Avdeling for informatikk og e-læring
Geir Maribu

1. Introduksjon til operativsystemer

Resymé: I denne leksjonen vil du få en oversikt over hva et operativsystem er for noe, hvordan det er bygget opp og hvordan det virker.

1.1. Hva er et operativsystem

Operativsystemet er det mest grunnleggende programmet som kjører på datamaskinen din. Ja, for operativsystemet er et program som kjører på datamaskinen på lignende måte som alle andre programmer. Når du kjører et regnearkprogram på maskinen din er det operativsystemet som skaffer ressurser til dette programmet, dvs. plass i minnet, tid for å kjøre på cpu etc. Det er også operativsystemet som hjelper til når du skal skrive ut regnearket til skriver, når tallene skrives til skjerm, og når filen lagres på disk.

Operativsystemet sørger også for at du kan utføre flere jobber samtidig på maskinen, f.eks mens du jobber med beregninger i et regneark, kan det samtidig foregå andre aktiviteter på maskinen, f.eks nedlasting av filer fra Internett, sjekke filer for virus, ta sikkerhetskopi av data etc. Videre kan andre personer være innlogget på den samme maskinen og jobbe på samme måte som du, dvs. kjøre sine egne programmer samtidig med at du kjører dine programmer. I slike tilfeller kreves det av datamaskinen at den håndterer *samtidige* forespørsler mot de samme ressursene, f.eks flere programmer spør etter plass i minnet, tid på cpu etc. på samme tid. Da må operativsystemet styre tilgangen til disse ressursene.

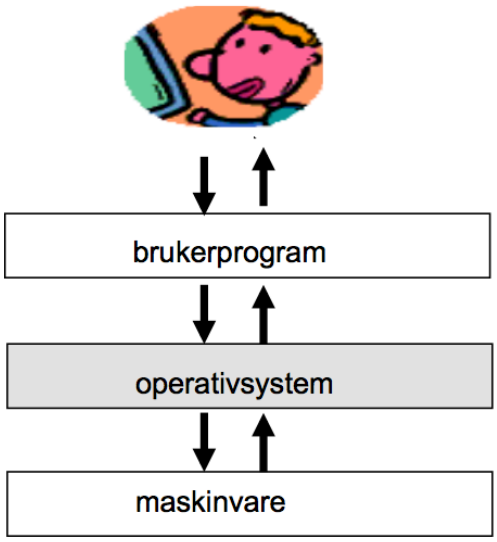
Operativsystemet er et stort og komplisert system av programmer som sørger for at maskinen er tilgjengelig for oss brukere. Vi skal nå sammen prøve å finne ut mer om hvilken rolle operativsystemet har i datamaskinsystemet, hvordan det er bygd opp og hvordan det virker.

1.2. Operativsystemets plassering i systemet

Operativsystemet er plassert mellom maskinvaren og brukerprogrammene. Med maskinvaren menes datamaskinen slik den ser ut når den kommer rett fra fabrikken. Maskinen inneholder da ingen programmer, og du kan derfor ikke gjøre noe fornuftig med maskinen. Med brukerprogrammer (ofte også kalt applikasjonsprogrammer) mener vi de programmene du benytter deg av for å få utført ting, f.eks tekstbehandling, regneark, lese post, surfe på nettet, spille musikk etc.

Operativsystemet stiller seg i mellom disse to lagene slik som vist på figuren nedenfor. Alle brukerprogrammene er avhengig av operativsystemet for å kunne fungere. Uten operativsystemet ville ikke disse brukerprogrammene kunne lastes inn i maskinens minne, ikke kunne ta i mot data fra tastatur, ikke skrive ut data til skjerm etc. Operativsystemet har den fulle kontroll med alt som skjer i datamaskinen.

Operativsystemet aktiviseres automatisk med en gang maskinen slås på, og etter en kort stund er maskinen klar til å ta i mot de første kommandoene fra deg. Egentlig er det operativsystemet som tar i mot kommandoene siden det har kontroll og styring med hele maskinen.



Brukerprogrammene har oppgaver som skal utføres mot maskinvaren, f.eks skrive ut et bilde, lagre ei fil etc. Alle disse oppgavene må kanaliseres gjennom operativsystemet. I operativsystemet finnes det egne funksjoner som sørger for at brukerprogrammene kan skrive til disk eller diskett, vise tekst på skjermen, ta i mot data fra tastatur etc. Operativsystemet fungerer som et *grensesnitt mellom maskinvaren og brukerprogrammene.* Sjelden eller aldri trenger brukerne å forholde seg direkte til maskinvaren som benyttes, og det er en stor fordel da maskinvaren er komplisert, den endrer seg hele tiden, blir mer avansert etc. Ved å la operativsystemet ta seg av disse til dels kompliserte oppgavene, slipper brukerprogrammene å ta hensyn til disse tingene, og dermed blir også jobben mye enklere for de personene som lager brukerprogrammene. De slipper nemlig å ta hensyn til komplisert maskinvare når de lager programmer; det holder at de kaller opp en aktuell funksjon i operativsystemet.

Datamaskiner kan variere ganske mye rent maskinvaremessig alt etter hvilken type maskinen er og hvilken prosessor (cpu) som står i maskinen. Pc-er er basert på INTEL x86-prosessor-klassen er slik sett like maskinvaremessig. Men det finnes også maskiner basert på andre cpu-arkitekturer, f.eks macintosh-maskiner og maskiner basert på alpha- og risc-prosessor.

En programmerer som skal lage et program trenger ikke forholde seg til alle disse forskjellige maskinarkitekturene. Det vil bli et alt for omfattende prosjekt. I stedet forholder programmereren seg til operativsystemet og kaller opp nødvendige funksjoner (systemkall) når f.eks programmet skal skrive til fil, skrive til skjerm etc. Programmereren forholder seg altså til operativsystemet og dets hjelpeprogrammer når brukerprogrammer skal lages. Dersom det aktuelle operativsystemet finnes på den aktuelle maskinarkitekturen, vil programmet kunne kjøre uten problemer og uten at programmereren trenger å tenke på hvordan maskinvaren virker.

1.3. Basisfunksjoner i operativsystemet

Programmereren ser på operativsystemet som et sett av funksjoner som skal forenkle programutviklingen og gjøre det enklere for programmereren å benytte seg av alle de forskjellige maskinvarekomponentene som finnes i datamaskinen og som er koblet til datamaskinen. Viktige basisfunksjoner som programmereren benytter seg av er for eksempel:

- Standard inn/ut-funksjoner for å lese og skrive data til inn/ut-enheter, f.eks skrive til skjerm, tastatur, disk etc.
- Grafikkfunksjoner for å tegne figurer på skjermer.

Det finnes også annen viktig basis programvare for programmereren som ikke nødvendigvis er en del av operativsystemet, men som er en viktig del av datamaskinsystemet, og som "lever" ganske nært operativsystemet. Det er basis programvare for bruk av vindussystemer og databasesystemer. Med enkle funksjonskall kan programmereren bringe fram vinduer på skjermen og hente ut data fra en database.

Når programmereren benytter seg av operativsystemets funksjoner for å få adgang til maskinvaren, er det et standardisert grensesnitt mot maskinvaren som tilbys av disse funksjonene. Det å tilby et slik standardisert grensesnitt kalles for *ressursabstraksjon* fordi programmereren ikke ser ressursen (eller maskinvaren) slik den egentlig er laget, men et standardisert og idealisert bilde av ressursen laget ved hjelp av programvare.

Husk også at andre programmer kjører samtidig på datamaskinen. Alle disse programmene kaller opp operativsystemfunksjoner og ber om å få bruke maskinvareressurser. Siden det er begrenset med ressurser i datamaskinen, kan det oppstå ressurskonflikter mellom programmene. Operativsystemet har derfor også som oppgave å se til at ressursene blir delt mellom programmene, og at de blir *rettferdig* fordelt mellom programmene.

Ressursabstraksjon og *deling av ressurser* er to hovedpilarer som operativsystemer hviler på. Begge disse områdene skal derfor omtales nærmere i de følgende kapitlene.

1.3.1. Ressursabstraksjon – menneskeliggjør oppgaven å programmere

Operativsystemfunksjoner er laget slik at de kan fungere på et bredt sett av lignende maskinvare, og dermed gjøre det til en relativt enkel jobb for programmereren å bruke maskinvaren via disse funksjonene. Operativsystemet tilbyr en *abstrakt modell* av maskinvaren, en modell som *glatter over forskjeller mellom maskinvare av samme type*. Programmereren får dermed en modell av maskinvaren. Denne modellen tilbyr et sett av standardiserte funksjoner for å utføre oppgaver på maskinvaren. Ulempen med denne måte å betrakte maskinvaren på er at spesialegenskaper som den aktuelle maskinvaren har blir utelatt.

Maskinvarekomponenter kalles ofte for *ressurser* i operativsystemsammenheng, og idéen med å lage funksjoner som tilbyr programmererne en standart modell av maskinvaren kalles *ressursabstraksjon*. Det betyr at en ressurs som f.eks en harddisk tilbyr et standardisert grensesnitt via operativsystemfunksjoner. Dette er funksjoner som f.eks *hwrite*, *hread*, *hseek* og lignende. De to første for å skrive og lese datablokker til og fra disk, og hseek for å plassere lese/skrive-hodet til disken på et bestemt sted på diskplata. Ved å bruke disse funksjonene trenger ikke programmereren detaljert kunnskap om hvordan disken virker og hvordan den er bygd opp. Programmereren trenger ikke vite om hvordan disken er organisert i diskplater, spor og sektorer.

Det som programmereren får av operativsystemet er et enkelt, høynivå grensesnitt mot disken; et grensesnitt som er lett å forholde seg til fordi programmereren bare angir nummeret på den datablokka som skal leses. Dette grensesnittet kalles for *systemkallgrensesnittet*. Via disse funksjonene får vi et idealisert bilde av disken bestående av en rekke datablokker i serie nummerert fra 0 til N-1 der N er antall blokker.

Det fine med denne måten å betrakte maskinvareenhetene på er at abstraksjonen kan benyttes på flere lignende maskinvareenheter. De samme funksjonene kan for eksempel også brukes mot en diskettstasjon.

En ulempe med denne måten å betrakte maskinvaren på er selvsagt at spesielle egenskaper som er lagt ned i maskinvaren ikke kan benyttes. Operativsystemet forholder seg nemlig til en standard maskinvareenhet.

1.3.2. Ressursdeling – fordeling av goder

Moderne operativsystemer består av en mengde ressurser (maskinvareenheter) som må administreres av operativsystemet. Grunnen til dette er at det kjøres mange programmer samtidig på datamaskinen, og i alle disse programmene gjøres det kall til operativsystem-funksjoner for å bruke disse ressursene. Operativsystemet har da som oppgave å administrere og fordele ressursene mellom programmene som ber om det. Her er det snakk om programmer som ber om en ressurs, bruker ressursen og deretter frigir ressursen. Det er altså snakk om en samling av ressurser som operativsystemet må administrere.

De ressursene vi snakker om i denne sammenhengen er prosessorer (cpu-er), minne, klokker, disker, terminaler, båndstasjoner, nettverkskort, skrivere, filer, databaser etc.

Et eksempel på hva som kan skje dersom ikke operativsystemet holder orden på dette er at for eksempel tre programmer prøver å skrive ut data til samme skriver samtidig. Da kan en lett se for seg at hvert program får skriveren for en kort periode om gangen. Resultatet kan bli at det først kommer 3-4 linjer fra det første programmet, deretter 3-4 linjer fra det neste etc.

Programmene bruker disse ressursene ved å kalle opp funksjoner i operativsystemet, og det er derfor operativsystemet som må holde orden på disse tingene. Et kall fra et program til en slik funksjon er derfor å betrakte som en forespørsel til operativsystemet om å få lov til å bruke ressursene. Operativsystemet sitter med den fulle oversikten og vil, dersom ressursen ikke er ledig, legge spørrende program i kø, og først når ressursen er tilgjengelig kan forespørselen oppfylles, og dermed kan programmet kjøre videre.

Ressursdeling kan foregå på to forskjellige måter. Det er 1) *plass-deling* og 2) *tids-deling*. Deling av minne (dvs. RAM) er et eksempel på deling av plass mellom flere programmer. Det er nemlig ingenting i veien for at flere programmer kan ligge inne i minnet samtidig. I ressursen cpu derimot er det kun plass til en om gangen. Ressursen cpu er derfor tidsdelt der hvert program får tildelt et lite tidskvant (f.eks 100 msek) om gangen for å kjøre programmet.

Vi kommer sterkt tilbake seinere når det gjelder deling av minne og cpu, da disse ressursene sammen med inn/ut-ressursene er de viktigste i datamaskinsystemet.

1.3.3. Ressursisolering - sikkerhet

Som en konsekvens av at flere programmer kjører samtidig i datamaskinen, og at de etterspør de samme ressursene, dukker problemet med *sikkerhet* opp. Operativsystemet må garantere at ingen andre programmer får tilgang til den samme ressursen på samme tid. Operativsystemet må garantere at ingen andre programmer får lese dataområder tilhørende ditt program.

Med *ressursisolering* garanterer operativsystemet at det ikke foregår uautorisert tilgang til ressurser. Det betyr f.eks at operativsystemet legger programmer (kode og data) i forskjellige områder av minnet uten muligheter for et program å lese dataene til et annet program.

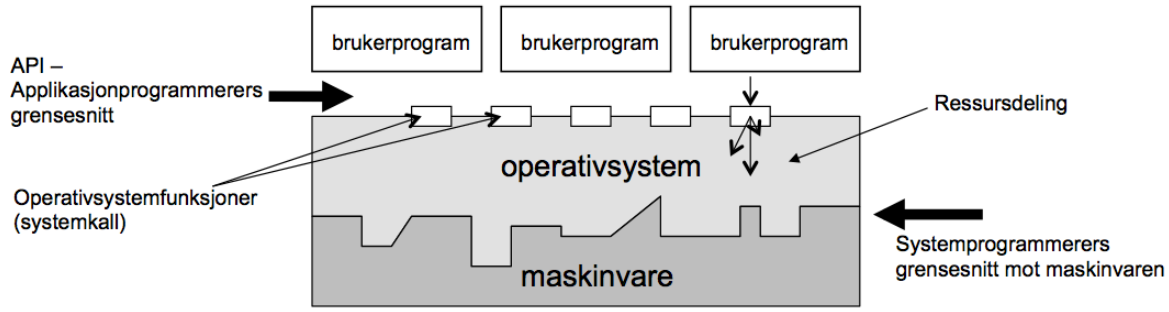
1.3.4. Felles bruk av ressurser - kommunikasjon

Sjøl om ressursisolering er et viktig begrep innenfor operativsystemer, er det også av største viktighet at programmer kan *dele* på ressurser. Det skjer f.eks nå programmer skal kommunisere med hverandre, dvs. utveksle data med hverandre. Det skjer via felles minneområde der begge programmene har tilgang til det samme minneområdet.

Problemet da blir for operativsystemet å tilby mekanismer til programmene, slik at denne kommunikasjonen via felles dataområder foregår på en kontrollert og sikker måte. Derfor trengs det både mekanismer for *ressursisolering* og for *felles bruk av samme ressurs* i operativsystemer.

1.3.5. Oppsummert

Figuren nedenfor viser en skisse over nivåene i et datamaskinsystem med brukerprogrammene øverst, deretter operativsystemet, og nederst maskinvaren. Mellom brukerprogrammene og operativsystemet finner vi operativsystemfunksjonene (også kalt *systemkall*). Dette systemkallgrensesnittet utgjør det som vi kaller for ressursabstraksjon.



Etter at et brukerprogram har kalt opp en funksjon i operativsystemet er det operativsystemet som har kontrollen. Alt som har med ressursdeling, sikringen av ressursisolasjon og felles bruk av ressurser er nå en oppgave som operativsystemet tar seg av slik som vist på figuren over.

Figuren viser også et noe ujevnt og hakkete grensesnitt mellom operativsystem og maskinvare. Dette illustrerer at dette grensesnittet er komplisert og er avhengig av hva slags maskinvare som finnes i datamaskinsystemet.

Det viktige i denne sammenhengen er at når maskinvare skiftes ut, må operativsystemet tilpasse seg denne nye maskinvaren. Dette gjøres ved å legge inn såkalte *driverprogrammer* som styrer maskinvaren (mer om driverprogram seinere). Selv om maskinvaren skiftes ut, behøver en ikke endre grensesnittet mellom brukerprogram og operativsystem. Dette er det samme hele tiden, og brukerprogrammene vil derfor ikke merke at maskinvaren er endret.

1.4. Programmer og prosesser

Foran har vi snakket mye om programmer som kjører på datamaskinen, og at flere programmer kjører samtidig. Vi skal nå innføre et nytt begrep kalt *prosess* som skiller seg en del fra begrepet program.

- Kort kan en si at en *prosess* er et program under utførelse.

Med et program mener vi selve programkoden, dvs. programkoden slik den ligger på disken "helt død" på ei fil. Programmet kan sammenlignes med ei kakeoppskrift i ei kokebok. Den bare er der i boka, ingenting baker seg selv. Vi må sette i gang en aktivitet før det blir kake av det, vi må altså bruke oppskriften. I dette tilfellet kan vi si at kakeoppskriften er programmet, og kakeform, mixer, mel, sukker etc. er ressurser. Den aktiviteten som oppstår kaller vi for en prosess.

Tilsvarende er det med et program. Det blir til en prosess når det starter opp et dataprogram på datamaskinen, og da trenger det ressurser som minne, cpu, inn/ut-enhet etc. *En prosess er altså et program under kjøring i maskinen*. Det er i programmet prosessen finner hvilke instruksjoner som skal utføres, og for å kunne utføre instruksjoner trengs ressursen *cpu*, og for å lagre instruksjoner og data trengs ressursen *minne*.

Når prosessen er ferdig å kjøre, gir den fra seg ressursene og avslutter. Prosessene finnes ikke lenger i datamaskinssystemet. Det eneste som er tilbake er programmet og dataene som er produsert (på samme måte som med kakebaking: Det som er igjen er kakeoppskriften og den ferdige kaken).

I det følgende vil vi alltid snakke om prosesser når det er snakk om programmer som kjører. Med et program vil vi nå mene selve instruksjonene som er lagret på ei fil på disken, dvs. den kjørbare filen. Heretter vil vi snakke om flere prosesser som kjører samtidig på maskinen, vi vil snakke om en regnearkprosess, nettleserprosess, virussjekkeprosess etc. når det er snakk om programmer som kjører.

1.5. Ressurser

Foran har vi snakket om maskinvareenheter som ressurser, og at prosesser spør operativ-systemet etter ressurser, og at operativsystemet må administrere ressursene. Mer formelt kan en si at en ressurs kan betraktes på følgende måte:

- En prosess må spørre operativsystemet etter en ressurs.
- Prosessen må suspendere kjøringen inntil ressursen er gjort tilgjengelig for prosessen (eller *allokert* til prosessen som vi sier).

En *ressurs* er altså en abstrakt maskinvarekomponent som prosessen må ha allokert til seg før den kan kjøre videre. Prosessen blir midlertidig stoppet dersom den har bedt om en ressurs, og denne ressursen ikke er tilgjengelig.

Du som vanlig bruker vil nok se på ei fil som den mest iøynefallende ressursen i et datamaskinsystem. En prosess må åpne filen (eller be om filen) før den kan lese og skrive fra filen. Det kan hende at filen ikke er tilgjengelig idet prosessen ber om filen, f.eks fordi en annen prosess bruker filen. Da må prosessen vente til den blir tilgjengelig.

1.5.1. Filer

En sekvensiell *fil* er slik den er f.eks definert i Linux en sekvens av bytes (tegn). Først åpnes filen, deretter skrives det data til filen, f.eks et gitt antall bytes. På tilsvarende måte kan du lese data fra filen.

Filen er et navngitt objekt som lagres på disken sine spor og sektorer. Her ser vi altså den samme abstraksjonen som vi har snakket om tidligere. For oss brukere ser filen at som en sekvens av bytes, mens det fra operativsystemet sin side er datablokker som lagres på ledige

plasser på disken. Datablokkene er nummerert, og de trenger ikke lagres sammenhengende på disken. På enda lavere nivå i operativsystemet, på drivernivå, snakker vi om lagring av data på bestemte spor og sektorer til disken. Men som sagt, på brukernivå og programnivå er filen et navngitt objekt som inneholder en sekvens av tegn. Enkelt, ikke sant?

Alle ressurser, også filer, har en beskrivelse knyttet til seg. Dette kalles ofte en deskriptor, og for objektet fil inneholder denne deskriptoren dato filen ble opprettet, størrelsen til filen, adresse til datablokker, tilgangsrettigheter etc.

Deskriptoren er et meget viktig element for operativsystemet når det skal administrere alle ressursene, stille ressursene inn i kø til de forskjellige prosessene, sjekke om ressursen er ledig eller opptatt etc.

1.5.2. Prosessoren (cpu-en)

Prosessoren er den ressursen som trengs for i det hele tatt å kunne kjøre prosessen. Denne ressursen blir ikke forespurt eksplisitt siden den er nødvendig for i det hele tatt å kunne kjøre prosessen.

Først når alle andre ressurser er på plass hos prosessen kan prosessen nyttiggjøre seg cpu-ressursen. Prosesser kan befinne seg i forskjellige tilstander i datamaskinsystemet: 1) Ventende på i/o-ressurser, 2) ventende på cpu-ressursen eller 3) kjørende på cpu-ressursene.

1.5.3. Minnet

En prosess som skal kjøre vil alltid trenge plass i minnet. Dette kan gjøres på forskjellige måter, enten ved at prosessen får alt minnet den trenger ved oppstart, eller at minnet som prosessen har behov for allokeres dynamisk etter hvert som behovet melder seg.

1.6. Prosesser

Kort fortalt sier vi at en prosess er et program under utførelse. En prosess består av følgende komponenter:

- *Programkoden* som skal kjøres. Med programkoden menes instruksjonene som befinner seg i den kjørbare fil som er lagret på disken, dvs. binærkoden til programmet.
- *Dataene* som programkoden bruker under kjøringen. I programkoden vil det alltid være referanser til dataområder. Det kan være enkle referanser som i=i+1 eller det kan være regnearkprogrammets referanse til tall og formler.
- *Ressurser* som kreves av programmet, f.eks ei fil som må åpnes for å hente fram nødvendige data.
- *Status* til prosessen under kjøringen. Med status menes her ganske mange data som beskriver prosessens liv og levnet. Den kalles ofte for en prosess-deskriptor.

Forskjellen mellom program og prosess kan også uttrykkes på følgende måte: Programmet er en *statisk* enhet bestående av programinstruksjoner som *definerer* prosessens oppførsel nå den kjøres på et sett av data. Prosessen derimot er en *dynamisk* enhet som *kjører* programmet på et sett av data.

Dette betyr f.eks at to forskjellige prosesser kan kjøre det samme programmet, men bruker sine egne private dataområder.

1.7. Flere prosesser kjører samtidig

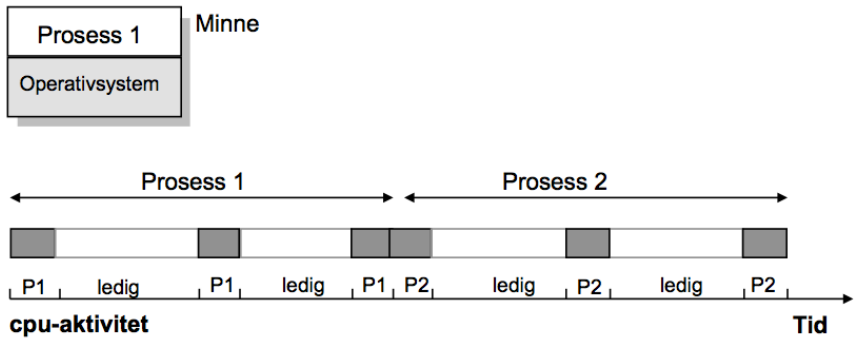
1.7.1. Sekvensiell kjøring av prosess

Ved sekvensiell kjøring er det kun en prosess i minnet om gangen. Først kjører prosess nr 1 på cpu-en. Når denne prosessen utfører i/o, enten mot disk eller mot tastatur/skjerm så trengs ikke cpu-en lenger, og er derfor ledig. Den har rett og slett ingenting å gjøre. Den bare venter.

I forhold til den tida som cpu-en bruker på å utføre programinstruksjoner er denne ventetida et "hav" av tid. I så måte er målestokken i figuren nedenfor ganske misvisende. Et typisk forhold mellom kjøring og venting vil være 1 til 1 million, dvs. cpu-en kjører i 1 tidsenhet og venter i 1 million tidsenheter. Det kan bli noe vanskelig å illustrere direkte grafisk. Men husk at det er slike størrelsesordener vi snakker om her.

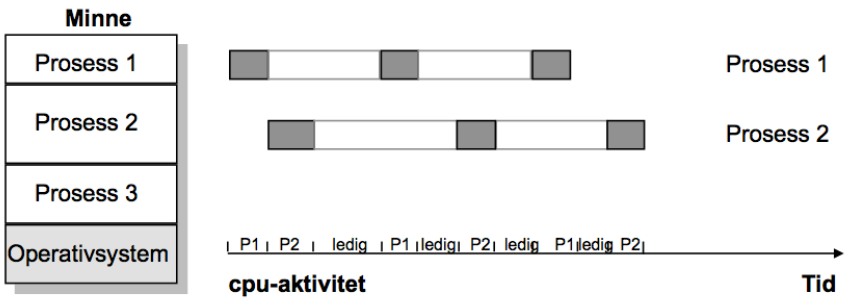
Etter at prosess 1 er helt ferdig er det prosess 2 sin tur å kjør på akkurat samme måte.

Som vi allerede skjønner - det kunne vært greit å utnytte cpu-en til andre ting i den lange ventetida når en prosess utfører i/o. Sekvensiell kjøring av prosesser gir meget dårlig utnyttelse av cpu-en.



1.7.2. Kjøring av flere prosesser samtidig

Nå skal vi se hvordan det er mulig å kjøre flere prosesser samtidig på maskinen, såkalt *multitasking* (en ser ofte at ordet *task* brukes i stedet for prosess, og da særlig når vi snakker om flere prosesser som kjører samtidig). Forutsetningen for at dette skal kunne virke noenlunde skikkelig er at flere prosesser får plass i minnet samtidig slik som vist i figuren nedenfor.



Grunnlaget for at multitasking fungerer er at prosessene må utføre i/o-operasjoner inni mellom. Det betyr at prosessen må avgi cpu-en til annen prosess, og denne prosessen må befinne seg i minnet allerede. I/o-operasjoner tar meget lang tid i forhold til kjøring på cpu-en

så flere prosesser kan faktisk få utført en god del arbeid mens den første prosessen utfører i/o-operasjon.

I/o-operasjonen kan bestå i å lese data fra tastatur eller å lese/skrive data fra/til harddisk. I begge disse tilfellene kan det være lang ventetid, f.eks innskriving av data via tastatur, flytting av lese-skrive-hode til rett sted på diskplata etc.

Det er mulig å kjøre flere prosesser samtidig dersom alle prosessene er i minnet, og de vekselvis har behov for kjøring på cpu og lesing/skriving av data mot i/o-utstyr. Dette kalles *multi-tasking*.

Datamaskinen begynner altså å pådra seg en del administrative oppgaver i forbindelse med kjøring av prosesser. Det opereres med flere prosesser samtidig inne i minnet, og disse må kunne veksle på å bruke cpu-en. For å få til dette må det organiseres en kø-ordning i datamaskinen. Videre må det ordnes med beskyttelsesmekanismer slik at prosessene ikke ødelegger for hverandre.

Aner vi et begynnende operativsystem her? Vi trenger altså et system som kan forholde seg til mer enn en aktivitet på maskinen, et system som kan fordele og administrere ressurser.

1.8. Instruksjonssyklusen

For å kunne forstå hvordan operativsystemet arbeider må du ha kjennskap til instruksjons-syklusen. Nedenfor gis det en kort repetisjon.

Programmet som kjører på maskinen består av et sett med instruksjoner som ligger i minnet. Under kjøringen av programmet, dvs. når prosessen kjører, hentes instruksjoner fra minnet og over til cpu-en og utføres der en etter en.

Enkelt kan en se på programkjøring som bestående av to steg:

- Cpu-en henter en instruksjon fra minnet
- Cpu-en utfører denne instruksjonen

Kjøring av program består av gjentatte operasjoner der henting og deretter utføring av instruksjoner blir utført om og om igjen. Selve utføringen av instruksjonen består av flere trinn, alt etter hva slags type instruksjon det er snakk om.

Instruksjonssyklusen er all den prosesseringen som foregår fra en instruksjon hentes i minnet og til den neste skal hentes. De to operasjonene nevnt ovenfor kalles henholdsvis for *hentesyklusen* og for *utføringssyklusen*.

1.8.1. Henting og utføring av instruksjon

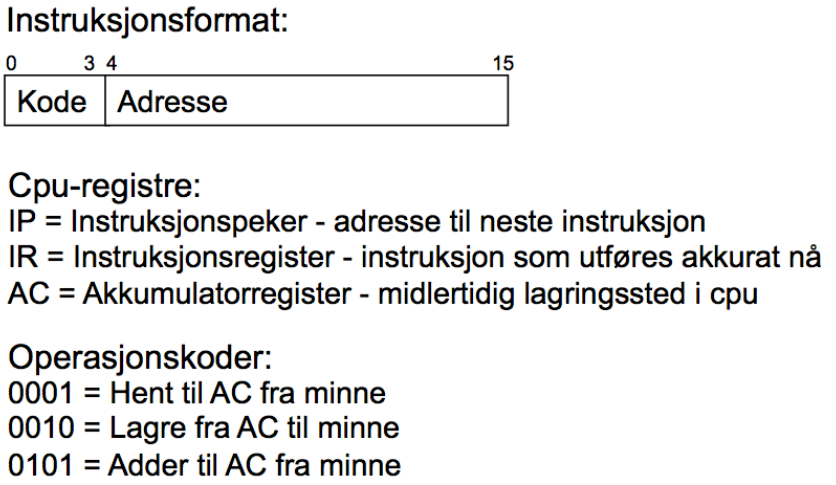
Hver instruksjonssyklus starter med at neste instruksjon hentes fra minnet og flyttes over til cpu. I cpu-en brukes typisk et register kalt IP-registeret (instruksjonspekeren) for å angi neste instruksjon som skal hentes til cpu. Ved oppstart av et program peker denne alltid til første instruksjon som skal utføres (vanligvis på programadresse 0). Etter hvert som instruksjoner hentes, blir innholdet i IP-registeret økt med en for dermed å peke på neste instruksjon som skal hentes. Noen ganger vil IP-registeret få helt nytt innhold. Det skjer når program-instruksjoner (IF-setninger, løkker etc.) fører til hopp i programmet.

Instruksjonen som hentes fra minnet plasseres i IR-registeret (instruction register) i cpu-en. Instruksjonen er en binær kode som angir hvilken operasjon (f.eks legge sammen tall, flytte tall til/fra minnet etc.) som cpu-en skal utføre. Cpu-en tolker instruksjonen og utfører deretter

de operasjonene som er angitt i instruksjonen. Nedenfor ser du en oversikt over hva slags operasjoner som er aktuelle:

- *Cpu-minne:* Flytting av data fra cpu til minne og motsatt.
- *Cpu-i/o*: Flytting av data mellom cpu og i/o-enheter.
- *Dataprosessering:* Utføring av aritmetiske og logiske operasjoner på data
- *Styring:* En instruksjon kan angi at rekkefølgen på utførelsen av instruksjoner skal endres. Det kan f.eks hende at den instruksjonen som er hentet inn akkurat nå (fra adresse 162) angir at neste instruksjon ligger på adresse 195. Neste instruksjon hentes derfor fra adresse 195, og ikke fra 163 som normalt.

Figuren nedenfor viser instruksjonsformat, cpu-registre og mulige operasjonskoder for en tenkt maskin.

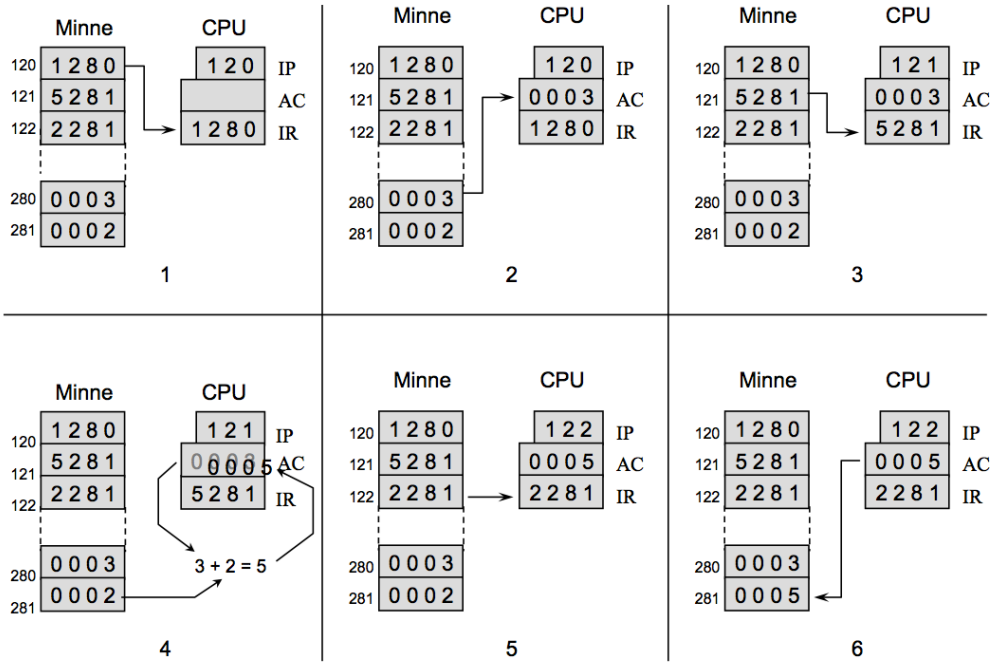


Den tenkte maskinen ovenfor har kun et dataregister som i dette eksemplet kalles akkumulator-registeret (AC). Instruksjonsformatet angir 4 bit for operasjonskoden og 12 bit for direkte adressering av minnet. Det betyr 16 mulige operasjonskoder og 4096 minneadresser som kan nås direkte.

Figuren vist nedenfor viser et utsnitt av en programkjøring der aktuelle deler av cpu og minne er vist. Programsekvensen legger sammen innholdet i minneadresse 280 med innholdet i minneadresse 281, og lagrer deretter resultatet i sistnevnte minnelokasjon, dvs. adresse 281. Til sammen 3 instruksjoner trengs for å utføre denne operasjonen. Hente- og utføringsfasene til hver av disse instruksjonene er:

1. IP inneholder adresse 120 som er adressen til første instruksjon i denne sekvensen. Innholdet til adresse 120 hentes til IR-registeret i cpu-en.
2. De 4 første bit'ene i IR angir at AC-registeret skal fylles opp med data fra minne, nærmere bestemt fra adresse 280 slik som angitt i de 12 neste bit-ene i instruksjonen.
3. IP økes med 1 og neste instruksjon hentes til IR.
4. Innholdet i AC-registeret og innholdet i minne-adresse 281 adderes, og resultatet lagres i AC.
5. IP økes med 1 og neste instruksjon hentes til IR
6. Innholdet i AC lagres i minneadresse 281

7. I eksemplet er der brukt 3 instruksjonssykluser, som hver består av en hentesyklus og en utføringssyklus.



1.8.2. I/O-instruksjoner

Ovenfor så vi på operasjoner der cpu-en hentet og lagret data i primærlager. Men cpu-en kan også utveksle data med i/o-enheter, f.eks lese data fra harddisken, skrive data til harddisken, sende data til skriveren etc.

Ved hjelp av i/o-kontrollere kan cpu-en utveksle data direkte med i/o-enheter. På samme måte som cpu-en kan utføre operasjoner for å lese og skrive data til minnet, kan cpu-en også lese og skrive data til en i/o-kontroller, og dermed til en i/o-enhet.

Dette foregår ved at cpu-en kjører program-instruksjoner som opererer mot kontrolleren på samme måte som vist ovenfor der cpu-en opererte mot minnet. Program-instruksjoner som opererer mot i/o-kontrollere befinner seg i *driver-programmet* for i/o-enheten.

Noen ganger kan det også være aktuelt å la i/o-operasjonen gå direkte mellom minnet og i/o-kontrolleren uten inngripen fra cpu-en. Dette kalles for *direct memory access* (DMA).

1.9. Avbrudd

Avbrudd er en mekanisme som trengs i alle datamaskiner for å kunne utnytte ressursene effektivt.

Avbrudd er nødvendig for å kunne gjøre bl.a følgende:

- Ved f.eks utskrift til harddisk vil operativsystemet operere mot en i/o-enhet som er mye langsommere enn f.eks cpu og primærlager. Det kommer bl.a av det faktum at det tar tid å flytte lese/skrive-hodet til disken, det tar tid å overføre data etc. I stedet for at cpu-en skal "sitte og vente" til dataoverføringen til disken er ferdig bør cpu heller kjøre et annet program i denne perioden. Men da må en ha et system slik at i/o-enheter (i dette tilfellet disken) kan si fra når dataoverføringen er ferdig, dvs. et *avbruddssystem.*

- Når et vanlig program kjører må det være mulig å få oppmerksomhet fra maskinen på et eller annet vis. F.eks hvis du flytter på musa, taster inn noen tegn, det kommer meldinger inn via nettverkskabelen, skriveren er tom for papir etc. I alle disse tilfellene må kjørende program kunne avbrytes og operativsystemet må behandle avbruddssignalet som kom.
- Moderne operativsystemer fungerer etter et prinsipp der alle program skal få hver sin del av cpu-tida, dvs. multitasking. Da er det behov for å kunne bryte av kjørende prosess og erstatte den med neste prosess i køen.

Avbruddsmekanismen er altså med på å effektivisere bruken av cpu-en. Noe av poenget er å unngå at cpu må stå ledig mens langvarige og langsomme i/o-operasjoner utføres.

1.9.1. Avbrudd og instruksjonssyklusen

Tenk deg følgende situasjon: En i/o-enhet (f.eks et nettverkskort) har allerede fått en del data som skal sendes ut på nettverkskabelen. Når den første mengden av data er sendt ut vil i/o-enheten gi et avbrudd til cpu for å si fra at den er klar til å ta i mot mer data. I/o-enheten sender et såkalt *interrupt-request*-signal til cpu-en.

Cpu-en svarer med å avbryte kjørende program, og i stedet kjøre instruksjoner til et såkalt *avbruddsprogram*. Avbruddsprogrammet er et sett av instruksjoner for å behandle akkurat det avbruddet som kom i dette tilfellet. Hver avbruddstype har sitt eget avbruddsprogram.

Etter at avbruddsprogrammet er kjørt ferdig (tar kun kort tid) returneres det til programmet som ble avbrutt.

Det er viktig å merke seg følgende her: Sett fra det programmet som ble avbrutt er dette et helt vanlig avbrudd i programutførelsen. Etter en kort stopp for å håndtere selve avbruddet vendes det tilbake til kjørende program igjen. *Det avbrutte programmet trenger ikke inneholde spesiell programkode for å håndtere avbrudd; dette er det operativsystemet som skal ta seg av.*

Cpu sjekker om nye avbrudd er ankommet før hver instruksjon hentes fra primærlageret. En kan si at instruksjonssyklusen består av 3 steg:

1. Hente ny instruksjon
2. Utføre (kjøre) instruksjon
3. Sjekke om nye avbrudd er ankommet

Det betyr selvsagt noe ekstra administrasjon, men når alternativet er at cpu må vente til i/o-enheter blir ferdige, så er dette allikevel mer effektivt.

1.9.2. Hva skjer når avbrudd ankommer?

Et avbrudd setter i gang flere aksjoner både i maskinvare og programvare. Nedenfor er det satt opp ei liste over disse aksjonene:

1. I/o-enheten sender et avbruddssignal til cpu-en
2. Cpu gjør ferdig instruksjonen den holder på med akkurat nå, og svarer deretter på avbruddssignalet
3. Cpu sender et signal tilbake til i/o-enheten for å bekrefte at avbruddet er mottatt.
4. Cpu gjør forberedelser for å kjøre avbruddsprogrammet til dette avbruddet. Først må det lagre unna statusinformasjon om kjørende program, bl.a innhold i cpu-registre, adresse til neste instruksjon som skal kjøres, etc.

5. Cpu laster inn en verdi i IP-registeret som peker på første instruksjon i avbrudds-programmet. Det betyr at nå hentes instruksjonene fra avbruddsprogrammet sine minne-adresser.
6. Avbruddsprogrammet kjører på cpu-en. Det sjekker bl.a status til i/o-enheten, finner ut hvorfor avbruddet kom, og sørger deretter for at nødvendige operasjoner for å betjene avbruddet blir utført.
7. Når avbruddsprogrammet er ferdig vil cpu hente inn den lagrede informasjonen om det avbrutte programmet, og fortsette kjøringen av dette på samme sted som det ble avbrutt. Dette er mulig fordi både registerinnhold og innholdet i instruksjonspekeren (IP) ble gjemt unna.

1.9.3. Avbrudd og multitasking

Multitasking bygger på det faktum at i/o-operasjoner tar lang tid, og at i stedet for å vente kan en kjøre andre program på cpu-en. Mens et program venter på overføring av data fra f.eks harddisk, kan et annet kjør på cpu-en.

Med flere prosesser inne i minnet samtidig kan en oppnå en meget effektiv utnyttelse av cpu. Når en prosess kaller opp en i/o-rutine for å lese ei datablokk fra disk kan operativsystemet gjøre klar for denne operasjonen, og deretter overføre cpu til neste prosess som ligger i køen for å bruke cpu. På denne måten får en kjørt flere prosesser "samtidig". Hver gang en prosess kaller opp en i/o-rutine hentes neste prosess inn (den som ligger først i køen, eller den som har høyest prioritet).

Teknikken med multitasking er helt og holdent avhengig av en avbruddsmekanisme som fungerer.

1.10. Oppgaver til operativsystemet

Operativsystemet har mange forskjellige oppgaver, og dersom vi skulle forsøke å klassifisere disse vil følgende fire oppgaver stå sentralt:

- Utstyrsadministrasjon
- Prosess- og ressursadministrasjon
- Minneadministrasjon
- Filadministrasjon

Her skal det gis en kort omtale av hver av disse funksjonene. Seinere skal hver av disse gis en mye mer utførlig omtale.

1.10.1. Utstyrsadministrasjon

Vi har tidligere sette at ressursabstraksjon er meget viktig i sammenheng med operativ-systemer. Det innebærer at operativsystemet ser på alle utstyrsenheter som f.eks disker, terminaler, skrivere etc. via et standardisert grensesnitt. Sett fra operativsystemet ser derfor alle disse enhetene like ut.

Utstyrsadministrasjon dreier seg om hvordan operativsystemet håndterer *allokering*, *isolering* og *deling* av utstyrsenheter.

Utstyrsenheter trenger et driverprogram for å kunne fungere sammen med operativsystemet. Driverprogrammet er et program som står mellom operativsystemet og selve utstyrsenheten og sørger for at brukerprogrammene kan nyttiggjøre seg utstyrsenheten.

I prinsippet kan en vanlig bruker skrive et driverprogram for en utstyrsenhet, og installere dette sammen med resten av operativsystemet. Moderne operativsystemer, som f.eks Linux, er laget slik at dette er fullt mulig uten at hele operativsystemet må rekompileres.

1.10.2. Prosess- og ressursadministrasjon

Prosessen er den grunnleggende enheten for kjøring av programinstruksjoner. Men en prosess har ingen hensikt uten ressurser som f.eks cpu, minne etc.

Prosessadministrasjon innebærer funksjoner for å opprette nye prosesser, avslutte prosesser, sette prosesser i ventekø, sette prosesser i kjørende tilstand etc. I tillegg til selve prosessbegrepet finnes det også en enhet for kjøring av instruksjoner som kalles for en tråd-prosess (”threads” eller lettvekts-prosess). Innenfor en prosess kan den det være mange slike tråd-prosesser som løper samtidig.

Operativsystemer skiller mellom 1) kjøring av prosesser på cpu, og 2) allokering av ressurser til prosessen. Operativsystemet vil alltid fordele kjøretid mellom prosesser, dvs. sette av tid på cpu-en til hver prosess. Når en prosess består av flere tråd-prosesser er det hver slik tråd-prosess som tildeles cpu. Det betyr at innenfor en prosess kan det være flere slike tråd-prosesser aktive, og det er tråden som er enhet for cpu-tildeling.

I tillegg er det også snakk om allokering av ressurser til prosesser, f.eks minneområde, åpne filer, utstyrsenheter etc. Ressurser tildeles prosessen som helhet. Det betyr at alle trådene deler på de ressursene som prosessen skaffer.

Kort kan en si at *prosessen er enhet for ressurstildeling* og *tråden er enhet for tildeling av kjøretid* (dvs. cpu).

I operativsystemet finnes det en cpu-administrator som automatisk tildeler ledig cpu-tid til alle prosessene (eller trådene kan vi også si) i datamaskinsystemet. Tildelingen av cpu foregår automatisk; prosessene behøver ikke be om cpu eksplisitt. Det antas at en prosess som har fått tak i alle øvrige ressurser den trenger, også har behov for cpu, og derfor tildeles denne automatisk. Hele poenget er jo at prosessen skal kjøre på cpu-en

I tillegg til cpu trenger prosessene også andre ressurser, f.eks en i/o-enhet. Disse enhetene må prosessene be eksplisitt om ved å spørre operativsystemet. En egen ressursadministrator tar i mot slike forespørsler. Prosessen får ressursen dersom den er ledig, eller prosessen settes i ventekø og venter inntil ressursen igjen er ledig.

For at operativsystemet skal kunne administrere prosesser og ressurser må disse være representert på en måte slik at de lar seg administrere av et system slik som operativsystemet er, dvs. av et dataprogram. Måten dette gjøres på er via såkalte *deskriptorer*. Det er et sett av data (en datastruktur) som representerer prosessen eller ressursene. I deskriptoren lagres all nødvendig informasjon om prosess eller ressurs. Det er denne deskriptoren som operativsystemet flytter fram og tilbake mellom cpu-kø og diverese vente-køer (f.eks i/o-kø).

Når en prosess får tak i en ressurs, så er det denne deskriptoren den får tilgang til. Dersom prosessen skal sende data ut på nettverkskabelen sendes dataene via en deskriptor som er knyttet til nettverkskortet. Den inneholder både et bufferområde og statusinformasjon om trafikken ut på nettet.

1.10.3. Minneadministrasjon

For å kunne kjøre flere prosesser samtidig må det finnes en avansert minneadministrasjon som tilbyr plass til flere prosesser i minnet samtidig, tillate at prosessen plasseres på vilkårlig plass i minnet, tillate at kun aktive deler av prosessen er i minnet under kjøring etc.

Minneadministratoren må sørge for at prosessene ikke ødelegger for hverandre. Det betyr at de ”lever” i isolasjon og dermed ikke har tilgang til hverandres minneområder når de kjører. Men når det er sagt skal det også sies at et avansert system for minneadministrasjon også må kunne håndtere deling av et felles minneområde mellom prosesser når det behovet er til stede.

Moderne systemer for minneadministrasjon benytter såkalt *virtuelt minne*. Det betyr at prosessene ”ser” et minneområde som er mye større en tilgjengelig plass i fysisk minne (dvs. RAM). Dette er mulig fordi minneadministratoren bruker harddisken som et utvidet minne, og betrakter minnet som en abstrakt ressurs bestående av både fysisk minne og tilgjengelig lagringskapasitet på harddisken. RAM og disk sees altså under ett.

Minneadministrasjonen kan utføres på forskjellig vis. *Sidedelt minneadministrasjon* og *segmentdelt minneadministrasjon* er de viktigste. I moderne operativsystemer er disse slått sammen til en metode kalt *minneadministrasjon med sidedelte segmenter*.

1.10.4. Filadministrasjon

Filer representerer lagringsenheten til datamaskinen, og slik sett er filene en abstraksjon av harddisken (eller en annen fysisk lagringsenhet). Filen er en navngitt lagringsenhet som brukere og programmer kan lagre dataene sine på. Filer kan opprettes, kopieres, slettes, navngis, sette tilgangsrettigheter på etc. Videre kan det skrives data til en fil, og det kan leses data fra en fil.

Bak kulissene vil operativsystemet hele tiden arbeide mot en harddisk bestående av plater, spor og sektorer for å oppfylle alle de ønskene som brukeren eller programmet har.

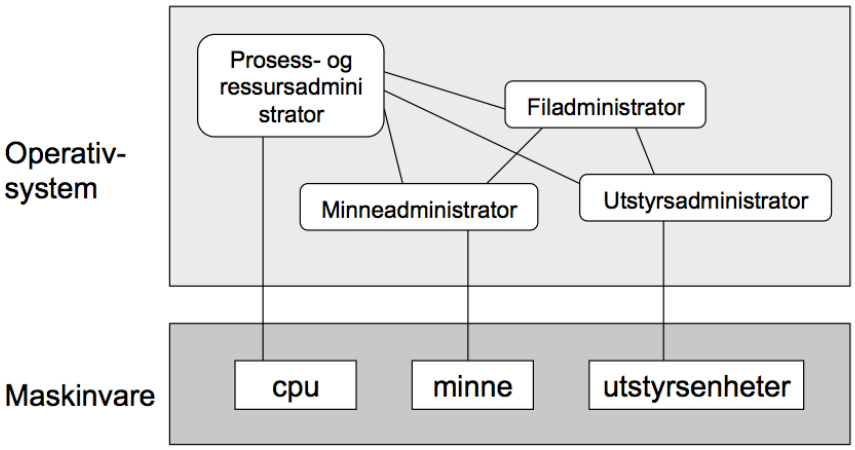
Fil-abstraksjonen har vist seg så nyttig at den benyttes generelt mot alle enheter der data leses og skrives. I Linux for eksempel er alle i/o-enheter, som f.eks tastatur, mus, cd-rom, skriver etc. representert av filer på */dev*-katalogen. Det er faktisk bare å sende data til den aktuelle utstyrsfilen så havner dataene automatisk på det aktuelle utstyret. /dev-katalogen viser vei til de aktuelle utstyrsdriverne.

Filadministrasjon må også kunne håndtere distribuerte filsystemer der filene like gjerne kan ligge lagret på en lokal maskin som på en fjern maskin når disse maskinene er sammenkoblet i et nettverk.

1.10.5. Organiseringen av operativsystemets basisfunksjoner

Basisfunksjonene som er beskrevet foran utgjør hoveddelene av operativsystemet slik som vist i figuren nedenfor. Disse funksjonene legger et abstraksjonsnivå på toppen av maskinvaren, et abstraksjonsnivå som oppretter prosesser og tildeler ressurser til prosessene, som fordeler cpu til prosessene, som administrerer kjøring av flere prosesser samtidig etc.

Via disse funksjonene gis prosessene er virtuelt minne som er en abstraksjon av det fysiske minnet. Dette virtuelle minnet understøtter prosess- og ressursadministratoren slik at flere prosesser kan ligge i minnet samtidig, og dermed mye enklere kan settes til å kjøre samtidig.



Legg merke til at det er stor grad av kommunikasjon mellom basisfunksjonene. De er altså i høy grad avhengig av hverandre. Prosesser trenger f.eks å lese filer, og henvendelsen går derfor til filadministrator som igjen benytter seg av utstyrsadministrator, som sender henvendelsen videre til aktuelt utstyr (som for eksempel harddisken).

Filadministrator benytter seg også av minneadministrator fordi i mange tilfeller ligger dataene i minnet i stedet for på disken. Det er for eksempel tilfelle når dataene befinner seg i bufferområder. Da er det ingen grunn til å gå videre ut til disken for å hente dataene. I stedet tas det en snarvei for å hente de i bufferområdet i minnet.

1.11. Kjernemodus og brukermodus

I tidligere operativsystemer, f.eks MS-DOS og tidlige versjoner av Windows var alt tillatt. Der kunne du kan lage programmer som kunne lese fra og skrive til hvilken som helst minneadresse i minnet. Det samme kunne også gjøres mot harddisken. Det fantes nemlig ingen beskyttelses-mekanismer som hindret et program i å lese/skrive vilkårlig steder i minnet og på harddisk.

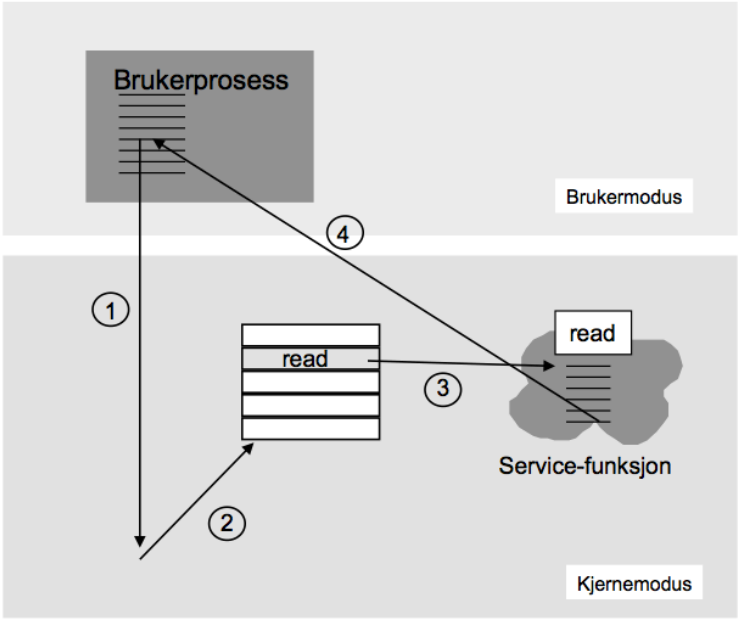
Dette er kanskje akseptabelt for enkle systemer som MS-DOS, men for operativsystemer av typen Linux med mange brukere og prosesser som kjører samtidig er dette ikke holdbart. En må for all del unngå at en gitt prosess leser data tilhørende andre brukere - enten dette er i minnet eller på disk. Det skal være umulig for en prosess å lese fra andre deler av minnet enn de delene som er reservert til denne prosessen. Mens en prosess i enkle operativsystemer av typen MS-DOS i prinsippet har adgang til hele minnet, vil f.eks Linux kun reservere en del av minnet til hver prosess. Hvis en prosess prøver å referere til andre adresser enn de som er avsatt til denne prosessen, vil prosessen bli avbrutt, og deretter avsluttet.

Alle prosesser kan befinne seg i to forskjellige modus:

- brukermodus
- kjernemodus

En prosess som utfører instruksjoner mot sine egne dataområder i minnet opererer i *brukermodus*. Men så snart prosessen trenger å lese fra tastatur, fra disk, skrive til skjerm, etc. må prosessen utføre *systemkall*. Dette innebærer at prosessen går over i *kjernemodus*. I kjernemodus opererer programmets instruksjoner mot operativsystemets dataområder. Dette er en modus som er fullstendig styrt og kontrollert av operativsystemet. På denne måten sikrer

operativsystemet at din prosess ikke gjør ulovlige operasjoner, f.eks leser fra andre sine områder på disken, skriver over viktige systemområder på disken, endrer viktige systemparametre etc. Når operativsystemet har utført systemkallet returneres det til kallende prosess, og dermed til brukermodus slik som vist i figuren nedenfor.



Kort forklaring til figuren ovenfor:

1. Brukerprosessen gjør et såkalt systemkall til kjernen i operativsystemet. Det kan f.eks være et read-systemkall for å lese data fra ei fil.
2. Operativsystemet bestemmer hvilken tjeneste som trengs for å utføre denne oppgaven
3. Operativsystemet lokaliserer tjenesten og gjør et kall til aktuell funksjon (som i dette eksemplet er read-funksjonen)
4. Etter at aktuell funksjon er utført av kjernen i operativsystemet returneres kontrollen tilbake til prosessen som fortsetter i brukermodus

Et viktig kjennetegn ved brukermodus og kjernemodus er følgende:

- I *brukermodus* er det brukerprosessens egne instruksjoner (de som f.eks du har skrevet) som opererer mot prosessens egne lokale data, dvs. de dataene som er under kontroll av prosessen.
- I *kjernemodus* er det operativsystemet sine egne instruksjoner som utføres, og disse instruksjonene opererer mot operativsystemets (kjernen) sine dataområder, f.eks fildeskriptorer, bufferområder etc.

Dette å skille mellom brukermodus og kjernemodus er altså svært viktig for kunne beskytte brukernes programmer og data, og også for å sikre at det er operativsystemet som hele tiden har kontrollen.

Det betyr at du som programmerer kan skrive programmer som om du hadde hele data-maskinen aleine. Et moderne operativsystem, som f.eks Linux, vil sørge for at alle prosessene som fins på maskinen får del i ressursene, f.eks CPU. Hvis f.eks din prosess krasjer vil dette ikke påvirke resten av systemet. De andre prosessene på maskinen kjører ufortrødent videre.

1.12. Fra program til i/o-enhet

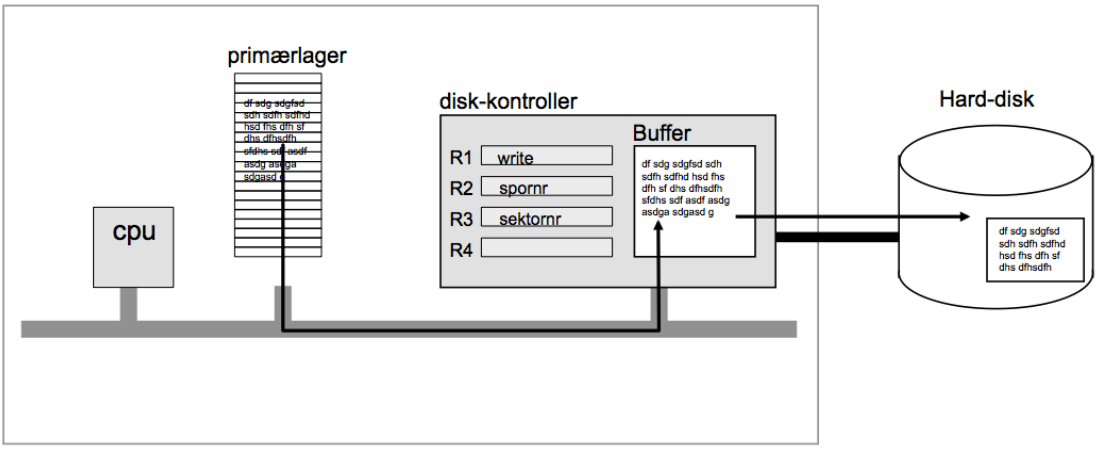
Utstyrsenheter som cpu, minne, skjerm, harddisk, nettverkskort etc. er knyttet sammen ved hjelp av busser, og all datatransport mellom enhetene går via den ene av disse bussene, nemlig databussen.

Operativsystemet har også et sett av drivere som trengs for å administrere alle enhetene som er koblet til datamaskinen, f.eks skjerm, diskett, harddisk, nettverk, skanner, høyttaler, CD-ROM-spiller etc. Til mange av disse enhetene trengs også spesielle kort eller en kontroller i maskinen for at i/o-enhetene skal fungere.

For å kommunisere med i/o-enhetene trengs driverprogram som betraktes som en del av operativsystemet. Driverprogrammet kan ikke kommunisere direkte med i/o-enheten, men kommuniserer med selve kortet for det aktuelle utstyret. Dette kortet kaller vi ofte for *kontroller*-kortet. For hard-disken heter det disk-kontrolleren, nettverkskortet heter nettverks-kontrolleren, for skjerm heter det ofte skjermkontrolleren (i stedet for grafikk-kort).

Diskkontrolleren finner du som en del av selve disken. Når du holder selve harddisken i hånda finner du kontrolleren på undersida som en rekke elektroniske komponenter. Harddisken og kontrolleren "hører" altså sammen i en pakke.

Driverprogrammene kommuniserer med i/o-enhetene ved å legge fra seg (skrive) og hente (lese) data fra registre på kontroller-kortet. Disse registrene har egne adresser og kan adresseres på vanlig måte via adresse-bussen.



I figuren over er disk-kontrolleren brukt som eksempel. I register R1 er det lagt inn koder for aktuell operasjon som skal utføres, i dette tilfellet skriving. Videre er det lagt inn parametre i de andre registrene som angir hvor på disken (spornummer og sektornummer) dataene skal skrives.

Driverprogrammet for harddisken starter alltid en slik dataoverføring til disken ved først å legge inn aktuell operasjonskode (write) og nødvendige parametre. Deretter starter driveren overføringen av data fra primærlager til et buffer i kontrolleren. Dette er i hovedsak driveren sin oppgaver. Deretter overtar selve kontrolleren overføringen av data fra bufferet i kontrolleren til selve disken. Denne siste overføringen er fullstendig styrt av elektronikken i kontrolleren, og altså ikke styrt av programvare. I figuren ovenfor er både overføringen fra primærlager til kontrollerkort, og videre til harddisk markert med pil.

Driverprogrammet kan altså ikke kontakte i/o-enhetene direkte. All kontakt med i/o-enhetene går via registre på kontroller-kortet ved at operasjonskoder og parametre legges inn i registre på kontrollerkortet. Dette gjelder for disk-kontrolleren så vel som for nettverkskontroller og

skjermkontroller. Andre aktuelle operasjonskoder for disk-kontrolleren er å lese en sektor, parkere lese/skrive-hode, finne status til siste operasjon, formatere et spor, lese diskparametre etc.

Husk også at driverprogrammet utføres på cpu-en som et helt vanlig program ved at instruksjoner fra driverprogrammet som ligger i minnet hentes til cpu-en, dekodes og utføres. Mens vanlig programmer skriver og leser data fra vanlige lagerceller, så skriver og leser driverprogrammet data til adresser (og dermed til lokasjoner) på kontrollerkortet.

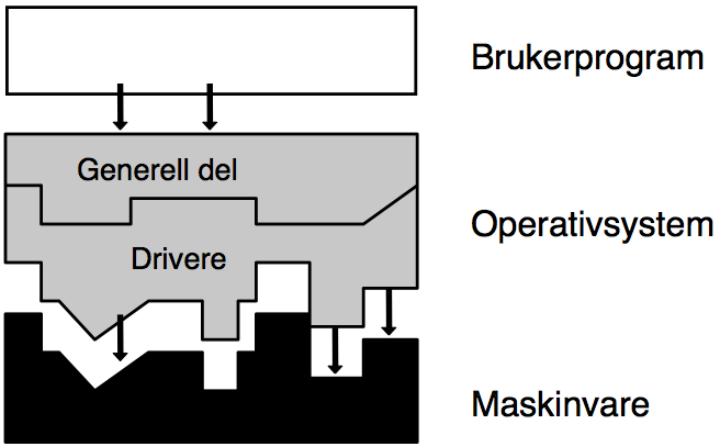
1.12.1. Driverprogrammet

Operativsystemet står til tjeneste når brukerprosessene trenger å utføre operasjoner mot mekanisk utstyr som er koblet opp mot maskinen, f.eks. skjermen, skriveren, diskettstasjonen, tastaturet. Operativsystemet gjør dette ved å benytte seg av spesielle *driverprogram* som finnes på maskinen. For hver utstyrsenhet som er tilkoblet maskinen trengs det et driverprogram.

Dette programmet er spesiallaget for å kunne operere direkte mot maskinvaren. Driverne genererer koder som trengs for å styre maskinvaren. Dersom det kobles en ny skriver til maskinen må det også installeres en ny driver samtidig. Tilsvarende hvis det kobles til lydkort og høyttaler trenger man også et driverprogram for å generere nødvendige signaler for å få lyd i høyttaleren.

Operativsystemet kan grovt deles inn i to lag som vist på figuren nedenfor. Det nederste laget nærmest maskinvaren er driverprogrammet. Dette er programvare som er skrevet spesielt for å kunne operere direkte på maskinvaren. Her er det snakk om programvare for å styre diskettstasjon, harddisk, skriver etc.

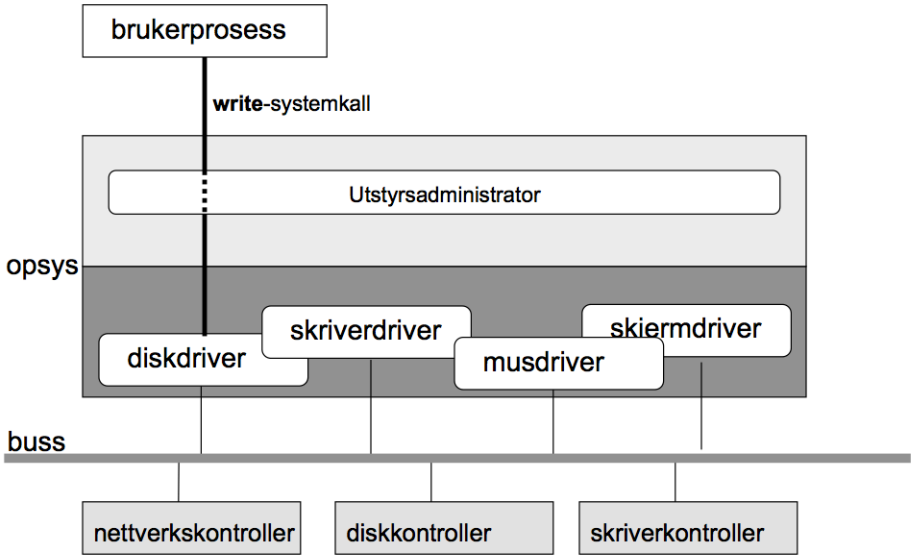
Det neste laget er den generelle delen av operativsystemet. Her finner vi ting som er uavhengig av selve maskinvaren, f.eks hvordan dataene organiseres på disken, f.eks flatt eller er det hierarkisk, adgangsrettigheter til data, buffere etc.



På nivået over operativsystemet finner vi brukerprosesser som regneark, tekstbehandler, tegneprogram etc. Alle disse prosessene har bruk for maskinvare f.eks når ei fil skal lagres på diskett, når ei fil skal skrives ut, etc. Brukerprosessene gir kommandoer til operativsystemet som sender kommandoen videre til utstyrsenheter via driveren for det aktuelle utstyret. På denne måten kan brukerprosessene bruke mange forskjellige typer utstyrsenheter.

I figuren ovenfor ser vi at det aktuelle driverprogrammet passer som ”hånd i hanske” med maskinvaren, og slik må det være fordi driverprogrammet skal “kjenne” maskinvaren så godt at alle nødvendige operasjoner mot maskinvaren skal være mulig. I figuren ovenfor kan f.eks

maskinvaren være et skjermkort/grafikkort og driveren kan være driverprogrammet for det aktuelle grafikkortet. Eller maskinvaren kan være en ny skriver av type blekkskriver, og driveren er da programmet som trengs for å åpne og lukke blekkdyser, flytte papiret fram etc.



Figuren over viser et mer komplett bilde som viser gangen fra brukerprosess til utstyrs-kontroller. En brukerprosess gjør at write-systemkall for å skrive data til harddisken. Systemkallet mottas av operativsystemet ved utstyrsadministrator som ser til at det aktuelle utstyret er ledig og eventuelt legger brukerprosessen i kø dersom utstyret er opptatt.

Selve brukerprosessen vil ikke merke noe til utstyrsadministratoren og de oppgavene den må utføre. Dette er nemlig en operativsystemfunksjon som må utføres fordi det er mange prosesser som kjører samtidig, og den enkelte prosess vil ikke være bevisst at flere andre er ute etter de samme ressursene.

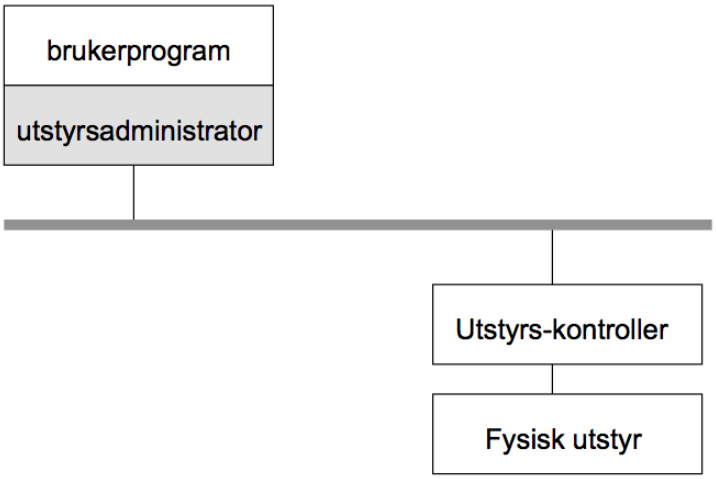
Sett fra prosessen sitt ståsted ser det ut som den er aleine på maskinen. Derfor er det vist ei linje direkte fra brukerprosessen og ned til driveren. Det er nemlig slik brukerprosessen ”opplever” situasjonen, som et kall direkte til driveren.

Drivere til forskjellig utstyr har omtrent samme grensesnitt mot brukerprosessene. Det betyr at forskjellig utstyr har det samme kallgrensesnittet, og programmere trenger derfor ikke bry seg med detaljer og forskjeller mellom forskjellig utstyr. Et write-systemkall til disk, skriver, nettverkskort etc. vil være utformet likt.

1.12.2. Oppsummert om utstyrsenheter og drivere

La oss stoppe opp litt og se nærmere på forholdet mellom brukerprogrammer og utstyrs-enheter. Figuren nedenfor viser at brukerprogrammet henvender seg til operativsystemet som via utstyrsadministratoren holder orden på alle utstyrsenhetene.

Utstyrsadministratoren gjør sine henvendelser til utstyret via driverprogrammet til enheten. Driverprogrammet opererer mot kontrolleren, og etter at kontrolleren har fått tak i dataene er det den som på egen hånd styrer all trafikk mot selve den fysiske utenheten.



Utstyrsadministrator er en del av operativsystemet og mottar henvendelser fra bruker-programmene om å få lov til å bruke det aktuelle utstyret. Utstyrsadministrator administrer enhetene ved hjelp av de såkalte *deskriptorene* hvor det er lagret data om hver ustyrsenhet. I deskriptorene finnes også bufferområder for datatransporten mellom operativsystem og fysisk utstyrsenhet.

Etter at utstyrsadministrator har akseptert forespørselen, er det driveren til det aktuelle utstyret som sørger for videre kommunikasjon med utstyret. Det er nemlig driveren som kjenner alle detaljene i ustyrets virkemåte. Driveren flytter dataene videre over til buffere i kontrolleren, og setter kommandoer i aktuelle registre hos kontrolleren, og gir til slutt kontrolleren beskjed om å overføre dataene til aktuell enhet, f.eks til disk eller til skriver.

Kontrolleren er en maskinvareenhet og kan jobbe på egen hånd mot det aktuelle utstyret etter at nødvendige kommandoer er gitt. Da kan kontroller jobbe mot utstyr samtidig som en annen prosess kjører på cpu.