KOMPENDIUM

Våren 2020

Operativsystemer

Dette er et kompendium for emnet TDT4186 holdt ved NTNU våren 2020. Kompendiet er basert på boka *Operating System Internals and Design Principles* av William Stallings (B), forelesningsnotatene (F) og løsningsforslag på eksamensoppgaver (E). Kompendiet er delt inn i seks deler, i likhet med eksamen. De seks delene er:

1. Operativsystemer generelt (kapittel 1 og 2)
2. Prosesser og tråder (kapittel 3 og 4)
3. Synkronisering av prosesser (kapittel 5 og 6)
4. Håndtering av lager (kapittel 7 og 8)
5. Tidsstyring av prosesser (kapittel 9 og 10)
6. Håndtering av IO (kapittel 11 og 12)

Del 1 – Operativsystemer generelt

Denne delen av kompendiet ser på operativsystemer generelt og inkluderer:

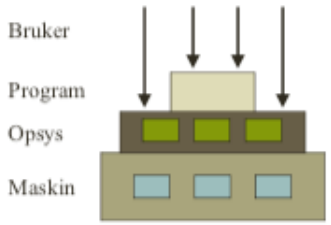
- **Kapittel 1** – Introduksjon til datasystemer
- **Kapittel 2** – Introduksjon til operativsystem

Kapittel 1 – Introduksjon til datasystemer

Et operativsystem (OS) er den grunnleggende programvaren i en datamaskin som kobler sammen maskinvaren med all annen programvare. **OS bruker hardware ressursene hos en eller flere prosessorer for å gi et sett med tjenester til systembrukerne og kontrollere sekundærminne og IO enheter**. Formålet til operativsystem er å:

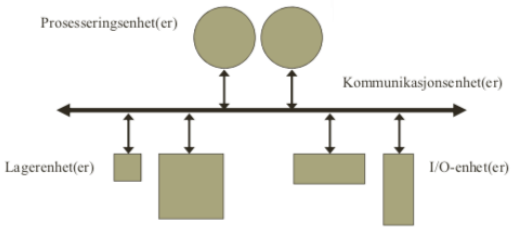
s. 10
s. 131

1. Sikre at datamaskinens tjenester for brukere er så ***enkelt*** som mulig
2. Sikre at forvaltning av ressurser hos systemet er så ***effektivt*** som mulig
3. Sikre at tjenestetilbud og ressursforvaltning kan ***utvikles*** over tid på en billig og fleksibel måte



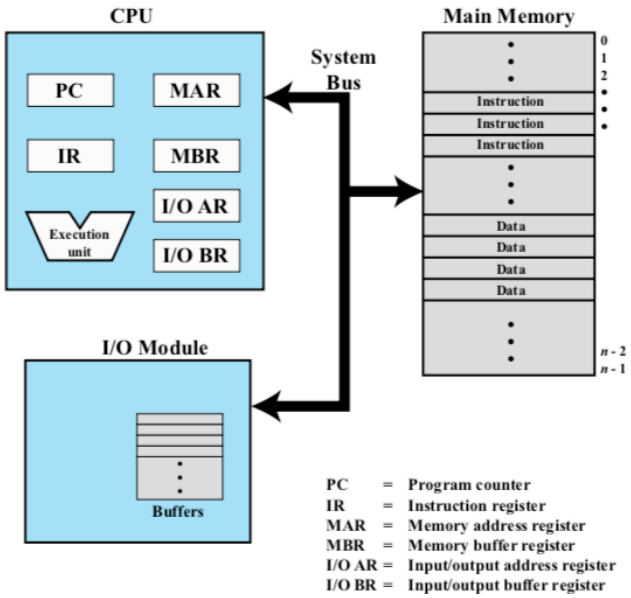
Dette er ikke lett å kombinere. Operativsystemets delprogrammer inkluderer (1) implementering, synkronisering og tidsstyring av tråder og prosesser, (2) implementering og håndtering av lager, (4) implementering og håndtering av IO og (4) sikring av datamaskinressurser. Dette blir gjerne oppnådd vha lagdelt arkitektur. For å forstå hvordan operativsystem fungerer, må man først forstå underliggende datasystem hardware.

1. 1 Grunnleggende elementer



Datamaskinen er bygd opp av noen hovedkomponenter som er koblet sammen på en bestemt måte. Disse komponentene er:

- **Prosessor (prosesseringsenhet)** = kontrollerer driften av datamaskinen og utfører prosesseringsfunksjoner på data. Kalles også CPU (Central Processing Unit).
- **Hovedminne (lagerenhet)** = lagrer data og program. Dette minnet er ofte flyktig, altså innholdet blir tapt når datamaskinen skrus av. Kalles også primærminne.
- **IO moduler (input/output enhet)** = flytter data mellom datamaskinen og dens eksterne miljø som består av en rekke enheter (eks: datamus, minnepenn, osv.).
- **Systembuss (kommunikasjonsenhet)** = gir kommunikasjon mellom prosessorer, hovedminne og IO moduler.



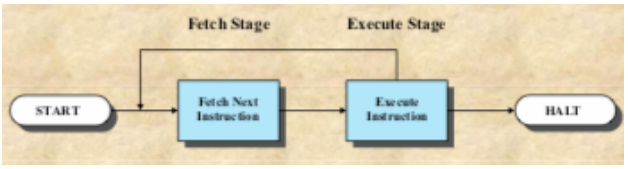
Figuren viser disse hovedkomponentene. For å utveksle data med hovedminnet, bruker prosessoren to interne register kalt **MAR (Memory Address Register)** og **MBR (Memory Buffer Register)**. MAR gir adressen i minnet som skal leses eller skrives, mens MBR inneholder dataen som skal skrives inn i minnet eller mottar dataen som leses fra minnet. På tilsvarende måte vil prosessoren utveksle data med IO moduler, ved å bruke **IO AR (Address Register)** som gir IO modulen og **IO BR (Buffer Register)** som inneholder dataen. En minnemodul består av et sett med lokasjoner som er definert av en sekvensiell nummeradresse. Hver lokasjon inneholder et bit-mønster som kan tolkes som en instruksjon eller data. En IO modul vil overføre data mellom eksterne enheter og prosessoren og minnet. Den inneholder interne buffere for midlertidig lagring av data.

1. 2 Evolusjonen hos mikroprosessor

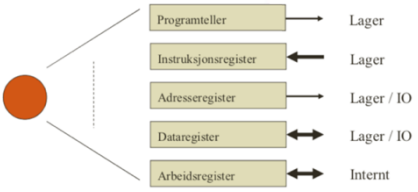
Utviklingen av **mikroprosessoren**, der en enkel chip inneholder en prosessor, har gjort at man kan utvikle bærbare og håndholdte datamaskiner. I begynnelsen var mikroprosessoren tregere enn multichip prosessoren, men pga kontinuerlig utvikling har den nå blitt mye raskere for de fleste typer utregninger. Andre viktige utviklinger er multiprosessorer (flere prosessorer per chip), GPUs (Graphical Processing Units), DSPs (Digital Signal Processor) og SoC (System on a Chip).

1. 3 Instruksjonssyklus

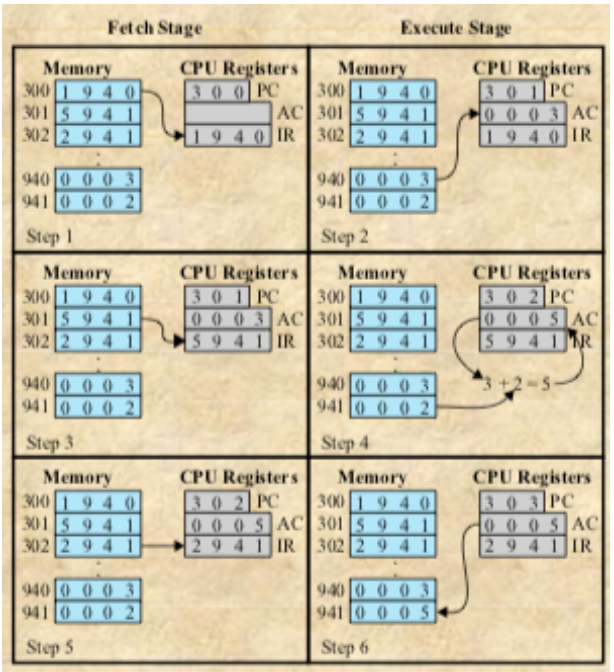
Et program som skal utføres av en prosessor, består av et sett med instruksjoner som er lagret i minnet. **Den enkleste formen for instruksjonsutførelse innebærer at prosessoren henter (*fetch*) instruksjoner fra minnet og utfører én instruksjon om gangen (dvs. ingen parallellitet).** Prosesseringen som trengs for én instruksjon, kalles **instruksjonssyklusen**.



På figuren kan vi se at den grunnleggende instruksjonssyklusen består av to steg: *fetch* og *execute*. Ved begynnelsen av syklusen vil prosessoren hente en instruksjon fra minnet. Som regel vil PC (programteller) gi adressen til neste instruksjon som skal hentes, og den økes etter hver henting. **Instruksjonen som er hentet blir lastet inn i IR (instruksjonsregister), og den består av bits som gir handlingen som prosessoren skal utføre**. Handlingene kan kategoriseres som:



- **Prosessor-minne** = data overføres fra prosessor til minnet, eller omvendt
- **Prosessor-IO** = data overføres til eller fra en perifer enhet ved å overføres mellom prosessoren og IO modulen
- **Data prosessering** = prosessoren utfører aritmetiske eller logiske operasjoner på dataen
- **Kontroll** = instruksjonen kan gi at sekvensen av oppgaver skal endres (eks: gi adresse for neste instruksjon)

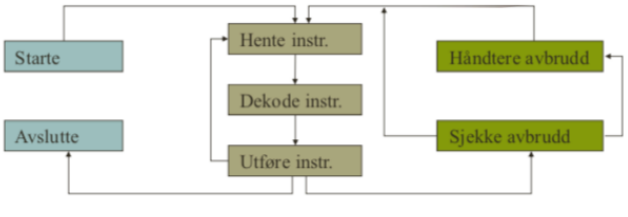


Figuren viser et eksempel på programutføring, der AC (akkumulator) er dataregisteret som lastes med dataen ved de gitte adressene. PC gir adressen til neste minnelokasjon i heksadesimale tall og instruksjonen ved denne lokasjonen lastes inn i IR. De første fire bitene (opcode) gir hvilken handling som skal utføres, mens de gjenværende 12 bitene gir en adresse. Denne prosessen innebærer bruken av MAR og MBR, men det er ikke vist på figuren. Dette programmet går ut på å summere innholdet i lokasjon 940 og 941, og lagre resultatet i 941.

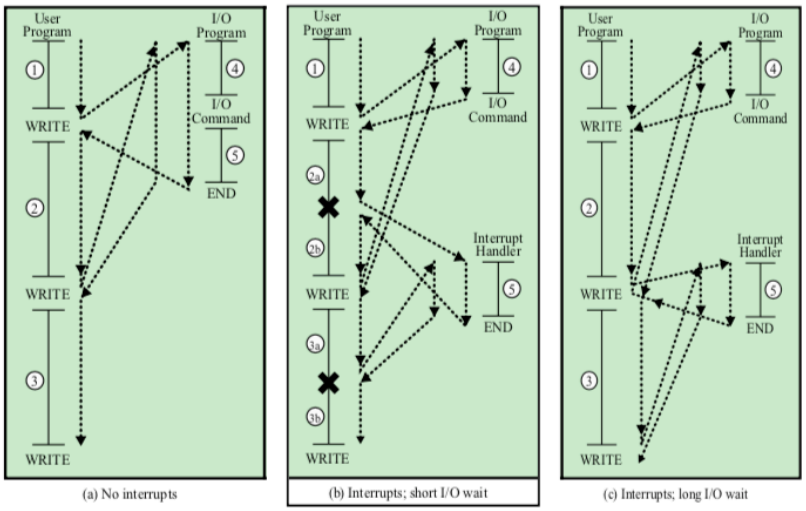
1. 4 Avbruddsmekanisme

Nesten alle datamaskiner tilbyr en avbruddsmekanisme som lar andre moduler (IO, minnet) avbryte den normale sekvenseringen til prosessoren. De vanligste typene avbrudd er:

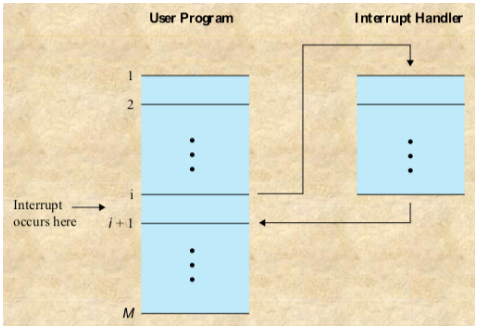
1. **Programfeil** = lages av en betingelse som oppstår ved utføring av en instruksjon, for eksempel deling på 0, referanse utenfor minnet, osv.
2. **Tidsforbruk** = lages av en timer i prosessoren som lar OS regelmessig utføre funksjoner
3. **Enhetsfunksjon (IO)** = lages av en IO kontroller for å signalisere at en prosess er ferdig eller at en feil har oppstått
4. **Maskinfeil** = lages av en feil (eks: strømbrudd)



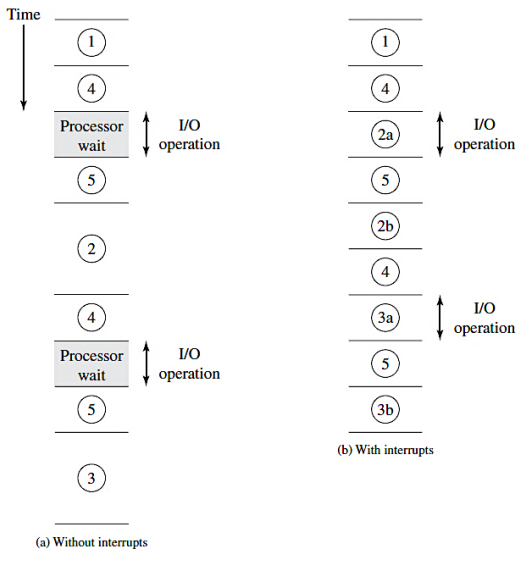
Avbrudd brukes for å oppnå en bedre utnyttelse av prosessoren (parallellitet). For eksempel blir det brukt for at prosessoren skal slippe å vente på trege IO enheter, ved at det lar prosessoren gjøre andre ting helt til IO enheten er ferdig. Dette unngår bortkastet bruk av prosessoren. Figuren viser tilfellene med og uten avbruddsmekanisme, der stiplet linje viser sekvensen som utføres. Brukerprogrammet utfører en rekke WRITE forespørsler som er flettet sammen med annen prosessering som ikke involverer IO (segment 1, 2 og 3). IO programmet består av en rekke instruksjoner som forbereder til IO operasjonen (segment 4), IO kommandoen som utfører operasjonen og en rekke instruksjonen som fullfører operasjonen (segment 5). De to typene programflyt er:



- **Ingen avbruddsmekanisme (figur a)** = prosessoren må vente på at IO enheten fullfører den forespurte operasjonen. Når prosessoren når WRITE, vil den fullføre IO programmet før den fortsetter med utføring av brukerprogrammet.
- **Avbruddsmekanisme (figur b)** = prosessoren kan utføre andre instruksjoner mens IO operasjonen utføres. Når prosessoren når WRITE, vil den kalle IO programmet. Når segment 4 har blitt utført vil prosessoren returnere til brukerprogrammet, slik at IO operasjonen blir utført samtidig som instruksjoner i brukerprogrammet. **Når den eksterne enheten er klar for å få mer data fra prosessoren vil IO modulen sende en forespørsel om avbrudd (avbruddsignal) til prosessoren som dermed vil pause utføringen av brukerprogrammet for å tjene IO enheten.** Avbruddet er markert med kryss. Utføringen av brukerprogrammet vil gjenopptas når prosesseringen av avbruddet er ferdig (se figur). Brukerprogrammet trenger ikke å inneholde noe spesiell kode for å støtte avbruddet, fordi **det er prosessoren og OS som er ansvarlig for å pause og gjenoppta programmet ved samme punkt.**

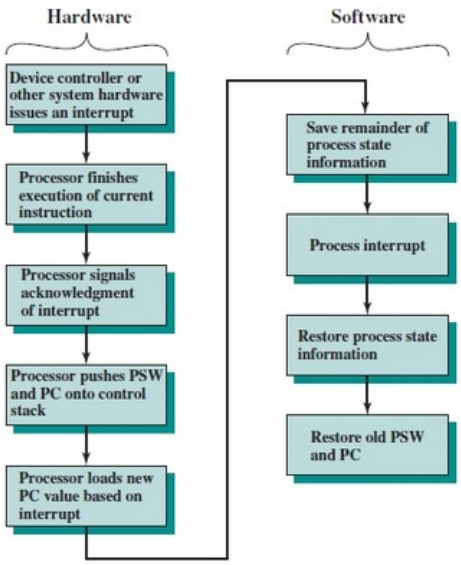


For å håndtere avbrudd vil vi legge til et avbruddssteg i instruksjonssyklusen. I dette steget vil prosessoren sjekke om den har mottatt noen avbruddsignaler. Hvis det er ingen signaler vil den hente neste instruksjon i nåværende program, mens hvis den har mottatt signaler vil prosessoren pause utføringen av nåværende program og utføre en rutine som håndterer avbruddet, ved å utføre nødvendige handlinger (eks: bestemme IO modul som har laget signal).



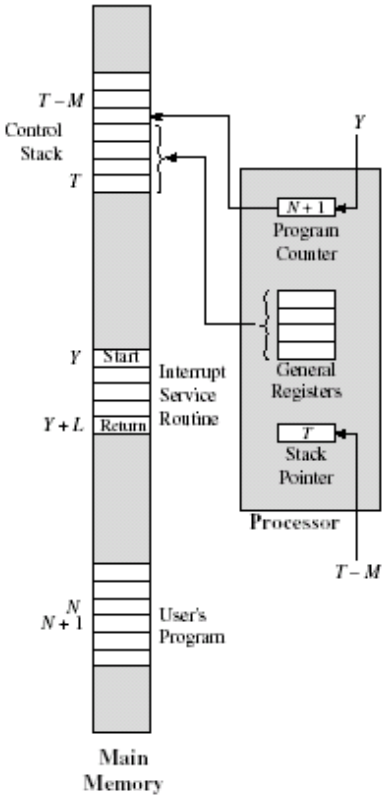
Når rutinen er fullført, vil prosessoren gjenoppta utføringen av brukerprogrammet. **Bruken av avbrudd gjør at prosessoren blir mye mer effektiv, siden den slipper å vente på IO operasjonen**. Tidsdiagrammene på figuren viser denne økningen i effektivitet. I dette tilfellet er det en kort IO ventetid, som vil si at IO operasjonen rekker å bli fullført før prosessoren når neste WRITE i brukerprogrammet. Det er likevel vanligere med lang IO ventetid, noe som er illustrert på figur c på forrige side. I dette tilfellet vil prosessoren nå det andre WRITE kallet før IO operasjonen til det første kallet er fullført, og den må derfor vente på at første IO operasjon er ferdig før neste kan starte. Selv om dette innebærer noe ventetid, vil det **fortsatt gi bedre effektivitet enn tilfellet der det er ingen avbruddsmekanisme.**

Avbruddsprosessering



Et avbruddssignal vil trigge en rekke hendelser i prosessoren (se figur). Når IO operasjonen er fullført, vil følgende skje:

1. IO enheten sender et avbruddssignal til prosessoren
2. Prosessoren fullfører utføringen av nåværende instruksjon, før den pauser utføringen av resterende brukerprogram og responderer på avbruddet
3. Prosessoren sjekker at det er et ventende avbruddssignal og sender en acknowledgement (ACK-signal) til IO enheten som dermed fjerner signalet
4. Prosessoren forbereder seg til å starte avbruddshåndtering-rutinen ved å lagre informasjonen som trengs for å gjenoppta nåværende program. Dette involverer PSW (Program Status Word) som gir informasjon om kjørende prosess og PC som gir lokasjonen til neste instruksjon som skal utføres. Disse lagres på kontroll stakken.
5. Prosessoren laster PC med lokasjonen til rutinen som skal respondere på avbruddet. Hvis OS har flere rutiner, må prosessoren bestemme hvilken som skal brukes. Når PC er lastet vil prosessoren fortsette til neste instruksjonssyklus og utfører programmet som håndterer avbruddet
6. Innholdet i prosessorregistrene er en del av det utførende programmet som ble avbrutt, så dette og annen tilstandsinformasjon blir lagret på stakken. Dermed kan registrene brukes av avbruddshåndteringen. Figuren viser et eksempel der innholdet til alle registrene og PC lagres på stakken, og stakkpekeren og PC oppdateres.
7. Avbruddshåndteringen starter prosesseringen av avbruddet, ved å undersøke statusinformasjon, sende kommandoer til IO enheten, osv.
8. Når avbruddet er ferdig håndtert, blir registrene lastet med innholdet fra stakken
9. PSW og PC verdiene blir hentet fra stakken, slik at prosessoren kan fortsette utførelsen av det tidligere avbrutte programmet

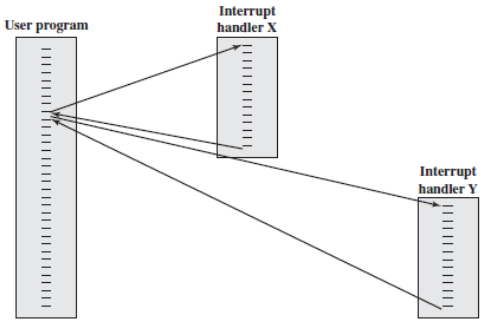


Det er viktig at prosessoren lagrer informasjon om det avbrutte programmet, fordi dette er essensielt for at det skal kunne gjenoppta utførelsen. Avbruddet er ingen rutine som kalles fra programmet og kan skje når som helst.

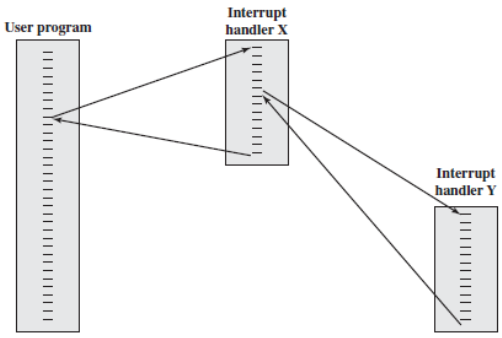
Prosessering ved multiple avbrudd

Det er mulig at flere avbrudd kan skje samtidig som et annet avbrudd blir prosessert. For eksempel kan programmet motta data fra en kommunikasjonslinje samtidig som det printer resultat. I dette tilfellet vil printeren generere et avbrudd hver gang den fullfører en print-operasjon, mens kontrolleren for kommunikasjonslinjen vil generere et avbrudd hver gang en dataenhet ankommer. Det er to ulike tilnærminger til håndtering av avbrudd:

1. **Sekvensiell avbruddsprosessering** = når et avbrudd blir prosessert, blir andre avbrudd deaktivert, ved at prosessoren ignorerer alle nye avbruddssignaler den mottar. Når håndteringen av avbruddet er fullført, vil prosessoren reaktivere avbrudd og sjekker om det har skjedd flere avbrudd. Denne tilnærmingen er enkel, siden avbruddene blir håndtert i en streng sekvensiell



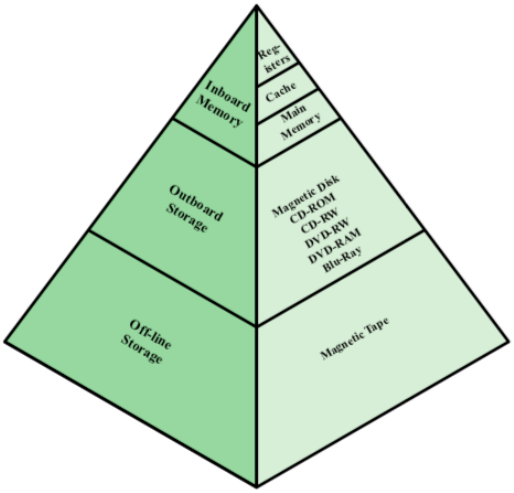
rekkefølge. Ulempen er at den tar ikke hensyn til relativ prioritet eller tidskritiske behov. For eksempel er det viktig at input fra kommunikasjonslinjen tas inn raskt for å gjøre plass til mer input (dvs. unngå tapt data som følge av full buffer).



2. **Nøstet avbruddsprosessering** = avbrudd har definert prioritet, og et avbrudd med høyere prioritet kan føre til at håndteringen av et avbrudd med lavere prioritet blir avbrutt. Et avbrudd med lavere prioritet må vente. Dersom en avbruddshåndtering blir avbrutt vil informasjon om utføringen pushes på stakken.

1.5 Minnehierarkiet

Tre hovedegenskaper ved minnet er kapasitet (størrelse), aksesseringstid (hastighet) og kostnad. Applikasjoner vil som regel benytte seg av hele kapasiteten. For å få best mulig ytelse, er det viktig at prosessoren slipper å vente på data fra minnet. Det er også viktig at kostnaden til minnet er rimelig sammenlignet med de andre komponentene i systemet.

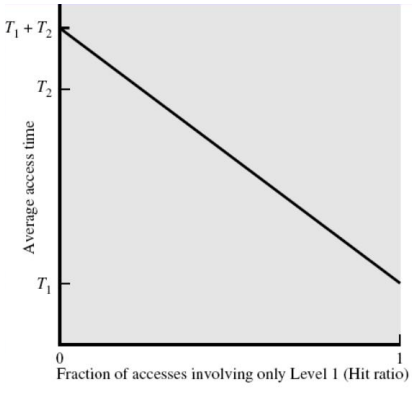


Det vil være **en balanse mellom kapasitet, aksesseringstid og kostnad**, og man har følgende forhold:

- **Mindre aksesseringstid, gir større kostnad per bit**
- **Større kapasitet, gir mindre kostnad per bit**
- **Større kapasitet, gir større aksesseringstid**

Det er ønsket med teknologier som gir større kapasitet, siden man trenger kapasitet og det gir lavere kostnad per bit. Samtidig er det nødvendig med lavere kapasitet, større kostnad og raskere aksesseringstid for å tilfredsstille ytelseskravene. Løsningen på dette dilemmaet er å bruke et **minnehierarki** (se figur). **Nedover i hierarkiet vil kostnaden reduseres, kapasiteten økes, aksesseringstiden økes og mengde aksess reduseres**. Mindre, dyrere og raskere minner blir kombinert med større, billigere og tregere minner. **Nøkkelen til suksess for denne organiseringen er at det er en lavere frekvens av aksess til de nedre nivåene**.

Eksempel – utregning av gjennomsnittlig aksesseringstid



Vi ser på eksempelet der prosessoren har aksess til to nivåer med minne. Nivå 1 inneholder 1000 bytes og har en aksesstid på $0.1\mu s$, mens nivå 2 inneholder 100 000 bytes og har aksesstid på $1\mu s$. Bytes i nivå 1 kan hentes direkte, mens bytes i nivå 2 må først overføres til nivå 1. Vi ser bort i fra tiden som trengs for å bestemme hvilket nivå byten befinner seg i. Som vi kan se på grafen vil **gjennomsnittlig aksesseringstid reduseres med økt *hit ratio* (h), som er fraksjonen av alle aksesser som er i det raskeste minnet (dvs. cache).** T_1 er aksesseringstiden til nivå 1, mens T_2 er aksesseringstiden til nivå 2. Dersom det er mye aksess til nivå 1, vil total gjennomsnittlig aksesseringstid være nærmere T_1. For eksempel hvis 95% av aksessene er i nivå 1 ($h = 0.95$), vil gjennomsnittlig aksesseringstid for en byte være:

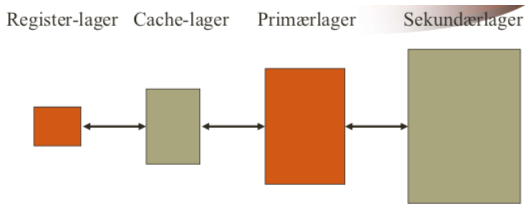
$$0.95 * 0.1\mu s + 0.05 * (0.1\mu s + 1\mu s) = 0.15\mu s$$

Resultatet er altså nærmere aksesseringstiden til det raskeste minnet. Dette vil kun være tilfellet dersom minnehierarkiet har de gitte egenskapene, altså at nedover hierarkiet vil kostnaden og aksessfrekvensen reduseres, mens kapasiteten og aksesseringstiden økes.

Referanselokasjon – reduser aksessfrekvens

Grunnlaget for at aksessfrekvensen til nedre nivå reduseres, er prinsippet kalt referanselokasjon, som gir at minnereferanser har en tendens til å gjentas. Dette skyldes at program ofte inneholder iterative løkker, delrutiner og arrays som bruker samme instruksjoner og data. **Over korte perioder vil prosessoren arbeide med samme klynge av minnereferanser,** noe som gjør det mulig å ordne dataen slik at det blir færre referanser til lavere nivå i minnehierarkiet. For eksempel ved to nivåer, kan nivå 2 inneholde alle instruksjoner og data, mens nivå 1 blir midlertidig fylt med nåværende klynger. Av og til må en klynge i nivå 1 byttes ut, men de fleste referansene vil gå til nivå 1.

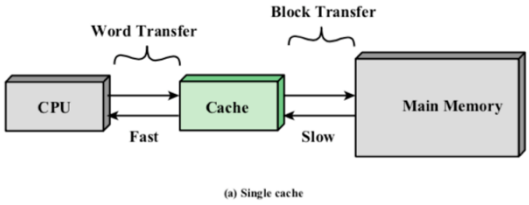
Dette prinsippet gjelder også når det er flere nivåer. Det raskeste, minste og mest kostbare minnet er **registrene** til prosessoren. Deretter følger **cache** som er raskere og mindre enn **hovedminnet** (primærminnet), som ofte har en unik adresse for hver lokasjon. Disse tre minnene er som regel flyktige, som vil si at dataen blir tapt når strømmen slås av. Et eksternt, ikke-flyktig minne, kalt **sekundærminne**, brukes for å lagre program og datafiler. Det er også vanlig at virtuelt minne i form av en harddisk, brukes for å utvide hovedminnet (kapittel 8). Programvare kan brukes for å legge til flere nivåer, for eksempel kan deler av hovedminnet brukes som en buffer som midlertidig holder data som skal skrives til disken (kapittel 11).



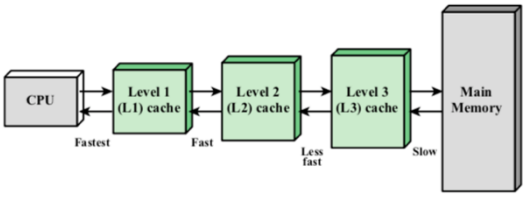
1.6 Cache-minne

I løpet av en instruksjonssyklus vil prosessoren aksessere minnet for å hente instruksjonen og eventuelle operander og/eller lagre resultat. **Hastigheten til utføringen av instruksjoner er begrenset av syklustiden til minnet, siden prosessoren er mye raskere enn hovedminnet**. Ideelt sett burde hovedminnet vært bygd på samme måte som prosessorregistrene, men dette er svært kostbart. **Løsningen er heller å bruke cache som et mindre og raskere minne mellom prosessoren og hovedminnet**.

Cache prinsippet

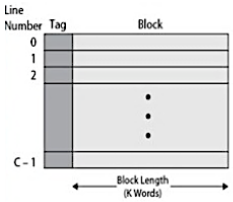


(a) Single cache

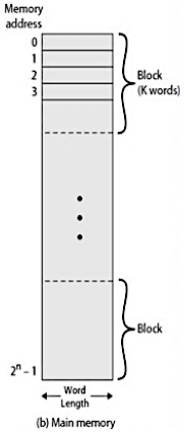


(b) Three-level cache organization

Målet med cache er å oppnå et stort og billig minne med rask aksesseringstid. På figur a kan vi se at konseptet er basert på et mindre og raskere cache-minne som plasseres mellom CPU og det relativt store og trege hovedminnet. **Cache inneholder en kopi av deler av hovedminnet. Når prosessoren forsøker å lese en byte eller et ord fra minnet, vil det først utføres en sjekk for å bestemme om dette ligger i cache.** Hvis det er tilfellet blir innholdet levert til prosessoren, mens hvis det ikke er tilfellet vil en blokk med et fast antall bytes leses fra hovedminnet inn i cache, før byten/ordet så leveres til prosessoren. **Referanselokasjon gjør at mange av de fremtidige minnereferansene sannsynligvis er til blokken som ble hentet inn i cache**. Det er også vanlig å bruke flere nivåer med cache, som figur b illustrerer. I dette tilfellet vil størrelsen og aksesseringstiden øke jo nærmere man kommer hovedminnet.



(a) Cache



(b) Main memory

Figuren til venstre viser strukturen til cache og hovedminnet. Hovedminnet består av opptil 2^n adresserbare ord, der hvert ord har en unik n-bit adresse. For eksempel hvis adressen består av 8 bits, vil minnet kunne ha $2^8 = 256$ ord. Hovedminnet deles inn i M antall blokker, som hver består av K antall ord (dvs. $M = 2^n/K$). Cache deles inn i C antall luker, som hver består av K antall ord. **Antall luker er signifikant mindre enn antall blokker (dvs. $C \ll M$),**

og hver luke er en kopi av en blokk fra hovedminnet. Hvis prosessoren ber om et ord som ikke befinner seg i cache, må blokken til dette ordet leses inn i en av lukene. En luke vil ikke være reservert for en bestemt blokk, så derfor har hver luke en tagg som identifiserer hvilken blokk som er lagret ved nåværende tidspunkt. Denne taggen er ofte de første bitene av adressen, slik at den refererer til alle adressene som begynner med den sekvensen av bits. For eksempel hvis adressene består av 6 bits, vil taggen 01 referere til 010000, 010001, etc.

Cache design

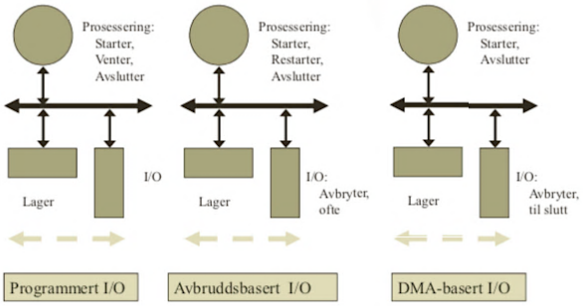
Merk: ved design av cache må man ta viktige avgjørelse innenfor lagring, slik som når data skal erstattes, hvor mye data hver blokk skal inneholde og hvilken data som skal erstattes.

Når man designer en cache må man se på følgende egenskaper:

- **Cache størrelse** – hvis cachen er relativt liten, kan den ha signifikant påvirkning på ytelsen til datamaskinen
- **Blokk størrelse** – store blokker betyr at mer nyttig data hentes inn i cache, slik at hit ratioen øker som følge av referanselokasjon. **Dersom blokkene blir for store, vil hit ratioen reduseres fordi sannsynligheten for å bruke den nye dataen blir mindre enn sannsynligheten for å gjenbruke data som ble flyttet ut av cache for å lage plass.**
- **Mapping funksjon** – bestemmer hvor den nye blokken skal plasseres i cache, noe som påvirker hvor lang tid det tar å bestemme om en bestemt blokk befinner seg i cachen.
- **Erstatningsalgoritme** – velger hvilken blokk som skal erstattes når en ny blokk leses inn i en full cache. Ideelt sett vil vi erstatte blokken som det er minst sannsynlig blir brukt i nær fremtid (eks: Least-Recently-Used blokk).
- **Skriveregler** – hvis cache inneholder en blokk der innholdet har blitt endret, må denne blokken skrives til hovedminnet før den erstattes. Skriveregler bestemmer når dette skal utføres (eks: ved hver endring, ved erstatning, osv.).
- **Antall cache nivåer** – jo nærmere prosessoren, desto mindre og raskere er cache

1.7 IO varianter og direkte minneaksess (DMA)

Når prosessoren utfører et program og møter en instruksjon som er relatert til IO, vil den utføre instruksjonen ved å sende en kommando til den passende IO modulen. Deretter er det tre ulike teknikker som brukes av IO operasjoner:



1. **Programmert IO** – IO modulen utfører de forespurte handlingene og plasserer de resulterende bitene i IO status register, men gir ingen beskjed til prosessoren (dvs. ingen avbrudd eller forstyrrelser). Prosessoren må periodisk undersøke statusen til IO modulen for å oppdage når IO instruksjonen er fullført. Dette gjør at prosessoren må vente lenge, og den gjentatte undersøkelsen reduserer ytelsen til systemet.
2. **Avbruddsbaserte IO** – IO modulen utfører de forespurte handlingene og avbryter prosessoren når den er klar til å utveksle data. Prosessoren utfører overføringen av data, før den gjenopptar oppgavene den holdt på med før avbruddet. **Dette er mer effektivt en programmert IO, siden prosessoren kan utføre andre oppgaver mens den venter**. Ulempen er fortsatt at prosessoren trengs for å overføre data mellom IO modulen og minnet (begrenser IO overføringshastigheten), og all overføring av data må skje via prosessoren (bortkastet bruk av prosessor).
3. **DMA-basert IO** – brukes når store mengder data skal flyttes mellom IO modulen og minnet (dvs. lese eller skrive mye data). Prosessoren sender kommandoen til en DMA modul som kan være en separat modul på systembussen eller en del av IO modulen. IO operasjonen blir håndtert av DMA modulen, slik at prosessoren kan fokusere på andre oppgaver. DMA modulen kan overføre data direkte fra minnet og avbryter prosessoren når overføringen er ferdig, slik at prosessoren kun er involvert i starten og slutten av

Merk: avbruddsbasert- og DMA-basert IO er spesielt viktig når IO enheten er treg for å unngå lange ventetider.

prosessen. DMA modulen bruker systembussen for å overføre data, noe som kan føre til at prosessoren må vente på tilgang til bussen (merk: ikke det samme som avbrudd). **DMA er likevel mye mer effektivt når mye data skal overføres.**

1.8 Multiprosessor- og multikjernesystem

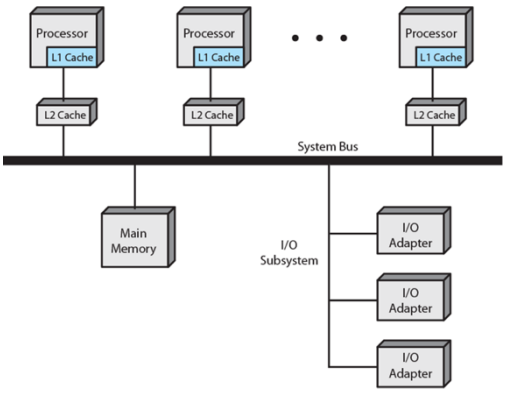
Merk: løsningsforslag fra eksamen inkluderer også **distribuerte system**, som er en samling av datamaskiner forbundet i et nettverk. Disse kommuniserer ved å sende meldinger

Parallellisme har blitt mer normalt ettersom datavitenskapen har utviklet seg og maskinvare har blitt billigere, og det brukes for å øke ytelsen og påliteligheten (tryggheten). To eksempler på tilnærminger som brukes for å oppnå parallellisme er multiprosessor- og multikjernesystem.

Multiprosessorsystem

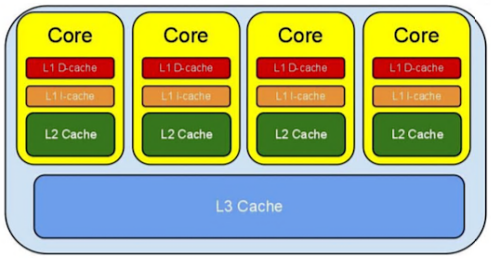
Et multiprosessorsystem har flere prosessorer til å håndtere prosesseringslasten. Et multiprosessor OS gir samme funksjon som et multiprogrammert OS, med ekstra funksjonalitet for å imøtekomme flere prosesser. Fordeler med multiprosessorsystem er at det gir økt pålitelighet, økt ytelse og reell parallellitet.

OBS: deler av arbeidet må kunne gjøres parallelt for at multiprosessor skal gi større ytelse enn uniprosessor!



En symmetrisk multiprosessor (SMP) er et system med to eller flere lignende prosessorer, som deler hovedminnet og IO enheter og er koblet sammen via en buss eller andre interne koblinger (dvs. har lik aksesseringstid). Hver prosessor har som regel to dedikerte nivåer med cache (se figur). **Det kalles symmetrisk fordi alle prosessorer kan utføre samme funksjoner**. Systemet kontrolleres av et integrert operativsystem, som legger til rette for interaksjon mellom prosessorene ved ulike nivå. Dette gjør at SMP tillater et høyt nivå med samarbeid mellom prosesser. Sammenlignet med systemer som har én prosessor, vil fordeler med SMP være økt ytelse dersom deler av arbeidet kan utføres parallelt og pålitelighet (hvis en prosessor feiler vil ikke hele systemet stoppe). Det lar også salget av datamaskiner skaleres, siden prisen kan justeres basert på antall prosessorer.

Multikjernesystem



Et multikjernesystem har flere prosesseringskjerner i hver prosessor. Hver kjerne består som regel av alle komponentene til en uavhengig prosessor, slik som register, ALU, kontrollenheten, pipeline maskinvare og øverste nivå med cache.
Multikjernesystem tilbyr altså caching på flere nivåer (se figur). Fordeler er at ytelsen kan økes uten å endre klokkefrekvensen eller plassere cache nærmere prosessoren. Ulemper er at det blir mer komplisert for OS å fordele ressurser og koordinere hendelser, siden flere ting kjører samtidig (kan løses med synkronisering).

Kategorisering av operativsystemer

Operativsystemer kan ofte deles inn i ulike kategorier, basert på størrelse (gitt av pris) og bruk (gitt av bruksområde):

- Stormaskin – 5M $ (Bankapplikasjoner)
- Tjenermaskin – 5K $ (Nettverkstjenester)
- Personlig datamaskin – 500 $ (Bord / Fang)
- Mobil datamaskin – 50 $ (Brett / Mobil)
- Mikrokontroller – 5 $ (Vaskemaskin / Vekkeklokke)
- Kastbar kontroller – 0,5 $ (Hilse-kort / Id-kode)

OBS: Kapittel 1 har appendix om ytelsesegenskaper ved bruk av cache og lokalitetstyper (s. 62-67).

Kapittel 2 – Introduksjon til operativsystem

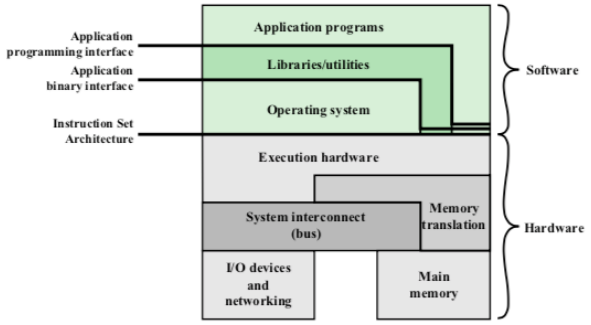
Operativsystem har utviklet seg fra å være primitive batch systemer til bli sofistikerte systemer som kan utføre flere oppgaver samtidig og støtte flere samtidige brukere. Operativsystemer har et stort spenn av bruksområder, som inkluderer personlige datamaskiner, stormaskiner, mobile enheter, osv.

2.1 Operativsystemer – mål og funksjon

Et operativsystem (OS) er et dataprogram som kontrollerer utføringen av applikasjonsprogram, og det fungerer som et grensesnitt mellom applikasjoner og maskinvaren til datamaskinen. Et OS har tre mål:

1. **Bekvemmelig for brukere (brukerfokus)** – et OS gjør at det blir mer bekvemmelig for brukere å bruke datamaskinen
2. **Effektivt for systemet (systemfokus)** – et OS lar datasystemet bruke ressursene på en effektiv måte
3. **Evne til å utvikle seg (utviklingsfokus)** – et OS bør være laget på en måte som tillater utvikling, testing og introduksjon av nye systemfunksjoner på en billig og fleksibel måte som ikke forstyrrer tjenesten

Vi ser nærmere på disse.



Brukerfokus – bekvemmelig for brukere

Det brukersentrerte målet ved et operativsystem er å gjøre det mer bekvemmelig å bruke datamaskinen, ved at det fungerer som et grensesnitt mellom brukeren og maskinen. Maskinvaren og programvaren som brukes av en applikasjon kan deles inn i flere lag, som vist på figuren. Brukeren er som regel ikke opptatt av detaljene ved maskinvaren og ser heller systemet som et sett med applikasjoner. Disse applikasjonene er uttrykt i et programmeringsspråk og er laget av utviklere. Det vil være svært utfordrende å utvikle en applikasjon som består av et sett med maskininstrukser som fullstendig kontrollerer maskinvaren. For at dette skal bli lettere blir det gitt et sett med systemprogrammer som implementerer funksjoner som ofte blir brukt og hjelper til i programskapelse, kontroll av IO enheter og management av filer. Når applikasjonen kjører vil den bruke disse systemene for å utføre bestemte funksjoner. **Operativsystemet er en samling av de viktigste systemprogrammene. OS vil dermed skjule detaljene om maskinvare fra utviklerne og gir et grensesnitt for å enkelt bruke systemet**. Det fungerer som en mellommann som gjør det enklere for utviklere og applikasjoner å bruke tjenestene til maskinvaren.

Noen av områdene der operativsystemet gir tjenester er:

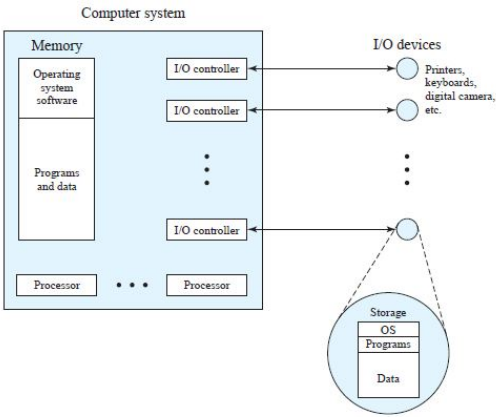
- **Programutvikling** – OS gir et sett med tjenester, for eksempel editorer og debugger for å hjelpe utviklere når de lager applikasjoner. Disse gis som regel i egne programmer som ikke er en del av kjernen til OS, men leveres med OS.
- **Programutføring** – OS håndterer planlegging av steg som må gjennomføres for å utføre et program, slik som lasting av instruksjoner og data inn i hovedminnet.
- **Tilgang til IO enheter** – hver IO enhet krever en spesiell kombinasjon av instruksjoner og kontrollsignaler. OS skjuler disse detaljene og gir et generelt grensesnitt som lar utviklere få tilgang til disse enhetene vha. enkel lesing og skriving.
- **Kontrollert tilgang til filer** – hvis systemet har flere brukere, kan OS gi beskyttelsesmekanismer som kontrollerer tilgangen til filer.
- **Kontrollert tilgang til systemet** – OS kontrollerer tilgangen til systemet og spesifikke systemressurser. Ressurser og data må beskyttes fra uautoriserte brukere.

- **Deteksjon og respons til feil** – når datasystemet kjører kan det oppstå flere ulike typer feil i maskinvaren eller programvaren (eks: minnefeil og deling på 0), og OS må gi en respons som fikser feilen med minst mulig innvirkning på kjørende applikasjoner (eks: avslutte program som forårsaker feil eller forsøke på nytt).
- **Regnskap** – OS vil samle statistikk om bruken og følge med på ytelsesparametere, slik som responstid. Dette brukes for å forutse behovet for fremtidige forbedringer og for å finjustere systemet for å forbedre ytelsen.

Operativsystemet vil altså hjelpe til med å lage og utføre program, beskytte datasett, sikre lovlig og riktig bruk av systemet og oppnå kort responstid for den enkelte brukeren.

Systemfokus – effektivt for systemet

Det systemsentrerte målet ved et operativsystem er å fungere som en ressursmanager som kontrollerer bruken av datamaskinens ressurser, slik som minne og prosesseringstid. Denne kontrollen er spesiell, siden OS selv bruker ressursene som følger av at det er et program som må utføres av prosessoren (dvs. ikke ekstern til det som kontrolleres). I tillegg vil OS ofte frasi seg kontrollen og er avhengig av prosessoren for å få tilbake kontrollen. OS består altså av instruksjoner som må utføres av prosessoren, og mens dette blir utført vil OS bestemme hvordan prosesseringstiden skal fordeles og hvilke ressurser som er tilgjengelige. For at prosessoren skal handle basert på avgjørelsene til OS, må den slutte å utføre OS programmet, slik at den kan utføre andre program. OS vil derfor frasi seg kontrollen for at prosessoren skal gjøre annet nyttig arbeid, og etterhvert vil OS få kontrollen på nytt slik at den kan forberede prosessoren til å gjøre neste arbeidsdel.



Figuren viser noen av ressursene som kontrolleres av OS. For å sikre effektiv bruk av ressursene til systemet, vil OS:

- **Sikre utnyttelsen av prosesseringsenhetene** – OS bestemmer hvor mye prosesseringstid som kan tildeles utføringen av et bestemt program.
- **Sikre og koordinere utnyttelsen av minnet** – deler av OS befinner seg i hovedminnet, og sammen med minne-management vil OS kontrollere tildelingen av minnet.
- **Koordinere utnyttelsen av IO-enhetene** – OS bestemmer når en IO enhet kan brukes av programmet og kontrollerer aksessen.

Utviklingsfokus – evne til å utvikle seg

Det evolusjonssentrerte målet ved et operativsystemet er at det skal være lett å utvikle systemet. Årsaker til at OS må kunne utvikle seg over tid, er:

- **Oppgradering av maskinvare** – når maskinvaren endres kan det kreve mer sofistikert støtte i OS (eks: paging maskinvare krever OS som utnytter paging)
- **Utvikling av nye tjenester** – OS må kunne utvides med nye tjenester for å tilfredsstille behovet hos brukere eller systemmanagere (eks: legge til nye verktøy dersom man oppdager at eksisterende verktøy gir dårlig ytelse)
- **Fiksing** – alle OS vil ha feil som oppdages over tiden, og disse må fikses.

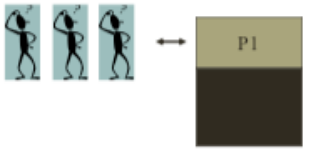
For at det skal bli enklere å utvikle OS må det designes på bestemte måter, for eksempel bør operativsystemet ha en fleksibel og modulær oppbygning med god dokumentasjon.

2.2 Historie – evolusjonen hos operativsystemer

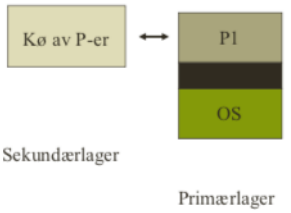
For å forstå kravene og egenskapene hos moderne operativsystem, er det nyttig å se på hvordan OS har utviklet seg over årene.

- **Serieprosessering (urtype)** – de første datamaskinene på slutten av 1940-tallet hadde ingen operativsystem, slik at programmene måtte interagere direkte med

maskinvaren. Programmer i maskinkode ble gitt til inputenheten og output ble gitt av printeren, dersom programmet ble suksessfullt utført. Det tok lang tid å sette opp programmet før det kunne kjøre, planlegging av bruk krevde at man skrev seg på fysiske lister og det var dårlig utnyttelse av minnet og CPU.

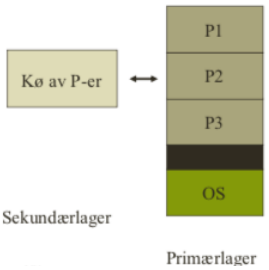


- **Uniprogrammerte batch systemer** – batch OS bruker programvare kalt monitor for å kontrollere sekvensen av hendelser. Brukere sender inn program som plasseres i en batch som gis til monitoren. Monitoren vil derfor ha en kø av program som utføres av prosessoren, og når et program er fullført vil monitoren automatisk laste neste program. Resultatet sendes til outputenhet. Batch OS innebærer at monitoren forbruker noe minne og prosesseringstid, men det vil forbedre ytelsen. Det introduserer også behovet for **lagerbeskyttelse** (brukerprogram kan ikke endre minnet der monitoren er lokalisert), **timer** (hindre at ett program monopoliserer systemet), **privilegerte instruksjoner** (maskinnivå instruksjoner som kun kan utføres av monitor) og avbrudd (større fleksibilitet). Dette oppnås vha. modeller (brukermodell, kernel modell).

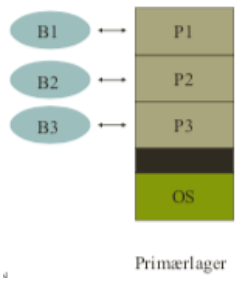


Merk: **multiprogrammert OS** betyr at flere prosesser håndteres samtidig vha avbruddsmekanismer. Det er ikke reell parallellitet, men gir en form for multiprosessering

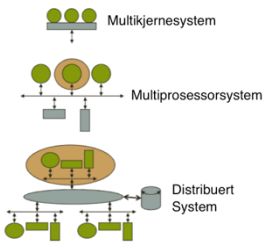
- **Multiprogrammerte batch systemer** – ved uniprogrammering vil prosessoren bruke mye tid på å vente på tregt minne og trege IO enheter. Ved multiprogrammering blir minnet fylt med flere programmer, og prosessoren vil begynne på et nytt program når forrige program ble avsluttet eller forrige program venter på IO. Dersom programmene ikke bruker samme ressurser, vil dette øke ytelsen. Det introduserer også behovet for **IO avbrudd eller DMA**, **minnemanagement** (programmene som skal være klare til å kjøre må holdes i hovedminnet) og **scheduling** (prosessor må kunne avgjøre hvilket program som skal kjøres)



- **Timesharing systemer** – multiprogrammering kan også bruke for å håndtere flere interaktive brukere vha time sharing, der prosessortid deles blant brukerne. I et timesharing system vil flere brukere samtidig aksessere systemet via terminaler og OS vil sammenflette utføringen av hvert brukerprogram. **Multiprogrammering brukes for å minimere responstiden til hver bruker** (merk: ikke for å maksimere bruk av prosessor). En systemklokke vil generere avbrudd med et bestemt intervall (eks: 0.2s), og ved hvert avbrudd vil OS få tilbake kontrollen og tildele prosessoren til en ny bruker (kalles *time slicing*). Prosessoren vil altså begynne på et nytt program når forrige program avslutter, venter på IO eller tidskvanten utløper. Det introduserte også behovet for **tidskontroll** og **aksesskontroll**



I nyere tid har det blitt utviklet multikjernesystem, multiprosessorsystem og distribuerte system som er basert på flere av de samme konseptene som disse systemene. Disse introduserer behovet for flere mekanismer vi skal se på senere.

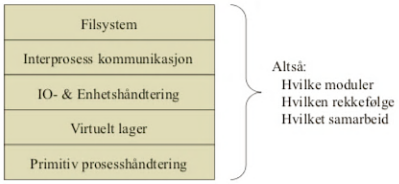


2.3 Viktige fremskritt

Vi ser på fem fremskritt som har hatt stor påvirkning på utviklingen av operativsystem.

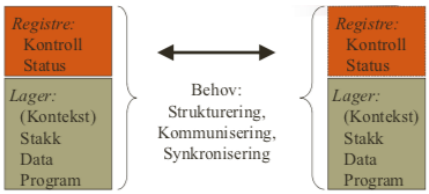
Modulisert og lagdelt systemstruktur (F)

OS har blitt utviklet med en lagdelt struktur, der hvert lag består av en modul som kun vil bruke funksjonalitet som gis av lavere lag. Et lag vil ikke kjenne til implementasjonen av andre lag, fordi eksistensen av bestemte datastrukturer, operasjoner og hardware blir skjult. Dette gjør at lag kan oppdateres uavhengig av andre lag, så lenge output-funksjonaliteten er lik.



Prosessbegrepet

En prosess er et program som utføres eller venter på å utføres (finnes flere definisjoner), og prosesser er grunnleggende for strukturen til operativsystem. Koordineringen av avbrudd, multiprogrammering, timesharing og sanntid-transaksjonsprosessering viste seg å være svært komplisert. Disse introduserte subtile feil som var vanskelig å detektere og finne årsaken til, siden betingelsene som gjorde at feilen oppstod var vanskelig å gjenskape (eks: deadlock).

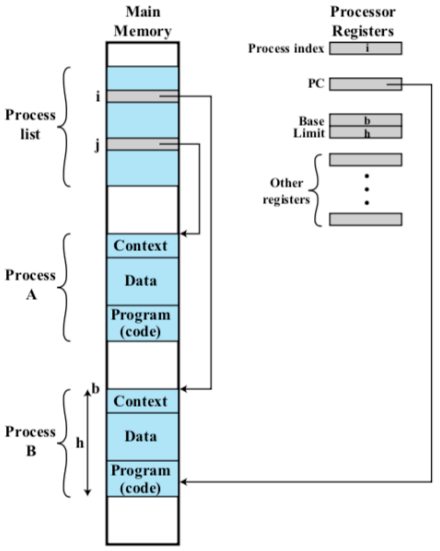


Prosessbegrepet ble introdusert for å gi en systematisk måte å overvåke og kontrollere de ulike programmene som utføres av prosessoren. En prosess består av:

1. **Et utførbart program**
2. **Assosiert data som trengs av programmet**
3. **Utføringskonteksten (prosesstilstand) til programmet**

Prosesstilstanden inneholder all informasjon som OS trenger for å kontrollere prosessen og prosessoren trenger for å riktig utføre prosessen. Dette inkluderer prosessprioritet, info om prosessen venter på IO, innhold hos PC, osv.

Figuren viser hvordan prosesser kan håndteres. Hovedminnet inneholder to prosesser A og B, og begge er registrert i en **prosess-liste som lages og opprettholdes av OS**. Hver enhet i listen inneholder en peker til prosessen og kanskje deler av eller hele konteksten. Prosessindeks-register gir indeksen i prosess-listen til nåværende prosess som utføres av prosessoren og programtelleren peker til neste instruksjon i denne prosessen. Base og limit registrene definerer hvor prosessen ligger i minnet (sørger for at PC slutter ved enden av prosessen). På figuren ser vi at prosess B utføres. Hvis prosess B blir avbrutt vil innholdet i alle registrene lagres i utføringskonteksten, slik at hvis OS senere bytter tilbake til prosess B, trenger kun innholdet å lastes tilbake i registrene og utføringen av prosess B kan gjenopptas. Dette viser at en prosess vil enten utføres eller venter på å utføres. Konteksten tillater utviklingen av teknikker som sikrer koordinering og samarbeid blant prosesser. Man kan legge til nye egenskaper til OS ved å utvide konteksten med informasjon som støtter egenskapene.

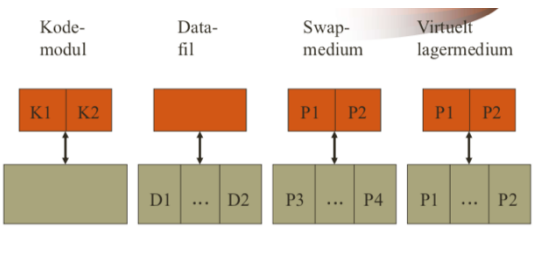


Lagerhåndtering (*memory management*)

Følgende egenskaper ved lagring bør dekkes av OS:

1. **Prosessisolering** – OS må hindre at uavhengige prosesser kan forstyrre lagringen til hverandre, både lagring av data og instruksjoner.
2. **Automatisk tildeling og håndtering** – program bør dynamisk flyttes i minnehierarkiet etter behov og tildelingen bør være transparent for utviklere. Dette gjør at utvikler slipper å tenke på begrensninger for minnet, for dette håndteres av OS.
3. **Støtte av modulær programmering** – utviklere bør kunne definere programmoduler og dynamisk lage, slette og endre størrelsen hos moduler.
4. **Beskyttelse og aksesskontroll** – OS må sørge for autorisert aksess og kontrollert deling av minne mellom program.
5. **Langsiktig lagring** – OS må la applikasjonsprogram lagre informasjon over lengre perioder på en ikke-flyktig måte.

For å dekke disse kravene vil OS bruke ulike teknikker i lagerhierarkiet, slik som datafil-systemet og virtuelt minne. Filsystemet implementerer en langsiktig lagring, der informasjon lagres i navngitte objekter kalt filer. Virtuelt minne brukes for at det skal se ut som om maskinen har mer minne enn det den egentlig har. Dette innebærer bruk av blant annet paging, som vi ser nærmere på i kapittel 7 og 8. OS kan bruke virtuelt minne for å oppnå



prosessisolering, ved at hver prosess får et unikt, ikke-overlappende virtuelt minne. Deling av minne kan oppnås ved at deler av virtuelle minner overlapper. Files holdes i langsiktig lagring, men kan kopieres inn i det virtuelle minnet for å manipuleres av program.

Beskyttelse og -sikkerhet

Mer bruk av timesharing system og nettverk har ført til et større behov for beskyttelse av informasjon. Det vil variere mellom organisasjoner hva truslene innebærer, men det finnes generelle verktøy som kan bygges inn i OS for å støtte en rekke mekanismer for beskyttelse og sikkerhet. **OS vil fokusere mest på kontroll av aksessen til datasystem og informasjon som er lagret i disse**. Arbeidet med sikkerhet og beskyttelse kan deles inn i:

1. **Tilgjengelighet** – systemet må beskyttes mot forstyrrelser
2. **Konfidensialitet** – data må beskyttes fra uautorisert lesing
3. **Dataintegritet** – data må beskyttes fra uautorisert modifisering
4. **Autentisering** – identiteten til brukere må verifiseres og datameldinger må valideres

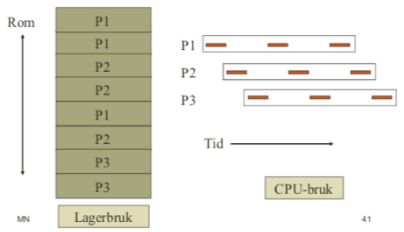
Det vil være en utfordrende balanse mellom deling og beskyttelse.

Ressurskontroll og tidsstyring

OS har ansvar for å kontrollere de tilgjengelige ressursene og planlegge/styre hvordan disse brukes av aktive prosesser. Resursen kan være minne, IO enhet eller prosessor. Tildeling av ressurser og tidsstyring må ta hensyn til følgende faktorer:

- **Rettferdighet (*fairness*)** – prosesser som konkurrerer om bruken av en bestemt ressurs bør ofte få omtrent lik og rettferdig aksess til ressursen.
- **Forskjellig respons** – OS bør samtidig forsøke å tildele ressurser og planlegge slik at det totale settet med krav blir oppfylt, noe som kan innebære at OS må forskjellsbehandle program med ulike krav. Beslutningene bør tas dynamisk.
- **Effektivitet** – OS bør forsøke å maksimere gjennomstrømmingen, minimere responstid og ved timesharing håndtere så mange brukere som mulig. Disse kravene er i konflikt, så det vil være nødvendig å finne en balanse.

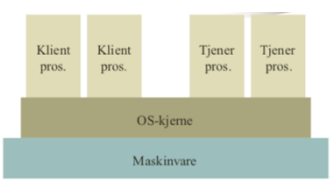
Det er viktig at OS måler aktivitetene til systemet for å følge med på ytelsen og justere faktorer deretter. OS vil opprettholde flere køer, der hver kø er en liste av prosesser som venter på en ressurs. Short-term køen består av prosesser som er i hovedminnet og er klar til å kjøre med en gang prosessoren blir ledig. Short-term scheduler vil velge hvilken prosess som skal utføres, og dette kan være basert på prioritetsnivåer, tid, osv. Long-term køen består av nye prosesser som venter på å bruke prosessoren, og disse flyttes etterhvert til short-term køen av OS. Dette vil innebære at deler av hovedminnet tildeles prosessen, så det er viktig at OS er forsiktig så minnet ikke blir overbelastet. Prosesser som venter på å få tilgang til en IO enhet blir plassert i IO køen hos enheten. Dersom køen inneholder flere prosesser, må OS bestemme hvilken som skal få tilgang først. Det finnes flere ulike typer algoritmer for å oppnå dette.



2.4 Moderne operativsystem

I løpet av de siste årene har det blitt introdusert nye designelementer som har ført til store endringer i egenskapene til operativsystem. Nye utviklinger innenfor hardware (eks: multiprosessorsystem), nye applikasjoner (eks: multimedia) og nye sikkerhetstrusler har hatt stor påvirkning på OS design. **Moderne operativsystemer bruker ofte følgende designelement:**

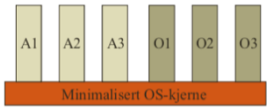
- **Mikrokjerne** – tidligere har operativsystem brukt en stor monolittisk kjerne (*kernel*) som har håndtert all OS funksjonaliteten, slik som scheduling, filsystem, osv. **En mikrokjernearkitektur tildeler kun essensielle funksjoner til kjernen**, inkludert adresserom-håndtering, interprosess kommunikasjon (IPC) og



Merk: dersom det er snakk om **trygghet** innenfor moderne OS, vil dette innebære hvorvidt tjenestene gir riktige resultater. Dette vil ofte gå på bekostning av effektiviteten (se eksamen 2019K)

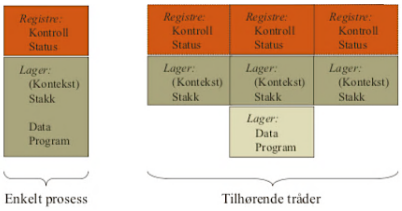
Merk: løsere kobling betyr at enhetene samhandler, men man kan endre en komponent uten å må gjøre endringer i den andre = endringsdyktig

grunnleggende tidsstyring (*scheduling*). Andre tjenester gis av prosesser/servere som kjører i brukermodus og behandles som andre applikasjoner av mikrokjernen. Dette gjør at kjerne og servere kan utvikles hver for seg, så det gir fleksibilitet og enklere implementasjon. Det er også lettere å tilpasse distribuerte system og objektorientert design. Bruk av mikrokjerne gir mindre effektivitet, men øker sikkerheten/tryggheten, siden mindre kode kjører i kjernemodus og mindre programvare får aksess til hele lageret og hele instruksjonssettet.



Via egne systemprosesser – Mikrokjernebasert

- **Tråder** – moderne OS bruker ofte *multithreading* som er en teknikk der en prosess som utfører en applikasjon deles inn i tråder som kan utføres samtidig. En tråd er en komponent av en prosess og utføres sekvensielt, mens en prosess er en samling av en eller flere tråder og korresponderer til et program som utføres (mer i kapittel 3 og 4). Dette er nyttig for applikasjoner som utfører flere uavhengige oppgaver som ikke trenger å kjøres serielt.



Merk: multithreading og SMP er to ulike konsepter. Multithreading vil også være nyttig for uniprosessor system, og SMP er nyttig for ikke-tråd prosesser. Men, de komplimenterer hverandre og kan brukes effektivt sammen

- **Symmetrisk multiprosessering** – hardware arkitektur som består av flere lignende prosessorer og OS som utnytter dette. OS og SMP vil gjennomføre en tidsstyring der alle prosessorer utfører prosesser eller tråder. Hvis arbeid kan gjøres parallelt vil multiprosessor gi større ytelse enn uniprosessor (flere prosesser/tråder kan utføres samtidig). Det kan også øke tilgjengelighet (feil i en prosessor vil ikke hindre hele systemet), inkrementell vekst (øke ytelse ved å legge til en prosessor) og skalering (tilby flere produkter med ulik pris og ytelse). Dette er potensialer og ikke garantier (krever riktig implementering for å utnytte parallellisme)

OBS: deler av arbeidet må kunne gjøres parallelt for at multiprosessor skal gi større ytelse enn uniprosessor!

- **Distribuerte operativsystem** – brukes for å gi illusjonen om at en samling av separate datamaskiner med egne minner og IO moduler, har et enkelt minne og enhetlig aksess til for eksempel det distribuerte filsystemet. Disse ligger noe etter uniprosessor og SMP operativsystem.
- **Objektorientert design** – gjør det mulig å legge til modulære utvidelser til en liten kjerne, og ved OS nivå vil objektbasert struktur gi programmere mulighet til å skreddersy OS uten å forstyrre systemintegriteten. Det gjør det også enklere tilpasse programvaren til nye maskinvaretilbud og programvarebehov, og det blir lettere å utvikle distribuerte verktøy og fullstendig distribuerte OS.

2.5 Feiltoleranse (s. 95-97)

Feiltoleranse er evnen systemet eller komponenten har til å fortsette normal drift selv om hardware eller software inneholder feil. Dette vil som regel innebære en form for redundans (overflødighet). Feiltoleranse vil øke påliteligheten til systemet, men det vil også som regel øke kostnaden (økonomisk eller ytelse). Det bør derfor tilpasses etter hvor kritisk komponenten er. Delkapittelet ser videre på fundamentale konsepter, definisjon og kategorisering av feil, typer redundans og OS mekanismer som støtter feiltoleranse.

2.6 OS design for multiprosessor- og multikjernesystem (s. 98-100)

Multiprosessor- og multikjernesystem har bestemte utfordringer ved OS design:

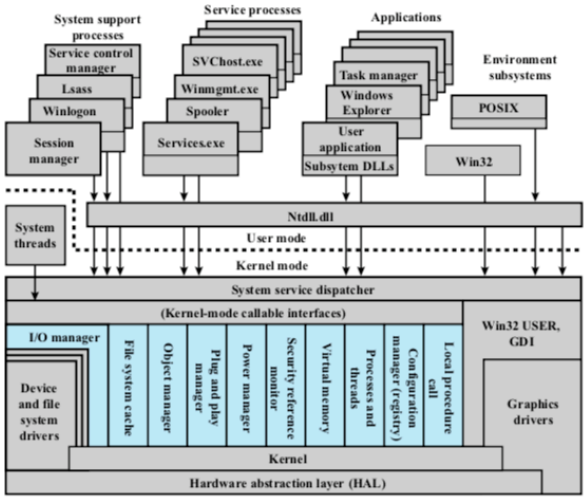
- **Multiprosessorsystem (SMP)** – OS designet blir mer komplisert siden det må håndtere deling av ressurser og koordinering av handlinger. Det må gi alle funksjonaliteter hos et multiprogrammering system i tillegg til egenskaper som legger til rette for flere prosessorer. Utfordringer er innenfor prosessynkronisering og samtidig utføring (kap. 5 og 6), lagerhåndtering (kap 7 og 8) og tidsstyring (kap 9 og 10).
- **Multikjernesystem** – slike system er under en pågående utvikling, så det er enda ikke sikkert hvilke fordeler som kan oppnås. OS designet må oppfølge samme krav som multiprosessorsystem, og i tillegg må det effektivt utnytte multikjerne-prosessorkraften

og kontrollere de betydelige ressursene på chipen. Det er altså en utfordring hvordan man best skal utnytte parallellismen ved arbeidslasten, noe som kan oppnås vha. sentralt støtte (GCD) og virtuelle maskiner (VM).

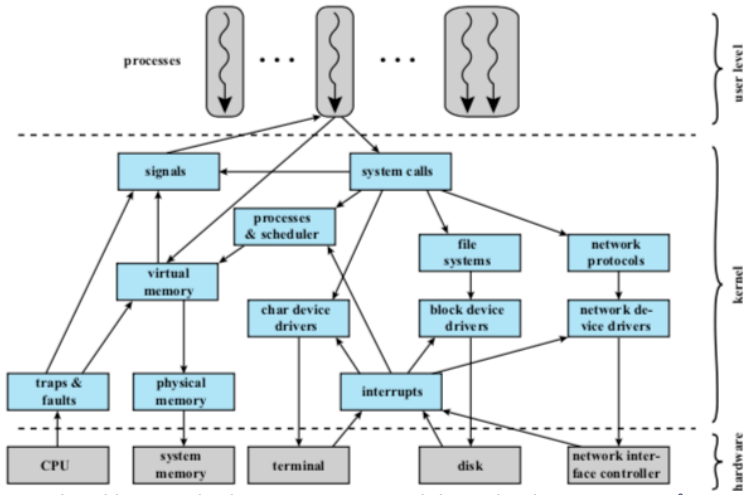
2.7-2.11 Typer operativsystem (s. 101-126)

Kompendiet fokuserer på de generelle konseptene og går derfor ikke inn i detaljer på de ulike typene operativsystem. Boka fokuserer på følgende OS:

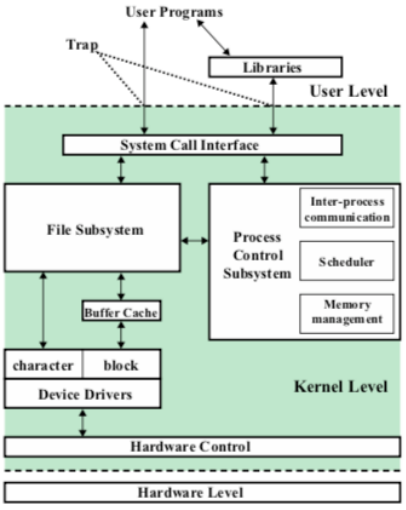
- **Microsoft Windows** (s. 101-107) – høyt modulær arkitektur, der applikasjonsorientert programvare separeres fra kjerne OS programvare (mikrokjerne). Hver systemfunksjon håndteres av kun en komponent i OS og viktig systemdata kan kun aksesseres gjennom passende funksjon. Enhver modul kan fjernes, oppgraderes eller erstattes uten at hele systemet må skrives om. Det støtter tråder og SMP (s. 105) og bruker flere konsepter fra objektorientert design (s. 106).
- **Tradisjonelle og moderne Unix system** (s. 108-112) – modulær arkitektur med en liten kjerne som gir funksjoner og tjenester som trengs av et antall OS prosesser/servere som kan implementeres på ulike måter (s. 110).
- **Linux** (s. 113-118) – bruker ikke mikrokjerne tilnærming, men oppnår flere av fordelene ved å bruke en modulær arkitektur. Linux er strukturert som en samling av moduler, som er relativt uavhengige blokker (s. 114). Modulen vil ikke gjennomføre en egen prosess eller tråd, men utføres i ved å linkes til kjernen (*kernel*)
- **Android** (s. 118-126) – en fullstendig programvare stack (ikke bare OS), som inkluderer en modifisert versjon av Linux kjernen, middleware og noen applikasjoner. Det består av flere lag som er tilpasset mobile omgivelser (s. 124-126)



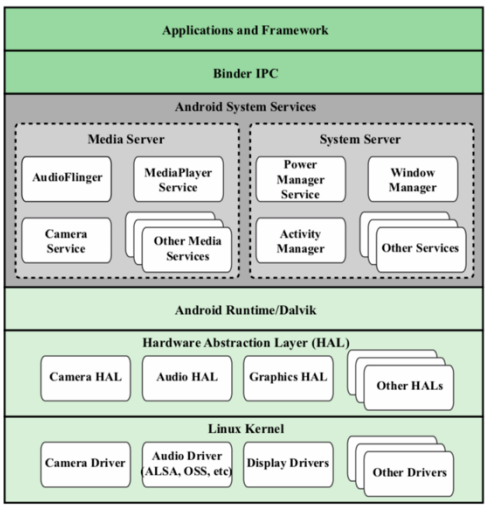
Microsoft Windows har en modulær arkitektur med mikrokjerne, som gir fleksibilitet og lett implementasjon



Linux har ikke en mikrokjerne, men en modulær arkitektur som oppnår tilnærmede fordeler. Kjernemodulene utføres på prosessoren



Tradisjonell Unix arkitektur som er designet for å kjøre på en prosessor og støtter ikke samtidig utføring



Android har lagdelt systemarkitektur som er en software stack som inkluderer applikasjoner, middleware og OS

Del 2 – Prosesser og tråder

Denne delen av kompendiet ser på prosesser og tråder, og det inkluderer:

- **Kapittel 3** – Prosesser
- **Kapittel 4** – Tråder

Merk: **multiprogrammert OS** betyr at flere prosesser håndteres samtidig vha avbruddsmekanismer. Prosessor kan kjøre en annen prosess dersom nåværende prosess må vente på for eksempel IO modul. Det er ikke reell parallellitet, slik multiprosessering er.

Kapittel 3 – Prosesser

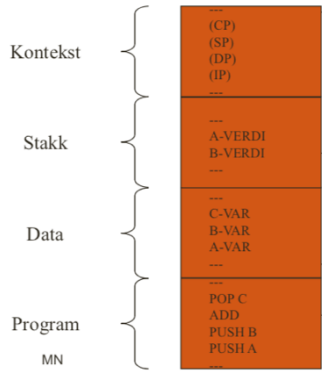
Alle multiprogram operativsystem er bygd rundt prosesskonseptet. De fleste kravene som må oppfylles av OS kan uttrykkes mht. prosesser:

- OS må flette sammen utføringen av flere prosesser for å maksimere bruken av prosessor og gi fornuftig responstid
- OS må tildele ressurser til prosesser samtidig som deadlock unngås
- OS må støtte interprosess kommunikasjon og prosesser som lages av brukere

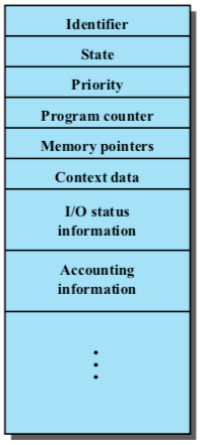
3.1 Hva er en prosess?

OBS: relasjonen mellom program og prosess er ikke en-til-en. Prosessen kan involvere flere utførende program og ulike prosesser kan inneholde identiske program, s. 165

En prosess representerer et program under utførelse, og den gis som en datastruktur som inneholder områder for program, data, stakk og kontekst (figur viser prosess for A + B som lagres i C). Det finnes flere definisjoner, for eksempel at det er en entitet som tildeles og utføres av prosessoren. Når programmet utføres vil prosessoren kunne unikt karakteriseres vha:



- **Identifikator** – unik ID som skiller prosessen fra andre prosesser
- **Tilstand** – hvis prosessen utføres vil den være i *running state*
- **Prioritet** – relativ prioriteringsnivå
- **Program counter** – adressen til neste instruksjon i programmet som skal utføres
- **Minnepekere** – pekere til programmet og dataen som er assosiert med denne prosessen og minneblokker som deles med andre prosesser
- **Kontekstdata** – data som er i registrene til prosessoren under utføring av prosessen
- **IO status** – inneholder gjenstående IO requests, tildelte IO enheter, liste av filer som brukes av prosessen, osv.
- **Accounting informasjon** – kan gi brukt prosessortid, tidsgrenser, osv.



Denne informasjonen lagres i en **prosesskontrollblokk** som lages og håndteres av OS (merk: kalles **prosess-kontekst** i forelesning). Det er denne blokken som gjør at en kjørende prosess kan avbrytes og gjenopptas ved et senere tidspunkt som om avbrytelsen aldri tok sted. **Det er altså prosesskontrollblokken som lar OS tilby multiprosessering.** Når prosessen avbrytes vil nåværende verdi av PC og prosessorregistrene lagres i prosesskontrollblokken, og tilstanden endres til *blocked* eller *ready*. Dermed kan OS plassere en annen prosess i *running state*. PC og register hos denne prosessen lastes inn i prosessorregistrene og utføringen kan begynne.

Hvorfor trengs prosesskonseptet?

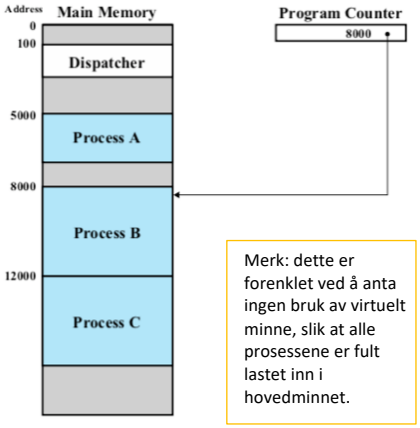
En prosess representerer et program eller en applikasjon som er under utførelse, og prosesskonseptet er en god måte å tillate og håndtere kjøring av flere programmer eller applikasjoner samtidig. Prosesser gir altså en mer effektiv og sikker utføring i multiprogrammerte system. **En oppgave som skal løses på en datamaskin kan deles opp i flere mindre, delvis uavhengige underoppgaver, og man tilordner gjerne en prosess til hver underoppgave for å få en effektiv utførelse av den samlede oppgaven**. Prosesser gir effektiv

utnyttelse av flere prosessnivåer og riktig synkronisering og kommunikasjon innen avanserte applikasjoner. Andre grunner til å bruke prosessorer for å representere utføringen av applikasjoner/program, er at:

1. Ressurser skal bli tilgjengelig for flere applikasjoner
2. Prosessoren kan svitsje mellom flere applikasjoner, slik at alle ser ut til å ha progresjon
3. Prosessoren og IO enheter kan effektivt brukes

3.2 Prosesstilstander

For at et program skal utføres blir det laget en prosess for programmet som består av et sett med instruksjoner. Prosessoren vil utføre instruksjonen som pekes mot av program counter og tar egentlig ikke hensyn til hvilket program denne instruksjonen tilhører. Utføring av en prosess innebærer at instruksjonene i prosessen blir utført. **Sekvensen av prosessinstruksjoner som utføres kalles sporet (*trace*) hos prosessen, og oppførselen til prosessoren kan beskrives av hvordan den sammenfletter flere spor.** Figuren viser et eksempel der minnet inneholder tre prosesser og et dispatcher program (del av OS), som svitsjer prosessoren mellom prosesser.



Merk: dette er forenklet ved å anta ingen bruk av virtuelt minne, slik at alle prosessene er fult lastet inn i hovedminnet.

(a) Trace of Process A	(b) Trace of Process B	(c) Trace of Process C
5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011

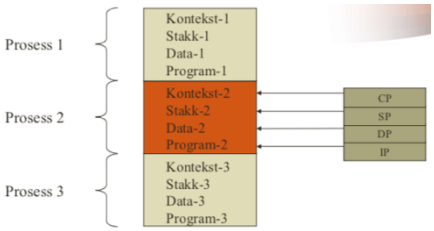
5000 = Starting address of program of Process A
8000 = Starting address of program of Process B
12000 = Starting address of program of Process C

Figuren til venstre viser spor hos tidlig utføring av hver prosess, der de 12 første instruksjonene er gitt for A og C og den fjerde instruksjonen til B utløser en IO operasjon som gjør at prosessoren må vente.

1	5000	27	12004
2	5001	28	12005
3	5002		Timeout
4	5003	29	100
5	5004	30	101
6	5005	31	102
	Timeout	32	103
7	100	33	104
8	101	34	105
9	102	35	5006
10	103	36	5007
11	104	37	5008
12	105	38	5009
13	8000	39	5010
14	8001	40	5011
15	8002		Timeout
16	8003	41	100
	I/O Request	42	101
17	100	43	102
18	101	44	103
19	102	45	104
20	103	46	105
21	104	47	12006
22	105	48	12007
23	12000	49	12008
24	12001	50	12009
25	12002	51	12010
26	12003	52	12011
			Timeout

100 = Starting address of dispatcher program

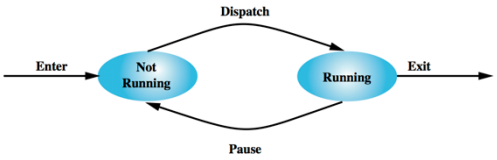
Figuren til høyre viser hvordan prosessoren sammenfletter sporene hos prosess A, B og C. De blå områdene representerer koden som utføres av dispatcher programmet. Vi antar at OS kun lar en prosess utføres i seks instruksjonssykluser, og etter dette vil den bli avbrutt (hindrer monopolisering av prosessortid). Prosessoren vil begynne å utføre de seks første instruksjonene i prosess A, før den når timeout og prosess A blir avbrutt. Dispatcher vil gi kontrollen til prosess B. Etter utføring av fire instruksjoner vil prosess B vente på en IO operasjon, så prosessoren vil slutte utføringen av prosess B og begynner utføring av prosess C. Når den når timeout vil prosessoren bytte til prosess A. Når den når timeout vil prosess B fortsatt vente på IO operasjon, så den prosessoren bytter til prosess C.



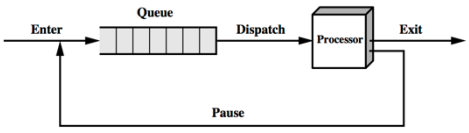
Dette eksempelet illustrerer at minnet kan inneholde tre kjørbare prosesser, men i uniprosessor system vil det være kun én kjørende prosess om gangen. Registrene til prosessoren vil inneholde verdiene til denne prosessen. Prosesskontroll blokken inneholder informasjonen som tillater avbrudd og gjenoppretting av prosesser.

To-tilstand prosessmodell

Hovedansvaret til operativsystemet er å kontrollere utføringen av prosesser, noe som inkluderer å bestemme sammenflettingen av utføringen og tildelingen av ressurser til prosesser (dvs. prosessor, lager og IO). Første steg i OS design er derfor å beskrive ønsket



(a) State transition diagram



(b) Queuing diagram

oppførsel hos prosesser. Den enkleste modellen innebærer at en prosess vil enten utføres av en prosessor eller ikke. **I to-tilstand prosessmodellen vil prosessen være i en av to tilstander: kjørbar eller kjørende (se figur).** Når OS lager ny prosess vil den lage en prosesskontrollblokk og legge til prosessen i systemet (*enter*) i kjørbar tilstand (*not running*). Når kjørende prosess blir avbrutt vil dispatcher delen av OS flytte kjørende prosessen til kjørbar-tilstand og en av prosessene fra kjørbar-tilstand flyttes til kjørende. Prosesskontrollblokken gjør at prosessen er representert på en måte som lar OS holde styr over den. I denne modellen blir prosessene i kjørbar-tilstand plassert i en FIFO kø (ulik implementasjon, eks: pekere til kontrollblokk, linked-list av prosesser, osv.). Dette kalles en **round-robin** metode, siden hver prosess får en bestemt mengde utføringstid før den returneres til køen. Hvis prosessen er fullført eller avsluttet blir den fjernet fra systemet (*exit*).

Prosesskapning og -terminering

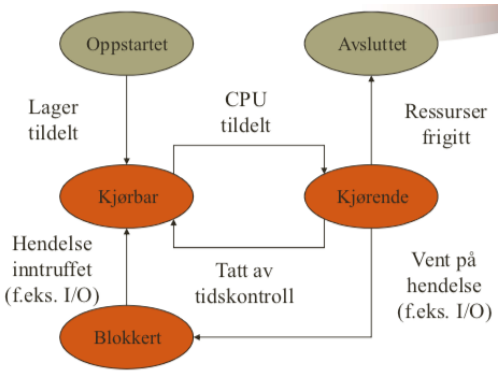
To viktige deler av prosessmodellen er:

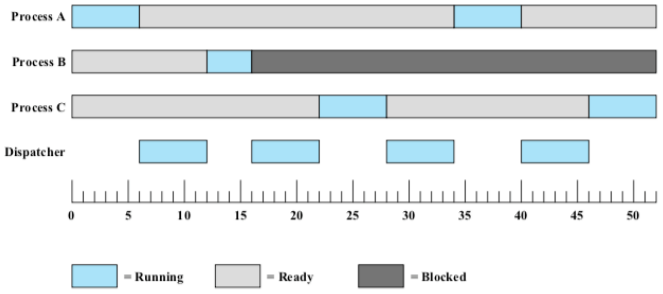
- **Prosesskapning (*creation*)** – når en ny prosess skal legges til, vil OS lage datastrukturene som brukes for å håndtere prosessen og tildele adresserom i hovedminnet for prosessen. Fire vanlige årsaker til prosesskapning er:
  - **Ny batch jobb** – i batch omgivelser vil OS lage en prosess i respons til innlevering av en jobb (eks: innsatt disk).
  - **Interaktiv pålogging** – i interaktive omgivelser vil OS lage en prosess i respons til at en ny bruker forsøker å logge inn
  - **OS ønsker å utføre tjeneste** – OS kan lage en prosess som utfører en funksjon på vegne av et brukerprogram, slik at bruker slipper å vente (eks: prosess som kontrollerer printing)
  - **Skapt av eksisterende prosess (*spawning*)** – en eksisterende prosess kan danne nye prosesser for å utnytte modularitet eller parallellitet. Eksisterende prosess kalles da parent prosessen.
- **Prosessterminering** – en prosess må ha en måte å indikere at den er fullført (eks: Halt instruksjon i batch jobber, avlogging i interaktive applikasjoner, osv.), slik at den kan sende request om terminering til OS. Vanlige årsaker til prosessterminering er:
  - **Normal fullføring** – prosessen sender terminering-request til OS som indikerer at den har fullført utføringen
  - **Time-out** – prosessen har kjørt lengre enn den gitte tidsgrensen
  - **Utilgjengelig minne** – prosessen krever mer minne enn det som er tilgjengelig
  - **Grensebrudd** – prosessen forsøker å aksessere minnelokasjoner som den ikke lov til å aksessere
  - **Beskyttelsesfeil** – prosessen utfører feil handlinger på en ressurs, slik som uautorisert aksess eller skriving av read-only fil
  - **IO feil** – en feil oppstår ved input eller output
  - osv. (s. 139)

Fem-tilstand prosessmodell (basismodell)

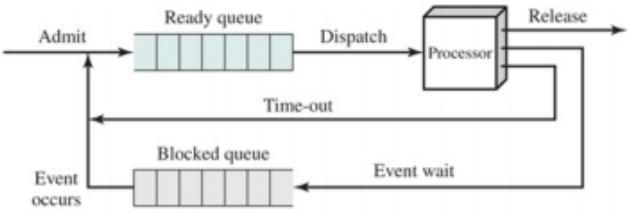
To-tilstand prosessmodellen er effektiv dersom alle prosesser er klare til å utføres, men **som regel vil noen prosesser være klare til å kjøre, mens andre er blokkerte fordi de venter på en hendelse (eks: IO operasjon). For å håndtere dette kan man dele Kjørbar-tilstanden inn i to tilstander: Blokkert og Kjørbar**. **Fem-tilstand prosessmodellen** bruker disse i tillegg til Oppstartet- og Avsluttet-tilstand, som er nyttige for prosesshåndtering. Oppstartet-tilstand inneholder prosesser som nettopp har blitt skapt, men ikke lagt til Kjørbar-tilstanden

(prosesskontrollblokk har blitt laget, men prosess har ikke blitt overført til hovedminnet, s. 140). Avsluttet-tilstand inneholder prosesser som har blitt fjernet fra Kjørende-tilstand fordi den er fullført eller avsluttes (ressurser frigjøres, men noe informasjon blir midlertidig bevart for å evt. brukes av andre program, s. 141). Figuren viser fem-tilstand prosessmodellen (se s. 142 for mer beskrivelse av overgangene). Modellen kan også inkludere en overgang fra Kjørbar til Avsluttet (terminering ved parent-prosesser og etterfølgere) og fra Blokkert til Avsluttet (prosess terminerer når IO operasjon er ferdig).





Figuren til venstre viser overgangene for eksemplet med prosess A, B og C. Legg merke til at Dispatcher programmet vil være prosessen som er i Kjørende-tilstand når det utføres for å bytte mellom de andre prosessene.



Figuren til høyre viser hvordan modellen kan implementeres med to køer, en for Kjørbar prosesser og en for Blokkert prosesser. Dersom det ikke er noen prioritering, kan dette gjøres med FIFO. Det kan være nyttig å ha flere Blokkert køer, en for hver IO enhet, slik at OS slipper å skanne hele køen for å finne alle prosesser som venter på en bestemt IO enhet. Det kan også være nyttig med flere Kjørbar køer hvis prosessene har prioriteringer.

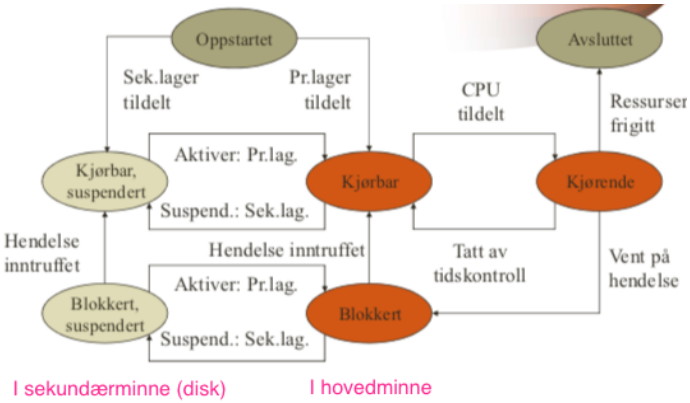
Merk: det kan hende at køene inneholder prosesskontrollblokker istedenfor prosesser, s. 156

Prosessmodell med Suspendert-tilstand

OBS: multiprogrammering er ikke immun mot at prosessor må vente på IO enheter!

Prosessoren er mye raskere enn IO, så i fem-tilstand prosessmodellen vil det ofte skje at alle prosessene som er i hovedminnet venter på IO. Selv med multiprogrammering, kan det altså hende at prosessoren må vente på IO operasjoner. **Løsningen på dette problemet kalles swapping, som innebærer at deler eller hele prosessen flyttes fra hovedminnet til disken (dvs. fra primær- til sekundærminnet).** Når ingen av prosessene i hovedminnet er i Kjørbar-tilstand, vil OS swappe en av de blokkerte prosessene til disken og inn i en suspendert-kø. OS tar deretter inn en annen prosess fra suspendert-køen eller en ny prosess på forespørsel. Dette innebærer IO til disk, men totalt sett er ikke dette mer krevende enn å vente på prosesser og vil derfor som regel øke ytelsen.

Dette kan inkluderes i prosessmodellen ved å legge til en Suspendert-tilstand. Når alle prosesser er i Blokkert-tilstand kan en av disse flyttes til Suspendert-tilstand. Den frigjorte plassen i hovedminnet kan brukes for å hente inn en annen prosess. Dersom prosessen skal hentes fra suspendert-køen er det viktig at man henter en prosess som ikke er fortsatt blokkert. **Derfor kan man dele Suspendert-tilstanden inn i Kjørbar, suspendert- og Blokkert, suspendert-tilstand**. Prosessen flyttes mellom disse når hendelsen den venter på inntreffer. Når det blir ledig plass i hovedminnet, kan OS hente prosess som er i Kjørbar, suspendert-tilstand eller ny prosess hvis alle suspenderte prosesser fortsatt er blokkerte. Se s. 146-147 for beskrivelse av overganger.



Merk: den suspenderte prosessen er ikke umiddelbart tilgjengelig for utføring, selv om den er kjørbar. Den må aktiveres ved å føres inn i hovedminnet (primært lager)

Merk: virtuelt minne gjør at det blir mulig å utføre en prosess som kun er delvis i hovedminnet, ved at deler av prosessen kan hentes inn ved behov. Det er likevel fortsatt nødvendig med swapping for å bevare ytelsen (mer i kap 8).

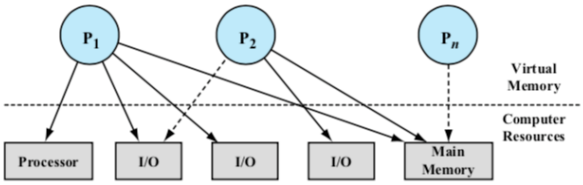
OS kan ha flere grunner til å suspendere en prosess, annet enn at hovedminnet er fult. **Noen årsaker til at en prosess suspenderes er:**

- **Swapping** – OS trenger å frigjøre tilstrekkelig hovedminne for å hente inn en prosess som er klar til å kjøres
- **Andre OS årsaker** – OS kan stoppe en bakgrunns- eller verktøysprosess, eller en prosess som er mistenkt for å forårsake et problem
- **Interaktiv bruker request** – en bruker kan ønske å avbryte utførelsen av et program
- **Timing** – det kan hende prosessen utføres periodisk (eks: system monitoring prosess) og må avbrytes for å vente på neste tidsintervall
- **Parent prosess request** – en parent prosess kan ønske å avbryte en etterfølger prosess for å undersøke eller endre prosessen eller koordinere aktiviteten til flere etterfølgere. Hvis parent prosess avbrytes kan det hende OS også vil terminere etterfølgerne.

Avbruddssystem (kompendium)

Avbruddssystemet sørger for at maskinvaren kan sende signaler til prosessoren for å indikere at en hendelse har oppstått. Det kan også brukes for å varsle om at en tilstand med høyere prioritet krever at utførende prosess med lavere prioritet avbrytes. Et avbrudd varsler prosessoren om at en tilstand med høyere prioritet krever at den utførende koden på prosessoren må avbrytes. Avbruddsystemet har innvirkning på effektivitet og funksjonalitet i datamaskinen, og det skaper både utfordringer og muligheter ved utviklingen av operativsystem. Det viktig at OS-utviklere har kjennskap til maskinens avbruddssystem, siden OS har i oppgave å blant annet tidsstyre prosesser, håndtere lagret, osv.

3.3 Prosessbeskrivelse



OS vil kontrollere hendelser som foregår innenfor datasystemet og håndtere hvordan prosesser bruker systemressurser (se figur). I et multiprogrammering system har det blitt laget flere prosesser ($P_1, \dots, P_n$) som eksisterer i et virtuelt minne. I løpet av utføring vil hver prosess kreve tilgang til bestemte systemressurser, slik som prosessor, IO enhet og hovedminne. For eksempel kan vi se at P_1 har tilgang til flere ressurser og P_2 er blokkert fra tilgang til en IO enhet som følger av at den er opptatt. Det er OS som har bestemt hvilke prosesser som skal ha tilgang til hvilke ressurser. P_n har blitt swappet og er derfor suspendert. Mht. prosesser vil OS ha følgende rolle (F):

- **Direkte** – operativsystemet vil direkte kontrollere oppstart og avslutning av prosesser, tilstandskontroll, synkronisering, kommunikasjon, tilordning av CPU-kraft og håndtering av avbrudd.
- **Indirekte** – operativsystemet vil indirekte kontrollere tildeling av lagerplass, swapping og virtuelt lager, tilordning av IO-kapasitet, buffring av filer, systemoppsyn og regnskapsføring (*accounting*)

Merk: de to neste delseksjonene gir kun et overblikk, siden innholdet ikke nevnes i forelesning

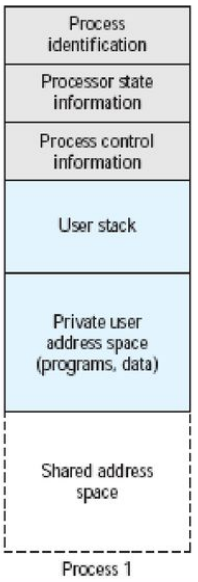
OS kontrollstrukturer (s. 149-150)

For å håndtere prosesser og ressurser må OS ha informasjon om deres nåværende tilstand, noe det oppnår ved å lage og opprettholde tabeller med informasjon om hver entitet. OS vil ha tabeller for minnet, IO, filer og prosesser. Detaljene varierer mellom ulike typer, men alle operativsystem vil opprettholde informasjon i disse fire kategoriene. **Minnetabeller** brukes for å holde styr over primær- og sekundærminnet, **IO tabeller** brukes for å kontrollere IO enheter og kanaler i datasystemet, **filtabeller** brukes for å gi informasjon om eksistensen, lokasjonen og statusen hos filer og **prosesstabeller** brukes for å håndtere prosesser. Disse tabellene er koblet sammen på en eller annen måte.

Prosess kontrollstrukturer (s. 151-157)

For at OS skal kunne håndtere og kontrollere en prosess, må den vite:

1. **Lokasjonen til prosessen** – prosessen (*process image*) vil bestå prosessdata, program, stakk (kalles også user stack, lagrer parametere og adresser som brukes av prosessen i løpet av utføring) og prosess kontroll-blokken. Lokasjonen til prosessen vil avhenge av lagerhåndteringen som brukes. Den primære prosesstabellen må inneholde en peker til hver prosess som befinner seg i primær- eller sekundærminnet (ved paging vil pekeren peke mot page).
2. **Attributter hos prosessen** – et multiprogrammering system krever mye informasjon om hver prosess og dette ligger i prosesskontrollblokken. Attributtene deles inn i:
    a. **Identifikator** – inkluderer unik identifikator for prosessen (brukes for kryss-referanser) og ofte identifikator for parent-prosessen og for brukeren som er ansvarlig for jobben
    b. **Tilstandsinformasjon** – fylles med innholdet til prosessorregistrene dersom prosessen blir avbrutt (gjør gjenoppretting mulig). Antall register varierer, men det vil som regel inkludere brukerregister, systemregister, PC, PSW, osv.
    c. **Kontrollinformasjon** – tilleggsinformasjon som trengs av OS for å kontrollere og koordinere ulike aktive prosesser. Dette inkluderer prioritet, prosesstilstand, tildelte ressurser, avventede hendelser, osv. Vi vil se mer på disse i senere kapitler.



Figuren viser en mulig struktur for en prosess (*process image*) som er lagret i det virtuelle minnet. Prosessen består av en prosesskontrollblokk i grått (bruker-kontekst), en bruker-stakk, bruker-adresserom (peker til bruker-data og bruker-program) og eventuelt et delt adresserom (peker til adresserom som deles med andre prosesser). **Prosesskontrollblokken (prosess-kontekst) er den viktigste datastrukturen i et OS og den vil leses og/eller modifiseres av nesten alle moduler i OS.** **Settet av prosesskontrollblokker vil definere tilstanden til OS**. Det er derfor viktig at disse blokkene blir tilstrekkelig beskyttet (s. 156).

Merk: bruker-adresserom hos en prosess er en del av hovedminnet som er tildelt prosessen (program eller data). Dette kan være privat for prosessen eller delt med flere prosesser som bruker samme program/data.

3.4 Prosesskontroll

Utføringsmoduser

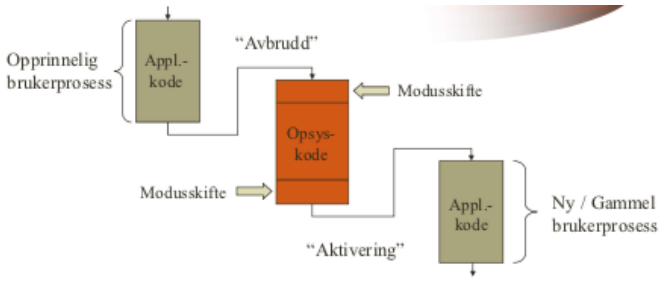
Vi introduserer utføringsmoduser for å skille mellom når prosessoren utfører prosesser assosiert med OS og når den utfører prosesser assosiert med brukerprogram. Bestemte instruksjoner kan kun utføres i mer privilegerte moduser (eks: lese/skrive kontroll register, lagerhåndtering, osv.). I tillegg vil det være bestemte minneregioner som kun kan aksesseres i mer privilegerte moduser. Brukermodus er den mindre privilegerte modusen og brukerprogram blir som regel utført i denne modusen. Den mer privilegerte modusen kalles systemmodus, kontrollmodus eller kjernemodus (*kernel mode*). **Moduser brukes for å beskytte OS og viktige OS tabeller, slik som prosesskontrollblokker, fra forstyrrelser pga. brukerprogram**. **Programvare i kjernemodus har fullstendig kontroll over prosessoren og alle dens instruksjoner, register og minne, noe brukerprogram ikke bør ha**. En bit i PSW vil gi prosessoren hvilken modus den skal være i, og denne endres ved spesifikke hendelser (eks: avbrudd). Endring av modus innebærer endring av CPL i prosessor status register. Når et avbrudd oppstår vil prosessoren tømme registret, slik at CPL blir satt til nivå 0 som har høyest privilegium. Ved enden av avbruddshåndteringen vil siste instruksjon være IRT (interrupt return) som gjør at prosessor gjenoppretter prosessor status register og CPL blir satt til forrige privilegium-nivå.

Prosesskapning

Når OS har bestemt at det skal lages en ny prosess (s. 19) vil OS utføre følgende steg:

1. **Tildele en unik prosessidentifikator til prosessen** – ny enhet legges til i primær prosesstabell (dvs. peker til prosess)
2. **Tildele rom for prosessen** – OS må vite hvor mye rom som trengs for å få plass til hele prosess image (standardverdi, gis av bruker request eller parent, osv.)
3. **Initialisere prosesskontrollblokken** – identifikatorene settes lik gitte verdier, tilstandsinformasjonen blir satt lik 0 (bortsett fra PC og stakkpekere) og kontrollinformasjonen settes til standard verdier pluss attributter som har blitt forespurt for denne prosessen (eks: prioritet)
4. **Sette passende koblinger** – prosessen må plasseres i en kø (Kjørbar eller Kjørbar, suspendert), som for eksempel kan være en linked-list
5. **Lag eller utvid andre datastrukturer** – hvis OS opprettholder en regnskapsfil, må den nye prosessen legges til

Prosesskifte

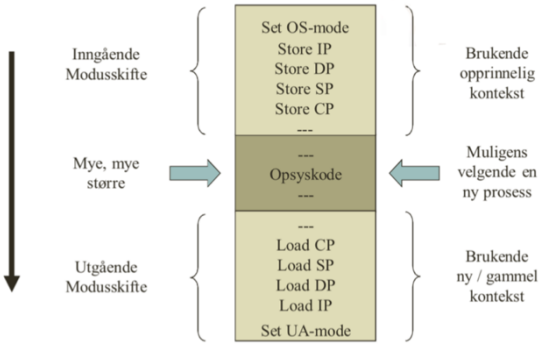


Prosesskifte innebærer at OS tildeler Kjørende-tilstand til en annen prosess og gir kontrollen til denne prosessen. CPU blir altså byttet fra en prosess til en annen, og de to prosessene vil gjennomføre tilstandsendringer. Prosesskifte vil også involvere modusskifte inn og ut av operativsystem (se figur), siden det er OS som vil avgjøre hvordan skiftningen skal skje (dvs. hvilke prosesser som skal være involvert). Prosesskifte kan skje ved ethvert tidspunkt OS har fått kontroll fra kjørende prosess, noe som kan skyldes:

- **Avbrudd** – skyldes en ekstern hendelse som er uavhengig av kjørende prosess (eks: fullført IO-operasjon). Kontrollen blir først overført til en avbruddshåndterer, som utfører noen oppgaver og forgreines til et OS rutine som er relatert til den bestemte typen avbrudd som har forekommet. Tre eksempler på avbrudd som kan forekomme:
  - **Klokkeavbrudd** – OS bestemmer om kjørende prosess har overgått maksimum tillatt utføringstid, som kalles tidsspalte (*time slice*). Hvis det er tilfellet vil prosessen skiftes til Kjørbar-tilstand og en annen prosess blir utsendt av dispatcher
  - **IO avbrudd** – når en IO-operasjon er fullført må OS bestemme hvilken operasjon dette er og flytte prosesser som venter på denne operasjonen fra Blokkert (suspendert eller ikke) til Kjørbar-tilstand. OS må deretter bestemme om utførelse av kjørende prosess skal gjenopptas eller om det finnes en prosess med høyere prioritet i Kjørbar-tilstand.
  - **Minnefeil** – hvis prosessor møter en referanse til en virtuell minneadresse som ikke er i hovedminnet må OS bringe blokken fra sekundær minne til hovedminne. Kjørende prosess som sendte IO request om minneblokken blir plassert i Blokkert-tilstand helt til blokken er hentet inn i hovedminnet (blir da plassert i Kjørbar-tilstand). OS vil skifte til utføring av annen prosess.
- **Felle** – skyldes en feil eller et unntak som genereres innenfor nåværende kjørende prosess (eks: forsøk på uautorisert filaksess). OS vil bestemme om feilen/unntaket er fatal, og hvis det er tilfellet vil kjørende prosess flyttes til Avsluttet-tilstand og et prosesskifte skjer for en prosess i Kjørbar-tilstand. Hvis den ikke er fatal vil handlingen til OS avhenge av type feil/unntak og OS-designet. Den kan forsøke en gjenoppretting, varsle brukeren, utføre prosesskifte, osv.
- **Supervisor call (OS-kall)** – skyldes en intern hendelse i prosessen som gjør at den sender request til en OS-funksjon (eks: filaksess). Dette resulterer i overføringen til et rutine som er del av OS-koden. Det kan hende at brukerprosessen plasseres i Blokkert-tilstand.

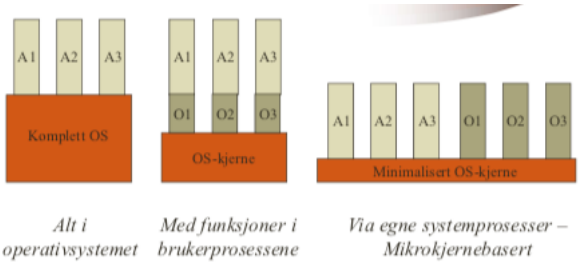
Modusskifte

Et modusskifte innebærer endring av privilegier, for eksempel kan prosessor skifte fra brukermodus til kjernemodus, slik at fremtidig kjørende prosess får større aksess til adresserom og instruksjonssett (og motsatt). Typiske funksjoner i kjernemodus inkluderer styring av prosesser, minnehåndtering, IO-styring og støttefunksjoner, noe brukermodus ikke har tilgang til. **Hendelse som kan utløse et modusskifte er eksternt avbrutt, instruksjonsreaksjon (felle) og OS-kall.**



Ved avbruddssteget i instruksjonssyklusen vil prosessoren sjekke om den har mottatt noen avbruddssignaler (s. 4). Hvis den har mottatt et avbruddssignal vil prosessoren sette PC lik startadressen til avbruddshåndterer-programmet og **skifter til kjernemodus slik at avbruddshåndteringen kan inkludere privilegerte instruksjoner** (husk: bit i PSW gjør at prosessor vet at den skal skifte modus, s. 22). Prosessoren vil deretter hente første instruksjon i håndteringsprogrammet og innholdet hos prosessen som har blitt avbrutt blir lagret inn i prosesskontrollblokken hos prosessen (dvs. tilstandsinformasjonen). På figuren kan vi se at **konteksten hos avbrutt prosess blir lagret, mens konteksten til innsendt prosess blir lastet inn i registrene til prosessoren**. Avbruddshåndterer-programmet er som regel et kort program som utfører grunnleggende oppgaver relatert til avbruddet (eks: resetting av flagg, sende acknowledgement til IO modul, osv.). Det som skjer videre vil avhenge av hva som utløste modusskifte. For eksempel dersom det var et klokkeavbrudd vil håndtereren gi kontrollen til dispatcher som vil gi kontrollen til en ny prosess og konteksten hos denne blir lastet inn i prosessorregistrene. **Det finnes også avbrudd som ikke vil innebære et prosesskifte, slik at nåværende prosess kan fortsette å utføres**. I dette tilfellet vil konteksten som ble lagret ved avbrudd gjenopprettes når prosessen får tilbake kontrollen. Lagring og lasting av konteksten blir ofte utført av hardware.

3.5 Operativsystemorganisering



Operativsystemet fungerer på samme måte som vanlig software ved at det er et sett med program som utføres av prosessoren. OS vil ofte gi fra seg kontrollen og avhenger av at prosessoren gir tilbake kontrollen til OS. Dette har ført til at det er flere tilnærminger til design av operativsystemer, inkludert:

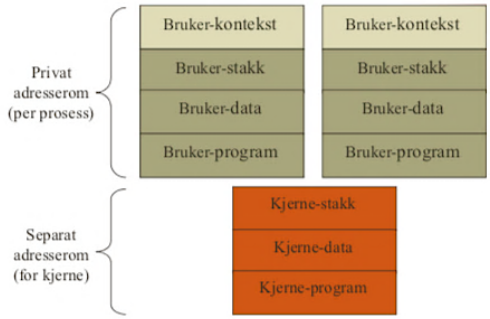
- **Komplett OS (alt i operativsystemet)** – kjernen hos OS utføres utenfor andre prosesser. Når en prosess blir avbrutt eller sender et OS-kall vil konteksten hos prosessen lagres og kontrollen gis til kjernen. OS har et eget minneområde som kan brukes og en egen systemstakk for å kontrollere prosedyrekall og returneringer. Prosesskonseptet blir kun brukt på brukerprogram og OS-koden utføres som en separat entitet i privilegert modus. Dette er lite effektivt, men det er trygt, siden privilegert aksess er begrenset til OS.
- **OS-kjerne (med funksjoner i brukerprosessene)** – nesten all OS programvare blir utført i konteksten til brukerprosesser. OS behandles som en samling av rutiner som brukeren kaller for å utføre ulike funksjoner innenfor omgivelsen til brukerprosessen. OS-kjernen vil bestå av prosesskifte-funksjoner. Hvert prosess image vil inkludere en kjernestakk som håndterer kall/returneringer når prosessen er i kjernemodus og OS-kode og -data som deles blant alle brukerprogrammene. Når et avbrudd, felle eller OS-kall oppstår vil det gjennomføres et modusskifte, men ingen prosesskifte fordi OS-rutinen utføres innenfor nåværende brukerprosess. Hvis OS bestemmer at nåværende prosess skal fortsette utførelsen, vil det kun være behov for et modusskifte og systemet unngår dermed prosesskifte (fordel ved OS-kjerne = mer effektiv). Hvis OS

Merk: prosesskifter er tidkrevende, så ved å redusere mengde prosesskrift vil effektiviteten kunne øke

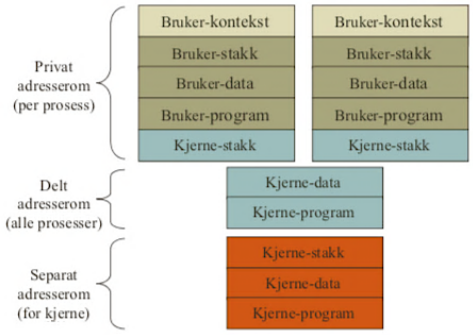
bestemmer at prosessen skal skiftes vil kontrollen gis til en prosesskifte-rutine (utenfor prosesser). Brukeren kan ikke tukle med OS-rutinene, fordi når prosessoren er i kjernemodus vil den utføre den delte OS-koden og ikke brukerkode.. Dette illustrerer skillet mellom prosesser og program, og at dette ikke er en en-til-en relasjon. Innenfor en prosess kan både et brukerprogram og et OS-program utføres, og OS-programmet som utføres i ulike brukerprosesser er identisk. Det vil være mer effektivt, men mindre trygt sammenlignet med komplett OS.

- **Minimalisert OS-kjerne (mikrokjerne)** – OS implementeres som en samling av systemprosesser, der kjerne-programvare utføres i kjernemodus og viktige kjernefunksjoner er organisert i separate prosesser. Prosesskifting utføres utenfor prosesser. Dette fremmer bruk av modulær OS med minimale grensesnitt mellom modulene. Ikke-kritiske OS-funksjoner blir implementert som separate prosesser som er nyttig for multiprosessor og multikjerne-omgivelser, siden noen OS-tjenester kan håndteres av dedikerte prosessorer for å bedre ytelsen. Sammenlignet med komplett OS vil dette designet likevel gi mindre effektivitet (flere prosesskifter siden det også krever skifte til OS-prosesser) og større trygghet (mindre software har mye privilegerer)

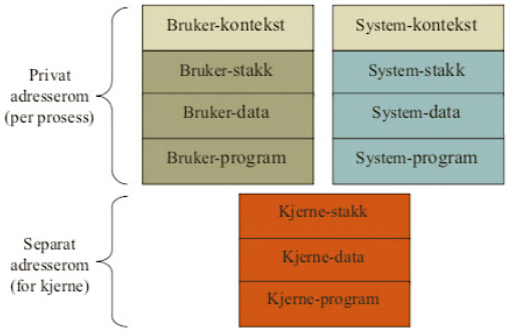
Mer på s. 32



Komplett OS der OS-koden utføres som separat entitet i privilegert modus (dvs. behandles ikke som prosess). Dette er den tradisjonelle tilnærmingen. Lite effektivt, men trygt.



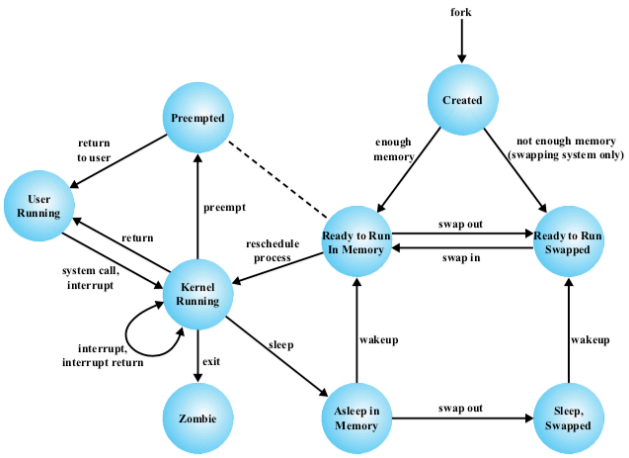
OS-kjerne der brukerprosess inneholder kjernestakk og OS-kode og data deles mellom alle brukerprosessene. Dette brukes i tidlig UNIX. Mer effektivt, men mindre trygt



Minimalisert OS-kjerne der kjerne-programvare utføres i kjernemodus og andre OS-tjenester er implementert som separate prosesser. Dette er den moderne tilnærmingen. Lite effektivt, men trygt. Mer effektivt for multiprosessorsystem.

3.6 UNIX SVR4 prosesshåndtering (s. 166-170)

UNIX System V følger modellen for OS-kjerne ved at det meste av OS utføres i omgivelsen til en brukerprosess. Det bruker to prosesskategorier: **systemprosesser** som kjører i kjernemodus for å utføre administrative funksjoner (eks: tildeling av minne og swapping), og **brukerprosesser** som kjører i brukermodus for å utføre brukerprogram og kjernemodus for å utføre instruksjoner som hører til kjernen. **UNIX SVR4 bruker en standard prosesshåndtering uten tråder, multiprosessering eller mikrokjerne**. Figuren viser prosesshåndteringen som er basert på ni prosesstilstander (se tabell).



User Running	Executing in user mode.
Kernel Running	Executing in kernel mode.
Ready to Run, in Memory	Ready to run as soon as the kernel schedules it.
Asleep in Memory	Unable to execute until an event occurs; process is in main memory (a blocked state).
Ready to Run, Swapped	Process is ready to run, but the swapper must swap the process into main memory before the kernel can schedule it to execute.
Sleeping, Swapped	The process is awaiting an event and has been swapped to secondary storage (a blocked state).
Preempted	Process is returning from kernel to user mode, but the kernel preempts it and does a process switch to schedule another process.
Created	Process is newly created and not yet ready to run.
Zombie	Process no longer exists, but it leaves a record for its parent process to collect.

The **Zombie** state is used by UNIX to denote that a process was terminated.

Se side 166-170 for beskrivelse av tilstandene, prosesskontekst, prosess image og prosesskontroll som brukes av UNIX SVR4.

Kapittel 4 – Tråder

Konseptet med prosesser er mer komplekst enn det vi har sett på i kapittel 3. Prosesser kan deles inn i to separate og potensielt uavhengige konsepter, der en er relatert til ressurs-eierskap og den andre til utføring. Dette skillet har ført til utviklingen av **tråder**.

4.1 Prosesser og tråder

Prosesskonseptet omfatter to egenskaper:

1. **Ressurseierskap** – en prosess vil inkludere et virtuelt adresserom som holder prosess image (program, data, stakk og prosesskontrollblokk). Prosessen kan også tildeles eierskap over ressurser, slik som hovedminne, IO enheter, filer, osv. OS utfører en beskyttelsesfunksjon for å hindre uønsket forstyrrelse mellom prosesser mht. ressurser
2. **CPU-bruk** – utføringen av prosessen følger en utførelsesbane (spor, *trace*) gjennom en eller flere programmer, og denne kan være sammenflettet med andre prosesser. Prosessen har en prosesstilstand (eks: Kjørbar, Kjørende) og en prioritering, og det er entiteten som tidsstyres og utsendes av OS.

Disse egenskapene er uavhengige, og for å skille disse blir utføringsenheten kalt en tråd eller lettvekts prosess, mens enheten for ressurseierskap kalles prosess eller oppgave.

Tråder vs. prosesser

Fordeler ved tråder	Ulemper ved tråder
Skiller ressurseierskap og CPU-bruk	Flere gjensidig avhengige konsepter
Passer naturlig til applikasjonsstruktur	Mer kompleks kontroll og tidsstyring
Gir ny kommunikasjon via deling av lager	Trådskifter kan ikke tilfredsstille alle behov
Gir færre tidkrevende prosesskifter	Mer kompleks synkronisering

En enkel prosess som tildeles bestemte ressurser kan deles inn i flere, samtidige tråder som kan utføres i samarbeid for å utføre arbeidet til prosessen. Dette introduserer et nytt nivå med parallell aktivitet som må håndteres av programvare og maskinvare. Tråder er altså miniprosesser som tilhører en overordnet prosess. En prosess kan opprette flere undertråder for å kunne parallelt utføre ulike deloppgaver og samtidig oppnå kommunikasjon og håndtering av deloppgavene på en billigere måte enn med fulle prosesser. Tabellen viser fordeler og ulemper ved å bruke tråder kontra prosesser. **Tråder vil også introdusere nye utfordringer:**

1. Effektiv utnyttelse av flere prosessnivåer
2. Riktig synkronisering og kommunikasjon innen avanserte applikasjoner

Prosesser vil derfor passe best til noen formål (eks: isolerte aktiviteter), mens tråder passer best for andre formål (eks: mer samhørende aktiviteter). Ofte brukes en kombinasjon.

Multithreading

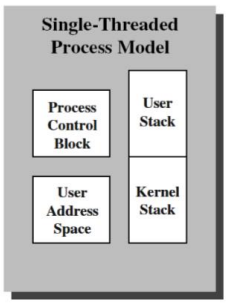
Multithreading er operativsystemets evne til å støtte flere, samtidige utførelser innenfor en enkelt prosess. Single-threaded tilnærming er den tradisjonelle tilnærmingen der hver prosess utføres som én tråd. Noen operativsystem (eks: varianter av UNIX) støtter flere brukerprosesser, der det er en tråd per prosess. Multithreaded tilnærming innebærer at en prosess har flere tråder (eks: Java). Mange moderne OS tilbyr flere prosesser, der hver prosess har flere tråder (eks: Windows, Solaris, moderne UNIX, osv.). **I et multithreaded miljø vil prosessen defineres som en enhet for ressurstildeling og beskyttelse**. Følgende er assosiert med hver prosess:

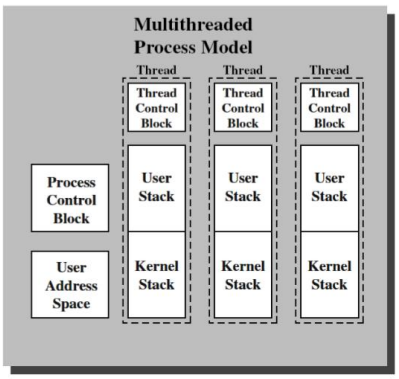
- **Virtuelt adresserom** – holder prosess image med prosesskontekst, prosesstakk, prosessdata og prosessprogram.
- **Beskyttet aksess til ressurser** – tilgangen til prosessorer, andre prosesser (IPC), filer og IO ressurser beskyttes

Innenfor en prosess kan det være en eller flere tråder, og følgende er assosiert med hver tråd:

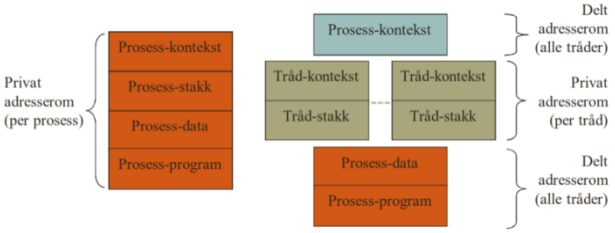
- **Tilstand** – Kjørbar, Kjørende, Blokkerts, osv.
- **Kontekst** – når tråden ikke kjører vil trådkonteksten lagres. En tràs kan ses på som en uavhengig program counter som operer innenfor en prosess
- **Stakk** – brukes under utførelse av program
- **Statisk minne** – hver tråd har statisk lagring av lokale variabler
- **Aksess til minne og ressurser** – tilgang til minne og ressurser hos prosessen som tråden tilhører, delt med andre tråder i samme prosess

Merk: bruker-adresserommet inneholder pekere til hovedminnet der program og data befinner seg. Bruker-stakken er LIFO køer som brukes for å lagre parametere og adresser underveis i utføring av brukerprogram, mens kjerne-stakken brukes under utføring av OS-program





Figuren viser overgangen fra prosesser til tråder. I en single-threaded prosess (venstre) vil prosessen bestå av prosesskontrollblokk (prosess-kontekst), prosess-stakk (bruker og kjerne) og bruker-adresserom (program og data). I en multithreaded prosess (høyre) vil prosessen fortsatt ha en prosesskontrollblokk og et bruker-adresserom med data og program assosiert med prosessen, men **hver tråd har sin egen bruker- og kjernestakk og en trådkontrollblokk som inneholder register, prioritet og annen tilstandsinformasjon relatert til den tråden**. **Hver tråd vil altså ha egne stakker for å lagre parametere og adresser som brukes av tråden i løpet av utføring, men de har også tilgang til det delte bruker-adresserommet hos prosessen (figur til venstre).** Alle trådene hos en prosess vil altså dele tilstand og ressurser med prosessen. De ligger i samme adresserom og har tilgang til samme data. Dette gjør at en tråd kan se når andre tråder endrer verdien hos dataenheter i minnet. Hvis en tråd åpner en fil med leseprivilegier, kan andre tråder i samme prosess også lese fra denne filen.



Fordeler med tråder – knyttet til utførelse

De viktigste fordelene ved bruke av tråder omhandler ytelse:

1. Det tar **mye mindre tid å lage en ny tråd** i en eksisterende prosess enn å lage en helt ny prosess (10x raskere)
2. Det tar **mindre tid å terminere en tråd** enn en prosess
3. Det tar **mindre tid å skifte mellom tråder** innenfor samme prosess enn å skifte mellom prosesser
4. **Tråder øker effektiviteten i kommunikasjon mellom ulike utførende programmer**. I de fleste OS vil kommunikasjon mellom prosesser kreve at kjernen gir beskyttelse og andre nødvendige mekanismer. Ettersom tråder i samme prosess deler minne og filer, kan de kommunisere med hverandre uten å innblande kjernen.

Hvis applikasjonen skal implementeres som et sett med relaterte enheter som utføres, vil det være mye mer effektivt å bruke en samling tråder enn en samling separate prosesser. Et eksempel er en filserver, der hver tråd håndterer en fil-request (s. 180).

Bruk av tråder i uniprosessor- og multiprosessor-program

Tråder kan være nyttige i uniprosessor-program for å forenkle strukturen til program som gjør flere ulike funksjoner eller for å sammenflette flere tråder fra flere prosesser. I multiprosessorsystem kan flere tråder innenfor samme prosess utføres samtidig på ulike prosessorer. Fire eksempler på bruk av tråder i single-user multiprosessorsystem er:

1. **Forgrunns- og bakgrunnsarbeid** – i et regnearkprogram kan en tråd viser frem menyer og lese brukerinput, mens en annen kjører bruker-kommandoer og oppdaterer regnearket. Dette kan gjøre at oppfattet hastighet øker, siden programmet kan be om neste kommando før forrige kommando er ferdig
2. **Asynkron prosessering –** asynkrone elementer i programmet kan implementeres som tråder, for eksempel kan beskyttelse mot strømavbrudd håndteres ved å lage en tråd som utfører periodisk backup (lagrer RAM til disk én gang i minuttet)
3. **Utføringshastighet** – parallellitet gir økt hastighet. En flertrådet prosess kan beregne en batch med data samtidig som neste batch med data leses fra en enhet. I et

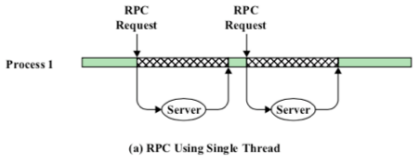
multiprosessorsystem kan flere tråder fra samme prosess utføres parallelt, slik at hastigheten til utførelsen av programmet øker.

4. **Modulær programstruktur** – programmer med varierte aktiviteter eller varierte kilder og destinasjoner av input/output kan være enklere å designe og implementere med tråder

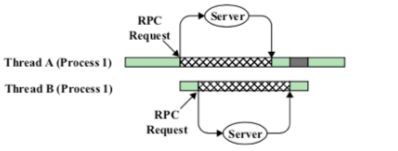
Trådfunksjonalitet

Trådtilstander

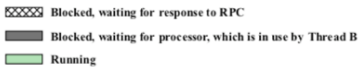
I likhet med prosesser vil nøkkeltilstander for tråder være Running (Kjørende), Ready (Kjørbar) og Blocked (Blokkert). Generelt gir det ikke mening å assosiere suspenderingstilstander med tråder, ettersom dette er et konsept på prosessnivå. **Dersom en prosess byttes ut (suspender/swappet), vil alle trådene i prosessen også byttes ut siden de deler adresserom med prosessen**. Terminering av en prosess vil også terminere trådene i prosessen. I en OS som støtter tråder vil tidsstyring og dispatching gjøres på tråd-nivå, men det er altså noen handlinger som vil påvirke alle trådene i prosessen og som må håndteres på prosess-nivå. Trådoperasjonene som endrer trådtilstanden er Spawn (tråd-skapelse), Block (venter på hendelse), Unblock (hendelse forekommer) og Finish (ressurser frigis).



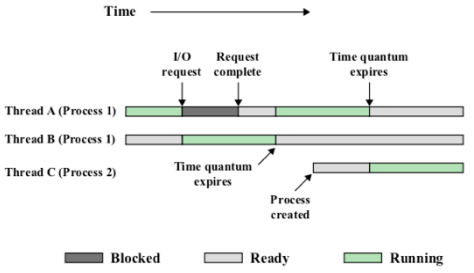
(a) RPC Using Single Thread



(b) RPC Using One Thread per Server (on a uniprocessor)



Figuren til venstre viser ytelsesfordelene gitt at en blokkert tråd ikke vil blokkere hele prosessen (mer senere). Ett program utføres av to ulike servere for å oppnå et kombinert resultat. Del a viser tilfellet for enkelttrådet prosess, der programmet må vente på responsen hos hver server i sekvens. Del b viser tilfellet for flertrådet prosess, der hver server utfører hver sin tråd slik at programmet utføres mye raskere.



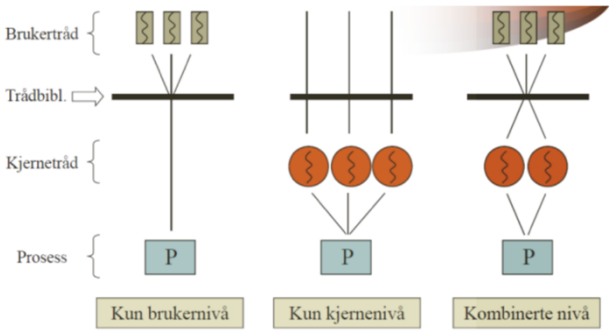
Figuren til høyre viser ytelsesfordelene ved en uniprosessor. **Multiprogrammering gjør at flere tråder innenfor flere prosesser kan flettes sammen**, slik at prosessoren kan slippe å vente når tråder blir blokkert av IO-operasjoner.

Trådsynkronisering

Enhver endring en tråd gjør på en ressurs påvirker miljøet til de andre trådene i prosessen, siden de deler ressurser. **For å hindre at trådene forstyrrer hverandre eller ødelegger datastrukturene, er det derfor nødvendig å synkronisere aktivitetene til de ulike trådene.** Problemet som oppstår og teknikkene som brukes for å synkronisere tråder, er i hovedsak den samme som for prosesser (kapittel 5 og 6).

4.2 Trådtyper

Det er to hovedkategorier for implementasjon av tråder: User-Level Threads (ULT) og Kernel-Level Threads (KLT). Vi ser nærmere på disse.

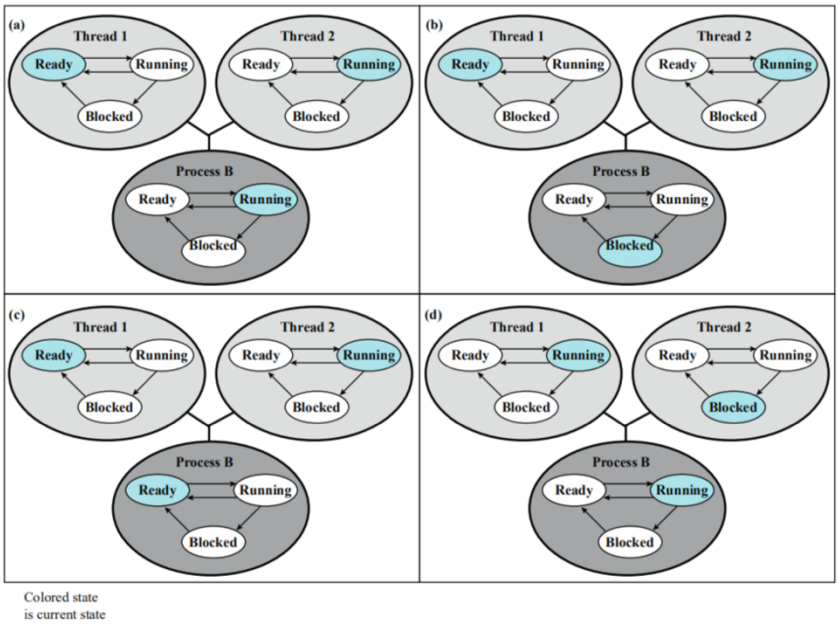


Brukertråder: User-Level Threads (ULT)

Ved brukertråder vil trådhåndtering kun utføres av applikasjonen og kjernen er ikke klar over eksistensen av tråder. **Hver tråd (miniprosess) er altså representert av en egen aktivitet som ikke er kjent av OS, så de må opprettes, kjøres og styres på en mindre fleksibel måte av et programvarebibliotek**. Programmerere kan bruke et **trådbibliotek** for å lage applikasjoner med flere tråder, og dette biblioteket inneholder kode for å lage og ødelegge tråder, sende meldinger og data mellom tråder, tidsstyre tråder og lagre og gjenopprette trådkontekster. En applikasjon begynner vanligvis med en enkelt tråd

og utvider til flere tråder ettersom prosessen utføres (dvs. er i Running-tilstand). Tråder lages via et kall til trådbiblioteket, som vil lage en datastruktur for den nye tråden. Biblioteket vil så gi kontrollen til en av trådene som er i Ready-tilstand vha. en tidsstyringsalgoritme. Når kontrollen gis til biblioteket vil konteksten til nåværende tråd lagres og når kontrollen gis tilbake til tråden vil konteksten gjenopprettes. Konteksten består av innholdet i brukerregistre, PC og stakkpekere. Dette foregår i brukerrommet innenfor en enkelt prosess, og kjernen er ikke klar over denne aktiviteten (tidsstyrer prosessen som en enhet og gir prosesstilstand).

Tidsstyring av tråder vs. prosesser



Vi ser på forholdet mellom tidsstyring av tråder og prosesser. Figur a viser tilfellet der vi har prosess B som utføres i sin tråd 2. Figuren viser tilstandene til prosessen og de to brukertrådene (ULTs) som er en del av prosessen. Basert på denne initiale situasjonen ser vi på tre mulige utfall:

1. Figur b – tråd 2 utfører et systemkall (eks: IO) som blokkerer B. Kontrollen overføres til kjernen som plasserer B i Blocked-tilstand og skifter til en annen prosess. I følge datastrukturen som håndteres av trådbiblioteket vil tråd 2 fortsatt være i Running-tilstand, selv om den ikke utføres av prosessoren. Den oppfattes å være i Running-tilstand av trådbiblioteket.
2. Figur c – et klokkeavbrudd gir kontroll til kjernen som vil plassere kjørende prosess B i Ready-tilstand og skifter til en annen prosess. I følge datastrukturen som håndteres av trådbiblioteket vil tråd 2 fortsatt være i Running-tilstand.
3. Figur d – tråd 2 har nådd et punkt der den trenger at tråd 1 utfører en handling. Tråd 2 entrer Blocked-tilstand og tråd 1 går fra Ready- til Running-tilstand. Prosessen forblir i Running-tilstand.

I tilfelle 1 og 2 når prosess B får tilbake kontrollen vil utføringen gjenopptas i tråd 2. Hvis prosessen avbrytes midt i et trådskifte, dvs. utføring av kode i trådbiblioteket, vil utføringen fortsette når prosessen får kontrollen tilbake og trådskifte blir fullført.

Fordeler ved å bruke brukertråder

Noen fordeler ved å bruke brukertråder (ULT) istedenfor kjernetråder (KLT) er:

1. **Trådskifte krever ikke kjernemodus** siden all trådhåndtering gjøres innenfor brukeradresserommet til en enkelt prosess. Dette sparer tid (unngår modusskifte)
2. **Tidsstyringen kan skreddersys applikasjonen**. For eksempel vil noen applikasjoner ha fordel av en round-robin algoritme, mens andre vil rangere etter prioritet.
3. **ULT kan kjøre på ethvert OS** og det kreves ingen endringer i den underliggende kjernen for å støtte ULT
4. **ULT har billigere systemkall enn KLT**

Ulemper ved å bruke brukertråder

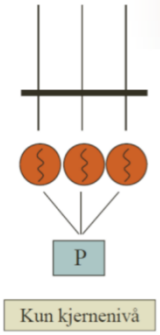
Noen ulemper ved å bruke brukertråder (ULT) istedenfor kjernetråder (KLT) er:

1. **ULT har mange blokkerende systemkall**, og når et systemkall utføres av en tråd vil tråden og alle andre tråder i prosessen blokkeres.

2. **Ved bruk av ren ULT kan man ikke bruke multiprosessering fordi kjernen kun kan gi en prosess til prosessoren av gangen**. Derfor kan kun en tråd i prosessen utføres av gangen. Det er mulig å signifikant øke hastigheten ved å bruke multiprogrammering innenfor en enkel prosess, men flere applikasjoner har god nytte av å utføre deler av koden samtidig (dvs. multiprosessering)

For å overkomme disse to problemene kan man skrive applikasjonen som flere prosesser istedenfor flere tråder. Dette vil likevel eliminere hovedfordelen ved tråder, siden hvert skifte blir et prosesskifte istedenfor et trådskifte, noe som resulterer i en **stor overhead** (dvs. mindre effektivitet). En annen løsning er **jacketing**, der blokkerende systemkall omdannes til ikke-blokkerende systemkall via trådbiblioteket. For eksempel istedenfor å direkte kalle en IO-rutine kan trådene kalle en IO jacket-rutine på applikasjonsnivå. I denne rutinen er det kode som sjekker om IO-enheten er opptatt. Dersom det er tilfellet går tråden i Blocked-tilstand og en annen tråd får kontrollen. Når denne tråden får tilbake kontrollen senere vil jacket-rutine sjekke IO-enheten på nytt.

Kjernetråder: Kernel-Level Threads (KLT)



Kjernetråder er det motsatte av brukertråder, som vil si at all trådhåndtering gjøres av kjernen. Det er ikke noe trådhåndtering på applikasjonsnivå, kun et API til kjernetråd-håndteringen. Windows er et eksempel på denne tilnærmingen. Kjernen opprettholder kontekstinformasjon for prosessen som en helhet og for de individuelle trådene i prosessen. Kjernen utfører tidsstyring på tråd-nivå. **Hver tråd (miniprosess) er representert av en egen aktivitet som er kjent av OS, så de kan opprettes, kjøres og styres på en fleksibel måte av OS-kjernen.**

Fordeler ved å bruke kjernetråder

Fordelen ved å bruke kjernetråder (KLT) istedenfor brukertråder (ULT) er at man unngår de to problemene ved ULT:

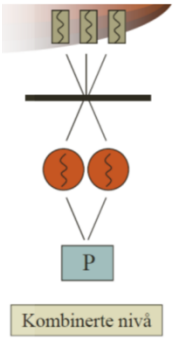
1. **KLT gir ikke-blokkerende systemkall**, som vil si at hvis en tråd i prosessen er blokker, så kan kjernen sette opp en annen tråd fra samme prosess til å kjøres
2. **KLT tillater multiprosessering**, fordi tidsstyring gjøres på trådbasis, slik at kjernen kan sette opp flere tråder fra samme prosess på ulike prosessorer

Ulemper ved å bruke kjernetråder

Operation	User-Level Threads	Kernel-Level Threads	Processes
Null Fork	34	948	11,300
Signal Wait	37	441	1,840

Ulempene ved KLT er i hovedsak at de har dyre systemkall. Hovedulempen er at kontrollbytte fra en tråd til en annen innenfor samme prosess krever modusskifte til kjernen, noe som er tidskrevende. Dette er illustrert på tabellen, der Null Fork viser tiden det tar å lage, tidsstyre, utføre og fullføre en prosess/tråd som kaller null prosedyren, mens Signal-Wait er tiden det tar for å signalisere en ventende prosess/tråd og deretter vente på en betingelse. Vi kan se at KLT vil bruke betydelig mindre tid enn single-threaded prosesser (høyre kolonne), mens ULT vil gi en enda mindre tid enn KLT igjen. Bruk av KLT vil altså gi en signifikant større hastighet enn bruke av single-threaded prosesser, og det vil være en enda mer signifikant større hastighet ved å bruke ULT. **Forbedringen i hastighet vil likevel avhenge av egenskapene til applikasjonen. For eksempel hvis de fleste trådskiftene i applikasjonen krever kjernemodus-aksess, kan det hende at ULT ikke har noe bedre ytelse enn KLT.**

Kombinasjon av brukertråder (ULT) og kjernetråder (KLT)

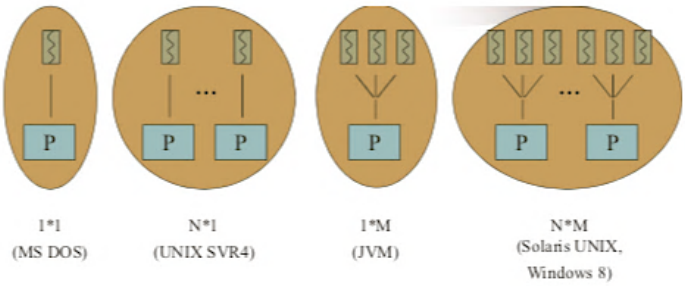


I et kombinert system blir tråder fullstendig laget i brukerrommet og det meste av tidsstyring og synkronisering utføres her. Brukertrådene fra en enkel applikasjon kartlegges til et mindre eller likt antall kjernetråder. Antall kan justeres for en bestemt

applikasjon og prosessor for å oppnå de beste helhetlige resultatene. I en kombinert tilnærming kan flere tråder fra samme applikasjon kjøre parallelt på flere prosessorer, noe som løsere problemet med blokkerende systemkall**. Denne metoden kan kombinere fordelene og unngå ulempene ved ULT og KLT dersom den designes riktig**. Et eksempel på OS som bruker den kombinerte metoden er Solaris.

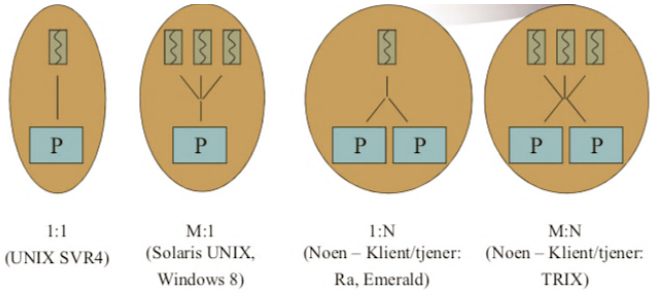
Andre kombinasjoner

Forholdet mellom tråder og prosesser har tradisjonelt sett være 1:1, men i senere tid har det vært større fokus på å gi flere tråder innenfor hver enkelt prosess, altså et mange-til-en forhold. Figuren til venstre viser noen enkle kombinasjoner (tråder : prosesser).



Det har også blitt større fokus på mer avanserte kombinasjoner, i form av mange-til-mange forhold (M:N) og en-til-mange forhold (1:N). De ulike typene forhold er:

- **1:1** – hver tråd er en unik prosess med eget adresserom og ressurser (dvs. vanlig prosess)
- **M:1** – en prosess definerer et adresserom og et dynamisk ressurseierskap. Flere tråder kan lages og utføres innenfor prosessen
- **1:M** – en tråd kan migrere fra et prosessmiljø til et annet. Dette gjør at en tråd kan lett flyttes mellom ulike system
- **M:N** – kombinerer attributter fra M:1 og 1:N



Mange-til-mange forhold

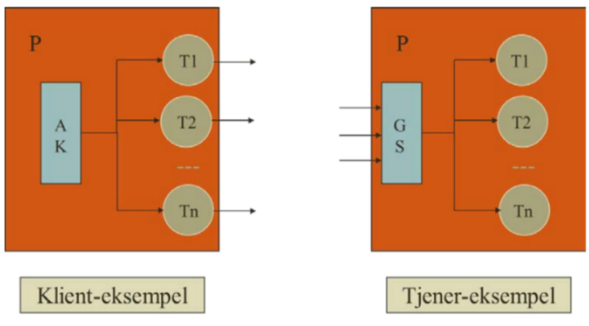
Mange-til-mange forhold har blitt utforsket i operativsystemet TRIX som bygger på to konsepter: domene og tråd. Et domene er en statisk entitet som består av adresserom og porter som kan motta og sende meldinger. En tråd er en enhet for utførelse med en stakk, tilstand og informasjon om tidsstyring. Flere tråder kan utføres innenfor et enkelt domene, noe som vil gi større effektivitet. I tillegg er det mulig at en applikasjon kan utføres i flere domener. I dette tilfellet vil det eksistere tråder som kan flyttes fra et domene til et annet. Se s. 189 for forklaring på bruk av en enkelt tråd i flere domener.

En-til-mange forhold

Distribuerte OS designes for å kontrollere distribuerte datasystem, og innenfor dette området har det vært interesse for konseptet tråd som primært en entitet som kan flyttes langs adresserom (dvs. en tråd som hører til flere prosesser). Eksempel på OS som bruker dette er Cloud (se s. 190).

Bruk av tråder i prosess-tjener-eksempel (F, Kompendium)

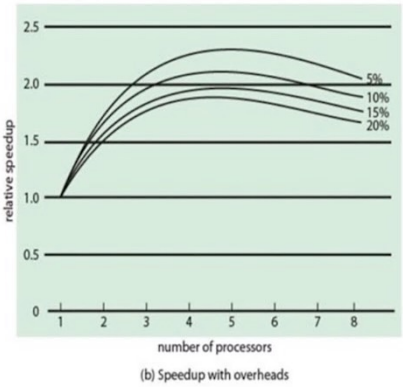
En klientapplikasjon kan sette i gang flere parallelle aktiviteter ved å operere med en separat tråd/prosess for hver av dem. En tjenesteapplikasjon kan også respondere med flere parallelle aktiviteter ved å dele hver inn i en tråd eller prosess. Tråder er enklere og raskere å opprette, håndtere og terminere enn prosesser, som er hovedgrunnen til å velge tråder istedenfor prosesser.



4.3 Multikjerne og multithreading

De potensielle ytelsesfordelene ved multikjernesystem avhenger av evnen det har til å effektivt utnytte de parallelle ressursene som er tilgjengelig for applikasjonen. Amdahls lov gir mulig speedup for multithreading som kjører på multikjernesystem:

$$\text{Speedup} = \frac{\text{utføringstid på uniprosessor}}{\text{utføringstid på N parallelle prosessorer}} = \frac{1}{(1-f)+f/N}$$



(b) Speedup with overheads

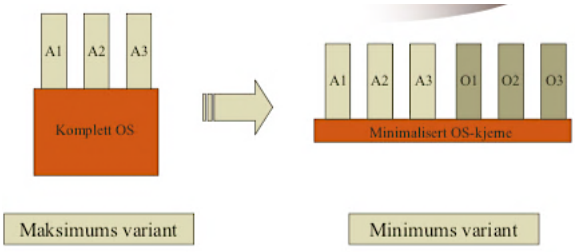
Der f er andel kode som er uendelig parallelliserbar uten tidsstyring overhead, og $(1-f)$ er andel kode som er seriell. En liten mengde seriell kode vil påvirke ytelsesgevinsten. I tillegg vil programvare som bruker flere prosessorkjerner som regel ha et ekstra overhead som resultat av kommunikasjon og distribusjon av arbeid på kjernene (det vil også være et cache-overhead). Dette gjør at ytelsesgevinsten vil nå en toppunkt ved økende antall prosessorer og deretter begynne å synke (se figur). Programvareutviklere har likevel adressert dette problemet, og det har blitt laget flere applikasjoner der det er mulig å effektivt utnytte et multikjernesystem. For eksempel har det blitt utviklet database management system (DBMS) og flere typer servere som kan håndtere flere relativt uavhengige transaksjoner i parallell. Andre applikasjoner som drar nytte av flere kjerner er multiprosess applikasjoner, Java applikasjoner, multithreaded native applikasjoner og multi-instans applikasjoner (s. 192). Et annet eksempel er spillutvikler Valve, som bruker multithreading og tråd-granularitet for å utnytte kraften til multikjerne prosessorchips (s. 193-194).

Fordeler og ulemper ved multikjernesystem

Fordeler ved å bruke multikjernesystem er at det kan gi økt ytelse siden CPU-ene kan utføre ulike oppgaver samtidig. Det vil også gi økt reell parallellitet og økt pålitelighet, fordi hvis en CPU dør så kan de andre fungere som backup. Hovedulempen ved å bruke multikjerne er at krever kompleks programvare. I tillegg resulterer det i et større ytelsesgap ettersom minnet er tregt, mens CPU er raskt. Derfor vil ikke multikjernesystem nødvendigvis gi økt ytelse, for eksempel i tilfeller der det er mange IO-operasjoner til minnet. Det vil også introdusere nye utfordringer:

- Gradvis systemutfall kan og bør sikres
- Gjensidig utelukkelse blir vanskeligere. Det er ikke lenger nok å slå av avbruddssystemet
- Samtidige kjernetråder blir tilfellet
- Réentrente kjernerutiner blir nødvendige (dvs. flere, separate kjernestakker)
- Parallelle lagerenheter kan og bør utnyttes

Mikrokjerne (F, Kompendium)



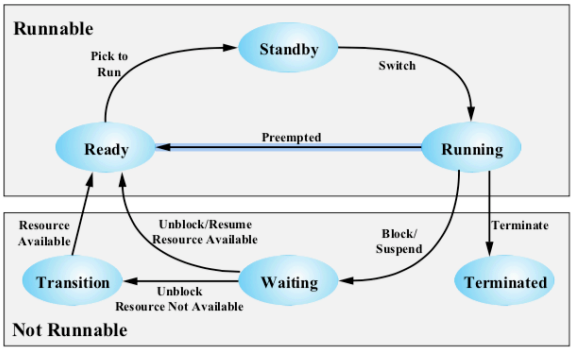
Vi så på side 25 at minimalisert OS-kjerne (dvs. mikrokjerne) er nyttig for multiprosessor og multikjerne-omgivelser, fordi OS-tjenester kan håndteres av dedikerte prosesser for å bedre ytelsen. Fordeler med mikrokjerner er at det gir enklere programvare, det er lettere å tilpasse, utvide og flytte og det støtter en mer distribuert og objektorientert tankegang. Ulemper ved mikrokjerner er at det krever endring fra eksisterende systemer og det er unødvendig for mindre systemer. Det vil også introdusere nye utfordringer i form av å oppnå riktig skille og effektivt samarbeid mellom kjernekomponenter og komponenter utenfor kjernen

4.4-4.8 Prosess- og trådhåndtering for de ulike operativsystemene (s. 195-217)

Vi ser på et overblikk på prosess- og trådhåndteringen for de ulike typene OS:

Windows 8 (s. 195-201)

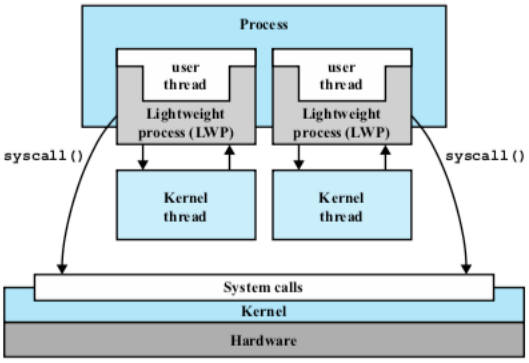
Windows bruker moderne prosesshåndtering med tråder, multiprosessering og mikrokjerne. Strukturen til Windows er objektorientert, der prosesser og tråder er implementert som objekter med tilhørende attributter (s 197). En prosess kan inneholde en eller flere tråder, og både prosesser og tråder har innebygde synkroniseringsmuligheter. Prosessen opprettes som en ny prosess eller som en kopi av en eksisterende prosess. En prosess er en entitet som korresponderer til en brukerjobb eller applikasjon som eier ressurser (eks. Minne og filer), mens en tråd er en utsendbar arbeidsenhet som utføres sekvensielt og kan avbrytes, slik at prosessoren kan skifte mellom tråder (s. 198). **Windows støtter samtidig utførelse blant prosesser fordi tråder i ulike prosesser kan utføres samtidig (s. 200).** I tillegg kan tråder fra samme prosess tildeles ulike prosesser og utføres samtidig. En multithreaded prosess oppnår samtidighet uten overhead av flere prosesser. Figuren viser trådtilstandene hos Windows.



Solaris UNIX (s. 202-206)

Solaris bruker også moderne prosesshåndtering med tråder, multiprosessering og mikrokjerne. Det bruker **fire separate trådrelaterte konsepter**:

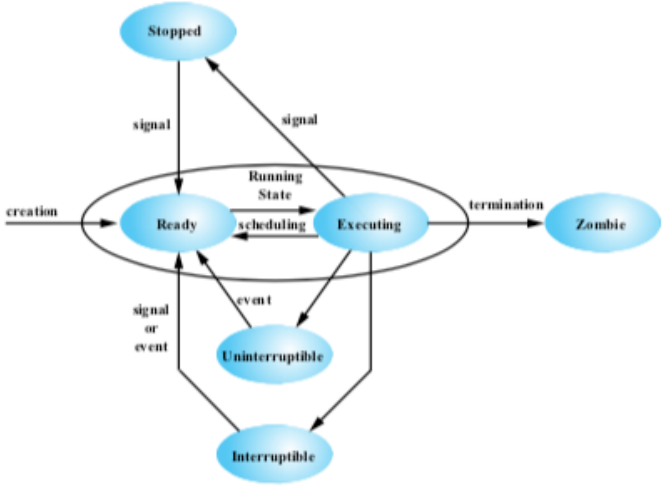
1. (**Tungvekts-) Prosess** – vanlig UNIX prosess som inkluderer bruker adresserom, stakk og prosesskontrollblokk.
2. **Brukertråd (ULT)** – implementert via et trådbibliotek i adresserommet til prosessen (usynlig for OS).
3. **Lettvekts-Prosess (LWP)** – en kartlegging mellom ULT og KLT. Hver LWP støtter ULT og kartlegger til en kjernetråd (dvs. LWP er en-til-en bundet til KLT). Disse blir tidsstyrt av kjernen og kan utføres parallelt på multiprosessorer.
4. **Kjernetråd (KLT)** – grunnleggende entiteter som kan tidsstyres og utsendes for å kjøre på en av systemprosessorene. Det eksisterer en KLT per LWP.



Basert på disse egenskapene kan man se at samtidskjøring og utførelse er håndtert på kjernenivået. Applikasjonen har i tillegg tilgang til hardware gjennom et API, og dette kan brukes for å utføre privilegerte oppgaver som lese/skrive til fil, danne nye prosesser, tildele minne, osv.

LINUX (s. 206-211)

LINUX har delvis arvet fra moderne prosess- og trådkonsepter, og det har en unik løsning som **ikke skiller mellom tråder og prosesser**. Ved å bruke en mekanisme lignende LWP i Solaris, blir ULT kartlagt til kjernenivå-prosesser. En ny prosess i LINUX dannes ved å kopiere attributtene til en annen prosess. En prosess kan også klones slik at den deler ressurser, slik som filer, signalbehandlere og virtuelt minne (s. 208). **Når to prosesser deler virtuelt minne, fungerer de som tråder innenfor en bestemt prosess.** Figuren viser prosess/tråd modellen.

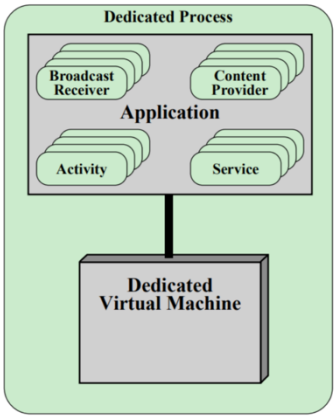


Til hver prosess i Linux er det assosiert et navnerom (s. 209). Navnerommet deler kjerneressurser inn i kontrollgrupper, slik at prosesser innenfor et navnerom ser ett sett med ressurser, mens prosesser i et annet navnerom ser et annet sett med ressurser. Dette former konseptet for en **virtuell maskin**, der prosesser har illusjonen at de er den eneste prosessen i systemet, noe som **støtter utnyttelse av multikjernesystem.**

Android (s. 211-215)

Android benytter seg også av moderne prosess- og trådkonsepter med arv fra UNIX. Android bygger på konseptet om applikasjoner og aktiviteter. Enhver Android-applikasjon består av en eller flere instanser som videre består av en eller flere komponenter. Hver komponent har en ulik rolle og kan aktiveres i eller utenfra applikasjonen. De ulike komponentene er:

- **Aktiviteter** – et vindu som er synlig i brukergrensesnittet som brukeren kan interagere med. For eksempel kan e-post applikasjonen ha et vindu som viser listen av nye e-poster som en aktivitet. Aktivitetene ordnes i en LIFO-kø, med de nyeste brukte aktivitetene først.
- **Tjenester** – en tjeneste utfører langtids operasjoner i bakgrunnen og har ikke et brukergrensesnitt. Dette sørger for raskere responstid fra main-tråden som brukeren direkte interagerer med.
- **Content provider** – fungerer som et grensesnitt til applikasjonsdataen for å for eksempel hente eller lagre data
- **Broadcast receiver** – responderer til systembrede kringkastingskunngjøringer. Dette kan være fra andre applikasjon som for eksempel skal gi beskjed om at data har blitt lastet ned og er tilgjengelig for å brukes av applikasjonen



Hver applikasjon kjører i en bestemt prosess som omfatter applikasjonen og en dedikert virtuell maskin for utnyttelse av multikjerner (se figur, s. 212). Dette **isolerer** hver applikasjon, slik at den ikke kan aksessere ressursene til en annen applikasjon uten tilgang (større sikkerhet/trygghet). Hver applikasjon fungerer dermed som en separat LINUX-bruker.

Hver applikasjon tildeles en prosess og en tråd til å begynne med som komponentene kjører i (s. 214). Utviklere kan lage flere tråder innenfor en prosess og/eller flere prosesser i en applikasjon for å oppnå økt ytelse. I begge tilfellene vil alle prosessene og deres tråder til applikasjonen kjøre i den samme virtuelle maskinen. For å frigjøre minne i et system kan man terminere prosesser. Det er bestemt et presedenshierarki for hvilke prosesser som skal termineres først. I synkende orden er dette: forgrunnsprosesser, synlige prosesser, service prosesser, bakgrunnsprosesser og tomme prosesser (dvs. tomme prosesser termineres først).

MAC OS (s. 215-217)

MAC IOS bruker moderne prosess- og trådkonsepter uten noen spesiell arv og har et eget sentralt støtte-konsept for utnyttelse av multikjerner som benytter Grand Central Dispatch (GCD istedenfor virtuell maskin). GCD oppretter en tråd-pool med tilgjengelige tråder som er lettere å bruke og mer effektiv enn andre OS. **Designere kan dele opp applikasjoner i blokker som kan kjøres samtidig for å utnytte multikjerner**. En blokk er en enkel utvidelse av C eller andre språk som C++, og formålet med å definere en blokk er å definere en selvstendig enhet for arbeid. Blokker lar utvikleren innkapsle komplekse funksjoner sammen med argumenter og data, slik at de kan refereres og sendes rundt i programmet på lignende måte som variabler. Blokkene legges i en FIFO-kø (s. 215). Bruken av forhåndsdefinerte tråder gir bedre ytelse for prosessering av blokker. Størrelsen på tråd-poolene vil automatisk justeres av systemet for å maksimere ytelsen til applikasjonen som bruker GCD samtidig som det minimerer antall inaktive tråder (s. 216).

Del 3 – Synkronisering av prosesser

Denne delen av kompendiet ser på prosesser og tråder, og det inkluderer:

- **Kapittel 5** – Samtidighet: gjensidig utelukkelse og synkronisering
- **Kapittel 6** – Samtidighet: vranglås og utsulting

Kapittel 5 – Samtidighet: gjensidig utelukkelse og synkronisering

Denne delen av kompendiet bruker en annen struktur enn boka, i et forsøk på å oppnå mer forståelse for sammenhenger i teorien.

Samtidighet (parallellitet) (s. 224)

Samtidighet er grunnleggende for OS-design og det omfatter problemer innenfor kommunikasjon mellom prosesser, deling og konkurranse om ressurser, synkronisering av aktiviteter og tildeling av prosesseringstid. Samtidighet er essensielt for:

- **Multiprogrammering** – håndtering av flere prosesser i et uniprosessorsystem. Dette er ikke reell parallellitet, men har lignende utfordringer som multiprosessering.
- **Multiprosessering** – håndtering av flere prosesser i et multiprosessorsystem. Dette er reell parallellitet
- **Distribuert prosessering** – håndtering av flere prosesser som utføres på flere, distribuerte datasystem. Går ikke i dybden i dette temaet.

Tabellen under viser viktige begrep relatert til samtidighet

Begrep	Beskrivelse
Atomiske operasjon	En operasjon er implementert som en sekvens av instruksjoner, der sekvensen er udelelig (hele utføres eller ikke i det hele tatt). Atomisitet garanterer isolasjon fra samtidige prosesser
Kritisk seksjon	En del av koden i en prosess som krever tilgang til delte ressurser, slik at den ikke kan utføres samtidig av to ulike prosesser
Vranglås (deadlock)	En situasjon der to eller flere prosesser ikke klarer å fortsette fordi begge venter på at den andre skal gjøre noe
Livelock	En situasjon der to eller flere prosesser kontinuerlig endrer deres tilstand i respons til endringer i andre prosesser, uten å gjøre noe nyttig arbeid
Gjensidig utelukkelse	Når en prosess er i en kritisk seksjon som aksesserer delte ressurser, kan ingen andre prosesser være i en kritisk seksjon som aksesserer noen av disse delte ressursene.
Race condition	En situasjon der flere tråder eller prosessorer leser og skriver en delt dataenhet og endelig resultat avhenger av relativ timing av utføringene
Utsulting	En kjørbar prosess blir uendelig oversett av scheduler. Selv om den er klar til å fortsette, blir den aldri valgt.

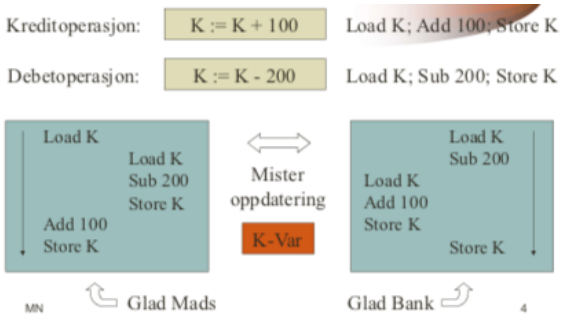
5. 2 Utfordringer ved samtidighet/parallellitet (s. 232)

Problemet for multiprogrammering og multiprosessering er at man ikke kan vite den relative hastigheten til utførelsen av en prosess. Denne hastigheten vil avhenge av aktiviteten hos andre prosesser, hvordan OS håndterer avbrudd og tidsstyringen til OS. Utfordringer som kan oppstå ved samtidig utføring av prosesser er:

1. **Det blir vanskelig å dele globale ressurser**, fordi hvis flere prosesser kan lese og skrive samme globale variabel, så vil rekkefølgen bli kritisk (= race condition)

2. **Det er utfordrende for OS å optimalt tildele ressurser**, for eksempel kan prosess som får aksess bli suspendert før den får tilgang. Ikke-optimal tildeling kan føre til vranglås eller utsulting
3. **Det blir veldig vanskelig å lokalisere feil,** siden oppførselen som regel ikke er deterministisk og reproduserbar

Figuren viser et eksempel der flere prosesser forsøker å skrive og lese verdier i bankkontoer. Rekkefølgen av handlingene vil avgjøre hvilket resultat de ulike prosessene oppnår. **Disse problemene kan unngås ved å oppnå gjensidig utelukkelse.** Vi ser nærmere på utfordringene tilknyttet samtidighet.



Dette problemet kalles Local Procedure Call (LPC) og er et tilfelle av Race condition. Se Appendix A.1 for hvordan det kan løses vha. semaforer. Det kan også løses med monitor som har to begrenset buffere (dvs. kø for prosesser som venter på betingelse (s. 45).

Eksempel – kontroll av tilgang til delte ressurser (s. 233)

Dersom flere applikasjoner bruker et program, vil dette være en delt ressurs som er lastet inn i den globale delen av minnet for alle applikasjoner (kun én kopi = sparer minne). **Deling av hovedminnet mellom prosesser er nyttig for å tillate effektiv og nær interaksjon mellom prosesser, men det kan også føre til problemer med parallellitet**. For eksempel kan en prosess oppdatere en global variabel og deretter bli forstyrret. En annen prosess kan så endre variabelverdien, før den første prosessen bruker variabelen. Dette vil resultere i at prosessen bruker feil verdi, og det skyldes at begge prosessene har utført sin **kritiske seksjon** som bruker den delte ressursen. Dette kan unngås ved å kun la én prosess være i kritisk seksjon om gangen. **For å beskytte delte globale ressurser må man altså kontrollere koden som aksesserer ressursene**. Dette vil gjelde både multiprogrammering og multiprosessorsystem.

Race condition (kappløpsbetingelse) (s. 235)

En race condition oppstår når flere prosesser eller tråder leser og skriver dataenheter, slik at endelig resultat avhenger av rekkefølgen til utføringen av instruksjoner i prosessene. Et eksempel er to prosesser P1 og P2 som deler global variabel a. P1 oppdaterer a til 1, mens P2 oppdaterer a til 2. De to prosessene er i et kappløp om å skrive variabel a og taperen (dvs. den som kommer sist) vil bestemme den endelige verdien. Dette kan også oppstå indirekte dersom verdien til en variabel avhenger av andre variabelverdier.

Utfordringer knyttet til OS (s. 235)

I sammenheng med samtidighet og synkronisering, vil OS ha følgende roller:

1. **Holde kontroll over de ulike prosessene** vha prosesskontrollblokkene
2. **Tildele og fradele ressurser til aktive prosesser** og håndtere at flere prosesser ønsker tilgang til samme ressurs samtidig. Disse ressursene inkluderer prosesseringstid, minne, filer og IO-enheter.
3. **Beskytte data og fysiske ressurser** hos en prosess mot forstyrrelser fra andre prosesser.
4. **Sørge for at funksjonaliteten til en prosess og output den produserer er uavhengig av hvor rask prosessen utføres sammenlignet med andre parallelle prosesser**.

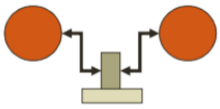
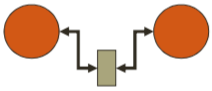
Former for parallellitet – typer prosessinteraksjon (s. 236)

For å forstå hvordan man kan håndtere utfordringene ved samtidighet, må vi først se på de ulike måtene prosesser kan interagere med hverandre. **Interaksjonen mellom prosesser kan klassifiseres basert på i hvilken grad de er klar over eksistensen til hverandre:**

- **Prosesser uvitende om hverandre** – prosessene er uavhengige og det er ikke meningen at de skal arbeide sammen. De vil **konkurrere** om ressursene og resultatet til en prosess er uavhengig av handlingene til andre prosesser. Timingen kan derimot påvirkes. To uavhengige applikasjoner kan ønske å aksessere samme ressurs, så OS må regulere aksessene.

- **Prosesser indirekte vitende om hverandre** – prosessene vil ikke nødvendigvis kjenne prosess-ID til andre prosesser, men de deler tilgang til noen ressurser (eks: IO-buffer). Resultatet til en prosess kan være avhengig av handlingene til andre prosesser og timingen kan også blir påvirket. Prosessene **samarbeider** ved å dele felles ressurser
- **Prosesser direkte vitende om hverandre** – prosessene kjenner prosess ID til hverandre, og de er laget for arbeide sammen om en aktivitet. Resultatet til en prosess kan være avhengig av handlingene til andre prosesser og timingen kan også blir påvirket. Prosessene **samarbeider** ved å kommunisere med hverandre.

Noen prosesser vil vise aspekter ved både konkurranse og samarbeid. Tabellen gir mer informasjon om de ulike interaksjonstypene.

Type	Bevissthet	Beskrivelse	Figur	Potensielle problemer
Uavhengighet	Uavhengige	Prosessene aksesserer ikke noen delte ressurser.		
Konkurranse	Prosesser uvitende om hverandre	Parallelle prosesser vil komme i konflikt når de konkurrerer om å bruke samme ressurs. Prosessene trenger å aksessere ressursene i løpet av utføringen og de er uvitende om eksistensen til andre prosesser. Det er ingen utveksling av informasjon mellom konkurrerende prosesser, men utføringen av en prosess kan påvirke oppførselen til konkurrerende prosesser. Hvis to prosesser ønsker tilgang til en ressurs, vil den ene gis tilgang av OS, mens den andre må vente. Prosessen som ikke får tilgang blir tregere.		Gjensidig utelukkelse Vranglås Utsulting
Samarbeid via deling	Prosesser indirekte vitende om hverandre	Samarbeid ved deling omfatter prosesser som interagerer med andre prosesser uten å være eksplisitt klar over dem. Prosesser må samarbeide om å sikre at den delte dataen blir riktig kontrollert. I samarbeid ved deling må man også håndtere problemene med gjensidig utelukkelse, vranglås og utsulting, men det er kun skriveoperasjonene som må være gjensidig utelukket (ikke lesing)		Gjensidig utelukkelse Vranglås Utsulting Datakoherens (s. 239)
Samarbeid via kommunikasjon	Prosesser direkte vitende om hverandre	Når prosessene samarbeider ved kommunikasjon, vil prosessene delta i en felles innsats som kobler sammen alle prosessene. Kommunikasjonen legger til rette for **synkronisering av aktiviteter**. Kommunikasjonen innebærer at prosessene sender og mottar meldinger, og siden ingenting deles i disse meldingene er det ikke noe behov for gjensidig utelukkelse ved denne typen samarbeid. Utfordringene med vranglås og utsulting er fortsatt aktuelle. Et eksempel på vranglås er at begge prosessene kan blokkeres ved at de venter på melding fra den andre prosessen. Et eksempel på utsulting er når P3 ikke får kommunisere med P1, fordi P1 er opptatt i gjentatt kommunikasjon med P2.		Vranglås Utsulting

Gjensidig utelukkelse

Krav for gjensidig utelukkelse (s. 240)

Noen krav for funksjonalitet som støtter gjensidig utelukking er:

1. Gjensidig utelukkelse må håndheves, som vil si at **blant prosesser som har kritiske seksjoner for samme ressurs, må kun én prosess få tilgang om gangen**. Kritiske regioner kan ikke overlappe
2. **En prosess skal ikke forhindre andre prosesser**, så hvis en prosess pauser sin ikke-kritiske seksjon, kan ikke dette påvirke andre prosesser

3. **En prosess som ønsker tilgang til sin kritisk seksjon, kan ikke utsettes uendelig** (dvs. ingen vranglås eller utsulting)
4. Når ingen prosesser er i en kritiske seksjon, kan enhver prosess som ønsker det **få aksess uten forsinkelse** (dvs. ingen unødvendige forsinkelser)
5. Det gjøres **ingen antagelser** om relative hastigheter hos prosesser eller antall prosessorer
6. **Prosesser vil bruke en endelig tid i sine kritiske seksjoner** (dvs. begrenset tid)

Disse kravene kan tilfredsstilles på ulike måter. Vi skal se på hvordan det kan oppnås vha programvare- og maskinvareløsninger som bruker aktiv venting (*busy waiting*). Deretter vil vi se på hvordan det oppnås vha tre ulike synkroniseringsverktøy: semaforer, monitorer og meldinger.

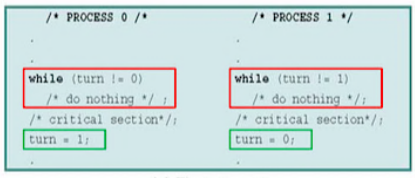
5.1 Gjensidig utelukkelse – programvarebasert løsning

For maskiner med en eller flere prosessorer og delt hovedminne kan parallelle prosesser implementeres med programvare. **Ved programvaretilnærmingen får prosessene ansvaret for å oppnå gjensidig utelukking ved å koordinere med hverandre (dvs. ingen støtte fra OS)**. Denne tilnærmingene antar gjensidig utelukkelse ved minneaksess-nivået, som vil si at samtidige aksesser til samme lokasjon vil serialiseres på en eller annen måte. Bortsett fra det er det ingen andre antagelser om maskinvare.

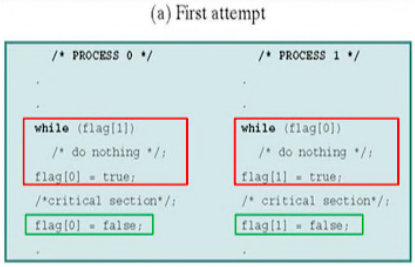
Dekkers algoritme

Dekkers algoritme gir gjensidig utelukkelse for to prosesser, og vi utvikler denne vha Dijkstras fire gale forsøk:

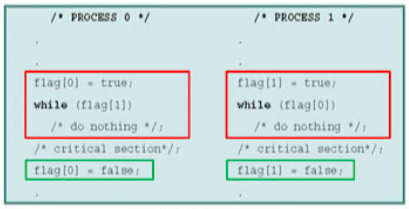
1. **Forsøk 1** – kontrollen har en streng alternering, der en global variabel (turn) brukes for å avgjøre hvilken prosess som kan utføre kritisk seksjon. Hvis verdien til turn ikke er lik nummeret til prosessen, må den vente på tillatelse (dvs. den andre prosessen blir ferdig og endrer turn). Dette kalles aktiv venting (*busy waiting*). Det vil garantere gjensidig utelukkelse, men tregeste prosess vil avgjøre hastighet og hvis en prosess feiler vil den andre prosessen bli permanent blokkert.
2. **Forsøk 2** – hver prosess har et boolean flagg som oppgir om prosessen er i kritisk seksjon. Når en prosess ønsker å utføre den kritiske seksjonen, vil den vente helt til ingen andre prosesser er utfører kritisk seksjon og deretter endre sitt flagg til *true*. Dette vil unngå permanent blokkering (med mindre prosessen feiler i kritisk seksjon), men det vil ikke garantere gjensidig utelukkelse (pga. ulik hastigheter)
3. **Forsøk 3** – prosessen setter flagget sitt lik *true* før den sjekker flagget til andre prosesser. Dette hindrer at prosess 1 kan endre sitt flagg i perioden mellom prosess 0 har sjekket flaggene og entret kritisk seksjon. Det garanterer gjensidig utelukkelse, men hindrer ikke blokkering hvis prosess feiler i kritisk seksjon, og det er utsatt for vranglås.
4. **Forsøk 4** – legger til mulighet for å resette flagget for å unngå vranglås. Det garanterer gjensidig utelukkelse, men er utsatt for livelock og utsulting.



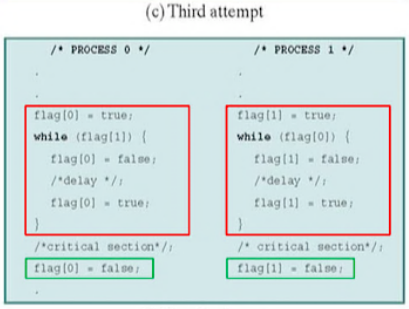
(a) First attempt



(b) Second attempt



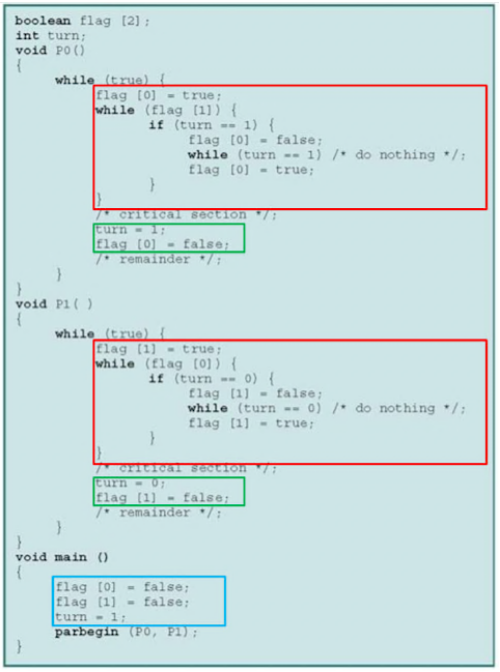
(c) Third attempt

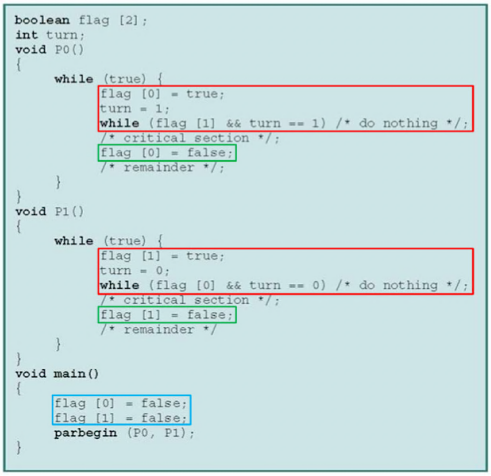


(d) Fourth attempt

Feilen til fjerde forsøk er at vi ikke har noen rekkefølge på aktivitetene til de to prosessene for å unngå «gjensidig høflighet». For å løse dette kan vi **bruke variabelen `turn` for å gi hvilken prosess som har rett til å entre sin kritiske seksjon**. Denne løsningen kalles **Dekkers**

algoritme (se figur). Når P0 ønsker å entre sin kritiske seksjon vil den først sette flagget sitt lik *true* og deretter sjekke flagget til P1. Hvis flag1 = *false* vil P0 umiddelbart entre den kritiske seksjonen, mens hvis flag1 = *true* vil den sjekke turn. Hvis turn = 0 vil P0 vite at den har rett til å insistere på å entre kritisk seksjon og vil derfor periodisk sjekke flagget til P1. Ved et øyeblikk vil P1 oppdage at P0 har satt flagget lik *true* og siden turn = 0, vil den sette sitt flagg til *false*, slik at P0 får fortsette. Når P0 er ferdig med sin kritiske seksjon vil den sette sitt flagg til *false* og endrer turn = 1, slik at P1 får førsterett til å entre kritisk seksjon. På figuren vil parbegin(P0, P1) bety at utføring av main-programmet utsettes og parallell utføring av P0 og P1 blir initiert. Main-programmet blir gjenopptatt når P0 og P1 har terminert. **Denne algoritmen er lett å utvide dersom man har flere prosesser (N > 2).**





Petersons Algoritme – alternativ løsning

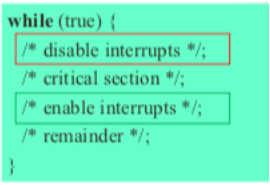
Dekers algoritme gir gjensidig utelukkelse, men **Petersons algoritme gir en enklere og mer elegant løsning** (se figur). Denne bruker også array flag for å gi tilstanden til hver prosess og variabelen turn for å løse samtidighetskonflikter. Gjensidig utelukkelse oppnås fordi hvis P0 har satt flagget lik *true*, så kan ikke P1 entre sin kritiske seksjon (og motsatt). Hvis P1 allerede er i sin kritiske seksjon, så vil flag1 = *true* og P0 blokkeres fra å entre sin kritiske seksjon. Algoritmen unngår også gjensidig blokkering. Hvis P0 blokkeres mens den er i while-løkken, vil flag1 = *true* og turn = 1, og P0 må vente helt til minst en av disse endres. Vi vet at P1 ønsker å entre kritisk seksjon og lykkes med dette, siden flag1 = *true* og turn = 1. I tillegg vet vi at P1 ikke kan gjentatt entre kritisk seksjon, slik at P0 ikke slipper til, fordi P1 vil sette turn = 0 før hvert forsøk på å entre kritisk seksjon. Dermed vil P0 få mulighet til å entre kritisk seksjon, og algoritmen unngår «gjensidig høflighet». **Ulempen med Petersons Algoritme er at det er vanskeligere å utvide til flere prosesser (N > 2).**

OBS

Selv om programvarebaserte løsninger kan garantere gjensidig utelukkelse vha aktiv venting, så vil dette være ineffektivt pga høyt prosessering overhead og bugs. (s. 240).

5.3 Gjensidig utelukkelse – maskinvarebasert løsning

Maskinvarebasert løsning kan brukes for korte, kritiske regioner (lavest nivå), mens programvarebasert løsning kan brukes for lengre kritiske regioner (dvs. høyere nivå). De korte kritiske regionene er vanligere og maskinvarebasert løsning er mer "effektivt".



Avbrudd-deaktivering (*interrupt disabling*)

I et uniprosessorsystem med multiprogrammering er det tilstrekkelig å forhindre avbrudd i en prosess for å garantere gjensidig utelukkelse. Avbrudd kan aktiveres eller deaktiveres i OS-kjernen, og figuren viser hvordan en prosess kan oppnå gjensidig utelukkelse. Gjensidig utelukkelse er garantert siden den kritiske seksjonen ikke kan avbrytes. Dette kan likevel føre til en **signifikant reduksjon i effektiviteten** til utførelsen, siden prosessoren får en begrenset mulighet til å sammenflette prosesser. **Denne løsningen vil heller ikke fungere for et multiprosessor system (dvs. kun anvendbar på uniprosessorsystem).** Når systemet har flere prosessorer, kan flere prosesser utføres samtidig og gjensidig utelukking er dermed ikke garantert ved å deaktivere avbrudd.

Spesielle maskininstruksjoner

På maskinvarenivå vil aksess til minnelokasjon utelukke enhver annen aksess til samme lokasjon. Det har blitt utviklet flere maskininstruksjoner som bruker dette for å utføre to handlinger atomisk (dvs. uten forstyrrelser), slik som lesing og skriving av en minnelokasjon. **Ved utføringen av en instruksjonen vil aksess til minnelokasjonen blokkeres for alle andre instruksjoner som refererer til denne lokasjonen**. Fordelen med å forhindre avbrudd er at det fungerer for uni- og multiprosessorer som deler hovedminnet. Metoden er enkel, noe som gjør den lett å verifisere. Den vil også støtte flere kritiske seksjoner, siden hver kritisk seksjon kan defineres som en egen variabel. **Ulempen med å forhindre avbrudd er at prosessen som venter på tilgang til kritisk seksjon blir satt i aktiv venting, noe som forbruker prosessortid** (den vil kontinuerlig sjekke om den kan få tilgang). Det er også utsatt for utsulting, fordi når kritisk seksjon blir ledig og flere prosesser venter, vil valg av prosess være tilfeldig, noe som kan føre til at en prosess aldri får tilgang. Dersom man innfører prioritetsnivå vil det også være utsatt for vranglås (*deadlock*).

To av de mest vanlige instruksjonene er:

Merk: kalles TestSet instruksjon i forelesning

1. **Compare&Swap instruksjon (CAP)** – en atomisk instruksjon med to steg: sammenligning og bytte. Ved sammenligning vil den sjekke om en minneverdi (*word) er lik en testverdi (testval), og hvis det er tilfellet vil den bytte minneverdien med en ny verdi (newval). Den gamle minneverdien blir alltid returnert, og hvis denne er lik testverdien vil minnelokasjonen være oppdatert. Hele funksjonen utføres atomisk, altså vil den ikke avbrytes. Dette kan brukes for å oppnå gjensidig utelukkelse (se figur). En delt variabel bolt initialiseres til 0, og det er kun prosessen som finner bolt = 0 som får entre kritiske seksjon. Andre prosesser må **aktivt vente (*busy waiting*)**. Når prosessen forlater kritisk seksjon vil den sette bolt til 0 og én av ventende prosesser får entre kritisk seksjon. Valg av prosess er den som utfører CAP først. Merk: CAP brukes for å oppnå gjensidig utelukkelse ved at det sikrer at kun én prosess får til å oppdatere bolt til 1.

```
int compare_and_swap (int *word, int testval, int newval)
{
    int oldval;
    oldval = *word
    if (oldval == testval) *word = newval;
    return oldval;
}
```

```
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
  while (true) {
    while (compare_and_swap(&bolt, 0, 1) == 1)
        /* do nothing */;
    /* critical section */;
    bolt = 0;
    /* remainder */;
  }
}
void main()
{
  bolt = 0;
  parbegin (P(1), P(2), . . . ,P(n));
}
```

2. **Exchange instruksjon** – bytter innholdet i et register med innholdet i en minnelokasjon. Dette kan brukes i protokoll som oppnår gjensidig utelukkelse (se figur). En delt variabel bolt initialiseres til 0, og hver prosess bruker en lokal variabel key som er initialisert til 1. Det er kun prosessen som finner bolt = 0 som får entre kritiske seksjon og den ekskluderer andre prosesser ved å sette bolt = 1. Når den forlater kritiske seksjon vil prosessen sette bolt = 0. Merk: exchange-metoden vil ha oppførselen $\text{bolt} + \sum_i \text{key}_\text{i} = n$, slik at ved bolt = 1 vil det være kun én prosess med key = 0 som er i kritiske seksjon (slik at 1 + 1 + 1 +...+ 0 = n). Exchange instruksjonen sikrer altså at det er kun en prosess som får mulighet til å endre bolt til 1 og entre kritisk seksjon.

```
void exchange (int *register, int *memory)
{
    int temp;
    temp = *memory;
    *memory = *register;
    *register = temp;
}
```

```
/* program mutualexclusion */
int const n = /* number of processes*/;
int bolt;
void P(int i)
{
  while (true) {
    int keyi = 1;
    do exchange (&keyi, &bolt)
    while (keyi != 0);
    /* critical section */;
    bolt = 0;
    /* remainder */;
  }
}
void main()
{
  bolt = 0;
  parbegin (P(1), P(2), . . ., P(n));
}
```

OBS

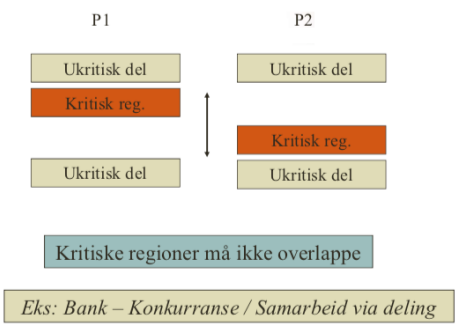
Det er signifikante ulemper ved både programvarebasert og maskinvarebasert løsning, så derfor ser vi på andre mer effektive og generelle mekanismer innenfor OS og programmeringsspråk som brukes for å synkronisere prosesser og dermed sikre at samtidig utføring gir riktig resultat.

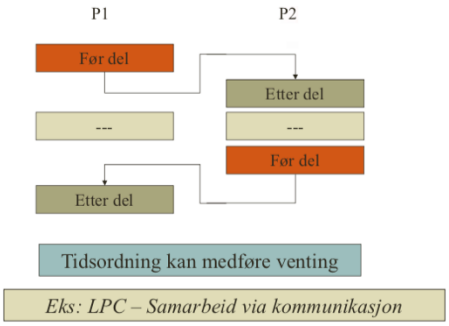
Merk: synkronisering krever også støtte for trygg og effektiv håndtering av vranglås og utsulting (kapittel 6)

Synkronisering og gjensidig utelukkelse

Prosessynkronisering innebærer å ordne tiden til hendelsene i de ulike prosessene, slik at man sikrer riktig oppførsel og riktig resultat av aktiviteter som utføres i parallell. Dette kan oppnås ved å sikre **gjensidig utelukkelse** mellom prosessene som aksesserer en felles ressurs (dvs. kun én prosess i kritisk seksjon om gangen) eller det kan oppnås ved at prosessene samarbeider via kommunikasjon for å oppnå en **felles tidsordning** av hendelser. Figurene under illustrerer de to ulike formene for synkronisering.

Merk: på mikronivå vil synkronisering innebære **aktiv venting** (siden metodene baseres på programvare- eller maskinvarebasert løsning), mens på makronivå vil det innebære **passiv venting.**





Merk: alle verktøyene vil implementere både gjensidig utelukkelse og generell tidsordning

Synkronisering sikrer konsistent oppførsel og hindrer at parallell utføring av programmer eller delprogrammer gir gale resultater, ved at sørge for at enkeltoperasjoner ordnes i riktig rekkefølge og at ulike sett kan utføres uten overlapp. Dvs. synkroniseringen øker tryggheten til systemet. **Det er tre ulike verktøy som brukes for å oppnå prosessynkronisering**:

1. **Semaforer** – generelt og lavnivåverktøy, som det er lett å gjøre feil med. Det kan gi plassbesparende løsninger og brukes ofte for å implementere andre høynivåvirktøy.
2. **Monitorer** – spesifikke og høynivåverktøy, som gjør det lettere å vise korrekthet. De vil implementere gjensidig utelukkelse automatisk, så de brukes ofte i tilfeller der gjensidig utelukkelse er et essensielt behov
3. **Meldinger** – brukes for å la to prosesser utveksle informasjon, slik at de kan synkroniseres. Det kan implementere generell tidsordning svært direkte, så det brukes ofte i tilfeller der generell tidsordning er et essensielt behov. Det er kun meldinger som kan brukes i distribuerte system i tillegg til sentraliserte.

De tre verktøyene er funksjonelt sett likeverdige ved at de kan løse samme problem ved et system med delt lager. Det vil likevel være noe forskjell i anvendelighet, slik at enkelte verktøy passer bedre for bestemte formål. Prosesser kan kontrolleres med semaforer, monitorer eller meldinger for å sikre at kun én prosess er inni gjensidig avhengige kritiske regioner om gangen og at deres tilhørende aktiviteter ordnes i tid, slik at tilsvarende overordnete krav overholdes (E). Vi ser nærmere på disse verktøyene.

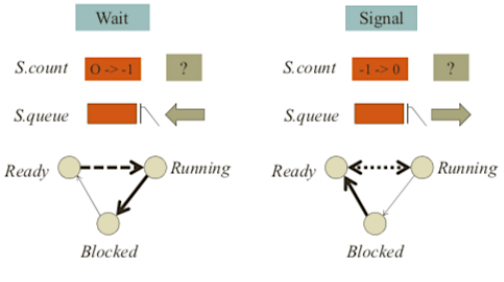
5.4 Semaforer

Semaforer:
- Gjensidig utelukkelse – lett å gjøre feil, vanlig å bruke med monitor
- Generell tidsordning – kun telling av tilgjengelige spørsmål/svar og bufferslotter

En semafor er en heltallsverdi som brukes for å sende signaler via prosesser. **Det kan utføres tre atomiske operasjoner på en semafor: *initialize*, *decrement* og *increment*.** Decrement-operasjonen kan resultere i blokkering av prosessen, mens inkrement-operasjonen kan resultere i avblokkering av prosessen. **Semaforer meget generelle og lavnivåverktøy som det er lett å gjøre feil med.** Det kan gi plassbesparende løsninger og brukes ofte for å implementere andre høynivåvirktøy.
Semaforer meget generelle og lavnivåverktøy som det er lett å gjøre feil med. Det kan gi plassbesparende løsninger og brukes ofte for å implementere andre høynivåvirktøy.

Generell semafor (tellende semafor)
Det grunnleggende prinsippet er at en prosess kan tvinges til å stoppe ved en spesifikk lokasjon, helt til den mottar et bestemt signal. **De tre operasjonene defineres som følger:**

1. **Initialisering** – semaforen settes lik et ikke-negativt heltall.



2. **Decrement** - semWait operasjonen utføres for å motta et signal via semaforen og det innebærer at verdien til semaforen blir redusert. Hvis verdien blir negativ vil prosessen som utfører semWait blokkeres, hvis ikke vil prosessen fullføre utføringen.
3. **Increment** - semSignal operasjonen utføres for å sende et signal via semaforen og det innebærer at verdien til semaforen blir økt. Hvis resulterende verdi er mindre eller lik 0 vil det bety at minst en prosess har tidligere utført semWait og blitt blokkert, så en av disse kan avblokkeres.

```
struct semaphore {
     int count;
     queueType queue;
};
void semWait(semaphore s)
{
     s.count--;
     if (s.count < 0) {
          /* place this process in s.queue */;
          /* block this process */;
     }
}
void semSignal(semaphore s)
{
     s.count++;
     if (s.count <= 0) {
          /* remove a process P from s.queue */;
          /* place process P on ready list */;
     }
}
```

Det er ingen andre måter å inspisere eller manipulere verdien til semaforen. På figuren antas semWait og semSignal å være atomiske. I begynnelsen vil semaforen være 0 eller antall prosesser som kan utføre en semWait og umiddelbart fortsette utføring (dvs. antall prosesser som kan utføres samtidig). Hvis semaforen blir negativ må prosessen vente på at de andre prosessene skal fullføres, og størrelsen til den negative verdien gir antall prosesser som er blokkert (dvs. plassert i ventende kø). Prosessen de venter på vil etterhvert kalle semSignal(s) for å øke s, og da vil en av de ventede prosessene avblokkeres. Når semaforen er negativ vil det altså bety at en eller flere prosesser venter, og en av disse blir avblokkert når et signal fra utførende prosesser blir sendt. Tre konsekvenser av definisjonen: (1) man vil ikke vite på forhånd om en prosess som reduserer semaforen vil blokkeres eller ikke, (2) hvis en prosess sender et signal som vekker en annen, vil man ikke vite hvilken som får fortsette på uniprosessorsystem og (3) ved sending av signal vet man ikke om noen prosesser venter.

Binær semafor

En binær semafor er en semafor som kun tar 1- og 0-verdier. De tre operasjonene defineres som følger:

1. **Initialisering** – semaforen settes lik 0 eller 1
2. **Decrement** - semWaitB operasjonen utføres for å motta et signal via semaforen og det innebærer at verdien til semaforen blir sjekket. Hvis verdien er 0 vil prosessen blokkeres, mens hvis den er 1 vil den endres til 0 og prosessen utføres.
3. **Increment** - semSignal operasjonen utføres for å sende et signal via semaforen og det innebærer at det sjekkes om noen prosesser er blokkert på denne semaforen (dvs. lik 0). Hvis en prosess er blokkert blir denne avblokkert, hvis ikke vil semaforen settes lik 1.

Merk: blokkerte prosesser plasseres i kø

```
struct binary_semaphore {
     enum {zero, one} value;
     queueType queue;
};
void semWaitB(binary semaphore s)
{
     if (s.value == one)
          s.value = zero;
     else {
          /* place this process in s.queue */;
          /* block this process */;
     }
}
void semSignalB(binary semaphore s)
{
     if (s.queue is empty())
          s.value = one;
     else {
          /* remove a process P from s.queue */;
          /* place process P on ready list */;
     }
}
```

Den er enklere å implementere og har samme utrykningskraft. **Ikke-binær semafor kalles ofte tellende eller generell semafor.**

Mutual exclusion lock (mutex)

Mutex er et konsept som er relatert til den binære semaforen, og det er et binært flagg som brukes for å låse og frigi et objekt. Den settes til låst (typisk 0) dersom et objekt som ikke kan deles er tildelt en prosess, slik at objektet er blokkert fra andre prosesser som ønsker å bruke det. Forskjellen mellom mutex og binære semaforer er at prosessen som låser mutexen må også være den som låser den opp. For en binær semafor kan en prosess låse den binære semaforen, mens en annen låser den opp.

Sterke vs. svake semaforer

For både binær og tellende semafor blir det brukt en kø for å holde prosessene som venter på semaforen. Vi skiller mellom:

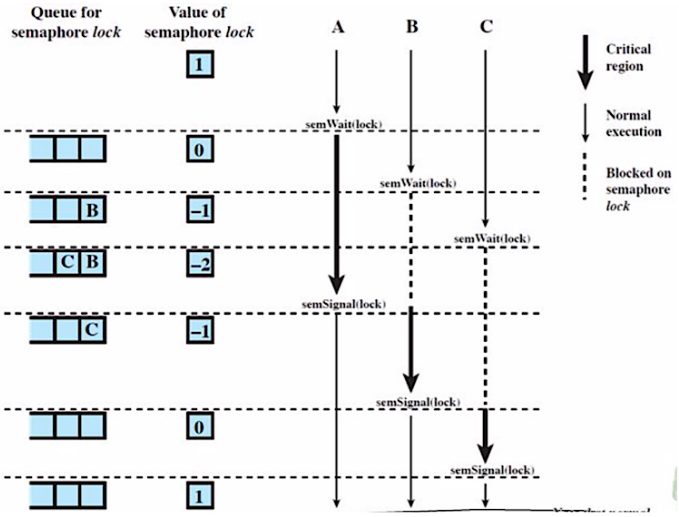
- **Sterk semafor** – køen er en FIFO kø, der prosessen som har vært blokkert lengst blir avblokkert først, slik at man unngår utsulting.
- **Svak semafor** – det er ikke spesifisert hvilken prosess som blir avblokkert først, noe som kan resultere i utsulting (en prosess får aldri tilgang)

Se s. 247-248 for eksempler på dette. OS vil som regel bruke sterke semaforer.

Gjensidig utelukkelse med semafor

```
/* program mutualexclusion */
const int n = /* number of processes  */;
semaphore s = 1;
void P(int i)
{
     while (true) {
          semWait(s);
          /* critical section   */;
          semSignal(s);
          /* remainder   */;
     }
}
void main()
{
     parbegin (P(1), P(2), . . ., P(n));
}
```

Figuren viser en løsning for å oppnå gjensidig utelukkelse vha. en semafor *s*. Programmet ser på *n* prosesser som er plassert i listen $P(i)$ og har behov for tilgang til samme ressurs. Hver prosess har en kritisk seksjon som brukes for å aksessere ressursen. I hver prosess vil semWait(s) utføres før den entrer sin kritiske seksjon, og hvis *s* blir negativ vil prosessen blokkeres. Hvis *s = 1* vil denne reduseres til 0 og prosessen entrer den kritiske seksjonen umiddelbart. Siden s ikke lenger er positiv vil ingen andre prosesser kunne entre sin kritiske seksjon. Når prosessen har fullført kritiske seksjon vil den øke semaforen og en av de blokkerte prosessene plasseres i Ready-tilstand og får utføre sin kritiske seksjon når de blir utsendt til Running-tilstand av OS. Dersom flere prosesser skal ha lov til å entre kritisk seksjon samtidig, kan dette implementeres ved å initialisere semaforen til en korresponderende verdi.



Figuren til venstre viser en mulig seksjon for tre prosesser som bruker semafor lock for å oppnå gjensidig utelukkelse. Legg merke til at normal utførelse kan fortsette i parallell, men at kritiske seksjoner må utføres i serie. For mer detaljert beskrivelse se s. 249.

Producer/consumer-problemet (s. 250)

Dette er et av de vanligste problemene som oppstår ved samtidig prosessering. **En eller flere prosesser (*producers*) genererer data til en bounded buffer, men en prosess (*consumer*) tar dataen ut av bufferen og forbruker den**. De kommuniserer via en buffer og har begrensningene:

1. Kun én bufferoperasjon kan skje av gangen (dvs. kun én agent kan aksessere bufferen av gangen for å hindre overlappende bufferoperasjoner)
2. Consumer må vente på producer hvis køen er tom
3. Producer må vente på consumer hvis køen er full

Problemet er altså å hindre at producer forsøker å legge til data i en full buffer eller at consumer forsøker å fjerne data fra en tom buffer. De to siste begrensningene krever en synkronisering mellom producer og consumer.

Merk: problemet kan ikke løses med en «rør»-implementasjon, siden det krever antagelser om hastigheten til prosessene

Semaforløsningen til problemet bruker tilstanden til tre semaforer: e som gir antall tomme plasser i køen, n som gir antall elementer i køen og s som sikrer integriteten til bufferen (eks: hindrer at to producers forsøker å legge til elementer til bufferen samtidig). **Vi ser på et eksempel for å illustrere bruken:**

1. En consumer entrer kritiske seksjon, men blokkeres av semWait(n) siden n = 0.
2. Flere producere ønsker å entre kritisk seksjon for å legge til dataelementer i køen, som kan inneholde maksimalt sizeofbuffer antall elementer. Siden det er noen tomme plasser (e ≠ 0) vil ikke producer blokkeres, og semaforen s brukes for å sikre at kun én producer får tilgang til bufferen av gangen. Producer som får tilgang legger til data til bufferen, før den øker s og n.
3. Siden producer har økt n vil det sendes et signal til ventende consumer, som så vil plasseres i Ready-tilstand. Semaforen s brukes for å sikre at consumer og producer ikke får tilgang til bufferen samtidig.

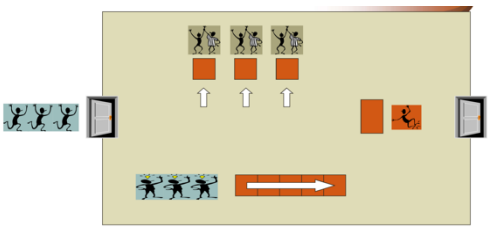
```
/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1, n= 0, e= sizeofbuffer;
void producer()
{
     while (true) {
          produce();
          semWait(e);
          semWait(s);
          append();
          semSignal(s);
          semSignal(n);
     }
}
void consumer()
{
     while (true) {
          semWait(n);
          semWait(s);
          take();
          semSignal(s);
          semSignal(e);
          consume();
     }
}
void main()
{
     parbegin (producer, consumer);
}
```

Merk: boka gir en mer detaljert beskrivelse på feil og riktig implementasjon ved bruk av binær semafor. Feil implementasjon innebærer at consumer kan ende opp med å forbruke en enhet fra bufferen som ikke eksisterer (s. 252). Dette kan løses ved å introdusere en lokal variabel som kan settes ila. kritiske seksjon eller bruke n som generell semafor (løsningen vi ser på). Se s 251-255 for mer detaljer

Merk: A.1 i appendix ser på hvordan semaforer kan brukes for å løse Race Condition

Barbersalong-problemet (Appendix A)

Barbersalong-problemet er et annet eksempel som illustrerer bruken av semaforer for å implementere samtidighet. Barbersjappen har tre stoler, tre barberere og et venterom med en sofa for fire og et stående rom for flere. Totalt antall kunder er begrenset til 20, og en kunde vil ikke entre sjappen hvis dette antallet er nådd. Når en barberer er ledig, vil kunden som har sittet lengst på sofaen får neste time og kunden som har stått lengst blir plassert på sofaen. Det er kun ett betalingssystem, så kun en kunde kan betale om gangen. Appendix nevner to løsninger:



1. **Urettferdig løsning (A.11)** – tabellen viser semaforene som brukes av løsningen. Denne løsningen har et timing problem som kan føre til urettferdig behandling av kunder. På grunn av måten køen er organisert, må kundene entre og forlate barberstolen i samme rekkefølge. Dersom en kunde blir fort ferdig kan dette føre til at kunden må vente i stolen eller at de andre kundene må forlate stolen selv om de ikke er ferdig. Andre problemer er at kunden ikke frigjøres med en gang betaling er mottatt, det kan hende en kunde tildeles en stol som ikke er tom (skyldes FIFO kø) og det krever at kunder setter seg i sofaen selv om det er en tom stol (se A.3 problemer)

Semaphore	Wait Operation	Signal Operation
max_capacity	Customer waits for space to enter shop.	Exiting customer signals customer waiting to enter.
sofa	Customer waits for seat on sofa.	Customer leaving sofa signals customer waiting for sofa.
barber_chair	Customer waits for empty barber chair.	Barber signals when that barber's chair is empty.
cust_ready	Barber waits until a customer is in the chair.	Customer signals barber that customer is in the chair.
finished	Customer waits until his haircut is complete.	Barber signals when cutting hair of this customer is done.
leave_b_chair	Barber waits until customer gets up from the chair.	Customer signals barber when customer gets up from chair.
payment	Cashier waits for a customer to pay.	Customer signals cashier that he has paid.
receipt	Customer waits for a receipt for payment.	Cashier signals that payment has been accepted.
coord	Wait for a barber resource to be free to perform either the hair cutting or cashiering function.	Signal that a barber resource is free.

Merk: dette viser at løsningen kan bli urettferdig selv med sterk semafor

2. **Rettferdig løsning (A.13)** – for å løse timing problemet der kunder får sitte for kort eller for lenge i barberstolen kan man introdusere flere semaforer. Bruken av et unikt kundenummer gjør at kunden som plasseres i barberstolen kan vente på sin egen unike semafor og når barberer er ferdig kan denne semaforen avblokkeres. Dermed slipper man å implementere barberstolene som en FIFO kø og kundene kan entre og forlate stolen uavhengig av hverandre.

Implementasjon av semaforer

Det er avgjørende at semWait og semSignal implementeres som atomiske operasjoner for å oppnå gjensidig utelukkelse, slik at kun en prosess av gangen kan endre verdien til en semafor. For å oppnå dette kan man bruke **Dekkers eller Petersons algoritme**, men det vil føre til en signifikant prosessering overhead. Et annet alternativ er å bruke en **maskinvarebasert løsning, slik som** CAP (figur a). Dette vil kreve aktiv venting, men siden semWait og semSignal er relativt korte vil mengden venting være liten. For uniprosessorsystem kan man **deaktivere avbrudd**, og siden semWait og semSignal er korte kan denne tilnærmingen brukes (figur b).

```
semWait(s)
{
    While (compare_and_swap(s.flag, 0 , 1) == 1)
        /* do nothing */;
    s.count--;
    if (s.count < 0) {
        /* place this process in s.queue*/;
        /* block this process (must also set s.flag to 0)
*/;
    }
    s.flag = 0;
}

semSignal(s)
{
    While (compare_and_swap(s.flag, 0 , 1) == 1)
        /* do nothing */;
    s.count++;
    if (s.count <= 0) {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
    s.flag = 0;
}
```

(a) Compare and Swap Instruction

```
semWait(s)
{
    inhibit interrupts;
    s.count--;
    if (s.count < 0) {
        /* place this process in s.queue*/;
        /* block this process and allow interrupts*/;
    }
    else
        allow interrupts;
}

semSignal(s)
{
    inhibit interrupts;
    s.count++;
    if (s.count <= 0) {
        /* remove a process P from s.queue */;
        /* place process P on ready list*/;
    }
    allow interrupts;
}
```

(b) Interrupts

For semaforer er altså ansvaret for gjensidig utelukkelse og synkronisering lagt på utvikleren. Semaforer er et enkelt, kraftig og fleksibelt verktøy for å oppnå gjensidig utelukkelse og koordinere prosesser. Det kan likevel være vanskelig å produsere et riktig program, siden semWait og semSignal er spredt rundt i programmet og det er utfordrende å se hvordan disse totalt sett påvirker semaforen.

Monitorer
- Gjensidig utelukkelse – automatisk gitt
- Generell tidsordning – ikke engang telling av tilgjengelige spørsmål/svar og bufferslotter

5.5 Monitorer

En monitor er en del av et programmeringsspråk som innkapsler variabler, aksesserer prosedyrer og initierer kode innenfor en abstrakt datatype. Monitorer vil utvide klassekonseptet med synkroniseringsformer. Variabler og metoder hos monitoren kan kun aksesseres gjennom aksessprosedyren til monitoren og kun én prosess kan aktivt aksessere monitoren om gangen. Dette sikrer gjensidig utelukkelse mellom metoder som utføres på vegne av prosesser. **Aksessprosedyrene er kritiske seksjoner og monitoren kan opprettholde en kø med prosesser som venter på aksess.**

Monitorer er mer høynivå enn semaforer, noe som gjør det lettere å verifisere korrekthet. Synkroniseringsfunksjonene er begrenset til monitoren, noe som gjør det lettere å verifisere at synkroniseringen er gjort riktig og detektere feil (kontra semaforer der semWait og semSignal er spredt over hele programmet). Monitorer har automatisk gjensidig utelukkelse, mens for semaforer må alle prosesser som aksessere ressursene være riktig programmert av utvikler.

Monitor med signal

Hovedegenskapene ved en monitor er følgende:

1. De lokale datavariablene kan kun aksesseres av prosedyrene til monitoren
2. En prosess entrer monitoren ved å benytte en av dens prosedyrer/metoder
3. Kun én prosess kan utføres i monitoren av gangen, så alle andre prosesser som forsøker å entre monitoren blir blokkert og må vente på at monitoren blir tilgjengelig. Dette gir **automatisk gjensidig utelukkelse**

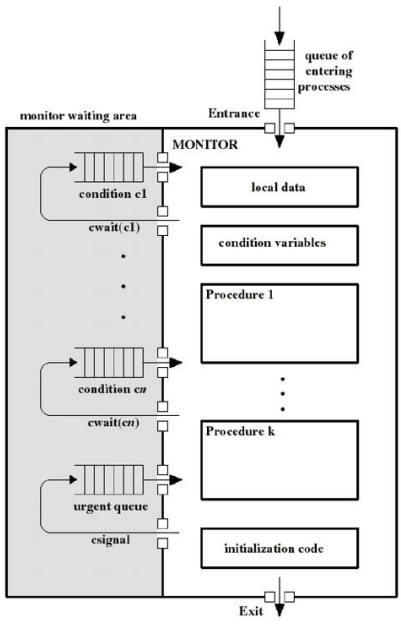
De to første punktene minner om objektorientert struktur, og en monitor kan enkelt implementeres som et objekt med spesielle egenskaper. **En delt datastruktur kan beskyttes ved å plasseres i en monitor**, siden monitoren vil sikre gjensidig utelukkende aksess.

For å kunne utføre parallelle prosesser, må monitoren ha synkroniseringsverktøy. For eksempel hvis prosessen som entrer monitoren blir blokkert av en betingelse, må det finnes et verktøy som gjør at prosessen frigjør monitoren, slik at andre prosesser kan entre den. Når betingelsen oppfylles og prosessen blir avblokkert, må den kunne entre monitoren igjen når den blir ledig. **En monitor støtter synkronisering vha. betingelsesvariabler** som holdes i monitoren og kan bare aksesseres innenfor monitoren. Det er to typer betingelsesvariabler:

- **cwait(c)** – suspender utføring av kallende prosess på tilstand c, slik at monitoren blir ledig for en annen prosess
- **csignal(c)** – gjenoppta utføring av en prosess som er blokkert etter en cwait på samme betingelse. Velg én hvis det er flere og gjør ingenting hvis det er ingen.

Merk at disse operasjonene er forskjellig fra semaforer. Hvis en prosess i en monitor sender et signal og ingen oppgave venter på betingelsesvariabelen, vil signalet bli tapt.

Figuren viser strukturen til en monitor. Vi kan tenke at monitoren har et enkelt inngangspunkt som er voktet slik at kun en prosess kan være i monitoren av gangen. Andre prosesser som forsøker å entre vil blokkeres og plasseres i en kø. Legg merke til at venteområdet består av flere køer, slik at prosesser kan vente på flere betingelser. Området til høyre er monitoren og det er i denne delen det kun er en prosess som kan utføres om gangen. Når en prosess er i monitoren kan den midlertidig blokkere seg selv ved å kalle på cwait(x) på betingelsen x. Den vil da plasseres i en kø med ventende prosesser, helt til en prosess som utføres i monitoren detekterer en endring i betingelsen x og kaller csignal(x) for å gi beskjed til køen som venter på denne betingelsen.



Merk: urgent queue inneholder prosesser som er blokkert på csignal funksjonen, s. 261

```
/* program producerconsumer */
monitor boundedbuffer;
char buffer [N];                                    /* space for N items */
int nextin, nextout;                                /* buffer pointers */
int count;                                          /* number of items in buffer */
cond notfull, notempty;          /* condition variables for synchronization */

void append (char x)
{
    if (count == N) cwait(notfull);      /* buffer is full; avoid overflow */
    buffer[nextin] = x;
    nextin = (nextin + 1) % N;
    count++;
    /* one more item in buffer */
    csignal(notempty);                       /* resume any waiting consumer */
}
void take (char x)
{
    if (count == 0) cwait(notempty);   /* buffer is empty; avoid underflow */
    x = buffer[nextout];
    nextout = (nextout + 1) % N;
    count--;                                    /* one fewer item in buffer */
    csignal(notfull);                          /* resume any waiting producer */
}
{                                                       /* monitor body */
    nextin = 0; nextout = 0; count = 0;          /* buffer initially empty */
}
```

```
void producer()
{
    char x;
    while (true) {
    produce(x);
    append(x);
    }
}
void consumer()
{
    char x;
    while (true) {
      take(x);
      consume(x);
    }
}
void main()
{
    parbegin (producer, consumer);
}
```

Figuren viser hvordan en monitor kan brukes for å løse producer/consumer problemet. Monitormodulen boundedbuffer brukes for å kontrollere bufferen som brukes til å lagre data. Monitoren består av to betingelsesvariabler notfull (*true* når det er plass til minst en ekstra data i bufferen) og notempty (*true* når det er minst en data i bufferen). For å legge til data i bufferen må produceren bruke append-funksjonen til monitoren og denne funksjonen krever at producer først sjekker betingelsen notfull. Hvis det ikke er plass i bufferen vil producer blokkeres på denne betingelsen, og en annen prosess (producer eller consumer) kan entre monitoren. Når en consumer har fjernet data fra bufferen vil den kalle csignal(notfull) slik at blokkeringen på producer fjernes, og den kan gjenoppta utførelsen. Etter å ha lagt til data til bufferen vil producer kalle csignal(notempty). En lignende beskrivelse vil gjelde for consumer funksjonen. **Dette viser forskjellen i ansvar mellom semaforer og monitorer. For monitorer er det monitorkonstruksjonen som sørger for gjensidig utelukkelse (ikke mulig for producer og consumer å aksessere bufferen samtidig) og utvikler må plassere cwait og cnotify innenfor monitoren for å hindre at prosesser legger til elementer i full buffer eller fjerner fra tom buffer (dvs. utvikler har ansvar for synkroniseringen). For semaforer er utvikler ansvarlig for både den gjensidige utelukkelsen og synkroniseringen!**

Det er mulig å gjøre feil ved monitorer dersom synkroniseringen ikke implementeres riktig (dvs. cwait og csignal blir brukt feil). Fordelen sammenlignet med semaforer er likevel at synkroniseringen er begrenset til monitoren, slik at det er lettere å se om det har blitt implementert riktig og detektere bugs (i semaforer må alle prosesser implementeres riktig). I tillegg vil monitorer gi automatisk gjensidig utelukkelse, slik vi så over.

Monitor med Notify og Broadcast (Mesa)

Monitor med signal krever at dersom det er minst en prosess i betingelseskøen, må en prosess fra køen kjøre umiddelbart når en annen prosess kaller csignal på betingelsen. Prosessen som kaller csignal må derfor ut av monitoren umiddelbart eller blokkeres. Dette introduserer to ulemper:

1. **Hvis prosessen som kaller csignal ikke er ferdig, vil det resultere i to ekstra prosesskifter**: en for å blokkere prosessen og en for å gjenoppta utføringen når monitoren er tilgjengelig
2. **Tidsstyringen må være 100% pålitelig for å unngå feil**. Når csignal er kalt må en prosess fra betingelsesløen umiddelbart aktiveres og tidsstyringen må sørge for at ingen andre prosesser entrer monitoren i mellomtid. Hvis ikke kan betingelsen endres.

Løsningen på disse problemene er en Mesa-monitor-struktur, der csignal byttes ut med cnotify. Når en prosess som utføres i monitoren utfører cnotify(x) vil det vekke betingelseskøen til x, men den signaliserende prosessen vil fortsette utføringen. Resultatet er at prosessen først i betingelseskøen vil fortsette når monitoren blir ledig. Siden det ikke er noen garanti på at ingen andre prosesser har entret monitoren før dette, må den ventende prosessen sjekke betingelsen på nytt før den entrer monitoren. Figuren viser koden for boundedbuffer i dette tilfellet. If-setningen er erstattet med en while-løkke,

```
void append (char x)
{
    while(count == N) cwait(notfull);   /* buffer is full; avoid overflow */
    buffer[nextin] = x;
    nextin = (nextin + 1) % N;
    count++;                                    /* one more item in buffer */
    cnotify(notempty);                       /* notify any waiting consumer */
}

void take (char x)
{
    while(count == 0) cwait(notempty); /* buffer is empty; avoid underflow */
    x = buffer[nextout];
    nextout = (nextout + 1) % N;
    count--;                                    /* one fewer item in buffer */
    cnotify(notfull);                        /* notify any waiting producer */
}
```

Merk: forskjellen fra vanlig monitor er at man bytter til while-løkke (retesting av betingelse) og signaliseringsformen byttes til cnotify (unngår umiddelbar gjenstart av tidligere stoppet prosess).

som gir minst en ekstra evaluering av betingelsesvariabelen. Det vil samtidig innebære at det ikke blir noen ekstra prosessbytte og det er ingen krav om at ventende prosess må kjøre umiddelbart etter cnotify.

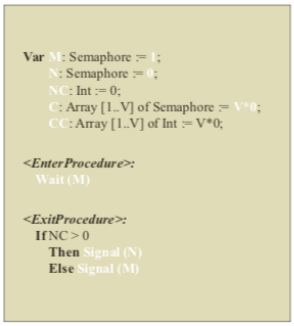
For å hindre utsulting, som følger av at en prosess feiler før den rekker å signalisere cnotify, kan man introdusere et timeout-intervall på hver betingelse. Hver prosess som har ventet lengre enn maksimum tidsintervall blir plassert i Ready-tilstand uansett om det har blitt signalisert eller ikke. Når den aktiveres vil prosessen sjekke betingelsen og utføres hvis betingelsen er tilfredsstilt. **Man kan også legge til funksjonen cbroadcast, som vil gjøre at alle prosesser som venter på en betingelse blir plassert i Ready-tilstand**. Dette er nyttig når prosessen ikke vet hvor mange prosesser som trengs å reaktiveres eller hvilken prosess den skal reaktivere. For eksempel kan en memory manager som har et bestemt antall bytes ledig vekke opp mange prosesser, slik at de selv kan sjekke om det er nok minne ledig (s. 263).

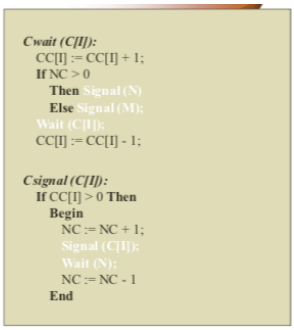
Monitor med signal vs. Mesa-monitor

Fordelen med Mesa-monitor er at den er mindre utsatt for feil, fordi selv om en prosess signaliserer feil, vil den ventede prosessen sjekke betingelsen ved while-løkken før den utføres (vil vente hvis betingelse ikke er oppfylt). En annen fordel er at Mesa-monitoren legger mer til rette for en modulær programkonstruksjon. Se s. 263 for et eksempel. **Monitoren med signal krever ingen retesting av betingelsen (billigere for applikasjonen), men de ekstra prosess- og/eller trådskiftene er kostbare for systemet**. Mesa-monitoren tilbyr i tillegg cbroadcast-funksjonen som kan være nyttig for flere formål.

OBS

Implementering av monitorer vha. semaforer (E)





Figuren viser hvordan semaforer kan brukes for å implementere monitor-betingelsesvariablene cwait og csignal. Løsningen bruker semaforene M og N. I cwait vil if-setningen sjekke om betingelsen er oppfylt ($NC > 0$) og hvis det er tilfellet vil den inkrementere semaforene (dvs. kalle semSignal), slik at ventede prosesser kan avblokkeres. Hvis ikke vil den vente. I csignal vil if-setningen sjekke om det er ventede prosesser og hvis det er tilfellet vil den signalisere til ventede prosesser og deretter blokkeres. Det er relativt enkelt å implementere cnotify hos Mesa-monitor, men vanskelig å implementere cbroadcast.

Meldinger:
- Gjensidig utelukkelse – bruker tomme meldinger
- Generell tidsordning – bruker fylte meldinger som stafettpinne + beskjed. Tjener og klient port blir angitt eksplisitt av motpart.

5.6 Meldinger

Meldinger er en måte for to prosesser å utbytte informasjon, og det kan benyttes for synkronisering (oppnå gjensidig utelukkelse). For å oppnå synkronisering og kommunikasjon mellom prosessene, kan man bruke meldingssystem. Dette kan også benyttes i distribuerte system, i tillegg til uniprosessor- og multiprosessorsystem med delt minne. Et meldingssystem kommer i mange former, men det minimale settet med nødvendige operasjoner er:

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

Tabellen viser designkarakteristikker for meldingssystem.

Synchronization	**Format**
Send	Content
blocking	Length
nonblocking	fixed
Receive	variable
blocking	
nonblocking	**Queueing Discipline**
test for arrival	FIFO
	Priority
Addressing	
Direct	
send	
receive	
explicit	
implicit	
Indirect	
static	
dynamic	
ownership	

Synkronisering

Sender og mottaker kan være blokkerende eller ikke-blokkerende:

- **Blokkerende sender og mottaker** – sender og mottaker er blokkert inntil meldingen er levert. Dette kalles *Rendezvous* og gir tett synkronisering mellom prosesser

- **Ikke-blokkerende sender, blokkerende mottaker** – senderen kan fortsette utføringen, men mottakeren er blokkert inntil den forespurte meldingen ankommer. Dette er sannsynligvis den mest nyttige kombinasjonen, siden det lar prosesser sende en eller flere meldinger til ulike destinasjoner så raskt som mulig. Et eksempel er servere som eksisterer for å gi tjenester eller ressurser til andre prosesser
- **Ikke-blokkerende sender og mottaker** – ingen av partene trenger å vente

Ikke-blokkerende sender er mest naturlig for oppgaver i samtidsprogrammering. En ulempe er at feil i den ikke-blokkerte sendefunksjonen kan resultere i at prosessen gjentatt genererer meldinger, noe som vil føre til **overbruk av systemressurser** (eks: prosessortid og bufferplass). I tillegg blir det utviklerens oppgave å bestemme om en melding har blitt mottatt vha acknowledgement meldinger. For mottaker er det mest vanlig med blokkerende variant. En ulempe ved denne er at mottakeren kan bli blokkert dersom meldingen går tapt eller sender feiler før den får sendt meldingen. Dette unngås med ikke-blokkerende mottaker, men den er igjen utsatt for at meldinger kan gå tapt dersom de sendes etter at en prosess allerede har utført en matchende receive.

Adressering

Sender må kunne oppgi hvilken prosess som skal motta meldingen og mottaker må kunne oppgi kilden til meldingen som mottas. Prosesser kan adresseres ved:

- **Direkte adressering** – senderen inkluderer en spesifikk identifikator til destinasjonsprosessen. Hvis mottaker kjenner sender på forhånd kan den eksplisitt gi kildeprosessen, ellers blir det brukt en implisitt adressering vha kilde-parameteren til receive operasjonen (s. 266)
- **Indirekte adressering** – meldinger sendes ikke direkte fra sender til mottaker, men via en delt datastruktur som består av køer som kan midlertidig holde meldinger (kalles mailbokser). Prosesser kommuniserer ved at en prosess sender en melding til passende mailboks og den andre henter meldingen fra mailboksen. Fordelen er at sender og mottaker kobles fra hverandre, noe som gir større fleksibilitet i meldingssystemet. Forholdet mellom sender og mottaker kan være:
    - **En-til-en** – privat kommunikasjon mellom to prosesser
    - **En-til-mange** – en sender har mange mottakere, for eksempel broadcasting
    - **Mange-til-en** – nyttig i klient/server-interaksjon, der en prosess gir tjeneste et antall prosesser
    - **Mange-til-mange** – lar flere serverprosesser gi samtidige tjenester til flere klienter

Prosesser kan ha statisk eller dynamisk assosiasjon til mailbokser. Ved statisk assosiasjon vil prosessen være permanent koblet til en port (vanlig ved en-til-en), mens ved dynamisk assosiasjon vil koblingene kunne endres (vanlig ved mange sendere, brukes ofte med connect og disconnect).

Meldingsformat

Meldingsformatet avhenger av formålet til meldingssystemet og om systemet kjører på en enkelt datamaskin eller et distribuert system. Korte meldinger med fast lengde gir lite prosessering og lagring overhead, mens meldinger med variabel lengde gir større fleksibilitet.

Kødisipling

Enkleste kødisiplin er FIFO, men det kan være utilstrekkelig dersom noen meldinger haster mer enn andre. Dette kan håndteres med prioritetsnivå for meldinger basert på meldingstype eller sender. Et annet alternativ er at mottaker kan inspisere meldingskøen og velge neste melding den skal motta.

Meldinger – gjensidig utelukkelse

```
/* program mutualexclusion */
const int n = /* number of processes  */;
void P(int i)
{
    message msg;
    while (true) {
     receive (box, msg);
     /* critical section   */;
     send (box, msg);
     /* remainder   */;
    }
}
void main()
{
    create_mailbox (box);
    send (box, null);
    parbegin (P(1), P(2), . . ., P(n));
}
```

Figuren viser en måte å oppnå gjensidig utelukkelse med meldinger, når receive er blokkerende og send er ikke-blokkerende. Et sett med parallelle prosesser deler en mailboks (box) som brukes for å sende og motta meldinger. Mailboksen initieres med en tom melding. En prosess som ønsker å entre kritisk seksjon vil forsøke å motta en melding, og dersom mailboksen er tom vil den blokkeres. Hvis mailboksen ikke er tom og prosessen mottar meldingen, vil den utføre kritiske seksjon og deretter plassere meldingen tilbake til mailboksen. **Den tomme meldingen vil altså fungere som en token mellom prosesser**. Denne løsningen antar at hvis mer enn en prosess gjennomfører receive samtidig, så vil følgende gjelde:

- Hvis det er en melding vil denne leveres til kun én prosess og andre prosesser blokkeres
- Hvis meldingskøen er tom, vil alle prosesser blokkeres. Når en melding blir tilgjengelig aktiveres kun en av prosessene for å motta meldingen

Disse antagelsene er oppfylt for nesten alle meldingssystem.

Meldinger – løsning på producer/consumer problem

```
const int
    capacity = /* buffering capacity */ ;
    null =/* empty message */ ;
int i;
void producer()
{   message pmsg;
    while (true) {
     receive (mayproduce, pmsg);
     pmsg = produce();
     send (mayconsume, pmsg);
    }
}
void consumer()
{   message cmsg;
    while (true) {
     receive (mayconsume, cmsg);
     consume (cmsg);
     send (mayproduce, null);
    }
}

void main()
{
    create_mailbox (mayproduce);
    create_mailbox (mayconsume);
    for (int i = 1; i <= capacity; i++) send (mayproduce,
null);
    parbegin (producer, consumer);
}
```

Figuren viser en løsning på producer/consumer problemet som er basert på et meldingssystem. Løsningen utnytter meldingssystemets evne til å sende data i tillegg til signaler, og den bruker to mailbokser. Producer sender data som meldinger til mailboksen mayconsume og så lenge det er minst en melding i mailboksen vil consumer kunne forbruke disse. Dermed vil mayconsume fungere som bufferen, der dataen er ordnet i en kø av meldinger og størrelsen er gitt av den globale variabelen capacity. Den andre mailboksen kalles mayproduce, og den vil initialt fylles opp med et antall tomme meldinger som er likt kapasiteten til bufferen. Antall tomme meldinger reduseres for hver produksjon og øker med hvert forbruk (dvs. holder kontroll over antall meldinger i bufferen). **Denne løsningen er fleksibel og den tillater flere producers og consumers, så lenge alle har tilgang til begge mailboksene. Systemet kan også være distribuert (producers og mayconsume på en side og consumers og mayproduce på den andre siden).**

5.7 Readers/Writers problemet

Det er et område med data som deles av mange prosesser (eks: fil, minneblokk, prosessorregister, osv.). Det er et antall prosesser som kun vil lese dataen (readers) og et antall prosesser som kun vil skrive dataen (writers). Følgende betingelser gjelder:

1. Et vilkårlig antall readers kan lese filen samtidig
2. Kun én writer kan skrive filen av gangen
3. Hvis en writer skriver til filen, kan ingen reader lese den

Readers er altså prosesser som ikke trenger å ekskludere noen prosesser, mens writers er prosesser som må ekskludere alle andre prosesser (readers og writers). Dersom alle prosesser kunne ha lest eller skrevet til dataområdet kunne aksessen blitt behandlet som en kritisk seksjon og løsningen ville ha vært en generell gjensidig utelukkelse. **Begrensningene er nødvendig fordi det finnes mer effektive løsninger for dette tilfellet**. Man kan heller ikke si at producer/consumer problemet er et enkelt tilfelle med readers/writers, fordi producer er ikke kun en skriver (eks: sjekker om buffer er full). Consumer vil heller ikke være kun en leser (eks: justerer kø-pekere for å vise at den har fjernet enhet fra bufferen).

Problemtyper:

- Barbersalong – løses med semafor
- Producer/consumer – løses med monitor
- Writer/reader – løses med meldinger

```
/* program readersandwriters */
int readcount;
semaphore x = 1, wsem = 1;
void reader()
{
    while (true) {
      semWait (x);
      readcount++;
      if (readcount == 1) semWait (wsem);
      semSignal (x);
      READUNIT();
      semWait (x);
      readcount--;
      if (readcount == 0) semSignal (wsem);
      semSignal (x);
    }
 }
void writer()
{
    while (true) {
      semWait (wsem);
      WRITEUNIT();
      semSignal (wsem);
    }
}

void main()
{
    readcount = 0;
    parbegin (reader, writer);
}
```

Readers har prioritet – semaforer

Figuren viser løsningen basert på semaforer for en reader og en writer (løsning er tilsvarende for flere). Semaforen wsem brukes for å oppnå gjensidig utelukkelse, og den initialiseres til 1. Når en writer aksesserer dataen vil den kalle semWait(wsem), slik at wsem reduseres til 0 og alle andre writers og readers som forsøker å aksessere dataen blir blokkert. Reader bruker også wsem for å oppnå gjensidig utelukkelse med writer (dvs. hindre at writer kommer etter påbegynt lesing). Dersom det er flere readers, er det kun nødvendig at første reader kaller semWait(wsem), for å sikre gjensidig utelukkelse. Global variabel readcount brukes for å holde styr over antall readers, mens semafor x brukes for å sikre at readcount oppdateres riktig.

Writers har prioritet – semaforer

Problemet med forrige løsning er at dersom en reader har begynt å aksessere et dataområde, kan readers bevare kontrollen over området så lenge en reader er aktiv. Dette gjør at writers utsettes for utsulting. Løsningen er å sikre at ingen nye readers får aksess når en writer har utrykt ønske om å skrive. Figuren viser løsningen, der følgende er lagt til:

- Semafor rsem som hindrer alle readers så lenge det er en writer som vil aksessere dataen
- Variabel writecount som kontrollerer innstillingen av
- Semafor y som kontrollerer oppdateringen av writecount
- En semafor z som brukes for å hindre at rsem blir så full at writers ikke får plass (dvs. reduseres for mye). Det er kun én reader som får vente i rsem og resten venter i z

Tabellen under viser alle mulighetene.

Readers only in the system	•*wsem* set •no queues
Writers only in the system	•*wsem* and *rsem* set •writers queue on *wsem*
Both readers and writers with read first	•*wsem* set by reader •*rsem* set by writer •all writers queue on *wsem* •one reader queues on *rsem* •other readers queue on *z*
Both readers and writers with write first	•*wsem* set by writer •*rsem* set by writer •writers queue on *wsem* •one reader queues on *rsem* •other readers queue on *z*

```
/*program readersandwriters*/
int  readcount, writecount;
semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader()
{
    while (true) {
      semWait (z);
          semWait (rsem);
              semWait (x);
                  readcount++;
                  if (readcount == 1) semWait (wsem);
              semSignal (x);
          semSignal (rsem);
      semSignal (z);
      READUNIT();
      semWait (x);
          readcount--;
          if (readcount == 0) semSignal (wsem);
      semSignal (x);
    }
}
void writer ()
{
    while (true) {
      semWait (y);
          writecount++;
          if (writecount == 1) semWait (rsem);
      semSignal (y);
      semWait (wsem);
      WRITEUNIT();
      semSignal (wsem);
      semWait (y);
          writecount--;
          if (writecount == 0) semSignal (rsem);
      semSignal (y);
    }
}
void main()
{
    readcount = writecount = 0;
    parbegin (reader, writer);
}
```

Writers har prioritet – meldingssystem

Figuren viser løsningen basert på meldinger, der det brukes en kontroller-prosess som har aksess til det delte dataområdet. Andre prosesser som ønsker aksess til dataområdet sender en request til kontrolleren, får aksess ved en «ok»-melding og indikerer fullført aksess ved å sende en «finished»-melding tilbake. Kontrolleren har tre mailbokser, en for hver type melding den kan motta. Writers har prioritet ved at kontrolleren fullfører write-requester før read-requester. For å oppnå gjensidig utelukkelse vil kontrolleren bruke variabelen count som initialiseres til et antall som er større enn maksimalt antall readers (antar 100). Vi får:

- count > 0: det er ingen ventende writer og kan være aktive readers. Alle «finished»-meldinger blir håndtert først for å klarere aktive readers. Deretter håndteres write-request og til slutt read-requests

- count = 0: eneste gjenstående request er write-request, så la writer fortsette og vent på «finished»-meldingen
- count < 0: en writer har kommet med en request og er satt på vent for å klarere alle aktive readers. Det er kun «finished»-meldinger som behandles

```
void reader(int i)
{
   message rmsg;
      while (true) {
         rmsg = i;
         send (readrequest, rmsg);
         receive (mbox[i], rmsg);
         READUNIT ();
         rmsg = i;
         send (finished, rmsg);
      }
 }
void writer(int j)
{
   message rmsg;
   while(true) {
      rmsg = j;
      send (writerequest, rmsg);
      receive (mbox[j], rmsg);
      WRITEUNIT ();
      rmsg = j;
      send (finished, rmsg);
   }
}
```

```
void  controller()
{
      while (true)
      {
         if (count > 0) {
            if (!empty (finished)) {
               receive (finished, msg);
               count++;
            }
            else if (!empty (writerequest)) {
               receive (writerequest, msg);
               writer id = msg.id;
               count = count - 100;
            }
            else if (!empty (readrequest)) {
               receive (readrequest, msg);
               count--;
               send (msg.id, "OK");
            }
         }
         if (count == 0) {
            send (writer id, "OK");
            receive (finished, msg);
            count = 100;
         }
         while (count < 0) {
            receive (finished, msg);
            count++;
         }
      }
}
```

Figuren til høyre viser hvordan **meldinger kan implementeres med semaforer**.

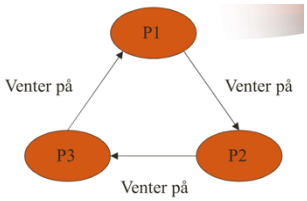
```
Var Slot : Semaphore := N;
    Buffer : Semaphore := 1;
    Port : Array [1..M] of Semaphore := M*0;
    BufferQ : List of Item;
    PortQ : Array [1..M] of List of Item;

For I := 1 To N
  <Link Item into BufferQ>
```

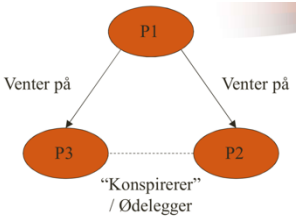
```
Send (Destination, Message):
   Wait (Slot);
   Wait (Buffer);
   <Unlink Item from BufferQ>;
   <Copy Message into Item >;
   <Link Item into PortQ[Destination]>;
   Signal (Port[Destination]);
   Signal (Buffer)

Receive (Source, Message):
   Wait (Port[This]);
   Wait (Buffer);
   <Unlink Item from PortQ[This]>;
   <Copy Message from Item >;
   <Link Item into BufferQ >;
   Signal (Slot);
   Signal (Buffer)
```

Semaforer vs. Monitorer vs. Meldinger

Semaforer, monitorer og meldinger kan alle løse synkroniseringsproblemer, men de passer bedre eller dårligere for ulike formål eller problemtyper. Funksjonelt vil semaforer og monitorer være likeverdige, og de kan implementeres sammen. Semaforer er mer generelle, som vil si at de kan brukes til et bredt spekter av problemer. Samtidig er det lettere å begå feil og de har ikke automatisk gjensidig utelukkelse slik som monitorer. Siden semaforer er et lavnivå verktøy kan de gi plassbesparende løsninger. Monitorer følger en objektorientert struktur og er enklere å vise korrekthet for. Ettersom monitorer har automatisk gjensidig utelukkelse, er de spesielt egnet for problemer der gjensidig utelukkelse er et essensielt behov. Fordelen med meldinger er at de kan brukes uten delt lager, som vil inkludere distribuerte system. De er også svært egnet dersom generell tidsstyring er et essensielt behov.

Kapittel 6 – Samtidighet: Vranglås og utsulting



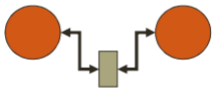


Tre typer problemer som ofte oppstår ved samtidig prosessering er:

1. **Vranglås (*deadlock*)** = en situasjon der to eller flere prosesser ikke klarer å fortsette utføringen fordi alle venter på at en annen prosess skal gjøre noe (se figur).
2. **Livelock** = en situasjon der to eller flere prosesser kontinuerlig endrer tilstanden sin i respons til endring i tilstanden til andre prosesser, uten å gjøre noe nyttig arbeid.
3. **Utsulting (*starvation*)** = kjørbar prosess blir uendelig oversett av scheduler; selv om den er klar til å fortsette, blir den aldri valgt (se figur).

6.1 Prinsipper ved vranglås

En vranglås defineres som en permanent blokkering av et sett med prosesser som enten konkurrerer om systemressurser eller kommuniserer med hverandre. Et sett med prosesser er i vranglås dersom hver prosess i settet er blokkert og venter på en hendelse som kan trigges av en annen prosess i settet. Det finnes ingen generell løsning på vranglås. Avhengig av logikken til prosessene, kan de mulige progresjonsbanene (dvs. rekkefølge til utførte aktiviteter) inneholde fatale regioner. **Hvis banen til prosessene entrer den fatale regionen vil det være umulig å unngå vranglås** (s. 293). For at vranglås skal være uunngåelig må altså prosessene utføres slik at de entrer fatal region (dvs. dersom det ikke eksisterer progresjonsbane som entrer denne regionen, vil ikke vranglås kunne oppstå). Dette kan illustreres med joint progress diagram som viser progresjonen til to prosesser som konkurrerer om ressurser (s. 292-294).



For å karakterisere tildeling av ressurser bruker man **ressurstildelingsgrafer**, som er en rettet graf der prosesser gis som sirkler og ressurser gis som firkanter (se figur). Pil fra prosess til ressurs betyr at prosessen ber om tilgang, mens pil fra ressurs til prosess betyr at prosessen har tilgang (s. 296).

Vranglås og synkronisering (E)

Vranglås er et resultat av prosessynkronisering, siden synkroniseringen innebærer at prosesser må vente på hverandre, og når dette foregår i en ring, vil det være vranglås. Synkroniseringsverktøy som semaforer, monitorer eller meldinger kan alle lede til vranglås som må håndteres, så **vranglåshåndtering er et følgeproblem av prosessynkronisering** (tilsvarende for utsulting). Ulike synkroniseringsverktøy kan gjøre det lettere eller vanskeligere å håndtere vranglåser i ulike situasjoner, men i utgangspunktet må man legge til egne mekanismer for vranglåshåndtering, slik som umuliggjøring, unngåing eller oppdaging og oppretting.

Vranglås karakteristikker

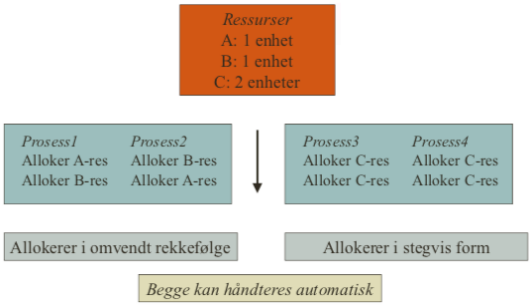
Fire betingelser må være tilstede for at vranglås skal kunne oppstå:

1. **Gjensidig utelukkelse** – en ressurs kan kun tildeles en prosess av gangen. Ingen prosess kan aksessere en ressurs som har blitt tildelt en annen prosess.
2. **Ingen oppgivelse (*hold&wait*)** – en prosess som holder en ressurs kan be om flere ressurser. Dvs. prosessen kan holde ressurs selv om den venter.
3. **Ingen fjerning** – prosesser kan ikke miste tilgang til en ressurs mot sin vilje. Ingen ressurs kan fjernes med tvang fra prosessen som holder dem
4. **Sirkulær venting** – det må være en sirkulær kjede av to eller flere prosesser som venter på en ressurs som er holdt av den neste prosessen i kjeden. Dette kan være en konsekvens av de tre første (dvs. hvis de tre første eksisterer, kan punkt 4 oppstå).

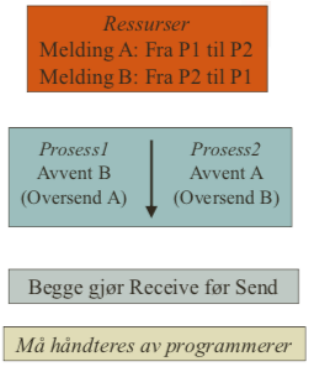
Merk: et fatalt område vil eksistere dersom de tre første betingelsene er oppfylt og fjerde betingelse vil innebære at det har blitt utført en sekvens med ressurs-requests som har gjort at progresjonsbanen har entret den fatale regionen. Dermed er vranglås uunngåelig. Betingelsene 1-3 gjør altså at vranglås blir mulig (fatalt område eksisterer), men det er betingelse 4 som gjør at vranglås oppstår (entrer fatalt

Vranglås ved ulike typer ressurser

Fornybare ressurser



En fornybar ressurs er en ressurs som kan benyttes sikkert av én prosess av gangen og som ikke blir oppbrukt. Eksempler er IO-kanaler, hovedminne, sekundærminne, filer, databaser og semaforer. Figuren viser et eksempel på to tilfeller der vranglås oppstår: (1) hver prosess holder en ressurs og sender request om den andre og (2) det er to like ressurser og hver prosess holder en og ønsker tilgang til den andre. **En måte å håndtere slik vranglås er å innføre systemdesign-begrensninger på rekkefølgen ressursene tildeles/etterspørres (vil gjøre at problemene håndteres automatisk)**. For eksempel for tilfelle 1 kan man legge til en begrensning på at ressurs A må tildeles før en prosess kan be om ressurs B (gjør at prosess 1 får B, og prosess 2 må vente). For tilfelle 2 kan man kreve at begge ressursene blir tildelt av gangen. **Dette er likevel en ineffektiv løsning, fordi det vil ofte gjør at ressursene holdes lenger enn nødvendig.**

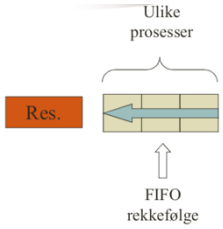


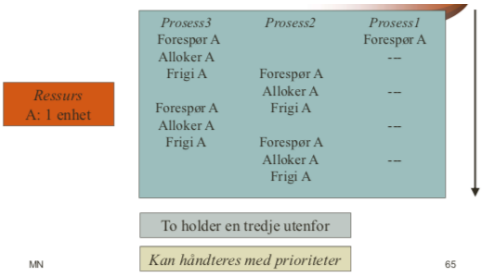
Konsumerbare ressurser

En konsumerbar ressurs er en ressurs som kan opprettes og forkastes etter bruk, og det er som regel ingen begrensning på mengde. En blokkert prosess kan ofte opprette flere slike ressurser. Eksempler er avbrudd, signaler og informasjon i IO-buffere. Figuren viser et eksempel på et tilfelle der vranglås på konsumerbar ressurs oppstår. Begge prosessene forsøker å motta og sende en melding til den andre prosessen. Vranglås vil oppstå dersom receive er blokkerende (dvs. blokkeres helt til meldingen mottas) og receive skjer før send, siden begge prosessene venter på melding fra den andre og kan derfor ikke sende melding.

Utsulting – karakteristikker og vanlig håndtering

Utsulting innebærer at en kjørbar prosess blir uendelig oversett av scheduler. Selv om prosessen er klar til å fortsette, blir den aldri valgt. Figuren til høyre viser et eksempel der prosess3 og prosess2 holder prosess1 utenfor. **Karakteristikken ved utsulting er ofte uensartet prosessprioritering.** En vanlig måte å unngå utsulting er å **tildele ressurser ut fra ventetid** (FIFO rekkefølge), slik at prosessen som har ventet lengst får høyest prioritering (se figur til venstre).





6.2-6.4 Vranglåsmekanismer

Vranglåsmekanismer trengs for å håndtere situasjoner der synkronisering gjør at to eller flere prosesser/tråder venter på hverandre i ring. Vranglåshåndtering er altså et følgeproblem av prosessynkronisering. Vi ser på tre ulike vranglåsmekanismer:

- **Umuliggjøre vranglås** – sikre i forkant at det ikke kan oppstå vranglåser, ved å bruke en policy som eliminerer en av betingelsene (1-4).
- **Unngå vranglås –** sikre der og da at det ikke vil oppstå vranglåser, ved å gjøre passende dynamiske valg basert på nåværende tilstand hos ressurstildelingen
- **Oppdage og rette opp vranglås** – la det oppstå vranglåser og deretter rett opp disse, altså detekter og fiks vranglåser.

De løser samme type problem, men hver mekanisme har bestemte situasjoner der de er best egnet. Umuliggjøring og unngåelse bør brukes når vranglåser kan oppstå ofte, for da vil gjenoppretting etter oppdaging være ressurskrevende i etterkant. Oppdaging bør brukes når vranglåser sjeldent vil oppstå for da vil umuliggjøring og unngåing være ressurskrevende i forkant. Umuliggjøring er mer begrenset enn unngåelse angående parallellitet, så umuliggjøring brukes når man kan akseptere liten resulterende parallellitet (og motsatt).

6.2 Umuliggjøre vranglås

Strategien ved umuliggjøre vranglås er å designe systemet slik at muligheten for at vranglås skal oppstå ikke er tilstede, ved at man forhindrer minst en av de fire betingelsene. Metodene kan klassifiseres som indirekte metoder (hindrer betingelse 1-3) eller direkte metoder (hindrer betingelse 4).

Gjensidig utelukkelse

Generelt sett er dette et krav som ikke kan forhindres. Hvis tilgang til en ressurs krever gjensidig utelukkelse, må det støttes av OS. For noen ressurser, slik som filer, kan man delvis forhindre gjensidig utelukkelse ved å tillate samtidig aksess ved lesing, men ikke for skriving.

Ingen oppgivelse (hold&wait)

Hold&wait betingelsen kan forhindres ved å kreve at en ressurs etterspør alle ressursene den trenger samtidig, og deretter blokkere prosessen helt til alle etterspurte ressurser er tilgjengelig og kan tildeles prosessen (alle ressurser allokeres samtidig). Ulemper er:

1. En prosess kan blokkeres lenge, når den i mange tilfeller kunne ha fortsatt med kun noen av ressursene.
2. Flere av ressursene som er tildelt en prosess kan stå ubrukt over lengre perioder, når de i stedet kunne ha blitt utnyttet av andre prosesser.
3. Det krever at prosessen vet på forhånd alle ressursene den trenger.

Det er også problematisk for applikasjoner med modulær programmering eller multithreaded struktur, siden det vil kreve at applikasjonen er klar over alle ressursene som har blitt etterspurt ved alle nivåene eller i alle modulene.

Ingen fjerning (*no preemption*)

Denne betingelsen kan forhindres på flere måter:

1. En prosess som holder på en ressurs og får avslag på en annen ressurs, må **frigjøre ressursen den holder**, og be om tilgang til begge ressursene igjen (dvs. omstart)
2. Dersom en prosess ber om en ressurs som holdes av en annen prosess, kan **OS trekke tilbake rettighetene slik at den andre prosessen må frigjøre ressursene sine**. Dette vil kun hindre vranglås dersom ingen prosesser har samme prioritet.

Denne metoden er kun praktisk når tilstanden til ressursene kan lett lagres og gjenopprettes senere (slik som prosessor).

Sirkulær venting

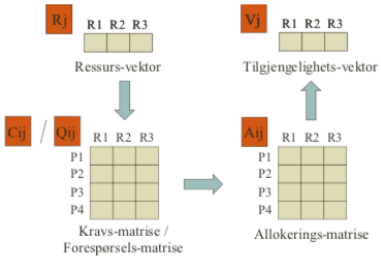
Sirkulær venting kan forhindres ved å definere en lineær ordning av ressurstypene. Hvis en prosess har blitt tildelt en ressurs av type R, kan den i ettertid kun etterspørre ressurser av typen som kommer etter R i ordningen. Dette kan være en ineffektiv løsning, siden det kan senke farten til prosesser og unødvendig hindre tilgang til ressurser.

6.3 Unngå vranglås

Unngåelse av vranglås tillater de tre betingelsene som gjør at vranglås kan bli mulig, men gjør kloke valg for å sørge for at vranglås ikke oppstår (dvs. hindrer betingelse 4). Unngåelse tillater derfor mer parallellitet enn umuliggjøring. Avgjørelsen om at en etterspurt ressursallokering kan potensielt lede til vranglås blir tatt dynamisk. Dette krever derfor kunnskap om prosessers fremtidige behov for ressurser. Vi ser på to tilnærminger:

1. **Avslag på prosessinitiering** – en prosess blir ikke startet dersom den har behov som kan føre til vranglås
2. **Avslag på ressursallokering** – ikke innvilg en trinnvis ressurs-request dersom allokeringen kan føre til vranglås

Avslag på prosessinitiering



En ny prosess blir akseptert dersom problem aldri kan oppstå. Dersom systemet består av n prosesser og m ulike ressurstyper, lar vi $R = (R_1, \dots, R_m)$ være total mengde av hver ressurs, $V = (V_1, \dots, V_m)$ er andel av hver ressurs som ikke er tildelt en prosess, C er en matrise der C_{ij} er behovet prosess i har for ressurs j og A er en matrise der A_{ij} er nåværende allokering hos prosess i for ressurs j (s. 301). Matrisen C har en rad for hver prosess og gir maksimum krav for hver ressurs (gis på forhånd av prosessen). Følgende relasjoner må gjelde:

1. **Alle ressurser er allokert eller tilgjengelig**: $R_j = V_j + \sum_{i=1}^{n} A_{ij}$ for alle j
2. **Ingen prosess kan kreve mer enn totalt mengde ressurser**: $C_{ij} \leq R_j$ for alle j
3. **Ingen prosess allokeres flere ressurser enn det den originalt krevde**: $A_{ij} \geq C_{ij}$ for alle i, j

Vranglås-unngåelse policyen nekter å starte en ny prosess dersom ressurskravene til prosessen kan føre til en vranglås. En ny prosess P_{n+1} vil kun startes dersom:

$$R_j \geq C_{(n+1)j} + \sum_{i=1}^{n} C_{ij} \qquad \text{for alle } j$$

Altså, prosessen vil kun startes dersom det maksimale kravet hos alle nåværende prosesser pluss kravet hos ny prosess kan tilfredsstilles. Denne strategien er ikke optimal, fordi den antar at alle prosesser gjør maksimum krav samtidig.

Avslag på ressursallokering (Bankers algoritme)

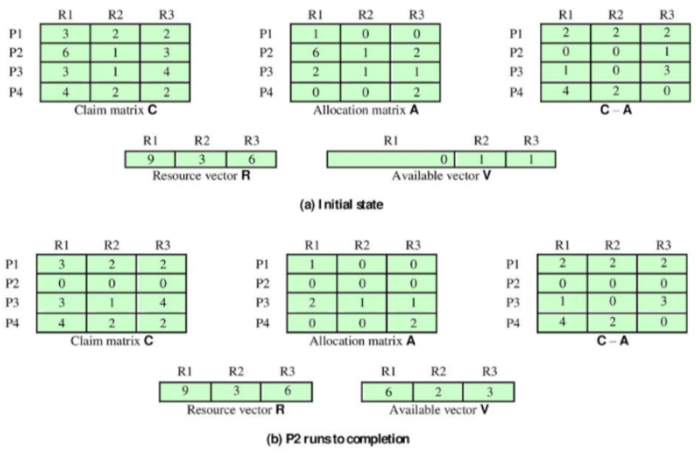
En ny ressurs-forespørsel aksepteres dersom problemet alltid kan unngås. **Bankers algoritme er en metode som forsøker å unngå vranglås ved å fokusere på tidspunktet for allokering av ressurser.** Systemet har n prosesser og m ressurstyper. Tilstanden til systemet reflekterer nåværende allokering av ressurser til prosessene, og den består av vektorene: Resource (R) og Available (V), og matrisene: Claim (C) og Allocation (A). Vi skiller mellom:

- **Trygg tilstand** – minst en sekvens av ressursallokeringer fører ikke til vranglås (dvs. alle prosessene kan fullføres uten at vranglås oppstår)
- **Utrygg tilstand** – alle sekvenser av ressursallokering fører til vranglås (merk: det betyr ikke at en vranglås har oppstått)

Bankers algoritme vil teste om en tilstand er sikker, ved å se om nåværende tilgjengelige ressurser kan tilfredsstille kravet hos en prosess gitt nåværende allokering (dvs. $C_{ij} - A_{ij} \leq V_j$, for alle j). Dersom det er tilfellet, kan prosessen fullføres og ressursene tildelt denne prosessen blir frigjort. Dermed vil algoritmen gjentas for gjenværende prosesser. Dersom dette kan gjentas helt til alle prosessene er fullført, vil tilstanden være trygg. Merk at dette er kun en simulering, så ressursene blir ikke faktisk tildelt prosessene.

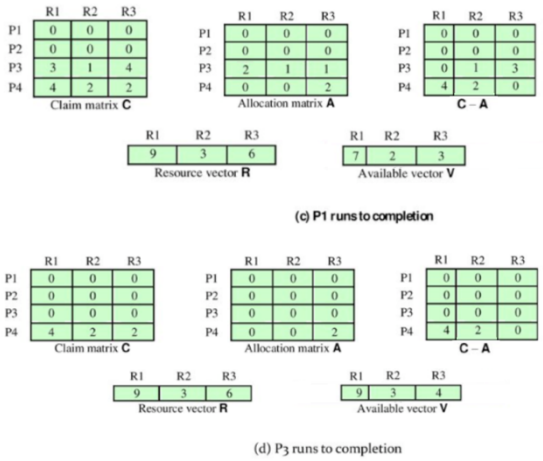
Figuren viser et eksempel for et system med fire prosesser og tre ressurstyper (R1, R2, R3).
For å finne ut om tilstanden er trygg blir følgende steg utført

1. Initial tilstand (figur a): allokeringen har blitt utført slik at en enhet med R2 og en enhet med R3 er tilgjengelig. Algoritmen ser at basert på nåværende allokeringsmatrise mangler P2 en enhet av R3, så P2 kan fullføres.
2. P2 fullføres (figur b): når P2 fullføres kan ressursene frigjøres (dvs. returnerer til V). Algoritmen ser at basert på nåværende allokeringsmatrise og tilgjengelige ressurser, så kan P1 fullføres.

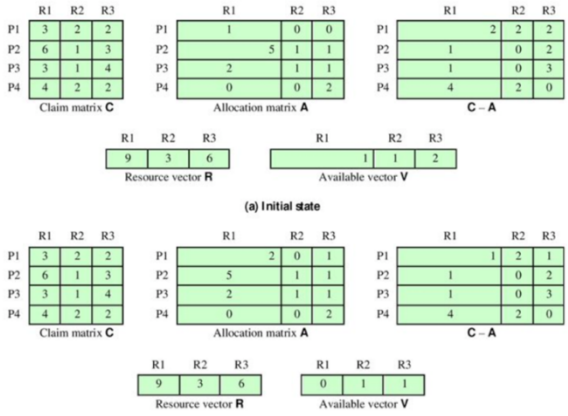


3. P1 fullføres (figur c): P3 kan fullføres
4. P3 fullføres (figur d): P4 kan fullføres

Siden det finnes en sekvens med ressursallokeringer som vil gjøre at vranglås kan unngås (alle prosesser kan fullføres), vil initial tilstand på figur a være en trygg tilstand.



Vranglås-unngå strategien er at når en prosess ber om tilgang til et sett med ressurser, anta at forespørselen akseptere og sjekk om resulterende tilstand er trygg. Hvis tilstanden er trygg kan tilgangen gis, mens hvis ikke vil prosessen blokkeres helt til det blir trygt å gi tilgang.



Dersom nåværende tilstand er vist på figur a og prosess P2 ber om en enhet R1 og en enhet R3, og vi antar at dette aksepteres, vil vi få tilstanden ved figur a på forrige side. Vi har allerede vist at denne er trygg, så derfor kan forespørselen aksepteres.

Dersom nåværende tilstand er vist på figur a og prosess P1 ber om en enhet R1 og en enhet R3, og vi antar at dette aksepteres, vil vi få tilstanden på figur b. Dette er ikke en utrygg tilstand, fordi alle prosesser krever en enhet R1 og det er ingen tilgjengelig R1 enhet. Denne forespørselen blir derfor avslått og P1 blir dermed blokkert. **Legg merke til at dette ikke er en vranglås**, for eksempel hvis P1 forsetter å kjøre kan det hende den frigir en R1 og R3 før den trenger disse igjen. Hvis det skjer vil systemet returnere til en trygg tilstand. **Vranglås-unngåelse vil ikke forutsi vranglås med full sikkerhet, den vil kun si noe om forventet mulighet for vranglås og unngå denne**.

OBS: situasjonen er ulovlig hvis request overgår opprinnelig krav gitt nåværende allokering, umulig hvis det ikke er tilstrekkelig ressurser og utrygg hvis det kan fører til vranglås

Figuren viser implementasjon av vranglås-unngåelse. Tilstanden gis av *state*, mens request[*] gir ressursene etterspurt av prosess i. Metoden vil utføre følgende steg:

1. Det sjekkes at ressurs-etterspørsel ikke overgår opprinnelig krav hos prosessen (dvs. lovlig/ulovlig)
2. Hvis request er gyldig, vil neste steg være å bestemme om den er mulig (dvs. nok tilgjengelig ressurser). Hvis ikke vil prosessen suspenderes.
3. Hvis etterspørsel kan innfris, vil neste steg være å bestemme om det er trygt. Dette innebærer å lage en forsøkstilstand newstate som gis til bankers algoritme (figur c). Hvis den er utrygg vil prosessen suspenderes
4. Hvis etterspørselen er trygg, vil den aksepteres og ressursen allokeres prosessen.

```
struct state {
        int resource[m];
        int available[m];
        int claim[n][m];
        int alloc[n][m];
}
```

(a) global data structures

```
if (alloc [i,*] + request [*] > claim [i,*])
    < error >;                                  /* total request > claim*/
else if (request [*] > available [*])
    < suspend process >;
else {                                          /* simulate alloc */
    < define newstate by:
    alloc [i,*] = alloc [i,*] + request [*];
    available [*] = available [*] - request [*] >;
}
if (safe (newstate))
    < carry out allocation >;
else {
    < restore original state >;
    < suspend process >;
}
```

(b) resource alloc algorithm

```
boolean safe (state S) {
    int currentavail[m];
    process rest[<number of processes>];
    currentavail = available;
    rest = {all processes};
    possible = true;
    while (possible) {
        <find a process Pk in rest such that
            claim [k,*] - alloc [k,*] <= currentavail;>
        if (found) {                            /* simulate execution of Pk */
            currentavail = currentavail + alloc [k,*];
            rest = rest - {Pk};
        }
        else possible = false;
    }
    return (rest == null);
}
```

(c) test for safety algorithm (banker's algorithm)

Fordelen med vranglås-unngåelse er at det ikke er nødvendig å fjerne eller rulle tilbake prosesser, og det er mindre begrensende (og dermed mer effektiv) enn vranglås-umuliggjøring. Ulemper er at (1) maksimalt krav hos hver prosesser må være kjent på forhånd, (2) prosesser som vurderes må være uavhengige (dvs. rekkefølge hos utføring er ubetydelig), (3) det må være et fast antall ressurser å allokere og (4) ingen prosess kan avsluttes mens den holder ressurser.

6.4 Oppdage og rette opp vranglås

Umuliggjøring av vranglås er svært konservativ, siden de løser problemet ved å begrense tilgangen til ressurser. **Oppdagelse av vranglås er det motsatte, siden det ikke setter noen begrensninger på ressursaksess eller prosesshandlinger**. **Etterspurte ressurser blir tildelt når det er mulig og OS vil periodisk gjennomføre en algoritme som detekterer sirkulær venting.**

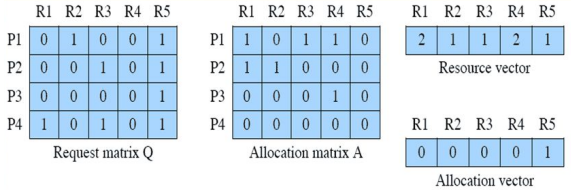
Oppdage vranglås

Sjekken etter vranglås kan utføres for hver ressursforespørsel eller mindre frekvent, avhengig av hvor sannsynlig det er at en vranglås oppstår. Hvis det sjekkes ved hver ressursforespørsel kan algoritmen være enklere og vranglås kan oppdages tidligere, men det krever mye prosessortid. Vi ser på en algoritme som bruker matrisen Q, der Q_{ij} er mengde ressurser av type j som er etterspurt av prosess i. Algoritmen vil markere prosesser som ikke er en del av en vranglås og alle prosesser er initialt umarkert. Følgende er stegene:

1. Marker hver prosess som har en rad i A-matrisen med kun 0'ere. En prosess som ikke har noen tildelte ressurser kan ikke delta i en vranglås
2. Lag en midlertidig vektor $W = V$
3. Finn en indeks i, slik at prosess i er umarkert og i-ende rad av Q er mindre eller lik W, altså $Q_{ik} \leq W_k$ for $1 \leq k \leq m$.
    a. Hvis ingen rad eksisterer, terminer algoritmen
    b. Hvis en rad eksisterer, marker prosess i og legg til prosessens rad i A-matrisen til W, altså $W_k = W_k + A_{ik}$. Gjenta steg 3.

En vranglås vil eksistere kun his det er en umarkert prosess ved enden av algoritmen. Settet med umarkerte rader vil være settet med prosesser som deltar i vranglåsen. En prosess blir markert når algoritmen finner ut at tilgjengelige ressurser kan oppfylle etterspørselen til prosessen, altså prosessen kan fullføres. Merk at algoritmen vil kun gi om det eksistere en nåværende vranglås, og den vil ikke gi om en vranglås oppstår i fremtiden.

Følgende er resultatet av å bruke algoritmen på verdiene gitt av figuren:



1. Marker P4, fordi P4 har ingen tildelte ressurser
2. Sett $W = (0\ 0\ 0\ 0\ 1)$
3. Ved $i = 3$ ser algoritmen at $Q_{3k} \leq W_k$ for $k = 0, 1, 2, 3, 4$. Derfor kan P3 markeres og $W = W + A_{3k} = (0\ 0\ 0\ 0\ 1) + (0\ 0\ 0\ 1\ 0) = (0\ 0\ 0\ 1\ 1)$
3. Hverken $i = 1$ eller $i = 2$ har rad i Q som er mindre eller lik W, så de kan ikke markeres

Algoritmen konkluderer med at P1 og P2 er umarkert, noe som betyr at disse er i vranglås med hverandre.

Rette opp vranglås

Når en vranglås har blitt oppdaget, trengs det en strategi for å rette opp vranglåsen. Mulige tilnærminger er:

1. Aborter alle prosessene i vranglåsen
2. Rull tilbake hver prosess i vranglåsen til forrige definerte checkpoint og restart prosessene.
3. Suksessivt aborter prosesser i vranglås helt til vranglåsen ikke eksisterer lenger.
4. Suksessivt fjern tilgang på ressurser helt til vranglåsen ikke eksisterer lenger. En prosess der ressursen fjernes må rulles tilbake til punktet før den fikk tilgang til ressursen

For tilnærming 3 eller 4 bør man velge prosessen som har forbrukt minst prosesseringstid, produsert minst output, mest gjenværende tid, minst allokert ressurser eller lavest prioritet.

6.5 En integrert vranglås strategi

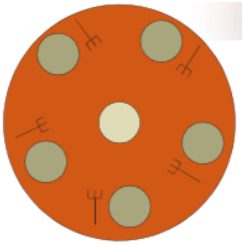
Alle strategiene for å håndtere vranglås har styrker og svakheter. **Ofte er det mest effektiv å integrere de ulike strategiene i OS-designet for ulike situasjoner.** En foreslått metode er å

gruppere ressurser i ulike ressursklasser og bruk lineær ordning (s. 54) på klassene for å unngå vranglås mellom disse. Innenfor ressursklassen kan man bruke algoritmen som er mets passende for denne klassen (eks: basert på hvor sannsynlig vranglås er). Et eksempel på ressursklasser er:

- **Swappable space** – blokker med sekundærminne som brukes av prosesser som blir swapped. Denne klassen kan bruke umuliggjøring ved å kreve at alle ressurser blir allokert samtidig (dvs. hold&wait). Nyttig hvis maksimum lagring er kjent (ofte tilfellet).
- **Prosessressurser** – enheter som kan tildeles (eks: filer). Denne klassen kan bruke unngåelse, siden det er rimelig å forvente at prosesser oppgir på forhånd hvilke ressurser de trenger fra denne klassen. Lineær ordning er også mulig.
- **Hovedminne** – kan tildeles prosesser i pages eller segmenter. Denne klassen kan bruke umuliggjøring ved å tillate fjerning av tildelte ressurser som swapped prosesser holder
- Interne ressurser – slik som IO-kanaler. Denne klassen kan bruke umuliggjøring ved ordning av ressurser

6.6 Dining philosophers problemet

Det er fem filosofer som bor i et hus der et bord er satt til dem (se figur). Hver filosof trenger to gafler for å kunne spise og de vil spise når de blir sultne (dvs. starter ikke nødvendigvis samtidig). Algoritmen må tilfredsstille gjensidig utelukkelse (to filosofer kan ikke bruke samme gaffel samtidig), i tillegg til å utelukke vranglås og utsulting.



Hver trenger to gafler

```
/* program diningphilosophers */
semaphore fork[5] = {1};
semaphore room = {4};
int i;
void philosopher (int i)
{
    while (true) {
      think();
      wait (room);
      wait (fork[i]);
      wait (fork [(i+1) mod 5]);
      eat();
      signal (fork [(i+1) mod 5]);
      signal (fork[i]);
      signal (room);
    }

}
void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher (2),
          philosopher (3), philosopher (4));
}
```

Løsning med semaforer

Semaforer kan brukes for å implementere at hver filosof vil først plukke opp gaffelen til venstre og deretter gaffelen til høyre. Når filosofen er ferdig med å spise, blir gaflene erstattet. Dersom alle filosofene blir sultne samtidig vil dette føre til en vranglås. En løsning kan være å kun tillate fire filosofer til bordet av gangen (se figur, semafor room = 4 legger begrensningen). Da vil minst en filosof ha tilgang til to gafler og kan fullføres for å videre frigjøre gafler. Dermed vil man forhindre vranglås og utsulting.

Løsning med monitor

Løsningen kan bruke en monitor ved å definere en vektor med fem betingelsesvariabler, én per gaffel, som vil la filosofer vente på at gaffelen blir ledig. I tillegg er det en boolean vektor som gir tilgjengelighetsstatusen til hver gaffel (*true* betyr at gaffelen er ledig). Monitoren består av to prosedyrer:

1. get_forks – brukes av filosofen for å gripe venstre og høyre gaffel. Hvis en av gaflene er utilgjengelig vil filosofen blokkeres på passende betingelsesvariabelen. Dermed kan en annen filosof entre monitoren
2. release_forks – brukes for å gjøre to gafler tilgjengelig.

Merk at løsningen ligner semafor-løsningen (filosof griper venstre og deretter høyre gaffel), **Monitoren vil likevel ikke være utsatt for vranglås, fordi det er kun én filosof som kan forsøke å gripe to gafler i monitoren av gangen.** Eks: den første filosofen som entrer monitoren garantert få begge gaflene, før en annen filosof forsøker å gripe en av dem.

```
monitor dining controller;
cond ForkReady[5];          /* condition variable for synchronization */
boolean fork[5] = {true};       /* availability status of each fork */

void get_forks(int pid)         /* pid is the philosopher id number */
{
   int left = pid;
   int right = (++pid) % 5;
   /*grant the left fork*/
   if (!fork[left])
      cwait(ForkReady[left]);          /* queue on condition variable */
   fork[left] = false;
   /*grant the right fork*/
   if (!fork[right])
      cwait(ForkReady[right]);         /* queue on condition variable */
   fork[right] = false:
}
void release_forks(int pid)
{
   int left = pid;
   int right = (++pid) % 5;
   /*release the left fork*/
   if (empty(ForkReady[left])      /*no one is waiting for this fork */
      fork[left] = true;
   else                        /* awaken a process waiting on this fork */
      csignal(ForkReady[left]);
   /*release the right fork*/
   if (empty(ForkReady[right])     /*no one is waiting for this fork */
      fork[right] = true;
   else                        /* awaken a process waiting on this fork */
      csignal(ForkReady[right]);
}
```

```
void philosopher[k=0 to 4]          /* the five philosopher clients */
{
   while (true) {
      <think>;
      get_forks(k);          /* client requests two forks via monitor */
      <eat spaghetti>;
      release_forks(k);      /* client releases forks via the monitor */
   }
}
```

6.7-6.11 Samtidighet for de ulike operativsystemene (s. 313-331)

Vi ser på et overblikk på samtidighetsmekanismer for de ulike typene OS.

UNIX samtidighet (s. 313-315)

UNIX støtter flere mekanismer for interprosess-kommunikasjon (IPC) og synkronisering. Pipes (rør), meldinger og delt minne brukes for å kommunisere data mellom prosesser, mens semaforer og signaler brukes for å trigge handlinger av andre prosesser. Viktige elementer er:

- **Pipes (rør)** – en sirkulær buffer som lar to prosesser kommunisere på producer-consumer modellen, ved at det er en FIFO-lø som en prosess kan skrive til og en annen prosess kan lese fra. En prosess som forsøker å skrive til pipen utføres umiddelbart dersom det er plass i bufferen, hvis ikke blokkeres den. En leseprosess vil blokkeres hvis den forsøker å lese mer bytes enn det er i pipen, ellers vil den umiddelbart utføres. OS sørger for gjensidig utelukkelse, slik at kun en prosess aksesserer pipen av gangen.
- **Meldinger** – hver prosess har en unik meldingskø som fungerer som en mailboks. Mottaker kan motta meldinger i FIFO eller ordne etter meldingstype (gis av sender)
- **Delt minne** – den raskeste formen for interprosess-kommunikasjon gitt av UNIX er delt minne. Dette er en blokk med virtuelt minne som deles av flere prosesser. Tilgangen er read-only eller read-write, og det bestemmes for hver prosess. Gjensidig utelukkelse er ikke en del av det delte lageret, så det må kontrolleres av prosesser som bruker det.
- **Semaforer** – UNIX bruker en generalisert versjon av semWait og semSignal semaforer. Kjernen utfører alle operasjoner atomisk, altså ingen andre prosesser kan aksessere semaforen før alle operasjonene er fullført. Semaforen har derfor en tilhørende kø med blokkerte prosesser. Generalisering av semaforer gir fleksibilitet i synkronisering og koordinering av prosesser.
- **Signaler** – et signal (bit) er en programvaremekanisme som informerer en prosess om forekomsten av asynkrone hendelser. Det ligner et hardware-avbrudd og bruker ikke prioriteter. Signalene behandles derfor likt, så signaler som oppstår samtidig gis uten en bestemt ordning til prosessen. Prosesser kan sende signaler til hverandre eller kjernen kan sende signaler internt. En prosess responderer til et signal ved å utføre standardhandling (eks: terminere), utføre signal-håndteringsfunksjon eller ignorerer signalet.

LINUX samtidighet (s. 315-323)

LINUX inkluderer alle samtidsmekanismene hos UNIX SVR4, som pipes, meldinger, delt lager og signaler. I tillegg støtter LINUX en type signaler som kalles real-time (RT) signaler. Disse er annerledes enn de vanlige UNIX-signalene på tre måter: (1) de støtter prioritet ved levering, (2) signaler kan vente i kø og (3) RT-signaler kan inneholde verdi eller peker. Andre samtidsmekanismer støttet av LINUX er:

- **Atomiske operasjoner** – LINUX har et sett med operasjoner som garanterer atomisitet på en variabel, og de brukes for å unngå race condition.
- **Spinlocks** – den vanligste teknikken i LINUX for å beskytte kritiske seksjoner. En spinlock bygges på en heltallslokasjon i minnet som sjekkes av hver tråd før den entrer kritiske seksjon. Hvis verdien er 0, setter tråden verdien til 1 og entrer kritisk seksjon, mens hvis den ikke er 0 vil den kontinuerlig sjekke verdien til den blir 0. Spinlocks er lette å bruke, men har ulempen at den setter flere tråder i aktiv venting. Den har også innretning for lesere og skrivere som tillater høyere grad av samtidighet.
- **Semaforer** – på brukernivå implementerer LINUX samme semaforgrensesnitt som UNIX, mens internt har LINUX egen funksjonalitet for semaforer. Disse semaforene implementeres som funksjoner i kjernen, og er dermed mer effektive enn de på brukernivå. Det er tre typer semaforer på kjernenivå: binære semaforer, generelle semaforer og read-write semaforer.

- **Barrierer** – sørger for at rekkefølgen på minneaksess blir som spesifisert (eks: a=1 ;b=a vil gjøre at rekkefølgen er viktig). Det er relatert til maskininstruksjonene load og store, og det dikterer oppførselen til prosessoren og kompilatoren.

Solaris samtidighet (s. 323-326)

I tillegg til samtidsmekanismene hos UNIX SVR4, vil Solaris støtte følgende metoder for synkronisering av brukertråder og kjernetråder:

- **Gjensidig utelukkelse (MUTEX-låser)** – en mutex-lås brukes for å sikre at kun en tråd kan aksessere en ressurs av gangen. Tråden som låser mutex må være den samme som låser den opp
- **Spesialiserte semaforer** – Solaris implementerer tellende semaforer for å blokkere og avblokkere tråder etter behov
- **Readers/writers-lås** – lar flere tråder ha samtidig lese-aksess til et objekt, men lar kun én skriver aksessere objektet av gangen (dvs. ekskluderer andre skrivere og alle lesere)
- **Monitoraktige betingelsesvariabler** – brukes for å vente inntil en bestemt betingelse er sann. De må brukes sammen med mutex-lås for å implementere en monitor.

Windows samtidighet (s. 326-329)

Windows gir synkronisering av tråder som en del av objekt-arkitekturen, ved at prosesser og tråder direkte implementeres som objekter som har innebygde muligheter for synkronisering. Ventefunksjonen og de viktigste metodene for synkronisering er:

- **Basis ventefunksjon** – gjør at en tråd kan blokkere sin egen utførelse og ikke returnere før et spesifikt krav har blitt oppfylt (passiv venting = bruker ingen prosessortid). Type ventefunksjon bestemmes av kravet og dette benyttes av Dispatcher objektet.
- **Dispatcher objekt** – bruker normale synkroniseringsobjekter, som mutex, semaforer, osv. (se tabell s. 327).
- **Brukerorienterte kritiske regioner** – gir synkroniseringsmuligheter lignende mutex-objekter, bortsett fra at kritiske regioner kun kan brukes av trådene i en enkelt prosess
- **Slanke leser/skriver-låser og betingelsesvariabler** – slim reader/writer (SRW) låser lar tråder til en enkelt prosess aksessere delte ressurser. De er optimalisert for hastighet og okkuperer veldig lite minne (slank). I tillegg bruker Windows betingelsesvariabler som deklareres av en prosess, og kan brukes med kritiske regioner eller SRW-låser.
- **Låsfri synkronisering** – brukes maskinvarefunksjonalitet for å garantere at en minneadresse kan leses, modifiseres og skrives i en atomisk operasjon.

Android samtidighet (s. 330-331)

Android bruker SVR4/Solaris sine metoder for prosesser og tråder, men legger til en ekstra funksjonalitet i kjernen for interprosess-kommunikasjon (IPC), kalt Binder. En Binder består av en lettvekts LPC/RPC som brukes til å formidle interaksjon mellom to prosesser. RPC-mekanismen virker mellom to prosesser som er ved samme system, men kjører på ulike virtuell maskiner.

Del 4 – Håndtering av lager

Denne delen av kompendiet ser på håndtering av lager (minne), og det inkluderer:

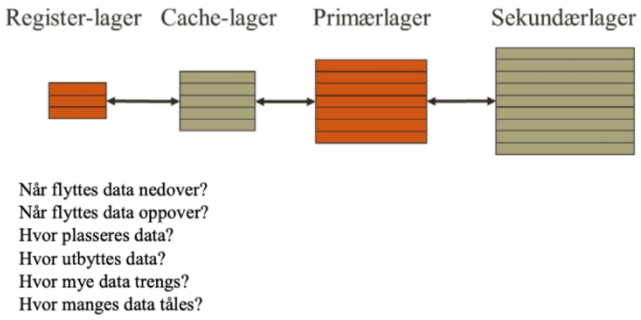
- **Kapittel 7** – Lagerhåndtering
- **Kapittel 8** – Virtuelt lager

Kapittel 7 – Lagerhåndtering

Lagerhåndtering innebærer å holde oversikt over og tilby aksess til ulike deler av kode/data for ulike prosesser/tråder ved forskjellige tidspunkt. I de fleste datamaskiner vil prosessorer være kritiske ressurser som bør holdes aktive hele tiden, noe som krever at flere (del)programmer er tilgjengelig for utføring i primærlageret (hovedminnet) til enhver tid.
Dette kan gjøres ved å:

1. **Holde et mindre antall totalprogrammer i primærlageret og resten på sekundærlageret**, mens man kontinuerlig bytter ut programmer som ikke er under utføring (dvs. swapping til og fra sekundærlager).
2. **Holde et større antall delprogrammer i primærlageret og resten på sekundærlageret**, der delprogrammer som antas å ikke bli brukt i nærmeste fremtid byttes ut med delprogrammer som skal brukes i nærmeste fremtid. Dette gir større virtuelt primærlager enn reelt primærlager.

Begge varianter medfører ulike oppgaver som må håndteres av operativsystemet, i form av lagerhåndtering. I tillegg trengs det maskinvarestøtte i form av prosessoren(e). OS trengs for å håndtere oppgavene som må utføres for at bestemt kode eller data skal være i primærlageret ved spesifikke tidspunkt og at dette tildeles prosesser/tråder etter behov. Dette innebærer å bruke lagerhierarkiet som har store forskjeller mht. hastighet, kapasitet, varighet og kostnad.



7.1 Krav for lagerhåndtering

Lagerhåndtering skal innfri følgende krav: relokalisering, beskyttelse, deling, logisk organisering og **fysisk organisering**. Tabellen viser noen viktige begrep.

Ramme (*frame*)	En fast-lengde blokk i primærlager
Side (*page*)	En fast-lengde blokk i sekundærlager som kan midlertidig kopieres inn i en ramme i primærlager (paging)
Segment	En variabel-lengde blokk i sekundærlager. Hele segmentet kan kopieres inn i et tilgjengelig område i primærlager (segmentering) eller det kan deles opp i flere sider som kan individuelt kopieres inn i hovedminne (kombinert segmentering og paging)

Relokasjon

For å maksimere utnyttelsen av prosessoren er det mulig å swappe prosesser inn og ut av primærlageret (hovedminnet), slik at man kan lage en pool med prosesser som er klare for utførelse. Når en prosess blir byttet ut vil det være svært begrensende dersom den må plasseres ved samme minnelokasjon når den byttes inn igjen. **I stedet brukes relokasjon for at prosessen skal kunne plasseres ved et annet området av minnet.** Dette gjør at vi ikke vet på forhånd hvor programmet er plassert. OS må vite adressen til blant annet prosessorkontrollblokken, mens prosessoren må håndtere minnereferanser i program.

Merk: prosesser kan også flyttes mellom ulike lokasjoner i primærlageret

Maskinvaren hos prosessoren og OS må derfor ha en måte å oversette minnereferanser i kode til fysiske minneadresser som reflekterer nåværende lokasjon til programmet i primærlageret.

Beskyttelse

Hver prosess bør beskyttes mot uønsket inferens av andre prosesser, så program i andre prosesser bør ikke ha muligheten til å referere minnelokasjoner i prosessen for lesing eller skriving uten tillatelse. Relokasjon og dynamisk utregning av adresser ved run time gjør at det ikke er tilstrekkelig å sjekke adresser ved kompilering. **Alle minnereferanser hos en prosess må sjekkes ved run time for å sikre at de kun refererer til minnelokasjoner som er tildelt prosessen**. Et program i en prosess vil ikke kunne aksessere data i en annen prosess uten spesielle ordninger. Merk at beskyttelse implementeres på maskinvarenivå av prosessoren istedenfor OS, fordi OS kan ikke forutsi alle minnereferanser hos et program og det ville ha vært ineffektivt om mulig. Mekanismer som støtter relokasjon støtter også beskyttelse.

Merk: kompilering er når koden oversettes til utførbar kode, mens runtime er når utførbar kode kjøres/utføres.

Deling

Prosesser som samarbeider på en oppgave eller hører til samme program, kan ha behov for å aksessere samme datastruktur. Beskyttelsesmekanismen må derfor ha nok fleksibilitet til at flere prosesser kan aksessere samme del av primærlageret. Lagerhåndteringen må tillate aksess til delt minne uten å kompromittere den essensielle beskyttelsen. Mekanismer som støtter relokasjon støtter også deling.

Logisk organisering

Primær- og sekundærlageret er organisert som et lineært eller en-dimensjonalt adresserom som består av en sekvens av bytes eller ord. De fleste programmene er derimot ofte organisert som moduler. **Dersom OS og maskinvaren kan effektivt håndtere programmer og data som moduler, vil det ha følgende fordeler:**

1. Moduler kan skrives og kompileres uavhengig
2. Ulike grader med beskyttelse (eks: read only) kan gis til ulike moduler med lite ekstra overhead
3. Det er mulig å introdusere mekanismer slik at moduler kan deles mellom prosesser. Deling på modulnivå korresponderer også med måten brukeren ser på problemet, slik at det blir lettere for brukere å spesifisere ønsket deling.

Segmentering blir ofte brukt for å oppnå dette.

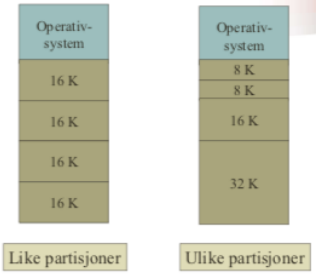
Fysisk organisering

Lageret i datamaskiner er organisert i minst to nivåer: primær- og sekundærlager. En utfordring med denne ordningen er organiseringen og flyten av informasjon i systemet. Ansvaret for å flytte informasjon mellom de to nivåene ligger på systemet, og det er essensen ved lagerhåndtering.

7.2 Lagerpartisjonering

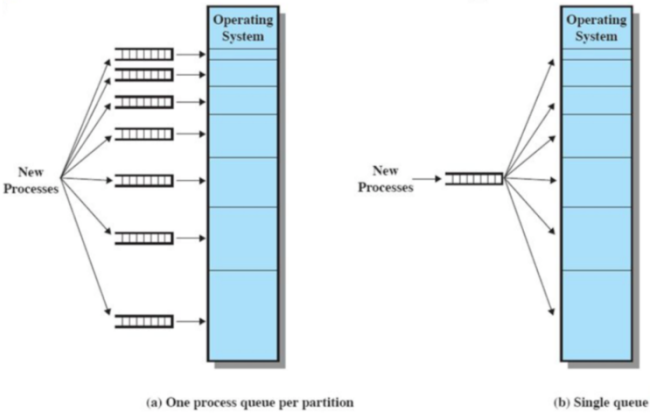
Hovedoppgaven til lagerhåndteringen er å hente prosesser inn i hovedminnet (primærlager), slik at de kan utføres av prosessoren. I nesten alle moderne multiprogrammeringssystem vil dette involvere virtuelt minne som er basert på bruken av segmentering og/eller paging. Før vi ser på disse ser vi på enklere teknikker som ikke involverer virtuelt minne. En av disse er partisjonering som kan brukes i flere varianter. Vi ser også på enkel paging og enkel segmentering uten virtuelt minne (brukes ikke alene, men nyttig for videre forståelse). Som regel vil OS okkupere en fast del av primærlageret, mens resten er tilgjengelig for å brukes av prosesser.

Fast partisjonering (*fixed*)



Den enkleste formen for lagerhåndtering er å dele tilgjengelig lager inn i deler med faste grenser (blir ikke brukt lengre). Delene kan ha lik eller ulik størrelse. Hvis alle partisjoner har lik størrelse vil det introdusere to problemer: (1) et program kan være for stort til å passe i en partisjonering, så det må designes slik at kun deler av programmet trenger å være i hovedminnet av gangen og (2) det fører ofte til intern fragmentering, siden programmet som hentes inn kan være mindre enn partisjonen (dvs. all plass blir ikke utnyttet). Begge disse problemene kan reduseres, men ikke løses med bruk av partisjoner med ulik størrelse.

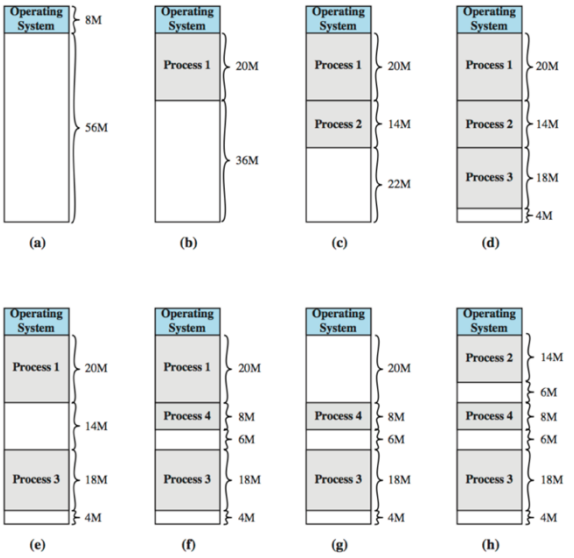
For partisjoner med lik størrelse kan en prosess lastes inn i en partisjon så lenge det er ledig plass. Siden alle partisjoner har lik størrelse, spiller det ingen rolle hvilken partisjon som velges. Hvis alle partisjoner er lastet med prosesser som ikke er kjørbare, så vil en av disse bli swapped for å få plass til en ny prosess. Det er tidsstyringen som velger hvilken prosess som byttes ut (del 4). **For partisjoner med ulik størrelse er det to måter å fordele minnet. Den enkleste metoden er å tildele hver prosess den minste tilgjengelige partisjonen den passer i.**



Dette krever en kø for hver partisjonering som holder utbyttet prosesser for denne partisjonen (figur a). Dette vil minimere intern fragmentering, men det er ikke optimalt for systemet som en helhet (eks: hvis det kun er små prosesser, vil de store partisjonene stå ubrukt). **En mer egnet metode er å ha én kø for alle prosessene** (figur b). Når en prosess skal lastes inn i hovedminnet vil den velge den minste tilgjengelige partisjonen som prosessen har plass i. Hvis alle partisjoner er fulle, må det gjøres en swapping som kan være basert på størrelse, prioritet, prosesstilstand, osv.

Fordeler med fast partisjonering er at det er relativt enkelt og krever lite OS programvare og prosesseringsoverhead (ulik partisjonsstørrelse gir også noe fleksibilitet), mens ulemper er at antall partisjoneringer begrenser antall aktive prosesser i systemet og små prosesser vil ikke kunne utnytte plassen effektivt (= intern fragmentering).

Dynamisk partisjonering



Ved dynamisk partisjonering vil delene i lageret være av varierende lengde og antall (blir ikke brukt lengre). Når en prosess hentes inn i hovedminnet, blir den tildelt den eksakte mengden minne som den trenger. Dette fører etterhvert til **ekstern fragmentering**, der det dannes «hull» i hovedminnet som følger av at det ikke er plass til en prosess til (se figur). Det kalles **ekstern fragmentering**, siden hullene er eksterne til partisjonene (merk ved intern partisjonering er hullene inni partisjonene), og det gjør at utnyttelsen av minnet blir dårligere med tiden. Det kan løses vha. **kompaktering** (prosessene forskyves av OS), men det er tidkrevende, gir bortkastet prosessortid og krever en dynamisk relokasjon.

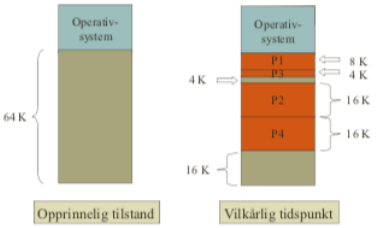
Siden kompaktering er tidskrevende, er det viktig at OS systemet bruker en god strategi for å plassere en prosess når flere blokker er ledige. De tre plasseringsalgoritmene som brukes er:

1. **Best-fit** – velger blokken som er nærmest i størrelse. Gir ofte dårligst ytelse, siden primærlageret ofte ender opp med deler som er for små til å allokeres (mer kompaktering)

2. **First-fit** – skanner minnet fra starten og velger første tilgjengelige blokk som er stor nok. Denne er enklest og vil ofte være best og raskest
3. **Next-fit** – skanner minnet fra forrige plassering og velger den neste tilgjengelige blokken som passer. Denne er som regel litt dårligere enn first-fit, siden det ofte gjør at største ledige blokk (ofte ved enden) blir fragmentert (mer kompaktering)

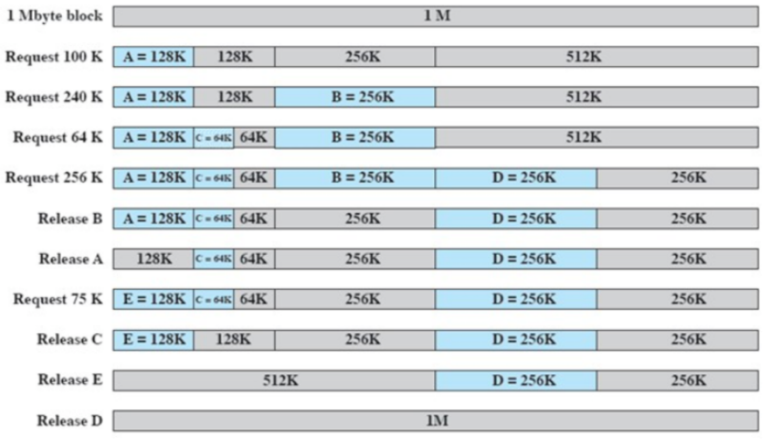
Fordeler med dynamisk partisjonering er at det unngår intern fragmentering, mens ulemper er at det innebærer ekstern fragmentering og ekstra overhead for kompaktering, og det er mer komplisert å opprettholde.

Buddy systemer

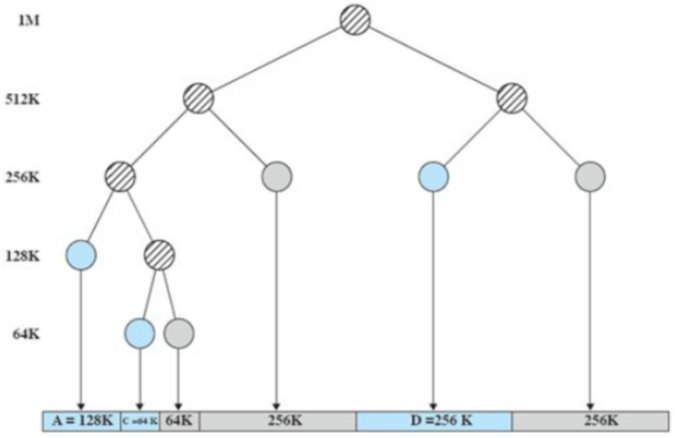


Buddy systemer er et kompromiss mellom fast og dynamisk partisjonering, der minneblokker kun er tilgjengelig i størrelse på 2^K ord, der $L \leq K \leq U$. I begynnelsen vil hele minnet være tilgjengelig og behandles som én blokk av størrelse 2^U. Hvis en aksessforespørsel s er slik at $2^{U-1} < s \leq 2^U$, så vil hele blokken allokeres. Hvis ikke vil blokken deles inn i to like *buddies* av størrelse 2^{U-1}. Hvis $2^{U-2} < s \leq 2^{U-1}$, så vil forespørselen tildeles en av *buddiesene*. Hvis ikke vil en av *buddiesene* deles i to igjen. Prosessen gjentas helt til den minste blokken som er større eller lik s blir laget og dermed tildelt s. Buddy systemet vil ha en liste over hullene (ikke-tildelte blokker). Dersom begge buddies blir tilgjengelige vil de slås sammen til en blokk (dvs. hvis to buddies er bladnoder, må minst en være tildelt.

```
void get_hole(int i)
{
  if (i == (U + 1)) <failure>;
  if (<i_list empty>) {
      get_hole(i + 1);
      <split hole into buddies>;
      <put buddies on i_list>;
  }
  <take first hole on i_list>;
}
```



Eksempel på Buddy system. Legg merke til at når E frigjøres blir først to buddies på 128K slått sammen til én på 256K og deretter blir to buddies på 256K slått sammen til én på 512K. Når D frigjøres vil systemet slås sammen til en blokk.



Trerepresentasjon av Buddy systemet etter B er frigjort.

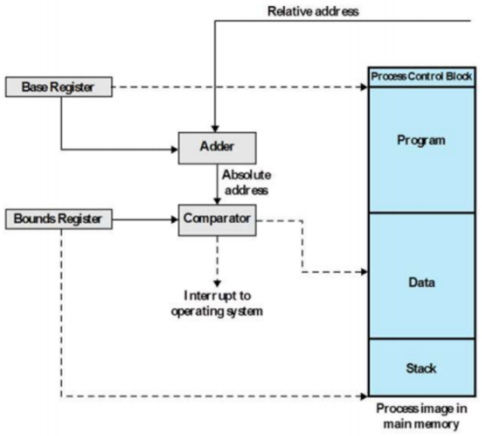
Fordelen ved Buddy system er at det vil inkludere fordeler fra både fast og dynamisk partisjonering, mens ulempen er at det også vil inkludere ulempene fra disse (eks: intern fragmentering siden størrelsen må være 2er-potens). Det blir brukt i enkelte parallelle systemer for å tildele og frigjøre parallelle program og i UNIX kjerne-minneallokering.

Relokasjon

Som nevnt tidligere må det være mulig å bytte ut prosesser fra hovedminne for å senere hente dem inn og ikke plassere dem ved samme lokasjon. Ved kompaktering vil også prosesser skifte lokasjon, selv om de fortsatt er i hovedminnet. **Adressen som refererer til en prosess (instruksjoner/data) vil altså ikke være fast.** For å løse denne utfordringen må man skille mellom ulike typer adresser:

1. **Logisk adresse** – en virtuell adresse som er uavhengig av nåværende tildeling av data i minnet (ses av bruker). Den generes av programmet ved kjøretid og må oversettes til en fysisk adresse før minneaksessen kan gjennomføres.

2. **Relativ adresse** – en type logisk adresse der lokasjonen gis ved å spesifisere avstanden fra et annet kjent punkt (ofte verdi i et prosessorregister)
3. **Fysisk adresse** – en faktisk lokasjon i hovedminnet (kan ikke ses av bruker)

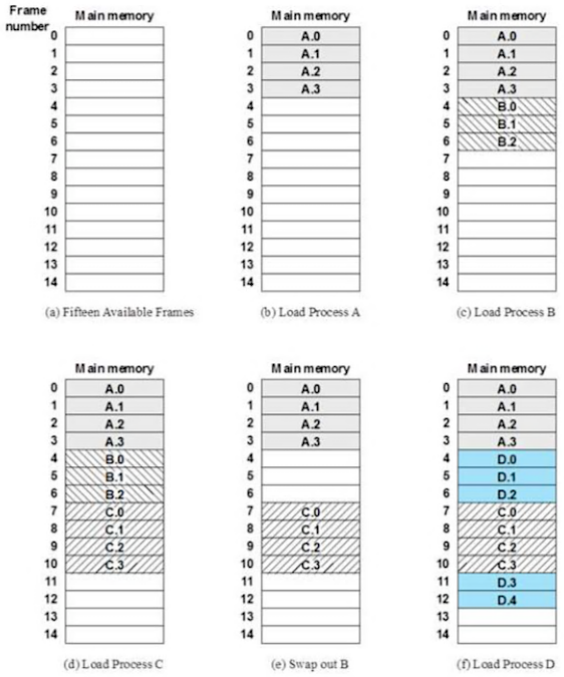


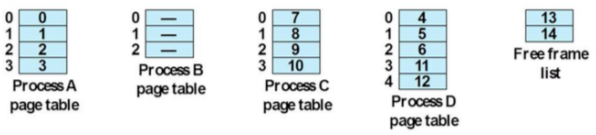
Som regel vil alle minnereferansene i en prosess være relativt til opphavet hos programmet som utføres i prosessen. **Det trengs derfor en maskinvaremekanisme som oversetter relative adresser til fysiske adresser i løpet av utføringen av prosessen** (se figur). Når en prosess tildeles Running-tilstand, vil base-registret i prosessoren lastes med startadressen til programmet i hovedminnet, mens bounds-registret lastes med sluttadressen (start og slutt settes når programmet lastes inn i hovedminnet). Når prosessoren utfører prosessen og mottar relative adresser vil den (1) finne fysisk adresse ved å legge til verdien i base-registret og (2) sammenligne med verdien i bounds-register for å sjekke at referansen er innenfor programmet. Hvis det er tilfellet vil instruksjonen utføres (hvis ikke vil det oppstå feil som OS må håndtere). **Denne løsningen gjør at program kan byttes inn og ut av minnet, og det gir beskyttelse, siden base- og bounds-registrene sørger for at hver prosess er trygg fra uønsket aksess fra andre prosesser** (med mindre de inneholder samme program).

7.3 Enkel paging

Ved paging blir hovedminnet delt inn i mange rammer (*frames*) av samme størrelse, mens prosesser deles inn i sider (*pages*) som er like store som rammene. Hver side kan dermed lastes inn i en ramme. Små prosesser kraver færre sider, mens store krever flere. **Enkel paging ligner fast partisjonering, men forskjellen er at partisjonene er små, et program kan okkupere flere partisjoner og disse trenger ikke å være sammenhengende. Paging unngår ekstern fragmentering, men er utsatt for intern fragmentering (ofte mindre pga. mindre partisjoner).**

Figuren viser bruken av sider og rammer. **Ved ethvert tidspunkt vil OS opprettholde en liste over ledige rammer**. For eksempel kan vi se på prosess A som består av fire sider. OS vil finne fire ledige rammer og laster de fire sidene inn i disse (figur b). Det samme skjer for prosess B og C. Prosess B blir senere suspendert og flyttes ut av hovedminnet. Senere blir alle prosessene blokkert, så OS flytter inn ny prosess D. Som vi kan se vil prosess D ikke lastes inn i en kontinuerlig samling av rammer (den deles opp). **OS vil opprettholde en sidetabell (*page table*) for hver prosess, og denne viser rammelokasjonen hos hver side i prosessen.** I programmet vil hver logiske adresse bestå av et sidenummer og en offset innenfor siden (dvs. gir page og lokasjon i page). Oversettelsen fra logisk til fysisk adresse blir fortsatt utført av prosessor-maskinvare, men **ved paging vil prosessoren motta logisk adresse (sidenummer, offset) og bruker dette med sidetabellen for å finne den fysiske adressen (rammenummer, offset).**

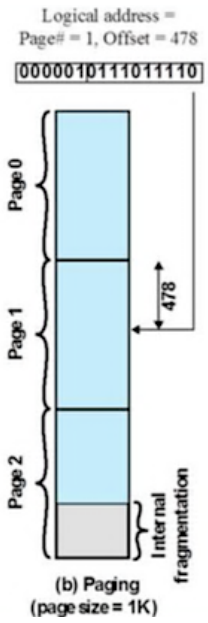




Figuren til venstre viser sidetabellene og listen over ledige rammer for eksempelet over. Sidetabellen inneholder en enhet per side, så den kan indekseres med sidenummeret (starter med side 0). Hver enhet i sidetabellen vil gi nummeret til rammen i hovedminnet som holder siden, dersom det eksiterer en slik ramme.

Beregning av fysisk adresse ved paging

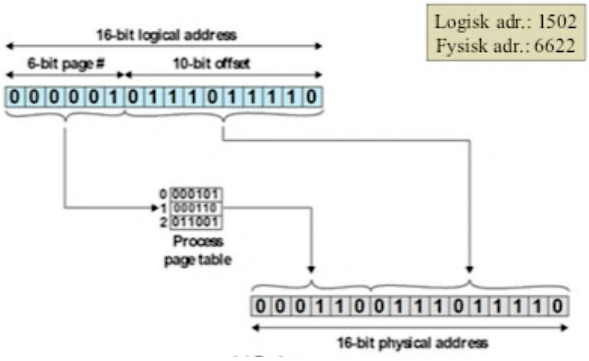
Når side- og rammestørrelsen er en toerpotens vil den relative adressen (refererer til programopphav) og logiske adressen (sidetall, offset) være den samme. Figuren viser et eksempel for 16-bit adresser og sidestørrelse på 1K = 1024 bytes. Den relative adressen 1502 er 0000010111011110 i binærform. Dersom sidestørrelsen er 1K trengs det et offset-felt på 10 bits, slik at 6 bits gjenstår til sidenummeret. Programmet kan dermed bestå av $2^6 = 64$ sider (husk: hver side må kunne ha unikt binært sidenummer). Den relative adressen 1502 korresponderer til en offset på 478 (0111011110) i page 1 (000001), som kombinert vil gi samme 16-bit nummer: 0000010111011110. Dermed har vi vist at relativ adresse er lik logisk adresse. Bruk av sidestørrelser i toerpotens gjør at den logiske adresseringen blir transparent til programmerere og at det blir lettere for maskinvare å utføre dynamisk oversettelse ved run time. **Dersom vi har en adresse på $m + n$ bits, der n venstre bits er sidenummer og m høyre bits er offset, blir følgende steg brukt for å oversette gitte logiske adresse (sidenummer, offset) til fysisk adresse (rammenummer, offset):**



Logisk adr.: 1502
Fysisk adr.: 6622

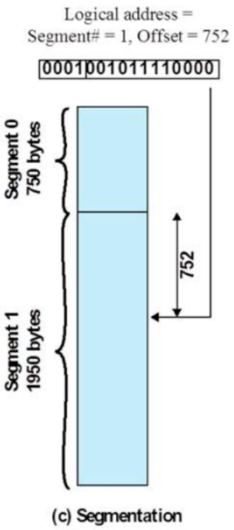
1. **Hent ut sidenummer** n fra logisk adresse
2. Bruk sidenummer som **indeks i sidetabellen** hos prosessen for å finne rammenummer k
3. Finn fysisk adresse ved å **fest offset til rammenummer**

Figuren viser prosessen for eksempelet over.



7.4 Enkel segmentering

Ved segmentering blir et program med assosiert data delt inn i et antall segmenter som kan ha ulike størrelser så lenge det er under gitte maksimum. I likhet med paging vil den logiske adressen ved segmentering være (segmentnummer, offset). **Enkel segmentering ligner dynamisk partisjonering, men forskjellen er at et program kan okkupere flere partisjoner og disse trenger ikke å være sammenhengende. Segmentering unngår intern fragmentering, men er utsatt for ekstern fragmentering (ofte mindre siden prosess deles opp i flere deler).**



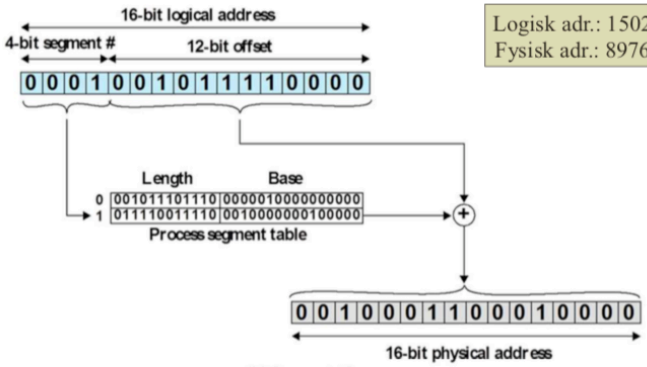
Beregning av fysisk adresse ved paging

Segmenter av ulike størrelser gjør at det ikke er noen enkle relasjoner mellom logisk og fysisk adresse. I likhet med paging vil segmentering bruke en segmenttabell for hver prosess og en liste over ledige blokker i hovedminnet. Hver enhet i tabellen gir startadressen til korresponderende segment i hovedminnet (hvis det eksisterer) og lengden til segmentet (unngår ugyldige adresser). Dersom vi har en adresse **på $m + n$ bits, der n venstre bits er segmentnummer og m høyre bits er offset, blir følgende steg brukt for å oversette gitte logiske adresse (sidenummer, offset) til fysisk adresse (rammenummer, offset):**

1. **Hent ut segmentnummer** n fra logisk adresse
2. Bruk segmentnummer som **indeks i segmenttabellen** hos prosessen for å finne fysiske startadresse hos segmentet
3. **Sammenlign offset (*m* høyre bits) med lengden til segmentet. Hvis offset er større eller lik lengden, vil adressen være ugyldig.**
4. Finn fysisk adresse ved å **summere fysiske startadresse med offset**

Figuren viser prosessen for eksempelet på figuren over til høyre. Legg merke til summeringen (0+0=0, 1+0=1 og 1+1=0 med carry 1 som flyttes til venstre).

Logisk adr.: 1502
Fysisk adr.: 8976



Paging vs. Segmentering

Paging er en utvidelse av fast partisjonering, ved at sidene og rammene har faste størrelser. Det vil unngå ekstern fragmentering, men er **utsatt for intern fragmentering**. Paging er **usynlig for programmereren**. **Segmentering er en utvidelse av dynamisk partisjonering**, ved at segmentene har varierende størrelser. Det vil unngå intern fragmentering, men er **utsatt for ekstern fragmentering**. Segmentering er ofte **synlig for programmereren** og kan brukes for å organisere programmer og data (eks: tildele programmer/data til ulike segmenter). Ulempen ved dette er at programmerer må være klar over maksimal segmentstørrelse. Begge vil sikre at prosessene er beskyttet, men det er lettere å oppnå ved segmentering (offset sammenlignes med lengden til segmentet).

Paging bruker maskinvarens faste oppdeling av lageret i rammer (dvs. tilpasset maskin/systemsynspunkt), mens segmentering bruker utviklerens oppdeling av programvare i moduler (dvs. tilpasset program/brukersynspunkt). Paging kan brukes når den forhåndsdefinerte rammestørrelsen ikke gir for mye intern fragmentering, mens segmentering brukes hvis søking etter hull til segmentene ikke blir for ressurskrevende. **De kan kombineres for å oppnå tonivå organisering (eks: kode/data deles opp i segmenter som igjen deles opp i sider**). Dette gir fordelene og ulempene fra begge typene. Det krever større plass for tabeller og mer oppslagsbehov, så det kan velges hvis man har råd til dette.

OBS: Kapittel 7 har appendix om lasting av program inn i hovedminnet og linking mellom moduler (s. 363-369).

Kapittel 8 – Virtuelt lager

Virtuelt lager er en lagerhåndteringsfunksjon for et operativsystem, der minneallokeringen lar sekundærminnet adresseres som om det var en del av primærminnet. Det er en måte å sikre at flere (del)programmer er tilgjengelig for utføring i primærlageret til enhver tid. Virtuelt lager bygges oppå andre lagerhåndteringsaktiviteter som segmentering, paging eller kombinasjoner av disse. Det krever også både maskinvarestøtte fra prosessorer eller kjerner og programvarestøtte i form av kompilatorer og OS. **Virtuelt lager kan løse alle lagerhåndteringsproblem, men det forutsetter altså støtte fra andre komponenter**. **Virtuelt lager blir dermed mer et konsept enn et system i seg selv.**

Adressene som brukes av programmet for å referere til minnet skilles fra de fysiske adressene som brukes av lagersystemet, og programgenererte adresser blir automatisk oversatt til korresponderende maskinadresser. **Størrelsen til det virtuelle lageret blir begrenset av adresseringsskjemaet og mengde tilgjengelig sekundærlagring, og ikke av antall lokasjoner i primærlageret.** Noen viktige begrep:

- **Virtuell adresse** – adressen som er tildelt en lokasjon i det virtuelle lageret, slik at lokasjonen kan aksesseres som om den var en del av hovedminnet
- **Virtuelt adresserom** – virtuell lagring tildelt en prosess
- **Adresserom** – området med minneadresser tilgjengelig for en prosess
- **Reell adresse** – adressen til en lokasjon i hovedminnet

8.1 ser på maskinvareaspektet ved virtuell lagring, mens 8.2 ser på virtuell lagring med OS.

8.1 Maskinvare og kontrollstrukturer

To egenskaper ved paging og segmentering har lagt grunnlaget for et viktig gjennombrudd innen minnehåndtering:

1. Alle minnereferanser innenfor en prosess er logiske adresser som dynamisk oversettes til fysiske ved run time (kjøretid). Dette gjør at prosesser kan byttes inn og ut av primærlageret og okkupere ulike områder av minnet under utførelse.
2. Prosesser kan deles inn i mange biter (sider/segmenter) som ikke trenger å ligge sammenhengende i minnet (mulig pga. oversettelse og side-/segmenttabell)

Disse to egenskapene la grunnlaget for at det ikke er nødvendig å ha alle sidene eller segmentene til en prosess i hovedminnet i løpet av utføring. Utførelsen kan fortsettes så lenge delen som holder neste instruksjon og dataen som skal aksesseres ligger i hovedminnet. Delen som ligger i hovedminnet kalles *resident set*, så utførelsen kan fortsette hvis minnereferansene er til dette settet (avgjøres ved å se om sidetabell/segmenttabell gir at siden/segmentet er i hovedminnet). Når utførelsen krever en del som ligger i sekundærminnet vil prosessen blokkeres mens OS gjør en disk IO-request. En annen prosess kan utføres i mellomtiden. Når OS mottar minnet vil den få tilbake kontrollen og plassere prosessen i Ready-tilstand.

Det er to implikasjoner som fører til økt utnyttelse av systemet:

1. **Flere prosesser kan holdes i hovedminnet**. Siden kun noen biter av hver prosess ligger i hovedminnet, er det plass til flere prosesser, noe som gir bedre utnyttelse av prosessoren (mer sannsynlig at en prosess er kjørbar).
2. **En prosess kan være større enn hele hovedminnet**. Programmereren trenger ikke å strukturere programmet slik at det kan lastes inn i hovedminnet separat, fordi dette løses nå av OS og maskinvaren.

Minnet tilgjengelig for programmereren kalles virtuelt lager, og det er mye større enn det reelle minnet (dvs. hovedminnet), siden det benytter seg av sekundærminnet. Virtuelt lager tillater effektiv multiprogrammering og lar bruker slippe begrensningene hos hovedminnet.

Tabellen viser egenskapene hos paging og segmentering med og uten bruk av virtuelt lager

Enkel paging	Virtuelt lager paging	Enkel segmentering	Virtuelt lager segmentering
Hovedminnet deles inn i små rammer med fast størrelse		Hovedminnet blir ikke partisjonert	
Programmet deles inn i sider (*pages*) av kompilator eller lagerhåndteringssystem		Programmet deles inn i segmenter etter spesifisering som utvikler gir til kompilator	
Intern fragmentering innenfor rammer		Ingen intern fragmentering	
Ingen ekstern fragmentering		Ekstern fragmentering mellom segmenter	
OS må opprettholde en sidetabell for hver prosess som gir hvilken ramme hver side okkuperer		OS må opprettholde en segmenttabell for hver prosess som gir startadressen i hovedminnet og lengden til hvert segment	
OS må opprettholde en liste over ledige rammer		OS må opprettholde en liste over ledige hull i hovedminnet	
Prosessor bruker sidenummer og offset for å regne ut absolutt adresse		Prosessor bruker segmentnummer og offset for å regne ut absolutt adresse	
Alle sider hos en prosess må være i hovedminnet for at prosessen skal kunne kjøres (med mindre overlays brukes)	Prosessen kan kjøres selv om ikke alle sider er i hovedminnet. Sider kan leses inn etter behov	Alle segmenter hos en prosess må være i hovedminnet for at prosessen skal kunne kjøres (med mindre overlays brukes)	Prosessen kan kjøres selv om ikke alle segmenter er i hovedminnet. Segmenter kan leses inn etter behov
	Når en side leses inn i hovedminnet kan det kreve at en side skrives ut til disk		Når et segment leses inn i hovedminnet kan det kreve at et eller flere segmenter skrives ut til disk

Lokalitetsprinsippet

Virtuelt lager gjør at kun deler av prosessene kan ligge i hovedminnet av gangen, slik at det blir plass til flere prosesser. I tillegg spares det tid på å ikke bytte hele prosesser ut og inn av minnet ofte (inkludert ubrukte deler av prosessen). Når OS henter inn en ny del av en prosess, må en annen del byttes ut. Dersom den bytter ut en del rett før den skal brukes, må OS hente inn denne igjen umiddelbart. **Da vil det gå mer tid til å hente inn og kaste ut prosesser enn det brukes på å utføre instruksjoner, noe som kalles trashing**. For å forhindre dette vil OS bruke **lokalitetsprinsippet** for å beholde prosesser som med stor sannsynlighet vil brukes i nær fremtid. Lokalitetsprinsippet sier at program- og data-referanser innenfor en prosess har en tendens til å klynges sammen. Dette gjør at det er gyldig å anta at mindre biter av en prosess i hovedminnet av gangen vil gi bedre utførelse. Det brukes også for å ta gode avgjørelser for hvilke biter som trengs i fremtiden og bør derfor ikke byttes ut (= unngå trashing).

Merk: lokalitetsprinsippet gir at kode/data som ligger ved nære lokasjoner til nåværende kode/data, gjerne også aksesseres innen kort tid. Derfor hentes mer kode/data til primærlageret enn det som trengs nå.

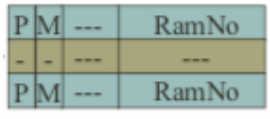
Merk: vi sier at virtuelt lager utnytter lokalitetsprinsippet

For at virtuelt lager skal være praktisk og effektivt, trengs det to ingredienser:

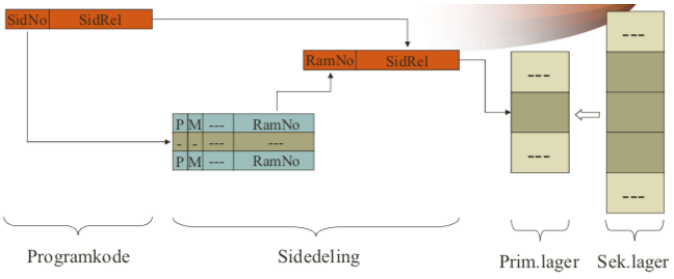
1. **Maskinvarestøtte for paging og/eller segmentering**
2. **OS må inkludere programvare for å håndtere bevegelsen av sider og/eller segmenter mellom primær- og sekundærminnet**.

Paging

Paging innebærer at hovedminnet blir delt inn i et antall rammer av lik størrelse, og hver prosess deles inn i sider som er like store som rammene. **Ved virtuelt lager kan prosessen utføres selv om ikke alle sidene er lastet inn i rammer i hovedminnet**. En sidetabell trengs også ved virtuelt lager, men innholdet er noe annerledes enn for simple paging, siden man må holde styr over blant annet hvilke sider av prosessen som ligger i hovedminnet. En bit P brukes for å indikere om en side ligger i hovedminnet, og hvis det er tilfellet (dvs. P = 1) vil enheten også gi rammenummer som oppgir hvor rammen ligger i hovedminnet. En annen bit M brukes for å indikere om innholdet i siden har blitt endret siden den sist var lastet inn i hovedminnet. Ved ingen endringer er det ikke nødvendig å skrive siden til disken når den flyttes ut av hovedminnet. Det kan også brukes andre kontrollbits.



Sidetabell struktur



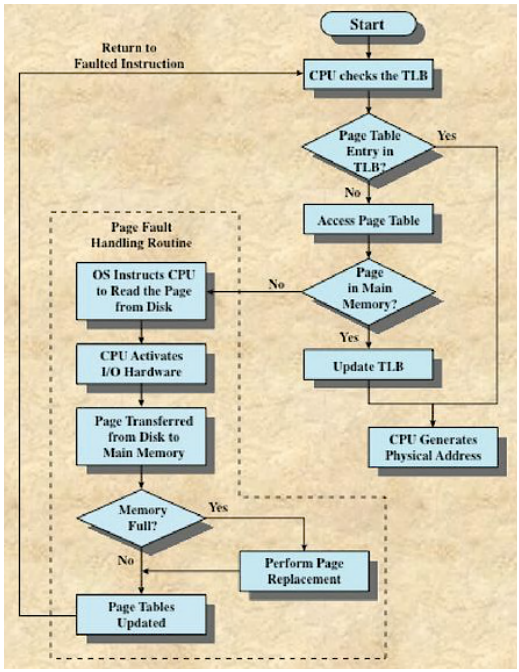
Sidetabellen brukes for å konvertere mellom en virtuell (logisk) adresse og en fysisk adresse. Når en bestemt prosess kjører vil et register inneholde startadressen til prosessens sidetabell (slik at den kan lokaliseres). Sidenummeret til en virtuell adresse brukes til å indeksere tabellen og hente korresponderende rammenummer. Dette kombineres med offset for å danne den fysiske adressen (se figur). De fleste sidetabellene lagres i det virtuelle lageret for å spare plass, slik at paging også gjøres på sidetabellen. Når en prosess kjører må deler av sidetabellen være i hovedminnet (kan bruke flere nivåer, s. 377). En ulempe med en- eller flernivå sidetabeller er at størrelsen er proporsjonal med det virtuelle adresserommet (tregt). En alternativ tabellstruktur er **invertert sidetabell**, der sidenummeret i den virtuell adressen kartlegges til en hashverdi ved å benytte en enkel hashfunksjon (rask). Hashverdien er en peker til den inverterte sidetabellen, der det er en entry per reell ramme istedenfor per virtuell side (s. 378).

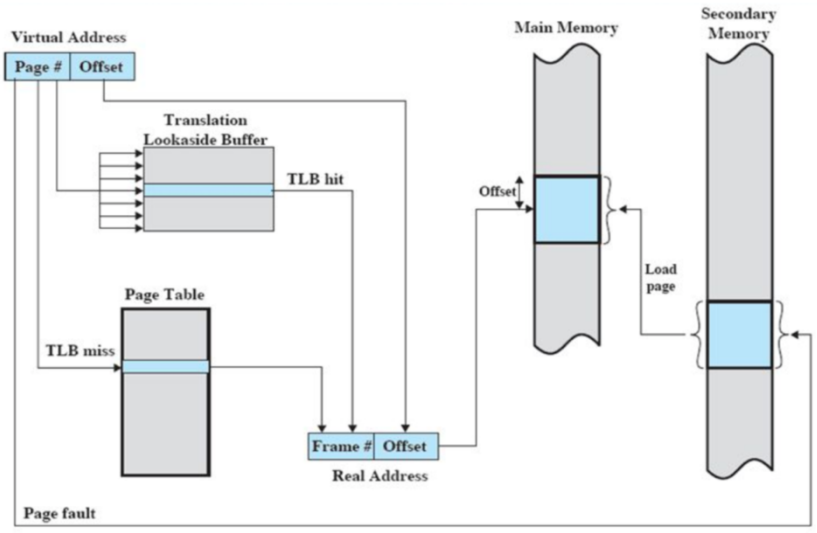
Merk: det finnes flere typer sidetabell-struktur: lineær søking, hashet og assosiativ.

TLB – Translation Lookaside Buffer

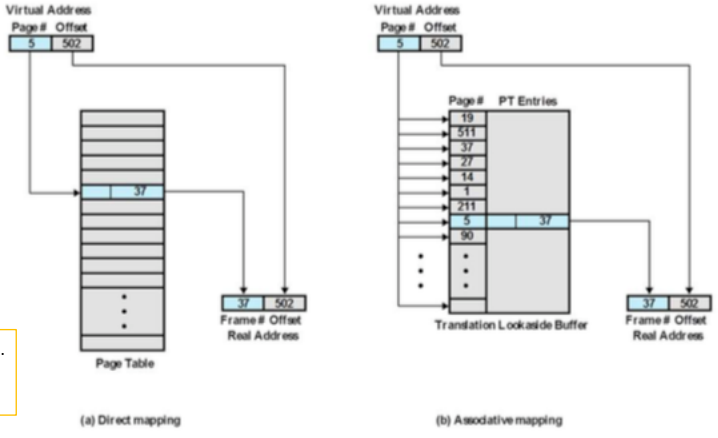
I prinsippet kan hver virtuelle minnereferanse gjøre to fysiske aksesser, en for å hente passende del av sidetabell og en for hente data. For at dette ikke skal doble minne-aksesstiden har de feste virtuelle lagerskjemaene en **TLB, som er en høyhastighets cache som inneholder sidetabelldeler som nylig har blitt brukt.** Lokalitetsprinsippet gir at de fleste virtuelle minnereferansene vil være til lokasjoner i nylig brukte sider (dvs. sider i TLB). Gitt en virtuell adresse vil prosessoren først sjekke om TLB inneholder matchende sidetabelldel. Hvis den er det (TLB hit), kan rammenummeret i delen kombineres med offset for å lage den fysiske adressen. Hvis den ikke er det (TLB miss), må delen hentes fra sidetabellen og prosessoren undersøker P-biten. Hvis P=1 vil siden være i hovedminnet og den fysiske adressen lages og sidetabelldelen legges til TLB. Hvis P=0 vil ikke siden være i hovedminnet (*page fault*) og prosessoren må kalle OS for å hente siden inn i hovedminnet og oppdatere sidetabellen.

Forelesning:
TC = TLB
ST-bom = page fault





Siden TLB kun inneholder noen enheter av den fulle sidetabellen kan den ikke indekseres på samme måte som den opprinnelige sidetabellen. TLB må derfor inneholde sidenummeret og hele sidetabelldelen. Assosiativ mapping brukes for å bestemme om TLB inneholder en match for gitte sidenummer (se figur).



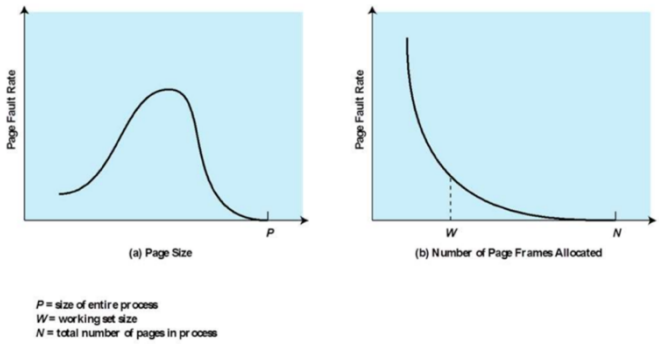
Merk: virtuelt lager må også interagere med cache-systemet som lagrer programkode/data. Hvis siden ikke er i hovedminnet vil fysisk adresse først sendes til cache for å se om den inneholder siden. Hvis ikke (SC-bom) må siden hentes inn i cache fra hovedminnet.

Merk: bruk av objektorienterte teknikker og multithreading gjør at lokaliteten reduseres, noe som gjør at TLB kan bli en flaskehals (s. 385).

Page-størrelse

En viktig maskinvare-beslutning er å velge størrelsen til sidene som må ta hensyn til faktorene:

1. **Intern fragmentering** – jo mindre sidestørrelse, desto mindre intern fragmentering og desto større utnyttelse av hovedminnet.
2. **Sidetabell størrelse** – jo mindre sidestørrelse, desto flere prosesser trengs per prosess, noe som betyr større sidetabeller. Dette kan kreve at deler av sidetabellen hos aktive prosesser er i virtuelt minne og ikke hovedminne, så man kan få dobbel *page fault* (1: hente inn sidetabelldel, 2: hente inn side)
3. **Sekundærminnets fysiske egenskaper** – jo større sidestørrelser, desto mer effektiv transportert av blokker med data
4. **Feilrate (*page fault rate*)** – denne effekten er illustrert på figur a og er basert på lokalitetsprinsippet. Liten sidestørrelse betyr at relativt mange sider er tilgjengelig i hovedminnet for en prosess. Etter en tid vil sidene i hovedminnet inneholde noen deler av nylig bruke referanser, og lokalitetsprinsippet gir dermed at sidefeilraten blir lav (venstre på figur). Ettersom størrelsen økes vil hver individuelle side inneholde lokasjoner som er lengre og lengre fra nylige referanser. Dette gjør at effekten til lokalitetsprinsippet svekkes og sidefeilraten økes. Til slutt vil sidefeilraten reduseres siden sidestørrelsen nærmer seg størrelsen til hele prosessen (P i figur). Når hele prosessen er i en side vil sidefeilraten være 0 (gir andre ulemper). Sidefeilraten vil også avhenge av antall rammer som er tildelt en prosess. For fast sidestørrelse vil sidefeilraten reduseres med økende antall rammer.



Figur a viser altså følgende effekt av sidestørrelsen:

- Hvis sidestørrelsen er stor nok vil en side romme hele prosessen/tråden, slik at sidefeil ikke oppstår. Ulempen er at hovedminnet vil ha plass til mindre prosesser/tråder
- Hvis sidestørrelsen er liten nok vil hovedminnet romme nok deler av gjeldende prosess/tråd, slik at sidefeil sjeldent oppstår. Ulempen er at hovedminnet ikke vil ha plass til så mange prosesser/tråder grunnet enorm overhead til blant annet sidetabeller.

Målet blir å finne en sidestørrelse mellom de to ytterpunktene, slik at resulterende sidefeilsfrekvens blir under en gitt grenset.

Segmentering

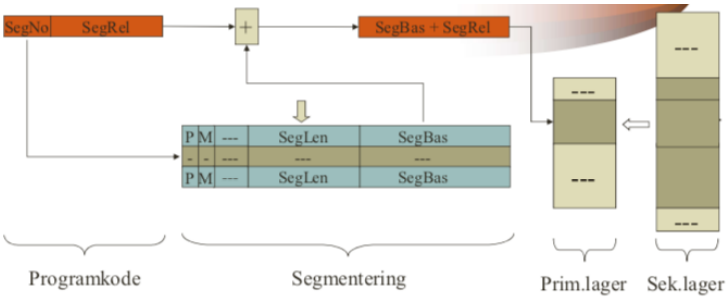
Segmentering lar programmereren se på minnet som bygd opp av flere adresserom eller segmenter som kan være av ulike og dynamiske størrelser. Minnereferanser består av segmentnummer og offset. Denne organiseringen har flere fordeler for programmereren:

1. **Håndterer voksende datastrukturer**. Segmentene er dynamiske, så programmereren trenger ikke å på forhånd vite hvilken størrelse datastrukturen vil få. OS vil justere størrelsen til segmentet etter behov. Hvis det er ledig plass i hovedminnet kan OS flytte segmentet ellers må det byttes ut og vente til det blir ledig plass.
2. **Støtter modularitet ved å tillate at programmer endres og rekompileres uavhengig**, uten at hele settet av programmer må omlinkes og omlastes. Dette oppnås ved å bruke flere segmenter.
3. **Tillater deling mellom prosesser**. Programmereren kan definere et segment som kan aksesseres av andre prosesser.
4. **Tillater beskyttelse** – hver segmenttabell inneholder en lengde som kan sammenlignes med offset for å hindre aksesser utenfor segmentet. Dette kan også oppnås i paging, men det er mer utfordrende. Deling oppnås ved at delt segment er i sidetabell hos flere prosesser.

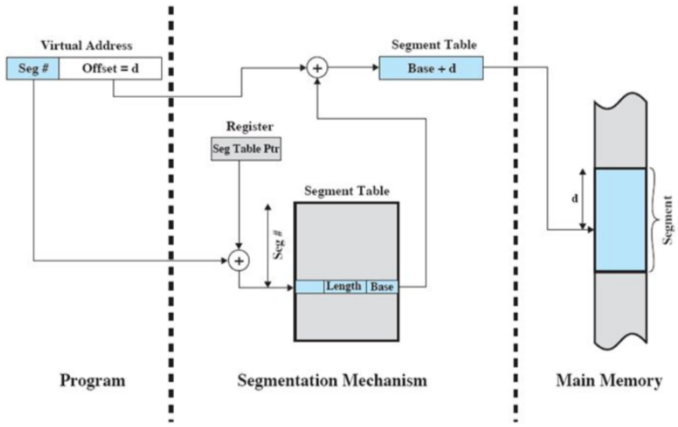
Segmenttabell struktur

Det er typisk å assosiere en segmenttabell med hver prosess. Ved virtuell lagring vil tabellen være noe mer kompleks, siden det må inkludere en P-bit brukes for å si om segmentet er i hovedminnet og en M-bit brukes for å si om innholdet i segmentet har blitt endret siden det siste ble lastet inn i hovedminnet (ingen endring = ingen behov for å skrive segmentet til disk når det byttes ut). Det kan også inkluderes andre kontrollbits, for eksempel for beskyttelse.

Mekanismen

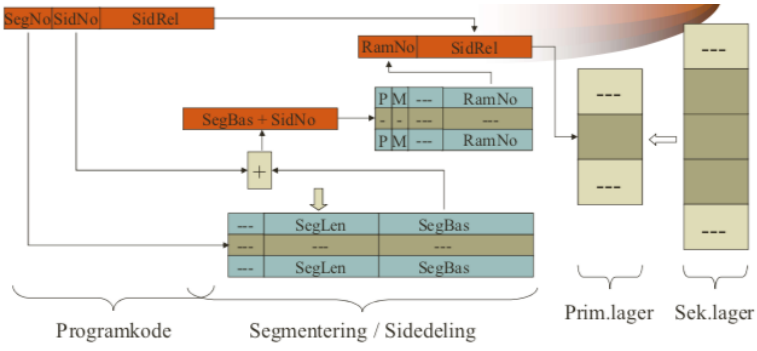


Mekanismen for å lese fra minnet involvere oversetting av en virtuell (logisk) adresse som består av segmentnummer og offset, til en fysisk adresse vha. segmenttabellen. Siden segmenttabellen har varierende lengde må den ligge i hovedminnet for å kunne aksesseres (ikke i register). Når en prosess utføres vil startadressen til segmenttabellen hos prosessen ligge i et register. Segmentnummeret i den virtuelle adressen brukes for å indeksere tabellen og dermed hente korresponderende startadresse hos segmentet i hovedminnet. Offset sammenlignes med lengden for å se at referansen er gyldig (dvs. innenfor segmentet). Hvis det er tilfellet blir adressen summert med offset i virtuell adresse, for å lage den ønskede reelle adressen (se figur).

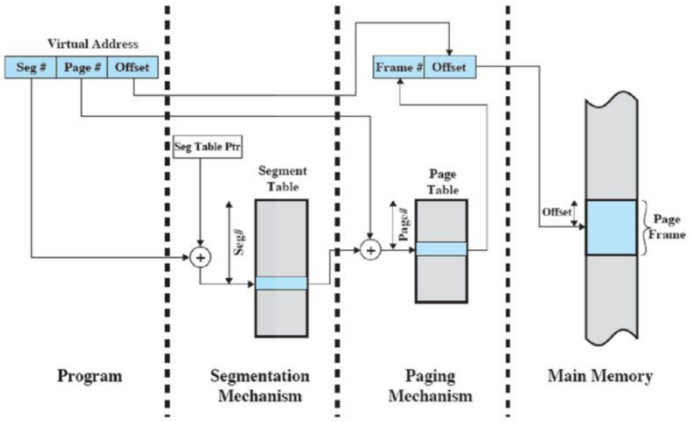


Kombinasjon av paging og segmentering

En kombinasjon av segmentering og paging gir fordeler (og ulemper) fra begge metodene. Kombinasjonen gir større overhead mht. plassbehov for tabeller og oppslagsbehov i tabellene, så den kan velges hvis man har råd til dette. Paging eliminerer ekstern fragmentering og gir dermed effektiv bruk av minnet. I tillegg vil bruken av faste og like størrelser tillate sofistikert lagerhåndtering som utnytter oppførselen til programmer. Segmentering er synlig for programmereren og har styrkene listet på forrige side (håndtering av voksende datastrukturer og støtte for modularitet, deling og beskyttelse).



I et kombinert system blir brukeradresserommet delt inn i et antall segmenter og hvert segment deles inn i flere sider. Hver side er like stor som en ramme i hovedminnet. Hvis segmentet er mindre enn en side, vil det okkupere kun en side. **Fra programmererens perspektiv vil den logiske adressen fortsatt bestå av segmentnummer og segment-offset, mens systemet ser segment-offset som et sidenummer og side-offset for siden i det spesifiserte segmentet.** Hver prosess har assosiert en segmenttabell og et antall sidetabeller (en per segment hos prosessen). Når prosessen kjører vil et register inneholde startadressen til segmenttabellen hos prosessen. Prosessoren bruker segmentnummer for å indeksere segmenttabellen og finne sidetabellen for dette segmentet. Videre blir sidenummeret i segment-offset brukt for å indeksere sidetabellen og finne korresponderende rammenummer. Dette kombineres med offset for å produsere ønsket reell adresse. På figuren kan vi se at segmenttabellen inneholder et segment-basefelt som gir sidetabellen hos sidene segmentet er plassert i. P- og M-bits brukes kun i sidetabellen.



Figuren til venstre viser også kombinasjonen av segmentering og paging. **Dette kombinerte systemet gir fokus på responstid og gjennomstrømming på samme tid.**

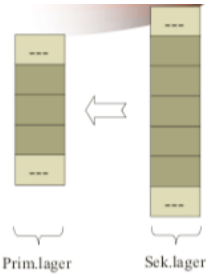
Merk: programvare komponenter som brukes ved lagerhåndtering er mekanismer for trygg og effektiv bruk av paging, segmentering eller en kombinasjon av disse, og støtte for trygg og effektiv innhenting, tilbakeføring, plassering og utbytting av kode og data.

8.2 Operativsystem programvare

Design av lagerhåndteringen i OS avhenger av (1) om man skal bruke virtuell lagring, (2) om man bruker segmentering, paging eller begge og (3) valg av lagerhåndteringsalgoritme. De to første valgene vil avhenge av tilgjengelig maskinvareplattform. De fleste OS bruker virtuelt lager og paging eller kombinasjon av paging og segmentering (rene segmenteringssystem er sjeldne, så vi fokuserer på paging). **Denne seksjoner ser på det tredje valget, der formålet er å minimere sidesidefeilraten** og dermed redusere overhead forbundet med avgjørelser om hvilken side som skal byttes ut og IO ved utbyttingen av sider. I tillegg må OS kjøre en annen prosess under IO, noe som gir kostbare prosesskifter. **Målet er altså å sørge for at når en prosess utføres vil det være lite sannsynlig at den refererer til en manglende side.**

Innhenting (*fetch policy*)

Fetch policy bestemmer når en page skal hentes til hovedminnet, og to vanlige alternativer er:



- **Demand paging** – en side hentes til hovedminnet ved behov, altså kun når det blir utført en referanse til denne siden. Ettersom flere sider hentes inn vil lokalitetsprinsippet gi at fremtidige referanser mest sannsynlig er til sidene som nylig ble hentet inn, slik at sidefeilraten blir svært liten.
- **Prepaging –** andre sider blir hentet i tillegg til den etterspurte. Dette utnytter at de fleste sekundærminnene har roterende ventetid, slik at det er mer effektivt å hente inn et antall sammenhengende sider av gangen istedenfor hver for seg. Det er ineffektivt dersom de fleste sidene som hentes inn ikke refereres. Metoden kan brukes når prosessen først starter (programmerer utpeker valgte sider) eller hver gang en sidefeil oppstår (siste er foretrukket, siden det er usynlig for programmerer).

Segmentering: Første, neste, beste, værste

Sidedeling / Kombinasjon: Uproblematisk

Plassering (*placement policy*)

Placement policy bestemmer hvor en prosessdel skal oppholdes i det reelle minnet. Dette er et viktig problem ved ren segmentering, der man kan bruke best-fit, first-fit, osv. For systemer med paging med eller uten kombinasjon av segmentering, vil plassering være irrelevant siden oversettelsesmekanismen er like effektiv for alle sideramme kombinasjoner.

Utbytting (*replacement policy*)

Replacement policy bestemmer hvilken side i hovedminne som skal byttes ut når en ny side hentes inn. Altså, når alle rammene i hovedminne er okkupert og en ny side må hentes inn for å tilfredsstille en sidefeil, så må det avgjøres hvilken side som skal erstattes. Det finnes flere metoder, og **målet ved alle er å ta ut siden som minst sannsynlig vil refereres til i nærmest fremtid. Metodene bruker lokalitetsprinsippet for å forsøke å forutsi fremtidig oppførsel basert på tidligere oppførsel**. Det vil være en balanse, for jo mer sofistikert utbyttingen er desto mer maskinvare og programvare overhead trengs for å implementere den. **Noen**

rammer er låst og sider som er lagret i disse kan ikke erstattes. Eksempler er deler av OS-kjernen, IO buffere, tidskritiske områder, osv. Dette oppnås med lock-bit i rammen. Vi ser på ulike grunnleggende algoritmer.

Optimal (kan ikke implementeres)

Optimal policy velger siden som har lengst tid til neste referanse, siden dette vil medføre færrest sidefeil. Det er umulig å implementere denne metoden, fordi det vil kreve at OS har perfekt kunnskap om fremtidige hendelser. Det fungerer heller som en standard som de andre metodene måles opp mot.

LRU (Least-Recently-Used)

LRU policy velger siden som ikke har blitt referert til på lengst tid. Lokalitetsprinsippet gir at det er minst sannsynlighet for at denne siden refereres til i nærmest fremtid. Denne metoden gjør det nesten like bra som optimal, men den er utfordrende å implementere. En løsning er å bruke tidsmerke for siste referanse hos hver side, noe som gir stort overhead. En annen løsning er å opprettholde en stack med sidereferanser, noe som også er kostbart.

LFU (Least-Frequently-Used)

LFU policy bytter ut den minst frekvente refererte siden. Denne metoden innebærer at man må holde en referanseteller for hver side hos en prosess, noe som gir stort overhead.

FIFO (First-in-First-out)

FIFO policy bytter ut siden som har vært lengst i minnet, ved at siderammer som er tildelt prosessen plasseres i en sirkulær buffer og sider fjernes med round-robin stil. Denne metoden er enklest å implementere og krever lite overhead, siden det kun trengs en sirkulær peker for hver prosess. Ofte vil det være programdeler eller data som brukes mye gjennom utførelsen av et program, og disse vil flyttes inn og ut av denne metoden.

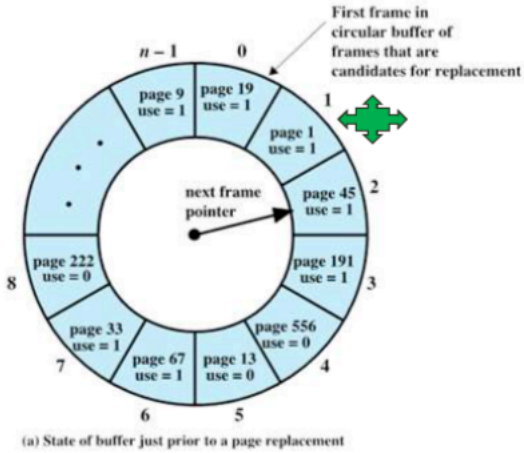
Clock

Klokkemetoder er resultatet av å forsøke å implementere LRU uten så mye overhead. Algoritmene kalles klokker fordi rammene kan visualiseres som en sirkel med en peker i midten. Vi ser på **to typer klokkealgoritme**:

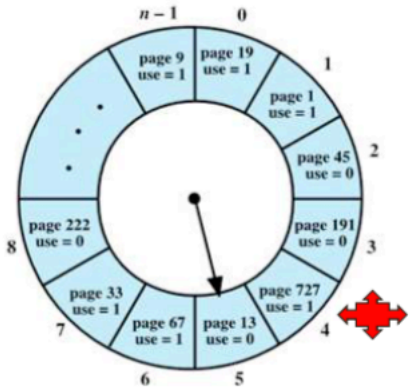
- **U-CLOCK**
- **UM-CLOCK**

U-Clock

Den enkleste formen for klokkealgoritme bruker en bit kalt *use bit* (U) assosiert til hver ramme. Når siden først lastes inn i rammen blir U satt lik 1. Hver gang siden refereres til i ettertid vil også U settes lik 1. En sirkulær buffer med peker holder på rammene som er kandidater for utbytting. Når en side blir erstattet vil pekeren indikere neste ramme i bufferen etter den som nettopp ble oppdatert. **Når en side skal erstattes vil OS skanne bufferen for å finne en ramme som har U=0**. **Hver gang den møter en ramme med U=1 vil den sette U=0 og fortsette. Første ramme OS møter med U=0 blir valgt for erstatning**. Hvis alle rammene har U=1 vil pekeren utføre en syklus der alle U settes til 0 og ender på den første rammen som velges for erstatning (dvs. den etter forrige erstattet ramme). Strategien ligner FIFO bortsett fra at rammer med U=1 passeres.



(a) State of buffer just prior to a page replacement



(b) State of buffer just after the next page replacement

UM-Clock

Klokkealgoritmen kan gjøres kraftigere ved å øke antall bits som brukes. Alle prosessorer som støtter paging bruker en modify-bit (M) assosiert til hver side og dermed også til hver ramme i hovedminnet. Denne biten brukes for å avgjøre om en side har blitt modifisert, slik at den ikke kan erstattes før den har blitt skrevet tilbake til sekundærminnet. Dette utnyttes av UM-CLOCK algoritmen, ved at hver ramme faller inn i en av fire kategorier:

- **Ikke nylig aksessert, ikke modifisert (U=0, M=0)**
- **Nylig aksessert, ikke modifisert (U=1, M=0)**
- **Ikke nylig aksessert, modifisert (U=0, M=1)**
- **Nylig aksessert, modifisert (U=1, M=1)**

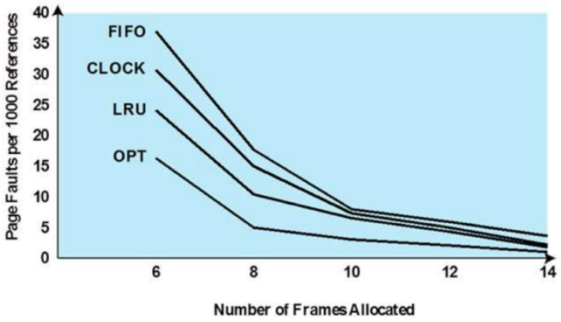
UM-CLOCK algoritmen bruker denne klassifiseringen på følgende måte:

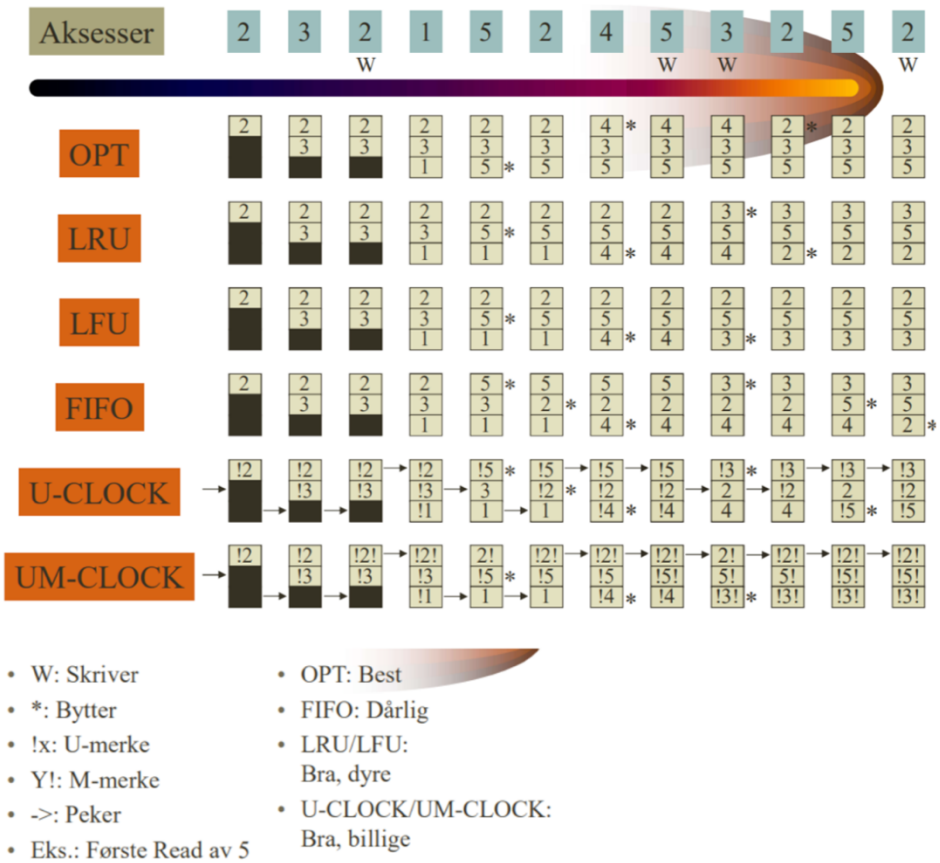
1. Begynn i nåværende pekerposisjon og skann rammebufferen. I løpet av skanningen vil ingen endringer utføres på U-biten. Den første rammen med U=0 og M=0 velges for utbytting.
2. Hvis steg 1 feiler, skann igjen og se etter rammen med U=0 og M=1. Den første rammen i denne kategorien velges for utbytting og U settes lik 0 for rammer som passeres
3. Hvis steg 2 feiler, skal pekeren har returnert til original posisjon og alle rammer skal ha U=0. Gjenta steg 1, og hvis nødvendig steg 2.

Fordelen med dette er at den vil prioritere å bytte ut umodifiserte sider, noe som vil spare tid siden disse ikke må skrives til disk. Dersom det ikke er noen umodifiserte sider vil den velge siden som ikke er aksessert nylig. Selv om man bruker tid på å skrive denne til disk, vil den ha større sannsynlighet for å ikke refereres til i nærmeste fremtid (pga. lokalitetsprinsippet).

Sammenligning av metoder

Figuren viser sidefeilraten for FIFO, U-CLOCK, LRU og optimal. Her kan vi se at U-CLOCK vil nesten ha like lav feilrate som LRU, samtidig som det gir lavere overhead. UM-CLOCK gir videre forbedring. Forskjellen mellom metodene er størst når få rammer er tildelt prosessen (s. 396).





Figuren viser oppførselen hos de ulike metodene for replacement policy. **U-CLOCK har brukerbar ytelse og mindre overhead, mens LRU/LFU har bedre ytelse med stor overhead (forbundet med plassbehov og utføringstid).** UM-CLOCK er krever mer plass og tid enn U-CLOCK, men gir bedre resultat (brukes om man har råd til ekstra overhead).

Sidebuffer er en strategi som kan forbedre ytelsen hos paging og samtidig tillate enklere replacement policyer (eks: FIFO). I dette tilfellet blir erstattet side i en ledig eller modifisert liste (s. 397). Størrelse hos cache er også viktig faktor (s. 398).

Mengde per prosess (*Resident Set Management*) – responstidsfokusert

Delen av prosessen som er i hovedminnet ved enhver tid kalles *resident set*, og *resident set management* handler om at OS må bestemme hvor mye av hovedminnet som skal allokeres til en bestemt prosess (dvs. hvor mange sider som skal hentes inn hos en prosess). Dette avhenger av flere faktorer, slik som:

- Jo mindre minne som tildeles en prosess, desto flere prosesser er det plass til og desto mindre blir behovet for swapping (større sannsynligheten for at en prosess er kjørbar)
- Jo mindre tildelte rammer, desto større er sidefeilraten
- Over en viss størrelse, vil ikke ekstra hovedminne kunne gi en merkbar endring i sidefeilraten pga. lokalitetsprinsippet

Det er to typer policyer som brukes i moderne OS:

1. **Gitt allokering** – prosessen får et fast antall rammer i hovedminnet som den kan utføres innenfor. Antallet bestemmes når prosessen lages og det kan være basert på type prosess (eks: interaktiv, batch, osv.), veiledning fra systemmanager, osv. Ved sidefeil under utføring av prosessen, må en av sidene til prosessen erstattes med den nye siden.
2. **Variabel allokering** – antall sider som er tildelt prosessen kan varieres i løpet av utføringen. Ideelt sett vil en prosess som har høy sidefeilrate (indikerer lav lokalitetsprinsipp) få flere rammer for å redusere mengde sidefeil (og motsatt). Denne metoden er ofte bedre, men det krever at OS vurderer oppførselen hos aktive prosesser (= større OS overhead og avhenger av maskinvaremekanismer hos prosessor).

Utbyttingsomfang

Omfanget til en utbyttingsstrategi kan kategoriseres som lokal eller global, og begge aktiveres av en sidefeil når det er ingen ledige rammer:

- **Lokal utbytting** – velger kun blant sidene hos prosessen som genererte sidefeilen når den skal bestemme hvilken side som skal erstattes. Enklere å analysere
- **Global utbytting** – velger blant alle ikke-låste sider i hovedminnet når den skal bestemme hvilken side som skal erstattes. Enklere å implementere og minimalt overhead (ingen bevis på at de yter dårligere enn lokal).

Tabellen viser sammenhengen mellom utbyttingsomfang og størrelsen til resident set.

	Lokal utbytting	**Global utbytting**
Gitt allokering	OK. Antall rammer som tildeles prosessen er fast og side som skal erstattes velges blant rammene som er tildelt prosessen. Ulempen er at liten allokering gir mye sidefeil og dermed tregt system, mens for stor allokering gir få programmer i hovedminnet og dermed mye ventetid eller mye swapping	Umulig. For at antall rammer skal være fast må ny side erstatte side hos samme prosess.
Variabel allokering	Avansert. Antall rammer som tildeles prosessen kan varieres for å opprettholde arbeidssettet til prosessen. Side som skal erstattes velges blant rammene som er tildelt prosessen. Den totale ytelsen til systemet kan økes ved å reevaluere allokeringen for så å øke eller senke den. Dette er avansert, men ofte bedre.	OK. Antall rammer som tildeles prosessen kan varieres for å opprettholde arbeidssettet til prosessen. Side som skal erstattes velges blant alle rammene i hovedminnet. Fordeler er at den er lett å implementere og en prosess som har mye sidefeil vil gradvis vokse i størrelse, slik at total mengde sidefeil i systemet reduseres. Ulempen er at det er utfordrende å velge side som skal erstattes for OS må også bestemme hvilken prosess som skal miste en side fra resident settet. Bruk av sidebuffer kan gjøre valget mindre signifikant (lettere å hente tilbake)

Vi ser nærmere på variabel allokering med lokal utbyttingsomfang.

Variabel allokering, lokal utbytting – arbeidssett (WS)

Ved variabel allokering vil antall rammer hos en prosess kunne variere og side som skal erstattes velges blant sidene hos prosessen. Fra tid til annen vil allokeringen hos en prosess reevalueres, og den kan økes eller minkes for å forbedre den totale ytelsen til systemet. Revurderingen er basert på en vurdering av sannsynlige fremtid etterspørsel av aktive prosesser. Det gjør metoden mer avansert, men gir også bedre ytelse. En metode for å revurdere allokeringen er basert på bruk av arbeidssett.

Arbeidssettet (WS: *Working Set*) hos en prosess ved virtuell tid t betegnes som $W(t, \Delta)$, og det er settet av sider hos prosessen som har blitt referert til i løpet av de Δ siste virtuelle tidsenhetene. Virtuell tid måles i minnereferanser, for eksempel for referansene $r(1), r(2), \ldots$ vil tiden være $t = 1, 2, \ldots$. Parameteren Δ er et vindu med virtuell tid der prosessen blir observert. Størrelsen til arbeidssettet er en ikke-synkende funksjon av Δ. Figuren viser en sekvens med sidereferanser for en prosess (dot er tidsenheter der arbeidssettet er uendret). **Merk at større Δ gir større eller lik WC**:

Sequence of Page References	Window Size, Δ			
	2	3	4	5
24	24	24	24	24
15	24 15	24 15	24 15	24 15
18	15 18	24 15 18	24 15 18	24 15 18
23	18 23	15 18 23	24 15 18 23	24 15 18 23
24	23 24	18 23 24	•	•
17	24 17	23 24 17	18 23 24 17	15 18 23 24 17
18	17 18	24 17 18	•	18 23 24 17
24	18 24	•	24 17 18	•
18	•	18 24	•	24 17 18
17	18 17	24 18 17	•	•
17	17	18 17	•	•
15	17 15	17 15	18 17 15	24 18 17 15
24	15 24	17 15 24	17 15 24	•
17	24 17	•	•	17 15 24
24	•	24 17	•	•

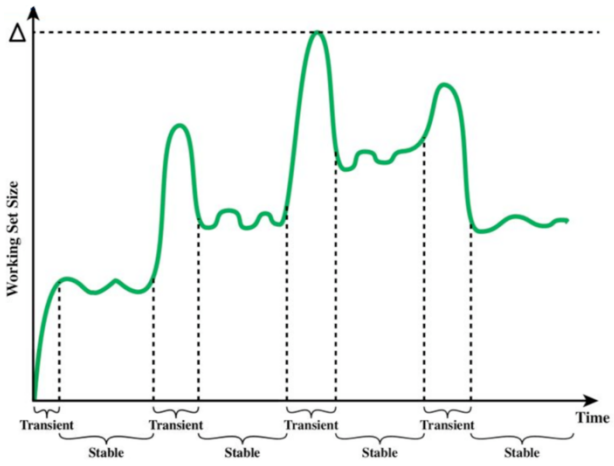
Merk: et større virtuelt tidsvindu betyr flere minnereferanser og dersom disse er til like sider vil arbeidssettet være like stort, mens dersom de er til nye sider vil arbeidssettet bli større.

$$WC(t, \Delta + 1) \supseteq W(t, \Delta)$$

Arbeidssettet er også en funksjon av tid. Hvis prosessen utføres over Δ tidsenheter og bruker kun en side, så vil $|W(t, \Delta)| = 1$. Arbeidssettet kan også bli like stort som antall sider N i prosessen dersom mange ulike sider adresseres hurtig og vindustørrelsen tillater det. Dermed har vi at:

$$1 \leq |W(t, \Delta) \leq \min(\Delta, N)$$

Figuren indikerer hvordan arbeidssettet kan variere over tid for en fast verdi for Δ. For mange programmer vil perioder med relativt stabile arbeidssett alternere med perioder med hurtige endringer. Størrelsen til arbeidssettet speiler omfanget av settet med sider som aksesseres til enhver tid. Når prosessen først begynner utføring vil den gradvis bygge opp et arbeidssett ettersom den refererer til nye sider. **Lokalitetsprinsippet bør gjøre at prosessen stabiliseres på et bestemt sett med sider og man får et relativt stabilt arbeidssett**. Et skifte i programmet fører deretter til en ny lokalitet og arbeidssettet endrer seg. I slike overgangssituasjoner vil noen av sidene fra den gamle lokaliteten være igjen i vinduet, slik at sidene som aksesseres i praksis tilhøre «to arbeidssett». Derfor vil størrelsen til arbeidssettet få en stor økning, før den etterhvert reduseres når vinduet glir forbi de gamle sidereferansene.

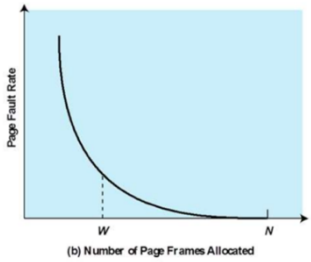


Arbeidssettet (WS) brukes for å bestemme størrelsen til resident set (RS) via følgende prosess:

1. **Følg med på arbeidssettet hos hver prosess for gitt Δ**
2. **Periodisk fjern sider fra RS hos prosess dersom de ikke er i WS** (essensielt LRU policy, siden det betyr at sider som ikke har blitt referert til i løpet av Δ blir fjernet)
3. **Kjør kun prosesser der RS omfatter WS** (dvs. WS er i hovedminnet)

Fordelen med metoden er at den utnytter lokalitetsprinsippet for å minimere sidefeil. **Problemene er at i praksis vil ikke fortid alltid forutsi fremtid (både størrelse og innhold i WS endres over tid), og det er vanskelig å utføre kontinuerlige målinger av WS og velge**

optimal verdi for Δ. Tanken bak er likevel gyldig og mange operativsystemer forsøker å tilnærme arbeidssett-strategien for å revurdere allokeringen av minne hos prosesser**.** For eksempel kan man bruke **sidefeilraten hos hver prosess istedenfor eksakte sidereferanser**, siden denne raten reduseres når RS øker (se figur). Størrelsen til arbeidssettet bør være i nærheten av W på figuren. Istedenfor å følge med på størrelsen til arbeidssettet direkte, kan vi oppnå lignende resultat ved å se på sidefeilraten:



1. Øk størrelsen på RS hvis det er for mange sidefeil (over maksimum threshold)
2. Mink størrelsen på RS hvis det er for få sidefeil (under minimum threshold)
3. Behold størrelsen på RS hvis det er passende antall sidefeil

Dette følges av algoritmen *Page Fault Frequency* (PFF, s. 404).

Cleaning policy

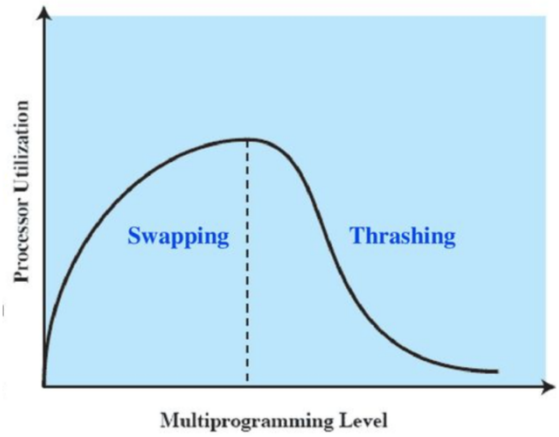
En cleaning policy bestemmer når en page skal skrives ut til sekundærminnet, og to vanlige alternativer er:

- **Demand cleaning** – en side skrives til minnet kun når den har blitt valg for erstatning
- **Precleaning** – modifiserte sider skrives til sekundærminnet før rammene deres trengs, slik at de kan skrives i bulker.

Det er ulemper ved begge (s. 450). En god løsning kan være å bruke sidebuffer med en modifisert og en ikke-modifisert liste.

Load control – gjennomstrømningsfokusert

Load control bestemmer antall prosesser som skal ligge i hovedminne, noe som kalles multiprogrammeringsnivå og er kritisk for prosessorytelsen. **Hvis det er for få prosesser vil det ofte være tilfellet at alle prosessene er blokkert og mye tid må brukes på swapping. Hvis det er for mange prosesser vil RS hos hver prosess ha utilstrekkelig størrelse noe som resulterer i mye sidefeil (= trashing)**. Trashing innebærer altså at OS bruker mer tid på å hente og kaste ut sider enn å utføre instruksjoner.



Det er flere tilnærminger for å løse dette problemet, for eksempel vil load control være implisitt inkorporert i arbeidssett-strategien. Det er kun prosesser med tilstrekkelig RS som får kjøre og dersom man gir et størrelseskrav på RS vil policyen automatisk og dynamisk avgjøre antall aktive program. **Man kan også forsøke å etterstrebe en balanse ved å overvåke sidefeilraten til hver enkelte prosess/tråd**. **Hver prosess/tråd må da ha så mye data i hovedminnet at sidefeilraten holdes under en viss grense**. Antall aktive prosesser/tråder justeres da til enhver tid ut i fra svingningene i sidefeilraten hos hver enkelt prosess/tråd. En annen tilnærming er å justere multiprogrammeringsnivået slik at gjennomsnittstiden mellom sidefeil er lik gjennomsnittstiden som trengs for å prosessere en sidefeil, noe som kan gi maksimum prosessorutnyttelse. En annen tilnærming innebærer å tilpasse klokkealgoritmen slik at skanning-raten holdes under en gitt threshold (s. 407).

8.4-8.6 Lagerhåndtering i operativsystem (s. 407-420)

Vi ser på et overblikk på lagerhåndtering for de ulike typene OS:

- **UNIX og Solaris** (s. 407-412) – brukerdata benytter paging og virtuelt lager, mens systemdata benytter forsinket, Buddy system. Sideutbytting er en U-CLOCK-variant.
- **LINUX** (s. 413-416) – brukerdata og systemdata benytter et Buddy system
- **Windows** (s. 417-419) – brukerdata og systemdata benytter paging og virtuelt lager
- **Android** (s. 419) – brukerdata og systemdata benytter et Buddy system

Del 5 – Tidsstyring av prosesser

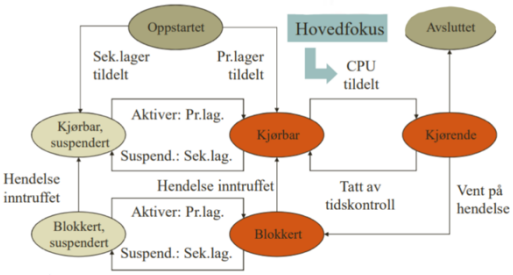
Denne delen av kompendiet ser på tidsstyring av prosesser og det inkluderer:

- **Kapittel 9** – Uniprosessor tidsstyring
- **Kapittel 10** – Multiprosessor, multikjerne og samtidstidsstyring

Kapittel 9 – Uniprosessor tidsstyring

I et multiprogrammeringssystem vil flere prosesser eksistere samtidig i hovedminnet. Hver prosess alternerer mellom å bruke prosessoren og vente på at en hendelse skal skje (eks: IO operasjon). Prosessoren holdes opptatt ved at en prosess utføres mens andre prosesser venter. **Nøkkelen til multiprogrammering er scheduling eller tidsstyring, som er prosessen av å tildele datamaskinens ressurser.** Når det er flere prosesser/tråder som ønsker adgang til felles ressurs, må man bruke tidsstyring for å velge hvilken rekkefølge prosessene/trådene skal få adgang til ressursen (eks: CPU). Valget av rekkefølge kan gjøres basert på ulike kvalitative og kvantitative kriterier, og det vil resultere i ulike effekter på forskjellige målparametere. Det er operativsystemet som vil gjennomføre tidsstyringen i henhold til valgt algoritme. **I et multiprogrammeringssystem er det som regel fire typer tidsstyring:**

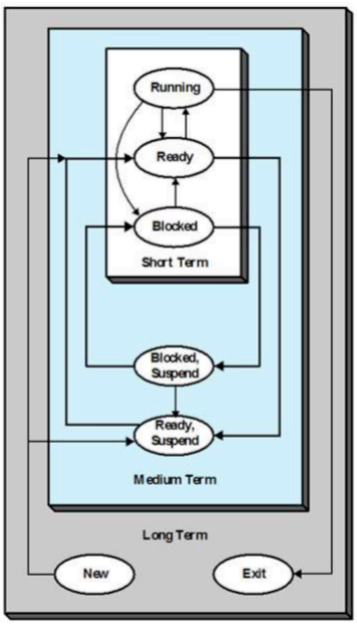
1. **Long-term scheduling** – utføres når en ny prosess lages og er beslutningen om å legge til en ny prosess i settet av aktive prosesser
2. **Medium-term scheduling** – utføres ved swapping og er beslutningen om å legge til en prosess i settet av prosesser som er minst delvis i hovedminnet og dermed kjørbar.
3. **Short-term scheduling** – beslutningen om hvilken kjørbar prosess som skal utføres neste gang av prosessoren
4. **I/O-scheduling** – beslutningen om hvilken prosess sin IO-request som skal håndteres av en tilgjengelig IO-enhet (kapittel 11).

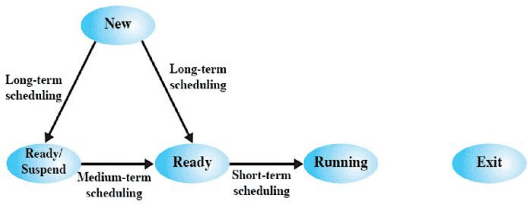


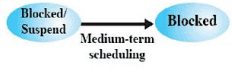
Hovedfokuset i dette kapitlet er på hvordan OS bestemmer hvilken prosess som skal flyttes fra Kjørbar til Kjørende tilstand (dvs. velger neste prosess som skal utføres av prosessor).

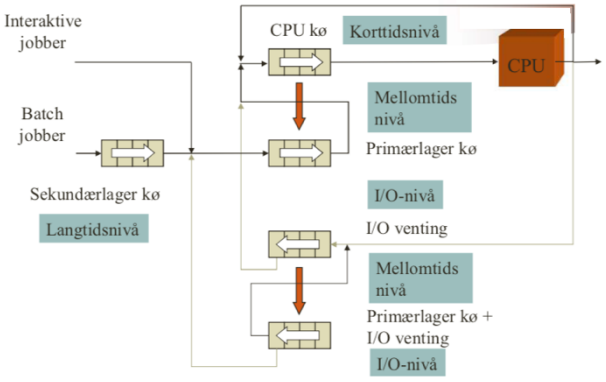
9.1 Typer prosessortidsstyring

Hensikten med prosessortidsstyring er å tildele prosesser for utføring av prosessoren(e) over tid på en måte som tilfredsstiller systemets mål, slik som responstid, gjennomstrømming og prosessoreffektivitet. Dette deles ofte inn i long-, medium- og short-time scheduling. Figuren viser prosesstilstand-diagram med disse tidsstyringsfunksjonene. Tidsstyring vil påvirke ytelsen til systemet fordi det bestemmer hvilke prosesser som vil vente og hvilke som får fortsette. **Det handler i grunnlag om å håndtere køer for å minimere køforsinkelse og optimere ytelse (se figur under).**









Long-term tidsstyring

Long-term tidsstyring bestemmer hvilke programmer som skal tillates i systemet for å senere prosesseres. **Det kontrollerer derfor graden av multiprogrammering.** Når en jobb eller et brukerprogram er godkjent vil omgjøres til en prosess og legges i kø for short-term tidsstyreren. I noen system kan en prosess begynne i swapped-out tilstand, og i dette tilfellet vil prosessen legges til i køen til medium-term tidsstyreren. I et batch-system vil nye jobber legges til en disk og holdes i en batch-kø. Long-term tidsstyrer lager prosesser fra køen og det er to beslutninger som er involvert:

1. **Når kan OS ta inn en ekstra prosess?** Dette bestemmes i hovedsak av ønsket multiprogrammeringsgrad. Jo flere prosessorer som opprettes, desto mindre prosessortid per prosess (flere konkurrer om prosessor). Long-term tidsstyrer kan derfor begrense antall prosesser for å gi bedre tjeneste til nåværende sett med prosesser.
2. **Hvilke jobber skal aksepteres og gjøres om til prosesser?** Dette kan bestemmes på flere måter, for eksempel basert på *first-come-first-served*, prioritetsnivå, forventet utføringstid, IO-krav, osv.

Medium-term tidsstyring

Medium-term tidsstyring er en del av swapping-funksjonen og er diskutert i kapittel 3, 7 og 8. Swapping-beslutningen er som regel basert på graden av multiprogrammering. For system som ikke bruker virtuelt lager vil lagerhåndteringen være utfordrende, så swapping-beslutningen må også ta hensyn til lagerkravene hos suspenderte prosesser.

Medium-term tidsstyring

Short-term tidsstyreren, også kalt dispatcher, utføres oftest og bestemmer hvilken prosess som skal kjøres neste (finkornet beslutning). Dispatcheren involveres hver gang en hendelse oppstår som kan lede til at en nåværende prosess blir blokkert eller at en prosess kan utføres istedenfor nåværende. Eksempler på slike hendelser er klokkeavbrudd, IO-avbrudd, OS systemkall eller signaler (eks: semaforer).

9.2 Tidsstyringsalgoritmer

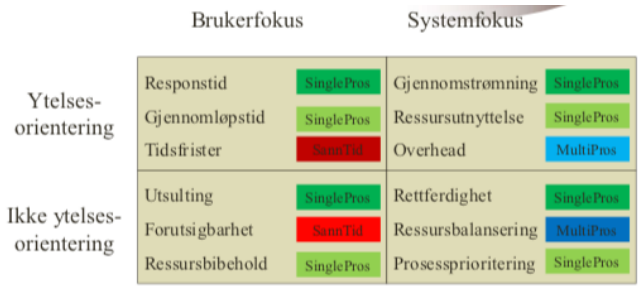
Kriterier for short-time tidsstyring

Målet med short-term tidsstyring er at prosessortiden skal fordeles slik at en eller flere aspekter ved systemoppførselen optimaliseres. Kriteriene som tidsstyringen evalueres opp mot kan kategoriseres i to dimensjoner:

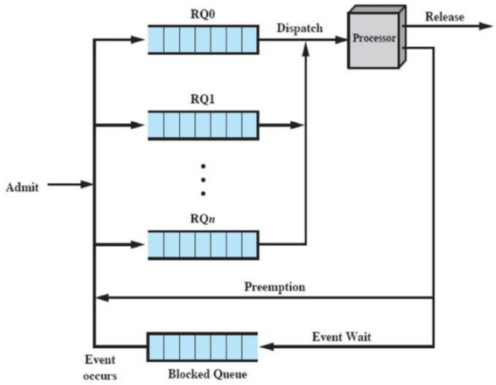
1. **Bruker- eller systemfokusert** – brukerfokus relaterer til systemoppførselen sett fra en individuell bruker eller prosess (viktig for alle system), mens systemfokus fokuserer på effektiv utnyttelse av prosessoren (mindre viktig for enkelt-bruker system)
2. **Ytelse- eller ikke-ytelsesrelatert** – ytelsesrealterte kriterier er kvantitative og kan lett måles, mens ikke-ytelsesrelaterte kriterier er vanskeligere å mål og analysere.

Tabellen viser viktige kriterier som er gjensidig avhengige, så det er umulig å optimalisere alle samtidig. Eks: lav responstid krever ofte prosesskifting noe som gir lavere gjennomstrømming.

	Brukerfokus	Systemfokus
Ytelsesrelatert	• **Turnaround time (gjennomløpstid)** – tiden mellom innsending av prosess og fullføring. Minimeres • **Responstid** – tiden mellom innsending av request og mottakelse av respons. Minimeres • **Deadlines** – antall tidsfrister for gjennomførelse av prosesser som møtes. Maksimeres	• **Gjennomstrømming** – antall prosesser som fullføres per tidsenhet (dvs. arbeidsmengde). Maksimeres • **Prosessorutnyttelse** – andel tid prosessoren er opptatt. Maksimeres (viktig for delt system)
Ikke-ytelsesrelatert	• **Forutsigbarhet** – jobb utføres med samme tid og kostnad uavhengig av systemets last. Stor variasjon i respons- og gjennomløpstid er forvirrende for bruker.	• **Rettferdighet** – ved fravær av brukerveiledning skal alle prosesser behandles likt og utsulting skal unngås. • **Prioritet** – tidsstyrer følger prosessprioriteter • **Ressursbalanse** – alle ressurser bør være i bruk



Design av tidsstyrer innebærer kompromiss blant konkurrerende krav og riktig balanse avhenger av egenskapene og tenkt bruk av systemet. For brukeren vil responstid ofte være viktigst, mens for systemet vil gjennomstrømming og prosessorutnyttelse ofte være viktigst.



Bruk av prioritet

Ved bruk av prosessprioritet vil tidsstyreren alltid velge en høyere prioritet over en lavere, og det eksisterer en kø for hver prioritetsgruppe. Når tidsstyreren skal velge neste prosess som skal utføres, vil den sjekke køen med høyest prioritet først og deretter resten i synkende rekkefølge. Utfordringen er at prosesser med lavere prioritet kan utsettes for utsulting, noe som kan løses ved å for eksempel endre prioriteten med prosessalderen.

Tidsstyringsalgoritmer

Figuren under viser en oversikt over de ulike tidsstyringsalgoritmene.

	FCFS	Round robin	SPN	SRT	HRRN	Feedback
Selection function	max[w]	constant	min[s]	min[s − e]	$\max\left(\frac{w+s}{s}\right)$	(see text)
Decision mode	Non-preemptive	Preemptive (at time quantum)	Non-preemptive	Preemptive (at arrival)	Non-preemptive	Preemptive (at time quantum)
Throughput	Not emphasized	May be low if quantum is too small	High	High	High	Not emphasized
Response time	May be high, especially if there is a large variance in process execution times	Provides good response time for short processes	Provides good response time for short processes	Provides good response time	Provides good response time	Not emphasized
Overhead	Minimum	Minimum	Can be high	Can be high	Can be high	Can be high
Effect on processes	Penalizes short processes; penalizes I/O bound processes	Fair treatment	Penalizes long processes	Penalizes long processes	Good balance	May favor I/O bound processes
Starvation	No	No	Possible	Possible	No	Possible

Seleksjonsfunksjonen bestemmer hvilken prosess som skal velges for utførelse blant prosessene som er kjørbare. Den kan være basert på prioritet, ressurskrav eller utføringsegenskaper hos prosessen. For utføringsegenskaper ser man på:

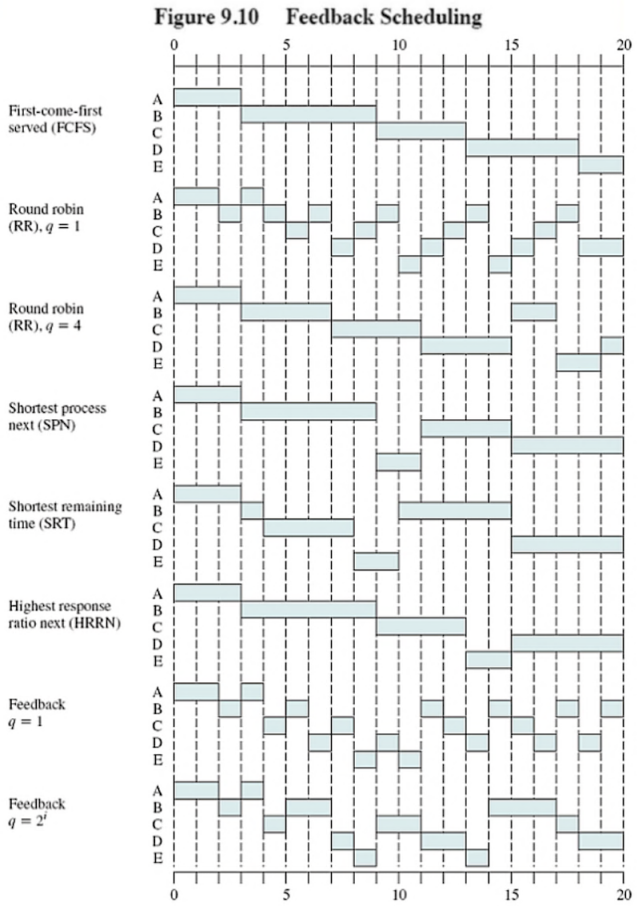
- ***w* – tid brukt i systemet så langt (ventende)**
- ***e* – tid brukt under utførelse så langt**
- ***s* – total tjenestetid prosessen krever (inkluderer *e*). Dette estimeres eller gis av bruker**

Beslutningsmodus spesifiserer ved hvilken tid seleksjonsfunksjonen utføres, og det er to vanlige varianter:

- **Ikke-avbrytbar (*nonpreemptive*)** – prosessen i Running-tilstand vil fortsette å kjøre til den terminerer eller blokkerer seg selv for IO eller forespørsel om OS-tjeneste

- **Avbrytbar (*preemptive*)** – kjørende prosess kan avbrytes og flyttes til Ready-tilstand av OS. Dette kan gjøres når ny prosess ankommer, ved avbrudd eller klokkeavbrudd. Det krever mer overhead, men kan gi bedre tjeneste for total mengde prosesser, siden det hindrer at en prosess får monopol på prosessoren. Kostnaden kan reduseres med effektiv prosesskifte og stort hovedminne med plass til mange prosesser.

Prosess	Ankomsttid	Tjenestetid
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2



Gitt prosessene på tabellen over, vil de ulike tidsstyringsalgoritmene ha utføringsmønster gitt på figuren til venstre. Ved å bruke fullføringstiden kan man bestemme turnaround time (tid mellom innsending av prosess og fullføring). For kømodellen vil **turnaround time (TAT) være oppholdstiden T_r (r for resident) som et element bruker i systemet (ventetid + tjenestetid).** Et mer nyttig mål er normalisert turnaround time som er turnaround time over tjenestetid (dvs. T_r/T_s). **Normalisert turnaround time vil indikere relativ forsinkelse som prosessen opplever**. Jo lengre prosessutføringstiden er, desto større forsinkelse kan ofte tolereres. Minimum verdi er 1.0 og økende verdi betyr redusert tjenestenivå.

First-come-first-served (FCFS) (s. 435)

Merk: denne delen vil fokusere på å få et overblikk over de ulike metodene. For detaljer, se i boka.

FCFS har ikke avbrudd eller prioritering. Når en prosess blir kjørbar blir den plassert i Kjørbar-køen og neste prosess som velges er den som har vært i køen lengst (dvs. har $\max(w)$, kalles også FIFO). Aktive prosesser kjøres til de er fullført eller blokkeres av forespørsel på IO eller annen OS-tjeneste. FCFS yter bedre for lange prosesser enn for korte (korte prosesser straffes hvis de ankommer etter lang prosess), og den vil ofte gjøre det bedre for prosessorbundet prosesser enn for IO-bundet (dvs. prosesser som bruker prosessor mer enn IO). FCFS kan gi ineffektiv bruk av prosessor og IO-enheter. **Det er som regel ikke bra alene i et uniprosessorsystem, men dersom det kombineres med prioritering kan det bli nyttig (en kø for hver prioritet og FCFS på hver av køene). FCFS brukes i satsvise (dvs. ingen interaktiv tilgang til program for brukere), der høy gjennomstrømming ikke oppfattes som viktig (E).**

Merk: satsvis = batch-system

Round Robin (RR) (s. 437)

Round Robin fordeler tiden på de ulike prosessene i periodiske intervaller. Et klokkeavbrudd blir generert i periodiske intervaller og ved et avbrudd vil kjørende prosess plasseres i Ready-køen og neste prosess velges på FCFS basis (dvs. $\max(w)$ per runde). Det kalles også time slicing, siden hver prosess får en tidsspalte før de avbrytes. Utfordringen med RR er å velge lengden til tidsspalten. Håndtering av klokkeavbrudd og utføring av tidsstyring og dispatcher vil også innebære noe overhead. Derfor bør man unngå svært korte tidsspalter. Som regel vil spalten settes lik tiden det tar for en typisk interaksjon eller prosessfunksjon, slik at de fleste prosessene krever én spalte. Hvis spalten er like lang som lengste prosess vil det tilsvare FCFS. En ulempe med RR er at det kan gi ekstra avbrudd som er unødvendige (øker overhead).

En ulempe med RR er at den behandler prosessorbundet og IO-bundet prosesser likt, selv om IO-bundet generelt er kortere. Dette gjør at prosessorbundet prosesser får en urettferdig del av prosessortiden, noe som resulterer i dårlig ytelse for IO-bundet prosesser og økt variasjon i responstiden.. Dette kan løses ved bruk av Virtuell Round Robin, som har egen kø for IO-prosesser som prioriteres over prosessorbundet prosesser (s. 439). **Round Robin brukes i systemer med interaktiv tilgang for brukere til programmene som kjøres (E).**

Merk: prosesseringstid og tjenestetid er det samme. s er et estimat på forventet verdi for denne tiden.

Shortest Process Next (SPN) (s. 440)

SPN har ikke avbrudd, men prioriterer prosessen som har kortest forventet total prosesseringstid, slik at denne velges for utføring først (dvs. $\mathbf{min}(s)$). Korte prosesser flyttes til staren av køen forbi lengre prosesser. Dette gir en økning i total ytelse mht. responstid. Det vil derimot øke variabiliteten i responstiden, spesielt for lengre prosesser, noe som gjør at forutsigbarheten reduseres. Dessuten kan det være vanskelig å vite prosesseringstiden for hver prosess. Et estimat kan gis av programmerer eller i systemer der samme jobb utføres ofte kan det samles statistikk for å regne ut gjennomsnitt eller eksponentielt gjennomsnitt (favoriserer nylige prosesser = ofte mer korrekt, s. 440). En utfordring ved SPN er at det kan føre til utsulting av lengre prosesser, dersom det er mange korte prosesser. Manglende avbrudd gjør det også lite nyttig for time-sharing- eller transaksjonsprosesseringssystem. **SPN brukes i satsvise system (dvs. uten interaktiv tilgang for brukere til program), der det er ønsket en viss gjennomstrømming via prioritering av små og store prosesser (E).**

Shortest Remaining Time (SRT) (s. 441)

SRT er en avbruddsversjon av SPN, der tidsstyrer alltid velger å utføre prosessen med minst forventet *gjenværende* prosesseringstid (dvs. $\mathbf{min}(s - e)$). Hvis en ny prosess ankommer køen og har kortere prosesseringstid enn kjørende prosess, kan tidsstyrer avbryte kjørende prosess og begynne utføring av den nye prosessen. I likhet med SPN må SRT ha et estimat på prosesseringstid (krever opptak av forløpt tjenestetid = større overhead), og det er risiko for utsulting av lange prosesser. SRT vil ikke favorisere lange prosesser slik som FCFS og vil ikke generere ekstra avbrudd slik som RR (reduserer overhead). SRT gir bedre turnaround time enn SPN, siden kortere jobber får umiddelbar prioritet. **SRT brukes i satsvise system (dvs. uten interaktiv tilgang for brukere til program), der det er ønsket en viss gjennomstrømming via prioritering av små prosesser (E).**

Highest Response Ratio Next (HRRN) (s. 443)

Vi ønsker å minimere T_r/T_s for hver individuell prosess og oppnå minste gjennomsnittsverdi over alle prosesser. Generelt kan vi ikke vite tjenestetiden T_s på forhånd, men den kan tilnærmes basert på tidligere historie eller input fra bruker eller konfigurasjonsmanager. Responsratioen er gitt av:

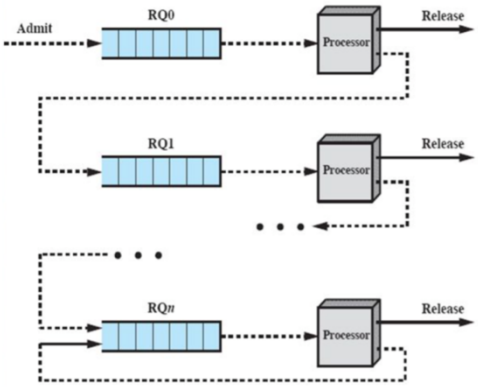
$$R = \frac{w + s}{s}$$

Der w er ventetiden og s er forventet tjenestetid. **HRRN har ikke avbrudd, og etter nåværende prosess er fullført eller blokkert vil den velge neste prosess som har høyest responsratio (dvs $\mathbf{max}(\mathbf{R})$).** Fordelen med HRRN er at det tar hensyn til alderen til prosessen. Selv om korte prosesser favoriseres (mindre s), vil lengre prosesser tilslutt få større R fordi de har ventet tilstrekkelig lenge (større w). Dermed hindres utsulting. Ulempe er at forventet tjenestetid må estimeres (likhet med SRT og SPN).

OBS: HRRN, SRT og SPN kan ikke brukes hvis vi ikke har noen indikasjon på relativ lengde hos prosessene. For å favorisere korte prosesser kan FB brukes

Feedback (FB) (s. 443)

En annen måte å favorisere kortere jobber er å straffe jobber som har kjørt lenge, noe FB oppnår ved å se på utføringstid så langt hos prosesser. Tidsstyringen innebærer tidsavbrudd og en dynamisk prioriteringsmekanisme. Når en ny prosess entrer systemet blir den lagt til i



RQ0 (se figur). Etter første avbrudd når prosessen plassers i Read-tilstand igjen, vil den legges til RQ1. For hvert avbrudd blir prosessen lagt til en kø av lavere prioritet. En kort prosess vil fullføres raskt uten å flyttes langt ned i hierarkiet av Ready-køer, mens en lang prosess vil gradvis flyttes nedover (= korte prosesser favoriseres). Alle køene bortsett fra den med lavest prioritet, bruker en FCFS-mekanisme (lavest kø bruker RR, siden den returneres til samme kø ved avbrudd, se figur). Metoden kalles multilevel feedback, siden OS tildeler prosessoren til en prosess og når prosessen blokkeres blir den «matet» tilbake til en av flere prioritetskøer. En ulempe er at FB kan føre til utsulting av lange prosesser dersom mange kortere prosesser ofte entrer systemet. Dette kan kompenseres ved å variere tidsavbruddet for hver kø, for eksempel kan RQ0 ha tidsavbrudd etter 1 tidsenhet, RQ1 etter 2 enheter og RQi etter 2^i tidsenheter. Utsulting kan fortsatt oppstå for lange prosesser, så en løsning kan være å flytte prosesser til en kø med høyere prioritet etter en gitt ventetid i nåværende kø.

Sammenligning av de ulike algoritmene

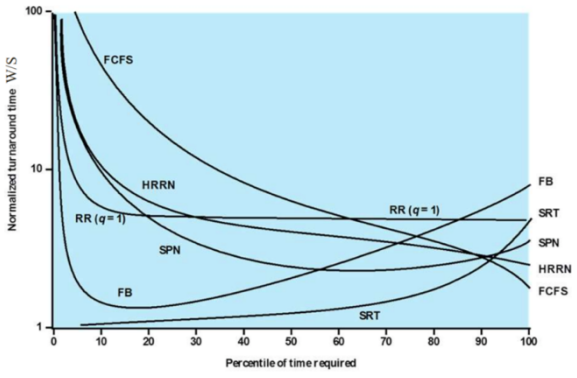
Sammenligningen av ytelsen til de ulike tidsstyringsalgoritmene er generell, fordi det er umulig å utføre en definitiv sammenligning siden den relative ytelsen avhenger av flere faktorer (dvs. kan være ulik i ulike tilfeller).

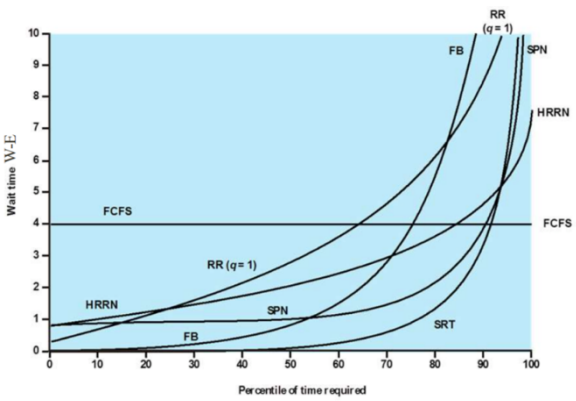
Køanalyse

For en prioritetsbasert tidsstyrer der prioriteten gis uavhengig av forventet tjenestetid, vil gjennomsnittlig turnaround time være tilnærmet lik en enkel FCFS. Tilstedeværelsen av avbrudd gjør ingen forskjell i disse gjennomsnittene. For en prioritetsbasert tidsstyrer der prioriteten baseres på tjenestetiden vil forskjellen være større. **Ved å prioritere kortere jobber vil gjennomsnittlig turnaround tid reduseres ved høyere grader med utnyttelse av prosessoren**. Forbedringen er størst ved bruk av avbrudd, men overordnet ytelse vil ikke påvirkes mye av dette. Dersom vi skiller mellom korte og lange prosesser med hhv. høy og lav prioritet vil forskjellen bli tydeligere. For korte prosesser vil det være en signifikant forbedring som kan videre forsterkes ved å introdusere avbrudd. For lange prosesser vil prioritering gi lavere ytelse enn ikke-prioritering (dvs. favorisering av korte prosesser gjør at lange prosesser får lengre responstid).

Simuleringsmodell

Venstre figur viser normalisert turnaround time, mens høyre figur viser gjennomsnittlig ventetid. For turnaround time ser vi at FCFS gjør det dårligst, siden en tredjedel av prosessene har en normalisert turnaround time som er 10 ganger større enn tjenestetiden og dette er de korteste prosessene. FCFS har ventetid som er uniformt fordelt på alle prosessene, noe som skyldes at tidsstyringen er uavhengig av tjenestetiden. RR med tidsspalte på 1 tidsenhet, har en normalisert turnaround time på 5 for alle prosesser, slik at de behandles rettferdig (kortere prosesser utføres innenfor en tidsspalte). SPN vil yte bedre enn RR bortsett fra for de korteste prosessene. SRT vil yte bedre enn SPN bortsett fra for de 7% lengste prosessene. For algoritmer uten avbrudd vil FCFS favorisere lange prosesser, mens SPN favoriserer korte. HRRN er et kompromiss mellom disse, noe som vises på figurene. FB yter godt for korte prosesser. x



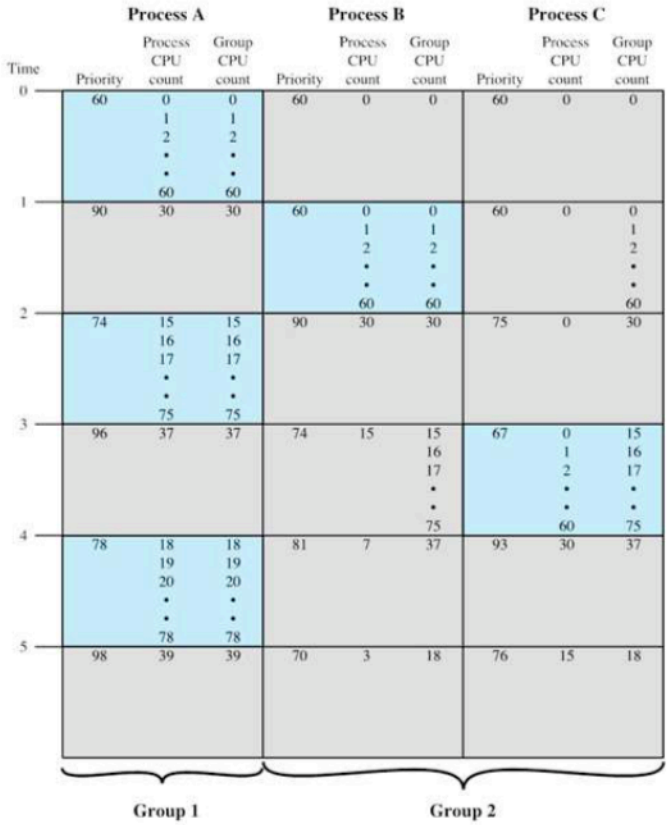


Bruk av avbrudd og prioritering (E)

Prosessavbrudd gjør at lange prosesser kan fratas prosessoren før de er ferdige. Dermed kan korte prosesser få raskere tilgang til ressursene de trenger. Fordelen ved bruk av avbrytning er at det kan sikre bedre responstid og/eller gjennomstrømming, mens ulempen er at det kan gi større overhead og utsulting. Fordelen med prioritering er at det kan sikre bedre responstid og/eller gjennomstrømning, mens ulempen er at det kan gi større urettferdighet og utsulting.

Fair-Share tidsstyring (FSS) – gruppebetraktning

I et multiuser system der individuelle brukerapplikasjoner eller jobber kan organiseres i flere prosesser (eller tråder), vil det være en struktur til samlingen av prosesser som ikke utnyttes av tradisjonelle tidsstyrere. Fra brukerens perspektiv er det ikke viktig hvordan en bestemt prosess yter, men hvordan hans/hennes sett med prosesser yter (settet utgjør en applikasjon). **Fair-share tidsstyring** brukes for å gjøre beslutninger innenfor tidsstyring basert på disse prosess-settene. **Hver bruker blir tildelt en vekt som definerer brukerens andel av ressursene (prosessoren) sammenlignet med total bruk av disse ressursene.** Hvis bruker A tildeles dobbelt så mye vekt som bruker B, så bør bruker A i det lange løp kunne utføre dobbelt så mye arbeid som bruker B. Det har blitt utviklet flere fair-share tidsstyrere, for eksempel fair-share scheduler (FSS) som brukes i flere UNIX systemer. FSS vil dele brukerne inn i et sett med fair-share grupper og tildeler fraksjoner av prosessorressurser til hver gruppe. For eksempel kan det være fire grupper som hver har 25% av prosessorbruk.



Tidsstyringen vil gjøres på grunnlag av prioritet, som avhenger av prosessprioritet, nylig prosessorbruk og gjenværende prosessorbruk hos gruppen som prosessen hører til. Prioriteten til prosessen vil dermed synke ettersom prosessen eller andre prosesser hos gruppen bruker prosessoren (s. 451).

9.3 Tradisjonell UNIX tidsstyring (s. 452-453)

Tidsstyringsalgoritmen for tradisjonell UNIX er designet for å gi god responstid for interaktive brukere i time-sharing omgivelse, samtidig som det hindrer utsulting av jobber med lavere prioritet. Det bruker en multilevel feedback med RR for hver prioriteringskø. Hvis prosess ikke fullføres eller blokkeres innen ett sekund blir den avbrutt. Prioritet er basert på prosesstype og utføringshistorikk.

Kapittel 10 – Multiprosessor, multikjerne og samtidstidsstyring

10.1 Multiprosessor og multikjerne tidsstyring

Når et datasystem inneholder mer en enkelt prosessor vil det introduseres nye designutfordringer. Multiprosessorsystem kan kategoriseres som følger:

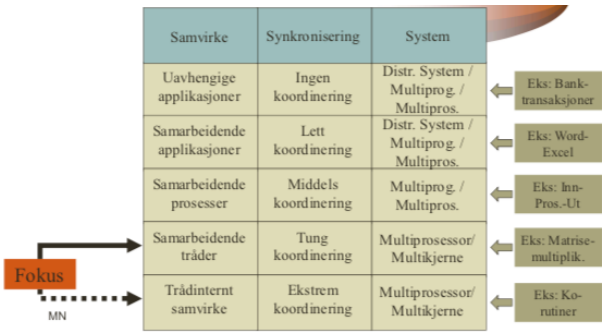
- **Løst koblede eller distribuerte multiprosessorer, eller *cluster*** – hver prosessor har eget hovedminne og egne IO-kanaler (kapittel 16)
- **Funksjonelt spesialiserte prosesser** – en master, generelt-formål prosessor kontrollerer spesialiserte prosessorer, eksempel IO-prosessor (kapittel 11)
- **Tett koblede prosessorer** – et sett med prosessorer som deler hovedminnet og er under integrert kontroll av OS (dette kapitlet)

Type system	Type effekt	Type kopling	Type referanse
Singleprosessor system	Multiprog. via programvare	Komplett	Kap. 3-4
Multiprosessor/-kjerne system	Multipros. med delt lager	Tett	Kap. 9-10
Distribuert system	Multipros. uten delt lager	Løs	Kap. 16
Spesialisert system	Klient / tjener via maskinvare	Tett / løs	Kap. 11-12

Multiprogrammert uniprosessor vs. multiprosessor tidsstyring

For å vurdere bruken av multiprogrammert uniprosessor eller multiprosessor kan man se på frekvensen av synkroniseringer mellom prosesser i et system (kalles granularitet).

Granularitet	Beskrivelse	Synkroniseringsintervall
Uavhengig parallellisme	Når det er flere urelaterte prosesser vil det være ingen eksplisitt synkronisering blant prosessene. Denne typen parallellisme brukes ofte innen time-sharing system, der hver bruker utfører en bestemt applikasjon (eks: regneark).	Ingen
Veldig grov parallellisme	Distribuert prosessering på tvers av nettverksnoder for å danne et enkelt datamiljø. Det er synkronisering mellom prosesser, men ved et mindre nivå. Brukes når det er sjelden interaksjon mellom prosessene.	2000-1M
Grov parallellisme	Multiprosessering av samtidige prosesser i et multiprogrammeringsnivå. Det er synkronisering mellom prosesser, men ved et mindre nivå. Brukes når samtidige prosesser trenger å kommunisere eller det er en fordel for synkroniseringen å bruke multiprosessor-arkitektur.	200-2000
Medium parallellisme	Parallell prosessering eller multitasking innenfor en enkelt applikasjon. En applikasjon kan implementeres som en samling av tråder innenfor en prosess, og programmereren må eksplisitt spesifisere potensiell parallellitet i applikasjonen. En relativt høy grad med koordinering og interaksjon vil foregå mellom trådene, slik at det krever medium synkroniseringsnivå.	20-200
Fin parallellisme	Parallellitet er innebygd i en enkelt instruksjonsstrøm. Dette gir en enda mer kompleks bruk av parallellitet enn ved tråder.	< 20



Uavhengig, grov og veldig grov parallellisme kan støttes av en multiprogrammert uniprosessor eller en multiprosessor uten at det har noen stor effekt på tidsstyringen. Det er ikke tilfellet for medium parallellisme med tråder, fordi trådene interagerer så ofte at tidsstyringsbeslutninger for en tråd kan påvirke hele applikasjonen. **Hvis det er prosesser som skal tidsstyres kan man altså bruke de samme algoritmene for multiprosessorer som for uniprosessorer, siden behovene til prosessene ikke vil variere så mye i slike sammenhenger. Hvis det er tråder som skal tidsstyres, vil man derimot ofte bruke andre algoritmer, siden tråder ofte har andre behov enn prosesser for slike tilfeller**. For eksempel kan man ønske å sikre at en enkelt tråd alltid kjører på samme prosessor pga. cache-hensyn. Man kan også ønske at alle tråder til en gitt prosess skal utføres samtidig på hver sin

prosessor pga. ytelseshensyn. Når det er flere prosessorer tilgjengelig kan man fokusere mer på slike hensyn enn på å holde hver enkelt prosessor aktiv til enhver tid.

Ved multiprosessor- og multikjernesystem vil fokuset som regel skiftes fra å maksimalt utnytte hver prosessor/kjerne, til å få til effektivt samarbeid mellom delprogrammer eller delapplikasjoner. Dette vil ofte innebære at tråder som hører til en gitt prosess ønsker å kjøre samtidig og dersom de avbrytes ønsker de å gjenstartes på en prosessor/kjerne som har rask tilgang til samme cache som siste. Dette er spesielt viktig for multikjerner med felles cache innen en felles chip (mindre viktig for multiprosessor).

Designutfordringer ved tidsstyring på multiprosessorer

Tidsstyring på en multiprosessor involverer tre utfordringer og løsningen på disse vil avhenge av granularitetsnivået og antall prosessorer som er tilgjengelig.

1. **Tildeling av prosesser til prosessorer** – den enkleste måten er å se på prosessorer som en samlet ressurs og dermed tildele prosesser til prosessorer etter behov. Da er spørsmålet om tildelingen skal skje **statisk eller dynamisk**. Ved statisk tildeling vil en prosess bli permanent tildelt en prosessor til den fullføres, og det opprettes kun en kortidskø for hver prosessor. Fordelen er at det kan gi lite overhead for tidsstyringen, men ulempen er at en prosessor kan gå på tomgang samtidig som en annen har full kø. Ved dynamisk tildeling vil det være en felles kø for alle prosessorer, slik at prosessen kan utføres på ulike prosessorer ved ulike tidspunkt (unngår problemet over). Ved tett-kobling med delt minne vil alle prosessorene ha tilgang til prosesskonteksten hos alle prosessene, slik at kostnaden til tidsstyringen blir uavhengig av identiteten til prosessoren. Det er vanlig å bruke både statisk og dynamisk tildeling med cache. For å tildele prosesser til prosessorer kan man bruke master/slave eller peer (s. 464).
2. **Bruk av multiprogrammering på individuelle prosessorer** – hvis en prosess er statisk tildelt en prosessor, må man avgjøre om denne prosessoren skal multiprogrammeres. For tradisjonelle multiprosessorer som håndterer synkronisering ved grov eller uavhengig parallellisme bør hver prosessor kunne skifte mellom prosesser for å oppnå høy utnyttelse og dermed bedre ytelse (= multiprogrammering). For applikasjoner med medium parallellisme som kjører på multiprosessorer, vil det bli mer komplisert. For svært parallelle system med flere tilgjengelige prosessorer, vil ikke prosessorutnyttelse være like viktig. **Det er viktigere å se på hva som gir god ytelse i gjennomsnitt for applikasjonene**. For eksempel kan en applikasjon ha dårlig ytelse med mindre alle trådene er tilgjengelig for å kjøre samtidig på ulike prosessorer.
3. **Dispatching av prosesser** – tidsstyringen hos multiprosessor må kunne velge en prosess som skal kjøres av prosessoren. For multiprogrammert uniprosessor kan bruk av prioritet og tidsstyringsalgoritmer basert på historikk øke ytelsen, mens for multiprosessor vil dette være for komplisert. **Multiprosessor tidsstyring bruker enklere metoder som kan være mer effektiv siden de har mindre overhead.** For svært parallelle systemer med flere prosessorer, vil ikke prosessorutnyttelse være like viktig

Tidsstyring av prosesser i multiprosessering

Forskning har vist at valg av tidsstyringsalgoritme blir mindre viktig når man har flere prosessorer enn kun én, og viktighetsgraden synker med økende prosessorer. Derfor kan en enkel FCFS-algoritme eller FCFS med prioritetsordning være tilstrekkelig for et multiprosessorsystem. Merk at dette er samme algoritme som brukes for multiprogrammert uniprosessor, bortsett fra at valget faller på en av de enklere algoritmene (enklere metode gir mindre overhead, se punkt 3 over). **Hvis det er prosesser som skal tidsstyres kan man altså bruke de samme algoritmene for multiprosessorer som for uniprosessorer**, siden behovene til prosessene ikke vil variere så mye i slike sammenhenger.

Tidsstyring av tråder i multiprosessering

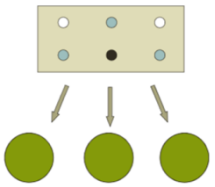
En applikasjon kan implementeres som et sett med tråder som samarbeider og utføres samtidig i samme adresserom. Tråder utføres separat fra resten av prosessdefinisjonen. I uniprosessorsystem blir tråder brukt for å hjelpe programstrukturen, overlappe IO med prosessering og oppnå mindre overhead med trådskifte istedenfor prosesskifte. **Den virkelige fordelen med tråder blir likevel først synlig ved multiprosessorsystem, der de kan brukes for å utnytte den reelle parallelliteten**. Dersom ulike tråder kan kjøre samtidig på separate prosessorer kan man få dramatisk økning i ytelse, men det krever riktig håndtering av trådene og tidsstyring. **Dette innebærer at man velger andre tidsstyringsalgoritmer enn de som brukes av uniprosessorsystem eller for grov eller uavhengig parallellisme.** Vi ser på fire metoder for tidsstyring av tråder og tildeling av prosessor:

1. **Load sharing**
2. **Gang scheduling**
3. **Dedicated processor assignment**
4. **Dynamic scheduling**

Det er særlig to ting vi ønsker å oppnå med tråder: (1) Sikre at en tråd alltid kjører på samme prosess for å utnytte bruken av delt cache (s. 89), og (2) Sørge for at tråder kan kjøre på hver sin prosessor for å sikre samkjøring og dermed god kommunikasjon mellom trådene. Flere prosessorer gjør at man kan fokusere på disse faktorene istedenfor å holde prosessoren aktiv til enhver tid.

Merk: denne delen vil fokusere på å få et overblikk over de ulike metodene. For detaljer, se i boka.

Lastdeling (*Load sharing*) (s. 467)



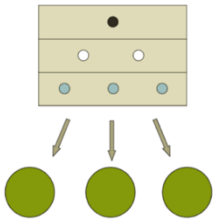
Lastdeling er kanskje den enkleste og mest brukte metoden, og den innebærer at hver prosesser selv velger neste prosess/tråd fra en felleskø basert på en gitt metode. Prosesser tildeles ikke en bestemt prosessor, for det dannes heller en felleskø med kjørbare tråder og hver prosessor henter en tråd når den er ledig. Fordeler ved lastdeling er:

- **Lasten fordeles likt utover prosessorene**, slik at ingen prosessor går på tomgang når det er arbeid som skal gjøres
- **Ingen behov for sentralisert tidsstyrer**, siden prosessoren selv velger neste tråd
- **Felleskøen kan organiseres og aksesseres vha. vanlige algoritmer (kapittel 9).** Trådene kan hentes basert på FCFS (tråder plasseres i kø og prosessor velger tråd som har vært i køen lengst; ofte best) eller prioritering basert på antall tråder (tråder som hører til jobber med få antall tråder blir prioritert, med eller uten avbrudd).

Ulemper ved lastdeling er:

- Den sentraliserte felleskøen vil oppta en region i minnet som må aksesseres med gjensidig utelukkelse, noe som kan bli en **flaskehals** hvis mange prosessorer søker etter arbeid samtidig.
- Det er lite sannsynlig at avbrutte tråder vil fortsette utførelsen på samme prosessor ved senere tidspunkt, noe som gir **dårlig utnyttelse av lokale cache hos prosessor**
- Det er lite sannsynlig at trådene hos en prosess vil få tilgang til prosessor samtidig, noe som gir **dårligere ytelse** dersom det trengs høy koordinering mellom trådene (se under).

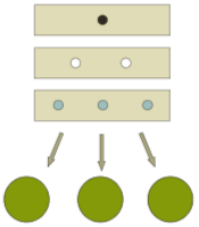
Samkjøring (*Gang scheduling*) (s. 469)



Samkjøring er en tidsstyringsalgoritme der flere/alle tråder innenfor en prosess blir tidsstyrt samtidig slik at de kan utføres parallelt på ulike prosessorer. Det er nyttig for medium og fin parallellisme der ytelsen vil reduseres mye dersom deler av applikasjonen ikke kjører samtidig som andre deler. Siden flere/alle tråden får hver sine prosessor, vil dette sikre **samkjøring av trådene**, slik at de er klare til å kommunisere med hverandre. **Samkjøring vil kunne øke ytelsen ved at det reduserer behovet for trådskifter og**

prosesskifter ved koordinering. Hvis en kjørende tråd når et punkt der den må synkronisere med en annen tråd i samme prosess og denne tråden ikke kjører, må tråden vente til det utføres et trådskifte på en annen prosessor som er tildelt prosessen eller et prosesskifte ved en annen prosessor som er tildelt en annen prosess. Slike skifter vil dramatisk redusere ytelsen. Samkjøring kan også spare tid ved ressursallokering, siden trådene kan aksessere en ressurs samtidig (unngår gjentatt aksess til samme ressurs, eks: fil). Ulempen er at det kan gi dårligere utnyttelse av prosessorene (dvs. mer tomgang).

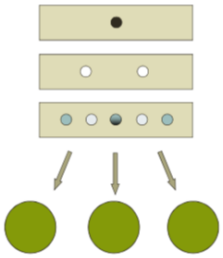
Prosessorholdning (*Dedicated processor assignment*) (s. 470)



Dedikert prosessortildeling er en ekstrem form for Gang scheduling, der en gruppe med prosessorer blir dedikert til en applikasjon i løpet av varigheten til applikasjonen. Hver av trådene i applikasjonen blir tildelt en prosessor som forblir dedikert til denne tråden helt til applikasjonen er fullført. Dette sikrer **samkjøring og ferdigkjøring av trådene** og gir **ingen trådskifter** ved koordinering, noe som sikrer maksimal ytelse og full hastighet for applikasjonen. Det vil likevel gi **dårlig utnyttelse av prosessorer**, siden de blir stående i tomgang for eksempel ved en IO-forespørsel (ingen multiprogrammering). For svært parallelle systemer med flere prosessorer, vil ikke prosessorutnyttelse være like viktig og ved å totalt unngå prosesskifte vil programmet kunne utføres mye raskere. Dette kan utnyttes ved å begrenser antall aktive tråder til antall prosessorer i systemet. Både Gang scheduling og dedikert prosessortildeling vil unngå trashing og prosessor fragmentering (noen prosessorer blir ikke tildelt noen prosess).

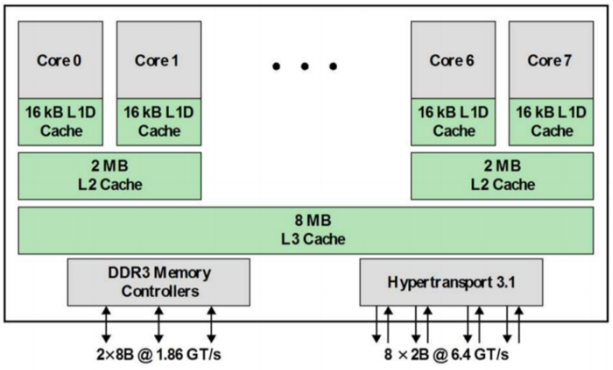
Antallsvariasjon (*Dynamic scheduling*) (s. 472)

Dynamisk tidsstyring innebærer at antall tråder i en prosess kan endres under utførelse, noe som lar operativsystemet tilpasse lasten for å forbedre utnyttelsen av prosessorene. Antall prosessorer som er tildelt en prosess kan variere dynamisk over tid på tilsvarende måte som antall tråder innenfor prosessen varierer. Applikasjonsprogrammet og operativsystemet kan og bør samhandle om tilordning av tråder til prosessorer (s. 472). **Antallsvariasjon er mest effektivt, men minst brukt.** Dersom applikasjonen kan utnytte seg av dynamisk tidsstyring vil den yte bedre enn Gang scheduling og dedikert prosessortildeling, men overhead kan negere ytelsesfordelen (trengs mer forskning). Antallsvariasjon gjør det mulig å tilpasse variasjoner i prosessorbehovet over tid, men det vil ha flere effekter (både positive og negative).



Tidsstyring ved multikjerne vs. multiprosessor

Mange multikjernesystem benytter samme algoritme for tidsstyring som multiprosessorsystem, som ofte er load sharing. Lastdeling fokuserer på å sikre utnyttelse av prosessorene, men ettersom avbrutte tråder ikke utføres på samme prosessor vil det ikke gi full utnyttelse av caching. F**or multiprosessorsystem er det viktig å sikre høy utnyttelse av prosessorkraften, mens i multikjernesystem er det viktig å minimere aksessen til lager utenfor chipen.** Denne aksessen blir minimert ved å bruke caches som utnytter lokalitetsprinsippet. Når en cache deles av noen, men ikke alle kjerner, vil måten tråder tildeles kjerner i løpet av tidsstyring ha en stor effekt på ytelsen. Figuren viser et tilfelle der kjerne 1 og 2 deler L2 cache. Tråder som deler minneressurser bør tildeles nabokjerner som deler cache for å forbedre effekten hos lokalitetsprinsippet. Tråder som ikke deler minneressurser bør tildeles ikke-nabokjerner for å fremme lastbalanse. **Målet er å tildele tråder til kjerner på en måte som maksimerer effektiviteten til delt cache og dermed minimerer behovet for aksess til lager utenfor chipen**.



10.2 Tidsstyring ved sanntid (*Real-time scheduling*)

Sanntidsberegninger kan defineres som beregning der korrektheten til systemet ikke kun avhenger av det logiske resultatet til beregningen, men også tiden resultatet produseres. Det blir stadig viktigere innenfor blant annet telekommunikasjon, trafikkontroll, robotikk, osv. I sanntidssystem vil noen av oppgavene være sanntidsoppgaver som forsøker å kontrollere eller reagere til hendelser som foregår i sanntid den virkelige verden. Sanntidsoppgaver er derfor som regel knyttet til en tidsfrist (*deadline*). **Harde sanntidsoppgaver** må møte fristen for å unngå at det forårsakes skade eller fatale feil i systemet, mens **myke sanntidsoppgaver** har en frist som det er ønsket å nå, men ikke obligatorisk (dvs. fortsatt nyttig å tidsstyre og fullføre oppgaver som har passert fristen). Vi skiller også mellom **aperiodiske oppgaver** som har frist på starttid og/eller fullføringstid (dvs. ingen mønster), og **periodiske oppgaver** som et bestemt mønster som er repeterbart (eks: «en gang per periode T»).

Egenskaper ved sanntid OS

Følgende er egenskaper ved sanntid OS (s. 475):

1. **Determinisme (forutsigbarhet)** – i hvilken grad OS utfører operasjoner i faste forhåndsbestemte tidspunkt eller tidsintervaller. For sanntid OS vil graden av determinisme avhenge av hvor raskt OS kan respondere til avbrudd og om systemet har tilstrekkelig kapasitet for å håndtere alle forespørsler fra eksterne hendelser innenfor krevd tid. Dvs: avbruddshåndtering.
2. **Responsivitet –** hvor lang tid OS bruker på å respondere på et avbrudd. Det vil sammen med determinisme avgjøre responstiden på eksterne hendelser, noe som er kritisk for sanntid system (nødvendig for å møte tidsfrister). Dvs: behovsbehandling.
3. **Brukerkontroll** – i sanntid OS er det essensielt at brukeren får finkornet kontroll over oppgaveprioritet, slik at bruker kan skille mellom harde og myke oppgaver og spesifisere relative prioriteter innenfor hver klasse. Dvs: prioritetsfastlegging.
4. **Pålitelighet** – i sanntid OS er pålitelighet svært viktig, siden det responderer til og kontrollerer hendelser i sanntid, slik at tapt ytelse kan ha katastrofale følger (eks: store finansielle tap og tap av liv). Dvs: tjenestegarantering
5. **Feiltilpasning** – systemets evne til å feile på en måte som bevarer så mange evner og data som mulig. Sanntid OS vil forsøke å korrigere feilen eller minimere effektene mens den fortsetter å kjøre. Målet er å sikre stabilitet ved at det fortsetter å møte fristene til de mest kritiske og høyest prioriterte oppgavene.

De fleste sanntid OS vil ha strengere bruk av prioritet med tidsstyring som benytter avbrudd, bundet og kort ventetid for avbrudd og mer presis og forutsigbar timing-egenskaper. **Hjertet i alle sanntidssystem er short-term tidsstyreren, som fokuserer på at alle harde sanntidsoppgaver og så mange som mulige myke sanntidsoppgaver møtes før fristen**. Ofte kan ikke OS operere direkte med frister, men designes i stedet slik at de er så responderende som mulig for sanntidsoppgaver (dvs. oppgaver blir raskt tidsstyrt når fristen nærmer seg). Dette kan oppnås ved å kombinere prioritet med klokkeavbrudd, der sanntidsoppgaver får høyere prioritet. For krevende sanntidsapplikasjoner kan man bruke umiddelbare avbrudd, slik at OS vil respondere med engang med mindre systemet er i en kritisk lockout seksjon.

Sanntid-tidsstyring

Tidsstyringsalgoritmene for sanntid kan deles inn i følgende klasser:

- **Statisk tabelldrevne (vurderingsbasert)** – en statistisk analyse av mulige tidsstyringer blir utført offline basert på et gitt sett med oppgaver og deres egenskaper. Resultatet er en tidsstyring som bestemmer ved kjøretid når en oppgave må begynne utføringen. Brukes for periodiske oppgaver. Den er forutsigbar, men lite fleksibel (tidsstyring må gjøres på nytt hvis oppgavesettet endres). **Forelesning**: bruker ofte periodebasert tidsstyring (RMS).

- **Statisk prioritetsdrevne med avbrudd (innsatsbasert)** – en statistisk analyse blir utført basert på gitte oppgaver og deres egenskaper. Resultatet er en tildeling av prioriteter til oppgavene, slik at en tradisjonell prioritetsdrevet tidsstyrer kan brukes. Brukes for periodiske oppgaver. Prioriteten er relatert til tidsbegrensningen hos hver oppgave. **Forelesning:** tidsfristen er ofte på avslutning og primært brukes tidligste tidsfrist først (EDF) og deretter høyest prioritet først (HPF).
- **Dynamisk planleggingsbaserte (vurderingsbasert)** – gjennomførbarhet blir bestemt dynamisk ved kjøretid istedenfor offline før utføringsstart (statisk). En ankommende oppgave blir akseptert for utføring kun hvis det er mulig å møte fristen til oppgaven og det ikke gjør at andre tidsstyrte oppgaver misser fristen. Resultatet er en tidsstyring eller plan som brukes for å bestemme når oppgaven skal utføres. **Forelesning**: bruker ofte slakkbasert tidsstyring (LLS)
- **Dynamisk innsatsbasert** – ingen mulighetsanalyse på forhånd, så systemet forsøker i stedet å møte alle tidsfrister og aborterer alle prosesser der fristen ikke møtes. Når oppgaven ankommer systemet vil det tildeles en prioritet basert på egenskaper ved oppgaven (ofte aperiodisk). Ulempen er at man ikke vet om tidsfrist nås før den ankommer eller oppgaven fullføres, mens fordelen er at det er lett å implementere. **Forelesning**: tidsfristen er ofte på oppstart og primært brukes tidligste tidsfrist først (EDF) og deretter FCFS.

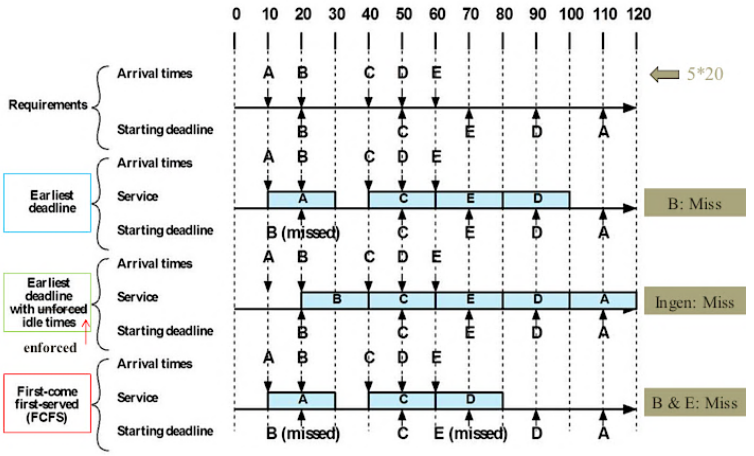
Vi skiller altså mellom **tidsstyringstyper** som kan være vurderingsbasert (mulighetsanalyse gir kjøreplan) eller innsatsbasert (ingen mulighetsanalyse og bruk av prioritet). Disse kan igjen deles inn i to ulike **tidsstyringsformer** som kan være statisk (fastlegges i forkant utfra tilstrekkelig tilgjengelig info) eller dynamisk (fastlegges underveis utfra utilstrekkelig tilgjengelig info). Harde sanntidsoppgaver håndteres ofte av vurderingsbasert algoritme, mens bløte sanntidsoppgaver håndteres med innsatsbaserte. Periodiske oppgaver håndteres ofte av statisk algoritme, mens aperiodiske håndteres ofte av dynamiske. Som regel vil vurderingsbaserte algoritmer være statiske, mens innsatsbaserte er dynamisk (merk: ofte, men ikke alltid).

Deadline scheduling (s. 480 + F)

Moderne sanntid OS er ofte designet med formålet om å starte sanntidsoppgaver så raskt som mulig og fremmer hurtig avbruddshåndtering og dispatching av oppgaver. Dette vil sikre hastigheten, men **sanntidsapplikasjoner er generelt mer opptatt av å fullføre oppgaver ved verdifulle tider (ikke for tidlig/sent)**. Oppgaveprioritet (dvs. hard/myk) vil ikke kunne fange denne informasjonen. Andre tilnærminger til sanntidstidsstyring innebærer at man har mer informasjon om hver oppgave, slik som startfrist, fullføringsfrist, prosesseringstid (mengde tid som trengs for å fullføre oppgave), prosessprioritet, ressurskrav eller deloppgave-struktur.

Ulike typer tidsstyringsalgoritmer for sanntidssystem er:

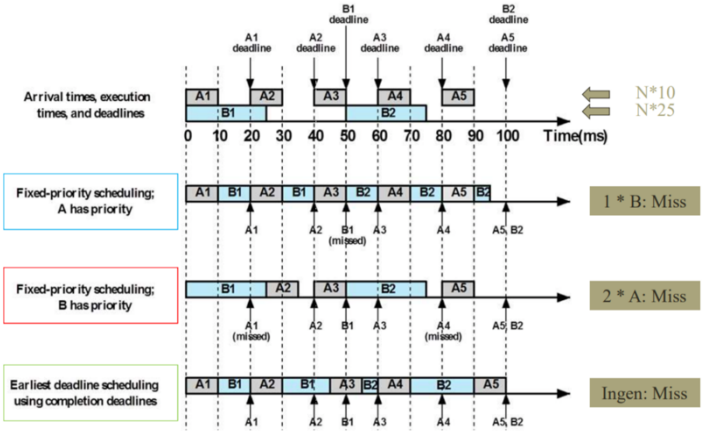
- **EDF (Earliest Deadline First)** – oppgaven med tidligst tidsfrist blir dispatched. Det kan vises at for en gitt avbruddsstrategi og start- eller fullføringsfrist, vil denne algoritmen minimere fraksjonen av oppgaver som misser fristen. Dette gjelder både uniprosessor og multiprosessor. EDF brukes hvis det er tilstrekkelig å gi gjennomsnittsgaranti, altså det er ikke nødvendig med absolutt garanti for hver enkelt prosess/tråd i et sett med ellers likeverdige prosesser/tråder. Det gir ikke garanti for hver enkelt prosess/tråd eller variansen mellom dem.
- **HPF (Highest Priority First)** – oppgaven med høyest prioritet blir dispatched. HPF brukes for en gruppe prosesser/tråder med klar prioritering dem imellom mht. harde og myke prioriteringer.



Figuren viser tidsstyring hos aperiodiske sanntidsoppgaver med startfrist for EDF, EDF med tvunget venting og FCFS. EDF med tvunget venting innebærer at oppgave med tidligst frist blir tidsstyrt selv om den ikke har ankommet (dvs. prosessor må vente på at den ankommer). FCFS gis for sammenligning.

- **RMS (Rate Monotonic Scheduling)** – periodebasert tidsstyring der høyest prioritet gis til minste periode (se under). RMS brukes når det trengs variansgarantier for periodisk tilfelle, altså når absolutte garantier for hver enkelt prosess/tråd i et sett med ellers likeverdige prosesser/tråder er nødvendig (hvis ulikhet holder)
- **LLS (Low Level Scheduling)** – slakkbasert tidsstyring der høyest prioritet gis til oppgaven med minst slakk. LLS brukes når det trengs garantier for aperiodisk tilfelle, siden den gir garanti for hver enkelt prosess/tråd (hvis ulikhet holder). Krever mye info kjent.

Tidsstyringsalgoritmene vil også ha ulik bruk av avbrudd. **Når startfrist er spesifisert, er det vanligst å bruke en ikke-avbrytbar tidsstyrer, mens når fullføringsfristen er spesifisert, er det vanligst å bruke en avbrytbar tidsstyrer**.



Figuren viser ulikt typer sanntidstidsstyring ved periodiske sanntidsoppgaver, fullføringsfrist og avbrytbar oppgaver. Her kan vi se at algoritmer basert på prioritering av A eller B vil gjøre at den andre oppgaven misser sine frister. Den nederste løsningen viser EDF tidsstyring der oppgaver med tidligst tidsfrist blir dispatched. Denne løsningen vil gjøre at ingen oppgaver misser sin fullføringstidsfrist. Legg merke til at avbrudd er tillatt, for når A2 ankommer vil B1 avbrytes siden A2 har tidligere tidsfrist. Siden oppgavene er periodiske og forutsigbare kan man bruke statisk tabelldreven tidsstyring.

Rate Monotonic Scheduling (RMS)

RMS er en av de bedre metodene for å løse tidsstyring hos periodiske oppgaver, og det innebærer å gi prioritet til oppgaver basert på deres perioder. For RMS vil oppgaver med kortest periode få høyest prioritet. Når flere oppgaver er tilgjengelig for utføring, vil den som har kortest periode betjenes først. Perioden T til en oppgave er mengde tid mellom ankomst av en oppgaveinstans og ankomst av neste oppgaveinstans. Utføringstiden C er mengde prosesseringstid som trengs for hver forekomst av oppgaven. Disse kan brukes for å lage et mål på effektiviteten til periodiske tidsstyringsalgoritmer. **For å garantere at alle tidsfrister skal møtes må summen av prosessorutnyttelsen ($U = C/T$) for hver oppgave være mindre eller lik 1 (total utnyttelse av prosessor):**

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_n}{T_n} \leq 1$$

Altså, hvis ulikheten er oppfylt vil den periodiske tidsstyringsalgoritmen sikre at alle rangerte prosesser vil kunne bli utført innenfor tidsrommet de må utføres innen. Hvis det er tilfellet vil summen av prosessorutnyttelsen for de individuelle oppgavene ikke overgå den totale prosessorutnyttelsen. For RMS kan det vises at følgende ulikhet gjelder:

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_n}{T_n} \leq n\left(2^{1/n} - 1\right)$$

Når antall oppgaver øker vil RMS konvergere mot $\ln 2 \approx 0.693$. Gitt T og C for flere oppgaver kan man regne ut om ulikheten er oppfylt, og dersom det er tilfellet vil bruk av RMS kunne garantere at alle oppgaver blir suksessfullt tidsstyrt. RMS vil ofte oppnå utnyttelse på 90%, og det er lettere å oppnå stabilitet. Informasjonen som må være kjent for å rangere prosesser er liten, men garantikravet er ganske tøft siden det forutsetter slakk på 30.7% = (1-0.693), som er når antall oppgaver går mot uendelig (lav grense).

Merk: det er typisk å bruke RMS ($\leq \ln 2$) og EDF (≤ 1), der RMS garanterer for varians (viktig for harde oppgaver) og EDF garanterer for snitt (nok for bløte oppgaver). Dermed blir harde og bløte krav blandet og man unngår at harde krav ofrer bløte krav.

Invertering av prioriteter (s. 486)

Invertering av prioritet oppstår ved alle prioritetsbaserte tidsstyringsformer, men det er spesifikt relevant for sanntidstidsstyring. Det oppstår når forholdene i et system tvinger en oppgave av høyere prioritet til å utføres etter en oppgave av lavere prioritet. Et enkelt eksempel er når en oppgave med lavere prioritet har låst en ressurs som en oppgave av høyrere prioritet trenger. Dermed må høyere-prioritet oppgave vente på lavere-prioritet oppgave. **Et mer kritisk problem er ubegrenset prioritetsinvertering, der varigheten til inverteringen avhenger av både tiden det tar å håndtere den delte ressursen og uforutsigbare handlinger til andre urelaterte oppgaver** (figurene under viser eksempel, s. 487). Det er to løsninger på dette problemet:

1. **Prioriteringsarv** – en lavere-prioritet oppgave arver prioriteten til en høyere-prioritet dersom den holder en ressurs som høyere-prioritet oppgave venter på. Prioriteten økes med en gang høyere-prioritet oppgave blokkeres på ressursen og senkes så fort lavere-prioritet oppgave frigir ressursen
2. **Prioritetstak** – hver ressurs har en prioritet som er ett nivå høyere enn prioriteten til høyest-prioriterte bruker. Tidsstyreren vil dynamisk tildele denne prioriteten til enhver oppgave som aksesserer ressursen. Når oppgaven er ferdig med ressursen vil den returnere til normal prioritet.

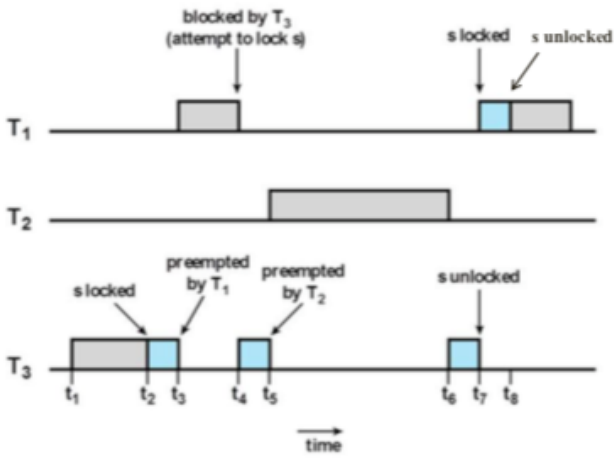


Figure 10.9 (a) Unbounded priority inversion

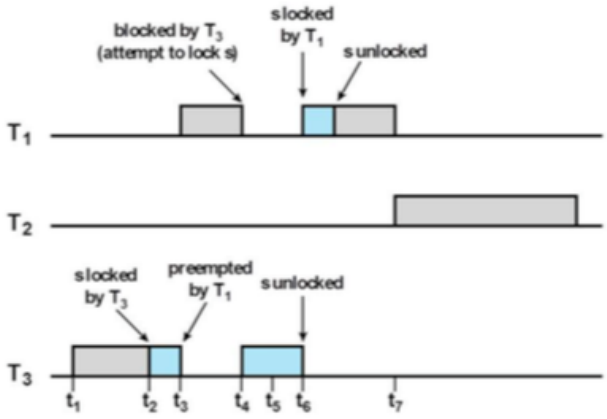


Figure 10.9 (b) Use of priority inheritance

På figur a vil T_1 ha høyere prioritet enn T_3, slik at når T_1 ankommer vil T_3 avbrytes. T_1 vil likevel blokkeres av T_3, siden denne har aksess til semaforen s. T_3 vil derfor gjenta utføring helt til den blokkeres av T_2 som har større prioritering. Her kan vi se at T_1 må vente på både T_3 og T_2, selv om det er kun T_3 som blokkerer aksessen pga. ressursallokering. På figur b blir dette løst ved å bruke prioriteringsarv. Når T_1 blir blokkert av T_3 vil T_3 bli tildelt samme prioritet. Dette gjør at når T_2 ankommer vil T_3 ha høyere prioritet og blir dermed ikke avbrutt. Dette gjør at semaforen låses opp tidligere, slik at T_1 får raskere tilgang.

10.3-10.6 Tidsstyring hos operativsystemene (s. 489-499)

Vi ser på et overblikk på tidsstyring for de ulike typene OS:

- **LINUX** (s. 489-492) – fokus på resultat og sanntid med 3 klasser og 140 nivå. Det bruker FIFO-, RR- og ikke-sanntid-tråder med aktive og utløpte prosesser/tråder. Både uniprosessor og multiprosessor benyttes (N prosessorer deles av alle M tråder)
- **UNIX** (s. 492-497) – fokus på resultat og sanntid med 3 klasser og 160 nivå. Det bruker en punktbasert avbrytning for toppklasse og statiske og variable prioriteter. Både uniprosessor og multiprosessor benyttes (N prosessorer deles av alle M tråder)
- **Windows** (s. 498-499) – fokus på resultat og sanntid med 2 klasser og 32 nivå. Det bruker en RR innen begge klasser og statiske og variable prioriteter. Både uniprosessor og multiprosessor benyttes (N prosessorer deles av alle M tråder)

Del 6 – Håndtering av I/O

Denne delen av kompendiet ser på håndtering av I/O og det inkluderer:

- **Kapittel 11** – Håndtering av I/O og disk scheduling
- **Kapittel 12** – Filhåndtering

Merk: denne delen er basert på kompendiet og LF til eksamen, men oppgir sidetall for at det skal være lettere å hente mer informasjon fra boka ☺

Kapittel 11 – Håndtering av I/O og disk scheduling

Input/output er kanskje den mest rotete delen av operativsystemer, siden det er vanskelig å utvikle generelle og konsistente løsninger fordi det finnes flere ulike enheter og applikasjoner for enhetene. **I/O-håndtering innebærer en konsistent håndtering av I/O-operasjoner som tilhører svært ulike brukerforespørsler, på svært ulike måter og mot svært ulik maskinvare**. **Dette må gjøres på en trygg og effektiv måte av programvaren som implementeres rett på maskinvaren, altså operativsystemet.** Dette krever mekanismer for tilgang til ulike overføringsverktøy for ulike behov og støtte for trygg og effektiv buffring, caching, operasjonsstyring og RAID-bruk.

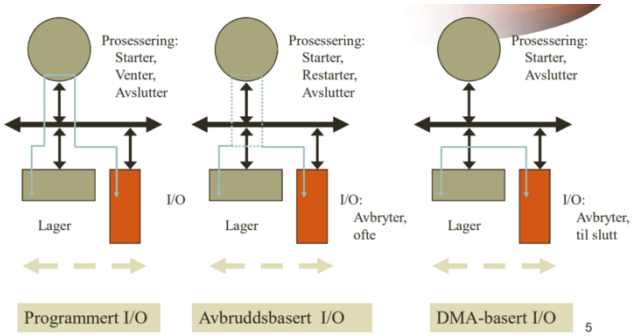
11.1 IO enheter (s. 506)

Eksterne enheter som brukes i IO kan grupperes i de tre kategoriene: **menneskelig-lesbare** (kommuniserer med menneskelig bruker, eks: mus, tastatur), **maskin-lesbare** (kommuniserer med elektronisk utstyr, eks: USB, sensorer) og **kommunikasjon** (kommuniserer med eksterne enheter, eks: modem). De tre klassene skilles fra hverandre mht. kapasitet, hastighet, representasjon og overføring. Tabellen viser andre faktorer som kan være variere mellom kategorier eller innenfor en kategori.

Datarate	Det kan være forskjeller i dataoverføringsrate (bps)
Applikasjon	Bruken av enheten har innflytelse på programvaren i OS, samt støtteverktøy
Kontrollkompleksitet	En printer krever et enkelt kontrollgrensesnitt, mens en disk er mer kompleks
Overføringsenhet	Data kan overføres som en strøm av bytes eller karakterer eller i større blokker
Datarepresentasjon	Ulike enheter representerer data ulikt
Feilbetingelser	Egenskapene til feil, måten de rapporteres, konsekvensen og respons varierer veldig

11.2 Organisering av IO-funksjonen (s. 508)

I/O kan utføres ved tre ulike teknikker:

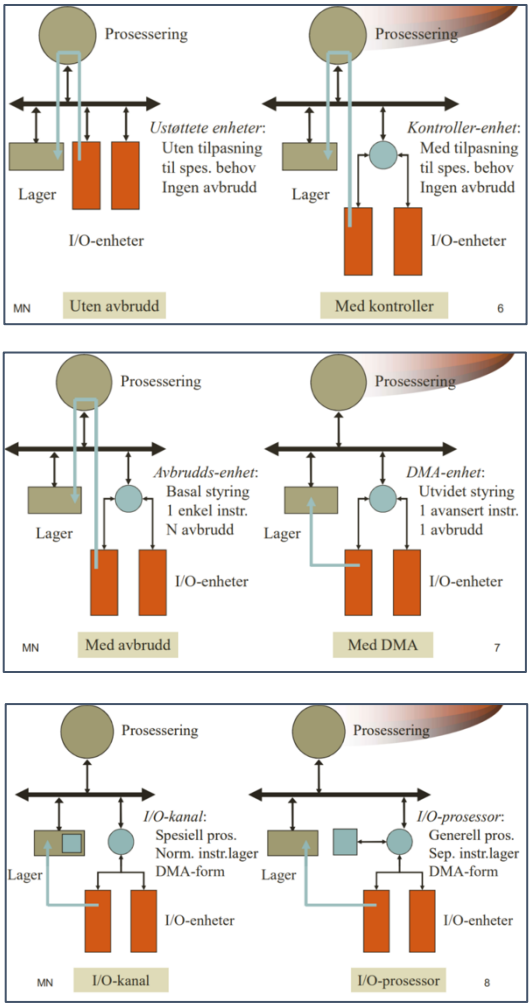


1. **Programmert I/O** – prosessoren sender en IO-kommando til en IO-modul på vegne av en prosess. Prosessen settes i aktiv venting og prosessoren går på tomgang til operasjonen er fullført og utføringen kan fortsette
2. **Avbruddsdrevet I/O** – prosessoren sender en IO-kommando på vegne av prosessen. Det gir to muligheter. Hvis IO-instruksjonen fra prosessen ikke er blokkerende vil prosessoren fortsette å kjøre prosessen. Hvis den derimot er blokkerende vil OS sette prosessen i blokkert tilstand og henter inn en ny prosess.
3. **Direkte minneaksess (DMA)** – en DMA-modul kontrollerer flytting av data mellom primærminnet og IO-modulen. Prosessoren sender etterspørsel for overføring av blokkdata til DMA-modulen og avbrytes kun når hele blokken er overført.

Evolusjonen av IO-funksjonen (s. 508)

Ettersom datasystemer har utviklet seg, har også I/O-funksjonen blitt utviklet. Denne utviklingen innebærer at mer og mer av I/O-funksjonen blir utført uten innblanding fra prosessoren, slik at I/O-modulen fungerer som en selvstendig enhet. Den sentrale prosessoren slipper flere I/O-relaterte oppgaver, noe som øker ytelsen. Noen viktige utviklinger er:

- **Kontroller** – kontroller-funksjonen legger til rette for tilpasning av spesielle behov. Introduksjon av kontroller gjør også at prosessor separeres fra spesifikke detaljer på eksterne grensesnitt, siden de eksterne I/O-enhetene ikke lenger er direkte styrt av prosessoren.
- **Avbrudd og DMA** – introduksjon av avbrudd førte til at prosessoren ikke lenger trengte å vente på gjennomføringen av I/O-operasjoner, noe som økte effektiviteten. Videre introduksjon av DMA gjorde at I/O-modulen kunne få direkte kontroll over minnet, slik at den kan flytte blokkdata til eller fra minnet uten å involvere prosessoren. Det er kun ved starten og slutten at prosessoren er involvert.
- **I/O-kanal og I/O-prosessor** – I/O-modulen kan forbedres ved å legge til en separat prosessor med spesialisert instruksjonssett som er skreddersydd for I/O (dette kalles I/O-kanal). Prosessoren kan da dirigere I/O-prosessoren til å utføre I/O-programmer i hovedminnet, der henting og utførelse av instruksjoner kan gjøres uten innblanding fra prosessoren. Dermed kan prosessoren spesifisere sekvenser med I/O-aktiviteter og kun avbrytes når hele sekvensen er fullført. I/O-modulen kan også få sitt eget lokale minne, slik at den kan kontrolleres med minimal prosessorinnblanding (dette kalles I/O-prosessorsteget)



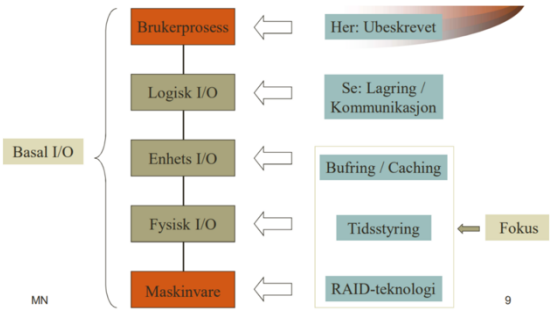
Direkte minneaksess (DMA) (s. 509)

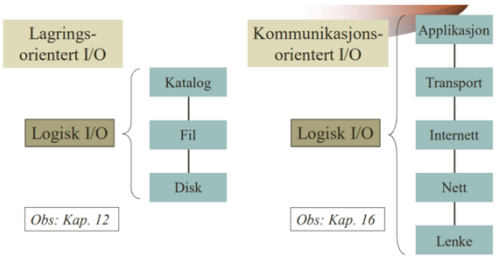
DMA-enheten kan etterligne prosessoren og ta over kontrollen på systembussen, slik som en prosessor. Dette er nødvendig for å kunne overføre data til og fra minnet over systembussen. Når prosessoren ønsker å lese eller skrive en blokk med data, vil den sende en forespørsel med informasjonen til DMA-modulen (s. 509 for mer detaljer).

11.3 Designutfordringer i et operativsystem (s. 511)

Ved design av I/O-fasiliteter er det i hovedsak to mål: effektivitet og generalitet. Effektivitet er viktig fordi prosessoren er mye mer effektiv enn I/O-modulen, slik at I/O-operasjoner kan fungere som flaskehalser. Dette kan motvirkes med multiprogrammering og swapping, men swapping er selv en I/O-operasjon. Det har derfor vært **mye fokus på forbedring av effektiviteten hos I/O, og spesielt innenfor disk I/O**. Generalitet innebærer ønsket om at alle enheter skal kunne håndteres på en uniform måte, for å fremme enkelhet og hindre feil. Diversiteten i egenskapene til enhetene gjør at det er vanskelig å oppnå sann generalitet. Dette kan bedres ved å bruke en **hierarkisk og modulær tilnærming til design av I/O-funksjoner. Dette vil skjule de fleste detaljene ved lavere nivå, slik at de øvre nivåene og brukernivåene ser ut som generelle funksjoner,** slik som read, write, open, lock og unlock.

IO-struktur (s. 512)





For å fremme generalitet vil altså I/O-strukturen bruke en hierarkisk og lagdelt tilnærming (se figur). Det enkleste hierarkiet bruker tre abstraksjonsnivå mellom brukerprosessen og maskinvaren:

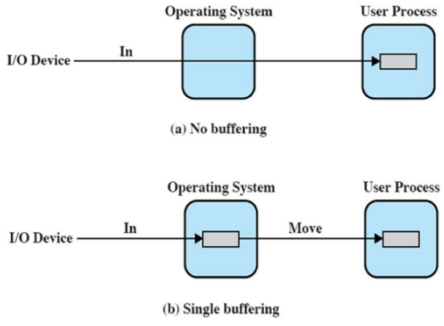
1. **Logisk I/O** – håndterer de logiske ressursene og er dermed ikke med på den fysiske kontrollen av enheten. Dette inkluderer håndtering av generelle I/O-funksjoner på vegne av brukerprosesser, slik som open, close, read og write. Figuren viser at logisk I/O kan bestå av ulike lag avhengig av om I/O er lagringsorientert eller kommunikasjonsorientert.
2. **Enhet I/O** – etterspurte operasjoner og data konverteres til passende sekvenser med I/O-instruksjoner, kommandoer og kontrollordre. Buffer/caching kan brukes for å bedre utnyttelsen.
3. **Fysisk I/O (tidsstyring og kontroll)** – den faktiske tidsstyringen av I/O operasjonene i tillegg til kontroll av operasjoner. Dette laget håndterer avbrudd og henter og rapporterer I/O-status. Det er programvare i dette laget som interagerer med I/O-modulen og dermed maskinvaren til enheten.

11.4 IO-buffering (s. 514)

En brukerprosess ønsker å lese datablokker fra disk en av gangen, ved at data leses inn i et dataområde innenfor adresserommet til brukerprosessen. Prosessen utfører I/O-kommandoen og venter på at dataen blir tilgjengelig (aktiv venting eller suspendert prosess). Problemet med denne tilnærmingen er at programmet må vente på at den relativt trege I/O-operasjonen fullføres og data kan tapes dersom delen av adresserommet som data leses inn i byttes ut (dvs. forstyrrer swapping, siden prosessen ikke kan byttes fullstendig ut). Det samme gjelder output-operasjoner der en datablokk overføres fra brukerprosessen til en I/O-modul. **For å unngå overhead og øke effektivitet kan det være nyttig å utføre input-overføringer før forespørsel gjøres og output-overføringer etter forespørsel gjøres, noe som kalles buffering.** OS kan støtte buffering for å øke ytelsen til systemet, og det brukes når man er fokusert på effektiv dataoverføring for mange ulike dataelementer samlet sett. **Vi skiller mellom to typer I/O-bufferen:**

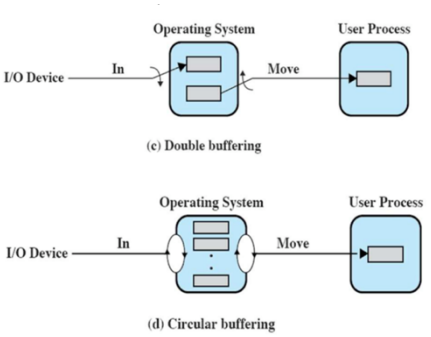
- **Blokkorientert enhet** – lagrer informasjon i form av blokker som vanligvis har fast størrelse og overføringer er en blokk av gangen (eks: disk og USB).
- **Strømorientert enhet** – overfører data inn og ut av enheten, uten blokkstruktur (eks: printer).

Singel buffer (s. 514)



Singel buffer unngår blokkering ved utførelse og swapping (dvs. prosess slipper å vente på treg I/O og kan byttes helt ut ved swapping). **Ved singel buffering vil input-overføringer gjøres til bufferen og når denne er fullført, vil prosessen flytte blokken inn i brukerrommet og sender umiddelbart forespørsel om en annen blokk. Dette kalles *reading ahead*, fordi det gjøres med forventning om at det til slutt vil være behov for blokken som leses.** Dette er rimelig, siden data ofte aksesseres sekvensielt. Resultatet er økt hastighet sammenlignet til system uten buffer, siden brukerprosessen kan prosessere en blokk med data samtidig som neste blokk leses inn i bufferen. OS kan bytte ut prosessen siden inputoperasjonen foregår i systemminnet istedenfor brukerprosess-minnet. Det vil samtidig komplisere logikken til OS, siden den må holde styr over tildelingen av systembuffere til brukerprosesser. Det vil også påvirke swapping (s. 515).

Dobbel buffer (s. 516)



Dobbel buffer er en forbedring av singel buffer, der to systembuffere blir tildelt operasjonen. **En prosess kan overføre data til (eller fra) en buffer, mens OS tømmer (eller fyller) den andre**. Dette vil utjevne mindre hastighetsvariasjoner (dvs. sikrer jevn dataflyt mellom I/O enhet og prosessen), men fører også til mer kompleksitet.

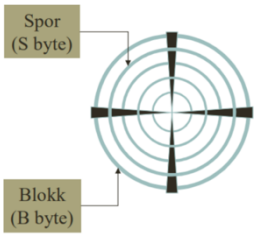
Sirkulær buffer (s. 516)

Dobbel buffering kan være utilstrekkelig dersom prosessen utfører hyppige kall til I/O, og i slike tilfeller kan det være behov for mer enn to buffere. Når det bruker mer enn to buffere vil samlingen av buffere kalles en sirkulær buffer. Dette er samme modell som bounded-buffer i producer/consumer modellen i kapittel 5 (s. 43). Bruk av sirkulær buffer vil utjevne større hastighetsvariasjoner, men fører også til mer kompleksitet.

Bruk av buffering (s. 517 + F)

Uten I/O-buffering kan det altså oppstå blokkering av enheter og prosessorer ved utførelse og swapping. Buffering er en teknikk som vil glatte ut topper i I/O-etterspørsel, men ingen mengde buffering vil gjøre at I/O-enheten kan holde tritt med prosessen på ubestemt tid når gjennomsnittlig behov hos prosessen er større enn det I/O-enheten kan tjene. Selv med flere buffere, vil alle til slutt fylles opp, slik at prosessen må vente. Ved kun I/O-oppgaver vil altså forsinkelsen etterhvert ta igjen tiden spart i bufferen, slik at prosessen uansett må vente. **Full utnyttelse av bufferen får man i systemer som varierer mellom I/O-oppgaver og andre prosessoppgaver. Bufferen vil da hjelpe til med å skjule variasjonen i hastigheten til de ulike oppgavene, og er dermed et verktøy som vil øke effektiviteten til OS og ytelsen til individuelle prosesser.**

11.5 Tidsstyring av disk (s. 517)



Gapet mellom hastigheten til prosessor og hovedminnet sammenlignet med diskaksess er veldig stort, noe som gjør at det er enda viktigere å ha en effektiv tidsordning av disken. Aksessformen for en disk innebærer å (1) søke frem til sporet, (2) rotere frem til blokken og (3) overføre hele blokken. Dette gir følgende aksesstid:

$$T_s + T_r + T_o = (D_o + D_1 N) + \frac{1}{2R} + \frac{B}{SR}$$

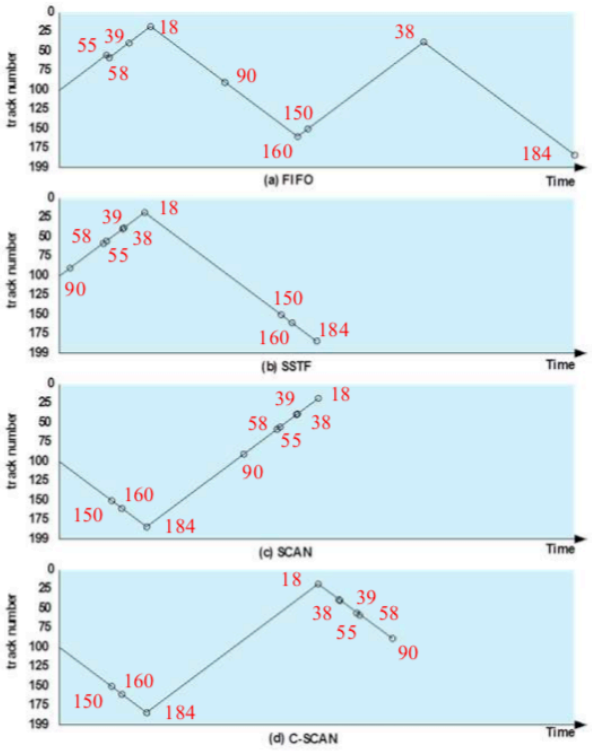
Aksesstiden vil altså inkludere søketid (T_s), rotasjonsforsinkelse (T_r) og overføringstid (T_o). Disk I/O-operasjoner vil også som regel involvere flere køforsinkelser, for eksempel når en prosess sender en I/O-forespørsel kan det hende den må vente i en kø til enheten er tilgjengelig. Se s. 518-519 for mer informasjon om disse tidene og et eksempel på utregning.

Tidsstyringsalgoritmer (s. 520)

En diskaksess vil altså som regel involvere flytting av lese-/skrivehode til riktig spor, etterfulgt av en rotasjon frem til riktig blokk og avsluttes med overføring av informasjon til eller fra tilhørende blokk(er). **Den mest tidkrevende komponenten er som regel den første, altså søking fra nåværende spor til utpekt spor.** I et multiprogrammeringsmiljø vil OS opprettholde en kø over forespørsler for hver I/O-enhet. **For en disk vil dette innebære en kø av read- og write-forespørsler fra ulike prosesser**. Ved **random tidsstyring** vil forespørslene fra køen hentes i tilfeldig rekkefølge, slik at sporene på disken sannsynligvis besøkes i tilfeldig rekkefølge og ytelsen blir så dårlig som mulig. Tabellen viser tidsstyringsalgoritmer for diskaksess, og disse kan sammenlignes opp mot random tidsstyring (*worst-case*).

Algoritme	Kommentar
FIFO	For rettferdighets fokus
SSTF	God ressursutnyttelse
LIFO	God lokalitetsutnyttelse
PRIO	For sanntids fokus
SCAN	Bedre tjenestesnitt
C-SCAN	Mindre tjenestevarians
N-SCAN	Faktisk tjenestegaranti
F-SCAN	Réelt lastavhengig

Figuren viser en sammenligning av ytelsen til ulike tidsstyringsalgoritmer for en sekvens av I/O-instruksjoner. Den vertikale aksen er spor på disken, mens den horisontale aksen er tiden eller antall spor som traverseres. **Jo flere spor som traverseres desto dårligere er ytelsen.**



(a) FIFO (starting at track 100)		(b) SSTF (starting at track 100)		(c) SCAN (starting at track 100, in the direction of increasing track number)		(d) C-SCAN (starting at track 100, in the direction of increasing track number)	
Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed
55	45	90	10	150	50	150	50
58	3	58	32	160	10	160	10
39	19	55	3	184	24	184	24
18	21	39	16	90	94	18	166
90	72	38	1	58	32	38	20
160	70	18	20	55	3	39	1
150	10	150	132	39	16	55	16
38	112	160	10	38	1	58	3
184	146	184	24	18	20	90	32
Average seek length	55.3	Average seek length	27.5	Average seek length	27.8	Average seek length	35.8

First-in-First-out (FIFO)

FIFO er den enkleste formen for tidsstyring, der enheter fra køen prosesseres i sekvensiell rekkefølge. Denne strategien er **rettferdig**, siden alle forespørsler håndteres og det gjøres i samme rekkefølge som de ankommer køen. Hvis det er få prosesser som krever tilgang og mange er samlet om filsektorer, kan FIFO gi god ytelse. Hvis det er mange prosesser som konkurrer om tilgang til disken, vil det derimot tilnærme random tidsstyring.

Prioritet (PRI)

PRI innebærer at forespørslene ordnes ut fra eksternt angitte prioriteter, slik at kontrollen for tidsstyring er utenfor diskhåndteringen. Denne tilnærmingen er ikke designet for å utnytte disken, men for å tilfredsstille andre krav innenfor OS. Som regel vil korte batch-jobber og interaktive jobber prioriteres over lengre jobber, slik at de kan gå raskt igjennom systemet og gi god interaktiv responstid. Det kan føre til at lengre jobber må vente overdrevent lenge. **PRI brukes for sanntidsfokus.**

Last-in-First-out (LIFO)

LIFO velger alltid den mest nylige forespørselen og fungerer godt for diskaksess, siden lokalitetsprinsippet kan gjøre at det blir mindre søking etter riktig spor. Problemet er at det kan føre til utsulting dersom mange forespørsler mottas. **LIFO gir god lokalitetsutnyttelse**.

Shortest-Service-Time-First (SSTF)

SSTF vil velge I/O-forespørselen som vil gi minst bevegelse av diskarmen fra nåværende posisjon, slik at forespørslene ordnes ut fra søkeavstand fra nåværende spor. Det er et forsøk på å minimere søketiden (T_s). Det vil ikke garantere at gjennomsnittlig søketid blir minimert, men det gir ofte bedre ytelse enn FIFO. **SSTF gir god ressursutnyttelse og lav gjennomsnittlig responstid** for diskaksesser, men også en høy varians på responstiden.

SCAN

Ulempen med algoritmene over er at en forespørsel kan gjentatt ikke oppfylles fordi det alltid ankommer nye forespørsler som velges før denne. SCAN-algoritmen fungerer som en heis og

unngår denne formen for utsulting. **SCAN innebærer at diskarmen kun beveger seg i én retning, helt til den har tilfredsstilt alle forespørsler i denne retningen eller når siste spor. For hvert spor vil alle lese/skrive-forespørsler som foreligger for tilhørende spor håndteres. Retningen blir deretter snudd og skanningen fortsetter i motsatt retning**. Figuren på forrige side viser eksempel der skanning starter i 100 og retningen er økende spornummer (150, 160, 184), før den snur og skanner i synkende spornummer fra 100 (90, 58, 55, osv.). **SCAN gir godt tjenestesnitt** og ligner SSTF, men utnytter ikke lokaliteten like bra. Ulempen er at det favoriserer forespørsler som er for spor nær endene (unngås ved C-SCAN).

C-SCAN (sirkulær SCAN)

C-SCAN begrenser skanningen til kun en retning. Etter en fullført skanning vil diskarmen returnere til den andre siden av disken for å gjennomføre skanningen på nytt. Dette vil redusere maksimal forsinkelse på nye forespørsler. **C-SCAN gir mindre tjenestevarians**, men også høyere gjennomsnittlig responstid på diskaksess. Figuren på forrige side viser eksempel der skanningen er i økende retning. Legg merke til at når den når enden (184) vil den flyttes til den andre siden av disken (18).

N-step SCAN og FSCAN

Ved SSTF, SCAN og C-SCAN er det mulig at armen kan stå stille en betraktelig mengde tid dersom noen prosesser har mye aksess til et spor (dvs. svært mange som skal gå av og på heisen ved én etasje). For å unngå en slik monopolisering kan man bruke N-step SCAN eller FSCAN. **N-step SCAN innebærer at køen for diskforespørsler segmenteres inn i delkøer med lengde N. Delkøene blir prosessert en av gangen vha SCAN og når en kø prosesseres vil nye forespørsler legges til i en annen kø** (dvs. heis som bare slipper til og håndterer ferdig et visst antall passasjerer, før neste lignende gruppe slippes til). Hvis færre enn N forespørsler er tilgjengelig ved slutten av skanningen, vil alle prosesseres ved neste SCAN. Denne har samme ytelse som SCAN for store verdier av N og vil samtidig gi **faktisk tjenestegaranti** (unngår utsulting). Det tilsvarer FIFO ved N = 1. **FSCAN bruker to delkøer** (merk: ingen størrelsesbegrensning N). Når skanningen begynner vil alle forespørslene være i en kø, mens den andre er tom. I løpet av skanningen vil alle nye forespørsler legges til den andre køen, slik at tjeneste av nye forespørsler blir utsatt til alle gamle forespørsler har blitt prosessert. **FSCAN er reelt lastavhengig.**

11.6 RAID (s. 524)

Dersom det er begrenset hvor langt en komponent kan nå, kan ytelsesgevinsten økes videre ved å bruke flere parallelle komponenter. Innenfor disklagring har dette ført til utviklingen av rekker med uavhengige disker som opererer parallelt. Flere disker gjør at separate I/O-forespørsler kan håndteres parallelt, så lenge den krevde dataen ligger på separate disker. **RAID (Redundant Array of Independent Disks) er en standard for ordning for multidisk-databasesystem som består av syv nivåer (fra 0 til 6)**. Disse nivåene er ikke hierarkisk ordnet, men utpeker ulike designarkitekturer som deler tre egenskaper:

1. RAID er et sett med uavhengige fysiske harddisker som oppfattes som en enkelt logisk harddisk av operativsystemet
2. Data er distribuert over de fysiske harddiskene på en måte kjent som striping
3. Overflødig diskkapasitet brukes til å lagre paritetsinformasjon, noe som garanterer gjenopprettelse av data dersom det oppstår diskfeil

Detaljene ved nivå 2 og 3 varierer ved de ulike RAID nivåene (eks: RAID 0 og 1 støtter ikke egenskap 3). **Bruk av RAID gjør at man kan øke hastigheten til dataoverføring ved å spre data over flere harddisker og bruke parallell lesing/skriving (dvs. gir økt I/O ytelse)**. Det kan også legge til rette for gjenoppretting av tapt data ved at samme informasjon lagres på flere harddisker (krever at deler av diskkapasiteten lagrer paritetsinformasjon). Det tillater også inkrementell økning i kapasitet, siden man kan legge til flere fysiske harddisker.

Category	Level	Description	Disks required	Data availability	Large I/O data transfer capacity	Small I/O request rate
Striping	0	Nonredundant	N	Lower than single disk	Very high	Very high for both read and write
Mirroring	1	Mirrored	$2N$	Higher than RAID 2, 3, 4, or 5; lower than RAID 6	Higher than single disk for read; similar to single disk for write	Up to twice that of a single disk for read; similar to single disk for write
Parallel access	2	Redundant via Hamming code	$N + m$	Much higher than single disk; comparable to RAID 3, 4, or 5	Highest of all listed alternatives	Approximately twice that of a single disk
	3	Bit-interleaved parity	$N + 1$	Much higher than single disk; comparable to RAID 2, 4, or 5	Highest of all listed alternatives	Approximately twice that of a single disk
Independent access	4	Block-interleaved parity	$N + 1$	Much higher than single disk; comparable to RAID 2, 3, or 5	Similar to RAID 0 for read; significantly lower than single disk for write	Similar to RAID 0 for read; significantly lower than single disk for write
	5	Block-interleaved distributed parity	$N + 1$	Much higher than single disk; comparable to RAID 2, 3, or 4	Similar to RAID 0 for read; lower than single disk for write	Similar to RAID 0 for read; generally lower than single disk for write
	6	Block-interleaved dual distributed parity	$N + 2$	Highest of all listed alternatives	Similar to RAID 0 for read; lower than RAID 5 for write	Similar to RAID 0 for read; significantly lower than RAID 5 for write

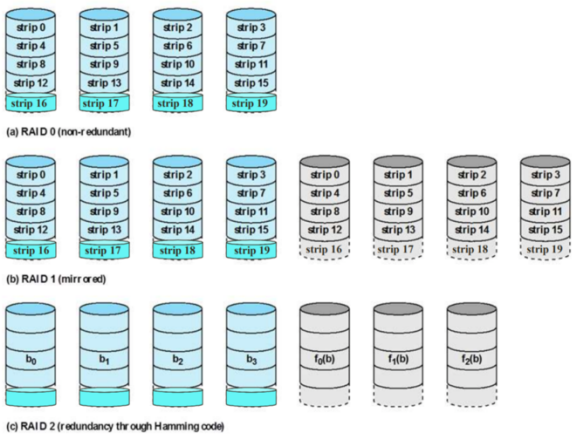
Tabellen viser de ulike nivåene, der fordelene er markert med grønt. Det er vanligst å bruke nivå 0, 1, 5 og 6. **Alle RAID nivåene benytter seg av striping for å øke parallell aksessering.** All bruker- og systemdata ses på som lagret på én logisk disk og denne disken er delt inn i flere strips. Disse stripsene blir kartlagt på en round robin måte på etterfølgende fysiske blokker i RAID-arrayen. På figuren kan vi se at strip 0 er i fysisk disk 1, strip 1 er i fysisk disk 2, osv. Et sett med logisk etterfølgende strips som kartlegger én strip til hver array kalles en stripe. Dersom det er n disker vil altså de n første logiske stripsene utgjøre første stripe (0, 1, 2 og 3 på figur), de neste n logiske stripsene utgjør andre stripe (4, 5, 6 og 7 på figur), osv. **Bruken av strips gjør at en enkelt I/O-forespørsel på logisk etterfølgende strips kan parallelt aksessere opptil n stripes, noe som gir en stor reduksjon i overføringstid.**

RAID Level 0 (s. 528)

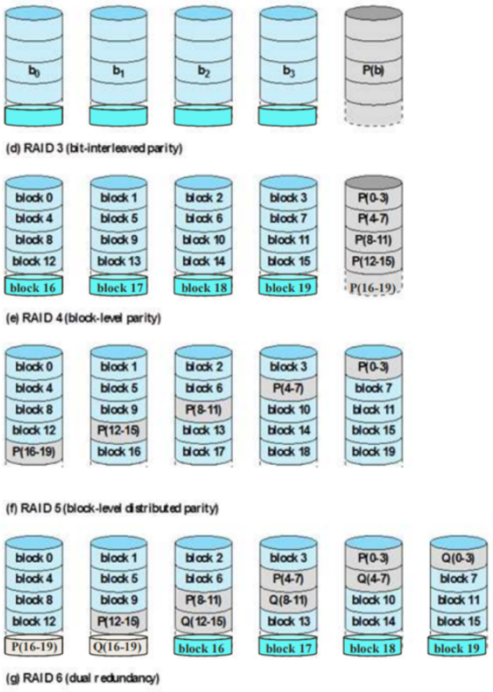
I RAID nivå 0 er bruker og systemdataen fordelt over de ulike diskene, uten noen form for duplisering (dvs. ingen redundans). Dette er ikke et ”ekte” RAID siden det har ingen kopiering, og man kan ikke gjenopprette data dersom en av harddiskene blir ødelagt. Dersom prosessene etterspør ulik data vil det fortsatt være en betraktelig økning i overføringshastighet (og dermed ytelse), siden dataen kan hentes parallelt hvis den befinner seg på ulike disker. Dataen er striped på tvers av tilgjengelige disker.

RAID Level 1 (s. 529)

I RAID nivå 1 blir redundans oppnådd via duplisering av all data. Dette kalles speiling (mirroring), siden hver logisk strips mappes til en separat fysisk disk, slik at hver disk i arrayen har en speildisk med samme data. Fordelen med speiling er at lese-forespørsler kan gjøres til begge diskene samtidig (den med minst tid velges). I tillegg må skrive-forespørsler gjøres til begge diskene, men dette kan gjøres parallelt (går fortere enn 2-6, siden det ikke er noen paritetsbit). Dersom det oppstår feil er det lett å ordne, siden dataen kan aksesseres fra den andre disken. Ulempen er at det gir mye overhead.



RAID 1 brukes i kritiske system som trenger backup av dataen eller i systemer som har mange leseforespørsler



RAID Level 2 (s. 530)

I RAID nivå 2 og 3 blir det brukt en parallell aksessteknikk, der alle diskene som er medlem av en parallell aksessarray vil delta i utføring av alle I/O-forespørsler. Som regel vil dette innebære at drivene er synkronisert, ved at diskhodet hos hver disk er i samme posisjon hele tiden. Dette er passende for å oppnå høy dataoverføringsrate, men ikke for å utføre separate I/O-forespørsler parallelt (da er nivå 4-6 bedre). Det brukes også stripes, men de er veldig små. **RAID 2 bruker en hamming-kode for å korrigere single-bit feil og detektere dobbel-bit feil.** RAID 2 krever også ganske stor plass, der antall overflødige disker er proporsjonal med logaritmen av antall disker. Ved read vil alle diskene aksesseres samtidig og leveres til array-kontroller som kan gjenkjenne og korrigere feil. Ved write må alle datadisker og paritetsdisker aksesseres. RAID 2 brukes kun i tilfeller der feil forekommer ofte.

RAID Level 3 (s. 531)

I RAID nivå 3 blir det brukt en parallell aksessteknikk (se over). Det brukes også stripes, men de er veldig små. **RAID 3 er organisert på tilsvarende måte som RAID 2, men bruker kun én redundansdisk og kun én paritetsbit istedenfor feil-korreksjonskode.** RAID 3 gir parallell aksess og siden stripsene er små kan det oppnå høy dataoverføringsrate. En I/O-forespørsel vil innebære parallell overføring av data fra alle datadiskene, så det gir stor overføringskapasitet for store overføringer. Ulempen er at kun én I/O-instruksjon kan utføres av gangen, så RAID 3 vil yte dårlig ved transaksjonsorientert omgivelse.

RAID Level 4 (s. 531)

I RAID nivå 4, 5 og 6 blir det brukt uavhengig aksess, der hver disk opererer uavhengig av hverandre. Dette gjør at separate I/O-forespørsler kan tilfredsstilles parallelt. Dette er passende for å utføre separate I/O-forespørsler parallelt, men ikke for å oppnå høy dataoverføringsrate (da er nivå 2-3 bedre). Det brukes også stripes, men de er relativt store. **RAID 4 benytter en bit-by-bit paritetsstripe som regnes ut for hver datadisk og lagres i en separat paritetsdisk (se figur).** For hver write må både dataen og paritetsbiten oppdageres, noe som introduserer en mulig flaskehals.

RAID Level 5 (s. 532)

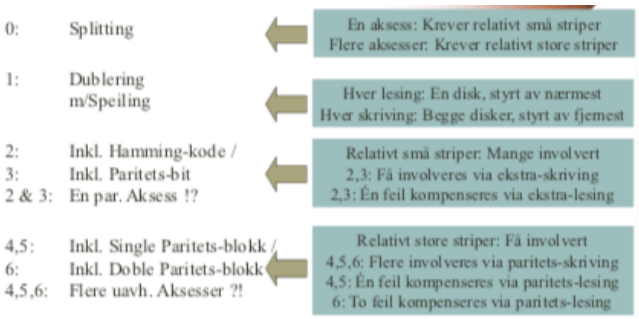
I RAID nivå 5 blir det brukt uavhengig aksess, der hver disk opererer uavhengig av hverandre (se over). **RAID 5 er organisert på tilsvarende måte som RAID 4, men forskjellen er at RAID 5 distribuerer paritetsbitene på alle diskene (som regel RR, se figur)**. Distribusjonen av paritetsstrips unngår en potensiell flaskehals som RAID 4 har og tap av en disk i RAID 5 vil ikke gi tapt data.

RAID Level 6 (s. 532)

I RAID nivå 6 blir det brukt uavhengig aksess, der hver disk opererer uavhengig av hverandre (se over). **RAID 6 bruker to ulike paritetsberegninger som lagres i separate blokker på ulike disker.** Dersom bruker krever N disker vil RAID 6 derfor ha N + 2 disker. P og Q er to ulike datasjekkalgoritmer, der en bruker OR (likhet med RAID 4 og 5) og den andre er en uavhengig datasjekkalgoritme. Dette gjør det mulig å regenerere data, selv om to disker har

brukerdatafeil. Fordelen med RAID 6 er at den gir svært høy datatilgjengelighet og tre disker må feile innen MTTR (mean time to repair) for at dataen skal gå tapt. RAID 6 har derimot høy straff for skriving, fordi hver skriving påvirker begge paritetsblokkene. Sammenlignet med RAID 5 kan RAID 6 ha en total reduksjon på 30% i skriveytelse. Leseytelsen er lik.

Nivå	Type	Overføringsrate for en	Tjenesterate for flere	Typisk applikasjon
0	Splitting	Små striper: ++	Store striper: ++	Ikkekritiske data
1	Dublering	R: +, W: -	R: +, W: -	Kritiske data
2	En parallell aksess	++	÷	---
3		++	÷	Høy overføringsrate
4	Flere uavhengige aksesser	R: -, W: ÷	R: ++, W: -	---
5		R: -, W: ÷	R: ++, W: -	Høy tjenesterate
6		R: -, W: ÷	R: ++, W: -	Høy tjenesterate & Ekstrem høy pålitelighet

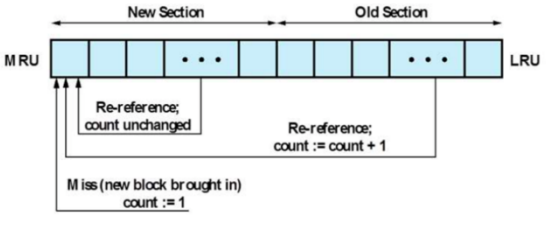


11.7 Disk cache (s. 533)

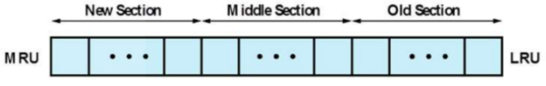
Cache refererer til minnet som er raskere og mindre enn primærminnet og ligger mellom primærminnet og prosessoren. Hensikten med cache er å redusere aksesstiden ved å bruke lokalitetsprinsippet. Samme prinsipp kan brukes på diskminnet. En disk cache er en buffer i hovedminnet for disksektorer, altså vil det inneholde kopier av sektorer på disken. Dersom det utføres en I/O-forespørsel om en bestemt sektor, vil det først undersøkes om sektoren er i disk cache. Hvis ikke vil sektoren leses inn i disk cache fra disken. Lokalitetsprinsippet gir at fremtidige forespørsel sannsynligvis vil referere til samme sektor.

Det finnes flere varianter disk cache avhengig av hvilken erstatningsalgoritme som brukes i tilfeller der cache er full slik at ny sektor må erstatte eksisterende. Ulike varianter er:

- **Least Recently Used (LRU)** – blokken som erstattes er den som det er lengst siden forrige referanse. Dette kan implementeres med en stakk av pekere, der blokkpeker flyttes oppover dersom blokken refereres til. Da vil blokken ved bunnen erstattes. Alternativt kan det implementeres ved at hver blokk har et tidsmerke.
- **Least Frequently Used (LFU)** – blokken som erstattes er den som det har blitt referert minst til. Dette kan implementeres ved at hver blokk har en referanseteller. Da vil blokken med minst teller erstattes. LFU kan være misledende i tilfeller der en blokk blir referert mye til i korte tidsintervaller, men brukes lite utenfor disse intervallene. Dette gjør at referansetelleren totalt sett ikke vil reflektere sannsynligheten for at blokken blir referert til igjen (løses med FBS).
- **Frekvensbasert stakk (FBS)** – det er en kombinasjon av LRU og LFU, der blokkene er organisert i en stack (likt LRU) og det brukes en referanseteller (likt LFU). Den øvre delen av stacken utgjør en ny seksjon som vil ha plass til et bestemt antall blokker. Når det er en cache-hit vil den refererte blokken flyttes til toppen av den nye seksjonen og referansetelleren vil kun økes med 1 dersom blokken ikke allerede var i denne seksjonen. **Dersom den nye seksjonen er tilstrekkelig stor vil dette gjøre at gjentatte referanser innenfor korte intervaller ikke regnes med i referansetelleren**. Stacken blir i tillegg delt inn i middel- og gammelseksjon. **Det er kun blokker som er i gammelseksjonen som kan erstattes og blant disse velges blokken med lavest referanseteller**. Dette gjør at nye blokker får mulighet til å bygge opp referansetelleren før de blir tilgjengelig for erstatning. **FBS er signifikant bedre enn vanlig LRU og LFU**.



(a) Use of two sections



(b) Use of three sections

Cache vs. buffer (E)

Buffering innebærer at data leses inn i bufferen på forhånd, noe som gir optimalisering av en aksess og utjevner hastighetsvariasjoner mellom I/O-enheter og prosess/minne. Cache innebærer at data som er nylig aksessert lagres for å tilfredsstille fremtidige referanser til samme data, noe som gir optimalisering av flere aksesser og utnytter lokalitetsprinsippet. Buffering brukes derfor når fokuset er på effektiv dataoverføring av mange ulike dataelementer, mens caching brukes når fokuset er på effektiv dataoverføring av mange ulike dataelementer som aksesseres flere ganger av samme eller ulike program. **Verken buffering eller caching bør brukes hvis fokuset er på hastighet hos isolert dataoverføring for et enkelt dataelement, siden elementet må lastes via buffer/cache før det gis til programmet.**

11.8-11.10 I/O håndtering ved operativsystemene (s. 537-546)

Vi ser på et overblikk på I/O håndtering for de ulike typene OS:

- **UNIX SVR4 (s. 537-540)** – hver I/O enhet er assosiert med en spesiell fil som håndteres av et filsystem og leses/skrives på samme måte som brukerdata-filer. Det er to typer I/O: buffered (buffer cache og karakterkø) og unbuffered (DMA).
- **LINUX (s. 540-544)** – ligner UNIX og bruker heis-tidsstyring og deadline tidsstyring for disken. Det bruker caching av disk-sider som er felles for virtuelt minne og fildata.
- **Windows (s. 544-546)** – I/O håndtering bruker cache manager, filsystem-driver, nettverk driver og maskinvare driver. I/O manager er ansvarlig for all I/O og gir uniformt grensesnitt for alle typer drivere. Det gir asynkron og synkron I/O og støtter programvare RAID.

E: Programvarebasert RAID tilsvarer en simulering av maskinvarebasert RAID, og det gir funksjonell tilgang til RAID-konseptet, men dårligere ytelse enn maskinvarebasert RAID (dvs. fysiske harddisker)

Kapittel 12 – Filhåndtering

Filhåndteringssystemet er et system med programvare som gir tjenester til brukere og applikasjoner som bruker filer, inkludert filaksess, kataloghåndtering og aksesskontroll. Dette systemet blir som regel sett på som en systemtjeneste som OS tjener istedenfor en del av selve OS. En liten del av filhåndteringsfunksjonen utføres likevel av OS.

12.1 Overblikk (s. 551)

Fra brukerens perspektiv vil filsystemet være en av de viktigste delene av et operativsystem. Et filsystem lar brukere opprette filer med følgende egenskaper:

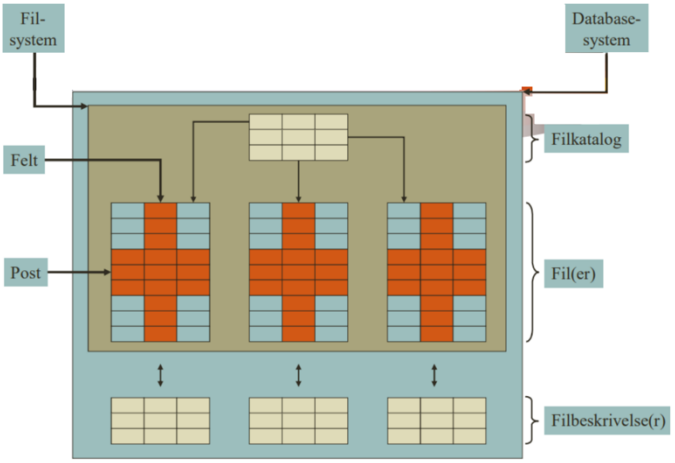
- **Langtidseksistens** – filer lagres i sekundærminnet og slettes ikke når brukeren logger av
- **Deling mellom prosesser** – filer har navn og kan ha tildelte aksesser som tillater deling
- **Struktur** – en fil kan ha en intern struktur som er tilpasset en applikasjon. I tillegg kan filer organiseres hierarkisk eller i en mer kompleks struktur for å representere forholdene mellom filer.

Filsystemet har også en rekke funksjoner som kan utføres på filer, for eksempel Create, Delete, Open, Close, Read og Write. Det vil også som regel opprettholde et sett med attributter assosiert med filen, slik som eier, aksessprivilegier og tidspunkt for opprettelse og sist endring.

Filstruktur (s. 552)

Fire begrep er ofte i bruk når man diskuterer filer:

1. **Felt** – et enkelt element med data (eks: etternavn hos ansatt)
2. **Post (*record*)** – en samling av relaterte felter (eks: navn, personnummer, osv. hos ansatt)
3. **Fil –** en samling av relaterte poster som behandles som en enkelt entitet av brukere og applikasjoner, kan refereres til med navn og kan opprettes og slettes. Aksesskontroll er ofte implementert på filnivå.
4. **Database** – en samling med relatert data.



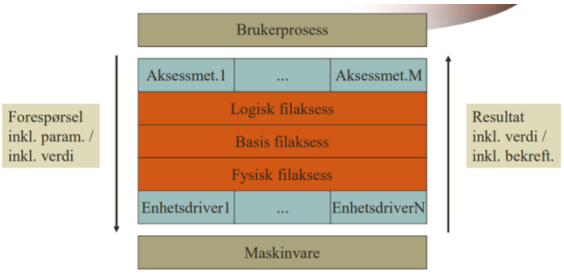
Filhåndteringssystem (s. 554)

Et filhåndteringssystem er et sett med system-programvare som gir tjenester til brukere og applikasjoner ved bruk av filer. Et filhåndteringssystem har i oppgave å:

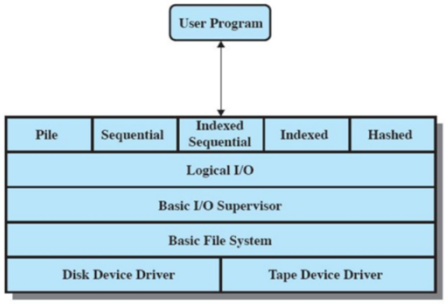
- Tilfredsstille brukerens behov for datahåndtering, slik som lagring og evnen til å utføre filoperasjoner som kontrollert aksess, flytting av innhold mellom filer, gjenopprette filer, identifisere filer via navn, mulighet til å opprette, slette, lese, skrive og endre filer, osv.
- Sørge for at dataen i filen er gyldig
- Optimere ytelse både via lav responstid (bruker) og høy gjennomstrømming (system)
- Gi I/O-støtte for ulike typer lagringsenheter
- Minimere og eliminere muligheten for tapt eller ødelagt data
- Gi et standardisert sett med rutiner for I/O grensesnitt til brukerprosesser
- Gi I/O-støtte i flerbrukersystem

Funksjonelle behov innebærer overordnete krav som å organisere i kataloger, skape/forandre/slette filer, strukturere innhold og kontrollere egen og andres aksess, og detaljerte krav som å kunne angi filer ved navn og innsette/endre/slette post. **Operasjonelle krav** innebærer organiseringskrav slik som lav responstid, høy gjennomstrømming, flere aksessmåter, tillatt parallellitet og mulig angre, mens aktivitetskrav innebærer rask aksess, rask oppdatering, effektiv og enkel lagring og trygge verdier.

Filsystemarkitektur (s. 554)

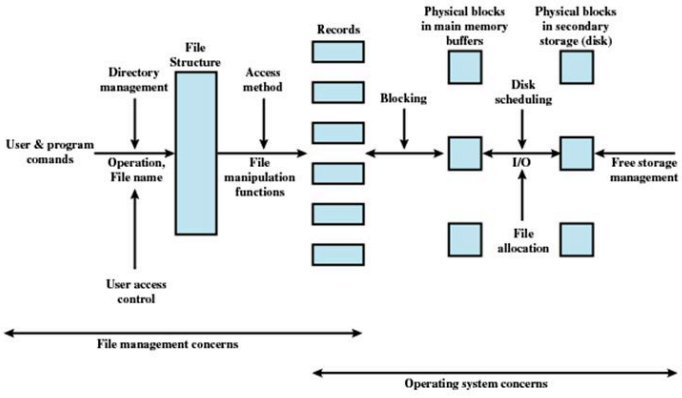


Figuren viser en mulig arkitektur for filsystemet som filsystemet som består av:



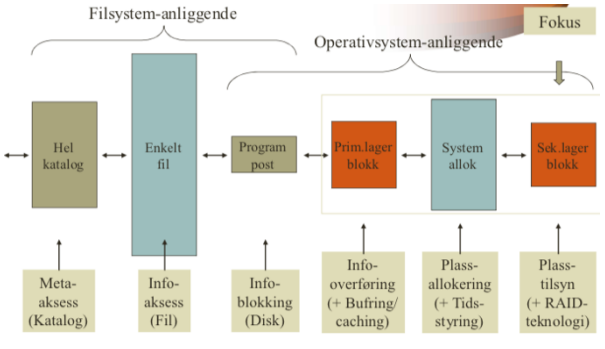
- **Enhetsdriver** – det nederste laget som kommuniserer direkte med eksterne enheter eller deres kontrollere eller kanaler. Enhetsdrivere er ansvarlige for å starte I/O-operasjoner på enheter og prosessere fullførelsen av en I/O-forespørsel. For filsystem kan dette være disk.
- **Fysisk filaksess (*basic file system*)** – hovedgrensesnittet til enheter utenfor datasystemet. Dette nivået håndterer datablokker som utveksles med enhetsdrivere (eks: plassering av blokkene i sekundær lagring og buffering av blokkene i hovedminnet).
- **Basis filaksess (*basic U/O supervisor*)** – har ansvar for all initiering og terminering av I/O-filer. Dette nivået består av kontrollstrukturer som håndterer I/O-enheter, tidsstyring av diskaksess for å optimalisere ytelse og filstatus. I/O buffere og sekundær minne blir tildelt ved dette nivået.
- **Logisk filaksess (*logical I/O*)** – lar brukere og applikasjoner aksessere poster og jobber dermed med hele filposter. Dette nivået gir generelle metoder for I/O-poster og opprettholder grunnleggende data om filene.
- **Aksessmetoder** – det øverste laget nærmest brukeren som gir standard grensesnitt mellom applikasjoner og filsystemet og enheter som holder dataen. Ulike aksessmetoder reflekterer ulike filstrukturer og forskjellige måter å aksessere og prosessere data (eks: sekvensiell, indeksert, hashed, osv.).

Enhetsdrivere, fysisk filaksess og basis filaksess ses på som deler av operativsystemet.



Filhåndtering funksjoner (s. 555)

Figuren under viser en annen måte å se på funksjonene til filsystemet. Brukere og applikasjoner vil interagere med filsystemet via kommandoer for å opprette og slette filer og utføre operasjoner på filer. Før filoperasjoner kan utføres må filsystemet identifisere og lokalisere den valgte filen. Dette krever bruk av en form for katalog som beskriver lokasjonen og attributter hos alle filer. De fleste delte systemene vil også bruke aksesskontroll, slik at kun autoriserte brukere får tilgang til bestemte filer. De grunnleggende operasjonene som utføres på filen vil utføres på post-nivå. Brukeren og applikasjonen vil se filen i form av en struktur som organiserer postene (eks: sekvensiell struktur). For å oversette brukerkommandoer til spesifikke filmanipulering-kommandoer blir det brukt en passende aksessmetode for filstrukturen. I/O gjøres på blokk-basis, så postene/feltene må organiseres som en sekvens av blokker, noe som krever flere funksjoner. Sekundærlagringen må håndteres, noe som innebærer å blant annet tildele filer til ledige blokker i sekundærlageret. Individuelle blokker med I/O-forespørsler må også tidsstyres. Både tidsstyring av disken og filallokering er opptatt av ytelsesoptimalisering, så disse vurderes sammen. Optimalisering vil også avhenge av strukturen til filene og aksessmønstret. **Utvikling av filhåndteringssystem mht. ytelse er altså svært komplisert!** Figuren viser en oppdeling av hva som kan anses som del av filsystemet og hva som er en del av operativsystemet. Som vi kan se vil OS utføre noen oppgaver som er en del av filhåndteringen.

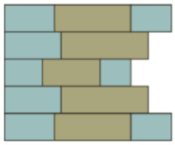


12.2 Filorganisering og aksess (s. 557)

Filorganisering er den logiske organiseringen av poster som bestemmes av måten de aksesseres. Viktige faktorer i valg av filorganisering er kort aksesstid, enkel oppdatering, økonomisk lagring (lite redundans), enkel vedlikehold og pålitelighet. Den relative prioriteringen av disse vil avhenge av applikasjonen som skal bruke filen. Faktorene kan konkurrere, for eksempel vil mye redundans være en måte å oppnå kortere aksesstid. Vi ser på fem typer filorganisering:

- Haugfil
- Sekvensiell fil
- Indeksert sekvensiell fil
- Indeksert fil
- Direkte eller hashet fil

Haugfil (ustrukturert) (s. 558)

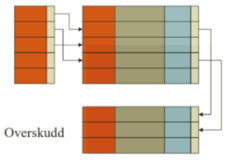


Data samles kronologisk i rekkefølgen de ankommer. Postene kan ha variabel lengde og variabelt antall felter. Hensikten er å akkumulere en rekke data og lagre det. Poster kan ha ulike felt, så feltene bør være selvbeskrivende med feltnavn og verdi. Siden det er ingen struktur vil søk være veldig ineffektivt (må søke gjennom alle poster). **Haugfil brukes derfor når det er vanskelig å organisere dataen (brukes sjeldent)**. Fordelen er at de er plasseffektive og lette å oppdatere.

Sekvensiell fil (s. 559)

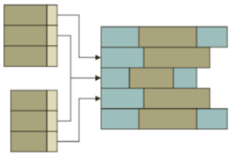
Sekvensielle filer er den vanligste formen for filer. Postene har et bestemt antall felter i en fast rekkefølge, og de er basert på et nøkkelfelt. Sekvensielle søk er lite effektivt, og ikke ideelt hvis man skal gjøre enkeltsøk på nøkkel eller andre kriterier. De brukes i batch-applikasjoner der **filen skal aksesseres som en helhet** (*exhaustive*, dvs. alle postene). De yter dårlig for interaktive applikasjoner som involverer queries eller oppdateringer av individuelle poster.

Indeksert sekvensiell fil (s. 559)



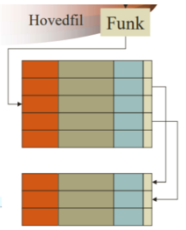
Indekserte sekvensielle filer overkommer en rekke av ulempene ved den sekvensielle filen. **Poster er fortsatt organiserte i sekvens basert på et nøkkelfelt, men indeks til filen støtter random aksess og overflow fil lagrer overflødige poster** (peker fra sekvensielle poster som ligger foran). Indeksen gir en mulighet til å raskt søke opp poster i nærheten av ønsket post. Den enkleste formen for søk innebærer at indeksen er en peker til hovedfilen og for å finne et spesifikt felt kan indeksen søkes for å finne høyeste verdi som er lik eller overgår ønsket verdi. Søket kan deretter fortsette i hovedfilen ved lokasjonen gitt av pekeren. Dersom man bruker færre nøkler enn poster i hovedfilen, vil søket bli mer effektivt. **Indeksert sekvensiell fil brukes når filen skal prosesseres som helhet (*exhaustive*), men det også er ønsket med random aksess til individuell post**.

Indeksert fil (s. 560)



Dersom man ønsker å søke basert på noe annet enn nøkkelfeltet, vil indekserte filer være uegnet. For å oppnå fleksibiliteten kan man strukturere på flere indekser, en for hver felt som det kan søkes etter. Poster blir aksessert via indeksene (dvs. ingen nøkler). Dette gjør at det er ingen begrensning på plasseringen av poster, så lenge en peker i minst en av indeksene refererer til en post. I tillegg kan postene være av variabel lengde. Indekserte filer brukes i applikasjoner der punktlighet hos informasjonen er kritisk.

Direkte eller hashet fil (s. 561)



En direkte eller hashet fil utnytter evnen disker har til å direkte aksessere en blokk med kjent adresse. Det krever derfor et nøkkelfelt, men sekvensiell ordning benyttes ikke. Disse brukes gjerne ved direkte aksess med poster av fast lengde.

File Method	Space Attributes: Variable	Space Attributes: Fixed	Update Record Size: Equal	Update Record Size: Greater	Retrieval: Single record	Retrieval: Subset	Retrieval: Exhaustive
Pile	A	B	A	E	E	D	B
Sequential	F	A	D	F	F	D	A
Indexed sequential	F	B	B	D	B	D	B
Indexed	B	C	C	C	A	B	D
Hashed	F	B	B	F	B	F	E

A = Excellent, well suited to this purpose ≈ $O(r)$
B = Good ≈ $O(o \times r)$
C = Adequate ≈ $O(r \log n)$
D = Requires some extra effort ≈ $O(n)$
E = Possible with extreme effort ≈ $O(r \times n)$
F = Not reasonable for this purpose ≈ $O(n^{r-1})$

where
r = size of the result
o = number of records that overflow
n = number of records in file

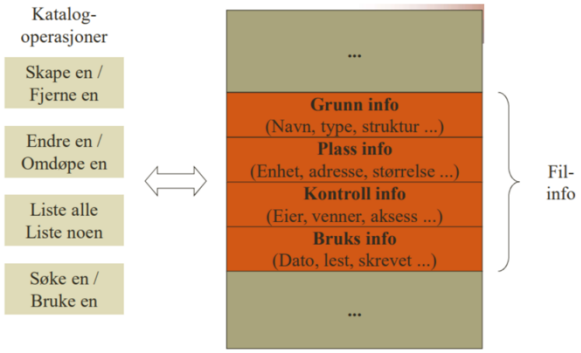
Figuren viser en sammenligning av de ulike filorganiseringene og deres bruksområder.

12.3 B-trær (s. 561)

For store filer eller databaser vil ikke en enkel sekvensiell indeksfil kunne gi hurtig aksess. I slike tilfeller kan man bruke B-trær for å oppnå en mer effektiv aksessering.

12.4 Filkataloger (s. 564)

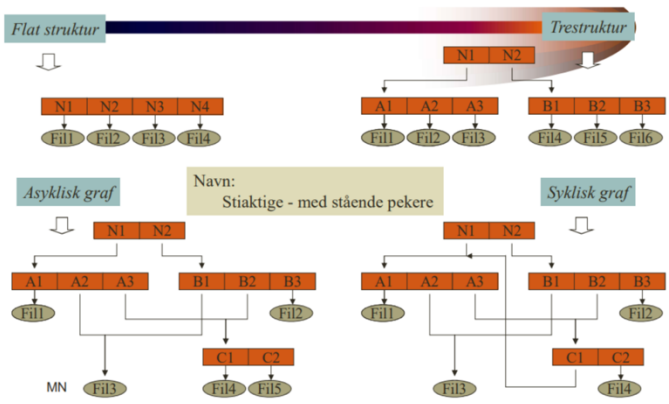
En filkatalog vil være assosiert med ethvert filhåndteringssystem, og det inneholder informasjon om filene, inkludert attributter, plassering og eierskap (se figur). Mye av denne informasjonen håndteres av operativsystemet. Katalogen er selv en fil som kan aksesseres av ulike filhåndteringsrutiner. Katalogtjenesten gjør at filene kan organiseres på en hierarkisk måte. Denne organiseringen er nyttig for det lar bruker holde styr over filer og det lar filhåndteringssystemet implementere aksesskontroll og andre tjenester.



Katalogstruktur (s. 566)

Når man bestemmer strukturen til katalogen må man ta hensyn til de ulike filoperasjonene som skal gjøres: søk, opprettelse, terminering, områdesøk og oppdateringer. Noen vanlige typer katalogstrukturer er:

- **Flat struktur (liste)** – den enkleste formen for katalogstruktur er en liste med filer. Dette kan representeres av en sekvensiell fil med filnavnet som nøkkel. Dette er derimot ikke en effektiv løsning dersom det er flere brukere eller mange filer. Det vil ikke la bruker organisere filer etter prosjekt, type, osv. og det krever unike filnavn (problematisk i flerbruker system). Aksesskontroll er også mer utfordrende.
- **2-nivå ordning** – bruker en katalog for hver bruker og en masterkatalog med adresse- og aksessinformasjon. Dette gjør at filnavn trenger kun å være unike innenfor én katalog (enklere for flerbruker system), men det gir fortsatt ingen form for struktur som lar bruker organisere filene etter bestemte egenskaper.
- **Trestruktur** – en mer fleksibel og mye brukt metode som benytter en masterkatalog med et antall brukerkataloger, der hver brukerkatalog kan ha flere underkataloger med filer. Hver underkatalog kan inneholde filer eller flere underkataloger, og disse kan være ordnet som en sekvensiell fil eller hashed struktur.

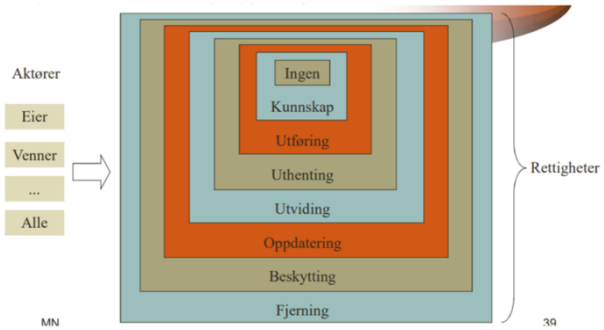


Filnavn (s. 567)

Brukere må kunne aksessere filer vha. et symbolsk navn, noe som krever at navnene er unike. Det vil være svært utfordrende dersom det kreves at filnavnet skal være unikt for alle filene i systemet, spesielt dersom det er flere brukere. Kataloger med trestruktur skiller mellom brukerne og legger til rette for at navnet kun trenger å være unikt innenfor én mappe, ved at stinavnet til mappen benyttes. Dette trenger ikke å spesifiseres av brukeren ved hver aksess, fordi det er registrert en *working directory* som brukeren jobber i.

12.5 Fildeling (s. 569)

Det er to utfordringer ved fildeling: aksessrettigheter og håndtering av samtidig aksess.



Aksessrettigheter (s. 569)

Filsystemet bør gi et fleksibelt verktøy for å tillate omfattende fildeling blant brukere, og det er viktig at det gir flere muligheter for å kontrollere hvilke filer som blir aksessert. Som regel vil brukere eller brukergrupper tildeles en bestemt aksessrettighet til filer, slik som None, Reading, Appending, Updating, Deletion, osv. Noen av disse er hierarkiske, for eksempel hvis en bruker har fått aksess til å oppdatere en fil, har brukeren også fått aksess for kunnskap om filen, utførelse og lesing. Vanligvis vil filer ha en eier som ofte er brukeren som har opprettet filen. Eieren kan også gi aksess til andre brukere, brukergrupper eller alle andre.

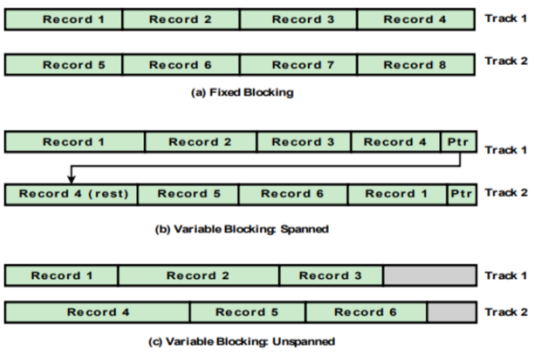
Samtidig aksess (s. 570)

Når flere brukere får tilgang til å oppdatere en fil, må OS eller filhåndteringssystemet kontrollere den samtidige aksessen. Dette kan oppnås på flere måter, for eksempel kan en bruker låse hele filen eller individuelle poster i løpet av oppdatering. Dette er essensielt readers-writers problemet fra kapittel 5.

12.6 Post-blokker (s. 570)

Poster er den logiske aksessenheten ved en strukturert fil, mens blokker er I/O-enheten ved sekundær lagring. **For at I/O skal kunne utføres, må poster derfor organiseres som blokker.** Dette introduserer nye beslutninger, slik som størrelsen til blokkene og om de skal ha fast eller variabel lengde. Gitt en blokkstørrelse, er det tre metoder som kan brukes:

1. **Fast blokkering** – poster har fast lengde og et heltall antall poster lagres i hver blokk. Dette kan føre til intern fragmentering. Mest brukt for sekvensielle filer som har poster med fast størrelse. Størrelsen til poster er begrenset av blokkstørrelsen.
2. **Variabel lengde blokkering, med spenn** – poster har variabel lengde og de lagres i blokker uten ubrukt plass. Noen poster vil spenne over flere blokker og fortsettelsen gis av peker til etterfølgende blokk. Gir effektiv lagring og begrenser ikke størrelsen til poster, men er vanskelig å implementere.
3. **Variabel lengde blokkering, uten spenn** – poster har variabel lengde, men det er ikke noe spenn, slik at hvis gjenværende plass i en blokk ikke er stor nok til å romme hele neste blokk vil denne plassen ikke brukes (dvs. står tom). Gir bortkastet bruk av plass og størrelsen til poster er begrenset av blokkstørrelsen.



Det er vanlig å bruke fast blokkering, siden dette passer best ved sekvensielt søk og forenkler I/O, buffer-allokering i primærminnet og organisering av blokker til disk (merk: ligner paging der rammer og sider har fast størrelse). Blokkstørrelsen er som regel stor sammenlignet med gjennomsnittlig poststørrelse. Dette gjør at flere poster kan plasseres i samme blokk, og disse sendes deretter til I/O-operasjonen.

12.7 Håndtering av sekundærminnet (s. 572)

En viktig funksjon ved filhåndteringssystemet er håndtering av disklagringen, som inkluderer strategien for å tildele diskblokker til en fil.

Plassallokering (s. 572)

Ved plassallokering må man ta hensyn til tre ting:

1. **Tidspunkt** (skal allokering skje i forkant eller ved behov)
2. **Enhetstype** (skal størrelsen på enheten som allokeres til en fil være fast eller variabel)
3. **Datastruktur** (hvilken struktur brukes for å holde styr over plassen som er tildelt filen)

Preallokering vs. dynamisk allokering (s. 572)

Ved preallokering må man vite hvor mye plass man har behov for på forhånd. I noen tilfeller er dette veldig vanskelig, og man ender ofte opp med å allokere for mye plass, noe som er veldig lite effektivt. Derfor kan det være en fordel å bruke dynamisk allokering, som tildeler rom til en fil etter behov.

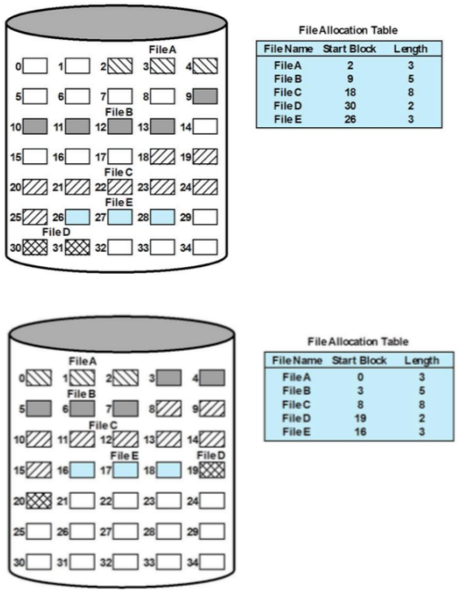
Fast vs. variabel plass (s. 573)

Ved plassallokering må man også avgjøre størrelsen til plassen som tildeles en fil. Den ene ekstreme siden innebærer at det tildeles en plass som er stor nok til å holde hele filen (dvs. flere blokker), mens den andre ekstreme siden er at diskplass tildeles en blokk av gangen. Når man velger en størrelse vil det være en balanse mellom effektivitet fra filens synspunkt og total effektivitet fra systemets synspunkt. Faktorer man må ta hensyn til er at (1) kontinuitet vil øke ytelsen, (2) et større antall små plasser øker størrelsen på tabellene som må holde allokeringsinformasjon, (3) faste størrelser forenkler reallokering av plass og (4) variabel størrelse gir mindre bortkastet bruk av plass. Det er to hovedalternativer:

1. **Variable, store kontinuerlige deler** – gir bedre ytelse, siden variabel størrelse unngår bortkastet bruk av plass og allokeringstabellene er små. Plass er vanskelig å gjenbruke. Kan bruke *first fit*, *best fit* eller *nearest fit* for plassallokering.
2. **Blokker** – små faste plasser gir større fleksibilitet. Det kan kreve større allokeringstabeller og komplekse strukturer for tildeling. Blokker blir tildelt etter behov.

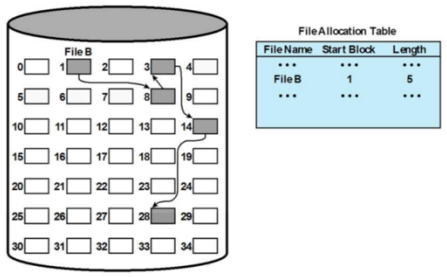
Plassallokeringsmetoder (s. 574)

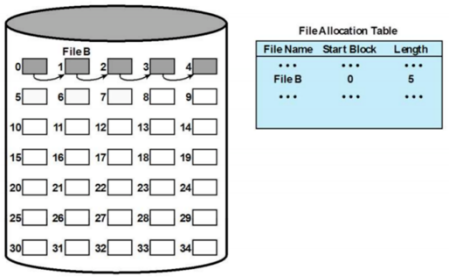
Det er i hovedsak tre metoder for plassallokering av disk:



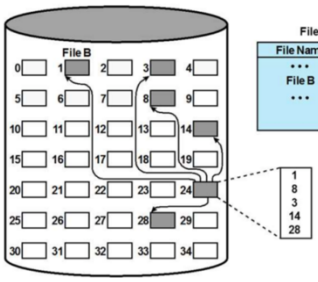
- **Kontinuerlig/sammenhengende allokering** – et enkelt sammenhengende sett med blokker blir tildelt filen ved filopprettelse (se figur). Dette er derfor en preallokeringsstrategi med variable størrelse. Denne metoden unngår mye overhead i allokeringstabellen fordi den kun trenger en plass per fil. I tillegg gjør den at aksessering av en blokk er enkelt, siden alt ligger etter hverandre på disken. Ulempen er at det kan gi ekstern fragmentering og at det ikke nødvendigviser plass til å legge en fil mellom andre filer, slik at det blir stående mye ledig plass. En annen ulempe er at man må vite størrelsen til filen på forhånd. En kompakt-algoritme kan brukes innimellom for å samle de ulike filene og redusere ekstern fragmentering (nederst figur). Sammenhengende plassallokering brukes i statiske situasjoner med små endringer av filinnhold underveis.
- **Kjedet allokering** – er på den andre siden av skalaen fra kontinuerlig allokering, og det innebærer at allokering gjøres på individuelt blokknivå. Hver blokk vil inneholde en peker til neste blokk i kjeden. Allokeringstabellen trenger en enhet per fil som gir

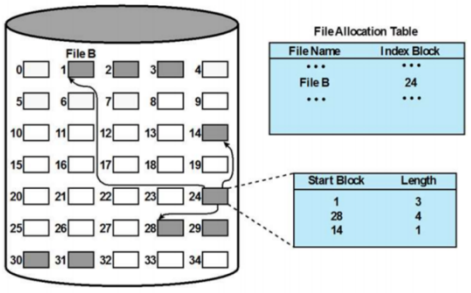
startblokken og lengden til filen. Preallokering er mulig, men det er vanligere med dynamisk allokering der blokker tildeles etter behov. Det er lett å legge til en blokk siden enhver ledig blokk kan legges til kjeden vha. en peker. En fordel er at det ikke gir ekstern fragmentering, slik at plassen utnyttes godt. Søk etter individuell blokk er ineffektivt, siden hele kjeden må gjennomsøkes (passer for sekvensielle filer, der hele filen prosesseres). Det er heller ingen utnyttelse av lokalitetsprinsippet, så hvis flere blokker må hentes kan det kreve aksess til ulike deler av disken. Dette kan løses ved å bruke konsolidering-algoritme som samler filene (nederst figur). Kjedet allokering brukes i dynamiske situasjoner med store endringer av filinnhold underveis, når man har råd til stor overhead mht. tidsforbruk ved gjennomsøkning av lenker som ofte er lange.





- **Indeksert allokering** – løser mange av problemene ved kontinuerlig og kjedet allokering. Allokeringstabellen inneholder en separat indeks for hver fil som gir blokkene som er allokert til filen. Disse filindeksene lagres som regel i separate blokker og hver fil i allokeringstabellen har en peker til sin filindeks-blokk. Allokeringene kan være både blokker med faste-størrelser (øverst figur) eller deler av variabel størrelse (nederst figur). Fast-størrelse eliminerer ekstern fragmentering, mens variabel-størrelse forbedrer lokalitet. Det støtter både sekvensiell og direkte aksess til filen og er dermed den mest populære formen for plassallokering. Indeksert allokering brukes i dynamiske situasjoner med store endringer av filinnhold underveis, når man har råd til overhead mht. oppretting av tabeller som ofte er store.





Aspekt	Sammen-hengende	Kjedet - fast	Kjedet - variabel	Indeksert - fast	Indeksert - variabel
Tidspunkt	I forkant	I forkant / Ved behov		I forkant / Ved behov	
Enhetstype	Variabel	Fast	Variabel	Fast	Variabel
Størrelse	Stor	Liten	Middels	Liten	Middels
Frekvens	Minimal	Høy	Middels	Høy	Middels
Plass	Minimal	Liten	Middels	Stor	Middels
Tid	Middels	Lang	Middels	Kort	Middels

Tid for allokering:
Fast: Lange lenker
Variabel: Korte lenker

Plass for allokering:
Fast: Små indeksinnslag
Variabel: Store indeksinnslag

MN 53

Figuren viser en sammenligning av de ulike typene diskallokering. Sammenhengende allokering kan gi lav aksesstid for en enkelt prosess, og det brukes heller når filer aksesseres sekvensielt. Kjedet allokering kan gi lav plass-overhead for alle prosesser samlet (lite ekstern fragmentering), og det brukes når filer skal aksesseres direkte. Indeksert allokering kan gi en balanse mellom aksesstid og plass-overhead og det brukes når filer skal aksesseres direkte eller sekvensielt.

Håndtering av ledig plass (s. 576)

For å kunne allokere plass til filer, må man vite hvilken plass som er ledig. For dette trenger vi en diskallokeringstabell i tillegg til filallokeringstabellen. Ulike teknikker som kan brukes er:

- **Bit-tabeller** – bruker en vektor med en bit for hver blokk på disken, der 0 representerer ledig og 1 representerer opptatt. Den gjør det relativt enkelt å finne en eller ledig blokk eller en sammenhengende gruppe med ledige blokker, og den fungerer godt for plassallokeringene vi har sett på over. Den er så liten som mulig, men vil fortsatt bruke en del plass i minnet. I tillegg vil sekvensielt søk være tregt.

- **Kjedet ledige plasser** – ledige plasser kan være kjedet sammen vha. en peker og en lengdeverdi for hver ledig plass. Dette krever neglisjerbar overhead, siden det ikke er behov for en diskallokeringstabell, men heller en peker til begynnelsen av kjeden og lengde til første del. Ulempen er at det kan gi fragmentering og hver gang man tildeler en blokk må man først lese blokken for å hente peker til den nye første ledige blokken, slik at pekeren kan oppdateres (tidkrevende dersom mange blokker skal tildeles)
- **Indeksering** – ledige plasser behandles som en fil og en indekstabell brukes på lignende måte som ved plassallokering. Denne løsningen brukes ved variabel-størrelse deler, siden det gir større effektivitet. Det vil gi en plass i tabellen for hver ledig del på disken.
- **Ledig-blokk liste** – hver blokk får et tall og listen av tallene til ledige blokker opprettholdes i en reservert del av disken.

12.8-12.11 Filhåndtering ved operativsystemene (s. 580-594)

Vi ser på et overblikk på filhåndtering for de ulike typene OS:

- **UNIX (s. 580-584)** – skiller mellom seks typer filer: normale datafiler, katalogfiler, spesielle filer, rørfiler, lenker og symbolske lenker. En inode (indeks node) brukes av OS og er en kontrollstruktur som inneholder nøkkelinformasjon om filer som OS trenger. Filallokering gjøres på blokk-basis og er dynamisk, indeksert og fast-størrelse. Katalogen er hierarkisk med ett masternivå (inode) og flere undernivå.
- **LINUX (s. 585-588)** – tilbyr et homogent virtuelt filsystem som avbildes på flere heterogene fysiske filstrukturer. Det er objektorientert, men skrevet i C. Grunnleggende komponenter er arvet fra UNIX SVR4 eller Solaris. Det bruker ulik caching for ulike katalog- og fil-komponenter.
- **Windows (s. 589-593)** – bruker et eget filsystem kalt NTFS som skal tilfredsstille krav fra arbeidsstasjoner og servere. Det fokuserer på gjenopprettbarhet, sikkerhet, størrelse, ytelse (indeks) og flere datastrømmer (fleksibilitet). Disklagringen er sektor, klynge eller volum. Allokeringen er klynget, variabel og dynamisk (dvs. ved behov).
- **Android (s. 594-595)** – grunnleggende funksjonalitet ligner LINUX. Det bruker en toppnivå katalogstruktur med system-katalog (OS-filer), data-katalog (brukerfiler), cache-katalog (mellomlagringsfiler) og enhet-katalog (koblebare filer). Databasen kan være alt fra isolert SQL til integrert SQLite.