ANDERS FONGEN

# Operativsystemer

## Et Java-perspektiv

© NKI Forlaget  2002

1. utgave, 1. opplag

Utgiver:NKI Forlaget, Hans Burums vei 30, 1357 Bekkestua
Postboks 111, 1319 Bekkestua
Telefon: Sentralbord 67588800
Ordrekontor 67588900
Telefaks: 67581902
e-postadresse: fapost@adm.nki.no
webadresse: www.nki.no/forlaget

Materialet i denne publikasjonen er omfattet av åndsverkslovens bestemmelser. Uten særskilt avtale med NKI Forlaget er enhver eksemplarfremstilling og tilgjengeliggjøring bare tillatt i den utstrekning det er hjemlet i lov eller tillatt gjennom avtale med Kopinor, Interesseorgan for rettighetshavere til åndsverk. Utnyttelse i strid med lov eller avtale kan medføre erstatningsansvar og inndragning, og kan straffes med bøter eller fengsel.

ISBN 82-562-5743-1

# Innhold

## Om denne boka 1

Hvem er boka skrevet for? 1

Er dette en teoribok? 1

Hvor stort omfang har dette pensumet? 2

Laboratorieøvinger 2

Nettressurser til denne boka 2

Om forfatteren 3

## Kapittel 1 Operativsystemer og mellomvare 5

Hvorfor trenger vi et operativsystem? 5

Oppgavene til et operativsystem 6
- Deling 6
- Abstraksjon 7
- Styring 8

Mellomvare 9

Distribusjon 10
- Hvorfor ønsker vi distribusjon? 11
- Distribuerte operativsystemer 11

Sanntids operativsystemer 12

Historien om Windows 12
- Xerox Star 12
- GEM, MacOS 13
- Windows v.3, OS/2 14
- Windows NT 15
- Sikkerhetsproblemer 15

Historien om Linux 15
- Linus Torvalds, Richard Stallman 17
- Dugnadsånd 17

Kommersiell distribusjon av Linux 18
Open Source vs. kommersiell utvikling 18
Linux 21
Windows 21

## Kapittel 2 Oppbygning av maskinvaren 23

CPU-en bestemmer 24
CPU-ens oppgaver 24
CPU-ens instruksjoner 25
CPU-ens interne organisering 26
Pipelining - overlappet instruksjonsutføring 27
Adresserbare celler 28
Minneceller 28
i/o-celler 28
Adresseres likt 28
Minne lagrer, i/o knytter til omverdenen 28
Bussene 29
Databussen 29
Adressebussen 30
Kontrollbussen 31
En buss-syklus 31
Sammenheng mellom adresselinjer og minnestørrelse 32
Bussenes hastighet 34
Bruk av caching 34
Multiprosessorer 36
Tett eller løst koplet? 38
Når trenger vi multiprosessorer? 38
Blue Gene 39
Avbrudd 39
Telefonen ringer.. 41
..mens du leser en bok 41
Timere 41
Direct Memory Access 42
Får være "sjef" en liten stund 42

## Kapittel 3 Oppbygningen av operativsystemet 47

Konstruksjonskriterier 47
Ytelse, vedlikehold, korrekthet, standarder 47
Skillet mellom virkemåte og grensesnitt 48
Grunnfunksjoner i operativsystemet 49
Prosesshåndtering 50
Minnehåndtering 50
Filhåndtering 50
Utstyrshåndtering 51
(Meldingshåndtering) 53
Konstruksjonsstrategier 53
Inndeling i moduler - abstrakte datatyper 53
Objektorientering 55
Konfigurasjon av operativsystemet 56
Mikrokjerne 57
Bootstrap 57
Grensesnitt til operativsystemet 58
API og SPI 58
Programvareavbrudd - INT-instruksjonen 60
User mode / supervisor mode 60
Funksjonsgrensesnitt eller meldingsgrensesnitt? 61
Programmering i høynivåspråk 63
Kompilator 64
Klassebibliotek 64
Kjøremiljø 65
Organisering av variabler og objekter 65
Stakken og heapen 67
Java Virtual Machine 68
Skallet 68
Tegnbasert skall 69
Symbolbasert skall 69

## Kapittel 4 Prosesser og tråder 73

Hva er en prosess? Hva er en tråd? 73
Kjøkkenet 73
En definisjon 74

Kan vi *se* en prosess eller en tråd? 74

Prosessens tilstand 75
Lagring og gjenskaping 75
Prosess-deskriptorer 76
Trådenes tilstand 76
Slektskap mellom prosesser 77

Administrasjon og styring av prosesser og tråder 78
Kontekst svitsj 78
Kjøreplanen 79
Avbruddshåndtering 80
Kjernetråder vs. brukertråder 82
Init-prosessen 84

Tråder i Java 84

Bakgrunnsartikkel: *Hvorfor er java-tråder så nyttige? 87*
Utføring og beskyttelse 88
Portabilitet 89

**Kapittel 5 Minnestyring 93**

Behov og prinsipper 93
Hva mener vi med «minne»? 94
Beskyttelse og deling 94
Relokasjon 95

CPU-ens adresseringsmekanismer 96
Beskyttede registre 97
Deling av minne med segmentregistre 99

Mulige tildelingsstrategier 100
Stiv tildeling 101
Fleksibel tildeling 102
Fragmentering 104

Virtuelt minne 104
Lagringshierarkiet 105
Paging 107
Segmentering 112

Minnebruk i et høynivåspråk 115

Minnestyring i Intel Pentium 116

## Kapittel 6 Synkronisering I 121


Hva mener vi med «synkronisering»? 121
- Samordning i tid 121
- Vente/varsle i Java - 1.utkast 123

Tre typer av synkronisering 126
- Reservasjon 126
- Vente/varsle i Java - 2.utkast 130
- Klient/tjener 131
- Bufret dataflyt 134

Synkronisering internt i operativsystemet 140
- Kritiske mikroregioner 142


## Kapittel 7 Synkronisering II 145


Hvorfor innebygd synkronisering? 145
- Semaforer 147

Javas synkroniseringsmekanismer 148
- Synchronized-ordet 148
- wait/notify 150
- Spesielle forhold ved monitorer 152
- join 154
- Sikker versjon av FIFO 156

Andre synkroniseringsvarianter 157
- Mange-til-én 157
- Én-til-mange - notifyAll 158
- Tidsgrenser 159

Vranglås 160
- Eksempler på vranglås 160
- Fire forutsetninger 161
- Reservasjonsgrafer 163
- Unngå vranglås? 165

Synkronisering i multiprosessorer 165


## Kapittel 8 Permanent lagring 169


Behovet for permanent lagring 169

Hvordan ser det permanente lageret ut? 171
- Navnesystem 171

Dataorganisering 171
Tilgangsmodell 172

Oppbygningen av en disk 173
Geometri 174
Ytelse 176
Feilkorreksjon 176

Bruk av det permanente lageret 177
Java-kode for lesing av en tekstfil 178

Deling av filer 180
Fil-låsing 181
Atomiske operasjoner 182

Filsystemets oppbygning 182

Lagringstjenesten 183
Administrasjon av plassen 183
Fysisk og logisk integritet 183
Utnytte lagringshierarkiet 184
Tilby et API til omgivelsene 184
Bufring av lese- og skriveoperasjoner 185

Katalogtjenesten 187
Hierarkiske kataloger 188

Metadata 189
Eksistenskontroll (livssykluskontroll) 189
inode-tabellen 190

Ytelse / caching 190

**Kapittel 9 Distribusjon 195**

Hvorfor distribusjon? 195
Datadeling 195
Ressursdugnad 197
Nettverksøkonomi 197
Feiltoleranse 199

Eksempel på distribuerte anvendelser 199
Fjernlagring 199
Programmerte tjenere 200
Fjernutføring av kommandoer 200
Tjenerklynger 200
Peer-to-peer 201
Hierarkiske kataloger 202

Spesielle problemer i distribuerte applikasjoner 204
Ingen felles tilstand 205
Uavhengig krasj og feil 205
Ingen felles klokke 206
Tilstandsløse tjenere 207
Operativsystemets rolle 208
Abstraksjon - «single system image» 208
Styring 210
Deling 212
Meldingsorientert synkronisering 213
Andre synkroniseringsvarianter 214
Multiprosessor-miljø 215
Bruk av Java for distribuert programmering 216
Sockets 216

**Stikkordregister 225**

# Om denne boka

***Operativsystemer – et Java-perspektiv** er skrevet for deg som ønsker å forstå operativsystemer for å bli en bedre programmerer. I boka knyttes teoretiske temaer på området til programmeringsøvelser i Java. Boka forutsetter ikke grundige matematikkkunnskaper, men det forutsettes at leseren har grunnleggende kunnskaper i bruk av operativsystemet Windows og grunnleggende programmering i Java.*

## Hvem er boka skrevet for?

Boka er skrevet for høgskolestudenter som skal ta fatt på fagområdet «operativsystemer» med et omfang på 6 studiepoeng. Den er tiltenkt studenter på lavere grads studier som ikke har fordypning i matematikk fra videregående skole (generell studiekompetanse). Det er en forutsetning med bakgrunn fra programmering i Java, da kodeeksempler er vist i dette språket. Grunnleggende ferdigheter i bruk av pc er også en forutsetning.

## Er dette en teoribok?

Først og fremst *ja*. Den inneholder forslag til laboratorieøvelser innen Linux, Unix og Windows, og en del praktisk informasjon om disse operativsystemene. Den er derimot ingen lærebok i administrasjon eller bruk av et bestemt operativsystem.

Boka omhandler i første rekke de teoretiske aspektene ved et operativsystem, men bruker praktiske eksempler som et hjelpemiddel i denne gjennomgåelsen.

## Hvor stort omfang har dette pensumet?

Boka er laget for Diplomstudiet ved Norges Informasjonsteknologiske Høgskole, NITH, som underviser 2 vekttall innen operativsystemer i forbindelse med kursene «Datateknikk I» og «Datateknikk II». Disse kursene har 9 dobbelttimer med teoretisk gjennomgåelse på forelesninger, og 9 dobbelttimer i laboratorium. Boka er derfor inndelt i ni fagkapitler.

Denne boka har et mindre dyptpløyende teoretisk innhold enn bøker som er skrevet for informatikkstudier på universitetsnivå, og den er skrevet fordi det ikke finnes moderne bøker om operativsystemer på norsk.

Mye historikk, som preger andre bøker, er utelatt, og boka benytter som nevnt Java i eksempler og øvinger.

## Laboratorieøvinger

Hvert kapittel i boka avsluttes med noen kontrollspørsmål og et mål for foreslåtte laboratorieøvinger på nett. Disse kan også være egnet for obligatoriske innleveringer i forbindelse med kurset.

For å gjennomføre øvingene er det nødvendig at du har alminnelig brukertilgang på en maskin med Windows 2000, og det er nødvendig at du kjenner til hvordan Java-programmer skrives inn, kompileres og kjøres på denne maskinen (Java-kompilator med minimum versjon 1.2 må være installert). Dessuten er det nødvendig at studenten har konsolltilgang til en maskin med Linux hvor Java er installert (minimum v.1.2). Noen av de foreslåtte øvingene krever at du har «root»-passordet til Linux-maskinen.

## Nettressurser til denne boka

Til denne boka hører det et nettsted som inneholder forslag til øvinger og pekere til en rekke ressurser innenfor dette området. Fordi slike pekere er «ferskvare» har vi valgt å ikke trykke dem i boka, men samle dem på et nettsted. De blir hyppig gjennomgått for opprydding og ajourhold. Nettstedet ligger på denne adressen:

`http://www.fongen.no/OSbok`

# Om forfatteren

Anders Fongen er cand.scient i informatikk og høgskolelektor ved Norges Informasjonsteknologiske Høgskole, NITH. Han har undervist i operativsystemer ved NITH siden 1996. Fongen er utdannet fra Universitetet i Bergen, og har arbeidet 18 år som programmerer og konsulent før han gikk over til undervisning.

# Kapittel 1

# Operativsystemer og mellomvare

*I dette kapitlet vil vi forklare hvorfor det er nødvendig med et operativsystem, og hvordan operativsystemer og applikasjons-programmer samarbeider. Vi skal også diskutere begrepene «distribusjon» og «distribuerte operativsystemer».*

## Hvorfor trenger vi et operativsystem?

En datamaskin er et ganske komplisert stykke elektronikk som kan utføre programmer. Prosessoren i maskinen gjør faktisk ikke annet så lenge strømmen er påslått enn å lese *programinstruksjoner* fra intern-minnet og utføre dem. For at programmer i maskinen skal kunne gjøre noe nyttig må de kjenne alle detaljer om hvordan maskinen er bygd opp. Skjermkort, nettadapter, mus, tastatur og lydkort må være kjent for programmet.

Programmet som skal kjøre i maskinen må altså inneholde instruk-sjoner som behandler maskinvaren riktig. Dette er en omfattende jobb. Ikke bare er utvalget av maskinvarekomponenter stort og komplisert, men det er også i hurtig omskifting hele tiden.

Vi ønsker at de programvareinstruksjonene som behandler maskin-varen ikke skal være en del av *applikasjonsprogrammene*[1], men heller være skilt ut som et stykke programvare som følger datamaskinen.

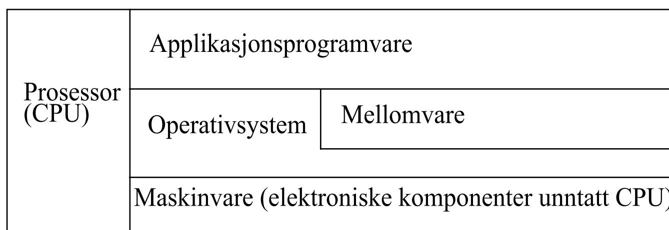
Når du kjøper en datamaskin er det vanlig at et operativsystem (ofte bare kalt OS), f.eks. Microsoft Windows 2000, er installert på maskinen. Fordi operativsystemet inneholder all den programvaren som skal til for å styre denne maskinen, kan en «vanlig» versjon av f.eks. Microsoft Word startes og kjøres på denne maskinen. All «spesialinformasjon» om maskinen er skjult for Word.

---

1. Med dette begrepet mener vi programmer som løser bestemte behov, f.eks. tek-stbehandlingsprogrammer. *Anvendeprogrammer* og *applikasjonsprogrammer* er syn-onymer.

Utviklerne av Word trenger derfor ikke å bekymre seg om hvordan maskinen din er satt sammen. All den nødvendige kunnskap for å utnytte skjermkortet, musa og lydkortet ligger i form av programinstruksjoner som en del av operativsystemet.

Et operativsystem blir som et «mellomlag» som skjuler og styrer maskinvareressursene slik at det blir enklere å lage applikasjonsprogrammer. Figur 1-1 viser dette forholdet. Begrepet *mellomvare* skal vi forklare om litt.




*Figur 1-1: Forholdet mellom prosessor, maskinvare, operativsystem, mellomvare og applikasjonsprogram*

## Oppgavene til et operativsystem

Et operativsystem er et stykke programvare. Alle regler som gjelder for fornuftig programutvikling kommer til anvendelse også når vi skal lage et operativsystem. Vi skal diskutere oppbygningen av et operativsystem i kapittel 3.

Oppgavene til et operativsystem kan deles i tre typer: *deling, abstraksjon og styring.*

### Deling

En pc representerer i de fleste tilfeller en stor investering og eieren ønsker at utstyret skal utnyttes best mulig. De fleste som bruker en pc trenger en skriver av og til, men ikke ofte nok til at det er fornuftig å kjøpe og vedlikeholde en egen skriver for hver enkelt bruker. Ett av operativsystemets oppgaver blir derfor å sørge for at skriveren kan brukes av flere samtidig og dermed stå minst mulig uvirksom.

Vi tar det kanskje også som en selvfølge at alle som bruker en pc kan lagre dataene sine på den samme harddisken, men det er operativsystemet som sørger for at dette er mulig.

Det er hovedsakelig maskinvarekomponenter som kan deles på denne måten. Ikke bare skriver og harddisk, men også plassen i internminnet og prosessorkraften må deles mellom alle som bruker maskinen.

**Multiprogrammering** Deling av prosessorkraft mellom flere oppgaver skjer ved at prosessoren jobber en stund med én oppgave før den fortsetter med den neste. I operativsystemet ligger det derfor noen oppgaver som er «midtveis», og som blir utført stykkevis. Skiftet mellom oppgaver skjer ofte (10–50 ganger/sekund), og for en bruker gir dette inntrykk av at oppgaven blir utført uten opphold.

**Hvem er «brukerne» av en datamaskin?** Det er ikke bare mennesker som er brukere av en datamaskin. En datamaskin må utføre oppgaver som er gitt av andre maskiner, eller oppgaver som hører til «husholdningen» av maskinen. Regelmessig vil en datamaskin overføre e-post, ta sikkerhetskopi av harddisken eller lete etter datavirus. Slike oppgaver er også inkludert i uttrykket «brukere», og delingen av maskinutstyret skal også omfatte disse.

**Sikkerhetsproblem kan oppstå!** Der hvor utstyret skal deles mellom flere brukere oppstår det alltid et spørsmål om sikkerheten blir tilstrekkelig tatt vare på. Slike spørsmål kan være:

- Kan én bruker *få se* dataene til en annen bruker? Om en bruker får tildelt plass på disken eller i minnet, ligger da det forrige innholdet fortsatt der fra den gang plassen ble brukt av en annen? Blir alle utskrifter liggende slik at alle kan lese hverandres utskrifter?
- Kan én bruker *endre* dataene som tilhører en annen bruker? Er plassen som tildeles en bruker for lagring av data (minne eller harddisk) svakt beskyttet, slik at andre kan lese og endre innholdet av disse dataene?
- Kan én bruker *hindre* en annen bruker i å utføre sine oppgaver? Kan en bruker ta opp så mye plass i minnet eller på harddisken at andre ikke får oppfylt sine plassbehov? Kan en bruker monopolisere en skriver i flere døgn ved å sende svære utskriftsjobber?

De svarene vi ønsker å få på disse spørsmålene avhenger av det brukermiljøet vi har med å gjøre. Er dette et miljø av samarbeidende brukere som tar hensyn til hverandre, eller har vi med konkurrerende eller fiendtlige brukere, som «stjeler og lyver» og bare tenker på seg selv?

## Abstraksjon

Når du bruker et operativsystem, ser harddisken ut som et sett av kataloger og filer, fint og ryddig organisert slik at du lett kan flytte deg mellom kataloger og se filene som ligger der. Alle filene og katalogene har sine egne navn, og de er påført opplysninger f.eks. om når de først ble skapt.

På selve magnetplatene på disken er det derimot ingen spor etter disse strukturene. Alt vi finner her er magnetiske spor på hver plate delt inn i buestykker kalt *sektorer*. Kombinasjonen av sektornr., overflate og spor er «koordinatene» til et lagringssted på diskens overflate.

Programvaren har ikke lyst til å lete seg frem på disken med disse koordinatene, og operativsystemet tilbyr derfor en «abstraksjon» (et bilde) av disken i form av et *filsystem*.

Operativsystemet tilbyr de fleste av sine tjenester og ressurser i form av abstraksjoner.

- Skriveren ser ut som en tjeneste som tar et filnavn og som fremstiller innholdet på papir.
- En kommunikasjonsprotokoll ser ut som et «hull» som man sender og mottar bytes gjennom.
- Internminnet ser ut som om en serie sammenhengende minneplasser, mens deler av det i virkeligheten ligger på harddisken.

**Objektorienterte abstraksjoner** Tenk på objektorientert programmering. Der fremstilles et sett av minneplasser som «objekter», som har en tilstand, og som det kan utføres operasjoner på. Operativsystemets abstraksjoner følger gjerne denne tankegangen, og vil fremstille sine «ressurser» på denne måten, f.eks.

- andre maskiner i nettverket
- andre brukere på maskinen
- oppgaver under utførelse
- utskriftsjobber i kø
- filer og kataloger

## Styring

Den tredje gruppen av oppgaver for et operativsystem har med det vi innledningsvis i dette kapitlet begrunnet behovet for et operativsystem med, nemlig å avlaste applikasjonsprogrammet for oppgaven med å styre maskinvaren selv.

### Styringsoppgavene har to hensikter

1. Tilby et funksjonsgrensesnitt for applikasjonsprogrammer slik at maskinvareressurser får et enhetlig utseende. Ulike nettverkskort skal kunne betjenes med funksjonskall som er mest mulig like, slik at programvaren som bruker nettverkskortet kan kalle på disse enhetlige funksjonene fremfor å kjenne den detaljerte virkemåten ved hvert enkelt kortfabrikat.

2 Gi en overordnet administrasjon av delte maskinvareressurser. Der hvor mange brukere vil dele en skriver må operativsystemet være den som ordner køen av utskrifter og bestemmer rekkefølgen av utskriftene. Det er operativsystemet som skal bestemme hvordan prosessorkraften skal deles (kjøreplan) og som bestemmer hvordan det tilgjengelige minnet skal fordeles mellom oppgavene i maskinen. Det vil ofte foreligge en konflikt mellom å velge den beste løsningen for en enkelt oppgave og den som gir den beste utnyttelsen av hele maskinen.

Styringsoppgavene er noe som foregår internt i operativsystemet og som på sett og vis er hjelpefunksjoner for de andre to typene av oppgaver, deling og abstraksjon. I den objektorienterte modellen som vi nevnte ovenfor er styringsoppgavene å betrakte som *metodene* i objektet.

# Mellomvare

Det finnes programvare som verken er en del av operativsystemet eller en del av applikasjonsprogramvaren. Slik programvare har til oppgave å tilby et funksjonsgrensesnitt for spesielle tjenester som ikke er en del av operativsystemet.

Et eksempel på slik programvare har du på maskinen din om du har installert programvare for å bruke en databasetjener. Programvaren har følgende oppgaver:

- Formidle kontakt gjennom nettverket til databasetjeneren, dvs. betjene de nødvendige nettverksprotokollene.
- Tilby et funksjonsgrensesnitt som applikasjonsprogrammet kan bruke for å kalle på de tjenestene (med metodekall) som trengs for å åpne en database og utføre SQL-setninger.

Slik programvare kaller vi *mellomvare,* og oppgavene til mellomvaren kan også deles inn i gruppene deling, abstraksjon og styring.

### Forskjellen mellom operativsystem og mellomvare

1 Operativsystemet er uunnværlig, og betjener de grunnleggende funksjoner som alle programmer har behov for. Mellomvaren er et tilleggsprodukt som man normalt skaffer separat.

2 Operativsystemet har ansvar for å styre maskinvaren. Mellomvaren gjør aldri det, men kaller på operativsystemets tjenester når det er nødvendig å benytte maskinvaren.

Mange slags programprodukter kan kalles mellomvare. Eksempler på mellomvareprodukter kan være:

**Databasetjenere** Produkter som lar deg lagre data i tabeller og hvor du kan gjøre operasjoner på tabellene med SQL-setninger. Databasetjeneren kan være et program installert på din egen maskin, eller på en annen maskin i nettverket. I det siste tilfellet er en databasetjener velegnet for å dele data mellom mange maskiner i et nettverk. I begge tilfeller må din maskin ha installert mellomvare for å gi et funksjonsgrensesnitt for bruken av databasetjeneren.

**Objektmonitorer** Produkter som lar programmer under utførelse på forskjellige maskiner snakke med hverandre på en objektorientert måte. Et objekt på én maskin kan kalle på metoder på et objekt på en annen maskin. Vi skal omtale såkalte *fjernobjekter* i kapittel 8.

**Transaksjonsmonitorer** Produkter som samordner forespørsler mot flere databasetjenere samtidig, slik at feilsituasjoner blir korrekt behandlet. Transaksjonsmonitorer bidrar også til å regulere belastningen på databasetjenerne for å unngå trafikkork. Denne boka omtaler ikke transaksjonsmonitorer.

## Distribusjon

En viktig oppgave til dagens operativsystemer er å samordne maskiner som er koplet sammen i nettverk. Det koster bare et par hundrelapper å kople en datamaskin til et nettverk, og alle moderne operativsystemer har nå mange funksjoner for å støtte slikt samarbeid. De tjenestene vi oftest ser i forbindelse med såkalt distribusjon er:

**Felles disklager** Maskiner i nettverk kan lese og skrive data på andres harddisk som om det var en harddisk på deres egen lokale maskin.

**Felles utskrift** Maskiner kan skrive ut på skrivere tilkoplet andre maskiner i nettverket.

**Filoverføring** Data lagret på én maskin skal kunne overføres til en annen maskin og lagres der.

**Fjernutføring av kommandoer** Et program under utføring på én maskin skal kunne starte en kommando på en annen maskin. Dette kan innebære at hele programmer skal startes, eller at det skal kalles på delprogrammer (funksjoner/metoder). Resultatet av operasjonen skal returneres til programmet på maskinen som startet kommandoen.

**Fjernadministrasjon** I nettverk med et stort antall maskiner er det nødvendig at all konfigurasjon og administrasjon av maskinen og operativsystemet kan skje «på avstand» fra et sentralt sted. Noe feilsøking skal også kunne foregå fra et sentralt sted.

### Hvorfor ønsker vi distribusjon?

Distribuert databehandling kan gi fordeler i mange situasjoner. Tre vanlige situasjoner er:

- Brukere ønsker å behandle data lagret på forskjellige maskiner. Det kan være aktuelt å samle dem på ett sted før behandling, eller behandle dem direkte der de befinner seg. Det siste er oftest aktuelt der data skal deles av mange.
- Brukere på forskjellige maskiner ønsker å samarbeide. Det kan innebære at de ønsker å skrive en bok sammen, at de ønsker tale- eller videokonferanser, eller at de skal drive saksbehandling sammen.
- Vi ønsker å aggregere prosessorkapasitet spredt rundt på mange maskiner for å utføre store og krevende beregninger. Datamaskiner med superhøy ytelse er svært dyre i anskaffelse og blir fort umoderne, og er derfor risikable investeringer. Dersom mange enkle maskiner i nettverk kan gi det samme resultatet ved å samarbeide om beregningsoppgaven, kan det gi billigere og mer *skalerbare*[2] løsninger.

### Distribuerte operativsystemer

Et distribuert operativsystem er et operativsystem hvor de grunnleggende tjenestene for distribuert databehandling er støttet.

- Fjernlagring
- Fjernutskrift
- Fjernutføring av kommandoer
- Filoverføring
- Fjernadministrasjon

**Hvilke operativsystemer er distribuerte?** Vi vil i denne boka omtale operativsystemene Microsoft Windows NT og Linux. I hvilken grad kan disse operativsystemene kalles distribuerte?

- Microsoft Windows NT regnes som et distribuert operativsystem, fordi det inkluderer tjenestene som er nevnt ovenfor. Windows NT er derimot lite *kompatibelt* med andre operativsystemer, og fungerer best der det er maskiner med Windows NT i begge ender av forbindelsen.
- Linux er bare delvis et distribuert operativsystem. Fjernutføring av kommandoer og fjernadministrasjon krever tilleggsprogramvare (mellomvare). Linux er derimot, i motsetning til Windows NT, mer kompatibelt med andre operativsystemer.

---

2. Skalerbarhet betegner systemets evne til å vokse i størrelse.

## Sanntids operativsystemer

Det eksisterer en gruppe operativsystemer som kalles «sanntids» (eng. real time) operativsystemer. De benyttes i anvendelser hvor det er viktig å gi *tidsmessig respons* på hendelser, og er typisk å finne i reguleringssystemer.

Inndata til en datamaskin trenger ikke komme fra et tastatur eller en mus, men kan også være måledata fra et termometer. Utdata trenger ikke å være på skjerm eller papir, men kan være en ventil som åpner eller lukker seg.

Vi finner reguleringssystemer i alt fra vaskemaskiner til atomkraftverk, og alle har de sensorer som måler tilstander og aktuatorer (mekaniske innretninger som styrer f.eks. ventiler og luker) som korrigerer tilstander. En temperatur som er blitt for høy krever korreksjon innen en viss tid, ellers kan noe koke over eller brenne opp. Derfor er det nødvendig at et operativsystem som støtter et reguleringsprogram kan garantere at en avlest tilstand blir behandlet innen et bestemt tidsrom.

Sanntids operativsystemer er spesialiserte produkter som ikke brukes i generelle datamaskiner, og vi vil ikke omtale dem i denne boka.

Selv om sanntids operativsystemer stiller større krav til hvordan denne delingen foregår, er de ofte enklere produkter enn de generelle operativsystemene (f.eks. Linux eller Windows NT). Dette skyldes at sanntids operativsystemer oftest har enklere funksjoner for abstraksjon, deling og distribusjon enn de generelle operativsystemene. Oftest har de også et lavt sikkerhetsnivå.

## Historien om Windows

### Xerox Star

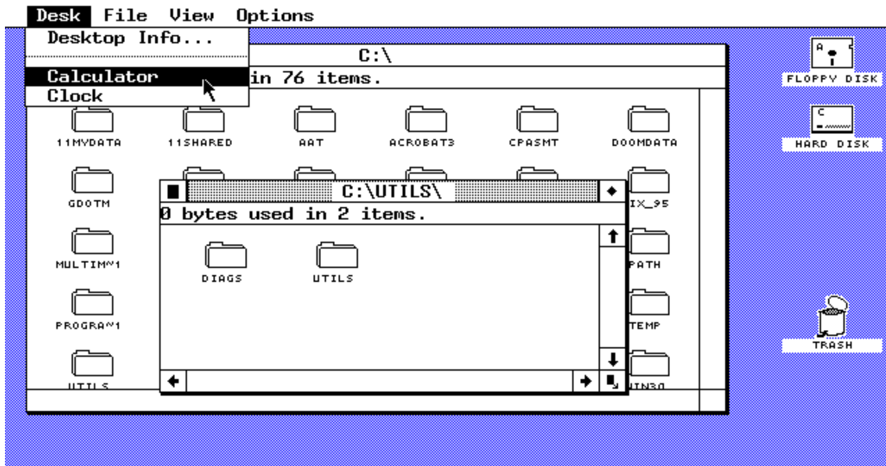
Det er slett ikke Windows som var det første operativsystemet med symbolbasert betjening. Heller ikke Macintosh med MacOS. Derimot var det Xerox som oppfant både mus og «ikoner», og betjeningsmåten hvor man manipulerer ikoner med museoperasjoner. Dette foregikk på Xerox' forskningssenter PARC (Palo Alto Research Center) rundt 1980.

På denne tiden ble også IBM PC lansert, med operativsystemet MS-DOS som bare hadde tegnbasert betjening (kommandolinje). Tegnbasert betjening var det alle var vant til den gang, og symbolbasert betjening slik det ble laget for Apple Lisa (forløperen til Macintosh) var forbeholdt dyre og eksperimentelle maskiner.

*Figur 1-2: Xerox Star (a), Apple Lisa (b) og IBM PC (c)*

## GEM, MacOS

Microsoft startet utviklingen av MS Windows, men før det kom på markedet var det faktisk Digital Research (et firma som er glemt i dag) som lyktes å lage et symbolbasert grensesnitt for IBM PC. Produktet het GEM, og lignet av utseende mye på MacOS. Programtilbudet til GEM forble ganske magert, og produktet fikk ingen stor utbredelse (ble også benyttet i maskinen Atari ST).




*Figur 1-3: Skjermbilde fra GEM*

Microsoft Windows kom på markedet rundt 1985, men både versjon 1 og 2 var uestetisk og unyttig (lite tilgjengelig programvare). Både GEM og MS Windows ble laget som ekstraprogrammer som ble «lastet» oppå MS-DOS, og var derfor ikke selvstendige operativsystemer.

## Windows v.3, OS/2

Med versjon 3 (1990) av MS Windows fikk utseendet en kraftig forbedring, og litt etter litt økte programvaretilbudet, bl.a. ble Word og Excel tilgjengelig på denne tiden. Fortsatt ble Windows v.3 lastet oppå MS-DOS. MS-DOS var aldri ment å brukes på denne måten, og det oppsto mange problemer i maskinen som skyldes mangelfull programvarearkitektur. På samme tid ble det vanlig å knytte pc-ene sammen i lokalnettverk, men mangelen på fornuftig minnestyring i MS-DOS gjorde det «trangt» i en maskin som skulle ha nettverks-programvare i tillegg til nødvendige applikasjoner.

Microsoft startet rundt 1987 et samarbeid med IBM om utvikling av en erstatning for MS-DOS. Det nye operativsystemet skulle hete OS/2, og kom på markedet i versjon 1 før samarbeidet sprakk. Microsoft valgte å satse på utviklingen av sitt eget produkt, kalt Windows NT, mens IBM fortsatte utviklingen av OS/2 (som fortsatt er et IBM-produkt).

IBM utviklet OS/2 (i versjon 2) til et teknisk vellykket produkt, det var mye mer stabilt og effektivt enn Windows v.3. OS/2 hadde ikke stor tilgang på «egen» programvare, men kjørte både MS-DOS og Windows-programvare uten problemer. Markedet var derimot lite villig til å ta det i bruk, mange lot heller til å vente på det kommende produktet fra Microsoft.

### Windows NT

Windows NT (New Technology) var en stor satsing fra Microsoft; et operativsystem skrevet fra bunnen av og med ambisjoner om å være et generelt operativsystem både for arbeidsstasjoner og for tjenermaskiner. Først i versjon 4 ble det noenlunde akseptert av markedet, og da i hovedsak som operativsystem for kraftige pc-er og for tjenermaskiner. Windows NT var ressurskrevende, men stabilt. pc-er med mindre ressurser ble anbefalt å vente litt til på Windows 95 (lansert i 1995), som skulle være bedre egnet for f.eks. hjemme-pc-er.

Resten av historien kjenner vi. Microsoft Windows har en markedsandel på personlige maskiner på 95 %, og programvaretilbudet er overveldende. Windows har orientert seg mot Internett med TCP/IP-programvare som standard del av operativsystemet, og har vært en viktig pådriver også i utviklingen av Internett-bruk.

### Sikkerhetsproblemer

Men uniformeringen har hatt sin pris. Mangelfull sikkerhet har vært et gjennomgående problem i produkter fra Microsoft. Det later til at de alltid har prioritert funksjonalitet fremfor sikkerhet. Utbredelsen av *datavirus* beror helt og holdent på mangelfull sikkerhet i MS-DOS (og pga. kravet om *kompatibilitet* var det ikke mulig å forbedre dette senere).

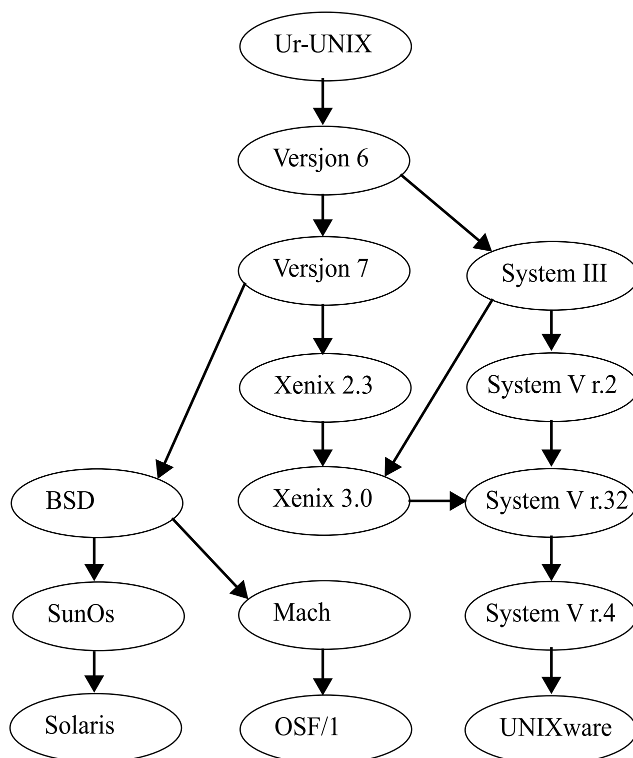
Kombinasjonen av svak sikkerhet, orientering mot nettverk og stor markedsandel skaper ideelle forhold for utbredelse av virus, ormer, bomber m.m. Det finnes mange produkter for å beskytte seg, men det er oftest opp til brukerens ansvar, initiativ og kunnskap å utnytte disse.

## Historien om Linux

Linux er en avlegger av operativsystemet UNIX, lansert som et forskningsprodukt i 1974 av Ritchie og Thompson ved Bell Laboratories. Etter mange år begrenset til forskning og utvikling, og enda noen år som operativsystem i avanserte og spesialiserte arbeidsstasjoner (bl.a. data-assistert konstruksjon, DAK), ble UNIX et alternativt operativsystem på pc-er rundt 1985. Historien om UNIX er broket; allerede den første pc med harddisk (IBM XT) kunne utstyres med UNIX, og senere versjoner dro også nytte av den forbedrede arkitekturen i IBM AT (80286-prosessoren) og senere modeller med 80386-prosessor.

UNIX eksisterer i mange ulike versjoner, og de har et felles stamtre som viser hvordan de ulike versjonene har oppstått over tid. Disse forskjellene, og det faktum at UNIX finnes for mange forskjellige

prosessorer har begrenset populariteten noe. Det at et program «finnes for UNIX» har nemlig ikke vært noen garanti for at det lar seg kjøre på din spesielle maskin. UNIX-historien er derfor full av alle slags allianser og konsortier som har forsøkt å lage standarder for en felles-versjon av UNIX, men uten at det har påvirket dette forholdet noe særlig. Man har hørt programutviklere si «Vårt produkt finnes i versjoner for 50 operativsystemer. 40 av dem heter UNIX». En av de mest kjente standardene kalles POSIX (Portable Operating System Interface).




*Figur 1-4: Slektstreet til UNIX*

Skikkelige UNIX-versjoner har vært ganske dyre sammenlignet med MS-DOS og Windows. Og det har opp gjennom tiden vært flere initiativ til å lage gratis-versjoner av UNIX[3] (laget på dugnad helt fra

3. UNIX er et varemerke, og ikke noe gratisprodukt får lov til å kalle seg UNIX, men må velge andre navn.

bunnen av). Minix, FreeBSD og Linux har vært de mest kjente gratisversjoner av UNIX. Linux er den som har fått mest interesse, og vi velger å bruke Linux som en referanse i denne boka.

## Linus Torvalds, Richard Stallman

I en melding på en USENET nyhetsgruppe (elektronisk konferansetavle) i august 1991 forteller en finsk gutt at han har laget kjernen til et operativsystem og at han inviterer til synspunkter på hva en slik kjerne bør inneholde. Operativsystemet het Linux, gutten het Linus Torvalds og han er fortsatt en av de sentrale personene i videreutviklingen av Linux. Vi antar at første beta-versjon ble sendt ut i september samme år. Etter hvert ble flere personer interessert i videreutviklingen av Linux, slik at tilbudet av verktøyprogrammer økte raskt. Den første tiden var det kun en versjon for 80386-prosessoren som ble utviklet, og fra starten av var det POSIX de forsøkte å følge under utformingen av systemets virkemåte.

Et annet dugnadsprojekt som allerede hadde eksistert en stund, var GNU. Forkortelsen er en spøk og betyr «Gnu is Not Unix». GNU-prosjektet ble startet av Richard Stallman og hadde som mål å lage gratis verktøy-programmer, men med tiden også et helt operativsystem. GNU laget editoren Emacs, og egne versjoner av Zip, Awk, Lex, Yacc m.m. (Lex og Yacc er verktøyer for å lage kompilatorer).

GNU og Linux passet sammen som hånd i hanske, og raskt etablerte paret seg med et operativsystem som faktisk kunne brukes til noe. Mye av programvaren som fulgte med kommersielle UNIX-versjoner var opprinnelig utviklet i ikke-kommersielle miljøer (bl.a. X11, det grafiske vindussystemet), og derfor var det mulig å flytte denne programvaren over til Linux uten at det kostet noe.

## Dugnadsånd

Linux har skapt en dugnadsånd vi knapt har sett maken til. Hundrevis av frivillige har flyttet programvare, videreutviklet kjernen, utviklet ny programvare eller laget støtteprogrammer for nettverkskort, disker, lydkort osv. Linux har nå støtte for majoriteten av moderne maskinvare, multiprosessormaskiner, og har en standardfunksjonalitet som langt overgår Windows-produktene. Vi finner store RDBMS-produkter på Linux, velutviklede kontorstøttesystemer (K-Office, OpenOffice), og alt sammen er gratis.

### Kommersiell distribusjon av Linux

Det er fritt frem å laste ned Linux fra Internett og installere det sammen med all slags programvare, men det er litt komplisert å skulle kompilere så mye kildekode, sette opp disker og konfigurasjonsfiler riktig m.m. Dersom man ønsker en «ferdigpakket» versjon av Linux med komplette programmer og enkel installasjon kan man kjøpe et kommersielt Linux-produkt. Prisen for dette er lavere enn proprietære systemer (MS Windows) og innebærer mye spart arbeid for brukeren. Programvaren på en slik CD-ROM vil være tilpasset og uttestet, og leverandøren kan også tilby brukerstøtte. De mest kjente produktene heter

- Red Hat Linux
- SuSe Linux
- Progeny Debian
- Linux-Mandrake
- Caldera Open Linux

Adressene til disse leverandørene finner du på bokas nettsted.

## Open Source vs. kommersiell utvikling

Windows og Linux er de mest aktuelle alternativene til et operativsystem for skrivebordsmaskinen, og det er interessant at de samtidig representerer to helt ulike utviklingsstrategier.

Microsoft Windows er et «vanlig» åndsverk, i den forstand at eierskapet er helt fastlagt. Programmet utvikles, selges og støttes av Microsoft, som gjør alt arbeidet og får alle pengene. Bruk og videresalg av programvaren er underlagt omfattende og detaljerte lisensbestemmelser. Pengene som Microsoft tjener på salget går bl.a. med til å videreutvikle produktet.

Linux er derimot utviklet i såkalt «Open Source». Det betyr at alle kan hente kildekoden til Linux og endre den etter eget behov. Man kan også distribuere nye produkter, men ikke selge dem kommersielt. Open Source-programvare har en eier, men eieren kan aldri kalle tilbake et program som en gang er utgitt på disse vilkårene. Dette er en strategi som minner om en stor dugnad. Ingen jobber lønnet, men alle har nytte av det som lages.

Tanken om Open Source har eksistert ganske lenge i form av såkalt «freeware», dvs. programvare som distribueres fritt i binærform (ikke som kildekode). Open Source går lenger enn dette, og tillater som nevnt at produkter blir endret og distribuert videre under et nytt navn.

Viktige programmer som er utviklet under OpenSource er:

- Linux operativsystem
- Mozilla nettleser, e-postleser og HTML-editor
- Apache web-tjener
- OpenOffice kontorpakke

Viktige programmer som ikke er Open Source, men som distribueres fritt i binærform:

- Internet Explorer nettleser
- Java programmeringsspråk (alle Java-produktene)
- Perl programmeringsspråk
- PHP programmeringsspråk for web-tjenere

## Sammendrag

- Operativsystemer er uunværlige i generelle datamaskiner. Operativsystemet «følger maskinen» og skaper et *kjøremiljø* for applikasjonsprogrammene.
- Operativsystemer tilbyr deling, abstraksjon og styring.
- Operativsystemer og mellomvare er beslektede produkter.
- Distribuerte operativsystemer støtter deling av disklager og utskrifter og fjernutføring mellom maskiner koplet sammen i et nettverk.
- Sanntids operativsystemer brukes i tidssensitive applikasjoner.

| **Sentrale begreper i dette kapitlet** | |
| --- | --- |
| Operativsystem | Applikasjonsprogram |
| Mellomvare | Deling, abstraksjon og styring |
| Distribusjon | Sanntid |
| Proprietær utvikling | Windows |
| Open Source-utvikling | Linux |

## Teorioppgaver

**Gå sammen i grupper og besvare disse oppgavene, helst skriftlig**

1 Tenk deg programvaren i følgende enheter, og beskriv hvilke oppgaver det har, inndelt i kategoriene deling, abstraksjon og styring.

- En mobiltelefon
- En filtjener i et nettverk
- Din egen personlige datamaskin
- En søkemotor på Internett
- En håndholdt datamaskin (Palm eller PocketPC)

2 En relasjonsdatabase (RDBMS) kan bygges inn som en fast del av et operativsystem. Nevn noen fordeler og ulemper ved en slik løsning.

3 Linux har ikke gjort noe gjennombrudd som operativsystem for arbeidsstasjoner, men er blitt ganske populært som operativsystem for tjenermaskiner. Forklar årsaken til dette forholdet.

4 Diskuter fordeler og ulemper ved bruk av gratis programvare som f.eks. Linux.

5 Man kan velge å kjøpe all programvare i en bedrift fra samme leverandør. Diskuter fordelene og ulempene ved et slikt valg.

6 Finn frem til nettsiden for operativsystemet QNX (www.qnx.com). Hva slags operativsystem er dette, og hvilket anvendelsesområde retter produktet seg mot?

## Øvingsoppgaver

Denne boka er ingen lærebok i bruk eller administrasjon av Linux. Derfor krever disse øvelsene at du har tilgang til annet instruksjonsmateriell. Mye slikt materiell finnes på Internett, og nettstedet som følger denne boka har pekere til gode nettsteder for å lære seg bruk av Windows og Linux.

Øvingsoppgavene er delt opp i to grupper for henholdsvis Windows og Linux, og er laget både for å illustrere den gjennomgåtte teorien, og for å øke de generelle ferdighetene i bruk av operativsystemet. Etter dette generelle innledningskapitlet vil vi fokusere på ferdigheter i vanlig bruk av Linux og Windows.

## Linux

Betjeningen av Linux kan skje på flere måter:

- Gjennom et tegnbasert skall, hvor du skriver kommandoer på en kommandolinje og trykker `<enter>` for å få utført kommandoen.
- Gjennom et symbolbasert skall (grafisk brukergrensesnitt - GUI) hvor du manipulerer filer og programmer i form av «ikoner» gjennom museoperasjoner.

Heretter skal vi ta utgangspunkt i at betjeningen skjer gjennom et tegnbasert skall som skalles «bash» (Bourne Again Shell). Om du bruker et symbolbasert skall, må du starte et «terminalvindu» for å bruke det tegnbaserte skallet der.

**Bruk nå en instruksjonsbok eller hjelpemidler du finner på Internett, og sørg for at du lærer deg disse ferdighetene:**

1. Bruk av utdelt brukernavn og passord. Du må ved starten av arbeidet oppgi disse for å slippe inn i maskinen. I en maskin med symbolbasert skall trenger du å oppgi brukernavn og passord i et separat vindu før du får starte et terminalvindu med skallet.
2. Forstå begrepet «working directory» (arbeidskatalog). Opprette og slette underkataloger. Kopiere og flytte filer mellom kataloger.
3. Opprette og redigere tekstfiler med en editor.
4. Kompilere og kjøre Java-programmer.
5. Overføre filer gjennom nettverket med File Transfer Protocol (ftp).
6. Kunnskap om de vanlige hjelpekommandoene *more, cat, grep, ls, rm, wc, find*.

## Windows

Vi tar utgangspunkt i at du har adgang til en maskin som kjører Windows NT eller Windows 2000. Vi tar utgangspunkt i det symbolbaserte skallet til Windows som kalles «Windows utforsker»[4]. Almin-

4. Vi vil bruke betegnelsene fra Windows i norsk versjon.

nelig kjennskap til betjeningen av Windows er så utbredt at vi ikke gir noen veiledning her, bare en pekepinn på hva du bør beherske for å arbeide effektivt med stoffet i boka.

**Disse ferdighetene bør du ha:**

1. Bruk av utdelt brukernavn og passord for å logge deg inn på maskinen
2. Organisere vinduer på skrivebordet: flytte, endre størrelse, minimere og maksimere. Bruk av verktøylinjene
3. Stille klokken på maskinen.
4. Åpne «Min Datamaskin» og navigere rundt mellom katalogene i filsystemet.
5. Høyreklikke i en katalog og opprette nye filer av ulike typer.
6. Høyreklikke på en fil og utføre operasjoner på filen, bl.a. redigere innholdet, endre navn og slette filen.
7. Flytte og kopiere filer mellom kataloger og disker.
8. Redigere, kompilere og kjøre et Java-program.
9. Overføre en fil mellom maskiner med File Transfer Protocol.
10. Laste ned en fil med nettleseren og plassere den på et angitt sted i filsystemet.
11. Bruke «Søk»-programmet (F3) til å lete etter filer med ulike søkekriterier.

Kapittel 2

# Oppbygning av maskinvaren

*I dette kapitlet skal vi forklare litt av maskinvarens virkemåte. Vi skal ikke diskutere maskinvaredesign i detalj, men forklare de prinsippene som er relevante for et kurs i operativsystemer.*

Et operativsystem må samarbeide tett med maskinvaren[1] for å løse oppgavene sine. Derfor er det nødvendig at en kjenner til hovedprinsippene for virkemåten av maskinvaren for å forstå et OS.

**En datamaskin vil alltid inneholde minimum disse komponentene:**

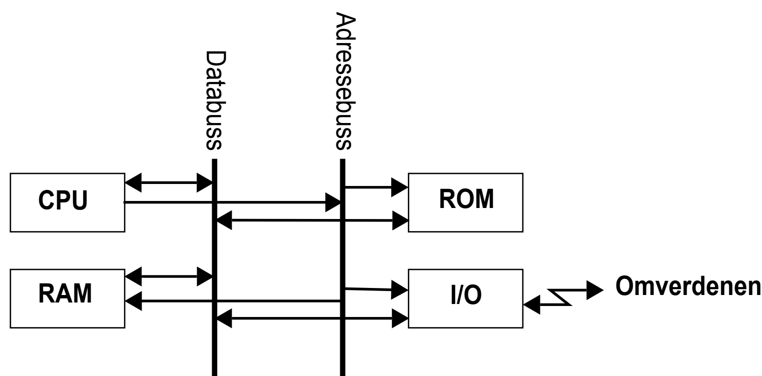
1. Prosessor (CPU – Central Processing Unit) som utfører instruksjoner.
2. Arbeidsminne (RAM – Random Access Memory) som lagrer data og instruksjoner midlertidig (så lenge strømmen er på).
3. Programminne (ROM – Read-Only Memory) som lagrer program og data permanent (også når strømmen er avslått).
4. I/O-kretser (f.eks. parallell- eller serie-port) som tar imot data fra omverdenen og sender ut resultatdata.
5. Busser som forbinder disse komponentene sammen i et elektrisk kretsløp.

Legg merke til at komponenter som tastatur, skjerm eller harddisk ikke er obligatorisk i en datamaskin. En vaskemaskin er ofte utstyrt med en datamaskin uten noen av disse komponentene.

Figur 2-1 viser skjematisk hvordan bussene (kalt adressebuss og databuss) lager *signalveier* mellom de andre komponentene (vist med pilspisser). Signalene som overføres er pulser av svak spenning, f.eks. 5 volt. Adressebussen og databussen er begge *knipper* av elektriske ledere som kan overføre flere pulser i parallell.

1. Med maskinvaren mener vi all elektronikken som er i virksomhet i en datamaskin.

Legg merke til at databussen overfører signaler begge veier mellom alle komponentene i datamaskinen, mens adressebussen overfører signaler fra CPU-en til de andre komponentene. Vi skal omtale bussene senere i dette kapitlet, men først skal vi beskrive den absolutte sjefen i enhver datamaskin:




*Figur 2-1: Hovedkomponentene i en datamaskin*

## CPU-en bestemmer

Hjertet og hjernen i enhver datamaskin er prosessoren (CPU), og prosessoren er den som bestemmer alt som skal skje i maskinen. Utføringen av programinstruksjoner skjer her, og innholdet av instruksjonene er det som i sin tur bestemmer hvordan maskinvaren skal behandles og hva som skal skje med de lagrede dataene.

Om du titter under lokket på en datamaskin, finner du mange svarte «brikker» montert på kretskort. Disse brikkene er *integrerte kretser* som inneholder tusenvis av transistorer og andre elektroniske komponenter. I en datamaskin har disse brikkene logiske funksjoner, dvs. at de hjelper til med å behandle data i to-tallssystemet. En av de største brikkene i datamaskinen pleier å være CPU-brikken.

### CPU-ens oppgaver

Oppgavene til en CPU er egentlig ganske enkle.

**Dekode instruksjoner** CPU-en skal hente instruksjoner fra minnet (RAM eller ROM). Den gjør det ved hjelp av signaler på adresse- og databussen. En instruksjon er en serie med 1- og 0-bits som beskriver hva som skal gjøres. CPU-en må nå, etter at instruksjonen er hentet, finne ut hva slags instruksjon det er (dekoding).

**Utføre instruksjoner** Når en instruksjon er hentet og dekodet, skal den utføres. Utføring av en instruksjoner kan innebære å

- gjøre tallberegninger på data som ligger inne i CPU-en eller i minnet
- teste en minnecelle eller en indikator (flagg) for en bestemt verdi
- kalle på en programfunksjon eller hoppe til en annen instruksjon i minnet

Når en instruksjon er ferdig utført vil CPU-en normalt hente den neste instruksjonen i minnet, og dekode og utføre den (med mindre den har utført en instruksjon som ber om en endring i instruksjonsrekkefølgen).

## CPU-ens instruksjoner

Her ser du et lite program av CPU-instruksjoner skrevet i såkalt *assemblerspråk*. Vi erstatter hver instruksjon (som egentlig er en tallverdi) med *symbolske* instruksjoner.

| Instruksjon | Forklaring |
|---|---|
| `mov ax,[210]` | Flytt verdien av minnecelle nr. 210/211 inn i registeret AX |
| `mov bx,234` | Flytt verdien 234 inn i registeret BX |
| `add ax,bx` | Addere verdien av AX til registeret BX |
| `cmp ax,500` | Sammenligne verdien av registeret AX med 500 ... |
| `jge 112` | ... og hopp til adresse 112 om AX er større eller lik |
| `mov [230],ax` | Flytt verdien av AX til minnecelle nr. 230/231 |
| `jmp 115` | Hopp til instruksjonen i minnecelle nr. 115 |
| `mov [232],ax` | Flytt verdien av AX til minnecelle nr. 232/233 |
| `int 20` | Avslutt programmet (systemkall til MS-DOS) |

Alle tall i dette eksemplet er heksadesimale. Dette er den vanligste måten å angi minneadresser på. Programmet legger sammen to 16-bits verdier, og legger dem på én av to steder, avhengig av om summen er mindre enn 0x500 eller ikke.

Dette eksemplet er assemblerspråk for Intel-familien av prosessorer. Du trenger ikke et separat assemblerprogram for å lage dette programmet. Øvingen til dette kapitlet vil vise deg hvordan du kan lage programmer med «debug».

Prosessorer fra ulike produsenter har ganske ulike instruksjoner og adresseringstyper. Det er derfor ikke mulig å flytte et program med maskininstruksjoner mellom ulike prosessortyper, f.eks. mellom en Macintosh (som bruker prosessor fra Motorola) eller en Windows pc (som bruker prosessor fra Intel).

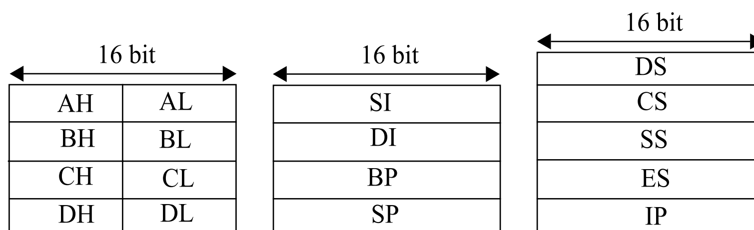
## CPU-ens interne organisering

Dersom du satt inne i CPU-en, ville du hatt denne utsikten til omverdenen:

- En serie minneceller. Hver av disse minnecellene har en adresse (plassnummer) og et innhold. Dette innholdet kunne du lese og endre
- En serie I/O-celler. Disse cellene har også et plassnummer, men er ikke egnet for datalagring. Betjeningen av maskinvarekomponentene for input/output av data skjer gjennom disse cellene. Noen av cellene bare leses, noen kan endres, og noen kan bare skrives til.

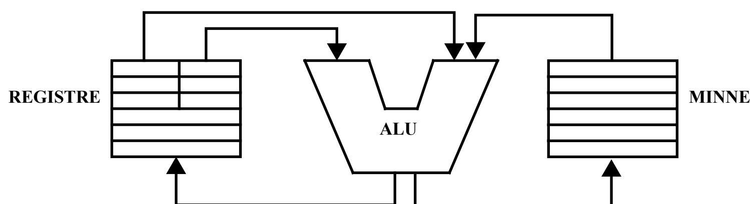
Internt i CPU-en ville du hatt disse elementene til bruk:

- Et lite antall (10–50) registre. Registre er som minneplasser, de kan leses og skrives. Noen registre har spesielle funksjoner, f.eks. har ett av dem oppgavene med å peke til den minnecellen som inneholder den neste instruksjonen (instruksjonspekeren).




*Figur 2-2: Registrene inne i en 8086/8088 Intel CPU. Registrene har ulike bruksområder, og alle kan ikke brukes til beregninger*

- Et lite antall indikatorer (flagg). Disse brukes under testing, f.eks. om to tall er like, og under forskjellige regneoperasjoner.
- Aritmetisk og logisk regneenhet (ALU). Her skjer alle regneoperasjoner (addisjon m.m.), logiske operasjoner (and, or, shift) og tester. Mange operasjoner i ALU bruker registrene som operander.
- Instruksjonsdekoderen. Dette er en del av det elektroniske kretsløpet i CPU-en som har til oppgave å analysere instruksjoner og kontrollere utføringen av dem ved bl.a. å bruke funksjonene i ALU.




*Figur 2-3: Operasjoner i ALU tar operander fra registre eller en minnecelle, og legger resultatet inn i et register eller i en minnecelle*

## Pipelining – overlappet instruksjonsutføring

Inne i prosessorens kretsløp ligger funksjonene for henting og utføring av instruksjoner i separate deler. Det er derfor mulig å øke ytelse på prosessoren ved å la de to delene arbeide parallelt.

I begrepet *pipelining*[2] legger vi at prosessoren henter påfølgende instruksjoner mens den ennå holder på med utføring av den pågående instruksjonen. Når utføringen av denne instruksjonen er ferdig, ligger neste instruksjon allerede på plass inne i prosessorens kretser, klar for utføring. Dette sparer tid i forhold til om prosessoren måtte stoppe opp for å hente instruksjoner fra minnet.

Pipelining «spekulerer» i det forholdet at instruksjonene oftest utføres i rekkefølge. I de tilfeller en instruksjon medfører et hopp i instruksjons-rekkefølgen må de innsamlede instruksjonene kastes, og innhentingen starte på nytt fra det nye stedet. Men siden dette er unntaket, er netto-effekten av pipelining at prosessoren arbeider raskere.

En avansert form for pipelining kalles for *branch prediction*, hvor pipeline-logikken kan huske hvor det sist skjedde et hopp i instruksjon-srekkefølgen, og beregne sannsynligheten for at et nytt hopp vil skje. Gjetter den riktig blir ikke pipeliningen avbrutt, og ytelsen blir bedre. Branch prediction benyttes i Pentium og er effektiv ved repeterte hopp i løkker o.l..

Avanserte prosessorer forsøker å parallellisere mest mulig av sin interne arbeidsgang, og en overlapp mellom instruksjonsdekoding og -utføring er vanlig. I noen tilfeller finner vi også overlapp i selve utføringen av instruksjoner, hvor f.eks. en test og en beregning kan foregå samtidig i forskjellige deler av prosessorens kretsløp.

---

2. Uttrykket «instruction prefetching» betegner det samme som pipelining.

# Adresserbare celler

## Minneceller

Minnet er den del av maskinvaren hvor dataene blir lagret. De er lagret slik at det er raskt å få tak i dem, men dataene forsvinner når maskinen slås av (flyktig minne). I en vanlig pc er minnet organisert i form av en rekke minneceller, som hver kan romme en verdi med 8 bits (binære sifre).

Enhver minnecelle har sin egen adresse i form av et tall. Vi liker å tenke på minneceller som liggende på en rad med påfølgende adresseverdier. Da kan vi avgjøre om to minneceller er «naboer» eller om en mengde minneceller utgjør en «kontinuerlig serie».

Minnecellene er samlet i *minnebrikker*, som er montert på datamaskinens hovedkort. En minnebrikke (monteres ofte som et sett av brikker) kan inneholde flere millioner minneceller.

## i/o-celler

En CPU kan ikke kommunisere med omverdenen (motta eller sende data) på annen måte enn gjennom å skrive og lese minneceller. Derfor blir maskinvarekomponenter for i/o laget slik at deres grensesnitt (kontaktflate) mot CPU-en *ser ut som* minneceller. I noen typer av maskiner (f.eks. en Intel-pc) holdes i/o-celler og minneceller fra hverandre i separate adresserom (slik at en minnecelle og en i/o-celle kan ha samme adresseverdi), i andre typer deler de adresserom (Motorola).

## Adresseres likt

I begge tilfeller leses og skrives minneceller ved å overføre dataverdiene mellom cellen og et CPU-register. I begge tilfeller plukkes en celle ut på basis av sin adresse.

I en Intel-pc brukes ulike instruksjoner for å adressere minneceller og i/o-celler. Bortsett fra dette er det sterke likheter mellom hvordan data leses og skrives i de to typene.

## Minnet lagrer, i/o knytter til omverdenen

Selv om det er stor likhet på adresseringen av minneceller og i/o-celler, er det vesentlig forskjell på bruken av dem:

- Minneceller brukes til alminnelig lagring av data

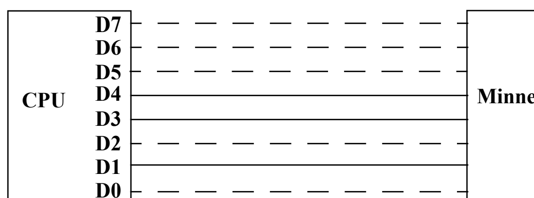
- i/o-celler brukes til kontroll av maskinvaren, og til å sende og motta data gjennom i/o-kretsene i maskinen. For å få dette til er det nødvendig å ha god kjennskap til den detaljerte virkemåten for en i/o-krets. Det er bl.a. denne kunnskapen vi ønsker å gjemme inne i operativsystemet for at programmeringen skal bli enklere

## Bussene

Bussene er knipper av elektriske ledere mellom brikkene i en datamaskin, som vist på figur 2-1. Denne figuren viser to busser, kalt *adressebussen* og *databussen.* I tillegg har vi noen ledere for spesielle signaler (kalt *kontrollbussen*). Vi forklarer bussenes oppgave og virkemåte slik:

### Databussen

har minst 8 ledere, og brukes til å frakte data mellom minneceller (ev. i/o-celler) og CPU. Hver av lederne representerer en bit av den verdien som skal overføres ved at den enten fører en elektrisk spenning (typisk 5 volt) eller ikke. Om binærverdien som skal overføres er f.eks. 00011010, vil databussen være i denne tilstanden under overføring:




*Figur 2-4: Tilstanden på databussen ved overføring av verdien 00011010. Stiplet linje forestiller 0 volt spenning, heltrukken linje forestiller 5 volt*

Vi ser av figur 2-4 at databussens ledninger er kalt D0–D7. Spenningen på D0 representerer bitverdien «lengst til høyre» (minst *signifikante* bit), mens D7 representerer den mest signifikante bit (lengst til venstre).

Som vist på figur 2-1 er «alle» brikkene koplet til databussen. Det er kun én av gangen som kan «snakke» på databussen, dvs. legge en elektrisk spenning ut på lederne. På figur 2-4 vil det være minnebrikken som legger ut denne spenningen dersom CPU-en vil *lese* innholdet av

en minnecelle. Om CPU-en vil *skrive* en verdi til en minnecelle er det CPU-en som legger ut spenningen på databussen. Alle de andre brikkene som er koplet til databussen må kun «lytte» på bussen.

Men hvordan vet en minnebrikke (eller en i/o-brikke) når den skal snakke og når den skal lytte? Og hvordan skal minnebrikken sørge for at det er den riktige minnecellen som blir brukt? Dette er et spørsmål som løses ved å lytte på *adressebussen*.

## Adressebussen

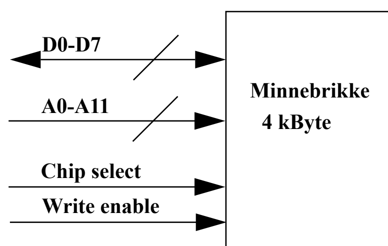
er et sett av ledninger mellom brikkene i en datamaskin som CPU-en benytter for å fortelle hvilken celle som skal leses eller endres. Forut for hver eneste overføring av verdier på databussen legger CPU-en cellens adresse på adressebussen i form av elektriske spenninger.

Adressebussen er «enveis» i den forstand at det alltid er CPU-en som snakker, og alle andre lytter (vi skal modifisere denne påstanden litt senere i kapitlet, men tro at det er sant inntil videre).

Alle brikkene som er koplet til databussen har et kontaktben (eng.: pin) som brukes til å fortelle at brikken skal «våkne opp» og lytte på adressebussen for å ta del i en overføring på databussen. Dette benet kalles ofte «chip select».

Et annet ben på brikken brukes til å fortelle hvilken vei databussen skal overføre en verdi, altså om cellen skal leses eller endres. Benet kalles ofte «write enable».

CPU-en ser minnecellene som en rad med nummererte celler, og har ingen kunnskap om hvilken minnebrikke de ligger i. Når vi konstruerer en datamaskin må vi derfor lage dekodingskretser som kopler inn den riktige minnebrikken til en gitt adresse.




*Figur 2-5: En minnebrikke er koplet til både adresse- og databussen, og noen kontroll-linjer i tillegg. Legg merke til bruken av skrålinjen for å vise et knippe av ledere*

## Kontrollbussen

I tillegg til den informasjonen som databussen og adressebussen overfører, er det behov for en del «løse ledninger» mellom CPU-en og den øvrige maskinvaren. Disse ledningene er samlet i *kontrollbussen*. Hvilke funksjoner er representert i kontrollbussens ledere? Dette varierer noe mellom forskjellige typer CPU-er og maskinvaredesign, men de vanligste er:

- Synkronisering av CPU og minneceller (ev. i/o-celler) under lese- og skriveoperasjoner (Adress Latch Enable, Data Transfer Acknowledge).
- Angivelse av om cellene skal leses eller skrives (Read/Write, koplet til «Write enable»-benet på minnebrikkene).
- Signal fra en i/o-enhet om at «noe» er skjedd (Interrupt, se side 39).
- Synkronisering mellom CPU-en og DMA-brikken (side 42).
- Synkronisering mellom CPU-ene i en multiprosessormaskin, slik at det ikke oppstår konflikt i bruken av bussene (side 36).

(Her brukte vi en del begreper som vi ikke har forklart ennå, men som blir omtalt senere i kapitlet.)

Kontrollbussen er altså toveis, men ikke på samme måte som databussen. Noen linjer fører signaler fra CPU-en til resten av maskinvaren, andre linjer fører signaler motsatt vei. Toveis signalering på samme linje er uvanlig.

## En buss-syklus

Hvordan skjer så samspillet mellom databussen og adressebussen når en adresserbar celle leses eller skrives?

**Adressebussen starter alltid** En buss-syklus (eng. bus cycle) starter alltid med at CPU-en legger den ønskede minnecellens (ev. i/o-cellens) adresse som signaler på adressebussen. Som alle slags elektriske signaler tar det litt tid før den elektriske spenningen er stabil over hele adressebussen.

**Kontrollbussen gir tilleggsinformasjon** Samtidig som CPU-en legger signaler på adressebussen vil den også angi om cellen skal leses eller skrives. Dette foregår på kontrollbussens «Read/Write»-linje, hvor f.eks. 5 volt betyr «Write» og 0 volt betyr «Read». Et annet signal som også sendes på kontrollbussen er en beskjed om at signalet på adressebussen nå er «gyldig». Dette signalet (ofte kalt *Address Latch Enable, ALE*) sendes like etter signalene på adressebussen, slik at adressesignalene skal ha stabilisert seg først. Signalet forteller bl.a.

kretsene som driver med adressedekoding[3] og som har ansvaret for å skru på «chip select»-signalet for den riktige minnebrikken, at adressesignalene nå er gyldige og stabile.

**Databussen blir aktiv** En liten stund etter at adressebussen er stabil, skal signalet på databussen også være stabilt.

- Om det er en skriveoperasjon vil CPU-en legge signaler på databussen omtrent samtidig som den legger signaler på adressebussen, og holde signalet aktivt i en viss tidsperiode
- Om det er en leseoperasjon, vil minnekretsen (ev. i/o-kretsen) reagere på «chip-select»-signalet og signalene på adressebussen ved å legge innholdet av én av minnecellene som signaler på databussen. Dette må skje innen en tidsfrist, for CPU-en vil snart avlese signalene og bruke verdiene i en eller annen beregning (eller som en instruksjon).

**Bussene frigjøres** Når operasjonen er fullført, vil ALE-signalet opphøre, deretter signalene på adressebussen og databussen. Etter en kort tidsperiode med «utladning» av de elektriske spenningene er bussene klare til å brukes i en ny operasjon.

**En variant: Read/Modify/Write** En variant av en buss-syklus er en «Read/Modify/Write»-syklus. I en slik operasjon vil ALE-signalet og signalene på adressebussen holde seg så lenge at CPU-en kan lese en minnecelle og skrive et innhold tilbake til den samme cellen (kun Read/Write-signalet vil endre seg). En slik variant er nyttig i multiprosessormaskiner, men ikke alle CPU-typer bruker den.

> **Viktig:** All dataoverføring mellom CPU-en og den øvrige maskinvaren foregår i form av buss-sykler, også innlesing av instruksjoner.

## Sammenheng mellom adresselinjer og minnestørrelse

En minnebrikke har et visst antall minneceller, og trenger informasjon fra adressebussen for å adresse hver enkelt celle. Vi skal se at en minnebrikke bare lytter på så mange adresselinjer som er nødvendig for å adressere cellene inne i brikken. Det forutsettes at du er noenlunde fortrolig med det binære tallsystemet.

---

3. Med adressedekoding mener vi de logiske kretsene i maskinen som er koplet til deler av adressebussen, og som sørger for å «skru på» den minnebrikken som har den aktuelle adressen. Utformingen av adressedekodingen er en del av maskinvaredesignet.

Et binært tall med *n* sifre kan ha $2^n$ forskjellige verdier. F.eks. kan et tall med to sifre ha de fire verdiene 00, 01, 10, 11 fordi $2^2 = 4$. For hver gang du øker tallet med ett siffer dobles det mulige verdiområdet (og omvendt).

Du bør gjøre deg fortrolig med 2-potensene og lære disse tallene utenat:

$2^4 = 16$ (et heksadesimalt siffer viser 4 binære siffer og har 16 mulige verdier)

$2^8 = 256$ (en byte har 8 bits og har 256 mulige verdier)

$2^{10} = 1024$ (også kalt «kilo» i datasjangeren)

$2^{16} = 65536$ (64k)

$2^{20}$ = «1 million» (1024x1024 = 1M)

En minnebrikke med 256k minneceller vil derfor være koplet til 18 ledere på adressebussen, fordi $2^{18}$ = 256k. I slike tilfeller er det linjene A0-A17 som brukes. Linjene A18 og oppover koples til dekodingskretsene som lager et signal for «chip select»-benet ved den riktige adressekombinasjonen.

**CPU-ens adresseområde** Størrelsen på adressebussen (antall linjer) vil derfor bestemme hvor mange minneceller som CPU-en kan adressere. Størrelsen på adresseområdet er en egenskap ved en CPU-type og er uavhengig av hvor mye minne som faktisk er installert i maskinen. Det er heller ikke alltid praktisk mulig å installere så mye minne i en maskin som CPU-ens adresseområde skulle tilsi. Instruksjonene i CPU-en er oftest tilpasset adresseområdet slik at antall bits i en celleadresse er det samme som antall linjer på adressebussen.

Den vanligste størrelsen på en minnecelle er 1 Byte, derfor angis adresseområdets størrelse oftest i antall Byte (kByte, MByte eller GByte[4]).

Her er noen eksempler på CPU-typer og deres adresseområder:

| CPU-type | Størrelse på adresseområdet |
|---|---|
| 6502 (Acorn BBC, Commodore 64) | 64 kByte (16 bits adresser) |
| 8088 (Original IBM PC) | 1 MByte (20 bits adresser) |

4. GByte betegner 1 «milliard» byte (egentlig 1024*1024*1024 bytes).

| CPU-type | Størrelse på adresseområdet |
|---|---|
| 80286 (IBM AT) | 16 MByte (24 bits adresser) |
| 80386, 80486, Pentium (moderne pc-er) | 4 GByte (32 bits adresser) |

## Bussenes hastighet

Operasjonstakten på en buss er begrenset av de elektriske egenskapene vi finner i maskinen. Det er derfor slik at en buss-syklus (vise adresse på adressebussen, så overføre data på databussen) tar et minimum av tid (typisk 10–100 nanosekunder på en moderne pc). Moderne CPU-er kan arbeide mye raskere enn dette, og i mange tilfeller blir bussene en flaskehals. Hvordan synkroniseres CPU-en med busshastighetene? Her finnes to alternativer.

**Asynkron buss** Hver gang CPU-en starter en byss-syklus kan den adresserte minnekretsen (ev. i/o-kretsen) kvittere med et signal på kontrollbussen kalt *Data Transfer Acknowledge* (DTA) som forteller at kretsen er «ferdig», dvs. har mottatt eller sendt signaler på databussen. En asynkron buss kan derfor koples til mange slags kretser med ulik responstid, og dermed være mer fleksibel. Mange kretser er derimot ikke konstruert for å avgi det nødvendige DTA-signalet, som da må genereres av separate kretser.

**Synkron buss** I en synkron buss forventes det at alle de tilkoplede kretsene er «ferdige» etter en forutbestemt tid. Klarer de ikke det, kan de ikke tilkoples bussen. Designet av en synkronbuss er dermed enklere, men mindre fleksibel. En synkron buss har ikke et DTA-signal i kontrollbussen.

Maskinfabrikanten forsøker å bruke forskjellige teknikker for å bli mindre avhengig av buss-hastigheten. En vanlig teknikk er å plassere mye brukte data i minneceller som ligger mellom CPU-brikken og internminnet. Slike minneceller kaller vi for «cache». En annen teknikk er å ha flere busser med forskjellige hastigheter i samme maskin.

## Bruk av caching

Vi har hittil nevnt at data kan lagres i CPU-registrene (rask tilgang, men lite plass) eller i minneceller (mer plass, men langsom tilgang). Vi har i tillegg til dette i alle moderne maskiner et middels stort minneområde mellom disse to som vi kaller for en «cache». En cache har disse egenskapene:

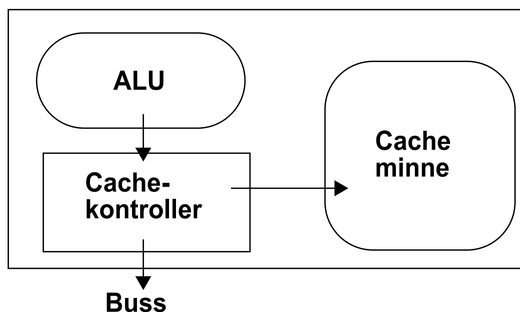
- Den er raskere å bruke enn minneceller fordi den unngår hastighetsbegrensningene forbundet med en buss-syklus.

- Bruken av en cache er «usynlig» (transparent) for programkoden. Innholdet i en cache ser ut som vanlige minneceller, og bruken av cache-minnet styres helt og holdent i maskinvarekretsene inne i CPU-en.

**Viktig.** Cache er minneceller, men er ikke *adresserbare*. De er derfor «usynlige» for vanlige maskininstruksjoner

**Hyppigst brukte data** En cache er konstruert slik at den rommer innholdet av de minnecellene som er hyppigst brukt.

- Ved lesing av en minnecelle kontrollerer cache-kontrolleren om innholdet av cellen allerede ligger i cache-minnet. Dersom den gjør det, hentes innholdet fra cache.
- I motsatt fall hentes innholdet fra minnecellen gjennom en vanlig buss-syklus, og innholdet blir i tillegg lagret på en ledig plass i cache-minnet.
- Når cache-minnet er fullt, skapes det plass for nytt innhold ved at det innholdet som har vært minst brukt ,fjernes.




*Figur 2-6: Bruk av cache-minne i en CPU*

**Speilbilde av minneceller** Innholdet i cache er ikke selvstendige data, men kopier av innhold i minneceller. Ved skriving til en minnecelle som har sitt innhold i cache vil innholdet av *både cache og minnecellen* endres. For å spare tid på å vente på buss-sykler kan skriving til minne-

cellen utsettes til senere, f.eks. når innholdet i cache må vike plass for nyere data. En cache som kopierer endringer videre til minneceller på denne måten kalles *write-through cache*.

**Hvordan finner vi de minst brukte dataene?** I klesskapet mitt henger alle klærne på en stang. Hver gang jeg har brukt noen klær henger jeg dem tilbake på høyre side av skapet. Når jeg skal levere klær til loppemarkedet, trenger jeg å vite hvilke klær som jeg bruker minst. *Hvor finner jeg disse klærne? Jo, på venstre side!* En algoritme for å finne et dataelement som kan fjernes fra en cache kan bruke denne algoritmen (kalt *Fongens klesskap*).

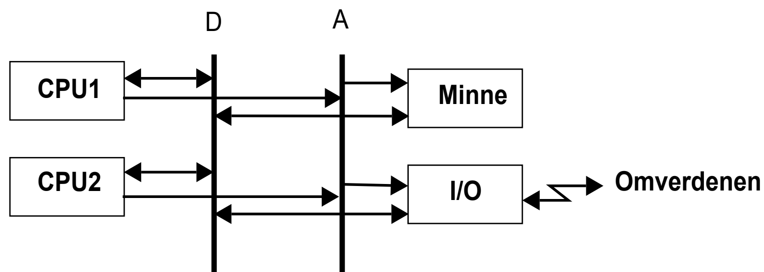
For ytterligere å nyansere bruk av cache opererer avanserte prosessorer (Pentium II og høyere) med bruk av to-nivå cacher. *Level I cache* sitter inne i selve CPU-brikken og er meget hurtig i bruk. *Level II cache* sitter i en ekstern brikke, men kommuniserer med CPU-brikken gjennom en separat buss som er raskere enn den ordinære bussen. Level II har mer plass enn Level I.

Tidlige modeller av Pentium-prosessoren hadde 16 kByte Level I cache, delt opp i to 8 kByte-deler reservert for henholdsvis instruksjoner og data. Senere modeller av prosessoren har mange ulike størrelser av cachene, og noen har også Level II cache montert på samme brikke som CPU-en.

## Multiprosessorer

Vi kan i noen tilfeller øke behandlingskapasiteten i en datamaskin ved å la flere CPU-er arbeide «i spann». Datamaskiner med flere CPU-er inni kalles for *multiprosessorer*. Vi sier «i noen tilfeller» fordi det forutsetter at oppgavene som skal løses er egnet for bli delt opp i separate deloppgaver.

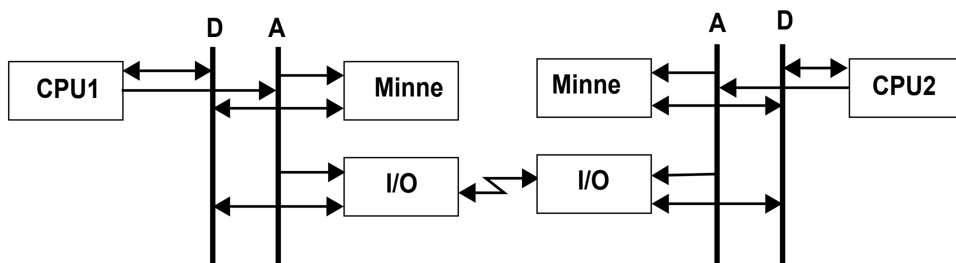
Når vi konstruerer datamaskiner med flere CPU-er må vi velge om de skal være koplet til samme data- og adressebuss, eller om de skal ha hver sine busser. De to alternativene kan vi illustrere slik:




*Figur 2-7: Tett koplet multiprosessor*

Figur 2-7 viser en multiprosessormaskin hvor CPU-ene er koplet til samme adresse- og databuss. Vi kaller en dette en *tett koplet* multiprosessor. Dette innebærer at maskinen får disse egenskapene:

- CPU-ene må dele på overføringskapasiteten til bussene, f.eks. ved å bruke bussen etter tur. Dette setter en praktisk grense for antall CPU-er i en slik konfigurasjon. Over et visst antall (anslått til 30) CPU-er blir bussene en flaskehals i arbeidsgangen.
- CPU-ene vil «se» de samme minne- og i/o-cellene på de samme adressene. Dette innebærer at programmer som utføres på CPU-ene kan samarbeide tett gjennom *felles dataområder*.




*Figur 2-8: Løst koplet multiprosessor*

Den alternative konfigurasjonen er vist på figur 2-8 og kalles *løst koplet* multiprosessor. I dette tilfellet består datamaskinen av CPU-er som har sine egne busser, sitt eget minne og i/o-enheter. Hver CPU danner sin selvstendige, komplette datamaskin som bare er knyttet sammen gjennom en kommunikasjonskanal (f.eks. et lokalnett). En slik konfigurasjon innebærer disse egenskapene:

- Bussene er separate, derfor kan denne konfigurasjonen ha et stort antall CPU-er.

- Deling av data mellom programmene som kjører på forskjellige CPU-er skjer ved kopiering via i/o-kanalen. Dvs. de arbeider ikke på samme «eksemplar» av dataene, men må ha hver sin kopi.

### Tett eller løst koplet?

Begrepene *tett* og *løst* koplet multiprosessor ble forklart i forrige avsnitt. I operativsystemfaget er det først og fremst aspektet knyttet til *deling av data* som utgjør den viktigste skillelinjen mellom de to alternative konfigurasjonene. Vi skal komme tilbake til dette mange ganger i boka, og vil se at både de teoretiske problemstillingene og programmeringsteknikkene vil reflektere denne skillelinjen.

Når du ser datamaskiner som kaller seg «multi-CPU», «dual Pentium» o.l. dreier det seg oftest om tett koplede multiprosessorer. Løst koplede multiprosessorer finner vi sjeldnere i kommersielle produkter. Vi kan realisere en løst koplet multiprosessor ved å utstyre standard-pc-er med egnet programvare og la dem kommunisere gjennom et lokalnett.

Storskala-multiprosessorer utviklet i forskningsøyemed benytter begge konfigurasjonene i kombinasjon, ved at grupper av tett koplede CPU-er er løst koplet gjennom en i/o-kanal. I slike eksperimenter finner vi også segmenterte i/o-kanaler for å unngå at kanalen blir en flaskehals.

### Når trenger vi multiprosessorer?

Som vi nevnte i starten av dette avsnittet, så er det ingen selvfølge at en multiprosessor vil øke behandlingshastigheten for en gitt anvendelse. En multiprosessor er til nytte der det er mulig å dele opp oppgaven i relativt uavhengige deler. Her er noen eksempler på slike oppgaver:

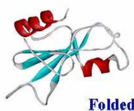
- Numerisk løsning av en likning. Det mulige området av parameterverdier blir delt mellom CPU-ene, som leter etter løsninger og rapporterer dem uavhengig av hverandre.
- Søking etter tekstforekomster i store dokumentsamlinger. Dokumentsamlingen deles mellom CPU-ene, som søker etter tekstforekomster uavhengig av hverandre.
- Dataflyt-orienterte anvendelser. Én CPU har f.eks. ansvar for innsamling av data, en annen filtrering og «vasking» av data, en tredje for aggregering og rapportering. Data som «produseres» av en CPU «konsumeres» av den neste i en kjede av prosessorer. Slik sammenstilling av prosessorer kalles «pipelining[5]».

---

5. Må ikke forveksles med betydningen av ordet i avsnittet “Pipelining – overlappet instruksjonsutføring” på side 27.

I motsetning til disse typer anvendelser er vanlige «personlige» anvendelser som tekstbehandling, tegning eller kompilering lite egnet til å utnytte kapasiteten i multiprosessorer.

### Blue Gene




Verdens kraftigste multiprosessormaskin under konstruksjon er IBMs Blue Gene. Maskinen konstrueres delvis for å forske på parallellarkitektur, men også for å skaffe regnekraft til å studere et biomolekylært problem kalt *protein folding*. Simulering av molekylære bevegelser av denne typen krever $10^{23}$ instruksjoner for å simulere 100 mikrosekunder av virkeligheten.

Blue Gene/L (som prosjektet heter for øyeblikket) vil få en ytelse på 200 teraflops/sekund (tera = $10^{12}$, flops = flyttallsberegninger). Maskinen vil være en blanding av tett og løst koplede maskiner. En «node» bestående av 30 CPU-er med samlet beregningskraft på 32 gigaflops/sekund, 8 MByte minne og kommunikasjonskanal på 12 gigabytes/sekund vil bli laget på én enkelt chip. En slik node er en tett koplet multiprosessor. Nodene blir pakket sammen i «kuber» i antallet 32x32x32 og samarbeider gjennom kommunikasjonskanalen. Nodene er dermed løst koplet til hverandre. Kubene blir så bygd sammen i enda større enheter. Totalt blir systemet på 30000 noder (1 mill. CPU-er), og vil kreve et areal på størrelse med en håndballbane. Konstruktørene beregner et strømforbruk på 2MW for et system med 1 petaflops/sekund ytelse (peta = $10^{15}$).

Blue Gene/L er planlagt ferdigstilt i 2004.

## Avbrudd

Vi har tidligere sagt av CPU-en er sjefen i maskinen og bestemmer hva som skal skje til enhver tid. Vel, her kommer en nyansering av den påstanden. En i/o-krets kan fortelle CPU-en at den har noe «på hjertet». Ved hjelp av et elektrisk signal gjennom en separat ledning kan en slik beskjed formidles fra i/o-kretsen uavhenging av hva CPU-en holder på med. Denne ledningen er ofte kalt «interrupt» (INT) eller «interrupt request» (IRQ), og forbinder et ben på CPU-brikken til et ben på en i/o-brikke[6].

Hvorfor er det nødvendig å sende slike beskjeder til CPU-en? Jo, i tilfeller der CPU-en trenger å varsles om hendelser som er uavhengig av CPU-ens instruksjonsrekkefølge. Slike hendelser kan være:

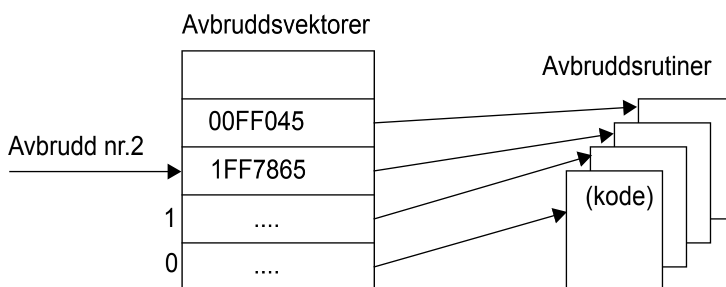
6. Der mange i/o-brikker vil forbinde sin avbruddslinje til CPU-brikken, velges en egnet kretsløsning for dette.

- Data ankommer fra et nettverk, sendt av en annen maskin med helt uavhengig programutførelse.
- En temperaturmåler i en kjele viser for høy temperatur.
- En bruker trykker en tast på tastaturet.

Slike hendelser kalles *asynkrone* fordi de oppstår på helt uavhengige tidspunkter. I et datamaskin uten bruk av avbrudd kan vi la CPU-en til stadighet «spørre» i/o-enheten om en hendelse er inntruffet, men det er en uøkonomisk bruk av CPU-ressurser.

**Avbruddsrutine** Når CPU-en mottar et avbruddssignal vil den endre instruksjonsrekkefølgen. Den vil fullføre den instruksjonen som den holder på med for deretter å hente neste instruksjon fra en annen, nærmere angitt adresse. På denne adressen har operativsystemet programinstruksjoner liggende for å behandle den hendelsen som avbruddet varsler om. Vi kaller denne programkoden for en avbruddsrutine (eng.: Interrupt Service Routine, ISR).

**Avbruddsvektor** Avbrudd kan komme fra mange kilder, og det er nødvendig at CPU-en klarer å skille mellom dem. Signaler fra forskjellige avbruddskilder må selvsagt behandles av forskjellige avbruddsrutiner. Avbruddskildene blir tildelt et nummer og en separat ledning for å sende sitt avbruddssignal gjennom. Maskinvaren klarer å skille mellom signalene og hente adressen til riktig avbruddsrutine fra en tabell av adresser som kalles *avbruddsvektorer*, se figur 2-9.




*Figur 2-9: Eksempel på avbruddsvektorer. Når avbrudd nr. 2 oppstår, finner CPU-en adressen til vedkommende avbruddsrutine i element nr. 2 i tabellen med avbruddsvektorer, i dette tilfellet 1FF7865. Tabellen med avbruddsvektorer ligger alltid på et kjent sted i minnet.*

## Telefonen ringer ...

Om du ikke hadde en ringelyd på telefonen din, ville du vært nødt til å løfte på røret med korte mellomrom for å høre etter om noen forsøkte å ringe deg. Det ville vært komisk og unødvendig å holde på sånn. Det er åpenbart at en ringelyd kan bidra til at du løfter av røret bare når det virkelig er noen som ringer.

Slik er det også i konstruksjonen av et operativsystem. Vi utnytter kapasiteten mye bedre i et system som tillater at omgivelsene varsler om oppståtte behandlingsbehov.

## ... mens du leser en bok

Når telefonen ringer mens du holder på med et arbeid, f.eks. mens du leser en bok, så tar du telefonen og gjennomfører en samtale. Etterpå fortsetter du i boka der du ble avbrutt. Det ville vært ganske håpløst om du skulle starte forfra i boka igjen.

Derfor er det viktig ved bruk av avbrudd at vi klarer å huske «tilstanden» vår når avbruddet oppstår, slik at vi kan gjenoppta aktiviteten på rett sted etterpå.

Vi skal omtale programmeringsteknikker for avbruddsbehandling senere i boka, men i denne omgang er det viktig at du har dette klart for deg:

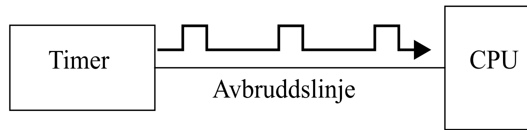
**Avbrudd er et elektrisk signal som går fra en i/o-krets til CPU-en gjennom en elektrisk ledning. Denne ledningen kalles avbruddslinjen og brukes ikke til noe annet formål. Avbruddslinjen er en del av kontrollbussen.**

## Timere

Vi bruker avbrudd ikke bare i forbindelse med i/o. De fleste datamaskiner trenger å holde rede på hva klokken er, og må kunne måle tidsforløp. Dette klarer ikke CPU-en uten hjelp utenfra. En maskinvarebrikke som kalles en *timer* gir denne hjelpen i form av avbruddssignaler med jevne mellomrom.

En timer har det til felles med en i/o-krets at den betjenes gjennom adresserbare celler. Betjeningen kan bestå i å be om at den sender et avbruddssignal f.eks. 50 ganger/sekund. Vi skal senere se at et operativsystem er helt avhengig av timer-avbrudd for å fungere skikkelig.

Timeren i en pc sender avbruddssignal 18.2 ganger/sekund. Dette gir 65536 ($2^{16}$) avbrudd pr. time.

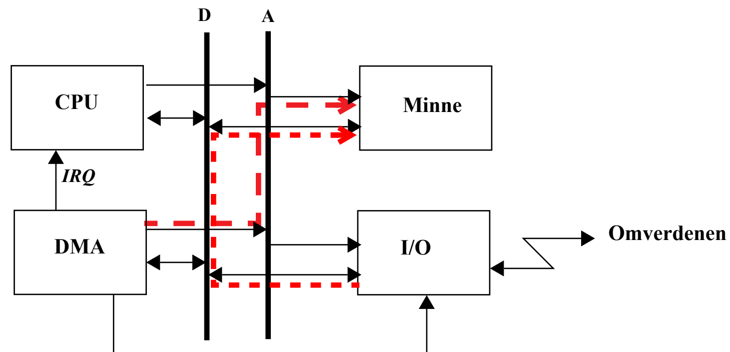



*Figur 2-10: En timer-krets som sender regelmessige avbrudd til CPU-en. Timeren vil også være koplet til data- og adressebussen for at kretsen skal kunne settes opp av programvaren, men dette er ikke vist på figuren.*

## Direct Memory Access

I de tilfellene hvor store datavolumer skal flyttes på kort tid er det lite hensiktsmessig å instruere CPU-en til å flytte byte for byte fra i/o-kretsene til internminnet. CPU-en vil måtte bruke mye tid på å dekode instruksjoner, og kapasiteten i bussene blir ikke godt utnyttet.

Vi vil heller la denne oppgaven bli utført av spesiallaget maskinvare som gjør dette mer effektivt. Denne maskinvarekretsen kalles *Direct Memory Access* (DMA) og brukes f.eks. i forbindelse med i/o mot disker.




*Figur 2-11: Signalveien for adresseinformasjon og data ved DMA-overføring*

### Får være "sjef" en liten stund

Vi har tidligere nevnt at kun CPU-en kan sende data på adressebussen. Unntaket er DMA-brikken, som også kan gjøre dette. Overføring av en databyte fra en i/o-krets til en minnecelle er vist på figur 2-11 og skjer på denne måten:

- DMA-brikken legger minnecellens adresse på adressebussen, og skrur på «write enable»-signalet.
- DMA-brikken sender et signal til i/o-brikken om å legge en databyte på databussen. i/o-kretsen legger signaler på databussen.
- Minnecellen har nå mottatt signal både på adressebussen og databussen, og lagrer databyten i minnecellen.

Når en byte er ferdig overført, står den neste for tur, og DMA-brikken vil repetere operasjonene med en annen adresseverdi, slik at bytene blir liggende i en rad i påfølgende minneceller.

DMA kan brukes for både inn-data som i tilfellet over, og for ut-data, hvor «write enable»-signalet ikke blir slått på.

En DMA-brikke kan ikke legge et signal på adressebussen i konflikt med CPU-en. For å unngå konflikt må en DMA-overføring skje på én av disse måtene:

- DMA-brikken sender et separat signal til CPU-en som medfører at den stopper opp. Så kan DMA-overføringen skje og signalet skrus av etterpå. Slik DMA-overføring sies å foregå i *burst mode.*
- DMA-brikken overfører bare data mens CPU-en ikke trenger bussen. CPU-brikken har et ben som gir signal om at den ikke trenger bussen et øyeblikk (det skjer under beregninger og instruksjonsdekoding). I slike perioder kan DMA-overføring foregå, men må stoppe og vente straks dette signalet skrus av. Slik DMA-overføring sier å foregå med *cycle stealing.*

## Sammendrag

- CPU-brikken er sjefen i datamaskinen. Den henter instruksjoner fra minnet og utfører dem.
- CPU-instruksjoner kan innebære at data blir lest eller endret i minneceller eller i/o-celler.
- Adressebussen og databussen er veinettet hvor overføring av signaler til og fra CPU-en foregår. Kontrollbussen overfører signaler for synkronisering og kontroll.
- Minnebrikker og i/o-brikker lytter på adressebussen for å finne ut hvilken celle som skal leses eller endres. Databussen brukes til å overføre verdien.
- Multiprosessorer skilles i *tett* og *løst* koplede konfigurasjoner.
- Avbrudd er en måte for i/o-kretser å varsle om hendelser til CPU-en på.

- DMA overfører data mellom minne og i/o-kretser raskere enn CPU-en klarer. Vi bruker DMA ved overføring til og fra disk.

| Sentrale begreper i dette kapitlet | |
|---|---|
| CPU-brikke, CPU-instruksjoner | Minneceller, i/o-celler |
| Adressebuss, databuss | Avbrudd, avbruddsvektor |
| Direct Memory Access | Cache, level I og II |
| Timer | Multiprosessor |
| Overlappet utføring | Buss-syklus |

# Teorioppgaver

**Gå sammen i grupper og finn løsning på disse spørsmålene:**

1 Beskriv hvilke maskinvarekomponenter som inngår i datamaskinen som

- er innebygd i en mobiltelefon
- er innebygd i en videospiller
- står som filtjener i et nettverk
- er din egen personlige datamaskin

2 Tegn et skjema som viser en tett koplet multiprosessor med to CPU-er og to i/o-brikker. Tegn inn hvordan avbruddslinjene fra de to i/o-brikkene skal koples til CPU-ene og vurder disse to alternativene:

- Begge avbruddslinjene koples sammen og til begge CPU-ene.
- De to avbruddslinjene føres til hver sin CPU.

3 Hvor mange adresselinjer har en minnebrikke med dette antallet minneceller:

- 2048?
- 131072?
- 2?

4 Nevn 5 forskjellige typer maskinvarekomponenter av typen i/o-kretser.

5 En datamaskin trenger et oppstartsprogram som skal utføres i det strømmen slås på. I hva slags maskinvarekomponent ligger oppstartsprogrammet? Begrunn svaret.

# Øvingsoppgaver

Et forslag til øvingsopplegg for dette kapitlet ligger på bokas nettsted. Etter øvingsopplegget bør du ha disse ferdighetene:

6 Være kjent med hvilke hjelpeprogrammer som operativsystemet har for å kartlegge maskinvareressursene i maskinen

7 Kunne bestemme CPU-type og minnestørrelsen i maskinen

8 Kunne avgjøre hvilke avbruddslinjer som er i bruk

9 Være kjent med BIOS' konfigurasjonsmeny i en pc

10 Gjenkjenne hovedkomponentene inne i datamaskinen (CPU, busser, skjermkontroller, diskkontroller, minnebrikker etc.

11 Vite funksjonene til portene på baksiden av kabinettet

12 Kunne demontere og montere tilleggskort og minnebrikker

Kapittel 3

# Oppbygningen av operativsystemet

*Dette kapitlet beskriver hvordan et operativsystem kan konstrueres, hvilke funksjoner det skal inneholde og hvilke alternative strategier som finnes ved oppbygningen av operativsystemer.*

## Konstruksjonskriterier

Et operativsystem er et programvareprodukt, og konstruksjonen av et operativsystem er underlagt den samme lovmessigheten som all annen programvarekonstruksjon. Kriteriene som kjennetegner et godt operativsystem, er:

### Ytelse, vedlikehold, korrekthet, standarder

**Ytelse** Et operativsystem «stjeler» behandlingskapasitet fra applikasjonsprogrammene, men forventes å stjele så *lite som mulig*. Operativsystemet må derfor være laget med gjennomtenkte algoritmer og fornuftig bruk av minne- og lagringsressurser.

**Vedlikehold** Et operativsystem er et stort programvareprodukt, et moderne OS som Windows98 inneholder millioner av programlinjer. Vedlikehold av et så stort program kan være svært kostbart med mindre det er konstruert slik at det består av moduler som kan vedlikeholdes uavhengig av hverandre. Mange personer er involvert i vedlikehold og videreutvikling av et operativsystem, og dette blir en kostbar prosess dersom det kreves utstrakt koordinering mellom dem.

**Korrekthet** Et operativsystem har helt uforutsigbare omgivelser. Mange brukere kjører programmer og ber om tjenester og ressurser i en rekkefølge og i et mønster som tilsynelatende er helt tilfeldig. Brukere

kan komme i konflikt med hverandre i måten de bruker operativsystemet på, i den forstand at de kan hindre hverandre i å fullføre oppgavene sine.

Et operativsystem som fungerer korrekt, dvs. i henhold til sine spesifikasjoner, er noe vi oppnår ved å ha god teoretisk forankring i våre programløsninger, god separasjon mellom delprogrammene og en skikkelig plan for testing og feilretting. Tilfeldig testing inntil «det ser bra ut» er en ubrukelig metode for å utvikle operativsystemer.

**Standarder** Om du lager et operativsystem som er ulikt alle andre, må du samtidig skrive alle programmer som trengs på denne maskinen. Det kan være redigeringsprogrammer, kompilatorer, databaseprogram og alle slags hjelpeprogrammer. For å dra nytte av programmer skrevet for andre operativsystemer (f.eks. Linux eller Windows) er det vanlig å lage operativsystemet slik at det følger visse vedtatte måter å tilby sine tjenester på. Vi følger derfor en masse standarder dersom vi har et ønske om at et operativsystem skal samhandle med eksisterende programvareprodukter.

Å følge standarder er en absolutt nødvendighet for at et operativsystem skal bli en kommersiell suksess. Windows-operativsystemene har hele tiden passet på at de kan kjøre programmer skrevet for tidligere versjoner av Windows, og dessuten MS-DOS (det første operativsystemet for IBM PC). Linux er helt fra starten av laget for å følge den såkalte POSIX-standarden, som angir hvordan et program skrevet i C kan benytte tjenestene fra operativsystemet.

Standarder som må tas i betraktning under konstruksjonen av et operativsystem finnes bl.a. på disse områdene:

- Hvilke tjenester som skal tilbys av operativsystemet og hvordan de skal brukes.
- Hvordan et høynivåspråk skal kunne bruke operativsystemets tjenester.
- Organiseringen av dataene på utskiftbare lagringsmedier som diskett og CD.
- Tegnsett og tallformat, dvs. hvilke binærverdier som skal representere bokstavene (særlig aktuelt problem for bokstavene æ, ø og å) og hvordan binærverdier skal tolkes som tallverdier.
- Representasjon av kjørbare programmer på fil (programfiler).

## Skillet mellom virkemåte og grensesnitt

Vi tar med oss den objektorienterte tenkemåten inn i dette kapitlet og skiller mellom *objektets grensesnitt* og *klassens metoder*. I dette perspektivet blir «innmaten» (virkemåten) til operativsystemet noe

som vedkommer dem som utvikler og vedlikeholder det, mens grensesnittet er noe som vedkommer alle som bruker operativsystemet (anvendere og anvenderprogrammerere). Slik er det i objektorientert programmering også.

| Grensesnittet er | Virkemåten er |
|---|---|
| godt kjent av mange | kjent kun for OS-programmererne |
| stabilt, følger standarder | i hurtig utvikling for å utnytte og støtte ny maskinvare |
| viktig for «kompatibiliteten» | uvesentlig for kompatibiliteten |
| tillatt å kopiere | ulovlig å kopiere (åndsverk) |

For å illustrere at grensesnittet betyr «alt» når det gjelder OS-kompatibilitet, vil vi peke på de såkalte «Windows-emulatorer» som finnes for Linux (f.eks. Wine). Dette er programmer som kjører på Linux og som lager et Windows-grensesnitt slik at Windows-programmer kan kjøres på denne maskinen. Det er altså Linux som «gjør jobben», men et Windows-program lar seg lure til å tro at dette er en Windows-maskin, ene og alene fordi programmet finner et grensesnitt som er likt Windows' grensesnitt.

## Grunnfunksjoner i operativsystemet

Et hvert operativsystem har noen basisfunksjoner. I diskusjonen om hvorfor vi trenger operativsystemer (i kapittel 1) identifiserte vi tre grupper av oppgaver (deling, abstraksjon og styring). Disse oppgavene løses i form av *funksjoner* som til dels tilbys programomgivelsene som *kallbare* funksjoner, eller i form av «husholdningsoppgaver» som OS-et tar hånd om.

Grunnfunksjonene i operativsystemet er:

- Prosesshåndtering
- Minnehåndtering
- Filhåndtering
- Utstyrshåndtering
- Meldingshåndtering

## Prosesshåndtering

De ulike oppgavene som er under utføring er representert som *prosesser*. Prosesser inneholder uavhengige *utførende enheter* som utfører en eller flere *instruksjonsstrømmer*, og som styres og støttes av operativsystemet. Antall prosesser som eksisterer i operativsystemet varierer hele tiden.

Håndtering av prosesser innebærer å

- skape og fjerne dem (livssyklus-kontroll)
- starte og stoppe dem (synkronisering)
- fordele CPU-kapasitet mellom dem basert på relativ prioritet (kjøreplan)

Prosesshåndtering behandles i kapitlene 4, 6 og 7.

## Minnehåndtering

Minnet er en ressurs for lagring av data og instruksjoner. Den finnes i en endelig (begrenset) mengde i maskinen, og det er nødvendig at operativsystemet har en fornuftig fordelingspolitikk når oppgavene som utføres i maskinen i sum krever mer minneplass enn hva som er tilgjengelig i maskinen.

Oppgaver i datamaskinen kan be om å få reservert minneplass for seg både i forbindelse med start av oppgaven og underveis i utføringen. En slik forespørsel vil ofte bli innfridd, men det kan også hende at operativsystemet tildeler mindre enn hva som har vært ønsket.

Tildelt minne skal *beskyttes*, dvs. at det skal være beskyttet både mot lesing og endring fra andre oppgaver i maskinen med mindre det er bedt om noe annet. Når flere oppgaver ønsker at minnet skal være delt mellom dem, skal det være mulig å angi hvilke deler av minnet som skal deles, og hvilke deler som skal være beskyttet.

Minneområder som er lite brukt kan overføres midlertidig til disk for å frigjøre minnet for andre oppgaver. Denne teknikken kalles *paging*.

Minnehåndtering behandles i kapittel 5.

## Filhåndtering

Datamaskinen har behov for permanent lagring, dvs. lagring som tar vare på dataene også når strømmen er skrudd av. Slik lagring skjer på mange ulike slags fysiske medier: disk, floppy, CD eller magnetbånd. Vi ønsker at dataene skal lagres med en *logisk struktur* som er mest mulig uavhengig av det fysiske mediet, dvs. vi ønsker oss den samme logiske strukturen på det permanente lageret for flest mulig av de fysiske mediene.

Vi ønsker at den logiske strukturen skal ha form av et *filsystem*. Både i Windows NT og Linux finner vi et filsystem som

- kan samle filer i kataloger. Filnavn må være entydige i samme katalog.
- er hierarkisk, dvs. kataloger kan inneholde både filer og kataloger.
- sørger for at filer og kataloger har navn, datostempler, en eier og en beskyttelse.

Filsystemets egenskaper er i stor grad uavhengig av det fysiske mediet som lagrer filene (Windows tilbyr ikke eierinformasjon og beskyttelse av filer på floppydisk eller CD), og det betyr at anvenderprogrammene kan arbeide med filer uten å ta hensyn til om de ligger på floppy, CD eller disk.

*Ytelsen* i forbindelse med filhåndteringen er viktig. De fleste anvenderprogrammer har utstrakt bruk av filsystemet, og det danner seg fort flaskehalser i denne delen av operativsystemet. Effektivt design av selve filsystemets strukturer og den programkoden som implementerer operasjonene blir en forutsetning for et effektivt operativsystem.

Filhåndtering behandles i kapittel 8.

### Utstyrshåndtering

En datamaskin inneholder mange maskinvarekomponenter. Noen er «faste» deler av inventaret (CPU, timer, DMA) som alltid er til stede, og andre komponenter er «utskiftbare», ofte montert på *innstikkskort* som vi installerer i maskinen selv.

De som utvikler programkoden for operativsystemet vet om noe av denne maskinvaren. Alt det faste inventaret sammen med det vanligste av utskiftbare maskinvarekomponenter vil være støttet av den koden som følger med operativsystemet i standardversjon.

Maskinvareprodukter montert på innstikkskort strømmer derimot ut på markedet i et tempo som programmererne av operativsystemet ikke klarer å henge med på. Men produsentene av maskinvare og produsentene av operativsystemet har en felles interesse i å få dette til å fungere.

**Device Driver** Løsningen ligger i å legge til rette for at produsentene av maskinvare kan lage nødvendig kode selv, og legge den koden ved i pakken med maskinvaren. En slik tilrettelegging kalles *Service Provider Interface* (SPI) og programkoden kalles *Device Driver*.

**Eksempel:** Kjøper du et lydkort, ISDN-adapter, et grafikkort eller lignende, følger det oftest med en diskett eller CD i pakken. Programvaren på den er bl.a. en Device Driver. Denne programvaren installerer du i operativsystemet slik at operativsystemet kan ta ansvaret for styringen av denne maskinvaren.

En Device Driver er en del av operativsystemet, men er ikke skrevet av operativsystemleverandøren. Kvaliteten av en Device Driver er kritisk for operativsystemet; en dårlig skrevet Device Driver kan gjøre hele operativsystemet ustabilt.

**Bruk av avbrudd** Det er i forbindelse med utstyrshåndteringen at OS-et gjør bruk av avbrudd. Husk fra kapittel 2 (side 39) at avbrudd er et elektrisk signal fra en maskinvarekomponent (oftest en i/o-krets) til CPU-en som et varsel om at det er skjedd noe i kretsen som krever betjening.

**Eksempel:** Når et program ber om å få lest data fra en harddisk, så tar det «lang» tid før disse dataene er lest. Lesehodet inne i disken skal flytte seg til riktig spor (typisk 10 ms) og så skal platen inne i disken rotere slik at området der dataene ligger lagret kommer under lesehodet (typisk 4 ms). Denne tiden ønsker vi å bruke til noe fornuftig, så operativsystemet vil utføre andre oppgaver mens disken arbeider med å finne og lese data. Først når dataene er ferdig lest inn i minnet (med bruk av DMA) vil maskinvaren som styrer harddisken varsle CPU-en med et avbrudd. Operativsystemet vet da at oppgaven som trenger dataene er klar til å fortsette utføringen.

Et avbrudd medfører at CPU-en endrer instruksjonsrekkefølgen, ved at den kaller på en *avbruddsrutine* (omtrent som et funksjons- eller metodekall). Vi finner en avbruddsrutine for hver type maskinvarekomponent. Innmaten i avbruddsrutinen inneholder de nødvendige instruksjonene for å betjene maskinvaren (starte eller avslutte operasjonen), flytte på dataene og å varsle ventende oppgaver om at arbeidet kan fortsette. Vi skal omtale avbruddsbehandling mer utfyllende i kapittel 4.

**Felles grensesnitt** Vi ønsker at ulike typer maskinvarekomponenter «abstraheres» til å være mest mulig like, slik at programmene kan behandle «generelle» utstyrsenheter. Vi forsøker å dele inn i/o-enheter med felles egenskaper inn i «familier»:

- Blokk i/o-enheter er de som ekspederer data i blokker. Harddisker, magnetbåndstasjoner og nettverkskort hører til denne familien.
- Tegn i/o-enheter ekspederer ett og ett tegn. Skjermterminaler, skrivere og mus hører til denne familien.

Familiene av i/o-enheter har felles funksjonskall for betjening. Device-Driver-arkitekturen sørger for dette, og det gir fordeler ved at anvender-programmene blir enklere å skrive.

### (Meldingshåndtering)

Meldingshåndtering innebærer å sende data i pakker mellom oppgaver (program under utføring.: En oppgave *mottar* en melding som en «blokk» av data, omtrent slik den mottar data fra nettverket. En oppgave kan også *sende* data adressert til andre oppgaver i maskinen.

Det er først og fremst i moderne operativsystemer at vi finner melding-shåndtering som en grunnfunksjon. Windows NT tilbyr melding-shåndtering gjennom «named pipes», men Linux har *ikke* meldingshåndtering innebygd. Vi har derfor valgt å sette overskriften i parentes.

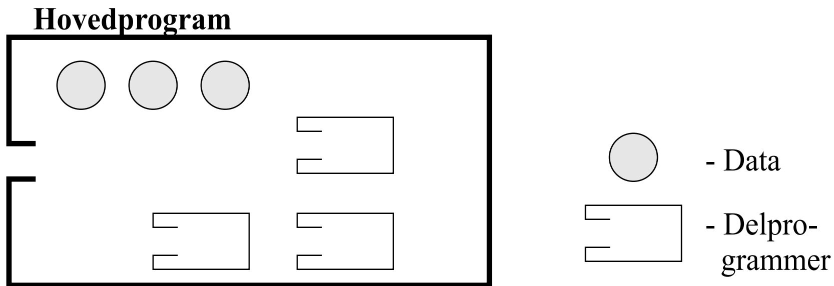
En meldingsmekanisme kan erstatte de vanlige funksjonskallene til operativsystemet. Dette gir en del fordeler i distribuerte operativsystemer.

## Konstruksjonsstrategier

Husk avsnittet “Konstruksjonskriterier” i starten av dette kapitlet, hvor vi ga noen kriterier for programutviklingsarbeidet for et operativ-system. I dette avsnittet vil vi diskutere noen konstruksjonsstrategier som kan hjelpe oss til å møte disse kriteriene.

### Inndeling i moduler – abstrakte datatyper

Helt fra programutviklingens barndom har man forstått betydningen av å bryte ned et stort problem i mange små. Allerede de første høynivå programmeringsspråkene (f.eks. Fortran fra 1959) tilbød delprogrammer i form av *subrutiner*. Subrutiner løste delproblemer, men tillot ikke lokal deklarasjon av data inne i funksjoner/metoder slik f.eks. Java gjør. Se figur 3-1

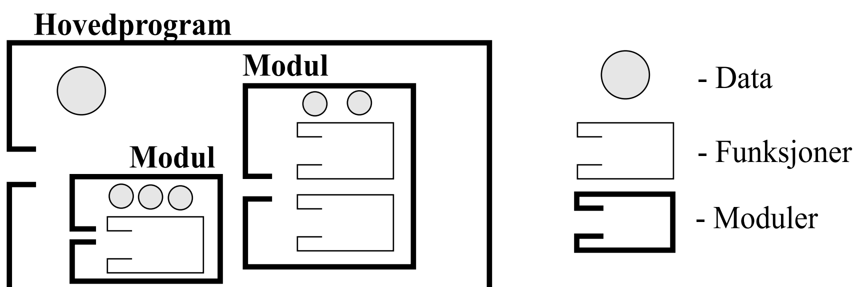



*Figur 3-1: Et hovedprogram inneholder data og subrutiner. Subrutiner inneholder ikke egne data, men må bruke dataområdet som er felles for alle subrutinene.*

Med utviklingen av de *strukturerte* programmeringsspråkene (Pascal, C) fikk vi anledning til å ha *automatiske* variabler, som eksisterer kun under utføringen av delprogrammet (funksjonen). Dette gir oss en bedre isolasjon mellom delprogrammene. Men vi ønsker oss fortsatt en konstruksjonsteknikk som lar delprogrammene ha faste (statiske) data[1] som eksisterer under hele programutførelsen.

De *modulære* programmeringsspråkene kom på slutten av 1970-tallet og tilbød delprogrammer i form av moduler. Moduler var samlinger av funksjoner sammen med datavariabler som var felles for alle funksjonene, men skjult for alle andre. Eksempler på disse programmeringsspråkene het Modula-2 og Ada. Se figur 3-2 for en skjematisk fremstilling av moduler.

1. Med faste/statiske data menes det at datavariablene eksisterer hele tiden mens programmet utføres. Det menes *ikke* at verdiene er faste!




*Figur 3-2: Et hovedprogram som inneholder felles data og moduler. Moduler inneholder data felles for funksjonene i modulen, skjult for omverdenen.*

**Abstrakte datatyper** Moduler gir oss muligheten til å opprette abstrakte datatyper. Vi kan opprette dataelementer som er skjult for direkte adgang fra omverdenen, men indirekte tilgjengelig gjennom modulens funksjoner. Vi kan tenke oss modulen «dato», som gir oss en datatype som inneholder en datoverdi, og funksjoner som lar oss gjøre addisjon og subtraksjon og test av variabler med denne typen.

## Objektorientering

Abstrakte datatyper ble et slags forstadium for den programmeringsteknikken som vi lærer i dag, nemlig objektorientert programmering. De fire egenskapene til et objektorientert programmeringsspråk er:

- klassifisering
- dataskjuling (innkapsling)
- arv
- polymorfi

Hovedforskjellen på et modulært og et objektorientert programmeringsspråk er at de abstrakte datatypene nå kan bekrives i et slektstre. Høyt opp i treet kan vi beskrive de *generelle* egenskapene til alle medlemmer i den aktuelle grenen, og gi mer *spesifikke* egenskaper til datatyper lenger ned i treet.

Hvilke fordeler har vi av å benytte objektorienterte programmeringsteknikker i et operativsystem? I hovedsak de samme fordelene som ved annen programutvikling. I tillegg kan vi tenke oss fremtidige operativsystemer som tilbyr sine tjenester gjennom klasser som vi kan arve fra.

Vi kunne tenke oss at operativsystemet har klassen «fil» som vi kan arve fra og lage klassen «regnskapsfil», som har den egenskapen at den ikke kan slettes før etter 10 år.

**Viktig:** Stikkordene er *isolasjon* og *gjenbruk*! Med disse egenskapene blir det

- lettere å dele programmeringen mellom flere personer
- enklere å vedlikeholde programmene siden
- lettere å teste programmene
- mindre feil og uønskede sideeffekter

## Konfigurasjon av operativsystemet

Operativsystemet skal benyttes i mange ulike sammenhenger, og det må ha en viss fleksibilitet med hensyn til hvilke funksjoner og programmoduler som skal inngå i et gitt tilfelle. Det må derfor være mulig å *konfigurere* operativsystemet for et bestemt formål. Skal datamaskinen være tilkoplet nettverk, en skriver eller ha et lydkort installert? Skal det betjene få, men store oppgaver, eller mange små oppgaver?

Ordningen med Device Drivere kan i noen grad avgjøre hvilke programmoduler som skal inngå i systemet, men i tillegg er det vanlig at vi kan inkludere eller fjerne også «grunnmodulene» i systemet. Konfigurasjon av operativsystemet kan også være å endre de parametrene som systemet opererer etter (skal f.eks. operativsystemet støtte multiprosessor-konfigurasjon?).

Endring av operativsystemets konfigurasjon vil oftest kreve en omstart av systemet, etter at operativsystemets programfiler og konfigurasjonsfiler er blitt endret. Endring av operativsystemets programfiler kan innebære en re-lenking av programfilene[2] eller bare utskifting av enkelte dynamisk lenkede programfiler.

**Eksempel:** Linux krever re-lenking av kjernens programfil for en del type endringer. Dette er tidkrevende, men det finnes et konfigurasjonsprogram som gjør det noenlunde enkelt. For andre typer endringer finnes programmet `linuxconf` for endring av nettverkskonfigurasjon, start og stopp av tjenerprosesser o.l.

---

2. Lenking av programfiler er ukjent for en Java-programmerer. I andre programmeringsspråk, som C/C++, er dette en nødvendig arbeidsoperasjon mellom kompilering og kjøring av et program. Hensikten er bl.a. å binde separat kompilerte funksjoner og funksjonskall til hverandre.

Forskjellen mellom statisk og dynamisk binding av programmer vises tydelig når vi skal konfigurere et operativsystem:

- Statisk binding er en mer tidkrevende og tungvint prosess, men er sikrere i den forstand at feil i konfigurasjonen kan oppdages i løpet av lenkeprosessen.
- Dynamisk binding er mer fleksibel og enkel, men gir ikke den samme kvalitetskontrollen. Windows-operativsystemene hadde tidligere store problemer med versjonskontroll av sine såkalte DLL-filer, men har nå løst disse problemene med versjonskontrollen som er innebygd i COM/ActiveX-arkitekturen.

## Mikrokjerne

En måte å designe et operativsystem på, hvor fleksibel konfigurasjon har vært et viktig mål, er mikrokjernearkitekturen. I en mikrokjerne har operativsystemet kun det minimale av tjenester (f.eks. minnestyring, prosesstyring og meldingshåndtering), men er laget for å kunne kalle på funksjoner i dynamisk lastede programmoduler. Ikke bare i forbindelse med i/o-styring og Device Drivere, men i alle typer tjenester.

Dersom man i et mikrokjerne-operativsystem ønsker en annen type prosesskontroll, kan man lage en modul som passer til det riktige SPI og laste inn denne modulen. Det samme gjelder filsystem og brukergrensesnitt. Flere moduler for f.eks. brukergrensesnitt kan være lastet inn og i bruk samtidig, slik at operativsystemet har flere «ansikter», f.eks. Windows for én bruker og MacOS for en annen.

Det finnes ikke mange mikrokjerneoperativsystemer på markedet, men forskningen på mikrokjerner har gitt resultater som har styrket de kommersielle operativsystemene. Windows NT har f.eks. en del elementer av et mikrokjerneoperativsystem i seg.

## Bootstrap

Når maskinen blir skrudd på er den «naken og nyfødt», og minnecellene i RAM inneholder ingen ting. Innholdet i ROM-cellene er derimot bevart, og det er dette innholdet som skal danne grunnlaget for å laste inn nødvendig programkode og å starte de oppgavene som er nødvendig for at operativsystemet skal være operativt. Bootstrap er avledet av et engelsk uttrykk for å «trekke seg opp etter hårene».

**Bootstrap foregår i flere steg:**

1. Når CPU-en får strømmen påslått vil den begynne å utføre instruksjoner fra en bestemt minneadresse. Minnecellene på denne adressen må ligge i ROM (permanent minne).

2 Instruksjonene i ROM-minnet (på en pc er dette kalt BIOS - Basic Input/Output System) foretar en selv-sjekk av maskinvaren.
3 BIOS vil deretter finne hvilken disk som har operativsystemets programfiler, og sette opp en leseoperasjon for de første sektorene på denne disken.
4 Disse første sektorene inneholder «kjernen» av operativsystemet. De blir kopiert inn i minnet (RAM) og instruksjonsutføringen hopper til en bestemt adresse inne i denne programkoden.
5 Filsystemet blir etablert, konfigurasjonsfiler lest inn og nødendige oppgaver blir startet.
6 Brukergrensesnittet blir etablert og brukeren kan logge seg inn og begynne arbeidet.

Bootstrap-planen kan endres i forbindelse med konfigurasjonen av operativsystemet. Vi er oftest interessert i å kontrollere hvilke programmer som skal startes automatisk ved oppstart.

## Grensesnitt til operativsystemet

Alle oppgavene som utføres i et operativsystem er representert ved at det er et program under utførelse. Dette programmet vil hele tiden be om tjenester fra operativsystemet gjennom et grensesnitt. Tjenester som det aktuelt å be om, kan være:

- Reservering av minneområde
- Åpning, lesing og lukking av filer
- Sende og motta data i nettverket
- Vise tekst og figurer på skjermen

En annen type program som også trenger et grensesnitt til operativsystemet, er en Device Driver, som kan tenkes å be om å bli *registrert* som en Device Driver for en gitt maskinvarekomponent, men som siden ikke selv vil kalle, men *blir kalt* av operativsystemet når det er nødvendig å utføre operasjoner mot denne komponenten. En Device Driver er ikke en selvstendig oppgave, men et stykke kode som «betjenes» av operativsystemet. Begge disse to typer av grensesnitt skal vi nå beskrive nærmere.

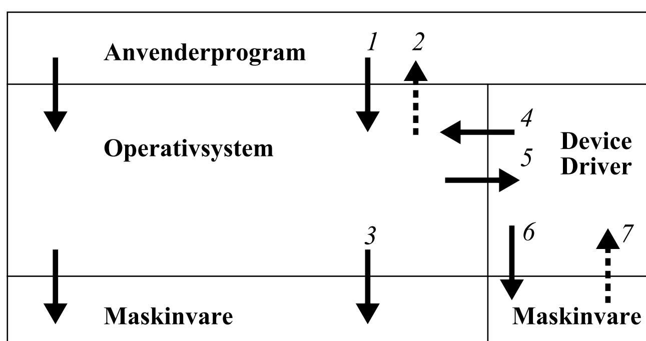
### API og SPI

Det grensesnittet som et anvenderprogram benytter for å kalle på operativsystemets tjenester, kalles *Application Program Interface* (API). Utvalget av funksjoner i et API er preget av to forhold:

- Det er oftest anvenderprogrammet som kaller på operativsystemets funksjoner, sjelden omvendt[3].
- Utvalget av funksjoner gjenspeiler anvenderprogrammets tjenestebehov, og har et høyt abstraksjonsnivå (befatter seg sjelden direkte med maskinvarekomponenter).

For programkode som tilbyr «tjenester» til operativsystemet, f.eks. Device Drivere, tilbyr operativsystemet et *Service Provider Interface* (SPI). Fordi slik programkode ikke er en selvstendig oppgave, er et SPI oftest bygd på denne virkemåten:

- Programkoden blir lastet inn i minnet, og en «registreringsfunksjon» i SPI blir kalt for å fortelle at programkoden står til tjeneste.
- Når operativsystemet trenger tjenestene til programkoden, blir det gjort kall til på forhånd bestemte funksjoner i denne programkoden.




*Figur 3-3: Bruk av API og SPI . Se teksten for en forklaring til tallene på pilene*

**Figur 3-3 viser en del kall i API og SPI. Pilene viser (ref. numrene) retning på kallene**

1. API: Ordinært systemkall fra anvenderprogrammet til operativsystemet, f.eks. om å åpne en fil for lesing.
2. API: Kall fra operativsystemet tilbake til anvenderprogrammet. I forbindelse med bruk av grafisk brukergrensesnitt skjer dette f.eks. når programmet må tegne opp det grafiske vinduet på nytt, eller når brukeren velger en menyfunksjon.
3. Operasjoner fra operativsystemet mot en maskinvarekomponent, f.eks. starte en i/o-operasjon mot harddisk. Denne operasjonen hører verken til API eller SPI.

---

3. Ved bruk av grafisk brukergrensesnitt finner vi unntak fra denne regelen.

4 SPI: Kall fra en Device Driver mot operativsystemet, f.eks. med ønske om å registrere driveren for bruk.
5 SPI: Kall fra operativsystemet mot Device Driver, f.eks. med beskjed om å starte en i/o-operasjon mot den aktuelle maskinvarekomponenten.
6 Operasjoner fra en Device Driver mot en maskinvarekomponent. Verken API eller SPI
7 Avbruddssignal fra en maskinvarekomponent tvinger CPU-ens instruksjonsrekkefølge til å hoppe inn i Device Driverens avbruddsrutine. Dette skjer f.eks. når i/o-operasjonen er fullført.

## Programvareavbrudd – INT-instruksjonen

Et kall til operativsystemet gjennom et API skiller seg fra et vanlig funksjonskall ved at det må være *lokasjonsuavhengig.*

Når et anvenderprogram kaller en vanlig funksjon, trenger anvenderprogrammet å kjenne den minneadressen hvor funksjonen er lagret. Denne «lenkingen» skjer typisk som en del av programutviklingen og binder funksjonen til samme «pakke« som anvenderprogrammet.

Et anvenderprogram kan ikke bindes til operativsystemets funksjoner på denne måten, og er nødt til å finne de aktuelle minneadressene ved hjelp av indirekte metoder: Et kjent sted i minnet ligger det tabeller som peker til disse minneadressene.

En slik indirekte metode finner vi i form av en avbruddsrutine. I likhet med et avbrudd som oppstår som et resultat av et elektrisk signal fra en i/o-krets, kan et avbrudd også oppstå som et resultat av en særskilt avbrudds-instruksjon. Denne instruksjonen har ulike navn hos de forskjellige CPU-produsentene, men hos Intel kalles den for «INT».

Resultatet av INT-instruksjonen er at instruksjonsrekkefølgen vil hoppe til en minneadresse som er oppgitt i en peker på en kjent adresse. Vi kaller en slik peker for en avbruddsvektor. Denne teknikken løser problemet med uavhengig plassering av anvenderprogrammet og operativsystemet i minnet.

## User mode/supervisor mode

Under utføringen av funksjonskode som hører til operativsystemet, krever anvenderprogrammets oppgave adgang til minneceller, i/o-kretser og CPU-instruksjoner som det normalt ikke har adgang til. Dette er nødvendig fordi

- det er i forbindelse med kall til operativsystemet at minneplass blir tildelt et program. Dette krever endring av CPU-registre som er beskyttet under normal programutførelse.
- operativsystemets egne dataområder må oppdateres når f.eks. en ny oppgave blir påbegynt. Disse dataområdene ligger i minneceller som bør være beskyttet under normal utførelse.

Det er altså nødvendig å skille mellom to tilstander som oppgaven kan være i: Mens den utfører normal programkode (anvendeprogrammet), og når den utfører kode *inne i* operativsystemet (som en del av et operativsystemkall).

Disse to tilstandene kaller vi henholdsvis *user mode* og *supervisor mode*. Det er sikkerhetskravet i et operativsystem som er bakgrunnen for at vi lager et slikt skille. Det er bare under utføring av operativsystemets kode at oppgaven skal ha utvidede rettigheter, *aldri når egen brukerkode utføres!*

Et slikt krav er INT-instruksjonen i stand til å oppfylle. Når oppgaven utfører en slik instruksjon vil CPU-en skifte fra user mode til supervisor mode, og tilbake igjen når avbruddsrutinen avslutter sin utføring (med en instruksjon som sier «return from interrupt» – IRET). Oppgaven som utfører en INT-instruksjon hopper til en minneadresse som den ikke kan angi selv, men som er oppgitt i en peker kalt avbruddsvektoren. Avbruddsvektoren må ligge på minneadresser som er beskyttet slik at de ikke kan endres mens CPU-en er i user mode.

## Funksjonsgrensesnitt eller meldingsgrensesnitt?

Fra et teoretisk synspunkt ønsker vi å vurdere en alternativ måte å kalle operativsystemets tjenester på. Fremfor å bruke funksjonskall gjennom INT-instruksjonen kan vi tenke oss at *meldinger* blir sendt fra en oppgave til operativsystemet. Med melding mener vi en «pakke» med alle nødvendige opplysninger for at operativsystemet skal kunne utføre tjenesten. Resultatet av operasjonen kommer i form av en melding fra operativsystemet til klient-oppgaven.

Fordi en «pakke» er enkel å flytte via et nettverk (fremfor et funksjonskall) er et operativsystem med *meldingsgrensesnitt* enklere å tilpasse et distribuert miljø enn et operativsystem med et funksjonsgrensesnitt. Vi kan f.eks. tenke oss et kall til et filsystem.

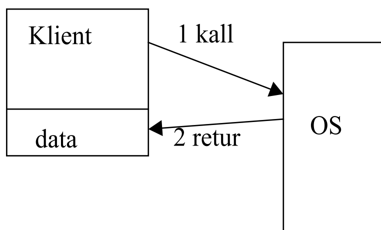
**Eksempel:** En klientoppgave lager en pakke som inneholder filnavn, og en billett som beviser identiteten til brukeren som «eier» oppgaven. Denne pakken sendes til operativsystemet (på egen eller en annen maskin) som utfører kallet og returnerer en «filbillett» som representerer den åpne filen.

Senere kan klientoppgaven lage en ny pakke med denne filbilletten sammen med en beskjed om å lese en del av denne filen. Operativsystemet utfører denne ordren og returnerer de leste dataene i en pakke som sendes motsatt vei.

Et meldingsgrensesnitt vil ha både fordeler og ulemper i forhold til et funksjonsgrensesnitt.

**Fordeler** Et meldingsgrensesnitt gir en bedre plattform for distribuerte løsninger, fordi det er akkurat like enkelt å bruke tjenestene til et operativsystem på en annen maskin som på den lokale maskinen. En annen fordel er at alle kallene blir «asynkrone», dvs at oppgaven kan holde på med andre aktiviteter mens tjenesten blir utført i operativsystemet, og at vi kan ha mange tjenester «under utføring» i flere operativsystemer samtidig.

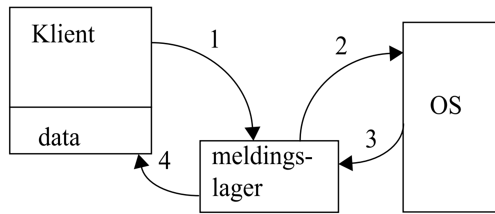
**Ulemper** Meldingsoverføring krever mellomlagring av data. I et funksjonsgrensesnitt vil lesing av data fra en fil kreve en parameter som peker til en rad med minneceller hvor fildataene kan skrives inn. Operativsystemet skriver disse dataene inn direkte i klientoppgavens minneområde. I et meldingsgrensesnitt må dataene mellomlagres i en pakke inne i operativsystemet, deretter overføres tilbake til klientens lokale operativsystem, som igjen krever mellomlagring før den overføres til klientens minneområde. Asynkrone kall har også en mer kompleks feilhåndtering.




(1) Klienten kaller en funksjon for å lese data fra en fil, og legger til en parameter som peker til dens eget dataområde.

(2) OS returnerer dataene ved å skrive dem direkte inn i klientens dataområde.

Ingen mellomlagring er nødvendig.




(1) Klienten sender en «bestilling» til OS, som mottar denne gjennom et meldingslager.

(2) Operativsystemet sender svar i form av en melding som mellomlagres i meldingslageret.

(3) Når klienten er klar til å motta resultatet, henter den selv meldingen fra meldingslageret og (4) kopierer dataene til sitt eget dataområde.

Mellomlagring er nødvendig fordi sending og mottak av meldinger kan skje til forskjellig tid.

*Figur 3-4: Funksjonsgrensesnitt (t.v.) og meldingsgrensesnitt*

# Programmering i høynivåspråk

Når du programmerer i et høynivåspråk, f.eks. Java, ser du aldri INT-instruksjonen. Du ser aldri CPU-registrene, minneadresser eller i/o-enhetene. Alt dette er skjult bak en stor abstraksjon som sier[4]:

- «Dette er en maskin som forstår Java-instruksjoner. Komplekse beregninger eller tester kan uttrykkes i én eneste setning».
- «Denne maskinen har ikke minneceller, men symbolske variabler av bestemte datatyper, objekter og tabeller».
- «Variablene har en levetid avhengig av hvor de er deklarert i programkoden».
- «Her finnes ikke et operativsystem, men et bibliotek av objekter som du kan opprette og manipulere for å lagre data eller gjøre i/o-operasjoner».

For å skape disse abstraksjonene må det ligge et «lag» mellom høynivåspråket og operativsystemet i form av programkode. Laget består av disse delene: *Kompilator, klassebibliotek og kjøremiljø.*

---

4. Disse eksemplene knytter seg spesifikt til programmeringsspråket Java.

## Kompilator

Kompilatoren er et program kjent for alle som programmerer i et høynivå programmeringsspråk. Det tar et kildeprogram som inndata og produserer et objektprogram[5]. Objektprogrammet inneholder opplysninger om programmets minnebehov og en serie med enkle instruksjoner som er ekvivalente til programsetningene i kildeprogrammet. Instruksjonene kan være CPU-instruksjoner som utføres direkte av CPU-en, eller «tenkte» maskininstruksjoner som utføres av en tolk (det siste er tilfellet med Java-kompilatoren).

Dessuten inneholder objektprogrammet opplysninger om hvilke andre funksjoner og klasser som det refereres til. Dette kan dreie seg om andre, separat kompilerte deler av samme anvenderprogram, eller funksjoner og klasser som hører til spåkets *kjøremiljø*.

## Klassebibliotek

Mye brukte fellesfunksjoner ønsker vi å skille ut i separate bibliotek. På den måten forenkler vi utviklingen av anvenderprogrammer. Kode som styrer dialogen i grafisk brukergrensesnitt, i/o mot filsystem og nettverk, og samarbeid med andre oppgaver i maskinen, ønsker vi å få med som en del av «pakken» når vi anskaffer et språksystem.

Et eksempel er klassebiblioteket som kalles «Java API». Her finner vi et stort antall klasser som hjelper oss med å utføre matematiske beregninger eller manipulere tekststrenger. Men her finner vi også det *abstraksjonslaget* som fremstiller operativsystemets tjenester som et sett av klasser og metoder.

Om vi vil lese en fil fra et Java-program velger vi altså ikke å finne ut hvilket API-kall som vi skal bruke for å få dette til, men vi instansierer et objekt av klassen `java.io.FileReader`. For å lese data fra filen kaller vi deretter metoder på dette objektet, så det ser ut som om `FileReader`-objektet «er» filen. Fordelen med denne måten å gjøre det på er at vi i mindre grad eksponerer programkoden for forskjeller mellom APIene i ulike operativsystemer, og at operativsystemets tjenester «ser ut som» funksjons- eller metodekall.

Et klassebibliotek (ev. funksjonsbibliotek) følger alltid med når du anskaffer en kompilator. Kunnskap om bruken av biblioteket er like viktig som kunnskap om selve programmeringsspråket.

---

5. Dette er en gammel betegnelse, og har ingenting med objektorientert programmering å gjøre!

## Kjøremiljø

Et program som skal startes og utføres under «overvåking» av et operativsystem, krever en del støttefunksjoner i tillegg til klassebiblioteket, bl.a. i forbindelse med oppstart og avslutning av programmet. Disse støttefunksjonene skaper «omgivelsene» til det utførende programmet, og kalles derfor ofte *kjøremiljøet* (eng.: run-time environment).

Kjøremiljøet er ofte representert i form av en «program loader», et program som påtar seg å laste inn et kjørbart program fra disk, plassere det i minnet og be operativsystemet om de nødvendige tjenester for å starte programmet, bl.a. å tildele tilstrekkelig minne.

En program loader kan være et separat program, men et kjørbart program kan også være utformet som sin egen programloader[6], ved at den nødvendige programkoden er lagt til i form av klassebibliotek. Program loaderen for Java-programmer har flere tilleggsoppgaver som vi skal nevne litt senere.

Organiseringen av minneområdet for et program skrevet i et høynivåspråk er en viktig oppgave for kjøremiljøet, og vi skal nå se spesielt på hvordan denne organiseringen finner sted.

## Organisering av variabler og objekter

Et moderne høynivåspråk har variabler med ulik levetid. Når levetiden opphører ønsker vi at de minnecellene som representerer variabelen skal «returneres» slik at de kan brukes til andre formål. Ut fra dette perspektivet kan vi dele variabelene opp i tre typer.

**Statiske variabler** Variabler som skapes når programmet starter sin utføring og eksisterer helt til utføringen er ferdig, kalles statiske variabler. Plassbehovet for de statiske variablene kan kalkuleres av kompilatoren, fordi alle slike variabler må deklareres (eksplisitt eller implisitt) i programkoden. Plassbehovet vil deretter inngå i det absolutte kravet som kjøremiljøet vil stille til minneområdet for at programmet skal kunne startes.

I Java er de statiske variabler alle variabler som er deklarert med nøkkelordet *static*[7].

**Automatiske variabler og parametre** Variabler som er deklarert som lokale for en metode eller funksjon kalles automatiske variabler. Disse variablene blir skapt når metoden starter sin utføring, og

---

6. Ikke alle funksjonene til kjøremiljøet kan utføres her, f.eks. innlasting i minnet.
7. Statiske variabler i Java blir egentlig opprettet når klassen lastes inn i minnet, noe som kan skje underveis i programutføringen.

opphører å eksistere når metoden returnerer. Det samme gjelder for parametre som er deklarert sammen med metoden, ofte kalt *formelle* parametre.

Det er ikke mulig for kompilatoren å kalkulere plassbehovet for automatiske variabler. Hver gang en metode kalles blir det opprettet et nytt «sett» av automatiske variabler, og plassbehovet blir derfor avhengig av hvor «dypt» metodekallene er nøstet nedover i løpet av programutføringen. Dybden avhenger ofte av de dataene som behandles, slik tilfellet er f.eks. i *rekursive* algoritmer.

En viktig egenskap ved automatiske variabler er at de forsvinner i *omvendt rekkefølge* av slik de ble skapt, fordi returveien for metodene alltid er omvendt av den veien kallene gikk.

**Dynamiske variabler** Variabler og objekter som skapes gjennom uttrykkelige kommandoer, kalles dynamiske variabler. I Java blir objekter opprettet av «new»-setningen, og forsvinner når de ikke lenger blir referert[8]. I andre språk (f.eks. C, C++ og Pascal) forsvinner variabler og objekter som et resultat av en «delete»-setning. Det er av opplagte årsaker ikke mulig å kalkulere plassbehovet for de dynamiske variablene og objektene.

Dynamiske variabler og dynamiske objekter skapes og forsvinner i en uforutsigbar rekkefølge, og krever en annen lagringsmetode enn de automatiske variablene.

```
public class Variabler {
  static int a,b,c; // Statiske variabler
  static String s1,s2; //Statiske objektreferanser
  int auto(int i, int j) { //Parametre er automatiske
    float f1,f2;         // Automatiske variabler
    String s3 =
      new String(«123»); // Automatisk obj-ref.
  }
}
```

*Listing 3-1: Ulike typer lagringsmetoder for Java-variabler og -objekter*

8. En lærebok i Java forklarer disse tingene fyldigere enn hva vi har plass til her.

## Stakken og heapen

For å lagre de automatiske variablene brukes en datastruktur som kalles *stakk*. Den er lik en stabel med tallerkener. Når du skal legge noe på stabelen legges det på toppen, og det du henter kommer også fra toppen. Det er på samme måte slik at det som sist ble lagt på stakken som først hentes ut. En stakk er derfor velegnet for å holde rede på metodekall, der vi skal fjerne automatiske variabler i motsatt rekkefølge av den de ble skapt i, og der returveien er motsatt av kallveien.

En stakk er meget enkel å programmere ved å bruke et array (tabell) sammen med en variabel som peker til toppen av stabelen. Java har dessuten en standardklasse som heter `java.util.Stack`

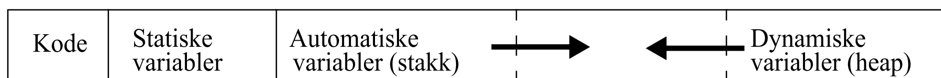
**Viktig:** Ved kall til en metode eller funksjon skapes automatiske variabler, formelle parametre og returadressen på kjøremiljøets stakk.

For de dynamiske variablene gjelder ikke denne rekkefølge-egenskapen. Data skapes og opphører i vilkårlig rekkefølge. Når data opphører å eksistere ønsker vi at den plassen skal kunne gjenbrukes til et annet formål senere. En *heap* er en slik datastruktur som lar oss reservere en gruppe med minneceller til vårt eget bruk, og siden returnere dem når vi ikke trenger dem lenger.

En heap lages ofte som en lenket liste av ledige grupper av minneceller, sortert på størrelse. En forespørsel om å reservere minne vil resultere i at den rette gruppen blir tatt ut av listen (minst blant dem som er større enn forespørselen). Det returnerte minnet blir hektet tilbake på den lenkede listen og er tilgjengelig for bruk siden.[9]

Heapen er en viktig del av kjøremiljøet i programmeringsspråk som benytter dynamiske data (og det gjør de fleste).

Stakken og heapen varierer i størrelse, og det er ikke mulig å kalkulere plassbehovet for dem. For å organisere et minneområde, slik det er tildelt av operativsystemet, og slik at det passer med bruksmønsteret i et høynivåspråk, kan vi f.eks. gjøre det slik:




*Figur 3-5: Organisering av kode og hhv. statiske, automatiske og dynamiske variabler i et tildelt minneområde.*

9. Dette er bare hovedtrekkene i en ellers ganske komplisert algoritme.

### Java Virtual Machine

Et program (en klasse) skrevet i programmeringsspråket Java, blir under kompileringen oversatt til et sett av «abstrakte» maskininstruksjoner, såkalt *bytecode*. Under utføring av programmet er det ikke CPU-en som utfører instruksjonene direkte (slik tilfellet er i f.eks. C++), men en *tolk* som etterligner en CPU. Bytecode er såkalt arkitekturnøytral og kan utføres på alle maskiner som har en Java-tolk installert. Av den grunn er det mulig å distribuere ferdig kompilert Javakode over Internett, f.eks. i form av *Java applets*.

Kjøremiljøet for et Java-program er av flere grunner ganske komplisert. Koden for kjøremiljøet finner vi i form av programmet «Java Virtual Machine» (JVM) sammen med en del Java-klasser. JVM har bl.a. disse funksjonene:

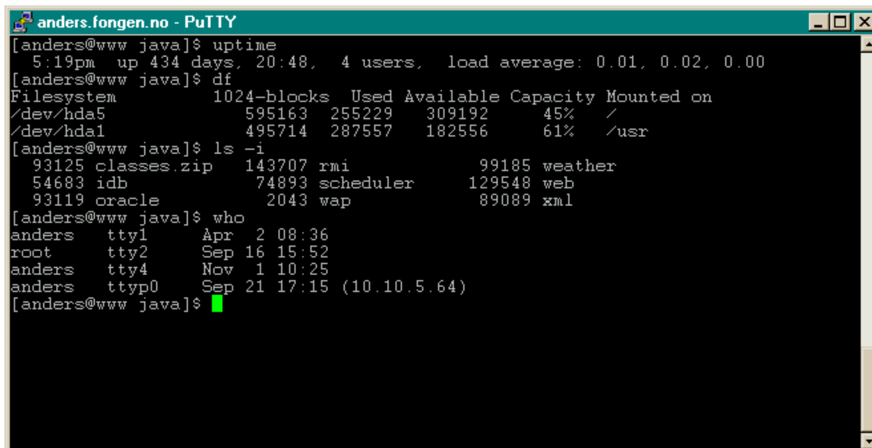
- Være tolk for de kompilerte klassene (bytecode)
- Lage et grensesnitt mellom operativsystemets tråder (forklares i neste kapittel) og Java-trådene
- Utføre automatisk fjerning av foreldreløse objekter (garbage collection)
- Laste inn Java-klasser etter behov
- Støtte sikkerhetsmodellen til Java

**Viktig:** Java Virtual Machine er en del av kjøremiljøet til et Java-program.

## Skallet

Alle generelle operativsystemer må kunne tilby brukeren å gi kommandoer, starte programmer og studere resultatet av programkjøringen. Operativsystemet trenger derfor et program som tilbyr brukeren et betjeningspanel. Dette programmet kaller vi ofte et *skall* fordi det «omslutter» operativsystemet.

## Tegnbasert skall



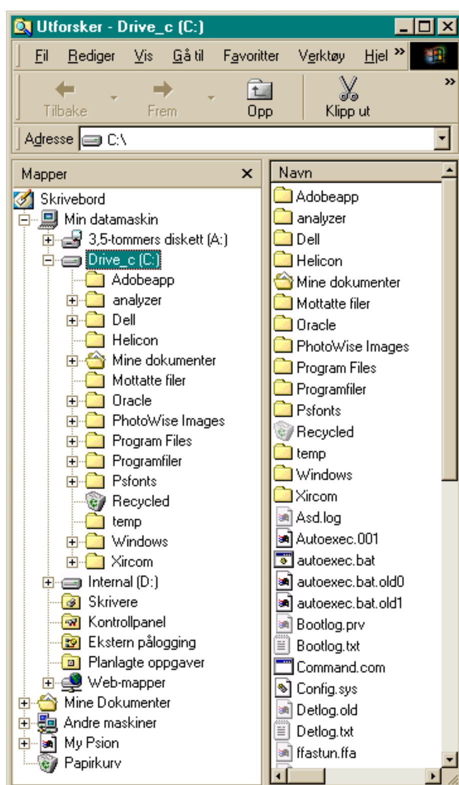

*Figur 3-6: Et tegnbasert skall i en Linux-maskin*

Vi finner såkalt tegnbaserte skall for de fleste operativsystemer, også Windows og Linux. Tegnbaserte skall kjennetegnes ved

- at de er vanskeligere å bruke. Betjeningen skjer ved kommandoer som skrives inn på en kommandolinje. Du har lite hjelp tilgjengelig for å vite hvilken kommando som kreves for en gitt oppgave. Det kreves derfor mer kunnskap å bruke et tegnbasert skall.
- at de er programmerbare i den forstand at en serie kommandoer kan sammenstilles til et program. Skallene har også egne setninger for test- og løkke-operasjoner og enkel variabelhåndtering. Skallene under Linux er såpass avanserte at man kan lage ganske omfattende programmer med kommandospråket (om man orker).
- at de krever mindre dataoverføring mellom brukeren og maskinen (kun tastetrykk og tegnkoder) og tillater derfor at maskinen betjenes over langsomme kommunikasjonskanaler. Om du vil betjene en Linux-maskin på den andre siden av kloden vil du være mest komfortabel med å bruke et tegnbasert skall.

## Symbolbasert skall

Et alternativ til tegnbasert skall er å bruke symbolbasert skall, noe som du helt sikkert er fortrolig med gjennom bruk av Utforskeren under Windows




*Figur 3-7: Symbolbasert skall under Windows (Windows Explorer)*

Symbolbaserte skall kjennetegnes ved at de:

- er enkle å bruke. Operativsystemets tilstand (f.eks. filene som finnes på en katalog) er hele tiden synlig, og du har hele repertoaret av mulige operasjoner tilgjengelig på menyer.
- er vanskelig å automatisere, fordi det ikke er noen naturlig måte å sammenstille operasjoner på.
- krever større overføringskapasitet mellom terminalen og maskinen. Under Windows er ikke dette noe problem, fordi der er skjermen, tastaturet og musen knyttet direkte til maskinens buss gjennom i/o-kretser med høy overføringskapasitet. Med Linux er det mulig å bruke symbolbasert skall også via nettverk. Slik bruk av Linux vil gjerne begrenses innenfor et lokalnett, fordi det kreves stor overføringskapasitet.

**Viktig:** Både tegnbaserte og symbolbaserte skall har sine fordeler, og en erfaren bruker av et operativsystem må beherske begge typer!

## Sammendrag

- Konstruksjonskriteriene for et operativsystem er *ytelse, vedlikehold, korrekthet, standarder.*
- Et operativsystem skal styre *prosesser, minne, filer og utstyr.*
- Konstruksjonsstrategier for et operativsystem er: *modularisering* og *objektorientering*, skille mellom *grensesnitt* og *implementasjon.*
- Grensesnittet til operativsystemet (API og SPI) avgjør hvordan operativsystemet "ser ut" for omverdenen.
- Skillet mellom *user mode* og *supervisor mode* setter operativsystemet i stand til å tilby tjenester til brukerprogrammene uten å sette sikkerheten i fare.
- Et meldingsgrensesnitt gir enklere distribusjon av operativsystem-tjenester, men er mer ressurskrevende (mer minne) i forhold til et funksjonsgrensesnitt.
- Et program må kompileres for å oversette kildeprogrammet til maskininstruksjoner. Et klassebibliotek inneholder standard fellesfunksjoner og lenkes til programmet.
- Kjøremiljøet er «omgivelsene» til et program, satt opp av operativsystemet.
- Skallet er grensesnittet mellom operativsystemet og brukeren, og tilbyr betjening av vanlige funksjoner.

| **Sentrale begreper i dette kapitlet:** | |
|---|---|
| Konstruksjonskriterium | Konstruksjonsstrategi |
| Modulariseing | Objektorientering |
| User Mode/Supervisor Mode | API/SPI |
| Kompilator | Klassebibliotek |
| Kjøremiljø | Skall |

## Teorioppgaver

**Gå sammen i grupper og besvar følgende oppgaver:**

1 Hvordan kaller vi på funksjonene i operativsystemet fra et Java-program? Hvilke forskjeller finner vi i disse kallene når vi bruker henholdsvis Linux og Windows?

2 Søk på Internett og finn noen ulike oppfatninger av begrepet «objektorientert operativsystem». Hva er deres meninger om hva dette begrepet bør bety?

3 Søk på Javas nettsted og finn eksempler på SPI brukt i Java-teknologi.

4 «Java Virtual Machine» danner kjøremiljøet for et Java-program, og klassebiblioteket inneholder standardfunksjonene. Hvilken av disse to komponentene har ansvaret for:
- Kalle- `static void main(String[]..)` funksjonen?
- Åpne og lukke filer?
- Kontrollere at stakken ikke forvokser seg?
- Sette opp det grafiske brukergrensesnittet til programmet?
- Fjerne ubrukte objekter (garbage collection)?

5 Finn frem API-et til klasse `java.util.Stack.` Skriv Java-koden for å implementere disse metodene (ikke de arvede metodene).

6 Er Windows NT et mikrokjerne-operativsystem? (Dette spørsmålet krever en del søking og undersøkelser.)

## Øvingsoppgaver

Bokas nettsted gir forslag til noen ferdighetsøvinger knyttet til dette kapitlet.

**Etter øvingene skal du beherske:**

1 Redigering, kompilering og kjøring av Java-programmer både på Linux og Windows.

2 Flytte ferdigkompilerte klasser mellom Windows og Linux og demonstrere portabiliteten av kompilert kode.

3 Redigering og kjøring av enkle skall-programmer på Linux.

4 Enkel omkonfigurering av Windows Utforsker.

5 Passordbeskyttelse av dine egne nettsider.

6 Lese- og skrivebeskyttelse av dine egne filer og kataloger på Linux.

Kapittel 4

# Prosesser og tråder

*I dette kapitlet vil vi diskutere en del teoretiske problemer knyttet til samarbeidende prosesser og tråder i et operativsystem. Et viktig begrep i denne sammenheng er kjøreplan, som betegner den metoden operativsystemet benytter for å bestemme hvordan oppgavene skal ordnes under utføringen.*

## Hva er en prosess? Hva er en tråd?

### Kjøkkenet

En prosess kan sammenlignes med et restaurantkjøkken. I kjøkkenet finner du en eller flere *kokker*, arbeidsbenker, *verktøy og råvarer*. Og kjøkkenet har en *hensikt*, nemlig å servere kundene mat. For å lage mat trenger kokkene *oppskrifter*.

En prosess er i likhet med dette kjennetegnet ved:

**En hensikt** Prosessen er innrettet mot å utføre en oppgave. Vi har hittil i boka brukt begrepet *oppgaver* for å betegne utførte operasjoner i operativsystemet. Dette er synonymt med begrepet *prosess*, som vi vil bruke heretter.

**Utførende enheter** En prosess har ressurser knyttet til seg som kan utføre maskininstruksjoner. Slike ressurser kaller vi en *tråd*, og er å sammenligne med kokken på kjøkkenet. En prosess kan ha flere tråder som hver for seg utfører instruksjoner mer eller mindre uavhengig av hverandre. Tråder er viktige fordi de representerer den *aktive og utførende* delen av en prosess.

**Omgivelser** For å utføre en oppgave må prosessen ha råvarer og verktøy, dvs. den må ha tildelt minneplass, diskplass, prioritet og rettigheter (forklares senere) som er nødvendig for å behandle dataene slik oppgaven krever.

**Program** Programmet er oppskriften på hvordan oppgaven skal utføres, og inneholder de nødvendige instruksjonene (ofte maskininstruksjoner) og informasjon om hvor mye minneplass som programmet krever.

## En definisjon

Her kommer vi frem til et punkt hvor det er mye uklarhet og forskjeller mellom ulike lærebøker. Denne boka bruker prosess-begrepet på denne måten:

> **En prosess er en allokeringsenhet.** Operativsystemet oppretter en prosess og tildeler den de nødvendige ressurser (tråder, program mer og minne) slik at den ønskede oppgaven kan utføres.

Tilsvarende vil vi si dette om begrepet «tråd»:

> **En tråd er en utføringsenhet.** Den hører til en prosess og utfører programkode som hører til denne prosessen. Tråder som hører til samme prosess deler prosessens ressurser.

I faglig dagligtale og i bøker råder det en viss forvirring i bruken av begrepene prosess og program. Det er ganske vanlig å si eller skrive «program» når man egentlig mener «prosess», f.eks. «kan du stoppe dette programmet?», «utskriften fra programmet ligger på bordet ditt». *Programmet er en beskrivelse av hvordan en oppgave skal utføres (matoppskriften), men det er prosessen som produserer resultatet (maten).*

## Kan vi *se* en prosess eller en tråd?

Operativsystemet må holde orden på alle prosessene som eksisterer og den minneplassen som er tildelt. Prosessene skal beskyttes mot hverandre, og de skal levere tilbake minne som de har fått tildelt når de opphører. Dette krever at operativsystemet fører et nøyaktig bokholderi over alle prosessene i systemet.

Operativsystemet viser frem prosessene i systemet på kommando. I Windows og Linux brukes kommandoene slik:

**Windows NT** «Task Manager» (høyreklikk på klokka nederst til høyre på skjermen og velg «Task Manager» fra menyen) viser deg alle prosesser i systemet, sammen med litt informasjon om hver enkel prosess.

**Windows 98** Ved tastekombinasjonen Ctrl-Alt-Del viser operativsystemet frem et vindu med navn på prosessene i systemet. Ingen andre opplysninger vises.

**Linux** Med «ps -Al» får du se alle prosessene i systemet. ps-kommandoen har mange varianter av parametre for å få se mer eller mindre informasjon om prosessene. Skriv «man ps» for å få se en oversikt over parametrene. Kommandoen «top» gir også en oversikt over prosessene i systemet.

En prosess kan som nevnt inneholde flere tråder, men operativsystemet er i ulik grad kjent med dem. Dette skyldes forhold som vi vil forklare under avsnittet “Kjernetråder og brukertråder” på side 84. Windows NT viser deg det totale antall tråder i systemet, men ikke hvor mange tråder som finnes i hver enkelt prosess. Det finnes programmer som viser slik informasjon, men som ikke er del av standardinstallasjonen av Windows NT.

Linux har en parameter i «ps»-kommandoen som viser tråder i systemet: «ps -m».

## Prosessens tilstand

En prosess er til enhver tid i en *tilstand*. Akkurat som restaurantkjøkkenet har et sett av oppskrifter som er i bruk, halvferdige matretter, bestillinger og kokker som holder på med en arbeidsoperasjon, så er prosessens tilstand bestemt av prosessens *data*, prosessens *omgivelser* og trådenes *tilstand*.

### Lagring og gjenskaping

Prosessens tilstand er viktig i de tilfeller hvor vi trenger å «fryse» en prosess og lagre den, enten i flyktig eller i permanent minne. Da er vi nødt til senere å kunne gjenskape prosessen akkurat slik den var i det øyeblikket den ble lagret.

Om du har en bærbar pc, har du trolig forsøkt å sette maskinen i «standby»-modus. Når du skrur maskinen på igjen, kommer maskinen opp med akkurat de samme prosessene i den samme tilstanden som da maskinen ble skrudd av.

I avsnittet “Avbrudd” på side 39 påpekte vi viktigheten av at tilstanden i «oppgaven» ble tatt vare på under behandlingen av avbruddet, slik at den avbrutte oppgaven kunne fullføre oppgaven med det samme resultatet som om det ikke hadde skjedd et avbrudd.

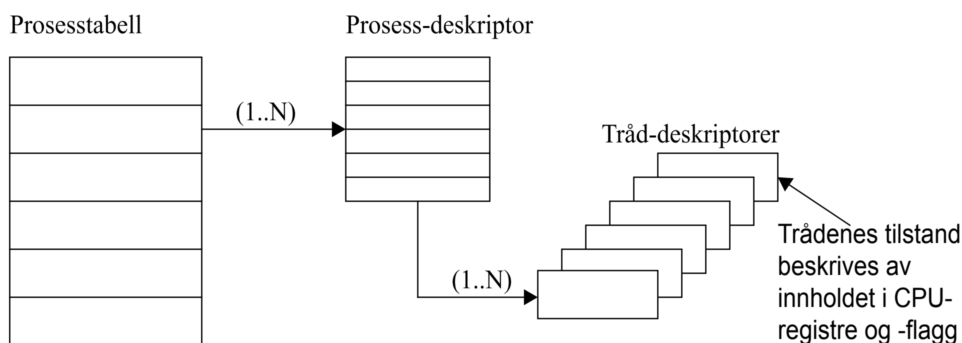
Både standby-modus og avbruddsbehandling er mulig fordi det lar seg gjøre å lagre og gjenskape tilstanden i en prosess. Vi skal nå se på det bokholderiet som er nødvendig for å få dette til.

## Prosess-deskriptorer

Hver prosess i et operativsystem er representert med et «ark» som kalles prosess-deskriptor. På dette arket står det skrevet alt som operativsystemet trenger å vite om prosessen, f.eks.:

- Prosessens navn
- Hvem (hvilken bruker) som eier prosessen
- Prosessens prioritet og rettigheter
- Opplysninger om tildelt minne
- Trådene som hører til prosessen
- Programmet som prosessen er knyttet til
- Pekere til prosess-slektninger (kalt mor-prosess og datter-prosesser)
- Akkumulert CPU-tid i denne prosessen

Deskriptorene til alle prosessene i operativsystemet er samlet i en stor datastruktur som kalles *prosesstabell*. Prosesstabellen er en av de viktigste datastrukturene i operativsystemet.




*Figur 4-1: Prosesstabell, prosessdeskriptor og tråddeskriptor*

## Trådenes tilstand

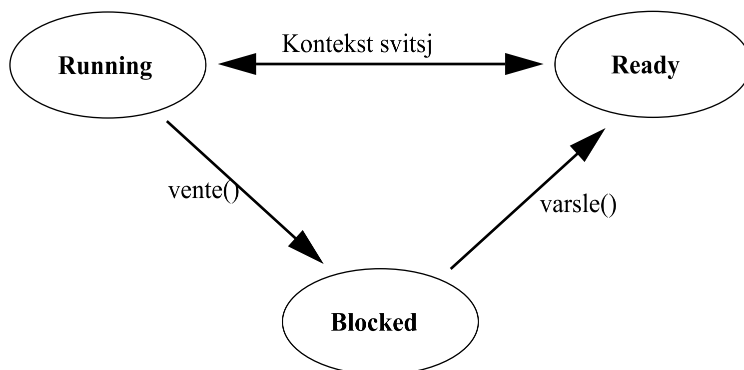
Trådene i prosessen er å sammenligne med kokkene på kjøkkenet. De er også i en tilstand som kan beskrives ut fra hvor langt de er kommet i oppskriften, og hva de har liggende av halvferdige matretter på arbeidsbenken. Tilsvarende vil tilstanden til en tråd være bestemt ut fra

- hvor i programmet tråden er i ferd med å utføre instruksjoner (instruksjonspekeren).
- hvilke verdier som ligger i CPU-ens registre.

- om tråden er klar for å utføre instruksjoner *(ready)*, om den utfører instruksjoner i øyeblikket *(running)*, eller om den står i en ventetilstand *(blocked)*. Denne verdien kaller vi for *utføringsstatus*.

Opplysningene om trådtilstandene blir oppbevart i prosess-deskriptoren til den prosessen som «eier» tråden.

**Viktig:** Trådens utføringsstatus er et meget viktig begrep. Tråden er alltid i en av tre tilstander: *running*, *ready* eller *blocked*.




*Figur 4-2: Trådens utføringsstatus og overgangen mellom tilstandene. Noen av de viste begrepene forklares senere i boka*

## Slektskap mellom prosesser

Hvordan skapes nye prosesser? Jo, det skjer ved at tråder i andre prosesser gjør kall til operativsystemets API og ber om at dette skjer. En prosess «føder» en annen, og vi betrakter dem derfor som mor og datter. Slik skapes et slektskap mellom alle prosessene i operativsystemet helt opp til stam-moderen som er øverst i slektstreet.

Det er flere grunner til at det er praktisk å holde rede på slektsforholdene mellom prosesser. En prosess som ønsker å dele sine aktiviteter opp i mange mindre deler kan gjøre dette gjennom å skape datterprosesser. Om du som bruker starter mange programmer[1] vil de resulterende prosessene være datterprosessene til den prosessen som representerer deg (eng. *user session* eller *user program*). I begge tilfellene trenger mor-prosessen å ha kontroll over datter-prosessene ved at hun kan:

1. Ja, vi mener at å «starte et program» er en riktig måte å si at vi «skaper en prosess knyttet til et gitt program» på.

- kontrollere eksistensen til datter-prosessen, og bestemme om den skal stoppes og opphøre å eksistere.
- få beskjed om når en datter-prosess har opphørt å eksistere, evt når alle datterprosesser har opphørt å eksistere.

Slektskap mellom prosesser er noe som avspeiler fordelingen av ansvar og kontroll mellom dem, og som vi derfor kan knytte spesielle rettigheter til, slik som vist.

Tilsvarende eksisterer det et slektskap mellom *trådene innenfor en prosess*. Den tråden som skaper en annen tråd (se bakgrunnsartikkelen for en forklaring på hvordan dette foregår i Java) blir moren, og den skapte tråden blir datteren.

**Viktig:** Fra et Java-program er det lett å lage nye tråder. Det er vanskelig (lite portabelt) å lage nye prosesser.

## Administrasjon og styring av prosesser og tråder

Prosessene i et operativsystem konkurrerer om CPU-ressursene. Under avsnittet “Deling” på side 6 bruker vi begrepet «Multiprogrammering» og forklarer hvordan vi deler CPU-tiden opp i små biter og lar hver oppgave få utføre sine instruksjoner en liten stund om gangen.

Fra nå av vet vi at det er *trådene* som utfører instruksjonene, og det er altså mellom dem at CPU-tiden skal fordeles. En tråd er bare interessert i CPU-tid dersom dens utføringsstatus er *ready.* Sammen med den tråden som er *running* er det disse trådene som utgjør kandidatene for å ta del i konkurransen om CPU-tiden.

### Kontekst svitsj

CPU-tiden går på «rundgang» mellom kandidatene, som er tråden med *running-* og alle trådene med *ready*-status. Operasjonen med å skifte CPU-utføringen fra én tråd (den som er *running*) og til en annen tråd (som blir *running*) kalles for en *kontekst svitsj* (eng. context switch).

**En kontekst svitsj er en operasjon som besørges av operativsystemet og som består av disse del-operasjonene:**

1. Den tråden som er *running* stoppes og trådens tilstand lagres. Trådens utføringsstatus settes til *ready.*

2 Blant de trådene som er *ready* velges en som nå skal få fortsette utføringen. Tilstanden til denne tråden gjenskapes og utføringsstatus settes til *running.*
3 Kontekst svitsj er nå fullført og den tråden som har fått *running* utføringsstatus fortsetter sin utføring på CPU-en.

### Kjøreplanen

Operativsystemet har en *politikk* for å fordele CPU-tid mellom trådene. Vi bruker ordet *kjøreplan* for å betegne den algoritmen som brukes for å gjennomføre kontekst svitsj. Under designet av en kjøreplan må man ta stilling til disse spørsmålene:

**Når skal kontekst svitsj finne sted?** Vi deler alternative kjøreplan-algoritmer opp i to grupper:

- En *ikke-preemptiv* kjøreplan vil utføre en kontekst svitsj fordi den utførende tråden (*running*) selv ber om det. Den utførende tråden må være «høflig» og gjøre dette for å la andre tråder «slippe til». Dersom tråden velger å monopolisere CPU-en ved aldri å be om kontekst svitsj, vil ingen andre tråder få fortsette sin utføring, og systemet vil «henge seg opp».
- En *preemptiv*[2] kjøreplan vil benytte avbrudd fra en timer (les om “Avbrudd” på side 39) til å fremtvinge en kontekst svitsj til bestemte tider, f.eks. 50 ganger pr. sekund. Vi unngår dermed problemet med «usolidariske tråder», og er sikret en bedre kontroll med fordelingen av CPU-tid mellom trådene. I dette tilfellet vil koden i avbruddsrutinen kalle på kjøreplanens funksjoner for å skape en kontekst svitsj. Se også “Avbruddshåndtering” på side 80.

De fleste generelle operativsystemer bruker en preemptiv kjøreplan. I enkle systemer hvor alle trådene er laget for å løse en felles oppgave (f.eks. i en mobiltelefon) er det praktisk å benytte ikke-preemptiv kjøreplan.

**Hvordan skal en tråd plukkes ut for å settes running?** I de tilfeller hvor det er mange tråder som er *ready* må det foreligge noen utvelgelseskriterier som er rettferdige og effektive. Kjøreplanen må finne en balanse mellom hensynet til hver enkelt tråd og til systemet som helhet. Kriterier som kan tenkes brukt er:

- *Prioritet*. En tråd kan være tildelt en relativ prioritet som et uttrykk for hvor viktig den deloppgaven er (som utføres av tråden) i forhold til andre tråder i systemet. Blant de trådene som er *ready* kan den foretrekkes som har den høyeste prioriteten, eller at CPU-tiden fordeles mellom trådene i proporsjon med deres prioritet.

2. Eng. preempt = avbryte.

- *Akkumulert kjøretid*. Man kan føre et regnskap over hvor mye hver tråd allerede har forbrukt i form av CPU-tid, og så foretrekke den tråden som har forbrukt minst. Slik kan man hindre at store oppgaver (som kanskje burde kjøres om natten) ikke forsinker de små, interaktive oppgavene (som kjøres av brukere i arbeidstiden).
- *Gjenværende kjøretid*. På samme måte kan man foretrekke småjobber fremfor de store jobbene. Denne varianten krever at hver tråd deklarerer sitt CPU-behov på forhånd, og det er ikke alltid realistisk.

Bruk av prioritet er en populær og enkel måte å kontrollere fordelingen av CPU-tid på. I praksis finner vi at både tråden og den prosessen den tilhører er gjenstand for en prioritering, slik at utvelgelsesprosessen baserer seg på egenskaper både ved tråden og prosessen. I noen tilfeller finner vi også at det eksisterer en preemptiv kjøreplan for å fordele CPU-tiden mellom prosesser, og en ikke-preemptiv kjøreplan for å fordele tiden mellom trådene som hører til samme prosess.

Under Linux (og Unix) finner vi den varianten at tråder som utfører inne i systemkall ikke blir avbrutt av en kontekst svitsj, men at tråder som utfører brukerkode, blir det. En kombinasjon av preemptiv og ikke-preemptiv kjøreplan altså, hvor user/supervisor mode (se “User mode/supervisor mode” på side 60) inngår som en parameter i kjøreplanen.

## Avbruddshåndtering

Vi har lest om avbrudd i avsnittet “Avbrudd” på side 39. Der leste vi hvordan et elektrisk signal tvinger CPU-en til å avbryte sin ordinære instruksjonsrekkefølge og hoppe til et stykke programkode som kalles en *avbruddsrutine*. Adressen til avbruddsrutinene ligger i tabellform på en kjent adresse, og denne tabellen kalles en avbruddsvektor.

> **Viktig:** Et avbrudd er ikke det samme som en kontekst svitsj! En avbruddsrutine er ingen egen prosess!

Når vi skal forstå hvordan avbruddshåndtering skal gå sammen med styring av tråder og prosesser, så trenger vi å forstå dette begrepet:

> **Prosess-kontekst:** Den prosessen som eier den tråden som for øyeblikket er running

All utføring av instruksjoner i et operativsystem må skje i en prosess-kontekst. Under utføring av en avbruddsrutine «låner» operativsystemet prosess-konteksten til den prosessen som er blitt avbrutt. Dette

betyr at «din» prosess gjerne utfører operativsystemfunksjoner som ikke har noe med dine programmer å gjøre i det hele tatt, men som har å gjøre med helt andre oppgaver i maskinen.

Om denne avbruddsrutinen ønsker å endre dataene i programmet ditt, eller endre din instruksjonsrekkefølge, er det ingenting i veien for det, siden avbruddsrutinen «ser» alle dataene som ligger i din prosess. Det samme gjelder filer, nettverksforbindelser o.l. En avbruddsrutine vil normalt ikke ha dette som noen hensikt, men en dårlig skrevet avbruddsrutine kan komme til å endre dataene dine *i vanvare* og representere en trussel mot kvaliteten og sikkerheten i operativsystemet.

For å unngå dette må avbruddsrutiner skrives etter bestemte regler og testes grundig.

**De hovedreglene som må følges er:**

1. Avbruddsrutinen må starte sin utføring med å lagre tilstanden til den avbrutte prosessen (inkludert trådtilstandene)
2. Avbruddsrutinen må under utføring ikke berøre ressursene til den avbrutte prosessen, men må bruke separate minneceller for sitt lagringsbehov
3. Den må sette tilbake tilstanden til den avbrutte prosessen når den er ferdig med sin utføring, og med en avbruddsretur returnere kontrollen tilbake til prosessen

Vi har lært at prosess-deskriptoren (se side 76) kan holde rede på prosess-status. En avbruddsrutine kan lagre status her i forbindelse med steg 1, men dette er ikke nødvendig. Normalt vil en avbruddsrutine kun trenge å lagre verdien i CPU-ens registre og flagg, alle andre deler av prosessens status kan avbruddsrutinen klare å holde seg unna. Bruk av CPU-ens registre er derimot en forutsetning for å få utført instruksjoner, og derfor må innholdet av disse lagres.

**Utføres raskt** Utføringen av en avbruddsrutine må være kortvarig og består som regel av ganske enkle operasjoner: Flytting av data mellom minnebufre, betjening av i/o-kretser (f.eks. start av i/o-operasjoner), og signalering til operativsystemet om endring i utføringsstatus for en tråd. Kompliserte OS-funksjoner, f.eks. skriving til skjerm eller fil, forekommer aldri. Minnebehovet til en avbruddsrutine er gjerne statisk, slik at nødvendig minneplass kan settes av når avbruddsrutinen lastes inn i minnet (ved systemstart).

Utføringen av en avbruddsrutine foregår alltid med CPU-en i «supervisor mode», der CPU-en har en slik tilstand. Dette er nødvendig fordi avbruddsrutinen krever adgang direkte til maskinvarekretsene og til minneområder som er avstengt for vanlige brukerprogrammer.

Ved avslutning av utføringen vil avbruddsrutinen sette det opprinnelige innholdet tilbake i CPU-registrene og så utføre en «return from interrupt»-instruksjon. Da fortsetter den avbrutte prosessen sin utføring.

**Programmeres i assemblerspråk** Fordi en avbruddsrutine må utøve nøyaktig kontroll over CPU-registrene, sitt eget statiske minneområde og dessuten utføre spesielle instruksjoner, er det vanlig at avbruddsrutinen er skrevet i assemblerspråk (symbolske maskininstruksjoner). Programmeringen krever i tillegg detaljert kunnskap om grenseflaten mellom operativsystemet og avbruddsrutiner, og detaljert kunnskap om den maskinvaren som skal behandles. Programmering av avbruddsrutiner regnes derfor som en krevende programmeringsoppgave.

## Kjernetråder vs. brukertråder

Vi har tidligere introdusert begrepet kontekst svitsj, som har litt forskjellig betydning for tråder og for prosesser.

**Kontekst svitsj for prosesser** foregår ved at den prosessen som har den tråden som er «running» får sin prosess-deskriptor oppdatert (med tilstand og akkumulert ressursforbruk), og den kjørende tråden får satt sin kjøretilstand til «ready». Deretter plukkes det en prosess blant dem som har tråder i «ready» kjøretilstand, og denne får satt en av trådene i «running»-tilstand. I den grad operativsystemet støtter virtuelt minne (forklares i et senere kapittel) kan det også være aktuelt å «swappe» minneområdet til den nye aktive prosessen inn i minnet fra disk.

**Kontekst svitsj for tråder** foregår ved at den tråden som har «running» kjørestatus blir satt til «ready». En annen tråd (i samme prosess) som er «ready», blir satt til «running». Prosess-deskriptoren blir ikke oppdatert, fordi det ikke skjer noe skifte mellom prosessene. Heller ikke er det aktuelt med noen «swapping» av minneområder.

Kontekst svitsj av tråder og prosesser innebærer altså ganske forskjellige operasjoner, og de kan gjerne utføres helt uavhengig av hverandre.

I operativsystemer som ikke støtter flere tråder inne i en prosess (bl.a. tidligere versjoner av Linux) kan kontekst svitsjing av tråder overlates til programmets kjøremiljø (se “Kjøremiljø” på side 65). Ved denne organiseringen er trådene ukjent for operativsystemet, operativsystemet ser på prosessen som den primære utførende enhet, og utføring-

stilstandene «ready», «blocked» og «running» knyttes direkte til prosessen. I slike systemer må all administrasjon av trådene (skaping, sletting, kontekst svitsjing osv.) foregå i programmets kjøremiljø.

Uttrykket «programmets kjøremiljø» innebærer at slike løsninger må programmeres i et språk som støtter slike løsninger. Java er et eksempel på et slikt språk. Tråder som styres på dette viset kalles *brukertråder* (eng. user-level threads).

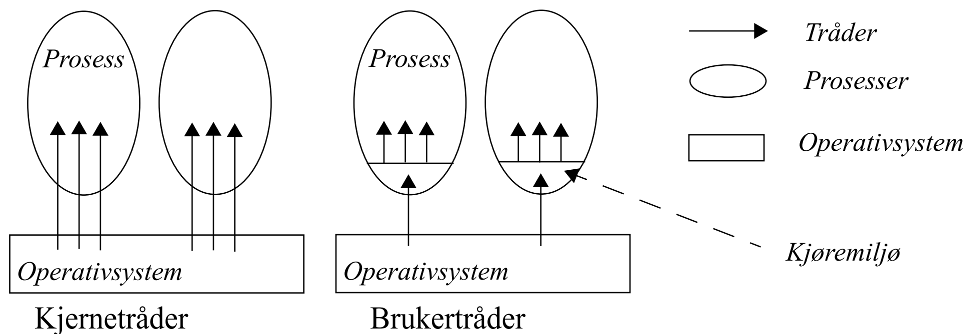
I de fleste av dagens operativsystemer, derimot, ligger styringen av trådene inne i operativsystems kjerne. Slike tråder kalles derfor *kjernetråder.*

| Kjernetråder | Brukertråder |
|---|---|
| Kontekst svitsj i operativsystemkjernen | Kontekstsvitsj i kjøremiljøet |
| Blokkeringsfrie | Ikke blokkeringsfrie |

Kjernetråder har den store fordel at de er *blokkeringsfrie*.

**Blokkeringsfrie tråder** betyr at trådene inne i en prosess kan ha ulik utføringsstatus: Én kan være *running*, noen *ready* og andre *blocked*. De trådene som er blokkert venter f.eks. på at en i/o-operasjon skal fullføres. Derfor er det operativsystemet som gjennom en avbruddsrutine må flytte tråden fra *blocked*-tilstand til *ready*-tilstand når dette skjer. *Men det er bare mulig dersom operativsystemet vet om hver enkelt tråd, dvs. at de er kjernetråder!*

I et system med brukertråder vil dette foregå på en annen måte: En tråd som ber operativsystemet om en i/o-operasjon (kanskje lese data fra nettverket) vil medføre at selve prosessen som inneholder brukertråden blir satt til blocked. Operativsystemet vet ikke om noen annen utføringsenhet enn prosessen, så den har egentlig ikke noe valg. Nå kan det tenkes at programmet setter opp mange tråder som er tiltenkt å være aktive også mens den ene tråden venter på i/o, men istedet vil alle brukertrådene fryse til i/o-operasjonen er fullført. Dette er en ganske uheldig situasjon, og nytteverdien av å bruke tråder som ikke er blokkeringsfrie er begrensede.




*Figur 4-3: Kjernetråder og brukertråder*

### Init-prosessen

Dersom en prosess blir til ved at den blir født av en annen prosess, hvem er da «Eva» blant dem? Det er en prosess som er stam-mor til alle de andre, og som lages «direkte» av bootstrap-prosessen (trinn 5 under “Bootstrap” på side 57). Denne prosessen kalles «init» under Linux og «idle process» under Windows 2000, og har flere oppgaver:

**Init-prosessens oppgaver er å**

1. lese konfigurasjonsfiler og starte andre prosesser, f.eks. innloggingsbildet
2. stå «bakerst i køen» blant de prosessene som har en *ready*-tråd, slik at det alltid er en kandidat for kjøreplanen (dette gjør kjøreplan-algoritmen enklere).

## Tråder i Java

I Java er en tråd representert av et objekt av klassen `java.lang.Thread`. For å skape ekstra tråder i programmet må vi skape objekter av denne klassen, ev. av en klasse som arver fra `Thread`. Når tråden skal startes må den ha noen instruksjoner som den kan utføre, og det finner den på én av to måter:

- Thread-objektets konstruktør kan ta en parameter i form av et objekt som implementerer interfacet `Runnable`. Dette objektets klasse må ha definert metoden `void run()`, og denne metoden er det som den nye tråden starter opp i.

- Dersom tråden representeres av et objekt som arver fra `Threads`, kan det definere sin egen `void run()`-metode. I dette tilfellet skal konstruktøren være parameterløs.

I begge tilfeller startes tråden med et kall til `Thread`-objektets `start()`-metode. Objektet har en rekke metoder som vi skal omtale i senere kapitler, spesielt i forbindelse med synkronisering.

Programeksemplet i listing 4-1 deklarerer et GUI-element som viser tiden som en løpende digital klokke. Hovedprogrammet går som vanlig og venter på hendelser i brukergrensesnittet, mens en annen tråd oppdaterer tekstfeltet med riktig klokke. For å få dette til har vi fulgt reglene som ble nevnt ovenfor:

- Vi oppretter et objekt av klassen `Threads`. En parameter til klassens konstruktør angir det objektet som skal være "vert" for tråden, dvs. dele sine data og sin kode med tråden. Her har vi brukt `this`, derfor er det altså klassen `List1` som inneholder programkode som denne tråden skal starte med å utføre.
- Klassen som skal være vert for tråden implementerer `Runnable`. Det er metoden `void run()` som kalles når vi kaller `start()`-metoden på `Threads`-objektet.

```
import java.awt.TextField;
import java.util.Date;
public class List1 extends TextField
    implements Runnable {
  public List1() {
    super(25); // Kaller TextField-konstruktøren
    Thread t1 = new Thread(this);
    t1.start();
  }
  public void run() {
  // Her starter utføringen av tråden
    while (true) {
      Date d0 = new Date();
      // Finn riktig tid of konvertere til String
      setText(d0.toString());
      try {
          // Sov i ett sekund og repetere
          Thread.sleep(1000);
```

```
            } catch (InterruptedException e) {}
        }
    }
}
```

*Listing 4-1: En Java-klasse som lager en løpende klokke*

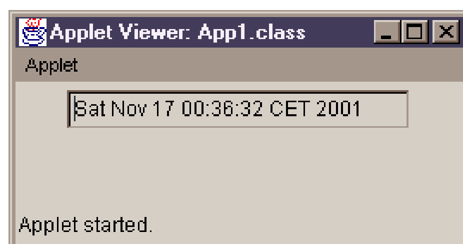
Legg merke til metodekallet `Thread.sleep(1000)`. Dette kallet setter den kallende trådens utføringsstatus til *blocked*, deretter tilbake til *ready* etter 1000 millisekunder. Når tråden etter en påfølgende kontekst- svitsj blir satt til *running,* fortsetter utføringen med neste programlinje. Programteknisk er det slik at `sleep(1000)`-kallet først returnerer ett sekund etter at det blir kallet.

Om vi nå ønsker å bruke klassen `List1` som en digital klokke i et GUI-basert program kan vi lage en liten Java-applet som bruker `List1` som et tekstfelt:

```
/*<applet code=App1.class height=100
  width=300></applet> */
import java.awt.*;
public class App1 extends java.applet.Applet {
  public void init() {
    add(new List1());
  }
}
```

*Listing 4-2: Bruk av klokkeklassen List1*

Og når vi kjører denne Appleten i AppletViewer får vi se følgende bilde:




*Figur 4-4: Kjøring av Appleten App1*

Klokken oppdateres løpende i tekstfeltet, samtidig som appleten også kan lytte til hendelser i brukergrensesnittet eller gjøre andre oppgaver, fordi oppdateringen av klokken er skilt ut og betjenes av en separat tråd.

Så enkelt er det å lage frittløpende Java-tråder. I mange sammenhenger vil det derimot ikke passe å la trådene løpe fritt, men man vil trenge å la en tråd holdes tilbake inntil en eller annen tilstand i programmet er oppfylt. Dette temaet vil vi behandle i kapittel 6 – Synkronisering I.

Ethvert Java-program kjører med flere tråder. Selv om programmereren selv ikke benytter seg av dette, vil oppgaver som *Garbage Collection* utføres i egne tråder.

## Bakgrunnsartikkel: *Hvorfor er Java-tråder så nyttige?*[3]

Når du starter en av de to vanlige nettleserne, og venter på en side som ikke er tilgjengelig av en eller annen grunn, da kan du korte ventetiden ved å trykke på "stopp"-knappen. Derimot, om du starter "telnet" fra Windows, og den vertsmaskinen som du kaller på er ute av drift, da må du pent sitte og vente på feilmeldingen før du får gjøre et nytt forsøk. Dette er egentlig ganske treigt, og vi ønsker noe mer av et moderne program, synes vi.

Men hva ligger denne forskjellen i? Jo, i det faktum at en nettleser kjøres i flere samtidige utførende enheter, kalt *tråder* . På samme måte som du kan ha flere programmer liggende på arbeidsflaten din som utfører sine oppgaver helt uavhengig av hverandre, slik kan vi også inne i et program ha flere aktiviteter som utføres samtidig, men ikke fullt så uavhengig av hverandre.

Tråder er et begrep som har eksistert i cirka 10 år. I tradisjonell operativsystemteknikk har vi i 35 år hatt begrepet prosesser, som et uttrykk for en utførende aktivitet som skal være beskyttet mot andre aktiviteter i maskinen, den skal ha tildelt private ressurser og typisk representere arbeidsoppgavene til én bruker. Det typiske operativsystemet av den tradisjonelle sorten har gjerne tungvinte mekanismer for å tillate deling av ressurser mellom prosesser (f.eks. minneområde), eller ingen mulighet for deling overhodet.

En slik inndeling av aktivitetene i isolerte prosesser var en adekvat løsning i mange år. Den gangen var målet å tilby brukeren "virtuelle maskiner", hvor nærværet av andre brukere skulle merkes minst mulig.

---

3. Dette avsnittet er første del fra en artikkel i tidsskriftet *PC World Nettverk* nr. 5/1999, skrevet av Anders Fongen.

Den typiske applikasjonen på disse maskinene benyttet kun lokalt lager og lokal prosessorkraft, og dersom det var lang responstid på maskinen, da var det ikke noe annet å ta seg til enn å vente.

I nettverk, og i distribuerte applikasjoner, der forholder dette seg annerledes. Mange applikasjoner benytter seg nå av en fjernressurs, f.eks. i form av en e-posttjener. Dersom denne tjeneren er ute av drift, eller er utilgjengelig i nettverket, da er det mange andre ting brukeren kan gjøre mens hun venter: Hun kan jobbe på et regneark, eller lese nettsider. Dette kan vi oppnå i et tradisjonelt operativsystem uten bruk av tråder, dersom alle disse oppgavene betjenes av programmer med hver sine prosesser. Men det programmet som står og venter på svar fra en tjener, det er låst fast, noe som hindrer brukeren i f.eks. å lese tidligere meldinger fra innkurven. *Dette er ikke godt nok.*

Et moderne program vil derfor i stor grad ha nytte av å dele sine oppgaver på uavhengige samtidige aktiviteter, slik vi innledet med å si: Mens én tråd venter på respons fra nettverket, kan en annen tråd lytte etter hendelser i brukergrensesnittet, f.eks. trykking av stopp-knappen, eller beskjed om å hente data fra et annet sted.

En dyktig programmerer kan langt på vei lage programmet i flere tråder uten at operativsystemet støtter dette. Operativsystemet kan bare "se" én eneste aktivitet i programmet, mens programmet selv lager "aktiv-itetsobjekter", og skaper en viss uavhengighet mellom dem ved å la dem utføre sine oppgaver på skift. I enkelte programmeringsspråk er aktivitetsobjekter "standard" (ofte kalt co-rutiner). Et problem med en slik programmeringsmodell er at ved blokkerende kall til operativsys-temet som f.eks. sette opp en nettverksforbindelse, vil operativsys-temet blokkere den eneste aktiviteten som den kan "se", og denne aktiviteten rommer alle trådene i programmet. Vi oppnår altså ikke det som vi kaller for blokkeringsfrie tråder, og dette er en nødvendig egenskap i den typen programmer vi her har med å gjøre.

Vi trenger derfor operativsystemer som "ser" hver enkelt tråd i alle programmene, og kan ta ansvar for hvordan de skal utføres. Mange slike operativsystemer er kommet på markedet, og de mest kjente er WindowsNT, Windows95, OS/2, og nyere versjoner av UNIX og Linux. Disse kaller vi for flertråds operativsystemer, fordi de skaper et skille mellom begrepene utføring og beskyttelse .

## Utføring og beskyttelse

Også i et flertråds operativsystem vil det være et begrep om en "prosess", i form av tildelte og beskyttede ressurser (med ressurser mener vi f.eks. minneområde, kjørekvoter og kjøreprioritet). Ordet prosess blir dermed mer knyttet til ressurtildeling enn til utføring. Tilhørende denne prosessen lages det en tråd, som er et begrep som beskriver en utførende aktivitet.

På denne måten vil et program som startes under et flertråds operativsystem "se" et tradisjonelt kjøremiljø: Én utførende enhet, og ett sett av tildelte og beskyttede ressurser. Programmet kan nå inneholde kode som skaper nye tråder , og disse trådene vil utføre sine oppgaver samtidig med "mortråden" og dele dennes ressurser, bl.a. vil den ha adgang til de samme minneområdene som mortråden. Beskyttelsen mellom trådene er svak, derfor er dette en programmeringsmodell som egner seg for *samarbeidende* aktiviteter, ikke konkurrerende.

Dessuten er ressursfordelingen mellom trådene innenfor samme prosess enklere enn mellom tråder fra forskjellige prosesser. Av og til ser vi endog at det benyttes en ikke-preemptiv metode for å skifte mellom kjørende tråd, hvor flytting av utføringen fra en tråd til en annen skjer "frivillig", dvs. at operativsystemet ikke tvinger frem et skifte.

### Portabilitet

Om vi programmerer med flere tråder i programmet vårt, vil vi gjerne støte på visse problemer knyttet til portabiliteten av koden mellom ulike operativsystemer. De krever litt ulike oppsett for å gjøre dette, slik at koden må skrives om når den skal flyttes. Hvor mye, det avhenger av hvilket programmeringsspråk som er brukt, og til hvilken grad operativsystemet følger POSIX 1003.x-standardene.

Men det er Java-tråder vi bryr oss om nå. I Java er trådene representert som "standardobjekter" (fra klassebiblioteket), og det er helt entydig hvordan de opprettes og brukes, uansett hva slags operativsystem som ligger under Java-systemet. Derimot er det visse egenskaper ved utføringen som kan endres fra sted til sted, f.eks. om det benyttes preemptiv eller ikke-preemtpiv metode for å skifte mellom utførende tråder.

Derfor er det lett å lage et flertråds program i Java. Og vanskelig. Forbausende enkelt og vanskelig på samme tid. For mens det er enkelt å starte utføringen av flere tråder, innebærer denne programmeringsteknikken at det er del nye mulige feilsituasjoner å forholde seg til.

## Sammendrag

- En prosess er en enhet for *allokering*, en tråd er en enhet for *utføring*.
- En prosess eier ressursene som brukes av tråden. En tråd tilhører alltid en prosess.
- Operativsystemet holder alle opplysninger om en prosess lagret i en *prosess-deskriptor*.
- En tråd kan ha utføringsstatus *blocked, ready* eller *running.*

- Skifte av running-status skjer i form av en *kontekst svitsj* og utføres av *kjøreplanen*. Vi skiller mellom *preemptiv* og *ikke-preemptiv* kjøreplan.
- En avbruddsrutine låner *prosess-kontekst* til den avbrutte prosessen, og må passe på å ikke endre den avbrutte prosessens tilstand under utføringen.
- Kun kjernetråder er blokkeringsfrie.
- Tråder er støttet i Java i form av objekter av klassen `Thread`.

| **Sentrale begreper i dette kapitlet:** | |
|---|---|
| Prosess, tråd | Allokering, utføring |
| Utføringsstatus | Avbrudd, avbruddsvektor |
| Prosess-deskriptor | Kjøreplan |
| Prosess-kontekst | Preemptiv/ikke-preemptiv |
| Brukertråder/Kjernetråder | java.lang.Thread |

# Teorioppgaver

**Gå sammen i grupper og løs disse oppgavene:**

1. Tegn en skisse til en datastruktur som organiserer prosess-deskriptorene og tråd-deskriptorene slik at kjøreplanen raskt kan utføre en kontekst svitsj.
2. I avsnittet om trådenes tilstand (side 76) skriver vi «Hvor i programmet tråden er i ferd med å utføre instruksjoner«. Er det tråden eller CPU-en som utfører instruksjoner. Forklar denne tilsynelatende selvmotsigelsen.
3. Lag en skisse til et Java-program som kan hjelpe deg til å avgjøre hva slags kjøreplan operativsystemet bruker (preemptiv eller ikke-preemptiv).
4. Diskuter om andre kjøreplan-algoritmer enn de som er nevnt i dette kapitlet kan være fornuftige. Kan vi f.eks. tenke oss høyere prioritet for prosesser under *deler* av utføringen?
5. En utskriftskø trenger ikke nødvendigvis å skrive ut jobbene i den rekkefølge de ankom køen. Hvilke kriterier kan legges til grunn for å avgjøre hvilken utskriftsjobb som skal være den neste som blir startet?

6 Tenk deg at en avbruddsrutine selv blir avbrutt. Går det bra, eller kan det oppstå feilsituasjoner som følge av dette?
7 Kan vi oppnå blokkeringsfrie brukertråder om vi skriver alle avbruddsrutinene selv (hypotetisk spørsmål)?

## Øvingsoppgaver

Forslag til øvingsoppgaver ligger på bokas nettsted.

**Etter fullførte øvinger bør du beherske følgende:**

1. Skrive Java-programmer som utfører i «frittløpende» tråder som vist i eksemplene.
2. Studere Java-tråder med monitoreringsverktøy («task manager» og ps-kommandoen).
3. Bruke prioritet i Java-tråder for å påvirke kjøreplanen.
4. Fastslå om maskinen bruker preemptiv eller ikke-preemptiv kjøreplan.
5. Skrive programmer som demonstrerer hvordan Java-tråder deler minneplasser (variabler) og andre ressurser (f.eks. åpne filer), og hvordan hver Java-tråd kan ha sine egne private data.
6. Skrive programmer som bruker Thread-klassens `yield()`-metode for å skape kontekst svitsj.

Kapittel 5

# Minnestyring

*Dette kapitlet vil omhandle oppgavene som operativsystemet har i forbindelse med administrasjon av det flyktige lageret i maskinen. Vi skal omtale tildelings- og beskyttelsesteknikker, og se hvordan disklageret kan brukes til å utnytte internminnet bedre.*

## Behov og prinsipper

Under utføringen av et program er det nødvendig å sette av en viss mengde minneceller for å lagre innholdet av:

- de variablene og objektene som programmet bruker, inkludert objekter instansiert fra klassebibliotekene
- variabler brukt i programmets kjøremiljø (side 64)
- programkode for brukerprogrammet, standardklasser og kjøremiljøets kode

Det er noe spesielt med minnet i maskinen. Kun når data eller programkode ligger i minnet kan det sees av CPU-en (se "CPUens interne organisering" på side 26). Instruksjoner som skal utføres, eller data som skal behandles (beregnes, sammenlignes etc.), må ligge i minnet for at dette skal være mulig. Lagring, derimot, kan vi oppnå også andre steder i maskinen, f.eks. på en magnetisk disk.

Størrelsen på det installerte minnet i maskinen er ikke uendelig stort, og det er mulig å benytte disklageret som et supplement til minnet for at det skal se større ut. Dette kalles *virtuelt minne* og omhandles senere i kapitlet.

Operativsystemets oppgave i forbindelse med minnestyringen er å

- administrere minnet i maskinen slik at det blir rettferdig fordelt, og slik at datamaskinen arbeider så effektivt som mulig (styring)
- kontrollere hvilke områder som skal være tilgjengelig for flere (deling)

- tilby virtuelt minne der det er mulig (abstraksjon)

### Hva mener vi med «minne»?

Med *minne* mener vi det elektroniske, flyktige minnet i maskinen (ofte kalt Random Access Memory, RAM) som kan brukes av CPU-en til lagring av data og instruksjoner. Denne definisjonen utelukker:

- CMOS-basert minne i en pc som lagrer konfigurasjonsopplysninger
- Permanent skrivebeskyttet minne (Read-Only Memory, ROM)
- CPU-cache (level I og II)
- RAM installert på skjermkortet eller diskkontrolleren

### Beskyttelse og deling

Minnet tildeles ikke et program, men den *prosessen* som får oppgaven med å utføre dette programmet. Med tildeling mener vi også «reservasjon» i den forstand at prosessen kan være trygg på at andre prosesser ikke kan lese eller endre innholdet i de tildellte minnecellene (jf. avsnittet “Deling” på side 6).

Begrunnelsen for slik beskyttelse er at det skal være mulig å behandle private og konfidensielle data på maskinen. I tillegg ønsker man å forhindre at programfeil i én prosess skal forstyrre resultatet av en annen prosess. Et typisk resultat av en programfeil er at tilfeldige minneceller får innholdet sitt endret fordi programinstruksjonene blir feiltolket. Dette kan skje med alle de minnecellene som prosessen har adgang til å skrive til.

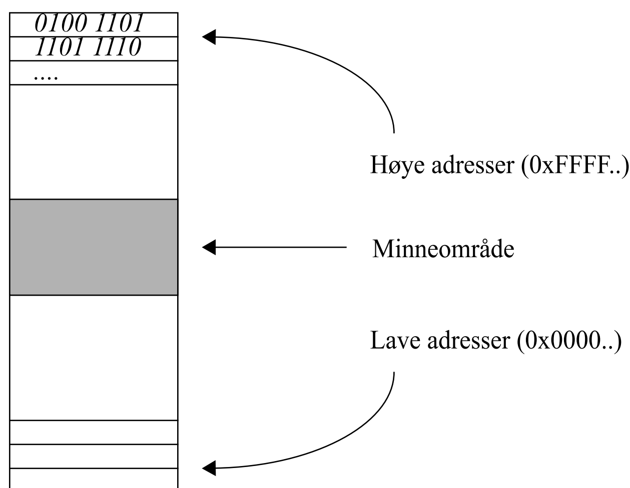
I et system uten minnebeskyttelse (f.eks. MS-DOS) kan til og med operativsystemet selv få ødelagt sine instruksjoner og data ved en feil i et anvenderprogram. I moderne operativsystemer vil vi begrense konsekvensene av programfeil til å gjelde prosessens tildelte minneområde. I praksis bedrer dette stabiliteten til maskinen betydelig.

Samarbeidende prosesser, derimot, vil ønske å utveksle data seg i mellom gjennom å ha et minneområde som er felles for dem alle. *Deling* av minneområder mellom prosesser er derfor også en av oppgavene til operativsystemet.

Deling og beskyttelse kan virke som motsetninger, men deling betyr ikke det samme som fravær av beskyttelse: Deling av minneområder skal være avgrenset, dvs. bestemte minnområder skal være delt mellom bestemte prosesser. Det kan også være ønskelig å la noen prosesser få lov til å *lese* innholdet av minnecellene, men ikke *endre* dem.

## Relokasjon

Når operativsystemet tildeler minne til en prosess, skal det være likegyldig for prosessen hvor (på hvilke adresser) dette minnet ligger. Vi tenker oss gjerne minnet i en maskin organisert som en søyle av minneceller som er organisert etter stigende adresser.




*Figur 5-1: Minnet vist som en søyle av minneceller*

I et slikt diagram kan en serie av minneceller fremstilles som et areal innenfor søylen, slik som vist på figur 5-1. Disse minnecellene ligger på en serie av fortløpende adresser, noe som gjør det velegnet for å lagre strukturerte data, som f.eks. en stakk eller et array (tabell). Vi skal bruke denne typen tegninger når vi kommer til diskusjonen om tildelingsstrategier.

Når et minneområde skal tildeles en prosess, trenger vi å lete etter et passende stort ledig minneområde hvor som helst i søylen der det er ledige minneceller. Dette krever at prosessen benytter *relokerbar kode* og *relokerbare data*. Relokerbarhet (også kalt relokasjon) betyr altså **uavhengig av plassering i minnet, uavhengig av besteme minneadresser**. Hvordan vi oppnår dette skal vi se i neste avsnitt.

## CPU-ens adresseringsmekanismer

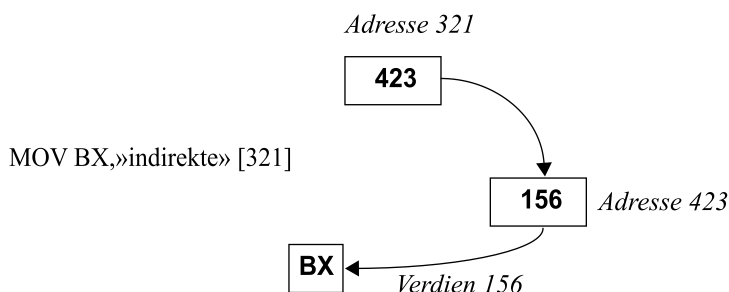
Når en instruksjon i CPU-en vil lese eller skrive en minnecelle, må denne minnecellens adresse angis på adressebussen. Minnecellens adresse kan beregnes på mange forskjellige måter. Det er CPU-ens *adresseringsmekanismer* som avgjør hvordan dette skal foregå. De vanligste alternativene er:

**Direkte adressering** Der hvor minnecellens «virkelige» adresse inngår som en del av instruksjonen. Et eksempel på direkte adressering er instruksjonen:

```
MOV AX,[1433]
```

hvor verdien av minnecelle nr. 1433 (heksadesimalt) blir lastet inn i register AX.

**Indirekte adressering** Der hvor minnecellens adresse ligger som *innhold i en annen minnecelle* snakker vi om indirekte adressering. Dette er omtrent som en peker i C/C++:




*Figur 5-2: Indirekte adressering*

Intel-CPU av 8086/Pentium-familien støtter ikke indirekte adressering.

**Indeksert adressering** Vi kan adressere minneceller gjennom en adresse som ligger i et CPU-register. Med instruksjonen:

```
MOV CX,[BX]
```

mener vi «flytt innholdet av den minnecellen som har adresse lik innholdet av register BX, inn i register CX». På denne måten kan vi kalkulere en adresse, legge verdien inn i (i dette tilfellet) BX og deretter lese eller endre innholdet i minnecellen som har denne adressen. Bruk av arrays (tabeller) benytter seg av slik adressering.

Som en variant av den viste instruksjonen, kan vi på en Intel-CPU også skrive:

```
MOV CX,[BX+2455]
```

I dette tilfellet blir verdien av BX addert til tallet 2455 for å kalkulere adressen til den aktuelle minnecellen. BX blir i dette tilfellet brukt som et *indeksregister*.

**Relativ adressering** I forbindelse med hopp i instruksjonsrekkefølgen er det vanlig å angi det stedet man ønsker å hoppe til (adressen til den neste instruksjonen) som en adresse *relativt til* den instruksjonen som nå utføres (verdien av *instruksjonspekeren*). Vi kan gjerne betrakte relativ adressering som en variant av indeksert adresering, hvor instruksjonspekeren brukes som indeksregister.

I en Intel-CPU benytter *betingede hopp* (hopp som resultat av en test) alltid relativ adressering. Relativ adressering vil sørge for at hoppet lander på riktig sted uansett hvor i minnet koden befinner seg.

## Beskyttede registre

Indeksregistre og relativ adressering er nøkkelfaktorer for å oppnå relokasjon. Dersom alle minnereferansene til data og instruksjoner skjer relativt eller indeksert er det mulig å plassere programmer og data hvor som helst i minnet, bare man sørger for at indeksregisteret får den riktige verdien under utføring av programmet.

I en Intel-CPU skjer all adressering relativt til ett av fire *segmentregistre*. I MS-DOS er det mulig at programloaderen setter opp verdien av segmentregistrene etter at det er bestemt hvor i minnet programmet skal plasseres, og at programmet siden aldri endrer verdien av segmentregistrene. Dette er tilfellet med programmer som har «.com» i etternavnet på filen (f.eks. «command.com»). Disse programmene er relokerbare, men har så mange begrensninger knyttet til seg (maksimum 64kB kode og 64kB data) at de kun er egnet for små og enkle programmer.

Programfiler som har «.exe» i etternavnet er også relokerbare, men baserer seg på at programloaderen endrer litt på programkoden for at programmet skal «passe inn» i det minneområdet som er tildelt programmet[1]. Slike programmer har en mer avansert organisering av minnet og kan håndtere større mengder data og kode i samme program.

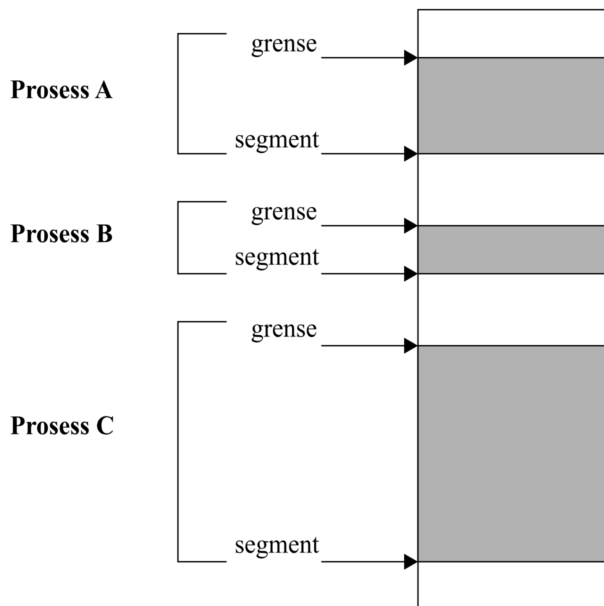
Segmentregistrene er synlige og ubeskyttede for programmet, som altså både kan lese og endre innholdet av dem. Minnestyringen i MS-DOS blir dermed avhengig av at programmene ikke «saboterer» ved å endre innholdet i segmentregistrene. Dette gjør MS-DOS sårbart for programfeil.

---

1. MS-DOS har ikke et prosess-begrep, derfor blir det ikke riktig å si at minnet tildeles en prosess.

**Et robust operativsystem vil basere sin relokasjon på**

1. at segmentregistrene er beskyttet og utilgjengelige i «user mode», slik at det kun er operativsystemet som kan endre innholdet i dem.
2. at det i tillegg finnes «grense»-registre som beskriver størrelsen på det tildelte minneområdet, og som hindrer et program å bruke minneceller som ligger «høyere opp» enn hva som er tildelt (og kanskje tildelt andre prosesser).




*Figur 5-3: Bruk av segment- og grenseregistre*

Figur 5-3 viser hvordan segment- og grenseregistre kan «ramme inn» deler av minnet i «segmenter». Innholdet av registrene er som tidligere sagt (“Prosessens tilstand” på side 75) en del av prosessens/trådens tilstand. Vi kan altså med bruk av segmentergistre lagre opplysninger om det tildelte minneområdet sammen med data som ligger i CPU-ens øvrige registre.

> **Viktig:** et robust operativsystem må basere adresseringsmekanismene på segment- og grenseregistre som er beskyttet i «user mode».

Når en instruksjon utføres i CPU-en som sier «hent inneholdet av minnecelle 1344», kan følgende skje:

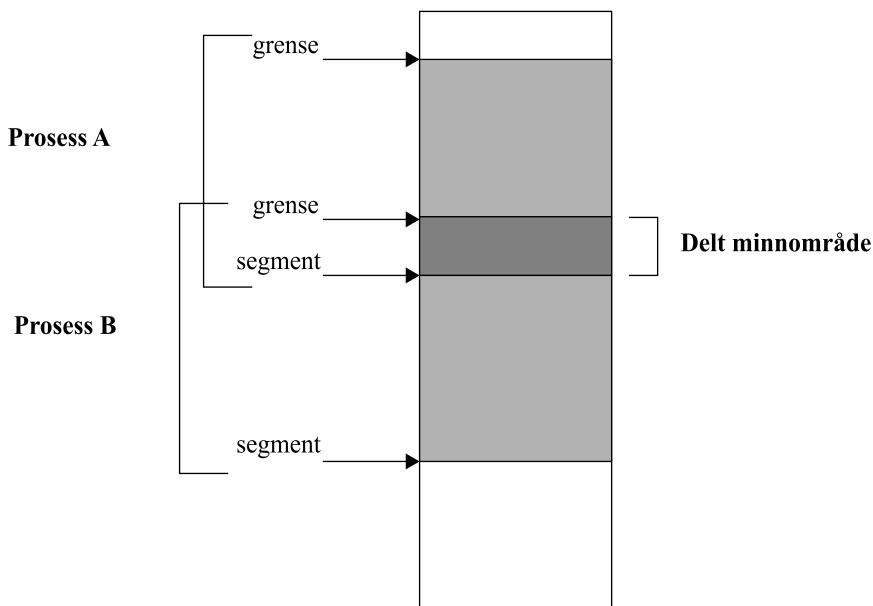
- Addere adressen (1344) og innholdet av segmentregisteret.
- Sjekk om summen blir større enn verdien av grenseregisteret. Om den gjør det, skap et programvareavbrudd (jf. “Programvareavbrudd - INT-instruksjonen” på side 59).
- Hent nå verdien av denne minnecellen slik instruksjonen ber om.

Vi ser at denne instruksjonen vil fungere hvor som helst i minnet, så lenge segmentregisteret reflekterer den aktuelle posisjonen til datasegmentet i bruk.

## Deling av minne med segmentregistre

Flere prosesser kan dele et minneområde dersom verdiene som brukes i segmentregistrene er like for alle prosessene. Figur 5-4 viser hvordan minnesegmentene kan tenkes å være overlappende slik at to prosesser kan ha deler av sitt minneområde felles, og det øvrige beskyttet.

Problemet med dette skjemaet er at kun to prosesser kan lage delvis overlappende segmenter. Tre eller flere må velge å dele enten hele minneområdet eller ingenting av det. Deling av minneområde skjer av denne grunn aldri gjennom delvis overlapp, men at prosesser deler minnet gjennom identiske verdier i segment- og grensesregistrene, slik som vist i figur 5-5. Her deler Prosess B og D hele sine minnesegmenter, mens prosess A og D har kun private minnområder.

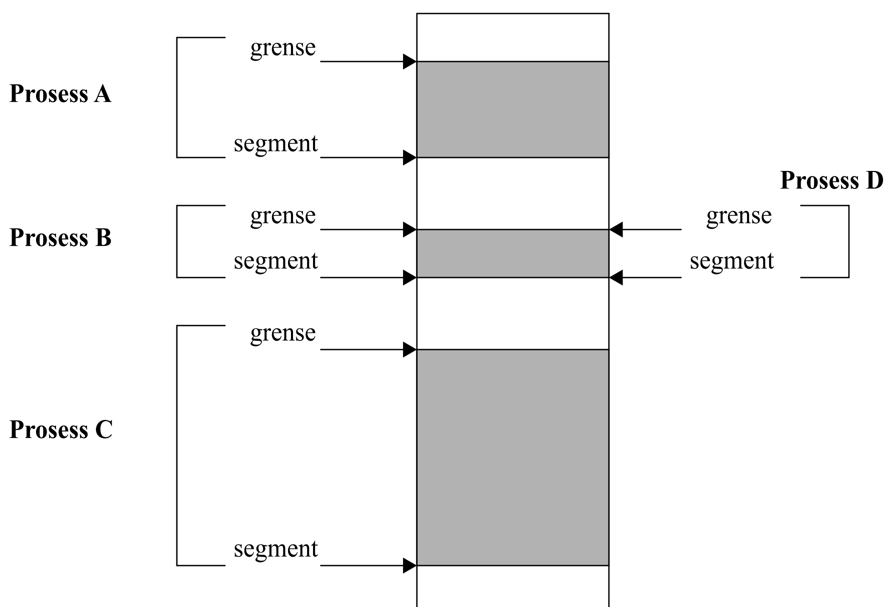



*Figur 5-4: Deling av minne gjennom delvis overlappende segmenter (ikke gjennomførbart i praksis)*

Problemet med en slik «alt-eller-ingenting»-deling er at den ikke er praktisk. Vi ønsker at prosessene skal dele akkurat det minnet som er nødvendig for et planlagt samarbeid, og det øvrige skal være privat.

Vi kan tenke oss varianter av CPU-ens adresseringsmekanismer som bruker flere par av segment- og grenseregistre, slik at en prosess har mer enn ett segment. På denne måten kunne vi skille delte og private minnesegmenter, og plassere programkoden i et separat segment (kanskje skrivebeskyttet).

Flere sett av segment- og grenseregistre brukes ikke i praksis, men ideen om flere minnesegmenter pr. prosess skal vi diskutere nærmere under avsnittet “Segmentering” på side 112.




*Figur 5-5: Deling av minneområde gjennom felles segmenter. Prosess B og D har i dette tilfellet like verdier i segment- og grenseregisteret*

## Mulige tildelingsstrategier

Vi har nå diskutert hvordan et område av minnet (minnesegment) kan tildeles en prosess gjennom å gjøre verdiene i segment- og grenseregisteret til en del av prosessens tilstand. Vi skal senere i kapitlet studere en mer avansert metode for dette formålet, nemlig «segmentering».

Vi har derimot ikke sett på hvordan vi faktisk finner et ledig minneområde som kan tildeles en prosess som vil utføre et program. Vi trenger en strategi for tildeling av minne i form av en plan for *organiseringen* av minnet, og en *algoritme* for å reservere et passende minneområde.

**Tildelingsstrategien har som oppgave å**

1. holde rede på hvilke deler av minnet som er ledig og hvilke som er opptatt
2. ved tildeling: finne et ledig område og merke dette som opptatt
3. ved tilbakelevering: merke det reserverte minnområdet som ledig

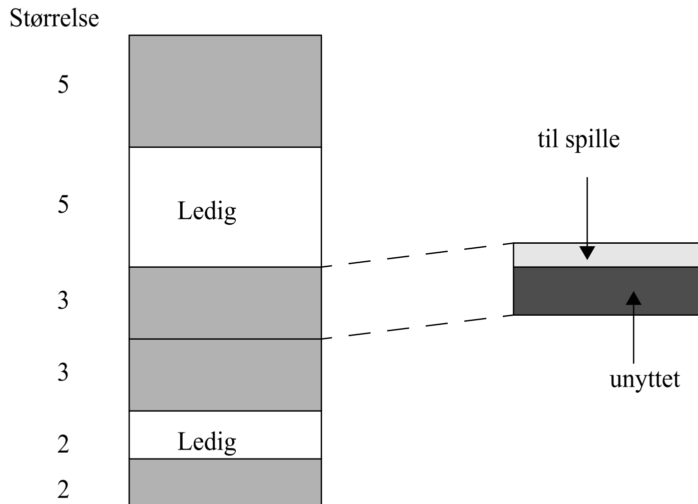
Tildelingsstrategien må dessuten ta hensyn til disse forutsetningene:

- Noen prosesser krever mye minne, andre lite. Størrelsen på det tildelte minnet vil derfor variere.
- Noen prosesser blir fort ferdig og leverer tilbake det reserverte minnet snart. Andre prosesser bruker minnet lenge. Det er derfor ingen bestemt rekkefølge i tildelings- og tilbakeleveringsoperasjonene.

Vi skal i de følgende avsnittene se på to alternative strategier, nemlig tildeling av segmenter med fast størrelse (kalt stiv tildeling) og tildeling av segmenter som varierer i størrelse (kalt fleksibel tildeling).

## Stiv tildeling

En stiv tildelingsstrategi vil dele inn minnet på forhånd i segmenter av fast størrelse (ikke samme størrelse). Når det kommer en forespørsel om tildeling av minne (med en angitt størrelse) vil algoritmen søke etter et segment som er like stort eller større. Blant de aktuelle segmentene (som er større eller lik forespørselen) plukker vi gjerne det minste (kalt *minimum-fit* strategi).




*Figur 5-6: Organisering av minnet i faste segmenter (stiv tildelingsstrategi). De uskraverte områdene er ledige. Til høyre ser at vi noe minne vil gå til spille fordi segmentet alltid er litt for stort*

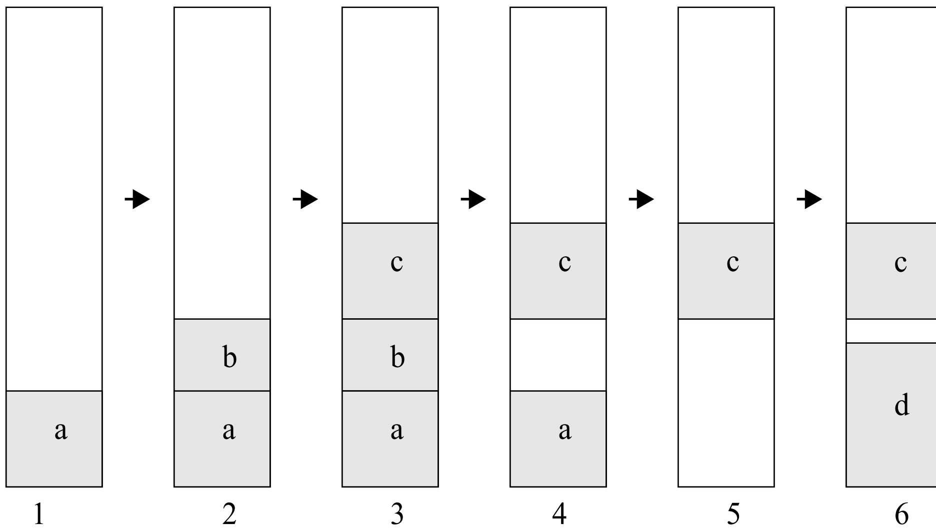
Hver gang vi reserverer et segment, går litt av minnet til spille, fordi segmentet ofte er litt større enn hva det ble spurt etter. Dette er imidlertid bare midlertidig. Når segmentet leveres tilbake, går det som er gått til spille tilbake til det ledige området sammen med det som har vært brukt.

Om størrelsene på segmentene er dårlig tilpasset det aktuelle behovet, vil relativt mye mer minne gå til spille, og vi vil oftere oppleve at operativsystemet ikke kan fullføre en forespørsel om tildeling. Vi ser også at operativsystemet aldri kan kjøre flere samtidige prosesser enn det antall minnesegmenter som finnes.

Vi erklærer derfor stiv tildelingsstrategi som ubrukelig, fordi den er for lite fleksibel. I tidligere operativsystemer har vi derimot sett denne strategien i bruk.

## Fleksibel tildeling

En fleksibel tildelingssstrategi kan reservere deler av et ledig minnesegment, og la resten forbli ledig. Under tilbakelevering vil to ledige nabosegmenter slås sammen til ett segment.




*Figur 5-7: Tildeling av segmenter med fleksibel størrelse. Se teksten for en forklaring*

**Figur 5-7 viser et forløp av tildeling og tilbakelevering av minnesegmenter**

1. Et segment av en eller annen størrelse er reservert. Vi kaller segmentet for (a). Alt øvrig minne er ledig (uskravert).
2. Segmentet (b) er nå reservert, og har redusert størrelsen av det ledige minnet.
3. Segmentet (c) er nå tildelt og reservert.
4. Segmentet (b) blir levert tilbake til operativsystemet (f.eks. når prosessen som brukte det er avsluttet). Minneormådet blir merket som ledig.
5. Segmentet (a) blir levert tilbake. Minneområdet blir markert som ledig og slått sammen med det ledige naboområdet til ett samlet ledig område.
6. Segmentet (d) blir reservert og tildelt. Størrelsen på segmentet er større enn hver av (a) og (b), og utnytter det samlede området som oppsto i steg (5).

I trinn 6 valgte vi det minste av de som var store nok, kalt *minimum-fit*. Dette er den vanligste metoden. Vi kunne også ha valgt det største av de ledige segmentene, kalt *maximum-fit.*

Vi ser straks at en fleksibel tildelingsstrategi nettopp er mer fleksibel. Vi trenger ikke å gjøre noen antagelser om hvorvidt operativsystemet skal romme et fåtall prosesser med stort minnebehov eller mange prosesser med små minnebehov.

Fleksibel tildeling er mye brukt. Ikke bare i forbindelse med tildeling av minne fra operativsystemet, men også tildeling av diskplass og plass til dynamiske variabler på heapen (jf. “Stakken og heapen” på side 66).

### Fragmentering

Fleksibel tildeling har derimot et problem knyttet til seg som vises på stegene 4, 5 og 6 på figur 5-7: Det ledige området i minnet er delt opp i isolerte regioner. Vi kan tenke oss at en forespørsel om tildeling av et stort minnesegment må avvises selv om det *i sum* er tilstrekkelig ledig minne, fordi det ledige minnet er delt opp i regioner som *hver for seg* er for små.

Vi kaller dette fenomenet for *fragmentering*, som er en naturlig prosess der tildeling og tilbakelevering skjer i varierende størrelser og ubestemt rekkefølge. I en datamaskin finner vi fenomenet også i forbindelse med tildeling av plass på disklageret.

**Eksempel:** Tenk deg at du kun hadde lov til å betale varer med én mynt eller én seddel. Etter hvert ville lommeboka din være fylt opp med småmynter som du hadde mindre og mindre glede av. Du ville *i sum* ha nok penger til en vare, men være ute av stand til å gjennomføre en handel.

Fragmentering oppstår bl.a. fordi systemet ikke er i stand til utnytte summen av isolerte minneregioner. Vi skal nå se på teknikker for minnestyring (kalt «paging») som bl.a. setter oss i stand til unngå fragmenteringsproblemet.

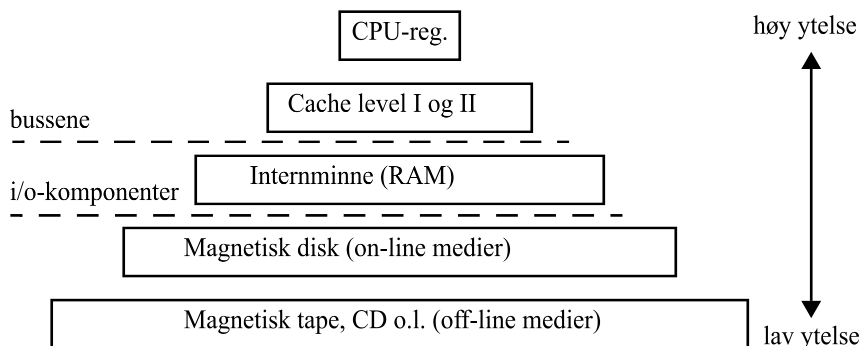
## Virtuelt minne

Virtuelt betyr «tilsynelatende», og betegner de moderne teknikkene for minnestyring som bl.a. har disse egenskapene:

- får minnet til å se større ut enn det egentlig er, gjennom å utnytte disklageret til å lagre mindre brukte data og kode
- er mindre påvirket av fragmentering

## Lagringshierarkiet

Mediene for lagring i en datamaskin kan stilles opp i et hierarki, hvor de hurtigste mediene står øverst, og de langsomme nederst.




*Figur 5-8: Lagringshierarkiet*

Med «hurtig» og «langsom» sikter vi til tiden det tar for behandle dataene, dvs. flytte dem inn i CPU-en og utføre instruksjoner på dem.

**CPU-registre** Øverst i lagringshierarkiet finner vi CPU-registrene. Data som befinner seg der kan uten videre behandles av CPU-en slik programinstruksjonene angir. Behandlingen blir dermed raskere enn noe annet lagringsmedium, men det er kun et fåtall CPU-registre tilgjengelig, og mange av dem er knyttet til spesialoppgaver som gjør dem mindre egnet for generell lagring.

**CPU-cache** På det neste trinnet finner vi CPU-cachene, som vi har beskrevet i avsnittet “Bruk av caching” på side 34. Data som ligger lagret her må flyttes inn i et CPU-register gjennom en transportvei som enten ligger inne i CPU-brikken eller er en separat buss. I begge tilfeller skjer dette noe senere enn med data lagret i CPU-registrene.

**Internminne** Nedenfor finner vi internminnet, ofte kalt RAM (Random Access Memory). For å flytte data fra internminnet til CPU-en kreves det en buss-syklus, beskrevet på side 31. En buss-syklus tar igjen noe lengre tid, men fortsatt er det slik at data lagret i internminnet er *direkte adresserbart* fra CPU-en, slik at dataene kan hentes av CPU-ens adresseringsmekanismer gjennom maskinens busser. Tiden det tar for å transportere data fra internminnet til CPU-en er i størrelsesordenen 10-100 nanosekunder.

**Magnetisk disk** På trinnet nedenfor finner vi magnetisk disk, som skiller seg fra lagene over ved at:

- dataene ikke er direkte adresserbare, men må hentes gjennom betjening av i/o-kretser. Dette skjer gjerne i form av funksjonskall til programvare som er skrevet for dette formålet.
- de lagrede dataene ikke er *flyktige*. De beholder sin verdi også etter at vi har slått av strømmen på maskinen.

Tiden det tar for å behandle data som ligger på magnetiske disker ligger i området ett millisekund og oppover til flere sekunder.

**Magnetbånd, CD** Enda lenger ned i hierarkiet finner vi langsomme medier som CD-plater og magnetbånd. De har en svært lav pris regnet pr. lagret megabyte, og behandlingstiden kan være opptil flere minutter.

**Langsomt = billig** Hurtige lagringsmedier er dyrere (målt pr. lagret megabyte) enn langsomme. Dette kommer av at det er hastighet vi er villige til å betale mer for, ingenting annet. Og under «innredningen» av en datamaskin skaffer vi mer av det som er billig, og mindre av det som er dyrt. Følgelig blir lagringshierarkiet smalt på toppen og bredest nederst.

**Hyppig brukt = hurtig lagret** De dataene som er under behandling i en datamaskin vil behandles med ulik «hyppighet». Noen dataelementer vil stadig vekk bli lest og endret, andre ganske sjelden. Det er en god idé å plassere hyppig brukte data høyt opp i lagringshierarkiet, og sjeldent brukte data lenger ned. Da utnytter vi de hurtige lagringsmediene best, ved at vi faktisk bruker dem til oppgaver som haster.

**Automatisk plassering på rett sted i hierarkiet** Vi ønsker å utnytte lagringshierarkiet slik at dataene blir plassert på «rett» lagringsmedium automatisk. En slik teknikk kan f.eks. forsøke å oppnå dette ved å sørge for at et hyppig brukt dataelement kan skifte plass med et sjeldent brukt dataelement på et høyere nivå i lagringshierarkiet.

For å få dette til må vi holde et slags regnskap over hvor hyppig dataelementene er i bruk. Vi har allerede under diskusjonen om bruk av caching sett hvordan algoritmen «Fongens klesskap» kan føre et slikt regnskap, og i forbindelse med caching finner vi nettopp et eksempel på at data blir automatisk skiftet mellom internminne og CPU-cache etter hvor hyppig de blir brukt.

Alle steder der vi ser ordet «caching» i bruk (i en disk-kontroller, eller en nettleser) er de samme teknikkene i bruk: Hyppig brukte data flyttes til hurtigere (men trangere) lagringsmedier. Sjeldent brukte data flyttes motsatt vei.

Vi skal nå se hvordan vi kan realisere lignende teknikker i styringen av internminnet ved hjelp av *paging*.

## Paging

Uttrykket «paging» betegner en teknikk for minnestyring som har disse formålene:

- Redusere fragmenteringsproblemet
- Flytte mindre brukte data (og instruksjoner) til magnetisk disk for å utnytte internminnet bedre

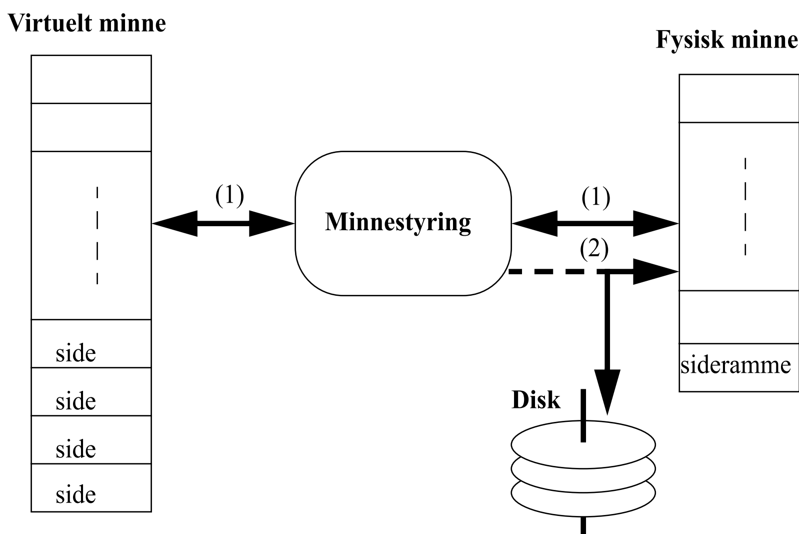
For å forstå paging-mekanismene må vi ha noen nye begreper klart for oss:

**Virtuelt minne** Programinstruksjoner som adresserer minneceller vil nå ikke lenger adressere virkelige minneceller, men «tenkte minneceller» hvor innholdet kan være lagret i minneceller, eller et sted på en magnetisk disk (i en fil).

**Fysisk minne** Det virkelige internminnet. Det fysiske minnet er «usynlig» for programinstruksjonene, som bare ser minneceller i det virtuelle minnet.

Det er minnestyringen som må sørge for «avbildningen» mellom virtuelt og fysisk minne, dvs. sørge for at innholdet av de virtuelle minnecellene finnes i det fysiske minnet når det trengs, og oversette de virtuelle minneadressene til fysiske minneadresser. Instruksjoner som skal utføres og data som skal behandles må alltid ligge i internminnet.

**Sider (pages)** Vi deler det virtuelle minnet inn i like store (like mange minneceller) regioner, kalt *sider*. Dette er den minste samlingen av minneceller som blir skiftet mellom disklageret og internminnet. *En side i det virtuelle minnet (også kalt virtuell side) vil i sin helhet* ***enten*** *eksistere på disklager eller i internminnet.* En region av det fysiske minnet som kan romme en side, kaller vi en *sideramme* (page frame).

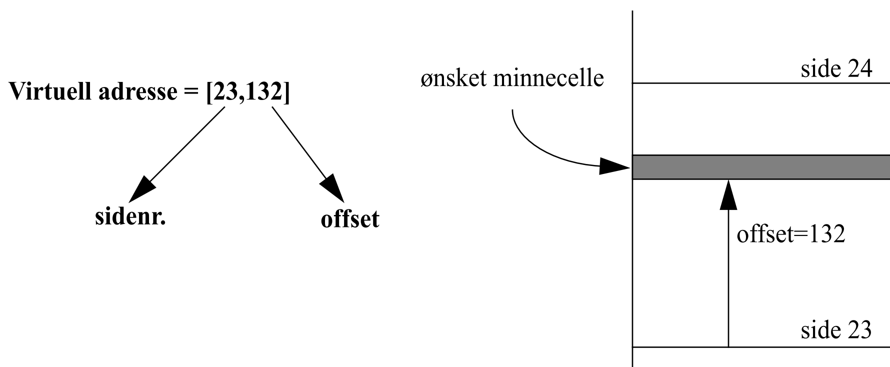



*Figur 5-9: Sammenhengen mellom sider, virtuelt minne og fysisk minne*

Figur 5-9 viser hvordan minnestyringen står som en «oversetter» mellom virtuelle adresser og fysiske adresser (1). Figuren viser også at det virtuelle minnet alltid er større enn det fysiske, og at minnestyringen må transportere data mellom disklager og internminnet for at dataelementene skal ligge i internminnet når de skal behandles (2).

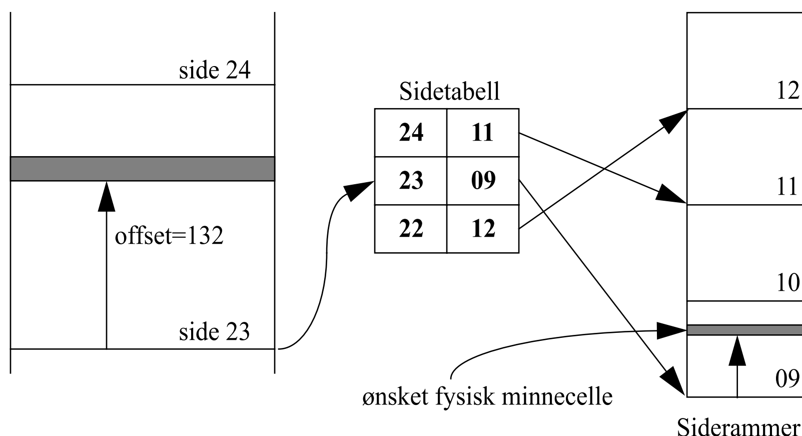
Nå som vi har «det store bildet» klart for oss, kan vi se nærmere på «innmaten» av boblen midt på figuren, og vi kan studere detaljene i hvordan skiftet mellom disklager og internminnet foregår.

**Sidetabell (page table)** Minnestyringen bruker en tabell som oversetter en virtuell adresse til en fysisk adresse, men tabellen har bare én rad for hver virtuell side. En virtuell adresse må derfor kunne skrives som paret [sidenr.,offset]. Offset forteller oss hvilken minnecelle på den aktuelle siden vi adresserer.




*Figur 5-10: En virtuell adresse og en virtuell minnecelle*

Når minnestyringen skal finne den fysiske adressen til denne minnecellen (forutsatt at den ikke ligger på disklager), må den titte i sidetabellen og slå opp på «side 23». På denne raden finnes *nummeret på siderammen* hvor denne siden ligger lagret. Den virtuelle siden ligger i sin helhelt i denne siderammen, derfor finner vi innholdet i den virtuelle minnecellen i 132 minneplasser (offset) opp i siderammen.




*Figur 5-11: Bruk av sidetabell. Innholdet i den skraverte minnecellen i det virtuelle minnet (til venstre) ligger lagret i den skraverte minnecellen i det fysiske minnet (til høyre)*

**Oversettelsen skjer i maskinvaren** Operasjonene som består i å skille sidenr. og offset, gjøre oppslag i sidetabell og sette sammen en fysisk adresse, skjer inne i CPU-brikkens adresseringskretser[2]. Programinstruksjonene ser vanlige ut, og de opererer på minneceller som ser helt ekte ut. Vi trenger altså ikke å skrive programmene annerledes i et system som benytter paging. Det er operativsystemets minnestyring som må sørge for å administrere *innholdet* i sidetabellen.

**Kan utnytte fragmentert fysisk minne** Vi ser nå hvordan «småbiter» av ledig fysisk minne kan stilles sammen til en sammenhengende blokk av virtuelt minne: Den siste cellen på side nr. 23 og den første cellen på side nr. 24 er naboceller i det virtuelle minnet, men innholdet er lagret i fysiske minneceller som ligger langt fra hverandre (i sideramme 9 og 11). Gjennom innholdet av sidetabellen har vi gjort oss uavhengige av fragmenteringstilstanden i det fysiske minnet.

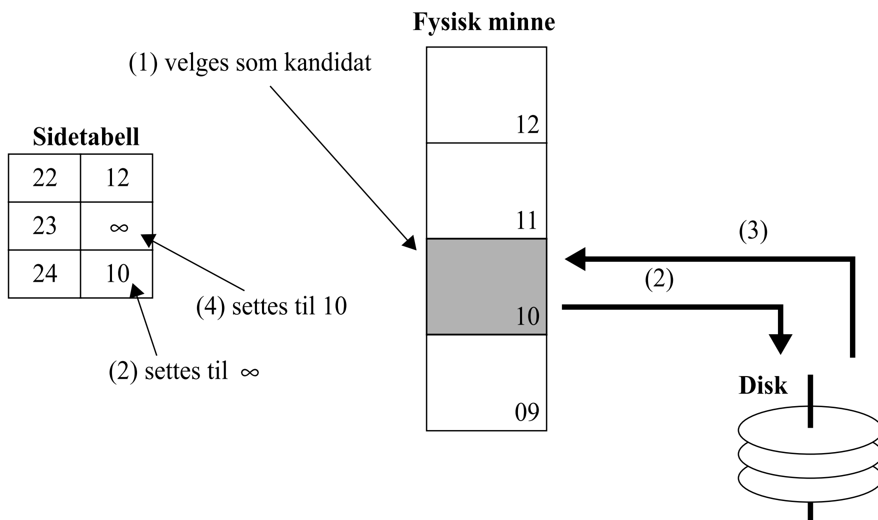
**Sidebrudd (page fault)** Vi har ennå ikke snakket om hvordan vi kan skifte innhold mellom disklager og internminne. Dette skjer gjennom de samme mekanismene som vi nettopp har diskutert. En reservert verdi i sidetabellen kan bety «ingensteds» og fortelle adresseringskretsene i CPU-en at denne siden i det virtuelle minnet ikke ligger noen steder i det fysiske minnet, men derimot på disklageret. Under adressering av en celle i denne siden oppstår noe som vi kaller et «sidebrudd» (eng. page fault), og resulterer i et internavbrudd. Det blir avbruddsrutinens oppgave å sørge for at innholdet av siden hentes fra disklager og plasseres i en sideramme.

**Et sidebrudd kan behandles i følgende steg (illustrert i figur 5-12):**

1. Det må skapes en ledig sideramme, og den siderammen som har vært «minst brukt» i det seneste tidsrommet kan være en god kandidat.
2. Innholdet av denne siderammen lagres til disk, og sidetabellen oppdateres slik at siden som tidligere lå i rammen nå er merket med «på disk».
3. Den siden som skal hentes fra disk lokaliseres på disken og kopieres inn i den siderammen som nå er ledig.
4. Sidetabellen oppdateres for den siden som nå er hentet inn fra disk, slik at den henviser til den aktuelle siderammen.

---

2. Forutsatt at CPU-brikkens design støtter paging, og at paging er «slått på».




*Figur 5-12: Skifte av sideinnhold mellom disklager og sideramme*

Resultatet av sidebrudd-behandlingen er at den adresserte virtuelle siden blir tilgjengelig i en sideramme, og adresseringsoperasjonen kan fullføre. Husk at dette skjer i form av en avbruddsrutine:

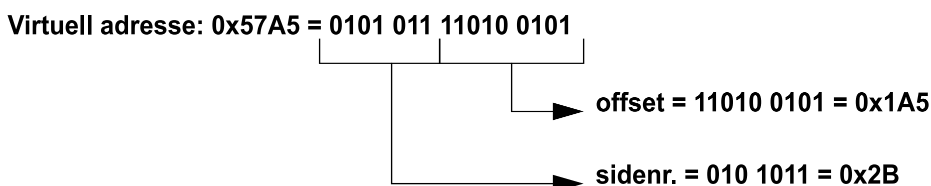
- adresseringsoperasjonen blir avbrutt når CPU-en (etter å ha tittet i sidetabellen) fastslår at det er oppstått et sidebrudd
- internavbruddet som oppstår vil behandles av en avbruddsrutine som ligger på en adresse angitt av en avbruddsvektor
- avbruddsrutinen lagrer tilstanden til prosesskonteksten (se "Avbruddshåndtering" på side 80)
- avbruddsrutinen gjennomfører skiftet som beskrevet i stegene 1–4 ovenfor

avbruddsrutinen gjenskaper tilstanden i prosesskonteksten og lar den avbrutte adresseringsoperasjonen fullføre

Dette skjemaet er ganske komplisert å forstå (og enda mer komplisert å programmere). Det vi derimot oppnår med paging, er en automatisk utnyttelse av lagringshierarkiet slik at mindre brukte sider blir flyttet til disk, mens mye brukte sider blir liggende i internminnet.

> **Hint:** For å forstå virkemåten for paging, må du være fortrolig med avbruddshåndtering. Gå tilbake og repeter om nødvendig!

**Hvor stor er en side?** Sidestørrelsen er alltid et antall bytes som er en potens av 2, f.eks. 512, 1024, 2048 eller 4096 bytes. Grunnen til det er at det skal være enkelt å skille sidenr. og offset-verdi fra hverandre i en virtuell adresse. Ved å bruke sidestørrelser som er en 2-er potens, vil et visst antall bits til høyre i adressen være offset, og resten av bitene til venstre være sidenr.




*Figur 5-13: Skillet mellom sidenr. og offset i en virtuell adresse (tilfeldig valgt verdi)*

Eksemplet i figur 5-13 viser at det er lettere å forstå og beregne delene av en virtuell adresse om man skriver ned tallene i binærformat.

## Segmentering

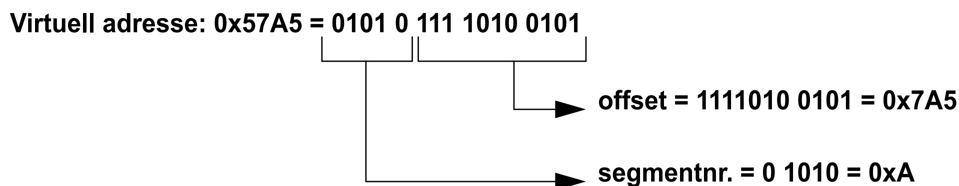
Segmentering er en teknikk for minnestyring som har disse målene:

- Dele minnet inn i enheter (segmenter) som kan håndteres individuelt med tanke på tildeling, beskyttelse, deling og tilbakelevering,
- slik at hver prosess kan ha et vilkårlig antall segmenter

Tidligere i dette kapitlet diskuterte vi bruken av segment- og grenseregistre for å håndtere minnesegmenter, og vi konkluderte med at ett sett av slike registre ikke ga den nødvendige fleksibiliteten. Heller ikke bruk av flere alternative sett av segment- og grenseregistre er aktuelt.

Vi kan derimot tenke oss bruk av en segmenttabell, og bruk av virtuelle adresser hvor én del peker til et segmentnr. og den andre delen til en offset-verdi i dette segmentet. Segmenttabellen vil (i likhet med en sidetabell) fortelle hvor i minnet segmentet er plassert og hvor stort segmentet er. Sammen med denne informasjonen kan segmenttabellen

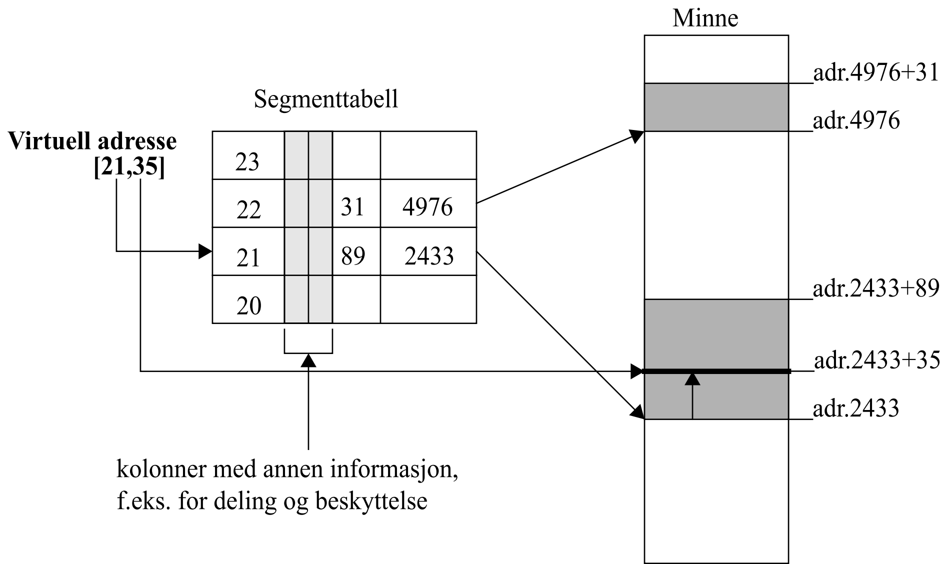
også fortelle hvilken prosess som er tildelt segmentet, om det kan deles mellom flere, om det er et read-only[3] eller read-write segment, og når segmentet sist ble adressert.




*Figur 5-14: Virtuell adresse brukt i segmentering*

Segmenttabellen gir minnestyringen den fleksibiliteten som vi har savnet ved bruk av segmentregistre. Figur 5-15 viser hvordan de minneadressene som prosessen nå kan bruke består av «flak» av tilgjengelig minne som har forskjellige bruksområder og ulik beskyttelse. Mellom disse flakene ligger det adresser som ikke hører til noe segment (fordi segmentene oftest er mindre enn den maksimalt tillatte størrelsen). Mellom adressene [21,88] og [22,00] ligger det ingen minneceller. Adresser i dette området er derfor ugyldige og ulovlige å bruke.

3. Med «read-only» mener vi at segmentet er beskyttet mot skriving. Dette gjør segmentet velegnet for å lagre programkode, som aldri skal endres.




*Figur 5-15: Bruk av segmenttabell. Kolonnen til høyre peker til starten på det gjeldende segmentet, og kolonnen nest lengst til høyre angir lengden på segmentet*

**Swapping** Det er mulig å merke segmenter i tabellen med adresseverdier som indikerer at de ikke ligger i det fysiske minnet, men på disk. I eldre systemer har denne teknikken gått under navnet «swapping», og har vært en slags forløper til paging. Kun hele segmenter kan swappes mellom disk og internminnet, noe som gjør teknikken mindre fleksibel enn paging. Ellers minner swapping mye om paging, i måten et «segmentbrudd» skaper et internavbrudd på, og en avbruddsrutine sørger for å skifte segmentene mellom disk og internminne.

**Sidedelte segmenter** Vi kan kombinere fordelene med segmentering og paging slik at

- segmentene støtter den logiske styringen av minneområdet
- sidestrukturen plasserer dataene automatisk i lagringshierarkiet og motvirker fragmentering

Legg merke til at vi i omtalen av segmentering ikke har sagt noe om hvorvidt segmentene ligger i fysisk eller virtuelt minne (vi har bare brukt uttrykket «minne»). Det er en god idé å la segmenteringen foregå på det virtuelle minnet. I dette tilfellet vil altså søylen til høyre på figur 5-15 representere virtuelt minne i likhet med søylen til venstre på figur 5-9. Dette kaller vi for *sidedelte segmenter* (eng.: paged segments), og

det er en teknikk i vanlig bruk i moderne operativsystemer. Segmentadressen og offset-verdien som vist på figur 5-15 danner da ingen fysisk minneadresse, men en virtuell adresse som består av et sidenr. og en offsetverdi (som skal tolkes som en offset i siden, ikke i segmentet).

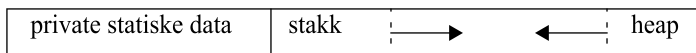
## Minnebruk i et høynivåspråk

I kapittel 3 diskuterte vi organiseringen av variabler i et høynivåspråk, og rundt figur 3-5 på side 67 viste vi hvordan programmets lagringsbehov (kode, statiske data, stakk og heap) kan organiseres innenfor et sammenhengende minneområde. Om vi nå igjen vurderer en slik organisering av dataene ut fra deres behov for deling og skrivebeskyttelse, får vi tabellen under:

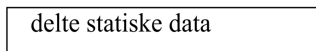
| Lagringsbehov | Delt? | Read-Only? |
|---|---|---|
| Kode | Ja | Ja |
| Statiske data | Muligens | Nei |
| Stakk | Nei | Nei |
| Heap | Nei | Nei |

Et minnesegment vil tildeles av operativsystemet med én type deling og beskyttelse. Tabellen gir oss dermed mulighet til å se hvilke lagringsbehov som kan samles i samme minnesegment og hvilke som må ligge separat. Vi finner at vi må organisere dataene i tre segmenter, vist på figur 5-16. De statiske dataene er her delt på to segmenter, avhengig av om de skal deles mellom flere prosesser eller være private. Stakk og heap kan ikke deles fordi variablene der har en «ubestemt» levetid. Programkoden skal aldri endres og kan derfor med fordel legges i et skrivebeskyttet minnesegment. Programkoden er velegnet for deling (av flere prosesser som ønsker å kjøre det samme programmet) og legges derfor i et minnesegment som kan deles.

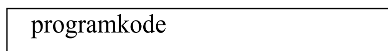
*Ikke delt, ikke skrivebeskyttet*




*Delt, ikke skrivebeskyttet*




*Delt, skrivebeskyttet*




*Figur 5-16: Organisering av programmets data i segmenter med ulik beskyttelse*

## Minnestyring i Intel Pentium

Intel Pentium har de samme mekanismene for minnestyring som sine forgjengere, tilbake til 80386-prosessoren, som ble introdusert i 1987. Siden den gang er det kommet mange forbedringer av ytelsesmessig karakter, men *arkitekturen* i prosessoren er fortsatt den samme. Operativsystemer skrevet for Pentium kan derfor prinsipielt kjøres på 80386-prosessorer, men den relativt lave hastigheten og minnestørrelsen på gamle maskiner kan gjøre dem praktisk uegnet for dette.

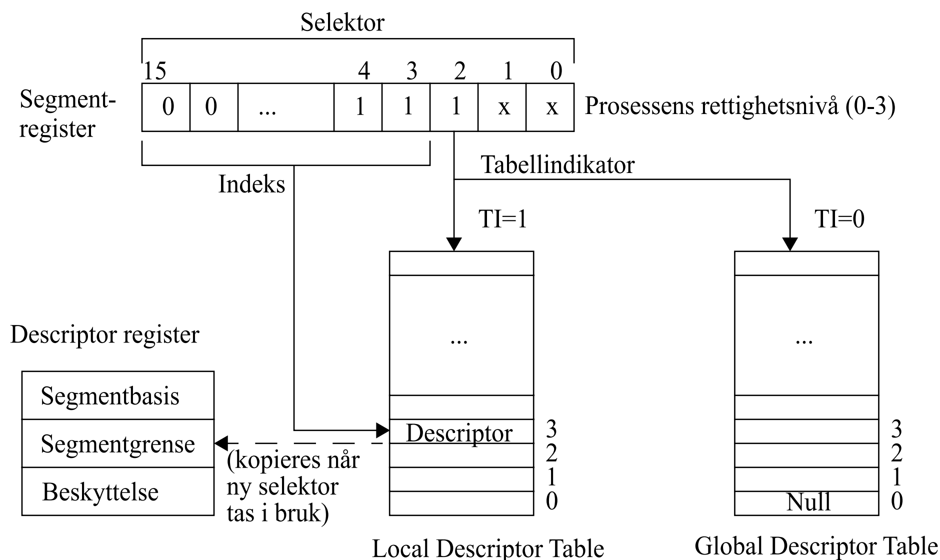
Intel Pentium er også kompatibel med de gamle prosessorene 8086/8088 (brukt i IBM PC) og 80286 (brukt i IBM AT) slik at programmer og operativsystemer for disse prosessorene kan kjøres på Pentium. Men det er minnestyringen som ble introdusert i 80386 som vi her skal omtale.

Intel Pentium støtter sidedelte segmenter og segmentbeskyttelse. Den organiserer segmentene slik:

- Alle referanser til minneceller brukt til instruksjoner (henting av instruksjoner) adresserer via et *kodesegmentregister.*
- Alle referanser til dataceller (lesing og skriving av data) adresserer via et *datasegmentregister*[4].
- Segmentregistrene innholder en såkalt *selektor* med disse opplysningene: Angivelse av hvilken segmenttabell som skal brukes (kalt Local/Global Descriptor Table) og angivelse av rad i segmenttabellen.

4. Helt nøyaktig er det ett av *flere* datasegmentregistre.

- Hver rad i segmenttabellen inneholder en *deskriptor* med tre opplysninger: Segmentets basisadresse, segmentets lengde og segmentets beskyttelsesnivå (0–3).
- Hver gang innholdet i segmentregisteret blir endret, blir den aktuelle deskriptoren i segmenttabellen lastet inn i et tilhørende register kalt *descriptor register*.
- Alle minneadresser brukt i programmet blir addert til segmentets basisadresse, kontrollert mot segmentets størrelse og beskyttelse. Dersom adressen blir godkjent som «lovlig» vil adresseringen fullføres (til fysisk eller virtuelt minne), i motsatt fall vil prosessoren skape et internavbrudd.

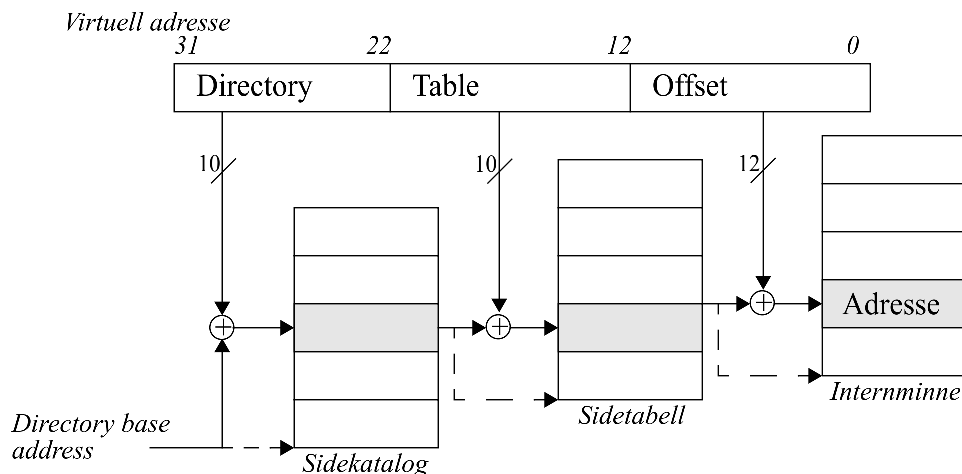



*Figur 5-17: Et 16-bits segmentregister velger et segment ved å angi en rad i enten «Local Descriptor Table» (LDT) eller «Global Descriptor Table» (GDT). Hver prosess kan ha sin egen LDT*

Legg merke til at Intel ikke benytter en todelt «virtuell adresse» slik som vist på figur 5-14, men at segmentnr. og offset kommer fra helt separate komponenter. Segmentregisteret og den selektoren det inneholder vil i stor grad være usynlig (og uinteressant) for et program som nøyer seg med å bruke to segmenter. Maksimum segmentstørrelse er 4 GBytes.

Den adressen som dannes ved å addere segmentbasis til «offset» (adressen innenfor segmentet) kan referere enten til fysisk eller virtuelt minne (sidedelt gjennom paging).

Paging i Pentium minner om den metoden som er beskrevet på side 107, men den virtuelle adressen som brukes er i motsetning til den på figur 5-10 delt i tre deler: *Page Directory index*, *Page Table Inde*x og *Page Offset*. Vi opererer i Pentium med *mange* sidetabeller som til sammen dekker det virtuelle minnet som er tildelt en prosess:




*Figur 5-18: Pentium bruker en to-trinns mekanisme for å adressere sidedelt minne. En «Page Directory» er en katalog over de «Page Tables» som dekker det virtuelle minnet til prosessen. Offset er 12 bits, noe som gir en sidestørrelse på 4096 bytes*

Fordelen med en slik tredeling er at sidetabellene blir mindre i de tilfellene det virtuelle minnet består av adskilte «adresseflak». Da kan hvert flak ha sin egen sidetabell, alle sammen representert i sidekatalogen.

## Sammendrag

- Minnestyringen skal tildele minneplass til prosesser slik at de har plass til å lagre verdiene i sine variabler.
- Minnestyringen skal *beskytte* tildelt minne, og adminstrere *deling* av minne der det er ønskelig.
- *Segmentering* av minnet gir en fleksibel styring av minnet, antall segmenter og størrelsen på segmentet kan variere sterkt.

- *Paging* motvirker *fragmentering* og utnytter *lagringshierarkiet*. Mindre brukte minneområder kan legges midlertidig til disk.
- *Sidedelte segmenter* har flere fordeler og er aktuell minnestyringsstrategi i moderne prosessorer.

| **Sentrale begreper i dette kapitlet:** | |
|---|---|
| relokasjon | indeksert adressering |
| relativ adressering | lagringshierarki |
| fysisk minne | virtuelt minne |
| segmentering | paging |
| virtuell adresse | side, sideramme |
| sidetabell | sidebrudd |

## Teorioppgaver

**Gå sammen i grupper og arbeid med disse oppgavene:**

1. Ved stiv tildeling av minne (figur 5-6) er det mulig å beregne hvor stor del av minnet som går til spille. Hvordan blir minneutnyttelsen om 7 prosesser alle trenger størrelsen 1? Størrelsen 2? Hva om det kommer 8 prosesser som vil kjøre samtidig?
2. Skriv en skisse til en Java-klasse for fleksibel tildeling. Et objekt av denne klassen skal rå over et array av byte, og ha metoder for tildeling og tilbakelevering. Forsøk å finne ut hvordan klassen kan organisere arrayet, og forsøk også å skrive koden for tilbakelevering av en bit av dette arrayet.
3. I avsnittet om “Fleksibel tildeling” på side 102 kom det frem at «minumum-fit» er den vanligste tildelingsalgoritmen. Kan det tenkes at en «maximum-fit» er en bedre strategi fordi den etterlater mindre «smårusk» i form av små, ubrukelige minnesegmenter? Diskuter dette alternativet.
4. Fremfor å ha en eller flere sidetabeller pr. prosess, kan vi tenke oss å ha én sidetabell for hele det virtuelle minnet, brukt av alle prosessene. Finn frem til fordeler og ulemper ved denne løsningen.

5. Om et program adresserer minnet slik at det stadig skjer sidebrudd, kan det oppstå en situasjon som kalles *trashing*, og da bruker maskinen all sin kapasitet på minnestyringen og får gjort lite annet. Hvordan kan et program skrives slik at det oppstår trashing?

## Øvingsoppgaver

Forslag til øvingsoppgaver ligger på bokas nettsted.

**Etter fullførte øvinger bør du beherske følgende:**

1. Forstå hvordan Java Virtual Machine organiserer Java-kode og data på Windows (NT/2000).
2. Kjenne til verktøyprogrammene for å studere programmenes minneforbruk.
3. Forstå hvordan et Java-program skal skrives for ikke å sløse med minneressurser.
4. Kunne skrive Java-programmer som demonstrerer effekten av paging-teknikkene ved et det oppstår *trashing*.

# Kapittel 6

# Synkronisering I

*I dette kapitlet skal vi diskutere behovet for at tråder samordner sin utføring av instruksjoner i tid. Vi skal også se på noen programmeringsteknikker for å synkronisere Java-tråder.*

## Hva mener vi med «synkronisering»?

Samarbeidende tråder vil under utføring ha behov for å samordne sin utføring. La oss gå tilbake til kjøkkenet (side 73) og se hvordan kokkene der samarbeider.

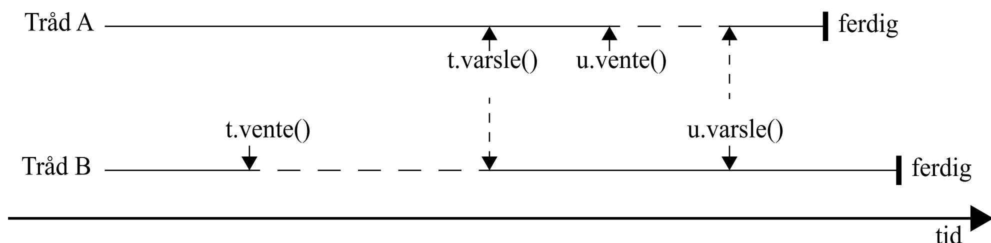
- Kokken Arne skjærer grønnsaker, mens kokken Erik trenger grønnsakene til suppen han lager. Dersom Arne skjærer for langsomt, må Erik stoppe opp litt i arbeidet og vente. Han sier kanskje «si ifra når du har fylt en bolle». På den annen side må Arne ta seg en pause dersom Erik ikke bruker grønnsakene han skjærer, og de hoper seg opp på arbeidsbenken.
- Kjøkkenet har bare én hvitløkpresse, og kokkene Lars og Per trenger begge å bruke denne. Når Lars trenger pressen, må han kanskje vente på at Per skal gjøre seg ferdig med den. Det samme gjelder også i omvendt rekkefølge; når den ene holder på med pressen, må den andre vente.
- Dersom oppvaskmaskinen går i stykker, må en reparatør tilkalles. Inntil reparatøren har gjort seg ferdig, vil oppvaskarbeidet måtte vente. Kanskje sier oppvaskgutten «gi meg beskjed når maskinen er reparert, så tar jeg en pause imens».

### Samordning i tid

Synkronisering er et behov som oppstår når flere tråder samarbeider om en oppgave. På ulike måter vil de da være avhengig av hverandres forløp, og de må av og til vente på at en annen tråd har oppfylt en *betingelse*. En tråd må også kunne gi beskjed til én eller flere andre tråder om at en gitt betingelse nå er oppfylt.

**Betingelser: vente, varsle** Vi ser at alle tre eksemplene fra kjøkkenet i forrige avsnitt viser at samarbeidende aktiviteter er avhengig av at *betingelser* er oppfylt, og at det er mulig å *vente på* eller *varsle* at betingelser er oppfylt.

Vi skal tegne mange figurer med bruk av *tidsakse* i løpet av dette kapitlet. Langs tidsaksen finner vi de samarbeidende trådene, og figuren viser hvordan de påvirker hverandres tilstander gjennom å vente og varsle:




*Figur 6-1: Tråder som venter og varsler langs en tidsakse. Heltrukket linje betyr «aktiv», og stiplet linje betyr «passiv» (ventende)*

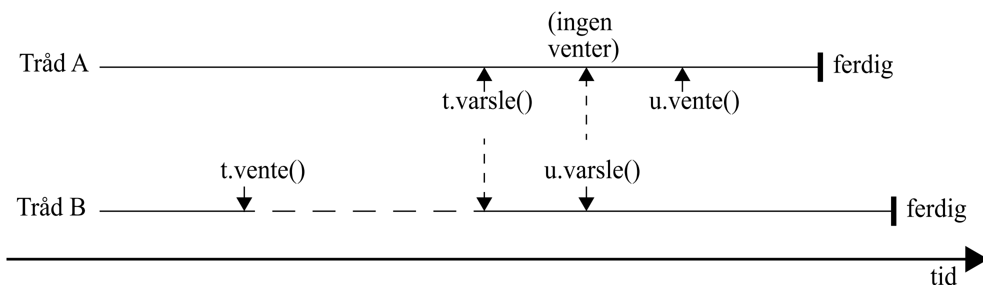
På figur 6-1 vises forløpet av to tråder som synkroniseres over betingelsene *t* og *u*. Tråd B ber om å få vente til *t* er oppfylt, og forholder seg deretter passiv inntil dette er skjedd. På et senere tidspunkt vil tråd A konstatere at *t* er oppfylt og varsle om dette. Senere skjer det samme over betingelse *u*, men nå er det tråd A som venter og tråd B som varsler.

Hva mener vi med at tråden er «aktiv» eller «passiv»? Vi skal senere kople disse begrepene til begrepet *utføringsstatus* som vi innførte i kapittel 4 (“Trådenes tilstand” på side 76), men i denne omgang nøyer vi oss med denne definisjonen:

- Aktiv: Tråden utfører instruksjoner uten opphold
- Passiv: Tråden utfører ikke instruksjoner, men «står stille»

Vi ser av figuren at overgangen fra aktiv til passiv tilstand skjer i forbindelse med en *betingelse.vente()*-operasjon. Hvor skal tråden «gjøre av seg» mens den er i passiv modus? Den vanligste løsningen er at den ikke returnerer fra *vente*-metoden før den igjen er kommet i aktiv tilstand. Den første programsetningen etter *vente*-metoden vil derfor først utføres når tråden igjen er i aktiv tilstand.

**Rekkefølgen av hendelser** Figuren er tegnet med en tidsakse som gjør det mulig å vise rekkefølgen av hendelser. Vi kan derfor være sikker på at hendelser til venstre på en linje skjer før hendelser til høyre på samme linje. Men vi kan ikke si mye om rekkefølgen av hendelser langs forskjellige linjer (i forskjellige tråder). Tenk om hendelsen *u.varsle()* skjer før *u.vente()*, hvordan skal det tolkes?




*Figur 6-2: I dette tilfellet vil tråd A vente på en betingelse (u) som allerede er oppfylt. Den vanligste tolkningen av denne situasjonen er at tråden fortsetter å være aktiv*

I noen tilfeller er dette helt ok. Om ingen bruker hvitløkpressen, er betingelsen «hvitløkpresse ledig» allerede oppfylt i det noen ønsker å vente på denne betingelsen. I dette tilfellet er det naturlig at *vente()*-metoden ikke setter tråden i passiv modus, men lar tråden fortsette straks.

I andre tilfeller er det ikke ok. Om vi har bedt om å få reparert oppvask-maskinen, og det viser seg at betingelsen «oppvaskmaskin reparert» allerede er oppfylt, er det grunn til å tro at det er noe feil. Kanskje er det forrige reparasjon som er ferdig og som skaper denne betingelsen? I slike tilfeller av synkronisering ønsker vi å reagere på *varsle()*-hendelser som skjer *etter vente()*-hendelser, ikke de som er skjedd tidligere.

> **Viktig:** Vente/varsle-synkronisering har to varianter. Forskjellen på dem er om de reagerer på betingelser som allerede er oppfylt.

## Vente/varsle i Java – 1. utkast

Vi skal nå lage Java-kode som skaper den ønskede vente/varsle-synkro-niseringen som vi har skissert i forrige avsnitt. Listing 6-1 viser en Java-klasse som representerer en betingelse. Trådene som har en referanse til et slikt objekt kan dermed vente på eller varsle at betin-gelsen er oppfylt.

Klassen inneholder en boolsk variabel som *varsle()*-metoden setter til «true». *Vente()*-metoden ligger i en løkke som avsluttes når variabelen settes til «true», dvs. når noen har kalt *varsle()*-metoden.

Inne i venteløkken ligger det et kall til metoden `Thread.sleep(10)`. Kallet resulterer at Java-tråden «sover» i 10 millisekunder. Vi kan fjerne denne linjen, med det resultat at denne tråden hele tiden vil teste variabelverdien. Dette forbruker en del av maskinens regnekapasitet til liten nytte. Denne situasjonen kalles «busy waiting», og er noe vi forsøker å unngå når vi programmerer med flere tråder.

Vi velger derfor heller å teste variabelen 100 ganger i sekundet, noe som gir maskinen mer tid til å utføre andre tråder. Tallet 10 (millisekunder) er et kompromiss mellom systemøkonomi og responstid (hvor fort venteren kan reagere på et varsel) og kan gis andre verdier om det er ønskelig.

*Vente()*-metoden avslutter med å sette variabelen til *false*, for å «nullstille» varselet for senere *vente()*-kall.

```
public class Betingelse {
     private boolean oppfylt = false;
     public void varsle() {oppfylt = true;}
     public void vente() {
        while(!oppfylt) {
           try {
              Thread.sleep(10);
           } catch (InterruptedException e) {}
        }
        oppfylt = false;
     }
}
```

*Listing 6-1: Klassen Betingelse har enkle vente()- og varsle()-metoder*

Et eksempel på bruk av Betingelses-klassen finner vi i listing 6-2. Programmet starter en «tjenertråd» som utfører en kalkulasjon når betingelsen *t* er oppfylt. Programmet varsler om denne betingelsen når *a* har fått en ny verdi, og venter deretter på betingelsen *u*. Tjeneren varsler at betingelsen *u* er oppfylt når kalkuleringen er ferdig, og resultatet kan skrives ut.

```
public class Synk1 extends Thread {
     Betingelse t,u;
     int a,b;

     public Synk1() {
       t = new Betingelse();
       u = new Betingelse();
       a = 2;
       setDaemon(true); // For at JVM skal kunne
                        // avslutte
       start(); // Starter "tjeneren"
       while (a<100) {
         a++;
         t.varsle();
         u.vente();
         System.out.println("a=" + a + ", b=" + b);
       }
     }

     public void run() {
       // Dette er en liten tjener, som venter
       // på betingelsen t, deretter gjør en
       // enkel beregning, og oppfyller så
       // betingelsen u
       while (true) {
         t.vente();
         b = a*a;
         u.varsle();
       }
     }

     public static void main(String[] args) {new
Synk1();}
}
```

*Listing 6-2: Et Java-program som benytter Betingelses-klassen*

Program-listinger er en viktig del av pensum i boka. Pass på at du forstår programeksemplene. Kjør gjerne programmene på din egen maskin.

Dette utkastet til en klasse for å representere en betingelse virker som det skal i vårt lille eksempelprogram, men det har en stor svakhet: *Utkastet vårt virker ikke riktig når flere vil vente på samme betingelse.*

Tilfeller av synkronisering hvor flere tråder må vente på samme betingelse vil behandles senere i kapitlet. Da vil vi også vise en Betingelsesklasse som virker under slike omstendigheter.

# Tre typer av synkronisering

Vi kommer langt med å dele synkroniseringsbehov i tre typer: *reservasjon, klient/tjener* og *bufret dataflyt.*

## Reservasjon

I et system av samarbeidende tråder vil det være ønskelig å dele ressurser (jf. eksemplet med hvitløkpressen). En skriver er velegnet for deling mellom flere, men har den egenskapen at den bare kan brukes av én om gangen. Papirarkene kan ikke inneholde tekst fra flere utskrifter, og vi lar derfor én utskrift få gjøre seg ferdig med skriveren før den neste utskriften får lov til å starte.

Det synes opplagt at skriveren er en ressurs som bare kan brukes av én ad gangen, og at en *reservasjonsmekanisme* er påkrevd. En slik mekanisme kan synkronisere trådene over en «ledig»-betingelse (vente på ledig, varsle om ledig).

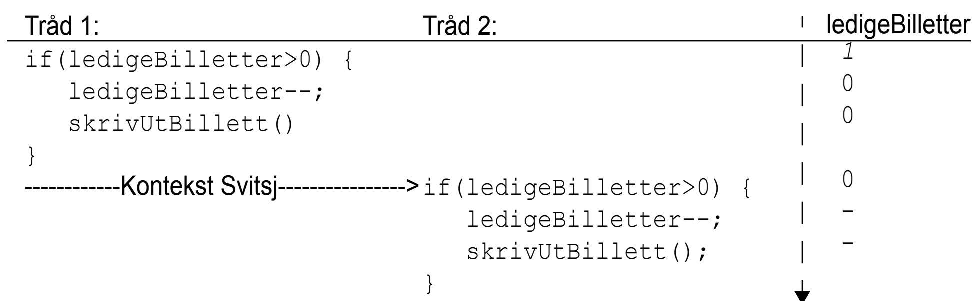
**Gjensidig utelukking (mutex)** En reservasjon krever en synkronisering som er en variant av den vi hittil har diskutert. Her kan det nemlig være flere enn to parter som deler en ressurs, med den følge at mer enn én av dem venter på en «ledig»-betingelse. Den varianten av synkronisering som behøves, kalles *gjensidig utelukking*, ofte forkortet *mutex* (mutual exclusion). Mutex henspeiler på det forhold at når én part har reservert en ressurs, må alle andre vente på at den skal bli ledig. Alle har en likeverdig adgang til å utelukke alle andre, og ressursen kan under normale forhold bare gjøres ledig av den som har reservert den. Den som har reservert en ting kaller vi ofte for «eieren».

**Kritiske regioner** Det er ikke bare utstyrsenheter som må kunne reserveres. Også *operasjoner på data* kan ha egenskaper som gjør at de må reserveres. La oss tenke oss følgende situasjon som et eksempel: Vi ønsker å bestille en billett dersom det er noen ledige. Det kan foregå med følgende kodesetninger:

```
if(ledigeBilletter>0) {
     ledigeBilletter--;
   skrivUtBillett()
}
```

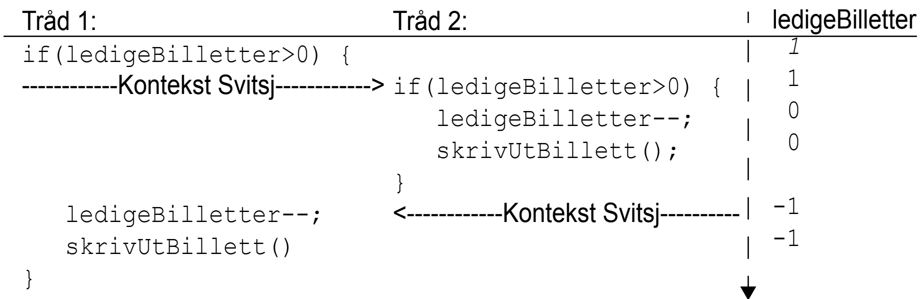
*Listing 6-3: Kodesetninger for å reservere en billett*

Om to tråder begge ønsker å bestille en billett, kan det skje med følgende forløp:




*Figur 6-3: To tråder reserverer billetter, riktig resultat. Tråd 2 konstaterer at det ikke er flere ledige billetter*

Forløpet på figur 6-3 er at tråd 1 fullfører billettbestillingen før det skjer en kontekst svitsj, og tråd 2 gjennomfører billettbestillingen. I dette tilfellet blir resultatet riktig. Figur 6-4 viser derimot et forløp som *ikke* blir riktig, for om det i dette tilfellet er én billett ledig, vil trådene etter tur finne én ledig billett, og begge vil bestille og skrive ut denne billetten. Resultatet er at variabelen ledigeBilletter blir lik -1, og vi har gitt to kunder den samme billetten.




*Figur 6-4: To tråder reserverer billetter, feil resultat*

Årsakene til at denne feilen oppstår er at

- flere tråder tester og endrer delte data (variabler eller objekter, i dette tilfellet *ledigeBilletter*)
- vi har ingen kontroll over når kontekst svitsj oppstår

Det er den samme programkoden som utføres i figur 6-3 og 6-4, forskjellen på riktig og feil resultat skyldes tidpunktene for kontekst svitsj. For å sikre et riktig resultat må vi derfor kontrollere kontekst svitsj slik at situasjonen på figur 6-4 ikke oppstår.

Kodelinjene vist på figurene utgjør en såkalt *kritisk region*, et område av programkoden som bare kan utføres av én tråd om gangen. En kritisk region kjennetegnes av at programkoden utfører sammensatte operasjoner på delte data (delt mellom flere tråder).

**Viktig:** En kritisk region oppstår der hvor koden oppdaterer delte data. En kritisk region kan bare bare utføres av én tråd om gangen.

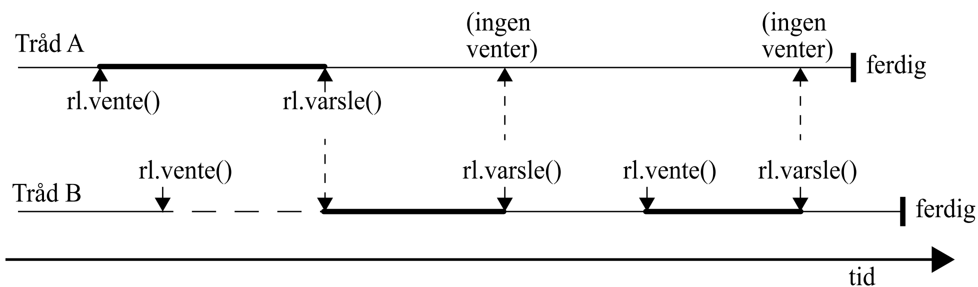
Vi kan bruke vente/varsle-begrepet for å kontrollere kontekst svitsj mellom tråder som utfører inne i en kritisk region. Ved å gjøre «kritisk region ledig» til en betingelse som vi kan vente på og varsle om, kan vi endre koden i eksemplet ovenfor slik:

```
regionLedig.vente();
if(ledigeBilletter>0) {
    ledigeBilletter--;
```

```
    skrivUtBillett()
}
regionLedig.varsle();
```

*Listing 6-4: Modifisert billettbestilling med vente/varsle-operasjoner*

Et tidsdiagram i likhet med det på figur 6-2 vil vise effekten av disse ekstra operasjonene når to tråder forsøker å reservere en billett samtidig (tråd nr. 2 starter på koden ovenfor før tråd nr. 1 er fullført). Figur 6-5 fremstiller effekten av vente/varsle-operasjonene, og kombinert med programkoden vist i figur 6-4 kan vi forsikre oss om at den tykke streken (utføring av den kritiske regionen) ikke overlappes i flere tråder.




*Figur 6-5: Bruk av vente/varsle rundt kritiske regioner. Den kritiske regionen er vist med tykk strek. Betingelsen «region ledig» er representert med objektet «rl»*

Legg merke til hva som skjer etter at tråd B har fullført den kritiske regionen og varsler om at den er ledig: Betingelsen «region ledig» er nå oppfylt selv om det ikke er noen som venter på den. Når tråd B senere gjør *rl.vente()* får den derfor fortsette uten opphold (jf. figur 6-2). Det er også en forutsetning at betingelsen «region ledig» er oppfylt fra starten av, slik at tråd A kan fortsette etter sin første *rl.vente()*-operasjon (til venstre på figur 6-5).

Denne virkemåten kan minne litt om toalettnøkkelen på en bensinstasjon. Om du ber om få bruke toalettet, titter ekspeditøren under disken for å se om nøkkelen henger på plass. Gjør den det, gir han deg nøkkelen og du gjør ærendet ditt. Om det i mellomtiden kommer en annen kunde og ber om å få bruke toalettet, vil ekspeditøren se at nøkkelen ikke henger der, og be kunden om å vente. Først når du kommer tilbake med nøkkelen kan neste kunde bruke toalettet, og når han er ferdig henges nøkkelen tilbake på plass som en allerede oppfylt «ledig»-betingelse. To personer kan ikke bruke toalettet samtidig, og

om den andre kunden hadde kommet ett sekund før deg, hadde du måttet vente. Dette er altså *mutex* - gjensidig utelukking. Startbetingelsen for at dette skal fungere er at toalettnøkkelen henger på plass når bensinstasjonen åpner om morgenen.

> Vi bruker uttrykket *race condition* om kritiske regioner som kan feile på grunn av tilfeldighetene ved kontekst svitsj. **Det er viktig å skrive programmene slik at de ikke inneholder race conditions!** Det oppnår vi ved å synkronisere adgangen til de kritiske regionene.

## Vente/varsle i Java – 2. utkast

**Organisere flere ventere** Der hvor flere kunder venter på toalettnøkkelen ønsker vi at de skal bruke toalettet i en ordnet rekkefølge, f.eks. i den rekkefølgen de kom og spurte. Klassen Betingelse (listing 6-1) har ingen slik køordning. Når betingelsen som objektet representerer blir oppfylt (gjennom at noen kaller *varsle()*-metoden), er det den første som oppdager dette som får fortsette. Vi skal se mange versjoner av Betingelses-klassen: Den neste lager en køorden av de trådene som venter, basert på den rekkefølgen de har kalt *vente()*-metoden:

```
public class Betingelse {
     private int varsleTeller=0,
           venteTeller=1;
     public void varsle() {varsleTeller++;}
     public void vente() {
        int minTeller=venteTeller;
        venteTeller++;
        while(minTeller>varsleTeller) {
          try {
             Thread.sleep(10);
          } catch (InterruptedException e) {}
        }
     }
}
```

*Listing 6-5: Betingelses-klasse som ordner køen av ventende tråder*

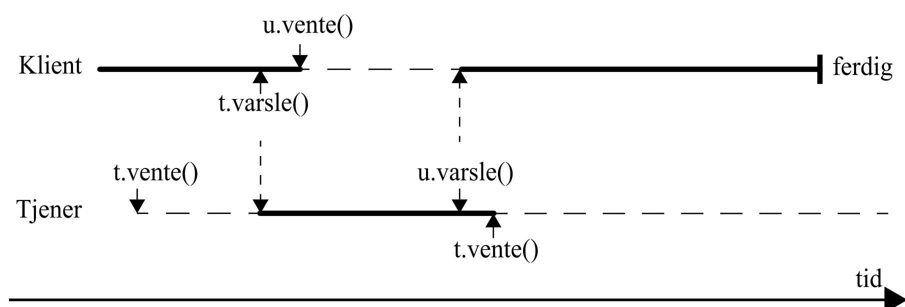
Også denne versjonen av Betingelses-klassen har sine svakheter. Den teller hendelser hele tiden oppover, og når den kommer til det største heltallet (MAXINT=$2^{31}$-1) vil den feile. En annen svakhet er at den i

seg selv inneholder ubeskyttede kritiske regioner (race condition)! Vi tar opp denne diskusjonen igjen litt senere i kapitlet under avsnittet "Kritiske mikroregioner".

## Klient/tjener

Denne typen synkronisering var det vi startet med å vise da vi introduserte vente/varsle-operasjonene i begynnelsen av kapitlet. Eksemplet fra kjøkkenet som trengte en reparatør for oppvaskmaskinen er også av denne typen.

**Likner metodekall** Klient/tjener-synkronisering kan også minne om et vanlig metodekall. Den som kaller metoden blir liksom klienten, og metoden blir liksom tjeneren. Programsetningen som kommer etter metodekallet blir ikke utført før etter at metoden har fullført sin utføring. Mens det ved metodekall er samme tråd som utfører både kallende program og metoden, ønsker vi nå å fokusere på bruk av uavhengige tråder som må påvirke hverandres utføringsstatus gjennom vente/varsle-operasjoner. Figur 6-6 viser et tidsforløp av klient/tjener-synkroniseringen fra programlisting 6-2 på side 125.




*Figur 6-6: Klient/tjener-synkronisering. Aktiv tråd er merket med tykk linje. Merk de overlappende områdene (tykk strek både hos klient og tjener)*

Klient/tjener-synkronisering vil ideelt sett være av den varianten som ikke reagerer på betingelser som allerede er oppfylt. En oppvaskmaskin kan ikke være ferdig reparert før man tilkaller operatøren, eller hva?

Figur 6-6 viser noen små detaljer som kan tyde på noe annet. Klienten bestiller en reparatør med metoden *t.varsle()*, og venter så på beskjed om at reparatøren er ferdig med den påfølgende metoden *u.vente()*. Hvor lang tid tar det mellom disse to kallene? Selv om de er påfølgende programsetninger, kan vi generelt sett aldri gi et svar (annet enn at tiden er større enn 0).

Når *u.vente()* blir kallet, har altså reparatøren allerede holdt på en liten stund (vist som overlappende tykke streker), og det er en mulighet for at han allerede er ferdig med jobben og har gitt beskjed om det med *u.varsle()*-metoden. Vi kan derfor ikke utelukke at reparasjonen *tilsynelatende* er ferdig før den er bestilt, og at en situasjon som vist i figur 6-2 kan oppstå.

Konklusjonen blir at det også i denne typen av synkronisering er nødvendig med vente/varsle-metoder som reagerer på betingelser som allerede er oppfylt.

**Asynkrone tjeneroperasjoner** I klient/tjener-sammenhenger vil vi ofte ha den situasjon at klienten kan gjøre annet arbeid mens tjeneren jobber, fremfor å sitte uvirksom og vente. I vårt eksempel kunne vi tenke oss at klienten skriver ut resultatet av én beregning mens tjeneren holder på med den neste beregningen. Programlisting 6-6 viser en modifisert konstruktør til klassen *Synk1* som utnytter denne muligheten. Legg merke til at det lages en kopi av a og b for å «låse» det gjeldende resultatet.

```
public Synk1() {
     t = new Betingelse();
     u = new Betingelse();
     a = 2;
     setDaemon(true); // For at JVM
                      //skal kunne avslutte
     start(); // Starter "tjeneren"
     t.varsle();
     while (a<100) {
       u.vente();
       int a1=a; int b1=b;
       a++;
       t.varsle();
       // Skriver ut mens tjeneren beregner ny verdi
       System.out.println("a="+a1+", b="+b1);
     }
}
```

*Listing 6-6: Overlappende utføring hos klient og tjener*

Vi kaller slike operasjoner for *asynkrone*, fordi de utføres uavhengig av hverandre. Vi vet ikke om det er klienten eller tjeneren som blir ferdig først, men det spiller heller ingen rolle; Kallet *u.vente()* blir et «møtested» for disse to aktivitetene, og i den påfølgende setningen kan klienten være sikker på at tjeneren har fullført sin oppgave.

**Flere tjenere i arbeid samtidig** Asynkrone tjeneroperasjoner kan vi med fordel benytte for å sette flere tjenere i arbeid samtidig. I likhet med de eksemplene vi nå har vist, kan flere tråder vente på hver sin betingelse for å starte en bestemt operasjon. Etter fullført arbeid oppfyller trådene hver sin betingelse, og klienten venter etter tur på alle disse tre betingelsene:

```
public Synk3() {
     // Objektene t1,t2,t3,u1,u2,u3 er
     // av klassen Betingelse
     // Dette er bare fragmenter av kode,
     // ikke et kjørbart program
     t1.varsle();  // Start tjeneren som venter på t1
     t2.varsle();  // Start tjeneren som venter på t2
     t3.varsle();  // Start tjeneren som venter på t3
     ... // Gjør noe i mellomtiden
     u1.vente();   // Fortsett når tjeneren
                   // «u1» er ferdig
     u2.vente();   // Fortsett når «u2» er ferdig
     u3.vente();   // Fortsett når «u3» er ferdig
     ... // Alle tjenerne er ferdige, fortsett
}
```

*Listing 6-7: Asynkrone operasjoner mot flere tjenere. Med uttrykket «u1» sikter vi til den tjenertråden som varsler fullført arbeid ved å oppfylle betingelsen u1*

Programmet i listing 6-7 sørger for at programsetningen etter *u3.vente()* ikke utføres før alle tre tjenerne er ferdige. Det spiller ingen rolle i hvilken rekkefølge tjenerne gjør seg ferdig, heller ikke i hvilken rekkefølge vi utfører de tre *vente()*-operasjonene, fordi det ikke spiller noen rolle om betingelsen allerede er oppfylt eller ikke.

For å lage flere tjenertråder må vi bruke en annen type Java-programmering enn hva vi har gjort i *Synk1*: Fordi vi lar applikasjonsobjektet arve fra `java.lang.Thread`, kan det bare romme én ekstra tråd. For å lage flere tråder må vi benytte én av følgende metoder:

- La Sync1 «`implements Runnable`», og lage nye tråder med
  `Thread tx=new Thread(this);`
- Lage interne klasser som «`extends Thread`», har sin separate `run()`-metode og kan utføre forskjellige oppgaver. Dette er den beste metoden. Eksempel på en klient med flere tjenere finnes som klassen *Synk3* på bokas nettsted.

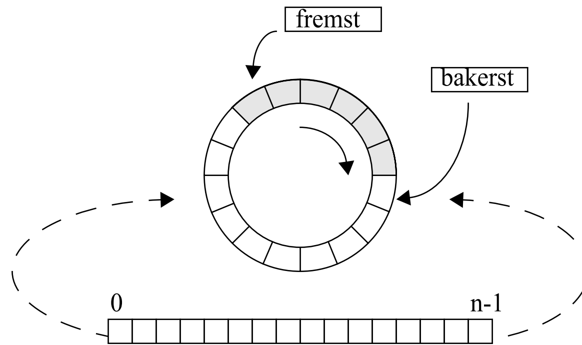
## Bufret dataflyt

En tredje type av synkroniseringsbehov finner vi under overføring av data mellom to tråder gjennom et felles minneområde (en buffer) med begrenset størrelse. Dette blir å sammenligne med kokkene Arne og Erik, som har en «flyt» av grønnsaker fra den ene til andre, mellomlagret på en arbeidsbenk av begrenset størrelse. Slike samarbeidsforhold vil vi kalle for *produsent/konsument*-samarbeid. Synkroniseringsbehovet i et slikt samarbeid vil være at

- produsenten skriver data til bufferen. Dersom bufferen er full, skal den forholde seg passiv inntil det blir ledig plass.
- konsumenten leser data fra bufferen. Dersom bufferen er tom, skal konsumenten forholde seg passiv inntil det kommer data (produsenten skriver).

**Ringbuffer** Dette er omtrent slik samarbeidet mellom kokkene Arne og Erik må foregå. En buffer som mellomlagrer data for dette formålet vil måtte tilby lesing og skriving på «FIFO-maner»[1] (First-In, First-Out) dvs. at data som ble skrevet først også skal leses først (vanlig køordning). En kø-buffer kan lages som en ringbuffer, et minneområde knyttet sammen i endene slik at det danner en ring. I tillegg til ringen trenger vi to pekere som kan holde orden på hvor fremst og bakerst i køen er, slik som vist på figur 6-7.

1. Vi bruker uttrykkene «FIFO» og «kø» synonymt.




*Figur 6-7: Kø-organisering av et minneområde. «Bakerst» er den plassen som det skal skrives til neste gang (første ledig). «Fremst» er den plassen det skal leses fra neste gang. «Bakerst» flyttes med klokken ved en skriveoperasjon, i likhet med «Fremst» ved en leseoperasjon. Skraverte plasser viser at de inneholder data*

En Java-klasse som representerer en FIFO kan se slik ut:

```
public class FIFO {
     private int fremst,bakerst,elem,kapasitet;
     public FIFO(int kap) {
       lager = new Object[kap];
       kapasitet = kap;
       fremst=0; bakerst=0; elem=0;
     }
     public void leggTil(Object obj) {
     // Skriv til køen
       if (elem < kapasitet) {
         lager[bakerst] = obj;
         bakerst = (bakerst+1) % kapasitet;
         elem++;
       }
     }

     public Object taUt() { // Les fra køen
       Object obj=null;
       if (elem > 0) {
         obj = lager[fremst];
         fremst = (fremst+1) % kapasitet;
         elem--;
       }
```

```
        return obj;
    }
}
```

*Listing 6-8: Java-klasse som skaper en FIFO-lagringsstruktur (kø)*

Denne koden sørger for at rekkefølgen på dataene ved skriveoperasjoner (*leggTil*) blir beholdt ved leseoperasjoner (*taUt*). Vi legger merke til at koden rommer data av klassen *Object*, og vi kan derfor bruke denne klassen til å organisere alle slags data. Legg merke til bruken av %-operatoren (divisjonsrest) for å få «fremst»- og «bakerst»-pekerne til å starte forfra igjen når de kommer til enden av arrayet. Det er slik vi oppnår ringstrukturen.

**Telle mange varsler** Vi mangler derimot fortsatt koden som kan sørge for den nødvendige synkroniseringen over tilstanden i ringbufferen (skriving til full buffer, og lesing fra tom buffer). For å få det til må vi ha en mulighet for å varsle om en oppfylt betingelse *mange ganger*. Om den tidligere nevnte bensinstasjonen har flere toaletter, vil det også være flere nøkler i omløp. Med begge toalettene ledig vil det henge to nøkler bak disken, som om betingelsen «toalett ledig» var **oppfylt to ganger**. Da kan to kunder komme til å vente på «toalett ledig» og få fortsette uten venting, mens den tredje kunden må ev. vente på at én av kundene skal oppfylle en «toalett ledig»-betingelse. Dette er enkelt å få til dersom

- en Betingelse kan telle hvor mange ganger betingelsen er oppfylt
- *Vente()*-operasjonen teller ned dette tallet
- betingelsen er oppfylt to ganger som en en starttilstand

Det skal bare ganske små endringer til i koden vist på listing 6-1 for å oppnå disse egenskapene[2]: Vi erstatter den boolske *oppfylt*-variabelen med en heltallsvariabel som telles opp i *varsle()*-metoden og ned i *vente()*-metoden. While-løkken tester om variabelen er lik 0:

```
public class Betingelse {
    private int oppfylt = 0;
    public Betingelse(int initverdi) {
        oppfylt=initverdi;
    }
```

2. Varianten av klassen Betingelse som er vist i listing 6-5 har allerede disse egenskapene, og kan brukes uten videre.

```
    public void varsle() {oppfylt++;}
    public void vente() {
      while(oppfylt==0) {
        try {
          Thread.sleep(10);
        } catch (InterruptedException e) {}
      }
      oppfylt--;
    }
}
```

*Listing 6-9: Variant av klassen Betingelse som teller flere varsler*

Når Betingelses-klassen har disse egenskapene er det mulig å lese fra en kø (*taUt*) dersom man først venter på at betingelsen «data i køen» er oppfylt. En skriveoperasjon (*leggTil*) vil på sin side oppfylle denne betingelsen. Betingelsen «data i køen» vil dermed være oppfylt akkurat så mange ganger som det er dataelementer (objekter) inne i køen. Det betyr at når en konsument vil lese fra en kø som er tom (ingen dataelementer i køen), vil den forbli passiv inntil en produsent kommer og skriver til køen.

Omvendt vil en produsent vente på at betingelsen «ledig plass» skal være oppfylt før vi skriver til køen. I en tom kø med 10 plasser vil denne betingelsen være oppfylt 10 ganger, like mange ganger kan en produsent gjøre skriveoperasjoner. Den 11. gangen vil vente()-operasjonen gjøre produsenten passiv inntil konsumenten har lest data og varslet om ledig plass.

Vi vil bygge inn de nødvendige vente()- og varsle()-operasjonene i selve FIFO-klassen, slik at denne synkroniseringen skal skje automatisk ved lese- og skriveoperasjoner i køen. Listing 6-10 viser den modifiserte Java-koden med endringene i uthevet skrift:

```
public class FIFO {
    private int fremst,bakerst,kapasitet;
    private Betingelse dataElementer,ledigPlass;
    private Object[] lager;
    public FIFO(int kap) {
      lager = new Object[kap];
      kapasitet = kap;
      dataElementer = new Betingelse(0);
      // ledigPlass er «forhåndsoppfylt» kap ganger
```

```
            ledigPlass = new Betingelse(kap);
            fremst=0; bakerst=0;
        }
        public void leggTil(Object obj) {
            ledigPlass.vente();
            lager[bakerst] = obj;
            bakerst = (bakerst+1) % kapasitet;
            dataElementer.varsle();
        }
        public Object taUt() {
            Object obj;
            dataElementer.vente();
            obj = lager[fremst];
            fremst = (fremst+1) % kapasitet;
            ledigPlass.varsle();
            return obj;
        }
}
```

*Listing 6-10: Modifisert FIFO-klasse som synkroniserer konsument og produsent over tilstanden i køen (kø full/kø tom)*

Legg merke til fraværet av *items*-variabelen, og if-setningene som kontrollerte om køen er full eller tom. Den funksjonen er nå overlatt til Betingelses-objektene (*dataElementer, ledigPlass*). Listing 6-11viser et testprogram som demonstrerer hvordan en tjenertråd (produsenten) kan skrive data inn i en kø én gang i sekundet, og hvordan klientprogrammet (konsumenten) kan synkronisere seg med denne datastrømmen og skrive ut dataene etterhvert som de leses ut fra køen:

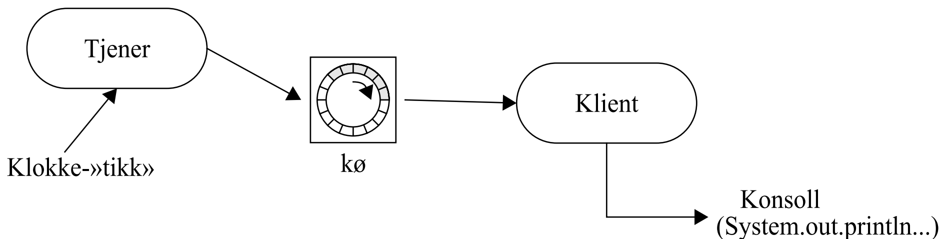
```
// Testprogram for synkronisert FIFO
public class Synk4 implements Runnable {
        FIFO q;
        public Synk4() {
            q = new FIFO(10);
            Thread t = new Thread(this);
            t.setDaemon(true);
            t.start();
            while (true) {
                // Kallet taUT() leser (synkronisert)
                // fra køen
                float f = ((Float)q.taUt()).floatValue();
                System.out.println("Verdi fra kø er: "+f);
```

```
        }
      }

      public void run() {
        while (true) {
          Float f = new Float(Math.random()*1000);
          // leggTil() skriver (synkronisert)
          // til køen
          q.leggTil(f);
          try { Thread.sleep(1000); }
          catch (InterruptedException e) {}
        }
      }

      public static void main(String[] args) {
        new Synk4();
      }
}
```

*Listing 6-11: Testprogram for FIFO-klassen. Resultatet er tilfeldige tall som skrives ut én gang i sekundet*




*Figur 6-8: Figurativ fremstilling av programlisting 6-11*

> **Merk:** De versjonene av Betingelses- og FIFO-klasser som vi har brukt i denne gjennomgåelsen har noen svakheter som gjør dem uegnet for bruk i virkelige anvendelser (de inneholder Race Conditions). I neste kapittel skal vi se versjoner som er helt korrekte.

**Dekker nye behov** Bufret dataflyt er en interessant teknikk fordi den dekker både et dataoverføringsbehov og et synkroniseringsbehov. Fordelen er bl.a.

- større «frikopling» mellom klient og tjener (mer asynkron operasjon).
- velegnet for dataflyt-orienterte anvendelser (pipelining, jf. avsnittet “Når trenger vi multiprosessorer?” på side 38).
- lettere å realisere mange-til-en og én-til-mange synkronisering (mange produsenter, én konsument, eller én produsent, mange konsumenter).
- enkel og abstrakt programmeringsmodell.

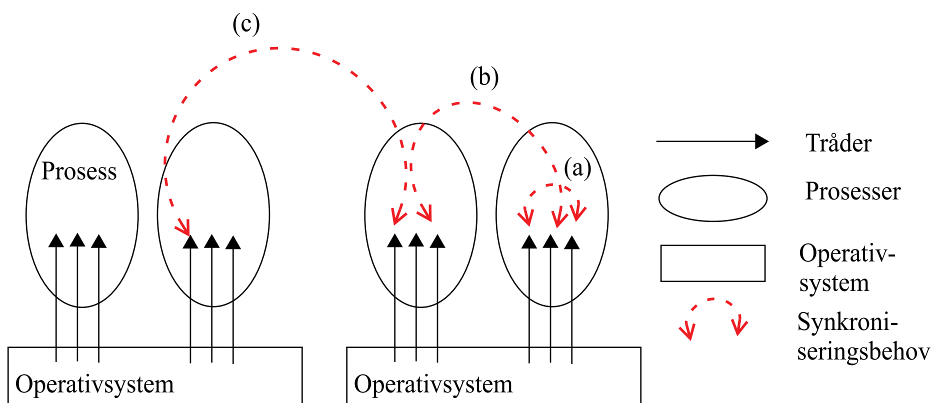
## Synkronisering internt i operativsystemet

Vi har nå beskrevet tre synkroniseringstyper (reservasjon, klient/tjener og bufret dataflyt) som et fenomen som foregår mellom tråder i samme prosess. Inne i operativsystemet er det naturligvis mange synkroniseringsbehov:

- Data skal flyttes mellom Device Driver og operativsystemets prosesser på et synkronisert måte (bufret dataflyt)
- En login-prosess skal startes hver gang noen kopler seg på en av terminalportene (klient/tjener)
- Beskytte et stort antall kritiske regioner (reservasjon) mot samtidig bruk av tråder i forskjellige prosesser

Synkronisering i operativsystemet vil derfor ikke dreie seg om kun synkronisering mellom tråder i samme prosess, men mellom tråder i forskjellige prosesser (og i neste omgang mellom tråder i forskjellige maskiner). Dette forholdet vises i figur 6-9 (jf. figur 4-3 på side 84).

Mellom tråder i prosesser som ikke deler minne kan vi ikke basere synkroniseringsmekanismene på Betingelses-objekter eller lignende innretninger som krever at trådene deler minne. Operativsystemet må derfor tilby synkroniseringsmekanismer til prosessene slik at tråder i ulike prosesser kan synkronisere seg med hverandre. Alle operativsystemer tilbyr dette, men på måter som ikke er portable (Windows og Linux har helt ulike mekanismer).




*Figur 6-9: Synkroniseringsbehov i operativsystemet. (a) viser synkronisering mellom tråder i samme prosess (vist i dette kapitlet). (b) viser synkronisering mellom tråder i forskjellige prosesser (besørges av operativsystemet). (c) viser synkronisering mellom tråder i ulike maskiner (behandles i kapitlet «Distribusjon»)*

**Synkronisering besørget av operativsystemet** Når operativsystemet tilbyr synkronisering mellom tråder i forskjellige prosesser vil dette basere seg på mekanismer som ligner på Betingelses-objekter[3]. Tråder i forskjellige prosesser deler ikke brukerminne, men operativsystemet kan knytte dem til minneområder inne i operativsystemet hvor Betingelses-objektene kan ligge. Operasjoner på disse objektene skjer da ikke med direkte metodekall, men gjennom operativsystemets API. Det samme gjelder for bufret dataflyt, som også baserer seg på at partene kan dele minneplass (FIFO-objekter) inne i operativsystemets minneområde. Ord som «Named Pipes» (Windows) og «pipes» (Linux) betegner operativsystemtjener som flytter data synkronisert mellom tråder i ulike prosesser.

**Synkroniseringsbehov i avbruddsrutiner** Det kan være en avbruddsrutine som «oppdager» at en betingelse er oppfylt og vil varsle om dette. Det kan f.eks. dreie seg om betingelsen «diskoperasjon ferdig», som en tråd i et brukerprogram venter på, og som nå kan fortsette utføringen. Det er ganske vanlig at en tråd forholder seg passiv mens den venter på at maskinvaren fullfører en oppgave, og da er varsling fra avbruddsrutinen den teknikken som gjør at tråden fortsetter sin utføring etterpå.

---

3. Et vanlig navn på Betingelses-objekter brukt slik er «Semaphores».

## Kritiske mikroregioner

Om vi ser nøye på koden for Betingelses-klassene (listing 6-1, 6-5, 6-9) kan vi se at de alle innholder ubeskyttede kritiske regioner (race conditions). Om det i likhet med situasjonen på figur 6-4 skjer en kontekst svitsj rett etter «while»-setningen i *vente()*-metoden, kan flere vente-tråder feilaktig konsumere samme varsel.

Selv setningen *oppfylt++* inneholder en kritisk region. Inne i CPU-en vil en slik setning bli utført av flere instruksjoner (eksemplet viser Intel-instruksjoner):

```
mov ax,[1465]       # Hent innholdet av oppfylt
inc ax              # Øke tallverdien med 1
mov [1465],ax       # Skriv den nye verdien tilbake
```

Mellom hver av disse instruksjonene kan det oppstå en kontekst svitsj. Derfor kan også en slik programsetning utgjøre en kritisk region dersom det er en variabel som er delt mellom flere tråder.

*Vi har altså den situasjonen at Betingelses-klassen, som vi har brukt til å beskytte kritiske regioner, selv inneholder kritiske regioner!*

Vi må ty til andre metoder for å beskytte de kritiske regionene som oppstår på «mikronivå». Et operativsystem som tilbyr Betingelses-mekanismer kan bruke disse metodene:

**Stenge av avbrudd** På en maskin med bare én CPU kan vi gjøre CPU-en «døv» for avbruddssignaler et lite øyeblikk. I denne perioden vil det ikke skje kontekst-svitsj (fordi den er avhengig av klokkeavbrudd), og vi unngår da den typen feil vi har sett kan oppstå som følge av kontekst svitsj inne i en kritisk region.

Det å stenge av avbrudd er et grovkornet virkemiddel som også gjør maskinen døv for i/o-operasjoner, tastatur, mus m.m., og er noe vi aldri vil drømme om å la et brukerprogram få lov til (den nødvendige instruksjonen er dessuten en privilegert instruksjon og ulovlig brukt i «user mode»). Operativsystemet kan derimot *selv* betjene en slik mekanisme i forbindelse med at det tilbyr Betingelser og vente/varsle-operasjoner. Og siden disse operasjonene kun medfører svært korte perioder hvor avbruddene er avstengt, er dette blitt en vanlig metode.

**«Read/Modify/Write» buss-syklus** Annerledes blir det i maskiner med flere CPU-er med delt minne (tett koplet multiprosessor). Tråder som utføres på hver sin CPU er ikke avhengig av kontekst svitsj for å flette sammen sin instruksjonsutføring i en kritisk region slik at det oppstår feil.

Løsningen ligger i å flytte den kritiske regionen inn i én og samme buss-syklus. I avsnittet “En buss-syklus” på side 31 viste vi hvordan en CPU kan lese, teste/modifisere og skrive tilbake i én udelelig operasjon, og ingen andre CPU-er kan benytte bussene i dette tidsrommet. Enkel betingelseshåndtering (som vist på listing 6-2 og 6-9) kan skrives slik at de kritiske regionene utføres innenfor en enkel buss-syklus.

## Sammendrag

- Synkronisering kan deles i tre typer: Reservasjon, Klient/tjener og Bufret dataflyt.
- Et Betingelses-objekt med vente/varsle-metoder kan løse de fleste synkroniseringsbehov forutsatt at (1) de ordner køen av ventende tråder, og at (2) de teller hvor mange ganger betingelsen er oppfylt.
- Kritiske regioner er uunngåelig der hvor tråder deler data.
- Reservasjon er synkronisering av kritiske regioner for å unngå race conditions.
- Klient/tjener-operasjoner minner om metodekall, men kan også innebære overlappede eller asynkrone tjeneroperasjoner.
- Bufret dataflyt bruker Betingelses-objekter til å holde styr på dataelementer og ledige plasser i databufferen. Bufret dataflyt løser både et dataoverføringsbehov og et synkroniseringsbehov.
- Der trådene ikke deler minne, må deres behov for dataoverføring og synkronisering dekkes av operativsystemets tjenester (f.eks. Named Pipes).
- Kun operativsystemet tillates å stenge for avbrudd for å kontrollere kontekst svitsj inne i en kritisk region.

| **Sentrale begreper i dette kapitlet:** | |
|---|---|
| kritisk region | race condition |
| reservasjon | bufret dataflyt |
| kontekst svitsj | Betingelse – vente/varsle |
| FIFO, kø | tråder med/uten delt minne |

## Teorioppgaver

**Gå sammen i grupper og løs disse oppgavene:**

1. Lag et diagram over forløpet i programlisting 6-6, etter modell av figur 6-6.
2. Vis med tilsvarende figurer hvordan Betingelses-klassen i listing 6-5 sørger for at flere tråder som har gjort vente()-operasjon får fortsette i den rekkefølgen de kom.
3. Lag et testprogram i Java som bekrefter at listing 6-5 virker som den skal.
4. Finn de ubeskyttede kritiske regionene (race conditions) i FIFO-klassen på listing 6-10. Hvilke forutsetninger må være tilstede for at det skal oppstå feil på grunn av disse?
5. Gi eksempler på dataflyt mellom samarbeidende tråder i én-til-mange- og mange-til-én-sammenstilling.

## Øvingsoppgaver

Forslag til øvingsoppgaver ligger på bokas nettsted.

**Etter fullførte øvinger bør du beherske:**

1. Utvikling av Java-programmer med flere tråder, dvs. være fortrolig med programmering, kompilering (tolke feilmeldinger) og debugging.
2. Demonstrere race conditions og feil som følge av dette i et program du skriver selv.
3. Skrive program med 2 produsenter og 1 konsument i en anvendelse som benytter bufret dataflyt.
4. Argumentere for en enkel og robust programmeringsteknikk med bruk av få, men velprøvede Java-klasser.

# Kapittel 7

# Synkronisering II

*I dette kapitlet skal vi se nærmere på synkroniseringsmekanismene som er innebygd i Java, og hvordan de kan brukes i tråd med de teknikkene som vi utviklet i forrige kapittel. Vi skal også se på fenomenet vranglås, og hvilke forutsetninger som ligger til grunn for dette fenomenet.*

## Hvorfor innebygd synkronisering?

I forrige kapittel diskuterte vi behovet for synkronisering, og skrev noe Java-kode for å skape de nødvendige synkroniseringsmekanismene. De fleste synkroniseringsbehovene lot seg dekke med de to metodene vi laget i Betingelses-klassen, men det gjensto tre uløste problemer:

**Kritiske mikroregioner** Inne i koden for `vente()` og `varsle()` fant vi noen kritiske regioner. Der ble delte variabler testet og oppdatert, og da vet vi at det oppstår kritiske regioner. Vi hadde altså forsøkt å løse reservasjonsproblemet, men den koden vi laget viste seg å selv inneholde et reservasjonsbehov.

Men forrige kapittel inneholdt også en beskrivelse av hvordan operativsystemet kan beskytte kortvarige kritiske regioner ved å «stenge av» avbruddsbehandlingen et kort øyeblikk. I multiprosess-maskiner kan noe lignende oppnås gjennom å bruke buss-sykler som utfører «read/modify/write» i én og samme buss-operasjon.

**Synkronisering av tråder i forskjellige prosesser** Med Betingelses-objekter i Java kan vi kun synkronisere tråder som har felles minne. Det vi også har bruk for er å synkronisere tråder som tilhører forskjellige prosesser, men som ikke deler minne.

**Trådenes utføringsstatus** Vi har også sett hvordan en tråd inne i `vente()`-metoden blir gående i en løkke hvor den ofte (100 ganger pr. sekund i vårt tilfelle) kontrollerer status på «oppfylt»-variabelen (listing 6-9 side 137). 100 ganger i sekundet skal derfor dette skje med den ventende tråden:

- Dens utføringsstatus skal skifte fra *blocked* (sovende) til *ready* fordi soveperioden på 10 millisekunder er utløpt. Det er operativsystemets avbruddsrutine for timer-avbrudd som må sørge for dette. Tråden er nå klar til å fortsette utføringen.
- Dens utføringsstatus skal deretter skifte fra *ready* til *running* som et resultat av en kontekst svtisj, hvor denne tråden blir plukket ut som «kandidat» for å få bruke CPU-en og fortsette sin utføring.
- Tråden returnerer nå fra `Thread.sleep(10)`-kallet, og kontrollerer «oppfylt»-variabelen.
- Dersom betingelsen ennå ikke er oppfylt, kalles `Thread.sleep(10)` igjen.
- Som et resultat av dette endres trådens utføringsstatus fra *running* til *blocked*, og en ny kontekst svitsj blir gjennomført.

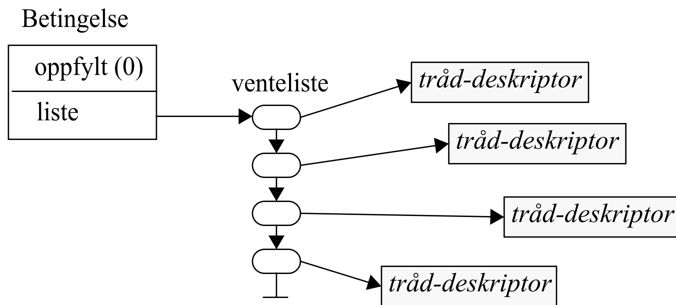
Dette er ganske mye administrasjon og ressursforbruk bare for å holde øye med en tellevariabel. Fullt mulig, men altså ganske uøkonomisk.

**Vi ønsker at tråden forblir i *blocked*-tilstand inntil betingelsen er oppfylt.**

For å oppnå dette må vi knytte `vente()`- og `varsle()`-metodene til operativsystemets mekanismer for å endre trådenes utføringsstatus:

- I forbindelse med en `vente()`-operasjon kan den kallende tråden få endret sin utføringsstatus fra *running* til *blocked*
- I forbindelse med en `varsle()`-operasjon kan en tråd som venter på denne betingelsen få endret sin utføringsstatus fra *blocked* til *ready*

Dette forholdet er vist på figur 4-2 på side 77. Det synes åpenbart at en slik variant av synkroniseringsmetoder er noe som operativsystemet må levere, så la oss se på hvordan det kan skje:




*Figur 7-1: Datastrukturen for en Betingelse. Bare når «oppfylt» er 0 kan det være tråder på ventelisten*

## Semaforer

Figur 7-1viser en mulig datastruktur for en Betingelse som oppfyller våre ønsker. En teller som holder rede på hvor mange ganger betingelsen er oppfylt, og en lenket liste med pekere til tråd-deskriptorer som holder rede på de trådene som venter på at betingelsen skal oppfylles. Dersom vi tenker oss at et objekt som representerer en tråd har metodene

`block()` – endrer trådens utføringsstatus til *blocked*
`unblock()` – endrer trådens utføringsstatus til *ready*

så kan vi skissere algoritmene for `vente()` og `varsle()` slik:

```
void vente() {
   Thread me;
   if (oppfylt==0) {
      me = Thread.currentThread();
      addToList(me);
      me.block();
   } else {
      oppfylt--;
   }
}
```

```
void varsle() {
   Thread him;
   if (!listEmpty()) {
      him = getFromList();
      him.unblock();
   } else {
      oppfylt++;
   }
}
```

*Listing 7-1: Algoritmene for vente() og varsle() laget som operativsystemtjenester*

Merk at ingen av disse metodene skaper en kontekst svitsj. Begge metodene inneholder kritiske mikroregioner, og må derfor beskyttes ved å stenge av avbrudd eller benytte read/modify/write buss-sykler som omtalt i forrige kapittel.

Med disse mekanismene oppnår vi den samme virkemåten som vi har vist med Betingelses-klassen, men unngår de problemene vi beskrev ovenfor.

Virkemåten til Betingelses-klassen ble beskrevet av Dijkstra i 1968, og kalt *semaforer*. Operasjonene på en semafor blir ofte kalt `P()` (vente) og `V()` (varsle). En *binær semafor* lar bare betingelsen bli oppfylt én gang mellom hver `vente()`-operasjon, og ignorerer overtallige `varsle()`-operasjoner utover denne ene (slik som i listing 6-1 på side 124).

En semafor kan brukes til reservasjon på samme måte som Betingelses-objektet: En `P()`-operasjon i toppen av den kritiske regionen, og en `V()`-operasjon i slutten, dessuten må semaforen være initialisert med oppfylt=1 (toalettnøkkel klar på kroken). Tilsvarende kan en FIFO styres av to semaforer, men bare dersom trådene deler minne (de må begge ha ringbufferen tilgjengelig).

# Javas synkroniseringsmekanismer

Alle operativsystemer tilbyr en eller annen form for innebygde synkroniseringstjenester, og semaforer er en vanlig virkemåte. Ofte tilbys «mutex» som en lignende tjeneste, men kan bare brukes til reservasjon (gjensidig utelukking). Java benytter seg av de innebygde synkroniseringstjenestene i operativsystemet til å tilby sin egen form for synkronisering. Til tross for at operativsystemene har forskjeller seg i mellom, vil Java tilby disse mekanismene likt på alle slags plattformer. Vi skal gå nøye gjennom Javas mekanismer og etterhvert se hvordan vi kan utvikle Betingelses- og FIFO-klasser som er uten feil.

## Synchronized-ordet

Ved å deklarere en metode som *synchronized* er vi garantert at den bare kan utføres av én om gangen! La oss se på enda en ny utgave av Betingelses-klassen:

```
class Betingelse {
     private int oppfylt=0;
     public Betingelse(int initverdi) {
       oppfylt=initverdi;
     }
     public synchronized void varsle() {oppfylt++;}

     private synchronized boolean erOppfylt() {
       if (oppfylt==0) {
         return false;
       } else {
         oppfylt--;
         return true;
       }
     }
     public void vente() {
       while(!erOppfylt())
       try {
         Thread.sleep(10);
       } catch (InterruptedException e) {}
     }
}
```

*Listing 7-2: En Betingelses-klasse som bruker synchronized-ordet*

Et Java-objekt er alltid assosiert med en «lås» (kanskje implementert som en semafor) som kan brukes til reservasjon av en kritisk region som måtte finnes i metodene. I listing 7-2 ser vi to metoder (`varsle()` og `erOppfylt()`) som er deklarert *synchronized* for å beskytte *én kritisk region*, nemlig den som oppstår under delt behandling av variabelen `oppfylt`. Det innebærer at dersom tråd A holder på med utføringen av metoden `varsle()`, vil tråd B bli forhindret i å starte utføringen av metoden `erOppfylt()`. Dette foregår slik at tråd B får sin utføringsstatus endret til *blocked* inntil tråd A har forlatt metoden `varsle()`.

Dette kan foregå ved hjelp av de reservasjonsmekanismene som operativsystemet måtte tilby. Dersom operativsystemet tilbyr semaforer, kan det assosieres en semafor til et Betingelses-objekt, og settes inn et kall til `P()`-metoden som første instruksjon i alle metoder som er deklarert

*synchronized*. Før hver utgang av metoden (legg merke til at `erOppfylt()` har to return-setninger) skal det settes inn et kall til semaforens `V()`-metode.

Med flere Betingelses-objekter brukes det flere semaforer som reserverer objektenes metoder uavhengig av hverandre. Et kall til objektets metoder reserverer den kritiske regionen kun i dette objektet.

**Monitorer** Vi *må* ikke ha synchronized-ordet. Dersom vi hadde adgang til å operere på operativsystemets semaforer kunne vi benytte teknikken nevnt ovenfor og oppnådd det samme resultatet. Java foretrekker derimot å bruke sin mekanisme fordi:

- Det er lettere å *deklarere* de kritiske regionene enn å *beskytte* dem. Med synchronized-ordet overlater vi til kompilatoren å knytte inngang og utgang av metodene til de reservasjonsmekanismene som operativsystemet tilbyr.
- Det reduserer mulighetene for programmeringsfeil (f.eks. om du glemmer en utgang og etterlater en kritisk region reservert for evig) og
- gjør Java-koden mer portabel (glatter ut forskjellene mellom operativsystem-tjenestene).

Teknikken med å innkapsle delte data sammen med den koden som behandler disse dataene setter oss i stand til å isolere en kritisk region inne i en klasse (samling av data og metoder). Slike klasser kalles *monitorer.*

> **Viktig:** *Semaforer* er en operativsystem-tjeneste som tilbys gjennom operativsystemets API. *Monitorer* er en språkegenskap som tilbys av programmeringsspråket.

Av de aktuelle programmeringsspråkene er det kun Java som tilbyr monitorer[1]. Vi omtaler det å utføre en synkronisert metode som å «holde monitoren».

## Wait/notify

Se igjen på programkoden i listing 7-2 og spør deg hvorfor vi ikke har deklarert den originale `vente()`-metoden som synchronized. Ser du hvorfor? Jo, det er fordi dersom en tråd holder monitoren mens den ligger i en løkke og venter på at en delt variabel blir endret, så venter den forgjeves: Ingen andre kan kalle metoden `varsle()` uten at monitoren «slippes».

1. Ada og Modula-2 tilbyr også monitorer, men betraktes som mindre aktuelle programmeringsspråk.

Derfor er `vente()`-metoden skrevet slik at den går inn og tester/modifiserer oppfylt-verdien inne i en metode som er synchronized `(erOppfylt()`, og slipper så monitoren før den venter i 10 millisekunder.

Denne versjonen av Betingelses-klassen er «sikker» i den forstand at den ikke innholder feil, men den er samtidig uøkonomisk fordi den bruker mye maskinressurser (jf. avsnittet “Trådenes utføringsstatus” på side 146).

Den endelige versjonen av Betingelses-klassen kommer til å basere seg på en ekstra synkroniseringsmekanisme som Java knytter til ethvert objekt. Inne i metoder som er synchronized er disse to metodene alltid tilgjengelige:

**`wait()`** setter kallende tråd i *blocked* tilstand samtidig som monitoren «slippes» (reservasjonen opphører)

**`notify()`** dersom det er tråder som har gjort `wait()` på dette objektet (og er i *blocked* tilstand) blir én av dem satt i *ready* tilstand og kan fortsette utføringen etter at den igjen *har reservert monitoren*[2]. Dersom det ikke er noen tråder som venter vil ikke `notify()`-kallet ha noen effekt.

Vi har i disse to kallene en mekanisme som minner om en semafor, men med den viktige forskjell at `wait()` ikke reagerer på `notify()`-kall som er skjedd på et tidligere tidspunkt.

```
class Betingelse {
     private int oppfylt;
     public Betingelse(int initial)
       {oppfylt=initial; }
     public Betingelse() { this(0); }
     public synchronized void varsle()
       {oppfylt++; notify();}
     public synchronized void vente() {
       while (oppfylt==0)
         try {wait();}
         catch (InterruptedException e) {}
```

2. Som figur 7-2 viser, forblir den egentlig i *blocked* tilstand inntil monitoren igjen kan reserveres

```
        oppfylt--;
      }
}
```

*Listing 7-3: Betingelses-klasse som bruker synchronized, wait() og notify()*

I listing 7-3 ser vi at `vente()`-metoden kaller `wait()` dersom betingelsen ikke er oppfylt. En annen tråd kan deretter kalle `varsle()` som øker verdien av `oppfylt` for å vise at betingelsen nå er oppfylt, og så kalle `notify()`. Den trenger ikke å teste om det er noen tråder som venter; for i motsatt tilfelle har dette kallet ingen effekt.

Det siste vi nå skal spørre oss om er: Hvorfor er `wait()`-kallet i `vente()`-metoden omsluttet av en while-løkke, istedenfor en if-setning? For å forstå det må vi studere noen intrikate detaljer ved monitorens egenskaper:

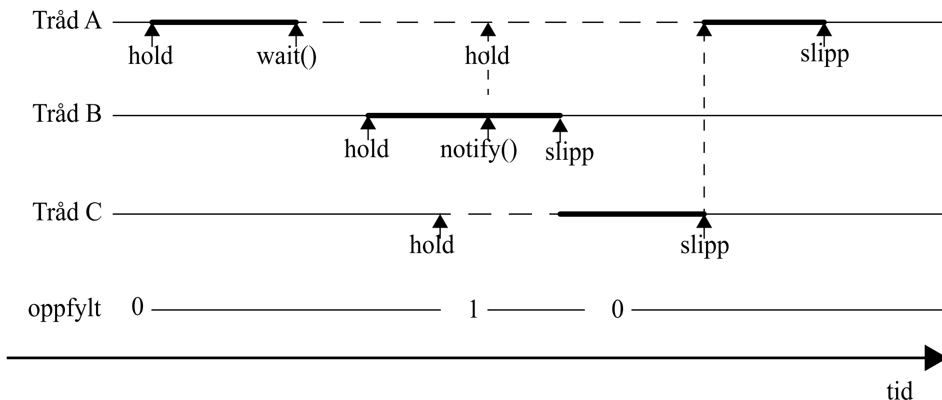
## Spesielle forhold ved monitorer

Idet en tråd kaller `wait()` endres dens utføringsstatus til *blocked*, og den opphever reservasjonen på monitoren. Grunnen til at tråden vil vente er som regel at den venter på at en eller annen betingelse skal bli oppfylt, her representert ved `oppfylt`-variabelen.

Når en annen tråd kaller `notify()`, er dette som regel for å varsle om at betingelsen er oppfylt, og én av de ventende trådene (gjerne den som har ventet lengst) får sin utføringsstatus endret til *ready og* vil kunne returnere fra `wait()`-kallet og fortsette sin utføring inne i monitoren etter at den igjen har *reservert monitoren*. Dette kan skje først når

- den andre tråden (den som kaller `notify()`) forlater monitoren og opphever reservasjonen av den
- alle andre tråder som allerede venter på å reservere monitoren har fullført sin bruk av monitoren og opphevet reservasjonen

I figur 7-2 viser vi et slikt forhold. Tråd A holder monitoren (vist med en tykk strek) og utfører et `wait()`-kall. Resultatet er at tråden settes til *blocked* og monitoren slippes. Senere kommer tråd B og kaller `notify()`, noe som gjør at tråd A ber om å få reservert monitoren og fortsette utføringen (den forblir derimot i *blocked* tilstand inntil videre).




*Figur 7-2: Et tidsforløp av tre tråder som bruker en monitor. Tykk strek betyr at tråden «holder» monitoren. Stiplet strek betyr at tråden er i blocked modus. Legg merke til at tråd A forblir i blocked modus etter at tråd B kaller `notify()`, men heretter venter den på at monitoren skal bli ledig*

I mellomtiden har tråd C bedt om å få reservere monitoren, og når tråd B slipper monitoren, har tråd C ventet lenger enn tråd A. Derfor vil tråd C være den neste som reserverer monitoren, *ikke tråd A*.

Om vi med utgangspunkt i Betingelses-klassen i listing 7-3 tenker oss at tråd A og C utfører `vente()`-operasjoner, mens tråd B utfører en `varsle()`-operasjon, har vi i figur 7-2 vist hvordan verdien av `oppfylt`-variabelen endrer seg i løpet av et slikt forløp:

- Tråd A konstaterer at `oppfylt==0`, og venter på at den skal endres
- Tråd B setter `oppfylt=1`, og varsler om dette med `notify()`
- Tråd C kommer inn i `vente()`-metoden, finner `oppfylt==1`, og setter den til 0 og fortsetter (gjør ikke et `wait()`-kall)
- Tråd A returnerer fra `wait()`-kallet, men allikevel er `oppfylt==0`, fordi den betingelsen som er oppfylt av tråd B allerede er konsumert av tråd C, som har tatt «innersvingen» på tråd A

Fordi dette er et mulig forløp i all bruk av `wait()` og `notify()`, er det nødvendig at den betingelsen man venter på, blir kontrollert etter at `wait()`-kallet er returnert. Derfor blir det feil å bruke:

```
if (oppfylt==0) try { wait(); } catch (...) {...}
```

men riktig å bruke:

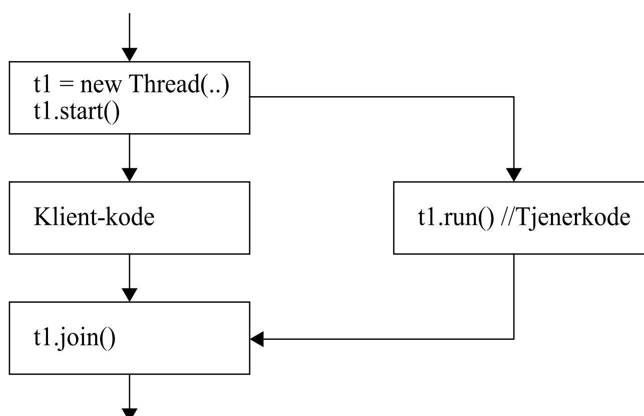
```
while (oppfylt==0) try { wait(); } catch (...) {...}
```

**Viktig:** Tilstanden i monitoren må alltid kontrolleres *etter* at wait()-kallet har returnert!

Javas synkroniseringsmekanismer er identisk med det som ellers i litteraturen kalles *monitorer*. Bruken av monitorer krever at man styrer klar av noen fallgruver som kan være litt vanskelig å få øye på.

## Join

Java tilbyr en rekke metoder på Thread-objekter. En av dem er `join()`, som blokkerer kallende tråd inntil denne tråden dør (opphører å eksistere). Metoden er nyttig når vi ber en tråd om å gjøre én eneste ting og så avslutte (tråden dør når `run()`-metoden «returnerer»), og vi trenger å vite at den er ferdig før vi passerer et visst punkt i vår egen programutføring.




*Figur 7-3: Bruk av `join()`-metoden skaper en møteplass mellom to tråder*

Figur 7-3 viser en vanlig måte å bruke `join()`-metoden på. En tråd skaper en annen tråd (dattertråd), og starter den slik at moren og datteren gjør sine oppgaver samtidig. På et senere tidspunkt trenger moren å forvisse seg om at dattertråden har fullført sine oppgaver før den selv fortsetter (kanskje fordi den trenger resultatene).

Problemstillingen er dermed lik den som ble beskrevet i avsnittet “Klient/tjener” på side 131, og bruk av `join()` -metoden er et alternativ til bruk av vente/varsle-metoder på et Betingelses-objekt. Listing 7-

4 viser bruk av `join()` for å oppnå overlappende klient/tjener-operasjon i likhet med listing 6-6 på side 132, der vi brukte Betingelses-objekter for å oppnå det samme resultatet.

```
// Eksempel på enkel varsle/vente-synkronisering,
// basert på nye tråder for hver operasjon
// bruk av join-operasjoner
public class Synk2_2 extends Thread {
     int a,b;
     Thread t;

     public Synk2_2() {
       a = 2;
       t = new Thread(this);
       t.start(); // Starter "tjeneren"

       while (a<100) {
         try {
           t.join(); // Vent her til t er død
         } catch (InterruptedException e) {}
         int a1=a; int b1=b;
         a++;
         t = new Thread(this);
         t.start();
         System.out.println("a="+a1+", b="+b1);
       }
     }

     public void run() {
       b = a*a;
     }

     public static void main(String[] args) {
       new Synk2_2();
     }
}
```

*Listing 7-4: Klient/tjener -synkronisering med bruk av `join()`-metoden. Jf. listing 6-6 på side 132*

## Sikker versjon av FIFO

Med kjennskap til synchronized-ordet er det nå mulig å lage en sikker versjon FIFO-klassen. Vi har i listing 7-3 oppnådd en Betingelses-klasse som er sikker og økonomisk, og som vi trygt kan bruke heretter.

FIFO-klassen bruker Betingelses-klassen for å dekke sitt synkroniseringsbehov, men inneholder fortsatt noen kritiske regioner som vi ikke har diskutert ennå.

I forbindelse med innsetting og uttak av data i ringbufferen blir de delte variablene `fremst` og `bakerst` lest og endret, og danner således en kritisk region som må beskyttes med en reservasjonsmekanisme.

**Vi kan ikke deklarere taUt() og leggTil() som synchronized for å oppnå dette!** (Begrunnelsen for dette er en av teorioppgavene ved slutten av kapitlet.) Vi må derimot omkalfatre litt på koden slik at den ser ut som på listing 7-5:

```
public class FIFO {
     private int fremst,bakerst,kapasitet;
     private Betingelse dataElementer,ledigPlass;
     private Object[] lager;
     public FIFO(int kap) {
        lager = new Object[kap];
        kapasitet = kap;
        dataElementer = new Betingelse(0);
        ledigPlass = new Betingelse(kap);
        fremst=0; bakerst=0;
     }

     private synchronized void in(Object obj) {
     // Legg inn i ringbuffer
        lager[bakerst] = obj;
        bakerst = (bakerst+1) % kapasitet;
}

     private synchronized Object out() {
     // Ta ut av ringbuffer
        Object obj;
        obj = lager[fremst];
        fremst = (fremst+1) % kapasitet;
        return obj;
}
```

```
    public void leggTil(Object obj) {
        ledigPlass.vente();
        in(obj);
        dataElementer.varsle();
    }

    public Object taUt() {
        dataElementer.vente();
        Object obj = out();
        ledigPlass.varsle();
        return obj;
    }
}
```

*Listing 7-5: En FIFO-klasse med de kritiske regionene beskyttet i private metoder som er deklarert synchronized*

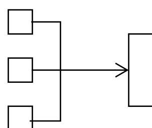
Når er det vi trenger slik beskyttelse? Vi ser i koden at metodene `taUt` og `leggTil` ikke deler noen variabler som danner en kritisk region (delingen av `lager[]` danner ikke en kritisk region), men at de hver for seg bruker variabler som danner kritiske regioner (rundt bruken av `fremst` og `bakerst`) dersom de blir delt mellom flere tråder.

Den beskyttelsen vi nå har laget er nødvendig først når det er flere som skriver til køen eller flere som leser fra den. Eksempler på dette har vi ikke studert ennå, men en sikker FIFO-klasse kan ikke ha som begrensning at den kun skal betjene bufret dataflyt med én produsent eller én konsument.

# Andre synkroniseringsvarianter

Vi har nå beskrevet hvordan en Betingelses-klasse kan tilby de nødvendige synkroniseringsmekanismene for å støtte tre viktige typer behov: reservasjon, klient/tjener og bufret dataflyt. Vi skal i dette avsnittet se på synkroniseringsbehov som bare delvis lar seg løse med Betingelses-objekter.

## Mange-til-én



I en sammenheng hvor en tjenertråd skal kunne starte en oppgave på ordre fra en rekke klienter vil det være nødvendig med en mange-til-én synkronisering.

- Én tråd styrer et alarmkonsoll, og mange andre tråder må kunne be om at alarmklokken ringer

- Én tråd sørger for å replikere[3] to databaser, og mange tråder må kunne be om at dette skal skje

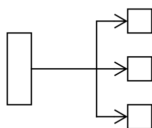
Et Betingelses-objekt kan brukes til slike formål, fordi mange tråder kan kalle `varsle()`-metoden. Allikevel oppstår det ofte behov for dataoverføring og retur-synkronisering ved slike anvendelser som et Betingelses-objekt ikke klarer å ta hånd om.

- Tråden som ber om at databasen blir replikert vil gjerne vite når dette er utført. Tjenertråden vil på sin side ikke vite hvem som kalte `varsle()`-metoden, så det er ikke mulig for den å sende noe varsel om at jobben er utført.
- Tråden som styrer alarmkonsollet vil kunne presentere en tekst i et display, men det krever at det i tillegg skjer en dataoverføring fra klienten til tjeneren. Betingelses-objektet kan heller ikke tilby det.

Begge disse problemene kan løses ved bufret dataflyt. Et FIFO-objekt kan overføre et Betingelses-objekt som tjeneren skal bruke til å varsle fullført oppgave, eller et String-objekt som inneholder en melding-stekst.

**Viktig:** Mange-til-en synkronisering løses best med bruk av bufret dataflyt.

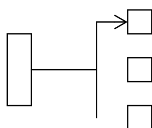
## Én-til-mange – notifyAll



**Kringkasting** Vi kan tenke oss en situasjon hvor mange tråder venter på samme «startskudd» for å fortsette sin utføring. Et eksempel på dette er at en ny fil er lagt til i en filkatalog, og alle trådene som viser vinduer med inneholdet i denne katalogen må oppdatere disse vinduene.

For å få dette til kan vi ikke bruke Betingelses-objektet eller semaforer. Det startskuddet som avfyres skal ikke «konsumeres» eller «huskes» slik tilfellet er i en semafor. Heller ikke skal den som avfyrer startskuddet trenge å vite hvor mange tråder som venter.

Java tilbyr en variant av `notify()` som heter `notifyAll()`, og dette kallet setter da ikke bare én tråd i stand til å fortsette, men alle tråder som har gjort `wait()` på dette objektet får fortsette.



**Room service** I andre situasjoner ønsker vi å sende et signal til én av mange mottakere (tjenere), men vi er ikke nøye på hvem av dem, fordi de alle gjør den samme jobben.

---

3. Med «replikering» mener vi å sørge for at innholdet er identisk

Denne mekanismen kan sammenlignes med det vi i forrige avsnitt kalte «mange-til-én», og på samme måte kan dette behovet dekkes med Betingelses-objekter eller FIFO-objekter.

En *flertråds tjener* kan fordele sine oppgaver mellom likeverdige tråder som alle kan utføre den samme oppgaven. Fordelingen kan skje ved at oppgavene blir beskrevet i objekter og lagt i en FIFO. Trådene leser data fra FIFOen når de er ledige for nye oppdrag. Slik fordeles oppgavene dynamisk på en måte som bidrar til at alle er beskjeftiget så lenge det finnes oppgaver i FIFO-en.

## Tidsgrenser

En annen synkroniseringsvariant er at en tråd ikke ønsker å vente på en betingelse uendelig lenge, men ha en tidsgrense for sin tålmodighet. Etter denne tidsgrensen fortsetter tråden allikevel, men må selvsagt reagere annerledes enn om betingelsen hadde vært oppfylt.

I Java finner vi flere metoder som tilbyr tidsgrenser ved venting. Både `wait()`- og `join()`-metodene beskrevet tidligere har varianter hvor en tidsgrense kan inngå som en parameter. Metodene gir ingen indikasjon om de returnerer som følge av at deres betingelse er oppfylt, eller om de returnerer som følge av at tidsgrensen er overskredet. Den som vil bruke disse metodevariantene må derfor skrive kode som ved retur fastslår hva tilstanden i systemet er. Ved bruk av `t.join(grense)`-metoden kan man teste om tråden `t` er død eller ikke ved metoden `t.isAlive()`. Figur 7-6 viser en modifisert del av listing 7-4 hvor tidsgrenser er tatt i bruk:

```
...
while (a<100) {
  try {
    t.join(1); // Venter max.1 millisekund
  } catch (InterruptedException e) {}
  if (t.isAlive()) {
    System.out.println("Tjeneren somler");
  }
  int a1=a; int b1=b;
...
```

*Listing 7-6: Klient/tjener-synkronisering med tidsbegrenset join(..)*

Synkronisering med tidsgrenser er nyttig i brukerdialoger hvor brukeren ikke gis ubegrenset tid til å svare, eller i nettverksprotokoller der mangel på tidsmessig respons fra motparten kan indikere at det er en feil i nettverket eller at motparten har krasjet.

## Vranglås

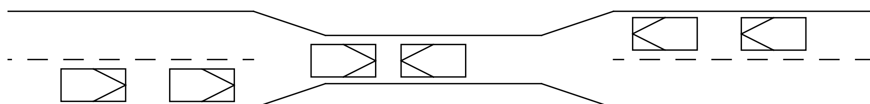
Et velkjent problem i forbindelse med reservasjoner er at de kan skape situasjoner der tråder forblir i *blocked* tilstand for alltid, uten mulighet for noen gang å komme videre i utføringen. Vi kaller slike situasjoner for *vranglås*.

Når en slik situasjon oppstår er systemet delvis «låst» og ute av stand til å fortsette utføringen. I sin tur vil en slik situasjon påvirke hele maskinens kjøremiljø og hindre fornuftig bruk av maskinen. En vranglås er derfor en alvorlig situasjon for datamaskinen, og det er viktig å redusere sannsynligheten for at den skal oppstå.

Vranglås er et «klassisk» emne innenfor operativsystem-faget, og er blitt studert så grundig at vi vet godt hvorfor det oppstår, og vi vet også en del om hvordan det kan unngås.

### Eksempler på vranglås

Et eksempel på en vranglås er den situasjonen som kan oppstå når to biler møtes midt på en smal bro. Begge er avhengig av å kunne reservere kjørebanen foran seg etterhvert som de kjører fremover, men fordi de kjører i motsatt retning oppstår det et problem som bare kan løses dersom én av dem rygger. To biler i samme kjøreretning vil aldri oppleve dette problemet.



*Figur 7-4: Eksempel på vranglås. To biler som møtes på en smal bro vil ikke bare hindre hverandre i å fortsette kjøringen, men også de bilene som ennå ikke er kommet ut på broen*

Et annet eksempel på vranglås er om du for en arbeidsoperasjon trenger sag og hammer. Du plukker opp sagen og oppdager at hammeren er opptatt. Du holder derfor på sagen mens du venter på at hammeren blir ledig. Det er bare det at den som har hammeren venter på at sagen skal

bli ledig, og holder på hammeren så lenge. Dersom ingen av dere oppdager at situasjonen kan løses med litt sunn fornuft, da blir dere stående og vente til evig tid.

I det virkelige liv er «rygging» den naturlige måten å løse slike situasjoner på, dvs. at man oppgir noen av sine reservasjoner for at en annen part skal kunne fullføre sine oppgaver, og deretter kan man selv forsøke igjen. Det gjør vi hele tiden i dagliglivet, men det er vanskelig å få en *tråd som utfører et program* til å rygge.

## Fire forutsetninger

Vi har i disse to eksemplene vist noen av forutsetningene for at vranglås skal oppstå.

**Alle disse fire forutsetningene må være oppfylt for at det skal oppstå vranglås:**

1. *Gjensidig utelukking:* En ressurs[4] må kunne reserveres slik at andre som ber om å få reservert den må vente inntil den blir ledig.
2. *Hold-og-vent:* Man kan holde reservasjonen på en ressurs mens man venter på at en annen blir ledig.
3. *Ingen tvungen tilbakelevering:* Kun den tråden som har reservert en ressurs kan frigjøre den igjen.
4. *Syklisk reservasjon:* Det må eksistere en kjede av reservasjoner og ønskede reservasjoner som danner et kretsløp.

Dersom én av disse forutsetningene brytes, da kan ikke vranglås oppstå, og vi vil nå derfor se nærmere på hvilke av forutsetningene som ev. kan fjernes fra systemet:

**Gjensidig utelukking** Kan vi klare oss uten mulighet for å reservere ressurser? Nei, det kan vi ikke. Idet vi ønsker å gjøre operasjoner på delte objekter eller data, da oppstår det kritiske regioner. Og den eneste metoden vi vet om for å sikre operasjoner i kritiske regioner, det er å bruke reservasjon.

**Hold-og-vent** Om du på forhånd vet at du trenger både en hammer og en sag for å utføre en oppgave, da kan vi tenke oss at du gjør en «begge-eller-ingen»-reservasjon. Du kontrollerer at begge gjenstandene er ledig, og reserverer dem i én og samme operasjon. Men sett at du skulle sage i én time, og bruker hammeren i ett minutt til slutt? Da ville hammeren være unyttig i den timen, og alle som bare trengte hammeren

---

4. Med *ressurs* mener vi data og objekter. De blir reservert «indirekte» gjennom å sperre adgangen til koden som behandler dataene (en kritisk region).

et øyeblikk ville måtte vente til du ble ferdig. En annen sak er at du kanskje ikke vet hva du trenger når du starter arbeidsoperasjonen. Resultatet av sagingen avgjør om du trenger hammeren eller ikke.

Komplett reservasjon av ressurser ved starten av en operasjon er derfor

- i fare for å gi dårlig utnyttelse av ressursene, fordi de blir reservert mye før det egentlig er bruk for dem. Det kan også medføre at flere ressurser enn nødvendig blir reservert.
- vanskelig å gjennomføre i praksis, fordi det ikke alltid er mulig å vite reservasjonsbehovet fra starten av. Resultatet av en operasjon inne i en kritisk region kan avgjøre om det er nødvendig å reservere mer.

**Ingen tvungen tilbakelevering** Dersom vi kan ha en »inspektør» som oppdager vranglås-situasjoner og som fratar en tråd en ressurs for å gi den til en annen tråd, da kunne vi «komme videre» og sørge for at i det minste noen oppgaver ble fullført. Eller kanskje at noen tråder blir erklært som «viktigere» enn andre slik at de kan tvinge til seg ressurser som andre har reservert? Det høres enkelt ut med sag og hammer, men inne i en datamaskin er bildet mer komplisert.

Hvorfor blir en ressurs reservert? Jo, oftest fordi det skal utføres operasjoner på delte data som i en periode vil være i en mellomtilstand. Om alle prisene i en varetabell skal justeres, må det skje slik at alle som leser priser fra tabellen kan vite om dette er gammel eller ny pris. Dette er ikke mulig å få til med mindre man utelukker slike operasjoner i en periode (ved hjelp av reservasjon). Om det i denne perioden skjer en tvungen tilbakelevering, kan det skje at en delvis oppdatert varetabell gjøres tilgjengelig for andre, som ikke kan vite om de leser gammel eller ny pris.

En måte å styre dette på er gjennom å «rulle tilbake»[5] endringene som er gjort i løpet av reservasjonsperioden. Dette blir i realiteten som om tråden hadde krasjet i løpet av den kritiske regionen. Det er uhyre ressurskrevende å skrive programmer slik at alle operasjoner i kritiske regioner kan tilbakestilles på denne måten, og det er sannsynligvis umulig i noen situasjoner.

**Syklisk reservasjon** Et eksempel på syklisk reservasjon er at du har sag og ønsker hammer, mens din motpart har hammer, men ønsker sag. Tegnet som en figur vil dette danne et kretsløp. Vranglåsen oppstår i dette tilfellet fordi dere reserverer objekter i motsatt rekkefølge. Bilene

---

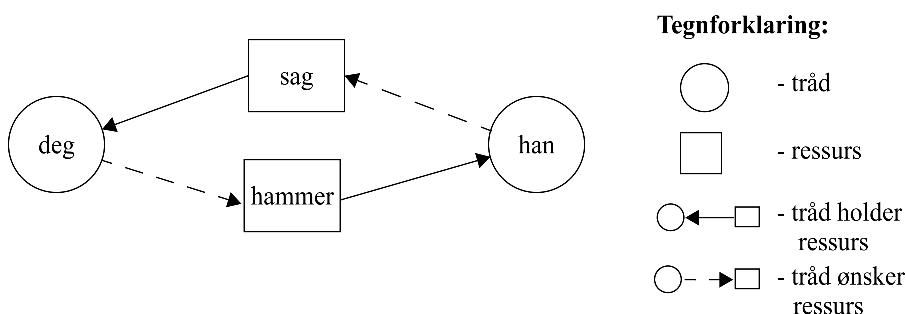
5. Dette begrepet henspeiler på databaseteorien og begrepet *transaksjoner*.

skaper vranglås over broen fordi de kjører i motsatt retning. Merk at kretsløpet kan involvere mer enn to parter, selv om vi ikke har vist dette i eksemplene.

Dette er den eneste forutsetningen som det er realistisk å klare seg uten. Om vi hadde en regel som sa «du kan reservere sag og deretter hammer, men aldri omvendt», så ville vi ha unngått det nevnte problemet, og smale broer kan gjøres lysregulerte for å sikre at alle biler kjører over i samme retning. Det er derimot ikke realistisk alltid å innrette ressursene i en datamaskin med en slags obligatorisk «reservasjon-srekkefølge»; et alternativ er å studere hver enkel reservasjon og se etter mulige kretsløp som dannes i den forbindelse. Som et hjelpemiddel for dette kan vi bruke en notasjonsteknikk som kalles *reservasjonsgraf*.

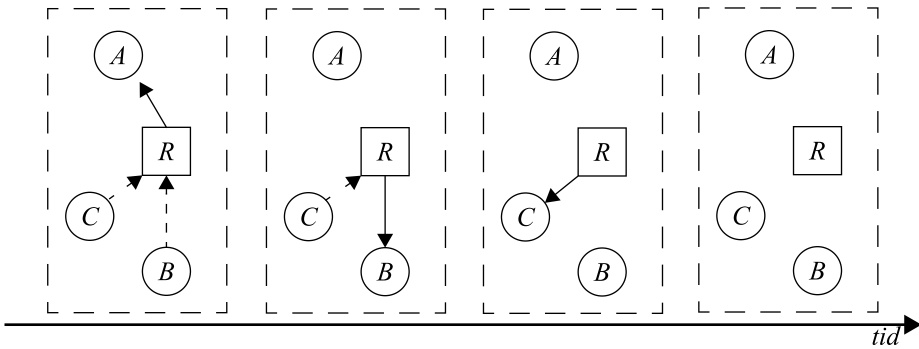
## Reservasjonsgrafer

Situasjonen med sag og hammer kan vi tegne slik som vist på figur 7-5. Her er reservasjoner tegnet som en heltrukket linje med pilspissen mot tråden, mens en ønsket reservasjon (et kall til `vente()` i Betingelses-objektet) er vist som en stiplet linje med pilspiss mot ressursen. Ved tegning av et reservasjonsforløp vil en «innvilget reservasjon» (retur av `vente()`-kallet) vises som en strek som skifter fra stiplet til heltrukken og som får snudd pilretningen.




*Figur 7-5: Reservasjonsgraf som viser vranglåsen med sag og hammer. Legg merke til at pilene i pilretning danner et kretsløp*

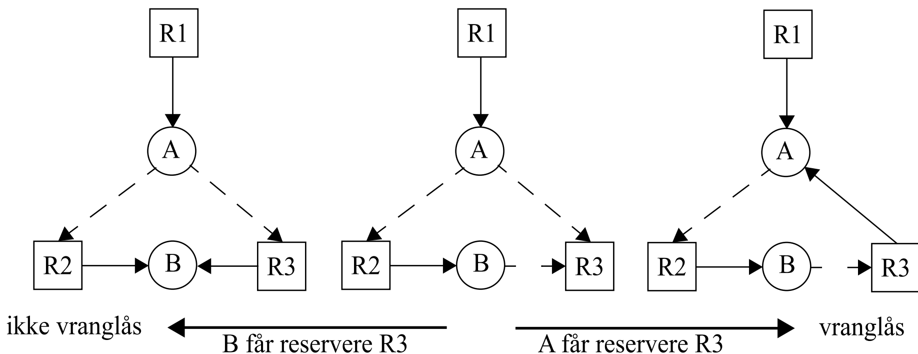
Et normalt forløp av tre tråder som konkurrerer om en ressurs kan tegnes som i figur 7-6. Her ser vi at tråd B og C venter (i *blocked* tilstand) på at ressursen R skal bli ledig. Når tråd A frigjør ressursen (linjen forsvinner), kan neste tråd reservere den (stiplet linje blir heltrukken) og fortsette sin utføring slik at ressursen senere frigjøres igjen.




*Figur 7-6: Normalt reservasjonsforløp. Tråd A, B og C bruker ressursen R etter tur. Deretter er ressursen ledig*

Vi kan bruke en reservasjonsgraf til å studere alternative forløp i skjemaer som vist ovenfor. I tilfellet ovenfor er det likegyldig hvilken rekkefølge trådene B og C får reservert ressursen, ingen vranglås vil oppstå uansett.

Et annet forløp er vist på figur 7-7. Her viser vi to alternative forløp som gir ulikt resultat. Både tråd A og tråd B ønsker å reservere ressursen R3 (figurens midtre del). Dersom tråd A får reservere R3 først (figurens høyre del) så oppstår det en vranglås (kretsløp av linjer i pilenes retning). Dersom B får reservere R3 (figurens venstre del) kan nå B fullføre sine oppgaver og deretter frigjøre R3 slik at tråd A kan reservere den.




*Figur 7-7: Hvilken tråd (A eller B) skal få reservere R3? Til høyre A, noe som skaper vranglås. Til venstre B, som ikke skaper vranglås. Tilstedeværelsen av R1 er bare «til pynt» og har ingen betydning for situasjonen*

### Unngå vranglås?

Ved å ha slike reservasjonsgrafer representert i operativsystemet kan vi overvåke reservasjoner og *oppdage* at vranglås er oppstått, slik at en form for oppretting kan finne sted. Vi kan med de samme teknikkene også forsøke å *unngå* vranglås-situasjoner slik som vist på figur 7-7. Algoritmene som er utviklet for dette formålet[6] kan også analysere hele scenarier av mange påfølgende reservasjoner, ikke bare den foreliggende ønsket. Slik kan operativsystemet forsøke å holde sine ressurser i en tilstand som gjør at vranglås ikke oppstår.

Men i en rekke situasjoner vil det ikke være mulig å være sikker på at vranglås ikke oppstår, og hva skal systemet gjøre da?

Systemet kan avvise reservasjonen, med den begrunnelse at den ikke er «vranglås-trygg». Den tråden som ønsker reservasjonen får da ikke fullført oppgavene sine, og må forsøke å «rulle tilbake» til sin starttilstand og prøve igjen en annen dag. Vi klarer kanskje å unngå vranglås, men vi hindrer samtidig utføringen av oppgaver som bare med en viss sannsynlighet ville bli blokkert. Vi skaper altså de samme problemene som vi løser, og konsekvensene blir uansett en redusert produksjon i datamaskinen.

Konklusjonen er at vranglås er et problem man bør klare å leve med. En fornuftig applikasjonsprogrammering kan redusere sannsynligheten for vranglås. Avanserte algoritmer for å unngå vranglås er lite brukt i moderne operativsystemer.

## Synkronisering i multiprosessorer

Vi har i diskusjonen om synkroniseringsmekanismer lagt til grunn at det eksisterer minneceller som alle partene kan bruke, og som kan lagre de nødvendige datastrukturene. Disse minnecellene kan ligge i prosessens minneområde i form av Betingelses-objekter der hvor de samarbeidende trådene tilhører samme prosess, eller i operativsystemets minneområde i form av semaforer eller lignende der trådene tilhører forskjellige prosesser.

I maskiner med flere CPU-er skiller vi mellom to konfigurasjoner: tett koplet og løst koplet (se avsnittet “Multiprosessorer” på side 36). I en *tett koplet* multiprosessor er situasjonen den samme som på en uniprosessor. Trådene har adgang til de samme minnecellene uansett hvilken CPU de utføres på, og de samme mekanismene for synkronisering kan brukes.

---

6. Den mest kjente algoritmen blir kalt «banker’s algorithm».

Noe annet er tilfellet i en *løst koplet* multiprosessor: Her kan tråder som kjøres på forskjellige CPU-er *ikke* samarbeide via felles minneceller, men må bruke en kommunikasjonskanal til å sende hverandre *meldinger*.

Vi har vist denne situasjonen på figur 6-9 på side 141 som alternativ (c). Synkroniseringsmekanismer som benytter meldinger i en kommunikasjonskanal blir nødvendig å bruke i løst koplede multiprosessorer så vel som i frittstående maskiner koplet sammen i et nettverk.

De teknikkene som vi har diskutert i dette kapitlet er derfor ikke anvendelig i en såkalt *distribuert konfigurasjon*. Vi vil i et senere kapittel diskutere distribuerte synkroniserings- og kommunikasjonsmekanismer.

## Sammendrag

- Vi ønsker at vente/varsle-mekanismer skal knyttes til kjøreplanen, slik at tråder som venter på en Betingelse holdes i *blocked* tilstand.
- En *semafor* oppfører seg som et Betingelses-objekt, men er en operativsystem-tjeneste som muliggjør synkronisering av tråder i forskjellige prosesser.
- Java tilbyr innebygde synkroniseringsmekanismer gjennom *synchronized*-ordet, samt `wait()` og `notify()`-metoder.
- En klasse med metoder som er deklarert synchronized kalles en *monitor*.
- Java tilbyr også eksplisitt synkronisering mellom tråder med `join()`-metoden.
- Betingelses-objekter og FIFO-objekter egner seg for mange-til-én og i noen grad én-til-mange (room service). Kringkastings-varsling krever bruk av `notifyAll()`.
- Vranglås er et uønsket fenomen som opptrer i forbindelse med reservasjon, og opptrer bare når fire bestemte forutsetninger er oppfylt.
- Vranglås kan unngås, men teknikken for dette er uøkonomisk og lite brukt. Vranglås er heller noe vi velger å leve med.
- Semaforer og Betingelses-objekter kan ikke brukes i en distribuert konfigurasjon. I slike tilfeller bruker vi meldingsbaserte teknikker.

| **Sentrale begreper i dette kapitlet:** | |
|---|---|
| semafor | monitor |
| synchronized,wait(),notify() | én-til-mange-synkronisering |

| Sentrale begreper i dette kapitlet: | |
|---|---|
| mange-til-én synkronisering | vranglås |
| reservasjonsgraf | multiprosessor-synkronisering |

# Teorioppgaver

**Gå sammen i grupper og løs disse oppgavene:**

1 I avsnittet “Sikker versjon av FIFO” på side 156 skrev vi at vi ikke kan beskytte taUt() og settInn() med synchronized-ordet. Hvorfor ikke? Beskriv et forløp der vi har gjort nettopp dette, og det to tråder ønsker å bruke disse metodene i et produsent/konsument-samarbeid.

2 På venstre side av figur 7-7 viser vi en reservasjon som ikke medfører fare for vranglås. Tegn forløpet for de resterende reservasjonene i likhet med figur 7-6 og vis at både tråd A og B klarer å fullføre sin utføring.

3 Kan vi ved å studere tilstanden til et Betingelses-objekt se *hvem* som har reservert et objekt?

4 Figur 6-6 og 7-4 viser to forskjellige programmeringsteknikker for å oppnå det samme resultatet. De skiller seg ut ulik bruk av systemressurser. I hvilke situasjoner vil du foretrekke den ene fremfor den andre av systemøkonomiske hensyn?

5 Skriv en Java-klasse som styrer én-til-mange-synkronisering i form av et «startskudd». Skriv også Java-kode for å teste denne klassen.

6 Tenk på en byggeplass. Hvilke muligheter finner du her for at vranglås-situasjoner kan oppstå? Forsøk å lage regler for å unngå slike situasjoner.

7 Beskriv en vranglås som involverer tre parter.

# Øvingsoppgaver

Forslag til øvingsoppgaver ligger på bokas nettsted.

**Etter fullførte øvinger bør du beherske:**

1. Java-programmering hvor flere tråder samarbeider gjennom bruk av monitorer.
2. Skrive programmer i flere tråder som skaper en vranglås-situasjon.
3. Forstå hvordan synkronisering med bruk av tidsgrenser kan kontrolleres.
4. Beherske én-til-mange-synkronisering med bruk av notifyAll().

Kapittel 8

# Permanent lagring

*Vi flytter nå fokus vekk fra synkroniseringteori og flertråds Java-programmering for igjen å se på oppbygningen av operativsystemet. Vi skal studere filstyringen, dvs. hvordan operativsystemet betjener behovene for permanent lagring.*

## Behovet for permanent lagring

Det går et skarpt skille mellom de lagringsmediene som mister sitt innhold når strømmen slås av, og de som beholder sitt innhold. Vi har lært at internminnet ikke tar vare på innholdet når maskinen slås av. Vi kaller derfor internminnet for et *flyktig* (eng. volatile) medium. De permanente lagringsmediene har derimot den egenskapen at innholdet forblir inntakt inntil det blir endret[1]. Bruksområdet for permanente lagringsmedier blir dermed å ta vare på alt det vi trenger «i morgen».

**Lagringsmedier** Det permanente lageret er oftest basert på magnetisk lagring. Bitene ligger der som små punkter på en magnetisk overflate som blir lest og skrevet av en elektromagnetisk innretning omtrent som i en båndopptaker. 1- og 0-biter er representert ved retningen av magnetiseringen, og lesing og skriving skjer ved at et elektromagnetisk lese/skrive-hode flyttes over det punktet som representerer den biten som er av interesse.

Andre mediatyper er optomagnetiske, optiske og elektroniske. De optomagnetiske mediene har en overflate av et materiale som endrer polarisasjonen av reflektert lys avhengig av overflatens magnetisering (Kerr-effekten). Avlesingen av en bit skjer derfor med en laser, ikke med en elektromagnetisk pick-up. Skriving til et optomagnetisk medium skjer ved at overflaten varmes opp med en laser (til ca. 200 ºC) og magnetiseres av et magnetisk felt. Minidisc er eksempel på et optomagnetisk medium.

1. Dersom mediet tillater at innholdet skrives mer enn én gang.

Optiske medier blir skrevet og lest med lys. En CD-RW blir preget av en laser som varmer overflaten opp til én av to mulige temperaturer, med det resultat at platen blir punktvis enten matt eller reflekterende. Disse tilstandene kan så avleses med en laveffekt-laser og en fotocelle.

De permanente elektroniske lagringsmediene er oftest minnebrikker (RAM) basert på CMOS-teknologi med et påmontert batteri som sørger for at innholdet bevares uavhengig av ekstern strømforsyning. Konfigurasjonsminnet (BIOS Setup) på en pc er lagret i et slikt minne.

**Tilgjengeliget, hastighet og pris** En viktig egenskap ved det permanente minnet er prisen, fordi vi ofte har behov for å lagre store mengder med data. Billige medier (målt i kr./Byte) er som regel langsomme i bruk, delvis fordi selve utstyret opererer langsomt, men også fordi utstyret gjerne benytter utskiftbare medier (f.eks. CD). Tiden det tar for å få tak i dataene inkluderer selvfølgelig operasjonen med å finne frem mediet og montere det i utstyret.

Der hvor svært mye data skal lagres er vi også interessert i plassbehovet til lageret (målt i Terabytes pr. $m^2$ gulvplass). Holdbarhet av lageret er også interessant. Der hvor vi lagrer data på magnetbånd eller CD-R[2]/CD-RW trenger vi å vite hvor lenge vi kan stole på at dataene er lesbare.

**Online/offline** Vi skiller mellom lagringsmedier som er *online*, dvs. hele tiden tilgjengelige for bruk, og de som er *offline*. Online-medier er gjerne de magnetiske diskene, mens offline-medier er magnetbånd, CD, floppydisker o.l. For å få tak i dataene på et offline medium må mediet (båndet eller platen) monteres i maskinutstyret, og denne oppgaven utføres oftest manuelt av en operatør. Det finnes roboter som kan gjøre denne jobben i store maskinsystemer, og da kalles mediene for «near-line».

Offline-medier er svært billige (mindre enn 10 kr./Gigabyte) og egner seg godt til backup- og arkiveringsformål. De opptar også stadig mindre plass.

**Deling, abstraksjon, styring** Vi har i flere av bokas kapitler valgt å diskutere operativsystemets deler ut fra disse tre perspektivene: deling, abstraksjon og styring. Det perspektivet skal vi ha også i behandlingen av maskinens permanente lager. Vi skal også skille mellom *grensesnitt* og *implementasjon* slik som vi har gjort i tidligere kapitler.

---

2. CD-R er et medium det bare kan skrives til én gang, men regnes som mer holdbart enn CD-RW.

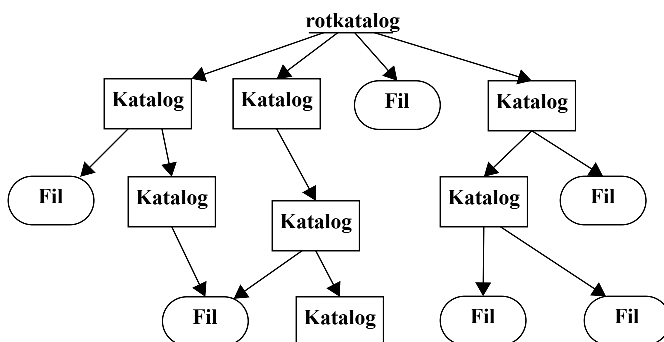
# Hvordan ser det permanente lageret ut?

Det permanente lageret i maskinen er typisk utformet som et *filsystem* hvor dataene lagres i navngitte filer. Navnet danner grunnlag for all behandling av filen og må derfor være unikt. Inne i filen ligger en samling av data i form av bytes, rader eller poster som vi kan lese og skrive etter nærmere bestemte regler.

## Navnesystem

Vi liker at det permanente lageret ligner på et tradisjonelt arkivskap. Inne i skapet finner vi papirene organisert i *mapper*, og mappene ligger gjerne alfabetisk ordnet. Mappene representerer et prosjekt eller en sak, og dokumentene inne i en mappe hører derfor sammen og har felles regler for beskyttelse og bruk.

Dette oppnår vi i de fleste operativsystemene i form av et *hierarkisk navnesystem:* Inne i en mappe kan vi legge filer eller andre mapper, og vi kan derfor organisere en mappestruktur så «dypt ned» som vi har lyst. Navnene må være unike innen én mappe, slik at *stinavnet* (forklares senere) blir unikt for hver fil på dette lageret.

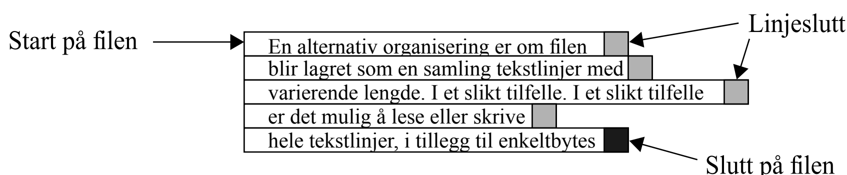



*Figur 8-1: Det permanente lageret formet som et tre gjennom et hierarkisk navnesystem. En katalog ligner en mappe og kan inneholde filer eller andre kataloger*

## Dataorganisering

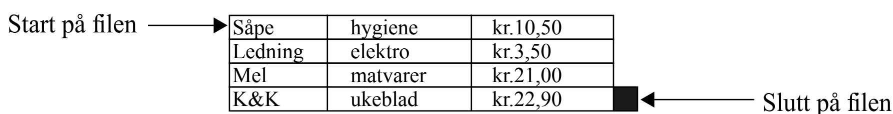
Innholdet i en fil er gjerne organisert som en strøm av bytes, dvs. at det ikke er noen *struktur* på dataene som er lagret Vi kan lese eller skrive et vilkårlig antall bytes under operasjoner på denne filen. Dette er den vanligste formen for organisering og benyttes både i Linux og Windows, som overlater til applikasjonene å skape de nødvendige datastrukturene over denne bytestrømmen.

**Organisering som tekstfil** En alternativ organisering er om filen blir lagret som en samling av tekstlinjer med varierende lengde. I et slikt tilfelle er det mulig å lese eller skrive hele tekstlinjer, i tillegg til enkelt-bytes.




*Figur 8-2: Fil organisert som en tekstfil*

**Organisering som en samling av poster** En annen organisering er å tenke seg at filen består av et antall poster som alle har bestemte «felt». Dette begynner å ligne på en databasetabell, men er enklere enn som så. Den setter av et visst antall bytes til hvert felt slik at det under lesing og skriving er mulig å behandle feltene direkte.




*Figur 8-3: Fil organisert som en samling av poster*

Ingen av disse måtene å organisere dataene på er «universal-løsninger» i den forstand at de egner seg for alle typer anvendelser. Den ustrukturerte varianten som vi beskrev innledningsvis i dette avsnittet må fortsatt være der for å lagre ustrukturerte binærdata som f.eks. filer med programkode.

## Tilgangsmodell

I tillegg til en lagringsmodell vil en fil også være knyttet til en modell for hvordan man kan få *tilgang* til et bestemt dataelement i filen. En lese- eller skriveoperasjon vil foregå der vi har satt en *posisjonspeker.* Tilgangsmodellen vil avgjøre hvordan vi flytter posisjonspekeren.

**Sekvensiell tilgang** Med sekvensiell tilgang til dataene i filen kan dataene bare leses eller skrives i rekkefølge. Posisjonspekeren vil flytte seg fremover i filen for hver lese- eller skriveoperasjon. Det finnes ingen andre måter å flytte posisjonspekeren på. Dersom vi ønsker å lese

et dataelement et stykke ut i filen, må vi flytte posisjonspekeren dit gjennom en serie med leseoperasjoner, ev. flytte pekeren til starten av filen først dersom vi allerede har passert dette punktet.

Sekvensiell tilgang er praktisk der selve lagringsmediet er sekvensielt organisert, slik tilfellet er med et magnetbånd. Til mange slags anvendelser er derimot sekvensiell tilgang en lite effektiv tilgangsmodell.

**Direkte tilgang** Dersom vi kan flytte posisjonspekeren gjennom egne operasjoner som ikke involverer lesing eller skriving av data, snakker vi om direkte tilgang i filen. Dersom vi skal lese eller skrive et bestemt data-element, kan vi flytte posisjonspekeren til starten av dette dataelementet og så starte lese- eller skriveoperasjonen. Etter operasjonen er posisjonspekeren flyttet forbi dette dataelementet, slik at vi uten å flytte pekeren kan starte operasjonen på det påfølgende dataelementet.

Metoden krever at vi kan *beregne* posisjonen til dataelementet, dvs. på hvilket byte-/post-nummer i filen det ligger. Direkte tilgang tilbyr effektiv utføring av anvendelser som gjør mye *oppslag* av data i lageret.

**Indeksert tilgang** Der vi ikke kjenner posisjonen til et dataelement i filen, kan vi referere til dataelementet gjennom en *nøkkel*. Vi kan be posisjonspekeren flytte seg til et dataelement som har en bestemt nøkkel, og deretter starte lese- eller skrive-operasjonen. En slik organisering krever at alle dataelementene er representert med en nøkkel i en *indeks*, og derfor kaller vi denne metoden for *indeksert tilgang*.

Indeksert tilgang er kun aktuell i en lagringsmodell hvor det er en form for struktur i filen, altså ikke filer organisert som en bytestrøm. En samling av poster er velegnet for indeksert tilgang, hvor én av feltene i hver post kan ha oppgaven som «primærnøkkel». Med indeksert tilgang til filen er det oftest mulig å be posisjonspekeren flytte seg til forrige/neste nøkkel i en rekkefølge, f.eks. alfabetisk på nøkkelinnhold.

Heller ikke indeksert tilgang er en «universalmodell», fordi en rekke anvendelser er ute av stand til å utnytte denne måten å bruke en fil på. En vanlig tekstfil som inneholder et dokument har ingen fordeler av indeksert tilgang. Denne tilgangsmodellen er derfor alltid et *supplement* til mer grunnleggende tilgangsmetoder.

# Oppbygningen av en disk

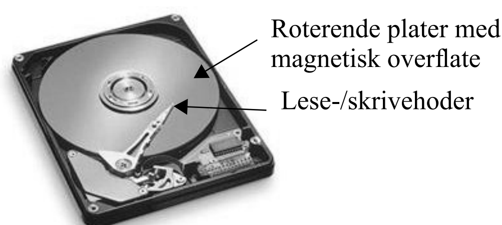
Virkemåten til en magnetisk disk ligner på virkemåten til en båndopptaker. Et signal blir lagret i en magnetisk overflate ved hjelp av en elektromagnet, og avlest med en elektromagentisk pick-up (begge

funksjonene ligger i samme enhet, kalt lese-/skrive-hodet). Det lagrede signalet tar liten plass på den magnetiske overflaten, og signaler med ulik verdi kan ligge tett i tett.

På et magnetbånd står lese/skrive-hodet fast, mens båndet beveger seg forbi i én retning. På en disk beveger hodet seg radialt ut og inn over en roterende skive. Skiven har mange magnetiske spor, og bevegelsen av hodet gjør at man kan velge hvilket spor det skal leses og skrives fra.

En disk har ofte en stabel av slike plater montert på en felles aksel, og et sett av lese-/skrive-hoder som beveger seg som en enhet.

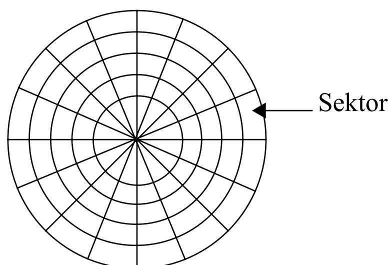
En disk er, i motsetning til en båndopptaker, bare interessert i å lagre 0- og 1-biter. Den trenger derfor bare å skille mellom to typer signaler når den leser og skriver data.




*Figur 8-4: Innmaten i en magnetisk disk*

## Geometri

Den magnetiske overflaten på en disk er satt sammen av mange overflater (to pr. skive) og mange spor på hver skive. I tillegg lager vi en oppdeling av hvert spor inn i et sett av såkalte *sektorer*: Vi kan ikke lagre signaler fritt langs sporet, men må lagre dem inne i sektorene.




*Figur 8-5: Diskoverflate delt inn i spor og sektorer*

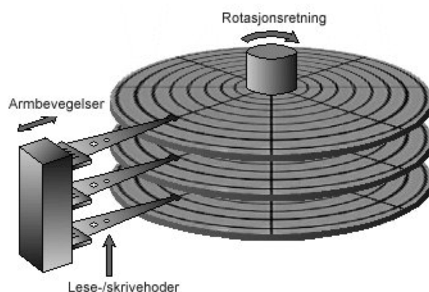
**Sektorer er en logisk inndeling** Sektorer er avmerket med magnetiske signaler langs sporene på skiven, og er derfor ikke mulig å se på diskoverflaten (overflaten er blank som et speil).

En vanlig sektorstørrelse er 512 bytes, og antall sektorer rundt et spor avgjøres av hvor tett signalene kan ligge på overflaten. 63 sektorer pr. spor er et vanlig tall.

En disk-operasjon kan derfor være «les *den* sektoren og legg innholdet i internminnet *der*». Men hvordan angir vi (adresserer) en bestemt sektor?

**Sylinder, hode, sektor** En disk vil alltid ha flere overflater; minst to, fordi en disk alltid utnytter begge sidene av en skive. De sporene som (på hver overflate) befinner seg like langt fra sentrum av skiven, altså de sporene som ligger rett over hverandre, kaller vi for en sylinder. Siden alle lesehodene er montert sammen, kan de til enhver tid bare lese fra spor som ligger i samme sylinder.

En sylinder har flere spor, ett på hver overflate. Hvert av sporene blir lest og skrevet av et separat lese-/skrive-hode. Vi angir derfor hvilken overflate vi vil bruke ved å referere til det hodet som ligger på denne overflaten.




*Figur 8-6: Geometrien i en disk er bygd opp av sylindre, hoder og sektorer*

Sektorene på en disk er den minste adresserbare delen av en disk, og de ligger plassert i noe som kan minne om et tredimensjonalt rom:

- en rekkefølge av *sylindre* (dybde)
- en rekkefølge av *hoder* (høyde)
- en rekkefølge av *sektorer* (bredde)

Adressen til en sektor kan derfor skrives som tre verdier; (a, b, c), hvor hver verdi angir sylinder, hode og sektor.

**Total lagringsplass på disken** Det er lett å regne ut den totale plassen på disken når antall sektorer, hoder og sylindre er oppgitt (det står trykt på disk-kassen) og vi vet at en sektor er 512 bytes.

På disken står det også trykt den totale lagringsplassen, men merk at når diskprodusenten skriver «Megabyte», mener han 1 000 000 bytes, ikke 1024*1024 slik en produsent av internminne mener. Innholdet av 1 MB internminne får altså ikke plass på en disk som rommer 1 MB.

## Ytelse

Det er mange faktorer som påvirker ytelsen på det permanente lageret, men siden mekanikk alltid er langsommere enn elektronikk er ytelsen på selve disken er en viktig faktor. Fra en operasjon starter til den er fullført er det en del ting som skal gjøres:

- Armen som holder lesehodene skal flyttes over den riktige sylinderen. Tiden dette tar kalles «seek time» og er i størrelseordenen 10 ms.
- Skiven skal roteres slik at riktig sektor kommer under lesehodet og blir lest i sin helhet. Tiden dette tar er gjennomsnittlig litt mer enn en halv rotasjon. Vanlig rotasjonshastighet er 7200 omdreininger/minutt. En halv omdreining tar da ca. 4 ms. Dette kalles «latency time».

Lesing av en serie tilfeldige sektorer tar altså ca. 14 ms pr. sektor. Lesing av en serie sektorer på samme sylinder går mye fortere. Fordi lesehodet etter første sektor allerede er på riktig plass, vil påfølgende sektorer leses med kun «latency time» som forsinkelse (4 ms). Lesing av en serie sektorer som ligger etter hverandre på samme spor går enda fortere. Etter den første sektoren vil påfølgende sektorer kunne leses så fort som skiven dreier rundt, uten «latency time»-forsinkelse.

**Viktig:** Ved lesing eller skriving av en serie sektorer har plasseringen av sektorene på disken stor betydning for ytelsen! Sektorer etter hverandre på samme sylinder er raskest å lese.

## Feilkorreksjon

Det kan oppstå bitfeil på disken, dvs. at noe som ble lagret som en 1-bit blir lest en 0-bit (eller omvendt). Dette kan skyldes elektriske forstyrrelser, eller fordi diskoverflaten har svake punkter her og der.

Vi tillater ikke at det oppstår feil i de lagrede dataene på grunn av dette fenomenet, så vi ønsker at det skal kunne korrigeres. I likhet med bruk av en sjekksum for å oppdage bitfeil (slik det gjøres i nettverksprotokoller), vil vi på en disk benytte *feilkorrigerende koder* som i noen utstrekning kan rette oppståtte bitfeil.

Under skriving til en disksektor vil de dataene som skrives inngå i en beregning i diskens styringselektronikk hvor resultat er en slags sjekksum som lagres på disken sammen med dataene. Under lesing vil elektronikken beregne sjekksummen på nytt og varsle om feil dersom den er forskjellig fra den lagrede sjekksummen. Den varianten av sjekksum som brukes setter elektronikken i stand til ikke bare å detektere feilen, men også å korrigere den.

Feilkorrigerende koder er alltid i bruk i en disk, og sannsynligheten for feil som ikke lar seg korrigere er i området 1 bit pr. $10^{15}$ bits lest.

## Bruk av det permanente lageret

Det permanente lageret skaper et bilde av seg selv (abstraksjon) for omverdenen, og de som vil bruke lageret (klientene) må følge et sett av regler (en protokoll) som er fastsatt av operativsystemet. Disse reglene varierer noe fra system til system, men vil i hovedsak ha disse punktene:

**Lagring i filer** All lagring av data skjer i filer, som er navngitte enheter for lagring. Dataene i filen behandles og beskyttes under ett og ut fra de samme reglene.

**Åpning og lukking** Før lesing og skriving av filen må filen åpnes, og i den forbindelse må klienten beskrive om filen skal leses og/eller skrives, og om filen må være utilgjengelig for andre mens den er i bruk. Med andre ord, kun åpne filer kan leses og skrives. Filnavnet er det som refererer til en fil under åpning.

Etter bruk må filen lukkes. Først etter at filen er lukket vil utførte endringer i innholdet være garantert *permanente* og *synlig for omverdenen*.

**Åpne filer har et håndtak** En åpen fil refereres til med et håndtak, dvs. en variabel i brukerprosessens minne som inngår som en parameter i påfølgende lese- og skriveoperasjoner. Håndtaket blir laget under åpning av filen, og det er ikke mulig å «svindle» seg til et håndtak, f.eks. ved å bruke det samme håndtaket som sist filen var åpen.

**Uavhengig av utstyrstype** En viktig egenskap ved et permanent lager er at filene behandles likt uavhengig av hva slags utstyr de er lagret på. Nøyaktig de samme reglene skal følges for filer som ligger lagret på floppy, disk, CD, DVD eller elektronisk lager.

## Java-kode for lesing av en tekstfil

Siden innlesing av tekstdata er en vanlig oppgave, skal vi starte med å vise Java-kode som gjør dette:

```
import java.io.*;
public class ReadFileByLines {
  public static void main(String[] args)
    throws IOException {
    FileReader fr = new FileReader(«abc.txt»);
    BufferedReader br =
      new BufferedReader(fr);
    while (true) {
      // Leser en hel tekstlinje uten <nl>
      String s = br.readLine();
      if (s == null) break; // Slutt på filen
      System.out.print(s);
    }
  }
}
```

*Listing 8-1: Java-klasse som leser tekstfilen «abc.txt» og skriver innholdet til konsollvinduet.*

I listing 8-1 ser vi at det lages et objekt av klassen *FileReader* som representerer håndtaket til filen «abc.txt». Åpningen av filen skjer i forbindelse med at objektet skapes. Ved å kalle metodene på dette objektet kan vi lese data tegn for tegn, men *FileReader*-klassen tilbyr ikke metoder for å lese en hel tekstlinje. Vi kan lage slike metoder selv, men klassen *BufferedReader* er ferdiglaget og tilbyr den metoden vi trenger.

Objektet *br* vil derfor fremstille filen som om den er organisert som tekstlinjer, mens den i «virkeligheten» er organisert som en bytestrøm (i objektet *fr*). Organiseringen som en tekstfil er et abstraksjonslag som ikke ligger i operativsystemet, men i Javas standard klassebibliotek.

Programmet ovenfor baserer seg på en sekvensiell tilgangsmodell (se side 172) hvor posisjonspekeren flytter seg forbi de dataene som blir lest. En serie med leseoperasjoner vil lese dataene i den rekkefølge de ligger lagret i filen. Ved en slik tilgangsmodell er det nødvendig å vite hvor filen slutter. Vi ser i programkoden at denne tilstanden indikeres ved at *readLine()*-metoden returnerer et null-objekt.

**Lesing av en binærfil** Eksemplet i listing 8-1 bruker FileReader-klassen, som bare fungerer ved lesing av tekstfiler. Den endrer bl.a. på tegnet for linjeslutt der det er nødvendig. Ved lesing av binærdata, f.eks. et digitalisert bilde eller en kompilert Java-klasse, da må vi bruke en metode som leverer de «rå» dataene uendret til programmet.

```
import java.io.*;
public class ReadFileByBytes {
  public static void main(String[] args)
    throws IOException {
    byte[] fileData = new byte[10000];
    FileInputStream fis =
      new FileInputStream("bytes.dat");
    // Les nå opptil 10000 bytes inn i fileData
    int bytesRead = fis.read(fileData);
    System.out.println(bytesRead +
      " bytes ble lest");
  }
}
```

*Listing 8-2: Java-klassen `FileInputStream` (eller andre Stream-klasser) skal brukes ved behandling av binærdata.*

I listing 8-2 gir ikke Java-klassene noe ekstra abstraksjonslag, objektet *fis* eksponerer filorganiseringen slik operativsystemet viser det. Om vi vil bruke disse klassene til å behandle tekstdata, må vi vite hvordan det underliggende operativsystemet organiserer teksten (f.eks. hvilke tegn som brukes for linjeslutt).

Java-klassene støtter også en direkte tilgangsmodell gjennom klassen *RandomAccessFile*. Den har metoder for å flytte posisjonspekeren frem og tilbake i filen. En fil som både skal leses og skrives fra et Java-program må være av denne klassen. Listing 8-3 gir et eksempel på bruk.

```
import java.io.*;
public class Random {
  public static void main(String[] args)
      throws IOException {
    RandomAccessFile raf =
      new RandomAccessFile("random.dat","rw");
    for (int i=0;i<1000;i++) {
      raf.writeInt(i);
      raf.writeFloat(i*i);
    }
    // Leser nå filen baklengs, bare for å
    // demonstrere
    for (int i=999;i>=0;i--) {
      int j; float f;
      raf.seek(i*8);// Hver gruppe tar 8 bytes
      j = raf.readInt(); f = raf.readFloat();
      System.out.println("Verdien er: "
        + j + " og " + f);
    }
    raf.close();
  }
}
```

*Listing 8-3: En fil representert ved et objekt av klassen* `RandomAccessFile` *kan både leses og skrives, og posisjonspekeren kan flyttes med* `seek(..)`*-metoden. Dette programmet skriver en tallrekke til filen og leser den så tilbake i omvendt rekkefølge.*

**Tips:** Skriv Java- programmene slik at de vet minst mulig om det underliggende operativsystemet. Da blir programmene lettere å flytte mellom ulike plattformer. Bruk *Reader*-klassene når du behandler tekst.

# Deling av filer

I liket med deling av data i internminnet er det et opplagt behov for deling av data i filer. Og de samme problemstillingene dukker opp også her, nemlig de som er knyttet til kritiske regioner, race conditions, synkronisering og deadlock.

Kritiske regioner kan oppstå i forbindelse med at flere tråder (i ulike prosesser) leser og endrer innholdet i en fil samtidig. En slik kritisk region kan beskyttes ved at en semafor (side 147) settes opp til å

synkronisere adgangen til filen slik at kun én tråd får holde tråden oppe til enhver tid. Men det er vanligere at filsystemet tilbyr en mekanisme som kalles *fil-låsing.*

## Fil-låsing

I forbindelse med deling av filer er det nødvendig med en reservasjonsmekanisme, men vi er sjelden interessert i en mekanisme som tvinger trådene til å vente i blokkert tilstand. Fordi fil-operasjoner ofte involverer handlinger av brukeren, kan det ta lang tid å fullføre en kritisk region (filen må være reservert mens brukeren endrer opplysningene på skjermen), og det er upraktisk å «låse» programmet til en annen bruker i mellomtiden.

Låsemekanismen i filsystemet er enklere enn en semafor, mer i likhet med en «test-and-set»-instruksjon. Om filen er låst, vil forsøk på å åpne filen returnere en feilkode, og kalleren kan selv velge om han vil vente og forsøke igjen senere.

**Låsing av hele filen** Der hvor filen inneholder ett eneste «dataelement», f.eks. et tekstdokument, kan det være nødvendig å låse hele filen under ett. Den som redigerer et dokument må ha full frihet til å gjøre endringer i hele filen, og dette forutsetter at hele filen er reservert.

Låsing av hele filen skjer i forbindelse med at filen åpnes. Frem til filen lukkes igjen vil andre som forsøker å åpne filen få en feilkode i retur.

**Låsing av deler av filen** I andre tilfeller vil filen bestå av mange dataelementer, «poster», «rader» eller hva vi måtte ønske å kalle dem. Disse er uavhengige av hverandre, slik at flere elementer kan endres samtidig, mens hvert element må være reservert hver for seg. Det er derfor bare deler av filen som må være reservert for én prosess, og det i mye kortere tidsrom enn den tiden filen er åpen.

Låsing av deler av filen kan ikke knyttes til åpning og lukking, men må skje gjennom separate kall til filsystemet med parametre som angir hvilken del filen som skal være låst for andre. I den perioden denne delen av filen er låst, vil forsøk fra andre på å lese eller skrive filen returnere en feilkode dersom operasjonen berører den låste delen av filen.

Med låsing av fil-deler kan man oppnå avanserte delingsskjemaer som låsing av «rader» (poster) der flere felt i posten må oppdateres under ett (i en kritisk region).

### Atomiske operasjoner

Et annet delingsskjema går ut på å låse filen slik at den fortsatt er lesbar for andre, men det som blir lest er det gamle innholdet. Filen har altså tilsynelatende det gamle innholdet for alle andre inntil låsen blir opphevet.

Det fenomenet at sammensatte endringer tilsynelatende utføres momentant, er noe som gjør programmeringen av anvenderprogrammet mye enklere. Det blir ikke nødvendig å planlegge alternative handlinger dersom filen skulle være låst av noen andre.

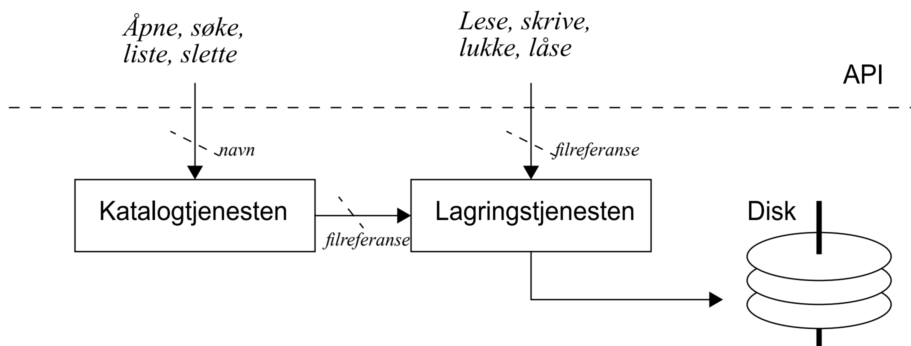
Momentane endringer kalles *atomiske operasjoner*, fordi det ikke er mulig å se noen mellomtilstand i operasjonen. Enten er operasjonen ikke påbegynt, ellers er den fullført.

Atomiske operasjoner finner vi i databaser som har støtte for såkalte *transaksjoner*. En oppdatering av en database i en transaksjon vil være usynlig for andre inntil den er fullført.

## Filsystemets oppbygning

Vi har nå beskrevet hvordan et filsystem kan «se ut», hvordan det skal brukes, og gitt noen regler for hvordan deling av filer skal foregå. Hvordan skal nå filsystemet implementeres slik at det får disse egenskapene?

Vi skal i dette avsnittet tegne en modell av filsystemets oppbygning. Det er altså ikke nødvendigvis slik filsystemet faktisk er programmert. Dette er heller ikke den eneste tenkelige modellen av et filsystem. Vår modell fordeler oppgavene til filsystemet på to komponenter: *Lagrings-tjenesten* og *katalogtjenesten*.




*Figur 8-7: Forbindelsen mellom lagringstjeneste og katalogtjeneste. Se teksten for en nærmere forklaring*

# Lagringstjenesten

Den fysiske lagringen av data i filer utføres av lagringstjenesten. Lagringstjenesten holder derimot *ikke* rede på navnene til filene, men gir hver fil et unikt *filnummer* (ofte kalt inode-nr.). Oppgavene til lagringstjenesten kan oppsummeres slik:

- administrasjon av plassen
- sikre fysisk og logisk integritet (pålitelig lagring)
- utnytte lagringshierarkiet
- tilby lese- og skrivetjenester gjennom et API
- bufring av lese- og skriveoperasjoner

## Administrasjon av plassen

På diskoverflaten ligger det noen magnetiske merker som skiller sektorene fra hverandre. Dessuten ligger det plass for å lagre sjekksummen til hver sektor. Utover dette har ikke en disk (slik den kommer fra fabrikken) noen struktur som støtter lagring av filer.

En av oppgavene til lagringstjenesten er å skape en logisk struktur på disken som et hjelpemiddel for å

- reservere plass til en fil som skal skapes
- utvide denne plassen senere
- frigjøre plassen til denne filen når den slettes, slik at plassen kan brukes til andre filer
- gjenfinne plassen som er reservert for denne filen

**For å få dette til må lagringstjenesten holde styr på to datastrukturer:**

1. En liste over ledige blokker (en blokk er en serie påfølgende sektorer) på disken, kalt *frilisten.*
2. En oversikt som viser hvilke blokker som lagrer innholdet av hver enkelt fil. Denne oversisken legges til selve filtabellen (kalt *inode-tabellen* under Unix/Linux).

Disse to datastrukturene må også lagres på den fysiske disken. Som regel settes det av plass til dette under formateringen av disken.

## Fysisk og logisk integritet

Lagringstjenesten skal sikre at lagrede data aldri går tapt ved et uhell. Hva slags hendelser kan medføre at lagrede data ødelegges eller forsvinner?

- fysiske skader på lagringsmediet (hodekrasj, brann, fysiske støt, magnetfelt)
- uorden eller skader på den logiske lagringsstrukturen (f.eks. inode-tabellen eller frilisten). Slike feil kan oppstå som en følge av feil i programkoden til operativsystemet

For å beskytte mot feil har lagringstjenesten muligheter for å gjenopprette innholdet ved bl.a.

- backup. Alle data skal være regelmessig kopiert til sekundære lagringsmedier i tilfelle fysiske skader
- gjenopprettingsprogrammer som forsøker å bygge opp frilisten og inode-listen ved å søke gjennom alle dataene på disken. Under Linux (og UNIX) heter dette programmet `fchk`. Under Windows 2000 med filsystemet NTFS skjer gjenopprettingen automatisk. Under Windows 98 heter programmet `scandisk`..

## Utnytte lagringshierarkiet

I likhet med minnestyringen kan også lagringstjenesten utnytte lagringshierarkiet slik at mye brukte data lagres høyt opp i hierarkiet, dvs. internminnet eller on-line disker, mens lite brukte data legges på magnetbånd eller utskiftbare disker. Slik kan vi oppnå en god kombinasjon av ytelse og kostnad ved lageret. For å få dette til kan lagringstjenesten benytte to ulike innfallsvinkler:

- Den kan holde et løpende regnskap over hvor ofte og hvor nylig en fil er blitt brukt, og flytte filen i sin helhet til et medium høyere eller lavere i minnehierarkiet basert på dens bruksfrekvens
- Den kan lage en «avbildning» av filen i virtuelt minne, og knytte en separat sidetabell til dette minneområdet. Filen vil da se ut som en samling av minneceller, og minnestyringskoden vil sørge for at de mest brukte delene av filen er i minnet, og at endringene som er gjort på filen blir skrevet tilbake til filen med tiden. Denne teknikken er blitt ganske populær i flere operativsystemer, og innebærer at minnestyringen og filstyringen gjør felles bruk av programkoden for styring av virtuelt minne.

## Tilby et API til omgivelsene

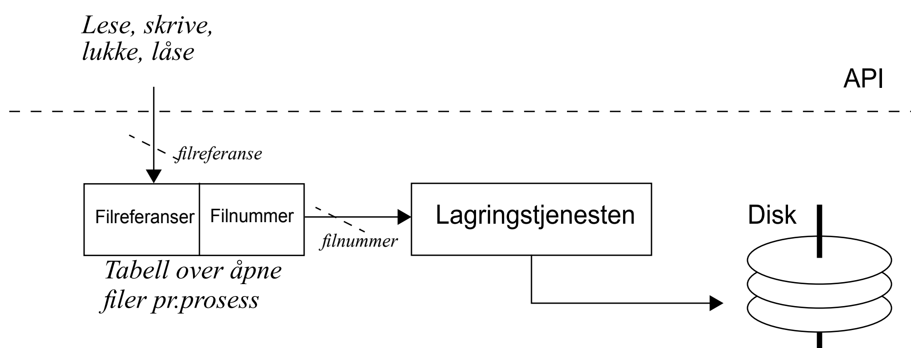
Gjennom et standardbiblliotek (se “klassebibliotek” på side 63) skal brukerprogrammet kunne gjøre fil-operasjoner. Standardbiblioteket (f.eks. `java.io.*`-klassene) er avhengig av at lagringstjenesten tilbyr sine tjenester gjennom et API.

Med API mener vi det «bildet» som operativsystemet lager av lagringstjenesten. Dette er som regel noe annet enn hva et brukerprogram får se (slik vi har beskrevet det i avsnittet “Bruk av det permanente lageret”

på side 177). Mens brukerprogrammet refererer til filer gjennom f.eks instansierte Java-objekter, vil et API bruke mye enklere (og nøytrale) mekanismer.

Lagringstjenestens API vil tilby lesing, skriving og flytting av posisjonspekeren på filer som allerede er åpnet (av katalogtjenesten). Både Windows og Linux tilbyr operasjoner for å lese eller skrive et vilkårlig antall bytes i filen, ikke nødvendigvis hele sektorer.

API-et tillater ikke at en fil referes til ved navn, men forlanger at det brukes en *filreferanse*. En filreferanse er *ikke* det samme som filnummer (side 183). Mens filnummeret er unikt for hele lagringstjenesten, vil filreferansen være unikt innen hver prosess, og tildeles prosessen i forbindelse med åpning av filen. Figur 8-8 er en presisering av figur 8-7 og viser hvordan katalogtjenesten og lagringstjenesten har felles bruk av en tabell som for hver prosess angir hvilke filnumre som hører til hver åpnet fil.

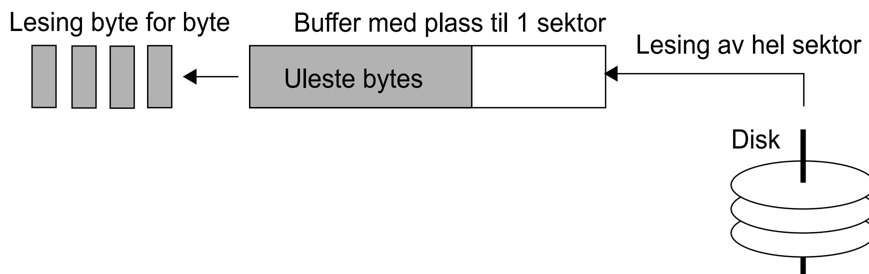



*Figur 8-8: Sammenheng mellom filreferanse og filnummer*

## Bufring av lese- og skriveoperasjoner

En i/o-operasjon mot den fysiske disken kan bare lese og skrive hele sektorer. Lagringstjenestens API tilbyr derimot skriving eller lesing av et vilkårlig antall bytes. Derfor blir det nødvendig å ha en buffer som kan mellomlagre sektorer som ikke er fullstendig lest eller skrevet.

Figur 8-9 viser hvordan lesing av enkeltbytes fra en fil kan foregå. Bufferet fylles med innholdet av en ny sektor fra disk når alle bytene i bufferet er lest.

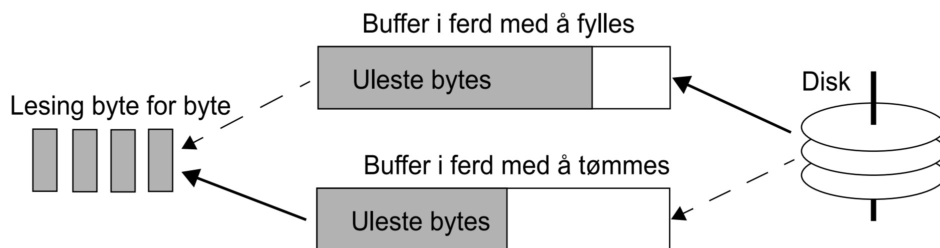



*Figur 8-9: Bruk av enkeltbufring under lesing av en fil byte for byte*

```
public class Buffer {
  byte[] sector = new byte[512];
  int p = 0;
  public byte readByte() {
    byte b = sector[p++];
    if (p == 512) {
      disk.readNextSector(sector);
      p = 0;
    }
    return b;
  }
}
```

*Listing 8-4: Java-kode for enkel bufring under lesing*

**Dobbelt bufring** En variant av skjemaet i figur 8-9 som øker effektiviteten er å benytte to buffere. Dette kalles dobbelt bufring og innebærer at én buffer fylles opp med sektorer fra disk mens den andre leses byte for byte.




*Figur 8-10: Bruk av dobbelt bufring ved lesing av en fil*

Figur 8-10 viser bruk av dobbelt bufring på ett bestemt øyeblikk. Den øverste bufferen er i ferd med å fylles med sektorer fra disk, mens den nederste bufferen blir tømt byte for byte. Når den nederste bufferen er tom, er den øverste (forhåpentligvis) fylt opp, og bufrene kan bytte roller. Lesingen kan dermed pågå uten å måtte vente på diskoperasjoner.

Bruk av bufferteknikker er nyttig også i andre former for dataorganisering, nemlig der hvor filen er organisert som tekstliner eller poster (side 171).

## Katalogtjenesten

Den andre hovedtjenesten i filsystemet er den som setter navn på filene, dvs. tilbyr det *navnesystemet* som ble omtalt på side 171. Katalogtjenesten trer i arbeid i to typer av situasjoner:

**Åpning og lukking av filer** Før et program kan lese eller skrive i en fil må den åpnes. Åpning av filen innebærer disse operasjonene:

- kontroll med om prosessen har de nødvendige rettighetene for å gjøre operasjoner (lese eller skrive) i filen
- bestemme filens filnummer
- reservere nødvendig minneplass for buffere
- skape en filreferanse for senere operasjoner på filen

Ved lukking av filer skal det «ryddes opp» i bufferplass og tabeller, slik at alle ressurser som var reservert i forbindelse med bruk av filen blir frigjort til annet bruk.

**Organisering av filer** Alle operativsystemer har kommandoer og verktøy for å administrere filer (flytte, omdøpe, liste). Dette er operasjoner som ikke berører innholdet av filen, og det er derfor ikke nødvendig å åpne filen.

## Hierarkiske kataloger

En tidlig måte å organisere navnetjenesten på var å organisere alle filene i én eneste katalog. De første versjonene av MS-DOS (1.x) benyttet dette prinsippet, på den tiden da en IBM PC hadde floppydisk som eneste permanente lager. Det begrensede antall filer som får plass på en floppy (den gang rommet en floppy 360 kBytes) gjorde dette til en akseptabel løsning.

Enhver fil må ha sitt unike navn, og en felles katalog for alle filene er uaktuelt der

- mange brukere deler lageret. En bruker vil ikke ønske å ta hensyn til om et ønsket filnavn er «opptatt» av en annen.
- lageret rommer filer som hører til mange prosjekter. Det blir vanskelig å identifisere de riktige filene, og det blir vanskelig å rydde bort filer som ikke er i bruk.

Noen forsøk på å utvide «navnerommet» til katalogen, ved å gjøre eierens brukernavn til en del av filnavnet o.l., var i bruk i en del operativsystemer på 1970-tallet (bl.a. *Sintran* fra Norsk Data).

Hierarkiske kataloger er enerådende i dag, og trenger derfor ingen forklaring. Kataloger som inneholder enten filer eller andre kataloger har vi vist på figur 8-1. Innenfor en katalog må filnavnene være unike, men samme filnavn kan brukes i flere kataloger. Hierarkiske kataloger skaper to nye begreper:

**Stinavn** Kravet til unikt filnavn i hver katalog garanterer oss at det eksisterer en unik identifikasjon av hver fil i form av navnene på katalogene som danner stien fra rotkatalogen ned til (og inkludert) den aktuelle filen. Et stinavn kan ta denne formen:

Windows: ***C:\Anders\NITH\OS-bok\kapittel8.doc***

Linux: ***/home/anders/NITH/OS-bok/kapittel8***

**Gjeldende katalog** Vi kan referere til en fil med et stinavn som ikke har sitt utgangspunkt i rotkatalogen, men en annen katalog i filsystemet. Dette utgangspunktet kaller vi *gjeldende katalog* (Windows: *Current Directory*, Linux: *Working Directory*). Fra gjeldende katalog ramser vi opp veien til den aktuelle filen, med bruk av tegnet «..» om vi trenger å bevege oss oppover i katalogstrukturen.

Filen *C:\Anders\NITH\OS-bok\kapittel8.doc* kan refereres slik:

Gjeldende katalog er *C:\Anders\NITH:* **OS-bok\kapittel8.doc**

Gjeldende katalog er *C:\Anders\Priv:* **..\NITH\OS-bok\kapittel8.doc**

Et stinavn som skal forstås relativt til en gjeldende katalog kalles *relativt stinavn*.

## Metadata

Det er behov for å lagre opplysninger i filsystemet som verken er deler av filens datainnhold eller som det er mulig å lagre i katalogtjenesten. Utgangspunktet for diskusjonen om *metadata* er hvordan filens *eksistens* må kunne kontrolleres.

### Eksistenskontroll (livssykluskontroll)

I figur 8-1 viser vi også et annet forhold. En av filene er representert i mer enn én katalog, gjerne med forskjellige filnavn. Det finnes dermed flere stier som fører frem til samme fil. Dette er uproblematisk, bortsett fra ett spørsmål:

*Når skal filen opphøre å eksistere?*

Dette er et trivielt spørsmål i tilfeller der hver fil bare har ett stinavn. Da skal filen opphøre å eksistere (fjernes fra lagringstjenesten) idet den slettes i katalogtjenesten. Men der hvor samme fil har flere «bruksområder» i form av flere stinavn, må vi forutsette at når filen slettes gjennom ett stinavn, skal den fortsatt eksistere gjennom de andre.

Dette krever at filsystemet holder rede på hvor mange stinavn som fører frem til hver fil. Denne verdien kaller vi en *bruksteller*. Når en fil slettes (fra katalogtjenesten) skal brukstelleren reduseres, og om brukstelleren blir 0 skal filen opphøre å eksistere (fjernes av lagringstjenesten).

Brukstelleren kan ikke lagres i noen katalog. Den er en verdi som hører til den fysiske filen slik den ligger i lagringstjenesten, og det er lagringstjenesten som må lagre den sammen med andre opplysninger om filen (f.eks. fysisk plassering i diskblokker).

Slike data som ikke er en del av filens innhold, men som heller ikke kan lagres i katalogtjenesten (på grunn av forholdet med flere stinavn), kaller vi *metadata*. Under Linux finner vi metadata lagret i den såkalte inode-tabellen

### Inode-tabellen

En *inode* er en datastruktur som beskriver de egenskapene til en fil som ikke kan lagres i katalogtjenesten. Husk at katalogtjenesten kan referere til samme fil under forskjellige sti- og filnavn. Opplysninger som skal være entydige for filen blir derfor lagt i inoden. Tabellen under viser hvilke opplysninger som lagres i en inode:

| Felt | Beskrivelse |
|---|---|
| Modus | Angivelse av hvordan filen skal være beskyttet for skriving og lesing fra eieren og andre brukere |
| UID | UID (brukernummeret) til eieren av filen |
| Group ID | GID (gruppenummeret) til eieren av filen |
| Lengde i bytes | Antall bytes i filen |
| Lengde i blokker | Antall blokker (à 512 bytes) reservert for denne filen |
| Sist endret | Dato og klokkeslett da filen sist ble åpnet for skriving |
| Sist brukt | Dato og klokkeslett da filen sist ble åpnet |
| Sist endret inode | Dato og klokkeslett da filens inode sist ble endret |
| Bruksteller | Antall stier som fører frem til denne filen |
| Blokkpekere | Referanser til de blokkene på disken hvor filen er lagret |

Inodene ligger samlet i en tabell på disken. Denne tabellen opprettes når disken klargjøres for bruk (med `mkfs`-kommandoen). Bruken av inoder innebærer at om man endrer opplysningene i den (gjennom endring av beskyttelse eller eierskap), vil dette være gjeldende og synlig for alle filnavn som refererer til denne filen. Bruk av inode løser også behovet for eksistens-kontroll med filen, av samme årsak.

## Ytelse / caching

Permanent lager baserer seg på teknologi som tildels er svært gammeldags. Den magnetiske disken bruker prinsippene fra båndopptakeren, som ble patentert i 1898. Tiden det tar å få lest eller skrevet data på eksterne permanente media er minst 100 000 ganger lengre enn hva det tar å gjøre operasjoner på internminnet. Derfor er interessen stor for løsninger som kan forbedre ytelsen ved bruk av permanent lager.

**Løsningene ligger i lagringshierarkiet** Forbedring av ytelsen på disker o.l. er en kontinuerlig prosess hos fabrikantene av slikt utstyr, men vi vil under alle omstendigheter kunne øke ytelsen mye mer ved å bruke de samme idéene om lagringshierarkiet som vi diskuterte i forbindelse med minnestyringen (“Lagringshierarkiet” på side 105).

**Viktig:** Mye brukte data mellomlagres i internminnet!

I tillegg til å måle bruksfrekvensen av data, kan vi i forbindelse med filstyringen også legge til grunn at dataene ofte blir lest sekvensielt, og at en «spekulativ» bufring som vist i figur 8-10 også kan øke ytelsen.

Vi har behandlet caching og annen bruk av lagringshierarkiet ved flere anledninger i boka, så vi vil her kun oppsummere de teknikkene som er aktuelle i forbindelse med filstyringen:

- En lokal cache på disk-kontrolleren (maskinvaren som kontrollerer disken) kan øke hastigheten på gjentatte operasjoner (samme sektor leses flere ganger innen et tidsrom). Cachen ligger i minnebrikker som *ikke* er en del av maskinens ordinære internminne.
- En samling av bufre i lagringstjenesten som holder de oftest etterspurte sektorer i internminnet.
- En buffer for hver åpen fil som holder deler av filen i brukerens minneområde. Dette kan gjøres av kjøremiljøet eller standardbiblioteket såvel som av operativsystemet.

Merk at alle disse tre teknikkene også kan lese deler av filen «spekulativt» i påvente av at dataene blir forespurt på et senere tidspunkt.

**Kjøreplan for diskoperasjoner** En teknikk for å få diskene til å operere raskere er å redusere veilengden til lese- og skrivehodene. Vi husker fra avsnittet “Ytelse” på side 176 at det er flytting av lese-/skrivehodene som tar lengst tid i disken, og vi ønsker derfor å redusere den samlede veilengden som disse hodene må vandre.

Dersom det ligger tre forespurte diskoperasjoner i kø, vil det være rettferdig å betjene dem i den rekkefølge de ankom køen (first come – first served). La oss tenke oss at forespørslene retter seg mot disse stedene på disken:

| Kø av diskforespørsler: |
|---|
| Forespørsel nr. 1: (hode=2, sektor=15, sylinder=3) |
| Forespørsel nr. 2: (hode=1, sektor=10, sylinder=20003) |
| Forespørsel nr. 3: (hode=4, sektor=14, sylinder=3) |

Om vi behandler dem i vanlig rekkefølge ser vi at armen med lese-/skrivehodene må flytte seg over 40 000 sylindre (spor) sammenlagt. Dersom vi behandler forespørslene i rekkefølge (1, 3, 2) får vi derimot en samlet veilengde på 20 000 sylindre.

Operativsystemet kan altså lage en «kjøreplan» for diskforespørsler som forsøker å øke ytelsen gjennom å redusere veilengden for diskarmene. Dette er ikke en oppgave for verken lagrings- eller katalogtjenesten, men vil ligge i den programvaren som kontrollerer maskinvarekretsene for disken, nemlig *device driveren* (se "Utstyrshåndtering" på side 51).

## Sammendrag

- Det permanente lageret inkluderer alle *ikke-flyktige* lagringsmedier.
- De viktigste egenskapene til et lagringsmedium er dets *hastighet* og *pris.*
- Bruken av et filsystem avhenger av *navnesystemet*, *dataorganiseringen* og *tilgangsmodellen.*
- En magnetisk disk har en geometri som gjør aksesstiden avhengig av *rekkefølgen* av operasjonene.
- Når filer skal deles, bør filsystemet også tilby *deling* og *låsing* av filer.
- Filsystemets oppgaver kan deles i *lagringstjenesten* og *katalogtjenesten.*
- Ytelsen til filsystemet kan økes gjennom bruk av *caching* og bruk av *virtuelt minne* for å utnytte lagringshierarkiet. *Kjøreplan for diskoperasjoner* kan også brukes.

| **Sentrale begreper i dette kapitlet:** | |
|---|---|
| Lagringstjeneste | Katalogtjeneste |
| Filbeskyttelse/fillåsing | Dataorganisering/tilgangsmodell |
| Atomiske operasjoner | Bufring/caching |
| Diskgeometri | Inode |

# Teorioppgaver

**Gå sammen i grupper og løs disse oppgavene:**

1 Beregn korteste vei for flytting av lese-/skrivehodene for disse seriene av diskoperasjoner:

- Sylindre nr. 3, 9, 5, 7, 12
- Sylindre nr. 2309, 134, 2433, 2567
- Sylindre nr. 2, 9, 2, 2, 2

Foreslå en algoritme for å beregne beste rekkefølge av diskoperasjoner i kø.

2 Sektorene på en fysisk disk kan organiseres i en logisk rekkefølge slik at de danner en kontinuerlig nummerering. Gitt at disken har 450 sylindre, 6 hoder og 17 sektorer pr. spor kan det logiske nummeret til en sektor beregnes på minst tre forskjellige måter (sektor, sylinder og hode starter alle nummereringen på 0):

- logisk nummer = sylinder*17 + hode*(450+17) + sektor
- logisk nummer = sylinder*(6+17) + hode*17 + sektor
- logisk nummer = sylinder + hode*(450+17) + sektor*450

Gitt at alle sektorene på disken blir lest i denne logiske rekkefølgen, hvilken av disse nummereringsalternativene gir den korteste veilengden for lese-/skrivehodene?

3 Finn frem gjeldende priser for standard disker med forskjellig kapasitet, og tegn en grafisk fremstilling som viser pris (i kr./MB) som en funksjon av lagringskapasiteten på disken.

4 Beregne tiden det tar å ta backup fra en stor pc-disk (finn kapasiteten fra en annonse) til et passende backup-medium (f.eks. magnetbånd i kassetter). Flaskehalsen er trolig hastigheten på backup-mediet. Ta hensyn til tiden det tar å bytte medium underveis om nødvendig.

# Øvingsoppgaver

Forslag til øvingsoppgaver ligger på bokas nettsted.

**Etter fullførte øvinger bør du beherske:**

1 Hvilke kommandoer under Linux som endrer dataene i inoden, og hvilke som kun endrer katalogdataene.

2. Forskjellen på *links/symbolic links* i Linux, og *snarveier* i Windows.
3. Sette riktig beskyttelse på filer i Linux og Windows.

# Stikkordregister


## A

abstraksjon 7
abstrakte datatyper 55
Address Latch Enable 31
administrasjon 9
adressebuss 30
adressedekoding 32
adresseringsmekanismer 96
API 58, 184
Apple Lisa 12
asynkrone tjeneroperasjoner 132
atomiske operasjoner 182
automatiske variabler 54, 65
avbrudd 39
  programvare- 60
  -rutine 40, 52
  -vektor 40, 60

## B

BIOS 58
blocked-tilstand 77
blokk 183
blokk i/o 53
Blue Gene 39
bootstrap 57
branch prediction 27
bruksteller 189
bufret dataflyt 134
bufring 185
buss-syklus 31
busy waiting 124
bytecode 68

## C

cache 34, 198
  bruk i DNS 204
  write-through 36
CD-RW 170
CPU 24

## D

Data Transfer Acknowledge 34
databasetjener 9, 200
databuss 29
datavirus 15
device driver 52
direkte tilgang 173
distribuerte operativsystemer 11
distribusjon 10, 195
DMA 42
  burst mode 43
  cycle stealing 43
dobbelt bufring 186
Domain Name Services 202
dynamiske variabler 66

## E

eksistenskontroll 189

## F

feilkorreksjon 176
feilkorrigerende koder 177
feiltoleranse 199
filhåndtering 50
fil-låsing 181
filnummer 183
filreferanse 185
filsystem 8, 171
filtjener 199
fjernadministrasjon 10, 212
fjernutføring av kommandoer 10
fleksibel tildeling 102
flertråds tjener 159
fragmentering 104
friliste 183
funksjonsgrensesnitt 8, 61

## G

garbage collection 67, 87
GEM 13
gjeldende katalog 188
gjensidig utelukking 126
GNU 17
grensesnitt 48, 58

## H

harddisk 7
heap 67
hierarkisk navnesystem 171
hierarkiske kataloger 188, 202
hode 175
høynivåspråk 63

## I

i/o-celler 28
idempotent 206
ikke-preemptiv kjøreplan 79
indeksregister 97
inode 183
inode-tabellen 190
instruksjoner 24
instruksjonsdekoder 26
instruksjonspeker 26, 97
integrerte kretser 24
Intel Pentium 116
internavbrudd 110

## J

Java Virtual Machine 68
JVM 68

## K

katalogtjenesten 187
Kerr-effekten 169
kjøremiljø 65, 93
kjøreplan 79
  ikke-preemptiv 79
  preemptiv 79
kjøreplan for diskoperasjoner 191
klassebibliotek 64
klient/tjener 196
kompilator 64
konfigurasjon 56
kontekst svitsj 78
kontrollbuss 31
kritisk region 128
kritiske regioner 200

## L

lagringshierarkiet 105
latency time 176
lese-/skrive-hodet 174
Linux 15
livssykluskontroll 189
loader 65
lokasjonstransparent 211
lokasjonsuavhengig 60

## M

meldingsgrensesnitt 61
meldingshåndtering 53
meldingsorientert synkronisering 213
mellomvare 5, 9
Message-Oriented Middleware (MOM) 209
metadata 189
mikrokjerne 57
Minidisc 169
minneceller 28
minnehåndtering 50
minnestyring 93
moduler 53
modulære programmeringsspråk 54
monitorer 150
multiprogrammering 7
multiprosessor 36, 197, 215
  anvendelsesområder 38
  løst koplet 37
  tett koplet 37
mutex 126

## N

navnesystem 171, 187
navnetjeneste 211
near-line 170
Network File System (NFS) 212

## O

objektmonitor 10
objektorientering 55
objektprogram 64

offset 109
Open Source 18
optisk lagring 169
optomagnetisk lagring 169
organisering som 172
OS/2 14

P

page fault 110
paged segments 114
paging 50, 107
peer-to-peer 201
permanent lagring 169
pipelining 27, 140
posisjonspeker 172
POSIX 16, 48
preemptiv kjøreplan 79
prioritet 79
produsent/konsument 134
programvareavbrudd 60, 99
prosess 73
    deskriptorer 76
    gjenskaping 75
    lagring 75
    -tabell 76
prosesshåndtering 49
prosess-kontekst 80
publish-and-subscribe 210

R

ready-tilstand 77
redundans 199
relativt stinavn 189
Remote Method Invocation (RMI) 208
repeterbare operasjoner 206
replikert tjener 198
ressursdugnad 197
RMIregistry 211
running-tilstand 77

S

Samba 212
sanntids operativsystemer 12
seek time 176
segmenter
    sidedelte 114
segmentering 112
segmentregistre 97
sektor 175
sekvensiell tilgang 172
semaforer 147
sidebrudd 110
sidedelte segmenter 114
sidetabell 109
single system image 208
Sintran 188
skall 68
    symbolbasert 69
    tegnbasert 69
skeleton 208
SMB-protokollen 212
SNMP 212
sockets 216
SPI 51, 58
stakk 67
statiske variabler 65
stinavn 171, 188
stiv tildeling 101
strukturerte programmeringsspråk 54
stub 208
styring 8
subrutiner 53
supervisor mode 60
sylinder 175

T

tegn i/o 53
tekstfil 172
tildeling
    fleksibel 102
    stiv 101
tilgangsmodell 172
tilstandsløse tjenere 207
timer 41
tjenerklynger 200
transaksjonsmonitor 10
tråd 73
    tilstand 76

U

UNIX 15
user mode 60
utføringsstatus 77

utstyrshåndtering 51

V

variabler
- automatiske 65
- dynamiske 66
- statiske 65

virtuelt minne 93, 104, 184
virus 15

X

Xerox PARC 12

Kapittel 9

# Distribusjon

*Behovet for å samordne programutføringen mellom maskiner koplet sammen i et nettverk tilsier at operativsystemet bør legge til rette for slik samordning. I dette kapitlet skal vi diskutere nytteverdien av distribuert utføring av programmer, hvilke spesielle problemer som operativsystemet må ta hensyn til, og hvordan noen av disse problemene kan løses. Vi skal også studere hvordan Java legger til rette for samordnet programutføring i et distribuert miljø.*

## Hvorfor distribusjon?

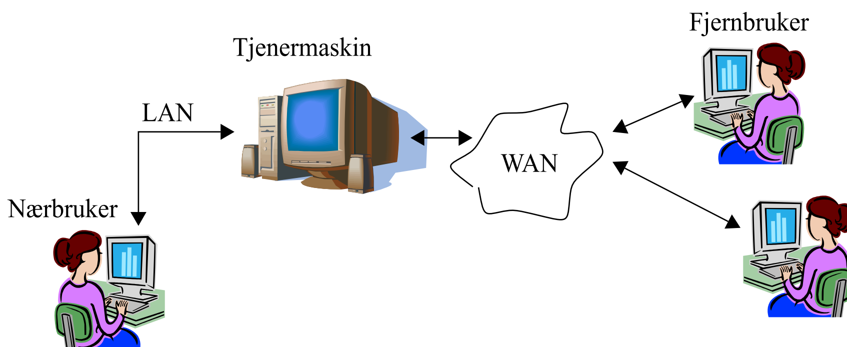
Med ordet *distribusjon* sikter vi til de sammenhengene hvor flere maskiner kommuniserer over et nettverk for å løse en bestemt oppgave. Distribuerte anvendelser er en dagligdags ting i våre IT-omgivelser, f.eks. e-post og World Wide Web.

Hvorfor har vi nytte av å lage distribuerte anvendelser? Vi kan peke på fire grunner i stikkordsform: *datadeling*, *ressursdugnad*, *nettverks-økonomi* og *feiltoleranse*.

### Datadeling

Der hvor en datamaskin behandler data som skal deles mellom brukere, trenger alle brukerne en eller annen form for tilkopling til denne datamaskinen, dvs. en nettverksforbindelse. Over denne forbindelsen skal dataene fraktes frem og tilbake etter som brukerne leser og endrer dataene.

Figur 9-1 viser en distribuert anvendelse hvor brukere leser og skriver innholdet i det samme permanente lageret gjennom nettverksforbindelser. Den samme anvendelsen i en ikke-distribuert konfigurasjon ville ha krevd at alle brukerne hadde hver sin kopi av dataene. Lokale kopier av dataene har noen vesentlige ulemper:




*Figur 9-1: En enkel klient/tjener-distribusjon*

- Endringer som gjøres av én bruker blir ikke synlige for de andre. Flere brukere kan komme til å gjøre uforenlige endringer på de samme dataene.
- Det totale lagringsvolumet i systemet øker.

Det sier seg selv at World Wide Web er avhengig av distribusjon. Litt av nytten av å lagre en nettside sentralt i en en web-tjener er at den kan ajourholdes fortløpende av eieren. Volumet av dataene i World Wide Web er dessuten altfor stort til å lagres lokalt på en privat maskin.

**Klient/tjener** En distribuert konfigurasjon som baserer seg på et samarbeidsforhold hvor den ene parten «bestiller» og den andre parten «leverer», kalles *klient/tjener*. Anvendelser av datadeling som vist ovenfor er oftest utformet som klient/tjener, fordi det gir god kontroll med gyldigheten og tilgjengeligheten av de lagrede dataene. Brukere av en anvendelse er *alltid klienter* i en slik konfigurasjon.

**Utstyrsdeling** På samme måte som ved deling av data, finner vi også klient/tjener-anvendelser som tilbyr deling av utstyr. Vi kan f.eks. tenke oss at en kostbar fargeskriver ikke er en fornuftig investering dersom den kun skal betjene én bruker, men at den gjennom en klient/tjener-anvendelse kan benyttes av flere. I slike tilfeller er det tjeneren som kontrollerer skriveren og «leverer» utskrifter, mens brukerne er klienter som «bestiller» utskrifter.

## Ressursdugnad

En annen type distribuert anvendelse er når én enkel maskin ikke har den ønskede behandlingskapasitet (minne, CPU-kapasitet eller permanent lager), men at det er mulig å forene ressursene i flere maskiner slik at de *i sum* har det. Eksempler på anvendelser som egner seg for ressursdugnad er:

**Søking og sortering i store datamengder** Der hvor vi vil søke etter en bestemt dataforekomst i et stort lager, kan oppgaven deles mellom mange maskiner som får oppgaven med å søke i hver sin delmengde av data. Deloppgavene er ikke avhengig av hverandre, og samordningsbehovet mellom maskinene er lite.

Tilsvarende kan mange maskiner sortere hver sin del av en stor datamengde. Resultatet av disse delsorteringene kan til slutt *flettes* sammen for å fullføre sorteringen av hele datamengden.

**Arkiv av enorme datamengder** På Internett finner vi eksempler på anvendelser som er distribuert for å spre lagringsbehovet over mange datamaskiner (se bokas nettsted for henvisninger). Typisk er det filer som inneholder musikk og video som blir lagret, og spredningen på mange maskiner gjør det vanskelig å håndheve opphavsrettighetene på innholdet i filene.

**Dataflyt-orienterte anvendelser** I anvendelser hvor dataene behandles i flere «steg», og hvor resultatet fra ett steg er inndata til det neste, kan vi distribuere anvendelsen etter en «dataflyt»-modell. Da vil hver maskin behandle dataene i ett av stegene, og motta data fra maskinen som behandler forrige steg, og levere dataene (gjennom nettverket) til maskinen som behandler neste steg.

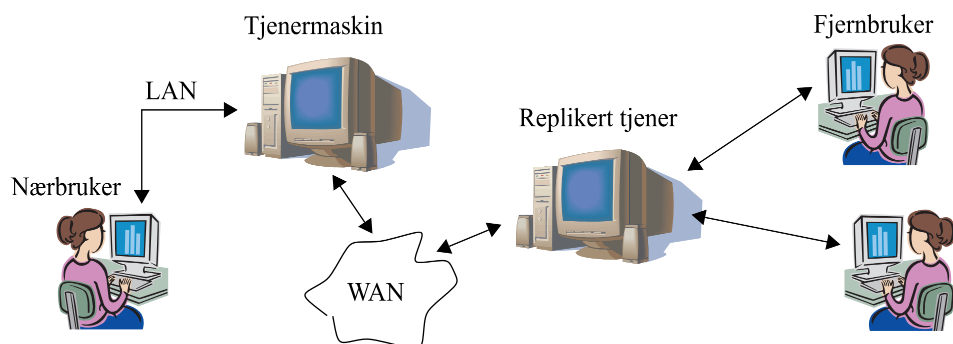
Se avsnittet “Når trenger vi multiprosessorer?” på side 38. Der diskuterer vi hvilke anvendelser som egner seg for multiprosessorer, og de viste eksemplene er mye av de samme som her. Bruksområdene til en *løst koplet multiprosessor* vil ha sterke likhetstrekk med de distribuerte anvendelsene vi her omtaler. Hovedforskjellen er at distribuerte anvendelser kan involvere maskiner med stor geografisk avstand mellom seg. Begrepet »løst koplet multiprosessor» forstås oftest som maskiner i nærheten av hverandre.

## Nettverksøkonomi

Gjennom distribusjon kan vi også utnytte nettverksressursene bedre. Ved å:

**Lagre data nær de brukerne som har størst overføringsbehov** Et vanlig bruksmønster av lagrede data er slik at det er en liten del av brukerne som skaper mestparten av trafikken ut og inn av lageret. Lagringstjeneren kan plasseres slik at disse brukerne har kort avstand, f.eks. gjennom et hurtig lokalnett. Da unngår vi å belaste kostbare WAN-ressurser med denne trafikken. Dette er den vanlige situasjonen ved mange nettsteder, hvor redaksjonen sitter nær web-tjeneren fordi de skaper mer trafikk mot den enn de øvrige brukerne.

**Unngå å overføre de samme dataene flere ganger** Dersom mange av tjenerens brukere henter de samme dataene og overfører dem langs samme transportvei (rute i nettverket), da utnytter vi nettverkskapasiteten dårlig. I slike tilfeller kan det være gunstig å plassere en lokal kopi av dataene nærmere disse brukerne slik at de henter sine data fra den lokale kopien fremfor fra tjeneren selv.




*Figur 9-2: Klient/tjener-konfigurasjon med replikert tjener*

Dersom den lokale kopien kun brukes til å støtte leseoperasjoner, dvs. operasjoner som ikke endrer innholdet av dataene i tjeneren, kaller vi den en *cache*. Dersom den lokale kopien kan endres (og de endrede dataene siden blir lagt tilbake i tjeneren) kalles vi tjeneren *replikert* eller et *replika*.

**Lokal behandling og presentasjon** Volumet av de overførte data kan reduseres ved at dataene blir behandlet lokalt for presentasjon og kontroll:

- En grafisk fremstilling trenger ikke å overføres som et bilde. Den kan overføres som en serie av talldata, som klienten tolker og viser frem som et grafisk bilde.

- Data som overføres fra en klient til en tjener skal være gyldige. Tjeneren vil nekte å godta «31. februar» som en dato, men da er det allerede brukt nettverksressurser på å overføre denne datoen til tjeneren. Klientprogrammet bør selv kontrollere så langt det lar seg gjøre at kun gyldige data blir sendt til tjeneren.

### Feiltoleranse

Distribuerte anvendelser kan også innebære at flere maskiner gjør akkurat samme jobb, og behandler de samme dataene. Dersom én maskin krasjer eller produserer feil resultat, kan de andre maskinene sørge for å opprettholde normal drift.

Distribusjon kan også gjøre anvendelsen vår mindre sensitiv for nettverksfeil. Dersom veien frem til én tjener er utilgjengelig, kan vi forsøke en annen tjener som gjør samme jobben, og hvor veien dit er i normal virksomhet.

Bruk av *redundans* for å oppnå *feiltoleranse* er et fagfelt som bare delvis benytter distribusjon som virkemiddel, og vi vil derfor ikke behandle det separat. I en del sammenhenger er det derimot mulig å oppnå feiltoleranse som et «biprodukt» av distribuerte anvendelser, og det skal vi se eksempler på senere i kapitlet.

## Eksempel på distribuerte anvendelser

I vår daglige omgang med datamaskiner finner vi mange eksempler på bruk av distribuerte anvendelser.

### Fjernlagring

Deler av det tilgjengelige filsystemet kan være plassert på en disk som er kontrollert av en annen maskin. En slik maskin kaller vi ofte for *filtjeneren*, og bruk av lager på en filtjener har mange fordeler:

- lageret er felles for mange klienter, og egner seg for dokumentarkiv og for distribusjon av programmer
- det er lettere å ta sikkerhetskopi av et felles lager
- plassen på disken utnyttes bedre, og det er lettere å utvide lagringsplassen etter behov

Filtjeneren kan bare gjøre fil-operasjoner, dvs. åpne, lukke, lese og skrive filer. Det er altså ikke mulig å be en filtjener gjøre operasjoner på de lagrede dataene, f.eks. søke eller oppdatere.

## Programmerte tjenere

Mer avanserte operasjoner på de lagrede dataene krever at tjeneren kan utføre programmerte operasjoner på bestilling fra klientene. Dette innebærer bedre nettverksøkonomi, fordi dataene i tjeneren i noen grad kan behandles uten å bli transportert mellom klienten og tjeneren. Det innebærer også at det er lettere for tjeneren å reservere data under kritiske regioner (se avsnittet “Reservasjon” på side 126) som måtte oppstå under denne behandlingen.

Et vanlig tjenerprogram er en *databasetjener*. Denne administrerer en database med tabeller, nøkler, datatyper og lagrede prosedyrer. Klienten får ikke lov til å bruke fil-operasjoner for å behandle dataene i databasen, men må bruke et spørrespråk (SQL) for å manipulere dataene. Spørreoperasjonene utføres i tjeneren og bidrar til redusert nettverkstrafikk. Kun de dataene som skal vises eller endres må transporteres over nettverket, mens søking, indeksering o.l. foregår i tjeneren.

Andre tjenerprogrammer som fungerer på denne måten er en web-tjener og en ftp-tjener.

## Fjernutføring av kommandoer

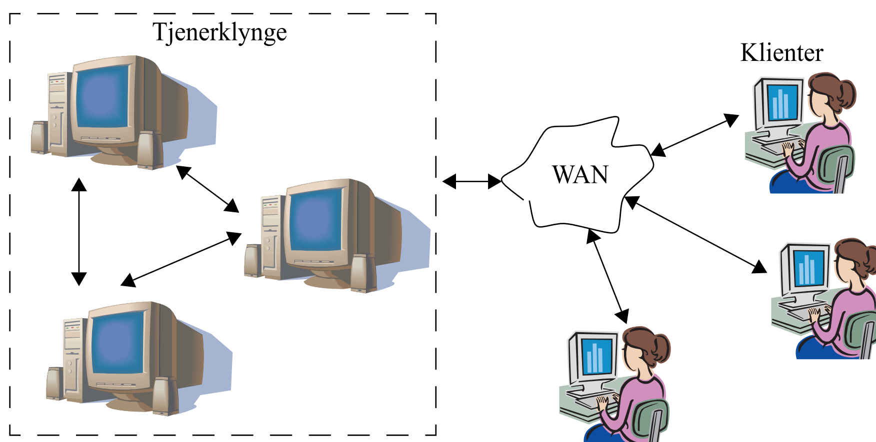
En friere form av tjeneren er en som kan motta kommandoer om å starte et hvilket som helst program, ta i mot inndata over nettverket (eller fra en fil), og til slutt presentere utdata tilbake til klienten.

For å gi kommandoer til tjeneren trenges et skall (se avsnittet om “Skallet” på side 68) som kan føre en dialog med klienten og sende kommandoer og inndata over nettverket. Slike skall kan være både tegnbaserte (telnet) og grafiske (X, PCAnywhere).

## Tjenerklynger

Flere tjenere kan arbeide skulder ved skulder i den forstand at de kan dele på arbeidsmengden eller ta over for hverandre ved feil.

I en slik konfigurasjon får vi et trafikkbilde som vist på figur 9-3. Vi har i tillegg til den vanlige klient-tjener-trafikken også behov for dataoverføring mellom tjenerne. Tjener-tjener-trafikken sørger for at tjenerne er mest mulig enig i systemets tilstand (innloggede brukere, låste data, innhold i databaser).




*Figur 9-3: Tjenerklynge*

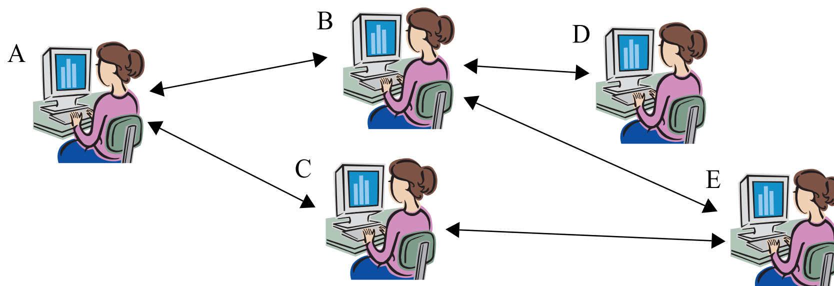
Det er ikke meningen at klientene skal se enkeltmaskinene i en tjenerklynge, og heller ikke vite hvilken tjenermaskin de faktisk samhandler med. Derfor vises ikke klient-tjener-trafikken som en pil helt frem til maskinene, men stopper ved «gjerdet». Bak dette gjerdet blir trafikken dirigert etter kriterier som klienten ikke vet noe om.

## Peer-to-peer

En alternativ organisering av distribuerte anvendelser er å la hver maskin være både en klient og en tjener: Samtidig som den tar hånd om klientoppgaver overfor den «lokale» brukeren (den som sitter ved konsollet), tar den hånd om tjeneroppgaver bestilt av andre maskiner.

**Symmetrisk og selvkonfigurerende** Peer-to-peer skaper et «symmetrisk» forhold mellom maskinene som adskiller seg fra klient/tjener-konfigurasjonen, og fremstår som en «dugnad» av mange små enkeltmaskiner som bidrar til anvendelsen, men bare i en tidsperiode. Når maskinen skrus av eller avslutter programmet, vil resten av anvendelsen måte klare seg uten denne maskinen. Av denne grunn vil samarbeidsmekanismene (anvenderprotokollene) for et peer-to-peer-nettverk inkludere en «oppdagelsesfase», hvor et nytt medlem i nettverket kan presentere seg slik at de andre maskinene kan nyttiggjøre seg dennes ressurser og tjenester. *Et peer-to-peer-nettverk må være selvkonfigurerende.*

Figur 9-4 viser en skisse av et peer-to-peer-nettverk. Pilene illustrerer de logiske forbindelsene (transportforbindelsene) som finnes mellom maskinene, ikke de fysiske forbindelsene.




*Figur 9-4: Et peer-to-peer-nettverk*

**Indirekte forbindelser** Figur 9-4 viser at ikke er forbindelser mellom alle mulige par av maskiner, og at det derfor er nødvendig med bruk av *transittmaskiner* som videresender data på vegne av andre. Veien mellom A og E går via enten B eller C. Om maskinen B skrus av blir maskin D isolert fra nettverket om ikke den klarer å sette opp forbindelser til en av de øvrige maskinene.

Designet av en peer-to-peer-anvendelse er komplisert, men bærer i seg en mulighet til å romme et enormt antall parter. På Internett finner vi eksempler (Gnutella, Freenet, Kazaa m.m.) på peer-to-peer-anvendelser som kan romme titusener av maskiner.

**Anvendelsesormåder** Peer-to-peer-anvendelser har sin styrke i de sammenhenger hvor det ikke er strenge krav til komplette og ajourførte data, og der hvor sikkerhetsbehovet ikke er høyt. Samarbeidsteknologi og «gruppevare» er områder hvor noen peer-to-peer-anvendelser har sett dagens lys. En billett-database, et formelt dokumentarkiv eller e-post er derimot anvendelser uegnet for peer-to-peer-konfigurasjon.

## Hierarkiske kataloger

Et eksempel på et vellykket anvendelse for å distribuere en stor katalog finner vi i Internetts navnetjeneste *Domain Name Services* (DNS). Katalogen som oversetter alle mulige Internett-navn (f.eks. *www.vg.no*) til IP-adresser har altfor mye trafikk rettet mot seg i form av forespørsler til at den kan ligge på ett enkelt sted. Den blir også så mye endret at det er umulig å administrere den i sentralisert form.

**Nærhet mellom klient og tjener** Organiseringen av DNS baserer seg derfor på den formodning at de fleste navneforespørslene oppstår i nærheten av den ressursen det spørres etter: De aller fleste forespørsler

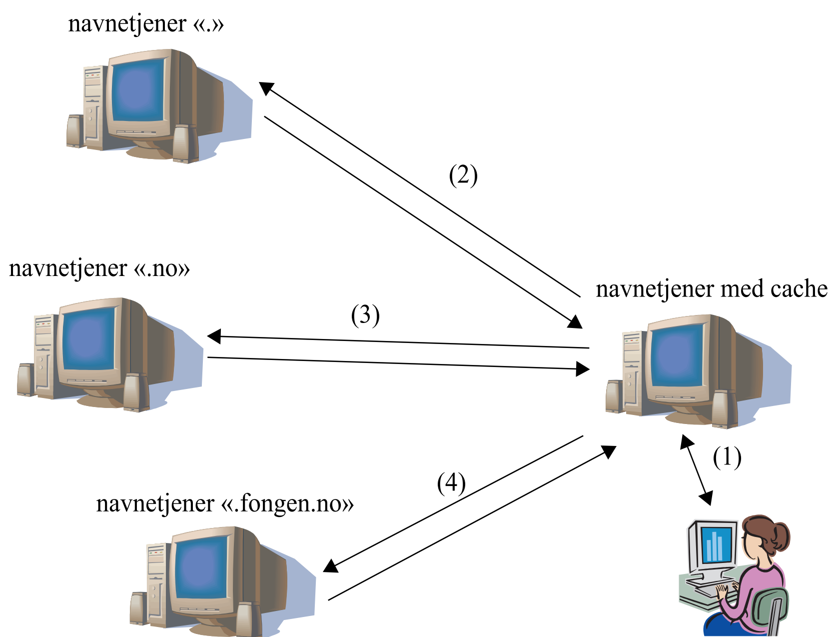
etter Internettnavn som slutter på «.no» oppstår i Norge, derfor er det fornuftig å legge denne delen av katalogen i Norge. Likedan legges katalogen til alle navn som slutter på «nith.no» i NITHs nettverk, fordi de fleste forespørslene kommer herfra.

**Desentralisert administrasjon** DNS forutsetter dessuten at det eksisterer et myndighetsforhold som følger hierarkiet i navnesystemet. Administratoren av «.no»-domenet delegerer myndigheten til å bestemme over «.nith.no»-grenen til NITH, og over «.fongen.no» til forfatteren av denne boka. Dette gjør at «.no»-administratoren må passe på at bare Fongen får endre opplysninger om «.fongen.no», noe som administratoren av «.com», «.se» eller topp-nivået ikke trenger å bry seg om. Den hierarkiske organiseringen av katalogen gjør det altså mulig å håndtere rettighetene til dataene som ligger lagret der.

**Hierarkisk oppslag, med caching** Figur 9-5 viser rekkefølgen i oppslaget av navnet «www.fongen.no»:

**Et DNS-oppslag følger disse stegene:**

1. Klienten forespør den lokale navnetjeneren om den kjenner IP-adressen til navnet «www.fongen.no». Muligens finnes denne i tjenerens cache. I så fall besvares spørsmålet her.
2. Den lokale navnetjeneren er konfigurert med IP-adressene til «topp-tjenerne». Disse vet om hvem som er navnetjenere for «hoved-områdene», og spørsmålet som sendes dit er «Hvem er navnetjener for .no-området?». Svaret gir IP-adressen til denne tjeneren.
3. På samme måte sendes dette spørsmålet til «.no»-navnetjeneren: «Hvem er navnetjener for .fongen.no-området?». Svaret gir IP-adressen til den riktige navnetjeneren.
4. Spørsmålet som sendes til «.fongen.no»-navnetjeneren er: «Kan jeg få IP-adressen til www.fongen.no, takk?». Svaret er IP-adressen til denne maskinen, og dette svaret returneres til klienten som svar på forespørselen i (1).




*Figur 9-5: DNS-oppslag av navnet «www.fongen.no»*

I alle stegene vist ovenfor vil svaret caches i navnetjeneren. Det er derfor bare en sjelden gang i mellom at den lokale navnetjeneren må spørre topp-navnetjeneren om hvem som betjener «.com»- og «.no»-området, fordi svaret ofte vil være lagret fra en tidligere forespørsel.

Skalerbarheten i DNS beror i stor utstrekning på bruk av caching. Alle data i DNS-katalogene er oppført med en «levetid», som angir hvor lenge de kan lagres i cachen før de må hentes på nytt. Denne levetiden gir oss en mulighet til å endre data i katalogen på en kontrollert måte.

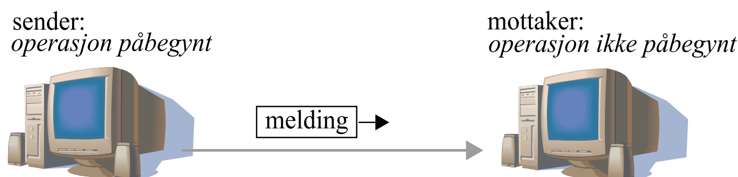
## Spesielle problemer i distribuerte applikasjoner

Vi vil nå diskutere noen problemer som kan oppstå i distribuerte anvendelser, men som ikke (eller sjelden) opptrer i sentraliserte anvendelser. Problemene oppstår bl.a. som følge av uunngåelige forsinkelser og feil i det nettverket som skal transportere dataene mellom partene i den distribuerte anvendelsen.

## Ingen felles tilstand

Etter at en klient har sendt en forespørsel til en tjener, men den ennå ikke er mottatt av tjeneren, da er de to ikke enige om tilstanden i systemet. Klienten mener at operasjonen er påbegynt, mens tjeneren er av en annen oppfatning.

Den samme situasjonen vil opptre når operasjonen er fullført, og resultatet er underveis fra tjener til klient, og i alle andre former for meldingsutveksling i nettverket. Det er altså ikke mulig å betrakte systemets tilstand som en universell tilstand. Fra perspektivet til én av partene er systemet i en eller annen tilstand, men ikke nødvendigvis den samme som hva en annen part ser.




*Figur 9-6: Mens en melding er underveis er sender og mottaker uenige om systemets tilstand*

Fordi partene følger regler (en protokoll) for hvordan operasjoner endrer systemets tilstand, vil dette ikke ha stor betydning *unntatt i forbindelse med krasj*.

## Uavhengig krasj og feil

I en sentralisert anvendelse er programutføringen organisert slik at hele anvendelsen stopper opp dersom det skjer en alvorlig feil. Dvs. er det alltid *hele* anvendelsen som krasjer, ikke bare deler av den. Dette innebærer også at det er hele anvendelsen som starter opp under ett og at alle delfunksjoner starter opp i et veldefinert «startpunkt».

Annerledes blir det i en distribuert anvendelse, hvor maskiner som deltar i anvendelsen kan krasje og starte opp igjen, mens øvrige maskiner er i normal drift. Maskinen som starter opp en delfunksjon må derfor forholde seg til at den «fødes» til en verden med innloggede brukere, åpnede loggfiler, klienter som venter på svar fra den krasjede tjeneren m.m.

En tjener som krasjer midt under utføringen av en operasjon kan vise seg ved at den mottar en forespørsel fra en klient, men ikke sender noe svar tilbake. Når krasjet den?

- før operasjonen ble utført?
- etter at operasjonen ble fullført, men før meldingen tilbake til klienten ble sendt?

Klienten har ingen mulighet for å avgjøre dette. Dersom klienten velger å bestille operasjonen på nytt (kanskje fra en annen tjener) risikerer den å dublisere resultatet (to uttak fra en konto istedenfor ett). Alternativet er å ikke gjøre noenting, med den risikoen at operasjonen aldri blir utført.

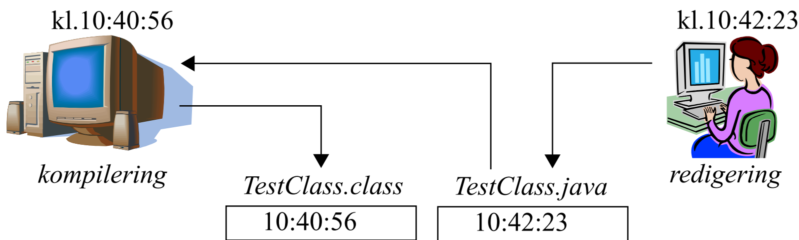
Løsningen på dette dilemmaet er å lage den distribuerte anvendelsen med operasjoner som kan kjøres flere ganger uten at resultatet endres. Såkalte *repeterbare* operasjoner (også kalt *idempotente*) har denne egenskapen. De kan kjøres mange ganger, med det samme resultatet som om den bare ble kjørt én gang. Repeterbare operasjoner er noe som må planlegges fra bunnen av i anvendelsen, fordi det bl.a. påvirker hvordan datalagringen skal foregå.

## Ingen felles klokke

Det er ikke mulig for partene i en distribuert anvendelse å ha eksakt samme oppfatning av hva klokka er. De innebygde klokkekretsene i maskinvaren vil, som alle andre klokker, ha en viss avdrift, og vil vise ulik tid etter hvert. Vi kan velge ut én maskin som «urmester» som kan sende meldinger til andre maskiner med riktig tid, men disse meldingene vil ha en forsinkelse som er umulig å forutse nøyaktig, derfor kan vi heller ikke beregne nøyaktig tid ut fra disse meldingene. Vi kan lage mekanismer som reduserer avviket mellom klokkene, men situasjonen forblir den samme i praksis, nemlig:

*Kronologiske hendelser kan synes å ha skjedd i motsatt rekkefølge.*

Vi skal vise et eksempel på et slikt hendelsesforløp i form av en «kompilatortjener», som holder oversikt over filene på en filtjener og kompilerer java-filene automatisk dersom den tilhørende class-filen er eldre. Konfigurasjonen er vist i figur 9-7.




*Figur 9-7: Eksempel på feilsituasjon som følge av unøyaktige klokker. Fordi de to partene bruker sine egne klokker som tidsstempler på filene, later det til at class-filen ikke er ajour*

Det er vanlig at filtjenere ikke bruker sin egen klokke til å tidsstemple filer, men lar klientene bestemme denne verdien. Dersom klokken til kompilatortjeneren sakner bak klokken til maskinen som redigerer filen (til høyre på figuren), vil den resulterende class-filen få et eldre tidsstempel enn java-filen. Feilaktig vil det derfor se ut som om java-filen trenger å kompileres, og den vil kompileres gang på gang inntil tidsforskjellen er «tatt igjen».

## Tilstandsløse tjenere

Et mye brukt prinsipp i designet av distribuerte anvendelser er å gjøre tjenerne *tilstandsløse*. Dette innebærer at tjeneren returnerer til en «startposisjon» hver gang den har fullført en operasjon.

Om en tilstandsløs tjener krasjer og må restartes, kommer den tilbake til den samme tilstanden som hva den har mellom vanlige operasjoner. Derfor trenger den ikke å re-etablere noen tilstand mellom seg og sine klienter (f.eks. i form av åpne filer eller innloggede brukere). Krasj i en tilstandsorientert tjener vil typisk kreve at alle klientprogrammene avsluttes og starter opp igjen.

En web-tjener er tilstandsløs, og betrakter operasjoner som uavhengige og isolert fra hverandre[1]. Om en web-tjener krasjer og må restartes, merker ikke en web-klient dette med mindre den forsøker å gjøre operasjoner mot tjeneren i det tidsrommet den er ute av drift. Annerledes er det med Netware filtjenere, som krever at alle brukerne logger seg av og på igjen om tjeneren må restartes.

1. Dette er ikke helt sant. Microsoft Internet Information Server og en web-tjener som benytter *Java servlets* vil ha en midlertidig tilstand i et *sesjonsobjekt*.

## Operativsystemets rolle

Både operativsystemet, mellomvaren (side 9), klassebibliotekene og kjøremiljøet (side 63) bør støtte opp under utviklingen og utføringen av distribuerte anvendelser. Støtten kan være ferdige tjenerprogrammer, klassebibliotek eller utviklingsverktøy. Vi vil i det følgende dele opp mulige oppgaver i den forbindelse etter de tre stikkordene vi har brukt flere ganger tidligere i boka: *abstraksjon*, *styring* og *deling*.

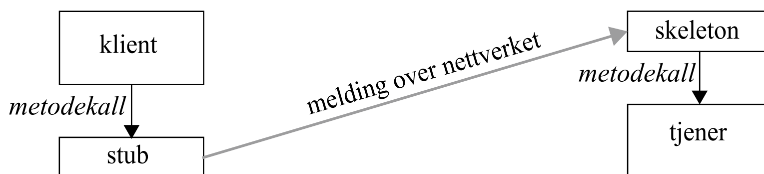
### Abstraksjon – «single system image»

Et gjennomgående trekk ved mellomvare som støtter distribuerte anvendelser er at den presenterer et «single system image». I dette begrepet ligger det at et distribuert miljø skal fremstå som en enkel maskin, og at det (i den grad dette er fornuftig) skal være omtrent den samme jobben å programmere et distribuert som et sentralisert miljø.

**Remote Method Invocation (RMI)** En del av «single system image» kan være at tjeneroperasjoner bestilt av en klient ser ut som et vanlig metodekall[2]. Det innebærer at klientkoden gjør metodekall på et objekt som på forhånd er satt opp til å «representere» tjeneren.

Tjeneren på sin side har et objekt som implementerer de tjenestene den tilbyr, og metodekallene til dette objektet kommer fra et objekt (i tjeneren) som representerer klienten.

Figur 9-8 viser dette forholdet hvor objektet som representerer tjeneren kalles en *stub*, og objekter som representerer klienten kalles en *skeleton*.




*Figur 9-8: En RMI-konfigurasjon med stub- og skeleton-objekter*

2. RMI er et begrep knyttet til Java. Prinsippene i denne diskusjonen er derimot relevant også for annen type teknologi som f.eks. CORBA og «Remote Procedure Calls» (Sun)

**Stub og skeleton lages automatisk** Det er nødvendig at stub-objektet har det samme «utseendet» som tjener-objekter, dvs. har metoder med den samme signaturen (metodenavn og parametertyper). Stub- og skeleton-objektene har forøvrig disse oppgavene:

- etablere nettverksforbindelse seg imellom
- formidle metodekall og parameterverdier over nettverket
- returnere feilsituasjoner i tjeneren tilbake til klienten

Fordi de ikke har oppgaver i forbindelse med selve informasjonsbehandlingen, kun administrative oppgaver, er de velegnet for å bli laget automatisk. Hva trenger vi å vite for å lage stub og skeleton?

- metodedeklarasjonene til tjener-objektet (navn og parametertyper)

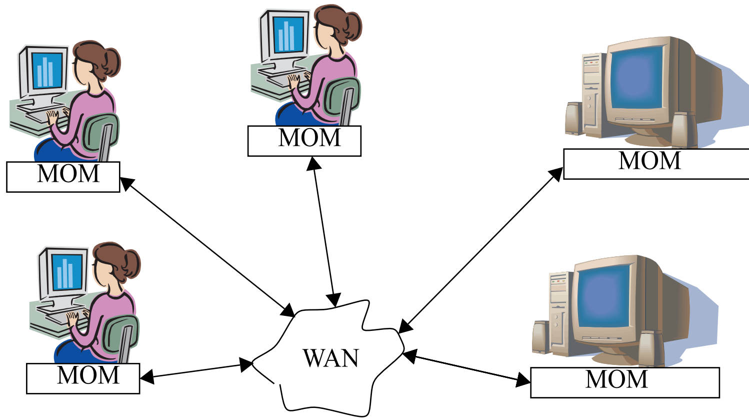
Vi bruker et hjelpeprogram kalt RMI-kompilator for å lage stub og skeleton til et tjenerobjekt. Alt RMI-kompilatoren trenger som inndata er klassen til tjenerobjektet.

> **Merk:** Selv om et RMI-kall ser ut som et vanlig metodekall, bør du huske på at et RMI-kall bruker betydelig mer ressurser. Unngå overdreven og ufornuftig bruk av RMI, ellers får du dårlig ytelse i anvendelsen.

Vi får av plasshensyn ikke anledning til å vise eksempler på bruk av RMI. En «middels» avansert lærebok i Java vil derimot diskutere RMI-programmering utførlig.

**MOM – meldingsbasert grensesnitt** I avsnittet “Funksjonsgrensesnitt eller meldingsgrensesnitt?” på side 61 viste vi hvordan tjenester fra operativsystemet kan bestilles gjennom meldinger. På samme måte kan en distribuert anvendelse fremstilles som et «postverk», hvor vi aldri setter oss i «direkte» kontakt med en tjener gjennom metodekall, men baserer oss på å sende meldinger mellom partene som inneholder kommandoer, parametre og resultatverdier. Noen fordeler og ulemper med denne teknikken er diskutert i dette avsnittet, og vi skal her peke på noen forhold som angår distribuerte anvendelser spesielt.

MOM er en forkortelse for «Message-Oriented Middleware», og betegner programvare som støtter et meldingsgrensesnitt mellom parter som er koplet sammen i nettverk. Det betyr at et MOM må samarbeide med operativsystemet om transport, mellomlagring og adressering. Et MOM kan bidra til et distribuert design som er mye mer «avslappet» i forhold til krasj i maskiner og feil eller forsinkelser i nettverket.




*Figur 9-9: Slik inngår et MOM i en distribuert meldingsflyt*

Som figur 9-9 viser, er det MOM som har kontakt med nettverket, ikke anvenderprogrammet. Bruken av et MOM fremfor å sende meldinger gjennom nettverket på egen hånd (gjennom vanlige TCP-forbindelser) kan dessuten by på flere alternative former for meldingsutveksling:

- man kan sende til mer enn én mottaker
- istedenfor å sende til en identifisert mottaker, kan vi sende til alle dem som «abonnerer» på et bestemt «emne». Dette prinsippet kalles *publish-and-subscribe*
- mottakeren av meldingen trenger ikke å være i drift ved sending. MOM kan lagre slike meldinger på permanent lager, slik at meldinger vil blir levert en eller gang i fremtiden

## Styring

Blant operativsystemets styringsoppgaver har vi tidligere nevnt typiske «husholdningsoppgaver», dvs. håndtering av utstyr og ressursadministrasjon. Slik er det også ved behandling av distribusjon. Operativsystemet må styre:

**Nettverks- og transportprotokoller** Dersom datamaskinen er koplet til et lokalnett gjennom en Ethernet-adapter, må denne adapteren styres med en device driver på lik linje med annet utstyr, slik vi har beskrevet det i avsnittet "Utstyrshåndtering" på side 51. Ved bruk av en Ethernet-adapter vil reglene i Ethernet-protokollen (innramming, adressering, kollisjonshåndtering) bli fulgt av maskinvaren i nettverks-adapteret.

Linjeprotokollene som benyttes over telefonlinjer (POTS[3] eller ISDN). Protokollene på høyere lag (IP, TCP, UDP osv.) må derimot støttes av programvare i operativsystemet.

Nettprotokollen IP og transportprotokollene TCP og UDP er støttet av Linux og Windows. Andre nettverksprotokoller som er støtter er:

**Navnetjeneste/lokalisering** Vi ønsker ikke å huske IP-adresser til andre maskiner i nettverket av flere grunner:

- IP-adresser skifter ofte, f.eks. når maskinen flyttes fra ett nettverk til et annet (f.eks. når den skifter Internett-leverandør). Navnet på maskinen kan derimot holdes uendret over lang tid.
- IP-adresser er vanskelig å huske. Det er lettere å huske symbolske navn som er av mer «beskrivende» karakter (f.eks. *www.aftenposten.no*).

En tjeneste som oversetter symbolske navn til nettverksadresser kalles en *navnetjeneste*. Fordi en nettverksadresse også forteller noe om *hvor* denne maskinen befinner seg, kan vi si at navnetjenesten *lokaliserer* maskinen. Et distribuert system som ved hjelp av en navnetjeneste skjuler lokasjonen til en maskin, kaller vi for *lokasjonstransparent*.

Den vanligste navnetjenesten heter *Domain Name Services* (DNS) og er som beskrevet tidligere i kapitlet, bygd opp som en hierarkisk katalog. Alle operativsystemer som har innebygd støtte for TCP/IP-protokollene har også innebygd støtte for DNS.

En navnetjeneste finner vi også i forbindelse med RMI, hvor klienten må lokalisere tjenerobjektet. Navnetjenesten til Java RMI kalles *RMIregistry* og holder rede på tjenerobjekter på sin egen maskin. Vi bruker altså ikke RMIregistry til å lokalisere tjeneren i nettverket, men til å lokalisere tjenerobjektene i tjenermaskinen. RMIregistry vil dessuten tilby klienten en kopi av stub-klassen, slik at denne ikke trenger å være installert på forhånd.

**Filoverføring og andre applikasjonsprotokoller** En del anvendelser er så vanlig i bruk at de gjerne støttes av operativsystemleverandøren i form av verktøyprogrammer. Slike anvendelser kan være:

- filoverføring
- hente og vise nettsider
- fjernbetjening og fjernutføring av kommandoer

3. POTS: Gammeldags telefon (Plain Old Telephone Service)

Slike anvendelser finner vi ikke støttet gjennom operativsystemets API, men som nevnt gjennom separate verktøyprogrammer. Klassebiblioteket til programmeringgspråkene kan også tilby slik støtte; Javas klassebibliotek (Java API) tilbyr henting av data med HTTP- og FTP-protokoll gjennom klassen `java.net.URL`.

**Fjernadministrasjon** Maskiner koplet til nettverk kan administreres og overvåkes under ett gjennom å ha små tjenerprosesser under utføring i hver maskin. Disse tjenerprosessene svarer på spørsmål om diskens fyllingsgrad, ledig minne, CPU-belastning m.m., og om maskinens konfigurasjon (installert program- og maskinvare). Tjenerprogrammet kan også i noen grad tilby fjernbetjening for å automatisere konfigurasjonsoppgaver (f.eks. installere en ny versjon av et program) og selv varsle om oppståtte situasjoner (disk full, virusangrep, krasjede prosesser).

Programvare for fjernadministrasjon følger ofte protokollen SNMP (Simple Network Management Protocol). I noen grad finnes støtte for denne protokollen i Linux og Windows, men for å ta i bruk fjernadministrasjon kreves i praksis ekstra innkjøpt programvare.

## Deling

Deling av ressurser i nettverket er en viktig fordel knyttet til distribuerte anvendelser. Derfor har både Windows og Linux i sin standardutførelse støtte for de vanligste delingsoppgavene i form av tjenerprogrammer.

- Deling av disker. Mellom maskiner som bruker Windows er det vanlig å dele filsystem (se avsnittet “Datadeling” på side 195) basert på den såkalte SMB-protokollen. Linux kan ta del i slik deling (både som klient og tjener) med programmet *Samba*, som leveres sammen med Linux-distribusjonene.
  Innen UNIX-miljøet er det vanlig å dele filer gjennom NFS-protokollen (Network File System). Linux kan uten videre ta del i slik deling (både som klient og tjener), men Windows krever tilleggsprogramvare for å gjøre dette.
- Deling av utskrift. En maskin som kjører Windows kan dele sin skriver med andre maskiner gjennom SMB-protokollen, og andre maskiner (både med Linux og Windows) kan sende sine utskrifter til denne tjeneren. En Linux-maskin kan tilby skrivertjenester basert på programmet *Samba*.

**Deling av prosessorkapasitet** Vi har flere ganger trukket frem fordelene med å la flere maskiner gå sammen i en dugnad for å øke den totale behandlingskapasiteten. Ved bruk av Linux og Windows er dette kun mulig gjennom egne tjenerprogrammer skrevet med dette formålet

for øye. De generelle mulighetene for fjernutføring av kommandoer og programmer kan brukes, men vil ikke gi den nødvendige kontroll i forbindelse med start, stopp og synkronisering av partene i anvendelsen.

## Meldingsorientert synkronisering

I kapitlene om synkronisering (kap. 6 og 7) hadde vi en grundig diskusjon om behovet for synkronisering, og hvordan vi kunne oppnå de nødvendige tjenestene for de ulike synkroniseringsbehovene.

Gå tilbake til figur 6-9 på side 141. Legg merke til det synkroniseringsbehovet som er merket (3). Dette dreier seg om synkronisering mellom tråder i ulike maskiner. Her er det ingen mulighet for å benytte betingelses-objekter i operativsystemet, fordi det heller ikke i operativsystemet finnes minneceller som er felles for trådene.

I et operativsystem har vi mulighet for å utveksle meldinger, dvs. sende meldinger til én bestemt mottaker eller til en gruppe mottakere (ev. alle mottakere på et lokalnett). En tråd kan også motta meldinger, eller sette sin utføringsstatus til *blocked* inntil en melding ankommer.

Hvor i operativsystemet er det vi finner slike tjenester? I de primitive nett-tjenestene for sending og mottak av data med TCP- og UDP-protokollen. Dersom MOM mellomvare er installert kan også den tilby lignende tjenester.

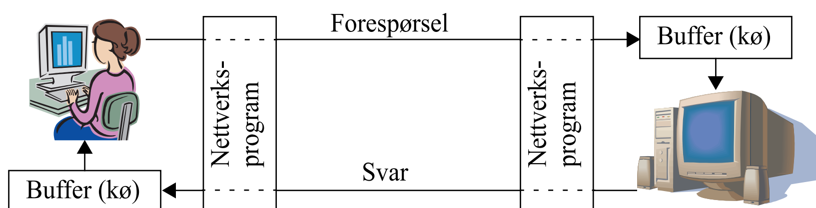
**Varsle/vente = Sende/motta** Mellom to parter som trenger å synkronisere seg med hverandre kan sending og mottak av «tomme» meldinger erstatte varsle/vente-operasjonene omtalt i kap.6, fordi virkemåten er tilsvarende. En tråd som venter på en melding vil gå i *blocked* utføringsmodus inntil en annen tråd sender en slik melding. Dersom meldingen allerede er sendt, vil den være mellomlagret hos mottakeren slik at mottakeren vil få meldingen og fortsette uten opphold.

Virkemåten av slike meldinger sendt mellom to parter vil altså være lik den virkemåten som er vist i figur 6-2 på side 123. Det betyr at vi kan tenke oss et reservasjonsbehov (se avsnittet "Reservasjon" på side 126) løst ved at to parter sender hverandre «toalettnøkler» i form av små meldinger. Innholdet av meldingene spiller da ingen rolle, kun *eksistensen av dem*.

I praksis vil vi synkronisere partene i en distribuert anvendelse med meldinger brukt på en annen måte enn å kopiere vente/varsle-virkemåten. De er to grunner til dette:

- Et reservasjonsbehov mellom flere enn to parter løses ikke med meldinger adressert til enkeltmottakere. Heller ikke meldinger med flere mottakere (multicast) kan løse dette behovet
- Klient/tjener-synkronisering har bl.a. til hensikt å synkronisere over tilgjengeligheten av data (parametre og returverdier) i delte minneceller, også synkronisering over bufret dataflyt. Men, når det er ikke er noen delte minneceller mellom trådene, er heller ikke synkroniseringsbehovet det samme.

**Meldingsgrensesnitt** Vi er tilbake i diskusjonen fra avsnittet om Message Oriented Middleware (side 209): En synkronisert to-veis meldingsflyt som inneholder data (parametre og returverdier) kan brukes til å tilby tjenester som også «kapsler inn» eventuelle reservasjonsbehov. Kritiske regioner kan beskyttes av koden i tjeneren, hvor semaforer (eller lignende) sørger for at flere tråder ikke samtidig utfører kode i kritiske regioner og skaper «race conditions».




*Figur 9-10: Klient-tjener-kommunikasjon via et meldingsgrensesnitt*

Figur 9-10 viser hvordan en lokal buffer bidrar til synkroniseringen av trådene: Nettverksprogramvaren opptrer som «produsent» til en ringbuffer og plasserer data der når de mottas fra nettverket. Tråden som leser meldingene leser fra ringbufferen og får sin utføringsstatus satt til *blocked* om det ikke er noe data i bufferen.

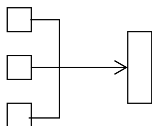
Dersom man har et separat reservasjonsbehov som ikke lar seg kapsle inn i tjeneren på denne måten, kan vi nokså enkelt lage en «toalett-nøkkel-tjener» som tar i mot forespørsler om en toalettnøkkel, stiller dem opp i en kø og deler ut toalettnøkler i en bestemt rekkefølge (f.eks. first-come-first-served) etter hvert som de kommer i retur. Bare den som har en toalettnøkkel kan benytte en bestemt ressurs, og må levere nøkkelen tilbake etter bruk.

## Andre synkroniseringsvarianter

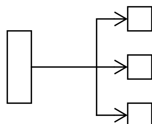
I avsnittet "Andre synkroniseringsvarianter" på side 157 identifiserte vi behovet for alternative modeller for synkronisering:

- mange-til-én-synkronisering (mange klienter - én tjener)
- én-til-mange-synkronisering (kringkasting)
- én-til-en-av-flere-synkronisering (room service)

Vi trenger å finne ut om disse modellene kan brukes også ved meldingsbasert synkronisering.

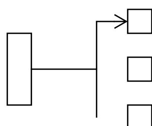


**Mange-til-én** Denne varianten er uten videre tilgjengelig ved meldingsbasert synkronisering: Mange kan gi en tjener beskjed om å utføre en gitt oppgave, og tjeneren vil starte utføringen av denne oppgaven når den får beskjed om det fra én av klientene. Dette er omtrent hva som foregår i en web-tjener: Tråden som utfører tjenerprogrammet er uvirksom (*blocked*) inntil den får en bestilling fra én av klientprogrammene (nettleserne).



**Én-til-mange** Denne varianten krever at en melding kan sendes til mer enn én mottaker: Ved å ta i bruk «multicast»-adresserte UDP-segmenter kan vi oppnå dette, men slik UDP-trafikk blir oftest begrenset til maskiner på samme lokalnett. Routere og brannvegger er ofte konfigurert til ikke å videresende slike UDP-segmenter.

Gjennom *publish-and-subscribe* abstraksjonen kan et MOM (side 209) sende meldinger til flere mottakere, uavhengig av brannvegger og routere (fordi meldingene sendes i flere kopier til individuelle adresser).



**Én-til-en-av-flere** I denne modellen skal meldingen sendes i én kopi, men til en mottaker som vi ikke selv bestemmer. Tidligere i boka har vi vist hvordan et slikt skjema kan brukes til å få en forespørsel «plukket opp» av den tjeneren med ledig kapasitet.

Et MOM vil ikke uten videre tilby et slik meldingsskjema. Vi kan tenke oss en separat «formann» i form av en tjener som andre tjenere kan hente oppdrag fra. Designet av en formann er tatt opp som et teorispørsmål i slutten av kapitlet.

## Multiprosessor-miljø

Vi skiller multiprosessor-konfigurasjoner i to typer: *Tett koplet* og *løst koplet* (side 38). Vi er interessert i å finne ut hvordan vi kan skape nødvendig synkronisering mellom tråder som utfører på forskjellige prosessorer (CPU-er).

I en tett koplet konfigurasjon finner vi minneceller som kan adresseres fra begge CPU-ene, derfor er det mulig å etablere synkroniseringsmekanismer som baserer seg på bruk av *betingelsesobjekter*. Dette

innebærer at det ikke er noen forskjell i måten vi synkroniserer tråder på i en tett koplet multiprosessor-konfigurasjon og i en enkeltprosessor-konfigurasjon.

I en løst koplet multiprosessor har vi en situasjon som er mye mer lik en vanlig distribuert anvendelse. Trådene i de forskjellige CPU-ene har ingen felles adresserbare minneceller, og kommunikasjon og synkronisering mellom trådene må skje gjennom i/o-baserte mekanismer, dvs. meldinger.

**Vi har derfor to typer mekanismer for synkronisering og kommunikasjon:**

1. De som er basert på bruk av felles minneceller, og felles *betingelses*-objekter eller lignende som synkroniserer utføring og datautveksling. Slike mekanismer er begrenset til uniprosessor og tett koplet multiprosessor-konfigurasjon.
2. De som er basert på bruk av i/o-mekanismer, og bruk av *meldinger* for utveksling av data og for synkronisering. Meldingsmekanismer kan kan brukes i alle typer maskinkonfigurasjoner, men er mindre effektive enn bruk av felles minneceller.

| Arkitektur | bruk av felles minneceller | bruk av meldinger |
|---|---|---|
| UNI-prosessor, tett koplet multiprosessor | mulig | mulig |
| Løst koplet multi-prosessor, distribuert arkitektur | ikke mulig | mulig |

## Bruk av Java for distribuert programmering

Java er et moderne programmeringsspråk som er velegnet for utvikling av distribuerte anvendelser. Vi vil beskrive en programmeringsteknikk som kan brukes til dette formålet, i forbindelse med meldingsutveksling gjennom bruk av TCP-forbindelser. Bruk av MOM mellomvare (kalt Java Message Service) eller RMI blir ikke vist som eksempler.

## Sockets

Etablering av TCP-forbindelser og dataoverføring gjennom dem skjer ved objekter av klassen *java.net.Socket* som representerer TCP-forbindelsen.

Etablering av en TCP-forbindelse skjer ved å instansiere et *Socket*-objekt. Objektet er knyttet til inn- og ut-strømmer (Streams) som kan brukes til dataoverføring. Dersom det oppstår feilsituasjoner under instansieringen av objektet eller under dataoverføringen kastes det unntaksobjekter (Exceptions) inn i programutføringen.

På tjenersiden (den siden som «tar imot» en TCP-oppkopling) kreves det instansiert objekt av klasse *ServerSocket*. Metoden *accept()* på dette objektet stopper opp og venter på en TCP-oppkopling fra en klient, og returnerer så et *Socket*-objekt som representerer den etablerte forbindelsen.

Listing 9-1 viser en Java-klasse egnet for å sende en melding (gitt som et String-objekt) og vente på et svar, altså en klasse egnet for bruk av klienten. Det er også denne klassen som setter opp en TCP-forbindelse til tjeneren.

```
import java.net.*;
import java.io.*;
public class SockClient {
   Socket s1;
   BufferedReader br;
   PrintWriter pw;
   // Konstruktøren setter opp TCP-forbindelsen
   public SockClient(String server, int port)
      throws Exception {
      s1 = new Socket(server,port);
      br = new BufferedReader(new
         InputStreamReader(s1.getInputStream()));
      pw = new
         PrintWriter(s1.getOutputStream(),true);
   }
   // Send en String, vent på svar
   public String sendMessageAndReceiveReply
         (String message) throws Exception {
      pw.println(message);
      return br.readLine();
   }
   // Stenger TCP-forbindelsen
```

```
  public void close() throws Exception {
    s1.close();
  }
}
```

*Listing 9-1: Klientside-klasse for en TCP-basert meldingtjeneste*

For klient-siden er en slik meldingsutveksling en «synkron» oppgave, fordi det er klienten som initierer TCP-forbindelsen og som avgjør når det skal sendes en melding. Derfor blir klassen `SockClient` en enkel samling av klassemetoder.

Annerledes blir det med tjenersiden. Tjeneren må alltid være parat til å ta imot en «innkommende» TCP-forbindelse, og må derfor ha en separat tråd som venter på dette. Dette løses ved at tjenerklassen arver fra `Threads`, og vi plasserer `accept()`-kallet inne i `run()`-metoden.

**Flertråds tjener** Java-koden til tjenerklassen er vist i figur 9-2. Legg merke til at den inneholder en indre klasse kalt `SockThread`. Denne klassen er nødvendig for å skape en flertråds tjener. Med en flertråds tjener mener vi i dette tilfellet at den kan motta og sende meldinger over flere TCP-forbindelser samtidig, dvs. snakke med flere klienter samtidig. Dette oppnår vi ved at hver TCP-forbindelse passes av en separat tråd, og denne tråden skapes idet en ny TCP-forbindelse aksep-teres (når `accept()`-kallet returnerer).

Tjeneren kaller kontruktøren til `SockThread` med den etablerte TCP-forbindelsen som en parameter, representert som et `Socket`-objekt. Konstruktøren lagrer en referanse til `Socket`-objektet, skaper de nødvendige datastrømmene og starter den nye tråden med `start()`-kallet. Heretter er det `run()`-metoden i `SockThread`-klassen som leser og skriver data over TCP-forbindelsen gjennom streams-objektene.

```
import java.net.*;
import java.io.*;

public class SockServer extends Thread {
  ServerSocket s1;
  SockMessageHandler callback;
  public SockServer(int port,SockMessageHandler cb)
    throws Exception {
    s1 = new ServerSocket(port);
    callback = cb;
    start();
```

```
  }

  public void close() throws Exception {
    s1.close();
  }

  public void run() {
    try {
      while (true) {
        Socket s2 = s1.accept();
        SockThread st = new SockThread(s2);
      }
    } catch (Exception e) { e.printStackTrace(); }
  }

  class SockThread extends Thread {
    Socket sock;
    BufferedReader br;
    PrintWriter pw;
    public SockThread(Socket s) throws Exception {
      sock = s;
      br = new BufferedReader(
        new InputStreamReader
          (s.getInputStream()));
      pw=new PrintWriter(s.getOutputStream(),true);
      start();
    }
    public void run() {
      try {
        while (true) {
          String message = br.readLine();
          if (message == null) break;
          InetAddress ip = sock.getInetAddress();
          String reply =
              callback.receive(message,ip);
          pw.println(reply);
        }
        sock.close();
      } catch (Exception e) {}
    }
  }
}
```

*Listing 9-2: Tjenerside-klasse for en TCP-basert meldingtjeneste. Legg merke til bruken av flere tråder i utføringen av koden*

Tjenerside-klassen skal kun betjene meldingstjenesten, men ikke behandle meldingsinnholdet. Koden som skal behandle en melding til tjeneren må derfor gjøres kjent for *SockServ*-klassen slik at den kan kalles på riktig tid med riktige parametre. Dette skjer gjennom bruk av *interfaces*.

Koden som skal kalles må foreligge som en metode i en klasse som *implementerer* *SockMessageHandler*. Dette innebærer at den har en metode med signaturen *receive(String, InetAddress)*. Koden til *SockMessageHandler* er vist i listing 9-3. Et objekt av denne klassen legges ved som konstruktørparameter for klasse *SockServer*. Metoden *receive(..)* får overført den mottatte meldingen og klientens IP-adresse, og returverdien fra metoden er en String som sendes tilbake til klienten.

```
import java.net.InetAddress;
public interface SockMessageHandler {
  public String receive(String message,
    InetAddress sender);
}
```

*Listing 9-3: Grensesnittet som må implementeres av tjenerkoden*

Et eksempel på et tjenerprogram som bruker disse klassene er vist på figur 9-4. En mottatt melding blir sendt til metoden receive(..), som i dette tilfellet viser innholdet på skjermen og returnerer svaret «Meldingen har xx tegn».

```
public class SockMain implements SockMessageHandler {

  public SockMain() throws Exception {
    SockServer ss = new SockServer(1958,this);
  }

  public String receive(String message,
      java.net.InetAddress sender) {
    System.out.println("Melding fra: " +
      sender.getHostAddress());
    System.out.println(message);
```

```
      return "Meldingen har " + message.length()
            + " tegn";
   }

   public static void main(String[] args)
         throws Exception {
      new SockMain();
   }
}
```

*Listing 9-4: Tjenerprogram som benytter SockServer-klassen*

Programmet `SockMain` vil lytte på TCP port 1958, og for å teste dette programmet kan vi bruke «telnet». Skriv kommandoen

```
$ telnet localhost 1958
```

(Både Linux og Windows har denne kommandoen.) Skriv inn en melding. Alternativt er å skrive et lite klientprogram i Java, som vist under på listing 9-5. Vi kan også kjøre flere klientprogrammer samtidig for å teste ut tjenerprogrammets flertrådsegenskaper og mange-til-én-synkronisering. Et bilde fra en slik test er vist i figur 9-11.

```
public class SockMainClient {
   public static void main(String[] args)
   throws Exception {
      SockClient sc = new
         SockClient("localhost",1958);
      String reply =
         sc.sendMessageAndReceiveReply(args[0]);
      System.out.println("Svaret er: " + reply);
   }
}
```

*Listing 9-5: Klientprogram som bruker SockClient-classen*

*Figur 9-11: Skjermbilde fra test med én tjener og to klienter. Merk hvordan meldingene fra to klienter blir «flettet» i tjeneren*

## Sammendrag

- De fire fordelene med distribusjon er *datadeling*, *ressursdugnad*, *nettverksøkonomi* og *feiltoleranse.*
- Typiske distribuerte anvendelser er *fildeling*, *databasetjener*, *peer-to-peer-anvendelse*r og *domain name services* (DNS).
- De spesielle problemene knyttet til distribusjon er *mangel på felles klokke*, *mangel på felles tilstand* og *uavhengig krasj og feil.*
- Krasjproblematikken motvirker vi ved bruk av *repeterbare operasjoner* og *tilstandsløse tjenere.*
- Peer-to-peer-anvendelser er *symmetriske*. De må være *selv-konfigurerende* og *skalerbare.*
- Et av operativsystemets oppgaver er å presentere et distribuert miljø som et *single system image.*
- Tråder i ulike maskiner må synkronisere seg gjennom bruk av *meldinger*. Betingelsesobjekter kan ikke brukes.
- Samhandling mellom programmer kan foregå i form av meldinger (MOM) eller som kall på fjernobjekter (RMI).
- Java er et velegnet programmeringsmiljø for distribuerte anvendelser.

| **Sentrale begreper i dette kapitlet:** | |
|---|---|
| klient/tjener | peer-to-peer |

| Sentrale begreper i dette kapitlet: | |
|---|---|
| feiltoleranse | fjernlagring vs. fjernutføring |
| meldingsbasert synkronisering | singe system image |
| message-oriented middleware | remote method invocation (rmi) |

## Teorioppgaver

**Gå sammen i grupper og løs disse oppgavene:**

1 Hvilket av alternativene MOM er RMI gir den beste plattformen for følgende distribuerte anvendelser:

- Et e-postsystem
- Overføring av tale i sann tid
- En database med klient/tjener-anvendelse
- En toalettnøkkel-tjener

2 Finn ut hvordan du kan skrive et Java-program som henter en nettside med HTTP-protokollen og lagrer HTML-koden på en lokal fil.

3 Skriv algoritmen (designet) for en toalettnøkkel-tjener.

4 Lag et design for en «formann» som besørger én-til-en-av-flere-synkronisering. Beskriv hvordan klientene skal bruke denne tjeneren for å «bli satt i arbeid» (room service).

5 En Netware filtjener er «tilstandsorientert»; tjeneren husker hvilke filer som er åpnet, og kan selv sørge for fil-låsing. En NFS-tjener (fildeling på UNIX) er «tilstandsløs», og har ingen kunnskap om tilstandene (åpnet/lukket, låst/ulåst osv.). Begge variantene har sine fordeler og ulemper. Beskriv disse.

## Øvingsoppgaver

Forslag til øvingsoppgaver ligger på bokas nettsted.

**Etter fullførte øvinger bør du beherske:**

1 Kunne skrive en enkel distribuert anvendelse som benytter meldinger som kommunikasjonsform

2. Kunne forstå og modifisere en web-tjener programmert i Java slik at den kan utføre enkle programmerte tjenester.
3. Skrive Java-programmer som henter data fra ftp- og web-tjenere.
4. Demonstrere distribuert synkronisering gjennom bruk av meldinger i nettverket.
5. Demonstrere problemet knyttet til tilstandsorienterte parter i en distribuert anvendelse.