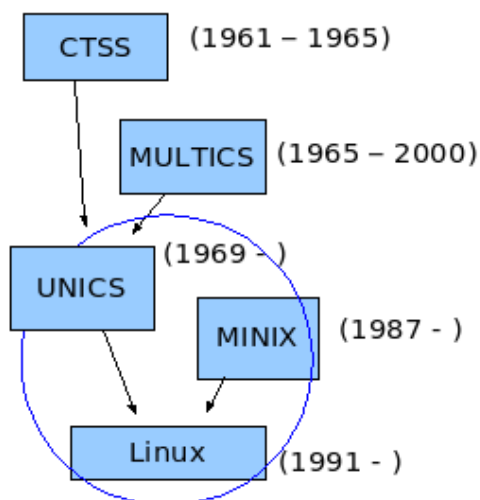
- Regn ut gjennomstrømningen (throughput) for batchen. D.v.s. antall fullførte jobber pr. tidsenhet.
- I løpet av 160 sekunder fullføres 3 jobber.  $3 \text{ jobber} / 160 \text{ s} = 0,02 \text{ jobb/sek.}$

c)

- Regn ut prosessor-utnyttelsen (processor utilization) for batchen. Utnyttelsesgraden finner du ved å regne ut hvor stor prosent av medgått tid, som prosessoren har vært aktiv.
- Ser av figuren at prosessoren har hvert aktiv hele tiden. Utnyttelsesgraden er dermed 100%

### 3.3 Historie/bakgrunn

#### historie



#### Historie

- Hvorfor ble UNIX laget?
- Hvorfor ble Linux laget?
- Hva er forskjellen på

UNIX og LINUX?  
 • Hvordan kan UNIX/Linux fremdeles være aktuelt?

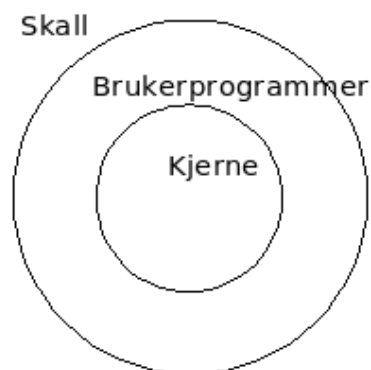
## bakgrunn

### Bakgrunn

- GNU, FSF, GPL
- Distribusjoner
  - Debian
  - RedHat
  - Suse
- Standarder
  - POSIX, LSB

## UNIX-arkitektur

### UNIX-Arkitektur



- API til kjernen består av systemkall



## 3.4 Filbehandling

### UNIX-filtyper

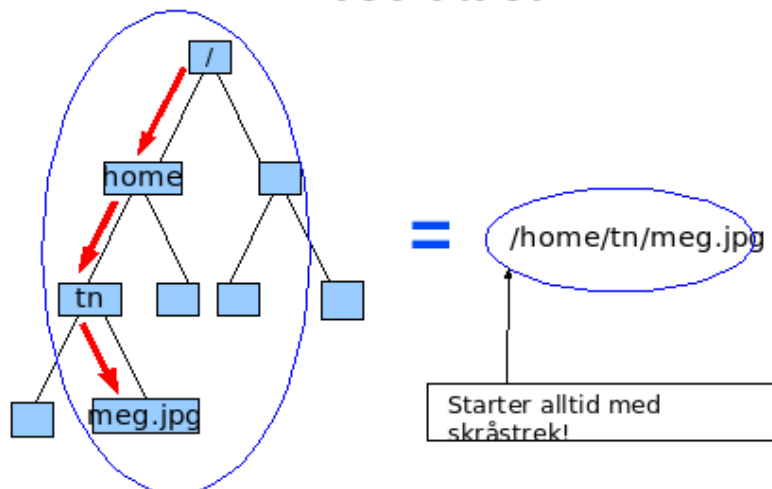
### UNIX-filtyper

1. Vanlige filer
2. Kataloger
3. Karakter-enhetsfiler
4. Blokk-enhetsfiler
5. Symbolske lenker
6. "Local domain sockets"
7. Navngitte rør (FIFO's)

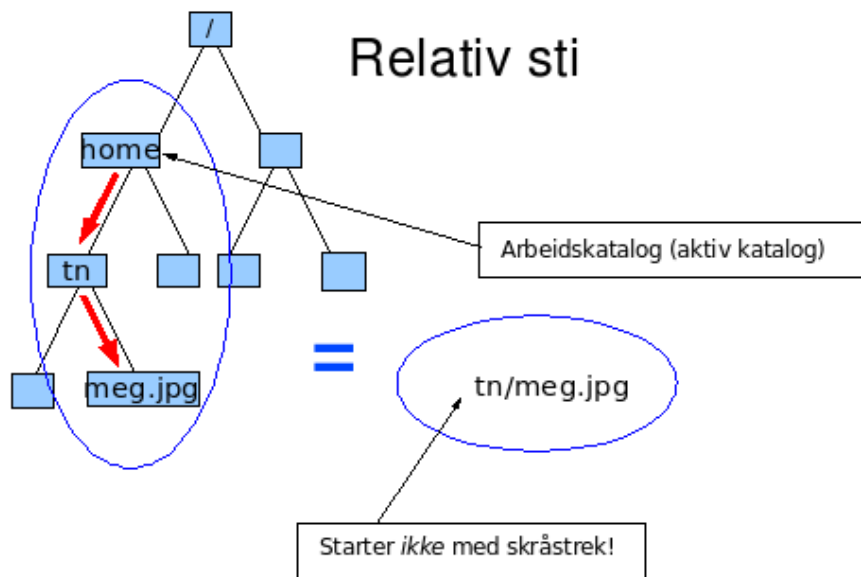
Hvorfor er så mye forskjellig representert som filer?

### Absolutt sti

### Absolutt sti



## Relativ sti



## UNIX-kommandoer

### UNIX kommander

- ls – lister innholdet i en katalog
- pwd – lister aktiv katalog (arbeidskatalog/ «current directory»)
- mkdir – lager katalog
- rmdir – fjerner tom katalog
- rm -r – fjerner katalog med innhold (r'en står for for rekursiv)

## flere unix-kommandoer

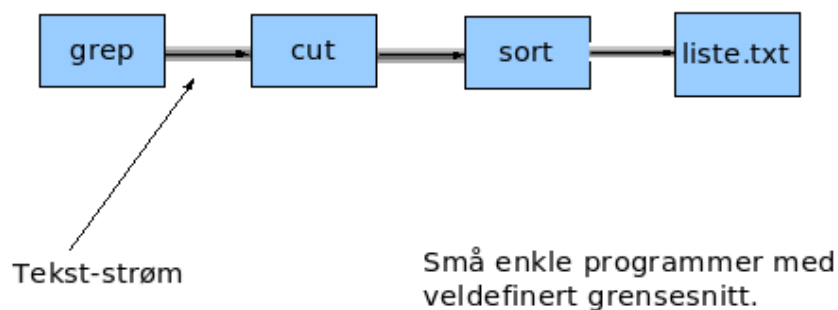
## UNIX-kommandoer

- touch
  - endrer tids-stempel
  - oppretter tom fil hvis filen ikke eksister
- cp – kopierer fil
- mv
  - flytter
  - brukes også ved bytte navn
- rm - fjerner fil

## 3.5 Redirigering

## pipeline

## «Modulbasert»

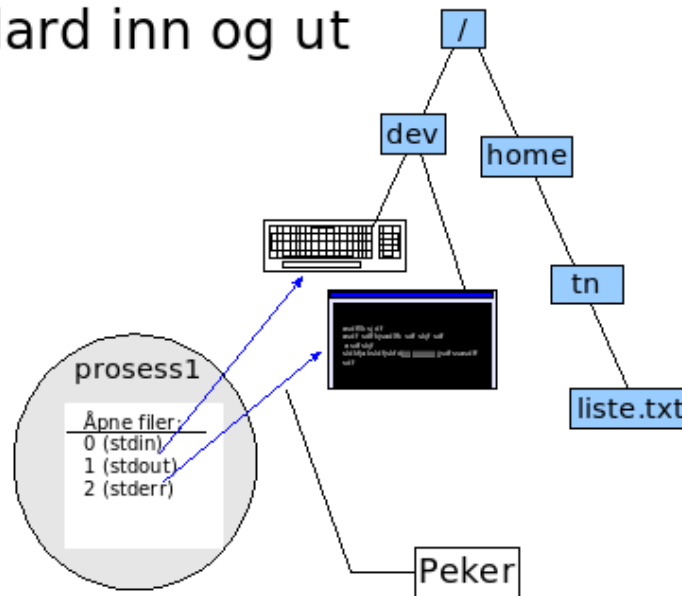


Eksempel:

```
grep Hansen /etc/passwd | cut -f5 -d: | sort > liste.txt
```

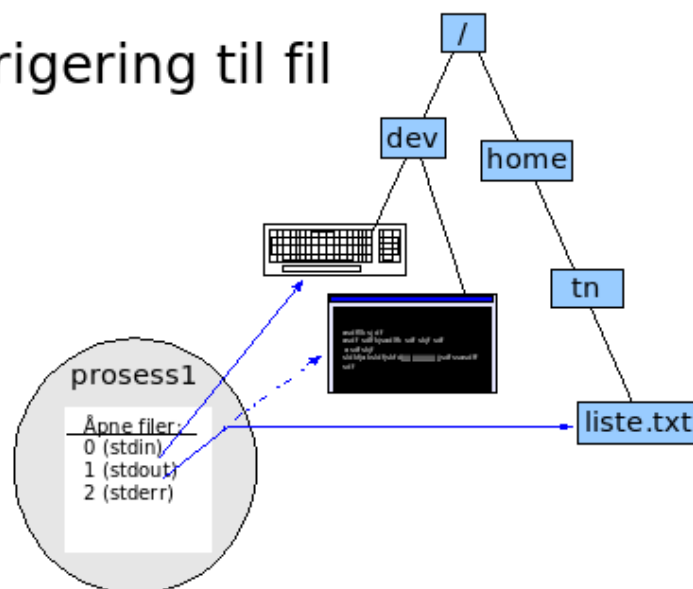
## stdin og stdout

## Standard inn og ut



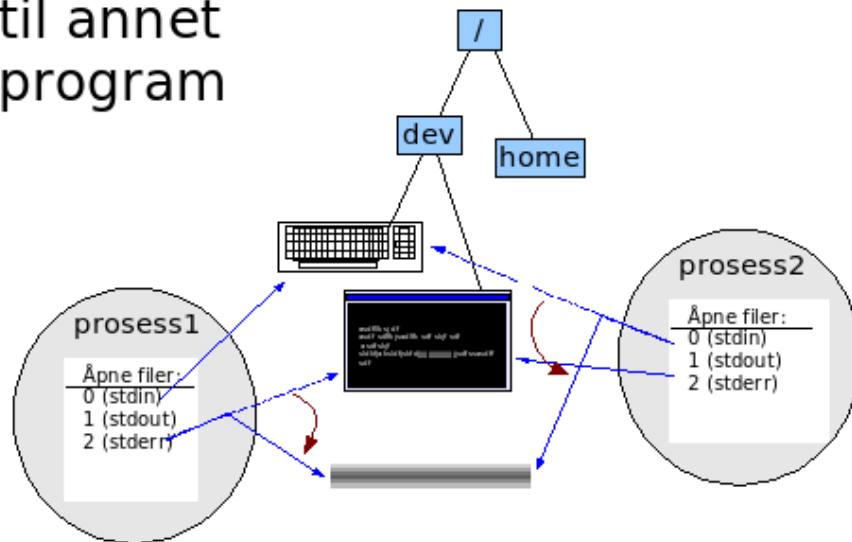
## Redirigering til fil

## Redirigering til fil



## Redirigering til prosess

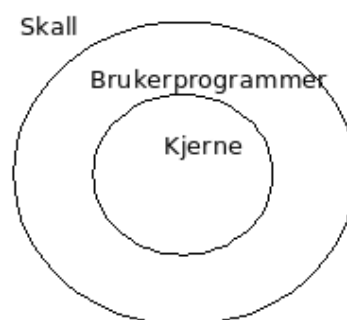
### Redirigering til annet program



## 3.6 Bruk av skall (shell)

Skallets rolle?

### Skallets rolle



Hva er formålet med skallet?

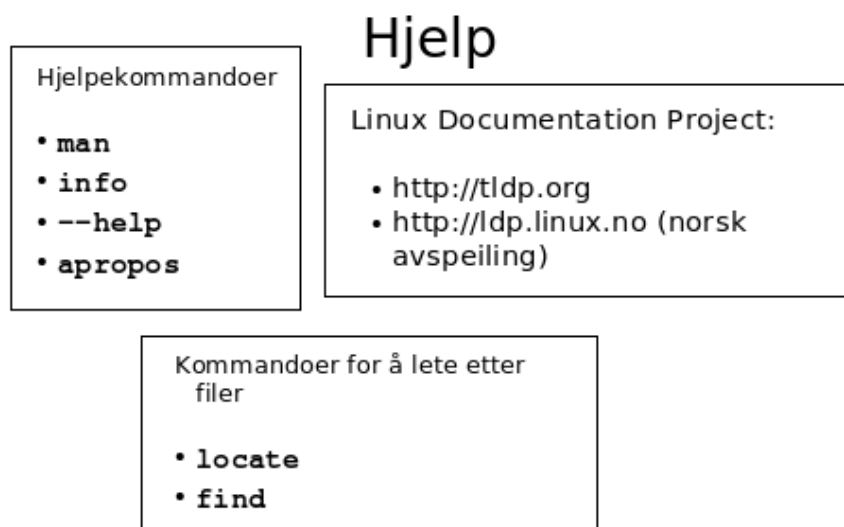
## Ulike skall

# Ulike skall

- **sh** - Bourne shell (skrevet av Steve Bourne)
- **bash** - Bourne again shell (standard skallet i linux)
- **csh** - C shell (mer "C-aktig" syntax enn *bash*)
- **mc** - Midnight Commander (Norton Commander klone)

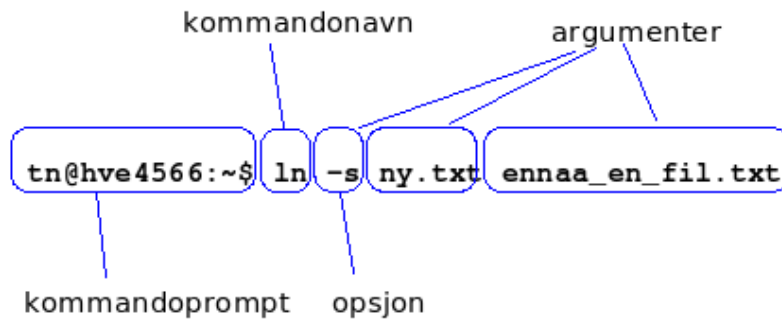
I dette kurset konsentrerer vi oss om standardskallet *bash*.

## finne hjelp



## Kommandolinjen

### Kommandolinjen



## Kommandohistorie

### Kommandohistorien

Filen `~/bash_history` inneholder kommandohistorien til skallet `bash`.

`ctrl-p`



```
tn@hve4566:~$ ln -s ny.txt ennaa_en_fil.txt
```

`ctrl-n`

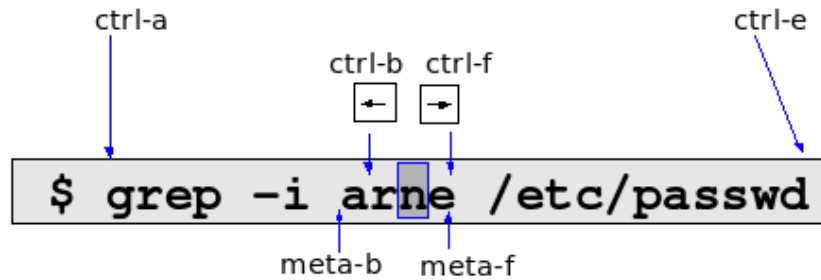


```
ls -l
cat ny.txt
cp ny.txt ny2.txt
ln -s ny.tzt filen_som_er.txt
rm ny2.txt
grep HINT02-B | cut -f5 -d:
cat enna_en_fil.txt
```

`ctrl-r` lar deg søke bakover i kommandohistorien

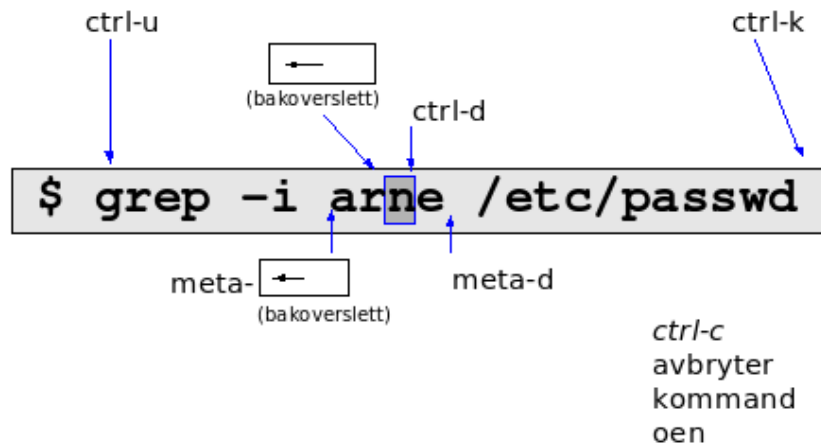
## Navigering på kommandolinjen

### Navigering på kommandolinjen



## Sletting på kommandolinjen

### Sletting på kommandolinjen





## Skrivehjelp



## Jokertegn

### Jokertegn

```
$ ls
eks1.pl  eks2.sh  eks4.sh  eks5.pl~  eks5.sh~
eks1.sh  eks3.sh  eks5.pl  eks5.sh  liste_med_kommandoer

$ echo *pl
eks1.pl eks5.pl

$ echo eks5???
eks5.pl eks5.sh
```

Omskrivningene skjer før kommandoen utføres

## Andre omskrivninger

### Andre omskrivninger

```
$ echo Vertsnavnet er $HOSTNAME
Vertsnavnet er hve4566

$ echo Arb.kat. er $(pwd)
Arb.kat. er /home/tn/public_html/net203/forelesning2/

$echo Hjemmekatalogen min er ~
Hjemmekatalogen min er /home/tn
```

Omskrivningene skjer før kommandoen utføres

## 3.7 Oppgave 3

- Oppgavene under er tenkt løst på kommandolinjen.

a)

- 1. Logg deg inn som en vanlig bruker
- 2. Lag tre kataloger inni hverandre
- 3. Lag en fil i den katalogen du lagde sist.
- 4. Kopier denne filen inn i de to andre katalogene du lagde.
- 5. Slett alle filene du lagde v.h.a. en enkelt kommando.

b)

<http://people.ischool.berkeley.edu/~kevin/unix-tutorial/section4.html>

# Kapittel 4

## 04 - Shell scripting

### 4.1 Repetisjon

- Hvilke filtyper finner vi i UNIX?
- Nevn noen kommandoer for filbehandling i UNIX.

#### Hva menes med

- Absolutt sti?
- Relativ sti?
- Standard inn (stdin)?
- Standard ut (stdout)?
- Standard error (stderr)?
- pipeline?
- Hvordan får vi til redigering til/fra filer i shellen?
- Hvordan får vi til redigering mellom prosesser i shellen?

## 4.2 Shell-skripting

### Innlesing av kommandoer

#### Innlesing av kommandoer

Tastaturet er representert som en fil. *Bash* leser kommandoer fra tastaturet. Kan skallet da lese kommandoer fra en annen fil?

```
HILSEN="God dag"
echo $HILSEN verden
```

filnavn: liste\_med\_kommandoer

Eksempler:

```
cat liste_med_kommandoer | bash
bash < liste_med_kommandoer
bash liste_med_kommandoer
```

### Skript som kjørbare fil

#### Skript som kjørbare fil

Går det an å få en liste over kommandoer til å oppføre seg mer som et kompilert program?

Gjøre filen kjørbare:

```
chmod u+x sti/til/liste_over_kommandoer
```

Oppgi stien til tolkeren, som skal tolke kommandoene i fila:

```
#!/bin/bash
# Kommentarer begynner
# også med '#'

HILSEN="God dag"
echo $HILSEN verden
```

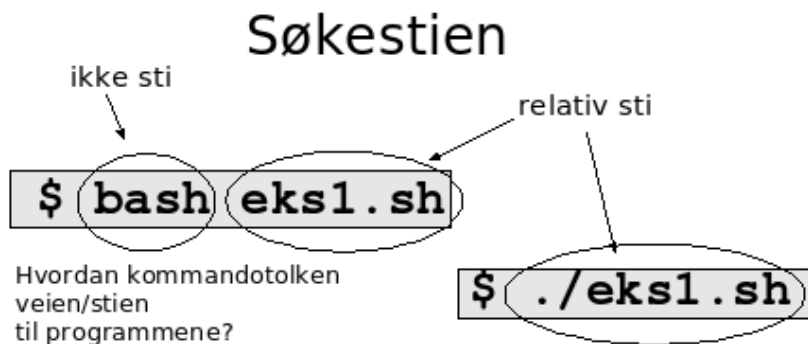
filnavn: eks1.sh

```
#!/usr/bin/perl
# Kommentarer i perl begynner
# også med '#'

$HILSEN="God dag";
print "$HILSEN verden\n";
```

filnavn: eks1.pl

## Søkestien



Kommando som viser innholdet i søkestien:

```
echo $PATH
```

Hvilken sti kommer til å bli funnet:

```
which bash
```

Hvordan får variabelen *PATH* sin verdi?

## Plassere en variabel i miljøet

### Plassere en variabel i miljøet

```
#!/bin/bash
echo $HILSEN verden
```

filnavn: eks2.sh

Hvordan kan verdien av en variabel tilordnet i skallet, overføres til en ny prosess?

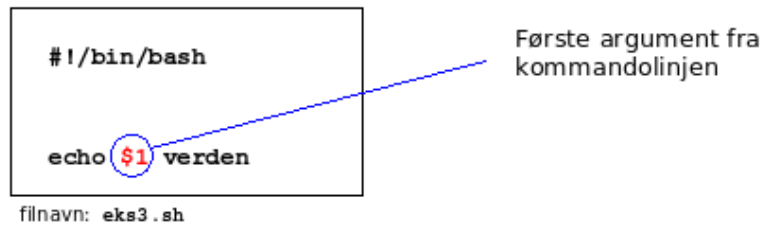
```
export HILSEN
eller
export HILSEN="God dag"
```

For å inspisere miljøet:

```
env
```

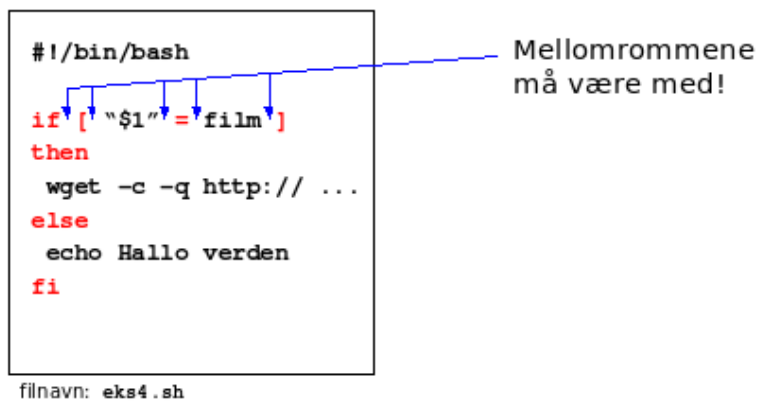
## Tilgang til kommandolinjeargumenter

### Tilgang til kommandolinjeargumenter



## Valgsetninger

### Valgsetninger



## Løkker

## Løkker

```
#!/bin/bash

for I in 1 2 3 4
do
    echo $I;
done
```

filnavn: eks5.sh

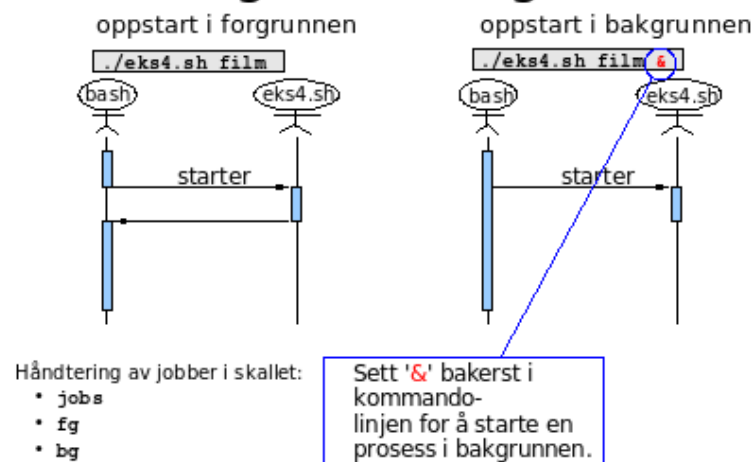
```
#!/usr/bin/perl

foreach $I (1, 2, 3, 4){
    print "$I\n";
}
```

filnavn: eks5.pl

## 4.3 Forgrunn/bakgrunn

## Forgrunn/bakgrunn



## 4.4 Oppgaver

Noen kommandoer som kan være nyttige for oppgaveløsningen.

- Produsere en tallsekvens fra null til ti:

```
seq 10
```

- Løkke som skriver tallene 1 2 3:

```
for I in 1 2 3; do echo $I; done
```

- Skrive ut første kommandolinjeargument:

```
echo $1
```

- Tilordne utskriften av en kommando til en variabel:

```
MEG=$(whoami)
```

- Plassere en variabel i miljøet:

```
export MEG
```

- Opprette en tom fil

```
touch filnavn
```

- Kompilere C-program:

```
gcc -Wall filnavn.c
```

- Slette fil

```
rm filnavn
```

- Legge en variabel til miljøet, slik at den arves av prosessens etterkommere:

```
export VARIABELNAVN
```

### 4.1 (tidligere 3.2)

- Lag et skript som:
  - 1. kompilerer et C-program
  - 2. kjører programmet
  - 3. sletter kildekoden og den kjørbare fila



**4.2 (tidligere 3.3)****a)**

Lag et I/O-intensivt og et CPU-intensivt program.

**b)**

Lag et skript som starter begge.

**c)**

Hvordan vil du få skriptet til å fullføre raskest mulig?

**4.3 (tidligere 3.4)**

- Lag et bash-skript som lager to tusen tomme filer i samme katalog. Filene ska hete: fil\_1, fil\_2, fil\_3, ...
- Slett katalogene etterpå med en enkelt kommando på kommandolinjen.

**4.4 (tidligere 3.5)**

Endre skriptet i forrige oppgave slik at antall filer bestemmes av et kommandolinjeargument.

**4.5 (tidligere 3.6)**

Utvid skriptet i forrige oppgave slik at antall filer bestemmes av en miljøvariabel (environment variable) dersom ikke kommandolinjeargument er oppgitt.



# Kapittel 5

## 05 - Filer og filsystemer

### 5.1 Løsningsforslag

#### 4.1

##### Oppgave

- Lag et skript som:
- 1. kompilert et C-program
- 2. kjører programmet
- 3. sletter kildekoden og den kjørbare fila

##### Løsningsforslag

Listing 5.1: losninger/04/4.1.sh

```
1 #!/bin/sh
2
3 gcc -Wall kopi.c -o kopi
4 ./kopi
5 rm kopi.c kopi
```

#### 4.2

##### Oppgave

- a) Lag et I/O-intensivt og et CPU-intensivt program.
- b) Lag et skript som starter begge.
- c) Hvordan vil du få skriptet til å fullføre raskest mulig?

##### Løsningsforslag 1 - med c-programmer

Listing 5.2: losninger/04/io-intensiv.c

```

1 #include <stdio.h>
2 #include <unistd.h>
3
4 int main () {
5
6     FILE *fil;
7     int i;
8
9
10    int buffer[BUFSIZ];
11
12    for (i=0; i<20; i++){
13
14        fil = fopen("film", "r");
15
16        while ( fread(buffer, BUFSIZ, 1, fil) )
17            sync();
18
19        fclose(fil);
20
21    }
22
23    return 0;
24
25 }
```

Listing 5.3: losninger/04/cpu-intensiv.c

```

1 #include <unistd.h>
2
3 int main () {
4
5     double x = 3.14;
6     int i, j;
7
8     for (i=0; i < 5000; i++)
9         for (j=0; j < 2500; j++)
10            x=x*x;
11
12    return 0;
13
14 }
```

Listing 5.4: losninger/04/4.2.b.sh

```

1 #!/bin/sh
2
3 # Ved aa starte det I/O-sensitive programmet foerst,
4 # og la det jobbe i bakgrunnen,
5 # vil ikke cpu-jobben maatte vente til io-jobben er ferdig.
6
7 ./io &
8 ./cpu
```

## Løsningsforslag 2 - med shellskript

Listing 5.5: losninger/04/last\_film.sh

```

1 #!/bin/bash
2 wget -c -q http://www.warriorsofthe.net/mirror/warriors-700-se-VBR.mpg

```

Listing 5.6: losninger/04/spill\_film.sh

```

1 #!/bin/bash
2 vlc warriors-700-se-VBR.mpg

```

Listing 5.7: losninger/04/4.2.b-alternativ.sh

```

1 #!/bin/sh
2
3 # Ved aa starte det I/O-sensitive programmet (filnedlastingen) foerst ,
4 # og la det jobbe i bakgrunnen ,
5 # vil ikke den cpu-intensive jobben (filmavspillingen) maatte vente til io-
6   jobben (nedlastingen) er ferdig .
7
8 ./last_film.sh &
9 ./spill_film.sh

```

## 4.3

### Oppgave

- Lag et bash-skript som lager to tusen tomme filer i samme katalog. Filene ska hete: fil\_1, fil\_2, fil\_3, ...
- Slett katalogene etterpå med en enkelt kommando på kommandolinjen.

### Løsningsforslag

Listing 5.8: losninger/04/4.3.sh

```

1 #!/bin/bash
2
3 for I in ${seq 2000}
4 do
5     touch fil_${I}
6 done

```

- For å slette alle filene kan kommandoen 'rm fil\_\*' gis.

## 4.4

### Oppgave

Endre skriptet i forrige oppgave slik at antall filer bestemmes av et kommandolinjeargument.

## Løsningsforslag

Listing 5.9: losninger/04/4.4.sh

```
1 #!/bin/bash
2
3 for i in $(seq 1 $1)
4 do
5
6     touch fil_ $i
7
8 done
```

## 4.5

### Oppgave

Utvid skriptet i forrige oppgave slik at antall filer bestemmes av en miljøvariabel (environment variable) dersom ikke kommandolinjeargument er oppgitt.

## Løsningsforslag

Listing 5.10: losninger/04/4.5.sh

```
1 #!/bin/bash
2 # Skriptet forventer miljøvariabelen ANT_FILER
3
4 if [ "$1" == "" ]
5 then
6
7     J=$ANT_FILER
8
9 else
10
11     J=$1
12 fi
13
14 for i in $(seq 1 $J)
15 do
16
17     touch fil_ $i
18
19 done
```

## 5.2 Generelt om filsystem

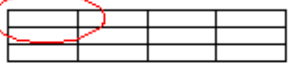
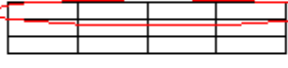
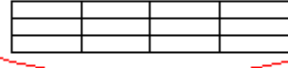
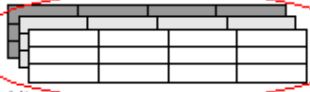
### Filsystem

## Filsystem

- Filer er viktige i mange, applikasjoner
- Mål for filsystem
  - Langtidslagring av data
  - Muligheter for deling av data mellom prosesser
  - Struktur (f.eks tre-struktur)
- Filhåndtering er helt eller delvis et OS-anliggende. Minimum betjente I/O-enheter.

### Noen termer

### Noen termer

- felt (field)
  - en enkel verdi
  - fast lengde, navn=verdi el. skille tegn
- post (record)
  - flere felt som behandles som en enhet.
- fil/tabell (file)
  - flere lignende poster som er sammenholdt
- database
  - flere relaterte filer/tabeller som er sammenholdt

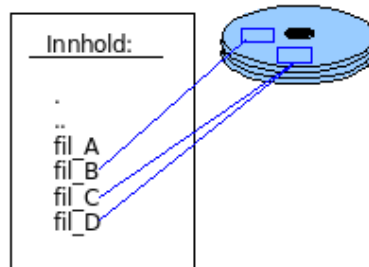
Eksempel på fil med poster og felt: `/etc/passwd`

### Kataloger

## Kataloger (mapper)

Filkataloger inneholder informasjon om en samling filer

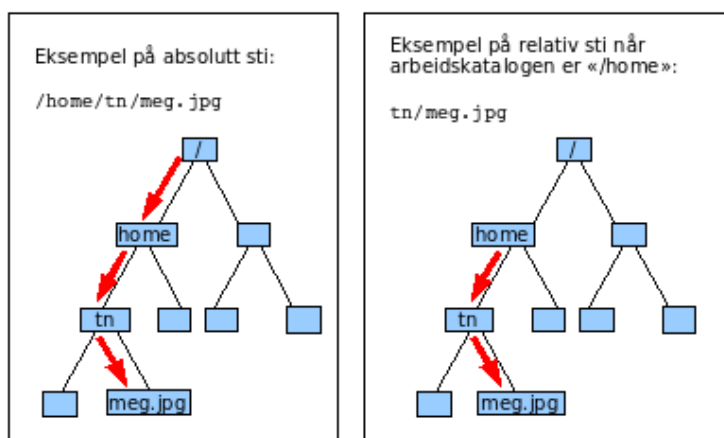
- F.eks. tidstemping, rettigheter, størrelse, eiendomsforhold, hvor filene er lagret, etc.
- For brukeren en kobling mellom filer og filers navn.
- Alle kataloger kan inneholde kataloger – slik danner katalog-strukturen en hierarkisk trestruktur.
- En bestemt katalog kalles *rotkatalogen* den kan ikke være i en annen katalog – den kan ikke ha noen *foreldrekatalog*.



## Navngivning av filer

## Navngivning av filer

- symbolsk for brukere (i stedet for numerisk)
- hver fil må kunne nevnes entydig



## Fil-funksjonalitet



### Typiske funksjoner for filhåndterings-system

- opprette - eks.: creat(2)/open(2)
- slette - eks.: unlink(2)
- åpne - eks.: open(2)
- lese - eks.: read(2)
- skrive - eks.: write(2)
- Tilgangskontroll ...

### Vi ser på et eksempel:

Listing 5.11: eksempler/05/openeksempel.c

```

1 #include <sys/types.h>
2 #include <fcntl.h>
3 #include <unistd.h>
4 #include <string.h>
5 #include <sys/stat.h>
6
7 /*
8
9  Demonstrasjon av systemkall:
10
11  - open(2)
12  - write(2)
13  - read(2)
14  - close(2)
15
16 */
17
18 int main()
19 {
20
21     char buffer[100];
22     int fd, ant;
23
24     fd = open("testfil", O_RDWR | O_EXCL | O_CREAT, S_IRUSR);
25
26     if (fd < 0){
27
28         strcpy(buffer, "Feil_ved_aapning_av_fil.\n");
29         ant=strlen(buffer);
30
31         write(2, buffer, ant);
32         return 1;
33
34     }
35

```

```

36 strcpy(buffer, "Fil_aapen.\nTrykk_enter.\n");
37 ant=strlen(buffer);
38
39 write(1, buffer, ant);
40 read(0, NULL, 1);
41
42 close(fd);
43
44 strcpy(buffer, "Fil_lukket.\nTrykk_enter.\n");
45 ant=strlen(buffer);
46
47 write(1, buffer, ant);
48 read(0, NULL, 1);
49
50 unlink("testfil");
51
52 return 0;
53 }

```

## 5.3 Filhåndtering i Linux

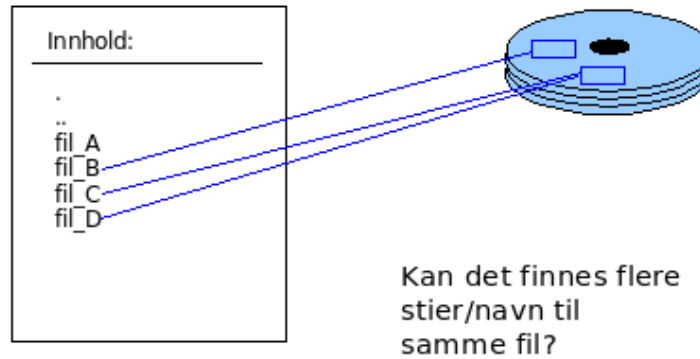
### Filtyper i Linux

### Filtyper i Linux

- |                              |  |
|------------------------------|--|
| 1. Vanlige filer             |  |
| 2. Kataloger                 |  |
| 3. Karakter-<br>enhetsfiler  |  |
| 4. Blokk-enhetsfiler         |  |
| 5. Symbolske lenker          |  |
| 6. "Local domain<br>sockets" |  |
| 7. Navngitte rør<br>(FIFO's) |  |
|                              | Hvorfor er så mye<br>forskjellig<br>representert som<br>filer? |

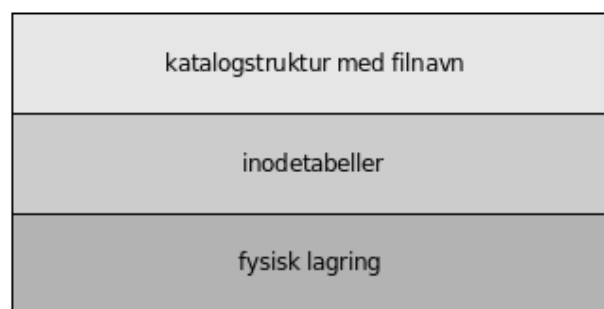
flere stier til samme fil?

## Kataloger/mapper



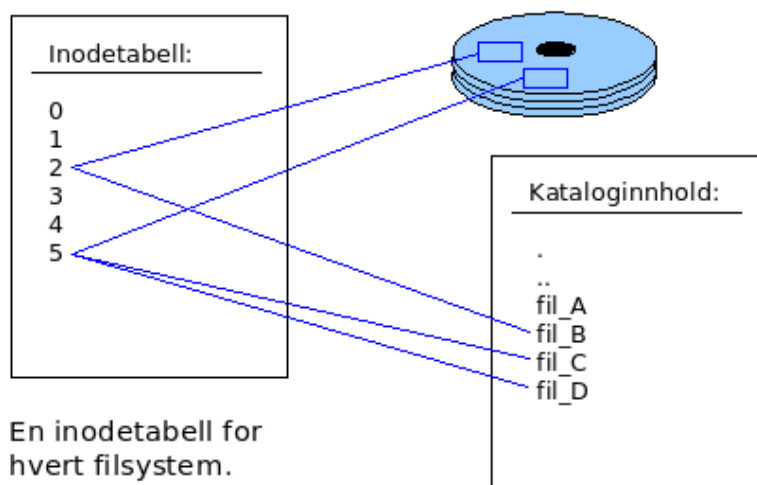
### Forenklet lagdelt modell

## Forenklet lagdelt modell av filsystemet linux

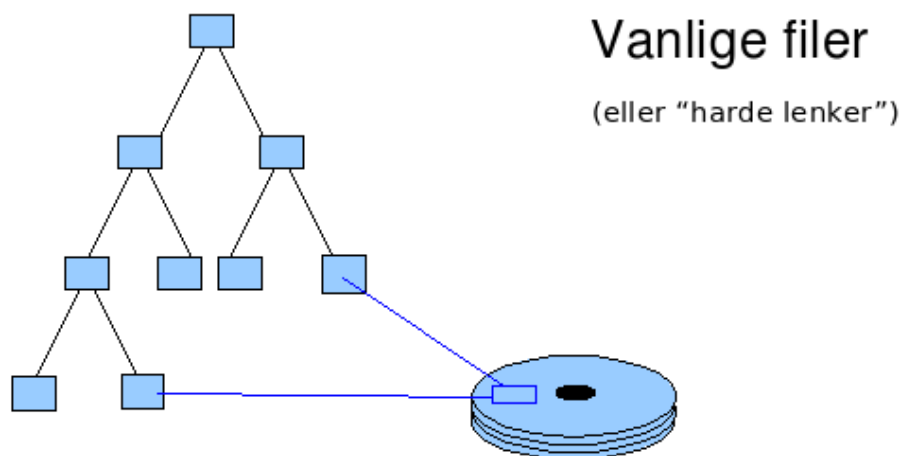


### Inodetabeller

## Inodetabeller



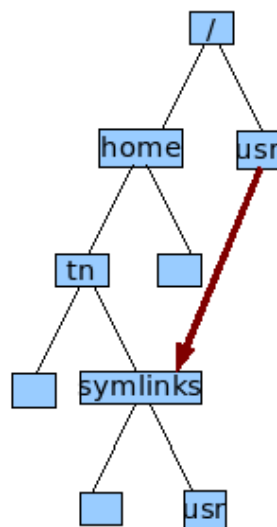
## Hardlinks



Eksempel:

```
ln sti/til/opprinnelig/fil1 sti/til/ny/link
```

## Symlinks



## Symbolske lenker

- Kan det finnes flere stier til samme kata

Eksempel:

```
ln -s /usr /home/tn/symlinks
```

## Løkker i filsystemet

## Løkker i filsystemet

Ved bruk av symbolske lenker, kan løkker oppstå.

Kjernen oppdager løkken og gir feilmelding:

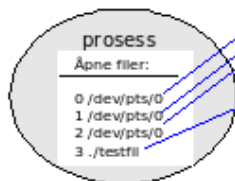
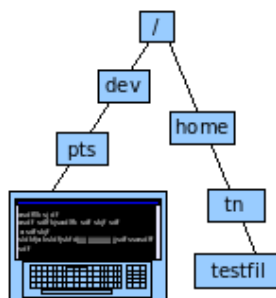
```
[eks]# mkdir testkatalog
[eks]# cd testkatalog/
[testkatalog]# ln -s ../testkatalog/filB filB
[testkatalog]# cat filB
cat: filB: For mange nivåer med symbolske linker
[testkatalog]#
```

Eksempel: Åpne filer

## Eksempel på åpne filer

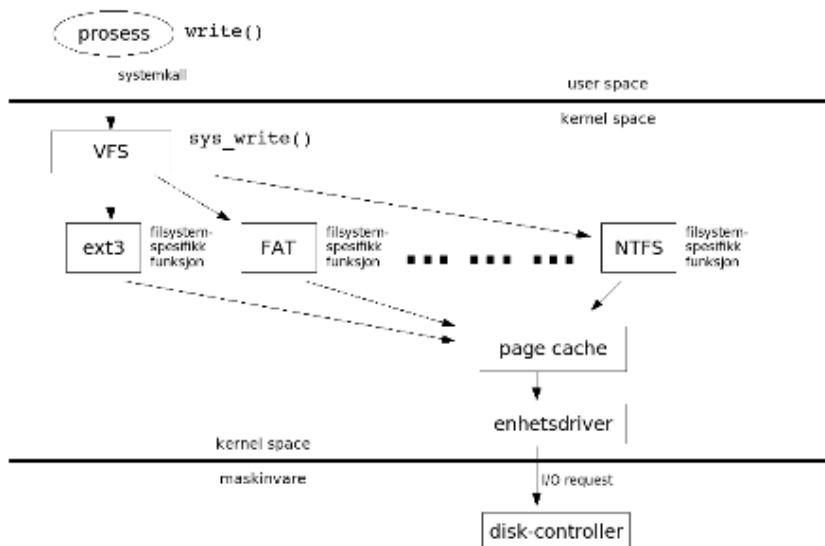
```

$ gcc -Wall openeksempel.c -o op_ex
$ ./op_ex &
[1] 26499
$ F11 Åpen Trykk enter.
jobs -l
[1]+ 26499 Stoppet (ville lese fra tty) ./op_ex
$ ls -l /proc/26499/fd
totalt 4
lrwx----- 1 tn tn 64 2008-09-10 11:57 0 -> /dev/pts/0
lrwx----- 1 tn tn 64 2008-09-10 11:57 1 -> /dev/pts/0
lrwx----- 1 tn tn 64 2008-09-10 11:57 2 -> /dev/pts/0
lrwx----- 1 tn tn 64 2008-09-10 11:57 3 -> /home/tn/testfil
$
    
```



## Virtual File System

## VFS



## Opprettelse av filsystem

## Opprettelse av filsystem

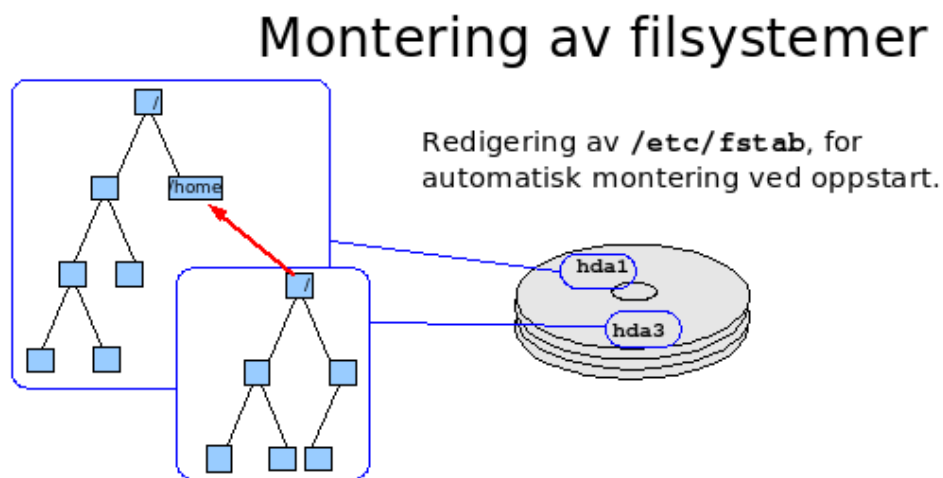
Ved opprettelse av et filsystem, blir det skrevet noen datastrukturer til disken.

Kommando for å skrive filsystem til disken:

- `mkfs (8)`

Filen `/proc/filesystems` inneholder en liste over hvilke filsystemer den kjørende kjernen har støtte for.

## Montering av filsystem



Eksempel:

```
mount /dev/hda3 /home
```

## 5.4 Oppgaver

- Noen av oppgavene under krever at du har root-tilgang. Du kan f.eks. bruke lab-maskinene eller egen PC. Det er også mulig å løse oppgavene på debbie.

- Dersom du ønsker å gjøre oppgaven på `debbie.hive.no`, kan du starte en virtuell maskin med kommanden: `vstart VMNAVN`, hvor VMNAVN er navnet du gir til den virtuelle maskinen. Denne kommandoen starter en virtuell maskin og et konsollvindu hvor du automatisk logges inn som root. Se `vstart(1)`. Tallet i parentes etter kommandoenavnet angir seksjonen i manualen. For å lese manualen: `'man 1 vstart'`. For mer informasjon om den virtuelle maskinen, for de spesielt interesserte, se: <http://netkit.org>.
- I oppgavene under finner du mange referanser til kommandoer med et tall i parentes bak. F.eks. slik: `cp(1)`. Tallet i parentes etter kommandoenavnene angir seksjonen i manualen. Seksjonsnummeret skrives før kommandonavnet. F.eks. slik: `'man 1 cp'`.

## 5.1 Lenker

- For å løse denne oppgaven skal du bruke kommandoene: `ln(1)`, `rm(1)`. I tillegg trenger du å bruke en editor av fritt valg (`emacs(1)`, `vim(1)`, `nano(1)`, `mcedit(1)`, ...).

### 5.1.1 Harde lenker

- a) Opprett en fil med navn A og med teksten "Dette er fil A".
- b) Lag en hardlink med navn B, til filen A.
- c) Slett deretter A.
- d) Hva har skjedd med B?
- e) Lag en ny fil med navn A og med teksten "Dette er NYE fil A".
- f) Hva er nå skjedd med B?

### 5.1.2 Symbolske lenker

- a) Opprett en fil med navn A og med teksten "Dette er fil A".
- b) Lag en softlink med navn B, til filen A.
- c) Slett deretter A.
- d) Hva har skjedd med B?
- e) Lag en ny fil med navn A og med teksten "Dette er NYE fil A".
- f) Hva er nå skjedd med B?



## 5.2 Navnelister

- I denne oppgaven trenger du å bruke programmene `cut(1)`, `sort(1)`, `uniq(1)` og filen `/etc/passwd`.
- a) Lag en kommandolinje («pipeline») som skriver en alfabetisk sortert liste over brukernavn på et UNIX/Linux-system til en fil ved navn `brukernavn.txt`. Prøv den på `debbie`.
- b) Lag en kommandolinje («pipeline») som skriver en alfabetisk sortert liste over alle brukernes etternavn. Forsøk å unngå duplikate etternavn. Prøv den på `debbie`.

## 5.3 Systemkall

- Rediger eksemplet `'openeksempel.c'` slik at programmet lukker standard utgang før testfil åpnes. Du finner eksemplet i fila `/home/tn/osyda50/eksempler/05/openeksempel.c` på `debbie`,
- Sørg også for at testfil ikke slettes før programmet terminerer.
- Kompiler og kjør programmet. Ser du noen forskjell på utskriften?
- Hva er innholdet i testfil?
- Kan du forklare hva som er skjedd?

## 5.4 Montering

- For å gjøre denne oppgaven må du ha root-tilgang. Du kan f.eks. bruke lab-maskinene, egen pc eller virtuell-maskin på `debbie`.
- I denne oppgaven trenger du å bruke `dd(1)`, `zero(4)`, `mkfs(8)`, `mount(8)`, `cp(1)` og `umount(8)`

### a) Lag en fil med størrelse 3 MB

- Du kan bruke kopieringsprogrammet `dd(1)`. Bruken av `dd` er litt uortodoks. Det stammer fra JCL (Job Control Language).
- Det er fin artikkel om 'dd', med eksempler, på wikipedia [http://en.wikipedia.org/wiki/Dd\\_\(Unix\)](http://en.wikipedia.org/wiki/Dd_(Unix)).
- Du kan f.eks. fylle en fil med nuller ved å kopiere fra spesialfilen `/dev/zero` – `zero(4)`.
- b) Lag et filsystem av valgfri type på filen v.h.a `mkfs(8)`.

**c) Monter filsystemet og kopier noen filer over på det.**

- Du må vanligvis være 'root' for å kunne montere et filsystem.
- Til slutt avmonterer du filsystemet.
- d) Legg inn en linje filen /etc/fstab, slik at filsystemet monteres ved oppstart.
- e) Gjør en om start av systemet med kommandoen reboot(8), og kontroller at monteringen ble utført korrekt ved oppstart.

# Kapittel 6

## 06 - Filsystem-sikkerhet og brukere i Linux

### 6.1 Løsningsforslag

#### 5.1.1 Harde lenker

##### Oppgave

- a) Opprett en fil med navn A og med teksten "Dette er fil A".
- b) Lag en hardlink med navn B, til filen A.
- c) Slett deretter A.
- d) Hva har skjedd med B?
- e) Lag en ny fil med navn A og med teksten "Dette er NYE fil A".
- f) Hva er nå skjedd med B?

##### Løsning

<http://debbie.hive.no/osyda50/losninger/05/5.1.1.txt>

#### 5.1.2 Symbolske lenker

##### Oppgave

- a) Opprett en fil med navn A og med teksten "Dette er fil A".
- b) Lag en softlink med navn B, til filen A.
- c) Slett deretter A.
- d) Hva har skjedd med B?
- e) Lag en ny fil med navn A og med teksten "Dette er NYE fil A".
- f) Hva er nå skjedd med B?

## Løsning

<http://debbie.hive.no/osyda50/losninger/05/5.1.2.txt>

## 5.2 Navnelister

### Oppgave

- a) Lag en kommandolinje («pipeline») som skriver en alfabetisk sortert liste over brukernavn på et UNIX/Linux-system til en fil ved navn brukernavn.txt. Prøv den på debbie.
- b) Lag en kommandolinje («pipeline») som skriver en alfabetisk sortert liste over alle brukernes etternavn. Forsøk å unngå duplikate etternavn. Prøv den på debbie.

## Løsning

```
cut -f1 /etc/passwd -d: | sort > brukernavn.txt
```

## 5.3 Systemkall

### Oppgave

- Rediger eksemplet 'openeksempel.c' slik at programmet lukker standard utgang før testfil åpnes. Du finner eksemplet i fila /home/tn/osyda50/eksempler/05/openeksempel.c på debbie,
- Sørg også for at testfil ikke slettes før programmet terminerer.
- Kompiler og kjør programmet. Ser du noen forskjell på utskriften?
- Hva er innholdet i testfil?
- Kan du forklare hva som er skjedd?

## Løsning

Listing 6.1: losninger/05/5.3.c

```

1 #include <sys/types.h>
2 #include <fcntl.h>
3 #include <unistd.h>
4 #include <string.h>
5 #include <sys/stat.h>
6
7 /*
8
9  Demonstrasjon av systemkall:
10
11  - open(2)
12  - write(2)
13  - read(2)
14  - close(2)
15

```

```

16 */
17
18 int main()
19 {
20
21     char buffer[100];
22     int fd, ant;
23
24     // Endring 1: Lagt til linjen under.
25     close(1);
26
27     fd = open("testfil", O_RDWR | O_EXCL | O_CREAT, S_IRUSR);
28
29     if (fd < 0){
30
31         strcpy(buffer, "Feil ved aapning av fil.\n");
32         ant=strlen(buffer);
33
34         write(2, buffer, ant);
35         return 1;
36
37     }
38
39
40     strcpy(buffer, "Fil aapen. Trykk enter.\n");
41     ant=strlen(buffer);
42
43     write(1, buffer, ant);
44     read(0, NULL, 1);
45
46     close(fd);
47
48     strcpy(buffer, "Fil lukket.\nTrykk enter.\n");
49     ant=strlen(buffer);
50
51     write(1, buffer, ant);
52     read(0, NULL, 1);
53
54     // Endring 2: Kommentert bort linjen under
55     // unlink("testfil");
56
57     return 0;
58 }

```

- Når standard ut lukkes, blir fildeskriptor nr. 1 ledig i tabellen over åpne filer. Kallet til `open()` returnerer laveste ledige fildeskriptor, nemlig fildeskriptor 1. Utskriften vil dermed havne i fila, istedet for på konsollet.

## 5.4 Montering

a)

**Oppgave** Lag en fil med størrelse 3 MB

**Løsning** `dd if=/dev/zero of=/filssystem count=3 bs=1M`

b)

**Oppgave** Lag et filsystem av valgfri type på filen v.h.a mkfs(8).

**Løsning** mkfs.ext3 /filssystem

c)

**Oppgave**

- Monter filsystemet og kopier noen filer over på det.
- Du må vanligvis være 'root' for å kunne montere et filsystem.
- Til slutt avmonterer du filsystemet.

**Løsning**

- mkdir /monteringspunkt
- mount -o loop /filssystem /monteringspunkt
- cp /etc/passwd /etc/shadow /monteringspunkt/
- umount /monteringspunkt

d)

**Oppgave** Legg inn en linje i filen /etc/fstab, slik at filsystemet monteres ved oppstart.

**Løsning** echo /filssystem /monteringspunkt/ ext3 loop 0 0 » /etc/fstab

e)

**Oppgave** Gjør en omstart av systemet med kommandoen reboot(8), og kontroller at monteringen ble utført korrekt ved oppstart.

**Kommentar** På den virtuelle maskinen som kan startes med vstart(1), blir ikke kommandoen 'mount -a' kjørt ved oppstart, slik den vanligvis blir. Denne må derfor kjøres manuelt (eller legges til en av oppstartsfilene) etter omstart.

## 6.2 Filsystem-sikkerhet

### ACM

## Access Control Matrix

- Generell modell av system for tilgangskontroll
  - mengden av *rettigheter*:  $R$ .
  - mengden av *objekter* som skal beskyttes:  $O$
  - mengden av handlende objekter – *subjekter*:  $S$
  - matrisen  $A$  med elementer  $a[s,o]$
  - når  $s \in S$ ,  $o \in O$  gir  $a[s,o] \subseteq R$
  - eks.

$$\bullet R = \{r, w, x\}$$

$$\bullet S = \{s_1, s_2\}$$

$$\bullet O = \{o_1, o_2\}$$

A=

	$o_1$	$o_2$	$s_1$	$s_2$
$s_1$	r	r	r,w	
$s_2$		r,w,x		r,w

### ACL

## Access Control Lists

- Elementer *med innhold* i en *kolonne* i matrisen tilsvarer en ACL

A=

	$o_1$	$o_2$	$s_1$	$s_2$
$s_1$	r	r	r,w	
$s_2$		r,w,x		r,w

## Capability Lists

### Capability Lists

- Elementer *med innhold* i en *rad* i matrisen tilsvarer en *Capability List*.

A=

	$o_1$	$o_2$	$s_1$	$s_2$
$s_1$	r	r	r,W	
$s_2$		r,W,X		r,W

## ACL i linux

### ACL i Linux

- trad. UNIX tilnærming – minimal ACL

	User	Group	Others
	Eieren	Gruppemedlemmer	Alle Brukere
R	Lese	0/1	0/1
W	Skrive	0/1	0/1
X	Utføre	0/1	0/1

$0-7$       $0-7$       $0-7$   
**rwx**    **rwx**    **rwx**  
 U        G        O



## Filrettigheter: Notasjon

### Notasjon av filrettigheter i UNIX/Linux

Eksempel:

```
-rw-rw-r-- 1 tn tn 20K jan 5 15:16 foreles1.sxi
```

Binær notasjon: 

110	110	100
6	6	4

Otal notasjon:

110	110	100
6	6	4

## 6.3 Brukere i Linux

### Brukerkonto

### Brukerkonto

- En brukerkonto består av filer, resursser og informasjon som tilhører en bruker.
- Tilhørighet til en bruker, angis med et brukernummer (UID)

## Brukerdatabasen /etc/passwd

## “Brukerdatabasen”

brukeravn
kryptert passord (dersom ikke skyggepassord er installert)
bruker nummer (UID)
gruppenummer (GID)
GECOS (fullt navn, kontor nummer og bygning)
sti til hjemmekatalog
sti til innloggings-skall

```
tippmann:x:1117:100:Lars Kjerland,02HINT-B,,:/home/tippmann:/bin/bash
laktose:x:1118:100:Herbjørn Fortaasen,02HINT-B,,:/home/laktose:/bin/bash
filip:x:1119:100:Filip Kandal Gjersøe,02HINT-B,,:/home/filip:/bin/bash
kenjo:x:1120:100:Kenneth Johannessen,03HINT-B,,:/home/kenjo:/bin/bash
krset:x:1121:100:Kristian Syse Setsaas,02HINT-B,,:/home/krset:/bin/bash
janterje:x:1122:100:Jan Terje Nilsen,02HINT-B,,:/home/janterje:/bin/bash
```

utdrag fra filen /etc/passwd

Eksempler på programmer som slår opp i /etc/passwd:

- finger
- ls

## Skyggepassord - /etc/shadow

## Skyggepassord

brukeravn
kryptert passord
dato for siste endring av passord
mininum antall dager mellom passordendringer
maximum antall dager mellom passordendringer
antall dager før utløpsdato, som advarsel skal gis
antall dager etter utløpsdato, som kontoen blir deaktivert
kontoens utløpsdato
reservert for fremtidig bruk

```
terjef:mXTveXqIac6teX:12288:0:99999:7:0::
truls:troInJZAoZetg:12288:0:99999:7:0::
johob:jo/KNv118E:12288:0:99999:7:0::
ifreska:ifUPS7v3schXuNmE:12288:0:99999:7:0::
manou:magzAUa2jtKzI:12288:0:99999:7:0::
bravoman:brz5rm9cFeDq:12288:0:99999:7:0::
```

utdrag fra filen /etc/shadow

## Opprettelse av brukere

### Opprette brukere

Følgende utføres ved opprettelse av en ny bruker:

- Ny linje i filen `/etc/passwd`
- Endringer i filen `/etc/group`
- (Ny linje i filen `/etc/shadow`)
- Opprette hjemmekatalog
- Kopiere innholdet i `/etc/skel` til hjemmekatalogen
- Endre eierskap og rettigheter på hjemmekatalogen (med innhold).
- Eventuelle andre tilpasninger

Eksempler på kommandoer som oppretter ny bruker:

- `adduser (8)`
- `useradd (8)`

## Deaktivering av brukere

### Deaktivere brukere

- Deaktivere passordet – umulig for brukeren å logge inn v.h.a passord
  - sette et utropstegn foran det krypterte passordet
- `passwd -l`
- Fjerne mulighetene for brukeren å få et shell når han logger inn
  - endre stien til oppstarts-skallet i `/etc/passwd` til et program som ikke er et shell. F.eks. `/bin/false`

## Fjerne brukere

# Fjerne brukere

Dette må gjøres for å fjerne en brukerkonto:

- Alle brukerens filer må fjernes
- Alle brukerens prosesser må stoppes
- Evt. annet.

Eksempler på programmer som kan fjerne brukerkonti:

- **userdel (8)**
- **deluser (8)**

## 6.4 Oppgaver

### 6.1 Brukerhåndtering

- For å gjøre denne oppgaven må du ha root-tilgang. Du kan f.eks. bruke lab-maksinene, egen pc eller virtuell-maskin på debbie.

a)

Lag en brukerkonto for Dolly Duck med adduser(8)

b)

- Du trenger å bruke en editor, mkdir(1), cp(1) og chown(1), for å gjøre denne oppgaven.
- Lag en brukerkonto for Skrue McDuck manuelt ved å gjøre alle trinnene beskrevet tidligere i notatene. Du skal altså redigere /etc/passwd med en editor o.s.v.

c)

Deaktiver onkel Skrues konto, og kontroller at han ikke kan logge inn.

d)

Slett Dollys konto med deluser(8)

e)

Slett Skrues konto v.h.a. `rm(8)` og en editor.

## 6.2 Hemmelig passordfil

- For å gjøre denne oppgaven må du ha root-tilgang. Du kan f.eks. bruke lab-maskinene, egen pc eller virtuell-maskin på debbie.
- Du bestemmer deg for at det er for usikkert at ordinære brukere kan lese bruker-databasen (`/etc/passwd`) i linux-systemet ditt. Derfor endrer du rettighetene slik at det bare er mulig for brukeren root å lese den. Noter opprinnelige rettigheter. Gjør endringen v.h.a. kommandoen `chmod(1)`.
- Logg inn som en ordinær bruker. Merker du noe forskjell? Kan du forklare hva som har skjedd? Endre rettighetene tilbake til det opprinnelige – som du noterte.



# Kapittel 7

## 07-prosesser

### 7.1 Løsningsforlag

#### 6.1 (tidligere 5.5) Brukerhåndtering

- For å gjøre denne oppgaven må du ha root-tilgang. Du kan f.eks. bruke lab-maksinene, egen pc eller virtuell-maskin på debbie.

a)

**Oppgave** Lag en brukerkonto for Dolly Duck med `adduser(8)`

Listing 7.1: `losninger/06/6.1.a.txt`

```
1 Script started on Tue 28 Sep 2010 06:33:33 AM UTC
2 test1:~# adduser dolly
3 Adding user 'dolly' ...
4 Adding new group 'dolly' (1001) ...
5 Adding new user 'dolly' (1001) with group 'dolly' ...
6 Creating home directory '/home/dolly' ...
7 Copying files from '/etc/skel' ...
8 Enter new UNIX password:
9 Retype new UNIX password:
10 passwd: password updated successfully
11 Changing the user information for dolly
12 Enter the new value, or press ENTER for the default
13   Full Name []: Dolly Duck
14   Room Number []:
15   Work Phone []:
16   Home Phone []:
17   Other []:
18 Is the information correct? [Y/n]
19 test1:~# exit
20
21 Script done on Tue 28 Sep 2010 06:34:26 AM UTC
```

b)

**Oppgave** Lag en brukerkonto for Skrue McDuck manuelt ved å gjøre alle trinnene beskrevet tidligere i notatene. Du skal altså redigere `/etc/passwd` med en editor o.s.v.

Listing 7.2: losninger/06/6.1.b.sh

```

1 #!/bin/sh
2 echo "skrue:x:1002:1002:Skrue_Mc_Duck,,,:/home/skrue:/bin/bash" >> /etc/
  passwd
3 echo "skrue:x:1002" >> /etc/group
4 echo "skrue*:14880:0:99999:7:::" >> /etc/shadow
5 cp -r /etc/skel/ /home/skrue
6 chown -R skrue.skrue /home/skrue
7 passwd skrue

```

c)

**Oppgave** Deaktiver onkel Skrues konto, og kontroller at han ikke kan logge inn.

**Løsningsforslag** Kommandoen 'passwd -l skrue' vil kun sørge for at skrue ikke kan logge inn med passord. Men dersom Skrue kan autentiseres på andre måter (f.eks. med offentlig nøkkel), vil han allikevel kunne logge inn. Ved å gi kommandoen 'chage -E0 skrue', blir utløpsdatoen satt til dag 0 i år 0 ("etter UNIX"), d.v.s 1. jan. 1970. Skrues konto der dermed deaktivert (forutsatt at klokka på maskinen går riktig).

d)

**Oppgave** Slett Dollys konto med deluser(8)

**Løsningsforslag** deluser -remove-all-files dolly

e)

**Oppgave** Slett Skrues konto v.h.a. rm(8) og en editor.

**Løsningsforslag**

- Med en editor sletter du linjene, som gjelder skrues konto, i filene passwd(5), group(5) og shadow(5).
- For å slette skrues hjemmekatalog kan du bruke kommandoen 'rm -r /home/skrue'.
- Dersom du er usikker på om skrue har andre filer rundt omkring, som bør slettes, kan du f.eks. bruke programmet find(1) for finne dem. Kommandoen 'find / -nouser' finner eierløse filer.

**Kommentar**

- Bruk helst editoren vipw(8) for å redigere filene /etc/passwd, /etc/group og /etc/shadow. Denne sørger for å låse filen som redigeres, slik at ingen andre gjør endringer (f.eks. endrer passord), mens du redigerer.
- Default editor for vipw(8) er vi(1). Ved å sette miljøvariabelen \$EDITOR til stien til din favoritt-editor vil vipw(8) bruke denne i stedet.



**Eksempel:**

- På 'netkit virtuell maskin', som kan startes med vstart(1) på debbie, finner du den brukervennlige editoren mcedit(1). Denne har et menybasert og ganske intuitivt grensesnitt. Du kan bruke både mus og funksjonstaster.
- For å finne denne stien bruker du kommandoen 'which mcedit'. Kommandoen 'export EDITOR=\$(which mcedit)' setter variabelen i miljøet med korrekt sti.

**6.2 (tidligere 5.6) Hemmelig passordfil****Oppgave**

- Du bestemmer deg for at det er for usikkert at ordinære brukere kan lese bruker-databasen (/etc/passwd) i linux-systemet ditt. Derfor endrer du rettighetene slik at det bare er mulig for brukeren root å lese den. Noter opprinnelige rettigheter. Gjør endringen v.h.a. kommandoen chmod(1).
- Logg inn som en ordinær bruker. Merker du noe forskjell? Kan du forklare hva som har skjedd? Endre rettighetene tilbake til det opprinnelige – som du noterte.

**Løsning**

Listing 7.3: losninger/06/6.2.txt

```

1 Script started on Tue 28 Sep 2010 09:37:23 AM UTC
2 test1:/hosthome/osyda50/losninger/06# stat /etc/passwd
3   File: '/etc/passwd'
4   Size: 1637      Blocks: 4          IO Block: 1024   regular file
5 Device: 6201h/25089d Inode: 54957      Links: 1
6 Access: (0644/-rw-r--r--)  Uid: (   0/   root)   Gid: (   0/   root)
7 Access: 2010-09-28 09:37:23.000000000 +0000
8 Modify: 2010-09-28 09:32:41.000000000 +0000
9 Change: 2010-09-28 09:37:12.000000000 +0000
10 test1:/hosthome/osyda50/losninger/06# chmod 600 /etc/passwd
11 test1:/hosthome/osyda50/losninger/06# stat /etc/passwd
12   File: '/etc/passwd'
13   Size: 1637      Blocks: 4          IO Block: 1024   regular file
14 Device: 6201h/25089d Inode: 54957      Links: 1
15 Access: (0600/-rw-----)  Uid: (   0/   root)   Gid: (   0/   root)
16 Access: 2010-09-28 09:37:27.000000000 +0000
17 Modify: 2010-09-28 09:32:41.000000000 +0000
18 Change: 2010-09-28 09:37:35.000000000 +0000
19 test1:/hosthome/osyda50/losninger/06# login
20 test1 login: dolly
21 Password:
22 Last login: Tue Sep 28 09:35:14 UTC 2010 on tty0
23 I have no name!@test1: I have no name!@test1:~$ ls /home -l
24 total 3
25 drwxr-xr-x 2 1001 dolly    1024 2010-09-28 09:35 dolly
26 drwxr-xr-x 2 110  nogroup 1024 2008-12-06 16:50 mftp
27 drwxr-xr-x 2 1000 guest   1024 2008-12-06 16:53 guest
28 I have no name!@test1: ~I have no name!@test1:~$ logout
29 test1:/hosthome/osyda50/losninger/06#

```

## 7.2 Om prosesser

### Problemet ved utviklingen av "batch"-, "time share" og "real time"-OS

- feilaktig synkronisering
- feil ved gjensidig utelukkelse
- uberegnelig utfall av progamkjøring
- vranglåser

### Mot en løsning

- Problemene imøtegåes med systematisk måte å kontrollere programutførelsene
- Begrepet "process" ble først brukt under utvikling av Multics på 60-tallet

### Begrepet Prosess - ulike definisjoner

- Et program under utførelse
- En instans av et program som kjører på en datamaskin
- Entitet som kan tildeles og kjøres på en prosessor

### En enhet av aktivitet karakterisert av

- en enkel sekvensiell tråd av utførelse
- en tilstand
- tilhørende system resursser
- «an address space with one or more threads executing within that address space, and the required system resources for those threads.» [ Single UNIX Specification, Versjon1 (UNIX95) og Versjon 2 (UNIX98) ]

### Prosess består av

- et kjørbart program
- tilhørende data

### kjøre-kontekst

- info nødveding for å kontrollere prosessen
- f.eks. ulike prosessor-registre, prioritet, tid-brukt, etc.

## 7.3 Representasjon/beskrivelse av prosesser

- Operativsystemets representasjon av en prosess kan kalles et "Process Image"

### Process Image kan deles inn i

#### Adresserom - for programkjøring

##### data

- data og program som skal kunne modifiseres
- i "userspace"

##### program

- maskinkoden som skal utføres
- "read only"

**stakk** LIFO (minst en) for lagring ved prosedyrer og systemkall

### Process Control Block (PCB) - for prosesshåndtering

#### Prosess-identifikasjon

- ID for prosessen, brukeren, foreldreprosess, etc.

#### Eksempel fra Linux-kjernen:

```
struct task_struct {
    ...
    pid_t pid;
    ...
    struct task_struct *parent;    /* parent process */
    ...
    uid_t uid, euid, suid, fsuid;
    gid_t gid, egid, sgid, fsgid;
    ...
}
```

<http://lxr.linux.no/#linux+v2.6.31/include/linux/sched.h#L1166>

#### Finne prosess-identifikator

```
pid_t getpid(void);
pid_t getppid(void);
```

- `getpid(2)`

Eksempler

Listing 7.4: eksempler/07/getpid-eksempel.c

```

1  #include <unistd.h>
2  #include <stdio.h>
3
4  /*
5
6   Programmet skriver ut sin egen
7   prosess-identifikator
8
9  */
10
11 int main(void)
12 {
13     int pid;
14
15     pid = getpid();
16     printf("Jeg_er_prosess_nr._%d\n", pid);
17     return 0;
18
19 }
```

- echo \$\$

**Prosess-tilstand** Verdien på ulike registre

### Informasjon for prosess-kontroll

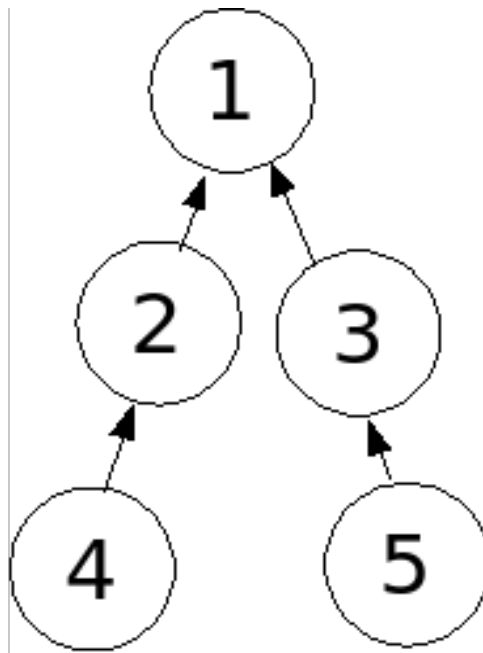
- Skedulering og tilstandsinformasjon
- Strukturering
- Interprosess-kommunikasjon
- Privilegier
- Minnehåndtering
- Tilknytning til resursser
- Måling av resurss-bruk

<http://lxr.linux.no/#linux+v2.6.31/include/linux/sched.h#L1166>

## Vi ser på prosesser i Linux

### Prosess-treet i Linux

Prosessene er organsiert i en trestruktur.



Hver prosess har en foreldreprosess (bortsett fra den første, `init`)

```
struct task_struct {  
    ...  
    struct task_struct *parent;  
    ...  
}
```

<http://lxr.linux.no/#linux+v2.6.31/include/linux/sched.h#L1166>

Eksempel på kommandoer som viser prosess-treet:

- `pstree(1)`
- `ps -forest`

Programmer for overvåking av prosesser:

- `ps(1)`
- `pstree(1)`
- `top(1)`

Et spesialfilssystem som gir et “filaktig” grensesnitt til prosessenes ressurser:

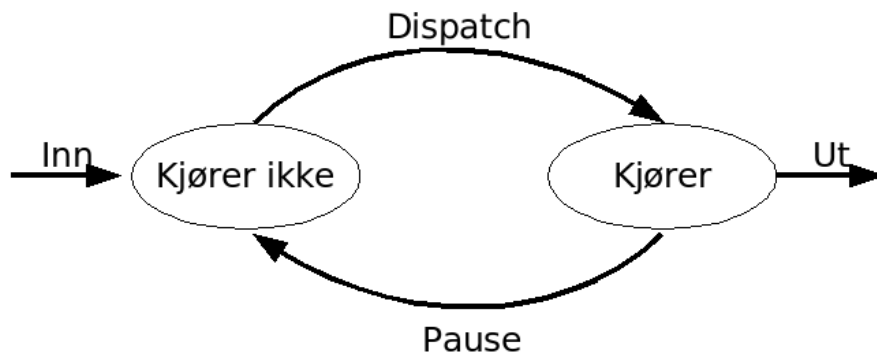
`/proc`

## 7.4 Noen tilstands-modeller

### modell med to tilstander

Kun en prosess kan kjøre om gangen (pr. prosessor)

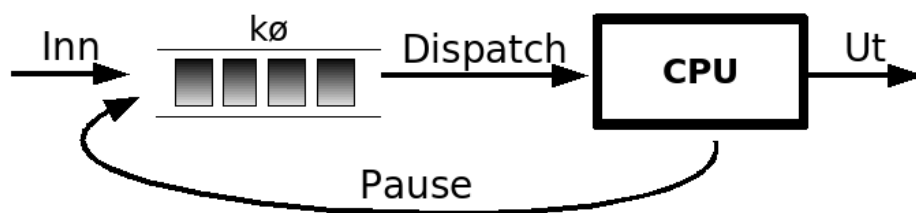
figur



- Kjørende (running) - en prosess pr. prosessor
- Klar (ready) - kan kjøre umiddelbart dersom valgt av "dispatcher"

### Kø i modell med to tilstander

figur



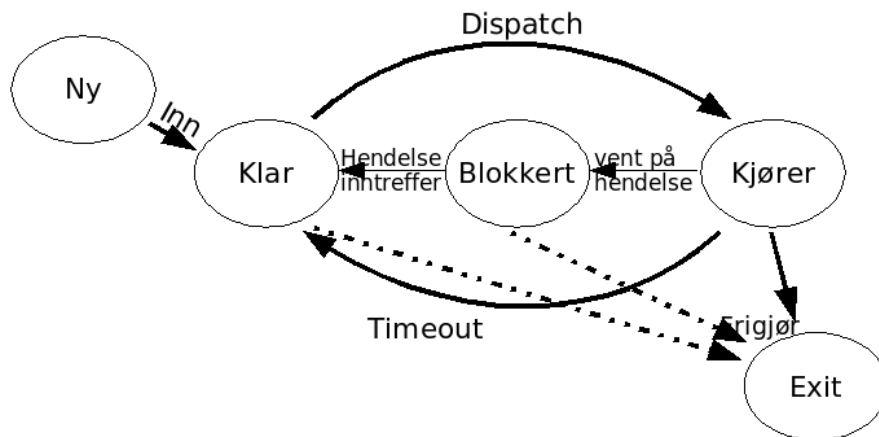
- En kø av ventende prosesser

Elementene i køen kan f.eks. være

- pekere til PCB'er
- sammenlenkede PCB'er

## modell med fem tilstander

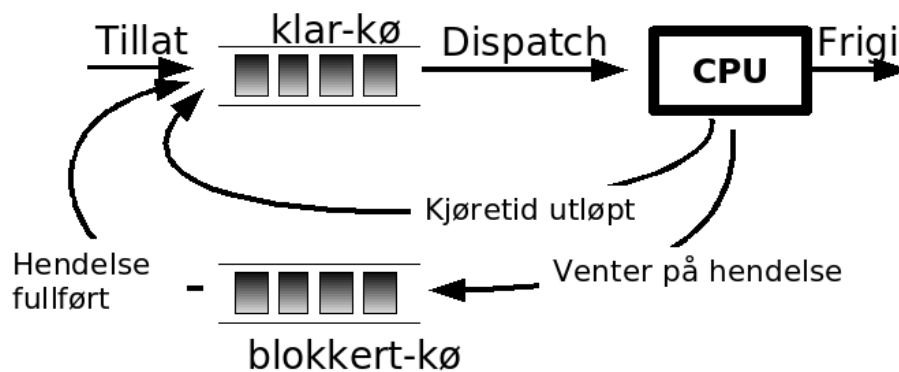
figur



- Ikke alle prosesser er klare til å kjøre til enhver tid.
- Blokkert (blocked) venter på hendelse (f.eks fullføring av I/O-operasjon)

## Fem tilstander – to køer

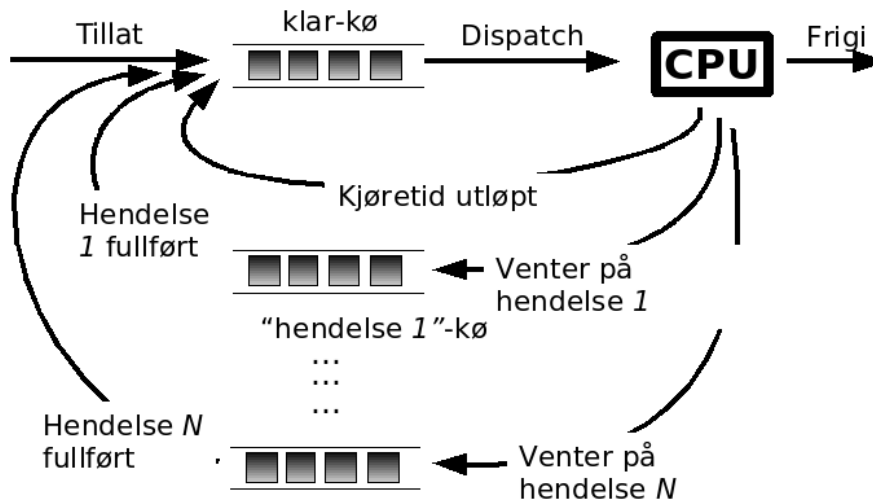
figur



- En kø for alle prosesser som er klare til å kjøre
- En kø for alle prosesser som venter på en hendelse

## Fem tilstander – mange køer

figur



- En kø for hver hendelse

### modell med “suspended” prosesser

- Plass til håndtering av flere prosesser
- ”Swapping” - prosessen flyttes helt eller delvis over i sekundærminne
- suspend

<http://debbie.hive.no/osyda50/figurer/3.9.png>

### prosess-/tråd-modellen i Linux

<http://debbie.hive.no/osyda50/figurer/4.18.png>

```
struct task_struct {
volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
...
}
```

<http://lxr.linux.no/#linux+v2.6.31/include/linux/sched.h#L1166>

## 7.5 Kontroll av prosesser

### Kjøremodus

For å beskytte OS’et har de fleste to modus

- kernel/system/control mode: Full tilgang
- user mode: Begrenset tilgang
- Modus bestemmes vanligvis av et bit et register i CPU.



**Modus settes typisk til kernel mode når:**

- Et avbrudd hender
- En prosess utfører et systemkall

**Modus settes typisk til user mode når:**

OS-rutine er fullført og kjøring av bruker-prosess fortsetter

### **Oppstart av prosesser**

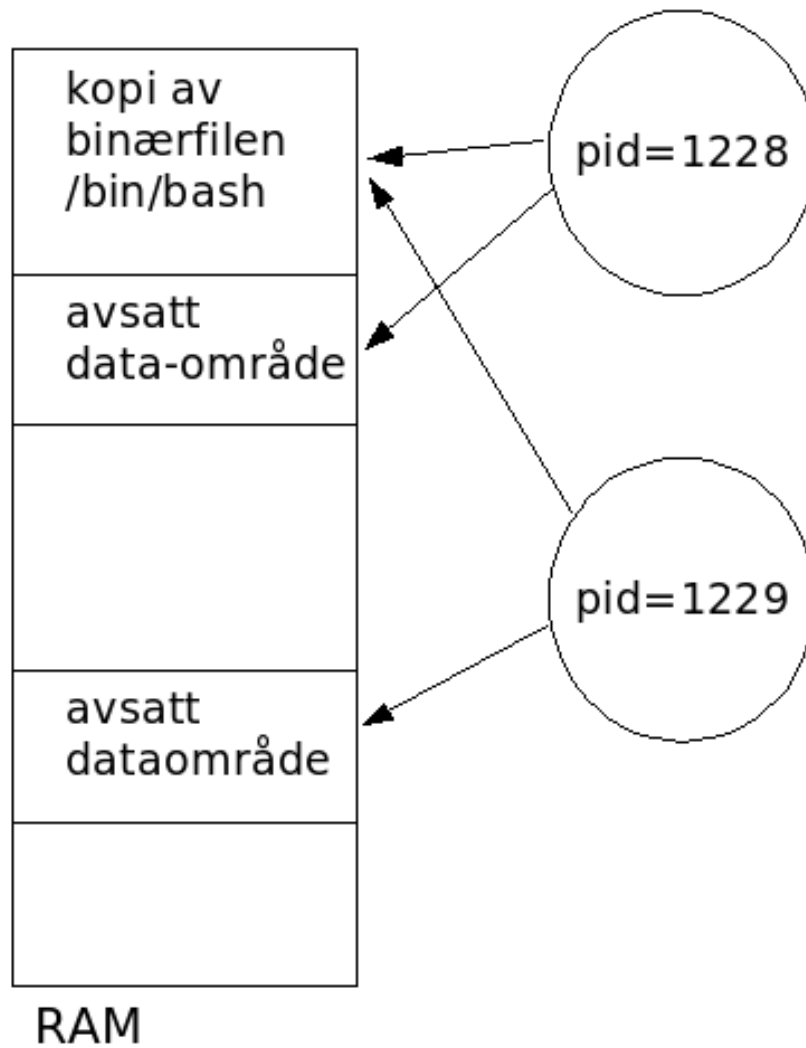
- Prosessen får tildelt unik ID og oppføring i prosesstabellen
- Tildeling av minne (til alle deler av “process image”)
- Initialisering av PCB
- Lenke prosessen inn i riktige strukturer
- evt. annet ...

**Eksempel: Oppstart av program i UNIX**

**Fase 1: fork(2)**

- Systemkallet fork() oppretter en ny prosess med nye minne-områder for data avsatt.

**figur**



- Innholdet i minne-områdene for data blir kopiert (egentlig copy on write).
- Fildeskriptor-tabellen blir kopiert – d.v.s. at barne-prosessen arver de åpne filene.

### fork(2)

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

Listing 7.5: eksempler/07/fork-eksempel\_I.c

```
1 #include <sys/types.h>
2 #include <unistd.h>
3 #include <stdio.h>
4
5 /*
6  Demonstrasjon av systemkallet fork()
```

```

7
8 */
9
10 int main(void)
11 {
12     fork();
13     printf("Jeg er en prosess nr. %d\n", getpid());
14     return 0;
15 }

```

Listing 7.6: eksempler/07/fork-eksempel\_II.c

```

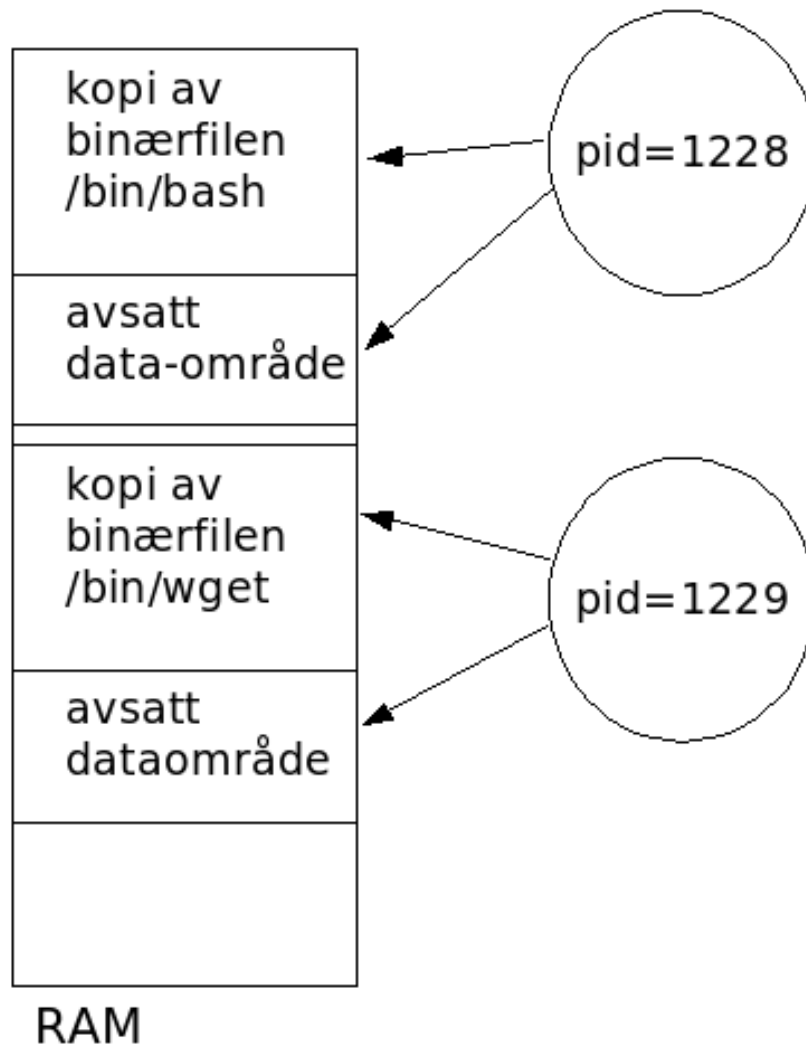
1 #include <unistd.h>
2 #include <stdio.h>
3
4 /*
5     Viser hvordan vi kan skille mellom foreldre
6     og barn etter en fork.
7
8 */
9
10 int main(void)
11 {
12     int forkretur;
13     forkretur = fork();
14
15     if (forkretur == 0)
16         printf("\nBarnet sier: \t");
17     else
18         printf("\nForeldren til %d, sier: \t", forkretur);
19
20     printf("Jeg er en prosess nr. %d\n", getpid());
21     return 0;
22 }

```

### Fase 2: exec(3)

- exec() endrer innholdet i prosessens minne-område.
- Et nytt program blir lastet. Data-området blir initiert, bortsett fra miljøet som beholdes som det er.

**figur**



```
#include <unistd.h>

extern char **environ;

extern char **environ;
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execl_e(const char *path, const char *arg, ..., char *const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

Listing 7.7: eksempler/07/exec-eksempel\_I.c

```
1 #include <unistd.h>
2
3 /*
4  * Demonstrasjon av systemkallet execl()
5  */
6 */
```

```

7
8 int main(void)
9 {
10
11     execl("/bin/ps","ps","-l", NULL);
12     return 0;
13
14 }
```

Listing 7.8: eksempler/07/exec-eksempel\_II.c

```

1 #include <unistd.h>
2 #include <stdio.h>
3
4 /*
5  Demonstrasjon av systemkallet execl()
6  i kombinasjon med fork.
7
8 */
9
10 int main(void)
11 {
12
13     int forkretur;
14
15     forkretur = fork();
16
17     if (forkretur == 0)
18         execl("/bin/ps","ps","-l", NULL);
19
20     printf("Jeg_er_foreldren_(%d).\n", getpid());
21     return 0;
22
23 }
```

## Bytte av prosess (Process switch)

Når kan kjørende prosess byttes ut?

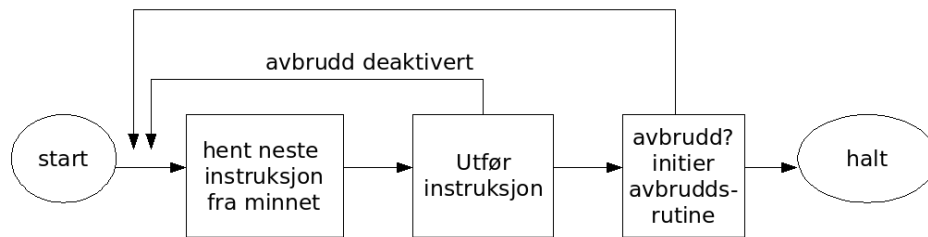
Når OS får kontroll, kan det bytte prosess

### Avbrudd (ekstern hendelse)

- tidsavbrudd
- I/O-avbrudd
- "page fault"
- "trap" - feil/unntak
- systemkall (intern hendelse)

### Bytte av modus (Mode switch)

Husk figur fra kap. 1 – fetch execute m/avbrudd



### Hvis en avbrudd er skjedd

- Programtelleren settes til starten av avbruddsrutinen.
- Modus settes til kernel mode.
- Informasjonen som trengs for å gjenoppta kjøringen av prosessen, prosessens kontekst, lagres i PCB (i delen kalt prosessortilstand)

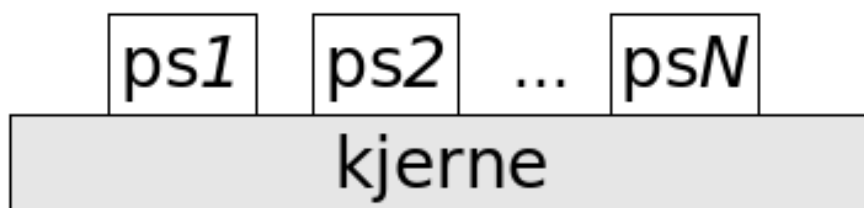
### Bytte av tilstand

Når en prosess forlater tilstanden “kjørende”:

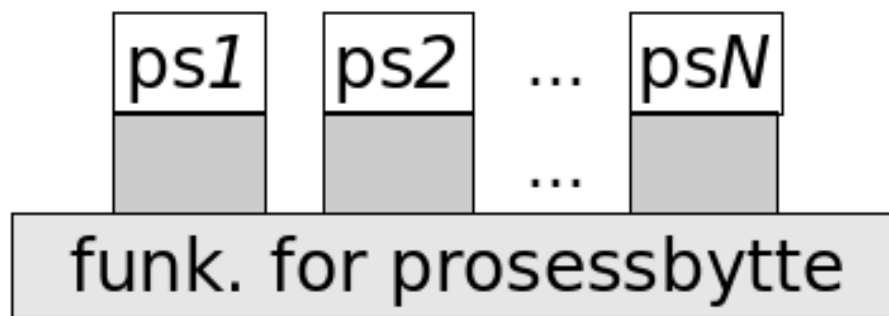
- Prosessor-kontekst lagres
- Oppdater PCB
- Flytt PCB til passende kø
- Velg ny prosess for kjøring
- Oppdater PCB til valgt prosess
- Oppdater strukturer for minnehåndtering
- Gjennopprett prosessortilstand

## 7.6 Alternativer for operativsystemets kjøring

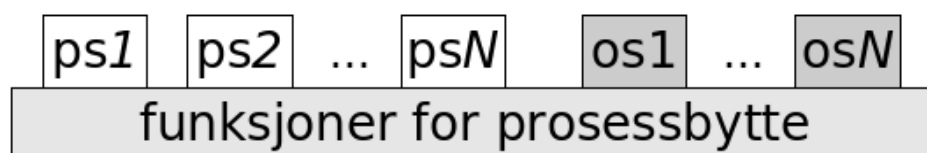
### Separat kjerne



## OS-funksjoner kjører i brukerprosessene



## OS-funksjoner kjører som separate prosesser



## 7.7 Oppgaver

### 7.1

- Hva tror du blir utskriften av dette programmet?

```
int main() {
fork();
fork();
write(1,"A",1);
return 0;
}
```

- Kompiler og kjør programmet. Ble utskriften som forventet?

### 7.2

- a) Ta utgangspunkt i eksemplet `open_eksempel.c` fra forrige tema. Endre det slik at det "forker" etter at filen er åpnet, men før kallet til `read()`.
- b) Kjør programmet og inspiser fildeskriptor-tabellen til både foreldreprosessen og barnet mens de har filen åpen. Er det noen forskjell i prosessenenes fildeskriptor-tabeller?
- c) Kan du finne hvilken tilstand prosessenene fra b) er i. Bruk `ps(1)` eller `top(1)`.

### 7.3

- Lag et C-program som går i en løkke og bruker `fork(2)` og `exec(3)` for å kjøre følgende kommandoer:

```
cut -f1 -d: /etc/passwd
cut -f2 -d: /etc/passwd
...
cut -f7 -d: /etc/passwd
```

### 7.4

I filsystemet `/proc` finnes en katalog for hver prosess på systemet. Inne i hver av disse katalogene, ligger finner du informasjon om prosessene. Dette oppgaven går ut på at du skal lage et shell-script som tar en prosessidentifikator (PID) som argument og skriver ut hvilken tilstand prosessen med oppgitt PID befinner seg i. Skriptet skal slå opp denne informasjonen i filsystemet `/proc`. Til denne oppgaven trenger du å bruke en editor, kommandoen `cut(1)` eller `grep(1)` og filsystemet `proc(5)`.

### 7.5

Lag et c-program som gjentatte ganger leser inn en kommando fra "standard-input" og kjører denne kommandoen i en egen prosess. Du trenger å bruke systemkallenene `fork(2)` og `exec(3)` for å få til dette.



# Kapittel 8

## 08-Tråder

### 8.1 Løsningsforslag

#### 7.3

#### Oppgave

- Lag et C-program som går i en løkke og bruker `fork(2)` og `exec(3)` for å kjøre følgende kommandoer:

```
cut -f1 -d: /etc/passwd
cut -f2 -d: /etc/passwd
...
cut -f7 -d: /etc/passwd
```

#### Løsning

Listing 8.1: losninger/07/7.3.c

```
1 #include <unistd.h>
2
3 int main () {
4     char s[] = "-f1 ";
5     int i;
6
7     for (i=1; i<=7; i++){
8         if ( fork() == 0 )
9             execl("/usr/bin/cut", "cut", s, "-d:", "/etc/passwd", NULL );
10
11         s[2]++;
12
13     }
14
15     return 0;
16
17 }
18
19 }
```

## 7.4

### Oppgave

I filsystemet `/proc` finnes en katalog for hver prosess på systemet. Inne i hver av disse katalogene, ligger finner du informasjon om prosessene. Denne oppgaven går ut på at du skal lage et shell-script som tar en prosessidentifikator (PID) som argument og skriver ut hvilken tilstand prosessen med oppgitt PID befinner seg i. Skriptet skal slå opp denne informasjonen i filsystemet `/proc`. Til denne oppgaven trenger du å bruke en editor, kommandoen `cut(1)` eller `grep(1)` og filsystemet `proc(5)`.

### Løsning

Listing 8.2: losninger/07/7.4.sh

```
1 #!/bin/bash
2
3 grep State /proc/$1/status
4 #cut -f3 -d" " /proc/$1/stat
```

## 8.2 Vi repeterer: To definisjoner av prosess:

### En enhet av aktivitet karakterisert av

- en en enkel sekvensiell tråd av utførelse
- en tilstand
- tilhørende system resursser
- «an address space with one or more threads executing within that address space, and the required system resources for those threads.»

## 8.3 Om prosesser og tråder

### Begrepet 'prosess' kan deles i to uavhengige deler

- eierskap til resursser
- skedulering/kjøring

Vi beholder prosess-begrepet og knytter det til den første delen.

### ”prosess/task”

- enhet for eierskap (til resursser)
- enhet for beskyttelse (beskyttelse-funksjon i OS)

## Til den andre delen knytter vi begrepet 'tråd'.

### "tråd/lettvekts-prosess"

- enhet for kjøring/skedulering
- kjøring følger en kjøringsti/"execution path/"trace"

[http://debbie.hive.no/osyda50/figurer/figur\\_1.26.png](http://debbie.hive.no/osyda50/figurer/figur_1.26.png)

- kjøringsti kan være sammenflettet med andre prosesskjøringer

### Ulike varianter

[http://debbie.hive.no/osyda50/figurer/figur\\_4.1.png](http://debbie.hive.no/osyda50/figurer/figur_4.1.png)

### Prosess-håndtering

[http://debbie.hive.no/osyda50/figurer/figur\\_4.2.png](http://debbie.hive.no/osyda50/figurer/figur_4.2.png)

Listing 8.3: eksempler/08/lokal\_vs\_global\_prosess.c

```

1 #include <unistd.h>
2 #include <stdio.h>
3
4 void *funksjon();
5 int global;
6
7 int main() {
8     int childpid;
9
10    childpid=fork();
11    if (childpid != 0)
12        wait(NULL);
13
14    funksjon();
15
16    return 0;
17 }
18
19
20 void *funksjon(){
21
22    int lokal = 0;
23
24    printf("%d: Startverdier:\tGlobal:_%d\tLokal:_%d.\nSkriv_nye:\n", getpid()
25        , global , lokal );
26    scanf("%d_%d", &global , &lokal);
27    printf("%d: Nye verdier:\tGlobal:_%d\tLokal:_%d\n", getpid() , global ,
28        lokal );

```

Listing 8.4: eksempler/08/lokal\_vs\_global\_traad.c

```

1 #include <unistd.h>
2 #include <stdio.h>
3 #include <pthread.h>

```

```

4
5  /*
6   Husk aa ta med argumentet "-pthread" ved kompilering.
7  */
8
9  void *funksjon ();
10 int global;
11
12 int main() {
13
14     pthread_t traad;
15
16     pthread_create(&traad, NULL, funksjon, NULL );
17     pthread_join(traad, NULL);
18
19     pthread_create(&traad, NULL, funksjon, NULL );
20     pthread_join(traad, NULL);
21
22     return 0;
23
24 }
25
26 void *funksjon () {
27
28     int lokal = 0;
29
30     printf ("%d: Startverdier: \tGlobal: %d\tLokal: %d.\nSkriv_nye:\n", getpid ()
31             , global, lokal );
32     scanf ("%d_%d", &global, &lokal);
33     printf ("%d: Nye_ verdier: \tGlobal: %d\tLokal: %d\n", getpid (), global,
34             lokal );
35 }

```

## 8.4 Hvorfor tråder?

- I noen tilfeller: Raskere programutførelse
- Opplever raskere – ting gjøres i «bakgrunnen» - bruker opplever ikke venting.
- Asynkrone hendelser (eks. sikkerhetkopiering).
- Lettere programutvikling – modularisering
- (De tre første punktene vil også gjelde prosesser som kjøres parallelt/konkurrerende)

[http://debbie.hive.no/osyda50/figurer/figur\\_4.3.png](http://debbie.hive.no/osyda50/figurer/figur_4.3.png)

## 8.5 Fordeler med tråder fremfor prosesser

- raskere å opprette en ny tråd i en eksisterende prosess, enn å opprette en ny prosess.
- raskere å terminere en tråd enn en prosess

- raskere å veksle mellom tråder enn prosesser
- raskere kommunikasjon mellom tråder (sparer to modus-bytte) - trenger ikke involvere kjernen

## 8.6 Handlinger som involverer alle tråder i en prosess

- suspensjon - veksle hele prosessens adresserom over på sekundærminne
- terminering av prosess

## 8.7 Trådenes tilstand

- Kjørende
- Klar
- Blokkert
- IKKE suspendert – kun hele prosesser suspenderes

## 8.8 Operasjoner knyttet til endring av tråders tilstand

- "spawn" - opprette
- blokkere - Dersom hele prosessen blokkeres mens en tråd venter på en hendelse? ... da forsvinner noe av vitsen med tråder
- avblokkere
- avslutt

[http://debbie.hive.no/osyda50/figurer/figur\\_4.4.png](http://debbie.hive.no/osyda50/figurer/figur_4.4.png)

## 8.9 Tråder på "user level" eller "kernel level"

User Level Threads (heretter også kalt brukertråder)

[http://debbie.hive.no/osyda50/figurer/figur\\_4.6.png](http://debbie.hive.no/osyda50/figurer/figur_4.6.png)

- brukerapplikasjonene håndterer trådene
- kjernen ikke involvert
- tråd-bibliotek ("threads library")

**ulemper med brukertråder**

- Ved blokkerende systemkall blokkeres hele prosessen - og dermed alle trådene
- Kan ikke utnytte flerprosesskjøring ("multiprocessing")

**Brukertrådenes ulemper kan omgås****implementere applikasjonen v.h.a. flere prosesser**

- løser begge problemene
- mister fordelene med tråder

**"jacketing"**

- løser blokkeringsproblemet
- gjør om til ikke-blokkerende
- egen tråd som kontrollerer om ressurs er ledig før systemkall utføres

**fordeler med brukertråder**

- ved veksling sparer to bytte av modus.
- muligheter for applikasjonsesifikk skedulering.
- uavhengig av OS-funksjonalitet - kan brukes på alle OS.

**Kernel Level Threads (heretter også kalt kjernetråder)**

[http://debbie.hive.no/osyda50/figurer/figur\\_4.6.png](http://debbie.hive.no/osyda50/figurer/figur_4.6.png)

- kjernen håndterer trådene
- API mot trådfunksjonalitet i kjernen
- Eks. Windows

**Fordeler med kjernetråder****trådene er skedulerings-enhet**

- ved blokkerende systemkall blokkeres kun tråden
- har dermed ingen av "user level"-trådens ulemper
- kjerne-rutiner kan også være flertrådkjørende

**Ulemper ved bytte av kjernetråder**

- tråd-veksling medfører modus-bytte

<http://debbie.hive.no/osyda50/figurer/04-tabeller-0.png>

## Kombinert

[http://debbie.hive.no/osyda50/figurer/figur\\_4.6.png](http://debbie.hive.no/osyda50/figurer/figur_4.6.png)

- Opprettelse av tråder skjer i "user space"
- "Bruker-trådene" i en prosess kan knyttes til like mange (el. færre) "kjerne-tråder"
- Hovedtyngden av trådenes skedulering og synkronisering gjøres i "user space"
- Kan unngå ulempene fra begge de to variantene
- Eks. Solaris

## 8.10 Tråder i Linux

- eldre versjoner - kun "brukertråder" (typisk pthreads)
- nyere versjoner: Lignende Solaris
- brukertråder knyttes til kjernetråder
- flere brukertråder som utgjør en brukerprosess kobles til flere kjerneprosesser
- deler samme gruppe ID
- unngår kontekst switch
- clone(2) brukes i implementasjon av tråder

### Eksempel på inspeksjon av tråder

Listing 8.5: eksempler/08/traadtest.c

```

1  #include <stdio.h>
2  #include <pthread.h>
3
4  /*
5
6   Husk aa ta med argumentet "-pthread" ved kompilering.
7  */
8
9  void *traadtest () {
10
11     printf("Trykk_enter.");
12     getchar();
13     return NULL;
14 }
15
16 int main () {
17
18     pthread_t traad;
19     pthread_create(&traad, NULL, traadtest, NULL);
20     pthread_join(traad, NULL);
21     return 0;
22
23 }
```

- `ps tree`
- `ps tree -c`
- `ps -L`
- `ps -Lf`

## 8.11 Oppgaver

### 8.1

- a) Endre programmet eksemplet `openeksempel.c` slik at filen åpnes i en egen tråd.
- b) Inspiser prosessens fildeskriptortabell, mens programmet kjører.

### 8.2

- Nettleseren Google Chrome er implementert slik at hver fane ("tab") kjører som en egen prosess. Finn ut hvorfor dette valget er gjort.

<http://www.google.com/googlebooks/chrome/>

### 8.3

#### a)

Lag et C-program som starter en ny prosess og venter til barnet har terminert, før det selv terminerer. Bruk `fork()` for å starte ny prosess og `wait()` for å få foreldreplassen til å vente på barnet. Barneprosessen skal kun terminere så fort som mulig.

#### b)

Utvid programmet i a) slik at det starter og venter 10000 ganger.

#### c)

Ta en kopi av programmet i b) og endre kopien, slik at den starter og venter på en tråd i stedet for en prosess. Bruk `pthread_create()` for å starte tråden og `pthread_join()` for å vente på at tråden er ferdig-kjørt. Pass på at også tråden returnerer så fort som mulig.

#### d)

- Med en null-fork menes å opprette, skedulere, utføre og fullføre en prosess/tråd som starter en null-prosedyre. Programmene i b) og d) kan brukes til å lage et gjennomsnitt av hvor lang tid en null-fork av en prosess og en tråd tar. Mål og beregn gjennomsnittelig tid for nullfork av både tråder og prosesser. Bruk `clock()` og konstanten `CLOCKS_PER_SEC`.



**Eksempel på tidtaking:**

```
clock_t start, stopp;  
double sekunder;  
start = clock();  
...  
stopp = clock();  
'sekunder= (double) (stopp-start)/CLOCKS_PER_SEC;
```



# Kapittel 9

## 10 Samtidig behandling

### 9.1 Produsent-konsument-problemet (semaforer nå igjen...)

#### Problemformulering

- Data lagres i et lager
- Produsent putter data inn i lageret
- Konsument henter data ut fra lageret

#### To varianter

- 1. lager med ubegrenset størrelse (urealistisk)
- 2. lager med begrenset størrelse (“Bounded Buffer Problem”)

#### Eksempel med semaforer

Listing 9.1: eksempler/10/produsent-konsument\_1.c

```
1 #include <pthread.h>
2 #include <unistd.h>
3 #include <stdio.h>
4
5 void *produser ();
6 void *konsumer ();
7
8 int lager=0;
9
10 int main () {
11     pthread_t konsument , produsent;
12
13     pthread_create(&konsument , NULL, konsumer , NULL);
14     pthread_create(&produsent , NULL, produser , NULL);
15
16     while (1) {
17
18
```

```

19     sleep(1);
20     printf("Lager:_%d\n", lager);
21 }
22
23     return 0;
24 }
25
26 void *produser () {
27
28     while(1)
29         lager++;
30 }
31
32 void *konsumer () {
33
34     while(1)
35         lager--;
36 }

```

Listing 9.2: eksempler/10/produsent-konsument\_2.c

```

1 #include <pthread.h>
2 #include <unistd.h>
3 #include <stdio.h>
4
5 #define LAGERSTR 100000
6
7 void *produser ();
8 void *konsumer ();
9
10 pthread_mutex_t laas= PTHREAD_MUTEX_INITIALIZER;
11 int lager=0;
12
13 int main () {
14
15     pthread_t konsument, produsent;
16
17     pthread_create(&konsument, NULL, konsumer, NULL);
18     pthread_create(&produsent, NULL, produser, NULL);
19
20     while(1){
21         sleep(1);
22         printf("Lager:_%d\n", lager);
23     }
24
25     return 0;
26 }
27
28 void *produser () {
29
30     while(1){
31
32         pthread_mutex_lock(&laas);
33
34         if (lager < LAGERSTR)
35             lager++;
36
37         pthread_mutex_unlock(&laas);
38

```

```

39     }
40 }
41
42 void *konsumer () {
43
44     while (1) {
45
46         pthread_mutex_lock(&laas);
47
48         if (lager > 0)
49             lager--;
50
51         pthread_mutex_unlock(&laas);
52     }
53 }

```

Listing 9.3: eksempler/10/produsent-konsument\_3.c

```

1  #include <semaphore.h>
2  #include <pthread.h>
3  #include <unistd.h>
4  #include <stdio.h>
5
6  #define LAGERSTR 100000
7
8  void *produser ();
9  void *konsumer ();
10
11 pthread_mutex_t laas = PTHREAD_MUTEX_INITIALIZER;
12 sem_t ledige_plasser;
13 sem_t klare_data;
14 int lager=0;
15
16 int main () {
17
18     pthread_t konsument, produsent;
19
20     sem_init(&ledige_plasser, 0, LAGERSTR );
21     sem_init(&klare_data, 0, 0 );
22
23     pthread_create(&konsument, NULL, konsumer, NULL);
24     pthread_create(&produsent, NULL, produser, NULL);
25
26     while (1) {
27         sleep(1);
28         printf("Lager: %d\n", lager);
29     }
30
31     return 0;
32 }
33
34 void *produser () {
35
36     while (1) {
37         sem_wait(&ledige_plasser);
38
39         pthread_mutex_lock(&laas);
40
41         lager++;

```

```

42
43     pthread_mutex_unlock(&laas);
44
45     sem_post(&klare_data);
46 }
47 }
48
49 void *konsumer() {
50
51     while(1) {
52         sem_wait(&klare_data);
53
54         pthread_mutex_lock(&laas);
55
56         lager--;
57
58         pthread_mutex_unlock(&laas);
59
60         sem_post(&ledige_plasser);
61     }
62 }

```

## 9.2 Vranglås

### Definisjon av vranglås

Permanent blokkering av et sett prosesser som enten konkurrerer om system-resursser eller kommuniserer med hverandre.

### Eksempel:

<http://debbie.hive.no/osyda50/figurer/6.1.png>

### Hvordan kan vranglås oppstå?

#### Kriterier for vranglås

- 1. Gjensidig utelukkelse
- 2. Hold og vent
- 3. Ingen "preemption"
- 4. Sirkulær venting
- Vranglås kan oppstå dersom de tre første kriteriene er tilstede:
- Dersom en vranglås oppstår, er i tillegg det fjerde kriterium oppfylt.

### Resurss-allokerings-graf

- Prosess er sirkel.
- Resurss er firkant

- En prikk inne i firkanten for hver instans av resurssen.
- Pil fra Prosess til resurss: En resurssforespørsel som ikke er innvilget ennå
- Pil fra resurss-instans (prikk) til prosess: Prosessen holder instansen
- Sykler i grafen betyr at vi har sirkulær venting.

<http://debbie.hive.no/osyda50/figurer/6.5.png>

Hvordan ser resurss-allokerings-grafen ut for figurene 6.1 a) og 6.1 b) ?

## Håndtering av vranglås

### Forebygging (prevention)

#### Indirekte - fjerne et av de tre første punkt i kriteriene

**Gjensidig utelukkelse** Dette må vi ha

#### Hold og vent:

- Prosess venter (blokkert) til den får tilgang til alle resursene den kommer til å trenge på en gang.
- Unødvendig mye venting.
- Ofte vanskelig å vite på forhånd hvilke resursser en prosess kommer til å trenge.

#### “Preemption”

- Tvinge prosesser til å gi slipp på resurss.
- Dette kan gjøres med resursser hvis tilstand lett kan lagres og gjenopprettes (som f.eks. en prosessor)

#### Direkte - hindre det fjerde punktet

- hindre det fjerde punktet, sirkulær venting, ved å definere en linjær rekkefølge av resursser slik at:
- dersom en prosess har en resurss, kan den kun få tilgang til resursser som kommer etter i rekkefølgen

#### Unngåelse (avoidance)

- Resurss-tilgang gis kun dersom det ikke vil oppstå sirkulær venting. Om det vil oppstå sirkulær venting, avgjøres ved tidspunkt for (1) forespørsel om prosessoppsettelse eller (2) ved resurssforespørsel fra en eksisterende prosess.
- Tillater de tre nødvendige betingelsene for vranglås.

## Begrensninger

- Prosessens fremtidige forespørsler må være kjent på forhånd.
- Det må ikke være noen krav til synkronisering av prosessene. De må være uavhengige.
- Antall allokerbare resursser må være bestemt.
- Ingen prosess må avslutte mens den holder resursser.

## Unngår

- "preemption".
- tilbakerulling (rollback)

## Oppdagelse (detection)

- Algoritme for å oppdage vranglås

## Måter å løse opp vranglås:

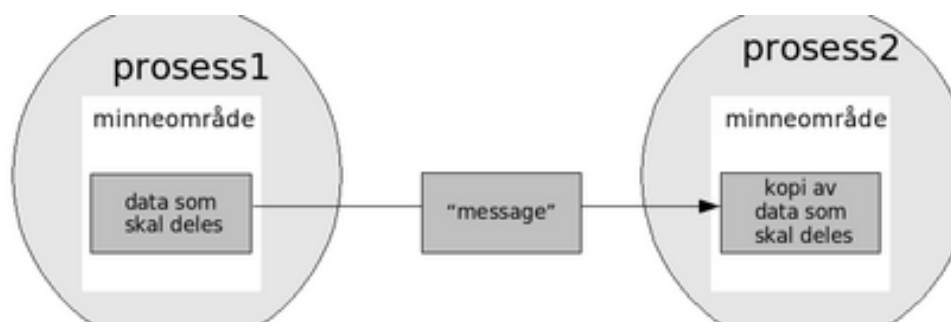
- Avslutte alle de fastlåste prosessene
- Tilbakeføre de fastlåste prosessene til en tidligere tilstand ("rollback") og starte dem igjen.
- Avslutt en og en prosess i låsen, inntil låsen løses opp.
- "Preempt" en og en resurss i låsen, inntil låsen løses opp.

## 9.3 "Message Passing"

### Krav til samhandling av prosesser

- synkronisering - for gjensidig utelukkelse
- kommunikasjon - dele informasjon

"Message Passing" håndterer både synkronisering og kommunikasjon





## Primitiver

- send(destination, message)
- receive(source, message)

<http://debbie.hive.no/osyda50/figurer/5.18.png>  
<http://debbie.hive.no/osyda50/figurer/5.19.png>

- Passer til distribuerte systemer

## Eksempler

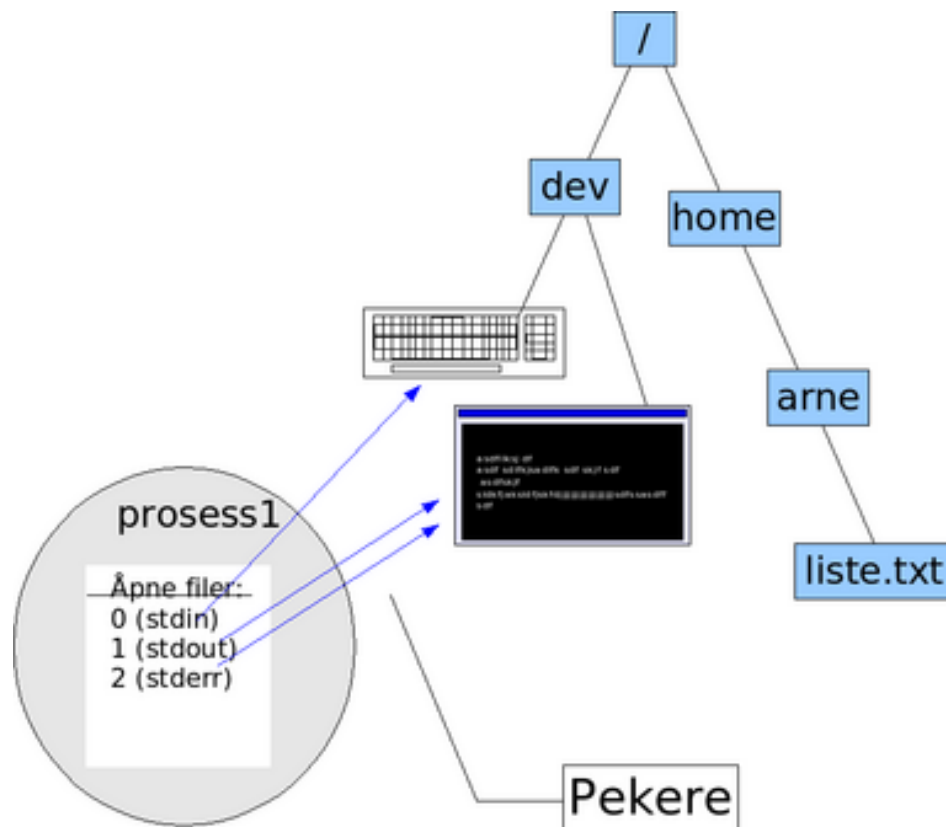
### Eksempler fra UNIX

- message queues
- rør (pipes)
- navgitte rør (FIFOs/named pipes)

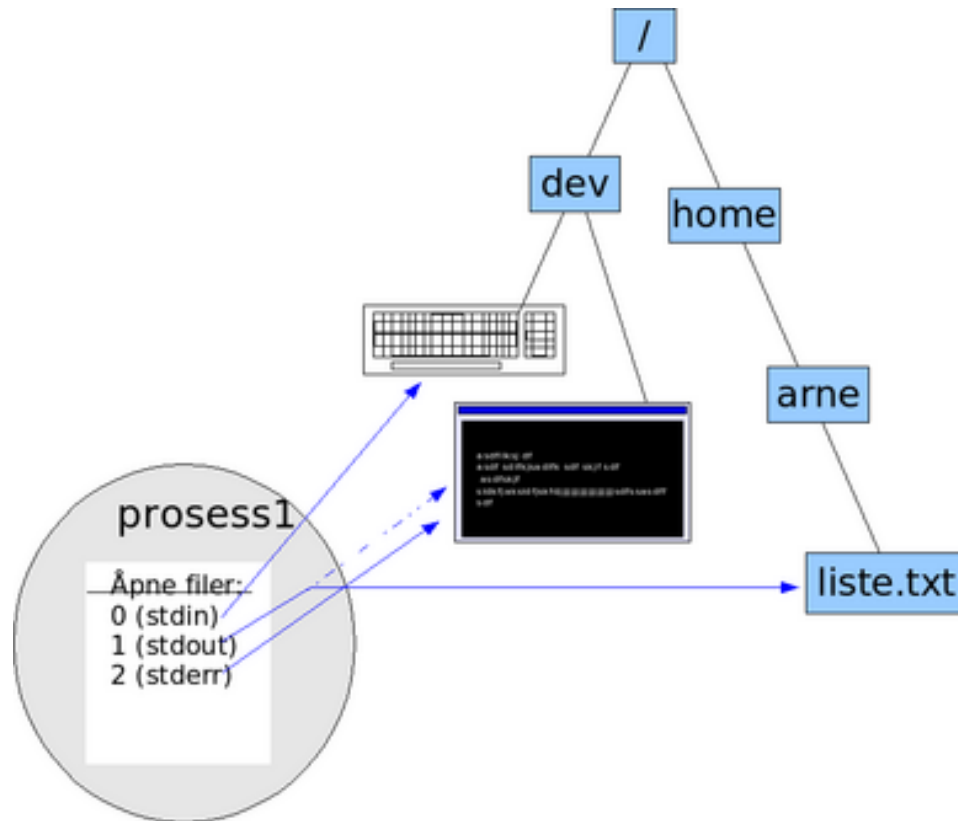
De to siste ser vi nærmere på...

### Rør og FIFO'er

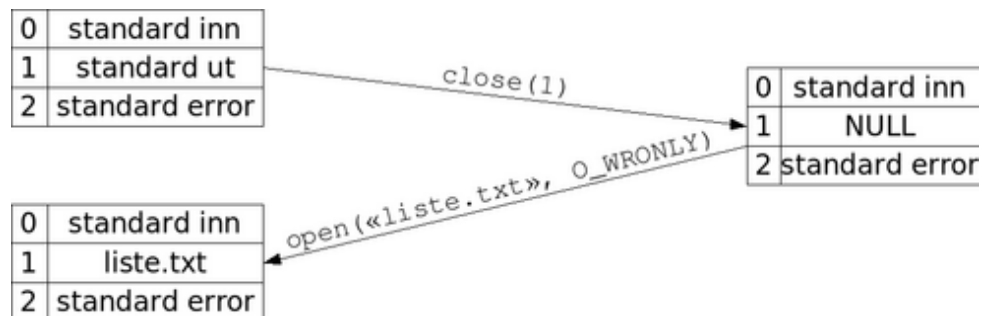
**Rør («pipes»)** Redirigering (nå igjen...) Standard inn og ut



Redirigering til fil



Redirigering v.h.a. `close()` og `open()` figur



- Når en fil åpnes, får den tildelt den ledige fildeskriptoren med lavest verdi.

Listing 9.4: eksempler/10/redirigering.c

```

1 #include <fcntl.h>
2 #include <unistd.h>
3
4 /*
5  * Demonstrasjon av redirigering.
6  *
7  * Programmet tilsvareer kommandolinjen
8  * "ls > liste.txt".
9  */
10
11
12 int main(void)
13 {
14

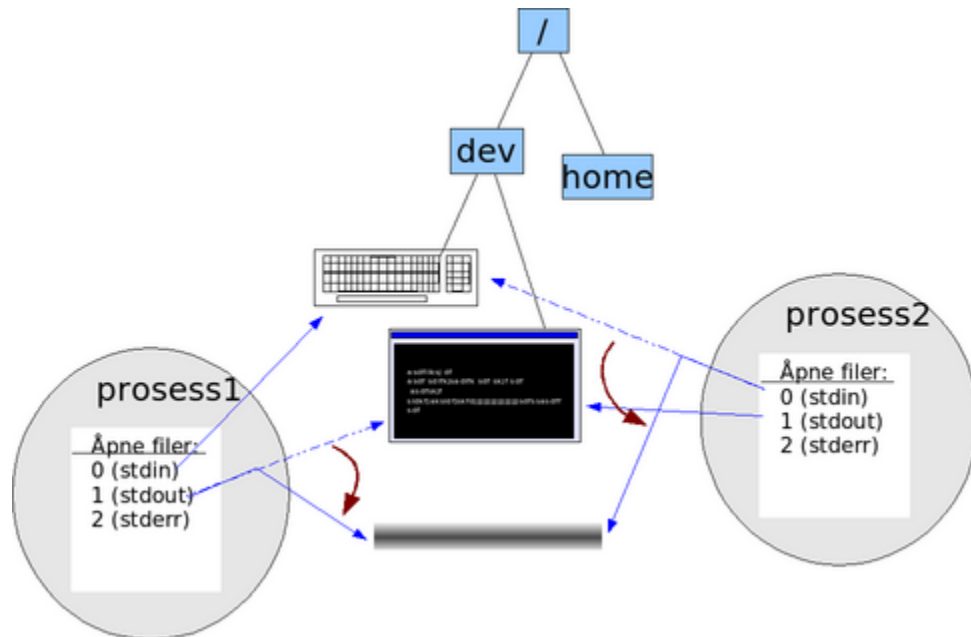
```

```

15 // Stenger standard ut
16 close(1);
17
18 if ((open("liste.txt", O_WRONLY | O_CREAT)) > -1)
19     execl("/bin/ls", "ls", NULL);
20
21 return 1;
22
23 }

```

Redirigering til annen prosess v.h.a.figur



rør («pipes»)Duplisering av fildeskriptor

```

#include <unistd.h>
int dup(int oldfd);
int dup2(int oldfd, int newfd);

```

- Manualsida: dup(2)

Opprettelse av rør

```

#include <unistd.h>int pipe(int filedes[2]);

```

- Manualsida: pipe(2)

Eksempler

Listing 9.5: eksempler/10/pipe-eksempel\_I.c

```

1 #include <unistd.h>
2
3 /*
4  * Demonstrerer bruk av rør, v.h.a. systemkallene
5  * pipe() og dup().
6  *
7  * Programmet gjør det samme som kommandolinjen

```

```

8   "ls | rev ".
9
10  ('rev' skriver tekstlinjer baklengs)
11
12  */
13
14  int main (void)
15  {
16      int fda[2];
17
18      // Oppretter et rør
19      pipe(fda);
20
21      if(fork() == 0) {
22
23          // Stenger standard utgang
24          close(1);
25
26          // Dupliserer skrive-enden (inngangen) av røret
27          dup(fda[1]);
28
29          close(fda[0]); // stenger lese-enden (utgangen) av røret
30          close(fda[1]); // stenger skrive-enden (inngangen) av røret
31
32          execl("/bin/ls", "ls", 0);
33
34      }
35
36      else {
37
38          // stenger standard inngang
39          close(0);
40
41          // dupliserer lese-enden (utgangen) av røret
42          dup(fda[0]);
43
44          close(fda[0]); // stenger lese-enden (utgangen) av røret
45          close(fda[1]); // stenger skrive-enden (inngangen) av røret
46
47          execl("/usr/bin/rev", "rev", 0);
48
49      }
50
51      return 1;
52
53  }

```

Listing 9.6: eksempler/10/pipe-eksempel\_II.c

```

1  #include <unistd.h>
2
3  /*
4   Demonstrerer bruk av rør, v.h.a. systemkallene
5   pipe() og dup2().
6
7   Programmet gjør det samme som kommandolinjen
8   "ls | rev ".
9
10  ('rev' skriver tekstlinjer baklengs)

```

```

11
12 */
13
14 int main (void)
15 {
16     int fda[2];
17
18     // Oppretter et rør
19     pipe(fda);
20
21     if(fork()==0) {
22
23         // stenger standard utgang
24         // og dupliserer skrive-enden (inngangen) av røret
25         dup2(fda[1], 1);
26
27         close(fda[0]); // stenger lese-enden (utgangen) av røret
28         close(fda[1]); // stenger skrive-enden (inngangen) av røret
29
30         execl("/bin/ls", "ls", 0);
31     }
32
33     else {
34
35         // Stenger standard inn og
36         // dupliserer lese-enden (utgangen) av røret
37         dup2(fda[0], 0);
38
39         close(fda[0]); // stenger lese-enden (utgangen) av røret
40         close(fda[1]); // stenger skrive-enden (inngangen) av røret
41
42         execl("/usr/bin/rev", "rev", 0);
43     }
44
45 }
46
47 return 1;
48
49 }

```

1. Vanlige filer
2. Kataloger
3. Karakter-  
enhetsfiler
4. Blokk-enhetsfiler
5. Symbolske lenker
6. "Local domain  
sockets"
7. **Navngitte rør  
(FIFO's)**

Hvorfor er så mye  
forskjellig  
representert som  
filer?

Bruk av navngitt rør på kommandolinjen For å opprette en navgitt rør fra kommandolinjen: `mkfifo navngitt_ror` Eksempel:

- Skrive og lese til fila fra to ulike konsollvinduer:
- lese: `cat navngitt_ror`
- skrive: `cat > navngitt_ror`

Opprettelse av node i et filsystem

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
```

```
int mknod(const char *pathname, mode_t mode, dev_t dev);
```

- mode: Kombinasjon (bitvis eller) av rettighetsmaske og filtype.
- dev: NULL hvis ikke filtypen er blokk-enhet/karakter-enhet
- Manualsider: `mknod(2)`

Eksempler

Listing 9.7: `eksempler/10/navngitt_pipe.c`

```
1 #include <sys/types.h>
2 #include <sys/stat.h>
3 #include <fcntl.h>
4 #include <unistd.h>
5
6 /*
7  Program som gjør det samme som kommandolinjen "ls | rev",
8  v.h.a. et navngitt rør.
9
10 */
```

```

11
12 int main (void)
13 {
14
15     int fd;
16
17     // Oppretter navngitt rør
18     mknod("navngitt_ror", S_IFIFO | S_IRUSR | S_IWUSR, 0);
19
20     if(fork()==0) {
21
22         fd=open("navngitt_ror", O_WRONLY);
23
24         // stenger stdout og dupliserer filhåndtaket
25         dup2(fd, 1);
26         close(fd);
27
28         execl("/bin/ls", "ls", NULL);
29
30     }
31
32     else {
33
34         fd=open("navngitt_ror", O_RDONLY);
35
36         // stenger stdin og dupliserer filhåndtaket
37         dup2(fd, 0);
38         close(fd);
39
40         execl("/usr/bin/rev", "rev", NULL);
41
42     }
43
44     return 1;
45
46 }

```

Listing 9.8: eksempler/10/pipe-tjener.c

```

1 #include <fcntl.h>
2 #include <unistd.h>
3 #include <sys/wait.h>
4 #include <stdio.h>
5
6 /*
7     Eksempel på enveiskommuniserende tjener
8
9 */
10
11 int main(void) {
12
13     int fd;
14
15     if((fd=open("navngitt_ror", O_RDONLY)) < 0) {
16         perror("Feil ved åpning av FIFO:");
17         return 1;
18     }
19
20     // Redirigerer

```

```
21 dup2(fd, 0);
22 close(fd);
23
24 while(1)
25
26     if (fork() == 0)
27         execl("/usr/bin/rev", "rev", NULL);
28     else
29         (void)wait(NULL);
30
31 return 2;
32
33 }
```

## 9.4 Oppgaver

### 10.1

- Observer og forklar hva som skjer når du gjør følgende:

a)

Lag en FIFO i arbeidskatalogen med:

```
mkfifo fifo-fil.
```

b)

Åpne FIFO'n i en grafisk nettleser på samme maskin som du lagde FIFO'n:

```
firefox fifo-fil &
```

c)

I et konsoll-vindu, skriv:

```
echo Hallo > fifo-fil
```

### 10.2

- Eksemplene produsent-konsument\_1.c, produsent-konsument\_2.c og produsent-konsument\_3.c gjør ingenting.

a)

- Endre programmene slik at de jobber mot en tabell (array) av typen char.
- La produsenten skrive bokstaver inn i tabellen.
- La konsumenten hente bokstavene ut av tabellen og skrive dem ut på standard utgang ("stdout").



**b)**

Skriv om programmet i a) slik at du i stedet for semaforer og `char[]`, bruker et navngitt rør.

### 10.3

Lag et program med to tråder og to resursser, slik at betingelsene for vranglås er tilstede. Som resursser kan dere bruke enkle variabler f.eks. en `'char'` eller `'int'`. Bruk semaforer til gjensidig utelukkelse.

### 10.4

- Eksemplet `pipe-tjener.c` har kun enveis-kommunikasjon. Gjør den toveis ved å:
- Opprett en FIFO til, og rediriger utskriften fra `pipe-tjeneren` inn i den.
- Lag en klient til `pipe-tjeneren`.



# Kapittel 10

## 11 - minne og signaler

### 10.1 Løsninger

#### 10.2

##### Oppgave

- Eksempelene produsent-konsument\_1.c, produsent-konsument\_2.c og produsent-konsument\_3.c gjør ingenting.

a)

- Endre programmene slik at de jobber mot en tabell (array) av typen char.
- La produsenten skrive bokstaver inn i tabellen.
- La konsumenten hente bokstavene ut av tabellen og skrive dem ut på standard utgang ("stdout").

##### Løsningsforslag på oppgave 10.2a)

Listing 10.1: losninger/10/sirkeltabell.c

```
1 #include <unistd.h>
2 #include <fcntl.h>
3
4 int main () {
5
6     char abc[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
7
8     int a_len = sizeof(abc);
9     int b = 0;
10    int i;
11
12    for (i=0; i< 5*a_len; i++) {
13
14        write( 1, abc+b, 1 );
15        b++;
16        b %= a_len;
17    }
18
```

```

19  write( 1, "\n", 1);
20
21  return 0;
22  }

```

Listing 10.2: losninger/10/10.2.a.c

```

1  #include <semaphore.h>
2  #include <pthread.h>
3  #include <unistd.h>
4  #include <stdio.h>
5
6  #define LAGERSTR 6
7
8  /*
9   *Husk -pthread ved kompilering med gcc'
10  */
11
12
13  void *produser ();
14  void *konsumer ();
15
16  pthread_mutex_t laas= PTHREAD_MUTEX_INITIALIZER;
17
18  sem_t ledige_plasser;
19  sem_t klare_data;
20
21  char lager [LAGERSTR];
22
23  int inn = 0;
24  int ut  = 0;
25
26  int main () {
27
28      pthread_t konsument;
29
30      sem_init(&ledige_plasser, 0, LAGERSTR );
31      sem_init(&klare_data, 0, 0 );
32
33      pthread_create(&konsument, NULL, konsumer, NULL);
34
35      // produsent
36
37      while(1) {
38
39          sem_wait(&ledige_plasser);
40          pthread_mutex_lock(&laas);
41
42          if ( 1 > read(0, lager+inn, 1) )
43              break;
44
45          inn++;
46          inn%=LAGERSTR;
47
48          pthread_mutex_unlock(&laas);
49          sem_post(&klare_data);
50
51      }
52

```

```

53     sleep(1);
54     return 0;
55 }
56 }
57
58 void *konsumer () {
59
60     while(1) {
61
62         sem_wait(&klare_data);
63         pthread_mutex_lock(&laas);
64
65         printf ("%c", lager[ut]);
66         //write(1, lager+ut, 1);
67
68         ut++;
69         ut%=LAGERSTR;
70
71         pthread_mutex_unlock(&laas);
72         sem_post(&ledige_plasser);
73     }
74 }

```

## 10.2 Repetisjon: Pipes

Listing 10.3: eksempler/11/pipe\_illustrated.c

```

1  #include <sys/types.h>
2  #include <sys/wait.h>
3  #include <unistd.h>
4  #include <stdio.h>
5
6  void skriv_fdtab(char *overskrift);
7
8  int main () {
9
10     /* Bruker funksjonen skriv_fdtab()
11        til aa vise fildeskriptortabellen
12        til en prosessene underveis i
13        eksemplet 'pipe-eksempel_1.c'
14
15        To do:
16        Programmet trenger synkronisering
17        av utskrift til konsollet.
18        */
19
20     int fda[2];
21
22     skriv_fdtab("Ved_oppstart");
23     pipe(fda);
24     skriv_fdtab("Etter_pipe(fd)");
25
26
27
28     if(fork() == 0) {
29
30         // Stenger standard utgang
31         close(1); skriv_fdtab("Etter_close(1)");

```

```

32
33 // Dupliserer skrive-enden (inngangen) av roeret
34 dup(fda[1]); skriv_fdtab("Etter_dup(fda[1])");
35
36 // stenger lese-enden (utgangen) av roeret
37 close(fda[0]); skriv_fdtab("Etter_close(fda[0])");
38
39 // stenger skrive-enden (inngangen) av roeret
40 close(fda[1]); skriv_fdtab("Etter_close(fda[1])");
41
42 execl("/bin/ls", "ls", NULL);
43
44 }
45
46 else {
47
48 // stenger standard inngang
49 close(0); skriv_fdtab("Etter_close(0)");
50
51 // dupliserer lese-enden (utgangen) av roeret
52 dup(fda[0]); skriv_fdtab("Etter_dup(fda[0])");
53
54 // stenger lese-enden (utgangen) av roeret
55 close(fda[0]); skriv_fdtab("Etter_close(fda[0])");
56
57 // stenger skrive-enden (inngangen) av roeret
58 close(fda[1]); skriv_fdtab("Etter_close(fda[1])");
59
60 execl("/usr/bin/rev", "rev", NULL);
61
62 }
63
64 return 1;
65
66 }
67
68
69 void skriv_fdtab(char *overskrift) {
70
71 /*Funksjonen skriver ut fil-deskriptor-tabellen
72 til prosessen som kaller den
73 */
74
75 char sti[50];
76 int pid = getpid();
77
78 sprintf(sti, "/proc/%d/fd", pid);
79
80 if ( fork() == 0 ){
81 dup2(2,1); // Soerger for at utskriften kommer paa konsollet
82 printf("\n\n%s(%d)\n—————\n", overskrift, pid);
83 fflush(NULL);
84 sleep(1);
85 execl("/bin/ls", "ls", "-l", sti, NULL);
86
87 }
88
89 wait(NULL);

```

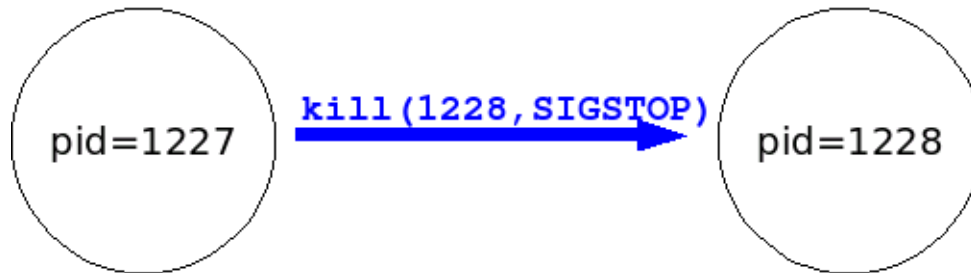
```
90 }
91 }
```

## 10.3 Signaler

- Prosesser kan kommunisere v.h.a. signaler.

### Signaler i linux

figur



```
struct task_struct {
    ...
    struct signal_struct *signal;
    ...
}
```

UNIX-programmer for å sende signaler:

- kill(1)
- killall(1)
- Oversikt over de ulike signalene finnes i manualen, seksjon 7: man 7 signal

Sende et signal til en prosess

```
#include <types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
```

- kill(2)

Listing 10.4: eksempler/11/kill-eksempel.c

```
1 #include <stdio.h>
2 #include <signal.h>
3 #include <unistd.h>
4
5 /*
6  Demonstrasjon av kill(2)
7
8 */
```

```

9
10 int main(void)
11 {
12
13     int forkretur, den_andres_pid, i;
14     char *tekst;
15
16     forkretur=fork();
17
18     if(forkretur==0){
19         tekst="BARN";
20         den_andres_pid = getppid();
21     }
22
23     else{
24         tekst="FORELDRE";
25         den_andres_pid = forkretur;
26     }
27
28     for (i = 0; i < 10; i++){
29         printf("%s\n", tekst);
30         sleep(1);
31         kill(den_andres_pid, SIGCONT);
32         kill(getpid(), SIGSTOP);
33     }
34
35     return 0;
36 }

```

### Oppfange signaler

```
#include <signal.h>
```

```

int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
struct sigaction {
    ...
    void (*sa_handler)(int);
    sigset_t sa_mask;
    int sa_flags;
    ...
}

```

- sigaction(2)

Listing 10.5: eksempler/11/signal-eksempel.c

```

1 #include <stdio.h>
2 #include <signal.h>
3 #include <unistd.h>
4
5 /*
6  Demonstrasjon av hvordan utfoere
7  egendefinert rutine ved mottak
8  av signal.
9

```



```

10 */
11
12 void signalrutine(int sig)
13 {
14     printf("/\\");
15 }
16
17 int main(void)
18 {
19
20     struct sigaction handling;
21
22     // Skriver ut egen pid
23     printf("%d\\n", getpid());
24
25     // Initierer sigaction-struktur
26     handling.sa_handler = signalrutine;
27     (void) sigemptyset(&handling.sa_mask);
28     handling.sa_flags = 0;
29
30     // Aktiverer ny signalrutine
31     sigaction(SIGINT, &handling, 0);
32
33     while(1){
34         printf(".");
35         fflush(NULL);
36         sleep(1);
37     }
38
39     return 0;
40
41 }

```

### Vente på terminering av en prosess

```

#include <types.h>
#include <wait.h>

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);

```

- wait(2)

### Eksempler

Listing 10.6: eksempler/11/konkurranse.c

```

1 #include <sys/types.h>
2 #include <unistd.h>
3 #include <stdio.h>
4
5 /*
6  Demonstrasjon av konkurrerende
7  kjoering av to prosesser.
8
9  */

```

```

10
11 int main(void)
12 {
13
14     int forkretur, i, j;
15     char *tekst;
16
17     forkretur=fork();
18
19     if(forkretur==0)
20         tekst="BARN";
21     else
22         tekst="FORELDRE";
23
24     for (i = 0; i < 20; i++) {
25         for (j = 0; j < 9999999; j++)
26             ;
27         printf("%s\n", tekst);
28         fflush(NULL);
29     }
30
31     return 0;
32
33 }

```

Listing 10.7: eksempler/11/wait-eksempel\_I.c

```

1 #include <sys/types.h>
2 #include <unistd.h>
3 #include <stdio.h>
4 #include <wait.h>
5
6 /*
7  Demonstrasjon av kjoering av to prosesser,
8  hvor den ene venter paa den andres terminering.
9
10 */
11
12 int main(void)
13 {
14
15     int forkretur, i, j;
16     char *tekst;
17
18     forkretur=fork();
19
20     if(forkretur==0)
21         tekst="BARN";
22     else{
23         tekst="FORELDRE";
24         (void) wait(NULL);
25     }
26
27     for (i = 0; i < 20; i++){
28         for (j = 0; j < 9999999; j++)
29             ;
30         printf("%s\n", tekst);
31         fflush(NULL);
32     }

```

```
33  
34     return 0;  
35  
36 }
```

## 10.4 Minnehåndtering

### krav til minnebehandling

<http://debbie.hive.no/osyda50/figurer/7.1.png>

#### ”relocation”

- prosessene bør kunne eksistere i varierende posisjon i minnet

<http://debbie.hive.no/osyda50/figurer/7.8.png>

- Logisk adresse
- Relativ adresse
- Base- og ”bounds”-register
- Absolutt adresse

#### protection

prosessen må kun lese og skrive i tildelt område av minnet.

#### deling

minnebehandlings-systemet bør gjøre det mulig for prosesser å dele minne.

#### logisk organisering

- De fleste dataprogram er delt inn i ulike deler avhengig funksjonalitet. Noen deler skal f.eks ikke være skrivbare. Noen skal være kjørbare, etc.

**Segmentering er et eksempel på en slik logisk organisering av minnet tildelt prosesser.**

- En prosess deles i ulike segmenter.
- Segmentene kan ha ulik størrelse
- Segmentene lastes inn i minnet.

**Eks:** pmap \$\$

## fysisk organisering

### minst to nivåer:

- primærminne
- sekundærminne
- minnehierarkiet

## partisjonering av minnet

### statisk partisjonering

<http://debbie.hive.no/osyda50/figurer/7.2.png>

- En partisjon pr. prosess
- Antall partisjoner er fast
- Intern fragmentering

### dynamisk partisjonering

<http://debbie.hive.no/osyda50/figurer/7.4.png>

- Partisjonene opprettes ved behov
- Ekstern fragmentering
- Størrelsen tilpasses prosessenes behov
- "Compactiton"

## Noen bibliotekskall for minnehåndtering

`void *malloc(size_t size);`

- allokterer minne for size antall byte
- minnet blir ikke initialisert!

### returnerer

- peker til allokert område, eller
- NULL hvis minne ikke ble allokert

`void *calloc(size_t nmemb, size_t size);`

- allokterer minne for en array med nmemb antall elementer av størrelse size.
- minnet blir initialisert med nuller.

**returnerer**

- peker til allokert område, eller
- NULL hvis minne ikke ble allokert

**void free(void \*ptr);**

- frigjør minneområde som ptr peker på.
- ptr må være returverdien fra et tidligere oppkall til malloc(),calloc()eller realloc().
- returnerer ingen verdi

**Eksempler**

Listing 10.8: eksempler/11/malloc\_eks.c

```

1 #include <unistd.h>
2 #include <stdlib.h>
3
4 int main() {
5
6     char s1[] = "1234567890";
7     char s2[] = "abc";
8     char nl[] = "\n";
9
10    char *t2, *t1;
11    int i;
12
13    t1 = malloc(10);
14    for (i=0; i<10; i++)
15        t1[i] = s1[i];
16    write(1, t1, 10);
17    free(t1);
18    write(1, nl, 1);
19
20    t2 = malloc(10);
21    for (i=0; i<3; i++)
22        t2[i] = s2[i];
23    write(1, t2, 10);
24    free(t2);
25    write(1, nl, 1);
26
27    return 0;
28
29 }
```

Listing 10.9: eksempler/11/calloc\_eks.c

```

1 #include <unistd.h>
2 #include <stdlib.h>
3
4 int main() {
5
6     char s1[] = "1234567890";
7     char s2[] = "abc";
```

```

8   char nl[] = "\n";
9
10  char *t1, *t2;
11  int i;
12
13  t1 = calloc(10,1);
14  for (i=0; i<10; i++)
15      t1[i] = s1[i];
16  write(1, t1, 10);
17  free(t1);
18  write(1,nl,1);
19
20  t2 = calloc(1,10);
21  for (i=0; i<3; i++)
22      t2[i] = s2[i];
23  write(1, t2, 10);
24  free(t2);
25  write(1,nl,1);
26
27  return 0;
28
29  }

```

## Vanlige feil i forbindelse med frigjøring av minne

- bruk av frigjort minne v.h.a. «dangling pointer»
- frigjøring av samme minneområde flere ganger
- forsøk på frigjøring av minne, v.h.a. en peker som ikke er returverdi fra malloc().
- bruk av minne som er utenfor allokert «chunk»

## unnlate å frigjøre minne

- minnelekasje
- trashing

## buffer overflow

Listing 10.10: eksempler/11/spill.c

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4
5  void gratulerer() {printf("gratulerer\n");}
6  void kondolerer() {printf("kondolerer\n");}
7
8  void spill(int gjetning, int terning){
9
10     if(gjetning == terning)
11         gratulerer();
12     else
13         kondolerer();

```

```

14 }
15
16
17 int main() {
18
19     int terning, gjetning;
20     static void (*peker)(int, int);
21     static char buffer[2];
22     peker=spill;
23
24     // "Saar et tilfeldig froe"
25     srand(time(NULL));
26
27     // Kaster en terning
28     terning = rand()%12+1;
29
30     printf("Terningene_er_kastet_ Gjett_tallet_(1-12)\n");
31     scanf("%s", buffer);
32     printf("Du_skrev_%s,_og_det_ble_lagret_i_%p.\n", buffer, buffer);
33     printf("Pekeren_inneholder_%p,_og_ble_lagret_i_%p.\n", peker, &peker);
34     gjetning = atoi(buffer);
35     peker(gjetning, terning);
36
37     return 0;
38
39 }

```

- Vi kompilerer spillet:

```
gcc spill.c -o spill
```

- Vi gjetter tallet 2:

```
echo 2 | ./spill
```

- Sjansen for å vinne er 1/12.
- Vi oversvømmer bufferet:

```
echo -e "AAAAAAAAAAAA" | ./spill
```

### Hvor kommer tallet 41 fra?

- man ascii
- Vi kan angi ascii-verdien til 'echo':

```
echo -e "\x41"
```

```
echo -e "\x00\x01\x02\x03\x04\x05\x06\x08" | ./spill
```

- Nå kan vi se at vi kan endre innholdet i 'peker' til det vi måtte ønske.
- Vi finner adressen til funksjonen gratulerer:

```
nm spill | grep gratulerer
```

Nå kan vi vinne hver eneste gang.

## 10.5 Oppgaver

### 11.1

- I denne oppgaven skal du lage et program som simulerer terningkast uten å bruke noen randomfunksjon.

a)

Endre programmet `losninger/10/sirkeltabell.c` slik at det skriver tallene 1 til 6 i stedet for alfabetet.

b)

Endre programmet i a) slik at det skriver tallene fra 1 til 6, i en evig løkke.

c)

Endre programmet i b) slik at løkken fortsatt går i evig løkke, men at utskrift av et tall kun kommer når prosessen får mottar et signal. Et tall for hvert signal. Velg selv hvilket signal du ønsker å benytte.

### 11.2

Lag et program som dramatisk endrer minnebruken under kjøring. Inspiser under kjøring med f.eks. `top`.

### 11.3

**Demoner (tjenere) er ofte implementert slik at de leser sine konfigurasjonsfiler ved oppstart og ved mottak av signalet SIGHUP. Lag et program som simulerer dette. Ved å implementere en funksjon som:**

- 1. kalles ved oppstart
- 2. kalles ved mottak av signalet SIGHUP
- 3. åpner en fil med navn `oppgave3.ini`
- 4. leser et heltall fra filen, og lagrer det i en variabel.

### 11.4

a)

Utnytt sårbarheten i eksempler/11/spill.c til å vinne hver gang, slik det ble demonstrert i forelesningen

b)

Utnytt sårbarheten i `spill.c` til å tape hver gang.



# Kapittel 11

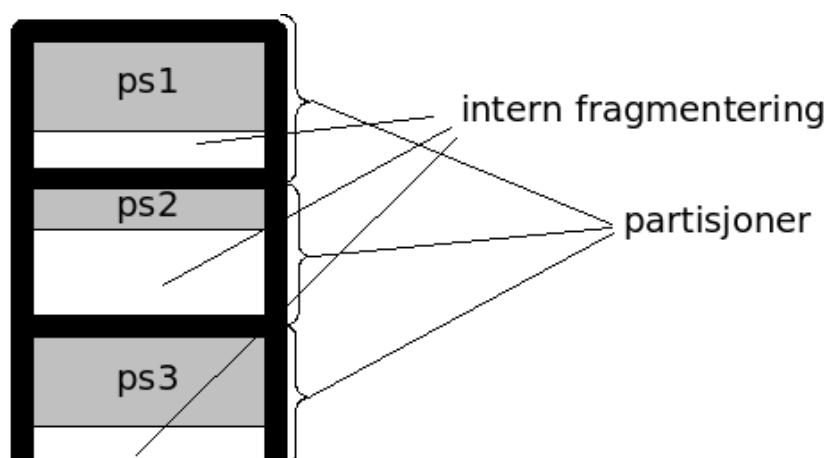
## 12 Sideveksling, virtuelt minne og sikkerhetstrusler

### 11.1 Kort repetisjon fra forrige gang

#### Partisjonering av minnet

##### Statisk partisjonering

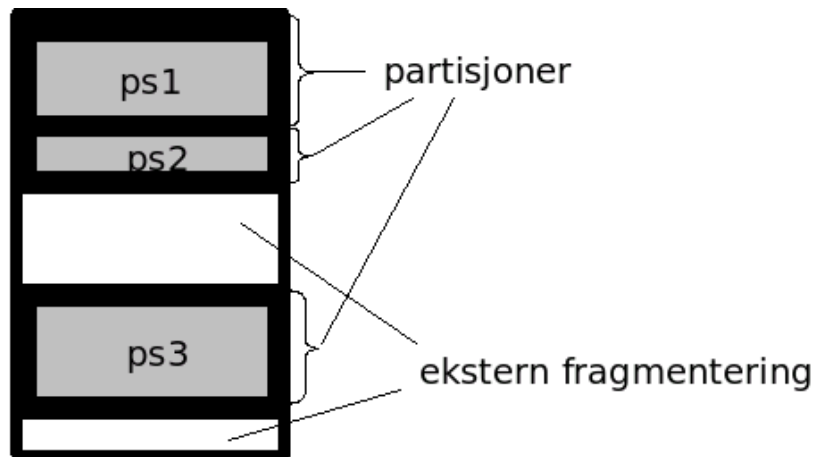
figur



- En partisjon pr. prosess
- Antall partisjoner er fast
- Intern fragmentering

##### Dynamisk partisjonering

figur



- Partisjonene opprettes ved behov
- Størrelsen tilpasses prosessenes behov
- Ekstern fragmentering
- "Compactiton"

## Segmentering

- En prosess deles i ulike segmenter.
- Segmentene kan ha ulik størrelse
- Segmentene lastes inn i minnet.
- Programmerer/kompilator kan kontrollere
- Ulike rettigheter (lese/skrive/kjøre)

### Typiske inndeling i segmenter:

- tekst-segment
- data-segment
- stakk

Eks:

pmap \$\$

## 11.2 Sideveksling (paging)

- minnet delt i rammer
- rammene har samme størrelse
- prosessen deles i sider (pages)
- sidene har samme størrelse som rammene
- Vi får ingen ekstern fragmentering
- Vi får litt intern fragmentering

## Vi ser på figurer

<http://debbie.hive.no/osyda50/figurer/7.9.png>  
<http://debbie.hive.no/osyda50/figurer/7.11.png>  
<http://debbie.hive.no/osyda50/figurer/7.12.png>

## 11.3 Virtuelt minne

### Forutsetninger

#### Hvis

- oversettelse fra logisk til fysisk adresse skjer dynamisk ved "runtime" (av MMU).
- og en prosess kan deles opp i mindre biter

#### så

trenger ikke hele prosessen være i minnet samtidig under kjøring

- Delene i minnet utgjør "resident set"
- "principle of locality"

### Rutine for minnetilgangsfeil (Memory Access Fault):

- referanse til minne utenfor "resident set"
- avbrudd indikerer minnetilgangsfeil
- prosess settes i blokkert tilstand
- OS sender I/O-read for å hente siden som mangler
- OS kjører i gang annen prosess

### Når I/O-operasjonen er fullført

- sendes avbrudd
- og prosess-tilstanden endres til "klar"

### Fordeler

- prosesser kan være større enn primærminnet

### flere prosesser i minnet om gangen

- større sannynlighet for at det finnes prosesser klare til kjøring
- -> høyere grad av prosessor-utnyttelse

## «Thrashing»

- Hvis det fysiske minnet er fullt: For at en side skal kunne hentes inn, må en annen settes ut.
- Det hender at siden som settes ut, snart må hentes inn igjen.
- Hvis dette skjer for ofte - kaller vi det for "thrashing"

Hva kan vi gjøre?

## Virtuelt minne (vm) m/sideveksling

Vi ser på figurer fra boka

<http://debbie.hive.no/osyda50/figurer/8.2.png>  
<http://debbie.hive.no/osyda50/figurer/8.3.png>

## Translation Lookaside Buffer

- Page Table er stor – får ikke plass i registre

En egen cache for sidetabellen:

Translation Lookaside Buffer

Vi ser på figurer fra boka:

<http://debbie.hive.no/osyda50/figurer/8.7.png>  
<http://debbie.hive.no/osyda50/figurer/8.8.png>  
<http://debbie.hive.no/osyda50/figurer/8.9.png>

## VM m/sideveksling og segmentering

Vi ser på figur fra boka

<http://debbie.hive.no/osyda50/figurer/8.12.png>  
<http://debbie.hive.no/osyda50/figurer/8.13.png>

## Opprettelse av veksleområde i Linux

Opprettelse av veksleområde

- Veksleområde kan være en fil eller en partisjon.

Følgende må gjøres for å sette opp et nytt veksleområde:

- Klargjøre området (Eks.: `mkswap /dev/hda4`)
- Aktivere området (Eks.: `swapon /dev/hda4`)
- Deaktivere området (Eks.: `swapoff /dev/hda4`)
- Aktivisering av veksleområder ved oppstart
- For at aktivisering skal skje automatisk ved oppstart, må det legges til en ny linje i filen `/etc/fstab`.

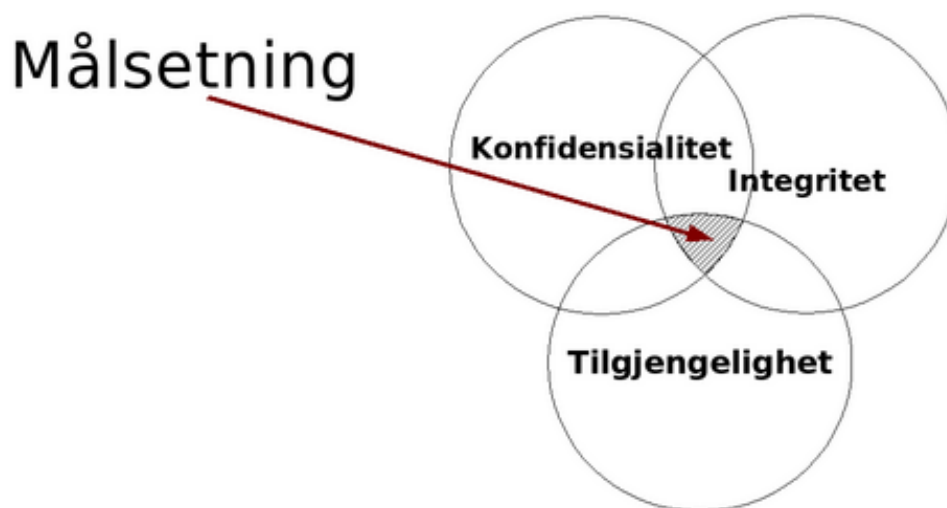
eks. på /etc/fstab

```
# /etc/fstab: filesystem table. #
# filesystem mountpoint type options dump pass
/dev/hda1 / ext3 defaults,errors=remount-ro 0 1
/dev/hda2 none swap sw 0 0
proc /proc proc defaults 0 0
/dev/fd0 /floppy vfat defaults,user,noauto,showexec,umask=022 0 0
/dev/cdrom /cdrom iso9660 defaults,ro,user,noexec,noauto 0 0
/dev/cdrom1 /cdrom1 iso9660 defaults,ro,user,noexec,noauto 0 0
/dev/dvd /dvd iso9660 defaults,ro,user,noexec,noauto 0 0
/dev/cdaudio /cdaudio iso9660 defaults,ro,user,noexec,noauto 0 0
```

## 11.4 Sikkerhetstrusler

### Konsepter

tre hovedmål



- konfidensialitet
- integritet
- tilgjengelighet

### Trusselbildet

#### Trusler

Trusler og angrepsformer <http://debbie.hive.no/osyda50/figurer/t14.1.png>

Sikkerhetssystemets ramme <http://debbie.hive.no/osyda50/figurer/14.2.png>

Trusler og "assets" <http://debbie.hive.no/osyda50/figurer/t14.2.png>

## **Inntrengere**

### **AND80's klassifisering**

- masquerader
- misfeasor
- clandestine user

### **RADC04's klassifisering**

- hacker
- kriminell organisasjon
- intern trussel

## **Ondsinnet kode - (malware)**

### **parasittisk**

- virus
- logisk bombe
- bakdør

### **selvstendig**

- ormer

### **"bots"**

- DDOS (Distributed Denial of Service) - distritibuert tjenestenektangrep
- spam
- sniffing
- keylogging
- spredning av malware
- spredning av adware
- manipulering av avstemminger/spill

### **Trojansk hest**

- Program gjør noe annet enn det det tilsynelatende gjør.

**Eksempel:** <http://cm.bell-labs.com/who/ken/trust.html>

**mobil kode**

- plattform-uavhengighet
- f.eks. makrovirus/skript

**rootkit**

- opprettholde root-tilgang
- endrer systemets funksjonalitet
- godt skjult

<http://debbie.hive.no/osyda50/figurer/14.6.png>

## 11.5 Oppgaver

### 12.1

Lag et program 'minneallokasjon.c' som leser inn et heltall fra standard inn. Programmet skal så allokere det antall bytes som brukeren skrev inn. Tips: malloc(3)

### 12.2

a)

Start en virtuell maskin på debbie med kommandoen: vstart minnetest.

b)

Finn ut hvor mye virtuelt minne den virtuelle maskinen har. Ikke steng den virtuelle maskinen ennå, for du skal bruke den mer. Tips: free(1), top(1).

c)

Du finner din "debbie-hjemmekatalog" i katalogen '/hosthome' på den virtuelle maskinen 'minnetest'. På den virtuelle maskinen 'minnetest' skal du finne og kjøre programmet du lagde i Oppgave 1. Oppgi et antall bytes som er større en det virtuelle minnet du fant i b). Hva skjer?

### 12.3

a)

- I denne oppgaven skal du øke det virtuelle minnet til den virtuelle maskinen 'minnetest' v.h.a.en såkalt 'vekslefil'.

```
dd if=/dev/zero of=/vekslefil bs=1024 count=40
```

```
mkswap /vekslefilswapon /vekslefil
```

e)

Prøv nå å gjenta det du gjorde i Oppgave 2 c) med samme antall bytes.

f)

Gi kommandoen 'halt' i konsollet på den virtuelle maskinen for å ta den ned. Eller 'vhalt minnetest' i konsollet til verskmaskinen (debbie).



# Kapittel 12

## 13 - Sikkerhetstiltak (og litt mer om trusler)

### 12.1 Autentisering

#### Definisjon

- "The process of verifying an identity claimed by or for a system entity"

<http://ietf.org/rfc/rfc2828>

#### To faser

- Identifisering
- Verifisering

#### Generelle metoder

noe du vet

Eksempel: Passord <http://debbie.hive.no/osyda50/figurer/15.1.png>

#### hashing

- DES
- MD5
- Blowfish

#### salt

- like passord fremstår ikke
- "dictionary attack" blir vanskeligere

### Passord i Linux

- På linux brukes krypteringsalgoritmen enveis-DES (eller MD5). At den er enveis, vil si at algoritmen ikke kan dekryptere det den har kryptert.
- Brukerenes passord blir lagret kryptert i en fil (/etc/passwd el. /etc/shadow).
- For å autentisere en bruker, blir det passordet brukeren taster inn, kryptert med samme algoritme, og sammenlignet med det krypterte passordet i filen.
- Det finnes programmer ved å benytte seg av ordlister og kjente krypteringsalgoritmer (og mye prossessortid), som klarer å finne enkle passord. F.eks. Crack og John the Ripper.

<http://manpages.debian.net/cgi-bin/man.cgi?query=crypt&apropos=0&sektion=3&manpath=Debian+5.0+lenny&format=html&locale=en>

### noe du har

- «Token» = berettigelsestegn

### Eksempler på ”tokens”

#### minnekort

- krever leser
- ofte kombinert med passord/PIN

eks.

- minibankkort
- nøkkeltkort

#### smartkort grensesnittelektronisk (krever leser)

- elektriske kontaktpunkter
- trådløs forbindelse
- manuell (taster og display)

algoritme

- statistisk
- dynamisk passordgenerator
- ”challenge-response”

noe du er/gjør

### Eksempler på biometri

- Ansiktsgjenkjenning
- Fingeravtrykk
- håndgeometri
- retina-mønster
- Iris
- Underskrift/signatur
- Stemme

## 12.2 Tilgangskontroll

Tre kategorier av adgangskontroll-policies:

### DAC

<http://debbie.hive.no/osyda50/figurer/15.4.png>

- Discretionary Access Control - brukerbestemt tilgangskontroll
- Tilgang til objekter er basert på subjektets identitet og autoriseringsregler.
- Subjektet kan ha rett til å gi rettigheter videre til andre - derav ordet 'discretionary'.

Eks. filrettigheter i UNIX

fig.: ACL i linux

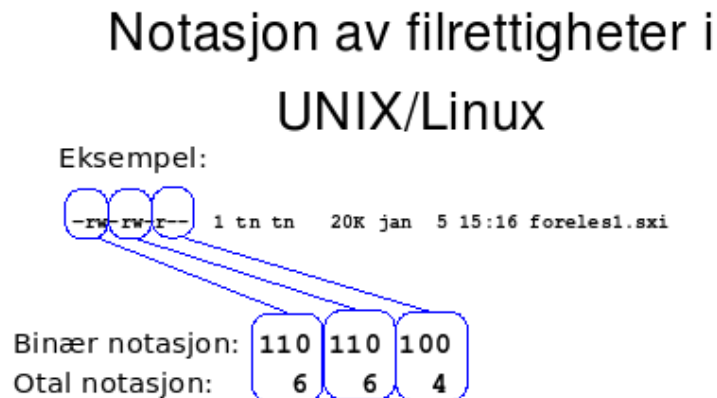
## ACL i Linux

- trad. UNIX tilnærming – minimal ACL

	User	Group	Others
	Eieren	Gruppemedlemmer	Alle Brukere
R	0/1	0/1	0/1
W	0/1	0/1	0/1
X	0/1	0/1	0/1

$\overset{0-7}{rwx} \quad \overset{0-7}{rwx} \quad \overset{0-7}{rwx}$   
 U            G            O

fig.: Notasjon



## MAC

- Mandatory Access Control
- Objekter er klassifisert
- Subjekter er klarert
- Tilgang er basert på subjektets klarereingsnivå og objektets klasse.
- Subjekter kan ikke gi rettigheter videre - derav ordet 'mandatory'
- Reglene settes sentralt.

[http://www.nsa.gov/public\\_info/press\\_room/2001/se-linux.shtml](http://www.nsa.gov/public_info/press_room/2001/se-linux.shtml)

## RBAC

- Role Based Access Control - rollebasert tilgangskontroll.

<http://debbie.hive.no/osyda50/figurer/15.7.png>

## 12.3 IDS - Intrusion Detection System

### ”Host-based IDS”

- Programmer for integritetskontroll beregner tall ut fra innholdet til filer, sjekksommer (en: checksum), slik at tallet blir annerledes dersom innholdet er endret.

- Tallene lagres i en database sammen med filattributter.
- Programmet kjøres periodisk, f.eks. som en "cron"-jobb, og kan dermed varsle endringer i konfigurasjonfiler og viktige systemprogrammer.
- Tips: Det kan være lurt å kjøre programmet fra en medium som er skrivebeskyttet. (f.eks.cdrom)

#### Data kilde

- logger
- systemtilstand

#### Programmer som utfører integritets-kontroll:

- aide
- tripwire
- osiris

#### Network Based

- nettverkstrafikk

<http://www.snort.org/>

#### Analyse metode

- Anomalideteksjon
- Signaturbasert

## 12.4 Antivirus

- Signaturer (må oppdateres)

#### Vanskelig å forhindre virusinfeksjon...

- detektere infeksjon
- identifisere viruset
- fjerne viruset

#### Generic Decryption

- polymorfe virus må dekryptere seg selv for å aktiveres
- Virusene kjøres i en et beskyttet virtuelt miljø - en sandkasse (sandboxing)
- Dette kan være "real time". Det må dermed gå veldig fort.
- Utfordring: Hvor lange skal viruset få kjøre?

**GD-scanner består av:**

- CPU-emulator
- signatur-scanner
- kontrollmodul for emulering

## 12.5 forsvar mot buffer overflyt-angrep

### compile time

#### programmeringsspråk

eks.: java bytecode environment

#### kodeteknikker

<http://openbsd.org/>

#### biblioteker/språktillegg

<http://www.research.avayalabs.com/gcm/usa/en-us/initiatives/all/nsr.htm&Filter=ProjectTitle:Libsafe&Wrapper=LabsProjectDetails&View=LabsProjectDetails>

#### beskyttelse av stakken

eks.: Stackguard (inserts canary)

### runtime

- noexec-bit i MMU for stack/heap

#### randomisert minne-plassering

- stack
- standard bibliotek-rutiner

#### guard pages

- mellom minneområder
- MMU-flag - illegal adresse

## 12.6 Fysiske tiltak

### Plassering

sett f.eks. ikke en maskinen i kjeller som trues av oversvømmelse

## Fysiske beskyttelse

- spenningsvern
- låser, vegger, skap, dører, etc...

### Velkjent sårbarhet i vanlige sylindrelåser

[http://upload.wikimedia.org/wikipedia/commons/6/6e/Pin\\_tumbler\\_with\\_key.svg](http://upload.wikimedia.org/wikipedia/commons/6/6e/Pin_tumbler_with_key.svg)  
[http://upload.wikimedia.org/wikipedia/commons/e/e8/Newtons\\_cradle\\_animation\\_book.gif](http://upload.wikimedia.org/wikipedia/commons/e/e8/Newtons_cradle_animation_book.gif)  
[http://upload.wikimedia.org/wikipedia/en/6/68/Snap\\_gun.jpg](http://upload.wikimedia.org/wikipedia/en/6/68/Snap_gun.jpg)  
<http://www.youtube.com/watch?v=7xkkS2p7SuQ&feature=fvw>

## 12.7 Redundans

- Redundans styrker systemets robusthet:
- Hindrer avbrudd
- Forbedrer tilgjengeligheten

### Eksempler:

- UPS
- Raid
- Speiling
- Sikkerhetskopier

## 12.8 BIOS

- Oppstartspassord – systemet starter ikke opp igjen etter eventuelt strømbrudd.
- Deaktivere oppstart fra diskett/cd og sette passord på BIOS-oppsett-programmet.
- BIOS kan/kunne resettes ved å ta ut batteriet på hovedkortet eller med en bryter på hovedkortet.

## 12.9 Sikkerhet på linux

### Hva er spesielt med Linux?

- In a World Without Walls and Fences, Who Needs Windows and Gates?
- Åpen kildekode; en sikkerhetsrisiko eller sikkerhetsgaranti?

**Hvorfor finnes det (nesten) ikke linux-virus?**

## **Enbruker-modus**

**Enbruker-modus i krisesituasjoner - f. eks. ved brudd på sikkerheten:**

- init 1 (på skallets kommandolinjen)
- linux single (oppstartslasterens kommandolinje)

## **Konsollet**

- Låsing av display/virtuelt konsoll
- Eksempler:
  - xlock/xtrlock - låser X-display. Brukes kun ved bruk av display manager (f.eks. xdm el. kdm)
  - vlock låser alle eller noen av de virtuelle konsollene – ubrukelig fra X.

## **Oppdage fysiske brudd på sikkerhet**

- Er systemet startet om - uten at det var du selv som gjorde det: Bruk f.eks. uptime(1) til å finne opptid..
- kontrollere kabinettet
- kontrollerer logg-filer
- logg-filer kan lagres på en sentral syslog-tjener - Det finnes syslog-varianter som sender loggene kryptert men vanligvis sendes dataene ukryptert.

## **Trusselen fra lokale brukere**

- Det er ofte lettere for en inntrenger å få tilgang til en brukerkonto enn en root-konto.
- Når en inntrenger har en brukerkonto, er det lettere for ham/henne å skaffe seg tilgang til root-kontoen.

**Beskyttelse mot angrep fra lokale brukere.**

### **Opprettelse av nye brukere**

- Gi ikke nye brukere fler rettigheter enn det de trenger.
- Fjern inaktive konti
- Bruk samme brukeridentifikatorer for brukere som har konti på flere maskiner - letter administrasjon og lesing av logger.
- La aldri flere personer dele en konto.



## Brukeren root

- Opptre som root, kun når du trenger det.
- Testkjør kompliserte kommandoer før de faktisk utfører sin oppgave.

### Vær forsiktig med søkestien (PATH) til root, den bør:

- være kort
- aldri inneholde aktiv katalog (.)
- aldri inneholde kataloger som er skrivbare for brukerne.
- Vær forsiktig med hva som legges i /etc/securetty
- Ikke forhast deg, når du opererer som root
- Vær utrolig forsiktig med hvem du gir root-tilgang
- sudo(8) kan brukes for å gi brukere rettigheter til visse kommandoer som root

## Instillinger i /etc/fstab

### Man kan i /etc/fstab sette restriksjoner på monterte enheter/partisjoner

- nosuid - Forhindrer SUID/SGID -programmer å kjøre
- nodev - Forhinder opprettelse av blokk-enhets-filer
- noexec - Forhindrer programmer å kjøre
- Dersom man f.eks. har en egen partisjon for /home, så kan man med fordel sette nodev og nosuid på denne.

### Eksempler:

- Linux Security Howto foreslår nodev og noexec på /var dersom den er på en egen partisjon.
- På fjernmonterte områder (NFS) bør man bruke nodev og nosuid (i noen tilfeller også noexec)

## 12.10 Et tenkt tilfelle: Innbrudd i fire akter (kursorisk pensum)

### 1. akt: Rekognosering

#### Port-scannere

- Tyven forsøker forgjeves å finne åpne dører, ved å portscanne målet.
- Port-scannere er programmer som prøver å koble seg til alle portene på en maskin eller et helt nettverk, og forsøker å bestemme hvilke tjenester som er tilgjengelige. Slike programmer gjør det lettere å finne maskiner som kjører tjenester med kjente svakheter/hull.
- Eksempler på nettverks-scannere:
  - nmap(1) og nessus
  - Nessus
  - Grafisk grensesnitt (GTK)
  - Plugin oppsett for port-scannings tester

<http://www.nessus.org>

#### Mot-tiltak

##### Hold systemet ajour

**Eksempel: APT (brukt i f.eks. i debian og ubuntu)**

- Advanced Packaging Tool

Hjelpemiddel for å:

- laste ned programvare
- håndtere programvare-avhengigheter
- installere/avinstallere
- oppgradere
- holde systemet ajour

`/etc/apt/source.list`

```
# See sources.list(5) for more information, especially
# Remember that you can only use http, ftp or file URIs
# CDRoms are managed through the apt-cdrom tool.
# Security updates for "stable"
deb http://security.debian.org stable/updates main contrib non-free
deb http://security.debian.org testing/updates main contrib non-free
```

```
# Stable
deb http://ftp2.de.debian.org/pub/debian stable main contrib non-free
```

Bruk av APT <http://www.debian.org/doc/manuals/apt-howto/> Eksempel på håndtering av OpenOfficeSøk etter programpakker: `apt-cache search openoffice` Vis info om en programpakke:

- `apt-cache show openoffice.org`
- `apt-cache showpkg openoffice.org`

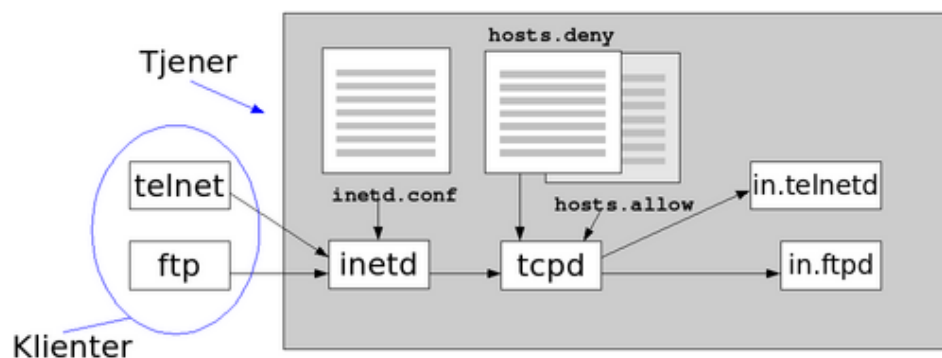
Installasjon av programpakke: `apt-get install openoffice.org` Avinstallasjon av programpakke: `apt-get remove openoffice.org` Oppgradering av alle installerte pakker:

- `apt-get update`
- `apt-get upgrade`

## System-tjenester

- Tyven finner noen åpne porter, og undersøker om programmene som lytter på disse har noen kjente svakheter.
- Jo flere tjenester som kjører (og lytter), jo større sjanse er det for at tyven finner et hull. Kjør derfor ikke flere tjenester på systemet enn det er behov for. Kommandoen under gir en oversikt over tjenester som lytter på porter: `netstat -tap`
- Kommenter ut de tjenestene du ikke vil tilby, i filen `/etc/inetd.conf` og start om `inetd()`.
- Fjern programpakker du vet at du ikke har bruk for.
- Sjekk hvilke tjenester som startes ved oppstart.

## inetd og TCP\_wrappers figur



- Tidligere benyttet de fleste linux-distribusjoner seg av super-demonen inetd() og TCP\_wrappers, tcpd(), for å håndtere forespørsler mot mange av tjenestene.
- Innholdet i konfigurasjonsfilen inetd.conf bestemmer hvilke tjenester super-demonen skal representere, mens filene /etc/host.allow og /etc/hosts.deny hvem som får tilgang. ikke.
- En ny variant extended internet daemon xinetd(), er også i vanlig bruk. Denne kombinerer funksjonaliteten i inetd(), tcpd() og portmapd().

/etc/inted.conf

- navn på tjeneste (/etc/services)
- socket type (stream for TCP, dgram for UDP, raw for IP)
- protokoll (/etc/protocol)
- wait/nowait[.max] (wait for UDP og nowait for TCP)
- bruker[.gruppe] (/etc/passwd)
- stien til tjenerprogram (1. arg til exec())
- argumenter til tjenerprogram (resten av arg. til exec())

Eksempel:

```
#ftp streamtcpnowaitroot /usr/sbin/tcpd /usr/sbin/in.ftpdssh streamtcpnowaitroot /usr,
```

## Brannmurer

- Brannmurer er typisk installert på maskiner som er koblet til to nettverk, f.eks Internettet og det lokale nettverket. En brannmur konfigureres ved å lage regler for hva som sendes fra det ene nettverket til det andre.
- Selv om man har en brannmur på nettverket, er det viktig å sikre maskinene på innsiden.
- Det er også lurt at hver maskin kjører en brannmur, som kontrollerer inngående og utgående trafikk.
- Fra og med linux-kjerne 2.0, kan brannmur-kode kompiles inn i kjernen.
- Ved hjelp av programmene under kan man, i farta, endre reglene.

## Implementasjoner i linux

- ipfwadm (for 2.0 kjerne)
- ipchains (for 2.2 kjerne)
- iptables (fra 2.4 kjerne)

Er det noe vits i å kjøre en brannmur på maskinen dersom alle unødvendige tjenester er skrudd av?

### Å oppdage port-scanning

- Det finnes verktøy for å oppdage port-scannere, men ved bruk av tcp\_wrappers og rutinemessig kontroll av logger, vil det oppdages.
- Det finnes dog såkalte "stealth scanners", som ikke setter spor i loggene.

Hva er vitsen med å oppdage port-scanning?

## 2. akt: Spionering

### Pakke-sniffere

- Tyven har ikke mistet motet, og har tatt seg inn i bygningen hvor målet står. Han har klart å skaffe seg fysisk tilgang på en maskin i et åpent kontorlandskap.
- Han setter ethernet-porten i promiskjøst modus og starter en pakke-sniffer. Pakke-snifferen lytter på ethernetporten etter brukernavn og passord og som lagrer sine funn i en logg.
- Tyven trekker seg tilbake og håper at loggen etter en stund vil inneholde mange loginnavn og passord for konti på flere maskiner på nettverket.
- Tjenester som sender informasjonen ukryptert som sårbare for slik avlytting.
- I mange tilfeller trenger ikke inntrengeren engang bryte seg inn på en maskin i nettet; han kan ta med seg sin egen, og koble den til det lokale nettverket.
- Dersom nettverket er trådløst, kan det holde å ta med pc'en i bilen og parkere utenfor bygningen der det trådløse nettverket er.

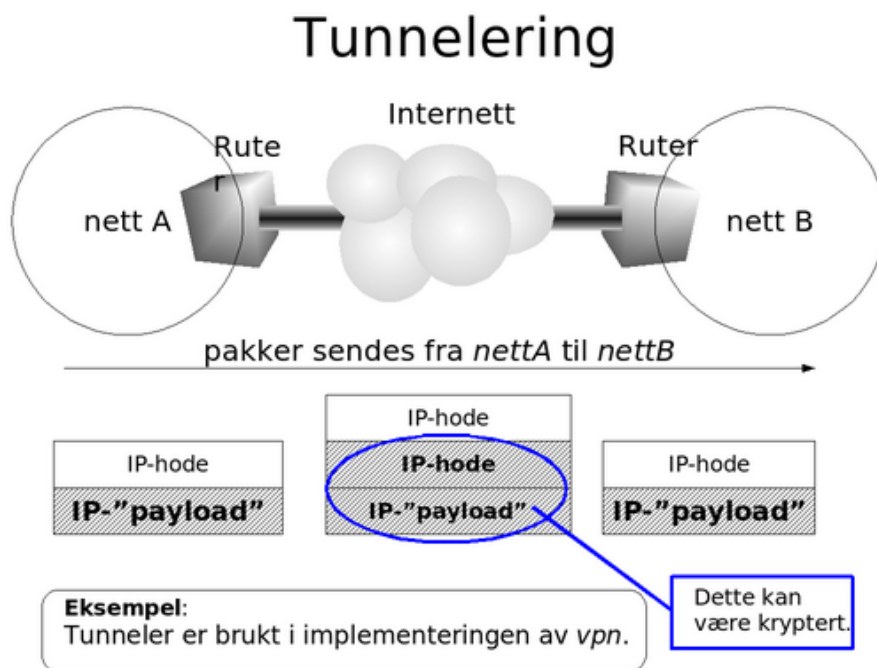
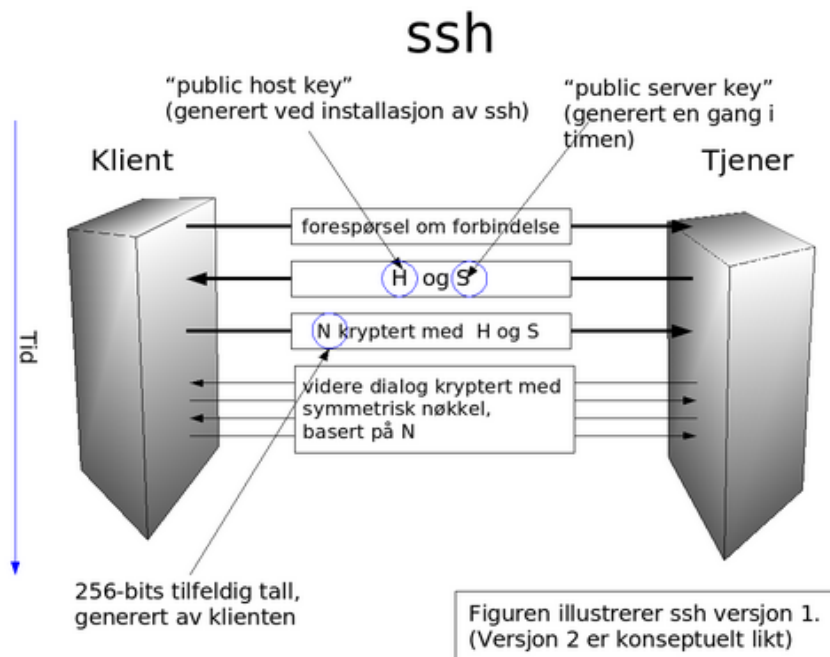
### Eksempel på pakkesniffere:

- wireshark(1)
- tcpdump(8)

### Mottiltak

- Kryptering av internettforbindelser

ssh



### 3. akt Inntrengning og utkastelse

Tyven kommer seg inn og blir kastet ut igjen

- Systemadministratoren oppdager inntrengeren og tar kontakt med den rettmessige eieren av kontoen som er opphavet til "uregelmessighetene", v.h.a. et alternativt medium (f.eks. telefon eller personlig oppmøte).
- Systemadministratoren vurderer: Om hun skal koble systemet fra nettverket stenge for tilgang fra stedet brukeren logget seg inn fra v.h.a. tcp\_wrappers/pakkefilter.

- Hun deaktiverer brukerens konto/endre passord og dreper eller stopper brukerens prosesser.
- Hun følger nøye med en stund etterpå, for å se om angrepet fortsetter via annen konto og/eller fra et annet sted.

### Når innbruddet allerede har hendt

#### Stenge hullet

- Forsøk å avgjøre hvordan innbruddet har skjedd
- stenge hullet

#### Reparere det som er skadet

- Forsøk å avgjøre hvilken skade som har skjedd, og reparere det.
- Ofte er det nødvendig å reinstallere all programvare (f.eks. hvis inntrengeren har fått root-tilgang).

#### Sikkerhetskopier

- Vurder å tilbakeføre sikkerhetskopier for å gjenopprette endrede/skadede filer.
- Kontrollere integriteten til filene på sikkerhetskopien før tilbakeføring.

#### Oppspore inntrengeren

- Ta kontakt med systemadministratoren på det systemet inntrengeren kom fra.
- Forklar hva som skjedde. Utdrag av logger etc.

## 4. og siste akt: Hevnen

### DoS-angrep

- Tyven tyven blir forbannet og bestemmer seg for å ødelegge, ved å gjøre netttjenestene til offeret helt eller tilnærmet ubrukelig. Han vil gjøre det med et såkalt DoS-angrep (Denial of Service) .
- For å gjøre det vanskelig å stanse angrepet, vil han forsøke seg på et distribuert DoS-angrep. D.v.s. at angrepet kommer fra mange maskiner samtidig. Dette realiseres f.eks. med tids-innstilte eller fjernstyrte trojanske hester.

## 12.11 Oppgaver

### Eksempler på kommandoer som kan brukes i oppgave 1 og 2:

#### Opprette en fil med navn og passord for brukeren arne:

```
htpasswd -c passordfil arne
```

**Legge til navn og passord i en eksisterende fil:**

```
htpasswd passordfil beate
```

**Endre passord til navn som allerede eksisterer i fil:**

```
htpasswd passordfil arne
```

**Teste alle passord i en passordfil:**

```
/usr/sbin/john passordfil
```

**Starte virtuell maskin:**

```
vstart virtuellmaskin
```

**13.1**

- a) Opprett en fil med noen krypterte passord v.h.a. programmet `htpasswd(1)`. Lag noen passord du tror er enkle å gjette, og noen vanskelige.

**b)**

- Forsøk så om programmet `john(8)`, klarer å knekke passordene.
- Programmet `john` ligger i katalogen `/usr/sbin`, som normalt bare ligger i `PATH`-variabelen til 'root'. For å kjøre programmet må dere derfor enten legge til `/usr/sbin` i `PATH` variabelen eller oppgi full sti til programmet.

**13.2**

- Start en virtuell maskin på `debbie`. Opprett noen brukere med kommandoen 'adduser'. La brukerne få passord av ulik kompleksitet - noen enkle og noen kompliserte. Du skal nå forsøke å finne passordene til disse brukerne.
- `John` er ikke installert på den virtuelle maskinen, så filen med de passord-hashene (`/etc/shadow`) må kopieres til din hjemmekatalog på `debbie` (`/hosthome`), og kjøre `john` på `debbie`.
- Tips: bruk opsjonen `-format=md5` til programmet `john(8)`.

**13.3****a) Lag et C-program som:**

- leser et passord fra `STDIN`
- Velger salt-verdi
- genererer passord-hash
- lagrer hashen og saltet i en fil
- tips: bruk `crypt(3)`



**b) Lag et C-program som**

- leser passord fra STDIN
- henter salt fra filen i a)
- genererer passord-hash
- sammenligner generert hash
- med hashen lagret i fila

**13.4**

Finn ut hvilke porter som er åpne på matiksi.hive.no og debbie.hive.no. Vær forsiktig med portscanning. Det vil av noen oppfattes som fiendtlig aktivitet. Tips: nmap(1).

**13.5**

- I denne oppgaven skal du bli root på labmaskin på C258 uten å bruke rootpassordet
- a) Start maskinen og velg med piltastene linjen hvor det står "CentOS"
- b) Trykk 'a' for å redigere argumentene til kjernen. Legg til argumentet '1' eller 'single' og trykk enter.
- c) Skriv 'whoami' når du er ferdig med å "boote"
- d) Hva kunne vi gjort for å forhindre at noe slikt kunne skjedd?

**13.6**

Ekstraoppgave: Forsøk å sette opp et software-raid på en virtuell maskin (eller lab-maskin)



Del I

Obligatoriske oppgaver



# Tillegg A

## Obligatorisk oppgave 1 - 2011

### Administrativt

Oppgaven kan løses individuelt eller i grupper (max tre pr. gruppe).

Besvarelsen skal

- leveres i fronter innen utløpet av onsdag 12. oktober 2011.
- fremvises til faglærer for godkjenning og tilbakemelding.

Estimert arbeidsmengde: Mellom 2 og 20 timer (avhengig av forhåndskunnskaper og evner).

### Oppgavetekst

#### Oppgave 1

Du skal i denne oppgaven lage to varianter av et *shell script*, for `bash(1)`. Skriptet skal inneholde en *løkke* som laster ned de seks filene 01.html, 02.html, ... 06.html fra <http://osyda50.hive.no/>. Det skal også sørge for at de seks html-filene slettes umiddelbart *etter at alle nedlastingene er ferdig*. De to variantene er beskrevet i a) og b) under.

- I denne varianten skal nedlastingene skje *etter hverandre*, slik at nedlasting av 02.html *venter med å starte* til 01.html er ferdig nedlastet, o.s.v.
- I denne varianten skal nedlastingene skje *i bakgrunnen*, slik at nedlasting av 02.html *ikke venter med å starte* før 01.html er ferdig nedlastet, o.s.v.

#### Oppgave 2

I denne oppgaven skal du sammenligne skriptene du lagde i forrige oppgave.

- Lag et *shell script*, for `bash(1)`, som beregner gjennomsnittlig omløpstid for de to skriptene fra forrige oppgave. Snittet skal baseres på hundre kjøringar av hvert skript. Til måling av omløpstid skal du bruke `time(1)`. For beregning av gjennomsnitt (addisjon og flyttallsdivisjon) skal du bruke `bc(1)`.
- Kjør skriptet fra a) og kom med et forslag til forklaring på resultatet.

## Tips

### Tips til 1b)

For å få skriptet i 1b) til å vente med sletting av html-filene, til alle er bakgrunnsnedlastingene er ferdige, kan du bruke kommandoen `wait`, som er innebygd i `bash(1)`. Siden `wait` ikke er et selvstendig program, har den ikke egen manualsider. Gi `bash` kommandoen `help wait` for å få opp bruksanvisning.

### Tips til 2a)

- Kommandoen `time(1)` skriver normalt ut til *standard error*. For å redigere dette til *standard utgang*, kan du føye `2>&1` til kommandoen.
- I `bash` finnes en innebygd kommando som heter `time`. Den har ikke så mange opsjoner som `gnu`-varianten som er installert på `debbie`. For at `bash`-skriptet dere lager skal bruke `gnu`-versjonen, må dere skrive stien til dette programmet, i stedet for bare `time`, ellers vil den innebygde varianten brukes. Kommandoen `which(1)` kan brukes for å finne stien til kommandoer.

# Tillegg B

## Obligatorisk oppgave 2 - 2011

### Administrativt

Oppgaven kan løses individuelt eller i grupper (max tre pr. gruppe).

Besvarelsen skal

- leveres i fronter innen utløpet av onsdag 9. november 2011.
- fremvises til faglærer for godkjenning og tilbakemelding.

### Oppgavetekst

#### Oppgave 1

Denne oppgaven handler om få *to prosesser eller tråder* til å skrive sin egen PID *annenhver gang* til en fil. Filen kan godt være STDOUT.

Tips: Finn først ut om du skal bruke prosesser eller tråder. Det kan spare deg for ergrelse.

- a) Programmer en løsning i C. Bruk `pthread(7)` for å synkronisere prosessene/trådene.
- b) Lag en løsning som gjør det samme ved hjelp av et bash-skript. Bruk `flock(1)` for å synkronisere prosessene/trådene.

#### Oppgave 2

I denne oppgaven skal du sammenligne skriptene du lagde i forrige oppgave.

- a) Lag et *shell script*, for `bash(1)`, som beregner gjennomsnittlig omløpstid for de to skriptene fra forrige oppgave. Snittet skal baseres på hundre kjøring av hvert skript.

Til måling av omløpstid skal du bruke `time(1)`. Pass på at du ikke kjører den innebygde `time`-kommandoen i bash. <sup>1</sup> <sup>2</sup>

---

<sup>1</sup>Kommandoen `time(1)` skriver normalt ut til *standard error*. For å redigere dette til *standard utgang*, kan du føye `2>&1` til kommandoen.

<sup>2</sup>I bash finnes en innebygd kommando som heter `time`. Den har ikke så mange opsjoner som `gnu`-varianten som er installert på `debbie`. For at bash-skriptet dere lager skal bruke `gnu`-versjonen, må dere skrive stien til dette programmet, i stedet for bare `time`, ellers vil den innebygde varianten brukes. Kommandoen `which(1)` kan brukes for å finne stien til kommandoer.

For beregning av gjennomsnitt (addisjon og flyttallsdivisjon) skal du denne gangen bruke matlab-klonen `octave(1)`, og ikke `bc(1)` som i forrige oppgave.

- b) Kjør skriptet fra a). Sammenlign gjennomsnittstidene og skriv et forslag til forklaring på resultatet.



Del II

**Eksamensoppgaver**



# Tillegg C

## Eksamensoppgaver høst 2008 med løsningsforslag

1.

Forklar hvordan DMA (Direct Memory Access) bidrar til å øke graden av utnyttelse i et datasystem.

**Svar:**

Når I/O-enheter kan lese og skrive til minne uten å involvere CPU'en, vil CPU'en kunne jobbe parallelt med I/O-enhetene.

2.

a)

Hva ligger i begrepet kjøretidsdeling (Time sharing)?

**Svar:**

Flere brukere (eller prosesser/tråder) kan bruke datasystemet samtidig – kjøretiden fordeles mellom brukerne (prosessene/trådene).

b)

Hvordan realiseres det?

**Svar:**

Jobb (el. prosess/tråd) blir avbrutt ved faste tidsintervaller og lagt i kø. Ny jobb velges ut av køen. Intervallet (og køen) er, idéelt sett, så kort at brukerne ikke oppfatter denne vekslingen.

c)

Beskriv bakgrunnen for at denne teknikken ble utviklet.

**Svar:**

P.g.a. høye maskinvare-kostnader, måtte mange dele samme datasystem. Brukerne ønsket å interagere direkte med maskinen (ikke via operatør).

**3.**

a)

Hva menes med et 'shell' i UNIX/Linux?

**Svar:**

Brukergransesnitt mot datasystemet. Vanligvis kommandolinjebaserte (Command Line Interface - CLI). Program som leser, tolker og utfører kommandoer.

b)

Nevn et slikt 'shell'.

**Svar:**

bash (Bourne Again Shell)

c)

Hva menes med et 'shell script'?

**Svar:**

En fil som er ment å leses av et shell Hver linje i fila leses, tolkes og utføres av shellet.

d)

Skriv et 'shell script' som - lager en katalog - kopierer en fil fra foreldrekatalogen (parent directory) inn i den nyopprettede katalogen

Du kan selv bestemme hva filen og katalogen skal hete.

**Svar:**

```
1 #/bin/sh
2 mkdir katalog
3 cp ../fil katalog/
```

e)

Hva må til for å kjøre skriptet i d)?

**Svar:**

Jeg antar at filen heter d.sh.

```
bash d.sh
eller
bash < d.sh
eller
chmod +x d.sh
./d.s
```

**4.****a)**

Hva er en 'prosess'?

En enhet av aktivitet i et datasystem karakterisert av

- en eller flere sekvensielle tråder av utførelse
- en tilstand
- tilhørende system resursser

**b)**

Hva er en 'tråd'?

Sekvensiell utførelse av instruksjoner med tilhørende minne og tilstand i tillegg til minnet og tilstanden den deler med resten av prosessen den tilhører.

**c)**

Skriv et C-program som starter en annen prosess. Den nye prosessen skal kjøre et annet program. Du kan selv bestemme hvilket program som skal startes.

**Svar:**

```
1 if (fork() == 0)
2     execlp( "ls", "ls", NULL );
```

**5.****a)**

Hva er Symmetrisk Multiprocessing? Tegn og forklar.

I et SMP-system finnes flere prosessorer som deler minne, buss og I/O-enheter. Prosesser/tråder kan kjøres på en hvilken som helst av prosessorene. Hver prosessor selv-skedulerer.

**b)**

Hva er et Cluster? Tegn og forklar.

Selvstendige datasystemer (noder) som fungerer som et datasystem. Hver node har sitt eget sett med CPU'er, minne, busser og I/O-enheter. Nodene kommuniserer f.eks. ved hjelp av standard nettverkskomponenter.

**c)**

Sammenlign systemene i a) og b).

**Svar:**

Cluster skalerer veldig bra i motsetning til SMP. SMP er mer moden teknologi.

SMP: raskere kommunikasjon mellom prosesser/tråder, enn på ulike noder i et cluster, siden kan dele minne (og buss).

**6.**

Hva menes med 'concurrency'?

**Svar:**

Prosesser/tråder veksler på å kjøre eller kjører helt/delvis parallelt, innenfor et gitt tidsintervall. Det er uforutsigbart når et bytte skjer, slik at selv om rekkefølgen av instruksjonene i de konkurrerende prosessene hver for seg er forutsigbar, vil rekkefølgen på instruksjonene til alle prosessene *sett som ett system* ikke være forutsigbar.

**7.**

For å kunne håndtere 'concurrency', må vi kunne håndheve gjensidig utelukkelse (mutual exclusion). Innføring av gjensidig utelukkelse medfører to nye problemstillinger.

**a)**

Nevn disse problemstillingene.

**Svar:**

- Vranglås (deadlock)
- Utsulting (Starvation)

**b)**

Beskriv en av disse problemstillingene.

**Svar:**

Ved utsulting vil noen prosesser bytte på å bruke en resurs, på en slik måte at en eller flere prosesser som trenger tilgang til ressursen, aldri får tilgang. At en prosess aldri får tilgang kalles utsulting.

c)

Gi et eksempel på problemstillingen du valgte i b).

**Svar:**

Prosess 01 og prosess 02 bytter på å ha eksklusiv tilgang til ressurs R. En prosess 003 har lavere prioritet, trenger også tilgang. Den er blokkert mens den venter på tilgang, men vil aldri få tilgang fordi den har lavere prioritet i fordelingsmekanismen (scheduler).

8.

a)

Beskriv produsent/konsument-problemet.

**Svar:**

Dette er et synkroniseringsproblem som oppstår når en produsent og en konsument ikke produserer og konsumerer i samme takt.

b)

Skisser en løsning på problemet i a)

**Svar:**

En løsning på problemet er å bruke en kø, slik at produsenten blokkeres når køen er full, og konsumenten blokkeres når køen er tom.

9.

a)

Hva menes med sideveksling (paging)?

**Svar:**

Minnet deles rammer på lik størrelse. Prosessenes minneområdet deles i sider av samme størrelse. Operativsystemet kan flytte (veksle) sidene mellom primærminnet og sekundærminnet etter behov.

b)

Hva menes med virtuelt minne?

**Svar:**

Et adresserbart minneormåde som (vanligvis) er større enn det fysiske minnet.

Implementert med en mekanisme som dynamisk frigjør plass i primærminnet ved å flytte data til sekundærminnet ved behov. Mekanismen henter også dataene tilbake til primær-minnet når de blir adressert.

c)

Forklar hvordan sideveksling brukes i forbindelse virtuelt minne.

**Svar:**

Prosessene er delt i sider. Alle sidene til en prosess trenger ikke være i minnet samtidig. Dette gjør at vi kan ha flere prosesser i minnet samtidig som er klare til å kjøre (og utnytte maskinen bedre). Prosessene kan også være større enn det fysiske minnet. Når sideveksling brukes i forbindelse med virtuelt minne, vil systemet flytte en side av gangen mellom primærminnet og sekundærminnet. Når en minnecelle adresseres med en virtuell adresse som befinner seg i en side som ikke er i primærminnet, skjer en såkalt "page fault". Den adresserende prosessen blokkeres mens siden bringes inn til primærminnet.

## 10.

a)

Hva menes med autentisering?

**Svar:**

Kan beskrives som en to-trinns-prosess:

- Identifisering: Brukeren hevder en identitet.
- Verifisering: Systemet utfører en kontrollrutine for å avgjøre om brukeren kan anses å tilsvare oppgitt identitet.

b)

Beskriv hvordan autentisering tradisjonelt gjøres i UNIX/Linux.

**Svar:**

Brukernavn og passord-hash produseres og lagres i en tabularisk fil (/etc/shadow el./etc/passwd), sammen med en tilfeldig valgt verdi kalt 'salt' som brukes i hashfunksjonen. Filen inneholder altså linjer med (brukernavn, salt, hashverdi)



Ved autentisering identifiserer brukeren seg med et brukernavn. Systemet søker opp brukernavnet i fila, og henter ut tilhørende salt og hash (dersom brukernavnet finnes i fila).

For å verifisere identiteten oppgir brukeren et passord. Dette passordet brukes sammen med saltet fra fila for å produsere en ny hash-verdi. Denne hash-verdien vil være identisk med den som ble hentet fra fila, dersom passordet er korrekt.

**Lykke til :)**



# Tillegg D

## Eksamensoppgaver høst 2009 med løsningsforslag

### 1.(10%)

a)

Beskriv kort hva som menes med *avbrudd (interrupt)* i forbindelse med datasystemer.

**Svar**

Prosesor kan avbrytes med elektrisk signal slik at neste instruksjon som utføres fra en forhåndsbestemt minneadresse knyttet til signalet.

b)

Forklar kort hvordan avbrudd kan bidra til å øke graden av utnyttelse i et datasystem.

**Svar**

Siden I/O-enhetene selv kan "si i fra" v.h.a. avbrudd-mekanismen når I/O-operasjonen er ferdig, kan maskinen jobbe med noe annet istedenfor å ikke gjøre noe mens den venter på I/O-enheten. Mindre dødtid gir mer arbeid pr. tidsenhet og dermed høyere utnyttelsesgrad.

**Svar**

### 2. (15%)

a)

Beskriv kort hva som menes med kjøretidsdeling (time sharing)?

**Svar**

Systemet deler tiden inn i faste tidsintervaller, og ruller på kjøring av av brukere/prosesser/jobber/tråder

**b)**

Hvordan skapes kjøretidsdeling?

**Svar**

En klokke sender et avbrudd-signal Ved utløp av hvert tidsintervall. Den kjørende bruker/jobb/prosess/tråd blir blokkert og lagt i kø. En ny bruker/jobb/prosess/tråd blir valgt til å kjøre på prosessoren.

**c)**

Nevn minst en gevinst/fordel og en kostnad/ulempe ved å innføre kjøretidsdeling i et system.

**Svar**

Fordel: Lavere responstid.

Ulempe: Lavere utnyttelsesgrad i forhold til batch-systemer med multiprocessing (p.g.a. mer tid til vekslng mellom brukere/jobber/prosesser/tråder).

### **3. (15%)**

**a)**

Beskriv kort hva som menes med kappløp (race condition)?

**Svar**

To eller flere tråder/prosesser leser/skriver til samme resurss i løpet av samme tidsrom (konkurrerende/samtidig), på en slik måte at utfallet av kjøringen blir uforutsigbar.

**b)**

Hvordan kan en slik situasjon oppstå.

**Svar**

Dette kan oppstå når vekslngen mellom kjøring av trådene/prosessene er avhengig av hendelser som ikke er forutsigbare.

**c)**

Gi et eksempel og skisser en prinsipiell løsning.

## Svar

I et system med kjøretidsdeling har trådene A og B begge tilgang variabelen C. Både A og B skal utføre:  $C = \text{compute}(C)$ .

Verdien av C vil dermed avhenge om når vekslingen skjer. Dersom en av prosessene blir avbrutt av klokka etter at *compute()* har lest verdien av C, men før beregningene ferdige. Dette er f.eks. avhengig av hvor mange andre prosesser/tråder som til en hver tid kjører på systemet.

Dette kan håndteres med mekanismer for eksklusiv tilgang (gjensidig utelukkelse) til den delte ressursen, som i eksemplet her er representert med variabelen C, slik at trådene kan fullføre operasjoner på delte ressurser uten at andre tråder endrer ressursenes tilstand.

## 4. (10%)

Under ser du innholdet i en fil.

```
1 #!/bin/sh
2 echo a > b
3 mkdir c
4 cd c
5 cp ../b .
```

### a)

Hva slags fil er dette? Gi en kort begrunnelse.

## Svar

Dette er et shell-skript fordi den første som gir stien till shellen og resten er kommandoer som kan gis til et shell.

### b)

Forklar kort hensikten med hver enkelt linje i fila.

## Svar

1. Stien til programmet som skal tolke linjene under.
2. Åpne filen b og skriv tegnet a i filen (evt. gammelt filinnhold forsvinner).
3. Lag katalogen c som en underkatalog til arbeidskatalogen.
4. Endre arbeidskatalogen til den nyopprettede katalogen.
5. Kopier filen b fra foreldrekatalogen til arbeidskatalogen.

**5. (20%)**

Et dataprogram inneholder bl.a. følgende kode:

```

1 #include <stdlib.h>
2 #define N 4
3
4 void compute(int a){
5
6     // gjoer noen beregninger her
7 }
8
9 void f1 () {
10
11     int i;
12
13     for (i=0; i<N; i++){
14
15         if (fork()==0 ){
16             compute(i);
17             exit(0); // terminerer prosessen
18         }
19     }
20
21     for (i=0; i<N; i++)
22         wait(NULL);
23 }
24
25 void f2 () {
26
27     int i;
28     for (i=0; i<N; i++){
29
30         if (fork()==0){
31
32             compute(i);
33             exit(0); // terminerer prosessen
34         }
35
36         wait(NULL);
37     }
38 }
39
40 int main() {
41
42     f1 ();
43     f2 ();
44     return 0;
45 }
```

Anta at koden kjører på en system med *symmetrisk multiprosessering* (SMP). Operativsystemet har 4 prosessorer tilgjengelige.

**a)**

Gi en kort overordnet forklaring på hva koden gjør.

**Svar**

Mainfunksjonen starter to funksjoner  $f1()$  og  $f2()$ . Begge disse starter fire nye prosesser. Hver av de nye prosessene kjører funksjonen  $compute()$ . Deretter terminerer de.

**f1()**

Funksjonen  $f1()$  starter fire prosesser. Når alle fire er startet, venter den til alle er terminert før den returnerer.

**f2()**

Funksjonen  $f2()$  starter fire prosesser. Etter hver oppstart av en prosess, venter til den nyoppstartede prosessen er terminert før neste prosess startes.

**b)**

Vil kjøretiden til  $f1()$  bli kortere, lengre eller like lang dersom koden hadde kjørt på et system med kun en enkel prosessor. Gi en kort begrunnelse for svaret.

**Svar**

Kjøretiden blir sannsynligvis blir lengre fordi de fire prosessene ikke lenger har anledning til å kjøre parallellt.

**c)**

Vil kjøretiden til  $f2()$  bli kortere, lengre eller like lang dersom koden hadde kjørt på et system med kun en enkel prosessor. Gi en kort begrunnelse for svaret.

**Svar**

Prosessene som startes i  $f2$  vil aldri kjøres parallell, siden den ny prosess ikke startes før den pågående er fullført. Antall prosessorer vil dermed ikke påvirke funksjonens kjøretid. D.v.s. at kjøretiden blir tilnærmet lik.

**d)**

Vil kjøretiden til  $main()$  bli kortere, lengre eller like lang dersom koden hadde kjørt på et system med 8 prosessorer. Begrunn svaret.

**Svar**

Siden  $f2()$  starter etter at  $f1()$  og dens barneprosesser, er fullført, og den opprinnelige prosessen ikke gjør stort annet enn å vente, vil det kun være fire prosesser som kan jobbe parallellt selv om antall prosessorer skulle øke. Det vil derfor ikke gi noen kortere kjøretid selv om antall prosessorer øker.

**6. (20%)**

a)

Hva menes med *vranglås* (dead lock)?

**Svar**

Med *vranglås* i et datasystem menes at to eller flere prosesser/tråder/jobber aldri kommer videre i sin programkjøring på grunn av at de venter på hverandre i det som kalles *sirkulær venting*.

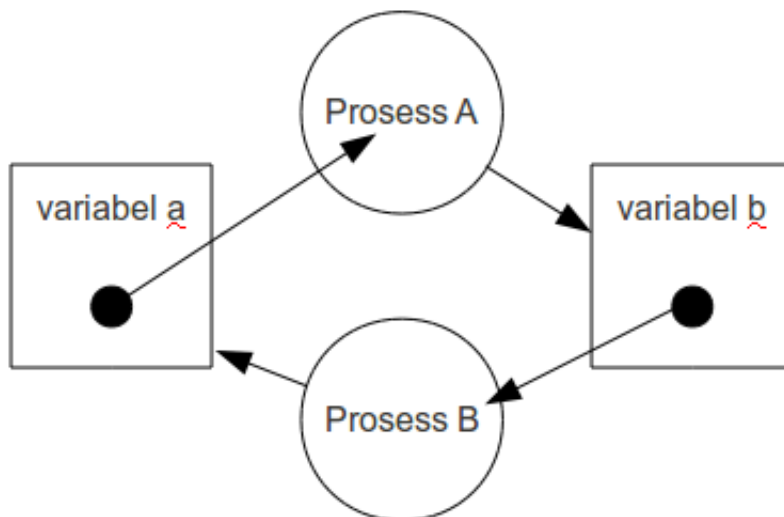
b)

Gi et praktisk eksempel på en *vranglås* i et datasystem.

Prosess *A* har en lås på variablene *a* og venter på tilgang på variabelen *b*. Prosess *B* har en lås på variablene *b* og venter på tilgang på variabelen *a*.

c)

Beskriv eksemplet du ga i b) som en *resurss-allokerings-graf* (Resource Allocation Graph)

**Svar**

d)

Foreslå kort hvordan hvordan *vranglåsen* i b) kunne vært unngått.



**Svar**

*A* og *B* kunne endres slik at de sørget for tilgang til begge variablene i samme rekkefølge, før de begynte å jobbe med dem.

**7. (10%)****a)**

Gi en kort beskrivelse av og et enkelt eksempel på av hva produsent/konsument-problemet går ut på.

**Svar**

Problemet går ut på å få til synkronisering i en situasjon hvor produksjon ikke skjer i samme takt som konsum.

**b)**

Beskriv en måte dette løses på i praksis. Bruk gjerne eksempel i C-kode eller shell-skript.

**Svar**

Dette kan løses ved at det som produserers plasseres i en kø, og det som skal konsumeres hentes ut av køen. Når køen er tom, må konsumenten vente. Når køen er full må produsenten vente. I kommandolinjen `cat /etc/passwd | grep Hansen` vil *cat* være produsenten og *grep* være konsumenten. Produsentens utskrift sendes inn i en *pipe*, som er implementert ved hjelp av en kø. Konsumenten leser fra køen.



# Tillegg E

## Eksamensoppgaver høst 2010 *uten* løsningsforslag

### 1 (25%)

a)

Skisser *hovedelementene* i en datamaskin.

b)

Skisser *rutinen* en prosessor gjentar så lenge den har strøm.

c)

Forklar kort hvorfor operativsystemer med *kjøretidsdeling* (*time sharing*) ble utviklet.

d)

Skisser et *minne-hierarki*.

e)

*Hurtigminne* (*cache*) baserer seg på tendensen til *locality of reference*. Forklar kort hva denne tendensen går ut på.

### 2.(10%)

a)

Vis hvordan vi kan redirigere *standard ut* til en fil ved hjelp av en kommandolinje i et shell.

b)

Vis hvordan vi kan redirigere *standard ut* til fil ved hjelp av C-kode.

**3 (15%)**

Under ser du utskrift av en fil.

```

1 #!/bin/sh
2
3 echo 'int main() {fork(); write(2,"AUAU",3); return 0;}' > b.c
4 gcc b.c
5 ./a.out

```

**a)**

Hva slags fil er dette? Gi en kort begrunnelse.

**b)**

Forklar kort hensikten med hver enkelt linje i fila.

**c)**

Hva blir utskriften når fila kjøres?

**4.(25%)**

I et operativsystem kan prosesser og tråder kommunisere med hverandre.

**a)**

Vis et eksempel på en shell-kommandolinje eller C-kommando som får en prosess til å sende et *signal* til en annen prosess.

**b)**

Vis et eksempel på en shell-kommandolinje eller C-kommando som får en prosess til å sende data til en annen prosess ved hjelp av *meldingssending* (*message passing*) gjennom en datastruktur av typen *kø* (first in first out).

**c)**

Tråder og prosesser kan kommunisere ved bruk av felles minne. Ved bruk av felles minne kan vi få en bestemt situasjon som krever at at systemet håndhever *gjensidig utelukkelse* (*mutual exclusion*). Hva kalles den denne situasjonen?

**d)**

Identifiser kritisk(e) område(r) i koden under:

```

1  #include <pthread.h>
2
3  int compute(int a);
4  void *f();
5  int a;
6
7  int main() {
8
9      pthread_t t2,t1;
10
11     pthread_create(&t1, NULL, f, NULL );
12     pthread_create(&t2, NULL, f, NULL );
13
14     pthread_join(t1, NULL);
15     pthread_join(t2, NULL);
16
17     return 0;
18
19 }
20
21
22 void *f() {
23
24     while(1)
25         a = compute(a) + compute(a);
26
27 }
28
29 int compute(int a){
30
31     /* Her gjoeres noen tunge
32        beregninger paa lokal variabel.
33        Resultatet av beregningene returneres */
34
35 }

```

e)

Kom med et forslag til hva vi bør gjøre med de(t) kritiske område(r) du fant i d).

## 5.(10%)

I forbindelse med *virtuelt minne* har vi noe som kalles *minnetilgangsfeil* (*memory access fault*).

a)

Når frekvensen av minnetilgangsfeil blir for høy, oppstår en uønsket tilstand som har fått et eget navn (på engelsk). Hva kalles denne tilstanden? Tips: Det er ofte mulig å høre den.

**b)**

Forklar kort hvilken hvilken rolle *avbrudd (interrupt)* spiller i håndtering av minnetilgangsfeil.

**6.(15%)**

**a)**

Forklar kort hva som ligger i de tre sikkerhets-aspektene: *Integritet, konfidensialitet og tilgjengelighet*.

**b)**

Forklar kort hvordan *hashing* brukes i den tradisjonelle *autentiserings-mekanismen* på Linux-systemer.

**c)**

Når vi har pekerfeil i C-programmer, hender det at operativsystemet terminerer prosessen som kjører programmet med pekerfeilen. Forklar hvorfor operativsystemet gjør dette.

# Tillegg F

## Eksamensoppgaver høst 2011 *uten* løsningsforslag

### 1. Skallet (25%)

- a) Forklar kort hva som menes med *kommandolinje-basert grensesnitt (Command Line Interface - CLI)*?
- b) Forklar kort hva som menes med et et *UNIX shell*?
- c) Skriv et *shell script* som oppretter en katalog som heter “kat” og skriver “hallo verden” til en fil i den nye katalogen. Den nye filen skal hete “fil”. Forklar også hva som må gjøres for at fila skal kunne kjøres med kommandoen *./fil*.
- d) Tegn og forklar hvordan *fildeskriptortabellen* manipuleres ved *redirigering av standard ut (STDOUT) til en fil* (i UNIX/Linux).
- e) Under ser du innholdet av en fil. Hva blir utskriften når fila kjøres?

```
1 #!/bin/sh
2 echo A > B
3 mkdir C
4 cat B > C/D
5 cp B C/E
6 cat C/*
```

### 2 Avbrudd (25%)

- a) Beskriv kort hva som menes med *avbrudd (interrupt)* i forbindelse med datasystemer.
- b) Beskriv kort *'fetch-execute'-syklusen* i et system med avbrudd.
- c) Forklar kort hvordan avbrudd kan bidra til å øke graden av utnyttelse i et *flerprogramkjørende system for satsvis behandling (multiprogrammed batch system)*.
- d) Forklar kort hvordan avbrudd brukes i forbindelse med *kjøretidsdeling (time sharing)*.
- e) Forklar kort hvordan avbrudd brukes i forbindelse med *direkte minnetilgang (Direct Memory Access - DMA)*.

### 3. Synkronisering(20%)

I noen operativsystem trenger vi mekanismer for *gjensidig utelukkelse*.

- a) Nevn eller beskriv en type operativsystem som trenger en slik mekanisme. Gi en kort begrunnelse.
- b) Nevn eller beskriv en type operativsystem som *ikke* trenger en slik mekanisme. Gi en kort begrunnelse.
- c) Den abstrakte datatypen *semafor* kan brukes til gjensidig utelukkelse. Den består av et heltall og tre operasjoner. Beskriv de tre operasjonene.
- d) Vis med pseudokode (eller C-kode) hvordan du ville beskyttet et kritisk område v.h.a. semaforer.

### 4. Virtuelt minne (15%)

Tenk deg at du har fått i oppdrag å utvide det virtuelle minnet i en datamaskin som kjører linux. Du har fått et satt et nytt platelager (disk) i maskinen. Grensesnittet mot platelageret er representert i filsystemet med stien `/dev/sdb`. Du har inndelt platelageret i fire partisjoner.

- a) Forklar kort som menes med *virtuelt minne*.
- b) Tegn og forklar kort som menes med hvordan *sideveksling (paging)* virker.
- c) Vis med kommandoer og kommentarer hvordan du ville gått frem for å utvide det virtuelle minnet.

### 5. Sikkerhet (15%)

- a) Forklar kort hva som menes med *buffer-overflyt (buffer overflow)*.
- b) Forklar kort hvorfor vi ønsker å forhindre muligheten for buffer-overflyt.
- c) Gi et eksempel på et mottiltak som forhindrer buffer-overflyt. Forklar også kort hvordan mottiltaket virker.

**Lykke til :)**



# Tillegg G

## Oppgavesett kontinuasjonseksamen 2011 *uten* løsningsforslag

### 1. Skallet (25%)

- a) Hva kalles et program som gir et kommandolinje-basert grensesnitt til operativsystemkjernen i UNIX/Linux? Gi også et eksempel på et program som gir et kommandolinje-basert grensesnitt til operativsystemkjernen i UNIX/Linux?
- b) Skriv et *skall-skript* (*shell script*) som tar en sikkerhets-kopi av alle filene i katalogen "kat".
- c) Hva er kriteriene for at filen "a" skal kunne kjøres med kommandoen *./a* på kommandolinjen?
- d) Hva er kriteriene for at filen "a" skal kunne kjøres med kommandoen *a* på kommandolinjen?
- e) Når følgende kommando gis: "cat a | sort", startes to prosesser som skal kommunisere via et anonymt rør ("pipe"). Tegn og forklar hvordan *fildeskriptortabellene* til de to prosessene manipuleres for å klargjøre for kommunikasjonen.

### 2 Grunnleggende om operativsystemer (25%)

- a) Gi en overordnet beskrivelse av de grunnleggende elementene i en datamaskin.
- b) Beskriv kort hvordan *'fetch-execute'-syklusen* i en datamaskin virker.
- c) Gjør kort rede for hvordan et operativsystem kan utnytte *avbrudd(interrupt)-mekanismene* i en datamaskin.
- d) Nevn en operativsystem-type som er har som mål å gi *god prosessorutnyttelse*. Gi også en kort begrunnelse.
- e) Nevn en operativsystem-type som er har som mål å gi *lav responstid*. Gi også en kort begrunnelse.

### 3. Samtidighet (20%)

- a) Beskriv kort forskjellene mellom tråder og prosesser.
- b) Forklar kort hvordan vi kan ha *samtidighet (concurrency)* i systemer med en enkel prosessor.
- c) Skriv C-kode for et POSIX/UNIX/Linux-system som starter en *ny* prosess som kjører programmet 'ls' med argumentet '-l'.
- d) Den abstrakte datatypen *semafor* kan brukes til *gjensidig utelukkelse*. Vis med pseudokode (eller C-kode) hvordan du ville beskyttet et kritisk område v.h.a. semaforer.

### 4. Minnehierarkiet (15%)

- a) Gi en overordnet beskrivelse av *minnehierarkivet* i en datamaskin.
- b) Forklar kort hvordan et operativsystem kan utnytte minnehierarkiet til å implementere virtuelt minne.
- c) Ved bruk av hurtigminne (cache) kan gjennomsnittstiden for minnetilgang (memory access) reduseres. Gevinsten kan oppnåes dersom lokalitets-prinsippet (principle of locality) gjør seg gjeldene. Forklar kort hva dette prinsippet betyr i denne forbindelse.

### 5. Sikkerhet (15%)

- a) Nevn en mekanisme i UNIX/Linux-systemer som sørger for fil-konfidensialitet. Oppgi også en kort forklaring på hvordan mekanismen gir mulighet for konfidensielle filer.
- b) Forklar kort hvordan *hashing* brukes ved autentisering UNIX/Linux.
- c) Alle prosesser i et UNIX/Linux har en *aktiv arbeidskatalog (current working directory)*. Et *punktum* i en filsti (path) representerer den, til enhver tid, aktive arbeidskatalogen. Et sikkerhets-råd sier at miljøvariabelen (environment variable) *PATH*, aldri skal inneholde et punktum. Dette rådet har sammenheng med en svakhet i tilgangskontrollmodellen *discretionary access control (DAC)*.  
Gjør kort rede for denne sammenhengen.

**Lykke til :)**

# Tillegg H

## Oppgavesett siste gangs eksamen 2012 *uten løsningsforslag*

### 1. Diverse (30%)

- a) Beskriv kort forskjellene mellom tråder og prosesser.
- b) Skriv C-kode for et POSIX/UNIX/Linux-system som starter en *ny* prosess som kjører programmet 'ps' med argumentet '-aux'.
- c) Skriv et *skall-skript* (*shell script*) som kopierer av alle filene i katalogen "kat1" til katalogen "kat2/kat3". Skriptet skal utføres av "bash" som i dette tilfellet befinner seg (mot normalt) i katalogen "/usr/local/bin"
- d) Forklar kort hvordan vi kan ha *samtidighet* (*concurrency*) i systemer med en enkel prosessor.
- e) Forklar kort hva som menes med "sideveksling" (paging).
- f) Forklar kort hva som menes med "kappløp" (race condition).

### 2. Grunnleggende om operativsystemer (40%)

- a) Beskriv hvordan '*fetch-execute*'-syklusen i en datamaskin virker. Vis også med et eksempel som adderer to tall på en forenklet modell av en datamaskin.
- b) Batch-systemer (systemer for satsvis behandling) ble utviklet med økt grad av prosessorutnyttelse som mål. Fortell hvorfor dette målet ble satt og forklar hvordan målet ble oppnådd.
- c) Batch-systemer krever maksinvarerstøtte for avbrudd og klokke. Forklar hvorfor.
- d) Tidlige batch-systemer kunne bare ha et program i minnet om gangen. Senere kom batch-systemer som satt av plass til flere programmer i minnet samtidig (multiprogrammed batch systems). Forklar hvordan dette kunne øke prosessorutnyttelse ytterligere under bestemte forutsetninger. Gjør også rede for disse forutsetningene.

### 3. Sikkerhet (30%)

- a) Gi en detaljert beskrivelse av den *miniale ACL* (*access control list*) som vanligvis er brukt til tilgangskontroll i UNIX/Linux-systemer.
- b) Gi en detaljert beskrivelse av hvordan *autentisering* foregår på et klassisk UNIX/Linux-system.
- c) Forklar hvorfor det er uheldig med programmer som tilater at data fra bruker kan medføre bufferoverflyt (buffer overflow). Nevn noen mottiltak og forklar hvordan de virker.

**Lykke til :)**